

Learning Python

第4版
涵盖Python 2.6和3.x

Python

学习手册



O'REILLY®



机械工业出版社
China Machine Press



华章科技

Mark Lutz 著
李军 刘红伟 等译

第4版

Python学习手册

Mark Lutz 著

李军 刘红伟 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

Python学习手册：第4版/ (美) 鲁特兹 (Lutz, M.) 著；李军，刘红伟等译.—北京：机械工业出版社，2011.1

(O'Reilly精品图书系列)

书名原文：Learning Python, Fourth Edition

ISBN 978-7-111-32653-3

I. P… II. ①鲁… ②李… ③刘… 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 (2010) 第238848号

北京市版权局著作权合同登记

图字：01-2009-7718号

©2010 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2010. Authorized translation of the English edition, 2010 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2010。

简体中文版由机械工业出版社出版 2010。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。

封底无防伪标均为盗版

本书法律顾问

北京市展达律师事务所

书 名/ Python学习手册 (第4版)

书 号/ ISBN 978-7-111-32653-3

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 58.25印张

版 次/ 2011年4月第1版 2011年4月第1次印刷

定 价/ 119.00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

第4版

Python学习手册

O'Reilly Media, Inc.介绍

为了满足读者对网络 and 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's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到 GNN（最早的Internet门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

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

译者序

Python是一种简单的、解释型的、交互式的、可移植的、面向对象的超高级语言。Python作为一种功能强大且通用的编程语言而广受好评，它具有非常清晰的语法特点，适用于多种操作系统，目前在国际上非常流行，正在得到越来越多的应用。

Python有一个交互式的开发环境，因为Python是解释运行，这大大节省了每次编译的时间。Python语法简单，且内置了多种高级数据结构，如字典、列表等，所以使用起来特别简单，程序员很快就可学会并掌握它。Python具有大部分面向对象语言的特征，可完全进行面向对象编程。Python具有简单易用、可移植性强等特点，得到了众多程序员的青睐。它可以在MS-DOS、Windows、Windows NT、Linux等多种操作系统上运行。在最新的TIOBE开发语言排行中，Python名列第七。

本书是学习Python编程语言的入门书籍，目标是让读者快速掌握核心Python语言基础。本书设计成一本教程，主要关注核心Python语言本身，而不是其特定的应用程序。本书基于一个带有测试和练习的3天的Python培训课程，所以可以充当该语言的一个自学教程。本书至今已更新至第4版，每一版都得到广大读者的喜爱。本书内容详尽，从Python语言最基础和最核心的功能开始介绍，每章、每部分都配有丰富的习题，帮助读者巩固所学的知识。

本书篇幅很大，作者的介绍力求详尽而细致，有些地方难免显得冗长。加上新版的翻译工作量非常巨大，译者团队为此付出了艰辛的工作和努力，牺牲了很多的时间。但由于内容的广度和深度，难免有未尽之处，还请读者多多批评指正。参加本书翻译工作的有李军、刘金华、刘伟超、罗庚臣、刘二然、郑芳菲、庄逸川、王世高、郭莹、陈垠、邓勇、何进伟、贾晓斌、汪蔚、齐国涛、刘红伟、景龙、景文正、孙海军、李振胜、李秋强、楚亚军、景文生、王志刚、安宁宁、党耀云等。读者在阅读和学习过程中，如有问题可通过lijun961@sina.com与译者联系。

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

 Python如何获得支持 28

 Python有哪些技术上的优点 28

面向对象	28
免费	29
可移植	29
功能强大	30
可混合	31
简单易用	31
简单易学	32
Python和其他语言比较起来怎么样	32
本章小结	33
本章习题	33
习题解答	34
Python是工程，不是艺术	34
第2章 Python如何运行程序	36
Python解释器简介	36
程序执行	37
程序员的视角	37
Python的视角	39
执行模块的变体	41
Python实现的替代者	42
执行优化工具	43
冻结二进制文件	44
其他执行选项	45
未来的可能性	46
本章小结	46
本章习题	47
习题解答	47
第3章 如何运行程序	48
交互提示模式下编写代码	48
交互地运行代码	49
为什么使用交互提示模式	51

使用交互提示模式	52
系统命令行和文件	54
第一段脚本	55
使用命令行运行文件	56
使用命令行和文件	57
UNIX可执行脚本(#!)	58
UNIX env查找技巧	59
点击文件图标	60
在Windows中点击图标	60
input的技巧	61
图标点击的其他限制	63
模块导入和重载	63
模块的显要特性：属性	65
import和reload的使用注意事项	68
使用exec运行模块文件	69
IDLE用户界面	70
IDLE基础	71
使用IDLE	72
高级IDLE工具	74
其他的IDE	74
其他启动选项	76
嵌入式调用	76
冻结二进制的可执行性	77
文本编辑器启动的选择	77
其他的启动选择	77
未来的可能	77
我应该选用哪种	78
调试Python代码	78
本章小结	80
本章习题	80
习题解答	80
第一部分 练习题	81

第二部分 类型和运算

- 第4章 介绍Python对象类型 87
 - 为什么使用内置类型88
 - Python的核心数据类型88
 - 数字90
 - 字符串92
 - 序列的操作.....92
 - 不可变性94
 - 类型特定的方法94
 - 寻求帮助96
 - 编写字符串的其他方法97
 - 模式匹配98
 - 列表98
 - 序列操作98
 - 类型特定的操作99
 - 边界检查100
 - 嵌套.....100
 - 列表解析101
 - 字典103
 - 映射操作103
 - 重访嵌套104
 - 键的排序：for 循环.....105
 - 迭代和优化.....107
 - 不存在的键：if 测试107
 - 元组109
 - 为什么要用元组109
 - 文件110
 - 其他文件类工具111
 - 其他核心类型111
 - 如何破坏代码的灵活性113
 - 用户定义的类型114

剩余的内容..... 115

本章小结 115

本章习题 116

习题解答 116

第5章 数字 117

 Python的数字类型 117

 数字常量 118

 内置数学工具和扩展 119

 Python表达式操作符 120

 在实际应用中的数字 125

 变量和基本的表达式..... 125

 数字显示的格式 126

 比较：一般的和连续的 127

 str和repr显示格式 128

 除法： 传统除法、Floor除法和真除法 129

 整数精度 133

 复数..... 133

 十六进制、八进制和二进制记数 134

 位操作 136

 其他的内置数学工具..... 137

 其他数字类型 139

 小数数字 139

 分数类型 141

 集合 145

 布尔型 151

 数字扩展 152

 本章小结 153

 本章习题 153

 习题解答 153

第6章 动态类型简介	155
缺少类型声明语句的情况	155
变量、对象和引用	156
类型属于对象，而不是变量	157
对象的垃圾收集	158
共享引用	159
共享引用和在原处修改	161
共享引用和相等	163
动态类型随处可见	164
本章小结	165
本章习题	165
习题解答	165
第7章 字符串	167
字符串常量	169
单双引号字符串是一样的	170
用转义序列代表特殊字节	171
raw字符串抑制转义	173
三重引号编写多行字符串块	175
实际应用中的字符串	176
基本操作	176
索引和分片	177
为什么要在意：分片	181
字符串转换工具	181
修改字符串	184
字符串方法	185
字符串方法实例：修改字符串	187
字符串方法实例：文本解析	189
实际应用中的其他常见字符串方法	190
最初的字符串模块（在Python 3.0中删除）	191
字符串格式化表达式	192
更高级的字符串格式化表达式	194

基于字典的字符串格式化	196
字符串格式化调用方法	196
基础知识	197
添加键、属性和偏移量	198
添加具体格式化	198
与%格式化表达式比较.....	200
为什么用新的格式化方法	203
通常意义下的类型分类	206
同样分类的类型共享其操作集合	206
可变类型能够在原处修改	207
本章小结	208
本章习题	208
习题解答	208
第8章 列表与字典	210
列表	210
实际应用中的列表	213
基本列表操作	213
列表迭代和解析	213
索引、分片和矩阵	214
原处修改列表	215
字典	220
实际应用中的字典	222
字典的基本操作	222
原处修改字典	223
其他字典方法	224
语言表	225
字典用法注意事项	226
为什么要在意字典接口	229
创建字典的其他方法	230
Python 3.0中的字典变化	231
本章小结	237

本章习题	237
习题解答	237
第9章 元组、文件及其他.....	239
元组	239
实际应用中的元组	241
为什么有了列表还要元组	243
文件	243
打开文件	244
使用文件	245
实际应用中的文件	246
其他文件工具	252
重访类型分类	254
为什么要在意操作符重载	255
对象灵活性	255
引用 VS 拷贝	256
比较、相等性和真值	258
Python 3.0的字典比较	260
Python中真和假的含义	261
Python的类型层次	263
Type对象	263
Python中的其他类型	265
内置类型陷阱	265
赋值生成引用，而不是拷贝	265
重复能够增加层次深度	266
留意循环数据结构	266
不可变类型不可以在原处改变	267
本章小结	267
本章习题	268
习题解答	268
第二部分练习题	269

第三部分 语句和语法

第10章 Python语句简介..... 275

 重访Python程序结构275

 Python的语句276

 两个if的故事278

 Python增加了什么279

 Python删除了什么279

 为什么使用缩进语法281

 几个特殊实例283

 简短实例：交互循环285

 一个简单的交互式循环285

 对用户输入数据做数学运算287

 用测试输入数据来处理错误288

 用try语句处理错误289

 嵌套代码三层290

 本章小结290

 本章习题291

 习题解答291

第11章 赋值、表达式和打印 292

 赋值语句292

 赋值语句的形式293

 序列赋值294

 Python 3.0中的扩展序列解包297

 多目标赋值语句301

 增强赋值语句302

 变量命名规则305

 Python的废弃协议306

 表达式语句308

 表达式语句和在原处的修改309

 打印操作310

Python 3.0的print函数	311
Python 2.6 print语句	313
打印流重定向	315
版本独立的打印	318
为什么要注意print和stdout	319
本章小结	320
本章习题	321
习题解答	321

第12章 if测试和语法规则 322

if语句	322
通用格式	322
基本例子	323
多路分支	323
Python语法规则	325
代码块分隔符	326
语句的分隔符	328
一些特殊情况	329
真值测试	330
if/else三元表达式	332
为什么要在意布尔值	334
本章小结	335
本章习题	335
习题解答	335

第13章 while和for循环 336

while循环	336
一般格式	336
例子	337
break、continue、pass和循环else	338
一般循环格式	338
pass	338

continue	340
break	340
循环else	341
为什么要在意“模拟C 语言的while循环”	342
for循环	343
一般格式	343
例子	344
为什么要在意“文件扫描”	349
编写循环的技巧	350
循环计数器：while和range	351
非完备遍历：range和分片	352
修改列表：range	353
并行遍历：zip和map	354
产生偏移和元素：enumerate	357
本章小结	358
本章习题	358
习题解答	359
 第14章 迭代器和解析，第一部分	 360
迭代器：初探	360
文件迭代器	361
手动迭代：iter和next	363
其他内置类型迭代器	365
列表解析：初探	367
列表解析基础知识	368
在文件上使用列表解析	369
扩展的列表解析语法	370
其他迭代环境	371
Python 3.0中的新的可迭代对象	375
range迭代器	376
map、zip和filter迭代器	377
多个迭代器 VS 单个迭代器	378

字典视图迭代器	379
其他迭代器主题	381
本章小结	381
本章习题	381
习题解答	382
第15章 文档	383
Python文档资源	383
#注释	384
dir函数	384
文档字符串: __doc__	385
PyDoc: help函数	388
PyDoc: HTML报表	390
标准手册集	393
网络资源	394
已出版的书籍	394
常见编写代码的陷阱	395
本章小结	397
本章习题	397
习题解答	397
第三部分练习题	398
第四部分 函数	
第16章 函数基础	403
为何使用函数	404
编写函数	405
def语句	406
def语句是实时执行的	407
第一个例子: 定义和调用	408
定义	408
调用	408

Python中的多态	409
第二个例子：寻找序列的交集	410
定义	410
调用	411
重访多态	411
本地变量	412
本章小结	413
本章习题	413
习题解答	413
第17章 作用域	415
Python作用域基础	415
作用域法则	416
变量名解析：LEGB原则	418
作用域实例	419
内置作用域	420
在Python 2.6中违反通用性	422
global语句	422
最小化全局变量	423
最小化文件间的修改	424
其他访问全局变量的方法	426
作用域和嵌套函数	427
嵌套作用域的细节	427
嵌套作用域举例	427
nonlocal语句	433
nonlocal基础	433
nonlocal应用	435
为什么使用nonlocal	437
本章小结	440
本章习题	441
习题解答	442

第18章 参数	444
传递参数	444
参数和共享引用	445
避免可变参数的修改	447
对参数输出进行模拟	448
特定的参数匹配模型	449
基础知识	449
匹配语法	450
细节	452
关键字参数和默认参数的实例	452
任意参数的实例	455
Python 3.0 Keyword-Only参数	459
min调用	462
满分	463
加分点	464
结论	465
一个更有用的例子：通用set函数	465
模拟Python 3.0 print函数	466
使用Keyword-Only参数	467
为什么要在意：关键字参数	469
本章小结	469
本章习题	470
习题解答	470
 第19章 函数的高级话题	 472
函数设计概念	472
递归函数	474
用递归求和	474
编码替代方案	475
循环语句VS递归	476
处理任意结构	477
函数对象：属性和注解	478

间接函数调用	478
函数内省	479
函数属性	480
Python 3.0中的函数注解	481
匿名函数: lambda.....	483
lambda表达式	483
为什么使用lambda	484
如何（不要）让Python代码变得晦涩难懂	486
嵌套lambda和作用域.....	487
为什么要在意：回调.....	488
在序列中映射函数: map	489
函数式编程工具: filter和reduce	490
本章小结	492
本章习题	492
习题解答	492

第20章 迭代和解析，第二部分 494

回顾列表解析：函数式编程工具	494
列表解析与map	495
增加测试和嵌套循环	496
列表解析和矩阵	498
理解列表解析	499
为什么要在意：列表解析和map	500
重访迭代器：生成器	501
生成器函数：yield VS return	502
生成器表达式：迭代器遇到列表解析	506
生成器函数 VS 生成器表达式	507
生成器是单迭代器对象	508
用迭代工具模拟zip和map	510
为什么你会留意：单次迭代	514
内置类型和类中的值生成	515
Python 3.0解析语法概括	516

解析集合和字典解析	517
针对集合和字典的扩展的解析语法	517
对迭代的各种方法进行计时	518
对模块计时	519
计时脚本	519
计时结果	520
计时模块替代方案	523
其他建议	527
函数陷阱	528
本地变量是静态检测的	528
默认和可变对象	529
没有return语句的函数	531
嵌套作用域的循环变量	532
本章小结	532
本章习题	532
习题解答	533
第四部分练习题	533

第五部分 模块

第21章 模块：宏伟蓝图	539
为什么使用模块	540
Python程序架构	540
如何组织一个程序	541
导入和属性	541
标准库模块	543
import如何工作	543
1.搜索	544
2.编译（可选）	544
3.运行	545
模块搜索路径	545
配置搜索路径	547

搜索路径的变动	548
sys.path列表	548
模块文件选择	549
高级的模块选择概念	550
第三方工具: distutils	550
本章小结	551
本章习题	551
习题解答	551
第22章 模块代码编写基础	553
模块的创建	553
模块的使用	554
import语句	554
from语句	555
from *语句	555
导入只发生一次	555
import和from是赋值语句	556
文件间变量名的改变	557
import和from的对等性	557
from语句潜在的陷阱	558
模块命名空间	560
文件生成命名空间	560
属性名的点号运算	562
导入和作用域	562
命名空间的嵌套	563
重载模块	564
reload基础	565
reload实例	566
为什么要在意: 模块重载	567
本章小结	568
本章习题	568
习题解答	568

第23章 模块包..... 570

包导入基础.....570

包和搜索路径设置571

__init__.py包文件572

包导入实例.....573

包对应的from语句和import语句574

为什么要使用包导入.....575

三个系统的传说576

包相对导入.....578

Python 3.0中的变化.....578

相对导入基础知识579

为什么使用相对导入.....581

相对导入的作用域583

模块查找规则总结583

相对导入的应用584

为什么要在意：模块包589

本章小结590

本章习题590

习题解答590

第24章 高级模块话题..... 592

在模块中隐藏数据.....592

最小化from *的破坏：_X和__all__593

启用以后的语言特性593

混合用法模式：__name__和__main__.....594

以__name__进行单元测试.....595

使用带有__name__的命令行参数.....596

修改模块搜索路径.....599

Import语句和from语句的as扩展599

模块是对象：元程序.....600

用名称字符串导入模块603

过渡性模块重载604

xviii | 目录

Download at <http://www.pin5i.com/>

模块设计理念607

模块陷阱607

 顶层代码的语句次序的重要性.....608

 from复制变量名，而不是连接.....609

 from *会让变量语义模糊610

 reload不会影响from导入.....610

 reload、from以及交互模式测试.....611

 递归形式的from导入无法工作.....612

本章小结613

本章习题613

习题解答613

第五部分练习题614

第六部分 类和OOP

第25章 OOP：宏伟蓝图 619

 为何使用类.....620

 概览OOP.....621

 属性继承搜索621

 类和实例623

 类方法调用.....624

 编写类树624

 OOP是为了代码重用.....627

本章小结629

本章习题629

习题解答630

第26章 类代码编写基础 631

 类产生多个实例对象631

 类对象提供默认行为.....632

 实例对象是具体的元素632

 第一个例子.....632

类通过继承进行定制635

 第二个例子635

 类是模块内的属性637

类可以截获Python运算符638

 第三个例子639

 为什么要使用运算符重载641

世界上最简单的Python类641

 类与字典的关系644

本章小结646

本章习题646

习题解答646

第27章 更多实例 649

步骤1：创建实例650

 编写构造函数650

 在进行中测试651

 以两种方式使用代码652

 版本差异提示654

步骤2：添加行为方法654

 编写方法656

步骤3：运算符重载658

 提供打印显示658

步骤4：通过子类定制行为659

 编写子类660

 扩展方法：不好的方式660

 扩展方法：好的方式661

 多态的作用663

 继承、定制和扩展664

 OOP：大思路664

步骤5：定制构造函数665

 OOP比我们认为的要简单666

 组合类的其他方式667

在Python 3.0中捕获内置属性	669
步骤6：使用内省工具	670
特殊类属性	670
一种通用显示工具	671
实例与类属性的关系	672
工具类的命名考虑	673
类的最终形式	674
步骤7（最后一步）：把对象存储到数据库中	676
Pickle和Shelve	676
在shelve数据库中存储对象	677
交互地探索shelve	678
更新shelve中的对象	680
未来方向	681
本章小结	683
本章习题	684
习题解答	684
 第28章 类代码编写细节	 686
class语句	686
一般形式	686
例子	687
方法	689
例子	690
调用超类构造函数	691
其他方法调用的可能性	691
继承	692
属性树的构造	692
继承方法的专有化	693
类接口技术	694
抽象超类	695
Python 2.6和Python 3.0的抽象超类	696
命名空间：完整的内容	698

简单变量名：如果赋值就不是全局变量	698
属性名称：对象命名空间	698
Python命名空间的“禅”：赋值将变量名分类	699
命名空间字典	701
命名空间链接	704
回顾文档字符串	706
类与模块的关系	707
本章小结	708
本章习题	708
习题解答	708

第29章 运算符重载 710

基础知识	710
构造函数和表达式：__init__和__sub__	711
常见的运算符重载方法	711
索引和分片：__getitem__和__setitem__	713
拦截分片	713
Python 2.6中的分片和索引	715
索引迭代：__getitem__	716
迭代器对象：__iter__和__next__	717
用户定义的迭代器	717
有多个迭代器的对象	719
成员关系：__contains__、__iter__和__getitem__	721
属性引用：__getattr__和__setattr__	723
其他属性管理工具	725
模拟实例属性的私有性：第一部分	725
__repr__和__str__会返回字符串表达形式	726
右侧加法和原处加法：__radd__和__iadd__	729
原处加法	730
Call表达式：__call__	731
函数接口和回调代码	732
比较：__lt__、__gt__和其他方法	734

Python 2.6的__cmp__方法（已经从Python 3.0中移除了）	734
布尔测试：__bool__和__len__	735
Python 2.6中的布尔	736
对象析构函数：__del__	738
本章小结	739
本章习题	739
习题解答	739
第30章 类的设计	741
Python和OOP	741
通过调用标记进行重载（或不要）	742
OOP和继承：“是一个”关系	743
OOP和组合：“有一个”关系	744
重访流处理器	746
为什么要在意：类和持续性	748
OOP和委托：“包装”对象	749
类的伪私有属性	751
变量名压缩概览	751
为什么使用伪私有属性	752
方法是对象：绑定或无绑定	754
在Python 3.0中，无绑定方法是函数	756
绑定方法和其他可调用对象	757
为什么要在意：绑定方法和回调函数	760
多重继承：“混合”类	760
编写混合显示类	761
类是对象：通用对象的工厂	771
为什么有工厂	772
与设计相关的其他话题	773
本章小结	773
本章习题	774
习题解答	774

第31章 类的高级主题.....	775
扩展内置类型	775
通过嵌入扩展类型	776
通过子类扩展类型	777
新式类	779
新式类变化.....	780
类型模式变化.....	781
钻石继承变动	785
新式类的扩展	789
slots实例.....	789
类特性	793
__getattribute__ 和描述符.....	795
元类	795
静态方法和类方法.....	796
为什么使用特殊方法	796
Python 2.6和Python 3.0中的静态方法	797
静态方法替代方案	799
使用静态和类方法	800
使用静态方法统计实例	801
用类方法统计实例	802
装饰器和元类：第一部分	805
函数装饰器基础	805
装饰器例子.....	806
类装饰器和元类	807
更多详细信息	808
类陷阱	809
修改类属性的副作用	809
修改可变的类属性也可能产生副作用.....	810
多重继承：顺序很重要	811
类、方法以及嵌套作用域	812
Python中基于委托的类：__getattr__和内置函数	814

“过度包装”814

本章小结815

本章习题815

习题解答815

第六部分练习题816

为什么要在意：大师眼中的OOP821

第七部分 异常和工具

第32章 异常基础 825

为什么使用异常826

异常的角色.....826

异常处理：简明扼要827

默认异常处理器827

捕获异常828

引发异常829

用户定义的异常830

终止行为830

为什么要在意：错误检查832

本章小结833

本章习题833

习题解答834

第33章 异常编码细节..... 835

try/except/else语句835

try语句分句836

try/else分句839

例子：默认行为840

例子：捕捉内置异常.....841

try/finally语句841

例子：利用try/finally编写终止行为842

统一try/except/finally语句843

统一try语句语法845

通过嵌套合并finally和except845

合并try的例子846

raise语句847

 利用raise传递异常849

 Python 3.0异常链：raise from849

assert语句850

 例子：收集约束条件（但不是错误）850

with/as环境管理器851

 基本使用852

 环境管理协议853

本章小结855

本章习题855

习题解答856

第34章 异常对象 857

异常：回到未来858

 字符串异常很简单858

 基于类的异常858

 类异常例子859

为什么使用类异常861

内置Exception类864

 内置异常分类865

 默认打印和状态866

定制打印显示867

定制数据和行为868

 提供异常细节868

 提供异常方法869

本章小结870

本章习题870

习题解答870

第35章 异常的设计 872

 嵌套异常处理器872

 例子：控制流程嵌套.....873

 例子：语法嵌套化874

 异常的习惯用法876

 异常不总是错误876

 函数信号条件和raise.....876

 关闭文件和服务器连接877

 在try外进行调试878

 运行进程中的测试879

 关于sys.exc_info879

 与异常有关的技巧.....880

 应该包装什么881

 捕捉太多：避免空except语句881

 捕捉过少：使用基于类的分类.....883

 核心语言总结884

 Python工具集884

 大型项目的开发工具.....885

 本章小结888

 第七部分练习题889

前言

本书是学习Python编程语言的入门书籍。Python是一种很流行的开源编程语言，可以在各种领域中用于编写独立的程序和脚本。Python免费、可移植、功能强大，而且使用起来相当容易。来自软件产业各个角落的程序员都已经发现，Python对于开发者效率和软件质量的关注，这无论在大项目还是小项目中都是一个战略性的优点。

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，你会对Python有足够的了解，能够将其应用于所从事的领域中。

本书设计成一本教程，主要关注核心Python语言本身，而不是其特定的应用程序。因此，它作为一个两卷本的合集中的第一本：

- 《Learning Python》，也就是这本书，介绍Python本身。
- 《Programming Python》，另外一本书，介绍在学习了Python之后可以用它来做什么。

也就是说，《Programming Python》这本基于应用的图书选择了本书所省略的话题，介绍了Python在Web、图形用户界面（GUI）和数据库这样的常用领域的作用。此外，《Python Pocket Reference》一书提供了本书所没有的额外参考资料，可将它作为本书的补充。

本书在策划之初就力求向读者展示比众多程序员初次学习这门语言的时候更深层次的话题。并且，本书基于一个带有测试和练习的3天的Python培训课程，所以可以作为该语言的一个自学教程。

关于第4版

本书第4版从以下3个方面做出了修改：

- 覆盖了Python 3.0和Python 2.6，本书强调Python 3.0，但是对Python 2.6中的不同之处给出了提示。
- 包含了一些新的章节，主要介绍高级的核心语言话题。
- 重新组织了一些已有的材料，并且使用新的示例扩展它们以便更清楚。

我在2009年撰写本书这一版时，Python分为两支——Python 3.0是新兴的版本并且不兼容地修改了该语言；Python 2.6保持与大量已有的Python代码向后兼容。尽管Python 3被视作是Python的未来，Python 2仍然使用广泛并且会在未来的几年内与Python 3并列地得到支持。尽管只是同一种语言的不同版本，但Python 3.0几乎无法运行为之前版本编写的代码（单单print从语句修改为函数，听上去更合理，但是，它几乎影响到所有已经编写好的Python程序）。

版本的划分使得程序员和图书作者都陷入了两难的境地。尽管编写一本好像Python 2不存在而只介绍Python 3的图书很容易，但这可能无法满足大量基于已有代码的Python用户的需求。大量已有代码都是针对Python 2编写的，并且它们不会很快过时。尽管现在的初学者更关注Python 3，但如果他们必须使用过去编写的代码，那么就必须熟悉Python 2。所有的第三方库和扩展都移植到Python 3可能还需要数年时间，所以Python 2这一分支可能不完全是临时性的。

覆盖Python 3.0和Python 2.6

为了解决这一分歧并且满足所有潜在读者的需求，本书的这一版更新为覆盖Python 3.0和Python 2.6（以及Python 3.X和Python 2.X系列的后续发布）。本书针对使用Python 2编程的程序员、使用Python 3的程序员，以及介于这二者之间的程序员。

也就是说，你可以使用本书来学习任何的Python版本。尽管这里主要关注Python 3.0，但Python 2.6的不同之处和工具也都针对使用旧代码的程序员给出了提示。尽管这两个版本大部分是相同的，但它们还是在一些重要的方面有所不同，对此我将指出两者的区别。

例如，在大多数示例中，我们使用Python 3.0的print调用，但是，我也将介绍Python 2.6的print语句，以便使你能够理解较早的代码。我还广泛地介绍了新功能，例如Python 3.0中的nonlocal语句和Python 2.6以及Python 3.0中的字符串的format方法，当较早的Python中没有这样的扩展时，我将会指出来。

如果你初次学习Python并且不需要使用任何遗留代码，我鼓励你从Python 3.0开始，它清理了这一语言中长久以来的一些瑕疵，同时保留了所有最初的核心思想并且添加了一些漂亮的新工具。

当你阅读本书时，很多流行的Python库和工具可能也支持Python 3.0了，特别是在未来的Python 3.1版本中，可以预期文件I/O性能会有较大的提升。如果你使用基于Python 2.X的一个系统，将会发现本书解决了你所关心的问题，并且将帮助你在未来过渡到Python 3.0。

此外，本版也介绍了其他的Python 2和Python 3的发布版本，尽管一些旧的Python 2.X代码可能无法运行本书的所有示例。例如，尽管在Python 2.6和Python 3.0中都有类装饰器，但我们无法在还没有这一功能的旧Python 2.X中使用它。参见前言中的表0-1和表0-2，它们概括了Python 2.6和Python 3.0中的变化。

注意：就在付梓前不久，本书中还添加了关于未来的Python 3.1版的一些突出的扩展的提示，如：字符串format方法调用中的逗号分隔符和自动字段编号、with语句中的多环境管理器语法、针对数字的新方法等。由于Python 3.1的主要目标是优化，本书也直接应用这一新发布。事实上，由于Python 3.1在Python 3.0后接踵而来，并且最新的Python通常是最好的可用Python，在本书中，术语“Python 3.0”通常指的是Python 3.0引入的但在整个Python 3.X版本中都将存在的语言变化。

新增章

尽管本版的主要目标是针对Python 3.0和Python 2.6更新之前的版本的示例和内容，但我们也增加了5章新内容，以介绍新的主题和增加的内容。

- 第27章是一个新的类教程，使用更加实际的示例来说明Python面向对象编程的基础知识。
- 第36章提供了关于Unicode和字节字符串的详细介绍，并且概括了Python 2.6和Python 3.0中字符串和文件的区别。
- 第37章介绍了特性这样的管理属性工具，并且对描述符给出了新的介绍。
- 第38章介绍了函数和类装饰器，并且给出了全面的示例。
- 第39章介绍了元类，并且将它们与描述符进行了比较和对比。

第27章针对Python中的类和OOP提供了一个渐进的、按部就班的教程。它基于我在近年所教授的培训课程中已经使用的一个现场展示，但是，为了在本书中使用已经对它进行了修改。该章设计来在比此前的示例更为实际的背景中展示OOP，并且说明类概念如何综合运用于较大的、实用的程序中。我期望它在这里与在实际的课程中一样有效。

后面新增的4章收录到了本书的最后一个新增部分中，即“高级话题”部分。尽管这些主题从技术上讲都属于核心语言，但不是每个Python程序员都需要深入了解Unicode文本或元类的细节。因此，这4章单独放到了新的部分中，并且正式地作为可选的阅读材料。例如，关于Unicode和二进制数据字符串的细节已经放入到了此部分中，因为大多数程序员使用简单的ASCII字符串，而不需要了解这些主题。类似地，装饰器和元类通常也只是API构建者才感兴趣的专门话题，而不是应用程序员所感兴趣的话题。

然而，如果你确实使用这些工具，或者使用代码来做这些工作，“高级话题”部分的章节应该能够帮助你掌握其基础知识。此外，这些章的示例包含了学习案例，这些案例把核心语言概念绑定到了一起，并且它们比本书其他部分的示例更充实。由于这个新的部分是可选阅读材料，所以该部分最后只有问答题但没有练习题。

已有内容的修改

此外，之前版本的一些内容已经重新组织了，或者用新的示例进行了补充。例如多继承，在第30章增加了列出类树的一个新的学习示例；第20章增加了手动实现map和zip的生成器的示例；第31章新增的代码说明了静态方法和类方法；第23章介绍了包相对导入；第29章的示例介绍了`_contains_`、`_bool_`和`_index_`运算符重载方法，以及针对分片和比较的新的重载协议。

本版还进行了一些结构上的调整以便更清晰。例如，为了融入新的内容和主题，并且为了避免各章主题的重叠，将前5章划分为两部分。这样一来关于运算符重载、作用域和参数、异常语句细节，以及解析和迭代主题就都有了新的独立的章。已有的章内部也进行了一些重新排序，以便更好地介绍主题。

本版还试图通过一些重新排序来减少一些向后引用，尽管Python 3.0的变化使得在某些情况下不可能这么做。要理解打印和字符串格式化方法，现在必须知道针对函数的关键字参数；要理解字典键列表和键测试，现在必须知道迭代；要使用exec来运行代码，需要能够使用文件对象，等等。顺序阅读可能还是最有意义的，但是一些主题可能需要非线性的跳跃和随机查找。

总的来说，本版中有几百处修改。前言的下一个小节，记录下了Python中的27处增加和57处修改。实际上，可以说本版变得更加高级，因为Python多少变得更加高级了。针对Python 3.0自身，你最好能自己发现本书中的修改之处，而不是通过这个前言来了解这些修改。

Python 2.6和Python 3.0中的特定语言扩展

Python 3.0是一种清晰的语言，但是它也是在某些方面更为复杂的一种语言。实际上，它的一些修改似乎假设你必须为了学习Python而已经了解Python。前面的部分概括了Python 3.0中的一些基础知识。随便举个例子，把字典视图包含到一个list调用中的合理性，该问题是难以置信的细微，并且需要实质预测。除了教授Python的基础知识，本书还充当了跨越这些知识鸿沟的桥梁。表0-1列出了本版中介绍的大多数显著的新的语言功能，并且列出了介绍它们的主要的章。

表0-1: Python2.6和Python 3.0中的扩展

扩展	介绍的章
Python 3.0中的print函数	11
Python 3.0中的nonlocal x,y语句	17
Python 2.6和Python 3.0中的str.format方法	7
Python 3.0中的字符串类型: str用于Unicode文本, bytes用于二进制数据	7、36
Python 3.0中的文本和二进制文件的区别	9、36
Python 2.6和Python 3.0中的类装饰器: @private('age')	31、38
Python 3.0中的新的迭代器: range、map、zip	14、20
Python 3.0中的字典视图: D.keys、D.values、D.items	8、14
Python 3.0中的除法运算符: 余数、/和//	5
Python 3.0中的集合常量: {a, b, c}	5
Python 3.0中的集合解析: {x**2 for x in seq}	4、5、14、20
Python 3.0中的字典解析: {x: x**2 for x in seq}	4、8、14、20
Python 2.6和Python 3.0中的二进制位字符串支持: 0b0101、bin(I)	5
Python 2.6和Python 3.0中的分数类型: Fraction(1, 3)	5
Python 3.0中的函数注解: def f(a:99, b:str)->int	19
Python 3.0中的Keyword-only参数: def f(a, *b, c, **d)	18、20
Python 3.0中的扩展的序列分解: a, *b = seq	11、13
Python 3.0中可用的针对包的相对导入语法: from	23
Python 2.6和Python 3.0中可用的环境管理器: with/as	33、35
Python 3.0中的异常语法修改: raise、except/as、superclass	33、34
Python 3.0中的异常链: raise e2 from e1	33
Python 2.6和Python 3.0中的保留字的变化	11
Python 3.0中的新式类的取消	31

表0-1: Python2.6和Python 3.0中的扩展 (续)

扩展	介绍的章
Python 2.6和Python 3.0中的特性装饰符: @property	37
Python 2.6和Python 3.0中的描述符	31、38
Python 2.6和Python 3.0中的元类	31、39
Python 2.6和Python 3.0中的抽象基类支持	28

Python 3.0中特定的语言删除

除了扩展，还有一些语言工具从Python 3.0中删除了，以清理其设计。表0-2概括了影响到本书的这些变化，以及本版中介绍它们的不同的章。表0-2中列出的很多删除都有直接的替代者，其中的一些在Python 2.6中还可用，以支持未来向Python 3.0的迁移。

表0-2: 影响到本书的Python 3.0中的删除

删除的	替代的	介绍的章
reload(M)	imp.reload(M) (或exec)	3、22
apply(f, ps, ks)	f(*ps, **ks)	18
'X'	repr(X)	5
X <> Y	X != Y	5
long	int	5
9999L	9999	5
D.has_key(K)	K in D (或D.get(key) != None)	8
raw_input	input	3、10
old input	eval(input())	3
xrange	range	14
file	open (以及io模块类)	9
X.next	X.__next__, 由next(X)调用	14、20、29
X.__getslice__	X.__getitem__, 传入一个slice对象	7、29
X.__setslice__	X.__setitem__, 传入一个slice对象	7、29
reduce	functools.reduce (或循环代码)	14、19
execfile(filename)	exec(open(filename).read())	3
exec open(filename)	exec(open(filename).read())	3
0777	0o777	5
print x, y	print(x, y)	11

表0-2: 影响到本书的Python 3.0中的删除 (续)

删除的	替代的	介绍的章
<code>print >> F, x, y</code>	<code>print(x, y, file=F)</code>	11
<code>print x, y,</code> <code>u'ccc'</code>	<code>print(x, y, end=' ')</code> <code>'ccc'</code>	11 7、36
<code>'bbb'</code> for byte strings	<code>b'bbb'</code>	7、9、36
<code>raise E,V</code>	<code>raise E(V)</code>	32、33、34
<code>except E, X:</code>	<code>except E as X:</code>	32、33、34
<code>def f((a, b)):</code>	<code>def f(x): (a, b) = x</code>	11、18、20
<code>file.xreadlines</code>	<code>for line in file: (or X=iter(file))</code>	13、14
<code>D.keys()</code> 等	<code>list(D.keys())</code> (字典视图)	8、14
<code>map()</code> , <code>range()</code> , etc. as lists	<code>list(map())</code> , <code>list(range())</code> (内置函数)	14
<code>map(None, ...)</code>	<code>zip</code> (或手动代码来补充结果)	13、20
<code>X=D.keys(); X.sort()</code>	<code>sorted(D)</code> (或 <code>list(D.keys())</code>)	4、8、14
<code>cmp(x, y)</code>	<code>(x > y) - (x < y)</code>	29
<code>X.__cmp__(y)</code>	<code>__lt__</code> 、 <code>__gt__</code> 、 <code>__eq__</code> 等	29
<code>X.__nonzero__</code>	<code>X.__bool__</code>	29
<code>X.__hex__</code> , <code>X.__oct__</code>	<code>X.__index__</code>	29
排序比较函数	使用 <code>key=transform</code> 或 <code>reverse=True</code>	8
Dictionary <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	<code>Compare sorted(D.items())</code> (或循环代码)	8、9
<code>types.ListType</code>	<code>list</code> (<code>types</code> 只用于非内建名称)	9
<code>__metaclass__ = M</code>	<code>class C(metaclass=M):</code>	28、31、39
<code>__builtin__</code>	<code>builtins</code> (重命名)	17
<code>Tkinter</code>	<code>tkinter</code> (重命名)	18、19、24、29、30
<code>sys.exc_type</code> , <code>exc_value</code>	<code>sys.exc_info()[0]</code> , <code>[1]</code>	34、35
<code>function.func_code</code>	<code>function.__code__</code>	19、38
由内置函数运行的 <code>__getattr__</code>	在包装类中重定义 <code>__x__</code> 方法	30、37、38
<code>-t</code> , <code>__tt</code> 命令行切换	不一致地使用制表符/空格总是一个错误	10、12
一个函数中的 <code>from ... *</code>	只能够出现在一个文件的顶层	22
导入模块, 在同一包中	<code>from . import mod</code> , 包相关的形式	23

表0-2：影响到本书的Python 3.0中的删除（续）

删除的	替代的	介绍的章
<code>class MyException:</code>	<code>class MyException(Exception):</code>	34
<code>exceptions module</code>	内置作用域，库手册	34
<code>thread、Queue modules</code>	<code>_thread, queue</code> (二者都改名了)	17
<code>anydbm module</code>	<code>dbm</code> (改名了)	27
<code>cPickle module</code>	<code>_pickle</code> (改名了，自动使用)	9
<code>os.popen2/3/4</code>	<code>subprocess.Popen</code> (<code>os.popen</code> 保留)	14
基于字符串的异常	基于类的异常 (Python 2.6中也是如此)	32、33、34
字符串模块函数	字符串对象方法	7
未绑定方法	函数 (通过实例调用静态方法)	30、31
混合类型比较、排序	非数字的混合类型比较是错误	5、9

Python 3.0中还有其他的修改没有包含在该表中，只是因为它们没有影响到本书。例如，标准库中的修改，那些修改可能对《Programming Python》这样关注应用程序的图书的影响比对本书的影响要大，尽管很多标准库的功能仍然存在，Python 3.0很大程度地重命名了模块，将它们组织到包中等。要更全面地了解Python 3.0中的变化，可参阅Python的标准手册集中的“*What’s New in Python 3.0*”文档，其中包含一个更全面的列表。

如果你正在从Python 2.X迁移到Python 3.X，那么应该看一下Python 3.0中的2到3自动代码转换脚本。它并不能够转换任何内容，但是，它做了合理的工作来把大量的Python 2.X代码转换为在Python 3.X下可运行的代码。在我编写本书时，一个新的3到2的反向转换项目也在进行之中，它可以把Python 3.X代码转换为在Python 2.X下运行。如果你必须针对两个Python系列版本维护代码的话，这两种工具都很有用，想深入了解可参见Web上的详细介绍。

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，所以阅读之前了解本书的写作目的是很重要的。

事前准备

事实上，阅读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从容地阅读本书。如果打算学习Python，本书可能很适合你。如果你曾有过编程或编写脚本的经验，那么会对你阅读本书有一点帮助。但是，并不要求每位读者都得这样。

本书设计为程序员^{注1}学习Python的入门书籍。对于那些从来没有接触过计算机的人，可能就不适合了（例如，我不会花时间讨论计算机是什么）。但是，这并不意味着你需要有编程的经历。

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，我们不会谈论太多的Python/C集成，这个复杂话题显然是许多基于Python的系统的核心。我们也不会谈论太多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的学习，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

注1：所谓的“程序员”，是指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。

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》提供了有用的参考，而O'Reilly的《Python Cookbook》则为那些已熟知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，建议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，以至于这里没有空间能够容纳。

尽管这么说，我想读者还是会发现本书是学习Python的首选书籍，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，不仅学习了语言本身，也会学到如何将其合理地运用于日常工作中。此外，当读者遇到更高级的主题和例子时，将会有足够的能力去解决它们。

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中给出。习题可以帮助读者复习学过的内容，而练习题可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。

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介绍语言的基础的，所以以语言的主要功能进行组织并介绍，而不是以例子为主。这里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，一个部分介绍一种主要语言功能（类型以及函数等），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）。具体来说，本书内容如下。

第一部分，使用入门

我们以概览Python作为开始，回答了一些常见的问题：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。

第二部分，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，因为这部分内容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Python的关键。

第三部分，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。

第四部分，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。

第五部分，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。

第六部分，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，但是可以帮助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更是如此。

第七部分，异常和工具

本书介绍语言基础的最后一部分，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）。尽管异常是相当轻量级的工具，这一部分放在类介绍之后，是因为异常现在应该都是类了。

第八部分，高级话题（第4版新增部分，请到华章网站下载）

在最后一部分中，我们介绍了一些高级话题。这里，我们学习了Unicode和字节字符串、特性和描述符这样的管理属性工具、函数和类装饰器，以及元类。这些章都是选读的，因为并不是所有的程序员都需要理解这些章所介绍的话题。另一方面，必须处理国际化文本或二进制数据的读者，或者负责开发API供其他程序员使用的读者，应该会对本部分感兴趣。

第九部分，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。

书籍更新

本书在不断地进行完善（输入错误也包括在内）。本书原版的更新、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/9780596158064/>（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

书中所有程序的例子都是基于Python 3.0版的。此外，这些例子中的大多数都能在Python 2.6下运行，正如本书所提到的，Python 2.6读者的注意事项也会在其中给出提示。

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/9780596158064/>。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。

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，不需要许可。销售和分发O'Reilly范例光盘需要许可。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、出版社和ISBN。例如，“*Learning Python*, Fourth Edition, by Mark Lutz. Copyright 2009 Mark Lutz, 978-0-596-15806-4”。

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。

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。

等宽粗体 (**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。

等宽斜体 (*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。

注意： 本书例子中，系统命令行开头的%字符指的是系统提示符，这取决于读者的机器（例如，DOS窗口是C:\Python30>）。不要自行输入%字符（或者有时候它表示的系统提示）。

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，一般也不需要输入程序清单中以#开头的文字，这些是注释，不是可执行的代码。

联系我们

有关本书的任何建议和疑问，可以与下面的出版社联系：

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

我们为本书提供了一个网页，其中给出了勘误表、示例和所有的附加信息。可以通过以下地址访问该网页：

<http://www.oreilly.com/catalog/9780596158064>

要对本书发表评论或询问技术问题，请发电子邮件到：

bookquestions@oreilly.com

有关我们的书籍、会议、资源中心以及O'Reilly网络，可以访问我们的网站：

<http://www.oreilly.com>

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

致谢

当我在2009年写本书第4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 17年了，而且也已经有12年Python的培训经验了。尽管时间在不断流逝，我依然惊讶于Python这些年来取得的成功。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17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2009年中写下这些话时，已经编写了12本Python书籍（3本是第4版）；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225个Python短期培训课程，在此过程中遇到了超过3000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，教学磨炼了书籍，而书籍又反过来磨炼了教学。事实上，你读的这本书的大部分内容都源于我的课程。

因此，我想要感谢过去12年来参加我培训的所有学生。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）。本版主要根据2003年后的课程进行修正，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、巴塞罗纳、伦敦、埃德蒙顿以及波多黎各举办课程的那些客户，无法想象有比这更好的培训地点了。

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Julie Steele、前一版的Tatiana Apandi，以及前几版的许多人。感谢Doug Hellmann和Jesse Noller参与本书的技术校对。感谢O'Reilly让我有机会写出这12本书；真的很有趣（感觉上有点像电影《偷天情缘（Groundhog Day）》）。

感谢最初的共同作者David Ascher对本书前两版的帮助。David在前几版中贡献了“外层”部分，但是从第3版开始，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实例和一个全新的高级主题部分作为弥补。请参见本前言前面对于后续应用级内容的说明，一旦你学习了基础知识之后，可能想要学习这些应用级内容。

特别感谢Guido van Rossum和Python社区的人们创造了如此有趣和实用的语言。就像多数开源系统一样，Python是许多英雄努力的结果。拥有了17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O'Reilly公司我最初的编辑：已故的Frank Willison。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。感谢我的妈妈给予我鼓励。感谢我这几年遇到的所有大型公司，提醒我自己有多么幸运，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。

最需要感谢的是Vera，我最好的朋友、女友以及妻子。我最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。

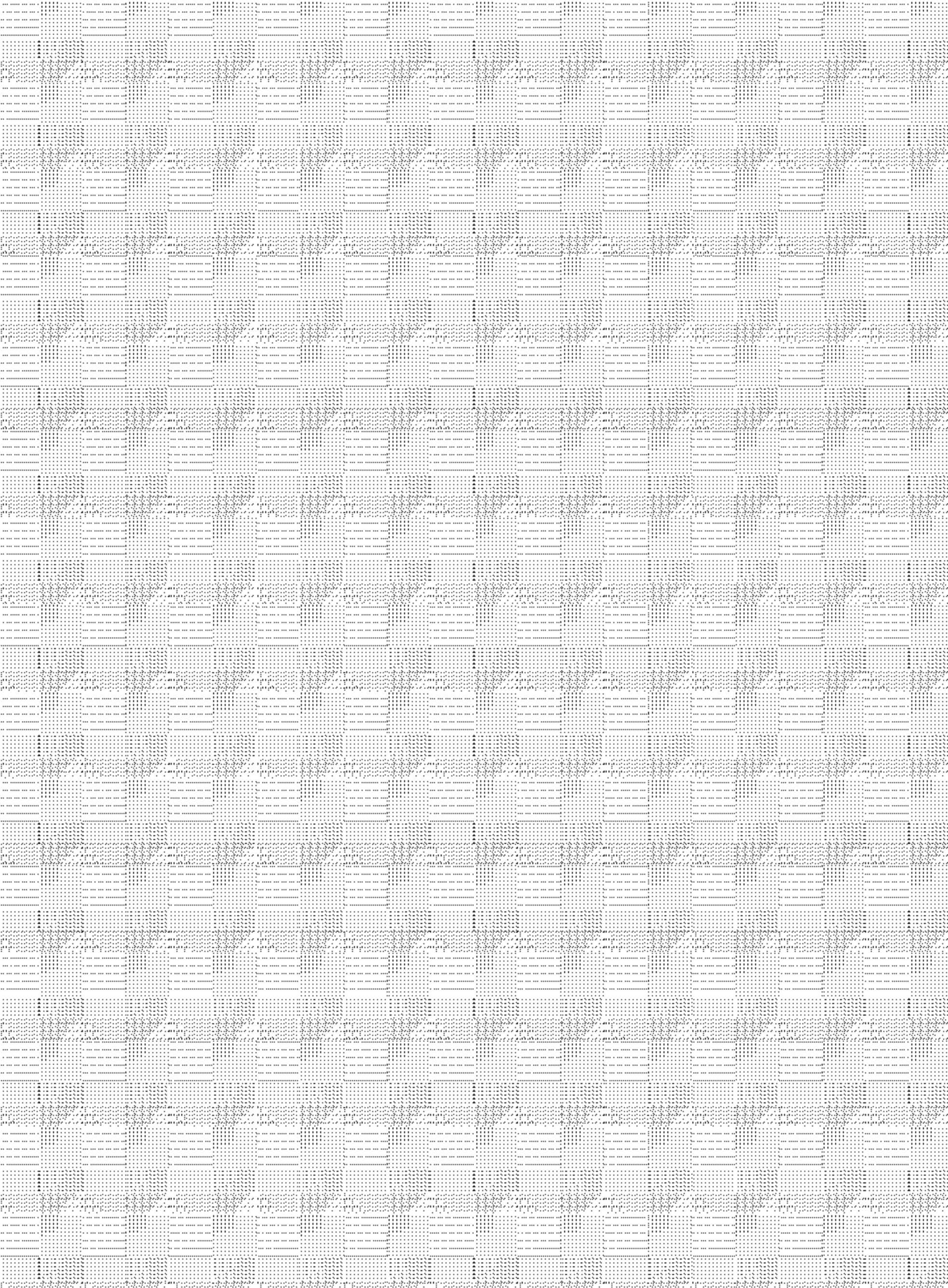
Mark Lutz

佛罗里达州萨拉索市

2009年7月

第一部分

使用入门



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。本书首先会简要介绍一下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100万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12年中，在对近225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的主要因素有以下几个方面。

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。即使代码并非你亲手所写，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，object-oriented programming）。

提高开发者的效率

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1/5~1/3。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/静态语言所必需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的系统等。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，这些功能模块叫做标准库（standard 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是一个免费的、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代码可以调用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高可读性的语法，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确实确实是“符合大脑思维习惯”，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，往往要求开发者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。

注1： 要了解完整的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。

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。

由于“脚本语言”从不同的视角来观察时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，其中有些角色相对其余的角色更重要。

Shell 工具

当人们听到Python是脚本语言时，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。它不只是一种很好的Shell脚本语言。

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，测试硬件设备时，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不只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。

好吧，Python的缺点是什么呢

在经过17年的Python使用和12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。

本书后面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是采用双语言混编策略的一个重要例子：作为一个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

在编写本书的时候，乐观估计，这个时候全球的Python用户将达到100万（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社区中受益不少。由于Python有近

19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：

- 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。
- EVE Online这款大型多人网络游戏（Massively Multiplayer Online Game, MMOG），广泛地使用Python。
- Maya这款强大的集成化3D建模和动画系统，提供了一个Python脚本编程API。
- 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。

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，如果你对这些问题感兴趣，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。

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。EVE Online所采用的Stackless Python还为多处理需求提供了高级的解决方案。

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，你可以在Python中使用其他的GUI工具包，例如，通过PyQt使用Qt、通过PyGTK使用GTK、通过PyWin32使用MFC、通过IronPython使用.NET，以及通过Jython（Java版本的Python，我们将会在第2章中进行介绍）使用Swing等。对于运行于浏览器中的应用或具有一些简单界面需求的应用，Jython和Python Web框架以及服务器端CGI脚本（下一节将介绍）都提供了其他的用户界面的选择。

Internet脚本

Python提供了标准Internet模块，它使得Python程序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脚本的表单中提取信息；通过FTP传输文件；解析、生成和分析XML文件；发送、接受、编写和解析Email；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，Jython系统提供了无缝的Python/Java集成而且支持在客户端运行的服务器端Applet。

此外，涌现了许多针对Python的Web开发工具包，例如，*Django*、*TurboGears*、*web2py*、*Pylons*、*Zope*和*WebWare*，它们使得Python能够快速构建功能完善和高质量的网站。很多这样的工具包包含了诸如对象关系映射器、模型/视图/控制器架构、服务器端脚本和模板，以及支持AJAX等功能，从而提供了完整的、企业级的Web开发解决方案。

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黏合语言，可以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，SWIG和SIP这样的代码生成工具可以让这部分工作自动完成，并且CPython系统允许代码混合到Python和类似C的代码中。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等。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）——你只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对象保存和恢复到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。

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。例如，常用的SciPy和ScientificPython扩展，为使用科学编程工具以及NumPy代码提供了额外的库。

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。
- 使用PySerial扩展在Windows、Linux以及更多系统上进行串口通信。
- 用PIL、PyOpenGL、Blender、Maya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。

你甚至可以使用PySol程序下棋娱乐。可以从PyPI网站或通过网络搜索（请在<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如何获得支持

作为流行的开源系统之一，Python拥有一个很大而且活跃的开发社区，它以令众多商业软件开发者认为不凡（如果没有完全震惊的话）的速度进行版本更新和开发改进。Python开发者使用一个源代码控制系统在线协同地工作。修改遵从从一个正式的PEP（Python Enhancement Proposal）协议并且必须经过Python的扩展回归测试系统。实际上，现在修改Python差不多和修改商业软件一样的，与早期的Python大不相同，那时候，只需要给Python的创始人发一封E-mail就够了，但在当前用户量巨大的情况下，前面的修改方法更好。

一个非正式的组织PSF（Python Software Foundation，Python软件基金会），负责组织会议并处理知识产权的问题。世界各地举办了大量的Python会议，O'Reilly的OSCON和PSF的PyCon是其中最大的会议。前者还涉及多个开源项目，后者则是专门的Python会议并且近年来规模显著扩大。PyCon 2008的参会者几乎是前一年的两倍，从2007年的586名参会者增加到2008年的超过1000名。这一增长仅次于2007年的40%的增速，2006年参加这一会议的人数是410人。PyCon 2009有943名参会者，和2008年相比略有减少，但是这在全球经济萧条的环境中仍然显得很强势。

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，这将对Python一些最优的技术特性的快速介绍。

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。尽管研究或改变一种程序语言的实现并不是对每一个人来说都那么有趣，但是当你知道如果需要的话可以做到这些，该是多么的令人欣慰。你不需要去依赖商业厂商的智慧，有无尽的文档和源代码随你使用。

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）：

- 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。

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（更多详细内容将在第2章中介绍）。

这些意味着Python程序的核心语言 and 标准库可以在Linux、Windows和其他带有Python解释器的平台无差别地运行。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。这些工具允许你把系统组织为组件，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。

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。

简单易用

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++）所必需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。

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语言的核心是非常简单易学。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。

Python和其他语言比较起来怎么样

最后，你也许已经知道了，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PHP更易懂并且用途更广。Python有时候用来构建Web站点，但是，它也广泛地应用于几乎每个计算机领域，从机器人到电影动画。
- 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，而不是推销它。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。

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。目的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。

本章习题

本书的这个版本，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。而问题的答案紧随问题之后，建议你独立完成测验后马上查看参考答案。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？

习题解答

做得怎么样？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、组件集成和享受简便其中，质量和效率这两条是人们选择Python的主要原因。
2. Industrial Light & Magic、EVE Online、Jet Propulsion Labs、Maya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。
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。

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。

这个故事反映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，而其设计反映了这种传统。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用户向我们透露过，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。

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。根据选用的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。
- Linux和Mac OS X 用户也许已经拥有了一个可用的Python预先安装在了计算机上：如今Python已成为这些平台的标准组件。
- 一些Linux用户和Mac OS X用户（和大多数UNIX用户一样）可从Python的完整源代码分发包中编译安装。
- Linux用户也可以找到RPM文件，并且Mac OS X用户可以找到各种特定于Mac的安装包。
- 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。

我们可以通过Python官方网站（<http://www.python.org>）下载获得Python，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。

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。

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`script0.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^{100} ）。不用为这段代码中的语法担心，我们这一章的



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）

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^{100} 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，这也将在这本书后边进行介绍，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、从IDE中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屏幕的某处：一般默认是显示在运行程序的那个窗口。

```
hello world
1267650600228229401496703205376
```

例如，这就是我在一个Windows笔记本的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。

```
C:\temp> python script0.py
hello world
1267650600228229401496703205376
```

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编程所必需的要求，然而对Python的运行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（参考本章的“冻结二进制文件”小节获得其他发布的选项）。

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），之后的字节码发送到通常称为Python虚拟机（Python Virtual Machine，简称为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”的最后一步。

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。

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）。字节码是特定于Python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：

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章说明。

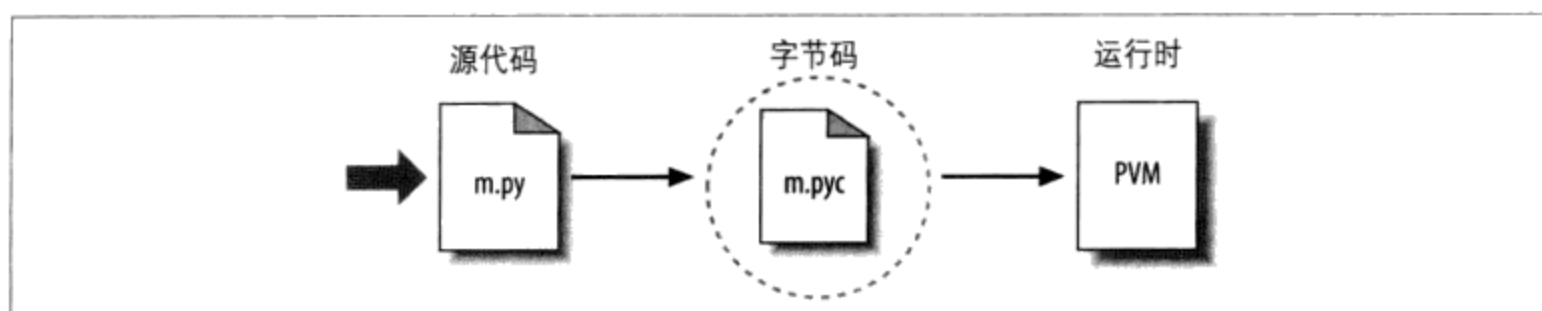


图2-2: Python的传统运行执行模式: 录入的源代码转换为字节码, 之后字节码在Python虚拟机中运行。代码自动被编译, 之后再解释

PVM循环（而不是CPU芯片）仍然需解释字节码，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，编译器总是在运行时出现，并且是运行程序系统的一部分。

这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中真正拥有的只有运行时：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。

执行模块的变体

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。

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，它往往看做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的第三种实现方式IronPython（比CPython和Jython都要新），其设计目的是让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，即通过把源代码编译为字节码，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。

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，JPytype以及.NET系统的Python，允许Python调用Java以及.NET的组件。

因为字节码的转换与程序运行同时发生，所以Psyco往往被看做是一个即时编译器（JIT）。Psyco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。

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移到C的那部分代码。使用了Psyco后，这样的迁移甚至没有必要了。

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，写作本书的时候，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。在编写本书的时候，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。

其他执行选项

还有一些其他的方案可以用来运行Python程序，它们具有更加专注的目标：

- *Stackless Python*系统是标准CPython实现的一个变体，它不会在C语言调用栈上保存状态。这使得Python更容易移植到较小的栈架构中，提供了更高效的多处理选项，并且促进了像coroutine这样的新奇的编程结构。

- *Cython*系统（基于Pyrex项目所完成的工作）是一种混合的语言，它为Python代码结合了调用C函数以及使用变量、参数和类属性的C类型声明的能力。*Cython*代码可以编译成使用Python/C API的C代码，随后可以再完整地编译。尽管与标准Python并不完全兼容，*Cython*对于包装外部的C库以及提高Python的C扩展的编码效率都很有用。

要了解有关这些系统的详细信息，可以在Web上搜索最近的链接。

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（尽管在最近的20年里还没有一款这样的编译器）。未来也许会有新的字节码格式和实现方式的变体将被采用。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>）。Python自己的PVM运行Python代码比Parrot效率更高，但Parrot如何发展还不明晰。
- 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。其目标是产生一个快速而灵活的Python实现。

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体（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。

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标准执行模块的变体的名字。
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，它是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体。
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。

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。假设已经在计算机上安装好了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，我们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，以及IDLE这样的GUI中的菜单选项等内容。

如果你只想知道如何快速地运行Python程序，建议你阅读与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，能帮助你开发出更为精致的Python程序。

交互提示模式下编写代码

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平台无关的方法，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 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
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“运行”的对话框中输入也可以。
- 在UNIX、Linux以及Mac OS X中，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，在手持设备上，通常可以点击主窗口或应用程序窗口中的Python图标来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UNIX或Linux上，可以输入**/usr/local/bin/python**（或**/usr/bin/python**）；在Windows上，可以尝试输入**C:\Python30\python**（对于3.0版本）。

```
C:\misc> c:\python30\python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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:\python30**）之后运行“python”。例如：

```
C:\misc> cd C:\Python30
C:\Python30> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在Windows中，除了在shell窗口中输入**python**，也可以通过启动IDLE的主窗口（随后介绍）或者通过从Python的Start按钮菜单的菜单选项中选择“Python (command line)”来开始类似的交互会话（如第2章中的图2-1所示）。这两种方式都会产生一个具有同样功能的Python交互式命令提示符，而不必输入一条shell命令。

交互地运行代码

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），然后显示等待输入新的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按下Enter键后在**>>>**这一行之后显示。

例如，这里是两条Python `print`语句的结果（`print`在Python 3.0中确实是一个函数调用，但在Python 2.6中不是，因此，这里的括号只在Python 3.0中需要）：

```
% 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，正如每个`>>>`输入行下边的输出行显示的那样（在Python中，`2 ** 8`的意思是2的8次方）。

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会立即运行。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
>>>                                     <== Use Ctrl-D (on Unix) or Ctrl-Z (on Windows) to exit
%
```

此处，第一行把一个值赋给了一个变量从而保存它，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系统中输入Ctrl-D退出；在MS-DOS和Windows系统中输入Ctrl-Z退出。在随后讨论到的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的了解：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的学习。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。在本章后面你将看到，也可以在交互提示符中运行多行语句，在你输入了所有语句行并且两次按下Enter键添加一个空行之后，会立即运行这条语句。

为什么使用交互提示模式

交互提示模式根据用户的输入运行代码并响应结果，但是，它不会把代码保存到一个文件中，尽管这意味着你不能在交互会话中编写大量的代码，但交互提示仍然是体验语言和测试编写中的程序文件的好地方。

实验

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。

例如，假设你在阅读一个Python程序的代码并且遇到了像 `'Spam!' * 8` 这样一个不理解其含义的表达式。此时，你可能要花上10分钟来尝试搞清楚这段代码做什么，或者你可以直接交互式地运行它：

```
>>> 'Spam!' * 8                                     <== Learning by trying
'Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!'
```

通过交互提示模式接收到的直接反馈，通常是搞清楚一段代码到底做什么的最快的方式。这里，它清楚地显示：这条语句重复字符串，在Python中，`*`表示数字相乘，但对于字符串来说，表示重复，就像是重复地把一个字符串连接到其自身（本书第4章将详细介绍字符串）。

这种体验方式不会带来任何破坏（至少目前还没有），这是不错的。要进行真正的破坏，例如删除文件并运行shell命令，你必须尝试显式地导入模块（你还需要对Python的系统接口了解更多，才能变得这么有危险性）。直接的Python代码总是可以安全运行的。

例如，当你在交互提示模式中犯了一个错误的时候，看看会发生什么情况：

```
>>> X                                               <== Making mistakes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
```

在Python中，给一个变量赋值之前就使用它，这总是一个错误（否则，如果名字总是填充了默认值，一些错误将变得无法检测）。我们稍后将更详细地了解这一点，这里的重要之处在于，当你犯了这样的一个错误的时候，不会导致Python或计算机崩溃。相反，你会得到一条有意义的出错消息，指出该错误以及出错的代码行，并且你可以继续自己

的会话或脚本。实际上，一旦你熟悉了Python，其出错消息通常能够提供尽你所需的调试支持。

测试

除了充当学习语言的体验工具，交互式解释器也是测试已经写入到文件中的代码的好地方。你可以交互地导入模块文件，并且通过在交互提示模式中输入命令从而在它们定义的工具上运行测试。

例如，下面的代码在Python的标准库所附带的一个预编码的模块中测试一个函数（它显示出我们当前所工作的目录的名称），但一旦开始编写自己的模块文件，也可以做同样的事情：

```
>>> import os
>>> os.getcwd()                <== Testing on the fly
'c:\\Python30'
```

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，你可以在Python文件中导入并测试函数和类，通过输入命令来连接C函数，在Jython中使用的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。

使用交互提示模式

尽管交互提示模式简单易用，这里还有一些初学者需要牢记的技巧。在本章中，我列出了一些常见错误的列表以供参考，但是，如果你提前阅读它们的话，会帮助你避免一些令人头疼的问题。

- **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两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。

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。

- **在交互提示模式中，用一个空行结束复合语句。**在交互提示模式中，要告诉交互式Python已经输入完了多行语句，必须要插入一个空行（通过在一行的起始处按下Enter键）。也就是说，你必须按下Enter键两次，才能运行一条复合语句。相反，在文件中空行是不需要的，并且如果有的话也将会忽略。在交互模式下工作的时候，如果你没有在一条复合语句的末尾两次按下Enter键，将会陷入到尴尬的境地，因为交互式解释器根本什么也不会做，它等着你再次按下Enter键。
- **交互提示模式一次运行一条语句。**在交互提示模式中，你必须运行完一条语句，然后才能输入另一条语句。对于简单语句来说，这很自然，因为按下Enter键就可以运行输入的语句。然而，对于复合语句，记住必须提交一个空行来结束该语句，然后运行它，之后才能够输入下一条语句。

输入多行语句

冒着重复自己的风险，在更新本章内容的时候，我收到了受最后两项错误伤害的读者的邮件，因此，这两项错误还是值得强调的。我将在下一章中介绍多行（即复合）语句，并且我们将在本书后面更正式地介绍其语法。由于它们在文件中和在交互提示模式中的行为略有不同，因此，这里有两点要注意。

首先，在交互提示模式中，注意像结束for循环和if测试那样，用一个空行结束多行复合语句。必须两次按下Enter键，来结束整个多行语句，然后让其运行。例如：

```
>>> for x in 'spam':  
...     print(x)  
...                                     <== Press Enter twice here to make this loop run
```

在脚本文件中，复合语句的后面不需要空行；只在交互提示模式下，才需要该空行。在

文件中，空行不是必须的，如果出现了的话，将会直接忽略掉；在交互提示模式中，它们会结束多行语句。

还要记住，交互提示模式每次只运行一条语句：必须两次按下Enter键来运行一个循环或其他的多行语句，然后才能输入下一条语句：

```
>>> for x in 'spam':  
...     print(x)                                     <== Need to press Enter twice before a new statement  
...     print('done')  
File "<stdin>", line 3  
    print('done')  
    ^  
SyntaxError: invalid syntax
```

这意味着不能在交互提示模式中复制并粘贴多行代码，除非这段代码的每条复合语句的后面都包含空行。这样的代码最好在一个文件中运行，下一小节将讨论这一话题。

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－粘贴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。为了从交互对话模式中复制－粘贴代码，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，这实在不是一种现代的软件开发方法。

为了能够永久的保存程序，需要在文件中写入代码，这样的文件通常叫做模块。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叫做脚本（一个顶层程序文件的非正式说法）。有些人将“模块”这个说法应用于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。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。尽管对某些人来说这似乎有些粗糙简单，但对于很多程序员而言，一

个系统shell命令行窗口加上一个文本编辑器窗口，这就组成了他们所需的一个集成开发环境的主要部分。

第一段脚本

让我们开始吧。打开文本编辑器（例如，vi、Notepad或IDLE编辑器），并在命名为 *script1.py* 的新文本文件中输入如下Python语句：

```
# A first Python script
import sys                # Load a library module
print(sys.platform)
print(2 ** 100)           # Raise 2 to a power
x = 'Spam!'
print(x * 8)              # String repetition
```

这个文件是我们第一个正式Python脚本（不算第2章中仅2行的那个脚本）。对于这个文件中的代码，我们应该不会担心太多，但是，简要来说，这个文件：

- 导入一个Python模块（附加工具的库），以获取系统平台的名称。
- 运行3个print函数调用，以显示脚本的结果。
- 使用一个名为x的变量，在创建的时候对其赋值，保存一个字符串对象。
- 应用我们将从下一章开始学习的各种对象操作。

这里的sys.platform只是一个字符串，它表示我们所工作的计算机的类型，它位于名为sys的标准Python模块中，我们必须导入以加载该模块（稍后将详细介绍导入）。

为了增加乐趣，我在这里还添加了一些正式的Python注释，即#符号之后的文本。注释可以自成一行，也可以放置在代码行的右边。#符号后的文本直接作为供人阅读的注释而忽略，并且不会看做是语句的语法的一部分。如果你要复制这些代码，也可以忽略掉注释。在本书中，我们通常使用一种不同的格式体例来让注释更加容易识别，但是，在代码中，它们是作为正常文本显示的。

此外，现在不要关注这个文件中的代码的语法；我们随后将学习其所有的语法。要注意的主要一点是，我们已经把这段代码输入到一个文件中，而不是输入到交互提示模式中。在这个过程中，我们已经编写了一个功能完整的Python脚本。

注意，这个模板文件叫做 *script1.py*。对于所有的顶层文件，也应该直接叫做脚本，但是，要导入到客户端的代码的文件必须用.py后缀。我们将在本章稍后学习导入。此外，一些文本编辑器通过.py后缀来检测Python文件；如果没有这个后缀，可能无法使用诸如语法着色和自动缩进等功能。

使用命令行运行文件

一旦已经保存了这个文本文件，可以将其完整的文件名作为一条python命令的第一个参数，在系统shell提示中输入，从而要求Python来运行它：

```
% python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

再次，我们可以在系统所提供的用于命令行的任何入口（例如一个Windows命令提示符窗口、一个xterm窗口，或者类似的窗口）中，输入这样的一个系统shell命令。记住，如果你的PATH设置没有配置的话，要像前面一样，用完整的目录路径替换“python”。

如果一切工作按计划进展，这条shell命令将使得Python一行一行地运行这个文件中的代码，并且，我们将会看到该脚本的3条print语句的输出：底层平台的名称、 2^{100} 以及我们前面所见过的相同的字符串重复表达式的结果（第4章将介绍关于后两者的更多细节）。如果一切没有按计划进行，你将会得到一条错误消息，确保已经在文件中输入了这里所示的代码，并再次尝试。我们将在本章后面的“调试Python代码”部分介绍调试选项，但是，目前你可能只能死记硬背。

由于这种方法使用shell命令行来启动Python程序，所有常用的shell语法都适用。例如，我们可以使用特定的shell语法，把一个Python脚本的输出定向到一个文件中，从而保存起来以备以后使用或查看：

```
% python script1.py > saveit.txt
```

在这个例子中，前面的运行中的3个输出行都存储到了*saveit.txt*，而不是显示出来。这通常叫做流重定向（*stream redirection*），它用于文本的输入和输出，而且在Windows和类似UNIX的系统上都可以使用。它几乎和Python不相关（Python只是支持它而已），因此，我们在这里略过有关shell重定向语法的细节。

如果你仍然在Windows平台上工作，这个例子也同样有效，但是，系统提示通常有所不同：

```
C:\Python30> python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

通常，如果你没有把PATH环境变量设置为包含这一路径，或者没有执行切换目录命令来找到该路径的话，要确保输入了到Python的完整路径：

```
D:\temp> C:\python30\python script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

在较新的Windows版本上，我们也可以只是输入脚本的名字，并省略掉Python本身的名字。由于新的Windows系统使用Windows注册表找到用哪个程序来运行一个文件，我们不需要在命令行上显式地使用名字“python”来运行一个.py文件。例如，在大多数Windows机器上，前面的命令可以缩写为：

```
D:\temp> script1.py
```

最后，如果你所在的目录与工作的目录不同，别忘了给出脚本文件的完整路径。例如，如下的系统命令行，运行自D:\other，假设Python在你的系统路径中，但要运行的文件存放在其他地方：

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

如果你的PATH没有包含Python的目录，并且Python和你的脚本文件都没有位于你所工作的目录中，那么，针对两者都使用完整的路径：

```
D:\other> C:\Python30\python c:\code\otherscript.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 script1.py`而不是`python script1`。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 script`）。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，`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print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，我们往往需要使用print语句来看程序文件的输出。如果没有看到如何输出，确保在你的文件中已经使用了“print”。然而，在交互式会话中是不需要print语句的，因为Python自动响应表达式的结果；这里的print无伤大雅，但确实是不必要的录入。

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Shell语言编写的csh或ksh程序一样。这样的脚本往往叫做可执行脚本。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（常常叫做“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...')          # + means concatenate for strings
```

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。就像之前所介绍的一样，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:\misc> 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。

注1： 介绍命令行时，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**相当于输入**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释并使用顶端的#!行，像UNIX那样，但是，Windows的DOS系统shell会完全忽略它。

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程序文件点击打开时Python自动注册为所运行的那个程序。正因如此，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，让我们继续使用前面编写的`script1.py`脚本，其内容如下：

```
# A first Python script
import sys                      # Load a library module
print(sys.platform)
print(2 ** 100)                 # Raise 2 to a power
x = 'Spam!'
print(x * 8)                    # String repetition
```

我们已经介绍了，总是可以从一个系统命令行来运行这个文件：

```
C:\misc> c:\python30\python script1.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，通过在开始菜单中选择“计算机”或者XP中的“我的电脑”，找到C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 Vista）。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。要运行这里的文件，直接点击`script1.py`的图标。

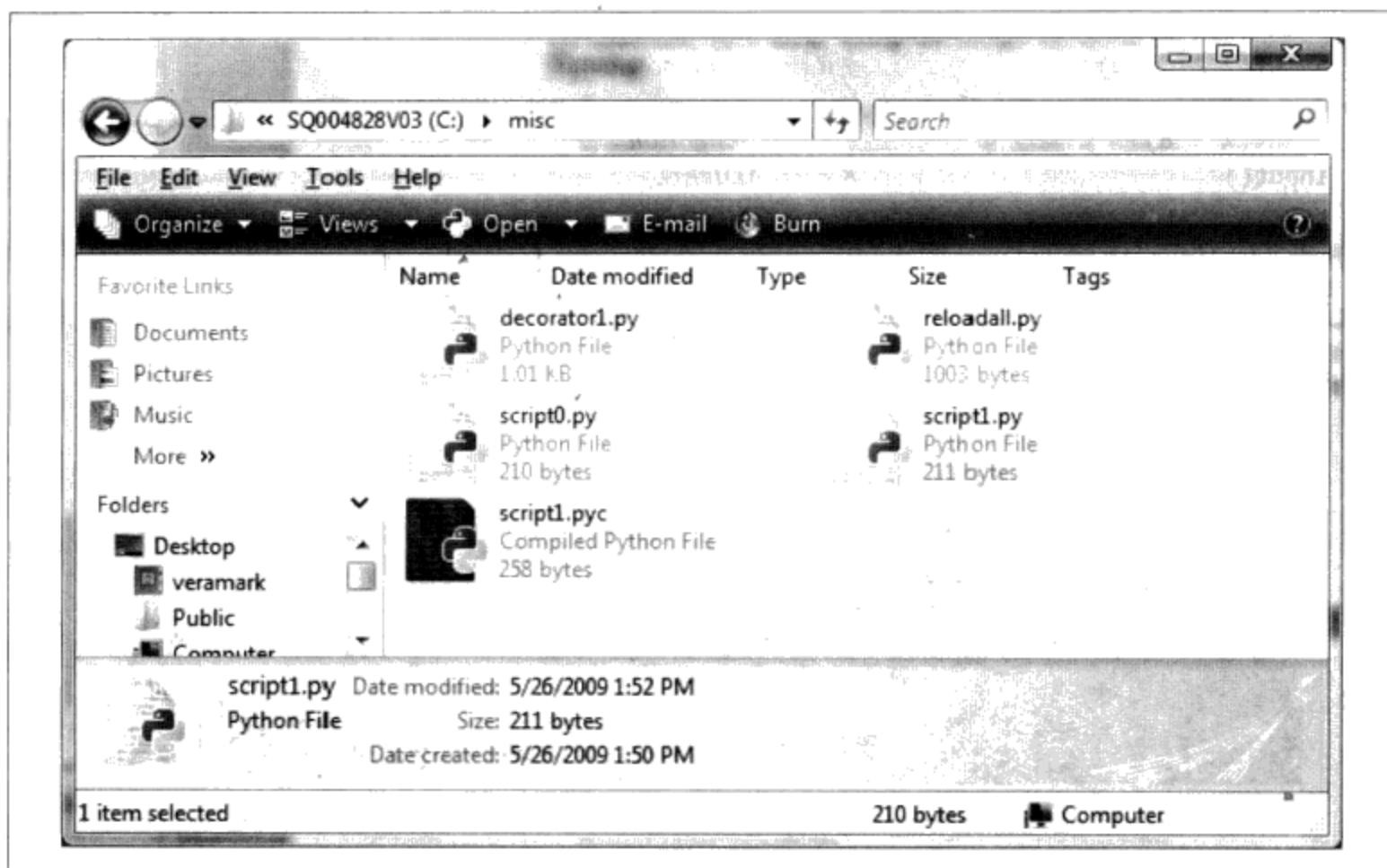


图3-1：在Windows中，Python程序在文件管理器窗口中显示为一个图标，并通过鼠标双击能够自动运行（尽管你采用这种办法也许不会看到打印的输出或错误的提示）

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。

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，脚本输出后暂停，可以简单地在脚本的最后添加内置input函数的一条调用语句（Python 2.6中的raw_input，参见前面的注释）。例如：

```
# A first Python script
import sys                                # Load a library module
```

```

print(sys.platform)
print(2 ** 100)           # Raise 2 to a power
x = 'Spam!'
print(x * 8)             # String repetition
input()                  # <== ADDED

```

一般来说，`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。

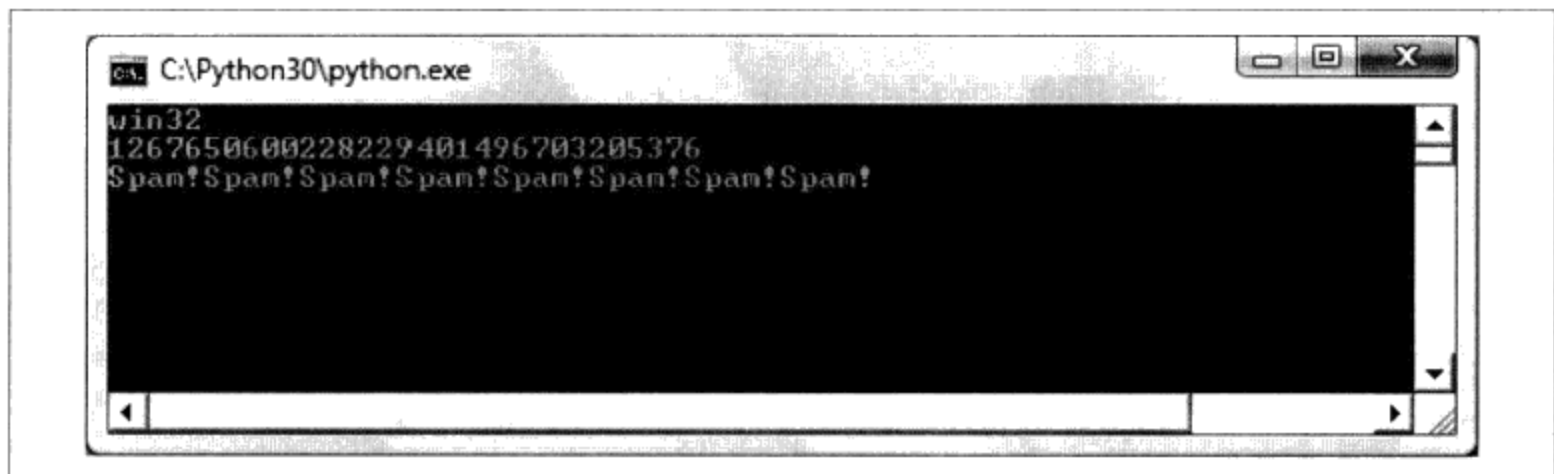


图3-2：当在Windows中点击程序图标时，如果在脚本的最后包含一个`input`调用，将会得到其输出的窗口。但是只需要在这种情况下这样做

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（除非你超出常理地热衷于按下计算机的Enter键）^{注2}。这听起来是显而易见的事情，但确是现实的类中的另一个常见的错误。

在我们继续学习之前，注意在输入时所使用的`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，`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，`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 = input()`] 。

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（这些用户界面构建自己的窗口），和各种其他技术一起使用，把打印完成的输出和错误保存到文件中。

- 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。

在后续章节中，我们将会以更复杂的方法使用`input`。例如，第10章我们将会在交互循环中使用它。

注意： 版本差异提示：如果你使用Python 2.6或者更早的版本，在这段代码中使用`raw_input()`而不要使用`input()`。在Python 3.0中，前者重新命名为后者。从技术上讲，Python 2.6也有一个`input`，但是，它对字符串求值，就好像它们是输入到一个脚本的程序代码一样，因此，该函数在这个环境中无效（一个空字符串会产生错误）。Python 3.0的`input`（以及Python 2.6的`raw_input()`）直接把输入的文本作为一个字符串返回，而不会求值。要在Python 3.0中模拟Python 2.6的`input`，使用`eval(input())`。

图标点击的其他限制

即使使用了`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`input`的调用也无济于事，因为早在调用`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。

正是由于这些限制，最好将点击图标看做是在程序调试之后或已经将其输出写入到一个文件之后，启动运行程序的一种方法。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。我们将会在第五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地探索。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（从系统命令行、从开始菜单或者在IDLE中），你可以运行之前创建的文件`script1.py`，通过简单的`import`来实现（确保删除了在上一节中添加的`input`行，或者你无缘无故地需要按Enter键）。

```
C:\misc> c:\python30\python
>>> import script1
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

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 script1
>>> import script1
```

这是有意设计的结果。导入是一个开销很大的操作，以至于每个文件、每个程序运行不能够重复多于一次。当你学习第21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。

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`imp`标准库模块中可用的`reload`函数（这个函数也是一个Python 2.6内置函数，但在Python 3.0中不是内置的）。

```
>>> from imp import reload          # Must load from module in 3.0
>>> reload(script1)
win32
65536
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
<module 'script1' from 'script1.py'>
>>>
```

这里的`from`语句直接从一个模块中复制出一个名字（稍后更详细地介绍）。`reload`函数载入并运行了文件最新版本的代码，如果已经在另一个窗口中修改并保存了它，那将反映出修改变化。

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`script1.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把模块名称传递给`reload`函数作为括号中的参数，并且这也是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16章介绍。

注意：版本差异提示：Python 3.0把`reload`内置函数移到了`imp`标准库模块中。它仍然像以前一样重载文件，但是，必须导入它才能使用。在Python 3.0中，运行`import imp`并使用`imp.reload(M)`，或者像这里所示的，运行`from imp import reload`并使用`reload(M)`。我们将在下一节介绍`import`和`from`语句，并且在本书稍后更加正式地讨论这些内容。

如果你在使用Python 2.6（或者是更常见的2.X），`reload`可以作为内置函数使用，因此，不需要导入。在Python 2.6中，`reload`可以以两种形式使用，内置函数或者模块函数，这有助于向Python 3.0的转换。换句话说，在Python 3.0中仍然可以使用重载，但是需要一行额外的代码来导入对`reload`的调用。

向Python 3.0迁移，可能部分动机是由一些众所周知的问题所引起的，这些问题包括我们将在下一节讨论的`reload`和`from`语句。简而言之，用一个`from`载入的名字不会通过一个`reload`直接更新，但是，用一条`import`语句访问的名字则会。如果你的名字似乎不会在一次重载后改变，尝试使用`import`和`module.attribute`名称引用。

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，这将在第五部分学到。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（就像一个模块）。

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。

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载入，并使用模块名后跟一个属性名来获取它：

```
% python                                # Start Python
>>> import myfile                       # Run file; load module as a whole
>>> print(myfile.title)                 # Use its attribute names: '.' to qualify
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（实际上是复制）变量名：

```
% python                                # Start Python
>>> from myfile import title            # Run file; copy its names
>>> print(title)                       # Use name directly: no need to qualify
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，from复制了模块的属性，以便属性能够成为接收者的直接变量。因此，能够直接以title（一个变量）引用导入字符串而不是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：

注3： 注意，import和from列出模块名时，都是使用myfile，没有.py后缀。到了第五部分，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shell命令行中，一定要记得加上后缀名，但是import语句中则不用。

```

a = 'dead'                # Define three attributes
b = 'parrot'              # Exported to other files
c = 'sketch'
print(a, b, c)            # Also used in this file

```

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复本。

```

% python
>>> import threenames          # Grab the whole module
dead parrot sketch
>>>
>>> threenames.b, threenames.c
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c    # Copy multiple names
>>> b, c
('parrot', 'sketch')

```

这里的结果打印在括号中，因为它们实际上是元组（本书的下一部分介绍的一种对象）；目前我们可以暂时忽略它们。

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。下面代码返回了一个Python字符串列表（我们将从下一章开始学习列表）：

```

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

```

我们在Python 3.0和Python 2.6中分别运行它，较早的Python可能返回较少的名字。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。

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，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，除非它显式地导入了那个文件，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。

注意： *import* VS *from*：我应该指出，*from*语句在某种意义上战胜了模块的名称空间分隔的目的，因为*from*把变量从一个文件复制到另一个文件，这可能导致在导入的文件中相同名称的变量被覆盖（并且，如果发生这种情况的话，不会为你给出警告）。这根本上会导致名称空间重叠到一起，至少在复制的变量上会重叠。

因此，有些人建议使用import而不是from。然而，我不建议这么做，不仅因为from更短，而且因为它传说中的问题在实际中几乎不是问题。此外，这是由你来控制的问题，可以在from中列出想要的变量；只要你理解它们将是要赋的值，这不会比编写赋值语句更危险，而赋值是你可能想要使用的另一功能。

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，而忽略了能够运行最新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：你需要记住是何时导入的，才能知道能不能够reload，你需要记住当调用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。此外，reload是不可传递的，重载一个模块的话只会重载该模块，而不能够重载该模块所导入的任何模块，因此，有时候必须reload多个文件。

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，包括本章前面的提示中所讨论的reload/from问题），从现在开始就要避免使用import和reload启动程序，这是一个好主意。例如，下一节所介绍的IDLE Run→Run Module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，并且总是运行代码的最新版本。系统shell命令行提供了类似的优点。如果使用这些技术的话，不需要使用reload。

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21章并学习搜索路径的模块。

现在，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4}。

也就是说，import和reload已经证明了是Python类中的一种常用测试技术，并且你可能也喜欢使用这种方法。然而，通常如果你发现自己碰壁了，那就停止继续碰壁。

使用exec运行模块文件

实际上，除了这里介绍的，还有更多的方法可以运行模块文件中保存的代码。例如，`exec(open('module.py').read())`内置函数调用，是从交互提示模式启动文件而不必导入以及随后的重载的一种方法。每次exec都运行文件的最新版本，而不需要随后的重载（`script1.py`保留我们在前面小节中一次重载它之后的样子）：

```
C:\misc> c:\python30\python
>>> exec(open('script1.py').read())
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam! Spam!

...change script1.py in a text edit window...

>>> exec(open('script1.py').read())
win32
4294967296
Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

exec调用有着类似于import的效果，但是，它从技术上不会导入模块，默认情况下，每次以这种方式调用exec的时候，它都重新运行文件，就好像我们把文件粘贴到了调用exec的地方。因此，exec不需要在文件修改后进行模块重载，它忽略了常规的模块导入逻辑。

缺点是，由于exec的工作机制就好像在调用它的地方粘贴了代码一样，和前面提到的

注4： 如果你不想一直等到第21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的文件，这个目录必须在`PYTHONPATH`中列出。想了解更多细节，请阅读第21章。

from一样，对于当前正在使用的变量有潜在的默认覆盖的可能。例如，我们的`script1.py`赋给了一个名为`x`的变量。如果这个名字已经在`exec`调用的地方使用了，那么这个名称的值将被覆盖。

```
>>> x = 999
>>> exec(open('script1.py').read())    # Code run in this namespace by default
...same output...
>>> x                                  # Its assignments can overwrite names here
'Spam!'
```

相反，基本的`import`语句每个进程只运行文件一次，并且它会把文件生成到一个单独的模块名称空间中，以便它的赋值不会改变你的作用域中的变量。为模块名称空间分隔所付出的代价是，在修改之后需要重载。

注意：版本差异提示：除了允许`exec(open('module.py'))`的形式，Python 2.6也包含一个`execfile('module.py')`内置函数，这两个函数都会自动读取文件的内容。这两种形式都等同于`exec(open('module.py').read())`的形式，后者更为复杂，但是在Python 2.6和Python 3.0中都可以运行。

遗憾的是，两种较简单的Python 2.6的形式，在Python 3.0中都不可用，这意味着我们必须理解文件及其读取方法，以便今天完全理解这一技术（唉，这似乎是针对Python 3.0的实用性的美学痛击）。实际上，Python 3.0中的`exec`形式需要如此多的录入，以至于最佳建议都是干脆不要使用它，通常启动文件的最好方式是输入系统shell命令行或者使用下一节所介绍的IDLE菜单选项。要了解关于Python 3.0的`exec`形式的更多内容，请参阅第9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、模块导入和`exec`调用来运行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许多开发任务^{注5}。

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。

注5： IDLE是IDE的一个官方变形，但是其实际上是为了纪念Monty Python的成员Eirc Idle而命名的。

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:\Python30\Lib\idlelib或者在Python 2.6中是C:\Python26\Lib\idlelib中的一个Python脚本^{注6}。

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。

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件）。

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。这有助于给代码中的组件一个更好的外观（甚至可以帮助你发现错误，例如，连续的字符串都是一种颜色）。

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改变了文件，Python将会提醒需要首先保存文件（这是编程中常常深陷其中的错误）。

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，窗口中部的“RESTART”后面的三行反映了在独立编辑窗口打开的script1.py脚本的执行情况。“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（如果IDLE已经在没使用用户代码子进程的情况下启动了，它将不会显示）。

注6： 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，也意味着你需要让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。



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)

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, 或使用复制粘贴的操作, 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，需要亲自小心地输入.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）。
- **你只需要重载交互地测试的模块。**像shell命令行一样，IDLE的Run→Run Module菜单选项一般只是运行顶层文件以及它导入的任何模块的最新版本。因此，Run→Run Module避免了常见的令人混淆的嵌套导入。你只需要重载那些将在IDLE中交互地导入和测试的模块。如果选择import和reload来替代Run→Run Module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（当然，没有人真的会用后一种方法，但是，它听上去比按下Enter键更高超）。
- **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，但是是一些程序仍然会发生GUI没有响应的情况。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。

IDLE还自动把目录修改为刚刚运行的一个文件的目录，并且将其目录添加到模块导入查找路径中，这是一项方便的功能，允许你在没有搜索路径设置的时候导入文件，但是，当你运行IDLE之外的文件的时候，该功能无法同样地工作。使用这样的功能没有问题，但是，别忘了它是IDLE行为，而不是Python行为。

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→Debugger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。它包含了对代码完成、语法突出显示、语法分析、重构、调试等功能的支持。其缺点就是需要安装较大的系统，并且对于某些功能可能需要共享扩展（经过一段时间后这可能会有所变化）。尽管如此，当你希望从IDLE升级时，Eclipse/PyDev这个组合是值得你注意的。

Komodo

作为一个全功能的Python（及其他语言的）开发环境GUI，Komodo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曳模式（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。

NetBeans IDE Python版

NetBeans是一款强大的开源开发环境GUI，针对Python开发者支持很多高级功能：代码完成、自动缩进和代码着色、编辑器提示、代码折叠、重构、调试、代码覆盖和测试、项目等等。它可以用来开发CPython和Jython代码。和Eclipse一样，要获得超越IDLE GUI的那些功能，NetBeans也需要一些安装步骤，但是，很多人认为值得这么做。请搜索Web以查找最新的信息和链接。

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COM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挂起的现象）。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

概括计算我所知道的IDE（例如，Wing IDE、PythonCard）有近十个，还有更多的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）。

其他启动选项

到现在为止，我们已经看到了如何运行代码交互地输入，以及如何以各种不同的方式启动保存在文件中的代码：在系统命令行中运行、import和exec、使用IDLE这样的GUI等。这基本上包括了本书中提到的所有情况。然而，还有运行Python代码的其他方法，其中大多数都有专门或有限的用途。下一小节我们将简要介绍这些方法。

嵌入式调用

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）。用户可以提供或修改Python代码来定制这种系统。由于Python代码是解释性的，不必重新编译整个系统以融入修改（参见第2章更详细地了解Python代码是如何运行的）。

在这种方式下，当使用Jython系统的时候，运行你的代码的封闭的系统可能是使用C、C++或者甚至Java编写的。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();                               // This is C, not Python
PyRun_SimpleString("x = 'brave ' + 'sir robin'"); // But it runs Python code
```

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，你也许会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7}。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。

冻结二进制的可执行性

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。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，你也许可以通过拖曳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单独进程中的其他Python程序来启动Python程序（例如，`os.popen`和`os.system`），并且像Web这样的较大的环境也可能产生Python脚本（例如一个Web页面可能调用服务器上的一个脚本）。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，很有可能时不时地会有新的程序启动选择出现。

注7： 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。

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 中拖曳图标，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）。如果我知道未来会是怎样，我可能会去找一个股票经纪人谈谈，而不是在这里写下这些话。

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。

调试Python代码

一般的，我们的读者或学生不会在他们的代码里包含Bug（在此置之一笑吧），但为了极少数可能遭遇不幸的朋友，这里快速介绍现实世界的Python程序员调试代码时候常用的一些策略：

- **什么也不做。**我这么讲，并不是说Python程序员不要调试自己的代码，但是，当你在一个Python程序中犯错的时候，会得到一条非常有用而容易读懂的出错消息（如果你已经有了一些错误的话，很快会见到这些消息）。如果你已经了解Python了，特别是如果你已经熟悉自己的代码了，那么，这么做通常就够了：阅读出错消息，并修改标记的行和文件。对于很多人来说，这就是Python中的调试。但是，对于你没有编写过的那些大型系统来说，这并不总是理想的做法。
- **插入print语句。**可能Python程序员调试自己的代码的主要方式（以及我调试Python代码的方式），就是插入print语句并再次运行。由于Python在修改后

立即运行，这通常是获取比出错消息所提供的更多的信息的一种快捷方式。`print`语句不必很复杂，一条简单的“I am here”或变量值的显示，通常就能够提供你所需的足够的背景信息。只是别忘了，在发布你的代码之前，删除掉或注释掉（如，在前面添加一个#）用来调试的`print`。

- **使用IDE GUI调试器。**对于你没有编写过的较大的系统，以及对于那些想要更详细地跟踪代码的初学者，大多数Python开发GUI都有某种指向点击调试器。IDLE也有一个调试器，但是，它在实际中似乎并不常用，可能是因为它没有命令行，或者可能是因为添加`print`语句通常比设置一个GUI调试会话要快。要了解更多内容，请查阅IDLE的帮助，或者直接自己尝试；其基本界面如本章前面的“高级IDLE 工具”部分所述。其他的IDE，如Eclipse、NetBeans、Komodo和WingIDE也都提供了高级的指向点击调试器，如果你使用这些IDE，请查阅它们的文档。
- **使用pdb命令行调试器。**为了实现最终控制，Python附带了一个名为`pdb`的源代码调试器，可以作为Python的标准库中的一个模块使用。在`pdb`中，我们输入命令来一行一行地步进执行，显示变量，设置和清除断点，继续执行到一个断点或错误，等等。通过导入可以交互地启动`pdb`，或者作为一个顶层脚本启动。不管哪种方式，由于我们可以输入命令来控制会话，它都提供了强大的调试工具。`pdb`还包含了一个`postmortem`函数，可以在异常发生后执行它，从而获取发生错误时的信息。参见Python的库手册以及本书第35章了解关于`pdb`的更多细节。
- **其他选项。**如果有更具体的调试需求，你可以在开源领域找到其他的工具，包括支持多线程程序、嵌入式代码和进程附件的工具。例如，Winpdb系统是一个独立的调试器，具有高级的调试支持、跨平台的GUI和控制台界面。

随着我们开始编写较大的脚本，这些选项将变得更加重要。然而，可能关于调试最好的消息是，在Python中检测出并报告错误，而不是默默地传递错误或最终导致系统崩溃。实际上，错误本身是一种定义良好的机制，叫做异常，我们可以捕获并处理它们（更多关于异常的讨论在本书第三部分进行）。当然，犯错并不好玩，但是，正如某人所说的：当进行调试意味着最终得出一个十六进制计算器和仔细钻研成堆的内存转储输出的时候，有了Python的调试器支持，所犯的 error 不会像没有调试器的情况下那样令人痛苦不堪。

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、`exec`调用以及像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。

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。

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. 指出运行保存在一个脚本文件中的代码的四种或更多的方法。
4. 指出在Windows下点击文件图标运行脚本的两个缺点。
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:\Python30\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. 一个脚本（实际上是模块）文件中的代码可以通过系统命令行、文件鼠标点击、导

入和重载、`exec`内置函数以及像IDLE的Run→Run Module菜单选项这样的IDE GUI选取来运行。在UNIX上，还可以使用`#!`技巧来运行，并且一些平台还支持更为专用的启动技术（例如，拖曳）。此外，一些文本编辑器有运行Python代码的独特方式，一些Python程序作为独立的“冻结二进制”可执行文件提供；并且一些系统在嵌入式模式下使用Python代码，其中代码由C、C++或Java等语言编写的一个封闭程序自动运行。后几种技术通常用来提供一个用户定制层级。

4. 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）。
5. 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。
6. 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。
7. 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。
8. 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，文件必须明确地导入其他的文件，才能使用这些文件的变量名。

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径，或者增加Python可执行文件的路径至PATH环境变量。如果想要的话，你可以在.cshrc或.kshrc文件中设置PATH，使Python在UNIX系统中永久可用；在Windows中，使用setup.bat、autoexec.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. 错误和调试。在Python交互命令行中，试着输入数学表达式和赋值。首先输入2 ** 500和1 / 0，并且像我们在本章中所做的那样引用一个未定义的变量名。发生了什么？

你也许还不知道，但是你正在做的是异常处理（一个将会在第七部分探索的主题）。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。

异常总是和Python中的调试概念密切相关的。当你第一次开始的时候，Python关于异常的默认错误消息总是为你的错误处理提供尽可能多的支持，它们给出错误的原因，并且在代码中显示出错误发生时所执行的行。要了解关于调试的更多内容，参见本章的“调试Python代码”部分。

6. 中断。在Python命令行中，输入：

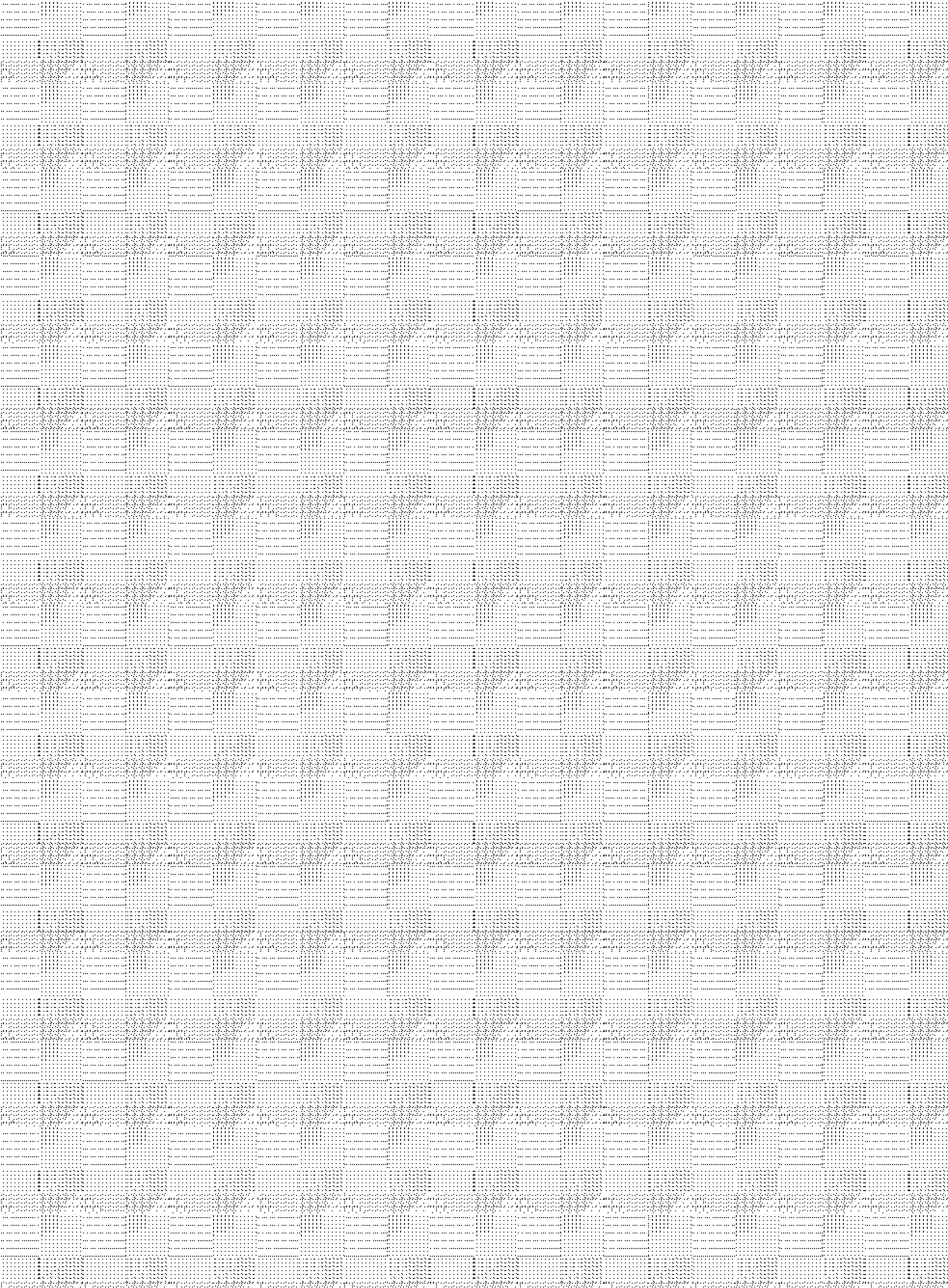
```
L = [1, 2]           # Make a 2-item list
L.append(L)          # Append L as a single item to itself
L                    # Print L
```

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？

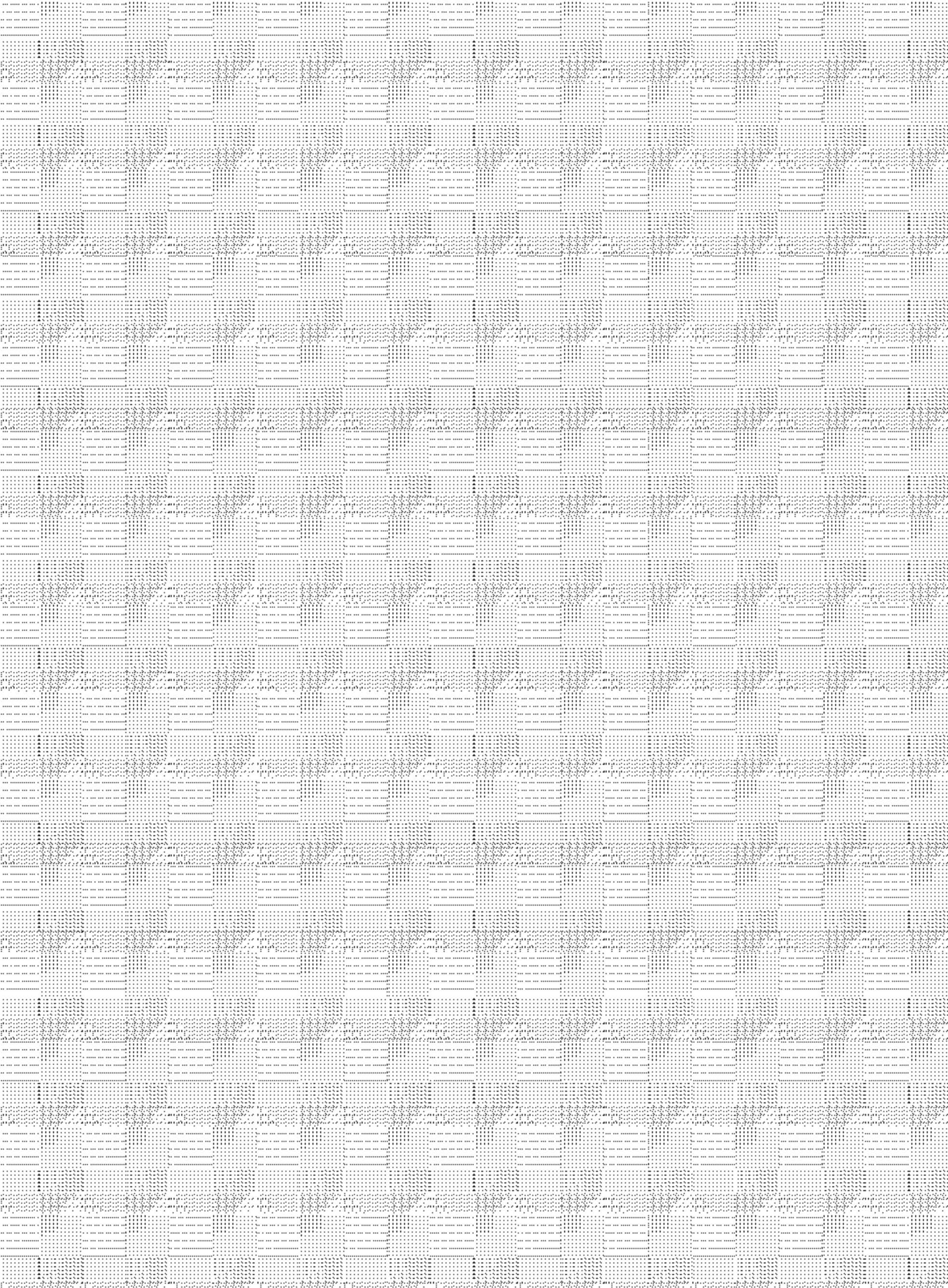
警告： 如果你有一个比1.5.1版更老的Python，在运行这个测试之前，保证你的机器能够通过中断组合键Ctrl-C中止一个程序，不然的话你得等很长时间。

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15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。



类型和运算



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。在本书这一部分中，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，让我们先来简单地说明这一章如何符合Python全景。从更具体的视角来看，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。本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。

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（或者叫做数据结构）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（且容易出错），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Python内置对象类型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，与我们从零开始所创建的工具相比，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。无论你是否实现新的对象类型，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（literal）所使用到的语法，也就是能

够生成这些对象的表达式^{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字和文本的值，而文件则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

对象类型	例子 常量/创建
数字	1234, 3.1415, 3+4j,Decimal, Fraction
字符串	'spam', "guido's",b'a\xolc'
列表	[1,[2,'three'],4]
字典	{'food':'spam','taste':'yum'}
元组	(1,'spam',4,'U')
文件	myfile=open('eggs','r')
集合	set('abc'), {'a', 'b', 'c'}
其他类型	类型、None、布尔型
编程单元类型	函数、模块、类（参见第四部分、第五部分和第六部分）
与实现相关的类型	编译的代码堆栈跟踪（参见第四部分和第七部分）

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。

我们将在本书稍后的部分介绍，像函数、模块和类这样的编程单元在Python中也是对象，它们由def、class、import和lambda这样的语句和表达式创建，并且可以在脚本间自由地传递，存储在其他对象中等。Python还提供了一组与实现相关的类型，例如编译过的代码对象，它们通常更多地关系到工具生成器而不是应用程序开发者；本书后续部分也将讨论它们。

我们通常把表4-1中的对象类型称作是核心数据类型，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'
```

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）。

从技术上讲，你正在运行一个常量表达式，这个表达式生成并返回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，大括号中的表达式会建立一个字典等。尽管这样，就像我们看到的那样，Python中没有类型声明，运行的表达式的语法决定了创建和使用的对象的类型。事实上，在Python语言中，诸如表4-1中的那些对象生成表达式就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适合该类型的有效的操作）。

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。

我们将会在后续章节学习表4-1中的每一种对象类型。在深入介绍细节之前，我们迅速地浏览在实际应用中的Python的核心对象。本章的余下部分将提供一些操作，而在本章以后的章节我们将会更深入地学习这些操作。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会比较眼熟。即使你没有编程经验，数字也是比较直接的。Python的核心对象集合包括常规的类型：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（有虚部的复数、固定精度的十进制数、带分子和分母的有理分数以及集合等）。

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（*）表示乘法，双星号（**）表示乘方。

```
>>> 123 + 222                                # Integer addition
345
>>> 1.5 * 4                                   # Floating-point multiplication
6.0
>>> 2 ** 100                                  # 2 to the power 100
```


1267650600228229401496703205376

注意这里的最后一个结果：当需要的时候，Python 3.0的整数类型会自动提供额外的精度，以用于较大的数值（在Python 2.6中，一个单独的长整型会以类似的方式来处理那些对于普通整型来说太大的数值）。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，你就得等一会儿了！）。

```
>>> len(str(2 ** 1000000))           # How many digits in a really BIG number?
301030
```

一旦你开始接触浮点数，很可能会遇到一些乍看上去有些奇怪的事情：

```
>>> 3.1415 * 2                        # repr: as code
6.283000000000000004
>>> print(3.1415 * 2)                 # str: user-friendly
6.283
```

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，第一种形式看做是对象的代码形式`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。

除了表达式外，和Python一起分发的还有一些常用的数学模块，模块只不过是我们导入以供使用的一些额外工具包。

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

`math`模块包括更高级的数学工具，如函数，而`random`模块可以作为随机数字的生成器和随机选择器（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、有理数、集合和布尔值，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。想要更好地利用内置类型的话，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，字符串是单个字符的字符串的序列，其他类型的序列还包括列表和元组（稍后介绍）。

序列的操作

作为序列，字符串支持假设其中各个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。

```
>>> 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。

注意我们在这里是如何把字符串赋给一个名为S的变量的。我们随后将详细介绍这是如何做到的（特别是在第6章中），但是，Python变量不需要提前声明。当给一个变量赋值的时候就创建了它，可能赋的是任何类型的对象，并且当变量出现在一个表达式中的时候，就会用其值替换它。在使用变量的值之前必须对其赋值。我们需要把一个对象赋给一个变量以便保存它供随后使用，要学习本章内容，知道这一点就足够了。

在Python中，我们能够反向索引，从最后一个开始（正向索引是从左边开始计算，反向索引是从右边开始计算）。

```
>>> 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，负的索引号会简单地与字符串的长度相加，因此，以下两个操作是等效的（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，而不仅仅是使用数字常量——只要Python需要一个值，我们可以使用一个常量、一个变量或任意表达式。Python的语法在这方面是完全通用的。

除了简单地从位置进行索引，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'
```

也许认识分片的最简单的办法就是把它们看做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中从偏移量为I，直到但不包括偏移量为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。

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。这引入了一些常用法的变体：

```
>>> 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'
```

注意负偏移量如何用作分片的边界，在上面最后一个操作中如何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（将两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。正如将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——在创建后不能就地改变。例如，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
'zspam'
```

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），去掉字符串后的空格字符。

```
>>> line = 'aaa,bbb,cccc,dd'
>>> line.split(',')              # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'cccc', 'dd']
>>> S = 'spam'
>>> S.upper()                   # Upper- and lowercase conversions
'SPAM'

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

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

字符串还支持一个叫做格式化的高级替代操作，可以以一个表达式的形式（最初的）和一个字符串方法调用（Python 2.6和Python 3.0中新引入的）形式使用：

```
>>> '%s, eggs, and %s' % ('spam', 'SPAM!')          # Formatting expression (all)
'spam, eggs, and SPAM!'

>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!')    # Formatting method (2.6, 3.0)
'spam, eggs, and SPAM!'
```

注意：尽管序列操作是通用的，但方法不通用（虽然某些类型共享某些方法名，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，你会更顺利地从此类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。假设`S`是一个字符串，这里是其在Python 3.0中的属性（Python 2.6中略有不同）：

```
>>> dir(S)
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', '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的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。

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

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

replace(...)
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new. If the optional argument count is
    given, only the first count occurrences are replaced.
```

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)`]，但是你将看到的也许比你需要的更多（将得到关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，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

到目前为止，我们学习了字符串对象的序列操作方法和类型特定的方法。Python还提供了各种编写字符串的方法，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, a binary zero byte, does not terminate 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）。它也允许在三个引号（单引号或双引号）中包括多行字符串常量。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换行符。这是一个微妙的语法上的便捷方式，但是在Python脚本中嵌入像HTML或XML这样的内容时，这是很方便的。

```
>>> msg = """ aaaaaaaaaaaaaa
bbb' 'bbbbbbbbbb'bbbbbb'bbbb
cccccccccccccc"""
>>> msg
'\naaaaaaaaaaaaaa\nbbb'\''\''bbbbbbbbbb'bbbbbb'\bbbb\ncccccccccccccc'
```

Python也支持原始（raw）字符串常量，即去掉反斜线转义机制（这样的字符串常量是以字母“r”开头的）。Python还支持Unicode字符串形式从而支持国际化。在Python 3.0中，基本的`str`字符串类型也处理Unicode（这是有意义的，因为ASCII文本是一种简单的Unicode），并且用`bytes`类型表示原始字节字符串；在Python 2.6中，Unicode是一种单独的类型，`str`处理8位字符串和二进制数据。在Python 3.0中，文件也改变为返回和

接受`str`，从而处理二进制数据的文本和字节。我们将会在后续章节学到这些特殊的字符串。

模式匹配

在继续学习之前，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 '
```

这个例子的目的是搜索子字符串，这个子字符串以“Hello,”开始，后面跟着零个或多个制表符或空格，接着有任意字符并将其保存至匹配的`group`中，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，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，确实能够修改列表的大小。

序列操作

由于列表是序列的一种，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
```

我们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，列表没有固定大小，也就是说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）移除给定偏移量的一项，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就地改变列表对象，而不是创建一个新的列表：

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> 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。超出列表末尾之外的索引总是会导致错误，对列表末尾范围之外赋值也是如此：

```
>>> 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，而是会提示错误。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，并可以多个层次进行嵌套（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 × 3 matrix, as nested lists
          [4, 5, 6],        # Code can span lines if bracketed
          [7, 8, 9]]

>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这里，我们编写了一个包含3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。

```
>>> 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入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。它是一种通过对序列中的每一项运行一个表达式来创建一个新列表的方法，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），并且由使用了同一个变量名的（这里是`row`）表达式和循环结构组成。之前的这个列表解析表达基本上就是它字面上所讲的：“把矩阵M的每个`row`中的`row[1]`，放在一个新的列表中”。其结果就是一个包含了矩阵的第二列的新列表。

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]
```

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坐标的一个硬编码列表和一个字符串：

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被看成把Python变成对等于MatLab系统的免费版本，而且功能更强大。此外，诸如NASA、Los Alamos以及JPMorgan Chase等机构都使用这个工具以从事科学和金融工作。上网搜索可以了解更多细节。

```
>>> 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']
```

列表解析以及相关的内容函数map和filter比较复杂，本书不过多讲述了。这里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，在实际应用中比较方便，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。

然而，作为一个预览，我们会发现在Python的最近版本中，括号中的解析语法也可以用来创建产生所需结果的生成器（例如，内置的sum函数，按一种顺序汇总各项）：

```
>>> G = (sum(row) for row in M)                   # Create a generator of row sums
>>> next(G)
6
>>> next(G)                                       # Run the iteration protocol
15
```

内置函数map可以做类似的事情，产生对各项运行一个函数的结果。在Python 3.0中，将其包装到列表中，会使其返回所有值：

```
>>> list(map(sum, M))                             # Map sum over items in M
[6, 15, 24]
```

在Python 3.0中，解析语法也可以用来创建集合和字典：

```
>>> {sum(row) for row in M}                       # Create a set of row sums
{24, 6, 15}

>>> {i : sum(M[i]) for i in range(3)}              # Creates key/value table of row sums
{0: 6, 1: 15, 2: 24}
```

实际上，在Python 3.0中，列表、集合和字典都可以用解析来创建：

```
>>> [ord(x) for x in 'spaam']                     # List of character ordinals
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaam'}                     # Sets remove duplicates
{112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'}                  # Dictionary keys are unique
{'a': 97, 'p': 112, 's': 115, 'm': 109}
```

要理解生成器、集合和字典这样的对象，我们必须继续学习。

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（mapping）。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——可以就地改变，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。与列表中禁止边界外的赋值不同，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，如描述某人的记录中的名字字段。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

在上面的例子中，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化信息。

```
>>> 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的值从而支持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，而不是列表的偏移量。

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

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

>>> 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，将会很枯燥并会使用更多的代码——我们将不得不去事先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具有一种叫做垃圾收集的特性，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。

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，我们确实需要强调某种顺序的时候，应该怎样做呢？一个常用的解决办法就是通过字典的keys方法收集一个键的列表，使用列表的sort方法进行排序，然后使用Python的for循环逐个进行显示结果（正如第3章所介绍的，确保在循环的代码下面两次按下Enter键，交互提示模式中的一个空行意味着“执行”，某些接口中提示符是“...”）：

```
>>> Ks = list(D.keys())                                # Unordered keys list
>>> Ks                                                  # A list in 2.6, "view" in 3.0: use list()
['a', 'c', 'b']

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

>>> for key in Ks:                                     # Iterate though sorted keys
    print(key, '=>', D[key])                           # <== press Enter twice here
a => 1
b => 2
c => 3
```

注3： 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保存Python原生对象的简单方式），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。

这是一个有三个步骤的过程，然而，就像我们将会在后文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内置函数可以一步完成。sorted调用返回结果并对各种对象类型进行排序，在这个例子中，自动对字典的键排序：

```
>>> 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一种方法。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。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
```

Python的while循环是一种更为常见的排序循环工具，它不仅限于遍历序列：

```
>>> x = 4
>>> while x > 0:
    print('spam!' * x)
    x -= 1

spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

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（这是Python中无处不在的一个概念，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。如果一个对象在响应next之前先用一个对象对iter内置函数做出响应，那么它属于后一种情况。我们在前面所见到的生成器解析表达式就是这样的对象。

本书稍后将会详细介绍迭代协议。现在记住，从左到右地扫描一个对象的每个Python工具都使用迭代协议。这就是为什么前面一节所介绍的sorted调用直接工作于字典之上，我们不必调用keys方法来得到一个序列，因为字典是可选代的对象，可以用一个next返回后续的键。

这也意味着像下面这样的任何列表解析表达式都可以计算一系列数字的平方：

```
>>> 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]:
    squares.append(x ** 2)
>>> squares
[1, 4, 9, 16, 25]
```

This is what a list comprehension does
Both run the iteration protocol internally

尽管这样，列表解析和相关的函数编程工具，如map和filter，通常运行得比for循环快（也许快了两倍）：这是对有大数据集合的程序有重大影响特性之一。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，在程序可以工作，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 a nonexistent key 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。`in`关系表达式允许我们查询字典中一个键是否存在，并可以通过使用Python的`if`语句对结果进行分支处理（就像`for`一样，确保在这里两次按下Enter键来交互地运行`if`）：

```

>>> 'f' in D
False

>>> if not 'f' in D:
    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`语句也可以有`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：`get`方法（带有一个默认值的条件索引）、Python 2.X的`has_key`方法（在Python 3.0中不可用）、`try`语句（一个捕获异常并从异常中恢复的工具，我们将在第10章中介绍），以及`if/else`表达式（实质上是挤在一行中的一条`if`语句）。下面是一些例子：

```

>>> value = D.get('x', 0)                        # Index but with a default
>>> value
0
>>> value = D['x'] if 'x' in D else 0           # if/else expression form
>>> value
0

```

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。

元组

元组对象 (tuple, 发音为 “toople” 或 “tuhple”) 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。从语法上讲, 它们编写在圆括号中而不是方括号中, 它们支持任意类型、任意嵌套以及常见的序列操作:

```
>>> 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
```

在Python 3.0中, 元组还有两个专有的可调用方法, 但它的专有方法不像列表所拥有的那么多:

```
>>> T.index(4)                     # Tuple methods: 4 appears at offset 3
3
>>> T.count(4)                    # 4 appears once
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
```

与列表和字典一样, 元组支持混合的类型和嵌套, 但是不能增长或缩短, 因为它们是不可变的:

```
>>> T = ('spam', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

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, 元组在实际中往往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, 它可能在任何地方改变; 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。现在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。要创建一个文件对象，需调用内置的`open`函数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，创建一个文本输出文件，可以传递其文件名以及'`w`'处理模式字符串以写数据：

```
>>> f = open('data.txt', 'w')           # Make a new file in output mode
>>> f.write('Hello\n')                   # Write strings of bytes to it
6
>>> f.write('world\n')                   # Returns number of bytes written in Python 3.0
6
>>> f.close()                           # Close to flush output buffers to disk
```

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字符串，并显示它。对脚本而言，文件的内容总是字符串，无论文件包含的数据是什么类型：

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

>>> print(text)                           # print interprets control characters
Hello
world

>>> text.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（`seek`移动到一个新的文件位置）。我们在本书后面会看到，如今读取一个文件的最佳方式就是根本不读它，文件提供了一个迭代器（`iterator`），它在`for`循环或其他环境中自动地一行一行地读取。

我们将在本书的后面看到文件方法的一个完整列表，但是，如果现在想要快速预览一下，在任何打开的文件上运行一个`dir`调用并且在返回的任何方法名上调用一个`help`：

```
>>> dir(f)
[ ...many names omitted...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
'writelines']

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

在本书后面，我们还将看到Python 3.0中的文件在文本和二进制数据之间划出了一条清晰的界限。文本文件把内容显示为字符串，并且自动执行Unicode编码和解码；而二进制文件把内容显示为一个特定的字节字符串类型，并且允许你不修改地访问文件内容：

```
>>> data = open('data.bin', 'rb').read()           # Open binary file
>>> data                                           # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]
b'spam'
```

如果你只处理ASCII文本的话，通常不需要关心这一区别，尽管如此，如果你处理国际化的应用程序或者面向字节的数据，Python 3.0的字符串和文件是很有用的。

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还有额外的类文件工具：管道、先进先出队列（FIFO）、套接字、通过键访问文件、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，描述符文件（descriptor file）支持文件锁定和其他的底层工具，而套接字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。

其他核心类型

到目前为止除了我们看到的核心类型外，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，它不是映射也不是序列，相反，它们是唯一的不可变的对象的无序集合。集合可以通过调用内置set函数而创建，或者使用Python 3.0中新的集合常量和表达式创建，并且它支持一般的数学集合操作（Python 3.0中新的用于集合常量的{...}语法是有意义的，因为集合更像是一个无值的字典的键）：

```
>>> X = set('spam')                               # Make a set out of a sequence in 2.6 and 3.0
>>> Y = {'h', 'a', 'm'}                           # Make a set with new 3.0 set literals
>>> X, Y
```



```

({ 'a', 'p', 's', 'm' }, { 'a', 'h', 'm' })

>>> X & Y                                # Intersection
{ 'a', 'm' }

>>> X | Y                                # Union
{ 'a', 'p', 's', 'h', 'm' }

>>> X - Y                                # Difference
{ 'p', 's' }

>>> {x ** 2 for x in [1, 2, 3, 4]}        # Set comprehensions in 3.0
{16, 1, 4, 9}

```

此外，Python最近添加了一些新的数值类型：十进制数（固定精度浮点数）和分数（有一个分子和一个分母的有理数）。它们都用来解决浮点数学的局限性和内在的不精确性：

```

>>> 1 / 3                                # Floating-point (use .0 in Python 2.6)
0.33333333333333331
>>> (2/3) + (1/2)
1.1666666666666665

>>> import decimal                       # Decimals: fixed precision
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')

>>> decimal.getcontext().prec = 2
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')

>>> from fractions import Fraction        # Fractions: numerator+denominator
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)

```

Python最近还添加了布尔值（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（它通常用来初始化名字和对象）：

```

>>> 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, None, None, None, None, None, None, None, ...a list of 100 Nones...]
```

如何破坏代码的灵活性

本书稍后将把所有的这些对象进行介绍，但是还有一点值得注意。内置函数`type`返回的类型对象是赋给该类型的另一个对象的一个对象，其结果在Python 3.0中略有不同，因为类型已经完全和类结合起来了（我们将在本书第五部分的新式类部分中介绍这些）。假设`L`仍然是前面小节中的那个列表：

```
# In Python 2.6:

>>> type(L)
<type 'list'>
>>> type(type(L))
<type 'type'>

# Types: type of L is list type object
# Even types are objects

# In Python 3.0:

>>> type(L)
<class 'list'>
>>> type(type(L))
<class 'type'>

# 3.0: types are classes, and vice versa
# See Chapter 31 for more on class types
```

除了允许交互地探究对象，这个函数的实际应用是，允许编写代码来检查它所处理的对象的类型。实际上，在Python脚本中至少有3种方法可做到这点：

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

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

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

yes
```

现在本书已经介绍了所有的类型检验的方法，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（这也是一个有经验的C程序员刚开始使用Python时的一个标志）。在本书后面，当我们开始编写函数这样较大的代码单元的时候，才会澄清其原因，但这是一个（可能是唯一的）核心Python概念。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，这是必要的），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
    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（有时候叫做状态信息），也有两个小的行为编写为函数（通常叫做方法）的形式。就像函数那样去调用类，会生成我们新类型的实例，并且类的方法调用时，类的方法自动获取被处理的实例（其中的self参数）：

```
>>> bob = Worker('Bob Smith', 50000)      # Make two instances
>>> sue = Worker('Sue Jones', 60000)      # Each has name and pay attrs
>>> 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
```

隐含的“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，你可以阅读第六部分的章节。

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东西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中的类型有的是与程序执行相关的对象（如函数、模块、类和编译过的代码），这些我们将在后面学习；有的是由导入的模块函数实现的，而不是语言语法。后者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。

本章小结

到此，我们就完成了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），以及可以作为方法调用的特定类型操作（例如，字符串分隔和列表增加）。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，比如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，首先让我们进行另一个测验来复习吧。

本章习题

未来的几章将会探索本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、文件和集合一般被认为是核心对象（数据类型）。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、浮点数、复数、分数和十进制数）和多种字符串类型（Python 2.X中的一般字符串和Unicode字符串，以及Python 3.X中的文本字符串和字节字符串）。
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，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，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。尽管无法就地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”，表示将键与相关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。
6. “多态”意味着一个操作符（如+）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，使代码自动适用于多种类型。

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的数字类型。

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，也许会很熟悉。它能够保持记录你的银行余额、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以及处理数字的表达式。此外，Python为更高级的工作提供了很多高级数字编程支持和对象。Python数字类型的完整工具包括：

- 整数和浮点数
- 复数

- 固定精度的十进制数
- 有理分数
- 集合
- 布尔类型
- 无穷的整数精度
- 各种数字内置函数和模块

本章从基本的数字开始，然后介绍这个列表中的其他工具。然而，在开始介绍编程之前，下面两个小节首先概述一下如何在脚本中编写和处理数字。

数字常量

在基本类型中，Python提供了：整数（正整数和负整数）和浮点数（带有小数部分的数字）。Python还允许我们使用十六进制、八进制和二进制常量来表示整数，提供一个复数类型，并且允许整数具有无穷的精度（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数字类型在程序中的显示方式（作为常量）。

表 5-1：基本数字常量

数字	常量
1234, -24, 0, 999999999999999	整数（无穷大小）
1.23, 1., 3.14e-10, 4E210, 4.0e+210	浮点数
0177, 0x9ff, 0b101010	Python 2.6中的八进制、十六进制和二进制常量
0o177, 0x9ff, 0b101010	Python 3.0中的八进制、十六进制和二进制常量
3+4j, 3.0+4.0j, 3j	复数常量

一般来说，Python的数字类型是很容易使用的，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。浮点数就像C语言中的“双精度”一样实现，因此，其精度与用来构建Python解释器的C编译器所给定的双精度一样。

Python 2.6中的整数：一般整数和长整数

Python 2.6中有两种整数类型：一般整数（32位）和长整数（无穷精度），并且一

个整数可以以l或L结尾，从而强迫其成为长整数。由于当整数的值超过32位的时候会自动转换为长整数，我们不需要自己输入字母L，当需要额外的精度的时候，Python会自动地转换为长整数。

Python 3.0中的整数：一个单独的类型

在Python 3.0中，一般整数和长整数类型已经合二为一了，只有整数这一种，它自动地支持Python 2.6的单独的长整数类型所拥有的无穷精度。因此，整数在程序中不再用末尾的l或L表示，并且整数也不再会显示出这个字符。除此之外，这一修改并没有影响到大多数程序，除非你确实进行类型测试来检测Python 2.6的长整数。

十六进制数、八进制和二进制常量

整数可以编写为十进制（以10为基数）、十六进制（以16为基数）、八进制（以8为基数）和二进制（以2为基数）形式。十六进制数以0x或0X开头，后面接十六进制的数字0~9和A~F。十六进制的数字编写成大写或小写都可以。八进制数常量以数字0o或0O开头（0和小写或大写的字母“o”），后面接着数字0~7构成的字符串。在Python 2.6及更早的版本中，八进制常量也可以写成前面只有一个0的形式，但在Python 3.0中不能这样（这种最初的八进制形式太容易与十进制数混淆，因此用新的0o的形式替代了）。Python 2.6和Python 3.0中的新的二进制常量，以0b或0B开头，后面跟着二进制数字（0~1）。

注意所有这些常量在程序代码中都产生一个整数对象，它们仅仅是特定值的不同语法表示而已。内置函数hex(I)、oct(I)和bin(I)把一个整数转换为这3种进制表示的字符串，并且int(str, base)根据每个给定的进制把一个运行时字符串转换为一个整数。

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。也可以通过内置函数complex(real, imag)来创建复数。

编写其他的数字类型

我们将在本章后面看到，还有表5-1中所没有包含的其他的、更高级的数字类型。其中的一些通过调用导入的模块中的函数来创建（例如，十进制数和分数），其他的一些拥有它们自己的常量语法（例如，集合）。

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、round、int、hex、bin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。

尽管数字主要是通过表达式、内置函数和模块来处理，它们如今也拥有很多特定于类型的方法，我们也将在本章中介绍这些。例如，浮点数拥有一个as_integer_ratio方法，它对于分数数字类型很有用；还有一个is_integer方法可以测试数字是否是一个整数。整数有各种各样的属性，包括一个将要在未来的Python 3.1中发布的新的bit_length方法，它给出表示对象的值所必需的位数。此外，集合既像一些集合一样也像一些数字一样，它也支持这两者的方法和表达式。

表达式是大多数数字类型最基本的工具，我们接下来将介绍它。

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）。如果你曾经使用过其他语言的话，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。

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
lambda args: expression	生成匿名函数
x if y else z	三元选择表达式
x or y	逻辑或（只有x为假，才会计算y）

表5-2: Python表达式操作符及程序 (续)

操作符	描述
<code>x and y</code>	逻辑与 (只有x为真, 才会计算y)
<code>not x</code>	逻辑非
<code>x in y, x not in y</code>	成员关系 (可迭代对象、集合)
<code>x is y, x is not y</code>	对象实体测试
<code>x < y, x <= y, x > y, x >= y</code>	大小比较, 集合子集和超集值相等性操作符
<code>x == y, x != y</code>	
<code>x y</code>	位或, 集合并集
<code>x ^ y</code>	位异或, 集合对称差
<code>x & y</code>	位与, 集合交集
<code>x << y, x >> y</code>	左移或右移y位
<code>x + y, x - y</code>	加法/合并, 减法, 集合差集
<code>x * y, x % y, x / y, x // y</code>	乘法/重复, 余数/格式化, 除法: 真除法或floor除法
<code>-x, +x</code>	一元减法, 识别
<code>~x</code>	按位求补 (取反)
<code>x ** y</code>	幂运算
<code>x[i]</code>	索引 (序列、映射及其他) 点号取属性运算, 函数调用
<code>x[i:j:k]</code>	分片
<code>x(...)</code>	调用 (函数、方法、类及其他可调用的)
<code>x.attr</code>	属性引用
<code>(...)</code>	元组, 表达式, 生成器表达式
<code>[...]</code>	列表, 列表解析
<code>{...}</code>	字典、集合、集合和字典解析

由于本书既介绍Python 2.6又涉及Python 3.0, 这里给出关于表5-2中的操作符的版本差异和最新添加:

- 在Python 2.6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, 因为它是多余的。值不相等测试使用`X != Y`就行了。
- 在Python 2.6中, 一个后引号表达式`'X'`和`repr(X)`的作用相同, 转换对象以显示字符串。由于其不好理解, Python 3.0删除了这个表达式, 使用更容易理解的`str`和`repr`内置函数, 本章后面的“数字显示的格式”小节介绍了这点。
- 在Python 2.6和Python 3.0中, floor除法表达式`(X // Y)`总是会把余数小数部分去

掉。在Python 3.0中， X / Y 表达式执行真正的除法（保留余数）和Python 2.6中的传统除法（截除为整数）。参见后面的“除法：传统除法、Floor除法和真除法”一节。

- 列表语法（`[...]`）用于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。参见第4章、第14章和第20章中的实例。
- `(...)`语法用于表示元组和表达式，以及生成器表达式，后者是产生所需结果的列表解析的一种形式，而不是构建一个最终的列表。参见第4章和第20章的示例。在所有3种结构中，圆括号有时候会省略。
- `{...}`语法表示字典常量，并且在Python 3.0中可以表示集合常量以及字典和集合解析。参见本章、第4章、第8章、第14章和第20章中关于集合的示例。
- `yield`和三元选择表达式在Python 2.5及其以后的版本中可用。前者返回生成器中的`send(...)`参数，后者是一个多行`if`语句的缩写形式。如果`yield`不是单独地位于一条赋值语句的右边的话，需要用圆括号。
- 比较操作符可以连续使用： $X < Y < Z$ 的结果与 $X < Y$ and $Y < Z$ 相同。参见本书后面的“比较：一般的和连续的”小节。
- 在最近的Python中，分片表达式 $X[I:J:K]$ 等同于用一个分片对象索引：`X[slice(I, J, K)]`。
- 在Python 2.X，混合类型的广义比较是允许的（把数字转换为一个普通类型，并且根据类型名称来排列其他的混合类型）。在Python 3.0中，非数字的混合类型的大小比较是不允许的，并且会引发异常，这包括按照代理排序。
- 在Python 3.0中，对字典的大小比较也不再支持（尽管支持相等性测试）；比较`sorted(dict.items())`是一种可能的替代。

在后面，我们将看到表5-2中的大多数操作符的应用；但是，首先，我们需要快速浏览一下这些操作符可能组合为表达式的方式。

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。表5-2根据操作符的优先级来排序：

- 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。
- 表5-2中位于同一行的表达式在组合的时候通常从左到右组合（除了幂运算，它是从右向左组合的，还有比较运算，是从左到右连接的）。

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：

$$\begin{array}{l} (X + Y) * Z \\ X + (Y * Z) \end{array}$$

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语言中的类型转换是很相似的。

Python是这样划分数字类型的复杂度的：整数比浮点数简单，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升级为一个复杂的数字，使得表达式获得一个复杂的结果。（在Python 2.6中，一般整数的值太大了以至于一般整数存储不下的时候，它会转换为长整数；在Python 3.0中，整数完全是长整数的形式）。

可以通过手动调用内置函数来强制转换类型：

```
>>> int(3.1415)           # Truncates float to integer
3
>>> float(3)              # Converts integer to float
3.0
```

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转换仅仅在将数字类型（例如，一个整数和一个浮点数）混合到一个表达式中的时候才适用，这包括那些使用数字和比较操作符的表达式。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。

注意：在Python 2.6中，非数字混合类型也可以比较，但是，不执行转换（混合类型比较根据一个确定但任意的规则）。在Python 3.0中，非数字混合类型的比较是不允许的，并且会引发异常。

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类型上，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+表达式做加法或连接，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类型。第16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它们在实际中的应用。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章查找提示）。

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。变量就是简单的名字（由你或Python创建），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1}。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，即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, 2.0)
>>> a % 2, b ** 2         # Modulus (remainder), power (4 ** 2)
(1, 16)
```

注1：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。


```
>>> 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)
5.0
>>> 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各部分分组。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 2.6进行整数的除法并相加得到结果为5；而Python 3.0执行带有余数的真除法并且得到上面所示的结果。如果想要在Python 3.0中执行整数除法，把这段代码编写为b // 2 + a（稍后更详细地介绍除法）。

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）。如果这个表达式中所有的数字都是整数，在Python 2.6中整数除法（4 / 5）将会产生结果并截断为0；但在Python 3.0中得到浮点数0.8（同样，等到稍后介绍除法的详细知识）。

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感到奇怪：

```

>>> b / (2.0 + a)                                # Auto echo output: more digits
0.800000000000000004

>>> 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。实际上，这真的只是一个显示问题——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，使用`print`；就像本章后面的“`str`和`repr`显示格式”小节所介绍的，你将会得到一个用户友好的显示。

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计算机中的数字的位数：

```

>>> num = 1 / 3.0
>>> num                                            # Echoes
0.33333333333333331
>>> print(num)                                    # print rounds
0.333333333333

>>> '%e' % num                                    # String formatting expression
'3.333333e-001'
>>> '%4.2f' % num                                 # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num)                        # String formatting method (Python 2.6 and 3.0)
'0.33'

```

这些方法中的最后三个使用了字符串格式化，这是灵活地进行格式化的一种工具，我们将会在以后的关于字符串的章节中（第7章）介绍它。其结果通过打印来显示或报告字符串。

比较：一般的和连续的

到目前为止，我们已经介绍了标准数字操作（加法和乘法），但是，还可以比较数字。一般的比较就像我们所期待的那样对数字起作用，它们比较操作数的相对大小，并且返回一个布尔类型的结果（我们通常会在更大的语句中测试该结果）：

```

>>> 1 < 2                                         # Less than
True
>>> 2.0 >= 1                                     # Greater than or equal: mixed-type 1 converted to 1.0
True

```

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：

```
>>> num = 1 / 3
>>> 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示：repr（也就是默认的交互模式回显）产生的结果看起来就好像它们是代码。str（也就是打印语句）转变为一种通常对用户更加友好的格式。一些对象两种方式都有：str用于一般用途，repr用于额外细节。这个概念将会为我们学习字符串以及类中的运算符重载做好铺垫，并且本书稍后会介绍关于这些内置函数的更多内容。

除了为任意对象提供打印字符串，str内置函数也是字符串数据类型的名字，并且能够用一个编码的名字来调用，从而从一个字节字符串解码一个Unicode字符串。我们将在本书第36章学习这一高级功能。

```
>>> 2.0 == 2.0           # Equal value
True
>>> 2.0 != 2.0          # Not equal value
False
```

再次注意数字表达式中是如何允许混合类型的（仅仅是数字表达式）；在这里的第二个测试中，Python比较了更为复杂的类型（浮点类型）的值。

有趣的是，Python还允许我们把多个比较连续起来执行范围测试。连续的比较是更大的布尔表达式的缩写。简而言之，Python允许我们把大小比较测试连接起来，成为诸如范围测试的连续比较。例如，表达式(A < B < C)测试B是否在A和C之间；它等同于布尔测试(A < B and B < C)，但更容易辨识（和录入）。例如，假设如下的赋值：

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

如下的两个表达式具有相同的效果，但是，第一个表达式简单而便于录入，并且，由于Python只需要计算Y一次，它运行起来可能略快一点：

```
>>> X < Y < Z           # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

获得false结果也是一样的，并且允许任意的连续长度：

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False

>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

我们可以在连续测试中使用其他的比较，但是，最终的表达式可能变得很晦涩，除非你按照Python的方式来计算它们。例如，如下表达式结果是false，因为1并不等于2：

```
>>> 1 == 2 < 3      # Same as: 1 == 2 and 2 < 3
False               # Not same as: False < 3 (which means 0 < 3, which is true)
```

Python并不会把`1 == 2`的False的结果和3进行比较，这样做的话，在技术上的含义和`0 < 3`相同，将会得到True（我们将在本章稍后了解到，True和False只不过定制为1和0）。

除法：传统除法、Floor除法和真除法

之前部分已经介绍了除法的工作方式，你应该知道其行为在Python 3.0和Python 2.6中略有差异，实际上，有3种类型的除法，有两种不同的除法操作符，其中一种操作符在Python 3.0中有了变化：

`X / Y`

传统除法和真除法。在Python 2.6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。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，在Python2.6和Python3.0中均能使用。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。

添加真除法是为了解决最初的传统除法的结果依赖于操作数类型（因此，其结果在Python这样的动态类型语言中很难预料）这一现象。由于这一限制，Python 3.0取消了传统除法：`/`和`//`操作符在Python 3.0中分别实现真除法和Floor除法。

概括来讲：

- 在Python 3.0中，/现在总是执行真除法，不管操作数的类型，都返回包含任何余数的一个浮点结果。//执行Floor除法，它截除掉余数并且针对整数操作数返回一个整数，如果有任何一个操作数是浮点类型，则返回一个浮点数。
- 在Python 2.6中，/表示传统除法，如果两个操作数都是整数的话，执行截断的整数除法；否则，执行浮点除法（保留余数）。//执行Floor除法，并且像在Python 3.0中一样工作，对于整数执行截断除法，对于浮点数执行浮点除法。

下面是两个操作符在Python 3.0和Python 2.6中的用法：

```
C:\misc> C:\Python30\python
>>>
>>> 10 / 4          # Differs in 3.0: keeps remainder
2.5
>>> 10 // 4         # Same in 3.0: truncates remainder
2
>>> 10 / 4.0        # Same in 3.0: keeps remainder
2.5
>>> 10 // 4.0       # Same in 3.0: truncates to floor
2.0

C:\misc> C:\Python26\python
>>>
>>> 10 / 4
2
>>> 10 // 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4.0
2.0
```

注意，在Python 3.0中，//的结果的数据类型总是依赖于操作数的类型：如果操作数中有一个是浮点数，结果就是浮点数；否则，结果是一个整数。尽管这可能与Python 2.X中类型依赖行为类似（正是该因素引发了在Python 3.0中的变化），但返回值类型的差异比返回值本身的差异要轻微很多。此外，对于依赖截断整数除法的程序（这种程序可能比你所意识到的要更为常见），//作为向后兼容工具的一部分而提供，对于整数类型，它必须返回整数。

支持两个Python版本

尽管/的行为在Python 2.6和Python 3.0中不同，我们仍然能够在自己的代码中支持这两个版本。如果你的程序依赖于截断整数除法，在Python 2.6和Python 3.0中都使用//。如果你的程序对于整数需要带有余数的浮点数结果，使用浮点数，从而确保代码在Python 2.6中运行的时候/的一个操作数是浮点数：

```
X = Y // Z          # Always truncates, always an int result for ints in 2.6 and 3.0
```

```
X = Y / float(Z)      # Guarantees float division with remainder in either 2.6 or 3.0
```

作为替代方法，我们可以使用一个`__future__` import在Python 2.6中打开Python 3.0的`/`，而不是用浮点转换来强制它：

```
C:\misc> C:\Python26\python
>>> from __future__ import division      # Enable 3.0 "/" behavior
>>> 10 / 4
2.5
>>> 10 // 4
2
```

Floor除法VS截断除法

一个细微之处在于：`//`操作符通常叫做截断除法，但是，更准确的说法是`floor`除法，它把结果向下截断到它的下层，即真正结果之下的最近的整数。其直接效果是向下舍入，并不是严格地截断，并且这对负数也有效。你可以使用Python的`math`模块来自己查看其中的区别（在使用模块中的内容之前，必须先导入模块；随后将更详细地介绍这些内容）：

```
>>> import math
>>> math.floor(2.5)
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)
2
>>> math.trunc(-2.5)
-2
```

在执行除法操作的时候，只是真正地截断了正的结果，因此截断除法和`floor`除法是相同的；对于负数，它就是一个`floor`结果（实际上，它们都是`floor`，但是对于正数来说，`floor`和截断是相同的）。下面是在Python 3.0中的情况：

```
C:\misc> c:\python30\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)

>>> 5 // 2, 5 // -2      # Truncates to floor: rounds to first lower integer
(2, -3)                  # 2.5 becomes 2, -2.5 becomes -3

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0  # Ditto for floats, though result is float too
(2.0, -3.0)
```

Python 2.6中的情况类似，但是`/`的结果再次有所不同：

```

C:\misc> c:\python26\python
>>> 5 / 2, 5 / -2          # Differs in 3.0
(2, -3)

>>> 5 // 2, 5 // -2        # This and the rest are the same in 2.6 and 3.0
(2, -3)

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)

```

如果你真的想要截断而不管符号，可以总是通过`math.trunc`来得到一个浮点除法结果，而不管是什么Python版本（请查看内置的`round`函数以了解相关的功能）：

```

C:\misc> c:\python30\python
>>> import math
>>> 5 / -2          # Keep remainder
-2.5
>>> 5 // -2         # Floor below result
-3
>>> math.trunc(5 / -2) # Truncate instead of floor
-2

C:\misc> c:\python26\python
>>> import math
>>> 5 / float(-2)    # Remainder in 2.6
-2.5
>>> 5 / -2, 5 // -2  # Floor in 2.6
(-3, -3)
>>> math.trunc(5 / float(-2)) # Truncate in 2.6
-2

```

为什么截断很重要

如果使用Python 3.0，这里有一个关于除法操作符的简短故事：

```

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)          # 3.0 true division
(2.5, 2.5, -2.5, -2.5)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)      # 3.0 floor division
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)          # Both
(3.0, 3.0, 3, 3.0)

```

对于Python 2.6的用户，除法像下面这样工作：

```

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)          # 2.6 classic division
(2, 2.5, -2.5, -3)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)      # 2.6 floor division (same)
(2, 2.0, -3.0, -3)

```


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 $2 + -3j$ 。下面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，并且可以通过标准的`cmath`模块（复数版的标准数学模块）中的工具进行处理。复数通常在面向工程的程序中扮演重要的角色。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、八进制和二进制记数

正如本章之前提到的那样，Python整数能够以十六进制、八进制和二进制记数法来编写，作为一般的以10位基数的十进制记数法的补充。本章开始处已经介绍了编码规则，这里让我们来看一些实际的例子。

记住，这些常量只是指定一个整数对象的值的一种替代方法。例如，Python 3.0和Python 2.6中的如下常量编码会产生具有3种进制的指定值的常规整数：

```
>>> 0o1, 0o20, 0o377          # Octal literals
(1, 16, 255)
>>> 0x01, 0x10, 0xFF          # Hex literals
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111  # Binary literals
(1, 16, 255)
```

这里，八进制值`0o377`、十六进制值`0xFF`和二进制值`0b11111111`，都表示十进制的255。Python默认地用十进制值（以10为基数）显示，但它提供了内置的函数，允许我们把整数转换为其他进制的数字字符串：

```
>>> oct(64), hex(64), bin(64)
('0100', '0x40', '0b1000000')
```

`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，而`bin`会将十进制数转换为二进制。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('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
```

```
>>> int('0x40', 16), int('0b1000000', 2)          # Literals okay too
(64, 64)
```

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。耍小聪明的用户也许能够提交一个删除机器上文件的字符串）：

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

最后，你能够使用字符串格式化方法调用和表达式将一个整数转换成八进制数和十六进制数的字符串：

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)
'100, 40, 1000000'

>>> '%o, %x, %X' % (64, 255, 255)
'100, ff, FF'
```

再次强调，字符串格式化将会在第7章介绍。

在继续学习之前，有两点需要注意。首先，Python 2.6的用户应该记住在编写八进制之前，直接用一个0开头，Python中最初的八进制格式如下：

```
>>> 0o1, 0o20, 0o377          # New octal format in 2.6 (same as 3.0)
(1, 16, 255)
>>> 01, 020, 0377            # Old octal literals in 2.6 (and earlier)
(1, 16, 255)
```

在Python 3.0中，这些例子中的第二组的语法将会产生错误。即便它在Python 2.6中不是一个错误，还是要小心，不要用0开始一个数字字符串，除非你真的是想要表示一个八进制的值。Python 2.6会将其当作是八进制数，但可能不像你所期待的那样工作，010在Python 2.6中总是八进制数，而不是十进制数（不管你是否这样认为）。也就是说，为了保持与十六进制和二进制的形式对称，Python 3.0修改了八进制的形式，在Python 3.0中必须使用0o010，并且，在Python 2.6中，也应该尽可能使用0o010。

其次，注意这些常量可以产生任意长度的整数。例如，下面的例子创建了十六进制形式的一个整数，然后先用十进制形式显示它，再将其转换为八进制和二进制的形式：

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
>>> X
5192296858534827628530496329220095L
>>> oct(X)
'017777777777777777777777777777777777L'
>>> bin(X)
```


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。

注意： 在即将发布的Python 3.1中，整数的`bit_length`方法也允许我们查询以二进制表示一个数字的值所需的位数。通过`bin`和内置函数`len`（本书第4章介绍过）得到二进制字符串的长度，然后在减去2，我们往往可以得到同样的效果，尽管这种方法效率较低：

```
>>> X = 99
>>> bin(X), X.bit_length()
('0b1100011', 7)
>>> bin(256), (256).bit_length()
('0b100000000', 9)
>>> len(bin(256)) - 2
9
```

其他的内置数学工具

除了核心对象类型以外，Python还支持用于数字处理的内置函数和内置模块。例如，内置函数`pow`和`abs`，分别计算幂和绝对值。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
>>> 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)

>>> pow(2, 4), 2 ** 4                                    # Exponentiation (power)
(16, 16)

>>> abs(-42.0), sum((1, 2, 3, 4))                      # Absolute value, summation
(42.0, 10)

>>> min(3, 1, 2, 4), max(3, 1, 2, 4)                   # Minimum, maximum
(1, 4)
```

这里展示的`sum`函数作用于数字的一个序列，`min`和`max`函数接受一个参数序列或者单个的参数。有各种各样的方法可以删除一个浮点数的小数位。我们前面介绍了截断和`floor`方法，我们也可以用`round`，不管是为了求值还是为了显示：

```

>>> math.floor(2.567), math.floor(-2.567)           # Floor (next-lower integer)
(2, -3)

>>> math.trunc(2.567), math.trunc(-2.567)           # Truncate (drop decimal digits)
(2, -2)

>>> int(2.567), int(-2.567)                         # Truncate (integer conversion)
(2, -2)

>>> round(2.567), round(2.467), round(2.567, 2)    # Round (Python 3.0 version)
(3, 2, 2.5699999999999998)

>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)         # Round for display (Chapter 7)
('2.6', '2.57')

```

正如我们在前面看到的，最后一个例子产生了我们通常会打印出的字符串，并且它支持各种格式化选项。同样，如果我们把倒数第二行的例子包含到一个`print`调用中以要求一个更加用户友好的显示，它将会输出(3, 2, 2.57)。然而，最后两行还是有差异的，`round`舍入一个浮点数但是仍然在内存中产生一个浮点数，而字符串格式化产生一个字符串并且不会得到一个修改后的数字：

```

>>> (1 / 3), round(1 / 3, 2), ('%.2f' % (1 / 3))
(0.3333333333333333, 0.33000000000000002, '0.33')

```

有意思的是，在Python中有3种方法可以计算平方根：使用一个模块函数、一个表达式或者一个内置函数（如果你关心性能，我们将在第四部分末尾的一个练习及其解答中回顾这些，可以看到哪种方法运行得更快）：

```

>>> import math
>>> math.sqrt(144)           # Module
12.0
>>> 144 ** .5               # Expression
12.0
>>> pow(144, .5)            # Built-in
12.0

>>> math.sqrt(1234567890)    # Larger numbers
35136.418286444619
>>> 1234567890 ** .5
35136.418286444619
>>> pow(1234567890, .5)
35136.418286444619

```

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Python 3.0中名为**`builtins`**的模块（Python 2.6中是**`__builtin__`**）。在本书后面的函数和模块部分中，有更多关于变量名解析的介绍。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。这个模块提供了工具，可以选出一个在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：

```
>>> import random
>>> random.random()
0.44694718823781876
>>> random.random()
0.28970426439292829

>>> 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使用`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、浮点数和复数。对于绝大多数程序员来说，需要进行的绝大多数数字处理都满足了。然而，Python还自带了一些更罕见的数字类型，值得让我们在这里快速浏览一下。

小数数字

Python 2.4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，只不过它们有固定的位数和小数点，因此小数是有固定的精度的浮点值。

例如，使用了小数对象，我们能够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断额外的小数数字。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一个理想的工具。

基础知识

最后一点值得探究。你可能已经知道，也可能还不知道：浮点数学缺乏精确性，因为用

来存储数值的空间有限。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-17
```

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-17
```

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')
```

注意：在Python 3.1中，将能够从一个浮点对象创建一个小数对象，通过decimal.Decimal.from_float(1.25)形式的调用。这一转换是精确的，但有时候会产生较多的位数。

设置全局精度

decimal模块中的其他工具可以用来设置所有小数数值的精度、设置错误处理等。例如，这个模块中的一个上下文对象允许指定精度（小数位数）和舍入模式（舍去、进位等）。该精度全局性地适用于调用线程中创建的所有小数：

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

这对于处理货币的应用程序特别有用，其中，美分表示为两个小数位数。在这个上下文里，小数实际上是手动舍入和字符串格式化的一种替代方式：

```
>>> 1999 + 1.33
```

```

2000.3299999999999
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')

```

小数上下文管理器

在Python 2.6和Python 3.0（及其以后的版本中），可使用上下文管理器语句来重新设置临时精度。在语句退出后，精度又重新设置为初始值：

```

C:\misc> C:\Python30\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')

```

尽管这条语句很有用，但它需要比这里所介绍的更多的背景知识；参见本书第33章对with语句的介绍。

由于小数类型在实际中仍然很少用到，请参考Python的标准库手册和交互式帮助来了解更多细节。并且由于小数和分数类型一样解决了浮点数的某些精度问题，让我们继续下一节，看看如何比较这两种类型。

分数类型

Python 2.6和Python 3.0引入了一种新的数字类型——分数，它实现了一个有理数对象。它明确地保留一个分子和一个分母，从而避免了浮点数学的某些不精确性和局限性。

基本知识

分数是前面小节所介绍的已有的小数固定精度类型的“近亲”，它们都可以通过固定小数位数和指定舍入或截断策略来控制数值精度。分数以类似于小数的方式使用，它也存在于模块中；导入其构造函数并传递一个分子和一个分母就可以产生一个分数。下面的交互式例子展示了如何做到这一点：

```

>>> from fractions import Fraction
>>> x = Fraction(1, 3)                                     # Numerator, denominator

```

```
>>> y = Fraction(4, 6)                # Simplified to 2, 3 by gcd

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

一旦创建了分数，它可以像平常一样用于数学表达式中：

```
>>> x + y
Fraction(1, 1)
>>> x - y                # Results are exact: numerator, denominator
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

分数对象也可以从浮点数字符串来创建，这和小数很相似：

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

数值精度

注意，这和浮点数类型的数学有所区别，那是受到浮点数硬件的底层限制的约束。相比较而言，这里与对浮点数对象执行的操作是相同的，注意它们有限的精度：

```
>>> a = 1 / 3.0                # Only as accurate as floating-point hardware
>>> b = 4 / 6.0                # Can lose precision over calculations
>>> a
0.33333333333333331
>>> b
0.66666666666666663

>>> a + b
1.0
>>> a - b
-0.33333333333333331
>>> a * b
0.22222222222222221
```

对于那些用内存中给定的有限位数无法精确表示的值，浮点数的局限尤为明显。分数和小数都提供了得到精确结果的方式，虽然要付出一些速度的代价。例如，在下面的例子中（重复前一小节的），浮点数并没有准确地给出期望的0的答案，但其他的两种类型都做到了：

```
>>> 0.1 + 0.1 + 0.1 - 0.3                # This should be zero (close, but not exact)
5.5511151231257827e-17

>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

此外，分数和小数都能够提供比浮点数更直观和准确的结果，它们以不同的方式做到这点（使用有理数表示以及通过限制精度）：

```
>>> 1 / 3                                # Use 3.0 in Python 2.6 for true "/"
0.33333333333333331

>>> Fraction(1, 3)                       # Numeric accuracy
Fraction(1, 3)

>>> import decimal
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal(1) / decimal.Decimal(3)
Decimal('0.33')
```

实际上，分数保持精确性，并且自动简化结果。继续前面的交互例子：

```
>>> (1 / 3) + (6 / 12)                   # Use ".0" in Python 2.6 for true "/"
0.83333333333333326

>>> Fraction(6, 12)                       # Automatically simplified
Fraction(1, 2)

>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)

>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')

>>> 1000.0 / 1234567890
8.1000000737100011e-07
>>> Fraction(1000, 1234567890)
Fraction(100, 123456789)
```

转换和混合类型

为了支持分数转换，浮点数对象现在有一个方法，能够产生它们的分子和分母比，分数有一个`from_float`方法，并且`float`接受一个`Fraction`作为参数。跟踪如下的交互看看这是如何做到的（第二个测试中的`*`是一种特殊的语法，它把一个元组扩展到单个的参数中；当我们在第18章中学习函数参数的时候再详细地讨论这一点）：

```
>>> (2.5).as_integer_ratio()              # float object method
(5, 2)
```

```

>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio())      # Convert float -> fraction: two args
>>> z                                          # Same as Fraction(5, 2)
Fraction(5, 2)

>>> x                                          # x from prior interaction
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                               # 5/2 + 1/3 = 15/6 + 2/6

>>> float(x)                                 # Convert fraction -> float
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)                # Convert float -> fraction: other way
Fraction(7, 4)
>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)

```

最终，表达式中允许某些类型的混合，尽管Fraction有时必须手动地传递以确保精度。研究如下的交互示例来看看这是如何做到的：

```

>>> x
Fraction(1, 3)
>>> x + 2                                    # Fraction + int -> Fraction
Fraction(7, 3)
>>> x + 2.0                                  # Fraction + float -> float
2.3333333333333335
>>> x + (1./3)                               # Fraction + float -> float
0.6666666666666666

>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)                       # Fraction + Fraction -> Fraction
Fraction(5, 3)

```

警告：尽管可以把浮点数转换为分数，在某些情况下，这么做的时候会有不可避免的精度损失，因为这个数字在其最初的浮点形式下是不精确的。当需要的时候，我们可以通过限制最大分母值来简化这样的结果：

```

>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()             # Precision loss from float
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())

```

```

>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488.          # 5 / 3 (or close to it!)
1.6666666666666667

>>> a.limit_denominator(10)                          # Simplify to closest fraction
Fraction(5, 3)

```

要了解有关分数类型的更多细节，自己进一步体验，并且查询Python 2.6和Python 3.0的库手册以及其他的文档。

集合

Python 2.4引入了一种新的类型——集合（set），这是一些唯一的、不可变的对象的一个无序集合（collection），这些对象支持与数学集合理论相对应的操作。根据定义，一个项在集合中只能出现一次，不管将它添加了多少次。同样，集合有着广泛的应用，尤其是在涉及数字和数据库的工作中。

因为它是其他对象的集合，因此，它具有列表和字典这样的对象的某些共同行为，而这些对象超出了本章讨论的范围。例如，集合是可以迭代的，可以根据需要增长或缩短，并且能够包含各种对象类型。我们将会看到，一个集合的行为很像一个无值的字典的键，但是，它还支持额外的操作。

然而，由于集合是无序的，并且不会把键匹配到值，它们既不是序列也不是映射类型；它们是自成一体的类型。此外，由于集合本质上具有基本的数学特性（它对于很多读者来说，可能更加学院派，并且比像字典这样更为普遍的对象用得要少很多），在这里，我们将介绍Python的集合对象的基本工具。

Python 2.6中的集合基础知识

根据你使用的是Python 2.6还是Python 3.0，有几种不同方法来创建集合。既然本书涉及这两个版本，让我们先从Python 2.6的情况开始，其方法在Python 3.0中也是可用的（并且有时候还是必须的）；稍后，我们将根据Python 3.0的扩展来细化。要创建一个集合对象，向内置的set函数传递一个序列或其他的可迭代的对象：

```

>>> 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'])          # 2.6 display format

```

集合通过表达式操作符支持一般的数学集合运算。注意，不能在一般序列上应用这些表达式，必须通过序列创建集合后才能使用这些工具。

```
>>> 'e' in x                    # Membership
True

>>> x - y                       # Difference
set(['a', 'c', 'e'])

>>> x | y                       # Union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                       # Intersection
set(['b', 'd'])

>>> x ^ y                       # Symmetric difference (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])

>>> x > y, x < y                # Superset, subset
(False, False)
```

除了表达式，集合对象还提供了对应这些操作的方法，以及更多的支持改变集合的方法，集合`add`方法插入一个项目、`update`是按位置求并集，`remove`根据值删除一个项目（在任何集合实例或集合类型名上运行`dir`来查看所有可用的方法）。假设`x`和`y`仍然像前面交互中的一样：

```
>>> z = x.intersection(y)      # Same as x & y
>>> z
set(['b', 'd'])
>>> z.add('SPAM')              # Insert one item
>>> z
set(['b', 'd', 'SPAM'])
>>> z.update(set(['X', 'Y']))   # Merge: in-place union
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])
>>> z.remove('b')              # Delete one item
>>> z
set(['Y', 'X', 'd', 'SPAM'])
```

作为可迭代的容器，集合也可以用于`len`、`for`循环和列表解析这样的操作中。然而，由于它们都是无序的，所以不支持像索引和分片这样的操作：

```
>>> for item in set('abc'): print(item * 3)
...
aaa
ccc
bbb
```

最后，尽管前面介绍的集合表达式通常需要两个集合，它们基于方法的对应形式往往对任何可迭代类型也有效：


```

>>> S = set([1, 2, 3])

>>> S | set([3, 4])                # Expressions require both to be sets
set([1, 2, 3, 4])
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> S.union([3, 4])                # But their methods allow any iterable
set([1, 2, 3, 4])
>>> S.intersection([1, 3, 5])
set([1, 3])
>>> S.issubset(range(-5, 5))
True

```

要了解关于集合操作的更多细节，参阅Python的库参考手册，或者其他参考书。尽管集合操作可以在Python中和其他类型（如列表和字典）一起手动编写，Python的内置集合还是使用高效率的算法和实现技术来提供快速和标准的操作。

Python 3.0中的集合常量

如果你认为集合很“酷”，它们最近会变得更酷。在Python 3.0中，我们仍然使用集合内置函数来创建集合对象，但是Python 3.0也添加了新的集合常量形式，该形式使用前面为字典所保留的花括号。在Python 3.0中，如下的形式是等同的：

```

set([1, 2, 3, 4])                # Built-in call
{1, 2, 3, 4}                     # 3.0 set literals

```

这个语法是有意义的，因为集合基本上就像是无值的字典，集合的项是无序的、唯一的、不可改变的，因此，它们的行为和字典的键很像。由于字典键列表在Python 3.0中是视图对象，它支持像交集和并集这样的类似集合的行为，这种相似性甚至更加惊人（参见第8章了解关于字典视图对象的更多内容）。

实际上，不管如何创建集合，Python 3.0都使用新的常量格式来显示它。在Python 3.0中要创建空的集合或从已有的可迭代对象构建集合（使用集合解析的简述，将在本章稍后介绍），还是需要内置的set函数，但是新的常量便于初始化具有已知结构的集合：

```

C:\Misc> c:\python30\python
>>> set([1, 2, 3, 4])                # Built-in: same as in 2.6
{1, 2, 3, 4}
>>> set('spam')                     # Add all items in an iterable
{'a', 'p', 's', 'm'}

>>> {1, 2, 3, 4}                     # Set literals: new in 3.0
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S.add('alot')
>>> S
{'a', 'p', 's', 'm', 'alot'}

```

前面小节中所讨论的所有集合处理操作在Python 3.0中都同样有效，但是结果集合显示有所不同：

```
>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}                                # Intersection
{1, 3}
>>> {1, 5, 3, 6} | S1                          # Union
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}                             # Difference
{2}
>>> S1 > {1, 3}                                # Superset
True
```

注意，在Python中{}仍然是一个字典。空的集合必须通过内置函数set来创建，并且以同样方式显示：

```
>>> S1 - {1, 2, 3, 4}                          # Empty sets print differently
set()
>>> type({})                                   # Because {} is an empty dictionary
<class 'dict'>

>>> S = set()                                  # Initialize an empty set
>>> S.add(1.23)
>>> S
{1.23}
```

与Python 2.6一样，Python 3.0中用常量创建的集合支持同样的方法，其中的一些支持表达式所不支持的通用可迭代操作数：

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}

>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

不可变限制和冻结集合

集合是强大而灵活的对象，但是，它们在Python 3.0和Python 2.6中都有一个限制，我们需要铭记，很大程度上是由于其实现，集合只能包含不可变的（即可散列的）对象类

型。因此，列表和字典不能嵌入到集合中，但是，如果你需要存储复合值的话，元组是可以嵌入的。在集合操作中使用元组的时候，元组比较其完整的值：

```
>>> S
{1.23}
>>> S.add([1, 2, 3])           # Only mutable objects work in a set
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))
>>> S                           # No list or dict, but tuple okay
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}  # Union: same as S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S              # Membership: by complete values
True
>>> (1, 4, 3) in S
False
```

例如，集合中的元组可以用来表示日期、记录、IP地址等（本书的本部分后续将更详细地介绍元组）。集合本身也是不可改变的，因此，不能直接嵌入到其他集合中；如果需要在另一个集合中存储一个集合，可以像调用`set`一样来调用`frozenset`，但是，它创建一个不可变的集合，该集合不可修改并且可以嵌套到其他集合中。

Python 3.0中的集合解析

除了常量，Python 3.0还引入了一个集合解析构造，它类似于我们在第4章中介绍过的列表解析的形式，但是，编写在花括号中而不是方括号中，并且作用于集合而不是列表。集合解析运行一个循环并在每次迭代时收集一个表达式的结果，通过一个循环变量来访问当前的迭代值以用于集合表达式中。结果是通过运行代码创建的一个新的集合，它具备所有一般的集合行为：

```
>>> {x ** 2 for x in [1, 2, 3, 4]}    # 3.0 set comprehension
{16, 1, 4, 9}
```

在这个表达式中，循环部分编写在右边，而集合表达式编写在左边(`x**2`)。和列表解析一样，我们可以很好地理解这个表达式的含义：“对于列表中的每一个X，给出包含X的平方的一个新的集合”。解析也可以迭代其他类型的对象，例如字符串（下面的第一个例子展示了如何从一个已有的可迭代对象创建一个集合）：

```
>>> {x for x in 'spam'}              # Same as: set('spam')
{'a', 'p', 's', 'm'}

>>> {c * 4 for c in 'spam'}          # Set of collected expression results
{'ssss', 'aaaa', 'pppp', 'mmmm'}
>>> {c * 4 for c in 'spamham'}
```

```
{'ssss', 'aaaa', 'hhhh', 'pppp', 'mmm'}

>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmm', 'xxx'}
{'ssss', 'aaaa', 'pppp', 'mmm', 'xxx'}
>>> S & {'mmm', 'xxx'}
{'mmm'}
```

因为其他的关于解析的知识要依赖于我们现在还不准备介绍的底层概念，这些概念将推迟到本书后面再详细介绍。在第8章中，我们将遇到Python 3.0中的另一种解析，即字典解析，并且，我们随后将介绍关于所有解析（列表、集合、字典和生成器）的更多内容，特别是在第14章和第20章中。随后你将会了解到，在没有学习较大的语句之前，很难理解所有的解析，包括集合、这里所没有介绍的其他语法的支持、包括嵌套循环和if测试等。

为什么使用集合

集合操作有各种各样常见的用途，其中一些比数学更加实用。例如，由于项在集合中只能存储一次，集合（set）可以用来把重复项从其他集合（collection）中过滤掉。直接把集合（collection）转换为一个集合（set），然后再转换回来即可（因为集合是可迭代的，这里的list调用对其有效）：

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                                # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

当你遍历图形或其他的回环结构的时候，集合可以用来记录已经访问过的位置。例如，我们将分别在第24章和第30章学习的传递性模块重载和继承树列表程序示例，必须确保访问过的项不再循环。尽管把访问状态作为键记录到字典中很高效，但集合提供了几乎等同的一种替代方式（并且可能更直观或更晦涩，取决你征求谁的意见）。

最后，在处理较大的数据集合的时候（例如，数据库查询结果），两个集合的交集包含了两个领域中共有的对象，并集包含了两个集合中的所有项目。为了说明，这里给出集合操作的一些实际例子，这些操作应用于一个假定公司的人员名单，使用Python 3.0集合常量（在Python 2.6中使用set）：

```
>>> engineers = {'bob', 'sue', 'ann', 'vic'}
>>> managers = {'tom', 'sue'}

>>> 'bob' in engineers                                # Is bob an engineer?
True

>>> engineers & managers                              # Who is both engineer and manager?
```

```

{'sue'}

>>> engineers | managers          # All people in either category
{'vic', 'sue', 'tom', 'bob', 'ann'}

>>> engineers - managers          # Engineers who are not managers
{'vic', 'bob', 'ann'}

>>> managers - engineers          # Managers who are not engineers
{'tom'}

>>> engineers > managers          # Are all managers engineers? (superset)
False

>>> {'bob', 'sue'} < engineers    # Are both engineers? (subset)
True

>>> (managers | engineers) > managers  # All people is a superset of managers
True

>>> managers ^ engineers          # Who is in one but not both?
{'vic', 'bob', 'ann', 'tom'}

>>> (managers | engineers) - (managers ^ engineers)  # Intersection!
{'sue'}

```

在Python库手册以及关系数据库理论的一些相关资料中，我们可以找到关于集合操作的更多细节。在本书第8章中介绍Python 3.0中的字典视图对象的时候，我们将再次回顾这里所见到的一些集合操作。

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以不同的形式显示后的定制版本而已。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如今正式地有了一种明确的布尔型数据类型，叫做`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，除了它们有特定的显示逻辑：它们是作为关键字`True`和`False`显示的，而不是数字1和0（从技术上来讲，`bool`为它的两个对象重新定义了`str`和`repr`的字符串格式）。

由于这个定制，布尔表达式在交互提示模式的输出就作为关键字`True`和`False`来显示，而不是曾经的1和0。此外，布尔型让真值更精确。例如，一个无限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。我们将在第三部分深入讨论这些语句。

还有对于其他实际的用途，你能够将True和False看做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+4得到了5！

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1                                # Same value
True
>>> True is 1                                # But different object: see the next chapter
False
>>> True or False                            # Same as: 1 or 0
True
>>> True + 4                                # (Hmmm)
5
```

你可能不会在真正的Python代码中遇到像这里例子中最后一个那样的表达式，你可以完全忽略其更深入的形式上的含义。

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。

数字扩展

最后，尽管Python的核心数字类型提供的功能对于大多数应用程序已经够用了，还是有大量的第三方开源扩展可以用来解决更加专门的需求。由于数字编程是Python的常用领域，你将会发现众多的高级工具。

例如，如果你需要做一些正式的数字计算，一个叫做NumPy (Numeric Python)的可选的Python扩展提供了高级的数字编程工具，例如矩阵数据类型、向量处理和高级的计算库。像Los Alamos和NASA这样的核心科学编程组织，使用带有NumPy的Python来实现此前用C++、FORTRAN、Matlab编写的任务。Python和NumPy的组合往往可以比作是一款免费的、更加灵活的Matlab，可以得到NumPy的性能以及Python语言及其库。

由于NumPy如此高级，我们不打算在本书中进一步介绍它。你可以通过在Python的PyPI站点或者通过搜索Web，来找到对高级数字编程的其他支持，包括图形工具和绘制工具、统计库以及流行的SciPy包。另外，还要注意，NumPy目前是一个可选的扩展；还没有纳入到Python中，必须单独安装。

本章小结

本章介绍了Python数字对象类型和能够应用于它们的操作。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、分数和集合。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。

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. 怎样能够截断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、十六进制或二进制的形式？
9. 如何将一个八进制、十六进制或二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. 求平方根、 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)。上述两种方式的任何一种也可以用来计算一个数的0.5次方（例如， $X ** .5$ ）。

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)`函数和`math.trunc(N)`函数会省略小数部分，而`round(N, digit)`函数做四舍五入。我们可以使用 `math.floor(N)`来计算floor，并且使用字符串格式化操作来舍入以便于显示。
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。在某种意义上，Python 3.0 的/除法也会转换，它总是返回一个包含余数的浮点数结果，即便两个操作数都是整数。
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字符串的形式返回。%字符串表达式也会实现这样的目标。在Python 2.6和Python 3.0中，`bin(I)`也会返回一个数字的二进制数字字符串。%字符串格式化表达式和字符串格式化方法也为这样的转换提供方法。
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8、16或2作为base的参数）。`eval(S)`函数也能够用作这个目的，但是运行起来开销更大也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转换而已。

动态类型简介

在上一章学习了Python数字类型，本章开始对Python的核心对象类型进行深入探索。我们将会在下章继续对象类型之旅，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，这些概念无疑是Python语言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，但变量还可以工作。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，或者没有告诉它`a`应该作为一个整数类型对象，但一样也能工作。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：

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就创建了它。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。

总而言之，变量在赋值的时候才创建，它可以引用任何类型的对象，并且必须在引用之前赋值。这意味着，不需要通过脚本声明所要使用的名字，但是，必须初始化名字然后才能更新它们；例如，必须把计数器初始化为0，然后才能增加它。

这种动态类型与传统语言的类型相比有明显的不同。刚入门时，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）。

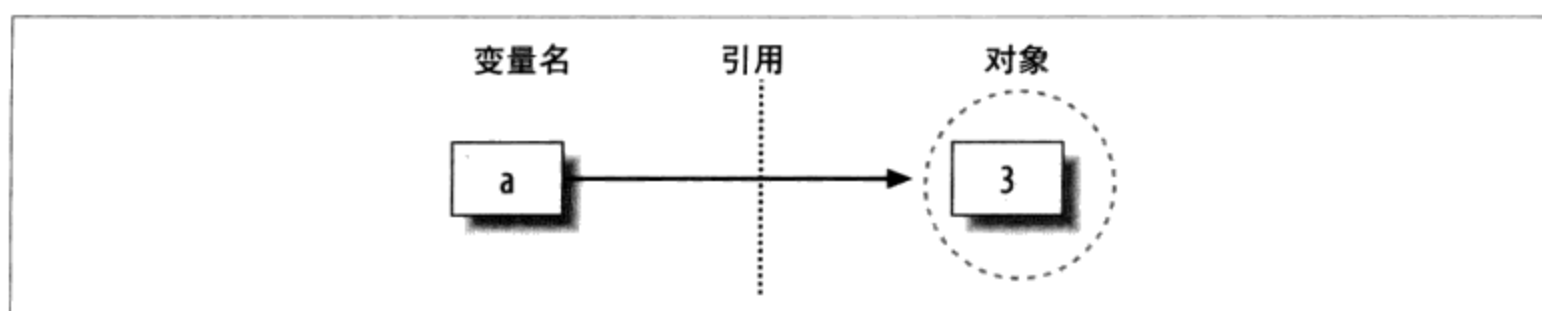


图6-1：变量名和对象，在运行`a=3`后。变量`a`变成对象`3`的一个引用。在内部，变量事实上是到对象内存空间(通过运行常量表达式`3`而创建)的一个指针

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。
- 对象是分配的一块内存，有足够的空间去表示它们所代表的值。
- 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都创建了一个新的对象（换言之，一块内存）去表示这个值。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示它的值那么简单。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识这个对象的类型，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。要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
```

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，尤其是那些在原处修改的对象（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。

```
>>> 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 = 'spam'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修改为对不同的对象的引用。因为变量没有类型，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（叫做str）。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'
```

答案是，在Python中，每当一个变量名被赋予了一个新的对象，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叫做垃圾收集。

为了讲清楚，考虑下面的例子，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。

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是这样来实现这一功能的：它在每个对象中保持了一个计数器，计数器记录了当前指向该对象的引用的数目。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，就会导致它的空间被回收。

垃圾收集最直接的、可感受到的好处就是，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。

注意：从技术上讲，Python的垃圾收集主要基于引用计数器，正如前面所介绍的。然而，它也有有一部分功能可以及时地检测并回收带有循环引用的对象。如果你确保自己的代码没有产生循环引用，可以关闭这部分功能，但该功能是默认可用的。

由于引用实现为指针，一个对象有可能会引用自身，或者引用另一个引用了自身的对象。例如，第一部分末尾的练习3及其在附录B中的解答，展示了如何把一个列表的引用嵌入其自身中，从而创建一个循环。对来自用户定义的类的对象的属性赋值的时候，会产生同样的现象。尽管相对很少，由于这样的对象的引用计数器不会清除为0，必须特别对待它们。

要了解Python的循环检测器的更多细节，参见Python库手册中gc模块的文档。还要注意，这里对于Python的垃圾收集器的介绍只适用于标准的Cpython、Jython和IronPython且可能使用不同的方案，尽管直接效果都是类似的，都是未使用的空间又自动重新申请。

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。变量a正在使用，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中叫做共享引用——多个变量名引用了同一个对象。

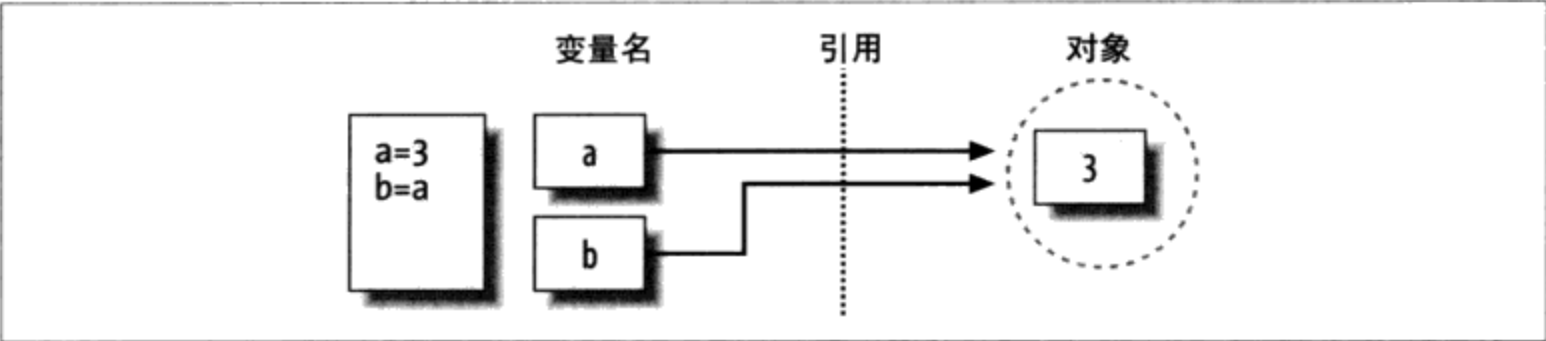


图6-2：运行赋值语句 `b = a` 之后的变量名和对象。变量b成为对象3的一个引用。在内部，变量实际上是一个指针指向了对象的内存空间，该内存空间是通过运行常量表达式3创建的

下一步，假设运行另一个语句扩展了这样的情况：

```
>>> 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。

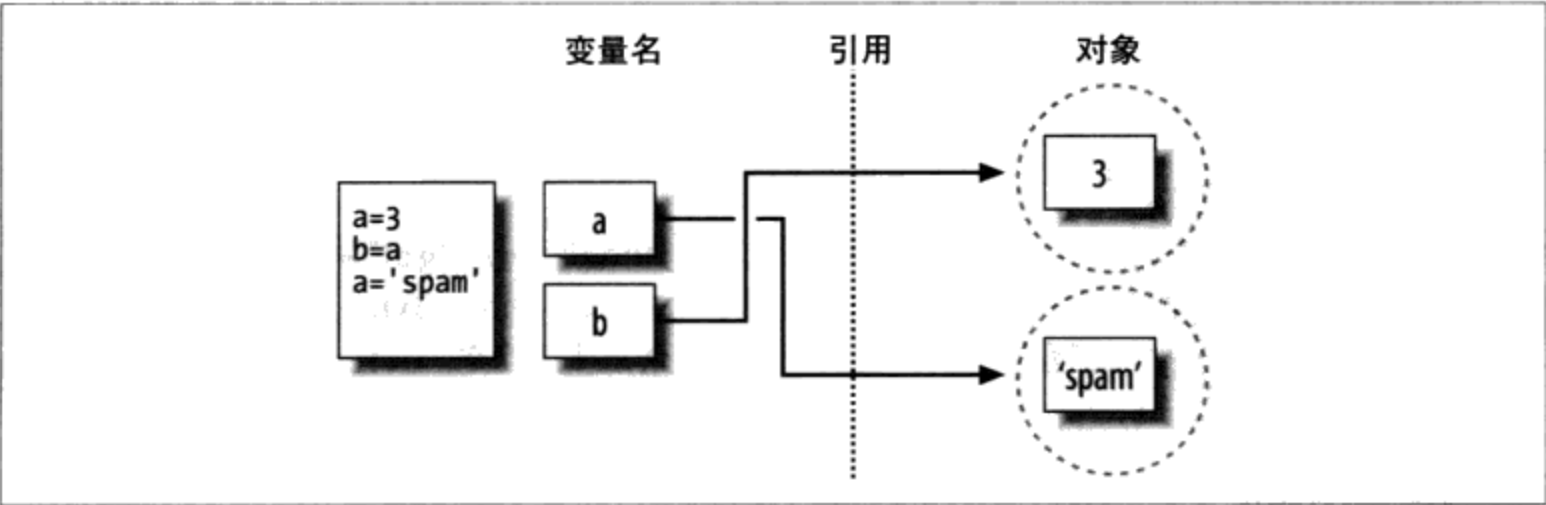


图6-3：最终运行完赋值语句 `a = 'spam'` 后的变量名和对象。变量a引用了由常量表达式 'spam' 所创建的新对象（例如，内存空间），但是变量b仍然引用原始的对象3。因为这个赋值运算改变的不是对象3，仅仅改变了变量a，变量b并没有发生改变

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）。这并不会产生改变了b的副作用。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，因此没有方法在原处修改它。

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：给一个变量赋一个新的值，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。当可变的对象以及原处的改变进入这个场景，那么这个情形会有某种改变。想知道是怎样一种变化的话，请继续学习。

共享引用和在原处修改

正如你在这部分后边的章节将会看到的那样，有一些对象和操作确实会在原处改变对象。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是一个包含了对象2、3和4的列表。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直接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都引用了相同的对象。另外，我们实际上并没有改变L2，但是它的值将发生变化，因为它已经被修改了。

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]

```

这里，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（字典和集合，因为它们不是序列）上，复制一个字典或集合应该使用X.copy()方法调用。而且，注意标准库中的copy模块有一个通用的复制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 top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)          # Make deep copy of any object Y: copy all nested parts

```

我们将会在第8章和第9章更深入地了解列表和字典，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，这种对象包括了列表、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，可以根据需要直接拷贝对象。

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是否相等。让我们创建一个共享引用来说明：

```
>>> L = [1, 2, 3]
>>> 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，但是是不同的对象，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为了执行速度而采用的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，类属性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中变量和对象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。在下一章将会继续我们的学习对象之旅——学习字符串。

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。我们永远都不会在原处覆盖一个字符串（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。

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，字符串可以用来表示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。它们可以用来存储字节的绝对二进制值，以及在国际化程序中用到的多字节的Unicode。

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和像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，并且它们不可以在原处修改。实际上，字符串是我们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。

表 7-1：常见字符串常量和表达式

操作	解释
<code>s= ''</code>	空字符串
<code>s= "spam's"</code>	双引号和单引号相同
<code>S = 's\np\ta\x00m'</code>	转义序列
<code>s = """..."""</code>	三重引号字符串块
<code>s= r'\temp\spam'</code>	Raw字符串
<code>S = b'spam'</code>	Python 3.0中的字节字符串（参见第36章）
<code>s = u'spam'</code>	仅在Python 2.6中使用的Unicode字符串（参见第36章）
<code>s1 + s2</code>	合并，重复
<code>s * 3</code>	
<code>s [i]</code>	索引，分片，求长度
<code>s [i:j]</code>	
<code>len(s)</code>	
<code>"a %s parrot" % kind</code>	字符串格式化表达式
<code>"a {0} parrot".format(kind)</code>	Python 2.6和Python 3.0中的字符串格式化方法
<code>s.find('pa')</code>	字符串方法调用：搜索
<code>s.rstrip()</code>	移除空格
<code>s.replace('pa', 'xx')</code>	替换
<code>s.split(',')</code>	用展位符分隔
<code>s.isdigit()</code>	内容测试
<code>s.lower()</code>	短信息转换
<code>S.endswith('spam')</code>	结束测试
<code>'spam'.join(strlist)</code>	插入分隔符
<code>S.encode('latin-1')</code>	Unicode编码等
<code>for x in S: print(x)</code>	迭代，成员关系
<code>'spam' in S</code>	
<code>[c * 2 for c in S]</code>	
<code>map(ord, S)</code>	

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；甚至还有更高级的文本处理工具，如XML解析器，我们将在第36章简单介绍。然而，本书主要关注表7-1介绍的基本表示。

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。Python带有很多字符串工具，我们不会在这里介绍所有这些工具；完整的介绍可以在Python的库手册中找到。这里，我们的目标是介绍很常用的工具以给出一个有代表性的例子，我们在这里没有介绍的那些方法，和我们见到的方法在很大程度上是类似的。

注意： 内容提示：从技术上讲，本章介绍的只是Python中的字符串内容的一部分，即大多数程序员需要知道的内容。它介绍了基本的str字符串类型，该类型用来处理ASCII文本并且不管使用何种Python版本都能同样地工作。也就是说，本章有意把讨论范围限制在大多数Python脚本中会用到的字符串处理基础知识。

从更为正式的角度讲，ASCII是Unicode文本的一种简单形式。Python通过包含各种不同的对象类型，解决了文本和二进制数据之间的区别：

- Python 3.0中，有3种字符串类型：str用于Unicode文本（ASCII或其他），bytes用于二进制数据（包括编码的文本），bytearray是bytes的一种可变的变体。
- 在Python 2.6中，unicode字符串表示宽Unicode文本，str字符串处理8位文本和二进制数据。

bytearray类型在Python 2.6及其以后的版本中可用，但在更早的版本就不可用了，并且它在其他版本中并不像在Python 3.0中那样与二进制数据紧密相连。由于大多数程序员不需要深入了解Unicode编码或二进制数据格式的细节，我们将这些细节放到本书的高级话题部分的第36章中介绍。

如果你需要了解替代字符集或打包的二进制数据和文件这样更高级的字符串概念，在阅读完本章内容后继续阅读第36章。现在，我们将关注基本的字符串类型及其操作。你将会发现，我们这里所学习的基础知识也直接应用于Python的工具集中更高级的字符串类型。

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"`
- Python 3.0中的Byte字符串 (参见第36章): `b'sp\x01am'`
- 仅在Python 2.6中使用的Unicode字符串 (参见第36章): `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,并且我们将推迟到本书第36章再讨论最后两种高级形式。让我们先快速看看其他的形式。

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杠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串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(在第12章中将会看到,把这种形式放到圆括号中,就可以允许它跨越多行):

```
>>> title = "Meaning " 'of' " Life"      # Implicit concatenation
>>> title
'Meaning of Life'
```

注意,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。你也能通过反斜杠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杠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：反斜杠用来引入特殊的字节编码，是转义序列。

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“\”，以及在他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
```

这个字符串长度为五个字节：分别包含了一个ASCII a字符，一个换行字符、一个ASCII b字符等。注意原始的反斜杠字符并不真正和字符串一起存储在内存中；它们告诉Python字符串中保存的特殊字节值。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：字符串反斜杠字符

转义	意义
\newline	忽视（连续）
\\	反斜杠（保留\）
\'	单引号（保留'）
\"	双引号（保留"）
\a	响铃

表7-2：字符串反斜杠字符（续）

转义	意义
<code>\b</code>	倒退
<code>\f</code>	换页
<code>\n</code>	新行（换行）
<code>\r</code>	返回
<code>\t</code>	水平制表符
<code>\v</code>	垂直制表符
<code>\N{id}</code>	Unicode数据库ID
<code>\uhhhh</code>	Unicode 16位的十六进制值
<code>\Uhhhhhhhh</code>	Unicode 32位的十六进制值 ^a
<code>\xhh</code>	十六进制值
<code>\ooo</code>	八进制值
<code>\0</code>	Null（不是字符串结尾）
<code>\other</code>	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（将八进制编码转义为一个数字）：

```
>>> s = 'a\b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

在Python中，零（空）字符不会像C语言那样去结束一个字符串。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没有字符会结束一个字符串。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
```

注意，Python以十六进制显示非打印的字符，不管是如何指定它们的。我们可以自由地组合表7-2中的绝对值转义和更多符号转义类型。如下的字符串包含了字符“spam”一个制表符和换行符，以及以十六进制编码的一个绝对零值字节：

```
>>> S = "s\tp\na\x00m"
>>> S
's\tp\na\x00m'
>>> len(S)
7
>>> print(S)
s      p
a m
```

当使用Python处理二进制数据文件时，了解这些知识显得格外重要。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}。

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，它就直接在最终的字符串中保留反斜杠。

```
>>> 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杠，重复两个反斜杠（“\\”是“\”的转义字符）或者使用raw字符串。在下一部分内容中我将对raw进行介绍。

raw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，为了引入转义字符而使用适应的反斜杠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”，它

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二进制模式下打开的（使用open模式的标志位b，例如'rb'，'wb'等）。在Python 3.0中，二进制文件的内容是一个bytes字符串，带有一个类似于常规字符串的接口；在Python 2.6中，这样的内容是一个常规的str字符串。请参照第9章中介绍的标准struct模块，它可以解析处理从一个文件载入的二进制数据；第36章将更广泛地介绍二进制文件和字节字符串。

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。

会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会将反斜杠作为常量来保持，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中增加字母r：

```
myfile = open(r'C:\new\text.dat', 'w')
```

还有一种办法：因为两个反斜杠是一个反斜杠的转义序列，能够通过简单地写两个反斜杠去保留反斜杠：

```
myfile = open('C:\\new\\text.dat', 'w')
```

实际上，当打印一个嵌入了反斜杠字符串时，Python自身也会使用这种写两个反斜杠的方法：

```
>>> 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
```

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杠。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杠。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的字节数，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，你会发现每个反斜杠只占一个字符，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的路径中使用斜杠表示字符串路径，因为Python试图以可移植的方法解释路径（例如，打开文件的时候，'C:/new/text.dat'也有效）。尽管这样，如果你编写的路径使用Windows的反斜杠，raw字符串是很有用处的。

注意： 尽管有用，但一个raw字符串也不能以单个的反斜杠结尾，因为，反斜杠会转义后续引用的字符，仍然必须转义外围引号字符以将其嵌入到该字符串中。也就是说，r"...\"不是一个有效的字符串常量，一个raw字符串不能以奇数个反斜杠结束。如果需要用单个的反斜杠结束一个raw字符串，可以使用两个反斜杠并分片掉第二个反斜杠(r'1\nb\tc\\'[:-1])、手动添加一个反斜杠(r'1\nb\tc' + '\\')，或者忽略raw字符串语法并在常规字符串中把反斜

杠改为双反斜杠('1\\nb\\tc\\')。以上三种形式都会创建同样的8字符的字符串，其中包含3个反斜杠。

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（在一些界面中，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把所有在三重引号之内的文本收集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。要查看带有换行解释的字符串，打印出它，而不是回应：

```
>>> print(mantra)
Always look
  on the bright
side of life.
```

三重引号字符串在程序需要输入多行文本的任何时候都是很有用的。例如，嵌入多行错误信息或在源文件中编写HTML或XML代码。我们能够直接在脚本中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                                # Disable this code temporarily
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，但是这对性能来说也许没有什么显著影响。对于大段的代码，这也比手动在每一行之前加入井号，之后再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。

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'
>>> '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预声明，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。

注3：与C字符数组不同的是，使用Python字符串时，你不用分配或管理存储数组，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）。

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：这里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循环迭代，并使用in表达式操作符对字符和子字符串进行成员关系的测试，这实际上是一种搜索。对于子字符串，in很像是本章稍后介绍的str.find()方法，但是，它返回一个布尔结果而不是子字符串的位置：

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')          # Step through items
...

h a c k e r
>>> "k" in myjob                               # Found
True
>>> "z" in myjob                               # Not found
False
>>> 'spam' in 'abcspamdef'                     # Substring search, no position returned
True
```

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讨论类似的迭代工具以及表7-1中列出的其他内容（特别是在本书第14章和第20章中涉及的内容）。

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。能够将负偏移看做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四个字符的字符串，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}。

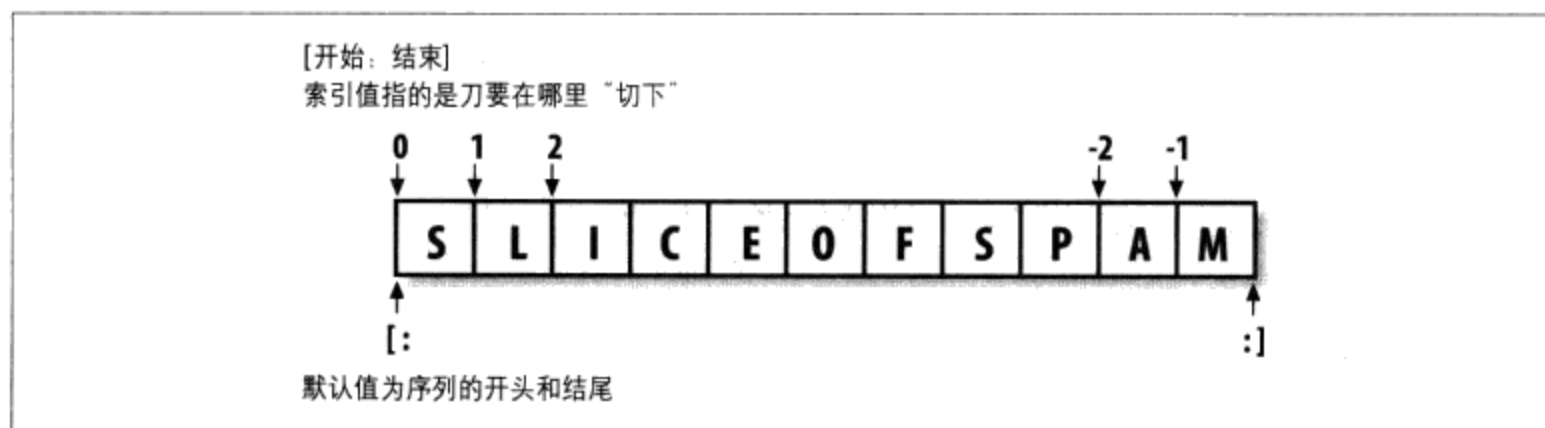


图7-1：偏移和分片：位置偏移从左至右（偏移0为第一个元素），而负偏移是由末端右侧开始计算（偏移-1为最后一个元素）。这两种偏移均可以在索引及分片中作为所给出的位置

上边的例子中的最后一行对分片进行了演示，这是索引的一种通用形式，返回的是整个一部分，而不是一个单个的项。也许理解分片最好的办法就是将其看做解析（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。

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来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。左边的偏移作为下边界（包含下边界在内），而右边的偏移作为上边界（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

注4： 具有数学思维的读者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。

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。接着，`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，我将会介绍如何通过一个分片（不像是字符串一样不可改变）进行赋值从而改变一个特定对象的一部分内容。下面概括一些细节以供参考：

- 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]` 获取了从偏移为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，但对于可以在原地修改的对象来说却很实用，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)。步进添加到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 = 'abcdefg'
>>> 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）。我们将会在本书稍后部分再次学习这种三重限制的分片，届时将与for循环配合使用。

在本书稍后，我们将看到分片等同于用一个分片对象进行索引，这对于那些试图两种操作都支持的类编写者来说，是一个重要的发现：

```
>>> 'spam'[1:3]                                # Slicing syntax
'pa'
>>> 'spam'[slice(1, 3)]                        # Slice objects
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, -1)]
'maps'
```

为什么要在意：分片

贯穿本书始末，经常会使用边栏（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，这些边栏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做是预览的途径，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）：

```
>>> "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，这种转换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。

接着要做的就是，如果脚本从文件或用户界面得到了一个作为文本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)                     # Convert to as-code string
'42'
```

int函数将字符串转换为数字，而str函数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（以及之前的反引号表达式，在Python 3.0中删除了）也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，其结果需用引号括起来。

```
>>> print(str('spam'), repr('spam'))
('spam', "'spam'")
```

参见本书第5章的“数字显示格式”一节中的“str和repr显示格式”部分了解关于这一主题的更多内容。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
```

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。随后我们将对格式进行讨论。

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如下的数学运算：

```
>>> 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（本书第4章介绍了循环语句，本书后续的部分将更深入地介绍它），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'                                # Convert binary digits to integer with ord
>>> I = 0
>>> while B != '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
... 
```

```
>>> I
13
```

左移运算 ($I \ll 1$) 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，并且我们在第5章遇到的内置函数 `int` 和 `bin` 在 Python 2.6 和 Python 3.0 中用来处理二进制转换任务，那么建议把这部分程序当作一个试验来运行：

```
>>> int('1101', 2)          # Convert binary to integer: built-in
13
>>> bin(13)                 # Convert integer to binary
'0b1101'
```

只要时间足够，未来的 Python 倾向于把大多数任务自动化。

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原地修改一个字符串（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，还要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做是对原字符串的“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。下面的两种方式都把对象替换为字符串，在某种意义上，是把对象转换为字符串并且根据指定的格式来改变最初的字符串：

```
>>> 'That is %d %s bird!' % (1, 'dead')           # Format expression
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead')     # Format method in 2.6 and 3.0
'That is 1 dead bird!'
```

尽管用替换这个词来比喻，但格式化的结果是一个新的字符串对象，而不是修改后的对象。我们将在本章稍后学习格式化，那时候，你会发现，格式化比这个例子所展示的更为通用和有用。由于上述调用的第二个作为一个方法提供，在深入介绍格式化之前，让我们先来看看字符串方法调用。

注意：正如我们将在第36章见到的，Python 3.0和Python 2.6引入了一种叫做`bytearray`的新字符串类型，它是可以修改的，因此可以修改其中的值。`bytearray`对象并不是真正的字符串，它们是较小的、8位整数的序列。然而，它们支持和常规字符串相同的大多数操作，并且显示的时候打印为ASCII字符。同样，它们为必须频繁修改的大量文本提供了另一个选项。在第36章中，我们还将看到`ord`和`chr`处理Unicode字符，这是不能存储在单个字节中的字符。

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，表达式和内置函数可能在不同范围的类型有效，但方法通常特定于对象类型，例如，字符串方法仅适用于字符串对象。Python 3.0中某些类型的方法集有所交叉（例如，很多类型都有一个`count`方法），但它们仍然比其他的工具更加特定于类型。

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”。

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。

表7-3概括了Python 3.0中内置字符串对象的方法及调用模式；这些经常有变化，因此，确保查看Python的标准库手册以获取最新的列表，或者在交互模式下在任何字符串上调用help。Python 2.6的字符串方法变化很小，例如，它包含一个decode，因为它对Unicode数据的处理有所不同（我们将在本书第36章中讨论Unicode数据）。在这个表中，S是一个字符串对象，并且可选的参数包含在方括号中。这个表中的字符串方法实现了分隔和连接、大小写转换、内容测试、子字符串查找和替换这样的高级操作。

表7-3：Python 3.0中的字符串方法调用

S.capitalize()	S.ljust(width [, fill])
S.center(width [, fill])	S.lower()
S.count(sub [, start [, end]])	S.lstrip([chars])
S.encode([encoding [,errors]])	S.maketrans(x[, y[, z]])
S.endswith(suffix [, start [, end]])	S.partition(sep)
S.expandtabs([tabsize])	S.replace(old, new [, count])
S.find(sub [, start [, end]])	S.rfind(sub [,start [,end]])
S.format(fmtstr, *args, **kwargs)	S.rindex(sub [, start [, end]])
S.index(sub [, start [, end]])	S.rjust(width [, fill])
S.isalnum()	S.rpartition(sep)
S.isalpha()	S.rsplit([sep[, maxsplit]])
S.isdecimal()	S.rstrip([chars])
S.isdigit()	S.split([sep [,maxsplit]])
S.isidentifier()	S.splitlines([keepends])
S.islower()	S.startswith(prefix [, start [, end]])
S.isnumeric()	S.strip([chars])

<code>S.isprintable()</code>	<code>S.swapcase()</code>
<code>S.isspace()</code>	<code>S.title()</code>
<code>S.istitle()</code>	<code>S.translate(map)</code>
<code>S.isupper()</code>	<code>S.upper()</code>
<code>S.join(iterable)</code>	<code>S.zfill(width)</code>

正如你所看到的，确实有一些字符串方法，我们没有足够的篇幅全部介绍，参见Python的库手册或参考文献来了解所有的细节。为了帮助你入门，让我们来看一些展示了最常用的方法的代码，并随之介绍Python文本处理的一些基础知识。

字符串方法实例：修改字符串

我们知道字符串是不可变的，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，你可以如下代码来完成：

```
>>> 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。它的参数是原始子字符串（任意长度）和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。注意到这次我们直接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。如同我们在前面所见到的，这就像是`in`表达式中的一次子字符串查找操作，但是，`find`返回的是所找到的字符串的位置：

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')        # Replace all
'xxxxEGGSxxxxEGGSxxxx'

>>> 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）以任意序列中的元素创立一个新的列表——在这个例子中，它将字符串的字符“打散”为一个列表。一旦字符串以这样的形式出现，你无需在每次修改后进行复制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。一般地，对任何字符串分隔符和可迭代字符串都会是这样的结果：

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

实际上，像这样一次性连接子字符串比单独地合并每一个要快很多。请确保阅读了前面关于Python 3.0和Python 2.6中可变的bytearray字符串的提示，第36章还将详细介绍。由于有些情况下它会改变，它还为此种列表/连接合并操作提供了一种替换，以针对某些必须经常修改的文本。

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——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：

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

这组数据出现在固定偏移处，因此可以通过分片从原始字符串分出来。这一技术称为解析，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，你就可以使用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隔（因此有时分解）一个字符串，这个字符串是使用某些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']
```

```
['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或起始的子字符串：

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
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 say 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
```

还请参阅本章稍后介绍的字符串格式化方法format，它提供了更高级的替换工具，在一个单个的步骤中组合多个操作。因为字符串有很多方法可以使用，我不会逐一介绍。你会在本书后边见到一些其他的字符串例子，如果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。也可以查看`help(S.method)`的结果来得到关于任何字符串对象S的方法的更多提示。

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（第36章给出了一个更进一步的例子）。虽然，有这方面的限制，但字符串方法有时与`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（在Python 3.0中删除）

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一直保留着它。

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，最初的模块调用形式已经在Python 3.0中删除了。然而，因为你还是会在较旧的Python代码中看见这个模块，因此在这里要简单地看一下。

这种历史问题的结果就是，在Python 2.6中，从技术上来说，有两种方式可以启用高级的字符串操作：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，设定变量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'
```

在Python 2.6中，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，除了模块调用已经从Python 3.0版删除以外，还有一个理由，那就是模块调用需要你导入string模块（而方法调用不需要导入）。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模块本身保存在了Python 3.0中（而没有其同等的字符串方法），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，除非你真的想把代码从Python 2.6修改为使用Python 3.0，否则，就应该放弃基本字符串运算调用。

字符串格式化表达式

尽管已经掌握了所介绍的字符串方法和序列操作，Python还提供了一种更高级的方法来组合字符串处理任务——字符串格式化允许在一个单个的步骤中对一个字符串执行多个特定类型的替换。它不是严格必须的，但它很方便使用，特别是当格式化文本以显示给程序的用户的时候。由于Python世界中充满了很多新思想，如今的Python中的字符串格式化可以以两种形式实现：

字符串格式化表达式

这是从Python诞生的时候就有的最初的技术；这是基于C语言的“printf”模型，并且在大多数现有的代码中使用。

字符串格式化方法调用

这是Python 2.6和Python 3.0新增加的技术，这是Python独有的方法，并且和字符串格式化表达式的功能有很大重叠。

由于方法调用是很新，其中的某些或另外一些可能会随着时间的推移而废弃。表达式更有可能在以后的Python版本中废弃，尽管这应该取决于真正的Python程序员未来的实际使用情况。然而，由于它们很大程度上只是同一个方案的变体，现在这两种技术都是有效的。既然字符串格式化表达式是最初的方法，让我们从它开始。

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，%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，而不是构建并组合单个的部分。

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（或多个，嵌入到元组中）对象，这些对象将会插入到左侧想让Python进行格式化字符串的一个（或多个）转换目标的位置上去。

例如，在上一个格式化示例中，整数1替换在格式化字符串左边的%d，字符串'dead'替换%s。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：

```
>>> 'That is %d %s bird!' % (1, 'dead')           # Format expression
That is 1 dead bird!
```

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'，代替标记%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）。%格式化表达式操作符在其右边期待一个单独的项或者一个或多个项的元组。

第三个例子同样是插入三个值：一个整数、一个浮点数对象和一个列表对象。但是注意到所有目标左侧都是%s，这就表示要把它们转换为字符串。由于对象的每个类型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。如前所述，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表达式

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7-4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、%f和%g都可以用于浮点数的格式化。

表7-4：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
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]typecode
```

表7-4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。*width*和*percision*都可以编码为一个*，以指定它们应该从输入值的下一项中取值。

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（%E和%e相同，只不过指数是大写表示的）：

```
>>> x = 1.23456789
>>> x
1.2345678899999999
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'
>>> '%E' % x
'1.234568E+00'
```

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')
```

如果在运行的时候才知道大小，你可以在格式化字符串中用一个*来指定通过计算得出width和precision，从而迫使它们的值从%运算符右边的输出中的下一项获取，在这里，元组中的4指定为precision：

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

如果你对这一功能感兴趣，可以自行体验这些例子和操作以了解更多信息。

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，并提取它们相应的值。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：

```
>>> reply = """                                # Template with substitution targets
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40}          # Build up values to substitute
>>> print(reply % values)                       # Perform substitutions

Greetings...
Hello Bob!
Your age squared is 40
```

这样的小技巧也常与内置函数vars配合使用，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。第5章中也有几个使用%x和%o格式化目标代码来转换十六进制和八进制的字符串的例子。

字符串格式化调用方法

正如前面提到的，Python 2.6和Python 3.0引入了一种新的方式来格式化字符串，在某些人看来，这种方式更特定于Python。和格式化表达式不同，格式化方法调用不是紧密地基于C语言的“printf”模型，并且它们的意图更详细而明确。另一方面，新的技术仍然

依赖于一些“printf”概念，例如，类型代码和格式化声明。此外，它和格式化表达式有很大程度的重合，并且有时比格式化表达式需要更多的代码，且在高级用途的时候很复杂。正因如此，目前在表达式和方法调用之间没有最佳使用建议，所以大多数程序员只需对这两种方案有基本的理解。

基础知识

简而言之，Python 2.6和Python 3.0（及其以后版本）中的新的字符串对象的format方法使用主体字符串作为模板，并且接受任意多个表示将要根据模板替换的值的参数。在主体字符串中，花括号通过位置（例如，{1}）或关键字（例如，{food}）指出替换目标及将要插入的参数。正如我们在第18章中深入学习参数时将学习到的，函数和方法的参数可以使用位置或关键字名称来传递，并且Python收集任意多个位置和关键字参数的能力允许这种通用的方法调用模式。例如，在Python 2.6和Python 3.0中：

```
>>> template = '{0}, {1} and {2}'                                # By position
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}'                      # By keyword
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}'                          # By both
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'
```

本质上，字符串也可以是创建一个临时字符串的常量，并且任意的对象类型都可以替换：

```
>>> '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
'3.14, 42 and [1, 2]'
```

就像%表达式和其他字符串方法一样，format创建并返回一个新的字符串对象，它可以立即打印或保存起来方便以后使用（别忘了，字符串是不可变的，因此，format必须创建一个新的对象）。字符串格式化不只是用来显示：

```
>>> X = '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
>>> X
'3.14, 42 and [1, 2]'

>>> X.split(' and ')
['3.14, 42', '[1, 2]']

>>> Y = X.replace('and', 'but under no circumstances')           *
>>> Y
'3.14, 42 but under no circumstances [1, 2]'
```

添加键、属性和偏移量

像%格式化表达式一样，格式化调用可以变得更复杂以支持更多高级用途。例如，格式化字符串可以指定对象属性和字典键——就像在常规的Python语法中一样，方括号指定字典键，而点表示位置或关键字所引用的一项的对象属性。如下例子中的第一个，索引字典上的键“spam”，然后从已经导入的sys模块对象获取“platform”属性。第二个例子做同样的事情，但是，通过关键字而不是位置指定对象：

```
>>> import sys

>>> 'My {1[spam]} runs {0.platform}'.format(sys, {'spam': 'laptop'})
'My laptop runs win32'

>>> 'My {config[spam]} runs {sys.platform}'.format(sys=sys,
                                                    config={'spam': 'laptop'})
'My laptop runs win32'
```

格式化字符串中的方括号可以指定列表（及其他的序列）偏移量以执行索引，但是，只有单个的正的偏移才能在格式化字符串的语法中有效，因此，这一功能并不是像你想的那样通用。和%表达式一样，要指定负的偏移或分片，或者使用任意表达式，必须在格式化字符串自身之外运行表达式：

```
>>> somelist = list('SPAM')
>>> somelist
['S', 'P', 'A', 'M']

>>> 'first={0[0]}, third={0[2]}'.format(somelist)
'first=S, third=A'

>>> 'first={0}, last={1}'.format(somelist[0], somelist[-1])      # [-1] fails in fmt
'first=S, last=M'

>>> parts = somelist[0], somelist[-1], somelist[1:3]             # [1:3] fails in fmt
>>> 'first={0}, last={1}, middle={2}'.format(*parts)
'first=S, last=M, middle=['P', 'A']"
```

添加具体格式化

另一种和%表达式类似的是，可以在格式化字符串中添加额外的语法来实现更具体的层级。对于格式化方法，我们在替换目标的标识之后使用一个冒号，后面跟着可以指定字段大小、对齐方式和一个特定类型编码的格式化声明。如下是可以在一个格式字符串中作为替代目标出现的形式化结构：

```
{fieldname!conversionflag:formatspec}
```

在这个替代目标语法中：

- *fieldname*是指定参数的一个数字或关键字，后面跟着可选的“.name”或“[index]”成分引用。
- *Conversionflag*可以是r、s，或者a分别是在该值上对repr、str或ascii内置函数的一次调用。
- *Formatspec*指定了如何表示该值，包括字段宽度、对齐方式、补零、小数点精度等细节，并且以一个可选的数据类型编码结束。

冒号后的*formatspec*组成形式上的描述如下（方括号表示可选的组成，并且不能编写为常量）：

```
[[fill]align][sign][#][0][width][.precision][typecode]
```

*align*可能是<、>、=或^，分别表示左对齐、右对齐、一个标记字符后的补充或居中对齐。*Formatspec*也包含嵌套的、只带有{}的格式化字符串，它从参数列表动态地获取值（和格式化表达式中的*很相似）。

参见Python的库手册可以了解关于替换语法的更多信息和可用的类型编码的列表，它们几乎与前面表7-4中列出的以及%表达式中使用的那些完全重合，但是格式化方法还允许一个“b”类型编码用来以二进制格式显示整数（它等同于使用bin内置函数），允许一个“%”类型编码来显示百分比，并且使用唯一的“d”表示十进制的整数（而不是“i”或“u”）。

例如，下面的{0:10}意味着一个10字符宽的字段中的第一个位置参数，{1:<10}意味着第2个位置参数在一个10字符宽度字段中左对齐，{0.platform:>10}意味着第一个参数的platform属性在10字符宽度的字段中右对齐：

```
>>> '{0:10} = {1:10}'.format('spam', 123.4567)
'spam      =    123.457'

>>> '{0:>10} = {1:<10}'.format('spam', 123.4567)
'      spam = 123.457 '

>>> '{0.platform:>10} = {1[item]:<10}'.format(sys, dict(item='laptop'))
'      win32 = laptop '
```

在格式化方法调用中，浮点数支持与%表达式中相同的类型代码和格式化声明。例如，下面的{2:g}表示，第三个参数默认地根据“g”浮点数表示格式化，{1:.2f}指定了带有2个小数位的“f”浮点数格式，{2:06.2f}添加一个6字符宽度的字段并且在左边补充0：

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
```

```
'3.141590, 3.14, 003.14'
```

格式化方法也支持十六进制、八进制和二进制格式。实际上，字符串格式化是把整数格式化为指定的进制的某些内置函数的替代方法：

```
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)           # Hex, octal, binary
'FF, 377, 11111111'

>>> bin(255), int('11111111', 2), 0b11111111              # Other to/from binary
('0b11111111', 255, 255)

>>> hex(255), int('FF', 16), 0xFF                          # Other to/from hex
('0xff', 255, 255)

>>> oct(255), int('377', 8), 0o377, 0377                  # Other to/from octal
('0377', 255, 255, 255)                                   # 0377 works in 2.6, not 3.0!
```

格式化参数可以在格式化字符串中硬编码，或者通过嵌套的格式化语法从参数列表动态地获取，后者很像是格式化表达式中的星号语法：

```
>>> '{0:.2f}'.format(1 / 3.0)                               # Parameters hardcoded
'0.33'
>>> '%.2f' % (1 / 3.0)
'0.33'

>>> '{0:.{1}f}'.format(1 / 3.0, 4)                          # Take value from arguments
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)                                    # Ditto for expression
'0.3333'
```

最后，Python 2.6和Python 3.0还提供了一种新的内置format函数，它可以用来格式化一个单独的项。它是字符串格式化方法的一种更简洁的替代方法，并且大致类似于用%格式化表达式来格式化一个单独的项：

```
>>> '{0:.2f}'.format(1.2345)                                # String method
'1.23'
>>> format(1.2345, '.2f')                                   # Built-in function
'1.23'
>>> '%.2f' % 1.2345                                         # Expression
'1.23'
```

从技术上讲，内置函数format运行主体对象的__format__方法，对于每个被格式化项目，str.format方法都是内部的。它仍然比最初的%表达式的对等体要冗长，这引发了我们在下一小节中讨论的话题。

与%格式化表达式比较

如果你仔细地学习了前面的小节，可能会注意到，至少对于位置的引用和字典键，字符串格式化方法看上去和%格式化表达式很像，特别是提前使用类型代码和额外的格式化

语法。实际上，在通常的使用情况下，格式化表达式可能比格式化方法调用更容易编写，特别是当使用通用的%s打印字符串替代目标的时候：

```
print('%s=%s' % ('spam', 42))          # 2.X+ format expression
print('{0}={1}'.format('spam', 42))    # 3.0 (and 2.6) format method
```

稍后我们将看到，更复杂的格式化倾向于降低复杂性（复杂的任务通常都很困难，不管用什么方法），并且某些人把格式化方法看做是很大程度的冗余。

另一方面，格式化方法还提供了一些潜在的优点。例如，最初的%表达式无法处理关键字、属性引用和二进制类型代码，尽管%格式化字符串中的字典键引用常常能够达到类似的目标。要看看两种技术是如何重合的，把如下的%表达式与前面显示的对等的格式化方法调用进行比较：

```
# The basics: with % instead of format()

>>> template = '%s, %s, %s'
>>> template % ('spam', 'ham', 'eggs')          # By position
'spam, ham, eggs'

>>> template = '%(motto)s, %(pork)s and %(food)s'
>>> template % dict(motto='spam', pork='ham', food='eggs')  # By key
'spam, ham and eggs'

>>> '%s, %s and %s' % (3.14, 42, [1, 2])        # Arbitrary types
'3.14, 42 and [1, 2]'

# Adding keys, attributes, and offsets

>>> 'My %(spam)s runs %(platform)s' % {'spam': 'laptop', 'platform': sys.platform}
'My laptop runs win32'

>>> 'My %(spam)s runs %(platform)s' % dict(spam='laptop', platform=sys.platform)
'My laptop runs win32'

>>> somelist = list('SPAM')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
'first=S, last=M, middle=[\'P\', \'A\']'
```

当较为复杂的格式化应用两种技术方法进行复杂性的比较的时候，把如下的代码与前面列出的对等的格式化方法调用比较，你将会发现%表达式往往能够更简单一些并且更简练：

```
# Adding specific formatting

>>> '%-10s = %10s' % ('spam', 123.4567)
'spam      =    123.4567'

>>> '%10s = %-10s' % ('spam', 123.4567)
```

```

'    spam = 123.4567 '

>>> '%(plat)10s = %(item)-10s' % dict(plat=sys.platform, item='laptop')
'    win32 = laptop '

# Floating-point numbers

>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'

# Hex and octal, but not binary

>>> '%x, %o' % (255, 255)
'ff, 377'

```

格式化方法拥有%表达式所没有的很多高级功能，但是，更复杂的格式化似乎看起来也能从根本上降低复杂性。例如，如下的代码展示了用两种技术产生同样的结果，带有字段大小和对齐以及各种参数引用方法：

```

# Hardcoded references in both

>>> import sys

>>> 'My {1[spam]:<8} runs {0.platform:>8}'.format(sys, {'spam': 'laptop'})
'My laptop    runs    win32'

>>> 'My %(spam)-8s runs %(plat)8s' % dict(spam='laptop', plat=sys.platform)
'My laptop    runs    win32'

```

实际上，相比执行那些事先构建起一组替代数据的代码（例如，一次性把用来替代的数据收集到一个HTML模板中），程序不太可能像这样硬编码引用。当我们考虑像这个例子中的共同之处的时候，格式化方法和%表达式之间的比较更加直接（正如我们将在第18章所见到的，这里方法调用中的**数据是特殊的语法，它把键和值的一个字典包装到单个的“name=value”关键字参数中，以便可以在格式化字符串中用名字来引用它们）：

```

# Building data ahead of time in both

>>> data = dict(platform=sys.platform, spam='laptop')

>>> 'My {spam:<8} runs {platform:>8}'.format(**data)
'My laptop    runs    win32'

>>> 'My %(spam)-8s runs %(platform)8s' % data
'My laptop    runs    win32'

```

随着时间的推移，Python社区将决定，这三种方法（使用%表达式、格式化方法调用、带

有两种技术的工具集)哪种更好。自行实验这些技术,将会更好地体会到它们能够带来些什么,查看Python 2.6和Python 3.0的库手册可以了解更多细节。

注意: Python 3.1中的字符串格式化方法扩展:即将发布的Python 3.1(在编写本章时正处在alpha版阶段)将添加针对数字的千分隔位语法,它在3位一组之间插入逗号。在类型代码前添加一个逗号将会使其生效,如下所示:

```
>>> '{0:d}'.format(999999999999)
'999999999999'

>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

如果没有显式地包含相对数目的话,Python 3.1还将自动为替换目标分配相对数,尽管使用这一扩展可能会消除格式化方法的主要优点,参见下面小节介绍:

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'

>>> '{:,d} {:,d}'.format(9999999, 8888888)
'9,999,999 8,888,888'

>>> '{:,.2f}'.format(296999.2567)
'296,999.26'
```

本书并不正式介绍Python 3.1,因此,你可以将此视为预览。Python 3.1还将解决Python 3.0中与文件输入/输出操作速度相关的一个重要性能问题,这个问题造成Python 3.0对很多类型的程序不切实际。参见Python 3.1的发布声明来了解更多细节。参见第24章中的*formats.py*逗号插入以及货币格式化函数示例来了解一个手动解决方案,在Python 3.1发布之前可以将其导入并使用。

为什么用新的格式化方法

我们已经对两种格式化方法做了许多比较,我需要说明一下,为什么有时可能想要考虑使用格式化方法。简而言之,尽管格式化方法有时候需要更多的代码,它还是:

- 拥有%表达式所没有的一些额外功能。
- 可以更明确地进行替代值引用。
- 考虑到操作符会有一个更容易记忆的方法名。
- 不支持用于单个和多个替代值大小写的不同语法。

尽管这两种技术现在都可用,并且格式化表达式的应用也很广泛,格式化方法可能最终会囊括它。因为当前仍然需要作出选择,所以我们继续简单地讨论一下二者的区别,然后再继续学习。

额外功能

方法调用支持表达式所没有的一些额外功能，例如二进制类型编码和（Python 3.1中引入的）千分位分组。此外，方法调用支持直接的键和属性引用。正如我们已经见到的，格式化表达式通常可以以其他方法实现同样的效果：

```
>>> '{0:b}'.format((2 ** 16) - 1)
'1111111111111111'

>>> '%b' % ((2 ** 16) - 1)
ValueError: unsupported format character 'b' (0x62) at index 1

>>> bin((2 ** 16) - 1)
'0b1111111111111111'

>>> '%s' % bin((2 ** 16) - 1)[2:]
'1111111111111111'
```

参见前面的例子中对%表达式中基于字典的格式化和格式化方法中的键和属性引用的比较，特别是在一些共同实践中，这两者很大程度上似乎是一个主题的两变体。

显式值引用

格式化方法存在一个颇具争议的使用情况——很多值都要替换到格式化字符串中。例如，我们将在第30章遇到的`lister.py`类示例，把6个项替换到一个单独的字符串中，并且，在这个例子中，方法的`{i}`位置标签似乎比表达式的`%s`更易读：

```
'\n%s<Class %s, address %s:\n%s%s>\n' % (...)           # Expression
'\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(...) # Method
```

另一方面，在%表达式中使用字典键可能会大大减少这一差异的程度。这也是格式化复杂性的一个最坏情况例子的一部分，并且不是很常见的情况，很多典型的用例很大程度上是有随机性的。此外，在Python 3.1中（alpha版阶段），数字替换值将会变为可选的，因此，会推翻这一传说中的好处：

```
C:\misc> C:\Python31\python
>>> 'The {0} side {1} {2}'.format('bright', 'of', 'life')
'The bright side of life'
>>>
>>> 'The {} side {} {}'.format('bright', 'of', 'life')           # Python 3.1+
'The bright side of life'
>>>
>>> 'The %s side %s %s' % ('bright', 'of', 'life')
'The bright side of life'
```

像这样使用Python 3.1的自动相对计数似乎会取消这一方法的大部分优点。例如，在浮点数格式化上比较这一效果，格式化表达式更为精确，并且似乎要整齐一些：

```

C:\misc>C:\Python31\python
>>> '{0:f}, {1:.2f}, {2:05.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 03.14'
>>>
>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
>>>
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'

```

方法名和通用参数

有了Python 3.1的这一自动技术修改，格式化方法唯一明确保留的潜在优点就是它用一个更加便于记忆的格式化方法名替代了%操作符，并且不区分单个和多个替代值。前者可能会使得一个方法对初学者来说乍看上去很容易（“format”可能比多个“%”字符更容易理解），尽管这在调用的时候太主观化。

后一个不同可能更显著——使用格式化表达式，单个值可以独自给定，但多个值必须放入一个元组中：

```

>>> '%.2f' % 1.2345
'1.23'
>>> '%.2f %s' % (1.2345, 99)
'1.23 99'

```

从技术上讲，格式化表达式接受一个单个替换值，或者一项或多项的元组。实际上，由于单个项可以独自给定，也可以在元组中给定，要格式化的元组必须作为嵌套的元组提供：

```

>>> '%s' % 1.23
'1.23'
>>> '%s' % (1.23,)
'1.23'
>>> '%s' % ((1.23,))
'(1.23,)'

```

另一方面，格式化方法通过在两种情况下接受通用的函数参数，把这两种情况绑定到一起：

```

>>> '{0:.2f}'.format(1.2345)
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99)
'1.23 99'
>>> '{0}'.format(1.23)
'1.23'
>>> '{0}'.format((1.23,))
'(1.23,)'

```

因此，对于初学者来说，不易混淆，并且很少引发编程错误。这仍然是一个相对较小的

问题，如果你总是把值包含在一个元组中并忽略非元组选项，表达式基本上和这里的方法调用相同。此外，方法带来了另一个额外的代价，就是冗长的代码实现了有限的灵活性。由于表达式已经在Python中广泛使用，因此是否为了一个新的工具拆散已有的代码还尚存争论。

将来可能废弃

正如前面提到的，Python开发者可能在将来废弃%表达式而使用format方法，现在下这样的结论还有些风险。实际上，在Python 3.0的手册中有关于这一效果的一个提示。

当然，这种情况还没有发生，并且两种格式化技术在Python 2.6和Python 3.0（本书所介绍的两种版本）中都完全能够使用。即将发布的Python 3.1也支持这两种技术，因此，在可以预见的未来，任何一种似乎都不太可能废弃。此外，由于格式化表达式已经在目前为止所编写的几乎所有代码中广泛使用，在未来的很多年里，大多数程序员还将会因为两种技术都熟悉而受益匪浅。

如果这一废弃确实发生了，你可能需要重新把所有的%表达式编写为格式化方法，并且转换本书中提到的那些代码，从而使用一个更新的Python发布。依我个人之见，我希望这样的修改基于将来真正的Python程序员的常用实践，而不是一群核心开发者的一时兴起，特别是Python 3.0现在针对众多不兼容的修改关闭了大门。坦率地讲，这种废弃似乎可能用一种复杂的东西换来另一种复杂的东西，很大程度上等同于它将要替换的工具。如果你关心到未来Python发布的迁移，那么，确保不断地关注这一方面的发展。

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概念，这些概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。更正式的说法，在Python中有三个主要类型（以及操作）的分类：

数字（整数、浮点数、二进制、分数等）

支持加法和乘法等。

序列（字符串、列表、元组）

支持索引、分片和合并等。

映射（字典）

支持通过键的索引等。

集合是自成一体的一个分类（它们不会把键映射到值，并且没有逐位的排序顺序），我们还没有深入地学习映射（字典将会在下一章讨论）。但是我们遇到的很多其他类型都与数字和字符串类似。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的。如果一个对象类型是不可变的，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。Python中的主要核心类型划分为如下两类：

不可变类型（数字、字符串、元组、不可变集合）

不可变的分类中没有哪个对象类型支持原处修改，尽管我们总是可以运行表达式来创建新的对象并将其结果分配给变量。

可变类型（列表、字典、可变集合）

相反，可变的类型总是可以通过操作原处修改，而不用创建新的对象。尽管这样的对象可以复制，但原处修改支持直接修改。

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。要了解列表、字典和元组分别属于哪种类型，我们需要继续学习下一章。

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。正如前面所提到的，在本书的最后部分，我们将返回字符串模型并介绍Unicode文本和二进制数据的一些细节，这些内容对某些Python程序员而不是所有Python程序员有用。在继续学习之前，先来看本章的习题以复习这里所介绍的内容。

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，只能用于单一数据类型上。像X+Y这样的表达式和len(X)这样的内置函数是通用的，可以用于多种类型上。在这个例子中，in关系表达式和字符串查找具有类似的效果，但它可以用来查找字符串和列表。在Python 3.0中，有一些对方法分组的尝试（例如，不可变序列类型list和bytearray具有类似的方法集合），但方法仍然比其他的操作集更特定于类型。
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]`以逗号分隔字符串，再进行索引运算。通过交互模式亲自实验，看一下结果。
6. 6个。字符串`"a\n b\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。参见表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`模块已经弃用，Python 3.0完全删除其调用。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。现在，在非常陈旧的Python代码中，它才会出现。

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，你可以把它们看做组。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，所以可以创建列表的子列表的子列表等。

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：当应用于字符串上的合并和分片这样的操作应用于列表时，返回新的列表。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列表包含了零个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
<code>L = []</code>	一个空的列表
<code>L = [0, 1, 2, 3]</code>	四项：索引为0到3
<code>L = ['abc', ['def', 'ghi']]</code>	嵌套的子列表
<code>L = list('spam')</code>	可迭代项目的列表，连续整数的列表
<code>L = list(range(-4, 4))</code>	
<code>L[i]</code>	索引，索引的索引，分片，求长度
<code>L[i][j]</code>	
<code>L[i:j]</code>	
<code>len(L)</code>	

表8-1：常用列表常量和操作（续）

操作	解释
<code>L1 + L2</code>	合并
<code>L * 3</code>	重复
<code>for x in L: print(x)</code>	迭代，成员关系
<code>3 in L</code>	
<code>L.append(4)</code>	方法：增长，排序，搜索，插入，反转等
<code>L.extend([5,6,7])</code>	
<code>L.insert(I, X)</code>	
<code>L.index(1)</code>	
<code>L.Count(X)</code>	
<code>L.sort()</code>	
<code>L.reverse()</code>	
<code>del L[k]</code>	方法，语句：缩短
<code>del L[i:j]</code>	
<code>L.pop()</code>	
<code>L.remove(2)</code>	
<code>L[i:j] = []</code>	
<code>L[i] = 1</code>	索引赋值，分片赋值
<code>L[i:j] = [4,5,6]</code>	
<code>L = [x**2 for x in range(5)]</code>	列表解析（见13章和17章）
<code>list(map(ord, 'spam'))</code>	

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

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。

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。

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

由于列表是序列，它支持很多与字符串相同的操作。例如，列表对“+”和“*”操作的响应与字符串很相似，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：

```
% 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!']
```

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']
```

列表迭代和解析

更广泛地说，列表对于我们在上一章对字符串使用的所有序列操作都能做出响应，包括迭代工具：

```
>>> 3 in [1, 2, 3]           # Membership
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')    # Iteration
...
1 2 3
```

我们将在第13章更正式地讨论迭代和range内置函数，因为它们都与语句语法有关。简而言之，for循环从左到右地遍历任何序列中的项，对每一项执行一条或多条语句。

表8-1中的最后一项，列表解析和map调用在本书第14章中更详细地介绍，并且在本书第20章还会展开介绍。正如第4章所提到的，它们的基本操作是很简单的，列表解析只不过是通过对序列中的每一项应用一个表达式来构建一个新的列表的方式，它与for循环密切相关：

```
>>> res = [c * 4 for c in 'SPAM']           # List comprehensions
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

这个表达式功能上等同于手动构建一个结果的列表的一个for循环，但是，正如我们在本章稍后将要了解到的，列表解析的编码更简单，而且如今运行起来更快：

```
>>> res = []
>>> for c in 'SPAM':                         # List comprehension equivalent
...     res.append(c * 4)
...
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

正如第4章介绍的，内置函数map做类似的工作，但它对序列中的各项应用一个函数并把结果收集到一个新的列表中：

```
>>> list(map(abs, [-1, -2, 0, 1, 2]))        # map function across sequence
[1, 2, 0, 1, 2]
```

由于我们还没有准备好完整地介绍迭代，我们将进一步推迟介绍，但是，在本章稍后可以看到关于字典的一个类似的解析表达式。

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，它们支持原处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。

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及大多数其他语言极为相似——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}。

实际情况并非如此，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，分片赋值是一次性替换整个片段或“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），实际上那些才是Python程序员比较喜欢用的。

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，其中很多调用可以在原处修改主体列表：

```
>>> L.append('please')           # Append method call: add item at end
>>> 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

注2： 当赋值的值与分片重叠时，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。

端。与合并不同的是，`append`允许传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。

我们可以通过传入关键字参数来修改排序行为——这是指定按名称传递的函数调用中特殊的“`name=value`”语法，常常用来给定配置选项。在排序中，`key`参数给出了一个单个参数的函数，它返回在排序中使用的值，`reverse`参数允许排序按照降序而不是升序进行：

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                   # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)     # Change sort order
>>> L
['aBe', 'ABD', 'abc']
```

当排序字典的列表的时候，排序的`key`参数也很有用，可以通过索引每个字典挑选出一个排序键。我们将在本章稍后学习字典，并且将在本书第四部分了解关于关键字函数参数的更多内容。

注意： Python 3.0中的比较和排序：在Python 2.6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）——语言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，但是在Python 3.0中则会发生异常。

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。

Python 3.0也不再支持：传入一个任意的比较函数来进行排序以实现不同的顺序。建议的解决方法是使用`key=func`关键字参数在排序时编码值的转换，并且使用`reverse=True`关键字参数把排序顺序改为降序。这些都是过去比较函数的典型用法。

注意：要当心`append`和`sort`原处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。

一部分原因是这样的限制，排序也在最近的Python中可以作为内置函数使用了，它可以排序任何集合（不只是列表）并且针对结果返回一个新的列表（而不是原处修改）：

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)           # Sorting built-in
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)     # Pretransform items: differs!
['abe', 'abd', 'abc']
```

注意这里的最后一个例子——我们可以用一个列表解析在排序之前转换为小写，但是结果不包含最初的列表的值，因为这是用`key`参数来实现的。后者在排序中临时应用，而不会改变排序的值。随着我们进一步学习，将会看到这样的情况，内置函数`sorted`有时候比`sort`方法更有用。

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。也有一个`reversed`内置函数，像`sorted`一样地工作，但是，它必须包装在一个`list`调用中，因为它是一个迭代器（后面更详细地讨论迭代器）：

```
>>> L = [1, 2]
>>> L.extend([3,4,5])                               # Add many items at end
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                                          # Delete and return last item
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                                     # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))                               # Reversal built-in with a result
[1, 2, 3, 4]
```

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]

```

`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：

```

>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')       # Index of an object
1
>>> L.insert(1, 'toast')  # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')      # Delete by value
>>> L
['spam', 'toast', 'ham']
>>> L.pop(1)              # Delete by position
'toast'
>>> L
['spam', 'ham']

```

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:] = []
>>> L
['Already']

```

```
>>> L[0] = []
>>> L
[[]]
```

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）以及其他在前言中所提到的参考书籍。

还想再次提醒你，我们这里讨论的原处修改操作都只适用于可变对象：无论你怎么绞尽脑汁，都是不能用在字符串上的（或者是第9章中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。如果把列表看做是有序的对象集合，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，而这些在较低级的语言中你可能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。Python字典的主要属性如下。

通过键而不是偏移量来读取

字典有时又叫做关联数组（associative array）或者是散列表（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）。

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，所以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(散列表)

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，字典是作为散列表（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采用最优化的散列算法来寻找键，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
<code>D = {}</code>	空字典
<code>D= {'spam': 2, 'eggs': 3}</code>	两项目字典
<code>D= {'food': {'ham': 1, 'egg': 2}}</code>	嵌套
<code>D= dict.fromkeys(['a', 'b'])</code>	其他构造技术
<code>D= dict(zip(keyslst, valslst))</code>	关键字、对应的对、键列表
<code>D= dict(name='Bob', age=42)</code>	
<code>D ['eggs']</code>	以键进行索引运算
<code>D ['food']['ham']</code>	
<code>'eggs' in D</code>	成员关系：键存在测试
<code>D.keys()</code>	方法：键
<code>D.values()</code>	值
<code>D.items()</code>	键+值
<code>D.copy()</code>	副本
<code>D.get(key, default)</code>	默认

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）。

表8-2：常见字典常量和操作（续）

操作	解释
D.update(D2)	合并
D.pop(key)	删除等
len(D)	长度（储存的元素的数目）
D[key] = 42	新增/修改键，删除键
del D [key]	根据键删除条目
list(D.keys())	字典视图（Python 3.0）
D1.keys() & D2.keys()	
Dictionary views (Python 3.0)	
D = {x: x*2 for x in range(10)}	字典解析（Python 3.0）

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
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}           # Make a dictionary
>>> D['spam']                                       # Fetch a value by key
2
>>> D                                              # 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，使用相同的方括号语法，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（也就是散列查找），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。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，它能够返回存储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。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：

```
>>> len(D)                                # Number of entries in dictionary
3
>>> 'ham' in D                             # Key membership test alternative
True
>>> list(D.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14章和第20章进一步讨论迭代器。

还要注意上面列出的最后一个示例的语法。由于类似的原因，在Python 3.0中，我们必须将其放到一个list调用中——Python 3.0中的keys返回一个迭代器，而不是一个物理的列表。list调用迫使它一次生成所有的值，以便我们可以将其打印出来。在Python 2.6中，keys构建并返回一个真正的列表，因此，list调用不需要显示结果。更多细节将在本章稍后介绍。

注意：字典中的键的顺序是随意的，并且可以在版本之间变化，因此，如果你的字典打印出来的顺序与这里给出的不同，不必感到惊诧。实际上，对我来说顺序也变了——我在Python 3.0下运行所有这些示例，但是，它们的键和之前显示的版本有着不同的顺序。你不应该依赖于字典键的顺序，不管是程序中的还是图书中的。

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：

```
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}
```



```

>>> D['ham'] = ['grill', 'bake', 'fry']          # Change entry
>>> D
{'eggs': 3, 'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> del D['eggs']                                # Delete entry
>>> D
{'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> D['brunch'] = 'Bacon'                        # Add new entry
>>> D
{'brunch': 'Bacon', 'ham': ['grill', 'bake', 'fry'], 'spam': 2}

```

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（和keys一样，在Python 3.0中将它们放入到一个list调用中，来收集它们的值以显示）：

```

>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> list(D.values())
[3, 1, 2]
>>> list(D.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：

```

>>> D.get('spam')                                # A key that is there
2
>>> print(D.get('toast'))                        # A key that is missing
None
>>> D.get('toast', 88)
88

```

字典的update方法有点类似于合并，但是，它和从左到右的顺序无关（再一次强调，字典中没有这样的事情）。它把一个字典的键和值合并到另一个字典中，盲目地覆盖相同键的值：

```

>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}

```

```
>>> D2 = {'toast':4, 'muffin':5}
>>> D.update(D2)
>>> D
{'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
>>> D
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
>>> D.pop('muffin')
5
>>> D.pop('toast')           # Delete and return from a key
4
>>> D
{'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:           # Same as: 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，这条命令只不过是通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。

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，类似串联（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（我们将在本书第六部分进行讨论）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所需预先分配足够大的列表（例如，`[0]*100`），但你也可以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是一个有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，然而我们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：

```
>>> if (2,3,6) in Matrix:           # Check for key before fetch
...     print(Matrix[(2,3,6)])       # See Chapter 12 for if/else
... else:
...     print(0)
...
0
>>> try:
...     print(Matrix[(2,3,6)])        # Try to index
... except KeyError:                 # Catch and recover
...     print(0)                      # See Chapter 33 for try/except
...
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：

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
>>> mel['home']['zip']
80513
```

尽管我们将来在本书第四部分学习这些类（它们既按照数据也按照逻辑分组）时，它可能比这里用做记录的用途更好，但字典是满足简单需求的一种易用的工具。

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 'name' in form:
    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。

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)              # dict keyword argument form

dict([('name', 'mel'), ('age', 45)])  # dict key/value tuples form
```

这四种形式都会建立相同的两键字典，但它们在不同的条件下有用：

- 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。

在前面排序的时候，我们遇到了关键字参数，这段代码中展示的第三种形式已经在如今的Python代码中特别流行，因为它语法简单（因此，也不太容易出错）。如前面的表8-2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的不同列表合并在一起（例如，分析数据文件的列）。下一节将更详细地介绍这一选项。

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。

本节中的列表介绍了在Python 2.6和Python 3.0中创建字典的各种方式。然而，还有创建字典的另一种方式，仅在Python 3.0（及其以后的版本）中可用：字典解析表达式。要了解如何使用这最后一种形式，我们需要继续学习下一小节。

Python 3.0中的字典变化

本章到目前为止还是关注于在不同版本中可用的字典的基本知识，但是，字典的功能在Python 3.0中已经有所变化。如果你使用Python 2.X代码，可能会遇到表现不同的字典工具，或者干脆在Python 3.0中取消的字典工具。此外，Python 3.0代码也会使用在Python 2.0中所没有的工具。具体来说，Python 3.0中的字典：

- 支持一种新的字典解析表达式，这是列表和集合解析的“近亲”。
- 对于D.key、D.values和D.items方法，返回可迭代的视图，而不是列表。
- 由于前面一点，需要新的编码方式通过排序键来遍历。
- 不再直接支持相对大小比较——取而代之的是手动比较。
- 不再有D.has_key方法——相反，使用in成员关系测试。

让我们看看在Python 3.0中的字典有什么新特性。

字典解析

正如上一小节末尾提到的，Python 3.0中的字典也可以用字典解析来创建。就像我们在第5章中遇到的集合解析一样，字典解析也只在Python 3.0中可用（Python 2.6中不可用）。就像较早的时候我们在第4章中以及在本章开始简单介绍过的列表解析一样，字典解析也隐式地运行一个循环，根据每次迭代收集表达式的键/值结果，并且使用它们来填充一个新的字典。一个循环变量允许解析在过程中使用循环迭代值。

例如，在Python 2.6和Python 3.0中，动态初始化一个字典的标准方式都是：将其键和值对应起来并把结果传递给dict调用。正如我们将在本书第13章学习到的，zip函数是在一个单个调用中从键和值的列表来构建一个字典的方式之一。如果不能在代码中预计键和值的集合，总是可以将它们构建为列表然后再对应起来：

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))      # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]

>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))  # Make a dict from zip result
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

在Python 3.0中，可以用一个字典解析表达式来实现同样的效果。如下代码使用对应结果（它的样子和在Python代码中几乎相同，只不过更正式一些）中的每一个键/值对构建了一个新的字典：

```
C:\misc> c:\python30\python                  # Use a dict comprehension

>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
```

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

这个例子中，解析实际上需要更多的代码，但是，它们也比这个例子所暗示的要更为通用——我们可以使用它们把单独的一串值映射到字典，并且键和值一样，也可以用表达式来计算：

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}           # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'}                 # Loop over any iterable
>>> D
{'A': 'AAAA', 'P': 'PPPP', 'S': 'SSSS', 'M': 'MMMM'}

>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'ham': 'HAM!', 'spam': 'SPAM!'}
```

字典解析对于从键列表来初始化字典也很有用，这和我们在前一小节末尾遇到的 `fromkeys` 方法很相似：

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)           # Initialize dict from keys
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = {k:0 for k in ['a', 'b', 'c']}               # Same, but with a comprehension
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = dict.fromkeys('spam')                       # Other iterators, default value
>>> D
{'a': None, 'p': None, 's': None, 'm': None}

>>> D = {k: None for k in 'spam'}
>>> D
{'a': None, 'p': None, 's': None, 'm': None}
```

和相关的工具一样，字典解析支持这里没有介绍的额外的语法，包括嵌套循环和 `if` 子句。遗憾的是，要真正理解字典解析，我们还需要了解Python中有关迭代语句和概念的更多知识，并且，我们目前还没有足够的信息来介绍这些内容。我们将在第14章和第20章学习有关各种解析（列表解析、集合解析和字典解析）的更多知识，因此，稍后再介绍更多细节。我们还将在第13章中介绍循环的时候，再次学习本节中用到的 `zip` 内置函数。

字典视图

在Python 3.0中，字典的 `keys`、`values` 和 `items` 都返回视图对象，而在Python 2.6中，它们返回实际的结果列表。视图对象是可选代的，这就意味着对象每次产生一个结果项，而

不是在内存中立即产生结果列表。除了是可迭代的，字典视图还保持了字典成分的最初的顺序，反映字典未来的修改，并且能够支持集合操作。另一方面，它们不是列表，并且不支持像索引和列表`sort`方法这样的操作，打印的时候它们也不显示自己的项。

我们将在第14章更正式地讨论可迭代的概念，但是，现在只要知道这一点就够了：如果想要应用列表操作或者显示它们的值，我们必须通过内置函数`list`来运行这3个方法的结果：

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

>>> K = D.keys()                                # Makes a view object in 3.0, not a list
>>> K
<dict_keys object at 0x026D83C0>
>>> list(K)                                       # Force a real list in 3.0 if needed
['a', 'c', 'b']

>>> V = D.values()                               # Ditto for values and items views
>>> V
<dict_values object at 0x026D8260>
>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> K[0]                                          # List operations fail unless converted
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'a'
```

除非在交互提示模式显示结果，我们可能很少会注意到这一改变，因为Python中的循环结构会自动迫使可迭代的对象在每次迭代上产生一个结果：

```
>>> for k in D.keys(): print(k)                  # Iterators used automatically in loops
...
a
c
b
```

此外，Python 3.0的字典自己仍然拥有迭代器，它返回连续键——就像在Python 2.6中一样，它往往仍然没有必要直接调用`keys`：

```
>>> for key in D: print(key)                     # Still no need to call keys() to iterate
...
a
c
b
```

然而，和Python 2.X的列表结果不同，Python 3.0中的字典视图并非创建后不能改变——它们可以动态地反映在视图对象创建之后对字典做出的修改：

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

>>> K = D.keys()
>>> V = D.values()
>>> list(K)                                # Views maintain same order as dictionary
['a', 'c', 'b']
>>> list(V)
[1, 3, 2]

>>> del D['b']                             # Change the dictionary in-place
>>> D
{'a': 1, 'c': 3}

>>> list(K)                                # Reflected in any current view objects
['a', 'c']
>>> list(V)                                # Not true in 2.X!
[1, 3]
```

字典视图和几何

与Python 2.X中的列表结果不同，`keys`方法所返回的Python 3.0的视图对象类似于集合，并且支持交集和并集等常见的集合操作；`values`视图不是这样的，因为它们不是唯一的；但`items`结果是的，如果(`key`, `value`)对是唯一的并且可散列的话。由于集合的行为很像是无值的字典（并且甚至像Python 3.0中的字典一样编写在花括号中），这是一种符合逻辑的对称。就像字典键一样，集合的项是无序的、唯一的并且不可变。

如下是键列表用于集合操作中的样子。在集合操作中，视图可能与其他的视图、集合和字典混合（在这种环境中，字典与它们的键视图一样对待）：

```
>>> K | {'x': 4}                            # Keys (and some items) views are set-like
{'a', 'x', 'c'}

>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'

>>> D = {'a':1, 'b':2, 'c':3}
>>> D.keys() & D.keys()                      # Intersect keys views
{'a', 'c', 'b'}
>>> D.keys() & {'b'}                         # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1}                     # Intersect keys and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'}              # Union keys and set
{'a', 'c', 'b', 'd'}
```

如果字典项视图是可散列的话，它们是类似于集合的——也就是说，如果它们只包含不可变的对象的话：

```
>>> D = {'a': 1}
>>> list(D.items())           # Items set-like if hashable
[('a', 1)]
>>> D.items() | D.keys()      # Union view and view
{('a', 1), 'a'}
>>> D.items() | D             # dict treated same as its keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}    # Set of key/value pairs
{('a', 1), ('d', 4), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)}) # dict accepts iterable sets too
{'a': 1, 'c': 3, 'd': 4}
```

要了解集合操作的更多内容，参阅第5章。现在，让我们看看Python 3.0中字典的另外3个快速编码注意事项。

排序字典键

首先，由于keys不会返回一个列表，在Python 2.X中通过排序键来浏览一个字典的编码模式，在Python 3.0中并不适用。必须要么手动地转换为一个列表，要么在一个键视图或字典自身上使用第4章及本章前面介绍的sorted调用：

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

>>> Ks = D.keys()           # Sorting a view object doesn't work!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'

>>> Ks = list(Ks)           # Force it to be a list and then sort
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> Ks = D.keys()           # Or you can use sorted() on the keys
>>> for k in sorted(Ks): print(k, D[k])    # sorted() accepts any iterable
...                                       # sorted() returns its result
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}           # Better yet, sort the dict directly
```

```
>>> for k in sorted(D): print(k, D[k])           # dict iterators return keys
...
a 1
b 2
c 3
```

字典大小比较不再有效

其次，尽管在Python 2.6中可以直接用<、>等比较字典的相对大小，但在Python 3.0中这不再有效。然而，可以通过手动地比较排序后的键列表来模拟：

```
sorted(D1.items()) < sorted(D2.items())          # Like 2.6 D1 < D2
```

Python 3.0中字典相等性测试仍然有效。由于我们将在下一章详细讨论比较的时候再次回顾这一点，我们将在那里介绍更多细节。

has_key方法已死：in永生

最后，广为使用的字典has_key键存在测试方法在Python 3.0中取消了。相反，使用in成员关系表达式，或者带有默认测试的一个get（其中，in通常是首选的）：

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

>>> D.has_key('c')                               # 2.X only: True/False
AttributeError: 'dict' object has no attribute 'has_key'

>>> 'c' in D
True
>>> 'x' in D
False
>>> if 'c' in D: print('present', D['c'])          # Preferred in 3.0
...
present 3

>>> print(D.get('c'))
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c']) # Another option
...
present 3
```

如果你使用Python 2.6并且关心Python 3.0的兼容性，注意前两个改变（解析和视图）只能在Python 3.0中编码，但后3个（排序、手动比较和in）如今可以在Python 2.6中编写并且未来很容易迁移到Python 3.0中。

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的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，不过它是以键来存储元素而不是位置，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。

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，这种方法比较费工夫。
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)`。在Python 3.0中，还可以使用一个字典解析：`{k:0 for k in 'ab'}`。
3. `append`和`extend`方法可在原处增长列表，`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。

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在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。

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，元组最好看做是对象引用的数组。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：常见元组常量和运算

运算	解释
()	空元组
T= (0,)	单个元素的元组（非表达式）
T = (0, 'Ni', 1.2, 3)	四个元素的元组
T = 0, 'Ni', 1.2, 3	另一个四元素的元组（与前列相同）
T = ('abc', ('def', 'ghi'))	嵌套元组
T = tuple('spam')	一个可迭代对象的项的元组
T[i]	索引、索引的索引、分片、长度
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	合并、重复
T * 3	
for x in T: print(x)	迭代、成员关系
'spam' in T	
[x ** 2 for x in T]	

表9-1：常见元组常量和运算（续）

运算	解释
T.index('Ni')	Python 2.6和Python 3.0中的方法：搜索、计数
T.count('Ni')	

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。

```
>>> (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，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9-1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）。仅当元组作为常量传给函数调用（圆括号很重要）以及当元组在Python 2.X的print语句中列出（逗号很重要）的特殊情况下，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、方法以及不可变性

除了常量语法不同以外，元组的操作（表9-1的中间行）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并使其成为一个可变对象，才能获得使用排序方法调用的权限，或者使用新的sorted内置方法，它接受任何序列对象（以及更多）：

```
>>> 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')

>>> sorted(T)                   # Or use the sorted built-in
['aa', 'bb', 'cc', 'dd']
```

这里的列表和元组内置函数用来将对象转换为列表，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。

尽管元组的方法与列表和字符串不同，它们在Python 2.6和Python 3.0中确实有两个自己的方法——index和count就像对列表一样工作，但是，它们也针对元组对象定义了：

```
>>> T = (1, 2, 3, 2, 4, 2)      # Tuple methods in 2.6 and 3.0
>>> T.index(2)                 # Offset of first appearance of 2
1
```

```

>>> T.index(2, 2)                # Offset of appearance after offset 2
3
>>> T.count(2)                   # How many 2s are there?
3

```

在Python 2.6和Python 3.0之前，元组根本没有方法——对于不可变对象，这是一种旧的Python惯例，多年前在实际使用字符串的时候打破了这一惯例，更近一点，对于数字和元组都打破了这一惯例。

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，并提到过把元组看做是简单的对象组合，把列表看成是随时间改变的数据结构。实际上，单词“元组”就借用自数学领域，它通常用来指关系数据库表的一行。

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，尽管近年来这样的操作往往已经通用化了。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。

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、序列也不是对应。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，但其他文件方法可查找文件中的新位置、刷新输出缓存等。表9-2总结了常见的文件操作。

表9-2：常见文件运算

操作	解释
<code>output = open(r'C:\spam', 'w')</code>	创建输出文件（'w'是指写入）
<code>input = open('data', 'r')</code>	创建输入文件（'r'是指读写）
<code>input = open('data')</code>	与上一行相同（'r'是默认值）
<code>aString = input.read()</code>	把整个文件读进单一字符串
<code>aString = input.read(N)</code>	读取之后的N个字节（一或多个）到一个字符串
<code>aString = input.readline()</code>	读取下一行（包括行末标识符）到一个字符串
<code>aList = input.readlines()</code>	读取整个文件到字符串列表
<code>output.write(aString)</code>	写入字节字符串到文件
<code>output.writelines(aList)</code>	把列表内所有字符串写入文件
<code>output.close()</code>	手动关闭（当文件收集完成时会替你关闭文件）
<code>output.flush()</code>	把输出缓冲区刷到硬盘中，但不关闭文件
<code>anyFile.seek(N)</code>	修改文件位置到偏移量N处以便进行下一个操作
<code>for line in open('data'): use line</code>	文件迭代器一行一行地读取
<code>open('f.txt', encoding='latin-1')</code>	Python 3.0 Unicode文本文件（str字符串）
<code>open('f.bin', 'rb')</code>	Python 3.0二进制byte文件（bytes字符串）

打开文件

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。处理模式参数也可以指定为其他选项：

- 在模式字符串尾部加上b可以进行二进制数据处理（行末转换和Python 3.0 Unicode 编码被关闭了）。

- 加上“+”意味着同时为输入和输出打开文件（也就是说，我们可以对相同文件对象进行读写，往往与对文件中的修改的查找操作配合使用）。

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）。这里我们将介绍文件的基础知识并探讨一些基础的示例，但是，我们不会介绍所有的文件处理模式选项；与往常一样，请查看Python库手册以了解其他详细信息。

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程序中的文本文件都采用字符串的形式。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。如下是一些基础用法的提示：

文件迭代器是最好的读取行工具

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14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。

内容是字符串，不是对象

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两者进行介绍。

close是通常选项

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（要了解确保自动文件关闭的一种替代方法，请参阅本部分随后对文件对象的上下文管理器的讨论，它在Python 2.6和Python 3.0中与新的`with/as`语句一起使用）。

文件是缓冲的并且是可查找的

前面一段关于关闭文件的提示很重要，因为关闭既释放了操作系统资源也清空了缓冲区。默认情况下，输出文件总是缓冲的，这意味着写入的文本可能不会立即自动从内存转换到硬盘——关闭一个文件，或者运行其`flush`方法，迫使缓存的数据进入硬盘。可以用额外的`open`参数来避免缓存，但是，这可能会影响到性能。Python文件也是在字节偏移的基础上随机访问的，它们的`seek`方法允许脚本跳转到指定的位置读取或写入。

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.txt', 'w')           # Open for text output: create/empty
>>> myfile.write('hello text file\n')          # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                             # Flush output buffers to disk

>>> myfile = open('myfile.txt')                 # Open for text input: 'r' is default
>>> myfile.readline()                           # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                           # Empty string: end of file
''
```

注意，文件`write`调用返回了在Python 3.0中写入的字符数；在Python 2.6中，它们不会，因此，你不会看到交互地响应这些数字。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）。

如果想要显示带有末行字符解释的文件内容，用文件对象的`read`方法把整个文件读入到一个字符串中，并打印它：

```
>>> open('myfile.txt').read()           # Read all at once into string
'hello text file\ngoodbye text file\n'

>>> print(open('myfile.txt').read())     # User-friendly display
hello text file
goodbye text file
```

如果想要一行一行地扫描一个文本文件，文件迭代器往往是最佳选择：

```
>>> for line in open('myfile'):          # Use file iterators, not reads
...     print(line, end='')
...
hello text file
goodbye text file
```

以这种方式编码的时候，`open`临时创建的文件对象将自动在每次循环迭代的时候读入并返回一行。这种形式通常很容易编写，对于内存使用很好，并且比其他选项更快（当然，根据有多少变量）。由于我们还没有介绍语句或迭代器，你将必须等到第14章才能完全理解这段代码。

Python 3.0中的文本和二进制文件

严格地讲，前面小节中的示例使用了文本文件。在Python 3.0和Python 2.6中，文件类型都由`open`的第二个参数决定，模式字符串包含一个“b”表示二进制。Python总是支持文本和二进制文件，但是在Python 3.0中，二者之间有明显的区别：

- 文本文件把内容表示为常规的`str`字符串，自动执行Unicode编码和解码，并且默认执行末行转换。
- 二进制文件把内容表示为一个特殊的`bytes`字符串类型，并且允许程序不修改地访问文件内容。

相反，Python 2.6文本文件处理8位文本和二进制数据，并且有一种特殊的字符串类型和文件接口（`unicode`字符串和`odecs.open`）来处理Unicode文本。Python 3.0中的区别源自于简单文本和Unicode文本都合并为一种常规字符串类型这一事实——这是有意义的，因为所有的文本都是Unicode，包括ASCII和其他的8位编码。

由于大多数程序员只是处理ASCII文本，他们可以用前面的示例中所使用的基本文本文件接口和常规的字符串来做到。在Python 3.0中，所有的字符串从技术上讲都是Unicode，但是ASCII用户通常不会留意。实际上，如果你的脚本仅限于处理如此简单形式的文本的话，文件和字符串在Python 3.0和Python 2.6中同样地工作。

如果需要处理国际化应用程序或者面向字节的数据，Python 3.0中的区别会影响到代码（通常是更好的影响）。通常，你必须使用`bytes`字符串处理二进制文件，并且用常规

的`str`字符串处理文本文件。此外，由于文本文件实现了Unicode编码，不能以文本模式打开一个二进制数据文件——将其内容解码为Unicode文本可能会失败。

让我们来看一个示例。当你读取一个二进制数据文件的时候，得到了一个`bytes`对象——表示绝对字节值的较小整数的一个序列（可能会也可能不会对应为字符），其外观几乎与常规的字符串完全相同：

```
>>> data = open('data.bin', 'rb').read()      # Open binary file: rb=read binary
>>> data                                         # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]                                   # Act like strings
b'spam'
>>> data[0]                                     # But really are small 8-bit integers
115
>>> bin(data[0])                               # Python 3.0 bin() function
'0b1110011'
```

此外，二进制文件不会对数据执行任何换行转换；在根据转化写入和读取并实现Unicode编码的时候，文本文件默认地把所有形式和`\n`之间映射起来。由于Unicode和二进制数据对很多Python程序员来说并不直接相关，我们推迟到第36章再完整地介绍。现在，让我们继续看一些更为实际的文件示例。

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（对于空格，在这里我将从写方法中忽视字节计数返回值）。

```
>>> X, Y, Z = 43, 44, 45                        # Native Python objects
>>> S = 'Spam'                                   # Must be strings to store in file
>>> 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：

```
>>> chars = open('datafile.txt').read()          # Raw string display
>>> chars
```

```
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)                                # User-friendly display
Spam
43,44,45
[1, 2, 3]${'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：

```
>>> 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。现在我们可以脚本内应用列表和字典操作了。

用pickle存储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。

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                # Pickle any object to file
>>> F.close()
```

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.pkl', 'rb')
>>> 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.pkl', 'rb').read()          # Format is prone to change!
b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'
```

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。与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（尽管你可以在本书第27章看到应用shelve的一个示例，并且在第30章和第36章看到其他的pickle示例）。

注意：我们以二进制模式打开用来存储pickle化的对象的文件，二进制模式总是Python 3.0中必需的，因为pickle程序创建和使用一个bytes字符串对象，并且这些对象意味着二进制模式文件（文本模式文件意味着Python 3.0中的str字符串）。在早期的Python中，对于协议0使用文本模式文件是没有问题的（默认情况下，它创建ASCII文本），只要一致地使用文本模式；较高的协议要求二进制模式文件。Python 3.0的默认协议是3（二进制），但是，它即便对于协议0也创建bytes。参见第36章Python的库手册或其他参考书来了解关于这一点的更多细节。

Python 2.6也有一个cPickle模块，它是pickle的一个优化版本，可以直接导入以提高速度。Python 3.0把这模块改名为_pickle，并且在pickle中自动使用它——脚本直接导入pickle并且允许Python优化它。

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
>>> data = struct.pack('>i4sh', 7, 'spam', 8)    # Make packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                                # 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
b'\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它的内容的。

文件上下文管理器

我们还要看看第33章对于文件的上下文管理器支持的讨论，在Python 3.0和Python 2.6中有所更新。因而比文件自身多了一个异常处理功能，它允许我们把文件处理代码包装到一个逻辑层中，以确保在退出后可以自动关闭文件，而不是依赖于垃圾收集上的自动关闭：

```
with open(r'C:\misc\data.txt') as myfile:          # See Chapter 33 for details
    for line in myfile:
        ...use line here...
```

我们在第33章看到过的try/finally语句可以提供类似的功能，但是，需要一些额外代码的成本——更准确地说，是3个额外的代码行（尽管我们常常可以避免两个选项，并且使用Python为我们自动关闭文件）：

```
myfile = open(r'C:\misc\data.txt')
try:
    for line in myfile:
        ...use line here...
finally:
    myfile.close()
```

由于这两个选项都需要比我们现在已经知道的更多的信息，我们将推迟到本书后面详细介绍。

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`能够强制性地将在缓存输出写入磁盘（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`，可以给它们传入到一个打开文件对象（在Python 2.6中，可以传入名称`file`，但在Python 3.0中不能这么做）。更多有关文件处理的例子请参考第13章中“为什么要在意文件扫描”一节。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，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，还有其他可用的，例如：

标准流

在`sys`模块中预先打开的文件对象，例如`sys.stdout`（参见本书第11章的“打印操作”小节）

`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。

`sockets`、`pipes`和FIFO文件

文件类对象，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（第27章中用到）。

Shell命令流

像`os.popen`和`subprocess.Popen`这样的工具，支持产生shell命令，并读取和写入到标准流。

第三方开源领域提供了甚至更多的文件类工具，包括`PySerial`扩展中支持与窗口交流，以及`pexpect`系统中的交互程序。请参见更多高级Python资料以及Web，以了解关于文件类工具的更多信息。

注意：版本差异提示：在Python 2.5及更早的版本中，内置的名称`open`基本上是名称`file`的同义词，并且文件可以通过调用`open`或`file`来打开（通常更倾向于使用`open`打开）。在Python 3.0中，名称`file`不再可用，因为它和`open`冗余。

Python 2.6用户可能也使用名称`file`作为一种文件对象类型，以便使用面向对象编程来定制文件（本书后面介绍）。在Python 3.0中，文件已经彻底变化了。用来实现文件对象的类位于标准库模块`io`中。可以参见这个模块的文档，或者编写可以使用它来定制类，并且在打开的文件`F`上运行一条`type(F)`调用，以得到相关提示。

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。

下面是需要记住的一些要点：

- 对象根据分类来共享操作；例如，字符串、列表和元组都共享诸如合并、长度和索引等序列操作。
- 只有可变对象（列表、字典和集合）可以原处修改；我们不能原处修改数字、字符串或元组。
- 文件导出唯一的方法，因此可变性并不真的适用于它们——当处理文件的时候，它们的状态可能会修改，但是，这与Python的核心类型可变性限制不完全相同。
- 表9-3中的“数字”包含了所有数字类型：整数（与Python 2.6的整数有区别）、浮点数、复数、小数和分数。
- 表9-3中的字符串包括str，以及Python 3.0中的bytes和Python 2.6中的unicode；Python 3.0中的bytearray字符串类型是可变的。
- 集合类似于一个无值的字典的键，但是，它们不能映射为值，并且没有顺序；因此，集合不是一个映射类型或者一个序列类型，frozenset是集合的一个不可变的版本。
- 除了类型分类操作，表9-3中的Python 2.6和Python 3.0的所有类型都有可调用的方法，这些方法通常特定于它们的类型。

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
Sets	集合	是
frozenset	集合	否
bytearray (3.0)	序列	是

为什么要在意操作符重载

本书后面的第五部分中，我们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。

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。
- 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)]
```

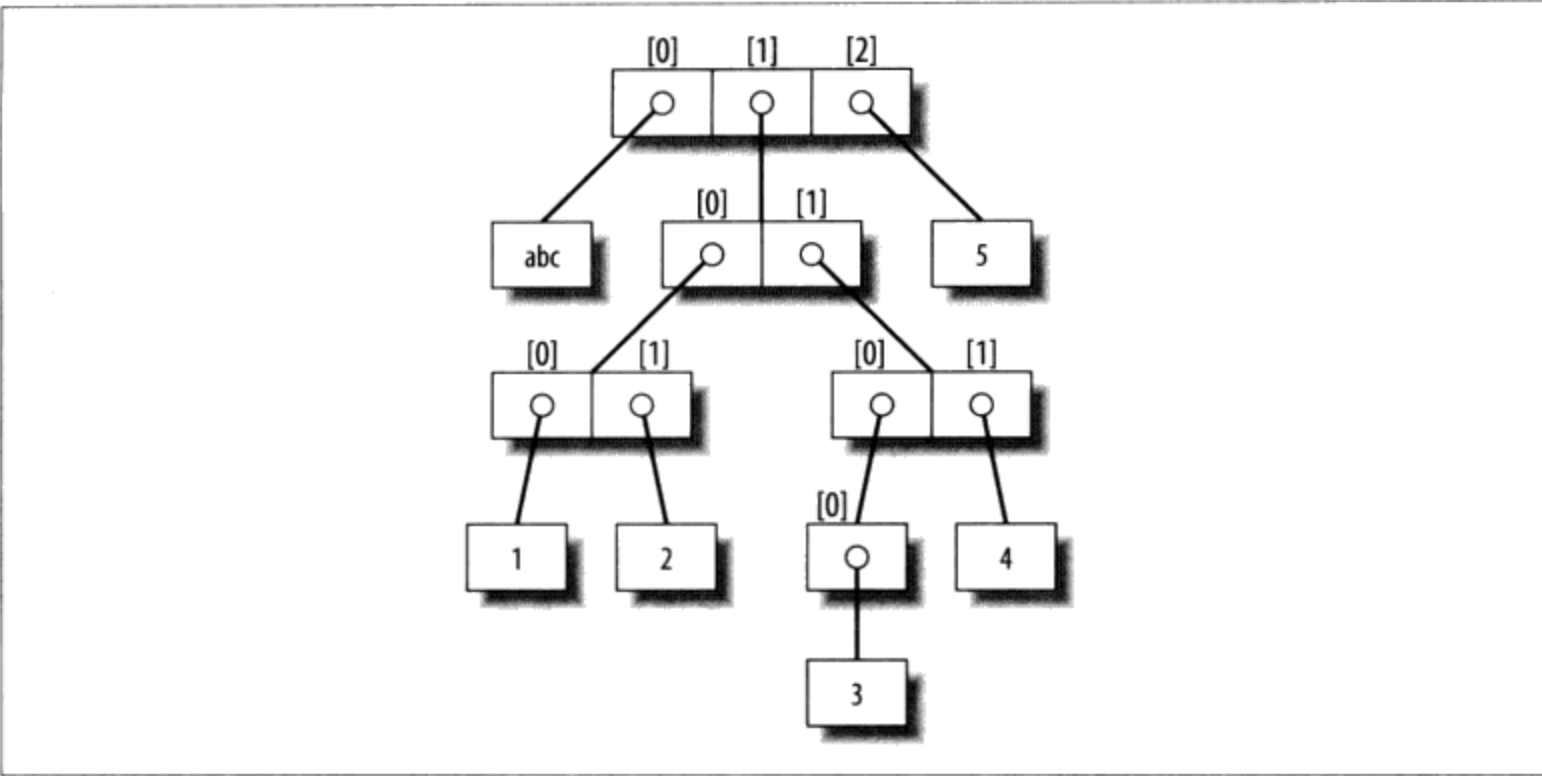


图9-1：一个由运行常量表达式['abc', [(1, 2), ([3], 4)], 5]生成的元素偏移的嵌套对象树。从语法上来说，嵌套对象在内部被表示为对不同内存区域的引用（也就是指针）

```
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

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']
>>> D = {'x':X, 'y':2}
# Embed references to X's object
```

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。

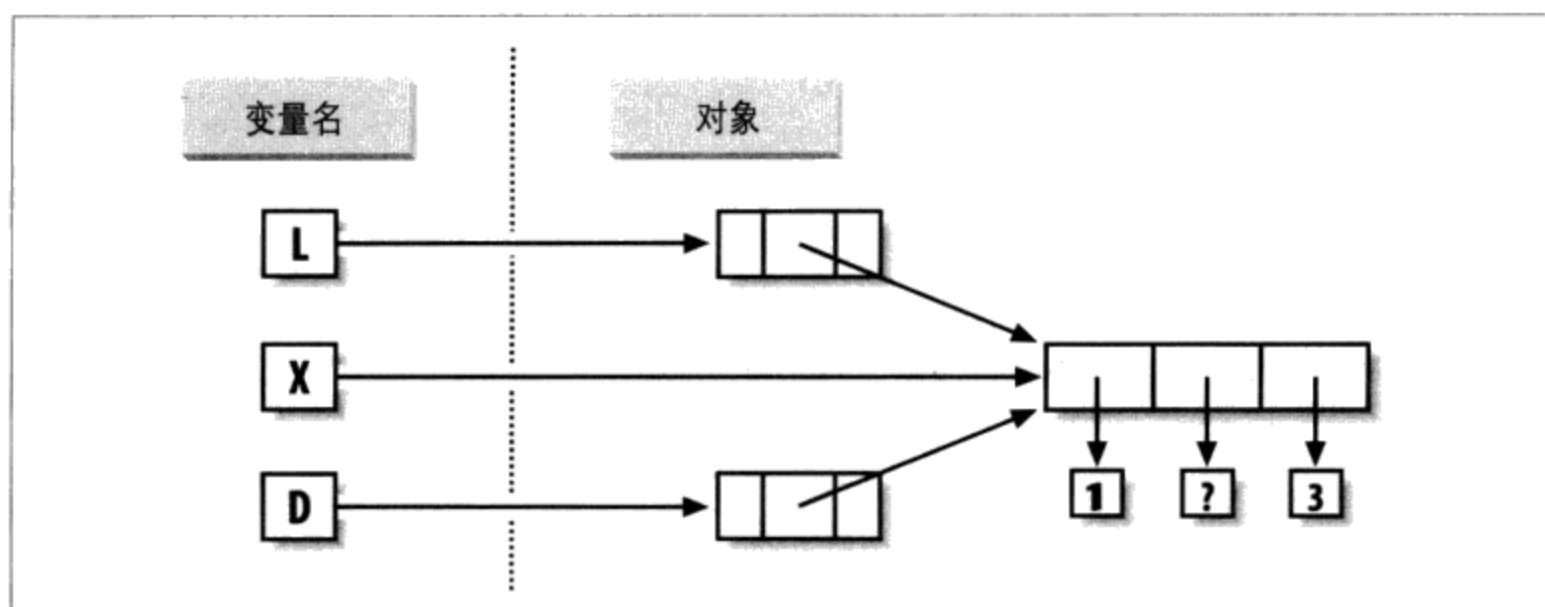


图9-2：共享对象引用：因为变量X引用的列表也在被L和D引用的对象内引用，修改X的共享列表与L和D的看起来也有所不同

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（`X.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 (ditto for sets)
```

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。

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'
>>> 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。字典的相对大小比较在Python 3.0中不支持，但是，它们在Python 2.6及更早的版本中有效，就像是比较排序的（键、值）列表一样。
- 数字混合类型比较（例如，`1 < 'spam'`）在Python 3.0中是错误的。Python 2.6中允许这样的比较，但是使用一种固定但任意的排序规则。通过代理，这也适用于排序，它在内部使用比较：非数字的混合类型集合不能在Python 3.0中排序。

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 3.0的字典比较

再次看看前面小节介绍的优点中的最后一点。在Python 2.6和更早的版本中，字典支持大小比较，就好像要比较排序的键/值列表：

```
C:\misc> c:\python26\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
True
```

在Python 3.0中，字典的大小比较删除了，因为当期望相等的时候，它们导致太多的负担（在Python 3.0中，相等性使用一种优化的方案，并且不会直接比较排序的键/值列表）。Python 3.0中的替代方法，要么编写循环来根据键比较值，要么手动比较排序的键/值列表——items字典方法的和内置的sorted足够了：

```
C:\misc> c:\python30\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()

>>> list(D1.items())
[('a', 1), ('b', 2)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]

>>> sorted(D1.items()) < sorted(D2.items())
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

实际上，大多数程序需要这种行为，以开发出比这里的解决方案或Python 2.6中的最初行为更高效的方式来比较字典中的数据。

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测试结果代表着真和假。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，对于这些名称的真实含义需要一点更多的形式。

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。

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

作为一个应用，判断对象自身是真或假，通常会看到Python程序员像if X:这样编写测试，其中，假设X是一个字符串，那么，等同于if X != '':。换句话说，可以测试对象自身，而不是将它们和一个空的对象比较（第三部分更详细地介绍if语句）。

None对象

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, ... ]
```

这不会限制列表的大小（它随后仍然可以增长或缩短），但是直接预先设置一个初始大小，会考虑到将来的索引赋值。当然，可以以同样的方式用0来初始化一个列表，但最佳的实践是如果还不知道列表的内容的话使用None。

记住，None不是意味着“未定义”。也就是说，None是某些内容，而不是没有内容（尽管起名字是没有内容）——它是一个真正的对象，并且有一块内存，由Python给定一个内置的名称。在本书随后查看这一特殊对象的其他用法，正如我们将在本书第四部分看到的，它还是函数的默认返回值。

bool类型

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1和0，但它们使得程序的意图更明确。
- 交互模式下运行的布尔测试的结果打印成True和False的字样，而不是1和0，以使得程序的结果更明确。

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。Python还提供了一个内置函数bool，它可以用来测试一个对象的布尔值（例如，它是否为True，也就是说，非零或非空）：

```
>>> bool(1)
True
>>> bool('spam')
True
>>> bool({})
False
```

实际上，我们很少真正注意到逻辑测试所产生的布尔类型，因为if语句和其他的选择工具自动地使用布尔结果。

我们将在本书第12章学习逻辑语句的时候进一步介绍布尔类型。

Python的类型层次

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）。

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。

Type对象

事实上，即使是类型本身在Python中也是对象类型（快速念三遍）。

严格地说，对内置函数`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，因为它限制了代码的灵活性。

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`、`float`、`complex`、`byte`、`type`、`set`和`file`（在Python 2.6中`file`也是类型名称，是`open`的同义词，但在Python 3.0中并非如此）。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。

此外，Python 3.0中的类型标准库模块同样提供其他不能作为内置类型使用类型的名称（例如，一个函数的类型；在Python 2.6中，这个模块也包含了内置类型名称的同义词，但在Python 3.0中并非如此）而且用`isinstance`函数进行类型测试也是有可能的。例如，下列所有类型测试都为真：

```
type([1]) == type([])           # Type of another list
type([1]) == list               # List type name
isinstance([1], list)          # List or customization thereof

import types                   # types has names for other types
def f(): pass
type(f) == types.FunctionType
```

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31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。

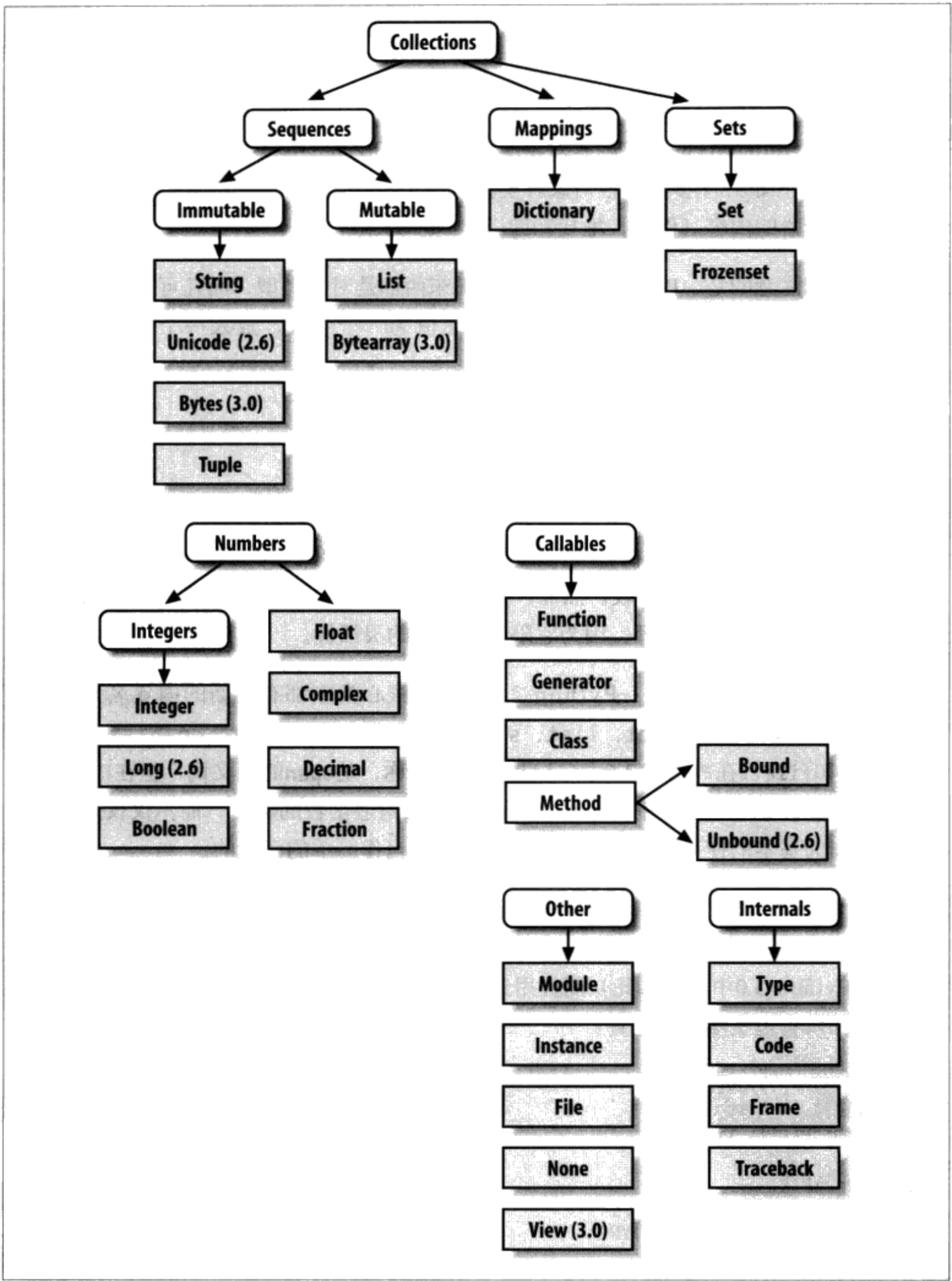


图9-3：按类别组织的Python的主要内置对象类型。Python中所有一切都是某种类型的对象，即便是某个对象的类型！任何对象的类型都是类型为“type”的对象

在第31章中，我们介绍了`type(X)`类型测试如何广泛地应用于用户定义的类。简而言之，在Python 3.0中，一个类实例的类型就是该实例产生自哪个类。对于Python 2.6及其以前的版本中的传统类，所有的类实例都是“实例”类型，并且我们要有意义地比较实例的类型，必须比较实例的`__class__`属性。由于我们还没有准备好类的知识，我们将推迟到第31章详细介绍其他内容。

Python中的其他类型

除了本书这一部分中我们所学的核心对象之外，以及我们在后面将要遇到的函数、模块和类，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]]
```

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

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。你可以参考附录B中的“第一部分练习题”得到有关这一问题的更多信息，也可以在第24章的reloadall.py程序部分找到相关的解决方法。

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，否则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。

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个本章的习题，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？

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。通常，内置函数和表达式可以跨越多种对象类型；方法特定于一种单个的对象类型，尽管通过某些方式可以在多种类型上使用（例如，索引，在列表和元组上都有效）。
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，可以从其组成成分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。

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。注意，在这些情况中，分号用作语句分隔符，以把多条语句压缩成单独的一行：例如，X=1;X赋值并打印出一个变量（本书下一部分将更多地介绍语句语法）。还要记住表达式之间的逗号通常用来构建元组，即便没有包围的圆括号：X,Y,Z是一个三项元组，Python打印它的时候会带上圆括号。

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
'green {0} and {1}'.format('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
```

```
list(D.keys()), list(D.values()), (1,2,3) in D
[[[]], ["",[],(),{}],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；你可能不会在真实的程序中看到这些例子，但是，它们用来促使你思考底层的模式，并且其中一些可能在较少的人为编写的形式中 useful：

- 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？

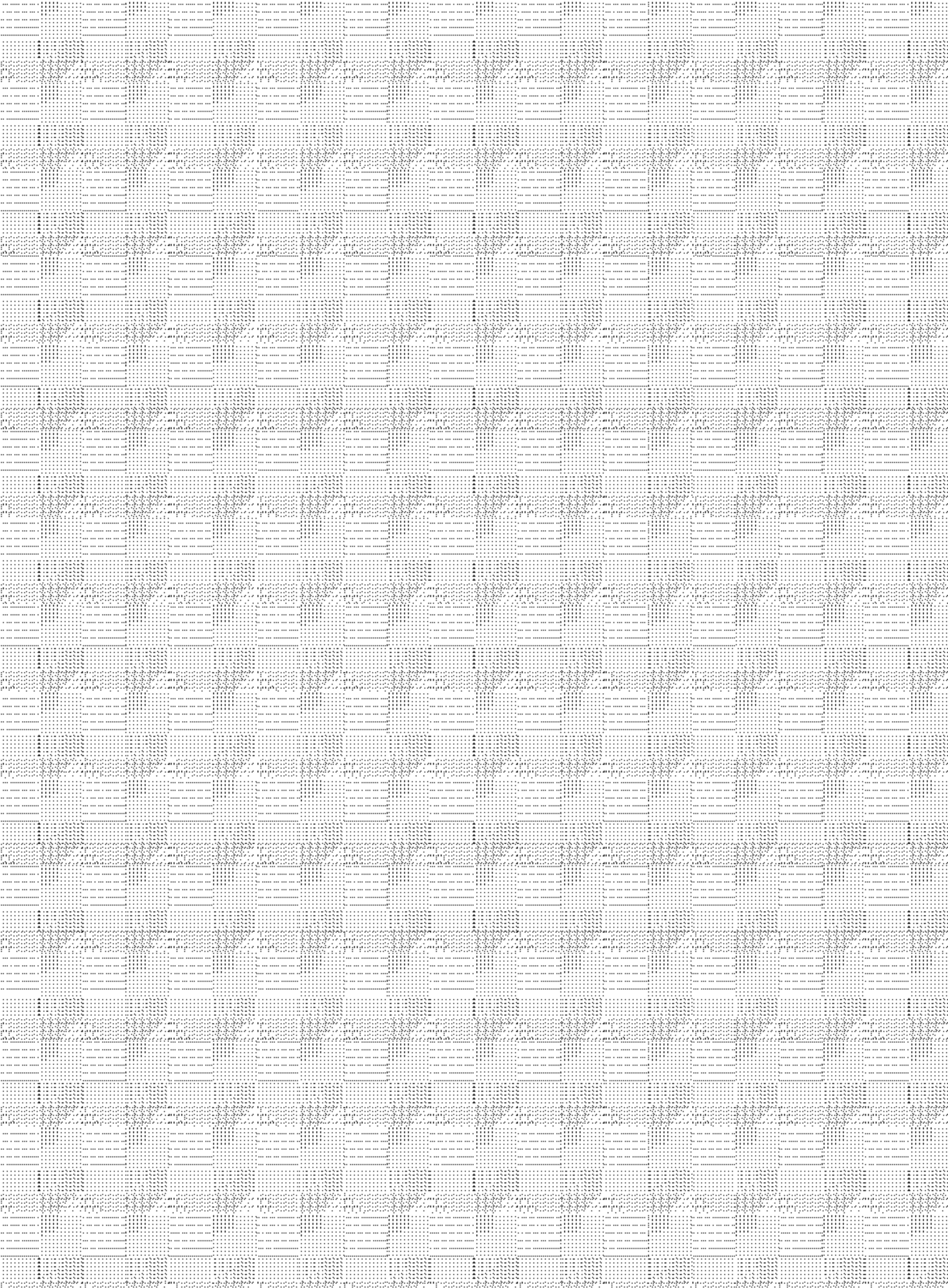
5. 字典键。考虑下列代码片段：

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

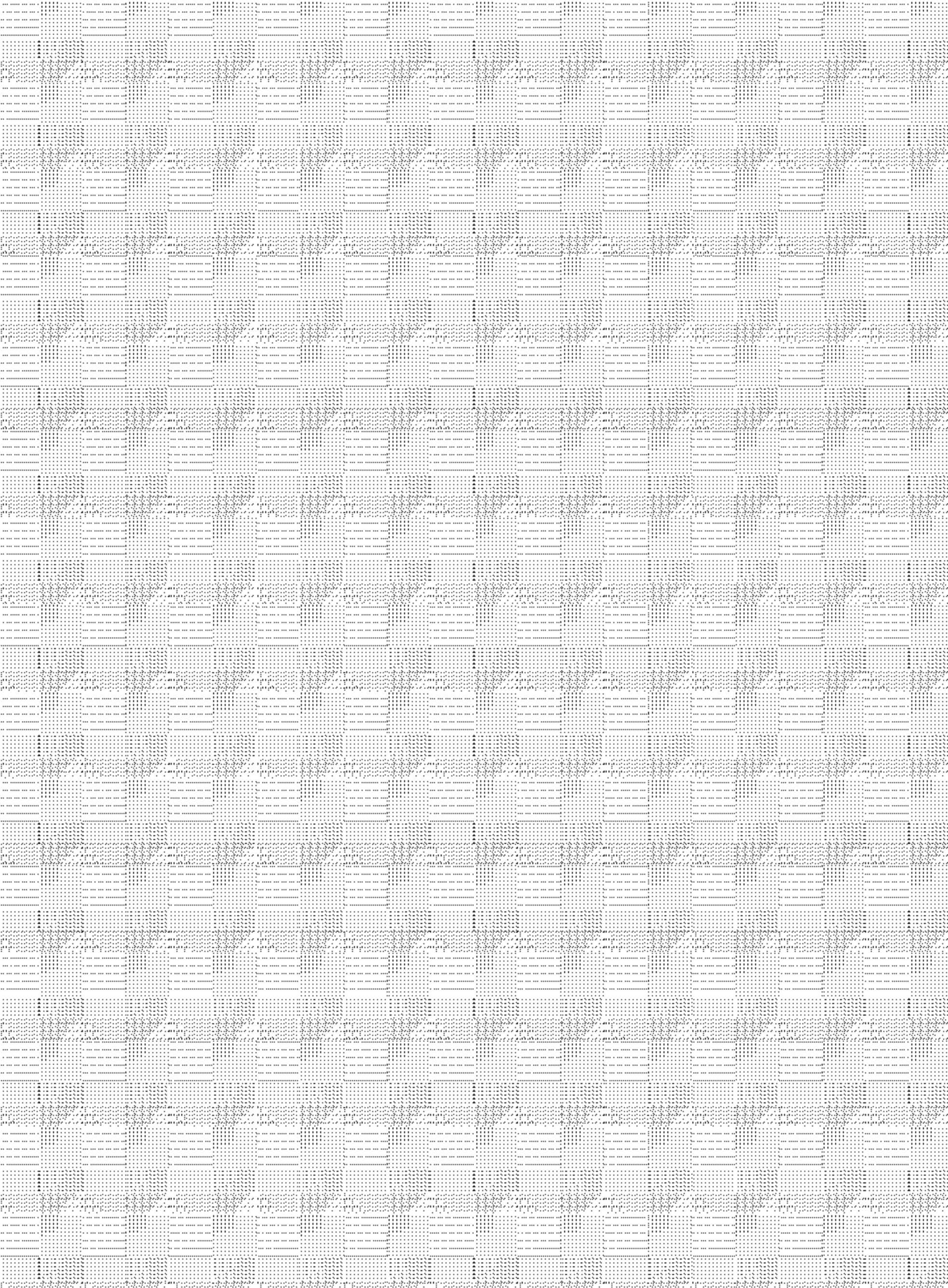
已知道字典不是按偏移值读取的，那么这里是在做什么？下面的例子让你看见了其主题吗？（提示：字符串、整数以及元组共享哪种类型分类？）

```
>>> 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？
 - 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。



语句和语法



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“用一些内容做事情”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。语句编码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。

Python的语句

表10-1总结了Python的语句集。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略过的细节，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。更多相关的语句（例如删除各种成分的del语句）将会在本书其他部分进行讨论，或者可以参考Python标准手册。

表10-1：Python语句

语句	角色	例子
赋值	创建引用值	<code>a, b, c = 'good', 'bad', 'ugly'</code>
调用	执行函数	<code>log.write("spam, ham")</code>
打印调用	打印对象	<code>print ('The Killer', joke)</code>
if/elif/else	选择动作	<code>if "python" in text: print (text)</code>
for/else	序列迭代	<code>for x in mylist: print (x)</code>
while/else	一般循环	<code>while X > Y: print ('hello')</code>
pass	空占位符	<code>while True: pass</code>
break	循环退出	<code>while True: if exittest(): break</code>
continue	循环继续	<code>while True: if skiptest(): continue</code>
def	函数和方法	<code>def f(a, b, c=1, *d): print(a+b+c+d[0])</code>
return	函数结果	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
yield	生成器函数	<code>def gen(n): for i in n: yield i*2</code>

表10-1: Python语句 (续)

语句	角色	例子
global	命名空间	<pre>x = 'old' def function(): global x, y; x = 'new'</pre>
nonlocal	Namespaces(3.0+) 命名空间 (Python 3.0及其以上版本)	<pre>def outer(): x = 'old' def function(): nonlocal x; x = 'new'</pre>
import	模块访问	<pre>import sys</pre>
from	属性访问	<pre>from sys import stdin</pre>
class	创建对象	<pre>class Subclass(Superclass): staticData = [] def method(self): pass</pre>
try/except/ finally	捕捉异常	<pre>try: action() except: print ('action error')</pre>
raise	触发异常	<pre>raise EndSearch(location)</pre>
assert	调试检查	<pre>assert X > Y, 'X too small'</pre>
with/as	环境管理器 (2.6)	<pre>with open('data') as myfile: process(myfile)</pre>
del	删除引用	<pre>del data[k] del data[i:j] del obj.attr del variable</pre>

表10-1中包含了Python 3.0中的语句形式，每部分代码都说明了其具体语法和用途。如下是关于内容的一些说明：

- 赋值语句以不同的语法形式呈现，如本书第11章所介绍的：基本的、序列的、扩展的等等。
- 从技术上讲，print在Python 3.0中不是一个保留字，也不是一条语句，而是一个内置的函数调用；由于它几乎总是作为一条表达式语句运行（即，自己单独一行），通常将其看做是一条语句类型。我们将在第11章学习打印操作。
- yield实际上是一个表达式，而不是一条语句，与Python 2.5中一样；就像print一

样，它通常单独用在一行中，因此，它包含在本表中，但是脚本偶尔会赋值或使用其结果，正如我们将在第20章中见到的。与`print`不同，作为一个表达式，`yield`也是一个保留字。

表中的大多数语句也适用于Python 2.6，除非个别不适用的地方——如果你使用Python 2.6或更早的版本，下面是对于Python的一些注意事项：

- 在Python 2.6中，`nonlocal`不可用；正如我们将在第17章所看到的，有替代方法可以实现这条语句的可写状态保持效果。
- 在Python 2.6中，`print`是一条语句，而不是一个内置函数调用，其具体语法在第11章中介绍。
- 在Python 2.6中，Python 3.0的`exec`代码执行内置函数是一条语句，具有特定的语法；因为它支持带有圆括号的形式，通常在Python 2.6代码中使用其Python 3.0的调用形式。
- 在Python 2.5中，`try/except`和`try/finally`语句合并了：这两条语句曾经是形式不同的语句，但是，现在我们在同一条`try`语句中使用`except`和`finally`都可以。
- 在Python 2.5中，`with/as`是一个可选的扩展，并且通常它是不可用的，除非你通过运行`__future__ import with_statement`语句（参见第33章）来打开它。

两个if的故事

在我们深入介绍表10-1中的任何一条具体语句之前，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发现这一错误，而使得输入冒号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，但是你必须要在类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：

```
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形式。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一名正在学习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，就是你可以省略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，你依然是一名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，也许对刚刚入门的前C程序员的人来讲是最不寻常的一个（直到他们用上10分钟之后才意识到它实际上是Python的一大特色），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：

```
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并不在乎怎么缩进（你可以使用空格或制表符）或者缩进多少（你可以使用任意多个空格或是制表符）。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，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：

```
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，它们将很难解释、修改或者再使用，因为代码不再能形象地反应其逻辑含义。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！

这种事在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`就是其逻辑上的`if`（外层的`if x`）。从某种意义上来说，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其运行的方式。

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，我在一家成功地用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来分析代码中的缩进。如果这个脚本检查到我们没有一致的缩进程序代码，我们将会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。

作为首要的规则，不应该在同一段Python代码中混合使用制表符和空格，除非你一贯这么做；在一段给定的代码中，使用制表符或空格，但不要二者都用（实际上，Python 3.0现在发布了一个关于使用制表符和空格的不一致性的错误，参见本书第12章的介绍）。但是，不应该在任何结构化语言中混合使用制表符和空格来缩进——如果下一位程序员用与你不同的自己的编辑器来显示制表符，这样的代码可能会引发较大的可读性问题。类C的语言可能会使程序员绕过这样的问题，但是，它们不应该这么做：结果会被搞得乱糟糟的。

不管用何种语言编写代码，都应该一致地缩进以保持可读性，这一点无论怎么强调都不过分。实际上，如果你在此前的职业生涯中没有学习如何做到这点，你的老师可能给你留下了伤害。大多数程序员，尤其是那些必须阅读其他人的代码的程序员，认为这是提升自己的Python语法级别的重要资本。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：

```
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基本上是相反的——可以让一个语句的范围横跨多行。为了能够实现这一操作，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。花括号包含的字典（以及Python 3.0中的集合常量、字典解析以及集合解析）也可以用这个方法横跨数行，并且圆括号可以处理元组、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的——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
```

An error-prone alternative

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）。另外

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。

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。

一般来说，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。此外，一些代码探查和覆盖工具能够把压缩到单个一行中的多条语句和一个单行复合语句的头部和主体区分开来。在Python中，保持事情简单对你来说总是一大优点。

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 = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

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看做是永远为真）时不停地迭代的嵌套代码块。
- 我们之前曾在本书中见到过的input内置函数，在这里用于通用控制台输出，它打印可选的参数字符串作为提示，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
```

注意： 版本差异提示：这个例子是针对Python 3.0编写的。如果你使用Python 2.6或之前的版本，这段代码也能工作，但是，你应该使用raw_input而不是input，并且你可以在print语句中省略外围的圆括号。在Python 3.0中，前者是重新命名了，后者是一个内置函数而不再是一条语句（关于print的更多介绍参加下一章）。

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。这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 = 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，不会看做是循环体的一部分，所以只能退出循环之后运行一次。

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

这里注意一点：我假设这段代码存储在一个脚本文件中并通过它运行。如果你在交互模式下遇到这段代码，请确保在最后的print语句之前包含一个空行（例如，按下Enter键两次），以终止循环。在交互模式下，最后的print意义不大（在与循环交互之后，必须编写它）。

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)
```

这样就给了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。

```
while True:
    reply = 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，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。本书的第七部分深入探索这个语句，在这里先简单介绍一下，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 = 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。在这个例子中，try语句从单词try开始，一直到else语句下面缩进的代码结束，因为else和try为相同的缩进层级。这段代码中的if语句是一个单行语句，并且在break之后结束。

我们会在本书后面重新介绍try语句。就目前而言，只需知道，因为try可用于拦截任何错误，于是可以减少你必须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。例如，如果想要支持输入浮点数而不只是整数，使用try将比手动的错误检测要容易得多——我们可能直接运行一个float调用并捕获其异常，而不是试图分析所有可能的浮点语法。

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的再深入一步。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 = 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。本章介绍了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。

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. 只有当你需要把一系列以上的语句挤进一行代码时。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。如果你还没有犯过这样的错误，你可能很快会犯。

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。本书前面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的赋值语句把对象赋给一个名称。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住的，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存储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，而不是数据存储区域。
- **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，就会被其所引用的值取代。
- **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。如果你试图这么做，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。

- **执行隐式赋值的一些操作。**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
<code>spam = 'Spam'</code>	基本形式
<code>spam, ham = 'yum', 'YUM'</code>	元组赋值运算（位置性）
<code>[spam, ham] = ['yum', 'YUM']</code>	列表赋值运算（位置性）
<code>a, b, c, d = 'spam'</code>	序列赋值运算，通用性
<code>a, *b = 'spam'</code>	扩展的序列解包（Python 3.0）
<code>spam = ham = 'lunch'</code>	多目标赋值运算
<code>spams += 42</code>	增强赋值运算（相当于 <code>spams = spams + 42</code> ）

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。实际上，仅仅使用这些基本的形式就可以搞定所有的工作了。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，字符串'yum'赋值给变量名spam，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。

扩展的序列解包

在Python 3.0中，一种新形式的序列赋值允许我们更灵活地选择要赋值的一个序列

的部分。例如，表11-1中的第五行，用右边的字符串的第一个字母来匹配a，用剩下的部分来匹配b：a赋值为's'，b赋值为'pam'。这为手动分片操作的结果的赋值提供了一种简单的替代方法。

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两者都赋值成对相同的字符串对象'lunch'的引用。效果就好像我们写成ham = '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）。

增强赋值语句

表11-1的最后一行是增强赋值语句的例子——一种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。此外，如果操作主体是可变的并且支持这一操作，增强赋值通过选择原处更新操作而不是对象副本，从而可以更快地运行。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部分练习题的解答中介绍过。因为语句执行时，Python会建立临时的元组，来存储右侧变量原始的值，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
(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14章和第20章介绍。

高级序列赋值语句模式

注意：虽然可以在“=”符号两侧混合相匹配的序列类型，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。Python 3.0允许我们使用更为通用的扩展解包语法，这在前面小节中介绍过。但是，通常，并且在Python 2.X中总是如此：赋值目标中的项数和主体的数目必须一致：

```
>>> 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')

```

如最后的例子所示，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 by shape and position
>>> a, b, c
('S', 'P', 'AM')

```

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（'A M'）一次赋值给变量c。在这个过程中，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算是比较高级，也很少见到的，但是利用已知形状取出数据结构的组成成分，也是很方便的。

例如，我们将会在第13章中看到，这一技术在for循环中也有效，因为循环项赋值给了在循环头部给定的目标：

```

for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...      # Simple tuple assignment
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...  # Nested tuple assignment

```

在第18章中的一个提示中，我们还将看到这种嵌套元组（实际上是序列）解包赋值形式在Python 2.6（尽管不是在Python 3.0中）中用于函数参数列表，因为函数参数也是通过赋值来传递的：

```

def f(((a, b), c)):                             # For arguments too in Python 2.6, but not 3.0
    f(((1, 2), 3))

```

序列解包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）。为了使其变得有意义，你需要知道range内置函数会产生连续整数列表：

```

>>> range(3)                                     # Use 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:]           # See next section for 3.0 alternative
...     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，但这是在原处进行的修改。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。

Python 3.0中的扩展序列解包

前面小节展示了如何使用手动分片以使得序列赋值更为通用。在Python 3.0中（但在Python 2.6中），序列赋值变得更为通用，从而使这一过程变得容易。简而言之，一个带有单个星号的名称，可以在赋值目标中使用，以指定对于序列的一个更为通用的匹配——一个列表赋给了带星号的名称，该列表收集了序列中没有赋值给其他名称的所有项。对于前面小节的最后示例中那种把一个序列划分为其“前面”和“剩余”部分的常用编码模式，这种方法特别方便。

扩展的解包的实际应用

让我们来看一个示例。正如我们所看到的，序列赋值通常要求左边的目标名称数目与右边的主体中的项数完全一致。如果长度不同的话，将会得到一个错误（除非像前面小节所介绍的那样手动地在右边分片）：

```
C:\misc> c:\python30\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
>>> a, b = seq
```

ValueError: too many values to unpack

然而，在Python 3.0中，我们可以在目标中使用带单个星号的名称来更通用地匹配。在如下的后续交互会话中，a匹配序列中的第一项，b匹配剩下的内容：

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

当使用一个带星号的名称的时候，左边的目标中的项数不需要与主体序列的长度匹配。实际上，带星号的名称可以出现在目标中的任何地方。例如，在下面的交互中，b匹配序列中的最后一项，a匹配最后一项之前的所有内容：

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

当带星号的名称出现在中间，它收集其他列出的名称之间的所有内容。因此，在下面的交互中，第一项和最后一项分别赋给了a和c，而b获取了二者之间的所有内容：

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

更一般的，不管带星号的名称出现在哪里，包含该位置的每个未赋值名称的一个列表都将赋给它：

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

和常规的序列赋值一样，扩展的序列解包语法对于任何序列类型都有效，而不只是对列表有效。下面，它解包一个字符串中的字符：

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])
```

```
>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')
```

这和分片内在的相似，但是不完全相同——一个序列解包赋值总是返回多个匹配项的一个列表，而分片把相同类型的一个序列作为分片的对象返回：

```
>>> S = 'spam'

>>> S[0], S[1:]           # Slices are type-specific, * assignment always returns a list
('s', 'pam')

>>> S[0], S[1:3], S[3]
('s', 'pa', 'm')
```

Python 3.0中有了这一扩展，我们处理前面小节的最后一个例子的列表变得容易很多了，因为我们不必手动地分片来获取第一项和剩下的项：

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L           # Get first, rest without slicing
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

边界情况

尽管扩展的序列解包很灵活，一些边界情况还是值得注意。首先，带星号的名称可能只匹配单个的项，但是，总是会向其赋值一个列表：

```
>>> seq
[1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

其次，如果没有剩下的内容可以匹配带星号的名称，它会赋值一个空的列表，不管该名称出现在哪里。如下所示，a、b、c和d已经匹配了列表中的每一项，但Python会给e赋值一个空的列表，而不是将其作为错误情况对待：

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

最后，如果有多个带星号的名称，或者如果值少了而没有带星号的名称（像前面一样），以及如果带星号的名称自身没有编写到一个列表中，都将会引发错误：

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

一个有用的便利形式

记住，扩展的序列解包赋值只是一种便利形式。我们通常可以用显式的索引和分片实现同样的效果（并且实际上必须在Python 2.X中使用），但是，扩展的解包更容易编写。例如，常用的“第一个，其余的”分片编码模式可以用任何一种方式来编写，但是，分片还包括其他的工作：

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq                                # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:]                      # First, rest: traditional
>>> a, b
(1, [2, 3, 4])
```

常见的“剩下的，最后一项”分隔模式类似地也可以用任何一种方式来编写，但是，新的扩展解包语法显然要减少很多录入：

```
>>> *a, b = seq                                # Rest, last
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1]                  # Rest, last: traditional
>>> a, b
([1, 2, 3], 4)
```

由于扩展的序列解包语法不仅更简单，而且更自然，这种语法可能会随着时间在Python代码中变得更广泛。

应用于for循环

由于for循环语句中的循环变量可能是任何赋值目标，扩展的序列赋值在这里也有效。

我们在本书第二部分简单介绍了for循环迭代工具，并且将在第13章中正式地学习它。在Python 3.0中，扩展赋值可能出现在单词for之后，而更常见的用法是一个简单的变量名称：

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    ...
```

当在这种环境中使用的时候，在每次迭代中，Python直接把下一个值的元组分配给名称的元组。例如，在第一次循环中，就好像我们运行如下的赋值语句：

```
a, *b, c = (1, 2, 3, 4)                # b gets [2, 3]
```

名称a、b和c可以在循环的代码中用来引用提取的部分。实际上，这真的不是特殊情况，只是通用的赋值用法的一种情况。正如我们在本章前面所见到的，在Python 2.X和Python 3.X中，我们都可以使用简单的元组赋值做同样的事情：

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:    # a, b, c = (1, 2, 3), ...
```

我们总是可以在Python 2.6中使用手动分片来模拟Python 3.0的扩展赋值行为：

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    a, b, c = all[0], all[1:3], all[3]
```

在第13章中，我们已经学习了足够多的知识以了解有关for循环语法的更多细节，那时候我们还将返回来讨论这一话题。

多目标赋值语句

多目标赋值语句就是直接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，所以你在增加值之前，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在原处附加值上去，而我们通过a也会看见所有的效果。这其实是我们第6章所见的共享引用值现象的另一个例子而已。为避免这个问题，要在单独的语句中初始化可变对象，以便分别执行独立的常量表达式来创建独立的空对象：

```
>>> a = []
>>> b = []
>>> 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致相等：

```
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。

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]
```

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没有像对数字那样对不可变对象进行在原处的修改的概念。

```
>>> 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^{注2}。

```
>>> 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]
```

在两种情况下，合并对共享对象引用产生的副作用可能会更小，但是，通常会比对等的原处形式运行得更慢。合并操作必须创建一个新的对象，把左侧的复制到列表中，然后再把右侧的复制到列表中。相比而言，原处方法调用直接在一个内存块末尾添加项。

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。

```
>>> L += [9, 10]               # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

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

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。

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，为程序中的事物选择变量名时，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 3.0中的保留字

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

表11-3针对Python 3.0。在Python 2.6中，保留字的范围略有不同：

- `print`是一个保留字，因为打印是一条语句，而不是一个内置函数（本章稍后更详细地介绍）。
- `exec`是一个保留字，因为它是一条语句，而不是一个内置函数。
- `nonlocal`不是一个保留字，因为这条语句不可用。

在更早的Python中，情况或多或少也有些不同，有以下一些变化：

- `with`和`as`在Python 2.6之前都不是保留字，只有在上下文管理器正式可用之后，它们才是。
- `yield`在Python 2.3之前都不是保留字，只有在生成器函数可用后它才是。
- 在Python 2.5中，`yield`从语句变为表达式，但它仍然是一个保留字，而不是一个内置函数。

你将会看到，大多数Python保留字都是小写的。而且它们确实是保留字，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}。

除了大小写是混合的，表11-3中的前三项`True`、`False`和`None`的含义多少有些特殊——它们也出现在第17章所介绍的Python的内置函数作用域中，并且它们也是可以分配给对象的技术名称。在所有其他的含义下，它们确实都是保留字，并且，在脚本中，除了它们所表示的对象之外，不可用于任何其他的用途。所有其他的保留字在Python的语法中都是固定的，只能在其本意的特定环境中使用。

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，因为其变量名没有“.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。第五部分我们会再次介绍这个概念。

Python的废弃协议

注意保留字在一种语言的各个阶段是如何变化的，这是一件有趣的事情。当一种新的功能可能会影响到已有的代码的时候，Python通常会使其成为可选的，并且将其发布为“废弃的”，以便在正式使用该功能之前的一个或多个发布中提出警告。这种思路使你有足够的时间注意到警告，并且在迁移到新的发布之前更新自己的代码。对于像Python 3.0这样的重要的新的发布（它广泛地影响到已有的代码），情况并非如此，但在大多数情况下，这是成立的。

例如，在Python 2.2中，`yield`是一个可选的扩展，但是，它在Python 2.3中是一个标准的关键字。它和生成器函数联合使用。这是Python违反向后兼容的一小部分实例中的一个。然而，`yield`随着时间逐步变化：在Python 2.2中它开始产生废弃警告，并且直到Python 2.3都还没有使用。

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。参见本书第2章对Jython的概要介绍。

类似地，在Python 2.6中，`with`和`as`变成了用于上下文管理器（异常处理的一种新形式）的新的保留字。这两个单词在Python 2.5中不是保留字，除非用一个`from __future__ import`（本书稍后介绍）手动打开环境管理器功能。使用Python 2.5的时候，`with`和`as`产生关于将要发生的变化的警告——除非是在Python 2.5中的IDLE版本中，它似乎已经支持此功能（即，在Python 2.5中使用这些单词作为变量名称确实会产生错误，但只是在其IDLE GUI的版本中会产生错误）。

命名惯例

除了这些规则外，还有一组命名惯例——这些并非必要的规则，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（第22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30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Python程序员通常会遵循的各种其他惯例。例如，本书后面会看见类变量名通常以一个大写字母开头，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17章，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。

在不同时刻把相同的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
```

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程序设计具有的决定性的优点^{注4}。第17章会介绍变量名也会存在于所谓的作用域内，作用域定义了变量名在哪里可以使用；对一个名字赋值的地点，决定了它在哪里可见。

注意：要了解其他的命名建议，参见Python的半官方风格指南PEP 8中的“Naming conventions”。可以从<http://www.python.org/dev/peps/pep-0008>访问该指南，或者通过Web搜索“Python PEP 8”。从技术上讲，这个文档把Python库代码的编码标准形式化了。

尽管很有用，但编码标准通常的缺陷在这里也适用。首先，PEP 8所附带的细节比你在本书中目前为止所了解的要更详细。并且，坦率地说，它变得比需要的更为复杂、严格和主观——其一些建议根本没有被实际工作的Python程序员普遍接受和遵守。此外，当今使用Python的一些最为优秀的公司，它们有自己的不同的编码标准。

然而，PEP 8确实包含了Python知识中的一些有用的规则，并且，对于Python初学者来说，它是很好的读物，只要你把它的推荐当做指南，而不是真理。

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存储，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过程。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。函数和方法的调用写在函数/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）。

注4：如果你用过像C++这样更加严格限定的语言，可能会想知道，Python中并没有所谓的C++的const声明的概念。有些对象是不可变的，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（如果你对隐藏属性感兴趣，可以参阅第24章对_X模块名的介绍，第30章对_X类名的介绍，以及第38章中的Private和Public类装饰器示例）。

表11-4：常见Python表达式语句

运算	解释
<code>spam(eggs, ham)</code>	函数调用
<code>spam.ham(eggs)</code>	方法调用
<code>spam</code>	在交互模式解释器内打印变量
<code>print(a, b, c, sep='')</code>	Python 3.0中的打印操作
<code>yield x ** 2</code>	产生表达式的语句

表11-4中的最后两条是特殊情况——正如我们在本章稍后所见到的，在Python 3.0中，打印是一个函数调用而通常独自编写为一行，并且生成器函数（第20章介绍）中的`yield`操作通常也编写为一条语句。这二者都只是表达式语句的实际例子。

例如，尽管我们通常在独自一行上运行一个`print`调用，它像任何其他函数调用一样返回一个值（它的返回值是`None`，这是那些不返回任何有意义内容的函数的默认返回值）：

```
>>> x = print('spam')           # print is a function call expression in 3.0
spam
>>> print(x)                     # But it is coded as an expression statement
None
```

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末尾的（第15章的）“常见编写代码的陷阱”小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。

打印操作

在Python 中，`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，这是把一个或多个对象转换为其文本表达形式，然后发送给标准输出或另一个类似文件的流。更详细地说，在Python中，打印与文件和流的概念紧密相连。

文件对象方法

在第9章，我们学习了写入文件的文件对象方法（例如，`file.write(str)`）。打印操作是类似的，但更加专注——文件写入方法是把字符串写入到任意的文件，`print`默认地把对象打印到`stdout`流，添加了一些自动的格式化。和文件方法不同，在使用打印操作的时候，不需要把对象转换为字符串。

标准输出流

标准输出流（通常叫做`stdout`）只是发送一个程序的文本输出的默认的地方。加上标准输入流和错误流，它只是脚本启动时所创建的3种数据连接中的一种。标准输出流通常映射到启动Python程序的窗口，除非它已经在操作系统的shell中重定向到一个文件或管道。

由于标准输出流在Python中可以作为内置的`sys`模块中的`stdout`文件对象使用（例如，`sys.stdout`），用文件的写入方法调用来模拟`print`成为可能。然而，`print`很容易使用，这使得很容易就能把文本打印到其他文件或流。

打印是Python 3.0和Python 2.6差异最明显的地方之一。实际上，这种差异通常是大多数Python 2.X代码不经修改就无法在Python 3.X下运行的首要原因。特别的，你编写打印操作的方式取决于所使用的Python的版本：

- 在Python 3.X中，打印是一个内置函数，用关键字参数来表示特定模式。
- 在Python 2.X中，打印是语句，拥有自己的特定语法。

由于本书既包含Python 3.0也包括Python 2.6，我们将依次看看每种打印情况。如果你

有幸能够只使用针对一种版本的Python编写的代码，请选择与你相关的那部分内容；然而，由于你的情况可能会有所变化，两种情况都熟悉也无伤大雅。

Python 3.0的print函数

严格地讲，在Python 3.0中，打印不是一种单独的语句形式，它只是我们在前面小节中所学习过的表达式语句的一个例子。

`print`内置函数通常在其自身的一行中调用，但是，它不会返回我们所关心的任何值（从技术上讲，它返回`None`）。因为它是一个常规的函数，因此，在Python 3.0中打印使用标准的函数调用语法，而不是一种特殊的语句形式。因为它通过关键字参数提供了一种特殊的操作模式，这种形式更加通用，并且能更好地支持未来的扩展。

相比较而言，Python 2.6的`print`语句有一些临时性的语法来支持末行抑制和目标文件这样的扩展。此外，Python 2.6语句根本不支持分隔符指定；在Python 2.6中，我们提前构建字符串比在Python 3.0中更常见。

调用格式

从语法上讲，调用Python 3.0的`print`函数有如下的形式：

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])
```

在这个正式的表示中，方括号中的项是可选的，并且可能会在一个给定的调用中省略，并且`=`后面的值都给出了参数的默认值。这个内置的函数把字符串`sep`所分隔开的一个或多个对象的文本表示，后面跟着的字符串`end`，都打印到流`file`中。

`sep`、`end`和`file`部分如果给出的话，必须作为关键字参数给定——也就是说，必须使用一种特殊的“`name=value`”语法来根据名称而不是位置来传递参数。本书第18章将深入介绍关键字参数，但是，它们很容易使用。发送给这个调用的关键字参数可以以任何的从左到右的顺序显示，这些参数跟在要打印的对象的后面，并且，它们控制`print`操作：

- `sep`是在每个对象的文本之间插入的一个字符串，如果没有传递的话，它默认地是一个单个的空格；传递一个空字符串将会抑制分隔符。
- `end`是添加在打印文本末尾的一个字符串，如果没有传递的话，它默认的是一个`\n`换行字符。传递一个空字符串将会避免在打印的文本的末尾移动到下一个输入行——下一个`print`将会保持添加到当前输出行的末尾。
- `file`指定了文本将要发送到的文件、标准流或者其他类似文件的对象；如果没有传递的话，它默认的是`sys.stdout`。带有一个类似文件的`write(string)`方法的任何对象都可以传递，但真正的文件应该已经为了输出而打开。

要打印的每个对象的文本表示，通过把该对象传递给`str`内置函数调用而获得；正如我们已经看到的，这个内置函数针对任何对象返回一个“用户友好的”显示字符串。^{注5}根本没有参数的时候，`print`函数直接把一个换行字符打印到标准输出流，它通常显示为一个空白的行。

Python 3.0的`print`函数的应用

Python 3.0中的打印可能比它所暗含的细节要简单一些。为了说明这点，让我们运行一些快速示例。下面把各种对象类型打印到默认的标准输出流，带有一个默认的分隔符和行末格式化添加（这里都是默认值，因为它们是最常见的例子）。

```
C:\misc> c:\python30\python
>>>
>>> print()                                # Display a blank line

>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>>
>>> print(x, y, z)                          # Print 3 objects per defaults
spam 99 ['eggs']
```

这里不需要把对象转换为字符串，而在文件写入方法中则需要这么做。默认情况下，`print`调用在打印的对象之间添加一个空格。要取消这个空格，给`sep`关键字参数发送一个空字符串，或者发送一个自己所选择的替代分隔符：

```
>>> print(x, y, z, sep='')                  # Suppress separator
spam99['eggs']
>>>
>>> print(x, y, z, sep=', ')                # Custom separator
spam, 99, ['eggs']
```

默认情况下，`print`添加一个行末字符来结束输出行。你可以通过向`end`关键字参数传递一个空字符串来抑制这一点并避免换行，或者可以传递一个自己的不同的终止符（包含一个`\n`符号来手动地换行）：

```
>>> print(x, y, z, end='')                  # Suppress line break
spam 99 ['eggs']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)    # Two prints, same output line
spam 99 ['eggs']spam 99 ['eggs']
```

注5：从技术上讲，打印在Python的内部实现中使用了等价的`str`，但效果也是相同的。除了这一项字符串转换的作用，`str`还是字符串类型的名称，可以用一个额外的编码参数把`raw`字节解码为Unicode字符串，我们将在第36章学习这一点；后者属于高级用法，我们在这里暂且略过。

```
>>> print(x, y, z, end='...\n')           # Custom line end
spam 99 ['eggs']...
>>>
```

也可以组合关键字参数来指定分隔符和行末字符串——它们可以以任何顺序出现，但是必须出现在所有要打印的对象的后面：

```
>>> print(x, y, z, sep='...', end='!\n')    # Multiple keywords
spam...99...['eggs']!
>>> print(x, y, z, end='!\n', sep='...')    # Order doesn't matter
spam...99...['eggs']!
```

这里展示了如何使用file关键字——它在单个打印的过程中，直接把文本打印到一个输出文件或者其他的可兼容对象（这其实是流重定向的一种形式，我们在本小节稍后回顾这一主题）：

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))  # Print to a file
>>> print(x, y, z)                                         # Back to stdout
spam 99 ['eggs']
>>> print(open('data.txt').read())                         # Display file text
spam...99...['eggs']
```

最后，别忘了，print操作提供的分隔符和行末选项只是为了方便起见。如果你需要显示更具体的格式，不要以这种方式打印，相反，提前构建一个更复杂的字符串或者在print自身之中使用我们在第7章介绍的字符串工具，并一次性打印该字符串：

```
>>> text = '%s: %-.4f, %05d' % ('Result', 3.14159, 42)
>>> print(text)
Result: 3.1416, 00042
>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))
Result: 3.1416, 00042
```

我们将在下一小节中看到，几乎在Python 3.0的print函数中见到的所有内容也可以直接应用于Python 2.6的print语句，这是有意义的，因为该函数的本意就是模拟并改进Python 2.6的打印支持。

Python 2.6 print语句

正如前面所提到的，Python 2.6中的打印使用具有独特和具体的语法的一条语句，而不是一个内置函数。实际上，Python 2.6的打印几乎是同一主题的一个变体，除了分隔符字符串（在Python 3.0中支持，但Python 2.6不支持），我们在Python 3.0的print函数中可以做的所有事情都可以直接转换到Python 2.6的print语句中。

语句形式

表11-5列出了Python 2.6中的print语句的形式，并且给出了它们在Python 3.0中的等价形

式以供参考。注意，`print`语句中的逗号很重要，它分隔开要打印的对象，并且最终的逗号抑制了通常添加到打印文本末尾的行末字符（不要和元组的语法搞混淆了）。`>>>`语法通常用作位右移操作，在这里也用到了，用来指定一个目标输出流而不是使用默认的`sys.stdout`。

表11-5: Python 2.6 `print`语句形式

Python 2.6语句	Python 3.0对等形式	说明
<code>print x, y</code>	<code>print(x, y)</code>	把对象的文本形式打印到 <code>sys.stdout</code> ，并且在各项之间添加一个空格，在末尾添加一个行末
<code>print x, y,</code>	<code>print(x, y, end='')</code>	同样，但是不会在文本末尾添加行末
<code>print >> afile, x, y</code>	<code>print(x, y, file=afile)</code>	把文本发送到 <code>myfile.write</code> 而不是 <code>sys.stdout.write</code>

Python 2.6 `print`语句应用

尽管Python 2.6 `print`有着与Python 3.0的函数相比更为独特的语法，但同样容易使用。让我们再次看一些基础的示例。默认情况下，Python 2.6 `print`语句在逗号分隔的项之间添加一个空格，并且在当前输出行的末尾添加一个换行：

```
C:\misc> c:\python26\python
>>>
>>> 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（以便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（下面是两条语句写在一行之中，之间用一个分号隔开）。

```
>>> print x, y,; print x, y
a b a b
```

要取消各项之间的空格，再一次，不要以这种方式打印。相反，使用第7章介绍的字符串合并和格式化工具来构建一个输出字符串，并且一次性打印该字符串：

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```


你将会看到，除了使用模式的特殊语法，Python 2.6的print语句基本上与Python 3.0的函数一样易于使用。下一小节将会介绍在Python 2.6的打印中指定文件的方式。

打印流重定向

在Python 3.0和Python 2.6中，打印都默认地发送到标准输出流。然而，通常发送到其他的地方也是有用的，例如发送到一个文本文件，以保存结果供以后使用和测试。尽管这样的重定向可以在Python自身之外的系统shell中实现，事实上，在脚本中重定向一个脚本的流也是很容易做到的。

Python的“hello World”程序

让我们从通常的（并且大多数时候是无意义的）语言基准线开始——“hello world”程序。要在Python中打印“hello world”信息，只需在各个版本中打印这个字符串：

```
>>> print('hello world')           # Print a string object in 3.0
hello world
>>> print 'hello world'           # Print a string object in 2.6
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：

```
>>> 'hello world'                  # Interactive echoes
'hello world'
```

这段代码在语言学习中并非举足轻重，但它说明了打印的行为。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, Y)                        # Or, in 2.6: print X, Y
```

等价于：

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

它通过`str`手动执行一次字符串转换，再通过“+”增加一个分隔符和一个换行，并且调用输出流的`write`方法。你宁愿编写哪种代码？（这里希望强调打印的程序员友好的本质。）

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`语句所做的是有用处的，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 a 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`边栏部分所做的说明，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`重设为修改每条`print`语句或者使用系统`shell`重定向语法提供一种方便的替代方式。

自动化流重定向

通过赋值`sys.stdout`而将打印文字重定向的技巧实际上非常常用。但是，上一节代码中还有个潜在的问题，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，你可以存储它，需要时恢复它^{注6}。

注6： 在Python 2.6和Python 3.0中，你也可以使用`sys`模块中的`__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__`从而回到原始流的值。参考库手册中`sys`模块的更多细节。

```

C:\misc> c:\python30\python
>>> import sys
>>> temp = sys.stdout
>>> sys.stdout = open('log.txt', 'a')
>>> print('spam')
>>> print(1, 2, 3)
>>> sys.stdout.close()
>>> sys.stdout = temp

>>> print('back here')
back here
>>> print(open('log.txt').read())
spam
1 2 3

```

然而正如你所见到的，像这样的手动保存和恢复原始的输出流包含了相当多的额外工作。因为这种操作出现的相当频繁，一个print扩展功能使得它显得多余。

在Python 3.0中，file关键字允许一个单个的print调用将其文本发送给一个文件的write方法，而不用真正地重设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。在Python 2.6中，当print语句以>>开始，后面再跟着输出的文件对象（或其他兼容对象）时，会有同样的效果。例如，如下语句再次把打印的文本发送到一个名为log.txt的文件：

```

log = open('log.txt', 'a')
print(x, y, z, file=log)
print(a, b, c)

log = open('log.txt', 'a')
print >> log, x, y, z
print a, b, c

```

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：

```

C:\misc> c:\python30\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)
>>> print(4, 5, 6, file=log)
>>> 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消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('Bad!' * 8, file=sys.stderr)           # 2.6: print >> sys.stderr, 'Bad' * 8
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

既然已经了解了打印重定向的所有内容，打印和文件写入方法之间的对等性就相当清楚了。如下交互会话在Python 3.0中使用了两种打印，然后把输出重定向到一个外部文件中，以验证打印了相同的文本：

```
>>> X = 1; Y = 2
>>> print(X, Y)                                # Print: the easy way
1 2
>>> import sys                                # Print: the hard way
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))        # Redirect text to file

>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n')  # Send to file manually
4
>>> print(open('temp1', 'rb').read())            # Binary mode for bytes
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

正如你所看到的，除非你喜欢录入，`print`操作通常是显示文本的最佳选择。对于打印和文件写入之间的对等性的另一个示例，请参见本书第18章中的Python 3.0的`print`函数模拟示例；它使用这种代码模式，提供了与Python 2.6中用法等效的Python 3.0通用`print`函数。

版本独立的打印

最后，如果你没有限制在Python 3.0下工作，但仍然想要打印能够与Python 3.0兼容，还有一些选择。其中之一就是，可以编写Python 2.6的`print`语句并且使用Python 3.0的`2to3`转换脚本自动将它们转换为Python 3.0函数调用。参见Python 3.0的文档以更详细地了解这一脚本，它试图把Python 2.X代码转换为可在Python 3.0下运行。

此外，可以在Python 2.6代码中编写Python 3.0的`print`函数，通过像下面这样一条语句支持该函数调用的变体：

```
from __future__ import print_function
```

这条语句把Python 2.6修改为支持Python 3.0的`print`函数。通过这种方法，我们可以使用Python 3.0 `print`功能，并且如果随后迁移到Python 3.0的话不必修改`print`。

还要记住，简单的打印，就像表11-5的头两行那样，在两个Python版本中都有效，因为任何表达式都可以包含在圆括号中，我们总是可以在Python 2.6中通过添加外围的圆括号，来伪装成调用一个Python 3.0 `print`函数。这么做的唯一缺点是，如果有多个要打印对象的话，它会产生一个元组(超出了已打印的对象)，因为它们会打印出额外的外围圆括号。例如，在Python 3.0中，调用的圆括号中可能列出任意多个对象：

```
C:\misc> c:\python30\python
>>> print('spam')                                # 3.0 print function call syntax
spam
>>> print('spam', 'ham', 'eggs')                  # These are mutiple arguments
spam ham eggs
```

第一个在Python 2.6中也同样地工作，但是，第二个会在输出中产生一个元组：

```
C:\misc> c:\python26\python
>>> print('spam')                                # 2.6 print statement, enclosing parens
spam
>>> print('spam', 'ham', 'eggs')                  # This is really a tuple object!
('spam', 'ham', 'eggs')
```

要真正做到可移植，可以把打印字符串格式化为一个单个的对象，使用字符串格式化表达式或方法调用，或者我们在第7章介绍的其他字符串工具：

```
>>> print('%s %s %s' % ('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('{0} {1} {2}'.format('spam', 'ham', 'eggs'))
spam ham eggs
```

当然，如果你专门地使用Python 3.0的话，可以完全忘记这一映射，但很多程序员不编写Python 2.X的代码和系统的话，至少会遇到它们。

注意：我在整本书中使用Python 3.0的`print`函数调用。我通常将会提醒你，该结果在Python 2.6下可能有额外的外围圆括号，因为多个项是一个元组，但我有时候不会提醒你，因此，请把这个提醒当作一个额外的警告——如果你在Python 2.6的打印文本中看到额外的圆括号的话，要么在`print`语句中删除圆括号，使用这里给出的版本独立方案重新编写打印代码；要么学习习惯多余的文本。

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，它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的代码段。

```
class FileFaker:
    def write(self, string):
        # Do something with printed text in 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Python 3.0中使用文件关键字参数以及在Python 2.6中使用print的扩展形式>>，这种对象的重定向甚至变得更简单，因为我们不再需要刻意重设sys.stdout——常规的print仍然定向到stdout流：

```
myobj = FileFaker()                                # 3.0: Redirect to object for one print
print(someObjects, file=myobj)                     # Does not reset sys.stdout

myobj = FileFaker()                                # 2.6: same effect
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
```

Python的打印操作重定向工具实质上是这些Shell语法形式的纯Python替代。

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。

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。只有对列表或字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. 要把一个单个的打印操作打印到一个文件，可以使用Python 3.0的`print(X, file=F)`调用形式，使用Python 2.6的扩展的`print >> file, X`语句形式，或者在打印前把`sys.stdout`指定为手动打开的文件并在之后恢复最初的值。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到一个文件，但这是在Python的范围之外的内容了。

if测试和语法规则

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。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组合语句（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，以及一个最终可选的else块。测试和else部分都有一个相关的嵌套语句块，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>
```

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：当在这里使用的基本接口中交互地输入时，提示符在接下去的行中变成...（在IDLE中，你只会直接向下到缩进的行，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，可改写成下面的程序：

```
>>> 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下面的嵌套语句，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。

```
>>> 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：当没有键匹配时，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，字典默认值能编码到has_key测试、get方法调用或异常捕捉中。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。如下是通过get方法处理默认值的情况。

```
>>> 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
```

位于一条if语句中的in成员测试也有同样的默认效果：

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

字典适用于将值和键相关联，但如果通过if语句来编写的更复杂动作呢？在第四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，通常写成函数名或lambda，通过增加括号调用来触发其动作。第19章将就此继续深入介绍。

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，就遵循简易性原则和可读性原则。

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叫做控制流程，像if这类会对其产生影响的语句，通常叫做控制流程语句。
- **块和语句的边界会自动检测。**就像我们所见到的，Python的程序块中没有大括号或“begin/end”等分隔字符；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。缩进语句叫做块（有时叫做组）。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忽略。**文件中空白行将忽略（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忽略（除了在字符串常量内，以及用在缩进时）。注释总是忽略：它们以#字符开头（不是在字符串常量内），而且延伸至该行的末尾。
- **文档字符串（docstring）会忽略，但会保存并由工具显示。**Python支持的另一种注释，叫做文档字符串（简称docstring）。和#注释不同的是，文档字符串会在运行时保留下来以便查看。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识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。

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')
```

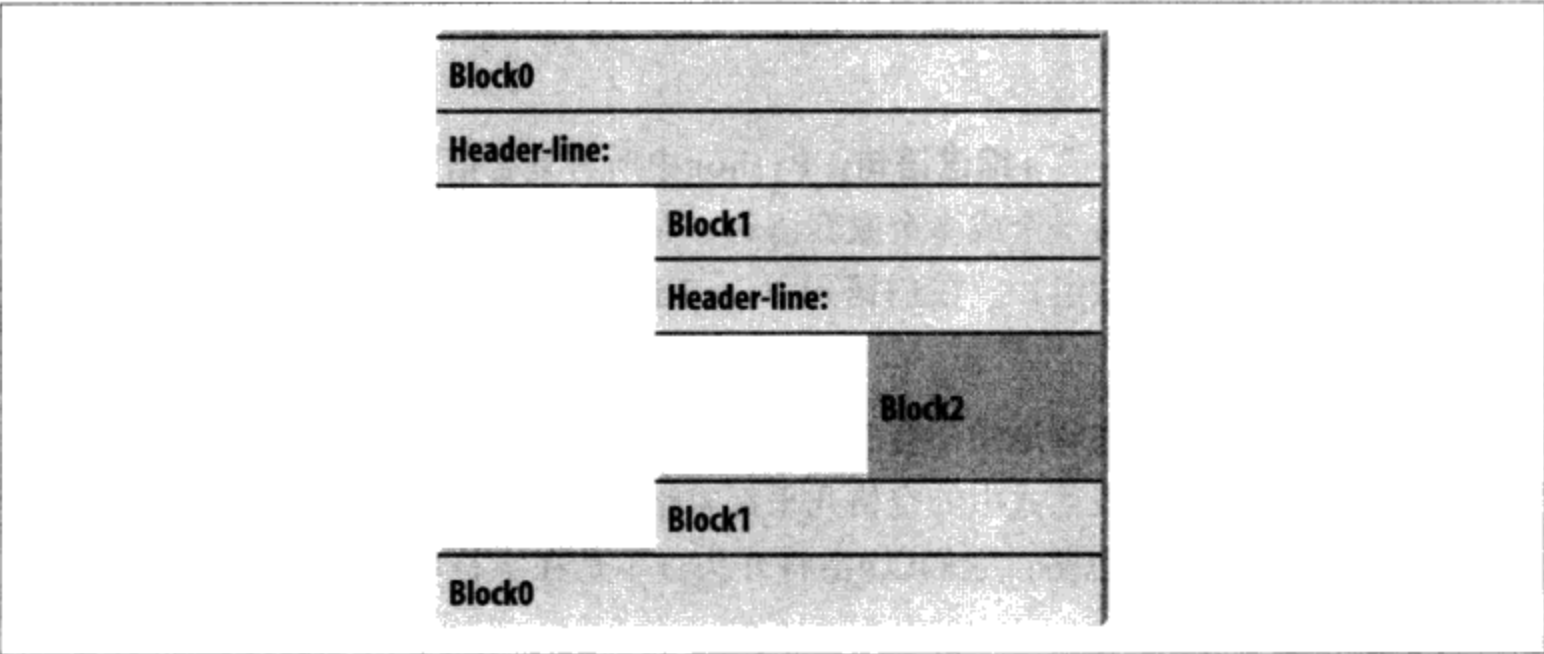


图12-1：嵌套块代码：一个嵌套块以再往右缩进的语句开始，碰到缩进量较少的语句或文件末尾时就结束

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。每个缩进层级使用4个空格或者一个制表符，这是通常的惯例，但是Python世界中没有绝对的标准。

缩进代码实际上是相当自然的事情。例如，如下的代码段（肯定很傻）展示了Python代码中通常的缩进错误：

```
x = 'SPAM'                                # Error: first line indented
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'                             # Error: unexpected indentation
    if x.endswith('NI'):
        x *= 2
        print(x)                         # Error: inconsistent indentation
```

这段代码的正确的缩进版本如下所示——即便对于这样的一个人工编写的示例，正确的缩进也会使得代码看上去更好：

```
x = 'SPAM'
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'
    if x.endswith('NI'):
        x *= 2
        print(x)                        # Prints "SPAMNISPAMNI"
```

在Python中，知道空白的一种主要用途就是用于代码左侧作为缩进，这点很重要。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，因为你不需明确标识嵌套代码块的开头和结尾，其他语言中常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。

缩进是再自然不过的事情了，并且它使得你的代码反映出了其逻辑结构。一致性的缩进代码总是可以满足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。

避免混合使用制表符和空格：Python 3.0中的新的错误检查

一条首要的规则是：尽管可以使用空格或制表符来缩进，在一段代码块中混合使用这两者通常不是好主意，请使用其中的一种。从技术上讲，制表符考虑到保留足够的空间以便把当前的栏数按照8的倍数来移动，并且，如果持续混合制表符和空格的话，代码也可以工作。然而，这样的代码可能很难修改。更糟糕的是，混合制表符和空格会使得代码难以阅读——制表符在另一个程序员的编辑器中看上去与在你的编辑器中的样子有很大不同。

实际上，当一段脚本在代码块中混合使用制表符和空格来缩进的时候（也就是说，以使得缩进依赖于制表符在空格上的等价形式），恰恰由于这些原因，Python 3.0现在发布了一个错误。Python 2.6允许这样的脚本运行，但是，它有一个`-t`命令行标志，会警告你制表符用法上的不一致，还有一个`-tt`标志会对这样的代码产生错误（你可以在一个系统shell窗口中使用诸如`python -t main.py`的命令行来切换）。Python 3.0的错误情况等同于Python 2.6的`-tt`切换。

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（超出该语句之外的第2行）可在任何缩进层次开始，而且应该尽可能让它们垂直对齐以便于阅读。这一开放对的规则也涉及Python 3.0中的集合和字典解析。
- **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。这种方法容易导致错误：偶尔忘掉一个`\`通常会产生语法错误，并且可能导致下一行默默地被错误地看做一条新语句，这会产生不可预期的结果。
- **字符串常量有特殊规则。**正如我们在第7章中所了解到的，三重引号字符串块可以横跨数行。我们还在第7章中学到过，相邻的字符串常量是隐式地连接起来的，当与前面提到的开放对规则一起使用的时候，把这个结果包含到圆括号中就可以允许它跨越多行。
- **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。

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、元组和生成器表达式，以及可以放到花括号中的任何内容（字典以及Python 3.0中的集合常量、集合和字典解析）等内容。其中的一些是我们将要在后面的各章中学习的工具，但这条规则自然地涉及实际应用中大多数跨行的结构。

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，你通常可以改用开放对技术——直接把语句的部分包含在圆括号中：

```
if (a == b and c == d and
    d == e and e == f):
    print('new')             # But parentheses usually do too
```

实际上，反斜线不太好用，因为它们太容易不被注意并且太容易漏掉。在下面的例子中，x通过反斜杠赋值为10，这是本来的意图；如果偶然漏掉了反斜杠，那么，x赋值为6，并且不会报告错误（+4本身是一个有效的表达式语句）。

在带有一个复杂赋值的实际程序中，这可能会引发一个非常令人讨厌的bug^{注1}：

```
x = 1 + 2 + 3 \
+4                # Omitting the \ makes this very different
```

注1：坦率地讲，这没有从Python 3.0中删除，有点令人惊讶，因为Python 3.0有了某些其他修改（参见前言的表P-2中针对Python 3.0删除内容的列表，有些删除的内容与连续的反斜线的危险相比显得不足挂齿）。再次强调，本书的目标是教授Python，而不是惹起群情激奋，因此，我的建议很简单：不要那么做。

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，如果你坚持多数代码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
```

正如第7章所介绍的，三重引号字符串常量也跨多行。此外，如果两个字符串常量彼此相邻地出现，它们会合并，就好像在它们之间已经放置了一个+——当和开放对规则一起使用的时候，包括在圆括号中就允许这种形式跨越多行。例如，如下的第一个示例在换行处插入换行字符，并且把'\naaaa\nbbbb\ncccc'赋给S，而第二个示例隐式地合并，并且把S赋值为'aaaabbbbcccc'。第二种形式中的注释忽略了，但是，在第一个示例中则包含了：

```
S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
     'bbbb'
     'cccc')                # Comments here are ignored
```

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：

- 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 True or False (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。这通常叫做短路计算，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，看一看本章的边栏部分“为什么要在意布尔值”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。下一小节也介绍了利用这一行为的一种常用方式，以及在最新的Python版本中的该方法的替代方法。

if/else三元表达式

Python中布尔运算符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，而不是将其结果赋值给变量。因此（坦白地讲，因为C语言有类似工具）^{注2}，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：

注2：事实上，Python的`XifYelseZ`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。

```
A = Y if X else Z
```

这个表达式和前边四行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之前（以及2.5版以后，如果你坚持的话），相同的效果可以小心地用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，Python 2.5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'
>>> ['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，在Python 2.5中你最好还是使用更简单、更易懂的if/else表达式。不过，你还是应该少用，只有当组成成分都很简单时才用。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，你还是会看到Python 2.5之前的代码中看到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一种常见方式就是，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）。在更简单的形式中，这也通常用来指定一个默认值，下面的例子中，如果A为真（或非空）的话将X设置为A，否则，将X设置为default：

```
X = A or default
```

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: ...
```

你已经在本章看过这个行为的另一种应用了：基于布尔运作方式，表达式((A and B) or C)几乎可用来模拟if/else语句（参见本章对这一形式的详细讨论）。

在前面的章节我们遇到了额外的布尔用法示例。正如我们在第9章看到的，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。正如我们在第5章中学到的，预先设置的布尔值True和False与整数1和0是相同的，并且对于初始化变量（X = False）、循环测试（while True:）以及在交互提示模式中显示结果是很有用的。

还请参阅本书第六部分中对于运算符重载的讨论：当我们用类定义新的对象类型的时候，我们可以用__bool__或__len__方法指定其布尔特性（在Python 2.6中__bool__叫做__nonzero__）。如果前者通过返回一个长度为0而成为空缺的并指定为假的对象的话（一个空的对象看做是假），将测试后者。

本章小结

在这一章，我们研究了Python的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，并比先前更深入地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. 怎样使单个语句横跨多行？
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，返回Z。这相当于4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，该语句之外的第2行可以以任意缩进层级开始。
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。它们可以用来进行真测试、变量初始化，以及在交互提示模式中打印表达式结果。

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，用它来遍历序列对象内的元素，并对每个元素运行一个代码块。

我们已经非正式地见过这两种循环，但在这里，我们介绍一些其他的有用的细节。此外，我们也会在这里研究一些在循环中不太常用的语句（例如，break和continue），并且会介绍循环中常用的一些内置函数（例如range、zip和map）。

尽管这里介绍的while和for语句是用来编写重复操作的主要语法，但Python中还是有其他的循环操作和概念。因此，下一章将继续介绍迭代，我们将介绍和Python的迭代协议（for循环用到的）以及列表解析（for循环的近亲）相关的概念。稍后的各章介绍了更加奇特的迭代工具，如生成器、filter和reduce。现在，让我们从最基础的内容学起。

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。

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

及一个可选的else部分（控制权离开循环而又没有碰到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，直到字符串为空返回假为止。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其他方法。

```
>>> x = 'spam'
>>> while x:                # While x is not empty
...     print(x, end=' ')
...     x = x[1:]           # Strip first character off x
...
spam pam am m
```

注意：这里使用end=' '关键字参数，使所有输出都出现在同一行，之间用空格隔开；如果你忘了为什么这么做，请参阅第11章。下面的代码会从a的值向上计算到b的值（但不含b）。稍后，我们会以Python for循环和内置range函数来实现，更为简单的程序如下所示：

```
>>> a=0; b=10
>>> while a < b:            # One way to code counter loops
...     print(a, end=' ')
...     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，下一节我们将学习break语句。

break、continue、pass和循环else

现在，我们已看过一些Python循环的例子，接下来看两个简单的语句，它们只有嵌套在循环中时才起作用：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子句，因为它和break关联在一起。此外，还要学习Python的空占位语句pass（它本身与循环没什么关系，但属于简单的单个单词语句的范畴）。在Python中：

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。

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。

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`。

```
while True: pass                # Type Ctrl-C to stop me!
```

因为主体只是空语句，Python陷入了死循环。就语句而言，`pass`差不多就像对象中的`None`一样，表示什么也没有。注意：在冒号之后，`while`循环主体和首行处在同一行上；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，坦率地讲，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，忽略`try`语句所捕获的异常，以及定义带属性的空类对象，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
```

我们无法保持函数体为空而不产生语法错误，因此，可以使用`pass`来替代。

注意：版本差异提示：Python 3.0（而不是Python 2.6）允许在可以使用表达式的任何地方使用...（三个连续的点号）来省略代码。由于省略号自身什么也不做，这可以当作是`pass`语句的一种替代方案，尤其是对于随后填充的代码——这是Python的“TBD”（未确定内容）的一种：

```
def func1():
    ...                # Alternative to pass

def func2():
    ...

func1()                # Does nothing if called
```

省略号可以和语句头出现在同一行，并且，如果不需要具体类型的话，可以用来初始化变量名：

```
def func1(): ...      # Works on same line too
def func2(): ...

>>> X = ...          # Alternative to None
>>> X
Ellipsis
```

这种表示法是Python 3.0中新增的（并且这超越了分片扩展中的“...”的最初意图），因此，它能否广泛流传以与pass和None的这类用法相抗衡，还需拭目以待。

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并大于或等于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, end=' ')
```

因为continue会跳到循环的开头，所以不需要在if测试内放置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, end=' ')
```

break

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体），通过input（在Python 2.6中叫做raw_input）输入数据，而当用户在请求的name处输入“stop”时就结束。

```
>>> while True:
...     name = input('Enter name:')
...     if name == 'stop': break
...     age = 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`input`是以字符串返回用户输入的数据的。在第35章中，你会看到`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`input`括起来。

循环else

和循环`else`子句结合时，`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 -= 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。

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。

注意：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没有小数位数的浮点数也会失败。还要注意，正如第5章所介绍的，由于/向“真除法”迁移的问题，在Python 3.0中必须使用`//`而不是`/`（我们需要初始的除法来截断余数，而不是保留余数）。如果你想实验这个程序代码，一定要看一看第四部分结尾的练习题，将其括在函数中。

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，而且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，也的确可以运行。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情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某些情况下并不适用。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。

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中会不小心打成“=”

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 = True
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。在学习序列对象类型的时候,我们曾经遇到过它;让我们在这里更正式地讨论其用法。

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>:
    <statements>
else:
    <statements>
```

Assign object items to target
Repeated loop body: use target
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依次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, end=' ')
...
spam eggs ham
```

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，我们会介绍一些工具，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, end=' ')          # Iterate over a string
...
l u m b e r j a c k

>>> for x in T: print(x, end=' ')          # 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，对于文件和字典也有效！

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。这只是元组解包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
...     print(a, b)
...
1 2
3 4
5 6

```

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。

这种形式通常和我们在本章后面所介绍的zip调用一起使用，以实现并行遍历。在Python中，它通常还和SQL数据库一起使用，其中，查询结果表作为这里使用的列表这样的序列的序列而返回——外围的列表就是数据库表，嵌套的元组是表中的行，元组赋值和列对应。

for循环中的元组使得用items方法来遍历字典中的键和值变得很方便，而不必再遍历键并手动地索引以获取值：

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])                # Use dict keys iterator and index
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)                # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

注意for循环中的元组赋值并非一种特殊情况，这一点很重要；单词for之后的任何赋值目标在语法上都是有效的。尽管我们总是在for循环中手动地赋值以解包：

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both                            # Manual assignment equivalent
...     print(a, b)
...
1 2
3 4
5 6
```

在遍历序列的序列的时候，循环头部的元组为我们节省了一个额外的步骤。正如第11章所介绍的，在一个for中，即便嵌套的结构也能够以这种方式自动解包：

```
>>> ((a, b), c) = ((1, 2), 3)                  # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

但这不是特殊情况——在每次迭代上，for循环直接运行我们在其之前运行的那种赋值。任何嵌套的序列结构都可以按照这种方式解包，只不过因为序列赋值是如此通用：

```
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]: print(a, b, c)
...
```

```
1 2 3
X Y 6
```

Python 3.0在for循环中扩展的序列赋值

实际上，由于for循环中的循环变量真的可以是任何赋值目标，在这里，我们也可以使用Python 3.0的扩展序列解包赋值语法，来提取序列中的序列的元素和部分。实际上，这也不是特殊情况，只不过是Python 3.0中的一种新的赋值形式（正如本书第11章所介绍），因为它在赋值语句中有效，它自动地在for循环中有效。

考虑前面小节介绍的元组赋值形式。在每次迭代时，值的一个元组赋给了名称的一个元组，就像是一条简单的赋值语句一样：

```
>>> a, b, c = (1, 2, 3)                                # Tuple assignment
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:            # Used in for loop
...     print(a, b, c)
...
1 2 3
4 5 6
```

在Python 3.0中，由于一个序列可以赋值给一组更为通用的名称（其中有一个带有星号的名称收集多个元素），我们可以在for循环中使用同样的语法来提取嵌套的序列的部分：

```
>>> a, *b, c = (1, 2, 3, 4)                            # Extended seq assignment
>>> a, b, c
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

实际上，这种方式可以用来从表示为嵌套序列的数据的行中选取多个列。在Python 2.X中，带星号的名称是不允许的，但是，我们可以通过分片实现类似的效果。唯一的区别是分片返回一个特定类型的结果，而星号名称总是赋值一个列表：

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:            # Manual slicing in 2.6
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

参见第11章了解这种赋值形式的更多内容。

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，这个示例就会比较易于编写。因为in会隐性地扫描列表来找到匹配，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!
```

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。因为文件包含了许多字符和行，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')                # Read contents into a string
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。

```

file = open('test.txt')
while True:
    line = file.readline()                # Read line by line
    if not line: break
    print(line, end='')                  # Line already has a \n

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)                 # Read byte chunks: up to 10 bytes
    if not chunk: break
    print(chunk)

```


通常是按照块读入二进制数据的。不过，逐行读取文本文件时，`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, end='')

for line in open('test.txt'):
    print(line, end='')           # Use iterators: best text input mode
```

文件`readlines`方法会一次把文件载入到行字符串的列表，这里的最后的例子则按照文件迭代器来自动在每次循环迭代的时候读入一行（迭代器将会在第14章中讨论）。参见库手册以了解关于这里用到的调用的更多内容。这里的最后一个例子通常是文本文件的最佳选择——它除了简单，还对任意大小的文件都有效，并且不会一次把整个文件都载入到内存中。迭代器版本可能会更快，但是在Python 3.0中I/O性能稍差。

在一些Python 2.X代码中，也可以看到`open`替代为`file`以及文件对象的较早的`xreadlines`，以实现与文件的自动行迭代器同样的效果（它就像是`readlines`，但是不会一次把文件载入到内存中）。Python 3.0中删除了`file`和`xreadlines`，因为它们都是多余的；也不能在Python 2.6中使用它们，但是，可能会在更早的代码和资源中出现。参阅第36章了解关于读取文件的更多内容，在那里将会看到，文本文件和二进制文件在Python 3.0中的含义有细微的差别。

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元素或每隔两个元素访问，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：

- 内置`range`函数返回一系列连续增加的整数，可作为`for`中的索引。
- 内置`zip`函数返回并行元素的元组的列表，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。在Python 3.0中，`range`是一个迭代器，会根据需要产生元素，因此，我们需要将其包含到一个`list`调用中以一次性显示其结果（第14章更详细地介绍迭代器）：

```
>>> list(range(5)), list(range(2, 5)), list(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：

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(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3行时，可以使用`range`产生适当的整数数字；在Python 3.0中，`for`循环迫使`range`的结果自动化，因此，我们在这里不必列出：

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
1 Pythons
2 Pythons
```

`range`也常用来间接地迭代一个序列。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。

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ')          # 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):
...     print(X[i], end=' ')                    # while loop iteration
```

```
...     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。这是一个多步骤的过程，但是，对于产生偏移值（而不是产生位于那些偏移值的元素）来说足够了：

```
>>> X
'spam'
>>> len(X)                                # Length of string
4
>>> list(range(len(X)))                    # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ')    # Manual for indexing
...
s p a m
```

注意，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和分片

上一节最后例子可以实现，但可能比它要求的更慢，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。

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
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，这可能不是如今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：

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

结果是相同的，但对我们来说更容易编写，对其他人来说更容易阅读。在这里，使用range唯一的真正优点是——它没有复制字符串，并且不会在Python 3.0中创建一个列表，对于很大的字符串来说，这会节省内存。

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。你可以通过简单的for循环来做，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，第一轮迭代中，x是整数1。下一轮迭代中，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一个已更新的值。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)):
...     L[i] += 1
...
>>> L
[2, 3, 4, 5, 6]
```

Add one to each item in L
Or L[i] = L[i] + 1

以这种方式编写时，随着循环的执行，列表中的内容会改变。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的解决方案依然不理想。

```
[x+1 for x in L]
```

这种形式的列表解析表达式也能做类似的工作，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，我们将在下一章对列表解析做一个完整的介绍。

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来创建一个元组对的列表（和range一样，zip在Python 3.0中也是一个可迭代对象，因此，我们必须将其包含在一个list调用中以便一次性显示所有结果——下一章将详细地介绍迭代器）：

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                # list() required in 3.0, not 2.6
[(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
```

在这里，我们步进获得了`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解包`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。

严格来讲，`zip`函数比这个例子所示意的更为一般化。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。对于3个参数，像如下的例子那样，它构建了3元素元组的一个列表，其中带有来自每个序列的元素，基本上按照列对应（从技术上讲，我们得到了`N`个参数的一个`N`维元组）。

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(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。在下面的例子中，我们把两个字符串`zip`到一起以并行地选取字符，但是结果所拥有的元组数只和最短的序列的长度一致：

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Python 2.6中的map的等价形式

在Python 2.X中，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（而不是按照最短的长度截断）：

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)                                # 2.X only
[('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，Python 3.0不再支持该函数。一般来讲，`map`会带一个函数，以及一个或多个的序列参数，然后用从序列中取出的并行元素调用函数的结果收集起来。我们将在第19章和第20章更详细地学习`map`，但是，作为一个

简单的例子，下面的代码根据字符串中的每一元素map内置的ord函数并收集结果（和zip一样，map是Python 3.0中的一个值生成器，因此必须传递给list以一次性收集其结果）：

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```

这段代码和下面的循环语句效果相同，但它通常更快些：

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```

注意：版本差异提示：使用以一个None为函数参数的map函数的退化形式，在Python 3.0中已经不再支持了，因为它和zip很大程度上重复了（并且，坦率地说，map的函数应用目的有点奇怪）。在Python 3.0中，请自行使用zip或编写循环代码来补充结果。在学习了一些额外的迭代概念之后，我们将会在第20章看到如何做到这点。

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。

```
>>> 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。

```
>>> list(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31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。在下一章中，我们会探讨一个相关但更丰富的概念，也就是列表解析，它通过单个表达式建立列表；我们还将回顾Python 3.0的字典解析，它可作为针对配对的键/值对的dict调用的一种替代方法。

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`函数返回一个生成器对象：这种对象支持下一章将要学习的迭代协议，本书下一部分会再深入讨论迭代协议。简而言之，这个对象有一个`__next__`方法，由下一个内置函数调用它，并且循环中每次迭代的时候它会返回一个`(index,value)`的元组。我们可以在`for`循环中通过元组赋值运算将元组解包（很像是使用`zip`）：

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x02765AA8>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

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，也就是下一章的主题）会自动执行迭代协议：

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
```

要像`enumerate`、`zip`和列表解析那样完全理解迭代的概念，我们需要继续学习下一章，以更加正式地剖析它。

本章小结

在本章中，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，并且介绍了几个在`for`循环中常用的内置工具，包括`range`、`zip`、`map`和`enumerate`（尽管它们在Python 3.0中的角色是迭代器，并且在下一章才会正式介绍迭代器的概念）。

在下一章中，我们将继续迭代器的话题，讨论Python中的列表解析和迭代协议，这是和`for`循环密切相关的概念。在那里，我们还将介绍这里所见到的可迭代工具的某些细节，例如`range`和`zip`。不过，就像往常一样，继续学习之前，先做一做这里的习题。

本章习题

1. `While`和`for`之间的主要功能区别是什么？
2. `break`和`continue`之间有何区别？

3. 一个循环的else分句何时执行?
4. 在Python中怎样编写一个基于计数器的循环?
5. 怎样使range用于for循环中?

习题解答

1. while循环是一条通用的循环语句，for循环设计用来在一个序列中遍历各项（序列需要是真正可迭代的）。尽管while可以用计数器循环来模拟for循环，但它需要更多的代码并且可能运行起来更慢些。
2. break语句立即退出一个循环（省略了下面的整个while或for循环语句），continue跳回到循环的顶部（跳转到while中测试之前的部分或for中的下一次元素获取）。
3. 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。
4. 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。只要可能就改用简单的for循环，不用range或计数器。这样做不仅更容易编写，而且通常运行得更快。
5. range内置函数可以用在一个for循环中来实现固定次数的重复，以按照偏移值而不是偏移值处的元素来扫描，从而，在过程中省略连续的元素，并且在遍历一个列表的时候修改它。这样的用法并非都需要range，大多数有其他的替代方法——如今，扫描实际的元素、三重限制分片，以及列表解析往往是较好的解决方案（尽管老练的C程序员倾向于统计东西的数目）。

迭代器和解析，第一部分

在上一章中，我们学习了Python的两种循环语句，`while`和`for`。尽管它们能够处理程序所需执行的大多数重复性任务，在序列中迭代的需求是如此常见和广泛，以至于Python提供了额外的工具以使其更简单和高效。本章开始介绍这些工具。尤其是，本章介绍了Python的迭代协议的相关概念——`for`循环所使用的一种方法调用模式，并且介绍了关于列表解析的一些细节，列表解析是对迭代中的项应用一个表达式的`for`循环的一种近似形式。

由于这些工具都和`for`循环及函数相关，我们将在本书中分两个步骤来介绍它们：本章介绍循环工具背景的基础知识，作为上一章的某种延续；第20章在基于函数的工具的背景中回顾它们。在本章中，我们还将展示Python中的额外迭代工具的示例，并接触在Python 3.0中可用的新的迭代器。

首先注意一点：这几章中所介绍的某些概念乍看起来可能有些高级。然而，通过实践，你将发现这些工具很有用并且很强大。尽管这些工具不是严格必需的，但它们已经变成了Python代码中的常用内容，如果你必须阅读他人所编写的程序，那么就应基本理解这些工具。

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, end=' ')
...
1 4 9 16
```

```
>>>for x in (1, 2, 3, 4): print(x ** 3, end=' ')
...
1 8 27 64

>>>for x in 'spam': print(x * 2, end=' ')
...
ss pp aa mm
```

实际上，for循环甚至比这更为通用：可用于任何可迭代的对象。实际上，对Python中所有会从左至右扫描对象的迭代工具而言都是如此，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，但它在语言的设计中很普遍。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，就看做是可迭代的。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^{注1}。

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，在第9章中，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')           # Read a 4-line script file in this directory
>>>f.readline()                     # readline loads one line on each call
'import sys\n'
>>>f.readline()
'print(sys.path)\n'
>>>f.readline()
'x = 2\n'
>>>f.readline()
'print(2 ** 33)\n'
>>>f.readline()                     # Returns empty string at end-of-file
''
```

如今，文件也有一个方法，名为__next__，差不多有相同的效果：每次调用时，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__会引发内置的StopIteration异常，而不是返回空字符串。

注1： 这个主题中的术语的使用有点随意。本章交替地使用“可迭代的”和“迭代器”来表示通常支持迭代的一个对象。有时候，术语“可迭代的”指的是支持iter的一个对象，而“迭代器”指的是iter所返回的一个支持next(I)的对象，但是，在Python世界或本书中，这种习惯并不是普遍通用的。

```

>>>f = open('script1.py')           # __next__ loads one line on each call too
>>>f.__next__()                     # But raises an exception at end-of-file
'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):
...more exception text omitted...
StopIteration

```

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 to read by lines
...     print(line.upper(), end='')   # Calls __next__, catches StopIteration
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)

```

注意，这里的`print`使用`end=''`来抑制添加一个`\n`，因为行字符串已经有了一个（如果没有这点，我们的输出将会变成两行隔开）。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(), end='')
...
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，基于迭代器的版本会根据每次发布而改进，所以它运行的也应该更快（Python 3.0通过重写I/O以支持Unicode文本从而使得这一优点不那么明显，并且更少依赖于系统）。

当然也可以用`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(), end='')
...
...same output...
```

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。然而，并非绝对如此，尤其是在Python 3.0中，随后我将介绍一种计时技术，可以用它来衡量像这样的替代方案的相对速度。

手动迭代：iter和next

为了支持手动迭代代码（用较少的录入），Python 3.0还提供了一个内置函数`next`，它会自动调用一个对象的`__next__`方法。给定一个可迭代对象`X`，调用`next(X)`等同于`X.__next__()`，但前者简单很多。例如，对于文件的任何一种形式都可以使用：

```
>>>f = open('script1.py')
>>>f.__next__()                                # Call iteration method directly
'import sys\n'
>>>f.__next__()
'print(sys.path)\n'

>>>f = open('script1.py')
>>>next(f)                                      # next built-in calls __next__
'import sys\n'
>>>next(f)
'print(sys.path)\n'
```

从技术角度来讲，迭代协议还有一点值得注意。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):
...more omitted...
StopIteration

```

最初的一步对于文件来说不是必需的，因为文件对象就是自己的迭代器。也就是说，文件有自己的`__next__`方法，因此不需要像这样返回一个不同的对象：

```

>>>f = open('script1.py')
>>>iter(f) is f
True
>>>f.__next__()
'import sys\n'

```

列表以及很多其他的内置对象，不是自身的迭代器，因为它们支持多次打开迭代器。对这样的对象，我们必须调用`iter`来启动迭代：

```

>>>L = [1, 2, 3]
>>>iter(L) is L
False
>>>L.__next__()
AttributeError: 'list' object has no attribute '__next__'

>>>I = iter(L)
>>>I.__next__()
1
>>>next(I)                                    # Same as I.__next__()
2

```

尽管Python迭代工具自动调用这些函数，我们也可以使用它们来手动地应用迭代协议。如下的交互展示了自动和手动迭代之间的对等性^{注2}：

注2：从技术上讲，`for`循环调用内部等价的`I.__next__`，而不是这里所使用的`next(I)`。这两者之间几乎没有区别，但是，我们将在下一小节中看到，Python 3.0中有一些内置的对象（例如`os.popen`的结果）支持前者而不支持后者，但仍然可以在`for`循环中迭代。手动的迭代通常可以使用任何一种调用形式。如果你关注详细情况，Python 3.0中的`os.popen`结果已经用`subprocess`模块和一个包装类重新实现，在Python 3.0中，其`__getattr__`方法不再针对`next`内置函数所做出的显式`__next__`获取而调用，而是针对按名称的显式获取而调用，这是我们将在第37章和第38章中学习的一个Python 3.0的修改，它似乎也用到了一些标准库代码。还是在Python 3.0中，相关的Python 2.6调用`os.popen2/3/4`不再可用，而是使用带有相应参数的`subprocess.Popen`（参见Python 3.0库手册可以了解新的必需的代码）。

```

>>>L = [1, 2, 3]
>>>
>>>for X in L:
...     print(X ** 2, end=' ')
...
1 4 9

>>>I = iter(L)
>>>while True:
...     try:
...         X = next(I)
...     except StopIteration:
...         break
...     print(X ** 2, end=' ')
...
1 4 9

```

Automatic iteration
Obtains iter, calls __next__, catches exceptions

Manual iteration: what for loops usually do
try statement catches exceptions
Or call I.__next__

要理解这段代码，你需要知道，`try`语句运行一个动作并且捕获在运行过程中发生的异常（我们将在本书第七部分深入介绍异常）。还应该注意，`for`循环和其他的迭代环境有时候针对用户定义的类不同地工作，重复地索引一个对象而不是运行迭代协议。等到我们在第29章中学习运算符重载的时候，再介绍这一内容。

注意： 版本差异提示：在Python 2.6中，迭代方法叫做`X.next()`而不是`X.__next__()`。为了可移植性，`next(X)`内置函数在Python 2.6中也是可用的（但在更早的版本中不可以），并且，调用Python 2.6的`X.next()`而不是Python 3.0的`X.__next__()`。在Python 2.6中所有其他方式中，迭代都是一样地工作的，只是在手动迭代中直接使用`X.next()`或`next(X)`，而不是Python 3.0的`X.__next__()`。在Python 2.6之前的版本中，使用手动`X.next()`调用而不是`next(X)`。

其他内置类型迭代器

除了文件以及像列表这样的实际的序列外，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地获取其键的列表。

```

>>>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，字典有一个迭代器，在迭代环境中，会自动一次返回一个键。

```

>>>I = iter(D)
>>>next(I)

```

```
'a'
>>>next(I)
'c'
>>>next(I)
'b'
>>>next(I)
Traceback (most recent call last):
...more omitted...
StopIteration
```

直接的效果是，我们不再需要调用`keys`方法来遍历字典键——`for`循环将使用迭代协议在每次迭代的时候获取一个键：

```
>>>for key in D:
...     print(key, D[key])
...
a 1
c 3
b 2
```

我们不能在这里深入细节，但是，其他的Python对象类型也支持迭代协议，因此，也可以在`for`循环中使用。例如，*shelves*（用于Python对象的一个根据键访问的文件系统）和`os.popen`的结果（读取shell命令的输出的一个工具）也是可迭代的：

```
>>>import os
>>>P = os.popen('dir')
>>>P.__next__()
' Volume in drive C is SQ004828V03\n'
>>>P.__next__()
' Volume Serial Number is 08BE-3CD4\n'
>>>next(P)
TypeError: _wrap_close object is not an iterator
```

注意，在Python 2.6中，`popen`对象支持一个`P.next()`方法。在Python 3.0中，它们支持`P.__next__()`方法，但不支持`next(P)`内置函数。由于后者定义来调用前者，这种形式在未来的发布中是否会保持还不清楚（正如前面的脚注所提到的，这似乎是一个实现问题）。然而，这只是手动迭代的一个问题，如果用`for`循环或者其他的迭代环境（下一小节介绍）来自动迭代这些对象，在任何Python版本中，它们都将返回连续的行。

迭代协议也是我们必须把某些结果包装到一个`list`调用中以一次性看到它们的值的原因。可迭代的对象一次返回一个结果，而不是一个实际的列表：

```
>>>R = range(5)
>>>R                                     # Ranges are iterables in 3.0
range(0, 5)
>>>I = iter(R)                           # Use iteration protocol to produce results
>>>next(I)
0
>>>next(I)
```

```

1
>>>list(range(5))                # Or use list to collect all results at once
[0, 1, 2, 3, 4]

```

既然对这一协议已经有了较深入的理解，我们应该能够看到它是如何说明上一章所介绍的`enumerate`工具能够以其方式工作的原因：

```

>>>E = enumerate('spam')        # enumerate is an iterable too
>>>E
<enumerate object at 0x0253F508>
>>>I = iter(E)
>>>next(I)                        # Generate results with iteration protocol
(0, 's')
>>>next(I)                        # Or use list to force generation to run
(1, 'p')
>>>list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]

```

我们通常不会看到这种机制，因为`for`循环为我们自动遍历结果。实际上，Python中可以从左向右扫描的所有对象都以同样的方式实现了迭代协议，包括下一小节所涉及的主题。

列表解析：初探

既然已经看到了迭代协议是如何工作的，让我们来看一个非常常用的例子。与`for`循环一起使用，列表解析是最常应用迭代协议的环境之一。

在上一章中，我们学习了，在遍历一个列表的时候，如何使用`range`来修改它：

```

>>>L = [1, 2, 3, 4, 5]
>>>for i in range(len(L)):
...     L[i] += 10
...
>>>L
[11, 12, 13, 14, 15]

```

这是有效的，但是正如我们所提到的，它可能不是Python中的优化的“最佳实践”。如今，列表解析表达式使得早先许多的例子变得过时了。例如，我们可以用产生所需的结果列表的一个单个表达式来替代该循环：

```

>>>L = [x + 10 for x in L]
>>>L
[21, 22, 23, 24, 25]

```

直接结果是相同的，但是它需要较少的代码，并且可能会运行的更快。列表解析并不完

全和for循环语句版本相同，因为它产生一个新的列表对象（如果有对最初的列表的多个引用，可能会有关系），但是，对于大多数应用程序来说它足够接近，并且是一种足够常见和方便的方法，值得在这里进一步介绍。

列表解析基础知识

我们在第4章简单介绍过列表解析。从语法上讲，其语法源自于集合理论表示法中的一个结构，该结构对集合中的每个元素应用一个操作，但是，要使用这个工具并不一定必须知道集合理论。在Python中，大多数人发现列表解析看上去就像是一个反向的for循环。

为了在语法上进行了解，让我们更详细地剖析前面的例子：

```
>>>L = [x + 10 for x in L]
```

列表解析写在一个方括号中，因为它们最终是构建一个新的列表的一种方式。它们以我们所组成的一个任意的表达式开始，该表达式使用我们所组成的一个循环变量(x+10)。这后边跟着我们现在应该看做是一个for循环头部的部分，它声明了循环变量，以及一个可迭代对象(for x in L)。

要运行该表达式，Python在解释器内部执行一个遍历L的迭代，按照顺序把x赋给每个元素，并且收集对各元素运行左边的表达式的结果。我们得到的结果列表就是列表解析所表达的内容——包含了x+10的一个新列表，针对L中的每个x。

从技术上讲，列表解析并非真的是必需的，因为我们总是可以用一个for循环手动地构建一个表达式结果的列表，该for循环像下面这样添加结果：

```
>>>res = []
>>>for x in L:
...     res.append(x + 10)
...
>>>res
[21, 22, 23, 24, 25]
```

实际上，这和列表解析所做的事情是相同的。

然而，列表解析编写起来更加精简，并且由于构建结果列表的这种代码样式在Python代码中十分常见，因此可以将它们用于多种环境。此外，列表解析比手动的for循环语句运行的更快（往往速度会快一倍），因为它们的迭代在解释器内部是以C语言的速度执行的，而不是以手动Python代码执行的，特别是对于较大的数据集合，这是使用列表解析的一个主要的性能优点。

在文件上使用列表解析

让我们来看看列表解析的另一个常见用例，从而更详细地了解它。还记得吧，文件对象有一个`readlines`方法，它能一次性地把文件载入到行字符串的一个列表中：

```
>>>f = open('script1.py')
>>>lines = f.readlines()
>>>lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
```

这是有效的，因为结果中的行在末尾都包含了一个换行符号(`\n`)。对于很多程序来说，换行符号很讨厌，我们必须小心避免打印的时候留下双倍的空白，等等。如果我们可以一次性地去除这些换行符号，岂不是好事吗？

当我们开始考虑在一个序列中的每项上执行一个操作时，都可以考虑使用列表解析。例如，假设变量`lines`像前面交互模式中一样，如下的代码通过对列表中的每一行运行字符串`rstrip`方法，来移除右端的空白（一个`line[:-1]`分片也有效，但是，只有当我们能够确保所有的行都正确结束的时候，它才有效）：

```
>>>lines = [line.rstrip() for line in lines]
>>>lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

这会像计划的那样工作。由于列表解析像`for`循环语句一样是一个迭代环境，我们甚至不必提前打开文件。如果我们在表达式中打开它，列表解析将自动使用在本章前面所介绍的迭代协议。也就是说，它将会调用文件的`next`方法，每次从文件读取一行。再次，我们得到了想要的内容，即一行的`rstrip`结果，对于文件中的每一行：

```
>>>lines = [line.rstrip() for line in open('script1.py')]
>>>lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

这个表达式做了很多隐式的工作，但是我们将在此揭秘其大多数工作——Python扫描文件并自动构建了操作结果的一个列表。这也是编写这一操作的一种高效率的方式：因为大多数工作在Python解释器内部完成，这可能比等价的语句要快很多。再次，特别是对于较大的文件，列表解析的速度优势可能很显著。

除了其高效性，列表解析的表现力也很强。在我们的例子中，我们可以在迭代时在一个文件的行上运行任何的字符串操作。下面是与我们前面遇到的文件迭代器大写示例对等的列表解析，还有几个其他的示例（这些例子中的第二个中的方法链是有效的，因为字符串方法返回一个新的字符串，可以对该字符串应用其他的字符串方法）：

```
>>>[line.upper() for line in open('script1.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']
```

```
>>>[line.rstrip().upper() for line in open('script1.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(2 ** 33)']

>>>[line.split() for line in open('script1.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(2', '**', '33)']]

>>>[line.replace(' ', '!') for line in open('script1.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(2!**!33)\n']

>>>[('sys' in line, line[0]) for line in open('script1.py')]
[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]
```

扩展的列表解析语法

实际上，列表解析可以有更高级的应用。作为一个特别有用的扩展，表达式中嵌套的for循环可以有一个相关的if子句，来过滤那些测试不为真的结果项。

例如，假设我们想要重复前面小节的文件扫描示例，但是，我们只需要收集以字母*p*开头的那些行（可能每一行的第一个字母是某种类型的动作代码）。向表达式中添加一条if过滤子句来实现：

```
>>>lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>>lines
['print(sys.path)', 'print(2 ** 33)']
```

这条if子句检查从文件读取的每一行，看它的第一个字符是否是*p*；如果不是，从结果列表中省略该行。这是一个相当大的表达式，但是，如果我们将它转换为简单的for循环语句等价形式的话，它很容易理解。通常，我们总是可以把一个列表解析转换为一条for语句，通过逐步附加并进一步缩进每个后续的部分：

```
>>>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语句等价形式也有效，但是，它占据了4行而不是一行，并且可能运行起来要慢很多。

如果我们需要的话，列表解析可以变得更复杂——例如，它们可能包含嵌套的循环，也可能被编写为一系列的for子句。实际上，它们的完整语法允许任意数目的for子句，每个子句有一个可选的相关的if子句（在第20章中，我们将更正式地介绍其语法）。

例如，下面的例子构建了一个x+y连接的列表，把一个字符串中的每个x和另一个字符串中的每个y连接起来。它有效地收集了两个字符串中的字符的排列：


```
>>>[x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

再次，理解这个表达式的一种方式是通过缩进其各个部分将它转换为语句的形式。下面是其等价形式，但可能会更慢一些，这是实现相同效果的一种替代方式：

```
>>>res = []
>>>for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>>res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

然而，除了这一复杂的层级，列表解析表达式往往可以变为更紧凑的形式。通常，它们会缩进以简化迭代的类型；对于更多的相关工作，一条简单的for语句结构可能更容易理解，并且将来也更容易修改。与编程中的通常情况一样，如果某些内容对你来说难以理解，它可能不是一个好主意。

我们将在第20章学习函数式编程工具的时候再次回顾列表解析；我们将会看到，当列表解析要对语句进行循环的时候，它们就是和函数相关联的。

其他迭代环境

在本书后面，我们将看到用户定义的类也可以实现迭代协议。因此，有时候知道哪些内置工具使用了该协议是很重要的——实现了迭代协议的任何工具，都能够在提供了该工具的任何内置类型或用户定义的类上自动地工作。

到目前为止，我们已经在for循环语句的背景下介绍了迭代，因为本书的这一部分内容关注于语句。然而，别忘了，在对象中从左到右扫描的每种工具都使用了迭代协议。这包括我们已经介绍过的for循环：

```
>>>for line in open('script1.py'):
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)                                     # Use file iterators
```

然而，列表解析、in成员关系测试、map内置函数以及像sorted和zip调用这样的内置函数也都使用了迭代协议。当应用于一个文件时，所有这些使用文件对象的迭代器都自动地按行扫描：

```
>>>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'))          # map is an iterable in 3.0
<map object at 0x02660710>

>>>list( 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

```

我们在上一章介绍过这里所用到的map调用，它是一个内置函数，它把一个函数调用应用于传入的可迭代对象中的每一项。map类似于列表解析，但是它更有局限性，因为它需要一个函数而不是一个任意的表达式。在Python 3.0中，它还返回一个可迭代的对象自身，因此，我们必须将它包含到一个list调用中以迫使其一次性给出所有的值，关于这一修改的更多介绍，参见本章随后的内容。由于map像列表解析一样，与循环和函数都相关，我们将在第19章和第20章中再次介绍它们。

Python还包含了各种处理迭代的其它内置函数：sorted排序可迭代对象中的各项，zip组合可迭代对象中的各项，enumerate根据相对位置来配对可迭代对象中的项，filter选择一个函数为真的项，reduce针对可迭代对象中的成对的项运行一个函数。所有这些都接受一个可迭代的对象，并且在Python 3.0中，zip、enumerate和filter也像map一样返回一个可迭代对象。它们实际运行文件的迭代器会自动地按行扫描，如下所示：

```

>>>sorted(open('script1.py'))
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']

>>>list(zip(open('script1.py'), open('script1.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'), ('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(2 ** 33)\n')]

>>>list(enumerate(open('script1.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'), (3, 'print(2 ** 33)\n')]

>>>list(filter(bool, open('script1.py')))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>>import functools, operator
>>>functools.reduce(operator.add, open('script1.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(2 ** 33)\n'

```

所有这些都是迭代工具，但它们有独特的作用。我们在上一章见过zip和enumerate，在第19章讨论函数的时候将会介绍filter和reduce，因此，这里暂不详细介绍。

我们在第4章初次见到了在这里所用到的sorted函数，并且，我们在第8章将其用于字

典。`sorted`是应用了迭代协议的一个内置函数，它就像是最初的列表`sort`方法，但是它返回一个新的排序的列表作为结果并且可以在任何可迭代对象上运行。注意，和`map`及其他的函数不同，`sorted`在Python 3.0中返回一个真正的列表而不是一个可迭代对象。

其他的内置函数也支持可迭代协议（但坦率地讲，很难用在和文件相关的有趣示例中）。例如，`sum`调用计算任何可迭代对象中的总数，如果一个可迭代对象中任何的或所有的项为真的时候，`any`和`all`内置函数分别返回`True`；`max`和`min`分别返回一个可迭代对象中最大和最小的项。和`reduce`一样，如下示例中的所有工具接受任何可迭代对象作为一个参数，并且使用迭代协议来扫描它，但返回单个的结果：

```
>>>sum([3, 2, 4, 1, 5, 0])           # sum expects numbers only
15
>>>any(['spam', '', 'ni'])
True
>>>all(['spam', '', 'ni'])
False
>>>max([3, 2, 5, 1, 4])
5
>>>min([3, 2, 5, 1, 4])
1
```

严格地讲，`max`和`min`函数也可以应用于文件——它们自动使用迭代协议来扫描文件，并且分别选择具有最高的和最低的字符串值的行（然而，我们把有效的用例留给你自己去想象）。

```
>>>max(open('script1.py'))           # Line with max/min string value
'x = 2\n'
>>>min(open('script1.py'))
'import sys\n'
```

有趣的是，在当今的Python中，迭代协议甚至比我們目前所能展示的示例要更为普遍——Python的内置工具集中从左到右地扫描一个对象的每项工具，都定义为在主体对象上使用了迭代协议。这甚至包含了更高级的工具，例如`list`和`tuple`内置函数（它们从可迭代对象构建了一个新的对象），字符串`join`方法（它将一个子字符串放置到一个可迭代对象中包含的字符串之间），甚至包括序列赋值。总之，所有这些都将在一个打开的文件上工作并且自动一次读取一行：

```
>>>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')

>>>a, *b = open('script1.py')           # 3.0 extended form
>>>a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n'])
```

早先我们也见到过内置的dict调用接受一个可迭代的zip结果。为此，我们来看看set调用，以及Python 3.0中的新的集合解析和字典解析表达式，这些我们在第4章、第5章和第8章见到过：

```
>>>set(open('script1.py'))
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>>{line for line in open('script1.py')}
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>>{ix: line for ix, line in enumerate(open('script1.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(2 ** 33)\n'}
```

实际上，集合解析和字典解析都支持我们在本章前面介绍的列表解析的扩展语法，包括if测试：

```
>>>{line for line in open('script1.py') if line[0] == 'p'}
{'print(sys.path)\n', 'print(2 ** 33)\n'}

>>>{ix: line for (ix, line) in enumerate(open('script1.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(2 ** 33)\n'}
```

和列表解析一样，这些都逐行扫描文件并且挑选以字母“p”开始的行。它们最终也恰好构建了集合和字典，但是，我们通过文件迭代和解析语法使得很多工作自动完成。

还有最后一个值得介绍的迭代环境，尽管现在介绍有点超前。在第18章中，我们将学习在函数调用中用到的一种特殊的*arg形式，它会把一个集合的值解包为单个的参数。现在我们可以预计，它也会接受任何可迭代对象，包括文件（参见第18章了解该调用语法的更多细节）：

```
>>>def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>>f(1, 2, 3, 4)
1&2&3&4
>>>f(*[1, 2, 3, 4])           # Unpacks into arguments
1&2&3&4

>>>f(*open('script1.py'))     # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(2 ** 33)
```

实际上，由于调用中的参数解包语法接受可迭代对象，也可能使用zip内置函数来把zip过的元组unzip，只要对任何另一个zip调用使用之前的或嵌套的zip结果参数（警告：如果你准备在不久的任何时候运行更沉重的机制，你可能不应该阅读如下的示例）：

```
>>>X = (1, 2)
>>>Y = (3, 4)
>>>
>>>list(zip(X, Y))                # Zip tuples: returns an iterable
[(1, 3), (2, 4)]
>>>
>>>A, B = zip(*zip(X, Y))        # Unzip a zip!
>>>A
(1, 2)
>>>B
(3, 4)
```

Python中还有其他的工具，如内置函数range和字典视图对象，它们返回可迭代对象而不是处理它们。要了解这些工具是如何在Python 3.0中吸收到迭代协议中的，我们需要继续看下一节。

Python 3.0中的新的可迭代对象

Python 3.0中的一个基本的改变是，它比Python 2.X更强调迭代。除了与文件和字典这样的内置类型相关的迭代，字典方法keys、values和items都在Python 3.0中返回可迭代对象，就像内置函数range、map、zip和filter所做的那样。正如前一小节所介绍的，这些函数中的最后三个都返回可迭代对象并处理它们。所有这些工具在Python 3.0中都根据请求产生结果，而不是像它们在Python 2.6中那样构建结果列表。

尽管这样会节约内存空间，它可能在某种程度上会影响到我们的编码方式。目前为止，在本书中的各种地方，我们已经把各种函数和方法调用结果都包含到一个list(...)调用中，从而迫使它们一次产生其所有的结果：

```
>>>zip('abc', 'xyz')              # An iterable in Python 3.0 (a list in 2.6)
<zip object at 0x02E66710>

>>>list(zip('abc', 'xyz'))        # Force list of results in 3.0 to display
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

这在Python 2.6中不是必需的，因为像zip这样的函数返回结果的列表。在Python 3.0中，它们返回可迭代的对象，根据需要来产生结果。这意味着要在交互提示模式下（并且可能在某些其他的环境中）显示结果需要额外的录入，这对较大的程序来说很有用，在计算很大的结果列表的时候，像这样的延迟计算会节约内存并避免暂停。让我们快速地浏览Python 3.0可迭代对象的应用。

range迭代器

在上一章中，我们学习过range内置函数的基本行为。在Python 3.0中，它返回一个迭代器，该迭代器根据需要产生范围中的数字，而不是在内存中构建一个结果列表。这取代了较早的Python 2.X xrange（参见后面的版本差异提示），如果需要一个范围列表的话，你必须使用list(range(...))来强制一个真正的范围列表（例如，显示结果）：

```
C:\misc> c:\python30\python
>>>R = range(10)                                # range returns an iterator, not a list
>>>R
range(0, 10)

>>>I = iter(R)                                    # Make an iterator from the range
>>>next(I)                                         # Advance to next result
0                                                  # What happens in for loops, comprehensions, etc.
>>>next(I)
1
>>>next(I)
2

>>>list(range(10))                                # To force a list if required
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

和这个调用在Python 2.X中返回的列表不同，Python 3.0中的range对象只支持迭代、索引以及len函数。它们不支持任何其他的序列操作（如果你需要更多列表工具的话，使用list(...))：

```
>>>len(R)                                          # range also does len and indexing, but no others
10
>>>R[0]
0
>>>R[-1]
9

>>>next(I)                                        # Continue taking from iterator, where left off
3
>>>I.__next__()                                  # .next() becomes __next__(), but use new next()
4
```

注意： 版本差异提示：Python 2.X也有一个名为xrange的内置函数，它就像range一样，但是根据需要产生元素而不是一次性在内存中构建一个结果列表。由于这完全就是新的基于迭代的range在Python 3.0中所做的事情，xrange在Python 3.0中不再可用——它已经被取代了。然而，在Python 2.X的代码中我们仍将看到它，特别是由于range构建了结果列表并且因此不像在其内存用法中那么高效。正如上一章的边栏中提到，由于类似的原因，Python 2.X中用来最小化内存使用的file.xreadlines()方法已经在Python 3.0中取消了，而是倾向于使用文件迭代。

map、zip和filter迭代器

和range类似，map、zip以及filter内置函数在Python 3.0中也转变成迭代器以节约内存空间，而不再在内存中一次性生成一个结果列表。所有这3个函数不仅像是在Python 2.X一样处理可迭代对象，而且在Python 3.0中返回可迭代结果。和range不同，它们都是自己的迭代器——在遍历其结果一次之后，它们就用尽了。换句话说，不能在它们的结果上拥有在那些结果中保持不同位置的多个迭代器。

在上一章中我们见过一个map内置函数的例子。和其他迭代器一样，如果确实需要一个列表的话，可以用list(...)来强制一个列表，但是，对于较大的结果集来说，默认的行为可以节省不少内存空间：

```
>>>M = map(abs, (-1, 0, 1))           # map returns an iterator, not a list
>>>M
<map object at 0x0276B890>
>>>next(M)                             # Use iterator manually: exhausts results
1                                       # These do not support len() or indexing
>>>next(M)
0
>>>next(M)
1
>>>next(M)
StopIteration

>>>for x in M: print(x)                 # map iterator is now empty: one pass only
...

>>>M = map(abs, (-1, 0, 1))             # Make a new iterator to scan again
>>>for x in M: print(x)                 # Iteration contexts auto call next()
...
1
0
1
>>>list(map(abs, (-1, 0, 1)))           # Can force a real list if needed
[1, 0, 1]
```

上一章所介绍的zip内置函数，返回以同样方式工作的迭代器：

```
>>>Z = zip((1, 2, 3), (10, 20, 30))    # zip is the same: a one-pass iterator
>>>Z
<zip object at 0x02770EE0>

>>>list(Z)
[(1, 10), (2, 20), (3, 30)]

>>>for pair in Z: print(pair)           # Exhausted after one pass
...

>>>Z = zip((1, 2, 3), (10, 20, 30))
>>>for pair in Z: print(pair)           # Iterator used automatically or manually
...
```



```

(1, 10)
(2, 20)
(3, 30)

>>>Z = zip((1, 2, 3), (10, 20, 30))
>>>next(Z)
(1, 10)
>>>next(Z)
(2, 20)

```

我们将在本书下一部分中学习的`filter`内置函数，也是类似的。对于传入的函数返回`True`的可迭代对象中的每一项，它都会返回该项（正如我们已经学习过的，Python中的`True`包括非空的对象）：

```

>>>filter(bool, ['spam', '', 'ni'])
<filter object at 0x0269C6D0>
>>>list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']

```

和本小节讨论的大多数工具一样，`filter`可以接受一个可迭代对象并进行处理，返回一个可迭代对象并在Python 3.0中产生结果。

多个迭代器 VS 单个迭代器

看看`range`对象与本小节介绍的内置函数有何不同，这很有趣，它支持`len`和索引，它不是自己的迭代器（手动迭代时，我们使用`iter`产生一个迭代器），并且，它支持在其结果上的多个迭代器，这些迭代器会记住它们各自的位置：

```

>>>R = range(3)                # range allows multiple iterators
>>>next(R)
TypeError: range object is not an iterator

>>>I1 = iter(R)
>>>next(I1)
0
>>>next(I1)
1
>>>I2 = iter(R)                # Two iterators on one range
>>>next(I2)
0
>>>next(I1)                    # I1 is at a different spot than I2
2

```

相反，`zip`、`map`和`filter`不支持相同结果上的多个活跃迭代器：

```

>>>Z = zip((1, 2, 3), (10, 11, 12))
>>>I1 = iter(Z)
>>>I2 = iter(Z)                # Two iterators on one zip
>>>next(I1)

```

```

(1, 10)
>>>next(I1)
(2, 11)
>>>next(I2)           # I2 is at same spot as I1!
(3, 12)

>>>M = map(abs, (-1, 0, 1))   # Ditto for map (and filter)
>>>I1 = iter(M); I2 = iter(M)
>>>print(next(I1), next(I1), next(I1))
1 0 1
>>>next(I2)
StopIteration

>>>R = range(3)              # But range allows many iterators
>>>I1, I2 = iter(R), iter(R)
>>>[next(I1), next(I1), next(I1)]
[0 1 2]
>>>next(I2)
0

```

当我们在本书随后（第29章）使用类来编写自己的可迭代对象的时候，将会看到通常通过针对`iter`调用返回一个新的对象，来支持多个迭代器；单个的迭代器一般意味着一个对象返回其自身。在第20章中，我们还将看到，在这方面，生成器函数和表达式的行为就像`map`和`zip`一样支持单个的活跃迭代器，而不是像`range`一样。在第20章中，我们将会看到一些暗示：位于循环中的一个单个的迭代器试图多次扫描。

字典视图迭代器

正如我们在第8章中简单了解到的，在Python 3.0中，字典的`keys`、`values`和`items`方法返回可迭代的视图对象，它们一次产生一个结果项，而不是在内存中一次产生全部结果列表。视图项保持和字典中的那些项相同的物理顺序，并且反映对底层的字典做出的修改。既然已经对迭代器了解甚多，接下来我将继续介绍其他内容：

```

>>>D = dict(a=1, b=2, c=3)
>>>D
{'a': 1, 'c': 3, 'b': 2}

>>>K = D.keys()           # A view object in 3.0, not a list
>>>K
<dict_keys object at 0x026D83C0>

>>>next(K)                # Views are not iterators themselves
TypeError: dict_keys object is not an iterator

>>>I = iter(K)             # Views have an iterator,
>>>next(I)                # which can be used manually
'a'                       # but does not support len(), index
>>>next(I)
'c'

>>>for k in D.keys(): print(k, end=' ')   # All iteration contexts use auto

```

```
...
a c b
```

和所有的迭代器一样，我们总可以通过把一个Python 3.0字典视图传递到`list`内置函数中，从而强制构建一个真正的列表。然而，这通常不是必须的，除了交互地显示结果或者应用索引这样的列表操作：

```
>>>K = D.keys()
>>>list(K)                                # Can still force a real list if needed
['a', 'c', 'b']

>>>V = D.values()                          # Ditto for values() and items() views
>>>V
<dict_values object at 0x026D8260>
>>>list(V)
[1, 3, 2]

>>>list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>>for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 c 3 b 2
```

此外，Python 3.0字典仍然有自己的迭代器，它返回连续的键。因此，无需直接在此环境中调用`keys`：

```
>>>D                                       # Dictionaries still have own iterator
{'a': 1, 'c': 3, 'b': 2}                  # Returns next key on each iteration
>>>I = iter(D)
>>>next(I)
'a'
>>>next(I)
'c'

>>>for key in D: print(key, end=' ')      # Still no need to call keys() to iterate
...                                       # But keys is an iterator in 3.0 too!
a c b
```

最后，再次提醒，由于`keys`不再返回一个列表，按照排序的键来扫描一个字典的传统编码模式在Python 3.0中不再有效。相反，首先用一个`list`调用来转换`keys`视图，或者在一个键视图或字典自身上使用`sorted`调用，如下所示：

```
>>>D
{'a': 1, 'c': 3, 'b': 2}
>>>for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3

>>>D
{'a': 1, 'c': 3, 'b': 2}
>>>for k in sorted(D): print(k, D[k], end=' ') # Best practice key sorting
```

```
...  
a 1 b 2 c 3
```

其他迭代器主题

我们还将第20章学习列表解析和迭代器的更多内容，在第29章学习类的时候，我们还将再次遇到它们。在后面，我们将会看到：

- 使用yield语句，用户定义的函数可以转换为可迭代的生成器函数。
- 当编写在圆括号中的时候，列表解析转变为可迭代的生成器表达式。
- 用户定义的类通过__iter__或__getitem__运算符重载变得可迭代。

特别地，使用类定义的用户定义的迭代器，允许在我们这里所遇到的任何迭代环境中使用任意对象和操作。

本章小结

在本章中，我们介绍了Python中与循环相关的概念。我们对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。此外，我们还看到了其他内置迭代工具的使用，并且学习了Python 3.0中关于迭代的最新变化。

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，先做一下这里的习题。

本章习题

1. for循环和迭代器之间有什么关系？
2. for循环和列表解析直接有什么关系？
3. 举出Python中的4种迭代环境。
4. 如今从一个文本文件逐行读取行的最好的方法是什么？

习题解答

1. `for`循环会使用迭代协议来遍历迭代的对象中的每一个项。`for`循环会在每次迭代中调用该对象的`__next__`方法（由`next`内置函数运行），而且会捕捉`StopIteration`异常，从而决定何时停止循环。支持这种模式的任何对象，都可以用于`for`循环以及其他迭代环境中。
2. 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。
3. 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。
4. 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。

本书的这一部分谈的是用于编写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有两个原因，一是它涉及了Python的语法模型，二是它是那些努力想了解Python工具集的读者的资源。就后面这个原因而言，我会在这里展开第4章第一次给出的对文档的介绍。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。本节提供了一些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，一旦你编写代码的能力达到编写本书这一部分的例子和练习题的水平时，这就变成是重要的知识了。

如表15-1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。

表15-1: 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: <code>__doc__</code>	附加在对象上的文件中的文档
PyDoc: <code>help</code> 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`（这是Python 3.0中的结果，在Python 2.6中可能略有不同）：

```
>>>import sys
>>>dir(sys)
['_displayhook_', '__doc__', '__excepthook__', '__name__', '__package__',
'_stderr_', '_stdin_', '_stdout_', '_clear_type_cache', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding',
...more names omitted...]
```


在这里只显示诸多变量名中的一些而已。你可在机器上运行这些语句来查看完整的清单。

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__', '__contains__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>>dir('')
['__add__', '__class__', '__contains__', ...more...
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', '
maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
...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。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('')
True
>>>dir(list) == dir([])
True
```

Same result as prior example

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，会启用其构造函数，从而产生了该类型的实例。我会在第六部分讨论类时，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（#注释在其之前也没问题）。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。

```
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
    """
    function documentation
    can we have your liver then?
    """
    return x ** 2          # square

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串，可以通过路径访问类：`module.class.method.__doc__`（参考第28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，坦率地讲，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的为代码编写文档，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 functions that interact strongly with the interpreter.

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 reference count of object. The count returned is generally
one higher than you might expect, because it includes the (temporary)
...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. A floating
point argument will be truncated towards zero (this does not include a
...more text omitted...
```

```
>>>print(map.__doc__)
map(func, *iterables) --> map object
```

```
Make an iterator that computes the function using arguments from
each of the iterables. Stops when the shortest iterable is exhausted.
```

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，并将其格式化各种类型的排列友好的报表。开源领域还有很多其他的工具可以用来提取和格式化文档字符串（包括支持结构化文本的工具，在Web上进行搜索以获得信息），但Python在其标准库中附带了PyDoc。

有很多种方式可以启动PyDoc，包括命令行脚本选项（更多细节请参考Python库手册）。也许两种最主要的PyDoc接口是内置的help函数和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 in module sys:

getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object. The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    ...more omitted...
```

注意：调用help时，不是一定要导入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)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...more omitted...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it in builtins.
        ...more omitted...

DATA
    __stderr__ = <io.TextIOWrapper object at 0x0236E950>
    __stdin__ = <io.TextIOWrapper object at 0x02366550>
    __stdout__ = <io.TextIOWrapper object at 0x02366E30>
    ...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 builtins:

class dict(object)
| dict() -> new empty dictionary.
| dict(mapping) -> new dictionary initialized from a mapping object's
| ...more omitted...

>>>help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    ...more omitted...

>>>help(ord)
Help on built-in function ord in module builtins:

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。同样的，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。

```
>>>import docstrings
>>>help(docstrings.square)
Help on function square in module docstrings:

square(x)
    function documentation
    can we have your liver then?

>>>help(docstrings.Employee)
Help on class Employee in module docstrings:

class Employee(builtins.object)
    | class documentation
    |
    | Data descriptors defined here:
    ...more omitted...

>>>help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\misc\docstrings.py

DESCRIPTION
    Module documentation
    Words Go Here

CLASSES
    builtins.object
        Employee

    class Employee(builtins.object)
        | class documentation
        |
        | Data descriptors defined here:
        ...more omitted...

FUNCTIONS
    square(x)
        function documentation
        can we have your liver then?

DATA
    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，想要更宏观的显示的话，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PythonTools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，然后按下回车键。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。



图15-1: Pydoc顶层搜索引擎GUI：输入你想找的模块名称，再按下回车键，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15-2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15-3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 and 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

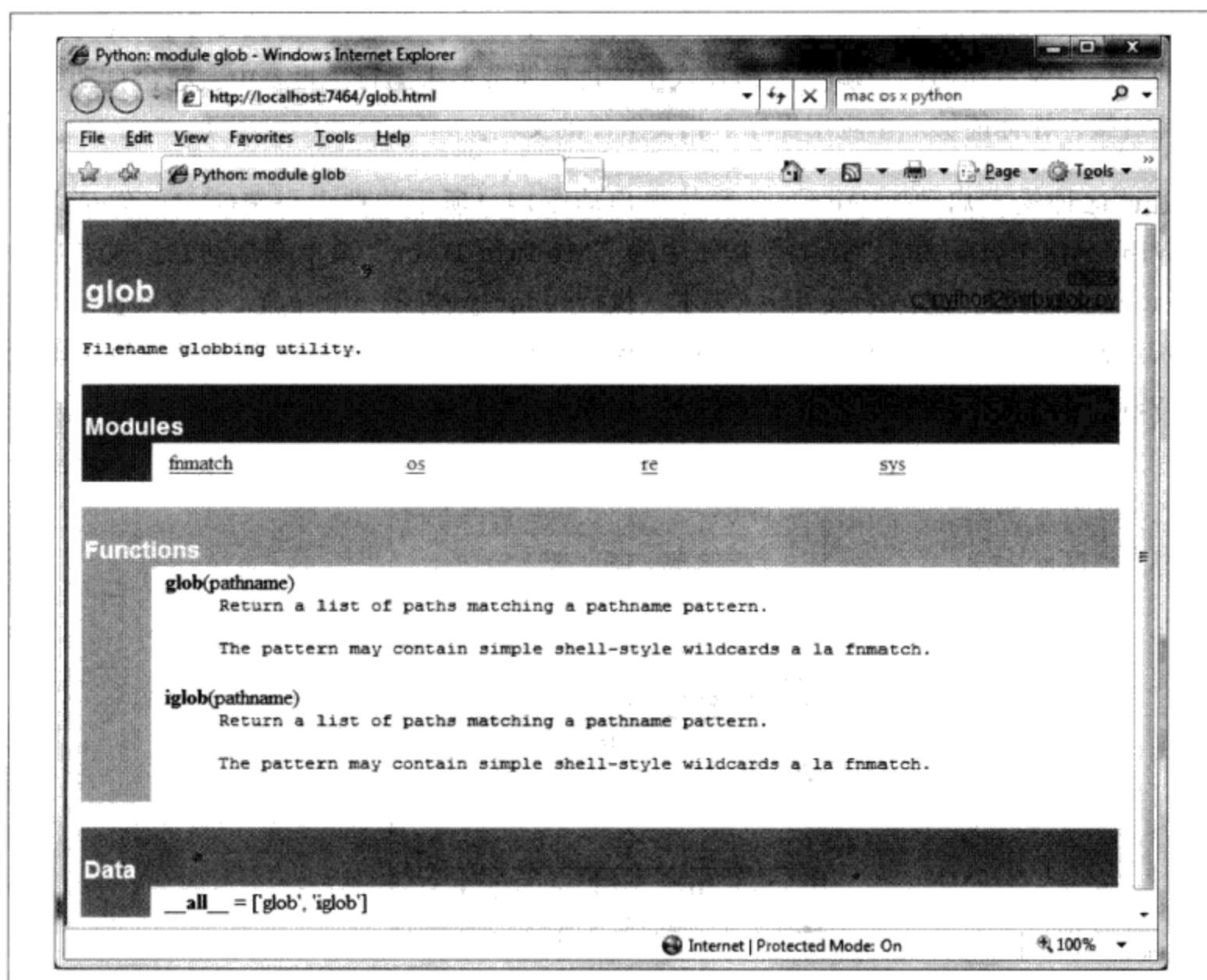


图15-2：当你在图15-1 GUI中找到一个模块并按下“go to selected”时，模块的文档会以HTML呈现，并显示在网页浏览器窗口中，就像这里所展示的

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。

现在我来介绍一下PyDoc技巧。如果你在图15-1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。

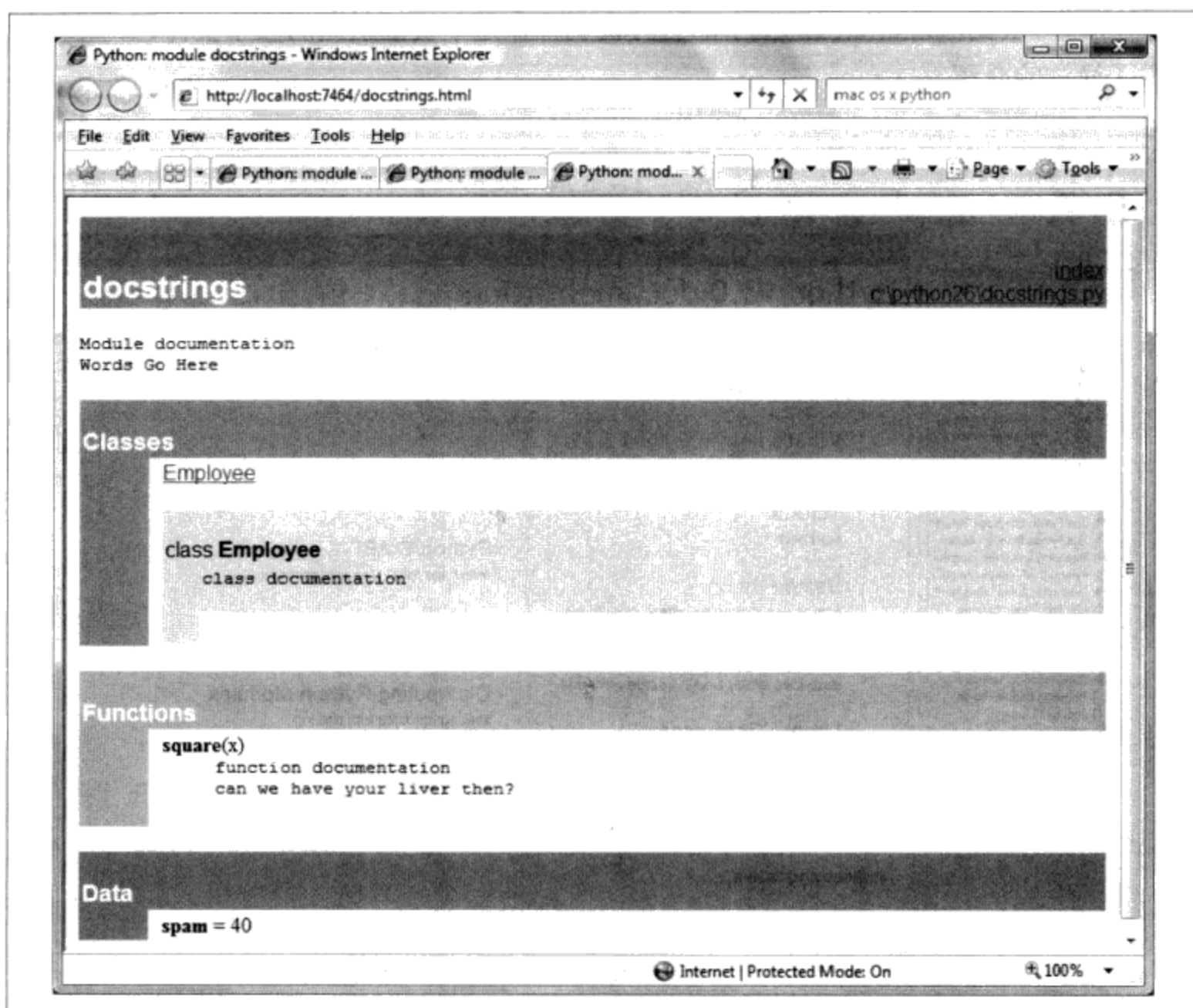


图15-3: PyDoc可作为显示内置和用户定义的模块的文档页。这里的网页是用户定义模块的说明网页，显示了它从源代码中提取出的所有文档字符串（文档字符串）

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。

开启时，Windows格式的手册会显示像图15-4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。

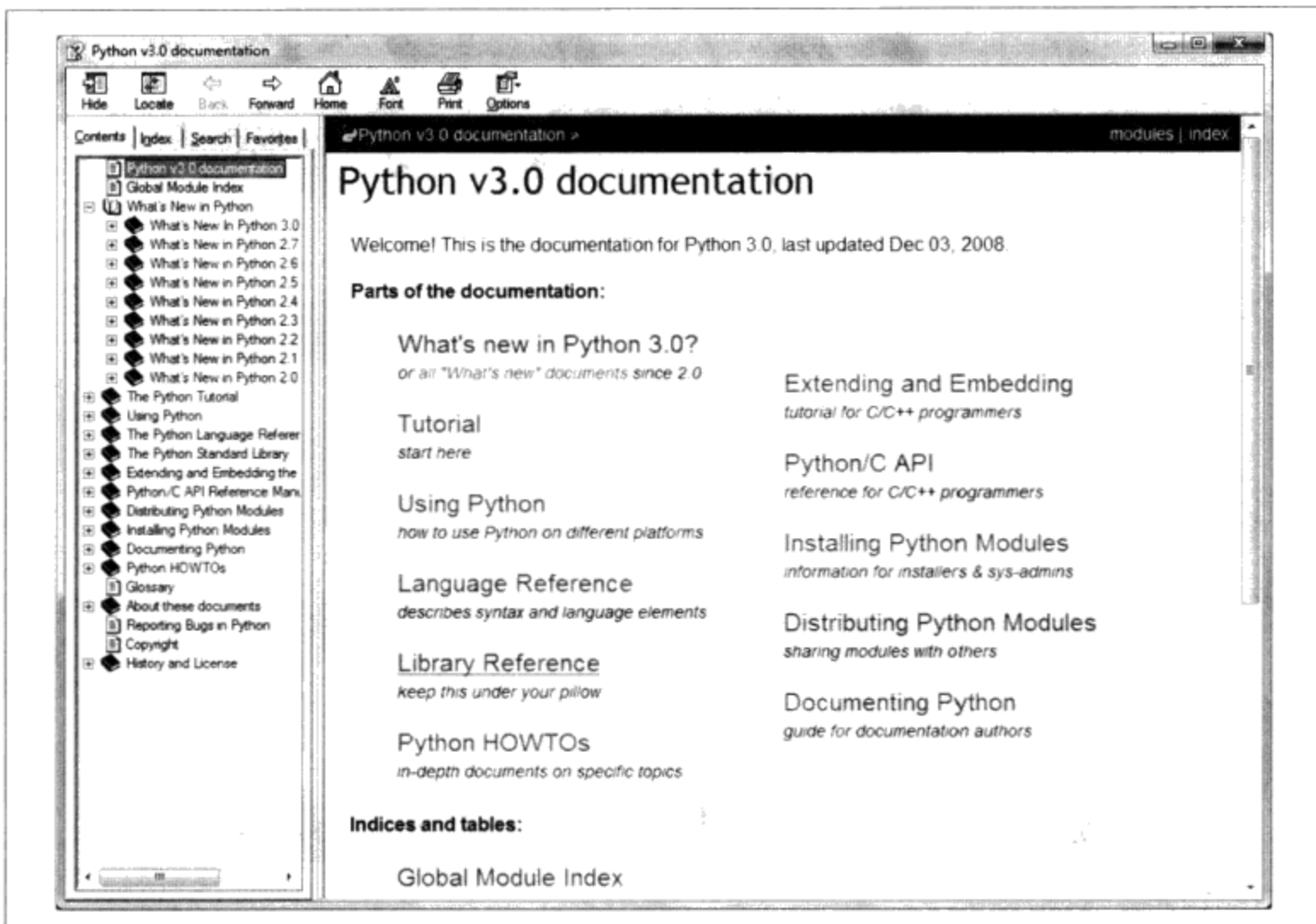


图15-4: Python的标准手册，可在www.python.org上在线阅读，从IDLE的“Help”菜单启动，以及从Windows的“开始”按钮菜单启动。这是Windows上可搜索的辅助文件，而且在线版本也有搜索引擎。其中，Library手册是大多数实际最常使用的工具之一

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。

已出版的书籍

你可以从大量Python参考书籍中做选择，以作为最终的资源。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3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，但是现在一些内容可能对你有帮助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3000多位学生多数也会），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章的内容。
- 不要期待在原处修改对象的函数会返回结果。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时，Python 2.X的代码中会有更复杂的例子，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。即便早期在Python 3.0中的时候，这也会失效，因为字典键是视图而不是列表。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 = list(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`]。在第四部分中，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，而第五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三部分的练习题。但在那之前，先来做本章习题。

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3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. 阅读本书之后，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，前言列出了一些我所推荐的进阶书籍，包括了参考书籍和应用程序开发的教程）。

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“第三部分 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14章）。

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 = False  
i = 0  
while not found and i < len(L):  
    if 2 ** X == L[i]:  
        found = True  
    else:  
        i = i+1
```



```
if found:
    print('at index', i)
else:
    print(X, 'not found')

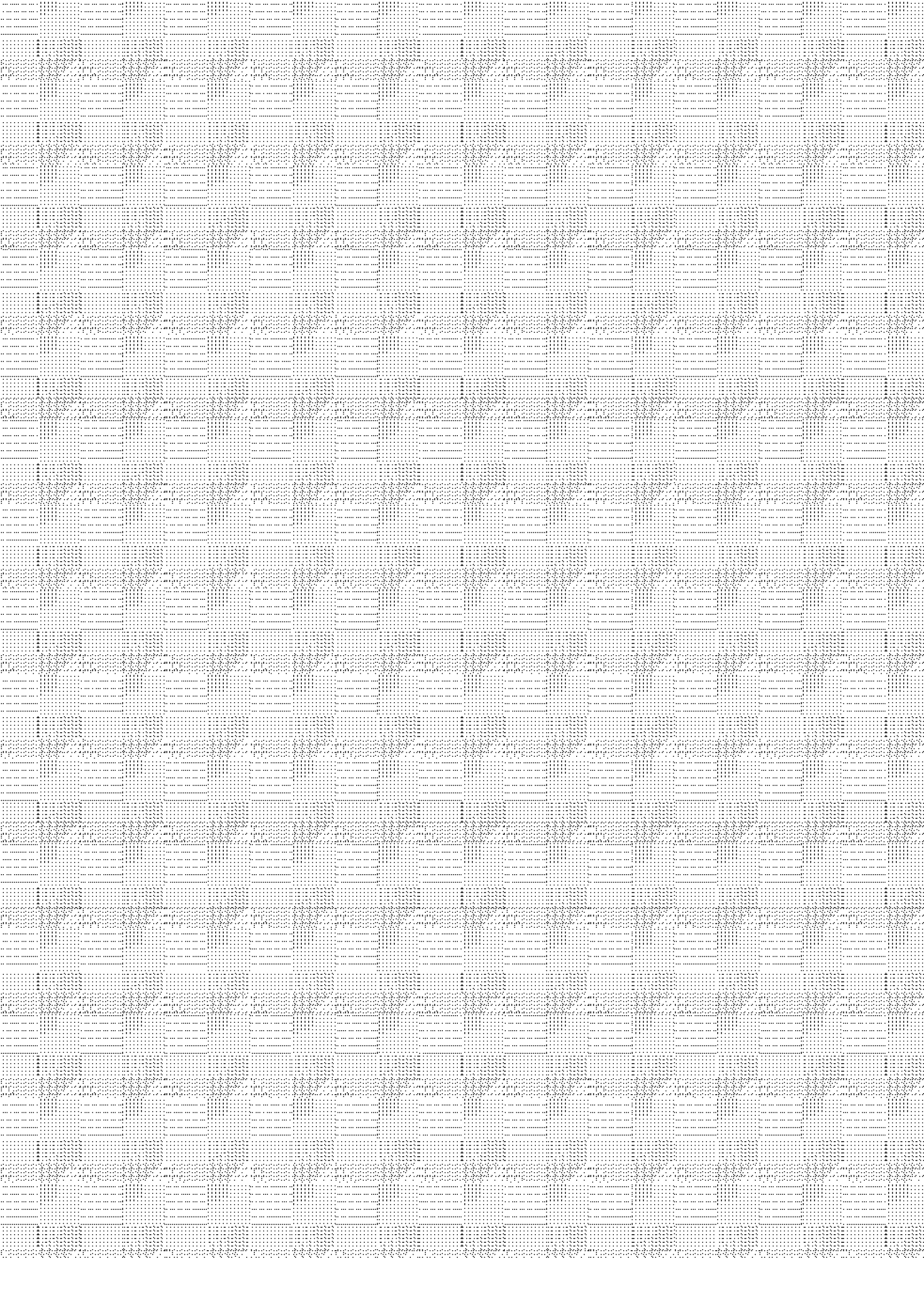
C:\book\tests>python power.py
at index 5
```

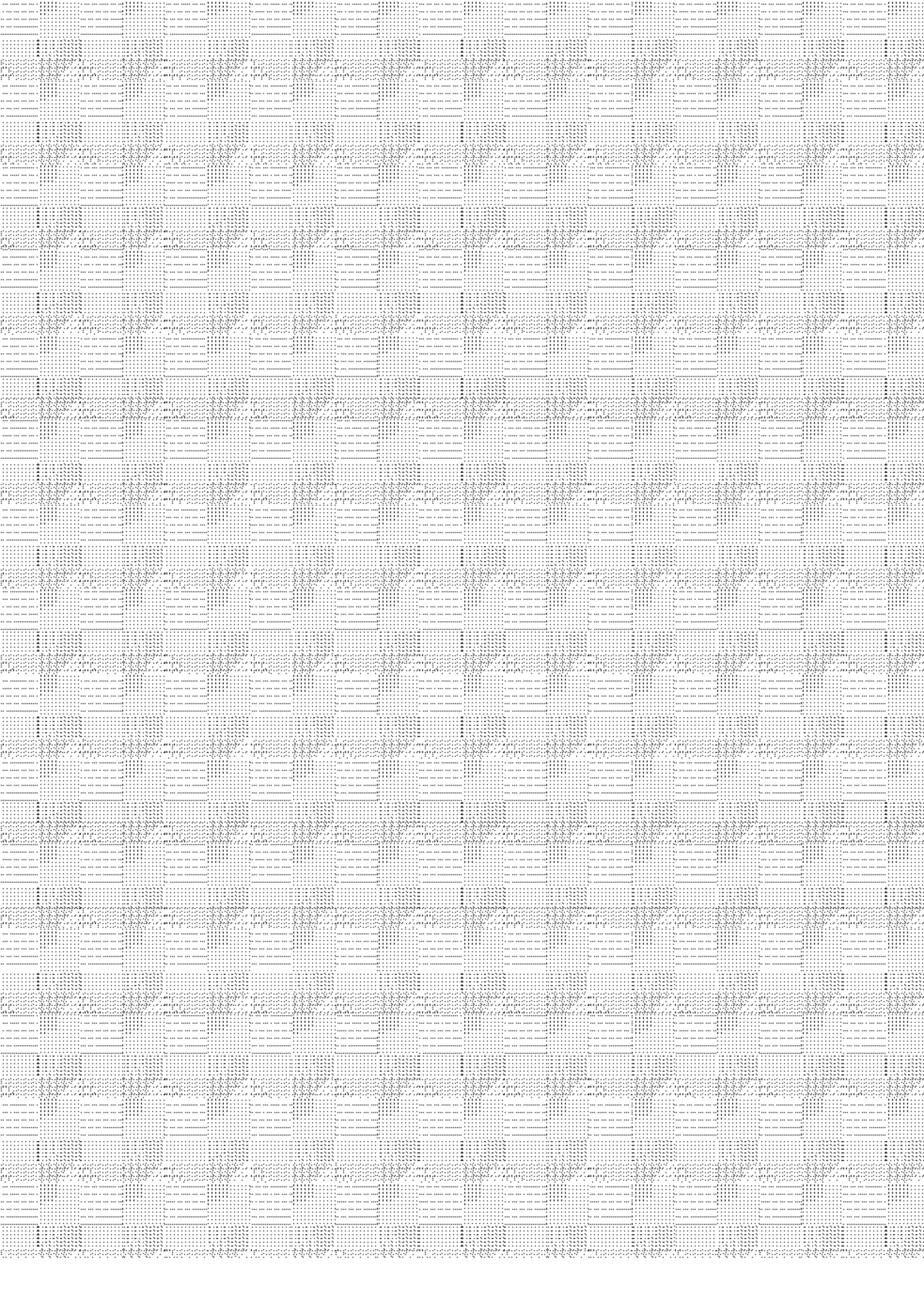
这个例子并没有遵循一般的Python代码编写的技巧。遵循这里所提到的步骤来改进它（就所有的转变而言，你可以在交互模式下输入代码，或者将其保存在脚本文件中从系统命令行来运行，使用文件会让这个练习更加容易）：

- 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？如何编写代码？
- f. 就像我们在练习题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19章正式引入lambda。





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，便于自己创建函数。

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可以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。

更具体地说，函数是在编程过程中剪剪贴贴的替代——我们不再有一个操作的代码的多个冗余副本，而是将代码包含到一个单独的函数中。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，使用它我们可以将复杂的系统分解为可管理的部件。表16-1总结了这一部分中我们将会学习到的与函数相关的主要语句和表达式。

表 16-1：函数相关的语句和表达式

语句	例子
Calls	<code>myfunc("spam", "eggs", meat=ham)</code>
def,	<code>def adder(a, b=1, *c):</code>
return	<code>return a+b+c[0]</code>

表 16-1：函数相关的语句和表达式（续）

语句	例子
global	<pre>def changer(): global x; x = 'new'</pre>
nonlocal	<pre>def changer(): nonlocal x; x = 'new'</pre>
yield	<pre>def squares(x): for i in range(x): yield i ** 2</pre>
lambda	<pre>Funcs = [lambda x: x**2, lambda x: x*3]</pre>

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，有时称为子例程或过程。简而言之，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，去做一份比萨，开始需要混合面粉，将面粉搅拌均匀，增加顶部原料和烤等。如果你是在编写一个制作比萨的机器人的程序，函数将会将整个“做比萨”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6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。
- **lambda创建一个对象但将其作为结果返回。**也可以用`lambda`表达式创建函数，这一功能允许我们把函数定义内联到语法上一条`def`语句不能工作的地方（这是一个更加高级的概念，我们推迟到第19章介绍）。
- **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。
- **yield向调用者发回一个结果对象，但是记住它离开的地方。**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。
- **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。通常情况下，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。

- **nonlocal**声明了将要赋值的一个封闭的函数变量。类似的，Python 3.0中添加的**nonlocal**语句允许一个函数来赋值一条语法封闭的def语句的作用域中已有的名称。这就允许封闭的函数作为保留状态的一个地方——当一个函数调用的时候，信息被记住了——而不必使用共享的全局名称。
- **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的模式中，调用者以及函数通过引用共享对象，但是不需要别名。改变函数中的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。
- **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。

如果说前面有些内容介绍得不够深入的话，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——也就是每当调用函数时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条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20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，它创建一个新的函数对象并将其赋值给一个变量名。（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）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，并在模块导入时运行，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。

```
if test:
    def func():
        ...
else:
    def func():
        ...
...
func()
# Define func this way
# Or else this way
# Call the version selected and built
```

它在运行时简单地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
othername()
# Assign function object
# 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。实际上，除了调用以外，函数允许任意的属性附加到记录信息以供随后使用：

```
def func(): ...           # Create function object
func()                   # Call object
func.attr = value        # Attach attributes
```

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的：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`。典型的情况是，这样一个语句编写在一个模块文件之中，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`赋值为2，`y`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
```

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）至关重要不同之处：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，也能够进一步地解释函数的基本概念。

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，一旦需要修改重复性的任务，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计算机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 = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

Start empty
Scan seq1
Common item?
Add to end

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，我们增加了一条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"
>>>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。

为了公平起见，我们的intersect函数相当慢（它执行嵌套循环），并不是真正的数学交集（结果中可能有重复的元素），并且也根本不必要（正如我们已经看到的，Python的集合数据类型提供了一个内置的交集操作）。实际上，这个函数可以用一个单独的列表解析表达式来替代，因为它展示了经典的循环收集器代码模式：

```
>>>[x for x in s1 if x in s2]
['S', 'A', 'M']
```

作为一个函数的基础示例，它完成了任务——这个单个的代码段可以应用于整个的对象类型范围，正如下一小节所述。

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14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

可能这个例子中最有趣的部分是其名称。它证明了，`intersect` 函数中的 `res` 变量在 Python 中叫做本地变量——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

注1：如果我们把用 `file.readlines()` 获取的文件内容相交，这段代码总是有效。然而，如果直接相交打开输入文件中的行，根据文件对象的 `in` 运算符和通用迭代的实现的不同，它有可能无法工作。在文件已经有一次读取到文件末尾的时候，文件通常必须重新查找（例如，用一个 `file.seek(0)` 或另一个 `open`）。正如我们在第29章中学习运算符重载时候将要看到的，类实现 `in` 运算符的时候，要么通过提供特定的 `__contains__` 方法，要么通过使用 `__iter__` 或较早的 `__getitem__` 方法来支持通用迭代协议；如果编码的话，类可以定义对其数据使用何种迭代方法。

`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17章。

本章小结

这一章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17章进行介绍。那么，下面让我们来做个简单测试吧。

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）。

作用域

第16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。我们将会看到，在代码中的何处给一个名字赋值，对于确定这个名字的含义很关键。我们还将看到，作用域的用法会对程序维护工作有着重要的影响，例如，过度地使用全局作用域通常是糟糕的事情。

Python作用域基础

既然现在你已经准备编写函数了，那么我们需要更正式地了解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。

在任何情况下，一个变量的作用域（它所使用地方）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。实际上，正如我们将在本章中学到的，变量可以在3个不同的地方分配，分别对应3种不同的作用域：

- 如果一个变量在def内赋值，它被定位在这个函数之内。
- 如果一个变量在一个嵌套的def中赋值，对于嵌套的函数来说，它是非本地的。
- 如果在def之外赋值，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。

作用域法则

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`）。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。当你在Python中听到“全局的”，你就应该想到“模块”。
- **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（一种叫做递归的高级技术），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。递归在处理不能提前预知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或非本地变量，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。如果需要给位于一个嵌套的def中的名称赋值，从Python 3.0开始可以通过在一条nonlocal语句中声明它来做到。
- **所有其他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。

这里还有一些细节需要注意。首先，记住以交互命令提示模式输入的代码也遵从这些规则。你可能还不知道，但是，交互模式运行的代码实际上真的输入到一个叫做__main__的内置模块中；这个模块就像一个模块文件一样工作，但是，结果随着输入而反馈。因此，交互模式也在一个模块中创建名称，并由此遵守常规的作用域规则：它们对于交互会话来说是全局的。我们将在本书的下一个部分学习有关模块的内容。

还要注意，一个函数内部的任何类型的赋值都会把一个名称划定为本地的。这包括=语句、import中的模块名称、def中的函数名称、函数参数名称等。如果在一个def中以任何方式赋值一个名称，它都将对于该函数成为本地的。

此外，注意原处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。通常，记住名称和对象之间的清楚的区分是有帮助的：修改一个对象并不是对一个名称赋值。

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。
- 全局声明和非本地声明将赋值的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，但是必须声明为非本地变量和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17-1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1}。

注1： 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初学者而言通常无关紧要，所以，我们会将其放到本章后面再谈。由于这一作用域在Python 3.0中通过nonlocal语句来解决，我建议查找规则现在最好叫做“LNGB”，但本书也要考虑到向后兼容的问题。

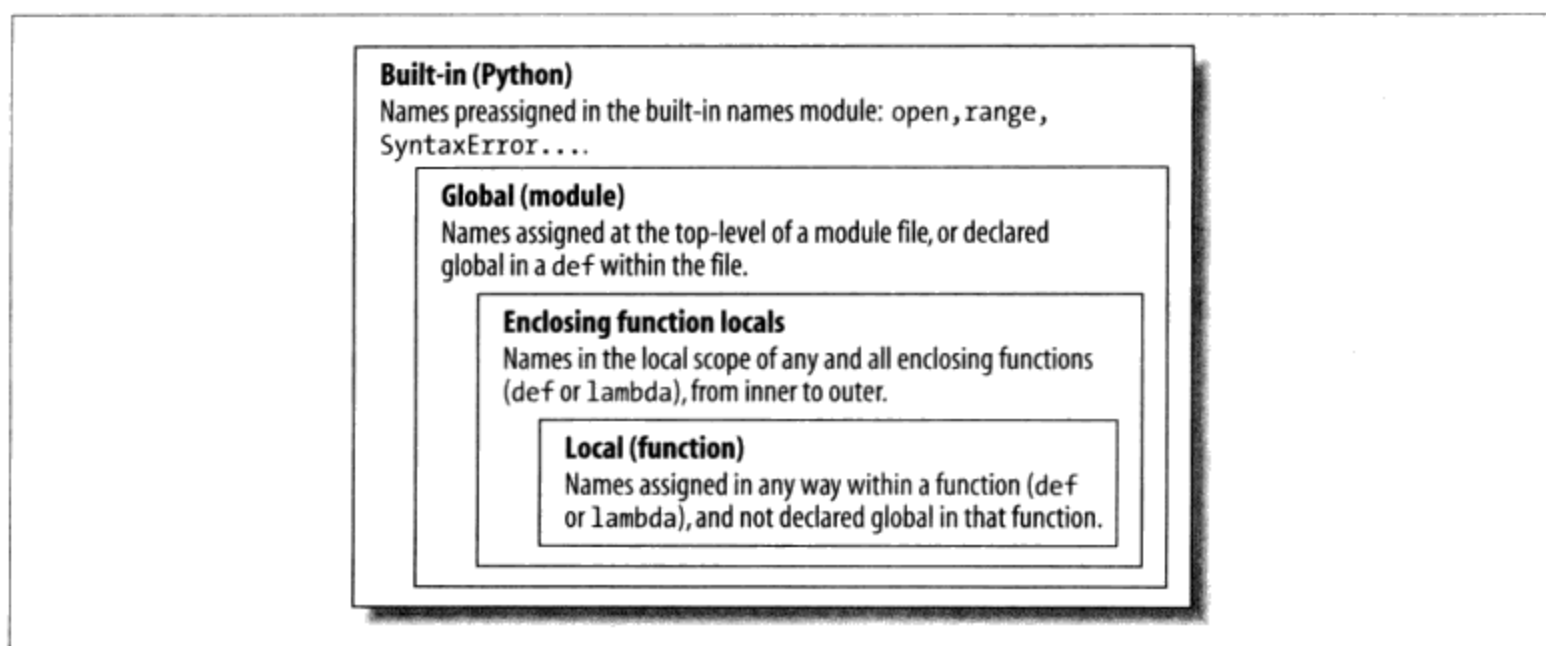


图17-1：LEGB作用域查找原则。当引用一个变量时，Python按以下顺序依次进行查找：从本地变量中，在任意上层函数的作用域，在全局作用域，最后在内置作用域中查找。第一个能够完成查找的就算成功。变量在代码中被赋值的位置通常就决定了它的作用域。在Python 3.0中，nonlocal声明也可以迫使名称映射到函数内部的作用域中，而不管是否对其赋值

此外，记住这些规则仅对简单的变量名有效（例如，spam）。在第五部分和第六部分中，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，并且有可能涉及所谓的“继承”的概念（将在第六部分讨论）。

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

我们已经简单地介绍了内置作用域，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。

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哪些变量名。在Python 3.0中：

```
>>>import builtins
>>>dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
...many more names omitted...
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', '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LEGB法则带来的好处，或者手动导入__builtin__模块。

```
>>>zip                                     # The normal way
<class 'zip'>
```

```
>>>import builtins                # The hard way
>>>builtins.zip
<class '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（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）。

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（或nonlocal）声明的话，是没有办法在函数内改变函数外部的变量的，正如下一小节所介绍的。

注意： 版本差异介绍：实际上，绕口令会变得更糟糕一些。这里所使用的Python 3.0 builtins模块，在Python 2.6中叫做__builtin__。并且只是为了有趣，在大多数全局作用域中，包括交互式会话中，都预先设置了名称__builtins__（带有s），来表示名为builtins的模块（即Python 2.6中的__builtin__）。

也就是说，在导入了builtins之后，在Python 3.0中，__builtins__ is builtins是True；并且在Python 2.6中__builtins__ is __builtin__是True。直接的效果是，我们可以直接运行dir(__builtins__)来查看内置作用域，而在Python 3.0和Python 2.6中都不用导入，但我们建议对于Python 3.0中的实际工作使用builtins。谁说讲清楚这些内容很容易呢？

在Python 2.6中违反通用性

还有在Python中可以做但不应该去做的另一件事——由于名称True和False在Python 2.6中是内置作用域中的变量而不是保留字，用诸如True = False的一条语句来重新为它们赋值就成为可能。不用担心，实际上，这么做不会破坏通用的逻辑一致性。这条语句只是在它所出现的单个的作用域中重新定义了单词True。所有其他的作用域仍然在内置作用域中查找其最初的定义。

更为有趣的是，在Python 2.6中，可以使用__builtin__.True = False，来在整个Python过程中把True重置为False。然而，这种类型的赋值在Python 3.0中已经取消了，因为True和False都看做是真正的保留字，就像None一样。然而，在Python 2.6中，它把IDLE置于一种特殊的莫名其妙的状态，它会重写设置用户代码的处理。

然而，这种技术可能有用，可以用来说明底层的命名空间模型，对于必须把open这样的内置函数修改为定制函数的工具编写者来说也会有用。此外，注意，PyChecker这样的第三方工具将会警告常见的编程错误，包括对内置名称的偶然性赋值（在PyChecker中，这叫做“阴影化”一个内置名称）。

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。

我们已经在前面讲过了全局变量名。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。

换句话说，global允许我们修改一个模块文件的顶层的一个def之外的名称。正如我们将在随后看到的，nonlocal语句几乎是相同的，但它应用于嵌套的def的本地作用域内的名称，而不是嵌套的模块中的名称。

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：

```
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
```

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中最直接保持状态信息的方法（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。^{注2}

到目前为止，在不熟悉编程的情况下，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。

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。本书下一部分将更深入地讨论第3章中介绍的模块文件。为了说明它们与作用域之间的关系，考虑下面这两个模块文件：

```
# first.py
X = 99                                     # This code doesn't know about second.py
```

注2：多线程与其他的程序并行地运行函数调用，并且得到Python的标准库模块_thread、threading和queue（在Python 2.6中分别是thread、threading和Queue）的支持。由于所有这些线程化函数都在同一进程中运行，全局作用域往往充当它们之间的共享内存。线程用于GUI中长时间运行的任务，以实现广泛地实现非阻塞的操作并利用CPU的能力。这也超出了本书的讨论范围，参见Python库手册和前言中列出的相关图书（例如O'Reilly的 *Programming Python*），可以了解更多细节。

```

# second.py
import first
print(first.X)
first.X = 88
# Okay: references a name in another file
# But changing it can be too subtle and implicit

```

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，我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从另一个模块中看到它内部的变量。这是关于模块的一个要点：通过在每个文件的基础上分隔变量，它们避免了跨文件的名称冲突。

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，都不一定知道有一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。

尽管这样的跨文件变量在Python中总是可能修改的，但它们通常比我们想要的更微妙。再者，这会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。这种隐含的跨文件依赖性，在最好的情况下会导致代码不灵活，最坏的情况下会引发bug。

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，会知道这是一个接入点，并且知道这将改变变量X。换句话说，它删除了令人惊讶的元素，在软件项目中过多地使用它们并非好事。虽然我们无法避免修改文件间的变量，但是通常的做法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

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，但是为了清晰地表达你的想法，这种方法要比直接使用`global`语句需要做更多的工作。

正如我们已经看到的，`global`允许我们修改一个函数之外的模块中的名称。还有一个类似的`nonlocal`，它也可以用来修改嵌套的函数中的名称，但是，要理解这有多大用处，我们需要首先概括地介绍嵌套函数。

作用域和嵌套函数

到现在为止，忽略了Python的作用域法则中的一部分（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。嵌套作用域有时也叫做静态嵌套作用域。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。

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：

- 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，从内到外查找；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。
- 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。另一方面，如果X在函数内声明为nonlocal，赋值会修改最近的嵌套函数的本地作用域中的名称X。

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，嵌套函数中的变量也许仅仅是引用，但它们需要nonlocal声明才能修改。

嵌套作用域举例

为了阐明上一小节的要点，让我们用一些真正的代码来说明。下面是一个嵌套函数作用域的例子。

```
X = 99                                # Global scope name: not used

def f1():
    X = 88                            # Enclosing def local
    def f2():
        print(X)                     # Reference made in nested def
        f2()
    f1()                              # Prints 88: enclosing def local
```

首先，这是一段合法的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（并且只对f1中的代码可见）。

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。

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)                # Remembers X in enclosing def scope
    return f2                   # Return f2 but don't call it

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（将在第六部分介绍）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):          # Make and return action
...         return X ** N      # action retains N from enclosing scope
...     return action
...
```

这定义了一个外部的函数，这个函数简单地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：

```
>>>f(3)                # Pass 3 to X, N remembers 2: 3 ** 2
9
>>>f(4)                # 4 ** 2
16
```

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`被作为执行的状态信息保留了下来，我们返回其参数的平方运算。

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 remembers 3, f remembers 2
>>>g(3)                 # 3 ** 3
27
>>>f(3)                 # 3 ** 2
9
```

这能够奏效，因为像这样对一个工厂函数的每次调用，都得到了自己的状态信息的集合。在我们的例子中，我们赋给名称`g`的函数记住了3，`f`记住了2，因为每个函数都有自己的状态信息由`maker`中的变量`N`保持。

这是一种相当高级的技术，除了那些拥有函数式编程背景的程序员们，以后在实际使用中也不会常常见到。另一方面，嵌套的作用域常常被`lambda`函数创建表达式使用（本章稍后将介绍它）——因为它们是表达式，它们几乎总是嵌套在一个`def`中。此外，函数嵌套通常用作装饰器（第38章将介绍）——在某些情况下，它是最为合理的编码模式。

通常来说，类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引用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。为了看看它们是如何实现的，第18章全面介绍了默认参数，而下一小节将介绍足够的默认参数的基础知识。

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):          # Remember enclosing scope X with defaults
        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。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，这是由于第二个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，这会让程序更加得简单。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                # Pass x along instead of nesting
...     f2(x)                # Forward reference okay
...
>>>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Python中的嵌套作用域，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19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。

像def一样，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能够看到所有在所编写的函数中可用的变量。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

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
    return action
```

由于lambda是表达式，所以它们自然而然地（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一个值得注意的特例：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):
...         acts.append(lambda x: i ** x)      # Tries to remember each i
...         # 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
```

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，特别是在生成应用于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）。我们将会在第18章对默认参数和第19章对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3}。

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，将在第六部分介绍）会被搜索：

注3： 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。第20章会再谈这个话题。

```

>>>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。如果你尽可能地少定义嵌套函数，那么你和同事的生活，都会变得更美好。

nonlocal语句

上一小节中，我们介绍了嵌套函数可以引用一个嵌套的函数作用域中的变量的方法，即便这个函数已经返回了。事实上，在Python 3.0中，我们也可以修改这样的嵌套作用域变量，只要我们在一条nonlocal语句中声明它们。使用这条语句，嵌套的def可以对嵌套函数中的名称进行读取和写入访问。

nonlocal语句是global的近亲，前面已经介绍过global。nonlocal和global一样，声明了将要在一个嵌套的作用域中修改的名称。和global的不同之处在于，nonlocal应用于一个嵌套的函数的作用域中的一个名称，而不是所有def之外的全局模块作用域；而且在声明nonlocal名称的时候，它必须已经存在于该嵌套函数的作用域中——它们可能只存在于一个嵌套的函数中，并且不能由一个嵌套的def中的第一次赋值创建。

换句话说，nonlocal即允许对嵌套的函数作用域中的名称赋值，并且把这样的名称的作用域查找限制在嵌套的def。直接效果是更加直接和可靠地实现了可更改的作用域信息，对于那些不想要或不需要带有属性的类的程序而言。

nonlocal基础

Python 3.0引入了一条新的nonlocal语句，它只在一个函数内有意义：

```

def func():
    nonlocal name1, name2, ...

```

这条语句允许一个嵌套函数来修改在一个语法嵌套函数的作用域中定义的一个或多个名称。在Python 2.X（包括2.6）中，当一个函数def嵌套在另一个函数中，嵌套的函数引用

嵌套的def的作用域中的赋值所定义的任何名称，但是不能修改它们。在Python 3.0中，在一条nonlocal语句中声明嵌套的作用域，使得嵌套的函数能够赋值，并且由此也能够修改这样的名称。

这提供了一种方式，使得嵌套的函数能够提供可写的状态信息，以便在随后调用嵌套的函数的时候能够记住这些信息。允许状态修改，这使得嵌套函数更有用（例如，想象嵌套的作用域中的一个计数器）。在Python 2.X中，程序员通常使用类或其他的方法来实现类似的目标。由于嵌套函数已经成为实现状态保持的一种较为常见的编码模式，然而，nonlocal使得它更广泛地应用。

除了允许修改嵌套的def中的名称，nonlocal语句还加快了引用——就像global语句一样，nonlocal使得对该语句中列出的名称的查找从嵌套的def的作用域中开始，而不是从声明函数的本地作用域开始。也就是说，nonlocal也意味着“完全略过我的本地作用域”。

实际上，当执行到nonlocal语句的时候，nonlocal中列出的名称必须在一个嵌套的def中提前定义过，否则，将会产生一个错误。直接效果和global很相似：global意味着名称位于一个嵌套的模块中，nonlocal意味着它们位于一个嵌套的def中。nonlocal甚至更严格——作用域查找只限定在嵌套的def。也就是说，nonlocal名称只能出现在嵌套的def中，而不能在模块的全局作用域中或def之外的内置作用域中。

nonlocal的添加并没有改变通用的名称引用作用域规则；它们仍然像以前一样工作，即前面所描述的每条“LEGB”规则。nonlocal语句主要作用是允许嵌套的作用域中的名称被修改，而不只是被引用。然而，当在一个函数中使用的时候，global和nonlocal语句都在某种程度上限制了查找规则：

- global使得作用域查找从嵌套的模块的作用域开始，并且允许对那里的名称赋值。如果名称不存在于该模块中，作用域查找继续到内置作用域，但是，对全局名称的赋值总是在模块的作用域中创建或修改它们。
- nonlocal限制作用域查找只是嵌套的def，要求名称已经存在于那里，并且允许对它们赋值。作用域查找不会继续到全局或内置作用域。

在Python 2.6中，对嵌套的def作用域名称的引用是允许的，但不能对其赋值。然而，我们仍然可以使用带有显式属性的类来实现与nonlocal相同的可改变的状态信息的效果（并且，在某些环境下，这么做可能会更好）；全局属性和函数属性有时候也能实现类似的目的。稍后我会更详细地介绍这一点，首先，让我们看一些实际的代码来更具体地了解它们。

nonlocal应用

这些例子都在Python 3.0中运行。对嵌套的def作用域的引用像在Python 2.6中一样工作。在下面的代码中，tester构建并返回函数nested以便随后调用，nested中的state引用使用超常规的作用域查找规则来映射tester的本地作用域：

```
C:\misc>c:\python30\python

>>>def tester(start):
...     state = start                # Referencing nonlocals works normally
...     def nested(label):
...         print(label, state)      # Remembers state in enclosing scope
...     return nested
...
>>>F = tester(0)
>>>F('spam')
spam 0
>>>F('ham')
ham 0
```

默认情况下，不允许修改嵌套的def作用域中的名称；这也是Python 2.6的一般情况：

```
>>>def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...         state += 1              # Cannot change by default (or in 2.6)
...     return nested
...
>>>F = tester(0)
>>>F('spam')
UnboundLocalError: local variable 'state' referenced before assignment
```

使用nonlocal进行修改

现在，在Python 3.0下，如果我们在nested中把tester作用域中的state声明为一个nonlocal，我们就可以在nested函数中修改它了。即便我们通过名称F调用返回的nested函数时，tester已经返回并退出了，这也是有效的：

```
>>>def tester(start):
...     state = start                # Each call gets its own state
...     def nested(label):
...         nonlocal state           # Remembers state in enclosing scope
...         print(label, state)
...         state += 1              # Allowed to change it if nonlocal
...     return nested
...
>>>F = tester(0)
>>>F('spam')                        # Increments state on each call
spam 0
>>>F('ham')
```

```
ham 1
>>>F('eggs')
eggs 2
```

通常使用嵌套作用域引用时，我们可以多次调用`tester`工厂函数，以便在内存中获得其状态的多个副本。嵌套作用域中的`state`对象基本上附加到了返回的`nested`函数对象，每次调用都产生一个新的、独特的`state`对象，以至于更新一个函数的`state`不会影响到其他的。如下代码继续前面的交互式程序：

```
>>>G = tester(42)                                # Make a new tester that starts at 42
>>>G('spam')
spam 42

>>>G('eggs')                                       # My state information updated to 43
eggs 43

>>>F('bacon')                                       # But F's is where it left off: at 3
bacon 3                                             # Each call has different state information
```

边界情况

有几件事情需要注意。首先，和`global`语句不同，当执行一条`nonlocal`语句时，`nonlocal`名称必须已经在一个嵌套的`def`作用域中赋值过，否则将会得到一个错误——不能通过在嵌套的作用域中赋给它们一个新值来创建它们：

```
>>>def tester(start):
...     def nested(label):
...         nonlocal state                # Nonlocals must already exist in enclosing def!
...         state = 0
...         print(label, state)
...     return nested
...
SyntaxError: no binding for nonlocal 'state' found

>>>def tester(start):
...     def nested(label):
...         global state                  # Globals don't have to exist yet when declared
...         state = 0                    # This creates the name in the module now
...         print(label, state)
...     return nested
...
>>>F = tester(0)
>>>F('abc')
abc 0
>>>state
0
```

其次，`nonlocal`限制作用域查找仅为嵌套的`def`，`nonlocal`不会在嵌套的模块的全局作用域或所有`def`之外的内置作用域中查找，即便已经有了这些作用域：

```
>>>spam = 99
>>>def tester():
...     def nested():
...         nonlocal spam           # Must be in a def, not the module!
...         print('Current=', spam)
...         spam += 1
...     return nested
...
SyntaxError: no binding for nonlocal 'spam' found
```

一旦你意识到Python不能普遍地知道在哪个嵌套的作用域中创建一个全新的名称，这些限制就有意义了。在前面的程序中，spam应该在tester中赋值，还是在模块之外赋值？由于这不明确，Python必须在函数创建的时候解析nonlocal，而不是在函数调用的时候。

为什么使用nonlocal

假设有了极其复杂的嵌套函数，你会为一团糟乱而感到吃惊。尽管很难在我们的示例中看到这点，但在很多程序中，状态信息变得很重要。在Python中，有各种不同的方法来“记住”跨函数和方法的信息。尽管都有利有弊，对于嵌套的作用域引用，nonlocal确实起到了改进作用——nonlocal语句允许在内存中保持可变状态的多个副本，并且解决了在类无法保证的情况下的简单的状态保持。

正如我们在前面小节所看到的，如下的代码允许在一个嵌套作用域中保持和修改状态。对tester的每次调用都创建了可变信息的一个小小的自包含包，可变信息的名称不会与程序的其他部分产生任何冲突：

```
def tester(start):
    state = start                # Each call gets its own state
    def nested(label):
        nonlocal state          # Remembers state in enclosing scope
        print(label, state)
        state += 1              # Allowed to change it if nonlocal
    return nested

F = tester(0)
F('spam')
```

遗憾的是，这段代码只能在Python 3.0中工作。如果你使用Python 2.6，根据你的目标的不同，也有其他的选择。下面两个小节介绍了一些替代方法。

与全局共享状态

在Python 2.6中实现nonlocal效果的一种通常方法也是较早的方法，就是直接把状态移出全局作用域（嵌套的模块）：

```

>>>def tester(start):
...     global state                # Move it out to the module to change it
...     state = start              # global allows changes in module scope
...     def nested(label):
...         global state
...         print(label, state)
...         state += 1
...     return nested
...
>>>F = tester(0)
>>>F('spam')                      # Each call increments shared global state
spam 0
>>>F('eggs')
eggs 1

```

在这个例子中，这是有效的，但它需要在两个函数中都有`global`声明，并且倾向于引起全局作用域中的名称冲突（如果“状态”已经使用了会怎样？）。更糟糕但更为微妙的问题是，它只考虑到模块作用域中状态信息的单个共享副本——如果我们再次调用`tester`，将会重新设置模块的状态变量，以至于前面的调用将会看到自己的状态被覆盖：

```

>>>G = tester(42)                  # Resets state's single copy in global scope
>>>G('toast')
toast 42

>>>G('bacon')
bacon 43

>>>F('ham')                        # Oops -- my counter has been overwritten!
ham 44

```

正如前面所示，当使用`nonlocal`而不是`global`的时候，对`tester`的每次调用都记得`state`对象的自己的独特副本。

使用类的状态（预览）

Python 2.6中针对可改变信息的另一种较早的方法是使用带有属性的类，从而让状态信息的访问比隐式的范围查找规则更明确。作为一个额外的优点，一个类的每个实例都得到状态信息的一个新副本，作为Python的对象模型的一个天然的副产品。

我们还没有详细地介绍类，作为一个简短的概览，这里把前面使用的`tester/nested`函数作为类来重新实现——`state`在对象创建的时候显式地保存在对象中。为了让这段代码有意义，我们需要知道像这样的类中的`def`与一个类之外的`def`完全一样的工作，除非函数的`self`参数自动接收隐式的调用主体（通过调用类自身创建的一个实例对象）：

```

>>>class tester:                  # Class-based alternative (see Part VI)

```

```

...     def __init__(self, start):           # On object construction,
...         self.state = start               # save state explicitly in new object
...     def nested(self, label):
...         print(label, self.state)         # Reference state explicitly
...         self.state += 1                 # Changes are always allowed
...
>>>F = tester(0)                           # Create instance, invoke __init__
>>>F.nested('spam')                         # F is passed to self
spam 0
>>>F.nested('ham')
ham 1

>>>G = tester(42)                           # Each instance gets new copy of state
>>>G.nested('toast')                       # Changing one does not impact others
toast 42

>>>G.nested('bacon')
bacon 43

>>>F.nested('eggs')                         # F's state is where it left off
eggs 2
>>>F.state                                 # State may be accessed outside class
3

```

我们将在本书后面更深入地介绍Python的神奇之处，我们也可以使用运算符重载让类看上去像是一个可调用函数。`__call__`获取了一个实例上的直接调用，因此，我们不需要调用一个指定的方法：

```

>>>class tester:
...     def __init__(self, start):
...         self.state = start
...     def __call__(self, label):          # Intercept direct instance calls
...         print(label, self.state)       # So .nested() not required
...         self.state += 1
...
>>>H = tester(99)
>>>H('juice')                             # Invokes __call__
juice 99
>>>H('pancakes')
pancakes 100

```

在本书中，此刻不要太多地探究这段代码的细节，我们将在第六部分深入探讨类，并且在第29章看到`__call__`这样的特定运算符重载工具，因此，你可能想要记下这段代码以供将来参考。这里的关键是类可以让状态信息更明显，通过利用显示属性赋值而不是作用域查找。

尽管使用类存储状态信息通常是可以遵从的良好规则，在这里所示的情况下，它们可能有些杀鸡用牛刀了，其中状态是一个单个的计数器。这样小的状态例子比你所想象的更常见，在这样的环境下，嵌套的`def`有时候比编写类还要简单，特别是，如果你还不熟

悉OOP的话。此外，还有一些情况，嵌套的def可能实际上比类表现得更好（参见第38章中对方法装饰器的介绍，那里有一个示例，这一话题超出了本书的范围）。

使用函数属性的状态

作为最后一种状态保持选项，我们有时候可以使用函数属性实现与nonlocal相同的效果——用户定义的名称直接附加给函数。这里给出了基于这一技术的示例的最后一个版本——它用附加给嵌套的函数的一个属性替代了nonlocal。尽管这种方法可能对某些人来说不那么容易理解，但它允许从嵌套的函数之外访问状态变量（使用nonlocals，我们只能在嵌套的def内部看到状态变量）：

```
>>>def tester(start):
...     def nested(label):
...         print(label, nested.state)      # nested is in enclosing scope
...         nested.state += 1               # Change attr, not nested itself
...         nested.state = start            # Initial state after func defined
...         return nested
...
>>>F = tester(0)
>>>F('spam')                               # F is a 'nested' with state attached
spam 0
>>>F('ham')
ham 1
>>>F.state                                   # Can access state outside functions too
2
>>>
>>>G = tester(42)                           # G has own state, doesn't overwrite F's
>>>G('eggs')
eggs 42
>>>F('ham')
ham 2
```

这段代码依赖于一个事实：函数名nested是包围nested的tester作用域中的一个本地变量；同样，它可以在nested内自由地引用。这段代码还依赖于这样一个事实：本地修改一个对象并不是给一个名称赋值；当它自增nested.state，它是在修改对象nested引用的一部分，而不是指定nested本身。由于我们不是要真的在嵌套作用域内给一个名称赋值，所以不需要nonlocal。

正如你所看到的，全局、非本地、类和函数属性都提供了状态保持的选项。全局只支持共享的数据，类需要OOP的基本知识，类和函数属性都允许在嵌套函数自身之外访问状态。通常，你的程序的最好的工具取决于程序的目的。

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）。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，除非它们特定地声明为全局变量或非本地变量。我们也学过了一些更高级的作用域概念，包括嵌套函数作用域和函数属性。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修改）。

在下一章中，我们将继续介绍函数，讲解与函数相关的第二个重要概念，即参数传递。你将发现，参数通过赋值传递给一个函数，但Python也提供了工具允许函数灵活地确定如何传递其各项的值。在继续学习之前，让我们来做本章的练习，以回顾在这里已经介绍过的作用域概念。

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. 下面的代码会输出什么？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？为什么？

```
>>>X = 'Spam'
>>>def func():
...     X = 'NI'
...     def nested():
...         print(X)
...     nested()
...
>>>func()
>>>X
```

6. 这段代码在Python 3.0下会输出什么？为什么？

```
>>>def func():
...     X = 'NI'
...     def nested():
...         nonlocal X
...         X = 'Spam'
...     nested()
...     print(X)
...
>>>func()
```

7. 举出三种或四种Python函数中保存状态信息的方法。

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. 这个示例打印出'Spam'，因为nonlocal语句（Python 3.0中可用，但Python 2.6中不可用）意味着在嵌套函数中对X赋值，以修改嵌套函数的本地作用域中的X。没有

这条语句，这个赋值将会把X当作是嵌套函数的本地变量，使它成为一个不同的变量，那么这段代码将会打印出'NI'。

7. 尽管函数返回的时候本地变量的值已经不在，我们可以使用共享的全局变量、嵌套函数内的嵌套函数作用域引用，或者使用默认参数值来让一个Python函数保持状态信息。函数属性有时候允许把状态附加到函数自身，而不是在作用域中查找。另一种替代方法，使用类来OOP，有时候比其他任何基于作用域的技术更好地支持状态保持，因为它使得属性赋值很明确，我们将在第六部分介绍这一选项。

参数

第17章介绍了Python的作用域背后的细节——即定义和查找变量的位置。正如我们所学过的，在代码中定义一个名称的位置决定了它大部分的含义。本章继续介绍函数，学习Python中的参数传递的概念，即对象作为输入发送给函数的方式。正如我们将看到的，参数（argument，也叫做parameter）赋值给一个函数中的名称，但是，它们更多地与对象引用相关，而不是与变量作用域相关。我们还将介绍Python所提供的额外工具，如关键字、默认值和任意参数收集器，它们为参数发送给函数的方式提供了广泛的灵活性。

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，因而会在这一部分进行详尽的阐述。下面是给函数传递参数时的一些简要的关键点。

- 参数的传递是通过自动将对象赋值给本地变量名来实现的。函数参数[调用者发送的（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。因为引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作用域中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，因为参数是简单地赋值给传入的对象，函数能够就地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。

更多关于引用细节请参看第6章。我们这里所学的对于函数的参数来说也适用，尽管对参数名的赋值是自动并且隐式的。

Python的通过赋值进行传递的机制与C++的引用参数选项并不完全相同，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“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，但是因为你无论如何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，并且无论对可变对象或不可变对象都是这样的。

参数和共享引用

为了说明参数传递属性的工作方式，考虑如下的代码：

```
>>>def f(a):                # a is assigned to (references) passed object
...     a = 99                # Changes local variable a only
...
>>>b = 88
>>>f(b)                      # a and b both reference same 88 initially
>>>print(b)                  # b is not changed
88
```

在这个例子中，在使用f(b)调用函数的时候，变量a赋值了对象88，但是，a只是存在于调用的函数之中。在函数中修改a对于调用函数的地方没有任何影响，它直接把本地变量a重置为一个完全不同的对象。

这就是没有名称冲突的含义——对函数中的一个参数名的赋值（例如，a=99）不会神奇地影响到函数调用作用域中的b这样的变量。参数名称可能最初共享传递的对象（它们实质上是指向这些对象的指针），但只是临时的，即当函数第一次调用的时候。只要对参数名进行重新赋值，这种关系就结束了。

至少，这是对参数名称自身赋值的情况。当参数传递像列表和字典这样的可修改对象的时候，我们还需要注意，对这样的对象的原处修改可能在函数退出后依然有效，并由此影响到调用者。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):        # Arguments assigned references to objects
```

```

...     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，并给参数b所引用的一个对象元素赋值。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，它仅仅把本地变量a修改为引用一个完全不同的对象，并没有改变调用者作用域中的名称X的绑定。这和前面的例子中的情况是相同的。
- b也是一个本地变量名，但是它被传给了一个可变对象（在调用者作用域中叫做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，changer中的第二条赋值语句没有修改b，我们修改的是b当前所引用的对象的一部分。这种原处修改，只有在修改的对象比函数调用生命更长的时候，才会影响到调用者。名称L也没有改变——它仍然引用同样的、修改后的对象，但是，就好像L在调用后变化了一样，因为它引用的值已经在函数中修改过了。

图18-1 表明了函数被调用后，函数代码在运行前，变量名/对象所存在的绑定关系。

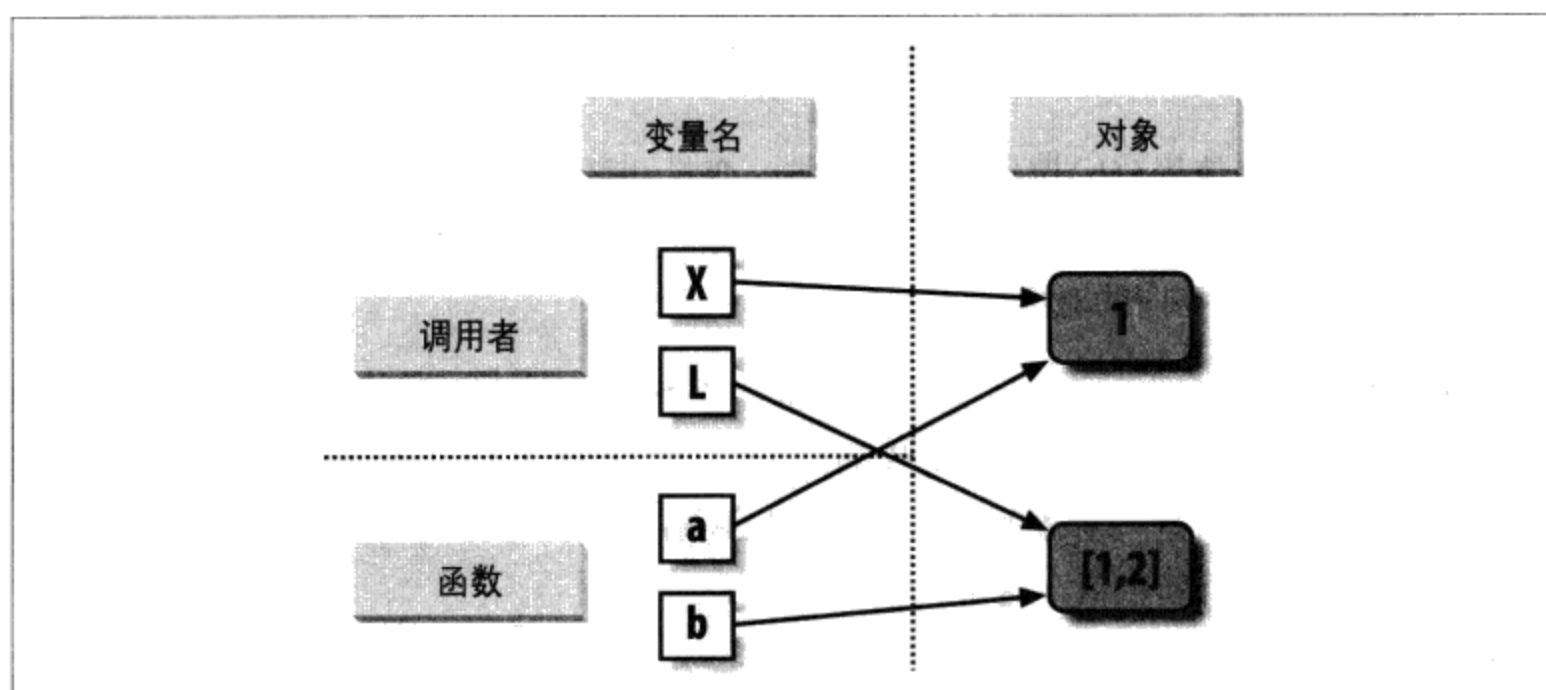


图18-1：引用：参数。因为参数是通过赋值传递的，函数中的参数名可以在调用时通过变量实现共享对象。因此，函数中对可变对象参数的在原处的修改能够影响调用者。这里，函数中的a和b在函数一开始调用时最初通过变量X和L进行了对象引用。通过变量b对列表的改变在函数调用返回后，L也会发生改变

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得就像函数的一个输入和一个输出。

避免可变参数的修改

对可变参数的原处修改的行为不是一个bug——它只是参数传递在Python中工作的方式。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：这意味着不需要创建多个拷贝就可以在我們的程序中传递很大的对象，并且能够按照需要方便地更新这些对象。实际上，正如我们将在本书第六部分看到的，Python的类模型依赖于原处修改一个传入的“self”参数来更新对象状态。

如果不想要函数内部在原处的修改影响传递给它的对象，那么，我们可以简单地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。

```
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。此外，原处修改传入的可变对象的函数，可能是为此而设计并有意而为之——修改可能是一个定义良好的API的一部分，而我們不应该通过产生副本来违反该API。

但是，你必须意识到这个属性：如果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”一节、第9章以及第11章的“赋值语句”一节）。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。

注意： Python 2.X中的解包参数：前面的例子用元组赋值解包了函数返回的一个元组。在Python 2.6中，可能在传递给函数的参数中自动解包元组。在Python 2.6中，通过如下头部定义的一个函数：

```
def f((a, (b, c))):
```

可以用与期望的结构匹配的元组来调用：f((1, (2, 3)))分别给a、b和c赋值为1、2和3。自然的，传递的元组可以是调用（f(T)）前创建的一个对象。这个def语法在Python 3.0中不再支持。相反，像下面这样编写函数：

```
def f(T): (a, (b, c)) = T
```

以便在一条显式赋值语句中解包。这种显式形式在Python 3.0和Python 2.6中都有效。参数解包在Python 2.X中是一个含糊并且很少用到的功能。此外，Python 2.6中的函数头部只支持序列赋值的元组形式；更通用的序列赋值（例如，def f((a, [b, c])):）在Python 2.6中因语法问题而无效，并且必须用显式赋值形式。

元组解包参数语法在Python 3.0的lambda函数参数列表中也是不允许的：参见本书第20章中的“为什么要在意：列表解析和map”给出的例子。有些不对称的是，元组解包赋值在Python 3.0中仍然是自动化的，以便实现循环，参见第13章的例子。

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，但是允许编写支持更复杂的调用模式的函数，并且你可能会遇到需要这些工具的一些库。

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。

基础知识

在学习语法的细节之前，我需要强调一下，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，你很有可能在这儿犯错，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，也是我们迄今为止最常使用的那种方法，是通过位置进行匹配把参数值传递给函数头部的参数名称，匹配顺序为从左到右。

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（这个特性常常叫做可变参数，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）。

可变参数解包：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。

Keyword-only参数：参数必须按照名称传递

在Python 3.0中（不包括Python 2.6中），函数也可以指定参数，参数必须用带有关键参数的名字（而不是位置）来传递。这样的参数通常用来定义实际参数以外的配置选项。

匹配语法

表18-1总结了与特定参数匹配模式有关的语法。

表18-1：函数参数匹配表

语法	位置	解释
<code>func(value)</code>	调用者	常规参数：通过位置进行匹配
<code>func(name=value)</code>	调用者	关键字参数：通过变量名匹配
<code>func(*sequence)</code>	调用者	以 <code>name</code> 传递所有的对象，并作为独立的基于位置的参数
<code>func(**dict)</code>	调用者	以 <code>name</code> 成对的传递所有的关键字/值，并作为独立的关键字参数
<code>def func(name)</code>	函数	常规参数：通过位置或变量名进行匹配

表18-1：函数参数匹配表（续）

语法	位置	解释
<code>def func(name=value)</code>	函数	默认参数值，如果没有在调用中传递的话
<code>def func(*name)</code>	函数	匹配并收集（在元组中）所有包含位置的参数
<code>def func(**name)</code>	函数	匹配并收集（在字典中）所有包含位置的参数
<code>def func(*args, name)</code>	函数	参数必须在调用中按照关键字传递
<code>def func(*, name=value)</code>		（Python 3.0）

这些特殊的匹配模式分解到如下的函数调用和定义中：

- 在函数的调用中（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 `*sequence` 或者 `**dict` 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，并且在将它们传递给函数的时候，将它们解包为分开的、单个的参数。
- 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Python 3.0及其以后的版本中，跟在 `*name` 或一个单独的 `*` 之后的、任何正式的或默认的参数名称，都是 *keyword-only* 参数，并且必须在调用中按照关键字传递。

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。在本书前面，我们已经非正式地使用过这两种形式了：

- 我们已经使用关键字来指定Python 3.0的`print`函数的选项，但是，它们有更广泛的用途——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。

正如我们将看到的，函数头部的默认参数和调用中的关键字的这些组合，进一步允许我们挑选要覆盖哪些默认参数。

简而言之，特定的参数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。

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，参数必须以此顺序出现：任何位置参数（`value`），后面跟着任何关键字参数（`name=value`）和`*sequence`形式的组合，后面跟着`**dict`形式。
- 在函数头部，参数必须以此顺序出现：任何一般参数（`name`），紧跟着任何默认参数（`name=value`），如果有的话，后面是`*name`（或者在Python 3.0中是`*`）的形式，后面跟着任何 `name`或`name=value` keyword-only参数（在Python 3.0中），后面跟着`**name`形式。

在调用和函数头部中，如果出现`**arg`形式的话，都必须出现在最后。如果你使用任何其他顺序混合了参数，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. 其他额外的关键字参数分配到`**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。当所有的匹配都完成了，Python把传递给参数名的对象赋值给它们。

Python使用的真正的匹配算法更复杂一些（例如，它必须考虑Python 3.0中的keyword-only参数），因此，要了解更为详细的介绍，请参考Python的标准语言手册。这不是必须阅读的材料，但是它所介绍的Python匹配算法能够帮助你理解一些令人费解的情况，特别是当模式混合的时候。

注意： 在Python 3.0中，函数头部中的参数名称也可以有一个注解值，特定形式如`name:value`（或者要给出默认值的话是`name:value=default`形式）。这只是参数的一个额外语法，不会增加或修改这里所介绍的参数顺序规则。函数自身也可以有一个注解值，以`def f()->value`的形式给出。参见第19章对于函数注解的介绍，以了解更详细的内容。

关键字参数和默认参数的实例

使用代码来解释要比前文描述所暗含的意思更简单。如果你没有使用过任何特殊的匹配

语法，Python默认会通过位置从左至右匹配变量名。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。

```
>>>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（这在Python 3.0和Python 2.6中都同样有效，但是，在Python 2.6中会显示额外的元组圆括号，因为我们使用Python 3.0的print调用）：

```
>>>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按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，我们将在下一部分介绍。

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。

```
>>>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还是关键字参数来实现。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，只有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，这是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。

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值参数）。

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)
...

```

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。例如，在下面的代码中，1按照位置传递给a，2和3收集到pargs位置元组中，x和y放入kargs关键字词典中：

```
>>>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。

解包参数

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。它会解包参数的集合，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将它们解包成不同的参数。

```
>>>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/值对的形式解包一个字典，使其成为独立的关键字参数。

```
>>>args = {'a': 1, 'b': 2, 'c': 3}
>>>args['d'] = 4
>>>func(**args)
1 2 3 4
```

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

>>>func(1, *(2, 3), d=4)
1 2 3 4
```

```
>>>f(1, *(2,), c=3, **{'d':4})
1 2 3 4
```

在编写脚本时，当我们不能预测将要传入函数的参数的数量的时候，这种代码是很方便的。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，别混淆函数头部或函数调用时`*/**`的语法：在头部，它意味着收集任意数量的参数，而在调用时，它解包任意数量的参数。

注意：正如我们在第14章所见到过的，调用中的`*args`形式是一个迭代环境，因此技术上它接受任何可迭代对象，而不仅是像这里的示例所示的元组或其他序列。例如，一个文件对象在`*`之后工作，并且将其行解包为单个的参数（例如，`func(*open('fname'))`）。

Python 3.0和Python 2.6都支持这种通用性，但是，它只对调用情况成立——调用中的`*args` 允许任何可迭代对象，但是，一个`def`头部中的相同格式总是把额外的参数绑定到一个元组。这个头部行为在内涵上和语法上都与我们在第11章所介绍的Python 3.0扩展序列解包语法形式类似（例如`x,*y = z`），尽管该功能总是创建列表而不是元组。

应用函数通用性

前面小节的示例可能有些笨拙，但它们比我们想象的更常用。很多程序需要以通用的形式调用任意函数，提前并不知道函数的名称和参数。实际上，特殊的“`varargs`”调用的真正强大之处是，在编写一段脚本之前不需要知道一个函数调用需要多少参数。例如，可以使用`if`逻辑来从一系列函数和参数列表中选择，并且通用地调用其中任何一个：

```
if <test>:
    action, args = func1, (1,)           # Call func1 with 1 arg in this case
else:
    action, args = func2, (1, 2, 3)      # Call func2 with 3 args here
...
action(*args)                           # Dispatch generically
```

更广泛地说，当你无法预计参数列表的任何时候，这种`varargs`调用语法很有用。例如，如果用户通过一个用户界面选择任意一个函数，你可能在编写自己的脚本的时候无法直接编写一个函数调用。要解决这个问题，直接用序列操作构建一个参数列表，并且用带星号的名称解包参数以调用它：

```
>>>args = (2,3)
>>>args += (4,)
>>>args
(2, 3, 4)
>>>func(*args)
```

由于这里的参数列表时作为元组传入的，程序可以在运行时构建它。这种技术对于那些

测试和计时其他函数的函数来说很方便。例如，在下面的代码中，我们通过传递任何发送进来的参数来支持具有任意参数的任意函数：

```
def tracer(func, *pargs, **kargs):           # Accept arbitrary arguments
    print('calling:', func.__name__)
    return func(*pargs, **kargs)            # Pass along arbitrary arguments

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))
```

当这段代码运行的时候，tracer收集参数，然后以varargs调用语法来传递它：

```
calling: func
10
```

我们将在本书随后看到这种用法的更大的例子，特别参见第20章的序列计时示例，以及我们将在第38章编写的装饰器工具。

废弃的apply内置函数（Python 2.6）

在Python 3.0之前，*args和**args varargs调用语法的效果可以通过一个名为apply的内置函数来实现。这一最初的技术在Python 3.0中已经废除了，因为它现在是多余的了（Python 3.0清除了众多的这类含糊不清但已经存在多年的工具）。它在Python 2.6中仍然可用，并且你可能会在旧的Python 2.X代码中遇到它。

简而言之，下面是前面的Python 3.0代码的对等形式：

```
func(*pargs, **kargs)           # Newer call syntax: func(*sequence, **dict)

apply(func, pargs, kargs)        # Defunct built-in: apply(func, sequence, dict)
```

例如，考虑如下的函数，它接受任意数量的可选关键字参数：

```
>>>def echo(*args, **kwargs): print(args, kwargs)
...
>>>echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

在Python 2.6中，我们可以用apply通用性地调用它，或使用Python 3.0中所必须的调用语法：

```
>>>pargs = (1, 2)
>>>kargs = {'a':3, 'b':4}

>>>apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}
```

```
>>>echo(*pargs, **kargs)
(1, 2) {'a': 3, 'b': 4}
```

解包调用语法形式比`apply`函数新，通常也推荐使用它，并且是Python 3.0所必需的。除了它与`def`头部的`*pargs`和`**kargs`收集器对称，实际上它总体上需要较少的键盘录入，新的调用语法还允许我们传递额外的参数而不必手动扩展参数序列或字典：

```
>>>echo(0, c=5, *pargs, **kargs)           # Normal, keyword, *sequence, **dictionary
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

也就是，这种调用语法形式更为通用。既然它在Python 3.0中是必须的，你现在应该忘掉关于`apply`的所有知识（除非，它出现在你必须使用或维护的Python 2.X代码中）。

Python 3.0 Keyword-Only参数

Python 3.0把函数头部的排序规则通用化了，允许我们指定`keyword-only`参数——即必须只按照关键字传递并且不会由一个位置参数来填充的参数。如果想要一个函数既处理任意多个参数，也接受可能的配置选项的话，这是很有用的。

从语法上讲，`keyword-only`参数编码为命名的参数，出现在参数列表中的`*args`之后。所有这些参数都必须在调用中使用关键字语法来传递。例如，在如下的代码中，`a`可能按照名称或位置传递，`b`收集任何额外的位置参数，并且`c`必须只按照关键字传递：

```
>>>def kwonly(a, *b, c):
...     print(a, b, c)
...
>>>kwonly(1, 2, c=3)
1 (2,) 3
>>>kwonly(a=1, c=3)
1 () 3
>>>kwonly(1, 2, 3)
TypeError: kwonly() needs keyword-only argument c
```

我们也可以在参数列表中使用一个`*`字符，来表示一个函数不会接受一个变量长度的参数列表，而是仍然期待跟在`*`后面的所有参数都作为关键字传递。在下一个函数中，`a`可能再次按照位置或名称传递，但`b`和`c`必须按照关键字传递，不允许其他额外的位置传递：

```
>>>def kwonly(a, *, b, c):
...     print(a, b, c)
...
>>>kwonly(1, c=3, b=2)
1 2 3
>>>kwonly(c=3, b=2, a=1)
1 2 3
>>>kwonly(1, 2, 3)
```

```
TypeError: kwonly() takes exactly 1 positional argument (3 given)
>>>kwonly(1)
TypeError: kwonly() needs keyword-only argument b
```

仍然可以对keyword-only参数使用默认值，即便它们出现在函数头部中的*的后面。在下面的代码中，a可能按照名称或位置传递，而b和c是可选的，但是如果使用的话必须按照关键字传递：

```
>>>def kwonly(a, *, b='spam', c='ham'):
...     print(a, b, c)
...
>>>kwonly(1)
1 spam ham
>>>kwonly(1, c=3)
1 spam 3
>>>kwonly(a=1)
1 spam ham
>>>kwonly(c=3, b=2, a=1)
1 2 3
>>>kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)
```

实际上，带有默认值的keyword-only参数都是可选的，但是，那些没有默认值的keyword-only参数真正地变成了函数必需的keyword-only参数：

```
>>>def kwonly(a, *, b, c='spam'):
...     print(a, b, c)
...
>>>kwonly(1, b='eggs')
1 eggs spam
>>>kwonly(1, c='eggs')
TypeError: kwonly() needs keyword-only argument b
>>>kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)

>>>def kwonly(a, *, b=1, c, d=2):
...     print(a, b, c, d)
...
>>>kwonly(3, c=4)
3 1 4 2
>>>kwonly(3, c=4, b=5)
3 5 4 2
>>>kwonly(3)
TypeError: kwonly() needs keyword-only argument c
>>>kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
```

排序规则

最后，注意keyword-only参数必须在一个单个星号后面指定，而不是两个星号——命名

的参数不能出现在**args任意关键字形式的后面，并且一个**不能独自出现在参数列表中。这两种做法都将产生语法错误：

```
>>>def konly(a, **pargs, b, c):
SyntaxError: invalid syntax
>>>def konly(a, **, b, c):
SyntaxError: invalid syntax
```

这意味着，在一个函数头部，keyword-only参数必须编写在**args任意关键字形式之前，且在*args任意位置形式之后，当二者都有的时候。无论何时，一个参数名称出现在*args之前，它可能是默认位置参数，而不是keyword-only参数：

```
>>>def f(a, *b, **d, c=6): print(a, b, c, d)           # Keyword-only before **!
SyntaxError: invalid syntax

>>>def f(a, *b, c=6, **d): print(a, b, c, d)           # Collect args in header
...
>>>f(1, 2, 3, x=4, y=5)                                # Default used
1 (2, 3) 6 {'y': 5, 'x': 4}

>>>f(1, 2, 3, x=4, y=5, c=7)                            # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>>f(1, 2, 3, c=7, x=4, y=5)                            # Anywhere in keywords
1 (2, 3) 7 {'y': 5, 'x': 4}

>>>def f(a, c=6, *b, **d): print(a, b, c, d)           # c is not keyword-only!
...
>>>f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

实际上，在函数调用中，类似的排序规则也是成立的：当传递keyword-only参数的时候，它们必须出现在一个**args形式之前。keyword-only参数可以编写在*args之前或者之后，并且可能包含在**args中：

```
>>>def f(a, *b, c=6, **d): print(a, b, c, d)           # KW-only between * and **
...
>>>f(1, *(2, 3), **dict(x=4, y=5))                     # Unpack args at call
1 (2, 3) 6 {'y': 5, 'x': 4}

>>>f(1, *(2, 3), **dict(x=4, y=5), c=7)                # Keywords before **args!
SyntaxError: invalid syntax

>>>f(1, *(2, 3), c=7, **dict(x=4, y=5))                 # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>>f(1, c=7, *(2, 3), **dict(x=4, y=5))                 # After or before *
1 (2, 3) 7 {'y': 5, 'x': 4}

>>>f(1, *(2, 3), **dict(x=4, y=5, c=7))                 # Keyword-only in **
1 (2, 3) 7 {'y': 5, 'x': 4}
```


请自行分析这些情况，结合前面正式介绍的一般的参数排序规则。在这些人为编写的例子中，它们可能是最糟糕的情况，但是，在实际中有可能会遇到它们，特别是那些编写库和工具供其他Python程序员使用的人。

为何使用keyword-only参数

那么，为什么要关心keyword-only参数？简而言之，它们使得很容易允许一个函数既接受任意多个要处理的位置参数，也接受作为关键字传递的配置选项。尽管它们的使用是可选的，没有keyword-only参数的话，要为这样的选项提供默认值并验证没有传递多余的关键字则需要额外的工作。

假设一个函数处理一组传入的对象，并且允许传递一个跟踪标志：

```
process(X, Y, Z)                # use flag's default
process(X, Y, notify=True)      # override flag default
```

没有keyword-only参数的话，我们必须使用*args和**args，并且手动地检查关键字，但是，有了keyword-only参数，需要较少的代码。下面的语句通过notify保证不会有位置参数错误匹配，并且要求它如果传递则作为一个关键字传递：

```
def process(*args, notify=False): ...
```

在本章后面“模拟Python 3.0 print函数”一节中，我们将继续看到一些更实用的例子，我们将在那里介绍剩下的内容。参阅本书第20章的迭代选项计时案例，那里给出了keyword-only参数用法的另一个例子。对于Python 3.0中的额外函数定义扩展，第19章将讨论函数注解语法。

min调用

让我们来介绍一些更为实际的内容吧。为了更加详细地介绍这一部分内容，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*的特性的自然示例：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的快，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1}。文件mins.py包含了所有三种解决办法的代码。

```
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)
    tmp.sort()
    return tmp[0]

# Or, in Python 2.4+: return sorted(args)[0]

print(min1(3,4,1,2))
print(min2("bb", "aa"))
```

注1： 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分次序。这种排序称为“timsort”，以其发明者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，如果参数有部分定序，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，你应该在第20章会碰到time和timeit模块进行计时。

```
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都会自动地抛出一个异常。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（而不是剪切和粘贴来改变一个字符）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！我们在第5章介绍数值工具的时候简略地介绍过它们，并且在第14章介绍迭代环境的时候再次遇到它们。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语言编写的，目的是为了优化运行速度，并接受一个单个的可迭代对象或多个参数。然而，尽管它在这一环境下是多余的，但我们在这里使用的通用编码模式可能在其他情况下有用。

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]:
        for other in args[1:]:
            if x not in other: break
        else:
            res.append(x)
    return res

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return res
```

Scan first sequence
For all other args
Item in each one?
No: break out of loop
Yes: add items to end

For all args
For all nodes

Add new items to result

因为这些工具是值得重用的（并且在交互模式下它们有些太大了以至于无法重新输入），我们将会把这些函数保存为一个名为`inter2.py`的模块文件（如果你已经忘记了模块和导入是如何工作的，参见第3章的介绍，或者参见第五部分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），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在不断地改进，本书中的例子有时会显得过时）。

模拟Python 3.0 print函数

为了圆满结束本章，让我们来看参数匹配用法的最后一个例子。这里看到的代码专门用于Python 2.6或更早的版本（它在Python 3.0下也能工作，但是没有什么意义）：它使用`*args`任意位置元组以及`**args`任意关键字参数字典来模拟Python 3.0 `print`函数所做的大多数工作。

正如我们在第11章所了解到的，这实际上不是必需的，因为Python 2.6程序员总是可以通过如下形式的一个导入来使用Python 3.0的`print`函数：

```
from __future__ import print_function
```

然而，为了说明一般性的参数匹配，如下的文件`print30.py`，用少量可重用的代码做了同样的工作：

```
"""
Emulate most of the 3.0 print function for use in 2.X
call signature: print30(*args, sep=' ', end='\n', file=None)
```

```

"""
import sys

def print30(*args, **kwargs):
    sep = kwargs.get('sep', ' ')          # Keyword arg defaults
    end = kwargs.get('end', '\n')
    file = kwargs.get('file', sys.stdout)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)

```

为了测试它，将其导入到另一个文件或交互提示模式中，并且像Python 3.0 `print`函数那样使用它。这里是一个测试脚本`testprint30.py`（注意，该函数必须叫做“`print30`”，因为“`print`”在Python 2.6中是保留字）：

```

from print30 import print30
print30(1, 2, 3)
print30(1, 2, 3, sep='')                # Suppress separator
print30(1, 2, 3, sep='...')
print30(1, [2], (3,), sep='...')        # Various object types

print30(4, 5, 6, sep='', end='')        # Suppress newline
print30(7, 8, 9)
print30()                                # Add newline (or blank line)

import sys
print30(1, 2, 3, sep='??', end='.\n', file=sys.stderr) # Redirect to file

```

在Python 2.6下运行的时候，我们得到了与Python 3.0的`print`函数相同的结果：

```

C:\misc>c:\python26\python testprint30.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9
1??2??3.

```

尽管在Python 3.0中没有意义，但运行的时候，结果是相同的。与通常情况一样，Python的通用性设计允许我们在Python语言自身中原型化或开发概念。在这个例子中，参数匹配工具在Python代码中与在Python的内部实现中一样的灵活。

使用Keyword-Only参数

这个例子可以使用本章前面所介绍的Python 3.0 keyword-only参数来编写，从而来自动验证配置参数，注意到这一点是很有趣的：

```
# Use keyword-only args

def print30(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

这个版本与最初的版本一样有效，并且它是说明keyword-only参数如何方便好用的基本例子。最初的版本假设所有的位置参数都要打印，并且所有的keyword-only参数都只是可选的。大多数情况下这样就够了，但是，任何额外的keyword-only参数都默默地忽略掉了。例如，如下的一个调用将会对keyword-only参数形式产生一个异常：

```
>>>print30(99, name='bob')
TypeError: print30() got an unexpected keyword argument 'name'
```

但是，会默默地忽略最初版本中的name参数。要手动检测多余的关键字，我们可以使用dict.pop()删除收到的条目，并检查字典是否为空。这里是keyword-only参数版本的一个对等形式：

```
# Use keyword args deletion with defaults

def print30(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('extra keywords: %s' % kargs)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

它和前面一样有效，但是，现在，它也会捕获外部的关键字参数：

```
>>>print30(99, name='bob')
TypeError: extra keywords: {'name': 'bob'}
```

这个版本的函数在Python 2.6下运行，但是，它比keyword-only参数版本需要额外的4行代码。遗憾的是，这个例子中需要额外的代码，keyword-only参数版本只在Python 3.0下工作，这否定了我编写这个例子的首要原因（一个Python 3.0的模拟程序，只能在Python 3.0下工作，其用途是令人难以置信的）。在编写来运行于Python 3.0的程序中，keyword-only参数可以简化一类既接受参数又接受选项的函数。在后面的第20章的迭代计时案例学习中，有Python 3.0的keyword-only参数的另一个示例。

为什么要在意：关键字参数

你可能已经知道了，高级参数匹配模式可能更复杂。它们也完全是可选的，你可以只使用简单的位置匹配，并且，当你刚开始编程的时候，这可能是一个好办法。然而，由于一些Python工具使用它们，了解一些关于这些模式的常识是很重要的。

例如，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（该模块在Python 2.6中的名称是Tkinter）。我们在本书的多个地方简单地介绍过tkinter，但只是介绍了当构建GUI组件的时候的调用模式、关键字参数设置配置选项。例如，一种调用形式：

```
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。

Python中的很多内置函数期待我们对使用模式的选项也用关键字参数，这可能有默认值也可能没有。例如，我们在第8章所学习过的sorted内置函数：

```
sorted(iterable, key=None, reverse=False)
```

期待我们传入一个可迭代对象来进行排序，但是，也允许我们传递可选的关键字参数来指定一个字典排序键和一个反向排序标志，其默认值分别为None和False。因为我们通常不会使用这些选项，它们可能会被省略而使用默认值。

本章小结

在本章中，我们学习了与函数相关的两个关键概念中的第二个：参数（对象如何传递给函数）。正如我们所学到的，参数通过赋值传递到函数中，赋值方式是通过对象引用，实际上是通过指针传递到函数中。我们还学习了一些更加高级的扩展，包括默认参数和关键字参数、使用任意的多个参数的工具，以及Python 3.0中的关键字参数。最后，我们还学习了可变参数如何表现出与其他的对象共享引用一样的行为——除非对象在发送进来的时候显式地复制，在一个函数中修改一个传入的可变对象可能会影响调用者。

下一章继续介绍函数，讨论一些与函数相关的更高级的概念：函数注解、lambda以及map和filter这样的函数工具。很多这样的概念都源自于这样一个事实：函数是Python

中的常规对象，并且支持一些高级的和非常灵活的处理方式。在学习这些主题之前，让我们先看看本章的习题以复习学习过的参数概念。

本章习题

1. 如下代码的输出是什么？为什么？

```
>>>def func(a, b=4, c=5):  
...     print(a, b, c)  
...  
>>>func(1, 2)
```

2. 如下代码的输出是什么？为什么？

```
>>>def func(a, b, c=5):  
...     print(a, b, c)  
...  
>>>func(1, c=3, b=2)
```

3. 如下代码的输出是什么？为什么？

```
>>>def func(a, *pargs):  
...     print(a, pargs)  
...  
>>>func(1, 2, 3)
```

4. 如下代码打印出什么？为什么？

```
>>>def func(a, **kargs):  
...     print(a, kargs)  
...  
>>>func(a=1, c=3, b=2)
```

5. 最后一次运行时，如下代码的输出是什么？为什么？

```
>>>def func(a, b, c=3, d=4): print(a, b, c, d)  
...  
>>>func(1, *(5,6))
```

6. 举出三种以上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“125”，因为1和2按照位置传递给了a和b，并且c在调用中被忽略了，默认为5。
2. 这次的输出是“123”：1按照位置传递给a，2和3按照名称传递给b和c（当像这样使用关键字参数的时候，从左到右的顺序无关紧要）。

3. 这段代码打印出“1 (2, 3)”，因为1传递给a，*pargs把其他的位置参数收集到一个新的元组对象中。我们可以用任何迭代工具来步进任何的额外的位置参数元组（例如，for arg in pargs:）。
4. 这次，代码打印出“1, {'c': 3, 'b': 2}”，因为1按照名称传递给a，**kargs把其他关键字参数收集到一个字典中。我们可以用任何迭代工具来步进任何额外的关键字参数字典（例如，for key in kargs:）。
5. 这里的输出是“1564”：1按照位置匹配a，5和6按照*name位置匹配b和c（6覆盖了c的默认值），并且d默认为4，因为它没有传递一个值。
6. 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。

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：递归函数、函数属性和注解、`lambda`表达式、如`map`和`filter`这样的函数式编程工具。这些都是有些高级的工具，根据你的职位分工，可能在日常的工作中不会碰到它们。然而，由于它们在某些领域中有用，有必要对它们有个基本的理解。例如，`lambda`在GUI中是很常用的。

之所以使用函数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。下一章继续这一高级话题，在这里已经学习的函数工具的基础上，进一步介绍生成器函数和表达式，并再次回顾列表解析。

函数设计概念

既然已经学习了Python中的函数的基本知识，让我们以介绍背景的几句话开始本章。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。你需要深入考虑函数的大小等概念，因为它们直接影响到代码的可用性。其中的一些属于结构分析和设计的范畴，但是，它们和其他概念一样也适用于Python代码。

我们在第17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初学者的一些通用的指导方针的复习。

- **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代码中少数醒目位置的外部的依赖关系的最好办法。

- **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，这会导致很多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比萨订单”），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。我们在第17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19-1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。就像将在第六部分看到的那样，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。

然而，通常来讲，我们应该竭力使函数和其他编程组件中的外部依赖性最小化。函数的自包含性越好，它越容易被理解、复用和修改。

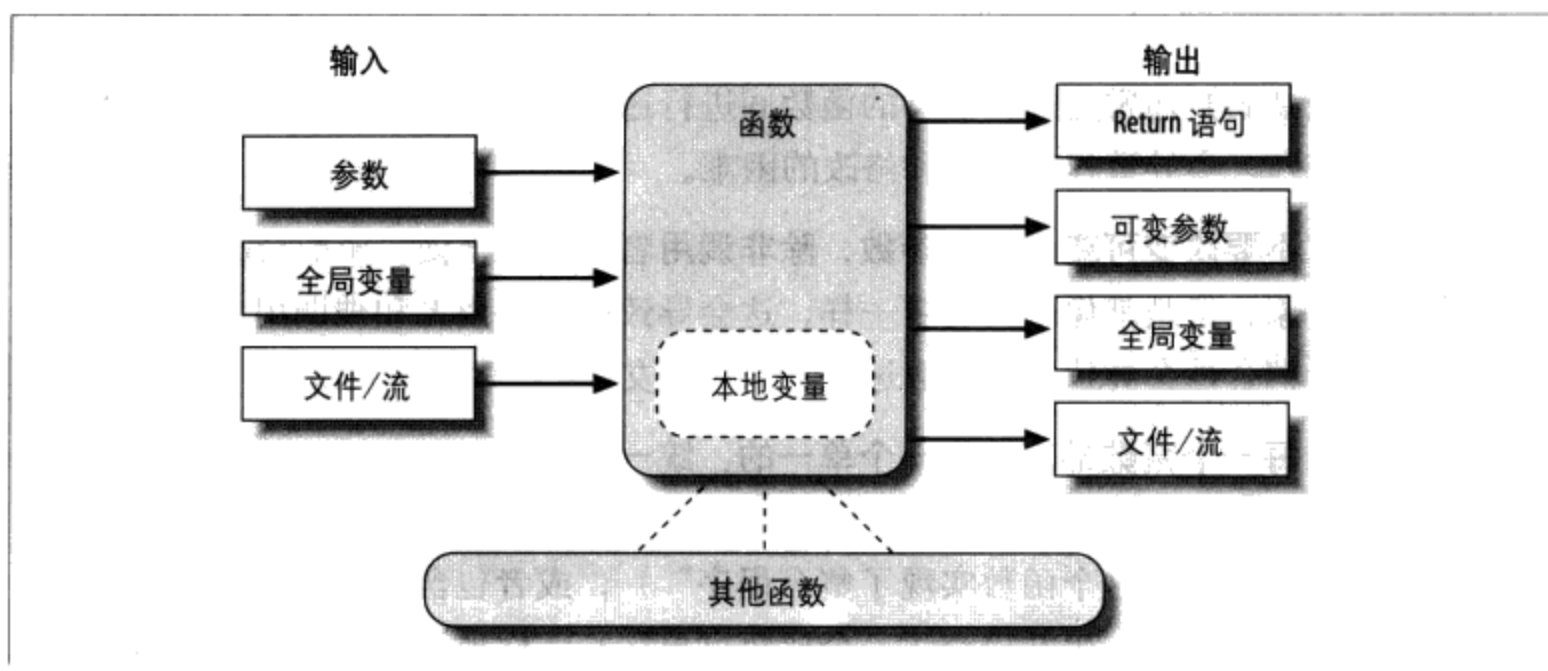


图19-1：函数执行环境。函数可以通过多种办法获得输入产生输出，尽管使用参数作为输入，return语句并配合可变参数的改变作为输出时，函数往往更容易理解和维护。在Python 3.0中，输出也可能采取存在于一个封闭的函数作用域中的声明的nonlocal名称的形式

递归函数

在第17章开始处讨论作用域规则的时候，我们简短地提及Python支持递归函数——即直接或间接地调用自身以进行循环的函数。递归是颇为高级的话题，并且它在Python中相对少见。然而，它是一项应该了解的有用的技术，因为它允许程序遍历拥有任意的、不可预知的形状的结构。递归甚至是简单循环和迭代的替换，尽管它不一定是最简单的或最高效的一种。

用递归求和

让我们来看一些例子。要对一个数字列表（或者其他序列）求和，我们可以使用内置的sum函数，或者自己编写一个更加定制化的版本。这里是用递归编写的一个定制求和函数的示例：

```
>>>def mysum(L):
...     if not L:
...         return 0
...     else:
...         return L[0] + mysum(L[1:])           # Call myself
>>>mysum([1, 2, 3, 4, 5])
15
```

在每一层，这个函数都递归地调用自己来计算列表剩余的值的和，这个和随后加到前面的一项中。当列表变为空的时候，递归循环结束并返回0。当像这样使用递归的时候，

对函数调用的每一个打开的层级，在运行时调用堆栈上都有自己的一个函数本地作用域的副本，也就是说，这意味着L在每个层级都是不同的。

如果这很难理解（并且对于新程序员来说，它常常是难以理解），尝试给函数添加一个L的打印并再次运行它，从而在每个调用层级记录下当前的列表：

```
>>>def mysum(L):
...     print(L)                # Trace recursive levels
...     if not L:                # L shorter at each level
...         return 0
...     else:
...         return L[0] + mysum(L[1:])
...
>>>mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

正如你所看到的，在每个递归层级上，要加和的列表变得越来越小，直到它变为空——递归循环结束。加和随着递归调用的展开而计算出来。

编码替代方案

有趣的是，我们也可以使用Python的三元if/else表达式（在第12章介绍过）在这里保存某些代码资产。我们也可以针对任何可加和的类型一般化（如果我们至少假设输入中的一项的话，这将会变得较容易些，就像我们在第18章最小最大值的示例中所做的那样），并且使用Python 3.0的扩展序列赋值来使得第一个/其他的解包更简单（正如第11章所介绍的）：

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])          # Use ternary expression

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Any type, assume one

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest)   # Use 3.0 ext seq assign
```

这些例子中的后两个由于空的列表而失败，但是考虑到支持+的任何对象类型的序列，而不只是数字：

```
>>>mysum([1])                # mysum([]) fails in last 2
1
```



```
>>>mysum([1, 2, 3, 4, 5])
15
>>>mysum(('s', 'p', 'a', 'm'))           # But various types now work
'spam'
>>>mysum(['spam', 'ham', 'eggs'])
'spamhameggs'
```

如果你研究这3个变体，将会发现，后两者在一个单个字符串参数上也有效（例如，`mysum('spam')`），因为字符串是一字符的字符串的序列；第三种变体在任意可迭代对象上都有效，包括打开的输入文件，但是，其他的两种不会有效，因为它们索引；并且函数头部`def mysum(first, * rest)`尽管类似于第三种变体，但根本没法工作，因为它期待单个参数，而不是一个单独的可迭代对象。

别忘了，递归是可以是直接的，就像目前为止给出的例子一样；也可以是间接的，就像下面的例子一样（一个函数调用另一个函数，后者反过来调用其调用者）。直接的效果是相同的，尽管这在每个层级有两个函数调用：

```
>>>def mysum(L):
...     if not L: return 0
...     return nonempty(L)           # Call a function that calls me
...
>>>def nonempty(L):
...     return L[0] + mysum(L[1:])   # Indirectly recursive
...
>>>mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

循环语句VS递归

尽管递归对于上一小节的求和的例子有效，但在那种环境中，它可能过于追求技巧了。实际上，递归在Python中并不像在Prolog或Lisp这样更加深奥的语言中那样常用，因为Python强调像循环这样的简单的过程式语句，循环语句通常更为自然。例如，`while`常常使得事情更为具体一些，并且它不需要定义一个支持递归调用的函数：

```
>>>L = [1, 2, 3, 4, 5]
>>>sum = 0
>>>while L:
...     sum += L[0]
...     L = L[1:]
...
>>>sum
15
```

更好的情况，`for`循环为我们自动迭代，使得递归在大多数情况下不必使用（并且，很可能，递归在内存空间和执行时间方面效率较低）：

```
>>>L = [1, 2, 3, 4, 5]
```

```
>>>sum = 0
>>>for x in L: sum += x
...
>>>sum
15
```

有了循环语句，我们不需要在调用堆栈上针对每次迭代都有一个本地作用域的副本，并且，我们避免了一般会与函数调用相关的速度成本（在第20章的时候，我们将通过一个计时器案例来学习比较这样的替代方案的执行时间）。

处理任意结构

另一方面，递归（或者对等的显式的基于堆栈的算法，我们也将在这里实现）可以要求遍历任意形状的结构。作为递归在这种环境中的应用的一个简单例子，考虑像下面这样的任务：计算一个嵌套的子列表结构中所有数字的总和：

```
[1, [2, [3, 4], 5], 6, [7, 8]]           # Arbitrarily nested sublists
```

简单的循环语句在这里不起作用，因为这不是一个线性迭代。嵌套的循环语句也不够用，因为子列表可能嵌套到任意的深度并且以任意的形式嵌套。相反，下面的代码使用递归来对应这种一般性的嵌套，以便顺序访问子列表：

```
def sumtree(L):
    tot = 0
    for x in L:                               # For each item at this level
        if not isinstance(x, list):
            tot += x                           # Add numbers directly
        else:
            tot += sumtree(x)                  # Recur for sublists
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]           # Arbitrary nesting
print(sumtree(L))                           # Prints 36

# Pathological cases

print(sumtree([1, [2, [3, [4, [5]]]]]))     # Prints 15 (right-heavy)

print(sumtree([[[[[1], 2], 3], 4], 5]))     # Prints 15 (left-heavy)
```

留意这段脚本末尾的测试案例，看看递归是如何遍历其嵌套的列表的。尽管这个例子是人为编写的，它是一类更大的程序的代表，例如，继承树和模块导入链可以展示类似的通用结构。实际上，我们在本书后面更为实用的示例中再次使用递归的这一用法：

- 在第24章的`reloadall.py`中，用来遍历导入链。
- 在第28章的`classtree.py`中，用来遍历类继承树。

- 在第30章的`lister.py`中，再次用来遍历类继承树。

尽管出于简单性和高效率的目的，对于线性迭代通常应该更喜欢使用循环语句而不是递归，我们还是会发现像后面的示例一样的不可缺少递归的情况。

此外，有时候需要意识到程序中无意的递归的潜在性。正如你将在本书后面看到，类中的一些运算符重载方法，例如`__setattr__`和`__getattr__`，如果使用不正确的话，都有潜在的可能会递归地循环。递归是一种强大的工具，但它会比预期的更好。

函数对象：属性和注解

Python函数比我们想象的更为灵活。正如我们在本书的这一部分中所看到的，Python中的函数比一个编译器的代码生成规范还要多——Python函数是俯拾皆是对象，自身全部存储在内存块中。同样，它们可以跨程序自由地传递和间接调用。它们也支持与调用根本无关的操作——属性存储和注解。

间接函数调用

由于Python函数是对象，我们可以编写通用的处理它们的程序。函数对象可以赋值给其他的名字、传递给其他函数、嵌入到数据结构、从一个函数返回给另一个函数，等等，就好像它们是简单的数字或字符串。函数对象还恰好支持一个特殊操作：它们可以由一个函数表达式后面的括号中的列表参数调用。然而，函数和其他对象一样，属于通用的领域。

我们已经在前面的示例中看到了函数的这些通用应用中的一些，但一个快速概览对于强调对象模型有帮助。例如，对于用于一条`def`语句中的名称，真的没有什么特别的：它只是当前作用域中的一个变量赋值，就好像它出现在一个`=`符号的左边。在`def`运行之后，函数名直接是一个对象的引用——我们可以自由地把这个对象赋给其他的名称并且通过任何引用调用它：

```
>>>def echo(message):           # Name echo assigned to function object
...     print(message)
...
>>>echo('Direct call')         # Call object through original name
Direct call

>>>x = echo                     # Now x references the function too
>>>x('Indirect call!')         # Call object through name by adding ()
Indirect call!
```

由于参数通过赋值对象来传递，这就像是把函数作为参数传递给其他函数一样容易。随后，被调用者可能通过把参数添加到括号中来调用传入的函数：

```
>>>def indirect(func, arg):
...     func(arg)                                # Call the passed-in object by adding ()
...
>>>indirect(echo, 'Argument call!')             # Pass the function to another function
Argument call!
```

我们甚至可以把函数对象的内容填入到数据结构中，就好像它们是整数或字符串一样。例如，下面的程序把函数两次嵌套到一个元组列表中，作为一种动作表。由于像这样的Python复合类型可以包含任意类型的对象，这里也没有什么特殊的：

```
>>>schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>>for (func, arg) in schedule:
...     func(arg)                                # Call functions embedded in containers
...
Spam!
Ham!
```

这段代码只是遍历schedule列表，每次遍历的时候使用一个参数来调用echo函数（注意for循环头部的元组解包赋值，我们在第13章中介绍过）。正如我们在第17章的示例中所见到的，函数也可以创建并返回以便之后使用：

```
>>>def make(label):
...     def echo(message):
...         print(label + ':' + message)
...     return echo
...
>>>F = make('Spam')                             # Label in enclosing scope is retained
>>>F('Ham!')                                     # Call the function that make returned
Spam:Ham!
>>>F('Eggs!')
Spam:Eggs!
```

Python的通用对象模式和无须类型声明使得该编程语言有了令人惊讶的灵活性。

函数内省

由于函数是对象，我们可以用常规的对象工具来处理函数。实际上，函数比我们所预料的更灵活。例如，一旦我们创建一个函数，可以像往常一样调用它：

```
>>>def func(a):
...     b = 'spam'
...     return b * a
...
>>>func(8)
'spamspamspamspamspamspamspam'
```

但是，调用表达式只是定义来在函数对象上工作的一个操作。我们也可以通用地检查它们的属性（如下代码在Python 3.0中运行，但是Python 2.6中的结果是类似的）：

```
>>>func.__name__
'func'
>>>dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

内省工具允许我们探索实现细节——例如，函数已经附加了代码对象，代码对象提供了函数的本地变量和参数等方面的细节：

```
>>>func.__code__
<code object func at 0x0257C9B0, file "<stdin>", line 1>

>>>dir(func.__code__)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
...more omitted...
'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab',
'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>>func.__code__.co_varnames
('a', 'b')
>>>func.__code__.co_argcount
1
```

工具编写者可以利用这些信息来管理函数（实际上，我们还将在第38章中，在装饰器中实现函数参数的验证）。

函数属性

函数对象不仅限于前面小节中列出的系统定义的属性。正如我们在第17章中所学习到的，也可能向函数附加任意的用户定义的属性：

```
>>>func
<function func at 0x0257C738>
>>>func.count = 0
>>>func.count += 1
>>>func.count
1
>>>func.handles = 'Button-Press'
>>>func.handles
'Button-Press'
>>>dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
'__str__', '__subclasshook__', 'count', 'handles']
```

正如我们在本章所见到的，这样的属性可以用来直接把状态信息附加到函数对象，而不必使用全局、非本地和类等其他技术。和非本地不同，这样的属性可以在函数自身的任何地方访问。从某种意义上讲，这也是模拟其他语言中的“静态本地变量”的一种

方式——这种变量的名称对于一个函数来说是本地的，但是，其值在函数退出后仍然保留。属性与对象相关而不是与作用域相关，但直接效果是类似的。

Python 3.0中的函数注解

在Python 3.0中（但不包括Python 2.6），也可以给函数对象附加注解信息——与函数的参数和结果相关的任意的用户定义的数据。Python为声明注解提供了特殊的语法，但是，它自身不做任何事情；注解完全是可选的，并且，出现的时候只是直接附加到函数对象的`__annotations__`属性以供其他用户使用。

我们在上一章中介绍了Python 3.0的keyword-only参数，注解则进一步使函数头部语法通用化。考虑如下的不带注解的函数，它编写为带有3个参数并且返回一个结果：

```
>>>def func(a, b, c):
...     return a + b + c
...
>>>func(1, 2, 3)
6
```

从语法上讲，函数注解编写在`def`头部行，就像与参数和返回值相关的任意表达式一样。对于参数，它们出现在紧随参数名之后的冒号之后；对于返回值，它们编写于紧跟在参数列表之后的一个`->`之后。例如，这段代码，注解了前面函数的3个参数及其返回值：

```
>>>def func(a: 'spam', b: (1, 10), c: float) -> int:
...     return a + b + c
...
>>>func(1, 2, 3)
6
```

调用一个注解过的函数，像以前一样，不过，当注解出现的时候，Python将它们收集到字典中并且将它们附加给函数对象自身。参数名变成键，如果编写了返回值注解的话，它存储在键“`return`”下，而注解键的值则赋给了注解表达式的结果：

```
>>>func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

由于注解只是附加到一个Python对象的Python对象，注解可以直接处理。下面的例子只是注解了3个参数中的两个，并且通用地遍历附加的注解：

```
>>>def func(a: 'spam', b, c: 99):
...     return a + b + c
...
>>>func(1, 2, 3)
6
```

```
>>>func.__annotations__
{'a': 'spam', 'c': 99}

>>>for arg in func.__annotations__:
...     print(arg, '=>', func.__annotations__[arg])
...
a => spam
c => 99
```

这里有两点值得注意。首先，如果编写了注解的话，仍然可以对参数使用默认值——注解（及其:字符）出现在默认值（及其=字符）之前。例如，下面的a: 'spam' = 4意味着参数a的默认值是4，并且用字符串'spam'注解它：

```
>>>def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
...     return a + b + c
...
>>>func(1, 2, 3)
6
>>>func()
15                                # 4 + 5 + 6 (all defaults)
>>>func(1, c=10)
16                                # 1 + 5 + 10 (keywords work normally)
>>>func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

还要注意前面例子中的空格都是可选的——你可以在函数头部的各部分之间使用空格，也可以不用，但省略它们对某些读者来说可能会提高代码的可读性：

```
>>>def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:
...     return a + b + c
...
>>>func(1, 2)
9                                # 1 + 2 + 6
>>>func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

注解是Python 3.0中的新功能，并且其一些潜在的用途还没有介绍。很容易想象，注释可以用作参数类型或值的特定限制，并且较大的API可能使用这一功能作为注册函数接口信息的方式。实际上，我们将会在第38章中看到一个潜在的应用，那里，我们将看到注解作为函数装饰器参数（这是一个更为通用的概念，其中，信息编写于函数头部之外，并且由此不仅限于一种用途）的一种替代方法。和Python自身一样，注解是一种功能随着你的想象来变化的工具。

最后，注意，注解只在def语句中有效，在lambda表达式中无效，因为lambda的语法已经限制了它所定义的函数工具。这把我们带入到下一个主题。

匿名函数： lambda

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。^{注1}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有时叫做匿名（也就是没有函数名）的函数的原因。实际上，它们常常以一种行内进行函数定义的形式使用，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（与一个def头部内用括号括起来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
```

由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有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的参数中。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。

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。除了继承了数学的含糊性，lambda比想象的要容易使用。

```
>>>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。

```
>>>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17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器，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,          # Inline function definition
      lambda x: x ** 3,
      lambda x: x ** 4]        # A list of 3 callable functions

for f in L:
    print(f(2))                # Prints 4, 8, 16

print(L[0](3))                 # Prints 9
```

当需要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是不会有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。对等的def代码可能需要在想要使用的环境之外有临时性函数名称和函数定义。

```
def f1(x): return x ** 2
def f2(x): return x ** 3      # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]              # Reference by name

for f in L:
    print(f(2))                # Prints 4, 8, 16

print(L[0](3))                 # Prints 9
```

实际上，我们可以用Python中的字典或者其他的数据结构来构建更多种类的行为表，从而做同样的事情。下面是以交互提示模式给出的另一个例子：

```
>>>key = 'got'
>>>{'already': (lambda: 2 + 2),
... 'got':      (lambda: 2 * 4),
... 'one':      (lambda: 2 ** 6)}[key]()
8
```

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。与在第12章中向你展示的if语句的扩展用法相比，这样编写代码可以使字典成为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]()
64

```

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变量名，这些变量名也许会与这个文件中的其他变量名发生冲突（也可能不会，但总是有可能）。

`lambda`在函数调用参数里作为行内临时函数的定义，并且该函数在程序中不在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`，直接编写`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）。类似地，要在一个`lambda`中嵌套逻辑，可以使用第12章曾经介绍过的`if/else`三元表达式，或者对等的但需要些技巧的`and/or`组合。正如我们前面所了解到的，如下语句：

```

if a:
    b
else:
    c

```

能够由以下的概括等效的表达式来模拟：

```

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（我们在上一章见过的工具，将会在这章及下一章进行复习）这样的工具来实现。

```
>>>import sys
>>>showall = lambda x: list(map(sys.stdout.write, x))    # Use list in 3.0

>>>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。这就是为什么lambda仅限于表达式。如果你有更复杂的代码要编写，可使用def，lambda针对较小的一段内联代码。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17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)
>>>act
<function <lambda> at 0x00A16A88>
>>>act(2)    # Call what action returned
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
```

这里嵌套的`lambda`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`。

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（这个模块在Python 2.6中叫做Tkinter）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，假设tkinter在你的计算机上可用的话（它在Windows和其他操作系统上是默认打开的）。

```
import sys
from tkinter import Button, mainloop # Tkinter in 2.6
x = Button(
    text = 'Press me',
    command=(lambda:sys.stdout.write('Spam\n')))
x.pack()
mainloop()
```

这里，回调处理器是通过传递一个用`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`直到事件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，它们也使回调处理器变得更简单易用，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六部分介绍）。

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.onPress("spam")))
    def onPress(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`lambda`中。

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`counters`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
...
>>>list(map(inc, counters))           # Collect results
[11, 12, 13, 14]
```

在第13章和第14章中曾简短地介绍过`map`，它对一个可迭代对象中的项应用一个内置函数。这里，我们将会传入一个用户定义的函数来对它进行充分的利用，从而可以对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都调用了`inc`函数，并将所有的返回值收集到一个新的列表中。别忘了，`map`在Python 3.0中是一个可迭代对象，因此，在这里，一个列表调用用来迫使它生成所有的结果以显示，这在Python 2.6中不是必需的。

由于`map`期待传入一个函数，它恰好是`lambda`通常出现的地方之一：

```
>>>list(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，你就能够自己编写一个一般的映射工具了。

```
>>>def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
```


假设函数`inc`仍然像前面出现时那样，我们可以用内置函数或我们自己的对等形式将其映射到一个序列：

```
>>>list(map(inc, [1, 2, 3]))           # Built-in is an iterator
[11, 12, 13]
>>>mymap(inc, [1, 2, 3])              # Ours builds a list (see generators)
[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)                           # 3**4
81
>>>list(map(pow, [1, 2, 3], [2, 3, 4])) # 1**2, 2**3, 3**4
[1, 8, 81]
```

对于多个序列，`map`期待一个N参数的函数用于N序列。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14章中学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。

函数式编程工具：filter和reduce

在Python内置函数中，`map`函数是用来进行函数式编程的这类工具中最简单的内置函数代表：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元素（`filter`），以及对每对元素都应用函数并运行到最后结果（`reduce`）。由于`range`和`filter`都返回可迭代对象，在Python 3.0中，它们需要`list`调用来显示其所有结果。例如，下面这个`filter`的调用实现了从一个序列中挑选出大于0的元素。

```
>>>list(range(-5, 5))                  # An iterator in 3.0
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>>list(filter((lambda x: x > 0), range(-5, 5))) # An iterator in 3.0
[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`在Python 2.6中只是一个简单的内置函数，但是在Python 3.0中则位于`functools`模块中，要更复杂一些。它接受一个迭代器来处理，但是，它自身不是一个迭代器，它返回一个单个的结果。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。

```
>>>from functools import reduce          # Import in 3.0, not in 2.6

>>>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列表中下一个的元素，传给列出的`lambda`函数。默认，序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
```

编写自己的`reduce`版本实际上相当直接。如下的函数模拟内置函数的大多数行为，并且帮助说明其一般性的运作：

```
>>>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])
15
>>>myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

这个内置的`reduce`还允许一个可选的第三个参数放置于序列的各项之前，从而当序列为空时充当一个默认的结果，但是，我们把这一扩展留作一个建议的练习。

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（要了解关于这一模块的更多内容，请参阅Python的库手册）。

```
>>>import operator, functools
>>>functools.reduce(operator.add, [2, 4, 6])    # Function-based +
12
>>>functools.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、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。

本章小结

本章介绍了和函数相关的高级概念：递归函数、函数注解、lambda表达式函数，常用函数工具如map、filter、reduce，以及通用的函数设计思想。下一章继续高级话题，介绍生成器，并再次回顾迭代器及列表解析——这些是既与函数式编程相关又与循环语句相关的工具。在继续学习之前，请看本章的练习，以确保已经掌握了这里所介绍的概念。

本章习题

1. lambda表达式和def语句有什么关系？
2. 使用lambda的要点是什么？
3. 比较和对比map、filter和reduce。
4. 什么是函数注解，如何使用它们？
5. 什么是递归函数，如何使用它们？
6. 编写函数的通用设计规则是什么？

习题解答

1. 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。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。
2. lambda允许“内联”小单元可执行代码，推迟其执行，并且以默认参数和封闭作用域变量的形式为其提供状态。使用lambda不是必需的，我们总可以编写一条def来

替代它，并且用名称来引用该函数。`lambda`很方便，以嵌套小段的推迟的代码，这些代码不可能在程序的某处用到。它们通常出现在GUI这样的基于回调的程序中，并且它们与`map`和`filter`这些期待一个处理函数的函数工具密切相关。

3. 这3个内置函数都对一个序列（可迭代）对象以及集合结果中的各项应用另一个函数。`map`把每一项传递给函数并收集结果，`filter`收集那些函数返回一个`True`值的项，并且`reduce`通过对一个累加器和后续项应用函数来计算一个单个的值。和其他两个函数不同，`reduce`在Python 3.0的`functools`模块中可用，而不是在内置作用域中可用。
4. 函数注解在Python 3.0及其以后的版本中可用，并且是函数的参数及其结果的语法上的修饰，它会收集到分配给函数的`__annotations__`属性的一个字典中。Python在这些注解上没有放置语义含义，而是直接将其包装，以供其他工具潜在地使用。
5. 递归函数调用本身可以直接地或间接地进行，从而实现循环。它们可以用来遍历任意形状的结构，但是，也可以用来进行一般性迭代（尽管后一种角色用循环语句来编写往往更简单和高效）。
6. 函数通常应该较小，尽可能自包含，拥有单一的、统一的用途，并且与输入参数和返回值等其他部分通信。如果期待修改的话，它们可以使用可变的参数来与结果通信，并且一些类型的程序暗含其他的通信机制。

迭代和解析，第二部分

本章继续高级函数主题，再次回顾第14章所介绍的解析和迭代概念。由于列表解析既与上一章的函数工具（如`map`和`filter`）相关，又与循环相关，我们将在这里再次复习它。我们还将回顾迭代，以便学习生成器函数及其相关的生成器表达式——这是用户定义的、按需产生结果的方式。

Python中的迭代也包括用户定义的类，但是，我们将推迟到第六部分介绍这一点，也就是当我们学习运算符重载的时候再介绍。然而，这是我们最后一次介绍内置的迭代工具，我们将概括目前为止所遇到的各种工具，并且计算它们中的一些的相对性能。最后，由于这是本部分的最后一章，我们将以常见的一组“问题”和练习来结束本章，从而帮助你开始利用已经学习的概念来编写代码。

回顾列表解析：函数式编程工具

在上一章中，我们学习了`map`和`filter`这样的函数式编程工具，它们将操作映射到序列和集合结果中。由于这是Python编程中的一种常见任务，Python最终产生了一种新的表达式——列表解析，它甚至比前面学习的工具更灵活。简而言之，列表解析把任意一个表达式而不是一个函数应用于一个迭代对象中的元素。同样，它可以是更为通用的工具。

我们在第14章学习循环语句的时候介绍了列表解析。但是因为它们与`map`和`filter`这样的函数式编程工具相关，所以我们将会在本章回顾这一话题的内容。从技术上讲，这个特性并没有与函数绑定在一起。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。

列表解析与map

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整数编码转换为字符）：

```
>>>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))
...
>>>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 = list(map(ord, 'spam'))           # Apply function to sequence
>>>res
[115, 112, 97, 109]
```

尽管如此，我们能够通过列表解析表达式得到相同的结果——map把一个函数映射遍一个序列，列表解析把一个表达式映射遍一个序列：

```
>>>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（为了提醒你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0~9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：

```
>>>list(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，我们在第14章介绍过，可以在for之后编写一个if分支，用来增加选择逻辑。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0~4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]

>>>list(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0~9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。

```
>>>list( 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 iterable1 [if condition1]
    for target2 in iterable2 [if condition2] ...
    for targetN in iterableN [if conditionN] ]
```

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。

```
>>>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0~4的偶数与从0~4的奇数的组合。其中if分句过滤出了每个序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，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。

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，如下代码使用嵌套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这样的任务。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，然后通过嵌套的列表解析来构建具有相同值的一个嵌套列表结构。

```
>>>[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]  
  
>>>[[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。它等同于如下的基于语句的代码。

```
>>>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1}。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。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14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，可以使用列表解析或map调用通过一个步骤从所有的行中将它们都去掉（map的结果在Python 3.0中是可迭代的，因此，我们必须通过list来运行它们以一次性看到其所有结果）：

```
>>>[line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']

>>>[line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']

>>>list(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。

注1： 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。

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]

>>>list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

第一种方法使用元组赋值来解包列表中的行元组，第二种方法使用索引。在Python 2.6（但不包含Python 3.0，参见第18章关于Python 2.6参数解包的说明），map也可以对其参数使用元组解包：

```
# 2.6 only
>>>list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

更多关于Python的数据库API请参考其他的书籍和资源。

除了运行函数和表达式之间的区别，Python 3.0中的map和列表解析的最大区别是：map是一个迭代器，根据需求产生结果；为了同样地实现内存节省，列表解析必须编码为生成器表达式（本章的主题之一）。

重访迭代器：生成器

如今Python对延迟提供更多的支持——它提供了工具在需要的时候才产生结果，而不是立即产生结果。特别地，有两种语言结构尽可能地延迟结果创建。

- 生成器函数：编写为常规的def语句，但是使用yield语句一次返回一个结果，在每个结果之间挂起和继续它们的状态。
- 生成器表达式类似于上一小节的列表解析，但是，它们返回按需产生结果的一个对象，而不是构建一个结果列表。

由于二者都不会一次性构建一个列表，它们节省了内存空间，并且允许计算时间分散到

各个结果请求。我们将会看到，这二者最终都通过实现我们在第14章所介绍的迭代协议来执行它们延迟结果的魔术。

生成器函数：yield VS return

在本书的这一部分中，我们已经学习了编写接收输入参数并立即送回单个结果的常规函数。然而，也有可能来编写可以送回一个值并随后从其退出的地方继续的函数。这样的函数叫做生成器函数，因为它们随着时间产生值的一个序列。

一般来说，生成器函数和常规函数一样，并且，实际上也是用常规的def语句编写的。然而，当创建时，它们自动实现迭代协议，以便可以出现在迭代背景中。我们在第14章学习了迭代器，这里，我们将再次回顾它们，看看它们是如何与生成器相关的。

状态挂起

和返回一个值并退出的常规函数不同，生成器函数自动在生成值的时刻挂起并继续函数的执行。因此，它们对于提前计算整个一系列值以及在类中手动保存和恢复状态都很有用。由于生成器函数在挂起时保存的状态包含它们的整个本地作用域，当函数恢复时，它们的本地变量保持了信息并且使其可用。

生成器函数和常规函数之间的主要的代码不同之处在于，生成器*yields*一个值，而不是返回一个值。yield语句挂起该函数并向调用者发送回一个值，但是，保留足够的状态以使得函数能够从它离开的地方继续。当继续时，函数在上一个yield返回后立即继续执行。从函数的角度来看，这允许其代码随着时间产生一系列的值，而不是一次计算它们并在诸如列表的内容中送回它们。

迭代协议整合

要真正地理解生成器函数，我们需要知道，它们与Python中的迭代协议的概念密切相关。正如我们已经看到的，可迭代的对象定义了一个__next__方法，它要么返回迭代中的下一项，或者引发一个特殊的StopIteration异常来终止迭代。一个对象的迭代器用iter内置函数接收。

如果支持该协议的话，Python的for循环以及其他的迭代背景，使用这种迭代协议来遍历一个序列或值生成器；如果不支持，迭代返回去重复索引序列。

要支持这一协议，函数包含一条yield语句，该语句特别编译为生成器。当调用时，它们返回一个迭代器对象，该对象支持用一个名为__next__的自动创建的方法来继续执行的接口。生成器函数也可能有一条return语句，总是在def语句块的末尾，直接终止值的生成。从技术上讲，可以在任何常规函数退出执行之后，引发一个StopIteration异

常来现实。从调用者的角度来看，生成器的__next__方法继续函数并且运行到下一个yield结果返回或引发一个StopIteration异常。

直接效果就是生成器函数，编写为包含yield语句的def语句，自动地支持迭代协议，并且由此可能用在任何迭代环境中以随着时间并根据需要产生结果。

注意：正如第14章所提到的，在Python 2.6和更早的版本中，可迭代的对象定义了一个名为next的方法而不是__next__方法。这包括我们在这里使用的生成器对象。在Python 3.0中，这个方法重命名为__next__。next内置函数作为一个方便的、可移植的工具提供：next(I)等同于Python 3.0中的I.__next__()和Python 2.6中的I.next()。在Python 2.6之前，程序直接调用I.next()而不是手动地迭代。

生成器函数应用

为了讲清楚基础知识，请看如下代码，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。

```
>>>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, end=' : ')             # 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（我们在第14章介绍过），也就是说，生成器对象有一个__next__方法，它可以开始这个函数，或者从它上次yield值后的地

方恢复，并且在得到一系列的值的最后一个时，产生`StopIteration`异常。为了方便起见，`next(X)`内置函数为我们调用一个对象的`X.__next__()`方法：

```
>>>next(x)           # Same as x.__next__() in 3.0
0
>>>next(x)           # Use x.next() or next() in 2.6
1
>>>next(x)
4
>>>next(x)
9
>>>next(x)
Traceback (most recent call last):
...more text omitted...
StopIteration
```

正如我们在第14章所学习过的，`for`循环（以及其他的迭代环境）以同样的方式与生成器一起工作：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, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
```

对于这样的例子，我们还能够使用`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, end=' : ')
...
0 : 1 : 4 : 9 : 16 :

>>>for x in map((lambda n: n ** 2), range(5)):
...     print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
```

尽管如此，生成器在内存使用 and 性能方面都更好。它们允许函数避免临时再做所有的工作，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时，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六部分介绍）。有了生成器，函数变量就能进行自动的保存和恢复^{注2}。

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)]。表达式必须包含在括号中，除非它是赋值语句右边的唯一一项。例如，X = yield Y没问题，就如同X = (yield Y) + 42。

当使用这一额外的协议时，值可以通过调用G.send(value)发送给一个生成器G。之后恢复生成器的代码，并且生成器中的yield表达式返回了为了发送而传入的值。如果提前调用了正常的G.__next__()方法（或者其对等的next(G)），yield返回None。例如：

```
>>>def gen():
...     for i in range(10):
...         X = yield i
...         print(X)
...
>>>G = gen()
>>>next(G)                                # Must call next() first, to start generator
0
>>>G.send(77)                             # Advance, and send value to yield expression
77
1
>>>G.send(88)
88
2
>>>next(G)                                # next() and X.__next__() send None
None
3
```

例如，用send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

注2： 有趣的是，生成器函数也是某些“穷人的”多线程设备。它们通过将操作划分为在yield之间运行的步骤，从而在函数的工作之间插入其调用者的工作。然而，生成器不是线程，程序在一个单线程控制中，显式地导入或导出函数。从某种意义上讲，线程更为通用（生成者可以真正独立地运行，并且把结果发布到一个队列），但是，生成器可能更容易编写。参见本书第17章中的第2个脚注对Python多线程工具的简单介绍。注意，由于控制在yield和下一个调用处显式地导向，生成器也是不能回溯的，但是，它与coroutine关系更密切，后者是超出本书讨论范围的正式概念。

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。

注意，尽管Python 3.0提供了一个`next(X)`方便的内置函数，它会调用一个对象的`X.__next__`方法，但是，其他的生成器方法，例如`send`，必须直接作为生成器对象的方法来调用（例如，`G.send(X)`）。这么做是有意义的，你要知道，这些额外的方法只是在内置的生成器对象上实现，而`__next__`方法应用于所有的可迭代对象（包括内置类型和用户定义的类）。

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种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>
```

实际上，至少在一个函数的基础上，编写一个列表解析基本上等同于：在一个`list`内置调用中包含一个生成器表达式以迫使其一次生成列表中所有的结果。

```
>>>list(x ** 2 for x in range(4))       # List comprehension equivalence
[0, 1, 4, 9]
```

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。

```
>>>G = (x ** 2 for x in range(4))
>>>next(G)
0
>>>next(G)
1
>>>next(G)
4
>>>next(G)
9
>>>next(G)
Traceback (most recent call last):
...more text omitted...
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14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，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。

```
>>>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
>>>list( 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。关于性能的更权威的评价，必须等到我们在本章稍后编写计时脚本的时候给出。

生成器函数 VS 生成器表达式

有趣的是，同样的迭代往往可以用一个生成器函数或一个生成器表达式编写。例如，如下的生成器表达式，把一个字符串中的每个字母重复4次。

```
>>>G = (c * 4 for c in 'SPAM')           # Generator expression
>>>list(G)                               # Force generator to produce all results
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

等价的生成器函数需要略微多一些的代码，但是，作为一个多语句的函数，如果需要的话，它将能够编写更多的逻辑并使用更多的状态信息。

```

>>>def timesfour(S):                # Generator function
...     for c in S:
...         yield c * 4
...
>>>G = timesfour('spam')
>>>list(G)                          # Iterate automatically
['ssss', 'pppp', 'aaaa', 'mmmm']

```

表达式和函数支持自动迭代和手动迭代——前面的列表自动调用迭代，如下的迭代手动进行。

```

>>>G = (c * 4 for c in 'SPAM')
>>>I = iter(G)                      # Iterate manually
>>>next(I)
'SSSS'
>>>next(I)
'pppp'
>>>G = timesfour('spam')
>>>I = iter(G)
>>>next(I)
'ssss'
>>>next(I)
'pppp'

```

注意，我们使得这里的新的生成器再次迭代，正如下一小节所介绍的，生成器是单次迭代器。

生成器是单迭代器对象

生产器函数和生成器表达式自身都是迭代器，并由此只支持一次活跃迭代——不像一些内置类型，我们无法有在结果集中位于不同位置的多个迭代器。例如，使用前面小节的生成器表达式，一个生成器的迭代器是生成器自身（实际上，在一个生成器上调用`iter`没有任何效果）。

```

>>>G = (c * 4 for c in 'SPAM')
>>>iter(G) is G                    # My iterator is myself: G has __next__
True

```

如果你手动地使用多个迭代器来迭代结果流，它们将会指向相同的位置。

```

>>>G = (c * 4 for c in 'SPAM')    # Make a new generator
>>>I1 = iter(G)                   # Iterate manually
>>>next(I1)
'SSSS'
>>>next(I1)
'pppp'
>>>I2 = iter(G)                   # Second iterator at same position!
>>>next(I2)
'AAAA'

```

此外，一旦任何迭代器运行到完成，所有的迭代器都将用尽，我们必须产生一个新的生成器以再次开始。

```
>>>list(I1)                # Collect the rest of I1's items
['MMMM']
>>>next(I2)                # Other iterators exhausted too
StopIteration

>>>I3 = iter(G)            # Ditto for new iterators
>>>next(I3)
StopIteration

>>>I3 = iter(c * 4 for c in 'SPAM')    # New generator to start over
>>>next(I3)
'SSSS'
```

对于生成器函数来说，也是如此，如下的基于语句的`def`等价形式只支持一个活跃的生成器并且在一次迭代之后用尽。

```
>>>def timesfour(S):
...     for c in S:
...         yield c * 4
...
>>>G = timesfour('spam')    # Generator functions work the same way
>>>iter(G) is G
True
>>>I1, I2 = iter(G), iter(G)
>>>next(I1)
'ssss'
>>>next(I1)
'pppp'
>>>next(I2)                # I2 at same position as I1
'aaaa'
```

这与某些内置类型的行为不同，它们支持多个迭代器并且在一个活动迭代器中传递并反映它们的原处修改。

```
>>>L = [1, 2, 3, 4]
>>>I1, I2 = iter(L), iter(L)
>>>next(I1)
1
>>>next(I1)
2
>>>next(I2)                # Lists support multiple iterators
1
>>>del L[2:]                # Changes reflected in iterators
>>>next(I1)
StopIteration
```

当我们在本书第六部分开始编写基于类的迭代器时，我们将看到，由我们来决定想要为自己的对象支持多少个迭代器。

用迭代工具模拟zip和map

要说明应用迭代工具的能力，让我们来看一些高级用例。一旦你了解了列表解析、生成器和其他的迭代工具，就知道模拟众多的Python的函数式内置工具既直接又很有益。

例如，我们已经看到了内置的zip和map函数如何组合可迭代对象和映射函数。使用多个序列参数，map以与zip配对元素相同的方式，把函数映射到取自每个序列的元素。

```
>>>S1 = 'abc'
>>>S2 = 'xyz123'
>>>list(zip(S1, S2))                                # zip pairs items from iterables
[('a', 'x'), ('b', 'y'), ('c', 'z')]

# zip pairs items, truncates at shortest

>>>list(zip([-2, -1, 0, 1, 2]))                      # Single sequence: 1-ary tuples
[(-2,), (-1,), (0,), (1,), (2,)]

>>>list(zip([1, 2, 3], [2, 3, 4, 5]))                # N sequences: N-ary tuples
[(1, 2), (2, 3), (3, 4)]

# map passes paired items to a function, truncates

>>>list(map(abs, [-2, -1, 0, 1, 2]))                 # Single sequence: 1-ary function
[2, 1, 0, 1, 2]

>>>list(map(pow, [1, 2, 3], [2, 3, 4, 5]))           # N sequences: N-ary function
[1, 8, 81]
```

尽管它们用于不同的目的，如果你研究这些示例足够长的时间，可能会注意到zip结果和执行map的函数参数之间的一种关系，下面的例子可以说明这种关系。

编写自己的map(func, ...)

尽管map和zip内置函数快速而方便，总是可以在自己的代码中模拟它们。例如，在上一章中，我们看到一个函数针对单个的序列参数来模拟map内置函数。针对多个序列的时候，也并不会费太多工夫就可以像内置函数那样操作。

```
# map(func, seqs...) workalike with zip

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

这个版本很大程度上依赖于特殊的*args参数传递语法。它收集多个序列（实际上，是可迭代对象）参数，将其作为zip参数解包以便组合，然后成对的zip结果解包作为参

数以便传入到函数。也就是说，我们在使用这样的一个事实，`zip`是`map`中的一个基本的嵌套操作。最后的测试代码对一个序列和两个序列都应用了这个函数，以产生这一输入（我们可以用内置的`map`得到同样的输出）。

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

然而，实际上，前面的版本展示了经典的列表解析模式，在一个`for`循环中构建操作结果的一个列表。我们可以更精简地编写自己的`map`，作为单行列表解析的对等体。

```
# Using a list comprehension

def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

当这段代码运行的时候，结果与前面相同，但是，这段代码更加精炼并且可能运行的更快（关于性能的更多讨论，参见本章后面的“对迭代的各种方法进行计时”一节）。之前的`mymap`版本一次性构建结果列表，并且对于较大的列表来说，这可能浪费内存。既然我们知道了生成器函数和表达式，重新编码这两种替代方案来根据需求产生结果是很容易的。

```
# Using generators: yield and (...)

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        yield func(*args)

def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))
```

这些版本产生同样的结果，但是返回设计用来支持迭代协议的生成器。第一个版本每次`yield`一个结果，第二个版本返回一个生成器表达式的结果来做同样的事情。如果我们把它们包含到一个`list`调用中迫使它们一次生成所有的值，它们会产生同样的结果。

```
print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

这里并没有做什么实际工作，直到`list`调用迫使生成器运行，通过激活迭代协议而进行。生成器由这些函数自身返回，也由它们所使用的Python 3.0式的`zip`内置函数返回，根据需要产生结果。

编写自己的zip(...)和map(None, ...)

当然，目前给出的示例中的很多魔力在于，它们使用`zip`内置函数来配对来自多个序列的参数。我们也注意到，我们的`map`近似版确实是模拟了Python 3.0的`map`的行为，它们从最短的序列的长度处截断，并且，当长度不同的时候，它们不支持补充结果的思路，这就像Python 2.X中带有有一个`None`参数的`map`所做的一样：

```
C:\misc>c:\python26\python
>>>map(None, [1, 2, 3], [2, 3, 4, 5])
[(1, 2), (2, 3), (3, 4), (None, 5)]
>>>map(None, 'abc', 'xyz123')
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

使用迭代工具，我们可以编写近似版来模拟截断的`zip`和Python 2.6的补充的`map`，这些其实在代码上近乎是相同的：

```
# zip(seqs...) and 2.6 map(None, seqs...) workalikes
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

这里编写的函数可以在任何类型的可迭代对象上运行，因为它们通过`list`内置函数来运行自己的参数以迫使结果生成（例如，文件像参数一样工作，此外，序列像字符串一样）。注意这里的`all`和`any`内置函数的使用，如果一个可迭代对象中的所有或任何元素为`True`（或者对等的为非空），它们分别返回`True`。当列表中的任何或所有参数在删除后变成了空，这些内置函数将用来停止循环。

还要注意Python 3.0的`keyword-only`参数`pad`，和Python 2.6的`map`不同，我们的版本将允许指定任何补充的对象（如果你使用Python 2.6，使用一个`**kwargs`形式来支持这一选项，参见第18章了解详细内容）。当这些函数运行的时候，打印出如下的结果——一个`zip`和两个补充的`map`。

```
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

```

[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

```

这些函数不能够用于列表解析转换，因为它们的循环太具体了。然而，和前面一样，既然我们的`zip`和`map`近似版构建并返回列表，用`yield`将它们转换为生成器以便它们每个都是每次返回结果中的一项，这还是很容易做到的。结果和前面的相同，但是，我们需要再次使用`list`来迫使该生成器产生其值以供显示。

```

# Using generators: yield

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))
print(list(mymapPad(S1, S2)))
print(list(mymapPad(S1, S2, pad=99)))

```

最后，这里是我们的`zip`和`map`模拟器的替代实现——下面的版本不是使用`pop`方法从列表中删除参数，而是通过计算最小和最大参数长度来完成其工作。有了这些长度，很容易编写嵌套的列表解析来遍历参数索引范围。

```

# Alternate implementation with lengths

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))

```

由于这些代码使用`len`和索引，它们假设参数是序列或类似的，而不是任意的可迭代对象。这里，外围的解析遍历参数索引范围，内部的解析（传递到元组）遍历传入的序列以并列地提取参数。当它们运行时，结果和前面相同。

更有趣的是，生成器和迭代器似乎在这个例子中泛滥。传递给`min`和`max`的参数是生成器

表达式，它在嵌套的解析开始迭代之前运行完成。此外，嵌套的列表解析使用了两个层级的延迟计算——Python 3.0的range内置函数是一个可迭代对象，就像生成器表达式参数对元组。

实际上，这里没有产生结果，直到列表解析的方括号要求放入到结果列表中的值——它们迫使解析和生成器运行。为了把这些函数自身转换为生成器而不是列表构建器，使用圆括号而不是方括号。zip的例子如下所示。

```
# Using generators: (...)

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))

print(list(myzip(S1, S2)))
```

在这个例子中，它用一个list调用来激活生成器和迭代器以产生它们自己的结果。自己体验这些来了解更多内容。进一步开发替代的编码留作一个建议练习（参见下面的“为什么你会留意：单次迭代”来了解这样的一个选项）。

为什么你会留意：单次迭代

在第14章，我们看到了一些内置函数（如map）如何只支持一个单个的便利，并且在发生之后为空，我提过会给出一个示例展示这在实际中是如何变得微妙而重要的。现在，已经学习了关于迭代话题的许多内容，我可以兑现这个承诺了。考虑本章的zip模拟示例的更优替代代码，该示例从Python手册中的一个例子改编而来。

```
def myzip(*args):
    iters = map(iter, args)
    while iters:
        res = [next(i) for i in iters]
        yield tuple(res)
```

由于这段代码使用iter和next，它对任何类型的可迭代对象都有效。注意，当这个参数的迭代器之一用尽时，没有任何理由捕获由这个解析内的next(it)来引发的StopIteration——允许它传递会终止这个生成器函数，并且与一条return语句具有相同的效果。如果至少传递了一个参数的话，while iters:对于循环来说足够了，并且，避免了无限循环（列表解析将总是返回一个空的列表）：

这段代码在Python 2.6中也工作得很好，如下所示：

```
>>>list(myzip('abc', 'lmnop'))
[('a', 'l'), ('b', 'm'), ('c', 'n')]
```

但是，在Python 3.0下，它陷入了一个无限循环中并失效，因为Python 3.0的map返回一个单次可迭代对象，而不是像Python 2.6中那样的一个列表。在Python 3.0中，只要我们在循环中运行了一次列表解析，iters将会永远为空（并且res将会是[]）。为了使其在Python 3.0下正常工作，我们需要使用list内置函数来创建一个支持多次迭代的对象：

```
def myzip(*args):
    iters = list(map(iter, args))
    ...rest as is...
```

自己运行并跟踪其操作。这里要记住的是：在Python 3.0中把map调用放入到list调用中不仅是为了显示。

内置类型和类中的值生成

最后，尽管我们在本节中关注自己编写值生成器，别忘了，很多内置的类型以类似的方式工作——正如我们在第14章中看到的一样，例如，字典拥有在每次迭代中产生键的迭代器。

```
>>>D = {'a':1, 'b':2, 'c':3}
>>>x = iter(D)
>>>next(x)
'a'
>>>next(x)
'c'
```

和手动编写的生成器所产生的值一样，字典键也可以手动迭代，或者使用包括for循环、map调用、列表解析和我们在第14章介绍的很多其他环境等在内的自动迭代工具。

```
>>>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, end='')
...
Tis but
a flesh wound.
```

尽管内置类型迭代器绑定到了一个特定类型的值生成，概念与我们使用表达式和函数编

写的生成器是类似的。像for循环这样的迭代环境接受任何的可迭代对象，不管是用户定义的还是内置的。

尽管这超出了本章的讨论范围，还是可能用遵守迭代协议的类来实现任意的用户定义的生成器对象。这样的类定义了一个特别的`__iter__`方法，它由内置的`iter`函数调用，将返回一个对象，该对象有一个`__next__`方法，该方法由`next`内置函数调用（一个`__getitem__`索引方法作为迭代的退而求其次的选项也是可以的）。

从这样一个类创建的实例对象，看做是可迭代的，并且可以用在for循环和所有其他的迭代环境中。然而，有了类，我们可以访问比其他生成器构造所能提供的更丰富的逻辑和数据结构选项。迭代器的内容不会真正结束，直到我们了解到它如何映射到类。现在，我们必须推迟到第29章学习基于类的迭代器的时候再结束这一话题。

Python 3.0解析语法概括

我们已经在本章中关注过列表解析和生成器，但是，别忘了，还有两种在Python 3.0中可用的解析表达式形式：集合解析和字典解析。我们在第5章和第8章曾遇到过这两种形式，但是，有了解析和生成器的知识，现在我们应该能够全面地理解这些Python 3.0扩展了。

- 对于集合，新的常量形式`{1, 3, 2}`等同于`set([1, 3, 2])`，并且新的集合解析语法`{f(x) for x in S if P(x)}`就像是生成器表达式`set(f(x) for x in S if P(x))`，其中`f(x)`是一个任意的表达式。
- 对于字典，新的字典解析语法`{key: val for (key, val) in zip(keys, vals)}`像`dict(zip(keys, vals))`形式一样工作，并且`{x:f(x) for x in items}`像生成器表达式`dict((x, f(x)) for x in items)`一样工作。

这里是Python 3.0中的所有解析替代方式的总结。最后两种是新的，并且在Python 2.6中不可用。

```
>>>[x * x for x in range(10)]           # List comprehension: builds list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]    # like list(generator expr)

>>>(x * x for x in range(10))           # Generator expression: produces items
<generator object at 0x009E7328>        # Parens are often optional

>>>{x * x for x in range(10)}           # Set comprehension, new in 3.0
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}    # {x, y} is a set in 3.0 too

>>>{x: x * x for x in range(10)}        # Dictionary comprehension, new in 3.0
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

解析集合和字典解析

从某种意义上讲，集合解析和字典解析只是把生成器表达式传递给类型名的语法糖。因此，二者都接受任何的可迭代对象，一个生成器在这里工作得很好。

```
>>>{x * x for x in range(10)}           # Comprehension
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>>set(x * x for x in range(10))        # Generator and type name
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>>{x: x * x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>>dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

然而，对于列表解析来说，我们总是可以用手动代码来构建结果对象。这里是最后两个解析的基于语句的等价形式。

```
>>>res = set()
>>>for x in range(10):                  # Set comprehension equivalent
...     res.add(x * x)
...
>>>res
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>>res = {}
>>>for x in range(10):                  # Dict comprehension equivalent
...     res[x] = x * x
...
>>>res
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

注意，尽管这两种形式都接受迭代器，它们没有根据需要产生结果的概念——两种形式都是一次构建所有对象。如果你想要根据需求产生键和值，生成器表达式更合适。

```
>>>G = ((x, x * x) for x in range(10))
>>>next(G)
(0, 0)
>>>next(G)
(1, 1)
```

针对集合和字典的扩展的解析语法

和列表解析及生成器表达式一样，集合和字典解析都支持嵌套相关的if子句从结果中过滤掉元素——如下的代码收集一个范围内的每个元素的平方（例如，元素被2除没有余数）。

```
>>>[x * x for x in range(10) if x % 2 == 0]   # Lists are ordered
[0, 4, 16, 36, 64]
>>>{x * x for x in range(10) if x % 2 == 0}   # But sets are not
```



```
{0, 16, 4, 64, 36}
>>>{x: x * x for x in range(10) if x % 2 == 0}           # Neither are dict keys
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}
```

嵌套的for循环也有效，尽管两种类型的对象无序的和无副本的特性可能会使得结果看上去缺乏直接性。

```
>>>[x + y for x in [1, 2, 3] for y in [4, 5, 6]]         # Lists keep duplicates
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>>{x + y for x in [1, 2, 3] for y in [4, 5, 6]}        # But sets do not
{8, 9, 5, 6, 7}
>>>{x: y for x in [1, 2, 3] for y in [4, 5, 6]}        # Neither do dict keys
{1: 6, 2: 6, 3: 6}
```

和列表解析一样，集合解析和字典解析也可以在任何类型的可迭代对象上迭代——列表、字符串、文件、范围以及支持迭代协议的任何其他类型。

```
>>>{x + y for x in 'ab' for y in 'cd'}
{'bd', 'ac', 'ad', 'bc'}

>>>{x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}
{'bd': (98, 100), 'ac': (97, 99), 'ad': (97, 100), 'bc': (98, 99)}

>>>{k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'sausagesausage', 'spamspace'}

>>>{k.upper(): k * 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}
{'SAUSAGE': 'sausagesausage', 'SPAM': 'spamspace'}
```

要了解更多细节，自己体验这些工具。它们的性能可能比生成器或for循环替代方案好，也可能没有它们好，我们将明确地记录其性能。

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。让我们简要地看一下这个实例，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。

对模块计时

幸运的是，Python使得对代码计时变得很容易。要看看迭代选项是如何叠加起来的，让我们从编写到一个模块文件中的简单但通用的计时器工具函数开始，从而使其可以用于各类程序中。

```
# File mytimer.py

import time
reps = 1000
repslist = range(reps)

def timer(func, *pargs, **kargs):
    start = time.clock()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = time.clock() - start
    return (elapsed, ret)
```

实际上，这个模块通过获取开始时间、调用函数固定的次数并且用开始时间减去停止时间，从而对使用任何位置和关键字参数调用任意函数进行计时。注意以下几点：

- Python的time模块允许访问当前时间，精度随着每个平台而有所不同。在Windows上，这个调用号称能够达到微妙的精度，已经相当准确了。
- range调用放到了计时循环之外，因此，它的构建成本不会计算到Python 2.6的计时函数中。在Python 3.0的range是一个迭代器，因此这个步骤是不需要的（但无伤大雅）。
- reps计数是一个全局变量，如果需要的话，导入者可以修改它：mytimer.reps = N。

当这些完成后，所有调用的总的使用时间在一个元组中返回，还带有被计时的函数的最终返回值，以便调用者可以验证其操作。

从一个更大的角度来看，由于这个函数编写到一个模块文件中，在我们想要导入它的任何地方，它都成为了一个广为有用的工具。在本书的下一部分中，我们还将学习有关模块和导入的更多内容，但是，通过这些代码，我们已经看到了足够多的基础知识——直接导入该模块并调用函数来使用这个文件的计时器（如果需要回顾的话，参见第3章对模块属性的介绍）。

计时脚本

现在，要计时迭代工具的速度，运行如下的脚本，它使用我们刚刚编写来统计已经学习过的各种列表构建技术的相对速度的计时器模块。

```

# File timeseqs.py

import sys, mytimer                                # Import timer function
reps = 10000                                       # Hoist range out in 2.6
repslist = range(reps)

def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repslist]

def mapCall():
    return list(map(abs, repslist))                # Use list in 3.0 only

def genExpr():
    return list(abs(x) for x in repslist)          # list forces results

def genFunc():
    def gen():
        for x in repslist:
            yield abs(x)
    return list(gen())

print(sys.version)
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    elapsed, result = mytimer.timer(test)
    print ('-' * 33)
    print ('%-9s: %.5f => [%s...%s]' %
          (test.__name__, elapsed, result[0], result[-1]))

```

这段脚本测试了五种构建结果列表的替代方法，并且，每种方法都执行了一千万次级别的步骤，也就是说，五个测试中的每一个都构建了拥有10000个元素的列表1000次。

注意，我们必须通过内置的`list`调用来运行生成器表达式和函数结果，从而迫使它们产生其所有的值；如果没有这么做，可能会得到并没有真正工作的生成器。在（仅在）Python 3.0中，我们必须对`map`结果做同样的事情，因为它现在也是一个可迭代对象。还要注意，底部的代码如何遍历4个函数对象的一个元组并打印出每一个的`__name__`。正如我们所看到的，这是一个内置的属性，它给出函数的名称。

计时结果

当上一小节中的脚本在Python 3.0下运行时，我在自己的Windows Vista笔记本上得到了如下的结果——`map`比列表解析略微快一点，但二者都比`for`循环要快很多，并且，生成器表达式和函数速度居中。

```
C:\misc>c:\python30\python timeseqs.py
```

```

3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
-----
forLoop : 2.64441 => [0...9999]
-----
listComp : 1.60110 => [0...9999]
-----
mapCall : 1.41977 => [0...9999]
-----
genExpr : 2.21758 => [0...9999]
-----
genFunc : 2.18696 => [0...9999]

```

如果我们研究这些代码及其输出足够长的时间，将会注意到，生成器表达式比列表解析运行得慢。尽管把一个生成器表达式包装到一个`list`调用中，会使得其功能等同于一个带有方括号的列表解析，两种表达式的内部实现看上去有所不同（尽管我们已经实际地对生成器测试的列表调用计时）。

```

return [abs(x) for x in range(size)]           # 1.6 seconds
return list(abs(x) for x in range(size))       # 2.2 seconds: differs internally

```

有趣的是，当我在Windows XP的Python 2.5运行它的时候，结果也是相对类似的——列表解析几乎比对等的`for`循环语句快一倍，并且当映射`abs`（求绝对值）这样的内置函数的时候，`map`比列表解析略快。我没有测试生成器函数，并且输出格式也并不是太复杂。

```

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

```

这里列出的实际的Python 2.5测试时间比前面给出的输出要慢两倍，可能是因为我最近的测试中使用了一款较快的笔记本，而不是因为Python 3.0的改进。实际上，如果从`map`测试中移除`list`调用以避免两次创建结果列表的话，这段脚本的所有Python 2.6的结果都比在同样机器上的Python 3.0要快一些（请自行测试以验证）。

如果我们修改这段脚本，在每次迭代上执行一个真正的操作（如加法），而不是调用`abs`这样的小的内置函数，看看会发生什么（如下代码中省略的部分与前面相同）。

```

# File timeseqs.py
...
...
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res

```

```

def listComp():
    return [x + 10 for x in repslist]

def mapCall():
    return list(map((lambda x: x + 10), repslist))    # list in 3.0 only

def genExpr():
    return list(x + 10 for x in repslist)            # list in 2.6 + 3.0

def genFunc():
    def gen():
        for x in repslist:
            yield x + 10
    return list(gen())
...

```

如果需要针对map调用来调用一个用户定义的函数，会使它比for循环语句慢，尽管循环语句的版本的代码更多。在Python 3.0上如下所示。

```

C:\misc>c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
-----
forLoop : 2.60754 => [10...10009]
-----
listComp : 1.57585 => [10...10009]
-----
mapCall : 3.10276 => [10...10009]
-----
genExpr : 1.96482 => [10...10009]
-----
genFunc : 1.95340 => [10...10009]

```

在一款较慢的机器上，Python 2.5的结果再一次与前面的版本类似，但是，由于测试机器不同，要慢一倍：

```

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

```

由于解释器优化是如此内部化的一个问题，像这样对Python代码进行性能分析是一件非常需要技术的事情。事实上不可能猜测哪种方法会执行的最好，最好的办法是在自己的计算机上、用自己的Python版本，对自己的代码计时。在这种情况下，我们可以肯定地说的是，在这个Python版本下，在map调用中使用一个用户定义的函数至少会因为两种因素中的一种而执行较慢，并且列表解析对于这一测试运行最快。

正如我们前面提到的，性能应该不是你编写Python代码时首要关心的问题——要优化Python代码，你应该做的第一件事情就是不要优化Python代码！首先为了可读性和简单

性而编写代码，然后，如果需要的话并且只有在需要的时候，再优化。如果对于你的程序需要处理的数据集合来说，五种替代方案中的任何一种足够快，这将是很好的事情；如果是这样的话，程序的清晰性应该是首要的目标。

计时模块替代方案

前面小节介绍的计时模块是有效的，但是，它在多个方面还有些简单：

- 它总是使用`time.clock`调用计时代码。尽管该选项在Windows上是最好的，`time.time`在某些UNIX平台上可能提供更好的解析。
- 调整重复的次数需要修改模块级别的全局变量——如果要使用`timer`函数并且有多个导入者共享的话，这是不太理想的安排。
- 此外，计时器必须通过运行测试函数很多次才能工作。要考虑随机的系统载入的波动，在所有的测试中选择最好的时间，而不是总的时间，可能会更好。

如下的替代实现了一种更为高级的计时器模块，它解决了前面所有这3点：根据平台选择一个计时器调用，允许重复计数作为一个名为`_reps`的关键字参数传入，并且提供N中最好的一个替代计时函数。

```
# File mytimer.py (2.6 and 3.0)

"""
timer(spam, 1, 2, a=3, b=4, _reps=1000) calls and times spam(1, 2, a=3)
_reps times, and returns total time for all runs, with final result;

best(spam, 1, 2, a=3, b=4, _reps=50) runs best-of-N timer to filter out
any system load variation, and returns best time among _reps tests
"""

import time, sys
if sys.platform[:3] == 'win':
    timefunc = time.clock           # Use time.clock on Windows
else:
    timefunc = time.time           # Better resolution on some Unix platforms

def trace(*args): pass            # Or: print args

def timer(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000) # Passed-in or default reps
    trace(func, pargs, kargs, _reps)
    repslist = range(_reps)        # Hoist range out for 2.6 lists
    start = timefunc()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)

def best(func, *pargs, **kargs):
```

```

_reps = kargs.pop('_reps', 50)
best = 2 ** 32
for i in range(_reps):
    (time, ret) = timer(func, *pargs, _reps=1, **kargs)
    if time < best: best = time
return (best, ret)

```

位于文件顶部的这个模块的文档字符串描述了模块的目标用途。它使用字典的`pop`操作，从用于测试函数的参数中删除`_reps`参数并为其提供一个默认值，并且，如果你将其`trace`函数修改为`print`的话，它会在开发过程中跟踪参数。要在Python 3.0或Python 2.6上测试这个新的计时器模块，把计时脚本做如下修改（这个版本中的测试函数对每个测试使用`x+1`操作，其中省略的部分和前面小节的代码相同）。

```

# File timeseqs.py

import sys, mytimer
reps = 10000
repslist = range(reps)

def forLoop(): ...
def listComp(): ...
def mapCall(): ...
def genExpr(): ...
def genFunc(): ...

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (forLoop, listComp, mapCall, genExpr, genFunc):
        elapsed, result = tester(test)
        print ('-' * 35)
        print ('%-9s: %.5f => [%s...%s]' %
              (test.__name__, elapsed, result[0], result[-1]))

```

在Python 3.0下运行的时候，计时结果基本与前面相同，并且对于N个之和与N中最好的计时技术来说相对都是相同的——多次运行测试似乎做了很好的工作来过滤掉系统载入波动而采取最好的情况，但是，当测试一个长时间运行的函数的时候，N中最好的方案可能更好。在我的机器上的结果如下所示。

```

C:\misc>c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----
forLoop : 2.35371  => [10...10009]
-----
listComp : 1.29640 => [10...10009]
-----
mapCall : 3.16556  => [10...10009]

```



```

-----
genExpr : 1.97440 => [10...10009]
-----
genFunc : 1.95072 => [10...10009]
<best>
-----
forLoop : 0.00193 => [10...10009]
-----
listComp : 0.00124 => [10...10009]
-----
mapCall : 0.00268 => [10...10009]
-----
genExpr : 0.00164 => [10...10009]
-----
genFunc : 0.00165 => [10...10009]

```

当然，这里N中最佳的计数器报告的时间是很小的，但是，如果你的程序在较大的数据集上迭代很多次的话，它可能变得很显著。至少在相对性能方面，列表解析在大多数情况下表现最好。当使用内置函数时，map表现更好。

在Python 3.0中使用keyword-only参数

在这里，我们可以使用Python 3.0的keyword-only参数来简化计时器模块代码。正如我们在第19章中所学到的，keyword-only参数对于_reps这样的配置选项来说是理想的选择。必须将它们写在函数头部的一个*之后和一个**之前，并在一个函数调用中，它们必须由关键字传递，并且如果使用的话，出现在**之前。这里是前面模块的一个基于keyword-only参数的替代。尽管简单，它只能在Python 3.0下而不能在Python 2.6下编译和运行。

```

# File mytimer.py (3.X only)

"""
Use 3.0 keyword-only default arguments, instead of ** and dict pops.
No need to hoist range() out of test in 3.0: a generator, not a list
"""

import time, sys
trace = lambda *args: None # or print
timefunc = time.clock if sys.platform == 'win32' else time.time

def timer(func, *pargs, _reps=1000, **kargs):
    trace(func, pargs, kargs, _reps)
    start = timefunc()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)

def best(func, *pargs, _reps=50, **kargs):
    best = 2 ** 32
    for i in range(_reps):

```

```

        (time, ret) = timer(func, *pargs, _reps=1, **kargs)
        if time < best: best = time
    return (best, ret)

```

不考虑从运行到运行之间的可忽略的测试时间差异的话，这个版本与前面的版本以同样的方式使用并产生相同的结果。

```

C:\misc>c:\python30\python timeseqs.py
...same results as before...

```

实际上，我们也可以从交互提示模式来测试模块的这个版本，从而完全独立于序列计时器脚本——它是一个通用目的的工具。

```

C:\misc>c:\python30\python
>>>from mytimer import timer, best
>>>
>>>def power(X, Y): return X ** Y           # Test function
...
>>>timer(power, 2, 32)                      # Total time, last result
(0.002625403507987747, 4294967296)
>>>timer(power, 2, 32, _reps=1000000)       # Override default reps
(1.1822605247314932, 4294967296)
>>>timer(power, 2, 100000)[0]                # 2 ** 100,000 tot time @1,000 reps
2.2496919999608878

>>>best(power, 2, 32)                       # Best time, last result
(5.58730229727189e-06, 4294967296)
>>>best(power, 2, 100000)[0]                 # 2 ** 100,000 best time
0.0019937589833460834
>>>best(power, 2, 100000, _reps=500)[0]     # Override default reps
0.0019845399345541637

```

对于像这个交互会话中一样的小函数，计时器的代码的成本可能像那些被计时函数一样显著，因此，我们不应该太绝对地取计时器结果（我们的计时不仅仅是这里的 $X ** Y$ ）。计时器的结果可以帮助我们判断代码替代方案的相对速度，并且可能对于如下这样较长时间运行的操作更有意义——计算2的一百万次方比前面的 $2**100\ 000$ 要长一个量级（10倍）。

```

>>>timer(power, 2, 1000000, _reps=1)[0]     # 2 ** 1,000,000: total time
0.088112804839710179
>>>timer(power, 2, 1000000, _reps=10)[0]
0.40922470593329763

>>>best(power, 2, 1000000, _reps=1)[0]      # 2 ** 1,000,000: best time
0.086550036387279761
>>>best(power, 2, 1000000, _reps=10)[0]     # 10 is sometimes as good as 50
0.029616752967200455
>>>best(power, 2, 1000000, _reps=50)[0]     # Best resolution
0.029486918030102061

```

尽管这里计算的时间很小，往往在计算能力不同的程序里，差别是显著的。

参见第19章了解关于Python 3.0中的keyword-only参数的更多内容，它们可以简化像这样的配置工具的编码，但是不能与Python 2.X向后兼容。例如，如果想要比较2.X和3.X的速度，或者支持使用任意一种Python版本的程序员，较早的版本可能是更好的选择。如果你使用Python 2.6，上面会话的运行将与前面的计时器模块版本相同。

其他建议

要更深入地了解，尝试修改这些模块所使用的重复计数，或者尝试Python的标准库中的替代的timeit模块，它自动对代码计时，支持命令行使用模式，并且解决了一些特定于平台的问题。Python的手册介绍了其用法。

你可能还想要看看profile标准库模块，以了解代码探查工具的完整源代码——我们将在第35章介绍大项目的开发工具的时候学习它。一般地，在重新编写代码并计时之前，你应该探查代码以孤立瓶颈。

使用Python 2.6和Python 3.0中新的str.format方法而不是%格式化表达式（未来可能潜在的废弃）来体验，这可能也是有用的，通过对计时脚本的格式化打印行做如下的修改。

```
print('<s>' % tester.__name__)          # From expression
print('<{0}>'.format(tester.__name__))    # To method call
print ('%-9s: %.5f => [%s...%s]' %
      (test.__name__, elapsed, result[0], result[-1]))
print('{0:<9}: {1:.5f} => [{2}...{3}]'.format(
      test.__name__, elapsed, result[0], result[-1]))
```

你可以自行判断这些技术之间的差别。

如果你有信心，也可以尝试修改或模拟计时脚本来测量本章所介绍的Python 3.0的集合和字典解析及其对等的for循环的速度。在Python程序中，它们比构建结果列表用得少，我们把这一任务留作建议的练习。

最后，准备好我们在这里所编写的计时模块以便将来参考——在本章末尾的一个求数值平方根的练习中，我们将再次重新使用它来度量性能。如果你对于进一步探讨这一主题有兴趣，我们还将针对交互的计时字典解析和for循环来体验这一技术。

函数陷阱

既然已经接近函数介绍的尾声，让我们来看一些常见的陷阱。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新闻组中入门者最为常见的陷阱之一。

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 for "import X", "def X" ...
>>>selector()
...error text omitted...
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读入并编译这段代码。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做是本地变量。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17章。

在Python最近的版本中，已经针对这种情况发布了更为专用的“unbound local”错误消息来改进这一问题，如前面的示例列表所示（它用来直接引起一个通用的名称错误）；然而，这个陷阱仍然普遍出现。

默认和可变对象

默认参数是在def语句运行时评估并保存的，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。

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。

如今，以一种较少令人混淆的方式来实现可变默认值效果的另一种方式，是使用我们在第19章讨论过的函数属性：

```
>>>def saver():
...     saver.x.append(1)
...     print(saver.x)
...
>>>saver.x = []
>>>saver()
[1]
>>>saver()
[1, 1]
>>>saver()
[1, 1, 1]
```

该函数的名称对于函数自身来说是全局的，但是，它不需要声明，因为它在函数内部是不会直接修改的。这并不是总是以完全相同的方式使用，但是，当这样编写代码的时候，一个对象到函数的附加总是更加明确（并且肯定更容易理解）。

没有return语句的函数

在Python函数中，`return`（以及`yield`）语句是可选的。当一个函数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，函数将会退出。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。

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15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17章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17章中关于这个话题的更多内容）。

本章小结

本章介绍了内置解析和迭代工具。它在函数工具中介绍了列表解析，并且把生成器函数和表达式作为另外的一种迭代协议工具介绍。最后，我们还度量了迭代替代方案的性能，并且我们最后回顾了与函数相关的常见错误以帮助你避开陷阱。

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。当函数出现在类方法的环境中的时候，在本书的这一部分中所学到的一切将会适用。

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。

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. `map`调用和`list comprehension`有什么关系？比较并对比两者。

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一个元素，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语法上相同，但是，它们由Python特别编译，以便在调用的时候返回一个可迭代的对象。
4. 当有了`yield`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. `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。

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，来计算任意数量的参数的和，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18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所做的是，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，但是确保在Python 3.0中将其放入到一个`list`调用中）。
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6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18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。

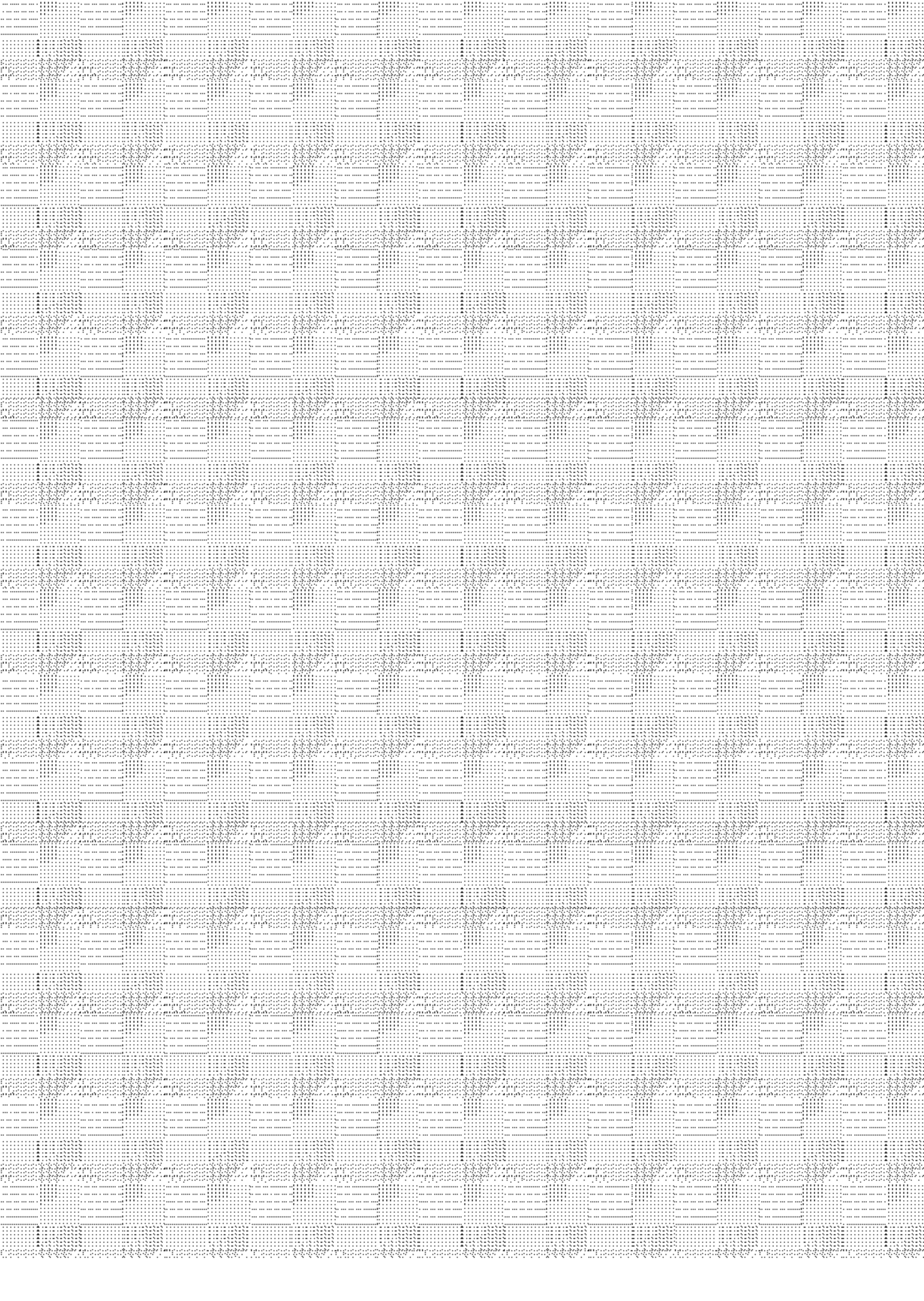
```
x = y // 2                # For some y > 1
while x > 1:
    if y % x == 0:        # Remainder
        print(y, 'has factor', x)
        break            # Skip else
    x -= 1
else:                     # Normal exit
    print(y, 'is prime')
```

把这个代码封装成模块文件中可重用的函数（*y*应该是一个传入参数），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/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？
10. 计时工具。在第5章中，我们看到了计算平方根的3种方式：math.sqrt(X)，X **.5和pow(X, .5)。如果你的程序运行很多这样的运算，它们的相对性能就变得重要。要看看哪一种最快，重新使用我们在本章中编写的timerseqs.py脚本来对这些工具中的每一种计时。对最好的函数使用mytimer.py计时器模块（我们可以使用Python 3.0的关键字变体，或者Python 2.6和Python 3.0的版本）。你可能想要把测试代码重新包装到这个脚本中以便更好地重用——例如，通过向一个通用的测试函

数传递一个测试函数元组（对于这个练习，复制并修改的方式是最好的）。3个平方根工具中的哪一个在你的机器上与一般的Python中运行的最快？最后，当你交互地计时字典解析和for循环的速度的时候，会是怎样的情况？



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。

`imp.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。第五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模块的细节：你将会学到`reload`、`__name__`和`__all__`属性、封装`import`、相对导入语法等。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了被导入的模块对象的属性。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程序架构

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程序的架构：这种架构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，我们也将探索Python模块、导入以及对象属性的一些核心概念。

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（又称为脚本）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21-1是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。

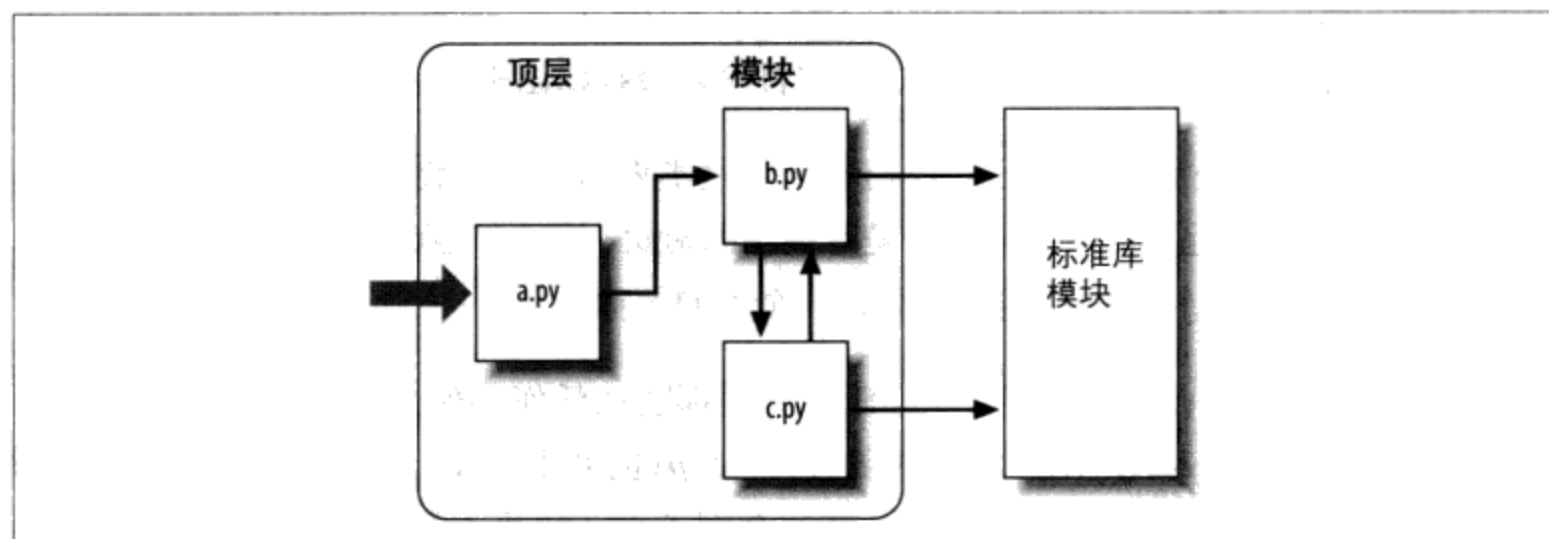


图21-1：Python的程序架构。一个程序是一个模块的系统。它有一个顶层脚本文件（启动后可运行程序）以及多个模块文件（用来导入工具库）。脚本和模块都是包含了Python语句的文本文件，尽管在模块中的语句通常都是创建之后使用的对象。Python的标准库提供了一系列的预先编写好的模块

例如，图21-1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多个的值来运行。

```
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（简单地认为是变量名）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23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编写的任何程序得以使用。例如，编写图21-1中的程序后，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21-1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 and 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。从技术上讲，Python把载入的模块存储到一个名为`sys.modules`的表中，并在一次导入操作的开始检查该表。如果模块不存在，将会启动一个三个步骤的过程。

1. 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`a.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。路径和后缀是刻意省略掉的，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。

2. 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，如果发现字节码文件比源代码文件旧（例如，如果你修改过源文件），就会在程序运行时自动重新生成字节代码。另一方面，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（这意味着你可以把一个程序只作为字节码文件发布，而避免发送源代码）。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。

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提高之后导入的速度。

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23章讨论的包导入，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。

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24章关于特定的`__name__`属性以及`__main__`的讨论。

3. 运行

`import`操作的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。

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2}。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。

模块搜索路径

正如前面所提到的，通常对程序员来说，导入过程的最重要的部分是最早的部分，也就是定位要导入的文件（搜索部分）。因为我们需要告诉Python到何处去找到要导入的文件，我们需要知道如何输入其搜索路径以扩展它。

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。

注2：正如前面所介绍的，Python已经导入的模块保存在一个内置的`sys.modules`字典中，以便可以记录哪些已经导入了。实际上，如果想要看看已经导入了哪些模块，可以导入`sys`并打印`list(sys.modules.keys())`。关于这一内部表的更多用法可参见第24章。

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。

主目录

Python首先会在主目录内搜索导入的文件。这一入口的含义与你如何运行代码相关。当你运行一个程序的时候，这个入口是包含程序的顶层脚本文件的目录。当在交互模式下工作时，这一入口就是你当前工作的目录。

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。另一方面，由于这个目录是先搜索的，其文件也将覆盖路径上的其他目录中具有同样名称的模块。如果你需要在自己的程序中使用库模块的话，小心不要以这种方式意外地隐藏库模块。

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。一旦你开始编写大量程序时，你可能想要设置PYTHONPATH变量，但是刚开始编写时，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之中或包含到路径文件（接下来介绍）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，但它们提供了PYTHONPATH设置的一种替代方案。

简而言之，当内含目录名称的文本文件放在适当目录中时，也可以概括地扮演与PYTHONPATH环境变量设置相同的角色。例如，如果你在运行Windows和Python 3.0，一个名为`myconfig.pth`的文件可以放在Python安装目录的顶层（`C:\Python30`）或者在标准库所在位置的`sitepackages`子目录中（`C:\Python30\Lib\sitepackages`），来扩展模块搜索路径。在类似Unix的系统中，文件可能位于`usr/local/lib/python3.0/site-packages`或`/usr/local/lib/site-python`中。

当存在目录的时候，Python会把文件每行所罗列的目录从头至尾加到模块搜索路径列表的最后。实际上，Python将收集它所找到的所有路径文件中的目录名，并且过滤掉任何重复的和不存在的目录。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。此外，某些用户文本文件可能比环境设置更容易编码。

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。这个模块允许配置Python库和路径文件的位置，并且其文档描述了一般的期待位置。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。路径文件作为第三方库经常使用，它通常在Python的`site-packages`目录安装一个路径文件，从而不需要用户设置（本章随后将介绍Python的`distutils`安装系统，自动化了很多安装步骤）。

配置搜索路径

所有这些的直接效果是，搜索路径的PYTHONPATH和路径文件部分允许我们调整导入查找文件的地方。设置环境变量的方法以及存储路径文件的位置，随着每种平台而变化。例如，在Windows上，我们可能使用控制面板的系统图标来把PYTHONPATH设置为分号隔开的一串目录：

```
c:\pycode\utilities;d:\pycode\package1
```

或者，可以创建一个名为`C:\Python30\pydirs.pth`的文本文件，其内容如下所示：

```
c:\pycode\utilities
d:\pycode\package1
```

这些设置在其他平台上也是类似的，但是细节可能有很大变化，无法在本章中一一介绍。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。

搜索路径的变动

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。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。更多内容请参考第3章有关从命令行启动程序。^{注3}

想知道你的Python在平台上配置的模块搜索路径，可以查看`sys.path`，这是下一小节的主题。

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，自动将顶级文件的主目录（或者指定当前工作目录的一个空字符串）、任何PYTHONPATH目录、已经创建的任何`.pth`文件路径的内容，以及标准库目录合并。结果是Python在每次导入一个新文件的时候查找的目录名的字符串的列表。

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：如果在列表中看不到设置值，就需要重新检查你的设置。例如，如下是我的模块搜索路径在Windows的Python 3.0中的样子，其中我的PYTHONPATH设置为C:\users并列出了C:\users\mark的一个C:\Python30\mypath.py路径文件。前面的空字符串表示当前路径，并且我的两个设置都合并了进去（其余的部分是标准库目录和文件）：

注3： 参考第23章对Python 3.0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。默认情况下，在Python 3.0中，一个包的目录不会被导入自动搜索，除非包自身中的文件使用了相对导入。

```
>>>import sys
>>>sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', 'c:\\Python30\\DLLs',
'c:\\Python30\\lib', 'c:\\Python30\\lib\\plat-win', 'c:\\Python30',
'C:\\Users\\Mark', 'c:\\Python30\\lib\\site-packages']
```

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。

- 源代码文件`b.py`。
- 字节码文件`b.pyc`。
- 目录`b`，包导入（在第23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。

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。

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。

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接在一起。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。

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。

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接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17章的相关内容）。

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，只不过是使用文本编辑器创建的Python程序代码文件而已。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，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，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>`。

正像上一章所提到的那样，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。这类模块称为扩展模块，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`）。

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。

注意：在Python 3.0中，这里所描述的from ...*语句形式只能用于一个模块文件的顶部，不能用于一个函数中。Python 2.6允许它用于一个函数中，但会给出一个警告。实际上，它用于函数中的情况是非常少见的，当出现的时候，可能是Python为了在函数运行之前静态地检查变量。

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，直到执行程序时，Python执行到这些语句，才会进行解析。换句话说，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`。

```
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18-1（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17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

名。`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：

```
from module import name1, name2          # Copy these two names out (only)
```

与下面这些语句是等效的：

```
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秘和模糊（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最后的“模块陷阱”再深入探讨这些话题（参考第24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
func()
# This overwrites the one we got from M
# 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
M.func()
N.func()
# Get the whole modules, not their names
# We can call both names now
# 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。如果你这么做，`import`允许你避免名字冲突。

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，赋值的变量名会存储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17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17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`。

```
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 0x026D3BB8>

>>>module2.klass
<class 'module2.klass'>

```

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。我们会在第六部分讨论类，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（别忘了在Python 3.0中将其包含到一个list调用中——它是一个视图对象）：

```

>>>list(module2.__dict__.keys())
['name', '__builtins__', '__file__', '__package__', 'sys', 'klass', 'func',
 '__name__', '__doc__']

```

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17章学过的作用域法则有什么关系呢？其实，二者没有关——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。

在第六部分中，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。

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，除非明确地进行了导入。

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。

```

X = 2
import mod3

print(X, end=' ')                        # My global X
print(mod3.X)                           # mod3's X

```

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。

`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, end=' ')           # My global X
print(mod2.X, end=' ')     # 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。
- 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。长时间运行的服务器可以以这种方式更新自己。

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/连接步骤：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。

注意：版本差异提示：在Python 2.6中，`reload`作为一个内置函数使用。在Python 3.0中，它已经移入了`imp`标准库模块中——在Python 3.0中叫做`imp.reload`。这直接意味着，需要一条额外的`import`语句或`from`语句来载入该工具（仅在Python 3.0中）。使用Python 2.6的读者可以在本书的示例中忽略这些导入，或者总是使用它们——Python 2.6在其`imp`模块中也有一个`reload`，以便更容易地迁移到Python 3.0。无论如何封装，重载都一样地工作。

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。
- `reload`在Python 3.0中位于模块之中，并且必须导入自己。

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
...
from imp import reload        # Get reload itself (in 3.0)
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。

- **`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
>>>from imp import reload
>>>reload(changer)                  # Forces new code to load/run
<module 'changer' from 'changer.py'>
>>>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。我们还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。

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。

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；每个源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，就是从被导入的模块中，复制一个或多个变量到`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：它会严重地污染命名空间，让变量意义变得模糊，少用为妙。

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。

由于它只与包中的代码相关，在这里，我们还将介绍Python最近的相对导入模块和语法。正如我们将看到的，这种方法修改了搜索路径并且扩展了在包中用于导入的from语句。

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。

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。

一般地，包导入路径中最左边的部分仍然是相对于我们在第21章所介绍的`sys.path`模块搜索路径列表中的一个目录。从此以后，脚本内的`import`语句明确指出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`.pth`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}。

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。

__init__.py包文件

如果选择使用包导入，就必须多遵循一条约束：包导入语句的路径中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，*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
```

以及这种形式的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钩子、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（我们将在第24章看到`__all__`的另一种用法）。

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。

注意：不要把包`__init__.py`文件和我们将在本书下一部分中介绍的类`__init__`构造函数方法搞混淆了。前者是当导入初次遍历一个包目录的时候所运行的代码的文件，而后者是当创建实例的时候才调用。它们都具有初始化的作用，但是，它们有很大的不同。

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`和`dir1`的子目录`dir2`中——这些文件的路径名在注释中给出：

```
# dir1\__init__.py
print('dir1 init')
x = 1

# dir1\dir2\__init__.py
print('dir2 init')
y = 2

# 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。

```
% 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
>>>
>>>from imp import reload              # Needed in 3.0
>>>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的`__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`引用的变量`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语句和import语句

`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重新输入路径。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
```

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 (see Chapter 24)
>>>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
```

包导入也可以大幅简化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。

三个系统的传说

实际中需要包导入的场合，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')
```

现在，问题开始出现了。为了让这能够工作，需要设置模块搜索路径，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，才需要使用`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。

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`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。

包相对导入

目前为止，对包导入的介绍主要集中在从包的外部导入包文件。在包自身的内部，包文件的导入可以使用和外部导入相同的路径语法，但是，它们也可能使用特殊的包内搜索规则来简化导入语句。也就是说，包内的导入可能相对于包，而不是列出包导入路径。

如今，这种工作与版本有关。Python 2.6首先在导入上隐式地搜索包目录，而Python 3.0需要显式地相对导入语法。这种Python 3.0的变化，通过使得相同的包的导入更为明显，从而增强代码的可读性。如果你刚开始使用Python 3.0，本节中你的关注点可能是其新的`import`语法。如果你过去已经使用过其他的Python包，你可能还对Python 3.0的模式有何不同感兴趣。

Python 3.0中的变化

包中的导入操作的工作方式在Python 3.0中略有变化。这种变化只适用于我们本章中已经学习过的包目录中的文件中的导入；其他文件中的导入像以前一样工作。对于包中的导入，Python 3.0引入了两个变化：

- 它修改了模块导入搜索路径语义，以默认地跳过包自己的目录。导入只是检查搜索路径的其他组件。这叫做“绝对”导入。

- 它扩展了from语句的语法，以允许显式地要求导入只搜索包的目录。这叫做“相对”导入语法。

这些变化在Python 3.0中完全展示了出来。新的from语句相对语法在Python 3.0中也可以使用，但是，默认搜索路径的变化必须作为一个选项打开。它目前计划添加到Python 2.7的发布中^{注2}，这一变化以这种方式纳入，是因为搜索路径部分不能和早期的Python向后兼容。

这一变化的影响是，在Python 3.0中，我们通常必须使用特殊的from语法来导入与导入者位于同一包中的模块，除非你从一个包根目录拼出一个完整的路径。没有这一语法，你的包不会自动搜索到。

相对导入基础知识

在Python 3.0和Python 2.6中，from语句现在可以使用前面的点号(“.”)来指定，它们需要位于同一包中的模块（所谓的包相对导入），而不是位于模块导入搜索路径上某处的模块（叫做绝对导入）。也就是说：

- 在Python 3.0和Python 2.6中，我们可以使用from语句前面的点号来表示，导入应该相对于外围的包——这样的导入将只是在包的内部搜索，并且不会搜索位于导入搜索路径（sys.path）上某处的同名模块。直接效果是包模块覆盖了外部的模块。
- 在Python 2.6中，包的代码中的常规导入（没有前面的点号），目前默认为一种先相对再绝对的搜索路径顺序，也就是说，它们首先搜索包自己的路径。然而，在Python 3.0中，在一个包中导入默认是绝对的——在缺少任何特殊的点语法的时候，导入忽略了包含包自身并在sys.path搜索路径上的某处查找。

例如，在Python 3.0和Python 2.6中，如下形式的一条语句：

```
from . import spam                # Relative to this package
```

告诉Python把位于与语句中给出的文件相同包路径中的名为spam的一个模块导入。类似的，语句：

```
from .spam import name
```

意味着“从名为spam的模块导入变量name，而这个spam模块与包含这条语句的文件位于同一个包下。”

注2： 是的，将有一个Python 2.7的发布版本，并且可能还有Python 2.8和随后的发布，这与新的Python 3.X的发布是并行的。正如本书前言中所介绍的，Python 2和Python 3产品线都期待在未来几年得到完全的支持，以容纳大量存在的Python 2用户和代码。

没有前面的点号的一条语句的行为，取决于你使用的Python的版本。在Python 2.6中，这样的一条import将会仍然默认为当前的先相对再绝对的搜索路径顺序（例如，先搜索包的目录，除非在导入文件中包含了如下形式的一条语句）：

```
from __future__ import absolute_import          # Required until 2.7?
```

如果出现这条语句，它打开了Python 3.0的默认绝对搜索路径变化，正如下面所介绍的。

在Python 3.0中，不带点号的一个import总是会引发Python略过模块导入搜索路径的相对部分，并且在sys.path所包含的绝对目录中查找。例如，在Python 3.0的方式中，如下形式的一条语句，总是在sys.path上的某处查找一个string模块，而不是查找该包中具有相同名称的模块。

```
import string                                  # Skip this package's version
```

没有了Python 2.6中的from __future__ 语句，如果包中有一个字符串模块，将会导入它。当导入修改打开的时候，要在Python 3.0和Python 2.6中得到相同的行为，运行如下形式的一条语句，来强制一个相对导入。

```
from . import string                          # Searches this package only
```

如今，这在Python 2.6和Python 3.0中都有效。Python 3.0中的方式的唯一区别是，当模块给定一个简单的名称，要把文件所在的同样的包目录中的该模块载入，它是必需的方式。

注意，前面的点号可以用来仅对from语句强制相对导入，而不对import语句这样。在Python 3.0中，import modname语句形式如今仍然执行相对导入（例如，首先搜索包的目录），但是，在Python 2.7中，这也将变成绝对的。前面没有点号的from语句与import语句的行为相同，在Python 3.0中是绝对的（略过包目录），并且在Python 2.6中是先相对再绝对（先搜索包目录）。

其他的基于点的相对引用模式也是可能的。在位于名为mypkg的一个包目录中的一个模块文件中，如下替代的import形式也像所述的那样工作。

```
from .string import name1, name2              # Imports names from mypkg.string  
from . import string                          # Imports mypkg.string  
from .. import string                        # Imports string sibling of mypkg
```

要更好地理解后面这些，我们需要理解这些变化背后的原理。

为什么使用相对导入

这个功能的设计初衷是，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。考虑如下包的目录。

```
mypkg\  
  __init__.py  
  main.py  
  string.py
```

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 2.6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。此外，我们无法使用包导入路径来解决这个问题，因为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。

Python 3.0中的相对导入解决方案

要解决这种问题，在包中运行的导入已经在Python 3.0中（并且作为Python 2.6的一个选项）改为绝对的。在这种方式下，在我们的示例文件`mypkg/main.py`中，一条如下形式的`import`语句将总是在包之外找到一个`string`，通过`sys.path`的绝对导入搜索。

```
import string                                # Imports string outside package
```

没有前面的点号的一条`from`语句，也看做是绝对的。

```
from string import name                      # Imports name from string outside package
```

如果你真的想要从包中导入一个模块，而没有给出从包根目录的完整路径，在`from`语句中使用点语法仍然可能做到相对导入。

```
from . import string # Imports mypkg.string (relative)
```

这种形式只是相对于当前的包导入`string`模块，并且是前面的导入示例的绝对形式的相对等价形式，当使用这一特殊的相对语法的时候，包的目录是唯一搜索的目录。

我们也可以使用相对语法从一个模块复制特定的名称。

```
from .string import name1, name2 # Imports names from mypkg.string
```

这条语句再次相对于当前包来引用`string`模块。如果这段代码出现在我们的`mypkg.main`模块中，它将从`mypkg.string`导入`name1`和`name2`。

实际上，相对导入中的“.”用来表示包含文件的包目录，而导入就出现在该文件中。前面再增加一个点，将执行从当前包的父目录的相对导入。例如，下面的语句。

```
from .. import spam # Imports a sibling of mypkg
```

将会导入`mypkg`的一个兄弟，位于该包自己的包含目录中的`spam`模块，该模块紧挨着`mypkg`。更通俗地说，位于某个模块`A.B.C`中的代码可以做下面任何一种导入。

```
from . import D # Imports A.B.D (. means A.B)
from .. import E # Imports A.E (.. means A)

from .D import X # Imports A.B.D.X (. means A.B)
from ..E import X # Imports A.E.X (.. means A)
```

相对导入 VS 绝对包路径

此外，一个文件有时候也可以在一条绝对导入语句中显式地指定其包。例如，在下面的语句中，将在`sys.path`的一个绝对路径中找到`mypkg`。

```
from mypkg import string # Imports mypkg.string (absolute)
```

然而，这依赖于配置以及模块搜索路径设置的顺序，尽管相对导入的点语法不会依赖于此。实际上，这种形式需要直接包含将要包含在模块搜索路径中的`mypkg`。通常，像这样显式地指定包的时候，绝对导入语句必须列出包的根目录下的所有目录。

```
from system.section.mypkg import string # system container on sys.path only
```

在较大或较深的包中，这可能比点语法要做更多的工作。

```
from . import string # Relative import syntax
```

使用后一种形式，包含的包自动搜索，而不管搜索路径的设置是什么。

相对导入的作用域

相对导入乍看可能有些令人困惑，但是，如果你记住一些关键点，会很有帮助。

- **相对导入适用于只在包内导入。**记住，这种功能的模块搜索路径修改只应用于位于包内的模块文件中的`import`语句。位于包文件之外的通常的导入将像前面所介绍的那样工作，首先自动搜索包含了顶级脚本的目录。
- **相对导入只是用于`from`语句。**还要记住，这一功能的新的语法只是用于`from`语句，而不适用于`import`语句。可以这样检测它，一个`from`中的模块名前面有一个或多个点号。包含点号但前面没有一个点号的模块名是包导入，而不是相对导入。
- **术语含糊不清。**坦率地讲，用来描述这一功能的术语可能太过令人混淆了。实际上，所有的导入对于某些事物来说都是相对的。在一个包外部，导入仍然是相对于`sys.path`模块搜索路径上列出的目录的。正如我们在第21章所了解到的，这个路径包含了程序的包含目录、`PYTHONPATH`设置、路径文件设置以及标准库。当交互地工作的时候，该程序的包含目录直接就是当前的工作目录。

对于包内部进行的导入，Python 2.6通过首先搜索包自身增强了这一行为。在Python 3.0的方式中，真正变化的只是常规的“绝对”导入语法忽略了一个包目录，但是，特殊的“相对”导入语法使其首先搜索并仅搜索它。当我们提及Python 3.0的导入是“绝对的”，我们真正的含义是它是相对于`sys.path`上的目录的，而不是相对于包本身。相反，当我们提及“相对”导入，我们的意思是，它们只是相对于包目录的。当然，一些`sys.path`目录是绝对路径或相对路径（我可能把事情讲的更容易令人混淆了些，但这可能是一种思路拓展）。

换句话说，Python 3.0中的“包相对导入”真的只是归结为删除了Python 2.6针对包的特殊搜索路径行为，并且添加了特殊的`from`语法来显式地要求相对行为。如果我们过去编写自己的包导入而不依赖于Python 2.6的特殊隐式相对查找（例如，总是通过拼写出从一个包根目录出发的完整路径），这种修改很大程度上是没有意义的。如果你不这么做，将需要更新自己的包文件，以使用针对本地包文件的新的`from`语法。

模块查找规则总结

使用包导入和相对导入，Python 3.0中的模块查找可以完整地概括为如下几条：

- 简单模块名（例如，A）通过搜索`sys.path`路径列表上的每个目录来查找，从左到右进行。这个列表由系统默认设置和用户配置设置组成。

- 包是带有一个特殊的`__init__.py`文件的Python模块的直接目录，这使得一个导入中可以使用`A.B.C`目录路径语法。在`A.B.C`的一条导入中，名为`A`的目录位于相对于`sys.path`的常规模块导入搜索，`B`是`A`中的另一个包子目录，`C`是一个模块或`B`中的其他可导入项。
- 在一个包文件中，常规的`import`语句使用和其他地方的导入一样的`sys.path`搜索规则。包中的导入使用`from`语句以及前面的点号，然而，它是相对于包的；也就是说，只检查包目录，并且不使用常规的`sys.path`查找。例如，在`from . import A`中，模块搜索限制在包含了该语句中出现的文件的目录之中。

相对导入的应用

理论介绍足够了，让我们来看一些快速测试，以展示相对导入背后的概念。

在包之外导入

首先，正如前面所提到的，这一功能不会影响到一个包之外的导入。因此，如下的代码像预期的那样查找标准库`string`模块。

```
C:\test>c:\Python30\python
>>>import string
>>>string
<module 'string' from 'c:\Python30\lib\string.py'>
```

但是，如果在所工作的目录中添加一个同名的模块，将会选择该模块，因为模块搜索路径的第一条是当前工作目录（CWD）。

```
# test\string.py
print('string' * 8)

C:\test>c:\Python30\python
>>>import string
stringstringstringstringstringstringstringstring
>>>string
<module 'string' from 'string.py'>
```

换句话说，常规的导入仍然相对于“主”目录（顶级的脚本的包含目录，或者我们正在工作的目录）。实际上，如果不是在用作一个包的部分的一个文件中，甚至不允许相对导入语法。

```
>>>from . import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

在本节的这个以及所有的示例中，在交互提示模式中输入的代码与将其放在一个顶层脚

本中运行的行为是相同的，因为`sys.path`上的第一个条目要么是交互的工作目录，要么是包含顶层文件的目录。唯一的区别是，`sys.path`的开始是一个绝对路径，而不是一个空字符串。

```
# test\main.py
import string
print(string)

C:\test>C:\python30\python main.py                # Same results in 2.6
stringstringstringstringstringstringstringstring
<module 'string' from 'C:\test\string.py'>
```

包内的导入

现在，让我们来去除在CWD中编写的本地`string`模块，并构建带有两个模块的一个包目录，包括必需空的`test\pkg__init__.py`文件（在这里省略了）。

```
C:\test>del string*
C:\test>mkdir pkg

# test\pkg\spam.py
import eggs                                # <== Works in 2.6 but not 3.0!
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)
```

这个包中的第一个文件试图用一条常规的`import`语句导入第二个文件。由于这在Python 2.6中当作相对的，而在Python 3.0中当做绝对的，因此，后者中将会失败。也就是说，Python 2.6首先搜索包含的包，但是Python 3.0不这么做。这是在Python 3.0中必须注意的非兼容行为。

```
C:\test>c:\Python26\python
>>>import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test>c:\Python30\python
>>>import pkg.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "pkg\spam.py", line 1, in <module>
      import eggs
ImportError: No module named eggs
```

为了使得这在Python 2.6和Python 3.0下都有效，使用特殊的相对导入语法来修改第一个文件，以便其导入在Python 3.0中也搜索包目录。

```

# test\pkg\spam.py
from . import eggs
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)

C:\test>c:\Python26\python
>>>import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test>c:\Python30\python
>>>import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>
99999

```

导入仍然是相对于CWD的

注意，在前面的示例中，包模块仍然必须访问`string`这样的标准库模块。实际上，它们的导入仍然相对于模块搜索路径上的条目，即便这些条目自身是相对的。如果我们再次向CWD添加了一个`string`模块，包中的导入将会在那里找到它，而不是在标准模块库中。尽管我们可以在Python 3.0中略过带有一个绝对导入的包目录，我们仍然不能略过导入包的程序的主目录。

```

# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import eggs
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)

C:\test>c:\Python30\python
>>>import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>
99999

```

使用相对导入和绝对导入选择模块

为了展示这如何应用于标准模块库的导入，再一次重新设置包。去除掉本地`string`模块，并在包自身之中定义一个新的。

```
C:\test>del string*
```

```
# test\pkg\spam.py
import string
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

<== Relative in 2.6, absolute in 3.0

现在，获得哪个版本的string模块取决于你使用哪个Python版本。和前面一样，Python 3.0把第一个文件中的导入解释为绝对的并略过了该包，但Python 2.6不会这么做。

```
C:\test>c:\Python30\python
>>>import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>

C:\test>c:\Python26\python
>>>import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

在Python 3.0中使用相对导入语法会迫使再次搜索包，就好像在Python 2.6中一样——通过在Python 3.0中使用绝对和相对导入语法，我们可以显式地跳过或选择包目录。实际上，这是Python 3.0方式所解决的情况。

```
# test\pkg\spam.py
from . import string
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

<== Relative in both 2.6 and 3.0

```
C:\test>c:\Python30\python
>>>import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>

C:\test>c:\Python26\python
>>>import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

相对导入语法实际上是一种绑定声明，而不只是一种偏好，注意到这点很重要。如果我们删除了这个例子中的string.py文件，string.py中的相对导入在Python 3.0和Python 2.6中都会失败，而不是回归到这一模块（或任何其他模块）的标准库版本。

```
# test\pkg\spam.py
from . import string

C:\test>C:\python30\python
>>>import pkg.spam
...text omitted...
ImportError: cannot import name string
```

<== Fails if no string.py here!

相对导入所引用的模块必须在包目录中存在。

导入仍然是相对于CWD的

尽管绝对导入允许我们略过包模块，它们仍然依赖于`sys.path`上的其他部分。为了最后进行一次测试，让我们定义自己的两个`string`模块。在下面的代码中，CWD中有一个为该名称的模块，包中有一个该名称的模块，并且，另一个`string`模块位于标准库中。

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import string          # <== Relative in both 2.6 and 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

当我们使用相对导入语法导入`string`模块时，我们如愿地得到了包中的那个版本。

```
C:\test>c:\Python30\python          # Same result in 2.6
>>>import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

使用绝对语法时，我们得到了不同的版本。Python 2.6将这条语句解释为相对于包的，但Python 3.0将其看做“绝对的”，在这种情况下，真的意味着它略过包并载入相对于CWD的版本（而不是相对于标准库的版本）。

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
import string          # <== Relative in 2.6, "absolute" in 3.0: CWD!
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test>c:\Python30\python
>>>import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>

C:\test>c:\Python26\python
>>>import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.pyc'>
```

你可以看到，尽管包显式地要求目录中的模块，它们的导入仍然是相对于常规模块搜索路径的剩余部分。在这个例子中，程序中的一个文件使用的包隐藏了该包可能想要的

标准库模块。在Python 3.0中真正发生的变化只是允许包代码在包的内部或外部选择文件。由于导入方法可能依赖于无法预见的一个封闭环境，Python 3.0中的绝对导入并不能保证找到标准库中的模块。

自行用这些例子进一步体验。实际上，并不像宣传的那样。在开发中，你通常可以构建自己的导入、搜索路径和模块名称，使其按预想的工作。然而，我们应该记住，在一个较大系统中的导入可能取决于所使用的环境，并且模块导入协议是一个成功的库设计的一部分。

注意：既然已经学习了有关包相对导入的知识，还要记住，它们不总是你的最佳选择。绝对包导入，相对于`sys.path`上的一个路径，有时候在Python 2的隐式包相对导入以及Python 2和3的显式包相对导入语法中，绝对导入都是首选。

包相对导入语法和Python 3.0的新的绝对导入搜索规则，至少从一个包的相对导入要显式化，并且由此更容易理解和维护。使用带点号的导入的文件，显式地绑定到一个包目录，并且需要做代码修改才能随处使用。

当然，这对于你的模块的影响程度可能随每个包而不同，当识别目录时，绝对导入也可能需要修改。

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，就需要使用如下的语句：

```
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，并且Python 3.0组甚至与包中的模块更为相关（包括tkinter GUI工具、HTTP网络连接工具等）。如下的语句导入了对Python 3.0中的各种标准库工具的访问：

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

无论你是否将要建立包的目录，最终可能还是从包的目录导入的。

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的部分从而找到模块。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去手动遍历搜索路径，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）。

由于它只与包中的代码相关，我们在这里也介绍了新的相对导入模式——这种方法在一个`from`之前用点号，从而导入包文件以选择同一包中的模块，而不是依赖于较旧的隐式包搜索规则。

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，就像往常一样，我们要以习题结束本章，来测试你在本章所学到的内容。

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？
5. `from mypkg import spam`和`from . import spam`有什么差别？

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。
5. `from mypkg import spam`是绝对导入：`mypkg`的搜索掠过包路径并且`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`。

高级模块话题

本章以一些更高级的模块相关话题作为第五部分的结尾：数据隐藏、`__future__` 模块、`__name__` 变量、`sys.path` 修改、列表工具、通过名称字符串来运行模块、过渡式重载等，此外还有本书这一部分所提到的相关陷阱和练习题。

在本章中，我们将构建一些比目前所见到的要更大和更有用的工具，它们组合了函数和模块。和函数一样，当模块接口定义良好时，它们要更有效率一些，因此，本章也简单地回顾了模块设计概念，其中的一些概念我们在之前的各章中已经介绍过。

尽管本章标题有“高级”二字，这是因为本章混合了一些额外的模块话题。这里所讨论的有些话题（例如，`__name__` 技巧）都得到了广泛使用，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，把下划线放在变量名前面（例如，__X__），可以防止客户端使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。

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，能良好地运行（参考第23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）。我们还使用该语句在第5章激活了Python 3.0的数字真除法，在第11章激活了Python 3.0 print调用，并且在第23章激活了Python 3.0绝对导入。

这些变动都有可能影响到Python 2.6中现有的程序代码，因此这种特定的导入形式，将作为可选的功能，逐步接受。

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：

- 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__':
    tester()                                # Only when run
                                           # Not when imported
```

这个模块定义了一个函数，让用户可以正常地导入并使用：

```
% 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...
```

实际上，一个模块的__name__变量充当一个使用模式标志，允许它编写成一个可导入的库和一个顶层脚本。尽管简单，我们将会看到这一钩子几乎在可能遇到的每个Python程序文件中应用。

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35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18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。

```
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，其名称不是__main__，就必须明确地调用这个函数来执行。

```
>>>import min
I am: min
>>>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。

使用带有__name__的命令行参数

这里是一个更为真实的模块示例，它展示了通常使用__name__技巧的另一种方式。如下的模块*formats.py*，为导入者定义了字符串格式化工具，还检查其名称看它是否作为一个顶层脚本在运行；如果是这样的话，它测试并使用系统命令行上列出的参数来运行一个定制的或传入的测试。在Python中，`sys.argv`列表包含了命令行参数，它是反映在命令行上录入的单词的一个字符串列表，其中，第一项总是将要运行的脚本的名称：

```
"""
Various specialized string display formatting utilities.
Test me with canned self-test or command-line arguments.
"""

def commas(N):
    """
    format positive integer-like N for display with
    commas between digit groupings: xxx,yyy,zzz
    """
    digits = str(N)
    assert(digits.isdigit())
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = (last3 + ',' + result) if result else last3
    return result

def money(N, width=0):
    """
    format number N for display with commas, 2 decimal digits,
    leading $ and sign, and optional padding: $ -xxx,yyy.zz
    """
    sign = '-' if N < 0 else ''
    N = abs(N)
    whole = commas(int(N))
```

```

    fract = ('%.2f' % N)[-2:]
    format = '%s%s.%s' % (sign, whole, fract)
    return '$%s' % (width, format)

if __name__ == '__main__':
    def selftest():
        tests = 0, 1          # fails: -1, 1.23
        tests += 12, 123, 1234, 12345, 123456, 1234567
        tests += 2 ** 32, 2 ** 100
        for test in tests:
            print(commas(test))

        print('')
        tests = 0, 1, -1, 1.23, 1., 1.2, 3.14159
        tests += 12.34, 12.344, 12.345, 12.346
        tests += 2 ** 32, (2 ** 32 + .2345)
        tests += 1.2345, 1.2, 0.2345
        tests += -1.2345, -1.2, -0.2345
        tests += -(2 ** 32), -(2**32 + .2345)
        tests += (2 ** 100), -(2 ** 100)
        for test in tests:
            print('%s [%s]' % (money(test, 17), test))

    import sys
    if len(sys.argv) == 1:
        selftest()
    else:
        print(money(float(sys.argv[1]), int(sys.argv[2])))

```

这个文件在Python 2.6和Python 3.0中都能工作。当直接运行时，它像前面那样测试自己，但是它使用命令行上的选项来控制测试行为。请你自己直接运行这个文件而不带命令行参数，看看它的自测试代码打印出什么。要测试特定的字符串，用一个最小的字段宽度将它们传入到命令行上：

```

C:\misc>python formats.py 999999999 0
$999,999,999.00

C:\misc> python formats.py -999999999 0
$-999,999,999.00

C:\misc>python formats.py 123456789012345 0
$123,456,789,012,345.00

C:\misc> python formats.py -123456789012345 25
$ -123,456,789,012,345.00

C:\misc>python formats.py 123.456 0
$123.46

C:\misc> python formats.py -123.454 0
$-123.45

C:\misc>python formats.py
...canned tests: try this yourself...

```

和前面一样，由于这段代码针对双模式用法编写，我们一般也可以把这些工具作为库的部分导入到其他的环境中：

[illegible]

由于这个文件使用了第15章介绍的文档字符串功能，我们也可以使用`help`函数来研究其工具——它充当一个通用目的的工具：

```
>>>import formats
>>>help(formats)
Help on module formats:

NAME
    formats

FILE
    c:\misc\formats.py

DESCRIPTION
    Various specialized string display formatting utilities.
    Test me with canned self-test or command-line arguments.

FUNCTIONS
    commas(N)
        format positive integer-like N for display with
        commas between digit groupings: xxx,yyy,zzz

    money(N, width=0)
        format number N for display with commas, 2 decimal digits,
        leading $ and sign, and optional padding: $ -xxx,yyy.zz
```

我们可以用类似的方式来使用命令行参数，为脚本提供通用的输入。这些脚本可能也会把自己的代码包装成函数和类以供导入者重用。想了解更高级的命令行处理，请参阅Python的标准库和手册中的`getopt`和`optparse`模块。在某些环境中，我们也可以使用第3章介绍的以及第10章使用的内置`input`函数，来提示shell用户以测试输入，而不是从命令行提取输入。

注意： 还请参阅第7章对于将在Python 3.1及以后的版本中可用的新的{,d}字符串格式化方法的语法；该方法像这里的代码一样，用千分位组的方式来格式化扩展。而这里列出的模块添加了货币格式，并且在Python 3.1之前的版本中，可以作为逗号插入的一种手动替代方式。

修改模块搜索路径

在第21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
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', ...more deleted...]

>>>sys.path.append('C:\\sourcedir')           # Extend module search path
>>>import string                             # All imports search the new dir last
```

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:\\lp4e\\examples')      # For this process only
>>>sys.path
['d:\\temp', 'c:\\lp4e\\examples']

>>>import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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PYTHONPATH和.pth，而且在程序结束后，它们还存在着。

Import语句和from语句的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 modulename as name
```

相当于：

```
import modulename
name = modulename
del modulename                                # 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名，赋值给脚本中的不同的变量名：

```
from modulename import attrname 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。

```
import reallylongmodulename as name    # Use shorter nickname
name.func()

from module1 import utility as util1    # Can have only 1 "utility"
from module2 import utility as util2
util1(); util2()
```

此外，使用第23章所提到的包导入功能时，也可将整个目录路径提供简短、简单的名称，十分方便。

```
import dir1.dir2.mod as mod              # Only list full path once
mod.func()
```

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，可很容易地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），因为它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。

```
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。

```
"""
mydir.py: a module that lists the namespaces of other modules
"""

seplen = 60
sepchr = '-'

def listing(module, verbose=True):
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print('name:', module.__name__, 'file:', module.__file__)
        print(sepline)

    count = 0
    for attr in module.__dict__:
        # Scan namespace keys
        print('%02d) %s' % (count, attr), end = ' ')
        if attr.startswith('__'):
            print('<built-in name>')
            # Skip __file__, etc.
        else:
            print(getattr(module, attr))
            # Same as __dict__[attr]
        count += 1

    if verbose:
        print(sepline)
        print(module.__name__, 'has %d names' % count)
        print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir)
    # Self-test code: list myself
```

注意顶端的文档字符串，就像在前面的`formats.py`示例中一样，因为我们可能想要将其用作一个通用的工具，编写一个文档字符串来提供通过`__doc__`属性或`help`函数可以访问的功能性信息（参见第15章了解详细内容）。

```
>>>import mydir
>>>help(mydir)
Help on module mydir:
```

注1：如第17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。


```

NAME
    mydir - mydir.py: a module that lists the namespaces of other modules

FILE
    c:\users\veramark\mark\mydir.py

FUNCTIONS
    listing(module, verbose=True)

DATA
    sepchr = '-'
    seplen = 60

```

我们在这个模块的最后也提供了自测试逻辑，它导入并列出自己。这里给出了在Python 3.0中产生的输出（要在Python 2.6中使用它，使用第11章介绍的`__future__import`导入来激活Python 3.0 print调用，end关键字只用于Python 3.0）：

```

C:\Users\veramark\Mark>c:\Python30\python mydir.py
-----
name: mydir file: C:\Users\veramark\Mark\mydir.py
-----
00) seplen 60
01) __builtins__ <built-in name>
02) __file__ <built-in name>
03) __package__ <built-in name>
04) listing <function listing at 0x026D3B70>
05) __name__ <built-in name>
06) sepchr -
07) __doc__ <built-in name>
-----
mydir has 8 names
-----

```

要使用这一工具来列出其他的模块，直接把模块作为对象传入到这个文件的函数中。这里，它列出了标准库中的tkinter GUI模块中的属性（即Python 2.6中的tkinter）：

```

>>>import mydir
>>>import tkinter
>>>mydir.listing(tkinter)
-----
name: tkinter file: c:\PYTHON30\lib\tkinter\__init__.py
-----
00) getdouble <class 'float'>
01) MULTIPLE multiple
02) mainloop <function mainloop at 0x02913B70>
03) Canvas <class 'tkinter.Canvas'>
04) AtSellLast <function AtSellLast at 0x028FA7C8>
...many more name omitted...
151) StringVar <class 'tkinter.StringVar'>
152) ARC arc
153) At <function At at 0x028FA738>
154) NSEW nsew
155) SCROLL scroll

```

```
-----  
tkinter has 156 names  
-----
```

稍后我们会再遇见`getattr`以及与其作用相似的操作。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}。

用名称字符串导入模块

一条`import`或`from`语句中的模块名是直接编写的变量名称。然而，有时候，我们的程序可以在运行时以一个字符串的形式获取要导入的模块的名称（例如，如果一个用户从一个GUI中选择一个模块名称）。遗憾的是，我们无法使用`import`语句来直接载入以字符串形式给出其名称的一个模块，Python期待一个变量名称，而不是字符串。例如：

```
>>> import "string"  
File "<stdin>", line 1  
    import "string"  
          ^  
SyntaxError: invalid syntax
```

直接把该字符串赋给一个变量名称也是无效的：

```
x = "string"  
import x
```

这里，Python将会尝试导入一个文件`x.py`，而不是`string`模块——一条`import`语句中的名称既变成了赋给载入的模块的一个变量，也从字面上标识了该外部文件。

为了解决这个问题，我们需要使用特殊的工具，从运行时生成的一个字符串来动态地载入一个模块。最通用的方法是，把一条导入语句构建为Python代码的一个字符串，并且将其传递给`exec`内置函数以运行（`exec`是Python 2.6中的一条语句，但是，它完全可以像这里展示的那样使用——直接省略掉圆括号）：

```
>>> modname = "string"  
>>> exec("import " + modname)           # Run a string of code  
>>> string                               # Imported in this namespace  
<module 'string' from 'c:\Python30\lib\string.py'>
```

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。

`exec`函数（及其近亲`eval`）编译一个代码字符串，并且将其传递给Python解释器以执行。在Python中，字节代码编译器在运行时可以使用，因此，我们像这样编写构建和运行其他程序的程序。默认情况下，`exec`运行当前作用域中的代码，但是，你可以通过传入可选的命名空间字典来更加具体地应用。

`exec`唯一的、真正的缺点是，每次运行时它必须编译`import`语句，如果它运行多次，如果使用内置的`__import__`函数来从一个名称字符串载入的话，代码可能会运行得更快。效果是类似的，但是，`__import__`运行模块对象，因此，在这里将其赋给一个名称以保存它：

```
>>>modname = "string"
>>>string = __import__(modname)
>>>string
<module 'string' from 'c:\Python30\lib\string.py'>
```

过渡性模块重载

我们在第22章学习了模块重载，这是选择代码中的修改而不需要停止或重新启动一个程序的一种方式。当我们重载一个模块时，Python只重新载入特殊模块的文件，它不会自动重载那些为了导入要重载文件的模块。

例如，如果要重载某个模块A，并且A导入模块B和C，重载只适用于A，而不适用于B和C。A中导入B和C的语句在重载的时候重新运行，但是，它们只是获取已经载入的B和C模块对象（假设它们之前已经导入了）。在实际的代码中，文件A.py如下：

```
import B                                # Not reloaded when A is
import C                                # Just an import of an already loaded module

% python
>>>...
>>>from imp import reload
>>>reload(A)
```

默认情况下，这意味着你不能依赖于重载来过渡性地选择程序中的所有模块中的修改；相反，必须使用多次`reload`调用来独立地更新子部分。对于交互测试的大系统而言，工作量很大。你可以通过在A这样的父模块中添加`reload`调用，从而设计自己的系统能够自动重载它们的子部分，但是，这会使模块的代码变复杂。

一种更好的办法是，编写一个通用的工具来自动进行过渡性重载，通过扫描模块的`__dict__`属性并检查每一项的`type`以找到要重新载入的嵌套模块。这样的工具函数应该递归地调用自己，来导航任意形式的导入依赖性链条。模块`__dict__`属性在前面已介绍过，并且第9章介绍过`type`调用，我们只需要把两种工具组合起来。

例如，下面列出的模块`reloadall.py`有一个`reload_all`函数来自动地重载一个模块，以及该模块导入的每个模块等，所有通往每个导入链条最底端的通路都被考虑到。它使用字典来记录已经重载的模块，递归地遍历导入链条，以及标准库的`types`模块，该模块直接为内置类型预定义`type`结果。访问字典的技术在这里用来在导入是递归或冗余的时候避免循环，因为模块对象可以是字典键（正如我们在第5章学习过的，如果我们使用`visited.add(module)`来插入的话，一个集合将提供类似的功能）：

```
"""
reloadall.py: transitively reload nested modules
"""

import types
from imp import reload                                # from required in 3.0

def status(module):
    print('reloading ' + module.__name__)

def transitive_reload(module, visited):
    if not module in visited:                          # Trap cycles, duplicates
        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.ModuleType:
            transitive_reload(arg, visited)

if __name__ == '__main__':
    import reloadall                                  # Test code: reload myself
    reload_all(reloadall)                             # Should reload this, types
```

要使用这一工具，导入其`reload_all`函数并将一个已经载入的模块的名称传递给它。当文件独立地运行，其自测试代码将会测试自己，它必须导入自己，因为它自己的名字并没有在（没有一个导入的文件中定义（这段代码在Python 3.0和Python 2.6中都有效，并且打印出相同的输出，因为我们已经在`print`中使用了`+`而不是一个逗号））：

```
C:\misc>c:\Python30\python reloadall.py
reloading reloadall
reloading types
```

如下是这个模块对于Python 3.0下的某些标准库模块工作的情况。注意，`os`是如何由`tkinter`导入的，但`tkinter`在`os`之前已经导入了`sys`（如果想要在Python 2.6下测试这段代码，用`Tkinter`替换`tkinter`）：

```

>>>from reloadall import reload_all
>>>import os, tkinter

>>>reload_all(os)
reloading os
reloading copyreg
reloading ntpath
reloading genericpath
reloading stat
reloading sys
reloading errno

>>>reload_all(tkinter)
reloading tkinter
reloading _tkinter
reloading tkinter._fix
reloading sys
reloading ctypes
reloading os
reloading copyreg
reloading ntpath
reloading genericpath
reloading stat
reloading errno
reloading ctypes._endian
reloading tkinter.constants

```

如下的会话展示了常规重载和过渡性重载的对比效果——除非使用过渡性工具，否则重载不会选取对两个嵌套的文件的修改：

```

import b                                # a.py
X = 1

import c                                # b.py
Y = 2

Z = 3                                    # c.py

C:\misc>C:\Python30\python
>>>import a
>>>a.X, a.b.Y, a.b.c.Z
(1, 2, 3)

# Change all three files' assignment values and save

>>>from imp import reload
>>>reload(a)                             # Normal reload is top level only
<module 'a' from 'a.py'>
>>>a.X, a.b.Y, a.b.c.Z
(111, 2, 3)

>>>from reloadall import reload_all
>>>reload_all(a)
reloading a
reloading b
reloading c

```

```
>>>a.X, a.b.Y, a.b.c.Z  
(111, 222, 333)
```

```
# Reloads all nested modules too
```

要更深入地了解，自己研究并体验这个示例，它是你可能想要添加到自己的源代码库中的另一个可导入工具。

模块设计理念

就像函数一样，模块也有设计方面的折中考虑：需要思考哪些函数要放进模块、模块通信机制等。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。
- **模块耦合要降到最低：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，除了与从模块导入的函数和类。
- **最大化模块的黏合性：统一目标。**可以通过最大化模块的黏合性来最小化模块的耦合性。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17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。

总之，图24-1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25章会介绍类（模块中的另一种对象）。

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。

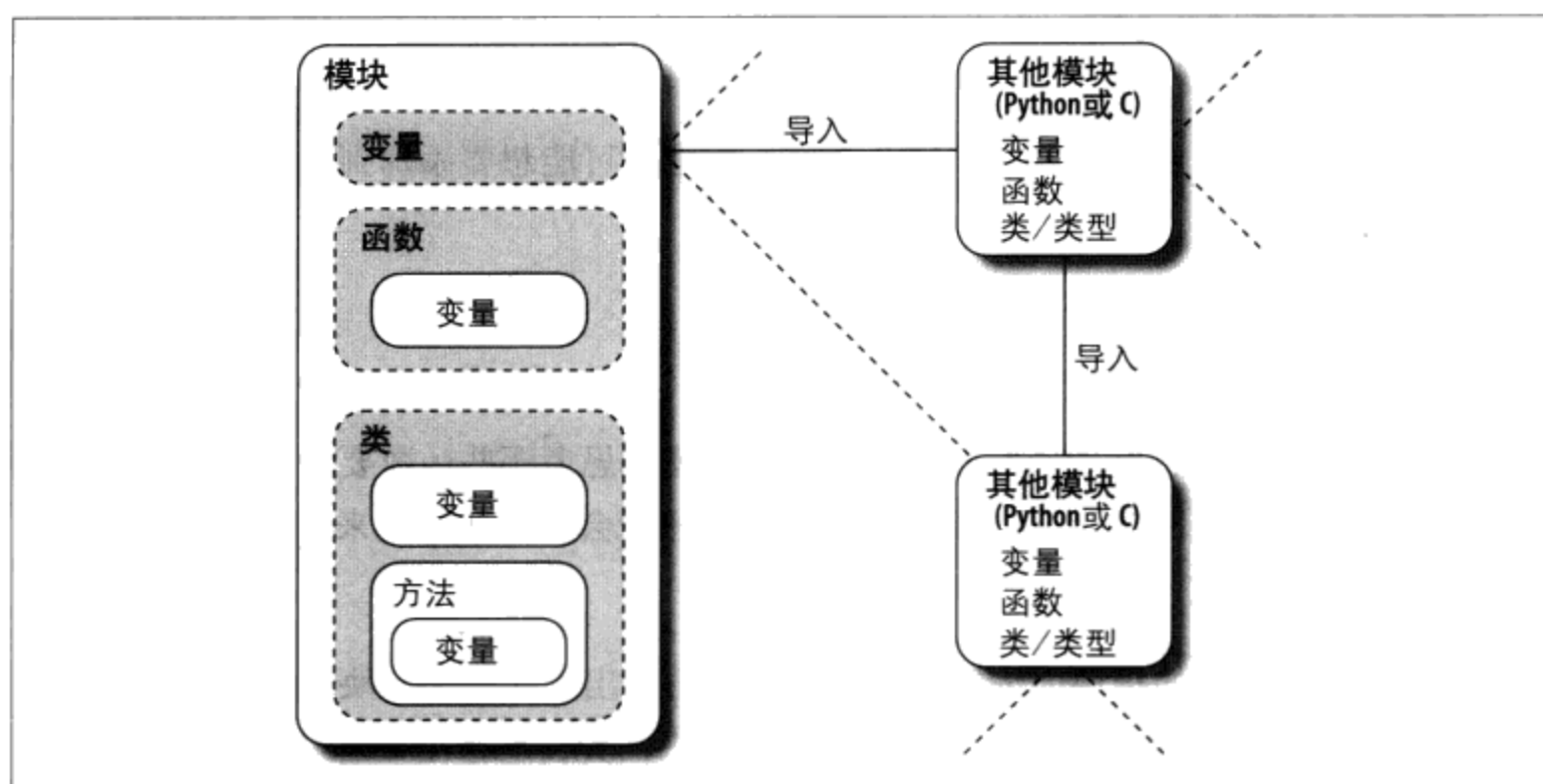


图24-1：模块的执行环境。模块是被导入的，但模块也会导入和使用其他模块，这些模块可以用Python或其他语言（例如，C语言）写成。模块可内含变量、函数以及类来进行其工作，而函数和类可包含变量和其他元素。不过，从最顶端来看，程序也只是一个模块的集合而已

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）相关的含义，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

def func1():
    print(func2())                     # Okay: "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，func2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。

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*）。

```
# 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：

```
# nested2.py
from nested1 import X, printer
X = 88
printer()

# Copy names out
# Changes my "X" only!
# 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*）。

```
# nested3.py
import nested1
nested1.X = 88
nested1.printer()

# Get module as a whole
# OK: change nested1's X

% python nested3.py
88
```

from *会让变量语义模糊

之前提到过这些内容，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，所以不会连接到变量名的那个模块。通过`from`导入的变量名就简单地变成了对象的引用，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!
...
from imp import reload
reload(module)                     # Changes module, but not my names
X                                  # Still references old object
```

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
...
```

```
from imp import reload
reload(module)
module.X
```

```
# Changes module in-place
# 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：

```
from imp import reload
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：

```
from imp import reload
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：

```
from imp import reload
import module
reload(module)
from module import function
function(1, 2, 3)
```

Or give up and use module.function()

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（实际上，这种情形在Python 3.0中甚至变得更糟糕，因为你必须也记住导入reload本身）。

不应该对reload和from能完美地合作抱有幻想。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）。

递归形式的from导入无法工作

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。

```
# recur1.py
X = 1
import recur2                                # Run recur2 now if it doesn't exist
Y = 2

# recur2.py
from recur1 import X                          # OK: "X" already assigned
from recur1 import Y                          # Error: "Y" not yet assigned

C:\misc>C:\Python30\python
>>>import recur1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "recur1.py", line 2, in <module>
    import recur2
  File "recur2.py", line 2, in <module>
    from recur1 import Y
ImportError: cannot import name Y
```

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耦合性，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、过渡性重载、由名称字符串导入等。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。

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. 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？
4. 改变sys.path和设置PYTHONPATH来修改模块搜索路径有什么不同？
5. 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. 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。
4. 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。
5. 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。

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]。

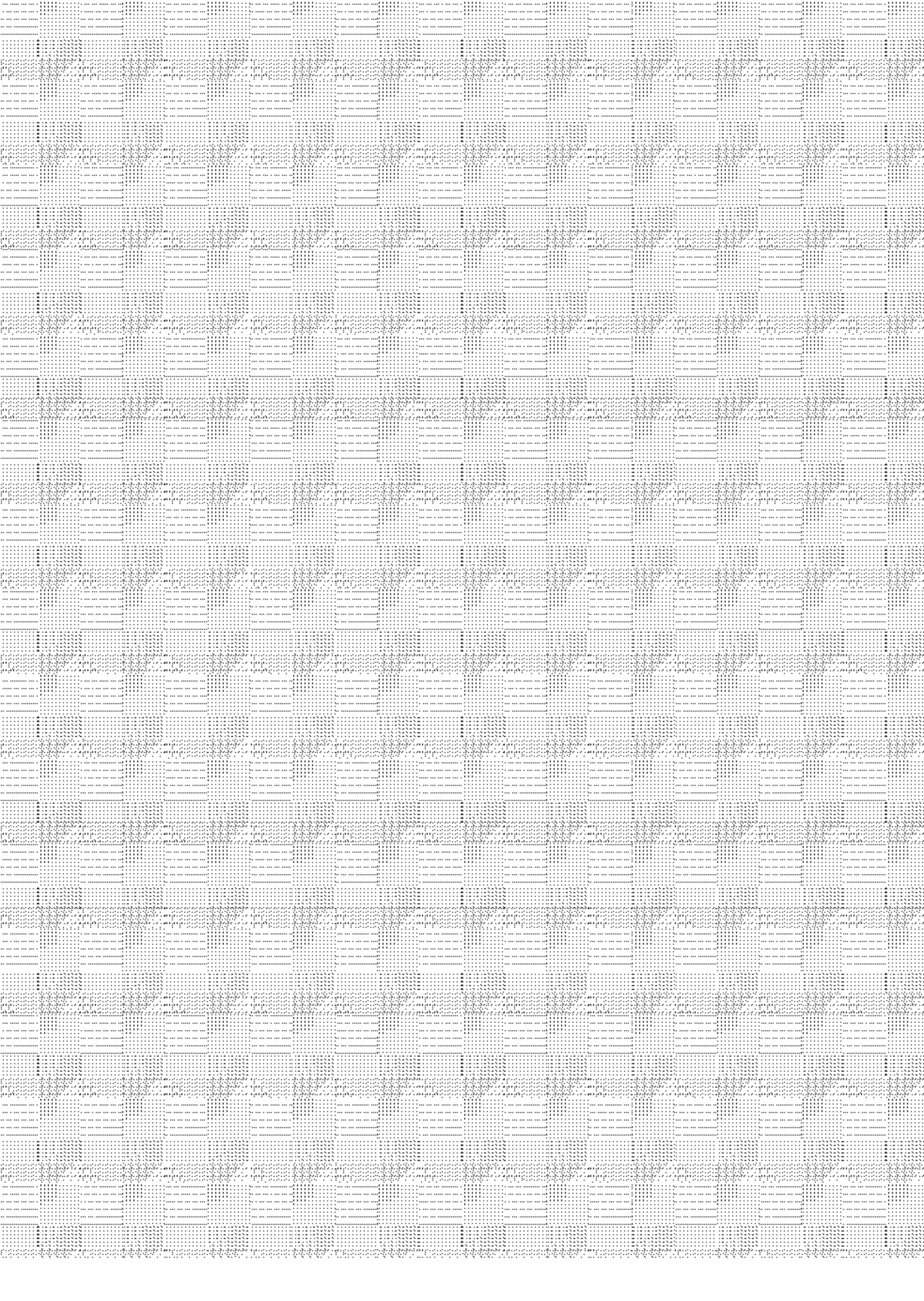
2. `from/from *`。使用 `from` 直接加载导出变量名，通过交互模式测试练习题1的 `mymod` 模块，先通过变量名导入，然后使用 `from *` 来获取所有。
3. `__main__`。在 `mymod` 模块中加入一行，只有在模块通过脚本运行时（而不是在其被导入时），才自动调用 `test` 函数。你加入的行可能会测试 `__name__` 的值是否为字符串 `"__main__"`，就像这一章所演示过的一样。试着从系统命令行运行模块。然后导入该模块，以交互模式测试其函数。两种模式都能用吗？

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。

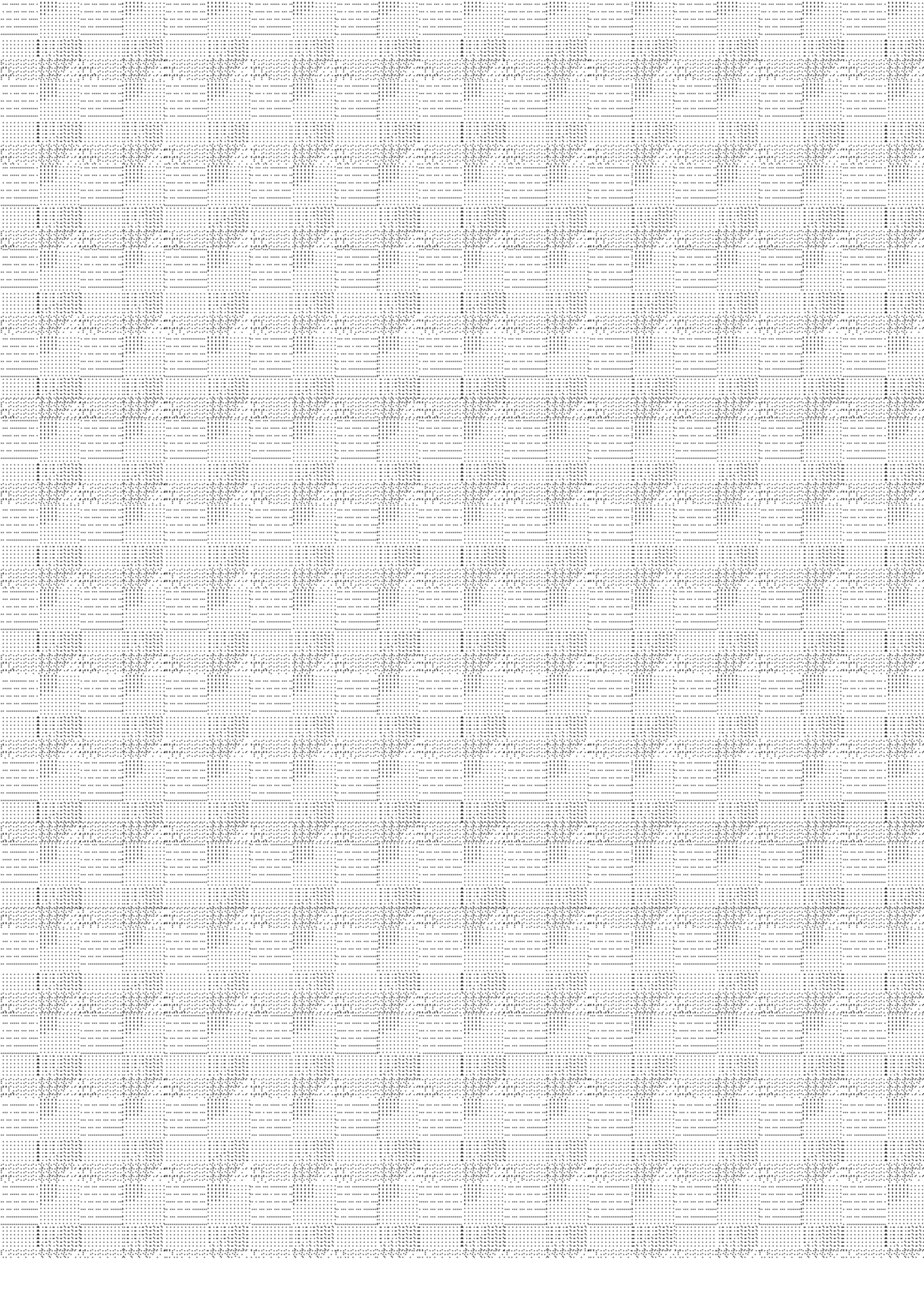
你需要在模块移入的目录中添加`__init__.py`文件才行，但是在所有主要Python平台上应该都能工作（这是Python使用“.”作为路径分隔字符的一部分原因）。创建的包目录可以是正在运行的目录底下的子目录。如果是这样，就可通过搜索路径的主目录元素找到，而不用去配置路径。加一些代码到`__init__.py`，并观察这些代码在每次导入时是不是都会运行。

6. 重载。用模块实验重载：运行第22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。



类和OOP



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），那么对象一般也需要参与到所谓的继承层次中。

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分。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，阅读本书这一部分你会得知，类是Python所提供的最有用的工具之一。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，学习类的基础知识是很有帮助的。

为何使用类

还记得本书曾介绍过，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16章作为例子的虚拟的制作比萨的机器人。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

制作比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，制作比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

制作比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，马达启动烤箱等。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能读取类的属性，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。

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。

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的故事，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——搜索对象连接的树，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。

换句话说，取出属性只是简单地搜索“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，连接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接对象树，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25-1是这种树的一个例子。

图25-1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）和两个实例对象（矩形的I1和I2）连接至继承搜索树。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。

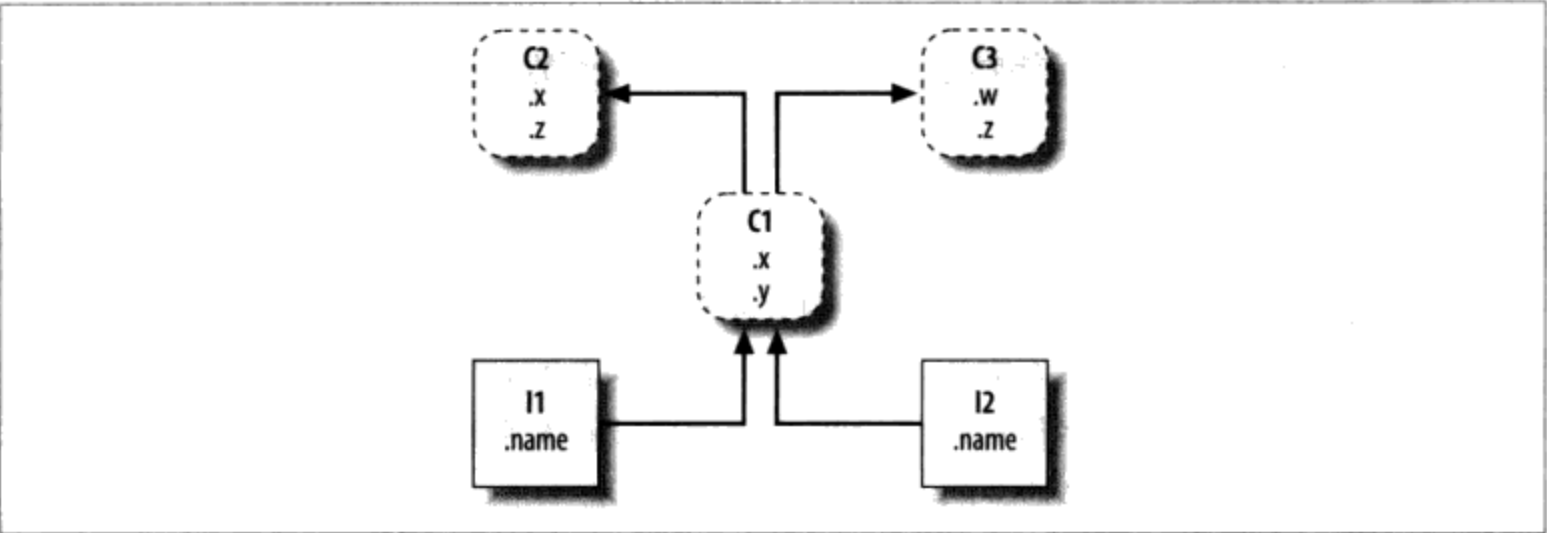


图25-1：类树，底端有两个实例（I1和I2），在它上有个类（C1），而顶端有两个超类（C2和C3）。所有这些对象都是命名空间（变量的封装），而继承就是由下至上搜索此树，来寻找属性名称所出现的最低的地方。代码隐含了这种树的形状

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25-1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。

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25-1的树，然后说：

注1： 在其他书籍中，偶尔也会遇到基类（base class）和派生类（derived class）来描述超类和子类。

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25-1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接的对象：

I2, C1, C2, C3

找到首个w之后就会停止搜索（但如果找不到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两者都会找到C1中的x并停止搜索，因为C1比C2位置更低。
- I1.y和I2.y两者都会找到C1中的y，因为这里是y唯一出现的地方。
- I1.z和I2.z两者都会找到C2中的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25-1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，相当于有效地取代其上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接，而类对应的是语句，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，更好地支持了软件定制。

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。那么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引用是一个函数调用，其实际的含义是“调用C3.w函数以处理I2”。也就是说，Python将会自动将I2.w()调用映射为C3.w(I2)调用，传入该实例作为继承的函数的第一个参数。

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。
- 实例自动连接至创建了这些实例的类。
- 类连接至其超类的方式是，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25-1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`语句创建了一个类对象，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'

```

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`出现在这种类的内部时，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}。

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。

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，它也称作是构造函数。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
```

注意我们可以对树中任何类创建实例，而不是只针对底端的类，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。

当我们想查看这些员工的薪资时，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章和第16章介绍过的多态概念的又一实例。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做什么。

```
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：

```
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

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。

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（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和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26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接对象树中的属性。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，在深入学习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。

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。这也称为构造函数。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__`构造函数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（下一章会介绍）。
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。

类代码编写基础

既然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。本章介绍Python中类模型的语法细节。

如果过去没用过OOP，若是囫圇吞枣的话，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架构有着很大的不同。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，反映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中用过的相比可能就大不相同了。

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
... 
```

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，后面再接一个或多个内嵌并且（通常）缩进的语句的主体。在这里，嵌套的语句是def，定义类要实现导出的行为的函数。

就像我们在IV部分学到的，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。其实是有三个连接命名空间，如图26-1所示。以OOP观点来看，我们说x是一个FirstClass对象，y也是。

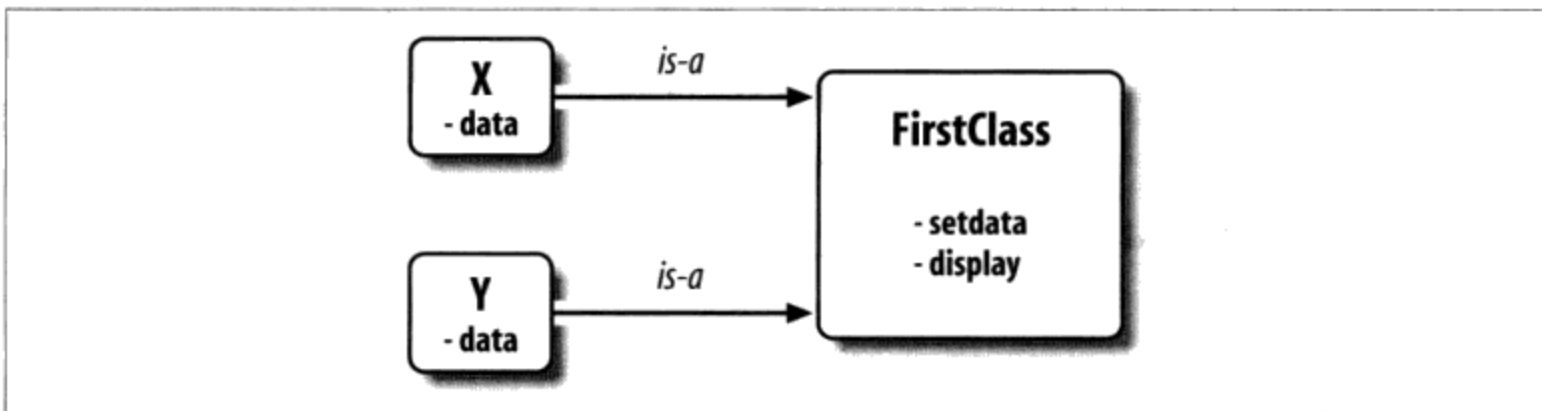


图26-1：类和实例是类树中通过继承搜索的相连的命名空间。这里，“data”属性会在实例内找到，但是“setdata”和“display”则是在它们之上的类中找到

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接搜

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，而且只与查找连接对象内的变量名（例如，图26-1的“是一个”连接）有关。

在FirstClass的setdata函数中，传入的值会赋给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26-1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（有时被称作成员）并没有声明。首次赋值后，实例就会存在，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，则会触发未定义变量名的错误：data属性以setdata方法赋值前，是不会有内存中存在的。

另一种正确判断这个模型动态方式的途径是，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。

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原地的修改。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。实际上，向层次的下端越深入，软件就会变得越特定。在这里和模块不一致：模块的属性存在于一个单一、平坦的命名空间之内（这个命名空间不接受定制化）。

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，而不是在原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
```

```
...     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有效地取代其超类内的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，所以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的、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26-2描绘了其中涉及的命名空间。

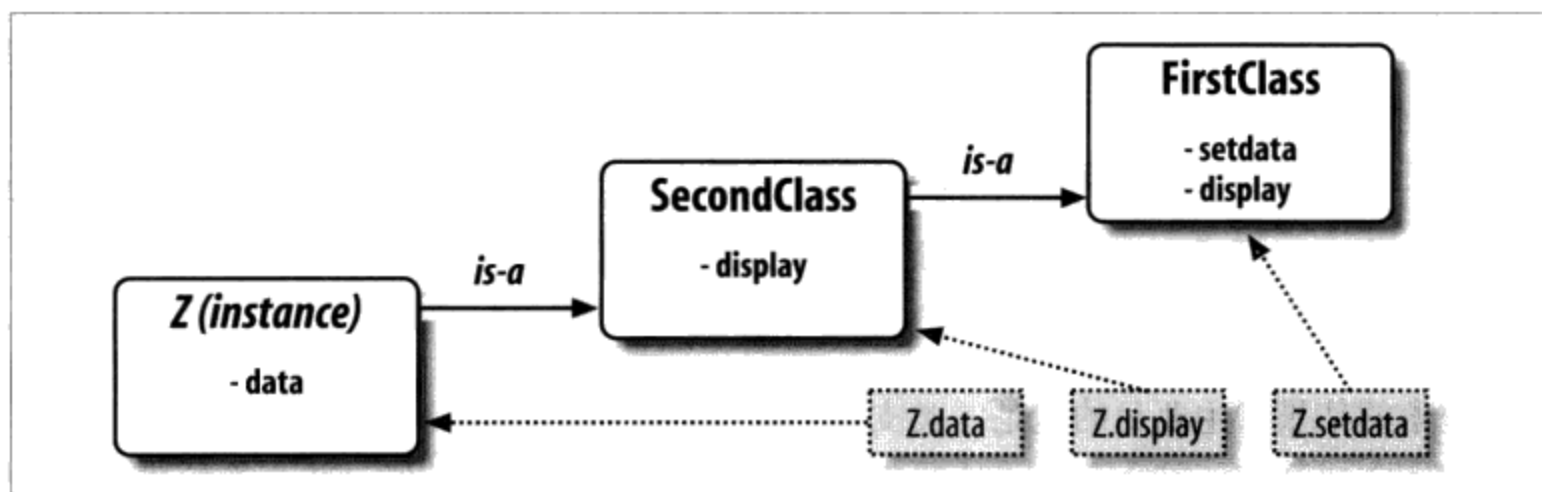


图26-2：在类树中较低的扩展类中重新定义变量名，从而覆盖了继承的变量名并将其专有化。在这里，SecondClass重新定义了方法display，从而定制了它的实例的display方法

这里有一件和OOP相关的很重要的事情要留意：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
```


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。

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地使用它的名称。

```
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，所以必须遵循第五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：

```
import person                            # Import module
```

```
x = person.person()
```

```
# Class within module
```

虽然这个路径看起来是多余的，但却是必需的：`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
x = person()
```

```
# Get class from module
# 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。实际上，Python中的通用惯例指出，类名应该以一个大写写字母开头，以使得它们更为清晰：

```
import person
x=person.person()
```

```
# Lowercase for modules
# Uppercase for classes
```

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**构造函数**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）^{注1}。
- 当`ThirdClass`实例出现在`+`表达式中时，则会调用`__add__`。
- 当打印一个对象的时候（从技术上讲，当通过`str`内置函数或者其Python内部的等价形式来将其转换为打印字符串的时候），运行`__str__`。

新的子类也定义了一个常规命名的方法，叫做`mul`，它在原处修改该实例的对象。如下是一个新的子类：

```
>>> class ThirdClass(SecondClass):                # Inherit from SecondClass
...     def __init__(self, value):                 # On "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):                   # On "self + other"
...         return ThirdClass(self.data + other)
```

注1： 不要与模块色中的`__init__.py`文件相混淆！见第23章，以获取更详细的信息。

```

...     def __str__(self):                                # On "print(self)", "str()"
...         return '[ThirdClass: %s]' % self.data
...     def mul(self, other):                             # In-place change: named
...         self.data *= other
...
>>> a = ThirdClass('abc')                                # __init__ called
>>> a.display()                                           # Inherited method called
Current value = "abc"
>>> print(a)                                              # __str__: returns display string
[ThirdClass: abc]

>>> b = a + 'xyz'                                         # __add__: makes a new instance
>>> b.display()                                           # b has all ThirdClass methods
Current value = "abcxyz"
>>> print(b)                                              # __str__: returns display string
[ThirdClass: abcxyz]

>>> a.mul(3)                                              # mul: changes instance in-place
>>> print(a)
[ThirdClass: abcabcabc]

```

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__构造函数内的参数value的，并将其赋值给self.data。直接效果是，ThirdClass计划在构建时自动设置data属性，而不是在构建之后请求setdata调用。

此外，ThirdClass对象现在可以出现在+表达式和print调用中。对于+，Python把左侧的实例对象传给__add__中的self参数，而把右边的值传给other，如图26-3所示；__add__返回的内容成为+表达式的结果。对于print，Python把要打印的对象传递给__str__中的self；该方法返回的字符串看做是对象的打印字符串。使用__str__，我们可以用一个常规的print来显示该类的对象，而不是调用特殊的display方法。

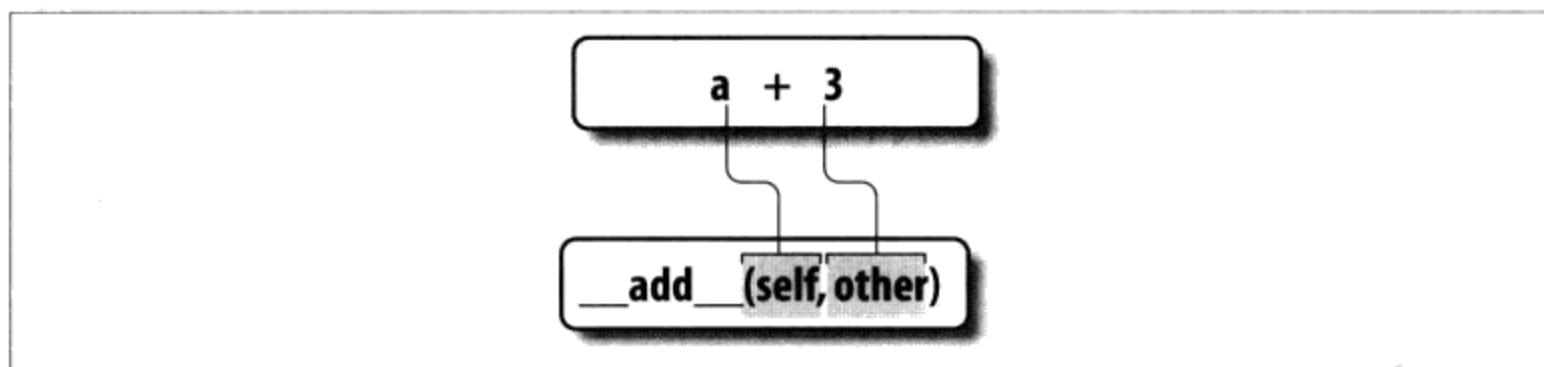


图26-3：在运算符重载中，在类的实例上执行的表达式运算符，和其他内置运算都会对应到类中特殊名称的方法。这些特殊方法是选用的，也可像平常那样继承。在这里，“+”表达式会触发“__add__”方法

__init__、__add__和__str__这样的特殊命名的方法会由子类 and 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函数）。

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）。我们可以通过重载`*`表达式来实现后者，但是，这一点和内置类型的行为不同，就像数字和字符串，总是替`*`运算符创建新对象。通常的实践说明，重载的运算符应该以与内置的运算符实现同样的方式工作。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。

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。尽管我们不能在本书中介绍每种运算符重载方法，我们将在第29章看看应用的一些额外的运算符重载技术。

几乎每个实际的类似乎都会出现的一个重载方法是：`__init__`构造函数。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，构造函数都是有用的。事实上，虽然Python不会对实例的属性进行声明，但通常也可以通过找到类的`__init__`方法的代码，而了解实例有哪些属性。

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：所涉及的就是在连接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
('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）只是连接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。

例如，`__dict__`属性是针对大多数基于类的对象的命名空间字典（一些类也可能在`__slots__`中定义了属性，这是一个高级而少用的功能，我们将在第30章和第31章学习）。如下的代码在Python 3.0中运行；名称和`__X__`内部名称集合所出现的顺序可能随着版本的不同而有所不同，但是，我们赋值的名称全部给出：

```
>>> rec.__dict__.keys()
['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__']

>>> list(x.__dict__.keys())
['name']

>>> list(y.__dict__.keys())
[]
```

list() not required in Python 2.6

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，每个实例都连接至其类以便于继承，如果你想查看的话，这个连接叫做__class__：

```
>>> x.__class__
<class '__main__.rec'>
```

类也有一个__bases__属性，它是其超类的元组：

```
>>> rec.__bases__
(<class 'object'>,)
# () empty tuple in Python 2.6
```

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）也可以完全独立地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）。

```
>>> upperName(x)
'SUE'
# Call as a simple function
```

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}。

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。


```

>>> 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其实就是在已连接命名空间对象内寻找属性而已。

类与字典的关系

尽管前面小节中简单类是用来说明类模型的基础知识，它们所使用的技术也可以用于实际的工作。例如，第8章介绍了如何使用字典来记录我们程序中实体的属性。它证实了类也可以充当这一角色，它们打包像字典这样的信息，但是，也可以以方法的形式绑定处理逻辑。为了便于参考，这里给出了我们在本书前面所使用过的基于字典的记录示例：

```

>>> rec = {}
>>> rec['name'] = 'mel'                        # Dictionary-based record
>>> rec['age'] = 45
>>> rec['job'] = 'trainer/writer'
>>>
>>> print(rec['name'])
mel

```

这段代码模拟了像其他语言中的记录这样的工具。正如我们所看到的，这里也有多种方式来用类做同样的事情。可能最简单的就是这种，利用键来记录属性：

```

>>> class rec: pass
...
>>> rec.name = 'mel'                          # Class-based record
>>> rec.age = 45
>>> rec.job = 'trainer/writer'
>>>
>>> print(rec.age)
40

```

这段代码的语法比其字典等价形式要少很多。它使用了一个空的`class`语句来产生一个空的命名空间对象。一旦我们产生了空类，我们随着时间用赋值类属性来填充它，这和以前一样。

这是有效的，但是，对于我们将需要的每一条不同的记录，都需要一条新的`class`语句。更通俗地说，我们可以产生一个空类的实例来表示每条不同的记录：

```
>>> class rec: pass
...
>>> pers1 = rec()                                # Instance-based records
>>> pers1.name = 'mel'
>>> pers1.job = 'trainer'
>>> pers1.age = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'vls'
>>> pers2.job = 'developer'
>>>
>>> pers1.name, pers2.name
('mel', 'vls')
```

这里，我们从相同的类产生了两条记录。实例开始的时候为空，就像类一样。然后，我们通过对属性赋值来填充记录。然而这一次，有两个分开的对象，由此有两个不同的`name`属性。实际上，同一个类的实例甚至不一定必须具有相同的一组属性名称；在这个示例中，其中一个就有唯一的`age`名称。实例实际上是不同的名称空间，因此，每个实例都有一个不同的属性字典。尽管它们通常由类方法一致地填充，但它们比我们所预期的更加灵活。

最后，我们可能编写一个更完整的类来实现记录及其处理：

```
>>> class Person:
...     def __init__(self, name, job):           # Class = Data + Logic
...         self.name = name
...         self.job = job
...     def info(self):
...         return (self.name, self.job)
...
>>> rec1 = Person('mel', 'trainer')
>>> rec2 = Person('vls', 'developer')
>>>
>>> rec1.job, rec2.info()
('trainer', ('vls', 'developer'))
```

这一方案也产生多个实例，但是，这次类不是空的：我们已经添加了**逻辑**（方法）在构建的时候来初始化实例并把属性收集到一个元组中。这里，构造函数在实例上强制了一些一致性，通过总是设置`name`和`job`属性。此外，类的方法和实例属性创建了一个包，它组合了数据和逻辑。

我们可以添加计算薪酬、罗列名称等逻辑。最后，我们可能把该类连接到一个更大的层级中，以通过类的自动属性搜索来继承已有的一组方法，或者甚至可能把类的实例存储

到一个文件中，该文件带有Python对象pickle化以使其持久。实际上，在下一章中，我们将用一个更加实际的运行实例来展示类基础知识的引用，从而实现类和记录的类比。

最后，尽管类型像字典一样是灵活的，但类允许我们以内置类型和简单函数不能直接支持的方式为对象添加行为。尽管我们也可以把函数存储到字典中，但再也没有比类更加自然的地方，可以使用它们来处理隐含的实例了。

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）。

既然我们已经学习了有关Python中编写类的所有机制，下一章将转向一个较大并较为实际的示例，它把我们目前已经学习的很多OOP知识连接到了一起。下一章我们要继续讨论类的编写，再次介绍这个模型，添加一些为了让事情保持简单而在这里省略的细节。不过，首先得做一做习题，复习本章介绍的基础知识。

本章习题

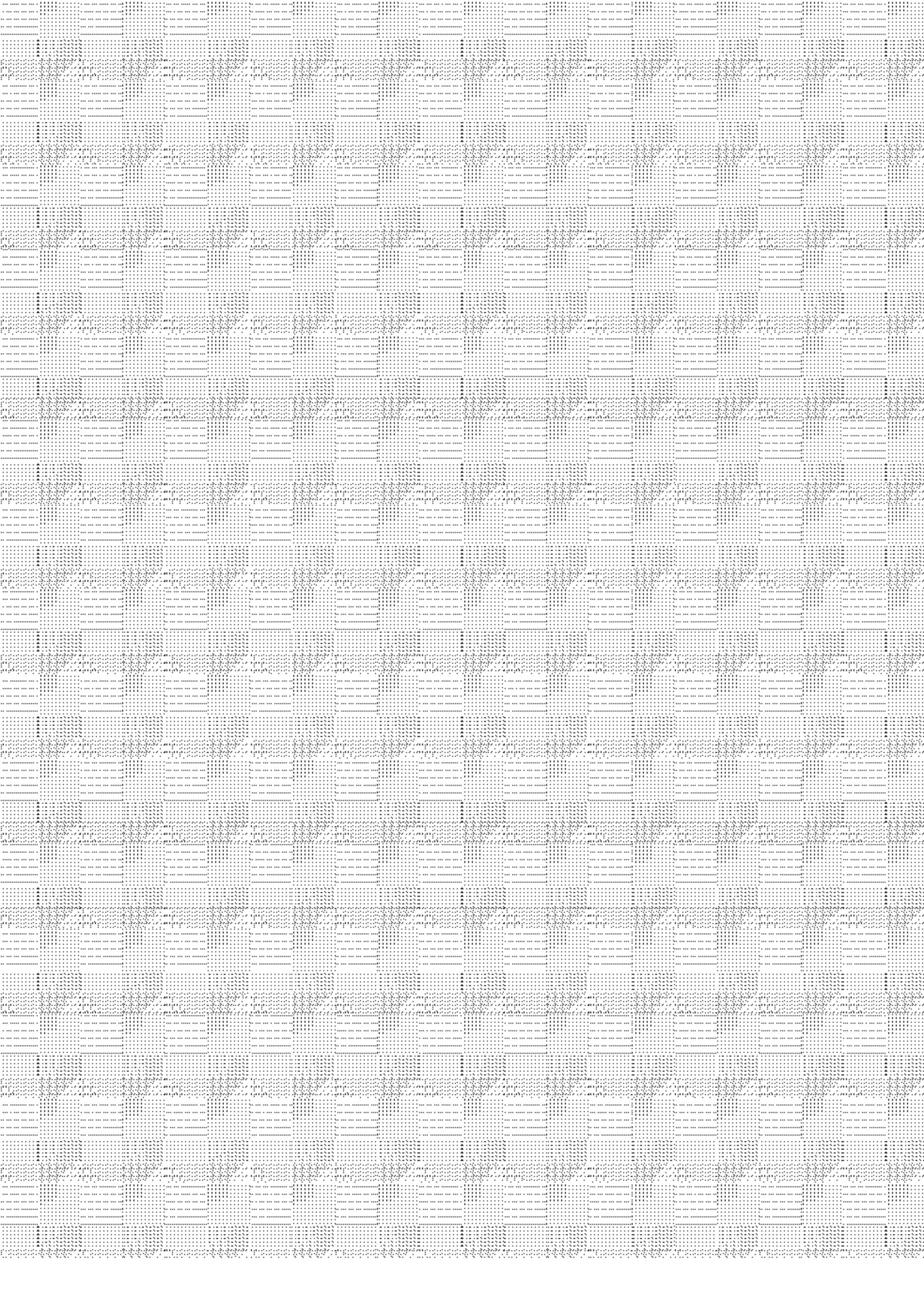
1. 类和模块之间有什么关系？
2. 实例和类是如何创建的？
3. 类属性是在哪里创建的？是怎样创建的？
4. 实例属性是在哪里创建的？是怎样创建的？
5. Python类中的self有什么意义？
6. Python类中如何编写运算符重载？
7. 什么时候可能在类中支持运算符重载？
8. 哪个运算符重载方法是最常用的？
9. Python OOP程序代码中最重要的两个概念是什么？

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__构造函数中初始化的。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用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__构造函数是最常用的。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__构造函数是Python中OOP的两个基石。



更多实例

我们将在下一章中介绍类语法的更多细节。但是，在开始之前，我想先展示关于类的一些更加实际的例子，这些例子比我们前面见到的例子更实用。在本章中，我们将构建一组类来做一些更加具体的事情，如记录和处理有关人员的信息。你将会看到，在Python编程中我们所谓的**实例**和**类**，往往与传统的术语**记录**和**程序**扮演着相同的角色。

特别是，在本章中，我们将编写两个类：

- **Person**——创建并处理关于人员的信息的一个类。
- **Manager**——一个定制的**Person**，修改了继承的行为。

在这个过程中，我们将创建两个类的实例并测试它们的功能。当我们完成之后，将给出使用类的一个漂亮的例子，我们把实例存储到一个*shelve*的面向对象数据库中，使它们持久化。通过这种方式，我们可以把这些代码用作模板，从而发展为完全用Python编写的一个完备的个人数据库。

除了实际的工具，我们在这里的目标还具有**教育意义**：本章提供了关于Python中面向对象程序设计的教程。通常，人们能够掌握上一章中书面上的类语法，但是，当面对必须从头编写一个新类的时候，要搞清楚如何下手还是有麻烦。出于这一目的，我们将在这里每次介绍一个步骤，帮助你学习基础知识；我们将逐渐地构建类，以便可以看到其功能如何组合到一个完整的程序中。

最后，我们的类在代码量上相对较小，但是它们将展示Python的OOP模型的所有主要思想。不管其语法细节如何，Python的类系统实际上很大程度上就是在一堆对象中查找属性，并为函数给定一个特殊的第一个参数。

步骤1：创建实例

好了，对于设计阶段介绍了这么多，让我们开始实现。第一个任务是开始编写主类 `Person`。在你喜欢的文本编辑器中，打开一个新的文件以便编写代码。在Python中，模块名使用小写字母开头，而类名使用一个大写字母开头，这是通用的惯例；就好像方法中使用 `self` 作为参数名，这可能不是语言所要求的，但是，这种违背惯例的做法很可能让随后阅读你的代码的人搞混淆了。按照惯例，我们将新的模块文件命名为 `person.py`，将其中的类命名为 `Person`，如下所示：

```
# File person.py (start)
```

```
class Person:
```

在本章前面部分，我们所有的工作都在这个文件中完成。在Python中的单个模块文件中，我们可以编写任意多个函数和类，并且，如果我们稍后添加不太相关的内容，这个 `person.py` 的名字可能显得意义不大。现在，我们将假设其中的所有内容都是和 `Person` 相关的。正如我们所了解的，当模块拥有一个单一、一致的用途的时候，它们会工作得更好。

编写构造函数

现在，我们要做的有关 `Person` 类的第一件事情就是记录关于人员的基本信息，如果你愿意的话，就填充记录字段。当然，在Python的术语中，这叫做实例对象属性，并且它们通常通过给类方法函数中的 `self` 属性赋值来创建。赋给实例属性第一个值的通常方法是，在 `__init__` 构造函数方法中将它们赋给 `self`，构造函数方法包含了每次创建一个实例的时候Python会自动运行的代码。让我们给自己的类添加一个构造函数：

```
# Add record field initialization
```

```
class Person:
```

```
    def __init__(self, name, job, pay):  
        self.name = name  
        self.job = job  
        self.pay = pay
```

```
# Constructor takes 3 arguments  
# Fill out fields when created  
# self is the new instance object
```

这是很常见的编码模式：我们传入的数据作为构造函数方法的参数附加到一个实例上，并且将它们赋给 `self` 以保持持久。在OO术语中，`self` 就是新创建的实例对象，而 `name`、`job` 和 `pay` 变成了状态信息，即保存在对象中供随后使用的描述性数据。尽管其他的技术（例如，封装作用域引用）也可以保存细节，但实例属性使得这个过程很明确而且易于理解。

注意，这里参数名出现了两次。这段代码乍看起来有点多余，但实际上不是这样。例

如，`job`参数在`__init__`函数的作用域里是一个本地变量，但是`self.job`是实例的一个属性，它暗示了方法调用的内容。这是两个不同的变量，但恰好具有相同的名字。通过`self.job=job`把本地的`job`赋给了`self.job`属性，我们在实例中保存了传入的`job`以供随后使用。在Python中，通常一个名字赋给什么地方（或者说，它赋给了什么对象）决定了它的含义是什么。

说到参数，`__init__`实在没有什么奇妙之处，只不过当产生一个实例的时候，会自动调用它并且它有特殊的第一个参数。尽管它的名字很怪异，它仍然是一个常规的函数，并且支持我们已经介绍的所有的函数特性。例如，我们可以为它的参数提供默认值，从而当参数的值不可用的情况下，就不必提供参数值。

为了便于说明，我们让`job`参数成为可选的参数，它将默认为`None`，这意味着所创建的人（目前）没有工作。如果`job`默认为`None`，我们可能也希望默认的`pay`为0，以保持一致性（除非你知道有某些人能够做到不干工作光拿钱）。实际上，我们必须为`pay`指定一个默认值，因为根据Python的语法规则，一个函数定义中，在第一个拥有默认值的参数之后的任何参数，都必须拥有默认值：

```
# Add defaults for constructor arguments

class Person:
    def __init__(self, name, job=None, pay=0):           # Normal function args
        self.name = name
        self.job = job
        self.pay = pay
```

这段代码意味着当我们创建`Person`的时候，需要给`name`传入值，但是`job`和`pay`现在是可选的；如果忽略这两个值的话，它们默认为`None`和0。通常，`self`参数由Python自动填充以引用实例对象，把值赋给`self`属性就会将值赋给新的实例。

在进行中测试

这个类目前还没有做太多事情，它基本上只是填充了一条新记录的字段，但是，它确实是一个有效的类。此时，我们可以给它添加更多的代码来获得更多的功能，但是，我们还没有这么做。你可能已经开始认识到了，用Python编程其实真的就是一种增量原型，编写一些代码、测试它、编写更多的代码，再次测试，以此类推。由于Python提供了一种交互式会话，并且几乎在代码修改后立即转变，所以在进行中测试比编写大量代码后再一次性测试要更加自然。

然而，在添加更多功能之前，让我们来测试目前的代码：生成类的几个实例，并且显示构造函数所创建的它们的属性。我们可以交互地做到这点，但是，你现在可能已经猜到了，交互测试有其局限性，每次开始一个新的测试会话的时候都需要重新导入模块和重

新输入测试实例，这实在让人厌烦。更常见的做法是，Python程序员使用交互式提示来进行简单的一次性测试，但是通过在包含测试对象的文件的底部编写代码来进行更多的大量测试，如下所示：

```
# Add incremental self-test code

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.pay)
print(sue.name, sue.pay)
```

Test the class
Runs __init__ automatically
Fetch attached attributes
sue's and bob's attrs differ

注意，这里的bob对象针对job和pay接受了默认值，但是，sue显式地提供了值。还要注意，在创建sue的时候我们如何使用**关键字参数**；我们可以根据位置来传递，但是，关键字随后可以提醒我们这些数据是什么（并且它允许我们按照所喜欢的顺序从左到右传递参数）。再次提醒你，尽管__init__的名字不同寻常，但它是常规的函数，支持你所知道的关于函数的任何特性，包括都是默认值的参数或按照名字传递的关键字参数。

当这个文件作为脚本运行的时候，底部的测试代码创建了类的两个实例，并且打印出每个实例的属性（name和pay）：

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000
```

我们也可以在Python的交互提示模式下输入这个文件的测试代码（假设你在此首先导入了Person类），但是，像这样把负责测试的代码封装到模块文件中，使得我们将来更容易再次运行它们。

尽管这是相当简单的代码，但它已经展示了一些重要的内容。注意，bob的name不是sue的name，sue的pay也不是bob的pay。每一个都是独立的记录。从技术上讲，bob和sue都是**命令空间对象**，像所有的类实例一样，它们每一个都拥有各自类所创建的状态信息的独立副本。由于类的每一个实例都有自己的一组self属性，类通过这种方式来记录多个对象的信息是再自然不过的事情了；就像是内置类型一样，类充当一种**对象工厂**。其他的Python编程结构，例如，函数和模块，没有这样的概念。

以两种方式使用代码

尽管文件底部的测试代码是有用的，但是有一个大问题，当文件作为脚本运行的时候，

或者当它作为一个模块导入的时候，它的顶层的`print`语句都会运行。这意味着，如果我们确定要在这个文件中导入该类以便在其他某个地方使用它（我们在本章稍后将要用到它），那么每次导入文件的时候，我们都会看到其测试代码的输出。这不是一种很好的软件关系，因为，客户端程序可能不关心我们内部的测试，也不希望看到我们的输出和它们自己的输出混合到一起。

尽管我们可以把测试代码分隔到一个单独的文件中，在与被测试的项目相同的一个文件中编写测试代码往往更加方便。但是，只有当文件为了测试而运行，而不是导入文件的时候，在文件底部运行测试语句才会比较好安排。`__name__`检查模块的设计意图正是如此，这一点我们在前面已经介绍过。这需要进行如下的添加：

```
# Allow this file to be imported as well as run/tested

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':                                # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

现在，我们得到了确实想要的操作：把文件作为顶层脚本运行的时候，测试它，因为其`__name__`是`__main__`，但随后将它作为类库导入的时候，则不会这么做：

```
C:\misc> person.py
Bob Smith 0
Sue Jones 100000

c:\misc> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) ...
>>> import person
>>>
```

导入的时候，文件现在定义了类，但是，没有使用它。当直接运行的时候，文件像前面一样创建了类的两个实例，并且打印出每个实例的两个属性。再次说明，由于每个实例都是一个独立的命名空间对象，它们的属性的值不同。

版本差异提示

我在Python 3.0下运行本章的所有代码，并且使用Python 3.0的print函数调用语法。如果你在Python 2.6下运行代码，代码也能正常工作，但是注意用圆括号括住一些输入行，因为打印语句中额外的圆括号把多个项目纳入一个元组：

```
c:\misc> c:\python26\python person.py
('Bob Smith', 0)
('Sue Jones', 100000)
```

如果这种差别太过详细，让你感到不适，直接去除掉圆括号以使用Python 2.6的print语句。可以使用格式化以产生一个单个的对象来打印，从而避免额外的圆括号。如下的两种形式在Python 2.6和Python 3.0中都有效，尽管使用方法的形式比较新。

```
print('{0} {1}'.format(bob.name, bob.pay))      # New format method
print('%s %s' % (bob.name, bob.pay))           # Format expression
```

步骤2：添加行为方法

一切看起来都很好，现在，我们的类基本上是一个记录工厂。它创建并填充了记录的字段（以Python的术语来说，是实例的属性）。即便有些局限性，但我们仍然可以在其对象上运行某些操作。尽管类添加了结构的一个额外的层级，它们最终还是通过嵌入和处理列表及字符串这样的基本核心数据类型来完成其大部分工作。换句话说，如果已经知道如何使用Python的简单的核心类型，那就已经知道了类的大部分用法；类其实只是一个最小的结构性扩展。

例如，对象的name字段只是一个简单的字符串，因此，我们通过在空格和索引处分隔来从对象提取姓氏。这些都是核心数据类型操作，不管其内容是否嵌入类实例中，这些操作都有效：

```
>>> name = 'Bob Smith'                # Simple string, outside class
>>> name.split()                       # Extract last name
['Bob', 'Smith']
>>> name.split()[-1]                   # Or [1], if always just two parts
'Smith'
```

与之类似，我们可以通过更新对象的pay字段来增加工资，即通过赋值来修改状态信息的当前状态。这一任务也涉及对Python的核心对象的基本操作，不管对象是独立的还是嵌入类结构中的：

```
>>> pay = 100000                       # Simple variable, outside class
```

```

>>> pay *= 1.10                                # Give a 10% raise
>>> print(pay)                                  # Or: pay = pay * 1.10, if you like to type
110000.0                                         # Or: pay = pay + (pay * .10), if you _really_ do!

```

要对脚本所创建的Person对象应用这些操作，像刚才对name和pay所做的那样，直接操作bob.name和sue.pay。操作是相同的，但目标对象和类结构中的属性联系起来了：

```

# Process embedded built-in types: strings, mutability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])                # Extract object's last name
    sue.pay *= 1.10                             # Give this object a raise
    print(sue.pay)

```

在这里我们添加了最后两行，当它们运行时，我们使用基本的字符串和列表操作提取了bob的姓氏，并且通过基本的数字操作修改sue的pay属性来给她涨工资。从某种意义上说，sue也是一个可修改的对象，原处修改其状态就好像是对一个列表进行append操作：

```

Bob Smith 0
Sue Jones 100000
Smith
110000.0

```

前面的代码按照计划工作，但是，如果你将其展示给一个资深的软件开发者，他可能会告诉你代码的一般方法在实际中并非好办法。像这样在类之外的硬编码操作可能会导致未来的维护问题。

例如，如果你已经在程序中的很多不同地方硬编码了姓氏提取操作，该怎么办呢？如果你需要改变其工作方式（例如，为了支持一种新的名字结构），你将需要查找这段代码每个出现的地方并进行更新。与之类似，如果涨工资代码发生变化（例如，需要提请批准或更新数据库），你可能有多个地方都需要修改。在较大的程序中，光是找到这些代码出现的所有地方就成问题，它们可能位于多个文件中、分散到多个单独的步骤中，等等。

编写方法

这里我们其实想要借用一种叫做**封装**的软件设计概念。封装的思想就是把操作逻辑包装到界面之后，这样，每种操作在我们的程序里只编码一次。通过这种方式，如果将来需要修改，只需要修改一个版本。此外，我们几乎可以随意地在这个单个副本内部修改代码，而不会影响到使用它的代码。

用Python术语来说，我们想要操作对象的代码位于**类方法**中，而不是分散在整个程序中。实际上，这是类非常好的地方之一，**构造**代码以删除冗余，并且由此优化维护。作为额外的好处，把操作放入方法中，还会使得这些操作可以应用于类的任何实例，而不是仅能应用于把它们硬编码来处理的那些对象。

理论上听起来有点复杂，但实际代码很简单。如下的动作通过把两个操作从类外部的代码移入类方法中，从而实现了封装。既然如此，让我们修改底部的self测试代码以使用所创建的新的方法，而不是硬编码操作：

```
# Add methods to encapsulate operations for maintainability

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue.pay)
```

Behavior methods
self is implied subject
Must change here only
Use the new methods
instead of hardcoding

正如我们已经介绍的，**方法**只是附加给类并旨在处理那些类的实例的常规函数。实例是方法调用的主体，并且会自动传递给方法的self参数。

这里的代码向方法中的转移很直接而简单。例如，新的lastName方法，直接对self做我们在前面版本中对bob硬编码所做的事，因为调用该方法的时候，self是隐藏的主体。lastName也返回结果，因为这个操作现在是一个调用的函数；它计算一个值以供其调用者使用，即便只是打印出它。类似的，新的giveRaise方法只是对self做我们在前面对sue所做的事情。

现在运行的时候，我们的文件的输出和前面很相似——我们主要只是重新组织了代码，以便将来可以更容易地修改，而不是修改代码的行为：

```
Bob Smith 0
Sue Jones 100000
Smith Jones
110000
```

这里有几个值得介绍的代码细节。首先，注意，`sue`的`pay`在涨工资之后仍然是一个整数，我们通过在方法中调用内置的`int`函数来把结果转换为整数。把值修改为`int`或`float`可能对于很多用途来说不是一个显著的问题（整数和浮点数对象具有同样的接口，并且可以在表达式中混合），但是，在一个正式的系统里，我们可能需要解决舍入问题（例如，钱对于`Person`来说是个问题）。

我们已经在第5章学习过，可以使用内置函数`round(N, 2)`来舍入并保留分币、使用`decimal`类型来修改精度，或者把货币值存储为一个完整的浮点数并且使用一个`%.2f`或`{0:.2f}`格式化字符串来显示它们从而显示出分币。对于这个例子，我们将直接用`int`截断任何分币（作为另一种思路，可以考虑第24章中的`formats.py`模块中的`money`函数，你可以导入这个工具，从而让显示带有逗号、分币符号和美元符号）。

其次，注意这一次我们已经打印了`sue`的姓氏名，因为姓氏逻辑已经封装到了方法中，我们可以对类的任何实例使用它。可以看到，Python通过自动把实例传递给第一个参数从而告诉一个方法应该处理哪个实例，通常这个参数叫做`self`。特别是：

- 在第一个调用`bob.lastName()`，`bob`是隐藏的主体，传递给了`self`。
- 在第二个调用`sue.lastName()`，`sue`传递给了`self`。

追踪这些调用，看看实例是如何传递给`self`中的。直接的效果是，每次方法来获取隐藏的主体的名字。对于`giveRaise`方法也是如此。例如，我们也可以按照这种方法对两个实例调用`giveRaise`，给`bob`涨工资。但是，遗憾的是，`bob`的0工资将会阻碍程序当前的代码增长其工资（这是我们要在软件的2.0版本中解决的问题）。

最后，注意`giveRaise`方法假设`percent`作为0和1之间的一个浮点数传递。在现实世界，这可能是一个基本的假设（一个1 000%的增长可能对于大多数人来说是一个Bug）。我们将允许按照这种模式传递，但是，我们可能想要在这段代码的一个未来的迭代中测试它或者至少记录下它。在本书稍后还将继续重复讨论这一思路，在那里，我们将编写所谓的函数装饰器，并且介绍Python的`assert`语句，这些机制也可以在开发中自动为我们进行验证性测试。

步骤3：运算符重载

现在，我们已经有了一个功能相当完备的类以及最初的实例，还有处理实例的两种新的行为（以方法的形式）。目前为止还不错。

按照目前的情况，测试还是不能像所需要的那样方便——要跟踪对象，必须手动地接受和打印单个的属性（例如，`bob.name`，`sue.pay`）。如果一次显示一个实例真的能给我们一些有用的信息，那还是不错的。遗憾的是，实例对象的默认的显示格式并不是很好，它显示对象的类名及其在内存中的地址（这在Python中基本上没有用处，除非是一个唯一的标识符）。

为了证实这点，把脚本的最后一行修改为`print(sue)`，以便它把对象显示为一个整体。将会得到如下的结果（在Python 3.0中，输出说sue是一个“对象”；在Python 2.6中，输出说sue是一个“实例”）：

```
Bob Smith 0
Sue Jones 100000
Smith Jones
<__main__.Person object at 0x02614430>
```

提供打印显示

好在通过使用**运算符重载**可以很容易做得更好——在一个类中编写这样的方法，当方法在类的实例上运行的时候，方法截获并处理内置的操作。特别是，我们可以使用可能是Python中第二常用的运算符重载方法，即上一章介绍过的继`__init__`之后的`__str__`方法。每次一个实例转换为其打印字符串的时候，`__str__`都会自动运行。由于这就是打印一个对象会做的事情，所以直接的效果就是，打印一个对象会显示对象的`__str__`方法所返回的内容，要么自己定义一个该方法，要么从一个超类继承一个该方法（双下划线的名称和任何其他名称一样继承）。

从技术上讲，我们已经编写的`__init__`构造函数方法也是运算符重载，它在构建的时候自动运行，以初始化一个新创建的实例。构造函数是如此常见，以至于它们几乎像是一个特殊的例子。像`__str__`这样更加专注的方法，允许我们输入专门的操作，并且当我们的对象用于这样的环境中的时候会提供**专门的行为**。

让我们来看看代码实现。下面的代码扩展了类以给出一个定制的显示，当类的实例作为一个整体显示的时候会列出属性，而不是依赖于用处较少的默认显示：

```
# Add __str__ overload method for printing objects

class Person:
    def __init__(self, name, job=None, pay=0):
```

```

        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
# Added method
# String to print

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)

```

注意，在这里，我们确实是在__str__中使用字符串%格式来构建显示字符串，在代码底部，类使用像这样的内置的类型对象和操作来完成任务。再一次说明，关于内置类型和函数，我们已经学习的所有内容都适用于基于类的代码。类很大程度上只是添加了一层结构，它把函数和数据包装在一起并且支持扩展。

我们已经直接把self测试代码修改为打印对象，而不是打印单个的属性。在运行的时候，现在的输出更加一致并且有意义了；新的__str__返回“[...]”行，该函数由打印操作自动运行：

```

[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]

```

这里有一个地方很微妙：正如我们在下一章将要学到的，一个相关的重载方法__repr__提供对象的一种代码低层级显示。有的时候，类提供了一个__str__以便实现用户友好的显示，同时也提供了一个__repr__以便让开发者看到额外的细节。由于打印运行__str__并且交互提示模式使用__repr__回应结果，因此，可以为两种目标观众都提供一种合适的显示。既然我们对于按照代码格式的显示没有兴趣，在类中，__str__就足够用了。

步骤4：通过子类定制行为

此时，我们的类已经具备了Python中的大多数OOP机制：它创建实例、在方法中提供行为，并且现在甚至做一些运算符重载来改变__str__中的打印操作。它有效地把我们的数据和逻辑一起包装到一个单个的、自包含的软件成分中，使得将来能够很容易地定位代

码并很直接地修改代码。通过允许我们封装行为，它还允许我们构建代码以避免冗余，并且还避免了难以维护。

它唯一没有涉及的主要的OOP概念就是**通过继承来定制化**。在某种意义上，我们已经实现过继承，因为实例从类那里继承了方法。要展示OOP的真正的能力，我们需要定义一个超类/子类关系，以允许我们扩展软件并替代一些继承的行为。毕竟，这是OOP背后的主要思想；基于已经完成的工作的定制来促进一种编码模式，可以显著地缩减开发时间。

编写子类

作为下一个步骤，让我们把OOP的方法付诸应用，并且通过扩展软件层级来定制Person类。为了编写这个教程，我们定义Person的一个子类，名为Manager，它用一个更加特殊的版本替代了继承的giveRaise方法。我们的新类从下面开始：

```
class Manager(Person):                                # Define a subclass of Person
```

这行代码意味着我们定义了一个名为Manager的新类，它继承自超类Person并且可能向Person添加了一些定制。更直白地说，Manager几乎像一个Person一样（必须承认，我绕了个大弯子讲了个小笑话……），但是Manager有一种定制的方法来涨工资。

为了防止争论，让我们假设当一个Manager要涨工资的时候，它像往常一样接受一个传入的百分比，但是，也会获得一个默认为10%的额外奖金。例如，如果一个Manager的加薪指定为10%，那么，它实际得到了20%的增长（与Person相关的生或死的问题，当然都与此严格地巧合）。我们新的方法像下面这样开始，因为giveRaise的这一新定义更接近于类树中的Manager实例，而不是最初的Person中的版本，它有效地替代了、从而定制了该操作。还记得吧，根据继承查找规则，名字的较低版本胜出：

```
class Manager(Person):                                # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10):          # Redefine to customize
```

扩展方法：不好的方式

现在，有两种方式编写Manager的定制方法：一种好的方式和一种不好的方式。让我们先从不好的方式开始，因为它可能更容易理解。不好的方式是复制和粘贴Person中的giveRaise的代码，然后针对Manager修改为如下所示：

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))    # Bad: cut-and-paste
```

这会像说明的那样工作，当我们随后调用Manager实例的giveRaise方法的时候，它会运行这个定制版本，算上额外的奖金。那么，运行正确难道有错吗？

这里的问题是一个非常常见的问题：任何时候，当你复制粘贴代码的时候，基本上都会使未来的维护工作**倍增**。考虑一下：因为我们复制了最初的版本，如果一旦改变了涨工资的方式（这完全是可能的），将必须修改两个地方而不是一个地方的代码。尽管这是一个小的、专门设计的例子，它也代表了一种广泛的问题，任何时候，当你试图按照这种复制代码的方式来编写程序的时候，可能需要考虑一下有没有更好的方法。

扩展方法：好的方式

我们这里真正想要做的事情是扩展最初的giveRaise，而不是完全替换它。在Python中做到这一点的好方式是，使用扩展的参数来直接调用其最初的版本，如下所示：

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)           # Good: augment original
```

这段代码利用了这样一个事实：类方法总是可以在一个**实例**中调用（这是通常的方式，Python把该实例自动地发送给self参数），或者通过**类**来调用（较少见的方式，其中，我们必须手动地传递实例）。用更有象征性的术语来说，记住，如下的常规方法调用：

```
instance.method(args...)
```

由Python自动地转换为如下的同等形式：

```
class.method(instance, args...)
```

其中，包含了要运行的方法的类，由应用于该方法的继承搜索规则来确定。我们可以在自己的脚本中以**任何一种形式**来编写，但是，两种形式之间略有差异——如果你直接通过类来调用，必须记住手动的传递实例。不管哪种方式，方法总是需要一个主体实例，并且Python只是对通过实例调用的方式自动提供实例。对于通过类名调用的方式，你需要自己给self发送一个实例；对于giveRaise这样的一个方法的内部代码，self已经是调用的主体，并且由此将实例传递过去。

通过类直接调用有效地扰乱了继承，并且把调用沿着类树向上传递以运行一个特定的版本。在这个例子中，我们可以使用这一技术来调用Person中默认的giveRaise，即便该方法在Manager层级已经重新定义了。在某种意义上，我们必须按照这种通过Person的方式来调用，因为，Manager的giveRaise代码内部的self.giveRaise()可能会循环——由于self已经是一个Manager，self.giveRaise()将会再次解析为Manager.giveRaise，以此类推，直到可用内存耗尽。

“好”的版本似乎喜欢在代码上略有不同，但是，它会对未来的代码维护意义重大，因为giveRaise现在只在一个地方（Person的方法），将来需要修改的时候，我们只要修改一个版本。实际上，这种形式更直接地抓住了我们的本意——我们想要执行标准的giveRaise操作，但直接加上一个额外的奖金。这里是实现了本步骤的整个模块文件：

```
# Add customization of one behavior in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):           # Redefine at this level
        Person.giveRaise(self, percent + bonus)      # Call Person's version

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)         # Make a Manager: __init__
    tom.giveRaise(.10)                                # Runs custom version
    print(tom.lastName())                             # Runs inherited method
    print(tom)                                         # Runs inherited __str__
```

为了测试我们的Manager子类定制，我们还添加了self测试代码，它创建一个Manager，调用其方法，并且打印它。这里是新版本的输出：

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

这里一切看上去都很好：bob和sue和以前一样，并且当Manager tom加薪10%的时候，他实际上得到了20%（它的pay从\$50K增加到了\$60K），因为Manager中定制的giveRaise只是对他运行。还要注意，测试代码的末尾如何把tom作为一个整体打印出来，按照

Person的__str__中定义的漂亮格式来显示：Manager对象获取lastName，而__init__构造函数方法的代码通过继承“免费”从Person得到。

多态的作用

为了使得这个从继承获取的行为更加惊人，我们在文件的末尾添加了如下代码：

```
if __name__ == '__main__':
    ...
    print('--All three--')
    for object in (bob, sue, tom):
        object.giveRaise(.10)
        print(object)
# Process objects generically
# Run this object's giveRaise
# Run the common __str__
```

输出结果如下所示：

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

在添加的代码中，对象是一个Person或Manager，Python自动运行相应的giveRaise——针对bob和sue使用Person中最初的版本，针对tom使用Manager中定制的版本。请自己跟踪方法调用，看看Python如何为每个对象选择正确的giveRaise方法。

这只不过是Python中所谓的**多态**，我们在本书前面介绍过，现在再次遇到，giveRaise具体做什么取决于你对它做什么。这里，它从我们已经在类中编写的代码中选取，从而让一切变得明朗。代码中实际的效果是，sue加薪10%而tom加薪20%，因为giveRaise根据对象的类型来分派。正如我们所知道的，多态是Python灵活性的核心。例如，把3个对象中的任何一个传递给调用了giveRaise方法的一个函数，将会有同样的效果：根据所传递的对象的类型，将会自动运行相应的版本。

另一方面，对于3个对象，打印都运行**相同的__str__**，因为其代码在Person中只出现一次。Manager既应用最初在Person中编写的代码，也对这些代码进行特殊化。尽管这个例子很小，我们已经利用OOP的特性实现了代码定制和复用。有了类，这些事情有时候会自动进行。

继承、定制和扩展

实际上，类比我们的例子所展示的更加灵活。通常，类可以**继承**、**定制**或**扩展**超类中已有的代码。例如，尽管我们在这里关注定制，但如果Manager需要一些完全不同的内容（Python同名引用扩展），我们也可以为Manager添加独特的、Person中所没有的方法。如下的代码段说明了这一情况。这里，giveRaise重新定义了一个超类方法以定制它，但somethingElse定义了一些新的内容已进行扩展：

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __str__(self): ...

class Manager(Person):
    def giveRaise(self, ...): ...      # Inherit
    def somethingElse(self, ...): ...  # Customize
                                     # Extend

tom = Manager()
tom.lastName()                       # Inherited verbatim
tom.giveRaise()                      # Customized version
tom.somethingElse()                  # Extension here
print(tom)                           # Inherited overload method
```

像这段代码的somethingElse这样的额外方法扩展了已有的软件，并且只能在Manager对象上使用，不能在Person上使用。然而，考虑到本教程的目的，我们将仅限于通过重新定义来定制Person的某些行为，而不是添加行为。

OOP：大思路

尽管我们的代码可能很小，但是它功能完备，并且它确实能够说明OOP背后一般性的要点：在OOP中，我们通过已经介绍过的**定制**来编程，而不是复制和修改已有的代码。初学者乍看上去，会觉得这没有什么突出的地方，特别是类需要额外的编码。但总的来说，类所隐藏的编程风格和其他的方法相比会显著地减少开发时间。

例如，在示例中，我们可能理论上已经实现了一个定制的giveRaise操作而没有子类化，但是，没有其他的选项能够产生像我们的代码那样优化的代码：

- 尽管我们可以**从头开始**编写Manager的全新的、独立的代码，但必须重新实现Person中所有那些与Manager相同的行为。
- 尽管我们可以直接原处**修改**已有的Person类来满足Manager的giveRaise的需求，但这么做可能会使需要原来的Person行为的地方无法满足需求。
- 尽管我们可以直接完整地**复制**Person类，将副本重新命名为Manager，并且修改其giveRaise，这么做将会引入代码冗余性，这会使我们将来的工作倍增——未来对

Person的修改无法自动找到位置，而是必须手动在Manager的代码中修改。通常，剪切复制的方法现在可能看上去很快，但是，会使未来的工作量倍增。

我们可以用类来构建的**可定制层级**，为那些将会随着时间而发展的软件提供了一个更好的解决方案。Python中没有其他的工具支持这种开发模式。因为我们可以通过编写新的子类来裁剪或扩展之前的工作，我们可以利用已经做过的工作，而不是每次从头开始、分解已经做过的工作或者引入代码的多个副本而所有的副本在将来可能都要更新。只要用对了，OOP就是程序员的强大同盟。

步骤5：定制构造函数

我们的代码现在能够正常工作了，但是，如果你仔细研究现在的版本，可能会对一些奇怪的地方感到沮丧——当我们创建Manager对象的时候，必须为它提供一个mgr工作名称似乎是没有意义的：这已经由类自身暗示了。如果在创建Manager的时候可以有某种方式自动填入这个值，那将会更好。

我们改善这点所需的技巧，证实与我们在前一节中所使用的技巧是**相同的**：我们想要以为Manager定制构造函数逻辑的方式来自动提供一个工作名。涉及代码，我们想要重新定义Manager中的__init__方法，从而提供mgr字符串。而且和giveRaise的定制一样，我们还想通过类名的调用来运行Person中最初的__init__，以便它仍然会初始化对象的状态信息属性。

如下的扩展将会完成这项工作——我们已经编写了新的Manager构造函数，并且把创建tom的调用修改为不传入mgr工作名称：

```
# Add customization of constructor in a subclass

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

Redefine constructor
Run original with 'mgr'

```

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)           # Job name not needed:
    tom.giveRaise(.10)                           # Implied/set by class
    print(tom.lastName())
    print(tom)

```

这里，我们再一次使用前面对giveRaise用过的相同技术来扩展__init__构造函数——通过类名直接调用并显式地传递self实例，从而运行超类的版本。尽管这个构造函数有一个奇怪的名字，效果还是相同的。由于我们也需要运行Person的构造函数逻辑（来初始化实例属性），我们实际上必须以这种方式调用它；否则，实例就不会附加任何的属性。

以这种方式调用重定义的超类构造函数，在Python中是一种很常用的编码模式。在构造的时候，Python自身使用继承来查找并调用唯一的一个__init__方法，也就是类树中最低的一个。如果你需要在构造的时候运行更高的__init__方法（并且常常会这么做），必须通过超类的名称手动调用它们。这种方法的积极之处在于，你可以明确指出哪个参数传递给超类的构造函数，并且可以选择根本就不调用它：不调用超类的构造函数允许你整个替代其逻辑，而不是扩展它。

这个文件的self测试代码和前面的相同，我们不用修改其内容，我们只是重新构造并去掉了一些冗余的逻辑：

```

[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]

```

OOP比我们认为的要简单

在这个完整的形式中，不管类的大小如何，它捕获了Python的OOP机制中几乎所有重要的概念：

- 实例创建——填充实例属性。
- 行为方法——在类方法中封装逻辑。
- 运算符重载——为打印这样的内置操作提供行为。

- 定制行为——重新定义子类中的方法以使其特殊化。
- 定制构造函数——为超类步骤添加初始化逻辑。

这些概念中的大多数都只是基于3个简单的思路：在对象树中继承查找属性、方法中特殊的self参数以及运算符重载对方法的自动派发。

通过这种方法，我们可以使自己的代码在未来易于修改，通过驾驭类的倾向以构造代码减少冗余。例如，我们把逻辑包装到方法中并返回来从扩展调用超类方法，以避免有相同代码的多个副本。这些步骤中的大多数都是类的结构性功能的自然产物。

大体上，这就是Python中的OOP的全部。类肯定可以变得比这更大，并且有一些更为高级的类概念，例如，装饰器和元类，我们将在后面的章节学习它们。但关于基础知识，我们的类已经都介绍到了。实际上，如果你掌握了我们所介绍的类的功能，大多数的OOP Python代码现在你都触手可及。

组合类的其他方式

说到这里，我还是应该告诉你，尽管Python中的OOP的基本机制很简单，但较大的程序中的一些技术在于组合类的方式。我们将在本教程中关注**继承**，因为这是Python语言所提供的机制，但是，程序员有时候也以其他的方式组合类。例如，一种常用的编码模式是把对象彼此嵌套以组成**复合对象**。我们将在本书第30章更详细地介绍这种模式，那更大程度上与设计有关而不是和Python有关；然而，作为一个快速浏览的例子，我们使用这种组合思想来编写Manager扩展的代码，将它嵌入一个Person中，而不是继承Person。

如下的替代方法使用__getattr__运算符（我们将在本书第29章介绍）重载方法来做到这点，我们将在本书第29章介绍使用内置函数getattr来拦截未定义属性的访问，并将它们委托给嵌入的对象。这里的giveRaise方法仍然实现定制，通过修改传递到嵌入的对象的参数。实际上，Manager变成了控制层，它把调用向下传递到嵌入的对象，而不是向上传递到超类方法：

```
# Embedding-based Manager alternative

class Person:
    ...same...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)           # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Intercept and delegate
    def __getattr__(self, attr):
```

```

        return getattr(self.person, attr)                # Delegate all other attrs
    def __str__(self):
        return str(self.person)                          # Must overload again (in 3.0)

if __name__ == '__main__':
    ...same...

```

实际上，这个Manager替代方案是一种叫做**委托**的常用代码模式的一个代表，委托是一种基于复合的结构，它管理一个包装的对象并且把方法调用传递给它。这种模式在我们的例子中有效，但是，它需要大约两倍的代码，并且对于我们想要表示的直接定制来说，它没有继承更适合（实际上，没有哪个明智的Python程序员会真正地按照这种方式编写这个例子，除非那些编写通用教程的人）。在这里，Manager不是一个真正的Person，因此，我们需要额外的代码手动为嵌入的对象分派方法；像__str__这样的运算符重载方法必须重新定义（至少，在Python 3.0中是这样，就像本章后面的方框“在Python 3.0中捕获内置属性”部分所介绍的），并且，由于状态信息是删除的一层，所以添加新的Manager行为不那么直接。

然而，当嵌入的对象比直接定制隐藏需要与容器之间有更多有限的交互时，**对象嵌入**以及基于其上的设计模式还是很适合的。例如，如果你想要跟踪或验证对另一个对象的方法的调用（实际上，当我们在本书稍后学习**类装饰器**的时候，我们将使用几乎相同的编码模式），像这里的替代Manager这样的控制器可能会很方便。此外，像下面这样的一个假设的Department可能**聚合**其他的对象，以便将它们当做一个集合对待。将这段代码添加到person.py文件的底部并自己尝试：

```

# Aggregate embedded objects into a composite

...
bob = Person(...)
sue = Person(...)
tom = Manager(...)

class Department:
    def __init__(self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

development = Department(bob, sue)                # Embed objects in a composite
development.addMember(tom)
development.giveRaises(.10)                        # Runs embedded objects' giveRaise
development.showAll()                             # Runs embedded objects' __str__s

```

有意思的是，这里的代码使用了继承和复合——Department是嵌入并控制其他对象的聚合的一个复合体，但是，嵌入的Person和Manager对象自身使用继承来定制。作为另一个例子，一个GUI可能类似地使用继承来定制标签和按钮的行为或外观，但也会复合以构建嵌入的挂件（如输入表单、计算器和文本编辑器）的一个更大的包。所使用的类结构取决于想要建模的对象。

第30章将讨论像复合这样的设计问题，因此，我们将在那里深入讨论该话题。这里再次强调，对于Python中OOP的基本机制，我们的Person和Manager类已经介绍完整了。由于已经掌握了OOP的基本知识，所以开发常用工具的时候更容易地将其应用于脚本之中，这往往是下一个很自然的步骤，也是下一节的主题。

在Python 3.0中捕获内置属性

在Python 3.0中（并且如果在Python 2.6中使用新样式的类），我们刚才编写的、替代的基于委托的Manager类，不重新定义它们不能够截取并委托像`__str__`这样的运算符重载方法属性。尽管我们知道，`__str__`是唯一的用于我们特定例子中的这样名字，但这对于基于委托的类是一个通用的问题。

还记得吧，像打印和索引这样的内置操作都隐式地调用`__str__`和`__getitem__`这样的运算符重载方法。在Python 3.0中，像这样的内置操作无法通过通用的属性管理器找到它们的隐式属性：`__getattr__`（针对未定义的属性运行）及其近亲`__getattribute__`（针对所有属性运行）都不会调用。这就是为什么我们必须在替代的Manager中冗余地重定义`__str__`，为了确保在Python 3.0中运行的时候，打印能够找到嵌入的Person对象。

从技术上讲，会发生这种情况是因为传统的类通常会在运行时在实例中查找运算符重载名，但是，新式的类不会这样，它们完全略过实例而在类中查找这样的方法。在Python 2.6的传统类中，内置方法不会像通常一样查找属性，例如，打印通过`__getattr__`找到`__str__`。新式的类也继承了一个默认的`__str__`，`__getattr__`无法找到它，但在Python 3.0中`__getattribute__`也不会截取这个名字。

这是一个改变，但是，这不是一个故障，基于委托的类可能通常会重定义运算符重载方法以委托它们去包装Python 3.0中的对象，或者手动，或者通过工具或超类。这个主题太高级，以至于不能在本书中进一步探讨，因此，这里不会过于详细介绍。本书第37章介绍属性管理的时候还会讨论它，并且本书第38章介绍私有类装饰符也会再次回顾。

步骤6：使用内省工具

在把我们的对象放入一个数据库中之前，让我们再来做最后一点调整。现在，类是完整的并且展示了Python中的大多数基本的OOP。然而，它们仍然有两个问题，我们应该在使用对象之前解决这两个问题：

- 首先，如果现在查看一下对象的显示，你会注意到，当打印tom的时候，Manager会把它标记为Person。这不是技术上的错误，因为Manager是一种定制的和特殊化的Person。然而，尽可能地用最确切（也就是说最低层）的类来显示对象，这可能会更准确些。
- 其次，可能也是更重要的，当前的显示格式只是显示了包含在__str__中的属性，而没有考虑未来的目标。例如，我们还无法通过Manager的构造函数验证tom工作名已经正确地设置为mgr，因为我们为Person编写的__str__没有打印出这一字段。更糟糕的是，如果我们改变或者修改了在__init__中分配给对象的属性集合，那么还必须记得也要更新__str__以显示新的名字，否则，将无法随着时间的推移而同步。

后面的一点意味着，未来在代码中引入冗余性的时候，我们自己必须做潜在的额外工作。由于__str__中的任何不一致将会反映到程序的输出中，所以这种冗余性可能比我们前面所解决的其他形式更为明显。然而，避免未来的额外工作通常总是好事。

特殊类属性

我们可以使用Python的内省工具来解决这两个问题，它们是特殊的属性和函数，允许我们访问对象实现的一些内部机制。这些工具较为高级，并且为其他程序员编写工具的人比开发应用程序的人要更为广泛地使用它们。即便如此，了解一些关于这些工具的基本知识也是有用的，因为它们允许我们编写以通用方式处理类的代码。例如，在我们的代码中，有两个钩子可以帮助我们解决问题，在上一章快结束的时候都介绍过：

- 内置的instance.__class__属性提供了一个从实例到创建它的类的链接。类反过来有一个__name__（就像模块一样），还有一个__bases__序列，提供了超类的访问。我们使用这些来打印创建一个实例的类的名字，而不是通过硬编码来做到。
- 内置的object.__dict__属性提供了一个字典，带有一个键/值对，以便每个属性都附加到一个命名控件对象（包括模块、类和实例）。由于它是字典，因此我们可以获取键的列表、按照键来索引、迭代其键，等等，从而广泛地处理所有的属性。我们使用这些来打印出任何实例的每个属性，而不是在定制显示中硬编码。

下面是这些工具在Python的交互模式中实际使用的情形。注意，我们如何在交互模式中

用一条from语句载入Person：类名存在语句之中，并且从模块导入，这与函数名和其他变量是完全相同的：

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> print(bob)                                # Show bob's __str__
[Person: Bob Smith, 0]

>>> bob.__class__                             # Show bob's class and its name
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())                 # Attributes are really dict keys
['pay', 'job', 'name']                       # Use list to force list in 3.0

>>> for key in bob.__dict__:
    print(key, '=>', bob.__dict__[key])      # Index manually

pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
    print(key, '=>', getattr(bob, key))      # obj.attr, but attr is a var

pay => 0
job => None
name => Bob Smith
```

正如前一章简单提到的，如果一个实例的类定义了__slots__，而实例可能没有存储在__dict__字典中，但实例的一些属性也是可以访问的，这是新式类（以及Python 3.0中的所有类）的一项可选的和相对不太明确的功能，即把属性存储在数组中，并且我们将在本书第30章和第31章讨论。既然slots其实属于类而不是实例，并且它们在任何事件中极少用到，那么我们在这里可以忽略它们而关注常规的__dict__。

一种通用显示工具

我们已经在超类中把接口投入使用，以显示准确的类名并格式化任何类的一个实例的所有属性。在文本编辑器中打开一个新文件，并编写如下代码：它是一个新的、独立的模块，名为classtools.py，仅仅实现了这样一个类。由于其__str__ print重载用于通用的内省工具，它将会对任何实例有效，不管实例的属性集合是什么。并且由于这是一个类，所以它自动变成一个公用的工具：得益于继承，它可以混合到想要使用它显示格式的任何类中。作为额外的好处，如果我们想要改变实例的显示，只需要修改这个类，于是，在其下一次运行的时候，继承其__str__的每一个类都将自动选择新的格式：

```
# File classtools.py (new)
"Assorted class utilities and tools"
```



```

class AttrDisplay:
    """
    Provides an inheritable print overload method that displays
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __str__(self):
        return ' [%s: %s]' % (self.__class__.__name__, self.gatherAttrs())

if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest):
        pass

    X, Y = TopTest(), SubTest()
    print(X)                                # Show all instance attrs
    print(Y)                                # Show lowest class name

```

注意这里的文档字符串，作为通用的工具，我们想要添加一些功能来产生文档，以供潜在的用户阅读。正如我们在第15章所介绍过的，文档字符串可以放在简单函数和模块的顶部，并且也可以放在类及其方法的开始出；`help`函数和PyDoc工具会自动地提取和显示它们（我们将在本书第28章再次介绍文档字符串）。

直接运行的时候，这个模块的self测试会创建两个实例并打印它们。这里定义的`__str__`显示了实例的类，及其所有的属性名和值，按照属性名排序：

```

C:\misc> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]

```

实例与类属性的关系

如果你仔细地研究`classtools`模块的self测试代码，将会注意到，其类只显示**实例属性**，属性附加到继承树底部的self对象；这就是self的`__dict__`所包含的内容。我们有意地不查看实例从其树上的类那里继承来的属性（例如，这个文件的self测试代码中的`count`）。继承的类属性只是附加到了类，而没有向下复制到实例。

如果你确实也想要包含继承属性，可以把`__class__`链接爬升到实例的类，使用这里的`__dict__`去获取类属性，然后迭代类的`__bases__`属性爬升到甚至更高的超类（有必要的还可以重复此过程）。如果你喜欢简单的代码，在实例上运行一个内置的`dir`调用而不是使用`__dict__`并爬升，二者有同样的效果，因为`dir`结果在排序的结果列表中包含了继承的名称：

```
>>> from person import Person
>>> bob = Person('Bob Smith')

# In Python 2.6:

>>> bob.__dict__.keys()           # Instance attrs only
['pay', 'job', 'name']

>>> dir(bob)                      # + inherited attrs in classes
['__doc__', '__init__', '__module__', '__str__', 'giveRaise', 'job',
'lastName', 'name', 'pay']

# In Python 3.0:

>>> list(bob.__dict__.keys())      # 3.0 keys is a view, not a list
['pay', 'job', 'name']

>>> dir(bob)                      # 3.0 includes class type methods
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__le__',
...more lines omitted...
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'giveRaise', 'job', 'lastName', 'name', 'pay']
```

这里的输出在Python 2.6和Python 3.0中有所不同，因为Python 3.0的`dict.keys`不是一个列表，并且Python 3.0的`dir`返回了确切的类类型实现属性。从技术上讲，Python 3.0的`dir`返回的更多，因为类都是“新式”的并且从类类型那里继承了很大一组运算符重载名称。实际上，你可能想要过滤掉Python 3.0的`dir`结果中的大多数`__X__`名称，因为它们都是内部实现细节，而不是我们通常想要显示的内容。

为了充分利用空间，作为现在建议的实验，我们将使用树爬升或`dir`来可选地显示继承的类属性。要了解关于这其中的更多内容，请查看我们在第28章编写的`classtree.py`继承树爬升器以及我们在第30章编写的`lister.py`属性列表器和爬升器。

工具类的命名考虑

这里最后需要考虑一点：由于`classtools`模块中的`AttrDisplay`类旨在和其他任意类混合的通用性工具，所以我们必须注意与客户类潜在的无意的命名冲突。为此，我们必须假设客户子类可能想要使用其`__str__`和`gatherAttrs`，但是，后者可能比一个子类的期

待还要多——如果一个子类无意地自己定义了一个`gatherAttrs`名称，它很可能会破坏我们的类，因为可能会使用子类中的这个低版本而不是我们的版本。

要自己看看效果，在文件的self测试代码中为`TopTest`添加一个`gatherAttrs`，除非新的方法是相同的，或者有意定制了最初的方法，我们的工具类将不再像计划的那样工作：

```
class TopTest(AttrDisplay):
    ....
    def gatherAttrs(self):                # Replaces method in AttrDisplay!
        return 'Spam'
```

这不一定是坏事，有时候，我们希望其他的方法可供子类使用，要么直接调用，要么定制。如果我们真的只想提供一个`__str__`，这还不够理想。

为了减少像这样的名称冲突的机会，Python程序员常常对于不想做其他用途的方法添加一个**单个下划线**的前缀：在我们的例子中就是`_gatherAttrs`。这不是很可靠（如果另一个也定义了`_gatherAttrs`，该如何是好？），但是，它通常已经够用了，并且对于类内部的方法，这是常用的Python命名惯例。

一种更好的但不太常用的方法是，只在方法名前面使用**两个下划线**符号：对我们的例子来说就是`__gatherAttrs`。Python自动扩展这样的名称，以包含类的名称，从而使它们变得真正唯一。这一功能通常叫做**伪私有类属性**，我们将在第30章展开讨论。现在，我们将两种方法都用到。

类的最终形式

现在，要在类中使用这一通用工具，所需要做的只是从其模块导入它，使用继承将其混合到顶层类中，并且删除掉我们之前编写的更专门的`__str__`方法。新的打印重载方法将会由`Person`的实例继承，`Manager`的实例也会继承；`Manager`从`Person`继承`__str__`，现在，`Person`从另一个模块的`AttrDisplay`获取它。下面是经过这些修改后的`person.py`文件的最终版本：

```
# File person.py (final)

from classtools import AttrDisplay          # Use generic display tool

class Person(AttrDisplay):
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                      # Assumes last is last
```

```

        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
        # Percent must be 0..1

class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)

```

在最后的版本中，我们添加了一些新的注释来记录所做的工作和每个最佳实践惯例——使用了功能性描述的文档字符串和用于简短注释的#。现在运行这段代码，将会看到对象的所有属性，而不只是在最初的`__str__`中直接编码的那些属性。并且，我们最终的问题也解决了：由于`AttrDisplay`直接从`self`实例中提取了类名，所以每个对象都显示其最近的（最低的）类的名称——`tom`现在显示为`Manager`，而不是`Person`。并且我们最终验证了其工作名称已经由`Manager`的构造函数正确地填充了：

```

C:\misc> person.py
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Tom Jones, pay=60000]

```

这正是我们所追求的更有用的显示。从更大的角度来看，我们的属性显示类已经变成了一个**通用工具**，可以通过继承将其混合到任何类中，从而利用它所定义的显示格式。此外，其所有客户都将自动获取我们工具中未来的修改。在本书稍后，我们将遇到甚至更加强大的类工具概念，例如，装饰器和元类。配合Python的内省工具，它们允许我们编写代码，从而以结构化的和可维护的方式来扩展和管理类。

步骤7（最后一步）：把对象存储到数据库中

此时，我们的工作几乎完成了。现在，我们有一个两模块的系统，不仅实现了显示人员的最初设计目标，而且提供了一个通用的属性显示工具，可供在未来的其他程序中使用。通过编写模块文件中的函数和类，我们已经确保了它们自然地支持重用。并且，通过把软件编写为类，我们已经确保了它们自然地支持扩展。

尽管我们的类按照计划工作，但是，它们创建的对象还不是真正的数据库记录。也就是说，如果我们关闭Python，实例也将消失——它们是内存中的临时性对象，而没有存储到文件这样的更为持久的媒介中，因此，未来程序运行的时候，它们不再可用。事实证明，让实例对象更加持久是很容易做到的，使用Python一项叫做**对象持久化**的功能——让对象在创建它们的程序退出之后依然存在。最为本教程的最后一步，让我们把对象持久化。

Pickle和Shelve

对象持久化通过3个标准的库模块来实现，这3个模块在Python中都可用：

`pickle`

任意Python对象和字节串之间的序列化

`dbm`（在Python 2.6中叫做`anydbm`）

实现一个可通过键访问的文件系统，以存储字符串

`shelve`

使用另两个模块按照键把Python对象存储到一个文件中

我们在第9章中学习文件基础知识的时候简单介绍过这些模块。它们提供了强大的数据存储选项。尽管我们不能在本教程或本书中完全介绍它们，它们还是很简单，一个简单的介绍就足够让你入门了。

`pickle`模块是一种非常通用的对象格式化和解格式化工具：对于内存中几乎任何的Python对象，它都能聪明地把对象转换为字节串，这个字节串可以随后用来在内存中重新构建最初的对象。`pickle`模块几乎可以处理我们所能够创建的任何对象，包括列表、字典、嵌套组合以及类实例。后者对于`pickle`来说特别有用，因为它们提供了数据（属性）和行为（方法），实际上，组合几乎等于“记录”和“程序”。由于`pickle`如此通用，所以我们可以不用编写代码来创建和解析对象的定制文本文件表示，它可以完全替代这些代码。通过在文件中存储一个对象的`pickle`字符串，我们可以有效地使其持久化：随后直接载入它并进行`unpickle`操作，就可以重新创建最初的对象。

尽管使用pickle本身把对象存储为简单的普通文件并随后载入它们是很容易的，但shelve模块提供了一个额外的层结构，允许按照键来存储pickle处理后的对象。Shelve使用pickle把一个对象转换为其pickle化的字符串，并将其存储在一个dbm文件中的键之下；随后载入的时候，shelve通过键获取pickle化的字符串，并用pickle在内存中重新创建最初的对象。这都很有技巧，但是，对于脚本来说，一个shelve的pickle化的对象看上去就像是字典^{注1}——我们通过键索引来访问、指定键来存储，并且使用len、in和dict.keys这样的字典工具来获取信息。Shelve自动把字典操作映射到存储在文件中的对象。

实际上，对于脚本来说，一个shelve和一个常规的字典之间唯一的编码区别就是，一开始必须打开shelve并且在修改之后必须关闭它。实际的效果是，一个shelve提供了一个简单的数据库来按照键存储和获取本地的Python对象，并由此使它们跨程序运行而保持持久化。它不支持SQL这样的查询工具，并且它缺乏在企业级数据库中可用的某些高级功能（例如，真正的事务处理），但是，一旦使用键获取了存储在shelve中的本地Python对象，就可以使用Python语言的所有功能来处理它。

在shelve数据库中存储对象

Pickle和shelve都是高级话题，并且我们不会在这里介绍其所有的细节。可以在标准库手册中找到更多相关的内容，或者像*Programming Python*这样专注于应用程序的图书中也有介绍。用Python说明它们比用语言文字说明要简单得多，因此，让我们来看一些代码。

让我们编写一个新的脚本，把类的对象存储到shelve中。在文本编辑器中，打开一个名为*makedb.py*的新文件。既然这是一个新文件，我们需要导入类以便创建一些实例来存储。我们像前面一样在交互提示模式下使用from载入一个类，但实际上，和函数与其他变量一样，有两种方式从一个文件载入一个类（类名和其他的名字一样也是变量，并且没什么特别的）：

```
import person                    # Load class with import
bob = person.Person(...)        # Go through module name

from person import Person       # Load class with from
bob = Person(...)               # Use name directly
```

我们使用from载入脚本中，只是因为它便于录入。在新的脚本中，复制或重新录入这些代码来创建类的实例，从而可以有些内容来存储（这是一个简单的示例程序，因此，我

注1： 没错，在Python中，我们把“shelve”用作一个名词，这使得多年来和我合作的编辑以及我所使用的电子文件编辑器懊恼不已。

们在这里不考虑测试代码的冗余性)。一旦有了一些实例,将它们存储到shelve中简直是小菜一碟。我们直接导入shelve模块,用一个外部文件名打开一个新的shelve,把对象赋给shelve中的键,当我们操作完毕之后关闭这个shelve,因为已经做过了修改:

```
# File makedb.py: store Person objects on a shelve database

from person import Person, Manager          # Load our classes
bob = Person('Bob Smith')                   # Re-create objects to be stored
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')                # Filename where objects are stored
for object in (bob, sue, tom):              # Use object's name attr as key
    db[object.name] = object                # Store object on shelve by key
db.close()                                  # Close after making changes
```

注意我们是如何把对象的名字用做键,从而把它们赋给shelve的。这么做只是为了方便,在shelve中,键可以是任何的字符串,包括我们使用诸如处理ID和时间戳(可以在os中和time标准库模块中使用)的工具所创建的唯一的字符串。唯一的规则是,键必须是字符串并且应该是唯一的,这样,我们就可以针对每个键只存储一个对象(尽管对象可以是包含很多对象的一个列表或字典)。然而,我们存储在键之下的值可以是几乎任何类型的Python对象:像字符串、列表和字典这样的内置对象,用户定义的类实例,以及所有这些嵌套式的组合。就是这些,如果这段脚本运行的时候没有输出,意味着它可能有效;我们没有打印任何内容,只是创建和存储对象:

```
C:\misc> makedb.py
```

交互地探索shelve

此时,当前的目录下会有一个或多个真实的文件,它们的名字都以“persondb”开头。实际创建的文件可能根据每个平台而有所不同,与在内置的open函数中一样,shelve.open()中的文件名也是相对于当前工作目录的,除非它包含了一个目录路径。不管文件存储在哪里,这些文件实现为一个通过键访问的文件,其中包含了我们的3个Python对象的pickle化的表示。不要删除这些文件,它们是你的数据库,并且是我们备份或移动存储的时候需要复制和转移的内容。

如果愿意的话,可以查看shelve的文件,从Windows Explorer或Python shell都可以看到,但是,它们是二进制散列文件,并且大多数内容对于shelve模块以外的环境没有太大意义。安装了Python 3.0并且没有安装额外的软件,我们的数据库存储在3个文件中(在Python 2.6中,它只是一个文件persondb,因为bsddb扩展模块在Python中为shelve预安装了;在Python 3.0中,bsddb是一个第三方开源插件):


```

# Directory listing module: verify files are present

>>> import glob
>>> glob.glob('person*')
['person.py', 'person.pyc', 'persondb.bak', 'persondb.dat', 'persondb.dir']

# Type the file: text mode for string, binary mode for bytes

>>> print(open('persondb.dir').read())
'Tom Jones', (1024, 91)
...more omitted...

>>> print(open('persondb.dat', 'rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00payq\x03K...'
...more omitted...

```

这些内容并非无法解读，但是它们在不同的平台上有所不同，并且无法确切地等同于一个用户友好的数据库界面！要更好地验证我们的工作，可以编写另外一个脚本，或者在交互模式下浏览shelve。由于shelve是包含了Python对象的Python对象，所以我们可以用常规的Python语法和开发模式来处理它。这里，交互提示模式有效地成为一个数据库客户端：

```

>>> import shelve
>>> db = shelve.open('persondb')                                # Reopen the shelve

>>> len(db)                                                       # Three 'records' stored
3

>>> list(db.keys())                                               # keys is the index
['Tom Jones', 'Sue Jones', 'Bob Smith']                          # list to make a list in 3.0

>>> bob = db['Bob Smith']                                         # Fetch bob by key
>>> print(bob)                                                    # Runs __str__ from AttrDisplay
[Person: job=None, name=Bob Smith, pay=0]

>>> bob.lastName()                                               # Runs lastName from Person
'Smith'

>>> for key in db:                                               # Iterate, fetch, print
    print(key, '=>', db[key])

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

>>> for key in sorted(db):
    print(key, '=>', db[key])                                     # Iterate by sorted keys

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

```

注意，在这里，为了载入或使用存储的对象，我们不一定必须导入Person或Manager类。例如，我们可以自由地调用bob的lastName方法，并且自动获取其定制的打印显示

格式，即便在我们的作用域中没有Person类。这之所以会起作用，是因为Python对一个类实例进行pickle操作，它记录了其self实例属性，以及实例所创建于的类的名字和类的位置。当随后从shelve中获取bob并对其unpickle的时候，Python将自动地重新导入该类并且将bob连接到它。

这种方法结果就是，类实例在未来导入的时候，会自动地获取其所有的类行为。只有在创建新实例的时候，我们才必须导入自己的类，而不是处理已有实例的时候也要这么做。尽管这是一项成熟的功能，但这个方案是多方综合结果：

- **缺点是：**当随后载入一个实例的时候，类及其模块的文件都必须导入。更正式地说，可以pickle的类必须在一个模块文件的顶部编码，而这个模块文件可以通过sys.path模块的查找路径所列出的目录来访问（并且，该模块文件不该在大多数脚本文件的模块__main__中，除非它们在使用的时候总是位于该模块中）。由于这一外部模块文件的需求，因此一些应用程序选择pickle更简单的对象，例如，字典或列表，特别是如果它们要通过Internet传送的时候。
- **好处是，**当该类的实例再次载入的时候，对类的源代码文件的修改会自动选取；这往往不需要更新存储的对象本身，因为更新它们的类代码就会改变它们的行为。

Shelve还有众所周知的限制（本章末尾的数据库建议会提到其中的一些）。然而，对于简单的对象存储，shelve和pickle是非常易于使用的工具。

更新shelve中的对象

现在来介绍最后一段脚本：让我们编写一个程序，在每次运行的时候更新一个实例（记录），以证实此时我们的对象真的是**持久的**（例如，每次一个Python程序运行的时候，它们的当前值都是可用的）。如下的文件updatedb.py打印出数据库，并且每次把我们所存储的对象之一增加一次。如果跟踪这里所发生的事情，你会注意到，我们发现可以“免费”地使用很多工具——自动使用通用的__str__重载方法打印对象，调用giveRaise方法增加之前写入的值。这些对基于OOP继承模型上的对象“就有效了”，即便当它们位于一个文件中：

```
# File updatedb.py: update Person object on database

import shelve
db = shelve.open('persondb')

for key in sorted(db):
    print(key, '\t=>', db[key])

sue = db['Sue Jones']
sue.giveRaise(.10)
db['Sue Jones'] = sue

# Reopen shelve with same filename
# Iterate to display database objects
# Prints with custom format
# Index by key to fetch
# Update in memory using class method
# Assign to key to update in shelve
```

```
db.close()
```

```
# Close after making changes
```

由于这段脚本启动的时候会打印数据库，我们必须运行它几次才能看到对象的改变。如下是它的运行情况，显示出所有的记录并且每次运行的时候增加了sue的pay（这对sue来说真是一段不错的脚本……）

```
c:\misc> updatedb.py
Bob Smith      => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones      => [Person: job=dev, name=Sue Jones, pay=100000]
Tom Jones      => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith      => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones      => [Person: job=dev, name=Sue Jones, pay=110000]
Tom Jones      => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith      => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones      => [Person: job=dev, name=Sue Jones, pay=121000]
Tom Jones      => [Manager: job=mgr, name=Tom Jones, pay=50000]

c:\misc> updatedb.py
Bob Smith      => [Person: job=None, name=Bob Smith, pay=0]
Sue Jones      => [Person: job=dev, name=Sue Jones, pay=133100]
Tom Jones      => [Manager: job=mgr, name=Tom Jones, pay=50000]
```

再一次，我们在这里看到了一个从Python中得到的成品的shelve和pickle工具，并且它具备我们自己在类中编写的行为。再一次，我们可以在交互模式（shelve的同等的数据库客户端）中验证脚本的作用：

```
c:\misc> python
>>> import shelve
>>> db = shelve.open('persondb')           # Reopen database
>>> rec = db['Sue Jones']                   # Fetch object by key
>>> print(rec)
[Person: job=dev, name=Sue Jones, pay=146410]
>>> rec.lastName()
'Jones'
>>> rec.pay
146410
```

本书第30章中的方框“为什么要在意：类和持续性”部分给出了另一个对象持久化的例子。它使用pickle而不是shelve在一个普通文件中存储了一个更大的复合对象，但是，效果是类似的。要了解有关pickle和shelve的更多细节，参见其他图书或者Python的手册。

未来方向

本教程到此结束了。此时，我们已经看到Python的OOP的所有基本机制的实际运作，并

且，我们已经学习了在代码中避免冗余性及其相关可维护性问题的方法。我们还构建了功能完备的类来完成实际的工作。此外，我们还通过把对象存储到Python的shelve中创建了正式的数据库记录，从而使它们的信息持久地存在。

当然，还有更多的内容可以探讨。例如，我们可以扩展类使它们更加实际，为它们添加新的行为，等等。例如，涨工资实际上应该验证工资增长比率在0到1之间，这是当我们在本书后面遇到装饰器的时候要添加的一个扩展。可能还会通过修改存储在对象中的状态信息以及用来处理对象的类方法，从而把这个例子变成一个个人联络数据库。我们还将给出一些建议的练习，帮助你开拓想象力。

我们还扩展了使用工具的范围，包括Python附带的工具以及开源世界中可以免费获取的工具：

GUI

目前，我们只能够使用交互提示的基于命令行的界面来处理数据库和脚本。我们还应该继续扩展对象数据库的易用性，添加一个图形化的用户界面来浏览和更新数据库记录。可以构建能够移植到Python的tkinter（在Python 2.6中是Tkinter）的GUI，或者可以移植到WxPython和PyQt这样的第三方工具的GUI。tkinter是Python自带的，允许我们快速地构建简单的GUI，并且是学习GUI编程技巧的理想工具；WxPython和PyQt使用起来更加复杂，但是往往能创建更高级的GUI。

Web站点

尽管GUI方便而且很快，但Web在易用性方面胜出。我们也可以实现一个Web站点来浏览和更新记录，而不是使用GUI和交互式提示。Web站点可以用Python自带的基本CGI脚本编程工具来构建，也可以用像Django、TurboGears、Pylons、web2Py、Zope或Google's App Engine这样的全功能的第三方Web开发框架来完成。在Web上，我们的数据可以仍然存储在shelve、pickle文件或其他基于Python的媒介中；处理它的脚本直接自动在服务器上运行，以响应来自Web浏览器和其他客户端的请求，并且它们生成HTML来与一个用户交互，而不是直接或通过框架API与用户交互。

Web服务

尽管Web客户端往往可以解析来自Web站点的回复中的信息（这种技术叫做屏幕抓取），我们还是应该更进一步提供一种更直接的方式来从Web获取记录：通过像SOAP或XML-RPC这样的Web服务接口来调用Python自身或第三方开源域所支持的API。这样的API以一种更加直接的形式返回数据，而不是嵌入一个回应页面的HTML中。

数据库

如果数据库变得更大或者更关键，我们可能甚至将其从shelve转移到像开源的ZODB面向对象数据库系统（OODB）这样的一个功能更完备的存储机制中，或者是像MySQL、Oracle、PostgreSQL或SQLite这样的一个更传统的基于SQL的数据库系统中。Python自身带有一个内置的、正在使用SQLite数据库，但是其他的开源选项也可以从Web上免费获得。例如，ZODB类似于Python的shelve，但是它解决了很多局限性，支持较大的数据库、并发更新、事务处理和内存中修改自动写入。像MySQL这样基于SQL的系统，为数据库存储提供了企业级的工具，并且可以在一个Python脚本中直接使用。

ORM

如果我们真的迁移到关系数据库中进行存储，不一定要牺牲Python的OOP工具。像SQLObject和SQLAlchemy这样的对象关系映射器（ORM），可以自动实现关系表和行与Python的类和实例之间的映射，这样一来，我们就可以使用常规的Python类语法来处理存储的数据。这种方法为shelve和ZODB提供了一个到OODB的替代，使得可以利用关系数据库和Python的类模型的双重威力。

尽管我希望这个介绍能够提起你进一步学习的胃口，但所有这些主题都很大程度上超出了本教程和本书的范围。如果你想要自己探究任何一个主题，查阅Web资料、Python的标准库手册以及像*Programming Python*这样关注应用程序的图书。在本书稍后，我挑选一个例子，介绍如何在数据库之上添加一个GUI和一个Web站点，从而允许浏览和更新实例记录。我希望最终在那里见到你，但是，让我们先返回到类的基础知识并完成核心Python语言的其他内容。

本章小结

在本章中，我们通过一步一步地构建一个简单但真实的实例，介绍了Python类和OOP的所有基础知识的实际应用。我们添加了构造函数、方法、运算符重载、子类定制和内省工具，并且我们一路上还介绍了其他的概念（例如复合、委托和多态）。

最后，我们通过类创建了对象，并且将它们存储到一个shelve对象数据库中以使其持久化，这是一种易于使用的系统，可以按照键来保存和获取本地Python对象。在探讨类基础知识的同时，我们还介绍了多种方式来构建代码以减少冗余性和最小化的未来维护代价。最后，我们简单地介绍了使用GUI和数据库这样的应用程序工具来扩展代码的方式，这些将在后续的图书中介绍。

在下一章中，我们将返回去学习Python的类模型背后的细节，并且展示其对于一些设计概念的应用，这些概念用来在较大的程序中组合类。在继续学习之前，让我们通过本章

的测试，并回顾本章介绍的内容。既然已经在本章中做了很多动手的工作，我们将通过一组主要面向理论设计的问题来帮助你回顾一些代码并思考其背后的重要思路。

本章习题

1. 当我们从shelve获取一个Manager对象并打印它的时候，显示格式逻辑来自何处？
2. 当我们从一个shelve获取一个Person对象而没有导入其模块的时候，该对象如何知道它有一个giveRaise方法可供我们调用？
3. 为什么把处理放入方法中而不是在类之外硬编码如此重要？
4. 为什么通过子类而不是复制并修改最初的代码来定制会更好？
5. 为什么回调一个超类的方法来运行默认操作而不是在子类中复制和修改其代码要更好？
6. 为什么使用__dict__这样的工具来允许一般性地处理对象，而不是为类的每个类型编写更多定制代码要更好？
7. 一般来说，何时可以选择使用对象嵌入和组合而不是继承？
8. 如何修改本章中的类，从而在Python中实现一个个人联络信息数据库？

习题解答

1. 在类的最终版本中，Manager最终从单独的classtools模块的AttrDisplay继承其__str__打印方法。Manager自己没有一个这样的方法，因此，继承查找爬升到其Person超类；由于那里也没有__str__，查找继续向上爬升，并在AttrDisplay中找到它。类语句的标题行中的圆括号中列出的类名，提供了到更高的超类的链接。
2. 当实例稍后载入内存中的时候，shelve（实际上，它们使用pickle模块）自动地把该实例重新连接到它创建自的类。Python从其模块内部重新导入该类，创建一个带有其存储的属性的实例，并且把实例的__class__连接设置到指向其最初的类。通过这种方式，载入实例自动获取所有其他最初方法（如lastName、giveRaise和__str__），即便我们没有把实例的类导入我们的作用域中。
3. 把处理放入方法中很重要，这样一来，未来只有一个副本需要修改，并且方法可以在任何实例之上运行。这就是Python封装的概念，把逻辑封装到接口背后，更好地支持未来的代码维护。如果没有这么做，就会产生代码冗余性，将来代码修改的时候工作就会加倍。
4. 用子类定制可以减少开发工作。在OOP中，我们通过定制已经做过的事情来编码，

而不是复制和修改已有的代码。这是OOP中真正的“大思路”，因为我们可以通过编写新的子类来很容易地扩展以前的工作，我们可以利用已经做过的事情。这比每次从头开始编写要好很多，也好过引入多个冗余的代码副本，它们未来可能全部都必须更新。

5. 不管是什么样的情况，复制和修改代码会使未来的潜在工作**翻倍**。如果一个子类需要执行超类方法中编写的默认行为，通过超类的名称回去调用最初的方法而不是复制其代码，这种做法要好很多。这对于超类的构造函数也有效。再次强调，复制代码会产生冗余性，当代码改进的时候这是一个主要的问题。
6. 通用性工具可以避免硬编码解决方案，而后者必须随着时间推移和类的改进保持与类的其他部分同步。例如，一个通用的__str__打印方法，不需要在__init__构造函数中每次为实例添加一个新属性时都更新。此外，一个通用的打印方法只由所有出现的类继承，并且只需要在一处修改，即在通用版本中修改，从通用类继承的所有类都会选取它。再一次说明，删除代码**冗余性**会减少未来的开发工作；这是类带来的主要好处之一。
7. 与直接定制相比较（像Manager特化Person），继承是最佳的代码扩展。对于多个对象聚合到一个完整的对象，并且由一个控制器层类主导的情况，组合非常实用。继承向上传递调用以实现复用，组合向下传递以实现委托。继承和组合不是互斥的；嵌入一个控制器中的对象，往往其本身是基于继承来定制的。
8. 本章中的类可以用作样本代码来实现各种类型的数据库。基本上，我们可以修改构造函数来记录不同的属性，并提供各种适用于目标应用程序的方法，从而改变用途。例如，可以使用诸如name、address、birthday、phone、email等属性来构建一个联系人数据库，并且可以采用适合这一用途的方法。例如，调用名为sendmail的方法的时候，可能会使用Python标准库的smtp模块来自动向一个联络人发送邮件（参见Python手册或应用层级的图书来了解关于这样的工具的更多细节）。我们这里编写的AttrDisplay工具可能用来逐字打印对象，因为它有意设计为通用的。这里的大多数shelve数据库代码都可以用来存储对象，只需要稍作修改即可。

类代码编写细节

如果你还没有完全搞懂所有的Python OOP的内容，请别担心，我们已经很快浏览了这些内容，现在将会更深入地研究之前介绍过的概念。在这一章及以后的各章中，我们要从另一个角度看待类机制。这里，我们将继续学习类、方法和继承，正式讲解第26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，所以我们也要在这里总结Python中命名空间的概念。

下一章将继续深入回顾类机制，介绍一个更为特殊的方面：运算符重载。除了介绍细节，本章和下一章还会介绍一些比以前所见规模更大的类。

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着很大的不同。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，它会产生类对象，并把其引用值存储在前面所使用的变量名。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。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（第30章会进一步讨论）。以下是`class`语句的一般形式。

<code>class <name>(superclass,...):</code>	<i># Assign to name</i>
<code>data = value</code>	<i># Shared class data</i>
<code>def method(self,...):</code>	<i># Methods</i>
<code>self.member = value</code>	<i># Per-instance data</i>

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`语句是本地作用域，由内嵌的赋值语句建立的变量名，就存在于这个本地作用域内。
- 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，所以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()
>>> y = SharedData()
>>> x.spam, y.spam
(42, 42)                                     # Make two instances
                                           # They inherit and share 'spam'
```

在这里，因为变量名`spam`是在`class`语句的顶层进行赋值的，因此会附加在这个类中，从而为所有的实例共享。我们可通过类名称修改它，或者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31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^{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。

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）或其他事物（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（第29章将讨论这一内容）。

当创建这个类的实例的时候，变量名`data`会在构造函数方法内对`self.data`进行赋值运算，从而把`data`附加在这些实例上。

```
>>> x = MixedNames(1)           # Make two instance objects
>>> y = MixedNames(2)           # Each has its own data
>>> x.display(); y.display()     # self.data differs, MixedNames.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，而不是它的名称）。这个参数给方法提供了一个钩子，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，被由这个类创建的每个实例所继承。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
'class call'
```

```
# Instance changed again
```

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，就会得出错信息。

```
>>> NextClass.printer('bad call')
TypeError: unbound method printer() must be called with NextClass instance...
```

调用超类构造函数

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。常见的场景涉及了构造函数。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函数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函数。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31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，并且可以用来管理类数据。一个相关的概念，**类方法**，当调用的时候接受一个类而不是一个实例，并且它可以用来管理基于每个类的数据。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都行。

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30章以获得更多细节。

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，寻找所能找到的第一个`attr`。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28-1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。
- 超类的连接是通过class语句首行的括号内列出类而生成的。

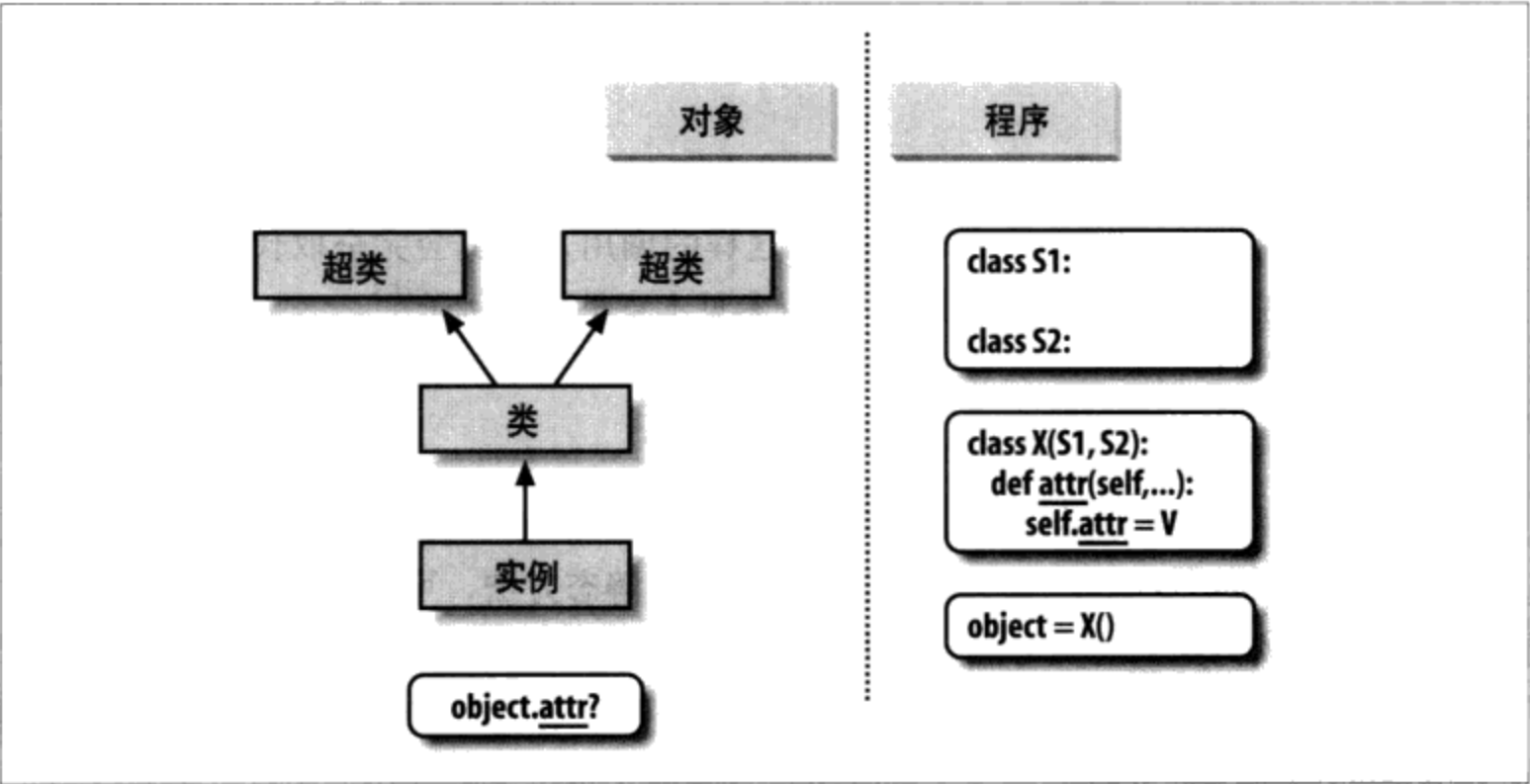


图28-1：程序代码会在内存中创建对象树，这个树是通过属性继承搜索的。调用类会创建记忆了这个类的新的实例。执行class语句会创建新的类，而列在class语句首行括号内的类则成为超类。即使`self`属性位于类的方法内每个属性引用，都会触发由下至上的树搜索

结果就是连接实例的属性命名空间树，到产生它的类、再到类首行中所列出的所有超

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, calls Super.method
starting Sub.method
in Super.method
ending Sub.method

```

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。在第29章学习__setattr__的时候，在第30章，当我们遇到__X名称的时候，以及在第38章再次遇到__X名称的时候（在那里，我们将用一个类装饰器来实现它），我们将更多地讨论属性名称私有性。

这种扩展编码模式常常用于构造函数。例如，参考本章之前的“方法”一节。

类接口技术

扩展只是一种与超类接口的方式。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`函数以及在子类中期待一个动作的`delegate`。

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')
    def delegate(self):
        self.action()

class Inheritor(Super):
    pass

class Replacer(Super):
    def method(self):
        print('in Replacer.method')

class Extender(Super):
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
```

```
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，在Provider子类中就会找到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超类——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（我们将在本书的下一部分中深入学习可能触发异常的语句）。作为提前介绍，下面是assert方法的实际应用示例：

```

class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!'           # If this version is called

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!

```

我们将会在第32章和第33章介绍`assert`。简而言之，如果其表达式运算结构为假，就会引发带有出错信息的异常。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，触发出错信息。此外，有些类只在该类的不完整方法中直接产生`NotImplemented`异常。

```

class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')

>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!

```

对于子类的实例，我们将得到异常，除非子类提供了期待的方法来替代超类中的默认方法：

```

>>> class Sub(Super): pass
...
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam

```

要查看这一节中概念的更实际的例子，可参考第31章结尾的习题8以及第六部分中的解答（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地将它去掉了。

Python 2.6和Python 3.0的抽象超类

在Python 2.6和Python 3.0中，前面小节的抽象超类（即“抽象基类”），需要由子类填充的方法，它们也可以以特殊的类语法来实现。我们编写代码的这种方法根据版本不同

而有所变化。在Python 3.0中，我们在一个class头部使用一个关键字参数，以及特殊的@装饰器语法，这二者我们都将在本书稍后更详细学习。

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):
        pass
```

但是在Python 2.6中，我们使用一个类属性：

```
class Super:
    __metaclass__ = ABCMeta
    @abstractmethod
    def method(self, ...):
        pass
```

不管哪种方法，效果都是相同的——我们不能产生一个实例，除非在类树的较低层级定义了该方法。例如，在Python 3.0中，与前一小节的例子等价的特殊语法如下：

```
>>> from abc import ABCMeta, abstractmethod
>>>
>>> class Super(metaclass=ABCMeta):
...     def delegate(self):
...         self.action()
...     @abstractmethod
...     def action(self):
...         pass
...
>>> X = Super()
TypeError: Can't instantiate abstract class Super with abstract methods action

>>> class Sub(Super): pass
...
>>> X = Sub()
TypeError: Can't instantiate abstract class Sub with abstract methods action

>>> class Sub(Super):
...     def action(self): print('spam')
...
>>> X = Sub()
>>> X.delegate()
spam
```

按照这种方式编写代码，带有一个抽象方法的类是不能继承的（即，我们不能通过调用它来创建一个实例），除非其所有的抽象方法都已经在子类中定义了。尽管这需要更多的代码，但这种方法的优点是，当我们试图产生该类的一个实例的时候，由于没有方法会产生错误，这不会比我们试图调用一个没有的方法更晚。这一功能可以用来定义一个期待的接口，在客户类中自动验证。

遗憾的是，这种方法也依赖于我们还没有介绍的两种高级语言工具——即第31章将要简介、第38章将深入的，以及第31章提及、第39章将要深入介绍的元类声明。因此，我们将在这里省略该选项的其他方面。参见Python的标准手册了解更详细的内容，并且可以查阅Python提供的预编码的抽象超类。

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17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。

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。

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命名空间的涅槃重生。当然,另一条达到涅槃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)            # FAILS: only visible in method
#print(g.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。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__构造函数内进行赋值的，但这并不是唯一的选择。

最后，正如我们在第17章所了解到的，一个函数在其外部修改名称也是可能的，使用global和（Python 3.0中的）nonlocal语句——这些语句提供了写入访问，但是也修改了赋值的命名空间绑定规则：

```

X = 11                  # Global in module

def g1():
    print(X)            # Reference global in module

```

```

def g2():
    global X
    X = 22                                # Change global in module

def h1():
    X = 33                                # Local in function
    def nested():
        print(X)                          # Reference local in enclosing scope

def h2():
    X = 33                                # Local in function
    def nested():
        nonlocal X                        # Python 3.0 statement
        X = 44                            # Change local in enclosing scope

```

当然，通常来说，在脚本内每个变量都不应该使用相同的变量名！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链接的字典而已。实际上，实例和类对象就是Python中带有链接的字典而已。Python暴露这些字典，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方式。在第26章中，我们看到了这种类型的代码的一个简单版本，既然我们已经了解了方法和超类，让我们在这里进一步介绍它。首先，我们定义一个超类和一个带方法的子类，而这些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__`属性链接到了它的类，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（我们将在Python 3.0下运行这段代码，名称格式和一些内部属性在Python 2.6中略有不同）。

```

>>> X = sub()
>>> X.__dict__
{}
# Instance namespace dict

>>> X.__class__
<class '__main__.sub'>
# Class of instance

>>> sub.__bases__
(<class '__main__.super'>,)
# Superclasses of class

>>> super.__bases__
(<class 'object'>,)
# () empty tuple in Python 2.6

```

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__.keys()
['__module__', '__doc__', 'hola']

>>> super.__dict__.keys()
['__dict__', '__module__', '__weakref__', '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15章讨论过的文档字符串）。

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，每个实例都有独立的命名空间字典，一开始是空的，可以记录与同一个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，所以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')

>>> 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15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，在 Python 3.0 中，它包含了从所有类的隐含超类 `object` 类继承的名称^{注5}：

```
>>> X.__dict__, Y.__dict__
({'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}, {})
>>> list(X.__dict__.keys())
['data1', 'data3', 'data2']                                     # Need list in 3.0

# In Python 2.6:

>>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']

# In Python 3.0:

>>> dir(X)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'data1', 'data2', 'data3', 'hello', 'hola']

>>> dir(sub)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello', 'hola']

>>> dir(super)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...more omitted...
'hello'
]
```

注5：正如你所看到的，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，就像这里我们针对 Python 3.0 下的用户定义类的 `dir` 结果一样。一般而言，前后有双下划线的属性名称是解释器专属的属性。类型子类会在第31章再讨论。

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。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。

```
# classtree.py

"""
Climb inheritance trees using namespace links,
displaying higher superclasses with indentation
"""

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 %s' % 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：它会使用`__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。当使用递归的时候，一个函数的每个活动层级都获取本地作用域的自己的副本；在这里，这意味`cls`和`indent`在每个`classtree`层级都是不同的。

这个文件的大部分内容都是自我测试程序代码。在Python 3.0下独立执行时，会创建空的类树，从中产生两个实例，并打印其类树结构：

```
C:\misc> c:\python26\python classtree.py
```

```

Tree of <__main__.B instance at 0x02557328>
...B
.....A
Tree of <__main__.F instance at 0x02557328>
...F
.....D
.....B
.....A
.....C
.....A
.....E

```

在Python 3.0下运行时，包含了隐含object超类的树会自动添加到独立的类上，因为所有的类在Python 3.0中都是“新式的”（关于这一修改的更多介绍在第31章）：

```

C:\misc> c:\python30\python classtree.py
Tree of <__main__.B object at 0x02810650>
...B
.....A
.....object
Tree of <__main__.F object at 0x02810650>
...F
.....D
.....B
.....A
.....object
.....C
.....A
.....object
.....E
.....object

```

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。

```

C:\misc> c:\python30\python
>>> class Emp: pass
...
>>> class Person(Emp): pass
>>> bob = Person()

>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person object at 0x028203B0>
...Person
.....Emp
.....object

```

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。我们还会在本书第30章中的“多重继承：‘混合’

类”一节编写`lister.py`通用类显示工具的时候看到另一个例子，在那里，我们将扩展这一技术从而也在显示一个类树的每个对象中的属性。在本书的最后一个部分，我们将在大范围介绍Python工具构建的时候再次回顾这些工具，以编写实现属性私有性、参数验证等工具。尽管这并不适用于每个Python程序员，但了解这些内幕就能够使用强大的开发工具。

回顾文档字符串

上一小节的示例包含了其模块的一个文档字符串，但是，别忘了，文档字符串也可以用于类的部分。我们在第15章详细介绍了文档字符串，它是出现在各种结构的顶部的字符串常量，由Python在相应对象的`__doc__`属性自动保存。它适用于模块文件、函数定义，以及类和方法。

既然我们了解了有关类和方法的知识，如下的文件`docstr.py`提供了一个快速但全面的示例，来概括文档字符串可以在代码中出现的位置。所有的这些都可以是三重引号的块：

```
"I am: docstr.__doc__"

def func(args):
    "I am: docstr.func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"
    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
        pass
```

文档字符串的主要优点是，它们在运行时能够保持。因此，如果它们已经编写为文档字符串，可以用其`__doc__`属性来获取文档：

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'

>>> docstr.func.__doc__
'I am: docstr.func.__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__'
```

第15章讨论了PyDoc工具，该工具知道如何格式化报表中的所有这些字符串。如下是在Python 2.6下运行我们代码的情况（Python 3.0还显示从新式类模式的隐含`object`超

类继承来的额外属性，请自己在Python 3.0下运行它，关于其差别的更多讨论参见第31章）：

```
>>> help(docstr)
Help on module docstr:

NAME
  docstr - I am: docstr.__doc__

FILE
  c:\misc\docstr.py

CLASSES
  spam

  class spam
  | I am: spam.__doc__ or docstr.spam.__doc__
  |
  | Methods defined here:
  |
  | method(self, arg)
  | I am: spam.method.__doc__ or self.method.__doc__

FUNCTIONS
  func(args)
  I am: docstr.func.__doc__
```

文档字符串在运行时可用，但是，它们从语法上比#注释（它可以出现在程序中的任何地方）要缺乏灵活性。两种形式都是有用的工具，并且任何程序文档都是很好的（当然，只要它够准确）。作为首要的最佳实践规则是：针对功能性文档（你的对象做什么）使用文档字符串；针对更加微观的文档（令人费解的表达式是如何工作的）使用#注释。

类与模块的关系

让我们通过简单地比较本书的最后两个话题来结束本章，即模块和类。由于它们都与命名空间有关，其中的区别有点令人费解。简而言之：

- 模块
 - 是数据/逻辑包。
 - 通过编写Python文件或C扩展来创建。
 - 通过导入来使用。
- 类
 - 实现新的对象。

- 由class语句创建。
- 通过调用来使用。
- 总是位于一个模块中。

类也支持模块所不支持的额外功能，例如，运算符重载、多实例生成和继承。尽管类和模块都是命名空间，我们现在应该能够辨别它们是不同的事物。

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，并且把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、命名空间字典和链接，以及对超类方法和构造函数手动调用。

现在，我们已经学会了Python中编写类的机制，第29章要转入这些机制的一个特定方面：运算符重载。在我们探索常用设计模式之后，来看看类常用和组合来以优化代码重用的某些方法。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。

本章习题

1. 什么是抽象超类？
2. 当简单赋值语句出现在class语句顶层时，会发生什么？
3. 类为什么可能会需要手动调用超类中的__init__方法？
4. 怎样增强（而不是完全取代）继承的方法？

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作为客户端定义、可定制的运算的实现方法。
2. 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。
3. 如果类定义自身的__init__构造函数，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函数：树中最低的那个。超类的构造函数是通过类名称来调用，手动传入`self`实例：`Superclass.__init__(self, ...)`。

4. 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, ...)`。

运算符重载

本章继续深入介绍类机制，主要关注运算符重载。我们在上一章简单介绍过运算符重载；在本章中，我们将深入更多的细节并看一些常用的重载方法。尽管我们不会展示众多可用的运算符重载方法中的每一种，但我们在这里给出的代码已经足够覆盖Python这一类功能的所有可能性。

基础知识

实际上，“运算符重载”只是意味着在类方法中**拦截**内置的操作——当类的实例出现在内置操作中，Python自动调用你的方法，并且你的方法的返回值变成了相应操作的结果。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。
- 类也可重载打印、函数调用、属性点号运算等内置运算。
- 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。

换句话说，当类中提供了某个特殊名称的方法，在该类的实例出现在它们相关的表达式时，Python自动调用它们。正如我们已经学过的，运算符重载方法并非必需的，并且通常也不是默认的；如果你没有编写或继承一个运算符重载方法，只是意味着你的类不会支持相应的操作。然而，当使用的时候，这些方法允许类模拟内置对象的接口，因此表现得更一致。

构造函数和表达式：__init__和__sub__

让我们看一个简单的重载例子吧。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函数（`__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):
        self.data = start
    def __sub__(self, other):
        return Number(self.data - other)

>>> from number import Number
>>> X = Number(5)
>>> Y = X - 2
>>> Y.data
3
```

就像前边讨论过的一样，该代码中所见到的`__init__`构造函数是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。

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29-1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29-1：常见运算符重载方法

方法	重载	调用
<code>__init__</code>	构造函数	对象建立：X = Class(args)
<code>__del__</code>	析构函数	X对象收回
<code>__add__</code>	运算符+	如果没有 <code>__iadd__</code> ，X + Y, X += Y
<code>__or__</code>	运算符 （位OR）	如果没有 <code>__ior__</code> ，X Y, X = Y
<code>__repr__</code> , <code>__str__</code>	打印、转换	print (X)、repr(X)、 str(X)
<code>__call__</code>	函数调用	X(*args,**kargs)
<code>__getattr__</code>	点号运算	X.undefined
<code>__setattr__</code>	属性赋值语句	X.any = value
<code>__delattr__</code>	属性删除	del X.any
<code>__getattribute__</code>	属性获取	X.any

表29-1：常见运算符重载方法（续）

方法	重载	调用
<code>__getitem__</code>	索引运算	<code>X[key]</code> , <code>X[i:j]</code> , 没 <code>__iter__</code> 时的 <code>for</code> 循环和其他迭代器
<code>__setitem__</code>	索引赋值语句	<code>X[key] = value</code> , <code>X[i:j] = sequence</code>
<code>__delitem__</code>	索引和分片删除	<code>del X[key]</code> , <code>del X[i:j]</code>
<code>__len__</code>	长度	<code>len(X)</code> , 如果没有 <code>__bool__</code> , 真值测试
<code>__bool__</code>	布尔测试	<code>bool(X)</code> , 真测试（在Python 2.6中叫做 <code>__nonzero__</code> ）
<code>__lt__</code> , <code>__gt__</code> , <code>__le__</code> , <code>__ge__</code> , <code>__eq__</code> , <code>__ne__</code>	特定的比较	<code>X < Y</code> , <code>X > Y</code> , <code>X <= Y</code> , <code>X >= Y</code> , <code>X == Y</code> , <code>X != Y</code> (或者在Python 2.6中只有 <code>__cmp__</code>)
<code>__radd__</code>	右侧加法	<code>Other + X</code>
<code>__iadd__</code>	原地（增强的）加法	<code>X += Y</code> (or else <code>__add__</code>)
<code>__iter__</code> , <code>__next__</code>	迭代环境	<code>I=iter(X)</code> , <code>next(I)</code> ; <code>for</code> loops, <code>in</code> if no <code>__contains__</code> , <code>all</code> comprehensions, <code>map(F,X)</code> , 其他 (<code>__next__</code> 在Python 2.6中称为 <code>next</code>)
<code>__contains__</code>	成员关系测试	<code>item in X</code> (任何可迭代的)
<code>__index__</code>	整数值	<code>hex(X)</code> , <code>bin(X)</code> , <code>oct(X)</code> , <code>0[X]</code> , <code>0[X:]</code> (替代Python 2中的 <code>__oct__</code> 、 <code>__hex__</code>)
<code>__enter__</code> , <code>__exit__</code>	环境管理器 (参见第33章)	<code>with obj as var:</code>
<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>	描述符属性 (参见第37章)	<code>X.attr</code> , <code>X.attr = value</code> , <code>del X.attr</code>
<code>__new__</code>	创建 (参见第39章)	在 <code>__init__</code> 之前创建对象

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 `+`。

如果没有定义运算符重载方法的话，它可能继承自超类，就像任何其他的方法一样。运算符重载方法也都是可选的——如果没有编写或继承一个方法，你的类直接不支持这些运算，并且试图使用它们会引发一个异常。一些内置操作，如打印，有默认的重载方法

（继承自Python 3.0中隐含object类），但是，如果没有给出相应的运算符重载方法的话，大多数内置函数会对类实例失效。

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__`构造函数常出现在绝大多数类中。我们已见到过`__init__`初始设定构造函数，以及表29-1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。

索引和分片：`__getitem__`和`__setitem__`

如果在类中定义了（或继承了）的话，则对于实例的索引运算，会自动调用`__getitem__`。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 X.__getitem__(i)
4

>>> for i in range(5):
...     print(X[i], end=' ')               # Runs __getitem__(X, i) each time
...
0 1 4 9 16
```

拦截分片

有趣的是，除了索引，对于分片表达式也调用`__getitem__`。正式地讲，内置类型以同样的方式处理分片。例如，下面是在一个内置列表上工作的分片，使用了上边界和下边界以及一个stride（如果需要回顾关于分片的知识，请参阅第7章）：

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                                  # Slice with slice syntax
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[:2]
[5, 6]
>>> L[::2]
[5, 7, 9]
```

实际上，分片边界绑定到了一个分片对象中，并且传递给索引的列表实现。实际上，我们总是可以手动地传递一个分片对象——分片语法主要是用一个分片对象进行索引的语法糖：


```
>>> L[slice(2, 4)]           # Slice with slice objects
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, -1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

对于带有一个`__getitem__`的类，这是很重要的——该方法将既针对基本索引（带有一个索引）调用，又针对分片（带有一个分片对象）调用。我们前面的类没有处理分片，因为它的数学假设传递了整数索引，但是，如下类将会处理分片。当针对索引调用的时候，参数像前面一样是一个整数：

```
>>> class Indexer:
...     data = [5, 6, 7, 8, 9]
...     def __getitem__(self, index):           # Called for index or slice
...         print('getitem:', index)
...         return self.data[index]           # Perform index or slice
...
>>> X = Indexer()
>>> X[0]           # Indexing sends __getitem__ an integer
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
```

然而，当针对分片调用的时候，方法接收一个分片对象，它在一个新的索引表达式中直接传递给嵌套的列表索引：

```
>>> X[2:4]           # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

如果使用的话，`__setitem__`索引赋值方法类似地拦截索引和分片赋值——它为后者接收了一个分片对象，它可能以同样的方式传递到另一个索引赋值中：

```
def __setitem__(self, index, value):           # Intercept index or slice assignment
```

```
...
self.data[index] = value                # Assign index or slice
```

实际上，`__getitem__`可能在甚至比索引和分片更多的环境中自动调用，正如下面的小节所介绍的。

Python 2.6中的分片和索引

在Python 3.0之前，类也可以定义`__getslice__`和`__setslice__`方法来专门拦截分片获取和赋值；它们将传递一系列的分片表达式，并且优先于`__getitem__`和`__setitem__`用于分片。

这些特定于分片的方法已经从Python 3.0中移除了，因此，你应该使用`__getitem__`和`__setitem__`来替代，以考虑到索引和分片对象都可能作为参数。在大多数类中，这不需要任何特殊的代码就能工作，因为索引方法可以在另一个索引表达式的方括号中传递分片对象（就像我们的例子中那样）。参见本章后面的“成员关系：`__contains__`、`__iter__`和`__getitem__`”节中关于分片拦截应用的另一个示例。

此外，不要混淆了（无疑是一个不幸的名称）Python 3.0中用于索引拦截的`__index__`方法；需要的时候，该方法针对一个实例返回一个整数值，供转化为数字字符串的内置函数使用：

```
>>> class C:
...     def __index__(self):
...         return 255
...
>>> X = C()
>>> hex(X)                # Integer value
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

尽管这个方法并不会拦截像`__getitem__`这样的实例索引，但它也可以在需要一个整数的环境中应用——包括索引：

```
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]        # As index (not X[i])
'C'
>>> ('C' * 256)[X:]       # As index (not X[i:])
'C'
```

该方法在Python 2.6中以同样的方式工作，只不过它不会针对`hex`和`oct`内置函数调用（在Python 2.6中使用`__hex__`和`__oct__`来拦截这些调用）。

索引迭代：__getitem__

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__
'p'
>>> for item in X:                               # for loops call __getitem__
...     print(item, end=' ')                    # 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14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
True

>>> [c for c in X]                             # List comprehension
['S', 'p', 'a', 'm']

>>> list(map(str.upper, X))                     # map calls (use list() in 3.0)
['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 object 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31章扩展内置类型时，我们会再谈这个观点。

迭代器对象：__iter__和__next__

尽管上一节中的__getitem__技术有效，但它真的只是迭代的一种退而求其次的方法。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14章所学到的迭代协议，然后才是重复对对象进行索引运算。只有在对象不支持迭代协议的时候，才会尝试索引运算。一般来讲，你也应该优先使用__iter__，它能够比__getitem__更好地支持一般的迭代环境。

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（对于手动迭代来说，一个next内置函数也可以很方便地使用：next(I)与I.__next__()是相同的）。

注意：版本差异提示：正如第14章所述，如果你使用Python 2.6，刚刚所提到的I.__next__()在你的Python中叫做I.next()，而next(I)内置函数展现出了可移植性：它在Python 2.6中叫做I.next()，而在Python 3.0中叫做I.__next__()。迭代的所有其他方面在Python 2.6中都同样起作用。

用户定义的迭代器

在__iter__机制中，类就是通过实现第14章和第20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:         # Also called by next built-in
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):
...     print(i, end=' ')
...
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有自己的状态信息的对象，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（本书下一部分会谈及引发异常的内容）。手动迭代对内置类型也有效：

```
>>> X = Squares(1, 5)                                # Iterate manually: what loops do
>>> I = iter(X)                                       # iter calls __iter__
>>> next(I)                                           # next calls __next__
1
>>> next(I)
4
...more omitted...
>>> next(I)
25
>>> next(I)                                           # Can catch this in try statement
StopIteration
```

__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、类型构造函数、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。

```
>>> 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20章介绍过迭代器相关的主题），这个例子可能更简单一些。

```
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):
...     print(i, end=' ')
...
1 4 9 16 25
```

*# or: (x ** 2 for x in range(1, 5))*

和类不同的是，这个函数会自动在迭代中存储其状态。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, end=' ')
...
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，而不管其他激活状态下的循环是什么状态。

我们前面在第14章和第20章看到了相关的例子。例如，生成器函数和表达式，以及map和zip这样的内置函数，都证明是单迭代对象；相反，range内置函数和其他的内置类型（如列表），支持独立位置的多个活跃迭代器。

当我们用类编写用户定义的迭代器的时候，由我们来决定是支持一个单个的或是多个活跃的迭代。要达到多个迭代器的效果，__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迭代时都重新创建，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(next(I), next(I), next(I))      # 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, end=' ')        # 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。

```
>>> S = 'abcdef'
>>> for x in S[::2]:
...     for y in S[::2]:                 # New objects on each iteration
...         print(x + y, end=' ')
... 
```


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存储在内存中；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。为了更接近于类，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, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

这样与基于类的解决办法更相似一些，但是，它仍是一次性把分片结果存储在内存中（目前内置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。

成员关系：__contains__、__iter__和__getitem__

迭代器的内容比我们目前所见到的还要丰富。运算符重载往往是多个层级的：类可以提供特定的方法，或者用作退而求其次选项的更通用的替代方案。例如：

- Python 2.6中的比较使用__lt__这样的特殊方法来表示少于比较（如果有的话），或者使用通用的__cmp__。Python 3.0只使用特殊的方法，而不是__cmp__，如本章后面所介绍的。
- 布尔测试类似于先尝试一个特定的__bool__（以给出一个明确的True/False结果），并且，如果没有它，将会退而求其次到更通用的__len__（一个非零的长度意味着True）。正如我们将在本章随后见到的，Python 2.6也一样起作用，但是，使用名称__nonzero__而不是__bool__。

在迭代领域，类通常把in成员关系运算符实现为一个迭代，使用__iter__方法或__getitem__方法。要支持更加特定的成员关系，类可能编写一个__contains__方

法——当出现的时候，该方法优先于`__iter__`方法，`__iter__`方法优先于`__getitem__`方法。`__contains__`方法应该把成员关系定义为对一个映射应用键（并且可以使用快速查找），以及用于序列的搜索。

考虑如下的类，它编写了所有3个方法和测试成员关系以及应用于一个实例的各种迭代环境。调用的时候，其方法会打印出跟踪消息：

```
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):
        print('get[%s]:' % i, end='')
        return self.data[i]
    def __iter__(self):
        print('iter=> ', end='')
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):
        print('contains: ', end='')
        return x in self.data

X = Iters([1, 2, 3, 4, 5])
print(3 in X)
for i in X:
    print(i, end=' | ')

print()
print([i ** 2 for i in X])
print(list(map(bin, X)))

I = iter(X)
while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

这段脚本运行的时候，其输出如下所示——特定的`__contains__`拦截成员关系，通用的`__iter__`捕获其他的迭代环境以至`__next__`重复地被调用，而`__getitem__`不会被调用：

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

但是，要观察如果注释掉`__contains__`方法后代码的输出发生了什么变化——成员关系现在路由到了通用的`__iter__`：

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

最后，如果`__contains__`和`__iter__`都注释掉的话，其输出如下——索引`__getitem__`替代方法会被调用，针对成员关系和其他迭代环境使用连续较高的索引：

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100', '0b101']
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

正如我们所看到的，`__getitem__`方法甚至更加通用：除了迭代，它还拦截显式索引和分片。分片表达式用包含边界的一个分片对象来触发`__getitem__`，既针对内置类型，也针对用户定义的类，因此，我们的类中分片是自动化的：

```
>>> X = Iters('spam')                                # Indexing
>>> X[0]                                                # __getitem__(0)
get[0]:'s'

>>> 'spam'[1:]                                         # Slice syntax
'pam'
>>> 'spam'[slice(1, None)]                             # Slice object
'pam'

>>> X[1:]                                              # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[:-1]
get[slice(None, ?1, None)]:'spa'
```

然而，在并非面向序列的、更加现实的迭代用例中，`__iter__`方法可能很容易编写，因为它不必管理一个整数索引，并且`__contains__`考虑到作为一种特殊情况优化成员关系。

属性引用：`__getattr__`和`__setattr__`

`__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，就会用属性名称作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，所以`__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
...
>>> 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，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__`，而在第七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，要确定是通过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）。也就是说，是使用`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
```

有两个属性访问重载方法，允许我们控制或特化对对象中的属性的访问。它们倾向于扮

演高度专用的角色，其中的一些我们将在本书后面介绍。

其他属性管理工具

为了便于将来参考，还要注意，有其他的方式来管理Python中的属性访问：

- `__getattribute__`方法拦截所有的属性获取，而不只是那些未定义的，但是，当使用它的时候，必须比使用`__getattr__`更小心地避免循环。
- `Property`内置函数允许我们把方法和特定类属性上的获取和设置操作关联起来。
- **描述符**提供了一个协议，把一个类的`__get__`和`__set__`方法与对特定类属性的访问关联起来。

由于这些颇有些高级的工具并不是对每个Python程序员都有用，所以我们将推迟到本书第31章再介绍这些特性，并且在第37章再详细地介绍所有属性管理技术。

模拟实例属性的私有性：第一部分

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列表，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其主要的目的。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。

我们将推迟到第38章再给出属性私有性的一个更完整的解决方案，在那里，我们将使用类装饰器来更加通用地拦截和验证属性。即使私有性可以以此方式模拟，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：这是关于访问控制的一般意义上的、有趣的发现，超出了我们这里的介绍范围。

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（第30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__构造函数和__add__重载方法，本例也会定义返回实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。

实例对象的默认显示既无用也不好看：

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                # Initialize data
...     def __add__(self, other):
...         self.data += other               # Add other in-place (bad!)
...
>>> x = adder()                             # Default displays
>>> print(x)
<__main__.adder object at 0x025D66B0>
>>> x
<__main__.adder object at 0x025D66B0>
```

但是，编写或继承字符串表示方法允许我们定制显示：

```
>>> class addrepr(adder):                   # Inherit __init__, __add__
...     def __repr__(self):                 # Add string representation
...         return 'addrepr(%s)' % self.data # Convert to as-code string
...
>>> 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__ for both
('addrepr(3)', 'addrepr(3)')

```

那么，为什么要两个显示方法呢？概括地讲，是为了进行用户友好的显示。具体来说：

- 打印操作会首先尝试__str__和str内置函数（print运行的内部等价形式）。它通常应该返回一个用户友好的显示。
- __repr__用于所有其他的环境中：用于交互模式下提示回应以及repr函数，如果没有使用__str__，会使用print和str。它通常应该返回一个编码字符串，可以用来重新创建对象，或者给开发者一个详细的显示。

总而言之，__repr__用于任何地方，除了当定义了一个__str__的时候，使用print和str。然而要注意，如果没有定义__str__，打印还是使用__repr__，但反过来并不成立——其他环境，例如，交互式响应模式，只是使用__repr__，并且根本不要尝试__str__：

```

>>> 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 object at 0x00B35EF0>
>>> print(x)                                # Runs __str__
[Value: 4]
>>> str(x), repr(x)
('[Value: 4]', '<__main__.addstr object 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，通过分别定义这两个方法，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。实际上，__str__只是覆盖了__repr__以得到用户友好的显示环境：

```

>>> class addboth(adder):
...     def __str__(self):                    # User-friendly string
...         return '[Value: %s]' % self.data
...     def __repr__(self):                  # As-code string
...         return 'addboth(%s)' % self.data
...
>>> x = addboth(4)
>>> x + 1
>>> x                                         # Runs __repr__
addboth(5)
>>> print(x)                                # Runs __str__

```



```
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

我在这里应该提到两种用法。首先，记住`__str__`和`__repr__`都必须返回字符串；其他的结果类型不会转换并会引发错误，因此，如果必要的话，确保用一个转换器处理它们。其次，根据一个容器的字符串转换逻辑，`__str__`的用户友好的显示可能只有当对象出现在一个打印操作顶层的时候才应用，嵌套到较大对象中的对象可能用其`__repr__`或默认方法打印。如下代码说明了这两点：

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __str__(self):
...         return str(self.val)
...                                     # Used for instance itself
...                                     # Convert to a string result
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)
...                                     # __str__ run when instance printed
...                                     # But not when instance in a list!
2
3
>>> print(objs)
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
>>> objs
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...
```

为了确保一个定制显示在所有的环境中都显示而不管容器是什么，请编写`__repr__`，而不是`__str__`；前者在所有的情况下都运行，即便是后者不适用的情况也是如此：

```
>>> class Printer:
...     def __init__(self, val):
...         self.val = val
...     def __repr__(self):
...         return str(self.val)
...                                     # __repr__ used by print if no __str__
...                                     # __repr__ used if echoed or nested
...
>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)
...                                     # No __str__: runs __repr__
...
2
3
>>> print(objs)
...                                     # Runs __repr__, not __str__
[2, 3]
>>> objs
[2, 3]
```

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。在可以打印对象并且看见定制显示的任何时候，可能就是使用了这两个工具中的一个。

右侧加法和原处加法：__radd__和__iadd__

从技术方面来讲，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)
...         return self.val + other
...     def __radd__(self, other):
...         print('radd', self.val, other)
...         return other + self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> x + 1                                # __add__: instance + noninstance
add 88 1
89
>>> 1 + y                                # __radd__: noninstance + instance
radd 99 1
100
>>> x + y                                # __add__: instance + instance, triggers __radd__
add 88 <__main__.Commuter object at 0x02630910>
radd 99 88
187
```

注意，__radd__中的顺序与之相反：self是在+的右侧，而other是在左侧。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。当我们把两个实例相加的时候，Python运行__add__，它反过来通过简化左边的运算数来触发__radd__。

在更为实际的类中，其中类类型可能需要在结果中传播，事情可能变得更需要技巧：类型测试可能需要辨别它是否能够安全地转换并由此避免嵌套。例如，下面的代码中如果没有isinstance测试，当两个实例相加并且__add__触发__radd__的时候，我们最终得到一个Commuter，其val是另一个Commuter：

```
>>> class Commuter:                                # Propagate class type in results
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         if isinstance(other, Commuter): other = other.val
...         return Commuter(self.val + other)
...     def __radd__(self, other):
...         return Commuter(other + self.val)
```

```

...     def __str__(self):
...         return '<Commuter: %s>' % self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)                                # Result is another Commuter instance
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>

>>> z = x + y                                     # Not nested: doesn't recur to __radd__
>>> print(z)
<Commuter: 187>
>>> print(z + 10)
<Commuter: 197>
>>> print(z + z)
<Commuter: 374>

```

原处加法

为了也实现+=原处扩展相加，编写一个__iadd__或__add__。如果前者空缺的话，使用后者。实际上，前面小节的Commuter类为此已经支持+=了，但是，__iadd__考虑到了更加高效的原处修改：

```

>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __iadd__(self, other):                # __iadd__ explicit: x += y
...         self.val += other                    # Usually returns self
...         return self
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):                # __add__ fallback: x = (x + y)
...         return Number(self.val + other)      # Propagates class type
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7

```

每个二元运算都有类似的右侧和原处重载方法，它们以相同的方式工作（例如，__mul__，__rmul__和__imul__）。右侧方法是一个高级话题，并且在实际中很少用到；只有在需要运算符具有交换性的时候，才会编写它们，并且只有在真正需要支

持这样的运算符的时候，才会使用。例如，一个`Vector`类可能使用这些工具，但一个`Employee`或`Button`类可能不会。

Call表达式：__call__

当调用实例时，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 Callee:
...     def __call__(self, *pargs, **kargs):      # Intercept instance calls
...         print('Called:', pargs, kargs)      # Accept arbitrary arguments
...
>>> C = Callee()
>>> C(1, 2, 3)                                  # C is a callable object
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

更正式地说，我们在第18章介绍的所有参数传递方式，`__call__`方法都支持——传递给实例的任何内容都会传递给该方法，包括通常隐式的实例参数。例如，方法定义：

```
class C:
def __call__(self, a, b, c=5, d=6): ...          # Normals and defaults

class C:
def __call__(self, *pargs, **kargs): ...         # Collect arbitrary arguments

class C:
def __call__(self, *pargs, d=6, **kargs): ...    # 3.0 keyword-only argument
```

都匹配如下所有的实例调用：

```
X = C()
X(1, 2)                                          # Omit defaults
X(1, 2, 3, 4)                                  # Positionals
X(a=1, b=2, d=4)                               # Keywords
X(*[1, 2], **dict(c=3, d=4))                   # Unpack arbitrary arguments
X(1, *(2,), c=3, **dict(d=4))                  # Mixed modes
```

直接的效果是，带有一个`__call__`的类和实例，支持与常规函数和方法完全相同的参数语法和语义。

像这样的拦截调用表达式允许类实例模拟类似函数的外观，但是，也在调用中保持了状态信息以供使用（我们在本书第17章中介绍作用域的时候看到过一个类似的示例，但是，在这里，你应该对运算符重载更熟悉）：

```

>>> class Prod:
...     def __init__(self, value):           # Accept just one argument
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)                             # "Remembers" 2 in state
>>> x(3)                                    # 3 (passed) * 2 (state)
6
>>> x(4)
8

```

在这个示例中，`__call__`乍一看可能有点奇怪。一个简单的方法可以提供类似的功能：

```

>>> 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编写接口时，`__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__`构造函数以及`__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（在Python 2.6中是Tkinter）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，可记住稍后按下按钮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)

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，有关绑定方法可参考下一章的内容。

在第31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，所以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。

比较：`__lt__`、`__gt__`和其他方法

正如表29-1所示，类可以定义方法来捕获所有的6种比较运算符：`<`、`>`、`<=`、`>=`、`==`和`!=`。这些方法通常很容易使用，但是，记住如下的一些限制：

- 与前面讨论的`__add__`/`__radd__`不同，比较方法没有右端形式。相反，当只有一个运算数支持比较的时候，使用其对应方法（例如，`__lt__`和`__gt__`互为对应）。
- 比较运算符没有隐式关系。例如，`==`并不意味着`!=`是假的，因此，`__eq__`和`__ne__`应该定义为确保两个运算符都正确地作用。
- 在Python 2.6中，如果没有定义更为具体的比较方法的话，对所有比较使用一个`__cmp__`方法。它返回一个小于、等于或大于0的数，以表示比较其两个参数（`self`和另一个操作数）的结果。这个方法往往使用`cmp(x, y)`内置函数来计算其结果。`__cmp__`方法和`cmp`内置函数都从Python 3.0中删除了：使用更特定的方法来替代。

我们没有篇幅来深入介绍比较方法，但是，作为一个快速介绍，考虑如下的类和测试代码：

```
class C:
    data = 'spam'
    def __gt__(self, other):
        return self.data > other
    def __lt__(self, other):
        return self.data < other
X = C()
print(X > 'ham')
print(X < 'ham')
```

3.0 and 2.6 version

True (runs __gt__)

False (runs __lt__)

在Python 3.0和Python 2.6下运行的时候，末尾的打印语句显示它们的注释中提到的结果，因为该类的方法拦截并实现了比较表达式。

Python 2.6的 `__cmp__` 方法（已经从Python 3.0中移除了）

在Python 2.6中，如果没有定义更加具体的方法的话，`__cmp__`方法作为一种退而求其次

的方法：它的整数结果用来计算正在运行的运算符。例如，如下的代码在Python 2.6下产生同样的结果，但是在Python 3.0中失败，因为`__cmp__`不再可用：

```
class C:
    data = 'spam'
    def __cmp__(self, other):
        return cmp(self.data, other)

X = C()
print(X > 'ham')
print(X < 'ham')
```

2.6 only
__cmp__ not used in 3.0
cmp not defined in 3.0

True (runs __cmp__)
False (runs __cmp__)

注意，这在Python 3.0中失效是因为`__cmp__`不再特殊，而不是因为`cmp`内置函数不再使用。如果我们把前面的类修改为如下的形式，以试图模拟`cmp`调用，那么代码将在Python 2.6中工作，但在Python 3.0下无效：

```
class C:
    data = 'spam'
    def __cmp__(self, other):
        return (self.data > other) - (self.data < other)
```

注意：那么，你可能会问，为什么我只是给出在Python 3.0中不再支持的比较方法呢？尽管它可能很容易完全删除，但本书内容旨在支持Python 2.6和Python 3.0。由于`__cmp__`可能出现在读者必须重用或维护的Python 2.6代码中，因此它也是本书的目标。此外，`__cmp__`比前面所述的`__getslice__`方法更唐突地删除了，并且由此可能有更长时间的后遗症。然而，如果你使用Python 3.0，或者关心将来在Python 3.0下运行你的代码，不要再使用`__cmp__`了：使用更精确的比较方法。

布尔测试： `__bool__` 和 `__len__`

正如前面所提到的，类可能也定义了赋予其实例布尔特性的方法——在布尔环境中，Python首先尝试`__bool__`来获取一个直接的布尔值，然后，如果没有该方法，就尝试`__len__`类根据对象的长度确定一个真值。通常，首先使用对象状态或其他信息来生成一个布尔结果：

```
>>> class Truth:
...     def __bool__(self): return True
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!

>>> class Truth:
...     def __bool__(self): return False
...
>>>
```

```
>>> X = Truth()
>>> bool(X)
False
```

如果没有这个方法，Python退而求其次地求长度，因为一个非空对象看作是真（如，一个非零长度意味着对象是真的，并且一个零长度意味着它为假）：

```
>>> class Truth:
...     def __len__(self): return 0
...
>>> X = Truth()
>>> if not X: print('no!')
...
no!
```

如果两个方法都有，Python喜欢__bool__胜过__len__，因为它更具体：

```
>>> class Truth:
...     def __bool__(self): return True           # 3.0 tries __bool__ first
...     def __len__(self): return 0              # 2.6 tries __len__ first
...
>>> X = Truth()
>>> if X: print('yes!')
...
yes!
```

如果没有定义真的方法，对象毫无疑问地看作为真：

```
>>> class Truth:
...     pass
...
>>> X = Truth()
>>> bool(X)
True
```

既然我们已经尝试突入哲学的领域，让我们进一步来看看最后一种重载环境：对象转让。

Python 2.6中的布尔

Python 2.6用户应该在“布尔测试：__bool__和__len__”节的所有代码中使用__nonzero__而不是__bool__。Python 3.0把Python 2.6的__nonzero__方法重新命名为__bool__，但布尔测试以相同的方式工作（Python 3.0和Python 2.6都使用__len__作为候补）。

如果你没有使用Python 2.6的名称，本节中的第一个测试将会同样地工作，但是，仅仅因为__bool__在Python 2.6中没有识别为一个特殊的方法名称，并且对象默认看作是真的！

要见证这个版本的不同之处，你需要返回False：

```
C:\misc> c:\python30\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
in bool
False
>>> if X: print(99)
...
in bool
```

这在Python 3.0中像宣传的那样有效。然而，在Python 2.6中，`__bool__`被忽视并且对象总是看作是真：

```
C:\misc> c:\python26\python
>>> class C:
...     def __bool__(self):
...         print('in bool')
...         return False
...
>>> X = C()
>>> bool(X)
True
>>> if X: print(99)
...
99
```

在Python 2.6中，针对布尔值使用`__nonzero__`（或者从设置为假的`__len__`候补方法返回0）：

```
C:\misc> c:\python26\python
>>> class C:
...     def __nonzero__(self):
...         print('in nonzero')
...         return False
...
>>> X = C()
>>> bool(X)
in nonzero
False
>>> if X: print(99)
...
in nonzero
```

但是，别忘了，`__nonzero__`只在Python 2.6中有效；如果在Python 3.0中使用，它将默认地忽略，并且对象将被默认地分类为真——就像是在Python 2.6中使用`__bool__`一样。

对象析构函数：__del__

每当实例产生时，就会调用__init__构造函数。每当实例空间被收回时（在垃圾收集时），它的对立面__del__，也就是析构函数（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。因此会触发其析构函数。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，析构函数不像其他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，对于空间管理来说，是不需要析构函数的^{注1}。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，使析构函数无法执行。

注意： 实际上，即便因为某些很细微的原因，__del__可能会很难使用。例如，直接向sys.stderr（标准错误流）打印一条警告消息，而不是触发一个异常事件，这也会从中引发异常，因为垃圾收集器在不可预料的环境下运行。此外，当我们期待垃圾收集的时候，对象间的循环引用可能会阻止其发生。一个可选的循环检测器，是默认可用的，最终可以自动检测这样的对象，但是，只有在它们没有__del__方法的时候才可用。由于这相对比较容易理解，所以我们再次略过进一步的细节。参见Python的标准手册对__del__和gc垃圾收集模块的介绍，以获取更多信息。

注1： 在当前C实现的Python中，不需要在析构函数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）。

本章小结

还有很多重载的示例，我们已经没有足够的篇幅去介绍了。大多数其他的运算符重载方法，与我们已经介绍过的工作方式类似，并且所有这些都只是拦截内置类型运算的钩子，例如，一些重载方法，拥有独特的参数列表或返回值。我们将在本书随后的部分看到一些其他应用：

- 第33章把 `__enter__` 和 `__exit__` 与语句环境管理器方法一起使用。
- 第37章使用 `__get__` 和 `__set__` 类描述器获取/设置方法。
- 第39章在元类的环境中使用 `__new__` 对象创建方法。

此外，我们在这里已经学过的一些方法，例如 `__call__` 和 `__str__`，随后将会在本书的示例中使用。然而，对于完全介绍，我们将在其他文档资源中查找，参见Python的标准语言手册或参考书，来获取有关额外的重载方法的细节。

下一章中，我们将离开类机制的领域，深入到常用的设计模式——即类经常使用并组合来优化代码复用的方式。然而，在继续学习之前，让我们花点时间来看看本章的练习题，以便回顾我们在这里学习的概念。

本章习题

1. 哪两种运算符重载方法可以用来支持类中的迭代？
2. 哪两种运算符重载方法处理打印，并且在何种环境下处理？
3. 如何在类中拦截分片操作？
4. 如何在类中捕获原处加法？
5. 何时应该提供运算符重载？

习题解答

1. 类可以通过定义（或继承） `__getitem__` 或 `__iter__` 来支持迭代。在所有的迭代环境中，Python首先尝试使用 `__iter__`（它返回支持迭代协议的一个对象，该对象带有一个 `__next__` 方法）：如果在继承搜索中没有找到 `__iter__`，Python退而求其次地用 `__getitem__` 索引方法（它可以重复地调用，使用连续较高的索引）。
2. `__str__` 和 `__repr__` 方法实现对象打印显示。前者由 `print` 和 `str` 内置函数调用；后者由 `print` 和 `str` 调用（如果没有 `__str__` 的话），并且总是由 `repr` 内置函数、

交互式响应和嵌套的出现。也就是说，`__repr__`随处可用，只是当定义了一个`__str__`的时候除外。`__str__`通常用于用户友好的显示，`__repr__`给出额外的细节，或者对象的编码形式。

3. 分片由`__getitem__`索引方法捕获：它用一个分片对象调用；而不是一个简单的索引。在Python 2.6中，`__getslice__`（在Python 3.0中删除了）也可以使用。
4. 原处加法首先尝试`__iadd__`，其次用`__add__`赋值。同样的模式对于所有的二进制运算也是如此。`__radd__`方法对于右端相加也可用。
5. 当一个类自然地匹配的或者需要模拟一个内置类型接口的时候。例如，集合可能模拟序列或映射接口。如果表达式运算符没有自然地映射对象的时候，我们通常不应该实现表达式运算符，而应该使用常规命名的方法。

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。但是，OOP也有设计问题，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，以及一些比目前展示过的例子更实际的额外例子。

在这里，我们将编写Python中常用的OOP设计模式，例如，继承、组合、委托和工厂。我们还将介绍一些类设计的概念，例如伪私有属性、多继承和边界方法。这里提到的很多设计术语，仅仅在本书中给出的这些介绍是不够的；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，所以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，尽管这可能使用代码来实现，就像第38章将要介绍的一样。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。

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，所以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18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16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）。

尽管Python的对象模型很直接，但OOP中的大多数艺术在于我们组合类以实现程序的目标的方式。下一节开始介绍较大的程序使用类来实现其优点的一些方式。

OOP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比萨的机器人的例子。假设我们决定探索另一条路径，开一家比萨餐厅。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，我们决定创建一个机器人制作比萨，但是为了符合逻辑，我们也决定把机器人做成有薪水的功能齐全的员工。

比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。这两个子类都会覆盖继承的work方法来打印更具体的信息。最后，我们的比萨机器人是由更具体的类来模拟：PizzaRobot是一种Chef，也是一种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**的制作比萨机器人，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，而给予**bob**加薪，则会运行**Employee.giveRaise**，因为继承会在这里找到这个方法。

```

C:\python\examples> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>

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，这些类只是模仿真实世界的对象。**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。

OOP和组合：“有一个”关系

在第25章中介绍过组合的概念。从程序员的角度来看，组合涉及把其他对象嵌入容器对象内，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。

组合也反映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），或者使用聚合描述容器和所含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。

既然我们已经有了员工，就把他们放到比萨店，开始忙吧。我们的比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，主厨制作比萨，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()

    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，其构造函数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，服务生是比萨店的组合成分。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，我们的比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)
```

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 False, 'converter must be defined'           # Or raise exception
```

这个类定义了一个转换器方法，它期待子类来填充。这是我们在第28章中介绍的**抽象超类模式**的一个例子（第7部分更多地介绍断言）。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**），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
```

```
obj = Uppercase(open('spam.txt'), sys.stdout)
obj.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:\lp4e> type spam.txt
spam
Spam
SPAM!

C:\lp4e> 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:\lp4e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp4e> 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:\lp4e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print('<PRE>%s</PRE>' % line.rstrip())
...
>>> 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并能继承，我们所需做的代码编写就是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31章结尾的练习题9以及其附录B的解法。这个例子类似于比萨店的例子。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请参考其他资源。

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。实际上，这些工具往往足够吸引人，可以促进类的通用用法——通过pickle或shelve一个类实例，我们得到了包含数据和逻辑的组合的数据存储。

例如，除了可以模拟真实世界的交互外，在这里开发的比萨店类，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。在第27章中的OOP教程中，我们使用shelve来存储类的实例，而对象的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：把这种实例对象pickle至文件，使其在Python程序执行时都能够永续保存。

```

>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.dat', 'wb'))

```

这一次性地把整个复合的shop对象保存到一个文件中。为了在另一个会话或程序中再次找回它，只需要一个步骤就够了。实际上，以这种方式存储的对象保存了状态和行为：

```

>>> import pickle
>>> obj = pickle.load(open('shopfile.dat', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> obj.order('Sue')
Sue orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Sue pays for item to <Employee: name=Pat, salary=40000>

```

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：“包装”对象

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（有时称为代理类）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。

```

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29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29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消息，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来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31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31章和第38章有关**函数装饰器**的讨论：这是关联性很强的概念，旨在用来增加特定函数或方法调用，而不是对对象的整个接口；还有**类装饰器**，它充当向一个类的所有实例自动添加诸如基于委托的包装器的一种方式。

注意： 版本差异提示：在Python 2.6中，运算符重载方法通过把内置操作导向`__getattr__`这样的通用属性拦截方法来运行。例如，直接打印一个包装对象，针对`__repr__`或`__str__`调用该方法，随后把调用传递给包装对象。在Python 3.0中，这种情况不再会发生：打印不会触发`__getattr__`，并且使用一个默认的显示。在Python 3.0中，新式类在类中查找运算符重载方法，并且完全忽略常规的实例查找。我们将在第37章再次回顾这一话题。现在，记住如果希望运算符重载方法在Python 3.0中被拦截，我们需要在包装类中重新定义它们（通过手动、通过工具或者通过超类）。

类的伪私有属性

除了较大的结构性目标，类设计往往也必须解决名称用法。在第五部分中，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。实际上，即便当它们可能应该使用的时候，也并非总是使用它们——更通俗地说，Python程序员用一个单个的下划线来编写内部名称（例如，`_x`），这只是一个非正式的惯例，让你知道这是一个不应该修改的名字（它对Python自身来说没有什么意义）。

由于你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的变量名，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，

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。作为脚本语言，Python更关心的是开放而不是约束。此外，回想一下第29章讨论的运算符重载，`__getattr__`和`__setattr__`可用于模拟私有性，但是，实际中通常也很少使用。关于编写一个更加实际的私有性装饰器的更多讨论，参见本书第38章。

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。

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                # I 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 I 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`_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。

```

% 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。

伪私有属性在较大的框架或工具中也是有用的，既可以避免引入可能在类树中某处偶然隐藏定义的新的方法名，也可以减少内部方法被在树的较低处定义的名称替代的机会。如果一个方法倾向于只在一个可能混合到其他类的类中使用，在前面使用双下划线，以确保该方法不会受到树中的其他名称的干扰，特别是在多继承的环境中：

```

class Super:
    def method(self): ...                # A real application method

class Tool:
    def __method(self): ...              # Becomes _Tool__method
    def other(self): self.__method()    # Use my internal method

class Sub1(Tool, Super): ...
    def actions(self): self.method()    # Runs Super.method as expected

class Sub2(Tool):
    def __init__(self): self.method = 99 # Doesn't break Tool.__method

```

我们在第25章简单地遇到过多继承，并且将会在本章稍后更详细地介绍它。别忘了，在类头部行中，超类按照它们从左到右的顺序搜索。在这里，这就意味着`Sub1`首选`Tool`属性，而不是`Super`中的那些属性。尽管在这个例子中，我们可能通过切换`Sub1`类头部列出的超类的顺序，来迫使Python首先选择应用程序类的方法，伪私有属性一起解决了这一问题。伪私有名称还阻止了子类偶然地重新定义内部的方法名称，就像在`Sub2`中那样。

同样，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。

要了解使用__X命名功能的更多示例，请参见本章随后的*lister.py*混合类，参见关于多继承的小节，以及第38章对于Private类装饰器的介绍。如果你更广泛地关心私有性，可能需要看一看第29章所提到的模拟私有实例属性的内容，在第29章“属性引用：__getattr__和__setattr__”一节，并且看看第38章中的Private类装饰器，该章将给予这一特殊方法。虽然有可能在Python类中模拟真正的读取控制，但实际中很少这么做，即使是大型的系统也是如此。

方法是对象：绑定或无绑定

方法（特别是绑定方法），通常简化了Python中的很多设计目标的实现。我们在第29章中学习__call__的时候简单地介绍了绑定方法。详细的介绍在这里给出，并且比你想象的还要通用和灵活。

在第19章中，我们学习了函数如何可以像常规对象一样处理。方法也是一种对象，并且可以用与其他对象大部分相同的方式来广泛地使用——可以对它们赋值、将其传递给函数、存储在数据结构中，等等。由于类方法可以从一个实例或一个类访问，它们实际上在Python中有两种形式。

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（unbound）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。在Python 3.0中，一个无绑定方法和一个简单的函数是相同的，可以通过类名来调用；在Python 2.6中，它是一种独特的类型，并且不提供一个实例就无法调用。

绑定实例方法对象：self+函数对

通过对实例进行点号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，就像字符串和数字。执行时，两者都需要它们在第一参数中的实例（也就是self的值）。这也就是为什么我们上一章在子类方法调用超类方法时，要刻意传入实例。从严格意义上来说，这类调用会产生无绑定的方法对象。

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（参考后面的方框“为什么要在意：绑定方法和回调函数”部分中的例子）。

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，在单步中调用了它的方法，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。

```
object1 = Spam()
t = Spam.doit             # Unbound method object (a function in 3.0: see ahead)
t(object1, 'howdy')       # Pass in instance (if the method expects one in 3.0)
```

扩展一下，如果我们引用的`self`的属性是引用类中的函数，那么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，就得小心了，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2}。

在Python 3.0中，无绑定方法是函数

在Python 3.0中，该语言已经删除了无绑定方法的概念。我们在这里所介绍的无绑定方法，在Python 3.0中当作一个简单函数对待。对于大多数用途来说，这对你的代码没什么影响；任何一种方式，当通过一个实例来调用一个方法的时候，都会有一个实例传递给该方法的第一个参数。

显式类型测试程序可能受到影响，如果你打印出一个非实例的类方法，它在Python 2.6中显示“无绑定方法”（unbound method），在Python 3.0中显示“函数”（function）。

此外，在Python 3.0中，不使用一个实例而调用一个方法没有问题，只要这个方法不期待一个实例，并且你通过类调用它而不是通过一个实例调用它。也就是说，只有对通过实例调用，Python 3.0才会向方法传递一个实例。当通过一个类调用的时候，只有在方法期待一个实例的时候，才必须手动传递一个实例：

```
C:\misc> c:\python30\python
>>> class Selfless:
...     def __init__(self, data):
...         self.data = data
...     def selfless(arg1, arg2):                # A simple function in 3.0
...         return arg1 + arg2
...     def normal(self, arg1, arg2):           # Instance expected when called
...         return self.data + arg1 + arg2
...
>>> X = Selfless(2)
>>> X.normal(3, 4)                             # Instance passed to self automatically
9
>>> Selfless.normal(X, 3, 4)                   # self expected by method: pass manually
9
>>> Selfless.selfless(3, 4)                   # No instance: works in 3.0, fails in 2.6!
7
```

这里的最后一个测试在Python 2.6中失效，因为无绑定方法默认地需要传递一个实例；它在Python 3.0中有效，因为这样的方法当作一个简单函数对待，而不需要一个实例。尽管这会删除Python 3.0中某些潜在的错误陷阱（如果一个程序员偶然忘记传入一个实

注2： 参考第31章有关静态和类方法的讨论，以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。Python支持3种类方法——实例方法、静态方法和类方法，并且Python 3.0也允许类中的简单函数。

例，会怎么样呢？），但它允许类方法用作简单的函数，只要它们没有被传递并且不期望一个“self”实例参数。

如下的两个调用仍然在Python 3.0和Python 2.6中都失效，第一个（通过实例调用）自动把一个实例传递给一个并不期待实例的方法，而第二个（通过类调用）不会把一个实例传递给确实期待一个实例的方法：

```
>>> X.selfless(3, 4)
TypeError: selfless() takes exactly 2 positional arguments (3 given)

>>> Selfless.normal(3, 4)
TypeError: normal() takes exactly 3 positional arguments (2 given)
```

由于这一修改，对于只通过类名而不通过一个实例调用的、没有一个self参数的方法，在Python 3.0中不再需要下一章介绍的staticmethod装饰器——这样的方法作为简单函数运行，不会接受一个实例参数。在Python 2.6中，这样的调用是错误的，除非手动地传递一个实例（下一章更详细地介绍静态方法）。

注意到在Python 3.0中的这一差别是很重要的，但是，从实用的角度来看，绑定方法通常更重要。因为，它们在一个单个的对象中把实例和函数配对起来，所以它们通常可以当做可调用对象对待。下一小节介绍了这样在代码中意味着什么。

注意： 对于Python 3.0和Python 2.6中的无绑定方法处理的可视化说明，请参见本章稍后的多继承小节的`lister.py`示例。它的类打印出了从实例和类获取的方法的值，在Python的两个版本中分别这么做。

绑定方法和其他可调用对象

正如前面所提到的，绑定方法可以作为一个通用对象处理，就像是简单函数一样——它们可以任意地在一个程序中传递。此外，由于绑定方法在单个的包中组合了函数和实例，因此它们可以像任何其他可调用对象一样对待，并且在调用的时候不需要特殊的语法。例如，如下的例子在一个列表中存储了4个绑定方法对象，并且随后使用常规的调用表达式来调用它们：

```
>>> class Number:
...     def __init__(self, base):
...         self.base = base
...     def double(self):
...         return self.base * 2
...     def triple(self):
...         return self.base * 3
...
>>> x = Number(2)                                     # Class instance objects
```

```

>>> y = Number(3)                                # State + methods
>>> z = Number(4)
>>> x.double()                                     # Normal immediate calls
4

>>> acts = [x.double, y.double, y.triple, z.double] # List of bound methods
>>> for act in acts:                               # Calls are deferred
...     print(act())                               # Call as though functions
...
4
6
9
8

```

和简单函数一样，绑定方法对象拥有自己的内省信息，包括让它们配对的实例对象和方法函数访问的属性。调用绑定方法会直接分配配对：

```

>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x0278F610>, <function double at 0x027A4ED0>)
>>> bound.__self__.base
2
>>> bound()                                         # Calls bound.__func__(bound.__self__, ...)
4

```

实际上，绑定方法只是Python中众多的可调用对象类型中的一种。正如下面所示，简单函数编写为一个def或lambda，实例继承了一个__call__，并且绑定实例方法都能够以相同的方式对待和调用：

```

>>> def square(arg):
...     return arg ** 2                             # Simple functions (def or lambda)
...
>>> class Sum:
...     def __init__(self, val):                     # Callable instances
...         self.val = val
...     def __call__(self, arg):
...         return self.val + arg
...
>>> class Product:
...     def __init__(self, val):                     # Bound methods
...         self.val = val
...     def method(self, arg):
...         return self.val * arg
...
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method]     # Function, instance, method

>>> for act in actions:                             # All 3 called same way
...     print(act(5))                               # Call any 1-arg callable
...
25
7

```

```

15
>>> actions[-1](5)                                # Index, comprehensions, maps
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]

```

从技术上讲，类也属于可调用对象的范畴，但是，我们通常调用它们来产生实例而不是做实际的工作，如下所示：

```

>>> class Negate:
...     def __init__(self, val):                # Classes are callables too
...         self.val = -val                    # But called for object, not work
...     def __repr__(self):                    # Instance print format
...         return str(self.val)
...
>>> actions = [square, subject, pobject.method, Negate] # Call a class too
>>> for act in actions:
...     print(act(5))
...
25
7
15
-5
>>> [act(5) for act in actions]                # Runs __repr__ not __str__!
[25, 7, 15, -5]

>>> table = {act(5): act for act in actions}    # 2.6/3.0 dict comprehension
>>> for (key, value) in table.items():
...     print('{0:2} => {1}'.format(key, value)) # 2.6/3.0 str.format
...
-5 => <class '__main__.Negate'>
25 => <function square at 0x025D4978>
15 => <bound method Product.method of <__main__.Product object at 0x025D0F90>>
7 => <__main__.Sum object at 0x025D0F70>

```

正如你所看到的，绑定方法和Python的可调用对象模型，通常都是Python的设计朝向一种难以置信的灵活语言方向努力的众多方式中的一些。

你现在应该理解方法对象模型了。要了解绑定方法使用的其他示例，参见后续的方框“为什么要在意：绑定方法和回调函数”部分，以及前面一章中关于__call__方法的一节中对回调处理器的讨论。

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，因此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（在Python 2.6中叫做Tkinter）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，通常是将一个不带参数的可调用对象传递给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（一个绑定方法对象），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29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。

多重继承：“混合”类

很多基于类的设计都要求组合方法的全异的集合。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。从技术上来讲，因为任何超类本身可能还有一些其他的超类，对于更大的类树，这个搜索可以更复杂一点。

- 在传统类中（默认类，直到Python 3.0），属性搜索处理对所有路径深度优先，直到继承树的顶端，然后从左到右进行。
- 在新式类（以及Python 3.0的所有类中），属性搜索处理沿着树层级、以更加广度优先的方式进行（参见下一章中关于新式类的介绍）。

不管哪种方式，当一个类拥有多个超类的时候，它们会根据`class`语句头部行中列出的顺序从左到右查找。

通常意义上讲，多重继承是建模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。使用多重继承，对象获得了所有其超类中行为的组合。

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。从某种意义上讲，混合类类似于模块：它们提供方法的包，以便在其客户子类中使用。然而，和模块中的简单函数不同，混合类中的方法也能够访问`self`实例，以使用状态信息和其他方法。下一小节展示了这些工具的一种常见使用方法。

编写混合显示类

正如我们所见到的，Python的默认方式来打印一个类实例对象，并不是难以置信的有用：

```
>>> class Spam:
...     def __init__(self):
...         self.data1 = "food"
...
>>> X = Spam()
>>> print(X)
<__main__.Spam object at 0x00864818>
```

No __repr__ or __str__
Default: class, address
Displays "instance" in Python 2.6

就像我们在第29章学习运算符重载的时候所见到的，你可以提供一个`__str__`或`__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？

这就是混合类的用处。在混合类中定义一个显示方法一次，使得我们能够在想要看到一个定制显示格式的任何地方重用它。我们已经看到了做相关工作的工具：

- 第27章的`AttrDisplay`类在一个通用的`__str__`方法中格式化了实例属性，但是，它没有爬升类树，并且只是用于单继承模式中。
- 第28章的`classtree.py`定义了函数以爬升和遍历类树，但是，它没有显示对象属性，并且没有架构为一个可继承类。

这里，我们将继续回顾这些示例的技术，并且在它们的基础上扩展编码一组3个混合类，这3个类充当通用的显示工具，以列出一个类树上所有对象的实例属性、继承属性

和属性。我们还将在多继承模式中使用我们的工具，并利用编码技术使得类更好地适合于用作通用工具。

用__dict__列出实例属性

让我们从一个简单的例子开始——列出附加给一个实例的属性。如下的类编写在文件 *lister.py* 中，它定义了一个名为 `ListInstance` 的混合类，它对于将其包含到头部行的所有类都重载了 `__str__` 方法。由于 `ListInstance` 编写为一个类，所以它成为一个通用工具，其格式化逻辑可以用于任何子类的实例：

```
# File lister.py

class ListInstance:
    """
    Mix-in class that provides a formatted print() or str() of
    instances via inheritance of __str__, coded here; displays
    instance attrs only; self is the instance of lowest class;
    uses __X names to avoid clashing with client's attrs
    """
    def __str__(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
    def __attrnames(self):
        result = ''
        for attr in sorted(self.__dict__):      # Instance attr dict
            result += '\tname %s=%s\n' % (attr, self.__dict__[attr])
        return result
```

`ListInstance` 使用前面介绍的一些技巧来提取实例的类名和属性：

- 每个实例都有一个内置的 `__class__` 属性，它引用自己所创建自的类；并且每个类都有一个 `__name__` 属性，它引用了头部中的名称，因此，表达式 `self.__class__.__name__` 获取了一个实例的类的名称。
- 这个类通过直接扫描实例的属性字典（别忘了，它从 `__dict__` 中导出），以构建一个字符串来显示所有实例属性的名称和值，从而完成其主要工作。字典的键通过排序，以避免Python跨版本的任何排序差异。

在这些方面，`ListInstance` 类似于第27章的属性显示；实际上，它很大程度上只是一个主题的变体。这里，我们的类显示了两种其他技术：

- 它通过调用 `id` 内置函数显示了实例的内存地址，该函数返回任何对象的地址（根据定义，这是一个唯一的对象标识符，在随后对这一代码的修改中 useful）。
- 它针对其工作方法使用伪私有命名模式：`__attrnames`。正如我们在本章前面所了

解到的，Python通过扩展属性名称以包含类名，从而把这样的名称本地化到其包含类中（在这个例子中，它变成了`_ListInstance__attrnames`）。对于附加到`self`的类属性（如方法）和实例属性，都是如此。这种行为在这样的通用工具中很有用，因为它确保了其名称不会与其客户子类中使用的任何名称冲突。

由于`ListInstance`定义了一个`__str__`运算符重载方法，所以派生自这个类的实例在打印的时候自动显示其属性，只给定了比简单地址多一点的信息。如下是使用中的类，在单继承模式中（这段代码在Python 3.0和Python 2.6下都同样工作）：

```
>>> from lister import ListInstance
>>> class Spam(ListInstance):                                # Inherit a __str__ method
...     def __init__(self):
...         self.data1 = 'food'
...
>>> x = Spam()
>>> print(x)                                                # print() and str() run __str__
<Instance of Spam, address 40240880:
    name data1=food
>
```

我们可以把列表输出获取为一个字符串，而不用`str`打印出它，并且交互响应仍然使用默认格式：

```
>>> str(x)
'<Instance of Spam, address 40240880:\n\tname data1=food\n>'
>>> x                                                        # The __repr__ still is a default
<__main__.Spam object at 0x026606F0>
```

`ListInstance`对于我们所编写的任何类都很有用，即便类已经有一个或多个超类。这就是多继承的用武之地，通过把`ListInstance`添加到一个类头部的超类列表中（例如，混合进去），我们可以在仍然继承自己有超类的同时“自由地”获得`__str__`。文件`testmixin.py`展示如下：

```
# File testmixin.py

from lister import *                                       # Get lister tool classes

class Super:
    def __init__(self):                                    # Superclass __init__
        self.data1 = 'spam'                                # Create instance attrs
    def ham(self):
        pass

class Sub(Super, ListInstance):                             # Mix in ham and a __str__
    def __init__(self):                                    # listers have access to self
        Super.__init__(self)
        self.data2 = 'eggs'                                # More instance attrs
        self.data3 = 42
    def spam(self):                                         # Define another method here
```

```

        pass

if __name__ == '__main__':
    X = Sub()
    print(X)                                # Run mixed-in __str__

```

这里，Sub从Super和ListInstance继承了名称，它是自己的名称与其超类中名称的组合。当我们生成一个Sub实例并打印它，就会自动获得从ListInstance混合进去的定制表示（在这个例子中，这段脚本的输出在Python 3.0和Python 2.6下都是相同的，除了对象地址不同）：

```

C:\misc> C:\python30\python testmixin.py
<Instance of Sub, address 40962576:
    name data1=spam
    name data2=eggs
    name data3=42
>

```

ListInstance在它混入的任何类中都有效，因为self引用拉入了这个类的子类的一个实例，而不管它可能是什么。从某种意义上讲，混合类是模块的类等价形式——它是在各种客户中有用的方法包。例如，下面是再次在单继承模式中工作的Lister，它作用于一个不同的类实例之上，使用import，并且带有类之外的属性设置：

```

>>> import lister
>>> class C(lister.ListInstance): pass
...
>>> x = C()
>>> x.a = 1; x.b = 2; x.c = 3
>>> print(x)
<Instance of C, address 40961776:
    name a=1
    name b=2
    name c=3
>

```

它们除了提供这一工具，还像所有的类一样，混入了优化代码维护。例如，如果你稍后决定扩展ListInstance的__str__也打印出一个实例继承的所有类属性，你是安全的；因为它是一个集成的方法，修改__str__自动地更新导入该类和混合该类的每个子类的显示。这会儿真的很晚了，我们将在下一小节看看这样的扩展。

使用dir列出继承的属性

我们的Lister混合类只显示实例属性（例如，附加到实例对象自身的名称）。扩展该类以显示从一个实例可以访问的所有属性，这也是很容易的——这包括它自己以及它所继承自的类。技巧是使用dir内置函数，而不是扫描实例的__dict__字典，后者只是保存了实例属性，但是，在Python 2.2及以后的版本中，前者也收集了所有继承的属性。

如下修改后的代码实现了这一方案，我们已经将其重新命名，以便使得测试更简单，但是，如果用这个替代最初的版本，所有已有的客户类将自动选择新的显示：

```
# File lister.py, continued
```

```
class ListInherited:
    """
    Use dir() to collect both instance attrs and names
    inherited from its classes; Python 3.0 shows more
    names than 2.6 because of the implied object superclass
    in the new-style class model; getattr() fetches inherited
    names not in self.__dict__; use __str__, not __repr__,
    or else this loops when printing bound methods!
    """
    def __str__(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
    def __attrnames(self):
        result = ''
        for attr in dir(self):                # Instance dir()
            if attr[:2] == '__' and attr[-2:] == '__':    # Skip internals
                result += '\tname %s=<>\n' % attr
            else:
                result += '\tname %s=%s\n' % (attr, getattr(self, attr))
        return result
```

注意，这段代码省略了__x__名称的值；这些大部分都是内部名称，我们通常不会在这样的通用列表中注意到。这个版本必须使用getattr内置函数来获取属性，通过指定字符串而不是使用实例属性字典索引——getattr使用了继承搜索协议，并且我们在这里列出的一些代码没有存储到实例自身中。

要测试新的版本，修改testmixin.py文件并使用新的类来替代：

```
class Sub(Super, ListInherited):                # Mix in a __str__
```

这个文件的输出随着每个版本而变化。在Python 2.6中，我们得到如下输出。注意，名称压缩在lister的方法名中起作用（我缩减了其全部的值显示，以节省篇幅）：

```
C:\misc> c:\python26\python testmixin.py
<Instance of Sub, address 40073136:
  name _ListInherited__attrnames=<bound method Sub.__attrnames of <...more...>>
  name __doc__=<>
  name __init__=<>
  name __module__=<>
  name __str__=<>
  name data1=spam
  name data2=eggs
  name data3=42
  name ham=<bound method Sub.ham of <__main__.Sub instance at 0x026377B0>>
```

```

    name spam=<bound method Sub.spam of <__main__.Sub instance at 0x026377B0>>
>

```

在Python 3.0中，更多的属性显示出来，因为所有的类都是“新式的”，并且从隐式的object超类那里继承了名称（关于object的更多内容在第31章介绍）。由于如此多的名称继承自默认的超类，我们已经在这里省略了很多。自行运行程序以得到完整的列表：

```

C:\misc> c:\python30\python testmixin.py
<Instance of Sub, address 40831792:
    name _ListInherited__attrnames=<bound method Sub.__attrnames of <...more...>>
    name __class__=<>
    name __delattr__=<>
    name __dict__=<>
    name __doc__=<>
    name __eq__=<>
    ...more names omitted...
    name __repr__=<>
    name __setattr__=<>
    name __sizeof__=<>
    name __str__=<>
    name __subclasshook__=<>
    name __weakref__=<>
    name data1=spam
    name data2=eggs
    name data3=42
    name ham=<bound method Sub.ham of <__main__.Sub object at 0x026F0B30>>
    name spam=<bound method Sub.spam of <__main__.Sub object at 0x026F0B30>>
>

```

这里注意一点，既然我们也显示继承的方法，我们必须使用__str__而不是__repr__来重载打印。使用__repr__，这段代码将会循环，显示一个方法的值，该值触发了该方法的类的__repr__，从而显示该类。也就是说，如果lister的__repr__试图显示一个方法，显示该方法的类将再次触发lister的__repr__。很微妙，但确实如此！在这里，自己把__str__修改为__repr__来看看。如果你在这样的环境中使用__repr__，可以使用isinstance来比较属性值的类型和标准库中的types.MethodType，以知道省略哪些项，从而避免循环。

列出类树中每个对象的属性

让我们来看最后一个扩展。我们的lister没有告诉我们一个继承名称来自哪个类。然而，正如我们在第28章末尾的classtree.py示例中看到的，在代码中爬升类继承树很容易。如下的混合类使用这一名称技术来显示根据属性所在的类来分组的属性，它遍历了整个类树，在此过程中显示了附加到每个对象上的属性。它这样遍历继承树：从一个实例的__class__到其类，然后递归地从类的__bases__到其所有超类，一路扫描对象的__dicts__：

File lister.py, continued

```
class ListTree:
    """
    Mix-in that returns an __str__ trace of the entire class
    tree and all its objects' attrs at and above self;
    run by print(), str() returns constructed string;
    uses __X attr names to avoid impacting clients;
    uses generator expr to recurse to superclasses;
    uses str.format() to make substitutions clearer
    """
    def __str__(self):
        self.__visited = {}
        return '<Instance of {0}, address {1}:>\n{2}{3}'.format(
            self.__class__.__name__,
            id(self),
            self.__attrnames(self, 0),
            self.__listclass(self.__class__, 4))

    def __listclass(self, aClass, indent):
        dots = '.' * indent
        if aClass in self.__visited:
            return '\n{0}<Class {1}:, address {2}: (see above)>\n'.format(
                dots,
                aClass.__name__,
                id(aClass))
        else:
            self.__visited[aClass] = True
            genabove = (self.__listclass(c, indent+4) for c in aClass.__bases__)
            return '\n{0}<Class {1}, address {2}:>\n{3}{4}{5}>\n'.format(
                dots,
                aClass.__name__,
                id(aClass),
                self.__attrnames(aClass, indent),
                ''.join(genabove),
                dots)

    def __attrnames(self, obj, indent):
        spaces = ' ' * (indent + 4)
        result = ''
        for attr in sorted(obj.__dict__):
            if attr.startswith('__') and attr.endswith('__'):
                result += spaces + '{0}=<>\n'.format(attr)
            else:
                result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
        return result
```

注意，这里使用一个**生成器表达式**来导向对超类的递归调用，它由嵌套的字符串join方法激活。还要注意，这个版本使用Python 3.0和Python 2.6的字符串格式化方法而不是%来格式化表达式，以使得替代更清晰。当像这样应用很多替代的时候，明确的参数数目可能使得代码更容易理解。简而言之，在这个版本中，我们把如下的第一行与第二行交换：

```

return '<Instance of %s, address %s:\n%s%s>' % (...)           # Expression
return '<Instance of {0}, address {1}:\n{2}{3}>'.format(...)    # Method

```

现在，修改`testmixin.py`，再次测试新类继承：

```

class Sub(Super, ListTree):                                   # Mix in a __str__

```

在Python 2.6中，该文件的树遍历输出如下所示：

```

C:\misc> c:\python26\python testmixin.py
<Instance of Sub, address 40728496:
  _ListTree__visited={}
  data1=spam
  data2=eggs
  data3=42

....<Class Sub, address 40701168:
  __doc__=<>
  __init__=<>
  __module__=<>
  spam=<unbound method Sub.spam>

.....<Class Super, address 40701120:
  __doc__=<>
  __init__=<>
  __module__=<>
  ham=<unbound method Super.ham>
.....>

.....<Class ListTree, address 40700688:
  _ListTree__attrnames=<unbound method ListTree.__attrnames>
  _ListTree__listclass=<unbound method ListTree.__listclass>
  __doc__=<>
  __module__=<>
  __str__=<>
.....>
....>
>

```

注意，在这一输出中，方法现在在Python 2.6下是**无绑定的**，因为我们直接从类获取它们，而不是从实例。还注意`lister`的`__visited`表把自己的名称压缩到实例的属性字典中；除非我们很不走运，这不会与那里的其他数据冲突。

在Python 3.0下，我们再次获得了额外的属性和超类。注意，无绑定的方法在Python 3.0中是简单**函数**，正如本章前面的提示中所介绍的（在这里，我们再次删除了对象中的大多数内置对象以节省篇幅，请自行运行这段代码以获取完整的列表）：

```

C:\misc> c:\python30\python testmixin.py
<Instance of Sub, address 40635216:
  _ListTree__visited={}
  data1=spam
  data2=eggs

```

```

data3=42

....<Class Sub, address 40914752:
    __doc__=<>
    __init__=<>
    __module__=<>
    spam=<function spam at 0x026D53D8>

.....<Class Super, address 40829952:
    __dict__=<>
    __doc__=<>
    __init__=<>
    __module__=<>
    __weakref__=<>
    ham=<function ham at 0x026D5228>

.....<Class object, address 505114624:
    __class__=<>
    __delattr__=<>
    __doc__=<>
    __eq__=<>
    ...more omitted...
    __repr__=<>
    __setattr__=<>
    __sizeof__=<>
    __str__=<>
    __subclasshook__=<>
.....>
.....>

.....<Class ListTree, address 40829496:
    __ListTree__attrnames=<function __attrnames at 0x026D5660>
    __ListTree__listclass=<function __listclass at 0x026D56A8>
    __dict__=<>
    __doc__=<>
    __module__=<>
    __str__=<>
    __weakref__=<>
.....<Class object:, address 505114624: (see above)>
.....>
....>
>

```

这个版本通过保留一个目前已经访问过的类的表来避免两次列出同样的类对象（这就是为什么一个对象的id包含其中，以充当一个之前显示项的键）。和第24章的过渡性模块重载程序一样，字典在这里用来避免重复和循环，因为类对象可能是字典键。集合也可以提供类似的功能。

这个版本还会再次通过省略__X__名称来避免较大的内部对象。如果你注释掉这些名称的测试，它们的值将会正常显示。这里是在Python 2.6下输出的摘要，带有这一临时性的修改（整个输出很大，并且在Python 3.0中这种情况甚至变得更糟，因此，这些名字可能会有所省略）：


```

C:\misc> c:\python26\python testmixin.py
...more omitted...

.....<Class ListTree, address 40700688:
    _ListTree__attrnames=<unbound method ListTree.__attrnames>
    _ListTree__listclass=<unbound method ListTree.__listclass>
    __doc__=
    Mix-in that returns the __str__ trace of the entire class
    tree and all its objects' attrs at and above self;
    run by print, str returns constructed string;
    uses __X attr names to avoid impacting clients;
    uses generator expr to recurse to superclasses;
    uses str.format() to make substitutions clearer

    __module__=lister
    __str__=<unbound method ListTree.__str__>
.....>

```

为了更有趣，尝试把这个类混合到更实质的某些内容中，例如Python的tkinter GUI工具箱模块的Button类。通常，我们想要在一个类的头部命名ListTree（最左端），因此，它的__str__会被选取；Button也有一个，并且在多继承中最左端的超类首先搜索。如下的输出相当庞大（18K个字符），因此，自己运行这段代码看看完整的列表（并且，如果你在使用Python 2.6，记住应该对模块名使用Tkinter而不是tkinter）：

```

>>> from lister import ListTree
>>> from tkinter import Button                # Both classes have a __str__
>>> class MyButton(ListTree, Button): pass      # ListTree first: use its __str__
...
>>> B = MyButton(text='spam')
>>> open('savetree.txt', 'w').write(str(B))    # Save to a file for later viewing
18247
>>> print(B)                                  # Print the display here
<Instance of MyButton, address 44355632:
    _ListTree__visited={}
    _name=44355632
    _tclCommands=[]
    ...much more omitted...
>

```

当然，在这里还有更多的事情可以做（遍历一个GUI中的树可能自然而然的是下一步），但是，我们将把进一步的工作保留为一个推荐练习。我们将在本书这一部分末尾的练习中扩展这些代码，以在实例和类显示的开始处的括号中列出超类名称。

这里的主要核心是，OOP是完全关于代码复用的，并且混合类也是一个强大的示例。和几乎编程中所有的内容一样，多继承在应用得当的时候也是一种有用的工具。实际上，它是一种高级功能，如果不注意或滥用的话，可能变得很复杂。我们将在下一章的末尾重新回顾这一主题成为陷阱的情况。在那一章中，我们还将遇到新式类模式，它针对一种特殊的多继承情况修改搜索代码。

注意：支持slot：由于它们扫描示例词典，所以这里介绍的ListInstance和ListTree类不能直接支持存储在slot中的属性——slot是一种新的、相对很少使用的选项，我们将在下一章中介绍，在那里，示例属性将在一个__slots__类属性中声明。例如，如果在textmixin.py中，我们在Super中赋值__slots__=['data1']，在Sub中赋值__slots__=['data3']，只有data2属性通过这两个lister类显示在该实例中；ListTree也会显示data1和data3，但是是作为Super和Sub类对象的属性，并且是它们的值的一种特殊格式（从技术上讲，它们都是类级别的描述符）。

要更好地支持这些类中的slot属性，把__dict__扫描循环修改为使用下一章给出的代码来迭代__slots__列表，并且使用getattr内置函数来获取值，而不是使用__dict__索引（ListTree已经这么做了）。既然实例只继承最低的类的__slots__，当__slots__列表出现在多个超类中的时候，你可能也需要提出一种政策（ListTree已经将它们显示为类属性）。ListInherited对所有这些都是免疫的，因为dir结果组合了__dict__名称和所有类的__slots__名称。

此外，作为一项政策，我们可以直接允许代码处理基于slot的属性（就像它当前所做的那样），而不是将其复杂化为一种少用的、高级的特性。slot和常规的实例属性是不同的名称。我还将在下一章进一步介绍slot，我在这些示例中省略了对它们的介绍，以避免进一步的依赖性——这并非一项有效的设计目标，但是对本书来说是合理的。

类是对象：通用对象的工厂

有时候，基于类的设计要求要创建的对象来响应条件，而这些条件是在编写程序的时候无法预料的。工厂设计模式允许这样的一种延迟方法。在很大程度上由于Python的灵活性，工厂可以采取多种形式，其中的一些根本不会显得特殊。

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 aClass(*args)              # Call aClass (or apply in 2.6 only)

class Spam:
    def doit(self, message):
        print(message)
```

注3： 其实，这种语法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，正如我们在第18章所介绍的，Python 2.6有一种aClass(*args)的替代方法：apply(aClass, args)内置调用，它在Python 3.0中已经删除了，因为有冗余性和局限性。

```

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），还有该类构造函数的一个或多个参数。这个函数使用特殊的“`varargs`”调用语法来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函数参数。

可能的改进之处就是，在构造函数调用中支持关键词参数。工厂函数能够通过`**args`参数收集参数，并在类调用中传递它们：

```

def factory(aClass, *args, **kwargs):    # +kwargs dict
    return 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。

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25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，以及选用构造函数的调

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函数参数，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。

与设计相关的其他话题

在本章中，我们已经介绍了继承、复合、委托、多继承、绑定方法和工厂，这些是在Python程序中组合类的所有常用模式。在设计模式领域，我们其实真的只是隔靴挠痒。在本书中的其他地方，你还会看到对与设计相关的其他话题的介绍，例如：

- 抽象超类（第28章）
- 装饰器（第31章和第38章）
- 类型子类（第31章）
- 静态方法和类方法（第31章）
- 管理属性（第37章）
- 元类（第31章和第39章）

对于设计模式的更多细节，请参考关于OOP的其他资源。尽管模式在OOP中很重要，并且在Python中往往比在其他语言中应用更自然，但它们并不是特定于Python本身的。

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（例如伪私有属性），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。不过，下面首先做一下本章的习题。

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？
5. 为什么使用伪私有属性？

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。
5. 伪私有属性（其名称以两个下划线开始：`__x`）用来把名称本地化到包含类中。这包括像定义在类中的方法这样的类属性，以及在类中赋值的`self`实例属性。这样的名称扩展来包含类名称，类名称使得它们独特。

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、新式类的变化和扩展、静态方法和类方法、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。

注意：本章集中了高级类主题，但是有些主题甚至太高级了，以至于本章无法很好地介绍。像特性、描述符、装饰器和元类这样的主题，在这里只是简单提及，并且将在本书最后一部分中更完整地介绍。对于这些主题的一些更完整的例子和扩展，请参阅后面的内容。

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，就像第

30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16章和第18章所写的那些集合函数吗？下面是它们以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。和我们前面的函数不同，这里使用类允许我们创建多个自包含的集合对象，带有预先设置的数据和行为，而不是手动把列表传入函数中：

```
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:' + repr(self.data)  # print()
```

要使用这个类，我们生成实例、调用方法，并且像往常一项运行定义的运算符：

```
x = Set([1, 3, 5, 7])
print(x.union(Set([1, 4, 7])))             # prints Set:[1, 3, 5, 7, 4]
print(x | Set([1, 4, 6]))                  # prints Set:[1, 3, 5, 7, 4, 6]
```

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函数。

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说明了如何做。

```
# Subclass built-in list type/class
# Map 1..N to 0..N-1; call back to built-in version.

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')
    print(x)

    print(x[1])
    print(x[3])

    x.append('spam'); print(x)
    x.reverse();      print(x)
```

__init__ inherited from list
__repr__ inherited from list
MyList.__getitem__
Customizes list superclass method
Attributes from list superclass

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映射为实际的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，可以说是很强大的工具。

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，就像本节前面所示。正如我们在第5章所学习过的，Python如今带有一个强大的内置集合对象，还有常量和解析语法可以生成新的集合。然而，自己编写一个集合，通常仍然是学习类型子类建立过程的一种好办法。

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

    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（散列），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。

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。在Python 3.0中，类的区分已经融合了，但是对于Python 2.X的用户来说，还是有所区分的。

- 对于Python 3.0来说，所有的类都是我们所谓的“新式类”，不管它们是否显式地继承自object。所有的类都继承自object，不管是显式的还是隐式的，并且，所有的对象都是object的实例。
- 在Python 2.6及其以前的版本中，类必须继承自的类看做是“新式” object（或者其他的内置类型），并且获得所有新式类的特性。

由于在Python 3.0中所有的类都自动是新式类，所以新式类的特性只是常规的类特性。然而，在本节中，我选择区分开它们，以便对Python 2.X代码的用户有所区分——这些代码中的类，只有在它们派生自object的时候才具有新式类的特性。

换句话说，当Python 3.0的用户在本节中看到“新式类”的叙述，它应该将其看作是对它们的类的已有特性的说明。对于Python 2.6的读者来说，还有一组可选的扩展。

在Python 2.6及其以前的版本中，唯一的编码差异是，它们要么从一个内置类型（如list）派生，要么从一个叫做object的特殊内置类派生。如果没有其他合适的内置类型可用，内置名称object就可以作为新式类的超类提供。

```
class newstyle(object):  
    ...normal code...
```

通常情况下，任何从object或其他内置类型派生的类，都会自动视为新式类。只要一个内置类型位于超类树中的某个位置，新类也当做一个新式类。不是从内置类型派生出来的类，就会当作经典类来对待。

新式类只是和经典类有细微的差别，并且它们之间的区分的方式，对于大多数主要的Python用户来说，是无关紧要的。此外，经典类形式在Python 2.6中仍然可用，并且与二十年前几乎完全一样地工作。

实际上，新式类在语法和行为上几乎与经典类完全向后兼容；它们主要只是添加了一些高级的新特性。然而，由于它们修改了一些类行为，它们必须作为一种不同的工具引入，以避免影响到依赖以前的行为的任何已有代码。例如，一些细微的区别，例如钻石模式继承搜索和带有`__getattr__`这样的管理属性方法的内置运算，如果保持不变的话，可能会导致一些遗留代码失效。

下面两个小节针对新式类的不同以及它们所提供的新工具给出概览。再一次，由于如今所有的类都是新式类，这些主题对Python 2.X读者表示了变化，但是对Python 3.0读者来说只是额外的高级类话题。

新式类变化

新式类在几个方面不同于经典类，其中的一些很细微，但可能会影响到已有的Python 2.X代码和编码方式。下面是它们不同的主要方面：

类和类型合并

类现在就是类型，并且类型现在就是类。实际上，这二者基本上是同义词。`type(I)`内置函数返回一个实例所创建自的类，而不是一个通用的实例类型，并且，通常是和`I.__class__`相同的。此外，类是`type`类的实例，`type`可能子类化为定制类创建，并且所有的类（以及由此所有的类型）继承自`object`。

继承搜索顺序

多继承的钻石模式有一种略微不同的搜索顺序，总体而言，它们可能先横向搜索再纵向搜索，并且先宽度优先搜索，再深度优先搜索。

针对内置函数的属性获取

`__getattr__`和`__getattribute__`方法不再针对内置运算的隐式属性获取而运行。这意味着，它们不再针对`__x__`运算符重载方法名而调用，这样的名称搜索从类开始，而不是从实例开始。

新的高级工具

新式类有一组新的类工具，包括`slot`、特性、描述符和`__getattribute__`方法。这些工具中的大多数都有非常特定的工具构建目的。

我们在第27章的边栏部分简单地介绍了这些变化中的3个；并且，我们将在第37章的属性管理介绍中以及第38章的私有性装饰器介绍中更深入地回顾它们。由于上面列出的第

1个变化和第2个变化可能影响到已有的Python 2.X代码，让我们在介绍新式类之前，更详细地看看这些工具。

类型模式变化

在新式类中，类型和类的区别已经完全消失了。类自身就是类型：`type`对象产生类作为自己的实例，并且类产生它们的类型的实例。实际上，像列表和字符串这样的内置类型和编写为类的用户定义类型之间没有真正的区别。这就是为什么我们可以子类化内置类型，就像本章前面所介绍的那样，由于子类化一个列表这样的内置类型，会把一个类变为新式的，因此，它变成了一个用户定义的类型。

除了允许子类化内置类型，还有一点变得非常明显的情况，就是当我们进行显式类型测试的时候。使用Python 2.6的经典类，一个类实例的类型是一个通用的“实例”，但是，内置对象的类型要更加特定：

```
C:\misc> c:\python26\python
>>> class C: pass                                     # Classic classes in 2.6
...
>>> I = C()
>>> type(I)                                           # Instances are made from classes
<type 'instance'>
>>> I.__class__
<class '__main__.C' at 0x025085A0>

>>> type(C)                                           # But classes are not the same as types
<type 'classobj'>
>>> C.__class__
AttributeError: class C has no attribute '__class__'

>>> type([1, 2, 3])
<type 'list'>
>>> type(list)
<type 'type'>
>>> list.__class__
<type 'type'>
```

但是，对于Python 2.6中的新式类，一个类实例的类型是它所创建自的类，因为类直接是用户定义的类型——实例的类型是它的类，并且，用户定义的类的类型与一个内置对象类型的类型相同。类现在有一个`__class__`属性，因为它们也是`type`的实例：

```
C:\misc> c:\python26\python
>>> class C(object): pass                             # New-style classes in 2.6
...
>>> I = C()
>>> type(I)                                           # Type of instance is class it's made from
<class '__main__.C'>
>>> I.__class__
<class '__main__.C'>
```

```

>>> type(C)                                # Classes are user-defined types
<type 'type'>
>>> C.__class__
<type 'type'>

>>> type([1, 2, 3])                        # Built-in types work the same way
<type 'list'>
>>> type(list)
<type 'type'>
>>> list.__class__
<type 'type'>

```

对于Python 3.0中的所有类都是如此，因为所有的类自动都是新式的，即便它们没有显式的超类。实际上，内置类型和用户定义类型之间的区分，在Python 3.0中消失了：

```

C:\misc> c:\python30\python
>>> class C: pass                          # All classes are new-style in 3.0
...
>>> I = C()
>>> type(I)                                # Type of instance is class it's made from
<class '__main__.C'>
>>> I.__class__
<class '__main__.C'>

>>> type(C)                                # Class is a type, and type is a class
<class 'type'>
>>> C.__class__
<class 'type'>

>>> type([1, 2, 3])                        # Classes and built-in types work the same
<class 'list'>
>>> type(list)
<class 'type'>
>>> list.__class__
<class 'type'>

```

正如你所看到的，在Python 3.0中，类就是类型，但是，类型也是类。从技术上讲，每个类都由一个元类生成——元类是这样的一个类，它要么是type自身，要么是它定制来扩展或管理生成的类的一个子类。除了影响到进行类型测试的代码，这对于工具开发者来说，是一个重要的钩子。我们将在本章后面讨论元类，并且将在第39章再次详细介绍它。

类型测试的隐含意义

除了提供内置类型定制和元类钩子，新的类模式中类和类型的融合，可能会影响到进行类型测试的代码。例如，在Python 3.0中，类实例的类型直接而有意义地比较，并且以与内置类型对象同样的方式进行。下面的代码源自于这样一个事实：类现在是类型，并且一个实例的类型是该实例的类。

```

C:\misc> c:\python30\python
>>> class C: pass
...
>>> class D: pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                     # 3.0: compares the instances' classes
False

>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)

>>> c1, c2 = C(), C()
>>> type(c1) == type(c2)
True

```

对于Python 2.6或更早版本中的经典类，比较实例类型几乎是无用的，因为所有的实例都具有相同的“实例”类型。要真正地比较类型，必须比较实例`__class__`属性（如果你关注可移植性，这在Python 3.0中也有效，但在那里不是必需的）：

```

C:\misc> c:\python26\python
>>> class C: pass
...
>>> class D: pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                     # 2.6: all instances are same type
True

>>> c.__class__ == d.__class__                             # Must compare classes explicitly
False

>>> type(c), type(d)
(<type 'instance'>, <type 'instance'>)
>>> c.__class__, d.__class__
(<class '__main__.C' at 0x024585A0>, <class '__main__.D' at 0x024588D0>)

```

并且，正如你所期待的，在这方面，Python 2.6中的新式类与Python 3.0中的所有类同样地工作——比较实例类型会自动地比较实例的类：

```

C:\misc> c:\python26\python
>>> class C(object): pass
...
>>> class D(object): pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                     # 2.6 new-style: same as all in 3.0
False

```



```
>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
```

当然，正如我们在本书中多次指出的，类型检查通常在Python程序中是错误的事情（我们编写对象接口，而不是编写对象类型），并且更加通用的`isinstance`内置函数很可能是你在极少数情况下（即必须查询实例类的类型的情况下）想要使用的。然而，知道Python的类型模型的知识，通常对于了解类模型有帮助。

所有对象派生自object

新式类模式中的另一个类型变化是，由于所有的类隐式地或显式地派生自（继承自）类`object`，并且，由于所有的类型现在都是类，所以每个对象都派生自`object`内置类，不管是直接地或通过一个超类。考虑Python 3.0中的如下交互模式（在Python 2.6中编写一个显式的`object`超类，会有等价的效果）：

```
>>> class C: pass
...
>>> x = C()

>>> type(x)                                     # Type is now class instance was created from
<class '__main__.C'>
>>> type(C)
<class 'type'>
```

和前面一样，一个类实例的类型就是它所产生自的类，并且，一个类的类型就是`type`类，因为类和类型都融合了。确实是这样，但是，实例和类都派生自内置的`object`类，因此，每个类都有一个显式或隐式的超类：

```
>>> isinstance(x, object)
True
>>> isinstance(C, object)                       # Classes always inherit from object
True
```

对于列表和字符串这样的内置类型来说，也是如此，因为在新模式中，类型是类——内置类型现在也是类，并且它们的实例也派生自`object`：

```
>>> type('spam')
<class 'str'>
>>> type(str)
<class 'type'>

>>> isinstance('spam', object)                   # Same for built-in types (classes)
True
>>> isinstance(str, object)
True
```

实际上，类型自身派生自object，并且object派生自type，即便二者是不同的对象——一个循环的关系覆盖了对象模型，并由此导致了这样一个事实：类型是生成类的类。

```
>>> type(type)                                # All classes are types, and vice versa
<class 'type'>
>>> type(object)
<class 'type'>

>>> isinstance(type, object)                   # All classes derive from object, even type
True
>>> isinstance(object, type)                   # Types make classes, and type is a class
True
>>> type is object
False
```

实际上，这种模式导致了比前面的经典类的类型/类区分的几个特殊情况，并且，它允许我们编写假设并使用一个object超类的代码。我们将在本书稍后看到示例，现在，让我们继续介绍其他的新式类变化。

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同一更高的超类）处理方式有点不同。钻石模式是高级设计概念，在Python编程中很少用到，并且在本书中目前为止还没有讨论过，因此我们没有必要深入讨论。

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。换句话说，搜索过程先水平进行，然后向上移动。搜索算法也比这里介绍的更复杂一些，但是，大多数程序员了解这些就够了。

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。此外，当从多个子类访问超类的时候，新式搜索规则避免重复访问同一超类。

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。这里，D是B和C的超类，B和C都导向相同的祖先A：

```
>>> class A:
    attr = 1                                # Classic (Python 2.6)

>>> class B(A):
    pass                                    # B and C both lead to A
```

```

>>> class C(A):
    attr = 2

>>> class D(B, C):
    pass                                # Tries A before C

>>> x = D()
>>> x.attr                             # Searches x, D, B, A
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）。

这里，对于派生自object这样的内置类的新式类，以及Python 3.0中的所有类，搜索顺序是不同的：Python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 ("object" not required in 3.0)

>>> class B(A):
    pass

>>> class C(A):
    attr = 2

>>> class D(B, C):
    pass                                # Tries C before A

>>> x = D()
>>> x.attr                             # Searches x, D, B, C
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。

在这个例子中，很可能程序员认为C应该覆盖A，尽管如此，新式类先访问C。否则，C将会在钻石环境中基本无意义：它不会定制A，并且只对同名的C使用。

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 (all 3.0)
2

```

在这里，经典类树模拟了新式类的搜索顺序：在D中为属性赋值，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 (default 2.6)
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，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）。

```

>>> 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
>>> x = D()
>>> x.meth()                                # Use default search order
A.meth                                     # Will vary per class type
                                           # Defaults to classic order in 2.6

```

```

>>> class D(B, C): meth = C.meth           # Pick C's method: new-style (and 3.0)
>>> 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，尤其是构造函数。

```

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的差异性中隔离出。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（除了在Python 2.6中，新式类需要从object或内置类型派生类以使用新式工具之外）。

注意：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都有所需的构造函数会调用A中的构造器，那该怎么办呢？），由于这样的语境在Python中很罕见，已不是本书范围之内（请参见super定制函数以获得提示——除了提供对单继承树中的超类的通用性访问，super还支持一种协作模式，以解决多继承树中的一些冲突）。

搜索顺序变化的范围

总而言之，默认情况下，钻石模式对于经典类和新式类进行不同的搜索，并且这是一个非向后兼容的变化。此外要记住，这种变化主要影响到多继承的钻石模式情况。新式类继承对于大多数其他的继承树结构都是不变的工作。此外，整个问题不可能在理论上比实践中更重要，因为，新式类搜索直到Python 2.2才足够显著地解决，并且在Python 3.0中才成为标准，它不可能影响到太多的Python代码。

正如已经提到的，我还应该注意到，即便你没有在自己编写的类中用到钻石模式，由于隐式的object超类在Python 3.0中的每个类之上，所以如今多继承的每个例子都展示了钻石模式。也就是说，在新式类中，object自动扮演了我们前面所讨论的实例中类A的

角色。因此，新的类搜索规则不仅修改了逻辑语义，而且通过避免多次访问相同的类而优化了性能。

同样重要的是，新模式中的隐式`object`超类为各种内置操作提供了默认方法，包括`__str__`和`__repr__`显示格式化方法。运行一个`dir(object)`来看看提供了哪些方法。没有一个新式的搜索顺序，在多继承情况中，`object`中的默认方法将总是覆盖用户编写的类中的重新定义，除非这些重定义总是放在最左边的超类之中。换句话说，新类模式自身使得使用新搜索顺序更关键！

要了解Python 3.0中隐式`object`超类的一个更清楚示例，以及该对象所创建的钻石模式的其他示例，参见上一章`lister.py`示例中的`ListTree`类的输出，以及第28章中的`classtree.py`示例。

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。下面的小节将对这些额外特性中的每一个给出概览，这些特性在Python 2.6的新式类和Python 3.0的所有类中都可用。

slots实例

将字符串属性名称顺序赋值给特殊的`__slots__`类属性，新式类就有可能既限制类的实例将有的合法属性集，又能够优化内存和速度性能。

这个特殊属性一般是在`class`语句顶层内将字符串名称顺序赋值给变量`__slots__`而设置：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'
```

Slot对于Python的动态特性来说是一种违背，而动态特性要求任何名称都可以通过赋值

来创建。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。如果创建了很多实例并且只有几个属性是必需的话，那么为每个实例对象分配一个命名空间字典可能在内存方面代价过于昂贵。要节省空间和执行速度（程度对每个程序而言有所不同），slot属性可以顺序存储以供快速查找，而不是为每个实例分配一个字典。

Slot和通用代码

实际上，有些带有slots的实例也许根本没有__dict__属性字典，使得有些书中所写的元程序过于复杂（包括本书中的一些代码）。工具根据字符串名称通用地列出属性或访问属性，例如，必须小心使用比__dict__更为存储中立的工具，例如getattr、setattr和dir内置函数，它们根据__dict__或__slots__存储应用于属性。在某些情况下，两种属性源代码都需要查询以确保完整性。

例如，使用slots的时候，实例通常没有一个属性字典——Python使用第37章介绍的类描述符功能来为实例中的slot属性分配空间。只有slot列表中的名称可以分配给实例，但是，基于slot的属性仍然可以使用通用工具通过名称来访问或设置。在Python 3.0中（以及在Python 2.6中派生自object的类中）：

```
>>> class C:
...     __slots__ = ['a', 'b']           # __slots__ means no __dict__ by default
...
>>> X = C()
>>> X.a = 1
>>> X.a
1
>>> X.__dict__
AttributeError: 'C' object has no attribute '__dict__'
>>> getattr(X, 'a')
1
>>> setattr(X, 'b', 2)                 # But getattr() and setattr() still work
>>> X.b
2
>>> 'a' in dir(X)                       # And dir() finds slot attributes too
True
>>> 'b' in dir(X)
True
```

没有一个属性命名空间字典，不可能给不是slots列表中名称的实例来分配新的名称：

```
>>> class D:
...     __slots__ = ['a', 'b']
...     def __init__(self): self.d = 4   # Cannot add new names if no __dict__
...
>>> X = D()
AttributeError: 'D' object has no attribute 'd'
```


然而，通过在`__slots__`中包含`__dict__`仍然可以容纳额外的属性，从而考虑到一个属性空间字典的需求。在这个例子中，两种存储机制都用到了，但是，`getattr`这样的通用工具允许我们将它们当做单独一组属性对待：

```
>>> class D:
...     __slots__ = ['a', 'b', '__dict__']      # List __dict__ to include one too
...     c = 3                                  # Class attrs work normally
...     def __init__(self): self.d = 4         # d put in __dict__, a in __slots__
...
>>> X = D()
>>> X.d
4
>>> X.__dict__                                # Some objects have both __dict__ and __slots__
{'d': 4}                                       # getattr() can fetch either type of attr
>>> X.__slots__
['a', 'b', '__dict__']
>>> X.c
3
>>> X.a                                         # All instance attrs undefined until assigned
AttributeError: a
>>> X.a = 1
>>> getattr(X, 'a'), getattr(X, 'c'), getattr(X, 'd')
(1, 3, 4)
```

然而，想要通用地列出所有实例属性的代码，可能仍然需要考虑两种存储形式，因为`dir`也返回继承的属性（这依赖于字典迭代器来收集键）：

```
>>> for attr in list(X.__dict__) + X.__slots__:
...     print(attr, '=>', getattr(X, attr))

d => 4
a => 1
b => 2
__dict__ => {'d': 4}
```

由于两种都可能忽略，更正确的编码方式如下所示（`getattr`考虑到默认情况）：

```
>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
...     print(attr, '=>', getattr(X, attr))

d => 4
a => 1
b => 2
__dict__ => {'d': 4}
```

超类中的多个`__slot__`列表

然而，注意，这段代码只是解决了由一个实例继承的最低`__slots__`属性中的slot名称。如果类树中的多个类都有自己的`__slots__`属性，通用的程序必须针对列出的属性开发其他的策略（例如，把slot名称划分为类的属性，而不是实例的属性）。

slot声明可能出现在一个类树中的多个类中，但是，它们受到一些限制，除非你理解slot作为类级别描述符（这是我们将在本书最后一部分要学习的一种工具）的实现，否则要说明这些限制有些困难：

- 如果一个子类继承自一个没有__slots__的超类，那么超类的__dict__属性总是可以访问的，使得子类中的一个__slots__无意义。
- 如果一个类定义了与超类相同的slot名称，超类slot定义的名称版本只有通过直接从超类获取其描述符才能访问。
- 由于一个__slots__声明的含义受到它出现其中的类的限制，所以子类将有一个__dict__，除非它们也定义了一个__slots__。
- 通常从列出实例属性这方面来讲，多类中的slots可能需要手动类树爬升、dir用法，或者把slot名称当做不同的名称领域的政策：

```
>>> class E:
...     __slots__ = ['c', 'd']           # Superclass has slots
...
>>> class D(E):
...     __slots__ = ['a', '__dict__']   # So does its subclass
...
>>> X = D()
>>> X.a = 1; X.b = 2; X.c = 3           # The instance is the union
>>> X.a, X.c
(1, 3)

>>> E.__slots__                         # But slots are not concatenated
['c', 'd']
>>> D.__slots__
['a', '__dict__']
>>> X.__slots__                         # Instance inherits *lowest* __slots__
['a', '__dict__']
>>> X.__dict__                          # And has its own an attr dict
{'b': 2}

>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
...     print(attr, '=>', getattr(X, attr))
...
b => 2                                  # Superclass slots missed!
a => 1
__dict__ => {'b': 2}

>>> dir(X)                             # dir() includes all slot names
[...many names omitted... 'a', 'b', 'c', 'd']
```

当这种通用性可能的时候，slots可能最好当做类属性来对待，而不是试图让它们表现出与常规类属性一样。要了解关于slots的更多内容，参见Python的标准手册。此外，参见第38章中考虑到基于__slots__和__dict__存储的属性的一个示例。

要了解为什么通用程序可能需要关注slots的一个例子，请参阅上一章的多继承小节中的 *lister.py* 显示混入类示例；那里的一个提示描述了示例的slot内容。在这样一个试图通用地列出属性的工具中，slot用法需要额外的代码或者实现相应的政策，以通用地处理基于slot的属性。

类特性

有一种称为特性（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29章谈过、目前用得很多的 `__getattr__` 和 `__setattr__` 重载方法的替代做法。特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。特性（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，特性是一种对象，赋值给类属性名称。特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。特性一般都是在 `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，改用特性来编写。（注意，特性对于所有的类可用，但是，对于拦截属性赋值，必须是Python 2.6中 `object` 派生的新式对象才有效）：

```
>>> 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，特性比起传统技术不是那么复杂，而且运行起来更快。例如，当我们新增属性赋值运算支持时，特性就变得更加有吸引力：输入的代码更少，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):
...         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（或者，对于新式类，会导向object超类的__setattr__）。

```
>>> 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，特性似乎是赢家。然而，__getattr__和__setattr__的某些应

用依然需要更为动态或通用的接口，超出特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的情况，特性大致上就只是选用的扩展功能了。

要了解两个选项的详细内容，请参阅本书最后一部分的第37章。我们将从那里看到，使用函数装饰器语法来编写特性是可能的，这是本章稍后将要介绍的一个主题。

`__getattribute__`和描述符

`__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，它远比`__getattr__`和`__setattr__`难用，而且很像`__setattr__`多用于循环，但二者的用法不同。

除了特性和运算符重载方法，Python支持属性描述符的概念——带有`__get__`和`__set__`方法的类，分配给类属性并且由实例继承，这拦截了对特定属性的读取和写入访问。描述符在某种意义上是特性的一种更加通用的形式。实际上，特性是定义特定类型描述符的一种简化方式，该描述符运行关于访问的函数。描述符还用来实现我们前面所介绍的slots特性。

由于特性、`__getattribute__`和描述符都是有些高级的话题，我们将推迟到后面介绍；还有关于特性的更多内容，也将在本书最后一部分的第37章介绍。

元类

新式类的大多数变化和功能增加，都是与本章前面提到的可子类化的类型的概念密切相连，因为在Python 2.2及其以后的版本中，可子类化的类型和新式类与类型和类的合并一起引入。正如我们所看到的，在Python 3.0中，合并完成了：类现在是类型，并且类型也是类。

除了这些变化，Python还针对编写元类增加了一种更加一致的协议，元类是子类化了`type`对象并且拦截类创建调用的类。此外，它们还为管理和扩展类对象提供了一种定义良好的钩子。它们也是大多数Python程序员可选的高级话题，因此，我们将推迟介绍其具体细节。我们将在本章稍后遇到元类和类装饰器一起使用，并且，我们将在本书最后一部分的第39章中详细介绍它们。

静态方法和类方法

在Python 2.2中，有可能在类中定义两种方法，它们不用一个实例就可以调用：**静态方法**大致与一个类中的简单的无实例函数类似地工作，**类方法**传递一个类而不是一个实例。尽管这一功能与前面小节所介绍的新式类一起添加，静态方法和类方法只对经典类有效。

要使用这些方法，必须在类中调用特殊的内置函数，分别名为`staticmethod`和`classmethod`，或者使用我们将在本章后面遇到的装饰语法来调用。在Python 3.0中，无实例的方法只通过一个类名调用，而不需要一个`staticmethod`声明，但是这样的方法确实通过实例来调用。

为什么使用特殊方法

正如我们已经学习过的，类方法通常在其第一个参数中传递一个实例对象，以充当方法调用的一个隐式主体。然而今天，有两种方法来修改这种模式。在说明这两种方法之前，我应该介绍一下为什么这与你有关。

有时候，程序需要处理与类而不是与实例相关的数据。考虑要记录由一个类创建的实例的数目，或者维护当前内存中一个类的所有实例的列表。这种类型的信息及其处理与类相关，而非与其实例相关。也就是说，这种信息通常存储在类自身上，不需要任何实例也可以处理。

对于这样的任务，一个类之外的简单函数编码往往就能够胜任——因为它们可以通过类名访问类属性，它们能够访问类数据并且不需要通过一个实例。然而，要更好地把这样的代码与一个类联系起来，并且允许这样的过程像通常一样用继承来定制，在类自身之中编写这类函数将会更好。为了做到这点，我们需要一个类中的方法不仅不传递而且也不期待一个`self`实例参数。

Python通过**静态方法**的概念来支持这样的目标——嵌套在一个类中的没有`self`参数的简单函数，并且旨在操作类属性而不是实例属性。静态方法不会接受一个自动的`self`参数，不管是通过一个类还是一个实例调用。它们通常会记录跨所有实例的信息，而不是为实例提供行为。

尽管较少用到，Python还支持**类方法**的概念，这是类的一种方法，传递给它们的第一个参数是一个类对象而不是一个实例，不管是通过一个实例或一个类调用它们。即便是通过一个实例调用，这样的方法也可以通过它们的`self`类参数来访问类数据。常规的方法（现在正规的叫法是**实例方法**）在调用的时候仍然接受一个主题实例，静态方法和类方法则不会。

Python 2.6和Python 3.0中的静态方法

静态方法的概念在Python 2.6和Python 3.0中都是相同的，但是，它的实现需求在Python 3.0中有所发展。由于本书涉及了两个版本，所以在开始接触代码之前，我们先来看看两种底层模式的不同。

实际上，我们已经在前一章中开始介绍这一主题了，当我们介绍未绑定方法的概念的时候。还记得，Python 2.6和Python 3.0总是给通过一个实例调用的方法传递一个实例。然而，Python 3.0对待从一个类直接获取的方法，与Python 2.6有所不同：

- 在Python 2.6中，从一个类获取一个方法会产生一个**未绑定方法**，没有手动传递一个实例的就不会调用它。
- 在Python 3.0中，从一个类获取一个方法会产生一个**简单函数**，没有给出实例也可以常规地调用。

换句话说，Python 2.6类方法总是要求传入一个实例，不管是通过一个实例或类调用它们。相反，在Python 3.0中，只有当一个方法期待实例的时候，我们才给它传入一个实例——没有一个self实例参数的方法可以通过类调用而不需要传入一个实例。也就是说，Python 3.0允许类中的简单函数，只要它们不期待并且也不传入一个实例参数。直接效果是：

- 在Python 2.6中，我们必须总是把一个方法声明为静态的，从而不带一个实例而调用它，不管是通过一个类或一个实例调用它。
- 在Python 3.0中，如果方法只通过一个类调用的话，我们不需要将这样的方法声明为静态的，但是，要通过一个实例调用它，我们必须这么做。

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。下面的文件`spam.py`做出了最初的尝试，它的类把一个计数器存储为类属性，每次创建一个新的实例的时候，构造函数都会对计数器加1，并且，有一个显示计数器值的方法。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，从而确保它可以在所有的实例中使用：

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)
```

`printNumInstances`方法旨在处理类数据而不是实例数据——它是关于**所有实例**的，而不是某个特定的实例。因此，我们想要不必传递一个实例就可以调用它。实际上，我们不

想生成一个实例来获取实例的数目，因为这可能会改变我们想要获取的实例的数目！换句话说，我们想要一个无self的“静态”方法。

然而，这段代码是否有效，取决于我们所使用的Python，以及我们调用方法的方式——通过类或者通过一个实例。在Python 2.6中（以及更通常的Python 2.X），通过类和实例调用无self方法函数都将失效（考虑到篇幅，我省略了一些错误提示）：

```
C:\misc> c:\python26\python
>>> from spam import Spam
>>> a = Spam()                                # Cannot call unbound class methods in 2.6
>>> b = Spam()                                # Methods expect a self object by default
>>> c = Spam()

>>> Spam.printNumInstances()
TypeError: unbound method printNumInstances() must be called with Spam instance
as first argument (got nothing instead)
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
```

这里的问题在于，在Python 2.6中无绑定实例的方法并不完全等同于简单函数。即便在def头部没有参数，该方法在调用的时候仍然期待一个实例，因为该函数与一个类相关。在Python 3.0中（以及随后的Python 3.X版本中），对一个无self方法的调用使得通过类调用有效，但从实例调用失效：

```
C:\misc> c:\python30\python
>>> from spam import Spam
>>> a = Spam()                                # Can call functions in class in 3.0
>>> b = Spam()                                # Calls through instances still pass a self
>>> c = Spam()

>>> Spam.printNumInstances()                  # Differs in 3.0
Number of instances created: 3
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
```

也就是说，对于printNumInstances这样的无实例方法的调用，在Python 2.6中通过类进行调用将会失效，但是在Python 3.0中将有效。另一方面，通过一个实例调用在两个版本的Python中都会失效，因为一个实例自动传递给方法，而该方法没有一个参数来接收它：

```
Spam.printNumInstances()                    # Fails in 2.6, works in 3.0
instance.printNumInstances()                # Fails in both 2.6 and 3.0
```

如果你能够使用Python 3.0并且坚持只通过类调用无self方法，你已经有了一个静态方法特性。然而，要允许非self方法在Python 2.6中通过类调用，并且在Python 2.6和Python 3.0中都通过实例调用，你需要采取其他设计，或者能够把这样的方法标记为特殊的。让我们依次看看这两种选项。

静态方法替代方案

如果不能使得一个无self方法称为特殊的，有一些不同的编码结构可以尝试。如果想要调用没有一个实例而访问类成员的函数，可能最简单的思路就是仅在类之外生成它们的简单函数，而不是类方法。通过这种方式，调用中不会期待一个实例。例如，对`spam.py`的如下修改在Python 3.0和Python 2.6中都有效（虽然在Python 2.6中会对其`print`语句显示额外的圆括号）：

```
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()           # But function may be too far removed
Number of instances created: 3         # And cannot be changed via inheritance
>>> 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。

在Python中的静态方法之前，这一结构是通用的方法。由于Python已经把模块提供为命名空间分隔工具，因此可以确定通常不需要把函数包装到一个类中，除非它们实现了对象行为。像这里这样的模块中的简单函数，做了无实例类方法的大多数工作，并且已经与类关联起来，因为它们位于同一模块中。

遗憾的是，这种方法仍然不是理想的。其一，它给该文件的作用域添加了一个额外的名称，该名称只用来处理单个的类。此外，该函数与类的直接关联很小；实际上，它的定义可能在数百行代码之外的位置。可能更糟糕的是，像这样的简单函数不能通过继承定制，由此，它们位于类的命名空间之外：子类不能通过重新定义这样的函数来直接替代或扩展它。

我们可能想要像通常那样使用一个常规方法并总是通过一个实例调用它，从而使得这个例子以独立于版本的方式工作：

```
class Spam:
    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
>>> Spam.printNumInstances(a)
Number of instances created: 3
>>> Spam().printNumInstances()           # But fetching counter changes counter!
Number of instances created: 4

```

遗憾的是，正如前面所提到的，如果我们没有一个实例可用，并且产生一个实例来改变类数据，就像这里的最后一行所说明的那样，这样的方法完全是无法工作的。更好的解决方案可能是在类中把一个方法标记为不需要一个实例。下一小节展示了如何做到这点。

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。它们都把一个函数标记为特殊的，例如，如果是静态方法的话不需要实例，如果是一个类方法的话需要一个类参数。例如：

```

class Methods:
    def imeth(self, x):                # Normal instance method: passed a self
        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。

从技术上讲，Python现在支持三种类相关方法：**实例**、**静态**和**类**。此外，Python 3.0也允许类中的简单函数在通过一个类调用的时候充当静态方法的角色，而不需要额外的协议，从而扩展了这一模式。

实例方法是我们在本书中所见的常规的（并且是默认的）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（为了简单起见，我们在这样的交互会话中省略了一些类导入）。

```
>>> obj = Methods()                # Make an instance

>>> obj.imeth(1)                    # Normal method, call through instance
<__main__.Methods object...> 1    # Becomes imeth(obj, 1)

>>> Methods.imeth(obj, 2)           # Normal method, call through class
<__main__.Methods object...> 2    # Instance passed explicitly
```

反之，**静态方法**调用时不需要实例参数。与类之外的简单函数不同，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。非实例函数通常在Python 3.0中可以通过类调用，但是，这在Python 2.6中并非默认的。使用`staticmethod`内置方法允许这样的方法在Python 3.0中通过一个实例调用，而在Python 2.6中通过类和实例调用（前者在Python 3.0中没有`staticmethod`也能工作，但后者不可以）：

```
>>> Methods.smeth(3)               # Static method, call through class
3                                  # No instance passed or expected

>>> obj.smeth(4)                   # Static method, call through instance
4                                  # Instance not passed
```

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，不管它是通过一个类或一个实例调用：

```
>>> Methods.cmeth(5)               # Class method, call through class
<class '__main__.Methods'> 5      # Becomes cmeth(Methods, 5)

>>> obj.cmeth(6)                   # Class method, call through instance
<class '__main__.Methods'> 6      # Becomes cmeth(Methods, 6)
```

使用静态方法统计实例

现在，有了这些内置函数，如下是本节的实例统计示例的静态方法等价形式——它把方法标记为特殊的，以便不会自动传递一个实例：

```
class Spam:
    numInstances = 0                # Use static method for class data
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances:", Spam.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

使用静态方法内置函数，我们的代码现在允许在Python 2.6和Python 3.0中通过类或任何实例来调用无self方法：

```
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()           # Call as simple function
Number of instances: 3
>>> a.printNumInstances()             # Instance argument not passed
Number of instances: 3
```

与把printNumInstances移到类之外的做法（如之前所做的）相比较，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冲突），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），并且允许子类用集成定制静态方法——这是比超类编码中从文件导入函数更方便的一种方法。如下是子类以及新的测试会话：

```
class Sub(Spam):
    def printNumInstances():           # Override a static method
        print("Extra stuff...")      # But call back to original
        Spam.printNumInstances()
    printNumInstances = staticmethod(printNumInstances)

>>> a = Sub()
>>> b = Sub()
>>> a.printNumInstances()             # Call from subclass instance
Extra stuff...
Number of instances: 2
>>> Sub.printNumInstances()           # Call from subclass itself
Extra stuff...
Number of instances: 2
>>> Spam.printNumInstances()
Number of instances: 2
```

此外，类可以继承静态方法而不用重新定义它，它可以没有一个实例而运行，不管定义于类树的何处：

```
>>> class Other(Spam): pass           # Inherit static method verbatim

>>> c = Other()
>>> c.printNumInstances()
Number of instances: 3
```

用类方法统计实例

有趣的是，类方法也可以做类似的工作——如下代码与前面列出的静态方法版本具有相同的行为，但是，它使用一个类方法来把实例的类接收到其第一个参数中。类方法使用通用的自动传递类对象，而不是硬编码类名称：

```

class Spam:
    numInstances = 0                                # Use class method instead of static
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances:", cls.numInstances)
        printNumInstances = classmethod(printNumInstances)

```

这个类与前面的版本使用方式相同，但是通过类和实例调用`printNumInstances`方法的时候，它接受类而不是实例：

```

>>> a, b = Spam(), Spam()
>>> a.printNumInstances()                        # Passes class to first argument
Number of instances: 2
>>> Spam.printNumInstances()                    # Also passes class to first argument
Number of instances: 2

```

当使用类方法的时候，别忘了，它们接收调用的主体的最具体（低层）的类。当试图通过传入类更行类数据的时候，这具有某些细微的隐藏含义。例如，如果在模块`test.py`中我们像前面那样对定制子类化，扩展`Spam.printNumInstances`以显示其`cls`参数，并且开始一个新的测试会话：

```

class Spam:
    numInstances = 0                                # Trace class passed in
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances:", cls.numInstances, cls)
        printNumInstances = classmethod(printNumInstances)

class Sub(Spam):
    def printNumInstances(cls):                      # Override a class method
        print("Extra stuff...", cls)                # But call back to original
        Spam.printNumInstances()
        printNumInstances = classmethod(printNumInstances)

class Other(Spam): pass                            # Inherit class method verbatim

```

无论何时运行一个类方法的时候，最低层的类传入，即便对于没有自己的类方法的子类：

```

>>> x, y = Sub(), Spam()
>>> x.printNumInstances()                        # Call from subclass instance
Extra stuff... <class 'test.Sub'>
Number of instances: 2 <class 'test.Spam'>
>>> Sub.printNumInstances()                      # Call from subclass itself
Extra stuff... <class 'test.Sub'>
Number of instances: 2 <class 'test.Spam'>
>>> y.printNumInstances()
Number of instances: 2 <class 'test.Spam'>

```

这里的第一个调用中，通过Sub子类的一个实例调用了一个类方法，并且Python传递了最低的类，Sub，给该类方法。在这个例子中，由于该方法的Sub重定义显式地调用了Spam超类的版本，Spam中的超类方法在第一个参数中接收自己。但是，对于直接继承类方法的一个对象，看看发生了什么：

```
>>> z = Other()
>>> z.printNumInstances()
Number of instances: 3 <class 'test.Other'>
```

这里的最后一个调用把Other传递给了Spam的类方法。这在这个例子中也有效，因为它通过继承获取了在Spam中找到的计数器。如果该方法试图把传递的类的数据赋值，它将更新Object，而不是Spam。在这个特定的例子中，可能Spam通过直接编写自己的类名来更新其数据会更好，而不是依赖于传入的类参数。

使用类方法统计每个类的实例

实际上，由于类方法总是接收一个实例树中的最低类：

- 静态方法和显式类名称可能对于处理一个类本地的数据来说是更好的解决方案。
- 类方法可能更适合处理对层级中的每个类不同的数据。

代码需要管理每个类实例计数器，这可能会更好地利用类方法。在下面的代码中，顶层的超类使用一个类方法来管理状态信息，该信息根据树中的每个类都不同，而且存储在类上——这类似于实例方法管理类实例中状态信息的方式：

```
class Spam:
    numInstances = 0
    def count(cls):
        cls.numInstances += 1
    def __init__(self):
        self.count()
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):
        Spam.__init__(self)

class Other(Spam):
    numInstances = 0

>>> x = Spam()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```


静态方法和类方法都有其他高级的作用，我们将在这里略过，请参见其他资源了解更多的使用示例。在最近的Python版本中，随着装饰语法的出现，静态方法和类方法的设计都变得更加简单——装饰语法是把一个函数应用于另一个函数的一种方法，该语法比它所激励的静态方法的使用具有更好的用途。这一语法允许我们在Python 2.6和Python 3.0中扩展类，以初始化最后一个示例中numInstances这样的计数器的数据。下一小节说明了如何做到这点。

装饰器和元类：第一部分

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。

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30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，标识静态方法，但是程序员也可以编写自己的任意装饰器。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。在Python 2.6和Python 3.0中，也有更新的相关扩展可用：**类装饰器**直接绑定到类模式，并且它们的用途与**元类**有所重叠。

函数装饰器基础

从语法上来讲，函数装饰器是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装饰器语法编写。

```
class C:
    @staticmethod                # Decoration syntax
    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地运行）。

经过这些添加，有了在Python 2.6或Python 3.0中编写前一节中的静态方法示例的一种更好的方法（`classmethod`装饰器以同样的方式使用）：

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

    @staticmethod
    def printNumInstances():
        print("Number of instances created: ", Spam.numInstances)
a = Spam()
b = Spam()
c = Spam()
Spam.printNumInstances()           # Calls from both classes and instances work now!
a.printNumInstances()              # Both print "Number of instances created: 3"
```

记住，`staticmethod`仍然是一个内置函数；它可以用于装饰语法中，只是因为它把一个函数当做参数并且返回一个可调用对象。实际上，任何这样的函数都可以以这种方式使用，即便是下节介绍的我们自己编写的用户定义函数。

装饰器例子

尽管Python提供了很多内置函数，它们可以用作装饰器，我们也可以自己编写定制装饰器。由于它们的广泛应用，我们准备在本书的下一部分中用整整一章的篇幅来编写装饰器。作为一个快速的示例，让我们看看一个简单的用户定义的装饰器的应用。

回想第29章，`__call__`运算符重载方法为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):
    print(a, b, c)

spam(1, 2, 3)
spam('a', 'b', 'c')
spam(4, 5, 6)

```

Same as spam = tracer(spam)
Wrap spam in a decorator object

Really calls the tracer wrapper object
Invokes __call__ in class
__call__ adds logic and runs original object

因为spam函数是通过tracer装饰器执行的，所以当最初的变量名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。这个装饰器对于任何接受位置参数的函数都有效，但是，它没有返回已装饰函数的结果，没有处理关键字参数，并且不能装饰类方法函数（简而言之，对于其__call__方法将只传递一个tracer实例）。正如我们将在第八部分见到的，有各种各样的方式来编写函数装饰器，包括嵌套def语句；其中的一些替代方法比这里给出的版本更适合于方法。

类装饰器和元类

函数装饰器如此有用，以至于Python 2.6和Python 3.0都扩展了这一模式，允许装饰器应用于类和函数。简而言之，**类装饰器**类似于函数装饰器，但是，它们在一条class语句的末尾运行，并且把一个类名重新绑定到一个可调用对象。同样，它们可以用来管理类（在类创建之后），或者当随后创建实例的时候插入一个包装逻辑层来管理实例。代码结构如下：

```

def decorator(aClass): ...

@decorator
class C: ...

```

被映射为下列相当代码：

```
def decorator(aClass): ...

class C: ...
C = decorator(C)
```

类装饰器也可以扩展类自身，或者返回一个拦截了随后的实例构建调用的对象。例如，在本章前面的“用类方法统计每个类的实例”小节的示例中，我们使用这个钩子来自动地扩展了带有实例计数器和任何其他所需数据的类：

```
def count(aClass):
    aClass.numInstances = 0
    return aClass                                # Return class itself, instead of a wrapper
@count
class Spam: ...                                # Same as Spam = count(Spam)

@count
class Sub(Spam): ...                           # numInstances = 0 not needed here

@count
class Other(Spam): ...
```

元类是一种类似的基于类的高级工具，其用途往往与类装饰器有所重合。它们提供了一种可选的模式，会把一个类对象的创建导向到顶级`type`类的一个子类，在一条`class`语句的最后：

```
class Meta(type):
    def __new__(meta, classname, supers, classdict): ...

class C(metaclass=Meta): ...
```

在Python 2.6中，效果是相同的，但是编码是不同的——在类头部中使用一个类属性而不是一个关键字参数：

```
class C:
    __metaclass__ = Meta
    ...
```

元类通常重新定义`type`类的`__new__`或`__init__`方法，以实现对一个新的类对象的创建和初始化的控制。直接效果就像类装饰器一样，是定义了类创建时自动运行的代码。两种方法都可以用来扩展一个类或返回一个任意的对象来替代它——几乎是拥有无限的、基于类的可能性的一种协议。

更多详细信息

当然，对于装饰器和元类的内容还有很多超出了我在这里所介绍的内容。尽管它们是一种通用的机制，装饰器和元类仍然是高级功能，对于工具编写者有意义，但对于应用程序员就没那么重要，因此，我们将推迟到本书最后一部分更多地介绍它们：

- 第37章介绍如何使用函数装饰器语法来编写特性。
- 第38章更详细地介绍装饰器，包括更全面的示例。
- 第39章介绍元类，以及关于类和实例管理的更多内容。

尽管这些章介绍了高级话题，它们还为我们提供了机会，可以看到Python在一些更真实的示例中的应用，而这些示例比本书其他部分所提供的示例都要现实。

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映在所有实例中，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。我们在第26章已经学习过，可以修改类的属性而不修改实例，就可以达到相同的目的。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'

```

修改可变的类属性也可能产生副作用

这个陷阱其实只是前面的陷阱的扩展。由于类属性由所有实例共享，所以如果一个类属性引用一个可变对象，那么从任何实例来原处修改该对象都会立刻影响到所有实例：

```

>>> class C:
...     shared = []          # Class attribute
...     def __init__(self):
...         self.perobj = [] # Instance attribute
...
>>> x = C()                 # Two instances
>>> y = C()                 # Implicitly share class attrs
>>> y.shared, y.perobj
([], [])

>>> x.shared.append('spam')  # Impacts y's view too!
>>> x.perobj.append('spam')  # Impacts x's data only
>>> x.shared, x.perobj
(['spam'], ['spam'])

```

```
>>> y.shared, y.perobj          # y sees change made through x
(['spam'], [])
>>> C.shared                    # Stored on class and shared
['spam']
```

这个效果与我们在本书中已经见到过的很多效果没有区别：可变对象通过简单变量来共享，全局变量由函数共享，模块级的对象由多个导入者共享，可变的函数参数由调用者和被调用者共享。所有这些都是通用行为的例子，并且如果从任何引用原处修改共享的对象的话，对一个可变对象的多个引用都将受到影响。在这里，这通过继承发生于所有实例所共享的类属性中，但是，这也是同样的现象在发挥作用。通过对实例属性自身的赋值的不同行为，这可能会更含蓄地发生：

```
x.shared.append('spam')          # Changes shared object attached to class in-place
x.shared = 'spam'                # Changed or creates instance attribute attached to x
```

但是，再一次说明，这不是一个问题，它只是需要注意的事情；共享的可变类属性在Python程序中可能有很多有效的用途。

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30章多重继承的例子中，假设Super类也实现了__str__方法。

```
class ListTree:
    def __str__(self): ...

class Super:
    def __str__(self): ...

class Sub(ListTree, Super):          # Get ListTree's __str__ by listing it first

x = Sub()                           # Inheritance searches ListTree before Super
```

我们想要继承ListTree的还是Super的？由于继承搜索是从左至右进行的，我们会从先列在Sub类首行的那个类取得该方法。假设，我们先编写ListTree，因为这个类的整个目的就是其定制了的__str__（实际上，当把这个类与拥有自己的一个__str__的tkinter.Button混入的时候，我们必须这么做）。

但现在，假设Super和ListTree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ListTree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。

```
class ListTree:
    def __str__(self): ...
```



```

    def other(self): ...

class Super:
    def __str__(self): ...
    def other(self): ...

class Sub(ListTree, Super):
    other = Super.other          # Get ListTree's __str__ by listing it first
    def __init__(self):          # But explicitly pick Super's version of other
        ...

x = Sub()                       # Inheritance searches Sub before ListTree/Super

```

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ListTree.other（继承搜索时正常会找到的属性）。同样，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ListTree中的方法。

```

class Sub(Super, ListTree):
    __str__ = Lister.__str__      # Get Super's other by order
                                # Explicitly pick Lister.__str__

```

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，对于任意关系较远的子类中变量的含义，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”模式时所提到的明确解决冲突。）

经验法则是，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。之前第30章研究过的伪私有__x属性功能可以把类依赖的变量名本地化，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ListTree只是要导出特殊的__str__，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):
            print(Spam.count)
    return Spam()

generate().method()

C:\python\examples> python nester.py
...error text omitted...
    Print(Spam.count)
NameError: Spam
```

Fails prior to Python 2.2, works later

Name Spam not visible:
not local (def), global (module), built-in

Not local (def), global (module), built-in

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（参考之后的可能解决方案）。

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
    class Spam:
        count = 1
        def method(self):
            print(Spam.count)
    return Spam()

generate().method()
```

Force Spam to module scope

Works: in global (enclosing module)

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`。

```
def generate():
    return Spam()

class Spam:
    count = 1
    def method(self):
```

Define at top level of module

```

        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()

```

Python中基于委托的类：`__getattr__`和内置函数

我们在第27章的类教程和第30章的委托介绍中简单地遇到过这个问题：使用`__getattr__`运算符重载方法来把属性获取委托给包装的对象的类，在Python 3.0中将失效，除非运算符重载方法在包装类中重新定义了。在Python 3.0（和Python 2.6中，当使用新式类的时候），内置操作没有导向到通用的属性拦截方法，从而隐式地获取运算符重载方法的名称。例如，打印所使用的`__str__`方法，不会调用`__getattr__`。相反，Python 3.0在类中查找这样的名字，并且完全略过常规的运行时实例查找机制。为了解决这一点，这样的方法必须在包装类中重定义，要么手动，要么使用工具，或者在超类中重新定义。我们将在第37章和第38章中回顾这一陷阱。

“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，系统的逻辑封装得太深，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，并且保持类层次的简短和平坦（除非有充足的理由不这样），就能够让调试变得简单，也有助于提高代码的可维护性。

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、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。正如前面所提到的，我们对于一些高级类工具的介绍，将在本书的最后一部分中继续；如果你需要了解关于特性、描述符、装饰器和元类的更多细节，请参阅后面的内容。

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，之所以留到最后，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。

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？

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继承了。
2. 函数修饰器通常是用来给现存的函数增加函数每次被调用时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，所有的类都将会自动成为新式类，因此不需要这么派生；在Python 2.6中，这样派生出来的类是新式类，那些没有派生的类是“经典类”。

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）。新式类还针对实例和类修改了`type`内置函数的结果，针对内置操作方法，没有运行`__getattr__`这样的通用属性获取方法，并且支持包括特性、描述符和`__slots__`实例属性列表这样的一组高级额外工具。
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。Python 3.0允许通过类而没有这个步骤就调用类中的简单函数，但是，通过实例调用仍然需要静态方法声明。

第六部分练习题

这些练习题会让你编写一些类，并且对一些现有的代码做些实验。当然，在现有代码中问题是必定会存在的。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函数保存一个对象（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.data, Y)`]。哪里是最适合放置构造函数和操作符重载方法的地方（也就是说，在哪个类里）？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`将那个参数加载到实例当前的`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。另外，为类提供一个构造函数接受现有的列表（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？
 - 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`+`和分片这样的操作应该返回什么类型的对象呢？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`Mylist`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，该如何编写呢？（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呢？用你编写的类试试加法、索引以及分片运算（注意：基于`__getattr__`的一个完全通用的方法在Python 2.6下有效，但在Python 3.0下无效，第30章提及了原因，并且本练习的解答中再次给出了原因）。
5. 集合对象。使用在“通过嵌入扩展类型”小节中所描述的集合类进行实验。运行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18章中类似的算法）。使用集合子类来计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__`可以将大多数的列表方法调用传递给被包装的列表。）
6. 类树链接。在第28章“命名空间：完整的内容”和第30章“多重继承：‘混合’类”节都提到了类有一个`__bases__`属性，它会返回它们的超类对象的元组（在类首行中括号中的对象）。使用`__bases__`来扩展`lister.py`混合类（参考第30章），以便能够打印实例类的直接超类的名称。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:
```

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

购买食物的顾客。

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 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

class Food:
    def __init__(self, name)          # Store food name

```

模拟的订单流程如下。

- a. Lunch类的构造函数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place Order方法应该转向要求Employee对象一个新的Food对象，通过调用Employee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的构造函数。顶层的Lunch类应该也导出一个名为result的方法，它要求顾客打印从Employee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24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. 动物园动物的层次。思考如图31-1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来引入下面子类中不同分类的信息（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。

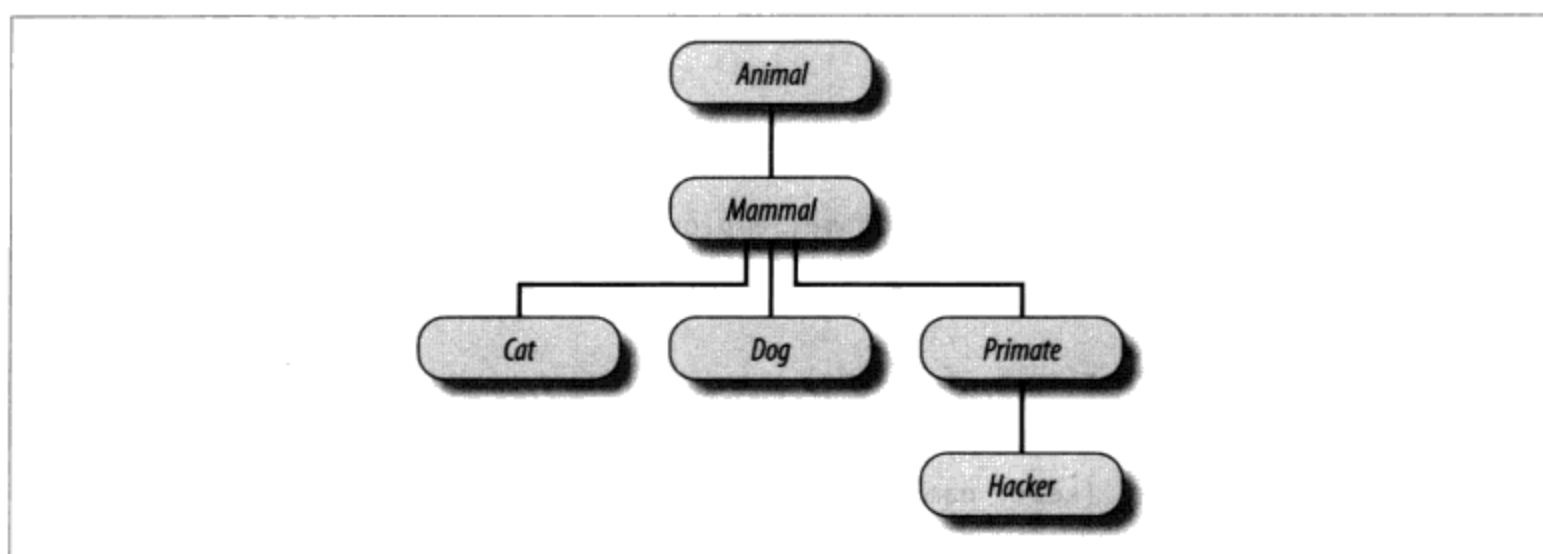


图31-1：动物园的继承层次是由连接在属性继承搜索树上的类构成的。Animal有一个通用的“reply”方法，但是每个类可能都有自己的由“reply”所调用的“speak”方法

```

% 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!
  
```

9. 描绘死鹦鹉。思考如图31-2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
  
```

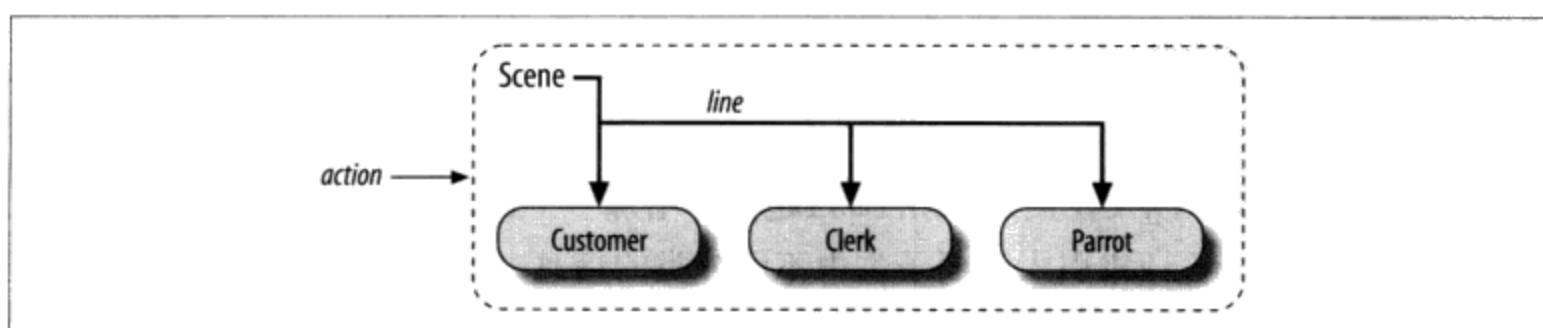


图31-2：这个场景由一个嵌入并指导其他三个类的实例（Customer、Clerk和Parrot）的控制器类(Scene)构成。嵌入的实例的类可以参与到继承层次中来。组合和继承常常是为了代码重复使用而组织类的相当有用的方法

为什么要在意：大师眼中的OOP

当我开始教Python类的时候，无一例外地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）。该技术背后的要点只是没有体现出来。

像这样的书籍，我大费笔墨地介绍了很多内容，就像第25章新的宏观概述以及第27章的渐进教程，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场所。

维护性

类自然而然地促进了代码的分解，这让我们减少了冗余。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，这样也简化了代码的调试、理解以及维护。

多态

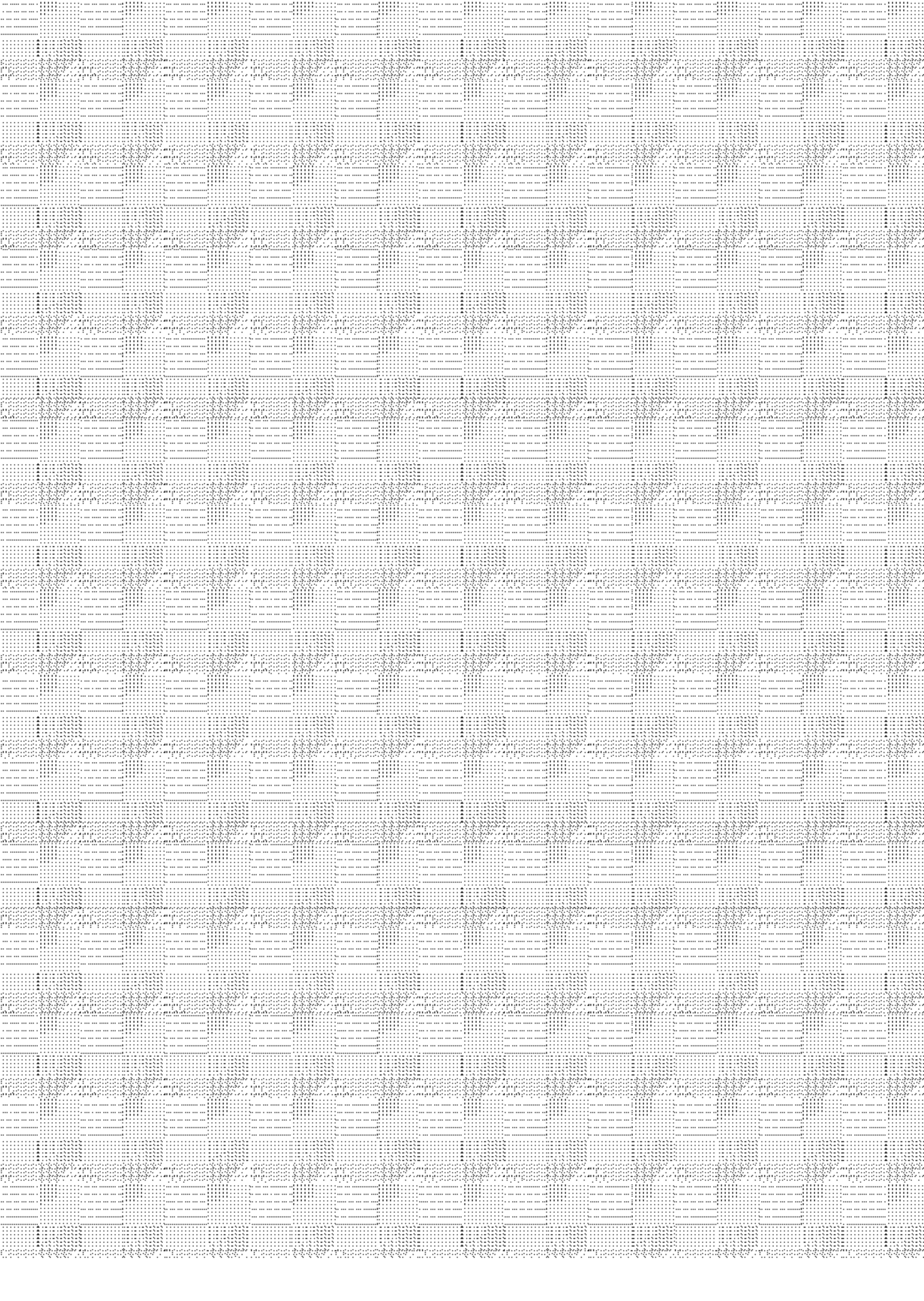
这更像是一个OOP的属性。而不是一条使用它的理由，但是通过广泛地支持代码，多态让代码更灵活和有了广泛的适用性，因此有了更好的可重用性。

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，记住我在这一部分开始说过的：在使用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。

异常和工具



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，异常会根据错误自动地被触发，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和3.0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，因为它已经整合到了语言本身中，成为另一种高级工具。

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比萨机器人的例子。假设认真对待这个想法，并且实际创造出这样的机器。要制作比萨时，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，在这些不常见的情况下，我们无法完成比萨，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，中止开始的所有函数调用而进入异常管理器。在异常处理器中编写代码，来响应在适当时候引发的异常（例如，调用消防部门！）。

异常是一种结构化的“超级goto”。异常处理器（try语句）会留下标识，并可执行一些代码。程序前进到某处代码时，产生异常，因而会使Python立即跳到那个标识，而放弃留下该标识之后所调用的任何激活的函数。这个协议提供了一种固有的方式响应不寻常的事件。再者，因为Python会立即跳到处理器的语句代码更简单——对于可能会发生失败的函数的每次调用，通常就没有必要检查这些函数的状态码。

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，打印出错消息。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）。

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1}。Python中没有“go to”语句（谢天谢地！），但是，异常有时候可以充当类似的角色。

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。

默认异常处理器

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1：反向跟踪是高级话题，并不是Python语言的一部分（即便是第20章介绍的生成器函数和表达式，也都不是真正的反向跟踪——它们直接响应next(G)请求），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。

```
>>> fetcher(x, 4)                                     # Default handler - shell interface
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  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**：就是打印标准出错消息。此时，你也许已经熟悉了标准出错消息。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。

这里的出错消息由Python 3.0打印出来；它随着每个版本略有不同，并且甚至随着每个交互式shell而有所不同。通过交互模式编写代码时，文件名就是“stdin”（标准输入流），表示标准的输入流。当在IDLE GUI的交互shell中工作的时候，文件名就是“pysHELL”，并且会显示出源行。不管哪种方式，当没有文件的时候，文件的行号在这里并没有太大的意义（我们将在本书的本部分中看到更多有趣的出错消息）。

```
>>> fetcher(x, 4)                                     # Default handler - IDLE GUI interface
Traceback (most recent call last):
  File "<pysHELL#6>", line 1, in <module>
    fetcher(x, 4)
  File "<pysHELL#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

在交互模式提示符环境外启动的更为现实的程序中，顶层的默认处理器也会立刻**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出错消息。

捕获异常

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:                                # Catch and recover
...     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。我们没有看见标准出错消息，而程序也将正常运行下去。

引发异常

目前为止，我们已经让Python通过生成错误（这次是故意的）来为我们引发异常，但是，我们的脚本也可以引发异常——也就是说，异常能由Python或程序引发，也能捕捉或忽略。要手动触发异常，直接执行raise语句。用户触发的异常的捕捉方式和Python引发的异常一样。如下的内容不是所编写的最有用的Python代码，但它能说明问题：

```
>>> try:
...     raise IndexError                                # Trigger exception manually
... except IndexError:
...     print('got exception')
...
got exception
```

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，并通过标准出错消息终止该程序。

```
>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
```

正如我们将在下一章中见到的，assert语句也可以用来触发异常——它是一个有条件的raise，主要是在开发过程中用于调试：

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

AssertionError: Nobody expects the Spanish Inquisition!

用户定义的异常

前面小节所介绍的`raise`语句触发Python的内置作用域中定义的一个内置异常。就像你将在本书这一部分的随后章节中看到的那样，也可以定义自己的新的异常，它特定于你的程序。用户定义的异常能够通过类编写，它继承自一个内置的异常类：通常这个类的名称叫做`Exception`。基于类的异常允许脚本建立异常类型、继承行为以及附加状态信息。

```
>>> class Bad(Exception):                                # User-defined exception
...     pass
...
>>> def doomed():
...     raise Bad()                                       # Raise an instance
...
>>> try:
...     doomed()
... except Bad:                                           # Catch class name
...     print('got Bad')
...
got Bad
>>>
```

终止行为

最后，`try`语句可以说“`finally`”，也就是说，它可以包含`finally`代码块。这看上去就像是异常的`except`处理器，但是`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:                                             # Termination actions
...     print('after fetch')
...
'm'
after fetch
>>>
```

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 <module>
  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子句：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子句捕捉到。

我们将在下一章中看到，在使用某些类型的对象的时候，Python 2.6和Python 3.0提供了try/finally的一种替代。with/as运行一个对象的环境管理逻辑，来确保终止行为的发生：

```
>>> with open('lumberjack.txt', 'w') as file:           # Always close file on exit
...     file.write('The larch!\n')
```

尽管这个选项需要寥寥数行代码，它只是在处理某些对象类型的时候才适用，因此，try/finally是一种更加通用的终止结构。另一方面，with/as还运行启动操作并且支持用户定义的环境管理代码。

为什么要在意：错误检查

了解异常是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语言编写稳健的程序，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()
{
    if (doFirstThing() == ERROR)           # C program
        return ERROR;                     # Detect errors everywhere
    if (doNextThing() == ERROR)            # even if not handled here
        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()
    except:
        badEnding()
    else:
        goodEnding()
```

*# This is where we care about results,
so it's the only place we must check*

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，所以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，并避免编写检查错误程序代码。

本章小结

本章是关于异常的主要内容；异常真的是一个简单的工具。

概括来说，Python异常是一种高级控制流设备。它们可能由Python引发，或者由你自己的程序引发。在这两种情况下，它们都可能被忽略（以触发默认的出错消息），或者由try语句捕获（由你的代码处理）。到Python 2.5为止，try语句有两种逻辑形式，可以组合起来——一种处理异常，一种不管是否发生异常都执行最终代码。Python的raise和assert语句根据需要触发异常（都是内置函数，并且都是我们用类定义的新异常）；with/as是一种替代方式，确保对它所支持的对象执行终结操作。

在本书的本部分其他各章中，我们将介绍一些相关语句的细节，介绍出现在一个try下的其他类型的子句，并且讨论基于类的异常对象。下一章开始进一步介绍这里所见到的语句。不过，继续学习之前，先做一做本章的习题。

本章习题

1. 说出异常处理的3个优点。
2. 如果你不想做任何特殊的事情来处理异常，那么异常会发生什么呢？
3. 如何从一个异常恢复你的脚本？
4. 说出在脚本中触发异常的两种方式。
5. 指出两种方式：不管异常是否发生，它们用来指定最终运行的行为。

习题解答

1. 异常处理对于错误处理、终止动作和事件通知有用。它可以简化特殊情况的处理，并且可以用来实现替代的控制流程。一般来讲，异常处理还可以减少程序所需的检测错误代码的数量，因为所有的错误都由处理器来过滤，你可能不需要测试每个操作的输出。
2. 任何未捕获的异常最终都流入默认的异常处理器，Python在程序的最顶端提供了它。这个处理器打印出类似的出错消息，并且退出程序。
3. 如果你不想要默认消息和退出，可以编写try/except语句来捕获并从触发的异常恢复。一旦捕获了一个异常，该异常将终止，并且程序继续。
4. raise和assert语句可以用来触发一个异常，就好像该异常已经由Python自身引发。原则上讲，我们可以通过生成一个程序错误来引发异常，但是，这通常不是一个明确的目标。
5. try/finally语句可以用来确保在一个代码块退出后执行的操作，而不管它是否会引发一个异常。with/as语句也可以用来确保要运行的终止操作，但是，只有当处理的对象类型支持它的时候才可用。

异常编码细节

在前一章中，我们快速地浏览了与异常相关的语句。这里，我们将深入一点介绍——本章针对Python中的异常处理语法给出了更正式一些的介绍。特别是，我们将介绍try、raise、assert和with语句背后的细节。正如我们将看到的，尽管这些语句大多比较简单，但它们提供了强大的工具来处理Python代码中的异常。

注意：此前的一点常规提示是：异常的内容近年来有了一些变化。从Python 2.5起，finally子句可以同样出现在try语句以及except和else语句中（此前，它们不能组合）。此外，从Python 3.0和Python 2.6开始，新的with环境管理器语句成为正式的，并且用户定义的异常现在必须编写为类实例。此外，Python 3.0支持raise语句和except子句的略微修改的语法。我们将在本书这一版本中关注Python 2.6和Python 3.0中的异常状态，但是，由于你仍然很可能在代码中看到最初的技术，因此，一路上我们会指出在此领域中有哪些发展变化。

try/except/else语句

既然了解了基础的知识，让我们来看一些细节。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，因为它们是不同的角色，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，然后是一个或多个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，以下是其在Python 3.0中的一般格式。

```

try:
    <statements>                                # Run this main action first
except <name1>:
    <statements>                                # Run if name1 is raised during try block
except (name2, name3):
    <statements>                                # Run if any of these exceptions occur
except <name4> as <data>:
    <statements>                                # Run if name4 is raised, and get instance raised
except:
    <statements>                                # Run for all (other) exceptions raised
else:
    <statements>                                # Run if no exception was raised during try block

```

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`子句定义`try`代码块内引发的异常的**处理器**，而`else`子句（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`代码块语句执行时的确发生了异常，Python就跳回`try`，执行第一个符合引发异常的`except`子句下面的语句。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`子句，异常就会向上传递到程序中的之前进入的`try`中，或者如果它是第一条这样的语句，就传递到这个进程的顶层（这会使Python终止这个程序并打印默认的出错消息）。
- 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`子句只在`try`代码块执行时不发生异常才会执行。

`except`子句是**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35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33-1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33-1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Python2.4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`所在的同一个`try`语句中了（在本章中，当我们遇到统一的`try`语句的时候，还会更多地讨论排序规则）。

表33-1: `try`语句分句形式

分句形式	说明
<code>except:</code>	捕捉所有（其他）异常类型
<code>except name:</code>	只捕捉特定的异常
<code>except name, value:</code>	捕捉所列的异常和其额外的数据（或实例）
<code>except (name1, name2):</code>	捕捉任何列出的异常
<code>except (name1, name2), value:</code>	捕捉任何列出的异常，并取得其额外数据
<code>else:</code>	如果没有引发异常，就运行
<code>finally:</code>	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。它们提供了对作为异常引发的对象的访问。

表33-1中第一和第四项是新的。

- `except`子句没列出异常名称（`except:`）时，捕捉没在`try`语句内预先列出的**所有**异常。
- `except`子句以括号列出一组异常[`except (e1, e2, e3):`]会捕捉所列出的任何异常。

因为Python会从头到尾检查`except`子句，在某个`try`中寻找是否有相符者，所以括号版本就像是每个异常列在其`except`子句内，但是语句主体只需编写一次而已。以下是多个`except`子句的例子，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子句，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。

```

try:
    action()
except NameError:
...                                     # Handle NameError
except IndexError:
...                                   # Handle IndexError
except:
...                                 # Handle all other exceptions
else:
...                               # Handle the no-exception case

```

空的except子句是一种通用功能：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。

Python 3.0引入了一个替代方案来解决这些问题之一——捕获一个名为Exception的异常几乎与一个空的except具有相同的效果，但是，忽略和系统退出相关的异常：

```

try:
    action()
except Exception:
...                                     # Catch all possible exceptions, except exits

```

这与空的except具有大多相同的便利性，但是，几乎同样具有危险性。我们将在下一章介绍这种形式如何发挥其魔力，在我们学习异常类的时候。

注意： 版本差异提示：Python 3.0要求表33-1中列出的except E as V:处理器子句形式，并且本书中使用该形式，而不是旧的except E, V:形式。后一种形式在Python 2.6中仍然可用（但是不推荐使用）：如果使用它，它将会转换为前者。做出这一改变，是剔除把旧的形式与两种替代的异常搞混淆的错误，这两种替代形式在Python 2.6中相应地编码为except (E1, E2):。由于Python 3.0只支持as形式，不管是否使用圆括号，值都会解释为替代的异常以供捕获。这一修改还改变了作用域规则：使用新的as语法，变量V在except语句块的末尾删除。

try/else分句

Python新手无法一眼看出else子句的用途。不过，如果没有else，是无法知道控制流程（没有设置和检查布尔标志）是否已经通过了try语句，因为没有异常引发或者因为异常发生了且已被处理过。

```
try:
    ...run code...
except IndexError:
    ...handle exception...
# Did we get here because the try failed or not?
```

就像循环内的else子句让退出原因更为明显，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，也几乎能模拟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，并打印标准出错消息）。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)
```

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。

```
% 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: int division or modulo by zero
```

我在Python 3.0下的一个shell窗口中运行它。消息包含了一个堆栈跟踪(“Traceback”)以及所引发的异常的名称和细节。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，所以这种情况下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，所以一般情况下，异常和错误处理及调试的想法紧密结合起来^{注1}。

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出错显示，有助于跟踪问题。通常来说，这个标准出错消息，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3章所介绍的并将在第35章再次概述的调试工具，例如，`pdb`标准库模块。

注1： 出错消息和堆栈跟踪的文字可能随时间不同而略有不同。如果你的出错消息和本书的不同，也别害怕。例如，在Python 3.0的IDLE GUI中执行这个例子时，出错消息正文在文件名中显示完整的目录路径。

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，没有必要再更加明确代码中的错误。

尽管如此，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）。

```
def kaboom(x, y):  
    print(x + y)                                # Trigger TypeError  
  
try:  
    kaboom([0,1,2], "spam")  
except TypeError:  
    print('Hello world!')                      # Catch and recover here  
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。

```
% python kaboom.py  
Hello world!  
resuming here
```

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子句，Python一定会在try语句后执行其语句代码块，无论try代码块执行时是否发生了异常。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
```

利用这个变体，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**异常发生，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。

要注意，在Python 2.4和更早版本中，finally子句无法和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，但是有许多选用的子句（等一下会介绍）。不过，无论选用哪个版本，finally子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：正如我们将在本章稍后看到的，在Python 2.6和Python 3.0中，新的with语句及其环境管理器提供了一种基于对象的方式来针对退出操作做类似的工作。和finally语句不同，这条新的语句还支持进入操作，但是它仅限用于实现了环境管理器协议的对象。

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
```

```
print('not reached')
```

```
# Continue here only if no exception
```

在这段代码中，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。

正如我们在第9章中学习过的，文件对象在垃圾回收时自动关闭；这对于我们不能分配赋值给变量的临时性文件特别有用。然而，当垃圾收集将要发生的时候，并不总是很容易预计到，特别是在较大的程序中。`try`语句会使得文件更显式地、可预料地关闭，并且适用于一个特定的代码块。它确保了文件会在块退出时关闭，而不管是否发生了异常。

这个特定的函数并不总是那么有用（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，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2}。

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（打印标准出错消息并关闭程序）；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。

注意，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，如今在Python 2.6和Python 3.0中，异常应该都是类的实例。

统一try/except/finally语句

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`子句无法与`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。

注2：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，通常是因为连接的C扩展代码中的bug，而这已在Python范围之外了。

不过，在Python 2.5及其以后的版本中（包括本书所用的Python 2.6和Python 3.0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子句。也就是说，我们现在可以编写下列形式的语句：

```
try:                                     # Merged form
    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。

就像往常一样，finally子句并没有终止异常：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。

统一try语句语法

当像这样组合的时候，try语句必须有一个except或一个finally，并且其部分的顺序必须如下所示：

```
try -> except -> else -> finally
```

其中，else和finally是可选的，可能会有0个或多个except，但是，如果出现一个else的话，必须有至少一个except。实际上，该try语句包含两个部分：带有一个可选的else的except，以及（或）finally。

实际上，下面的方式更准确地描述了这一组合的语句语法形式（方括号表示可选，星号表示0个或多个）：

```
try:                                # Format 1
    statements
except [type [as value]]:           # [type [, value]] in Python 2
    statements
[except [type [as value]]:
    statements]*
[else:
    statements]
[finally:
    statements]

try:                                # Format 2
    statements
finally:
    statements
```

由于这些规则，只有至少有一个except的时候，else才能够出现，并且总是可能混合except和finally，而不管是否有一个else。也可能混合finally和else，但只有在一个except也出现的时候（尽管except可能会忽略一个异常名以捕获所有的并运行一条raise语句，稍后介绍该语句，以重新引发当前的异常）。如果违反了这些顺序规则中的任意一条，在你的代码运行之前，Python将会引发一个语法错误异常。

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子句是可能的，也就是在try/finally语句的try代码块嵌套try/except（第35章会更全面地探索这门技术）。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:                                # Nested equivalent to merged form
    try:
        main-action
    except Exception1:
```

```

        handler1
    except Exception2:
        handler2
    ...
    else:
        no-error
finally:
    cleanup

```

此外，`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）。既然一个`else`总是需要一个`except`，嵌套形式甚至运用于前面小节概括的统一语句形式相同的混合约束。

然而，这种嵌套的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（至少是一个四个字符的行）。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。

合并try的例子

以下示范了合并的`try`语句的执行情况。下面的文件`mergedexc.py`编写了四种常见场景，通过`print`语句来说明其意义。

```

sep = '-' * 32 + '\n'
print(sep + 'EXCEPTION RAISED AND CAUGHT')
try:
    x = 'spam'[99]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')

print(sep + 'NO EXCEPTION RAISED')
try:
    x = 'spam'[3]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')

print(sep + 'NO EXCEPTION RAISED, WITH ELSE')
try:
    x = 'spam'[3]
except IndexError:
    print('except run')
else:
    print('else run')

```



```

finally:
    print('finally run')
print('after run')

print(sep + 'EXCEPTION 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，在Python 3.0中会产生下面的输出（实际上，在Python 2.6中的行为和输出是相同的，因为print调用每次打印出单个一项）。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。

```

c:\misc> C:\Python30\python mergedexc.py
-----
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
-----
NO EXCEPTION RAISED
finally run
after run
-----
NO EXCEPTION RAISED, WITH ELSE
else run
finally run
after run
-----
EXCEPTION RAISED BUT NOT CAUGHT
finally run
Traceback (most recent call last):
  File "mergedexc.py", line 36, in <module>
    x = 1 / 0
ZeroDivisionError: int 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

要显式地触发异常，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，后面跟着可选的要引发的类或者类的一个实例：

```

raise <instance>           # Raise instance of class
raise <class>               # Make and raise instance of class

```

```
raise
```

```
# Reraise the most recent exception
```

正如前面所介绍的，在Python 2.6和Python 3.0中异常总是类的实例。因此，这里第一个`raise`形式是最常见的，我们直接提供一个**实例**，要么是在`raise`之前创建的，要么是`raise`语句中自带的。如果我们传递一个**类**，Python调用不带构造函数参数的类，以创建被引发的一个实例；这个格式等同于在类引用后面添加圆括号。最后的形式重新引发最近引发的异常；它通常用于异常处理器中，以传播已经捕获的异常。

为了更清楚，让我们看一些示例。对于内置异常，如下两种形式是对等的，都会引发指定的异常类的一个实例，但是，第一种形式隐式地创建实例：

```
raise IndexError                                # Class (instance created)
raise IndexError()                             # Instance (created in statement)
```

我们也可以提前创建实例——因为`raise`语句接受任何类型的对象引用，如下的两个示例像前两个一样引发了`IndexError`：

```
exc = IndexError()                             # Create instance ahead of time
raise exc

excs = [IndexError, TypeError]
raise excs[0]
```

当引发一个异常的时候，Python把引发的实例与该异常一起发送。如果一个`try`包含了一个名为`except name as X:`子句，变量`X`将会分配给引发中所提供的实例：

```
try:
    ...
except IndexError as X:                        # X assigned the raised instance object
    ...
```

`as`在`try`处理器中是可选的（如果忽略它，该实例直接不会分配给一个名称），但是，包含它将使得处理器能够访问实例中的数据以及异常类中的方法。

这种模式对于我们用类编写的用户定义的异常也同样有效——例如，如下的代码，传递异常类构造函数参数，该参数通过分配的实例在处理器中变得可用：

```
class MyExc(Exception): pass
...
raise MyExc('spam')                            # Exception class with constructor args
...
try:
    ...
except MyExc as X:                             # Instance attributes available in handler
    print(X.args)
```

由于这涉及下一章的话题，我们将在下一章中详细介绍。

不管你是否指定异常，异常总是通过实例对象来识别，并且大多数时候在任意给定的时刻激活。一旦异常在程序中某处由一条except子句捕获，它就死掉了（例如，不会传递到另一个try），除非由另一个raise语句或错误重新引发它。

利用raise传递异常

raise语句不包括异常名称或额外数据值时，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')           # Exceptions remember arguments
... except IndexError:
...     print('propagating')
...     raise                             # Reraise most recent exception
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
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，它会停止程序，打印标准出错消息）。注意我们传递给异常类的参数是如何出现在出错消息中的，我们将在下一章中了解为什么会这样。

Python 3.0异常链：raise from

Python 3.0（而不是Python 2.6）也允许raise语句拥有一个可选的from子句：

```
raise exception from otherexception
```

当使用from的时候，第二个表达式指定了另一个异常类或实例，它会附加到引发异常的__cause__属性。如果引发的异常没有捕获，Python把异常也作为标准出错消息的一部分打印出来：

```
>>> try:
...     1 / 0
... except Exception as E:
...     raise TypeError('Bad!') from E
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero
```

上面的异常是如下异常的直接原因：

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
```

`TypeError: Bad!`

当在一个异常处理器内部引发一个异常的时候，隐式地遵从类似的过程：前一个异常附加到新的异常的`__context__`属性，并且如果该异常未捕获的话，再次显示在标准出错消息中。这是一个高级的并且多少还有些含糊的扩展，因此，请参阅Python的手册以了解详细内容。

注意：版本差异提示：Python 3.0不再支持`raise Exc, Args`形式，而该形式在Python 2.6中仍然可用。在Python 3.0中，使用本书中介绍的`raise Exc(Args)`示例创建调用形式。Python 2.6中等价的逗号形式是遗留的语法，为了与现在已经废弃的基于字符串的异常类型兼容，并且它在Python 3.0中也是废弃的。如果使用的话，它会转换为Python 3.0的调用形式。正如在前面的版本中一样，一个`raise Exc`形式总是允许的，它在两个版本中都会转换为`raiseExc()`形式，调用无参数的类构造函数。

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，`assert`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>)
```

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，在此情况下数据项将作为出错消息的一部分显示。

`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`标志，否则自动设为1（真值）。使用类似`python -O main.py`的一个命令行来在优化模式中运行，并且关闭`assert`。

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，其出错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 <module>
  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，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3}。另一个`assert`常见用法例子，可以参考第28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。

with/as环境管理器

Python 2.6和Python 3.0引入了一种新的异常相关的语句：`with`及其可选的`as`子句。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。这一功能在Python 2.5中也可选地使用，用一条如下形式的`import`来激活：

```
from __future__ import with_statement
```

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。

注3：至少，多数情况下是这样。就像本书前面建议的那样，如果函数必须运行长时间或无法恢复的动作，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。

基本使用

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`子句存在时，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:\misc\data') as myfile:  
    for line in myfile:  
        print(line)  
        ...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循环引发了异常也是如此。

尽管文件对象在垃圾回收时自动关闭，然而，并不总是能够很容易地知道会何时发生。with语句的这种用法作为一种替代，允许我们确定在一个特定代码块执行完毕后会发生关闭。正如前面所看到的，我们可以使用更通用而明确的try/finally语句来实现类似的效果，但是，这需要4行管理代码而不是1行：

```
myfile = open(r'C:\misc\data')  
try:  
    for line in myfile:  
        print(line)  
        ...more code here...  
finally:  
    myfile.close()
```

我们不会在本书讨论Python的多线程模块（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*），但那些模块所定义的锁和条件变量同步对象也可以和with语句一起使用，因为它们支持环境管理协议。

```
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自动获得，并且一旦代码块完成就释放，而不管异常输出是什么。

decimal模块（参考第5章有关小数类型的内容）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）。

```
with decimal.localcontext() as ctx:
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')
```

在这条语句运行后，当前线程的环境管理器状态自动恢复到语句开始之前的状态。要使用try/finally做到同样的事情，我们需要提前保存环境并手动恢复它。

环境管理协议

尽管一些内置类型带有环境管理器，我们还可以自己编写一个。要实现环境管理器，使用特殊的方法来接入with语句，该方法属于运算符重载的范畴。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子句存在，其返回值会赋值给As子句中的变量，否则，直接丢弃。
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语句之外。
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。

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。

```
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False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。如下是实际在Python 3.0中运行的脚本（它也能够运行在Python 2.6下运行，但是，会打印出一些额外的元组圆括号）：

```
% python withas.py
starting with block
running test 1
reached
exited normally

starting with block
running test 2
raise an exception! <class 'TypeError'>
Traceback (most recent call last):
  File "withas.py", line 20, 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

他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。

注意: 在即将发布的Python 3.1版中, `with`语句也可以使用新的逗号语法指定多个(有时候叫做“嵌套的”)环境管理器。例如, 在下面的例子中, 当语句块退出的时候, 两个文件的退出操作都会自动运行, 而不管异常输出什么:

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        if 'some key' in line:
            fout.write(line)
```

可以列出任意数目的环境管理器项, 并且多个项目和嵌套的`with`语句一样地工作。通常, Python 3.1 (及其以后版本)的代码:

```
with A() as a, B() as b:
    ...statements...
```

等同于如下的代码, 它们在Python 3.1、Python 3.0和Python 2.6下都有效:

```
with A() as a:
    with B() as b:
        ...statements...
```

参见Python 3.1的版本提示了解详细情况。

本章小结

在这一章, 我们详细地介绍了异常的处理, 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, 事实上也的确是如此, 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。

本章习题

1. `try`语句有什么用途?
2. `try`语句的两个常见变体是什么?
3. `raise`语句有什么用途?

4. `assert`语句是做什么用的？和其他哪些语句相像？
5. `with/as`语句是做什么用的？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`语句两个常见变体就是`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`有更丰富的基于对象的协议，可以定义进入和离开的动作。

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，在Python 2.6和Python 3.0中，内置异常和用户定义的异常都可以通过**类实例对象**来表示。尽管这意味着，你必须使用面向对象编程来在程序中定义新的异常，类和OOP通常提供了几种优点。

基于类的异常有如下特点。

- **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。
- **它们附加了状态信息**。异常类提供了存储在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。
- **它们支持继承**。基于类的异常允许参与继承层次，从而可以获得并定制共同的行为。例如，继承的显示方法可提供通用的出错消息的外观。

因为有这些差异，所以基于类的异常支持了程序的演进和较大系统，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。

在Python 3.0中，用户定义的异常继承自内置异常超类。正如我们将在这里介绍的，由于这些超类为打印和状态保持提供了有用的默认值，所以编写用户定义的异常的任务也涉及理解这些内置超类的作用。

注意：版本差异提示：Python 2.6和Python 3.0都要求异常通过类来定义。此外，Python 3.0要求异常类派生自BaseException内置异常超类，而不管是直接还是间接。正如我们将看到的，大多数程序都继承自这个类的Exception子类，以支持针对常规异常类型的全捕获处理器——在一个处理器中指定它将会捕获大多数程序应该捕获的所有内容。Python 2.6也允许独立的标准类来充当异常，但是，它要求新式类派生自内置异常类，这与Python 3.0相同。

异常：回到未来

曾经（在Python 2.6和Python 3.0之前）可以以两种不同的方式来定义异常。这通常使得try语句、raise语句和Python复杂化。如今，只有一种方式来定义异常。这是件好事情：它从语言中删除了为了实现向后兼容而保留的大量冗余内容。由于旧的方式有助于说明为什么异常成为今天的样子，并且，由于真的不可能完全擦除上百万人使用了近二十年的教程的历史，所以让我们先来简单看看异常的过去。

字符串异常很简单

在Python 2.6和Python 3.0之前，可以使用类实例和字符串对象来定义异常。基于字符串的异常在Python 2.5中就发布了废弃警告，并且在Python 2.6和Python 3.0中删除了，因此，今天你应该使用基于类的异常，就像本书中所介绍的那样。如果你使用遗留代码，可能还会遇到字符串异常。它们也可能会出现在几年前（这在Python的纪年中应该算作是不朽了）编写的教程和Web资料中。

字符串异常很容易使用——任何字符串都可以做到，它们根据对象标识来匹配，而不是根据值（也就是说，使用is，而不是==）：

```
C:\misc> C:\Python25\python
>>> myexc = "My exception string"                # Were we ever this young?
>>> try:
...     raise myexc
... except myexc:
...     print('caught')
...
caught
```

对于较大的程序和代码维护来说，这种形式并不像类那么好，因此，删除了它。尽管你如今不能使用字符串异常，但它们实际上提供了一种自然的工具来引入基于类的异常模式。

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子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测试来匹配except子句的。
- 类异常是由超类关系进行匹配的：只要except子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子句。

也就是说，当try语句的except子句列出一个超类时，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，类异常支持异常层次的架构：超类变成分类的名称，而子类变成这个分类中特定种类的异常。except子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。

字符串异常没有这样的概念：因为它们都通过简单对象标识来匹配，所以它们没有直接的方式来把异常组织到更为灵活的领域或分组。直接的结果是，异常处理器以一种难以做出修改的方式与异常集合匹配。

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）。由于它们提供类和OOP一般性的所有优点，比起现在废弃了的基于字符串的异常来说，它们提供一种更为强大的替代方案，而只需要一点点额外的代码。

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(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
    X = General()
    raise X
    # Raise superclass instance

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:\python30> python classexc.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>

```

这段代码相当直接，但是，这里有一些实现细节需要注意：

Exception超类

用来构建异常分类树的类拥有很少的需求——实际上，在这个例子中，它们主要是空的，其主体不做任何事情而直接通过。注意，这里顶层的类是如何从内置的Exception类继承的。这在Python 3.0中是必需的；Python 2.6也允许独立的经典类充当异常，但是，它要求新式类派生自内置异常类，这和在Python 3.0中一样。由于Exception提供了一些有用的行为，我们随后才会遇到这些行为，因此，在这里不能使用它们；但是，在任何Python版本中，从它那里继承是个好主意。

引发实例

在这段代码中，我们调用类来创建raise语句的实例。在类异常模式中，我们总是引发和捕获一个类实例对象。如果我们在一个raise中列出了类名而没有圆括号，那么Python调用该类而没有构造函数参数为我们产生一个实例。异常实例可以在该raise之前创建，就像这里所做的一样，或者在raise语句自身中创建。

捕获分类

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，因为它们是General的子类）。

异常细节

我们会在下一章再谈这里所用到的异常处理器sys.exc_info调用：这是一种抓取最近发生异常的常用方式。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。这里的except子句捕获了一个分类

中所有的类，在这样的一条通用的except子句中，`sys.exc_info`是决定到底发生了什么的一种方式。在这一特别的情况下，它等价于获取实例的`__class__`属性。

正如我们将在下一章中看到的，`sys.exc_info`方法通常也与捕获所有内容的空的except子句一起使用。最后一点值得进一步说明。当捕获了一个异常，我们可以确定该实例是except中列出的类的一个实例，或者是其更具体的子类中的一个。因此，实例的`__class__`属性也给出了异常类型。例如，如下的变体和前面的例子起着同样的作用：

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General as X:
        print('caught:', X.__class__)
```

X is the raised instance
Same as sys.exc_info()[0]

由于`__class__`可以像这样使用来决定引发的异常的具体类型，因此`sys.exc_info`对于空的except子句更有用，否则的话，没有一种方式来访问实例及其类。此外，更实用的程序通常根本不必关注引发了哪个具体的异常——通过一般调用实例的方法，我们自动把修改后的行为分派给引发的异常。下一章更多地介绍这一点以及`sys.exc_info`；如果你已经忘记了实例中的`__class__`的含义，请参见第28章以及第六部分的大部分内容。

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子句的括号内列出字符串异常名称的清单，从而达到相同效果。

```
try:
    func()
except (General, Specific1, Specific2):
    ...
```

Catch any of these

这种方法对于已经废弃的字符串异常模式也有效。然而，就大型或多层次的异常而言，在一个except子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

class Divzero(Exception): pass
class Oflow(Exception): pass

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

import mathlib

try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow):
    ...handle 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

class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

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):
    ...handle 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`子句来捕捉所有可能的异常。

```
# client.py
```

```

try:
    mathlib.func(...)
except:
    ...handle and recover...
# Catch everything here

```

但是，这种权宜之计可能捕捉到不想要捕捉的异常：像内存耗尽、键盘中断（Ctrl-C）、系统退出甚至自己的try块中的代码录入错误，都将触发异常，然而，你想让这些异常通过，而不是非得捕捉到并且误以为是库错误。

实际上，在这种情况下，用户想要捕获并从中恢复的**唯一的**具体异常，就是库定义并记录的要引发的异常；如果在库调用中发生了任何其他的异常，很可能是库中的一个真正的bug（并且可能是该联系厂商的时候了）。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，该怎么做呢？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。

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子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。

```
# 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，**没有改变**。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，基于类的异常层级可支持状态保留和继承，以一种对于大程序来说理想的方式。要理解这些用法，我们首先需要看看用户定义的异常类与它们所继承自的内置异常是如何相关的。

内置Exception类

前一节的例子并没有带来什么新的东西。Python自身能够引发的所有的内置异常，都是预定义的类对象。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。

在Python 3.0中，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，可以作为`builtin`模块中的内置名称使用（在Python 2.6中，它们位于`__builtin__`，并且也是标准库模块`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。

BaseException

异常的顶级根类。这个类不能当作是由用户定义的类直接继承的（使用`Exception`）。它提供了子类所继承的默认的打印和状态保持行为。如果在这个类的一个实例上调用`str`内置函数（例如，通过`print`），该类返回创建实例的时候所传递的构造函数参数的显示字符串（或者如果没有参数的话，是一个空字符串）。此外，除非子类替代了这个类的构造函数，在实例构造时候传递给这个类的所有参数都将作为一个元组存储于其`args`属性中。

Exception

与应用相关的异常的顶层根超类。这是`BaseException`的一个直接子类，并且是所有其他内置异常的超类，除了系统退出事件类之外（`SystemExit`、`KeyboardInterrupt`和`GeneratorExit`）。几乎所有的用户定义的类都应该继承自这个类，而不是`BaseException`。当遵从这一惯例的时候，在一条`try`语句的处理器中指明`Exception`，会确保你的程序将捕获除了系统退出事件之外的所有异常，通常该事件是允许通过的。实际上，`Exception`变成了`try`语句中的一个全捕获，并且比一条空的`except`更精确。

ArithmeticError

所有数值错误的超类（并且是Exception的一个子类）。

OverflowError

识别特定的数值错误的子类。

其他，等等——你可以在Python Pocket Reference或Python库手册这样的帮助文本中进一步阅读关于这个结构的内容。注意，异常类树在Python 3.0和Python 2.6中略有不同。还要注意，只有在Python 2.6中，我们可以在exception模块（这个模块在Python 3.0中删除了）的帮助文本中看到类树。参考第4章和第15章有关help的内容：

```
>>> import exceptions
>>> help(exceptions)
...lots of text omitted...
```

内置异常分类

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。在一条try中列出ArithmeticError，将会捕获所引发的任何类型的数值错误；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Exception是Python中所有应用程序级别的异常的超类，通常可以使用它作为一个全捕获，其效果与一条空的except很类似，但是它允许系统退出异常而像平常那样通过：

```
try:
    action()
except Exception:
    ...handle all application exceptions...
else:
    ...handle no-exception case...
```

这在Python 2.6中通常不会有效，因为编写为经典类的独立的用户定义异常，不要求必须是Exception根类的子类。这一技术在Python 3.0中不会更为可靠，因为它要求所有的类都派生自内置异常。即便在Python 3.0中，这种方案会像空的except一样遭遇大多数相同的潜在陷阱，就像前一章所介绍的那样——它可能拦截用于其他地方的异常，并且可能掩盖了真正的编程错误。既然这是如此常见的一个问题，我们将在下一章的“陷阱”部分回顾它。

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。

默认打印和状态

内置异常还提供了默认打印显示和状态保持，它往往和用户定义的类所需的逻辑一样的多。除非你重新定义了类继承自它们的构造函数，传递给这些类的任何构造函数参数都会保存在实例的`args`元组属性中，并且当打印该实例的时候自动显示（如果没有传递构造函数参数，则使用一个空的元组和显示字符串）。这说明了为什么传递给内置异常类的参数会出现在出错消息中，当打印实例的时候，附加给实例的任何构造函数参数就会显示：

```
>>> raise IndexError                                # Same as IndexError(): no arguments
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

>>> raise IndexError('spam')                      # Constructor argument attached, printed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: spam

>>> I = IndexError('spam')                          # Available in object attribute
>>> I.args
('spam',)
```

对于用户定义的异常也是如此，因为它们继承了其内置超类中存在的构造函数和显示方法：

```
>>> class E(Exception): pass
...
>>> try:
...     raise E('spam')
... except E as X:
...     print(X, X.args)                            # Displays and saves constructor arguments
...
spam ('spam',)

>>> try:
...     raise E('spam', 'eggs', 'ham')
... except E as X:
...     print(X, X.args)
...
('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')
```

注意，该异常实例对象并非字符串自身，但是，当打印的时候，使用我们在第29章介绍的`__str__`运算符重载协议来提供显示字符串；要连接真正的字符串，执行手动转换：`str(X) + "string"`。

尽管这种自动状态和现实支持本身是有用的，但对于特定的显示和状态保持需求，你总

是可以重新定义Exception子类中的__str__和__init__这样的继承方法，下一小节介绍如何做到这一点。

定制打印显示

正如我们在前一小节中看到的，默认情况下，捕获并打印基于类的异常的实例的时候，它们会显示我们传递给类构造函数的任何内容：

```
>>> class MyBad(Exception): pass
...
>>> try:
...     raise MyBad('Sorry--my mistake!')
... except MyBad as X:
...     print(X)
...
Sorry--my mistake!
```

当没有捕获异常的时候，如果异常作为一条出错消息的一部分显示，这个继承的默认显示模式也会使用：

```
>>> raise MyBad('Sorry--my mistake!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyBad: Sorry--my mistake!
```

对于很多用途来说，这已经足够了。要提供一个更加定制的数据显示，我们可以在类中定义两个字符串表示重载方法中的一个（__repr__或__str__），来返回想要为异常显示的字符串。如果异常被捕获并打印，或者异常到达默认的处理程序，方法返回的字符串都将显示：

```
>>> class MyBad(Exception):
...     def __str__(self):
...         return 'Always look on the bright side of life...'
...
>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Always look on the bright side of life...

>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyBad: Always look on the bright side of life...
```

这里要注意的细微一点是，我们通常为此目的必须重新定义__str__，因为内置的超类已经有一个__str__方法，并且在大多数环境下（包括打印），__str__优先于__repr__。

如果你定义了一个`__repr__`，打印将会很乐意地调用超类的`__str__`。参见第29章了解关于这一特殊方法的更多细节。

对于未捕获的异常，方法返回的内容都包含在出错消息中，并且打印异常的时候显式化。这里，方法返回一个硬编码的字符串来说明，但是，它也可以执行任意的文本处理，可能附加到实例对象的状态信息。下一小节介绍状态信息选项。

定制数据和行为

除了支持灵活的层级，异常类还提供了把额外状态信息存储为实例属性的功能。正如我们前面所见到的，内置异常超类提供了一个默认的构造函数，它自动把构造函数参数存储到一个名为`args`的实例元组属性中。尽管默认的构造函数对于很多情况都适用，但为了满足更多的定制需求，我们可以提供一个自己的构造函数。此外，类可以定义在处理器中使用的方法，来提供预先编码的异常处理逻辑。

提供异常细节

当引发一个异常的时候，可能会跨越任意的文件界限——触发异常的`raise`语句和捕获异常的`try`语句可能位于完全不同的模块文件中。在一个全局变量中存储额外的细节通常是不可行的，因为`try`语句可能不知道全局变量位于哪个文件中。在异常自身中传递额外的状态信息，这允许`try`语句更可靠地访问它。

使用类，这几乎是自动化的。正如我们已经看到的，当引发一个异常的时候，Python随着异常传递类实例对象。在`try`语句中的代码，可以通过在一个`except`处理器中的`as`关键字之后列出一个额外的变量，来访问引发的异常。这提供了一个自然的钩子，以用来为处理器提供数据和行为。

例如，解析数据文件的一个程序可能通过引发一个异常实例来表示一个格式化错误，而该实例用关于错误的额外细节来填充：

```
>>> class FormatError(Exception):
...     def __init__(self, line, file):
...         self.line = line
...         self.file = file
...
>>> def parser():
...     raise FormatError(42, file='spam.txt')           # When error found
...
>>> try:
...     parser()
... except FormatError as X:
...     print('Error at', X.file, X.line)
...
```

Error at spam.txt 42

在这这里的except子句中，对引发异常的时候所产生的实例的一个引用分配给了X变量。^{注2}这使得能够通过定制的构造函数来访问附加给该实例的属性。尽管我们可能依赖于内置超类的默认状态保持，它与我们的应用程序几乎不相关：

```
>>> class FormatError(Exception): pass                # Inherited constructor
...
>>> def parser():
...     raise FormatError(42, 'spam.txt')             # No keywords allowed!
...
>>> try:
...     parser()
... except FormatError as X:
...     print('Error at:', X.args[0], X.args[1])      # Not specific to this app
...
Error at: 42 spam.txt
```

提供异常方法

除了支持特定于应用程序的状态信息，定制构造函数还更好地支持用于异常对象的额外信息。也就是说，异常类也可以定义在处理器中调用的方法。例如，如下的代码添加了一个方法，它使用异常状态信息把错误记录到一个文件中：

```
class FormatError(Exception):
    logfile = 'formaterror.txt'
    def __init__(self, line, file):
        self.line = line
        self.file = file

def logerror(self):
    log = open(self.logfile, 'a')
    print('Error at', self.file, self.line, file=log)

def parser():
    raise FormatError(40, 'spam.txt')

try:
    parser()
except FormatError as exc:
    exc.logerror()
```

运行的时候，这段脚本把出错消息写入一个文件中，以响应异常处理器中的方法调用：

注2：正如前面所提到的，引发的实例对象通常作为sys.exc_info()调用的结果元组中的第二项是可用的——sys.exc_info()是返回有关最新引发的异常信息的一个工具。如果你没有在except子句中列出一个异常名称，但是仍然需要访问所发生的异常或者访问其附加的任何状态信息或方法，就必须使用这个接口。关于sys.exc_info的更多介绍在下一章给出。

```
C:\misc> C:\Python30\python parse.py
C:\misc> type formaterror.txt
Error at spam.txt 40
```

在这样的一个类中，方法（如`logerror`）也可能继承自超类，并且实例属性（例如`line`和`file`）提供了一个地方来保存状态信息，状态信息提供了额外环境用于随后的方法调用。此外，异常类可以自由地定制和扩展继承的行为。换句话说，由于它们是用类定义的，所以我们在本书第六部分中所学习的所有OOP的好处，对于Python中的异常来说都是可用的。

本章小结

在这一章中，我们介绍的是编写用户定义的异常。正如我们所学到的，在Python 2.6和Python 3.0中，异常可以实现为类实例对象（在更早的版本中，有一个基于类的异常模式替代方案，但是现在已经废弃了）。异常类支持异常的层次概念，该层级更容易维护，可以让数据和行为作为实例的属性以及方法附加在异常上，而且可以让异常继承超类的数据和行为。

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。

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。

本章习题

1. 在Python 3.0中，对于用户定义异常的两个新限制是什么？
2. 基于类的异常是怎样与处理器匹配的？
3. 说出把环境信息附加到异常对象上的两种方法。
4. 说出为异常对象指定出错消息的两种方法。
5. 如今为何不再使用基于字符串的异常？

习题解答

1. 在Python 3.0中，异常必须由类定义（也就是说，引发并捕获一个类实例对象）。

此外，异常类必须派生自内置类`BaseException`（大多数程序继承自其`Exception`子类，以支持常规类型的异常的全捕获）。

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. 我们可以通过在引发的实例对象中填充实例属性，来把环境信息附加到基于类的异常，通常是在一个定制类构造函数中做到这点。对于较简单的需求，内置异常超类提供了一个构造函数，它将其参数存储到实例上（在`args`属性中）。在异常处理器中，我们列出要分配给引发的实例的一个变量，然后，使用这个名称来访问附加的状态信息并调用类中定义的任何方法。
4. 基于类的异常中的出错消息可以用一个定制的`__str__`运算符重载方法来指定。对于较简单的需求，内置的异常超类自动显示你传递给类构造函数的任何内容。当显式地打印一个异常对象或将其作为一条出错消息的一部分的时候，像打印和`str`这样的操作会自动获取异常对象的显示字符串。
5. 因为Guido这么说：它们在Python 2.6和Python 3.0中会删除。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。

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。由于本章是本书基础话题部分的收尾，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，这代表了什么意思呢？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35-1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

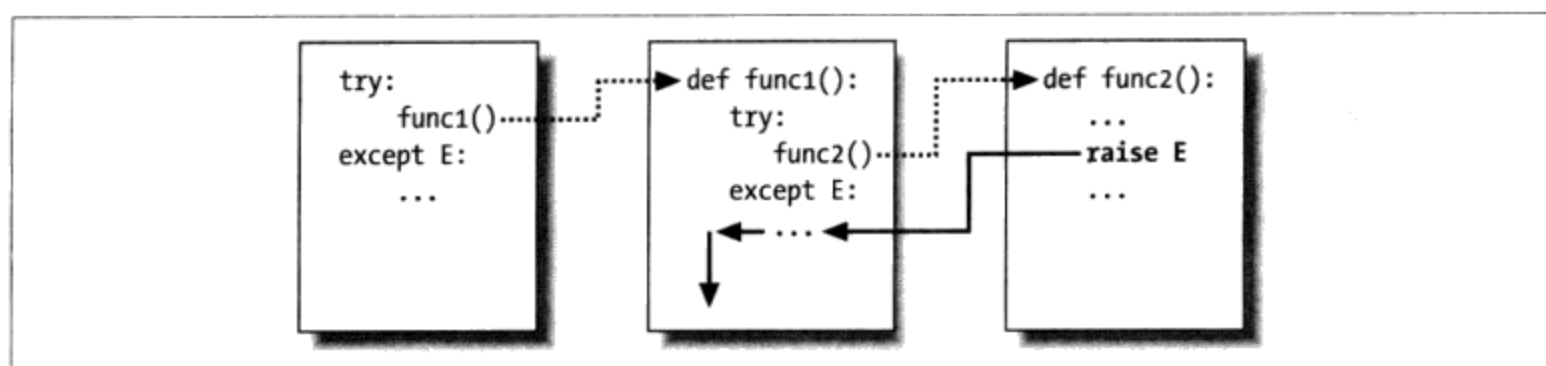


图35-1：嵌套的try/except语句：当异常引发时（由你或由Python引起），控制权会跳回具有相符的except子句、最近进入的try语句，而程序会在try语句后继续执行下去。except子句会拦截并停止异常，这里就是你处理异常并从中恢复的地方

句；只有第一个try有机会对它进行处理。如图35-1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。

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（标准出错消息打印器）。如图35-2所示，finally子句不会终止异常，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都在活动，它们就都会运行，除非有个try/except在这个过程中捕捉某处该异常。

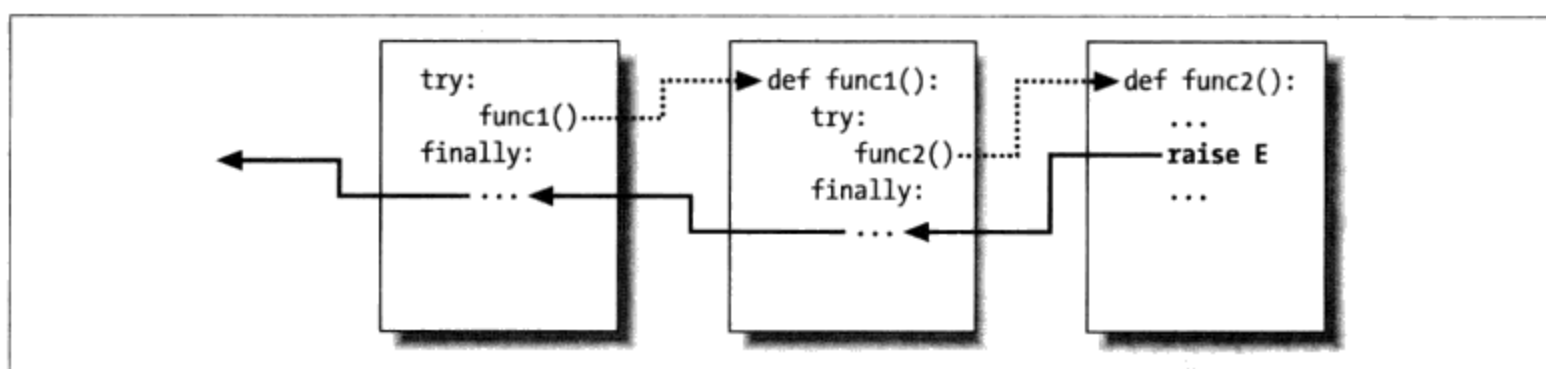


图35-2：嵌套的try/finally：当异常在这里引发时，控制权会回到最近进入的try去执行其finally语句，异常会持续传播到所有激活状态下try语句的finally，直到最终抵达默认顶层处理器，在那里打印出错消息。Finally子句会拦截（但不会停止）异常：只是定义了离开前要执行的动作而已

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except子句，传递就会停止，而通过finally子句时就不会。

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

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——等一下会看与其等效的情况。

例子：语法嵌套化

第33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35-1和图35-2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部分。

```
>>> try:
...     try:
...         raise IndexError
...     finally:
...         print('spam')
... finally:
...     print('SPAM')
...
spam
SPAM
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError
```

参考图35-2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, sep='')
    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。这需要一点时间去理解，但是其效果很像在单个try语句内结合except和finally（在Python 2.5及其以后的版本中）。

```
% python except-finally.py
<function raise1 at 0x026ECA98>
caught IndexError
finally run

<function noraise at 0x026ECA50>
finally run

<function raise2 at 0x026ECBB8>
finally run

Traceback (most recent call last):
  File "except-finally.py", line 9, in <module>
    func()
```

```
File "except-finally.py", line 3, in raise2
    def raise2(): raise SyntaxError
SyntaxError: None
```

就像我们在第33章见到过的，Python 2.5时，`except`和`finally`子句可以混合在相同`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`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`（这一功能在Python 2.6中叫做`raw_input`）。

和文件方法不同的是，这个函数并不返回空字符串：`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，`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 True:
    try:
        line = 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——例如，调用`sys.exit()`并在键盘上按下Ctrl-C，会分别引发`SystemExit`和`KeyboardInterrupt`。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

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:
    ...success...           # Exception if item was found
else:
    ...failure...          # else returned: not found
```

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信号，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，所以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。

关闭文件和服务器连接

我们在第33章遇到了这一类的例子。作为概括，异常处理工具通常也用来确保系统资源终结，不管在处理过程中是否发生了错误。

例如，一些服务器需要关闭连接，从而终止一个会话。与之类似，输出文件可能需要关闭把缓冲区写入磁盘的调用，并且，如果没有关闭输入文件的话，它可能会占用文件描述符；尽管在垃圾收集的时候，如果文件对象还打开它会自动关闭，但这有时候很难确保。

确保一个特殊代码块的终止操作的更通用和显式的方式是try/finally语句：

```
myfile = open(r'C:\misc\script', 'w')
try:
    ...process myfile...
finally:
    myfile.close()
```

正如我们在第33章所见到的，Python 2.6和Python 3.0中的一些对象使得这较为容易：提供由with/as语句运行的**环境管理器**，从而为我们自动终止或关闭对象：

```
with open(r'C:\misc\script', 'w') as myfile:
    ...process myfile...
```

那么，哪种选择更好呢？通常，这取决于你的程序。与try/finally相比，环境管理器**更为隐式**，它与Python通常的设计哲学背道而驰。环境管理器肯定也不太常见，它们通常只对选定的对象可用，并且编写用户定义的环境管理器来处理通用的终止需求，比编写一个try/finally更为复杂。

另一方面，使用已有的环境管理器，比使用try/finally需要更少的代码，如前面的例子所示。此外，环境管理器协议除了支持退出动作，还支持**进入动作**。尽管try/finally可能是更加广为应用的技术，环境管理器可能更适合于可以使用它们的地方，或者可以允许它们的额外复杂性的地方。

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子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类和引发的实例对象（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，在错误发生后，仍保持程序处于激活状态：这样可以执行其他测试，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

可以在测试驱动程序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('FAILED', testName(), sys.exc_info()[0:2], file=log)
        else:
            print('PASSED', testName(), file=log)
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，如果你想测试失败后让测试进程继续下去，就需要在`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输出的工具。

关于`sys.exc_info`

`sys.exc_info`结果在上两个小节中用到，通常允许一个异常处理器获取对最近引发的异常的访问。当使用空的`except`子句来盲目地捕获每个异常以确定引发了什么的时候，这种方式特别有用：

```
try:
    ...
except:
    # sys.exc_info()[0:2] are the exception class and instance
```

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。
- *value*是引发的异常类实例。
- *traceback*是一个traceback对象，代表异常最初发生时所调用的堆栈（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，从而可以手动产生出错消息的工具）。

正如我们在前一章中看到的，当捕获异常分类超类的时候，`sys.exc_info`对于确定特定的异常类型很有用。正如我们所看到的，由于在这种情况下，我们也可以通过`as`子句所获取的实例的`__class__`属性来获得异常类型，`sys.exc_info`如今主要由空的except使用：

```
try:
    ...
except General as instance:
    # instance.__class__ is the exception class
```

也就是说，使用实例对象的接口和多态，往往是比较测试异常类型更好的方法——可以为每个类定义异常方法并通用地运行：

```
try:
    ...
except General as instance:
    # instance.method() does the right thing for this instance
```

通常，在Python中太具体可能会限制代码的灵活性。像这里的最后一个例子的多态方法，通常更好地支持未来的改进。

注意：版本差异提示：在Python 2.6中，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单个的全局异常。这两个名称在Python 3.0中已经删除了。新的更倾向于使用的`sys.exc_info()`调用在Python 2.6和Python 3.0中都可以使用，它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（这个主题超出了本书范围）。参考其他资料获得更多的细节。

与异常有关的技巧

我在本章中集中把设计提示和陷阱结合起来介绍，因为事实证明大多数常见的陷阱主要源自于设计问题。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`子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。

- 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）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出错消息（如果不是崩溃的话），而且这通常就是所期望的最好结果。
- 应该在`try/finally`中实现终止动作，从而保证它们的执行，除非环境管理器作为一个`with/as`选项可用。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`子句会捕捉`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、键盘中断以及系统退出，等等，都会在Python中引发异常。这类异常通常是不应该拦截的。

例如，当控制权到达顶层文件末尾时，脚本正常是退出。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。使用上一章介绍的内置异常类，在这种特定情况下会有帮助，因为`Exception`超类不是`SystemExit`的一个超类：

```

try:
    bye()
except Exception:                             # Won't catch exits, but _will_ catch many others
    ...

```

在其他情况下，这种方案并不比空的`except`子句好——因为`Exception`是所有内置异常（除了系统退出事件以外）之上的一个超类，它仍然有潜力捕获程序中其他地方的异常。

最糟的情况是，空`except`和捕获`Exception`类也会捕捉一般程序设计错误，但这类错误

注1： 一个相关的调用`os._exit`也会结束程序，但是通过立即终止：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藉以获得更多细节。

多数时候都应让其通过的。事实上，这两种技术都会有效关闭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，但处理器会默默地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！

经验法则是，尽量让处理器具体化：空except子句很方便，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。

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异常的列表中。

我们在前一章中见到了这种现象。例如，下列处理器是把MyExcept1和MyExcept2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MyExcept3，就会视为错误并对其进行处理，除非更新异常列表。

```
try:
    ...
except (MyExcept1, MyExcept2):
    ...                               # Breaks if you add a MyExcept3
else:
    ...                               # Non-errors
    ...                               # Assumed to be an error
```

值得庆幸的是，小心使用第33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:
    ...
else:
    ...

```

OK if I add a myerror3 subclass
Non-errors

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。

核心语言总结

这里要总结核心Python程序设计语言。祝贺你！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。

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35-1总结Python程序员可用的内置或现有的功能来源，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。但既然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35-1：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、open
异常	IndexError、KeyError
模块	os、tkinter、pickle、re
属性	__dict__、__name__、__class__
外部工具	NumPy、SWIG、Jython、IronPython、Django等

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摘要。

PyDoc和文档字符串

PyDoc的help函数和HTML界面在第15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和Pylint

因为Python是的一门动态语言，所以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、自动出错消息以及异常模型，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和PyLint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和PyLint是第三方开源代码包。你可以在<http://www.python.org>或者PyPI网站上找到它们，或者通过常用的Web搜索引擎来查找。

PyUnit (也就是unittest)

在第24章中，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），提供了一个面向对象类框架，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、Komodo、NetBeans和Wing IDE）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的网页以及你爱用的Web搜索引擎，找到有关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，所以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20章比较迭代工具的速度时，我们看过time模块的用法。配置通常是你优化的第一步——通过配置隔离瓶颈，然后对它们的替代编码计时。

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模块可以作为一个脚本运行或导入，并且可以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。要交互地进行配

置，导入profile模块并调用profile.run('code')，把想要配置的代码作为一个字符串传入（例如，对函数的一次调用，或者对整个文件的导入）。要从一个系统shell命令行配置，使用一条形式为python -m profile main.py args（参见附录A了解这一格式的更多内容）的命令。请参阅Python的标准库手册来了解其他的配置选项；例如，cProfile模块，拥有与profile相同的接口，但是运行起来负载较少，因此，它可能更适合于配置长时间运行的程序。

调试器

我们已经在第3章讨论过调试选项（参见本书第3章中的边栏“调试Python代码”）。作为一个回顾，Python的大多数开发IDE都支持基于GUI的调试，并且Python标准库包含了一个命令行源代码调试器模块，称为pdb。这个模块提供了与常用的C语言的调试器（例如，dbx、gdb）工作非常类似的一个命令行界面。

和配置器很相似，pdb调试器可以交互地运行，或者从一个命令行运行，并且可以从一个Python程序导入并调用。要交互地使用它，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。要从一个系统shell命令行启动pdb，使用形式为python -m pdb main.py args...（参见附录A了解这一形式的更多内容）的一条命令。pdb包括了实用的事后分析调用，即pdb.pm()，它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”的调试界面，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。实际上，pdb和IDE在实际中用的不是很多——正如本书第3章所提到的，很多程序员插入print语句或者直接读取Python的出错消息（不是最高端的方法，但是，在当今的Python世界中，往往是最实用的方法能够胜出）。

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，我们在第2章学过，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21章和第33章讨论过），这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。

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23章）、基于类的异常（第33章）、类的伪私有属性（第30章）、文档字符串（第15章）、模块路径配置文件（第21章）、从from *以及__all__列表到_X风格的变量名来隐藏变量名（第24章）、以__name__ == '__main__'技巧来增加自我测试代码（第24章）和使用函数和模块的常见设计规则（第17章、第19章以及第24章），使用面向对象设计模式（第30章及其他），等等。

要学习公开场合的其他大型的Python开发工具，一定要去浏览位于<http://www.python.org>的PyPI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，也是相关主题的一个总结。我们看了常见的异常使用实例，并且简要地介绍了常用的开发工具。

本章还结束了本书核心内容介绍。此时，你已经接触到了大多数程序员所使用的Python的完整子集。实际上，如果你已经阅读到此，应该可以感受到自己是一位正式的Python程序员了。确保下次上网的时候挑选一件T恤衫。

本书的下一部分也是最后一部分，是处理高级的但仍然属于核心语言领域的话题的各章。这些章都是可选的阅读材料，因为并不是每个Python程序员都必须学习这些内容。实际上，大多数人可能在这里就停步了，并且开始在应用程序中体验Python的用途。坦率地讲，应用程序库在实际中比（有些深奥的）高级语言功能更重要。

另一方面，如果你需要关注Unicode或二进制数据这样的内容，必须处理描述符、装饰器和元类这样的API构建工具，或者只是想要更深入地学习，本书的下一部分将帮助你起步。最后一部分中的较大示例也将让你有机会看到已经学习过的概念如何以实际的方式应用。

这是本书核心内容的结束，你将在章末测试方面得到喘息之机——这次只有一个问题。然而，请确保练习本章末尾的测试题，从而评估你对过去的各章学习得如何；由于下一

部分是可选的阅读内容，这是最后的一个练习部分。如果你想要通过一些例子，看看自己已经学过的内容如何组合应用到一个常用应用程序的真实脚本中，查看附录B部分练习4的“解答”。

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，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。
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。你会得到哪种出错信息？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。

作者介绍

作为全球Python培训界的领军人物。**Mark Lutz**是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，这两本书于2009年都已经出版了第3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2009年，他已经开办了225个Python短期培训课程，教授了大约3500名学习者，销售了大约25万册有关Python的书籍。许多书被翻译成十多种语言。

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。你可以通过访问<http://www.rmi.net/~lutz>与他取得联系。

封面介绍

本书的封面动物为林鼠（wood rat，鼠科林鼠属），林鼠能够居住于各种环境（多岩石、灌木丛或沙地），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，有时也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的雕刻版画。

第八部分

高级话题

Unicode和字节字符串

在本书的核心类型部分关于字符串的一章中（第7章），我有意地限制了大多数Python程序员需要了解的字符串话题的子集的范围。因为大多数程序员只是处理像ASCII这样的文本的简单形式，他们快乐地使用着Python的基本的str字符串类型及其相关的操作，并且不需要掌握更加高级的字符串概念。实际上，这样的程序员很大程度上可以忽略Python 3.0中的字符串的变化，并且继续使用他们过去所使用的字符串。

另一方面，一些程序员处理更加专业的数据类型：非ASCII的字符串集、图像文件内容，等等。对于这些程序员（以及其他可能某一天加入这一队伍的程序员），在本章中，我们将介绍Python字符串的其他内容，并且探讨Python字符串模型中一些较为高级的话题。

特别是，我们将介绍Python支持的Unicode文本的基础知识——在国际化应用程序中使用的宽字符串，以及**二进制数据**——表示绝对的字节值的字符串。我们将看到，高级的字符串表示法在Python当前版本中已经产生了分歧：

- *Python 3.0*为二进制数据提供了一种替代字符串类型，并且在其常规的字符串类型中支持Unicode文本（ASCII看作Unicode的一种简单类型）。
- *Python 2.6*为非ASCII Unicode文本提供了一种替代字符串类型，并且在其常规的字符串类型中支持简单文本和二进制数据。

此外，由于Python的字符串模式对于如何处理非ASCII文件有着直接的影响，我们还将在这里介绍相关话题的基础知识。最后，我们还将简单地看看一些高级字符串和二进制工具，例如模式匹配、对象pickle化、二进制数据包装和XML解析，以及Python 3.0的字符串变化对它们产生影响的方式。

正式来说，本章是关于高级话题的一章，因为并不是所有的程序员都需要深入Unicode编码或二进制数据的世界。如果你需要关注处理这两种形式，那么，你将会发现Python的字符串模式提供了所需的支持。

Python 3.0中的字符串修改

Python 3.0中最引入注目的修改之一，就是字符串对象类型的变化。简而言之，Python 2.X的str和unicode类型已经融入了Python 3.0的str和bytes类型，并且增加了一种新的可变的类型bytearray。bytearray类型在Python 2.6中也可以使用（但在更早的版本中不能用），但是，它在Python 3.0中得到完全支持，并且不像是在Python 2.6中那样清楚地区分文本和二进制内容。

特别是，如果我们处理本质上是Unicode或二进制的的数据，这些修改对于代码可能会有切实的影响。实际上，作为首要的一般性规则，我们需要如何关注这一话题，很大程度上取决于遇到如下的哪种情况：

- 如果处理非ASCII Unicode**文本**，例如，在国际化应用程序或某些XML解析器的结果这样的环境中，你将会发现Python 3.0中对文本编码的支持是不同的，但是可能比Python 2.6中的支持更加直接、易用和无缝。
- 如果处理**二进制数据**，例如，使用struct模块处理的图形或音频文件的形式或打包的数据，我们需要理解Python 3.0中新的bytes对象，以及Python 3.0对文本和二进制数据和文件的不同和严格区分。
- 如果不属于前面两种情况的任何一种，在Python 3.0中，通常可以像是在Python 2.6中一样使用字符串：使用通用的str字符串类型、文本文件，以及我们前面所介绍的所有熟悉的字符串操作。字符串将使用平台默认的编码来进行编码和解码（例如，美国的Windows上的ASCII或UTF-8，如果我们仔细检查的话，`sys.getdefaultencoding()`给出默认的编码方式），但是，你可能不会注意。

换句话说，如果你的文本总是ASCII，可以使用常规的字符串对象和文本文件，并且避免下面介绍的大多数情况。正如稍后我们将见到的，ASCII是一种简单的Unicode，并且是其他编码的一个子集，因此，如果你的程序处理ASCII文本，字符串操作和文件“刚好够用”。

即便你遇到了刚刚提及的3种情况的最后一种，然而对Python 3.0字符串模式的基本理解，既可以帮助你理解一些底层的行为，也可以帮助你更容易地掌握Unicode或二进制数据问题，以免它们将来会影响到你。

Python 3.0对Unicode和二进制数据的支持在Python 2.6中也可以使用，虽然形式不同。尽管本章中主要关注的是Python 3.0中的字符串类型，在此过程中，我们还将讨论一些Python 2.6中的不同之处。不管你使用的是哪个版本，我们在这里介绍的工具在很多类型程序中将变得重要起来。

字符串基础知识

在查看任何代码之前，让我们先开始概览一下Python的字符串模型。要理解为什么Python 3.0改变了字符串的工作方式，我们必须先简短地看看字符实际是如何在计算机中表示的。

字符编码方法

大多数程序员把字符串看作是用来表示文本数据的一系列字符。但是，根据必须记录何种字符集，计算机内存中存储字符的方式有所不同。

ASCII标准在美国创建，并且定义了大多数美国程序员使用的文本字符串表示法。ASCII定义了从0到127的字符代码，并且允许每个字符存储在一个8位的字节中（实际上，只有其中的7位真正用到）。例如，ASCII标准把字符'a'映射为整数值97（十六进制中的0x61），它存储在内存和文件的一个单个字节中。如果想要看到这是如何工作的，Python的内置函数ord给出了一个字符的二进制值，并且chr针对一个给定的整数代码值返回其字符：

```
>>> ord('a')           # 'a' is a byte with binary value 97 in ASCII
97
>>> hex(97)
'0x61'
>>> chr(97)            # Binary value 97 stands for character 'a'
'a'
```

然而，有时候每个字符一个字节并不够。例如，各种符号和重音字符并不在ASCII所定义的可能字符的范围中。为了容纳特殊字符，一些标准允许一个8位字节中的所有可能的值（即0到255）来表示字符，并且把（ASCII范围之外的）值128到255分配给特殊字符。这样的标准叫做*Latin-1*，广泛地用于西欧地区。在Latin-1中，127以上的字符代码分配给了重音和其他特殊字符。例如，分配给字节值196的字符，是一个特殊标记的非ASCII字符：

```
>>> 0xC4
196
>>> chr(196)
'Ä'
```


这个标准考虑到范围较广的额外特殊字符。然而，一些字母表定义了如此多的字符，以至于无法把其中的每一个都表示成一个字节。Unicode考虑到更多的灵活性。Unicode文本通常叫做“宽字符”字符串，因为每个字符可能表示为多个字节。Unicode通常用在国际化的程序中，以表示欧洲和亚洲的字符集，它们往往拥有比8位字节所能表示的更多的字符。

要在计算机内存中存储如此丰富的文本，我们要确保字符与原始字节之间可以使用一种编码相互转换，而编码就是把一个Unicode字符转换为字节序列以及从一个字节序列提取字符串的规则。更程序化地说，字节和字符串之间的来回转换由两个术语定义：

- 编码是根据一个想要的编码名称，把一个字符串翻译为其原始字节形式。
- 解码是根据其编码名称，把一个原始字节串翻译为字符串形式的过程。

也就是说，我们从字符串**编码**为原始字节，并且从原始字节解码为字符串。对于某些编码，翻译的过程很简单，例如ASCII和Latin-1，把每个字符映射为一个单个字节，因此，不需要翻译工作。对于其他的编码，映射可能更复杂些，并且每个字符产生多个字节。

例如，广为使用的UTF-8编码，通过采用可变的字节数的方案，允许表示众多的字符。小于128的字符代码表示为单个字节；128和0x7ff (2047)之间的代码转换为两个字节，而每个字节拥有一个128到255之间的值；0x7ff以上的代码转换为3个或4个字节序列，序列中的每个字节的值在128到255之间。这保持了ASCII字符串的紧凑，避免了字节排序问题，并且避免了可能对C库和网络连接引发问题的空（零）字节。

由于编码的字符映射把字符分配给同样的代码以保持兼容性，因此ASCII是Latin-1和UTF-8的**子集**。也就是说，一个有效的ASCII字符串也是一个有效的Latin-1和UTF-8编码字符串。当数据存储到文件中的时候，这也是成立的：每个ASCII文件也是有效的UTF-8文件，因为ASCII是UTF-8的一个7位的子集。

反过来，对于所有小于128的字符代码，UTF-8编码与ASCII是二进制兼容的。Latin-1和UTF-8只不过是考虑到了额外的字符：Latin-1是为了在一个字节内映射为128到255的值，UTF-8是考虑到可能用多个字节表示的字符串。其他的编码以类似的方式允许较宽的字符集合，但是，所有这些，ASCII、Latin-1、UTF-8以及很多其他的编码，都被认为是Unicode。

对于Python程序员来说，编码指定为包含了编码名的字符串。Python带有大约100种不同的编码，参见Python库参考可以找到一个完整的列表。导入encodings模块并运行help(encodings)也会显示很多的编码名称，一些是在Python中实现的，一些是在C中实现的。一些编码也有多个名称，例如，*latin-1*、*iso_8859_1*和8859都是相同编码的名

称，即Latin-1。我们将会在本章稍后学习在脚本中编写Unicode字符串技术的时候，再次介绍编码。

要了解关于Unicode的更多内容，参见Python标准手册集。它在“Python HOWTOs”部分包括了一个“Unicode HOWTO”，其中提供了额外的背景知识，考虑到篇幅的问题，我们暂时在此省略。

Python的字符串类型

具体来说，Python语言提供了字符串类型在脚本中表示字符文本。在脚本中所使用的字符串类型取决于所使用的Python的版本。*Python 2.X*有一种通用的字符串类型来表示二进制数据和像ASCII这样的8位文本，还有一种特定的类型用来表示多字节Unicode文本：

- `str`表示8位文本和二进制数据。
- `unicode`用来表示宽字符Unicode文本。

Python 2.X的两种字符串类型是不同的（`unicode`考虑到字符的额外大小并且支持编码和解码），但是，它们的操作集大多是重叠的。Python 2.X中的`str`字符串类型用于可以用8位字节表示的文本，以及绝对字节值所表示的二进制数据。

相反，*Python 3.X*带有3种字符串对象类型——一种用于文本数据，两种用于二进制数据：

- `str`表示Unicode文本（8位的和更宽的）。
- `bytes`表示二进制数据。
- `bytearray`，是一种可变的`bytes`类型。

正如前面所提到的，`bytearray`在Python 2.6中可以使用，但它是从Python 3.0才有的升级功能，此后较少有特定内容的行为，并且通常看做是一个Python 3.0类型。

Python 3.0中所有3种字符串类型都支持类似的操作集，但是，它们都有不同的角色。Python 3.X之后关于这一修改的主要目标是，把Python 2.X中常规的和Unicode字符串类型合并到一个单独的字符串类型中，以支持常规的和Unicode文本：开发者想要删除Python 2.X中的字符串区分，并且让Unicode的处理更加自然。假设ASCII和其他的8位文本真的是一种简单的Unicode，这种融合听起来很符合逻辑。

为了实现这一点，Python 3.0的`str`类型定义为一个不可改变的字符序列（不一定是字节），这可能是像ASCII这样的每个字符一个字节的常规文本，或者是像UTF-8 Unicode这样可能包含多字节字符的字符集文本。你的脚本所使用的字符串处理，带有这种每个

平台默认编码的类型，但是，在内存中以及在文件之间来回转换的时候，将会提供明确的编码名，以便在`str`对象和不同的方案之间来回转换。

尽管Python 3.0新的`str`类型确实实现了想要的字符串/Unicode结合，但很多程序仍然需要处理那些没有针对每个任意文本格式都编码的raw字节数据。图像和声音文件，以及用来与设备接口的打包数据，或者你想要用Python的`struct`模块处理的C程序，都属于这一类型。因此，为了支持真正的二进制数据的处理，还引入了一种新的类型，即`bytes`。

在Python 2.X中，通用的`str`类型填补了二进制数据的这一角色，因为字符串也只是字节的序列（单独的`unicode`类型处理宽字符串）。在Python 3.0中，`bytes`类型定义为一个8位整数的不可变序列，表示绝对的字节值。此外，Python 3.0的`bytes`类型支持几乎`str`类型所做的所有相同操作：这包括字符串方法、序列操作，甚至`re`模块模式匹配；但是不包括字符串格式化。

一个Python 3.0 `bytes`对象其实只是较小整数的一个序列，其中每个整数的范围都在0到255之间；索引一个`bytes`将返回一个`int`，分片一个`bytes`将返回另一个`bytes`，并且在一个`bytes`上运行内置函数`list`将返回整数，而不是字符的一个列表。当用那些假设字符的操作处理`bytes`的时候，`bytes`对象的内容被假设为ASCII编码的字节（例如，`isalpha`方法假设每个字节都是一个ASCII字符代码）。此外，为了方便起见，`bytes`对象打印为字符串而不是整数。

尽管如此，Python的开发者也在Python 3.0中添加了一个`bytearray`类型，`bytearray`是`bytes`类型的一个变体，它是可变的并且支持原处修改。它支持`str`和`bytes`所支持的常见的字符串操作，以及和列表相同的很多原处修改操作（例如，`append`和`extend`方法，以及向索引赋值）。假设字符串可以作为raw字节对待，`bytearray`最终为字符串数据添加了直接原处可修改的能力，这在Python 2.0中不通过转换为一个可变类型是不可能做到的，并且也是Python 3.0的`str`或`bytes`所不支持的。

尽管Python 2.6和Python 3.0提供了很多相同的功能，但它们还是以不同方式包装了功能。实际上，从Python 2.6到Python 3.0的字符串类型映射并不是直接的，Python 2.8的`str`等同于Python 3.0中的`str`和`bytes`，并且Python 3.0的`str`等同于Python 2.6中的`str`和`Unicode`。此外，Python 3.0的可变的`bytearray`是独特的。

然而，实际上，这种不对称并不像听上去那么令人生畏。它可以概括如下：在Python 2.6中，我们可以对简单的文本使用`str`并且对文本的更高级的形式使用二进制数据和`unicode`；在Python 3.0中，我们将针对任何类型的文本（简单的和`Unicode`）使用`str`，并且针对二进制数据使用`bytes`或`bytearray`。实际上，这种选择常常由你所使用的工具决定，尤其是在文件处理工具的例子中，这是下一小节的主题。

文本和二进制文件

文件I/O（输入和输出）在Python 3.0中也有所改进，以反映str/bytes的区分以及对编码Unicode文本的自动支持。Python现在在文本文件和二进制文件之间做了一个明显的独立于平台的区分：

文本文件

当一个文件以**文本模式**打开的时候，读取其数据会自动将其内容解码（每个平台一个默认的或一个提供的编码名称），并且将其返回为一个str，写入会接受一个str，并且在将其传输到文件之间自动编码它。文本模式的文件还支持统一的行尾转换和额外的编码特定参数。根据编码名称，文本文件也自动处理文件开始处的字节顺序标记序列（稍后详细介绍）。

二进制文件

通过在内置的open调用的模式字符串参数添加一个b（只能小写），以**二进制模式**打开一个文件的时候，读取其数据不会以任何方式解码它，而是直接返回其内容raw并且未经修改，作为一个bytes对象；写入类似地接受一个bytes对象，并且将其传送到文件中而未经修改。二进制模式文件也接受一个bytearray对象作为写入文件中的内容。

由于str和bytes之间的语言差距明显，所以必须确定数据本质上是文本或二进制，并且在脚本中相应地使用str或bytes对象来表示其内容。最终，以何种模式打开一个文件将决定脚本使用何种类型的对象来表示其内容：

- 如果正在处理图像文件，其他程序创建的、而且必须解压的打包数据，或者一些设备数据流，则使用bytes和**二进制模式**文件处理它更合适。如果想要更新数据而不在内存中产生其副本，也可以选择使用bytearray。
- 如果你要处理的内容实质是文本的内容，例如程序输出、HTML、国际化文本或CSV或XML文件，可能要使用str和**文本模式**文件。

注意，内置函数open的**模式字符串**参数（函数的第二个参数）在Python 3.0中变得至关重要，因为其内容不仅指定了一个**文件处理模式**，而且暗示了一个Python对象类型。通过给模式字符串添加一个b，我们可以指定二进制模式，并且当读取或写入的时候，将要接收或者必须提供一个bytes对象来表示文件的内容。没有b，我们的文件将以文本模式处理，并且将使用str对象在脚本中表示其内容。例如，模式rb、wb和rb+暗示bytes，而r、w+和rt暗示str。

文本模式文件也处理在某种编码方案下可能出现在文件开始处的**字节顺序标记**（byte order marker, BOM）序列。例如，在UTF-16和UTF-32编码中，BOM指定大尾还是小

尾格式（基本上，是确定一个位字符串的哪一端最重要）。一般来说，一个UTF-8文本文件也包含了一个BOM来声明它是UTF-8，但并不保证这样。当使用这些编码方法来读取和写入数据的时候，如果BOM有一个通用的编码暗示或者如果提供一个更为具体的编码名来强制这点的话，Python会自动省略或写出BOM。例如，BOM总是针对“utf-16”处理，更具体的编码名“utf-16-le”表示小尾UTF-16格式，更具体的编码名“utf-8-sig”迫使Python在输入和输出上分别都针对UTF-8文本省略并写入一个BOM（通用名称“utf-8”并不这么做）。

我们还将在本章后面的“在Python 3.0中处理BOM”一节中介绍更多有关BOM和文件的内容。首先，让我们探讨Python的新的Unicode字符串模型的含义。

Python 3.0中的字符串应用

让我们再看一些例子，这些例子展示了如何使用Python 3.0字符串类型。提前强调一点：本节中的代码都只在Python 3.0下运行和使用。然而，基本的字符串操作通常在Python各版本中是可移植的。用str类型表示的简单的ASCII字符串在Python 2.6和Python 3.0下都能工作（并且确实像我们在本书第7章中所见到的那样）。此外，尽管Python 2.6中没有bytes类型（它只有通用的str），它通常按照这样的方式来运行代码：在Python 2.6中，调用bytes(X)作为str(X)的同义词出现，并且新的常量形式b'...'看做与常量'...'相同。然而，我们仍然可能在一些个别的例子中遇到版本差异；例如，Python 2.6的bytes调用，不允许Python 3.0中bytes所要求的第二个参数（编码名称）。

常量和基本属性

当调用str或bytes这样的内置函数的时候，会引发Python 3.0字符串对象，来处理调用open（下一小节介绍）所创建的一个文件，或者在脚本中编写常量语法。对于后者，一种新的常量形式b'xxx'（以及对等的B'xxx'）用来创建Python 3.0中的bytes对象，bytearray对象可能通过调用bytearray函数来创建，这会带有各种可能的参数。

更正式地说，在Python 3.0中，所有当前字符串常量形式，'xxx'、"xxx"和三引号字符串块，都产生一个str；在它们任何一种前面添加一个b或B，则会创建一个bytes。这个新的b'...'字节常量类似于用来抑制反斜杠转义的r'...' raw字符串。考虑在Python 3.0中运行如下语句：

```
C:\misc> c:\python30\python

>>> B = b'spam'           # Make a bytes object (8-bit bytes)
>>> S = 'eggs'            # Make a str object (Unicode characters, 8-bit or wider)

>>> type(B), type(S)
```

```
(<class 'bytes'>, <class 'str'>)

>>> B                                     # Prints as a character string, really sequence of ints
b'spam'
>>> S
'eggs'
```

`bytes`对象实际上是较小的整数的一个序列，尽管它尽可能地将自己的内容打印为字符：

```
>>> B[0], S[0]                           # Indexing returns an int for bytes, str for str
(115, 'e')

>>> B[1:], S[1:]                         # Slicing makes another bytes or str object
(b'pam', 'ggs')

>>> list(B), list(S)
([115, 112, 97, 109], ['e', 'g', 'g', 's']) # bytes is really ints
```

`bytes`对象是不可修改的，就像`str`（尽管后面将要介绍的`bytearray`是可以修改的），我们可以把一个`str`、`bytes`或整数赋给一个`bytes`对象的偏移。`bytes`前缀对于任何字符串常量形式也有效：

```
>>> B[0] = 'x'                           # Both are immutable
TypeError: 'bytes' object does not support item assignment

>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment

>>> B = B"""                             # bytes prefix works on single, double, triple quotes
... xxxx
... yyyy
... ""
>>> B
b'\nxxxx\nyyyy\n'
```

正如前面提到的，在Python 2.6中，为了兼容性而使用`b'xxx'`，但是它与`'xxx'`是相同的，并且产生一个`str`，并且，`bytes`只是`str`的同义词；正如你已经看到的，在Python 3.0中，这二者都解决了`bytes`类型之间的差异。还要注意，Python 2.6中的`u'xxx'`和`U'xxx'` Unicode字符串常量形式在Python 3.0中已经取消了，而是使用`'xxx'`替代，因为所有的字符串都是Unicode，即便它们包含所有的ASCII字符（在本章后面的“编码非ASCII文本”小节将更多地讨论）。

转换

尽管Python 2.X允许`str`和`unicode`类型对象自由地混合（如果该字符串只包含7位的ASCII文本的话），Python 3.0引入了一个更鲜明的区分——`str`和`bytes`类型对象不在表

达式中自动地混合，并且当传递给函数的时候不会自动地相互转换。期待一个str对象作为参数的函数，通常不能接受一个bytes；反之亦然。

因此，Python 3.0基本上要求遵守一种类型或另一种类型，或者手动执行显式转换：

- `str.encode()`和`bytes(S, encoding)`把一个字符串转换为其raw bytes形式，并且在此过程中根据一个str创建一个bytes。
- `bytes.decode()`和`str(B, encoding)`把raw bytes转换为其字符串形式，并且在此过程中根据一个bytes创建一个str。

`encode`和`decode`方法（以及文件对象，将在下一节介绍）针对你的平台使用一个默认编码，或者一个显式传入的编码名。例如，在Python 3.0中：

```
>>> S = 'eggs'
>>> S.encode()                                # str to bytes: encode text into raw bytes
b'eggs'

>>> bytes(S, encoding='ascii')                # str to bytes, alternative
b'eggs'

>>> B = b'spam'
>>> B.decode()                                # bytes to str: decode raw bytes into text
'spam'

>>> str(B, encoding='ascii')                  # bytes to str, alternative
'spam'
```

这里有两点要注意。首先，平台的默认编码在`sys`模块中可用，但是，`bytes`的编码参数不是可选的，即便它在`str.encode`（和`bytes.decode`）中亦是如此。

其次，尽管调用str并不像bytes那样要求编码参数，但在str调用中省略它并不意味着它是默认的，相反，不带编码的一个str调用返回bytes对象的打印字符串，而不是其str转换后的形式（这通常不是我们想要的！）假设B和S仍然和前面相同：

```
>>> import sys
>>> sys.platform                                # Underlying platform
'win32'
>>> sys.getdefaultencoding()                  # Default encoding for str here
'utf-8'

>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                                     # str without encoding
"b'spam'"                                     # A print string, not conversion!
>>> len(str(B))
7
>>> len(str(B, encoding='ascii'))             # Use encoding to convert to str
4
```


编码Unicode字符串

当我们开始处理真正的非ASCII Unicode文本的时候，编码和解码变得更有意义了。要在字符串中编码任意的Unicode字符，有些字符可能甚至无法在键盘上输入，Python的字符串常量支持"\xNN"十六进制字节值转义以及"\uNNNN"和"\UNNNNNNNNN" Unicode转义。在Unicode转义中，第一种形式给出了4个十六进制位以编码1个2字节（16位）字符码，而第二种形式给出8个十六进制位表示4字节（32位）代码。

编码ASCII文本

让我们来看一些例子以介绍文本编码的基础知识。正如我们已经介绍过的，ASCII文本是一种简单的Unicode，存储为表示字符的字节值的一个序列：

```
C:\misc> c:\python30\python

>>> ord('X')           # 'X' has binary value 88 in the default encoding
88
>>> chr(88)            # 88 stands for character 'X'
'X'

>>> S = 'XYZ'          # A Unicode string of ASCII text
>>> S
'XYZ'
>>> len(S)              # 3 characters long
3
>>> [ord(c) for c in S] # 3 bytes with integer ordinal values
[88, 89, 90]
```

像这样的常规7位ASCII文本，在本章前面所介绍的每种Unicode编码方案中，都是以每字节一个字符的方式表现：

```
>>> S.encode('ascii')   # Values 0..127 in 1 byte (7 bits) each
b'XYZ'
>>> S.encode('latin-1') # Values 0..255 in 1 byte (8 bits) each
b'XYZ'
>>> S.encode('utf-8')    # Values 0..127 in 1 byte, 128..2047 in 2, others 3 or 4
b'XYZ'
```

实际上，以这种方法编码ASCII文本而返回的bytes对象，其实是较短整数的一个序列，只不过这个序列尽可能地打印为ASCII字符：

```
>>> S.encode('latin-1')[0]
88
>>> list(S.encode('latin-1'))
[88, 89, 90]
```

编码非ASCII文本

要编码非ASCII字符，可能在字符串中使用十六进制或Unicode转义；十六进制转义限制于单个字节的值，但Unicode转义可以指定其值有两个和四个字节宽度的字符。例如，十六进制值0xCD 和0xE8，是ASCII的7位字符范围之外的两个特殊的重音字符，但是，我们可以将其嵌入Python 3.0的str对象中，因为如今的str支持Unicode：

```
>>> chr(0xc4)                                # 0xC4, 0xE8: characters outside ASCII's range
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'                            # Single byte 8-bit hex escapes
>>> S
'Äè'

>>> S = '\u00c4\u00e8'                        # 16-bit Unicode escapes
>>> S
'Äè'
>>> len(S)                                    # 2 characters long (not number of bytes!)
2
```

编码和解码非ASCII文本

现在，如果我们试图把一个非ASCII字符串**编码**为raw字节以像ASCII一样使用，我们会得到一个错误。像Latin-1这样的编码是有效的，并且为每个字符分配一个字节；像UTF-8这样的编码为每个字符分配2个字节。如果把这个字符串写入一个文件，这里显示的raw字节就是针对给定的编码类型而实际存储在文件中的内容：

```
>>> S = '\u00c4\u00e8'
>>> S
'Äè'
>>> len(S)
2

>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)

>>> S.encode('latin-1')                        # One byte per character
b'\xc4\xe8'

>>> S.encode('utf-8')                          # Two bytes per character
b'\xc3\x84\xc3\xa8'

>>> len(S.encode('latin-1'))                    # 2 bytes in latin-1, 4 in utf-8
2
>>> len(S.encode('utf-8'))
4
```

也可以用其他的办法，从一个文件读入raw字节并且将其解码回一个Unicode字符串。然而，正如我们随后将看到的，给open调用的编码模式会引发对输入自动使用这一解码（避免了在按照字节块读取的时候，由于读取部分字符序列可能引发的问题）：

```
>>> B = b'\xc4\xe8'
>>> B
b'\xc4\xe8'

>>> len(B)                                # 2 raw bytes, 2 characters
2
>>> B.decode('latin-1')                    # Decode to latin-1 text
'Äè'

>>> B = b'\xc3\x84\xc3\xa8'
>>> len(B)                                # 4 raw bytes
4
>>> B.decode('utf-8')
'Äè'
>>> len(B.decode('utf-8'))                  # 2 Unicode characters
2
```

其他Unicode编码技术

一些编码甚至使用较大的字节序列来表示字符。当需要的时候，我们可以为自己字符串中的字符指定16位或32位的Unicode值，例如，对于前者使用"\u..."表示4个十六进制位，对于后者使用"\U..."表示8个十六进制位。

```
>>> S = 'A\u00c4B\u000000e8C'
>>> S                                       # A, B, C, and 2 non-ASCII characters
'AÄBèC'
>>> len(S)                                 # 5 characters long
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))               # 5 bytes in latin-1
5

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))                 # 7 bytes in utf-8
7
```

有趣的是，另一些编码可能使用有很大不同的字节格式。例如，cp500 EBCDIC编码，它编码ASCII的方式，甚至和我们已经见过的方法都不同（由于Python为我们编码和解码，所以我们通常只是在提供编码名的时候才需要关注这一点）：

```
>>> S
'AÄBèC'
>>> S.encode('cp500')                      # Two other Western European encodings
```

```

b'\xc1c\xc2T\xc3'
>>> S.encode('cp850')           # 5 bytes each
b'A\x8eB\x8aC'

>>> S = 'spam'                   # ASCII text is the same in most
>>> S.encode('latin-1')
b'spam'
>>> S.encode('utf-8')
b'spam'
>>> S.encode('cp500')           # But not in cp500: IBM EBCDIC!
b'\xa2\x97\x81\x94'
>>> S.encode('cp850')
b'spam'

```

从技术上讲，你也可以使用chr而不是Unicode转义或十六进制转义来构建Unicode字符串片段，但是，对于较大的字符串来说，这可能变得很繁琐：

```

>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'
>>> S
'AÄBèC'

```

这里有两点要注意。首先，Python 3.0允许特殊的字符以十六进制和Unicode转义的方式编码到str字符串中，但是，只能以十六进制转义的方式编码到bytes字符串中：Unicode转义会默默地逐字转换为字节常量，而不是转义。实际上，bytes必须编码为str字符串，以便将其打印为非ASCII字符：

```

>>> S = 'A\xC4B\xe8C'           # str recognizes hex and Unicode escapes
>>> S
'AÄBèC'

>>> S = 'A\u00C4B\u000000E8C'
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xe8C'          # bytes recognizes hex but not Unicode
>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\u000000E8C'   # Escape sequences taken literally!
>>> B
b'A\\u00C4B\\u000000E8C'

>>> B = b'A\xC4B\xe8C'          # Use hex escapes for bytes
>>> B                             # Prints non-ASCII as hex
b'A\xc4B\xe8C'
>>> print(B)
b'A\xc4B\xe8C'
>>> B.decode('latin-1')         # Decode as latin-1 to interpret as text
'AÄBèC'

```

其次，字节常量要求字符要么是ASCII字符，要么如果它们的值大于127就进行转义。另

一方面，`str`字符串允许常量包含源字符集中的任何字符（稍后讨论，除非在源文件中给定一个编码声明，否则默认为UTF-8）：

```
>>> S = 'AÄBëC'                                # Chars from UTF-8 if no encoding declaration
>>> S
'AÄBëC'

>>> B = b'AÄBëC'
SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xc4B\xe8C'                          # Chars must be ASCII, or escapes
>>> B
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBëC'

>>> S.encode()                                  # Source code encoded per UTF-8 by default
b'A\xc3\x84B\xc3\xa8C'                          # Uses system default to encode, unless passed
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'

>>> B.decode()                                  # Raw bytes do not correspond to utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
```

转换编码

到目前为止，我们已经编码和解码字符串以查看其结构。更一般地讲，我们总是可以把一个字符串转换为不同于源字符集默认的一种编码，但是，我们必须显式地提供一个编码名称以进行编码和解码：

```
>>> S = 'AÄBëC'
>>> S
'AÄBëC'
>>> S.encode()                                  # Default utf-8 encoding
b'A\xc3\x84B\xc3\xa8C'

>>> T = S.encode('cp500')                       # Convert to EBCDIC
>>> T
b'\xc1c\xc2T\xc3'

>>> U = T.decode('cp500')                       # Convert back to Unicode
>>> U
'AÄBëC'

>>> U.encode()                                  # Default utf-8 encoding again
b'A\xc3\x84B\xc3\xa8C'
```

记住，只有当手动编写非ASCII Unicode的时候，才必须用到特殊的Unicode和十六进制字符转义。实际上，我们往往从文件载入这样的文本。正如我们将从本书稍后见到的，Python 3.0的文件对象（用`open`内置函数创建的）在读取文本字符串的时候自动地编码

它们，并且在写入文本字符串的时候自动解码它们。因此，脚本往往可以广泛地处理字符串，而不必直接编码特殊字符。

在本章稍后我们还将看到，从文件传出和向文件传入字符串的时候，也可以在编码之间进行转换，使用与上一个例子中非常类似的一种技术。尽管在打开一个文件的时候仍然需要显式地提供编码名称，但文件接口自动完成大多数转换工作。

在Python 2.6中编码Unicode字符串

既然已经介绍了Python 3.0中基本的Unicode字符串知识，我需要说明的是，在Python 2.6中可以做很多相同的事情，尽管所使用的工具是不同的。unicode在Python 2.6中是可用的，但它是与str截然不同的数据类型，并且当常规字符串和Unicode字符串兼容的时候，它允许自由组合。实际上，当遇到把raw字节编码为一个Unicode字符串的时候，我们基本上可以把Python 2.6的str当做Python 3.0的bytes，只要它保持正确的形式。如下是在Python 2.6中的实际应用（本章所有其他节都是在Python 3.0下运行的情况）：

```
C:\misc> c:\python26\python
>>> import sys
>>> sys.version
'2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)]'

>>> S = 'A\xC4B\xe8C'                                # String of 8-bit bytes
>>> print S                                             # Some are non-ASCII
AÄBèC

>>> S.decode('latin-1')                                # Decode byte to latin-1 Unicode
u'A\xc4B\xe8C'

>>> S.decode('utf-8')                                  # Not formatted as utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid data

>>> S.decode('ascii')                                  # Outside ASCII range
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal
not in range(128)
```

为了存储任意的编码的Unicode文本，用u'xxx'常量形式创建一个unicode对象（这个常量在Python 3.0中不再可用，因为Python 3.0中所有字符串都支持Unicode）：

```
>>> U = u'A\xC4B\xe8C'                                # Make Unicode string, hex escapes
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC
```

一旦创建了它，可以把Unicode文本转换为不同的raw字节编码，这类似于在Python 3.0中把str对象编码为bytes对象：

```
>>> U.encode('latin-1')           # Encode per latin-1: 8-bit bytes
'A\xc4B\xe8C'
>>> U.encode('utf-8')             # Encode per utf-8: multibyte
'A\xc3\x84B\xc3\xa8C'
```

在Python 2.6中，非ASCII字符可以用十六进制或Unicode转义来编写到字符串常量中，就像在Python 3.0中一样。然而，和Python 3.0中的bytes一样，在Python 2.6中，`"\u..."`和`"\U..."`转义只是识别为unicode字符串，而不是8位str字符串：

```
C:\misc> c:\python26\python
>>> U = u'A\xc4B\xe8C'           # Hex escapes for non-ASCII
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> U = u'A\u00C4B\u0000000E8C'  # Unicode escapes for non-ASCII
>>> U                             # u" = 16 bits, U" = 32 bits
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> S = 'A\xc4B\xe8C'            # Hex escapes work
>>> S
'A\xc4B\xe8C'
>>> print S                       # But some print oddly, unless decoded
A-BFC
>>> print S.decode('latin-1')
AÄBèC

>>> S = 'A\u00C4B\u0000000E8C'   # Not Unicode escapes: taken literally!
>>> S
'A\\u00C4B\\u0000000E8C'
>>> print S
A\u00C4B\u0000000E8C
>>> len(S)
19
```

就像Python 3.0中的str和bytes一样，Python 2.6的unicode和str共享几乎相同的操作集，因此，除非你需要转换为其他的编码，通常可以把unicode当做是str一样对待。然而，Python 2.6和Python 3.0之间的一个主要区别在于，unicode和非Unicode的str对象可以在表达式中自由地混合，并且，只要str和unicode的编码兼容，Python将自动将其向上转换为unicode（在Python 3.0中，str和bytes不会自动混合，并且需要手动转换）：

```
>>> u'ab' + 'cd'                 # Can mix if compatible in 2.6
u'abcd'                          # 'ab' + b'cd' not allowed in 3.0
```

实际上，类型的不同对于Python 2.6中的代码往往很小。像常规的字符串一样，Unicode字符串也可以合并、索引、分片，用re模块匹配，等等，并且它们不能原处修改。如果需要在两种类型之间显式地转换，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'
```

然而，Python 2.6中这种自由混合字符串类型的方法，只有在字符串和unicode对象的编码类型兼容的情况下才有效：

```
>>> S = 'A\xc4B\xe8C'           # Can't mix if incompatible
>>> U = u'A\xc4B\xe8C'
>>> S + U
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal
not in range(128)

>>> S.decode('latin-1') + U      # Manual conversion still required
u'A\xc4B\xe8CA\xc4B\xe8C'

>>> print S.decode('latin-1') + U
AÄBèCAÄBèC
```

最后，正如我们将在本章稍后更具体介绍的，Python 2.6的open调用只支持8位字节的文件，将其内容返回为str字符串；将其内容解释为文本，还是解释为二进制数并需要根据需要解码，这取决于你。要读取和编写Unicode文件并自动编码或解码器内容，使用Python 2.6的codecs.open调用，Python 2.6的库手册中有所介绍。这个调用提供了与Python 3.0的open相同的功能，并且使用Python 2.6的unicode对象来表示文件内容——读取一个文件，把编码的字节翻译为解码的Unicode字符，并且在文件打开的时候把翻译字符串写入想要的指定编码。

源文件字符集编码声明

Unicode转义代码对于字符串常量中偶尔出现的Unicode字符很好用，但是，如果需要频繁地在字符串中嵌入非ASCII文本的话，这会变得很繁琐。对于在脚本文件中编码的字符串，Python默认地使用UTF-8编码，但是，它允许我们通过包含一个注释来指明想要的编码，从而将默认值修改为支持任意的字符集。这个注释必须拥有如下的形式，并且在Python 2.6或Python 3.0中必须作为脚本的第一行或第二行出现：

```
# -*- coding: latin-1 -*-
```

当出现这种形式的注释时，Python将自然按照给定的编码来识别表示的字符串。这意味着，我们可以在一个文本编辑中编辑脚本文件来正确地接受和显示重音及其他非ASCII字符，并且Python将在字符串常量中正确地解码它们。例如，注意如下文件text.py的顶部，注释是如何允许Latin-1字符嵌入字符串中的：

```
# -*- coding: latin-1 -*-
```

```

# Any of the following string literal forms work in latin-1.
# Changing the encoding above to either ascii or utf-8 fails,
# because the 0xc4 and 0xe8 in myStr1 are not valid in either.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\u000000e8C'

myStr3 = 'A' + chr(0xc4) + 'B' + chr(0xe8) + 'C'

import sys
print('Default encoding:', sys.getdefaultencoding())

for aStr in myStr1, myStr2, myStr3:
    print('{0}, strlen={1}, '.format(aStr, len(aStr)), end='')

    bytes1 = aStr.encode()                # Per default utf-8: 2 bytes for non-ASCII
    bytes2 = aStr.encode('latin-1')        # One byte per char
    #bytes3 = aStr.encode('ascii')          # ASCII fails: outside 0..127 range

    print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))

```

运行这段脚本，将产生如下输出：

```

C:\misc> c:\python30\python text.py
Default encoding: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5

```

由于大多数程序员可能都遵从标准的UTF-8编码，所以关于这一选项以及其他高级Unicode支持的话题，例如字符串中的属性名和字符名转义，我们参考Python的标准手册以了解更多细节。

使用Python 3.0 Bytes对象

我们在第7章学习了各种针对Python 3.0通用str对象的操作，基本的字符串类型在Python 2.6和Python 3.0中都能同样地工作，因此，我们不会再回顾这一主题。相反，让我们深入一步介绍Python 3.0中新的bytes类型所提供的操作集。

正如前面提到的，Python 3.0 bytes对象是较小整数的一个序列，其中每个整数都在0到255之间，并且在显示的时候恰好打印为ASCII字符。它支持序列操作以及str对象（在Python 2.X中是str类型）上可用的大多数同样方法。然而，bytes不支持格式化方法或%格式化表达式，并且，不能不经过显式转换就将bytes和str类型对象混合和匹配——我们通常使用针对文本数据使用所有的str类型对象和文本文件，并且针对二进制数据使用所有的bytes对象和二进制文件。

方法调用

如果你真的想要看看`str`拥有哪些`bytes`所没有的属性，总是可以查看它们的`dir`内置函数调用结果。输出结果能够告诉你有关它们所支持的表达式操作符的一些事情（例如，`__mod__`和`__rmod__`实现了`%`操作符）：

```
C:\misc> c:\python30\python

# Attributes unique to str

>>> set(dir('abc')) - set(dir(b'abc'))
{'isprintable', 'format', '__mod__', 'encode', 'isidentifier',
 'formatter_field_name_split', 'isnumeric', '__rmod__', 'isdecimal',
 'formatter_parser', 'maketrans'}

# Attributes unique to bytes

>>> set(dir(b'abc')) - set(dir('abc'))
{'decode', 'fromhex'}
```

正如你所看到的，`str`和`bytes`拥有几乎相同的功能。它们唯一的属性是那些不能应用于对方的通用方法；例如，`decode`把一个`raw bytes`转换为其`str`表示，并且`encode`把一个字符串转换为其`raw bytes`表示。大多数这样的方法是相同的，尽管`bytes`方法需要`bytes`参数（再次，Python 3.0的字符串类型不能混合）。此外，还要记住，`bytes`对象是不可改变的，就像是Python 2.6和Python 3.0中的`str`对象一样（为了简单起见，这里简化了出错消息）：

```
>>> B = b'spam'                                # b'...' bytes literal
>>> B.find(b'pa')
1

>>> B.replace(b'pa', b'XY')                     # bytes methods expect bytes arguments
b'sXYm'

>>> B.split(b'pa')
[b's', b'm']

>>> B
b'spam'

>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment
```

一个显著的区别是，字符串格式化只在Python 3.0中对`str`有效，对`bytes`对象无效（参见本书第7章了解关于字符串格式化表达式和方法的更多内容）：

```
>>> b'%s' % 99
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'

>>> '%s' % 99
```

```
'99'

>>> b'{0}'.format(99)
AttributeError: 'bytes' object has no attribute 'format'

>>> '{0}'.format(99)
'99'
```

序列操作

除了方法调用，我们知道的在Python 2.X中用于字符串和列表的所有常见通用序列操作，也都期待在Python 3.0的str和bytes上有效，这包括索引、分片、合并，等等。注意，如下的代码索引一个bytes对象并返回一个给出了该字节的二进制值的整数；bytes实际上是8位整数的一个序列，但是，当作为整体显示的时候，为了方便起见，它打印为ASCII编码的字符的一个字符串。要查看一个给定的字节的值，使用chr内置函数来将其转换回字符，如下所示：

```
>>> B = b'spam'                                # A sequence of small ints
>>> B                                           # Prints as ASCII characters
b'spam'

>>> B[0]                                       # Indexing yields an int
115
>>> B[-1]
109

>>> chr(B[0])                                # Show character for int
's'
>>> list(B)                                  # Show all the byte's int values
[115, 112, 97, 109]

>>> B[1:], B[:-1]
(b'pam', b'spa')

>>> len(B)
4

>>> B + b'lmn'
b'spamlmn'
>>> B * 4
b'spamspamspamspam'
```

创建bytes对象的其他方式

到目前为止，我们已经见到了用b'...'常量语法创建bytes对象的主要方式；也可以用一个str和一个编码名来调用bytes构造函数，用一个可迭代的整数表示的字节值来调用bytes构造函数，或者按照每个默认（或传入的）编码来编码一个str对象，从而创建bytes对象。正如我们已经见到过的，编码会接受一个str并根据编码声明来返回该字

字符串的raw二进制字节值；相反，解码会接受一个raw bytes序列并将其编码为字符串表示，即一系列可能宽度的字符。这两种操作都会创建新的字符串对象：

```
>>> B = b'abc'
>>> B
b'abc'

>>> B = bytes('abc', 'ascii')
>>> B
b'abc'

>>> ord('a')
97
>>> B = bytes([97, 98, 99])
>>> B
b'abc'

>>> B = 'spam'.encode()                # Or bytes()
>>> B
b'spam'
>>>
>>> S = B.decode()                      # Or str()
>>> S
'spam'
```

从更大的角度来看，这些操作的最后两个是在str和bytes之间转换的真正工具，这是前面介绍过的话题，下一小节还将继续展开介绍。

混合字符串类型

在前面的“方法调用”小节的replace调用中，我们必须传入两个bytes对象，str对象在这里无效。尽管Python 2.X尽可能自动在str和unicode之间转换（例如，当str是一个7位ASCII文本的时候），Python 3.0需要在某些环境下要求特殊的字符串类型并且如果需要的话期待手动转换：

```
# Must pass expected types to function and method calls

>>> B = b'spam'

>>> B.replace('pa', 'XY')
TypeError: expected an object with the buffer interface

>>> B.replace(b'pa', b'XY')
b'sXYm'

>>> B = B'spam'
>>> B.replace(bytes('pa'), bytes('xy'))
TypeError: string argument without an encoding

>>> B.replace(bytes('pa', 'ascii'), bytes('xy', 'utf-8'))
b'sxym'
```

Must convert manually in mixed-type expressions

```
>>> b'ab' + 'cd'
TypeError: can't concat bytes to str

>>> b'ab'.decode() + 'cd'                                     # bytes to str
'abcd'

>>> b'ab' + 'cd'.encode()                                       # str to bytes
b'abcd'

>>> b'ab' + bytes('cd', 'ascii')                               # str to bytes
b'abcd'
```

尽管你可以自行创建`bytes`对象，以表示打包的二进制数据，但它们也可以在二进制模式下读取打开的文件从而自动创建对象，正如我们将在本章稍后详细介绍的。首先，我们应该介绍`bytes`的亲密且可变的“近亲”。

使用Python 3.0（和Python 2.6）`bytearray`对象

到目前为止，我们已经关注了`str`和`bytes`，因为它们包含了Python 2.X的`unicode`和`str`。然而，Python 3.0还有第三个字符串类型`bytearray`，这是范围在0到255之间的整数的一个可变的序列，其本质是`bytes`的可变的变体。同样，它支持和`bytes`同样的字符串方法和序列操作，并且与列表支持同样多的可变的原处修改操作。`bytearray`类型在Python 2.6中也可用，作为来自Python 3.0的一个功能升级，但是，它不像Python 3.0中那样实施严格的文本/二进制区分。

让我们来快速看看。在Python 2.6中，`bytearray`对象可以通过调用`bytearray`内置函数来创建，并且可以用字符串来初始化：

Creation in 2.6: a mutable sequence of small (0..255) ints

```
>>> S = 'spam'
>>> C = bytearray(S)                                           # A back-port from 3.0 in 2.6
>>> C                                                         # b'..' == '..' in 2.6 (str)
bytearray(b'spam')
```

在Python 3.0中，需要一个编码名称和字节字符串，因为文本和二进制字符串不能混合，尽管字节字符串能够反映编码的Unicode文本：

Creation in 3.0: text/binary do not mix

```
>>> S = 'spam'
>>> C = bytearray(S)
TypeError: string argument without an encoding

>>> C = bytearray(S, 'latin1')                                # A content-specific type in 3.0
>>> C
```

```

bytearray(b'spam')

>>> B = b'spam'                                # b'..' != '..' in 3.0 (bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'spam')

```

创建之后，`bytearray`对象像`bytes`一样也是较小的整数序列，并且可以像列表一样修改，尽管它们需要一个整数而不是一个字符串进行索引赋值（如下的例子都是本节内容的延续，并且除非特别声明，都是在Python 3.0下运行，参见关于Python 2.6用法提示的注释）：

```

# Mutable, but must assign ints, not strings

>>> C[0]
115

>>> C[0] = 'x'                                # This and the next work in 2.6
TypeError: an integer is required

>>> C[0] = b'x'
TypeError: an integer is required

>>> C[0] = ord('x')
>>> C
bytearray(b'xspam')

>>> C[1] = b'Y'[0]
>>> C
bytearray(b'xYam')

```

处理`bytearray`对象借用了字符串和列表的方法，因为它们都是可修改的字节字符串。除了命名方法，`bytearray`中的`__iadd__`和`__setitem__`方法分别实现了`+=`原处连接和索引赋值：

```

# Methods overlap with both str and bytes, but also has list's mutable methods

>>> set(dir(b'abc')) - set(dir(bytearray(b'abc')))
{'__getnewargs__'}

>>> set(dir(bytearray(b'abc'))) - set(dir(b'abc'))
{'insert', '__alloc__', 'reverse', 'extend', '__delitem__', 'pop', '__setitem__',
 '__iadd__', 'remove', 'append', '__imul__'}

```

我们可以使用两种索引赋值来原处修改一个`bytearray`，正如已经看到的，并且类似列表的方法如下所示（要在Python 2.6中原处修改文本，可能需要使用`list(str)`和`''.join(list)`，将其转换为一个列表并转换回来）：

```

# Mutable method calls

>>> C

```

```

bytearray(b'xYam')

>>> C.append(b'LMN')
TypeError: an integer is required
# 2.6 requires string of size 1

>>> C.append(ord('L'))
>>> C
bytearray(b'xYamL')

>>> C.extend(b'MNO')
>>> C
bytearray(b'xYamLMNO')

```

所有常见的序列操作和字符串方法都在bytearrays上有效，正如我们所期待的那样（注意，就像bytes对象一样，其表达式和方法期待bytes参数，而不是str参数）：

```

# Sequence operations and string methods

>>> C + b'!#'
bytearray(b'xYamLMNO!#')

>>> C[0]
120

>>> C[1:]
bytearray(b'YamLMNO')

>>> len(C)
8

>>> C
bytearray(b'xYamLMNO')

>>> C.replace('xY', 'sp')
TypeError: Type str doesn't support the buffer API
# This works in 2.6

>>> C.replace(b'xY', b'sp')
bytearray(b'spamLMNO')

>>> C
bytearray(b'xYamLMNO')

>>> C * 4
bytearray(b'xYamLMNOxYamLMNOxYamLMNOxYamLMNO')

```

最后，概括起来，下面的例子展示了bytes和bytearray对象如何是int的序列，而str对象是字符的序列：

```

# Binary versus text

>>> B
b'spam'
# B is same as S in 2.6
>>> list(B)
[115, 112, 97, 109]

```



```
>>> C
bytearray(b'xYamLMNO')
>>> list(C)
[120, 89, 97, 109, 76, 77, 78, 79]

>>> S
'spam'
>>> list(S)
['s', 'p', 'a', 'm']
```

尽管Python 3.0中所有的三种字符串类型都可以包含字符值并且支持很多相同的操作，但我们总是应该：

- 对文本数据使用`str`；
- 对二进制数据使用`bytes`；
- 对想要原处修改的二进制数据使用`bytearray`。

像文件这样的相关工具，常常也是我们的选择，这是下一节将要介绍的主题。

使用文本文件和二进制文件

本节进一步介绍Python 3.0的字符串模型对本书前面所介绍的文件处理基础知识的影响。正如前面提到的，关键是用哪种模式打开一个文件，它决定了在脚本中将要使用哪种对象类型表示文件的内容。文本模式意味着`str`对象，二进制模式意味着`bytes`对象：

- 文本模式文件根据Unicode编码来解释文件内容，要么是平台的默认编码，要么是我们传递进的编码名。通过传递一个编码名来打开文件，我们可以强行进行Unicode文件的各种类型的转换。文本模型的文件也执行通用的行末转换：默认地，所有的行末形式映射为脚本中的一个单独的'\n'字符，而不管在什么平台上运行。正如前面所描述的，文本文件也负责阅读和写入在某些Unicode编码方案中存储文件开始处的字节顺序标记（Byte Order Mark, BOM）。
- 二进制模式文件不会返回原始的文件内容，而是作为表示字节值的整数的一个序列，没有编码或解码，也没有行末转换。

`open`的第二个参数是想要处理文本文件还是二进制文件，就像在Python 2.X中所做的一样，给这个字符串添加一个“b”表示二进制模式（例如，“rb”表示读取二进制数据文件）。默认的模式是“rt”，这等同于“r”，意味着文本输入（就像在Python 2.X中一样）。

然而，在Python 3.0中，`open`的这种模式参数也意味着文件内容表示的一个对象类型，而不管底层的平台是什么——文本文件返回一个`str`供读取，并且期待一个`str`以写入；但二进制文件返回一个`bytes`供读取，并且期待一个`bytes`（或`bytearray`）供写入。

文本文件基础

为了便于展示，让我们从基本的文件I/O开始。只要你打算处理基本的文本文件（例如，ASCII）并且不担心绕过平台默认的字符串编码，Python 3.0中文件的显示和处理和它们在Python 2.X中有很相似之处（由此可见，字符串是通用的）。例如，如下示例在Python 3.0中把一行文本写入一个文件并将其读取回来，和在Python 2.6中的处理是一样的（注意，Python 3.0中file不再是内置的名称，因此，这里使用它作为变量完全没有问题）：

```
C:\misc> c:\python30\python

# Basic text files (and strings) work the same as in 2.X

>>> file = open('temp', 'w')
>>> size = file.write('abc\n')           # Returns number of bytes written
>>> file.close()                         # Manual close to flush output buffer

>>> file = open('temp')                  # Default mode is "r" (== "rt"): text input
>>> text = file.read()
>>> text
'abc\n'
>>> print(text)
abc
```

Python 3.0中的文本和二进制模式

在Python 2.6中，文本文件和二进制文件之间没有主要区别——都是接受并返回作为str字符串的内容。唯一的主要区别是，在Windows下文本文件自动把\n行末字符和\r\n相互映射，而二进制文件不这么做（这里，为了简单起见，我们把操作连接到了行之中）：

```
C:\misc> c:\python26\python

>>> open('temp', 'w').write('abd\n')     # Write in text mode: adds \r
>>> open('temp', 'r').read()             # Read in text mode: drops \r
'abd\n'

>>> open('temp', 'rb').read()            # Read in binary mode: verbatim
'abd\r\n'

>>> open('temp', 'wb').write('abc\n')    # Write in binary mode
>>> open('temp', 'r').read()             # \n not expanded to \r\n
'abc\n'

>>> open('temp', 'rb').read()
'abc\n'
```

在Python 3.0中，情况稍微复杂一些，因为用于文本数据的str和用于二进制数据的bytes之间存在区别。为了说明这点，让我们写入一个**文本文件**并在Python中以两种模式来读取它。注意，我们需要为写入提供一个str，但是，根据打开模式，读取给我们一个str或bytes：

```

C:\misc> c:\python30\python

# Write and read a text file

>>> open('temp', 'w').write('abc\n')           # Text mode output, provide a str
4

>>> open('temp', 'r').read()                     # Text mode input, returns a str
'abc\n'

>>> open('temp', 'rb').read()                     # Binary mode input, returns a bytes
b'abc\r\n'

```

注意，在Windows上，文本模式的文件是在输出中如何把\n行末符号转换为\r\n的；在输入上，文本模式把\r\n转换回\n，但二进制模式不会这么做。这在Python 2.6中是一样的，并且，这也是我们希望对二进制数据所做的，尽管如果我们愿意的话，可以在Python 3.0中用额外的open参数控制这一行为。

现在，让我们再次做同样的事情，但是是对二进制文件来做。我们提供一个bytes以便在这个例子中写入，并且根据输入模式，我们仍然得到一个str或一个bytes：

```

# Write and read a binary file

>>> open('temp', 'wb').write(b'abc\n')           # Binary mode output, provide a bytes
4

>>> open('temp', 'r').read()                     # Text mode input, returns a str
'abc\n'

>>> open('temp', 'rb').read()                     # Binary mode input, returns a bytes
b'abc\n'

```

注意，在二进制模式输出中，\n行末字符没有扩展为\r\n——再一次说明，这是二进制数据想要的结果。即便我们要写入二进制文件中的数据本身真的是二进制的，类型需求和文件行为还是相同的。例如，在下面的例子中，"\x00"是二进制0字节并且不是一个可打印的字符：

```

# Write and read truly binary data

>>> open('temp', 'wb').write(b'a\x00c')           # Provide a bytes
3

>>> open('temp', 'r').read()                     # Receive a str
'a\x00c'

>>> open('temp', 'rb').read()                     # Receive a bytes
b'a\x00c'

```

二进制模式文件总是作为一个bytes对象返回内容，但是接受一个bytes或bytearray对象以供写入。既然bytearray基本上是bytes的一个可变的变体，自然就遵从这一方式。实际上，Python 3.0中的大多数API接受一个bytes，也允许一个bytearray：

```

# bytearray works too

>>> BA = bytearray(b'\x01\x02\x03')

>>> open('temp', 'wb').write(BA)
3

>>> open('temp', 'r').read()
'\x01\x02\x03'

>>> open('temp', 'rb').read()
b'\x01\x02\x03'

```

类型和内容错误匹配

注意，当遇到文件的时候，我们不能违反Python的str/bytes类型差异并侥幸成功。正如如下的例子所示，如果试图向一个文本文件写入一个bytes或者向二进制文件写入一个str，将会得到错误（这里缩写出了错信息）：

```

# Types are not flexible for file content

>>> open('temp', 'w').write('abc\n')           # Text mode makes and requires str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: can't write bytes to text stream

>>> open('temp', 'wb').write(b'abc\n')          # Binary mode makes and requires bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: can't write str to binary stream

```

这是有意义的：对于二进制模式，在文本编码之前，它是没有意义的。尽管往往可能通过编码str和解码bytes在类型之间转换，但正如本章前面所介绍的那样，我们通常会坚持对文本数据使用str或对二进制数据使用bytes。由于str和bytes操作集有很大程度的重合，所以对于大多数程序来说，做出选择并不是那么难（参见本章最后一个部分针对这种情况的基础示例所介绍的字符串工具）。

除了类型限制，在Python 3.0中，文件内容也有关系。文本模式的输入文件需要一个str而不是一个bytes用于内容，因此，在Python 3.0中，没有方法把真正的二进制数据写入一个文本模式文件中。根据编码规则，默认字符集以外的bytes有时候可以嵌入一个常规的字符串中，并且它们总是可以在二进制模式中写入。然而，由于Python 3.0中的文本模式输入文件必须能够针对每个Unicode编码来解码内容，因此，没有办法在文本模式中读取真正的二进制数据：

```

# Can't read truly binary data in text mode

>>> chr(0xFF)           # FF is a valid char, FE is not
'ÿ'

```

```

>>> chr(0xFE)
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1...

>>> open('temp', 'w').write(b'\xFF\xFE\xFD')      # Can't use arbitrary bytes!
TypeError: can't write bytes to text stream

>>> open('temp', 'w').write('\xFF\xFE\xFD')        # Can write if embeddable in str
3

>>> open('temp', 'wb').write(b'\xFF\xFE\xFD')      # Can also write in binary mode
3

>>> open('temp', 'rb').read()                      # Can always read as binary bytes
b'\xff\xfe\xfd'

>>> open('temp', 'r').read()                      # Can't read text unless decodable!
UnicodeEncodeError: 'charmap' codec can't encode characters in position 2-3: ...

```

最后一个错误源自于一个事实——Python 3.0中的所有文本文件实际都是Unicode文本文件，正如下一节所介绍的那样。

使用Unicode文件

到目前为止，我们已经读取和写入了基本的文本文件和二进制文件，但是，如何处理Unicode文件呢？事实证明，读取和写入存储在文件中的Unicode文本很容易，因为Python 3.0的open调用针对文本文件接受一个编码，在数据传输的时候，它自动为我们编码和解码。这允许我们处理用不同编码创建的Unicode文本，而不仅是平台默认编码的Unicode文本，并且以不同的编码存储以供转换。

在Python 3.0中读取和写入Unicode

实际上，我们有两种办法可以把字符串转换为不同的编码：用方法调用手动地转换和在文件输入输出上自动地转换。在本节中，我们将使用如下的Unicode字符串来说明这一点：

```

C:\misc> c:\python30\python
>>> S = 'A\xc4B\xe8C'                                # 5-character string, non-ASCII
>>> S
'AÄBèC'
>>> len(S)
5

```

手动编码

我们已经介绍过，总是可以根据目标编码名称把一个字符串转换为raw bytes：

```

# Encode manually with methods

>>> L = S.encode('latin-1')                            # 5 bytes when encoded as latin-1
>>> L

```

```

b'A\xc4B\xe8C'
>>> len(L)
5

>>> U = S.encode('utf-8')           # 7 bytes when encoded as utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7

```

文件输出编码

现在，要把我们的字符串以特定编码写入一个文本文件，我们可以直接把想要的编码名称传递给`open`，尽管我们可以先手动地编码并以二进制格式写入，但没有必要这么做：

```

# Encoding automatically when written

>>> open('latindata', 'w', encoding='latin-1').write(S)           # Write as latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(S)              # Write as utf-8
5

>>> open('latindata', 'rb').read()                                  # Read raw bytes
b'A\xc4B\xe8C'

>>> open('utf8data', 'rb').read()                                  # Different in files
b'A\xc3\x84B\xc3\xa8C'

```

文件输入编码

类似地，要读取任意的Unicode数据，我们直接把文件的编码类型名称传入`open`，并且，它自动根据raw bytes解码出字符串；我们也可以手动地读取raw byte并解码，但是，当读取数据块的时候（我们可能读取不完整的字符），这可能有些繁琐，并且也没有必要这么做：

```

# Decoding automatically when read

>>> open('latindata', 'r', encoding='latin-1').read()             # Decoded on input
'AÄBèC'
>>> open('utf8data', 'r', encoding='utf-8').read()                # Per encoding type
'AÄBèC'

>>> X = open('latindata', 'rb').read()                             # Manual decoding
>>> X.decode('latin-1')                                            # Not necessary
'AÄBèC'
>>> X = open('utf8data', 'rb').read()
>>> X.decode()                                                    # UTF-8 is default
'AÄBèC'

```

解码错误匹配

最后，别忘了，Python 3.0中的这些文件行为仅限于可以作为文本载入的内容。正如前

面的部分所介绍的，Python 3.0真的必须能够把文本文件中的数据解码为一个`str`字符串，根据默认的或传入的Unicode编码名称。例如，试图以文本模式打开一个真正的二进制数据文件，即便使用了正确的对象类型，也不可能在Python 3.0中有效。

```
>>> file = open('python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...

>>> file = open('python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00'
```

这些例子中的第一个可能不会在Python 2.X中失效（常规文件不能解码文本），即便它可能应该失效：读取一个文件可能会以字符串返回毁坏的数据，由于在文本模式中的自动行末转换（读取的时候，任何嵌入的`\r\n`字节都将在Windows下转换为`\n`）。在Python 2.6中，要把文件内容当做Unicode文本对待，我们需要使用特殊的工具而不是通用的内置函数`open`，稍后我们将介绍这些。那么，首先，让我们来看一个更重要的话题。

在Python 3.0中处理BOM

正如本章前面所介绍的，一些编码方式在文件的开始处存储了一个特殊的字节顺序标记（BOM）序列，来指定数据的大小尾方式或声明编码类型。如果编码名暗示了BOM的时候，Python在输入和将其写入输出的时候都会忽略该标记，但是有时候必须使用一个特定的编码名称来迫使显式地处理BOM。

例如，当我们把一个文本文件保存到Windows Notepad中的时候，可以在一个下拉列表中指定其编码类型——简单的ASCII文本、UTF-8或者小尾或大尾的UTF-16。例如，如果一个单行的、名为`spam.txt`的文本文件在Notepad中按照编码类型“ANSI”保存，它会编写为一个简单的ASCII文件而没有一个BOM。当这个文件在Python中以二进制模式读取的时候，我们可以看到存储在文件中的真正的bytes。当它作为文本读取的时候，Python默认会执行行尾转换，既然ASCII是UTF-8的一个子集（并且UTF-8是Python 3.0的默认编码），我们可以将其显式地解码为UTF-8文本：

```
c:\misc> C:\Python30\python                                     # File saved in Notepad
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
>>> open('spam.txt', 'rb').read()                                # ASCII (UTF-8) text file
b'spam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()                                  # Text mode translates line-end
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
```

```
'spam\nSPAM\n'
```

如果文件在Notepad中保存为“UTF-8”，预先使用一个3字节UTF-8 BOM序列，并且我们需要给出更多具体编码名称（“utf-8-sig”）来迫使Python跳过标记：

```
>>> open('spam.txt', 'rb').read()                # UTF-8 with 3-byte BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()
'i>>¿spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\uffffspam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

如果文件作为“Unicode大尾”存储在Notepad中，我们得到了文件中的UTF-16格式的数据，预先使用一个两字节的BOM序列——在Python中，编码名“utf-16”忽略BOM，因为它是暗示的（因为所有的UTF-16文件都有一个BOM），并且“utf-16-be”处理大尾格式但不会忽略BOM：

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00r\x00n\x00S\x00P\x00A\x00M\x00r\x00n'
>>> open('spam.txt', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1:...
>>> open('spam.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\uffffspam\nSPAM\n'
```

对于输出通常也是这样。当用Python代码写入一个Unicode文件，我们需要一个更加显式的编码名称来强迫UTF-8中带有BOM——“utf-8”不会写入（或忽略）BOM，但“utf-8-sig”会这么做：

```
>>> open('temp.txt', 'w', encoding='utf-8').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                # No BOM
b'spam\r\nSPAM\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                # Wrote BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'

>>> open('temp.txt', 'r').read()
'i>>¿spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()    # Keeps BOM
'\uffffspam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read() # Skips BOM
'spam\nSPAM\n'
```

注意，尽管“utf-8”没有抛弃BOM，但不带BOM的数据可以用“utf-8”和“utf-8-sig”

读取——如果你不确定一个文件中是否有BOM，使用后者进行输入（在机场安全检测线上，不要大声读出这一段）：

```
>>> open('temp.txt', 'w').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                # Data without BOM
b'spam\r\nSPAM\r\n'
>>> open('temp.txt', 'r').read()                  # Any utf-8 works
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

最后，对于编码名“utf-16”，BOM自动处理：在输出上，数据以平台本地的大小尾方式写入，并且，BOM总是会写的；在输入上，数据根据每个BOM解码，并且BOM总是会去除掉。更具体的UTF-16编码名称可以指定不同的大小尾，尽管在某些情况下如果需要或显示BOM的话，我们必须自己手动地编写和略过BOM：

```
>>> sys.byteorder
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()
b'\xff\xfe\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n\x00'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'

>>> open('temp.txt', 'w', encoding='utf-16-be').write('\uffffspam\nSPAM\n')
11
>>> open('temp.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\uffffspam\nSPAM\n'
```

更具体的UTF-16编码名称对于缺乏BOM的文件都工作得很好，尽管“utf-16”在输入时需要一个BOM以便确定字节顺序：

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('SPAM')
4
>>> open('temp.txt', 'rb').read()                # OK if BOM not present or expected
b'S\x00P\x00A\x00M\x00'
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'SPAM'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

自己尝试实验这些编码，或者查看Python的库手册，以了解关于BOM的更多细节。

Python 2.6中的Unicode文件

前面的讨论适用于Python 3.0的字符串类型和文件。我们可以针对Python 2.6中的Unicode实现类似的效果，但是，接口是不同的。如果用unicode替代str并且用codecs.open来打开，在Python 2.6中的结果基本相同：

```
C:\misc> c:\python26\python
>>> S = u'A\xc4B\xe8C'
>>> print S
AÃBèC
>>> len(S)
5
>>> S.encode('latin-1')
'A\xc4B\xe8C'
>>> S.encode('utf-8')
'A\xc3\x84B\xc3\xa8C'

>>> import codecs
>>> codecs.open('latindata', 'w', encoding='latin-1').write(S)
>>> codecs.open('utfdata', 'w', encoding='utf-8').write(S)

>>> open('latindata', 'rb').read()
'A\xc4B\xe8C'
>>> open('utfdata', 'rb').read()
'A\xc3\x84B\xc3\xa8C'

>>> codecs.open('latindata', 'r', encoding='latin-1').read()
u'A\xc4B\xe8C'
>>> codecs.open('utfdata', 'r', encoding='utf-8').read()
u'A\xc4B\xe8C'
```

Python 3.0中其他字符串工具的变化

Python标准库中其他一些常用的字符串处理工具，也由于新的str/bytes类型区分而进行了修改。我们无法在这本介绍核心语言的图书里覆盖所有这些面向应用的工具的细节，但是，为了结束本章的讨论，这里快速看一下受到影响的4种主要的工具：`re`模式匹配模块、`struct`二进制数据模块、`pickle`对象序列化模块和用于解析XML文本的`xml`包。

re模式匹配模块

Python的`re`模式匹配模块提供的文本处理，比简单的方法调用所提供的查找、分隔、替换等更加通用。借助`re`，设定搜索和分隔目标的字符串可以用更通用的模式来描述，而不是用绝对文本描述。这个模块已经泛化为可以用于Python 3.0中的任何字符串类型的对象——`str`、`bytes`和 `bytearray`，并且返回同样类型的结果子字符串作为目标字符串。

如下是其在Python 3.0中的引用，从一行文本提取子字符串。在模式字符串中，`(.*)`表示任何字符`(.)`、0或多次`(*)`、，作为一个匹配的子字符串单独保存`()`。在成功匹配

之后，根据包含在圆括号中的模式部分而匹配的字符串部分就可以使用，通过`group`或`groups`方法：

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Bugger all down here on earth!'           # Line of text
>>> B = b'Bugger all down here on earth!'          # Usually from a file

>>> re.match('(.*) down (.*) on (.*)', S).groups() # Match line to pattern
('Bugger all', 'here', 'earth!')                 # Matched substrings

>>> re.match(b'(.*) down (.*) on (.*)', B).groups() # bytes substrings
(b'Bugger all', b'here', b'earth!')
```

Python 2.6中的结果是类似的，但是，`unicode`类型用于非ASCII文本，并且`str`处理8位的和二进制文本：

```
C:\misc> c:\python26\python
>>> import re
>>> S = 'Bugger all down here on earth!'           # Simple text and binary
>>> U = u'Bugger all down here on earth!'          # Unicode text

>>> re.match('(.*) down (.*) on (.*)', S).groups()
('Bugger all', 'here', 'earth!')

>>> re.match('(.*) down (.*) on (.*)', U).groups()
(u'Bugger all', u'here', u'earth!')
```

由于`bytes`和`str`支持基本相同的操作集，所以这种类型差异大部分很明显。但是，注意，像在其他API中一样，我们不能在Python 3.0调用的参数中混合`str`和`bytes`类型（尽管如果你不想在二进制数据上进行模式匹配，可能不需要关心这一点）：

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Bugger all down here on earth!'
>>> B = b'Bugger all down here on earth!'

>>> re.match('(.*) down (.*) on (.*)', B).groups()
TypeError: can't use a string pattern on a bytes-like object

>>> re.match(b'(.*) down (.*) on (.*)', S).groups()
TypeError: can't use a bytes pattern on a string-like object

>>> re.match(b'(.*) down (.*) on (.*)', bytearray(B)).groups()
(bytearray(b'Bugger all'), bytearray(b'here'), bytearray(b'earth!'))

>>> re.match('(.*) down (.*) on (.*)', bytearray(B)).groups()
TypeError: can't use a string pattern on a bytes-like object
```

Struct二进制数据模块

Python的`struct`模块，用来从字符串创建和提取打包的二进制数据，它在Python 3.0中也

像在Python 2.X中一样工作，但是，打包的数据只是作为bytes和bytearray对象显示，而不是str对象（这是有意义的，因为它本来就是要处理二进制数据的，而不是处理任意编码的文本）。

下面是在Python的两个版本中的应用，它根据二进制类型声明把3个对象打包到一个字符串中（它们创建了一个4字节的整数、一个4字节的字符串和一个两字节的整数）：

```
C:\misc> c:\python30\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'spam', 8)          # bytes in 3.0 (8-bit string)
b'\x00\x00\x00\x07spam\x00\x08'

C:\misc> c:\python26\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'spam', 8)          # str in 2.6 (8-bit string)
'\x00\x00\x00\x07spam\x00\x08'
```

由于bytes有一个几乎近似于Python 3.0和Python 2.6中的str的接口，然而，大多数程序员可能不需要关心——这一修改与大多数已有的代码无关，特别是由于读取一个二进制文件会自动创建一个bytes。尽管下面例子中最后的测试在一个类型错误匹配上失效，但大多数脚本将从一个文件读取二进制数据，而不是将其创建为一个字符串：

```
C:\misc> c:\python30\python
>>> import struct
>>> B = struct.pack('>i4sh', 7, 'spam', 8)
>>> B
b'\x00\x00\x00\x07spam\x00\x08'

>>> vals = struct.unpack('>i4sh', B)
>>> vals
(7, b'spam', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: 'str' does not have the buffer interface
```

除了bytes的新语法之外，创建和读取二进制文件在Python 3.0和Python 2.X中几乎同样地工作。像这里的代码，是程序员将会注意到bytes对象类型的主要地方：

```
C:\misc> c:\python30\python

# Write values to a packed binary file

>>> F = open('data.bin', 'wb')          # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'spam', 8)  # Create packed binary data
>>> data                                  # bytes in 3.0, not str
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                        # Write to the file
10
>>> F.close()
```

Read values from a packed binary file

```
>>> F = open('data.bin', 'rb')           # Open binary input file
>>> data = F.read()                     # Read bytes
>>> data
b'\x00\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data) # Extract packed binary data
>>> values                               # Back to Python objects
(7, b'spam', 8)
```

一旦像这样把二进制数据提取到Python对象中，如果必须做的话，可以更深入地探索二进制的世界——可以索引和分片字符串以得到单个bytes的值，可以用位操作符从整数提取单个的位，等等（参见本书前面的内容，了解这里应用的操作的更多细节）：

```
>>> values                               # Result of struct.unpack
(7, b'spam', 8)

# Accessing bits of parsed integers

>>> bin(values[0])                       # Can get to bits in ints
'0b111'
>>> values[0] & 0x01                     # Test first (lowest) bit in int
1
>>> values[0] | 0b1010                   # Bitwise or: turn bits on
15
>>> bin(values[0] | 0b1010)               # 15 decimal is 1111 binary
'0b1111'
>>> bin(values[0] ^ 0b1010)               # Bitwise xor: off if both true
'0b1101'
>>> bool(values[0] & 0b100)               # Test if bit 3 is on
True
>>> bool(values[0] & 0b1000)              # Test if bit 4 is set
False
```

由于解析的bytes字符串是小整数的序列，所以我们可以对其单个的bytes做类似的处理：

Accessing bytes of parsed strings and bits within them

```
>>> values[1]
b'spam'
>>> values[1][0]                         # bytes string: sequence of ints
115
>>> values[1][1:]                         # Prints as ASCII characters
b'pam'
>>> bin(values[1][0])                     # Can get to bits of bytes in strings
'0b1110011'
>>> bin(values[1][0] | 0b1100)            # Turn bits on
'0b1111111'
>>> values[1][0] | 0b1100
127
```

当然，大多数Python程序员不会处理二进制位；Python拥有更高级别的对象类型，例如

列表和字典，对于在Python脚本中表示信息，它们通常是一种更好的选择。然而，如果你必须使用或产生供C程序、网络库或其他接口所使用的低层级数据，Python也有辅助的工具。

pickle对象序列化模块

我们在本书第9章和第30章简单介绍了pickle模块。在第27章，我们也使用了shelve模块，它实际上使用了pickle。这里为了完整起见，别忘了pickle模块的Python 3.0版本总是创建一个bytes对象，而不管默认的或传入的“协议”（数据格式化层级）。我们通过使用该模块的dumps调用来返回一个对象的pickle字符串，从而查看这一点：

```
C:\misc> C:\Python30\python
>>> import pickle                                # dumps() returns pickle string

>>> pickle.dumps([1, 2, 3])                       # Python 3.0 default protocol=3=binary
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

```
>>> pickle.dumps([1, 2, 3], protocol=0)           # ASCII protocol 0, but still bytes!
b'(lp0\nL1L\naL2L\naL3L\na.'
```

这意味着，用来存储pickle化的对象的文件必须总是在Python 3.0中以二进制模式打开，因为文本文件使用str字符串来表示数据，而不是bytes——dump调用直接试图把pickle字符串写入一个打开的输出文件中：

```
>>> pickle.dump([1, 2, 3], open('temp', 'w'))      # Text files fail on bytes!
TypeError: can't write bytes to text stream        # Despite protocol value

>>> pickle.dump([1, 2, 3], open('temp', 'w'), protocol=0)
TypeError: can't write bytes to text stream

>>> pickle.dump([1, 2, 3], open('temp', 'wb'))      # Always use binary in 3.0

>>> open('temp', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\u20ac' in ...
```

由于pickle数据不是可解码的Unicode文本，所以对于输出也是如此——在Python 3.0中的正确用法总是要求以二进制模式写入和读取pickle数据：

```
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))
>>> pickle.load(open('temp', 'rb'))
[1, 2, 3]
>>> open('temp', 'rb').read()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

在Python 2.6（和更早的版本）中，我们可以使用文本模式文件来获得pickle化的数据，只要协议是0层级（Python 2.6中默认的协议）并且我们使用文本模式一致地转换行尾：

```

C:\misc> c:\python26\python
>>> import pickle
>>> pickle.dumps([1, 2, 3])                                # Python 2.6 default=0=ASCII
'(\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00)'

>>> pickle.dumps([1, 2, 3], protocol=1)
'q\x00(K\x01K\x02K\x03e.'

>>> pickle.dump([1, 2, 3], open('temp', 'w'))              # Text mode works in 2.6
>>> pickle.load(open('temp'))
[1, 2, 3]
>>> open('temp').read()
'(\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00)'

```

如果你关注版本独立，或者不想要关心协议或者其特定版本的默认值，总是对pickle化的数据使用二进制模式文件，如下的做法在Python 3.0和Python 2.6中都有效：

```

>>> import pickle
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))              # Version neutral
>>> pickle.load(open('temp', 'rb'))                          # And required in 3.0
[1, 2, 3]

```

由于几乎所有的程序都允许Python自动地pickle或unpickle对象，并且不会处理pickle化的数据自身的内容，总是使用二进制文件模式的要求是Python 3的新pickle化模式中唯一显著的不兼容。参阅参考书籍或Python的手册，来了解关于对象pickle化的更多细节。

XML解析工具

XML是一种基于标签的语言，用于定义结构化的信息，通常用来定义通过Web传输的文档和数据。尽管可以使用基本的字符串方法或re模式模块从XML文本提取一些信息，但XML的嵌套式的结构和任意的属性文本使得全面解析更为精确。

由于XML是如此广泛使用的格式，Python自身附带一个完整的XML解析工具包，所以它支持SAX和DOM解析模式，此外，还有一个名为`ElementTree`的包——这是特定于Python的专用于解析和构造XML的一个API。除了基本的解析，开源领域还提供了额外的XML工具，例如XPath、Xquery、XSLT，等等。

XML根据定义以Unicode形式表示文本，以支持国际化。尽管大多数Python的XML解析工具总是返回Unicode字符串，但在Python 3.0中，它们的结果必须从Python 2.X的unicode类型转变为Python 3.0通用的str字符串类型——这是有意义的，因为Python 3.0的str字符串是Unicode，不管编码是ASCII或是其他的。

我们无法在这里介绍更多细节，但是，示例对于了解这方面内容有帮助，假设我们有一个简单的XML文本文件`mybooks.xml`：

```

<books>
  <date>2009</date>
  <title>Learning Python</title>
  <title>Programming Python</title>
  <title>Python Pocket Reference</title>
  <publisher>O'Reilly Media</publisher>
</books>

```

并且我们想要运行一个脚本来提取并显示所有嵌套的title标记的内容，如下所示：

```

Learning Python
Programming Python
Python Pocket Reference

```

至少有4种基本的方法来做这点（不包括像XPath这样更高级的工具）。首先，我们可以在文件的文本上运行基本的**模式匹配**，尽管如果文本不可预期的话，这种方法可能不精确。采用这种方法的时候，我们前面介绍的re模块可以完成这项工作，它的match方法从字符串开始查找一个匹配，search方法向前查找一个匹配，这里使用的findall方法找出字符串中和模式匹配的所有地方（返回的结果是，和括号括起来的模式组或多个这样的组的元组对应的匹配子字符串的列表）：

```

# File patternparse.py

import re
text = open('mybooks.xml').read()
found = re.findall('<title>(.*?)</title>', text)
for title in found: print(title)

```

其次，为了更加稳健，我们可以用标准库的DOM解析支持来执行完整的XML解析。DOM把XML文本解析为一个对象树，并且提供一个接口在树中导航并提取标签属性和值；这个接口是一个正式规范，独立于Python：

```

# File domparse.py

from xml.dom.minidom import parse, Node
xmldata = parse('mybooks.xml')
for node1 in xmldata.getElementsByTagName('title'):
    for node2 in node1.childNodes:
        if node2.nodeType == Node.TEXT_NODE:
            print(node2.data)

```

作为第三种方式，Python的标准库支持SAX解析XML。在SAX模式下，类的方法接收回调作为一个解析过程，并且使用状态信息来记录它们在文档中的位置并收集其数据：

```

# File saxparse.py

import xml.sax.handler
class BookHandler(xml.sax.handler.ContentHandler):
    def __init__(self):

```



```

        self.inTitle = False
    def startElement(self, name, attributes):
        if name == 'title':
            self.inTitle = True
    def characters(self, data):
        if self.inTitle:
            print(data)
    def endElement(self, name):
        if name == 'title':
            self.inTitle = False

import xml.sax
parser = xml.sax.make_parser()
handler = BookHandler()
parser.setContentHandler(handler)
parser.parse('mybooks.xml')

```

最后，可以使用标准库的etree包中的`ElementTree`系统，它往往用来实现和XML DOM解析器相同的效果，但是，代码更少。这是一种特定于Python的方式，既解析XML文本也可以生成XML文本。在解析之后，其API可以访问文档的组件：

```

# File etreeparse.py

from xml.etree.ElementTree import parse
tree = parse('mybooks.xml')
for E in tree.findall('title'):
    print(E.text)

```

在Python 2.6或Python 3.0中运行的时候，所有这4段脚本都显示相同的结果：

```

C:\misc> c:\python26\python domparse.py
Learning Python
Programming Python
Python Pocket Reference

C:\misc> c:\python30\python domparse.py
Learning Python
Programming Python
Python Pocket Reference

```

然而，从技术上，在Python 2.6中，这些脚本中的某些产生`unicode`字符串对象，而在Python 3.0中都产生`str`字符串，因为该类型包含了Unicode文本（不管是ASCII或其他）：

```

C:\misc> c:\python30\python
>>> from xml.dom.minidom import parse, Node
>>> xmldata = parse('mybooks.xml')
>>> for node in xmldata.getElementsByTagName('title'):
...     for node2 in node.childNodes:
...         if node2.nodeType == Node.TEXT_NODE:
...             node2.data
...

```

```
'Learning Python'
'Programming Python'
'Python Pocket Reference'

C:\misc> c:\python26\python
>>> ...same code...
...
u'Learning Python'
u'Programming Python'
u'Python Pocket Reference'
```

必须处理XML解析的程序，导致了需要针对Python 3.0中的不同对象类型考虑不一般的方式。再次，尽管所有的字符串在Python 2.6和Python 3.0中都有几乎相同的接口，但大多数脚本不会受到这样修改的影响。可用于Python 2.6的`unicode`工具，通常也可用于Python 3.0中的`str`。

遗憾的是，继续深入讨论XML的细节超越了本书的范围。如果你对于文本或XML解析感兴趣，在后续基于应用程序的图书*Programming Python*中介绍了其更多细节。要了解关于`re`、`struct`、`pickle`和XML工具的更多知识，可在Web上搜索，阅读前面提到的书籍或其他书籍，或参阅Python的标准库手册。

本章小结

本章介绍了Python 3.0和Python 2.6中可以用来处理Unicode文本和二进制数据的高级字符串类型。正如我们所介绍的，很多程序员使用ASCII文本，并且使用基本的字符串类型及其操作就够了。对于一些较高级的应用程序，Python的字符串模型全面支持宽字符Unicode文本（通过Python 3.0中的常规字符串类型和Python 2.6中的一种特殊类型）以及面向字节的数据（在Python 3.0中有一个`bytes`类型表示，在Python 2.6中用常规字符串表示）。

此外，我们学习了Python 3.0中的文件对象如何自动编码和解码Unicode文本以及针对二进制模式文件处理字节字符串。最后，我们简单地介绍了Python库中的一些文本和二进制数据工具，并且示例了它们在Python 3.0中的应用。

下一章中，我们将把关注点转移到工具构建器的话题，看看通过插入有趣的自动运行代码来管理对对象属性访问的方法。在继续学习之前，让我们来进行一组测试，复习在本章中学习过的内容。

本章习题

1. Python 3.0中字符串对象类型的名称和角色是什么？

2. Python 2.6中字符串对象类型的名称和角色是什么?
3. Python 2.6和Python 3.0中的字符串类型是如何对应的?
4. Python 3.0的字符串类型在操作上有何不同?
5. 在Python 3.0中, 如何把非ASCII Unicode字符编写到字符串中?
6. Python 3.0中的文本模式文件和二进制模式文件之间的主要区别是什么?
7. 如何读取一个Unicode文本文件, 它包含按照一种不同于平台默认编码进行编码的文本。
8. 如何以一种特定的编码格式创建一个Unicode文本文件?
9. 为什么把ASCII文本看做是一种Unicode文本?
10. Python 3.0的字符串类型修改对你的代码有多大的影响?

习题解答

1. Python 3.0有3种字符串类型: `str` (用于Unicode文本, 包括ASCII)、`bytes` (用于带有绝对字节值的二进制数据) 和 `bytearray` (`bytes`的一种可变的形式)。`str` 类型通常表示存储在文本文件中的内容, 其他的两种形式通常表示存储在二进制文件中的内容。
2. Python 2.6有两种主要的字符串类型: `str` (用于8位文本和二进制数据) 以及 `unicode` (用于宽字符文本)。`str` 类型用于文本和二进制文件内容, `unicode` 用于通常比8位更复杂的文本文件内容。Python 2.6 (但不包括更早的版本) 也有Python 3.0的 `bytearray` 类型, 但它主要是一种向后兼容, 而且并没有表现出Python 3.0中所表现出来的鲜明的文本/二进制区别。
3. 从Python 2.6到Python 3.0的字符串类型对应并不是直接的, 因为Python 2.6的 `str` 等同于Python 3.0中的 `str` 和 `bytes`, 并且Python 3.0中的 `str` 等同于Python 2.6中的 `str` 和 `unicode`。Python 3.0中 `bytearray` 的可变性是唯一的。
4. Python 3.0的字符串类型共享了几乎所有相同的操作: 方法调用、序列操作, 甚至像模式匹配这样以相同方式工作的更大工具。另一方面, 只有 `str` 支持字符串格式操作, 并且 `bytearray` 有一组额外的操作来执行原处修改。`Str` 和 `bytes` 类型也分别拥有编码和解码文本的方法。
5. 非ASCII Unicode字符可以以十六进制转移 (`\xNN`) 和Unicode转义 (`\uNNNN`, `\UNNNNNNNN`) 编写到一个字符串中。在某些键盘上, 一些非ASCII字符——例如, 某些Latin-1字符, 也可以直接录入。

6. 在Python 3.0中，文本模式文件假设其文件内容是Unicode文本（即便它是ASCII），并且当读取的时候自动解码，写入的时候自动编码。对于二进制模式的文件，bytes和文件之间不经修改地转换。文本模式文件的内容通常在脚本中表示为str对象，并且二进制文件的内容表示为bytes（或bytearray）对象。文本模式文件也针对特定编码类型处理bytearray，并且输入和输出时自动在行末序列与单个\n之间转换，除非显式地关闭这一功能。二进制模式的文件不会执行任何这样的步骤。
7. 要读取和平台默认编码方式不同的方式编码的文件，直接把文件编码名传递给Python 3.0的内置函数open（在Python 2.6中是codecs.open()）；当从文件读取数据的时候，数据将针对每种特定编码来解码。我们也可以在二进制模式中读取，并且通过给定一个编码名来手动地把字节解码成一个字符串，但是，这涉及额外的工作，并且对于多字节字符更容易出错（可能偶尔要读取字节序列的一部分）。
8. 要以特定编码格式创建一个Unicode文本文件，把想要的编码名称传递给Python 3.0中的open（在Python 2.6中是codecs.open()）。当字符串写入文件中的时候，将会按照每个想要的编码来进行编码。也可以手动把一个字符串编码为字节，并在二进制模式下将其写入，但是，这通常需要额外的工作。
9. ASCII文本看作是一种Unicode文本，因为其7位范围值只是大多数Unicode编码的一个子集。例如，有效的ASCII文本也是有效的Latin-1文本（Latin-1只是把一个8位字节中其余可能的值分配给额外的字符），并且是有效的UTF-8文本（UTF-8为表示更多的字符定义了一个变量-字节方案，但是ASCII字符仍然用同样的代码表示，即单个的字节）。
10. Python 3.0的字符串类型修改的影响，取决于你所使用的字符串的类型。对于那些使用简单ASCII文本的脚本，可能根本没有影响：在此情况下，str字符串类型在Python 2.6和Python 3.0中的用法相同。此外，尽管标准库中像re、struct、pickle和xml这样字符串相关的工具可能在Python 3.0和Python 2.6中技术上的用法不同，但这一修改很大程度上与大多数程序不相关，因为Python 3.0的str和bytes以及Python 2.6的str都支持几乎相同的接口。如果你处理Unicode数据，只需要直接把Python 2.6的工具集unicode和codecs.open()转换为Python 3.0的str和open。如果你处理二进制数据文件，将需要处理作为bytes对象的内容。由于它们与Python 2.6字符串具有相似的接口，所以影响再次变得很小。

管理属性

本章将展开介绍前面所提到的**属性拦截**技术，并且还要介绍另一种技术，将它们应用到一些较大的示例中。就像本书中这一部分的其他各章一样，本章也作为高级话题并供读者选择阅读，因为大多数高级程序员不需要关心这里讨论的内容——他们可以获取和设置对象的属性，而不必关心属性的实现。然而，特别是对工具构建者来说，管理属性的访问可能是灵活的API的一个重要部分。

为什么管理属性

对象的属性是大多数Python程序的中心——它们是我们经常存储供脚本处理的相关实体信息的地方。通常，属性只是对象的名称，例如，一个人的name属性，可能是一个简单的字符串，通过基本的属性语法来获取和设置：

```
person.name           # Fetch attribute value  
person.name = value   # Change attribute value
```

在大多数情况下，属性位于对象自身之中，或者继承自对象所派生自的一个类。基本的模式对于我们编写职业生涯中的大多数Python程序来说已经足够了。

然而，有的时候需要更多的灵活性。假设你编写了一个程序来直接使用一个name属性，但是随后需要修改——例如，在以某种方式设置或修改名称的时候，需要进行逻辑验证。编写一个方法来管理对属性值的访问（valid和transform在这里是抽象的），这是很直接的：

```
class Person:  
    def getName(self):  
        if not valid():
```

```

        raise TypeError('cannot fetch name')
    else:
        return self.name.transform()
    def setName(self, value):
        if not valid(value):
            raise TypeError('cannot change name')
        else:
            self.name = transform(value)

person = Person()
person.getName()
person.setName('value')

```

然而，这需要在整个程序中用到了名称的所有地方都进行修改，这可能不是个小任务。此外，这种方法需要程序知道如何导出值：是作为简单的名称或是作为调用的方法。如果从一开始就使用基于方法的接口来访问数据，客户端不会受到修改的影响；如果没有这么做，它们可能就会变成问题。

这个问题可能比我们想象的要更常见。例如，像电子表格这样的程序中一个单元格的值，可能作为一个简单的离散值开始其存在，但是，随后要变为一个任意的计算。由于一个对象的接口应该足够灵活以支持未来这样的修改，而不会破坏已有的代码，因此以后再切换到方法就不是理想的做法了。

插入在属性访问时运行的代码

一个更好的解决方案是，如果需要的话，在属性访问时自动运行代码。在本书的各种不同位置，我们都遇到过这样的Python工具，当获取属性值以及在存储属性值对其进行验证或修改的时候，这样的工具允许脚本动态地计算属性值。在本章中，我们将展开介绍那些已经遇到过的工具，探讨其他可用的工具，并且学习此领域的一些较大的使用示例。特别强调，本章包括以下内容：

- `__getattr__`和`__setattr__`方法，把未定义的属性获取和所有的属性赋值指向通用的处理器方法。
- `__getattribute__`方法，把所有属性获取都指向Python 2.6的新式类和Python 3.0的所有类中的一个泛型处理器方法。
- `property`内置函数，把特定属性访问定位到`get`和`set`处理器函数，也叫做特性（Property）。
- 描述符协议，把特定属性访问定位到具有任意`get`和`set`处理器方法的类的实例。

这里的第一个和第三个方法在第六部分都简单介绍过了；其他两个是这里要介绍的新主题。

正如我们将要看到的，所有4种技术在某种程度上具有同样的目标，并且通常可能对于给定的问题使用任何一种技术来编写代码。然而它们确实存在某些重要的不同。例如，这里列出的最后两种技术适用于特定属性，而前两种则足够通用，可以用于那些必须把任意属性指向包装的对象的、基于委托的类。我们将会看到，所有4种方法在复杂性和优雅性上也都有所不同，在使用中，我们必须通过实际应用来自行判断。

本章除了学习本节列出的4种属性拦截技术背后的细节，还给出了机会来探究比我们在本书前面任何地方所见过的程序更大的程序。例如，最后的CardHolder案例，可以作为使用一个较大的类的自学示例。在下一章中，我们还将使用这里介绍的某些技术来编写装饰器，因此，在继续学习之前，确保对这里介绍的技术至少有个一般性的理解。

特性

特性协议允许我们把一个特定属性的get和set操作指向我们所提供的函数或方法，使得我们能够插入在属性访问的时候自动运行的代码，拦截属性删除，并且如果愿意的话，还可为属性提供文档。

通过property内置函数来创建特性并将其分配给类属性，就像方法函数一样。同样，可以通过子类和实例继承属性，就像任何其他类属性一样。它们的访问拦截功能通过self实例参数提供，该参数确保了在主体实例上访问状态信息和类属性是可行的。

一个特性管理一个单独的、特定的属性；尽管它不能广泛地捕获所有的属性访问，它允许我们控制访问和赋值操作，并且允许我们自由地把一个属性从简单的数据改变为一个计算，而不会影响已有的代码。正如你将看到的，特性和描述符有很大的关系，它们基本上是描述符的一种受限制的形式。

基础知识

可以通过把一个内置函数的结果赋给一个类属性来创建一个特性：

```
attribute = property(fget, fset, fdel, doc)
```

这个内置函数的参数都不是必需的，并且如果没有传递参数的话，所有都取默认值None。这样的操作是不受支持的，并且尝试使用默认值将会引发一个异常。当使用它们的时候，我们向fget传递一个函数来拦截属性访问，给fset传递一个函数进行赋值，并且给fdel传递一个函数进行属性删除；doc参数接收该属性的一个文档字符串，如果想要的话（否则，该特性会赋值fget的文档字符串，如果提供了fget的文档字符串的话，其默认值为None）。fget返回计算过的属性值，并且fset和fdel不返回什么（确实是None）。

这个内置的函数调用返回一个特性对象，我们将它赋给了在类的作用域中要管理的属性的名称，正是在类的作用域中每个实例都继承了类。

第一个例子

为了说明如何把这些转换成有用的代码，如下的类使用一个特性来记录对一个名为`name`的属性的访问，实际存储的数据名为`_name`，以便不会和特性搞混了：

```
class Person:                                # Use (object) in 2.6
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('fetch...')
        return self._name
    def setName(self, value):
        print('change...')
        self._name = value
    def delName(self):
        print('remove...')
        del self._name
    name = property(getName, setName, delName, "name property docs")

bob = Person('Bob Smith')                    # bob has a managed attribute
print(bob.name)                             # Runs getName
bob.name = 'Robert Smith'                   # Runs setName
print(bob.name)
del bob.name                                # Runs delName

print('-'*20)
sue = Person('Sue Jones')                   # sue inherits property too
print(sue.name)
print(Person.name.__doc__)                  # Or help(Person.name)
```

Python 2.6和Python 3.0中都可以使用特性，但是，它们要求在Python 2.6中派生一个新式对象，才能使赋值正确地工作——为了在Python 2.6中运行代码，这里把对象添加为一个超类（我们在Python 3.0中也可以使用超类，但是，这是暗含的，并且不是必需的）。

这个特定的特性所做的事情并不多——它只是拦截并跟踪了一个属性，这里将它作为展示协议的一个例子。当这段代码运行的时候，两个实例继承了该特性，就好像它们是附加到其类的另外两个属性一样。然而，捕获了它们的属性访问：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
-----
fetch...
Sue Jones
```



```
name property docs
```

就像所有的类属性一样，实例和较低的子类都**继承**特性。如果我们把例子修改为如下所示：

```
class Super:
    ...the original Person class code...
    name = property(getName, setName, delName, 'name property docs')

class Person(Super):
    pass                                     # Properties are inherited

bob = Person('Bob Smith')
...rest unchanged...
```

输出是同样的。Person子类从Super继承了name特性，并且bob实例从Person获取了它。关于继承，特性的方式和常规方法是一样的；由于它们能够访问self实例参数，所以它们可以访问像方法这样的实例状态信息，就像下面小节所介绍的一样。

计算的属性

前面小节的例子简单地跟踪了属性访问。然而，通常特性做的更多——例如，当获取属性的时候，动态地计算属性的值。下面的例子展示了这一点：

```
class PropSquare:
    def __init__(self, start):
        self.value = start
    def getX(self):                                # On attr fetch
        return self.value ** 2
    def setX(self, value):                         # On attr assign
        self.value = value
    X = property(getX, setX)                       # No delete or docs

P = PropSquare(3)                                # 2 instances of class with property
Q = PropSquare(32)                               # Each has different state information

print(P.X)                                       # 3 ** 2
P.X = 4
print(P.X)                                       # 4 ** 2
print(Q.X)                                       # 32 ** 2
```

这个类定义了一个X属性，并且将其当作静态数据一样访问，但实际运行的代码在获取该属性的时候计算了它的值。这种效果很像是一个隐式方法调用。当代码运行的时候，值作为状态信息存储在实例中，但是，每次我们通过管理的属性获取它的时候，它的值都会自动平方：

```
9
16
```

注意，我们已经创建了两个不同的实例——因为特性方法自动地接收一个`self`参数，所以它们都访问了存储在实例中的状态信息。在我们的例子中，这意味着获取会计算主体实例的数据的平方。

使用装饰器编写特性

尽管我们直到下一章才会介绍额外细节，但我们更早地在第31章就引入了函数装饰器的基本概念。回忆一下，函数装饰器的语法是：

```
@decorator
def func(args): ...
```

Python会自动将其翻译成对等形式，把函数名重新绑定到可调用的`decorator`的返回结果上：

```
def func(args): ...
func = decorator(func)
```

由于这一映射，证实了内置函数`property`可以充当一个装饰器，来定义一个函数，当获取一个属性的时候自动运行该函数：

```
class Person:
    @property
    def name(self): ...           # Rebinds: name = property(name)
```

运行的时候，装饰的方法自动传递给`property`内置函数的第一个参数。这其实只是创建一个特性并手动绑定属性名的一种替代语法：

```
class Person:
    def name(self): ...
    name = property(name)
```

对于Python 2.6，`property`对象也有`getter`、`setter`和`deleter`方法，这些方法指定相应的特性访问器方法赋值并且返回特性自身的一个副本。我们也可以使用这些方法，通过装饰常规方法来指定特性的组成部分，尽管`getter`部分通常由创建特性自身的行为自动填充：

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):           # name = property(name)
        "name property docs"
        print('fetch...')
```

```

        return self._name

    @name.setter
    def name(self, value):
        print('change...')
        self._name = value
        # name = name.setter(name)

    @name.deleter
    def name(self):
        print('remove...')
        del self._name
        # name = name.deleter(name)

bob = Person('Bob Smith')
print(bob.name)
bob.name = 'Robert Smith'
print(bob.name)
del bob.name

# bob has a managed attribute
# Runs name getter (name 1)
# Runs name setter (name 2)
# Runs name deleter (name 3)

print('-'*20)
sue = Person('Sue Jones')
print(sue.name)
print(Person.name.__doc__)
# sue inherits property too
# Or help(Person.name)

```

实际上，这段代码等同于本小节的第一个示例——在这个例子中，装饰只是编写特性的一种替代方法。当运行这段代码时，结果是相同的：

```

fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
-----
fetch...
Sue Jones
name property docs

```

和`property`手动赋值的结果相比，这个例子中，使用装饰器来编写特性只需要3行额外的代码（这是无法忽视的差别）。就像替代工具的通常情况一样，在这两种技术之间的选择很大程度上与个人爱好有关。

描述符

描述符提供了拦截属性访问的一种替代方法；它们与前面小节所讨论的特性有很大的关系。实际上，特性是描述符的一种——从技术上讲，`property`内置函数只是创建一个特定类型的描述符的一种简化方式，而这种描述符在属性访问时运行方法函数。

从功能上讲，描述符协议允许我们把一个特定属性的`get`和`set`操作指向我们提供的一个单

独类对象的方法：它们提供了一种方式来插入在访问属性的时候自动运行的代码，并且它们允许我们拦截属性删除并且为属性提供文档（如果愿意的话）。

描述符作为独立的类创建，并且它们就像方法函数一样分配给类属性。和任何其他类属性一样，它们可以通过子类 and 实例继承。通过为描述符自身提供一个 `self`，以及提供客户类的实例，都可以提供访问拦截方法。因此，它们可以自己保留和使用状态信息，以及主体实例的状态信息。例如，一个描述符可能调用客户类上可用的方法，以及它所定义的特定于描述符的方法。

和特性一样，描述符也管理一个单独的、特定的属性。尽管它不能广泛地捕获所有的属性访问，但它提供了对获取和赋值访问的控制，并且允许我们自由地把简单的数据修改为计算值从而改变一个属性，而不会影响已有的代码。特性实际上只是创建一种特定描述符的方便方法，并且，正如我们所见到的，它们可以直接作为描述符编写。

然而，特性的应用领域相对狭窄，描述符提供了一种更为通用的解决方案。例如，由于它们编码为常规类，所以描述符拥有自己的状态，可能参与描述符继承层级，可以使用复合来聚合对象，并且为编写内部方法和属性文档字符串提供一种自然的结构。

基础知识

正如前面所提到的，描述符作为单独的类编写，并且针对想要拦截的属性访问操作提供特定命名的访问器方法——当以相应的方式访问分配给描述符类实例的属性时，描述符类中的获取、设置和删除等方法自动运行：

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): ...    # Return attr value
    def __set__(self, instance, value): ...    # Return nothing (None)
    def __delete__(self, instance): ...        # Return nothing (None)
```

带有任何这些方法的类都可以看作是描述符，并且当它们的一个实例分配给另一个类的属性的时候，它们的这些方法是特殊的——当访问属性的时候，会自动调用它们。如果这些方法中的任何一个空缺，通常意味着不支持相应类型的访问。然而，和特性不同，省略一个 `__set__` 意味着允许这个名字在一个实例中重新定义，因此，隐藏了描述符——要使得一个属性是只读的，我们必须定义 `__set__` 来捕获赋值并引发一个异常。

描述符方法参数

在进行任何真正的编程之前，先来回顾一些基础知识。前面小节介绍的所有3种描述符方法，都传递了描述符类实例（`self`）以及描述符实例所附加的客户类的实例（`instance`）。

`__get__` 访问方法还额外地接收一个`owner`参数，指定了描述符实例要附加到的类。其`instance`参数要么是访问的属性所属的实例（用于`instance.attr`），要么当所访问的属性直接属于类的时候是`None`（用于`class.attr`）。前者通常针对实例访问计算一个值；如果描述符对象访问是受支持的，后者通常返回`self`。

例如，在下面的例子中，当获取`X.attr`的时候，Python自动运行`Descriptor`类的`__get__`方法，`Subject.attr`类属性分配给该方法（和特性一样，在Python 2.6中，要在这里使用描述符，我们必须派生自对象；在Python 3.0中，这是隐式的，但无伤大雅）：

```
>>> class Descriptor(object):
...     def __get__(self, instance, owner):
...         print(self, instance, owner, sep='\n')
...
>>> class Subject:
...     attr = Descriptor()           # Descriptor instance is class attr
...
>>> X = Subject()

>>> X.attr
<__main__.Descriptor object at 0x0281E690>
<__main__.Subject object at 0x028289B0>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x0281E690>
None
<class '__main__.Subject'>
```

注意在第一个属性获取中自动传递到`__get__`方法中的参数，当获取`X.attr`的时候，就好像发生了如下的转换（尽管这里的`Subject.attr`没有再次调用`__get__`）：

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

当描述符的实例参数为`None`的时候，该描述符知道将直接访问它。

只读描述符

正如前面提到的，和特性不同，使用描述符直接忽略`__set__`方法不足以让属性成为只读的，因为描述符名称可以赋给一个实例。在下面的例子中，对`X.a`的属性赋值在实例对象`X`中存储了`a`，由此，隐藏了存储在类`C`中的描述符：

```
>>> class D:
...     def __get__(*args): print('get')
...
>>> class C:
...     a = D()
...
... 
```

```

>>> X = C()
>>> X.a                                     # Runs inherited descriptor __get__
get
>>> C.a
get
>>> X.a = 99                               # Stored on X, hiding C.a
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                                     # Y still inherits descriptor
get
>>> C.a
get

```

这就是Python中所有实例属性赋值工作的方式，并且它允许在它们的实例中类选择性地覆盖类级默认值。要让基于描述符的属性成为只读的，捕获描述符类中的赋值并引发一个异常来阻止属性赋值——当要赋值的属性是一个描述符的时候，Python有效地绕过了常规实例层级的赋值行为，并且把操作指向描述符对象：

```

>>> class D:
...     def __get__(*args): print('get')
...     def __set__(*args): raise AttributeError('cannot set')
...
>>> class C:
...     a = D()
...
>>> X = C()
>>> X.a                                     # Routed to C.a.__get__
get
>>> X.a = 99                               # Routed to C.a.__set__
AttributeError: cannot set

```

注意： 还要注意不要把描述符`__delete__`方法和通用的`__del__`方法搞混淆了。调用前者是试图删除所有者类的一个实例上的管理属性名称；后者是一种通用的实例析构器方法，当任何类的一个实例将要进行垃圾回收的时候调用。`__delete__`与我们将要在本章后面遇到的`__delattr__`泛型属性删除方法关系更近。参见本书第29章了解关于操作符重载的更多内容。

第一个示例

要看这些如何组合到更为实际的代码中，让我们从前面为特性编写的第一个例子开始。如下代码定义了一个描述符，来拦截对其客户类中的名为`name`的一个属性的访问。其方法使用它们的`instance`参数来访问主体实例中的状态信息，其中指定了实际存储的名称字符串。和特性一样，描述符只对新式类能够很好地工作，因此，如果使用Python 2.6

的话，要确保下面例子中的类都派生自对象object：

```
class Name:                                # Use (object) in 2.6
    "name descriptor docs"
    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name
    def __set__(self, instance, value):
        print('change...')
        instance._name = value
    def __delete__(self, instance):
        print('remove...')
        del instance._name

class Person:                              # Use (object) in 2.6
    def __init__(self, name):
        self._name = name
        name = Name()                     # Assign descriptor to attr

bob = Person('Bob Smith')                  # bob has a managed attribute
print(bob.name)                           # Runs Name.__get__
bob.name = 'Robert Smith'                 # Runs Name.__set__
print(bob.name)                           # Runs Name.__delete__
del bob.name

print('-'*20)
sue = Person('Sue Jones')                 # sue inherits descriptor too
print(sue.name)
print(Name.__doc__)                       # Or help(Name)
```

注意，在这段代码中，我们如何把描述符类的一个实例分配给客户类中的一个类属性；正因为如此，它为该类的所有实例所继承，就像是一个类方法一样。实际上，我们必须像这样把描述符赋给一个类属性——如果赋给一个self实例属性，它将无法工作。当描述符的__get__方法运行的时候，它传递了3个对象来定义其上下文：

- self是Name类实例
- instance是Person类实例
- owner是Person类实例

当这段代码运行描述符的方法来拦截对该属性的访问的时候，和特性的版本十分相似。实际上，输出再一次相同：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
-----
fetch...
```

```
Sue Jones
name descriptor docs
```

还和特性示例中相似，我们的描述符类实例是一个类属性，并且因此由客户类和任何子类的所有实例所继承。例如，如果我们把示例中的Person类修改为如下的样子，脚本的输出是相同的：

```
...
class Super:
    def __init__(self, name):
        self._name = name
        name = Name()

class Person(Super):
    pass
...
# Descriptors are inherited
```

还要注意，当一个描述符类在客户类之外无用的话，将描述符的定义嵌入客户类之中，这在语法上是完全合理的。这里，我们的示例看上去就像使用一个嵌套的类：

```
class Person:
    def __init__(self, name):
        self._name = name

    class Name:
        "name descriptor docs"
        def __get__(self, instance, owner):
            print('fetch...')
            return instance._name
        def __set__(self, instance, value):
            print('change...')
            instance._name = value
        def __delete__(self, instance):
            print('remove...')
            del instance._name
    name = Name()
```

当按照这种方式编码时，Name变成了Person类声明的作用域中的一个局部变量，这样，它不会与类之外的任何名称冲突。这个版本和最初的版本一样地工作——我们已经直接把描述符类定义移动到了客户类的作用域中，但是，测试代码的最后一行必须改为从其新位置获取文档字符串：

```
...
print(Person.Name.__doc__)
# Differs: not Name.__doc__ outside class
```

计算的属性

和使用特性的例子一样，上一小节中我们的第一个描述符例子也没有做太多事情——它直接打印了属性访问的追踪消息。实际上，描述符也可以用来在每次获取属性的时候计

算它们的值。如下例子说明了这点——它是我们为特性编写的同一个例子的改写，这里使用了一个描述符，从而在每次获取属性值的时候对其值自动求平方：

```
class DescSquare:
    def __init__(self, start):                # Each desc has own state
        self.value = start
    def __get__(self, instance, owner):      # On attr fetch
        return self.value ** 2
    def __set__(self, instance, value):      # On attr assign
        self.value = value                  # No delete or docs

class Client1:
    X = DescSquare(3)                        # Assign descriptor instance to class attr

class Client2:
    X = DescSquare(32)                       # Another instance in another client class
                                           # Could also code 2 instances in same class

c1 = Client1()
c2 = Client2()

print(c1.X)                                # 3 ** 2
c1.X = 4
print(c1.X)                                # 4 ** 2
print(c2.X)                                # 32 ** 2
```

运行这个例子的时候，其输出与最初的基于特性的版本相同，但是在这里，一个描述符类对象正在拦截属性访问：

```
9
16
1024
```

在描述符中使用状态信息

如果已经学习了我们到目前为止编写的两个描述符的例子，你可能会注意到，它们从不同的地方获取其信息——第一个例子（name属性的示例）使用存储在客户**实例**中的数据，第二个例子（属性乘方的例子）使用附加到**描述符**对象本身的数据。实际上，描述符可以使用实例状态和描述符状态，或者二者的任何组合：

- 描述符状态用来管理内部用于描述符工作的数据。
- 实例状态记录了和客户类相关的信息，以及可能由客户类创建的信息。

描述符方法也可以使用，但是描述符状态常常使得不必要使用特定的命名惯例，以避免存储在一个实例上的描述符数据的名称冲突。例如，如下的描述符把信息附加到自己的实例，因此，它不会与客户类的实例上的信息冲突：

```

class DescState:                                # Use descriptor state
    def __init__(self, value):
        self.value = value
    def __get__(self, instance, owner):          # On attr fetch
        print('DescState get')
        return self.value * 10
    def __set__(self, instance, value):          # On attr assign
        print('DescState set')
        self.value = value

# Client class

class CalcAttrs:
    X = DescState(2)                            # Descriptor class attr
    Y = 3                                        # Class attr
    def __init__(self):
        self.Z = 4                            # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                    # X is computed, others are not
obj.X = 5                                     # X assignment is intercepted
obj.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)

```

这段代码的Value信息仅存在于**描述符**之中，因此，如果在客户类的实例中使用相同的名字，也不会有冲突。注意，这里只是管理了描述符的属性——对X的获取和设置访问被拦截，但是对Y和Z的访问没有拦截（Y附加到客户类，Z附加到其实例）。当这段代码运行的时候，X在获取的时候会计算：

```

DescState get
20 3 4
DescState set
DescState get
50 6 7

```

对描述符存储或使用附加到客户类的实例的一个属性，而不是自己的属性，这也是可行的。如下例子中的描述符假设实例有一个属性_Y通过客户类附加，并且使用它来计算它所表示的属性的值：

```

class InstState:                                # Using instance state
    def __get__(self, instance, owner):
        print('InstState get')                # Assume set by client class
        return instance._Y * 100
    def __set__(self, instance, value):
        print('InstState set')
        instance._Y = value

# Client class

class CalcAttrs:
    X = DescState(2)                            # Descriptor class attr

```

```

Y = InstState()
def __init__(self):
    self._Y = 3
    self._Z = 4

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)
obj.X = 5
obj.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)

```

Descriptor class attr

Instance attr

Instance attr

X and Y are computed, Z is not

X and Y assignments intercepted

这一次，X和Y都赋给了描述符，并且获取的时候会计算（X是赋给了前面的例子的描述符）。这里新的描述符本身没有信息，但是它使用了一个假设存在于实例中的属性——这个属性名为_Y，以避免与描述符自身的名称冲突。当这个版本的代码运行的时候，结果是类似的，但是，管理的是第二个属性，使用位于实例中的状态而不是描述符：

```

DescState get
InstState get
20 300 4
DescState set
InstState set
DescState get
InstState get
50 600 7

```

描述符和实例状态都有各自的用途。实际上，这是描述符优于特性的一个通用优点——因为它们都有自己的状态，所以可以很容易地在内部保存数据，而不用将数据添加到客户实例对象的命名空间中。

特性和描述符是如何相关的

正如前面提到的，特性和描述符有很强的相关性——property内置函数只是创建描述符的一种方便方式。既然已经知道了二者是如何工作的，我们应该能够看到，可以使用如下的一个描述符类来模拟property内置函数：

```

class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
        # Save unbound methods
        # or other callables

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")
        return self.fget(instance)
        # Pass instance to self
        # in property accessors

```

```

def __set__(self, instance, value):
    if self.fset is None:
        raise AttributeError("can't set attribute")
    self.fset(instance, value)

def __delete__(self, instance):
    if self.fdel is None:
        raise AttributeError("can't delete attribute")
    self.fdel(instance)

class Person:
    def getName(self): ...
    def setName(self, value): ...
    name = Property(getName, setName)           # Use like property()

```

这个Property类捕获了带有描述符协议的属性访问，并且把请求定位到创建类的时候在描述符状态中传入和保存的函数或方法。例如，属性获取从Person类指向Property类的__get__方法，再回到Person类的getName。有了描述符，这“恰好可以工作”。

注意，尽管这个描述符类等同于只是处理基本的特性用法，使用@decorator语法也只是指定了设置和删除操作，但是我们的Property类也必须用setter和deleter方法扩展，这可能会节省装饰的访问器函数并且返回特性对象（self应该足够了）。既然property内置函数已经做了这些，这里，我们将省略这一扩展的正式编码。

还要注意，描述符用来实现Python的__slots__，使用存储在类级别的描述符来截取slot名称，从而避免了实例属性字典。参见第31章了解关于slot的更多介绍。

注意： 在第38章中，我们还将使用描述符来实现应用于函数和方法的函数装饰器。正如你将在那里见到的，由于描述符接收描述符和主体类实例，它们在这种情况下工作得很好，尽管嵌套函数通常是一种更简单的解决方案。

__getattr__和__getattribute__

到目前为止，我们已经学习了特性和描述符——管理特定属性的工具。__getattr__和__getattribute__操作符重载方法提供了拦截类实例的属性获取的另一种方法。就像特性和描述符一样，它们也允许我们插入当访问属性的时候自动运行的代码。然而，我们将会看到，这两个方法有更广泛的应用。

属性获取拦截表现为两种形式，可用两个不同的方法来编写：

- __getattr__针对未定义的属性运行——也就是说，属性没有存储在实例上，或者没有从其类之一继承。

- `__getattribute__` 针对**每个**属性，因此，当使用它的时候，必须小心避免通过把属性访问传递给超类而导致递归循环。

我们在本书第29章中遇到过前一种情况，它在Python的所有版本中都可用。后者对于Python 2.6中的新式类可用，并且对于Python 3.0中的所有类（隐式都是新式类）可用。这两个方法是一组属性拦截方法的代表，这些方法还包括`__setattr__`和`__delattr__`。由于这些方法具有相同的作用，我们在这里将它们通常作为一个单独话题。

与特性和描述符不同，这些方法是Python的**操作符重载**协议的一部分——是类的特殊命名的方法，由子类继承，并且当在隐式的内置操作中使用实例的时候自动调用。和一个类的所有方法一样，它们每一个在调用的时候都接收第一个`self`参数，访问任何请求的实例状态信息或该类的其他方法。

`__getattr__`和`__getattribute__`方法也比特性和描述符更加**通用**——它们可以用来拦截对任何（几乎所有的）实例属性的获取，而不仅仅是分配给它们的那些特定名称。因此，这两个方法很适合于通用的基于**委托**的编码模式——它们可以用来实现包装对象，该对象管理对一个嵌套对象的所有属性访问。相反，我们必须为想要拦截的每个属性都定义一个特性或描述符。

最后，这两种方法比我们前面考虑的替代方法的应用领域更**专注集中**一些：它们只是拦截属性获取，而不拦截属性赋值。要捕获赋值对属性的更改，我们必须编写一个`__setattr__`方法——这是一个操作符重载方法，只对每个属性获取运行，必须小心避免由于通过实例命名空间字典指向属性赋值而导致的递归循环。

尽管很少用到，我们还是可以编写一个`__delattr__`重载方法（必须以同样的方式避免循环）来拦截属性删除。相反，特性和描述符通过设计捕获访问、设置和删除操作。

大多数这些操作符重载方法在本书前面都介绍过，这里，我们将展开讨论其用法并研究它们在更大的应用环境中的作用。

基础知识

`__getattr__`和`__setattr__`在本书第29章和第31章介绍，并且第31章简单地提到了`__getattribute__`。简而言之，如果一个类定义了或继承了如下方法，那么当一个实例用于后面的注释所提到的情况时，它们将自动运行：

```
def __getattr__(self, name):           # On undefined attribute fetch [obj.name]
def __getattribute__(self, name):      # On all attribute fetch [obj.name]
def __setattr__(self, name, value):    # On all attribute assignment [obj.name=value]
def __delattr__(self, name):           # On all attribute deletion [del obj.name]
```

所有这些之中，`self`通常是主体实例对象，`name`是将要访问的属性的字符串名，`value`是要赋给该属性的对象。两个`get`方法通常返回一个属性的值，另两个方法不返回什么（`None`）。例如，要捕获每个属性获取，我们可以使用上面的前两个方法；要捕获属性赋值，可以使用第三个方法：

```
class Catcher:
    def __getattr__(self, name):
        print('Get:', name)
    def __setattr__(self, name, value):
        print('Set:', name, value)

X = Catcher()
X.job                                # Prints "Get: job"
X.pay                                # Prints "Get: pay"
X.pay = 99                           # Prints "Set: pay 99"
```

这样的代码结构可以用来实现我们在第30章介绍的**委托**设计模式。由于所有的属性通常都指向我们的拦截方法，所以我们可以验证它们并将其传递到嵌入的、管理的对象中。例如，下面的类（取自第30章）跟踪了对传递给包装类的另一个对象的**每一次**属性获取：

```
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
```

特性和描述符没有这样的类似功能，做不到对每个可能的包装对象中每个可能的属性编写访问器。

避免属性拦截方法中的循环

这些方法通常都容易使用，它们唯一复杂的部分就是潜在的**循环**（即递归）。由于`__getattr__`仅针对未定义的属性调用，所以它可以在自己的代码中自由地获取其他属性。然而，由于`__getattribute__`和`__setattr__`针对所有的属性运行，因此，它们的代码要注意在访问其他属性的时候避免再次调用自己并触发一次递归循环。

例如，在一个`__getattribute__`方法代码内部的另一次属性获取，将会再次触发`__getattribute__`，并且代码将会循环直到内存耗尽：

```
def __getattribute__(self, name):
    x = self.other                       # LOOPS!
```

要解决这个问题，把获取指向一个更高的超类，而不是跳过这个层级的版本——`object`类总是一个超类，并且它在这里可以很好地起作用：

```
def __getattr__(self, name):
    x = object.__getattr__(self, 'other')           # Force higher to avoid me
```

对于`__setattr__`，情况是类似的。在这个方法内赋值任何属性，都会再次触发`__setattr__`并创建一个类似的循环：

```
def __setattr__(self, name, value):
    self.other = value                             # LOOPS!
```

要解决这个问题，把属性作为实例的`__dict__`命名空间字典中的一个键赋值。这样就避免了直接的属性赋值：

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value                 # Use attr dict to avoid me
```

尽管这种方法比较少用到，但`__setattr__`也可以把自己的属性赋值传递给一个更高的超类而避免循环，就像`__getattr__`一样：

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value)       # Force higher to avoid me
```

相反，我们不能使用`__dict__`技巧在`__getattr__`中避免循环：

```
def __getattr__(self, name):
    x = self.__dict__['other']                     # LOOPS!
```

获取`__dict__`属性本身会再次触发`__getattr__`，导致一个递归循环。很奇怪，但确实如此。

`__delattr__`方法实际中很少用到，但是，当用到的时候，它针对每次属性删除而调用（就像针对每次属性赋值调用`__setattr__`一样）。因此，我们必须小心，在删除属性的时候要避免循环，通过使用同样的技术：命名空间字典或者超类方法调用。

第一个示例

所有这些并不像我们前一小节所暗示的那样复杂。要看看如何把这些思想付诸应用，这里是与用来说明特性和描述符的示例一样的示例，不过这次是用属性操作符重载方法实现的。由于这些方法如此通用，所以我们在这里测试属性名来获知何时将要访问一个管理的属性；其他的则允许正常传递：

```
class Person:
    def __init__(self, name):                       # On [Person()]
        self._name = name                           # Triggers __setattr__!

    def __getattr__(self, attr):                     # On [obj.undefined]
        if attr == 'name':                           # Intercept name: not stored
```

```

        print('fetch...')
        return self._name
    else:
        raise AttributeError(attr)

    def __setattr__(self, attr, value):
        if attr == 'name':
            print('change...')
            attr = '_name'
            self.__dict__[attr] = value

    def __delattr__(self, attr):
        if attr == 'name':
            print('remove...')
            attr = '_name'
            del self.__dict__[attr]

bob = Person('Bob Smith')
print(bob.name)
bob.name = 'Robert Smith'
print(bob.name)
del bob.name

print('-'*20)
sue = Person('Sue Jones')
print(sue.name)
#print(Person.name.__doc__)

```

Does not loop: real attr
Others are errors
On [obj.any = value]
Set internal name
Avoid looping here
On [del obj.any]
Avoid looping here too
but much less common
bob has a managed attribute
Runs __setattr__
Runs __setattr__
Runs __delattr__
sue inherits property too
No equivalent here

注意，`__init__` 构造函数中的属性赋值也触发了 `__setattr__`，这个方法捕获了每次属性赋值，即便是类自身之中的那些。运行这段代码的时候，会产生同样的输出，但这一次，它是Python的常规操作符重载机制与我们的属性拦截方法的结果：

```

fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
-----
fetch...
Sue Jones

```

还要注意，与特性和描述符不同，这里没有为属性直接声明指定的**文档**，管理的属性存在于我们拦截方法的代码之中，而不是在不同的对象中。

要实现与 `__getattribute__` 相同的结果，用下面的代码替换示例中的 `__setattr__`，由于它会捕获所有的属性获取，这个版本必须通过把新的获取传递到超类来避免循环，并且通常不能假设未知的名称是错误：

```

# Replace __setattr__ with this

def __getattribute__(self, attr):

```

On [obj.any]


```

if attr == 'name':
    print('fetch...')
    attr = '_name'
return object.__getattr__(self, attr)

```

Intercept all names
Map to internal name
Avoid looping here

这个例子与我们为属性和描述符编写的代码相同，但是它有点人为的痕迹，并且它没有真正地强调这些工具的应用。由于它们是通用的，所以`__getattr__`和`__getattribute__`可能在基于委托的代码中更为常用（正如前面所介绍的），在那里属性访问验证并指向一个嵌入的对象。在只有单个的属性要管理的情况下，特性和描述符可能会做得更好。

计算属性

正如前面所述，我们前面的示例除了跟踪属性获取没有做任何事情。当获取属性并计算属性值时，它没有做太多额外工作。与介绍特性和描述符时的情况相同，下面的代码创建了一个虚拟的属性X，当获取的时候自动计算它：

```

class AttrSquare:
    def __init__(self, start):
        self.value = start

    def __getattr__(self, attr):
        if attr == 'X':
            return self.value ** 2
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):
        if attr == 'X':
            attr = 'value'
        self.__dict__[attr] = value

A = AttrSquare(3)
B = AttrSquare(32)

print(A.X)
A.X = 4
print(A.X)
print(B.X)

```

Triggers __setattr__!
On undefined attr fetch
value is not undefined
On all attr assignments
2 instances of class with overloading
Each has different state information
*# 3 ** 2*
4
*# 4 ** 2*
*# 32 ** 2*

运行这段代码，会产生与前面我们使用特性和描述符的时候相同的输出，但是，这段脚本的机制是基于通用的属性拦截方法：

```

9
16
1024

```

如前所述，我们可以用`__getattribute__`而不是`__getattr__`实现同样的效果。下面的代码用一个`__getattribute__`替换了获取方法，并且通过使用直接超类方法调用而不是

`__dict__` 键来修改 `__setattr__` 赋值方法从而避免循环:

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                                # Triggers __setattr__!

    def __getattribute__(self, attr):
        if attr == 'X':
            return self.value ** 2                        # Triggers __getattribute__ again!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value):
        if attr == 'X':
            attr = 'value'
            object.__setattr__(self, attr, value)         # On all attr assignments
```

当这个版本运行的时候, 结果再次相同。注意, 隐式的指向在类的方法内部进行:

- 构造函数中的 `self.value=start` 触发 `__setattr__`。
- `__getattribute__` 中 `self.value` 再次触发 `__getattribute__`。

实际上, 每次我们获取属性X的时候, `__getattribute__` 都运行了**两次**。这并没有在 `__getattr__` 版本中发生, 因为 `value` 属性没有定义。如果你关心速度并且要避免这一点, 修改 `__getattribute__` 以使用超类来获取 `value`:

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

当然, 这仍然会引发对超类方法的一次调用, 但是, 在获取值之前没有额外的递归调用。给这些方法添加 `print` 调用, 以记录它们如何运行和何时运行。

`__getattr__` 和 `__getattribute__` 比较

为了概括 `__getattr__` 和 `__getattribute__` 之间的编码区别, 下面的例子使用了这两者来实现3个属性——`attr1`是一个类属性, `attr2`是一个实例属性, `attr3`是一个虚拟的管理属性, 当获取时计算它:

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):
        print('get: ' + attr)
        return 3

X = GetAttr()
```

On undefined attrs only
Not attr1: inherited from class
Not attr2: stored on instance

```

print(X.attr1)
print(X.attr2)
print(X.attr3)

print('-'*40)

class GetAttribute(object):
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):
        print('get: ' + attr)
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)

```

运行时，`__getattr__` 版本拦截对 `attr3` 的访问，因为它是未定义的。另一方面，`__getattribute__` 版本拦截所有的属性获取，并且必须将那些没有管理的属性访问指向超类获取器以避免循环：

```

1
2
get: attr3
3
-----
get: attr1
1
get: attr2
2
get: attr3
3

```

尽管 `__getattribute__` 可以捕获比 `__getattr__` 更多的属性获取，但是实际上，它们只是一个主题的不同变体——如果属性没有物理地存储，二者具有相同的效果。

管理技术比较

为了概括我们在本章介绍的4种属性管理方法之间的编码区别，让我们快速地来看看使用每种技术的一个更全面的计算属性的示例。如下的版本使用特性来拦截并计算名为 `square` 和 `cube` 的属性。注意它们的基本值是如何存储到以下划线开头的名称中的，因此，它们不会与特性本身的名称冲突：

```

# 2 dynamically computed attributes with properties

```

```

class Powers:
    def __init__(self, square, cube):
        self._square = square          # _square is the base value
        self._cube = cube              # square is the property name

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)

X = Powers(3, 4)
print(X.square)      # 3 ** 2 = 9
print(X.cube)        # 4 ** 3 = 64
X.square = 5
print(X.square)      # 5 ** 2 = 25

```

要用描述符做到同样的事情，我们用完整的类定义了属性。注意，描述符把基础值存储为实例状态，因此，它们必须再次使用下划线开头，以便不会与描述符的名称冲突（正如我们将在本章最后的示例中见到的，我们可以通过把基础值存储为描述符状态，从而避免必须重新命名）：

```

# Same, but with descriptors

class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube          # Use (object) in 2.6

X = Powers(3, 4)
print(X.square)      # 3 ** 2 = 9
print(X.cube)        # 4 ** 3 = 64
X.square = 5
print(X.square)      # 5 ** 2 = 25

```

*# "self.square = square" works too,
because it triggers desc __set__!*

要使用 `__getattr__` 访问拦截来实现同样的结果，我们再次用下划线开头的名称存储基础

值，这样对被管理的名称访问是未定义的，并且由此调用我们的方法。我们还需要编写一个`__setattr__`来拦截赋值，并且注意避免其潜在的循环：

Same, but with generic __getattr__ undefined attribute interception

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube
    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('unknown attr:' + name)
    def __setattr__(self, name, value):
        if name == 'square':
            self._dict__['_square'] = value
        else:
            self._dict__[name] = value

X = Powers(3, 4)
print(X.square)      # 3 ** 2 = 9
print(X.cube)        # 4 ** 3 = 64
X.square = 5
print(X.square)      # 5 ** 2 = 25
```

最后一个选项，使用`__getattribute__`来编写，类似于前一个版本。由于我们现在捕获了每一个属性，因此必须把基础值获取指向超类以避免循环：

Same, but with generic __getattribute__ all attribute interception

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube
    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)
    def __setattr__(self, name, value):
        if name == 'square':
            self._dict__['_square'] = value
        else:
            self._dict__[name] = value

X = Powers(3, 4)
print(X.square)      # 3 ** 2 = 9
print(X.cube)        # 4 ** 3 = 64
X.square = 5
```

```
print(X.square)          # 5 ** 2 = 25
```

正如你所见到的，每种技术的编码形式都有所不同，但是，所有4种方法在运行的时候都产生同样的结果：

```
9
64
25
```

要了解如何比较这些替代方案以及其他编码选项的更多内容，在本章后面“示例：属性验证”节的属性验证示例中，我们会更多地尝试它们的实际应用。在此之前，我们需要先学习和这些工具中的两种相关的一个缺点。

拦截内置操作属性

在介绍 `__getattr__` 和 `__getattribute__` 的时候，我说它们分别拦截未定义的以及所有的属性获取，这使得它们很适合用于基于委托的编码模式。尽管对于常规命名的属性来说是这样，但它们的行为需要一些额外的澄清：对于隐式地使用内置操作获取的方法名属性，这些方法可能**根本不会运行**。这意味着操作符重载方法调用不能委托给被包装的对象，除非包装类自己重新定义这些方法。

例如，针对 `__str__`、`__add__` 和 `__getitem__` 方法的属性获取分别通过打印、+表达式和索引隐式地运行，而不会指向Python 3.0中的类属性拦截方法。特别是：

- 在Python 3.0中，`__getattr__` 和 `__getattribute__` **都不会** 针对这样的属性而运行。
- 在Python 2.6中，如果属性在类中未定义的话，`__getattr__` 会针对这样的属性运行。
- 在Python 2.6中，`__getattribute__` 只对于新式类可用，并且在Python 3.0中也可以使用。

换句话说，在Python 3.0的类中（以及Python 2.6的新式类中），没有直接的方法来通用地拦截像打印和加法这样的内置操作。在Python 2.X中，这样的操作调用的方法在运行时从实例中查找，就像所有其他属性一样；在Python 3.0中，这样的方法在**类**中查找。

这种修改使得基于委托的编码模式在Python 3.0中更为复杂，因为它们不能通用地拦截操作符重载方法调用并将它们指向一个嵌入的对象。这不是一个严重的错误——包装类可以通过在自身中重新定义所有相关的操作符重载方法，从而委托调用以解决这一约束。这些额外的方法可以手动添加，用工具添加，或者通过在共同超类中定义并从共同超类继承。然而，相对于操作符重载方法是被包装对象接口的一部分的情况，这种方法确实使包装更有用处。

记住，这个问题只适用于 `__getattr__` 和 `__getattribute__`。由于特性和描述符只针对特定属性定义，所以它们根本不能真正应用于基于代理的类——单个特性或描述符不能用于拦截任意属性。此外，定义操作符重载方法和属性拦截的一个类将能够正确地工作，而不管定义的属性拦截的类型。我们在这里只是关心没有定义操作符重载方法但是力图通用地拦截它们的类。

考虑如下的例子，即文件 `getattr.py`，它在包含了 `__getattr__` 和 `__getattribute__` 方法的类的实例上，测试各种属性类型和内置操作：

```
class GetAttr:
    eggs = 88                                # eggs stored on class, spam on instance
    def __init__(self):
        self.spam = 77
    def __len__(self):                        # len here, else __getattr__ called with __len__
        print('__len__: 42')
        return 42
    def __getattr__(self, attr):              # Provide __str__ if asked, else dummy func
        print('getattr: ' + attr)
        if attr == '__str__':
            return lambda *args: '[Getattr str]'
        else:
            return lambda *args: None

class GetAttribute(object):                  # object required in 2.6, implied in 3.0
    eggs = 88                                # In 2.6 all are isinstance(object) auto
    def __init__(self):                      # But must derive to get new-style tools,
        self.spam = 77                      # incl __getattribute__, some __X__ defaults
    def __len__(self):
        print('__len__: 42')
        return 42
    def __getattribute__(self, attr):
        print('getattribute: ' + attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))

    X = Class()
    X.eggs                                # Class attr
    X.spam                                # Instance attr
    X.other                                # Missing attr
    len(X)                                # __len__ defined explicitly

    try:                                  # New-styles must support [], +, call directly: redefine
        X[0]                              # __getitem__?
    except:
        print('fail []')
```

```

try:
    X + 99                                # __add__?
except:
    print('fail +')

try:
    X()                                    # __call__? (implicit via built-in)
except:
    print('fail ()')
X.__call__()                             # __call__? (explicit, not inherited)

print(X.__str__())                       # __str__? (explicit, inherited from type)
print(X)                                 # __str__? (implicit via built-in)

```

在Python 2.6下运行的时候，`__getattr__`的确接收针对内置操作的各种隐式属性获取，因为Python通常在实例中查询这样的属性。相反，对于任何操作符重载名，`__getattribute__`不会运行，因为这样的名称只在类中查找：

```

C:\misc> c:\python26\python getattr.py

GetAttr=====
getattr: other
__len__: 42
getattr: __getitem__
getattr: __coerce__
getattr: __add__
getattr: __call__
getattr: __call__
getattr: __str__
[GetAttr str]
getattr: __str__
[GetAttr str]

GetAttribute=====
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025EA1D0>

```

注意，在Python 2.6中，这里的`__getattr__`如何拦截`__call__`和`__str__`的隐式和显式获取。相反，对于内置操作的任何属性名，`__getattribute__`不能捕捉隐式获取。

确实，`__getattribute__`例子在Python 2.6中与其在Python 3.0中是相同的，因为在Python 2.6中类必须通过派生自`object`成为新式类，才能使用这个方法。这段代码的`object`派生在Python 3.0中是可选的，因为其中所有的类都是新式的。

然而，在Python 3.0下运行的时候，`__getattr__`的结果不同——当通过内置操作获取属性的时候，没有隐式运行的操作符重载方法会触发哪个属性拦截方法。在解析这样的名称的时候，Python 3.0省略了常规实例查找机制：

```
C:\misc> c:\python30\python getattr.py

GetAttr=====
getattr: other
__len__: 42
fail []
fail +
fail ()
getattr: __call__
<__main__.GetAttr object at 0x025D17F0>
<__main__.GetAttr object at 0x025D17F0>

GetAttribute=====
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025D1870>
```

我们可以跟踪这些输出，从而了解到脚本中的打印，看看这是如何工作的：

- 在Python 3.0中，`__str__`访问有两次未能被`__getattr__`捕获：一次是针对内置打印，一次是针对显式获取，因为从该类继承了一个默认方法（实际上，该类来自内置object，它是每个类的一个超类）。
- `__str__`只有一次未能被`__getattribute__`捕获，在内置打印操作中，显式获取绕过了继承的版本。
- `__call__`在Python 3.0中用于内置调用表达式的两次都没有捕获，但是，当显式获取的时候，它两次都拦截到了；和`__str__`不同，没有继承的`__call__`默认版本能够超越`__getattr__`。
- `__len__`被两个类都捕获了，直接原因是，它在类自身中是一个显式定义的方法——它的名称指明了，在Python 3.0中，如果我们删除了类的`__len__`方法，它不会指向`__getattr__`或`__getattribute__`。
- 所有其他的内置操作在Python 3.0中都没有被两种方案拦截。

再一次，直接的效果是，由内置操作隐式地运行的操作符重载方法始终不会通过Python

3.0中的某个属性拦截方法指向：Python 3.0在类中查找这样的属性，并且完全忽略了实例查找。

这使得基于委托的包装类在Python 3.0中更难以编写，如果被包装的类可能包含操作符重载方法，这些方法必须冗余地在包装类中重新定义，从而能够委托给被包装的对象。在一般的委托工具中，这可能会增加很多额外的方法。

当然，这些方法的增加可能一部分是工具自动进行的，通过用新的方法来扩展类做到（这里，下两章将要介绍的类装饰器和元类可能会有帮助）。此外，超类可能能够一次性定义所有这些额外方法，以便在基于委托的类中继承。然而，在Python 3.0中，委托编码模式需要额外的工作。

要了解关于这一现象的更实际的说明及其解决方法，参见下一章中的Private装饰器示例。在那里，我们将看到，也可能在客户类中插入一个__getattr__从而保留其最初的类型，尽管这个方法仍然不会为了操作符重载方法而调用；例如，打印仍然直接运行在这样的一个类中定义的__str__，而不是通过__getattr__指向请求。

作为另一个例子，下一小节重复了我们的类教程示例。既然理解了属性拦截是如何工作的，我们将能够解释稍微奇怪一点的一个例子。

注意：要查看这一Python 3.0改变在Python自身中工作的一个例子，参见本书第14章中Python 3.0的os.popen对象的介绍。由于它用一个包装类实现，而该包装类使用__getattr__把属性获取委托给一个嵌入的对象，它没有拦截Python 3.0中的next(X)内置迭代器函数，该函数定义为运行__next__。然而，它确实拦截并委托显式X.__next__()调用，因为这些不是通过内置函数指向的，并且没有像__str__那样继承自一个超类。

这等同于我们的例子中的__call__——对内置函数的隐式调用不会触发__getattr__，但对于不是继承自类类型的显式调用，则会触发__getattr__。换句话说，这一改变不仅影响到我们的委托者，而且会影响到Python标准库中的那些类！既然这一修改有了一定的范围，这些行为未来可能会发展，因此，确保在以后的版本中验证这个问题。

重访基于委托的Manager

第27章的面向对象教程展示了一个Manager类，它使用对象嵌套和方法委托来定制它的超类，而不是使用继承。这里再次引用那段代码，删除了一些不相关的测试：

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
```

```

        return self.name.split()[-1]
def giveRaise(self, percent):
    self.pay = int(self.pay * (1 + percent))
def __str__(self):
    return '[Person: %s, %s]' % (self.name, self.pay)

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)           # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)           # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)                 # Delegate all other attrs
    def __str__(self):
        return str(self.person)                           # Must overload again (in 3.0)

if __name__ == '__main__':
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000) # Manager.__init__
    print(tom.lastName())             # Manager.__getattr__ -> Person.lastName
    tom.giveRaise(.10)                 # Manager.giveRaise -> Person.giveRaise
    print(tom)                         # Manager.__str__ -> Person.__str__

```

这个文件末尾的注释展示了一行的操作调用哪个方法。特别是，注意`lastName`调用如何在`Manager`中是未定义的，并由此指向通用的`__getattr__`，又从那里到了嵌入的`Person`对象。如下是这段脚本的输出——Sue从`Person`那里接收了10%的加薪，但Tom得到了20%的加薪，因为`giveRaise`在`Manager`中定制了：

```

C:\misc> c:\python30\python getattr.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]

```

相反，注意当我们在这段脚本末尾打印`Manager`的时候发生了什么：调用了包装类的`__str__`，并且它委托到嵌入的`Person`对象的`__str__`。记住这一点，看看如果我们在代码中删除了`Manager.__str__`方法，将会发生什么事情：

```

# Delete the Manager __str__ method

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)           # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)           # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)                 # Delegate all other attrs

```

现在，在Python 3.0下，针对Manager对象，打印不会通过通用的__getattr__拦截器指向其属性获取。相反，继承自类的隐式object超类的一个默认__str__显示方法，被查找并运行（sue仍然正确地打印，因为Person有一个显式的__str__）：

```
C:\misc> c:\python30\python person.py
Jones
[Person: Sue Jones, 110000]
Jones
<__main__.Manager object at 0x02A5AE30>
```

奇怪的是，像这样没有一个__str__运行，在Python 2.6中却会触发__getattr__，因为操作符重载属性通过这个方法指向，并且类不会为__str__继承一个默认版本：

```
C:\misc> c:\python26\python person.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

切换到__getattribute__在这里也不会对Python 3.0有所帮助——像__getattr__一样，对于Python 2.6和Python 3.0中内置操作所暗示的操作符重载属性，它都不会运行：

```
# Replace __getattr__ with __getattribute__

class Manager:                                # Use (object) in 2.6
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)    # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)    # Intercept and delegate
    def __getattribute__(self, attr):
        print('***', attr)
        if attr in ['person', 'giveRaise']:
            return object.__getattribute__(self, attr) # Fetch my attrs
        else:
            return getattr(self.person, attr)        # Delegate all others
```

不管在Python 3.0中使用哪个属性拦截方法，我们仍然必须在Manager中包含一个重新定义的__str__（如上面所示），以便拦截打印操作并将它们指向嵌入的Person对象：

```
C:\misc> c:\python30\python person.py
Jones
[Person: Sue Jones, 110000]
** lastName
** person
Jones
** giveRaise
** person
<__main__.Manager object at 0x028E0590>
```

注意，__getattribute__针对方法获得两次调用——一次针对方法名，另一次针对

`self.person`嵌入对象获取。我们可以用一种不同的编码方式来避免如此，但是，我们仍然必须重定义`__str__`以捕获打印，尽管这里有所不同（`self.person`将导致`__getattr__`失效）：

```
# Code __getattr__ differently to minimize extra calls

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)
    def __getattr__(self, attr):
        print('**', attr)
        person = object.__getattr__(self, 'person')
        if attr == 'giveRaise':
            return lambda percent: person.giveRaise(percent+.10)
        else:
            return getattr(person, attr)
    def __str__(self):
        person = object.__getattr__(self, 'person')
        return str(person)
```

运行这一替代方案的时候，我们的对象正确地打印了，但是只是因为我们已经在包装类中添加了一个显式的`__str__`——这个属性仍然没有指向通用的属性拦截方法：

```
Jones
[Person: Sue Jones, 110000]
** lastName
Jones
** giveRaise
[Person: Tom Jones, 60000]
```

这里简单介绍了像`Manager`这样的基于委托的类，在Python 3.0中必须重定义某些操作符重载方法（例如`__str__`）才能将它们指向嵌套的对象，但是，在Python 2.6中不必，除非使用了新式类。我们唯一的直接选择似乎是使用`__getattr__`和Python 2.6，或者在Python 3.0中在包装类中冗余地重定义操作符重载方法。

再一次说明，这不是一个不可能的任务。很多包装类可以预计所需的操作符重载方法的集合，并且工具和超类可以将这个任务的一部分自动化。此外，并非所有的类都使用操作符重载方法（实际上，大多数应用程序类通常不会使用）。然而，对于在Python 3.0中使用的委托编码模式，需要记住一些事情。当操作符重载方法是一个对象的接口的一部分时，包装类必须通过在本本地重新定义它们来容纳它们。

示例：属性验证

为了结束本章的内容，让我们来看一个更实际的示例，以所有的4种属性管理方案来编写代码。我们将要使用的这个示例定义了一个`CardHolder`对象，它带有4个属性，其中3个属性是要管理的。管理的属性在获取或存储的时候要验证或转换值。对于同样的测试

代码，所有4个版本都产生同样的结果，但是，它们以不同的方式实现了它们的属性。这个示例包含了很大一部分需要自学的内容。然而我们不会详细介绍其代码，因为它们都使用了我们在本章中已经介绍过的概念。

使用特性来验证

我们的第一段代码使用了特性来管理3个属性。与通常一样，我们可以使用简单的方法而不是管理属性，但是，如果我们在已有的代码中已经使用了属性，特性就能帮忙了。特性根据属性访问自动运行代码，但是关注属性的一个特定集合，它们不会用来广泛地拦截所有属性。

要理解这段代码，关键是要注意到，`__init__`构造函数方法内部的属性赋值也触发了特性的setter方法。例如，当这个方法分配给`self.name`时，它自动调用`setName`方法，该方法转换值并将其赋给一个叫做`__name`的实例属性，以便它不会与特性的名称冲突。

这一重命名（有时候叫做**名称压缩**）是必要的，因为特性使用公用的实例状态并且没有自己的实例状态。存储在一个属性中的数据叫做`__name`，而叫做`name`的属性总是特性，而非数据。

最后，这个类管理了叫做`name`、`age`和`acct`的属性；允许直接访问属性`addr`，并且提供了一个名为`remain`的只读属性，该属性完全是虚拟的并且根据需要计算。为了进行比较，这个基于特性的程序包含了39行代码：

```
class CardHolder:
    acctlen = 8                               # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Instance data
        self.name = name                     # These trigger prop setters too
        self.age = age                       # __X mangled to have class name
        self.addr = addr                    # addr is not managed
                                           # remain has no data

    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)

    def getAge(self):
        return self.__age

    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
```

```

        self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('invalid acct number')
        else:
            self.__acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):
        return self.retireage - self.age
    remain = property(remainGet)

```

Could be a method, not attr
Unless already using as attr

self测试代码

如下的代码测试我们的类；将这段代码添加到文件的底部，或者把类放到一个模块中并先导入它。我们将对这个例子的所有4个版本使用这段同样的测试代码。当它运行的时候，我们创建了管理的属性类的两个实例，并且获取和修改其各种属性。期待失效的操作包装在try语句中：

```

bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')
bob.name = 'Bob Q. Smith'
bob.age = 50
bob.acct = '23-45-67-89'
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')

sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
print(sue.acct, sue.name, sue.age, sue.remain, sue.addr, sep=' / ')
try:
    sue.age = 200
except:
    print('Bad age for Sue')

try:
    sue.remain = 5
except:
    print("Can't set sue.remain")

try:
    sue.acct = '1234567'
except:
    print('Bad acct for Sue')

```

如下是我们的self测试代码的输出。再一次说明，这对这个示例的所有版本都是一样的。分析这段代码，看看类的方法是如何调用的。账户显示出来，其中一些数字隐藏

了，名称转换为一种标准格式，并且使用一个类属性访问拦截的时候，截止到退休的剩余时间就计算了出来：

```
12345*** / bob_smith / 40 / 19.5 / 123 main st
23456*** / bob_q._smith / 50 / 9.5 / 123 main st
56781*** / sue_jones / 35 / 24.5 / 124 main st
Bad age for Sue
Can't set sue.remain
Bad acct for Sue
```

使用描述符验证

现在，让我们使用描述符而不是使用特性来重新编写示例。正如我们已经看到的，描述符在功能和角色上与特性类似。实际上，特性基本上是描述符的一种受限制的形式。和特性一样，描述符设计来处理特定的属性访问，而不是通用的属性访问。和特性不同，描述符有自己的状态，并且它们是一种更为通用的方案。

要理解这段代码，注意`__init__`构造函数方法内部的属性赋值触发了描述符的`__set__`操作符方法，这一点还是很重要。例如，当构造函数方法分配给`self.name`时，它自动调用`Name.__set__()`方法，该方法转换值，并且将其赋给了叫做`name`的一个描述符属性。

和前面基于特性的变体不同，在这个例子中，实际的`name`值附加到了**描述符**对象，而不是客户类实例。尽管我们可以把这个值存储在实例状态或描述符状态中，后者避免了需要用下划线压缩名称以避免冲突。在`CardHolder`客户类中，名为`name`的属性总是一个描述符对象，而不是数据。

最后，这个类实现了和前面的版本同样的属性：它管理名为`name`、`age`和`acct`的属性。允许直接访问属性`addr`，并且提供一个名为`remain`的只读属性，`remain`是完全虚拟的并且根据需要计算。注意我们为何在其描述符中捕获对`remain`的赋值，并引发一个异常。正如我们前面所介绍的，如果没有这么做，对一个实例这一属性的赋值，将会默默地创建一个实例属性而隐藏了类属性描述符。为了进行比较，这个基于描述符的代码占了45行：

```
class CardHolder:
    acctlen = 8                               # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Instance data
        self.name = name                     # These trigger __set__ calls too
        self.age = age                       # __X not needed: in descriptor
        self.addr = addr                    # addr is not managed
                                           # remain has no data

class Name:
```



```

def __get__(self, instance, owner):
    return self.name
def __set__(self, instance, value):
    value = value.lower().replace(' ', '_')
    self.name = value
name = Name()

class Age:
    def __get__(self, instance, owner):
        return self.age
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.age = value
age = Age()

class Acct:
    def __get__(self, instance, owner):
        return self.acct[:3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:
            raise TypeError('invalid acct number')
        else:
            self.acct = value
acct = Acct()

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age
    def __set__(self, instance, value):
        raise TypeError('cannot set remain')
remain = Remain()

```

使用__getattr__来验证

正如我们已经见到的，__getattr__方法拦截所有未定义的属性，因此，它可能比使用特性或描述符更为通用。在我们的例子中，当获取一个管理的属性的时候，我们通过直接测试属性名来获知。其他的属性物理地存储在实例中，因而无法达到__getattr__。尽管这种方法比使用特性或描述符更为通用，但需要额外的工作来模拟专门关注属性的其他工具。我们需要在运行时检查名称，并且必须编写一个__setattr__以拦截并验证属性赋值。

对于这个例子的特性和描述符版本，注意__init__构造函数方法中的属性赋值触发了类的__setattr__方法，这还是很关键的。例如，当这个方法分配给self.name时，它自动地调用__setattr__方法，该方法转换值，并将其分配给一个名为name的实例属性。通过在该实例上存储name，它确保了未来的访问不会触发__getattr__。相反，acct存储为_acct，因此随后对acct的访问会调用__getattr__。

最后，像前两个例子中的情况一样，这个类管理名为name、age和acct的属性。允许直接访问属性addr；并且提供一个名为remain的只读属性，它是完全虚拟的并且根据需要计算。

为了进行比较，这个替代方法有32行代码——比基于特性的版本少了7行，比使用描述符的版本少了13行。当然，清晰与否比代码大小更重要，但额外的代码有时候意味着额外的开发和维护工作。可能这里更重要的是角色：像__getattr__这样的通用工具可能更适合于通用委托，而特性和描述符更直接是为了管理特定属性而设计。

还要注意，当设置未管理的属性（例如，addr）的时候，这里的代码引发额外调用，然而获取未管理的属性并不会引发额外调用，因为它们定义了。尽管这可能导致大多数程序都会导致不可忽视的额外开销，但只有当访问管理的属性的时候，特性和描述符才会引发额外调用。

下面是代码的__getattr__版本：

```
class CardHolder:
    acctlen = 8                               # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Instance data
        self.name = name                     # These trigger __setattr__ too
        self.age = age                       # _acct not mangled: name tested
        self.addr = addr                     # addr is not managed
                                           # remain has no data

    def __getattr__(self, name):
        if name == 'acct':
            return self._acct[:-3] + '***'    # On undefined attr fetches
        elif name == 'remain':
            return self.retireage - self.age   # name, age, addr are defined
        else:
            raise AttributeError(name)         # Doesn't trigger __getattr__

    def __setattr__(self, name, value):
        if name == 'name':
            value = value.lower().replace(' ', '_')  # On all attr assignments
        elif name == 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')
            # addr stored directly
            # acct mangled to _acct
        elif name == 'acct':
            name = '_acct'
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invalid acct number')
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value           # Avoid looping
```

使用__getattribute__验证

最后的变体使用__getattribute__在需要的时候拦截属性获取并管理它们。这里，每次属性获取都会捕获，因此，我们测试属性名称来检测管理的属性，并将所有其他的属性指向超类以实现常规的获取过程。这个版本和前面的版本一样，使用了同样的__setattr__来捕获赋值。

这段代码的工作和__getattr__版本很相似，因此，我不想在这里重复整个介绍。然而，注意，由于每个属性获取都指向了__getattribute__，所以这里我们不需要压缩名称以拦截它们（acct存储为acct）。另一方面，这段代码必须负责把未压缩的属性获取指向一个超类以避免循环。

还要注意，对于设置和获取未管理的属性（例如，addr），这个版本都会引发额外调用。如果速度极为重要，这个替代方法可能会是所有方案中最慢的。为了进行比较，这个版本也有32行代码，和前面的版本一样：

```
class CardHolder:
    acctlen = 8                               # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Instance data
        self.name = name                     # These trigger __setattr__ too
        self.age = age                       # acct not mangled: name tested
        self.addr = addr                    # addr is not managed
                                           # remain has no data

    def __getattribute__(self, name):
        superget = object.__getattribute__    # Don't loop: one level up
        if name == 'acct':                   # On all attr fetches
            return superget(self, 'acct')[:-3] + '***'
        elif name == 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        else:
            return superget(self, name)       # name, age, addr: stored

    def __setattr__(self, name, value):
        if name == 'name':                  # On all attr assignments
            value = value.lower().replace(' ', '_') # addr stored directly
        elif name == 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        elif name == 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invalid acct number')
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value         # Avoid loops, orig names
```

要确保自己学习和运行本节中的代码，以获得关于管理属性编码技术的更多技巧。

本章小结

本章介绍了在Python中管理对属性进行访问的各种技术，包括`__getattr__`和`__getattribute__`操作符重载方法、类特性和属性描述符。此外，还比较和对比了这些工具，并给出了一些用例来展示它们的行为。

第38章继续我们的构建工具学习，来看看装饰器，即在函数和类创建的时候而不是属性访问的时候运行的代码。在继续学习之前，让我们来看一组练习题，以复习在本章学习的内容。

本章习题

1. `__getattr__`和`__getattribute__`有何区别？
2. 特性和描述符有何区别？
3. 特性和装饰器有何关联？
4. `__getattr__`和`__getattribute__`以及特性和描述符之间主要的功能区别是什么？
5. 所有这些功能的比较只是某种争论吗？

习题解答

1. `__getattr__`方法只针对**未定义**属性的获取运行，即那些没有在一个实例上显示的属性，以及没有从它的任何类继承的属性。相反，`__getattribute__`方法针对所有的属性获取运行，不管属性是否定义了。因此，`__getattr__`中的代码可以自由地获取其他属性，如果它们定义了的话；而`__getattribute__`针对所有这样的属性获取使用特定代码以避免循环（它必须把获取指向一个超类以跳过自身）。
2. 特性充当一个特定角色，而描述符更为通用。特性定义了特定属性的获取、设置和删除功能。描述符也提供了一个类，带有完成这些操作的方法，但是，它们提供了额外的灵活性以支持更多任意行为。实际上，特性真的只是创建特定描述符的一种简单方法——即在属性访问上运行的一个描述符。编码上也有区别：特性通过一个内置函数创建，而描述符用一个类来编码；同样，描述符可以利用类的所有常用OOP功能，例如继承。此外，除了实例的状态信息，描述符有它们自己的本地状态，因此，它们可以避免在实例中的名称冲突。

3. 特性可以用装饰器语法编写。由于`property`内置函数接受一个单个的函数参数，它可以直接用作一个函数装饰器来定义一个获取访问特性。由于名称重新绑定装饰器的行为，所以被装饰的函数的名称分配给了一个特性，而特性的获取访问器设置为最初装饰的函数（`name = property(name)`）。特性的`setter`和`deleter`属性允许我们进一步用装饰器语法添加设置和删除访问器——它们把访问器设置为装饰的函数并且返回扩展的特性。
4. `__getattr__`和`__getattribute__`方法更为通用：它们用来捕获任意多的属性。相反，每个特性或描述符只针对一个特定属性提供访问拦截——我们不能用一个单个的特性或描述符捕获每个属性获取。另一方面，特性和描述符都通过设计来处理属性获取和赋值：`__getattr__`和`__getattribute__`只处理获取；要拦截赋值，必须编写`__setattr__`。实现也是不同的：`__getattr__`和`__getattribute__`是操作符重载方法，而特性和描述符是手动赋给类属性的对象。
5. 并非如此。引用Python同名的喜剧*Monty Python's Flying Circus*中的台词：

```
An argument is a connected series of statements intended to establish a
proposition.
No it isn't.
Yes it is! It's not just contradiction.
Look, if I argue with you, I must take up a contrary position.
Yes, but that's not just saying "No it isn't."
Yes it is!
No it isn't!
Yes it is!
No it isn't. Argument is an intellectual process. Contradiction is just
the automatic gainsaying of any statement the other person makes.
(short pause)
No it isn't.
It is.
Not at all.
Now look...
```

装饰器

在本书的高级类主题一章中（第31章），我们介绍了静态方法和类方法，并且快速浏览了Python提供的声明装饰器的语法@ decorator。我们还在上一章（第37章）遇到过函数装饰器，并探讨了property内置函数充当装饰器的能力，在第28章中，我们还学习了抽象超类的概念。

本章选择了前面对装饰器介绍所没有涉及的内容。这里，我们将深入装饰器的内部工作机制，并学习自己编写新的装饰器的更多高级方法。正如我们将要了解到的，我们在前面各章学习到的很多概念（例如状态保持），会在装饰器中常规出现。

这是一个颇为高级的话题，并且装饰器的构建对于工具构架者比对于应用程序员的意义更大。此外，装饰器在流行Python框架中变得越来越常见，对其基本的理解有助于认识它们的作用，即便你只是一个装饰器的用户。

除了详细介绍装饰器的构建，本章还作为装饰器在Python中应用的更为实际的**案例研究**。由于本章的示例比我们在本书其他各章见到的示例都要大，它们更好地说明了代码如何组合为较为完整的系统和工具。作为额外的好处，我们在这里编写的很多代码可以作为通用的工具用于日常编程之中。

什么是装饰器

装饰是为函数和类指定管理代码的一种方式。装饰器本身的形式是处理其他的可调对象的可调用的对象（如函数）。正如我们在本书前面所见到过的，Python装饰器以两种相关的形式呈现：

- 函数装饰器在函数定义的时候进行名称重绑定，提供一个逻辑层来管理函数和方法或随后对它们的调用。
- 类装饰器在类定义的时候进行名称重绑定，提供一个逻辑层来管理类，或管理随后调用它们所创建的示例。

简而言之，装饰器提供了一种方法，在函数和类定义语句的末尾插入**自动运行代码**——对于函数装饰器，在`def`的末尾；对于类装饰器，在`class`的末尾。这样的代码可以扮演不同的角色，参见后面小节介绍。

管理调用和实例

例如，通常的用法中，这种自动运行的代码可能用来增强对函数和类的调用。它通过针对随后的调用安装**包装器对象**来实现这一点：

- 函数装饰器安装包装器对象，以在需要的时候拦截随后的**函数调用**并处理它们。
- 类装饰器安装包装器对象，以在需要的时候拦截随后的**实例创建调用**并处理它们。

装饰器通过自动把函数和类名重绑定到其他的可调用对象来实现这些效果，在`def`和`class`语句的末尾做到这点。当随后调用的时候，这些可调用对象可以执行诸如对函数调用跟踪和计时、管理对类实例属性的访问等任务。

管理函数和类

尽管本章的大多数实例都使用包装器来拦截随后对函数和类的调用，但这并非使用装饰器的唯一方法：

- 函数装饰器也可以用来管理**函数对象**，而不是随后对它们的调用——例如，把一个函数注册到一个API。然而，我们在这里主要关注更为常见的用法，即调用包装器应用程序。
- 类装饰器也可以用来直接管理类对象，而不是实例创建调用——例如，用新的方法扩展类。因为这些用法和**元类**有很大的重合（实际上，都是在类创建过程的最后运行），我们将在下一章看到更多的例子。

换句话说，函数装饰器可以用来管理函数调用和函数对象，类装饰器可以用来管理类实例和类自身。通过返回装饰的对象自身而不是一个包装器，装饰器变成了针对函数和类的一种简单的后创建步骤。

不管扮演什么样的角色，装饰器都提供了一种方便而明确的方法，来编写在程序开发阶段和现实产品系统中都有用的工具。

使用和定义装饰器

根据你的工作形式，你可能成为装饰器的用户或提供者。正如我们所看到的，Python本身带有具有特定角色的内置装饰器——静态方法装饰器、属性装饰器以及更多。此外，很多流行的Python工具包括了执行管理数据库或用户接口逻辑等任务的装饰器。在这样的情况中，我们不需要知道装饰器如何编码就可以完成任务。

对于更为通用的任务，程序员可以编写自己的任意装饰器。例如，函数装饰器可能通过添加跟踪调用、在调试时执行参数验证测试、自动获取和释放线程锁、统计调用函数的次数以进行优化等的代码来扩展函数。你可以想象添加到函数调用中的任何行为，都可以作为定制函数装饰器的备选。

另外一方面，函数装饰器设计用来只增强一个特定函数或方法调用，而不是一个完整的对象接口。类装饰器更好地充当后一种角色——因为它们可以拦截实例创建调用，它们可以用来实现任意的对象接口扩展或管理任务。例如，定制类装饰器可以跟踪或验证对一个对象的每个属性引用。它们也可以用来实现代理对象、单体类以及其他常用的编程模式。实际上，我们将会发现很多类装饰器与在第30章中见到的委托编程模式有很大的相似之处。

为什么使用装饰器

像很多高级Python工具一样，从纯技术的视角来看，并不是严格需要装饰器：它们的功能往往可以使用简单的辅助函数调用或其他的技术来实现（并且从基本的层面出发，我们总是可以手动地编写装饰器所自动执行的名称重绑定）。

也就是说，装饰器为这样的任务提供了一种显式的语法，它使得意图明确，可以最小化扩展代码的冗余，并且有助于确保正确的API使用：

- 装饰器有一种非常**明确**的语法，这使得它们比那些可能任意地远离主体函数或类的辅助函数调用更容易为人们发现。
- 当主体函数或类定义的时候，装饰器应用一次；在对类或函数的每次调用的时候，不必添加额外的代码（在未来可能必须改变）。
- 由于前面两点，装饰器使得一个API的用户不太可能忘记根据API需求扩展一个函数或类。

换句话说，除了其技术模型之外，装饰器提供了一些和代码维护性和审美相关的优点。此外，作为结构化工具，装饰器自然地促进了代码的封装，这减少了冗余性并使得未来变得更加容易。

装饰器确实也有一些潜在的**缺点**——当它们插入包装类的逻辑，它们可以修改装饰的对象类型，并且它们可能引发额外的调用。另外一方面，同样的考虑也适用于任何为对象添加包装逻辑的技术。

我们将在本章随后的真实代码中说明这些权衡。尽管选择使用装饰器仍然多少有些主观性，但它们的优点引人注目，足以使其快速成为Python世界中的最佳实践。为了帮助你做出决定，让我们来看一些细节。

基础知识

让我们首先从一个符号的角度来第一次看一看装饰行为。我们很快将编写真正的代码，但是，由于装饰器的很多神奇之处可归结为自动重绑定操作，所以首先理解这一映射是很重要的。

函数装饰器

函数装饰器已经从Python 2.5开始可用。正如我们在本书前面所见到的，它们主要只是一种语法糖：通过在一个函数的`def`语句的末尾来运行另一个函数，把最初的函数名重新绑定到结果。

用法

函数装饰器是一种关于函数的**运行时声明**，函数的定义需要遵守此声明。装饰器在紧挨着定义一个函数或方法的`def`语句之前的一行编写，并且它由`@`符号以及紧随其后的对于**元函数**的一个引用组成——这是管理另一个函数的一个函数（或其他的可调用对象）。

在编码方面，函数装饰器自动将如下的语法：

```
@decorator          # Decorate function
def F(arg):
    ...

F(99)                # Call function
```

映射为这一对等的形式，其中装饰器是一个单参数的可调用对象，它返回与`F`具有相同数目的参数的一个可调用对象：

```
def F(arg):
    ...
F = decorator(F)      # Rebind function name to decorator result

F(99)                 # Essentially calls decorator(F)(99)
```

这一自动名称重绑定在`def`语句上有效，不管它针对一个简单的函数或是类中的一个方

法。当随后调用F函数的时候，它自动调用装饰器所返回的对象，该对象可能是实现了所需的包装逻辑的另一个对象，或者是最初的函数本身。

换句话说，装饰实际把如下的第一行映射为第二行（尽管装饰器实际上只运行一次，在装饰的时候）：

```
func(6, 7)
decorator(func)(6, 7)
```

这一自动名称重绑定说明了我们在本书前面遇到的静态方法和正确的装饰语法的原因：

```
class C:
    @staticmethod
    def meth(...): ...           # meth = staticmethod(meth)

class C:
    @property
    def name(self): ...          # name = property(name)
```

在这两个例子中，在def语句的末尾，方法名重新绑定到一个内置函数装饰器的结果。随后再调用最初的名称，将会调用装饰器所返回的对象。

实现

装饰器自身是一个返回可调用对象的可调用对象。也就是说，它返回了一个对象，当随后装饰的函数通过其最初的名称调用的时候，将会调用这个对象——不管是拦截了随后调用的一个包装器对象，还是最初的函数以某种方式的扩展。实际上，装饰器可以是任意类型的可调用对象，并且返回任意类型的可调用对象：函数和类的任何组合都可以使用，尽管一些组合更适合于特定的背景。

例如，要在一个函数创建之后接入装饰协议以管理函数，我们需要编写如下形式的装饰器：

```
def decorator(F):
    # Process function F
    return F

@decorator
def func(): ...           # func = decorator(func)
```

由于最初的装饰函数分配回给其名称，这么做将直接向函数的定义添加创建之后的步骤。这样的结构可能会用来把一个函数注册到一个API、赋值函数属性，等等。

更典型的用法，是插入逻辑以拦截对函数的随后调用，我们可以编写一个装饰器来返回和最初函数不同的一个对象：

```
def decorator(F):
```

```

# Save or use function F
# Return a different callable: nested def, class with __call__, etc.

@decorator
def func(): ...                # func = decorator(func)

```

这个装饰器在装饰的时候调用，并且当随后调用最初的函数名的时候，它所返回的调用对象将被调用。装饰器自身接受被装饰的函数，返回的调用对象会接受随后传递给被装饰函数的名称的任何参数。这和类方法的工作方式相同：隐含的实例对象只是在返回的可调用对象的第一个参数中出现。

更概括地说，有一种常用的编码模式可以包含这一思想——装饰器返回了一个包装器，包装器把最初的函数保持到一个封闭的作用域中：

```

def decorator(F):                # On @ decoration
    def wrapper(*args):          # On wrapped function call
        # Use F and args
        # F(*args) calls original function
    return wrapper

@decorator                        # func = decorator(func)
def func(x, y):                  # func is passed to decorator's F
    ...

func(6, 7)                       # 6, 7 are passed to wrapper's *args

```

当随后调用名称`func`的时候，它确实调用装饰器所返回的包装器函数；随后包装器函数可能会运行最初的`func`，因为它在一个**封闭的作用域**中仍然可以使用。当以这种方式编码的时候，每个装饰的函数都会产生一个新的作用域来保持状态。

为了对类做同样的事情，我们可以重载调用操作，并且使用实例属性而不是封闭的作用域：

```

class decorator:
    def __init__(self, func):      # On @ decoration
        self.func = func
    def __call__(self, *args):     # On wrapped function call
        # Use self.func and args
        # self.func(*args) calls original function

@decorator
def func(x, y):                  # func = decorator(func)
    ...                          # func is passed to __init__

func(6, 7)                       # 6, 7 are passed to __call__'s *args

```

现在，随后再调用`func`的时候，它确实会调用装饰器所创建的实例的`__call__`运算符重载方法；然后，`__call__`方法可能运行最初的`func`，因为它在一个**实例属性**中仍然可

用。当按照这种方式编写代码的时候，每个装饰的函数都会产生一个新的实例来保持状态。

支持方法装饰

关于前面的基于类的代码的细微的一点是，尽管它对于拦截简单函数调用有效，但当它应用于类方法函数的时候，并不是很有效：

```
class decorator:
    def __init__(self, func):           # func is method without instance
        self.func = func
    def __call__(self, *args):          # self is decorator instance
        # self.func(*args) fails! # C instance not in args!

class C:
    @decorator
    def method(self, x, y):             # method = decorator(method)
        ...                            # Rebound to decorator instance
```

当按照这种方式编码的时候，装饰的方法重绑定到装饰器类的一个实例，而不是一个简单的函数。

这一点带来的问题是，当装饰器的`__call__`方法随后运行的时候，其中的`self`接收装饰器类实例，并且类C的实例不会包含到一个`*args`中。这使得有可能把调用分派给最初的方法——即保持了最初的方法函数的装饰器对象，但是，没有实例传递给它。

为了支持函数和方法，嵌套函数的替代方法工作得更好：

```
def decorator(F):
    def wrapper(*args):                # F is func or method without instance
        # F(*args) runs func or method  # class instance in args[0] for method
    return wrapper

@decorator
def func(x, y):                        # func = decorator(func)
    ...
func(6, 7)                            # Really calls wrapper(6, 7)

class C:
    @decorator
    def method(self, x, y):            # method = decorator(method)
        ...                          # Rebound to simple function

X = C()
X.method(6, 7)                        # Really calls wrapper(X, 6, 7)
```

当按照这种方法编写的包装类在其第一个参数里接收了C类实例的时候，它可以分派到最初的方法和访问状态信息。

从技术上讲，这种嵌套函数版本是有效的，因为Python创建了一个绑定的方法对象，并

且由此只有当一个方法属性引用一个简单的函数的时候，才把主体类实例传递给`self`参数；相反，当它引用可调用的类的一个实例的时候，可调用的类的实例传递给`self`，以允许可调用的类访问自己的状态信息。在本章随后，我们还将看到这一细微的区别在实际实例中的作用。

还要注意，嵌套函数可能是支持函数和方法的装饰的最直接方式，但是不一定是唯一的方式。例如，上一章中的描述符，调用的时候接收了描述符和主体类实例。然而，更为复杂的是，在本章稍后，我们将看到这一工具如何在这一背景下起作用。

类装饰器

函数装饰器已经证明了是如此有用，以至于这一模式在Python 2.6和Python 3.0中扩展为允许类装饰器。类装饰器与函数装饰器密切相关，实际上，它们使用相同的语法和非常相似的编码模式。然而，不是包装单个的函数或方法，类装饰器是管理类的一种方式，或者用管理或扩展类所创建的实例的额外逻辑，来包装实例构建调用。

用法

从语法上讲，类装饰器就像前面的`class`语句一样（就像前面函数定义中出现的函数装饰器）。在语法上，假设装饰器是返回一个可调用对象的一个单参数的函数，类装饰器语法：

```
@decorator                                # Decorate class
class C:
    ...

x = C(99)                                  # Make an instance
```

等同于下面的语法——类自动地传递给装饰器函数，并且装饰器的结果返回来分配给类名：

```
class C:
    ...
C = decorator(C)                           # Rebind class name to decorator result
x = C(99)                                   # Essentially calls decorator(C)(99)
```

直接的效果就是，随后调用类名会创建一个实例，该实例会触发装饰器所返回的可调用对象，而不是调用最初的类自身。

实现

新的类装饰器使用函数装饰器所使用的众多相同的技术来编码。由于类装饰器也是返回一个可调用对象的一个可调用对象，因此大多数函数和类的组合已经足够了。

尽管先编码，但装饰器的结果是当随后创建一个实例的时候才运行的。例如，要在一个类创建之后直接管理它，返回最初的类自身：

```
def decorator(C):
    # Process class C
    return C

@decorator
class C: ...                                # C = decorator(C)
```

不是插入一个包装器层来拦截随后的实例创建调用，而是返回一个不同的可调用对象：

```
def decorator(C):
    # Save or use class C
    # Return a different callable: nested def, class with __call__, etc.

@decorator
class C: ...                                # C = decorator(C)
```

这样的类装饰器返回的可调用对象，通常创建并返回最初的类的一个新的实例，以某种方式来扩展对其接口的管理。例如，下面的实例插入一个对象来拦截一个类实例的未定义的属性：

```
def decorator(cls):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = cls(*args)
        def __getattr__(self, name):
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:
    def __init__(self, x, y):
        self.attr = 'spam'

x = C(6, 7)
print(x.attr)
```

On @ decoration
On instance creation
On attribute fetch
C = decorator(C)
Run by Wrapper.__init__
Really calls Wrapper(6, 7)
Runs Wrapper.__getattr__, prints "spam"

在这个例子中，装饰器把类的名称重新绑定到另一个类，这个类在一个封闭的作用域中保持了最初的类，并且当调用它的时候，创建并嵌入了最初的类的一个实例。当随后从该实例获取一个属性的时候，包装器的 `__getattr__` 拦截了它，并且将其委托给最初的类的嵌入的实例。此外，每个被装饰的类都创建一个新的作用域，它记住了最初的类。在本章后面，我们将用一些更有用的代码来充实这个例子。

就像函数装饰器一样，类装饰器通常可以编写为一个创建并返回可调用的对象的“工厂”函数，或者使用 `__init__` 或 `__call__` 方法来拦截所有调用操作的类，或者是由此产生的一些组合。工厂函数通常在封闭的作用域引用中保持状态，类通常在属性中保持状态。

支持多个实例

和函数装饰器一样，使用类装饰器的时候，一些可调用对象组合比另一些工作得更好。考虑前面例子的类装饰器的一个如下的无效替代方式：

```
class Decorator:
    def __init__(self, C):
        self.C = C
    def __call__(self, *args):
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...

x = C()
y = C()
```

On @ decoration
On instance creation
On attribute fetch
C = Decorator(C)
Overwrites x!

这段代码处理多个被装饰的类（每个都产生一个新的`Decorator`实例），并且会拦截实例创建调用（每个运行`__call__`方法）。然而，和前面的版本不同，这个版本没有能够处理给定的类的**多个实例**——每个实例创建调用都覆盖了前面保存的实例。最初的版本确实支持多个实例，因为每个实例创建调用产生了一个新的独立的包装器对象。更通俗地说，如下模式中的每一个都支持多个包装的实例：

```
def decorator(C):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = C(*args)
        return Wrapper

class Wrapper: ...
def decorator(C):
    def onCall(*args):
        return Wrapper(C(*args))
    return onCall
```

On @ decoration
On instance creation
On @ decoration
On instance creation
Embed instance in instance

我们将在本章随后一个更为实用的环境中研究这一现象，然而，在实际中，我们必须小心地正确组合可调用类型以支持自己的意图。

装饰器嵌套

有的时候，一个装饰器不够。为了支持多步骤的扩展，装饰器语法允许我们向一个装饰的函数或方法添加包装器逻辑的多个层。当使用这一功能的时候，每个装饰器必须出现在自己的一行中。这种形式的装饰器语法：

```
@A
@B
```

```
@C
def f(...):
    ...
```

如下这样运行：

```
def f(...):
    ...
f = A(B(C(f)))
```

这里，最初的函数通过3个不同的装饰器传递，并且最终的可调用对象返回来分配给最初名称。每个装饰器处理前一个的结果，这可能是最初的函数或一个插入的包装器。

如果所有的装饰器都插入包装器，直接的效果就是，当调用最初的函数名时，将会调用包装对象逻辑的3个不同的层，从而以3种不同的方式扩展最初的函数。列出的最后的装饰器是第一次应用的并且最深层次的嵌套。

就像对函数一样，多个类装饰器导致了多个嵌套的函数调用，并且可能导致围绕实例创建调用的包装器逻辑的多个层。例如，如下的代码：

```
@spam
@eggs
class C:
    ...

X = C()
```

等同于如下的代码：

```
class C:
    ...
C = spam(eggs(C))

X = C()
```

再次，每个装饰器都自由地返回最初的类或者一个插入的包装器对象。有了包装器，当最终请求最初C类的一个实例的时候，这一调用会重定向到spam和eggs装饰器提供的包装层对象，二者可能有任意的不同角色。

例如，如下的什么也不做的装饰器只是返回被装饰的函数：

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3
```



```
def func():
    print('spam')

func()

# func = d1(d2(d3(func)))
# Prints "spam"
```

同样的语法在类上也有效，就像这里什么也不做的装饰器一样。

然而，当装饰器插入包装器函数对象，调用的时候它们可能扩展最初的函数——如下的代码将其结果连接到一个装饰器层中，随着它从内向外地运行层：

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():
    return 'spam'

print(func())

# func = d1(d2(d3(func)))
# Prints "XYZspam"
```

我们在这里使用了`lambda`函数来实现包装器层（每个层在一个封闭的作用域里保持了包装的函数）。实际上，包装器可以采取函数、可调用的类以及更多形式。当设计良好的时候，装饰器嵌套允许我们以种类多样的方式来组合扩展步骤。

装饰器参数

函数装饰器和类装饰器似乎都能接受参数，尽管实际上这些参数传递给了真正返回装饰器的一个可调对象，而装饰器反过来又返回一个可调对象。例如，如下代码：

```
@decorator(A, B)
def F(arg):
    ...
    F(99)
```

自动地映射到其对等的形式，其中装饰器是一个可调对象，它返回实际的装饰器。返回的装饰器反过来返回可调用的对象，这个对象随后运行以调用最初的函数名：

```
def F(arg):
    ...
F = decorator(A, B)(F)      # Rebind F to result of decorator's return value
F(99)                      # Essentially calls decorator(A, B)(F)(99)
```

装饰器参数在装饰发生之前就解析了，并且它们通常用来保持状态信息供随后的调用使用。例如，这个例子中的装饰器函数，可能采用如下的形式：

```
def decorator(A, B):
```

```

# Save or use A, B
def actualDecorator(F):
    # Save or use function F
    # Return a callable: nested def, class with __call__, etc.
    return callable
return actualDecorator

```

这个结构中的外围函数通常会把装饰器参数与状态信息分开保存，以便在实际的装饰器中使用，或者在它所返回的可调用对象中使用，或者在二者中都使用。这段代码在封闭的函数作用域引用中保存了状态信息参数，但是通常也可以使用类属性。

换句话说，装饰器参数往往意味着可调用对象的3个层级：接受装饰器参数的一个可调用对象，它返回一个可调用对象以作为装饰器，该装饰器返回一个可调用对象来处理对最初的函数或类的调用。这3个层级的每一个都可能是一个函数或类，并且可能以作用域或类属性的形式保存了状态。我们将在本章后面看到应用装饰器参数的实际例子。

装饰器管理函数和类

尽管本章剩下的很大篇幅集中在包装对函数和类的随后调用，但我应该强调装饰器机制比这更加通用——它是在函数和类创建之后通过一个可调用对象传递它们的一种协议。因此，它可以用来调用任意的创建后处理：

```

def decorate(O):
    # Save or augment function or class O
    return O

@decorator
def F(): ...                # F = decorator(F)

@decorator
class C: ...                # C = decorator(C)

```

只要以这种方式返回最初装饰的对象，而不是返回一个包装器，我们就可以管理函数和类自身，而不只是管理随后对它们的调用。在本章稍后，我们将看到使用这一思想的更为实际的例子，它们用装饰把可调用对象注册到一个API，并且在创建函数的时候为它们赋值属性。

编写函数装饰器

现回到代码层面。在本章剩下的内容里，我们将学习实际的例子来展示刚刚介绍的装饰器概念。本小节展示几个函数装饰器的实际例子，下一小节展示实际的类装饰器例子。此后，我们将通过使用类和函数装饰器的一些较大的例子来结束本章。

跟踪调用

首先，让我们回顾一下第31章介绍的跟踪器的例子。如下的代码定义并应用一个函数装饰器，来统计对装饰的函数的调用次数，并且针对每一次调用打印跟踪信息：

```
class tracer:
    def __init__(self, func):          # On @ decoration: save original func
        self.calls = 0
        self.func = func
    def __call__(self, *args):        # On later calls: run original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        self.func(*args)

@tracer
def spam(a, b, c):                  # spam = tracer(spam)
    print(a + b + c)               # Wraps spam in a decorator object
```

注意，用这个类装饰的每个函数将如何创建一个新的实例，带有自己保存的函数对象和调用计数器。还要注意观察，`*args`参数语法如何用来打包和解包任意的多个传入参数。这一通用性使得这个装饰器可以用来包装带有任意多个参数的任何函数（这个版本还不能在类方法上工作，但是，我们将在本小节稍后修改这一点）。

现在，如果导入这个模块的函数并交互地测试它，将会得到如下的一种行为——每次调用都初始地产生一条跟踪信息，因为装饰器类拦截了调用。这段代码在Python 2.6和Python 3.0下都能运行，就像本章中所有其他的代码一样，除非特别提示：

```
>>> from decorator1 import spam

>>> spam(1, 2, 3)                  # Really calls the tracer wrapper object
call 1 to spam
6

>>> spam('a', 'b', 'c')          # Invokes __call__ in class
call 2 to spam
abc

>>> spam.calls                     # Number calls in wrapper state information
2

>>> spam
<decorator1.tracer object at 0x02D9A730>
```

运行的时候，`tracer`类和装饰的函数分开保存，并且拦截对装饰的函数随后的调用，以便添加一个逻辑层来统计和打印每次调用。注意，调用的总数如何作为装饰的函数的一个属性显示——装饰的时候，`spam`实际上是`tracer`类的一个实例（对于进行类型检查的程序，可能还会衍生一次查找，但是通常是有益的）。

对于函数调用，@装饰语法可能比修改每次调用来说明额外的逻辑层要更加方便，并且它避免了意外地直接调用最初的函数。考虑如下所示的非装饰器的对等代码：

```
calls = 0
def tracer(func, *args):
    global calls
    calls += 1
    print('call %s to %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)

>>> spam(1, 2, 3)                # Normal non-traced call: accidental?
1 2 3

>>> tracer(spam, 1, 2, 3)        # Special traced call without decorators
call 1 to spam
1 2 3
```

这一替代方法可以用在任何函数上，且不需要特殊的@语法，但是和装饰器版本不同，它在代码中调用函数的每个地方需要额外的语法。此外，它的意图可能不够明显，并且它不能确保额外的层将会针对常规调用而调用。尽管装饰器不是必需的（我们总是可以手动地重新绑定名称），它们通常是最为方便的。

状态信息保持选项

前面小节的最后一个例子引发了一个重要的问题。函数装饰器有各种选项来保持装饰的时候所提供的状态信息，以便在实际函数调用过程中使用。它们通常需要支持多个装饰的对象以及多个调用，但是，有多种方法来实现这些目标：实例属性、全局变量、非局部变量和函数属性，都可以用于保持状态。

类实例属性

例如，这里是前面的例子的一个扩展版本，其中添加了对关键字参数的支持，并且返回包装函数的结果，以支持更多的用例：

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

@tracer
def spam(a, b, c):
```

State via instance attributes
On @ decorator
Save func for later call
On call to original function
Same as: spam = tracer(spam)

```

    print(a + b + c)                # Triggers tracer.__init__

@tracer
def eggs(x, y):                    # Same as: eggs = tracer(eggs)
    print(x ** y)                  # Wraps eggs in a tracer object

spam(1, 2, 3)                      # Really calls tracer instance: runs tracer.__call__
spam(a=4, b=5, c=6)               # spam is an instance attribute

eggs(2, 16)                        # Really calls tracer instance, self.func is eggs
eggs(4, y=4)                      # self.calls is per-function here (need 3.0 nonlocal)

```

就像最初的版本一样，这里的代码使用**类实例属性**来显式地保存状态。包装的函数和调用计数器都是针对**每个实例**的信息——每个装饰都有自己的拷贝。当在Python 2.6和Python 3.0下运行一段脚本的时候，这个版本的输出如下所示。注意spam和eggs函数的每一个是如何有自己的调用计数器的，因为每个装饰都创建一个新的类实例：

```

call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256

```

尽管对于装饰函数有用，但是当应用于方法的时候，这种编码方案也有问题（随后更为详细地介绍）。

封闭作用域和全局作用域

封闭def作用域引用和嵌套的def常常可以实现相同的效果，特别是对于装饰的最初函数这样的静态数据。然而，在这个例子中，我们也需要封闭的作用域中的一个计数器，它随着每次调用而更改，并且，这在Python 2.6中是不可能的。在Python 2.6中，我们可以使用类和属性，正如我们前面所做的那样，或者使用全局声明把状态变量移出到**全局作用域**：

```

calls = 0
def tracer(func):                  # State via enclosing scope and global
    def wrapper(*args, **kwargs): # Instead of class attributes
        global calls              # calls is global, not per-function
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):                 # Same as: spam = tracer(spam)
    print(a + b + c)

```

```

@tracer
def eggs(x, y):
    print(x ** y)

spam(1, 2, 3)
spam(a=4, b=5, c=6)

eggs(2, 16)
eggs(4, y=4)

```

Same as: eggs = tracer(eggs)

Really calls wrapper, bound to func
wrapper calls spam

Really calls wrapper, bound to eggs
Global calls is not per-function here!

遗憾的是，把计数器移出到共同的全局作用域允许像这样修改它们，也意味着它们将为每个包装的函数所共享。和类实例属性不同，全局计数器是跨程序的，而不是针对每个函数的——对于任何跟踪的函数调用，计数器都会递增。如果你比较这个版本与前一个版本的输出，就可以看出其中的区别——单个的、共享的全局调用计数器根据每次装饰函数的调用不正确地更新：

```

call 1 to spam
6
call 2 to spam
15
call 3 to eggs
65536
call 4 to eggs
256

```

封闭作用域和nonlocal

共享全局状态可能是我们在某些情况下想要的。如果我们真的想要一个针对每个函数的计数器，要么像前面那样使用类，要么使用Python 3.0中新的nonlocal语句，第17章曾介绍过该语句。由于这一新的语句允许修改封闭的函数作用域变量，所以它们可以充当针对每次装饰的、可修改的数据：

```

def tracer(func):
    calls = 0
    def wrapper(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):
    print(a + b + c)

@tracer
def eggs(x, y):
    print(x ** y)

spam(1, 2, 3)
spam(a=4, b=5, c=6)

```

State via enclosing scope and nonlocal
Instead of class attrs or global
calls is per-function, not global

Same as: spam = tracer(spam)

Same as: eggs = tracer(eggs)

Really calls wrapper, bound to func
wrapper calls spam

```
eggs(2, 16)           # Really calls wrapper, bound to eggs
eggs(4, y=4)         # Nonlocal calls _is_ not per-function here
```

现在，由于封装的作用域变量不能跨程序而成为全局的，所以每个包装的函数再次有了自己的计数器，就像是针对类和属性一样。这里是在Python 3.0下运行时新的输出：

```
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256
```

函数属性

最后，如果你没有使用Python 3.X并且没有一条`nonlocal`语句，可能仍然能够针对某些可改变的状态使用**函数属性**来避免全局和类。在最新的Pythons中，我们可以把任意属性分配给函数以附加它们，使用`func.attr=value`就可以了。在我们的例子中，可以直接对状态使用`wrapper.calls`。如下的代码与前面的`nonlocal`版本一样地工作，因为计数器再一次是针对每个装饰的函数的，但是，它也可以在Python 2.6下运行：

```
def tracer(func):           # State via enclosing scope and func attr
    def wrapper(*args, **kwargs): # calls is per-function, not global
        wrapper.calls += 1
        print('call %s to %s' % (wrapper.calls, func.__name__))
        return func(*args, **kwargs)
    wrapper.calls = 0
    return wrapper
```

注意，这种方法有效，只是因为名称`wrapper`保持在封闭的`tracer`函数的作用域中。当我们随后增加`wrapper.calls`时，并不是在修改名称`wrapper`本身，因此，不需要`nonlocal`声明。

这种方案几乎作为一个脚注来介绍，因为它比Python 3.0中的`nonlocal`要隐晦得多，并且可能留待其他方案无济于事的情况下使用更好。然而，我们将解答本章末尾一个问题的时候使用它，那里，我们需要从装饰器代码的**外部**访问保存的状态；`nonlocal`只能从嵌套函数自身的内部看到，但是函数属性有更广泛的可见性。

由于装饰器往往意味着可调用对象的多个层级，所以我们可以用封闭的作用域和带有属性的类来组合函数，以实现各种各样的编码结构。正如我们随后将见到的，这有时候可能比我们所期待的要细微——每个装饰的函数应该有自己的状态，并且每个装饰的类都应该需要针对自己的状态和针对每个产生实例的状态。

实际上，正如下一小节所介绍的，如果我们也想要对一个类方法应用函数装饰器，必须小心Python在作为可调用类实例对象的装饰器编码和作为函数的装饰器编码之间的区分。

类错误之一：装饰类方法

当我编写上面的第一个`tracer`函数的时候，我幼稚地假设它也应该适用于任何方法——装饰的方法应该同样地工作，但是，自动的`self`实例参数应该直接包含在`*args`的前面。遗憾的是，我错了：当应用于类方法的时候，`tracer`的第一个版本失效了，因为`self`是装饰器类的实例，并且装饰的主体类的实例没有包含在`*args`中。在Python 3.0和Python 2.6中都是如此。

我在本章前面介绍了这一现象，但是现在，我们可以在真实的工作代码环境中看到它。假设基于类的跟踪装饰器如下：

```
class tracer:
    def __init__(self, func):                # On @ decorator
        self.calls = 0                     # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):     # On call to original function
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
```

简单函数的装饰与前面介绍的一样：

```
@tracer
def spam(a, b, c):                          # spam = tracer(spam)
    print(a + b + c)                       # Triggers tracer.__init__

spam(1, 2, 3)                              # Runs tracer.__call__
spam(a=4, b=5, c=6)                       # spam is an instance attribute
```

然而，类方法的装饰失效了（更明白的读者可能会认识到，这是我们在第27章面向对象教程中的`Person`类的再现）：

```
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):           # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)

    @tracer
    def lastName(self):                     # lastName = tracer(lastName)
        return self.name.split()[-1]
```



```

bob = Person('Bob Smith', 50000)           # tracer remembers method funcns
bob.giveRaise(.25)                         # Runs tracer.__call__(???, .25)
print(bob.lastName())                     # Runs tracer.__call__(???)

```

这里问题的根源在于，`tracer`类的`__call__`方法的`self`——它是一个`tracer`实例，还是一个`Person`实例？我们真的需要将其编写为**两者都是**：`tracer`用于装饰器状态，`Person`用于指向最初的方法。实际上，`self`必须是`tracer`对象，以提供对`tracer`的状态信息的访问；不管装饰一个简单的函数还是一个方法，都是如此。

遗憾的是，当我们用`__call__`把装饰方法名重绑定到一个类实例对象的时候，Python只向`self`传递了`tracer`实例；它根本没有在参数列表中传递`Person`主体。此外，由于`tracer`不知道我们要用方法调用处理的`Person`实例的任何信息，没有办法创建一个带有一个实例的绑定的方法，因此，没有办法正确地分配调用。

实际上，前面的列表最终传递了太少的参数给装饰的方法，并且导致了一个错误。在装饰器的`__call__`方法添加一行，以打印所有的参数来验证这一点。正如你所看到的，`self`是一个`tracer`，并且`Person`实例完全缺失：

```

<__main__.tracer object at 0x02D6AD90> (0.25,) {}
call 1 to giveRaise
Traceback (most recent call last):
  File "C:/misc/tracer.py", line 56, in <module>
    bob.giveRaise(.25)
  File "C:/misc/tracer.py", line 9, in __call__
    return self.func(*args, **kwargs)
TypeError: giveRaise() takes exactly 2 positional arguments (1 given)

```

正如前面提到的，出现这种情况是因为：当一个方法名绑定只是绑定到一个简单的函数，Python向`self`传递了隐含的主体实例；当它是一个可调用类的实例的时候，就传递这个类的实例。从技术上讲，当方法是一个简单函数的时候，Python只是创建了一个绑定的方法对象，其中包含了主体实例。

使用嵌套函数来装饰方法

如果想要函数装饰器在简单函数和类方法上都能工作，最直接的解决方法在于使用前面介绍的状态保持方法之一——把自己的函数装饰器编写为嵌套的`def`，以便对于包装器类实例和主体类实例都不需要依赖于单个的`self`实例参数。

如下的替代方案使用Python 3.0的`nonlocal`。由于装饰的方法重新绑定到简单的函数而不是实例对象，所以Python正确地传递了`Person`对象作为第一个参数，并且装饰器将其从`*args`中的第一项传递给真正的、装饰的方法的`self`参数：

```

# A decorator for both functions and methods

```

```

def tracer(func):
    calls = 0
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

# Applies to simple functions

@tracer
def spam(a, b, c):
    print(a + b + c)

spam(1, 2, 3)
spam(a=4, b=5, c=6)

# Applies to class method functions too!

class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

    @tracer
    def lastName(self):
        return self.name.split()[-1]

print('methods...')
bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())

```

这个版本在函数和方法上都有效：

```

call 1 to spam
6
call 2 to spam
15
methods...
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones

```

使用描述符装饰方法

尽管前一小节介绍的嵌套函数的解决方案是支持应用于函数和类方法的装饰器的最直接方法，其他的方法也是可能的。例如，我们在上一章中介绍的描述符功能，在这里也能派上用场。

还记得我们在前一章的讨论中，描述符可能是分配给对象的一个类属性，该对象带有一个 `__get__` 方法，当引用或获取该属性的时候自动运行该方法（在Python 2.6中需要对象派生，但在Python 3.0中不需要）：

```
class Descriptor(object):
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr                # Roughly runs Descriptor.__get__(Subject.attr, X, Subject)
```

描述符也能够拥有 `__set__` 和 `__del__` 访问方法，但是，我们在这里不需要它们。现在，由于描述符的 `__get__` 方法在调用的时候接收描述符类和主体类实例，因此当我们需要装饰器的状态以及最初的类实例来分派调用的时候，它很适合于装饰方法。考虑如下的替代的跟踪装饰器，它也是一个描述符：

```
class tracer(object):
    def __init__(self, func):                # On @ decorator
        self.calls = 0                      # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):      # On call to original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):       # On method attribute fetch
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):          # Save both instances
        self.desc = desc                    # Route calls back to descr
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)    # Runs tracer.__call__

@tracer
def spam(a, b, c):                          # spam = tracer(spam)
    ...same as prior...                     # Uses __call__ only

class Person:
    @tracer
    def giveRaise(self, percent):            # giveRaise = tracer(giveRaise)
        ...same as prior...                 # Makes giveRaise a descriptor
```

这和前面的嵌套的函数代码一样有效。装饰的函数只调用其`__call__`，而装饰的方法首先调用其`__get__`来解析方法名获取（在`instance.method`上），`__get__`返回的对象保持主体类实例并且随后调用以完成调用表达式，由此触发`__call__`。例如，要测试代码的调用：

```
sue.giveRaise(.10)                                # Runs __get__ then __call__
```

首先运行`tracer.__get__`，因为`Person`类的`giveRaise`属性已经通过函数装饰器重新绑定到了一个描述符。然后，调用表达式触发返回的包装器对象的`__call__`方法，它返回来调用`tracer.__call__`。

包装器对象同时保持描述符和主体实例，因此，它可以将控制指回到最初的装饰器/描述符类实例。实际上，在方法属性获取过程中，包装的对象保持了主体类实例可用，并且将其添加到了随后调用的参数列表，该参数列表会传递给`__call__`。在这个应用程序中，用这种方法把调用路由到描述符类实例是需要的，由此对包装方法的所有调用都使用描述符实例对象中的同样的调用计数器状态信息。

此外，我们也可以使用一个嵌套的函数和封闭的作用域引用来实现同样的效果——如下的版本和前面的版本一样的有效，通过为一个嵌套函数和作用域引用交换类和对象属性，但是，它所需的代码显著减少：

```
class tracer(object):
    def __init__(self, func):                # On @ decorator
        self.calls = 0                      # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):      # On call to original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):      # On method fetch
        def wrapper(*args, **kwargs):      # Retain both inst
            return self(instance, *args, **kwargs)  # Runs __call__
        return wrapper
```

为这些替代方法添加`print`语句是为了自己跟踪获取/调用过程的两个步骤，用前面嵌套函数替代方法中同样的测试代码来运行它们。在两种编码中，基于描述符的方法也比嵌套函数的选项要细致得多，因此，它可能是这里的又一种选择。在其他的环境中，它也可能是一种有用的编码模式。

在本章剩余的内容中，我们将相当随意地使用类或函数来编写函数装饰器，只要它们都只适用于函数。一些装饰器可能并不需要最初的类的实例，并且如果编写为一个类，它将在函数和方法上都有效——例如Python自己的静态方法装饰器，就不需要主体类的一个实例（实际上，它主要是从调用中删除实例）。

然而，这里的叙述的教训是，如果你想要装饰器在简单函数和类方法上都有效，最好使用基于嵌套函数的编码模式，而不是带有调用拦截的类。

计时调用

为了展示函数装饰器的各种各样能力的一个特殊样例，让我们来看一种不同的应用场景。下一个装饰器将对一个装饰的函数的调用进行计时——既有针对一次调用的时间，也有所有调用的总的时间。该装饰器应用于两个函数，以便比较列表解析和`map`内置调用所需的时间（为了便于比较，参见本书第20章中另一个非装饰器的示例，它可以作为这里的计时迭代替代方案）：

```
import time

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kwargs):
        start = time.clock()
        result = self.func(*args, **kwargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' % (self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return map((lambda x: x * 2), range(N))

result = listcomp(5)                                # Time for this call, all calls, return value
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime)             # Total time for all listcomp calls

print('')
result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)               # Total time for all mapcall calls

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

在这个例子中，一种非装饰器的方法允许主体函数用于计时或不用于计时，但是，当需

要计时的时候，它也会使调用签名变得复杂（我们需要在每个调用的时候添加代码，而不是在`def`中添加一次代码），并且可能没有直接的方法来保证一个程序中的所有列表生成器调用可以通过计时器逻辑路由，在找到所有签名并潜在地修改它们方面有所不足。

在Python 2.6中运行的时候，这个文件的self测试代码的输出如下：

```
listcomp: 0.00002, 0.00002
listcomp: 0.00910, 0.00912
listcomp: 0.09105, 0.10017
listcomp: 0.17605, 0.27622
[0, 2, 4, 6, 8]
allTime = 0.276223304917

mapcall: 0.00003, 0.00003
mapcall: 0.01363, 0.01366
mapcall: 0.13579, 0.14945
mapcall: 0.27648, 0.42593
[0, 2, 4, 6, 8]
allTime = 0.425933533452
map/comp = 1.542
```

测试细微差别：我没有在Python 3.0下运行这段代码，因为正如第14章所介绍的，`map`内置函数在Python 3.0中返回一个迭代器，而不是像在Python 2.6中那样返回一个实际的列表。由此，Python 3.0的`map`不能和一个列表解析的工作直接对应（即，`map`测试实际上在Python 3.0中没有花时间）。

如果你想要在Python 3.0下运行这段代码，那么就使用`list(map())`来迫使它像列表解析那样构建一个列表，否则，就不是真正地进行同类比较。然而，不要在Python 2.6中这么做，如果这么做了，`map`测试将会负责构建两个列表，而不是一个。

如下的代码可能会对Python 2.6和Python 3.0都公平，然而要注意，尽管这会使得Python 2.6和Python 3.0中列表解析和`map`之间的比较更公平，但因为`range`在Python 3.0中也是一个迭代器，因此Python 2.6和Python 3.0的结果不会直接比较：

```
...
import sys

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

if sys.version_info[0] == 2:
    @timer
    def mapcall(N):
        return map((lambda x: x * 2), range(N))
else:
    @timer
```

```
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))
...
```

最后，正如我们在本书的模块部分所了解到的，如果你想要能够在其他模块中重用这个装饰器，应该在文件的底部一个 `__name__ == '__main__'` 测试的下面缩进 `self` 测试代码，以便只有当文件运行的时候才运行它，而不是当导入它的时候运行。然而，我们不会这么做，因为打算给代码添加另一个功能。

添加装饰器参数

前面小节介绍的计时器装饰器有效，但是如果它更加可配置的话，那会更好——例如，提供一个输出标签并且可以打开或关闭跟踪消息，这些在一个通用目的的工具中可能很有用。装饰器参数在这里派上了用场：对它们适当编码后，我们可以使用它们来指定配置选项，这些选项可以根据每个装饰的函数而编码。例如，可以像下面这样添加标签：

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):
            ...
            print(label, ...
        return onCall
    return decorator

@timer('==>')
def listcomp(N): ...

listcomp(...)
# args passed to function
# func retained in enclosing scope
# label retained in enclosing scope
# Returns that actual decorator
# Like listcomp = timer('==>')(listcomp)
# listcomp is rebound to decorator
# Really calls decorator
```

这段代码添加了一个封闭的作用域来保持一个装饰器参数，以便随后真正调用的时候使用。当定义了 `listcomp` 函数的时候，它真的调用 `decorator`（`timer` 的结果，在真正装饰发生之前运行），带有其封闭的作用域内可用的 `label` 值。也就是说，`timer` 返回 `decorator`，后者记住了装饰器参数和最初的函数，并且返回一个可调用的对象，这个可调用对象在随后的调用时调用最初的函数。

我们可以把这种结构用于定时器之中，来允许在装饰的时候传入一个标签和一个跟踪控制标志。下面是这么做的一个例子，编码在一个名为 `mytools.py` 的模块文件中，以便它可以作为一个通用工具导入：

```
import time

def timer(label='', trace=True):
    class Timer:
        def __init__(self, func):
            self.func = func
            self.alltime = 0
    # On decorator args: retain args
    # On @: retain decorated func
```

```

def __call__(self, *args, **kwargs):    # On calls: call original
    start = time.clock()
    result = self.func(*args, **kwargs)
    elapsed = time.clock() - start
    self.alltime += elapsed
    if trace:
        format = '%s %s: %.5f, %.5f'
        values = (label, self.func.__name__, elapsed, self.alltime)
        print(format % values)
    return result
return Timer

```

我们在这里做的主要是把最初的Timer类嵌入一个封闭的函数中，以便创建一个作用域以保持装饰器参数。外围的timer函数在装饰发生前调用，并且它只是返回Timer类作为实际的装饰器。在装饰时，创建了Timer的一个实例来记住装饰函数自身，而且访问了位于封闭的函数作用域中的装饰器参数。

这一次，不是把self测试代码嵌入这个文件，我们将在一个不同的文件中运行装饰器。下面是时间装饰器的一个客户，模块文件testseqs.py，再次将其应用于序列迭代器替代方案：

```

from mytools import timer

@timer(label='[CCC]==>')
def listcomp(N):                                # Like listcomp = timer(...)(listcomp)
    return [x * 2 for x in range(N)]           # listcomp(...) triggers Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return map((lambda x: x * 2), range(N))

for func in (listcomp, mapcall):
    print('')
    result = func(5)                            # Time for this call, all calls, return value
    func(50000)
    func(500000)
    func(1000000)
    print(result)
    print('allTime = %s' % func.alltime)        # Total time for all calls

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))

```

再一次说明，如果想要在Python 3.0中正常地运行这段代码，把map函数包装到一个list调用中。当在Python 2.6中运行的时候，这文件打印出如下的输出——每个装饰的函数现在都有了一个子集的标签，该标签由装饰器参数定义：

```

[CCC]==> listcomp: 0.00003, 0.00003
[CCC]==> listcomp: 0.00640, 0.00643
[CCC]==> listcomp: 0.08687, 0.09330
[CCC]==> listcomp: 0.17911, 0.27241
[0, 2, 4, 6, 8]

```



```

allTime = 0.272407666337

[MMM]==> mapcall: 0.00004, 0.00004
[MMM]==> mapcall: 0.01340, 0.01343
[MMM]==> mapcall: 0.13907, 0.15250
[MMM]==> mapcall: 0.27907, 0.43157
[0, 2, 4, 6, 8]
allTime = 0.431572169089
map/comp = 1.584

```

与通常一样，我们也可以交互地测试它，看看配置参数是如何应用的：

```

>>> from mytools import timer
>>> @timer(trace=False)                # No tracing, collect total time
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> listcomp
<mytools.Timer instance at 0x025C77B0>
>>> listcomp.alltime
0.0051938863738243413

>>> @timer(trace=True, label='\t=>')    # Turn on tracing
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
=> listcomp: 0.00155, 0.00155
>>> x = listcomp(5000)
=> listcomp: 0.00156, 0.00311
>>> x = listcomp(5000)
=> listcomp: 0.00174, 0.00486
>>> listcomp.alltime
0.0048562736325408196

```

这个计时函数装饰器可以用于任何函数，在模块中和在交互模式下都可以。换句话说，它自动获得作为脚本中计时代码的资格。查看本章后面的“实现私有属性”小节的装饰器参数的示例，以及在本章后面的“针对位置参数的一个基本范围测试装饰器”小节。

注意： 计时方法：本小节的计数器装饰器在任何函数上都有效，但是，要将其应用于类方法上，需要进行细小的改写。简而言之，正如我们在本章前面的“类错误之一：装饰类方法”小节所介绍的，必须避免使用一个嵌套的类。由于这一变化将会是我们本章末尾测试题的主题，所以在这里，我们暂时不会给出完整的解答。

编写类装饰器

到目前为止，我们已经编写了函数装饰器来管理函数调用，但是，正如我们已经见到的，Python 2.6和Python 3.0扩展了装饰器使其也能在类上有效。如同前面所提到的，尽管类似于函数装饰器的概念，但类装饰器应用于类——它们可以用于管理类自身，或者用来拦截实例创建调用以管理实例。和函数装饰器一样，类装饰器其实只是可选的语法糖，尽管很多人相信，它们使得程序员的意图更为明显并且能使不正确的调用最小化。

单体类

由于类装饰器可以拦截实例创建调用，所以它们可以用来管理一个类的所有实例，或者扩展这些实例的接口。为了说明这点，这里的第一个类装饰器示例做了前面一项工作——管理一个类的所有实例。这段代码实现了传统的**单体**编码模式，其中最多只有一个类的一个实例存在。其单体函数为管理的属性定义并返回一个函数，并且@语法自动在这个函数中包装了一个主体类：

```
instances = {}
def getInstance(aClass, *args):
    if aClass not in instances:
        instances[aClass] = aClass(*args)
    return instances[aClass]

def singleton(aClass):
    def onCall(*args):
        return getInstance(aClass, *args)
    return onCall
```

Manage global table
*# Add **kwargs for keywords*
One dict entry per class

On @ decoration
On instance creation

为了使用它，装饰用来强化单体模型的类：

```
@singleton
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Spam:
    def __init__(self, val):
        self.attr = val

bob = Person('Bob', 40, 10)
print(bob.name, bob.pay())

sue = Person('Sue', 50, 20)
print(sue.name, sue.pay())
```

Person = singleton(Person)
Rebinds Person to onCall
onCall remembers Person

Spam = singleton(Spam)
Rebinds Spam to onCall
onCall remembers Spam

Really calls onCall

Same, single object

```

X = Spam(42)                                # One Person, one Spam
Y = Spam(99)
print(X.attr, Y.attr)

```

现在，当Person或Spam类稍后用来创建一个实例的时候，装饰器提供的包装逻辑层把实例构建调用指向了onCall，它反过来调用getInstance，以针对每个类管理并分享一个单个实例，而不管进行了多少次构建调用。这段代码的输出如下：

```

Bob 400
Bob 400
42 42

```

有趣的是，这里如果能像前面介绍的那样，使用nonlocal语句（在Python 3.0及其以后版本中可用）来改变封闭的作用域名称，我们在这里可以编写一个更为自包含的解决方案——后面的替代方案实现了同样的效果，它为每个类使用了一个**封闭作用域**，而不是为每个类使用一个全局表入口：

```

def singleton(aClass):                        # On @ decoration
    instance = None
    def onCall(*args):                       # On instance creation
        nonlocal instance                   # 3.0 and later nonlocal
        if instance == None:
            instance = aClass(*args)        # One scope per class
        return instance
    return onCall

```

这个版本同样地工作，但是，它不依赖于装饰器之外的全局作用域中的名称。在Python 2.6或Python 3.0版本中，我们也可以用类编写一个自包含的解决方案——如下代码对每个类使用一个实例，而不是使用一个封闭作用域或全局表，并且它和其他的两个版本一样地工作（实际上，依赖于我们随后会见到的同样的编码模式是一个公用的装饰器类错误，这里我们只想要一个实例，但并不总是这样的情况）：

```

class singleton:
    def __init__(self, aClass):               # On @ decoration
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args):                # On instance creation
        if self.instance == None:
            self.instance = self.aClass(*args) # One instance per class
        return self.instance

```

要让这个装饰器成为一个完全通用的工具，可将其存储在一个可导入的模块文件中，在一个__name__检查下缩进self测试代码，并且在构建调用中使用**kwargs语法支持关键字参数（我们将在建议的练习中讨论它）。

跟踪对象接口

前面小节的单体验例使用类装饰器来管理一个类的**所有**实例。类装饰器的另一个常用场景是**每个**产生实例的接口。类装饰器基本上可以在实例上安装一个包装器逻辑层，来以某种方式管理对其接口的访问。

例如，在第30章中，`__getattr__`运算符重载方法作为包装嵌入的实例的整个对象接口的一种方法，以便实现委托编码模式。我们在前一章介绍的管理的属性中看到过类似的例子。还记得吧，当获取未定义的属性名的时候，`__getattr__`会运行；我们可以使用这个钩子来拦截一个控制器类中的方法调用，并将它们传递给一个嵌入的对象。

为了便于参考，这里给出最初的非装饰器委托示例，它在两个内置类型对象上工作：

```
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

>>> 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
>>> list(x.keys())                             # Delegate to dictionary method
Trace: keys                                  # Use list() in 3.0
['a', 'b']
```

在这段代码中，`Wrapper`类拦截了对任何包装的对象的属性的访问，打印出一条跟踪信息，并且使用内置函数`getattr`来终止对包装对象的请求。它特别跟踪包装的对象的类之外发出的属性访问。在包装的对象内部访问其方法不会被捕获，并且会按照设计正常运行。这种整体接口模型和函数装饰器的行为不同，装饰器只包装一个特定的方法。

类装饰器为编写这种`__getattr__`技术来包装一个完整接口提供了一个替代的、方便的方法。例如，在Python 2.6和Python 3.0中，前面的类示例可能编写为一个类装饰器，来触发包装的实例创建，而不是把一个预产生的实例传递到包装器的构造函数中（在这里也用`**kwargs`扩展了，以支持关键字参数，并且统计进行访问的次数）：

```
def Tracer(aClass):
    class Wrapper:
        def __init__(self, *args, **kwargs):
            self.fetches = 0
            self.wrapped = aClass(*args, **kwargs)
        def __getattr__(self, attrname):
```

```

        print('Trace: ' + attrname)                # Catches all but own attrs
        self.fetches += 1
        return getattr(self.wrapped, attrname)     # Delegate to wrapped obj
    return Wrapper

@Tracer
class Spam:
    def display(self):
        print('Spam!' * 8)
    # Spam = Tracer(Spam)
    # Spam is rebound to Wrapper

@Tracer
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate
    # Person = Tracer(Person)
    # Wrapper remembers Person
    # Accesses outside class traced
    # In-method accesses not traced

food = Spam()
food.display()
print([food.fetches])
# Triggers Wrapper()
# Triggers __getattr__

bob = Person('Bob', 40, 50)
print(bob.name)
print(bob.pay())
# bob is really a Wrapper
# Wrapper embeds a Person

print('')
sue = Person('Sue', rate=100, hours=60)
print(sue.name)
print(sue.pay())
# sue is a different Wrapper
# with a different Person

print(bob.name)
print(bob.pay())
print([bob.fetches, sue.fetches])
# bob has different state
# Wrapper attrs not traced

```

这里与我们前面在“编写函数装饰器”一节中遇到的跟踪器装饰器有很大不同，注意到这点很重要，在那里，我们看到了装饰器可以使我们跟踪和计时对一个给定函数或方法的调用。相反，通过拦截实例创建调用，这里的类装饰器允许我们跟踪整个对象接口，例如，对其任何属性的访问。

下面是这段代码在Python 2.6和Python 3.0下的输出：Spam和Person类的实例上的属性获取都会调用Wrapper类中的__getattr__逻辑，由于food和bob确实都是Wrapper的实例，得益于装饰器的实例创建调用重定向：

```

Trace: display
Spam! Spam! Spam! Spam! Spam! Spam! Spam!
[1]
Trace: name
Bob
Trace: pay
2000

```

```

Trace: name
Sue
Trace: pay
6000
Trace: name
Bob
Trace: pay
2000
[4, 2]

```

注意，前面的代码装饰了一个用户定义的类。就像是在本书第30章最初的例子中一样，我们也可以使用装饰器来包装一个内置的类型，例如列表，只要我们的子类允许装饰器语法，或者手动地执行装饰——装饰器语法对于@行需要一条class语句。

在下面的代码中，由于装饰的间接作用，x实际是一个Wrapper（我把装饰器类放到了模块文件中，以便以这种方式重用它）：

```

>>> from tracer import Tracer           # Decorator moved to a module file

>>> @Tracer
... class MyList(list): pass             # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])                # Triggers Wrapper()
>>> x.append(4)                          # Triggers __getattr__.append
Trace: append
>>> x.wrapped
[1, 2, 3, 4]

>>> WrapList = Tracer(list)              # Or perform decoration manually
>>> x = WrapList([4, 5, 6])              # Else subclass statement required
>>> x.append(7)
Trace: append
>>> x.wrapped
[4, 5, 6, 7]

```

这种装饰器方法允许我们把实例创建移动到装饰器自身之中，而不是要求传入一个预先生成的对象。尽管这好像是一个细小的差别，它允许我们保留常规的实例创建语法并且通常实现装饰器的所有优点。我们只需要用装饰器语法来扩展类，而不是要求所有的实例创建调用都通过一个包装器来手动地指向对象：

```

@Tracer                                  # Decorator approach
class Person: ...
    bob = Person('Bob', 40, 50)
    sue = Person('Sue', rate=100, hours=60)

class Person: ...                        # Non-decorator approach
    bob = Wrapper(Person('Bob', 40, 50))
    sue = Wrapper(Person('Sue', rate=100, hours=60))

```

假设你将会产生类的多个实例，装饰器通常将会在代码大小和代码可维护性上双赢。

注意： 属性版本差异：正如我们在本书第37章所了解到的，在Python 2.6中，`__getattr__` 将会拦截对`__str__` 和 `__repr__` 这样的运算符重载方法的访问，但是，在Python 3.0中不会这样。

在Python 3.0中，类实例会从类中继承这些方法中的一些（而不是全部）的默认形式（实际上，是从自动对象超类），因为所有的类都是“新式的”。此外，在Python 3.0中，针对打印和+这样的内置操作显式地调用属性并不会通过`__getattr__`（或其近亲`__getattribute__`）路由。新式类在类中查找这样的方法，并且完全省略常规的实例查找。

此处意味着，在Python 2.6中，基于`__getattr__`的跟踪包装器将会自动跟踪和传递运算符重载，但是，在Python 3.0中不会如此。要看到这一点，直接在交互式会话的前面的末尾显示“x”，在Python 2.6中，属性`__repr__`被跟踪并且该列表如预期的那样打印出来，但是在Python 3.0中，不会发生跟踪并且列表打印为Wrapper类使用一个默认显示：

```
>>> x                                     # 2.6
Trace: __repr__
[4, 5, 6, 7]
>>> x                                     # 3.0
<tracer.Wrapper object at 0x026C07D0>
```

要在Python 3.0中同样工作，运算符重载方法通常需要在包装类中冗余地重新定义，要么手动定义，要么通过工具定义，或者通过在超类中定义。只有简单命名的属性会在两种版本中都同样工作。我们将在本章稍后的一个Private装饰器中再次看到版本差异的作用。

类错误之二：保持多个实例

令人好奇的是，这个例子中的装饰器函数几乎可以编写为一个类而不是一个函数，使用正确的运算符重载协议。如下的略微简化的替代版本类似地工作，因为当@装饰器应用于类的时候，触发`__init__`，并且当创建了一个主体类实例的时候触发其`__call__`。这次，我们的对象实际是Tracer的实例，并且这里，我们实际上只是为避免使用对一个实例属性的封闭作用域引用：

```
class Tracer:
    def __init__(self, aClass):           # On @decorator
        self.aClass = aClass             # Use instance attribute
    def __call__(self, *args):            # On instance creation
        self.wrapped = self.aClass(*args) # ONE (LAST) INSTANCE PER CLASS!
        return self
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer                                   # Triggers __init__
class Spam:                               # Like: Spam = Tracer(Spam)
    def display(self):
        print('Spam!' * 8)

...
```

```

food = Spam()                                # Triggers __call__
food.display()                               # Triggers __getattr__

```

正如我们在前面见到的，这个仅针对类的替代方法像前面一样处理多个类，但是，它对于一个给定的类的**多个实例**并不是很有效：每个实例构建调用会触发`__call__`，这会覆盖前面的实例。直接效果是`Tracer`只保存了一个实例，即最后创建的一个实例。自行体验一下看看这是如何发生的，但是，这里给出该问题的一个示例：

```

@Tracer
class Person:                                # Person = Tracer(Person)
    def __init__(self, name):                 # Wrapper bound to Person
        self.name = name

bob = Person('Bob')                           # bob is really a Wrapper
print(bob.name)                               # Wrapper embeds a Person
Sue = Person('Sue')
print(sue.name)                               # sue overwrites bob
print(bob.name)                              # OOPS: now bob's name is 'Sue!'

```

这段代码输出如下——由于这个跟踪器只有一个共享的实例，所以第二个实例覆盖了第一个实例：

```

Trace: name
Bob
Trace: name
Sue
Trace: name
Sue

```

这里的问题是一个糟糕的**状态保持**——我们为每个类创建了一个装饰器实例，但是不是针对每个类实例，这样一来，只有最后一个实例保持住了。其解决方案就像我们在前面针对装饰方法的类错误一样，在于放弃基于类的装饰器。

前面的基于函数的`Tracer`版本确实可用于多个实例，因为每个实例构建调用都会创建一个新的`Wrapper`实例，而不是覆盖一个单独的共享的`Tracer`实例的状态。由于同样的原因，最初的非装饰器版本正确地处理多个实例。装饰器不仅仅具有无可争辩的魔力，而且微妙的程度令人难以置信。

装饰器与管理器函数的关系

无顾这样的微妙性，`Tracer`类装饰器示例最终仍然是依赖于`__getattr__`来拦截对一个包装和嵌入实例对象的获取。正如我们在前面见到的，我们真正需要完成的只是，把实例创建调用移入一个类的内部，而不是把实例传递一个管理器函数。对于最初的非装饰器跟踪实例，我们将直接有差异地编写实例创建：


```

class Spam:                                # Non-decorator version
    ...                                    # Any class will do
    food = Wrapper(Spam())                # Special creation syntax

@Tracer
class Spam:                                # Decorator version
    ...                                    # Requires @ syntax at class
    food = Spam()                        # Normal creation syntax

```

基本上，**类装饰器**把特殊语法需求从实例创建调用迁移到了类语句自身。对于本节前面的单体示例来说也是如此，我们直接将类及其构建参数传递到了一个管理器函数中，而不是装饰一个类并使用常规的实例创建调用：

```

instances = {}
def getInstance(aClass, *args):
    if aClass not in instances:
        instances[aClass] = aClass(*args)
    return instances[aClass]

bob = getInstance(Person, 'Bob', 40, 10)    # Versus: bob = Person('Bob', 40, 10)

```

作为替代方案，我们可以使用Python的内省工具来从一个已准备创建好的实例来获取类（假设创建一个初始化实例是可以接受的）：

```

instances = {}
def getInstance(object):
    aClass = object.__class__
    if aClass not in instances:
        instances[aClass] = object
    return instances[aClass]

bob = getInstance(Person('Bob', 40, 10))    # Versus: bob = Person('Bob', 40, 10)

```

这对于我们前面所编写的跟踪器那样的**函数装饰器**也是成立的：我们可以直接把函数及其参数传递到负责分配调用的一个管理器中，而不是用拦截随后调用的逻辑来装饰一个函数：

```

def func(x, y):                             # Nondecorator version
    ...                                     # def tracer(func, args): ... func(*args)
    result = tracer(func, (1, 2))          # Special call syntax

@tracer
def func(x, y):                             # Decorator version
    ...                                   # Rebinds name: func = tracer(func)
    result = func(1, 2)                   # Normal call syntax

```

像这样的管理器函数的方法把使用特殊语法的负担放到了调用上，而不是期待在函数和类定义上使用装饰语法。

为什么使用装饰器（重访）

那么，为什么我们只是展示不使用装饰器的方法来实现单体呢？正如我在本章开始的时候提到的，装饰器展示给我们利弊权衡。尽管语法意义重大，当面对新工具的时候，我们通常都忘了问“为什么要用”的问题。既然已经看到了装饰器实际是如何工作的，让我们花点时间在这里看看更大的问题。

就像大多数语言功能一样，装饰器也有优点和缺点。例如，从负面的角度讲，**类装饰器**有两个潜在的缺陷：

类型修改

正如我们所见到的，当插入包装器的时候，一个装饰器函数或类不会保持其**最初**的**类型**——其名称重新绑定到一个包装器对象，在使用对象名称或测试对象类型的程序中，这可能会很重要。在单体的例子中，装饰器和管理函数的方法都为实例保持了最初的类类型；在跟踪器的代码中，没有一种方法这么做，因为需要有包装器。

额外调用

通过装饰添加一个包装层，在每次调用装饰对象的时候，会引发一次**额外调用**所需的额外性能成本——调用是相对耗费时间的操作，因此，装饰包装器可能会使程序变慢。在跟踪器代码中，两种方法都需要每个属性通过一个包装器层来指向；单体的示例通过保持最初的类类型而避免了额外调用。

类似的问题也适用于**函数装饰器**：装饰和管理器函数都会导致额外调用，并且当装饰的时候通常会发生类型变化（不装饰的时候就没有）。

也就是说，这二者都不是非常严重的问题。对于大多数程序来说，类型差异问题不可能有关系，并且额外调用对速度的影响也不显著；此外，只有当使用包装器的时候才会产生后一个问题，且这个问题常常可以忽略，因为需要优化性能的时候可以直接删除装饰器，并且添加包装逻辑的非装饰器解决方案也会导致额外调用的问题（包括我们将在第39章学习的元类）。

相反，正如我们在本章开始所见到的，装饰器有3个主要优点。与前面小节的管理器（即辅助）函数解决方案相比，装饰器提供：

明确的语法

装饰器使得扩展明确而显然。它们的@比可能在源文件中任何地方出现的特殊代码要容易识别，例如，在单体和跟踪器实例中，装饰器行似乎比额外代码更容易被注意到。此外，装饰器允许函数和实例创建调用使用所有Python程序员所熟悉的常规语法。

代码可维护性

装饰器避免了在每个函数或类调用中重复扩展的代码。由于它们只出现一次，在类或者函数自身的定义中，它们排除了冗余性并简化了未来的代码维护。对于我们的单体和跟踪器示例，要使用管理器函数的方法，我们需要在每次调用的时候使用特殊的代码——最初以及未来必须做出的任何修改都需要额外的工作。

一致性

装饰器使得程序员忘记使用必需的包装逻辑的可能性大大减少。这主要得益于两个优点——由于装饰是显式的并且只出现一次，出现在装饰的对象自身中，与必须包含在每次调用中的特殊代码相比较，装饰器促进了更加一致和统一的API使用。例如，在单体示例中，可能更容易忘了通过特殊代码来执行所有类创建调用，而这将会破坏单体的一致性管理。

装饰器还促进了代码的封装以减少冗余性，并使得未来的维护代价最小化。尽管其他的编码结构化工具也能做到这些，但装饰器使得这对于扩展任务来说更自然。

然而，这三个优点还不是使用装饰器语法的必需的原因，装饰器的用法最终还是一个格式选择。也就是说，大多数程序员发现了一个纯粹的好处，特别是它作为正确使用库和API的一个工具。

我还记得类中的构造函数函数的支持者和反对者也有过类似的争论——在介绍`__init__`方法之前，创建它的时候通过一个方法手动地运行一个实例，往往也能实现同样的效果（例如，`x=Class().init()`）。然而，随着时间的流逝，尽管这基本上是一个格式的选择，但`__init__`语法也变成了广泛的首选，因为它更为明确、一致和可维护。尽管这应该由你来决定，但装饰器似乎把很多同样的成功摆到了桌面上。

直接管理函数和类

本章中，我们的大多数示例都设计来拦截函数和实例创建调用。尽管这对于装饰器来说很典型，它们并不限于这一角色。因为装饰器通过装饰器代码来运行新的函数和类，从而有效地工作，它们也可以用来管理函数和类对象自身，而不只是管理对它们随后的调用。

例如，假设你需要被另一个应用程序使用的方法或类注册到一个API，以便随后处理（可能该API随后将会调用该对象，以响应事件）。尽管你可能提供一个注册函数，在对象定义之后手动地调用该函数，但装饰器使得你的意图更为明显。

这一思路如下的简单实现定义了一个装饰器，它既应用于函数也应用于类，把对象添加

到一个基于字典的注册中。由于它返回对象本身而不是一个包装器，所以它没有拦截随后的调用：

```
# Registering decorated objects to an API

registry = {}
def register(obj):
    registry[obj.__name__] = obj
    return obj

    # Both class and func decorator
    # Add to registry
    # Return obj itself, not a wrapper

@register
def spam(x):
    return(x ** 2)

    # spam = register(spam)

@register
def ham(x):
    return(x ** 3)

@register
class Eggs:
    def __init__(self, x):
        self.data = x ** 4
    def __str__(self):
        return str(self.data)

    # Eggs = register(Eggs)

print('Registry:')
for name in registry:
    print(name, '=>', registry[name], type(registry[name]))

print('\nManual calls:')
print(spam(2))
print(ham(2))
X = Eggs(2)
print(X)

    # Invoke objects manually
    # Later calls not intercepted

print('\nRegistry calls:')
for name in registry:
    print(name, '=>', registry[name](3)) # Invoke from registry
```

当这段代码运行的时候，装饰的对象按照名称添加到注册中，但当随后调用它们的时候，它们仍然按照最初的编码工作，而没有指向一个包装器层。实际上，我们的对象可以手动运行，或从注册表内部运行：

```
Registry:
Eggs => <class '__main__.Eggs'> <class 'type'>
ham => <function ham at 0x02CFB738> <class 'function'>
spam => <function spam at 0x02CFB6F0> <class 'function'>

Manual calls:
4
8
16

Registry calls:
```

```
Eggs => 81
ham  => 27
spam => 9
```

例如，一个用户界面可能使用这样的技术，为用户动作注册回调处理程序。处理程序可能通过函数或类名来注册，就像这里所做的一样，或者可以使用装饰器参数来指定主体事件；包含装饰器的一条额外的def语句可能会用来保持这样的参数以便在装饰时使用。

这个例子是仿造的，但是，其技术很通用。例如，函数装饰器也可能用来处理函数属性，并且类装饰器可能动态地插入新的类属性，或者甚至新的方法。考虑如下的函数装饰器——它们把函数属性分配给记录信息，以便随后供一个API使用，但是，它们没有插入一个包含器层来拦截随后的调用：

```
# Augmenting decorated objects directly

>>> def decorate(func):
...     func.marked = True                # Assign function attribute for later use
...     return func
...
>>> @decorate
... def spam(a, b):
...     return a + b
...
>>> spam.marked
True

>>> def annotate(text):                  # Same, but value is decorator argument
...     def decorate(func):
...         func.label = text
...         return func
...     return decorate
...
>>> @annotate('spam data')
... def spam(a, b):                      # spam = annotate(...) (spam)
...     return a + b
...
>>> spam(1, 2), spam.label
(3, 'spam data')
```

这样的装饰器直接扩展了函数和类，没有捕捉对它们的随后调用。我们将在下一章见到更多的管理类、类装饰的例子，因为这证明了它已经转向了元类的领域；在本章剩余的部分，我们来看看使用装饰器的两个较大的案例。

示例：“私有”和“公有”属性

本章的最后两个小节介绍了使用装饰器的两个较大的例子。这两个例子都用尽量少的说明来展示，部分是由于本章的篇幅已经超出了限制，但主要是因为你应该已经很好地理

解了装饰器的基础知识，足够能够自行研究这些例子。作为通用用途的工具，这些例子使我们有机会来看看装饰器的概念如何融入到更为有用的代码中。

实现私有属性

如下的类装饰器实现了一个用于类实例属性的`Private`声明，也就是说，属性存储在一个实例上，或者从其一个类继承而来。不接受从装饰的类的外部对这样的属性的获取和修改访问，但是，仍然允许类自身在其方法中自由地访问那些名称。它不是具体的C++或Java，但它提供了类似的访问控制作为Python中的选项。

在第29章，我们见到了实例属性针对修改成为私有的不完整的、粗糙的实现。这里的版本扩展了这一概念以验证属性获取，并且它使用委托而不是继承来实现该模型。实际上，在某种意义上，这只是我们前面遇到的属性跟踪器类装饰器的一个扩展。

尽管这个例子利用了类装饰器的新语法糖来编写私有属性，但它的属性拦截最终仍然是基于我们在前面各章介绍的`__getattr__`和`__setattr__`运算符重载方法。当检测到访问一个私有属性时，这个版本使用`raise`语句引发一个异常，还有一条出错消息；异常可能在一个`try`中捕获，或者允许终止脚本。

代码如下所示，在文件的底部还有一个self测试。它在Python 2.6和Python 3.0下都能够工作，因为它使用了Python 3.0的`print`和`raise`语法，尽管它在Python 2.6下只是捕获运算符重载方法属性（稍后更多讨论这一点）：

```
"""
Privacy for attributes fetched from class instances.
See self-test code at end of file for a usage example.
Decorator same as: Doubler = Private('data', 'size')(Doubler).
Private returns onDecorator, onDecorator returns onInstance,
and each onInstance instance embeds a Doubler instance.
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def Private(*privates):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):
                trace('get:', attr)
                if attr in privates:
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.wrapped, attr)
            def __setattr__(self, attr, value):
                trace('set:', attr, value)
        return onInstance
    return onDecorator(aClass)
```

```

        trace('set:', attr, value)                # Others run normally
        if attr == 'wrapped':                     # Allow my attrs
            self.__dict__[attr] = value           # Avoid looping
        elif attr in privates:
            raise TypeError('private attribute change: ' + attr)
        else:
            setattr(self.wrapped, attr, value)     # Wrapped obj attrs
    return onInstance                             # Or use __dict__
return onDecorator

if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')                      # Doubler = Private(...)(Doubler)
    class Doubler:
        def __init__(self, label, start):
            self.label = label                    # Accesses inside the subject class
            self.data = start                     # Not intercepted: run normally
        def size(self):
            return len(self.data)                 # Methods run with no checking
        def double(self):                         # Because privacy not inherited
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2
        def display(self):
            print('%s => %s' % (self.label, self.data))

    X = Doubler('X is', [1, 2, 3])
    Y = Doubler('Y is', [-10, -20, -30])

    # The following all succeed
    print(X.label)                               # Accesses outside subject class
    X.display(); X.double(); X.display()          # Intercepted: validated, delegated
    print(Y.label)
    Y.display(); Y.double()
    Y.label = 'Spam'
    Y.display()

    # The following all fail properly
    """
    print(X.size())                             # prints "TypeError: private attribute fetch: size"
    print(X.data)
    X.data = [1, 1, 1]
    X.size = lambda S: 0
    print(Y.data)
    print(Y.size())
    """

```

当traceMe为True的时候，模块文件的self测试代码产生如下的输出。注意，装饰器是如何捕获和验证在包装的类之外运行的属性获取和赋值的，但是，却没有捕获类自身内部的属性访问：

```

[set: wrapped <__main__.Doubler object at 0x02B2AAF0>]
[set: wrapped <__main__.Doubler object at 0x02B2AE70>]
[get: label]

```

```
X is
[get: display]
X is => [1, 2, 3]
[get: double]
[get: display]
X is => [2, 4, 6]
[get: label]
Y is
[get: display]
Y is => [-10, -20, -30]
[get: double]
[set: label Spam]
[get: display]
Spam => [-20, -40, -60]
```

实现细节之一

这段代码有点复杂，并且你最好自己跟踪运行它，看看它是如何工作的。然而，为了帮助你理解，这里给出一些值得注意的提示。

继承与委托的关系

第29章中给出的粗糙的私有示例使用**继承**来混入__setattr__捕获访问。然而，继承使得这很困难，因为从类的内部或外部的访问之间的区分不是很直接的（内部访问应该允许常规运行，并且外部的访问应该限制）。要解决这个问题，第29章的示例需要继承类，以使用__dict__赋值来设置属性，这最多是一个不完整的解决方案。

这里的版本使用的**委托**（在另一个对象中嵌入一个对象），而不是继承。这种模式更好地适合于我们的任务，因为它使得区分主体对象的内部访问和外部访问容易了很多。对主体对象的来自外部的属性访问，由包装器层的重载方法拦截，并且如果合法的话，委托给类。类自身内部的访问（例如，通过其方法代码内的self）没有拦截并且允许不经检查而常规运行，因为这里没有继承私有的属性。

装饰器参数

这里使用的类装饰器接受任意多个参数，以命名私有属性。然而，真正发生的情况是，参数传递给了Private函数，并且Private返回了应用于主体类的装饰器函数。也就是说，在装饰器发生之前使用这些参数；Private返回装饰器，装饰器反过来把私有的列表作为一个封闭作用域应用来“记住”。

状态保持和封闭作用域

说到封闭的作用域，在这段代码中，实际上用到了3个层级的状态保持：

- `Private`的参数在装饰发生前使用，并且作为一个封闭作用域引用保持，以用于`onDecorator`和`onInstance`中。
- `onDecorator`的类参数在装饰时使用，并且作为一个封闭作用域引用保持，以便在实例构建时使用。
- 包装的实例对象保存为`onInstance`中的一个实例属性，以便随后从类外部访问属性的时候使用。

由于Python的作用域和命名空间规则，这些都很自然地工作。

使用`__dict__`和`__slots__`

这段代码中`__setattr__`依赖于一个实例对象的`__dict__`属性命名空间字典，以设置`onInstance`自己的包装属性。正如我们在上一章所了解到的，不能直接赋值一个属性而避免循环。然而，它使用了`setattr`内置函数而不是`__dict__`来设置包装对象自身之中的属性。此外，`getattr`用来获取包装对象中的属性，因为它们可能存储在对象自身中或者由对象继承。

因此，这段代码将对大多数类有效。你可能还记得，在第31章中介绍过，带有`__slots__`的新式类不能把属性存储到一个`__dict__`中。然而，由于我们在这里只是在`onInstance`层级依赖于一个`__dict__`，而不是在包装的实例中，并且因为`setattr`和`getattr`应用于基于`__dict__`和`__slots__`的属性，所以我们的装饰器应用于使用任何一种存储方案的类。

公有声明的泛化

既然有了一个`Private`实现，泛化其代码以考虑`Public`声明就很简单了——它们基本上是`Private`声明的反过程，因此，我们只需要取消内部测试。本节列出的实例允许一个类使用装饰器来定义一组`Private`或`Public`的实例属性（存储在一个实例上的属性，或者从其类继承的属性），使用如下的语法：

- `Private`声明类实例的那些不能获取或赋值的属性，而从类的方法的代码内部获取或赋值除外。也就是说，任何声明为`Private`的名称都不能从类的外部访问，而任何没有声明为`Private`的名称都可以自由地从类的外部获取或赋值。
- `Public`声明了一个类的实例属性，它可以从类的外部以及在类的方法内部获取和访问。也就是说，声明为`Public`的任何名称，可以从任何地方自由地访问，而没有声明为`Public`的任何名称，不能从类的外部访问。

`Private`和`Public`声明规定为互斥的：当使用了`Private`，所有未声明的名称都被认为

是Public的；并且当使用了Public，所有未声明的名称都被认为是Private。它们基本上相反，尽管未声明的、不是由类方法创建的名称行为略有不同——它们可以赋值并且由此从类的外部在Private之下创建（所有未声明的名称都是可以访问的），但不是在Public下创建的（所有未声明的名称都是不可访问的）。

再一次，自己研究这些代码并体验它们是如何工作的。注意，这个方案在顶层添加了额外的第四层状态保持，超越了前面描述的3个层次：lambda所使用的测试函数保存在一个额外的封闭作用域中。这个示例编写为可以在Python 2.6或Python 3.0下运行，尽管它在Python 3.0下运行的时候带有一个缺陷（在文件的文档字符串之后简短地说明，并且在代码之后详细说明）：

```
"""
Class decorator with Private and Public attribute declarations.
Controls access to attributes stored on an instance, or inherited
by it from its classes. Private declares attribute names that
cannot be fetched or assigned outside the decorated class, and
Public declares all the names that can. Caveat: this works in
3.0 for normally named attributes only: __X__ operator overloading
methods implicitly run for built-in operations do not trigger
either __getattr__ or __getattribute__ in new-style classes.
Add __X__ methods here to intercept and delegate built-ins.
"""

traceMe = False
def trace(*args):
    if traceMe: print('[ ' + ' '.join(map(str, args)) + ' ]')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kwargs):
                self.__wrapped = aClass(*args, **kwargs)
            def __getattr__(self, attr):
                trace('get:', attr)
                if failIf(attr):
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
            def __setattr__(self, attr, value):
                trace('set:', attr, value)
                if attr == '_onInstance_wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
```

```
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

参见前面示例的self测试代码，它是一个用法示例。这里在交互提示模式下快速地看看这些类装饰器的使用（它们在Python 2.6和Python 3.0下一样地工作），正如所介绍的那样，非Private或Public名称可以从主体类之外访问和修改，但是Private或非Public的名称不可以：

```
>>> from access import Private, Public

>>> @Private('age')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
...                                     # Person = Private('age')(Person)
...                                     # Person = onInstance with state
>>> X = Person('Bob', 40)
>>> X.name
'Bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tom'
TypeError: private attribute change: age

>>> @Public('name')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
...                                     # X is an onInstance
>>> X = Person('bob', 40)
>>> X.name
'bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tom'
TypeError: private attribute change: age
```

实现细节之二

为了帮助你分析这段代码，这里有一些关于这一版本的最后提示。由于这只是前面小节的示例的泛化，所以那里的大多数提示也适用于这里。

使用__X伪私有名称

除了泛化，这个版本还使用了Python的__X伪私有名称压缩功能（我们在第30章遇到过），来把包装的属性局部化为控制类，通过自动将其作为类名的前缀就可以做到。这避免了前面的版本与一个真实的、包装类可能使用的包装属性冲突的风险，并且它也是一个有用的通用工具。然而，它不是很“私有”，因为压缩的名称可以在类之外自由地使用。注意，在__setattr__中，我们也必须使用完整扩展的名称字符串('__onInstance_wrapped')，因为这是Python对其的修改。

破坏私有

尽管这个例子确实实现了对一个实例及其类的属性的访问控制，它可能以各种方式破坏了这些控制——例如，通过检查包装属性的显式扩展版本（bob.pay可能无效，因为完全压缩的bob._onInstance_wrapped.pay可能会有效）。如果你必须显式地这么做，这些控制可能对于常规使用来说足够了。当然，私有控制通常在任何语言中都会遭到破坏，如果你足够努力地尝试的话（#define private public在某些C++实现中也可能有效）。尽管访问控制可以减少意外修改，但这样的情况大多取决于使用任何语言的程序员。不管何时，源代码可能会被修改，访问控制总是管道流中的一小部分。

装饰器权衡

不用装饰器，我们也可以实现同样的结果，通过使用管理函数或者手动编写装饰器的名称重绑定；然而，装饰器语法使得代码更加一致而明确。这一方法以及任何其他基于包装的方法的主要潜在缺点是，属性访问导致额外调用，并且装饰的类的实例并不真的是最初的装饰类的实例——例如，如果你用X.__class__或isinstance(X, C)测试它们的类型，将会发现它们是包装类的实例。除非你计划在对象类型上进行内省，否则类型问题可能是不相关的。

开放问题

还是老样子，这个示例设计为在Python 2.6和Python 3.0下都能工作（提供了运算符重载方法，以便在包装器中重定义委托）。然而，和大多数软件一样，总是有改进的地方。

缺陷：运算符重载方法无法在Python 3.0下委托

就像使用__getattr__的所有的基于委托的类，这个装饰器只对常规命名的属性能够跨版本工作。像__str__和__add__这样在新式类下不同工作的运算符方法，在Python 3.0下运行的时候，如果定义了嵌入的对象，将无法有效到达。

正如我们在上一章所了解到的，传统类通常在运行时在实例中查找运算符重载名

称，但新式类不这么做——它们完全略过实例，在类中查找这样的方法。因此，在Python 2.6的新式类和Python 3.0的所有类中，`__X__`运算符重载方法显式地针对内置操作运行，不会触发`__getattr__`和`__getattribute__`。这样的属性获取将会和我们的`onInstance.__getattr__`一起忽略，因此，它们无法验证或委托。

我们的装饰器类没有编写为新式类（通过派生自`object`），因此，如果在Python 2.6下运行，它将会捕获运算符重载方法。由于在Python 3.0下所有的类自动都是新式类，如果它们在嵌入的对象上编码，这样的方法将会失效。Python 3.0中最简单的解决方案是，在`onInstance`中重新冗余地定义所有那些可能在包装的对象中用到的运算符重载方法。例如，可以手动添加额外的方法，可以通过工具来自动完成部分任务（例如，使用类装饰器或者下一章将要介绍的元类），或者通过在超类中定义。

要亲自看到不同，可尝试在Python 2.6下对使用运算符重载方法的一个类使用该装饰器。验证与前面一样有效，但是打印所使用的`__str__`方法和为+而运行的`__add__`方法二者都会调用装饰器的`__getattr__`，并由此最终将验证并正确地委托给主体`Person`对象：

```
C:\misc> c:\python26\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
...     def __str__(self):
...         return 'Person: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()
>>> X.age                                     # Name validations fail correctly
TypeError: private attribute fetch: age
>>> print(X)                                  # __getattr__ => runs Person.__str__
Person: 42
>>> X + 10                                     # __getattr__ => runs Person.__add__
>>> print(X)                                  # __getattr__ => runs Person.__str__
Person: 52
```

同样的代码在Python 3.0下运行的时候，显式地调用`__str__`和`__add__`将会忽略装饰器的`__getattr__`，并且在装饰器类之中或其上查找定义；`print`最终查找到从类类型继承的默认显示（从技术上讲，是从Python 3.0中隐藏的`object`超类），并且+产生一个错误，因为没有默认继承：

```
C:\misc> c:\python30\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
```

```

...     self.age = 42
...     def __str__(self):
...         return 'Person: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()                                # Name validations still work
>>> X.age                                         # But 3.0 fails to delegate built-ins!
TypeError: private attribute fetch: age
>>> print(X)
<access.onInstance object at 0x025E0790>
>>> X + 10
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
>>> print(X)
<access.onInstance object at 0x025E0790>

```

使用替代的 `__getattr__` 方法在这里帮不上忙——尽管它定义为捕获每次属性引用（而不只是未定义的名称），它也不会由内置操作运行。我们在本书第37章介绍的Python的特性功能，在这里也帮不上忙。回忆一下，特性自动运行与在编写类的时候定义的特定属性相关的代码，并且不会设计来处理包装对象中的任意属性。

正如前面所提到的，Python 3.0中最直接的解决方案是：在类似装饰器的基于委托的类中，冗余地重新定义可能在嵌入对象中出现的运算符重载名称。这种方法并不理想，因为它产生了一些代码冗余，特别是与Python 2.6的解决方案相比较尤其如此。然而，这不会有太大的编码工作，在某种程度上可以使用工具或超类来自动完成，足以使装饰器在Python 3.0下工作，并且也允许运算符重载名称声明为Private或Public（假设每个运算符重载方法内部都运行failIf测试）：

```

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kwargs):
                self.__wrapped = aClass(*args, **kwargs)

            # Intercept and delegate operator overloading methods
            def __str__(self):
                return str(self.__wrapped)
            def __add__(self, other):
                return self.__wrapped + other
            def __getitem__(self, index):
                return self.__wrapped[index]          # If needed
            def __call__(self, *args, **kwargs):
                return self.__wrapped(*arg, *kwargs)   # If needed
            ...plus any others needed...

            # Intercept and delegate named attributes
            def __getattr__(self, attr):
                ...
            def __setattr__(self, attr, value):
                ...
        return onInstance
    return onDecorator

```

```
        return onInstance
    return onDecorator
```

添加了这样的运算符重载方法，前面带有`__str__`和`__add__`的示例在Python 2.6和Python 3.0下都能同样地工作，尽管在Python 3.0下可能需要增加大量的额外代码——从原则上讲，**每个**不会自动运行的运算符重载方法，都需要在这样的一个通用工具类中针对Python 3.0冗余地定义（这就是为什么我们的代码省略这一扩展）。由于在Python 3.0中每个类都是新式的，所以在这一版本中，基于委托的代码更加困难（尽管不是没有可能）。

另一方面，委托包装器可以直接从一个曾经重定义了运算符重载方法的公共超类继承，使用标准的委托代码。此外，像额外的类装饰器或元类这样的工具，可能会自动地向委托类添加这样的方法，从而使一部分工作自动化（参见第39章中类扩展的示例以了解更多信息）。尽管还是不像Python 2.6中的解决方案那样简单，这样的技术能够帮助Python 3.0的委托类更加通用。

实现替代：__getattr__插入，调用堆栈检查

尽管在包装器中冗余地定义运算符重载方法可能是前面介绍的Python 3.0难题的最直接解决方案，但它不是唯一的方法。我们没有足够篇幅来更深入地介绍这一问题，因此，研究其他潜在的解决方案就放在了一个建议的练习中。由于一个棘手的替代方案非常强调类概念，因此这里简单提及一下其优点。

这个示例的一个缺点是，实例对象并不真的是最初的类的实例——它们是包装器的实例。在某些依赖于类型测试的程序中，这可能就有麻烦。为了支持这样的类，我们可能试图通过在最初类中插入一个`__getattr__`方法来实现类似的效果，以捕获在其实例上的**每次**属性引用。插入的方法将会把有效的请求向上传递到其超类以避免循环，使用我们在前面一章学习过的技术。如下是对类装饰器代码的潜在修改：

```
# trace support as before

def accessControl(failIf):
    def onDecorator(aClass):
        def getattributes(self, attr):
            trace('get:', attr)
            if failIf(attr):
                raise TypeError('private attribute fetch: ' + attr)
            else:
                return object.__getattr__(self, attr)
        aClass.__getattr__ = getattributes
        return aClass
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
```

```
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

这一替代方案解决了类型测试的问题，但是带来了其他的问题。例如，它只处理属性获取——也就是，这个版本允许自由地对私有名称赋值。拦截赋值仍然必须使用 `__setattr__`，或者是一个实例包装器对象，或者插入另一个类方法。添加一个实例包装器来捕获赋值可能会再次改变类型，并且如果最初的类使用自己的 `__setattr__`（或者一个 `__getattribute__`，对前面的情况），插入方法将会失效。一个插入的 `__setattr__` 还必须考虑到客户类中的一个 `__slots__`。

此外，这种方法解决了上一小节介绍的内置操作属性问题，因为在这些情况下 `__getattribute__` 并不运行。在我们的例子中，如果 `Person` 有一个 `__str__`，打印操作将运行它，但只是因为它真正出现在了那个类中。和前面一样，`__str__` 属性不会一般性地路由到插入的 `__getattribute__` 方法——打印将会绕过这个方法，并且直接调用类的 `__str__`。

尽管这可能比在包装对象内根本不支持运算符重载方法要好（除非重定义），但这种方法仍然没有拦截和验证 `__x__` 方法，这使得它们不可能成为 `Private` 的。尽管大多数运算符重载方法意味着是公有的，但有一些可能不是。

更糟糕的是，由于这个非包装器方法通过向装饰类添加一个 `__getattribute__` 来工作，它也会拦截类自身做出的属性访问，并像对来自类外部的访问一样地验证它们——这也意味着类的方法不能够使用 `Private` 名称！

实际上，像这样插入方法功能上等同于继承它们，并且意味着与我们在第29章最初的私有代码同样的限制。要知道一个属性访问是源自于类的内部还是外部，我们的方法需要在Python调用堆栈上检查frame对象。这可能最终产生一个解决方案（例如，使用检查堆栈的特性或描述符来替代私有属性），但是它可能会进一步减慢访问，并且其内部对我们来说过于复杂，无法在此介绍。

尽管有趣并且可能与一些其他的使用情况相关，但这种插入技术的方法并没有达到我们的目标。这里，我们不会进一步介绍这一选项的编码模式，因为我们将下一章中学习类扩展技术，与元类联合使用。正如我们将在那里看到的，元类并不严格需要以这种方式修改类，因为类装饰器往往充当同样的角色。

Python不是关于控制

既然我已经用如此大的篇幅添加了对Python代码的 `Private` 和 `Public` 属性声明，必须再次提醒你，像这样为类添加访问控制不完全是Python的特性。实际上，大多数Python程

程序员可能发现这一示例太大或者完全无关，除非充当装饰器的使用的一个展示。大多数大型的Python程序根本没有任何这样的控制而获得成功。如果你确实想要控制属性访问以杜绝编码错误，或者恰好近乎是专家级的C++或Java程序员，那么使用Python的运算符重载和内省工具，大多数事情是可以完成的。

示例：验证函数参数

作为装饰器工具的最后一个示例，本节开发了一个**函数装饰器**，它自动地测试传递给一个函数或方法的参数是否在有效的数值范围内。它设计用来在任何开发或产品阶段使用，并且它可以用作类似任务的一个模板（例如，参数类型测试，如果必须这么做的话）。由于本章的篇幅限制已经超出了，所以这个示例的代码主要靠自学，带有有限的说明。和往常一样，浏览代码来了解更多细节。

目标

在第27章面向对象教程中，我们编写了一个类，根据一个传入的百分比用来给表示人的对象涨工资：

```
class Person:
    ...
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

那里，我们注意到，如果想要编写健壮的代码，检查百分比以确保它不会太大或太小，这是一个好主意。我们可以在方法自身中使用if或assert语句来实现这样的检查，使用内嵌测试：

```
class Person:
    def giveRaise(self, percent):
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))

class Person:
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))
```

然而，这种方法使得带有内嵌测试的方法变得散乱，可能只有在开发阶段有用。对于更为复杂的情况，这可能很繁琐（假设试图内嵌代码来实现上一小节的装饰器所提供的属性私有）。可能更糟糕的是，如果需要修改验证逻辑，这里可能会有任意多个内嵌副本需要找到并更新。

一种更为有用和有趣的替代方法是，开发一个通用的工具来自动为我们执行范围测试，针对我们现在或将来要编写的任何函数或方法的参数。装饰器方法使得这明确而方便：

```
class Person:
    @rangetest(percent=(0.0, 1.0))           # Use decorator to validate
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

在装饰器中隔离验证逻辑，这简化了客户类和未来的维护。

注意，我们这里的目标和前面编写的属性验证不同。这里，我们想要验证传入的**函数参数**的值，而不是设置的**属性的值**。Python的装饰器和内省工具允许我们很容易地编写这一新的任务。

针对位置参数的一个基本范围测试装饰器

让我们从基本的范围测试实现开始。为了简化，我们将从编写一个只对位置参数有效的装饰器开始，并且假设它们在每次调用中总是出现在相同的位置。它们不能根据关键字名称传递，并且我们在调用中不支持额外的**args关键字，因为这可能会使装饰器中的位置声明无效。在名为devtools.py的文件中编写如下代码：

```
def rangetest(*argchecks):                    # Validate positional arg ranges
    def onDecorator(func):
        if not __debug__:                     # True if "python -O main.py args..."
            return func                       # No-op: call original directly
        else:                                 # Else wrapper while debugging
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = 'Argument %s not in %s..%s' % (ix, low, high)
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

还是老样子，这段代码主要是修改了我们前面介绍的编码模式：我们使用装饰器参数，嵌套作用域以进行状态保持，等等。

我们还使用了嵌套的def语句以确保这对简单函数和方法都有效，就像在前面所学习到的。当用于类方法的时候，onCall在*args中的第一项接受主体类的实例，并且将其传递给最初的方法函数中的self；在这个例子中，范围测试中的参数数目从1开始，而不是从0开始。

还要注意到，这段代码使用了__debug__内置变量——Python将其设置为True，除非它

将以-O优化命令行标志运行（例如，python -O main.py）。当__debug__为False的时候，装饰器返回未修改的最初函数，以避免额外调用及其相关的性能损失。

这个第一次迭代解决方案使用如下：

```
# File devtools_test.py

from devtools import rangetest
print(__debug__)                                # False if "python -O main.py"

@rangetest((1, 0, 120))                        # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                      # age must be in 0..120
    print('%s is %s years old' % (name, age))

@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2009])
def birthday(M, D, Y):
    print('birthday = {0}/{1}/{2}'.format(M, D, Y))

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest([1, 0.0, 1.0])                  # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):              # Arg 0 is the self instance here
        self.pay = int(self.pay * (1 + percent))

# Comment lines raise TypeError unless "python -O" used on shell command line

persinfo('Bob Smith', 45)                     # Really runs onCall(...) with state
#persinfo('Bob Smith', 200)                   # Or person if -O cmd line argument

birthday(5, 31, 1963)
#birthday(5, 32, 1963)

sue = Person('Sue Jones', 'dev', 100000)
sue.giveRaise(.10)                             # Really runs onCall(self, .10)
print(sue.pay)                                 # Or giveRaise(self, .10) if -O
#sue.giveRaise(1.10)
#print(sue.pay)
```

运行的时候，这段代码中的有效调用产生如下的输出（本节中的所有代码在Python 2.6和Python 3.0下同样地工作，因为两个版本都支持函数装饰器，我们没有使用属性委托，并且我们使用Python 3.0式的print调用和意外构建语法）：

```
C:\misc> C:\python30\python devtools_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
```

取消掉对任何无效调用的注释，将会由装饰器引发一个TypeError。下面是允许最后两行运行的时候的结果（和往常一样，我省略了一些出错消息文本以节省篇幅）：

```
C:\misc> C:\python30\python devtools_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
TypeError: Argument 1 not in 0.0..1.0
```

在系统命令行，使用`-O`标志来运行Python，将会关闭范围测试，但是也会避免包装层的性能负担——我们最终直接调用最初未装饰的函数。假设这只是一个调试工具，可以使用这个标志来优化程序以供产品阶段使用：

```
C:\misc> C:\python30\python -O devtools_test.py
False
Bob Smith is 45 years old
birthday = 5/31/1963
110000
231000
```

针对关键字和默认泛化

前面的版本说明了我们使用的基础知识，但是它相当有局限——它只支持按照位置传递的参数的验证，并且它没有验证关键字参数（实际上，它假设不会有那种使得参数位置数不正确的关键字传递）。此外，对于在一个给定调用中可能忽略的默认参数，它什么也没有做。如果所有的参数都按照位置传递并且不是默认的，这没有问题，但在一个通用工具中，这是极少的理想状态。Python支持要灵活的多的参数传递模式，而这些我们还没有解决。

对示例的如下修改做得更好。通过把包装函数的期待参数与调用时实际传入的参数匹配，它支持对按照位置或关键字名称传入的参数的验证，并且对于调用忽略的默认参数，它会跳过测试。简而言之，要验证的参数通过关键字参数指定到装饰器，装饰器随后遍历`*pargs` positionals tuple和`**kargs` keywords字典以进行验证。

```
"""
File devtools.py: function decorator that performs range-test
validation for passed arguments. Arguments are specified by
keyword to the decorator. In the actual call, arguments may
be passed by position or keyword, and defaults may be omitted.
See devtools_test.py for example use cases.
"""

trace = True

def rangetest(**argchecks):
    def onDecorator(func):
        if not __debug__:
            return func
        else:
            import sys
            # Validate ranges for both+defaults
            # onCall remembers func and argchecks
            # True if "python -O main.py args..."
            # Wrap if debugging; else use original
```

```

code    = func.__code__
allargs = code.co_varnames[:code.co_argcount]
funcname = func.__name__

def onCall(*pargs, **kargs):
    # All pargs match first N expected args by position
    # The rest must be in kargs or be omitted defaults
    positionals = list(allargs)
    positionals = positionals[:len(pargs)]

    for (argname, (low, high)) in argchecks.items():
        # For all args to be checked
        if argname in kargs:
            # Was passed by name
            if kargs[argname] < low or kargs[argname] > high:
                errmsg = '{0} argument "{1}" not in {2}..{3}'
                errmsg = errmsg.format(funcname, argname, low, high)
                raise TypeError(errmsg)

            elif argname in positionals:
                # Was passed by position
                position = positionals.index(argname)
                if pargs[position] < low or pargs[position] > high:
                    errmsg = '{0} argument "{1}" not in {2}..{3}'
                    errmsg = errmsg.format(funcname, argname, low, high)
                    raise TypeError(errmsg)

            else:
                # Assume not passed: default
                if trace:
                    print('Argument "{0}" defaulted'.format(argname))
                return func(*pargs, **kargs)          # OK: run original call
    return onCall
return onDecorator

```

如下的测试脚本展示了如何使用装饰器——要验证的参数由关键字装饰器参数给定，并且在实际的调用中，我们可以按照名称或位置传递，或者如果它们有效的话，可以用默认方式来忽略验证：

```

# File devtools_test.py
# Comment lines raise TypeError unless "python -O" used on shell command line
from devtools import rangetest

# Test functions, positional and keyword

@rangetest(age=(0, 120))          # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print('%s is %s years old' % (name, age))

@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2009))
def birthday(M, D, Y):
    print('birthday = {0}/{1}/{2}'.format(M, D, Y))

persinfo('Bob', 40)
persinfo(age=40, name='Bob')
birthday(5, D=1, Y=1963)

```

```

#persinfo('Bob', 150)
#persinfo(age=150, name='Bob')
#birthday(5, D=40, Y=1963)

# Test methods, positional and keyword

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest(percent=(0.0, 1.0))    # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):    # percent passed by name or position
        self.pay = int(self.pay * (1 + percent))

bob = Person('Bob Smith', 'dev', 100000)
sue = Person('Sue Jones', 'dev', 100000)
bob.giveRaise(.10)
sue.giveRaise(percent=.20)
print(bob.pay, sue.pay)
#bob.giveRaise(1.10)
#bob.giveRaise(percent=1.20)

# Test omitted defaults: skipped

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):

    print(a, b, c, d)

omitargs(1, 2, 3, 4)
omitargs(1, 2, 3)
omitargs(1, 2, 3, d=4)
omitargs(1, d=4)
omitargs(d=4, a=1)
omitargs(1, b=2, d=4)
omitargs(d=8, c=7, a=1)

#omitargs(1, 2, 3, 11)           # Bad d
#omitargs(1, 2, 11)             # Bad c
#omitargs(1, 2, 3, d=11)        # Bad d
#omitargs(11, d=4)              # Bad a
#omitargs(d=4, a=11)            # Bad a
#omitargs(1, b=11, d=4)         # Bad b
#omitargs(d=8, c=7, a=11)       # Bad a

```

这段脚本运行的时候，超出范围的参数会像前面一样引发异常，但参数可以按照名称或位置传递，并且忽略的默认参数不会验证。这段代码在Python 2.6和Python 3.0下都可以运行，但额外的元组圆括号在Python 2.6中会打印出来。跟踪输出并进一步测试它以自行体验；它像前面一样工作，但是，它的范围已经拓展了很多：

```

C:\misc> C:\python30\python devtools_test.py
Bob is 40 years old
Bob is 40 years old
birthday = 5/1/1963

```

```

110000 120000
1 2 3 4
Argument "d" defaulted
1 2 3 9
1 2 3 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
1 2 8 4
Argument "b" defaulted
1 7 7 8

```

对于验证错误，当方法测试行之一注释掉的时候，我们像前面一样得到一个异常（除非 -0 命令行参数传递给 Python）：

```

TypeError: giveRaise argument "percent" not in 0.0..1.0

```

实现细节

装饰器的代码依赖于内省API和对参数传递的细微限制。为了完整地泛化，以便我们可以从原则上整体模拟Python的参数匹配逻辑，来看到哪个名称以何种模式传入，但是，这对于我们的工具来说太复杂了。如果我们能够根据所有期待的参数的名称集合来按照名称匹配传入的参数，从而判断哪个位置参数真正地出现在给定的调用中，那将会更好。

函数内省

已经证实了内省API可以在函数对象以及其拥有我们所需的工具的相关代码对象上实现。第19章简单介绍过这个API，我们在这里实际地使用它。期待的参数名集合只是附加给一个函数的代码对象的前N个变量名：

```

# In Python 3.0 (and 2.6 for compatibility):
>>> def func(a, b, c, d):
...     x = 1
...     y = 2
...
>>> code = func.__code__                # Code object of function object
>>> code.co_nlocals
6
>>> code.co_varnames                    # All local var names
('a', 'b', 'c', 'd', 'x', 'y')
>>> code.co_varnames[:code.co_argcount] # First N locals are expected args
('a', 'b', 'c', 'd')

>>> import sys                          # For backward compatibility

```

```
>>> sys.version_info                                     # [0] is major release number
(3, 0, 0, 'final', 0)
>>> code = func.__code__ if sys.version_info[0] == 3 else func.func_code
```

同样的API在较早的Python中也可以使用，但是，在Python 2.5及更早的版本中，`func.__code__` attribute拼写为`func.func_code` in 2.5（为了可移植性，新的`__code__`属性在Python 2.6中也冗余地可用）。在函数上和代码对象上运行一个`dir`调用，以了解更多细节。

参数假设

给定期待的参数名的这个集合，该解决方案依赖于Python对于参数传递顺序所施加的两条限制（在Python 2.6和Python 3.0中都仍然成立）：

- 在调用时，所有的位置参数出现在所有关键字参数之前。
- 在`def`中，所有的非默认参数出现在所有的默认参数之前。

也就是说，在一个调用中，一个非关键字参数通常不会跟在一个关键字参数后面，并且在定义中，一个非默认参数不会跟在一个默认参数后面。在两种位置中，所有的“`name=value`”语法必须出现在任何简单的“`name`”之后。

为了简化，我们也可以假设一个调用一般是有效的——例如，所有的参数要么接收值（按照名称或位置），要么将有意忽略而选取默认值。不一定要进行这种假设，因为当包装逻辑测试有效性的时候，函数还没有真正调用——随后包装逻辑调用的时候，调用仍然可能失效，由于不正确的参数传递。只要这不会引发包装器在糟糕地失效，我们可以改进调用的验证。这是有帮助的，因为在调用发生之前验证它，将会要求我们完全模仿Python的参数匹配算法——再一次说明，这对我们的工具来说是一个过于复杂的过程。

匹配算法

现在，给定了这些限制和假设，我们可以用这一算法来考虑调用中的关键字以及忽略的默认参数。当拦截了一个调用，我们可以作如下假设：

- `*pargs`中的所有 N 个传递的位置参数，必须与从函数的代码对象获取的前 N 个期待的参数匹配。对于前面列出的每个Python的调用顺序规则都是如此，因为所有的位置参数在所有关键字参数之前。
- 要获取按照位置实际传递的参数的名称，我们可以把所有其他参数的列表分片为长度为 N 的`*pargs`位置参数元组。
- 前 N 个期待的参数之后的任何参数，要么是按照关键字传递，要么是调用时候忽略的默认参数。

- 对于要验证的每个参数名，如果它在`**kwargs`中，它是按照名称传递的；如果它在前 N 个期待的参数中，它是按照位置传递的（在这种情况下，它在期待的列表中的相对位置给出了它在`*args`中的相对位置）；否则，我们可以假设它是在调用时候忽略的并且默认的参数，不需要检查。

换句话说，对于假设`*args`中前 N 个实际传递的位置参数，必须与期待的参数列表中的前 N 个参数匹配，并且任何其他参数要么必须是按照关键字传递并位于`**kwargs`中，要么必须是默认的参数，我们就可以略过对调用时忽略的参数的测试。在这种方法下，对于最右边的位置参数和最左边的关键字参数之间的、或者在关键字参数之间的、或者在所有的最右边的位置之后的那些忽略掉的参数，装饰器将省略对其检查。可以跟踪装饰器及其测试脚本，看看这是如何在代码中实现的。

开放问题

尽管我们的范围测试工具按照计划工作，但还是有两个缺陷。首先，正如前面提到的，对最初函数的无效调用在最终的装饰器中仍然会失效。例如，如下的两个调用都会触发异常：

```
omitargs()  
omitargs(d=8, c=7, b=6)
```

然而，这只是失效，而我们想要调用最初的函数，在包装器的末尾。尽管我们可以尝试模仿Python的参数匹配来避免如此，但没有太多的理由来这么做——因为无论如何调用将会在此处失效，所以我们可能也想让Python自己的参数匹配逻辑来为我们检测问题。

最后，尽管最终的版本处理位置参数、关键字参数和省略的参数，但它仍然不会对将要在接受任意多个参数的装饰器函数中使用的`*args`和`**kwargs`显式地做任何事情。然而，我们可能不需要关心自己的目标：

- 如果传递了一个额外的**关键字**参数，其名称将会出现在`**kwargs`中，并且如果提交给装饰器，可以对其进行常规的测试。
- 如果**没有**传递一个额外的关键字，其名称不会出现在`**kwargs`或者分片的期待位置列表中，由此，不会检查它——会当做默认参数来对待它，即便它实际上是一个可选的额外参数。
- 如果传递了一个额外的**位置**参数，没有办法在装饰器中引用它——其名称不会出现在`**kwargs`或者分片的期待位置列表中，因此会直接忽略它。由于这样的参数没有在函数的定义中列出，所以没有办法把一个给装饰器的名称映射回到一个期待的相对位置。

换句话说，由于代码支持按照名称测试任意的关键字参数，但是不支持那些未命名的并且在函数的参数签名中没有预定位置的任意位置参数。

原则上，我们可以扩展装饰器的接口，以便在装饰的函数中支持*args，但这么做在极少情况下会有用（例如，一个特定参数名称带有一个测试，该测试应用于包装器的*pargs中超出期待参数列表的长度之外的所有参数）。既然我们已经在这个示例上耗费了很大的篇幅，所以如果你对这样的改进感兴趣，请在建议的练习中进一步研究这一主题。

装饰器参数 VS 函数注解

有趣的是，Python 3.0中提供的函数注解功能，可能为我们在指定范围测试的示例中所使用的装饰器参数给出了一种替代方法。正如我们在第19章中学到的，注解允许我们把表达式和参数及返回值关联起来，通过在自己的def头部行中编写它们。Python把注解收集到字典中并且将其附加给注解的函数。

我们可以在示例中的标题行编写范围限制，而不是在装饰器参数中编写。我们将仍然需要一个函数装饰器来包装函数以拦截随后的调用，但我们基本上换掉了装饰器参数语法：

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)                                # func = rangetest(...)(func)
```

注解语法如下：

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

现在，范围限制移到了函数自身之中，而不是在外部编写。如下的脚本说明了两种方案下的最终装饰器的结构，以不完整的框架代码给出。装饰器参数编码模式就是我们前面给出的完整解决方案，注解替代方案需要一个较少层级的嵌套，因为它不需要保持装饰器参数：

```
# Using decorator arguments

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks: pass          # Add validation code here
            return func(*pargs, **kargs)
        return onCall
    return onDecorator
```

```

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):
    print(a + b + c)

func(1, 2, c=3)

# Using function annotations

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks: pass
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)

func(1, 2, c=3)

```

运行的时候，两种方案都会访问同样的验证测试信息，但是，以不同的形式——装饰器参数版本的信息保持在封闭作用域的一个参数中；注解版本的信息保持在函数自身的一个属性中：

```

{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6

```

我将充实基于注解的版本留作一个建议的练习，其编码和我们前面给出的完整解决方案是相同的，因为范围测试信息直接在函数上而不是在一个封闭作用域中。实际上，所有这些给我们带来的是工具的一个不同的用户接口——仍然需要像前面一样，根据期待的参数名称来匹配参数名称，以获取相对位置。

实际上，在这个例子中，使用注解而不是装饰器参数确实限制了其用途。首先，注解只在Python 3.0下有效，因此，Python 2.6不再得到支持；另一方面，带有参数的函数装饰器，在两个版本下都有效。

更重要的是，通过把验证规范移动到def标题中，我们基本上给函数指定了一个单独的角色——因为注解允许我们为每个参数只编写一个表达式，它可以只有一个用途。例如，我们不能为其他用途而使用范围测试注解。

相反，由于装饰器参数在函数自身之外编写，所以它们更容易删除并且更通用——函数自身的代码并没有暗含任何单一的装饰目的。实际上，通过带有参数的嵌套装饰器，我

们可以对同一个函数应用多个扩展步骤；注解直接只支持一个步骤。使用装饰器参数，函数自身也保留一个简单的、常规的外观。

然而，如果有单一的目的，并且只使用Python 3.X，那么在注解和装饰器参数之间的选择很大程度上只是风格和个人主观爱好了。就像生活中常见的现象，一个人的注解恰好是另一个人的语法垃圾……

其他应用程序：类型测试

在处理装饰器参数时，我们所使用的编码模式可以应用于其他环境。例如，在开发时检查参数数据类型，这是一种直接的扩展：

```
def typetest(**argchecks):
    def onDecorator(func):
        ....
        def onCall(*pargs, **kargs):
            positionals = list(allargs)[:len(pargs)]
            for (argname, type) in argchecks.items():
                if argname in kargs:
                    if not isinstance(kargs[argname], type):
                        ...
                        raise TypeError(errmsg)
                elif argname in positionals:
                    position = positionals.index(argname)
                    if not isinstance(pargs[position], type):
                        ...
                        raise TypeError(errmsg)
                else:
                    # Assume not passed: default
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@typetest(a=int, c=float)
def func(a, b, c, d):
    ...

func(1, 2, 3.0, 4)           # Okay
func('spam', 2, 99, 4)       # Triggers exception correctly
```

实际上，我们甚至可以通过传入一个测试函数来进一步泛化，就像我们在前面添加Public装饰时所做的那样；这种代码的单个副本足够用于范围和类型测试。正如前面的小节所述，对于这样的装饰器使用函数注解而不是装饰器参数，将会使其看起来更像是其他语言中的类型声明：

```
@typetest
def func(a: int, b, c: float, d):
    ...
# func = typetest(func)
# Gasp!...
```

正如我们已经在本书中学习到的，这一特定角色通常是工作代码中的一个糟糕思路，并且根本不是Python化的（实际上，它往往是有经验的C++程序员初次尝试使用Python的一种现象）。

类型测试限制了我们的函数只能在特定类型上工作，而不是允许它在任何具有可兼容性接口的类型上操作。实际上，它限制了代码并且破坏了其灵活性。另一方面，每个规则都有例外；类型检查可能在一种孤立的情况下很方便好用，即调试时以及用更为限制性的语言（如C++等）编写的代码接口的时候。参数处理的这一通用模式，可能也适用于各种争议性较少的角色。

本章小结

在本章中，我们探讨了装饰器——适用于函数和类的各种情况。正如我们所学习的，装饰器是当一个函数或类定义的时候插入自动运行的代码的一种方式。当使用一个装饰器的时候，Python把函数或类名重新绑定到它返回的可调用对象。本书允许我们为函数调用和类实例创建调用添加一层包装器逻辑，以便管理函数和实例。正如我们已经看到的，管理器函数和手动名称重新绑定都可以实现同样的效果，但装饰器提供了一种更加明确和统一的解决方案。

我们将在下一章看到，类装饰器也可以用来管理类本身，而不只是管理它们的实例。由于这一功能与下一章的主题元类有重合，所以需要阅读下一章。首先，做如下的练习题。由于本章主要关注其较大的示例，因此练习题将会要求你修改一些代码以便复习本章知识。

本章习题

1. 正如本章的提示中提到的，我们在本章“添加装饰器参数”小节所编写的带有装饰器参数的计时器函数装饰器，只能应用于简单函数，因为它使用了一个嵌套的类，该类带有一个`__call__`运算符重载方法来捕获调用。这种结构对于类方法无效，因为装饰器实例传递给`self`，而不是主体类实例。重新编写这个装饰器，以便它可以应用于简单函数和类方法，并且在函数和方法上都测试它（提示，参见本章“类错误之一：装饰类方法”小节）。注意，我们可以使用赋值函数对象属性来记录总的时间，因为你没有一个嵌套的类用于状态保持，并且不能从装饰器代码的外部访问非局部变量。
2. 我们在本章中编写的`Public/Private`类装饰器将会为一个装饰的类中的每次属性获取增加负担。尽管我们可以直接删除`@装饰行`以获取速度，但我们也可以扩展装饰

器自身类检查__debug__开关，并且在命令行传递-O Python标志的时候根本不执行包安装（正如我们对于参数范围测试装饰器所做的那样）。通过这种方法，我们可以加速程序而不用修改源代码，即通过命令行参数（python -O main.py...）。编写代码并测试这一扩展。

习题解答

1. 这里有一种方法来编写第一个问题的解决方案及其输出（虽然对类方法来说运行得太快而无法计时）。技巧在于用嵌套的函数替代嵌套的类，以便self参数不再是装饰器的实例，并且把整个事件赋值给装饰器函数自身，以便随后可以通过最初重绑定的名称来获取它（参见本章“状态信息保持选项”小节——函数支持任意的属性附件，并且函数名在这种环境中是一个封装的作用域引用）。

```
import time

def timer(label='', trace=True):
    def onDecorator(func):
        def onCall(*args, **kwargs):
            start = time.clock()
            result = func(*args, **kwargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator

# Test on functions

@timer(trace=True, label='[CCC]==>')
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

for func in (listcomp, mapcall):
    result = func(5)
    func(5000000)
    print(result)
    print('allTime = %s\n' % func.alltime)

# Test on methods

class Person:
    def __init__(self, name, pay):
```

```

        self.name = name
        self.pay = pay

    @timer()
    def giveRaise(self, percent):          # giveRaise = timer()(giveRaise)
        self.pay *= (1.0 + percent)      # tracer remembers giveRaise

    @timer(label='**')
    def lastName(self):                   # lastName = timer(...)(lastName)
        return self.name.split()[-1]     # alltime per class, not instance

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
bob.giveRaise(.10)
sue.giveRaise(.20)                      # runs onCall(sue, .10)
print(bob.pay, sue.pay)
print(bob.lastName(), sue.lastName())   # runs onCall(bob), remembers lastName
print('%.5f %.5f' % (Person.giveRaise.alltime, Person.lastName.alltime))

# Expected output

[CCC]==>listcomp: 0.00002, 0.00002
[CCC]==>listcomp: 1.19636, 1.19638
[0, 2, 4, 6, 8]
allTime = 1.19637775192
[MMM]==>mapcall: 0.00002, 0.00002
[MMM]==>mapcall: 2.29260, 2.29262
[0, 2, 4, 6, 8]
allTime = 2.2926232943

giveRaise: 0.00001, 0.00001
giveRaise: 0.00001, 0.00002
55000.0 120000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Smith Jones
0.00002 0.00002

```

2. 下面的代码解决了第二个问题——它已经扩展来在优化的模式中返回最初的类(-o)，因此，属性访问不会引发速度问题。实际上，我所做的是添加调试模式的测试语句，并且进一步向右缩进类。如果想要在Python 3.0下也支持把这些委托给主体类，那么向包装类添加运算符重载方法重定义（Python 2.6通过__getattr__路由这些，但Python 3.0和Python 2.6中的新式类不这么做）。

```

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        if not __debug__:
            return aClass
        else:
            class onInstance:

```

```

def __init__(self, *args, **kwargs):
    self.__wrapped__ = aClass(*args, **kwargs)
def __getattr__(self, attr):
    trace('get:', attr)
    if failIf(attr):
        raise TypeError('private attribute fetch: ' + attr)
    else:
        return getattr(self.__wrapped__, attr)
def __setattr__(self, attr, value):
    trace('set:', attr, value)
    if attr == '__onInstance__wrapped__':
        self.__dict__[attr] = value
    elif failIf(attr):
        raise TypeError('private attribute change: ' + attr)
    else:
        setattr(self.__wrapped__, attr, value)
    return onInstance
return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

# Test code: split me off to another file to reuse decorator

@Private('age')
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Person = Private('age')(Person)
# Person = onInstance with state
# Inside accesses run normally

X = Person('Bob', 40)
print(X.name)
X.name = 'Sue'
print(X.name)
#print(X.age)
#X.age = 999
#print(X.age)
# Outside accesses validated
# FAILS unless "python -O"
# ditto
# ditto

@Public('name')
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

X = Person('bob', 40)
print(X.name)
X.name = 'Sue'
print(X.name)
#print(X.age)
#X.age = 999
#print(X.age)
# X is an onInstance
# onInstance embeds Person
# FAILS unless "python -O main.py"
# ditto
# ditto

```


元类

在上一章中，我们介绍了装饰器并研究了其应用的各种示例。在本书的最后一章中，我们将继续关注工具构建器，并讨论另一个高级话题：**元类**。

从某种意义上讲，元类只是扩展了装饰器的代码插入模式。正如我们在上一章所学到的，函数和类装饰器允许我们拦截并扩展函数调用以及类实例创建调用。以类似的思路，元类允许我们拦截并扩展**类创建**——它们提供了一个API以插入在一条`class`语句结束时运行的额外逻辑，尽管是以与装饰器不同的方式。同样，它们提供了一种通用的协议来管理程序中的类对象。

就像本书的这一部分其他各章所讨论的主题一样，这是一个**高级话题**，需要有一定的基础。实际上，元类允许我们获得更高层级的控制，来控制一组类如何工作。这是一个功能强大的概念，并且元类并不是打算供大多数应用程序员使用的（或者，坦白地说，不适合懦弱之人）。

另一方面，元类为各种没有它而难以实现或不可能实现的编码模式打开了大门，并且对于那些追求编写灵活的API或编程工具供其他人使用的程序员来说，它特别有用。即便你不属于此类程序员，元类也可以教你很多有关Python的类模型的一般性知识。

和上一章一样，我们这里的部分目标只是展示比本书前面的示例更为实际的代码实例。尽管元类是一个核心主题而且并非自成一个应用领域，本章的部分目标是激发你在阅读完本书后继续研究较大的应用程序编程示例的兴趣。

要么是元类，要么不是元类

元类可能是本书中最高级的主题，如果不把Python语言算作整体的话。借用经验丰富

的Python开发者Tim Peters在*comp.lang.python*新闻组中的话来说（Tim Peters是著名的Python座右铭“import this”的作者）：

[元类]比99%的用户所担心的魔力要更深。如果你犹豫是否需要它们，那你不需要它们（真正需要元类的人，能够确定地知道需要它们，并且不需要说明为什么需要）。

换句话说，元类主要是针对那些构建API和工具供他人使用的程序员。在很多情况下（如果不是大多数的话），它们可能不是应用程序工作的最佳选择。在开发其他人将来会用的代码的时候，尤其如此。“因为某物很酷”而编写它，似乎不是一种合理的判断，除非你在做实验或者学习。

然而，元类有着各种各样广泛的潜在角色，并且知道它们何时有用是很重要的。例如，它们可能用来扩展具有跟踪、对象持久、异常日志等功能的类。它们也可以用来在运行时根据配置文件来构建类的一部分，对一个类的每个方法广泛地应用函数装饰器，验证其他的接口的一致性，等等。

在它们更宏观的化身中，元类甚至可以用来实现替代的编程模式，例如面向方面编程、数据库的对象/关系映射（ORM），等等。尽管常常有实现这些结果的替代方法（正如我们看到的，类装饰器和元类的角色常常有重合），元类提供了一种正式模型，可以裁减以完成那些任务。我们没有足够的篇幅在本章中介绍所有这些第一手的应用程序，但是，在此学完了基础知识后，你应该自行在Web上搜索以找到其他的用例。

在本书中学习元类的可能原因是，这个主题能够帮助更广泛地说明Python的类机制。尽管你可能在自己的工作中编写或重用它们，也可能不会这么做，大概理解元类也可以在很大程度上更深入地理解Python。

提高魔力层次

本书的大多数部分关注直接的应用程序编码技术，因为大多数程序员都花费时间来编写模块、函数和类来实现现实的目标。他们也使用类和创建实例，并且可能甚至做一些运算符重载，但是，他们可能不会太深入地了解类实际是如何工作的细节。

然而，在本书中我们已经看到了各种工具，它们允许我们以广泛的方式控制Python的行为，并且它们常常与Python的内部与工具构建有更多的关系，而与应用程序编程领域相关甚少：

内省属性

像`__class__`和`__dict__`这样的特殊属性允许我们查看Python对象的内部实现方面，以便更广泛地处理它们，列出对象的所有属性、显示一个类名，等等。

运算符重载方法

像`__str__`和`__add__`这样特殊命名的方法，在类中编写来拦截并提供应用于类实例的内置操作的行为，例如，打印、表达式运算符等等。它们自动运行作为对内置操作的响应，并且允许类符合期望的接口。

属性拦截方法

一类特殊的运算符重载方法提供了一种方法在实例上广泛地拦截属性访问：`__getattr__`、`__setattr__`和`__getattribute__`允许包装的类插入自动运行的代码，这些代码可以验证属性请求并且将它们委托给嵌入的对象。它们允许一个对象的任意数目的属性——要么是选取的属性，要么是所有的属性——在访问的时候计算。

类特性

内置函数`property`允许我们把代码和特殊的类属性关联起来，当获取、赋值或删除该属性的时候就自动运行代码。尽管不像前面一段所介绍的工具那样通用，特性考虑到了访问特定属性时候的自动代码调用。

类属性描述符

其实，特性只是定义根据访问自动运行函数的属性描述符的一种简洁方式。描述符允许我们在单独的类中编写`__get__`、`__set__`和`__delete__`处理程序方法，当分配给该类的一个实例的属性被访问的时候自动运行它们。它们提供了一种通用的方式，来插入当访问一个特定的属性时自动运行的代码，并且在一个属性的常规查找之后触发它们。

函数和类装饰器

正如我们在第38章看到的，装饰器的特殊的`@`可调用语法，允许我们添加当调用一个函数或创建一个类实例的时候自动运行的逻辑。这个包装器逻辑可以跟踪或计时调用，验证参数，管理类的所有实例，用诸如属性获取验证的额外行为来扩展实例，等等。装饰器语法插入名称重新绑定逻辑，在函数或类定义语句的末尾自动运行该逻辑——装饰的函数和类名重新绑定到拦截了随后调用的可调用对象。

正如本章的介绍中所提到的，元类是这些技术的延续——它们允许我们在一条`class`语句的末尾，插入当创建一个类对象的时候自动运行的逻辑。这个逻辑不会把类名重新绑定到一个装饰器可调用对象，而是把类自身的创建指向特定的逻辑。

换句话说，元类最终只是定义自动运行代码的另外一种方式。通过元类以及前面列出的其他工具，Python为我们提供了在各种环境中插入逻辑的方法——在运算符计算时、属性访问时、函数调用时、类实例创建时，现在是在类对象创建时。

和类装饰器不同，它通常是添加**实例**创建时运行的逻辑，元类在**类**创建时运行。同样的，它们都是通常用来管理或扩展类的钩子，而不是管理其实例。

例如，元类可以用来自动为类的所有方法添加装饰，把所有使用的类注册到一个API，自动为类添加用户接口逻辑，在文本文件中从简单声明来创建或扩展类，等等。由于我们可以控制如何创建类（并且通过它们的实例获取的行为），它们的实用性潜在地很广泛。

正如我们已经看到的，这些高级Python工具中的很多都有交叉的角色。例如，属性往往可以用特性、描述符或属性拦截方法来管理。正如我们在本章中见到的，类装饰器和元类往往可以交换使用。尽管类装饰器常常用来管理实例，它们也可以用来管理类；类似的，尽管元类设计用来扩展类构建，它们也常常插入代码来管理实例。尽管选择使用哪种技术有时候纯粹是主观的事情，但知道替代方案可以帮助我们为给定的任务挑选正确的工具。

“辅助”函数的缺点

也像上一章介绍的装饰器一样，元类常常是可选的，从理论的角度来看是这样。我们通常可以通过**管理器函数**（有时候叫做“辅助”函数）传递类对象来实现同样的效果，这和我们通过管理器代码传递函数和实例来实现装饰器的目的很相似。然而，就像装饰器一样，元类：

- 提供一种更为正式和明确的结构。
- 有助于确保应用程序员不会忘记根据一个API需求来扩展他们的类。
- 通过把类定制逻辑工厂化到一个单独的位置（元类）中，避免代码冗余及其相关的维护成本。

为了便于说明，假设我们想要在一组类中自动插入一个方法。当然，如果在编写类的时候知道主体方法，我们可以用简单的**继承**来做到这点。在那种情况下，我们可以直接在超类中编写该方法，并且让所有涉及的类继承它：

```
class Extras:
    def extra(self, args):          # Normal inheritance: too static
    ...

class Client1(Extras): ...         # Clients inherit extra methods
class Client2(Extras): ...
class Client3(Extras): ...

X = Client1()                      # Make an instance
X.extra()                          # Run the extra methods
```

然而，有时候，在编写类的时候不可能预计这样的扩展。考虑扩展类以响应在运行时用户界面中所做出的一个选择，或者在配置文件中指定类型的情况。尽管我们可以在我们想象的集合中编写每个类以**手动**检查这些，但有太多的问题要问客户类（这里的需求是抽象的，是需要填充的东西）：

```
def extra(self, arg): ...

class Client1: ...                # Client augments: too distributed
if required():
    Client1.extra = extra

class Client2: ...
if required():
    Client2.extra = extra

class Client3: ...
if required():
    Client3.extra = extra

X = Client1()
X.extra()
```

我们可以像这样在`class`语句之后把方法添加到类，因为类方法只不过是和一个类相关的函数，并且拥有第一个参数来接收`self`实例。尽管这有效，但它把所有的扩展负担放到了客户类上（并且假设它们能记住做这些）。

从维护的角度，把选择逻辑隔离到一个单独的地方，这可能会更好。我们可以通过把类指向一个**管理器函数**，从而把一些这些额外工作的一部分**封装**起来——这样的**一个**管理器函数将根据需求扩展类，并且处理运行时测试和配置的所有工作：

```
def extra(self, arg): ...

def extras(Class):                # Manager function: too manual
    if required():
        Class.extra = extra

class Client1: ...
extras(Client1)

class Client2: ...
extras(Client2)

class Client3: ...
extras(Client3)

X = Client1()
X.extra()
```

这段代码通过紧随类创建之后一个管理器函数来运行类。尽管像这样的**一个**管理器函数在这里可以实现我们的目标，但它们仍然给类的编写者增加了相当重的负担，所以编写

者必须理解需求并且将它们附加到代码中。如果有一种简单的方式在主体类中增强这种扩展，那将会更好，这样，编写者就不需要处理并且不会忘记使用扩展。换句话说，我们宁愿能够在一条class语句的末尾插入一些自动运行的代码，从而扩展该类。

这正是元类所做的事情——通过声明一个元类，我们告诉Python把类对象的创建路由到我们所提供的另一个类：

```
def extra(self, arg): ...

class Extras(type):
    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...    # Metaclass declaration only
class Client2(metaclass=Extras): ...    # Client class is instance of meta
class Client3(metaclass=Extras): ...

X = Client1()                          # X is instance of Client1
X.extra()
```

由于创建新的类的时候，Python在class语句的末尾自动调用元类，因此它可以根据需要扩展、注册或管理类。此外，客户类唯一的需求是，它们声明元类。每个这么做的类都将自动获取元类所提供的扩展，现在可以，如果将来元类修改了也可以。然而在一个小的示例中看到这点可能有些困难，元类通常比其他方法能够更好地处理这样的任务。

元类与类装饰器的关系：第一回合

我们已经讲过，前面一章所介绍的类装饰器有时候在功能上与元类有重合，注意到这点也很有趣。尽管类装饰器通常用来管理或扩展实例，但它们也可以扩展类，而独立于任何创建的实例。

例如，假设我们编写自己的管理器函数以返回扩展的类，而不是直接在原处修改它。这就允许更大程度的灵活性，因为管理器将会自由地返回实现了类期待接口的任何类型的对象：

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

class Client1: ...
Client1 = extras(Client1)

class Client2: ...
Client2 = extras(Client2)
```

```

class Client3: ...
Client3 = extras(Client3)

X = Client1()
X.extra()

```

如果你认为这只是回顾类装饰器的开始，那么你是对的。在前一章中，我们介绍了类装饰器作为扩展**实例**创建调用的工具。由于它们通过自动把一个类名绑定到一个函数的结果而工作，那么，没有理由在任何实例创建之前不能用它来扩展类。也就是说，在创建的时候，类装饰器可以对**类**应用额外的逻辑，而不只是对**实例**应用：

```

def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: ...           # Client1 = extras(Client1)

@extras
class Client2: ...           # Rebinds class independent of instances

@extras
class Client3: ...

X = Client1()                 # Makes instance of augmented class
X.extra()                     # X is instance of original Client1

```

这里，装饰器基本上会把前面示例的手动名称重新绑定自动化。就像是使用元类，由于装饰器返回最初的类，实例由此创建，而不是由包装器对象创建。实际上，根本没有拦截实例创建。

在这个特定的例子中（在类创建的时候给类添加方法），元类和装饰器之间的选择有些随意。装饰器可以用来管理实例和类，并且它们与元类在第二种角色中交叉了。

然而，这真的只是说明了元类的一种操作模式。正如我们将看到的，在这种角色中，装饰器对应到元类的 `__init__` 方法，但是，元类还有其他的定制钩子。正如我们还将看到的，除了类初始化，元类可以执行任意的构建任务，而这些可能对装饰器来说更难。

此外，尽管装饰器可以管理实例和类，反之却不然——元类设计来管理类，并且用它们来管理实例却不是很容易。我们将在本章稍后的代码中介绍这一区别。

本节的大多数代码都已经简化过，但是，我们将在本章后面将其补充为真实有用的示例。要完全理解元类是如何工作的，首先需要对其底层模型有一个清晰的印象。

元类模型

要真正理解元类是如何工作的，需要更多地理解Python的类型模型以及在class语句的末尾发生了什么。

类是类型的实例

到目前为止，在本书中，我们已经通过创建内置类型（如列表和字符串）的实例，以及我们自己编写的类的实例，完成了大多数工作。正如我们已经看到的，类的实例有一些自己的状态信息属性，但它们也从所创建自的类继承了行为属性。对于内置类型也是如此，例如，列表实例拥有自己的值，但是它们从列表类型继承方法。

尽管我们可以用这样的实例对象做很多事情，但Python的类型模型实际上比我前面所介绍的要丰富一些。实际上，该模型中有一个我们目前为止可以看到的漏洞：如果实例自类创建，那么，是什么创建了类？这证明了类也是某物的实例：

- 在Python 3.0中，用户定义的类对象是名为type的对象的实例，type本身是一个类。
- 在Python 2.6中，新式类继承自object，它是type的一个子类；传统类是type的一个实例，并且并不创建自一个类。

我们在本书第9章中介绍了类型的概念，在第31章介绍了类和类型的关系，但是，让我们在这里回顾一下基础知识，看看它们是如何应用于元类的。

还记得吧，type内置函数返回任何对象的类型（它本身是一个对象）。对于列表这样的内置类型，实例的类型是一个内置的列表类型，但是，列表类型的类型是类型type自身——顶层的type对象创建了具体的类型，具体的类型创建了实例。你将会在交互提示模式中亲自看到这点。例如，在Python 3.0中：

```
C:\misc> c:\python30\python
>>> type([])                # In 3.0 list is instance of list type
<class 'list'>
>>> type(type([]))          # Type of list is type class
<class 'type'>

>>> type(list)               # Same, but with type names
<class 'type'>
>>> type(type)               # Type of type is type: top of hierarchy
<class 'type'>
```

正如我们在第31章中学习新式类的时候所看到的，在Python 2.6中（及更早的版本中），通常也是一样的，但是，类型并不完全和类相同——type是一种独特的内置对象，它位于类型层级的顶层，并且用来构建类型：


```

C:\misc> c:\python26\python
>>> type([])                # In 2.6, type is a bit different
<type 'list'>
>>> type(type([]))
<type 'type'>

>>> type(list)
<type 'type'>
>>> type(type)
<type 'type'>

```

这说明了类型/实例关系对于类来说也是成立的：实例创建自类，而类创建自`type`。在Python 3.0中，“类型”的概念与“类”的概念合并了。实际上，这两者基本上是同义词——类是类型，类型也是类。即：

- 类型由派生自`type`的类定义。
- 用户定义的类是类型类的实例。
- 用户定义的类是产生它们自己的实例的类型。

正如我们在前面看到的，这一对等效果的代码测试实例的类型：一个实例的类型是产生它的类。这也暗示着，创建类的方式证明是本章主题的关键所在。由于类通常默认地创建自一个根类型类，因此大多数程序员不需要考虑这种类型/类对等性。然而，它开创了定制类及其实例的新的可能性。

例如，Python 3.0中的类（以及Python 2.6中的新式类）是`type`类的实例，并且实例对象是它们的类的实例。实际上，类现在有了连接到`type`的一个`__class__`，就像是一个实例有了链接到创建它的类的`__class__`：

```

C:\misc> c:\python30\python
>>> class C: pass           # 3.0 class object (new-style)
...
>>> X = C()                 # Class instance object

>>> type(X)                 # Instance is instance of class
<class '__main__.C'>
>>> X.__class__              # Instance's class
<class '__main__.C'>

>>> type(C)                 # Class is instance of type
<class 'type'>
>>> C.__class__              # Class's class is type
<class 'type'>

```

特别注意这里的最后两行——类是`type`类的实例，就像常规的实例是一个类的实例一样。在Python 3.0中，这对于内置类和用于定义的类类型都是成立的。实际上，类根本不是一个独立的概念：它们就是用户定义的类型，并且`type`自身也是由一个类定义的。

在Python 2.6中，对于派生自`object`的新式类，情况也是如此，因此，这保证了Python 3.0的类行为：

```
C:\misc> c:\python26\python
>>> class C(object): pass          # In 2.6 new-style classes,
...                                # classes have a class too
>>> X = C()

>>> type(X)
<class '__main__.C'>
>>> type(C)
<type 'type'>

>>> X.__class__
<class '__main__.C'>
>>> C.__class__
<type 'type'>
```

然而，Python 2.6中的传统类略有不同——因为它们反映了老Python版本中的类模型，它们没有一个`__class__`链接，并且像Python 2.6中的内置类型一样，它们是`type`的实例，而不是一个类型实例：

```
C:\misc> c:\python26\python
>>> class C: pass                  # In 2.6 classic classes,
...                                # classes have no class themselves
>>> X = C()

>>> type(X)
<type 'instance'>
>>> type(C)
<type 'classobj'>

>>> X.__class__
<class __main__.C at 0x005F85A0>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
```

元类是Type的子类

那么，为什么我们说类是Python 3.0中的一个`type`类的实例？事实证明，这是允许我们编写元类的钩子。由于类型的概念类似于今天的类，所以我们可以用常规的面向对象技术子类`type`，并且用类语法来定制它。由于类实际上是`type`类的实例，从`type`的定制的子类创建类允许我们实现各种定制的类。更详细地说，这是很自然的解决方案——在Python 3.0中以及在Python 2.6的新式类中：

- `type`是产生用户定义的类的一个类。
- 元类是`type`类的一个子类。

- 类对象是type类的一个实例，或一个子类。
- 实例对象产生于一个类。

换句话说，为了控制创建类以及扩展其行为的方式，我们所需做的只是指定一个用户定义的类创建自一个用户定义的元类，而不是常规的type类。

注意，这个**类型实例**关系与**继承**并不完全相同：用户定义的类可能也拥有超类，它们及其实例从那里继承属性（继承超类在class语句的圆括号中列出，并且出现在一个类的__bases__元组中）。类创建自的类型，以及它是谁的实例，这是不同的关系。下一小节将描述Python所遵从的实现这种实例关系的过程。

Class语句协议

子类化type类以定制它，这其实只是元类背后的魔力的一半。我们仍然需要把一个类的创建指向元类，而不是默认type。为了完全理解这是如何安排的，我们还需要知道class语句如何完成其工作。

我们已经学习过，当Python遇到一条class语句，它会运行其嵌套的代码块以创建其属性——所有在嵌套代码块的顶层分配的名称都产生结果的类对象中的属性。这些名称通常是嵌套的def所创建的方法函数，但是，它们也可以是分配来创建由所有实例共享的类数据的任意属性。

从技术上讲，Python遵从一个标准的协议来使这发生：在一条class语句的末尾，并且在运行了一个命名控件字典中的所有嵌套代码之后，它调用type对象来创建class对象：

```
class = type(classname, superclasses, attributedict)
```

type对象反过来定义了一个__call__运算符重载方法，当调用type对象的时候，该方法运行两个其他的方法：

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(class, classname, superclasses, attributedict)
```

__new__方法创建并返回了新的class对象，并且随后__init__方法初始化了新创建的对象。正如我们稍后将看到的，这是type的元类子类通常用来定制类的钩子。

例如，给定一个如下所示的类定义：

```
class Spam(Eggs):           # Inherits from Eggs
    data = 1                 # Class data attribute
    def meth(self, arg):    # Class method attribute
        pass
```

Python将会从内部运行嵌套的代码块来创建该类的两个属性（`data`和`meth`），然后在`class`语句的末尾调用`type`对象，产生`class`对象：

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

由于这个调用在`class`语句的末尾进行，它是用来扩展或处理一个类的、理想的钩子。技巧在于，用将要拦截这个调用的一个定制子类来替代类型，下一节将展示如何做到这一点。

声明元类

正如我们刚才看到的，类默认是`type`类创建的。要告诉Python用一个定制的元类来创建一个类，直接声明一个元类来拦截常规的类创建调用。怎么做到这点，依赖于你使用哪个Python版本。在Python 3.0中，在类标题中把想要的元类作为一个关键字参数列出来：

```
class Spam(metaclass=Meta):                                # 3.0 and later
```

继承超类也可以列在标题中，在元类之前。例如，在下面的代码中，新的类`Spam`继承自`Eggs`，但也是`Meta`的一个实例并且由`Meta`创建：

```
class Spam(Eggs, metaclass=Meta):                          # Other supers okay
```

我们可以在Python 2.6中得到同样的效果，但是，我们必须不同地指定元类——使用一个类属性而不是一个关键字参数。为了使其成为一个新式类，需要`object`派生，并且这种形式在Python 3.0中不再有效，而是作为属性直接忽略：

```
class spam(object):                                       # 2.6 version (only)
    __metaclass__ = Meta
```

在Python 2.6中，一个全局模块`__metaclass__`变量也可以用来把模块中的所有类链接到一个元类。这在Python 3.0中不再支持，因为它有意作为一个临时性措施，使得更容易预设为新式类而不用从`object`派生每个类。

当以这些方式声明的时候，创建类对象的调用在`class`语句的底部运行，修改为调用元类而不是默认的`type`：

```
class = Meta(classname, superclasses, attributedict)
```

由于元类是`type`的一个子类，所以`type`类的`__call__`把创建和初始化新的类对象的调用委托给元类，如果它定义了这些方法的定制版本：

```
Meta.__new__(Meta, classname, superclasses, attributedict)
Meta.__init__(class, classname, superclasses, attributedict)
```

为了展示，这里再次给出前一节的例子，用Python 3.0的元类声明扩展：

```
class Spam(Eggs, metaclass=Meta):      # Inherits from Eggs, instance of Meta
    data = 1                          # Class data attribute
    def meth(self, arg):               # Class method attribute
        pass
```

在这条class语句的末尾，Python内部运行如下的代码来创建class对象：

```
Spam = Meta('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

如果元类定义了__new__或__init__的自己版本，在此处的调用期间，它们将依次由继承的type类的__call__方法调用，以创建并初始化新类。下一节将介绍我们如何编写元类谜题的最后一块。

编写元类

到目前为止，我们已经看到了Python如何把类创建调用指向一个元类，如果提供了一个元类的话。然而，我们实际如何编写一个元类来定制type呢？

事实上，我们已经知道了大多数情况——用常规的Python class语句和语法来编写元类。唯一的实质区别是，Python在一条class语句的末尾自动调用它们，而且它们必须通过type超类附加到预期的接口。

基本元类

可能你能够编写的最简单元类只是带有一个__new__方法的type的子类，该方法通过运行type中的默认版本来创建类对象。像这样的元类__new__，通过继承自type的__new__方法而运行。它通常执行所需的任何定制并且调用type的超类的__new__方法来创建并运行新的类对象：

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Run by inherited type.__call__
        return type.__new__(meta, classname, supers, classdict)
```

这个元类实际并没有做任何事情（我们可能也会让默认的type类创建类），但是它展示了将元类接入元类钩子中以定制——由于元类在一条class语句的末尾调用，并且因为type对象的__call__分派到了__new__和__init__方法，所以我们在这些方法中提供的代码可以管理从元类创建的所有类。下面是应用中的实例，将打印添加到元类和文件以便追踪：

```
class MetaOne(type):
```

```

def __new__(meta, classname, supers, classdict):
    print('In MetaOne.new:', classname, supers, classdict, sep='\n...')
    return type.__new__(meta, classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaOne):    # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg):                # Class method attribute
        pass

print('making instance')
X = Spam()
print('data:', X.data)

```

在这里，Spam继承自Eggs并且是MetaOne的一个实例，但是X是Spam的一个实例并且继承自它。当这段代码在Python 3.0下运行的时候，注意在class语句的末尾是如何调用元类的，在我们真正创建一个实例之前——元类用来处理类，并且类用来处理实例：

```

making class
In MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AEBA08>}
making instance
data: 1

```

定制构建和初始化

元类也可以接入__init__协议，由type对象的__call__调用：通常，__new__创建并返回了类对象，__init__初始化了已经创建的类。元类也可以用做在创建时管理类的钩子：

```

class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('In MetaOne init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaOne):    # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg):                # Class method attribute
        pass

print('making instance')

```

```
X = Spam()
print('data:', X.data)
```

在这个例子中，类初始化方法在类构建方法之后运行，但是，两者都在`class`语句最后运行，并且在创建任何实例之前运行：

```
making class
In MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
In MetaOne.init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
making instance
data: 1
```

其他元类编程技巧

尽管重新定义`type`超类的`__new__`和`__init__`方法是元类向类对象创建过程插入逻辑的最常见方法，其他的方案也是可能的。

使用简单的工厂函数

例如，元类根本不是真的需要类。正如我们所学习的，`class`语句发布了一条简单的调用，在其处理的最后创建了一个类。因此，实际上任何可调用对象都可以用作一个元类，只要它接收传递的参数并且返回与目标类兼容的一个对象。实际上，一个简单的对象工厂函数就像一个类一样工作：

```
# A simple function can serve as a metaclass too

def MetaFunc(classname, supers, classdict):
    print('In MetaFunc: ', classname, supers, classdict, sep='\n...')
    return type(classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaFunc):           # Run simple function at end
    data = 1                                    # Function returns class
    def meth(self, args):
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

运行的时候，在声明`class`语句的末尾调用该函数，并且它返回期待的新的类对象。该函数直接捕获`type`对象的`__call__`通常会默认拦截的调用：

```
making class
In MetaFunc:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B8B6A8>}
making instance
data: 1
```

用元类重载类创建调用

由于它们涉及常规的OOP机制，所以对于元类来说，也可能直接在一条`class`语句的末尾捕获创建调用，通过定义`type`对象的`__call__`。然而，所需的协议有点多：

```
# __call__ can be redefined, metas can have metas

class SuperMeta(type):
    def __call__(meta, classname, supers, classdict):
        print('In SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        return type.__call__(meta, classname, supers, classdict)

class SubMeta(type, metaclass=SuperMeta):
    def __new__(meta, classname, supers, classdict):
        print('In SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('In SubMeta init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=SubMeta):
    data = 1
    def meth(self, arg):
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

当这段代码运行的时候，所有3个重新定义的方法都依次运行。这基本上就是`type`对象默认做的事情：

```
making class
In SuperMeta.call:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
```



```

In SubMeta.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
In SubMeta init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
making instance
data: 1

```

用常规类重载类创建调用

前面的例子被复杂化了，事实是用元类来创建类对象，但并不产生它们自己的实例。因此，元类名查找规则与我们所习惯的方式有点不同。例如，`__call__`方法在一个对象的类中查找；对于元类，这意味着一个元类的元类。

要使用常规的基于继承的名称查找，我们可以用常规类和实例实现同样的效果。如下的输出和前面的版本相同，但是注意，`__new__`和`__init__`在这里必须有不同的名称，否则，当创建SubMeta实例的时候它们会运行，而不是随后作为一个元类调用：

```

class SuperMeta:
    def __call__(self, classname, supers, classdict):
        print('In SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        Class = self._New__(classname, supers, classdict)
        self._Init__(Class, classname, supers, classdict)
        return Class

class SubMeta(SuperMeta):
    def _New__(self, classname, supers, classdict):
        print('In SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def _Init__(self, Class, classname, supers, classdict):
        print('In SubMeta init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=SubMeta()):          # Meta is normal class instance
    data = 1                                    # Called at end of statement
    def meth(self, arg):
        pass

print('making instance')
X = Spam()
print('data:', X.data)

```

尽管这种替代形式有效，但大多数元类通过重新定义`type`超类的`__new__`和`__init__`完

成它们的工作。实际上，这通常需要尽可能多的控制，并且它往往比其他方法简单。然而，我们随后将看到，一个简单的基于函数的元类往往更像一个类装饰器一样工作，这允许元类管理实例以及类。

实例与继承的关系

由于元类以类似于继承超类的方式来制定，因此它们乍看上去有点容易令人混淆。一些关键点有助于概括和澄清这一模型：

- **元类继承自type类。**尽管它们有一种特殊的角色元类，但元类是用`class`语句编写的，并且遵从Python中有用的OOP模型。例如，就像`type`的子类一样，它们可以重新定义`type`对象的方法，需要的时候重载或定制它们。元类通常重新定义`type`类的`__new__`和`__init__`，以定制类创建和初始化，但是，如果它们希望直接捕获类末尾的创建调用的话，它们也可以重新定义`__call__`。尽管元类不常见，它们甚至是返回任意对象而不是`type`子类的简单函数。
- **元类声明由子类继承。**在用户定义的类中，`metaclass=M`声明由该类的子类继承，因此，对于在超类链中继承了这一声明的每个类的构建，该元类都将运行。
- **元类属性没有由类实例继承。**元类声明指定了一个实例关系，它和继承不同。由于类是元类的实例，所以元类中定义的行为应用于类，而不是类随后的实例。实例从它们的类和超类获取行为，但是，不是从任何元类获取行为。从技术上讲，实例属性查找通常只是搜索实例及其所有类的`__dict__`字典；元类不包含在实例查找中。

为了说明最后两点，考虑如下的例子：

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
        print('toast')

class Super(metaclass=MetaOne):
    def spam(self):
        print('spam')

class C(Super):
    def eggs(self):
        print('eggs')

X = C()
X.eggs()
X.spam()
X.toast()
```

Redefine type method
MetaOne run twice for two classes
Superclass: inheritance versus instance
Classes inherit from superclasses
But not from metaclasses
Inherited from C
Inherited from Super
Not inherited from metaclass

当这段代码运行的时候，元类处理两个客户类的构建，并且实例继承类属性而不是元类属性：

```
In MetaOne.new: Super
In MetaOne.new: C
eggs
spam
AttributeError: 'C' object has no attribute 'toast'
```

尽管细节很有意义，但是，在处理元类的时候，头脑中保持大概念很重要。像我们已经在前面见到的元类将会对声明它们的每个子类自动运行。和我们在前面见到的辅助函数方法不同，这样的类将会自动获取元类所提供的任何扩展。此外，修改这样的扩展只需要在一个地方编码——元类中，这简化了所需要进行的修改。就像Python中如此众多的工具一样，元类通过除去冗余简化了维护工作。然而，要完全展示其功能，我们需要继续看一些更大的示例。

示例：向类添加方法

在本节以及下一节，我们将学习两个常见的元类示例：向一个类添加方法，以及自动装饰所有的方法。这只是元类众多用处中的两个，它们将占用本章剩余的篇幅。再一次说明，你应该在Web上查找以了解更多的高级应用。这些示例代表了元类的应用，因此它们足以说明基础知识。

此外，这两个示例都给了我们机会来对比类装饰器和元类——第一个示例比较了类扩展和实例包装的基于元类和基于装饰器的实现；第二个实例首先对元类应用一个装饰器，然后应用另一个装饰器。你将会看到，这两个工具往往是可以交换的，甚至是互补的。

手动扩展

在本章前面，我们看到了以不同方法向类添加方法来扩展类的骨干代码。正如我们所见到的，如果在编写类的时候，静态地知道额外的方法，那么简单的基于类的继承已经足够了。通过对象嵌入的组合，往往也能够实现同样的效果。然而，对于更加动态的场景，有时候需要其他的技术，辅助函数通常足够了，但元类提供了一种更加明确的结构并且减少了未来修改的成本。

让我们在这里通过实际代码把这些思想付诸实践。考虑下面示例中的手动类扩展——它向两个类添加了两个方法，在创建了它们之后：

```
# Extend manually - adding new methods to classes

class Client1:
    def __init__(self, value):
```

```

        self.value = value
    def spam(self):
        return self.value * 2

class Client2:
    value = 'ni?'

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'ham'

Client1.eggs = eggsfunc
Client1.ham = hamfunc

Client2.eggs = eggsfunc
Client2.ham = hamfunc

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))

```

这是有效的，因为方法总是在类创建之后分配给一个类，只要分配的方法是带有一个额外的第一个参数以接收主体`self`示例的函数，这个参数可以用来访问类实例中可用的状态信息，即便函数独立于类定义。

当这段代码运行的时候，我们接收到在第一个类的代码中编写的一个方法输出，以及在此后添加到类中的两个方法的输出：

```

Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham

```

这种方法在独立的情况下工作得很好，并且可以在运行时任意地填充一个类。但它有一个潜在的主要缺点，对于需要这些方法的每个类，我们必须重复扩展代码。在我们的例子中，对两个类都添加两个方法还不是太繁琐，但是，在更为复杂的环境中，这种方法可能是耗时的而且容易出错。如果我们曾经忘记一致地这么做，或者我们需要修改扩展，就可能遇到问题。

基于元类的扩展

尽管手动扩展有效，但在较大的程序中，如果可以对整个一组类自动应用这样的修改，

可能会更好。通过这种方式，我们避免了对任何给定的类修改扩展的机会。此外，在单独位置编写扩展更好地支持了未来的修改——集合中的所有类都将自动接收修改。

满足这一目标的一种方式就是使用元类。如果我们在元类中编写扩展，那么声明了元类的每个类都将统一且正确地扩展，并自动地接收未来做出的任何修改。如下的代码展示了这一点：

```
# Extend with a metaclass - supports future changes better

def eggfunc(obj):
    return obj.value * 4
def hamfunc(obj, value):
    return value + 'ham'

class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

class Client2(metaclass=Extender):
    value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

这一次，两个客户类都使用新的方法扩展了，因为它们是执行扩展的元类的实例。运行的时候，这个版本的输出和前面相同，我们没有做代码所做的修改，我们只是重构它们以便更整齐地封装修改：

```
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```

注意，这个示例中的元类仍然执行相当静态的工作：把两个已知的方法添加到声明了元类的每个类。实际上，如果我们所需要做的总是向一组类添加相同的两个方法，我们也

可以将它们编写为常规的超类并在子类中继承它们。然而，实际上，元类结构支持更多的动态行为。例如，主体类也可以基于运行时的任意逻辑配置：

```
# Can also configure class based on runtime tests

class MetaExtend(type):
    def __new__(meta, classname, supers, classdict):
        if sometest():
            classdict['eggs'] = eggsfunc1
        else:
            classdict['eggs'] = eggsfunc2
        if someothertest():
            classdict['ham'] = hamfunc
        else:
            classdict['ham'] = lambda *args: 'Not supported'
        return type.__new__(meta, classname, supers, classdict)
```

元类与类装饰器的关系：第二回合

如果本章中的示例还没有让你大脑爆炸，请记住，上一章的类装饰器常常和本章的元类在功能上有重合。这源自于如下的事实：

- 在class语句的末尾，类装饰器把类名重新绑定到一个函数的结果。
- 元类通过在一条class语句的末尾把类对象创建过程路由到一个对象来工作。

尽管这些是略有不同的模型，实际上，它们通常可实现同样的目标，虽然采用的方式不同。实际上，类装饰器可以用来管理一个类的实例以及类自身。尽管装饰器可以自然地管理类，然而，用元类管理实例有些不那么直接。元类可能最好用于类对象管理。

基于装饰器的扩展

例如，前面小节的元类示例，像创建的一个类添加方法，也可以用一个类装饰器来编写。在这种模式下，装饰器大致与元类的__init__方法对应，因为在调用装饰器的时候，类对象已经创建了。也与元类类似，最初的类类型是保留的，因为没有插入包装器对象层。如下的输出与前面的元类代码的输出相同：

```
# Extend with a decorator: same as providing __init__ in a metaclass

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'ham'

def Extender(aClass):
    aClass.eggs = eggsfunc          # Manages class, not instance
    aClass.ham = hamfunc           # Equiv to metaclass __init__
    return aClass
```

```

@Extender
class Client1:                                # Client1 = Extender(Client1)
    def __init__(self, value):                # Rebound at end of class stmt
        self.value = value
    def spam(self):
        return self.value * 2

@Extender
class Client2:
    value = 'ni?'

X = Client1('Ni!')                            # X is a Client1 instance
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))

```

换句话说，至少在某些情况下，装饰器可以像元类一样容易地管理类。反过来就不那么直接了，元类可以用来管理实例，但是只有有限的力量。下一小节将说明这点。

管理实例而不是类

正如我们已经见到的，类装饰器常常可以和元类一样充当**类管理**角色。元类往往和装饰器一样充当**实例管理**的角色，但是，这更复杂一点。即：

- 类装饰器可以管理类和实例。
- 元类可以管理类和实例，但是管理实例需要一些额外工作。

也就是说，某些应用可能用一种方法或另一种方法编写更好。例如，前一章中的类装饰器示例，无论何时，获取一个类实例的任意常规命名的属性的时候，它用来打印一条跟踪消息：

```

# Class decorator to trace external instance attribute fetches

def Tracer(aClass):                            # On @ decorator
    class Wrapper:
        def __init__(self, *args, **kargs):    # On instance creation
            self.wrapped = aClass(*args, **kargs) # Use enclosing scope name
        def __getattr__(self, attrname):
            print('Trace:', attrname)          # Catches all but .wrapped
            return getattr(self.wrapped, attrname) # Delegate to wrapped object
    return Wrapper

@Tracer
class Person:                                # Person = Tracer(Person)
    def __init__(self, name, hours, rate):    # Wrapper remembers Person
        self.name = name
        self.hours = hours

```

```

        self.rate = rate                                # In-method fetch not traced
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)                             # bob is really a Wrapper
print(bob.name)                                           # Wrapper embeds a Person
print(bob.pay())                                          # Triggers __getattr__

```

这段代码运行的时候，装饰器使用类名重新绑定来把实例对象包装到一个对象中，该对象在如下的输出中给出跟踪行：

```

Trace: name
Bob
Trace: pay
2000

```

尽管用一个元类也可能实现同样的效果，但它似乎概念上不太直接明了。元类明确地设计来管理类对象创建，并且它们有一个为此目的而设计的接口。要使用元类来管理实例，我们必须依赖一些额外力量。如下的元类和前面的装饰器具有同样的效果和输出：

```

# Manage instances like the prior example, but with a metaclass

def Tracer(classname, supers, classdict):                # On class creation call
    aClass = type(classname, supers, classdict)          # Make client class
    class Wrapper:
        def __init__(self, *args, **kargs):              # On instance creation
            self.wrapped = aClass(*args, **kargs)
        def __getattr__(self, attrname):
            print('Trace:', attrname)                    # Catches all but .wrapped
            return getattr(self.wrapped, attrname)       # Delegate to wrapped object
    return Wrapper

class Person(metaclass=Tracer):                          # Make Person with Tracer
    def __init__(self, name, hours, rate):               # Wrapper remembers Person
        self.name = name
        self.hours = hours
        self.rate = rate                                # In-method fetch not traced
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)                             # bob is really a Wrapper
print(bob.name)                                           # Wrapper embeds a Person
print(bob.pay())                                          # Triggers __getattr__

```

这也有效，但是它依赖于两个技巧。首先，它必须使用一个简单的函数而不是一个类，因为`type`子类必须附加给对象创建协议。其次，必须通过手动调用`type`来手动创建主体类；它需要返回一个实例包装器，但是元类也负责创建和返回主体类。其实，在这个例子中，我们将使用元类协议来模仿装饰器，而不是按照相反的做法。由于它们都在一条`class`语句的末尾运行，所以在很多用途中，它们都是同一方法的变体。元类版本运行的时候，产生与装饰器版本同样的输出：


```
Trace: name
Bob
Trace: pay
2000
```

你应该自己研究这两个示例版本，以权衡其利弊。通常，元类可能更适合于类管理，因为它们就设计为如此。类装饰器可以管理实例或类，然而，它们不是更高级元类用途的最佳选择（我们没有足够篇幅在本书中介绍，如果你在阅读完本章后，还想学习关于装饰器和元类的更多内容，请在Web上搜索或参阅Python标准手册的内容）。下一小节用一个更为常见的例子来结束本章，自动对一个类的方法应用操作。

示例：对方法应用装饰器

正如我们在前一小节中见到的，由于它们都在一条`class`语句的末尾运行，所以元类和装饰器往往可以**互换地**使用，虽然语法不同。在这二者之间的选择，在很多情况下是随意的。也可能将二者**组合**起来使用，作为互补的工具。在本小节中，我们将展示一个示例，它就是这样的组合——对一个类的所有方法应用一个函数装饰器。

用装饰器手动跟踪

在前面的一章中，我们编写了两个函数装饰器，其中之一跟踪和统计对一个装饰函数的调用，另一个计时这样的调用。它们采用各种形式，其中的一些对于函数和方法都适用，另一些并不适用。下面把两个装饰器的最终形式收入一个模块文件中，以便重用或引用：

```
# File mytools.py: assorted decorator tools

def tracer(func):                                # Use function, not class with __call__
    calls = 0                                    # Else self is decorator instance only
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

import time
def timer(label='', trace=True):                 # On decorator args: retain args
    def onDecorator(func):                       # On @: retain decorated func
        def onCall(*args, **kwargs):           # On calls: call original
            start = time.clock()                # State is scopes + func attr
            result = func(*args, **kwargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
```

```

        values = (label, func.__name__, elapsed, onCall.alltime)
        print(format % values)
        return result
    onCall.alltime = 0
    return onCall
return onDecorator

```

正如我们在上一章了解到的，要手动使用这些装饰器，我们直接从模块导入它们，并且在想要跟踪或计时的每个方法前编写@装饰语法：

```

from mytools import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

    @tracer
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())

```

giveRaise = tracer(giveRaise)
onCall remembers giveRaise

lastName = tracer(lastName)

Runs onCall(sue, .10)
Runs onCall(bob), remembers lastName

这段代码运行时，我们得到了如下输出——对装饰方法的调用指向了拦截逻辑，并且随后委托调用，因为最初的方法名已经绑定到了装饰器：

```

call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones

```

用元类和装饰器跟踪

前一小节的手动装饰方法是有效的，但是它需要我们在想要跟踪的**每个**方法前面添加装饰语法，并且在不再想要跟踪的使用后删除该语法。如果想要跟踪一个类的每个方法，在较大的程序中，这会变得很繁琐。如果我们可以对一个类的所有方法自动地应用跟踪装饰器，那将会更好。

有了元类，我们确实可以做到——因为它们在构建一个类的时候运行，它们是把装饰包装器添加到一个类方法中的自然地方。通过扫描类的属性字典并测试函数对象，我们可以通过装饰器自动运行方法，并且把最初的名称重新绑定到结果。其效果与装饰器的自动方法名重新绑定是相同的，但是，我们可以更全面地应用它：

```
# Metaclass that adds tracing decorator to every method of a client class

from types import FunctionType
from mytools import tracer

class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:                # Method?
                classdict[attr] = tracer(attrval)            # Decorate it
        return type.__new__(meta, classname, supers, classdict) # Make class

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

当这段代码运行的时候，结果与前面是相同的——方法的调用将首先指向跟踪装饰器以跟踪，然后传递到最初的方法：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

我们这里看到的的就是装饰器和元类组合工作的结果——在类创建的时候，元类自动把函数装饰器应用于每个方法，并且函数装饰器自动拦截方法调用，以便在此输出中打印出跟踪消息。这一组合能够有效，得益于两种工具的通用性。

把任何装饰器应用于方法

前面的元类示例只对一个特定的函数装饰器有效，即跟踪。然而，将这个通用化以把任何装饰器应用到一个类的所有方法，实际的意义不大。我们所要做的是，添加一个外围作用域层，以保持想要的装饰器，这很像是我们在上一章对装饰器所做的。如下的示例，编写了这样的一个泛化，然后使用它再次应用跟踪装饰器：

```
# Metaclass factory: apply any decorator to all methods of a class

from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return type.__new__(meta, classname, supers, classdict)
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):    # Apply a decorator to all
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

当这段代码运行的时候，输出再次与前面的示例相同——最终我们仍然在一个客户类中用跟踪器函数装饰器装饰了每个方法，但是，我们以一种更为通用的方式做到了这点：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

现在，要对方法应用一种不同的装饰器，我们只要在类标题行替换装饰器名称。例如，要使用前面介绍的计时器函数装饰器，定义类的时候，我们可以使用如下示例标题行的最后两行中的任何一个——第一个接收了定时器的默认参数，第二个指定了标签文本：

```

class Person(metaclass=decorateAll(tracer)):           # Apply tracer
class Person(metaclass=decorateAll(timer())):         # Apply timer, defaults
class Person(metaclass=decorateAll(timer(label='**'))): # Decorator arguments

```

注意，这种方法不支持对每个方法不同的非默认装饰器参数，但是，它可以传递到装饰器参数中以应用到所有方法，就像这里所做的一样。为了进行测试，使用这些元类声明的最后一个来应用定时器，并且在脚本的末尾添加如下的行：

```

# If using timer: total time per method

print('-'*40)
print('%5f' % Person.__init__.alltime)
print('%5f' % Person.giveRaise.alltime)
print('%5f' % Person.lastName.alltime)

```

新的输出如下所示——现在，元类把方法包装到了定时器装饰器中，以便我们可以说出针对类的每个方法的每次调用花费多长时间：

```

**__init__: 0.00001, 0.00001
**__init__: 0.00001, 0.00002
Bob Smith Sue Jones
**giveRaise: 0.00001, 0.00001
110000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Smith Jones
-----
0.00002
0.00001
0.00002

```

元类与类装饰器的关系：第三回合

类装饰器也与元类有交叉。如下的版本，用一个类装饰器替换了前面的示例中的元类。它定义并使用一个类装饰器，该装饰器把一个函数装饰器应用于一个类的所有方法。然而，前一句话听起来更像是禅语而不像是技术说明，这所有的工作相当自然——Python的装饰器支持任意的嵌套和组合：

```

# Class decorator factory: apply any decorator to all methods of a class

from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval))    # Not __dict__
    return DecoDecorate

```

```

        return aClass
    return DecoDecorate

@decorateAll(tracer)
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())

```

Use a class decorator
Applies func decorator to methods
Person = decorateAll(..)(Person)
Person = DecoDecorate(Person)

当这段代码运行的时候，类装饰器把跟踪器函数装饰器应用于每个方法，并且在调用时产生一条跟踪消息（输出和本示例前面的元类版本相同）：

```

call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones

```

注意，类装饰器返回最初的、扩展的类，而不是其包装器层（当返回包装示例对象的时候更为常见）。就像是元类版本一样，我们保留了最初的类的类型——`Person`的一个实例，而不是某个包装器类的实例。实际上，这个类装饰器只是处理了类创建，实例创建调用根本没有拦截。

这种区别对于需要对实例进行类型测试以产生最初的类而不是包装器的程序有影响。当扩展一个类而不是一个实例的时候，类装饰器可以保持最初的类类型，类的方法不是它们最初的函数，因为它们绑定到了装饰器，但是这在实际中并不重要，并在元类替代方案中也是如此。

还要注意，和元类版本一样，这种结构不支持每个方法不同的函数装饰器参数，但是，如果它们适用于所有方法的话，可以处理这种参数。例如，要使用这种方法应用计时器装饰器，下面声明行的最后两个中的任何一个就够了，如果类定义的代码和前面一样的话——第一个使用装饰器参数默认，第二个显式地提供了一个参数：

```

@decorateAll(tracer)                # Decorate all with tracer

@decorateAll(timer())               # Decorate all with timer, defaults

@decorateAll(timer(label='@@'))     # Same but pass a decorator argument

```

和前面一样，让我们使用这些装饰器行的最后一个，并且在脚本的末尾添加如下代码，以用一种不同的装饰器来测试示例：

```

# If using timer: total time per method

print('- '*40)
print('%5f' % Person.__init__.alltime)
print('%5f' % Person.giveRaise.alltime)
print('%5f' % Person.lastName.alltime)

```

同样的输出出现了，对于每个方法，我们针对每次调用和所有调用获取了计时数据，但是，我们已经给计数器装饰器传递了一个不同的标签参数：

```

@@__init__: 0.00001, 0.00001
@@__init__: 0.00001, 0.00002
Bob Smith Sue Jones
@@giveRaise: 0.00001, 0.00001
110000.0
@@lastName: 0.00001, 0.00001
@@lastName: 0.00001, 0.00002
Smith Jones
-----
0.00002
0.00001
0.00002

```

正如你所看到的，元类和类装饰器不仅常常可以交换，而且通常是互补的。它们都对于定制和管理类和实例对象，提供了高级但强大的方法，因为这二者最终都允许我们在类创建过程中插入代码。尽管某些高级应用可能用一种方式或另一种方式编码更好，但在很多情况下，我们选择或组合这两种工具的方法，很大程度上取决于你。

“可选的”语言功能

我在本章开始处引用了一句话，提到元类不是99%的Python程序员都感兴趣的，用以强调它们相对难以理解。这句话不是很准确，只是用一个数字来说明。

说这句话的人是我最初使用Python时的一个朋友，并且，我并不是想不公平地嘲笑某人。另外，实际上，在这本书中，对于语言功能的灰色性，我也常常做出这样的表述。

然而问题在于，这样的语句真的只是适用于单独工作的人而且只是那些可能使用他们自己曾经编写的代码的人。只要一个组织中的**任何人**使用了一项“可选的”高级语言功能，它就不再是可选的——它有效地施加于组织中的**每个人**身上。对于你在系统中所使用的外部开发的软件来说，也是如此——如果软件的作者使用了一项高级功能，它对你来说完全不再是可选的，因为你必须理解该功能以便使用或修改代码。

这种观察适用于在本章开始处列出的所有高级工具——装饰器、特性、描述符、元类，等等。如果与你一起工作的任何人或任何程序用到了它们，它们自动地变成所需的知识基础的一部分。也就是说，没有什么是真正“可选的”。我们当中的大多数数人不会去挑选或选择。

这就是为什么一些Python前辈（包括我自己）有时候悲叹，Python似乎随着时间的流逝变得更大并且更复杂了。由老手添加的新功能似乎已经增加了对初学者的智力障碍。尽管Python的核心思想，例如动态类型和内置类型，基本保持相同。它的高级附加功能，也变成了任何Python程序员所必须阅读的。正因为此，我选择在这里介绍这些主题，尽管在前面的版本中并没介绍它们。如果高级内容就在你必须理解的代码之中，那么省略它们是不可能的。

另外一方面，很多新的学习者可以挑选所需的高级话题。坦率地讲，应用程序员可能会把大多数的时间花在**处理库和扩展**上，而不是高级的并且有时候颇为不可思议的语言功能上。例如，本书的后续篇《Programming Python》，处理大多数把Python与应用库结合起来完成的任务，例如GUI、数据库以及Web，而不介绍深奥的语言工具。

这一增长的优点是，Python已经变得更为**强大**。当我们用好它的时候，像装饰器或元类这样的工具不仅毫无辩驳的“酷”，而且允许有创意的程序员来构建更为灵活和有用的API供其他程序员使用。正如我们已经看到的，它们也可以为封装和维护问题提供很好的解决方案。

是否使用所需的Python知识的潜在扩展，取决于你。遗憾的是，一个人的技能水平往往默认决定了这个问题——很多高级的程序员喜欢较为高级的工具，并且往往忘记它们对其他阵营的影响。幸运的是，这不是绝对的；好的程序员也理解**简单是最好的工程**，并且高级工具也应该只在需要的时候使用。对任何编程语言来说，这都是成立的，但是，特别是在像Python这样的语言中，它作为一种扩展工具广泛地展示给新的和初学的程序员。

如果你仍然不接受这一观点，别忘了，有很多Python用户不习惯基本的OOP和类。相信我的判断，我曾经遇到过数以千计这样的人。基于Python的系统需要它们的用户掌握元类、装饰器之间的细微差别，并且可能由此扩展它们的市场预期。

本章小结

在本章中，我们学习了元类并且介绍了它们的实际使用示例。元类允许我们接入Python的类创建协议，以便管理和扩展用户定义的类。由于它们使这一过程自动化了，因此它们可以为API编写者提供更好的解决方案，而且手动编码或使用辅助函数。由于它们封装了这样的代码，所以它们可以比其他的方法更好地减少维护成本。

在此过程中，我们还看到了类装饰器与元类的角色往往是如何交叉的：由于它们都在一条`class`语句的末尾运行，因而它们有时候可以互换地使用。类装饰器可以用来管理类和实例对象，元类也可以，尽管它们更直接地以类为目标。

由于本章介绍了一个高级话题，因此我们将通过一些练习题来回顾基础知识（如果你已经在关于元类的本章中读到了这里，你应该已经接受额外的奖励）。这是本书的最后一部分，我们将给出最后一部分的练习。确保查看后面的附录中关于安装的提示，以及前面各部分的练习的解答。

一旦完成了练习，你就真正完成了本书。既然你知道了Python里里外外的知识，那么下一步应该是选择接收它，即研究库、技巧，以及你的应用领域所能用到的工具。由于Python应用如此广泛，你可能会找到丰富的资源以在几乎可以考虑到的所有应用领域中使用它，从GUI、Web、数据库到数值计算、机器人以及系统管理。

Python由此变得真正有趣，但是这也只是本书的介绍到此结束，而另一段故事开始了。阅读完本书之后，要获取提示，可以参见前言中的推荐资料部分的列表。祝你好运。并且当然，“总是要看到生命的阳光灿烂的一面！”

本章习题

1. 什么是元类？
2. 如何声明一个类的元类？
3. 在管理类方面，类装饰器如何与元类重叠？
4. 在管理实例方面，类装饰器如何与元类重叠？

习题解答

1. 元类是用来创建一个类的类。常规的类默认的是`type`类的实例。元类通常是`type`类的子类，它重新定义了类创建协议方法，以便定制在一条`class`语句的末尾发布的类创建调用；它通常会重定义`__new__`和`__init__`方法以接入类创建协议。元类也可

以以其他的方式编码——例如，作为简单函数——但是它们负责为新类创建和返回一个对象。

2. 在Python 3.0及其以后的版本中，在类标题栏使用一个关键字参数：`class C(metaclass=M)`。在Python 2.X中，使用类属性：`__metaclass__ = M`。在Python 3.0中，类标题栏也可以在`metaclass`关键字参数之前命名常规的超类（例如，基类）。
3. 由于二者都是在一条`class`语句的末尾自动触发，因此类装饰器和元类都可以用来管理类。装饰器把一个类名重新绑定到一个可调用对象的结果，而元类把类创建指向一个可调用对象，但它们都是可以用作相同目的的钩子。要管理类，装饰器直接扩展并返回最初的类对象。元类在创建一个类之后扩展它。
4. 由于二者都是在一条`class`语句的末尾自动触发，因此类装饰器和元类都可以用来管理类实例，通过插入一个包装器对象来捕获实例创建调用。装饰器把类名重新绑定到一个可调用对象，而该可调用对象在实例创建时运行以保持最初的类对象。元类可以做同样的事情，但是，它们必须也创建类对象，因此，它们用在这一角色中更复杂一些。

附录

安装和配置

本附录提供其他安装和配置的细节，新接触这类话题的人可以参考这些资源。

安装Python解释器

因为需要用Python解释器运行Python脚本，所以使用Python的第一步通常就是安装Python。除非你的机器上已有一个Python，不然，你就得取得最新版Python，在计算机上安装和配置。每台机器只需安装和配置一次，如果你是运行冻结二进制文件（第2章介绍过）或自安装系统，那就完全不需要这样做了。

Python已经存在了吗

做任何事之前，应该检查机器上是否已有最新版本的Python。如果你用的是Linux、Mac OS X以及一些UNIX系统，Python可能已经安装在你的计算机上，尽管它可能和最新版本相比已经落后了一两个版本。可通过如下方式来检查：

- 在Windows上，查看“开始”→“所有程序”菜单中（位于屏幕左下方）是否有Python。
- 在Mac OS X下，打开一个Terminal窗口(Applications→Utilities→Terminal)，并且在提示符下输入**Python**。
- 在Linux和Unix上，在shell提示符下（有时被称作终端窗口）输入**python**，看看会发生什么事。此外，也可以在常见的位置搜索“python”：`/usr/bin`、`/usr/local/bin`等。

如果找到Python，要确保它是最近的一个版本。尽管任何较新的Python版本对本书中的大多数内容都适用，本书主要关注Python 3.0和Python 2.6，因此，你可能想要安装这两个版本之一来运行本书中的示例。

提到版本，如果你初次接触Python并且不需要处理已有的Python 2.X代码，我建议你从Python 3.0及其以后的版本开始；否则，应该使用Python 2.6。一些流行的基于Python的系统仍然使用旧版本（Python 2.5仍然很普遍），因此，如果你要用已有的系统工作，确保根据你的需要来选用版本；下一小节将介绍从哪里可以获取不同的Python版本。

从哪里获取Python

如果找不到Python，就需要自行安装。幸运的是，Python是开源系统，可在Web上免费获取，而且在大多数平台上安装都很简单。

你总是可以从Python的官方网站<http://www.python.org>获取最新、最好的标准Python版本。寻找网页上的Downloads链接，然后，选择所需平台的版本。你会发现预创建的Windows的自安装文件（点击文件图标就能安装）、针对Mac OS X的安装程序光盘镜像、完整的源代码包（通常在Linux、Unix或OS X机器上编译从而生成解释器），等等。

尽管如今Python是Linux上的标准，我们还是可以在Web上找到针对Linux的RPM（用`rpm`解压它们）。Python的Web站点也连接到站内或站外的各个页面，那里维护了针对其他平台的版本。Google Web搜索是找到Python包的另一种不错的方式。在这些平台中，我们可以找到针对iPod、Palm手机、Nokia手机、PlayStation和PSP、Solaris、AS/400和Windows Mobile的预编译的Python。

如果你发现自己在Windows机器上渴望一个UNIX环境，那么，可能对于安装Cygwin及其Python版本感兴趣（参见<http://www.cygwin.com>）。Cygwin是一个GPL许可的库和工具集，它在Windows机器上提供了完整的UNIX功能，并且它包含一个预编译的Python，它可以使用所提供的所有UNIX工具。

你也会在Linux CD发行版中找到Python。也许是随附在某些产品和计算机系统上，或者和其他Python书籍在一起。这些通常都会比当前版本落后，但通常不会落后太多。

此外，我们可以在其他的免费和商业开发包中找到Python。例如，ActiveState公司将Python作为其ActivePython包的一部分。这个包把标准Python与以下工具结合起来：支持Windows开发的PyWin32，一个名为PythonWin的IDE（第3章介绍过），以及其他常用的扩展包。Python如今还放入了Enthought Python包中，这个包瞄准了科学计算的需求，

而且还放入了*Portable Python*，它预配置来直接从一个便携设备启动。请在Web中搜索以了解更多细节。

最后，如果你对其他Python的实现感兴趣，可以搜索网络，看一看Jython（Python的Java实现）以及IronPython（Python的C#/.NET实现），而它们在第2章都介绍过。这些系统的安装说明不在本书的范围之内。

安装步骤

下载Python后，需要进行安装。安装步骤是与平台相关的，这里主要介绍安装Python平台的一些要点。

Windows

在Windows上，Python是自安装的MSI程序文件，只要双击文件图标，在每个提示文字下回答“*Yes*”或“*Next*”，就可执行默认安装。默认安装包括了Python的文档集以及tkinter（在Python 2.6中叫做Tkinter）、shelve数据库和IDLE GUI的支持。Python 3.0和Python 2.6一般是安装在目录C:\Python30和C:\Python26下的，这在安装时可进行修改。

为了方便，安装之后，Python会出现在“开始”→“所有程序”菜单中。Python的菜单有五个项目，可以快捷地打开常见的任务：打开IDLE用户界面、阅读模块文档、打开交互模式会话、在网页浏览器中阅读Python的标准手册以及卸载。大多数动作都涉及了本书各处所提到的概念细节。

在Windows上安装后，Python会自动注册，在单击Python文件图标时，打开Python文件程序（第3章谈到过这种程序启动技术）。也有可能Windows上通过源代码编译创建Python，但通常并不这样做。

Windows Vista用户要注意：当前Vista版本的安全特性修改了使用MSI安装文件的一些规则。如果Python安装程序无法使用，或者没有把Python放在机器上的正确位置，可以参考本附录中边栏部分寻求帮助。

Linux

在Linux上，Python可能是一个或多个RPM文件，按通常的方式将其解压（更多细节参考RPM的manpage）。根据下载的RPM，Python本身也许是一个文件，而另一个是tkinter GUI和IDLE环境的支持文件。因为Linux是类UNIX系统，下一段也同样适用。

UNIX

在UNIX系统上，Python通常是以C源代码包编译而成。这通常只需解压解文件，运行简单的config和make命令。Python会根据其编译所在的系统，自动配置其创建流

程。尽管这样，要确定你看过了包中的`README`文件从而了解这个流程的细节。因为Python是开放源代码的，其源代码可以免费使用和分发。

在其他平台上，这些细节可能大不一样：例如，要替PalmOS安装Python的Pippy移植版本，你的PDA就得有hotsync操作才行，而Python对Sharp Zaurus Linux PDA来讲，会有一个或多个`.ipk`文件，你只需执行它们就能安装。不过，可执行文件形式和源代码形式的额外安装程序都有完整说明，我们就在这里跳过其更深入的细节。

Windows Vista的Python MSI安装程序

在我编写本书时，Python的Windows自安装程序是`.msi`安装文件。这个格式在Windows XP上工作正常（只需对该文件进行双击，它就会运行），但是在某些Windows Vista版本上可能有些问题。特别是，单击MSI安装程序会使Python安装到机器的C盘根目录上，而不是正确的`C:\PythonXX`。虽然Python在根目录也能工作，但这并不是正确的安装位置。

这是与Vista安全相关的话题。简而言之，MSI文件并不是真正的可执行文件，所以不会正确地继承管理员权限，即使是由administrator用户执行。事实上，MSI文件是通过Windows注册表运行的，其文件名会和MSI安装程序相关联。

这个问题似乎是特定于Python的或者特定于Vista版本的。例如，在一款较新的笔记本上，Python 2.6和Python 3.0的安装都没有问题。要在基于Vista的OQO掌上电脑上安装Python 2.5.2，得使用命令行，强制得到所需要的管理员权限。

如果Python没有安装在正确的位置，下面是解决办法：依次选择“开始”、“所有程序”、“附件”，在“命令提示符”右击鼠标，选择“以系统管理员身份运行”，然后在访问控制对话框中选择“继续”。现在，在“命令提示符”窗口，输入`cd`命令，改变到Python MSI安装文件所在目录（例如，`cd C:\user\downloads`），然后，输入`msiexec /i python-2.5.1.msi`命令，手动运行MSI安装程序。最后，按照一般的GUI交互窗口来完成安装。

当然，这个行为会随时间而发生改变。以后的Vista版本中，这个流程也许就不需要了，而且可能还有其他可行的方法（例如，如果有胆量的话，也可关闭Vista的安全机制）。此外，Python最终会提供不同格式的自安装程序也是有可能的，从而以后解决这个问题——例如，提供真正的可执行文件。尝试任何其他安装方法前，一定要单击安装程序的图标来试一下，看是不是能够正确运作。

配置Python

安装好Python后，要配置一些系统设置，改变Python执行代码的方式（如果你刚开始使用这个语言，完全可以跳过这一节。对于基本的程序来说，通常没必要进行任何系统设置的修改）。

一般来说，Python解释器各部分的行为能够通过环境变量设置和命令行选项来配置。本节我们会简单看一看Python环境变量和Python命令行选项，但要获得更多细节参考其他文档资源。

Python环境变量

环境变量（有些人称为shell变量或DOS变量）存在于Python之外，可用于给定的计算机上定制解释器每次运行时的行为。Python识别一些环境变量的设置，但只有少数是常用的，值得在这里进行说明。表A-1是Python相关的主要环境变量的设置。

表A-1：重要环境变量

变量	角色
PATH（或path）	系统shell的搜索路径（查找“python”）
PYTHONPATH	Python模块的搜索路径（用来导入）
PYTHONSTARTUP	Python交互模式启动文件的路径
TCL_LIBRARY、TK_LIBRARY	GUI扩展包的变量（tkinter）

这些变量使用起来都很直接，这里有一些建议。

PATH

PATH设置列出一组目录，这些目录是操作系统用来搜索可执行程序的。一般来说，应该包含Python解释器所在的目录（UNIX上的python程序或Windows上的python.exe）。如果你打算在Python所在目录下工作，或者在命令行输入完整的Python路径，就不需要设置这个变量。例如，在Windows中，如果你在运行任何代码前，都要执行cd C:\Python30（来到Python所在目录），或者总是输入C:\Python30\python（给出完整路径）而不只是python。此外，PATH设置多半是和命令行启动程序有关的，通过图标点击和IDE启动时，通常就没有什么关系了。

PYTHONPATH

PYTHONPATH设置的角色类似于PATH：当你在程序中导入模块文件时，Python解释器会参考PYTHONPATH变量，找出模块文件的位置。使用时，这个变量会设置成一个平台特定的目录名的列表。在UNIX头是以冒号分隔，而Windows上则是以分号间

隔。在通常情况下，这份清单只包含了你自己的源代码目录。其内容合并到了`sys.path`模块导入搜索路径中，以及脚本的目录、任何路径文件设置以及标准库目录。

除非你要执行跨目录的导入，否则不用设置这个变量，因为Python会自动搜索程序顶层文件的主目录，只有当模块需要导入存在于不同目录的另一个模块时，才需要这个设置。参见本附录稍后对于`.pth`路径文件的介绍，它作为`PYTHONPATH`的一个替代方案。对于模块搜索路径的更多介绍，请参阅第21章。

PYTHONSTARTUP

如果`PYTHONSTARTUP`设为Python程序代码的路径名，每当启动交互模式解释器时，Python就会自动执行这个文件的代码，好像是在交互模式命令行中输入它一样。这很少使用，但是当通过交互模式工作时，要确保一定会加载某些工具，这样很方便，可以省去导入。

tkinter设置

如果想使用tkinter GUI工具集（在Python 2.6中叫Tkinter），可能要把表A-1的两个GUI变量，设成Tcl和Tk系统的源代码库的目录名（很像`PYTHONPATH`）。然而，这些设置在Windows系统上并不需要（tkinter会随Python一起安装），如果Tcl和Tk位于标准目录中，通常也是不需要的。

注意，因为这些环境设置（以及`.pth`文件）都位于Python外部，所以什么时候设置它们通常是无所谓。你可以在Python安装之前或之后设置，只要在Python实际运行前按照你的需要设置过就可以了。

获得Linux上tkinter（和IDLE）GUI的支持

第2章所提到的IDLE接口是Python tkinter GUI程序。tkinter是GUI工具集，而且是Windows和其他平台上Python的标准组件。不过，在某些Linux系统上，底层GUI库可能不是标准的安装组件。要在Linux上让Python新增GUI功能，可以试着运行**`yumt kinter`**命令来自动安装tkinter底层链接库。这样应该适用于具有yum安装程序的Linux发行版上（以及一些其他的系统）。

如何设定配置选项

设置Python相关环境变量的方式以及该设置成什么，取决于你所使用计算机的类型。同样要记住，不用马上把它们全部都设置好。尤其是，如果你使用的是IDLE（第3章所述），并不需要事先配置。

但是，假设你在机器上的`utilities`和`package1`目录中有一些有用的模块文件，而你想从其中

他目录中的文件导入这些模块。也就是说，要从`utilities`目录加载名为`spam.py`的文件，则要能够在计算机上其他位置的另一个文件中这么写：

```
import spam
```

为了让它能够工作，你得配置模块搜索路径，以引入包含`spam.py`的目录。下面是这个过程中的一些技巧。

UNIX/Linux shell变量

在UNIX系统上，设置环境变量的方式取决于你使用的shell。在`csh` shell下，你可以在`.cshrc`或`.login`文件中增加下面的行，来设置Python模块的搜索路径：

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

这是告诉Python，在两个用户定义的目录中寻找要导入的模块。但是，如果你使用`ksh` shell，此设置会出现在`.kshrc`文件内，看起来就像这样：

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

其他shell可能使用不同（但类似）的语法。

DOS变量（Windows）

如果你在使用MS-DOS，或旧版Windows，可能需要在`C:\autoexec.bat`文件中新增一个环境变量配置命令，重启电脑，让修改生效。这类机器上的配置命令有DOS独特的语法：

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

你也可以在DOS终端窗口中输入类似的命令，这样的设置只能在那个终端窗口中有效。修改`.bat`文件则可以永久的修改，对于所有的程序都有效。

其他Windows选项

在新的Windows中，可以通过系统环境变量GUI设置PYTHONPATH和其他变量，而不用编译文件或重启。在XP上，选择“控制面板”→“系统”→“高级”标签，然后单击“环境变量”按钮来编辑或新增变量（PYTHONPATH通常是用户的变量）。使用前面的DOS `set` 命令中给出的相同变量名和值语法。Vista上的过程是类似的，但是可能必须一路验证操作。

不需重新启动机器，不过如果Python开着，要记得重启它，从而让它也能使用你的修改（只在Python启动时才配置其路径）。如果在一个Windows命令提示符窗口中工作，可能需要重新启动并选择修改。

Windows注册表

如果你是有经验的Windows用户，也可以使用注册表编辑器来配置模块搜索路径。选择“开始”→“运行”，然后输入**regedit**。假设你的机器上有这个注册表工具，你就能浏览Python的项目，然后进行修改。不过，这是脆弱且易出错的方法，除非你非常熟悉注册表，不然建议使用其他方法（实际上，这类似于对你的计算机做脑手术，因此要慎重）。

路径文件

最后，如果你选择通过`.pth`文件扩展模块搜索路径，而不是使用PYTHONPATH变量，就可以改用编写文本文件，在Windows中，看起来就像这样（文件`C:\Python30\mypath.pth`）。

```
c:\pycode\utilities
d:\pycode\package1
```

其内容会随平台不同而各不相同，而它的容器目录也会随平台和Python版本而各不相同。Python在启动时会自动定位这个文件。

路径文件中的目录名，可以是绝对或相对于含有路径文件的目录。`.pth`文件可以有多个（所有目录都会加进来），而`.pth`文件可以出现在各种平台特定的以及版本特定的、自动检查的目录中。一般情况下，一个以Python N.M发布的Python版本，在Windows系统上在`C:\PythonNM`和`C:\PythonNM\Lib\site-packages`中查找路径文件，在UNIX和Linux上则在`/usr/local/lib/pythonN.M/site-packages`和`/usr/local/lib/site-python`中。关于使用路径文件配置`sys.path`导入搜索路径的更多介绍，参见第21章。

因为这些设置通常都是可选的，而且本书不是介绍操作系统shell的书，所以更多的细节请参考其他资源。参考系统shell的说明，或其他文档来了解更多的信息。此外，如果你不清楚你的设置应该是什么，可以询问系统管理员或本地的专家来获取帮助。

Python命令行选项

当我们从一个系统命令行启动Python的时候（即shell提示符），可以传入各种选项标志来控制Python如何运行。和系统范围的环境变量不同，每次运行脚本的时候，命令行选项可能不同。Python 3.0中的一个Python命令行调用的完整形式如下所示（Python 2.6中大致相同，只是一些选项不同）：

```
python [-bBdEhiOsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

大多数命令行只是使用这个形式的`script`和`args`部分，来运行程序的源文件，并带有供

程序自身使用的参数。为了说明这点，考虑脚本文件`main.py`，它打印出作为`sys.argv`可供脚本使用的命令行参数列表：

```
# File main.py
import sys
print(sys.argv)
```

在下面的命令行中，`python`和`main.py`都可以是完整的目录路径，并且3个参数(`a b -c`)用于出现在`sys.argv`列表中的脚本。`sys.argv`中的第一项总是脚本文件的名称：

```
c:\Python30> python main.py a b -c                # Most common: run a script file
['main.py', 'a', 'b', '-c']
```

其他代码格式化规范选项允许我们指定Python代码：在命令行自身上运行（`-c`），接受代码以从标准输入流运行（一个-意味着从一个管道或重定向输入文件读取），等等：

```
c:\Python30> python -c "print(2 ** 100)"           # Read code from command argument
1267650600228229401496703205376

c:\Python30> python -c "import main"               # Import a file to run its code
['-c']

c:\Python30> python - < main.py a b -c             # Read code from standard input
['-', 'a', 'b', '-c']

c:\Python30> python - a b -c < main.py             # Same effect as prior line
['-', 'a', 'b', '-c']
```

`-m`代码规范在Python的模块查找路径（`sys.path`）上定位一个模块，并且将其作为顶级脚本运行（作为模块`__main__`）。在这里省略了“.py”后缀，因为文件名是一个模块：

```
c:\Python30> python -m main a b -c                # Locate/run module as script
['c:\Python30\main.py', 'a', 'b', '-c']
```

`-m`选项还支持使用相对导入语法来运行包中的模块，以及位于`.zip`包中的模块。这个开关通常用来运行`pdb`调试器，并且针对一个脚本调用而不是交互来从一个命令行配置`profiler`模块，尽管这种用法在Python 3.0中有了一些变化（配置似乎受到了Python 3.0中移除`execfile`的影响，并且`pdb`在新的Python 3.0 `io`模块中划入了冗余输入/输出代码）：

```
c:\Python30> python -m pdb main.py a b -c          # Debug a script
--Return--
> c:\python30\lib\io.py(762)closed()->False
-> return self.raw.closed
(Pdb) c

c:\Python30> C:\python26\python -m pdb main.py a b -c  # Better in 2.6?
> c:\python30\main.py(1)<module>()
-> import sys
```

```
(Pdb) c
```

```
c:\Python30> python -m profile main.py a b -c           # Profile a script
```

```
c:\Python30> python -m cProfile main.py a b -c          # Low-overhead profiler
```

紧跟在“python”之后和计划要运行的代码之前，Python接受了控制器自身行为的额外参数。这些参数由Python自身使用，并且对于将要运行的脚本没有意义。例如，-O以优化模式运行Python，-u强制标准流为unbuffered，而-i在运行一段脚本后进入交互模式：

```
c:\Python30> python -u main.py a b -c                  # Unbuffered output streams
```

Python 2.6还支持额外的选项以提升对Python 3.0的兼容性（-3，-0），并且检测制表符缩进用法的 inconsistency，而这在Python 3.0中总是会检测并报告的（-t；参见第12章）。参见Python的手册或参考资料，以了解可用的命令行选项的具体细节。或者更好的做法是，问Python自己，即运行如下的命令行：

```
c:\Python30> python -?
```

以请求Python的帮助显示，它给出了可用的命令行选项。如果要处理复杂的命令行，应确保还查看标准库模块getopt和optparse，它们支持更加复杂的命令行处理。

寻求更多帮助

Python的标准手册集如今包含了针对各种平台上用法的有价值提示。在安装了Python之后，在Windows下通过“开始”按钮可以访问标准手册集，通过<http://www.python.org>也可以在线访问。找到手册集中标题为“Using Python”的顶级部分，以了解更加特定于平台的介绍和提示，以及最新的跨平台环境和命令行细节。

和往常一样，Web也是我们的朋友，尤其是在一个快速变化的领域，其变化速度比图书的更新快多了。由于Python广为采用，所以通过Web搜索可以找到关于Python使用问题的任何解答，这样的机会很大。

各部分练习题的解答

第一部分 使用入门

参考第3章“第一部分 练习题”中的习题。

1. 交互。假设Python已正确配置，交互模式看起来应该就像这样（可以在IDLE或shell提示符下运行）。

```
% python
...copyright information lines...
>>> "Hello World!"
'Hello World!'
>>> # Use Ctrl-D or Ctrl-Z to exit, or close window
```

2. 程序。你的程序代码（即模块）文件`module1.py`和操作系统shell的交互看起来应该像这样：

```
print('Hello module world!')

% python module1.py
Hello module world!
```

同样，你也可以用其他方式运行：单击文件图标、使用IDLE的Run/Run Module菜单选项等。

3. 模块。下面的交互说明了如何导入模块文件从而运行一个模块。

```
% python
>>> import module1
Hello module world!
>>>
```


要记住，不停止和重启解释器时，需要重载模块才能再次运行它。把文件移到不同目录并导入它，是很有技巧性的问题：如果Python在最初的目录中产生`module1.pyc`文件，即使源代码文件（.py）已被移到不在Python搜索路径中的目录，导入该模块时，Python依然会使用这个pyc文件。如果Python可读取源代码文件的目录，就会自动写.pyc文件，.pyc文件包含模块编译后的字节码的版本。参考第3章有关模块的内容。

4. 脚本。假设你的平台支持#!技巧，你的解法看起来应该像这样（虽然你的#!行可能需要列出机器上的另一路径）：

```
#!/usr/local/bin/python          (or #!/usr/bin/env python)
print('Hello module world!')
% chmod +x module1.py

% module1.py
Hello module world!
```

5. 错误。下面的交互模式（在Python 3.0下运行）示范了当你完成此练习题时会碰到的出错消息的种类。其实，你触发的是Python异常；默认异常处理行为会终止正在运行的Python程序，然后在屏幕上打印出错消息和堆栈的跟踪信息。堆栈的跟踪信息显示当异常发生时，程序所处在的位置。在第七部分中，你会学到，可以使用try语句捕捉它，并进行任意的处理。你也会看到Python包含成熟的源代码调试器，从而可以满足特殊的错误检测的需求。就目前而言，程序错误发生时，Python会提供有意义的消息而不是默默地就崩溃了：

```
% python
>>> 2 ** 500
32733906078961418700131896968275991522166420460430647894832913680961337964046745
54883270092325904157150886684127560071009217256545885393053328527589376
>>>
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

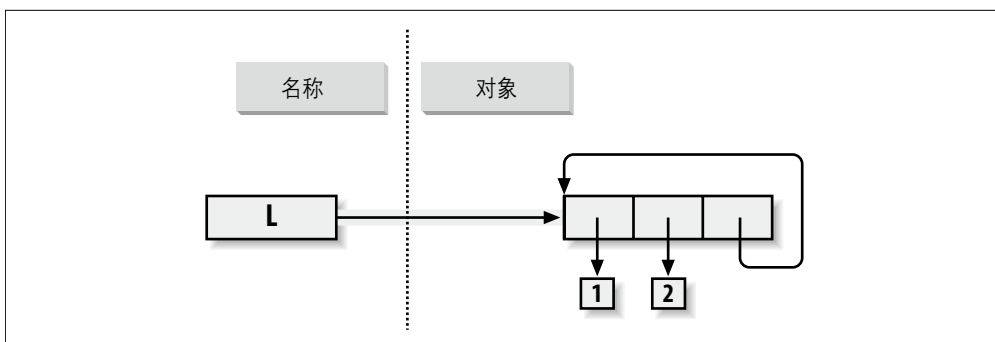
6. 中断和循环。当你输入以下代码的时候：

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

会在Python中创建循环数据结构。在1.5.1版以前，Python打印不够智能，无法检测对象中的循环，而且会打印无止境的[1, 2, [1, 2, [1, 2, [1, 2, 流，直到你按

下机器上的中断组合键（从技术上来讲，就是引发键盘中断异常，并打印默认消息）。从Python 1.5.1起，打印已经足够智能，可以检测循环，并打印[[...]]，而不是让你知道它已经在对象结构中检测到一个循环并避免永远打印。

循环的原因很微妙，而且需要第二部分的信息。但是，简而言之，Python中的赋值语句一定会产生对象的引用值，而不是它们的拷贝。你可以把对象看作是一块内存，把引用看作是隐式指向的指针。当你执行上面的第一个赋值语句时，名称L变成了对两个元素的列表对象的引用，也就是指向一段内存的指针。Python列表其实是对对象引用值的数组，有一个append方法会通过末尾添加另一个对象的引用，对数组进行原处修改。在这里，append调用会把L前面的引用加在L末尾，从而造成图B-1所示的循环：列表末尾的一个指针指回到列表的前面。



图B-1：循环对象，通过把列表附加在自身而生成。在默认情况下，Python是附加最初的列表的引用值，而不是列表的拷贝

除了特殊的打印，正如我们在第6章中学习的，循环对象还必须由Python的垃圾收集器特殊处理，否则，当它们不再使用的时候，其空间将保持未回收。尽管这种情况在实际中很少见，但在一些遍历任意对象或结构的程序中，你必须通过记录已经遍历到哪里，从而检测这样的循环，以避免陷入循环。不管你相信与否，循环数据结构偶尔也会很有用的（但不包括打印的时候）。

第二部分 类型和运算

参考第9章“第二部分 练习题”中的习题。

1. 基础。以下是你应该得到的各种结果，还有其含义的注释。其中一些使用分号“;”把一个以上的语句挤在一行中（这里的“;”是语句分隔符），逗号构建了在圆括号中显示的元组。还要记住，靠近顶部的/除法结果在Python 2.6和Python 3.0中有所不同（参见第5章了解更多细节），并且，list包装字典方法调用以显示结果，这在Python 3.0中是必需的，但在Python 2.6中不是（参见第8章）。

Numbers

```
>>> 2 ** 16                                     # 2 raised to the power 16
65536
>>> 2 / 5, 2 / 5.0                             # Integer / truncates in 2.6, but not 3.0
(0.40000000000000002, 0.40000000000000002)
```

Strings

```
>>> "spam" + "eggs"                             # Concatenation
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5                                         # Repetition
'hamhamhamhamham'
>>> S[:0]                                         # An empty slice at the front -- [0:0]
''                                                # Empty of same type as object sliced

>>> "green %s and %s" % ("eggs", S)             # Formatting
'green eggs and ham'
>>> 'green {0} and {1}'.format('eggs', S)
'green eggs and ham'
```

Tuples

```
>>> ('x',)[0]                                    # Indexing a single-item tuple
'x'
>>> ('x', 'y')[1]                                # Indexing a 2-item tuple
'y'
```

Lists

```
>>> L = [1,2,3] + [4,5,6]                       # List operations
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]
>>> [L[2], L[3]]                                # Fetch from offsets; store in a list
[3, 4]
>>> L.reverse(); L                               # Method: reverse list in-place
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                                  # Method: sort list in-place
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                                   # Method: offset of first 4 (search)
3
```

Dictionaries

```
>>> {'a':1, 'b':2}['b']                         # Index a dictionary by key
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                                   # Create a new entry
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                              # A tuple used as a key (immutable)
```

```
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}

>>> list(D.keys()), list(D.values()), (1,2,3) in D          # Methods, key test
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], True)

# Empties

>>> [[]], ["",[],(),{}],None                               # Lots of nothings: empty objects
([[]], [' ', [], (), {}], None)
```

2. 索引运算和分片运算。超出边界的索引运算（例如，L[4]）会引发错误。Python一定会检查，以确保所有偏移值都在序列边界内。

另外，分片运算超出边界（例如，L[-1000:100]）可工作，因为Python会缩放超出边界的分片以使其合用（必要时，限制值可设为零和序列长度）。

以翻转的方式提取序列是行不通的（较低边界值比较高边界值更大，例如，L[3:1]）。你会得到空分片（[]），因为Python会缩放分片限制值，以确定较低边界永远比较高边界小或相等（例如，L[3:1]会缩放成L[3:3]，空的插入点是在偏移值3处）。Python分片一定是从左至右抽取，即使你用负号索引值也是这样（会先加上序列长度转换成正值）。注意，Python 2.3的第三限制值分片会稍微修改此行为，例如L[3:1:-1]的确是从右至左抽取。

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. 索引运算、分片运算以及del。你和解释器的交互看起来应该像下列程序代码。注意把空列表赋值给一个偏移值，会将空列表对象保存在这里，不过赋值空列表给一个分片，则会删除该分片。分片赋值运算期待得到的是另一个序列，否则你就会得到类型错误。这是把元素插入赋值之序列之内，而非序列本身：

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
```

```

[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation

```

4. **元组赋值运算**。交换X和Y的值。当元组出现在赋值符号(=)左右两边时，Python会根据左右两侧对象的位置，把右侧对象赋值给左边的目标。注意，左边的那些目标其实并非真正的元组（虽然看起来很像），可能最容易理解。那些只是一组独立的赋值目标。右侧的元素则是元组，也就是会在赋值运算进行时分解（元组提供所需要的临时赋值运算从而达到交换的效果）：

```

>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'

```

5. **字典键**。任何不可变对象都可作为字典的键，包括整数、元组和字符串等。这其实是字典，即使有些键看起来像整数偏移值。混合类型的键也能够正常工作：

```

>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}

```

6. **字典索引运算**。对不存在的键进行索引运算（D['d']）会引发错误。对不存在的键做赋值运算（D['d']='spam'），则会创建新的字典元素。另一方面，列表超边界索引运算也会引发错误，超边界赋值运算也是。变量名称就像字典键那样，在引用时，必须已做了赋值。在首次赋值时，就会创建它。实际上，变量名能作为字典键来处理 [在模块命名空间或堆栈框架字典（stack-frame dictionary）中都是可见的]：

```

>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?

```

```

KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0, 1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range

```

7. 通用运算。问题解答：

- +运算符无法用于不同/混合类型（例如，字符串+列表，列表+元组）。
- +不适用于字典，因为那不是序列。
- append方法只适用于列表，不适用于字符串，而键只适用于字典。append假设其目标是可变的，因为这是一个原地的扩展，字符串是不可变的。
- 分片和合并运算一定会在对象处理后传回相同类型的新对象：

```

>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> list({}.keys())                                     # list needed in 3.0, not 2.6
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> [][::]
[]
>>> ""[::]
''

```

8. 字符串索引运算。因为字符串是单个字符的字符串的集合体，每次对字符串进行索引运算时，就会得到一个可再进行索引运算的字符串。`S[0][0][0][0][0]`就是一直对第一个字符做索引运算。这一般不适用于列表（列表可包含任意对象），除非列表包含了字符串：

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. 不可变类型。下列任意解答都行。索引赋值运算不行，因为字符串是不可变的：

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

（参见第36章中关于Python 3.0的`bytearray`字符串类型的介绍——它是小整数的一个可变的序列，基本上可与字符串一样处理。）

10. 嵌套。以下为例子：

```
>>> me = {'name':('John', 'Q', 'Doe'), 'age': '?', 'job': 'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'Doe'
```

11. 文件。下面是在Python中创建和读取文本文件的方法（`ls`是UNIX命令，在Windows中则使用`dir`）：

```
# File: maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')
file.close()

# Or: open().write()
# close not always needed

# File: reader.py
file = open('myfile.txt')
print(file.read())

# 'r' is default open mode
# Or print(open()).read())

% python maker.py
% python reader.py
Hello file world!
% ls -l myfile.txt
-rwxrwxrwa  1 0      0 19 Apr 13 16:33 myfile.txt
```

第三部分 语句和语法

参考第15章“第三部分 练习题”中的习题。

1. 编写基本循环。当你做这个练习题时，最后的代码会像这样：

```
>>> S = 'spam'
>>> for c in S:
...     print(ord(c))
...
115
112
97
109

>>> x = 0
>>> for c in S: x += ord(c)           # Or: x = x + ord(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> list(map(ord, S))                 # list() required in 3.0, not 2.6
[115, 112, 97, 109]
```

2. 反斜线字符。这个例子会打印铃声字符（\a）50次。假设你的机器能处理，而且是在IDLE外运行，你就会听到一系列哔哔声（或者如果你的机器够快的话，就是一长声）。
3. 排序字典。下面是做这个练习题的一种方式（如果看不懂的话，就参考第8章或第14章）。记住，你确实是应该把keys和sort调用像这样分开，因为sort会返回None。在Python 2.2和后续版本中，你可以直接迭代字典的键，而不需要调用keys（例如，for key in D:），但是，键列表无法像这段代码那样排序。在新近Python版本中，你也可以使用内置函数sorted来达到相同的效果：

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())             # list() required in 3.0, not in 2.6
>>> keys.sort()
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
```



```

c => 3
d => 4
e => 5
f => 6
g => 7

>>> for key in sorted(D):
...     print(key, '=>', D[key])
# Better, in more recent Pythons

```

4. 程序逻辑替代方案。这里是一些解答的样本代码。对于步骤e，把 $2 ** X$ 的结果赋给步骤a和步骤b的循环外的一个变量，并且在循环内使用它。你的结果也许不同。这个练习题的设计目的，主要就是让你练习代码的替代方案，所以任何合理的结果都是满分：

```

# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')

# b

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
else:
    print(X, 'not found')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# d

X = 5
L = []
for i in range(7): L.append(2 ** i)

```

```

print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# f

X = 5
L = list(map(lambda x: 2**x, range(7)))          # or [2**x for x in range(7)]
print(L)                                         # list() to print all in 3.0, not 2.6

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

```

第四部分 函数

参考第20章“第四部分 练习题”的习题。

1. 基础。这题没什么，但是要注意，使用`print`（以及你的函数），从技术上来讲就是多态运算，也就是为每种类型的对象做正确的事：

```

% python
>>> def func(x): print(x)
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}

```

2. 参数。下面是示范的解答。记住，你得使用`print`才能查看测试调用的结果，因为文件和交互模式下输入的代码并不相同。一般而言，Python不会回显文件中表达式语句的结果：

```

def adder(x, y):
    return x + y

print(adder(2, 3))
print(adder('spam', 'eggs'))
print(adder(['a', 'b'], ['c', 'd']))

% python mod.py
5
spameggs
['a', 'b', 'c', 'd']

```

3. 可变参数。在下面的`adders.py`文件中，有两个版本的`adder`函数。这里的难点在于，了解如何把累加器初始值设置为任何传入类型的空值。第一种解法是使用手动类型测试，从而找出整数，以及如果参数不是整数时，第一参数（假设为序列）的空分片。第二个解法是用第一个参数设定初始值，之后扫描第二元素和之后的元素，很像第18章中的各种`min`函数版本。

第二个解法更好。这两种解法都假设所有参数为相同的类型，而且都无法用于字典（正如第二部分所看到的，`+`无法用在混合类型或字典上）。你也可以加上类型检测和特殊代码从而兼容字典，但那是额外的加分项了。

```
def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0):
        sum = 0
    else:
        sum = args[0][:0]
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]
    for next in args[1:]:
        sum += next
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('spam', 'eggs', 'toast'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))

% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']
```

4. 关键字参数。下面是我对这个练习题第一部分的解答（文件`mod.py`）。要遍历关键词参数时，在函数开头列使用`** args`形式，并且使用循环 [例如，`for x in args.keys(): use args[x]`]，或者使用`args.values()`，使其等同于计算`*args`位置参数的和：

```
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print(adder())
print(adder(5))
print(adder(5, 6))
```

```

print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))

% python mod.py
6
10
14
18
18

# Second part solutions

def adder1(*args):                                # Sum any number of positional args
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):                               # Sum any number of keyword args
    argskeys = list(args.keys())                 # list needed in 3.0!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):                               # Same, but convert to list of values
    args = list(args.values())                   # list needed to index in 3.0!
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                               # Same, but reuse positional version
    return adder1(*args.values())

print(adder1(1, 2, 3),      adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))

```

5. (和6.) 下面是对练习题5和6的解答(文件`dicts.py`)。不过, 这些只是编写代码的练习, 因为Python 1.5新增了字典方法`D.copy()`和`D1.update(D2)`来处理字典的复制和更新(合并)等情况(参考Python的链接库手册或者O'Reilly的《Python Pocket Reference》以获得更多细节)。`X[:]`不适用于字典, 因为字典不是序列(参考第8章的细节)。此外, 记住, 如果你是做赋值(`e = d`), 而不是复制, 将产生共享字典对象的引用值, 修改`d`也会跟着修改`e`:

```

def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):

```

```

    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

% python
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}

```

6. 参见5。

7. 其他参数匹配的例子。下面是你应该得到的交互模式下的结果，还有注释说明了其匹配情况：

```

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

% python
>>> f1(1, 2)                             # Matched by position (order matters)
1 2
>>> f1(b=2, a=1)                         # Matched by name (order doesn't matter)
1 2

>>> f2(1, 2, 3)                          # Extra positionals collected in a tuple
1 (2, 3)

>>> f3(1, x=2, y=3)                      # Extra keywords collected in a dictionary
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)                 # Extra of both kinds
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                                # Both defaults kick in
1 2 3

```

```

>>> f5(1, 4)                                # Only one default used
1 4 3

>>> f6(1)                                    # One argument: matches "a"
1 2 ()

>>> f6(1, 3, 4)                              # Extra positional collected
1 3 (4,)

```

8. 再谈质数。下面是质数的实例，封装在函数和模块中（文件`primes.py`），可以多次运行。增加了一个`if`测试，从而考虑了负数、0以及1。把`/`改成`//`，从而使这个解答不会受到第5章提到的Python 3.0的/真除法改变的困扰，并且使其支持浮点数。（把`from`语句的注释去掉，把`//`改成`/`，看看在Python 2.6中的不同）：

```

#from __future__ import division

def prime(y):
    if y <= 1:                                # For some y > 1
        print(y, 'not prime')
    else:
        x = y // 2                             # 3.0 / fails
        while x > 1:
            if y % x == 0:                     # No remainder?
                print(y, 'has factor', x)
                break                           # Skip else
            x -= 1
        else:
            print(y, 'is prime')

prime(13); prime(13.0)
prime(15); prime(15.0)
prime(3); prime(2)
prime(1); prime(-3)

```

下面是这个模块的运行。即使可能不该这样，但`//`运算符也适用于浮点数：

```

% python primes.py
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 not prime
-3 not prime

```

这个函数没有太好的可重用性，但可以改为返回值，而不是打印，不过作为实验已经足够。这也不是严格的数学质数（浮点数也行），而且依然没有效率。改进的事就留给数学考虑周密的读者作为练习。（提示：通过`for`循环来运行`range(y, 1, -1)`，可能会比`while`快一些，真正的瓶颈在于算法。）要测试替代方案的时间，

可以使用内置的`time`模块以及下面这个通用的函数调用`timer`中所用到的编写代码的模式（参考库手册以获得更多细节）：

```
def timer(reps, func, *args):
    import time
    start = time.clock()
    for i in range(reps):
        func(*args)
    return time.clock() - start
```

9. 列表解析。下面是你应该写出来的代码的样子。其中有我自己的偏好，不要求都照着做：

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. 计时工具。下面是我编写来对3个平方根选项计时的代码，带有在Python 2.6和Python 3.0中的结果。每个函数最后的结果打印出来，以验证所有3个方案都做同样的工作：

```
# File mytimer.py (2.6 and 3.0)
...same as listed in Chapter 20...

# File timesqrt.py

import sys, mytimer
reps = 10000
repslist = range(reps)                                # Pull out range list time for 2.6

from math import sqrt                                  # Not math.sqrt: adds attr fetch time
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
```

```

    for i in repslist:
        res = i ** .5
    return res

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (mathMod, powCall, powExpr):
        elapsed, result = tester(test)
        print ('-'*35)
        print ('%s: %.5f => %s' %
                (test.__name__, elapsed, result))

```

如下是针对Python 3.0和Python 2.6的测试结果。对这两者而言，看上去math模块比**表达式更快，**表达式比pow调用更快。然而，应该在你自己的机器上以及Python版本中尝试一下。此外要注意，对于这一测试，Python 3.0几乎比Python 2.6慢两倍；Python 3.1或以后的版本可能表现更好些（将来进行测试自己看看结果）：

```

c:\misc> c:\python30\python timesqrt.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----
mathMod: 5.33906 => 99.994999875
-----
powCall: 7.29689 => 99.994999875
-----
powExpr: 5.95770 => 99.994999875
<best>
-----
mathMod: 0.00497 => 99.994999875
-----
powCall: 0.00671 => 99.994999875
-----
powExpr: 0.00540 => 99.994999875

c:\misc> c:\python26\python timesqrt.py
2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)]
<timer>
-----
mathMod: 2.61226 => 99.994999875
-----
powCall: 4.33705 => 99.994999875
-----
powExpr: 3.12502 => 99.994999875
<best>
-----
mathMod: 0.00236 => 99.994999875
-----
powCall: 0.00402 => 99.994999875
-----
powExpr: 0.00287 => 99.994999875

```


要计时Python 3.0字典解析和对等的for循环交互的相对速度，应运行如下的一个会话。事实表明，这两者在Python 3.0下大致是相同的；然而，和列表解析不同，手动循环如今比字典解析略快（尽管差异并不大，当我们生成50个字典每个字典1 000 000项的时候，会节省半秒钟）。再次说明，你应该自己进一步调查，在自己的计算机和Python中测试，而不是把这些结果作为标准：

```
c:\misc> c:\python30\python
>>>
>>> def dictcomp(I):
...     return {i: i for i in range(I)}
...
>>> def dictloop(I):
...     new = {}
...     for i in range(I): new[i] = i
...     return new
...
>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from mytimer import best, timer
>>> best(dictcomp, 10000)[0]                # 10,000-item dict
0.0013519874732672577
>>> best(dictloop, 10000)[0]
0.001132965223233029
>>>
>>> best(dictcomp, 100000)[0]                # 100,000 items: 10 times slower
0.01816089754424155
>>> best(dictloop, 100000)[0]
0.01643484018219965
>>>
>>> best(dictcomp, 1000000)[0]               # 1,000,000 items: 10X time
0.18685105229855026
>>> best(dictloop, 1000000)[0]               # Time for making one dict
0.1769041177020938
>>>
>>> timer(dictcomp, 1000000, _reps=50)[0]    # 1,000,000-item dict
10.692516087938543
>>> timer(dictloop, 1000000, _reps=50)[0]    # Time for making 50
10.197276050447755
```

第五部分 模块

参考第24章“第五部分 练习题”的习题。

1. 导入基础。这一道题比你想象的更简单。做完后，文件（*mymod.py*）和交互的结果看起来如下所示。记住，Python可以把整个文件读成字符串列表，而len内置函数可返回字符串和列表的长度：

```

def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):
    return countLines(name), countChars(name)      # Or pass file object
                                                    # Or return a dictionary

% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)

```

这些函数一次把整个文件加载到了内存中，当文件过大以至于机器的内存无法容纳时，就不能用了。为了更健壮一些，你可以改用迭代器逐行读取，在此过程中进行计数：

```

def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot

```

在UNIX上，你可以使用`wc`命令确认输出。在Windows中，对文件单击鼠标右键，来查看其属性。但是请注意，你的脚本报告的字符数可能会比Windows的少：为了可移植，Python把Windows `\r\n`行尾标识符转换成了`\n`，每行会少一个字节（字符）。为了和Windows的字节计数相同，你得使用二进制模式打开文件（`'rb'`），或者根据行数，加上对应的字节数。

顺便提一下，要做这道练习题中的“志向远大”的部分（传入文件对象，只打开文件一次），你可能需要使用内置文件对象的`seek`方法。本书没有提到，但其工作起来就像C的`fseek`调用（也是调用`seek`）：`seek`会把文件当前位置重设为传入的偏移值。`seek`运行后，未来的输入/输出运算就是相对于新位置而开始的。想要回滚到文件开头位置而又不关闭文件并重新打开，就需要调用`file.seek(0)`。文件的`read`方法会从文件当前位置开始读起，你得回滚到开头重新读取文件。下面是调整后的程序：

```

def countLines(file):
    file.seek(0)                                # Rewind to start of file
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                # Ditto (rewind if needed)

```

```

        return len(file.read())

def test(name):
    file = open(name)
    return countLines(file), countChars(file)    # Pass file object
                                                # Open file only once

>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)

```

2. `from`/`from *`。这里是`from *`部分；把`*`换成`countChars`就是其余的答案：

```

% python
>>> from mymod import *
>>> countChars("mymod.py")
291

```

3. `__main__`。如果你写得正确的话，哪种模式都能使用（运行程序或模块导入）：

```

def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):
    return countLines(name), countChars(name)    # Or pass file object
                                                # Or return a dictionary

if __name__ == '__main__':
    print(test('mymod.py'))

% python mymod.py
(13, 346)

```

在这里可能应该开始考虑使用命令行参数或用户输入来提供要统计的文件名，而不是在脚本中硬编码它（参见第24章了解关于`sys.argv`的更多内容，参见第10章了解关于输入的更多内容）：

```

if __name__ == '__main__':
    print(test(input('Enter file name:')))

if __name__ == '__main__':
    import sys
    print(test(sys.argv[1]))

```

4. 嵌套导入。下面是该题的解答（`myclient.py`文件）：

```

from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))

% python myclient.py
13 346

```

至于这个问题的其余部分，因为`from`只是在导入者中赋值变量名，所以`mymod`的

函数可以在myclient的顶层存取（可导入），就好像mymod的def是位于myclient中。例如，另一个文件可以写成：

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

如果myclient用的是import而不是from，就需要使用路径，通过myclient以获得mymod中的函数：

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

通常来说，你可以定义收集器模块，从其他模块导入所有的变量名，使得那些变量名能在单个方便的模块中使用。使用下面的代码，最后会有变量名somename的三个不同拷贝（mod1.somename、collector.somename以及__main__.somename）。这三个名称都共享相同的整数对象，而只有在交互模式提示符下存在着变量名somename：

```
# File mod1.py
somename = 42

# File collector.py
from mod1 import *
from mod2 import *
from mod3 import *

# Collect lots of names here
# from assigns to my names

>>> from collector import somename
```

5. 导入包。在这个练习题中，把练习题3的解答mymod.py文件放到一个目录包中。下面是我们所做的事：在Windows命令提示字符界面下，创建目录以及所需要的__init__.py文件。如果是其他的平台，你得进行修改（例如，使用mv和vi，而不是move和edit）。这些命令对于任意目录都适用（只是刚好在Python安装目录下运行命令），而你也可以从文件管理器GUI界面下完成其中的一些事。

当这样做以后，就有个mypkg子目录含有文件__init__.py和mymod.py。mypkg目录内需要有__init__.py，但其上层目录则不需要。mypkg位于模块搜索路径的主目录上。目录的初始设置文件中所写的print语句只会在导入时执行一次，不会有第二次：

```
C:\python30> mkdir mypkg
C:\Python30> move mymod.py mypkg\mymod.py
C:\Python30> edit mypkg\__init__.py
```

```

...coded a print statement...
C:\Python30> python
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg\mymod.py')
346

```

6. 重载。这道题只是要你实验一下修改本书的`changer.py`这个例子，所以这里没什么好写的。
7. 循环导入。简单来说，结果就是先导入`recur2`，因为递归导入是发生在`recur1`中的`import`，而不是`recur2`的`from`。

详细来讲是这样的：先导入`recur2`，这是因为从`recur1`到`recur2`的递归导入是整个取出`recur2`，而不是获取特定的变量名。从`recur1`导入时，`recur2`还不完整，因为其使用`import`而不是`from`，所以安全。Python会寻找并返回已创建的`recur2`模块对象，然后继续运行`recur1`剩余的部分，从而没有问题。当`recur2`的导入继续下去时，第二个`from`发现`recur1`（已完全执行）内的变量名`y`，所以不会报告错误。把文件当成脚本执行与将其当成模块导入并不相同。这些情况与通过交互模式先运行脚本中的第一个`import`或`from`相同。例如，将`recur1`作为脚本执行，与通过交互模式导入`recur2`一样，因为`recur2`是`recur1`中导入的第一个模块。

第六部分 类和OOP

参考第31章“第六部分 练习题”的习题。

1. 继承。下面是这个练习题的解答（`adder.py`文件），以及一些交互模式下的测试。
`__add__`重载方法只出现一次，就是在超类中，因为它调用了子类中类型特定的`add`方法：

```

class Adder:
    def add(self, x, y):
        print('not implemented!')
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(self.data, other)

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):

```

Or in subclasses?
Or return type?

```

        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new

% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands

```

在最后的测试中，得到表达式错误，因为+的右边出现类实例。如果你想修复它，可使用__radd__方法，就像第29章所描述的那样。

如果你在实例中保存一个值，可能也想重写add方法使其只带一个自变量（按照第六部分中其他例子的精神）：

```

class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(other)
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        pass

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)

```

Pass a single argument
The left side is in self

Change to use self.data instead of x

Prints [1, 2, 3, 4, 5, 6]

因为值是附加在对象上而不是到处传递，这个版本更加地面向对象。一旦你了解了，可能会发现，可以舍弃add，而在两个子类中定义类型特定的__add__方法。

2. 运算符重载。答案中（文件mylist.py）使用的一些运算符重载方法，书中没有多谈，但它们应该是很容易理解的。复制构造函数中的初始值很重要，因为它是可变的。你不会想修改或者拥有可能被类外其他地方共享的对象参照值。__getattr__方法把调用转给包装列表。有关以Python 2.2以及后续版本编写这个代码的更为容易方式的提示，可以参考第31章：

```
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]
        self.wrapped = []
        for x in start: self.wrapped.append(x)
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):
        return self.wrapped[offset]
    def __len__(self):
        return len(self.wrapped)
    def __getslice__(self, low, high):
        return MyList(self.wrapped[low:high])
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):
        return getattr(self.wrapped, name)
    def __repr__(self):
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('spam')
    print(x)
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x * 3)
    x.append('a')
    x.sort()
    for c in x: print(c, end=' ')

% python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s
```

要注意，通过附加而不是分片复制初值是很重要的，否则结果就不是真正的列表，也就不会响应预期的列表方法，例如，append（例如，对字符串进行分片运算会传

回另一字符串，而不是列表）。你可以通过分片运算复制MyList的初值，因为其类重载了分片运算，而且提供预期的列表接口。然而，你需要避免对对象（例如字符串）做分片式的复制。此外，集合已经是Python的内置类型，这大体上只是编写代码的练习而已（参考第5章有关集合的细节）。

3. 子类。解答如下所示（mysub.py）。你的答案应该也类似：

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                # Shared by instances

    def __init__(self, start):
        self.adds = 0                        # Varies in each instance
        MyList.__init__(self, start)

    def __add__(self, other):
        MyListSub.calls += 1                # Class-wide counter
        self.adds += 1                      # Per-instance counts
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds        # All adds, my adds

if __name__ == '__main__':
    x = MyListSub('spam')
    y = MyListSub('foo')
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x + ['toast'])
    print(y + ['bar'])
    print(x.stats())

% python mysub.py
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 'toast']
['f', 'o', 'o', 'bar']
(3, 2)
```

4. 元类方法。注意，在Python 2.6中，运算符尝试通过__getattr__取得属性。你需要返回一个值使其能够工作。警告：正如第30章所提到的，__getattr__不会在Python 3.0中针对内置操作而调用，因此，如下的表达式不会像介绍的那样工作；在Python 3.0中，像这样的类必须显式地重新定义__x__运算符重载方法。关于这一点的更多介绍，参见第30章、第37章和第38章：

```
>>> class Meta:
...     def __getattr__(self, name):
...         print('get', name)
...     def __setattr__(self, name, value):
```



```

...         print('set', name, value)
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function

>>> x[1:5]
get __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function

```

5. 集合对象。下面是应得到的交互模式下的结果。注释说明了调用的是哪个方法：

```

% python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])           # Runs __init__
>>> y = Set([3, 4, 5])

>>> x & y                           # __and__, intersect, then __repr__
Set:[3, 4]
>>> x | y                           # __or__, union, then __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")               # __init__ removes duplicates
>>> z[0], z[-1]                     # __getitem__
('h', 'o')

>>> for c in z: print(c, end=' ')   # __getitem__
...
h e l l o
>>> len(z), z                       # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])

>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])

```

对多个操作对象扩展的子类的解答，就像下面的类（*multiset.py*文件）一样。只需要取代最初集合中的两个方法。类的文档字符串说明了其工作原理：

```

from setwrapper import Set

```

```

class MultiSet(Set):
    """
    Inherits all Set names, but extends intersect
    and union to support multiple operands; note
    that "self" is still the first argument (stored
    in the *args argument now); also note that the
    inherited & and | operators call the new methods
    here with 2 arguments, but processing more than
    2 requires a method call, not an expression:
    """

    def intersect(self, *others):
        res = []
        for x in self:
            for other in others:
                if x not in other: break
            else:
                res.append(x)
        return Set(res)

    def union(*args):
        res = []
        for seq in args:
            for x in seq:
                if not x in res:
                    res.append(x)
        return Set(res)

```

与扩展的交互应该像下面所演示的。你可以使用&或调用intersect来做交集，但是对三个或以上的操作数，则必须调用intersect。&是二元（两边）运算符。此外，如果我们使用setwrapper.Set来引用multiset中的最初的类，那么也可以把MultiSet称为Set，让这样的改变变得更加透明。

```

>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])

>>> x & y, x | y
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])
Set:[2, 3]
>>> x.union(range(10))
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]

```

6. 类树链接。下面是修改Lister类的方法，并重新运行测试来显示其格式。对于基于dir的版本做同样的事情，并且当在树爬升变体中格式化类对象的时候也这么做：

```

class ListInstance:
    def __str__(self):
        return '<Instance of %s(%s), address %s:\n%s>' % (
            self.__class__.__name__,          # My class's name
            self.__supers(),                   # My class's own supers
            id(self),                          # My address
            self.__attrnames())               # name=value list

    def __attrnames(self):
        ...unchanged...

    def __supers(self):
        names = []
        for super in self.__class__.__bases__: # One level up from class
            names.append(super.__name__)       # name, not str(super)
        return ', '.join(names)

C:\misc> python testmixin.py
<Instance of Sub(Super, ListInstance), address 7841200:
    name data1=spam
    name data2=eggs
    name data3=42
>

```

7. 组合。解答如下 (*lunch.py*文件)，注释混在代码中。这可能是用Python描述问题比英文更简单的情况之一：

```

class Lunch:
    def __init__(self):                                # Make/embed Customer, Employee
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):                          # Start Customer order simulation
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                                   # Ask the Customer about its Food
        self.cust.printFood()

class Customer:
    def __init__(self):                                # Initialize my food to None
        self.food = None
    def placeOrder(self, foodName, employee):           # Place order with Employee
        self.food = employee.takeOrder(foodName)
    def printFood(self):                               # Print the name of my food
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):                     # Return Food, with desired name
        return Food(foodName)

class Food:
    def __init__(self, name):                          # Store food name
        self.name = name

if __name__ == '__main__':
    x = Lunch()                                        # Self-test code
    x.order('burritos')                                # If run, not imported
    x.result()
    x.order('pizza')
    x.result()

```

```
% python lunch.py
burritos
pizza
```

8. 动物园动物继承层次。下面是用Python编写的动物分类（*zoo.py*文件）。这是人工分法，这种通用化的编写代码模式适用于许多真实的结构，可以从GUI到员工数据库。Animal中引用的self.speak会触发独立的继承搜索，找到子类内的speak。通过交互模式测试这个练习题。试着用新类扩展这个层次，并在树中创建各种类的实例：

```
class Animal:
    def reply(self):    self.speak()           # Back to subclass
    def speak(self):   print('spam')         # Custom message

class Mammal(Animal):
    def speak(self):   print('huh?')

class Cat(Mammal):
    def speak(self):   print('meow')

class Dog(Mammal):
    def speak(self):   print('bark')

class Primate(Mammal):
    def speak(self):   print('Hello world!')

class Hacker(Primate): pass                  # Inherit from Primate
```

9. 描绘死鹦鹉。下面程序是我实现这道题的方法（*parrot.py*文件）。Actor超类的line方法的运作方式：读取self属性两次，让Python传回该实例两次。因此，会启动两次继承搜索（self.name和self.says()会在特定的子类内找到信息）：

```
class Actor:
    def line(self): print(self.name + ':', repr(self.says()))

class Customer(Actor):
    name = 'customer'
    def says(self): return "that's one ex-bird!"

class Clerk(Actor):
    name = 'clerk'
    def says(self): return "no it isn't..."

class Parrot(Actor):
    name = 'parrot'
    def says(self): return None

class Scene:
    def __init__(self):
        self.clerk = Clerk()                # Embed some instances
        self.customer = Customer()          # Scene is a composite
        self.subject = Parrot()
```

```
def action(self):
    self.customer.line()
    self.clerk.line()
    self.subject.line()
# Delegate to embedded
```

第七部分 异常和工具

参考第35章“第七部分 练习题”的习题。

1. `try/except`。本书的oops函数如下所示（*oops.py*文件）。对于不是编程的问题，修改oops来引发`KeyError`而不是`IndexError`，意味着try处理器不会捕捉这个异常（而是“传播”到顶层，并触发Python的默认出错消息）。变量名`KeyError`和`IndexError`来自于最外层内置作用域。导入**builtins**（在Python 2.6中是**__builtin__**），将其作为一个参数传给dir函数，亲自看看结果：

```
def oops():
    raise IndexError()

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    else:
        print('no error caught...')

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

2. 异常对象和列表。下面是扩展这个模块来增加自己的异常（一开始，这里用字符串）：

```
class MyError(Exception): pass

def oops():
    raise MyError('Spam!')

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    except MyError as data:
        print('caught error:', MyError, data)
    else:
        print('no error caught...')

if __name__ == '__main__':
    doomed()
```

```
% python oops.py
caught error: <class '__main__.MyError'> Spam!
```

就像所有类异常一样，实例变成了额外的数据。现在，出错信息会显示类(<...>)及其实例 (Spam!)。该实例必须从Python的Exception类继承一个__init__和一个__repr__或__str__；否则，它将像类一样打印。参阅第34章，详细了解这在内置异常类中如何工作。

3. 错误处理。下面是解这个练习题的方法 (safe2.py文件)。在文件中做测试，而不是在交互模式下进行，结果差不多相同：

```
import sys, traceback

def safe(entry, *args):
    try:
        entry(*args)
    except:
        traceback.print_exc()
        print('Got', sys.exc_info()[0], sys.exc_info()[1])

import oops
safe(oops.oops)

% python safe2.py
Traceback (innermost last):
  File "safe2.py", line 5, in safe
    entry(*args)
  File "oops.py", line 4, in oops
    raise MyError, 'world'
hello: world
Got hello world
```

4. 这里是一些供你研究的例子。要找更多例子的话，可以参考后续的书籍和网络：

```
# Find the largest Python source file in a single directory

import os, glob
dirname = r'C:\Python30\Lib'

allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])

# Find the largest Python source file in an entire directory tree

import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Python30\Lib'
```

```

else:
    dirname = '/usr/lib/python'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])

# Find the largest Python source file on the module import search path

import sys, os, pprint
visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])

# Sum columns in a text file separated by commas

filename = 'data.txt'
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

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

# Similar to prior, but using lists instead of dictionaries for sums

import sys

```

```

filename = sys.argv[1]
numcols = int(sys.argv[2])
totals = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]

print(totals)

# Test for regressions in the output of a set of scripts

import os
testscripts = [dict(script='test1.py', args=''),           # Or glob script/args dir
                dict(script='test2.py', args='spam')]

for testcase in testscripts:
    cmdline = '%(script)s %(args)s' % testcase
    output = os.popen(cmdline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
        else:
            print('Passed:', testcase['script'])

# Build GUI with tkinter (Tkinter in 2.6) with buttons that change color and grow

from tkinter import *                                # Use Tkinter in 2.6
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white', 'cyan', 'purple']

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack()
    L.config(fg=color)

def timer():
    L.config(fg=random.choice(colors))
    win.after(250, timer)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

win = Tk()

```



```

L = Label(win, text='Spam',
          font=('arial', fontsize, 'italic'), fg='yellow', bg='navy',
          relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='press', command=(lambda: reply('red'))).pack(side=BOTTOM, fill=X)
Button(win, text='timer', command=timer).pack(side=BOTTOM, fill=X)
Button(win, text='grow', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop()

```

Similar to prior, but use classes so each window has own state information

```

from tkinter import *
import random
class MyGui:
    """
    A GUI with buttons that change color and make the label grow
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Gui1', fg='white', bg='navy')
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Spam', command=self.reply).pack(side=LEFT)
        Button(parent, text='Grow', command=self.grow).pack(side=LEFT)
        Button(parent, text='Stop', command=self.stop).pack(side=LEFT)

    def reply(self):
        "change the button's color at random on Spam presses"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                       font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "start making the label grow on Grow presses"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "stop the button growing on Stop presses"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple'] # Customize to change color choices

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())

```

```

mainloop()

# Email inbox scanning and maintenance utility

"""
scan pop email box, fetching just headers, allowing
deletions without downloading the complete message
"""

import poplib, getpass, sys
mailserver = 'your pop email server name here'           # pop.rmi.net
mailuser = 'your pop email user name here'               # brian
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('There are', msgCount, 'mail messages, size ', mboxSize)
    msginfo = server.list()
    print(msginfo)
    for i in range(msgCount):
        msgnum = i+1
        msgsize = msginfo[1][i].split()[1]
        resp, hdrlines, octets = server.top(msgnum, 0)      # Get hdrs only
        print('-'*80)
        print('[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize))
        for line in hdrlines: print(line)

        if input('Print?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: print(line) # Get whole msg
        if input('Delete?') in ['y', 'Y']:
            print('deleting')
            server.dele(msgnum)                             # Delete on srvr
        else:
            print('skipping')
    finally:
        server.quit()                                       # Make sure we unlock mbox
        input('Bye.')                                       # Keep window up on Windows

# CGI server-side script to interact with a web browser

#!/usr/bin/python
import cgi
form = cgi.FieldStorage()                                # Parse form data
print("Content-type: text/html\n")                       # hdr plus blank line
print("<HTML>")
print("<title>Reply Page</title>")                       # HTML reply page
print("<BODY>")
if not 'user' in form:
    print("<h1>Who are you?</h1>")
else:
    print("<h1>Hello <i>%s</i>!</h1>" % cgi.escape(form['user'].value))

```

```

print("</BODY></HTML>")

# Database script to populate and query a MySQL database

from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='darling')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass # Did not exist

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev', 60000))
curs.execute('insert people values (%s, %s, %s)', ('Ann', 'mgr', 40000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Bob',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))

conn.commit() # Save inserted records

# Database script to populate a shelve with Python objects

# see also Chapter 27 shelve and Chapter 30 pickle examples

rec1 = {'name': {'first': 'Bob', 'last': 'Smith'},
        'job': ['dev', 'mgr'],
        'age': 40.5}

rec2 = {'name': {'first': 'Sue', 'last': 'Jones'},
        'job': ['mgr'],
        'age': 35.0}

import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close()

# Database script to print and update shelve created in prior script

```

```
import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])

bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close()
```