Think Python

How to Think Like a Computer Scientist

如何像计算机科学家一样思考

Version 2.0.5

December 2012

Think Python

How to Think Like a Computer Scientist

Version 2.0.5

December 2012

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.thinkpython.com`

Preface 前言

The strange history of this book 本书与众不同的历史

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in the class was too high and, even for students who succeeded, the overall level of achievement was too low.

1999年1月，我正准备使用Java教一门编程入门课程。 我之前已经教了三次，并且感到很沮丧。 课程的不及格率太高，即使对于及格的学生，整体的收获也太低。

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

我看到的问题之一是教材。 它们都太大大了，有太多不必要的关于Java的细节， 并且缺乏关于如何编程的上层的指导。 同时，它们也缺乏陷阱门效应（trap door effect），即从简单之处开始，并逐渐深入。 因此，大概到第五章左右，后面的学生开始掉队了。 学生们获得太多的材料，进展太快，并且我在剩下的学期选择一些片段。

Two weeks before the first day of classes, I decided to write my own book. My goals were:

在开始上课前两周，我决定写一本自己的书。 我的目标是：

- Keep it short. It is better for students to read 10 pages than not read 50 pages.

  尽量简短。对于学生，读10页比读50页好。

- Be careful with vocabulary. I tried to minimize the jargon and define each term at first use.

  关注术语表。我试图使用最少的术语并且在第一次使用的时候给出定义。

- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.

  循序渐进。为了避免陷阱门，我将最难的主题拆分成一系列小的步骤。

- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

  聚焦于编程，而不是编程语言。我只包括了Java最小的有用的子集， 并且忽略其余的。

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

我需要一个标题，所以一时兴起，我选择了《如何像计算机科学家一样思考》。

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

第一版很粗糙，但是它有效。学生们读它，并且他们足够理解它。 然后我可以将上课时间花在那些难的、有趣的主题上， 并且让学生们实践（这点最重要）。

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

我将此书以GNU免费文档许可的形式发布， 该许可允许用户拷贝、修改并且发布此书。

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book. As Green Tea Press, I published the first Python version in 2001.

接下来发生的非常棒。 弗吉尼亚一所高中的教师，Jeff Elkne修改了我的书并且翻译成了Python。 他将他的翻译发给了我一份，我有了一个与众不同的学习Python的体验， 即通过读我自己的书。 2001年，通过绿茶出版社（Green Tea Press），我发表了本书的第一个Python版本。

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

2003年，我开始在Olin College教书，并且第一次教Python。 与教Java的对比很显著，学生需要付出的努力更少，学到的更多， 他们致力于更有趣的项目，并且通常更快乐。

Over the last nine years I continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises.

过去9年我一直在改进这本书，纠正一些错误，改进一些实例， 增加一些材料，特别是练习题。

The result is this book, now with the less grandiose title *Think Python*. Some of the changes are:

本书的结果是，现在有了一个更具体的名字《Think Python》。 下面是一些变化：

- I added a section about debugging at the end of each chapter. These sections present general techniques for finding and avoiding bugs, and warnings about Python pitfalls.

  每章最后，我都增加了一节关于调试（debugging）的内容。 这些小节给出关于发现和避免一些错误的通常的技术， 以及关于Python的陷阱。

- I added more exercises, ranging from short tests of understanding to a few substantial projects. And I wrote solutions for most of them.

  增加了更多的练习题，涵盖了从为了理解的简短的测试到真正的项目。 并且为其中大部分写了答案。

- I added a series of case studies—longer examples with exercises, solutions, and discussion. Some are based on Swampy, a suite of Python programs I wrote for use in my classes. Swampy, code examples, and some solutions are available from `http://thinkpython.com`.

  我增加了一系列范例分析（case studies），即具有联系、答案以及讨论的更长的样例。其中一些基于Swampy，我写的一套用于我的课程的Pythong程序。Swampy、代码样例、以及一些答案可以从 `http://thinkpython.com` 获得。

- I expanded the discussion of program development plans and basic design patterns.

  我扩展了关于程序开发规划和基本设计模式的讨论。

- I added appendices about debugging, analysis of algorithms, and UML diagrams with Lumpy.

  我增加了关于调试、算法分析以及使用Lumpy画UML图的附录。

I hope you enjoy working with this book, and that it helps you learn to program and think, at least a little bit, like a computer scientist.

我希望你能使用该书愉快的工作，并且它能帮助你学习编程和思考，至少或多或少像一个计算机科学家。

Allen B. Downey
Needham MA

Allen Downey is a Professor of Computer Science at the Franklin W. Olin College of Engineering.

Allen Downey是Franklin W. Olin工学院一名计算机科学教授。

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 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 `spot`ted 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 11.10.

- 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 14.4.
- 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.

Contents

Chapter 1

The way of the program 编程的方法

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

本书的目标是教给你像计算机科学家一样思考。 这一思考的方式集成了数学、工程以及自然科学的一些最好的特点。 像数学家一样，计算机科学家使用正则语言表示思想（特别是计算）。 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。

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program."

对于计算机科学家，唯一最重要的技能是**解决问题**（**problem solving**）。 解决问题意味着对问题进行形式化、对解决方案创新地思考以及清晰、精确地表达解决方案的能力。 事实证明，学习编程的过程是一个极佳的实践解决问题技能的机会。 这也是为什么本章被称为"编程的方法"。

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

一个层面上，你将学习编程，本身就是一个有用的技能。 另一个层面上，你将使用编程作为达到目的的手段。 随着我们的深入，这将变得更清晰。

1.1 The Python programming language Python编程语言

The programming language you will learn is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C, C++, Perl, and Java.

你将学习的编程语言是Python。Python是**高级语言**（**high-level language**）的一个例子；其它你可能听说的高级语言包括C、C++、Perl以及Java。

There are also **low-level languages**, sometimes referred to as “machine languages” or “assembly languages.” Loosely speaking, computers can only run programs written in low-level languages. So programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

另外也有**低级语言**（**low-level languages**），有时指“机器语言”或“汇编语言”。 大概来讲，计算机只能运行低级语言写的程序。 因此使用高级语言写的程序运行之前必须被处理。 这一额外的处理过程将花些时间，这是高级语言的一个小缺点。

The advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

优点是巨大的。首先，使用高级语言编程要容易的多。 用高级语言写程序需要花更少的时间，它们更短、更易读并且更有可能正确。 其次，高级语言是**可移植的**（**portable**），也就是说只需很少的修改或者无需修改， 它们就可以运行在不同类型的计算机上。 低级语言只能运行于一种类型的计算机上，如果要运行于其它类型的计算机上， 不得不进行重写。

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

由于这些优点，几乎所有的程序都是用高级语言写的。 低级语言只被用于很少的特殊应用。

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. Figure 1.1 shows the structure of an interpreter.

将高级语言处理成低级语言有两种程序：**解释器**（**interpreters**）和**编译器**（**compilers**）。 解释器读高级语言程序并且执行它，也就是说它按照程序说的去做。它一次处理一些程序，交替地读几行程序并且执行计算。 图 1.1 展示了一个解释器的结构。

A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. Figure 1.2 shows the structure of a compiler.

编译器在程序开始运行之前，读取整个程序并且将其全部翻译。 在这种情况下，高级语言程序被称为**源代码**（**source code**）， 翻译后的程序被称为**目标代码**（**object code**）或

Figure 1.1: An interpreter processes the program a little at a time, alternately reading lines and performing computations. 解释器一次处理一些程序，交替地读几行程序并且执行计算。

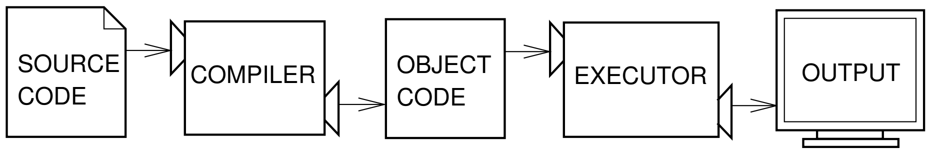

Figure 1.2: A compiler translates source code into object code, which is run by a hardware executor. 编译器将源代码翻译成可被硬件执行的目标代码。

者**可执行代码**（**executable**）。一旦一个程序被编译，你可以重复的执行它无需再翻译。图 1.2 展示了一个编译器的结构。

Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: **interactive mode** and **script mode**. In interactive mode, you type Python programs and the interpreter displays the result:

Python被认为是解释性语言，因为Python程序被解释器执行。有两种方法使用解释器：**交互式模式**（**interactive mode**）和**脚本模式**（**script mode**）。在交互式模式下，你键入Python程序，解释器显示结果：

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

The chevron, >>>, is the **prompt** the interpreter uses to indicate that it is ready. If you type `1 + 1`, the interpreter replies 2.

>>>是**提示符**（**prompt**），解释器用其表示它准备好了。如果你键入`1 + 1`，解释器回复`2`。

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with `.py`.

另外，你可以将代码存储在一个文件中，并且用解释器执行这个文件的内容，其被成为**脚本**（**script**）。习惯上，Python脚本以`.py`结尾命名。

To execute the script, you have to tell the interpreter the name of the file. If you have a script named `dinsdale.py` and you are working in a UNIX command window, you type `python dinsdale.py`. In other development environments, the details of executing scripts are different. You can find instructions for your environment at the Python website `http://python.org`.

为了执行脚本，你必须告诉解释器文件名。如果你有一个命名为`dinsdale.py`的脚本并且你在UNIX命令窗口中工作，你键入`python dinsdale.py`。在其它的开发环境中，执行

脚本的细节不太一样。 你可以在Python网站`http://python.org`中找到对于你的环境的说明。

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

在交互模式中工作便于测试零散的代码，因为你可以立即键入并且执行它们。 但是对于任何多于几行的代码，你应该保存你的代码为脚本以便将来修改并执行它。

1.2 What is a program? 什么是程序?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

一个**程序**（**program**）是一序列指令，其具体说明如何执行一个计算。 计算可能是一些数学的东西，例如解一个公式系统或者找到一个多项式的解， 而它也可能是一个符号计算，例如在一个文档中搜索并替换文本或者（不可思议地）编译一个程序。

The details look different in different languages, but a few basic instructions appear in just about every language:

不同语言细节看起来不一样，但是一些基本的指令几乎出现于每个语言当中：

input: Get data from the keyboard, a file, or some other device.

输入： 从键盘、文件或者其它设备获得数据。

output: Display data on the screen or send data to a file or other device.

输出： 在屏幕上显示数据或者将数据发送到一个文件或其它设备。

math: Perform basic mathematical operations like addition and multiplication.

数学： 执行基本的数学运算，如加法和乘法。

conditional execution: Check for certain conditions and execute the appropriate code.

有条件执行： 检查某一条件并执行适当的代码。

repetition: Perform some action repeatedly, usually with some variation.

重复： 重复执行某些动作，通常有一些变化。

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

无论你是否相信，这几乎是程序的全部了。 每个你曾经用过的程序，无论多么复杂，几乎都是由这些指令构成的。 因此你可以认为编程就是将大的、复杂的任务分解为越来越小的子任务， 直到这些子任务足够简单到被其中一个基本指令执行。

That may be a little vague, but we will come back to this topic when we talk about **algorithms**.

这可能有些模糊，但是当我们讨论**算法**（**algorithms**）的时候，将回到这些主题。

1.3 What is debugging? 什么是调试?

Programming is error-prone. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**.

编程是易于出错的。由于比较奇怪的原因，编程的错误被称为**虫子**（**bugs**）， 并且追踪它们的过程被称为**找虫子或调试**（**debugging**）。

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

程序中有三种错误：语法错误、运行时错误和语义错误。 为了更快的追踪它们，对其进行区分是很有帮助的。

1.3.1 Syntax errors 语法错误

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a **syntax error**.

Python只能执行语法正确的程序。 否则，解释器显示一个错误信息。 **语法**（**Syntax**）指的是一个程序的结构以及关于那些结构的规则。 例如，括号必须成对出现，所以`(1 + 2)`是合法的， 而`8)`是一个**语法错误**（**syntax error**）。

In English readers can tolerate most syntax errors, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

英语读者能容忍大多数语法错误， 这也是为什么我们能读e. e. cummings的诗而不会发出错误信息。 Python没这么宽容。如果你的程序任何地方有一个语法错误， Python将显示一个错误信息并且退出。你将不能运行你的程序。 在你编程生涯的前几周，你可能将花很多时间追踪语法错误。 当你更有经验时，你将犯更少的错误并且更快的找到它们。

1.3.2 Runtime errors 运行时错误

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

第二类错误是运行时错误，之所以这么叫，是因为程序开始运行以后， 这些错误才出现。这些错误也被称为**异常**（**exceptions**）， 因为它们通常意味着一些异常的（并且是坏的）事情发生。

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

在前几章简单的程序中很少有运行时错误，因此在你遇到它之前可能要一段时间。

1.3.3 Semantic errors 语义错误

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

第三类错误是**语义错误（semantic error）**。 如果你的程序中有一个语义错误，在计算机不会生成错误信息的意义上， 它将成功运行。但是，它将不会做正确的事情。它将做其它的事情。 特别地，它将做你告诉它要做的事情。

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

问题是你写的程序不是你想写的程序。程序的意义（它的语义）是错误的。 识别语义错误可能是棘手的，因为这需要你反过头来看一下程序的输出以及试图指出它在做什么。

1.3.4 Experimental debugging 实验性调试

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

你将获得的最重要的技能之一是调试。 虽然它可能令人泄气，但是调试是编程最富含智慧、挑战以及乐趣的部分之一。

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

在某些方面，调试像是侦探工作。 你面对一些线索并且你必须对导致你看到的结果的过程和事件进行推理。

Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four*)

调试也像是一种实验性科学。一旦你有一个关于什么出错了的想法， 你修改你的程序然后重试。 如果你的假设是正确的，那么你可以预测修改的结果，并且你向好使的程序迈近了一步。 如果你的假设是错误地，那么你不得不再提一个新的想法。 如Sherlock Holmes所指出的："当你排除了所有的不可能，无论剩下的是什么， 即使是不可能的，一定是真相。"（A. Conan Doyle, *The Sign of Four*）

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

对一些人来说，编程和调试是相同的事情。 那就是，编程是逐步调试一个程序的过程，直到它做了你想要的。 这一想法意味着你应该从一个能做些事情的程序开始并做小的改动，随着你进展调试它，以至于你总是有一个可以工作的程序。

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux." (*The Linux Users' Guide* Beta Version 1).

例如，Linux是一个操作系统，其包括成千上万行代码， 但是它起始于Linus Torvalds写的一个用于探索Intel 80386芯片的简单程序。 根据Larry Greenfield所说，Linus的早期项目之一是一个交替打印AAAA和BBBB的程序。 这后来演变为了Linux。"（*Linux*用户手册Beta Version 1）

Later chapters will make more suggestions about debugging and other programming practices.

后面的章节将给出更多关于调试和其它编程实践的建议。

1.4 Formal and natural languages 正则和自然语言

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 languages**）是人们说的语言，例如英语、西班牙语和法语。 它们不是人类设计的（虽然人们试图在其上强加上一些顺序）， 他们自然的演变。

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

正则语言（**formal languages**）是由人为了特殊应用设计的。 例如，数学家使用的符号就是正则语言，其特别擅长表示数字和符号直接的关系。 化学家使用正则语言表示分子的化学结构。 最重要的是：

> **Programming languages are formal languages that have been designed to express computations.**
>
> **编程语言是被设计用于表示计算的正则语言。**

Formal languages tend to have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but $3+ = 3\$6$ is not. H_2O is a syntactically correct chemical formula, but $_2Zz$ is not.

正则语言倾向于具有严格的语法规则。 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是语法正确的化学表达式，而$_2Zz$则不是。

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3+=3\$6$ is that \$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

语法规则有关于**记号**（**tokens**）和结构的两种类型。 记号是语言的基本元素，例如单词、数字和化学元素。 $3+=3\$6$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）。相似的，$_2Zz$也不合法，因为没有一个元素简称Zz。

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement $3+=3$ is illegal because even though $+$ and $=$ are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

第二种语法错误关于一个语句的结构。 也就是记号被安排的方式。语句$3+=3$是非法的，因为即使$+$和$=$都是合法的记号， 你也不能把其中一个放在另一个右面。 相似的，在化学式中，下标位于元素之后，不能在前面。

Exercise 1.1. *Write a well-structured English sentence with invalid tokens in it. Then write another sentence with all valid tokens but with invalid structure.*

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

当你读一个用英语写的句子或者用正则语言写的语句时， 你得指出句子的结构是什么（虽然在自然语言中，你潜意识地做这个）。 这个过程被称作**分析**（**parsing**）。

For example, when you hear the sentence, "The penny dropped," you understand that "the penny" is the subject and "dropped" is the predicate. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a penny is and what it means to drop, you will understand the general implication of this sentence.

例如，当你听到句子"The penny dropped"时，你理解"the penny"是主语， "dropped"是谓词。一旦你分析完一个句子，你可以指出它的含义是什么， 或者句子的语义。假设你知道penny是什么以及drop是什么意义， 你将理解这个句子的含义。

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are some differences:

虽然正则语言和自然语言有很多共同的特点—标记、结构、语法以及语义— 它们也有一些不同：

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

歧义性： 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。 正则语言被设计成几乎或者完全没有歧义，这意味着不管上下位是什么， 任何语句都只有一个意义。

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

冗余性： 为了弥补歧义性并且减少误解，自然语言使用很多冗余。 结果，它们经常很冗长。正则语言较少的冗余，更简洁。

literalness: Natural languages are full of idiom and metaphor. If I say, "The penny dropped," there is probably no penny and nothing dropping (this idiom means that someone realized something after a period of confusion). Formal languages mean exactly what they say.

非引申义性： 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（这个成语意思是有人经过一段时间的混乱后， 意识到一些事情）。正则语言含义和它们说的完全一样。

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

说自然语言长大的人—所有人—经常一时难于适应正则语言。 在某些方面，正则语言和自然语言直接的不同类似诗歌和散文直接的不同， 但是更重要的是：

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

诗歌： 单词不但被用于表示含义，还被用于表示声音。 整首诗一起表示一个效果或者情感上的响应。 歧义不但常见，而且经常是故意安排的。

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

散文： 单词表面的含义更重要，结构贡献的比含义更多。 散文比诗歌更适合分析，但仍然经常有歧义。

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

程序： 计算机程序的含义是无歧义、无引申义的， 并且通过对标记和结构的分析被整体理解。

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

这是对读程序（以及其它正则语言）的一些建议。 首先，记住正则语言比自然语言稠密的多，因此需要花更长的时间读它们。 其次，结构非常重要，因此从上到下，从左到右的读通常不是一个好的主意。 相反，要学会在你的头脑中分析一个程序，识别记号并且理解结构。 最后，注意细节问题。拼写和标点符号的小错误， 这些在自然语言中你可以侥幸成功，但是在正则语言中会有很大的不同。

1.5 The first program 第一个程序

Traditionally, the first program you write in a new language is called "Hello, World!" because all it does is display the words "Hello, World!". In Python, it looks like this:

传统上，你用一门新的语言写的第一个程序被称作"Hello, World!"， 因为它所做的所有事情就是显示单词"Hello, World!"。 在Python中，它看起来是这个样：

```
print 'Hello, World!'
```

This is an example of a **print statement**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the words

这是**打印语句**（**print statement**）的一个实例，其并不是真的在纸上打印任何东西。 它在屏幕上显示一个值。在此例中，结果是单词：

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

这个程序中的引号标记被打印文本的开始和结束。 它们并没有出现在结果中。

In Python 3, the syntax for printing is slightly different:

在Python 3中，打印的语法有些不同：

```
print('Hello, World!')
```

The parentheses indicate that `print` is a function. We'll get to functions in Chapter 3.

括号指出`print`是一个函数。 我们将在第 3章接触函数。

For the rest of this book, I'll use the print statement. If you are using Python 3, you will have to translate. But other than that, there are very few differences we have to worry about.

对于本书剩余的部分，我将使用print语句。 如果你正在使用Python 3，你将不得不进行翻译。 但是除此之外，几乎没有什么我们需要担心的不同。

1.6 Debugging 调试

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run most of the examples in interactive mode, but if you put the code in a script, it is easier to try out variations.

在一台计算机前阅读本书是一个好的主意，由此你可以边读书边试例子。 你可以在交互模式下运行大多数的例子，但是如果你将代码放入脚本中， 可以方便的试着改动他们。

Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the "Hello, world!" program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?

每当你试验一个新特征的时候，你应该试着犯些错误。 例如，在"Hello, world!"程序中，如果你少写一个引号会发生什么？ 如果两个引号都不写呢？ 如果将`print`拼写错呢？

This kind of experiment helps you remember what you read; it also helps with debugging, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

这种实验有助于帮你记住你所读过的内容。 他也有助于调试，因为你会知道各种错误信息的含义。 现在故意犯些错误总比将来发生意外的错误好。

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent or embarrassed.

编程，特别是调试，有时带有很强的情感。 如果你正和一个很难的错误奋战，你可能感到生气、沮丧或尴尬。

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

有证据表明，人们很自然地对计算机做出响应，仿佛它们就是人。 当它们做得很好的时候，我们认为它们就是队友。 当它们固执或无礼的时候，我们也会对它们也像对待固执或无礼的人一样。 （Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*）

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

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具有特定能力的雇员， 例如速度和精度，并且有些特别的缺点，像缺乏沟通以及不善于把握大局。

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

你的工作是成为一个好的管理者：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进行斗争的方法， 不让你的反应干扰你有效的工作。

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a debugging section, like this one, with my thoughts about debugging. I hope they help!

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都是一个非常有价值的技能。

1.7 Glossary 术语表

problem solving（解决问题）： The process of formulating a problem, finding a solution, and expressing the solution.

high-level language（高级语言）： A programming language like Python that is designed to be easy for humans to read and write.

low-level language（低级语言）： A programming language that is designed to be easy for a computer to execute; also called "machine language" or "assembly language."

portability（可移植性）： A property of a program that can run on more than one kind of computer.

interpret（解释）： To execute a program in a high-level language by translating it one line at a time.

compile（编译）： To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

source code（源代码）： A program in a high-level language before being compiled.

object code（目标代码）： The output of the compiler after it translates the program.

executable（可执行）： Another name for object code that is ready to be executed.

prompt（提示符）： Characters displayed by the interpreter to indicate that it is ready to take input from the user.

script（脚本）： A program stored in a file (usually one that will be interpreted).

interactive mode（交互模式）： A way of using the Python interpreter by typing commands and expressions at the prompt.

script mode（脚本模式）： A way of using the Python interpreter to read and execute statements in a script.

program（程序）： A set of instructions that specifies a computation.

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

bug（臭虫）： An error in a program.

debugging（调试）： The process of finding and removing any of the three kinds of programming errors.

syntax（语法）： The structure of a program.

syntax error（语法错误）： An error in a program that makes it impossible to parse (and therefore impossible to interpret).

exception（异常）： An error that is detected while the program is running.

semantics（语义）： The meaning of a program.

semantic error（语义错误）： An error in a program that makes it do something other than what the programmer intended.

natural language（自然语言）： Any one of the languages that people speak that evolved naturally.

formal language（形式语言）： Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token（记号）: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

parse（分析）: To examine a program and analyze the syntactic structure.

print statement（打印语句）: An instruction that causes the Python interpreter to display a value on the screen.

1.8 Exercises

Exercise 1.2. *Use a web browser to go to the Python website* `http://python.org`*. This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.*

For example, if you enter `print` *in the search window, the first link that appears is the documentation of the* `print` *statement. At this point, not all of it will make sense to you, but it is good to know where it is.*

Exercise 1.3. *Start the Python interpreter and type* `help()` *to start the online help utility. Or you can type* `help('print')` *to get information about the* `print` *statement.*

If this example doesn't work, you may need to install additional Python documentation or set an environment variable; the details depend on your operating system and version of Python.

Exercise 1.4. *Start the Python interpreter and use it as a calculator. Python's syntax for math operations is almost the same as standard mathematical notation. For example, the symbols* `+`, `-` *and* `/` *denote addition, subtraction and division, as you would expect. The symbol for multiplication is* `*`.

If you run a 10 kilometer race in 43 minutes 30 seconds, what is your average time per mile? What is your average speed in miles per hour? (Hint: there are 1.61 kilometers in a mile).

Chapter 2

Variables, expressions and statements 变量、表达式和语句

2.1 Values and types 值和类型

A **value** is one of the basic things a program works with, like a letter or a number. The values we have seen so far are `1`, `2`, and `'Hello, World!'`.

值（**value**）是基本的与程序打交道的事物之一，像字母或者数字。到目前，我们已经见过的值是`1`、`2`以及`'Hello, World!'`。

These values belong to different **types**: `2` is an integer, and `'Hello, World!'` is a **string**, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

这些值属于不同的**类型**（**types**）： `2`是一个整数，`'Hello, World!'`是一个**字符串**（**string**），之所以这么叫，是因为它包括"一串"字母。你（以及解释器）能识别字符串是因为他们被括在引号中间。

If you are not sure what type a value has, the interpreter can tell you.

如果你不确定一个值具有什么类型，解释器可以告诉你。

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating-point**.

不奇怪，字符串属于`str`类型，整数属于`int`类型。不太明显地，具有小数点的数字属于一种被称作`float`的类型，因为这些数字使用**浮点**（**floating-point**）格式表示。

```
>>> type(3.2)
<type 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

像'17'以及'3.2'的值是什么类型呢？ 它们看起来像数字，但是它们在引号里，像是字符串。

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

They're strings.

它们是字符串。

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

当你有一个比较大的整数，你可能在每三个数字之间使用逗号分割它，如 1,000,000。 在Python中，这不是一个合法的整数，但是它是合法的：

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

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

然而，那不是你期望的！ Python翻译1,000,000为逗号分割的一系列整数。 这是我们所见过的第一个语义错误的例子： 代码无错误信息地运行，但是它并没有做"正确的"事情。

2.2 Variables 变量

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

编程语言最强大的特征之一是操作**变量**（**variables**）的能力。 变量是指一个值的名字。

An **assignment statement** creates new variables and gives them values:

一个**赋值语句**（**assignment statement**）生成新的变量并且将值赋给它们。

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

This example makes three assignments. The first assigns a string to a new variable named message; the second gives the integer 17 to n; the third assigns the (approximate) value of π to pi.

这个例子进行了三次赋值。 第一个将一个字符串赋给一个新的变量名message； 第二个将整数17赋给n； 第三个将π的（近似）值赋给pi。

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). Figure 2.1 shows the result of the previous example.

message ⟶ 'And now for something completely different'
n ⟶ 17
pi ⟶ 3.1415926535897932

Figure 2.1: State diagram.

在纸上表示变量的方法通常是写一个变量名跟着一个箭头指向变量的值。 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（将其想为变量的心理状态）。 图 2.1展示了前面例子的的结果。

The type of a variable is the type of the value it refers to.

变量的类型是其所指向值的类型。

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Exercise 2.1. *If you type an integer with a leading zero, you might get a confusing error:*

```
>>> zipcode = 02492
                  ^
SyntaxError: invalid token
```

Other numbers seem to work, but the results are bizarre:

```
>>> zipcode = 02132
>>> zipcode
1114
```

Can you figure out what is going on? Hint: display the values `01`, `010`, `0100` *and* `01000`.

2.3 Variable names and keywords 变量名和关键字

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.

程序员通常为变量选择有意义的名字—它们说明了该变量的用途。

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

变量名可以任意长。他们可以包括字母和数字，但是必须以字母开始。 使用大写字母是合法的，但是一个变量名以一个小写字母开始是个不错的主意 （后面你会看到原因）。

The underscore character, `_`, can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`.

下划线字符`_`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`。

If you give a variable an illegal name, you get a syntax error:

如果你给了变量一个非法的名，将获得一个语法错误：

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

`76trombones` is illegal because it does not begin with a letter. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

`76trombones`是非法的，因为它不是以字母开始的。`more@`是非法的，因为它包含了一个非法字符`@`。但是，`class`错在哪儿了呢?

It turns out that `class` is one of Python's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

实际上，`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 2 has 31 keywords:

Python 2有31个关键字：

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

In Python 3, `exec` is no longer a keyword, but `nonlocal` is.

在Python 3中，`exec`不在是一个关键字，但`nonlocal`是。

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

你可能想随手携带这个表。如果解释器抱怨你的一个变量名并且你不知道为什么， 看一下是否它在这个列表中。

2.4 Operators and operands 运算符和运算数

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**.

运算符（operators）是特殊的符号，表示类似加法和乘法的计算。运算符被应用的值被称作**运算数（operands）**。

The operators `+`, `-`, `*`, `/` and `**` perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

运算符`+`、`-`、`*`、`/`和`**`分别执行加、减、乘、除和指数运算， 如下面的例子：

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

In some other languages, ^ is used for exponentiation, but in Python it is a bitwise operator called XOR. I won't cover bitwise operators in this book, but you can read about them at http://wiki.python.org/moin/BitwiseOperators.

在一些其它语言中，^表示指数运算， 但是在Python中，它表示XOR（异或）位运算。 本书不包括位运算符， 但是你可以在http://wiki.python.org/moin/BitwiseOperators上阅读关于它们的内容

In Python 2, the division operator might not do what you expect:

在Python 2中，除法运算符可能不是如你所期望的:

```
>>> minute = 59
>>> minute/60
0
```

The value of `minute` is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **floor division**. When both of the operands are integers, the result is also an integer; floor division chops off the fraction part, so in this example it rounds down to zero.

`minute`的值是59，通常的算数中，59除以60是0.98333而不是0. 造成这一差异的原因是Python执行**向下取整除**（**floor division**）。 当两个运算数都是整数时，结果也是整数； 向下取整除砍掉小数部分，因此此例中其下舍入成为了0。

In Python 3, the result of this division is a `float`. The new operator `//` performs floor division.

在Python 3中，这一除法的结果是一个浮点数。 新的运算符//执行向下取整除。

If either of the operands is a floating-point number, Python performs floating-point division, and the result is a `float`:

如果其中有一个运算数是浮点数，则Python执行浮点除，结果是浮点数：

```
>>> minute/60.0
0.98333333333333328
```

2.5 Expressions and statements 表达式和语句

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

一个**表达式**（**expression**）是值、变量和运算符的组合。 值自身被认为是一个表达式，变量也是，因此下面都是合法的表达式（假设变量x已经被赋值了）：

```
17
x
x + 17
```

A **statement** is a unit of code that the Python interpreter can execute. We have seen two kinds of statement: print and assignment.

一条**语句**（**statement**）是一个Python解释器可以执行的代码单元。 你已经见过两种语句了：打印和赋值。

Technically an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

技术上讲，一个表达式也是一条语句，但是将其认为是非语句可能会更简单。 重要的不同是表达式具有一个值；语句没有。

2.6 Interactive mode and script mode 交互模式和脚本模式

One of the benefits of working with an interpreted language is that you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

使用解释型语言的一个好处是，在你将一段代码放入脚本之前， 你可以在交互模式下测试一下。 但是交互模式和脚本模式之间也有不同，容易搞混。

For example, if you are using Python as a calculator, you might type

例如，如果你正将Python用作计算器，你可能键入：

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

The first line assigns a value to `miles`, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. So we learn that a marathon is about 42 kilometers.

第一行将一个值赋给`miles`，但是并没有显示效果。 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。 所以我们学到，一段马拉松大概是42公里。

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python actually evaluates the expression, but it doesn't display the value unless you tell it to:

但是如果你将相同的代码键入一个脚本并且运行它，你将根本不能获得输出。 在完全的脚本模式下，不会有显示效果。 Python实际上计算了表达式，但是如果你不告诉它，它不会显示结果。

```
miles = 26.2
print miles * 1.61
```

This behavior can be confusing at first.

此行为开始可能有些令人茫然。

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

一个脚本通常包括一序列语句。 如果有多余一条的语句，那么每条语句执行后都显示结果。

For example, the script

例如脚本

```
print 1
x = 2
print x
```

produces the output

产生输出

```
1
2
```

The assignment statement produces no output.

赋值语句不差生输出。

Exercise 2.2. *Type the following statements in the Python interpreter to see what they do:*

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

Now put the same statements into a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

2.7 Order of operations 运算的顺序

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

当一个表达式中有多于一个运算符时，计算的顺序依赖**优先级规则**（**rules of precedence**）。对于算数运算，Python遵循数学的惯例。缩写 **PEMDAS**有助于记住这一规则：

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.

- 括号（**P**arentheses）具有最高的优先级，并且可以被用于强制表达式按你需要的顺序计算。既然在括号中的表达式首先被计算，那么`2 * (3-1)`是4，`(1+1)**(5-2)`是8。你也可以用括号使得一个表达式更易读，如`(minute * 100) / 60`，即使它不能改变运算的结果。

- **E**xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.

- 指数运算（**E**xponentiation）具有其次的优先级，因此`2**1+1`是3而非4，`3*1**3`是3而非27。

- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.

- 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。

- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression `degrees / 2 * pi`, the division happens first and the result is multiplied by `pi`. To divide by 2π, you can use parentheses or write `degrees / 2 / pi`.

- 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`。

I don't work very hard to remember rules of precedence for other operators. If I can't tell by looking at the expression, I use parentheses to make it obvious.

我不太努力，所以记不住其它运算符的优先级规则。如果通过观察表达式，我不能说清楚，我使用括号使其变得更明显。

2.8 String operations 字符串运算

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

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，因此下面是非法的：

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

The `+` operator works with strings, but it might not do what you expect: it performs **concatenation**, which means joining the strings by linking them end-to-end. For example:

`+`运算符可用于字符串，但是它可能不会做你所期望的事情：它执行**级联**（**concatenation**）运算，也就是将字符串端到端的连起来。例如：

```
first = 'throat'
second = 'warbler'
print first + second
```

The output of this program is `throatwarbler`.

程序的输出是`throatwarbler`。

The `*` operator also works on strings; it performs repetition. For example, `'Spam'*3` is `'SpamSpamSpam'`. If one of the operands is a string, the other has to be an integer.

`*`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数。

This use of `+` and `*` makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `'Spam'*3` to be the same as `'Spam'+'Spam'+'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

+和*的 使 用 类 比 加 法 和 乘 法 也 讲 得 通 。 就 像4*3与4+4+4等 价 ， 我 们 希望'Spam'*3和'Spam'+'Spam'+'Spam'一样，而且确实是这样。 另一方面，字符串的级联和重复与整数的加法和乘法截然不同。 你能想出来一个加法具有，而级联不具有的性质么？

2.9 Comments 注释

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

随着程序变得越来越大，越来越复杂，它们变得越来越难读。 正则语言是稠密的，读一段代码并输出其做什么或者为什么通常很难。

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they start with the # symbol:

因此，增加在你的程序中增加标注来用自然语言解释程序做什么通常是比较好的办法。 这些标注被称为**注释**（**comments**），其以#符号开始。

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

此例中，注释独立一行。你也可以将注释放在行尾：

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

Everything from the # to the end of the line is ignored—it has no effect on the program.

从#开始到行尾的所有东西都被忽略了—其对程序没有影响。

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

当注释写出代码不明显的特征的时候最有效。 假设读者能够指出代码做了什么比较适当；解释为什么则更有用。

This comment is redundant with the code and useless:

下面这个注释是代码的重复而且没有什么用：

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

This comment contains useful information that is not in the code:

下面的注释包括包含了代码中没有的有用的信息：

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

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

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， 因此这有个均衡问题。

2.10 Debugging 调试

At this point the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `US$`, which contain illegal characters.

此时你经常犯的语法错误是非法变量名，如`class`和`yield`是关键字， `odd~job`和`US$`包含了非法字符。

If you put a space in a variable name, Python thinks it is two operands without an operator:

如果你在变量名中间放了一个空格，Python认为它是两个没有运算符的运算数。

```
>>> bad name = 5
SyntaxError: invalid syntax
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

对于语法错误，错误信息帮助不大。 最常见的信息是`SyntaxError: invalid syntax`以及 `SyntaxError: invalid token`，这两个都没什么信息量。

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

你最容易犯的运行时错误是"定义前使用"，也就是说试图在对一个变量赋值前使用它。 如果你拼写错了变量名这就可能发生。

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

变量名是大小写敏感的，因此`LaTeX`和`latex`不一样。

At this point the most likely cause of a semantic error is the order of operations. For example, to evaluate $\frac{1}{2\pi}$, you might be tempted to write

此时最容易引起的语义错误是运算的顺序。 例如，为了计算$\frac{1}{2\pi}$，你可能写成

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

但是，首先进行出发运行，因此你获得了$\pi/2$，这并不是一回事儿！ 对于Python，没有办法知道你的意图是写什么，因此此例中你不会获得错误信息； 你只会获得错误的结果。

2.11 Glossary 术语表

value（值）: One of the basic units of data, like a number or string, that a program manipulates.

type（类型）**:** A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

integer（整数）**:** A type that represents whole numbers.

floating-point（浮点）**:** A type that represents numbers with fractional parts.

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

variable（变量）**:** A name that refers to a value.

statement（语句）**:** A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

assignment（赋值）**:** A statement that assigns a value to a variable.

state diagram（状态度）**:** A graphical representation of a set of variables and the values they refer to.

keyword（关键字）**:** A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

operator（运算符）**:** A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

operand（运算数）**:** One of the values on which an operator operates.

floor division（向下取整除）**:** The operation that divides two numbers and chops off the fraction part.

expression（表达式）**:** A combination of variables, operators, and values that represents a single result value.

evaluate（计算）**:** To simplify an expression by performing the operations in order to yield a single value.

rules of precedence（优先级规则）**:** The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

concatenate（级联）**:** To join two operands end-to-end.

comment（注释）**:** Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

2.12 Exercises

Exercise 2.3. *Assume that we execute the following assignment statements:*

```
width = 17
height = 12.0
delimiter = '.'
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width/2`

2. `width/2.0`

3. `height/3`

4. `1 + 2 * 5`

5. `delimiter * 5`

Use the Python interpreter to check your answers.

Exercise 2.4. *Practice using the Python interpreter as a calculator:*

1. *The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5? Hint: 392.7 is wrong!*

2. *Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?*

3. *If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?*

Chapter 3

Functions 函数

3.1 Function calls 函数调用

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a **function call**:

在编程的语境下，**函数（function）**对是一序列执行计算语句的命名。 当你定义一个函数的时候，你指定了名字和语句序列。 随后，你可以通过名字"调用"该函数。 我们已经见过了一个**函数调用（function call）**的例子。

```
>>> type(32)
<type 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

函数名是`type`。括号中的表达是被称为函数的**实参（argument）**。 此函数的结果是实参的类型。

It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

人们常说函数"接受"实参并且"返回"一个结果。 该结果被称为**返回值（return value）**

3.2 Type conversion functions 类型转换函数

Python provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

Python提供内建函数将值从一种类型转换为另一种类型。 函数`int`接受任意值，并在其能做到的情况下，将其转换成一个整数， 否则会抗议：

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

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

`int`能将浮点数转换为整数，但是它并不完美，截掉了小数部分：

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

`float` converts integers and strings to floating-point numbers:

`float`将整数或者字符串转化为浮点数：

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

Finally, `str` converts its argument to a string:

最后，`str`将实参转成字符串。

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

3.3 Math functions 数学函数

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Python有数学模块，其提供了大部分熟悉的数学函数。**模块**（**module**）是一个包含一批相关函数的文件。

Before we can use the module, we have to import it:

在我们可以使用模块之前，我们需要导入它：

```
>>> import math
```

This statement creates a **module object** named math. If you print the module object, you get some information about it:

这条语句生成一个名为math的**模块对象**（**module object**）。如果你打印这个模块对象，你将获得关于它一些信息：

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

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

该模块对象包括定义在模块内的函数和变量。为了访问其中一个函数，你不得不指出该模块的名字以及函数名，并以点（也被称作句号）分割。这种格式被称作**点记法**（**dot notation**）。

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

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

The first example uses `log10` to compute a signal-to-noise ratio in decibels (assuming that `signal_power` and `noise_power` are defined). The math module also provides `log`, which computes logarithms base `e`.

第一个例子使用`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，其计算以`e`为底的对数。

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

第二个例子计算`radians`的sine值。变量名暗示`sin`函数以及其它三角函数（`cos`、`tan`等）接受弧度参数。为了从度转为弧度，除以360并乘以2π：

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

表达式`math.pi`从math模块中获得变量`pi`。该变量的值是对π的近似，精度大约15位数。

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

如果你懂几何学，你可以通过将之前的结果和2的平方根再除以2进行比较：

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

3.4 Composition 组合

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

目前为止，我们已经分别看到了程序的基本元素—变量、表达式和语句— 还没有讨论如何组合它们。

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

程序设计语言的最有用特征之一是分解成小块并将其**组合**（**compose**）的能力。例如，函数的实参能够接受任何类型的表达式，包括算数运算符：

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

And even function calls:

甚至函数调用：

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

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error (we will see exceptions to this rule later).

你几乎可以将一个任意值、表达式，放在任何地方，除了一个例外： 赋值语句的左侧必须是一个变量名，任何其它的表达式都是语法错误的 （后面我们会看到这个例外的规则）。

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

3.5 Adding new functions 增加新函数

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.

目前为止，我们只了使用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**）指出新函数的名 以及当函数被调用时执行的语句序列。

Here is an example:

这是一个例子：

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

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

def是一个关键字，其指明这是一个函数定义。 函数名是print_lyrics。 函数的命名规则和变量名相同：字母、数字以及一些标点符号是合法的， 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 一个变量和一个函数同名。

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

函数名后面的空括号表明该函数不接受任何实参。

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces (see Section 3.14). The body can contain any number of statements.

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**）。 函数体必须以冒号结尾，函数体必须是缩进的。 按照惯例，缩进经常是4个空格（见3.14节）。 函数体能包含任意条语句。

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

打印语句中的字符串被括在双引号中。单引号和双引号做同样的事情。 大多数人使用单引号，除了类似这里的情况，即单引号（也表示撇号）出现在字符串中。

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

如果你在交互模式中键入函数定义，解释器打印省略号（...）， 以便让你知道定义并不完整。

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

To end the function, you have to enter an empty line (this is not necessary in a script).

为了结束函数，你必须输入一个空行（这在脚本中不是必须的） 。

Defining a function creates a variable with the same name.

定义一个函数生成一个同名的变量。

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

The value of `print_lyrics` is a **function object**, which has type `'function'`.

`print_lyrics`的值是一个**函数对象**（**function object**）， 其类型是`'function'`。

The syntax for calling the new function is the same as for built-in functions:

调用新函数的语法和调用内建函数的相同：

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

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

一旦你定义了一个函数，你可以在另一个函数内部使用它。 例如，为了重复之前的叠句，我们可以写一个叫`repeat_lyrics`的函数：

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

And then call `repeat_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.
```

But that's not really how the song goes.

但那不是这首歌本来的样子。

3.6 Definitions and uses 定义和使用

Pulling together the code fragments from the previous section, the whole program looks like this:

汇集上一节的代码片段，整个程序看起来像这样：

```
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()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

改程序包含两个函数定义：`print_lyrics`和`repeat_lyrics`。 函数定义就像其它语句执行一样，但是左右是生成函数对象。 函数内部的语句直到被调用才会被执行，并且函数定义没有任何输出。

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

和你所期望的一样，函数执行之前，你必须生成它。 换句话说，函数定义必须在其第一次调用之前执行。

Exercise 3.1. *Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.*

Exercise 3.2. *Move the function call back to the bottom and move the definition of* `print_lyrics` *after the definition of* `repeat_lyrics`*. What happens when you run this program?*

3.7 Flow of execution 执行流程

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.

为了保证函数第一次使用之前被定义，你必须要知道语句被执行的顺序， 这也被称作**执行流程**（**flow of execution**）。

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

执行总是开始于程序的第一条语句。自顶向下，每次执行一条语句。

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

函数定义不改变程序执行的流程，但是请记住函数内部的语句直到该函数被调用时才执行。

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

函数调用像是在执行流程上饶了一个弯路。 流程不是到下一条语句，而是跳入函数体，执行那里所有的语句，然后回到它离开的位置。

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

听起来足够简单，直到你记得一个函数可以调用另一个函数。 当在一个函数中间的时候，程序可能不得不执行另一个函数里的语句。 但是当执行那个新函数的时候，程序可能不得不又执行另外一个函数！

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

幸运的是，Python善于跟踪它在哪里，因此每次一个函数完成时， 程序回到调用它的那个函数。当到达程序结尾时，程序终止。

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

什么是道德，这个肮脏的故事？当你读一个程序的时候， 你不总是从上到下的读。有时，如果你遵从执行流程会更有道理。

3.8 Parameters and arguments 形参和实参

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

我们已经见过的一些内建函数需要实参。 例如，当你调用`math.sin`时，你传递一个数字作为实参。 有些函数接受多于一个的实参：`math.pow` 接受两个，底数和指数。

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

在函数内部，实参被赋给被称作**形参**（**parameters**）的变量。 这是一个用户自定义函数的例子，其接受一个形参：

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

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.

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

This function works with any value that can be printed.

该函数对任意能被打印的值都有效。

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

相同的应用于内建函数的组合规则也适用于用户自定义函数， 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
```

The argument is evaluated before the function is called, so in the examples the expressions 'Spam '*4 and math.cos(math.pi) are only evaluated once.

实参在被调用之前被计算，因此在这些例子中， 表达式'Spam '*4和math.cos(math.pi)都只被计算一次。

You can also use a variable as an argument:

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.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn't matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

我们传递的实参名（michael）对形参（bruce）没有任何影响。 （在调用者里）无论值是什么都不要紧； 此处在print_twice里面，我们将所有人都叫做bruce。

3.9 Variables and parameters are local 变量和形参都是局部的

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

当你在一个函数里面生成一个变量时，它是**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)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

此函数接受两个实参，级联它们并打印结果两次。 这是使用它的一个例子：

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

When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

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

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

Parameters are also local. For example, outside `print_twice`, there is no such thing as `bruce`.

形参也都是局部的。例如在`print_twice`外面， 没有`bruce`这个东西。

3.10 Stack diagrams 栈图

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

为了跟踪哪个变量能在哪儿用，有时画一个**栈图（stack diagram）**比较有用。 像状态图一样，栈图展示每个变量的值，但是它们也展示了每个变量属于的函数。

Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in Figure 3.1.

每个函数用一个**框架（frame）**表示。 一个框架是一个有函数名的盒子，除此之外还有形参以及函数内部的变量。 前面例子的栈图如图3.1所示。

The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called

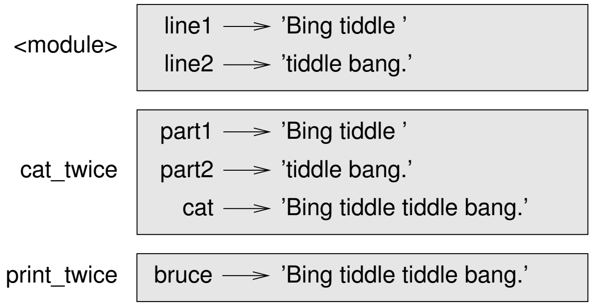

Figure 3.1: Stack diagram.

by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`.

栈里面的框架指出哪个函数被哪个函数调用等。 在此例中，`print_twice`被`cat_twice`调用， `cat_twice`被`__main__`调用，这是一个表示最上层框架的特殊的名字。 当你在所有函数之外生成一个变量时，它属于`__main__`。

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `bruce` has the same value as `cat`.

每个形参都指向其相应的实参的值。 因此`part1`和`line1`的值相同，`part2`和`line2`的值相同， `bruce`和 `cat`的值相同。

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

如果函数调用时发生错误，Python打印此函数的名字以及调用它的函数的名字， 以及那个调用的函数的名字，所有回到`__main__`的路径。

For example, if you try to access `cat` from within `print_twice`, you get a `NameError`:

例如，如果你在`print_twice`里面试图访问`cat`， 你获得一个`NameError`：

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print cat
NameError: name 'cat' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.

这个函数列表被称作一个**回溯**（**traceback**）。 它告诉你发生错误的程序文件，以及此时执行到什么行，什么函数。 它也展示引起错误的代码行。

The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

3.11 Fruitful functions and void functions 有返回值函数和无返回值函数

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them **fruitful functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.

有一些我们正在用的函数，例如数学函数，会产生结果； 由于没有更好的命名，我叫他们**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一个值。 他们被称为**无返回值函数（void functions）**。

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

当你调用一个有返回值函数时，你几乎总是想用结果做些事情， 例如你可能将它赋给一个变量或者把它作为一个表达式的一部分。

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

When you call a function in interactive mode, Python displays the result:

当你在交互模式下调用一个函数的时候，Python显示结果：

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

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

但是在脚本中，如果你只调用一个有返回值函数， 返回值永远就丢失了！

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

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

该脚本计算5的平方根，但是既然它没保存或者显示这个结果， 它不是非常有用。

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

无返回值函数可能在屏幕上显示一些东西或者有些其它的影响， 但是他们没有返回值。如果你试图将这个结果赋给一个变量， 你获得一个特殊的被称作`None`的值。

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

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

值`None`和字符串`'None'`不同。这是一个有它自己的类型的特殊变量：

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

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

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写有返回值函数。

3.12 Why functions? 为什么用函数?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

为什么值得麻烦得将一个程序分解成函数可能还不是很清楚。 有几个原因:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

  生成一个新的函数给你一个命名一组语句的机会, 这使得你的程序更容易读和调试。

- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

  通过避免重复调用代码, 函数使得程序更小。 之后, 如果你要做个变动, 你只需在一处变动即可。

- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

  将一个长程序分解为函数允许你一次调试一部分然后将它们集成为一个可行的整体。

- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

  良好设计的函数经常对多个程序都有用。一旦你写出并调试了一个函数, 你可以重用它。

3.13 Importing with `from` 使用`from`导入

Python provides two ways to import modules; we have already seen one:

Python提供两种方法导入模块。我们已经看到了一种:

```
>>> import math
>>> print math
<module 'math' (built-in)>
>>> print math.pi
3.14159265359
```

If you import `math`, you get a module object named `math`. The module object contains constants like `pi` and functions like `sin` and `exp`.

如果你导入`math`, 你获得一个名为`math`的模块对象。 该模块对象包括类似`pi`的常量以及类似`sin`和`exp`的函数。

But if you try to access `pi` directly, you get an error.

但是, 如果你试图直接访问`pi`, 会获得一个错误:

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

As an alternative, you can import an object from a module like this:

另一种方法是，你可以像这样从一个模块中导入一个对象：

```
>>> from math import pi
```

Now you can access pi directly, without dot notation.

现在，你可以直接访问pi，不需要点记法了。

```
>>> print pi
3.14159265359
```

Or you can use the star operator to import *everything* from the module:

或者你可以用星号运算符从模块中导入所有的东西：

```
>>> from math import *
>>> cos(pi)
-1.0
```

The advantage of importing everything from the math module is that your code can be more concise. The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

从数学模块中导入所有东西的优点是你的代码变得更简洁。 缺点是不同模块之间的名字，或者来自模块的名与你自己的变量名可能会冲突。

3.14 Debugging 调试

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.

如果你正在用一个文本编辑器写你的脚本，你可能会遇到空格和制表符的问题。 避免这些问题最好的办法是只使用空格（不用制表符）。 大多数识别Python的文本编辑器默认这么做，但有些不是。

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

制表符和空格通常是不可见的，这使得它们很难调试， 因此尽量找一个自动缩进的编辑器。

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case the program you are looking at in the text editor is not the same as the program you are running.

同时，别忘了在运行程序之前保存它。 一些开发环境自动做这个，但是有些不是。 如果是这样的话，你在文本编辑器上看到的程序和你运行的不一样。

Debugging can take a long time if you keep running the same, incorrect, program over and over!

如果你一遍一遍运行一个相同的、错误的程序，调试可能要花很长时间。

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print 'hello'` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

确定你所看到的代码是你所运行的。 如果不确定，将一些类似`print 'hello'`的代码放到程序的开始并再次运行它。 如果你么有看到`hello`，那么你没在运行一个正确的程序。

3.15 Glossary 术语表

function（函数）： A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function definition（函数定义）： A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object（函数对象）： A value created by a function definition. The name of the function is a variable that refers to a function object.

header（函数头）： The first line of a function definition.

body（函数体）： The sequence of statements inside a function definition.

parameter（形参）： A name used inside a function to refer to the value passed as an argument.

function call（函数调用）： A statement that executes a function. It consists of the function name followed by an argument list.

argument（实参）： A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable（局部变量）： A variable defined inside a function. A local variable can only be used inside its function.

return value（返回值）： The result of a function. If a function call is used as an expression, the return value is the value of the expression.

fruitful function（有返回值函数）： A function that returns a value.

void function（无返回值函数）： A function that doesn't return a value.

module（模块）： A file that contains a collection of related functions and other definitions.

import statement（导入语句）： A statement that reads a module file and creates a module object.

module object（模块对象）： A value created by an `import` statement that provides access to the values defined in a module.

dot notation（点记法）： The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition（组合）： Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution（执行流程）: The order in which statements are executed during a program run.

stack diagram（栈图）: A graphical representation of a stack of functions, their variables, and the values they refer to.

frame（框架）: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback（回溯）: A list of the functions that are executing, printed when an exception occurs.

3.16 Exercises

Exercise 3.3. *Python provides a built-in function called* `len` *that returns the length of a string, so the value of* `len('allen')` *is 5.*

Write a function named `right_justify` *that takes a string named* `s` *as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.*

```
>>> right_justify('allen')
                                                                 allen
```

Exercise 3.4. *A function object is a value you can assign to a variable or pass as an argument. For example,* `do_twice` *is a function that takes a function object as an argument and calls it twice:*

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

Here's an example that uses `do_twice` *to call a function named* `print_spam` *twice.*

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

do_twice(print_spam)
```

1. *Type this example into a script and test it.*
2. *Modify* `do_twice` *so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.*
3. *Write a more general version of* `print_spam`*, called* `print_twice`*, that takes a string as a parameter and prints it twice.*
4. *Use the modified version of* `do_twice` *to call* `print_twice` *twice, passing* `'spam'` *as an argument.*
5. *Define a new function called* `do_four` *that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.*

Solution: `http://thinkpython.com/code/do_four.py`*.*

Exercise 3.5. *This exercise can be done using only the statements and other features we have learned so far.*

1. *Write a function that draws a grid like the following:*

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

   Hint: to print more than one value on a line, you can print a comma-separated sequence:

```
print '+', '-'
```

   If the sequence ends with a comma, Python leaves the line unfinished, so the value printed next appears on the same line.

```
print '+',
print '-'
```

   The output of these statements is `'+ -'`.

   A `print` *statement all by itself ends the current line and goes to the next line.*

2. *Write a function that draws a similar grid with four rows and four columns.*

Solution: `http://thinkpython.com/code/grid.py`. *Credit: This exercise is based on an exercise in Oualline,* Practical C Programming, Third Edition*, O'Reilly Media, 1997.*

Chapter 4

Case study: interface design 案例学习：接口设计

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

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

4.1 TurtleWorld

To accompany this book, I have written a package called Swampy. You can download Swampy from http://thinkpython.com/swampy; follow the instructions there to install Swampy on your system.

作为对本书的补充，我写了一个叫Swampy的包。你可以从http://thinkpython.com/swampy下载Swampy，然后按照说明在你的系统上安装Swampy。

A **package** is a collection of modules; one of the modules in Swampy is `TurtleWorld`, which provides a set of functions for drawing lines by steering turtles around the screen.

一个包（**package**）是一套模块，`TurtleWorld`是Swampy中的一个模块，其提供了一组通过在屏幕上转向海龟的画线函数。

If Swampy is installed as a package on your system, you can import `TurtleWorld` like this:

如果Swampy作为包被安装在了你的系统上，你可以像这样导入`TurtleWorld`：

```
from swampy.TurtleWorld import *
```

If you downloaded the Swampy modules but did not install them as a package, you can either work in the directory that contains the Swampy files, or add that directory to Python's search path. Then you can import `TurtleWorld` like this:

如果你下载了Swampy模块但是没有作为包安装，你要么可以在含有Swampy的文件夹里工作，或者将此目录增加到Python的搜索路径。然后你可以像这样导入`TurtleWorld`：

```
from TurtleWorld import *
```

The details of the installation process and setting Python's search path depend on your system, so rather than include those details here, I will try to maintain current information for several systems at `http://thinkpython.com/swampy`.

安装和设置Python搜索路径的细节依赖于你的系统， 所以我不在此包括那些细节了， 我将试着在`http://thinkpython.com/swampy`维护多个系统的最新的信息。

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

生成一个叫`mypolygon.py`的文件并键入如下代码：

```
from swampy.TurtleWorld import *

world = TurtleWorld()
bob = Turtle()
print bob

wait_for_user()
```

The first line imports everything from the `TurtleWorld` module in the `swampy` package.

第一行在`swampy`里从`TurtleWorld`模块导入内容。

The next lines create a TurtleWorld assigned to `world` and a Turtle assigned to `bob`. Printing `bob` yields something like:

接下来几行生成一个TurtleWorld赋给`world`以及生成一个Turtle赋给`bob`。 打印`bob`产生像下面这样的东西：

```
<TurtleWorld.Turtle instance at 0xb7bfbf4c>
```

This means that `bob` refers to an **instance** of a Turtle as defined in module `TurtleWorld`. In this context, "instance" means a member of a set; this Turtle is one of the set of possible Turtles.

这意味着`bob`指向一个如`TurtleWorld`中定义的Turtle的**实例**（**instance**）。 在此处，"实例"的意思是集合的一个成员，该Turtle是所有可能的Turtle构成的集合的一个元素。

`wait_for_user` tells TurtleWorld 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.

`wait_for_user`告诉TurtleWorld等待用户做些什么， 但是在此例中除了关闭窗口外用户也做不了其它的事情。

TurtleWorld provides several turtle-steering functions: `fd` and `bk` for forward and backward, and `lt` and `rt` for left and right turns. 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 functions `pu` and `pd` stand for "pen up" and "pen down".

TurtleWorld提供几个海龟转向函数： `fd`和`bk`用于向前和向后，`lt`和`rt`是向左和向右。 每个海龟都握有一支笔，要么放下，要么抬起。 如果笔被放下，该海龟留下它走过的痕迹。函数`pu`和`pd`代表"笔抬起"和"笔放下"。

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

为了画一个向右的三角，在程序中加入这些代码 （在生成`bob`之后，调用`wait_for_user`之前）：

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

The first line tells bob to take 100 steps forward. The second line tells him to turn left.

第一行告诉bob向前走100步。第二行告诉它向左转。

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

当运行此程序时，你应该会看见bob现象东移动，然后向北， 在它后面留下两条线段。

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。如果没有成功，不要继续往下读。

4.2 Simple repetition 简单的重复

Chances are you wrote something like this (leaving out the code that creates TurtleWorld and waits for the user):

假设你写了如下的代码（不包括生成TurtleWorld以及等待用户的代码）：

```
fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
```

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

我们可以用更简洁的for语句做相同的事情。 将这个示例加入mypolygon.py并重新运行它：

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

You should see something like this:

你会看到如下内容：

```
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，后面我们会看到更多的。 但是这对于你重写你的画正方形程序已经足够了。 如果没有完成不要继续。

Here is a `for` statement that draws a square:

下面是使用`for`语句画正方形:

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

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头以及一个缩进的体。 体可以包含任意条数的语句。

A `for` statement is sometimes 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`语句有时被成为**循环（loop）**，因为执行流程运行整个体然后循环回顶部。 在此例中，它运行体4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 a little 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.

此版本和前面画正方形的代码有所不同，因为它在花完正方形最后一条边的时候， 又多转了一下。这个额外的转动多花了些时间， 但是如果我们通过循环做相同的事情也简化了代码。 此版本也有一个作用，将海龟留着了出发位置，朝向开始的方向。

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.

下面是一系列使用TurtleWorld的练习。 他们是很有趣，但是他们也都有一点。 当你做这些联系的时候，想一想这个点是什么?

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

下面几节是这些练习的答案，因此如果你没完成（或者至少试过了），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.

   写一个名为`square`的函数，有一个名为`t`的形参，它是一个海龟。 用这只海龟画一个正方形。

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

   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。

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`.

   给`square`加入另一个名为`length`的形参。修改函数体，使得边的长度是`square`，然后修改函数调用以提供第二个实参。重新运行程序。用一个范围内的`length`值测试你的程序。

3. The functions `lt` and `rt` make 90-degree turns by default, but you can provide a second argument that specifies the number of degrees. For example, `lt(bob, 45)` turns `bob` 45 degrees to the left.

   函数`lt`和`rt`默认转动90度，而你可以提供第二个实参来制定角度。例如`lt(bob, 45)`向左转动`bob` 45读。

   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.

   复制`square`并改名为`polygon`。增加另外一个名为`n`的参数并改变函数体，让它画一个n边形。提示：n边型的外角是$360/n$度。

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

   写一个名为`circle`的函数，它接受一个乌龟`t`和半径`r`作为形参，然后通过调用`polygon`，用合适的边的数目和长度画一个近似圆形。用一定范围内的`r`值测试你的函数。

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

   提示：指出圆的周长并确定`length * n = circumference`。

   Another hint: if `bob` is too slow for you, you can speed him up by changing `bob.delay`, which is the time between moves, in seconds. `bob.delay = 0.01` ought to get him moving.

   另外一个提示：如果`bob`对你来说太慢了，你可以通过修改`bob.delay`加速，它是两次移动之间的时间（秒）。`bob.delay = 0.01`应该让它移动。

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.

   完成一个更一般的`circle`版本`arc`，其接受一个额外的参数`angle`，确定画多大部分的圆。`angle`的单位是度，因此当`angle=360`时，`arc` 应该画一个整圆。

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:

第一个练习需要你将画正方形的代码放到函数定义中然后调用该函数，传递海龟作为参数。这是答案:

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

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, so that is the end of both the `for` loop and the function definition.

最内部的语句，`fd`和`lt`缩进两次以显示它们在`for`循环内， 该循环在函数内。下一行`square(bob)`和左边界对齐， 因此它是`for`循环和函数定义的结束。

Inside the function, `t` refers to the same turtle `bob` refers to, so `lt(t)` has the same effect as `lt(bob)`. So 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`和`bob`指的是同一只海龟， 所以`lt(t)`和`lt(bob)`功能相同。 那么为什么不将形参命名为`bob`呢？ 想法是`t`可以是任何海龟而不仅仅是`bob`， 所以你可以生成第二只海龟并且将它作为实参传递给`square`：

```
ray = Turtle()
square(ray)
```

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 **encapsulation（封装）**。 封装的好处之一是它将这些代码赋予一个名字， 这充当了一种文档。另一个好处是你可以重用这些代码， 调用函数两次要比拷贝粘贴函数体要简洁！

4.5 Generalization 泛化

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

下一步是给`square`增加一个`length`形参。这是解决方案：

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

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）**， 因为其使得函数更一般：在前面的版本中， 正方形总是相同大小的；此版本它可以是任意的大小。

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

下一步也是一个泛化。不是画一个正方形，polygon画任意的多边形。 这是解决方案：

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```

This draws a 7-sided polygon with side length 70. If you have more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. It is legal, and sometimes helpful, to include the names of the parameters in the argument list:

这画了一个7边形，边长为70。 如果你有更多的形参，很容易忘记它们多是什么或者它们的顺序应该是什么。 将形参名放入实参列表是合法的，而且有时很有用：

```
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**）， 因为它们包含了形参名作为"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.

这一语法使得程序更可读。 它也是关于实参和形参如何工作的一个提醒： 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:

下一步是写`circle`，它将半径`r`作为形参。 这是一个简单的解决方案，它使用`polygon`画一个50边形：

```
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.

第一行用半径`r`计算圆的周长，公式是$2\pi r$。 由于我们用了`math.pi`，我们需要导入`math`。 按照惯例，`import`语句通常位于脚本开始。

`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是我们估计的圆的线段的个数，所以length是每一段的长度。 这样polygon画一个50边形来估计一个半径为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.

这种解决方案的一个局限性是n是常数，这意味着对于非常大的圆， 线段会非常长，并且对于小圆，我们浪费时间画非常小的线段。 一个解决方案是通过将n作为形参来泛化函数。 这将给用户（调用circle的人）更多的控制， 但是接口将不那么干净了。

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 is "as simple as possible, but not simpler. (Einstein)"

函数的**接口**（**interface**）是一个如何使用它的摘要： 形参是什么？函数做什么？返回值是什么？ 如果接口"尽可能简单，但不能过于简单（爱因斯坦）"的话，则说它是"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，因为它确定了被画的圆。 n就不太合适，因为他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:

不要把接口弄乱，依赖circumference选择一个合适的n值比较好。

```
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 (approximately) 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 appropriate for any size circle.

现在线段的数量（大概）是circumference/3， 所以每条线段的长度（大概）是3，这足够的小了， 使得圆看上去不错，但是对于保证效率也足够大了， 适合任何大小的圆。

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的时候，我能够重用polygon， 因为一个多边形是对圆的很好的近似。 但是arc不太易于合作，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polygon的一个拷贝开始， 并将它转化为arc。结果看上去像这样：

```
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 = float(angle) / n

    for i in range(n):
        fd(t, step_length)
        lt(t, 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不能重用`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！ 让我们称这个更一般的函数为`polyline`：

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, 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 function interfaces and facilitate code re-use—is called **refactoring**. In this case, we noticed that there was similar code in `arc` and `polygon`, so we "factored it out" into `polyline`.

这个过程—重新整理一个程序以改进函数接口和促进代码的重用性— 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"放入`polyline`。

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，我们可能首先写`polyline`并避免重构， 但是在一个项目开始的时候，你经常知道的不足，不能设计全部的接口。 一旦你开始编码，你更好的理解该问题。 优势重构是一个信号，说明你已经学到某些东西了。

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:

开发方案（**development plan**）是写程序的一个过程。 此例中我们使用的过程是"封装和泛化"。 此过程的步骤是：

1. Start by writing a small program with no function definitions.

   从写一个没有函数定义的小程序开始。

2. Once you get the program working, encapsulate it in a function and give it a name.

   一旦该程序运转起来，将它封装进一个函数并给它一个名字。

3. Generalize the function by adding appropriate parameters.

   通过增加适当的形参泛化该函数。

4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).

   重复1–3步，直到你有一些可运转的函数。 拷贝和粘贴代码以避免重新键入（和重新调试）。

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.

   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.

这个过程有一些缺点—后面我们将看到替代方案— 但是如果你之前不知道如果将程序分解为函数，它可能很有用。 该方法让你一边编程，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:

一个文档字符串（**docstring**）是位于函数开始的字符串， 其解释函数的接口（"doc"是"document"的缩写）。 这是一个例子：

```
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):
        fd(t, length)
        lt(t, angle)
```

This docstring is a triple-quoted string, also known as a multiline string because the triple quotes allow the string to span more than one line.

此文档字符串是一个三引号字符串，也被称为多行字符串， 因为三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).

它很简单，但是它包括了该函数的必要信息。 它简要的说明了该函数做什么（没有介绍它如何工作的细节）。 它说明了每个形参对函数的行为有什么作用以及每个形参应有的类型（如果它并不明显）。

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

写这种文档是接口设计很重要的部分。 一个良好设计的接口应该解释起来很简单， 如果你很难解释你的函数，这也许是该接口可能需要被改进的一个信号。

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.

接口就像是函数和一个调用者之间的合同。 调用者同意提供某些参数，函数统一做某些工作。

For example, `polyline` requires four arguments: `t` has to be a Turtle; `n` is the number of line segments, so it 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`需要4个实参：`t`必须是一个Turtle； `n`是线段的个数，因此它必须是一个整数； `length`应该是一个正数； `angle`必须是一个数，它被理解为一个度数。

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 in the World).

这些需求被称作**先决条件**（**preconditions**）， 因为它们在函数开始执行之前被假设为正确的。 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）以及任何侧面效果 （如移动Turtle或者World中的其它改变）。

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.

先决条件是调用者的职责。如果调用者违反一个（恰当的说明） 先决条件并且函数不正确的工作，则错误在调用者中，而不是函数中。

4.11 Glossary 术语表

instance （实例）: A member of a set. The TurtleWorld in this chapter is a member of the set of TurtleWorlds.

loop （循环）: A part of a program that can execute repeatedly.

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

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

keyword argument （关键字实参）: An argument that includes the name of the parameter as a "keyword."

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

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

development plan （开发方案）: A process for writing programs.

docstring （文档字符串）: A string that appears in a function definition to document the function's interface.

precondition （先决条件）: A requirement that should be satisfied by the caller before a function starts.

postcondition （后置条件）: A requirement that should be satisfied by the function before it ends.

4.12 Exercises

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

1. *Write appropriate docstrings for* `polygon`*,* `arc` *and* `circle`*.*
2. *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.*
3. *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 units 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.*

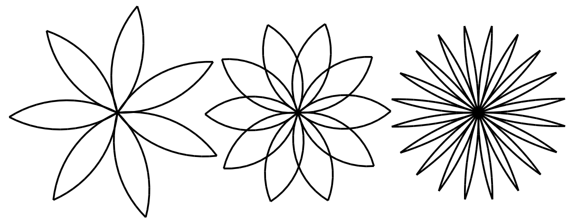

Figure 4.1: Turtle flowers.

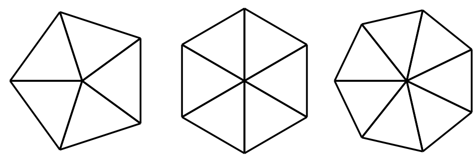

Figure 4.2: Turtle pies.

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

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

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

Solution: `http://thinkpython.com/code/pie.py`*.*

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 a font that can be drawn with a minimal number of basic elements and then write functions that draw letters of the alphabet.*

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://thinkpython.com/code/typewriter.py` *to help you test your code.*

Solution: `http://thinkpython.com/code/letters.py`*, also requires* `http://thinkpython.com/code/polygon.py`*.*

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://thinkpython.com/code/spiral.py`*.*

Chapter 5

Conditionals and recursion 条件和递归

5.1 Modulus operator 求余运算符

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

求余运算符（**modulus operator**）对整数进行运算并且产生第一个运算数和第二个的余数。 在Python中，求余运算符是一个百分号（%）。 语法和其它运算符相同：

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

So 7 divided by 3 is 2 with 1 left over.

因此7被3除得2剩余1.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

求余运算符非常有用。 例如你可以查看一个数是否能被另一个整除—如果`x % y`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.

同时，你能获得一个数的最右一位或者各位的数字。 例如`x % 10`产生`x`的最右一位数字（以10为底）。 相似的，`x % 100`产生最后两位数字。

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**）的结果为真或者假。下面的例子使用==运算符，它比较两个运算数， 如果它们相等，则产生`True`，否则产生`False`。

```
>>> 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`类型的特殊的值， 它们不是字符串。

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type '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， 其它的是：

```
x != y               # x is not equal to y（x和y不相等）
x > y                # x is greater than y（x比y大）
x < y                # x is less than y（x比y小）
x >= y               # x is greater than or equal to y（x大于或等于y）
x <= y               # x is less than or equal to y（x小于或等于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。 一个通常的错误是使用单独一个等号（=）而不是双等号（==）。 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的英语意思相似。 例如如果x大于0并且小于10，则`x > 0 and x < 10`为真

`n%2 == 0 or n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3. 只要有一个条件为真，也就是数字n能被2或者3整除， 则`n%2 == 0 or n%3 == 0`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`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，布尔运算符的运算数应该是布尔表达式， 但是Python并不严格。任何非0的数字都被解释成"真"。

```
>>> 17 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。 你可能需要避免它（除非你知道你正在做什么）。

5.4 Conditional execution 有条件的执行

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，我们几乎总是需要检测条件并相应的改变程序行为的能力。 **条件语句（Conditional statements）**给予我们这一能力。 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, then the indented statement gets executed. If not, nothing happens.

`if`之后的布尔表达式被称作**条件（condition）**。 如果它为真，则缩进的语句被执行。 如果不是，则什么也不好发生。

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

`if`和函数定义有相同的结构：一个头跟着一个缩进的体。 类似的语句被称作**合成语句（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.

体中可出现的语句数目没有限制，但是至少有一个。 有时候，没有任何语句的体会有用（通常对于你还没写的代码作一个占位）。 那样，你可以使用`pass`，它什么也不做。

```
if x < 0:
    pass          # 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 gets executed. The syntax looks like this:

`if`语句的第二种形式是**二选一之选那个**（**alternative execution**），此时有两个可能，条件决定哪一个被执行。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 a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. 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，并且程序对这一效果显示一个信息。如果条件为假，第2个语句集合被执行。既然条件只能是真或假，两个选择之一必被执行。此选择被称作**分支**（**branches**），因为它们是执行流的分支。

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条件并且我们需要多余两个的分支。一种表示像这样的计算的方法是**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 be executed. 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。再次，必有一个分支被执行。`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，但不是必须要只有一个。

```
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 executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

每个条件被按顺序检测。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。 即使不止一个条件为真，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 trichotomy example like this:

一个条件可以嵌到另一个里面。我们可以像这样写三分的例子：

```
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.

外面的条件包括两个分支。第一个分支包括一条简单的语句。 第二个分支包括另外一个`if`语句，它有自己的两个分支。 那两个分支都是简单的语句，当然它们也能有条件语句。

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, 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**嵌套条件**（**nested conditionals**） 很难快速的读它们。一般来讲，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:

逻辑运算符经常提供一个化简条件语句的方法。 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 is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

只有我们通过两个条件的时候，`print`语句才被执行， 因此我们能够获得和`and`运算符相同的效果：

```
if 0 < x and 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。为什么这一个和好事情可能不明显，但它是一个程序能做的最神奇的东西之一。例如，看一下下面的程序：

```
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，它输出单词"Blastoff!"。否则，它输出`n`然后调用一个名为`countdown`的函数—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...

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

> The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself...
>
> `countdown`开始以`n=2`执行，既然`n`大于0，它输出值2，然后调用它自己...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself...
> > `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...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, "Blastoff!" and then returns.
> > > `countdown`开始以`n=0`执行，既然`n`不大于0，它输出单词"Blastoff!"，然后返回。
> >
> > The `countdown` that got `n=1` returns.
> > 获得`n=1`的`countdown`返回。
>
> The `countdown` that got `n=2` returns.
>
> 获得`n=2`的`countdown`返回。

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

获得`n=3`的`countdown`返回。

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

然后你回到`__main__`中。因此整个输出类似于：

```
3
2
1
Blastoff!
```

A function that calls itself is **recursive**; the process 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.

作为另一个例子，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 are not executed.

如果`n <= 0`，`return`语句退出函数。 执行流马上返回到调用者，函数剩余的行不被执行。

The rest of the function is similar to `countdown`: if `n` is greater than 0, 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： 如果`n`比0大，它显示`s`并调用它自己来额外显示`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一个`for`循环很难写的例子， 所以早点儿开始有好处。

5.9 Stack diagrams for recursive functions 递归函数栈图

In Section 3.10, 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3.10节中，我们用一个栈图表示函数调用期间程序的状态。 相同类型的图能帮助理解一个递归函数。

Every time a function gets called, Python creates a new function frame, which contains 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.

一个函数每次被调用，Python生成一个新的函数框架， 其包括函数的局部变量和形参。 对于一个递归函数，在栈上可能同时有超过一个的框架。

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

图5.1展示了一个对于用`n = 3`调用`countdown`的栈图。

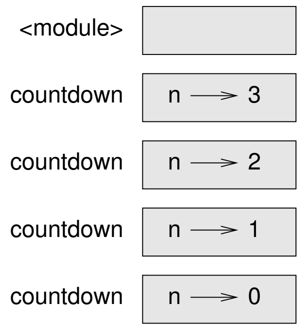

Figure 5.1: Stack diagram.

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，栈顶是`__main__`的框架。 因为我们在`__main__`中没有生成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.

对于形参`n`，4个`countdown`框架有不同的值。 栈底，其中`n=0`，被**基本条件**（**base case**）调用。 它不在进行递归调用了，所以没有更多的框架了。

Exercise 5.1. *Draw a stack diagram for* `print_n` *called with* `s = 'Hello'` *and* `n=2`*.*

Exercise 5.2. *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.*

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:

如果一个递归从不会到达基本条件，它永远进行递归调用， 并且程序从来不会终止。这被称作**无限递归**（**infinite recursion**）， 通常这不是一个好主意。这是一个具有无限递归的微型程序：

```
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大一点。 当错误出现的时候，在栈上有1000个递归框架。

5.11 Keyboard input 键盘输入

The programs we have written so far are a bit rude in the sense that they 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，从这个意义上讲有点儿粗鲁。 每次它们都只是做相同的事情。

Python 2 provides a built-in function called `raw_input` that gets input from the keyboard. In Python 3, it is called `input`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `raw_input` returns what the user typed as a string.

Python 2提供了一个被称为`raw_input`的内建函数，从键盘获得输入。 在Python 3中，它被称作`input`。 当此函数被调用时，程序停下来并且等待用户键入一些东西。 当用户按回车时，程序恢复执行并且`raw_input`作为字符串返回用户加入的内容。

```
>>> input = raw_input()
What are you waiting for?
>>> print input
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 input. `raw_input` can take a prompt as an argument:

在从用户哪儿获得输入之前，打印一个提示告诉用户输入什么是个好办法。 `raw_input`能接受一个提示作为实参。

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

提示最后的序列`\n`表示一个**新行**（**newline**）， 它是一个特别的字符，引起一个断行。 这也是为什么用户的输入出现在提示符的下面。

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 = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
```

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

但是，如果用户键入非数字字符串的其它东西，你获得一个错误：

```
>>> speed = raw_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()
```

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

我们后面将会看到如何处理这类错误。

5.12 Debugging 调试

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. The most useful parts are usually:

当一个错误发生时，Python显示的回溯包含许多信息， 但是它可能被淹没了，特别是在栈上有有许多框架时。 最有用的部分通常是：

- What kind of error it was, and

  错误是哪类，以及

- Where it occurred.

  它发生在哪儿。

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，但也有一些难度。 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会忽略它们。

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

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，问题是第二行被一个空格缩进了。 但是错误信息指向y，这是个误导。 通常，错误信息指向被发现错误的地方， 但是真实的错误可能在代码中的更早前的地方， 优势在前一行。

The same is true of runtime errors.

运行时错误也同样。

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:

假设你正试图给计算机键入一个信噪比。 公式是$SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$。 在Python中，你可能如此写：

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

But when you run it in Python 2, you get an error message.

但是，当你在Python 2中运行它的时候， 你获得一个错误信息。

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range 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, because dividing two integers does floor division. The solution is to represent signal power and noise power with floating-point values.

该错误信息指向第5行，但是那一行没什么错误。 为了找到真实的错误，打印`ratio`也许会有用，它变成了0。 问题是在第4行，因为两个整数的除法是向下取整除。 解决办法是用浮点值表示信号功率和噪声功率。

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

通常，错误信息告诉你问题在哪儿发现的，但那经常并是非引起错误的地方。

In Python 3, this example does not cause an error; the division operator performs floating-point division even with integer operands.

在Python 3中，此例不会引起一个错误。 即使是对整数运算数，除法运算符执行浮点除

5.13 Glossary 术语表

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

boolean expression（布尔表达式）: An expression whose value is either `True` or `False`.

relational operator（关系运算符）: One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

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

conditional statement（条件语句）: A statement that controls the flow of execution depending on some condition.

condition（条件）: The boolean expression in a conditional statement that determines which branch is executed.

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

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

chained conditional（链式条件）: A conditional statement with a series of alternative branches.

nested conditional（嵌套条件）: A conditional statement that appears in one of the branches of another conditional statement.

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

base case（基本条件）: A conditional branch in a recursive function that does not make a recursive call.

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

5.14 Exercises

Exercise 5.3. *Fermat's Last Theorem says that there are no integers a, b, and c such that*

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

for any values of n greater than 2.

1. *Write a function named* `check_fermat` *that takes four parameters—*`a`*,* `b`*,* `c` *and* `n`*—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that*

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

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

2. *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.*

Exercise 5.4. *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, it is clear that 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:*

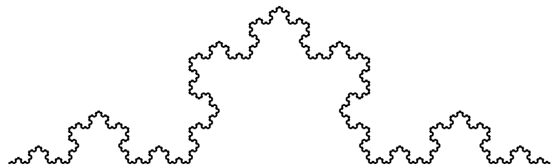

Figure 5.2: A Koch curve.

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.)

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. *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.*

The following exercises use TurtleWorld from Chapter 4:

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 4).*

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

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*

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 $x/3$.*

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

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://thinkpython.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.*

Chapter 6

Fruitful functions 有返回值的函数

6.1 Return values 返回值

Some of the built-in functions we have used, such as the math functions, produce results. Calling the function generates a value, which we usually assign to a variable or use as part of an expression.

有一些我们用过的内建函数，例如数学函数，产生结果。 调用此函数产生一个值，通常我们将其赋给一个变量或者作为一个表达式的一部分使用。

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

All of the functions we have written so far are void; they print something or move turtles around, but their return value is `None`.

目前为止所有我们写的函数都是无返回值的， 它们打印一些东西或者移动海龟，但是它们的返回值是`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:

在本章中，我们（最后）将写有返回值的函数。 第一个例子是`area`，给出半径，其返回一个圆的面积。

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

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。 这条语句意思是："马上从该函数返回并使用下面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 `temp` often make debugging easier.

另一方面，类似`temp`的**临时变量**（**temporary variables**）经常会使调试更容易些。

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

有时，有多个返回语句会很有用，在每个条件的分支内有一个。

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

Since these `return` statements are in an alternative conditional, only one will be executed.

既然这些`return`语句在不同的条件内，只有一个会被执行。

As soon as a return statement executes, 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。

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， 保证通过程序的每个可能的路径都遇到一个`return`语句是个好注意。 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.

该函数是不正确的，因为如果`x`恰好是`0`， 则没有条件为真，并且函数在没遇到任何`return`语句条件下终止。 如果执行流到达函数的结尾，则返回值是`None`， 它不是`0`的绝对值。

```
>>> print 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提供一个被称为`abs`的内建函数用来计算绝对值。

Exercise 6.1. *Write a* `compare` *function that returns* `1` *if* `x > y`*,* `0` *if* `x == y`*, and* `-1` *if* `x < y`*.*

6.2 Incremental development 增量式开发

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

当你写较大的函数时，你可能发现你自己花了大量的时间用于调试。

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.

为了对付越来越负责的程序， 你可能想试一种叫做**增量式开发**（**incremental development**）的过程。 增量式开发的目标是通过每次增加和测试一小部分代码，以避免调试长代码。

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:

作为一个例子，假设给定两个点的坐标(x_1, y_1)和(x_2, y_2)， 你想计算两个点之间的距离。通过毕达哥拉斯定力，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)?

第一步是考虑在Python中，一个`distance`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？

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

此例中，输入是可以用4个数表示的两个点。 返回值是距离，其是一个浮点数。

Already you can write an outline of the function:

因此，你可以写出此函数的轮廓：

```
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。 但它是语法上正确的，并且它能运行， 也就是说在你将它变得更复杂之前，你能够测试它。

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

为了测试这个新函数，使用样例实参调用它：

```
>>> 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.

我选择这些值以便水平距离是3，垂直距离是4。 这样结果是5（3-4-5三角形直角三角形的斜边）。 当测试一个函数时，知道正确的结果很有用。

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.

在这一点上，我们已经确定函数是语法正确的了， 我们可以在函数体中开始增加代码。 一个可行的下一步是找到$x_2 - x_1$和$y_2 - y_1$之间的不同。 下一个版本在临时变量中存储这些值并打印它们。

```
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.

如果此函数好使，它应该显示`'dx is 3'`以及`’dy is 4’`。 如果这样，我们知道此函数获得了正确的实参并且正确的执行第一步计算。 如果不是，只需检查很少的几行。

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

下面我们计算`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，你运行此程序并检查输出（它应该是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.

如果其正确工作，你完成了。 否则，你可能想在返回语句之前打印返回值。

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.

当该函数的最终版本运行的时候，其不显示任何东西，它值返回一个值。 我们写的`print`语句对于调试很有用，但是一旦你使得该寒素正确运行了， 你应该删除它们。

类似的代码被称作**脚手架（scaffolding）**， 因为它对建造程序很有用，但是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，你每次应该只增加一或两行代码。 随着你获得越来越多的经验，你可能发现你自己能写并调试更大的块了。 无论哪种方式，增量式开发能够节省你大量调试的时间。

The key aspects of the process are:

此过程的关键是：

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.

   从一个好使的程序开始并做小的增量式的修改。 如论何时，如果有一个错误，你应该知道它在哪儿。

2. Use temporary variables to hold intermediate values so you can display and check them.

   使用临时变量来存储中间值，因此你能显示并检查它们。

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.

   一旦程序正常运行，你可能想删掉一些脚手架或者将多条语句组成复合表达式， 但是只有这不会使程序难读懂时。

Exercise 6.2. *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 two legs as arguments. Record each stage of the development process as you go.*

6.3 Composition 组合

As you should expect by now, you can call one function from within another. This ability is called **composition**.

和你现在可能期望的一样，你可以从一个函数内部调用另一个函数。 这种能力被称作**组合（composition）**

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.

作为一个例子，我们将写一个接受两个点的函数， 圆心以及圆周上的一个点，然后计算此圆的面积。

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:

假设圆心点存储在变量`xc`和`yc`中，圆周点在`xp`在`yp`中。 第一步是计算圆的半径，其是两个点之间的距离。 我们刚写了一个函数`distance`来做这个：

```
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:

下一步是用那个半径计算一个圆的面积，我们也刚写过：

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

Encapsulating these steps in a function, we get:

将这些步骤封装在一个函数中，我们获得：

```
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`对于开发和调试很有用，但是一旦改程序工作了，我们可以通过组合函数调用，把它变得更简洁：

```
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:

函数能返回布尔值，将复杂的测试隐藏在函数中通常很方便。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`.

通常给布尔函数一个听起来像是/否问题的函数名，`is_divisible`返回`True`或`False`来指示是否`x`能被`y`整除。

Here is an example:

这是一个例子：

```
>>>   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，因此我们可以通过直接返回它更简洁的写出该函数：

```
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:

写一些类似的东西可能很诱人：

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

But the extra comparison is unnecessary.

但是额外的比较是不必要的。

Exercise 6.3. *Write a function* `is_between(x, y, z)` *that returns* `True` *if* $x \le y \le z$ *or* `False` *otherwise.*

6.5 More recursion 更多的递归

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 keyboard, mouse, disks, etc., but that's all).

我们已经覆盖了Python的一个小的子集， 但是当你知道该子集是全部的编程语言时你可能很感兴趣， 这意味着任何能被计算的东西都能用该语言表达。 过去写的任何程序都能用你已经学过的语言特点表示 （事实上，你可能需要一些命令来控制如键盘、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*.

证明这种说法是一个非凡的工作，首先由阿兰图灵完成，他是首批计算机科学家之一 （有些人认为他是一个数学家，但是很多早期的计算机科学家都开始是数学家）。 相应地，这被称作图灵理论。关于图灵理论更完整（和准确）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:

为了说明用目前学过的工具能做什么，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，在这个意义上定义包含一个指向已经被定义的事物的引用。 一个真的循环定义不是非常有用：

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

漩涡: 用一个形容词来描述漩涡状的东西。

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:

如果你看到词典中的定义，你可能很恼火。 另一方面：如果你查找用!符号表示的阶乘函数的定义， 你可能获得如下的东西：

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

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 $n-1$.

此定义说0的阶乘是1，任何其它值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!是3乘以2!， 它又是2乘以1!， 它又是1乘以0!。 将它放在一起，3!是3乘以2乘以1乘以1，是6。

If you can write a recursive definition of something, you can usually 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:

如果你能写一些东西的递归定义，你通常可以写一个Python程序来计算它。 第一步是决定形参应该是什么。在此例中应该很清楚的是`factorial`接受一个整数：

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

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

如果实参恰好是0，所有我们需要做的是返回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，这是一个很有趣的部分，我们必须进行递归调用来计算$n-1$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:

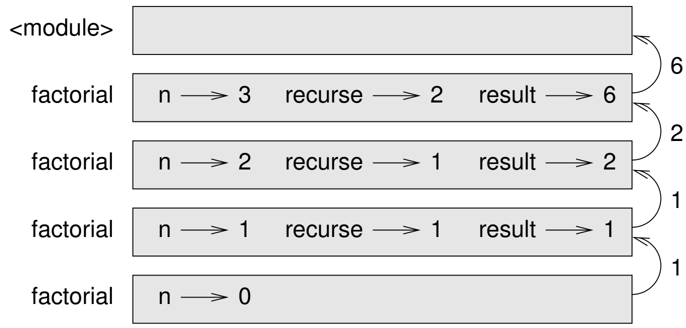

Figure 6.1: Stack diagram.

此程序的执行流程和5.8节中的`countdown`类似。 如果我们用值3调用`factorial`：

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

既然3不是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`...

既然2不是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`...

既然1不是0，我们执行第二个分支并计算`n-1`的阶乘...

Since 0 *is* 0, we take the first branch and return 1 without making any more recursive calls.

既然0是0，我们执行第一个分支并返回1，不进行任何递归调用。

The return value (1) is multiplied by n, which is 1, and the result is returned.

返回值(1)被与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)被与n（其为2）相乘，并返回结果。

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)被与n（其为3）相乘，并返回结果6，成为整个过程开始调用的函数的返回值。

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栈图看上去是什么样。

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`.

返回值被显示为传回到栈顶。在每个框架中，返回值是`result`的值， 其是`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 execute.

在最后一个框架中，局部变量`recurse`和`result`并不存在， 因为生成它们的分支并没有被执行。

6.6 Leap of faith 信心的飞跃

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. 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.

顺着执行流程读程序是一种方法，但是它可能很快变得错综复杂。 另一种是我称作“信心的飞跃”的方法。 当你遇到一个函数调用的时候，不是顺着执行流程， 而是假设该函数正确的工作并返回正确的结果。

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 的 时 候 ， 你 已 经 实 践 过 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。 你只是假设它们工作，因为写这些内建函数的人们都是好的程序员。

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 你 自 己 的 函 数 时 也 是 这 样 。 例 如 ， 在6.4节 中 ， 我 们 写 了 一 个 称 作`is_divisible`的函数， 它决定是否一个数能被另一个整除。 一旦我们相信该函数是正确的—通过检查代码和测试— 我们就能能使用该函数，而不用再看函数体了。

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 (yields the correct result) and then ask yourself, “Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?” In this case, 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，你应该假设次递归调用工作（产生正确的结果） 然后问你自己，“假设我能够获得$n-1$的阶乘，我能计算n的阶乘么？ ” 在此例中，很明显你能，通过乘以n。

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，当你没写完它的时候，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心的飞跃了！

6.7 One more example 一个更多的例子

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`):

factorial之后，最通常的一个被递归定义的数学函数是fibonacci， 其定义如下（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看起来是这样：

```
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，如果你试图跟随执行流程，即使对于相当小的n， 你的头也会爆炸。但是根据信心的飞跃，如果你假设这两个递归调用正确的工作， 那么很明显通过把它们加到一起，你获得了正确的结果。

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作为实参调用`factorial`会发生什么呢？

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

It looks like an infinite recursion. But how can that be? There is 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。但那是如何发生的的？ 有一个基本条件—当`n == 0`时。 但是如果`n`不是一个整数，我们会错过此基本条件并永远递归下去。

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.

在第一个递归调用中，`n`是0.5。下一次，它是-0.5。 从此，它变得越来越小（越来越负），但不会是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`检查它的实参的类型。 第一个选项被称作gamma函数，它有点儿超过本书的范围了。 所以我们将用第二种方法。

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。 此时，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 catches negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

第一个基本条件处理非整数，第二个抓住负整数。 在这两个条件中，程序打印一个错误信息并返回`None`以指明一些东西是错误的：

```
>>> 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, then 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， 因此我们可以保证递归终止。

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.

此程序演示了一个模式，有时称作**监护人**（**guardian**）。 前两个条件扮演监护人的角色，避免下面的代码使用引起错误的值。 该监护人使得验证代码的正确性成为可能。

In Section 11.3 we will see a more flexible alternative to printing an error message: raising an exception.

在11.3节中，我们将看到更灵活的方法来打印错误信息： 抛出一个异常。

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自然地为调试生成了检查点。 如果一个函数不工作，有三个可能需要考虑:

- 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错误，违反先决条件。

- There is something wrong with the function; a postcondition is violated.

  该函数有些错误，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错误。

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 来显示形参的值（也可能是它们的类型）。 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 that displays 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).

如果形参看起来很好，在每个`return`语句之前增加一条`print`语句， 来显示返回值。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6.2节中）。

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!).

如果该函数看起来工作，则看函数调用确保返回值被正确的使用（或者被用了！）

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束增加打印语句可以帮助执行流程更明显。 例如，这是`factorial`的具有打印语句的版本:

```
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(5)` :

`space`是一个空格字符的字符串，用来控制输出的缩进。 这是`factorial(5)`的结果:

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
```

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，这种输出可能有用。 开发有效的脚手架会花些时间，但是一点点的脚手架能够节省调试的时间。

6.10 Glossary 术语表

temporary variable（临时变量）： A variable used to store an intermediate value in a complex calculation.

dead code（死代码）： Part of a program that can never be executed, often because it appears after a `return` statement.

`None` **（空）：** A special value returned by functions that have no return statement or a return statement without an argument.

incremental development（增量式开发）： A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

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

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

6.11 Exercises

Exercise 6.4. *Draw a stack diagram for the following program. What does the program print? Solution:* `http://thinkpython.com/code/stack_diagram.py`.

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

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

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

Exercise 6.6. *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.*

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

```
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 8.

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.*

Solution: `http://thinkpython.com/code/palindrome_soln.py`.

Exercise 6.7. *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.*

Exercise 6.8. *The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.*

One way to find the GCD of two numbers is Euclid's algorithm, which 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* $gcd(a, 0) = a$*.*

Write a function called `gcd` *that takes parameters* `a` *and* `b` *and returns their greatest common divisor. If you need help, see* *http://en.wikipedia.org/wiki/Euclidean_algorithm*.

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

Chapter 7

Iteration 迭代

7.1 Multiple assignment 多次赋值

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).

如你可能已经发现的，对同一变量进行多次赋值是合法的。一个新的赋值使一个已有变量指向一个新的值（并停止指向旧的值）。

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

The output of this program is `5 7`, because the first time `bruce` is printed, its value is 5, and the second time, its value is 7. The comma at the end of the first `print` statement suppresses the newline, which is why both outputs appear on the same line.

程序的输出是`5 7`，因为`bruce`第一次被打印时，它的值是5，第二次值是7。第一条`print`语句结尾的逗号禁止产生新行，这也是为什么两个输出出现在同一行上。

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

图7.1显示在一个栈图中，**多次赋值（multiple assignment）**看起来是什么样子。

With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

有了多次赋值，区分赋值运算和相等语句就特别重要。因为Python使用等号（=）表示赋值，容易将类似`a = b`的语句理解成一个相等语句，但它不是！

First, equality is a symmetric relation 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.

首先，相等是一个对称关系但赋值不是。例如，在数学中，如果$a = 7$，那么$7 = a$。但是在Python中，`a = 7`语句是合法的，但`7 = a`不合法。

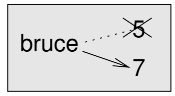

Figure 7.1: State diagram.

Furthermore, in mathematics, a statement 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:

更进一步，在数学中，一个相等语句总是或者为真或者为假。 如果现在$a = b$，那么a总是等于b。 在Python中，一条赋值语句可以使两个变量相等，但是它们不必总保持那样：

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

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

第三行改变了a的值，但是不改变b的值，因此它们不再相等。

Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

虽然多次赋值通常很有用，但是你应该小心使用它。 如果变量的值经常改变，可能使代码很难读和调试。

7.2 Updating variables 更新变量

One of the most common forms of multiple assignment is an **update**, where the new value of the variable depends on the old.

多次赋值的最常见形式之一是**更新**（**update**）， 变量的新值依赖于旧值。

```
x = x+1
```

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

这个意思是"获得x的当前值，加1，然后用新值更新x"。

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:

如果你试图更新一个不存在的变量，那么你会得到一个错误， 因为Python在给x赋值之前，先计算右边的值。

```
>>> 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:

在你可以更新一个变量之前，你必须**初始化**（**initialize**）它， 通常使用一个简单的赋值：

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

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

通过加1更新一个变量被称作**递增**（**increment**）， 减1被称作**递减**（**decrement**）。

7.3 The `while` statement `while`语句

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.

计算机经常被用于自动重复的任务。 无误地重复相同或者相似的任务是计算能做得很好，而人做得不好的事情。

We have seen two programs, `countdown` and `print_n`, that use recursion to perform repetition, which is also called **iteration**. Because iteration is so common, Python provides several language features to make it easier. One is the `for` statement we saw in Section 4.2. We'll get back to that later.

我们已经看过两个程序，`countdown`和`print_n`，它们使用递归来执行重复， 这也被称作**迭代**（**iteration**）。因为迭代是如此常见， 以至于Python提供了几个语言特性来使它变得容易。 一个是我们在4.2节中看到的`for`语句。 我们后面会回到它。

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

另一个是`while`语句。这是使用`while`语句的`countdown`版本：

```
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 reduce the value of `n` by 1. When you get to 0, display the word `Blastoff!`"

你几乎可以作为英语来读`while`语句。 它的意思是，"当`n`大于0时，显示`n`的值，然后将`n`的值减1。 当到达0的时候，显示单词`Blastoff!`"。

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

更正式地，这是`while`语句执行的流程：

1. Evaluate the condition, yielding `True` or `False`.

   求条件的值，产生`True`或者`False`。

2. If the condition is false, exit the `while` statement and continue execution at the next statement.

   如果条件为假，则退出`while`语句并继续执行下一条语句。

3. If the condition is true, execute the body and then go back to step 1.

   如条件为真，则执行循环体然后回到步骤1。

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

这类流程被称作**循环**（**loop**），因为第3步循环回顶部。

The body of the loop should change the value of one or more variables so that eventually the condition becomes false 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.

循环体应该改变一个或多个变量的值，以便最终条件变为假然后循环终止。否则，循环永远重复下去，这被称作**无限循环**（**infinite loop**）。对于计算机科学家一个消遣的无尽来源是观察香波的说明书，"打泡沫，漂洗，重复"，就是无限循环。

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

在`countdown`的例子中，我们可以证明循环能终止，因为我们知道`n`的值是有限的，并且我们可以看到通过循环，`n`的值每次都变小，所以最终我们一定获得0。在另一个例子中，就不这么简单了：

```
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.

该循环的条件是`n != 1`，所以循环将继续直到`n`是`1`，这使得条件为假。

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 sequence is 3, 10, 5, 16, 8, 4, 2, 1.

每次通过循环，程序输出`n`的值，然后检查它是偶数还是奇数。如果是偶数，`n`被2除。如果是奇数，`n`的值用`n*3+1`代替。例如，如果船体给`sequence`的实参是3，结果序列是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, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

既然`n`有时增大，有时减小，所以没有明显的证明`n`最终会到达1，或者程序终止。对于一些特殊的`n`值，我们能够证明终止。例如，如果起始值是2的幂指数，那么通过循环，`n`的值每次都是偶数直到它到达1. 前面的例子以16开始的这样的序列终止。

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`.)

难题是是否我们能够证明该程序对于所有正数n都会终止。 目前为止，没有人能证明它或者否定它！ （见：http://en.wikipedia.org/wiki/Collatz_conjecture）。

Exercise 7.1. *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.

有时你到循环的一半才知道该结束循环了。 在此情况下，你可以使用break语句跳出循环。

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

例如，假设你想接受来自用户的输入直到他们键入done。 你可以写成：

```
while True:
    line = raw_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.

该循环条件是True，它总是真的，所以直到遇到break语句， 循环将一直运行下去。

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:

每次循环，它用一个三角括号提示用户。 如果用户键入done，break语句退出循环。 否则程序回应用户键入的任何东西并且回到循环的顶部。 这是一个运行样例：

```
> 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.").

这种写while循环的方法很常见，因为你可以在循环中任何位置检查条件 （不是仅在顶部）并且你可以积极地表达停止条件（"当此发生时停止"）， 而不是消极地（"继续运行直到其发生"）。

7.5 Square roots 平方根

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

循环经常被用于计算数值结果的程序中，其开始于一个估计答案并迭代地改进它。

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:

例如，一种计算平方根的方法是牛顿法。假设你想知道a的平方根。 如果你用任意一个估计值x开始，你可以用下面的公式计算一个更好的估计值：

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

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

例如，如果a是4，x是3：

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

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

这和正确答案（$\sqrt{4} = 2$）更接近了。 如果我们用这个新的估计重复这个过程，它甚至更进了：

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

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

更多次的更新后，估计几乎是精确的：

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

一般来讲，我们事先不知道获得正确的答案需要多少步， 但是我们知道什么时候获得正确答案，因为估计值不再变化了：

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

```
>>> print 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:

当`y == x`时，我们可以停止。 下面是以一个初始估计`x`开始的循环，然后改进它直到它停止改变

```
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`.

对于大多数`a`其工作的很好，但是一般来讲测试`float`相等很危险。 浮点数值只是估计正确的：大多数有理数，像1/3，以及无理数， 像$\sqrt{2}$不能用`float`精确地表示。

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:

与其检查`x`和`y`是否精确地相等， 不如使用内建函数`abs`来计算它们之间不同的绝对或者数量级更安全。

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

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

其中`epsilon`的值类似`0.0000001`，其决定多近是足够近了。

Exercise 7.2. *Encapsulate this loop in a function called* `square_root` *that takes* `a` *as a parameter, chooses a reasonable value of* `x`*, and returns an estimate of the square root of* `a`.

7.6 Algorithms 算法

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).

牛顿法是一个**算法**（**algorithm**）的例子： 它是解决一类问题的机械化的过程（此例中是计算平方根）。

It is not easy to define an algorithm. 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.

定义算法并不容易。从一些不是算法的东西开始可能有帮助。 当你学习一位数相乘的时候，你可能记住乘法表。 实际上，你记下100个特定的答案。这类知识不是算法。

But if you were "lazy," you probably cheated by learning 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!

但是如果你“很懒”，你可能通过学习一些技巧来欺骗。 例如，为了计算n和9的乘积，你可以写下$n-1$作为第一位， $10-n$作为第二位。这种技巧是任意位数字和9相乘的一般解决方案。 这是一个算法！

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 in which each step follows from the last according to a simple set of rules.

类似的，你学过的进位加法、借位减法以及长除技术都是算法。 算法的特点之一是它们不需要任何智慧来执行。 它们是机械的过程，每一步根据简单的规则集，跟着上一步执行。

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

在我看来，人们在学校里花费如此大量的时间学习执行直白的， 不需要智力的算法真是很尴尬

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

另一方面，设计算法的过程是很有趣的， 充满智力的挑战并且是编程的核心部分。

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.

一些人们做得很自然、没什么难度或者有意识的思考的事情是最难用算法表示的。 理解自然语言是一个很好的例子。 我们都能做它，但是目前为止，没有人能解释我们是如何做的， 至少不能以算法的形式解释。

7.7 Debugging 调试

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 place for bugs to hide.

当你开始写较大的程序的时候，你可能发现你自己花了太多的时间在调试上。 更多的代码意味着更多的犯错机会以及更多隐藏错误的地方。

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.

一种减少调试时间的方法是“对分调试”。 例如，如果你的程序有100行，你每次检查一行，这将花费100步。

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.

相反，试着将问题对半分。在程序的中间部分或则附近看一下你可以检查的中间值。 增加一条`print`语句（或则其它有验证功能的东西）并运行程序。

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.

如果中间检查点不正确，则前半部分必然有一个问题。 如果正确，则问题在后半部分。

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.

每次你执行类似的检查，你将你需要搜索的行数分了一半。 经过6步（少于100），至少在理论上你将下到一两行代码。

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.

在实践中，"程序的中间部分"是什么通常不是很清楚， 并且检查它也不总是可能的。数行数并找到精确的中间点也没什么意义。 相反，要考虑程序中可能犯错的地方以及容易设置检查点的地方。 然后选择一个你认为错误会在其之前或之后的检查点。

7.8 Glossary 术语表

multiple assignment（多次赋值）： Making more than one assignment to the same variable during the execution of a program.

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

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

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

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

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

infinite loop（无限循环）： A loop in which the terminating condition is never satisfied.

7.9 Exercises

Exercise 7.3. *To test the square root algorithm in this chapter, you could compare it with* `math.sqrt`. *Write a function named* `test_square_root` *that prints a table like this:*

```
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 the function from Section 7.5; 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 7.4. *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)')
<type '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 7.5. *The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of π:*

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

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://thinkpython.com/code/pi.py`*.*

Chapter 8

Strings 字符串

8.1 A string is a sequence 字符串是一个序列

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

一个字符串是一个字符的**序列**（**sequence**）。 你可以用括号运算符一次访问一个字符:

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

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

第2条语句从`fruit`中选择编号为1的字符并将它赋给`letter`。

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

括号中的表达式被称作**索引**（**index**）。 索引指出在序列中你想要哪个字符（因此而得名）。

But you might not get what you expect:

但是你可能不会获得你期望的东西:

```
>>> print 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.

对于大多数人，`'banana'`的第一个字母是`b`而不是`a`。 但是对于计算机科学家，索引是从字符串开始的偏移量， 并且第一个字母的偏移量是0。

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

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

所以b是'banana'的第0个字母（“zero-eth”）， a是第1个字母（“one-eth”），n是第2个字母（“two-eth”）。

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

你可以使用任何表达式，包括变量名和运算符，都可以做为索引， 但是索引的值必须是整数。否则你获得：

```
>>> 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:

len是一个内建函数，其返回字符串中的字符数：

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

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

为了获得一个字符串的最后一个字符，你可能想尝试像这样：

```
>>> 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:

IndexError的理由是在'banana'中没有索引为6的字母。 既然我们从0开始数数，6个字母的编号是0到5。 为了获得最后一个字符，你必须从length中减1。

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

Alternatively, 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.

另一种作法是你可以使用负索引，其从字符串的结尾往后数。 表达式fruit[-1]产生最后一个字母，fruit[-2]产生倒数第二个字母，等等。

8.3 Traversal with a for loop 使用for循环遍历

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:

许多计算每次处理一个字符串的字符。它们经常从头开始，依次选择每个字符，对其做一些工作，然后继续直到结束。词处理模式被称作**遍历**）**traversal**）。一种写遍历的方法是用while循环：

```
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 is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

该循环遍历字符串并显示在每行显示一个字符串。该循环的条件是`index < len(fruit)`，所以当`index`和字符串的长度相等时，条件为假，并且循环体不被执行。被访问的最后一个字符的索引为`len(fruit)-1`，这是字符串的最后一个字符。

Exercise 8.1. *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:

另一种写遍历的方法是用`for`循环：

```
for char in fruit:
    print char
```

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

每次通过循环，字符串中的下一个字符被赋给变量`char`。循环继续，直到没有剩余的字符串了。

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:

下面的例子显示如何使用叠加（字符串加）和`for`循环生成一个字母系列（以字母序）。在Robert McCloskey的书《Make Way for Ducklings》中，小鸭子的名字是Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack。此循环按顺序输出这些名字：

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

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

The output is:

输出是：

```
Jack
Kack
Lack
Mack
Nack
```

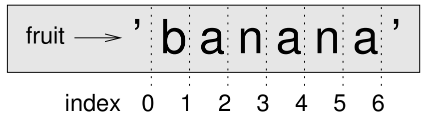

Figure 8.1: Slice indices.

```
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

当然，这不是非常正确，因为"Ouack"和"Quack"被错误拼写了。

Exercise 8.2. *Modify the program to fix this error.*

8.4 String slices 字符串片段

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

一段字符串被称作**片段**（**slice**）。选择一个片段类似于选择一个字符：

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print 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.

`[n:m]`操作符返回从第n个字符到第m个字符的部分字符串，包括第一个，但是不包括最后一个。这个行为违反直觉，但是它可能会帮助想象指向这两个字符之间的索引，如图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:

如果你省略第一个索引（冒号前面的），片段起始于字符串开始。如果你省略第二个索引，片段一直到字符串结尾：

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

如果第一个索引大于或等于第二个，结果是**空字符串**（**empty string**），表示为两个引号：

```
>>> 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.

一个空字符串不包括字符而且长度为0，但除此之外， 它和其它任何字符串一样。
Exercise 8.3. *Given that* `fruit` *is a string, what does* `fruit[:]` *mean?*

8.5 Strings are immutable 字符串是不可变的

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:

在一个赋值的左边使用`[]`很有诱惑力，意图是改变字符串的一个字符。 例如：

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 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. An **item** is one of the values in a sequence.

此例中的"object（对象）"是该字符串，"item（项）"是你要赋值的字符。 到目前，一个**对象**（**object**）和值是同一样的东西， 但是我们后面将改善此定义。一个**项**（**item**）是序列中的一个值。

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:

此错误的原因是字符串是**不可变的**（**immutable**）， 这意味着你不能改变一个已存在的字符串。 最好是生成一个新的字符串，它是原字符串的一个变种：

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

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

此例连接一个新的第一个字母到`greeting`的一个片段上。 它不影响原字符串。

8.6 Searching 搜索

What does the following function do?

下面的函数做什么？

```
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 opposite 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`.

在某种意义上，`find`和`[]`运算符相反。 不是接受一个索引并抽取相应的字符， 它接受一个字符并找到该字符出现的位置的索引。 如果没有找到该字符，函数返回`-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.

这是我们已经见过的第一个`return`语句在循环内部的例子。 如果`word[index] == letter`，函数停止循环并马上返回。

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

如果字符没出现在字符串中，那么程序正常退出循环并返回`-1`。

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

这种计算的模式—遍历一个序列并在我们找到我们正在找的东西时返回— 被称作**搜索**（**search**）。

Exercise 8.4. *Modify* `find` *so that it has a third parameter, the index in* `word` *where it should start looking.*

8.7 Looping and counting 循环和计数

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

下面的程序计算字母`a`在字符串中出现的次数：

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

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.

此程序演示另一种被称作**计数器**（**counter**）的计算模式。 变量`count`初始化为0然后每次出现`a`时递增。 当循环结束时，`count`包含结果—`a`的总数。

Exercise 8.5. *Encapsulate this code in a function named* `count`*, and generalize it so that it accepts the string and the letter as arguments.*

Exercise 8.6. *Rewrite this function so that instead of traversing the string, it uses the three-parameter version of* `find` *from the previous section.*

8.8 String methods 字符串方法

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:

方法（**method**）和函数类似—接受实参并返回一个值—但是语法不同。 例如，**upper**方法接受一个字符串并返回一个新的都是大写字母的字符串：

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

不是用函数的语法`upper(word)`，而是用方法的语法`word.upper()`。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print 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 argument.

点号的形式指出方法的名字，`upper`，以及应用该方法的字符串的名字，`word`。 空括号指出该方法不接受实参。

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

方法的调用被称作**调用**（**invocation**），在此例中， 我们说我们整在`word`上调用`upper`。

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

事实上，有一个被称为`find`的字符串方法， 其和我们写的函数异常相似：

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

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

此例中，我们在`word`上调用`find`并将我们要找的字母作为实参。

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

事实上，`find`方法比我们的函数更通用，它可以找到子串而不仅仅是字符：

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

It can take as a second argument the index where it should start:

它可以接受从何处开始的索引作为第二个实参：

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

And as a third argument the index where it should stop:

在哪个索引结束作为第三个实参。

```
>>> 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`).

此搜索失败是因为`b`没有出现在从`1`到`2`的索引之间 （不包括`2`）。

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

Exercise 8.8. *Read the documentation of the string methods at* `docs.python.org/lib/string-methods.html`*. 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.*

8.9 The `in` operator `in`运算符

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

单词`in`是一个布尔运算符，其接受两个字符串， 如果第一个作为子串出现在第二个中则返回`True`：

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

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

例如，下面的函数打印即出现在`word1`中也出现在`word2`中的字母：

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

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."

使用精心挑选的变量名，Python有时候读起来像是英语。 你可以读此循环，"对于（每个）在（第一个）单词中的字母， 如果（该）字母 （出现）在（第二个）单词中，打印（该）字母"。

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

如果你比较apples和oranges，这是你获得的东西：

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

8.10 String comparison 字符串比较

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

关系运算符在字符串上也工作。为了看两个字符串是否相等：

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

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

其它的关系运算符对于按字母序放置单词也很有用：

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

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

Python用和人不同的方式处理大写和小写字母。 所有的大写字母出现在所有小写字母之前，所以：

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

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.

解决此问题的通常的方式是在执行比较之前， 将字符串转化为标准格式，例如都是小写字母。 一旦你必须保卫自己免受一名手持菠萝的男子的袭击，记住这一点。

8.11 Debugging 调试

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:

当你使用索引在一个序列中遍历值的时候， 正确的获得遍历的开始和结束是一个技巧。 这是一个函数，其被假设用来比较两个单词， 如果一个单词是另一个的倒序，则返回真， 但是它包含两个错误：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
```

```
    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

The first `if` statement checks whether the words are the same length. If not, we can return `False` immediately and then, 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.

第一条`if`语句检查两个单词是否等长。 如果不是，我们可以马上返回假，然后对于函数其余的部分， 我们可以假设单词是等长的。 这是6.8节中的监护人模式的一个例子。

`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`.

`i`和`j`是索引：`i`向前遍历`word1`，`j`向后遍历`word2`。 如果我们找到两个不匹配的字母，我们可以立即返回假。 如果我们通过整个循环并且所有字母都匹配，我们返回真。

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

如果我们用单词"pots"和"stop"测试该函数， 我们期望返回真， 但是我们得到一个IndexError。

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

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.

为了调试该类错误，我的第一步是在错误出现的行之前，马上打印索引的值。

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

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

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

现在，当我再次运行该程序时，我获得更多的信息：

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

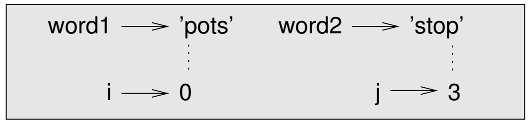

Figure 8.2: State diagram.

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.

第一次通过循环，j的值是4，其超出字符串'pots'的范围了。 最后一个字符的索引是3，所以j的初始值应该是len(word2)-1。

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

如果我修正了这个错误并重新运行程序，我获得：

```
>>> 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 shows in Figure 8.2.

这次，我获得正确的答案，但是看起来循环只运行了三次，这很奇怪。 为了更好的理解发生了什么，画出栈图会很有用。 在第一次循环期间，is_reverse的框架显示在图8.2中。

I took a little 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.

我安排了框图中变量的位置并且增加了虚线展示i和j的值来指明 word1和word2中的字符。

Exercise 8.9. *Starting with this diagram, execute 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 术语

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

sequence（序列）: An ordered set; that is, a set of values where each value is identified by an integer index.

item（项）: One of the values in a sequence.

index（索引）: An integer value used to select an item in a sequence, such as a character in a string.

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

empty string（空字符串）: A string with no characters and length 0, represented by two quotation marks.

immutable（不可变的）: The property of a sequence whose items cannot be assigned.

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

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

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

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

invocation（调用）: A statement that calls a method.

8.13 Exercises

Exercise 8.10. *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 6.6.*
Exercise 8.11. *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 8.12. *ROT13 is a weak form of encryption that involves "rotating" each letter in a word by 13 places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' shifted by 3 is 'D' and 'Z' shifted by 1 is 'A'.*

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

For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed".

You might want to use the built-in functions `ord`*, which converts a character to a numeric code, and* `chr`*, which converts numeric codes to characters.*

Potentially offensive jokes on the Internet are sometimes encoded in ROT13. If you are not easily offended, find and decode some of them. Solution: `http://thinkpython.com/code/rotate.py`.

Chapter 9

Case study: word play 案例分析：word play

9.1 Reading word lists 读取单词列表

For the exercises in this chapter we need a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project (see `http://wikipedia.org/wiki/Moby_Project`). It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is `113809of.fic`; you can download a copy, with the simpler name `words.txt`, from `http://thinkpython.com/code/words.txt`.

对于本章的习题，我们需要一个英语单词的列表。 互联网上有许多单词列表，但是最适合我们目的之一的列表是由 Grady Ward收集并贡献给公众的，并成为Moby词典项目的一部分（见：`http://wikipedia.org/wiki/Moby_Project`）。 它是一个由113,809个填字游戏单词组成的列表， 也就是在填字游戏以及其它文字有戏中被认为合理的单词。 在Moby集合中， 文件名是`113809of.fic`， 你可以从`http://thinkpython.com/code/words.txt`下载一个拷贝， 其使用了一个简化的名字`words.txt`。

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function `open` takes the name of the file as a parameter and returns a **file object** you can use to read the file.

该文件是纯文本，所以你可以用一个文本编辑器打开它， 但是你也可以从Python中读它。 内建函数open接受文件名作为形参，并返回一个 **文件对象）file object**）， 你可以使用它读取该文件。

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

`fin` is a common name for a file object used for input. Mode `'r'` indicates that this file is open for reading (as opposed to `'w'` for writing).

`fin`是输入文件对象的一个常用名。 模式`'r'`表明该文件为了读取而打开 （相反，`'w'`是为了写）。

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:

该文件对象为读取提供了几个方法，包括`readline`， 其从文件中读取字符直到到达新行并将结果作为字符串返回：

```
>>> fin.readline()
'aa\r\n'
```

The first word in this particular list is "aa," which is a kind of lava. The sequence `\r\n` represents two whitespace characters, a carriage return and a newline, that separate this word from the next.

在此列表中，第一个单词是"aa"，它是一种岩浆。 序列`\r\n`代表两个空白字符，回车和换行， 其将此单词和下一个分开。

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

此文件对象跟踪它在文件中的位置， 所以如果你再次调用`readline`，你获得下一个单词：

```
>>> fin.readline()
'aah\r\n'
```

The next word is "aah," which is a perfectly legitimate word, so stop looking at me like that. Or, if it's the whitespace that's bothering you, we can get rid of it with the string method `strip`:

下一个单词是"aah"，它是一个完全合法的单词， 所以不要那样看我。 或者，如果空格困扰了你，我们可以用字符串方法`strip`删掉它：

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print word
aahed
```

You can also use a file object as part of a `for` loop. This program reads `words.txt` and prints each word, one per line:

你也可以将文件对象用做`for`循环的一部分。 此程序读取`words.txt`并打印每个单词，每行一个：

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

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

9.2 Exercises 习题

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

下一节有这些习题的答案。 在你看这些答案之前，应该至少试这解决一下每个题目。

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 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 搜索

All of the exercises in the previous section have something in common; they can be solved with the search pattern we saw in Section 8.6. The simplest example is:

前一节的所有习题有一个共性，它们可以用我们在8.6节中看到的搜索模式解决。 最简单的例子是：

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

The `for` loop traverses the characters in `word`. If we find the letter "e", we can immediately return `False`; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn't find an "e", so we return `True`.

`for`循环遍历`word`中的字符。 如果我们找到字母"e"，那么我们可以马上返回`False`； 否则我们不得不到下一个字母。 如果我们正常停止循环，这意味着我们没有找到一个"e"， 所以我们返回`True`。

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

`avoids`是`has_no_e`的更一般的版本，但是它有相同的结构：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return `False` as soon as we find a forbidden letter; if we get to the end of the loop, we return `True`.

一旦我们找到一个禁止字母，我们返回`False`； 如果我们到达循环结尾，我们返回`True`。

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

除了条件的意思相反外，`uses_only`也是相似的：

```
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`.

不是有一个禁止字母的列表，而是我们有一个允许字母的列表。 如果我们在`word`中找到一个不在`available`中的字母， 我们可以返回`False`

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

除了我们翻转了单词和字母字符串的角色外，`uses_all`也类似：

```
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`.

不是在`word`中遍历字母，该循环遍历需要的字母。 如果任何需要的字母没出现在单词中，则我们返回`False`。

If you were really thinking like a computer scientist, you would have recognized that `uses_all` was an instance of a previously-solved problem, and you would have written:

如果你真的像计算机科学家一样思考， 你可能已经意识到`uses_all`是前面已经解决的问题的一个实例， 你可能会写成：

```
def uses_all(word, required):
    return uses_only(required, word)
```

This is an example of a program development method called **problem recognition**, which means that you recognize the problem you are working on as an instance of a previously-solved problem, and apply a previously-developed solution.

这是一个被称作**问题识别**（**problem recognition**）的程序开发方法的实例， 意思是你将正在解决的问题看做是之前已经解决的问题的一个实例， 并用用之前开发的解决方案。

9.4 Looping with indices 使用索引的循环

I wrote the functions in the previous section with for loops because I only needed the characters in the strings; I didn't have to do anything with the indices.

前一节我用for循环写函数，因为我只需要字符串中的字符， 我不必用索引做任何事情。

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

对于is_abecedarian，我们必须比较邻接的字母， 这是一个用for循环的小技巧。

```
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:

另外的替代方法是使用递归：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a while loop:

另一个选择是使用while循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

The loop starts at i=0 and ends when i=len(word)-1. Each time through the loop, it compares the 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).

循环起始于i=0，终止于i=len(word)-1。 每次循环比较第i个字符（我们可以将其认为是当前字符） 和第$i+1$个字符（我们可以将其认为是下一个字符） 。

If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.

如果下一个字符比当前的小（字母序靠前）， 那么我们在递增趋势中找到了停止点并返回False。

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like 'flossy'. The length of the word is 6, so the last time the loop runs is when i is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

如果到达循环结束，我们也没有找到一点错误，那么该单词通过测试。 为了说服你自己循环正确的结束了，考虑一个类似'flossy'的例子。 其长度为6，因此最后一次循环运行时，i是4，这是倒数第2个字符。 最后一次迭代，它比较倒数第二个和最后一个字符，这正是我们希望的。

Here is a version of is_palindrome (see Exercise 6.6) that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

这是is_palindrome的一个版本（见练习6.6）， 其使用两个索引，一个从最前面开始并往前上， 另一个从最后面开始并往下走。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Or, if you noticed that this is an instance of a previously-solved problem, you might have written:

或者，如果你注意到这是一个之前已经解决问题的一个实例， 你可能已经写成:

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

Assuming you did Exercise 8.9.

假设你做了练习8.9。

9.5 Debugging 调试

Testing programs is hard. The functions in this chapter are relatively easy to test because you can check the results by hand. Even so, it is somewhere between difficult and impossible to choose a set of words that test for all possible errors.

测试程序很难。本章的函数相对容易测试，因为你可以手工检查结果。 即使这样，选择一个单词的集合来测试所有可能的错误， 在某些方面也是介于困难和不可能的。

Taking has_no_e as an example, there are two obvious cases to check: words that have an 'e' should return False; words that don't should return True. You should have no trouble coming up with one of each.

例如has_no_e，有两个明显的用例需要检查： 含有'e'的单词应该返回False，不含的单词应该返回True。 对任何一个你应该都不会有麻烦。

Within each case, there are some less obvious subcases. Among the words that have an "e," you should test words with an "e" at the beginning, the end, and somewhere in the middle. You should test long words, short words, and very short words, like the empty string. The empty string is an example of a **special case**, which is one of the non-obvious cases where errors often lurk.

在每个用例中，还有一些不明显的子用例。 在含有"e"的单词中，你应该测试"e"在开始、结尾以及在中间的单词。 你应该测试长单词、短单词以及非常短的单词，如空字符串。 空字符串是**特殊用例**（**special case**）的一个例子， 其是一个经常隐藏错误的不明显的用例。

In addition to the test cases you generate, you can also test your program with a word list like `words.txt`. By scanning the output, you might be able to catch errors, but be careful: you might catch one kind of error (words that should not be included, but are) and not another (words that should be included, but aren't).

除了你生成的测试用例，你也可以用一个类似`words.txt`的单词列表测试你的程序。 通过扫描输出，你可能会捕获错误，但是请小心： 你可能捕获一类错误（包括了不应该包括的单词） 但不会捕获另一类错误（没有包括应该包括的单词）。

In general, testing can help you find bugs, but it is not easy to generate a good set of test cases, and even if you do, you can't be sure your program is correct.

一般来讲，测试能帮助你找到错误， 但是生成好的测试用例的集合并不容易， 并且即便你做到了，你仍然不能保证你的程序是正确的。

According to a legendary computer scientist:

据一个传奇计算机科学家所说：

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> 程序测试能被用于展现错误的存在，但是从不会显示其不存在！
>
> — Edsger W. Dijkstra

9.6 Glossary 术语表

file object（文件对象）: A value that represents an open file.

problem recognition（问题识别）: A way of solving a problem by expressing it as an instance of a previously-solved problem.

special case（特殊用例）: 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/puzzler/transcripts/200725):*

> *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://thinkpython.com/code/cartalk1.py`.
Exercise 9.8. *Here's another* Car Talk *Puzzler (*`http://www.cartalk.com/content/puzzler/transcripts/200803`*):*

> *"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://thinkpython.com/code/cartalk2.py`.
Exercise 9.9. *Here's another* Car Talk *Puzzler you can solve with a search (*`http://www.cartalk.com/content/puzzler/transcripts/200813`*):*

> *"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://thinkpython.com/code/cartalk3.py`.

Chapter 10

Lists 列表

10.1 A list is a sequence 列表是一个序列

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**.

像字符串一样，一个**列表**（**list**）是值的序列。在字符串中，值是字符；在列表中，它们可以是任意值。列表中的值被称作**元素**（**elements**）或者**项**（**items**）。

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

有几种方法生成一个新的列表；最简单的是把这些元素括在方括号中（`[` 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:

第一个例子是一个四个整数的列表。第二个是三个字符串。列表的元素不必是相同的类型。下面的列表包括一个字符串、一个浮点数、一个整数和（什么！）另一个列表：

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

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

在另一个列表中的列表是**嵌套**（**nested**）。

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

不包括任何元素的列表被称作空列表。你可以用空括号，`[]`，生成一个空列表。

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

如你所期望的，你可以将列表的值赋给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

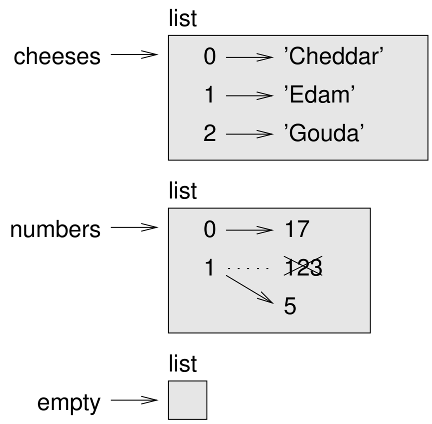

Figure 10.1: State diagram.

10.2 Lists are mutable 列表是可变的

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:

访问列表元素的语法和访问字符串的字符的语法相同—括号运算符。 括号内的表达式制定索引。记住索引以0开始:

```
>>> print 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.

和字符串不同，列表是可变的。 当括号运算符出现在赋值的左侧时，它识别出将被赋值的列表的元素。

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

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

`numbers`的第一个元素，过去是123，现在是5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index "maps to" one of the elements. Figure 10.1 shows the state diagram for `cheeses`, `numbers` and `empty`:

你可以将列表看成索引和元素之间的关系。 此关系被称作**映射**（**mapping**），每个索引"映射"到一个元素。 图10.1展示了`cheeses`、`numbers`和`empty`的栈图:

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.

列表用盒子表示，外面写着"list"里面是元素。 cheeses指向一个有三个元素的列表，索引是0、1和2. numbers包括两个元素，此图显示第二个元素的值已经被从123重新赋成了5. empty指向一个没有任何元素的列表。

List indices work the same way as string indices:

列表索引和字符串索引作用相同：

- Any integer expression can be used as an index.
  任何整数表达式可以被用作索引。
- If you try to read or write an element that does not exist, you get an IndexError.
  如果你试图读或写一个不存在的元素，你获得一个IndexError。
- If an index has a negative value, it counts backward from the end of the list.
  如果一个索引是一个负值，它从列表的结尾往回数。

The in operator also works on lists.

in运算符也可以用于列表。

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

10.3 Traversing a list 遍历一个列表

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

遍历一个列表元素的最常用方法是用for循环。 语法和遍历字符串相同：

```
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 functions range and len:

如果你只想读列表的元素，这工作的很好。 但是如果你想写或者更新元素，那么你需要索引。 一个常用的方法是组合range和len函数：

```
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.

该循环遍历列表并更新每个元素。 len返回列表中元素的数目。 range返回一个从0到$n-1$的索引的列表， 其中n是列表的长度。 循环i每次获得下一个元素的索引。 循环体内的赋值语句使用i来读旧的元素的值并赋予新值。

A for loop over an empty list never executes the body: 空列表上的for循环从不执行循环体：

```
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:

虽然一个列表可以包括另一个列表， 但是嵌套的列表仍然像一个单一元素一样计数。 该列表的长度是4：

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

10.4 List operations 列表运算

The + operator concatenates lists:

+运算符叠加列表：

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

Similarly, the * operator repeats a list a given number of times:

类似地，*运算符重复列表给定次数：

```
>>> [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.

第一个例子重复`[0]` 4次。 第二个例子重复列表`[1, 2, 3]` 3次。

10.5 List slices 列表片段

The slice operator also works on lists:

片段运算符也能用于列表：

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

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.

如果你不写第一个索引，片段起始于开始。 如果不写第二个，片段直到结束。 所以，如果你两个都不写，片段拷贝整个列表。

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

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.

既然列表是可变的，那么在对列表进行操作之前拷贝它通常很有用。

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

赋值语句左侧的片段运算符可以更新多个元素：

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

10.6 List methods 列表方法

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

Python提供在列表上运算的方法。 例如`append`在列表结尾增加一个新元素：

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

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

`extend`将一个列表作为实参并追加所有的元素：

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

This example leaves `t2` unmodified.

此例不改变`t2`。

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

`sort`从低到高排列列表的元素：

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

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

列表方法都是无返回值的，它们修改列表并返回`None`。 如果你偶然写了`t = t.sort()`，你将对结果很失望。

10.7 Map, filter and reduce 映射、过滤和削减

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

为了累加列表中的所有的数，你可以使用如下的循环：

```
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`初始化为0。每次循环，`x`从列表中获得一个元素。 +=运算符提供了一个简化的方法来更新一个变量。 此**增量赋值语句（augmented assignment statement）**：

```
    total += x
```

is equivalent to:

等价于：

```
    total = total + x
```

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

当循环执行时，`total`累计元素的和， 这样使用的变量有时被称作一个**累加器（accumulator）**。

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

累加一个列表的元素是如此通常的运算， 以至于Python为其提供了一个内建函数，`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**.

像这样将一序列元素组合成一个单一值的运算有时被称作**消减（reduce）**。

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

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:

有时，你想一边遍历一个列表，一边创建另一个。 例如，下面的函数接受一个字符串列表并返回一个新的包含大写字母字符串的列表：

```
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.

res用一个空列表初始化，循环每次追加下一个元素。 所以res是另一类累加器。

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.

类似capitalize_all的运算有时被称作a **映射（map）**， 因为它"映射"一个函数（此例中是capitalize方法）到一个序列的每个元素上。
Exercise 10.2. *Use* capitalize_all *to write a function named* capitalize_nested *that takes a nested list of strings and returns a new nested list with all strings capitalized.*

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:

另一个常见运算是从列表中选择一些元素并返回子列表。 例如，下面的函数接受一个字符串列表并返回只包含大写字母的字符串：

```
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.

isupper是一个字符串方法， 如果字符串只包含大写字母，其返回True。

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

类似only_upper的运算被称作**过滤（filter）**， 因为它选择一些元素并过滤掉其它的。

Most common list operations can be expressed as a combination of map, filter and reduce. Because these operations are so common, Python provides language features to support them, including the built-in function map and an operator called a "list comprehension."

最通常的列表运算能被表示成映射、过滤和消减的组合。 因为这些运算如此通常，以至于Python提供了支持它们的语言特征， 包括内建函数map以及一个被称作"列表理解"的操作符。
Exercise 10.3. *Write a function 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, the cumulative sum of* [1, 2, 3] *is* [1, 3, 6]*.*

10.8 Deleting elements 删除元素

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

有几种从列表中删除元素的方法。 如果你知道你想删除的元素的索引，你可以使用pop：

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

pop改变列表并返回被删除的元素。 如果不提供索引，它删除并返回最后一个元素。

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

如果你不需要被删除的值，你可以使用del运算符：

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

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

如果你知道你想删除的元素（但是不知道索引）， 你可以使用remove：

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

The return value from remove is None.

remove的返回值是None。

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

为了删除多余一个的元素，你可以使用del和一个片段索引：

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

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

和通常一样，该片段选择所有高于，但是不包含第二个索引的元素。

Exercise 10.4. *Write a function called* middle *that takes a list and returns a new list that contains all but the first and last elements. So* middle([1,2,3,4]) *should return* [2,3].

Exercise 10.5. *Write a function called* chop *that takes a list, modifies it by removing the first and last elements, and returns* None.

10.9 Lists and strings 列表和字符串

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:

一个字符串是字符的序列，列表是值的序列， 但是字符列表和字符串不一样。 为了将一个字符串转化为字符列表，你可以使用list：

```
>>> s = 'spam'
>>> t = list(s)
>>> print 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`.

因为`list`是一个内建函数的名字，你应该避免将其用作变量名。 我也避免用`l`，因为它看上去太像`1`。 所有这是为什么我用`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:

`list`函数把一个字符串分成独立的字母。 如果你想将字符串分成单词，你可以使用`split`方法：

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print 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:

一个被称作**分隔符**（**delimiter**）的可选实参指明那个字符被用于单词的边界。 下面的例子使用连字符号作为分隔符：

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['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:

`join`和`split`相反。它接受一个字符串的列表并将元素串联起来。 `join`是一个字符串方法，所以你必须在分隔符上调用它并传递列表作为一个形参：

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

此例中，分隔符是空格，所以`join`在两个单词中间放入一个空格。 为了不用空格串联字符串，你可以使用空字符串，`''`，作为分隔符：

10.10 Objects and values 对象和值

If we execute these assignment statements:

如果我们执行这些赋值语句：

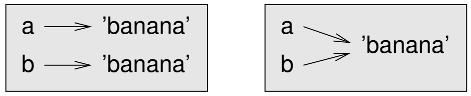

Figure 10.2: State diagram.

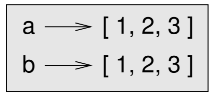

Figure 10.3: State diagram.

```
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.

我们知道a和b都引用一个字符串，但是我们不知道 它们是否引用相同的字符串。 有两个可能的状态，如图10.2所示。

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.

一种情况下，a和b引用两个不同的对象，它们有相同的值。 另一种情况是它们引用相同的对象。

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

为了查看两个变量是否引用同一个对象，你可以使用is运算符。

```
>>> 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.

此例中，Python生成一个字符串对象，a和b都引用它。

But when you create two lists, you get two objects:

但是，当你生成两个列表时，你获得两个对象：

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

So the state diagram looks like Figure 10.3.

所以状态图看起来类似图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.

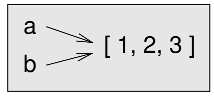

Figure 10.4: State diagram.

此例中，我们应该说两个列表**相等**（**equivalent**），因为它们具有相同的元素，但不是**相同**（**identical**），因为它们不是相同的对象。如果两个对象相同，那么它们也是相等的，但是如果它们相等，它们不一定相同。

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 execute `[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.

迄今为止，我们一直可互换的使用“对象”和“值”，但是更精确地说，一个对象有一个值。如果你执行`[1,2,3]`，那么你获得一个列表对象，其值是一个整数序列。如果另一个列表有相同的元素，我们说它有相同的值，但是它不是相同的对象。

10.11 Aliasing 别名

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

如果`a`引用一个对象并且你赋值`b=a`，那么两个变量引用相同的对象：

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

The state diagram looks like Figure 10.4.

状态图类似图 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.

变量和对象之间的关联被称作**引用**（**reference**）。此例中，有两个指向相同对象的引用。

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

具有超过一个引用的对象有超过一个的名字，所以我们说对象被**别名化**（**aliased**）了。

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

如果别名化的对象是可变的，用一个别名引起的变化影响另一个别名：

```
>>> b[0] = 17
>>> print a
[17, 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.

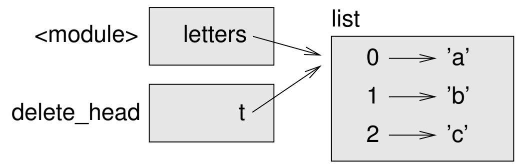

Figure 10.5: Stack diagram.

虽然此行为会是有用的，但是它易于出错。一般来讲，当你和可变对象一起工作时，避免别名会更安全。

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

对于想字符串这样的不可变对象，别名不是一个问题。在此例中：

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

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

`a`和`b`是否引用相同的字符串不会引起什么不同。

10.12 List arguments 列表实参

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

当你向一个函数传递列表时，函数获得指向该列表的引用。如果函数改变列表形参，那么调用者会看到改变。例如，`delete_head`从列表中删除第一个元素：

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

Here's how it is used:

这是它如何被使用的：

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

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

形参`t`和变量`letters`是同一个对象的别名。栈图如图10.5。

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

既然列表被两个框架共享，我把它画在他们之间。

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:

区分修改列表和生成新列表的操作非常重要。例如，`append`方法修改列表，但是+运算符生成一个新列表：

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

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

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:

当你写一些想要修改列表的函数的时候，此不同非常重要。 例如，此函数并没有删除列表的头：

```
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 none of that has any effect on the list that was passed as an argument.

片段运算符生成一个新的列表，赋值使得`t`引用它， 但是这些对被作为实参传递过来的列表毫无影响。

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:

替代的方法是写一个函数，生成并返回一个新列表。 例如，`tail`返回列表除第一个外的所有元素。

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

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

此函数不修改原来的列表。 这是如何使用它：

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

10.13 Debugging 调试

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:

不仔细的列表使用（以及其它可变对象）可能导致长时间的调试。 这是一些通常的陷阱以及避免他们的方法：

1. Don't forget that 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.

不要忘记大多数列表方法修改实参并返回`None`。这和字符串方法相反，其返回一个新的字符串并将原始的字符串放在一边。

If you are used to writing string code like this:
如果你想这样写过字符串代码：

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

It is tempting to write list code like this:
像这样写列表代码也很有诱惑力：

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

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.
因为`sort`返回`None`，所以对`t`的进一步运算很可能会失败。

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at `docs.python.org/lib/typesseq.html`. The methods and operators that only apply to mutable sequences are documented at `docs.python.org/lib/typesseq-mutable.html`.
在使用列表方法之前，你应该仔细的阅读文档，然后在交互模式下测试它们。列表和其它序列（如字符串）共享的方法和运算符的文档位于：`docs.python.org/lib/typesseq.html`。只能被用于可变序列的方法和运算符的文档在`docs.python.org/lib/typesseq-mutable.html`。

2. Pick an idiom and stick with it.
选择一个习惯并检查下去。

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.
列表的一部分问题是有太多的方法做事情。例如，为了从列表中删除一个元素，你可以使用`pop`、`remove`、`del`，甚至一个片段赋值。

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 right:
为了增加一个元素，你可以用`append`方法或者`+`运算符。假设`t`是一个列表，`x`是一个列表元素，这些都是正确的：

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

And these are wrong:
这些是错误的：

```
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.
在交互模式下测试每个例子，确保你理解他们做了什么。主意，只有最后一个引起运行时错误，其它三个是合法的，但是它们做了错误的事情。

3. Make copies to avoid aliasing.

   用复制，避免别名。

   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.

   如果你想使用类似`sort`的方法来修改实参， 但是也你需要保留原始列表，那么你可以复制它。

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

   此例中你也可以使用内建函数`sorted`， 其返回一个新的、排好序的列表并将原始列表放在一边。 但是在那个例子中，你应该避免将`sorted`作为变量名使用！

10.14 Glossary 术语表

list（列表）： A sequence of values.

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

index（索引）： An integer value that indicates an element in a list.

nested list（嵌套的列表）： A list that is an element of another list.

list traversal（列表遍历）： The sequential accessing of each element in a list.

mapping（映射）： A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.

accumulator（累加器）： A variable used in a loop to add up or accumulate a result.

augmented assignment（增量赋值）： A statement that updates the value of a variable using an operator like +=.

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

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

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

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

equivalent（相等的）： Having the same value.

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

reference（引用）： The association between a variable and its value.

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

delimiter（分隔符）： A character or string used to indicate where a string should be split.

10.15 Exercises

Exercise 10.6. *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. You can assume (as a precondition) that the elements of the list can be compared with the relational operators <, >, etc.*

For example, `is_sorted([1,2,2])` *should return* `True` *and* `is_sorted(['b','a'])` *should return* `False`*.*

Exercise 10.7. *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 10.8. *The (so-called) Birthday Paradox:*

1. *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.*

2. *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 read about this problem at `http://en.wikipedia.org/wiki/Birthday_paradox`*, and you can download my solution from* `http://thinkpython.com/code/birthday.py`*.*

Exercise 10.9. *Write a function called* `remove_duplicates` *that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order.*

Exercise 10.10. *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?*

Hint: use the `time` *module to measure elapsed time. Solution:* `http://thinkpython.com/code/wordlist.py`*.*

Exercise 10.11. *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, then 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 `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://thinkpython.com/code/inlist.py`*.*

Exercise 10.12. *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://thinkpython.com/code/reverse_pair.py`*.*

Exercise 10.13. *Two words "interlock" if taking alternating letters from each forms a new word. For example, "shoe" and "cold" interlock to form "schooled." Solution:* `http://thinkpython.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?*

Chapter 11

Dictionaries 字典

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.

字典（**dictionary**）和列表类似，但是更一般。 在列表中，索引必须是整数，在字典中，它们可以是（几乎）任意类型。

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

你可以将字典看成是索引集合（其被称作**键**（**keys**））和值集合直接的映射。 每个键映射到一个值。键和值的关联被称作**键-值对**（**key-value pair**） 或者 （**item**） 。

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

作为一个例子，我们将创建一个字典，从英语单词映射到西班牙语单词， 因此键和值都是字符串。

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.

`dict`函数生成一个新的没有任何项的字典。 因为`dict`是内建函数名，你应该避免将其用于变量名。

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

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

花括号`{}`表示一个空字典。为了向字典内增加项， 你可以使用方括号：

```
>>> 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:

该行生成一个项，从键`’one’`映射到值`'uno'`。 如果我们再次打印该字典，我们看到一个键-值对， 键和值之间有一个冒号。

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

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

此输出格式也是一个输入格式。 例如，你可以用三个项生成一个新的字典：

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

But if you print `eng2sp`, you might be surprised:

但是，如果你打印`eng2sp`，你可能很奇怪：

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

The order of the key-value pairs is not the same. In fact, 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.

键-值对的顺序和原来不同。事实上，如果你在你的计算机上键入相同的例子， 你可能获得不同的结果。通常，字典中项的顺序是不可预知的。

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:

但这不是个问题，因为字典的元素从不会被用整数索引来索引。 相反，你使用键来查找相应的值：

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

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

键`'two'`总是映射到值`'dos'`，因此项的顺序没有关系。

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

如果键不在字典中，那么你得到一个异常：

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

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

`len`函数对字典起作用，它返回键-值对的数量：

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

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

`in`运算符对字典起作用，他告诉你是否一些东西在字典中作为键出现 （作为值出现不够好）。

```
>>> '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 the values as a list, and then use the `in` operator:

为了看是否一些东西作为值出现，你可以使用`values`方法， 其作为一个列表返回值，然后使用`in`运算符：

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

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm, 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 there are in a dictionary. I won't explain how that's possible, but you can read more about it at `http://en.wikipedia.org/wiki/Hash_table`.

对于列表和字典，`in`运算符使用不同的算法。 对于列表， 它使用如8.6节中的搜索算法。 随着列表的增长，搜索时间成正比增长。 对于字典，Python使用一种叫做**哈希表**（**hashtable**）的算法， 其具有非凡的性质：无论字典中有多少项，`in`运算符几乎花费相同的时间。 我不会介绍这怎么可能，但是你可以在`http://en.wikipedia.org/wiki/Hash_table`读到更多关于它的内容。[1]

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 10.11, you can compare the speed of this implementation with the list `in` *operator and the bisection search.*

11.1 Dictionary as a set of counters 字典作为计数器集合

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:

假设给你一个字符串，你想计算每个字母出现的次数。 有多种方法可以可以使用：

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.

   你可以生成26个变量，每个对应一个字母表中的字母。 然后你可以遍历字符串，对于每个字符，递增相应的计数器， 可能使用链式条件。

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.

   你可以生成具有26个元素的列表。 然后你可以将每个字符传化为一个数字（使用内建函数`ord`）， 使用这些数字作为列表的索引，并递增适当的计数器。

3. 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.

[1]译者注：附录B.4介绍了关于哈希表的更多内容。

> 你可以生成一个字典，将字符作为键，计数器作为相应的值。 第一次看到一个字母，你应该向字典中增加一项。 这之后，你应该递增一个已有项的值。

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

每个选择执行相同的计算，但是它们用不同的方法。

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.

一个**实现（implementation）**是执行一个计算的方法； 一些实现比另一些好。 例如，字典实现的好处是我们不必事先知道字符串中出现了哪些字母， 我们只需要当这些字母出现时在给它们分配空间。

Here is what the code might look like:

代码可能是这样：

```
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 set of counters (or frequencies).

函数名是**直方图（histogram）**， 其是一个统计术语，用于计数器的集合（或者频率）。

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]`.

函数的第一行生成一个字典。`for`循环遍历该字符串。 每次循环，如果字符`c`不在字典中，我们用键`c`和初始值1生成一个新项 （既然我们已经见过一次该字母了）。 如果`c`已经在字典中了，那么我们递增`d[c]`。

Here's how it works:

这是它如何工作的：

```
>>> h = histogram('brontosaurus')
>>> print 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.

此直方图表明，字母`'a'`和`'b'`出现一次； `'o'`出现两次等等。

Exercise 11.2. *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')
>>> print h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Use `get` *to write* `histogram` *more concisely. You should be able to eliminate the* `if` *statement.*

11.2 Looping and dictionaries 循环和字典

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:

如果你在`for`中使用字典，那么它遍历该字典的键。 例如`print_hist`打印每个键以及相应的值：

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

Here's what the output looks like:

输出类似：

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

Again, the keys are in no particular order.

再一次，键没有什么顺序：

Exercise 11.3. *Dictionaries have a method called* `keys` *that returns the keys of the dictionary, in no particular order, as a list.*

Modify `print_hist` *to print the keys and their values in alphabetical order.*

11.3 Reverse lookup 逆向查找

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**.

给出一个字典`d`以及一个键`t`，很容易找到相应的值`v = d[k]`。 该运算被称作**查找**（**lookup**）

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.

但是，如果你有v并且想找到k该怎么办呢？ 你有两个问题：第一，可能有不止一个的键其映射到值v。 依赖于应用，你可能选择一个或者不得不生成一个包含所有键的列表。 第二，没有简单的语法完成**逆向查找**（**reverse lookup**）； 你必须搜索。

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

这是一个函数，接受一个值并返回其映射到该值的第一个键：

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

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 ValueError, which generally indicates that there is something wrong with the value of a parameter.

该函数是搜索模式的另一个例子，但是它使用了一个我们之前没有见过的特征，raise。raise语句引起一个异常；此例中它引起一个ValueError， 通常其表示形参的值有一些错误。

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.

我们我们到达循环结尾，这意味着 v在字典中不作为值存在， 所以我们触发一个异常。

Here is an example of a successful reverse lookup:

这是一个成功逆向查找的例子：

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

And an unsuccessful one:

和一个不成功的例子：

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

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

当你触发一个异常时，结果和Python触发一个一样： 它打印一个回溯和一个错误信息。

The raise statement takes a detailed error message as an optional argument. For example:

raise语句接受一个详细的错误信息作为可选的实参。 例如：

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: 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.

逆向查找比正向查找慢得多；如果你必须经常做它或者字典很大， 你的程序的性能将变得很差。

Exercise 11.4. *Modify* `reverse_lookup` *so that it builds and returns a list of* all *keys that map to* `v`*, or an empty list if there are none.*

11.4 Dictionaries and lists 字典和列表

Lists can appear as values in a dictionary. For example, if you were 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 with the same frequency, each value in the inverted dictionary should be a list of letters.

在字典中，列表可以作为值出现。例如，如果你给你一个从字母映射到频率的字典， 你可能想倒转它；也就是生成一个从频率映射到字母的字典。 既然可能有些字母具有相同的频率，那么在倒转字典中的每个值应该是一个字母的列表。

Here is a function that inverts a dictionary:

这是一个倒转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

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.

每次循环，`key`从`d`获得一个键和相应的值`val`。 如果`val`不在`inverse`中，这意味着我们之前没有见过它， 因此我们生成一个新项并用一个**单元素集合（singleton）**（包括唯一元素的列表） 初始化它。

Here is an example:

这是一个例子：

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
```

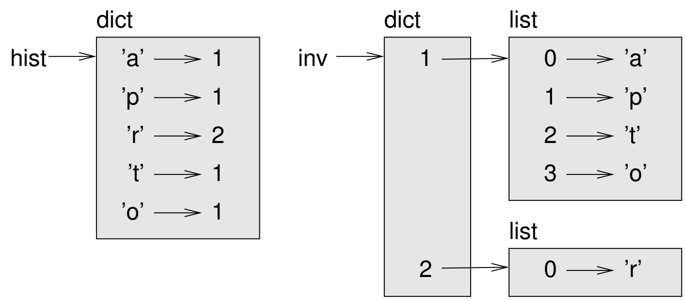

Figure 11.1: State diagram.

```
>>> inverse = invert_dict(hist)
>>> print 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 usually draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

图11.1是一个状态图，显示`hist`和`inverse`。一个字典被表示为一个盒子，类型类型`dict`在上面，键-值对在里面。如果值是整数、浮点数或者字符串，那么我通常把他们画在盒子里，但是我通常将列表画在盒子外，仅仅是为了使图简洁。

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

字典中，列表可以是值，如此例所示，但是它们不能是键。如果你试图这样做，这是会发生什么：

```
>>> 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**.

我之前提过，字典使用哈希表实现，这意味着键必须是**可哈希的（hashable）**。

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.

哈希（hash）是一个函数，接受一个值（任何类型）并返回一个整数。字典使用这些整数，被称作哈希值，来存储和查找键-值对。

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.

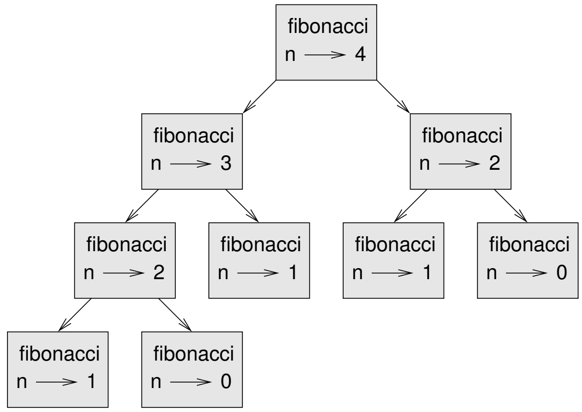

Figure 11.2: Call graph.

如果键是不可变的，此系统工作的很好。 但是如果键是可变的，如列表，那么坏事情发生了。 例如，当你生成一个键-值对的时候，Python哈希键并将其存储在相应的位置。 如果你改变键然后再次哈希它，它将到另一个位置。 在那种情况下，对于相同的键，你可能有两个入口， 或者你可能不能找到一个键。 无论如何，字典都不会正确的工作。

That's why the 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.

这是为什么键必须是可哈希的，以及为什么可变类型，如列表就不是。 绕过这种限制最简单的方法是使用元组， 我们将在下一章中介绍。

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

既然字典是可变的，它们不能被用作键， 但是它们可以被用作值。

Exercise 11.5. *Read the documentation of the dictionary method* `setdefault` *and use it to write a more concise version of* `invert_dict`*. Solution:* *http: // thinkpython. com/ code/ invert_ dict. py* .

11.5 Memos 备忘录

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 very quickly.

如果你调用过6.7节的`fibonacci`函数， 你可能注意到你提供越大的实参，函数运行的时间越长。 更进一步地，运行时间增长的非常快。

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

为了理解原因，考虑图11.2，其展示了`n=4`时， `fibonacci` 的**调用图**（**call graph**）。

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.

调用图显示了函数框架的集合，每个框架到其调用函数的框架用线相连。在图的顶端，n=4的fibonacci调用n=3和n=2的fibonacci。接着，n=3的fibonacci调用n=2和n=1的fibonacci，等等。

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.

计算fibonacci(0)和fibonacci(1)被调用多少次。对该问题，这不是一个有效的解，并且随着实参的变大会变得更糟。

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 an implementation of fibonacci using memos:

一个解决办法是保存已经计算过的值，将它们存在一个字典中。之前计算过的值，其被存储以便今后使用，被称作**备忘录**（**memo**）。这是使用备忘录的fibonacci的实现：

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

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

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

known 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.

known是一个字典，其记录我们已经知道的斐波那契数。它从两项开始：0映射到0，1映射到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.

当fibonacci被调用时，它检查known。如果结果已经在那儿了，则立即返回。否则，它不得不计算新的值，将其加入字典，并返回它。

Exercise 11.6. *Run this version of* fibonacci *and the original with a range of parameters and compare their run times.*

Exercise 11.7. *Memoize the Ackermann function from Exercise 6.5 and see if memoization makes it possible to evaluate the function with bigger arguments. Hint: no. Solution:* `http://thinkpython.com/code/ackermann_memo.py`.

11.6 Global variables 全局变量

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.

在前面的例子中，known在函数的外面被生成，因此它属于特殊的被称作__main__的框架。在__main__中的变量有时被称作**全局的**（**global**），因为它们可以从任意函数访问。

和局部变量不同，当它们的函数结束时它们消失， 从一个函数调用到另一个调用，全局变量都存在。

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:

经常使用全局变量作为**标记**（**flags**）； 也就是指示（标记）是否一个条件为真的布尔变量。 例如，一些程序使用一个被称作`verbose`的标记来控制 输出中细节的等级：

```
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:

如果你试图重新对全局变量赋值，你可能很奇怪。 下面的例子被假定用于记录是否该函数已经被调用过了：

```
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.

但是如果你运行它，你将发现`been_called`的值没有改变。 问题是`example2`生成一个新的被称作`been_called`的局部变量。 当函数结束的时候，该局部变量也消失了，并且对全局变量没有影响。

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

为了在一个函数内部重新对全局变量赋值， 你必须在使用它之前，**声明**（**declare**）该全局变量：

```
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."

`global`语句告诉解释器诸如此类的事情， "在此函数中，当我说`been_called`时，我的意思是全局变量； 不要生成局部变量"。

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

这是一个试图更新全局变量的例子：

```
count = 0

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

If you run it you get:

如果你运行它，你得到：

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

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

Python假设`count`是局部的，这意味着你在写它之前，正在读它。 再次，解决方案是声明`count`是全局的。

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

If the global value is mutable, you can modify it without declaring it:

如果全局值是可变的，你可以修改它，而不用声明：

```
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:

因此你可以增加、删除和替代全局列表或者字典的元素， 但是如果你想对变量重新赋值，你必须声明它：

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

11.7 Long integers 长整数

If you compute `fibonacci(50)`, you get:

如果你计算`fibonacci(50)`，你会得到：

```
>>> fibonacci(50)
12586269025L
```

The `L` at the end indicates that the result is a long integer, or type `long`. In Python 3, `long` is gone; all integers, even really big ones, are type `int`.

结尾的L表明结果是一个长整数，或者`long`类型。 在Python3中， `long`不存在了；所有的整数， 即使非常大的，也是 `int`类型。

Values with type `int` have a limited range; long integers can be arbitrarily big, but as they get bigger they consume more space and time.

int类型的值有一个有限的取值范围； 长整数可以任意任意大，但是随着它们变大， 它们消耗更多的空间和时间。

The mathematical operators work on long integers, and the functions in the math module, too, so in general any code that works with int will also work with long.

数学运算可用于长整数并且math模块中的函数也可以， 因此一般来讲，任何int好使的代码通用能用于 long。

Any time the result of a computation is too big to be represented with an integer, Python converts the result as a long integer:

任何时候，一个太大以至于不能用一个整数表示的计算的结果， Python会转化该结果为一个长整数：

```
>>> 1000 * 1000
1000000
>>> 100000 * 100000
10000000000L
```

In the first case the result has type int; in the second case it is long.

第一个例子中，结果类型是int； 第二个例子中，它是long。

Exercise 11.8. *Exponentiation of large integers is the basis of common algorithms for public-key encryption. Read the Wikipedia page on the RSA algorithm (http: // en. wikipedia. org/ wiki/ RSA) and write functions to encode and decode messages.*

11.8 Debugging 调试

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

当你操作较大的数据集时，通过打印并手工检查数据来调试很不方便。 这是对于调试大数据集合的一些建议：

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.

[缩小输入：]如果可能，减小数据集合的大小。 例如，如果程序读取一个文本文件，只以前10行开始， 或者用一个你能找到的小的例子开始。 你或者可以编辑文件，或者（更好）修改程序让它只读取前n行。

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.

如果有错误，你可以减小n到出现错误的最小的值， 然后逐渐增加它，随着你发现错误并改正错误。

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.

[检查摘要和类型：]不用打印并检查全部数据集合，而是考虑打印数据的摘要： 例如字典中项的数目或者数字列表的总和。

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.

通常的运行时错误原因是一个值的类型不正确。 为了调试此类错误，打印值的类型通常就足够了。

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."

[写自检：]有时你可以写代码来自动检查错误。 例如，如果你正在计算数字列表的平均数， 你可以检查其结果不用大于列表中最大的元素或者不小于最小的。 这被称作"合理性检查"，因为它探测出"不合理的"结果。

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

另一类检查比较两个不同计算的结果来看一下它们是否一致。 这被称作"一致性检查"。

Pretty print 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.

[美化打印输出：]格式化调试输出能够更容易定位一个错误。 6.9节中我们看到一个例子。 `pprint`模块提供了一个`pprint`函数， 其以更可读的格式显示内建类型。

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

再一次，你花在建立脚手架上的时间能减少你花在调试上的时间。

11.9 Glossary 术语表

dictionary（字典）: A mapping from a set of keys to their corresponding values.

key-value pair（键-值对）: The representation of the mapping from a key to a value.

item（项）: Another name for a key-value pair.

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

value（值）: 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（实现）: A way of performing a computation.

hashtable（哈希表）: The algorithm used to implement Python dictionaries.

hash function（哈希函数）: A function used by a hashtable to compute the location for a key.

hashable（可哈希的）: 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（查找）: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup（逆向查找）**:** A dictionary operation that takes a value and finds one or more keys that map to it.

singleton（单元素集合）**:** A list (or other sequence) with a single element.

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

histogram（直方图）**:** A set of counters.

memo（备忘录）**:** A computed value stored to avoid unnecessary future computation.

global variable（全局变量）**:** A variable defined outside a function. Global variables can be accessed from any function.

flag（标记）**:** A boolean variable used to indicate whether a condition is true.

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

11.10 Exercises

Exercise 11.9. *If you did Exercise 10.8, 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://thinkpython.com/code/has_duplicates.py`*.*

Exercise 11.10. *Two words are "rotate pairs" if you can rotate one of them and get the other (see* `rotate_word` *in Exercise 8.12).*

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

Exercise 11.11. *Here's another Puzzler from* Car Talk *(*`http://www.cartalk.com/content/puzzler/transcripts/200717`*):*

> *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://thinkpython.com/code/c06d` *and you can also download* `http://thinkpython.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://thinkpython.com/code/homophone.py`*.*

Chapter 12

Tuples 元组

12.1 Tuples are immutable 元组是不可变的

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.

元组是一个值的序列。值可以是任意类型，它们以整数索引， 所以从这方面讲元组更像列表。 重要的不同是元组是不可变的。

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

语法上，元组是以逗号分割的值的列表：

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

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

虽然并非必须，元组通常用括号括起来：

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

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

为了用一个单独的元素生成一个元组， 你必须包括最后的逗号：

```
>>> t1 = 'a',
>>> type(t1)
<type 'tuple'>
```

A value in parentheses is not a tuple:

括号中的一个值不是元组：

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

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

生成元组的另一种方法是内建函数`tuple`。 没有实参的话，它生成一个空元组：

```
>>> t = tuple()
>>> print 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、列表或元组）， 返回值是一个以该序列元组组成的元组。

```
>>> t = tuple('lupins')
>>> print 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，所以你应该避免使用它作为变量名。

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

大多数列表运算同样适用于元组。括号操作索引一个元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

And the slice operator selects a range of elements.

片段运算符选择一个范围的元素。

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

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

但是，如果你试图修改一个元组的元素，你会获得一个错误：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

你不能改变元组的元素，但是你可以用一个元组替换另一个：

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

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`:

交换两个变量的值通常很有用。 用传统的赋值方法，你必须使用一个临时变量。 例如，为了交换`a`和`b`：

```
>>> temp = a
>>> a = b
>>> b = temp
```

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

这个解决方案很麻烦；**元组赋值**（**tuple assignment**）更优雅:

```
>>> 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.

左侧是一个变量的元组；右侧是表达式的元组。 每个值被赋给它的各自的变量。 右侧的所有表达式在赋值之前被计算。

The number of variables on the left and the number of values on the right have to be the same:

左侧变量的数目和右侧变量的数目必须相同:

```
>>> 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:

更一般地，右侧可以是任意类型的序列（字符串、列表或者元组）。 例如，为了将一个电子邮件地址分割成用户名和域名， 你可以写:

```
>>> 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`的返回值是一个具有两个元素的列表； 第一个元素被赋给`uname`， 第二个赋给`domain`。

```
>>> print uname
monty
>>> print 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.

严格来讲，一个函数只能返回一个值，但是如果值是一个元组， 则会起到返回多个值的效果。 例如，如果你想除两个整数并计算商和余数， 则先计算`x/y`然后在计算`x%y`是低效的。 最好同时计算它们俩。

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`接受两个实参并返回两个值的元组，商和余数。 你可以将返回值存为元组:

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

Or use tuple assignment to store the elements separately:

或者使用元组赋值分别存储元素:

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

Here is an example of a function that returns a tuple:

这是一个返回元组的函数的例子:

```
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`是内建函数，其发现一个序列的最大和最小的元素。 `min_max`计算这两个值，并返回一个两个值的元组。

12.4 Variable-length argument tuples 变长实参元组

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:

函数可以接受一个可变数量的实参。 一个以*开始的形参名**汇集**（**gathers**）实参到一个元组中。 例如，`printall`接受任意数量的实参并打印它们:

```
def printall(*args):
    print args
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

汇集的形参可以是任何你喜欢的名字，但是习惯上用`args`。 这是函数如何工作的:

```
>>> 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:

汇集的补是**分散**（**scatter**）。如果你有一序列值， 你想将其作为多个实参传递给一个函数，你可以使用*运算符。 例如，`divmod`只接受两个实参；不可以用元组:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

但是如果你分散该元组，是可以的：

```
>>> divmod(*t)
(2, 1)
```

Exercise 12.1. *Many of the built-in functions use variable-length argument tuples. For example,* `max` *and* `min` *can take any number of arguments:*

```
>>> max(1,2,3)
3
```

But `sum` *does not.*

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

Write a function called `sumall` *that takes any number of arguments and returns their sum.*

12.5 Lists and tuples 列表和元组

`zip` is a built-in function that takes two or more sequences and "zips" them into a list of tuples where each tuple contains one element from each sequence. In Python 3, `zip` returns an iterator of tuples, but for most purposes, an iterator behaves like a list.

`zip`是一个内建函数，其接受两个或者更多的序列并把它们"zip"到一个元组列表中， 其中每个元组包括来自每个序列的一个元素。 在Python 3中，`zip`返回一个元组的迭代器， 但是对于大多数目的，迭代器行为类似列表。

This example zips a string and a list:

此例zip一个字符串和一个列表：

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

结果是一个元组的列表，其中每个元组包括一个来自字符串的字符和 相应的来自列表的元素。

If the sequences are not the same length, the result has the length of the shorter one.

如果序列长度不同，结果是较短的序列的长度。

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

你可以在`for`循环中使用元组赋值来遍历元组的列表：

```
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:

每次循环，Python选择列表的下一个元组并将元素赋给`letter`和`number`。 循环的输出是：

```
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]`:

如果你组合`zip`、`for`和元组赋值，你获得一个有用的惯用语法来 同时遍历两个（或者更多）序列。 例如，`has_match`接受两个序列，`t1`和`t2`， 并返回`True`如果有一个索引`i`使得`t1[i] == t2[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`:

如果你需要遍历一个序列的元素以及它们的索引值， 你可以使用内建函数`enumerate`：

```
for index, element in enumerate('abc'):
    print index, element
```

The output of this loop is:

此循环的输出是：

```
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 list of tuples, where each tuple is a key-value pair.

字典有一个被称作`items`的方法，其返回一个元组的列表， 其中每个元组是一个键-值对。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order. In Python 3, `items` returns an iterator, but for many purposes, iterators behave like lists.

和你期望的字典一样，项是无序的。 在Python 3中，`items`返回一个迭代器， 但是对于许多目的而言，迭代器的行为类似列表。

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

另一方面，你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)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

`dict`和`zip`的组合产生一个简介的生成字典的方法：

```
>>> d = dict(zip('abc', range(3)))
>>> print 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.

字典方法`update`也接受一个元组的列表， 并作为键-值对把它们加入一个已有的字典。

Combining `items`, tuple assignment and `for`, you get the idiom for traversing the keys and values of a dictionary:

组合`items`、元组赋值和 `for`，你获得遍历字典中键和值的惯用语法：

```
for key, val in d.items():
    print val, key
```

The output of this loop is:

此循环的输出是：

```
0 a
2 c
1 b
```

Again.

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:

使用元组作为字典的键很常见（主要是因为你不能使用列表）。 例如，一个电话簿可能从姓-名对映射到一个电话号码。 假设我们已经定义`last`、`first`和`number`， 我们可以写：

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

括号中的表达式是一个元组。 你可以使用元组赋值来遍历该字典。

```
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，其是一个元组。 它将元组的元素赋给`last`和`first`， 然后打印名字和相应的电话号码。

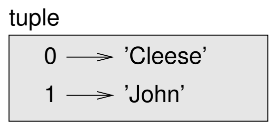

Figure 12.1: State diagram.

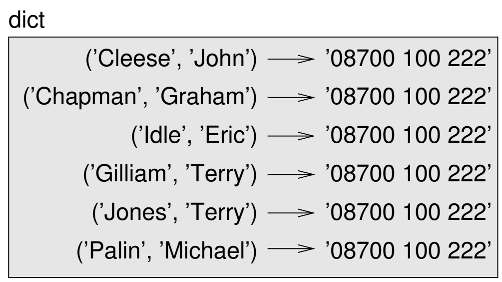

Figure 12.2: State diagram.

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.

有两种方法在状态图中表示元组。 更细致的版本和它们出现在列表中一样展示索引和元素。 例如，元组`('Cleese', 'John')`应该像图12.1一样出现。

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.

但是在一个更大的图中，你可能想省去细节。 例如，电话簿的图可能像图12.2一样。

Here the tuples are shown using Python syntax as a graphical shorthand.

这里，类似速记的方式，使用Python语法显示元组。

The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

此图中的电话号码是BBC的投诉热线，所以请不要拨打它。

12.7 Comparing tuples 元组比较

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).

关系运算符可用于元组和其它的序列；Python开始比较每个序列的第一个元素。 如果它们相等，它继续到下一个元素，以此类推，指导发现不同的元素。 后续的元素不被考虑了（即使它们相当的大）。

```
>>> (0, 1, 2) < (0, 3, 4)
True
```

```
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

`sort`函数以相同的方式工作。 它主要以第一个元素排序，但是在其相同的情况下， 以第二个元素排序，以此类推。

This feature lends itself to a pattern called **DSU** for

此特征导致被称作**DSU**的模式，其

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

[装饰]一个序列，通过用一个或者更多排序键跟着来自序列的元素来构建一个元组列表，

Sort the list of tuples, and

[排序]该元组列表，并且

Undecorate by extracting the sorted elements of the sequence.

[反装饰]通过抽取排好序的序列的元素。

For example, suppose you have a list of words and you want to sort them from longest to shortest:

例如，假设你有一个词列表并且你想按照从长到短排序它们：

```
def sort_by_length(words):
    t = []
    for word in words:
       t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

第一个循环构建一个元组列表，其中每个元组是一个单词的长度和该单词。

`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.

`sort`首先比较第一个元素，长度，并且只考虑到第二个元素。 实参`reverse=True`告诉`sort`按照降序排序。

The second loop traverses the list of tuples and builds a list of words in descending order of length.

第二个循环遍历元组列表并且构建一个以长度降序排列的单词列表。

Exercise 12.2. *In this example, ties are broken by comparing words, so words with the same length appear in reverse alphabetical order. For other applications you might want to break ties at random. Modify this example so that words with the same length appear in random order. Hint: see the* `random` *function in the* `random` *module. Solution:* `http://thinkpython.com/code/unstable_sort.py`.

12.8 Sequences of sequences 序列的序列

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.

到现在我一直专注于元组的列表，但是本章中几乎所有的例子也适用于列表的列表， 元组的元组以及列表的元组。为了避免列举所有可能的组合， 讨论序列的序列有时候比较容易。

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others?

在许多情况下，不同类型的序列（字符串、列表和元组）可以被互换使用。 所以如何以及为什么你选择一个而不是其它的呢?

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.

从明显的开始，字符串比其它的序列更受限，因为元素必须是字符。 它们也是不可变的。 如果你需要改变字符串中字符的能力（相对生成一个新的字符串）， 你可能想使用字符列表代替。

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

列表比元组更一般，主要是因为它们是可变的。 但是也有一些情况你可能更喜欢用元组:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.

   在一些情况下，如`return`语句，生成一个元组比生成一个列表语法更简单。 在其它情况下，你可能倾向于列表。

2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

   如果你想使用一个序列作为字典的键，那么你必须使用一个类似元组和字符串的不可变类型。

3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

   如果你正向函数传递一个序列作为实参，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 functions `sorted` and `reversed`, which take any sequence as a parameter and return a new list with the same elements in a different order.

因为元组是不可变的，所以它们不提供类似`sort`和`reverse`的方法， 其改变已有的列表。但是Python提供内建函数`sorted`和`reversed`， 其接受任意序列作为形参并返回一个新的相同元素，不同顺序的列表。

12.9 Debugging 调试

Lists, dictionaries and tuples are known generically as **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, and 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 composition. 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、字典和元组通常被看做**数据结构**（**data structures**）； 本章中我们正开始看到合成数据结构，如元组的列表， 元组作为键、列表作为值的字典。 合成数据结构很有用，但是它们倾向产生我称作的**形状错误**（**shape errors**）； 也就是，当一个数据结构有错误的类型、大小或组合时引起的错误。 例如，如果你期望一个具有一个整数的列表并且我给你一个整数（非列表）， 它将不会工作。

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://thinkpython.com/code/structshape.py`

为了帮助调试这些类型的错误，我写了一个被称作`structshape`的模块， 其提供一个函数，也称作`structshape`，其接受任意类型的数据结构作为实参 并返回一个字符串，其简要说明它的形状。 你可以从`http://thinkpython.com/code/structshape.py`下载它。

Here's the result for a simple list:

这是一个简单列表的结果：

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print 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:

一个更有想象力的程序可能写出"list of 3 ints"， 但是不处理复数更简单。这是一个列表的列表：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print 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:

如果列表的元素类型不同，`structshape`将它们按顺序按类型分组：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

Here's a list of tuples:

这是一个元组的列表:

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> print structshape(lt)
list of 3 tuple of (int, str)
```

And here's a dictionary with 3 items that map integers to strings.

这是一个具有3个项的字典，其从整数映射到字符串。

```
>>> d = dict(lt)
>>> print structshape(d)
dict of 3 int->str
```

If you are having trouble keeping track of your data structures, `structshape` can help.

如果你跟踪你的数据结构有困难，`structshape`能帮你。

12.10 Glossary 术语表

tuple（元组）: An immutable sequence of elements.

tuple assignment（元组赋值）: 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.

gather（汇集）: The operation of assembling a variable-length argument tuple.

scatter（分散）: The operation of treating a sequence as a list of arguments.

DSU: Abbreviation of "decorate-sort-undecorate," a pattern that involves building a list of tuples, sorting, and extracting part of the result.

data structure（数据结构）: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape (of a data structure)（数据结构的形状）: A summary of the type, size and composition of a data structure.

12.11 Exercises

Exercise 12.3. *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://thinkpython.com/code/most_frequenct.py`*.*

Exercise 12.4. *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 set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

2. *Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, 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 set of 8 letters forms the most possible bingos? Hint: there are seven.*

   Solution: `http://thinkpython.com/code/anagram_sets.py`.

Exercise 12.5. *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://thinkpython.com/code/metathesis.py`. Credit: This exercise is inspired by an example at `http://puzzlers.org`.*

Exercise 12.6. *Here's another Car Talk Puzzler (`http://www.cartalk.com/content/puzzler/transcripts/200651`):*

> *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.*

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://thinkpython.com/code/reducible.py`.

Chapter 13

Case study: data structure selection 案例学习：数据结构选择

13.1 Word frequency analysis 词频分析

As usual, you should at least attempt the following exercises before you read my solutions.

和往常一样，在你阅读我的答案之前，你至少应该尝试解决一下下面的习题。

Exercise 13.1. *Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.*

写一个程序，读取一个文件，每一行是一个单词， 删掉单词中的空格和标点，并且将它们转化为小写字母。

Hint: The `string` *module provides strings named* `whitespace`*, which contains space, tab, newline, etc., and* `punctuation` *which contains the punctuation characters. Let's see if we can make Python swear:*

提示：`string`模块提供了名为`whitespace`的字符串， 其包括空格、制表符、新行等等，以及`punctuation`， 其包括标点字符。让我们看一下是否我们能让*Python*说脏话：

```
>>> import string
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Also, you might consider using the string methods `strip`*,* `replace` *and* `translate`*.*

同时，你可能需要考虑使用字符串方法`strip`、`replace`和`translate`。

Exercise 13.2. *Go to Project Gutenberg (*`gutenberg.org`*) and download your favorite out-of-copyright book in plain text format.*

前往古腾堡项目（*gutenberg.org*）以纯文本格式下载你喜欢的版权已过期的图书。

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

修改你前面习题的程序来读取你下载的书， 跳过文件开始的头部信息，并如前面一样处理其余的单词。

Then modify the program to count the total number of words in the book, and the number of times each word is used.

然后修改程序来计算书中单词的总数， 以及每个单词被使用的次数。

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

打印该书使用的不同单词的数目。 比较不同年代，不同作者写的不同的书。 哪个作者使用的词汇最大?

Exercise 13.3. *Modify the program from the previous exercise to print the 20 most frequently-used words in the book.*

修改前面习题的程序来打印该书中最常使用的20个单词。

Exercise 13.4. *Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that* should *be in the word list, and how many of them are really obscure?*

修改前面的程序来读取一个词表（见*9.1*节）， 然后打印书中所有没有出现在该词表中的单词。 它们中有多少是拼写错误的? 有多少是词表中应该包括的常用词? 有多少是生僻词?

13.2 Random numbers 随机数

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

给定相同的输入，大多数计算机程序每次生成相同的输出， 所以它们被称作**确定性的**（**deterministic**）。 确定性通常是个好东西，因为我们期望相同的计算产生相同的结果。 然而，对于有些应用，我们期望计算机不可预知。 有戏是一个明显的例子，但是有更多的例子。

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

让一个程序真的非确定性并不容易，但是有办法使它看起来是非确定的。 其中之一是使用生成**伪随机**（**pseudorandom**）数算法。 伪随机数不是真正的随机，因为它们由一个确定性的计算生成， 但是仅看其生成的数字，不可能将它们和随机生成的相区分开。

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

`random`模块提供生成伪随机数的函数（从此之后，我将简单的称其为"随机"）。

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

函数`random`返回一个0.0到1.0之间的随机浮点数（包括0.0，但是不包括1.0）。 每次你调用`random`，你获得一个长序列中的下一个树。 为了看一个例子，运行此循环:

```
import random

for i in range(10):
    x = random.random()
    print x
```

The function `randint` takes parameters `low` and `high` and returns an integer between `low` and `high` (including both).

函数`randint`接受参数`low`和`high`， 并返回一个`low`和`high`之间的整数（两个都包括）。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

为了从一个序列中随机的选择一个元素，你可以使用`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.

`random`模块也提供从包括高斯、指数、伽马以及更多连续分布中生成随机值的函数。

Exercise 13.5. *Write a function named* `choose_from_hist` *that takes a histogram as defined in Section 11.1 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:*

写一个名为`choose_from_hist`的函数， 其接受一个如*11.1*节定义的直方图， 并从该直方图中返回一个随机值，其选择概率和频率成正比。 例如，对于此直方图：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> print hist
{'a': 2, 'b': 1}
```

your function should return `'a'` *with probability* 2/3 *and* `'b'` *with probability* 1/3*.*

你的函数应该返回`'a'`和概率2/3以及`'b'`和概率1/3。

13.3 Word histogram 单词直方图

You should attempt the previous exercises before you go on. You can download my solution from `http://thinkpython.com/code/analyze_book.py`. You will also need `http://thinkpython.com/code/emma.txt`.

在你继续之前，你应该尝试前面的习题。 你可以从`http://thinkpython.com/code/analyze_book.py`下载我的答案。 你也需要`http://thinkpython.com/code/emma.txt`。

Here is a program that reads a file and builds a histogram of the words in the file:

这是一个读取一个文件并建立文件中单词直方图的程序。

```
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.

改程序读取`emma.txt`，其包括Jane Austen写的*Emma*的文本。

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `hist` is being used as an accumulator.

`process_file`循环读取每行文件，每次把它们传递给`process_line`。直方图`hist`被用作一个累加器。

`process_line` uses the string method `replace` to replace hyphens with spaces before using `split` to break the line into a list of strings. It traverses the list of words and uses `strip` and `lower` to remove punctuation and convert to lower case. (It is a shorthand to say that strings are “converted;” remember that string are immutable, so methods like `strip` and `lower` return new strings.)

在使用`split`将一行文件分成一个字符串列表之前，`process_line`使用字符串`replace`方法来用空格替换连字符。它遍历单词的列表，并使用`strip`和`lower`来删除标点以及转化为小写。（简称字符串被“转化”；记住字符串是不可变的，所以类似`strip`和`lower`的方法返回新的字符串。）

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

最后，`process_line`通过生成一个新的项或者递增一个已有的项来更新直方图。

To count the total number of words in the file, we can add up the frequencies in the histogram:

为了计算文件中单词的总数，我们可以累加直方图中的频率：

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

不同单词的数量恰好是词典中项的数目：

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

这是打印结果的代码:

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

And the results:

结果是:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words 最常用单词

To find the most common words, we can apply the DSU pattern; `most_common` takes a histogram and returns a list of word-frequency tuples, sorted in reverse order by frequency:

为了找到最常用的单词，我们可以应用DSU模式； `most_common`接受一个直方图并且返回一个 以频率倒序排列的单词-频率元组列表:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

Here is a loop that prints the ten most common words:

这是一个打印10个最常用单词的循环:

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

And here are the results from *Emma*:

这是来自*Emma*的结果:

```
The most common words are:
to  5242
the  5205
and  4897
of  4295
i  3191
a  3130
it  2529
her  2483
was  2400
she  2364
```

13.5 Optional parameters 可选形参

We have seen built-in functions and methods that take a variable number of arguments. It is possible to write user-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

我们已经见过接受可变数目实参的函数和方法了。 写出具有可选实参的用户自定义函数也是有可能的。 例如，这是一个打印直方图中最常见单词的函数。

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[:num]:
        print word, '\t', freq
```

The first parameter is required; the second is optional. The **default value** of `num` is 10.

第一个形参是必须的；第二个是可选的。 `num`的**默认值**（**default value**）是10.

If you only provide one argument:

```
print_most_common(hist)
```

`num` gets the default value. If you provide two arguments:

`num`获得默认值。如果你提供两个实参：

```
print_most_common(hist, 20)
```

`num` gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

`num`获得实参的值。换句话说，可选实参**覆写**（**overrides**）了默认值。

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

如果一个函数必选和可选两类形参，则所有的必选形参必须首先出现， 后面跟着可选的。

13.6 Dictionary subtraction 字典差集

Finding the words from the book that are not in the word list from `words.txt` is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in another set (the words in the list).

从书中找到所有没出现在词表`words.txt`中的单词是一个差集问题。 也就是，我们想从一个集合中（书中的单词）找到所有没出现在另一个集合中 （列表中的单词）的单词。

`subtract` takes dictionaries `d1` and `d2` and returns a new dictionary that contains all the keys from `d1` that are not in `d2`. Since we don't really care about the values, we set them all to None.

`subtract`接受词典`d1`和`d2`，并返回一个新的词典， 其包括`d1`中的所有没出现在`d2`中的键。既然我们并不真正关心值，我们将它们都设为None。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in `words.txt`, we can use `process_file` to build a histogram for `words.txt`, and then subtract:

为了找到书中的没有出现在`words.txt`中的单词， 我们可以使用`process_file`来为`words.txt`构建一个直方图， 然后subtract：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

Here are some of the results from *Emma*:

这是来自*Emma*的一些结果：

```
The words in the book that aren't in the word list are:
 rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Some of these words are names and possessives. Others, like "rencontre," are no longer in common use. But a few are common words that should really be in the list!

这些单词一些是名字和所有歌。另外的，如"rencontre"不是常用单词。 但是有一些确实是真的应该包括在列表中的常用单词。

Exercise 13.6. *Python provides a data structure called* `set` *that provides many common set operations. Read the documentation at* `http://docs.python.org/lib/types-set.html` *and write a program that uses set subtraction to find words in the book that are not in the word list. Solution:* `http://thinkpython.com/code/analyze_book2.py`*.*

13.7 Random words 随机单词

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

为了从直方图中随机选择一个单词，最简单的算法是创建一个列表， 其中根据其出现的频率，每个单词都有多个拷贝， 然后从该列表中选择：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression [word] * freq creates a list with freq copies of the string word. The extend method is similar to append except that the argument is a sequence.

表达式[word] * freq生成一个具有freq个字符串word拷贝的列表。 extend方法和append类似，除了其实参为一个序列外。

Exercise 13.7. *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.*

该算法好使，但是不是很有效；每次你选择一个随机单词， 它都重建列表，其和原来的书一样大。 一个明显的改进是创建列表一次，然后进行多次选择， 但是该列表仍然很大。

An alternative is:

一个替代是：

1. *Use* keys *to get a list of the words in the book.*

   使用keys来获得该书中单词的列表。

2. *Build a list that contains the cumulative sum of the word frequencies (see Exercise 10.3). The last item in this list is the total number of words in the book, n.*

   创建一个包含单词频率累积和的列表（见习题*10.3*）。 此列表的最后一项是书中单词的数目n。

3. *Choose a random number from 1 to n. Use a bisection search (See Exercise 10.11) to find the index where the random number would be inserted in the cumulative sum.*

   选择一个从*1*到n的随机数。使用二分搜索（见习题*10.11*） 找到该随机数应该被在累积和中插入的索引。

4. *Use the index to find the corresponding word in the word list.*

   使用该索引从单词列表中找到相应的单词。

Write a program that uses this algorithm to choose a random word from the book. Solution: *http://thinkpython.com/code/analyze_book3.py.*

写一个使用该算法从书中选择一个随机单词的程序。 答案：*http://thinkpython.com/code/analyze_book3.py*。

13.8 Markov analysis 马尔科夫分析

If you choose words from the book at random, you can get a sense of the vocabulary, you probably won't get a sentence:

如果你从书中随机选择单词，那么你会感受到词表的概念， 你可能不会获得一个句子：

```
this the small regard harriet which knightley's it most things
```

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.

一系列随机单词没有意义，因为相邻的单词没有关系。 例如，在一个真实的句子中，你可能期望"the"后面跟着一个形容词或者名称， 不可能是一个动词或者副词。

One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the word that comes next. For example, the song *Eric, the Half a Bee* begins:

一种衡量这种关系的方法是马尔科夫分析，对于一个给定的单词序列， 其描述了接下来的单词的概率。 例如，歌曲*Eric, the Half a Bee*开始是：

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

In this text, the phrase "half the" is always followed by the word "bee," but the phrase "the bee" might be followed by either "has" or "is".

在此文本中，短语"half the"后面总是跟着单词"bee"， 但是短语"the bee"则可能跟着"has"或者"is"。

The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").

马尔科夫分析的结果是从每个前缀（如"half the"和"the bee"） 映射到所有可能的后缀（如"has"和"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.

给定此映射，你可以以任意前缀开始并从可能的后缀中随机选择一个来生成一个随机文本。 接下来，你可以组合前缀的结尾和新的后缀形参下一个前缀，并重复下去。

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."

例如，如果你以前缀"Half a"开始，然后下一个但是必须是"bee"， 因为此前缀在文本中仅出现一次。下一个前缀是"a bee"， 所以下一个后缀可能是"philosophically"，"be"或"due"。

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length. The length of the prefix is called the "order" of the analysis.

此例中，前缀的长度总是2，但是你可以以任意前缀长度进行马尔科夫分析。 前缀的长度被称作此分析的"阶"。

Exercise 13.8. *Markov analysis:*

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 analyze 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://thinkpython.com/code/markov.py`. You will also need `http://thinkpython.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.

Ok, the last one is the easy; the only mapping type we have seen is a dictionary, so it is the natural choice.

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 four 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 spend some time to build scaffolding.

ruminating: Take some time to think! What kind of error is it: syntax, runtime, 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?

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.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

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 paste the pieces back in a little bit 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 appear to be random, but are 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.

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://thinkpython.com/code/zipf.py`. To make the plots, you might have to install matplotlib (see `http://matplotlib.sourceforge.net/`).

Chapter 14

Files

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')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

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.

The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

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
>>> f.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 an integer (`d` stands for "decimal"):

```
>>> 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:

```
>>> camels = 42
>>> '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 (don't ask why), 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: illegal argument type for built-in operation
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at `docs.python.org/lib/typesseq-strings.html`.

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()
>>> print 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 `cwd` that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple filenames, so they are relative to the current directory. 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.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('music')
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.
Exercise 14.1. *The* `os` *module provides a function called* `walk` *that is similar to this one but more versatile. Read the documentation and use it to print the names of the files in a given directory and its subdirectories.*

Solution: *http: // thinkpython. 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')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

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` statement:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
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 executes 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.
Exercise 14.2. *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://thinkpython.com/code/sed.py`.

14.6 Databases

A **database** is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `anydbm` 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 anydbm
>>> db = anydbm.open('captions.db', '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. If you create a new item, `anydbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `anydbm` reads the file:

```
>>> print db['cleese.png']
Photo of John Cleese.
```

If you make another assignment to an existing key, `anydbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

Many dictionary methods, like `keys` and `items`, also work with database objects. So does iteration with a `for` statement.

```
for key in db:
    print key
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7 Pickling

A limitation of `anydbm` is that the keys and values have to be strings. 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)
'(lp0\nI1\naI2\naI3\na.'
```

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)
>>> print 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`.

Exercise 14.3. *If you download my solution to Exercise 12.4 from* `http://thinkpython.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://thinkpython.com/code/anagram_db.py`

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**. A pipe is an object that 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 just 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`:

[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.

```
>>> 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
```

Exercise 14.4. *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://thinkpython.com/code/find_duplicates.py`*.*

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:

```
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`:

```
>>> print wc
<module 'wc' from 'wc.py'>
```

That provides a function called `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 executes the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't execute 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 is executed. Otherwise, if the module is being imported, the test code is skipped.

Exercise 14.5. *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 might 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.

14.12 Exercises

Exercise 14.6. *The* `urllib` *module provides methods for manipulating URLs and downloading information from the web. The following example downloads and prints a secret message from* `thinkpython.com`*:*

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn:
    print line.strip()
```

Run this code and follow the instructions you see there. Solution: `http://thinkpython.com/code/zip_code.py`*.*

Chapter 15

Classes and objects

Code examples from this chapter are available from `http://thinkpython.com/code/Point1.py`; solutions to the exercises are available from `http://thinkpython.com/code/Point1_soln.py`.

15.1 User-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 (a little) more complicated than the other options, but it has advantages that will be apparent soon.

A user-defined type is also called a **class**. A class definition looks like this:

```
class Point(object):
    """Represents a point in 2-D space."""
```

This header indicates that the new class is a `Point`, which is a kind of `object`, which is a built-in type.

The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a class object.

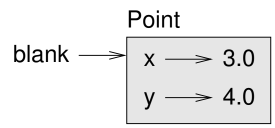

Figure 15.1: Object diagram.

```
>>> print 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()
>>> print blank
<__main__.Point instance 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).

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:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In this case, 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:

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print 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.
Exercise 15.1. *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:

```
class Rectangle(object):
    """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
```

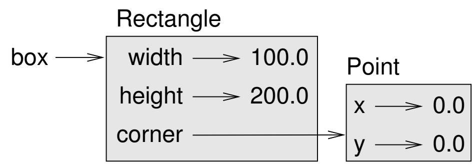

Figure 15.2: Object diagram.

```
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/2.0
    p.y = rect.corner.y + rect.height/2.0
    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)
(50.0, 100.0)
```

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.width + 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:

```
>>> print box.width
100.0
>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modifies `rect`, `box` changes.
Exercise 15.2. *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)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> 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. This behavior can be changed—we'll see how later.

If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

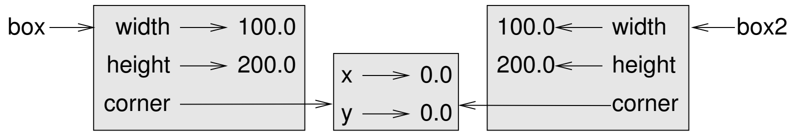

Figure 15.3: Object diagram.

```
>>> 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 contains 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.

Exercise 15.3. *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()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<type '__main__.Point'>
```

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.

15.8 Glossary

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

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 15.4. *Swampy (see Chapter 4) provides a module named* `World`*, which defines a user-defined type also called* `World`*. You can import it like this:*

```
from swampy.World import World
```

The following code creates a World object and calls the `mainloop` *method, which waits for the user.*

```
world = World()
world.mainloop()
```

A window should appear with a title bar and an empty square. We will use this window to draw Points, Rectangles and other shapes. Add the following lines before calling `mainloop` *and run the program again.*

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

You should see a green rectangle with a black outline. The first line creates a Canvas, which appears in the window as a white square. The Canvas object provides methods like `rectangle` *for drawing various shapes.*

`bbox` *is a list of lists that represents the "bounding box" of the rectangle. The first pair of coordinates is the lower-left corner of the rectangle; the second pair is the upper-right corner.*

You can draw a circle like this:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

The first parameter is the coordinate pair for the center of the circle; the second parameter is the radius.

If you add this line to the program, the result should resemble the national flag of Bangladesh (see `http://en.wikipedia.org/wiki/Gallery_of_sovereign-state_flags`*).*

1. *Write a function called* `draw_rectangle` *that takes a Canvas and a Rectangle as arguments and draws a representation of the Rectangle on the Canvas.*
2. *Add an attribute named* `color` *to your Rectangle objects and modify* `draw_rectangle` *so that it uses the color attribute as the fill color.*
3. *Write a function called* `draw_point` *that takes a Canvas and a Point as arguments and draws a representation of the Point on the Canvas.*
4. *Define a new class called Circle with appropriate attributes and instantiate a few Circle objects. Write a function called* `draw_circle` *that draws circles on the canvas.*
5. *Write a program that draws the national flag of the Czech Republic. Hint: you can draw a polygon like this:*

```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

I have written a small program that lists the available colors; you can download it from `http://thinkpython.com/code/color_list.py`*.*

Chapter 16

Classes and functions

Code examples from this chapter are available from `http://thinkpython.com/code/Time1.py`.

16.1 Time

As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time(object):
    """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.

Exercise 16.1. *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.*

Exercise 16.2. *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.*

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.

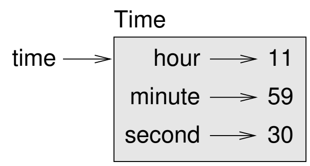

Figure 16.1: Object diagram.

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 the parameter `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.

Exercise 16.3. *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**.

Exercise 16.4. *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 **planned 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 the function that converts integers to Times (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.
Exercise 16.5. *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, then something has gone wrong.

Writing code to check your invariants can help you 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
```

Then 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.

planned 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 fruitless.

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.

16.7 Exercises

Code examples from this chapter are available from `http://thinkpython.com/code/Time1.py`; solutions to these exercises are available from `http://thinkpython.com/code/Time1_soln.py`.

Exercise 16.6. *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 16.7. *The* `datetime` *module provides* `date` *and* `time` *objects that are similar to the Date and Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at* `docs.python.org/lib/datetime-date.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.*

Chapter 17

Classes and methods

Code examples from this chapter are available from `http://thinkpython.com/code/Time2.py`.

17.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming.

It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Time` class defined in Chapter 16 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 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 the `Time` program, 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 user-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 simply 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 16, we defined a class named `Time` and in Exercise 16.1, you wrote a function named `print_time`:

```
class Time(object):
    """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(object):
    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(object):
    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, and makes it easier to maintain and reuse code.
Exercise 17.1. *Rewrite* `time_to_int` *(from Section 16.4) as a method. It is probably not appropriate to rewrite* `int_to_time` *as a method; what object you would 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, as in Exercise 17.1. 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 exactly 2 arguments (3 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.

17.4 A more complicated example

`is_after` (from Exercise 16.2) 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.

Exercise 17.2. *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.
Exercise 17.3. *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 user-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 quite a lot happening behind the scenes!

Changing the behavior of an operator so that it works with user-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see `docs.python.org/ref/specialnames.html`.
Exercise 17.4. *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 to do that. 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
```

Exercise 17.5. *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 will actually work for any kind of sequence. For example, in Section 11.1 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 can 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, then 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 are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually a good idea to initialize all of an objects 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 the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps attribute names (as strings) and values:

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'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 obj.__dict__:
        print attr, getattr(obj, attr)
```

`print_attributes` traverses the items in the object's 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 and `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 some 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.

Keeping the interface separate from the implementation means that you have to hide the attributes. Code in other parts of the program (outside the class definition) should use methods to read and modify the state of the object. They should not access the attributes directly. This principle is called **information hiding**; see `http://en.wikipedia.org/wiki/Information_hiding`.

Exercise 17.6. *Download the code from this chapter (`http://thinkpython.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://thinkpython.com/code/Time2_soln.py`*

17.12 Glossary

object-oriented language: A language that provides features, such as user-defined classes and method syntax, 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.

operator overloading: Changing the behavior of an operator like + so it works with a user-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 17.7. *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://thinkpython.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://thinkpython.com/code/GoodKangaroo.py`, which explains the problem and demonstrates a solution.

Exercise 17.8. *Visual is a Python module that provides 3-D graphics. It is not always included in a Python installation, so you might have to install it from your software repository or, if it's not there, from* `vpython.org`.

The following example creates a 3-D space that is 256 units wide, long and high, and sets the "center" to be the point $(128, 128, 128)$*. Then it draws a blue sphere.*

```
from visual import *

scene.range = (256, 256, 256)
scene.center = (128, 128, 128)

color = (0.1, 0.1, 0.9)          # mostly blue
sphere(pos=scene.center, radius=128, color=color)
```

`color` *is an RGB tuple; that is, the elements are Red-Green-Blue levels between 0.0 and 1.0 (see* `http://en.wikipedia.org/wiki/RGB_color_model`*).*

If you run this code, you should see a window with a black background and a blue sphere. If you drag the middle button up and down, you can zoom in and out. You can also rotate the scene by dragging the right button, but with only one sphere in the world, it is hard to tell the difference.

The following loop creates a cube of spheres:

```
t = range(0, 256, 51)
for x in t:
    for y in t:
        for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. *Put this code in a script and make sure it works for you.*
2. *Modify the program so that each sphere in the cube has the color that corresponds to its position in RGB space. Notice that the coordinates are in the range 0–255, but the RGB tuples are in the range 0.0–1.0.*
3. *Download* `http://thinkpython.com/code/color_list.py` *and use the function* `read_colors` *to generate a list of the available colors on your system, their names and RGB values. For each named color draw a sphere in the position that corresponds to its RGB values.*

You can see my solution at `http://thinkpython.com/code/color_space.py`.

Chapter 18

Inheritance

In this chapter I present classes to 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://thinkpython.com/code/Card.py`.

If you are not familiar with Anglo-American playing cards, you can read about them at `http://en.wikipedia.org/wiki/Playing_cards`.

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(object):
    """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.

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`.

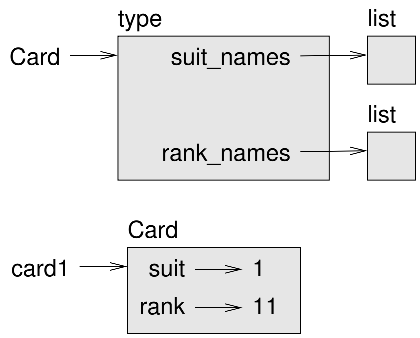

Figure 18.1: Object diagram.

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, so it has type `type`. `card1` has type `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 user-defined types, we can override the behavior of the built-in operators by providing a method named `__cmp__`.

`__cmp__` takes two parameters, `self` and `other`, and returns a positive number if the first object is greater, a negative number if the second object is greater, and 0 if they are equal to each 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 `__cmp__`:

```
# inside class Card:

    def __cmp__(self, other):
        # check the suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # suits are the same... check ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # ranks are the same... it's a tie
        return 0
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

The built-in function `cmp` has the same interface as the method `__cmp__`: it takes two values and returns a positive number if the first is larger, a negative number if the second is larger, and 0 if they are equal.

Exercise 18.1. *Write a `__cmp__` method for Time objects. Hint: you can use tuple comparison, but you also might consider using integer subtraction.*

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(object):

    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 `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. In real life "bottom dealing" is frowned upon, but in this context it's ok.

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 function without doing much real work is sometimes called a **veneer**. The metaphor comes from woodworking, where it is common to glue a thin layer of good quality wood to the surface of a cheaper piece of wood.

In this case we are defining a "thin" method that expresses a list operation in terms that are appropriate for decks.

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`.

Exercise 18.2. *Write a Deck method named* `sort` *that uses the list method* `sort` *to sort the cards in a* `Deck`*.* `sort` *uses the* `__cmp__` *method we defined to determine sort order.*

18.7 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.

It is called "inheritance" because the new class inherits the methods of the existing class. Extending this metaphor, the existing class is called the **parent** and the new class is called the **child**.

As an example, let's say we want a class to represent a "hand," that is, the set of cards held by one player. A hand is similar to a deck: both are made up of a set 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.

The definition of a child class is like other class definitions, but the name of the parent class appears 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.

`Hand` also 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
```

So when you create a Hand, Python invokes this init method:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

But 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.

Exercise 18.3. *Write a Deck method called* deal_hands *that takes two parameters, the number of hands and the number of cards per hand, and that creates new Hand objects, deals the appropriate number of cards per hand, and returns a list of Hand objects.*

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 program 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 scattered among several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8 Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in, "a Rectangle has a Point."

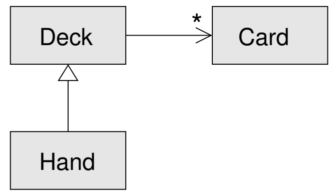

Figure 18.2: Class diagram.

- 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 changes in one class would require changes in the other.

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.

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.
Exercise 18.4. *Read* `TurtleWorld.py`*,* `World.py` *and* `Gui.py` *and draw a class diagram that shows the relationships among the classes defined there.*

18.9 Debugging

Inheritance can make debugging a challenge because when you invoke a method on an object, you might not know 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.

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()
>>> print 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."

Here's a program design suggestion: whenever 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 obey this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand.

If you violate this rule, your code will collapse like (sorry) a house of cards.

18.10 Data encapsulation

Chapter 16 demonstrates a development plan we might call "object-oriented design." We identified objects we needed—`Time`, `Point` and `Rectangle`—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://thinkpython.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(object):

    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 function—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.

Exercise 18.5. *Download my code from Section 13.8 (*`http://thinkpython.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://thinkpython.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: The relationship between a child class and its parent class.

HAS-A relationship: The relationship between two classes where instances of one class contain references to instances of the other.

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.

18.12 Exercises

Exercise 18.6. *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://thinkpython.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://thinkpython.com/code/PokerHandSoln.py`.

Exercise 18.7. *This exercise uses TurtleWorld from Chapter 4. You will write code that makes Turtles play tag. If you are not familiar with the rules of tag, see* `http://en.wikipedia.org/wiki/Tag_(game)`.

1. *Download* `http://thinkpython.com/code/Wobbler.py` *and run it. You should see a TurtleWorld with three Turtles. If you press the* Run *button, the Turtles wander at random.*

2. *Read the code and make sure you understand how it works. The* `Wobbler` *class inherits from* `Turtle`*, which means that the* `Turtle` *methods* `lt`, `rt`, `fd` *and* `bk` *work on Wobblers.*

   The `step` *method gets invoked by TurtleWorld. It invokes* `steer`*, which turns the Turtle in the desired direction,* `wobble`*, which makes a random turn in proportion to the Turtle's clumsiness, and* `move`*, which moves forward a few pixels, depending on the Turtle's speed.*

3. *Create a file named* `Tagger.py`*. Import everything from* `Wobbler`*, then define a class named* `Tagger` *that inherits from* `Wobbler`*. Call* `make_world` *passing the* `Tagger` *class object as an argument.*

4. *Add a* `steer` *method to* `Tagger` *to override the one in* `Wobbler`*. As a starting place, write a version that always points the Turtle toward the origin. Hint: use the math function* `atan2` *and the Turtle attributes* `x`, `y` *and* `heading`.

5. *Modify* `steer` *so that the Turtles stay in bounds. For debugging, you might want to use the* Step *button, which invokes* `step` *once on each Turtle.*

6. *Modify* `steer` *so that each Turtle points toward its nearest neighbor. Hint: Turtles have an attribute,* `world`*, that is a reference to the TurtleWorld they live in, and the TurtleWorld has an attribute,* `animals`*, that is a list of all Turtles in the world.*

7. *Modify* `steer` *so the Turtles play tag. You can add methods to* `Tagger` *and you can override* `steer` *and* `__init__`*, but you may not modify or override* `step`, `wobble` *or* `move`*. Also,* `steer` *is allowed to change the heading of the Turtle but not the position.*

   Adjust the rules and your `steer` *method for good quality play; for example, it should be possible for the slow Turtle to tag the faster Turtles eventually.*

Solution: `http://thinkpython.com/code/Tagger.py`.

Chapter 19

Case study: Tkinter

19.1 GUI

Most of the programs we have seen so far are text-based, but many programs use **graphical user interfaces**, also known as **GUIs**.

Python provides several choices for writing GUI-based programs, including wxPython, Tkinter, and Qt. Each has pros and cons, which is why Python has not converged on a standard.

The one I will present in this chapter is Tkinter because I think it is the easiest to get started with. Most of the concepts in this chapter apply to the other GUI modules, too.

There are several books and web pages about Tkinter. One of the best online resources is *An Introduction to Tkinter* by Fredrik Lundh.

I have written a module called `Gui.py` that comes with Swampy. It provides a simplified interface to the functions and classes in Tkinter. The examples in this chapter are based on this module.

Here is a simple example that creates and displays a Gui:

To create a GUI, you have to import `Gui` and instantiate a Gui object:

```
from Gui import *

g = Gui()
g.title('Gui')
g.mainloop()
```

When you run this code, a window should appear with an empty gray square and the title Gui. `mainloop` runs the **event loop**, which waits for the user to do something and responds accordingly. It is an infinite loop; it runs until the user closes the window, or presses Control-C, or does something that causes the program to quit.

This Gui doesn't do much because it doesn't have any **widgets**. Widgets are the elements that make up a GUI; they include:

Button: A widget, containing text or an image, that performs an action when pressed.

Canvas: A region that can display lines, rectangles, circles and other shapes.

Entry: A region where users can type text.

Scrollbar: A widget that controls the visible part of another widget.

Frame: A container, often invisible, that contains other widgets.

The empty gray square you see when you create a Gui is a Frame. When you create a new widget, it is added to this Frame.

19.2 Buttons and callbacks

The method `bu` creates a Button widget:

```
button = g.bu(text='Press me.')
```

The return value from `bu` is a Button object. The button that appears in the Frame is a graphical representation of this object; you can control the button by invoking methods on it.

`bu` takes up to 32 parameters that control the appearance and function of the button. These parameters are called **options**. Instead of providing values for all 32 options, you can use keyword arguments, like `text='Press me.'`, to specify only the options you need and use the default values for the rest.

When you add a widget to the Frame, it gets "shrink-wrapped;" that is, the Frame shrinks to the size of the Button. If you add more widgets, the Frame grows to accommodate them.

The method `la` creates a Label widget:

```
label = g.la(text='Press the button.')
```

By default, Tkinter stacks the widgets top-to-bottom and centers them. We'll see how to override that behavior soon.

If you press the button, you will see that it doesn't do much. That's because you haven't "wired it up;" that is, you haven't told it what to do!

The option that controls the behavior of a button is `command`. The value of `command` is a function that gets executed when the button is pressed. For example, here is a function that creates a new Label:

```
def make_label():
    g.la(text='Thank you.')
```

Now we can create a button with this function as its command:

```
button2 = g.bu(text='No, press me!', command=make_label)
```

When you press this button, it should execute `make_label` and a new label should appear.

The value of the `command` option is a function object, which is known as a **callback** because after you call `bu` to create the button, the flow of execution "calls back" when the user presses the button.

This kind of flow is characteristic of **event-driven programming**. User actions, like button presses and key strokes, are called **events**. In event-driven programming, the flow of execution is determined by user actions rather than by the programmer.

The challenge of event-driven programming is to construct a set of widgets and callbacks that work correctly (or at least generate appropriate error messages) for any sequence of user actions.

Exercise 19.1. *Write a program that creates a GUI with a single button. When the button is pressed it should create a second button. When* that *button is pressed, it should create a label that says, "Nice job!".*

What happens if you press the buttons more than once? Solution: `http://thinkpython.com/code/button_demo.py`

19.3 Canvas widgets

One of the most versatile widgets is the Canvas, which creates a region for drawing lines, circles and other shapes. If you did Exercise 15.4 you are already familiar with canvases.

The method `ca` creates a new Canvas:

```
canvas = g.ca(width=500, height=500)
```

`width` and `height` are the dimensions of the canvas in pixels.

After you create a widget, you can still change the values of the options with the `config` method. For example, the `bg` option changes the background color:

```
canvas.config(bg='white')
```

The value of `bg` is a string that names a color. The set of legal color names is different for different implementations of Python, but all implementations provide at least:

```
white   black
red     green    blue
cyan    yellow   magenta
```

Shapes on a Canvas are called **items**. For example, the Canvas method `circle` draws (you guessed it) a circle:

```
item = canvas.circle([0,0], 100, fill='red')
```

The first argument is a coordinate pair that specifies the center of the circle; the second is the radius.

`Gui.py` provides a standard Cartesian coordinate system with the origin at the center of the Canvas and the positive y axis pointing up. This is different from some other graphics systems where the origin is in the upper left corner, with the y axis pointing down.

The `fill` option specifies that the circle should be filled in with red.

The return value from `circle` is an Item object that provides methods for modifying the item on the canvas. For example, you can use `config` to change any of the circle's options:

```
item.config(fill='yellow', outline='orange', width=10)
```

`width` is the thickness of the outline in pixels; `outline` is the color.

Exercise 19.2. *Write a program that creates a Canvas and a Button. When the user presses the Button, it should draw a circle on the canvas.*

19.4 Coordinate sequences

The `rectangle` method takes a sequence of coordinates that specify opposite corners of the rectangle. This example draws a green rectangle with the lower left corner at the origin and the upper right corner at $(200, 100)$:

```
canvas.rectangle([[0, 0], [200, 100]],
                 fill='blue', outline='orange', width=10)
```

This way of specifying corners is called a **bounding box** because the two points bound the rectangle.

`oval` takes a bounding box and draws an oval within the specified rectangle:

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

`line` takes a sequence of coordinates and draws a line that connects the points. This example draws two legs of a triangle:

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

`polygon` takes the same arguments, but it draws the last leg of the polygon (if necessary) and fills it in:

```
canvas.polygon([[0, 100], [100, 200], [200, 100]],
               fill='red', outline='orange', width=10)
```

19.5 More widgets

Tkinter provides two widgets that let users type text: an Entry, which is a single line, and a Text widget, which has multiple lines.

`en` creates a new Entry:

```
entry = g.en(text='Default text.')
```

The `text` option allows you to put text into the entry when it is created. The `get` method returns the contents of the Entry (which may have been changed by the user):

```
>>> entry.get()
'Default text.'
```

`te` creates a Text widget:

```
text = g.te(width=100, height=5)
```

`width` and `height` are the dimensions of the widget in characters and lines.

`insert` puts text into the Text widget:

```
text.insert(END, 'A line of text.')
```

`END` is a special index that indicates the last character in the Text widget.

You can also specify a character using a dotted index, like `1.1`, which has the line number before the dot and the column number after. The following example adds the letters `'nother'` after the first character of the first line.

```
>>> text.insert(1.1, 'nother')
```

The `get` method reads the text in the widget; it takes a start and end index as arguments. The following example returns all the text in the widget, including the newline character:

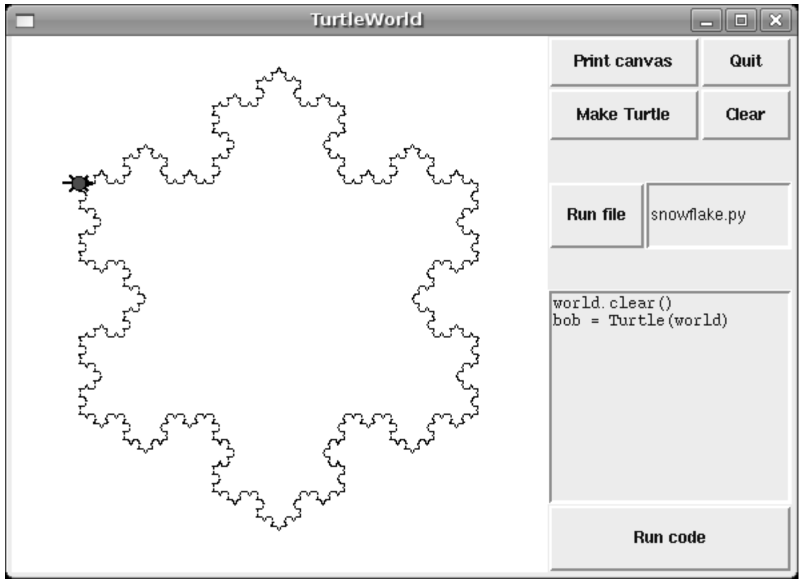

Figure 19.1: Class diagram.

```
>>> text.get(0.0, END)
'Another line of text.\n'
```

The `delete` method removes text from the widget; the following example deletes all but the first two characters:

```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
'An\n'
```

Exercise 19.3. *Modify your solution to Exercise 19.2 by adding an Entry widget and a second button. When the user presses the second button, it should read a color name from the Entry and use it to change the fill color of the circle. Use* `config` *to modify the existing circle; don't create a new one.*

Your program should handle the case where the user tries to change the color of a circle that hasn't been created, and the case where the color name is invalid.

You can see my solution at `http://thinkpython.com/code/circle_demo.py`.

19.6 Packing widgets

So far we have been stacking widgets in a single column, but in most GUIs the layout is more complicated. For example, Figure 19.1 shows a simplified version of TurtleWorld (see Chapter 4).

This section presents the code that creates this GUI, broken into a series of steps. You can download the complete example from `http://thinkpython.com/code/SimpleTurtleWorld.py`.

At the top level, this GUI contains two widgets—a Canvas and a Frame—arranged in a row. So the first step is to create the row.

```
class SimpleTurtleWorld(TurtleWorld):
    """This class is identical to TurtleWorld, but the code that
    lays out the GUI is simplified for explanatory purposes."""

    def setup(self):
        self.row()
        ...
```

`setup` is the function that creates and arranges the widgets. Arranging widgets in a GUI is called **packing**.

`row` creates a row Frame and makes it the "current Frame." Until this Frame is closed or another Frame is created, all subsequent widgets are packed in a row.

Here is the code that creates the Canvas and the column Frame that hold the other widgets:

```
        self.canvas = self.ca(width=400, height=400, bg='white')
        self.col()
```

The first widget in the column is a grid Frame, which contains four buttons arranged two-by-two:

```
        self.gr(cols=2)
        self.bu(text='Print canvas', command=self.canvas.dump)
        self.bu(text='Quit', command=self.quit)
        self.bu(text='Make Turtle', command=self.make_turtle)
        self.bu(text='Clear', command=self.clear)
        self.endgr()
```

`gr` creates the grid; the argument is the number of columns. Widgets in the grid are laid out left-to-right, top-to-bottom.

The first button uses `self.canvas.dump` as a callback; the second uses `self.quit`. These are **bound methods**, which means they are associated with a particular object. When they are invoked, they are invoked on the object.

The next widget in the column is a row Frame that contains a Button and an Entry:

```
        self.row([0,1], pady=30)
        self.bu(text='Run file', command=self.run_file)
        self.en_file = self.en(text='snowflake.py', width=5)
        self.endrow()
```

The first argument to `row` is a list of weights that determines how extra space is allocated between widgets. The list `[0,1]` means that all extra space is allocated to the second widget, which is the Entry. If you run this code and resize the window, you will see that the Entry grows and the Button doesn't.

The option `pady` "pads" this row in the y direction, adding 30 pixels of space above and below.

`endrow` ends this row of widgets, so subsequent widgets are packed in the column Frame. `Gui.py` keeps a stack of Frames:

- When you use `row`, `col` or `gr` to create a Frame, it goes on top of the stack and becomes the current Frame.

- When you use endrow, endcol or endgr to close a Frame, it gets popped off the stack and the previous Frame on the stack becomes the current Frame.

The method run_file reads the contents of the Entry, uses it as a filename, reads the contents and passes it to run_code. self.inter is an Interpreter object that knows how to take a string and execute it as Python code.

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

The last two widgets are a Text widget and a Button:

```
    self.te_code = self.te(width=25, height=10)
    self.te_code.insert(END, 'world.clear()\n')
    self.te_code.insert(END, 'bob = Turtle(world)\n')

    self.bu(text='Run code', command=self.run_text)
```

run_text is similar to run_file except that it takes the code from the Text widget instead of from a file:

```
def run_text(self):
    source = self.te_code.get(1.0, END)
    self.inter.run_code(source, '<user-provided code>')
```

Unfortunately, the details of widget layout are different in other languages, and in different Python modules. Tkinter alone provides three different mechanisms for arranging widgets. These mechanisms are called **geometry managers**. The one I demonstrated in this section is the "grid" geometry manager; the others are called "pack" and "place".

Fortunately, most of the concepts in this section apply to other GUI modules and other languages.

19.7 Menus and Callables

A Menubutton is a widget that looks like a button, but when pressed it pops up a menu. After the user selects an item, the menu disappears.

Here is code that creates a color selection Menubutton (you can download it from http://thinkpython.com/code/menubutton_demo.py):

```
g = Gui()
g.la('Select a color:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

mb creates the Menubutton. Initially, the text on the button is the name of the default color. The following loop creates one menu item for each color:

```
for color in colors:
    g.mi(mb, text=color, command=Callable(set_color, color))
```

The first argument of mi is the Menubutton these items are associated with.

The command option is a Callable object, which is something new. So far we have seen functions and bound methods used as callbacks, which works fine if you don't have to pass any arguments to the function. Otherwise you have to construct a Callable object that contains a function, like set_color, and its arguments, like color.

The Callable object stores a reference to the function and the arguments as attributes. Later, when the user clicks on a menu item, the callback calls the function and passes the stored arguments.

Here is what set_color might look like:

```
def set_color(color):
    mb.config(text=color)
    print color
```

When the user selects a menu item and set_color is called, it configures the Menubutton to display the newly-selected color. It also print the color; if you try this example, you can confirm that set_color is called when you select an item (and *not* called when you create the Callable object).

19.8 Binding

A **binding** is an association between a widget, an event and a callback: when an event (like a button press) happens on a widget, the callback is invoked.

Many widgets have default bindings. For example, when you press a button, the default binding changes the relief of the button to make it look depressed. When you release the button, the binding restores the appearance of the button and invokes the callback specified with the command option.

You can use the bind method to override these default bindings or to add new ones. For example, this code creates a binding for a canvas (you can download the code in this section from http://thinkpython.com/code/draggable_demo.py):

```
ca.bind('<ButtonPress-1>', make_circle)
```

The first argument is an event string; this event is triggered when the user presses the left mouse button. Other mouse events include ButtonMotion, ButtonRelease and Double-Button.

The second argument is an event handler. An event handler is a function or bound method, like a callback, but an important difference is that an event handler takes an Event object as a parameter. Here is an example:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

The Event object contains information about the type of event and details like the coordinates of the mouse pointer. In this example the information we need is the location of the mouse click. These values are in "pixel coordinates," which are defined by the underlying graphical system. The method canvas_coords translates them to "Canvas coordinates," which are compatible with Canvas methods like circle.

For Entry widgets, it is common to bind the <Return> event, which is triggered when the user presses the Return or Enter key. For example, the following code creates a Button and an Entry.

```
bu = g.bu('Make text item:', make_text)
en = g.en()
en.bind('<Return>', make_text)
```

make_text is called when the Button is pressed or when the user hits Return while typing in the Entry. To make this work, we need a function that can be called as a command (with no arguments) or as an event handler (with an Event as an argument):

```
def make_text(event=None):
    text = en.get()
    item = ca.text([0,0], text)
```

make_text gets the contents of the Entry and displays it as a Text item in the Canvas.

It is also possible to create bindings for Canvas items. The following is a class definition for Draggable, which is a child class of Item that provides bindings that implement drag-and-drop capability.

```
class Draggable(Item):

    def __init__(self, item):
        self.canvas = item.canvas
        self.tag = item.tag
        self.bind('<Button-3>', self.select)
        self.bind('<B3-Motion>', self.drag)
        self.bind('<Release-3>', self.drop)
```

The init method takes an Item as a parameter. It copies the attributes of the Item and then creates bindings for three events: a button press, button motion, and button release.

The event handler select stores the coordinates of the current event and the original color of the item, then changes the color to yellow:

```
    def select(self, event):
        self.dragx = event.x
        self.dragy = event.y

        self.fill = self.cget('fill')
        self.config(fill='yellow')
```

cget stands for “get configuration;” it takes the name of an option as a string and returns the current value of that option.

drag computes how far the object has moved relative to the starting place, updates the stored coordinates, and then moves the item.

```
    def drag(self, event):
        dx = event.x - self.dragx
        dy = event.y - self.dragy

        self.dragx = event.x
        self.dragy = event.y

        self.move(dx, dy)
```

This computation is done in pixel coordinates; there is no need to convert to Canvas coordinates.

Finally, `drop` restores the original color of the item:

```
def drop(self, event):
    self.config(fill=self.fill)
```

You can use the `Draggable` class to add drag-and-drop capability to an existing item. For example, here is a modified version of `make_circle` that uses `circle` to create an Item and `Draggable` to make it draggable:

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
    item = Draggable(item)
```

This example demonstrates one of the benefits of inheritance: you can modify the capabilities of a parent class without modifying its definition. This is particularly useful if you want to change behavior defined in a module you did not write.

19.9 Debugging

One of the challenges of GUI programming is keeping track of which things happen while the GUI is being built and which things happen later in response to user events.

For example, when you are setting up a callback, it is a common error to call the function rather than passing a reference to it:

```
def the_callback():
    print 'Called.'

g.bu(text='This is wrong!', command=the_callback())
```

If you run this code, you will see that it calls `the_callback` immediately, and *then* creates the button. When you press the button, it does nothing because the return value from `the_callback` is `None`. Usually you do not want to invoke a callback while you are setting up the GUI; it should only be invoked later in response to a user event.

Another challenge of GUI programming is that you don't have control of the flow of execution. Which parts of the program execute and their order are determined by user actions. That means that you have to design your program to work correctly for any possible sequence of events.

For example, the GUI in Exercise 19.3 has two widgets: one creates a Circle item and the other changes the color of the Circle. If the user creates the circle and then changes its color, there's no problem. But what if the user changes the color of a circle that doesn't exist yet? Or creates more than one circle?

As the number of widgets grows, it is increasingly difficult to imagine all possible sequences of events. One way to manage this complexity is to encapsulate the state of the system in an object and then consider:

- What are the possible states? In the Circle example, we might consider two states: before and after the user creates the first circle.

- In each state, what events can occur? In the example, the user can press either of the buttons, or quit.
- For each state-event pair, what is the desired outcome? Since there are two states and two buttons, there are four state-event pairs to consider.
- What can cause a transition from one state to another? In this case, there is a transition when the user creates the first circle.

You might also find it useful to define, and check, invariants that should hold regardless of the sequence of events.

This approach to GUI programming can help you write correct code without taking the time to test every possible sequence of user events!

19.10 Glossary

GUI: A graphical user interface.

widget: One of the elements that makes up a GUI, including buttons, menus, text entry fields, etc.

option: A value that controls the appearance or function of a widget.

keyword argument: An argument that indicates the parameter name as part of the function call.

callback: A function associated with a widget that is called when the user performs an action.

bound method: A method associated with a particular instance.

event-driven programming: A style of programming in which the flow of execution is determined by user actions.

event: A user action, like a mouse click or key press, that causes a GUI to respond.

event loop: An infinite loop that waits for user actions and responds.

item: A graphical element on a Canvas widget.

bounding box: A rectangle that encloses a set of items, usually specified by two opposing corners.

pack: To arrange and display the elements of a GUI.

geometry manager: A system for packing widgets.

binding: An association between a widget, an event, and an event handler. The event handler is called when the event occurs in the widget.

19.11 Exercises

Exercise 19.4. *For this exercise, you will write an image viewer. Here is a simple example:*

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

`PhotoImage` *reads a file and returns a* `PhotoImage` *object that Tkinter can display.* `Canvas.image` *puts the image on the canvas, centered on the given coordinates. You can also put images on labels, buttons, and some other widgets:*

```
g.la(image=photo)
g.bu(image=photo)
```

PhotoImage can only handle a few image formats, like GIF and PPM, but we can use the Python Imaging Library (PIL) to read other files.

The name of the PIL module is `Image`*, but Tkinter defines an object with the same name. To avoid the conflict, you can use* `import...as` *like this:*

```
import Image as PIL
import ImageTk
```

The first line imports `Image` *and gives it the local name* `PIL`*. The second line imports* `ImageTk`*, which can translate a PIL image into a Tkinter PhotoImage. Here's an example:*

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. *Download* `image_demo.py`*,* `danger.gif` *and* `allen.png` *from* `http://thinkpython.com/code`*. Run* `image_demo.py`*. You might have to install* `PIL` *and* `ImageTk`*. They are probably in your software repository, but if not you can get them from* `pythonware.com/products/pil/`*.*

2. *In* `image_demo.py` *change the name of the second PhotoImage from* `photo2` *to* `photo` *and run the program again. You should see the second PhotoImage but not the first.*

   The problem is that when you reassign `photo` *it overwrites the reference to the first PhotoImage, which then disappears. The same thing happens if you assign a PhotoImage to a local variable; it disappears when the function ends.*

   To avoid this problem, you have to store a reference to each PhotoImage you want to keep. You can use a global variable, or store PhotoImages in a data structure or as an attribute of an object.

   This behavior can be frustrating, which is why I am warning you (and why the example image says "Danger!").

3. *Starting with this example, write a program that takes the name of a directory and loops through all the files, displaying any files that PIL recognizes as images. You can use a* `try` *statement to catch the files PIL doesn't recognize.*

   When the user clicks on the image, the program should display the next one.

4. *PIL provides a variety of methods for manipulating images. You can read about them at* `http://pythonware.com/library/pil/handbook`*. As a challenge, choose a few of these methods and provide a GUI for applying them to images.*

Solution: `http://thinkpython.com/code/ImageBrowser.py`*.*

Exercise 19.5. *A vector graphics editor is a program that allows users to draw and edit shapes on the screen and generate output files in vector graphics formats like Postscript and SVG.*

Write a simple vector graphics editor using Tkinter. At a minimum, it should allow users to draw lines, circles and rectangles, and it should use `Canvas.dump` *to generate a Postscript description of the contents of the Canvas.*

As a challenge, you could allow users to select and resize items on the Canvas.

Exercise 19.6. *Use Tkinter to write a basic web browser. It should have a Text widget where the user can enter a URL and a Canvas to display the contents of the page.*

You can use the `urllib` *module to download files (see Exercise 14.6) and the* `HTMLParser` *module to parse the HTML tags (see* `docs.python.org/lib/module-HTMLParser.html`*).*

At a minimum your browser should handle plain text and hyperlinks. As a challenge you could handle background colors, text formatting tags and images.

Appendix A

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a `def` statement yields 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.
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.

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 compile 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 anything 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 that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section 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, an infinite recursion will cause the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then 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 or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. 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 invoked it, and then the function that invoked *that*, and so on. In other words, it traces the sequence of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

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. 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 or method. 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.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use `dir` to list the attributes that do exist.

If an AttributeError indicates that an object has `NoneType`, that means that it is `None`. One common cause is forgetting 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 `docs.python.org/lib/module-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 invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very 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 rules of precedence.

A.3.3 I've got a function or method 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 `return` value 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. A fresh pair of eyes is just the thing.

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.

Appendix 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 (2011). 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，你可能想继续读那一本。

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空间需求。 见：`http://en.wikipedia.org/wiki/Analysis_of_algorithms`。

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，以指导设计决策。

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年美国总统选举期间，当候选人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Obama，因为他很快回答， "我认为冒泡排序是错误的方法"。见：`http://www.youtube.com/watch?v=k4RRi_ntQc8`。

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 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."

这是对的：冒泡排序概念上很简单，但是对于大数据集非常慢。 Schmidt可能在寻找的答案是"基数排序（radix sort）"（http://en.wikipedia.org/wiki/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算法直接进行有意义的比较， 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（worst case）**。 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 to compare the functions **asymptotically** as the 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

The good thing about this kind of comparison 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 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。

[2] 但是如果你面试的时候被问及类似的问题， 我想更好的答案是，"对一百万整数排序的最快的方法是 使用我正在用的编程语言所提供的任何排序函数。 对于大多数的应用，其性能已经足够了，但是如果对于我的应用它太慢了， 我将用一个分析器查看时间都花在哪儿了。 如果看上去一个更快的排序算法对系能有显著的影响， 那么我将找一个好的基数排序实现。"

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且用输入的规模表示它们的运行时间： 算法A用$100n + 1$步解决一个规模为n的问题； 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看上去相当坏，它用了比算法B长10倍的时间。 但是对于$n = 100$，它们几乎相同， 对于更大的值，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 $an^2 > bn$.

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 asymptotic 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（**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标记**（**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 **quadratic**, which is a fancy word for functions with the leading term n^2.

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.

下表以越来越坏的顺序显示了算法分析中最通常的一些增长阶数。

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	"en log en"
$O(n^2)$	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:*

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 基本Python运算的分析

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。

If you use the same loop to "add" a list of strings, the run time is quadratic because string concatenation is linear.

如果你用相同的循环"累加"一个字符串列表，运行时间是二次的，因为字符串叠加是线性的。

The string method join is usually faster because it is linear in the total length of the strings.

字符串的join方法通常更快，因为它与整个字符串的长度是线性关系。

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.

作为一个经验法则，如果循环体是$O(n^a)$，那么整个循环是$O(n^{a+1})$。例外是如果你能展示循环经过常数次迭代后退出。如果一个循环运行k次，无论n是什么，那么即使对于很大的k，循环也是$O(n^a)$。

Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k.

乘以k不改变增长阶数，除以k也不会。因此，如果循环体是$O(n^a)$并且运行n/k次，那么即使对于很大的k，循环是$O(n^{a+1})$。

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。

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.

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 we say that the "amortized" time for one 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 `copy` is proportional to the number of elements, but not the size of the elements (it copies references, not the elements themselves).
  `copy`的运行时间正比于元素的个数，而非元素的大小（它）拷贝引用，而非元素自身。
- 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 linear because they return new lists; `iterkeys`, `itervalues` and `iteritems` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear. Using the "iter" functions saves some overhead, but it doesn't change the order of growth unless the number of items you access is bounded.
  `keys`、`values`和`items`是线性的，因为它们返回新的列表。`iterkeys`、`itervalues`和`iteritems`是常数时间，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。使用"iter"函数省些时间，但是如果你访问的项数没有界限，它不会改变增长阶数。

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B.4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**）是一个算法，其接受一个集合以及一个目标项， 并决定该目标项是否在集合中，通常返回目标的索引值。

The simplest search algorithm is a "linear search," which traverses the items of the collection in order, stopping if it finds the target. In the worst case it has to traverse the entire collection, so the run time is linear.

最简单的搜索算法是"线性搜索"，其按顺序遍历集合中的项， 如果找到目标则停止。最坏的情况下，它不得不遍历全部集合， 所以运行时间是线性的。

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

序列的`in`运算符使用线性搜索。 字符串方法，如`find`和`count`也是这样。

If the elements of the sequence are in order, you can use a **bisection search**, which is $O(\log n)$. Bisection search is similar to the algorithm you probably use to look a word up in a dictionary (a real 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。

Exercise B.3. *Write a function called* `bisection` *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!*

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的以及为什么它们的性能如此之好， 我以一个简单的map实现开始并逐步改进它，直到它成为一个哈希表。

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:

我使用Python来演示这些实现，但是在真实情况下， 你不会用Python写如此的代码。你只需用字典！ 因此对于本章剩下的内容，你需要假设字典并不存在， 并且你想实现一个数据结构，将关键字映射为值。 你需要实现的运算如下：

`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`.

`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(target)`: Look up and return the value that corresponds to key `target`. With a Python dictionary, `d`, this operation is written `d[target]` or `d.get(target)`.

`get(target)`: 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(object):

    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 (see http://en.wikipedia.org/wiki/Red-black_tree) 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：http://en.wikipedia.org/wiki/Red-black_tree） 能在log时间内实现add and get，但是这仍然不如常数时间好， 因此让我妈继续。

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(object):

    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Pythong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 equal return the same hash value, but the converse is not necessarily true: two different objects 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(object):

    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 wrap 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

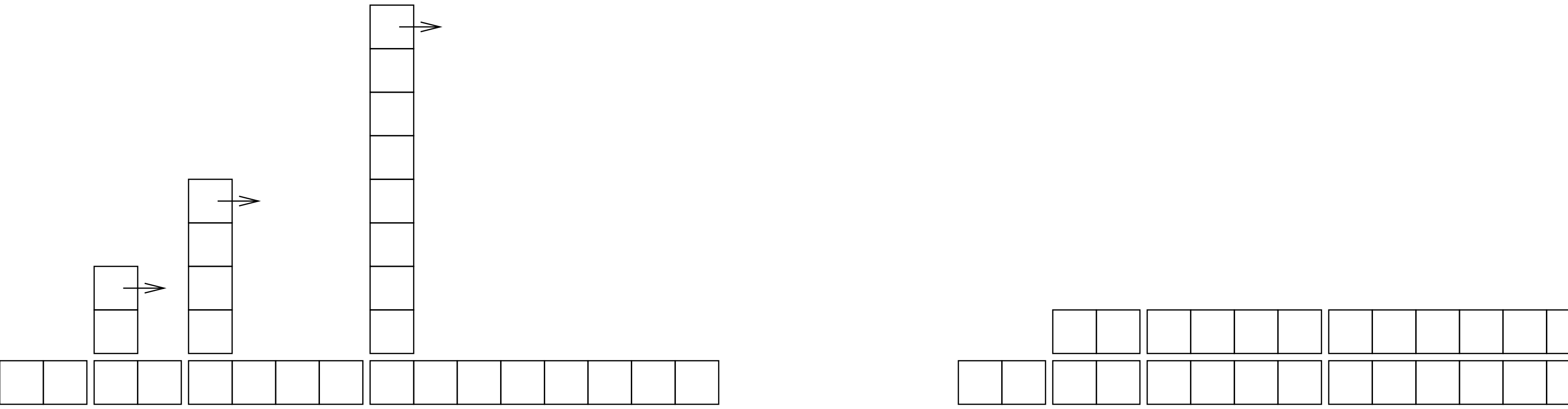

Figure B.1: The cost of a hashtable add.

作单元）然后增加第3项（一个额外单语）。 增加下一项花费1个单元，所以对于4项总共需要6个单元。

The next `add` costs 5 units, but the next three are only one unit each, so the total is 14 units for the first 8 adds.

下一个`add`花费5个单元，但是之后的3个每个只需要1个单元， 所以前8个add总共需要14个单元。

The next `add` costs 9 units, but then we can add 7 more before the next resize, so the total is 30 units for the first 16 adds.

下一个`add`花费9个单元， 但是之后在下一次resize之前， 可以增加额外的7个， 所以前16个add总共是30个单元。

After 32 adds, the total cost is 62 units, and I hope you are starting to see a pattern. After n adds, where n is a power of two, the total cost is $2n-2$ units, so the average work per add is a little less than 2 units. When n is a power of two, that's the 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, amortizing 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.

You can download my implementation of HashMap from `http://thinkpython/code/Map.py`, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

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。

Appendix C

Lumpy

Throughout the book, I have used diagrams to represent the state of running programs.

贯穿全书地，我使用图来表示运行的程序的状态。

In Section 2.2, we used a state diagram to show the names and values of variables. In Section 3.10 I introduced a stack diagram, which shows one frame for each function call. Each frame shows the parameters and local variables for the function or method. Stack diagrams for recursive functions appear in Section 5.9 and Section 6.5.

在2.2节，我们使用状态图来显示变量的名和值。在3.10节，我介绍了栈图，其为每个函数调用显示一个框架。每个框架为该函数或者方法显示了形参以及局部变量。对于递归函数的栈图出现在5.9和6.5节。

Section 10.2 shows what a list looks like in a state diagram, Section 11.4 shows what a dictionary looks like, and Section 12.6 shows two ways to represent tuples.

10.2节显示在栈图中，列表看起来的样子，11.4节显示字典的样子，12.6节显示表示元组的两个方法。

Section 15.2 introduces object diagrams, which show the state of an object's attributes, and their attributes, and so on. Section 15.3 has object diagrams for Rectangles and their embedded Points. Section 16.1 shows the state of a Time object. Section 18.2 has a diagram that includes a class object and an instance, each with their own attributes.

15.2节介绍了对象图，其展示了一个对象的属性的状态以及他们的属性等。15.3节有对于矩形以及他们嵌入的点的对象图。16.1 节显示了一个Time对象的状态。18.2有一个图，其包括一个类对象以及一个实例，每个都有他们自己的属性。

Finally, Section 18.8 introduces class diagrams, which show the classes that make up a program and the relationships between them.

最后，18.8节介绍了类图，其展示了组成程序的类以及它们之间的关系。

These diagrams are based on the Unified Modeling Language (UML), which is a standardized graphical language used by software engineers to communicate about program design, especially for object-oriented programs.

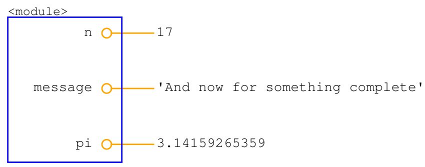

Figure C.1: State diagram generated by Lumpy.

这些图基于统一建模语言（UML），其是一个标准的图示语言，软件工程师用其交流程序设计，特别是对于面向对象的程序。

UML is a rich language with many kinds of diagrams that represent many kinds of relationship between objects and classes. What I presented in this book is a small subset of the language, but it is the subset most commonly used in practice.

UML是内容非常丰富的语言，具有多种图来表示许多种对象和类之间的关系。本书中我表示的只是该语言的一个小子集，但是它是实践中最常用的子集。

The purpose of this appendix is to review the diagrams presented in the previous chapters, and to introduce Lumpy. Lumpy, which stands for "UML in Python," with some of the letters rearranged, is part of Swampy, which you already installed if you worked on the case study in Chapter 4 or Chapter 19, or if you did Exercise 15.4,

本附录的目的是回顾之前章节中给出的图并介绍Lumpy。Lumpy，表示"UML in Python"，是对一些字母的重排列，是Swampy的一部分，如果你完成了第4章或第19的案例学习，或者做了练习15.4，你已经安装好了Swampy。

Lumpy uses Python's `inspect` module to examine the state of a running program and generate object diagrams (including stack diagrams) and class diagrams.

Lumpy使用Python的`inspect`模块来检查一个正在运行的程序的状态并生成对象图（包括栈图）以及类图。

C.1 State diagram 状态图

Here's an example that uses Lumpy to generate a state diagram.

此处是一个使用Lumpy生成状态图的例子。

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

message = 'And now for something completely different'
n = 17
pi = 3.1415926535897932

lumpy.object_diagram()
```

The first line imports the Lumpy class from `swampy.Lumpy`. If you don't have Swampy installed as a package, make sure the Swampy files are in Python's search path and use this `import` statement instead:

Figure C.2: Stack diagram.

第一行从`swampy.Lumpy`导入Lumpy类。 如果你没有作为包安装Swamp，确定Swampy文件位于Python的搜索目录 并使用此`import`语句代替：

```
from Lumpy import Lumpy
```

The next lines create a Lumpy object and make a "reference" point, which means that Lumpy records the objects that have been defined so far.

接下来的几行生成一个`Lumpy`对象并生成一个"参考"点， 其意味着Lumpy记录目前已经被定义的对象。

Next we define new variables and invoke `object_diagram`, which draws the objects that have been defined since the reference point, in this case `message`, `n` and `pi`.

接着我们定义新的变量并调用`object_diagram`， 来画这些自参考点之后定义的对象。

Figure C.1 shows the result. The graphical style is different from what I showed earlier; for example, each reference is represented by a circle next to the variable name and a line to the value. And long strings are truncated. But the information conveyed by the diagram is the same.

图C.1展示了结果。这个图的样式与我之前展示的不同。 例如，每个引用用一个变量名以及一个连接到值的圆表示。 长字符串被截去了。但是此图传递的信息是相同的。

The variable names are in a frame labeled `<module>`, which indicates that these are module-level variables, also known as global.

变量名位于一个标为 `<module>`的线框里， 其指明这些是模块级的变量，也被认为是全局变量。

You can download this example from `http://thinkpython.com/code/lumpy_demo1.py`. Try adding some additional assignments and see what the diagram looks like.

你可以从`http://thinkpython.com/code/lumpy_demo1.py`下载这个例子。 试着增加一些额外的赋值并看此图看起来像什么样子。

C.2 Stack diagram 栈图

Here's an example that uses Lumpy to generate a stack diagram. You can download it from `http://thinkpython.com/code/lumpy_demo2.py`.

这是一个使用Lumpy生成栈图的例子。 你可以从`http://thinkpython.com/code/lumpy_demo2.py`下载它。

```
from swampy.Lumpy import Lumpy

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

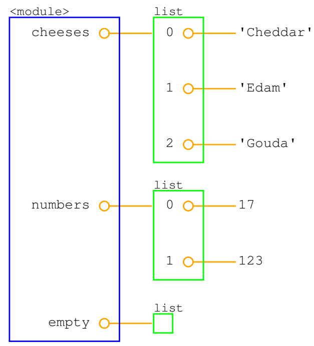

Figure C.3: Object diagram.

```
        lumpy.object_diagram()
    else:
        print n
        countdown(n-1)

lumpy = Lumpy()
lumpy.make_reference()
countdown(3)
```

Figure C.2 shows the result. Each frame is represented with a box that has the function's name outside and variables inside. Since this function is recursive, there is one frame for each level of recursion.

图C.2显示了结果。每个线框用一个盒子表示，它外面有函数名， 里面有变量。既然此函数是递归的，每层递归有一个线框。

Remember that a stack diagram shows the state of the program at a particular point in its execution. To get the diagram you want, sometimes you have to think about where to invoke `object_diagram`.

记住一个栈图显示了程序在它执行中一个特定点的状态。 为了获得你希望的图，有时你必须考虑在哪儿调用`object_diagram`。

In this case I invoke `object_diagram` after executing the base case of the recursion; that way the stack diagram shows each level of the recursion. You can call `object_diagram` more than once to get a series of snapshots of the program's execution.

在这个例子中，我在执行递归的基本条件后调用`object_diagram`， 这使得栈图显示递归的每一层。你可以不止一次调用`object_diagram` 来获得一系列程序执行的快照。

C.3 Object diagrams 对象图

This example generates an object diagram showing the lists from Section 10.1. You can download it from `http://thinkpython.com/code/lumpy_demo3.py`.

此例生成一个显示来自 10.1节列表的对象图。 你可以从`http://thinkpython.com/code/lumpy_demo3.py`下载它。

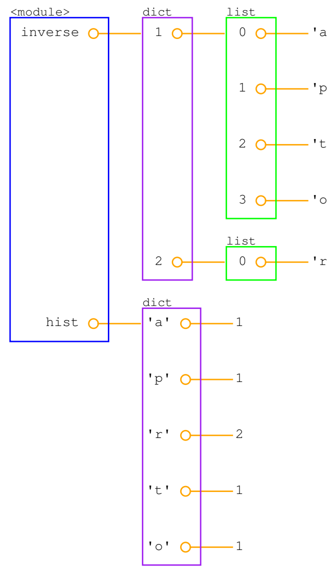

Figure C.4: Object diagram.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
empty = []

lumpy.object_diagram()
```

Figure C.3 shows the result. Lists are represented by a box that shows the indices mapping to the elements. This representation is slightly misleading, since indices are not actually part of the list, but I think they make the diagram easier to read. The empty list is represented by an empty box.

图C.3显示结果。 列表被表示成一个盒子，展示了映射到元素的索引。 这一表示有点儿误导，因为索引不是列表的实际部分， 但是我想它们使得图更容易被读。 空列表被表示为一个空盒子。

And here's an example showing the dictionaries from Section 11.4. You can download it from `http://thinkpython.com/code/lumpy_demo4.py`.

这是一个显示来自11.4节的字典的例子。 你可以从`http://thinkpython.com/code/lumpy_demo4.py`下载它。

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

hist = histogram('parrot')
```

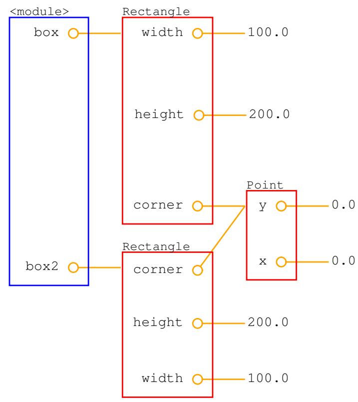

Figure C.5: Object diagram.

```
inverse = invert_dict(hist)

lumpy.object_diagram()
```

Figure C.4 shows the result. `hist` is a dictionary that maps from characters (single-letter strings) to integers; `inverse` maps from integers to lists of strings.

图C.4显示结果。 `hist`是一个字典，从字符（单字母字符串）映射到整数。 `inverse`从整数映射到字符串列表。

This example generates an object diagram for Point and Rectangle objects, as in Section 15.6. You can download it from `http://thinkpython.com/code/lumpy_demo5.py`.

此例生成一个Point和Rectangle对象的对象图，如15.6节所示。 你可以从`http://thinkpython.com/code/lumpy_demo5.py`下载它。

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

box2 = copy.copy(box)

lumpy.object_diagram()
```

Figure C.5 shows the result. `copy.copy` make a shallow copy, so `box` and `box2` have their own `width` and `height`, but they share the same embedded Point object. This kind of sharing is usually fine with immutable objects, but with mutable types, it is highly error-prone.

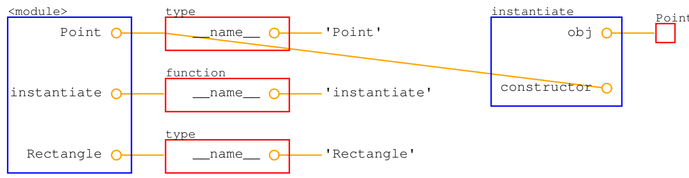

Figure C.6: Object diagram.

图C.5显示结果。 `copy.copy`进行浅拷贝，所以`box`和`box2`有它们自己的`width`和`height`，但是它们共享相同的嵌入Point对象。 这种共享通常对于不可变对象是好的，但是对于可变类型，它非常容易出错。

C.4 Function and class objects 函数和类对象

When I use Lumpy to make object diagrams, I usually define the functions and classes before I make the reference point. That way, function and class objects don't appear in the diagram.

当我用Lumpy生成对象图时，我通常在生成参考点之前定义函数和类。 这样，函数和类对象不出现在图中。

But if you are passing functions and classes as parameters, you might want them to appear. This example shows what that looks like; you can download it from `http://thinkpython.com/code/lumpy_demo6.py`.

但是如果你正作为形参传递函数和类，你可能希望它们出现。 此例展示那看起来是什么样：

```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

class Point(object):
    """Represents a point in 2-D space."""

class Rectangle(object):
    """Represents a rectangle."""

def instantiate(constructor):
    """Instantiates a new object."""
    obj = constructor()
    lumpy.object_diagram()
    return obj

point = instantiate(Point)
```

Figure C.6 shows the result. Since we invoke `object_diagram` inside a function, we get a stack diagram with a frame for the module-level variables and for the invocation of `instantiate`.

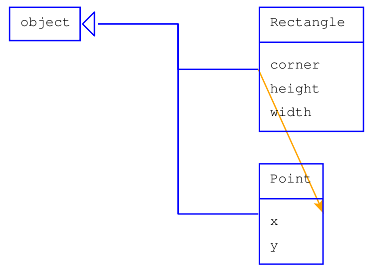

Figure C.7: Class diagram.

图C.6显示这一结果。 既然我们在一个函数中调用`object_diagram`， 我们为模块级的变量和`instantiate`的调用获得一个具有框架的栈图。

At the module level, `Point` and `Rectangle` refer to class objects (which have type `type`); `instantiate` refers to a function object.

在模块级，`Point`和`Rectangle`指类对象（其类型是`type`）。 `instantiate`指一个函数对象。

This diagram might clarify two points of common confusion: (1) the difference between the class object, `Point`, and the instance of Point, `obj`, and (2) the difference between the function object created when `instantiate` is defined, and the frame created with it is called.

此图可能会澄清通常易混淆的两点： （1）类对象`Point`和Point的实例`obj`之间的不同；（2）当`instantiate`被定义是生成的函数对象和其被调用时生成的框架之间的不同。

C.5 Class Diagrams 类图

Although I distinguish between state diagrams, stack diagrams and object diagrams, they are mostly the same thing: they show the state of a running program at a point in time.

虽然我区分了状态度、栈图以及类图，但是它们基本是同一个东西： 它们显示了一个正在运行的程序在某一时间点的状态。

Class diagrams are different. They show the classes that make up a program and the relationships between them. They are timeless in the sense that they describe the program as a whole, not any particular point in time. For example, if an instance of Class A generally contains a reference to an instance of Class B, we say there is a "HAS-A relationship" between those classes.

类图不一样。它们显示了组成程序的类以及它们之间的关系。 它们作为一个整体描述程序，在这个意义上，它们是永恒的， 不是任何特殊的时间点。例如，如果一个类A的实例通常包含类B的实例的一个引用， 我们说在这些类之间有一个"HAS-A关系"。

Here's an example that shows a HAS-A relationship. You can download it from `http://thinkpython.com/code/lumpy_demo7.py`.

这是一个展示HAS-A关系的示例。 你可以从`http://thinkpython.com/code/lumpy_demo7.py`下载它。

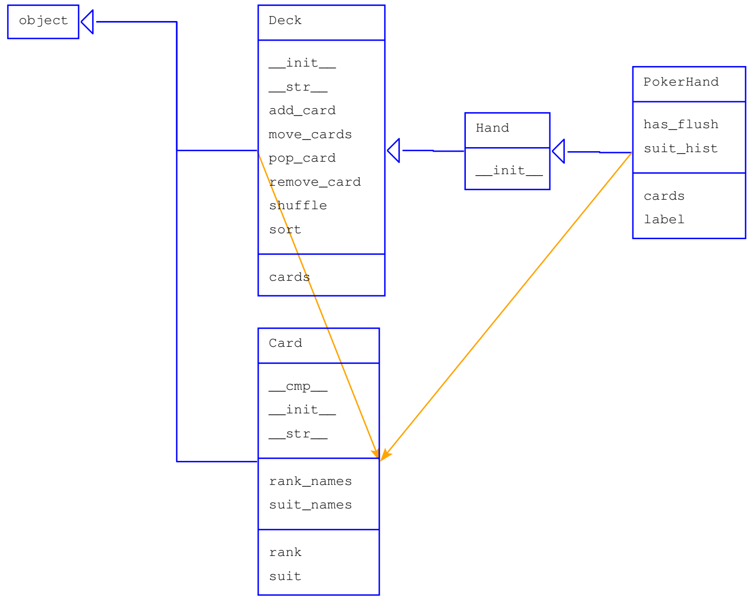

Figure C.8: Class diagram.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

lumpy.class_diagram()
```

Figure C.7 shows the result. Each class is represented with a box that contains the name of the class, any methods the class provides, any class variables, and any instance variables. In this example, `Rectangle` and `Point` have instance variables, but no methods or class variables.

图C.7显示了结果。每个类被表示成一个盒子， 包含类名、类提供的任何方法，类变量以及任何实例变量。 在此例中，`Rectangle`和`Point`有实例变量， 但是没有方法或类变量。

The arrow from `Rectangle` to `Point` shows that Rectangles contain an embedded Point. In addition, `Rectangle` and `Point` both inherit from `object`, which is represented in the diagram with a triangle-headed arrow.

从`Rectangle`到`Point`的 箭 头 显 示Rectangles包 含 一 个 嵌 入 的Point。 另外，`Rectangle`和`Point`都是继承自`object`， 这在图中用一个三角头的箭头表示。

Here's a more complex example using my solution to Exercise 18.6. You can download the code from `http://thinkpython.com/code/lumpy_demo8.py`; you will also need `http:`

//thinkpython.com/code/PokerHand.py.

这是一个更复杂的例子，使用了我的习题18.6的答案。你从http://thinkpython.com/code/lumpy_demo8.py可以下载代码；你也需要http://thinkpython.com/code/PokerHand.py。

```
from swampy.Lumpy import Lumpy

from PokerHand import *

lumpy = Lumpy()
lumpy.make_reference()

deck = Deck()
hand = PokerHand()
deck.move_cards(hand, 7)

lumpy.class_diagram()
```

Figure C.8 shows the result. PokerHand inherits from Hand, which inherits from Deck. Both Deck and PokerHand have Cards.

图C.8显示了结果。PokerHand继承自Hand，它继承自Deck。Deck和PokerHand都有Cards。

This diagram does not show that Hand also has cards, because in the program there are no instances of Hand. This example demonstrates a limitation of Lumpy; it only knows about the attributes and HAS-A relationships of objects that are instantiated.

此图没有显示Hand也有cards，因为在这个程序中，没有Hand的实例。此例展示了Lumpy的一个局限性，它只知道被实例化的对象的属性和HAS-A关系。

Index