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

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from http://www.thinkpython2.com

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. 你最好在电脑前阅读此书，因为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：

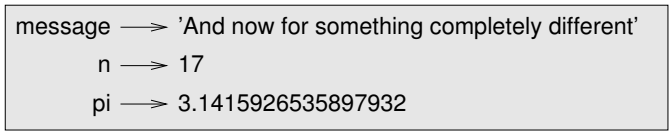

图 2.1: State diagram. | 状态图。

```
>>> 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__。

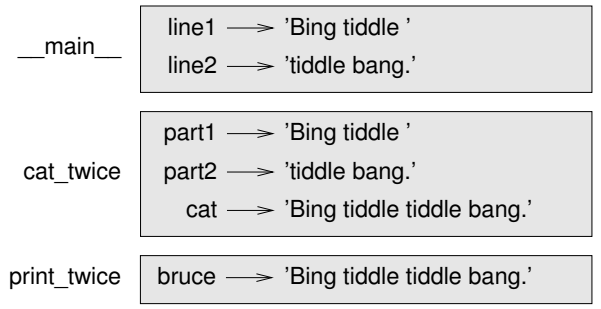

图 3.1: 堆栈图。

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 reuse—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。

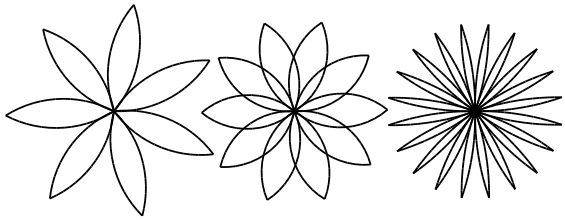

图 4.1: Turtle flowers.

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

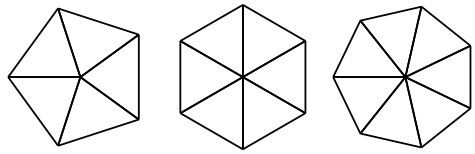

图 4.2: Turtle pies.

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　Conditionals and recursion | 条件和递归

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: floor division and modulus.

这章的中心话题是能够根据程序的状态执行不同命令的 if 语句。但是首先我想介绍两个新的运算符: 地板除法 (floor division) [1] 和求余 (modulus) 。

5.1　Floor division and modulus | 地板除法和求余

The **floor division** operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

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
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, dropping the fraction part:

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
```

[1] 译注：向下取整的除法。

To get the remainder, you could subtract off one hour in minutes:

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
```

An alternative is to use the **modulus operator**, `%`, which divides two numbers and returns the remainder.

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
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

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。

Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

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。

If you are using Python 2, division works differently. The division operator, `/`, performs floor division if both operands are integers, and floating-point division if either operand is a `float`.

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 Boolean expressions | 布尔表达式

A **boolean expression** is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

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`。

[2]译注：在 Python3 中，无论任何类型都会保持小数部分。

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

True and False are special values that belong to the type bool; they are not strings:

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'>
```

The == operator is one of the **relational operators**; the others are:

== 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
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator. There is no such thing as =< or =>.

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

There are three **logical operators**: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.

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。

n%2 == 0 or n%3 == 0 is true if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

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。

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false, that is, if x is less than or equal to y.

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。

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as True:

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
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

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 Conditional execution | 有条件执行

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the if statement:

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')
```

The boolean expression after if is called the **condition**. If it is true, the indented statement runs. If not, nothing happens.

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 statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**.

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) 。

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

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 Alternative execution | 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')
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**, because they are branches in the flow of execution.

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 Chained conditionals | 链式条件

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

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` is an abbreviation of "else if". Again, exactly one branch will run. There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

`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()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

程序将按顺序逐个检测条件，如果第一个为假，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，并且语句结束。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 Nested conditionals | 嵌套条件

One conditional can also be nested within another. We could have written the example in the previous section like this:

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')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

外层的条件 (outer conditional) 包括两个分支。第一个分支包括一条简单的语句。第二个分支又包括一个 `if` 语句，它有自己的两个分支。那两个分支都是简单的语句，当然它们也可以是条件语句。

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. It is a good idea to avoid them when you can.

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。

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

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.')
```

The `print` statement runs only if we make it past both conditionals, so we can get the same effect with the `and` operator:

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.')
```

For this kind of condition, Python provides a more concise option:

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 Recursion | 递归

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

一个函数调用另一个是合法的；一个函数调用它自己也是合法的。这样的好处可能并不是那么明显，但它实际上成为了程序能做到的最神奇的事情之一。例如，看一下这个程序：

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

If `n` is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.

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。

What happens if we call this function like this?

如果我们像这样调用该函数会发生什么呢？

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

The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, "Blastoff!" and then returns.
> >
> > The `countdown` that got `n=1` returns.
>
> The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you're back in `__main__`. So, the total output looks like this:

`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!
```

A function that calls itself is **recursive**; the process of executing it is called **recursion**.

一个调用它自己的函数是*递归的* (recursive)；这个过程被称作*递归* (recursion)。

As another example, we can write a function that prints a string `n` times.

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)
```

If `n <= 0` the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

如果 `n <= 0`，`return`语句 退出函数。执行流程马上返回到调用者，函数剩余的语句行不会被执行。

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display `s` $n - 1$ additional times. So the number of lines of output is `1 + (n - 1)`, which adds up to `n`.

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`。

For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

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 Stack diagrams for recursive functions | 递归函数的堆栈图

In Section 3.9, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

在 Section 3.9 一节中，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。

Every time a function gets called, Python creates a frame to contain the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

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。

Figure 5.1 shows a stack diagram for `countdown` called with `n = 3`.

图 5.1 展示了一个以 `n = 3` 调用 `countdown` 的堆栈图。

As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

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。

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

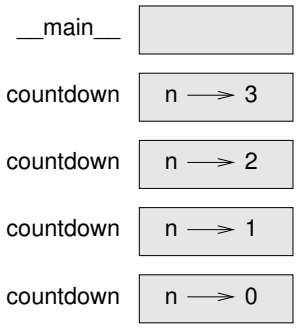

图 5.1: Stack diagram.

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。

As an exercise, draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=2`. Then write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function `n` times.

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 Infinite recursion | 无限递归

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

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()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

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
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

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*！

If you write encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

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 Keyboard input | 键盘输入

The programs we have written so far accept no input from the user. They just do the same thing every time.

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 provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string. In Python 2, the same function is called `raw_input`.

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?
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. `input` can take a prompt as an argument:

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!
```

The sequence \n at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

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。

If you expect the user to type an integer, you can try to convert the return value to `int`:

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
```

But if the user types something other than a string of digits, you get an error:

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
```

We will see how to handle this kind of error later.

我们后面将介绍处理这类错误的方法。

5.12 Debugging | 调试

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

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：

- What kind of error it was, and
- Where it occurred.
- 是哪类错误，以及
- 在哪儿出现。

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

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
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

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。

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

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)
```

When you run this program, you get an exception:

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
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, which uses floor division instead of floating-point division.

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。

You should take the time to read error messages carefully, but don't assume that everything they say is correct.

你应该花些时间仔细阅读错误信息，但是不要轻易地认为错误信息的提示都是准确的。

5.13 Glossary | 术语表

floor division: An operator, denoted //, that divides two numbers and rounds down (toward zero) to an integer.

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。

modulus operator: An operator, denoted with a percent sign (%), that works on integers and returns the remainder when one number is divided by another.

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。

boolean expression: An expression whose value is either `True` or `False`.

布尔表达式: 一个值要么为真要么为假的表达式。

relational operator: One of the operators that compares its operands: ==, !=, >, <, >=, and <=.

关系运算符: 对其运算符进行比较的运算符：==，!=，>，<，>=，<=。

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

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

conditional statement: A statement that controls the flow of execution depending on some condition.

条件语句: 一段根据某个条件决定程序执行流程的语句。

condition: The boolean expression in a conditional statement that determines which branch runs.

条件: 决定哪个分支会被执行的布尔表达式。

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

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。

branch: One of the alternative sequences of statements in a conditional statement.

分支: 条件语句中的选择性语句序列。

chained conditional: A conditional statement with a series of alternative branches.

链式条件: 由一系列替代分支组成的条件语句。

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

嵌套条件: 出现另一个条件语句某个分支中的条件语句。

return statement: A statement that causes a function to end immediately and return to the caller.

返回语句: 结束函数执行并且将结果返回给调用者的语句。

recursion: The process of calling the function that is currently executing.

递归: 调用正在执行的函数本身的过程。

base case: A conditional branch in a recursive function that does not make a recursive call.

基本情形: 在递归函数中，不进行递归调用的条件分支。

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

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 Exercises | 练习

Exercise 5.1. *The* `time` *module provides a function, also named* `time`*, that returns the current Greenwich Mean Time in "the epoch", which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.*

`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
```

Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.

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. *Fermat's Last Theorem says that there are no positive integers a, b, and c such that*

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

for any values of n greater than 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. *Write a function named* `check_fermat` *that takes four parameters—*`a`, `b`, `c` *and* `n`*—and checks to see if Fermat's theorem holds. If n is greater than 2 and*

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

   the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. 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.''*。

3. *Write a function that prompts the user to input values for* `a`, `b`, `c` *and* `n`*, converts them to integers, and uses* `check_fermat` *to check whether they violate Fermat's theorem.*

4. 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. *If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:*

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：

> *If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)*
>
> 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. *Write a function named* `is_triangle` *that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.*

2. 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"。

3. *Write a function that prompts the user to input three stick lengths, converts them to integers, and uses* `is_triangle` *to check whether sticks with the given lengths can form a triangle.*

4. 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. *What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.*

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. *What would happen if you called this function like this:* `recurse(-1, 0)`*?*

2. *Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).*

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）。

The following exercises use the `turtle` module, described in Chapter 四:

后面的习题要用到第 四章中的 `turtle`:

Exercise 5.5. *Read the following function and see if you can figure out what it does. Then run it (see the examples in Chapter 四).*

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)
```

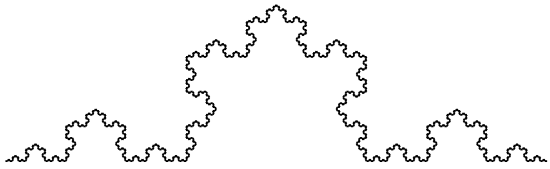

图 5.2: A Koch curve.

Exercise 5.6. *The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length x, all you have to do is*

科赫曲线 *(Koch Curve)* 是一个看起来类似图 5.2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. *Draw a Koch curve with length $x/3$.*
2. *Turn left 60 degrees.*
3. *Draw a Koch curve with length $x/3$.*
4. *Turn right 120 degrees.*
5. *Draw a Koch curve with length $x/3$.*
6. *Turn left 60 degrees.*
7. *Draw a Koch curve with length $x/3$.*

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。

The exception is if x is less than 3: in that case, you can just draw a straight line with length x.

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. *Write a function called* `koch` *that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.*
2. *Write a function called* `snowflake` *that draws three Koch curves to make the outline of a snowflake.*

   Solution: `http://thinkpython2.com/code/koch.py`.

3. *The Koch curve can be generalized in several ways. See* `http://en.wikipedia.org/wiki/Koch_snowflake` *for examples and implement your favorite.*

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　Fruitful functions | 有返回值的函数

Many of the Python functions we have used, such as the math functions, produce return values. But the functions we've written are all void: they have an effect, like printing a value or moving a turtle, but they don't have a return value. In this chapter you will learn to write fruitful functions.

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　Return values | 返回值

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

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)
```

The functions we have written so far are void. Speaking casually, they have no return value; more precisely, their return value is `None`.

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` 。

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

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
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This statement means: "Return immediately from this function and use the following expression as a return value." The expression can be arbitrarily complicated, so we could have written this function more concisely:

之前我们已经见过 `return` 语句了，但是在一个有返回值的函数中，`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
```

On the other hand, **temporary variables** like `a` can make debugging easier.

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

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

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
```

Since these `return` statements are in an alternative conditional, only one runs.

因为这些 `return` 语句在不同的条件内，最后只有一个会被执行。

As soon as a return statement runs, the function terminates without executing any subsequent statements. Code that appears after a `return` statement, or any other place the flow of execution can never reach, is called **dead code**.

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)。

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. For example:

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
```

This function is incorrect because if `x` happens to be 0, neither condition is true, and the function ends without hitting a `return` statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0.

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
```

By the way, Python provides a built-in function called abs that computes absolute values.

顺便说一下，Python 提供了一个的内建函数 abs 用来计算绝对值。

As an exercise, write a compare function takes two values, x and y, and returns 1 if x > y, 0 if x == y, and -1 if x < y.

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 Incremental development | 增量式开发

As you write larger functions, you might find yourself spending more time debugging.

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。

To deal with increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

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。

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

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：

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

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

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

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a floating-point value.

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。

Immediately you can write an outline of the function:

现在你就可以写出此函数的轮廓了：

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

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

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。

To test the new function, call it with sample arguments:

用样例实参调用它来进行测试。

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

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 triangle. When testing a function, it is useful to know the right answer.

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。

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. The next version stores those values in temporary variables and prints them.

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
```

If the function is working, it should display `dx is 3` and `dy is 4`. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

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。

Next we compute the sum of squares of `dx` and `dy`:

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
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use `math.sqrt` to compute and return the result:

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
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the return statement.

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。

The final version of the function doesn't display anything when it runs; it only returns a value. The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

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。

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

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。

The key aspects of the process are:

这种开发方式的关键是：

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.
2. Use variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

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。

As an exercise, use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments. Record each stage of the development process as you go.

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 Composition | 组合

As you should expect by now, you can call one function from within another. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

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。

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

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)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

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)
```

Encapsulating these steps in a function, we get:

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
```

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

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 Boolean functions | 布尔函数

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

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
```

It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.

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。

Here is an example:

请看下面的示例：

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

The result of the == operator is a boolean, so we can write the function more concisely by returning it directly:

== 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
```

Boolean functions are often used in conditional statements:

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')
```

It might be tempting to write something like:

很容易写出下面这样的代码：

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

But the extra comparison is unnecessary.

但这里的比较是多余的。

As an exercise, write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

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

6.5 More recursion | 再谈递归

We have only covered a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the mouse, disks, etc., but that's all).

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)。

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot

of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

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》。

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

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：

vorpal: An adjective used to describe something that is vorpal.

漩涡状： 一个用以描述漩涡状物体的形容词。

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

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：

$$0! = 1$$
$$n! = n(n-1)!$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$.

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。

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

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 。

If you can write a recursive definition of something, you can write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

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):
```

If the argument happens to be 0, all we have to do is return 1:

如果实参刚好是 0，我们就返回 1：

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

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$ and then multiply it by n:

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
```

The flow of execution for this program is similar to the flow of `countdown` in Section 5.8. If we call `factorial` with the value 3:

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：

Since 3 is not 0, we take the second branch and calculate the factorial of `n-1`...

由于 3 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...

> Since 2 is not 0, we take the second branch and calculate the factorial of `n-1`...
>
> 由于 2 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...
>
> > Since 1 is not 0, we take the second branch and calculate the factorial of `n-1`...
> >
> > 由于 1 不等于 0，我们执行第二个分支并计算 n-1 的阶乘...
> >
> > > Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.
> > >
> > > 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。
> >
> > The return value, 1, is multiplied by n, which is 1, and the result is returned.
> >
> > 返回值 1 与 n （其为 1）相乘，并返回结果。
>
> The return value, 1, is multiplied by n, which is 2, and the result is returned.
>
> 返回值 1 与 n （其为 2）相乘，并返回结果。

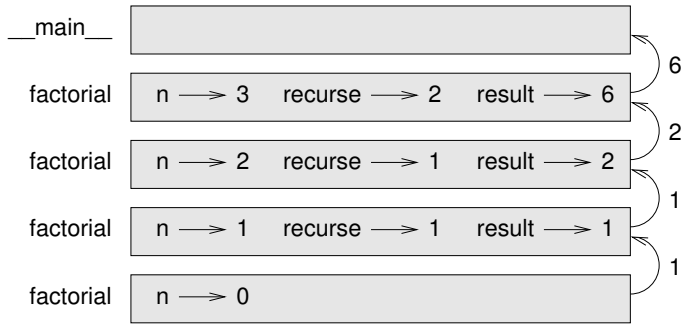

图 6.1: Stack diagram.

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

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。

Figure 6.1 shows what the stack diagram looks like for this sequence of function calls.

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。

The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

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。

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not run.

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 Leap of faith | 信仰之跃

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the "leap of faith". When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

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。

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don't examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

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。

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

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。

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" It is clear that you can, by multiplying by n.

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。

Of course, it's a bit strange to assume that the function works correctly when you haven't finished writing it, but that's why it's called a leap of faith!

当然，在你没写完函数的时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 One more example | 再举一例

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition (see http://en.wikipedia.org/wiki/Fibonacci_number):

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：

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

Translated into Python, it looks like this:

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)
```

If you try to follow the flow of execution here, even for fairly small values of n, your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

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 Checking types | 检查类型

What happens if we call `factorial` and give it 1.5 as an argument?

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
```

It looks like an infinite recursion. How can that be? The function has a base case—when `n == 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

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。

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

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 。

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

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。

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

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)
```

The first base case handles nonintegers; the second handles negative integers. In both cases, the program prints an error message and returns None to indicate that something went wrong:

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。

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

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。

In Section 11.4 we will see a more flexible alternative to printing an error message: raising an exception.

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 Debugging | 调试

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

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：

- There is something wrong with the arguments the function is getting; a precondition is violated.
- 该函数获得的实参有些问题，违反先决条件。
- There is something wrong with the function; a postcondition is violated.
- 该函数有些问题，违反后置条件。
- There is something wrong with the return value or the way it is being used.
- 返回值或者它的使用方法有问题。

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

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。

If the parameters look good, add a `print` statement before each `return` statement and display the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in Section 6.2).

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 一节 一节中那样）。

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

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!）。

Adding print statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with print statements:

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 is a string of space characters that controls the indentation of the output. Here is the result of `factorial(4)`:

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
```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

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 Glossary | 术语表

temporary variable: A variable used to store an intermediate value in a complex calculation.

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

dead code: Part of a program that can never run, often because it appears after a `return` statement.

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。

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

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。

scaffolding: Code that is used during program development but is not part of the final version.

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。

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

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 Exercises | 练习

Exercise 6.1. *Draw a stack diagram for the following program. What does the program print?*

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. *The Ackermann function, $A(m,n)$, is defined:*

Ackermann 函数 $A(m,n)$ 的定义如下：

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

See http://en.wikipedia.org/wiki/Ackermann_function. *Write a function named* `ack` *that evaluates the Ackermann function. Use your function to evaluate* `ack(3, 4)`*, which should be 125. What happens for larger values of* `m` *and* `n`*? Solution:* http://thinkpython2.com/code/ackermann.py.

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. *A palindrome is a word that is spelled the same backward and forward, like "noon" and "redivider". Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.*

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。

The following are functions that take a string argument and return the first, last, and middle letters:

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]
```

We'll see how they work in Chapter 八.

在 第八章 中我们将介绍他们是如何工作的。

1. *Type these functions into a file named* `palindrome.py` *and test them out. What happens if you call* `middle` *with a string with two letters? One letter? What about the empty string, which is written* `''` *and contains no letters?*

2. *Write a function called* `is_palindrome` *that takes a string argument and returns* `True` *if it is a palindrome and* `False` *otherwise. Remember that you can use the built-in function* `len` *to check the length of a string.*

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。

Solution: http://thinkpython2.com/code/palindrome_soln.py.

参考答案

Exercise 6.4. *A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called* `is_power` *that takes parameters* `a` *and* `b` *and returns* `True` *if* `a` *is a power of* `b`*. Note: you will have to think about the base case.*

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. *The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.*

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

One way to find the GCD of two numbers is based on the observation that if r is the remainder when a is divided by b, then $gcd(a,b) = gcd(b,r)$*. As a base case, we can use* $gcd(a,0) = a$*.*

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。

Write a function called `gcd` *that takes parameters* `a` *and* `b` *and returns their greatest common divisor.*

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。

Credit: This exercise is based on an example from Abelson and Sussman's Structure and Interpretation of Computer Programs*.*

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　Iteration

This chapter is about iteration, which is the ability to run a block of statements repeatedly. We saw a kind of iteration, using recursion, in Section 5.8. We saw another kind, using a `for` loop, in Section 4.2. In this chapter we'll see yet another kind, using a `while` statement. But first I want to say a little more about variable assignment.

7.1 Reassignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

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

The first time we display `x`, its value is 5; the second time, its value is 7.

Figure 7.1 shows what **reassignment** looks like in a state diagram.

At this point I want to address a common source of confusion. Because Python uses the equal sign (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a mathematical proposition of equality; that is, the claim that `a` and `b` are equal. But this interpretation is wrong.

First, equality is a symmetric relationship and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not.

Also, in mathematics, a proposition of equality is either true or false for all time. If $a = b$ now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

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

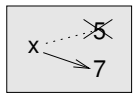

图 7.1: State diagram.

```
>>> b
5
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal.

Reassigning variables is often useful, but you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

7.2 Updating variables

A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

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

This means "get the current value of `x`, add one, and then update `x` with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

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

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

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

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

7.3 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called **iteration**.

We have already seen two functions, `countdown` and `print_n`, that iterate using recursion. Because iteration is so common, Python provides language features to make it easier. One is the `for` statement we saw in Section 4.2. We'll get back to that later.

Another is the `while` statement. Here is a version of `countdown` that uses a `while` statement:

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

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than 0, display the value of `n` and then decrement `n`. When you get to 0, display the word `Blastoff!`"

More formally, here is the flow of execution for a `while` statement:

1. Determine whether the condition is true or false.
2. If false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat", are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates: if `n` is zero or negative, the loop never runs. Otherwise, `n` gets smaller each time through the loop, so eventually we have to get to 0.

For some other loops, it is not so easy to tell. For example:

```
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
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is `1`, which makes the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, `n` is divided by 2. If it is odd, the value of `n` is replaced with `n*3 + 1`. For example, if the argument passed to `sequence` is 3, the resulting values of `n` are 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, `n` will be

even every time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all* positive values of `n`. So far, no one has been able to prove it *or* disprove it! (See http://en.wikipedia.org/wiki/Collatz_conjecture.)

As an exercise, rewrite the function `print_n` from Section 5.8 using iteration instead of recursion.

7.4 break

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

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

print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

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

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

7.5 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

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

For example, if a is 4 and x is 3:

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

The result is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

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

After a few more updates, the estimate is almost exact:

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

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

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

When `y == x`, we can stop. Here is a loop that starts with an initial estimate, `x`, and improves it until it stops changing:

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

For most values of `a` this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`.

Rather than checking whether `x` and `y` are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

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

Where `epsilon` has a value like `0.0000001` that determines how close is close enough.

7.6 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy", you might have learned a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

7.7 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is "debugging by bisection". For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

7.8 Glossary

reassignment: Assigning a new value to a variable that already exists.

update: An assignment where the new value of the variable depends on the old.

initialization: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

decrement: An update that decreases the value of a variable.

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

infinite loop: A loop in which the terminating condition is never satisfied.

algorithm: A general process for solving a category of problems.

7.9 Exercises

Exercise 7.1. *Copy the loop from Section 7.5 and encapsulate it in a function called* `mysqrt` *that takes* `a` *as a parameter, chooses a reasonable value of* `x`*, and returns an estimate of the square root of* `a`*.*

To test it, write a function named `test_square_root` *that prints a table like this:*

```
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
```

The first column is a number, a; the second column is the square root of a computed with `mysqrt`*; the third column is the square root computed by* `math.sqrt`*; the fourth column is the absolute value of the difference between the two estimates.*

Exercise 7.2. *The built-in function* `eval` *takes a string and evaluates it using the Python interpreter. For example:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Write a function called `eval_loop` *that iteratively prompts the user, takes the resulting input and evaluates it using* `eval`*, and prints the result.*

It should continue until the user enters `'done'`*, and then return the value of the last expression it evaluated.*

Exercise 7.3. *The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of* $1/\pi$*:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Write a function called `estimate_pi` *that uses this formula to compute and return an estimate of* π*. It should use a* `while` *loop to compute terms of the summation until the last term is smaller than* `1e-15` *(which is Python notation for* 10^{-15}*). You can check the result by comparing it to* `math.pi`*.*

Solution: http://thinkpython2.com/code/pi.py.

第八章　Strings

Strings are not like integers, floats, and booleans. A string is a **sequence**, which means it is an ordered collection of other values. In this chapter you'll see how to access the characters that make up a string, and you'll learn about some of the methods strings provide.

8.1　A string is a sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`.

The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> letter
'a'
```

For most people, the first letter of `'banana'` is `b`, not `a`. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> letter
'b'
```

So `b` is the 0th letter ("zero-eth") of `'banana'`, `a` is the 1th letter ("one-eth"), and `n` is the 2th letter ("two-eth").

As an index you can use an expression that contains variables and operators:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

But the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

8.3 Traversal with a `for` loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

As an exercise, write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a `for` loop:

```
for letter in fruit:
    print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable `letter`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled. As an exercise, modify the program to fix this error.

8.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in Figure 8.1.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
```

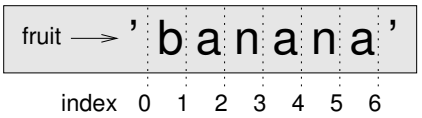

图 8.1: Slice indices.

```
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Continuing this example, what do you think `fruit[:]` means? Try it and see.

8.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10).

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

8.6 Searching

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

In a sense, `find` is the inverse of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

As an exercise, modify `find` so that it has a third parameter, the index in `word` where it should start looking.

8.7 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result—the total number of `a`'s.

As an exercise, encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.

Then rewrite the function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.

8.8 String methods

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters.

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no arguments.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

By default, `find` starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
4
```

This is an example of an **optional argument**; `find` can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2`, not including `2`. Searching up to, but not including, the second index makes `find` consistent with the slice operator.

8.9 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

Here's what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
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 does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

8.11 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two

words and return `True` if one of the words is the reverse of the other, but it contains two errors:

```
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
```

The first `if` statement checks whether the words are the same length. If not, we can return `False` immediately. Otherwise, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

`i` and `j` are indices: `i` traverses `word1` forward while `j` traverses `word2` backward. If we find two letters that don't match, we can return `False` immediately. If we get through the whole loop and all the letters match, we return `True`.

If we test this function with the words "pots" and "stop", we expect the return value `True`, but we get an IndexError:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
    while j > 0:
        print(i, j)         # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

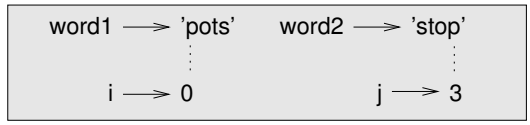

图 8.2: State diagram.

The first time through the loop, the value of `j` is 4, which is out of range for the string `'pots'`. The index of the last character is 3, so the initial value for `j` should be `len(word2)-1`.

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for `is_reverse` is shown in Figure 8.2.

I took some license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2`.

Starting with this diagram, run the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

8.12 Glossary

object: Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

sequence: An ordered collection of values where each value is identified by an integer index.

item: One of the values in a sequence.

index: An integer value used to select an item in a sequence, such as a character in a string. In Python indices start from 0.

slice: A part of a string specified by a range of indices.

empty string: A string with no characters and length 0, represented by two quotation marks.

immutable: The property of a sequence whose items cannot be changed.

traverse: To iterate through the items in a sequence, performing a similar operation on each.

search: A pattern of traversal that stops when it finds what it is looking for.

counter: A variable used to count something, usually initialized to zero and then incremented.

invocation: A statement that calls a method.

optional argument: A function or method argument that is not required.

8.13 Exercises

Exercise 8.1. *Read the documentation of the string methods at* http://docs.python.org/3/library/stdtypes.html#string-methods. *You might want to experiment with some of them to make sure you understand how they work.* `strip` *and* `replace` *are particularly useful.*

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`*, the brackets indicate optional arguments. So* `sub` *is required, but* `start` *is optional, and if you include* `start`*, then* `end` *is optional.*

Exercise 8.2. *There is a string method called* `count` *that is similar to the function in Section 8.7. Read the documentation of this method and write an invocation that counts the number of* `a`*'s in* `'banana'`.

Exercise 8.3. *A string slice can take a third index that specifies the "step size"; that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.*

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` *generates a reversed string.*

Use this idiom to write a one-line version of `is_palindrome` *from Exercise 6.3.*

Exercise 8.4. *The following functions are all* intended *to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).*

```
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. *A Caesar cypher is a weak form of encryption that involves "rotating" each letter by a fixed number of places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' rotated by 3 is 'D' and 'Z' rotated by 1 is 'A'.*

To rotate a word, rotate each letter by the same amount. For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed". In the movie 2001: A Space Odyssey, *the ship computer is called HAL, which is IBM rotated by -1.*

Write a function called `rotate_word` *that takes a string and an integer as parameters, and returns a new string that contains the letters from the original string rotated by the given amount.*

You might want to use the built-in function `ord`, *which converts a character to a numeric code, and* `chr`, *which converts numeric codes to characters. Letters of the alphabet are encoded in alphabetical order, so for example:*

```
>>> ord('c') - ord('a')
2
```

Because `'c'` *is the two-eth letter of the alphabet. But beware: the numeric codes for upper case letters are different.*

Potentially offensive jokes on the Internet are sometimes encoded in ROT13, which is a Caesar cypher with rotation 13. If you are not easily offended, find and decode some of them. Solution: http://thinkpython2.com/code/rotate.py.

第九章　Case study: word play

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.

9.1 Reading word lists

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.

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.

```
>>> 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.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.

The file object keeps track of where it is in the file, so if you call `readline` again, you get the next word:

```
>>> 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`:

```
>>> 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:

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 Exercises

There are solutions to these exercises in the next section. You should at least attempt each one before you read the solutions.

Exercise 9.1. *Write a program that reads* `words.txt` *and prints only the words with more than 20 characters (not counting whitespace).*

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.*

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.

All right, I'll stop now.

Write a function called `has_no_e` *that returns* `True` *if the given word doesn't have the letter "e" in it.*

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".

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.*

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?

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?"*

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`*?*

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?*

9.3 Search

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:

```
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`.

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.

`avoids` is a more general version of `has_no_e` but it has the same structure:

```
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`.

`uses_only` is similar except that the sense of the condition is reversed:

```
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`.

`uses_all` is similar except that we reverse the role of the word and the string of letters:

```
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`.

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:

```
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.

9.4 Looping with indices

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.

For `is_abecedarian` we have to compare adjacent letters, which is a little tricky with a `for` loop:

```
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:

```
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:

```
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 ith character (which you can think of as the current character) to the $i+1$th character (which you can think of as the next).

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`.

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.

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.

```
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:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Using `is_reverse` from Section 8.11.

9.5 Debugging

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.

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.

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.

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).

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

9.6 Glossary

file object: A value that represents an open file.

reduction to a previously solved problem: A way of solving a problem by expressing it as an instance of a previously solved problem.

special case: A test case that is atypical or non-obvious (and less likely to be handled correctly).

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):*

> *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?*

Write a program to find it. Solution: `http://thinkpython2.com/code/cartalk1.py`.

Exercise 9.8. *Here's another* Car Talk *Puzzler (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?"*

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.

Exercise 9.9. *Here's another* Car Talk *Puzzler you can solve with a search (*http://www.cartalk.com/content/puzzlers*):*

> *"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?"*

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.

第十章　Lists

This chapter presents one of Python's most useful built-in types, lists. You will also learn more about objects and what can happen when you have more than one name for the same object.

10.1　A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10.2　Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

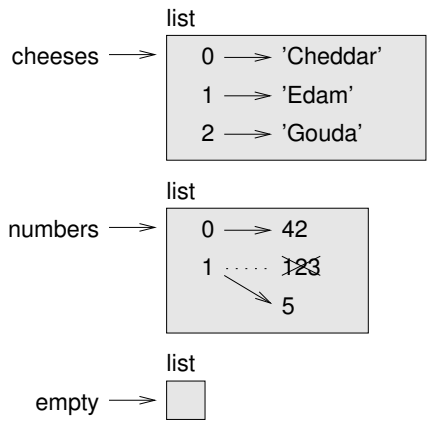

图 10.1: State diagram.

```
>>> cheeses[0]
'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

Figure 10.1 shows the state diagram for `cheeses`, `numbers` and `empty`:

Lists are represented by boxes with the word "list" outside and the elements of the list inside. `cheeses` refers to a list with three elements indexed 0, 1 and 2. `numbers` contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. `empty` refers to a list with no elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never runs the body:

```
for x in []:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1, 2, 3]` three times.

10.5 List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

10.7 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` is initialized to 0. Each time through the loop, `x` gets one element from the list. The `+=` operator provides a short way to update a variable. This **augmented assignment statement**,

```
    total += x
```

is equivalent to

```
    total = total + x
```

As the loop runs, `total` accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` is initialized with an empty list; each time through the loop, we append the next element. So `res` is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it "maps" a function (in this case the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
```

```
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` is a string method that returns `True` if the string contains only upper case letters.

An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce.

10.8 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.

10.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

10.10 Objects and values

If we run these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states, shown in Figure 10.2.

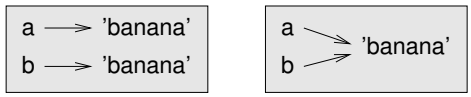

图 10.2: State diagram.

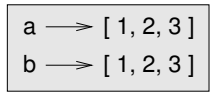

图 10.3: State diagram.

In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both `a` and `b` refer to it. But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

So the state diagram looks like Figure 10.3.

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you evaluate `[1, 2, 3]`, you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

10.11 Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

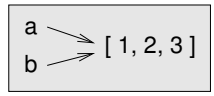

图 10.4: State diagram.

The state diagram looks like Figure 10.4.

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

10.12 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object. The stack diagram looks like Figure 10.5.

Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

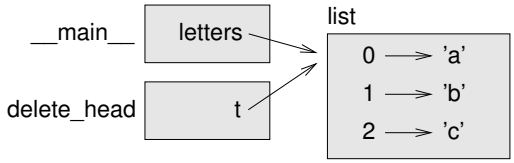

图 10.5: Stack diagram.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

`append` modifies the list and returns `None`.

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

The `+` operator creates a new list and leaves the original list unchanged.

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but that doesn't affect the caller.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

At the beginning of `bad_delete_head`, `t` and `t4` refer to the same list. At the end, `t` refers to a new list, but `t4` still refers to the original, unmodified list.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this:

   ```
   t = t.sort()            # WRONG!
   ```

   Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

   Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. Assuming that `t` is a list and `x` is a list element, these are correct:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   And these are wrong:

   ```
   t.append([x])           # WRONG!
   t = t.append(x)         # WRONG!
   t + [x]                 # WRONG!
   t = t + x               # WRONG!
   ```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone.

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14 Glossary

list: A sequence of values.

element: One of the values in a list (or other sequence), also called items.

nested list: A list that is an element of another list.

accumulator: A variable used in a loop to add up or accumulate a result.

augmented assignment: A statement that updates the value of a variable using an operator like `+=`.

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

object: Something a variable can refer to. An object has a type and a value.

equivalent: Having the same value.

identical: Being the same object (which implies equivalence).

reference: The association between a variable and its value.

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

10.15 Exercises

You can download solutions to these exercises from http://thinkpython2.com/code/list_exercises.py.

Exercise 10.1. *Write a function called* `nested_sum` *that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercise 10.2. *Write a function called* `cumsum` *that takes a list of numbers and returns the cumulative sum; that is, a new list where the ith element is the sum of the first* $i + 1$ *elements from the original list. For example:*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Exercise 10.3. *Write a function called* `middle` *that takes a list and returns a new list that contains all but the first and last elements. For example:*

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercise 10.4. *Write a function called* `chop` *that takes a list, modifies it by removing the first and last elements, and returns* `None`*. For example:*

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 10.5. *Write a function called* `is_sorted` *that takes a list as a parameter and returns* `True` *if the list is sorted in ascending order and* `False` *otherwise. For example:*

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 10.6. *Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called* `is_anagram` *that takes two strings and returns* `True` *if they are anagrams.*

Exercise 10.7. *Write a function called* `has_duplicates` *that takes a list and returns* `True` *if there is any element that appears more than once. It should not modify the original list.*

Exercise 10.8. *This exercise pertains to the so-called Birthday Paradox, which you can read about at http://en.wikipedia.org/wiki/Birthday_paradox.*

If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `randint` *function in the* `random` *module.*

You can download my solution from http://thinkpython2.com/code/birthday.py.

Exercise 10.9. *Write a function that reads the file* `words.txt` *and builds a list with one element per word. Write two versions of this function, one using the* `append` *method and the other using the idiom* `t = t + [x]`*. Which one takes longer to run? Why?*

Solution: http://thinkpython2.com/code/wordlist.py.

Exercise 10.10. *To check whether a word is in the word list, you could use the* `in` *operator, but it would be slow because it searches through the words in order.*

Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary. You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, you search the first half of the list the same way. Otherwise you search the second half.

Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called `in_bisect` *that takes a sorted list and a target value and returns the index of the value in the list if it's there, or* `None` *if it's not.*

Or you could read the documentation of the `bisect` *module and use that! Solution:* http://thinkpython2.com/code/inlist.py.

Exercise 10.11. *Two words are a "reverse pair" if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list. Solution:* http://thinkpython2.com/code/reverse_pair.py.

Exercise 10.12. *Two words "interlock" if taking alternating letters from each forms a new word. For example, "shoe" and "cold" interlock to form "schooled". Solution:* http://thinkpython2.com/code/interlock.py. *Credit: This exercise is inspired by an example at* http://puzzlers.org.

1. *Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!*
2. *Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?*

第十一章　Dictionaries

This chapter presents another built-in type called a dictionary. Dictionaries are one of Python's best features; they are the building blocks of many efficient and elegant algorithms.

11.1 A dictionary is a mapping

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key "maps to" a value. As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

The squiggly-brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `'uno'`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> eng2sp['two']
'dos'
```

The key 'two' always maps to the value 'dos' so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> eng2sp['four']
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The in operator works on dictionaries, too; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order, as in Section 8.6. As the list gets longer, the search time gets longer in direct proportion.

For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary. I explain how that's possible in Section B.4, but the explanation might not make sense until you've read a few more chapters.

11.2 Dictionary as a collection of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

The name of the function is `histogram`, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters `'a'` and `'b'` appear once; `'o'` appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

As an exercise, use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement.

11.3 Looping and dictionaries

If you use a dictionary in a `for` statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function `sorted`:

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4 Reverse lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**.

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application,

you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The **raise statement** causes an exception; in this case it causes a `LookupError`, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means `v` doesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

And an unsuccessful one:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

The `raise` statement can take a detailed error message as an optional argument. For example:

```
>>> 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
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

11.5 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters

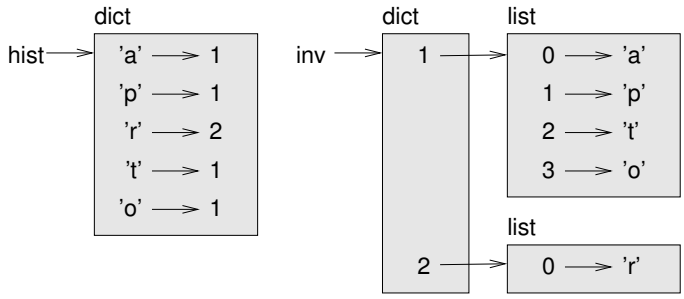

图 11.1: State diagram.

with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
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
```

Each time through the loop, `key` gets a key from `d` and `val` gets the corresponding value. If `val` is not in `inverse`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> 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']}
```

Figure 11.1 is a state diagram showing `hist` and `inverse`. A dictionary is represented as a box with the type `dict` above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
```

```
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter.

Since dictionaries are mutable, they can't be used as keys, but they *can* be used as values.

11.6 Memos

If you played with the `fibonacci` function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly.

To understand why, consider Figure 11.2, which shows the **call graph** for `fibonacci` with `n=4`:

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a "memoized" version of `fibonacci`:

```
known = {0:0, 1:1}
```

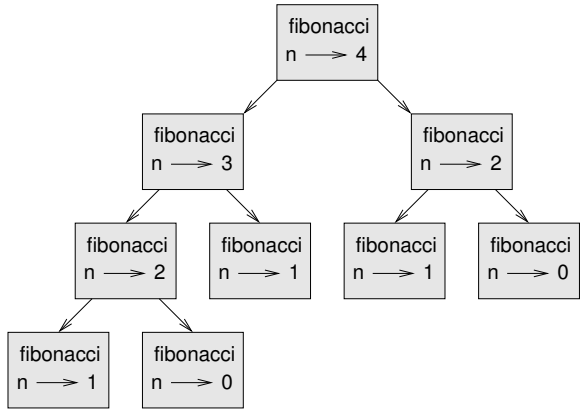

图 11.2: Call graph.

```
def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

`known` is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever `fibonacci` is called, it checks `known`. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

If you run this version of `fibonacci` and compare it with the original, you will find that it is much faster.

11.7 Global variables

In the previous example, `known` is created outside the function, so it belongs to the special frame called `__main__`. Variables in `__main__` are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for **flags**; that is, boolean variables that indicate ("flag") whether a condition is true. For example, some programs use a flag named `verbose` to control the level of detail in the output:

```
verbose = True

def example1():
```

```
    if verbose:
        print('Running example1')
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

```
been_called = False

def example2():
    been_called = True          # WRONG
```

But if you run it you will see that the value of `been_called` doesn't change. The problem is that `example2` creates a new local variable named `been_called`. The local variable goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to **declare** the global variable before you use it:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

The **global statement** tells the interpreter something like, "In this function, when I say `been_called`, I mean the global variable; don't create a local one."

Here's an example that tries to update a global variable:

```
count = 0

def example3():
    count = count + 1           # WRONG
```

If you run it you get:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python assumes that `count` is local, and under that assumption you are reading it before writing it. The solution, again, is to declare `count` global.

```
def example3():
    global count
    count += 1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

```
def example5():
    global known
    known = dict()
```

Global variables can be useful, but if you have a lot of them, and you modify them frequently, they can make programs hard to debug.

11.8 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking the output by hand. Here are some suggestions for debugging large datasets:

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines.

If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "insane".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

Format the output: Formatting debugging output can make it easier to spot an error. We saw an example in Section 6.9. The `pprint` module provides a `pprint` function that displays built-in types in a more human-readable format (`pprint` stands for "pretty print").

Again, time you spend building scaffolding can reduce the time you spend debugging.

11.9 Glossary

mapping: A relationship in which each element of one set corresponds to an element of another set.

dictionary: A mapping from keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: In a dictionary, another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value".

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

hashable: A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

lookup: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

raise statement: A statement that (deliberately) raises an exception.

singleton: A list (or other sequence) with a single element.

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

memo: A computed value stored to avoid unnecessary future computation.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

global statement: A statement that declares a variable name global.

flag: A boolean variable used to indicate whether a condition is true.

declaration: A statement like `global` that tells the interpreter something about a variable.

11.10 Exercises

Exercise 11.1. *Write a function that reads the words in* `words.txt` *and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the* `in` *operator as a fast way to check whether a string is in the dictionary.*

If you did Exercise 10.10, you can compare the speed of this implementation with the list `in` *operator and the bisection search.*

Exercise 11.2. *Read the documentation of the dictionary method* `setdefault` *and use it to write a more concise version of* `invert_dict`*. Solution:* `http://thinkpython2.com/code/invert_dict.py`.

Exercise 11.3. *Memoize the Ackermann function from Exercise 6.2 and see if memoization makes it possible to evaluate the function with bigger arguments. Hint: no. Solution:* http://thinkpython2.com/code/ackermann_memo.py.

Exercise 11.4. *If you did Exercise 10.7, you already have a function named* `has_duplicates` *that takes a list as a parameter and returns* `True` *if there is any object that appears more than once in the list.*

Use a dictionary to write a faster, simpler version of `has_duplicates`*. Solution:* http://thinkpython2.com/code/has_duplicates.py.

Exercise 11.5. *Two words are "rotate pairs" if you can rotate one of them and get the other (see* `rotate_word` *in Exercise 8.5).*

Write a program that reads a wordlist and finds all the rotate pairs. Solution: http://thinkpython2.com/code/rotate_pairs.py.

Exercise 11.6. *Here's another Puzzler from* Car Talk *(*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?*

You can use the dictionary from Exercise 11.1 to check whether a string is in the word list.

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from http://www.speech.cs.cmu.edu/cgi-bin/cmudict *or from* http://thinkpython2.com/code/c06d *and you can also download* http://thinkpython2.com/code/pronounce.py*, which provides a function named* `read_dictionary` *that reads the pronouncing dictionary and returns a Python dictionary that maps from each word to a string that describes its primary pronunciation.*

Write a program that lists all the words that solve the Puzzler. Solution: http://thinkpython2.com/code/homophone.py.

第十二章　Tuples | 元组

This chapter presents one more built-in type, the tuple, and then shows how lists, dictionaries, and tuples work together. I also present a useful feature for variable-length argument lists, the gather and scatter operators.

One note: there is no consensus on how to pronounce "tuple". Some people say "tuh-ple", which rhymes with "supple". But in the context of programming, most people say "too-ple", which rhymes with "quadruple".

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　Tuples are immutable | 元组的不可变性

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

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。

Syntactically, a tuple is a comma-separated list of values:

语法上, 元组是用逗号隔开的值的列表：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

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')
```

[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"。

To create a tuple with a single element, you have to include a final comma:

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'>
```

A value in parentheses is not a 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'>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

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。

```
>>> t = tuple()
>>> t
()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

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')
```

Because `tuple` is the name of a built-in function, you should avoid using it as a variable name.

因为`tuple`是内建函数名，所以应该避免将它用于变量名。

Most list operators also work on tuples. The bracket operator indexes an element:

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'
```

And the slice operator selects a range of elements.

切片操作可以选取一个范围内的元素:

```
>>> t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

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
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

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')
```

This statement makes a new tuple and then makes `t` refer to it.

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`。

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

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 Tuple assignment | 元组赋值

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

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
```

This solution is cumbersome; **tuple assignment** is more elegant:

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
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.

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。

The number of variables on the left and the number of values on the right have to be the same:

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
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

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('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

`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 Tuples as return values | 元组作为返回值

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x/y` and then `x%y`. It is better to compute them both at the same time.

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。

The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

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)
```

Or use tuple assignment to store the elements separately:

或者使用元组赋值分别存储它们：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Here is an example of a function that returns a tuple:

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 and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

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 Variable-length argument tuples | 可变长度参数元组

Functions can take a variable number of arguments. A parameter name that begins with * **gathers** arguments into a tuple. For example, printall takes any number of arguments and prints them:

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)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

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')
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the `*` operator. For example, `divmod` takes exactly two arguments; it doesn't work with a tuple:

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
```

But if you scatter the tuple, it works:

但是如果您将这个元组打散，它就可以被传递进函数：

```
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

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
```

But `sum` does not.

但是求和操作`sum`并不如此：

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called `sumall` that takes any number of arguments and returns their sum.

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 Lists and tuples | 列表和元组

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence. The name of the function refers to a zipper, which joins and interleaves two rows of teeth.

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。

This example zips a string and a list:

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>
```

The result is a **zip object** that knows how to iterate through the pairs. The most common use of zip is in a for loop:

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)
```

A zip object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

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。

If you want to use list operators and methods, you can use a zip object to make a list:

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)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

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。

If the sequences are not the same length, the result has the length of the shorter one.

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')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

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)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`. The output of this loop is:

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
```

If you combine `zip`, `for` and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

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
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

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)
```

The result from `enumerate` is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

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
```

Again.

12.6 Dictionaries and tuples | 字典和元组

Dictionaries have a method called `items` that returns a sequence of tuples, where each tuple is a key-value pair.

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)])
```

The result is a `dict_items` object, which is an iterator that iterates the key-value pairs. You can use it in a `for` loop like this:

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:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

As you should expect from a dictionary, the items are in no particular order.

和字典对象相似，dict_itmes内部的项是无序存放的。

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

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}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

dict和zip的结合使用产生了一个简洁的字典生成法:

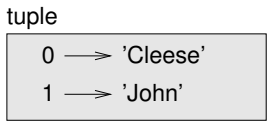

图 12.1: State diagram.

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

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。

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

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
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

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])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

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。

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple `('Cleese', 'John')` would appear as in Figure 12.1.

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。

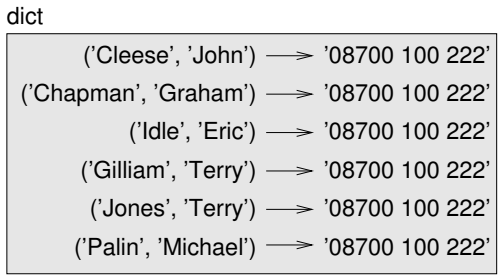

图 12.2: State diagram.

But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure 12.2.

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。

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

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 Sequences of sequences | 序列嵌套

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

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。

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

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？

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

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。

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

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. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

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。

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in function `sorted`, which takes any sequence and returns a new list with the same elements in sorted order, and `reversed`, which takes a sequence and returns an iterator that traverses the list in reverse order.

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 Debugging | 调试

Lists, dictionaries and tuples are examples of **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, or dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size, or structure. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

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。

To help debug these kinds of errors, I have written a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as

an argument and returns a string that summarizes its shape. You can download it from http://thinkpython2.com/code/structshape.py

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)。

Here's the result for a simple list:

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'
```

A fancier program might write "list of 3 ints", but it was easier not to deal with plurals. Here's a list of lists:

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'
```

If the elements of the list are not the same type, structshape groups them, in order, by type:

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)'
```

Here's a list of tuples:

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)'
```

And here's a dictionary with 3 items that map integers to strings.

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'
```

If you are having trouble keeping track of your data structures, `structshape` can help.

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 Glossary | 术语表

tuple: An immutable sequence of elements.

元组： 一组不可变的元素的序列。

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

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。

gather: The operation of assembling a variable-length argument tuple.

汇集： 组装可变长度变量元组的一种操作。

scatter: The operation of treating a sequence as a list of arguments.

分散： 将一个序列变换成一个参数列表的操作。

zip object: The result of calling a built-in function `zip`; an object that iterates through a sequence of tuples.

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。

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

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。

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

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。

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.

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 Exercises | 练习

Exercise 12.1. *Write a function called* `most_frequent` *that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at* http://en.wikipedia.org/wiki/Letter_frequencies. *Solution:* http://thinkpython2.com/code/most_frequent.py.

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上字母频率表相比较。

参考答案

Exercise 12.2. *More anagrams!*

1. *Write a program that reads a word list from a file (see Section 9.1) and prints all the sets of words that are anagrams.*

   Here is an example of what the output might look like:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Hint: you might want to build a dictionary that maps from a collection of letters to a list of words that can be spelled with those letters. The question is, how can you represent the collection of letters in a way that can be used as a key?

2. *Modify the previous program so that it prints the longest list of anagrams first, followed by the second longest, and so on.*

3. *In Scrabble a "bingo" is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What collection of 8 letters forms the most possible bingos? Hint: there are seven.*

   Solution: http://thinkpython2.com/code/anagram_sets.py.

   易位构词游戏 (*anagrams*)!

4. 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。

5. 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。

6. *Scrabble* 游戏 中，“*bingo*”...

   参考答案

Exercise 12.3. *Two words form a “metathesis pair” if you can transform one into the other by swapping two letters; for example, “converse” and “conserve”. Write a program that finds all of the metathesis pairs in the dictionary. Hint: don’t test all pairs of words, and don’t test all possible swaps. Solution:* `http://thinkpython2.com/code/metathesis.py`. *Credit: This exercise is inspired by an example at* `http://puzzlers.org`.

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. *Here’s another Car Talk Puzzler (*`http://www.cartalk.com/content/puzzlers`*):*

> *What is the longest English word, that remains a valid English word, as you remove its letters one at a time?*
>
> *Now, letters can be removed from either end, or the middle, but you can’t rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you’re eventually going to wind up with one letter and that too is going to be an English word—one that’s found in the dictionary. I want to know what’s the longest word and how many letters does it have?*
>
> *I’m going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we’re left with the word spite, then we take the e off the end, we’re left with spit, we take the s off, we’re left with pit, it, and I.*

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*。

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. *You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.*
2. *Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.*
3. *The wordlist I provided,* `words.txt`*, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.*
4. *To improve the performance of your program, you might want to memoize the words that are known to be reducible.*

Solution: `http://thinkpython2.com/code/reducible.py`.

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

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.

This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

13.1 Word frequency analysis

As usual, you should at least attempt the exercises before you read my solutions.
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.*

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:*

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Also, you might consider using the string methods `strip`*,* `replace` *and* `translate`*.*
Exercise 13.2. *Go to Project Gutenberg (*`http://gutenberg.org`*) and download your favorite out-of-copyright book in plain text format.*

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.

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?
Exercise 13.3. *Modify the program from the previous exercise to print the 20 most frequently used words in the book.*

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?*

13.2 Random numbers

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.

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.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

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:

```
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).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> 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.

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:*

```
>>> 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* $1/3$.

13.3 Word histogram

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:

```
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.

`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_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.)

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

And the results:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words

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:

```
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:

```
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*:

```
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.

13.5 Optional parameters

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

```
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.

If you only provide one argument:

```
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.

13.6 Dictionary subtraction

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).

`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.

```
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:

```
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*:

```
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!

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`.

13.7 Random words

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:

```
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.

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:

1. Use `keys` to get a list of the words in the book.
2. 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.

3. 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.

4. Use the index to find the corresponding word in the word list.

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`.

13.8 Markov analysis

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:

```
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.

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:

> 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".

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").

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.

Exercise 13.8. *Markov analysis:*

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. *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?

3. *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.*

Credit: This case study is based on an example from Kernighan and Pike, The Practice of Programming, *Addison-Wesley, 1999.*

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`.

13.9 Data structures

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:

- 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.

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

For the suffixes, one option is a list; another is a histogram (dictionary).

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".

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:

```
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.

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.

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).

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.

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.

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.

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.

13.10 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

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.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

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.

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.

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.

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.

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.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

13.11 Glossary

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom: Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

default value: The value given to an optional parameter if no argument is provided.

override: To replace a default value with an argument.

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

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.

13.12 Exercises

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.*

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:

$$f = cr^{-s}$$

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:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

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?

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.*

第十四章　Files

This chapter introduces the idea of "persistent" programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2 Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file. The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

14.3 Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value `42`.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

For more information on the format operator, see https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting. A more powerful alternative is the string format method, which you can read about at https://docs.python.org/3/library/stdtypes.html#str.format.

14.4 Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` stands for "current working directory". The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.

A string like `'/home/dinsdale'` that identifies a file or directory is called a **path**.

A simple filename, like `memo.txt` is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is `/home/dinsdale`, the filename `memo.txt` would refer to `/home/dinsdale/memo.txt`.

A path that begins with `/` does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` provides other functions for working with filenames and paths. For example, `os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
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` takes a directory and a file name and joins them into a complete path.

The `os` module provides a function called `walk` that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from http://thinkpython2.com/code/walk.py.

14.5 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does. The syntax is similar to an `if...else` statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

14.6 Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like keys and items, don't work with database objects. But iteration with a for loop works:

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7 Pickling

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to interpret. pickle.loads ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called shelve.

14.8 Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications.

For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen`[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a "checksum". You can read about MD5 at http://en.wikipedia.org/wiki/Md5. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> 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 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

[1] `popen` is deprecated now, which means we are supposed to stop using it and start using the `subprocess` module. But for simple cases, I find `subprocess` more complicated than necessary. So I am going to keep using `popen` until they take it away.

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

14.10 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at http://en.wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

14.11 Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

14.12 Exercises

Exercise 14.1. *Write a function called* `sed` *that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.*

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution: `http://thinkpython2.com/code/sed.py`.

Exercise 14.2. *If you download my solution to Exercise 12.2 from* `http://thinkpython2.com/code/anagram_sets.py`*, you'll see that it creates 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']`.

Write a module that imports `anagram_sets` *and provides two new functions:* `store_anagrams` *should store the anagram dictionary in a "shelf";* `read_anagrams` *should look up a word and return a list of its anagrams. Solution:* `http://thinkpython2.com/code/anagram_db.py`.

Exercise 14.3. *In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.*

1. *Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like* `.mp3`*). Hint:* `os.path` *provides several useful functions for manipulating file and path names.*

2. *To recognize duplicates, you can use* `md5sum` *to compute a "checksum" for each files. If two files have the same checksum, they probably have the same contents.*

3. *To double-check, you can use the Unix command* `diff`.

Solution: `http://thinkpython2.com/code/find_duplicates.py`.

第十五章　Classes and objects

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn "object-oriented programming", which uses programmer-defined types to organize both code and data. Object-oriented programming is a big topic; it will take a few chapters to get there.

Code examples from this chapter are available from http://thinkpython2.com/code/Point1.py; solutions to the exercises are available from http://thinkpython2.com/code/Point1_soln.py.

15.1 Programmer-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this:

```
class Point:
    """Represents a point in 2-D space."""
```

The header indicates that the new class is called `Point`. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a **class object**.

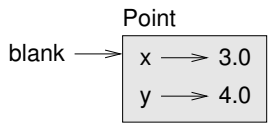

图 15.1: Object diagram.

```
>>> Point
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

Every object is an instance of some class, so "object" and "instance" are interchangeable. But in this chapter I use "instance" to indicate that I am talking about a programmer-defined type.

15.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**; see Figure 15.1.

The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In the example, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

As an exercise, write a function called `distance_between_points` that takes two Points as arguments and returns the distance between them.

15.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

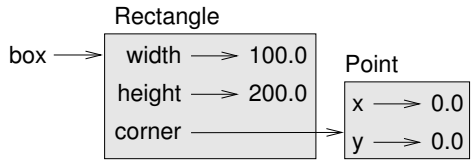

图 15.2: Object diagram.

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Here is an example that passes `box` as an argument and assigns the resulting Point to `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, `rect` is an alias for `box`, so when the function modifies `rect`, `box` changes.

As an exercise, write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.

15.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
```

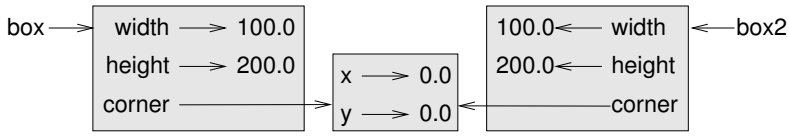

图 15.3: Object diagram.

```
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module provides a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

As an exercise, write a version of `move_rectangle` that creates and returns a new Rectangle instead of modifying the old one.

15.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<class '__main__.Point'>
```

You can also use `isinstance` to check whether an object is an instance of a class:

```
>>> isinstance(p, Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

You can also use a `try` statement to see if the object has the attributes you need:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

15.8 Glossary

class: A programmer-defined type. A class definition creates a new class object.

class object: An object that contains information about a programmer-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

instantiate: To create a new object.

attribute: One of the named values associated with an object.

embedded object: An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

15.9 Exercises

Exercise 15.1. *Write a definition for a class named* `Circle` *with attributes* `center` *and* `radius`*, where* `center` *is a Point object and radius is a number.*

Instantiate a Circle object that represents a circle with its center at $(150, 100)$ *and radius 75.*

Write a function named `point_in_circle` *that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.*

Write a function named `rect_in_circle` *that takes a Circle and a Rectangle and returns True if the Rectangle lies entirely in or on the boundary of the circle.*

Write a function named `rect_circle_overlap` *that takes a Circle and a Rectangle and returns True if any of the corners of the Rectangle fall inside the circle. Or as a more challenging version, return True if any part of the Rectangle falls inside the circle.*

Solution: `http://thinkpython2.com/code/Circle.py`.

Exercise 15.2. *Write a function called* `draw_rect` *that takes a Turtle object and a Rectangle and uses the Turtle to draw the Rectangle. See Chapter 四 for examples using Turtle objects.*

Write a function called `draw_circle` *that takes a Turtle and a Circle and draws the Circle.*

Solution: `http://thinkpython2.com/code/draw.py`.

第十六章　Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present "functional programming style" and two new program development plans.

Code examples from this chapter are available from http://thinkpython2.com/code/Time1.py. Solutions to the exercises are at http://thinkpython2.com/code/Time1_soln.py.

16.1 Time

As another example of a programmer-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

The state diagram for the `Time` object looks like Figure 16.1.

As an exercise, write a function called `print_time` that takes a Time object and prints it in the form `hour:minute:second`. Hint: the format sequence `'%.2d'` prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called `is_after` that takes two Time objects, `t1` and `t2`, and returns `True` if `t1` follows `t2` chronologically and `False` otherwise. Challenge: don't use an `if` statement.

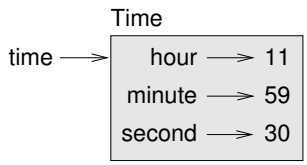

图 16.1: Object diagram.

16.2 Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```
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
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: `start` contains the start time of a movie, like *Monty Python and the Holy Grail*, and `duration` contains the run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> 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
```

The result, `10:80:00` might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
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
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

16.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
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
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if `seconds` is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of `increment` that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

As an exercise, write a "pure" version of `increment` that creates and returns a new Time object rather than modifying the parameter.

16.4 Prototyping versus planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see http://en.wikipedia.org/wiki/Sexagesimal.)! The `second` attribute is the "ones column", the `minute` attribute is the "sixties column", and the `hour` attribute is the "thirty-six hundreds column".

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that `time_to_int(int_to_time(x)) == x` for many values of `x`. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite `increment` using `time_to_int` and `int_to_time`.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

16.5 Debugging

A Time object is well-formed if the values of `minute` and `second` are between 0 and 60 (including 0 but not 60) and if `hour` is positive. `hour` and `minute` should be integral values, but we might allow `second` to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like `valid_time` that takes a Time object and returns `False` if it violates an invariant:

```
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
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
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)
```

Or you could use an **assert statement**, which checks a given invariant and raises an exception if it fails:

```
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` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

16.6 Glossary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return `None`.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

assert statement: A statement that check a condition and raises an exception if it fails.

16.7 Exercises

Code examples from this chapter are available from http://thinkpython2.com/code/Time1.py; solutions to the exercises are available from http://thinkpython2.com/code/Time1_soln.py.

Exercise 16.1. *Write a function called* `mul_time` *that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.*

Then use `mul_time` *to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).*

Exercise 16.2. *The* `datetime` *module provides* `time` *objects that are similar to the Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at http://docs.python.org/3/library/datetime.html.*

1. *Use the* `datetime` *module to write a program that gets the current date and prints the day of the week.*

2. *Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.*

3. *For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birthdays and computes their Double Day.*

4. *For a little more challenge, write the more general version that computes the day when one person is n times older than the other.*

Solution: http://thinkpython2.com/code/double.py

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 16.4) 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 16.3) 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.

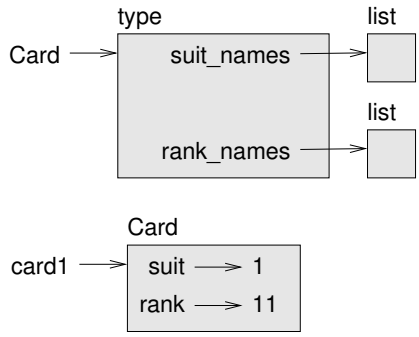

图 18.1: Object diagram.

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.

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

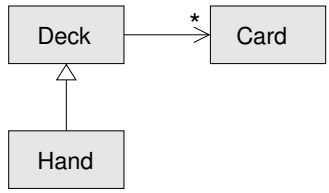

图 18.2: Class diagram.

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，其研究算法的性能，特别是他们运行的时间和对资源空间的需求。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 | 增长的阶数

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，我们希望一个算法有一个较小的首项，使得对于大的问题其是一个好算法，但是对于小问题，可能有一个交叉点 (crossover point)，在此另一个算法更好。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逐渐增大的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

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)

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 log 项，log 的基数没有什么关系。改变阶数等价于乘以一个常数，其不改变增长阶数。简单来讲，如果不考虑指数的基数，指数函数属于相同的增长结束。指数函数增长的非常快，因此指数级算法只对于小问题有用。

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 标记 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.

大部分算术运算是常数时间的。乘法通常比加减法用更长的时间，除法甚至更长，但是这些运行时间不依赖运算数的数量级。非常大的整数是个例外，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.

索引运算－在序列或字典中读或写元素也是常数时间，不考虑数据结构的大小。

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。用算法分析的语言讲，它有更小的首项系数。

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.

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.

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。字符串方法

String concatenation is linear; the run time depends on the sum of the lengths of the operands.

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—它们被认为是常数时间。

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—它们被认为是常数时间 —

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:*

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. *What is the order of growth of bubble sort, and why does Barack Obama think it is "the wrong way to go?"*
3. *What is the order of growth of radix sort? What preconditions do we need to use it?*
4. *What is a stable sort and why might it matter in practice?*
5. *What is the worst sorting algorithm (that has a name)?*
6. *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.*
7. *Many of the non-comparison sorts are linear, so why does does Python use an $O(n \log n)$ 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.

最简单的搜素算法是 ''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.

如果训练是排好序的，你可以用二分搜索 (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（真正的字典，不是数据结构）。不是从头开始并按顺序检查每个项，你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.

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.

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 来演示这个实现案例，事实上在真实情况下，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.

目前，我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 `log` 时间内实现 `add` 和 `get`，但是这仍然不如常数时间好，因此让我们继续。

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:

一种实现 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，其是 100 个 LinearMaps 的列表。正如一会我们要看到的，get 的增长阶数仍然是线性的，但是 BetterMap 是迈向哈希表的一步。

```
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.

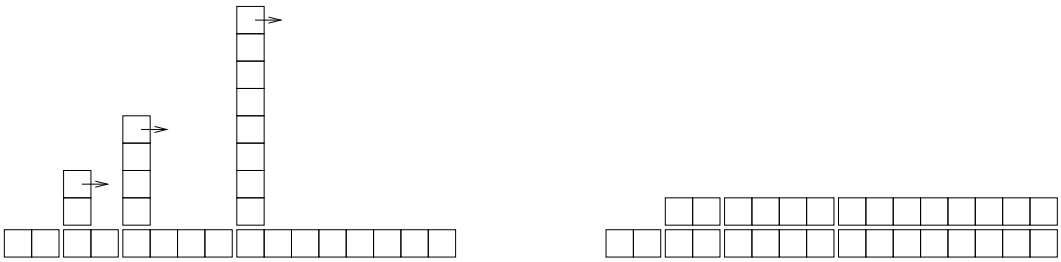

图 B.1: The cost of a hashtable add.

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 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.

machine model: A simplified representation of a computer used to describe algorithms.

worst case: The input that makes a given algorithm run slowest (or require the most space.

leading term: In a polynomial, the term with the highest exponent.

crossover point: The problem size where two algorithms require the same run time or space.

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.

Big-Oh notation: Notation for representing an order of growth; for example, $O(n)$ represents the set of functions that grow linearly.

linear: An algorithm whose run time is proportional to problem size, at least for large problem sizes.

quadratic: An algorithm whose run time is proportional to n^2, where n is a measure of problem size.

search: The problem of locating an element of a collection (like a list or dictionary) or determining that it is not present.

hashtable: A data structure that represents a collection of key-value pairs and performs search in constant time.

索引