Think Python

如何像计算机科学家一样思考

2nd Edition, Version 2.2.14

Think Python

如何像计算机科学家一样思考

2nd Edition, Version 2.2.14

Allen Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

前言

本书与众不同的历史

1999 年 1 月，我正准备使用 Java 教一门入门编程课．这门课之前我已经开过三次，但是却感到越来越沮丧．不及格率太高，即便对于及格的学生，他们整体的收获也不大．

我觉得问题之一便是教材．它们都过于厚重，写了太多无关课程的 Java 细节，而又缺乏关于编程的上层指导 [1]．这些教材都落入了陷阱门效应[2]：开始的时候简单，逐渐深入，然后大概到了第五章左右，基础差的学生就跟不上了．学生们需要看的材料太多，进展太快，而我却要在接下来的学期里收拾残局[3]．

在课两周前，我决定自己写一本书．我的目标很明确：

- 尽量简短．让学生们读 10 页，胜过让他们读 50 页．
- 谨慎使用术语．我会尽量少用术语，而且第一次使用时，会给出定义．
- 循序渐进．为了避免陷阱门，我将最难的主题拆分成了很多个小节．
- 聚焦于编程，而不是编程语言．我只涵盖了 Java 最小可用子集，剔除了其余的部分．

所以我需要一个书名，一时兴起我选择了《*如何像计算机科学家一样思考*》．

本书的第一版很粗糙，但却很管用．学生们读了它之后，对书中内容理解的很好，因此我才可以在课堂上讲授那些困难、有趣的主题，并让学生们动手实践（这点非常重要）．

我将此书以 GNU 自由文档许可 的形式发布，允许用户拷贝、修改和传播此书．

接下来有趣的事发生了．弗吉尼亚一所高中的教师 Jeff Elkne 采用了我的教材，并改编为基于 Python 语言．当他将修改过的书稿发给我时，我读着自己的书学会了 Python．2001 年，通过 Green Tea Press，我出版了本书的第一个 Python 版本．

[1]上层指导 high-level guidance
[2]陷阱门效应 trap door effect
[3]收拾残局 pick up the pieces

2003 年，我开始在 Olin College 教书，并且第一次教授 Python 语言．与 Java 教学的对比很明显．学生们遇到的困难更少，学到的更多，开发了更有趣的工程，并且大部分人都学的更开心．

此后，我一直致力于本书的改善，纠正错误，改进示例，新增教学材料，特别是习题部分．

最后的结果就是你看到的这本书．而现在的书名没有之前那么夸张，《Think Python》．下面本书的一些新变化：

- 我在每章的最后新增了一个名叫调试的小节．我会在这些小节中，为大家介绍如何发现及避免 bug 的一般技巧，并提醒大家注意使用 Python 过程中可能的陷阱．
- 我增补了更多的练习题，从测试是否理解书中概念的小测试，到部分较大的项目．大部分的练习题后，我都会附上答案的链接．
- 我新增了一系列案例研究——更长的代码示例，既有练习题，也有答题解释和讨论．
- 我扩充了对程序开发计划及基本设计模式的内容介绍．
- 我增加了关于调试和算法分析的附录．

《Think Python》第二版还有以下新特点：

- 本书及其中的代码都已更新至 Python 3．
- 我增加了一些小节内容，还在本书网站上介绍如何在网络浏览器上运行 Python．这样，如果你嫌麻烦的话，就可以先不用在本地安装 Python．
- 在 海龟绘图 小节中，我没有继续使用自己编写的海龟绘图包 “Swampy”，改用了一个更标准的 Python 包 turtle．这个包更容易安装，功能更强大．
- 我新增了一个叫 “The Goodies” 的章节，给大家介绍一些严格来说并不是必须了解的 Python 特性，不过有时候这些特性还是很方便的．

我希望你能愉快的阅读这本书，也希望它能帮助你学习编程，学会像计算机科学家一样思考，至少有那么一点像．

Allen B. Downey

Olin College

Acknowledgments

Many thanks to Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

Thanks also to Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

Thanks to the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible, and Creative Commons for the license I am using now.

Thanks to the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

Thanks to the editors at O'Reilly Media who worked on *Think Python*.

Thanks to all the students who worked with earlier versions of this book and all the contributors (listed below) who sent in corrections and suggestions.

Contributor List

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

If you have a suggestion or correction, please send email to `feedback@thinkpython.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.

- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word "unconsciously" in Chapter 1 needed to be changed to "subconsciously".
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of "return value" in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.

- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.

- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that "a error" is an error.
- Abel David and Alexis Dinno reminded us that the plural of "matrix" is "matrices", not "matrixes". This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of "argument" and "parameter".
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of "use before def".
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner `spotted` an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise 11.5.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a "use before def".

- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise 14.3.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions*, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4–11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an "instance" and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.
- Russell Coleman helped me with my geometry.
- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.

- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa sent a long list of corrections and suggestions.
- Alix Etienne fixed one of the URLs.
- Kuang He found a typo.
- Daniel Neilson corrected an error about the order of operations.
- Will McGinnis pointed out that `polyline` was defined differently in two places.
- Swarup Sahoo spotted a missing semi-colon.
- Frank Hecker pointed out an exercise that was under-specified, and some broken links.
- Animesh B helped me clean up a confusing example.
- Martin Caspersen found two round-off errors.
- Gregor Ulm sent several corrections and suggestions.
- Dimitrios Tsirigkas suggested I clarify an exercise.
- Carlos Tafur sent a page of corrections and suggestions.
- Martin Nordsletten found a bug in an exercise solution.
- Lars O.D. Christensen found a broken reference.
- Victor Simeone found a typo.
- Sven Hoexter pointed out that a variable named `input` shadows a build-in function.
- Viet Le found a typo.
- Stephen Gregory pointed out the problem with `cmp` in Python 3.
- Matthew Shultz let me know about a broken link.
- Lokesh Kumar Makani let me know about some broken links and some changes in error messages.
- Ishwar Bhat corrected my statement of Fermat's last theorem.
- Brian McGhie suggested a clarification.
- Andrea Zanella translated the book into Italian, and sent a number of corrections along the way.
- Many, many thanks to Melissa Lewis and Luciano Ramalho for excellent comments and suggestions on the second edition.
- Thanks to Harry Percival from PythonAnywhere for his help getting people started running Python in a browser.
- Xavier Van Aubel made several useful corrections in the second edition.

目录

第一章　程序之道

本书的目标是教你像计算机科学家一样思考．这一思考方式集成了数学、工程以及自然科学的一些最好的特点．像数学家一样，计算机科学家使用形式语言表示思想（具体来说是计算）．像工程师一样，计算机科学家设计东西，将零件组成系统，在各种选择之间寻求平衡．像科学家一样，计算机科学家观察复杂系统的行为，形成假设并且对预测进行检验．

对于计算机科学家，最重要的技能是*问题求解* 的能力．问题求解 (problem solving) 意味着对问题进行形式化，寻求创新型的解决方案，并且清晰、准确地表达解决方案的能力．事实证明，学习编程的过程是锻炼问题解决能力的一个绝佳机会．这就是为什么本章被称为“程序之道”.

一方面，你将学习如何编程，这本身就是一个有用的技能．另一方面，你将把编程作为实现自己目的的手段．随着学习的深入，你会更清楚自己的目的．

1.1　什么是程序？

*程序*是一系列定义计算机如何执行计算 (computation) 的指令．这种计算可以是数学上的计算，例如寻找公式的解或多项式的根，也可以是一个符号计算 (symbolic computation)，例如在文档中搜索并替换文本或者图片，就像处理图片或播放视频．

不同编程语言所写程序的细节各不一样，但是一些基本的指令几乎出现在每种语言当中：

输入 (input)：　从键盘、文件、网络或者其他设备获取数据．

输出 (output)：　在屏幕上显示数据，将数据保存至文件，通过网络传送数据，等等．

数学 (math)：　执行基本的数学运算，如加法和乘法．

有条件执行 (conditional execution): 检查符合某个条件后，执行相应的代码．

重复 (repetition): 检查符合某个条件后，执行相应的代码．

无论你是否相信，程序的全部指令几乎都在这了．每个你曾经用过的程序，无论多么复杂，都是由跟这些差不多的指令构成的．因此，你可以认为编程就是将庞大、复杂的任务分解为越来越小的子任务，直到这些子任务简单到可以用这其中的一个基本指令执行．

1.2 运行 Python

Python 入门的一个障碍，是你可能需要在电脑上安装 Python 和相关软件．如果你能熟练使用命令行 (command-line interface)，安装 Python 对你来说就不是问题了．但是对于初学者，同时学习系统管理 (system administration) 和编程这两方面的知识是件痛苦的事．

为了避免这个问题，我建议你首先在浏览器中运行 Python．等你对 Python 更加了解之后，我会建议你在电脑上安装 Python．

网络上有许多网页可以让你运行 Python．如果你已经有最喜欢的网站，那就打开网页运行 Python 吧．如果没有，我推荐 PythonAnywhere．我在 http://tinyurl.com/thinkpython2e 给出了详细的使用指南．

目前 Python 有两个版本，分别是 Python 2 和 Python 3．二者十分相似，因此如果你学过某个版本，可以很容易地切换到另一个版本．事实上，作为初学者，你只会接触到很少数的不同之处．本书采用的是 Python 3，但是我会加入一些关于 Python 2 的说明．

Python 的**解释器** 是一个读取并执行 Python 代码的程序．根据你的电脑环境不同，你可以通过双击图标，或者在命令行输入`python` 的方式来启动解释器．解释器启动后，你应该看到类似下面的输出：

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

前三行中包含了关于解释器及其运行的操作系统的信息，因此你看到的内容可能不一样．但是你应该检查下版本号是否以 3 开头，上面示例中的版本号是 3.4.0．如果以 3 开头，那说明你正在运行 Python 3．如果以 2 开头，那说明你正在运行（你猜对了）Python 2．

最后一行是一个**提示符** (prompt)，表明你可以在解释器中输入代码了．如果你输入一行代码然后按回车 (Enter)，解释器就会显示结果：

```
>>> 1 + 1
2
```

现在你已经做好了开始学习的准备．接下来，我将默认你已经知道如何启动 Python 解释器和执行代码．

1.3 第一个程序

根据惯例，学习使用一门语言写的第一个程序叫做 “Hello, World!”，因为它的功能就是显示单词 “Hello, World!”．在 Python 中，这个程序看起来像这样：

```
>>> print('Hello,␣World!')
```

这是一个 print 函数的示例，尽管它并不会真的在纸上打印．它将结果显示在屏幕上．在此例中，结果是单词：

```
Hello, World!
```

程序中的单引号标记了被打印文本的首尾；它们不会出现在结果中．

括号说明 print 是一个函数．我们将在第三章介绍函数．

Python 2 中的打印语句略微不同，打印语句在 Python 2 中并不是一个函数，因此不需要使用括号．

```
>>> print 'Hello,␣World!'
```

很快你就会明白二者之间的区别[1]，现在知道这些就足够了．

1.4 算术运算符

接下来介绍算术．Python 提供了许多代表加法和乘法等运算的特殊符号，叫做 *S* 运算符 (operators)．

运算符 + 、 – 和 * 分别执行加法、减法和乘法，详见以下示例：

```
>>> 40 + 2
42
>>> 43 – 1
42
>>> 6 * 7
42
```

运算符 / 执行除法运算：

```
>>> 84 / 2
42.0
```

你可能会疑惑结果为什么是 42.0 而不是 42 ． 下节中我们会进行解释．

最后，运算符 * 执行乘方运算；也就是说，它将某个数字乘以自身相应的次数：

```
>>> 6**2 + 6
42
```

某些语言使用 ^ 运算符执行乘方运算，但是在 Python 中，它却属于一种位运算符，叫做 XOR ．如果你对位运算符不太了解，那么下面的结果会让你感到惊讶：

[1]译注：Python 核心开发者 Brett Cannon 详细解释了 为什么 print 在 Python 3 中变成了函数．

```
>>> 6 ^ 2
4
```

我们不会在本书中过深涉及位运算符，你可以通过阅读 Python 百科 — 位运算符 ，了解相关内容.

1.5 值和类型

值 (value) 是程序处理的基本数据之一，一个单词或一个数字都是值的实例．我们目前已经接触到的值有：`2`，`42.0`，和 `'Hello␣World!'`.

这些值又属于不同的类型 (types) ：`2` 是一个整型数 (integer)，`42.0` 是一个浮点型数 (floating point number)，而 `'Hello,␣World!'` 则是一个字符串 (string)，这么称呼是因为其中的字符被串[2]在了一起.

如果你不确定某个值的类型是什么，解释器可以告诉你：

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello,␣World!')
<class 'str'>
```

"class" 一词在上面的输出结果中，是类别的意思；一个类型就是一个类别的值.

不出意料，整型数属于 `int` 类型，字符串属于 `str` 类型，浮点数属于 `float` 类型.

那么像 `'2'` 和 `'42.0'` 这样的值呢？它们看上去像数字，但是又和字符串一样被引号括在了一起？

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

它们其实是字符串.

当你输入一个大数值的整型数时，你可能会想用逗号进行区分，比如说像这样：`1,000,000`. 在 Python 中，这不是一个合法的整型数，但是确实合法的值.

```
>>> 1,000,000
(1, 0, 0)
```

结果和我们预料的完全不同！Python 把 `1,000,000` 当作成了一个以逗号区分的整型数序列．在后面的章节中，我们会介绍更多有关这种序列的知识.

[2]strung together

1.6 形式语言和自然语言

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

自然语言 (natural language) 是人们交流所使用的语言，例如英语、西班牙语和法语．它们不是人为设计出来的（尽管有人试图这样做）；而是自然演变而来．

形式语言 (formal languages) 是人类为了特殊用途而设计出来的．例如，数学家使用的记号 (notation) 就是形式语言，特别擅长表示数字和符号之间的关系．化学家使用形式语言表示分子的化学结构．最重要的是：

> 编程语言是被设计用于表达计算的形式语言．

形式语言通常拥有严格的*语法*规则，规定了详细的语句结构．例如，$3 + 3 = 6$ 是语法正确的数学表达式，而 3+ = 3$6 则不是；$H_2O$ 是语法正确的化学式，而 $_2Zz$ 则不是．

语法规则有两种类型，分别涉及*记号* (tokens) 和结构．记号是语言的基本元素，例如单词、数字和化学元素．3+ = 3$6 这个式子的问题之一，就是 $ 在数学中不是一个合法的记号（至少据我所知）．类似的，$_2Zz$ 也不合法，因为没有一个元素的简写是 Zz ．

第二种语法规则与标记的组合方式有关．3+ = 3 这个方程是非法的，因为即使 + 和 = 都是合法的记号，但是你却不能把它们俩紧挨在一起．类似的，在化学式中，下标位于元素之后，而不是之前．

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with. [3]

当你读一个用英语写的句子或者用形式语言写的语句时，你都必须要理清各自的结构（尽管在阅读自然语言时，你是下意识地进行的）．这个过程被称为*解析* (parsing)．

虽然形式语言和自然语言有很多共同点——标记、结构和语法，它们也有一些不同：

歧义性 (ambiguity)： 自然语言充满歧义，人们使用上下文线索以及其它信息处理这些歧义．形式语言被设计成几乎或者完全没有歧义，这意味着不管上下文是什么，任何语句都只有一个意义．

冗余性 (redundancy)： 为了弥补歧义性并减少误解，自然语言使用很多冗余．结果，自然语言经常很冗长．形式语言则冗余较少，更简洁．

字面性 (literalness)： 自然语言充满成语和隐喻．如果我说 "The penny dropped"，可能根本没有便士、也没什么东西掉下来（这个成语的意思是，经过一段时间的困惑后终于理解某事）．形式语言的含义，与它们字面的意思完全一致．

由于我们都是说着自然语言长大的，我们有时候很难适应形式语言．形式语言与自然语言之间的不同，类似诗歌与散文之间的差异，而且更加明显：

诗歌 (Poetry)： 单词的含义和声音都有作用，整首诗作为一个整理，会对人产生影响，或是引发情感上的共鸣．歧义不但常见，而且经常是故意为之．

[3]译注：上面两句英文都是不符合语法的，一个包含非法标记，另一个结构不符合语法．

散文 (Prose)： 单词表面的含义更重要，句子结构背后的寓意更深．散文比诗歌更适合分析，但仍然经常有歧义．

程序 (Programs)： 计算机程序的含义是无歧义、无引申义的，通过分析程序的标记和结构，即可完全理解．

形式语言要比自然语言更加稠密，因此阅读起来花的时间会更长．另外，形式语言的结构也很重要，所以从上往下、从左往右阅读，并不总是最好的策略．相反，你得学会在脑海里分析一个程序，识别不同的标记并理解其结构．最后，注重细节．拼写和标点方面的小错误在自然语言中无伤大雅，但是在形式语言中却会产生很大的影响．

1.7 调试

程序员都会犯错．由于比较奇怪的原因，编程错误被称为故障[4]，追踪错误的过程被称为调试 (debugging)．

编程，尤其是调试，有时会让人动情绪．如果你有个很难的 bug 解决不了，你可能会感到愤怒、沮丧抑或是难堪．

有证据表明，人们很自然地把计算机当人来对待．当计算机表现好的时候，我们认为它们是队友，而当它们固执或无礼的时候，我们也会像对待固执或无礼人的一样对待它们 (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*)．

对这些反应做好准备有助于你对付它们．一种方法是将计算机看做是一个雇员，拥有特定的长处，例如速度和精度，也有些特别的缺点，像缺乏沟通以及不善于把握大局．

你的工作是当一个好的管理者：找到充分利用优点、摒弃弱点的方法．并且找到使用你的情感来解决问题的方法，而不是让你的情绪干扰你有效工作的能力．

学习调试可能很令人泄气，但是它对于许多编程之外的活动也是一个非常有价值的技能．在每一章的结尾，我都会花一节内容介绍一些调试建议，比如说这一节．希望能帮到你！

1.8 术语表

问题求解 (problem solving)： 将问题形式化、寻找并表达解决方案的过程．

高级语言 (high-level language)： 像 Python 这样被设计成人类容易阅读和编写的编程语言．

低级语言 (low-level language)： 被设计成计算机容易运行的编程语言；也被称为 “机器语言 (machine language)” 或 “汇编语言” (assembly language)．

可移植性 (portability)： 程序能够在多种计算机上运行的特性．

解释器 (interpreter)： 读取另一个程序并执行该程序的程序．

[4]译注：英文为 bug，一般指虫子

提示符 (prompt)： 解释器所显示的字符，表明已准备好接受用户的输入.

程序 (program)： 一组定义了计算内容的指令.

打印语句 (print statement)： 使 Python 解释器在屏幕上显示某个值的指令.

运算符 (operator)： 代表类似加法、乘法或者字符串连接 (string concatenation) 等简单计算的特殊符号.

值 (value)： 程序所处理数据的基本元素之一，例如数字或字符串.

类型 (type)： 值的类别. 我们目前接触的类型有整型数（类型为 `int`）、浮点数（类型为 `float` ）和字符串（类型为 `str` ）.

整型数 (integer)： 代表整数的类型.

浮点数 (floating-point)： 代表一个有小数点的数字的类型.

字符串 (string)： A type that represents sequences of characters.

自然语言 (natural language)： 任何的人们日常使用的、由自然演变而来的语言.

形式语言 (formal language)： 任何由人类为了某种目的而设计的语言，例如用来表示数学概念或者电脑程序；所有的编程语言都是形式语言.

记号 (token)： 程序语法结构中的基本元素之一，与自然语言中的单词类似.

语法 (syntax)： 规定了程序结构的规则.

解析 (parse)： 阅读程序，并分析其语法结构的过程

故障 (bug)： 程序中的错误.

调试 (debugging)： 寻找并解决错误的过程.

1.9 练习

Exercise 1.1. 建议读者在电脑上阅读本书，这样你可以随时测试书中的示例.

每当你试验一个新特性的时候，你应该试着去犯错. 举个例子，在“Hello, World!”程序中，如果你漏掉一个引号会发生什么情况？如果你去掉两个引号呢？如果你把`print`写错了呢？

这类试验能帮助你记忆读过的内容；对你平时编程也有帮助，因为你可以了解不同的错误信息代表的意思. 现在故意犯错误，总胜过以后不小心犯错.

1. 在打印语句中，如果你去掉一个或两个括号，会发生什么？
2. 你想打印一个字符串，如果你去掉一个或两个引号，会发生什么？
3. 你可以使用减号创建一个负数，如`-2`. 如果你在一个数字前再加上个加号，会发生什么？`2++2` 会得出什么结果？

4. 在数学标记中，前导零 (leading zeros) 没有问题，如`02`．如果我们在 *Python* 中这样做，会发生什么？

5. 如果两个值之间没有运算符，又会发生什么？

Exercise 1.2. 启动 Python 解释器，把它当计算器使用．

1. 42 分 42 秒一共是多少秒？

2. 10 公里可以换算成多少英里？提示：一英里等于 1.61 公里．

3. 如果你花 42 分 42 秒跑完了 10 公里，你的平均配速[5]是多少（每英里耗时，分别精确到分和秒）？你每小时平均跑了多少英里（英里/时）？

[5]译注：配速（pace）是在马拉松运动的训练中常使用的一个概念，配速是速度的一种，是每公里所需要的时间．配速 = 时间/距离．

第二章　变量、表达式和语句

编程语言最强大的特性之一，是操作变量的能力。*变量*是指向某个值的名称。

2.1　赋值语句

赋值语句 (assignment statement) 可用于新建变量，并为该变量赋值。

```
>>> message = 'And␣now␣for␣something␣completely␣different'
>>> n = 17
>>> pi = 3.141592653589793
```

这个例子进行了三次赋值。第一句将一个字符串赋给了名为 `message` 的新变量；第二句将整型数 `17` 赋给变量 `n`；第三句将 π 的（近似）值赋给变量 `pi`。

书面上更常用的表示变量的方法是写下变量名，并用箭头指向变量的值。这种图被称为*状态图* (state diagram)，因为它展示了每个变量所处的状态（可以把其看成是变量的心理状态）。图 2.1 展示了前面例子的结果。

2.2　变量名

程序员通常为变量选择有意义的名字 — 用于记录变量的用途。

变量名长度可以任意，它们可以包括字母和数字，但是不能以数字开头。使用大写字母是合法的，但是根据惯例，变量名只使用小写字母。

下划线 (_) 可以出现在变量名中。它经常用于有多个单词的变量名，例如 `my_name` 或者 `airspeed_of_unladen_swallow`。

如果你给了变量一个非法的名称，解释器将抛出一个语法错误：

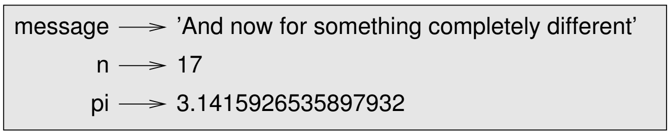

图 2.1: State diagram. ｜ 状态图。

```
>>> 76trombones = 'big␣parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced␣Theoretical␣Zymurgy'
SyntaxError: invalid syntax
```

76trombones 是非法的，因为它以数字开头。more@ 因为包含了一个非法字符@也是非法的。但是，class 错在哪儿了呢？

原来，class 是 Python 的关键字 (keywords) 之一。解释器使用关键字识别程序的结构，它们不能被用作变量名。

Python 3 有以下关键词：

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

你没有必要熟记这些关键词。大部分的开发环境会区分颜色显示关键词；如果你不小心使用关键词作为变量名，你会发现的。

2.3 表达式和语句

表达式 (expression) 是值、变量和运算符的组合。值和变量自身也是表达式，因此下面的表达式都是合法的：

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

当你在提示符后输入表达式时，解释器会计算 (evaluate) 该表达式，这就意味着解释器会求它的值。在上面的例子中，n 的值是 17，n + 25 的值是 42。

语句 (statement) 是一个会产生影响的代码单元，例如新建一个变量或显示某个值。

```
>>> n = 17
>>> print(n)
```

第一行是一个赋值语句，将某个值赋给了 n。第二行是一个打印语句，在屏幕上显示 n 的值。

当你输入一个语句后，解释器会执行 (execute) 这个语句，即按照语句的指令完成操作。一般来说，语句是没有值的。

2.4 脚本模式

到目前为止，我们都是在交互模式 (interactive mode) 下运行 Python，即直接与解释器进行交互。交互模式对学习入门很有帮助，但是如果你需要编写很多行代码，使用交互模式就不太方便了。

另一种方法是将代码保存到一个被称为脚本 (script) 的文件里，然后以脚本模式 (script mode) 运行解释器并执行脚本。按照惯例，Python 脚本文件名的后缀是 `.py`。

如果你知道如何在本地电脑新建并运行脚本，那你可以开始编码了。否则的话，我再次建议使用PythonAnywhere。我在 http://tinyurl.com/thinkpython2e 上贴出了如何以脚本模式运行解释器的指南。

由于 Python 支持这两种模式，在将代码写入脚本之前，你可以在交互模式下对代码片段进行测试。不过，交互模式和脚本模式之间存在一些差异，可能会让你感到疑惑。

举个例子，如果你把 Python 当计算器使用，你可能会输入下面这样的代码：

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

第一行将一个值赋给 `miles`，但是并没有产生可见的效果。第二行是一个表达式，因此解释器计算它并将结果显示出来。结果告诉我们，一段马拉松大概是 `42` 公里。

但是如果你将相同的代码键入一个脚本并且运行它，你得不到任何输出。在脚本模式下，表达式自身不会产生可见的效果。虽然 Python 实际上计算了表达式，但是如果你不告诉它要显示结果，它是不会那么做的。

```
miles = 26.2
print(miles * 1.61)
```

这个行为开始可能有些令人费解。

一个脚本通常包括一系列语句。如果有多于一条的语句，那么随着语句逐个执行，解释器会逐一显示计算结果。

例如，以下脚本

```
print(1)
x = 2
print(x)
```

produces the output

产生的输出结果是

```
1
2
```

赋值语句不产生输出。

在 Python 解释器中键入以下的语句，看看他们的结果是否符合你的理解：

```
5
x = 5
x + 1
```

现在将同样的语句写入一个脚本中并执行它。输出结果是什么？修改脚本，将每个表达式变成打印语句，再次运行它。

2.5 运算顺序

当一个表达式中有多于一个运算符时，计算的顺序由*运算顺序* (order of operations) 决定。对于算数运算符，Python 遵循数学里的惯例。缩写 **PEMDAS** 有助于帮助大家记住这些规则：

- *括号* (**P**arentheses) 具有最高的优先级，并且可以强制表达式按你希望的顺序计算。因为在括号中的表达式首先被计算，那么 `2 * (3-1)` 的结果是 `4`，`(1+1)**(5-2)` 的结果是 `8`。你也可以用括号提高表达式的可读性，如写成 `(minute * 100) / 60`，即使这样并不改变运算的结果。
- *指数运算* (**E**xponentiation) 具有次高的优先级，因此 `1 + 2**3` 的结果是 `9` 而非 `27`，`2 * 3**2` 的结果是 `18` 而非 `36`。
- *乘法* (**M**ultiplication) 和*除法* (**D**ivision) 有相同的优先级，比*加法* (**A**ddition) 和*减法* (**S**ubtraction) 高，加法和减法也具有相同的优先级。因此 `2*3-1` 是 `5` 而非 `4`，`6+4/2` 是 `8` 而非 `5`。
- 具有相同优先级的运算符按照从左到右的顺序进行计算（除了指数运算）。因此表达式 `degrees / 2 * pi` 中，除法先运算，然后结果被乘以 `pi`。为了被 2π 除，你可以使用括号，或者写成 `degrees / 2 / pi`。

我不会费力去记住这些运算符的优先级规则。如果看完表达式后分不出优先级，我会使用括号使计算顺序变得更明显。

2.6 字符串运算

一般来讲，你不能对字符串执行数学运算，即使字符串看起来很像数字，因此下面这些表达式是非法的：

```
'2'-'1'    'eggs'/'easy'    'third'*'a␣charm'
```

但有两个例外，`+` 和 `*`。

加号运算符 `+` 可用于字符串拼接[1]，也就是将字符串首尾相连起来。例如：

[1] string concatenation

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

乘法运算符 * 也可应用于字符串；它执行重复运算。例如，'Spam'*3 的结果是 'SpamSpamSpam'。如果其中一个运算数是字符串，则另外一个必须是整型数。

+ 和 * 的这个用法，类比加法和乘法也讲得通。就像 4*3 与 4+4+4 等价一样，我们也会期望 'Spam'*3 和 'Spam'+'Spam'+'Spam' 等价，而事实确实如此。另外，字符串拼接和重复与整数的加法和乘法也有很大的不同。你能想出来一个加法具有而字符串拼接不具有的特性么？

2.7 注释

随着程序变得越写越长，越来越复杂，它们的可读性也越来越差。形式语言是稠密的，通常很难在读一段代码后，说出其做什么或者为什么这样做。

因此，在你的程序中需要用自然语言做些笔记，解释程序将做些什么。这些笔记被称为注释 (comments)，以 # 符号开始。

```
# compute the percentage of the hour that has elapsed
# 计算逝去的时间占一小时的比例
percentage = (minute * 100) / 60
```

此例中，注释独占一行。你也可以将注释放在行尾：

```
percentage = (minute * 100) / 60     # percentage of an hour
```

从 # 开始到行尾的所有内容都会被解释器忽略—其内容对程序执行不会有任何影响。

在注释中记录代码不明显的特征，是最有帮助的。假设读者能够读懂代码做了**什么**是合理的；但是解释代码**为什么**这么做则更有用。

下面这个注释只是重复了代码，没有什么用：

```
v = 5     # assign 5 to v
```

下面的注释包括了代码中没有的有用信息：

```
v = 5     # velocity in meters/second.
```

好的变量名能够减少对注释的需求，但是长变量名使得表达式很难读，因此这里有个平衡问题。

2.8 调试

程序中可能会出现下面三种错误：*语法错误* (syntax error)、*运行时错误* (runtime error) 和*语义错误* (semantic error)。区别三者的差异有助于快速追踪这些错误。

语法错误： 语法指的是程序的结构及其背后的规则。例如，括号必须要成对出现，所以 `(1 + 2)` 是合法的，但是 `8)` 则是一个语法错误。

如果你的程序中存在一个语法错误，Python 会显示一条错误信息，然后退出运行。你无法顺利运行程序。在你编程生涯的头几周里，你可能会花大量时间追踪语法错误。随着你的经验不断积累，犯的语法错误会越来越少，发现错误的速度也会更快。

运行时错误： 第二种错误类型是运行时错误，这么称呼是因为这类错误只有在程序开始运行后才会出现。这类错误也被称为*异常* (exception)，因为它们的出现通常说明发生了某些特别的（而且不好的）事情。

在前几章提供的简单程序中，你很少会碰到运行时错误，所以你可能需要一段时间才会接触到这种错误。

语义错误： 第三类错误是“语义”错误，即与程序的意思的有关。如果你的程序中有语义错误，程序在运行时不会产生错误信息，但是不会返回正确的结果。它会返回另外的结果。严格来说，它是按照你的指令在运行。 识别语义错误可能是棘手的，因为这需要你反过来思考，通过观察程序的输出来搞清楚它在做什么。

2.9 术语表

变量 (variable)： 变量是指向某个值的名称。

赋值语句 (assignment)： 将某个值赋给变量的语句。

状态图 (state diagram)： 变量及其所指的值的图形化表示。

关键字 (keyword)： 关键字是用于解析程序的；你不能使用 if、def 和 while 这样的关键词作为变量名。

运算数 (operand)： 运算符所操作的值之一。

表达式 (expression)： 变量、运算符和值的组合，代表一个单一的结果。

计算 (evaluate)： 通过执行运算以简化表达式，从而得出一个单一的值。

语句 (statement)： 代表一个命令或行为的一段代码。目前为止我们接触的语句有赋值语句和打印语句。

执行 (execute)： 运行一个语句，并按照语句的指令操作。

交互式模式 (interactive mode)： 通过在提示符中输入代码，使用 Python 解释器的一种方式。

脚本模式 (script mode)： 使用 Python 解释器从脚本中读取代码，并运行脚本的方式。

脚本 (script)： 保存在文件中的程序。

运算顺序 (order of operations)： 有关多个运算符和运算数时计算顺序的规则。

拼接 (concatenate)： 将两个运算数首尾相连。

注释 (comment)： 程序中提供给其他程序员（任何阅读源代码的人）阅读的信息，对程序的执行没有影响。

语法错误 (syntax error)： 使得程序无法进行解析（因此无法进行解释）的错误。

异常 (exception)： 只有在程序运行时才发现的错误。

语义 (semantics)： 程序中表达的意思。

语义错误 (semantic error)： 使得程序偏离程序员原本期望的错误。

2.10 练习

Exercise 2.1. 和上一章一样，我还是要建议大家在学习新特性之后，在交互模式下充分试验，故意犯一些错误，看看到底会出什么问题。

- 我们已经知道`n = 42`是合法的。那么`42 = n`呢？
- `x = y = 1`合法吗？
- 在某些编程语言中，每个语句都是以分号;结束的。如果你在一个 Python 语句后也以分号结尾，会发生什么？
- 如果在语句最后带上分号呢？
- 在数学记法中，你可以将 x 和 y 像这样相乘：xy 。如果你在 Python 中也这么写的话，会发生什么？

Exercise 2.2. 继续练习将 Python 解释器当做计算器使用：

1. 半径为 r 的球体积是 $\frac{4}{3}\pi r^3$ 。半径为 5 的球体积是多少？
2. 假设一本书的零售价是 \$24.95，但书店有 40% 的折扣。运费则是第一本 \$3 ，以后每本 75 美分。购买 60 本的总价是多少？
3. 如果我上午 6:52 离开家，以轻松跑 (easy pace）的速度跑 1 里（即每英里耗时 8 分 15 秒），再以节奏跑 (tempo) 的速度跑 3 英里（每英里耗时 7 分 12 秒），之后又以放松跑的速度跑 1 英里，我什么时候回到家吃早饭？[2]

[2]译者注：配速 (pace) 是在马拉松运动的训练中常使用的一个概念，配速是速度的一种，是每公里所需要的时间。配速 = 时间/距离。Tempo run 一般被翻译成「节奏跑」或「乳酸门槛跑」，是指以比 10K 或 5K 比赛速度稍慢（每公里大约慢 10–15 秒）的速度进行训练，或者以平时 15K-半程的配速来跑。参考：https://www.zhihu.com/question/22237002

第三章　函数

在编程的语境下，*函数* (function) 是指一个有命名的、执行某个计算的语句序列 (sequence of statements) 。在定义一个函数的时候，你需要指定函数的名字和语句序列。之后，你可以通过这个名字 “调用 (call)” 该函数。

3.1　函数调用

我们已经看见过了*函数调用* (function call) 的例子。

```
>>> type(42)
<class 'int'>
```

这个函数的名字是 “type”。括号中的表达式被称为这个函数的*实参* (argument)。这个函数执行的结果，就是实参的类型。

人们常说函数 “接受 (accept)” 实参，然后 “返回 (return)” 一个结果。该结果也被称为*返回值* (return value)。

Python 提供了能够将值从一种类型转换为另一种类型的内建函数。函数 `int` 接受任意值，并在其能做到的情况下，将该值转换成一个整型数，否则会报错：

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` 能将浮点数转换为整型数，但是它并不进行舍入；只是截掉了小数点部分：

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` 可以将整型数和字符串转换为浮点数：

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

`str` 可以将其实参转换成字符串：

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 数学函数

Python 中有一个数学函数模块，提供了大部分常用的数学函数。模块 (module) 是指一个包含相关函数集合的文件。

在使用模块之前，我们需要通过导入语句 (import statement) 导入该模块：

```
>>> import math
```

这条语句会生成一个名为 “math” 的模块对象 (module object) 。如果你打印这个模块对象，你将获得关于它的一些信息：

```
>>> math
<module 'math' (built-in)>
```

该模块对象包括了定义在模块内的所有函数和变量。想要访问其中的一个函数，你必须指定该模块的名字以及函数名，并以点号（也被叫做句号）分隔开来。这种形式被称作点标记法 (dot notation)。

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

第一个例子使用`math.log10` 计算分贝信噪比（假设 `signal_power` 和`noise_power` 已经被定义了）。math 模块也提供了 `log` 函数，用于计算以 e 为底的对数。

第二个例子计算 `radians` 的正弦值。变量名暗示了 `sin` 及其它三角函数（`cos`、`tan` 等）接受弧度 (radians) 实参。度数转换为弧度，需要除以 180，并乘以 π ：

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

表达式 `math.pi` 从 `math` 模块中获得变量 `pi` 。该变量的值是 π 的一个浮点数近似值，精确到大约 15 位数。

如果你了解懂几何学 (trigonometry) ，你可以将之前的结果和二分之根号二进行比较，检查是否正确：

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3 构建

目前为止，我们已经分别介绍了程序的基本元素 — 变量、表达式和语句，但是还没有讨论如何将它们组合在一起。

编程语言的最有用特征之一，是能够将小块构建材料 (building blocks) 构建 (compose) 在一起。例如，函数的实参可以是任意类型的表达式，包括算术运算符：

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

甚至是函数调用：

```
x = math.exp(math.log(x+1))
```

几乎任何可以放一个值的地方，都可以放一个任意类型的表达式，除了一个例外：赋值语句的左侧必须是一个变量名。左侧放其他任何表达式都会产生语法错误（后面我们会讲到这个规则的例外）。

```
>>> minutes = hours * 60                 # right
>>> hours * 60 = minutes                 # wrong!
SyntaxError: can't␣assign␣to␣operator
```

3.4 增加新函数

目前为止，我们只使用了 Python 自带的函数，但是增加新函数也是可能的。一个函数定义 (function definition) 指定了新函数的名称以及当函数被调用时执行的语句序列。

下面是一个示例：

```
def print_lyrics():
    print("I'm␣a␣lumberjack,␣and␣I'm␣okay.")
    print("I␣sleep␣all␣night␣and␣I␣work␣all␣day.")
```

def 是一个关键字，表明这是一个函数定义。这个函数的名字是 print_lyrics。函数的命名规则与变量名相同：字母、数字以及下划线是合法的，但是第一个字符不能是数字。不能使用关键字作为函数名，并应该避免变量和函数同名。

The empty parentheses after the name indicate that this function doesn't take any arguments.

函数名后面的圆括号是空的，表明该函数不接受任何实参。

函数定义的第一行被称作*函数头* (header)；其余部分被称作*函数体* (body)。函数头必须以冒号结尾，而函数体必须缩进。按照惯例，缩进总是 4 个空格。函数体能包含任意条语句。

打印语句中的字符串被括在双引号中。单引号和双引号的作用相同；大多数人使用单引号，上述代码中的情况除外，即单引号（同时也是撇号）出现在字符串中时。

所有引号（单引号和双引号）必须是" 直引号" (straight quotes)，它们通常位于键盘上 Enter 键的旁边。像这句话中使用的 “弯引号” (curly quotes)，在 Python 语言中则是不合法的。

如果你在交互模式下键入函数定义，每空一行解释器就会打印三个句点 ...，让你知道定义并没有结束。

```
>>> def print_lyrics():
...     print("I'm␣a␣lumberjack,␣and␣I'm␣okay.")
...     print("I␣sleep␣all␣night␣and␣I␣work␣all␣day.")
...
```

为了结束函数定义，你必须输入一个空行。

定义一个函数会创建一个*函数对象* (function object)，其类型是 function：

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

调用新函数的语法，和调用内建函数的语法相同：

```
>>> print_lyrics()
I'm␣a␣lumberjack,␣and␣I'm okay.
I sleep all night and I work all day.
```

一旦你定义了一个函数，你就可以在另一个函数内部使用它。例如，为了重复之前的叠句 (refrain)，我们可以编写一个名叫 repeat_lyrics：

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

然后调用 repeat_lyrics

```
>>> repeat_lyrics()
I'm␣a␣lumberjack,␣and␣I'm okay.
I sleep all night and I work all day.
I'm␣a␣lumberjack,␣and␣I'm okay.
I sleep all night and I work all day.
```

不过，这首歌的歌词实际上不是这样的。

3.5　定义和使用

将上一节的多个代码段组合在一起，整个程序看起来是这样的：

```
def print_lyrics():
    print("I'm␣a␣lumberjack,␣and␣I'm␣okay.")
    print("I␣sleep␣all␣night␣and␣I␣work␣all␣day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

该程序包含两个函数定义：print_lyrics 和 repeat_lyrics。函数定义和其它语句一样，都会被执行，但是其作用是创建函数对象。函数内部的语句在函数被调用之前，是不会执行的，而且函数定义不会产生任何输出。

你可能猜到了，在运行函数之前，你必须先创建这个函数。换句话说，函数定义必须在其第一次被调用之前执行。

我们做个小练习，将程序的最后一行移到顶部，使得函数调用出现在函数定义之前。运行程序，看看会得到怎样的错误信息。

现在将函数调用移回底部，然后将 print_lyrics 的定义移到 repeat_lyrics 的定义之后。这次运行程序时会发生什么？

3.6　执行流程

为了保证函数第一次使用之前已经被定义，你必须要了解语句执行的顺序，这也被称作执行流程 (flow of execution) 。

执行流程总是从程序的第一条语句开始，自顶向下，每次执行一条语句。

函数定义不改变程序执行的流程，但是请记住，函数不被调用的话，函数内部的语句是不会执行的。

函数调用像是在执行流程上绕了一个弯路。执行流程没有进入下一条语句，而是跳入了函数体，开始执行那里的语句，然后再回到它离开的位置。

这听起来足够简单，至少在你想起一个函数可以调用另一个函数之前。当一个函数执行到中间的时候，程序可能必须执行另一个函数里的语句。然后在执行那个新函数的时候，程序可能又得执行另外一个函数！

幸运的是，Python 善于记录程序执行流程的位置，因此每次一个函数执行完成时，程序会回到调用它的那个函数原来执行的位置。当到达程序的结尾时，程序才会终止。

总之，阅读程序时，你没有必要总是从上往下读。有时候，跟着执行流程阅读反而更加合理。

3.7 形参和实参

我们之前接触的一些函数需要实参。例如，当你调用 `math.sin` 时，你传递一个数字作为实参。有些函数接受一个以上的实参：`math.pow` 接受两个，底数和指数。

在函数内部，实参被赋给称作*形参* (parameters) 的变量。下面的代码定义了一个接受一个实参的函数：

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

这个函数将实参赋给名为 `bruce` 的形参。当函数被调用的时候，它会打印形参（无论它是什么）的值两次。

该函数对任意能被打印的值都有效。

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

组合规则不仅适用于内建函数，而且也适用于开发者自定义的函数（programmer-defined functions），因此我们可以使用任意类型的表达式作为 `print_twice` 的实参：

```
>>> print_twice('Spam␣'*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

在函数被调用之前，实参会先进行计算，因此在这些例子中，表达式 `Spam '*4` 和 `math.cos(math.pi)` 都只被计算了一次。

你也可以用变量作为实参：

```
>>> michael = 'Eric,␣the␣half␣a␣bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

我们传递的实参名 `michael` 与形参的名字 `bruce` 没有任何关系。这个值在传入函数之前叫什么都没有关系；只要传入了 `print_twice` 函数，我们将所有人都称为 `bruce`。

3.8 变量和形参都是局部的

当你在函数里面创建变量时，这个变量是*局部的* (local)，也就是说它只在函数内部存在。例如：

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

该函数接受两个实参，拼接 (concatenates) 它们并打印结果两次。下面是使用该函数的一个示例：

```
>>> line1 = 'Bing␣tiddle␣'
>>> line2 = 'tiddle␣bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

当 `cat_twice` 结束时，变量 `cat` 被销毁了。如果我们试图打印它，我们将获得一个异常：

```
>>> print(cat)
NameError: name 'cat' is not defined
```

形参也都是局部的。例如，在 `print_twice` 函数的外部并没有 `bruce` 这个变量。

3.9 堆栈图

有时，画一个 ** 堆栈图（stack diagram）** 可以帮助你跟踪哪个变量能在哪儿用。与状态图类似，堆栈图要说明每个变量的值，但是它们也要说明每个变量所属的函数。

每个函数用一个栈帧 (frame) 表示。一个栈帧就是一个线框，函数名在旁边，形参以及函数内部的变量则在里面。前面例子的堆栈图如图 3.1所示。

这些线框排列成栈的形式，说明了哪个函数调用了哪个函数等信息。在此例中，`print_twice` 被 `cat_twice` 调用，`cat_twice` 又被 `__main__` 调用，`__main__` 是一个表示最上层栈帧的特殊名字。当你在所有函数之外创建一个变量时，它就属于 `__main__`。

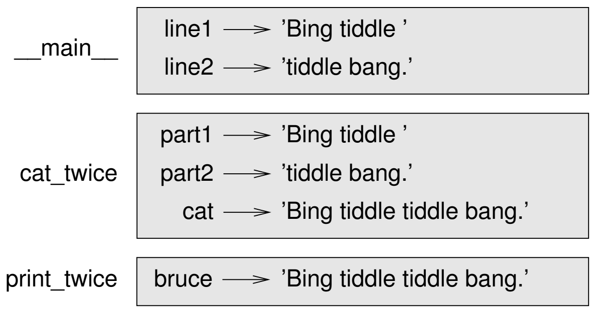

图 3.1: 堆栈图。

每个形参都指向其对应实参的值。因此，part1 和 line1 的值相同，part2 和 line2 的值相同，bruce 和 cat 的值相同。

如果函数调用时发生错误，Python 会打印出错函数的名字以及调用它的函数的名字，以及调用 * 后面这个函数 * 的函数的名字，一直追溯到 __main__ 为止。

例如，如果你试图在 print_twice 里面访问 cat ，你将获得一个 NameError :

这个函数列表被称作*回溯* (traceback) 。它告诉你发生错误的是哪个程序文件，错误在哪一行，以及当时在执行哪个函数。它还会显示引起错误的那一行代码。

回溯中的函数顺序，与堆栈图中的函数顺序一致。出错时正在运行的那个函数则位于回溯信息的底部。

3.10 有返回值函数和无返回值函数

有一些我们之前用过的函数，例如数学函数，会返回结果；由于没有更好的名字，我姑且叫它们*有返回值函数* (fruitful functions) 。其它的函数，像 print_twice ，执行一个动作但是不返回任何值。我称它们为*无返回值函数* (void functions) 。

当你调用一个有返回值函数时，你几乎总是想用返回的结果去做些什么；例如，你可能将它赋值给一个变量，或者把它用在表达式里：

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

当你在交互模式下调用一个函数时，Python 解释器会马上显示结果：

```
>>> math.sqrt(5)
2.2360679774997898
```

但是在脚本中，如果你单单调用一个有返回值函数，返回值就永远丢失了！

```
math.sqrt(5)
```

该脚本计算 5 的平方根，但是因为它没保存或者显示这个结果，这个脚本并没多大用处。

无返回值函数可能在屏幕上打印输出结果，或者产生其它的影响，但是它们并没有返回值。如果你试图将无返回值函数的结果赋给一个变量，你会得到一个被称作 None 的特殊值。

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

None 这个值和字符串 'None' 不同。这是一个具有独立类型的特殊值：

```
>>> print(type(None))
<class 'NoneType'>
```

目前为止，我们写的函数都是无返回值函数。我们将在几章之后开始编写有返回值函数。

3.11 为什么使用函数？

你可能还不明白为什么值得将一个程序分解成多个函数。原因包括以下几点：

- 创建一个新的函数可以让你给一组语句命名，这可以让你的程序更容易阅读和调试。
- 通过消除重复的代码，函数精简了程序。以后，如果你要做个变动，你只需在一处修改即可。
- 将一个长程序分解为多个函数，可以让你一次调试一部分，然后再将它们组合为一个可行的整体。
- 设计良好的函数经常对多个程序都有帮助。一旦你写出并调试好一个函数，你就可以重复使用它。

3.12 调试

调试，是你能获得的最重要的技能之一。虽然调试会让人沮丧，但却是编程过程中最富含智慧、挑战以及乐趣的一部分。

在某些方面，调试像是侦探工作。你面对一些线索，必须推理出是什么进程 (processes) 和事件 (events) 导致了你看到的结果。

调试也像是一门实验性科学。一旦你猜到大概哪里出错了，你可以修改程序，再试一次。如果你的假设是正确的，那么你就可以预测到修改的结果，并且离正常运行的程序

又近了一步。如果你的假设是错误的，你就不得不再提一个新的假设。如夏洛克・福尔摩斯所指出的 "当你排除了所有的不可能，无论剩下的是什么，不管多么难以置信，一定就是真相。"（阿瑟・柯南・道尔，《*四签名*》）

对某些人来说，编程和调试是同一件事。也就是说，编程是逐步调试一个程序，直到它满足了你期待的过程。这意味着，你应该从一个能*正常运行* (working) 的程序开始，每次只做一些小改动，并同步进行调试。

举个例子，Linux 是一个有着数百万行代码的操作系统但是它一开始，只是 Linus Torvalds 写的一个用于研究 Intel 80386 芯片的简单程序。根据 Larry Greenfield 的描述，"Linus 的早期项目中，有一个能够交替打印 AAAA 和 BBBB 的程序。这个程序后来演变为了 Linux。"（*Linux 用户手册* Beta 版本 1）。

3.13 术语表

函数 (function)： 执行某种有用运算的命名语句序列。函数可以接受形参，也可以不接受；可以返回一个结果，也可以不返回。

函数定义 (function definition：) 创建一个新函数的语句，指定了函数名、形参以及所包含的语句。

函数对象 (function object)： 函数定义所创建的一个值。函数名是一个指向函数对象的变量。

函数头 (header)： 函数定义的第一行。

函数体 (body)： 函数定义内部的语句序列。

形参 (parameters)： 函数内部用于指向被传作实参的值的名字。

函数调用 (function call) ： 运行一个函数的语句。它包括了函数名，紧随其后的实参列表，实参用圆括号包围起来。

实参 (argument) ： 函数调用时传给函数的值。这个值被赋给函数中相对应的形参。

局部变量 (local variable)： 函数内部定义的变量。局部变量只能在函数内部使用。

返回值 (return value)： 函数执行的结果。如果函数调用被用作表达式，其返回值是这个表达式的值。

有返回值函数 (fruitful function)： 会返回一个值的函数。

无返回值函数 (void function)： 总是返回 None 的函数。

`None`： 无返回值函数返回的一个特殊值。

模块 (module)： 包含了一组相关函数及其他定义的的文件。

导入语句 (import statement)： 读取一个模块文件，并创建一个模块对象的语句。

模块对象 (module object)： 导入语句创建的一个值，可以让开发者访问模块内部定义的值。

点标记法 (dot notation)： 调用另一个模块中函数的语法，需要指定模块名称，之后跟着一个点（句号）和函数名。

组合 (composition)： 将一个表达式嵌入一个更长的表达式，或者是将一个语句嵌入一个更长语句的一部分。

执行流程 (flow of execution)： 语句执行的顺序。

堆栈图 (stack diagram)： 一种图形化表示堆栈的方法，堆栈中包括函数、函数的变量及其所指向的值。

栈帧 (frame) ： 堆栈图中一个栈帧，代表一个函数调用。其中包含了函数的局部变量和形参。

回溯 (traceback) ： 当出现异常常时，解释器打印出的出错时正在执行的函数列表。

3.14 练习

Exercise 3.1. 编写一个名为right_justify 的函数，函数接受一个名为s的字符串作为形参，并在打印足够多的前导空格 (leading space) 之后打印这个字符串，使得字符串的最后一个字母位于显示屏的第 70 列。

```
>>> right_justify('monty')
                                                                     monty
```

提示：使用字符串拼接 (string concatenation) 和重复。另外，Python 提供了一个名叫len 的内建函数，可以返回一个字符串的长度，因此len('allen') 的值是 5。

Exercise 3.2. 函数对象是一个可以赋值给变量的值，也可以作为实参传递。例如，do_twice 函数接受函数对象作为实参，并调用这个函数对象两次：

```
def do_twice(f):
    f()
    f()
```

下面这个示例使用do_twice 来调用名为print_spam 的函数两次。

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. 将这个示例写入脚本，并测试。

2. 修改do_twice，使其接受两个实参，一个是函数对象，另一个是值。然后调用这一函数对象两次，将那个值传递给函数对象作为实参。

3. 从本章前面一些的示例中，将print_twice 函数的定义复制到脚本中。

4. 使用修改过的do_twice，调用print_twice 两次，将spam 传递给它作为实参。
5. 定义一个名为do_four 的新函数，其接受一个函数对象和一个值作为实参。调用这个函数对象四次，将那个值作为形参传递给它。函数体中应该只有两条语句，而不是四条。

参考答案

Exercise 3.3. 注意：请使用我们目前学过的语句和特性来完成本题。

1. 编写一个能画出如下网格 (grid) 的函数：

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

   提示：你可以使用一个用逗号分隔的值序列，在一行中打印出多个值：

```
print('+', '-')
```

   print 函数默认会自动换行，但是你可以阻止这个行为，只需要像下面这样将行结尾变成一个空格：

```
print('+', end='␣')
print('-')
```

   这两个语句的输出结果是+ -'。

   一个没有传入实参的print 语句会结束当前行，跳到下一行。

2. 编写一个能够画出四行四列的类似网格的函数。

参考答案

致谢：这个习题基于 Practical C Programming, Third Edition 一书中的习题改编，该书由 O’Reilly 出版社于 1997 年出版。

第四章　Case study: interface design | 案例研究：接口设计

This chapter presents a case study that demonstrates a process for designing functions that work together.

It introduces the `turtle` module, which allows you to create images using turtle graphics. The `turtle` module is included in most Python installations, but if you are running Python using PythonAnywhere, you won't be able to run the turtle examples (at least you couldn't when I wrote this).

If you have already installed Python on your computer, you should be able to run the examples. Otherwise, now is a good time to install. I have posted instructions at `http://tinyurl.com/thinkpython2e`.

Code examples from this chapter are available from `http://thinkpython2.com/code/polygon.py`.

本章将通过一个案例研究，介绍如何设计出相互配合的函数。

本章会介绍 `turtle` 模块，它可以让你使用海龟图形 (turtle graphics) 绘制图像。大部分的 Python 安装环境下都包含了这个模块，但是如果你是在 PythonAnywhere 上运行 Python 的，你将无法运行本章中的代码示例（至少在我写这章时是做不到的）。

如果你已经在自己的电脑上安装了 Python，那么不会有问题。如果没有，现在就是安装 Python 的好时机。我在 http://tinyurl.com/thinkpython2e 这个页面上发布了相关指南。

本章的示例代码可以从 http://thinkpython2.com/code/polygon.py 获得。

4.1　The turtle module | turtle 模块

To check whether you have the `turtle` module, open the Python interpreter and type

打开 Python 解释器，输入以下代码，检查你是否安装了 `turltle` 模块：

```
>>> import turtle
>>> bob = turtle.Turtle()
```

When you run this code, it should create a new window with small arrow that represents the turtle. Close the window.

上述代码运行后，应该会新建一个窗口，窗口中间有一个小箭头，代表的就是海龟。现在关闭窗口。

Create a file named `mypolygon.py` and type in the following code:

新建一个名叫 `mypolygon.py` 的文件，输入以下代码：

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

The `turtle` module (with a lowercase 't') provides a function called `Turtle` (with an uppercase 'T') that creates a Turtle object, which we assign to a variable named `bob`. Printing `bob` displays something like:

`turtle` 模块（小写的 `'t'` ）提供了一个叫作 `Turtle` 的函数（大写的 `'T'`），这个函数会创建一个 `Turtle` 对象，我们将其赋值给名为 `bob` 的变量。打印 `bob` 的话，会输出下面这样的结果：

```
<turtle.Turtle object at 0xb7bfbf4c>
```

This means that `bob` refers to an object with type `Turtle` as defined in module `turtle`.

这意味着，`bob` 指向一个类型为 Turtle 的对象，这个类型是由 `turtle` 模块定义的。

`mainloop` tells the window to wait for the user to do something, although in this case there's not much for the user to do except close the window.

`mainloop` 告诉窗口等待用户操作，尽管在这个例子中，用户除了关闭窗口之外，并没有其他可做的事情。

Once you create a Turtle, you can call a **method** to move it around the window. A method is similar to a function, but it uses slightly different syntax. For example, to move the turtle forward:

创建了一个 `Turtle` 对象之后，你可以调用 *方法* (method) 来在窗口中移动该对象。方法与函数类似，但是其语法略有不同。例如，要让海龟向前走：

```
bob.fd(100)
```

The method, `fd`, is associated with the turtle object we're calling `bob`. Calling a method is like making a request: you are asking `bob` to move forward.

方法 `fd` 与我们称之为 `bob` 的对象是相关联的。调用方法就像提出一个请求：你在请求 `bob` 往前走。

The argument of `fd` is a distance in pixels, so the actual size depends on your display.

`fd` 方法的实参是像素距离，所以实际前进的距离取决于你的屏幕。

Other methods you can call on a Turtle are `bk` to move backward, `lt` for left turn, and `rt` right turn. The argument for `lt` and `rt` is an angle in degrees.

`Turtle` 对象中你能调用的其他方法还包括：让它向后走的 `bk` ，向左转的 `lt` ，向右转的 `rt` 。`lt` 和 `rt` 这两个方法接受的实参是角度。

Also, each Turtle is holding a pen, which is either down or up; if the pen is down, the Turtle leaves a trail when it moves. The methods `pu` and `pd` stand for "pen up" and "pen down".

另外，每个 `Turtle` 都握着一支笔，不是落笔就是抬笔；如果落笔了，`Turtle` 就会在移动时留下痕迹。`pu` 和 `pd` 这两个方法分别代表 "抬笔 (pen up)" 和 "落笔 (pen down)"。

To draw a right angle, add these lines to the program (after creating `bob` and before calling `mainloop`):

如果要画一个直角 (right angle)，请在程序中添加以下代码（放在创建 `bob` 之后，调用 `mainloop` 之前）：

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

When you run this program, you should see `bob` move east and then north, leaving two line segments behind.

当你运行此程序时，你应该会看到 `bob` 先朝东移动，然后向北移动，同时在身后留下两条线段 (line segment)。

Now modify the program to draw a square. Don't go on until you've got it working!

现在修改程序，画一个正方形。在没有成功之前，不要继续往下看。

4.2 Simple repetition | 简单重复

Chances are you wrote something like this:

很有可能你刚才写了像下面这样的一个程序：

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

We can do the same thing more concisely with a for statement. Add this example to mypolygon.py and run it again:

我们可以利用一个 for 语句，以更简洁的代码来做相同的事情。将下面的示例代码加入 mypolygon.py ，并重新运行：

```
for i in range(4):
    print('Hello!')
```

You should see something like this:

你会看到以下输出：

```
Hello!
Hello!
Hello!
Hello!
```

This is the simplest use of the for statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

这是 for 语句最简单的用法；后面我们会介绍更多的用法。但是这对于让你重写画正方形的程序已经足够了。如果没有完成，请不要往下看。

Here is a for statement that draws a square:

下面是一个画正方形的 for 语句：

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

The syntax of a for statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements.

for 语句的语法和函数定义类似。它有一个以冒号结尾的语句头（header）以及一个缩进的语句体（body）。语句体可以包含任意条语句。

A for statement is also called a **loop** because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body four times.

for 语句有时也被称为 循环 (loop)，因为执行流程会贯穿整个语句体，然后再循环回顶部。在此例中，它将运行语句体四次。

This version is actually a little different from the previous square-drawing code because it makes another turn after drawing the last side of the square. The extra turn takes more time, but it simplifies the code if we do the same thing every time through the loop. This version also has the effect of leaving the turtle back in the starting position, facing in the starting direction.

这个版本事实上和前面画正方形的代码有所不同，因为它在画完正方形的最后一条边后，又多转了一下。这个额外的转动多花了些时间，但是如果我们每次都通过循环来做这件事情，这样反而是简化了代码。这个版本还让海龟回到了初始位置，朝向也与出发时一致。

4.3 Exercises | 练习

The following is a series of exercises using TurtleWorld. They are meant to be fun, but they have a point, too. While you are working on them, think about what the point is.

下面是一系列学习使用 `Turtle`[1] 的练习。这些练习也许很好玩，但它们是有针对性设计的。在做这些练习的时候，可以思考一下它们的目的是什么。

The following sections have solutions to the exercises, so don't look until you have finished (or at least tried).

后面几节是中介绍了这些练习的答案，因此如果你还没完成（或者至少试过），请不要看答案。

1. Write a function called `square` that takes a parameter named `t`, which is a turtle. It should use the turtle to draw a square.

   Write a function call that passes `bob` as an argument to `square`, and then run the program again.

2. Add another parameter, named `length`, to `square`. Modify the body so length of the sides is `length`, and then modify the function call to provide a second argument. Run the program again. Test your program with a range of values for `length`.

3. Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an n-sided regular polygon. Hint: The exterior angles of an n-sided regular polygon are $360/n$ degrees.

4. Write a function called `circle` that takes a turtle, `t`, and radius, `r`, as parameters and that draws an approximate circle by calling `polygon` with an appropriate length and number of sides. Test your function with a range of values of `r`.

   Hint: figure out the circumference of the circle and make sure that `length * n = circumference`.

5. Make a more general version of `circle` called `arc` that takes an additional parameter `angle`, which determines what fraction of a circle to draw. `angle` is in units of degrees, so when `angle=360`, `arc` should draw a complete circle.

1. 写一个名为 `square` 的函数，接受一个名为 `t` 的形参，`t` 是一个海龟。这个函数应用这只海龟画一个正方形。

   写一个函数调用，将 `bob` 作为实参传给 `square` ，然后再重新运行程序。

[1]译注：原文中使用的还是 `TurtleWorld` ，应该是作者忘了修改。

2. 给 `square` 增加另一个名为 `length` 的形参。修改函数体，使得正方形边的长度是 `length` ，然后修改函数调用，提供第二个实参。重新运行程序。用一系列 `length` 值测试你的程序。

3. 复制 `square` ，并将函数改名为 `polygon` 。增加另外一个名为 `n` 的形参并修改函数体，让它画一个正 n 边形 (n-sided regular polygon)。

   提示：正 n 边形的外角是 $360/n$ 度。

4. 编写一个名为 `circle` 的函数，它接受一个海龟 t 和半径 r 作为形参，然后以合适的边长和边数调用 `polygon` ，画一个近似圆形。用一系列 r 值测试你的函数。

   提示：算出圆的周长，并确保 `length \* n = circumference` 。

5. 完成一个更泛化 (general) 的 `circle` 函数，称其为 `arc` ，接受一个额外的参数 `angle` ，确定画多完整的圆。`angle` 的单位是度，因此当 `angle = 360` 时，`arc` 应该画一个完整的圆。

4.4 Encapsulation | 封装

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

第一个练习要求你将画正方形的代码放到一个函数定义中，然后调用该函数，将海龟作为形参传递给它。下面是一个解法：

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

The innermost statements, `fd` and `lt` are indented twice to show that they are inside the `for` loop, which is inside the function definition. The next line, `square(bob)`, is flush with the left margin, which indicates the end of both the `for` loop and the function definition.

最内层的语句 `fd` 和 `lt` 被缩进两次，以显示它们处在 `for` 循环内，而该循环又在函数定义内。下一行 `square(bob)` 和左边界（left margin）对齐，表示 `for` 循环和函数定义结束。

Inside the function, `t` refers to the same turtle `bob`, so `t.lt(90)` has the same effect as `bob.lt(90)`. In that case, why not call the parameter `bob`? The idea is that `t` can be any turtle, not just `bob`, so you could create a second turtle and pass it as an argument to `square`:

在函数内部，`t` 指的是同一只海龟 `bob` ，所以 `t.lt(90)` 和 `bob.lt(90)` 的效果相同。那么既然这样，为什么不将形参命名为 `bob` 呢？因为 `t` 可以是任何海龟而不仅仅是 `bob` ，也就是说你可以创建第二只海龟，并且将它作为实参传递给 `square` ：

```
alice = Turtle()
square(alice)
```

Wrapping a piece of code up in a function is called **encapsulation**. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you re-use the code, it is more concise to call a function twice than to copy and paste the body!

将一部分代码包装在函数里被称作 封装 (encapsulation) 。封装的好处之一，为这些代码赋予一个名字，这充当了某种文档说明。另一个好处是，如果你重复使用这些代码，调用函数两次比拷贝粘贴函数体要更加简洁！

4.5 Generalization | 泛化

The next step is to add a `length` parameter to `square`. Here is a solution:

下一个练习是给 `square` 增加一个 `length` 形参。下面是一个解法：

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

Adding a parameter to a function is called **generalization** because it makes the function more general: in the previous version, the square is always the same size; in this version it can be any size.

为函数增加一个形参被称作 泛化 (generalization) ，因为这使得函数更通用：在前面的版本中，正方形的边长总是一样的；此版本中，它可以是任意大小。

The next step is also a generalization. Instead of drawing squares, `polygon` draws regular polygons with any number of sides. Here is a solution:

下一个练习也是泛化。泛化之后不再是只能画一个正方形，`polygon` 可以画任意的正多边形。下面是一个方案：

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

This example draws a 7-sided polygon with side length 70.

这个示例代码画了一个边长为 70 的七边形。

If you are using Python 2, the value of `angle` might be off because of integer division. A simple solution is to compute `angle = 360.0 / n`. Because the numerator is a floating-point number, the result is floating point.

如果你在使用 Python 2，`angle` 的值可能由于整型数除法 (integer division) 出现偏差。一个简单的解决办法是这样计算 `angle : angle = 360.0 / n` 。因为分子 (numerator) 是一个浮点数，最终的结果也会是一个浮点数。

When a function has more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. In that case it is often a good idea to include the names of the parameters in the argument list:

如果一个函数有几个数字实参，很容易忘记它们是什么或者它们的顺序。在这种情况下，在实参列表中加入形参的名称是通常是一个很好的办法：

```
polygon(bob, n=7, length=70)
```

These are called **keyword arguments** because they include the parameter names as "keywords" (not to be confused with Python keywords like `while` and `def`).

这些被称作 关键字实参 (keyword arguments) ，因为它们 j 加上了形参名作为 "关键字"（不要和 Python 的关键字搞混了，如 `while` 和 `def` ）。

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

这一语法使得程序的可读性更强。它也提醒了我们实参和形参的工作方式：当你调用函数时，实参被赋给形参。

4.6 Interface design | 接口设计

The next step is to write `circle`, which takes a radius, `r`, as a parameter. Here is a simple solution that uses `polygon` to draw a 50-sided polygon:

下一个练习是编写接受半径 r 作为形参的 `circle` 函数。下面是一个使用 `polygon` 画一个 50 边形的简单解法：

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

The first line computes the circumference of a circle with radius `r` using the formula $2\pi r$. Since we use `math.pi`, we have to import `math`. By convention, `import` statements are usually at the beginning of the script.

函数的第一行通过半径 r 计算圆的周长，公式是 $2\pi r$ 。由于用了 math.pi ，我们需要导入 math 模块。按照惯例，import 语句通常位于脚本的开始位置。

n is the number of line segments in our approximation of a circle, so length is the length of each segment. Thus, polygon draws a 50-sides polygon that approximates a circle with radius r.

n 是我们的近似圆中线段的条数，length 是每一条线段的长度。这样 polygon 画出的就是一个 50 边形，近似一个半径为 r 的圆。

One limitation of this solution is that n is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking n as a parameter. This would give the user (whoever calls circle) more control, but the interface would be less clean.

这种解法的一个局限在于，n 是一个常量，意味着对于非常大的圆，线段会非常长，而对于小圆，我们会浪费时间画非常小的线段。一个解决方案是将 n 作为形参，泛化函数。这将给用户（调用 circle 的人）更多的掌控力，但是接口就不那么干净了。

The **interface** of a function is a summary of how it is used: what are the parameters? What does the function do? And what is the return value? An interface is "clean" if it allows the caller to do what they want without dealing with unnecessary details.

函数的 **接口** (interface) 是一份关于如何使用该函数的总结：形参是什么？函数做什么？返回值是什么？如果接口让调用者避免处理不必要的细节，直接做自己想做的式，那么这个接口就是 "干净的"。

In this example, r belongs in the interface because it specifies the circle to be drawn. n is less appropriate because it pertains to the details of *how* the circle should be rendered.

在这个例子中，r 属于接口的一部分，因为它指定了要画多大的圆。n 就不太合适，因为它是关于 *如何*画圆的细节。

Rather than clutter up the interface, it is better to choose an appropriate value of n depending on circumference:

与其把接口弄乱，不如根据周长 (circumference) 选择一个合适的n值：

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Now the number of segments is an integer near circumference/3, so the length of each segment is approximately 3, which is small enough that the circles look good, but big enough to be efficient, and acceptable for any size circle.

现在线段的数量，是约为周长三分之一的整型数，所以每条线段的长度（大概）是 3，小到足以使圆看上去逼真，又大到效率足够高，对任意大小的圆都能接受。

4.7 Refactoring | 重构

When I wrote circle, I was able to re-use polygon because a many-sided polygon is a good approximation of a circle. But arc is not as cooperative; we can't use polygon or circle to draw an arc.

当我写 circle 程序的时候，我能够复用 polygon ，因为一个多边形是与圆形非常近似。但是 arc 就不那么容易实现了；我们不能使用 polygon 或者 circle 来画一个弧。

One alternative is to start with a copy of polygon and transform it into arc. The result might look like this:

一种替代方案是从复制 polygon 开始，然后将它转化为 arc 。最后的函数看上去可像这样：

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

The second half of this function looks like polygon, but we can't re-use polygon without changing the interface. We could generalize polygon to take an angle as a third argument, but then polygon would no longer be an appropriate name! Instead, let's call the more general function polyline:

该函数的后半部分看上去很像 polygon ，但是在不改变接口的条件下，我们无法复用 polygon 。我们可以泛化 polygon 来接受一个角度作为第三个实参，但是这样 polygon 就不再是一个合适的名字了！让我们称这个更通用的函数为 polyline ：

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Now we can rewrite polygon and arc to use polyline:

现在，我们可以用 polyline 重写 polygon 和 arc ：

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Finally, we can rewrite circle to use arc:

最后，我们可以用 arc 重写 circle：

```
def circle(t, r):
    arc(t, r, 360)
```

This process—rearranging a program to improve interfaces and facilitate code re-use—is called **refactoring**. In this case, we noticed that there was similar code in arc and polygon, so we "factored it out" into polyline.

重新整理一个程序以改进函数接口和促进代码复用的这个过程，被称作 重构 (refactoring) 。在此例中，我们注意到 arc 和 polygon 中有相似的代码，因此，我们 "将它分解出来"（factor it out)，放入 polyline 函数。

If we had planned ahead, we might have written polyline first and avoided refactoring, but often you don't know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

如果我们提前已经计划好了，我们可能会首先写 polyline 函数，避免重构，但是在一个项目开始的时候，你常常并不知道那么多，不能设计好全部的接口。一旦你开始编码后，你才能更好地理解问题。有时重构是一个说明你已经学到某些东西的预兆。

4.8 A development plan | 开发方案

A **development plan** is a process for writing programs. The process we used in this case study is "encapsulation and generalization". The steps of this process are:

开发计划 (development plan) 是一种编写程序的过程。此例中我们使用的过程是 "封装和泛化"。这个过程的具体步骤是：

因此，我们 "将它分解出来" (factor it out)，放入 polyline 函数。

1. Start by writing a small program with no function definitions.
2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name.
3. Generalize the function by adding appropriate parameters.
4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).
5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

1. 从写一个没有函数定义的小程序开始。

2. 一旦该程序运行正常，找出其中相关性强的部分，将它们封装进一个函数并给它一个名字。

3. 通过增加适当的形参，泛化该函数。

4. 重复 13 步，直到你有一些可正常运行的函数。复制粘贴有用的代码，避免重复输入（和重新调试）。

5. 寻找机会通过重构改进程序。例如，如果在多个地方有相似的代码，考虑将它分解到一个合适的通用函数中。

This process has some drawbacks—we will see alternatives later—but it can be useful if you don't know ahead of time how to divide the program into functions. This approach lets you design as you go along.

这个过程也有一些缺点。后面我们将介绍其他替代方案，但是如果你事先不知道如何将程序分解为函数，这是个很有用办法。该方法可以让你一边编程，一边设计。

4.9 docstring | 文档字符串

A **docstring** is a string at the beginning of a function that explains the interface ("doc" is short for "documentation"). Here is an example:

文档字符串 (docstring) 是位于函数开始位置的一个字符串，解释了函数的接口（"doc" 是 "documentation" 的缩写）。下面是一个例子：

```
def polyline(t, n, length, angle):
    """Draws␣n␣line␣segments␣with␣the␣given␣length␣and
␣␣␣␣angle␣(in␣degrees)␣between␣them.␣␣t␣is␣a␣turtle.
␣␣␣␣"""
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

By convention, all docstrings are triple-quoted strings, also known as multiline strings because the triple quotes allow the string to span more than one line.

按照惯例，所有的文档字符串都是三重引号（triple-quoted）字符串，也被称为多行字符串，因为三重引号允许字符串超过一行。

It is terse, but it contains the essential information someone would need to use this function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

它很简要（terse），但是包括了他人使用此函数时需要了解的关键信息。它扼要地说明该函数做什么（不介绍背后的具体细节）。它解释了每个形参对函数的行为有什么影响，以及每个形参应有的类型（如果它不

Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.

写这种文档是接口设计中很重要的一部分。一个设计良好的接口应该很容易解释，如果你很难解释你的某个函数，那么你的接口也许还有改进空间。

4.10 Debugging | 调试

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

接口就像是函数和调用者之间的合同。调用者同意提供合适的参数，函数同意完成相应的工作。

For example, `polyline` requires four arguments: `t` has to be a Turtle; `n` has to be an integer; `length` should be a positive number; and `angle` has to be a number, which is understood to be in degrees.

例如，`polyline` 函数需要 4 个实参：`t` 必须是一个 `Turtle` ；`n` 必须是一个整型数；`length` 应该是一个正数；`angle` 必须是一个数，单位是度数。

These requirements are called **preconditions** because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are **postconditions**. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the Turtle or making other changes).

这些要求被称作 *先决条件* (preconditions) ，因为它们应当在函数开始执行之前成立（true）。相反，函数结束时的条件是 *后置条件* (postconditions) 。后置条件包括函数预期的效果（如画线段）以及任何其他附带效果（如移动 `Turtle` 或者做其它改变）。

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn't work correctly, the bug is in the caller, not the function.

先决条件由调用者负责满足。如果调用者违反一个（已经充分记录文档的！）先决条件，导致函数没有正确工作，则故障（bug）出现在调用者一方，而不是函数。

If the preconditions are satisfied and the postconditions are not, the bug is in the function. If your pre- and postconditions are clear, they can help with debugging.

如果满足了先决条件，没有满足后置条件，故障就在函数一方。如果你的先决条件和后置条件都很清楚，将有助于调试。

4.11 Glossary | 术语表

method: A function that is associated with an object and called using dot notation.

方法（method）： 与对象相关联的函数，并使用点标记法（dot notation）调用。

loop: A part of a program that can run repeatedly.

循环（loop）： 程序中能够重复执行的那部分代码。

encapsulation: The process of transforming a sequence of statements into a function definition.

封装（encapsulation）： 将一个语句序列转换成函数定义的过程。

generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

泛化（generalization）： 使用某种可以算是比较通用的东西（像变量和形参），替代某些没必要那么具体的东西（像一个数字）的过程。

keyword argument: An argument that includes the name of the parameter as a "keyword".

关键字实参（keyword argument）： 包括了形参名称作为“关键字”的实参。

interface: A description of how to use a function, including the name and descriptions of the arguments and return value.

接口（interface）： 对如何使用一个函数的描述，包括函数名、参数说明和返回值。

refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.

重构（refactoring）： 修改一个正常运行的函数，改善函数接口及其他方面代码质量的过程。

development plan: A process for writing programs.

开发计划（development plan）： 编写程序的一种过程。

docstring: A string that appears at the top of a function definition to document the function's interface.

文档字符串（docstring）： 出现在函数定义顶部的一个字符串，用于记录函数的接口。

precondition: A requirement that should be satisfied by the caller before a function starts.

先决条件（preconditions）： 在函数运行之前，调用者应该满足的要求。

postcondition: A requirement that should be satisfied by the function before it ends.

后置条件（postconditions）： 函数终止之前应该满足的条件。

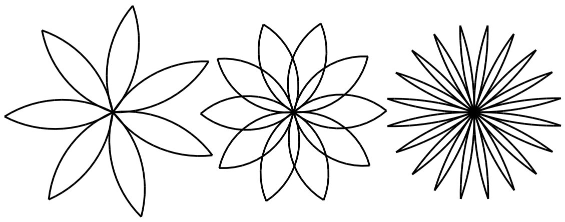

图 4.1: Turtle flowers.

4.12 Exercises | 练习

Exercise 4.1. *Download the code in this chapter from* *http://thinkpython2.com/code/polygon.py*.

可从 *http://thinkpython2.com/code/polygon.py* 下载本章的代码。

1. *Draw a stack diagram that shows the state of the program while executing* `circle(bob, radius)`. *You can do the arithmetic by hand or add* `print` *statements to the code.*
2. *The version of* `arc` *in Section 4.7 is not very accurate because the linear approximation of the circle is always outside the true circle. As a result, the Turtle ends up a few pixels away from the correct destination. My solution shows a way to reduce the effect of this error. Read the code and see if it makes sense to you. If you draw a diagram, you might see how it works.*

1. 画一个执行 `circle(bob, radius)` 时的堆栈图 (*stack diagram*)，说明程序的各个状态。你可以手动进行计算，也可以在代码中加入打印语句。
2. *Section 4.7* 中给出的 `arc` 函数版本并不太精确，因为圆形的线性近似（*linear approximation*）永远处在真正的圆形之外。因此，`Turtle` 总是和正确的终点相差几个像素。我的答案中展示了降低这个错误影响的一种方法。阅读其中的代码，看看你是否能够理解。如果你画一个堆栈图的话，你可能会更容易明白背后的原理。

Exercise 4.2. *Write an appropriately general set of functions that can draw flowers as in Figure 4.1.*

编写比较通用的一个可以画出像图 4-1 中那样花朵的函数集。

Solution: *http://thinkpython2.com/code/flower.py*, *also requires* *http://thinkpython2.com/code/polygon.py*.

参考答案，需要使用这个模块。

Exercise 4.3. *Write an appropriately general set of functions that can draw shapes as in Figure 4.2.*

编写比较通用的一个可以画出图 4-2 中那样图形的函数集，。

Solution: *http://thinkpython2.com/code/pie.py*.

参考答案

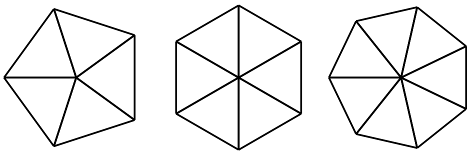

图 4.2: Turtle pies.

Exercise 4.4. *The letters of the alphabet can be constructed from a moderate number of basic elements, like vertical and horizontal lines and a few curves. Design an alphabet that can be drawn with a minimal number of basic elements and then write functions that draw the letters.*

字母表中的字母可以由少量基本元素构成，例如竖线和横线，以及一些曲线。设计一种可用由最少的基本元素绘制出的字母表，然后编写能画出各个字母的函数。

You should write one function for each letter, with names `draw_a`*,* `draw_b`*, etc., and put your functions in a file named* `letters.py`*. You can download a "turtle typewriter" from* *http://thinkpython2.com/code/typewriter.py* *to help you test your code.*

你应该为每个字母写一个函数，起名为 *draw_a*，*draw_b* 等等，然后将你的函数放在一个名为 *letters.py* 的文件里。你可以从 这里 下载一个"海龟打字员"来帮你测试代码。

You can get a solution from *http://thinkpython2.com/code/letters.py**; it also requires* *http://thinkpython2.com/code/polygon.py**.*

你可以在 这里找到参考答案；这个解法还要求使用 这个模块 。

Exercise 4.5. *Read about spirals at* *http://en.wikipedia.org/wiki/Spiral**; then write a program that draws an Archimedian spiral (or one of the other kinds). Solution:* *http://thinkpython2.com/code/spiral.py**.*

阅读关于 螺线 (*spiral*) 的相关知识；然后编写一个绘制阿基米德螺线（或者其他种类的螺线）的程序。

参考答案

第五章　条件和递归

本章中心议题是根据程序的状态执行不同命令的 if 语句。在这之前我们先介绍两个新的运算符: *地板除法* (floor division) [1] 和*求余* (modulus) 。

5.1　地板除法和求余

地板除运算符 (floor division operator) 为 `//` 即先做除法，然后将结果向下保留到整数。例如，如果一部电影时长 105 分钟，你可能想知道这代表着多少小时。传统的除法操作会返回一个浮点数：

```
>>> minutes = 105
>>> minutes / 60
1.75
```

但是，以小时做单位时我们通常不会写出小数部分。地板除法丢弃除法运算结果的小数部分，返回整数个小时:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

如果你希望得到余数，你可以从除数中减去一个小时也就是 60 分钟:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

另一个方法就是使用*求余运算符* (modulus operator)，`%`，它会将两个数相除，返回余数。

```
>>> remainder = minutes % 60
>>> remainder
45
```

[1]译注：向下取整的除法。

求余运算符比看起来更加有用。例如，你可以查看一个数是否可以被另一个数整除——如果 x % y 的结果是 0，那么 x 能被 y 整除。

此外，你也能获得一个数的最右边一位或多位的数字。例如，x % 10 返回 x 最右边一位的数字（十进制）。类似地，x % 100 返回最后两位数字。

如果你正在使用 Python 2, 那么除法就会和前面的介绍有点不同。除法运算符 / 在被除数和除数都是整数的时候，会进行地板除，但是当被除数和除数中任意一个是浮点数的时候，则进行浮点数除法。[2]

5.2 布尔表达式

布尔表达式 (boolean expression) 的结果要么为**真**要么为**假**。下面的例子使用 == 运算符。它比较两个运算数，如果它们相等，则结果为 True，否则结果为 False。

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True 和 False 是属于 bool 类型的特殊值；它们不是字符串。

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

== 运算符是关系运算符 (relational operators) 之一；其他关系运算符还有：

```
      x != y               # x is not equal to y
      x > y                # x is greater than y
      x < y                # x is less than y
      x >= y               # x is greater than or equal to y
      x <= y               # x is less than or equal to y
```

虽然这些运算符对你来说可能很熟悉，但是 Python 的符号与数学符号不相同。一个常见的错误是使用单独一个等号 (=) 而不是双等号 (==)。请记住，= 是赋值运算符，== 是关系运算符。没有类似 =< 或 => 的东西。

5.3 Logical operators | 逻辑运算符

有三个逻辑运算符 (logical operators)：and、or 和 not。这些运算符的含义和它们在英语的意思相似。例如，x > 0 and x < 10 只在 x 大于 0 **并且** 小于 10 时为真。

[2]译注：在 Python3 中，无论任何类型都会保持小数部分。

n%2 == 0 or n%3 == 0 中如果一个或两个 条件为真，那么整个表达式即为真。也就是说，如果数字n 能被 2 或者 3 整除，则为真。

最后，not 运算符对一个布尔表达式取反，因此，如果 x > y 为假，也就是说 x 小于或等于 y，则 not (x > y) 为真。

严格来讲，逻辑运算符的运算数应该是布尔表达式，但是 Python 并不严格要求。任何非 0 的数字都被解释成为真 (True)。

```
>>> 42 and True
True
```

这种灵活性很有用，但有一些细节可能容易令人困惑。你可能需要避免这种用法（除非你知道你正在做什么）。

5.4 有条件执行

为了写出有用的程序，我们几乎总是需要能够检测条件，并相应地改变程序行为。条件语句 (Conditional statements) 给予了我们这一能力。最简单的形式是 if 语句：

```
if x > 0:
    print('x␣is␣positive')
```

if 之后的布尔表达式被称作条件 (condition) 。如果它为真，则缩进的语句会被执行。如果不是，则什么也不会发生。

if 语句和函数定义有相同的结构：一个语句头跟着一个缩进的语句体。类似的语句被称作复合语句 (compound statements) 。

语句体中可出现的语句数目没有限制，但是至少得有一个。有时候，一条语句都没有的语句体也是有用的（通常是为你还没写的代码占一个位子）。这种情况下，你可以使用 pass 语句，它什么也不做。

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

5.5 二选一执行

A second form of the if statement is “alternative execution”, in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

if 语句的第二种形式是 “二选一执行” (alternative execution) ，此时有两个可能的选择，由条件决定执行哪一个。语法看起来是这样：

```
if x % 2 == 0:
    print('x␣is␣even')
else:
    print('x␣is␣odd')
```

如果 x 除以 2 的余数是 0，那么我们知道 x 是偶数，然后程序会打印相应的信息。如果条件为假，则执行第二部分语句。由于条件要么为真要么为假，两个选择中只有一个会被执行。这些选择被称作 ** 分支（branches）** ，因为它们是执行流程的分支。

5.6 链式条件

有时有超过两个可能的情况，于是我们需要多于两个的分支。表示像这样的计算的方法之一是链式条件 (chained conditional)：

```
if x < y:
    print('x␣is␣less␣than␣y')
elif x > y:
    print('x␣is␣greater␣than␣y')
else:
    print('x␣and␣y␣are␣equal')
```

`elif` 是 “else if” 的缩写。同样地，这里只有一个分支会被执行。`elif` 语句的数目没有限制。如果有一个 `else` 从句，它必须是在最后，但这个语句并不是必须。

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

程序将按顺序逐个检测条件，如果第一个为假，则检测下一个，以此类推。如果它们中有一个为真，相应的分支被执行，并且结束语句。即便有不止一个条件为真，也只执行第一个为真的分支。

5.7 嵌套条件

一个条件可以嵌到另一个里面。我们可以这样写前一节的例子：

```
if x == y:
    print('x␣and␣y␣are␣equal')
else:
    if x < y:
        print('x␣is␣less␣than␣y')
    else:
        print('x␣is␣greater␣than␣y')
```

外层的条件 (outer conditional) 包含两条分支。第一个分支包括一条简单的语句。第二个分支包括一个 `if` 语句，它又有两条子分支。这两条子分支都是简单的语句，当然它们也可以再嵌入条件语句。

虽然语句的缩进使得结构很明显，但是仍然很难快速地阅读嵌套条件 (nested conditionals) 。当你可以的时候，避免使用嵌套条件是个好办法。

逻辑运算符通常是一个简化嵌套条件语句的方法。例如，我们可以用一个单一条件重写下面的代码：

```
if 0 < x:
    if x < 10:
        print('x␣is␣a␣positive␣single−digit␣number.')
```

只有通过了两个条件检测的时候，`print` 语句才被执行，因此我们可以用 `and` 运算符得到相同的效果：

```
if 0 < x and x < 10:
    print('x␣is␣a␣positive␣single−digit␣number.')
```

对于这样的条件，Python 提供了一种更加简洁的写法。

```
if 0 < x < 10:
    print('x␣is␣a␣positive␣single−digit␣number.')
```

5.8 递归

一个函数调用另一个是合法的；一个函数调用它自己其实也是合法的。这样做的好处也许看上去不那么明显，但它实际上它是程序最神奇的魔法之一。例如，下面这个函数：

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n−1)
```

如果 `n` 是 0 或负数，程序输出单词 “Blastoff!”。否则，它输出 n 然后调用一个名为 `countdown` 的函数—即它自己 — 传递 `n−1` 作为实参。

如果我们像这样调用该函数会发生什么呢？

```
>>> countdown(3)
```

countdown 开始以 n=3 执行，由于 n 大于 0，它输出值 3，然后调用它自己...

> countdown 开始以 n=2 执行，由于 n 大于 0，它输出值 2，然后调用它自己...
>
> > countdown 开始以 n=1 执行，既然 n 大于 0，它输出值 1，然后调用它自己...
> >
> > > countdown 开始以 n=0 执行，由于 n 不大于 0，它输出单词 "Blastoff!"，然后返回。
> >
> > 获得 n=1 的 countdown 返回。
>
> 获得 n=2 的 countdown 返回。

获得 n=3 的 countdown 返回。

然后回到 __main__ 中。因此整个输出类似于：

```
3
2
1
Blastoff!
```

一个调用它自己的函数被称为*递归的* (recursive)；这种过程被称作*递归* (recursion)。

再举一例，我们可以写一个函数，其打印一个字符串 n 次。

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

如果 n <= 0，return语句 退出函数。执行流程马上返回到调用者，剩下的语句不会被执行。

函数的其余部分和 countdown 相似：它打印 s 的值，然后调用自身打印 s $n-1$ 次。因此，输出的行数是 1 + (n - 1)，加起来是 n。

对于像这样简单的例子，使用 for 循环可能更容易。但是我们后面将看到一些用 for 循环很难写，用递归却很容易的例子，所以早点儿开始学习递归有好处。

5.9 递归函数的堆栈图

在 3.9 小节中，我们用堆栈图表示了一个函数调用期间程序的状态。这种图也能帮我们理解递归函数。

每当一个函数被调用时，Python 生成一个新的栈帧，用于保存函数的局部变量和形参。对于一个递归函数，在堆栈上可能同时有多个栈帧。

图 5.2 展示了一个以 n = 3 调用 countdown 的堆栈图。

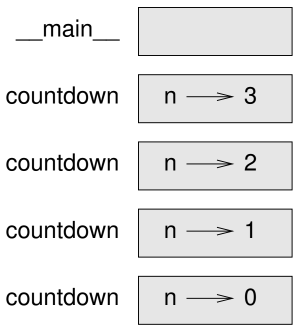

图 5.1: Stack diagram.

图 5.2: 堆栈图。

通常，堆栈的顶部是 `__main__` 栈帧。因为我们在 `__main__` 中没有创建任何变量，也没有传递任何实参给它，所以它是空的。

对于形参 `n`，四个 `countdown` 栈帧有不同的值。`n=0` 的栈底，被称作*基础情形* (base case) 。它不再进行递归调用了，所以没有更多的栈帧了。

接下来练习一下，请画一个以 `s = 'Hello'` 和 `n=2` 调用 `print_n` 的堆栈图。写一个名为 `do_n` 的函数，接受一个函数对象和一个数 `n` 作为实参，能够调用指定的函数 `n` 次。

5.10 无限递归

如果一个递归永不会到达基础情形，它将永远进行递归调用，并且程序永远不会终止。这被称作*无限递归* (infinite recursion)，通常这不是一个好主意。下面是一个最简单的无限递归程序：

```
def recurse():
    recurse()
```

在大多数编程环境里，一个具有无限递归的程序并非永远不会终止。当达到最大递归深度时，Python 会报告一个错误信息：

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

此回溯比我们在前面章节看到的长一些。当错误出现的时候，在堆栈上有 1000 个*递归栈帧*！

如果你不小心遇到了无限递归，检查你的函数，确保基础情形没有继续调用递归。同时如果确实有基础情形，请检查基础情形是不是能够出现这种情形。

5.11 键盘输入

到目前为止，我们所写的程序都不接受来自用户的输入。每次它们都只是做相同的事情。

Python 提供了一个内建函数 `input` ，可以暂停程序运行，并等待用户输入。当用户按下回车键 (Return or Enter)，程序恢复执行，`input` 以字符串形式返回用户键入的内容。在 Python 2 中，这个函数的名字叫 `raw_input` 。

```
>>> text = input()
What are you waiting for?
>>> text
What are you waiting for?
```

在从用户那儿获得输入之前，打印一个提示告诉用户输入什么是个好办法。`input` 接受提示语作为实参。

```
>>> name = input('What...is␣your␣name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
Arthur, King of the Britons!
```

提示语最后的 `\n` 表示一个新行 (newline)，它是一个特别的字符，会造成换行。这也是用户的输入出现在提示语下面的原因。

如果你期望用户键入一个整型数，那么你可以试着将返回值转化为 `int` ：

```
>>> prompt = 'What...is␣the␣airspeed␣velocity␣of␣an␣unladen␣swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

但是，如果用户输入不是数字构成的字符串，你会获得一个错误：

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

我们后面将介绍处理这类错误的方法。

5.12 调试

当出现语法错误和运行时错误的时候，错误信息中会包含了很多的信息，但是信息量有可能太大。通常，最有用的部分是：

- 是哪类错误，以及
- 在哪儿出现。

语法错误通常很容易被找到，但也有一些需要注意的地方。空白分隔符错误很棘手，因为空格和制表符是不可见的，而且我们习惯于忽略它们。

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

在这个例子中，问题在于第二行缩进了一个空格。但是错误信息指向 y，这是个误导。通常，错误信息指向发现错误的地方，但是实际的错误可能发生在代码中更早的地方，有时在前一行。

运行时错误也同样存在这个问题。假设你正试图计算分贝信噪比。公式是 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$ 。在 Python 中，你可能会写出这样的代码：

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

但是，当你运行它的时候，你却获得一个异常。

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

该错误信息指向第 5 行，但是那一行没什么错误。为了找到真正的错误，打印 `ratio` 的值也许会有用，结果发现它实际上是 0。那么问题是在第 4 行，使用了地板除而不是浮点数除法。

你需要花些时间仔细阅读错误信息，但不要轻易地相信错误信息的提示都是准确的。

5.13 术语表

地板除法: 一个操作符，用 `//` 表示，表示对两个数做除法同时向 0 取整。

求余运算符: 一个运算符，用百分号 `%` 表示，返回两个数相除的余数。

布尔表达式: 一个值要么为真要么为假的表达式。

关系运算符: 对其运算符进行比较的运算符：`==`，`!=`，`>`，`<`，`>=`，`<=`。

逻辑运算符: 将布尔表达式组合在一起的运算符: `and`，`or`，和 `not`。

条件语句: 一段根据某个条件决定程序执行流程的语句。

条件: 决定哪个分支会被执行的布尔表达式。

复合语句: 由语句头和语句体组成的语句。语句头以: 结尾，语句体相对语句头缩进。

分支: 条件语句中的选择性语句序列。

链式条件: 由一系列替代分支组成的条件语句。

嵌套条件: 出现另一个条件语句某个分支中的条件语句。

返回语句： 结束函数执行并且将结果返回给调用者的语句。

递归: 调用正在执行的函数本身的过程。

基本情形: 在递归函数中，不进行递归调用的条件分支。

无限递归: 没有基本情形或者无法出现基本情形的递归函数。最终无限递归会导致运行时错误。

5.14 练习

Exercise 5.1. `time` 模块提供了一个可以返回当前格林威治标准时间的函数，名字也是 time。这里的格林威治标准时间用纪元 (“the epoch”) 以来的秒数表示，纪元是一个任意的参考点。在 Unix 系统中，纪元是 1970 年 1 月 1 日。

```
>>> import time
>>> time.time()
1437746094.5735958
```

请写一个脚本读取当前时间，并且将其转换为纪元以来经过了多少天、小时、分钟和秒。

Exercise 5.2. 费马大定理 (Fermat’s Last Theorem) 称，没有任何整型数 a、b 和 c 能够使：

$$a^n + b^n = c^n$$

对于任何大于 2 的 n 成立。

1. 写一个名为check_fermat 的函数，接受四个形参 — a，b，c 以及n — 检查费马大定理是否成立。如果n 大于 2 且等式

$$a^n + b^n = c^n$$

成立，程序应输出 “Holy smokes, Fermat was wrong!”。否则程序应输出 “No, that doesn’t work.”。

2. 写一个函数提示用户输入a，b，c以及n 的值，将它们转换成整型数，然后使用check_fermat 检查他们是否会违反了费马大定理。

Exercise 5.3. 如果你有三根棍子，你有可能将它们组成三角形，也可能不行。比如，如果一根棍子是 12 英寸长，其它两根都是 1 英寸长，显然你不可能让两根短的在中间接合。对于任意三个长度，有一个简单的测试能验证它们能否组成三角形：

> 如果三个长度中的任意一个超过了其它二者之和，就不能组成三角形。否则，可以组成。(如果两个长度之和等于第三个，它们就组成所谓 “‘退化的” 三角形。)

1. 写一个名为is_triangle 的函数，其接受三个整数作为形参，能够根据给定的三个长度的棍子能否构成三角形来打印 “Yes” 或 “No”。

2. 写一个函数，提示用户输入三根棍子的长度，将它们转换成整型数，然后使用is_triangle 检查给定长度的棍子能否构成三角形。

Exercise 5.4. 下面程序的输出是什么？画出展示程序每次打印输出时的堆栈图。

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. 如果你这样调用函数：recurse(-1,0) ，会有什么结果？

2. 请写一个文档字符串，解释调用该函数时需要了解的全部信息（仅此而已）。

后面的习题要用到第 四章中的 turtle：

Exercise 5.5. 阅读如下的函数，看看你能否看懂它是做什么的。然后运行它（见第四章的例子）。

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

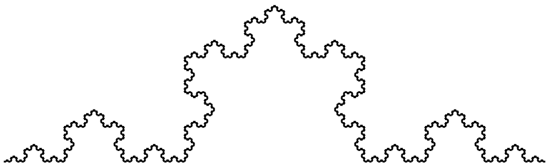
图 5.3: 科赫曲线。

Exercise 5.6. 科赫曲线 (Koch Curve) 是一个看起来类似图 5.3的不规则碎片几何体 (fractal)。要画一个长度为 x 的科赫曲线，你只需要：

1. 画一个长度为 $x/3$ 的科赫曲线。
2. 左转 60 度。
3. 画一个长度为 $x/3$ 的科赫曲线。
4. 右转 60 度。
5. 画一个长度为 $x/3$ 的科赫曲线。
6. 左转 60 度。
7. 画一个长度为 $x/3$ 的科赫曲线。

例外情况是 x 小于 3 的情形：此时，你只需要画一道长度为 x 的直线。

1. 写一个名为`koch`的函数，接受一个海龟和一个长度作为形参，然后使用海龟画一条给定长度的科赫曲线。
2. 写一个名为`snowflake`的函数，画出三条科赫曲线，构成雪花的轮廓。
   参考答案
3. 科赫曲线能够以多种方式泛化。
   点击此处查看例子，并实现你最喜欢的那种方式。

第六章　有返回值的函数

许多我们前面使用过的 Python 函数都会产生返回值，如数学函数。但目前我们所写的函数都是空函数 (void): 它们产生某种效果，像打印一个值或是移动乌龟，但是并没有返回值。在本章中，你将学习如何写一个有返回值的函数。

6.1　返回值

调用一个有返回值的函数会生成一个返回值，我们通常将其赋值给某个变量或是作为表达式的一部分。

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

目前我们所写的函数都是空函数。泛泛地来看，它们没有返回值；更准确地说，它们的返回值是 `None` 。

本章中，我们（终于）要开始写有返回值的函数了。第一个例子是 `area` ，返回给定半径的圆的面积。

```
def area(radius):
    a = math.pi * radius**2
    return a
```

我们之前已经见过 `return` 语句，但在有返回值的函数中，`return` 语句包含一个表达式。条语句的意思是：“马上从该函数返回，并使用接下来的表达式作为返回值。” 此表达式可以是任意复杂的，因此我们可以将该函数写得更简洁些：

```
def area(radius):
    return math.pi * radius**2
```

另一方面，像 a 这样的临时变量 (temporary variables) 能使调试变得更简单。

有时，在条件语句的每一个分支内各有一个返回语句会很有用：

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

因为这些 `return` 语句在不同的条件内，最后只有**一个**会被执行。

一旦一条返回语句执行，函数则终止，不再执行后续的语句。出现在某条 return 语句之后的代码，或者在执行流程永远不会到达之处的代码，被称为*死代码* (dead code)。

在一个有返回值的函数中，最好保证程序执行的每一个流程最终都会碰到一个 `return` 语句。例如：

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

这个函数是有问题的。原因是如果 `x` 恰好是 0，则没有条件为真，函数将会在未执行任何 `return` 语句的情况下终止。如果函数按照这种执行流程执行完毕，返回值将是 `None`，这可不是 0 的绝对值。

```
>>> absolute_value(0)
None
```

顺便说一下，Python 提供了一个的内建函数 `abs` 用来计算绝对值。

我们来做个练习，写一个比较函数 `compare`，接受两个值 `x` 和 `y`。如果 `x > y`，则返回 `1`；如果 `x == y`，则返回 `0`；如果 `x < y`，则返回 `-1`。

6.2 增量式开发

随着你写的函数越来越大，你在调试上花的时候可能会越来越多。

为了应对越来越复杂的程序，你可以开始尝试叫作*增量式开发* (incremental development) 的方法。增量式开发的目标，是通过每次只增加和测试少量代码，来避免长时间的调试。

举个例子，假设你想计算两个给定坐标点 (x_1, y_1) 和 (x_2, y_2) 之间的距离。根据毕达哥拉斯定理[1] (the Pythagorean theorem)，二者的距离是：

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

第一步要考虑的是在 Python 中，距离函数看起来会是什么样。换句话说，输入（形参）和输出（返回值）是什么？

[1]译注：即勾股定理

本例中，输入是可以用 4 个数表示的两个点。返回值是距离，用浮点数表示。

现在你就可以写出此函数的轮廓了：

```
def distance(x1, y1, x2, y2):
    return 0.0
```

显然，此版本不能计算距离；它总是返回 0 。但是在语法上它是正确的，并且能运行，这意味着你可以在使它变得更复杂之前测试它。

用样例实参调用它来进行测试。

```
>>> distance(1, 2, 4, 6)
0.0
```

我选择的这些值，可以使水平距离为 3 ，垂直距离为 4 ；这样结果自然是 5（构成一个勾三股四弦五的直角三角形）。测试一个函数时，知道正确的答案是很有用的。

此时我们已经确认这个函数在语法上是正确的，我们可以开始往函数体中增加代码。下一步合理的操作，应该是求 $x_2 - x_1$ 和 $y_2 - y_1$ 这两个差值。下一个版本在临时变量中存储这些值并打印出来。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx␣is', dx)
    print('dy␣is', dy)
    return 0.0
```

如果这个函数正常运行，它应该显示 `dx is 3` 以及 `dy is 4` 。这样的话我们就知道函数获得了正确的实参并且正确执行了第一步计算。如果不是，也只要检查几行代码。

下一步我们计算 `dx` 和 `dy` 的平方和。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared␣is:␣', dsquared)
    return 0.0
```

再一次运行程序并检查结果（应该是 25 ）。最后，你可以使用 `math.sqrt` 计算并返回结果。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

如果其正确运行的话，你就成功了。否则你可能想在 return 语句前打印结果检查一下。

该函数的最终版不会在运行时显示任何东西，仅仅返回一个值。我们之前写的 print 语句在调试时是很有用的，不过在函数能够正确运行之后，你就该删了它们。我们称这样的代码为*脚手架代码* (scaffolding) ，因为它对程序的构建很有用，但不是最终产品的一部分。

当你刚开始的时候，最好每次只加入一两行代码。随着经验见长，你会发现自己可以编写、调试更大的代码块了。无论哪种方式，增量式开发都能节省你大量的调试时间。

这种开发方式的关键是：

1. 从一个能运行的程序开始，并且每次只增加少量改动。无论你何时遇到错误，都能够清楚定位错误的源头。
2. 用临时变量存储中间值，这样你就能显示并检查它们。
3. 一旦程序正确运行，你要删除一些脚手架代码，或者将多条语句组成复合表达式，但是前提是不会影响程序的可读性。

我们来做个练习：运用增量开发方式，写一个叫作 hypotenuse 的函数，接受直角三角形的两直角边长作为实参，返回该三角形斜边的长度。记录下你开发过程中的每一步。

6.3　组合

你现在应该已经猜到了，你可以从一个函数内部调用另一个函数。作为示例，我们接下来写一个函数，接受两个点为参数，分别是圆心和圆周上一点，然后计算圆的面积。

假设圆心坐标存储在变量 xc 和 yc 中，圆周上的点的坐标存储在 xp 和 yp 中。第一步是计算圆半径，也就是这两个点的距离。我们刚写的 distance 函数就可以计算距离：

```
radius = distance(xc, yc, xp, yp)
```

下一步是用得到的半径计算圆面积；我们也刚写了这样的函数：

```
result = area(radius)
```

将这些步骤封装在一个函数中，可以得到下面的函数：

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

临时变量 radius 和 result 对于开发调试很有用的，但是一旦函数正确运行了，我们可以通过合并函数调用，将程序变得更简洁：

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 布尔函数

函数可以返回布尔值 (booleans)，通常对于隐藏函数内部的复杂测试代码非常方便。例如：

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

通常布尔函数名听起来像是一个疑问句，回答不是 Yes 就是 No，`is_divisible` 通过返回 `True` 或 `False` 来表示 `x` 是否可以被 `y` 整除。

请看下面的示例：

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

`==` 运算符的结果是布尔值，因此我们直接返回它，让代码变得更简洁。

```
def is_divisible(x, y):
    return x % y == 0
```

布尔函数通常被用于条件语句中：

```
if is_divisible(x, y):
    print('x␣is␣divisible␣by␣y')
```

很容易写出下面这样的代码：

```
if is_divisible(x, y) == True:
    print('x␣is␣divisible␣by␣y'
```

但这里的比较是多余的。

我们来做个练习：写一个函数 `is_between(x, y, z)`，如果 $x \le y \le z$ 返回 `True` 否则返回 `False`。

6.5 再谈递归

我们目前只介绍了 Python 中一个很小的子集，但是当你知道这个子集已经是一个完备的 编程语言，你可能会觉得很有意思。这意味任何能被计算的东西都能用这个语言表达。有史以来所有的程序，你都可以仅用目前学过的语言特性重写 (事实上，你可能还需要一些命令来控制鼠标、磁盘等设备，但仅此而已)。

阿兰·图灵 (Alan Turing) 首次证明了这种说法的正确性，这是一项非凡的工作。他是首批计算机科学家之一（一些人认为他是数学家，但很多早期的计算机科学家也是出身于数学家）。相应地，这被称为图灵理论。关于图灵理论更完整（和更准确）的讨论，我推荐 Michael Sipser 的书《*Introduction to the Theory of Computation*》。

为了让你明白能用目前学过的工具做什么，我们将计算一些递归定义的数学函数。递归定义类似循环定义，因为定义中包含一个对已经被定义的事物的引用。一个纯粹的循环定义并没有什么用：

漩涡状： 一个用以描述漩涡状物体的形容词。

如果你看到字典里是这样定义的，你大概会生气。另一方面，如果你查找用 ! 符号表示的阶乘函数的定义，你可能看到类似下面的内容：

$$
\begin{aligned}
0! &= 1 \\
n! &= n(n-1)!
\end{aligned}
$$

该定义指出 0 的阶乘是 1 ，任何其他值 n 的阶乘是 n 乘以 $n-1$ 的阶乘。

所以 3! 的阶乘是 3 乘以 2! ，它又是 2 乘以 1! ，后者又是 1 乘以 0! 。放到一起，3! 等于 3 乘以 2 乘以 1 乘以 1 ，结果是 6 。

如果你可以递归定义某个东西，你就可以写一个 Python 程序计算它。第一步是决定应该有哪些形参。在此例中 `factorial` 函数很明显接受一个整型数：

```
def factorial(n):
```

如果实参刚好是 0 ，我们就返回 1 ：

```
def factorial(n):
    if n == 0:
        return 1
```

否则，就到了有意思的部分，我们要进行递归调用来找到 $n-1$ 的阶乘然后乘以 n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

程序的执行流程和第 5.8节中的 `countdown` 类似。如果我们传入参数的值是 3 ：

由于 3 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...

　　由于 2 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...

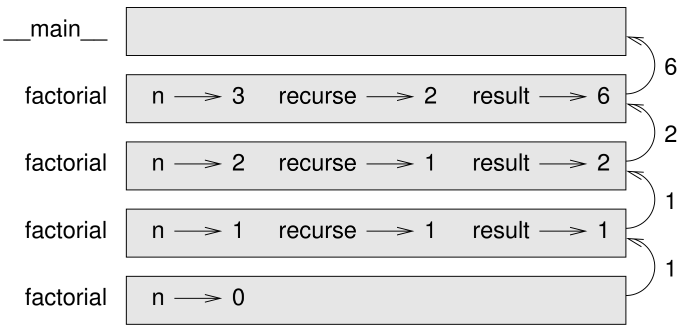
__main__
factorial
n → 3 recurse → 2 result → 6
factorial
n → 2 recurse → 1 result → 2
factorial
n → 1 recurse → 1 result → 1
factorial
n → 0
6
2
1
1

图 6.1: Stack diagram.

> 由于 1 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...
>
> > 由于 0 等于 0，我们执行第一个分支并返回 1，不再进行任何递归调用。
>
> 返回值 1 与 n （其为 1）相乘，并返回结果。
>
> 返回值 1 与 n （其为 2）相乘，并返回结果。

返回值 2 与 n （其为 3）相乘，而结果 6 也就成为一开始那个函数调用的返回值。

图 6.1 显示了该函数调用序列的堆栈图看上去是什么样子。

图中的返回值被描绘为不断被传回到栈顶。在每个栈帧中，返回值就是结果值，即是 `n` 和 `recurse` 的乘积。

最后一帧中，局部变量 `recurse` 和 `result` 并不存在，因为生成它们的分支并没有执行。

6.6 信仰之跃

跟随程序执行流程是阅读程序代码的一种方法，但它可能很快会变得错综复杂。有另外一种替代方法，我称之为"信仰之跃"。当你遇到一个函数调用时，不再去跟踪执行流程，而是 ** 假设 ** 这个函数正确运行并返回了正确的结果。

事实上，当你使用内建函数时，你已经在实践这种方法了。当你调用 `math.cos` 或 `math.exp` 时，你并没有检查那些函数的函数体。你只是假设了它们能用，因为编写这些内建函数的人都是优秀的程序员。

当你调用一个自己写的函数时也是一样。例如，在 6.4 节中，我们写了一个 `is_divisible` 函数来判断一个数能否被另一个数整除。通过对代码的检查，一旦我们确信这个函数能够正确运行 — 我们就能不用再查看函数体而直接使用了。

递归程序也是这样。当你遇到递归调用时，不用顺着执行流程，你应该假设每次递归调用能够正确工作（返回正确的结果），然后问你自己，"假设我可以找到:math:'n-1' 的阶乘，我可以找到 n 的阶乘吗？很明显你能，只要再乘以 n 即可。

当然，在你没写完函数的时候就假设函数正确工作有一点儿奇怪，但这也是为什么这被称作信仰之跃了！

6.7 再举一例

除了阶乘以外，使用递归定义的最常见数学函数是 fibonacci （斐波那契数列），其定义见 http://en.wikipedia.org/wiki/Fibonacci_number：

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

翻译成 Python ，看起来就像这样：

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

这里，如果你试图跟踪执行流程，即使是相当小的 n ，也足够你头疼的。但遵循信仰之跃这种方法，如果你假设这两个递归调用都能正确运行，很明显将他们两个相加就是正确结果。

6.8 检查类型

如果我们将 1.5 作为参数调用阶乘函数 (factorial) 会怎样？

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

看上去像是一个无限循环。但那是如何发生的？函数的基础情形是 n == 0 。但是如果 n 不是一个整型数呢，我们会错过 基础情形，永远递归下去。

在第一次递归调用中，n 的值是 0.5 。下一次，是 −0.5 。自此它会越来越小，但永远不会是 0 。

我们有两个选择。我们可以试着泛化 factorial 函数，使其能处理浮点数，或者我们可以让 factorial 检查实参的类型。第一个选择被称作 gamma 函数，它有点儿超过本书的范围了。所以我们将采用第二种方法。

我们可以使用内建函数 isinstance 来验证实参的类型。同时，我们也可以确保该实参是正数：

```
def factorial (n):
    if not isinstance(n, int):
        print('Factorial␣is␣only␣defined␣for␣integers.')
        return None
    elif n < 0:
        print('Factorial␣is␣not␣defined␣for␣negative␣integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

第一个基础情形处理非整型数；第二个处理负整型数。在这两个情形中，程序打印一条错误信息，并返回 None 以指明出现了错误：

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

If we get past both checks, we know that n is positive or zero, so we can prove that the recursion terminates.

如果我们通过了这两个检查，那么我们知道 n 是一个正数或 0 ，因此我们可以证明递归会终止。

此程序演示了一个有时被称作*监护人* (guardian) 的模式。前两个条件扮演监护人的角色，避免接下来的代码使用引发错误的值。监护人使得验证代码的正确性成为可能。

在反向查找 (Reverse Lookup) 一节中，我们将看到更灵活地打印错误信息的方式：抛出异常。

6.9 调试

将一个大程序分解为较小的函数为调试生成了自然的检查点。如果一个函数不如预期的运行，有三个可能性需要考虑：

- 该函数获得的实参有些问题，违反先决条件。
- 该函数有些问题，违反后置条件。
- 返回值或者它的使用方法有问题。

为了排除第一种可能，你可以在函数的开始增加一条 print 语句来打印形参的值（也可以是它们的类型）。或者你可以写代码来显示地检查先决条件。

如果形参看起来没问题，就在每个 return 语句之前增加一条 print 语句，来打印返回值。如果可能，手工检查结果。考虑用一些容易检查的值来调用该函数（类似在 小节中那样）。

如果该函数看起来正常工作，则检查函数调用，确保返回值被正确的使用（或者的确被使用了！）。

在一个函数的开始和结尾处增加打印语句，可以使执行流程更明显。例如，下面是一个带打印语句的阶乘函数：

```
def factorial(n):
    space = '␣' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning␣1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` 是一个空格字符的字符串，用来控制输出的缩进。下面是 `factorial(4)` 的输出结果：

```
                 factorial 4
             factorial 3
         factorial 2
     factorial 1
 factorial 0
 returning 1
     returning 1
         returning 2
             returning 6
                 returning 24
```

如果你对执行流程感到困惑，这种输出可能有助于理解。开发有效的脚手架代码会花些时间，但是一点点的脚手架代码能够节省很多的调试时间。

6.10 术语表

临时变量（temporary variable）： 一个在复杂计算中用于存储过度值的变量。

死代码（dead code）： 程序中永远无法执行的那部分代码，通常是因为其出现在一个返回语句之后。

增量式开发（incremental development）： 一种程序开发计划，目的是通过一次增加及测试少量代码的方式，来避免长时间的调试。

脚手架代码（scaffolding）： 程序开发中使用的代码，但并不是最终版本的一部分。

监护人（guardian）： 一种编程模式，使用条件语句来检查并处理可能引发错误的情形。

6.11 练习

Exercise 6.1. 画出下面程序的堆栈图。这个程序的最终输出是什么？

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Exercise 6.2. *Ackermann* 函数 $A(m,n)$ 的定义如下：

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

查看 http://en.wikipedia.org/wiki/Ackermann_function。编写一个叫作 `ack` 的函数来计算 *Ackermann* 函数。使用你的函数计算 `ack(3, 4)`，其结果应该为 125 。如果 `m` 和 `n` 的值较大时，会发生什么？参考答案

Exercise 6.3. 回文词（*palindrome*）指的是正着拼反着拼都一样的单词，如 *“noon”* 和 *“redivider”*。按照递归定义的话，如果某个词的首字母和尾字母相同，而且中间部分也是一个回文词，那它就是一个回文词。

下面的函数接受一个字符串实参，并返回第一个、最后一个和中间的字母：

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

在 第八章 中我们将介绍它们是如何工作的。

1. 将它们录入到文件 `palindrome.py` 中并测试。当你用一个两个字母的字符串调用 `middle` 时会发生什么？一个字母的呢？空字符串呢？空字符串这样 `''` 表示，中间不含任何字母。

2. 编写一个叫 `is_palindrome` 的函数，接受一个字符串作为实参。如果是回文词，就返回 `True` ，反之则返回 `False` 。记住，你可以使用内建函数 `len` 来检查字符串的长度。

参考答案

Exercise 6.4. 当数字 a 能被 b 整除，并且 ab' 是 b 的幂时，它就是 b 的幂。编写一个叫 `is_power` 的函数，接受两个参数 `a` 和 `b`，并且当 `a` 是 `b` 的幂时返回 `True`。注意：你必须要想好基础情形。

Exercise 6.5. a 和 b 的最大公约数 *(reatest common divisor, GCD)* 是能被二者整除的最大数。

求两个数的最大公约数的一种方法，是基于这样一个原理：如果 r 是 a 被 b 除后的余数，那么 $gcd(a,b) = gcd(b,r)$ 。我们可以把 $gcd(a,0) = a$ 当做基础情形。

编写一个叫 `gcd` 的函数，接受两个参数 `a` 和 `b`，并返回二者的最大公约数。

致谢：这道习题基于 *Abelson* 和 *Sussman* 编写的《*Structure and Interpretation of Computer Programs*》中的例子。

第七章　迭代

本章介绍迭代，即重复运行某个代码块的能力。我们已经在 5.8 节接触了一种利用递归进行迭代的方式；在 4.2 节中，接触了另一种利用 `for` 循环进行迭代的方式。在本章中，我们将讨论另外一种利用 `while` 语句实现迭代的方式。不过，首先我想再多谈谈有关变量赋值的问题。

7.1　重新赋值

可能你已发现对同一变量进行多次赋值是合法的。新的赋值会使得已有的变量指向新的值（同时不再指向旧的值）。

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

第一次打印 `x` 时，它的值为 `5`；第二次打印时，它的值是 `7`。

图 7.1 展示了重新赋值 在状态图中看起来是什么样子。

这里我想探讨一个常见的疑惑点。由于 Python 用等号 (`=`) 来赋值，所以很容易将 `a = b` 这样的语句理解为数学上的相等命题；即 `a` 和 `b` 相等。但是这种理解是错误的。

首先，相等是一种对称关系，赋值不是。例如，在数学上，如果 $a = 7$，则 $7 = a$。但是在 Python 中，语句 `a = 7` 是合法的，`7 = a` 则不合法。

此外，数学中，相等命题不是对的就是错的。如果 $a = b$，那么 a 则是永远与 b 相等。在 Python 中，赋值语句可以使得两个变量相等，但是这两个变量不一定必须保持这个状态：

```
>>> a = 5
>>> b = a    # a and b are now equal
>>> a = 3    # a and b are no longer equal
>>> b
5
```

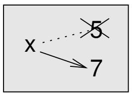

图 7.1: 重新赋值的状态图。

第三行改变了 a 的值，但是没有改变 b 的值，所以它们不再相等了。

给变量重新赋值非常有用，但是需要小心使用。对变量频繁重新赋值会使代码难于阅读，不易调试。

7.2 更新变量

重新赋值的一个常见方式是*更新* (update)，更新操作中变量的新值会取决于旧值。

```
>>> x = x + 1
```

这个语句的意思是，“获得 x 的当前值并与 1 做加法求和，然后将 x 的值更新为所求的和。”

如果试图去更新一个不存在的变量，则会返回一个错误。这是因为 Python 是先求式子右边的值，然后再把所求的值赋给 x：

```
>>> x = x + 1
NameError: name 'x' is not defined
```

在更新变量之前，你得先*初始化* (initialize) 它，通常是通过一个简单的赋值实现：

```
>>> x = 0
>>> x = x + 1
```

通过加 1 来更新变量叫做*递增* (increment)；减 1 叫做*递减* (decrement)。

7.3 while 语句

计算机经常被用来自动处理重复性的任务。计算机很擅长无纰漏地重复相同或者相似的任务，而人类在这方面做的不好。在计算机程序中，重复也被称为*迭代* (iteration)。

我们已经见过两个利用递归来迭代的函数：countdown 和 print_n 。由于迭代的使用非常普遍，所以 Python 提供了使其更容易实现的语言特性。其中之一就是我们在 refrepetition 一节看到的 for 语句。后面我们还会继续介绍。

另外一个用于迭代的语句是 while 。下面是使用 while 语句实现的 countdown：

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

你可以像读英语句子一样来读 while 语句。它的意思是：“只要 n 的值大于 0，则打印出 n 的值，然后让 n 减 1。当 n 递减至 0 时，打印单词 Blastoff！”。

更正式地来说，while 语句的执行流程如下：

1. 首先判断条件为**真** 还是为**假**。
2. 如果为假，退出 while 语句，然后执行接下来的语句；
3. 如果条件为真，则运行 while 语句*循环主体*，运行完再返回第一步；

这种形式的流程叫做*循环* (loop)，因为第三步后又循环回到了第一步。

循环主体应该改变一个或多个变量的值，这样的话才能让条件判断最终变为假，从而终止循环。否则，循环将会永远重复下去，这被称为*无限循环* (infinite loop)。在计算机科学家看来，洗发水的使用说明——“抹洗发水，清洗掉，重复” 便是个无限循环，这总是会让他们觉得好笑。

对于 countdown 来说，我们可以证明循环是一定会终止的：当 n 是 0 或者负数，该循环就不会执行；不然 n 通过每次循环之后慢慢减小，最终也是会变成 0 的。

有些其他循环，可能就没那么好理解了。例如：

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:          # n is even
            n = n / 2
        else:                   # n is odd
            n = n*3 + 1
```

循环的条件是 n != 1，所以循环会一直执行到 n 等于 1，条件判断为假时循环才终止。

每次循环，该程序打印出 n 的值，然后检查它是偶数还是奇数。如果它是偶数，那么 n 可以被 2 整除；如果是奇数，则它的值被替换为 n*3 + 1。例如，如果传递给 sequence 的实参为 3，那么打印出的结果将会是：3、10、5、16、8、4、2、1。

由于 n 的值时增时减，所以不能轻易保证 n 会最终变成 1，或者说这个程序能够终止。对于某些特殊的 n 的值，可以很好地证明它是可以终止的。例如，当 n 的初始值是 2 的倍数时，则每次循环后 n 一直为偶数，直到最终变为 1。上一个示例中，程序就打印了类似的序列，从 16 开始全部为偶数。

难点在于是否能证明程序对于**所有** 的正整数 n 都会终止。目前为止，还没有人证明**或者** 证伪该命题。（见：http://en.wikipedia.org/wiki/Collatz_conjecture 。）

我们做个练习，利用迭代而非递归，重写之前 5.8 节中的 print_n 函数。

7.4 break

有些时候循环执行到一半你才知道循环该结束了。这种情况下，你可以使用 break 语句来跳出循环。

例如，假设你想从用户那里获取输入，直到用户键入 'done'。你可以这么写：

```
while True:
    line = input('>␣')
    if line == 'done':
        break
    print(line)

print('Done!')
```

循环条件是 True，其总是为真，所以该循环会一直执行直到碰到 break。

每次循环时，程序都会给出一个尖括号 (>) 提示。如果用户输入 'done'，执行 break 语句跳出循环。否则，程序就会一直打印出用户所输入的内容并且跳到循环开始，0 以下是一个运行示例：

```
> not done
not done
> done
Done!
```

while 循环的这种写法很常见，因为你可以在循环的任何地方判断条件（而不只是在循环开始），而且你可以积极地表达终止条件（"当出现这个情况是终止"），而不是消极地表示（"继续运行直到出现这个情况"）。

7.5 平方根

循环常用于计算数值的程序中，这类程序一般从一个大概的值开始，然后迭代式地进行改进。

例如，牛顿法 (Newton's method) 是计算平方根的一种方法。假设你想求 a 的平方根。如果你从任意一个估算值 x 开始，则可以利用下面的公式计算出更为较为精确的估算值：

$$y = \frac{x + a/x}{2}$$

例如，假定 a 是 4，x 是 3：

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

可以看到，结果与真实值 ($\sqrt{4} = 2$) 已经很接近了，如果我们用这个值再重新运算一遍，它将得到更为接近的值。

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

再通过多几次的运算，这个估算可以说已经是很精确了。

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

一般来说，我们事先不知道要多少步才能得到正确答案，但是我们知道当估算值不再变动时，我们就获得了正确的答案。

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

当 `y == x` 时，我们可以停止计算了。下面这个循环就是利用一个初始估值 x，循序渐进地计算，直到估值不再变化。

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

对于大部分 a 的值，这个程序运行正常，不过一般来说，检查两个浮点数是否相等比较危险。浮点数只能大约表示：大多数有理数，如 1/3，以及无理数，如: $\sqrt{2}$，是不能用浮点数 (`float`) 来精确表示的。

与其检查 x 和 y 的值是否完全相等，使用内置函数 `abs` 来计算二者之差的绝对值或数量级更为安全：

```
    if abs(y-x) < epsilon:
        break
```

这里，变量 `epsilon` 是一个决定其精确度的值，如 `0.0000001`。

7.6　算法

牛顿法就是一个*算法* (Algorithm) 示例：它是解决一类问题的计算机制（本例中是计算平方根）。

为了理解算法是什么，先了解什么不是算法或许有点帮助。你在学习一位数乘法时，可能背出了乘法表。实际上，你只是记住了 100 个确切的答案。这种知识并**不是**算法性的。

不过，如果你想找 “懒人方法”，你可能就会找到一些诀窍。比如为了计算 n 和 9 的乘积，你可以把 $n - 1$ 作为乘积的第一位数，再把 $10 - n$ 作为第二位数，从而得到它们的乘积。这个诀窍是将任意个位数与 9 相乘的普遍解法。这就**是**一种算法。

类似地，你所学过的进位加法、借位减法、以及长除法都是算法。算法的特点之一就是不需要过多的脑力计算。算法是一个机械的过程，每一步都是依据一组简单的规则跟着上一步来执行的。

执行算法的过程是很乏味的，但是设计算法就比较有趣了，不但是智力上的挑战，更是计算机科学的核心。

一些人们自然而然无需下意识做到的事情，往往是难于用算法表达。理解自然语言就是这样的。我们每个人都听得懂自然语言，但是目前还没有人能够解释我们是**怎么** 做到的，至少无法以算法的形式解释。

7.7　调试

当你开始写更为复杂的程序时，你会发现大部分时间都花费在调试上。更多的代码意味着更高的出错概率，并且会有更多隐藏 bug 的地方。

减少调试时间的一个方法就是 “对分调试”。例如，如果程序有 100 行，你一次检查一行，就需要 100 步。

相反，试着将问题拆为两半。在代码中间部分或者附近的地方，寻找一个可以检查的中间值。加上一行 `print` 语句 (或是其他具有可验证效果的代码)，然后运行程序。

如果中间点检查出错了，那么就说明程序的前半部分存在问题。如果没问题，则说明是后半部分出错了。

每次你都这样检查，就可以将需要搜索的代码行数减少一半。经过 6 步之后（这比 100 小多了），你将会找到那或者两行出错的代码，至少理论上是这样。

在实践中，可能并不能很好的确定程序的 “中间部分” 是什么，也有可能并不是那么好检查。计算行数并且取其中间行是没有意义的。相反，多考虑下程序中哪些地方比较容易出问题，或者哪些地方比较容易进行检查。然后选定一个检查点，在这个断点前后出现 bug 的概念差不多。

7.8　术语表

重新赋值（reassignment）：　给已经存在的变量赋一个新的值。

更新（update）： 变量的新值取决于旧值的一种赋值方法。

初始化（initialize）： 给后面将要更新的变量一个初始值的一种赋值方法。

递增（increment）： 通过增加变量的值的方式更新变量（通常是加 1）。

递减（decrement）： 通过减少变量的值的方式来更新变量。

迭代（iteration）： 利用递归或者循环的方式来重复执行代一组语句的过程。

无限循环（infinite loop）： 无法满足终止条件的循环。

算法（algorithm）： 解决一类问题的通用过程。

7.9 练习

Exercise 7.1. 复制 7.5 小节中的循环，将其封装进一个叫mysqrt 的函数中。这个函数接受a 作为形参，选择一个合适的x 值，并返回a 的平方根估算值。

为测试上面的函数，编写一个名为test_squre_root 的函数，打印出如下表格：

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

其中第一列是 *a* 的值；第二列是通过mysqrt 计算得到的 *a* 的平方根；第三列是用math.sqrt 计算得到的平方根；第四列则是这两个平方根之差的绝对值。

Exercise 7.2. 内置函数eval 接受一个字符串，并使用 *Python* 解释器来计算该字符串。例如：

```
>>> eval('1␣+␣2␣*␣3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

编写一个名为eval_loop 的函数，迭代式地提示用户输入，获取输入的内容，并利用eval 来计算其值，最后打印该值。

该程序应持续运行，知道用户输入'done'，然后返回它最后一次计算的表达式的值。

Exercise 7.3. 数学家斯里尼瓦瑟·拉马努金 (*Srinivasa Ramanujan*) 发现了一个可以用来生成 $1/\pi$ 近似值的无穷级数 (*infinite series*)：

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

编写一个名为`estimate_pi` 的函数，利用上面公式来估算并返回 π 的值。这个函数应该使用 `while` 循环来计算所有项的和，直到最后一项小于`1e-15` (*Python* 中用于表达 10^{-15} 的写法) 时终止循环。你可以将该值与 *math.pi* 进行比较，检测是否准确。

参考答案

第八章　字符串

字符串不像整数、浮点数和布尔型。字符串是一个*序列* (sequence)，这就意味着它是其他值的一个有序的集合。在这章中，你将学习怎么去访问字符串里的字符，同时你也会学习到字符串提供的一些方法。

8.1　字符串是一个序列

字符串是由字符组成的序列。你可以用括号运算符一次访问一个字符：

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第 2 条语句从 `fruit` 中选择索引为 1 的字符并将它赋给 `letter` 。

括号中的表达式被称作*索引* (index)。索引指出在序列中你想要哪个字符（因此而得名）。

但是你可能不会获得你期望的东西：

```
>>> letter
'a'
```

对于大多数人，`'banana'` 的第一个字母是 `b` 而不是 `a`。但是对于计算机科学家，索引是从字符串起点开始的位移量 (offset)，第一个字母的位移量就是 0。

```
>>> letter = fruit[0]
>>> letter
'b'
```

所以 `b` 是 `'banana'` 的第 0 个字母，`a` 是第一个字母，`n` 是第二个字母[1]。

你可以使用一个包含变量名和运算符的表达式作为索引：

```
>>> i = 1
>>> fruit[i]
```

[1]译注：原文分别是 “zero-eth”, “one-eth”, “two-eth”

```
'a'
>>> fruit[i+1]
'n'
```

索引值必须使用整数。否则你会得到报错:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len

len 是一个内建函数，它返回字符串中的字符的数量：

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

为了获得某个字符串中最后一个字符，你可以尝试这样操作：

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

出现 IndexError 的原因在于 'banana' 中没有索引值为 6 的字母。由于我们从 0 开始计数，六个字母的编号是从 0 到 5 。为了获得最后一个字符，你必须将、力 length 减去一：

```
>>> last = fruit[length-1]
>>> last
'a'
```

你也可以使用负数索引，即从字符串的末尾倒着往前数。表达式 fruit[-1] 返回的是最后一个字母，fruit[-2] 返回倒数第二个字母，以此类推。

8.3 使用 for 循环遍历

许多计算中需要一个字符一个字符地处理字符串。通常计算从字符串的头部开始，依次选择每个字符，对其做一些处理，然后继续直到结束。这种处理模式被称作遍历 (traversal) 。编写遍历的方法之一是使用 while 循环：

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

该循环遍历字符串并在每行显示一个字符串。该循环的条件是 `index < len(fruit)`，所以当 `index` 和字符串的长度相等时，条件为假，循环体不被执行。被访问的最后一个字符的索引为 `len(fruit)-1`，这也是字符串的最后一个字符。

我们做个练习，编写一个函数，接受一个字符串作为实参，按照从后向前的顺序显示字符，每行只显示一个。

编写遍历的另一种方法是使用 `for` 循环：

```
for letter in fruit:
    print(letter)
```

每次循环时，字符串中的下一个字符被赋值给变量 `letter` 。循环继续，直到没有剩余的字符串了。

下面的例子演示了如何使用拼接（字符串相加）和 `for` 循环生成一个字母表序列（即按照字母表顺序排列）。在 Robert McCloskey 的书《*Make Way for Ducklings*》中，小鸭子的名字是 Jack、Kack、Lack、Mack、Nack、Ouack、Pack 和 Quack。此循环按顺序输出这些名字：

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

The output is: 输出是：

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

当然，输出并不完全正确，因为 “Ouack” 和 “Quack” 拼写错了。我们做个练习，修改这个程序，解决这个问题。

8.4 字符串切片

字符串的一个片段被称作切片 (slice)。选择一个切片的操作类似于选择一个字符：

```
>>> s = 'Monty␣Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

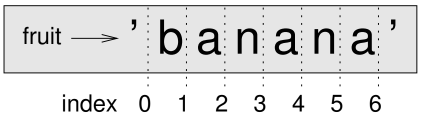

图 8.1: 切片索引。

操作符 `[n:m]` 返回从第 n 个字符到第 m 个字符的字符串片段，包括第一个，但是不包括最后一个。这个行为违反直觉，但是将指向两个字符之间的索引，想象成图 8.1 中那样或许有帮助。

如果你省略第一个索引（冒号前面的值），切片起始于字符串头部。如果你省略第二个索引，切片一直到字符串结尾：

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

如果第一个索引大于或等于第二个，结果是空字符串 (empty string)，用两个引号表示：

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

一个空字符串不包括字符而且长度为 0，但除此之外，它和其它任何字符串一样。

8.5 字符串是不可变的

你会很想在赋值语句的左边使用 `[]`，来改变字符串的一个字符。例如:

```
>>> greeting = 'Hello,␣world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

错误信息中的 "object（对象）" 是那个字符串，"item（元素）" 是你要赋值的字符。目前，我们认为对象和值是同样的东西，但是我们后面将改进此定义（详见 对象与值一节）。

出现此错误的原因是字符串是不可变的 (immutable)，这意味着你不能改变一个已存在的字符串。你最多只能创建一个新的字符串，在原有字符串的基础上略有变化：

```
>>> greeting = 'Hello,␣world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello,␣world!'
```

上面的示例中，我们将一个新的首字母拼接到 greeting 的一个切片上。它不影响原字符串。

8.6 搜索

下面的函数起什么作用?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

在某种意义上，find 和 [] 运算符相反。与接受一个索引并提取相应的字符不同，它接受一个字符并找到该字符所在的索引。如果没有找到该字符，函数返回 -1。

这是我们第一次在循环内部看见 return 语句。如果 word[index] == letter，函数停止循环并马上返回。

如果字符没出现在字符串中，那么程序正常退出循环并返回 -1。

这种计算模式——遍历一个序列并在找到寻找的东西时返回——被称作*搜索* (search)。

我们做个练习，修改 find 函数使得它能接受第三个参数，即从何处开始搜索的索引。

8.7 循环和计数

下面的程序计算字母 a 在字符串中出现的次数：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

此程序演示了另一种被称作*计数器* (counter) 的计算模式。变量 count 初始化为 0，然后每次出现 a 时递增。当循环结束时，count 包含了字母 a 出现的总次数。

我们做一个练习，将这段代码封装在一个名为 count 的函数中，并泛化该函数，使其接受字符串和字母作为实参。

然后重写这个函数，不再使用字符串遍历，而是使用上一节中三参数版本的 find 函数。

8.8 字符串方法

字符串提供了可执行多种有用操作的*方法* (method) 。方法和函数类似，接受实参并返回一个值，但是语法不同。例如，`upper` 方法接受一个字符串，并返回一个都是大写字母的新字符串。

不过使用的不是函数语法 `upper(word)` ， 而是方法的语法 `word.upper()`。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

点标记法的形式指出方法的名字，`upper`，以及应用该方法的字符串的名字，`word`。空括号表明该方法不接受实参。

这被称作*方法调用* (invocation)；此例中，我们可以说是在 `word` 上调用 `upper` 。

事实上，有一个被称为 `find` 的字符串方法，与我们之前写的函数极其相似：

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

此例中，我们在 `word` 上调用 `find` ， 并将我们要找的字母作为参数传入。

事实上，`find` 方法比我们的函数更通用；它还可以查找子字符串，而不仅仅是字符:

```
>>> word.find('na')
2
```

`find` 默认从字符串的首字母开始查找，它还可以接受第二个实参，即从何处开始的索引。

```
>>> word.find('na', 3)
4
```

这是一个*可选参数* (optional argument) 的例子；`find` 也可以接受结束查找的索引作为第三个实参：

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

此次搜索失败，因为 `'b'` 没有出现在索引 1–2 之间（不包括2)。一直搜索到第二个索引，但是并不搜索第二个索引，这使得 `find` 跟切片运算符的行为一致.

8.9 in 运算符

单词 in 是一个布尔运算符，接受两个字符串。如果第一个作为子串出现在第二个中，则返回 True：

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

例如，下面的函数打印所有既出现在 word1 中，也出现在 word2 中的字母：

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

变量名挑选得当的话，Python 代码有时候读起来像是自然语言。你可以这样读此循环，“对于（每个）在（第一个）单词中的字母，如果（该）字母（出现）在（第二个）单词中，打印（该）字母”。

如果你比较 'apples' 和 'oranges'，你会得到下面的结果：

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 字符串比较

关系运算符也适用于字符串。可以这样检查两个字符串是否相等：

```
if word == 'banana':
    print('All␣right,␣bananas.')
```

其它的关系运算符对于按字母序放置单词也很有用：

```
if word < 'banana':
    print('Your␣word,␣' + word + ',␣comes␣before␣banana.')
elif word > 'banana':
    print('Your␣word,␣' + word + ',␣comes␣after␣banana.')
else:
    print('All␣right,␣bananas.')
```

Python 处理大写和小写字母的方式和人不同。所有的大写字母出现在所有小写字母之前，所以：

```
Your word, Pineapple, comes before banana.
```

8.11 调试

当你使用索引遍历序列中的值时，正确地指定遍历的起始和结束点有点困难。下面是一个用来比较两个单词的函数，如果一个单词是另一个的倒序，则返回 `True` ，但其中有两个错误：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

第一条 `if` 语句检查两个单词是否等长。如果不是，我们可以马上返回 `False` 。否则，在函数其余的部分，我们可以假定单词是等长的。这是 6.8节中提到的监护人模式的一个例子。

`i` 和 `j` 是索引：`i` 向前遍历 `word1`，`j` 向后遍历 `word2`。如果我们找到两个不匹配的字母，我们可以立即返回 `False`。如果我们完成整个循环并且所有字母都匹配，我们返回 `True` 。

如果我们用单词 “pots” 和 “stop” 测试该函数，我们期望返回 `True` ，但是却得到一个 `IndexError`：

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

为了调试该类错误，我第一步是在错误出现的行之前，打印索引的值。

```
    while j > 0:
        print(i, j)        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

现在，当我再次运行该程序时，将获得更多的信息：

```
>>> is_reverse('pots', 'stop')
0 4
```

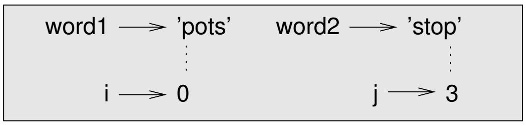

图 8.2: 堆栈图。

```
...
IndexError: string index out of range
```

第一次循环时，j 的值是 4，超出字符串 'post' 的范围了。最后一个字符的索引是 3，所以 j 的初始值应该是 len(word2)−1 。

如果我解决了这个错误，然后运行程序，将获得如下输出:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

这次我们获得了正确的答案，但是看起来循环只运行了三次，这很奇怪。画栈图可以帮我们更好的理解发生了什么。在第一次迭代期间，is_reverse 的栈帧如图 8.2 所示。

我对堆栈图做了些调整，重新排列了栈帧中的变量，增加了虚线来说明 i 和 j 的值表示 word1 和 word2 中的字符。

从这个堆栈图开始，在纸上运行程序，每次迭代时修改 i 和 j 的值。查找并解决这个函数的中第二个错误。

8.12 术语表

对象 (object)： 变量可以引用的东西。现在你将对象和值等价使用。

序列 (sequence)： 一个有序的值的集合，每个值通过一个整数索引标识。

元素 (item)： 序列中的一个值。

索引 (index)： 用来选择序列中元素（如字符串中的字符）的一个整数值。在 Python 中，索引从 0 开始。

切片 (slice)： 以索引范围指定的字符串片段。

空字符串 (empty string)： 一个没有字符的字符串，长度为 0，用两个引号表示。

不可变性 (immutable) 元素不能被改变的序列的性质。

遍历 (traversal)： 对一个序列的所有元素进行迭代，对每一元素执行类似操作。

搜索 (search)： 一种遍历模式，当找到搜索目标时就停止。

计数器 (counter)： 用来计数的变量，通常初始化为 0，并以此递增。

方法调用 (invocation)： 执行一个方法的声明.

可选参数 (optional argument)： 一个函数或者一个方法中不必要指定的参数。

8.13 练习

Exercise 8.1. 点击如下链接，阅读字符串方法的文档。为了确保你理解他们是怎么工作的，可以尝试使用其中的一些方法。strip 和 replace 尤其有用。

文档中使用了可能会引起困惑的句法。例如，在 find(sub[, start[, end]]) 中，方括号意味着这是可选参数。所以，sub 是必填参数，但是 start 是可选的，而且如果你提供了 start，也不一定必须提供 end。
Exercise 8.2. 有一个字符串方法叫 count，它类似于之前 8.7 节中的 counter。阅读这个方法的文档，写一个计算 'banana' 中 a 的个数的方法调用。
Exercise 8.3. 一个字符串切片可以接受指定步长的第三个索引; 也就是连续字符间空格的个数。步长为 2，意味着每隔一个字符；步长为 3，意味着每隔两个字符，以此类推。

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

步长为 -1 就是从单词的尾部开始进行，所以切片 [::-1] 生成一个倒序的字符串。

利用这个习惯用法 (*idiom*)，将习题 6.3中 is_palindrome 函数改写为一行代码版。
Exercise 8.4. 下面这些函数，都是用于 检查一个字符串是否包含一些小写字母的，但是其中至少有一些是错误的函数。检查每个函数，描述这个函数实际上做了什么（假设形参是字符串）。

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercise 8.5. 凯撒密码 (*Caesar cypher*) 是一种弱加密方式，它将每一个字母偏移固定的位置。偏移一个字母，指的是按着字母表偏移，如果需要的话再从尾部跳转至首字母，所以 *‘A’* 偏移三个位置即为 *‘D’*，*‘Z’* 偏移一个位置是 *‘A’*。

要偏移一个单词，可以将其中每一个字母偏移相同的量。例如，*“cheer”* 偏移 *7* 个位置后变成了 *“jolly”*，*“melon”* 偏移 *-10* 个位置变成了 *“cubed”*。在电影《*2001*：太空奥德赛 *(2001: A Space Odyssey)*》中，飞船上的电脑叫做 *HAL*，也就是 *IBM* 偏移 *1* 个位置后的单词。

编写一个叫 `rotate_word` 的函数，接受一个字符串和一个整数作为形参，并返回原字符串按照给定整数量偏移后得到的一个新字符串。

你可能想用内置函数 `ord` ，它可以将字符转化成数值代码，还有 `chr`，它可以将数值代码转化成字符. 字母表的字母以字母表顺序编码，例如：

```
>>> ord('c') - ord('a')
2
```

因为 `'c'` 是字母表中的第二个字母。但是请注意：大写字母的数值代码是不同的。

网上一些可能冒犯人的笑话有时以 *ROT13* 编码，即以 *13* 为偏移量的凯撒密码。如果你不是很容易就被冒犯，那么可以找些这样的笑话，并解码。

参考答案

第九章　Case study: word play | 文字游戏

This chapter presents the second case study, which involves solving word puzzles by searching for words that have certain properties. For example, we'll find the longest palindromes in English and search for words whose letters appear in alphabetical order. And I will present another program development plan: reduction to a previously solved problem.

这一章将介绍第二个案例研究，即通过查找具有特定属性的单词来解答字谜游戏。例如，我们将找出英文中最长的回文单词，以及字符按照字符表顺序出现的单词。另外，我还将介绍另一种程序开发方法：简化为之前已解决的问题。

9.1　Reading word lists | 读取单词列表

For the exercises in this chapter we need a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project (see `http://wikipedia.org/wiki/Moby_Project`). It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is `113809of.fic`; you can download a copy, with the simpler name `words.txt`, from `http://thinkpython2.com/code/words.txt`.

为了完成本章的习题，我们需要一个英语单词的列表。网络上有许多单词列表，但是最符合我们目的列表之一是由 Grady Ward 收集并贡献给公众的列表，这也是 Moby 词典项目的一部分（见：`http://wikipedia.org/wiki/Moby_Project`）。它由 113,809 个填字游戏单词组成，即在填字游戏以及其它文字游戏中被认为有效的单词。在 Moby 集合中，该列表的文件名是 113809of.fic ;你可以从 `http://thinkpython.com/code/words.txt` 下载一个拷贝，文件名已被简化为 words.txt。

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function `open` takes the name of the file as a parameter and returns a **file object** you can use to read the file.

该文件是纯文本，因此你可以用一个文本编辑器打开它，但是你也可以从 Python 中读取它。内建函数 `open` 接受文件名作为形参，并返回一个文件对象 (file object)，你可以使用它读取该文件。

```
>>> fin = open('words.txt')
```

fin is a common name for a file object used for input. The file object provides several methods for reading, including readline, which reads characters from the file until it gets to a newline and returns the result as a string:

`fin 是输入文件对象的一个常用名。该文件对象提供了几个读取方法，包括 readline，其从文件中读取字符直到碰到新行，并将结果作为字符串返回：

```
>>> fin.readline()
'aa\r\n'
```

The first word in this particular list is "aa", which is a kind of lava. The sequence \r\n represents two whitespace characters, a carriage return and a newline, that separate this word from the next.

在此列表中，第一个单词是 "aa"，它是一类熔岩的名称。序列 \r\n 代表两个空白字符，回车和换行，它们将这个单词和下一个分开。

The file object keeps track of where it is in the file, so if you call readline again, you get the next word:

此文件对象跟踪它在文件中的位置，所以如果你再次调用 readline，你获得下一个单词：

```
>>> fin.readline()
'aah\r\n'
```

The next word is "aah", which is a perfectly legitimate word, so stop looking at me like that. Or, if it's the whitespace that's bothering you, we can get rid of it with the string method strip:

下一个单词是 "aah"，不要惊讶，它是一个完全合法的单词。或者，如果空格困扰了你，我们可以用字符串方法 strip 删掉它：

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

You can also use a file object as part of a for loop. This program reads words.txt and prints each word, one per line:

你也可以将文件对象用做 for 循环的一部分。此程序读取 words.txt 并打印每个单词，每行一个：

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 Exercises | 练习

There are solutions to these exercises in the next section. You should at least attempt each one before you read the solutions.

下一节将给出了这些习题的答案。在你看答案之前，应该试着解答一下。

Exercise 9.1. *Write a program that reads* `words.txt` *and prints only the words with more than 20 characters (not counting whitespace).*

编程写一个程序，使得它可以读取 *words.txt* ，然后只打印出那些长度超过 20 个字符的单词（不包括空格）。

Exercise 9.2. *In 1939 Ernest Vincent Wright published a 50,000 word novel called* Gadsby *that does not contain the letter "e". Since "e" is the most common letter in English, that's not easy to do.*

1939 年，*Ernest Vincent Wright* 出版了一本名为《*Gadsby*》的小说，该小说里完全没有使用字符 *"e"*。由于 *"e"* 是英文中最常用的字符，因此写出这本书并不容易。

In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

事实上，不使用这个最常用的字符来构建一个简单的想法都是很难的。开始进展缓慢，但是经过有意识的、长时间的训练，你可以逐渐地熟练。

All right, I'll stop now.

好啦，不说题外话了，我们开始编程练习。

Write a function called `has_no_e` *that returns* `True` *if the given word doesn't have the letter "e" in it.*

写一个叫做 *has_no_e* 的函数，如果给定的单词中不包含字符 *"e"*，返回 *True* 。

Modify your program from the previous section to print only the words that have no "e" and compute the percentage of the words in the list that have no "e".

修改上一节中的程序，只打印不包含 *"e"* 的单词，并且计算列表中不含 *"e"* 单词的比例。

Exercise 9.3. *Write a function named* `avoids` *that takes a word and a string of forbidden letters, and that returns* `True` *if the word doesn't use any of the forbidden letters.*

编写一个名为 *avoids* 的函数，接受一个单词和一个指定禁止使用字符的字符串，如果单词中不包含任意被禁止的字符，则返回 *True* 。

Modify your program to prompt the user to enter a string of forbidden letters and then print the number of words that don't contain any of them. Can you find a combination of 5 forbidden letters that excludes the smallest number of words?

修改你的程序，提示用户输入一个禁止使用的字符，然后打印不包含这些字符的单词的数量。你能找到一个 *5* 个禁止使用字符的组合，使得其排除的单词数目最少么？

Exercise 9.4. *Write a function named* uses_only *that takes a word and a string of letters, and that returns* True *if the word contains only letters in the list. Can you make a sentence using only the letters* acefhlo*? Other than "Hoe alfalfa?"*

编写一个名为 *uses_only* 的函数，接受一个单词和一个字符串。如果该单词只包括此字符串中的字符，则返回 *True*。你能只用 *"acefhlo"* 这几个字符造一个句子么？除了 *"Hoe alfalfa"* 外。

Exercise 9.5. *Write a function named* uses_all *that takes a word and a string of required letters, and that returns* True *if the word uses all the required letters at least once. How many words are there that use all the vowels* aeiou*? How about* aeiouy*?*

编写一个名为 *uses_all* 的函数，接受一个单词和一个必须使用的字符组成的字符串。如果该单词包括此字符串中的全部字符至少一次，则返回 *True*。你能统计出多少单词包含了所有的元音字符 *aeiou*吗？如果换成 *aeiouy* 呢？

Exercise 9.6. *Write a function called* is_abecedarian *that returns* True *if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?*

编写一个名为 *is_abecedarian* 的函数，如果单词中的字符以字符表的顺序出现（允许重复字符），则返回 *True* 。有多少个具备这种特征的单词？

9.3 Search | | 搜索

All of the exercises in the previous section have something in common; they can be solved with the search pattern we saw in Section 8.6. The simplest example is:

前一节的所有习题有一个共同点；都可以用在 8.6一节中看到的搜索模式解决。

```python
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

The for loop traverses the characters in word. If we find the letter "e", we can immediately return False; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn't find an "e", so we return True.

for 循环遍历 word 中的字符。如果我们找到字符 "e"，那么我们可以马上返回 False ；否则我们必须检查下一个字符。如果我们正常退出循环，就意味着我们没有找到一个 "e"，所以我们返回 True 。

You could write this function more concisely using the in operator, but I started with this version because it demonstrates the logic of the search pattern.

你也可以用 in 操作符简化上述函数，但之所以一开始写成这样，是因为它展示了搜索模式的逻辑。

avoids is a more general version of has_no_e but it has the same structure:

avoid 是一个更通用的 has_no_e 函数，但是结构是相同的：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return False as soon as we find a forbidden letter; if we get to the end of the loop, we return True.

一旦我们找到一个禁止使用的字符，我们返回 False ；如果我们到达循环结尾，我们返回 True 。

uses_only is similar except that the sense of the condition is reversed:

除了检测条件相反以外，下面 uses_only 函数与上面的函数很像：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in word that is not in available, we can return False.

这里我们传入一个允许使用字符的列表，而不是禁止使用字符的列表。如果我们在 word 中找到一个不在 available 中的字符，我们就可以返回 False 。

uses_all is similar except that we reverse the role of the word and the string of letters:

除了将 word 与所要求的字符的角色进行了调换之外，下面的 uses_all 函数也是类似的。

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Instead of traversing the letters in word, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return False.

该循环遍历需要的字符，而不是遍历 word 中的字符。如果任何要求的字符没出现在单词中，则我们返回 False 。

If you were really thinking like a computer scientist, you would have recognized that uses_all was an instance of a previously solved problem, and you would have written:

如果你真的像计算机科学家一样思考，你可能已经意识到 uses_all 是前面已经解决的问题的一个实例，你可能会写成：

```
def uses_all(word, required):
    return uses_only(required, word)
```

This is an example of a program development plan called **reduction to a previously solved problem**, which means that you recognize the problem you are working on as an instance of a solved problem and apply an existing solution.

这是一种叫做简化为之前已解决的问题 (reduction to a previously solved problem) 的程序开发方法的一个示例，也就是说，你认识到当前面临的问题是之前已经解决的问题的一个实例，然后应用了已有的解决方案。

9.4 Looping with indices | 使用索引进行循环

I wrote the functions in the previous section with `for` loops because I only needed the characters in the strings; I didn't have to do anything with the indices.

前一节我用 `for` 循环来编写函数，因为我只需要处理字符串中的字符；我不必用索引做任

For `is_abecedarian` we have to compare adjacent letters, which is a little tricky with a `for` loop:

对于下面的 `is_abecedarian`，我们必须比较邻接的字符，用 `for` 循环来写的话有点棘手。

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

An alternative is to use recursion:

一种替代方法是使用递归：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a `while` loop:

另一中方法是使用 `while` 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

The loop starts at i=0 and ends when i=len(word)-1. Each time through the loop, it compares the *i*th character (which you can think of as the current character) to the $i+1$th character (which you can think of as the next).

循环起始于 i=0 ， i=len(word)−1 时结束。每次循环，函数会比较第 *i* 个字符（可以将其认为是当前字符）和第 $i+1$ 个字符（可以将其认为是下一个字符）。

If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.

如果下一个字符比当前的小（字符序靠前），那么我们在递增趋势中找到了断点，即可返回 False 。

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like 'flossy'. The length of the word is 6, so the last time the loop runs is when i is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

如果到循环结束时我们也没有找到一点错误，那么该单词通过测试。为了让你相信循环正确地结束了，我们用 'flossy' 这个单词来举例。它的长度为 6，因此最后一次循环运行时，i 是 4，这是倒数第 2 个字符的索引。最后一次迭代时，函数比较倒数第二个和最后一个字符，这正是我们希望的。

Here is a version of is_palindrome (see Exercise 6.3) that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

下面是 is_palindrome 函数的一种版本（详见练习 6.3 ），其中使用了两个索引；一个从最前面开始并往前上，另一个从最后面开始并往下走。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Or we could reduce to a previously solved problem and write:

或者，我们可以把问题简化为之前已经解决的问题，这样来写:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Using is_reverse from Section 8.11.

使用 8.11 节中描述的 is_reverse

9.5 Debugging | 调试

Testing programs is hard. The functions in this chapter are relatively easy to test because you can check the results by hand. Even so, it is somewhere between difficult and impossible to choose a set of words that test for all possible errors.

程序测试很困难。本章中介绍的函数相对容易测试，因为你可以手工检查结果。即使这样，选择一可以测试所有可能错误的单词集合，是很困难的，介于困难和不可能之间。

Taking `has_no_e` as an example, there are two obvious cases to check: words that have an 'e' should return `False`, and words that don't should return `True`. You should have no trouble coming up with one of each.

以 `has_no_e` 为例，有两个明显的用例需要检查：含有 'e' 的单词应该返回 `False`，不含的单词应该返回 `True` 。你应该可以很容易就能想到这两种情况。

Within each case, there are some less obvious subcases. Among the words that have an "e", you should test words with an "e" at the beginning, the end, and somewhere in the middle. You should test long words, short words, and very short words, like the empty string. The empty string is an example of a **special case**, which is one of the non-obvious cases where errors often lurk.

在每个用例中，还有一些不那么明显的子用例。在含有 "e" 的单词中，你应该测试 "e" 在开始、结尾以及在中间的单词。你还应该测试长单词、短单词以及非常短的单词，如空字符串。空字符串是一个*特殊用例* (special case) ，及一个经常出现错误的不易想到的用例。

In addition to the test cases you generate, you can also test your program with a word list like `words.txt`. By scanning the output, you might be able to catch errors, but be careful: you might catch one kind of error (words that should not be included, but are) and not another (words that should be included, but aren't).

除了你生成的测试用例，你也可以用一个类似 `words.txt` 中的单词列表测试你的程序。通过扫描输出，你可能会捕获错误，但是请注意：你可能捕获一类错误（包括了不应该包括的单词）却没能捕获另一类错误（没有包括应该包括的单词）。

In general, testing can help you find bugs, but it is not easy to generate a good set of test cases, and even if you do, you can't be sure your program is correct. According to a legendary computer scientist:

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> — Edsger W. Dijkstra

一般来讲，测试能帮助你找到错误，但是生成好的测试用例并不容易，并且即便你做到了，你仍然不能保证你的程序是正确的。正如一位传奇计算机科学家所说：

> 程序测试能用于展示错误的存在，但是无法证明不存在错误！
>
> — Edsger W. Dijkstra

9.6 Glossary | 术语表

file object: A value that represents an open file.

文件对象（file object）： 代表打开文件的变量。

reduction to a previously solved problem: A way of solving a problem by expressing it as an instance of a previously solved problem.

简化为之前已经解决的问题： 通过把未知问题简化为已经解决的问题来解决问题的方法。

special case: A test case that is atypical or non-obvious (and less likely to be handled correctly).

特殊用例（special case）： 一种不典型或者不明显的测试用例 (而且很可能无法正确解决的用例)。

9.7 Exercises

Exercise 9.7. *This question is based on a Puzzler that was broadcast on the radio program* Car Talk *(`http://www.cartalk.com/content/puzzlers`):*

这个问题基于广播节目 *《Car Talk》* *(http://www.cartalk.com/content/puzzlers)* 上介绍的一个字谜：

> *Give me a word with three consecutive double letters. I'll give you a couple of words that almost qualify, but don't. For example, the word committee, c-o-m-m-i-t-t-e-e. It would be great except for the 'i' that sneaks in there. Or Mississippi: M-i-s-s-i-s-s-i-p-p-i. If you could take out those i's it would work. But there is a word that has three consecutive pairs of letters and to the best of my knowledge this may be the only word. Of course there are probably 500 more but I can only think of one. What is the word?*
>
> 找出一个包含三个连续双字符的单词。我将给你一系列几乎能够符合条件但实际不符合的单词。比如，*committee* 这个单词，*c-o-m-m-i-t-t-e-e*。如果中间没有 *i* 的话，就太棒了。或者 *Mississippi* 这个单词: *M-i-s-s-i-s-s-i-p-p-i*。假如将这些 *i* 剔除出去，就会符合条件。但是确实存在一个包含三个连续的单词对，而且据我了解，它可能是唯一符合条件的单词。当然也可能有 *500* 多个，但是我只能想到一个。那么这

Write a program to find it. Solution: `http://thinkpython2.com/code/cartalk1.py`.

编写一个程序，找到这个单词。答案：*http://thinkpython2.com/code/cartalk1.py* 。

Exercise 9.8. *Here's another* Car Talk *Puzzler (`http://www.cartalk.com/content/puzzlers`):*

下面是另一个来自 *《Car Talk》* 的谜题（*http://www.cartalk.com/content/puzzlers* ）：

> *"I was driving on the highway the other day and I happened to notice my odometer. Like most odometers, it shows six digits, in whole miles only. So, if my car had 300,000 miles, for example, I'd see 3-0-0-0-0-0.*
>
> *Now, what I saw that day was very interesting. I noticed that the last 4 digits were palindromic; that is, they read the same forward as backward. For example, 5-4-4-5 is a palindrome, so my odometer could have read 3-1-5-4-4-5.*
>
> *One mile later, the last 5 numbers were palindromic. For example, it could have read 3-6-5-4-5-6. One mile after that, the middle 4 out of 6 numbers were palindromic. And you ready for this? One mile later, all 6 were palindromic!*
>
> *The question is, what was on the odometer when I first looked?"*

> "有一天，我正在高速公路上开车，我偶然注意到我的里程表。和大多数里程表一样，它只显示 6 位数字的整数英里数。所以，如果我的车开了 300,000 英里，我能够看到的数字是:3-0-0-0-0-0。
>
> 我当天看到的里程数非常有意思。我注意到后四位数字是回文数；也就是说，正序读和逆序读是一样的。例如，5-4-4-5 就是回文数。所以我的里程数可能是 3-1-5-4-4-5。
>
> 一英里后，后五位数字变成了回文数。例如，里程数可能变成了是 3-6-5-4-5-6。又过了一英里后，6 位数字的中间四位变成了回文数。你相信吗？一英里后，所有的 6 位数字都变成了回文数。
>
> 那么问题来了，当我第一次看到里程表时，里程数是多少？"

Write a Python program that tests all the six-digit numbers and prints any numbers that satisfy these requirements. Solution: `http://thinkpython2.com/code/cartalk2.py`.

编写写一个程序，测试所有的 6 位数字，然后输出所有符合要求的结果。答案：*http://thinkpython2.com/code/cartalk2.py* 。

Exercise 9.9. *Here's another* Car Talk *Puzzler you can solve with a search (*`http://www.cartalk.com/content/puzzlers`*):*

还是 *《Car Talk》* 的谜题（*http://www.cartalk.com/content/puzzlers* ），你可以通过利用搜索模式解答：

> *"Recently I had a visit with my mom and we realized that the two digits that make up my age when reversed resulted in her age. For example, if she's 73, I'm 37. We wondered how often this has happened over the years but we got sidetracked with other topics and we never came up with an answer.*
>
> *When I got home I figured out that the digits of our ages have been reversible six times so far. I also figured out that if we're lucky it would happen again in a few years, and if we're really lucky it would happen one more time after that. In other words, it would have happened 8 times over all. So the question is, how old am I now?"*

> "最近我探望了我的妈妈，我们忽然意识到把我的年纪数字反过来就是她的年龄。比如，如果她 73 岁，那么我就是 37 岁。我们想知道过去这些年来，

> 发生了多少次这样的巧合，但是我们很快偏离到其他话题上，最后并没有找到答案。
>
> 回到家后，我计算出我的年龄数字有 6 次反过来就是妈妈的年龄。同时，我也发现如果幸运的话，将来几年还可能发生这样的巧合，运气再好点的话，之后还会出现一次这样的巧合。换句话说，这样的巧合一共会发生 8 次。那么，问题来了，我现在多大了？”

Write a Python program that searches for solutions to this Puzzler. Hint: you might find the string method `zfill` *useful.*

Solution: *http://thinkpython2.com/code/cartalk3.py*.

编写一个查找谜题答案的 *Python* 函数。提示：字符串的 *“zfill“* 方法特别有用。答案：*http://thinkpython2.com/code/cartalk3.py* 。

第十章　列表

本章介绍 Python 中最有用的内置类型之一：*列表* (list)。你还将进一步学习关于对象的知识以及同一个对象拥有多个名称时会发生什么。

10.1　列表是一个序列

与字符串类似，*列表* 是由多个值组成的序列。在字符串中，每个值都是字符；在列表中，值可以是任何数据类型。列表中的值称为*元素* (element)，有时也被称为*项* (item)。

创建新列表的方法有多种；最简单的方法是用方括号 (`[` 和 `]`) 将元素包括起来:

```
[10, 20, 30, 40]
['crunchy␣frog', 'ram␣bladder', 'lark␣vomit']
```

第一个例子是包含 4 个整数的列表。第二个是包含 3 个字符串的列表。一个列表中的元素不需要是相同的数据类型。下面的列表包含一个字符串、一个浮点数、一个整数和另一个列表：

```
['spam', 2.0, 5, [10, 20]]
```

一个列表在另一个列表中，称为*嵌套* (nested) 列表。

一个不包含元素的列表被称为空列表；你可以用空的方括号 `[]` 创建一个空列表。

正如你想的那样，你可以将列表的值赋给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10.2　列表是可变的

访问列表中元素的语法，与访问字符串中字符的语法相同，都是通过方括号运算符实现的。括号中的表达式指定了元素的索引。记住，索引从 0 开始：

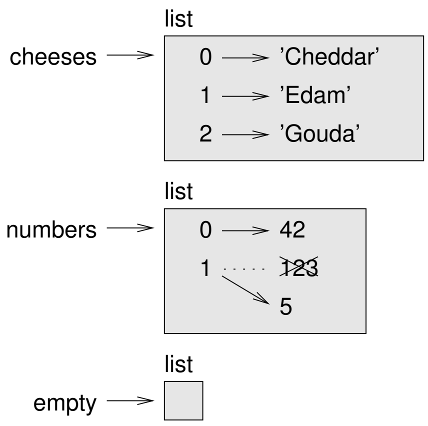

图 10.1: 状态图。

```
>>> cheeses[0]
'Cheddar'
```

和字符串不同的是，列表是可变的。当括号运算符出现在赋值语句的左边时，它就指向了列表中将被赋值的元素。

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

`numbers` 中索引为 1 的元素，原来是 123，现在变成了 5。

图 10.1 展示了 `cheeses` 、`nubmers` 和 `empty` 的状态图。

列表用外部标有 “list” 的盒子表示，盒子内部是列表的元素。`cheeses` 指向一个有 3 个元素的列表，3 个元素的下标分别是 0、1、2。`numbers` 包含两个元素；状态图显示第二个元素原来是 123，被重新赋值为 5。`empty` 对应一个没有元素的列表。

列表下标的工作原理和字符串下标相同：

- 任何整数表达式都可以用作下标。
- 如果你试图读或写一个不存在的元素，你将会得到一个索引错误 (IndexError)。
- 如果下标是负数，它将从列表的末端开始访问列表。

`in` 运算符在列表中同样可以使用。

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 列表遍历

最常用的遍历列表的方式是使用 for 循环。语法和字符串遍历类似：

```
for cheese in cheeses:
    print(cheese)
```

如果你只需要读取列表中的元素，这种方法已经足够。然而，如果你想要写入或者更新列表中的元素，你需要通过下标访问。一种常用的方法是结合内置函数 `range` 和 `len`：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

这个循环将遍历列表并更新每个元素。`len` 返回列表中的元素个数。`range` 返回一个包含从 0 到 $n-1$ 下标的列表，其中 n 是列表的长度。每次循环中，i 得到下一个元素的下标。循环主体中的赋值语句使用 i 读取该元素的旧值，并赋予其一个新值。

对一个空列表执行 `for` 循环时，将不会执行循环的主体：

```
for x in []:
    print('This␣never␣happens.')
```

尽管一个列表可以包含另一个列表，嵌套的列表本身还是被看作一个单个元素。下面这个列表的长度是 4：

```
['spam', 1, ['Brie', 'Roquefort', 'Pol␣le␣Veq'], [1, 2, 3]]
```

10.4 列表操作

加号运算符+ 拼接多个列表：

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

乘号运算符 * 以给定次数的重复一个列表:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

第一个例子重复 4 次。第二个例子重复了那个列表 3 次。

10.5 列表切片

切片 (slice) 运算符同样适用于列表：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

如果你省略第一个索引，切片将从列表头开始。如果你省略第二个索引，切片将会到列表尾结束。所以如果你两者都省略，切片就是整个列表的一个拷贝。

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

由于列表是可变的，通常在修改列表之前，对列表进行拷贝是很有用的。

切片运算符放在赋值语句的左边时，可以一次更新多个元素：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 列表方法

Python 为列表提供了一些方法. 例如, `append` 添加一个新元素到列表的末端:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` 将接受一个列表作为参数，并将其其中的所有元素添加至目标列表中：

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

这个例子中 `t2` 没有改动。

`sort` 将列表中的元素从小到大进行排序：

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

大部分的列表方法都是无返回值的；它们对列表进行修改，然后返回 None。如果你意外的写了 `t.sort()`，你将会对结果感到失望的。

10.7 映射、筛选和归并

你可以这样使用循环，对列表中所有元素求和:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` 被初始化为 0。每次循环时，`x` 从列表中获取一个元素。运算符 `+=` 提供了一个快捷的更新变量的方法。这个*增量赋值语句* (augmented assignment statement) 。

```
    total += x
```

等价于

```
    total = total + x
```

当循环执行时，`total` 将累计元素的和；一个这样的变量有时被称为*累加器* (accumulator) 。

把一个列表中的元素加起来是一个很常用的操作，所以 Python 将其设置为一个内建内置函数 `sum`：

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

一个像这样的将一系列的元素合并成一个单一值的操作有时称为*归并* (reduce) 。

有时，你在构建一个列表时还需要遍历另一个列表。例如，下面的函数接受一个字符串列表

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res 被初始化为一个空列表；每次循环时，我们添加下一个元素。所以 res 是另一种形式的累加器。

类似 capitalize_all 这样的操作有时被称为映射 (map) ，因为它 “映射” 一个函数（在本例中是方法 capitalize ）到序列中的每个元素上。

另一个常见的操作是从列表中选择一些元素，并返回一个子列表。例如，下面的函数读取一个字符串列表，并返回一个仅包含大写字符串的列表：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper 是一个字符串方法，如果字符串仅含有大写字母，则返回 True。

类似 only_upper 这样的操作被称为筛选 (filter) ，因为它选中某些元素，然后剔除剩余的元素。

大部分常用列表操作可以用映射、筛选和归并这个组合表示。

10.8 删除元素

有多种方法可以从列表中删除一个元素。如果你知道元素的下标，你可以使用 pop ：

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

pop 修改列表，并返回被移除的元素。如果你不提供下标，它将移除并返回最后一个元素。

如果你不需要被移除的元素，可以使用 del 运算符：

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

如果你知道要删除的值（但是不知道其下标），你可以使用 remove ：

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

remove 的返回值是 None。

要移除多个元素，你可以结合切片索引使用 del：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

同样的，切片选择到第二个下标（不包含第二个下标）处的所有元素。

10.9 列表和字符串

一个字符串是多个字符组成的序列，一个列表是多个值组成的序列。但是一个由字符组成的列表不同于字符串。可以使用 list 将一个字符串转换为字符的列表:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

由于 list 是内置函数的名称，你应避免将它用作变量名。我同样避免使用 l ，因为它看起来很像 1。这就是为什么我用了 t 。

list 函数将字符串分割成单独的字符。如果你想将一个字符串分割成一些单词，你可以使用 split 方法：

```
>>> s = 'pining␣for␣the␣fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

可以提高一个叫做分隔符 (delimiter) 的可选参数，指定什么字符作为单词之间的分界线。下面的例子使用连字符作为分隔符：

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

join 的功能和 split 相反。它将一个字符串列表的元素拼接起来。join 是一个字符串方法，所以你需要在一个分隔符上调用它，并传入一个列表作为参数：

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = '␣'
>>> s = delimiter.join(t)
>>> s
'pining␣for␣the␣fjords'
```

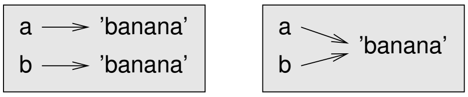

图 10.2: State diagram.

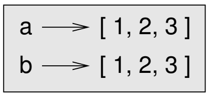

图 10.3: State diagram.

在这个例子中，分隔符是一个空格，所以 join 在单词之间添加一个空格。如果不使用空格拼接字符串，你可以使用空字符串 '' 作为分隔符。

10.10 对象和值

如果我们执行下面的赋值语句：

```
a = 'banana'
b = 'banana'
```

我们知道 a 和 b 都指向一个字符串，但是我们不知道是否他们指向**同一个** 字符串。这里有两种可能的状态，如图 10.2 所示。

一种情况是，a 和 b 指向两个有相同值的不同对象。第二种情况是，它们指向同一个对象。

为了查看两个变量是否指向同一个对象，你可以使用 is 运算符。

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

在这个例子中，Python 仅生成了一个字符串对象，a 和 b 都指向它。但是当你创建两个列表时，你得到的是两个对象：

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

所以状态图如图 10.3所示。

在这个例子中，我们称这两个列表是相等的 (equivalent)，因为它们有相同的元素。但它们并不相同 (identical)，因为他们不是同一个对象。如果两个对象相同，它们也是相等的，但是如果它们是相等的，它们不一定是相同的。

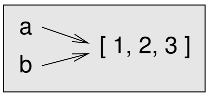

图 10.4: State diagram.

至此，我们一直在等价地使用 “对象” 和 “值”，但是更准确的说，一个对象拥有一个值。如果你对 `[1, 2, 3]` 求值，会得到一个值为整数序列的列表对象。如果另一个列表有同样的元素，我们说它们有相同的值，但是它们并不是同一个对象。

10.11 别名

如果 `a` 指向一个对象，然后你赋值 `b = a`，那么两个变量指向同一个对象：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

状态图如图 10.4 所示。

变量和对象之间的关联称为引用 (reference) 。在这个例子中，有两个对同一个对象的引用。

如果一个对象有多于一个引用，那它也会有多个名称，我们称这个对象是有别名的 (aliased) 。

如果一个有别名的对象是可变的，对其中一个别名 (alias) 的改变对影响到其它的别名：

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

尽管这个行为很有用，但是容易导致出现错误。通常，避免对于可变对象使用别名相对更安全。

对于像字符串这样的不可变对象，使用别名没有什么问题。例如：

```
a = 'banana'
b = 'banana'
```

`a` 和 `b` 是否指向同一个字符串基本上没有什么影响。

10.12 列表参数

当你将一个列表作为参数传给一个函数，函数将得到这个列表的一个引用。如果函数对这个列表进行了修改，会在调用者中有所体现。例如，“delete_head” 删除列表的第一个元素：

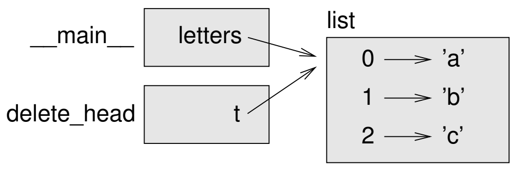

图 10.5: Stack diagram.

```
def delete_head(t):
    del t[0]
```

这样使用这个函数：

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

参数 t 和变量 letters 是同一个对象的别名。其堆栈图如图 10.5所示。

由于列表被两个帧共享，我把它画在它们中间。

需要注意的是修改列表操作和创建列表操作间的区别。例如，append 方法是修改一个列表，而 + 运算符是创建一个新的列表：

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append 修改列表并返回 None。

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

运算符 + 创建了一个新列表，而不改变原始的列表。

如果你要编写一个修改列表的函数，这一点就很重要。例如，这个函数不会 删除列表的第一个元素：

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

切片运算符创建了一个新列表，然后这个表达式让 t 指向了它，但是并不会影响原来被调用的列表。

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

在 bad_delete_head 的开始处，t 和 t4 指向同一个列表。在结束时，t 指向一个新列表，但是 t4 仍然指向原来的、没有被改动的列表。

一个替代的写法是，写一个创建并返回一个新列表的函数。例如，tail 返回列表中除了第一个之外的所有元素：

```
def tail(t):
    return t[1:]
```

这个函数不会修改原来的列表。下面是函数的使用方法：

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 调试

粗心地使用列表（以及其他可变对象）会导致长时间的调试。下面列举一些常见的陷阱以及避免它们的方法：

1. 大多数的列表方法会对参数进行修改，然后返回 None 。这和字符串方法相反，后者保留原始的字符串并返回一个新的字符串。

   如果你习惯这样写字符串代码：

   ```
   word = word.strip()
   ```

   那么你很可能会写出下面的列表代码：

   ```
   t = t.sort()           # WRONG!
   ```

   因为 sort 返回 None ，所以你的下一个对 t 执行的操作很可能会失败。

   在使用 list 方法和操作符之前，你应该仔细阅读文档，然后在交互模式下测试。

2. 选择一种写法，坚持下去。

   列表的一个问题就是有太多方法可以做同样的事情。例如，要删除列表中的一个元素，你可以使用 pop 、remove 、del 甚至是切片赋值。

要添加一个元素，你可以使用 append 方法或者 + 运算符。假设 t 是一个列表，x 是一个列表元素，以下这些写法都是正确的：

```
t.append(x)
t = t + [x]
t += [x]
```

而这些是错误的：

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

在交互模式下尝试每一个例子，保证你明白它们做了什么。注意只有最后一个会导致运行时错误；其他的都是合乎规范的的，但结果却是错的。

3. 通过创建拷贝来避免别名.

   如果你要使用类似 sort 这样的方法来修改参数，但同时有要保留原列表，你可以创建一个拷贝。

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

   在这个例子中，你还可以使用内置函数 sorted，它将返回一个新的已排序的列表，原列表将保持不变。

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14 术语表

列表（list）： 多个值组成的序列。

元素（element）： 列表（或序列）中的一个值，也称为项。

嵌套列表（nested list）： 作为另一个列表的元素的列表。

累加器（accumulator）： 循环中用于相加或累积出一个结果的变量。

增量赋值语句（augmented assignment）： 一个使用类似 += 操作符来更新一个变量的值的语句。

归并（reduce）： 遍历序列，将所有元素求和为一个值的处理模式。

映射（map）： 遍历序列，对每个元素执行操作的处理模式。

筛选（filter）： 遍历序列，选出满足一定标准的元素的处理模式。

对象（object） 变量可以指向的东西。一个对象有数据类型和值。

相等（equivalent）： 有相同的值。

相同（identical）： 是同一个对象（隐含着相等）。

引用（reference）： 一个变量和它的值之间的关联。

别名使用: 两个或者两个以上变量指向同一个对象的情况。

分隔符（delimiter）： 一个用于指示字符串分割位置的字符或者字符串。

10.15 练习

你可以从 此处 下载这些练习的答案。

Exercise 10.1. 编写一个叫做 *nested_sum* 的函数，接受一个由一些整数列表构成的列表作为参数，并将所有嵌套列表中的元素相加。例如：

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercise 10.2. 编写一个叫做 *cumsum* 的函数，接受一个由数值组成的列表，并返回累加和；即一个新列表，其中第 $i+1$ 个元素是原列表中前 i 个元素的和。例如：

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Exercise 10.3. 编写一个叫做 *middle* 的函数，接受一个列表作为参数，并返回一个除了第一个和最后一个元素的列表。例如：

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercise 10.4. 编写一个叫做 *chop* 的函数，接受一个列表作为参数，移除第一个和最后一个元素，并返回 *None*。例如：

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 10.5. 编写一个叫做 *is_sorted* 的函数，接受一个列表作为参数，如果列表是递增排列的则返回 *True* ，否则返回 *False*。例如：

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 10.6. 如果可以通过重排一个单词中字母的顺序，得到另外一个单词，那么称这两个单词是变位词。编写一个叫做 *is_anagram* 的函数，接受两个字符串作为参数，如果它们是变位词则返回 *True* 。

Exercise 10.7. 编写一个叫做 *has_duplicates* 的函数，接受一个列表作为参数，如果一个元素在列表中出现了不止一次，则返回 *True* 。这个函数不能改变原列表。

Exercise 10.8. 这个习题与所谓的生日悖论有关。你可以在 此处了解更多相关的内容。

如果你的班级上有 23 个学生，2 个学生生日相同的概率是多少？你可以通过随机产生 23 个生日，并检查匹配来估算概率。提示：你可以使用 *random* 模块中的 *randint* 函数来生成随机生日。

你可以参考 我的答案。

Exercise 10.9. 编写一个函数，读取文件 *words.txt*，建立一个列表，其中每个单词为一个元素。编写两个版本，一个使用 *append* 方法，另一个使用 *t = t + [x]*。那个版本运行得慢？为什么？

参考答案

Exercise 10.10. 使用 *in* 运算符可以检查一个单词是否在单词表中，但这很慢，因为它是按顺序查找单词。

由于单词是按照字母顺序排序的，我们可以使用两分法（也称二进制搜索）来加快速度，类似你在字典中查找单词的方法。你从中间开始，如果你要找的单词在中间的单词之前，你查找前半部分，否则你查找后半部分。

不管怎样，你都会将搜索范围减小一半。如果单词表有 113,809 个单词，你只需要 17 步就可以找到这个单词，或着得出单词不存在的结论。

编写一个叫做 *in_bisect* 的函数，接受一个已排序的列表和一个目标值作为参数，返回该值在列表中的位置，如果不存在则返回 *None* 。

或者你可以阅读 *bisect* 模块的文档并使用它！

参考答案

Exercise 10.11. 两个单词中如果一个是另一个的反转，则二者被称为是“反转词对”。编写一个函数，找出单词表中所有的反转词对。

参考答案

Exercise 10.12. 如果交替的从两个单词中取出字符将组成一个新的单词，这两个单词被称为是“连锁词”。例如，“shoe” 和 “cold” 连锁后成为 “schooled”。

1. 编写一个程序，找出单词表中所有的连锁词。提示：不要枚举所有的单词对。
2. 你能够找到三重连锁的单词吗？即每个字母依次从 3 个单词得到。

第十一章　字典

本章介绍另一个内建数据类型：字典 (dictionary)。字典是 Python 中最优秀的特性之一；许多高效、优雅的算法以此为基础。

11.1　字典即映射

字典 与列表类似，但是更加通用。在列表中，索引必须是整数；但在字典中，它们可以是（几乎）任何类型。

字典包含了一个索引的集合，被称为**键** (keys)，和一个值 (values) 的集合。一个键对应一个值。这种一一对应的关联被称为**键值对** (key-value pair)，有时也被称为**项** (item)。

在数学语言中，字典表示的是从键到值的映射，所以你也可以说每一个键 "映射到" 一个值。举个例子，我们接下来创建一个字典，将英语单词映射至西班牙语单词，因此键和值都是字符串。

`dict` 函数生成一个不含任何项的新字典。由于 `dict` 是内建函数名，你应该避免使用它来命名变量。

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

花括号 {} 表示一个空字典。你可以使用方括号向字典中增加项：

```
>>> eng2sp['one'] = 'uno'
```

这行代码创建一个新项，将键 `'one'` 映射至值 `'uno'`。如果我们再次打印该字典，会看到一个以冒号分隔的键值对：

```
>>> eng2sp
{'one': 'uno'}
```

输出的格式同样也是输入的格式。例如，你可以像这样创建一个包含三个项的字典：

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

但是，如果你打印 `eng2sp` ，结果可能会让你感到意外：

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

键-值对的顺序和原来不同。同样的例子在你的电脑上可能有不同的结果。通常来说，字典中项的顺序是不可预知的。

但这没有关系，因为字典的元素不使用整数索引来索引，而是用键来查找对应的值：

```
>>> eng2sp['two']
'dos'
```

键 `'two'` 总是映射到值 `'dos'` ，因此项的顺序没有关系。

如果键不存在字典中，会抛出一个异常：

```
>>> eng2sp['four']
KeyError: 'four'
```

`len` 函数也适用于字典；它返回键值对的个数：

```
>>> len(eng2sp)
3
```

`in` 操作符也适用于字典；它可以用来检验字典中是否存在某个*键* （仅仅有这个值还不够）。

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

想要知道字典中是否存在某个值，你可以使用 `values` 方法，它返回值的集合，然后你可以使用 `in` 操作符来验证：

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

`in` 操作符对列表和字典采用不同的算法。对于列表，它按顺序依次查找目标，如 8.6 节所示。随着列表的增长，搜索时间成正比增长。

对于字典，Python 使用一种叫做*哈希表* (hashtable) 的算法，这种算法具备一种了不起的特性：无论字典中有多少项，`in` 运算符搜索所需的时间都是一样的。我将在第二十一章的哈希表一节中具体解释背后的原理，但是如果你不再多学习几章内容，现在去看解释的话可能很难理解。

11.2 字典作为计数器集合

假设给你一个字符串，你想计算每个字母出现的次数。有多种方法可以使用：

1. 你可以生成 26 个变量，每个对应一个字母表中的字母。然后你可以遍历字符串，对于每个字符，递增相应的计数器，你可能会用到链式条件。
2. 你可以生成具有 26 个元素的列表。然后你可以将每个字符转化为一个数字（使用内建函数 ord ），使用这些数字作为列表的索引，并递增适当的计数器。
3. 你可以生成一个字典，将字符作为键，计数器作为相应的值。字母第一次出现时，你应该向字典中增加一项。这之后，你应该递增一个已有项的值。

每个方法都是为了做同一件事，但是各自的实现方法不同。

实现 是指执行某种计算的方法；有的实现更好。例如，使用字典的实现有一个优势，即我们不需要事先知道字符串中有几种字母，只要在出现新字母时分配空间就

代码可能是这样的：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

函数名叫 histogram (直方图)，是计数器（或是频率）集合的统计术语。

函数的第一行生成一个空字典。for 循环遍历该字符串。每次循环，如果字符 c 不在字典中，我们用键 c 和初始值 1 生成一个新项（因为该字母出现了一次）。如果 c 已经在字典中了，那么我们递增 d[c] 。

下面是运行结果：

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

histogram 函数表明字母 'a' 和 'b' 出现了一次，'o' 出现了两次，等等。

字典类有一个 get 方法，接受一个键和一个默认值作为参数。如果字典中存在该键，则返回对应值；否则返回传入的默认值。例如：

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

我们做个练习，试着用 get 简化 histogram 函数。你应该能够不再使用 if 语句。

11.3　循环和字典

在 for 循环中使用字典会遍历其所有的键。例如，下面的 print_hist 会打印所有键与对应的值：

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

输出类似：

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

重申一遍，字典中的键是无序的。如果要以确定的顺序遍历字典，你可以使用内建方法 sorted：

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4　逆向查找

给定一个字典 d 以及一个键 t ，很容易找到相应的值 v = d[k] 。该运算被称作查找 (lookup) 。

但是如果你想通过 v 找到 k 呢？有两个问题：第一，可能有不止一个的键其映射到值 v。你可能可以找到唯一一个，不然就得用 list 把所有的键包起来。第二，没有简单的语法可以完成逆向查找 (reverse lookup) ；你必须搜索。

下面这个函数接受一个值并返回映射到该值的第一个键：

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

该函数是搜索模式的另一个例子，但是它使用了一个我们之前没有见过的特性，raise。raise 语句能触发异常，这里它触发了 ValueError，这是一个表示查找操作失败的内建异常。

如果我们到达循环结尾，这意味着字典中不存在 v 这个值，所以我们触发一个异常。

下面是一个成功逆向查找的例子：

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

以及一个失败的例子：

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

你触发的异常和 Python 触发的产生效果一样：都打印一条回溯和错误信息。

raise 语句接受一个详细的错误信息作为可选的实参。例如：

```
>>> raise LookupError('value␣does␣not␣appear␣in␣the␣dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

逆向查找比正向查找慢得多；如果你频繁执行这个操作或是字典很大，程序性能会变差。

11.5 字典和列表

在字典中，列表可以作为值出现。例如，如果你有一个从字母映射到频率的字典，而你想倒转它；也就是生成一个从频率映射到字母的字典。因为可能有些字母具有相同的频率，所以在倒转字典中的每个值应该是一个字母组成的列表。

下面是一个倒转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

每次循环，key 从 d 获得一个键和相应的值 val 。如果 val 不在 inverse 中，意味着我们之前没有见过它，因此我们生成一个新项并用一个单元素集合 (singleton) （只包含一

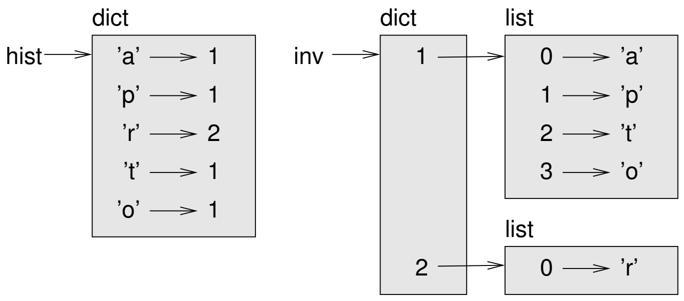

图 11.1: State diagram.

个元素的列表）初始化它。否则就意味着之前已经见过该值，因此将其对应的键添加至列表。

举个例子：

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

图 11.1 是关于 hist 与 inverse 的状态图。字典用标有类型 dict 的方框表示，方框中是键值对。如果值是整数、浮点数或字符串，我就把它们画在方框内部，但我通常把列表画在方框外面，目的只是为了不让图表变复杂。

如本例所示，列表可以作为字典中的值，但是不能是键。下面演示了这样做的结果：

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

我之前提过，字典使用哈希表实现，这意味着键必须是**可哈希的** (hashable) 。

哈希 (hash) 函数接受一个值 (任何类型) 并返回一个整数。字典使用被称作哈希值的这些整数，来存储和查找键值对。

如果键是不可变的，那么这种实现可以很好地工作。但是如果键是可变的，如列表，那么就会发生糟糕的事情。例如，当你生成一个键值对时，Python 哈希该键并将其存储在相应的位置。如果你改变键然后再次哈希它，它将被存储到另一个位置。在那种情况下，对于相同的键，你可能有两个值，或者你可能无法找到一个键。无论如何，字典都不会正确的工作。

这就是为什么键必须是可哈希的，以及为什么如列表这种可变类型不能作为键。绕过这种限制最简单的方法是使用元组，我们将在下一章中介绍。

因为字典是可变的，因此它们不能作为键，但是可 以 用作值。

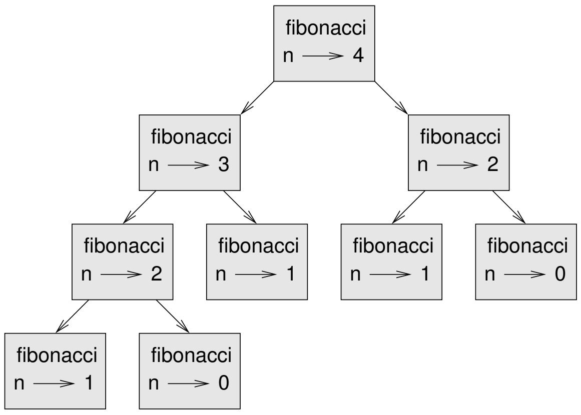

图 11.2: Call graph.

11.6 备忘

如果你在 6.7节中接触过 fibonacci 函数，你可能注意到输入的实参越大，函数运行就需要越多时间。而且运行时间增长得非常快。

要理解其原因，思考 ă 图 11.2 ，它展示了当 n=4 时 fibonacci 的调用图 (call graph)：

调用图中列出了一系列函数栈帧，每个栈帧之间通过线条与调用它的函数栈帧相连。在图的顶端，n = 4 的 fibonacci 调用 n = 3 和 n = 2 的 fibonacci 。接着，n = 3 的 fibonacci 调用 n = 2 和 n = 1 的 fibonacci。以此类推。

数数 fibonacci(0) 和 fibonacci(1) 总共被调用了几次。对该问题，这不是一个高效的解，并且随着实参的变大会变得更糟。

一个解决办法是保存已经计算过的值，将它们存在一个字典中。存储之前计算过的值以便今后使用，它被称作备忘录 (memo) 。下面是使用备忘录 (memoized) 的 fibonacci 的实现：

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

known 是一个字典，记录了我们已经计算过的斐波纳契数字。它一开始包含两个项：0 映射到 0，1 映射到 1。

当 fibonacci 被调用时，它先检查 known 。如果结果存在，则立即返回。否则，它必须计算新的值，将其加入字典，并返回它。

将两个版本的 fibonacci 函数比比看，你就知道后者快了很多。

11.7 全局变量

在前面的例子中，known 是在函数的外部创建的，因此它属于被称作 __main__ 的特殊帧。因为 __main__ 中的变量可以被任何函数访问，它们也被称作*全局变量* (global) 。与函数结束时就会消失的局部变量不同，不同函数调用时全局变量一直都存在。

全局变量普遍用作标记 (flag)；就是说明（标记）一个条件是否为真的布尔变量。例如，一些程序使用一个被称作 verbose 的标记来控制输出的丰富程度：

```
verbose = True

def example1():
    if verbose:
        print('Running␣example1')
```

如果你试图对一个全局变量重新赋值，结果可能出乎意料。下面的例子本应该记录函数是否已经被调用过了

```
been_called = False

def example2():
    been_called = True          # WRONG
```

但是如果你运行它，你会发现 been_called 的值并未发生改变。问题在于 example2 生成了一个新的被称作 been_called 的局部变量。当函数结束的时候，该局部变量也消失了，并且对全局变量没有影响。

要在函数内对全局变量重新赋值，你必须在使用之前*声明* (declare) 该全局变量：

```
been_called = False

def example2():
    global been_called
    been_called = True
```

global 语句告诉编译器，“在这个函数里，当我说 been_called 时，我指的是那个全局变量，别生成局部变量”。

下面是一个试图更新全局变量的例子：

```
count = 0

def example3():
    count = count + 1           # WRONG
```

一旦运行，你会发现：

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python 默认 count 是局部变量，在这个假设下，你这是在未写入任何东西前就试图读取。解决方法还是声明 count 是全局变量。

```
def example3():
    global count
    count += 1
```

如果全局变量是可变的，你可以不加声明地修改它：

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

因此你可以增加、删除和替代全局列表或者字典的元素，但是如果你想对变量重新赋值，你必须声明它：

```
def example5():
    global known
    known = dict()
```

全局变量有时是很有用的，但如果你的程序中有很多全局变量，而且修改频繁，这样会增加程序调试的难度。

11.8 调试

当你操作较大的数据集时，通过打印并手工检查数据来调试很不方便。下面是针对调试大数据集的一些建议：

缩小输入 (Scale down the input)： 如果可能，减小数据集合的大小。例如，如果程序读入一个文本文件，从前 10 行开始分析，或是找到更小的样例。你可以选择编辑读入的文件，或是（最好）修改程序使它只读入前 n 行。

如果出错了，你可以将 n 缩小为会导致该错误的最小值，然后在查找和解决错误的同时，逐步增加 n 的值。

检查摘要和类型 (Check summaries and types)： Instea 考虑打印数据的摘要，而不是打印并检查全部数据集合：例如，字典中项的数目或者数字列表的总和。

运行时错误的一个常见原因，是值的类型不正确。为了调试此类错误，打印值的类型通常就足够了。

编写自检代码 (Write self-checks)： 有时你可以写代码来自动检查错误。例如，如果你正在计算数字列表的平均数，你可以检查其结果是不是大于列表中最大的元素，或者小于最小的元素。这被称作 “合理性检查”，因为它能检测出 “不合理的” 结果。

另一类检查是比较两个不同计算的结果，来看一下它们是否一致。这被称作 “一致性检查”。

格式化输出 (Format the output)： 格式化调试输出能够更容易定位一个错误。我们在 6.9 一节中看过一个示例。pprint 模块提供了一个 pprint 函数，它可以更可读的格式显示内建类型（pprint 代表 “pretty print”）。

重申一次，你花在搭建脚手架上的时间能减少你花在调试上的时间。

11.9 术语表

映射（mapping）： 一个集合中的每个元素对应另一个集合中的一个元素的关系。

字典（dictionary）： 将键映射到对应值的映射。

键值对（key-value pair）： 键值之间映射关系的呈现形式。

项（item）： 在字典中，这是键值对的另一个名称。

键（key）： 字典中作为键值对第一部分的对象。

值（value）： 字典中作为键值对第二部分的对象。它比我们之前所用的 “值” 一词更具体。

实现（implementation）： 执行计算的一种形式。

哈希表（hashtable）： 用来实现 Python 字典的算法。

哈希函数（hash function）： 哈希表用来计算键的位置的函数。

可哈希的（hashable）： 具备哈希函数的类型。诸如整数、浮点数和字符串这样的不可变类型是可哈希的；诸如列表和字典这样的可变对象是不可哈希的。

查找（lookup）： 接受一个键并返回相应值的字典操作。

逆向查找（reverse lookup）： 接受一个值并返回一个或多个映射至该值的键的字典操作。

raise 语句： 专门印发异常的一个语句。

单元素集合（singleton）： 只有一个元素的列表（或其他序列）。

调用图（call graph）： 绘出程序执行过程中创建的每个栈帧的调用图，其中的箭头从调用者指向被调用者。

备忘录（memo）： 一个存储的计算值，避免之后进行不必要的计算。

全局变量（global variable）： 在函数外部定义的变量。任何函数都可以访问全局变量。

global 语句： 将变量名声明为全局变量的语句。

标记（flag）： 用于说明一个条件是否为真的布尔变量。

声明（declaration）： 类似 global 这种告知解释器如何处理变量的语句。

11.10 练习

Exercise 11.1. 编写一函数，读取 `words.txt` 中的单词并存储为字典中的键。值是什么无所谓。然后，你可以使用 `in` 操作符检查一个字符串是否在字典中。

如果你做过练习 *10.10*，可以比较一下 `in` 操作符和二分查找的速度。
Exercise 11.2. 查看字典方法 `setdefault` 的文档，并使用该方法写一个更简洁的 `invert_dict`。
Exercise 11.3. 将练习 6.2 中的 *Ackermann* 函数备忘录化 (*memoize*),看看备忘录化(*memoization*）是否可以支持解决更大的参数。没有提示!

参考答案
Exercise 11.4. 如果你做了练习 *10.7*，你就已经有了一个叫 `has_duplicates` 的函数，它接受一个列表作为参数，如果其中有某个对象在列表中出现不止一次就返回 `True`。

用字典写个更快、更简单的版本。

参考答案
Exercise 11.5. 两个单词如果反转其中一个就会得到另一个，则被称作"反转对"（参见练习 *8.5* 中的 `rotate_word`。

编写一程序，读入单词表并找到所有反转对。

参考答案
Exercise 11.6. 下面是取自 Car Talk 的另一个字谜题 (*http://www.cartalk.com/content/puzzlers*):

> *This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?*
>
> *Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.' As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.*
>
> *But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?*

> 这是来自一位名叫 *Dan O'Leary* 的朋友的分享。他有一次碰到了一个常见的单音节、有五个字母的单词，它具备以下独特的特性。当你移除第一个字母时，剩下的字母组成了原单词的同音词，即发音完全相同的单词。将第一个字母放回，然后取出第二个字母，结果又是原单词的另一个同音词。那么问题来了，这个单词是什么？

> 接下来我给大家举一个不满足要求的例子。我们来看一个五个字母的单词 *"wrack"*。*W-R-A-C-K*，常用短句为 *"wrack with pain"*。如果我移除第一个字母，就剩下了一个四个字母的单词 *"R-A-C-K"*。可以这么用，*"Holy cow, did you see the rack on that buck! It must have been a nine-pointer!"* 它是一个完美的同音词。如果你把 *"w"* 放回去，移除 *"r"*，你得到的单词是 *"wack"*。这是一个真实的单词，但并不是前两个单词的同音词。
>
> 不过，我们和 *Dan* 知道至少有一个单词是满足这个条件的，即移除前两个字母中的任意一个，将会得到两个新的由四个字母组成的单词，而且发音完全一致。那么这个单词是什么呢？

你可以使用练习 *11.1* 中的字典检查某字符串是否出现在单词表中。

你可以使用 *CMU* 发音字典检查两个单词是否为同音词。从 这里 或 这里 下载。你还可以下载这个脚本，其中提供了一个名叫 `read_dictionary` 的函数，可以读取发音字典，并返回一个将每个单词映射至描述其主要梵音的字符串的 *Python* 字典。

编写一个程序，找到满足字谜题条件的所有单词。

参考答案

第十二章　元组

本章介绍另一个内建的类型 — 元组[1]，同时向您展示列表、字典和元组如何结合使用。后面的章节也会展示关于可变长度参数列表的有用功能，以及汇集和离散操作。

12.1　元组的不可变性

元组是一组值的序列。其中的值可以是任意类型，使用整数索引其位置，因此元组与列表非常相似。而重要的不同之处在于元组的不可变性。

语法上, 元组是用逗号隔开的值的列表：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

尽管不是必须，通常我们在给元组赋值时会用括号把元素封装起来：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

使用单一元素建立元组时，需要在结尾使用一个逗号：

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

括号中仅包含元素（而没有使用逗号结尾）时被赋值的对象不是元组：

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

另一个建立元组的方法是使用内建函数 `tuple`。在没有参数传递时它会产生一个空元组。

[1]值得注意的是，“tuple” 并没有统一的发音，有些人读 “tuh-ple”，音律类似于 “supple”；而有人读 “too-ple” 音律类似于 “quadruple”。

```
>>> t = tuple()
>>> t
()
```

如果实参是一个序列（字符串、列表或者元组），结果将是包含序列内元素的一个元组。

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

因为tuple是内建函数名，所以应该避免将它用于变量名。

列表的大多数操作同样也适用于元组。例如使用方括号索引一个元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

切片操作可以选取一个范围内的元素:

```
>>> t[1:3]
('b', 'c')
```

但是，如果你试图元组中的一个元素，会得到错误信息：

```
>>> t[0] = 'A'
TypeError: object doesn't␣support␣item␣assignment
```

因为元组是不可变的，您无法改变其中的元素。但是您可以使用其他元组替换现有元组：

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

这个语句产生了一个新元组，并且将它赋给了原先的元组t。

关系型操作也适用于元组和其他序列；Python 会首先比较序列中的第一个元素，如果它们相同就会去比较下一组元素，以此往复，直至比值不同。其后的元素（即便是差异很大）也不会再参与比较。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 元组赋值

两个变量互换值的操作通常很有用。传统的，你需要使用一个临时变量。例如为了交换a和b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

这个方法很繁琐；通过**元组赋值**的来实现更为优雅:

```
>>> a, b = b, a
```

等号左侧是变量构成的元组；右侧是元组赋值的表达式。每个值都被赋给了对应的要互换的变量。变量被重新赋值前，右侧的表达式会被优先运行。

使用元组赋值，左右的变量数必须相同：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

一般说来，元组赋值的右侧表达式可以是任意类型（字符串、列表或者元组）的序列。例如，将一个电子邮箱地址分成用户名和域名，你可以：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

split函数返回的对象是一个包含两个元素的列表；第一个元素被赋给了uname的变量，第二个被赋给了domain。

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3 元组作为返回值

严格地说，一个函数只能返回一个值，但是如果以元组作为这个返回值，其效果等同于返回多个值。例如，你想对两个整数做除法，计算出商和余数，依次计算出x/y和x%y是很低效的。更好的方法就是同时计算出它们。

内建函数divmod接受两个参数，返回包含两个值的元组 — 输入参数做除法的商和余数。您可以使用元组来存储返回值:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

或者使用元组赋值分别存储它们：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

下面是另一个返回元组作为结果的函数例子：

```
def min_max(t):
    return min(t), max(t)
```

max 和 min 是用于找出一组元素序列中最大值和最小值的内建函数，min_max函数同时计算出它们并组装成元组返回结果。

12.4 可变长度参数元组

函数可以同时接受多个参数。以 * 开头的定义参数可以将输入的参数 汇集到一个元组中。例如 printall 可以接受任意数量的参数，并且打印出来:

```
def printall(*args):
    print(args)
```

汇集的形参可以使用任意名字，传统上使用args. 以下显示了这个函数的调用效果：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

离散 **scatter** 是汇集的补充。如果你有一个值的序列，并且希望将其作为多个参数传递给一个函数，你可以使用运算符*。例如，divmod 需要接受两个实参；一个元组则无法作为参数传递进去：

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

但是如果将这个元组打散，它就可以被传递进函数：

```
>>> divmod(*t)
(2, 1)
```

多数内建函数使用可变长度参数元组。例如，max 和 min 可以取任意数量的参数。

```
>>> max(1, 2, 3)
3
```

但是求和操作sum并不如此：

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

您可以尝试写一个叫做 sumall的函数作为练习，使它能够接受任何数量的传参并返回它们的和。

12.5 列表和元组

zip 是一个内建函数，用于将两个或多个序列组装成包含元组的列表返回出来，每个元组包含了各个序列中相对位置的一个元素。这个函数的起名来源于名词拉链 (zipper), 形象的显示年对应两列对应位置的牙齿组合起来。

下面例子显示了组合字符串和列表操作：

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

输出的结果是一个可以通过每个内在的元组对进行迭代的 zip 对象。zip函数最常见用法就是基于for循环的迭代遍历:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

zip对象是一个友善的**迭代器**，后者是指任何一种能够按照某个序列迭代的对象。迭代器在某些方面与列表非常相似，不同之处在于你无法通过索引来选择迭代器中的某个元素。

如果你想对zip对象使用列表的操作和方法，你可以通过zip对象创造一个列表：

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

结果就是产生了一个包含若干元组的列表；在这个例子中，每个元组又包含了字符串中的一个字符和列表t 中对应的一个元素。

如果用于创建zip的序列长度不一，返回的对象的长度以最短序列的长度为准。

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

您可以通过元组赋值在for循环中遍历包含元组的列表：

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

循环中的每次执行，Python 会选择列表中的下一个元组，并将其内容赋给 letter 和 number。因此循环打印的输出会是这样：

```
0 a
1 b
2 c
```

如果将zip、for和元组赋值结合起来使用，您会得出一个有用的惯用方法用于同时遍历两个（甚至多个）序列。如下例，has_match 接受两个序列，t1和t2，并返回一个真值True如果存在满足判别式 t1[i] == t2[i]的索引i：

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

如果需要遍历一个序列的元素以及它们的索引号，您可以使用内建函数enumerate：

```
for index, element in enumerate('abc'):
    print(index, element)
```

enumerate的返回结果是一个枚举对象 (enumerate object)，它可基于一个包含若干个对的序列进行迭代，每个对包含了（从 0 开始计数）的索引号和对应的元素。在刚才的例子中，对应的输出结果会和上次一样：

```
0 a
1 b
2 c
```

12.6 字典和元组

字典对象有一个内建方法叫做itmes，它返回一个以元组形式存放的键-值对的序列。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

其结果是一个dict_itmes对象，其实质是一个可以通过键-值对迭代的迭代器。您可以在for循环中像这样使用它:

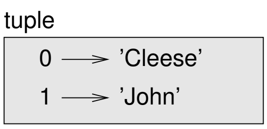

图 12.1: State diagram.

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

和字典对象相似，dict_itmes内部的项是无序存放的。

另一方面，您可以使用元组的列表初始化一个新的字典：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

dict和zip的结合使用产生了一个简洁的字典生成法:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

字典的update方法也接受元组的列表，并作为键-值对把它们加入到该字典中去。

更加常用的方法是在字典中使用元组作为键（因为列表做不了键）。例如，一个电话簿希望基于用户的姓（last）、名（first）对来映射号码（number），假设我们已经定义了last, first 和 number三个变量，我们可以这样实现映射：

```
directory[last, first] = number
```

方括号中的表达式是一个元组。为我们可以通过元组赋值来遍历这个字典：

```
for last, first in directory:
    print(first, last, directory[last,first])
```

该循环遍历directory中的键，它们其实是元组。它将元组的元素赋给last和first，然后打印出姓名和对应的电话号码。

用两个状态图来表述这些元组。细致的说，索引号和对应元素就像列表一样存放在元组中。例如，元组('Cleese', 'John')可像图 12.1一样存放。

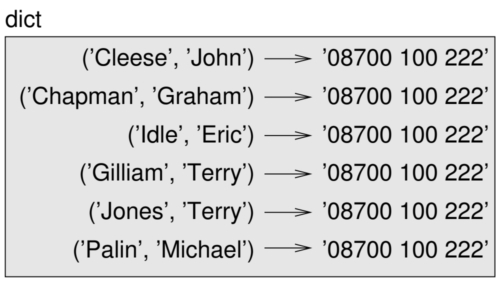

图 12.2: State diagram.

在大图中，我们忽略这些细节。该电话簿的结构图可能像图 12.2一样。

因此，Python 风格的元组用法可用这两幅图来描述。此图中的电话号码是 BBC 的投诉热线，请不要拨打它。

12.7 序列嵌套

我们已经谈过了包含元组的列表，事实上，本章大多数例子也适用于列表嵌套列表、元组嵌套元组，以及元组嵌套列表。为了避免一一穷举这类可能的嵌套组合，我们简称为序列嵌套。

在很多情况下，不同类型的序列（字符串、列表、元组）可以互换使用。因此，我们如何选用合适的嵌套对象呢？

首先，显而易见的是字符串的使用范围比其他序列更为有限，因为它的所有元素都是字符，且字符串不可变。如果你希望能够改变字符在字符串中的位置，使用列表嵌套字符比较合适。

列表比元组更常见，这源于它们可变性的易用。但是有些情况下我们不得不更青睐元组：

1. 在一些情况下（例如`return`语句），从句式上生成一个元组比列表要简单。
2. 如果你想使用一个序列作为字典的键，那么你必须使用元组或字符串这样的不可变类型。
3. 如果你向函数传入一个序列作为参数，那么使用元组以降低由于别名而产生的意外行为的可能性。

正由于元组的不可变性，元组没有类似于列表中的 `sort`（排序）和 `reverser`（逆序）这样的方法。然而 Python 提供了内建函数 `sorted`，用于对任意序列排序并输出相同元素的列表，以及 `reversed`，用于对序列逆向排序并生成一个可以遍历的迭代器。

12.8 调试

列表、字典和元组都是数据结构 (**data structures**) 的实例；本章中我们开始接触到复合数据结构 (**compound data structures**)，如：列表嵌套元组，又如使用元组作为键而列表作为值的字典。复合数据结构非常实用，然而使用时容易出现所谓的形状错误 (**shape errors**)，也就是说由于数据结构的类型、大小或结构问题而引发的错误。例如，当你希望使用封装整数的列表时却用成了没被列表包含的一串整数。

为了方面调试这类错误，笔者编写了一个叫做 structshape 的模块，它提供了一个名为 structshape的函数，用于接受并分析数据结构对象作，并返回描述它形状的文字信息。你可以在此处下载到它 (http://thinkpython2.com/code/structshape.py)。

这里是它分析简单列表的结果展示：

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list␣of␣3␣int'
```

更完美的程序应该显示 "list of 3 ints"，但是忽略英文复数使程序简单的多。我们再看一个列表嵌套的例子：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list␣of␣3␣list␣of␣2␣int'
```

如果列表内嵌套的元素不是相同类型，structshape 会按类型的组将它们归并：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list␣of␣(3␣int,␣float,␣2␣str,␣2␣list␣of␣int,␣int)'
```

以下是一个元组的例子：

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list␣of␣3␣tuple␣of␣(int,␣str)'
```

下面，一个包含 3 个映射整数到字符串的键值对的字典被分析：

```
>>> d = dict(lt)
>>> structshape(d)
'dict␣of␣3␣int->str'
```

因此如果你对使用的数据结构有疑惑，可以使用structshape来帮助解析。

12.9 术语表

元组： 一组不可变的元素的序列。

元组赋值： 一种通过赋值方式，通过等号右侧的序列向等号左侧的一组变量的元组进行赋值。右侧许两种的每个元素会被计算，然后赋给左侧元组中对应的变量。

汇集： 组装可变长度变量元组的一种操作。

分散： 将一个序列变换成一个参数列表的操作。

zip 对象： 使用内建函数`zip`所返回的结果，它是一个可通过元组序列逐个迭代的对象。

迭代器： ：一种可以通过一个序列逐个迭代的对象，但是它并不提供列表的某些操作和方法。

数据结构： 一个有相关关联的数据的集合，通常使用列表、字典和元组等综合构成。

形状错误： 由于数据结构的类型、大小或结构问题而引发的错误。

12.10 练习

Exercise 12.1. 写一个名为most_frequent 的函数，它接受字符串并按字母降序打印出字符出现频率。找一些不同语言的文本样本来试试看不同语言文本间区别。将你的结果和维基百科的 字母频率表相比较。

参考答案

Exercise 12.2. 易位构词游戏 (anagrams)!

1. 编写一个程序使之能从文件以列表形式读入单词（参考章节 9.1）并且打印出所有符合异位构词的组合。

   下面是一个输出异位构词的样例：

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   提示：也许你可以建立一个字典用于映射一个字符集合到一个该集合可异位构词的词汇集合。

2. 改写前面的程序，使之首先打印包含异位构词数量最多的词汇列表，第二多次之，依次按异位构词数量排列。

3. Scrabble 游戏 中，“bingo”...

   参考答案

Exercise 12.3. 如果两个单词中的某一单词可以通过调换两个字母变为另一个，这两个单词就构成了 “*metatheisi pair*”；比如 “*converse*” 和 “*conserve*”。写一个程序来找出给定字典里所有的 “*metatheisi pair*”。提示：不用测试所有的单词组合，也不用测试所有的字母调换组合。

参考答案

这个练习受*http://puzzlers.org*的案例启发而成。
Exercise 12.4. 另一个猜谜题 *car talk puzzler* :

> 世界上哪个最长的英文单词，当你每一次从中删掉一个字母以后，剩下的字符仍然能构成一个单词？
>
> 被删掉的字母可以位于首尾或是中间，但不允许重新去排列剩下的字母，这样你得到一个新单词。这样一直下去最终你只剩一个字母，并且它也是一个单词 — 一个你可以在字典里查到的单词。我们想找到最初的这个单词可以最长可以多长，有多少个字母构成？
>
> 我先给出一个短小的例子：“*Sprite*”，从 *sprite* 起，我们可以拿掉中间的 ‘*r*’ 从而获得单词 *spite*，拿去字母 ‘*e*’ 得到 *spit*，再去掉 ‘*s*’ 剩下 *pit*，*it*，最后 *I*。

写一个程序按照这种规则找到所有可以缩词的单词，然后看看其中哪个词最长。

以下是对这个稍具挑战的练习的一些建议：

1. 可能你需要写一个函数将输入单词的所有 “子词”（即拿掉一个字母后所有可能的新词）以列表形式输出。
2. 递归的看，如果一个可被缩词的单词是另一个单词的子词，那另一个单词也可被缩。我们可从空字符串开始考虑。
3. 我们提供的词汇表（`words.txt`）并未包含诸如 ‘*I*’、‘*a*’ 这样的单个字母词汇，因此你可能需要加上它们。
4. 为了提高你程序的性能，你可能需要暂存好已被发现的可被缩词的词汇。

参考答案

第十三章　Case study: data structure selection | 案例研究：数据结构选择

At this point you have learned about Python's core data structures, and you have seen some of the algorithms that use them. If you would like to know more about algorithms, this might be a good time to read Chapter B. But you don't have to read it before you go on; you can read it whenever you are interested.

目前为止，你已经学完了 Python 的核心数据结构，同时你也接触了利用到这些数据结构的一些算法。如果你希望学习更多算法知识，那么现在是阅读附录 B的好时机。但是不必急着马上读，什么时候感兴趣了再去读即可。

This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

本章是一个案例研究，同时给出了一些习题，目的是启发你思考如何选择数据结构，并练习数据结构使用。

13.1　Word frequency analysis | 词频分析

As usual, you should at least attempt the exercises before you read my solutions.

和之前一样，在查看答案之前，你可以试着解答一下这些习题。

Exercise 13.1. *Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.*

编写一个程序，读取一个文件，将每一行转换成单词列表，删掉单词中的空格和标点，然后将它们转换为小写字母。

Hint: The `string` *module provides a string named* `whitespace`*, which contains space, tab, newline, etc., and* `punctuation` *which contains the punctuation characters. Let's see if we can make Python swear:*

提示：`string` 模块提供了名为 `whitespace` 的字符串，其包括空格、制表符、新行等等，以及名为 `punctuation` 的字符串，其包括标点字符。试试能否让 Python 说脏话：

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Also, you might consider using the string methods `strip`*,* `replace` *and* `translate`*.*

同时，你可以考虑使用字符串方法 *strip*、*replace* 和 *translate*。

Exercise 13.2. *Go to Project Gutenberg (*`http://gutenberg.org`*) and download your favorite out-of-copyright book in plain text format.*

前往古腾堡项目 (*Project Gutenberg*)，以纯文本格式下载你喜欢的已无版权保护的图书。

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before. Then modify the program to count the total number of words in the book, and the number of times each word is used.

修改前面习题的程序，读取你下载的书，跳过文件开始的头部信息，像之前那样处理其余的单词。然后修改程序，计算书中单词的总数，以及每个单词使用的次数。

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

打印该书使用单词的总数。比较不同年代、不同作者写的书。哪个作者使用的词汇量最大？

Exercise 13.3. *Modify the program from the previous exercise to print the 20 most frequently used words in the book.*

修改上一个习题中的程序，打印书中最常使用的 20 个单词。

Exercise 13.4. *Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that* should *be in the word list, and how many of them are really obscure?*

修改上一个习题中的程序，读取一个单词列表（见 9.1 节），然后打印书中所有没有出现在该单词表中的单词。它们中有多少是拼写错误的？有多少是词表中**应该** 包括的常用词？有多少是生僻词？

13.2 Random numbers | 随机数

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

给定相同的输入，大多数计算机程序每次都会生成相同的输出，它们因此被称作确定性的 (**deterministic**) 。确定性通常是个好东西，因为我们期望相同的计算产生相同的结果。然而，对于有些应用，我们希望计算机不可预知。游戏是一个明显的例子，但是不限于此。

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms

that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

让程序具备真正意义上的非确定性并不容易，但是有办法使它至少看起来是不确定的。其中之一是使用生成伪随机 (pseudorandom) 数的算法。伪随机数不是真正的随机数，因为它们由一个确定性的计算生成，但是仅看其生成的数字，不可能将它们和随机生成的相区分开。

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

`random` 模块提供了生成伪随机数（下文中简称为 "随机数"）的函数。

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

函数 `random` 返回一个 0.0 到 1.0 之间的随机浮点数（包括 0.0 ，但是不包括 1.0 ）。每次调用 `random` ， 你获得一个长序列中的下一个数。举个例子，运行此循环：

```
import random
for i in range(10):
    x = random.random()
    print(x)
```

The function `randint` takes parameters `low` and `high` and returns an integer between `low` and `high` (including both).

函数 `randint` 接受参数 `low` 和 `high` ， 返回一个 `low` 和 `high` 之间的整数（两个都包括）。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

你可以使用 `choice` ， 从一个序列中随机选择一个元素：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

`random` 模块提供的函数，还可以生成符合高斯、指数、伽马等连续分布的随机值。

Exercise 13.5. *Write a function named* `choose_from_hist` *that takes a histogram as defined in Section 11.2 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:*

编写一个名为 *choose_from_hist* 的函数，其接受一个如 11.2 一节中定义的 *histogram* 对象作为参数，并从该对象中返回一个随机值，其选择概率和值出现的频率成正比。例如：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

your function should return `'a'` *with probability* 2/3 *and* `'b'` *with probability* 1/3.

你的函数返回 'a' 的概率应该是 2/3 ，返回 'b' 的概率应该是 1/3 。

13.3 Word histogram | 单词直方图

You should attempt the previous exercises before you go on. You can download my solution from `http://thinkpython2.com/code/analyze_book1.py`. You will also need `http://thinkpython2.com/code/emma.txt`. Here is a program that reads a file and builds a histogram of the words in the file:

在继续下面的习题之前，你应该尝试完成前面的练习。你可以下载 我的答案。你还需要下载这个文件 。下面这个程序将读取一个文件，并建立文件中单词的直方图：

```
import string
def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist
def process_line(line, hist):
    line = line.replace('-', '␣')
    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1
hist = process_file('emma.txt')
```

This program reads `emma.txt`, which contains the text of *Emma* by Jane Austen.

该程序读取 emma.txt ，其包括 Jane Austen 的小说《*Emma*》文本。

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `hist` is being used as an accumulator.

process_file 循环读取文件的每行，依次把它们传递给 process_line 。直方图 hist 被用作一个累加器。

process_line uses the string method replace to replace hyphens with spaces before using split to break the line into a list of strings. It traverses the list of words and uses strip and lower to remove punctuation and convert to lower case. (It is a shorthand to say that strings are "converted"; remember that strings are immutable, so methods like strip and lower return new strings.)

process_line 使用字符串的 replace 方法将连字符替换成空格。它会遍历单词列表，并使用 strip 和 lower 来删除标点以及将单词转换为小写。（"转换" 只是一种简略的说法；记住，字符串是不可变的，所以类似 strip 和 lower 这样的方法其实返回的是新字符串。）

Finally, process_line updates the histogram by creating a new item or incrementing an existing one.

最后，process_line 通过生成一个新的项或者递增一个已有的项来更新直方

To count the total number of words in the file, we can add up the frequencies in the histogram:

我们可以通过累加直方图中的频率，来统计文件中的单词总数：

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

不同单词的数量恰好是词典中项的数目：

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

以下打印结果的代码：

```
print('Total␣number␣of␣words:', total_words(hist))
print('Number␣of␣different␣words:', different_words(hist))
```

And the results:

其结果是：

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words | 最常用单词

To find the most common words, we can make a list of tuples, where each tuple contains a word and its frequency, and sort it. The following function takes a histogram and returns a list of word-frequency tuples:

为了找到最常用的单词，我们可以使用元组列表，其中每个元组包含单词和它的频率，然后排序这个列表。

下面的函数接受一个直方图并且返回一个单词-频率的元组列表：

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))
    t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the ten most common words:

每一个元组中，频率在前，所以这个列表是按照频率排序。下面是输出最常用的十个单词的循环：

```
t = most_common(hist)
print('The␣most␣common␣words␣are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

I use the keyword argument `sep` to tell `print` to use a tab character as a "separator", rather than a space, so the second column is lined up. Here are the results from *Emma*:

这里我通过关键词参数 `sep`，让 `print` 使用一个制表符 (Tab) 而不是空格键作为分隔符，所以第二行将对齐。下面是对《*Emma*》的分析结果：

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

This code can be simplified using the `key` parameter of the `sort` function. If you are curious, you can read about it at `https://wiki.python.org/moin/HowTo/Sorting`.

当然，这段代码也可以通过 `sort` 函数的参数 `key` 进行简化。如果你感兴趣，可以阅读这篇文章 。

13.5 Optional parameters | 可选形参

We have seen built-in functions and methods that take optional arguments. It is possible to write programmer-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

我们已经见过接受可变数量实参的函数和方法了。程序员也可以自己定义具有可选实参的函数。例如，下面就是一个打印直方图中最常见单词的函数。

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The␣most␣common␣words␣are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The default value of num is 10.

第一个形参是必须的；第二个是可选的。num 的 ** 默认值（default value）** 是 10。

If you only provide one argument:

如果你只提供了一个实参:

```
print_most_common(hist)
```

num gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

num gets the value of the argument instead. In other words, the optional argument overrides the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

如果一个函数同时有必选和可选两类形参，则所有的必选形参必须首先出现，可选形参紧随其后。

13.6 Dictionary subtraction | 字典差集

Finding the words from the book that are not in the word list from words.txt is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

从书中找到所有没出现在词表 words.txt 中的单词，可以认为是一个差集问题；也就是，我们应该从一个集合中（书中的单词）找到所有没出现在另一个集合中（列表中的单词）的单词。

subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keys from d1 that are not in d2. Since we don't really care about the values, we set them all to None.

`subtract` 接受词典 `d1` 和 `d2`，并返回一个新的词典，其包括 `d1` 中的所有没出现在 `d2` 中的键。由于并不真正关心值是什么，我们将它们都设为 `None`。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in `words.txt`, we can use `process_file` to build a histogram for `words.txt`, and then subtract:

为了找到书中没有出现在 `words.txt` 中的单词，我们可以使用 `process_file` 来为 `words.txt` 构建一个直方图，然后使用 `subtract`：

```
words = process_file('words.txt')
diff = subtract(hist, words)
print("Words␣in␣the␣book␣that␣aren't␣in␣the␣word␣list:")
for word in diff.keys():
    print(word, end='␣')
```

Here are some of the results from *Emma*:

这是针对小说《*Emma*》的部分运行结果：

```
Words in the book that aren't␣in␣the␣word␣list:
rencontre␣jane's blanche woodhouses disingenuousness
friend's␣venice␣apartment␣...
```

Some of these words are names and possessives. Others, like “rencontre”, are no longer in common use. But a few are common words that should really be in the list!

这些单词中，一些是名字和名词所有格。如“rencontre”这样的其他单词已经不常使用了。但是有一些真的应该包括在列表中！

Exercise 13.6. *Python provides a data structure called* `set` *that provides many common set operations. You can read about them in Section 19.5, or read the documentation at* *http://docs.python.org/3/library/stdtypes.html#types-set*. *Write a program that uses set subtraction to find words in the book that are not in the word list. Solution:* *http://thinkpython2.com/code/analyze_book2.py*.

Python 提供了一个叫做集合 (set) 的数据结构，支持许多常见的集合操作。你可以前往第十九章阅读相关内容，或者阅读官方文档 。

编写一个程序，使用集合的差集操作来找出一本书中不在 *work* `list` 中的单词。

参考答案

13.7 Random words | 随机单词

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

如果想从直方图中随机选择一个单词，最简单的算法是创建一个列表，其中根据其出现的频率，每个单词都有相应个数的拷贝，然后从该列表中选择单词：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)
    return random.choice(t)
```

The expression `[word] * freq` creates a list with `freq` copies of the string `word`. The `extend` method is similar to `append` except that the argument is a sequence.

表达式 `[word] * freq` 创建一个具有 `freq` 个 `word` 字符串拷贝的列表。除了它的实参要求是一个序列外，`extend` 方法和 `append` 方法很像。

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big. An alternative is:

该算法能够满足要求，但是效率不够高；每次你选择一个随机单词，它都重建列表，这个列表和原书一样大。一个明显的改进是，创建列表一次，然后进行多次选择，但是该列表仍然很大。

1. Use `keys` to get a list of the words in the book.
2. 使用 `keys` 来获得该书中单词的列表。
3. Build a list that contains the cumulative sum of the word frequencies (see Exercise 10.2). The last item in this list is the total number of words in the book, n.
4. 创建一个包含单词频率累积和的列表（见练习 10.2）。此列表的最后一项是书中单词的数目 n 。
5. Choose a random number from 1 to n. Use a bisection search (See Exercise 10.10) to find the index where the random number would be inserted in the cumulative sum.
6. 选择一个从 1 到 n 的随机数。使用二分搜索（见10.10节）找到该随机数应该被在累积和中插入的索引。
7. Use the index to find the corresponding word in the word list.
8. 使用该索引从单词列表中找到相应的单词。

Exercise 13.7. *Write a program that uses this algorithm to choose a random word from the book. Solution:* `http://thinkpython2.com/code/analyze_book3.py`.

编写一个使用该算法从书中选择一个随机单词的程序。参考答案

13.8 Markov analysis | 马尔科夫分析

If you choose words from the book at random, you can get a sense of the vocabulary, but you probably won't get a sentence:

如果你从书中随机选择单词，那么你会大致了解其使用的词汇，但可能不会得到一个完整的句子：

```
this the small regard harriet which knightley's␣it␣most␣things
```

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.

一系列随机单词很少有意义，因为相邻的单词之间没有关系。例如，在一个真实的句子中，你可能期望 "the" 这样的冠词后面跟着的是一个形容词或者名词，而大不可能会是一个动词或者副词。

One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the words that might come next. For example, the song *Eric, the Half a Bee* begins:

衡量相邻单词关系的方法之一是马尔科夫分析法，对于一个给定的单词序列，马尔科夫分析法将给出接下来单词的概率。例如，歌曲 *Eric, the Half a Bee* 的开头是：

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

In this text, the phrase "half the" is always followed by the word "bee", but the phrase "the bee" might be followed by either "has" or "is".

在此文本中，短语 "half the" 后面总是跟着单词 "bee"，但是短语 "the bee" 则可能跟着 "has" 或者 "is"。

The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").

马尔科夫分析的结果是从每个前缀（如 "half the" 和 "the bee"）到所有可能的后缀（如 "has" 和 "is"）的映射。

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix "Half a", then the next word has to be "bee", because the prefix only appears once in the text. The next prefix is "a bee", so the next suffix might be "philosophically", "be" or "due".

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

给定此映射，你能够通过以任意前缀开始并从可能的后缀中随机选择一个的方法，来生成一个随机文本。接下来，你可以将前缀的结尾和新的后缀组合成下一个前缀，并重复下去。

例如，如果你以前缀 **"Half a"** 开始，然后下一个但是必须是 **"bee"**，因为此前缀在文本中仅出现一次。下一个前缀是 **"a bee"**，所以下一个后缀可能是 **"philosophically"**, **"be"** 或 **"due"**。

此例中，前缀的长度总是 **2**，但是你可以以任意前缀长度进行马尔科夫分析。前缀的长度被称作此分析的 "阶"。
Exercise 13.8. *Markov analysis:* 马尔科夫分析：

1. *Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.*

2. 编写一个程序，从一个文件中读取文本并执行马尔科夫分析。结果应该是一个字典，即从前缀映射到一个可能的后缀集合。此后缀集合可以是一个列表、元组或字典；你需要做出合适的选择。你可以用长度为 2 的前缀测试程序，但是在编写程序时，应确保其很容易支持其它长度。

3. *Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from* Emma *with prefix length 2:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

   For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

   What happens if you increase the prefix length? Does the random text make more sense?

4. 在前面的程序中添加一个函数，基于马尔科夫分析生成随机文本。下面是使用《Emma》执行前缀为 2 的马尔科夫分析生成的示例：

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

在此例中，我保留了附在词后面的标点符号。从语法上看，结果几乎是正确的，但不完全。语义上讲，它几乎有意义，但也不完全。

如果你增加前缀的长度，会发生什么？随机文本更有意义是么？

5. *Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.*

6. 一旦程序正常运行，你可以想尝试一下混搭：如果你组合两本或更多书中的文本，你生成的随机文本将以有趣的方式混合这些书中的词汇和短语。

Credit: This case study is based on an example from Kernighan and Pike, The Practice of Programming, *Addison-Wesley, 1999.*

致谢：此案例研究基于 *Kernighan* 与 *Pike* 所著的《The Practice of Programming》一书中的示例。

You should attempt this exercise before you go on; then you can can download my solution from `http://thinkpython2.com/code/markov.py`. You will also need `http://thinkpython2.com/code/emma.txt`.

在继续阅读之前，你应该尝试解决这个习题；你可以从 此处下载我的答案。你还需要下载**Emma 的文本** 。

13.9 Data structures | 数据结构

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

使用马尔科夫分析生成随机文本很有趣，但是上面那道习题的目的是：学习数据结构选择。在解答上述习题时，你不得不选择：

- **How to represent the prefixes.**
- **How to represent the collection of possible suffixes.**
- **How to represent the mapping from each prefix to the collection of possible suffixes.**

- 如何表示前缀。
- 如何表示可能后缀的集合。
- 如何表示从前缀到可能后缀集合的映射。

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

最后一个选择很简单：明显应该选择字典作为键至对应值的映射。

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

对于前缀，最明显的选择是字符串、字符串列表或者字符串元组。

For the suffixes, one option is a list; another is a histogram (dictionary).

对于后缀，一个选择是列表；另一个是直方图（字典）。

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is "Half a", and the next word is "bee", you need to be able to form the next prefix, "a bee".

你如何选择呢？第一步是考虑对每个数据结构你需要实现的操作。对于前缀，我们需要能从头部删除单词，并在结尾处加入单词。例如，如果当前的前缀是 "Half a"，下一个词是 "bee"，你需要能构成下一个前缀 "a bee"。

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can't append or remove, but you can use the addition operator to form a new tuple:

你的第一个选择可能是列表，因为它能很容易的增加和删除元素，但是我们也需要让前缀作为字典的键，这就排除了列表。使用元组，你不能追加或删除元素，但是你能使用加号运算符来形成一个新的元组：

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` takes a tuple of words, `prefix`, and a string, `word`, and forms a new tuple that has all the words in `prefix` except the first, and `word` added to the end.

`shift` 接受一个单词元组 `prefix` 和一个字符串 `word` ，并形成一个新的元组，其具有 `prefix` 中除第一个单词外的全部单词，然后在结尾增加 `word` 。

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

对于后缀的集合，我们需要执行的运算包括增加一个新的后缀（或者增加一个已有后缀的频率），并选择一个随机后缀。

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 13.7).

对于列表或者直方图，增加一个新的后缀一样容易。从列表中选择一个随机元素很容易；在直方图中选择的难度更大（见练习 13.7）。

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes

there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the `in` operator is faster for dictionaries than for lists, at least when the number of elements is large.

目前为止，我们主要讨论实现的难易，但是选择数据结构时还要考虑其它因素。一个是运行时间。有时，一个数据结构比另一个快有理论依据；例如，我提到过 {`in` 运算符对于字典比对列表要快，至少当元素的数目很大的时候。

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called benchmarking. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the `profile` module, that can identify the places in a program that take the most time.

但是通常你事先不知道哪个实现更快。一个选择是两个都实现，然后再看哪个更快。此方法被称作基准测试 (benchmarking) 。另一个更实际的选择是选择最容易实现的数据结构，然后看它对于拟定的应用是否足够快。如果是的话，就不需要继续了。如果不是，可以使用一些工具，如 `profile` 模块，识别程序中哪处最耗时。

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

另一个要考虑的因素是存储空间。例如，使用直方图表示后缀集合可能用更少的空间，因为无论一个单词在文本中出现多少次，你只需要存储它一次。在一些情况下，节省空间也能让你的程序更快，极端情况下，如果内存溢出，你的程序可能根本不能运行。但是对于许多应用，空间是运行时间之后的第二位考虑。

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

最后一点：在此讨论中，我暗示了我们应该使用一种数据结构同时进行分析和生成。但是既然这些是独立的步骤，使用一种数据结构进行分析，然后采用另一种结构进行生成也是可能的。如果生成节省的时间超过了转化花费的时间，这也会提高程序的性能。

13.10 Debugging | 调试

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

在调试一个程序的时候，特别是调试一个很难的错误时，应该做到以下五点：

Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.

Ruminating: Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Rubberducking: If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called rubber duck debugging. I am not making this up; see `https://en.wikipedia.org/wiki/Rubber_duck_debugging`.

Retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

细读： 检查你的代码，仔细地阅读，并且检查是否实现了你的期望。

运行： 通过修改和运行不同的版本来不断试验。通常，如果你在程序中正确的地方打印了正确的东西，问题会变得很明显，但是有时你不得不搭建一些脚手架。

思考： 花些时间思考！错误的类型是什么：语法、运行时、语义？你从错误信息或者程序的输出中能获得什么信息？什么类型的错误能引起你看到的问题？问题出现前，你最后的修改是什么？

小黄鸭调试法（rubberducking）： 如果将你的问题解释给别人听，有时你会发现在解释完问题之前就能找到答案。你通常并不需要真的去问另外一个人；你可以对着一个小黄鸭说。这就是著名的小黄鸭调试法 (rubber duck debugging) 的由来。这可不是我编造的，看看维基的解释。

回退： 有时候，最好的做法是回退，撤销最近的修改，直到你回到一个能运行并且你能理解的程序。然后你可以开始重建。

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

初级程序员有时陷入这些步骤之一，忘记了还可以做其他的事情。事实上，每种方法都有失败的可能。

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

例如，如果程序是一个排版错误，读代码可能有帮助，但是如果问题是概念理解错误，则未必是这样。如果你不理解程序要做什么，可能读 100 遍程序都不会发现错误，因为错误在你的头脑中。

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

试验可能会有帮助，特别是如果你运行简单短小的测试。但是，如果你不思考或者阅读你的代码，就直接进行实验，你可能陷入一种我称为"随机游走编程"的模式。这指的是随机修改，直到程序通过测试。不用说，随机游走编程会花费很长的时间。

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

你必须花时间思考。调试就像是一门实验科学。你应该至少有一个关于问题是什么的假设。如果有两个或者更多的可能，试着考虑利用测试消除其中一个可能。

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

但是，如果有太多的错误，或者你正试图修复的代码太大、太复杂，即使最好的调试技巧也会失败。有时，最好的选择是回退，简化程序，直到你获得一个正常运行并且能理解的程序。

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

初级程序员经常不愿意回退，因为他们舍不得删除一行代码（即使它是错误的）。如果能让你好受些，在你开始精简之前，可以将你的代码拷贝到另一个文件中。然后你再把修改后的代码一块一块地拷贝回去。

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

发现一个错误，需要阅读、运行、沉思、和时而的回退。如果其中某个步骤没有进展，试一下其它的。

13.11 Glossary | 术语表

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

确定性的（**deterministic**）： 指的是给定相同的输入，一个程序每次运行的结果是一样的。

pseudorandom: Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

伪随机（**pseudorandom**）： 指的是一串数字看上去是随机的，但是实际是由一个确定性程序生成的。

default value: The value given to an optional parameter if no argument is provided.

默认值： 没有提供实参时，赋给可选形参的值。

override: To replace a default value with an argument.

覆盖： 用实参替代默认值。

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

基准测试（**benchmarking**）： 通过可能的输入样本对使用不同数据结构的实现进行测试，从而选择数据结构的过程。

rubber duck debugging: Debugging by explaining your problem to an inanimate object such as a rubber duck. Articulating the problem can help you solve it, even if the rubber duck doesn't know Python.

小黄鸭调试法（**rubberducking**）： 通过向小黄鸭这样的非生物体解释你的问题来进行调试。清晰地陈述问题可以帮助你解决问题，即使小黄鸭并不懂 **Python**。

13.12 Exercises | 练习

Exercise 13.9. *The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.*

单词的"秩"是指它在按照单词频率排序的列表中的位置：出现频率最高的单词，它的秩是 1，频率第二高的单词，它的秩是 2，以此类推。

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages (`http://en.wikipedia.org/wiki/Zipf's_law`). Specifically, it predicts that the frequency, f, of the word with rank r is:

Zipf 定律 描述了自然语言中秩和单词出现频率的关系。特别是，它预测对于秩为 r 的单词，其出现的频率 f 是：

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

其中，s 和 c 是依赖于语言和文本的参数。如果在上述等式两边取对数的话，你可以得到：

$$\log f = \log c - s \log r$$

So if you plot log f versus log r, you should get a straight line with slope $-s$ and intercept log c.

因此，如果绘出 *log f* 和 *log r* 的图像，你可以得到一条以 $-s$ 为斜率、以 c 为截距的直线。

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with log f and log r. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s?

编写一个程序，从文件中读取文本，计算单词频率，倒序输出每个单词，一个单词一行，同时在这行输出对应的 *log f* 和 *log r*。使用你喜欢的绘图程序，画出结果并检查是不是形成一条直线。你可以估算出 s 的值吗？

Solution: `http://thinkpython2.com/code/zipf.py`. To run my solution, you need the plotting module `matplotlib`*. If you installed Anaconda, you already have* `matplotlib`*; otherwise you might have to install it.*

参考答案

你需要安装绘图模块 *matplotlib* 才能运行我的答案。当然如果你安装了 *Anaconda*，你就已经有了 *matplotlib*；否则你需要安装它。

第十四章　文件

本章将介绍 “持久” (persistent) 程序的概念，即永久储存数据的程序，并说明如何使用不同种类的永久存储形式，例如文件和数据库。

14.1　持久化

目前我们所见到的大多数程序都是临时的（transient），因为它们只运行一段时间并输出一些结果，但当它们结束时，数据也就消失了。如果你再次运行程序，它将以全新的状态开始。

另一类程序是持久的 (persistent)：它们长时间运行（或者一直在运行）；它们至少将一部分数据记录在永久存储（如一个硬盘中）；如果你关闭程序然后重新启动时，它们将从上次中断的地方开始继续。

持久程序的一个例子是操作系统，在一台电脑开机后的绝大多数时间系统都在运行。另一个例子是网络服务器，不停地在运行，等待来自网络的请求。

程序保存其数据的一个最简单方法，就是读写文本文件。我们已经接触过读取文本文件的程序；在本章，我们将接触写入文本的程序。

另一种方法是使用数据库保存程序的状态。本章我将介绍一个简单的数据库，以及简化存储程序数据过程的 `pickle` 模块。

14.2　读取和写入

文本文件是储存在类似硬盘、闪存、或者 CD-ROM 等永久介质上的字符序列。我们在9.1 节中接触了如何打开和读取文件。

要写入一个文件，你必须在打开文件时设置第二个参数来为 `'w'` 模式：

```
>>> fout = open('output.txt', 'w')
```

如果该文件已经存在，那么用写入模式打开它将会清空原来的数据并从新开始，所以要小心！如果文件不存在，那么将创建一个新的文件。

`open` 会返回一个文件对象，该对象提供了操作文件的方法。`write` 方法将数据写入文件。

```
>>> line1 = "This␣here's␣the␣wattle,\n"
>>> fout.write(line1)
24
```

返回值是被写入字符的个数。文件对象将跟踪自身的位置，所以下次你调用 write的时候，它会在文件末尾添加新的数据。

```
>>> line2 = "the␣emblem␣of␣our␣land.\n"
>>> fout.write(line2)
24
```

完成文件写入后，你应该关闭文件。

```
>>> fout.close()
```

如果你不关闭这个文件，程序结束时它才会关闭。

14.3 格式化运算符

write 的参数必须是字符串，所以如果想要在文件中写入其它值，我们需要先将它们转换为字符串。最简单的法是使用 str ：

```
>>> x = 52
>>> fout.write(str(x))
```

另一个方法是使用格式化运算符 (**format operator**)，即 %。作用于整数时，% 是取模运算符，而当第一个运算数是字符串时，% 则是格式化运算符。

第一个运算数是格式化字符串 (**format string**)，它包含一个或多个格式化序列 (**format sequence**)。格式化序列指定了第二个运算数是如何格式化的。运算结果是一个字符串。

例如，格式化序列 '%d' 意味着第二个运算数应该被格式化为一个十进制整数：

```
>>> camels = 42
>>> '%d' % camels
'42'
```

结果是字符串 '42' ，需要和整数值 42 区分开来。

一个格式化序列可以出现在字符串中的任何位置，所以可以将一个值嵌入到一个语句中：

```
>>> 'I␣have␣spotted␣%d␣camels.' % camels
'I␣have␣spotted␣42␣camels.'
```

如果字符串中有多个格式化序列，那么第二个参数必须是一个元组。每个格式化序列按顺序和元组中的元素对应。

下面的例子中使用 `'%d'` 来格式化一个整数，`'%g'` 来格式化一个浮点数，以及 `'%s'` 来格式化一个字符串：

```
>>> 'In␣%d␣years␣I␣have␣spotted␣%g␣%s.' % (3, 0.1, 'camels')
'In␣3␣years␣I␣have␣spotted␣0.1␣camels.'
```

元组中元素的个数必须等于字符串中格式化序列的个数。同时，元素的类型也必须符合对应的格式化序列：

```
>>> '%d␣%d␣%d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

在第一个例子中，元组中没有足够的元素；在第二个例子中，元素的类型错误。

你可以前往 此处 了解关于格式化运算符的更多信息。一个更为强大的方法是使用字符串的 `format` 方法，可以前往 此处 了解。

14.4 文件名和路径

文件以目录 (directory)（也称为 "文件夹 (folder)"）的形式组织起来。每个正在运行的程序都有一个 "当前目录 (current directory)" 作为大多数操作的默认目录。例如，当你打开一个文件来读取时，Python 会在当前目录下寻找这个文件。

`os` 模块提供了操作文件和目录的函数（"os" 代表 "operating system"）。`os.getcwd` 返回当前目录的名称：

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

类似 `'/home/dinsdale'` 这样的字符串指明一个文件或者目录，叫做路径 (path) 。

一个简单的文件名，如 `memo.txt` ，同样被看做是一个路径，只不过是相对路径 (relative path) ，因为它是相对于当前目录而言的。如果当前目录是 `/home/dinsdale` ，那么文件名 `memo.txt` 就代表 `/home/dinsdale/memo.txt` 。

一个以 `/` 开头的路径和当前目录无关，叫做绝对路径 (absolute path)。要获得一个文件的绝对路径，你可以使用 `os.path.abspath` ：

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path 还提供了其它函数来对文件名和路径进行操作。例如，os.path.exists 检查一个文件或者目录是否存在：

```
>>> os.path.exists('memo.txt')
True
```

如果存在，可以通过 os.path.isdir 检查它是否是一个目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

类似的，os.path.isfile 检查它是否是一个文件。

os.listdir 返回给定目录下的文件列表（以及其它目录）。

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

接下来演示下以上函数的使用。下面的例子 "遍历" 一个目录，打印所有文件的名字，并且针对其中所有的目录递归的调用自身。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join 接受一个目录和一个文件名，并把它们合并成一个完整的路径。

os 模块提供了一个叫做 walk 的函数，和我们上面写的类似，但是功能更加更富。作为练习，阅读文档并且使用 walk 打印出给定目录下的文件名和子目录。你可以在 此处 下载我的答案。

14.5 捕获异常

试图读写文件时，很多地方可能会发生错误。如果你试图打开一个不存在的文件夹，会得到一个输入输出错误 (IOError)：

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

如果你没有权限访问一个文件：

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

如果你试图打开一个目录来读取，你会得到：

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

为了避免这些错误，你可以使用类似 `os.path.exists` 和 `os.path.isfile` 的函数来检查，但这将会耗费大量的时间和代码去检查所有的可能性（从 “Errno 21” 这个错误信息来看，至少有 21 种可能出错的情况）。

更好的办法是在问题出现的时候才去处理，而这正是 `try` 语句做的事情。它的语法类似 `if...else` 语句：

```
try:
    fin = open('bad_file')
except:
    print('Something␣went␣wrong.')
```

Python 从 `try` 子句 (clause) 开始执行。如果一切正常，那么 `except` 子句将被跳过。如果发生异常，则跳出 `try` 子句，执行 `except` 子句。

使用 `try` 语句处理异常被称为是捕获 (catching) 异常。在本例中，`except` 子句打印出一个并非很有帮助的错误信息。一般来说，捕获异常后你可以选择是否解决这个问题，或者继续尝试运行，又或者至少优雅地结束程序。

14.6 数据库

index 数据库

数据库 是一个用来存储数据的文件。大多数的数据库采用类似字典的形式，即将键映射到值。数据库和字典的最大区别是，数据库是存储在硬盘上（或者其他永久存储中），所以即使程序结束，它们依然存在。

`dbm` 模块提供了一个创建和更新数据库文件的接口。举个例子，我接下来创建建一个包含图片文件标题的数据库。

打开数据库和打开其它文件的方法类似：

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

模式 `'c'` 代表如果数据库不存在则创建该数据库。这个操作返回的是一个数据库对象，可以像字典一样使用它（对于大多数操作）。

当你创建一个新项时，`dbm` 将更新数据库文件。

```
>>> db['cleese.png'] = 'Photo␣of␣John␣Cleese.'
```

当你访问某个项时，dbm 将读取文件：

```
>>> db['cleese.png']
b'Photo␣of␣John␣Cleese.'
```

返回的结果是一个字节对象 (bytes object)，这就是为什么结果以 b 开头。一个字节对象在很多方面都和一个字符串很像。但是当你深入了解 Python 时，它们之间的差别会变得很重要，但是目前我们可以忽略掉那些差别。

如果你对已有的键再次进行赋值，dbm 将把旧的值替换掉：

```
>>> db['cleese.png'] = 'Photo␣of␣John␣Cleese␣doing␣a␣silly␣walk.'
>>> db['cleese.png']
b'Photo␣of␣John␣Cleese␣doing␣a␣silly␣walk.'
```

一些字典方法，例如 keys 和 items，不适用于数据库对象，但是 for 循环依然适用：

```
for key in db:
    print(key, db[key])
```

与其它文件一样，当你完成操作后需要关闭文件：

```
>>> db.close()
```

14.7 序列化

dbm 的一个限制在于键和值必须是字符串或者字节。如果你尝试去用其它数据类型，你会得到一个错误。

pickle 模块可以解决这个问题。它能将几乎所有类型的对象转化为适合在数据库中存储的字符串，以及将那些字符串还原为原来的对象。

pickle.dumps 读取一个对象作为参数，并返回一个字符串表示（dumps 是 “dump string” 的缩写）：

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

这个格式对人类来说不是很直观，但是对 pickle 来说很容易去解释。pickle.loads (“load string”) 可以重建对象：

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

尽管新对象和旧对象有相同的值，但它们（一般来说）不是同一个对象：

```
>>> t1 == t2
True
>>> t1 is t2
False
```

换言之，序列化然后反序列化等效于复制一个对象。

你可以使用 `pickle` 将非字符串对象存储在数据库中。事实上，这个组合非常常用，已经被封装进了模块 `shelve` 中。

14.8 管道

大多数的操作系统提供了一个命令行的接口，也被称为 *shell* 。shell 通常提供浏览文件系统和启动程序的命令。例如，在 Unix 系统中你可以使用 `cd` 改变目录，使用 `ls` 显示一个目录的内容，通过输入 `firefox` （举例来说）来启动一个网页浏览器。

任何可以在 shell 中启动的程序，也可以在 Python 中通过使用管道对象 **(pipe object)** 来启动。一个管道代表着一个正在运行的程序。

例如，Unix 命令 `ls -l` 将以详细格式显示当前目录下的内容。你可以使用 `os.popen` 来启动 `ls`：

```
>>> cmd = 'ls␣-l'
>>> fp = os.popen(cmd)
```

实参是一个包含 shell 命令的字符串。返回值是一个行为类似已打开文件的对象。你可以使用 `readline` 来每次从 `ls` 进程的输出中读取一行，或者使用 `read` 来一次读取所有内容：

```
>>> res = fp.read()
```

当你完成操作后，像关闭一个文件一样关闭管道：

```
>>> stat = fp.close()
>>> print(stat)
None
```

返回值是 `ls` 进程的最终状态。`None` 表示正常结束（没有出现错误）。

例如，大多数 Unix 系统提供了一个叫做 `md5sum` 的命令，来读取一个文件的内容并计算出一个 “校验和 (checksum)”。你可以在 维基百科 中了解更多 MD5 的信息。不同的内容产生相同校验和的概率非常小（也就是说，在宇宙坍塌之前不会发生）。

你可以使用一个管道来从 Python 中运行 `md5sum` ，并得到计算结果：

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum␣' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 编写模块

任何包含 Python 代码的文件，都可以作为模块被导入。例如，假设你有包含以下代码的文件 `wc.py` ：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

如果你运行这个程序，它将读取自身并打印文件的行数，结果是 7 。你也可以这样导入模块：

```
>>> import wc
7
```

现在你有了一个模块对象 `wc` ：

```
>>> wc
<module 'wc' from 'wc.py'>
```

这个模块对象提供了 `linecount` 函数：

```
>>> wc.linecount('wc.py')
7
```

以上就是如何编写 Python 模块的方法。

这个例子中唯一的问题在于，当你导入模块后，它将自动运行最后面的测试代码。通常当导入一个模块时，它将定义一些新的函数，但是并不运行它们。

作为模块的程序通常写成以下结构：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` 是一个在程序开始时设置好的内建变量。如果程序以脚本的形式运行，`__name__` 的值为 `__main__`，这时其中的代码将被执行。否则当被作为模块导入时，其中的代码将被跳过。

我们做个练习，将例子输入到文件 `wc.py` 中，然后以脚本形式运行它。接着，打开 Python 解释器并导入 `wc`。当模块被导入后，`__name__` 的值是什么？

警示：如果你导入一个已经被导入了的模块，Python 将不会做任何事情。它并不会重新读取文件，即使文件的内容已经发生了改变。

如果你要重载一个模块，可以使用内建函数 `reload`，但它可能会出错。因此最安全的方法是重启解释器，然后重新导入模块。

14.10 调试

当你读写文件时，可能会遇到空白带来的问题。这些问题会很难调试，因为空格、制表符和换行符通常是看不见的：

```
>>> s = '1␣2\t␣3\n␣4'
>>> print(s)
1 2   3
 4
```

内建函数 `repr` 可以用来解决这个问题。它接受任意一个对象作为参数，然后返回一个该对象的字符串表示。对于空白符号，它将用反斜杠序列表示：

```
>>> print(repr(s))
'1␣2\t␣3\n␣4'
```

这个对于调试会很有用。

另一个你可能会遇到的问题是，不同的的系统使用不同的符号来表示一行的结束。有些系统使用换行符 `\n`，有的使用返回符号 `\r`，有些两者都使用。如果你在不同的系统中移动文件，这些差异会导致问题。

对大多数的系统，有一些转换不同格式文件的应用。你可以在 维基 中找到这些应用的信息（并阅读更多相关内容）。当然，你也可以自己编写一个转换程序。

14.11 术语表

持久性（persistent）：用于描述长期运行并至少将一部分自身的数据保存在永久存储中的程序。

格式化运算符（format operator）： 运算符 `%`。读取一个格式化字符串和一个元组，生成一个包含元组中元素的字符串，按照格式化字符串的要求格式化。

格式化字符串（format string）： 一个包含格式化序列的字符串，和格式化运算符一起使用。

格式化序列（format sequence）： 格式化字符串中的一个字符序列，例如 `%d` ，指定了一个值的格式。

文本文件（text file）： 保存在类似硬盘的永久存储设备上的字符序列。

目录（directory）： 一个有命名的文件集合，也叫做文件夹。

路径（path）： 一个指定一个文件的字符串。

相对路径（relative path）： 从当前目录开始的路径。

绝对路径（absolute path）： 从文件系统顶部开始的路径。

捕获（catch）： 为了防止程序因为异常而终止，使用 `try` 和 `except` 语句来捕捉异常。

数据库（database）： 一个内容结构类似字典的文件，将键映射至对应的值。

字节对象（bytes object）： 和字符串类的对象。

shell： 一个允许用户输入命令，并通过启用其它程序执行命令的程序。

管道对象（pipe object）： 一个代表某个正在运行的程序的对象，允许一个 Python 程序去运行命令并得到运行结果。

14.12 练习

Exercise 14.1. *编写一个叫做 `sed` 的函数，它的参数是一个模式字符串 (pattern string)，一个替换字符串和两个文件名。它应该读取第一个文件，并将内容写入到第二个文件（需要时创建它）。如果在文件的任何地方出现了模式字符串，就用替换字符串替换它。*

如果在打开、读取、写入或者关闭文件时出现了错误，你的程序应该捕获这个异常，打印一个错误信息，并退出。

参考答案 。
Exercise 14.2. *如果你从 此处 下载了练习 12.2 的答案，你会看到答案中创建了一个字典，将从一个由排序后的字母组成的字符串映射到一个可以由这些字母拼成的单词组成的列表。例如，`'opst'` 映射到列表 `['opts', 'post', 'pots', 'spot', 'stop', 'tops']` 。*

编写一个模块，导入 `anagram_sets` 并提供两个新函数：函数 `store_anagrams` 在将 anagram 字典保存至 `shelf`中；`read_anagrams` 查找一个单词，并返回它的 anagrams 列表。
Exercise 14.3. *在一个很大的 MP3 文件集合中，或许会有同一首歌的不同拷贝，它们存放在不同的目录下或者有不同的名字。这个练习的目的是检索出这些拷贝。*

1. *编写一个程序，搜索一个目录和它的所有子目录，并返回一个列表，列表中包含所有的有给定后缀（例如.mp3）的文件的完整路径。提示：`os.path` 提供了一些可以操作文件和路径名的函数。*

2. 为了识别出重复的文件，你可以使用 `md5sum` 来计算每个文件的“校验和”。如果两个文件的校验和相同，它们很可能有相同的内容。

3. 你可以使用 *Unix* 命令 `diff` 再确认一下。

参考答案

第十五章　类和对象

目前你已经知道如何使用函数来组织你的代码，同时用内置的类型来管理数据。下一步我们将学习 “面向对象编程”，即使用程序员定义的类来组织代码和数据。面向对象编程是一个很大的话题，讲完需要一些章节。

本章的示例代码可以在此处 获取；练习题的答案可以在此处 获取。

15.1　程序员自定义类型

我们已经使用过了许多 **Python** 的内置类型；现在我们要定义一个新类型。举个例子，我们来创建一个叫做 `Point` 的类型，代表二维空间中的一个点。

在数学记法中，点通常被写成在两个小括号中用一个逗号分隔坐标的形式。例如 $(0,0)$ 代表原点，(x,y) 代表原点向右 x 个单位，向上 y 个单位的点。

在 **Python** 中，有几种表示点的方法：

- 我们可以将坐标存储在两个独立的变量，**x** 和 **y** 中。
- 我们可以将坐标作为一个列表或者元组的元素存储。
- 我们可以创建一个新类型将点表示为对象。

创建一个新类型比其他方法更复杂，但是它的优势一会儿会显现出来。

程序员自定义类型 (**A programmer-defined type**) 也被称作类 (**class**)。像这样定义一个对象：

```
class Point:
    """Represents␣a␣point␣in␣2−D␣space."""
```

头部语句表明新类的名称是 `Point` 。主体部分是文档字符串，用来解释这个类的用途。你可以在一个类的定义中定义变量和函数，稍后会讨论这个。

定义一个叫做 `Point` 的类将创建了一个类对象 (**class object**)。

```
>>> Point
<class '__main__.Point'>
```

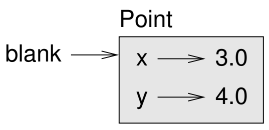

图 15.1: Object diagram.

由于 `Point` 是定义在顶层的，所以它的 “全名” 是 `__main__.Point` 。

类对象就像是一个用来创建对象的工厂。要创建一个点，你可以像调用函数那样调用 `Point` 。

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

返回值是一个 `Point` 对象的引用，我们将它赋值给 `blank` 。

创建一个新对象的过程叫做实例化 **(instantiation)** ，这个新对象叫做这个类的一个实例 **(instance)**。

当你试图打印一个实例，Python 会告诉你它属于哪个类，以及它在内存中的存储地址（前缀 `0x` 代表紧跟后面的数是以十六进制表示的）。

每一个对象都是某种类的实例，所以 “对象” 和 “实例” 可以互换。但是在这章我用 “实例” 来表示我在讨论程序员自定义类型。

15.2 属性

你可以使用点标记法向一个实例进行赋值操作：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

这个语法类似于从一个模块中使用变量的语法，比如 `math.pi` 和 `string.whitespace` 。不过在这个例子中，我们是给一个类中已命名的元素赋值。这类元素叫做属性 **(attributes)**。

作为名词的时候，“属性” 的英文 “AT-trib-ute” 的重音在第一个音节上，作为动词的时候，“a-TRIB-ute” 重音在第二个音节上。

下面这张图展示了这些赋值操作的结果。说明一个对象及其属性的状态图叫做对象图 **(object diagram)**；见图 15.1。

变量 `blank` 引用了一个 `Point` 类，这个类拥有了两个属性。每个属性都引用了一个浮点数。

你可以使用相同的语法读取一个属性的值：

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表达式 blank.x 的意思是，“前往 blank 所引用的对象并且获取 x 的值”。在这个例子中，我们将获取到的值赋值给了一个叫做 x 的变量。变量 x 和属性 x 并不会冲突。

你可以在任何表达式中使用点标记法。例如：

```
>>> '(%g,␣%g)' % (blank.x, blank.y)
'(3.0,␣4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

你可以将一个实例作为参数传递。例如：

```
def print_point(p):
    print('(%g,␣%g)' % (p.x, p.y))
```

print_point 接受一个点作为参数，打印出其在数学中的表示方法。调用它的时候，你可以将 blank 作为参数传递：

```
>>> print_point(blank)
(3.0, 4.0)
```

在这个函数内部，p 是 blank 的别名，所以，如果函数修改了 p ，blank 也会随之改变。

我们做个联系，编写一个叫做 distance_between_points 的函数，它接受两个 Point 作为参数，然后返回这两个点之间的距离。

15.3 矩形

有时候，一个对象该拥有哪些属性是显而易见的，但有时候你需要好好考虑一番。比如，你需要设计一个代表矩形的类。为了描述一个矩形的位置和大小，你需要设计哪些属性呢？角度是可以忽略的；为了使事情更简单，我们假设矩形是水平或者竖直的。

至少有两种可能的设计：

- 你可以指定矩形的一个角（或是中心）、宽度以及长度。
- 你可以指定对角线上的两个角。

这个时候还不能够说明哪个方法优于哪个方法。我们先来实现前者。

下面是类的定义：

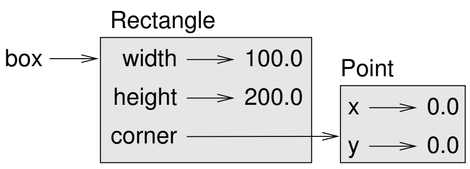

图 15.2: Object diagram.

```
class Rectangle:
    """Represents␣a␣rectangle.

␣␣␣␣attributes:␣width,␣height,␣corner.
␣␣␣␣"""
```

文档字符串中列出了属性：`width` 和 `height` 是数字；`corner` 是一个 `Point` 对象，代表左下角的那个点。

为了描述一个矩形，你需要实例化一个 `Rectangle` 对象，并且为它的属性赋值：

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表达式 `box.corner.x` 的意思是，“前往 `box` 所引用的对象，找到叫做 `corner` 的属性；然后前往 `corner` 所引用的对象，找到叫做 `x` 的属性。”

图 15.2 展示了这个对象的状态。一个对象作为另一个对象的属性叫做嵌套 **(embedded)**。

15.4　实例作为返回值

函数可以返回实例。例如，`find_center` 接受一个 `Rectangle` 作为参数，返回一个 `Point`，代表了这个 `Rectangle` 的中心坐标：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

下面这个例子将 `box` 作为参数传递，然后将返回的 `Point` 赋值给 `center`：

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 对象是可变的

你可以通过给一个对象的属性赋值来改变这个对象的状态。例如，要改变一个矩形的大小而不改变它的位置，你可以修改 width 和 height 的值：

```
box.width = box.width + 50
box.height = box.height + 100
```

你也可以编写函数来修改对象。例如，grow_rectangle 接受一个Rectangle 对象和两个数字，dwidth 和dheight，并将其加到矩形的宽度和高度上：

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

下面的例子展示了具体效果：

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

在函数内部，rect 是 box 的一个别名，所以如果函数修改了 rect，则 box 也随之改变。

我们做个练习，编写一个叫做 move_rectangle 的函数，接受一个 Rectangle 以及两个数字 dx 和 dy。它把 corner 的 x 坐标加上 dx，把 corner 的 y 坐标加上 dy，从而改变矩形的位置。

15.6 复制

别名会降低程序的可读性，因为一个地方的变动可能对另一个地方造成预料之外的影响。跟踪所有引用同一个对象的变量是非常困难的。

通常用复制对象的方法取代为对象起别名。copy 模块拥有一个叫做 copy 的函数，可以复制任何对象：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 和 p2 拥有相同的数据，但是它们并不是同一个 Point 对象。

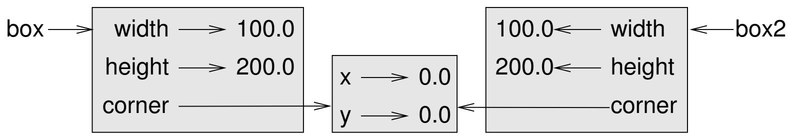

图 15.3: Object diagram.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

正如我们预期的，`is` 运算符显示了 `p1` 和 `p2` 并非同一个对象。不过你可能会认为 `==` 运算的结果应该是 `True` ，因为这两个点的数据是相同的。然而结果并不如你想象的那样，`==` 运算符的默认行为和 `is` 运算符相同；它检查对象的标识 **(identity)** 是否相同，而非对象的值是否相同。因为 Python 并不知道什么样可以被认为相同。至少目前不知道。

如果你使用 `copy.copy` 来复制一个 `Rectangle` ，你会发现它仅仅复制了 `Rectangle` 对象，但没有复制嵌套的 `Point` 对象。

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

图 15.3 展示了相应的对象图。这个操作叫做浅复制 **(shallow copy)** ，因为它仅复制了对象以及其包含的引用，但未复制嵌套的对象。

对大多数应用来说，这并非是你想要的结果。在这个例子中，对其中一个 `Rectangle` 对象调用 `grow_rectangle`并不会影响到另外一个，然而当对任何一个 `Rectangle` 对象调用 `move_rectangle`的时候，两者都会被影响！这个行为很容易带来疑惑和错误。

幸运的是，`copy` 模块拥有一个叫做 `deepcopy` 的方法，它不仅可以复制一个对象，还可以复制这个对象所引用的对象，甚至可以复制这个对象所引用的对象 所引用的对象，等等。没错！这个操作叫做深复制 **(deep copy)** 。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

我们做个练习，编写另一个版本的 `move_rectangle` ，函数创建并返回一个新的 `Rectangle` 对象而非修改原先的那个。

15.7　调试

当你开始学习对象的时候，你可能会遇到一些新的异常。如果你访问一个不存在的属性，你会得到 `Attributeerror` 的错误提示：

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

如果你不确定一个对象的类型，你可以询问：

```
>>> type(p)
<class '__main__.Point'>
```

你也可以用 `isinstance` 来检查某个对象是不是某个类的实例。

```
>>> isinstance(p, Point)
True
```

如果你不确定一个对象是否拥有某个属性，你可以使用内置函数 `hasattr` 检查：

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一个参数可以是任何对象；第二个参数是一个字符串，代表了某个属性的名字。

你也可以使用 `try` 语句来检查某个对象是不是有你需要的属性：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

这个方法可以让你更容易编写出可以适应多种数据结构的函数。你可以在 17.9 节查看更多内容。

15.8　术语表

类 (class): 一种程序员自定义的类型。类定义创建了一个新的类对象。

类对象 (class object): 包含程序员自定义类型的细节信息的对象。类对象可以被用于创建该类型的实例。

实例 (instance): 属于某个类的对象。

实例化 (instantiate): 创建新的对象。

属性 (attribute): 和某个对象相关联的有命名的值。

嵌套对象 (embedded object): 作为另一个对象的属性存储的对象。

浅复制 (shallow copy): 在复制对象内容的时候，只包含嵌套对象的引用，通过 copy 模块的 copy 函数实现。

深复制 (deep copy): 在复制对象内容的时候，既复制对象属性，也复制所有嵌套对象及其中的所有嵌套对象，由 copy 模块的 deepcopy 函数实现。

对象图 (object diagram): 展示对象及其属性和属性值的图。

15.9 练习

Exercise 15.1. 定义一个叫做Circle 类，类的属性是圆心 (center) 和半径 (radius), 其中，圆心 (center) 是一个 Point 类，而半径 (radius) 是一个数字。

实例化一个圆心 (center) 为 $(150, 100)$ ，半径 (radius) 为 75 的 *Circle* 对象。

编写一个名称为point_in_circle 的函数，该函数可以接受一个圆类 (Circle) 对象和点类 (Point) 对象，然后判断该点是否在圆内。在圆内则返回True 。

编写一个名称为rect_in_circle 的函数，该函数接受一个圆类 (Circle) 对象和矩形 (Rectangle) 对象，如果该矩形是否完全在圆内或者在圆上则返回True 。

编写一个名为rect_circle_overlap 函数，该函数接受一个圆类对象和一个矩形类对象，如果矩形有任意一个角落在圆内则返回True 。或者写一个更具有挑战性的版本，如果该矩形有任何部分落在圆内返回True 。

参考答案

Exercise 15.2. 编写一个名为draw_rect 的函数，该函数接受一个Turtle 对象和一个Rectangle 对象，使用Turtle 画出该矩形。参考 四 章中使用Turtle 的示例。

编写一个名为 draw_circle 的函数，该函数接受一个Turtle 对象和Circle 对象，并画出该圆。

参考答案

第十六章　类和函数

现在我们已经知道如何去定义一个新的类型，下一步就是编写以自定义对象为参数的函数，并返回自定义对象作为结果。在本章中，我还将介绍“函数式编程风格”和两种新的编程开发方案。

本章的代码示例可以从 这里 下载。练习的答案可以从 这里 下载。

16.1　时间

再举一例程序员自定义的类型，我们定义一个叫 `Time` 的类，用于记录时间。这个类将如下定义：

```
class Time:
    """Represents␣the␣time␣of␣day.

␣␣␣␣attributes:␣hour,␣minute,␣second
␣␣␣␣"""
```

我们可以创建一个新的 `Time` 类对象，并且给它的属性 `hour`，`minutes` 和 `seconds` 赋值：

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

`Time` 对象的状态图类似于图 16.1

我们做个练习，编写一个叫做 `print_time` 的函数，接收一个 `Time` 对象并用 `时:分:秒` 的格式打印它。提示：格式化序列 `%.2d` 可以至少两位数的形式打印一个整数，如果不足则在前面补 0。

编写一个叫做 `is_after` 的布尔函数，接收两个 `Time` 对象，`t1` 和 `t2`，若 `t1` 的时间在 `t2` 之后，则返回 `True`，否则返回 `False`。挑战：不要使用 `if` 语句。

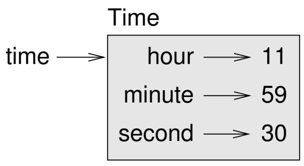

图 16.1: Object diagram.

16.2 纯函数

下面几节中，我们将编写两个用来增加时间值的函数。它们展示了两种不同的函数：纯函数 (**pure functions**) 和修改器 (**modifiers**)。它们也展示了我所称的原型和补丁 (**prototype and patch**) 的开发方案。这是一种处理复杂问题的方法，从简单的原型开始，逐步解决复杂情况。

下面是一个简单的 `add_time` 原型：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

这个函数创建了一个新的 `Time` 对象，初始化了对象的属性，并返回了这个对象的引用。我们把这个函数称为纯函数 (**pure function**)，因为它除了返回一个值以外，并不修改作为参数传入的任何对象，也没有产生如显示一个值或者获取用户输入的影响。

为了测试这个函数，我将创建两个 `Time` 对象：`start` 用于存放一个电影（如 **Monty Python and the Holy Grail**）的开始时间，`duration` 用于存放电影的放映时长，这里时长定为 1 小时 35 分钟。

`add_time` 将计算电影何时结束。

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

这个结果 `10:80:00` 可能不是你所希望得到的。问题在于这个函数并没有处理好秒数和分钟数相加超过 60 的情况。当发生这种情况时，我们要把多余的秒数放进分钟栏，或者把多余的分钟加进小时栏。

下面是一个改进的版本：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

这个函数虽然正确，但是它开始变得臃肿。我们会在后面看到一个较短的版本。

16.3 Modifiers

16.4 修改器

有时候用函数修改作为参数传入的对象是很有用的。在这种情况下，这种改变对调用者来说是可见的。这种方式工作的函数称为修改器 (**modifiers**)。

函数 `increment` 给一个 `Time` 对象增加指定的秒数，可以很自然地用修改器来编写。下面是一个原型：

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

第一行进行基础操作；其余部分的处理则是我们之前看到的特殊情况。

这个函数正确吗？如果 `seconds` 比 60 大很多会发生什么？

在那种情况下，只进位一次是不够的；我们要重复执行直到 `seconds` 小于 60。一种方法是用 `while` 语句代替 `if` 语句。这样能够让函数正确，但是并不是很高效。

任何能够用修改器实现的函数同样能够用纯函数实现。事实上，一些编程语言只允许用纯函数。一些证据表明用纯函数实现的程序比用修改器实现的函数开发更快、更不易出错。但是有时候修改器是很方便的，而函数式程序效率反而不高。

通常来说，我推荐只要是合理的情况下，都使用纯函数方式编写，只在有完全令人信服的原因下采用修改器。这种方法可以称为函数式编程风格 (**functional programming style**)。

我们做个练习，编写一个纯函数版本的 `increment` ，创建并返回一个 `Time` 对象，而不是修改参数。

16.5 原型 *v.s.* 方案

我刚才展示的开发方案叫做原型和补丁 (**protptype and patch**) 。针对每个函数，我编写了一个可以进行基本运算的原型并对其测试，逐步修正错误。

这种方法在你对问题没有深入理解时特别有效。但增量修正可能导致代码过度复杂，因为需要处理许多特殊情况。也并不可靠，因为很难知道你是否已经找到了所有的错误。

另一种方法叫做设计开发 (**designed development**)。对问题有高层次的理解能够使开发变得更容易。这给我们的启示是，`Time` 对象本质上是一个基于六十进制的三位数（详见 此文 。)！属性 `second` 是 "个位"，属性 `minute` 是 "六十位"，属性 `hour` 是 "360 位数"。

当我们编写 `add_time` 和 `increment` 时，其实是在基于六十进制累加，所以我们需要把一位进位到下一位。

这个观察意味着我们可以用另一种方法去解决整个问题——我们可以把 `Time` 对象转换为整数，并利用计算机知道如何进行整数运算的这个事实。

下面是一个把 `Time` 对象转成整数的函数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

下面则是一个把整数转换为 `Time` 对象（回忆一下 `divmod` 是用第一个参数除以第二个参数并以元组的形式返回商和余数）。

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

你可能需要思考一下，并运行一些测试，以此来说服自己这些函数是正确的。一种测试方法是对很多的 `x` 检查 `time_to_int(int_to_time(x)) == x` 是否正确。这是一致性检查的例子。

一旦你确信它们是正确的，你就能使用它们重写 `add_time` ：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

这个版本比先前的要更短,更容易校验。我们再做个练习,使用 time_to_int 和 int_to_time 重写 increment 函数。

从某个方面来说，六十进制和十进制相互转换比处理时间更难些。进制转换更加抽象；我们解决时间值的想法是更好的。

但如果我们意识到把时间当作六十进制，并预先做好编写转换函数（time_to_int 和 int_to_time）的准备，我们就能获得一个更短、更易读、更可靠的程序。

这让我们日后更加容易添加其它功能。例如，试想将两个 Time 对象相减来获得它们之间的时间间隔。最简单的方法是使用借位来实现减法。使用转换函数则更容易，也更容易正确。

讽刺的是，有时候把一个问题变得更难（或更加普遍）反而能让它更加简单（因为会有更少的特殊情况和更少出错的机会）。

16.6 调试

如果 minute 和 second 的值介于 0 和 60 之间（包括 0 但不包括 60），并且 hour 是正值，那么这个 Time 对象就是合法的。hour 和 minute 应该是整数值，但我们可能也允许 second 有小数部分。

这样的要求称为不变式 (**invariants**)。因为它们应当总是为真。换句话说，如果它们不为真，肯定是某些地方出错了。

编写代码来检查不变式能够帮助检测错误并找到出错的原因。例如，你可能会写一个 valid_time 这样的函数，接受一个 Time 对象，并在违反不变式的条件下返回 False：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

在每个函数的开头，你可以检查参数，确认它们是否合法：

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid␣Time␣object␣in␣add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

或者你可以使用 **assert 语句 **，检查一个给定的不变式并在失败的情况下抛出异常：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assert 语句非常有用，因为它们区分了处理普通条件的代码和检查错误的代码。

16.7　术语表

原型和补丁 (prototype and patch): 一种开发方案，编写一个程序的初稿，测试，发现错误时修正它们。

设计开发 (designed development): 一种开发方案，需要对问题有更高层次的理解，比增量开发或原型开发更有计划性。

纯函数 (pure function): 一种不修改任何作为参数传入的对象的函数。大部分纯函数是有返回值的（fruitful）。

修改器 (modifier): 一种修改一个或多个作为参数传入的对象的函数。大部分修改器没有返回值；即返回 `None`

函数式编程风格 (functional programming style): 一种程序设计风格，大部分函数为纯函数。

不变式 (invariant): 在程序执行过程中总是为真的条件。

(assert statement): 一种检查条件是否满足并在失败的情况下抛出异常的语句。

16.8　练习

本章的代码示例可以在 此处下载；练习的答案可以在 此处下载。
Exercise 16.1. *编写一个叫做 `mul_time` 的函数，接收一个 `Time` 对象和一个数，并返回一个新的 `Time` 对象，包含原始时间和数的乘积。*

然后使用 `mul_time` 编写一个函数，接受一个表示比赛完赛时间的 `Time` 对象以及一个表示距离的数字，并返回一个用于表示平均配速（每英里所需时间）的 `Time` 对象。
Exercise 16.2. *`datetime` 模块提供的 `time` 对象，和本章的 `Time` 对象类似，但前者提供了更丰富的方法和操作符。参考阅读 相关文档。*

1. *使用 `datetime` 模块来编写一个程序，获取当前日期并打印当天是周几。*

2. *编写一个程序，接受一个生日作为输入，并打印用户的年龄以及距离下个生日所需要的天数、小时数、分钟数和秒数。*

3. *对于两个不在同一天出生的人来说，总有一天，一个人的出生天数是另一个人的两倍。我们把这一天称为"双倍日"。编写一个程序，接受两个不同的出生日期，并计算他们的"双倍日"。*

4. *再增加点挑战，编写一个更通用的版本，用于计算一个人出生天数是另一个人 n 倍的日子。*

参考答案

第十七章 Classes and methods

Although we are using some of Python's object-oriented features, the programs from the last two chapters are not really object-oriented because they don't represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

Code examples from this chapter are available from `http://thinkpython2.com/code/Time2.py`, and solutions to the exercises are in `http://thinkpython2.com/code/Point2_soln.py`.

17.1 Object-oriented features

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

For example, the `Time` class defined in Chapter 十六 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes in Chapter 十五 correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in `Time1.py` there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for methods; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

17.2 Printing objects

In Chapter 十六, we defined a class named `Time` and in Section 16.1, you wrote a function named `print_time`:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the subject. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, "Hey `print_time`! Here's an object for you to print."
- In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says "Hey `start`! Please print yourself."

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite `time_to_int` (from Section 16.5) as a method. You might be tempted to rewrite `int_to_time` as a method, too, but that doesn't really make sense because there would be no object to invoke it on.

17.3 Another example

Here's a version of `increment` (from Section 16.4) rewritten as a method:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a positional argument is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

`parrot` and `cage` are positional, and `dead` is a keyword argument.

17.4 A more complicated example

Rewriting `is_after` (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter `self` and the second parameter `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

17.5 The init method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An init method for the `Time` class might look like this:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
        self.hour = hour
```

stores the value of the parameter `hour` as an attribute of `self`.

The parameters are optional, so if you call `Time` with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides `hour`:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override `hour` and `minute`.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

As an exercise, write an init method for the `Point` class that takes `x` and `y` as optional parameters and assigns them to the corresponding attributes.

17.6 The `__str__` method

`__str__` is a special method, like `__init__`, that is supposed to return a string representation of an object.

For example, here is a `str` method for Time objects:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

As an exercise, write a str method for the Point class. Create a Point object and print it.

17.7 Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

Here is what the definition might look like:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like __add__. For more details, see http://docs.python.org/3/reference/datamodel.html#specialnames.

As an exercise, write an add method for the Point class.

17.8 Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function `isinstance` takes a value and a class object, and returns `True` if the value is an instance of the class.

If `other` is a Time object, `__add__` invokes `add_time`. Otherwise it assumes that the parameter is a number and invokes `increment`. This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method `__radd__`, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

As an exercise, write an add method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

17.9 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used histogram to count the number of times each letter appears in a word.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an add method, they work with sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

17.10 Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 15.7).

Another way to access attributes is the built-in function `vars`, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

17.11 Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include `time_to_int`, `is_after`, and `add_time`.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a `Time` object are `hour`, `minute`, and `second`.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like `is_after`, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

17.12 Glossary

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

operator overloading: Changing the behavior of an operator like + so it works with a programmer-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

information hiding: The principle that the interface provided by an object should not depend on its implementation, in particular the representation of its attributes.

17.13 Exercises

Exercise 17.1. *Download the code from this chapter from* *http://thinkpython2.com/code/Time2.py*. *Change the attributes of* `Time` *to be a single integer representing seconds since midnight. Then modify the methods (and the function* `int_to_time`*) to work with the new implementation. You should not have to modify the test code in* `main`*. When you are done, the output should be the same as before. Solution:* *http://thinkpython2.com/code/Time2_soln.py*.

Exercise 17.2. *This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named* `Kangaroo` *with the following methods:*

1. *An* `__init__` *method that initializes an attribute named* `pouch_contents` *to an empty list.*
2. *A method named* `put_in_pouch` *that takes an object of any type and adds it to* `pouch_contents`*.*
3. *A* `__str__` *method that returns a string representation of the Kangaroo object and the contents of the pouch.*

Test your code by creating two `Kangaroo` *objects, assigning them to variables named* `kanga` *and* `roo`*, and then adding* `roo` *to the contents of* `kanga`*'s pouch.*

Download `http://thinkpython2.com/code/BadKangaroo.py`*. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.*

If you get stuck, you can download `http://thinkpython2.com/code/GoodKangaroo.py`*, which explains the problem and demonstrates a solution.*

第十八章　Inheritance

The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

If you don't play poker, you can read about it at `http://en.wikipedia.org/wiki/Poker`, but you don't have to; I'll tell you what you need to know for the exercises.

Code examples from this chapter are available from `http://thinkpython2.com/code/Card.py`.

18.1　Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like `'Spade'` for suits and `'Queen'` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to encode the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

I am using the $\mapsto$ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for `Card` looks like this:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

18.2 Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to class attributes:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called instance attributes because they are associated with a particular instance.

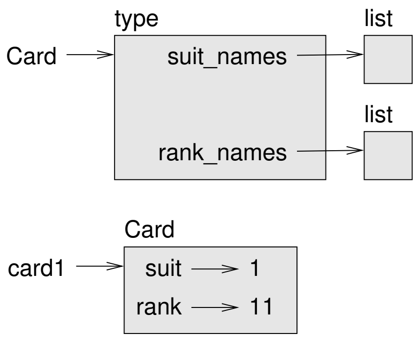

图 18.1: Object diagram.

Both kinds of attribute are accessed using dot notation. For example, in `__str__`, `self` is a Card object, and `self.rank` is its rank. Similarly, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

Every card has its own `suit` and `rank`, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means "use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string."

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string `'2'`, and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Figure 18.1 is a diagram of the `Card` class object and one Card instance. `Card` is a class object; its type is `type`. `card1` is an instance of `Card`, so its type is `Card`. To save space, I didn't draw the contents of `suit_names` and `rank_names`.

18.3 Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for "less than".

`__lt__` takes two parameters, `self` and `other`, and `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
# inside class Card:

    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

As an exercise, write an `__lt__` method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

18.4 Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for `Deck`. The init method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to `self.cards`.

18.5 Printing the deck

Here is a `__str__` method for `Deck`:

```
#inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

18.6 Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that:

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Since pop removes the *last* card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method append:

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a veneer. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case add_card is a "thin" method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named shuffle using the function shuffle from the random module:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import random.

As an exercise, write a Deck method named sort that uses the list method sort to sort the cards in a Deck. sort uses the __lt__ method we defined to determine the order.

18.7 Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the parent and the new class is called the child.

In this example, `Hand` inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize `cards` with an empty list.

If we provide an init method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

When you create a Hand, Python invokes this init method, not the one in `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

The other methods are inherited from `Deck`, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
#inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a `Deck`.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

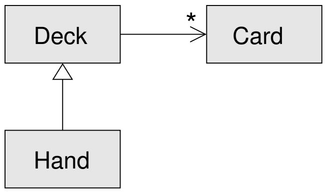

图 18.2: Class diagram.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8 Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, "a Rectangle has a Point."
- One class might inherit from another. This relationship is called IS-A, as in, "a Hand is a kind of a Deck."
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a dependency.

A class diagram is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between `Card`, `Deck` and `Hand`.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like `52`, a range, like `5..7` or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a *list* of Cards, but built-in types like list and dict are usually not included in class diagrams.

18.9 Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like `shuffle`, you might get the one defined in `Deck`, but if any of the subclasses override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Here's an example:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

So the `shuffle` method for this Hand is the one in `Deck`.

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order", which is the sequence of classes Python searches to "resolve" a method name.

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.

If you violate this rule, which is called the "Liskov substitution principle", your code will collapse like (sorry) a house of cards.

18.10 Data encapsulation

The previous chapters demonstrate a development plan we might call "object-oriented design". We identified objects we needed—like `Point`, `Rectangle` and `Time`—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by data encapsulation.

Markov analysis, from Section 13.8, provides a good example. If you download my code from http://thinkpython2.com/code/markov.py, you'll see that it uses two global variables—`suffix_map` and `prefix`—that are read and written from several functions.

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here's `process_word`:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from `http://thinkpython2.com/code/markov.py`, and follow the steps described above to encapsulate the global variables as attributes of a new class called `Markov`. Solution: `http://thinkpython2.com/code/Markov.py` (note the capital M).

18.11 Glossary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

instance attribute: An attribute associated with an instance of a class.

veneer: A method or function that provides a different interface to another function without doing much computation.

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass".

IS-A relationship: A relationship between a child class and its parent class.

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

class diagram: A diagram that shows the classes in a program and the relationships between them.

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.

18.12 Exercises

Exercise 18.1. *For the following program, draw a UML class diagram that shows these classes and the relationships among them.*

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Exercise 18.2. *Write a Deck method called* `deal_hands` *that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.*

Exercise 18.3. *The following are the possible hands in poker, in increasing order of value and decreasing order of probability:*

pair: *two cards with the same rank*

two pair: *two pairs of cards with the same rank*

three of a kind: *three cards with the same rank*

straight: *five cards with ranks in sequence (aces can be high or low, so* `Ace-2-3-4-5` *is a straight and so is* `10-Jack-Queen-King-Ace`*, but* `Queen-King-Ace-2-3` *is not.)*

flush: *five cards with the same suit*

full house: *three cards with one rank, two cards with another*

four of a kind: *four cards with the same rank*

straight flush: *five cards in sequence (as defined above) and with the same suit*

The goal of these exercises is to estimate the probability of drawing these various hands.

1. *Download the following files from* *http://thinkpython2.com/code*:

   `Card.py` *: A complete version of the* `Card`, `Deck` *and* `Hand` *classes in this chapter.*

   `PokerHand.py` *: An incomplete implementation of a class that represents a poker hand, and some code that tests it.*

2. *If you run* `PokerHand.py`, *it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.*

3. *Add methods to* `PokerHand.py` *named* `has_pair`, `has_twopair`, *etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for "hands" that contain any number of cards (although 5 and 7 are the most common sizes).*

4. *Write a method named* `classify` *that figures out the highest-value classification for a hand and sets the* `label` *attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled "flush".*

5. *When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in* `PokerHand.py` *that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.*

6. *Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at* *http://en.wikipedia.org/wiki/Hand_rankings*.

Solution: *http://thinkpython2.com/code/PokerHandSoln.py*.

第十九章　The Goodies

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

19.1 Conditional expressions

We saw conditional statements in Section 5.4. Conditional statements are often used to choose one of two values; for example:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

This statement checks whether `x` is positive. If so, it computes `math.log`. If not, `math.log` would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a conditional expression:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: "`y` gets log-`x` if `x` is greater than 0; otherwise it gets NaN".

Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of `factorial`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We can rewrite it like this:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from GoodKangaroo (see Exercise 17.2):

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

We can rewrite this one like this:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

19.2 List comprehensions

In Section 10.7 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method `capitalize` to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a list comprehension:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the `for` clause indicates what sequence we are traversing.

The syntax of a list comprehension is a little awkward because the loop variable, `s` in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of `t` that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

19.3 Generator expressions

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function `next` gets the next value from the generator:

```
>>> next(g)
0
>>> next(g)
1
```

When you get to the end of the sequence, `next` raises a StopIteration exception. You can also use a `for` loop to iterate through the values:

```
>>> for val in g:
...     print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the `for` loop picks up where `next` left off. Once the generator is exhausted, it continues to raise `StopException`:

```
>>> next(g)
StopIteration
```

Generator expressions are often used with functions like `sum`, `max`, and `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any and all

Python provides a built-in function, any, that takes a sequence of boolean values and returns True if any of the values are True. It works on lists:

```
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the in operator. But we could use any to rewrite some of the search functions we wrote in Section 9.3. For example, we could write avoids like this:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, "word avoids forbidden if there are not any forbidden letters in word."

Using any with a generator expression is efficient because it stops immediately if it finds a True value, so it doesn't have to evaluate the whole sequence.

Python provides another built-in function, all, that returns True if every element of the sequence is True. As an exercise, use all to re-write uses_all from Section 9.3.

19.5 Sets

In Section 13.6 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes d1, which contains the words from the document as keys, and d2, which contains the list of words. It returns a dictionary that contains the keys from d1 that are not in d2.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

In all of these dictionaries, the values are None because we never use them. As a result, we waste some storage space.

Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

For example, set subtraction is available as a method called difference or as an operator, -. So we can rewrite subtract like this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to `has_duplicates`, from Exercise 10.7, that uses a dictionary:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns `True`.

Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in `t` appears more than once, the set will be smaller than `t`. If there are no duplicates, the set will be the same size as `t`.

We can also use sets to do some of the exercises in Chapter 九. For example, here's a version of `uses_only` with a loop:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` checks whether all letters in `word` are in `available`. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The <= operator checks whether one set is a subset or another, including the possibility that they are equal, which is true if all the letters in `word` appear in `available`.

As an exercise, rewrite `avoids` using sets.

19.6 Counters

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a multiset, a Counter is a natural way to represent a multiset.

Counter is defined in a standard module called `collections`, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

```
>>> count['d']
0
```

We can use Counters to rewrite `is_anagram` from Exercise 10.6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent.

Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, `most_common`, which returns a list of value-frequency pairs, sorted from most common to least:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict

The `collections` module also provides `defaultdict`, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a factory. The built-in functions that create lists, sets, and other types can be used as factories:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notice that the argument is `list`, which is a class object, not `list()`, which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

```
>>> t = d['new key']
>>> t
[]
```

The new list, which we're calling `t`, is also added to the dictionary. So if we modify `t`, the change appears in `d`:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using `defaultdict`. In my solution to Exercise 12.2, which you can get from `http://thinkpython2.com/code/anagram_sets.py`, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, `'opst'` maps to the list `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Here's the original code:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

This can be simplified using `setdefault`, which you might have used in Exercise 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

We can avoid this problem and simplify the code using a `defaultdict`:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

My solution to Exercise 18.3, which you can download from http://thinkpython2.com/code/PokerHandSoln.py, uses setdefault in the function has_straightflush. This solution has the drawback of creating a Hand object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a defaultdict.

19.8 Named tuples

Many simple objects are basically collections of related values. For example, the Point object defined in Chapter 十五 contains two numbers, x and y. When you define a class like this, you usually start with an init method and a str method:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from namedtuple is a class object:

```
>>> Point
<class '__main__.Point'>
```

Point automatically provides methods like __init__ and __str__ so you don't have to write them.

To create a Point object, you use the Point class as a function:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes.

You can access the elements of the named tuple by name:

```
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

19.9 Gathering keyword args

In Section 12.4, we saw how to write a function that gathers its arguments into a tuple:

```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but `kwargs` is a common choice. The result is a dictionary that maps keywords to values:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat `d` as a single positional argument, so it would assign `d` to `x` and complain because there's nothing to assign to `y`:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

19.10 Glossary

conditional expression: An expression that has one of two values, depending on a condition.

list comprehension: An expression with a `for` loop in square brackets that yields a new list.

generator expression: An expression with a `for` loop in parentheses that yields a generator object.

multiset: A mathematical entity that represents a mapping between the elements of a set and the number of times they appear.

factory: A function, usually passed as a parameter, used to create objects.

19.11 Exercises

Exercise 19.1. *The following is a function computes the binomial coefficient recursively.*

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Rewrite the body of the function using nested conditional expressions.

One note: this function is not very efficient because it ends up computing the same values over and over. You could make it more efficient by memoizing (see Section 11.6). But you will find that it's harder to memoize if you write it using conditional expressions.

附录 A　Debugging

When you are debugging, you should distinguish among different kinds of errors in order to track them down more quickly:

- Syntax errors are discovered by the interpreter when it is translating the source code into byte code. They indicate that there is something wrong with the structure of the program. Example: Omitting the colon at the end of a `def` statement generates the somewhat redundant message `SyntaxError: invalid syntax`.

- Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error "maximum recursion depth exceeded".

- Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

A.1　Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.
3. Make sure that any strings in the code have matching quotation marks. Make sure that all quotation marks are "straight quotes", not "curly quotes".
4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an `invalid token` error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
5. An unclosed opening operator—`(`, `{`, or `[`—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
6. Check for the classic = instead of == inside a conditional.
7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.
8. If you have non-ASCII characters in the code (including strings and comments), that might cause a problem, although Python 3 usually handles non-ASCII characters. Be careful if you paste in text from a web page or other source.

If nothing works, move on to the next section...

A.1.1 I keep making changes and it makes no difference.

If the interpreter says there is an error and you don't see it, that might be because you and the interpreter are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run.

If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run it again. If the interpreter doesn't find the new error, you are not running the new code.

There are a few likely culprits:

- You edited the file and forgot to save the changes before running it again. Some programming environments do this for you, but some don't.
- You changed the name of the file, but you are still running the old name.
- Something in your development environment is configured incorrectly.
- If you are writing a module and using `import`, make sure you don't give your module the same name as one of the standard Python modules.
- If you are using `import` to read a module, remember that you have to restart the interpreter or use `reload` to read a modified file. If you import the module again, it doesn't do anything.

If you get stuck and you can't figure out what is going on, one approach is to start again with a new program like "Hello, World!", and make sure you can get a known program to run. Then gradually add the pieces of the original program to the new one.

A.2 Runtime errors

Once your program is syntactically correct, Python can read it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke a function to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure there is a function call in the program, and make sure the flow of execution reaches it (see "Flow of Execution" below).

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is "hanging". Often that means that it is caught in an infinite loop or infinite recursion.

- If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop".

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

  If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

- If neither of those steps works, start testing other loops and other recursive functions and methods.

- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x:␣', x)
    print('y:␣', y)
    print("condition:␣", (x > 0 and y < 0))
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `False`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, infinite recursion causes the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function is causing an infinite recursion, make sure that there is a base case. There should be some condition that causes the function to return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function that prints the parameters. Now when you run the program, you will see a few lines of output every time the function is invoked, and you will see the parameter values. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add `print` statements to the beginning of each function with a message like "entering function `foo`", where `foo` is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

A.2.3 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that called it, and then the function that called *that*, and so on. In other words, it traces the sequence of function calls that got you to where you are, including the line number in your file where each call occurred.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment. Check if the name is spelled right, or at least consistently. And remember that local variables are local; you cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
- You are passing the wrong number of arguments to a function. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking

the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key that the dictionary does not contain. If the keys are strings, remember that capitalization matters.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use the built-in function `vars` to list the attributes that do exist.

If an AttributeError indicates that an object has `NoneType`, that means that it is `None`. So the problem is not the attribute name, but the object.

The reason the object is none might be that you forgot to return a value from a function; if you get to the end of a function without hitting a `return` statement, it returns `None`. Another common cause is using the result from a list method, like `sort`, that returns `None`.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

The Python debugger (`pdb`) is useful for tracking down exceptions because it allows you to examine the state of the program immediately before the error. You can read about `pdb` at `https://docs.python.org/3/library/pdb.html`.

A.2.4 I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are searching a list, search a *small* list. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think shouldn't affect the program, and it does, that can tip you off.

A.3 Semantic errors

In some ways, semantic errors are the hardest to debug, because the interpreter provides no information about what is wrong. Only you know what the program is supposed to do.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "stepping" the program to where the error is occurring.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves functions or methods in other Python modules. Read the documentation for the functions you call. Try them out by writing simple test cases and checking the results.

In order to program, you need a mental model of how programs work. If you write a program that doesn't do what you expect, often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
 y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the order of operations.

A.3.3 I've got a function that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the result before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()
return count
```

Now you have the opportunity to display the value of `count` before returning.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these symptoms:

- Frustration and rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. You need a fresh pair of eyes.

Before you bring someone else in, make sure you are prepared. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have `print` statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

附录 B　Analysis of Algorithms | 算法分析

This appendix is an edited excerpt from *Think Complexity*, by Allen B. Downey, also published by O'Reilly Media (2012). When you are done with this book, you might want to move on to that one.

本附录摘自 Allen B. Downey 的 *Think Complexity*, 也由 O' Reilly Media (2011) 出版。当你读完本书后，也许你可以参考这本读读。

Analysis of algorithms is a branch of computer science that studies the performance of algorithms, especially their run time and space requirements. See http://en.wikipedia.org/wiki/Analysis_of_algorithms.

算法分析 (Analysis of algorithms) 是计算机科学的一个分支，着重研究算法的性能，特别是他们的运行时间和资源开销。见：算法分析(维基百科) 。

The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions.

算法分析的实际目的是预测不同算法的性能，用于指导设计层的决策。

During the 2008 United States Presidential Campaign, candidate Barack Obama was asked to perform an impromptu analysis when he visited Google. Chief executive Eric Schmidt jokingly asked him for "the most efficient way to sort a million 32-bit integers." Obama had apparently been tipped off, because he quickly replied, "I think the bubble sort would be the wrong way to go." See http://www.youtube.com/watch?v=k4RRi_ntQc8.

2008 年美国总统大选期间，当候选人奥巴马 (Barack Obama) 访问 Google 时，他被要求进行即席的分析。首席执行官 Eric Schmidt 开玩笑的问他 "对一百万个 32 位整数排序的最有效的方法"。显然有人暗中通知了奥巴马，因为他很快回答，"我认为不应该采用冒泡排序法"。详见这个视频。

This is true: bubble sort is conceptually simple but slow for large datasets. The answer Schmidt was probably looking for is "radix sort" (http://en.wikipedia.org/wiki/Radix_sort)[1].

[1] But if you get a question like this in an interview, I think a better answer is, "The fastest way to sort a

是真的：冒泡排序概念上很简单，但是对于大数据集速度非常慢。Schmidt 寻找的答案可能是 “基数排序 (radix sort)”[2]。

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

算法分析的目的是在不同算法间进行有意义的比较，但是有一些问题：

- **The relative performance of the algorithms might depend on characteristics of the hardware, so one algorithm might be faster on Machine A, another on Machine B. The general solution to this problem is to specify a machine model and analyze the number of steps, or operations, an algorithm requires under a given model.**

- 算法相对的性能依赖于硬件的特性，因此一个算法可能在机器 A 上比较快，另一个算法则在机器 B 上比较快。对此问题一般的解决办法是指定一个机器模型 (machine model) 并且分析一个算法在一个给定模型下所需的步骤或运算的数目。

- **Relative performance might depend on the details of the dataset. For example, some sorting algorithms run faster if the data are already partially sorted; other algorithms run slower in this case. A common way to avoid this problem is to analyze the worst case scenario. It is sometimes useful to analyze average case performance, but that's usually harder, and it might not be obvious what set of cases to average over.**

- 相对性能可能依赖于数据集的细节。例如，如果数据已经部分排好序，一些排序算法可能更快；此时其它算法运行的比较慢。避免该问题的一般方法是分析最坏情况。有时分析平均情况性能，但那通常更难而且可能对什么案例的集合进行平均并不明显。

- **Relative performance also depends on the size of the problem. A sorting algorithm that is fast for small lists might be slow for long lists. The usual solution to this problem is to express run time (or number of operations) as a function of problem size, and group functions into categories depending on how quickly they grow as problem size increases.**

- 相对性能也依赖于问题的规模。一个对于小列表很快的排序算法可能对于长列表很慢。对此问题通常的解决方法是将运行时间（或则运算的数目）表示成问题规模的函数，并且随着问题规模的增长渐近地 (asymptotically) 比较函数。

million integers is to use whatever sort function is provided by the language I'm using. Its performance is good enough for the vast majority of applications, but if it turned out that my application was too slow, I would use a profiler to see where the time was being spent. If it looked like a faster sort algorithm would have a significant effect on performance, then I would look around for a good implementation of radix sort."

[2]但是，如果你采访中被问到这个问题，更好的答案可能是，“对上百万个整数的最快的排序方法就是用你所使用的语言的内建排序函数。它的性能对于大多数应用而言已优化的足够好。但如果是我自己写的排序程序运行太慢，我会用性能分析器找出大量的运算时间被用在了哪儿。如果一个更快的算法会对性能产生显著的提升，我会先试试基数排序。”

The good thing about this kind of comparison is that it lends itself to simple classification of algorithms. For example, if I know that the run time of Algorithm A tends to be proportional to the size of the input, n, and Algorithm B tends to be proportional to n^2, then I expect A to be faster than B, at least for large values of n.

关于此类比较的好处是对算法的简单分类。例如，如果我知道算法 **A** 的运行时间与输入的规模 n 成正比，算法 **B** 与 n^2 成正比，那么我期望对于很大的 n 值，**A** 比 **B** 快。

This kind of analysis comes with some caveats, but we'll get to that later.

这类分析也有一些问题，我们后面会提到。

B.1 Order of growth | 增长阶数

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n + 1$ steps to solve a problem with size n; Algorithm B takes $n^2 + n + 1$ steps.

假设你已经分析了两个算法并能用输入计算量的规模表示它们的运行时间：若算法 **A** 用 $100n + 1$ 步解决一个规模为 n 的问题；而算法 **B** 用 $n^2 + n + 1$ 步。

The following table shows the run time of these algorithms for different problem sizes:

下表显示了这些算法对于不同问题规模的运行时间：

Input size	**Run time of Algorithm A**	**Run time of Algorithm B**
10	**1 001**	**111**
100	**10 001**	**10 101**
1 000	**100 001**	**1 001 001**
10 000	**1 000 001**	$> 10^{10}$

At $n = 10$, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B. But for $n = 100$ they are about the same, and for larger values A is much better.

当 $n = 10$ 时，算法 **A** 看上去很糟糕，它用 **10** 倍于算法 **B** 所需的时间。但当 $n = 100$ 时，它们性能几乎相同，而 n 取更大值时，算法 **A** 要好得多。

The fundamental reason is that for large values of n, any function that contains an n^2 term will grow faster than a function whose leading term is n. The leading term is the term with the highest exponent.

根本原因是对于较大的 n 值，任何包含 n^2 项的函数都比首项为 n 的函数增长要快。首项 **(leading term)** 是具有最高指数的项。

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small n. But regardless of the coefficients, there will always be some value of n where $an^2 > bn$, for any values of a and b.

对于算法 A，首项有一个较大的系数 100，这是为什么对于小 n，B 比 A 好。但是不考虑该系数，总有一些 n 值使得 $an^2 > bn$。

The same argument applies to the non-leading terms. Even if the run time of Algorithm A were $n + 1000000$, it would still be better than Algorithm B for sufficiently large n.

同样的理由适用于非首项。即使算法 A 的运行时间为 $n + 1000000$，对于足够大的 n，它仍然比算法 B 好。

In general, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a crossover point where another algorithm is better. The location of the crossover point depends on the details of the algorithms, the inputs, and the hardware, so it is usually ignored for purposes of algorithmic analysis. But that doesn't mean you can forget about it.

一般来讲，我们希望一个算法（的增长阶数）有一个较小的首项，使得对于规模大的问题是一个好算法，但是规模小的问题，可能存在有一个交叉点 (crossover point)，在此规模以下，另一个算法更好。交叉点的位置依赖于算法的细节、输入以及硬件，因此对于算法分析目的，它通常被忽略。但是这不意味着你可以忘记它。

If two algorithms have the same leading order term, it is hard to say which is better; again, the answer depends on the details. So for algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients.

如果两个算法有相同的首项，很难说哪个更好。再次，答案依赖于细节。所以，对于算法分析，具有相同首项的函数被认为是相当的，即使它们具有不同的系数。

An order of growth is a set of functions whose growth behavior is considered equivalent. For example, $2n$, $100n$ and $n + 1$ belong to the same order of growth, which is written $O(n)$ in Big-Oh notation and often called linear because every function in the set grows linearly with n.

增长阶数[3] (order of growth) 是一个函数集合，其渐近的增长行为被认为是相当的。例如 $2n$、$100n$ 和 $n+1$ 属于相同的增长阶数，被用 大 O 符号 (Big-Oh notation) 写成 $O(n)$，而且通常被称作线性的 (linear)，因为集合中的每个函数根据 n 线性增长。

All functions with the leading term n^2 belong to $O(n^2)$; they are called quadratic.

首项为 n^2 的函数属于 $O(n^2)$。它们是二次方的 (quadratic)，对于首项为 n^2 的函数，这是一个有趣的词。

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

下表按照计算性能开销效率降序顺序排列显示了算法分析中最通常的一些增长阶数。

[3]译注：又译增长数量级，增长级；即：算法性能

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

增长阶数	名称
$O(1)$	常数级
$O(\log_b n)$	对数级 (对于任意 b)
$O(n)$	线性级
$O(n \log_b n)$	线性对数级
$O(n^2)$	二次方级
$O(n^3)$	三次方级
$O(c^n)$	指数级 (对于任意 c)

For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

对于对数级数，对数的基数并不影响增长阶数。改变阶数等价于乘以一个常数，其不改变增长阶数。相应的，所有的指数级数都属于相同的增长阶数，而无需考虑指数的基数大小指数函数增长阶数增长的非常快，因此指数级算法只用于小规模问题。

Exercise B.1. *Read the Wikipedia page on Big-Oh notation at* `http://en.wikipedia.org/wiki/Big_O_notation` *and answer the following questions:*

阅读维基百科关于 大 O 符号 的介绍，回答以下问题：

1. *What is the order of growth of* $n^3 + n^2$*? What about* $1000000n^3 + n^2$*? What about* $n^3 + 1000000n^2$*?*

2. *What is the order of growth of* $(n^2 + n) \cdot (n + 1)$*? Before you start multiplying, remember that you only need the leading term.*

3. *If* f *is in* $O(g)$*, for some unspecified function* g*, what can we say about* $af + b$*?*

4. *If* f_1 *and* f_2 *are in* $O(g)$*, what can we say about* $f_1 + f_2$*?*

5. *If* f_1 *is in* $O(g)$ *and* f_2 *is in* $O(h)$*, what can we say about* $f_1 + f_2$*?*

6. *If* f_1 *is in* $O(g)$ *and* f_2 *is* $O(h)$*, what can we say about* $f_1 \cdot f_2$*?*

Programmers who care about performance often find this kind of analysis hard to swallow. They have a point: sometimes the coefficients and the non-leading terms make a real difference. Sometimes the details of the hardware, the programming language, and the characteristics of the input make a big difference. And for small problems asymptotic behavior is irrelevant.

关注性能的程序员经常发现这种分析很难忍受。他们有一个观点：有时系数和非首项会造成巨大的影响。有时，硬件的细节、编程语言以及输入的特性会造成很大的影响。对于小问题，渐近的行为没有什么影响。

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the "better" algorithms is usually better, and sometimes it is *much* better. The difference between two algorithms with the same order of growth is usually a constant factor, but the difference between a good algorithm and a bad algorithm is unbounded!

但是，如果你记得那些警告，算法分析就是一个有用的工具。至少对于大问题"更好的"算法通常更好，并且有时它要好的多。相同增长阶数的两个算法之间的不同通常是一个常数因子，但是一个好算法和一个坏算法之间的不同是无限的！

B.2 Analysis of basic Python operations | Python 基本运算操作分析

In Python, most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases with the number of digits.

大部分算术运算的开销是常数级的。乘法会比加减法用更长的时间，除法更长，但是这些运算时间不依赖被运算数的数量级。非常大的整数却是个例外，在这种情况下，运行时间随着位数的增加而增加。

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

索引操作 — 在序列或字典中读写元素 — 在不考虑数据结构的大小的情况下也是常数级的。

A for loop that traverses a sequence or dictionary is usually linear, as long as all of the operations in the body of the loop are constant time. For example, adding up the elements of a list is linear:

一个遍历序列或字典的 for 循环通常是线性的，只要循环体内的运算是常数时间。例如，累加一个列表的元素是线性的：

```
total = 0
for x in t:
    total += x
```

The built-in function sum is also linear because it does the same thing, but it tends to be faster because it is a more efficient implementation; in the language of algorithmic analysis, it has a smaller leading coefficient.

内建求和函数 `sum` 也是线性的，因为它做相同的事情，但是它倾向于更快因为它是一个更有效的实现。从算法分析角度讲，它具有更小的首项系数。

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if you can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n, then the loop is in $O(n^a)$, even for large k.

根据经验, 如果循环体内的增长阶数是 $O(n^a)$, 则整个循环的增长阶数是 $O(n^{a+1})$。如果这个循环在执行一定数目循环后退出则是例外。无论 n 取值多少，如果循环仅执行 k 次，整个循环的增长阶数是 $O(n^a)$，即便 k 值比较大。

Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k.

乘上 k 并不会改变增长阶数，除法也是。因此，如果循环体是 $O(n^a)$，循环执行 n/k 次，整个循环的增长阶数就是 $O(n^{a+1})$, 即使 k 值很大。

Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

大部分字符串和元组运算是线性的，除了索引和 `len`，它们是常数时间。内建函数 `min` 和 `max` 是线性的。划分运算与输出的长度成正比，但是和输入的大小无关。

String concatenation is linear; the run time depends on the sum of the lengths of the operands.

字符串拼接是线性的；它的运算时间取决于操作字符的总长度。

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time. The string method `join` is linear; the run time depends on the total length of the strings.

所有字符串方法是线性的，但是如果字符串的长度受限于一个常数 — 例如，在一个字符上运算一它们被认为是常数时间 — 它们都被认为是恒定时间。字符串方法 `join` 也是线性的；它的运算时间取决于字符串的总长度。

Most list methods are linear, but there are some exceptions:

大部分列表方法是线性的，但是有一些例外：

- **Adding an element to the end of a list is constant time on average; when it runs out of room it occasionally gets copied to a bigger location, but the total time for n operations is $O(n)$, so the average time for each operation is $O(1)$.**

- 平均来讲，在列表结尾增加一个元素是常数时间。当它超出了所占用空间时，它偶尔被拷贝到一个更大的地方，但是对于 n 个运算的整体时间仍为 $O(n)$ ，所以我们说一个运算的“分摊”时间是 $O(1)$ 。

- Removing an element from the end of a list is constant time.
- 从一个列表结尾删除一个元素是常数时间。
- Sorting is $O(n \log n)$.
- 排序是 $O(n \log n)$ 。

Most dictionary operations and methods are constant time, but there are some exceptions:

大部分字典运算和方法是常数时间，但有些例外：

- The run time of `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.
- `update` 的运行时间正比于作为形参被传递的字典的大小，而不是被更新的字典。
- `keys`, `values` and `items` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear.
- `keys`、`values` 和 `items` 是连续常数时间因为它们返回迭代器。但是如果你对迭代器进行循环，循环将是线性的。

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section B.4.

字典的性能是计算机科学的一个小奇迹之一。在 hashtable 节中，我们将看到它们是如何工作的。

Exercise B.2. *Read the Wikipedia page on sorting algorithms at* http://en.wikipedia.org/wiki/Sorting_algorithm *and answer the following questions:*

阅读排序算法在维基百科的介绍，回答下面的问题：

1. *What is a "comparison sort?" What is the best worst-case order of growth for a comparison sort? What is the best worst-case order of growth for any sort algorithm?*
2. 什么是 "比较排序 (*comparison sort*)"? 比较排序最优最差情况的增长阶数是多少？别的排序算法的最优最差情况的增长阶数又是多少？
3. *What is the order of growth of bubble sort, and why does Barack Obama think it is "the wrong way to go?"*
4. 冒泡排序法的增长阶数是多少？为什么奥巴马认为是"不应采用的方法"
5. *What is the order of growth of radix sort? What preconditions do we need to use it?*
6. 基数排序 (*radix sort*) 的增长阶数是多少？我们使用它所需要注意的前提条件有哪些？
7. *What is a stable sort and why might it matter in practice?*
8. 排序算法的稳定性是指什么？为什么它在实际操作中很重要？

9. *What is the worst sorting algorithm (that has a name)?*

10. *什么是 worst sorting algorithm ?*

11. *What sort algorithm does the C library use? What sort algorithm does Python use? Are these algorithms stable? You might have to Google around to find these answers.*

12. *C 语言使用哪种排序算法？Python 使用哪种排序算法？这些算法稳定吗？你可以谷歌一下这些答案。*

13. *Many of the non-comparison sorts are linear, so why does does Python use an $O(n \log n)$ comparison sort?*

14. *大多数 non-comarison 算法是线性的，因此为什 Python 使用一个 $O(n \log n)$ 的 comparison sort ?*

B.3 Analysis of search algorithms | 搜索算法分析

A search is an algorithm that takes a collection and a target item and determines whether the target is in the collection, often returning the index of the target.

搜索 **(search)** 算法，其接受一个集合以及一个目标项，并决定该目标项是否在集合中，通常返回目标的索引值。

The simplest search algorithm is a "linear search", which traverses the items of the collection in order, stopping if it finds the target. In the worst case it has to traverse the entire collection, so the run time is linear.

最简单的搜索算法是 "线性搜索"，其按顺序遍历集合中的项，如果找到目标则停止。最坏的情况下，它不得不遍历全部集合，所以运行时间是线性的。

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

序列的 **in** 运算符使用线性搜索。字符串方法，如 `find` 和 `count` 也是这样。

If the elements of the sequence are in order, you can use a bisection search, which is $O(\log n)$. Bisection search is similar to the algorithm you might use to look a word up in a dictionary (a paper dictionary, not the data structure). Instead of starting at the beginning and checking each item in order, you start with the item in the middle and check whether the word you are looking for comes before or after. If it comes before, then you search the first half of the sequence. Otherwise you search the second half. Either way, you cut the number of remaining items in half.

如果元素在序列中是排序好的，你可以用二分搜素 **(bisection search)**，其是 $O(\log n)$ 。二分搜索和你在字典中查找一个单词的算法类似（这里是指真正的字典，不是数据结构）。你不会从头开始并按顺序检查每个项，而是从中间的项开始并检查你要查找的单词在前面还是后面。如果它出现在前面，那么你搜索序列的前半部分。否则你搜索后一半。如论如何，你将剩余的项数分为一半。

If the sequence has 1,000,000 items, it will take about 20 steps to find the word or conclude that it's not there. So that's about 50,000 times faster than a linear search.

如果序列有 1,000,000 项，它将花 20 步找到该单词或说找不到。因此它比线性搜索快大概 50,000 倍。

Bisection search can be much faster than linear search, but it requires the sequence to be in order, which might require extra work.

二分搜索比线性搜索快很多，但是它要求已排序的序列，因此使用时会有额外的工作要做。

There is another data structure, called a hashtable that is even faster—it can do a search in constant time—and it doesn't require the items to be sorted. Python dictionaries are implemented using hashtables, which is why most dictionary operations, including the `in` operator, are constant time.

另一个检索速度更快的数据结构被称为哈希表 (hashtable) — 它可以在常数时间内检索出结果 — 并且不依赖于序列是否已排序。Python 的内建字典就通过哈希表技术而实现，因此大多数的字典操作，包括 `in` ，只花费常数时间就可完成。

B.4 Hashtables | 哈希表

To explain how hashtables work and why their performance is so good, I start with a simple implementation of a map and gradually improve it until it's a hashtable.

为了解释哈希表是如何工作，以及为什么它的性能非常优秀，我们从实现一个简单的映射 (map) 开始并逐步改进它，直到成为一个哈希表。

I use Python to demonstrate these implementations, but in real life you wouldn't write code like this in Python; you would just use a dictionary! So for the rest of this chapter, you have to imagine that dictionaries don't exist and you want to implement a data structure that maps from keys to values. The operations you have to implement are:

我们使用 Python 来演示这些实现案例，在真实情况下，你用不着用 Python 亲自写这样的代码，只需用内建的字典对象！因此以下的内容，是基于假设我们需要的字典对象并不存在，并且我们想实现一个数据结构，将关键字映射为值。你需要实现如下的函数运算：

`add(k, v)`: Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.

`get(k)`: Look up and return the value that corresponds to key `k`. With a Python dictionary, `d`, this operation is written `d[k]` or `d.get(k)`.

`add(k, v)`: 增加一个新的项，其从关键字 k 映射到值 v。使用 Pythong 的字典 d，该运算被写作 d[k] = v。

get(k)：查找并返回相应关键字为 target 的值。使用 Pythong 的字典 d，该运算被写作 d[target] 或 d.get(target)。

For now, I assume that each key only appears once. The simplest implementation of this interface uses a list of tuples, where each tuple is a key-value pair.

现在，假设每个关键字只出现一次。该接口最简单的实现是使用一个元组列表，其中每个元组是关键字-值对。

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

add appends a key-value tuple to the list of items, which takes constant time.

add 向项列表追加一个关键字---值元组，这是常数时间。

get uses a for loop to search the list: if it finds the target key it returns the corresponding value; otherwise it raises a KeyError. So get is linear.

get 使用 for 循环搜索该列表：如果它找到目标关键字，返回相应的值；否则触发一个 KeyError。因此 get 是线性的。

An alternative is to keep the list sorted by key. Then get could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures that can implement add and get in log time, but that's still not as good as constant time, so let's move on.

另一个方案是保持列表按关键字排序。那么 get 可以使用二分搜索，其是 $O(\log n)$ 。但是在列表中间插入一个新的项是线性的，因此这可能不是最好的选择。有其它的数据结构（见：维基百科）能在对数级时间内实现 add 和 get，但是这仍然不如常数时间好，那么我们继续。

One way to improve LinearMap is to break the list of key-value pairs into smaller lists. Here's an implementation called BetterMap, which is a list of 100 LinearMaps. As we'll see in a second, the order of growth for get is still linear, but BetterMap is a step on the path toward hashtables:

另一种改良 LinearMap 的方法是将关键字-值对的列表分成小列表。这是一个被称作 BetterMap 的更好的实现，它是 100 个 LinearMaps 的列表。正如一会儿我们将看到的，get 的增长阶数仍然是线性的，但是 BetterMap 是迈向哈希表的一步。

```python
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` makes a list of n LinearMaps.

`__init__` 产生 n 个 `LinearMap` 列表。

`find_map` is used by `add` and `get` to figure out which map to put the new item in, or which map to search.

`find_map` 被 `add` 和 `get` 用来指出在哪个 `map` 中加入新项或则搜索哪个 `map`。

`find_map` uses the built-in function `hash`, which takes almost any Python object and returns an integer. A limitation of this implementation is that it only works with hashable keys. Mutable types like lists and dictionaries are unhashable.

`find_map` 使用内建 `hash` 函数，其接受几乎任何 Python 对象并返回一个整数。这一实现的一个限制是它仅适用于哈希表关键字。如列表和字典等易变的类型是不能哈希的。

Hashable objects that are considered equivalent return the same hash value, but the converse is not necessarily true: two objects with different values can return the same hash value.

被认为是相等的可哈希的对象返回相同的哈希值，但是反之不必成立：两个不同的对象能够返回相同的哈希值。

`find_map` uses the modulus operator to wrap the hash values into the range from 0 to `len(self.maps)`, so the result is a legal index into the list. Of course, this means that many different hash values will wrap onto the same index. But if the hash function spreads things out pretty evenly (which is what hash functions are designed to do), then we expect $n/100$ items per LinearMap.

`find_map` 使用求余运算符将哈希值包在 0 到 `len(self.maps)` 之间，因此结果是对于该列表合法的索引值。当然，这意味着许多不同的哈希值将被包成相同的索引值。但是如果哈希函数散布相当均匀（这是哈希函数被设计的初衷），那么我们期望每个 `LinearMap` 有 $n/100$ 项。

Since the run time of `LinearMap.get` is proportional to the number of items, we expect BetterMap to be about 100 times faster than LinearMap. The order of growth is still linear, but the leading coefficient is smaller. That's nice, but still not as good as a hashtable.

既然 LinearMap.get 的运行时间与项数成正比，我们期望 BetterMap 比 LinearMap 快 100 倍。增长阶数仍然是线性的，但是首系数变小了。这很好，但是仍然不如哈希表好。

Here (finally) is the crucial idea that makes hashtables fast: if you can keep the maximum length of the LinearMaps bounded, LinearMap.get is constant time. All you have to do is keep track of the number of items and when the number of items per LinearMap exceeds a threshold, resize the hashtable by adding more LinearMaps.

在此（最终）是使哈希表变快的关键的想法：如果你能保证 LinearMaps 的最大长度是受限的，则 LinearMap.get 是常数时间。所有你需要做的是跟踪项数并且当每个 LinearMap 的项数超过一个阈值时，通过增加更多的 LinearMaps 调整哈希表的大小。

Here is an implementation of a hashtable:

这是哈希表的一个实现：

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

Each HashMap contains a BetterMap; __init__ starts with just 2 LinearMaps and initializes num, which keeps track of the number of items.

每个 HashMap 包含一个 BetterMap。__init__ 仅以两个 LinearMaps 开始并且初始化 num，其跟踪项的数目。

get just dispatches to BetterMap. The real work happens in add, which checks the number of items and the size of the BetterMap: if they are equal, the average number of items per LinearMap is 1, so it calls resize.

get 仅仅调度 BetterMap。真正的工作发生于 add 内，其检查项的数目以及 BetterMap 的大小：如果它们相同，每个 LinearMap 的平均项数为 1，因此它调用 resize。

resize make a new BetterMap, twice as big as the previous one, and then “rehashes” the items from the old map to the new.

resize 生成一个新的 BetterMap，是之前的两倍大，然后从旧的 map 到新的“重哈希”。

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in find_map. That means that some objects that used to hash into the same LinearMap will get split up (which is what we wanted, right?).

重哈希是必要的，因为改变 LinearMaps 的数目也改变了 find_map 中求余运算的分母。那意味着一些被包进相同的 LinearMap 的对象将被分离（这正是我们希望的，对吧？）

Rehashing is linear, so resize is linear, which might seem bad, since I promised that add would be constant time. But remember that we don’t have to resize every time, so add is usually constant time and only occasionally linear. The total amount of work to run add n times is proportional to n, so the average time of each add is constant time!

重哈希是线性的，因此 resize 是线性的，这可能看起来很糟糕，既然我保证 add 会是常数时间。但是记住，我们不必没每次都调整，因此 add 通常是常数时间并且只是偶然是线性的。运行 add n 次的整个工作的数目是与 n 成正比，因此 add 的平均时间是常数时间！

To see how this works, think about starting with an empty HashTable and adding a sequence of items. We start with 2 LinearMaps, so the first 2 adds are fast (no resizing required). Let’s say that they take one unit of work each. The next add requires a resize, so we have to rehash the first two items (let’s call that 2 more units of work) and then add the third item (one more unit). Adding the next item costs 1 unit, so the total so far is 6 units of work for 4 items.

为了看清这是如何工作的，考虑以一个空的哈希表开始并增加一系列项。我们以两个 LinearMap 开始，因此前两个 add 很快（不需要 resize）。我们说它们每个花费一个工作单元。下一个 add 需要一次 resize，因此我们必须重哈希前两项（我们调用两个额外的工作单元）然后增加第 3 项（一个额外单语）。增加下一项花费 1 个单元，所以对于 4 项总共需要 6 个单元。

The next add costs 5 units, but the next three are only one unit each, so the total is 14 units for the first 8 adds.

下一个 add 花费 5 个单元，但是之后的 3 个每个只需要 1 个单元，所以前 8 个 add 总共需要 14 个单元。

The next add costs 9 units, but then we can add 7 more before the next resize, so the total is 30 units for the first 16 adds.

下一个 add 花费 9 个单元，但是之后在下一次 resize 之前，可以增加额外的 7 个，所以前 16 个 add 总共是 30 个单元。

After 32 adds, the total cost is 62 units, and I hope you are starting to see a pattern. After n adds, where n is a power of two, the total cost is $2n - 2$ units, so the average work per add is a little less than 2 units. When n is a power of two, that’s the

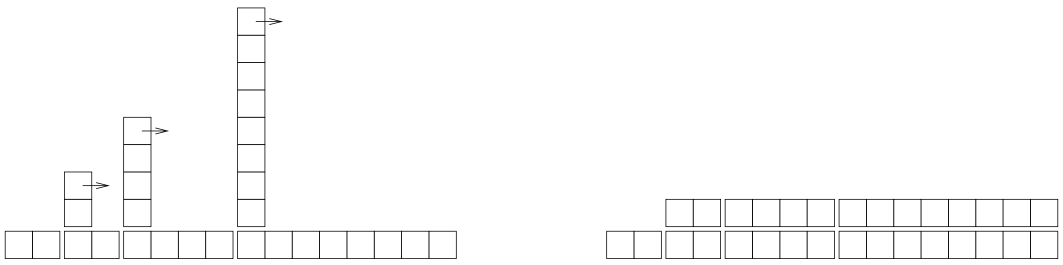

图 B.1: The cost of a hashtable add.

best case; for other values of n the average work is a little higher, but that's not important. The important thing is that it is $O(1)$.

在 32 次 add 后，总共花费 62 个单元，我希望你开始看到一个模式。n 次 add 后，其中 n 是 2 的指数，总共花费是 $2n-2$ 个单元，所以平均每个 add 要稍微少于 2 个单元。当 n 是 2 的指数时是最好的情况。对于其它的 n 值，平均花费稍高一点，但是那并不重要。重要的事情是增长阶数为 $O(1)$ 。

Figure B.1 shows how this works graphically. Each block represents a unit of work. The columns show the total work for each add in order from left to right: the first two adds cost 1 units, the third costs 3 units, etc.

图 B.1展示这如何工作的。每个块代表一个工作单元。按从左到右的顺序，每列显示每个 add 所需的单元：前两个 adds 花费 1 个单元，第 3 个花费 3 个单元等等。

The extra work of rehashing appears as a sequence of increasingly tall towers with increasing space between them. Now if you knock over the towers, spreading the cost of resizing over all adds, you can see graphically that the total cost after n adds is $2n-2$.

重哈希的额外工作显示为一序列增加的高塔并在它们之间增加空间。现在，如果你打翻这些塔，将 resize 的代价均摊到所有的 add 上，你会从图上看到 n 个 add 的整个花费是 $2n-2$ 。

An important feature of this algorithm is that when we resize the HashTable it grows geometrically; that is, we multiply the size by a constant. If you increase the size arithmetically—adding a fixed number each time—the average time per add is linear.

该算法一个重要的特征是当我们 resize 哈希表的时候，它几何级增长。也就是说，我们用常数乘以大小。如果你按算术级增加大小——每次增加固定的数目—每个 add 的平均时间是线性的。

You can download my implementation of HashMap from http://thinkpython2.com/code/Map.py, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

你可以 在此下载 到 HashMap 的实现代码，但在实际环境中直接使用 Python 的字典足矣。

B.5 Glossary | 术语表

analysis of algorithms: A way to compare algorithms in terms of their run time and/or space requirements.

算法分析 比较不同算法间运行时间和资源占用的分析方法。

machine model: A simplified representation of a computer used to describe algorithms.

模型机器 用于描述算法（性能）的简化的计算机表示。

worst case: The input that makes a given algorithm run slowest (or require the most space.

最坏情况 对给定算法话最长时间运行（或占用做多资源）的输入。

leading term: In a polynomial, the term with the highest exponent.

首项 在多项式中，拥有指数最高的项

crossover point: The problem size where two algorithms require the same run time or space.

交叉点 两个算法对求解给定问题在某个规模需要相同运行时间或资源开销。

order of growth: A set of functions that all grow in a way considered equivalent for purposes of analysis of algorithms. For example, all functions that grow linearly belong to the same order of growth.

增长阶数 一些函数的组合，其函数值的增长在某种程度上等同于算法分析考察的目标。例如，线性递增的所有的函数都属于同一个增长阶数。

Big-Oh notation: Notation for representing an order of growth; for example, $O(n)$ represents the set of functions that grow linearly.

大 **O** 符号 代表增长阶数的符号；例如，$O(n)$ 代表线性递增的函数。

linear: An algorithm whose run time is proportional to problem size, at least for large problem sizes.

线性级 算法的运行时间和所求解问题的规模成比例，至少是在问题规模较大时显现。

quadratic: An algorithm whose run time is proportional to n^2, where n is a measure of problem size.

二次方级 算法的运行时间和说求解问题的规模的二次方 (n^2) 成比例，n 用于描述问题的规模。

search: The problem of locating an element of a collection (like a list or dictionary) or determining that it is not present.

检索 定位一个集合（例如列表或字典）中某个元素位置的问题，也等价于确认某元素不再该集合中的问题。

hashtable: A data structure that represents a collection of key-value pairs and performs search in constant time.

哈希表 一种数据结构用于描述键-值对的集合，在该类集合运行检索任务只花费固定的时间（不依赖于集合的大小）。

索引