

TURING

图灵程序设计丛书



Python Crash Course

A Hands-On, Project-Based Introduction to Programming

Python编程 从入门到实践

【美】Eric Matthes 著 袁国忠 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

Eric Matthes

高中科学和数学老师，现居住在阿拉斯加，在当地讲授Python入门课程。他从5岁开始就一直在编写程序。

袁国忠

自由译者；2000年起专事翻译，主译图书，偶译新闻稿、软文；出版译著40余部，其中包括《C++ Prime Plus中文版》《CCNA学习指南》《CCNP ROUTE学习指南》《面向模式的软件架构：模式系统》《Android应用UI设计模式》《风投的选择：谁是下一个十亿美元级公司》等，总计700余万字；专事翻译前，从事过三年化工产品分析和开发，做过两年杂志和图书编辑。

TURING

图灵程序设计丛书

Python编程

从入门到实践

【美】Eric Matthes 著 袁国忠 译



Python Crash Course
A Hands-On, Project-Based Introduction to Programming

人民邮电出版社
北京

图灵社区会员 江子涛Tesla(jiangzitao201314@foxmail.com) 专享 尊重版权

图书在版编目 (C I P) 数据

Python编程：从入门到实践 / (美) 埃里克·马瑟斯 (Eric Matthes) 著；袁国忠译。-- 北京：人民邮电出版社，2016.7

(图灵程序设计丛书)

ISBN 978-7-115-42802-8

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

中国版本图书馆CIP数据核字(2016)第139461号

内 容 提 要

本书是一本针对所有层次的 Python 读者而作的 Python 入门书。全书分两部分：第一部分介绍用 Python 编程所必须了解的基本概念，包括 matplotlib、NumPy 和 Pygal 等强大的 Python 库和工具介绍，以及列表、字典、if 语句、类、文件与异常、代码测试等内容；第二部分将理论付诸实践，讲解如何开发三个项目，包括简单的 Python 2D 游戏开发，如何利用数据生成交互式的信息图，以及创建和定制简单的 Web 应用，并帮读者解决常见编程问题和困惑。

本书适合对 Python 感兴趣的任何层次的读者阅读。

-
- ◆ 著 [美] Eric Matthes
 - 译 袁国忠
 - 责任编辑 岳新欣
 - 执行编辑 杨琳 张曼
 - 责任印制 彭志环
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京 印刷
 - ◆ 开本：800×1000 1/16
 - 印张：29.75
 - 字数：703千字 2016年7月第1版
 - 印数：1-4 000册 2016年7月北京第1次印刷
 - 著作权合同登记号 图字：01-2016-1807号
-

定价：89.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版 权 声 明

Copyright © 2016 by Eric Matthes. *Python Crash Course : A Hands-On, Project-Based Introduction to Programming*, ISBN 978-1-59327-603-4, published by No Starch Press. Simplified Chinese-language edition copyright © 2016 by Posts and Telecom Press. All rights reserved.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

本书中文简体字版由No Starch Press授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

谨以此书献给我的父亲，以及儿子Ever。感谢父亲抽出时间来回答我提出的每个编程问题，而儿子Ever也开始向我提问了。

前　　言

如何学习编写第一个程序，每个程序员都有不同的故事。我还是个孩子时就开始学习编程了，当时我父亲在计算时代的先锋之一——数字设备公司（Digital Equipment Corporation）工作。我使用一台简陋的计算机编写了第一个程序，这台计算机是父亲在家里的地下室组装而成的，它没有机箱，裸露的主板与键盘相连，显示器是裸露的阴极射线管。我编写的这个程序是一款简单的猜数字游戏，其输出类似于下面这样：

```
I'm thinking of a number! Try to guess the number I'm thinking of: 25
Too low! Guess again: 50
Too high! Guess again: 42
That's it! Would you like to play again? (yes/no) no
Thanks for playing!
```

看到家人玩着我编写的游戏，而且它完全按我预期的方式运行，我心里不知有多满足。此情此景我永远都忘不了。

儿童时期的这种体验一直影响我至今。现在，每当我通过编写程序解决了一个问题时，心里都会感到非常满足。相比于孩提时期，我现在编写的软件满足了更大的需求，但通过编写程序获得的满足感几乎与从前一样。

读者对象

本书旨在让你尽快学会Python，以便能够编写能正确运行的程序——游戏、数据可视化和Web应用程序，同时掌握让你终身受益的基本编程知识。本书适合任何年龄的读者阅读，它不要求你有任何Python编程经验，甚至不要求你有编程经验。如果你想快速掌握基本的编程知识以便专注于开发感兴趣的项目，并想通过解决有意义的问题来检查你对新学概念的理解程度，那么本书就是为你编写的。本书还可供初中和高中教师用来通过开发项目向学生介绍编程。

本书内容

本书旨在让你成为优秀的程序员，具体地说，是优秀的Python程序员。通过阅读本书，你将迅速掌握编程概念，打下坚实的基础，并养成良好的习惯。阅读本书后，你就可以开始学习Python高级技术，并能够更轻松地掌握其他编程语言。

在本书的第一部分，你将学习编写Python程序时需要熟悉的基本编程概念，你刚接触几乎任何编程语言时都需要学习这些概念。你将学习各种数据以及在程序中将数据存储到列表和字典中的方式。你将学习如何创建数据集合以及如何高效地遍历这些集合。你将学习使用while和if语句来检查条件，并在条件满足时执行代码的一部分，而在条件不满足时执行代码的另一部分——这可为自动完成处理提供极大的帮助。

你将学习获取用户输入，让程序能够与用户交互，并在用户没停止输入时保持运行状态。你将探索如何编写函数来让程序的各个部分可重用，这样你编写执行特定任务的代码后，想使用它多少次都可以。然后，你将学习使用类来扩展这种概念以实现更复杂的行为，从而让非常简单的程序也能处理各种不同的情形。你将学习编写妥善处理常见错误的程序。学习这些基本概念后，你就能编写一些简短的程序来解决一些明确的问题。最后，你将向中级编程迈出第一步，学习如何为代码编写测试，以便在进一步改进程序时不用担心可能引入bug。第一部分介绍的知识让你能够开发更大、更复杂的项目。

在第二部分，你将利用在第一部分学到的知识来开发三个项目。你可以根据自己的情况，以最合适的顺序完成这些项目；你也可以选择只完成其中的某些项目。在第一个项目（第12~14章）中，你将创建一个类似于《太空入侵者》的射击游戏。这个游戏名为《外星人入侵》，它包含多个难度不断增加的等级。完成这个项目后，你就能够自己动手开发2D游戏了。

第二个项目（第15~17章）介绍数据可视化。数据科学家的目标是通过各种可视化技术来搞懂海量信息。你将使用通过代码生成的数据集、已经从网络下载下来的数据集以及程序自动下载的数据集。完成这个项目后，你将能够编写能对大型数据集进行筛选的程序，并以可视化方式将筛选出来的数据呈现出来。

在第三个项目（第18~20章）中，你将创建一个名为“学习笔记”的小型Web应用程序。这个项目能够让用户将学到的与特定主题相关的概念记录下来。你将能够分别记录不同的主题，还可以让其他人建立账户并开始记录自己的学习笔记。你还将学习如何部署这个项目，让任何人都能够通过网络访问它，而不管他身处何方。

为何使用 Python

继续使用Python，还是转而使用其他语言——也许是编程领域较新的语言？我每年都会考虑这个问题。可我依然专注于Python，其中的原因很多。Python是一种效率极高的语言：相比于众多其他的语言，使用Python编写时，程序包含的代码行更少。Python的语法也有助于创建整洁的代码：相比其他语言，使用Python编写的代码更容易阅读、调试和扩展。

大家将Python用于众多方面：编写游戏、创建Web应用程序、解决商业问题以及供各类有趣的公司开发内部工具。Python还在科学领域被大量用于学术研究和应用研究。

我依然使用Python的一个最重要的原因是，Python社区有形形色色充满激情的人。对程序员来说，社区非常重要，因为编程绝非孤独的修行。大多数程序员都需要向解决过类似问题的人寻求建议，经验最为丰富的程序员也不例外。需要有人帮助解决问题时，有一个联系紧密、互帮互

助的社区至关重要，而对于像你一样将Python作为第一门语言来学习的人而言，Python社区无疑是坚强的后盾。

Python是一门杰出的语言，值得你去学习，咱们现在就开始吧！

致 谢

要是没有No Starch Press出色的专业人士的帮助，本书根本不可能出版。Bill Pollock邀请我编写一本入门图书，因此这里要深深感谢他给予我这样的机会。Tyler Ortman在我编写本书的早期帮助我理清思路。Liz Chadwick和Leslie Shen详细阅读了每一章，并提出了宝贵的意见，而Anne Marie Walker让本书的很多地方都更清晰。Riley Hoffman回答了我就图书出版过程提出的每个问题，并且耐心地将我的作品变成了漂亮的图书。

感谢技术审稿人Kenneth Love。我与Kenneth相识于一次PyCon大会，他对Python和Python社区充满热情，一直是我获取专业灵感的源泉。Kenneth不仅检查了本书介绍的知识是否正确，还抱着让初学编程者对Python语言和编程有扎实认识的目的进行了审阅。即便如此，倘若书中有任何不准确的地方，责任都完全由我承担。

感谢我的父亲，感谢他在我很小的时候就向我介绍编程，而且一点都不担心我破坏他的设备。感谢妻子Erin在我编写本书期间对我一如既往的鼓励和支持。还要感谢儿子Ever，他的好奇心每天都会给我带来灵感。

目 录

第一部分 基础知识

第 1 章 起步	2
1.1 搭建编程环境	2
1.1.1 Python 2 和 Python 3	2
1.1.2 运行 Python 代码片段	3
1.1.3 Hello World 程序	3
1.2 在不同操作系统中搭建 Python 编程环境	3
1.2.1 在 Linux 系统中搭建 Python 编程环境	3
1.2.2 在 OS X 系统中搭建 Python 编程环境	6
1.2.3 在 Windows 系统中搭建 Python 编程环境	8
1.3 解决安装问题	12
1.4 从终端运行 Python 程序	13
1.4.1 在 Linux 和 OS X 系统中从终端运行 Python 程序	13
1.4.2 在 Windows 系统中从终端运行 Python 程序	13
1.5 小结	14
第 2 章 变量和简单数据类型	15
2.1 运行 hello_world.py 时发生的情况	15
2.2 变量	16
2.2.1 变量的命名和使用	16
2.2.2 使用变量时避免命名错误	17
2.3 字符串	18
2.3.1 使用方法修改字符串的大小写	19
2.3.2 合并（拼接）字符串	19

2.3.3 使用制表符或换行符来添加空白	20
2.3.4 删除空白	21
2.3.5 使用字符串时避免语法错误	22
2.3.6 Python 2 中的 print 语句	23
2.4 数字	24
2.4.1 整数	24
2.4.2 浮点数	25
2.4.3 使用函数 str() 避免类型错误	25
2.4.4 Python 2 中的整数	26
2.5 注释	27
2.5.1 如何编写注释	27
2.5.2 该编写什么样的注释	28
2.6 Python 之禅	28
2.7 小结	30
第 3 章 列表简介	31
3.1 列表是什么	31
3.1.1 访问列表元素	32
3.1.2 索引从 0 而不是 1 开始	32
3.1.3 使用列表中的各个值	33
3.2 修改、添加和删除元素	33
3.2.1 修改列表元素	34
3.2.2 在列表中添加元素	34
3.2.3 从列表中删除元素	35
3.3 组织列表	39
3.3.1 使用方法 sort() 对列表进行永久性排序	39
3.3.2 使用函数 sorted() 对列表进行临时排序	40
3.3.3 倒着打印列表	41

3.3.4 确定列表的长度	41	5.2 条件测试	65
3.4 使用列表时避免索引错误	42	5.2.1 检查是否相等	65
3.5 小结	43	5.2.2 检查是否相等时不考虑大 小写	65
第 4 章 操作列表.....	44	5.2.3 检查是否不相等	66
4.1 遍历整个列表	44	5.2.4 比较数字	67
4.1.1 深入地研究循环	45	5.2.5 检查多个条件	67
4.1.2 在 for 循环中执行更多的操作	46	5.2.6 检查特定值是否包含在列表中	68
4.1.3 在 for 循环结束后执行一些 操作	47	5.2.7 检查特定值是否不包含在列 表中	69
4.2 避免缩进错误	47	5.2.8 布尔表达式	69
4.2.1 忘记缩进	48	5.3 if 语句	70
4.2.2 忘记缩进额外的代码行	48	5.3.1 简单的 if 语句	70
4.2.3 不必要的缩进	49	5.3.2 if-else 语句	71
4.2.4 循环后不必要的缩进	49	5.3.3 if-elif-else 结构	72
4.2.5 遗漏了冒号	50	5.3.4 使用多个 elif 代码块	73
4.3 创建数值列表	51	5.3.5 省略 else 代码块	74
4.3.1 使用函数 range()	51	5.3.6 测试多个条件	74
4.3.2 使用 range() 创建数字列表	51	5.4 使用 if 语句处理列表	76
4.3.3 对数字列表执行简单的统计 计算	53	5.4.1 检查特殊元素	77
4.3.4 列表解析	53	5.4.2 确定列表不是空的	78
4.4 使用列表的一部分	54	5.4.3 使用多个列表	78
4.4.1 切片	54	5.5 设置 if 语句的格式	80
4.4.2 遍历切片	56	5.6 小结	80
4.4.3 复制列表	56	第 6 章 字典	81
4.5 元组	59	6.1 一个简单的字典	81
4.5.1 定义元组	59	6.2 使用字典	82
4.5.2 遍历元组中的所有值	59	6.2.1 访问字典中的值	82
4.5.3 修改元组变量	60	6.2.2 添加键-值对	83
4.6 设置代码格式	61	6.2.3 先创建一个空字典	83
4.6.1 格式设置指南	61	6.2.4 修改字典中的值	84
4.6.2 缩进	61	6.2.5 删除键-值对	85
4.6.3 行长	61	6.2.6 由类似对象组成的字典	86
4.6.4 空行	62	6.3 遍历字典	87
4.6.5 其他格式设置指南	62	6.3.1 遍历所有的键-值对	87
4.7 小结	63	6.3.2 遍历字典中的所有键	89
第 5 章 if 语句	64	6.3.3 按顺序遍历字典中的所有键	91
5.1 一个简单示例	64	6.3.4 遍历字典中的所有值	91
		6.4 嵌套	93

6.4.1 字典列表	93	8.4.1 在函数中修改列表	126
6.4.2 在字典中存储列表	95	8.4.2 禁止函数修改列表	129
6.4.3 在字典中存储字典	97	8.5 传递任意数量的实参	130
6.5 小结	99	8.5.1 结合使用位置实参和任意数量	
第 7 章 用户输入和 while 循环	100	实参	131
7.1 函数 <code>input()</code> 的工作原理	100	8.5.2 使用任意数量的关键字实参	131
7.1.1 编写清晰的程序	101	8.6 将函数存储在模块中	133
7.1.2 使用 <code>int()</code> 来获取数值输入	102	8.6.1 导入整个模块	133
7.1.3 求模运算符	103	8.6.2 导入特定的函数	134
7.1.4 在 Python 2.7 中获取输入	104	8.6.3 使用 <code>as</code> 给函数指定别名	134
7.2 while 循环简介	104	8.6.4 使用 <code>as</code> 给模块指定别名	135
7.2.1 使用 while 循环	104	8.6.5 导入模块中的所有函数	135
7.2.2 让用户选择何时退出	105	8.7 函数编写指南	136
7.2.3 使用标志	106	8.8 小结	137
7.2.4 使用 <code>break</code> 退出循环	107	第 9 章 类	138
7.2.5 在循环中使用 <code>continue</code>	108	9.1 创建和使用类	138
7.2.6 避免无限循环	109	9.1.1 创建 <code>Dog</code> 类	139
7.3 使用 while 循环来处理列表和字典	110	9.1.2 根据类创建实例	140
7.3.1 在列表之间移动元素	110	9.2 使用类和实例	142
7.3.2 删除包含特定值的所有列表		9.2.1 <code>Car</code> 类	143
元素	111	9.2.2 给属性指定默认值	143
7.3.3 使用用户输入来填充字典	112	9.2.3 修改属性的值	144
7.4 小结	113	9.3 继承	147
第 8 章 函数	114	9.3.1 子类的方法 <code>__init__()</code>	147
8.1 定义函数	114	9.3.2 Python 2.7 中的继承	149
8.1.1 向函数传递信息	115	9.3.3 给子类定义属性和方法	149
8.1.2 实参和形参	115	9.3.4 重写父类的方法	150
8.2 传递实参	116	9.3.5 将实例用作属性	150
8.2.1 位置实参	116	9.3.6 模拟实物	152
8.2.2 关键字实参	118	9.4 导人类	153
8.2.3 默认值	118	9.4.1 导入单个类	153
8.2.4 等效的函数调用	119	9.4.2 在一个模块中存储多个类	155
8.2.5 避免实参错误	120	9.4.3 从一个模块中导入多个类	156
8.3 返回值	121	9.4.4 导入整个模块	157
8.3.1 返回简单值	121	9.4.5 导入模块中的所有类	157
8.3.2 让实参变成可选的	122	9.4.6 在一个模块中导入另一个	
8.3.3 返回字典	123	模块	157
8.3.4 结合使用函数和 while 循环	124	9.4.7 自定义工作流程	158
8.4 传递列表	126	9.5 Python 标准库	159

9.6	类编码风格.....	161	11.1.1	单元测试和测试用例.....	188
9.7	小结.....	161	11.1.2	可通过的测试.....	188
第 10 章	文件和异常.....	162	11.1.3	不能通过的测试.....	190
10.1	从文件中读取数据.....	162	11.1.4	测试未通过时怎么办.....	191
10.1.1	读取整个文件.....	162	11.1.5	添加新测试.....	191
10.1.2	文件路径.....	164	11.2	测试类.....	193
10.1.3	逐行读取.....	165	11.2.1	各种断言方法.....	193
10.1.4	创建一个包含文件各行内容 的列表.....	166	11.2.2	一个要测试的类.....	194
10.1.5	使用文件的内容.....	166	11.2.3	测试 AnonymousSurvey 类.....	195
10.1.6	包含一百万位的大型文件.....	168	11.2.4	方法 setUp().....	197
10.1.7	圆周率值中包含你的生 日吗.....	168	11.3	小结.....	199
10.2	写入文件.....	169			
10.2.1	写入空文件.....	170	第二部分 项 目		
10.2.2	写入多行.....	170			
10.2.3	附加到文件.....	171	项目 1 外星人入侵.....	202	
10.3	异常.....	172	第 12 章 武装飞船.....	203	
10.3.1	处理 ZeroDivisionError 异常.....	172	12.1	规划项目.....	203
10.3.2	使用 try-except 代码块.....	173	12.2	安装 Pygame.....	204
10.3.3	使用异常避免崩溃.....	173	12.2.1	使用 pip 安装 Python 包.....	204
10.3.4	else 代码块.....	174	12.2.2	在 Linux 系统中安装 Pygame.....	206
10.3.5	处理 FileNotFoundError 异常.....	175	12.2.3	在 OS X 系统中安装 Pygame.....	207
10.3.6	分析文本.....	176	12.2.4	在 Windows 系统中安装 Pygame.....	207
10.3.7	使用多个文件.....	177	12.3	开始游戏项目.....	207
10.3.8	失败时一声不吭.....	178	12.3.1	创建 Pygame 窗口以及响应 用户输入.....	208
10.3.9	决定报告哪些错误.....	179	12.3.2	设置背景色.....	209
10.4	存储数据.....	180	12.3.3	创建设置类.....	210
10.4.1	使用 json.dump() 和 json. load().....	180	12.4	添加飞船图像.....	211
10.4.2	保存和读取用户生成的 数据.....	181	12.4.1	创建 Ship 类.....	212
10.4.3	重构.....	183	12.4.2	在屏幕上绘制飞船.....	213
10.5	小结.....	186	12.5	重构：模块 game_functions.....	214
第 11 章 测试代码.....		187	12.5.1	函数 check_events().....	214
11.1	测试函数.....	187	12.5.2	函数 update_screen().....	215
			12.6	驾驶飞船.....	216
			12.6.1	响应按键.....	216
			12.6.2	允许不断移动.....	217

12.6.3 左右移动	219	13.5 射杀外星人	246
12.6.4 调整飞船的速度	220	13.5.1 检测子弹与外星人的碰撞	246
12.6.5 限制飞船的活动范围	221	13.5.2 为测试创建大子弹	247
12.6.6 重构 check_events()	222	13.5.3 生成新的外星人群	248
12.7 简单回顾	223	13.5.4 提高子弹的速度	249
12.7.1 alien_invasion.py	223	13.5.5 重构 update_bullets()	249
12.7.2 settings.py	223	13.6 结束游戏	250
12.7.3 game_functions.py	223	13.6.1 检测外星人和飞船碰撞	250
12.7.4 ship.py	223	13.6.2 响应外星人和飞船碰撞	251
12.8 射击	224	13.6.3 有外星人到达屏幕底端	254
12.8.1 添加子弹设置	224	13.6.4 游戏结束	255
12.8.2 创建 Bullet 类	224	13.7 确定应运行游戏的哪些部分	255
12.8.3 将子弹存储到编组中	226	13.8 小结	256
12.8.4 开火	227	第 14 章 记分	257
12.8.5 删除已消失的子弹	228	14.1 添加 Play 按钮	257
12.8.6 限制子弹数量	229	14.1.1 创建 Button 类	258
12.8.7 创建函数 update_bullets()	229	14.1.2 在屏幕上绘制按钮	259
12.8.8 创建函数 fire_bullet()	230	14.1.3 开始游戏	261
12.9 小结	231	14.1.4 重置游戏	261
第 13 章 外星人	232	14.1.5 将 Play 按钮切换到非活动	
13.1 回顾项目	232	状态	263
13.2 创建第一个外星人	233	14.1.6 隐藏光标	263
13.2.1 创建 Alien 类	233	14.2 提高等级	264
13.2.2 创建 Alien 实例	234	14.2.1 修改速度设置	264
13.2.3 让外星人出现在屏幕上	235	14.2.2 重置速度	266
13.3 创建一群外星人	236	14.3 记分	267
13.3.1 确定一行可容纳多少个		14.3.1 显示得分	267
外星人	236	14.3.2 创建记分牌	268
13.3.2 创建多行外星人	236	14.3.3 在外星人被消灭时更新	
13.3.3 创建外星人群	237	得分	270
13.3.4 重构 create_fleet()	239	14.3.4 将消灭的每个外星人的点数	
13.3.5 添加行	240	都计入得分	271
13.4 让外星人群移动	242	14.3.5 提高点数	271
13.4.1 向右移动外星人	243	14.3.6 将得分圆整	272
13.4.2 创建表示外星人移动方向的		14.3.7 最高得分	274
设置	244	14.3.8 显示等级	276
13.4.3 检查外星人是否撞到了屏幕		14.3.9 显示余下的飞船数	279
边缘	244	14.4 小结	283
13.4.4 向下移动外星人群并改变移			
动方向	245		

项目 2 数据可视化	284		
第 15 章 生成数据	285		
15.1 安装 matplotlib	285	15.4.5 分析结果	305
15.1.1 在 Linux 系统中安装		15.4.6 绘制直方图	306
matplotlib	286	15.4.7 同时掷两个骰子	307
15.1.2 在 OS X 系统中安装		15.4.8 同时掷两个面数不同的	
matplotlib	286	骰子	309
15.1.3 在 Windows 系统中安装		15.5 小结	311
matplotlib	286		
15.1.4 测试 matplotlib	287	第 16 章 下载数据	312
15.1.5 matplotlib 画廊	287	16.1 CSV 文件格式	312
15.2 绘制简单的折线图	287	16.1.1 分析 CSV 文件头	313
15.2.1 修改标签文字和线条粗细	288	16.1.2 打印文件头及其位置	314
15.2.2 校正图形	289	16.1.3 提取并读取数据	314
15.2.3 使用 scatter()绘制散点图		16.1.4 绘制气温图表	315
并设置其样式	290	16.1.5 模块 datetime	316
15.2.4 使用 scatter()绘制一系		16.1.6 在图表中添加日期	317
列点	291	16.1.7 涵盖更长的时间	318
15.2.5 自动计算数据	292	16.1.8 再绘制一个数据系列	319
15.2.6 删除数据点的轮廓	293	16.1.9 给图表区域着色	320
15.2.7 自定义颜色	293	16.1.10 错误检查	321
15.2.8 使用颜色映射	294	16.2 制作世界人口地图：JSON 格式	324
15.2.9 自动保存图表	295	16.2.1 下载世界人口数据	324
15.3 随机漫步	295	16.2.2 提取相关的数据	324
15.3.1 创建 RandomWalk()类	296	16.2.3 将字符串转换为数字值	326
15.3.2 选择方向	296	16.2.4 获取两个字母的国别码	327
15.3.3 绘制随机漫步图	297	16.2.5 制作世界地图	329
15.3.4 模拟多次随机漫步	298	16.2.6 在世界地图上呈现数字	
15.3.5 设置随机漫步图的样式	299	数据	330
15.3.6 给点着色	299	16.2.7 绘制完整的世界人口地图	331
15.3.7 重新绘制起点和终点	300	16.2.8 根据人口数量将国家分组	333
15.3.8 隐藏坐标轴	301	16.2.9 使用 Pygal 设置世界地图的	
15.3.9 增加点数	301	样式	334
15.3.10 调整尺寸以适合屏幕	302	16.2.10 加亮颜色主题	335
15.4 使用 Pygal 模拟掷骰子	303	16.3 小结	337
15.4.1 安装 Pygal	304		
15.4.2 Pygal 画廊	304	第 17 章 使用 API	338
15.4.3 创建 Die 类	304	17.1 使用 Web API	338
15.4.4 掷骰子	305	17.1.1 Git 和 GitHub	338

17.1.5 处理响应字典	340	18.5 小结	381
17.1.6 概述最受欢迎的仓库	342	第 19 章 用户账户	382
17.1.7 监视 API 的速率限制	343	19.1 让用户能够输入数据	382
17.2 使用 Pygal 可视化仓库	344	19.1.1 添加新主题	382
17.2.1 改进 Pygal 图表	346	19.1.2 添加新条目	386
17.2.2 添加自定义工具提示	347	19.1.3 编辑条目	390
17.2.3 根据数据绘图	349	19.2 创建用户账户	392
17.2.4 在图表中添加可单击的 链接	350	19.2.1 应用程序 users	393
17.3 Hacker News API	350	19.2.2 登录页面	394
17.4 小结	353	19.2.3 注销	396
项目 3 Web 应用程序	354	19.2.4 注册页面	397
第 18 章 Django 入门	355	19.3 让用户拥有自己的数据	400
18.1 建立项目	355	19.3.1 使用 @login_required 限制 访问	400
18.1.1 制定规范	355	19.3.2 将数据关联到用户	402
18.1.2 建立虚拟环境	356	19.3.3 只允许用户访问自己的 主题	405
18.1.3 安装 virtualenv	356	19.3.4 保护用户的主题	405
18.1.4 激活虚拟环境	357	19.3.5 保护页面 edit_entry	406
18.1.5 安装 Django	357	19.3.6 将新主题关联到当前用户	406
18.1.6 在 Django 中创建项目	357	19.4 小结	408
18.1.7 创建数据库	358	第 20 章 设置应用程序的样式并对其进行 部署	409
18.1.8 查看项目	359	20.1 设置项目“学习笔记”的样式	409
18.2 创建应用程序	360	20.1.1 应用程序 django-bootstrap3	410
18.2.1 定义模型	360	20.1.2 使用 Bootstrap 来设置项目 “学习笔记”的样式	411
18.2.2 激活模型	362	20.1.3 修改 base.html	411
18.2.3 Django 管理网站	363	20.1.4 使用 jumbotron 设置主页的 样式	414
18.2.4 定义模型 Entry	365	20.1.5 设置登录页面的样式	415
18.2.5 迁移模型 Entry	366	20.1.6 设置 new_topic 页面的 样式	416
18.2.6 向管理网站注册 Entry	366	20.1.7 设置 topics 页面的样式	417
18.2.7 Django shell	367	20.1.8 设置 topic 页面中条目的 样式	417
18.3 创建网页：学习笔记主页	369	20.2 部署“学习笔记”	419
18.3.1 映射 URL	369	20.2.1 建立 Heroku 账户	420
18.3.2 编写视图	371		
18.3.3 编写模板	372		
18.4 创建其他网页	373		
18.4.1 模板继承	373		
18.4.2 显示所有主题的页面	375		
18.4.3 显示特定主题的页面	378		

20.2.2 安装 Heroku Toolbelt	420	20.2.14 改进 Heroku 部署	428
20.2.3 安装必要的包	420	20.2.15 确保项目的安全	429
20.2.4 创建包含包列表的文件 requirements.txt	421	20.2.16 提交并推送修改	430
20.2.5 指定 Python 版本	422	20.2.17 创建自定义错误页面	431
20.2.6 为部署到 Heroku 而修改 settings.py	422	20.2.18 继续开发	434
20.2.7 创建启动进程的 Procfile	423	20.2.19 设置 SECRET_KEY	434
20.2.8 为部署到 Heroku 而修改 wsgi.py	423	20.2.20 将项目从 Heroku 删除	434
20.2.9 创建用于存储静态文件的 目录	424	20.3 小结	435
20.2.10 在本地使用 gunicorn 服务器	424	附录 A 安装 Python	436
20.2.11 使用 Git 跟踪项目文件	425	附录 B 文本编辑器	441
20.2.12 推送到 Heroku	426	附录 C 寻求帮助	447
20.2.13 在 Heroku 上建立数据库	427	附录 D 使用 Git 进行版本控制	451
		后记	460

Part 1

第一部分

基础知识

本书的第一部分介绍编写 Python 程序所需要熟悉的基本概念，其中很多都适用于所有编程语言，因此它们在你的整个程序员生涯中都很有用。

第 1 章介绍在计算机中安装 Python，并运行第一个程序——它在屏幕上打印消息“Hello world!”。

第 2 章论述如何在变量中存储信息以及如何使用文本和数字。

第 3 章和第 4 章介绍列表。使用列表能够在一个变量中存储任意数量的信息，从而高效地处理数据：只需几行代码，你就能够处理数百、数千乃至数百万个值。

第 5 章讲解使用 if 语句来编写这样的代码：在特定条件满足时采取一种措施，而在该条件不满足时采取另一种措施。

第 6 章演示如何使用 Python 字典，将不同的信息关联起来。与列表一样，你也可以根据需要在字典中存储任意数量的信息。

第 7 章讲解如何从用户那里获取输入，以让程序变成交互式的。你还将学习 while 循环，它不断地运行代码块，直到指定的条件不再满足为止。

第 8 章介绍编写函数。函数是执行特定任务的被命名的代码块，你可以根据需要随时运行它。

第 9 章介绍类，它让你能够模拟实物，如小狗、小猫、人、汽车、火箭等，让你的代码能够表示任何真实或抽象的东西。

第 10 章介绍如何使用文件，以及如何处理错误以免程序意外地崩溃。你需要在程序关闭前保存数据，并在程序再次运行时读取它们。你将学习 Python 异常，它们让你能够未雨绸缪，从而让程序妥善地处理错误。

第 11 章为代码编写测试，以核实程序是否像你期望的那样工作。这样，扩展程序时，你就不用担心引入新的 bug。要想脱离初级程序员的阵容，跻身于中级程序员的行列，测试代码是你必须掌握的基本技能之一。

第1章

起 步

1



在本章中，你将运行自己的第一个程序——hello_world.py。为此，你首先需要检查自己的计算机是否安装了Python；如果没有安装，你需要安装它。你还要安装一个文本编辑器，用于编写和运行Python程序。你输入Python代码时，这个文本编辑器能够识别它们并突出显示不同的部分，让你能够轻松地了解代码的结构。

1.1 搭建编程环境

在不同的操作系统中，Python存在细微的差别，因此有几点你需要牢记在心。这里将介绍大家使用的两个主要的Python版本，并简要介绍Python的安装步骤。

1.1.1 Python 2 和 Python 3

当前，有两个不同的Python版本：Python 2和较新的Python 3。每种编程语言都会随着新概念和技术的推出而不断发展，Python的开发者也一直致力于丰富和强化其功能。大多数修改都是逐步进行的，你几乎意识不到，但如果系统的安装的是Python 3，那么有些使用Python 2编写的代码可能无法正确地运行。在本书中，我将指出Python 2和Python 3的重大差别，这样无论你安装的是哪个Python版本，都能够按书中的说明去做。

如果你的系统安装了这两个版本，请使用Python 3；如果没有安装Python，请安装Python 3；如果只安装了Python 2，也可直接使用它来编写代码，但还是尽快升级到Python 3为好，因为这样你就能使用最新的Python版本了。

1.1.2 运行 Python 代码片段

Python自带了一个在终端窗口中运行的解释器，让你无需保存并运行整个程序就能尝试运行Python代码片段。

本书将以如下方式列出代码片段：

```
❶ >>> print("Hello Python interpreter!")
Hello Python interpreter!
```

加粗的文本表示需要你输入之后按回车键来执行的代码。本书的大多数示例都是独立的小程序，你将在编辑器中执行它们，因为大多数代码都是这样编写出来的。然而，为高效地演示某基本概念，需要在Python终端会话中执行一系列代码片段。只要代码清单中包含三个尖括号（如❶所示），就意味着输出来自终端会话。稍后将演示如何在Python解释器中编写代码。

1.1.3 Hello World 程序

长期以来，编程界都认为刚接触一门新语言时，如果首先使用它来编写一个在屏幕上显示消息“Hello world!”的程序，将给你带来好运。

要使用Python来编写这种Hello World程序，只需一行代码：

```
print("Hello world!")
```

这种程序虽然简单，却有其用途：如果它能够在你的系统上正确地运行，你编写的任何Python程序都将如此。稍后将介绍如何在特定的系统中编写这样的程序。

1.2 在不同操作系统中搭建 Python 编程环境

Python是一种跨平台的编程语言，这意味着它能够运行在所有主要的操作系统中。在所有安装了Python的现代计算机上，都能够运行你编写的任何Python程序。然而，在不同的操作系统中，安装Python的方法存在细微的差别。

在这一节中，你将学习如何在自己的系统中安装Python和运行Hello World程序。你首先要检查自己的系统是否安装了Python，如果没有，就安装它；接下来，你需要安装一个简单的文本编辑器，并创建一个空的Python文件——hello_world.py。最后，你将运行Hello World程序，并排除各种故障。我将详细介绍如何在各种操作系统中完成这些任务，让你能够搭建一个对初学者友好的Python编程环境。

1.2.1 在 Linux 系统中搭建 Python 编程环境

Linux系统是为编程而设计的，因此在大多数Linux计算机中，都默认安装了Python。编写和维护Linux的人认为，你很可能会使用这种系统进行编程，他们也鼓励你这样做。鉴于此，要在

这种系统中编程，你几乎不用安装什么软件，也几乎不用修改设置。

1. 检查Python版本

在你的系统中运行应用程序Terminal（如果你使用的是Ubuntu，可按Ctrl + Alt + T），打开一个终端窗口。为确定是否安装了Python，执行命令python（请注意，其中的p是小写的）。输出将类似下面这样，它指出了安装的Python版本；最后的>>>是一个提示符，让你能够输入Python命令。

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

上述输出表明，当前计算机默认使用的Python版本为Python 2.7.6。看到上述输出后，如果要退出Python并返回到终端窗口，可按Ctrl + D或执行命令exit()。

要检查系统是否安装了Python 3，可能需要指定相应的版本。换句话说，如果输出指出默认版本为Python 2.7，请尝试执行命令python3：

```
$ python3
Python 3.5.0 (default, Sep 17 2015, 13:05:18)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

上述输出表明，系统中也安装了Python 3，因此你可以使用这两个版本中的任何一个。在这种情况下，请将本书中的命令python都替换为python3。大多数Linux系统都默认安装了Python，但如果你的Linux系统不知什么原因没有安装Python或只安装了Python 2，而你要安装Python 3，请参见附录A。

2. 安装文本编辑器

Geany是一款简单的文本编辑器：它易于安装；让你能够直接运行几乎所有的程序（而无需通过终端来运行）；使用不同的颜色来显示代码，以突出代码语法；在终端窗口中运行代码，让你能够习惯使用终端。附录B介绍了其他一些文本编辑器，但我强烈建议你使用Geany，除非你有充分的理由不这样做。

在大多数Linux系统中，都只需执行一个命令就可以安装Geany：

```
$ sudo apt-get install geany
```

如果这个命令不管用，请参阅<http://geany.org/Download/ThirdPartyPackages/>的说明。

3. 运行Hello World程序

为编写第一个程序，需要启动Geany。为此，可按超级（Super）键（俗称Windows键），并在系统中搜索Geany。找到Geany后，双击以启动它；再将其拖曳到任务栏或桌面上，以创建一个快捷方式。接下来，创建一个用于存储项目的文件夹，并将其命名为python_work（在文件名

和文件夹名中，最好使用小写字母，并使用下划线来表示空格，因为这是Python采用的命名约定）。回到Geany，选择菜单File ▶ Save As，将当前的空Python文件保存到文件夹python_work，并将其命名为hello_world.py。扩展名.py告诉Geany，文件包含的是Python程序；它还让Geany知道如何运行该程序，并以有益的方式突出其中的代码。

保存文件后，在其中输入下面一行代码：

```
print("Hello Python world!")
```

如果你的系统安装了多个Python版本，就必须对Geany进行配置，使其使用正确的版本。为此，可选择菜单Build（生成）▶ Set Build Commands（设置生成命令）；你将看到文字Compile（编译）和Execute（执行），它们旁边都有一个命令。默认情况下，这两个命令都是python，要让Geany使用命令python3，必须做相应的修改。

如果在终端会话中能够执行命令python3，请修改编译命令和执行命令，让Geany使用Python 3解释器。为此，将编译命令修改成下面这样：

```
python3 -m py_compile "%f"
```

你必须完全按上面的代码显示的那样输出这个命令，确保空格和大小写都完全相同。

将执行命令修改成下面这样：

```
python3 "%f"
```

同样，务必确保空格和大小写都完全与显示的相同。图1-1显示了该如何在Geany中配置这些命令。

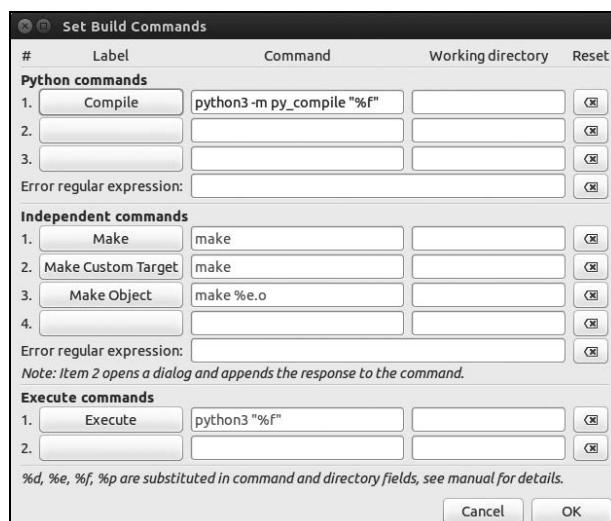


图1-1 在Linux中配置Geany，使其使用Python 3

现在来运行程序hello_world.py。为此，可选择菜单Build ▶ Execute、单击Execute图标（两个齿轮）或按F5。将弹出一个终端窗口，其中包含如下输出：

```
Hello Python world!  
-----  
(program exited with code: 0)  
Press return to continue
```

如果没有看到这样的输出，请检查你输入的每个字符。你是不是将print的首字母大写了？是不是遗漏了引号或括号？编程语言对语法的要求非常严格，只要你没有严格遵守语法，就会出错。如果代码都正确，这个程序也不能正确地运行，请参阅1.3节。

4. 在终端会话中运行Python代码

你可以打开一个终端窗口并执行命令python或python3，再尝试运行Python代码片段。检查Python版本时，你就这样做过。下面再次这样做，但在终端会话中输入如下代码行：

```
>>> print("Hello Python interpreter!")  
Hello Python interpreter!  
>>>
```

消息将直接打印到当前终端窗口中。别忘了，要关闭Python解释器，可按Ctrl + D或执行命令exit()。

1.2.2 在OS X系统中搭建Python编程环境

大多数OS X系统都默认安装了Python。确定安装了Python后，你还需安装一个文本编辑器，并确保其配置正确无误。

1. 检查是否安装了Python

在文件夹Applications/Utilities中，选择Terminal，打开一个终端窗口；你也可以按Command + 空格键，再输入terminal并按回车。为确定是否安装了Python，请执行命令python（注意，其中的p是小写的）。输出将类似于下面这样，它指出了安装的Python版本；最后的>>>是一个提示符，让你能够输入Python命令。

```
$ python  
Python 2.7.5 (default, Mar 9 2014, 22:15:05)  
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin  
Type "help", "copyright", "credits", or "license" for more information.  
>>>
```

上述输出表明，当前计算机默认使用的Python版本为Python 2.7.5。看到上述输出后，如果要退出Python并返回到终端窗口，可按Ctrl + D或执行命令exit()。

要检查系统是否安装了Python 3，可尝试执行命令python3。可能会出现一条错误消息，但如果输出指出系统安装了Python 3，则无需安装就可使用它。如果在你的系统中能够执行命令

python3，则对于本书的所有命令python，都请替换为命令python3。如果不知道出于什么原因你的系统没有安装Python，或者只安装了Python 2，而你又想安装Python 3，请参阅附录A。

2. 在终端会话中运行Python代码

你可以打开一个终端窗口并执行命令python或python3，再尝试运行Python代码片段。检查Python版本时，你就这样做过。下面再次这样做，但在终端会话中输入如下代码行：

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

消息将直接打印到当前终端窗口中。别忘了，要关闭Python解释器，可按Ctrl + D或执行命令exit()。

3. 安装文本编辑器

Sublime Text是一款简单的文本编辑器：它在OS X中易于安装；让你能够直接运行几乎所有程序（而无需通过终端）；使用不同的颜色来显示代码，以突出代码语法；在内嵌在Sublime Text窗口内的终端会话中运行代码，让你能够轻松地查看输出。附录B介绍了其他一些文本编辑器，但我强烈建议你使用Sublime Text，除非你有充分的理由不这样做。

要下载Sublime Text安装程序，可访问<http://sublimetext.com/3>，单击Download链接，并查找OS X安装程序。Sublime Text的许可策略非常灵活，你可以免费使用这款编辑器，但如果你喜欢它并想长期使用，建议你购买许可证。下载安装程序后，打开它，再将Sublime Text图标拖放到Applications文件夹。

4. 配置Sublime Text使其使用Python 3

如果你启动Python终端会话时使用的命令不是python，就需要配置Sublime Text，让它知道到系统的什么地方去查找正确的Python版本。要获悉Python解释器的完整路径，请执行如下命令：

```
$ type -a python3
python3 is /usr/local/bin/python3
```

现在，启动Sublime Text，并选择菜单Tools ▶ Build System ▶ New Build System，这将打开一个新的配置文件。删除其中的所有内容，再输入如下内容：

```
{
  "cmd": ["/usr/local/bin/python3", "-u", "$file"],
}
```

这些代码让Sublime Text使用命令python3来运行当前打开的文件。请确保其中的路径为你在前一步使用命令type -a python3获悉的路径。将这个配置文件命名为Python3.sublime-build，并将其保存到默认目录——你选择菜单Save时Sublime Text打开的目录。

5. 运行Hello World程序

为编写第一个程序，需要启动Sublime Text。为此，可打开文件夹Applications，并双击图标

Sublime Text；也可按Command + 空格键，再在弹出的搜索框中输入sublime text。

创建一个用于存储项目的文件夹，并将其命名为python_work（在文件名和文件夹名中，最好使用小写字母，并使用下划线来表示空格，因为这是Python采用的命名约定）。在Sublime Text中，选择菜单File ▶ Save As，将当前的空Python文件保存到文件夹python_work，并将其命名为hello_world.py。扩展名.py告诉Sublime Text，文件包含的是Python程序；它还让Sublime Text知道如何运行该程序，并以有益的方式突出其中的代码。

保存文件后，在其中输入下面一行代码：

```
print("Hello Python world!")
```

如果在系统中能够运行命令python，就可选择菜单Tools ▶ Build或按Ctrl + B来运行程序。如果你对Sublime Text进行了配置，使其使用的命令不是python，请选择菜单Tools ▶ Build System，再选择Python 3。这将把Python 3设置为默认使用的Python版本；此后，你就可选择菜单Tools ▶ Build或按Command+ B来运行程序了。

Sublime Text窗口底部将出现一个终端屏幕，其中包含如下输出：

```
Hello Python world!
[Finished in 0.1s]
```

如果没有看到这样的输出，请检查你输入的每个字符。你是不是将print的首字母大写了？是不是遗漏了引号或括号？编程语言对语法的要求非常严格，只要你没有严格遵守语法，就会出错。如果代码都正确，这个程序也不能正确地运行，请参阅1.3节。

1.2.3 在Windows系统中搭建Python编程环境

Windows系统并非都默认安装了Python，因此你可能需要下载并安装它，再下载并安装一个文本编辑器。

1. 安装Python

首先，检查你的系统是否安装了Python。为此，在“开始”菜单中输入command并按回车以打开一个命令窗口；你也可按住Shift键并右击桌面，再选择“在此处打开命令窗口”。在终端窗口中输入python并按回车；如果出现了Python提示符(>>>)，就说明你的系统安装了Python。然而，你也可能会看到一条错误消息，指出python是无法识别的命令。

如果是这样，就需要下载Windows Python安装程序。为此，请访问<http://python.org/downloads/>。你将看到两个按钮，分别用于下载Python 3和Python 2。单击用于下载Python 3的按钮，这会根据你的系统自动下载正确的安装程序。下载安装程序后，运行它。请务必选中复选框Add Python to PATH（如图1-2所示），这让你能够更轻松地配置系统。



图1-2 确保选中复选框Add Python to PATH

2. 启动Python终端会话

通过配置系统，让其能够在终端会话中运行Python，可简化文本编辑器的配置工作。打开一个命令窗口，并在其中执行命令python。如果出现了Python提示符（>>>），就说明Windows找到了你刚安装的Python版本。

```
C:\> python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果是这样，就可以直接跳到下一部分——“在终端会话中运行Python”。然而，输出可能类似于下面这样：

```
C:\> python
'python' is not recognized as an internal or external command, operable
program or batch file.
```

在这种情况下，你就必须告诉Windows如何找到你刚安装的Python版本。命令python通常存储在C盘，因此请在Windows资源管理器中打开C盘，在其中找到并打开以Python打头的文件夹，再找到文件python。例如，在我的计算机中，有一个名为Python35的文件夹，其中有一个名为python的文件，因此文件python的路径为C:\Python35\python。如果找不到这个文件，请在Windows资源管理器的搜索框中输入python，这将让你能够准确地获悉命令python在系统中的存储位置。

如果认为已知道命令python的路径，就在终端窗口中输入该路径进行测试。为此，打开一个命令窗口，并输入你确定的完整路径：

```
C:\> C:\Python35\python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 22:15:05) [MSC v.1900 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果可行，就说明你已经知道如何访问Python了。

3. 在终端会话中运行Python

在Python会话中执行下面的命令，并确认看到了输出“Hello Python world!”。

```
>>> print("Hello Python world!")
Hello Python world!
>>>
```

每当要运行Python代码片段时，都请打开一个命令窗口并启动Python终端会话。要关闭该终端会话，可按Ctrl + Z，再按回车键，也可执行命令exit()。

4. 安装文本编辑器

Geany是一款简单的文本编辑器：它易于安装；让你能够直接运行几乎所有的程序（而无需通过终端）；使用不同的颜色来显示代码，以突出代码语法；在终端窗口中运行代码，让你能够习惯使用终端。附录B介绍了其他一些文本编辑器，但我强烈建议你使用Geany，除非你有充分的理由不这样做。

要下载Windows Geany安装程序，可访问<http://geany.org/>，单击Download下的Releases，找到安装程序geany-1.25_setup.exe或类似的文件。下载安装程序后，运行它并接受所有的默认设置。

为编写第一个程序，需要启动Geany。为此，可按超级（Super）键（俗称Windows键），并在系统中搜索Geany。找到Geany后，双击以启动它；再将其拖曳到任务栏或桌面上，以创建一个快捷方式。接下来，创建一个用于存储项目的文件夹，并将其命名为python_work（在文件名和文件夹名中，最好使用小写字母，并使用下划线来表示空格，因为这是Python采用的命名约定）。回到Geany，选择菜单File ▶ Save As，将当前的空Python文件保存到文件夹python_work，并将其命名为hello_world.py。扩展名.py告诉Geany，文件包含的是Python程序；它还让Geany知道如何运行该程序，并以有益的方式突出其中的代码。

保存文件后，在其中输入下面一行代码：

```
print("Hello Python world!")
```

如果能够在系统中执行命令python，就无需配置Geany，因此你可以跳过下一部分，直接进入“运行Hello World程序”部分。如果启动Python解释器时必须指定路径，如C:\Python35\python，请按下面的说明对Geany进行配置。

5. 配置Geany

要配置Geany，请选择菜单Build ▶ Set Build Commands；你将看到文字Compile和Execute，它们旁边都有一个命令。默认情况下，编译命令和执行命令的开头都是python，但Geany不知道命

令python存储在系统的什么地方，因此你需要在其中添加你在终端会话中使用的路径。

为此，在编译命令和执行命令中，加上命令python所在的驱动器和文件夹。其中编译命令应类似于下面这样：

```
C:\Python35\python -m py_compile "%f"
```

在你的系统中，路径可能稍有不同，但请务必确保空格和大小写与这里显示的一致。

执行命令应类似于下面这样：

```
C:\Python35\python "%f"
```

同样，指定执行命令时，务必确保空格和大小写与这里显示的一致。图1-3显示了该如何在Geany中配置这些命令。

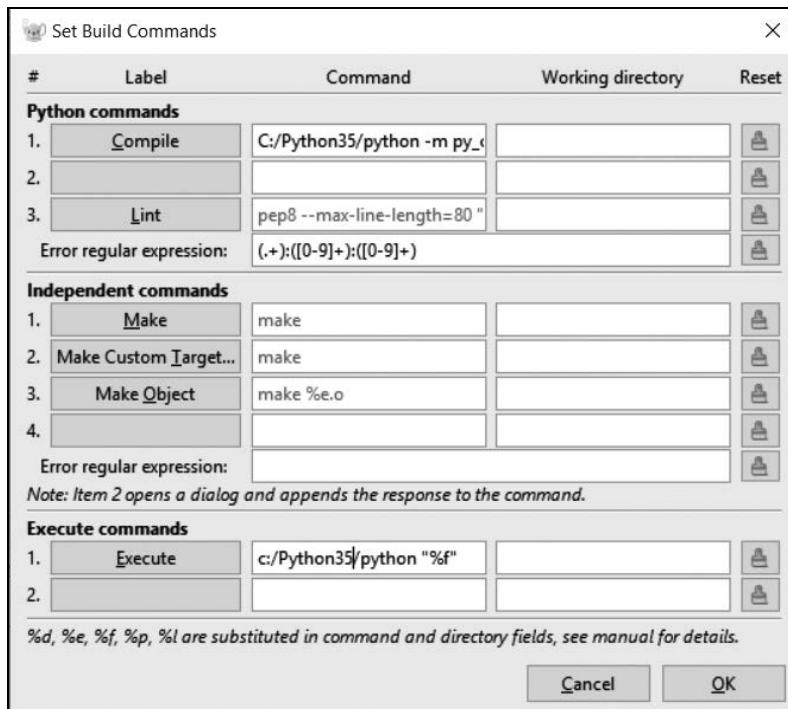


图1-3 在Windows中配置Geany，使其使用Python 3

正确地设置这些命令后，单击OK按钮。

6. 运行Hello World程序

现在应该能够成功地运行程序了。请运行程序hello_world.py；为此，可选择菜单Build ▶ Execute、单击Execute图标（两个齿轮）或按F5。将弹出一个终端窗口，其中包含如下输出：

```
Hello Python world!
```

```
-----  
(program exited with code: 0)  
Press return to continue
```

如果没有看到这样的输出，请检查你输入的每个字符。你是不是将print的首字母大写了？是不是遗漏了引号或括号？编程语言对语法的要求非常严格，只要你没有严格遵守语法，就会出错。如果代码都正确，这个程序也不能正确地运行，请参阅下一节。

1.3 解决安装问题

如果你按前面的步骤做，应该能够成功地搭建编程环境。但如果你始终无法运行程序hello_world.py，可尝试如下几个解决方案。

- ❑ 程序存在严重的错误时，Python将显示traceback。Python会仔细研究文件，试图找出其中的问题。trackback可能会提供线索，让你知道是什么问题让程序无法运行。
- ❑ 离开计算机，先休息一会儿，再尝试。别忘了，在编程中，语法非常重要，即便是少一个冒号、引号不匹配或括号不匹配，都可能导致程序无法正确地运行。请再次阅读本章相关的内容，再次审视你所做的工作，看看能否找出错误。
- ❑ 推倒重来。你也许不需要把一切都推倒重来，但将文件hello_world.py删除并重新创建它也许是合理的选择。
- ❑ 让别人在你的计算机或其他计算机上按本章的步骤重做一遍，并仔细观察。你可能遗漏了一小步，而别人刚好没有遗漏。
- ❑ 请懂Python的人帮忙。当你有这样的想法时，可能会发现在你认识的人当中就有人使用Python。
- ❑ 本章的安装说明在网上也可以找到，其网址为<https://www.nostarch.com/pythoncrash-course/>。对你来说，在线版也许更合适。
- ❑ 到网上寻求帮助。附录C提供了很多在线资源，如论坛或在线聊天网站，你可以前往这些地方，请求解决过你面临的问题的人提供解决方案。

不要担心这会打扰经验丰富的程序员。每个程序员都遇到过问题，而大多数程序员都会乐意帮助你正确地设置系统。只要能清晰地说明你要做什么、尝试了哪些方法及其结果，就很可能有人能够帮到你。正如前言中指出的，Python社区对初学者非常友好。

任何现代计算机都能够运行Python，如果你遇到了困难，请想办法寻求帮助吧。前期的问题可能令人沮丧，但很值得你花时间去解决。能够运行hello_world.py后，你就可以开始学习Python了，而且编程工作会更有趣，也更令人愉快。

1.4 从终端运行 Python 程序

1

你编写的大多数程序都将直接在文本编辑器中运行，但有时候，从终端运行程序很有用。例如，你可能想直接运行既有的程序。

在任何安装了Python的系统上都可以这样做，前提是你知道如何进入程序文件所在的目录。为尝试这样做，请确保已将文件hello_world.py存储到了桌面的python_work文件夹中。

1.4.1 在 Linux 和 OS X 系统中从终端运行 Python 程序

在Linux和OS X系统中，从终端运行Python程序的方式相同。在终端会话中，可使用终端命令cd（表示切换目录，change directory）在文件系统中导航。命令ls（list的简写）显示当前目录中所有未隐藏的文件。

为运行程序hello_world.py，请打开一个新的终端窗口，并执行下面的命令：

```

❶ ~$ cd Desktop/python_work/
❷ ~/Desktop/python_work$ ls
    hello_world.py
❸ ~/Desktop/python_work$ python hello_world.py
Hello Python world!

```

这里使用了命令cd来切换到文件夹Desktop/python_work（见❶）。接下来，使用命令ls来确认这个文件夹中包含文件hello_world.py（见❷）。最后，使用命令python hello_world.py来运行这个文件（见❸）。

就这么简单。要运行Python程序，只需使用命令python（或python3）即可。

1.4.2 在 Windows 系统中从终端运行 Python 程序

在命令窗口中，要在文件系统中导航，可使用终端命令cd；要列出当前目录中的所有文件，可使用命令dir（表示目录，directory）。

为运行程序hello_world.py，请打开一个新的终端窗口，并执行下面的命令：

```

❶ C:\> cd Desktop\python_work
❷ C:\Desktop\python_work> dir
    hello_world.py
❸ C:\Desktop\python_work> python hello_world.py
Hello Python world!

```

这里使用了命令cd来切换到文件夹Desktop\python_work（见❶）。接下来，使用命令dir来确认这个文件夹中包含文件hello_world.py（见❷）。最后，使用命令python hello_world.py来运行这个文件（见❸）。

如果你没有对系统进行配置以使用简单命令python，就可能需要指定这个命令的路径：

```
C:\$ cd Desktop\python_work  
C:\Desktop\python_work\$ dir  
hello_world.py  
C:\Desktop\python_work\$ C:\Python35\python hello_world.py  
Hello Python world!
```

大多数程序都可以直接从编辑器运行，但需要解决的问题比较复杂时，你编写的程序可能需要从终端运行。

动手试一试

本章的练习都是探索性的，但从第2章开始将要求你用那一章学到的知识来解决问题。

1-1 python.org: 浏览 Python 主页 (<http://python.org/>)，寻找你感兴趣的主題。你对 Python 越熟悉，这个网站对你来说就越有用。

1-2 输入错误：打开你刚创建的文件 `hello_world.py`，在代码中添加一个输入错误，再运行这个程序。输入错误会引发错误吗？你能理解显示的错误消息吗？你能添加一个不会导致错误的输入错误吗？你凭什么认为它不会导致错误？

1-3 无穷的技艺：如果你编程技艺无穷，你打算开发什么样的程序呢？你就要开始学习编程了；如果心中有目标，就能立即将新学到的技能付诸应用；现在正是草拟目标的大好时机。将想法记录下来是个不错的习惯，这样每当需要开始新项目时，都可参考它们。现在请花点时间描绘三个你想创建的程序。

1.5 小结

在本章中，你大致了解了Python，并在自己的系统中安装了Python。你还安装了一个文本编辑器，以简化Python代码的编写工作。你学习了如何在终端会话中运行Python代码片段，并运行了第一个货真价实的程序——`hello_world.py`。你还大致了解了如何解决安装问题。

在下一章，你将学习如何在Python程序中使用各种数据和变量。

变量和简单数据类型



在本章中，你将学习可在Python程序中使用的各种数据，还将学习如何将数据存储到变量中，以及如何在程序中使用这些变量。

2.1 运行hello_world.py时发生的情况

运行hello_world.py时，Python都做了些什么呢？下面来深入研究一下。实际上，即便是运行简单的程序，Python所做的工作也相当多：

```
hello_world.py
```

```
print("Hello Python world!")
```

运行上述代码时，你将看到如下输出：

```
Hello Python world!
```

运行文件hello_world.py时，末尾的.py指出这是一个Python程序，因此编辑器将使用Python解释器来运行它。Python解释器读取整个程序，确定其中每个单词的含义。例如，看到单词print时，解释器就会将括号中的内容打印到屏幕，而不会管括号中的内容是什么。

编写程序时，编辑器会以各种方式突出程序的不同部分。例如，它知道print是一个函数的名称，因此将其显示为蓝色；它知道“Hello Python world!”不是Python代码，因此将其显示为橙色。这种功能称为语法突出，在你刚开始编写程序时很有帮助。

2.2 变量

下面来尝试在hello_world.py中使用一个变量。在这个文件开头添加一行代码，并对第2行代码进行修改，如下所示：

```
message = "Hello Python world!"  
print(message)
```

运行这个程序，看看结果如何。你会发现，输出与以前相同：

```
Hello Python world!
```

我们添加了一个名为message的变量。每个变量都存储了一个值——与变量相关联的信息。在这里，存储的值为文本“Hello Python world!”。

添加变量导致Python解释器需要做更多工作。处理第1行代码时，它将文本“Hello Python world!”与变量message关联起来；而处理第2行代码时，它将与变量message关联的值打印到屏幕。

下面来进一步扩展这个程序：修改hello_world.py，使其再打印一条消息。为此，在hello_world.py中添加一个空行，再添加下面两行代码：

```
message = "Hello Python world!"  
print(message)  
  
message = "Hello Python Crash Course world!"  
print(message)
```

现在如果运行这个程序，将看到两行输出：

```
Hello Python world!  
Hello Python Crash Course world!
```

在程序中可随时修改变量的值，而Python将始终记录变量的最新值。

2.2.1 变量的命名和使用

在Python中使用变量时，需要遵守一些规则和指南。违反这些规则将引发错误，而指南旨在让你编写的代码更容易阅读和理解。请务必牢记下述有关变量的规则。

- ❑ 变量名只能包含字母、数字和下划线。变量名可以字母或下划线打头，但不能以数字打头，例如，可将变量命名为message_1，但不能将其命名为1_message。
- ❑ 变量名不能包含空格，但可使用下划线来分隔其中的单词。例如，变量名greeting_message可行，但变量名greeting message会引发错误。
- ❑ 不要将Python关键字和函数名用作变量名，即不要使用Python保留用于特殊用途的单词，如print（请参见附录A.4）。

□ 变量名应既简短又具有描述性。例如，name比n好，student_name比s_n好，name_length比length_of_persons_name好。

□ 慎用小写字母l和大写字母O，因为它们可能被人错看成数字1和0。

要创建良好的变量名，需要经过一定的实践，在程序复杂而有趣时尤其如此。随着你编写的程序越来越多，并开始阅读别人编写的代码，将越来越善于创建有意义的变量名。

2

注意 就目前而言，应使用小写的Python变量名。在变量名中使用大写字母虽然不会导致错误，但避免使用大写字母是个不错的主意。

2.2.2 使用变量时避免命名错误

程序员都会犯错，而且大多数程序员每天都会犯错。虽然优秀的程序员也会犯错，但他们也知道如何高效地消除错误。下面来看一种你可能会犯的错误，并学习如何消除它。

我们将有意地编写一些引发错误的代码。请输入下面的代码，包括其中以粗体显示但拼写不正确的单词mesage：

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

程序存在错误时，Python解释器将竭尽所能地帮助你找出问题所在。程序无法成功地运行时，解释器会提供一个traceback。 traceback是一条记录，指出了解释器尝试运行代码时，在什么地方陷入了困境。下面是你不小心错误地拼写了变量名时，Python解释器提供的traceback：

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷      print(mesage)
❸ NameError: name 'mesage' is not defined
```

解释器指出，文件hello_world.py的第2行存在错误（见❶）；它列出了这行代码，旨在帮助你快速找出错误（见❷）；它还指出了它发现的是什么样的错误（见❸）。在这里，解释器发现了一个名称错误，并指出打印的变量mesage未定义：Python无法识别你提供的变量名。名称错误通常意味着两种情况：要么是使用变量前忘记了给它赋值，要么是输入变量名时拼写不正确。

在这个示例中，第2行的变量名message中遗漏了字母s。Python解释器不会对代码做拼写检查，但要求变量名的拼写一致。例如，如果在代码的另一个地方也将message错误地拼写成了mesage，结果将如何呢？

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

在这种情况下，程序将成功地运行：

```
Hello Python Crash Course reader!
```

计算机一丝不苟，但不关心拼写是否正确。因此，创建变量名和编写代码时，你无需考虑英语中的拼写和语法规则。

很多编程错误都很简单，只是在程序的某一行输错了一个字符。为找出这种错误而花费很长时间的大有人在。很多程序员天资聪颖、经验丰富，却为找出这种细微的错误花费数小时。你可能觉得这很好笑，但别忘了，在你的编程生涯中，经常会有同样的遭遇。

注意 要理解新的编程概念，最佳的方式是尝试在程序中使用它们。如果你在做本书的练习时陷入了困境，请尝试做点其他的事情。如果这样做后依然无法摆脱困境，请复习相关内容。如果这样做后情况依然如故，请参阅附录C的建议。

动手试一试

请完成下面的练习，在做每个练习时，都编写一个独立的程序。保存每个程序时，使用符合标准 Python 约定的文件名：使用小写字母和下划线，如 simple_message.py 和 simple_messages.py。

2-1 简单消息：将一条消息存储到变量中，再将其打印出来。

2-2 多条简单消息：将一条消息存储到变量中，将其打印出来；再将变量的值修改为一条新消息，并将其打印出来。

2.3 字符串

大多数程序都定义并收集某种数据，然后使用它们来做些有意义的事情。鉴于此，对数据进行分类大有裨益。我们将介绍的第一种数据类型是字符串。字符串虽然看似简单，但能够以很多不同的方式使用它们。

字符串就是一系列字符。在Python中，用引号括起的都是字符串，其中的引号可以是单引号，也可以是双引号，如下所示：

```
"This is a string."  
'This is also a string.'
```

这种灵活性让你能够在字符串中包含引号和撇号：

```
'I told my friend, "Python is my favorite language!"'  
"The language 'Python' is named after Monty Python, not the snake."  
"One of Python's strengths is its diverse and supportive community."
```

下面来看一些使用字符串的方式。

2.3.1 使用方法修改字符串的大小写

对于字符串，可执行的最简单的操作之一是修改其中的单词的大小写。请看下面的代码，并尝试判断其作用：

`name.py`

```
name = "ada lovelace"
print(name.title())
```

将这个文件保存为`name.py`，再运行它。你将看到如下输出：

```
Ada Lovelace
```

在这个示例中，小写的字符串"ada lovelace"存储到了变量`name`中。在`print()`语句中，方法`title()`出现在这个变量的后面。方法是Python可对数据执行的操作。在`name.title()`中，`name`后面的句点(.)让Python对变量`name`执行方法`title()`指定的操作。每个方法后面都跟着一对括号，这是因为方法通常需要额外的信息来完成其工作。这种信息是在括号内提供的。函数`title()`不需要额外的信息，因此它后面的括号是空的。

`title()`以首字母大写的方式显示每个单词，即将每个单词的首字母都改为大写。这很有用，因为你经常需要将名字视为信息。例如，你可能希望程序将值Ada、ADA和ada视为同一个名字，并将它们都显示为Ada。

还有其他几个很有用的大小写处理方法。例如，要将字符串改为全部大写或全部小写，可以像下面这样做：

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

这些代码的输出如下：

```
ADA LOVELACE
ada lovelace
```

存储数据时，方法`lower()`很有用。很多时候，你无法依靠用户来提供正确的大小写，因此需要将字符串先转换为小写，再存储它们。以后需要显示这些信息时，再将其转换为最合适的小写方式。

2.3.2 合并（拼接）字符串

在很多情况下，都需要合并字符串。例如，你可能想将姓和名存储在不同的变量中，等要显

示姓名时再将它们合而为一：

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = first_name + " " + last_name

print(full_name)
```

Python使用加号（+）来合并字符串。在这个示例中，我们使用+来合并first_name、空格和last_name，以得到完整的姓名（见❶），其结果如下：

```
ada lovelace
```

这种合并字符串的方法称为拼接。通过拼接，可使用存储在变量中的信息来创建完整的消息。下面来看一个例子：

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

❶ print("Hello, " + full_name.title() + "!")
```

在这里，一个问候用户的句子中使用了全名（见❶），并使用了方法title()来将姓名设置为合适的格式。这些代码显示一条格式良好的简单问候语：

```
Hello, Ada Lovelace!
```

你可以使用拼接来创建消息，再把整条消息都存储在一个变量中：

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name

❶ message = "Hello, " + full_name.title() + "!"
❷ print(message)
```

上述代码也显示消息“Hello, Ada Lovelace!”，但将这条消息存储到了一个变量中（见❶），这让最后的print语句简单得多（见❷）。

2.3.3 使用制表符或换行符来添加空白

在编程中，空白泛指任何非打印字符，如空格、制表符和换行符。你可使用空白来组织输出，以使其更易读。

要在字符串中添加制表符，可使用字符组合\t，如下述代码的❶处所示：

```
>>> print("Python")
Python
```

❶ >>> print("\tPython")
Python

要在字符串中添加换行符，可使用字符组合\n：

2

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

还可在同一个字符串中同时包含制表符和换行符。字符串"\n\t"让Python换到下一行，并在下一行开头添加一个制表符。下面的示例演示了如何使用一个单行字符串来生成四行输出：

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

在接下来的两章中，你将使用为数不多的几行代码来生成很多行输出，届时制表符和换行符将提供极大的帮助。

2.3.4 删 除 空 白

在程序中，额外的空白可能令人迷惑。对程序员来说，'python'和'python '看起来几乎没什么两样，但对程序来说，它们却是两个不同的字符串。Python能够发现'python '中额外的空白，并认为它是有意义的——除非你告诉它不是这样的。

空白很重要，因为你经常需要比较两个字符串是否相同。例如，一个重要的示例是，在用户登录网站时检查其用户名。但在一些简单得多的情形下，额外的空格也可能令人迷惑。所幸在Python中，删除用户输入的数据中的多余的空白易如反掌。

Python能够找出字符串开头和末尾多余的空白。要确保字符串末尾没有空白，可使用方法rstrip()。

❶ >>> favorite_language = 'python '
❷ >>> favorite_language
'python '
❸ >>> favorite_language.rstrip()
'python'
❹ >>> favorite_language
'python '

存储在变量favorite_language中的字符串末尾包含多余的空白（见❶）。你在终端会话中向Python询问这个变量的值时，可看到末尾的空格（见❷）。对变量favorite_language调用方法rstrip()后（见❸），这个多余的空格被删除了。然而，这种删除只是暂时的，接下来再次询问

favorite_language的值时，你会发现这个字符串与输入时一样，依然包含多余的空白（见❸）。

要永久删除这个字符串中的空白，必须将删除操作的结果存回到变量中：

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

为删除这个字符串中的空白，你需要将其末尾的空白剔除，再将结果存回到原来的变量中（见❶）。在编程中，经常需要修改变量的值，再将新值存回到原来的变量中。这就是变量的值可能随程序的运行或用户输入数据而发生变化的原因。

你还可以剔除字符串开头的空白，或同时剔除字符串两端的空白。为此，可分别使用方法lstrip()和strip()：

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python'
❸ >>> favorite_language.lstrip()
' python '
❹ >>> favorite_language.strip()
' python'
```

在这个示例中，我们首先创建了一个开头和末尾都有空白的字符串（见❶）。接下来，我们分别删除末尾（见❷）、开头（见❸）和两端（见❹）的空格。尝试使用这些剥除函数有助于你熟悉字符串操作。在实际程序中，这些剥除函数最常用于在存储用户输入前对其进行清理。

2.3.5 使用字符串时避免语法错误

语法错误是一种时不时会遇到的错误。程序中包含非法的Python代码时，就会导致语法错误。例如，在用单引号括起的字符串中，如果包含撇号，就将导致错误。这是因为这会导致Python将第一个单引号和撇号之间的内容视为一个字符串，进而将余下的文本视为Python代码，从而引发错误。

下面演示了如何正确地使用单引号和双引号。请将该程序保存为apostrophe.py，再运行它：

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

撇号位于两个双引号之间，因此Python解释器能够正确地理解这个字符串：

```
One of Python's strengths is its diverse community.
```

然而，如果你使用单引号，Python将无法正确地确定字符串的结束位置：

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

而你将看到如下输出：

```
File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
               ^
SyntaxError: invalid syntax
```

从上述输出可知，错误发生在第二个单引号后面（见❶）。这种语法错误表明，在解释器看来，其中的有些内容不是有效的Python代码。错误的来源多种多样，这里指出一些常见的。学习编写Python代码时，你可能会经常遇到语法错误。语法错误也是最不具体的错误类型，因此可能难以找出并修复。受困于非常棘手的错误时，请参阅附录C提供的建议。

注意 编写程序时，编辑器的语法突出功能可帮助你快速找出某些语法错误。看到Python代码以普通句子的颜色显示，或者普通句子以Python代码的颜色显示时，就可能意味着文件中存在引号不匹配的情况。

2.3.6 Python 2 中的 print 语句

在Python 2中，print语句的语法稍有不同：

```
>>> python2.7
>>> print "Hello Python 2.7 world!"
Hello Python 2.7 world!
```

在Python 2中，无需将要打印的内容放在括号内。从技术上说，Python 3中的print是一个函数，因此括号必不可少。有些Python 2 print语句也包含括号，但其行为与Python 3中稍有不同。简单地说，在Python 2代码中，有些print语句包含括号，有些不包含。

动手试一试

在做下面的每个练习时，都编写一个独立的程序，并将其保存为名称类似于 name_cases.py 的文件。如果遇到了困难，请休息一会儿或参阅附录 C 提供的建议。

2-3 个性化消息：将用户的姓名存到一个变量中，并向该用户显示一条消息。显示的消息应非常简单，如“Hello Eric, would you like to learn some Python today?”。

2-4 调整名字的大小写：将一个人名存储到一个变量中，再以小写、大写和首字母大写的方式显示这个人名。

2-5 名言：找一句你钦佩的名人说的名言，将这个名人的姓名和他的名言打印出来。输出应类似于下面这样（包括引号）：

Albert Einstein once said, “A person who never made a mistake never tried anything new.”

2-6 名言 2：重复练习 2-5，但将名人的姓名存储在变量 famous_person 中，再创建要显示的消息，并将其存储在变量 message 中，然后打印这条消息。

2-7 剔除人名中的空白：存储一个人名，并在其开头和末尾都包含一些空白字符。务必至少使用字符组合"\t"和"\n"各一次。

打印这个人名，以显示其开头和末尾的空白。然后，分别使用剔除函数 lstrip()、rstrip() 和 strip() 对人名进行处理，并将结果打印出来。

2.4 数字

在编程中，经常使用数字来记录游戏得分、表示可视化数据、存储Web应用信息等。Python根据数字的用法以不同的方式处理它们。鉴于整数使用起来最简单，下面就先来看看Python是如何管理它们的。

2.4.1 整数

在Python中，可对整数执行加（+）减（-）乘（*）除（/）运算。

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

在终端会话中，Python直接返回运算结果。Python使用两个乘号表示乘方运算：

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Python还支持运算次序，因此你可在同一个表达式中使用多种运算。你还可以使用括号来修改运算次序，让Python按你指定的次序执行运算，如下所示：

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

2

在这些示例中，空格不影响Python计算表达式的方式，它们的存在旨在让你阅读代码时，能迅速确定先执行哪些运算。

2.4.2 浮点数

Python将带小数点的数字都称为浮点数。大多数编程语言都使用了这个术语，它指出了这样一个事实：小数点可出现在数字的任何位置。每种编程语言都须细心设计，以妥善地处理浮点数，确保不管小数点出现在什么位置，数字的行为都是正常的。

从很大程度上说，使用浮点数时都无需考虑其行为。你只需输入要使用的数字，Python通常都会按你期望的方式处理它们：

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

但需要注意的是，结果包含的小数位数可能是不确定的：

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

所有语言都存在这种问题，没有什么可担心的。Python会尽力找到一种方式，以尽可能精确地表示结果，但鉴于计算机内部表示数字的方式，这在有些情况下很难。就现在而言，暂时忽略多余的小数位数即可；在第二部分的项目中，你将学习在需要时处理多余小数位的方式。

2.4.3 使用函数 str() 避免类型错误

你经常需要在消息中使用变量的值。例如，假设你要祝人生日快乐，可能会编写类似于下面的代码：

birthday.py

```
age = 23
message = "Happy " + age + "rd Birthday!"
```

```
print(message)
```

你可能认为，上述代码会打印一条简单的生日祝福语：Happy 23rd birthday!。但如果你运行这些代码，将发现它们会引发错误：

```
Traceback (most recent call last):
  File "birthday.py", line 2, in <module>
    message = "Happy " + age + "rd Birthday!"
❶ TypeError: Can't convert 'int' object to str implicitly
```

这是一个类型错误，意味着Python无法识别你使用的信息。在这个示例中，Python发现你使用了一个值为整数（int）的变量，但它不知道该如何解读这个值（见❶）。Python知道，这个变量表示的可能是数值23，也可能是字符2和3。像上面这样在字符串中使用整数时，需要显式地指出你希望Python将这个整数用作字符串。为此，可调用函数str()，它让Python将非字符串值表示为字符串：

```
age = 23
message = "Happy " + str(age) + "rd Birthday!"

print(message)
```

这样，Python就知道你要将数值23转换为字符串，进而在生日祝福消息中显示字符2和3。经过上述处理后，将显示你期望的消息，而不会引发错误：

```
Happy 23rd Birthday!
```

大多数情况下，在Python中使用数字都非常简单。如果结果出乎意料，请检查Python是否按你期望的方式将数字解读为了数值或字符串。

2.4.4 Python 2 中的整数

在Python 2中，将两个整数相除得到的结果稍有不同：

```
>>> python2.7
>>> 3 / 2
1
```

Python返回的结果为1，而不是1.5。在Python 2中，整数除法的结果只包含整数部分，小数部分被删除。请注意，计算整数结果时，采取的方式不是四舍五入，而是将小数部分直接删除。

在Python 2中，若要避免这种情况，务必确保至少有一个操作数为浮点数，这样结果也将为浮点数：

```
>>> 3 / 2
1
>>> 3.0 / 2
```

```
1.5
>>> 3 / 2.0
1.5
>>> 3.0 / 2.0
1.5
```

2

从Python 3转而用Python 2或从Python 2转而用Python 3时，这种除法行为常常会令人迷惑。使用或编写同时使用浮点数和整数的代码时，一定要注意这种异常行为。

动手试一试

2-8 数字 8：编写 4 个表达式，它们分别使用加法、减法、乘法和除法运算，但结果都是数字 8。为使用 `print` 语句来显示结果，务必将这些表达式用括号括起来，也就是说，你应该编写 4 行类似于下面的代码：

```
print(5 + 3)
```

输出应为 4 行，其中每行都只包含数字 8。

2-9 最喜欢的数字：将你最喜欢的数字存储在一个变量中，再使用这个变量创建一条消息，指出你最喜欢的数字，然后将这条消息打印出来。

2.5 注释

在大多数编程语言中，注释都是一项很有用的功能。本书前面编写的程序中都只包含Python代码，但随着程序越来越大、越来越复杂，就应在其中添加说明，对你解决问题的方法进行大致的阐述。注释让你能够使用自然语言在程序中添加说明。

2.5.1 如何编写注释

在Python中，注释用井号（#）标识。井号后面的内容都会被Python解释器忽略，如下所示：

`comment.py`

```
# 向大家问好
print("Hello Python people!")
```

Python解释器将忽略第1行，只执行第2行。

```
Hello Python people!
```

2.5.2 该编写什么样的注释

编写注释的主要目的是阐述代码要做什么，以及是如何做的。在开发项目期间，你对各个部分如何协同工作了如指掌，但过段时间后，有些细节你可能不记得了。当然，你总是可以通过研究代码来确定各个部分的工作原理，但通过编写注释，以清晰的自然语言对解决方案进行概述，可节省很多时间。

要成为专业程序员或与其他程序员合作，就必须编写有意义的注释。当前，大多数软件都是合作编写的，编写者可能是同一家公司的多名员工，也可能是众多致力于同一个开源项目的人员。训练有素的程序员都希望代码中包含注释，因此你最好从现在开始就在程序中添加描述性注释。作为新手，最值得养成的习惯之一是，在代码中编写清晰、简洁的注释。

如果不确定是否要编写注释，就问问自己，找到合理的解决方案前，是否考虑了多个解决方案。如果答案是肯定的，就编写注释对你的解决方案进行说明吧。相比回过头去再添加注释，删除多余的注释要容易得多。从现在开始，本书的示例都将使用注释来阐述代码的工作原理。

动手试一试

2-10 添加注释：选择你编写的两个程序，在每个程序中都至少添加一条注释。如果程序太简单，实在没有什么需要说明的，就在程序文件开头加上你的姓名和当前日期，再用一句话阐述程序的功能。

2.6 Python之禅

编程语言Perl曾在互联网领域长期占据着统治地位，早期的大多数交互式网站使用的都是Perl脚本。彼时，“解决问题的办法有多个”被Perl社区奉为座右铭。这种理念一度深受大家的喜爱，因为这种语言固有的灵活性使得大多数问题都有很多不同的解决之道。在开发项目期间，这种灵活性是可以接受的，但大家最终认识到，过于强调灵活性会导致大型项目难以维护：要通过研究代码搞清楚当时解决复杂问题的人是怎么想的，既困难又麻烦，还会耗费大量的时间。

经验丰富的程序员倡导尽可能避繁就简。Python社区的理念都包含在Tim Peters撰写的“Python之禅”中。要获悉这些有关编写优秀Python代码的指导原则，只需在解释器中执行命令`import this`。这里不打算赘述整个“Python之禅”，而只与大家分享其中的几条原则，让你明白为何它们对Python新手来说至关重要。

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

Python程序员笃信代码可以编写得漂亮而优雅。编程是要解决问题的，设计良好、高效而漂亮的解决方案都会让程序员心生敬意。随着你对Python的认识越来越深入，并使用它来编写越来越多的代码，有一天也许会有人站在你后面惊呼：“哇，代码编写得真是漂亮！”

2

Simple is better than complex.

如果有两个解决方案，一个简单，一个复杂，但都行之有效，就选择简单的解决方案吧。这样，你编写的代码将更容易维护，你或他人以后改进这些代码时也会更容易。

Complex is better than complicated.

现实是复杂的，有时候可能没有简单的解决方案。在这种情况下，就选择最简单可行的解决方案吧。

Readability counts.

即便是复杂的代码，也要让它易于理解。开发的项目涉及复杂代码时，一定要为这些代码编写有益的注释。

There should be one-- and preferably only one --obvious way to do it.

如果让两名Python程序员去解决同一个问题，他们提供的解决方案应大致相同。这并不是说编程没有创意空间，而是恰恰相反！然而，大部分编程工作都是使用常见解决方案来解决简单的小问题，但这些小问题都包含在更庞大、更有创意空间的项目中。在你的程序中，各种具体细节对其他Python程序员来说都应易于理解。

Now is better than never.

你可以将余生都用来学习Python和编程的纷繁难懂之处，但这样你什么项目都完不成。不要企图编写完美无缺的代码；先编写行之有效的代码，再决定是对其做进一步改进，还是转而去编写新代码。

等你进入下一章，开始研究更复杂的主题时，务必牢记这种简约而清晰的理念。如此，经验丰富的程序员定将对你编写的代码心生敬意，进而乐意向你提供反馈，并与你合作开发有趣的项目。

动手试一试

2-11 Python 之禅：在 Python 终端会话中执行命令 `import this`，并粗略地浏览一下其他的指导原则。

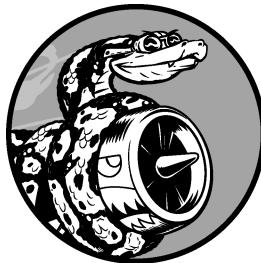
2.7 小结

在本章中，你学习了：如何使用变量；如何创建描述性变量名以及如何消除名称错误和语法错误；字符串是什么，以及如何使用小写、大写和首字母大写方式显示字符串；使用空白来显示整洁的输出，以及如何剔除字符串中多余的空白；如何使用整数和浮点数；使用数值数据时需要注意的意外行为。你还学习了如何编写说明性注释，让代码对你和其他人来说更容易理解。最后，你了解了让代码尽可能简单的理念。

在第3章，你将学习如何在被称为列表的变量中存储信息集，以及如何通过遍历列表来操作其中的信息。

列表简介

3



在本章和下一章中，你将学习列表是什么以及如何使用列表元素。列表让你能够在一个地方存储成组的信息，其中可以只包含几个元素，也可以包含数百万个元素。列表是新手可直接使用的最强大的Python功能之一，它融合了众多重要的编程概念。

8

3.1 列表是什么

列表由一系列按特定顺序排列的元素组成。你可以创建包含字母表中所有字母、数字0~9或所有家庭成员姓名的列表；也可以将任何东西加入列表中，其中的元素之间可以没有任何关系。鉴于列表通常包含多个元素，给列表指定一个表示复数的名称（如`letters`、`digits`或`names`）是个不错的主意。

在Python中，用方括号（`[]`）来表示列表，并用逗号来分隔其中的元素。下面是一个简单的列表示例，这个列表包含几种自行车：

`bicycles.py`

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

如果你让Python将列表打印出来，Python将打印列表的内部表示，包括方括号：

```
['trek', 'cannondale', 'redline', 'specialized']
```

鉴于这不是你要让用户看到的输出，下面来学习如何访问列表元素。

3.1.1 访问列表元素

列表是有序集合，因此要访问列表的任何元素，只需将该元素的位置或索引告诉Python即可。要访问列表元素，可指出列表的名称，再指出元素的索引，并将其放在方括号内。

例如，下面的代码从列表bicycles中提取第一款自行车：

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

❶处演示了访问列表元素的语法。当你请求获取列表元素时，Python只返回该元素，而不包括方括号和引号：

```
trek
```

这正是你要让用户看到的结果——整洁、干净的输出。

你还可以对任何列表元素调用第2章介绍的字符串方法。例如，可使用方法title()让元素'trek'的格式更整洁：

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

这个示例的输出与前一个示例相同，只是首字母T是大写的。

3.1.2 索引从0而不是1开始

在Python中，第一个列表元素的索引为0，而不是1。在大多数编程语言中都是如此，这与列表操作的底层实现相关。如果结果出乎意料，请看看你是否犯了简单的差一错误。

第二个列表元素的索引为1。根据这种简单的计数方式，要访问列表的任何元素，都可将其位置减1，并将结果作为索引。例如，要访问第四个列表元素，可使用索引3。

下面的代码访问索引1和3处的自行车：

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

这些代码返回列表中的第二个和第四个元素：

```
cannondale
specialized
```

Python为访问最后一个列表元素提供了一种特殊语法。通过将索引指定为-1，可让Python返回最后一个列表元素：

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
```

```
print(bicycles[-1])
```

这些代码返回'specialized'。这种语法很有用，因为你经常需要在不知道列表长度的情况下访问最后的元素。这种约定也适用于其他负数索引，例如，索引-2返回倒数第二个列表元素，索引-3返回倒数第三个列表元素，以此类推。

3.1.3 使用列表中的各个值

可像使用其他变量一样使用列表中的各个值。例如，你可以使用拼接根据列表中的值来创建消息。

下面来尝试从列表中提取第一款自行车，并使用这个值来创建一条消息：

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = "My first bicycle was a " + bicycles[0].title() + "."

print(message)
```

我们使用bicycles[0]的值生成了一个句子，并将其存储在变量message中（见❶）。输出是一个简单的句子，其中包含列表中的第一款自行车：

```
My first bicycle was a Trek.
```

8

动手试一试

请尝试编写一些简短的程序来完成下面的练习，以获得一些使用 Python 列表的第一手经验。你可能需要为每章的练习创建一个文件夹，以整洁有序的方式存储为完成各章的练习而编写的程序。

3-1 姓名：将一些朋友的姓名存储在一个列表中，并将其命名为 names。依次访问该列表中的每个元素，从而将每个朋友的姓名都打印出来。

3-2 问候语：继续使用练习 3-1 中的列表，但不打印每个朋友的姓名，而为每人打印一条消息。每条消息都包含相同的问候语，但抬头为相应朋友的姓名。

3-3 自己的列表：想想你喜欢的通勤方式，如骑摩托车或开汽车，并创建一个包含多种通勤方式的列表。根据该列表打印一系列有关这些通勤方式的宣言，如“I would like to own a Honda motorcycle”。

3.2 修改、添加和删除元素

你创建的大多数列表都将是动态的，这意味着列表创建后，将随着程序的运行增删元素。例

如，你创建一个游戏，要求玩家射杀从天而降的外星人；为此，可在开始时将一些外星人存储在列表中，然后每当有外星人被射杀时，都将其从列表中删除，而每次有新的外星人出现在屏幕上时，都将其添加到列表中。在整个游戏运行期间，外星人列表的长度将不断变化。

3.2.1 修改列表元素

修改列表元素的语法与访问列表元素的语法类似。要修改列表元素，可指定列表名和要修改的元素的索引，再指定该元素的新值。

例如，假设有一个摩托车列表，其中的第一个元素为'honda'，如何修改它的值呢？

motorcycles.py

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❷ motorcycles[0] = 'ducati'
print(motorcycles)
```

我们首先定义一个摩托车列表，其中的第一个元素为'honda'（见❶）。接下来，我们将第一个元素的值改为'ducati'（见❷）。输出表明，第一个元素的值确实变了，但其他列表元素的值没变：

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

你可以修改任何列表元素的值，而不仅仅是第一个元素的值。

3.2.2 在列表中添加元素

你可能出于众多原因要在列表中添加新元素，例如，你可能希望游戏中出现新的外星人、添加可视化数据或给网站添加新注册的用户。Python提供了多种在既有列表中添加新数据的方式。

1. 在列表末尾添加元素

在列表中添加新元素时，最简单的方式是将元素附加到列表末尾。给列表附加元素时，它将添加到列表末尾。继续使用前一个示例中的列表，在其末尾添加新元素'ducati'：

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ motorcycles.append('ducati')
print(motorcycles)
```

方法append()将元素'ducati'添加到了列表末尾（见❶），而不影响列表中的其他所有元素：

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

方法append()让动态地创建列表易如反掌，例如，你可以先创建一个空列表，再使用一系列的append()语句添加元素。下面来创建一个空列表，再在其中添加元素'honda'、'yamaha'和'suzuki'：

```
motorcycles = []
motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

最终的列表与前述示例中的列表完全相同：

```
['honda', 'yamaha', 'suzuki']
```

这种创建列表的方式极其常见，因为经常要等程序运行后，你才知道用户要在程序中存储哪些数据。为控制用户，可首先创建一个空列表，用于存储用户将要输入的值，然后将用户提供的每个新值附加到列表中。

8

2. 在列表中插入元素

使用方法insert()可在列表的任何位置添加新元素。为此，你需要指定新元素的索引和值。

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ motorcycles.insert(0, 'ducati')
print(motorcycles)
```

在这个示例中，值'ducati'被插入到了列表开头（见❶）；方法insert()在索引0处添加空间，并将值'ducati'存储到这个地方。这种操作将列表中既有的每个元素都右移一个位置：

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

3.2.3 从列表中删除元素

你经常需要从列表中删除一个或多个元素。例如，玩家将空中的一个外星人射杀后，你很可能要将其从存活的外星人列表中删除；当用户在你创建的Web应用中注销其账户时，你需要将该用户从活跃用户列表中删除。你可以根据位置或值来删除列表中的元素。

1. 使用del语句删除元素

如果知道要删除的元素在列表中的位置，可使用del语句。

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

```
❶ del motorcycles[0]
print(motorcycles)
```

❶ 处的代码使用`del`删除了列表`motorcycles`中的第一个元素——'honda'：

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

使用`del`可删除任何位置处的列表元素，条件是知道其索引。下例演示了如何删除前述列表中的第二个元素——'yamaha'：

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

下面的输出表明，已经将第二款摩托车从列表中删除了：

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

在这两个示例中，使用`del`语句将值从列表中删除后，你就无法再访问它了。

2. 使用方法`pop()`删除元素

有时候，你要将元素从列表中删除，并接着使用它的值。例如，你可能需要获取刚被射杀的外星人的`x`和`y`坐标，以便在相应的位置显示爆炸效果；在Web应用程序中，你可能要将用户从活跃成员列表中删除，并将其加入到非活跃成员列表中。

方法`pop()`可删除列表末尾的元素，并让你能够接着使用它。术语弹出（`pop`）源自这样的类比：列表就像一个栈，而删除列表末尾的元素相当于弹出栈顶元素。

下面从列表`motorcycles`中弹出一款摩托车：

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

我们首先定义并打印了列表`motorcycles`（见❶）。接下来，我们从这个列表中弹出一个值，并将其存储到变量`popped_motorcycle`中（见❷）。然后我们打印这个列表，以核实从其中删除了一个值（见❸）。最后，我们打印弹出的值，以证明我们依然能够访问被删除的值（见❹）。

输出表明，列表末尾的值'suzuki'已删除，它现在存储在变量`popped_motorcycle`中：

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

方法`pop()`是怎么起作用的呢？假设列表中的摩托车是按购买时间存储的，就可使用方法`pop()`打印一条消息，指出最后购买的是哪款摩托车：

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print("The last motorcycle I owned was a " + last_owned.title() + ".")
```

输出是一个简单的句子，指出了最新购买的是哪款摩托车：

```
The last motorcycle I owned was a Suzuki.
```

3. 弹出列表中任何位置处的元素

实际上，你可以使用`pop()`来删除列表中任何位置的元素，只需在括号中指定要删除的元素的索引即可。

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0)
❷ print('The first motorcycle I owned was a ' + first_owned.title() + '.')
```

首先，我们弹出了列表中的第一款摩托车（见❶），然后打印了一条有关这辆摩托车的消息（见❷）。输出是一个简单的句子，描述了我购买的第一辆摩托车：

```
The first motorcycle I owned was a Honda.
```

别忘了，每当你使用`pop()`时，被弹出的元素就不再在列表中了。

如果你不确定该使用`del`语句还是`pop()`方法，下面是一个简单的判断标准：如果你要从列表中删除一个元素，且不再以任何方式使用它，就使用`del`语句；如果你要在删除元素后还能继续使用它，就使用方法`pop()`。

4. 根据值删除元素

有时候，你不知道要从列表中删除的值所处的位置。如果你只知道要删除的元素的值，可使用方法`remove()`。

例如，假设我们要从列表`motorcycles`中删除值'ducati'。

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
```

```
❶ motorcycles.remove('ducati')
print(motorcycles)
```

❶ 处的代码让Python确定'ducati'出现在列表的什么地方，并将该元素删除：

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

使用`remove()`从列表中删除元素时，也可接着使用它的值。下面删除值'ducati'，并打印一条消息，指出要将其从列表中删除的原因：

```

❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
print(motorcycles)

❹ print("\nA " + too_expensive.title() + " is too expensive for me.")

```

在❶处定义列表后，我们将值'ducati'存储在变量`too_expensive`中（见❷）。接下来，我们使用这个变量来告诉Python将哪个值从列表中删除（见❸）。最后，值'ducati'已经从列表中删除，但它还存储在变量`too_expensive`中（见❹），让我们能够打印一条消息，指出将'ducati'从列表`motorcycles`中删除的原因：

```

['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

```

A Ducati is too expensive for me.

注意 方法`remove()`只删除第一个指定的值。如果要删除的值可能在列表中出现多次，就需要使用循环来判断是否删除了所有这样的值。你将在第7章学习如何这样做。

动手试一试

下面的练习比第2章的练习要复杂些，但让你有机会以前面介绍过的各种方式使用列表。

3-4 嘉宾名单：如果你可以邀请任何人一起共进晚餐（无论是在世的还是故去的），你会邀请哪些人？请创建一个列表，其中包含至少3个你想邀请的人；然后，使用这个列表打印消息，邀请这些人来与你共进晚餐。

3-5 修改嘉宾名单：你刚得知有位嘉宾无法赴约，因此需要另外邀请一位嘉宾。

□ 以完成练习3-4时编写的程序为基础，在程序末尾添加一条`print`语句，指出哪位嘉宾无法赴约。

□ 修改嘉宾名单，将无法赴约的嘉宾的姓名替换为新邀请的嘉宾的姓名。

□ 再次打印一系列消息，向名单中的每位嘉宾发出邀请。

3-6 添加嘉宾：你刚找到了一个更大的餐桌，可容纳更多的嘉宾。请想想你还想邀请哪三位嘉宾。

□ 以完成练习3-4或练习3-5时编写的程序为基础，在程序末尾添加一条`print`语句，指出你找到了一个更大的餐桌。

□ 使用`insert()`将一位新嘉宾添加到名单开头。

- 使用 `insert()` 将另一位新嘉宾添加到名单中间。
 - 使用 `append()` 将最后一位新嘉宾添加到名单末尾。
 - 打印一系列消息，向名单中的每位嘉宾发出邀请。
- 3-7 缩减名单：**你刚得知新购买的餐桌无法及时送达，因此只能邀请两位嘉宾。
- 以完成练习 3-6 时编写的程序为基础，在程序末尾添加一行代码，打印一条你只能邀请两位嘉宾共进晚餐的消息。
 - 使用 `pop()` 不断地删除名单中的嘉宾，直到只有两位嘉宾为止。每次从名单中弹出一位嘉宾时，都打印一条消息，让该嘉宾知悉你很抱歉，无法邀请他来共进晚餐。
 - 对于余下的两位嘉宾中的每一位，都打印一条消息，指出他依然在受邀人之列。
 - 使用 `del` 将最后两位嘉宾从名单中删除，让名单变成空的。打印该名单，核实程序结束时名单确实是空的。

8

3.3 组织列表

在你创建的列表中，元素的排列顺序常常是无法预测的，因为你并非总能控制用户提供数据的顺序。这虽然在大多数情况下都是不可避免的，但你经常需要以特定的顺序呈现信息。有时候，你希望保留列表元素最初的排列顺序，而有时候又需要调整排列顺序。Python 提供了很多组织列表的方式，可根据具体情况选用。

3.3.1 使用方法 `sort()` 对列表进行永久性排序

Python 方法 `sort()` 让你能够较为轻松地对列表进行排序。假设你有一个汽车列表，并要让其中的汽车按字母顺序排列。为简化这项任务，我们假设该列表中的所有值都是小写的。

`cars.py`

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
❶ cars.sort()
print(cars)
```

方法 `sort()`（见❶）永久性地修改了列表元素的排列顺序。现在，汽车是按字母顺序排列的，再也无法恢复到原来的排列顺序：

```
['audi', 'bmw', 'subaru', 'toyota']
```

你还可以按与字母顺序相反的顺序排列列表元素，为此，只需向 `sort()` 方法传递参数 `reverse=True`。下面的示例将汽车列表按与字母顺序相反的顺序排列：

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
```

```
cars.sort(reverse=True)
print(cars)
```

同样，对列表元素排列顺序的修改是永久性的：

```
['toyota', 'subaru', 'bmw', 'audi']
```

3.3.2 使用函数 sorted()对列表进行临时排序

要保留列表元素原来的排列顺序，同时以特定的顺序呈现它们，可使用函数sorted()。函数sorted()让你能够按特定顺序显示列表元素，同时不影响它们在列表中的原始排列顺序。

下面尝试对汽车列表调用这个函数。

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
```

- ❶ print("Here is the original list:")
 print(cars)
 - ❷ print("\nHere is the sorted list:")
 print(sorted(cars))
 - ❸ print("\nHere is the original list again:")
 print(cars)
-

我们首先按原始顺序打印列表（见❶），再按字母顺序显示该列表（见❷）。以特定顺序显示列表后，我们进行核实，确认列表元素的排列顺序与以前相同（见❸）。

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']
```

```
Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']
```

- ❶ Here is the original list again:
 ['bmw', 'audi', 'toyota', 'subaru']
-

注意，调用函数sorted()后，列表元素的排列顺序并没有变（见❶）。如果你要按与字母顺序相反的顺序显示列表，也可向函数sorted()传递参数reverse=True。

注意 在并非所有的值都是小写时，按字母顺序排列列表要复杂些。决定排列顺序时，有多种解读大写字母的方式，要指定准确的排列顺序，可能比我们这里所做的要复杂。然而，大多数排序方式都基于本节介绍的知识。

3.3.3 倒着打印列表

要反转列表元素的排列顺序，可使用方法reverse()。假设汽车列表是按购买时间排列的，可轻松地按相反的顺序排列其中的汽车：

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

注意，reverse()不是指按与字母顺序相反的顺序排列列表元素，而只是反转列表元素的排列顺序：

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

方法reverse()永久性地修改列表元素的排列顺序，但可随时恢复到原来的排列顺序，为此只需对列表再次调用reverse()即可。

8

3.3.4 确定列表的长度

使用函数len()可快速获悉列表的长度。在下面的示例中，列表包含4个元素，因此其长度为4：

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

在你需要完成如下任务时，len()很有用：确定还有多少个外星人未被射杀，需要管理多少项可视化数据，网站有多少注册用户等。

注意 Python计算列表元素数时从1开始，因此确定列表长度时，你应该不会遇到差一错误。

动手试一试

3-8 放眼世界：想出至少 5 个你渴望去旅游的地方。

- 将这些地方存储在一个列表中，并确保其中的元素不是按字母顺序排列的。
- 按原始排列顺序打印该列表。不要考虑输出是否整洁的问题，只管打印原始 Python 列表。
- 使用 sorted()按字母顺序打印这个列表，同时不要修改它。
- 再次打印该列表，核实排列顺序未变。
- 使用 sorted()按与字母顺序相反的顺序打印这个列表，同时不要修改它。

- 再次打印该列表，核实排列顺序未变。
- 使用 reverse()修改列表元素的排列顺序。打印该列表，核实排列顺序确实变了。
- 使用 reverse()再次修改列表元素的排列顺序。打印该列表，核实已恢复到原来的排列顺序。
- 使用 sort()修改该列表，使其元素按字母顺序排列。打印该列表，核实排列顺序确实变了。
- 使用 sort()修改该列表，使其元素按与字母顺序相反的顺序排列。打印该列表，核实排列顺序确实变了。

3-9 晚餐嘉宾：在完成练习 3-4~练习 3-7 时编写的程序之一中，使用 len()打印一条消息，指出你邀请了多少位嘉宾来与你共进晚餐。

3-10 尝试使用各个函数：想想可存储到列表中的东西，如山岳、河流、国家、城市、语言或你喜欢的任何东西。编写一个程序，在其中创建一个包含这些元素的列表，然后，对于本章介绍的每个函数，都至少使用一次来处理这个列表。

3.4 使用列表时避免索引错误

刚开始使用列表时，经常会遇到一种错误。假设你有一个包含三个元素的列表，却要求获取第四个元素：

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

这将导致索引错误：

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Python试图向你提供位于索引3处的元素，但它搜索列表motorcycles时，却发现索引3处没有元素。鉴于列表索引差一的特征，这种错误很常见。有些人从1开始数，因此以为第三个元素的索引为3；但在Python中，第三个元素的索引为2，因为索引是从0开始的。

索引错误意味着Python无法理解你指定的索引。程序发生索引错误时，请尝试将你指定的索引减1，然后再次运行程序，看看结果是否正确。

别忘了，每当需要访问最后一个列表元素时，都可使用索引-1。这在任何情况下都行之有效，即便你最后一次访问列表后，其长度发生了变化：

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

索引-1总是返回最后一个列表元素，这里为值'suzuki'：

```
'suzuki'
```

仅当列表为空时，这种访问最后一个元素的方式才会导致错误：

```
motorcycles = []
print(motorcycles[-1])
```

列表motorcycles不包含任何元素，因此Python返回一条索引错误消息：

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

注意 发生索引错误却找不到解决办法时，请尝试将列表或其长度打印出来。列表可能与你以为的截然不同，在程序对其进行动态处理时尤其如此。通过查看列表或其包含的元素数，可帮助你找出这种逻辑错误。

8

动手试一试

3-11 有意引发错误：如果你还没有在程序中遇到过索引错误，就尝试引发一个这种错误。在你的一个程序中，修改其中的索引，以引发索引错误。关闭程序前，务必消除这个错误。

3.5 小结

在本章中，你学习了：列表是什么以及如何使用其中的元素；如何定义列表以及如何增删元素；如何对列表进行永久性排序，以及如何为展示列表而进行临时排序；如何确定列表的长度，以及在使用列表时如何避免索引错误。

在第4章，你将学习如何以更高效的方式处理列表元素。通过使用为数不多的几行代码来遍历列表元素，你就能高效地处理它们，即便列表包含数千乃至数百万个元素。

第4章

操作列表

4



在第3章，你学习了如何创建简单的列表，还学习了如何操作列表元素。在本章中，你将学习如何遍历整个列表，这只需要几行代码，无论列表有多长。循环让你能够对列表的每个元素都采取一个或一系列相同的措施，从而高效地处理任何长度的列表，包括包含数千乃至数百万个元素的列表。

4.1 遍历整个列表

你经常需要遍历列表的所有元素，对每个元素执行相同的操作。例如，在游戏中，可能需要将每个界面元素平移相同的距离；对于包含数字的列表，可能需要对每个元素执行相同的统计运算；在网站中，可能需要显示文章列表中的每个标题。需要对列表中的每个元素都执行相同的操作时，可使用Python中的for循环。

假设我们有一个魔术师名单，需要将其中每个魔术师的名字都打印出来。为此，我们可以分别获取名单中的每个名字，但这种做法会导致多个问题。例如，如果名单很长，将包含大量重复的代码。另外，每当名单的长度发生变化时，都必须修改代码。通过使用for循环，可让Python去处理这些问题。

下面使用for循环来打印魔术师名单中的所有名字：

`magicians.py`

```
❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

首先，我们像第3章那样定义了一个列表（见❶）。接下来，我们定义了一个for循环（见❷）；

这行代码让Python从列表magicians中取出一个名字，并将其存储在变量magician中。最后，我们让Python打印前面存储到变量magician中的名字（见❸）。这样，对于列表中的每个名字，Python都将重复执行❷处和❸处的代码行。你可以这样解读这些代码：对于列表magicians中的每位魔术师，都将其名字打印出来。输出很简单，就是列表中所有的姓名：

```
alice
david
carolina
```

4.1.1 深入地研究循环

循环这种概念很重要，因为它是让计算机自动完成重复工作的常见方式之一。例如，在前面的magicians.py中使用的简单循环中，Python将首先读取其中的第一行代码：

```
for magician in magicians:
```

这行代码让Python获取列表magicians中的第一个值（'alice'），并将其存储到变量magician中。接下来，Python读取下一行代码：

```
    print(magician)
```

它让Python打印magician的值——依然是'alice'。鉴于该列表还包含其他值，Python返回到循环的第一行：

```
for magician in magicians:
```

Python获取列表中的下一个名字——'david'，并将其存储到变量magician中，再执行下面这行代码：

```
    print(magician)
```

Python再次打印变量magician的值——当前为'david'。接下来，Python再次执行整个循环，对列表中的最后一个值——'carolina'进行处理。至此，列表中没有其他的值了，因此Python接着执行程序的下一行代码。在这个示例中，for循环后面没有其他的代码，因此程序就此结束。

刚开始使用循环时请牢记，对列表中的每个元素，都将执行循环指定的步骤，而不管列表包含多少个元素。如果列表包含一百万个元素，Python就重复执行指定的步骤一百万次，且通常速度非常快。

另外，编写for循环时，对于用于存储列表中每个值的临时变量，可指定任何名称。然而，选择描述单个列表元素的有意义的名称大有帮助。例如，对于小猫列表、小狗列表和一般性列表，像下面这样编写for循环的第一行代码是不错的选择：

```
for cat in cats:  
for dog in dogs:  
for item in list_of_items:
```

这些命名约定有助于你明白for循环中将对每个元素执行的操作。使用单数和复数式名称，可帮助你判断代码段处理的是单个列表元素还是整个列表。

4.1.2 在for循环中执行更多的操作

在for循环中，可对每个元素执行任何操作。下面来扩展前面的示例，对于每位魔术师，都打印一条消息，指出他的表演太精彩了。

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
❶   print(magician.title() + ", that was a great trick!")
```

相比于前一个示例，唯一的不同是对于每位魔术师，都打印了一条以其名字为抬头的消息（见❶）。这个循环第一次迭代时，变量magician的值为'alice'，因此Python打印的第一条消息的抬头为'Alice'。第二次迭代时，消息的抬头为'David'，而第三次迭代时，抬头为'Carolina'。

下面的输出表明，对于列表中的每位魔术师，都打印了一条个性化消息：

```
Alice, that was a great trick!  
David, that was a great trick!  
Carolina, that was a great trick!
```

在for循环中，想包含多少行代码都可以。在代码行for magician in magicians后面，每个缩进的代码行都是循环的一部分，且将针对列表中的每个值都执行一次。因此，可对列表中的每个值执行任意次数的操作。

下面再添加一行代码，告诉每位魔术师，我们期待他的下一次表演：

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
    print(magician.title() + ", that was a great trick!")  
❶    print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

由于两条print语句都缩进了，因此它们都将针对列表中的每位魔术师执行一次。第二条print语句中的换行符"\n"（见❶）在每次迭代结束后都插入一个空行，从而整洁地将针对各位魔术师的消息编组：

```
Alice, that was a great trick!  
I can't wait to see your next trick, Alice.  
  
David, that was a great trick!  
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

在for循环中，想包含多少行代码都可以。实际上，你会发现使用for循环对每个元素执行众多不同的操作很有用。

4.1.3 在 for 循环结束后执行一些操作

4

for循环结束后再怎么做呢？通常，你需要提供总结性输出或接着执行程序必须完成的其他任务。

在for循环后面，没有缩进的代码都只执行一次，而不会重复执行。下面来打印一条向全体魔术师致谢的消息，感谢他们的精彩表演。想要在打印给各位魔术师的消息后面打印一条给全体魔术师的致谢消息，需要将相应的代码放在for循环后面，且不缩进：

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

❶ print("Thank you, everyone. That was a great magic show!")

你在前面看到了，开头两条print语句针对列表中每位魔术师重复执行。然而，由于第三条print语句没有缩进，因此只执行一次：

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

```
David, that was a great trick!
I can't wait to see your next trick, David.
```

```
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

```
Thank you, everyone. That was a great magic show!
```

使用for循环处理数据是一种对数据集执行整体操作的不错的方式。例如，你可能使用for循环来初始化游戏——遍历角色列表，将每个角色都显示到屏幕上；再在循环后面添加一个不缩进的代码块，在屏幕上绘制所有角色后显示一个Play Now按钮。

4.2 避免缩进错误

Python根据缩进来判断代码行与前一个代码行的关系。在前面的示例中，向各位魔术师显示消息的代码行是for循环的一部分，因为它们缩进了。Python通过使用缩进让代码更易读；简单地说，它要求你使用缩进让代码整洁而结构清晰。在较长的Python程序中，你将看到缩进程度各

不相同的代码块，这让你对程序的组织结构有大致的认识。

当你开始编写必须正确缩进的代码时，需要注意一些常见的缩进错误。例如，有时候，程序员会将不需要缩进的代码块缩进，而对于必须缩进的代码块却忘了缩进。通过查看这样的错误示例，有助于你以后避开它们，以及在它们出现在程序中时进行修复。

下面来看一些较为常见的缩进错误。

4.2.1 忘记缩进

对于位于for语句后面且属于循环组成部分的代码行，一定要缩进。如果你忘记缩进，Python会提醒你：

`magicians.py`

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

print语句（见❶）应缩进却没有缩进。Python没有找到期望缩进的代码块时，会让你知道哪行代码有问题。

```
File "magicians.py", line 3
    print(magician)
        ^
IndentationError: expected an indented block
```

通常，将紧跟在for语句后面的代码行缩进，可消除这种缩进错误。

4.2.2 忘记缩进额外的代码行

有时候，循环能够运行而不会报告错误，但结果可能会出乎意料。试图在循环中执行多项任务，却忘记缩进其中的一些代码行时，就会出现这种情况。

例如，如果忘记缩进循环中的第2行代码（它告诉每位魔术师，我们期待他的下一次表演），就会出现这种情况：

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
❶ print("I can't wait to see your next trick, " + magician.title() + ".\n")
```

第二条print语句（见❶）原本需要缩进，但Python发现for语句后面有一行代码是缩进的，因此它没有报告错误。最终的结果是，对于列表中的每位魔术师，都执行了第一条print语句，因为它缩进了；而第二条print语句没有缩进，因此它只在循环结束后执行一次。由于变量magician的终值为'carolina'，因此只有她收到了消息“looking forward to the next trick”：

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

这是一个逻辑错误。从语法上看，这些Python代码是合法的，但由于存在逻辑错误，结果并不符合预期。如果你预期某项操作将针对每个列表元素都执行一次，但它却只执行了一次，请确定是否需要将一行或多行代码缩进。

4

4.2.3 不必要的缩进

如果你不小心缩进了无需缩进的代码行，Python将指出这一点：

`hello_world.py`

```
message = "Hello Python world!"
❶   print(message)
```

`print`语句（见❶）无需缩进，因为它并不属于前一行代码，因此Python将指出这种错误：

```
File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

为避免意外缩进错误，请只缩进需要缩进的代码。在前面编写的程序中，只有要在`for`循环中对每个元素执行的代码需要缩进。

4.2.4 循环后不必要的缩进

如果你不小心缩进了应在循环结束后执行的代码，这些代码将针对每个列表元素重复执行。在有些情况下，这可能导致Python报告语法错误，但在大多数情况下，这只会导致逻辑错误。

例如，如果不小心缩进了感谢全体魔术师精彩表演的代码行，结果将如何呢？

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician.title() + ", that was a great trick!")
    print("I can't wait to see your next trick, " + magician.title() + ".\n")
❶    print("Thank you everyone, that was a great magic show!")
```

由于❶处的代码行被缩进，它将针对列表中的每位魔术师执行一次，如❷所示：

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.
```

❷ `Thank you everyone, that was a great magic show!`

David, that was a great trick!
I can't wait to see your next trick, David.

- ❷ Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
- ❸ Thank you everyone, that was a great magic show!

这也是一个逻辑错误，与4.2.2节的错误类似。Python不知道你的本意，只要代码符合语法，它就会运行。如果原本只应执行一次的操作执行了多次，请确定你是否不应该缩进执行该操作的代码。

4.2.5 遗漏了冒号

for语句末尾的冒号告诉Python，下一行是循环的第一行。

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

如果你不小心遗漏了冒号，如❶所示，将导致语法错误，因为Python不知道你意欲何为。这种错误虽然易于消除，但并不那么容易发现。程序员为找出这样的单字符错误，花费的时间多得令人惊讶。这样的错误之所以难以发现，是因为通常在我们的意料之外。

动手试一试

4-1 比萨：想出至少三种你喜欢的比萨，将其名称存储在一个列表中，再使用 for 循环将每种比萨的名称都打印出来。

- 修改这个 for 循环，使其打印包含比萨名称的句子，而不仅仅是比萨的名称。对于每种比萨，都显示一行输出，如 “I like pepperoni pizza”。
- 在程序末尾添加一行代码，它不在 for 循环中，指出你有多喜欢比萨。输出应包含针对每种比萨的消息，还有一个总结性句子，如 “I really love pizza!”。

4-2 动物：想出至少三种有共同特征的动物，将这些动物的名称存储在一个列表中，再使用 for 循环将每种动物的名称都打印出来。

- 修改这个程序，使其针对每种动物都打印一个句子，如 “A dog would make a great pet”。
- 在程序末尾添加一行代码，指出这些动物的共同之处，如打印诸如 “Any of these animals would make a great pet!” 这样的句子。

4.3 创建数值列表

需要存储一组数字的原因有很多，例如，在游戏中，需要跟踪每个角色的位置，还可能需要跟踪玩家的几个最高得分。在数据可视化中，处理的几乎都是由数字（如温度、距离、人口数量、经度和纬度等）组成的集合。

列表非常适合用于存储数字集合，而Python提供了很多工具，可帮助你高效地处理数字列表。明白如何有效地使用这些工具后，即便列表包含数百万个元素，你编写的代码也能运行得很好。

4.3.1 使用函数 range()

Python函数range()让你能够轻松地生成一系列的数字。例如，可以像下面这样使用函数range()来打印一系列的数字：

numbers.py

```
for value in range(1,5):
    print(value)
```

上述代码好像应该打印数字1~5，但实际上它不会打印数字5：

1
2
3
4

在这个示例中，range()只是打印数字1~4，这是你在编程语言中经常看到的差一行为的结果。函数range()让Python从你指定的第一个值开始数，并在到达你指定的第二个值后停止，因此输出不包含第二个值（这里为5）。

要打印数字1~5，需要使用range(1,6)：

```
for value in range(1,6):
    print(value)
```

这样，输出将从1开始，到5结束：

1
2
3
4
5

使用range()时，如果输出不符合预期，请尝试将指定的值加1或减1。

4.3.2 使用 range()创建数字列表

要创建数字列表，可使用函数list()将range()的结果直接转换为列表。如果将range()作为

list() 的参数，输出将为一个数字列表。

在前一节的示例中，我们打印了一系列数字。要将这些数字转换为一个列表，可使用 list()：

```
numbers = list(range(1,6))
print(numbers)
```

结果如下：

```
[1, 2, 3, 4, 5]
```

使用函数 range() 时，还可指定步长。例如，下面的代码打印 1~10 内的偶数：

even_numbers.py

```
even_numbers = list(range(2,11,2))
print(even_numbers)
```

在这个示例中，函数 range() 从 2 开始数，然后不断地加 2，直到达到或超过终值（11），因此输出如下：

```
[2, 4, 6, 8, 10]
```

使用函数 range() 几乎能够创建任何需要的数字集，例如，如何创建一个列表，其中包含前 10 个整数（即 1~10）的平方呢？在 Python 中，两个星号（**）表示乘方运算。下面的代码演示了如何将前 10 个整数的平方加入到一个列表中：

squares.py

```
❶ squares = []
❷ for value in range(1,11):
❸     square = value**2
❹     squares.append(square)

❺ print(squares)
```

首先，我们创建了一个空列表（见❶）；接下来，使用函数 range() 让 Python 遍历 1~10 的值（见❷）。在循环中，计算当前值的平方，并将结果存储到变量 square 中（见❸）。然后，将新计算得到的平方值附加到列表 squares 末尾（见❹）。最后，循环结束后，打印列表 squares（见❺）：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

为让这些代码更简洁，可不使用临时变量 square，而直接将每个计算得到的值附加到列表末尾：

```
squares = []
for value in range(1,11):
❶     squares.append(value**2)
```

```
print(squares)
```

❶处的代码与squares.py中❸处和❹处的代码等效。在循环中，计算每个值的平方，并立即将结果附加到列表squares的末尾。

创建更复杂的列表时，可使用上述两种方法中的任何一种。有时候，使用临时变量会让代码更易读；而在其他情况下，这样做只会让代码无谓地变长。你首先应该考虑的是，编写清晰易懂且能完成所需功能的代码；等到审核代码时，再考虑采用更高效的方法。

4

4.3.3 对数字列表执行简单的统计计算

有几个专门用于处理数字列表的Python函数。例如，你可以轻松地找出数字列表的最大值、最小值和总和：

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

注意 出于版面考虑，本节使用的数字列表都很短，但这里介绍的知识也适用于包含数百万个数字的列表。

4.3.4 列表解析

前面介绍的生成列表squares的方式包含三四行代码，而列表解析让你只需编写一行代码就能生成这样的列表。列表解析将for循环和创建新元素的代码合并成一行，并自动附加新元素。面向初学者的书籍并非都会介绍列表解析，这里之所以介绍列表解析，是因为等你开始阅读他人编写的代码时，很可能会遇到它们。

下面的示例使用列表解析创建你在前面看到的平方数列表：

squares.py

```
squares = [value**2 for value in range(1,11)]
print(squares)
```

要使用这种语法，首先指定一个描述性的列表名，如squares；然后，指定一个左方括号，并定义一个表达式，用于生成你要存储到列表中的值。在这个示例中，表达式为value**2，它计算平方值。接下来，编写一个for循环，用于给表达式提供值，再加上右方括号。在这个示例中，

for循环为`for value in range(1,11)`，它将值1~10提供给表达式`value**2`。请注意，这里的for语句末尾没有冒号。

结果与你在前面看到的平方数列表相同：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

要创建自己的列表解析，需要经过一定的练习，但能够熟练地创建常规列表后，你会发现这样做是完全值得的。当你觉得编写三四行代码来生成列表有点繁复时，就应考虑创建列表解析了。

动手试一试

4-3 数到 20：使用一个 for 循环打印数字 1~20（含）。

4-4 一百万：创建一个列表，其中包含数字 1~1 000 000，再使用一个 for 循环将这些数字打印出来（如果输出的时间太长，按 Ctrl + C 停止输出，或关闭输出窗口）。

4-5 计算 1~1 000 000 的总和：创建一个列表，其中包含数字 1~1 000 000，再使用`min()`和`max()`核实该列表确实是从 1 开始，到 1 000 000 结束的。另外，对这个列表调用函数`sum()`，看看 Python 将一百万个数字相加需要多长时间。

4-6 奇数：通过给函数`range()`指定第三个参数来创建一个列表，其中包含 1~20 的奇数；再使用一个 for 循环将这些数字都打印出来。

4-7 3 的倍数：创建一个列表，其中包含 3~30 内能被 3 整除的数字；再使用一个 for 循环将这个列表中的数字都打印出来。

4-8 立方：将同一个数字乘三次称为立方。例如，在 Python 中，2 的立方用`2**3`表示。请创建一个列表，其中包含前 10 个整数（即 1~10）的立方，再使用一个 for 循环将这些立方数都打印出来。

4-9 立方解析：使用列表解析生成一个列表，其中包含前 10 个整数的立方。

4.4 使用列表的一部分

在第3章中，你学习了如何访问单个列表元素。在本章中，你一直在学习如何处理列表的所有元素。你还可以处理列表的部分元素——Python称之为切片。

4.4.1 切片

要创建切片，可指定要使用的一个元素和最后一个元素的索引。与函数`range()`一样，Python 在到达你指定的第二个索引前面的元素后停止。要输出列表中的前三个元素，需要指定索引`0~3`，这将输出分别为0、1和2的元素。

下面的示例处理的是一个运动队成员列表：

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

❶处的代码打印该列表的一个切片，其中只包含三名队员。输出也是一个列表，其中包含前三名队员：

```
['charles', 'martina', 'michael']
```

你可以生成列表的任何子集，例如，如果你要提取列表的第2~4个元素，可将起始索引指定为1，并将终止索引指定为4：

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

这一次，切片始于'marita'，终于'florence'：

```
['martina', 'michael', 'florence']
```

如果你没有指定第一个索引，Python将自动从列表开头开始：

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

由于没有指定起始索引，Python从列表开头开始提取：

```
['charles', 'martina', 'michael', 'florence']
```

要让切片终止于列表末尾，也可使用类似的语法。例如，如果要提取从第3个元素到列表末尾的所有元素，可将起始索引指定为2，并省略终止索引：

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python将返回从第3个元素到列表末尾的所有元素：

```
['michael', 'florence', 'eli']
```

无论列表多长，这种语法都能够让你输出从特定位置到列表末尾的所有元素。本书前面说过，负数索引返回离列表末尾相应距离的元素，因此你可以输出列表末尾的任何切片。例如，如果你要输出名单上的最后三名队员，可使用切片players[-3:]：

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

上述代码打印最后三名队员的名字，即便队员名单的长度发生变化，也依然如此。

4.4.2 遍历切片

如果要遍历列表的部分元素，可在for循环中使用切片。在下面的示例中，我们遍历前三名队员，并打印他们的名字：

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

❶处的代码没有遍历整个队员列表，而只遍历前三名队员：

```
Here are the first three players on my team:
Charles
Martina
Michael
```

在很多情况下，切片都很有用。例如，编写游戏时，你可以在玩家退出游戏时将其最终得分加入到一个列表中。然后，为获取该玩家的三个最高得分，你可以将该列表按降序排列，再创建一个只包含前三个得分的切片。处理数据时，可使用切片来进行批量处理；编写Web应用程序时，可使用切片来分页显示信息，并在每页显示数量合适的信息。

4.4.3 复制列表

你经常需要根据既有列表创建全新的列表。下面来介绍复制列表的工作原理，以及复制列表可提供极大帮助的一种情形。

要复制列表，可创建一个包含整个列表的切片，方法是同时省略起始索引和终止索引（[:]）。这让Python创建一个始于第一个元素，终止于最后一个元素的切片，即复制整个列表。

例如，假设有一个列表，其中包含你最喜欢的四种食品，而你还想创建另一个列表，在其中包含一位朋友喜欢的所有食品。不过，你喜欢的食品，这位朋友都喜欢，因此你可以通过复制来创建这个列表：

foods.py

```
❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

我们首先创建了一个名为my_foods的食品列表（见❶），然后创建了一个名为friend_foods的新列表（见❷）。我们在不指定任何索引的情况下从列表my_foods中提取一个切片，从而创建了这个列表的副本，再将该副本存储到变量friend_foods中。打印每个列表后，我们发现它们包含

的食品相同：

```
My favorite foods are:  
['pizza', 'falafel', 'carrot cake']
```

```
My friend's favorite foods are:  
['pizza', 'falafel', 'carrot cake']
```

为核实我们确实有两个列表，下面在每个列表中都添加一种食品，并核实每个列表都记录了相应人员喜欢的食品：

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
❶ friend_foods = my_foods[:]  
  
❷ my_foods.append('cannoli')  
❸ friend_foods.append('ice cream')  
  
print("My favorite foods are:")  
print(my_foods)  
  
print("\nMy friend's favorite foods are:")  
print(friend_foods)
```

与前一个示例一样，我们首先将my_foods的元素复制到新列表friend_foods中（见❶）。接下来，在每个列表中都添加一种食品：在列表my_foods中添加'cannoli'（见❷），而在friend_foods中添加'ice cream'（见❸）。最后，打印这两个列表，核实这两种食品包含在正确的列表中。

```
My favorite foods are:  
❶ ['pizza', 'falafel', 'carrot cake', 'cannoli']  
  
My friend's favorite foods are:  
❷ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

❶处的输出表明，'cannoli'包含在你喜欢的食品列表中，而'ice cream'没有。❷处的输出表明，'ice cream'包含在你朋友喜欢的食品列表中，而'cannoli'没有。倘若我们只是简单地将my_foods赋给friend_foods，就不能得到两个列表。例如，下例演示了在不使用切片的情况下复制列表的情况：

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
  
❶ friend_foods = my_foods  
  
my_foods.append('cannoli')  
friend_foods.append('ice cream')  
  
print("My favorite foods are:")  
print(my_foods)  
  
print("\nMy friend's favorite foods are:")  
print(friend_foods)
```

这里将my_foods赋给friend_foods，而不是将my_foods的副本存储到friend_foods（见❶）。这种语法实际上是让Python将新变量friend_foods关联到包含在my_foods中的列表，因此这两个变量都指向同一个列表。鉴于此，当我们将'cannoli'添加到my_foods中时，它也将出现在friend_foods中；同样，虽然'ice cream'好像只被加入到了friend_foods中，但它也将出现在这两个列表中。

输出表明，两个列表是相同的，这并非我们想要的结果：

```
My favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

注意 现在暂时不要考虑这个示例中的细节。基本上，当你试图使用列表的副本时，如果结果出乎意料，请确认你像第一个示例那样使用切片复制了列表。

动手试一试

4-10 切片：选择你在本章编写的一个程序，在末尾添加几行代码，以完成如下任务。

- ❑ 打印消息“The first three items in the list are:”，再使用切片来打印列表的前三个元素。
- ❑ 打印消息“Three items from the middle of the list are:”，再使用切片来打印列表中间的三个元素。
- ❑ 打印消息“The last three items in the list are:”，再使用切片来打印列表末尾的三个元素。

4-11 你的比萨和我的比萨：在你为完成练习4-1而编写的程序中，创建比萨列表的副本，并将其存储到变量friend_pizzas中，再完成如下任务。

- ❑ 在原来的比萨列表中添加一种比萨。
- ❑ 在列表friend_pizzas中添加另一种比萨。
- ❑ 核实你有两个不同的列表。为此，打印消息“My favorite pizzas are:”，再使用一个for循环来打印第一个列表；打印消息“My friend's favorite pizzas are:”，再使用一个for循环来打印第二个列表。核实新增的比萨被添加到了正确的列表中。

4-12 使用多个循环：在本节中，为节省篇幅，程序foods.py的每个版本都没有使用for循环来打印列表。请选择一个版本的foods.py，在其中编写两个for循环，将各个食品列表都打印出来。

4.5 元组

列表非常适合用于存储在程序运行期间可能变化的数据集。列表是可以修改的，这对处理网站的用户列表或游戏中的角色列表至关重要。然而，有时候你需要创建一系列不可修改的元素，元组可以满足这种需求。Python将不能修改的值称为不可变的，而不可变的列表被称为元组。

4.5.1 定义元组

元组看起来犹如列表，但使用圆括号而不是方括号来标识。定义元组后，就可以使用索引来访问其元素，就像访问列表元素一样。

例如，如果有一个大小不应改变的矩形，可将其长度和宽度存储在一个元组中，从而确保它们是不能修改的：

`dimensions.py`

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
    print(dimensions[1])
```

我们首先定义了元组`dimensions`（见❶），为此我们使用了圆括号而不是方括号。接下来，我们分别打印该元组的各个元素，使用的语法与访问列表元素时使用的语法相同（见❷）：

```
200
50
```

下面来尝试修改元组`dimensions`中的一个元素，看看结果如何：

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

❶处的代码试图修改第一个元素的值，导致Python返回类型错误消息。由于试图修改元组的操作是被禁止的，因此Python指出不能给元组的元素赋值：

```
Traceback (most recent call last):
  File "dimensions.py", line 3, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

代码试图修改矩形的尺寸时，Python报告错误，这很好，因为这正是我们希望的。

4.5.2 遍历元组中的所有值

像列表一样，也可以使用for循环来遍历元组中的所有值：

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

就像遍历列表时一样，Python返回元组中所有的元素，：

```
200  
50
```

4.5.3 修改元组变量

虽然不能修改元组的元素，但可以给存储元组的变量赋值。因此，如果要修改前述矩形的尺寸，可重新定义整个元组：

```
❶ dimensions = (200, 50)  
print("Original dimensions:")  
for dimension in dimensions:  
    print(dimension)  
  
❷ dimensions = (400, 100)  
❸ print("\nModified dimensions:")  
for dimension in dimensions:  
    print(dimension)
```

我们首先定义了一个元组，并将其存储的尺寸打印了出来（见❶）；接下来，将一个新元组存储到变量dimensions中（见❷）；然后，打印新的尺寸（见❸）。这次，Python不会报告任何错误，因为给元组变量赋值是合法的：

```
Original dimensions:  
200  
50  
  
Modified dimensions:  
400  
100
```

相比于列表，元组是更简单的数据结构。如果需要存储的一组值在程序的整个生命周期内都不变，可使用元组。

动手试一试

4-13 自助餐：有一家自助式餐馆，只提供五种简单的食品。请想出五种简单的食品，并将其存储在一个元组中。

- 使用一个for循环将该餐馆提供的五种食品都打印出来。
- 尝试修改其中的一个元素，核实Python确实会拒绝你这样做。
- 餐馆调整了菜单，替换了它提供的其中两种食品。请编写一个这样的代码块：给元组变量赋值，并使用一个for循环将新元组的每个元素都打印出来。

4.6 设置代码格式

随着你编写的程序越来越长，有必要了解一些代码格式设置约定。请花时间让你的代码尽可能易于阅读；让代码易于阅读有助于你掌握程序是做什么的，也可以帮助他人理解你编写的代码。

为确保所有人编写的代码的结构都大致一致，Python程序员都遵循一些格式设置约定。学会编写整洁的Python后，就能明白他人编写的Python代码的整体结构——只要他们和你遵循相同的指南。要成为专业程序员，应从现在开始就遵循这些指南，以养成良好的习惯。

4.6.1 格式设置指南

若要提出Python语言修改建议，需要编写Python改进提案（Python Enhancement Proposal，PEP）。PEP 8是最古老的PEP之一，它向Python程序员提供了代码格式设置指南。PEP 8的篇幅很长，但大都与复杂的编码结构相关。

Python格式设置指南的编写者深知，代码被阅读的次数比编写的次数多。代码编写出来后，调试时你需要阅读它；给程序添加新功能时，需要花很长的时间阅读代码；与其他程序员分享代码时，这些程序员也将阅读它们。

如果一定要在让代码易于编写和易于阅读之间做出选择，Python程序员几乎总是会选择后者。下面的指南可帮助你从一开始就编写出清晰的代码。

4.6.2 缩进

PEP 8建议每级缩进都使用四个空格，这既可提高可读性，又留下了足够的多级缩进空间。

在字处理文档中，大家常常使用制表符而不是空格来缩进。对于字处理文档来说，这样做的效果很好，但混合使用制表符和空格会让Python解释器感到迷惑。每款文本编辑器都提供了一种设置，可将输入的制表符转换为指定数量的空格。你在编写代码时应该使用制表符键，但一定要对编辑器进行设置，使其在文档中插入空格而不是制表符。

在程序中混合使用制表符和空格可能导致极难解决的问题。如果你混合使用了制表符和空格，可将文件中所有的制表符转换为空格，大多数编辑器都提供了这样的功能。

4.6.3 行长

很多Python程序员都建议每行不超过80字符。最初制定这样的指南时，在大多数计算机中，终端窗口每行只能容纳79字符；当前，计算机屏幕每行可容纳的字符数多得多，为何还要使用79字符的标准行长呢？这里有别的原因。专业程序员通常会在同一个屏幕上打开多个文件，使用标准行长可以让他们在屏幕上并排打开两三个文件时能同时看到各个文件的完整行。PEP 8还建议注释的行长都不超过72字符，因为有些工具为大型项目自动生成文档时，会在每行注释开头添加格式化字符。

PEP 8中有关行长的指南并非不可逾越的红线，有些小组将最大行长设置为99字符。在学习

期间，你不用过多地考虑代码的行长，但别忘了，协作编写程序时，大家几乎都遵守PEP 8指南。在大多数编辑器中，都可设置一个视觉标志——通常是一条竖线，让你知道不能越过的界线在什么地方。

注意 附录B介绍了如何配置文本编辑器，以使其：在你按制表符键时插入四个空格；显示一条垂直参考线，帮助你遵守行长不能超过79字符的约定。

4.6.4 空行

要将程序的不同部分分开，可使用空行。你应该使用空行来组织程序文件，但也不能滥用；只要按本书的示例展示的那样做，就能掌握其中的平衡。例如，如果你有5行创建列表的代码，还有3行处理该列表的代码，那么用一个空行将这两部分隔开是合适的。然而，你不应使用三四个空行将它们隔开。

空行不会影响代码的运行，但会影响代码的可读性。Python解释器根据水平缩进情况来解读代码，但不关心垂直间距。

4.6.5 其他格式设置指南

PEP 8还有很多其他的格式设置建议，但这些指南针对的程序大都比目前为止本书提到的程序复杂。等介绍更复杂的Python结构时，我们再来分享相关的PEP 8指南。

动手试一试

4-14 PEP 8：请访问 <https://python.org/dev/peps/pep-0008/>，阅读PEP 8格式设置指南。当前，这些指南适用的不多，但你可以大致浏览一下。

4-15 代码审核：从本章编写的程序中选择三个，根据PEP 8指南对它们进行修改。

- 每级缩进都使用四个空格。对你使用的文本编辑器进行设置，使其在你按 Tab 键时都插入四个空格；如果你还没有这样做，现在就去做吧（有关如何设置，请参阅附录B）。
- 每行都不要超过80字符。对你使用的编辑器进行设置，使其在第80个字符处显示一条垂直参考线。
- 不要在程序文件中过多地使用空行。

4.7 小结

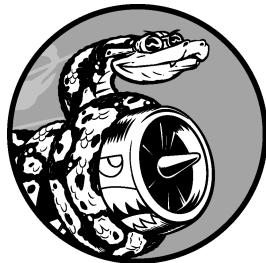
在本章中，你学习了：如何高效地处理列表中的元素；如何使用for循环遍历列表，Python如何根据缩进来确定程序的结构以及如何避免一些常见的缩进错误；如何创建简单的数字列表，以及可对数字列表执行的一些操作；如何通过切片来使用列表的一部分和复制列表。你还学习了元组（它对不应变化的值提供了一定程度的保护），以及在代码变得越来越复杂时如何设置格式，使其易于阅读。

在第5章中，你将学习如何使用if语句在不同的条件下采取不同的措施；学习如何将一组较复杂的条件测试组合起来，并在满足特定条件时采取相应的措施。你还将学习如何在遍历列表时，通过使用if语句对特定元素采取特定的措施。

第5章

if语句

5



编程时经常需要检查一系列条件，并据此决定采取什么措施。在Python中，if语句让你能够检查程序的当前状态，并据此采取相应的措施。

在本章中，你将学习条件测试，以检查感兴趣的任何条件。你将学习简单的if语句，以及创建一系列复杂的if语句来确定当前到底处于什么情形。接下来，你将把学到的知识应用于列表，以编写for循环，以一种方式处理列表中的大多数元素，并以另一种不同的方式处理包含特定值的元素。

5.1 一个简单示例

下面是一个简短的示例，演示了如何使用if语句来正确地处理特殊情形。假设你有一个汽车列表，并想将其中每辆汽车的名称打印出来。对于大多数汽车，都应以首字母大写的方式打印其名称，但对于汽车名'bmw'，应以全大写的方式打印。下面的代码遍历一个列表，并以首字母大写的方式打印其中的汽车名，但对于汽车名'bmw'，以全大写的方式打印：

`cars.py`

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

这个示例中的循环首先检查当前的汽车名是否是'bmw'（见❶）。如果是，就以全大写的方式打印它；否则就以首字母大写的方式打印：

```
Audi
BMW
Subaru
Toyota
```

这个示例涵盖了本章将介绍的很多概念。下面先来介绍可用来在程序中检查条件的测试。

5.2 条件测试

每条if语句的核心都是一个值为True或False的表达式，这种表达式被称为条件测试。Python根据条件测试的值为True还是False来决定是否执行if语句中的代码。如果条件测试的值为True，Python就执行紧跟在if语句后面的代码；如果为False，Python就忽略这些代码。

5

5.2.1 检查是否相等

大多数条件测试都将一个变量的当前值同特定值进行比较。最简单的条件测试检查变量的值是否与特定值相等：

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
True
```

我们首先使用一个等号将car的值设置为'bmw'（见❶），这种做法你已见过很多次。接下来，使用两个等号（==）检查car的值是否为'bmw'。这个相等运算符在它两边的值相等时返回True，否则返回False。在这个示例中，两边的值相等，因此Python返回True。

如果变量car的值不是'bmw'，上述测试将返回False：

```
❶ >>> car = 'audi'
❷ >>> car == 'bmw'
False
```

一个等号是陈述；对于❶处的代码，可解读为“将变量car的值设置为'audi'"。两个等号是发问；对于❷处的代码，可解读为“变量car的值是'bmw'吗？”大多数编程语言使用等号的方式都与这里演示的相同。

5.2.2 检查是否相等时不考虑大小写

在Python中检查是否相等时区分大小写，例如，两个大小写不同的值会被视为不相等：

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

如果大小写很重要，这种行为有其优点。但如果大小写无关紧要，而只想检查变量的值，可将变量的值转换为小写，再进行比较：

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

无论值'Audi'的大小写如何，上述测试都将返回True，因为该测试不区分大小写。函数lower()不会修改存储在变量car中的值，因此进行这样的比较时不会影响原来的变量：

```
❶ >>> car = 'Audi'
❷ >>> car.lower() == 'audi'
True
❸ >>> car
'Audi'
```

在❶处，我们将首字母大写的字符串'Audi'存储在变量car中；在❷处，我们获取变量car的值并将其转换为小写，再将结果与字符串'audi'进行比较。这两个字符串相同，因此Python返回True。从❸处的输出可知，这个条件测试并没有影响存储在变量car中的值。

网站采用类似的方式让用户输入的数据符合特定的格式。例如，网站可能使用类似的测试来确保用户名是独一无二的，而并非只是与另一个用户名的大小写不同。用户提交新的用户名时，将把它转换为小写，并与所有既有用户名的小写版本进行比较。执行这种检查时，如果有用户名'john'（不管大小写如何），则用户提交用户名'John'时将遭到拒绝。

5.2.3 检查是否不相等

要判断两个值是否不等，可结合使用惊叹号和等号（!=），其中的惊叹号表示不，在很多编程语言中都如此。

下面再使用一条if语句来演示如何使用不等运算符。我们将把要求的比萨配料存储在一个变量中，再打印一条消息，指出顾客要求的配料是否是意式小银鱼（anchovies）：

toppings.py

```
requested_topping = 'mushrooms'

❶ if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

❶处的代码行将requested_topping的值与'anchovies'进行比较，如果它们不相等，Python将返回True，进而执行紧跟在if语句后面的代码；如果这两个值相等，Python将返回False，因此不执行紧跟在if语句后面的代码。

由于requested_topping的值不是'anchovies'，因此执行print语句：

Hold the anchovies!

你编写的大多数条件表达式都检查两个值是否相等，但有时候检查两个值是否不等的效率更高。

5.2.4 比较数字

检查数值非常简单，例如，下面的代码检查一个人是否是18岁：

```
>>> age = 18
>>> age == 18
True
```

5

你还可以检查两个数字是否不等，例如，下面的代码在提供的答案不正确时打印一条消息：

`magic_number.py`

`answer = 17`

❶ `if answer != 42:`
 `print("That is not the correct answer. Please try again!")`

`answer (17)` 不是42，❶处的条件得到满足，因此缩进的代码块得以执行：

`That is not the correct answer. Please try again!`

条件语句中可包含各种数学比较，如小于、小于等于、大于、大于等于：

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

在if语句中可使用各种数学比较，这让你能够直接检查关心的条件。

5.2.5 检查多个条件

你可能想同时检查多个条件，例如，有时候你需要在两个条件都为True时才执行相应的操作，而有时候你只要求一个条件为True时就执行相应的操作。在这些情况下，关键字and和or可助你一臂之力。

1. 使用and检查多个条件

要检查是否两个条件都为True，可使用关键字and将两个条件测试合而为一；如果每个测试

都通过了，整个表达式就为True；如果至少有一个测试没有通过，整个表达式就为False。

例如，要检查是否两个人都不小于21岁，可使用下面的测试：

```

❶ >>> age_0 = 22
>>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
False
❸ >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True

```

在❶处，我们定义了两个用于存储年龄的变量：age_0和age_1。在❷处，我们检查这两个变量是否都大于或等于21；左边的测试通过了，但右边的测试没有通过，因此整个条件表达式的结果为False。在❸处，我们将age_1改为22，这样age_1的值大于21，因此两个测试都通过了，导致整个条件表达式的结果为True。

为改善可读性，可将每个测试都分别放在一对括号内，但并非必须这样做。如果你使用括号，测试将类似于下面这样：

```
(age_0 >= 21) and (age_1 >= 21)
```

2. 使用or检查多个条件

关键字or也能够让你检查多个条件，但只要至少有一个条件满足，就能通过整个测试。仅当两个测试都没有通过时，使用or的表达式才为False。

下面再次检查两个人的年龄，但检查的条件是至少有一个人的年龄不小于21岁：

```

❶ >>> age_0 = 22
>>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
True
❸ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False

```

同样，我们首先定义了两个用于存储年龄的变量（见❶）。由于❷处对age_0的测试通过了，因此整个表达式的结果为True。接下来，我们将age_0减小为18；在❸处的测试中，两个测试都没有通过，因此整个表达式的结果为False。

5.2.6 检查特定值是否包含在列表中

有时候，执行操作前必须检查列表是否包含特定的值。例如，结束用户的注册过程前，可能需要检查他提供的用户名是否已包含在用户名列表中。在地图程序中，可能需要检查用户提交的位置是否包含在已知位置列表中。

要判断特定的值是否已包含在列表中，可使用关键字in。来看你可能为比萨店编写的一些代码；这些代码首先创建一个列表，其中包含用户点的比萨配料，然后检查特定的配料是否包含在

该列表中。

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings
True
❷ >>> 'pepperoni' in requested_toppings
False
```

在❶处和❷处，关键字 `in` 让 Python 检查列表 `requested_toppings` 是否包含 `'mushrooms'` 和 `'pepperoni'`。这种技术很有用，它让你能够在创建一个列表后，轻松地检查其中是否包含特定的值。

5.2.7 检查特定值是否不包含在列表中

还有些时候，确定特定的值未包含在列表中很重要；在这种情况下，可使用关键字 `not in`。例如，如果有一个列表，其中包含被禁止在论坛上发表评论的用户，就可在允许用户提交评论前检查他是否被禁言：

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

❶ if user not in banned_users:
    print(user.title() + ", you can post a response if you wish.")
```

❶ 处的代码行明白易懂：如果 `user` 的值未包含在列表 `banned_users` 中，Python 将返回 `True`，进而执行缩进的代码行。

用户 `'marie'` 未包含在列表 `banned_users` 中，因此她将看到一条邀请她发表评论的消息：

```
Marie, you can post a response if you wish.
```

5.2.8 布尔表达式

随着你对编程的了解越来越深入，将遇到术语 **布尔表达式**，它不过是条件测试的别名。与条件表达式一样，布尔表达式的结果要么为 `True`，要么为 `False`。

布尔值通常用于记录条件，如游戏是否正在运行，或用户是否可以编辑网站的特定内容：

```
game_active = True
can_edit = False
```

在跟踪程序状态或程序中重要的条件方面，布尔值提供了一种高效的方式。

动手试一试

5-1 条件测试：编写一系列条件测试；将每个测试以及你对其结果的预测和实际结果都打印出来。你编写的代码应类似于下面这样：

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

详细研究实际结果，直到你明白了它为何为 True 或 False。

创建至少 10 个测试，且其中结果分别为 True 和 False 的测试都至少有 5 个。

5-2 更多的条件测试：你并非只能创建 10 个测试。如果你想尝试做更多的比较，可再编写一些测试，并将它们加入到 `conditional_tests.py` 中。对于下面列出的各种测试，至少编写一个结果为 True 和 False 的测试。

检查两个字符串相等和不等。

使用函数 `lower()` 的测试。

检查两个数字相等、不等、大于、小于、大于等于和小于等于。

使用关键字 `and` 和 `or` 的测试。

测试特定的值是否包含在列表中。

测试特定的值是否未包含在列表中。

5.3 if语句

理解条件测试后，就可以开始编写if语句了。if语句有很多种，选择使用哪种取决于要测试的条件数。前面讨论条件测试时，列举了多个if语句示例，下面更深入地讨论这个主题。

5.3.1 简单的if语句

最简单的if语句只有一个测试和一个操作：

```
if conditional_test:
    do something
```

在第1行中，可包含任何条件测试，而在紧跟在测试后面的缩进代码块中，可执行任何操作。如果条件测试的结果为True，Python就会执行紧跟在if语句后面的代码；否则Python将忽略这些代码。

假设有一个表示某人年龄的变量，而你想知道这个人是否够投票的年龄，可使用如下代码：

voting.py

```
age = 19
❶ if age >= 18:
❷     print("You are old enough to vote!")
```

在❶处，Python检查变量age的值是否大于或等于18；答案是肯定的，因此Python执行❷处缩进的print语句：

```
You are old enough to vote!
```

5

在if语句中，缩进的作用与for循环中相同。如果测试通过了，将执行if语句后面所有缩进的代码行，否则将忽略它们。

在紧跟在if语句后面的代码块中，可根据需要包含任意数量的代码行。下面在一个人够投票的年龄时再打印一行输出，问他是否登记了：

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

条件测试通过了，而两条print语句都缩进了，因此它们都将执行：

```
You are old enough to vote!
Have you registered to vote yet?
```

如果age的值小于18，这个程序将不会有任何输出。

5.3.2 if-else语句

经常需要在条件测试通过了时执行一个操作，并在没有通过时执行另一个操作；在这种情况下，可使用Python提供的if-else语句。if-else语句块类似于简单的if语句，但其中的else语句让你能够指定条件测试未通过时要执行的操作。

下面的代码在一个人够投票的年龄时显示与前面相同的消息，同时在这个人不够投票的年龄时也显示一条消息：

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

如果❶处的条件测试通过了，就执行第一个缩进的print语句块；如果测试结果为False，就执行❷处的else代码块。这次age小于18，条件测试未通过，因此执行else代码块中的代码：

```
Sorry, you are too young to vote.  
Please register to vote as soon as you turn 18!
```

上述代码之所以可行，是因为只存在两种情形：要么够投票的年龄，要么不够。if-else结构非常适合用于要让Python执行两种操作之一的情形。在这种简单的if-else结构中，总是会执行两个操作中的一个。

5.3.3 if-elif-else 结构

经常需要检查超过两个的情形，为此可使用Python提供的if-elif-else结构。Python只执行if-elif-else结构中的一个代码块，它依次检查每个条件测试，直到遇到通过了的条件测试。测试通过后，Python将执行紧跟在它后面的代码，并跳过余下的测试。

在现实世界中，很多情况下需要考虑的情形都超过两个。例如，来看一个根据年龄段收费的游乐场：

- 4岁以下免费；
- 4~18岁收费5美元；
- 18岁（含）以上收费10美元。

如果只使用一条if语句，如何确定门票价格呢？下面的代码确定一个人所属的年龄段，并打印一条包含门票价格的消息：

`amusement_park.py`

```
age = 12

❶ if age < 4:
    print("Your admission cost is $0.")
❷ elif age < 18:
    print("Your admission cost is $5.")
❸ else:
    print("Your admission cost is $10.")
```

❶处的if测试检查一个人是否不满4岁，如果是这样，Python就打印一条合适的消息，并跳过余下的测试。❷处的elif代码行其实是另一个if测试，它仅在前面的测试未通过时才会运行。在这里，我们知道这个人不小于4岁，因为第一个测试未通过。如果这个人未满18岁，Python将打印相应的消息，并跳过else代码块。如果if测试和elif测试都未通过，Python将运行❸处else代码块中的代码。

在这个示例中，❶处测试的结果为False，因此不执行其代码块。然而，第二个测试的结果为True（12小于18），因此将执行其代码块。输出为一个句子，向用户指出了门票价格：

```
Your admission cost is $5.
```

只要年龄超过17岁，前两个测试就都不能通过。在这种情况下，将执行else代码块，指出门票价格为10美元。

为让代码更简洁，可不在if-elif-else代码块中打印门票价格，而只在其中设置门票价格，并在它后面添加一条简单的print语句：

```
age = 12

if age < 4:
❶    price = 0
elif age < 18:
❷    price = 5
else:
❸    price = 10

❹ print("Your admission cost is $" + str(price) + ".")
```

5

❶处、❷处和❸处的代码行像前一个示例那样，根据人的年龄设置变量price的值。在if-elif-else结构中设置price的值后，一条未缩进的print语句❹会根据这个变量的值打印一条消息，指出门票的价格。

这些代码的输出与前一个示例相同，但if-elif-else结构的作用更小，它只确定门票价格，而不是在确定门票价格的同时打印一条消息。除效率更高外，这些修订后的代码还更容易修改：要调整输出消息的内容，只需修改一条而不是三条print语句。

5.3.4 使用多个 elif 代码块

可根据需要使用任意数量的elif代码块，例如，假设前述游乐场要给老年人打折，可再添加一个条件测试，判断顾客是否符合打折条件。下面假设对于65岁（含）以上的老人，可以半价（即5美元）购买门票：

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
❶ elif age < 65:
    price = 10
❷ else:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

这些代码大都未变。第二个elif代码块（见❶）通过检查确定年龄不到65岁后，才将门票价格设置为全票价格——10美元。请注意，在else代码块（见❷）中，必须将所赋的值改为5，因为仅当年龄超过65（含）时，才会执行这个代码块。

5.3.5 省略else代码块

Python并不要求if-elif结构后面必须有else代码块。在有些情况下，else代码块很有用；而在其他一些情况下，使用一条elif语句来处理特定的情形更清晰：

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
elif age < 65:
    price = 10
❶ elif age >= 65:
    price = 5

print("Your admission cost is $" + str(price) + ".")
```

❶处的elif代码块在顾客的年龄超过65（含）时，将价格设置为5美元，这比使用else代码块更清晰些。经过这样的修改后，每个代码块都仅在通过了相应的测试时才会执行。

else是一条包罗万象的语句，只要不满足任何if或elif中的条件测试，其中的代码就会执行，这可能会引入无效甚至恶意的数据。如果知道最终要测试的条件，应考虑使用一个elif代码块来代替else代码块。这样，你就可以肯定，仅当满足相应的条件时，你的代码才会执行。

5.3.6 测试多个条件

if-elif-else结构功能强大，但仅适合用于只有一个条件满足的情况：遇到通过了的测试后，Python就跳过余下的测试。这种行为很好，效率很高，让你能够测试一个特定的条件。

然而，有时候必须检查你关心的所有条件。在这种情况下，应使用一系列不包含elif和else代码块的简单if语句。在可能有多个条件为True，且你需要在每个条件为True时都采取相应措施时，适合使用这种方法。

下面再来看前面的比萨店示例。如果顾客点了两种配料，就需要确保在其比萨中包含这些配料：

toppings.py

```
❶ requested_toppings = ['mushrooms', 'extra cheese']

❷ if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

我们首先创建了一个列表，其中包含顾客点的配料（见❶）。❷处的if语句检查顾客是否点了配料蘑菇('mushrooms')，如果点了，就打印一条确认消息。❸处检查配料辣香肠('pepperoni')的代码也是一个简单的if语句，而不是elif或else语句；因此不管前一个测试是否通过，都将进行这个测试。❹处的代码检查顾客是否要求多加芝士('extra cheese')；不管前两个测试的结果如何，都会执行这些代码。每当这个程序运行时，都会进行这三个独立的测试。

在这个示例中，会检查每个条件，因此将在比萨中添加蘑菇并多加芝士：

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

如果像下面这样转而使用if-elif-else结构，代码将不能正确地运行，因为有一个测试通过后，就会跳过余下的测试：

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

第一个测试检查列表中是否包含'mushrooms'，它通过了，因此将在比萨中添加蘑菇。然而，Python将跳过if-elif-else结构中余下的测试，不再检查列表中是否包含'extra cheese'和'pepperoni'。其结果是，将添加顾客点的第一种配料，但不会添加其他的配料：

```
Adding mushrooms.

Finished making your pizza!
```

总之，如果你只想执行一个代码块，就使用if-elif-else结构；如果要运行多个代码块，就使用一系列独立的if语句。

动手试一试

5-3 外星人颜色#1：假设在游戏中刚射杀了一个外星人，请创建一个名为alien_color的变量，并将其设置为'green'、'yellow'或'red'。

- 编写一条if语句，检查外星人是否是绿色的；如果是，就打印一条消息，指出玩家获得了5个点。

- 编写这个程序的两个版本，在一个版本中上述测试通过了，而在另一个版本中未通过（未通过测试时没有输出）。

5-4 外星人颜色#2：像练习 5-3 那样设置外星人的颜色，并编写一个 if-else 结构。

- 如果外星人是绿色的，就打印一条消息，指出玩家因射杀该外星人获得了 5 个点。

- 如果外星人不是绿色的，就打印一条消息，指出玩家获得了 10 个点。

- 编写这个程序的两个版本，在一个版本中执行 if 代码块，而在另一个版本中执行 else 代码块。

5-5 外星人颜色#3：将练习 5-4 中的 if-else 结构改为 if-elif-else 结构。

- 如果外星人是绿色的，就打印一条消息，指出玩家获得了 5 个点。

- 如果外星人是黄色的，就打印一条消息，指出玩家获得了 10 个点。

- 如果外星人是红色的，就打印一条消息，指出玩家获得了 15 个点。

- 编写这个程序的三个版本，它们分别在外星人为绿色、黄色和红色时打印一条消息。

5-6 人生的不同阶段：设置变量 age 的值，再编写一个 if-elif-else 结构，根据 age 的值判断处于人生的哪个阶段。

- 如果一个人的年龄小于 2 岁，就打印一条消息，指出他是婴儿。

- 如果一个人的年龄为 2（含）~4 岁，就打印一条消息，指出他正蹒跚学步。

- 如果一个人的年龄为 4（含）~13 岁，就打印一条消息，指出他是儿童。

- 如果一个人的年龄为 13（含）~20 岁，就打印一条消息，指出他是青少年。

- 如果一个人的年龄为 20（含）~65 岁，就打印一条消息，指出他是成年人。

- 如果一个人的年龄超过 65（含）岁，就打印一条消息，指出他是老年人。

5-7 喜欢的水果：创建一个列表，其中包含你喜欢的水果，再编写一系列独立的 if 语句，检查列表中是否包含特定的水果。

- 将该列表命名为 favorite_fruits，并在其中包含三种水果。

- 编写 5 条 if 语句，每条都检查某种水果是否包含在列表中，如果包含在列表中，就打印一条消息，如“You really like bananas!”。

5.4 使用 if 语句处理列表

通过结合使用if语句和列表，可完成一些有趣的任务：对列表中特定的值做特殊处理；高效地管理不断变化的情形，如餐馆是否还有特定的食材；证明代码在各种情形下都将按预期那样运行。

5.4.1 检查特殊元素

本章开头通过一个简单示例演示了如何处理特殊值 'bmw'——它需要采用不同的格式进行打印。既然你对条件测试和 if 语句有了大致的认识，下面来进一步研究如何检查列表中的特殊值，并对其做合适的处理。

继续使用前面的比萨店示例。这家比萨店在制作比萨时，每添加一种配料都打印一条消息。通过创建一个列表，在其中包含顾客点的配料，并使用一个循环来指出添加到比萨中的配料，可以以极高的效率编写这样的代码：

5 toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print("Adding " + requested_topping + ".")  
  
print("\nFinished making your pizza!")
```

输出很简单，因为上述代码不过是一个简单的 for 循环：

```
Adding mushrooms.  
Adding green peppers.  
Adding extra cheese.
```

```
Finished making your pizza!
```

然而，如果比萨店的青椒用完了，该如何处理呢？为妥善地处理这种情况，可在 for 循环中包含一条 if 语句：

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
❶    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
❷    else:
        print("Adding " + requested_topping + ".")  
  
print("\nFinished making your pizza!")
```

这里在比萨中添加每种配料前都进行检查。❶ 处的代码检查顾客点的是不是青椒，如果是，就显示一条消息，指出不能点青椒的原因。❷ 处的 else 代码块确保其他配料都将添加到比萨中。输出表明，妥善地处理了顾客点的每种配料：

```
Adding mushrooms.  
Sorry, we are out of green peppers right now.  
Adding extra cheese.
```

```
Finished making your pizza!
```

5.4.2 确定列表不是空的

到目前为止，对于处理的每个列表都做了一个简单的假设，即假设它们都至少包含一个元素。我们马上就要让用户来提供存储在列表中的信息，因此不能再假设循环运行时列表不是空的。有鉴于此，在运行for循环前确定列表是否为空很重要。

下面在制作比萨前检查顾客点的配料列表是否为空。如果列表是空的，就向顾客确认他是否要点普通比萨；如果列表不为空，就像前面的示例那样制作比萨：

```

❶ requested_toppings = []

❷ if requested_toppings:
    for requested_topping in requested_toppings:
        print("Adding " + requested_topping + ".")
        print("\nFinished making your pizza!")

❸ else:
    print("Are you sure you want a plain pizza?")

```

在这里，我们首先创建了一个空列表，其中不包含任何配料（见❶）。在❷处我们进行了简单检查，而不是直接执行for循环。在if语句中将列表名用在条件表达式中时，Python将在列表至少包含一个元素时返回True，并在列表为空时返回False。如果requested_toppings不为空，就运行与前一个示例相同的for循环；否则，就打印一条消息，询问顾客是否确实要点不加任何配料的普通比萨（见❸）。

在这里，这个列表为空，因此输出如下——询问顾客是否确实要点普通比萨：

```
Are you sure you want a plain pizza?
```

如果这个列表不为空，将显示在比萨中添加的各种配料的输出。

5.4.3 使用多个列表

顾客的要求往往五花八门，在比萨配料方面尤其如此。如果顾客要在比萨中添加炸薯条，该怎么办呢？可使用列表和if语句来确定能否满足顾客的要求。

来看看在制作比萨前如何拒绝怪异的配料要求。下面的示例定义了两个列表，其中第一个列表包含比萨店供应的配料，而第二个列表包含顾客点的配料。这次对于requested_toppings中的每个元素，都检查它是否是比萨店供应的配料，再决定是否在比萨中添加它：

```

❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                      'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
    if requested_topping in available_toppings:
        print("Adding " + requested_topping + ".")
    else:
        print("Sorry, we don't have " + requested_topping + ".")

```

```
print("\nFinished making your pizza!")
```

在❶处，我们定义了一个列表，其中包含比萨店供应的配料。请注意，如果比萨店供应的配料是固定的，也可使用一个元组来存储它们。在❷处，我们又创建了一个列表，其中包含顾客点的配料，请注意那个不同寻常的配料——'french fries'。在❸处，我们遍历顾客点的配料列表。在这个循环中，对于顾客点的每种配料，我们都检查它是否包含在供应的配料列表中（见❹）；如果答案是肯定的，就将其加入到比萨中，否则将运行else代码块（见❺）：打印一条消息，告诉顾客不供应这种配料。

这些代码的输出整洁而详实：

5

```
Adding mushrooms.  
Sorry, we don't have french fries.  
Adding extra cheese.
```

```
Finished making your pizza!
```

通过为数不多的几行代码，我们高效地处理了一种真实的情形！

动手试一试

5-8 以特殊方式跟管理员打招呼： 创建一个至少包含 5 个用户名的列表，且其中一个用户名为 'admin'。想象你要编写代码，在每位用户登录网站后都打印一条问候消息。遍历用户名列表，并向每位用户打印一条问候消息。

- 如果用户名为 'admin'，就打印一条特殊的问候消息，如 "Hello admin, would you like to see a status report?"。
- 否则，打印一条普通的问候消息，如 "Hello Eric, thank you for logging in again"。

5-9 处理没有用户的情形： 在为完成练习 5-8 编写的程序中，添加一条 if 语句，检查用户名列表是否为空。

- 如果为空，就打印消息 "We need to find some users!"。
- 删除列表中的所有用户名，确定将打印正确的消息。

5-10 检查用户名： 按下面的说明编写一个程序，模拟网站确保每位用户的用户名都独一无二的方式。

- 创建一个至少包含 5 个用户名的列表，并将其命名为 current_users。
- 再创建一个包含 5 个用户名的列表，将其命名为 new_users，并确保其中有一两个用户名也包含在列表 current_users 中。
- 遍历列表 new_users，对于其中的每个用户名，都检查它是否已被使用。如果是这样，就打印一条消息，指出需要输入别的用户名；否则，打印一条消息，指出这个用户名未被使用。

□ 确保比较时不区分大消息；换句话说，如果用户名'John'已被使用，应拒绝用户名'JOHN'。

5-11 序数：序数表示位置，如1st和2nd。大多数序数都以th结尾，只有1、2和3例外。

□ 在一个列表中存储数字1~9。

□ 遍历这个列表。

□ 在循环中使用一个if-elif-else结构，以打印每个数字对应的序数。输出内容应为1st、2nd、3rd、4th、5th、6th、7th、8th和9th，但每个序数都独占一行。

5.5 设置if语句的格式

本章的每个示例都展示了良好的格式设置习惯。在条件测试的格式设置方面，PEP 8提供的唯一建议是，在诸如==、>=和<=等比较运算符两边各添加一个空格，例如，if age < 4:要比if age<4:好。这样的空格不会影响Python对代码的解读，而只是让代码阅读起来更容易。

动手试一试

5-12 设置if语句的格式：审核你在本章编写的程序，确保正确地设置了条件测试的格式。

5-13 自己的想法：与刚拿起本书时相比，现在你是一名能力更强的程序员了。鉴于你对如何在程序中模拟现实情形有了更深入的认识，你可以考虑使用程序来解决一些问题。随着编程技能不断提高，你可能想解决一些问题，请将这方面想法记录下来。想想你可能想编写的游戏、想研究的数据集以及想创建的Web应用程序。

5.6 小结

在本章中，你学习了如何编写结果要么为True要么为False的条件测试。你学习了如何编写简单的if语句、if-else语句和if-elif-else结构。在程序中，你使用了这些结构来测试特定的条件，以确定这些条件是否满足。你学习了如何在利用高效的for循环的同时，以不同于其他元素的方式对特定的列表元素进行处理。你还再次学习了Python就代码格式方面提出的建议，这可确保即便你编写的程序越来越复杂，其代码依然易于阅读和理解。

在第6章，你将学习Python字典。字典类似于列表，但让你能够将不同的信息关联起来。你将学习如何创建和遍历字典，以及如何将字典同列表和if语句结合起来使用。学习字典让你能够模拟更多现实世界的情形。



在本章中，你将学习能够将相关信息关联起来的Python字典。你将学习如何访问和修改字典中的信息。鉴于字典可存储的信息量几乎不受限制，因此我们会演示如何遍历字典中的数据。另外，你还将学习存储字典的列表、存储列表的字典和存储字典的字典。

理解字典后，你就能够更准确地为各种真实物体建模。你可以创建一个表示人的字典，然后想在其中存储多少信息就存储多少信息：姓名、年龄、地址、职业以及要描述的任何方面。你还能够存储任意两种相关的信息，如一系列单词及其含义，一系列人名及其喜欢的数字，以及一系列山脉及其海拔等。

6.1 一个简单的字典

来看一个游戏，其中包含一些外星人，这些外星人的颜色和点数各不相同。下面是一个简单的字典，存储了有关特定外星人的信息：

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

字典alien_0存储了外星人的颜色和点数。使用两条print语句来访问并打印这些信息，如下所示：

```
green
5
```

与大多数编程概念一样，要熟练使用字典，也需要一段时间的练习。使用字典一段时间后，你就会明白为何它们能够高效地模拟现实世界中的情形。