Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.4.0

Think Python

How to Think Like a Computer Scientist

2nd Edition, Version 2.4.0

Allen Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

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

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

序

本书由来

1999 年 1 月，我正准备以 Java 教学生编程入门，课程已讲过三次，但我沮丧依旧。因为多人无法完成目标，虽有个别人优秀，但整体效果令人难以接受。

依我看，问题出在书本教案上. 书籍厚重，画蛇添足，细枝末节过多，编程思考不足。因此学生总是陷于困境：起步易，行进难，半途而废。学生学得囫囵吞枣，老师教得步步回首。

此次开课前两周，我决定自己写本书，目标如下：

- 少。十页易，百页难。
- 精。词汇早解释，术语早提炼。
- 缓。大而化小，小步慢跑。
- 专。聚焦思想，精简程序。

莫名其妙地选择了个题目像计算机专家一样思考 *(How to Think Like a Computer Scientist)*.

首版略显粗略，但是简单有效。学生知其然，而知其所以然，我也能够聚焦课堂于难点，趣题和操作实践。绝知此事要躬行，实践是必须而困难的，但也是最有趣的。

我基于 GNU 自由文档许可证发布此书，也就是说，任何人都可以基于此协议，复制，修改，分享此书。

后续的事情就有趣了，弗吉尼亚州的 Jeff Elkner 高校教师采用了本书，并转换为了 Python 教材。通过他给我的 Python 版书籍，我有了一段不一样的 Python 学习之旅。同时基于绿茶出版商，于 2001 年，我首次发布了 Python 版。

2003 年，我开始在欧林学院任职，并且首次教授 Python。和教授 Java 相比，惊喜非凡，学生乐于学习并乐在其中。

自此，我不断优化此书，校正错误，提高案例水准，并增加一些新的例子和练习。

因此本书有个通俗的名字 *Think Python*. 部分修改如下:

- 每章结尾增加了调试部分。重点讲解如何发现问题，减少错误，预防漏洞。
- 增加了一些习题。从简单测验到项目实操，均有涉及，同时，我给出了我的解决方案。
- 增加了一系列的案例：涵盖练习-解答-讨论三个阶段的训练。
- 探讨了程序开发流程以及基础设计模式。
- 增加了关于调试和算法解析的附录。

第二版 *Think Python* 有如下变化：

- 本书代码均已支持 Python 3.
- 网页版功能更加丰富，从而使初学者可以无需安装 Python 便可在浏览器运行。
- 对于章节 4.1 我将我常用的海龟绘图包 Swampy 换成了功能更丰富，应用更广泛的 `turtle` 包。
- 增加了新章节“利器”，主要介绍 Python 一些不常用但是高效的功能。

希望本书能对大家在学习编程知识，以及培养计算机专家一样思维模式方面有所裨益。

Allen B. Downey

欧林学院

致谢

感谢 Jeff Elkner 把这本 Java 书籍转换为 Python，并让我爱上了 Python.

感谢 Chris Meyers 写了 *How to Think Like a Computer Scientist* 部分章节。

感谢创建 GNU 自由文档协议以及创作共用的自由软件基金会。基于此协议以及知识共享协议，我才能和 Jeff，Chris 一起合作。

感谢整理 *How to Think Like a Computer Scientist* 的 Lulu 的编辑。

感谢耐心编辑 *Think Python* 的 O'Reilly Media 的朋友。

感谢使用本书早期版本的学生们，感谢所有提出建议和指正错误的朋友。

贡献者名单

过去数年，上百位读者为本书献策纠误。他们无私的奉献，令本书意义非凡。

如若您发现问题或者有好的提议，请发邮件到 `feedback@thinkpython.com`。一经采用，您将被加入贡献者名单 (除非要求匿名)。

建议您最好附上详细语句而不仅仅是章节和页码，以方便快速定位，感谢。

- Lloyd Hugh Allen 修正了 8.4 节的一处错误。
- Yvon Boulianne 修正了第 5 章的一处语义错误。
- Fred Bremmer 修正了 2.1 节的一处错误。
- Jonah Cohen 撰写了 Perl 脚本，将本书 LaTex 源码转成了美观的 HTML。
- Michael Conlon 修正了第 2 章的一处语法错误，并优化第一章的一处格式，初次提出关于口译技术的讨论。
- Benoît Girard 修正了 5.6 节一个重大错误。
- Courtney Gleason 和 Katherine Smith 编写了 `horsebet.py`，这是本书早期版本的一个案例脚本，此程序现在仍然可以在网站找到。
- Lee Harr 修正了多个错误，所以应该把他列为主要编辑者。
- James Kaylin 作为学生，修正了大量错误。
- David Kershaw 修正了 3.10 节的 `catTwice` 函数错误。
- Eddie Lam 修正了第 1,2,3 章大量错误，同时修复了 Makefile 文件，从而我们可以创建索引。同时帮助我们建立了版本库。
- Man-Yong 修正了 2.4 节的一个样例。
- David Mayo 指出第 1 章的某个词语“unconsciously”应该替换为“subconsciously”。
- Chris McAloon 修正了 3.9 和 3.10 节的错误。
- Matthew J. Moelter 作为一个长期贡献者，为本书提供了大量修订和建议。
- Simon Dicon Montford 发现了第 3 章的一个函数定义遗失和几个变量的命名错误。同时指出第 13 章的 `increment` 函数错误。
- John Ouzts 修正了第三章的“return value”的定义。
- Kevin Parks 为本书的发行优化提供了宝贵的意见。
- David Pool 对本书给了善意的鼓励，并指出了第一章词汇表中的一个错字。
- Michael Schmitt 修正了文件和异常章节的一个错误。
- Robin Shaw 指出 13.1 节中，使用了没有预先定义的 printTime 函数。
- Paul Sleigh 指出了第 7 章的一个错误，以及 Jonah Cohen's 的生成 HTML 的 Prel 脚本中的一个错误。
- Craig T. Snydal 在 Drew University 使用本书教学，并给变量命名和修正提供了宝贵意见。

- Ian Thomas 和他的学生在编程课程中使用本书，他们是首批检验本书后半部分的贡献者，同时他们修订了大量错误，提供了大量建议。
- Keith Verheyden 修正了第 3 章的一个错误。
- Peter Winstanley 指出了拉丁版第 3 章的一个长期存在的错误。
- Chris Wrobel l 修正了 I/O 和异常章节的一个错误。
- Moshe Zadka 撰写了字典章节的早期草稿，为本书早期工作做出了杰出贡献。
- Christoph Zwerschke 指出多个修正和教学建议，并解释了 *gleich* 和 *selbe* 的不同。
- James Mayer 修正了一堆拼写和排版问题，甚至包括贡献列表中的两个问题。
- Hayden McAfee 解决了一个在两个例子之间令人困惑的差异。
- Angel Arnal 作为国际翻译组织的一员，参与了西班牙语的翻译，并在翻译过程中修正了英语版的一些错误。
- Tauhidul Hoque 和 Lex Berezhny 绘制了第一章的插图，并优化了其他章节插图。
- Michele Alzetta 博士修正了第 8 章的一个错误，并指出了 Fibonacci and Old Maid 案例的一些问题。
- Andy Mitchell 指出了第 1 章的一个错字，第 2 章的一个错误例子。
- Kalin Harvey 辨析了第 7 章的一个歧义，并修正了几个错字。
- Christopher P. Smith 修正了错字，并给予 Python2.2 进行了修改。
- David Hutchins 修正了序中的一个错词。
- Gregor Lingl 在奥地利的维也纳的一个高校教授 Python，他翻译了德语版，并在翻译过程中，修正了第 5 章的多个错误。
- Julie Peters 修正了序中的一个错字。
- Florin Oprina 优化了 `makeTime`，修正了 `printTime` 的错误，以及一个错字。
- D. J. Webre 辨析了第 3 章的一个歧义词。
- Ken 修正了 8,9,11 章的一堆错误。
- Ivo Wever 修正了第 3 章的错字和第 5 章的歧义词。
- Curtis Yanko 辨析了第 2 章一个歧义。
- Ben Logan 修正了本书 HTML 版的大量错误。
- Jason Armstrong 发现第 2 章遗漏的一个词汇。
- Louis Cordier 指出了第 16 章的一个描述和代码不一致的问题。
- Brian Cain 修正了第 2,3 章的数个错误。
- Rob Black 修正了一系列错误，以及优化了 Python2.2 版部分内容。
- Jean-Philippe Rey [巴黎中央理工学院] 针对 Python2.2 进行了部分优化。
- Jason Mader [乔治华盛顿大学] 提供了大量修改意见。
- Jan Gundtofte-Bruun 指出"a error" 错误。

- Abel David 和 Alexis Dinno 指出"matrix" 的复数是"matrices"，而不是"matrixes"。这个错误存在多年，但是两人在同一天指出此错误，神奇啊！
- Charles Thayer 建议我们在一些声明末尾勿用分号，以及避免使用"argument" 和"parameter"。
- Roger Sperberg 指出了第 3 章的一个逻辑问题。
- Sam Bull 指出第 2 章的一处歧义表述。
- Andrew Cheung 修正了"use before def." 的两个实例。
- C. Corey Capel 指出“Third Theorem of Debugging”章节的遗漏词汇，以及第 4 章的一个错词。
- Alessandra 澄清了一些"Turtle" 歧义。
- Wim Champagne 修正了字典例子中的一个"brain-o" 错误。
- Douglas Wright 修正了 `arc` 中的一个"floor" 错误。
- Jared Spindor 清理了数个画蛇添足。
- Lin Peiheng 提供了多个建议。
- Ray Hagtvedt 修正了两个错误，提出一个优化。
- Torsten Hübsch 指出 Swampy 中的一处不一致。
- Inga Petuhhov 修正了第 14 章的一个案例问题。
- Arne Babenhauserheide 修正了数个错误。
- Mark E. Casida 敏锐发现了多个重复词汇。
- Scott Tyler 补充了一处遗漏词汇，并修正了大量错误。
- Gordon Shephard 多次发送邮件，指出错误。
- Andrew Turner 指出第 8 章的一处错误。
- Adam Hobart 修正了 `arc` 中的一处错误。
- Daryl Hammond 和 Sarah Zimmerman 指出我太早使用 `math.pi`，同时 Zim 修正了一个拼写错误。
- George Sass 修复了调试一章的一处错误。
- Brian Bingham 提供 11.5习题。
- Leah Engelbert-Fenton 指出我误用 `tuple` 为变量名的问题，以至于发现大量类似问题”use before def “。
- Joe Funke 指出一个拼写错误。
- Chao-chao Chen 指出 Fibonacci 例子中的一处不一致。
- Jeff Paine 指出了 space 和 spam 之间的不同。
- Lubos Pintes 指出了一处拼写错误。
- Gregg Lind 和 Abigail Heithoff 提供 14.3习题。
- Max Hailperin 修正了大量错误，同时提供了大量建议。Max 是 *Concrete Abstractions* 一书的作者之一，读完此书，你可以阅读这本非凡之作。

- Chotipat Pornavalai 修正了错误信息中的一处问题。
- Stanislaw Antol 提供了大量建议。
- Eric Pashman 修正了 4-11 章的大量错误。
- Miguel Azevedo 发现了多个拼写错误。
- Jianhua Liu 修正了大量错误。
- Nick King 发现了一处词汇遗漏。
- Martin Zuther 提供了大量建议。
- Adam Zimmerman 指出了"instance" 中的一处不一致问题，以及数个错误。
- Ratnakar Tiwari 提供了退化三角形说明脚注。
- Anurag Goel 提供了 `is_abecedarian` 的另外一种方案，并修正了多个错误。同时他知晓如何拼写 Jane Austen。
- Kelli Kratzer 指出了一处拼写错误。
- Mark Griffiths 澄清了第 3 章的一个令人困惑的案例。
- Roydan Ongie 指出了 Newton 方法的一处错误。
- Patryk Wolowiec 修正了 HTML 版中的一个错误。
- Mark Chonofsky 让我知晓了 Python3 的一个关键字。
- Russell Coleman 让我增长了几何知识。
- Nam Nguyen 指出了一个拼写错误，并指出我用装饰器模式前，未有说明。
- Stéphane Morin 修正了数个错误，并提供了多个建议。
- Paul Stoop 修正了 `uses_only` 中的一处拼写错误。
- Eric Bronner 指出了关于操作顺序的撰文中的一处歧义。
- Alexandros Gezerlis 为他提交的建议数量和质量设定了新的标准，深表谢意。
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa 提供了大量修订。
- Daniel Neilson 修正了操作顺序处的一个错误。
- Will McGinnis 指出 `polyline` 在两处定义不同。
- Frank Hecker 指出某个习题异于规定，同时发现了多个异常链接。
- Animesh B 澄清了一个令人困惑的案例。
- Martin Caspersen 发现两处四舍五入的错误。
- Gregor Ulm 提供多个修订和建议。
- Dimitrios Tsirigkas 建议我修正一个练习。
- Carlos Tafur 提供了大量修订和建议。
- Martin Nordsletten 发现了习题答案中的一处异常。

- Sven Hoexter 指出某处误用 `input` 这一内置函数。
- Stephen Gregory 指出 Python3 中的 `cmp` 的问题。
- Ishwar Bhat 修订了关于费马大定理的描述。
- Andrea Zanella 将此书翻译为意大利文，同时修正了多个错误。
- 感谢 Melissa Lewis 和 Luciano Ramalho 对第二版提供的建议和帮助。
- 感谢来自 PythonAnywhere 的 Harry Percival 的帮助，使读者可以在浏览器运行 Python。
- Xavier Van Aubel 为第二版提供了多次修正。
- William Murray 纠正了取整的定义。
- Per Starbäck 令我了解了 Python3 中的通用换行符。
- Laurent Rosenfeld 和 Mihaela Rotaru 将本书翻译为了法文，并纠正了诸多错误。

  另外为本书献策纠误的人包括：Czeslaw Czapla, Dale Wilson, Francesco Carlo Cimini, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber, and Eric Ransom。

目录

第 1 章　程序思维

本书期望能带你做到像计算机专家一样思考。这意味着你要整合数学，工程和自然科学的知识去综合思考。比如数学，计算机专家用专业语言去表达观点 (尤其关于计算)。比如工程师，他们设计原件，组装组件，构建系统，权衡各种方案。比如科学家，他们观察复杂系统行为，小心假设，积极求证。

计算机专家最重要的能力是**解决问题**。解决问题意味着要分析问题，提出创造性解决方案，并能清晰准确表述。事实证明，学习编程是锻炼解决问题能力的极好机会。这也是本章叫做程序思维的缘由。

一方面，我们将学习编程这一技能，另一方面，我们将利用编程技能来实现目标。随着不断进步，前景将越来越明朗。

1.1　何为程序？

程序 简而言之，就是一堆执行运算的指令。运算可以是数学计算，例如求解方程或多项式。也可以是符号运算，例如检索或者替换文档中的语句词汇。也可以是图形运算，比如处理图片，播放视频。

不同语言虽各有特色，但基础指令相差无几：

输入: 从键盘，文件，网络或者其他设备中获取数据。

输出: 将数据显示于屏幕，保存至文件，发送到网络等。

数学计算: 加减乘除等数学运算。

条件选择: 只能在命运分支的安排下，低头前进。

循环执行: 在宿命轮回中不断改变，以摆脱命运的枷锁。

无论相信与否，基础指令不过这些。不管是哪种编程语言，也无论何等复杂，都由这些基础指令构成。所以你便了解了何为编程，那便是将庞大复杂的任务不断分解，不断细化，直至细化为这几个基础指令为止。

1.2　运行 Python

巧妇难为无米之炊，运行 Python 程序必然需要先安装 Python 和其他软件，熟悉操作系统和命令行的人还好，对于初学者，同时学习操作系统和编程，必然是件痛苦万分的事情。

我建议先在浏览器上进行 Python 编程学习，后续再按步骤进行 Python 安装。

在线运行 Python 代码的网站很多，如果无从下手，我推荐 PythonAnywhere。入门指导可以查阅`http://tinyurl.com/thinkpython2e`

本书有两个版本，分别是 Python 2 和 Python 3, 内容相似，可以互相借鉴。细微差异，初学时需要关注。本书主要面向 Python 3，同时包括了一些关于 Python 2 的内容。

运行 Python 代码的程序叫做 Python **解释器** 。通常点击图标或者在命令行输入 `python` 来启动程序。启动后，可以看到如下输出：

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

前三行主要是 Python 解释器信息以及操作系统信息，内容因人而异。注意版本信息，比如 `3.4.0`，第一位的 3 表示此解释器版本是 Python 3，如果是 2, 则表示 Pyhton 2。

最后一行是个**提示符**，表示一切就绪，只待代码。如果输入一行代码，并敲击 Enter 键，解释器就会显示运行结果。

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

到此，你应该学会了如何启动 Python 解释器，并运行代码了。

1.3 初识

通常，学习一门新的编程语言，遇到的第一个程序就是"Hello, World!"。Python 语言这样表示：

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

这就是 **print 语法**，不要误会是打印到纸上，而是显示在屏幕上，结果就是这个语句：

```
Hello, World!
```

单引号表示语句的开始和结束，并不会显示在输出上。

括号表示 `print` 是个函数，后续章节 3介绍。

Python 2 中，print 语法明显不同，它不是函数，所以后边不会有括号。

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

这种语言版本之间的差异需要注意，但是当前，我们可以开始学习 Python 编程了。

1.4 算术运算

讲完了 “Hello, World”，我们讲讲算术。Python 通过**运算符** 实现加减乘除等计算。

运算符 `+`, `-`, 和 `*` 分别表示加，减和乘法，参看以下样例：

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

运算符 / 表示除法:

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

对于结果为 42.0，而不是 42 的疑惑，下章解释。

运算符 ** 表示幂运算，表示一个数字的次方。

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

运算符 ^ 在某些编程语言中表示幂运算，但是 Python 中，表示位运算的“异或”操作。如果不了解位运算，那下面的例子只会让你困惑万分。

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

本书不准备讲解位运算，如果想了解，可以前往 http://wiki.python.org/moin/BitwiseOperators。

1.5 值和类型

值 是程序的基本要素，例如字母，数字。前文看到的 2，42.0，以及 'Hello, World!'，都属于值。

而这些值分属不同**类型**: 2 是**整树**，42.0 是**浮点数**，而 'Hello, World!' 是**字符串**，也就是一堆字符的集合。

想知道值是何种类型，输入命令:

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

”class“表示类别，值的类别就是类型。

所以整数属于整型，字符串属于字符型，浮点数属于浮点型。

'2' 和 '42.0' 是什么类型呢？看似是数字，其实是被引号括起来的字符串。

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

哈哈，是字符串吧。

当我们想使用很大的整数时，一般会在数字间加逗号分隔，类似 1,000,000。在 Python 中，这虽然不合整型之规，却容易理解:

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

很神奇！Python 解释器把 1,000,000 当作了一串逗号分隔的整数。后续我们还会多次接触这种格式。

1.6 规范语言和自然语言

自然语言 指人们沟通需要的语言，比如英语，西班牙语，法语。这通常是人类自然进化而来，并不是出于特定目的而创造的。

规范语言 指人们为了特定目的而创造的特定格式的语言，比如数学家们发明的数学符号，可以很方便表示数字和符号的关系。而化学家们发明的分子结构式也是一种规范语言，更重要的是：

> **人们为了表示机器运算而创造的规范语言就是编程语言。**

规范语言一般有严格的**句法** 来管理语句的结构。例如，数学表示，$3+3=6$ 是正确的，而 $3+=3\$6$ 则不正确。又比如在化学中，H_2O 正确，而 ${}_2Zz$ 则明显有问题。

句法包括两部分，**词汇** 和规则。词汇是语言的基本，比如单词，数字和化学元素等。$3+=3\$6$ 的问题在于，$ 在数学中不是一个合法的词汇 (据我所知)。同样，${}_2Zz$ 在化学领域不正确，也是因为 Zz 无法对应相应的元素。

句法的另外一个要素是**规则**，即把词汇有效组织起来所遵循的结构。公式 $3+/3$ 不合句法，不是因为 $+$ 和 $/$ 不正确，而是因为先后顺序问题。同样，上述化学公式不仅仅是词汇胡编乱造，其规则也有问题，元素下标不能在元素之前，而应放在后边。

This is @ well-structured Engli$h sentence with invalid t*kens in it. 这句话每个字母（词汇）都正确，但是拼在一起，就不正确了。

我们读一句话或者一个公式的时候，会自觉地分析句子结构（自然语言中往往是潜意识行为）。这个分析的过程，叫做**解析**。

规范语言和自然语言在词汇，规则以及句法等方面较为相似，但是也有很多不同之处：

歧义: 自然语言的表述不太精确，人人往往需要上下文和相关情境确定对方的意思。但是规范语言中，句法确定，意思清晰，不会出现同一句话表示不同意思的情况。

冗余: 为了让对方捕捉到自己的真实意思，避免误会，自然语言中往往需要补充更多相关信息，来辅助沟通。而规范语言则更加简炼而精确。

比喻: 自然语言中充满了俗语和比喻，我说"The penny dropped"，不是说谁钱掉了 (比喻恍然大悟)。规范语言则没有类似的表述。

我们生活在自然语言的环境下，突然要适应规范语言，有很多困难需要克服。而两者的不同，就好比诗词和散文：

诗词: 关注平仄，并用大量表意来渲染中心思想，从而引起情感共鸣。

散文: 用词浅显直白，行文多有章法，表意使用较少，相比于诗词，易于理解。

程序: 电脑程序的编写不允许有任何比喻象征等手法，必须遵照句法结构，不可逾矩一丝。

规范语言相比自然语言，更加细致繁密，所以需要花更多时间阅读理解。其次，句法结构也很重要，所以不要总是从上往下，从左往右地阅读，而要在大脑中分析代码，识别特有词汇，解读句法规则。最后，要格外关注细节，单词错误，标点缺失，这些问题在自然语言的使用中可能影响不大，但在规范语言中，却是致命错误。

1.7 调试

是人就会犯错。由于一些历史原因，编程中的异常或者错误，叫做 **bugs(臭虫)**，这里用到了比喻，一般解释为 **bugs(异常)**。而跟踪异常，定位问题的过程叫做 **debugging(调试)**。

编程，尤其是调试，往往会刺激你。如果你困扰于一个难题，那往往会经历愤怒，沮丧和绝望。

已有证据表明，人们对待计算机的方式，和对待他人的方式一样。计算机运行正常，我们就当其为伙伴，助手，如果运行异常了，就待其为无礼之人。(Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

对于计算机要有一定心理准备，把它当作一个算术快速而精确，但是缺乏同理心以及大局观的雇员。

你需要做好管理之责：扬机器之长而避其短，做情绪的主人，而非仆人。

调试过程是令人无比沮丧的，但这又是编程路上最有用的技能。每章末尾都会有关于调试的一些建议，希望有所帮助。

1.8 术语表

解决问题 (problem solving): 提出问题，分析问题，解决问题的过程。

高阶语言 (high-level language): 类似 Python 这种便于人类读写，对人友好的编程语言。

底层语言 (low-level language): 便于计算机运行的语言，也叫“机器语言”或者“汇编语言”。

可移植 (portability): 一个程序可运行于多种设备的特性。

解释器 (interpret): 读取并执行代码的程序。

提示符 (prompt): 解释器声明准备接收输入信息的提示。

程序 (program): 运算指令集。

打印语句 (print statement): Python 中输出到屏幕的指令。

运算符 (operator): 类似加减乘除的符号。

值 (value): 程序操作的基本元素，例如数字或者字符串。

类型 (type): 值的集合。目前遇到的有整数集合 (type `int`)，浮点数集合 (type `float`) 以及字符串集合 (type `str`).

整型 (integer): 整数类型。

浮点型 (floating-point): 小数表示的数字类型。

字符串 (string): 字符序列类型

自然语言 (natural language): 人类沟通用的语言。

规范语言 (formal language): 人类因特定目标而创造的语言。例如数学语言，编程语言; 所有的编程语言都是规范语言。

词汇 (token): 程序中的基本元素，类似自然语言中的单词。

句法 (syntax): 程序的结构规范。

解析 (parse): 分析句法结构并检验程序。

异常 (bug): 程序中的错误。

调试 (debugging): 定位异常并解决的过程。

1.9 习题

Exercise 1.1. 工欲善其事，必先利其器，搭配电脑实操来学习本书，效果更好。

学习新知识更有效的方式是不断试错，不断思考。比如*"Hello, World"* 的程序，试一试双引号只写一半是否可以？不写双引号，会怎么样？以及 `print` 拼错了，又会如何？

这种学习过程中的大胆假设，小心求证，助益颇多。同时能够快速了解编程中的各种错误信息。所以，当下有准备地试错，远胜过将来意外犯错。

1. *print* 使用时，如果丢掉了半个括号，甚至全部括号，会怎样？
2. 当输出字符串时，遗落了半个双引号，或者全部双引号，会如何？
3. 我们可以用负号 表示负数 `-2`。如果像 `2+-2` 在数字前又加上一个加号呢？
4. 数学表示中，首位补零很正常，比如 `09`。但是在 *Python* 世界中，如果输入 `011` 会出来什么结果呢？
5. 如果两个值之间没有运算符会如何？

Exercise 1.2. 启动 *Python* 解释器，进行以下运算：

1. *42* 分 *42* 秒总共有多少秒？
2. *10* 千米是多少英里？提示：*1* 英里 *=1.61* 千米。
3. 如果用 *42* 分 *42* 秒跑了 *10* 千米，那平均速度是多少（每英里的耗时）？平均速度又是多少（每小时英里数）？

第 2 章　变量、表达式和语句

编程语言的强大和高效，其中一方面便体现在操作**变量**。变量名是指向值的一个名字。

2.1　变量赋值

创建新的变量，并给其一个值，叫做**赋值**:

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

上面的例子提供了三个赋值语句。第一个是将一句话赋值给了名为 `message` 的变量；第二个是将整数 `17` 赋值给了 `n`；第三个是将 π 的近似值赋给了 `pi`。

形象表示变量的一个常用手段是，写好变量，然后画个箭头执行对应的值。这种形象展示变量所处的状态 (所对应的值) 的图叫做**状态图**, 图 2.1 便是上面示例的状态图。

2.2　变量名

程序工程师一般会为变量起个容易理解记忆的名字，使人一看即知用途。

变量名可长可短，包含字母和数字，但不能以数字开头，大小写均可用，但是方便起见，建议只用小写字母。

变量名也可以使用`_`，多用在形如`your_name` 或 `airspeed_of_unladen_swallow` 这种多单词命名的变量名中。

如果变量名包含了非法字符，就会句法错误:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
```

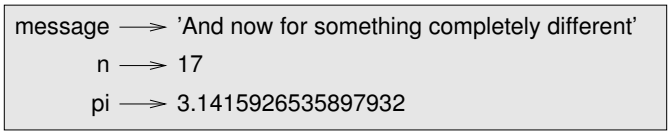

图 2.1: State diagram.

```
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

变量名 `76trombones` 错误在于数字开头，`more@` 包含了不合法字符 `@`。那么 `class` 为什么也不正确呢?

这是因为 `class` 是 Python 的一个**关键字**。解释器用关键字 (预留字) 来识别解析代码，所以关键字不能作为变量名使用。

Python 3 包含以下关键字:

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

你无须刻意背诵这些关键字。在多数开发环境中，关键字会以特殊颜色标识，所以如果误用作变量名时，很容易发觉。

2.3 表达式和语句

表达式 是值，变量以及运算符的集合。值本身可以认为是表达式，同样，变量本身也是。所以下面都可以认为是合法的表达式:

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

当根据提示符输入表达式时，解释器会进行**估算**，获取表达式的值。上例中，`n` 的值是 17，`n+25` 的值就是 42。

语句 是代码的基本有效单位，例如新建变量或者输出值。

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

第一行是给 `n` 赋值的赋值语句，第二行是显示 `n` 结果的输出语句。

键入语句，解释器根据语句规则**运行**，通常，语句没有值 (**注意: 表达式有值**)。

2.4 脚本模式

截至当前，我们一直使用**交互模式**运行代码，此模式需要不断和解释器交互。入门学习采用交互模式尚可行，但代码行数一旦增多，使用就会很麻烦。

替代方案是将代码保存至**脚本**文件，然后解释器在**脚本模式** 运行文件。通俗来说，Python 脚本以`.py` 结尾。

了解了如何创建以及运行脚本，便可开启后续学习。如果没有配套设施，建议继续采用 PythonAnywhere。同时，我已在`http://tinyurl.com/thinkpython2e` 撰写了运行脚本模式的相关步骤。

Python 提供两种模式，在写入脚本前，可以采用交互模式进行部分语句的验证。但是两种模式多有差异，会令人无所适从。

例如，以 Python 进行运算，可以键入：

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

首行代码给 `miles` 赋值，但是不显示结果。第二行是表达式，解释器会解析执行，并显示结果，约为 42 公里。

但是在脚本模式同样执行，不会获得输出结果。表达式在脚本模式不会输出结果，如果需要显示，可以采用 `print` 语句：

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

初次使用，会感到奇怪。为了更深入了解，可以在 Python 解释器键入以下内容，看看现象：

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

将同样语句放入脚本并运行，会如何？如果将每个表达式都放入 print 语句中，再次运行，又会如何？

2.5 运算次序

当表达式含有多个运算符时，执行顺序主要基于**运算次序**。对于数学运算符，Python 也遵循数学惯例。本书采用缩写 **PEMDAS** 来辅助记忆以下规则：

- 括号 (**P**arentheses) 优先，可以据此调整运算次序。因为括号中的表达式会优先加工，所以 `2 * (3-1)` 为 4，`(1+1)**(5-2)` 为 8。同时可以采用括号提高表达式的可读性。比如，`(minute * 100) / 60` 加上括号，结果虽然没有变化，但是可读性提高不少。

- 幂运算 (**E**xponentiation) 相对优先级更高。所以 `1 + 2**3` 等于 9 而非 27，`2 * 3**2` 等于 18 而非 36。

- 乘法 (**M**ultiplication) 和除法 (**D**ivision) 优先于加法 (**A**ddition) 和减法 (**S**ubtraction)。所以 `2*3-1` 等于 5 而非 4，`6+4/2` 等于 8 而非 5。

- 对于同优先级的运算，遵照自左向右的顺序 (除幂运算)。在 `degrees / 2 * pi` 中，先执行除法运算，获得的结果再乘以 `pi`。如果想除以 2π，可以使用括号处理或者改写为 `degrees / 2 / pi`。

不必强求记忆这些运算符的优先级，碰到比较复杂的表达式，采用括号会方便很多。

2.6 字符串操作

通常，字符串甚至类似数字的字符串的运算，一般不能使用数学运算符，下面的表达式均是非法运算：

```
'chinese'-'food'    'eggs'/'easy'    'third'*'a charm'
```

但是有两个例外，+ 和 *。

+ 可以实现**字符串拼接**，也就是首尾相接。例如：

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

* 同样可以作用于字符串，表示堆叠。例如'Spam'*3 表示 'SpamSpamSpam'。如果一个值是字符串，另外一个值就需要是整数。

+ 和 * 在字符串上的使用类似数值运算中的作用。4*3 即是 4+4+4，而 'Spam'*3 也等于 'Spam'+'Spam'+'Spam'。但是，数字的加法和乘法与字符串的拼接和堆叠在某方面的应用有明显不同。你是否可以想到，加法运算正常，而字符串拼接却意义大不一样的场景？

2.7 注释

随着代码量越来越大，越来越复杂，阅读理解也越来越困难。规范语言使用时又是密密麻麻，对着满页代码而明白其逻辑和作用，便异常艰难。

所以，在程序代码中加入一些自然语言来进行解释说明其作用，便显得异常重要。这些解释说明，便叫做**注释**，一般以井号 (#) 开头：

```
# 计算小时数的百分比
percentage = (minute * 100) / 60
```

此例中，注释独占一行。你也可以把注释写在代码末尾：

```
percentage = (minute * 100) / 60     # 小时数百分比
```

代码中所有 # 开头的行，在执行过程中都会被忽略。

编码中实现特殊逻辑时，注释显得尤为重要。要记住，说明编码的*原因* ，远胜于讲述代码的*作用*。

下面这条注释便显得多余:

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

而下面的注释则包含了未体现在代码中的有用信息：

```
v = 5     # 速度(米/秒)
```

好的变量命名可以减少注释的使用，但是变量名过长，又显得代码繁杂而难以阅读，所以如何取舍，需要权衡。

2.8 调试

一般程序中会出现三种异常：句法错误，运行时错误以及语义错误。有效区分三者差异，可以大大提高定位速度。

句法错误: “句法” 是指代码结构以及使用规范。例如，括号必须成对出现，`(1 + 2)` 正确，而 `8)` 就存在句法错误。

如果程序存在句法错误，Python 会直接报错并退出执行，程序便无法运行。开始学习编程时，需要耗费大量精力来跟踪句法错误。随着经验增长，此类错误会越来越少，而定位会越来越快。

运行时错误: 运行时错误只有在程序开始运行后才会出现。这些错误也叫作**异常**，因为一般表示异常或者坏的事情发生了。

本书前面几章的程序较为简单，运行时错误也较少遇到，所以想涉足其中，还需一段时间。

语义错误: 第三种错误是" 语义错误"，意如其字。如果代码中存在语义错误，代码不会报错，只是无法得到预期的结果。具体来说，便是会按照你的指示去做。

语义错误的识别很棘手，需要观察输出，回溯代码，才能确定其逻辑。

2.9 术语表

变量 (variable): 标识值的名称。

赋值 (assignment): 给变量赋值的语句。

状态图 (state diagram): 一堆变量和值的指向图。

关键字 (keyword): 编程语言用来解析代码的预留字，像 `if`,`def` 和 `while`，均不能作为变量名使用。

操作数 (operand): 运算符操作的单一值对象。

表达式 (expression): 变量，运算符以及值的集合，共同来表示单一结果。

求值 (evaluate): 通过运行表达式来产生值，从而简化表达式的过程。

语句 (statement): 表示命令或者执行的代码块。目前遇到的语句包括赋值语句以及打印语句。

执行 (execute): 运行代码语句。

交互模式 (interactive mode): 在 Python 解释器中根据提示符进行代码输入的模式。

脚本模式 (script mode): 用 Python 解释器读取脚本并运行代码的模式。

脚本 (script): 存储代码的文件。

运算次序 (order of operations): 包含多运算符和操作数的表达式执行时所遵循的先后顺序。

拼接 (concatenate): 操作数首位相接。

注释 (comment): 帮助他人理解代码的有效信息，对程序执行无影响。

句法错误 (syntax error): 导致代码无法解析 (无法解释) 的错误。

异常 (exception): 程序运行时遇到的错误。

语义 (semantics): 代码含义。

语义错误 (semantic error): 未获得预期结果的错误。

2.10 习题

Exercise 2.1. 重申从上节就给出的建议，学习新知识，尽量在交互模式下不断试错，大胆假设，小心求证。

- 既然 `n = 42` 正确，那么 `42 = n` 呢？
- `x = y = 1` 正确与否？
- 在一些编程语言中，每个语句都会以分号；结尾，如果在 *Python* 的语句末尾加上分号会怎样？
- 语句末尾加句号会如何？
- 数学公式中，可以用 xy 表示 x 和 y 相乘。*Python* 中如果如此使用会怎样？

Exercise 2.2. 用 *Pyhton* 解释器进行运算：

1. 半径 r 的球体体积是 $\frac{4}{3}\pi r^3$。那半径为 *5* 的球体体积是多少？
2. 假设书原价 *$24.95*，但书店得到了 *40%* 的折扣。同时邮寄首件需要邮费 *$3*，后续每件 *75* 美分，如果批发 *60* 件，总费用多少？
3. 如果我 *6:52* 离家，以慢跑方式 *(8:15/*每英里*)* 跑了 *1* 英里，然后以快跑方式 *(7:12/*每英里*)* 跑了 *3* 英里，最后，又以慢跑方式跑了 *1* 英里，那什么时候到家？

第 3 章　函数

在编程领域，**函数** 是指执行既定运算的语句组合。定义函数便是确定名称以及语句组合。然后，便可以通过函数名来" 调用" 函数。

3.1 Function calls

上文已展示过**函数调用**

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

此函数的名称为 type，括号中的表达式叫做函数的**参数**。此处获得的结果，表示参数的类型。

通俗来说，函数的作用便是根据“输入”的参数，“返回”相应的结果。这个结果也叫做**返回值**。

Python 提供了值类型转换的函数，正常情况下，int 函数会将任意值转为整型，除非输入有误:

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

int 虽然可以将浮点数转为整数，但是不会四舍五入，而是直接舍去小数部分:

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

float 会将整数以及字符串格式小数转为浮点数:

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

最后，str 会将任意输入值转为字符串:

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

3.2 数学函数

Pyhton 内置了数学模块，可以提供大部分的数学计算函数。**模块**是指包含一类函数的文件。

在使用模块中的函数前，我们需要用 **import 语句** 来导入模块:

```
>>> import math
```

导入语句会创建一个名为 `math` 的**模块对象**。输入模块对象名称，可以看到一些相关信息:

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

模块对象包含所有模块中的函数和变量。想调用相应函数，需要调用模块名和函数名，中间用点 (也可以当作英文中的句号) 来分隔。这种调用方法，就是**点标法**。

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

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

第一个例子使用`math.log10` 计算分贝信噪比 (已知 `signal_power` 和 `noise_power`)。math 模块也提供了 `log` 函数，此函数是以 `e` 为底。

第二个例子是计算弧度 (`radians`) 的正弦值。弧度 一般作为三角函数 (`sin`，`cos` 和 `tan`，等) 的输入参数。而角度转弧度，需要除以 180 然后乘以 π:

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

`math.pi` 表达式是从 math 模块调用 `pi`，这是一个 π 的浮点格式近似值，精确到小数点后 15 位。

如果熟悉三角函数原理，便可以通过比较 $\frac{\sqrt{2}}{2}$ 来验证结果：

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

3.3 组合

目前，我们学习了编程涉及的各个元素---变量，表达式以及语句。但都是分开来讲，并未整合在一起。

编程语言的强大之处在于，可以构建众多的小积木，然后**组合**起来。例如，函数的参数既然可以是各种表达式，那就包括了算数运算:

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

也可以是函数调用:

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

任何可以使用值的地方，都可以使用表达式来替换，除了一种情况：赋值语句的左侧必须是变量名。左侧出现表达式会报句法错误 (相关信息见后续章节)。

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

3.4 创建函数

我们一直在 Python 编程中使用函数，现在可以尝试创建一个函数。**函数定义** 一般指定义函数名以及调用时使用的语句组。

Here is an example:

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

def 关键字表示以下是函数定义。函数名是 print_lyrics。函数名的命名规则和变量名命名规则一致：可包含字母，数字或下划线，首字母不能为数字。同时不能使用关键字作为函数名，要避免变量和函数重名。

函数名后的空括号表示没有参数。

函数定义首行一般叫做**函数头**，其他部分叫**函数体**. 函数头以冒号结尾，函数体必须缩进。使用惯例来说，一个缩进等于四个空格。消息体可以包含任意行语句。

print 语句中的字符串需要被双引号包着。一般单引号和双银行作用相同，所以人们多数情况使用单引号，除非字符串中本来就有单引号，这种情况就要使用双引号了。

所有的引号（无论单双）都是 ‘‘直引号’’，就是键盘确认键 (Enter) 旁边的那个，本句中这种 ‘‘弯引号’’，在 Python 使用中就是非法的。

在交互模式定义函数，解释器会输出省略号 (...) 来提示函数定义尚未结束:

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

想结束函数定义，需要多敲一个换行。

函数定义完成便会创建一个 function 类型的**函数对象**:

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

调用自定义函数和调用内置函数的方式一样:

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

一旦函数定义完成，便可在另外函数中使用。比如，为了避免重复，可以直接定义一个repeat_lyrics 函数:

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

然后调用 repeat_lyrics:

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

当然，歌词没有这样的了。

3.5 定义和调用

将上述代码片段整理为一个完整程序:

```
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()
```

此程序包含两个函数：print_lyrics 和 repeat_lyrics。这些函数定义的执行和其他语句一样，但是会在执行后创建函数对象。函数内的语句只有在函数调用时才会执行，同时函数定义的执行并不会输出结果。

如你所料，函数运行前必须先定义函数，也就是说，函数定义语句先执行，函数才能被调用。

练习一下，将程序的末行提到开始，令函数先被调用，再被定义。运行程序，看看会报什么错误。

将函数调用再次移回末尾，然后将print_lyrics 函数定义放到repeat_lyrics 函数的后面，看看程序运行时，会发生什么?

3.6 运行流程

为保证函数定义在函数调用之前，需要了解语句执行顺序，也叫做**运行流程**。

程序运行从第一个代码语句开始，从上到下，运行一次。

函数定义的运行不会改变代码运行流程，但是要谨记，函数内的语句只会在函数被调用时运行。

函数调用就像代码运行流程中的绕路一样，不会继续向下执行语句，而是跳转到函数体内，运行内部语句，完成后再回到离开的地方。

听起来很简单，但是要意识到，函数可以不断调用函数 (*译者注：类似套娃*). 甲函数运行中，可能会调用乙函数，从而去执行其语句，然后乙函数运行中又可能去调用其他函数!

幸亏 Python 解释器能够有效追踪函数的执行流程，从而可以在函数执行完成后，回到函数调用的地方，从而继续执行，直到结束。

所以，不要简单地从上到下阅读程序，有时候更要关注代码运行流程。

3.7 形参和实参

我们观察到有些函数需要参数。例如调用 math.sin 函数，需要传递个数值作为参数。还有些函数不止一个参数：math.pow 需要两个参数，分别是底和幂。

函数内，实参会被赋值给**形参**。下面是函数传参（实参赋值给形参）的定义：

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

此函数会将实参赋值给形参 bruce，当此函数调用时，会输出两次参数值（无论参数为何）。

此函数适用于任何可被打印的值。

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

自定义函数和内置函数的规则适用是一致的，所以print_twice 的实参也可以是任意表达式:

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

实参在函数调用前运行，所以在上例中，表达式 'Spam '*4 和 math.cos(math.pi) 只会运行一次。

也可以使用变量作为实参:

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

作为实参传递给函数的变量命名 (michael) 和函数的形参命名 (bruce) 毫无关联。无论外部传递进来的参数命名为何，函数 print_twice 内部，我们统称为 bruce。

3.8 局部变量和参数

函数内部定义的变量都是**局部**的，也就是说，只能作用于函数内部，例如:

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

此函数将两个参数拼接后，结果输出两次。以下是用例:

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

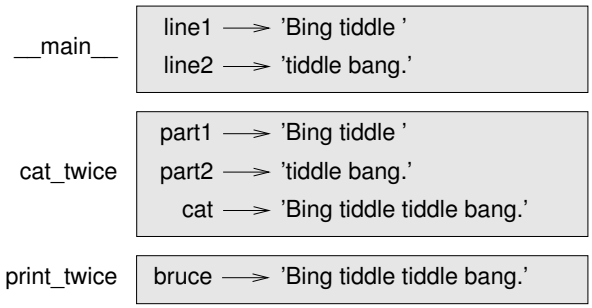

图 3.1: 栈图.

cat_twice 运行结束，内部变量 cat 便被销毁。如果现在尝试输出它，会报错:

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

形参都是局部使用的，例如 print_twice 函数外部不存在 bruce 变量。

3.9 栈图

若想跟踪变量使用情况，一般绘制**栈图**比较有效。和状态图类似，栈图也会标识每个变量的值，不同之处是，栈图也会标识变量所属函数。

每个函数都用一个**框** 来表示，一个框就是一个旁边有函数名，内部是参数以及变量值的箱体图。上面例子的箱体图见图 3.1。

栈图中的这些框的排列方式标示了各个函数的调用关系。此例中，print_twice 函数被 cat_twice 调用，cat_twice 被__main__ 调用。__main__ 是顶层框的一个特殊名字，任何函数外的变量，都属于它。

形参和实参一一对应，所以 part1 和 line1 的值相同，part2 和 line2 的值也相同，并且，bruce 和 cat 的值也一样。

如果函数调用过程中出错，Python 会输出此函数名称，以及调用它的函数的信息，以及更上级的函数，一直到 __main__ 函数。

例如，如果在print_twice 中调用 cat，会得到名字异常 (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
```

这一系列函数跟踪信息叫做**追溯**，这些信息显示是哪个文件出错，哪行出错，以及运行时函数，甚至引起错误的代码行号。

在追溯信息中的函数次序和栈图中的框的次序是一致的，当前运行的函数都是在最下面。

3.10 有值函数和无值函数

目前用到过的一些函数，例如数学函数，都会返回结果，一得之见，暂称**有值函数**. 其他函数，像 print_twice，运行完成，并不会返回值，便叫做**无值函数**。

当调用有值函数时，一般会期待使用返回结果的，比如，赋值结果到某个变量，或者置于表达式中使用:

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

在交互模式调用函数，Python 解释器会输出结果:

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

但是在脚本模式，调用有值函数，却永远不会看到返回值!

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

这个脚本文件主要计算 5 的平方根，但是没有什么用处，因为没有保存或者显示结果。

无值函数也许可以在屏幕显示信息或者执行其他操作，但是没有返回值。如果将函数运行结果赋值到变量，那变量会指向一个特殊的值：None.

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

None 是个具有既定类型的特殊值，和字符串 'None' 是不同的:

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

目前我们定义的函数都是无返回值的，后续章节将开始定义有返回值的函数。

3.11 函数何用?

对于将程序划分为一个个函数是否值得的考虑，参看以下缘由:

- 函数可以将一组语句归为一类，便于阅读和调试。
- 函数可以消除重复，简化代码，针对功能，修改一处即可。
- 将程序切分为程序，可以分而调试，然后组合它们即可。
- 设计良好的函数可以书写一次，调试一次，多处调用，有效复用。

3.12 调试

编程最重要的技能之一就是调试了，虽往往令人黯然神伤，愁眉难舒，却是编程中最考验才智，最富有挑战以及最有趣的地方.

在某些方面，调试和侦探工作很像。面对线索，有效推理，确定导致结果的流程和缘由。

调试又像实验科学，针对问题，一旦有所灵光，就要修改程序，再次尝试。如果假设正确，便可确定修改引起的变化，进一步推进程序正常运行。如果假设错误，便要继续探究了。正如 Sherlock Holmes(夏洛克 · 福尔摩斯) 所说，“除去所有不可能因素，留下来的东西，无论多么不愿意相信，但这就是事实的真相."(A. Conan Doyle, *The Sign of Four*)

一般而言，编程便是调试。这是因为，编程便是不断调试，达到目标的过程。所以，建议先写运行代码，然后逐步修改调试，实现预定效果。

比如，Linux 操作系统具有数百万行代码，但它起源于 Linus Torvalds 研究 Intel 80386 芯片的一段简单代码。据 Larry Greenfield 所说，“Linus 最早的一个程序是在输出 AAAA 和 BBBB 之间切换，这后来演化为了 Linux." (*The Linux Users' Guide* Beta Version 1).

3.13 术语表

函数 (function): 特别命名的一堆语句的组合，实现特定功能。可以有参数，也可以没有参数，可以返回值，也可以不返回。

函数定义 (function definition): 创建函数的语句，确定名称，参数以及内部语句。

函数对象 (function object): 函数定义执行后的值，函数名便是指向函数对象的变量。

函数头 (header): 函数定义首行.

函数体 (body): 函数定义内部语句.

形参 (parameter): 函数内部用于代表传入参数的名称。

函数调用 (function call): 运行函数的语句，由函数名，以及括号中的参数列表构成。

实参 (argument): 函数调用时传入函数的值。这个值会赋值给函数内部形参。

局部变量 (local variable): 函数内部变量，局部变量只能在函数内部使用。

返回值 (return value): 函数返回的结果。如果用函数调用作为表达式，那么函数返回值，便是表达式的结果。

有值返回 (fruitful function): 有返回值的函数。

无值返回 (void function: 返回 `None` 的函数。

空值 `None`: 无值返回函数返回的结果。

模块 (module): 包含相关函数以及相关定义的文件。

导入语句 (import statement): 读取模块文件并创建模块对象的语句。

模块对象 (module object): `import` 语句创建的值，通过其可以获取模块内部的值。

点标法 (dot notation): 通过在模块名后紧跟一个点 (英文句号) 和要调用的函数名，来调用模块函数的句法。

组合 (composition): 通过简单表达式构建复杂表达式，简单语句构建宏大语句的方法。

运行流程 (flow of execution): 语句运行的顺序.

栈图 (stack diagram): 表示函数，变量，以及值的关系图。

框 (frame): 栈图中表示函数调用的箱体图，包括局部变量以及参数。

追溯 (traceback): 异常发生后，输出相关运行函数的过程。

3.14 习题

Exercise 3.1. 书写名为right_justify 同时以字符串 s 为参数的函数，其输出的字符串左侧有足够的空格，以使输出结果的最后一个字符在第 *70* 字符处。

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

提示*:* 使用字符串拼接以及堆叠，同时 *Python* 提供内置 len 函数，可以返回字符串长度，例如，len('monty') 的结果为 *5.*

Exercise 3.2. 函数对象可以赋值给变量，或者作为实参传递。例如，do_twice 便将函数对象作为参数传入，并调用了两次*:*

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

下面是使用 do_twice 来调用两次 print_spam 的例子。

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


do_twice(print_spam)
```

1. 将上述代码写入脚本进行测试*.*
2. 修改 do_twice，给其传入两个参数：函数对象和值，使其调用两次传入的函数对象，这个函数对象会以传入的值为参数*.*
3. 将本章print_twice 代码复制到脚本*.*
4. 用新版 do_twice 函数调用 print_twice 两次，并传入参数 'spam'*.*
5. 定义新函数do_four，令其以函数对象和值为参数，并以传入的值为参数调用函数对象。要求函数体只有两行语句，而不是四行。

答案*: http://thinkpython2.com/code/do_four.py.*

Exercise 3.3. 说明*:* 此习题只涉及已学过的语句和功能。

1. 编写绘制下面网格图的函数*:*

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

   提示*:* 想一行输出多个值，可以向 *print* 传入多个值，这些值用逗号分隔*:*

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

一般 print 函数会附带换行符，但可以覆写此函数，使其以空格结尾，如下:

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

上边语句会将'+ -' 在同一行输出，如果再有 *print* 语句，则会令起一行。

2. 编写绘制四行四列相似网格的函数。

答案: *http://thinkpython2.com/code/grid.py*. 致谢: 此习题基于 *Oualline* 书中的习题 (Practical C Programming, Third Edition, *O'Reilly Media, 1997).*

第 4 章　案例分析: 接口设计

本章采用案例来讲述如何设计可以协同运行的函数。

通过介绍 `turtle`(乌龟) 模块，从而利用其绘图功能，制作图片。大部分本地 Python 版会预装 `turtle` 模块，如果你使用的是 PythonAnywhere 网站，现在还无法执行 turtle 示例 (至少写作本书时还不能)。

如果计算机安装了 Python，便可以直接运行这些实例。如果还没有，那么最好现在安装吧，可以参看我写的步骤`http://tinyurl.com/thinkpython2e`。

本章的代码样例可以从`http://thinkpython2.com/code/polygon.py` 获取。

4.1　turtle 模块

打开 Python 解释器，键入以下命令，以检查是否安装 `turtle` 模块:

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

运行此代码，会创建一个窗口，此窗口包含一个箭头，代表一只小乌龟。关闭窗口。

创建名为 `mypolygon.py` 的文件，并输入以下代码:

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

`turtle` 模块 (小写字母't') 调用函数 `Turtle` (大写字母'T') ，创建 Turtle 对象，并将其赋值给 `bob` 变量。输出 `bob` 信息:

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

这意味着 `bob` 了指向 `turtle` 模块中 `Turtle` 类型的对象.

`mainloop` 会令窗口等待，以执行其他操作。但是本例只需关闭窗口，暂不进行其他操作。

一旦建立 Turtle 对象，便可以调用**方法**在窗口中移动它。方法和函数类似，但是略有不同。比如，令小乌龟前进:

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

方法 `fd` 来自于乌龟对象 `bob`。调用方法就好比递交申请：请求 `bob` 向前移动。

`fd` 的参数是像素距离，实际距离要看屏幕尺寸。

其他可以调用的方法有 bk，实现向后移动，lt 实现左转，rt 实现右转。lt 和 rt 的参数都是角度。

同时，每个乌龟都带着笔，可以抬起或落下; 如果笔落下，乌龟移动时会留下痕迹。pu 和 pd 方法分别代表 “抬笔" 和 “落笔".

将下面代码加入程序，以实现绘制直角 (创建 bob 对象后，调用mainloop 前):

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

运行程序，可以看到 bob 对象会向东移动，然后向北移动，并留下两根垂直线段。

现在尝试绘制个正方形，先不要着急学习下面的内容。

4.2 简单重复

我猜你会写成下面的样子:

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

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

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

bob.fd(100)
```

用 for 语句可以更加简单地实现同样的效果。将下面代码写入 mypolygon.py 并运行: :

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

所见如下:

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

这是 for 语句的简单使用，后面会讲更多用法。但这些已经足够令我们重写绘制方块的程序了。暂时不要看下面内容，先尝试一下。

以下用 for 语句实现绘制正方形:

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

for 的句法使用类似函数定义。以冒号结尾的语句为函数头，以缩进内容为函数体。函数体可以是任意行语句。

for 语句也叫**循环**，因为代码执行流程会重复执行循环体，此例中，会执行四次循环体.

这一版代码和上一版略有不同，绘制完最后一条线后，会多一次转向。虽然多的这一步操作增加了耗时，但是利用循环，使重复操作更加简单。当然，这也令小乌龟回到了开始的地方，朝向了开始的方向。

4.3 习题集

以下为一系列操作 turtle 模块的练习。虽然是为了让大家高兴一下，但是也有别有用心的地方，做练习的时候，要好好思考一下醉翁之意。

习题后面有答案，但是尽量在完成或者尽力尝试后再看。

1. 编写 square 函数，参数是个 turtle 对象 t，用 turtle 绘制正方形。

   编写函数，将 bob 作为参数传给 square，然后运行程序。

2. 给 square 添加参数 length，令边的长度为 length，同时修改函数，令其调用第二个实参。再次运行程序，给 length 赋不同的值进行测试。

3. 复制 square 并命名为 polygon 。添加参数 n，令其绘制正 n 边形。提示：正 n 边形外角角度为 $360/n$ 度。

4. 编写 circle 函数，以 turtle 类型的 t，以及半径 r 为参数，通过调用 polygon 函数，输入适当长度和边数，绘制一个近似的圆形。用不同的 r 值，测试程序。

   提示: 确定园的周长，使其满足*边长 * 边数 = 周长*.

5. 升级 circle 版本，命名为 arc，新增参数 angle，用此值确定所绘制的圆弧的大小。angle 是角度的单位，当 angle=360，arc 会绘制一个完整的圆。

4.4 封装

第一道题是编写绘制正方形的函数代码，调用函数，传递 turtle 参数。代码见下:

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

square(bob)
```

函数内的 fd 和 lt 缩进两次，表示运行于函数定义里面的 for 循环中。下一行 square(bob)，左边距对齐同时没有缩进，表示上面的 for 循环以及函数定义都已结束。

函数内的 t 和 bob 指向的是同一个乌龟对象，因此 t.lt(90) 就好比 bob.lt(90)。那么为何不直接用变量 bob 呢? 这是因为 t 可以代指任意乌龟对象，而不仅仅是 bob，因此便可以再创建一个乌龟对象，并作为参数传入 square:

```
alice = turtle.Turtle()
square(alice)
```

将一段代码写在函数中，叫做**封装**。封装的好处之一在于用一个简单的名字来指代一段代码，好比文档说明。令一个好处是可以复用代码，复制粘贴大段代码，总是不如调用重复调用函数来的方便!

4.5 泛化

下一步是给 square 增加参数 length ，详见样例:

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

square(bob, 100)
```

给函数增加参数，叫做**泛化**。因为可以令函数使用场景更加广泛：上版代码中，调用函数，正方形变长总是同一个值，而此版，边长可以是任意值.

下一步也是泛化。不再是只绘制正方形，polygon 可以绘制任意多边形，详见示例:

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

polygon(bob, 7, 70)
```

上例绘制了边长 70 的正 7 边形。

如果你使用的是 Python 2, angle 的值可能因为整除而出现偏差。有个简单的办法，便是 angle = 360.0 / n。分子为浮点数，结果便为浮点数了。

当函数有多个数值参数时，很容易混淆各代表什么，以及参数顺序。所以比较好的办法，便是在传递实参列表时，附上形参的名字:

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

这也叫做**关键字参数**，因为是把形参的名字作为 “关键字" 包含进来。(不要同 Python 中 while 和 def 这种内置关键字混淆)。

这种句法令程序可读性更强。同时也提示了实参和形参如何作用：调用函数时，实参赋值给了形参。

4.6 接口设计

下一步要编写 circle，此函数以半径 r 为参数。以下是采用 polygon 来绘制正 50 边形的简单代码：

```
import math

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

首行根据公式 $2\pi r$ 和半径 r 计算圆的周长。要使用 math.pi，需要先导入 math。通常把 import 语句当作脚本的开头。

n 是要逼近圆的正多边形的边数，length 表示边长。最后，polygon 通过绘制正 50 边形，来逼近半径 r 的圆。

此代码的局限性在于，n 是一个常量，绘制大的圆时，不得不采用长线段。同时绘制小圆时，却要浪费时间绘制小线段。要想使此函数通用，可以将 n 作为函数的参数。这样用户 (circle 的调用者) 便更加方便使用，但是接口便显得不那么简洁了。

函数**接口** 是函数用途的简介: 参数是什么？函数做什么？返回什么？如果一个接口聚焦于特定功能，而非无关细节，那么这个接口就是 "简洁" 的。

这个案例中，r 放在接口中合适，是因为要用它来确定圆的大小。而 n 放进接口则显得不太友好，是因为它只与如何绘制圆的细节有关。

与其令接口复杂，不如根据周长来定义一个合适的 n:

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

现在边的个数便是近似 circumference/3 的整数，每个边的长度也就近似于 3 了，用这个长度，绘制大圆快速，小圆好看，对任意尺寸的圆都适用。

给 n 加 3 是为了保证多边形至少有 3 条边。

4.7 重构

我们可以复用 polygon 来构造 circle，是因为边足够多的正多边形近似于圆。但是，构造 arc 就不适合了，因为没有办法用 polygon 或者 circle 来绘制一段弧线。

一个替代策略是复制 polygon，修改为 arc，代码如下:

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

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

函数后半段和 polygon 很像，但如果我们不调整接口，便无法复用 polygon。我们需要给 polygon 加入角度作为第三参数，来令其通用，那么也不能继续用 polygon 作为名字了！相应的，叫 polyline 会更合适些:

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

现在我们可以用 polyline 来重写 polygon 和 arc 了:

```
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)
```

最后，我们再用 arc 来优化 circle：

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

此过程---调整代码，优化接口，方便复用---叫做**重构**。此案例中，我们发现 arc 和 polygon 具有一些相似代码，所以将其“抽离" 为 polyline.

如果早有规划，我们可能会优先编写 polyline，避免后续耗时重构。但一般你很难在项目开始的时候，便深入了解一切，规划好所有接口。只有开始编码，你才会更好地研究并理解个中辛酸。所以，有时候重构的过程也是你学习新知识的历程。

4.8 开发计划

开发计划便是指编码的流程。以上案例中，我们采用的流程便是“封装和泛化"，详细步骤如下:

1. 先不考虑函数定义，写段小程序.
2. 如果代码可用，抽离相关代码，封装进函数，并命名.
3. 增加合适参数，泛化函数.
4. 重复步骤 1-3, 尽量抽离各个函数，尽量复制粘贴，减少打字 (以及重复调试).
5. 尝试重构程序。比如，在多个地方有相似代码，试试能否将其分解为适当的通用函数.

这种流程存在一定的缺点---后续会讲述一些替代方案---但是在你无法提前规划，哪些代码需要抽离为函数时，这种方法可以让你在做的同时可以合理优化代码。

4.9 帮助文档

帮助文档是指函数开头部分，用来对接口 (“doc" 是 “documentation" 的简称) 进行解释说明的文档。下面是个例子:

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

一般帮助文档都要用三引号引起来，也叫多行字符串，因为三引号允许字符串跨行。

帮助文档虽然简洁，但是重要，因为它包含函数使用的相关说明。同时简要概括了函数的功能 (无需探查函数细节实现)，以及各个参数的类型和作用，以及应用场景。

设计接口，重要的一步，便是编写友好的帮助文档。一个设计优良的接口，应该是易于理解的; 如果需要耗费大量口舌讲解，那么这个接口还有待优化。

4.10 调试

接口类似于调用方和函数之间的中间人。调用方提供特定参数，函数执行特定操作。

例如，`polyline` 需要四个参数: Turtle 对象 `t`，整数 `n`，正数 `length`，以及以度为单位的数字 `angle`.

这些条件，叫做**前置条件**，因为这些条件满足了，函数才能执行。相反，函数结尾的内容叫做**后置条件**。后置条件包括函数预期效果 (比如绘制线段)，以及意外情况 (移动 Turtle 或其他影响).

前置条件是调用方的任务。如果调用方违背 (标注详细的) 前置条件，导致函数异常，那么责任在于调用方，而不是函数。

如果前置条件都满足而后置条件未达到预期，那问题便出在函数中。如果前置和后置条件都清晰明了，那么调试会方便很多。

4.11 术语表

方法 (method): 用句点于对象连接的函数。

循环 (loop): 程序中可以重复执行的部分。

封装 (encapsulation): 将一堆语句放入函数中的流程。

泛化 (generalization): 将无需特定的内容 (比如特定数字) 更换为通用内容 (变量或参数) 的过程。

关键字参数 (keyword argument): 一种实参格式，将形参名字作为 "关键字" 包含在内。

接口 (interface): 关于如何使用函数的描述，包括函数名，实参，以及返回值信息。

重构 (refactoring): 修改代码，以优化函数接口，提升代码质量的流程.

开发计划 (development plan): 编码预订安排.

帮助文档 (docstring): 函数开头部分的文档，用来描述函数接口.

前置条件 (precondition): 函数运行前，调用方需满足的条件。

后置条件 (postcondition): 函数结束前，需要满足的要求.

4.12 习题集

Exercise 4.1. 从 *http://thinkpython2.com/code/polygon.py* 下载本章代码。

1. 绘制栈图，描述 `circle(bob, radius)` 运行时的程序状态。可以手算，或者在代码中加入 `print` 语句.
2. 章节 *4.7* 中的那版 `arc` 并不是很精确，因为用来逼近圆的线段总会在圆的外侧。所以 *Turtle* 的最终位置总会和真实位置有偏差。我的方案可以有效降低误差。阅读代码，看看是否有所帮助。画个栈图，也许便明白了其原理。

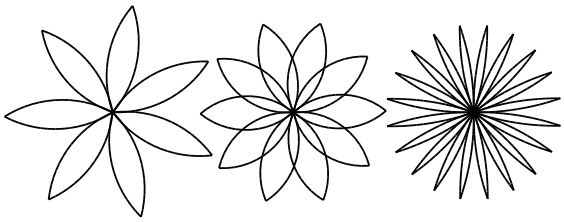

图 4.1: Turtle flowers.

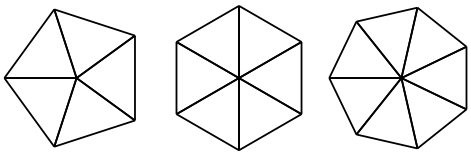

图 4.2: Turtle pies.

Exercise 4.2. 尝试编写函数，绘制图 *4.1*中的花朵。

参阅: *http://thinkpython2.com/code/flower.py*, 以及 *http://thinkpython2.com/code/polygon.py*.

Exercise 4.3. 编写函数，绘制图 *4.2*中图形。

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

Exercise 4.4. 字母表中的字母都是由几个基础元素构成，比如垂直线和水平线，以及曲线。设计个字母表，使用的基本元素种类最少，然后编写函数绘制字母。

你要为每个字母开发函数, 比如 `draw_a`, `draw_b`, 等等, 将这些函数放在名为 `letters.py` 的文件中。你可以从 *http://thinkpython2.com/code/typewriter.py* 下载个 *"turtle typewriter"*，来检验代码。

可参阅 *http://thinkpython2.com/code/letters.py*; 以及 *http://thinkpython2.com/code/polygon.py*.

Exercise 4.5. 去*http://en.wikipedia.org/wiki/Spiral* 了解一下螺旋线; 然后编码实现一个阿基米德螺线 (或其他种类)。参阅: *http://thinkpython2.com/code/spiral.py*.

第 5 章　条件和递归

本章重点是 `if` 语句，基于程序状态，从而执行不同的代码。但首先我要先介绍两个新的运算符: 向下取整和取模。

5.1　向下取整和求模

向下取整 运算符, 即 `//`, 两数相除，结果向下取整。例如，假设某电影时长 105 分钟，你想知道按小时计，是多长时间。一般除法返回的是浮点数:

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

但一般我们表示小时，不用小数格式。向下取整返回的就是整的小时数，舍弃了小数部分:

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

要想知道舍弃部分的分钟数，只需减去一小时，剩下的便是了:

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

还可以使用**求模运算符**，`%`，两数相除，返回余数。

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

求模运算的用途不止于此。例如，看一个数字是否可以被另一个整除--如果 `x % y` 结果为 0, 则 `x` 可以被 `y` 整除.

同样方法，便可以从数字提取一位或多位余数。例如，`x % 10` 获取的便是 `x` 除以 10 后剩下的余数。同样的，`x % 100` 获取的就是两位余数。

如果你使用的是 Python 2, 除法略有不同。其除法运算符，`/`，会在两个整数相除时，自动向下取整，如果有一个是浮点数，那么结果才是浮点格式。

5.2 布尔表达式

布尔表达式 是用来表示结果是对或错的表达式。下面例子使用运算符 ==，比较两个操作数，如果相等，则结果为 True, 否则为 False:

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

True 和 False 都是布尔格式的值，不是字符串:

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

== 运算符是**关系运算符**的一种，其他还有:

```
x != y               # x不等于 y
x > y                # x 大于 y
x < y                # x 小于 y
x >= y               # x 大于等于 y
x <= y               # x 小于等于 y
```

虽然这些运算符对你来说并不陌生，但是 Python 中的运算符和数学里的不同。一个很常见的错误就是混淆单等号 (=) 和双等号 (==)。要知道，= 是赋值运算符，而 == 是关系运算符。而且也没有 =< 和 => 这种等号在左侧的比较运算符。

5.3 逻辑运算符

有三种**逻辑运算符**: and, or, he not. 其语义和英文中的意思很相似。例如 x > 0 and x < 10 只有在 x 大于 0 且小于 10 时才为真.

n%2 == 0 or n%3 == 0 在一个或者全部条件都成立时为真，也就是说，被 2 整除或被 3 整除。

最后，not 运算符是个对布尔表达式取反的操作。所以，当 x > y 为假，not (x > y) 便为真，也就表示 x 小于或者等于 y 时，为真.

严格来说，逻辑运算符的操作数都应该是布尔表达式，不过 Python 在这方面不太严格。任何非零的数都当作 True:

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

这种灵活性固然有用，但是一些细微差异，会令人困惑。尽量不要如此使用 (除非你知道你在做什么).

5.4 条件执行

若想写出实用性强的程序，就必然需要程序可以根据条件，进行选择处理。**条件语句** 便可以解决此难题。最简单的便是 if 语句:

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

if 后面的布尔表达式，叫在**条件**。如果为真，则缩进的语句执行，如果为假，则不执行。

if 语句和函数定义的结构一样: 头部和紧随其后的缩进体。这样的语句统称为**复合语句**.

缩进体中对于语句行数没有限制，但是至少要有一行。但是有时候，缩进体暂时不想写语句(类似于待写代码的占位符)，这种情况，可以使用 pass 语句，表示不做任何操作.

```
if x < 0:
    pass          # TODO:需处理负数!
```

5.5 选择执行

if 语句第二种使用场景，便是 “选择执行", 一般有两种选择，根据条件判断，具体执行哪个。语义结构如下:

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

如果 x 对 2 取余后，结果为 0, 则 x 为偶数，并输出相应信息。如果不为 0, 则执行第二组语句。这种条件非真即假，必然有相应方案会执行。这些方案，叫做**分支**，因为它们属于执行流程上的分叉。

5.6 链式条件

有时候，会遇到多种可能方案，便需要更多的分支。一种方式是采用**链式条件**进行处理:

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

elif 是 “else if” 的缩写。同样，上述代码也只有一个分支会运行。对于 elif 语句的数量，没有限制。至于 else，不是必须的，但如果有的话，则必须放到结尾.

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

这些条件都是顺序检验。如果一个为假，则检验下一个，依此类推。如果条件为真，则相应分支执行，同时这些判断语句也都结束。即使存在多个条件为真，那也只会执行第一个为真的分支.

5.7 嵌套条件

条件判断也是可以嵌套于其他条件内的。可以将上一章的例子改写如下:

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

外部的条件判断包含两个分支。第一个只包含一个简单语句。第二个则包含另外一个 `if` 语句，这个语句又包含两个分支，这两个分支也都很简单，只是简单语句。同样，它们的位置也可以继续放条件语句.

虽然语句的缩进可以使代码结构清晰，但是**嵌套条件**的语句阅读起来却很麻烦。所以，最好还是尽量不用。

逻辑运算符可以有效简化嵌套条件语句，例如，可以用一行条件语句来重写下面的代码:

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

只有两个条件都满足，`print` 语句才会运行，这恰恰和 `and` 运算符的作用一样:

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

对于这种条件判断，Python 提供了更简洁的方案:

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

5.8 递归

一个函数可以调用另一个函数，那么，函数也就可以调用自身。虽然目前还未看到其用途，但是这却是程序最神奇的功能之一了。看看以下示例:

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

如果 `n` 为 0 或负数，输出单词 “Blastoff!”。否则，输出 `n` ，然后调用函数自身 `countdonw` 并以 `n-1` 为实参。

如果像下面一样，调用此函数，会如何?

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

调用 `countdown`，并且 `n=3`，由于 `n` 大于 0, 输出 3 并以 `n-1` 为参数调用自身...

> 调用 `countdown`，并且 `n=2`，由于 `n` 大于 0, 输出 2 并调用自身...

调用 countdown，并且 n=1，由于 n 大于 0, 输出 1 并调用自身...

调用 countdown，并且 n=0，由于 n 不大于 0，输出 “Blastoff!"，然会返回。

n=1 的 countdown 执行完结，返回。

n=2 的 countdown 执行完结，返回。

n=3 的 countdown 执行完结，返回。

然后会回到 __main__ 中，所有输出如下:

```
3
2
1
Blastoff!
```

函数内部调用了自身，这种函数便是**可递归的**; 其执行过程叫做**递归**.

再举个例子，我们写个使用 print 输出 n 次的函数.

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

如果 n<=0, 则 **return 语句**终止函数运行。运行流程立刻返回到调用者，函数其他代码不执行。

函数其余部分代码和 countdown 类似: 输出 s，以 $n-1$ 为参数，调用自己，并输出 s，再重复 $n-1$ 次。那么，所有输出的行数便是 1 + (n - 1)，其和为 n。

对于这种，用 for 循环更加方便。但是后续我们会遇到一些问题，用 for 循环会比较难写，而用递归却轻而易举，所以，这不啻为一个好的开始。

5.9 递归函数的栈图

在章节 3.9, 我们使用栈图来描述函数调用时程序的状态。同样，栈图也有助于我们更好理解递归函数。

一旦函数调用，Python 都会创建一个包含函数局部变量以及参数的框。所以对于递归函数，可能会同时创建多个框。

图 5.1 便是 countdown 以 n = 3 为参数，被调用时的栈图。

通常，最顶层的框属于__main__。其为空，是因为没有在__main__ 内创建变量或者传入实参。

四个 countdown 的框分别对应不同的 n 值，最底层的 n=0 的栈，叫做*边界条件*，也就是不再进行递归调用的栈，所以下面也不会再有其他框了。

做个练习，以s = 'Hello' 和 n=2 为参数，绘制print_n 调用的栈图。然后编写名为 do_n 的函数，参数为一个函数对象，和一个数值 n。使其调用传入的函数 n 次。

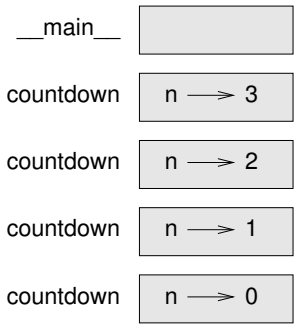

图 5.1: 栈图.

5.10 无穷递归

如果递归一直无法触及边界条件，则会一直调用，永不终止。这便叫做**无穷递归**，出现这种情况，往往意味着前路坎坷。下面是个无穷递归的小程序:

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

多数编程环境中，无穷递归的程序不会一直运行。在 Python 中，当达到了最大递归深度，便会报错:

```
  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
```

这次的追踪信息比以往的要长一些。这个错误发生时，栈中已经有 1000 个*递归*框了!

如若不幸遇到无穷递归，最好仔细检查一下函数，确保存在一个边界条件，来终止递归调用。如果已有边界条件，要确保其被触达。

5.11 键盘输入

目前涉及的程序，千篇一律，基本都没有涉及用户输入。

Python 提供了内置函数 `input`，以暂停程序运行，等待用户键入信息。当用户敲击 Return 或 Enter 键时，程序恢复运行，同时`input` 将用户输入作为字符串返回。在 Python 2 中，同样作用的函数叫做`raw_input`.

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

在接收用户输入时，最好给用户以提示，使其知晓要输入的内容。`input` 的参数便是提示信息内容:

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

提示信息末尾的\n 是个**换行符**，表示另起一行的特殊字符。所以用户的输入信息会处于提示信息下面.

如果想要一个整数，那就要将返回值转为 int:

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

但如果用户输入的不是数字类型的字符串，那便要报错了:

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

后续我们会学习如何应对这种错误。

5.12 调试

当遇到句法异常或者运行时异常，其报错信息庞大而繁杂，但可以对其进行提炼。一般最有用的也就下面两部分:

- 哪种错误
- 发生何处

句法错误通常容易识别，但是有些则略显迷惑。空格异常一般比较麻烦，因为空格和制表(Tab)符都看不到，所以容易被忽视。

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

此例中，问题在于，第二行多了一个空格。但是错误信息指向 y，便舞蹈了我们。通常，错误信息只标识了错误发生的位置，但是问题代码可能在此位置之前甚至是前行代码。

运行时错误也如此，假设以分贝为单位，计算信噪比。公式是 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$。在 Python 中，编码如下:

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

运行代码，错误如下:

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

错误信息显示，问题出在第 5 行，但是这行代码看不出任何问题。为了确定具体问题，输出 `ratio` 值，显示为 0. 那问题便出在第 4 行，浮点数除法，误用了整除.

对错误信息要仔细阅读，但也不能完全偏信。

5.13 术语表

向下取整 (floor division): 两数相除，结果向下 (负无穷方向) 取整.

求模运算符 (modulus operator): 百分号 (%) 表示的运算符，用来表示两整数相除后的余数。

布尔表达式 (boolean expression): 结果为 `True` 或 `False` 的表达式.

关系运算符 (relational operator): 比较操作数的运算符: `==`, `!=`, `>`, `<`, `>=`, 和 `<=`.

逻辑运算符 (logical operator): 拼接布尔表达式的运算符: `and`, `or`, 以及 `not`.

条件语句 (conditional statement): 根据条件，确定程序运行流程的语句.

条件 (condition): 条件语句中的布尔表达式，确定分支走向.

复合语句 (compound statement): 由头部和缩进体构成的语句，头部以冒号 (:) 结尾，缩进体紧临头部，并以缩进为标识.

分支 (branch): 条件语句中可选的一系列语句中的一种。

链式条件 (chained conditional): 一系列可选分支构成的条件语句。

嵌套条件 (nested conditional): 条件语句的分支中，又有条件语句。

返回语句 (return statement): 令函数立刻终止并跳转到调用方的语句。

递归 (recursion): 函数调用自身的过程。

边界条件 (base case): 递归函数中用来终止递归调用的条件分支。

无穷递归 (infinite recursion): 不存在或者永远无法触及边界条件的递归，最终，无穷递归会报运行时异常.

5.14 习题集

Exercise 5.1. time 模块提供同样名为 time 的函数，此函数以某个“时间点”为基准，返回当前格林威治时间戳。理论上，可以以任意时间为参考点，而在 *Unix* 系统中，一般以 *1970* 年 *1* 月 *1* 日为参考“时间点”。

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

编写脚本，实现将当前时间转换为一天中的时间 (以时分秒为格式)，以及基准时间点以来的天数。

Exercise 5.2. 费马大定理说，没有任何正整数 *a*, *b*, 和 *c* 满足

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

当 *n* 大于 *2* 时.

1. 编写函数 check_fermat ，四个入参—a, b, c 和 n— 以检验费马大定理是否成立。如果 *n* 大于 *2* 同时满足

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

   那么程序应输出 , *“Holy smokes, Fermat was wrong!”* , 否则 , 输出 *“No, that doesn't work.”*

2. 编写函数，令用户输入 a, b, c 和 n，并将其转换为整数，然后用check_fermat 来检验是否违背了费马大定理。

Exercise 5.3. 给你三根木棍，你不一定可以将其拼成三角形，比如，一根 *12* 英寸长，其余两根 *1* 英寸长，这两根太短，以至于都到不了长的那根的中间。所以，对于三根任意长度的木棍，有个简单方案，可以检验其是否可以拼成三角形:

> 三根木棍中，如果有任意一根长度大于另外两根之和，便拼不成三角形。否则，便可以拼成三角形。(如果两边之和等于第三边，便称其为“退化”三角形。)

1. 编写is_triangle 函数，以三个整数变量为入参，同时根据三个特定长度的木棍是否可以拼成三角形，来输出 *“Yes”* 或 *“No”*。

2. 编写函数，提示用户输入三个木棍的长度，并将其转换为整数，然后用is_triangle 检测这三个值是否可以拼成三角形。

Exercise 5.4. 下面的程序会输出什么？绘制栈图，表示输出结果时，程序状态.

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

recurse(3, 0)
```

1. 调用 recurse(-1, 0)，会发生什么?

2. 为此函数编写帮助文档，告知使用函数所须了解的相关信息 (仅此而已).

以下练习需要用到章节 4提到的 turtle 模块:

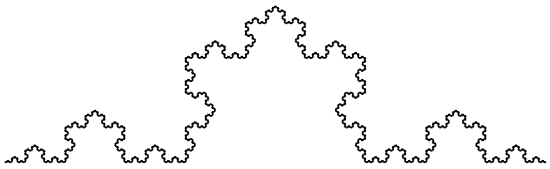

图 5.2: 科赫曲线.

Exercise 5.5. 阅读下面函数，看看是否能明白其功能 *(参阅章节 4中的案例)*。运行并看看是否正确。

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

Exercise 5.6. 科赫曲线 *(The Koch curve)* 是类似图 *5.2*的一种分形几何。要想绘制长度为 x 的曲线，下面是需要做的*:*

1. 绘制长为 $x/3$ 的科赫曲线。
2. 左转 *60* 度。
3. 继续绘制长 $x/3$ 的曲线。
4. 右转 *120* 度。
5. 再绘制长 $x/3$ 的曲线。
6. 左转 *60* 度。
7. 绘制长为 $x/3$ 的曲线。

当 x 小于 *3* 时，会有所不同*:* 在此情况下，绘制所得为长 x 的一段直线。

1. 编写 `koch` 函数，以 *turtle* 和长度 *length* 为参数，使用 *turtle*，根据指定长度，绘制科赫曲线。
2. 编写 `snowflake` 函数，使其绘制三条科赫曲线，从而构成雪花的轮廓.

   参阅*: http://thinkpython2.com/code/koch.py*.
3. 生成科赫曲线有多种方法。参看 *http://en.wikipedia.org/wiki/Koch_snowflake* 上的案例，选择你喜欢的进行实现。

第 6 章　有值返回函数

目前我们用到的很多 Python 函数都有返回值，比如 math 函数。但我们目前写的函数，都是无返回值的: 它们只是实现特定的效果，比如输出值，或者移动小乌龟，只是它们都没有返回值。本章，重点学习如何编写有值返回函数。

6.1　返回值

调用函数便会产生返回值，一般被赋值给变量或者作为表达式的一部分使用。

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

目前写的函数，多是无返回值的。笼统讲，是没有返回值，但是，更准确地说，返回值是空 (None).

本章，我们总算要写一些有返回值的函数了。第一个例子是 area, 根据给定的半径，返回面积:

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

前面我们学过了 return 语句，但是在有值返回的函数中，return 语句包含表达式。其意思为: ‘‘立即将此表达式作为返回值进行返回。" 鉴于表达式可简可繁，上述函数可以精炼为下面样子:

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

但是，使用类似 a 这样的**临时变量**，更加清晰明了，便于调试。

有时候，根据条件，设置多个不同返回语句，更加高效:

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

这些 return 语句都处于可选条件分支中，而且，只有一个会执行。

一旦返回语句执行，函数不再执行后续语句，立刻终止运行。return 语句后的代码，或者任何不被触及的代码，都叫做**无效代码**.

有值返回函数中，最好保证程序中的每种可能情况，都有 return 语句。例如:

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

这个函数错误之处在于，如果 x 恰好是 0, 便没有条件为真，也就不会触及任何 return 语句。即使运行到函数最后，返回值也会是 None，也不会是 0 的绝对值。

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

顺便提一下，Python 提供了内置函数 abs 来计算绝对值。

做个练习，编写 compare 函数，输入为 x 和 y，如果 x > y，返回 1，如果 x == y，返回 0，如果 x < y，返回-1。

6.2 增量开发

随着编写的函数越来越长，你会发现，调试时间也越来越恐怖。

若想解决越来越复杂的程序，可以试试新的方法，即**增量开发**. 增量开发是通过每次只编写并测试少量代码，不断补充完善程序，从而避免单次耗时庞大的开发调试。

比如，若想计算两坐标 (x_1, y_1) 和 (x_2, y_2) 之间的距离，通过勾股定理，可以得到距离为:

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

首先考虑 distance 函数是什么样子。换句话说，就是输入为何 (形参)，输出为何 (返回值)?

此例中，输入为两个点，即四个值表示的坐标。返回值为两点的距离，用浮点数表示。

马上便可以写出函数的大致轮廓:

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

显然，此函数无法计算距离，因为其总是返回 0. 但是，此函数语法格式又是正确的，所以可以运行，这意味着，在其变复杂之前，能够通过测试。

用样例参数调用此函数，看看效果:

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

选择这些值作为坐标，是因为其水平距离是 3, 垂直距离是 4, 两点距离便是 5. 也就是 3-4-5 这样一个直角三角形的斜边。知晓预期结果，在测试函数过程中，是及其重要的。

我们已经确认此函数语法正确，现在便可以补充完善代码了。首先，我们计算 $x_2 - x_1$ 和 $y_2 - y_1$ 的值，保存为临时变量，并输出。

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

如果函数正常，则会输出dx is 3 和 dy is 4。这样，我们便知道函数入参正确，同时前期的计算无误。如果意外发生，那么我们需要仔细检查一下这几行代码。

下一步，计算 dx 和 dy 的平方和:

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

再次运行代码，并检查输出 (输出值应为 25)。最后，使用 math.sqrt 计算最终的返回值:

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

如果程序运行正常，意味着工作顺利结束。否则，就要输出返回语句前的 result 的值，仔细排查。

最终版本的函数，除了返回一个值，不会输出任何信息。print 语句只是为了用来调试代码，一旦确认程序无误，就应将其移除。类似这种代码，一般叫做**脚手架代码**，主要用来辅助构建程序，而并不应忝列最终成品之中。

开始时，一般每次只增加一两行代码。但随着经验增长，慢慢你便可以驾驭大段代码了。无论如何，增量开发都能节约大量调试时间.

此流程的主要步骤如下:

1. 首先写个可运行的程序，然后逐步增加。任何时候遇到错误，都要弄清缘由，尽快解决。
2. 利用变量表示中间状态值，从而更方便展示和检验。
3. 一旦程序运行正常，便可以移除冗余代码，并精简繁琐的语句。但是要警惕不要使代码难以阅读和理解。

做个练习，用增量开发的方式，写个函数 hypotenuse，给定直角三角形的两个直角边，返回斜边长度。开发时，记录开发流程的各个阶段。

6.3 组合

如你所想，函数可以调用函数。例如，编写函数，输入两个坐标，一个是圆心，一个是圆周上的点，计算圆的面积。

假设圆心坐标为变量 xc 和 yc，圆周上点的坐标为 xp 和 yp. 首先要计算圆的半径，也就是两点之间的距离。便可以借用之前写过的 distance 的函数，如下:

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

下一步便是根据半径，计算面积，借用之前函数:

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

将上述步骤封装为一个函数，得到如下:

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

临时变量 `radius` 和 `result` 在开发和调试时有用，但是程序一旦正常运行，便可以通过组合函数调用，精简如下:

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

6.4 布尔函数

函数可以返回布尔值，从而便于隐藏函数内复杂的测试逻辑。例如:

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

通常给布尔函数命名，用需要是/否的提问语句; `is_divisible` 判断 `x` 是否可以被 `y` 整除，从而返回 `True` 或 `False`。比如:

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

既然 `==` 运算符的结果为布尔值，我们便可以直接返回，从而精简函数:

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

布尔函数一般用于条件语句:

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

也可以这么写:

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

但这个比较就显得多余了。

做个练习，编写函数 `is_between(x, y, z)`，如果 $x \leq y \leq z$，返回 `True`，否则返回 `False`.

6.5 More recursion

目前，我们只学到了 Python 的一小部分，但是麻雀虽小，五脏俱全，这一小部分便足以表示一门完整的编程语言了，也就是说，如果一切皆是计算，那么以上所学便已足够。任何程序都可以只用以上所学模块，重新塑造 (当然，可能还需要一些控制设备的命令，比如管理鼠标，硬盘等，但是也只额外需要这些).

最早证明了上述伟大结论的是艾兰图灵 (Alan Turing)，最早的计算机科学家之一 (有人会计较其是数学家，但是早期的计算机科学家，基本都是数学家)。因此，这个理论也叫做图灵理论断 (Turing Thesis)。关于图灵论断，如果想更加详细深入了解，建议阅读 Michael Sipser 的书，*计算理论导引 (Introduction to the Theory of Computation)*.

为了展示目前所学的威力，我们分析几个递归数学函数。递归定义和循环定义类似，就是函数定义体内包含了对定义体的引用。通常一个完全循环的定义，是无用的:

砍: 便是砍的形容词.

如果在辞典中看到这样的定义，一定郁闷。如果查询一下用符号 ! 表示的阶乘运算，会见到以下内容:

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

表示 0 的阶乘为 1, 同时对于任意 n 值的阶乘，便是 n 乘以 $n-1$ 的阶乘。

所以 3! 表示 3 乘以 2!，而 2! 表示 2 乘以 1!，1! 又是 1 乘以 0!。整理一下，3! 等于 3 乘以 2 乘以 1 再乘以 1, 也就是 6.

若要将上述过程用递归表示，那便可以写个 Python 程序实现。首先要确定参数。很显然，此处 `factorial` 函数的参数为整数:

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

如果参数恰好为 0, 只需要返回 1:

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

而其他情况就有意思了，需要递归调用 $n-1$ 的阶乘，然后与 n 相乘:

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

此程序的运行流程和章节 5.8中的 `countdown` 极为相似。如果以 3 为参数调用 `factorial`:

3 不等于 0, 则走第二分支，计算 `n-1` 的阶乘...

2 不等于 0, 则走第二分支，计算 `n-1` 的阶乘...

1 不等于 0, 则走第二分支，计算 `n-1` 的阶乘...

0 等于 0，则走第一分支，不再递归调用，返回 1.

返回值 1 乘以 n，而 n 此时为 1, 则返回 1。

返回值 1 乘以 n，此时 n 为 2, 返回为 2 的结果.

返回值 (2) 乘以此时为 3 的 n，结果为 6，也就是整个流程最终的结果。

图 6.1 表示此函数调用顺序的栈图.

返回值会在栈中被传递。每个框内，返回值就是 `result` 的值，也就是 `recurse` 和 `n` 相乘的结果。

最后的框中，没有局部变量 `recurse` 和 `result` ，是因为没有走第二分支。

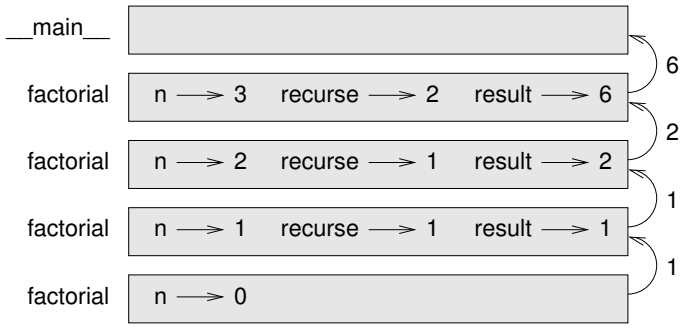

图 6.1: 栈图.

6.6 置信迁移

阅读程序的一种方式是跟踪其执行顺序，但是很快很快便不堪重负。另一种可行方案是“置信迁移"。当遇到某个函数调用时，不是根据执行流程深入跟踪，而是假设这个函数工作正常，可以返回正确结果。

实际你在使用内置函数时，就在实践置信迁移了。调用 `math.cos` 或 `math.exp` 时，并没有深入检查其函数执行体。这是因为你相信写出这些内置函数的人是优秀的编程人员，所以相信他们写的函数也是正确的函数。

对于你来说，调用自己的函数也是同理。例如章节 6.4中，我们编写了`is_divisible` 函数，其实现判断一个数是否可以被另一个数整除的功能。如果我们相信自己写的函数是正确的--分析代码并测试都通过– 我们便可以直接使用它，而无需再探查细节。

递归程序也一样。当递归调用时，无需一步步跟踪运行流程，你应该相信递归调用正常 (会返回正确的结果)，然后问问自己，“既然可以计算出 $n-1$ 的阶乘，那么是不是也可以计算出 n 的阶乘？" 很显然，乘以 n 即可。

当然，没有完成函数编写，便假设其正常运行，是有点奇怪，所以这也是我们称其为置信迁移的原因!

6.7 另例

对阶乘熟悉后，我们便来进阶更典型递归函数，斐波拉契数列。其详细定义可参阅`http://en.wikipedia.org/wiki/Fibonacci_number`:

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

翻译为 Python 代码，样子如下:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
```

```
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

若你尝试跟踪其流转，那即使一个很小的 n 值，都能令你头疼。但是，基于置信迁移，如果两次递归调用都正常，那么很显然，一直执行到最后，也会是正常的。

6.8 类型检查

试试给 `factorial` 函数传递个 1.5 的参数，会发生什么?

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

看起来陷入了无穷递归。怎么会这样? 函数有边界条件啊--当 `n == 0` 时。但如果 `n` 不是整数，那便会*无法触及*边界条件，一直递归下去。

第一次递归调用中， `n` 是 0.5。下一次，变成了-0.5. 再然后，会越来越小 (更小的负数)，也就永远不可能再成为 0.

我们有两个方案。一个是尝试改进 `factorial` 函数，使其支持浮点数，或者使其检验参数类型。第一个方案会写出伽玛函数，超出了本书的范畴。所以选择方案二。

可以使用内置 `isinstance` 函数来检验参数类型。同时，我们也需要保证参数是正数:

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

第一个边界条件，针对非整数; 第二个，则针对负整数。这两个边界条件中，都会输出错误信息，并返回 `None`，以标识运行错误:

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

如果通过两个检验，那可以确定 n 现在是非负整数，至此便可以确保递归会终止了。

此程序展示了一种叫做**哨兵**的角色。前两个条件，就像哨兵一样，避免程序犯错，从而正确运行。

在章节 11.4 中，我们会看到一种更灵活的方案来输出错误信息: 上报异常。

6.9 调试

将程序大而化小，也就天然地为调试创造了一个个的检查点。如果程序运行异常，需要考虑以下三种可能原因:

- 函数入参异常，前置条件未满足。
- 函数本身异常，后置条件不满足。
- 返回值或者调用方式异常。

若要避免第一种异常，可以在函数开始用 `print` 语句，打印参数值 (以及类型)。或者编写代码，校验前置条件。

如果参数没问题，那在每个 `return` 语句前，增加 `print` 语句，打印返回值。如果可以的话，最好亲自检查结果。同时尽量在调用函数时，传入合适的参数，以使返回结果便于校验 (如章节 6.2)。

如果函数都正常，那检验一下函数调用方式，看看是否正确使用了返回值 (或者至少用到了返回值！)

在函数开头添加打印语句，有助于更直观地观察运行流程。例如，下面是包括打印语句的 `factorial` 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
```

在此，用空格字符串来标识输出语句的缩进。以下为 `factorial(4)` 的运行结果:

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

若惑于其函数执行流程，那么，此输出相比有助于理解。有时候，增加脚手架需要耗费一点时间，但是，这一点时间的花费，往往能够节约大量的调试时间。

6.10 术语表

临时变量 (temporary variable): 复杂运算中，暂存中间值的变量。

无效代码 (dead code): 程序永远不会运行的语句，通常在 `return` 语句之后。

增量开发 (incremental development): 一次只开发并测试少量代码的开发方案，小步慢跑，从而防止费时调试。

脚手架 (scaffolding): 程序开发中起辅助作用的代码，最终程序中一般会被移除。

哨兵 (guardian): 一种编程模式，使用条件语句进行检验，并处理可能异常。

6.11 习题集

Exercise 6.1. 针对以下代码，绘制栈图，查看程序输出为何?

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

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

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

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

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

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

参考 *http://en.wikipedia.org/wiki/Ackermann_function*. 编写 `ack` 程序，实现 *Ackermann* 函数。用此代码执行 `ack(3, 4)`，结果应该为 *125*. 同时，换较大的 `m` 和 `n`，看看结果有何不同？答案参见: *http://thinkpython2.com/code/ackermann.py*.

Exercise 6.3. 像 *"noon"* 和 *"redivider"* 一样，正序和倒序拼写方式完全一样的词，称为回文词。从递归角度看，如果开始和结束字母相同，同时中间部分是回文词，那么就可以认为总体是回文词。下文为返回字符串首字母，尾字母以及中间字母的函数:

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

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

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

章节 *8*会详细解释其原理.

1. 将这些函数代码，写入文件 palindrome.py，并测试输出。分别用两个字母，一个字母，测试 middle 函数，看看会怎么样？尝试传入不包含任何字母的空字符串''，会如何?

2. 编写is_palindrome 函数，如果传入参数为回文字符串，则返回 True，否则返回 False。提示一下，你可以使用内置函数 len 检验字符串长度。

答案参阅: *http://thinkpython2.com/code/palindrome_soln.py*.

Exercise 6.4. *a* 是 *b* 是幂次方，表示 *a* 可以被 *b* 整除，同时 a/b 也是 *b* 的幂次方。编写函数is_power，以 a 和 b 为参数，如果 a 是 b 的幂次放，则返回 True。提示：注意边界条件。

Exercise 6.5. *a* 和 *b* 的最大公约数 *(GCD)*，是指能同时被整除的所有约数中的最大的一个。

寻找最大公约数的一种方法便是观察，如果 *r* 是 *a* 除以 *b* 的余数，那么 $gcd(a,b) = gcd(b,r)$，同时 $gcd(a,0) = a$.

编写函数gcd，以 a 和 b 为参数，返回其最大公约数。

致谢: 此习题借鉴了 *Abelson* 和 *Sussman* 的 Structure and Interpretation of Computer Programs 中的样例.

第 7 章　迭代

本章主讲迭代，其主要实现重复执行一段语句。在章节 5.8中的递归，便是一种迭代。在章节 4.2中的 `for` 循环，也是一种迭代。本章，我们会接触另一种迭代，`while` 语句。但这里要先讲一下变量赋值。

7.1　再赋值

你也许已经注意到，相同的变量可以被多次赋值。重新赋值会将已存在的变量指向新的值(并且不再指向旧的值).

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

第一次输出 `x`，值为 5, 第二次，值为 7.

图 7.1 展示了**再赋值**在栈图中的过程。

在此，我想澄清一下大家的困惑. 因为 Python 使用等号 (=) 进行赋值，有人便会将类似 `a = b` 这样的语句进行数学理解，认为其表示 `a` 和 `b` 相等。这总理解是极其错误的。

首先，等式是对称关系，而赋值不是。例如，数学中，如果 $a = 7$，那么 $7 = a$. 但是在 Python 中，`a = 7` 正确，但是 `7 = a` 则不然。

同样，数学领域，等式结果要么真要么假。如果 $a = b$，那么 a 总是等于 b。而在 Python 中，虽然可以将变量赋值相同，但是无法保证永远相等:

```
>>> a = 5
>>> b = a    # a 和 b 相等
>>> a = 3    # a 和 b 不等
>>> b
5
```

第三行代码，改变了 `a` 的值，但是没有改变 `b`，所以他们不再相等。

很多时候我们需要给变量再赋值，但是要谨慎使用。如果变量的值变动过于频繁，代码将难以理解维护。

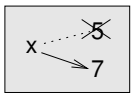

图 7.1: 栈图

7.2 变量更新

最常见的一种再赋值，便是**更新变量**, 通常是基于前值进行修改而得到新值。

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

这句代码表示“获取 x 的值，然后加一，得到新值，继而用新值更新 x。“

如果更新未定义变量，Python 会在赋值 x 前执行右侧表达式，从而报错:

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

更新变量前，一定要**初始化**，通常采用简单赋值来实现:

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

对变量进行加 1 来更新，叫做**递增**; 执行减 1 更新变量，叫做**递减**。

7.3 while 语句

计算机通常可以用来自动化一些重复性工作。对于大量重复的相同或相似任务，计算机可以永不犯错，而这也是计算机精擅之处，却恰恰是人类最不擅长的。在计算机编程中，这种重复性，便称作**迭代**.

我们已经遇到过两个函数，countdown 和 print_n, 它们都是使用递归进行迭代。因迭代应用场景较多，Python 提供了一些内置功能来便捷使用。一个便是在章节 4.2中见到的 for 语句，后续再讲。

另一个便是 while 语句，下面是 countdown 的 while 语句版本:

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

while 语句很容易理解，因为其便如英语表达一样。意为,“当 n 大于 0 时，打印 n 值，然后自减 1. 当等于 0 时，打印 Blastoff!”

正式些讲，下面为 while 语句的执行流程:

1. 确定条件之真假.
2. 如果为假，退出 while 语句段落，执行其后语句。
3. 如果为真，运行执行体，回到第一步。

这种程序流转，便叫做循环。因为第三步骤时，会返回到起点。

循环体中，通常需要修改变量的值，从而令条件最终为假，停止循环。否则，会一直循环下去，进入**无限循环**。对计算机科学家们，有个乐此不疲的玩笑，便是观察洗发水的使用说明，“起泡，冲洗，重复"，这就是个无限循环，哈哈。 在 countdown 中，保证循环终止的办法：如果 n 小于或等于 0, 循环终止。由于 n 每循环一次，就会变小，终究会变成 0.

而有些循环，则难以轻易讲清楚，比如:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n 是偶数
            n = n / 2
        else:                 # n 是奇数
            n = n*3 + 1
```

此循环的条件是 n != 1，那么只有 n 等于 1 时，条件为假，循环终止。

每次循环，程序都输出 n 的值，然后检查是奇是偶。如果是偶数，n 除以 2。如果是奇数，n 值更新为 n*3 + 1。例如，给 sequence 传入参数 3, n 值依次是 3, 10, 5, 16, 8, 4, 2, 1.

由于 n 有时增，有时减，很难确定 n 什么时候变为 1, 从而终止循环。对于某些 n 值，我们可以确定循环会终止。比如，初值为 2 的幂，那么每次循环都会是偶数，从而结果为 1. 上例中得到的数列，从 16 开始，便是如此。

难点在于如何针对所有 正数的 n，保证程序结束。目前，还没有人能证明或者证否此命题。(参见 http://en.wikipedia.org/wiki/Collatz_conjecture)

做个练习，用迭代替换递归，重写章节 5.8 的 print_n 函数，

7.4 中断

有时，我们想出红尘，需要先入红尘，只有进入循环体，才知道何时应当终止循环。这时，我们使用 break 语句跳出循环。

例如，直到用户输入 done，才跳出循环，遇到此情况，可以用下面代码实现:

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

print('Done!')
```

循环条件为 True, 便意味循环不止，所以只有满足中断条件，才跳出循环。

每次循环，都会打印大于号来提示用户输入，如果用户输入了 done，break 语句会终止循环。否则，程序会打印用户的输入，并再次进入下一轮循环。此处为样例:

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

这种写法在 while 循环中很常见，因为你可以随时检验其条件 (而不仅仅在头部验证)，同时你可以“主动去终止" 循环，而不是等待“被动结束".

7.5 平方根

在编程中，循环往往用来通过近似值，不断逼近真实值，来进行数值计算。

例如，可以采用牛顿公式 (Newton's method) 来计算平方根。比如，计算 a 的平方根，开始先确定任意一个估计值, x，然后通过下面公式，计算更优的值:

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

假设，a 为 4，x 为 3:

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

结果很接近正确答案 ($\sqrt{4} = 2$). 如果我们用新的近似值，重复刚才的流程，结果会更接近:

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

多重复几次，结果便更准确:

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

通常，我们很难清楚知道，运行多久，才能得到正确结果。但是，我们很容易确定，结果不再明显变化的时候:

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

当 `y == x` 时，我们便可以停止循环了。下面是个例子，其以估计值 `x` 开始，在结果不再变化时终止:

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

对于大部分的 a 值，此方法都有效。但用浮点数来比较等式，很危险。浮点值一般认为是近似正确：大部分的有理数，比如 1/3，以及类似 $\sqrt{2}$ 的无理数，都无法用浮点数来精确表示。

与其费劲比较 x 和 y 是否相等，不如用 abs 计算其差值的绝对值或者幅度:

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

当表达式在epsilon 为 0.0000001 时仍然满足，那便说明两个值已经足够接近。

7.6 算法

牛顿公式可以认为是一种**算法**: 通过既定步骤来解决一类问题 (此例中为计算平方根)。

要理解何为算法，先要明白什么不属于算法。当你学习乘法时，往往需要记忆乘法表。实际，你记忆了 100 个特定答案。这种知识不属于算法。

但如果你很“懒“，你可能会发现一些小技巧。比如，你想计算 n 和 9 的乘积，只需把 $n-1$，作为第一个数字。$n-2$ 作为第二个数字即可。这个小技巧对于任何数字乘以 9 都有效。这便是算法! 同样地，你学过的需进位的加法，需借位的减法，以及长除法，都是算法。这些算法的一个共性便是，都无需费力思考，它们都是机械的过程，遵循简单的规则，一步又一步，便可得到结果。

算法的执行过程很无聊，不过其设计过程却有趣又挑战智力，同时也是计算机科学的核心。

有些事情，对于人们来说没有难度，都是自然而然，下意识完成的事情，却是最难用算法解决的。比如，理解自然语言。我们所有人都能轻易做到，但是，却没有人能阐明我们是如何理解的，更不用说用算法的形式来解释。

7.7 调试

随着程序代码规模的增长，你会发现需要耗费更多时间在调试上。一般更多的代码，也就意味着更多的出错可能，以及更多的潜在问题。 一种有效节约调试时间的方法便是“二分调试". 比如，有 100 行代码，一次检查一行，需要 100 次。

那么，我们可以尝试在代码的中间位置，或者靠近中间的位置，开始检查。通过添加一些 print 语句 (或者其他可验证效果的东西)，再运行代码。

如果中间检查点出现异常，那代码前半部分有问题。如果中间没有出错，那么问题便在后半部分了。

如此检验代码，查询问题点的耗时大大减半。大约六步之后 (远远小于 100)，理论上，便只剩下一两行代码需要检查了。

实际上，很难清晰界定“代码中间位置", 同时也很难在其位置检验。所以没必要计较于行数，执着于中点。而是应该多思考代码的哪些位置容易出错，哪些地方又容易验证。然后选择一个恰到好处的点，进行验证。

7.8 术语表

再赋值 (reassignment): 给已存在的变量赋新值的过程。

变量更新 (update): 基于前值，计算新值，更新变量的过程。

初始化 (initialization): 给变量赋初始值，以待后续更新。

递增 (increment): 变量不断增加某个值 (通常为 1)。

递减 (decrement): 变量不断减少某个值。

迭代 (iteration): 采用递归或者循环，重复执行一段语句。

无限循环 (infinite loop): 循环中的终止条件永未达到。

算法 (algorithm): 解决一类问题的通用步骤。

7.9 习题集

Exercise 7.1. 复制章节 *7.5* 的循环，封装为mysqrt 函数，令其以 a 为参数，选择一个合适的初始值 x, 返回 a 的近似平方根。

编写名为test_square_root 的函数，进行测试，并输出以下表格:

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

第一列是数值 *a*; 第二列是 mysqrt 函数计算出的 *a* 的平方根; 第三列是 math.sqrt 计算出的 *a* 的平方根; 第四列是两者的差值绝对值。

Exercise 7.2. 内置函数 eval 以字符串为输入，并用 *Python* 解释器执行。如下:

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

编写函数eval_loop，对用户交互提示，获取输入，同时用 eval 执行，并打印结果。

直到用户输入 'done' 才终止，否则程序一直运行，同时输出最后一次表达式的结果。

Exercise 7.3. 数学家 *Srinivasa Ramanujan* 发现了一个无穷级数，此级数可以用来计算 $1/\pi$ 的近似值*:*

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

编写函数 `estimate_pi`，使用上述公式计算 π 的近似值，同时返回结果。用 `while` 循环计算求和的项，直到最后一项小于 `1e-15` *(python* 中对于 10^{-15} 的表示法*)*。可以和 `math.pi` 比较结果，检验效果。

答案参见：*http://thinkpython2.com/code/pi.py*

第 8 章　字符串

字符串不同于整数，浮点数以及布尔值。一个字符串就是一个**序列**，也就是说，字符串就是将值进行有序排列。本章你将学习如何通过字符构造字符串，以及字符操作相关方法。

8.1　字符串即序列

字符串便是字符系列。通过中括号，可以获取其中的字符:

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

第二句表达式会从 `fruit` 中选择在位置 1 处的字符，并赋给 `letter`。括号中的表达式，叫做**索引**. 索引标识了你将从序列中获取哪个字符 (类似名字).

但有时候你所得非所愿:

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

如众人所知，`'banana'` 的第一个字母是 `b`，而不是 `a`。但是，对于计算机科学家来说，索引是从字符串开始位置的偏移量，所以第一个字符的偏移量是 0.

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

所以 `b` 是`'banana'` 的第 0 个字母，`a` 是第 1 个，`n` 是第 2 个。

你可以使用包括变量和操作符的表达式作为索引:

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

但索引的值必须是整数。否则，会遇到错误:

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

8.2 len

len 是内置函数，可以返回字符串中的字符数量:

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

想要获得字符串最后一个字符，可以尝试下面的操作:

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

报出 IndexError 的原因是'banana' 中索引 6 的位置没有字母。因为索引从 0 开始，那 6 个字母对应的数字是 0 至 5. 想获取最后一个字符，需要字符串长度减 1:

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

你也可以使用负数索引，从字符串末尾向开始数。表达式 fruit[-1] 给出最后一个字母，fruit[-2] 给出了倒数第二个字母，以此类推。

8.3 用 for 循环遍历

很多操作一次仅操作字符中的一个字符。一般从头部开始，顺序获取字符，做些操作，直到末尾。这个流程叫做**遍历**. 可以用 while 循环进行遍历:

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

此循环遍历字符串，并将每个字符各显示一行。循环条件是 index < len(fruit)，所以当 index 等于字符串长度时，条件为假，循环体终止。最后字符获取时，索引为 len(fruit)-1，说明这就是最后一个字符。

做个练习，写个函数，以字符串为入参，倒序输出每个字符，一行一个。

遍历字符串的另一种方法是用 for 循环:

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

每次循环后，字符串中下一个字符会赋值给变量 letter。直到没有字符，循环终止。

下面的例子，展示了如何用拼接 (字符串相加) 以及 for 循环，来构建一个简单序列 (按照字母顺序)。在 Robert McCloskey 的书 *Make Way for Ducklings* 中，小鸭子们的名字便是 Jack, Kack, Lack, Mack, Nack, Ouack, Pack, 和 Quack. 循环依次输出名字如下:

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

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

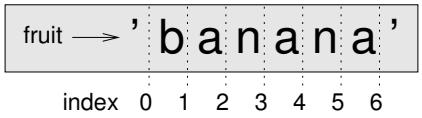

图 8.1: Slice indices.

输出如下:

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

显然，上面结果并不是全部正确，因为 “Ouack” 和 “Quack” 都拼错了。做个练习，修改程序，使其正常。

8.4 字符串切片

字符串的一部分，叫做**切片**. 选择切片和从字符串中选择字符很像:

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

操作符 `[n:m]` ，会返回字符串从第 n 个位置到第 m 个位置的字符，包括开头的第 n 个字符，但是不包括最后第 m 个字符。可能有点反常识，但想象一下如图 8.1 所示的索引点之间的字符，可能更好理解。

如果缺失第一个索引 (冒号前)，则切片从字符串头部开始。如果忽略了第二个索引，切片到末尾结束:

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

若第一个索引值大于或等于第二个，则结果为**空字符串**，用两个单引号表示:

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

空字符串一般不包括任何字符，同时长度为 0，除此之外，和其他字符串一样。

继续上面的例子，思考一下 `fruit[:]` 会得到什么？试试吧。

8.5 字符串不可变

尝试在表达式左侧用 [] 操作符修改字符串中的字符。例如:

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

“object” 在这里指字符串，“item” 指试图赋值的字符。到目前为止，你可以认为对象和值一样，但是后续 (章节 10.10) 中，会对此描述详细讨论。 错误的原因在于字符串是**不可变的**，也就是说，你无法改变一个既有的字符串。你能做的，是在原来字符串基础上，新建一个不同的字符串:

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

上述代码是用新的首字母和 greeting 的切片进行了拼接，而这并不会改变原来的字符串。

8.6 查找

下面的函数什么用途?

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

可以认为，find 是 [] 操作符的逆操作。此操作，不同于根据索引获取对应字符，而是根据字符，查找其索引。如果字符没有搜索到，则返回-1。

这是我们第一次见到，在循环内使用 return 语句。如果 word[index] == letter，函数会跳出循环，并立刻返回结果。

如果字符串中未搜索到想要的字符，程序一直执行到循环结束，并返回-1.

这种算法--遍历序列并返回预期结果--叫做**查找**。

做个练习，为 find 函数加入第三个参数，一个索引值，使其从 word 的此索引处开始查找。

8.7 循环和计数

以下程序会统计字符串中 a 出现的次数:

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

此程序描述了另一种算法，称之为**计数**. 初始化变量 count 为 0, 其后每找到一次 a，就加 1. 循环结束后，count 便包含了结果– a 的数量.

做个练习，封装以上代码为 count 函数，使其以字符串和字母为参数，从而普遍通用。

然后重写函数，用上一节的 find 函数替换字符串遍历操作。

8.8 字符串方法

字符串提供了诸多有用的方法，方法类似于函数--输入参数并返回结果--但是句法有些不同。例如，upper 方法，会读取字符串，并返回一个全部字母大写后的字符串。

与函数句法 upper(word) 不同的是，方法句法写作 word.upper().

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

这种点标法，声明了方法名字，upper，和要使用此方法的字符串的名字，word。括号为空表明此方法没有参数。令函数运行，叫做**调用**; 此例中，我们说调用 word 的 upper 方法。

你会发现，字符串实际内置了 find 方法，但和我们写的函数惊人相似:

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

此例中，调用 word 的 find 方法，将要查找的字母作为入参。

实际上，find 函数比我们写的要更通用; 它不仅可以定位字符，也能定位字符串段落:

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

默认情况，find 从字符串开头进行查找，不过也可以给其传入索引值作为第二个参数，使其从既定位置开始:

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

这是一个**可选惨素**的例子; find 方法也可以接收第三个索引参数，使其在某处结束:

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

b 没有出现在字符串索引 1 和 2 且不包括 2 的范围内，所以搜索失败。而这种到达第二个索引位但不包括此索引的规则，和切片操作一样。

8.9 操作符 in

单词 in 是一个布尔运算符，其比较两个字符串，如果前者是后者的一部分，则返回 True:

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

例如，下面的函数会输出同在 word1 和 word2 中出现的字母:

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

变量名选择得足够好的话，Python 读起来便如同英语。读一下这个循环，for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

此处为 apples 和 oranges 比较的结果:

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

8.10 字符串比较

关系运算符同样适用于字符串。比如要判断两字符串是否相等:

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

其他的关系运算符也适用于按照字母顺序进行比较:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python 处理大小写的方式同人类思维不同，在其看来，大写字母都排在消协字母之前，所以:

```
 Pineapple 在前， banana 在后.
```

通常来说，一般在比较前，先统一字符串的格式，比如都转小写。谨记这一点，以免遇到 Pineapple 时，变得一团糟。

8.11 调试

想要用索引来遍历序列中值，难点在于确定遍历的起点和终点。下面是一个比较字符串的函数，如果某字符串恰好是另一字符串的倒序排列，则返回 True，但是代码中有两处错误:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
```

```
    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

第一个 if 语句会判断两个单词的长度是否一样。如果不同，立刻返回 False，现在假设单词长度相同，执行后续。这是一个哨兵模式，在章节 6.8中已经介绍过。

i 和 j 是索引: i 正向遍历 word1，同时 j 倒序遍历 word2。如果遇到两个字母不同，则即刻返回 False。如果通过循环检验，则所有字母都匹配，返回 True。

用 “pots” 和 “stop” 测试函数，我们期望得到 True，但是会遇到索引错误:

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

调试此种异常，第一步就是在错误出现位置前，先输出索引的值。

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

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

再次运行程序，得到如下信息:

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

首次循环，j 的值是 4, 超出了'pots' 的索引范围。最后一个字符的索引应该是 3, 所以 j 的初始值应该是 len(word2)-1.

如果修复此错误，并再次执行，输出如下:

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

这次我们的到了正确结果，但是循环只运行了三次，有点奇怪。为了弄清怎么回事，可以绘制栈图来辅助理解。第一次迭代，is_reverse 的框见图 8.2.

我尝试在框中对齐变量，并用虚线标识 i 和 j 的值，以表示 word1 和 word2 中的字符，从而帮助理解。

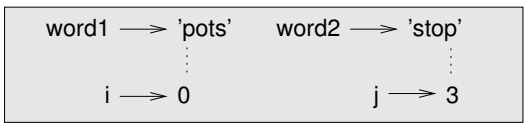

图 8.2: 栈图.

根据程序流转，以及每次迭代中 i 和 j 值的变化，继续绘制栈图。请在此函数中继续寻找并修复第二个错误。

8.12 术语表

对象 (object): 变量引用的东西，目前，可以将“对象”与“值”同样看待。

序列 (sequence): 一些值的有序集合，每个值都对应一个整数索引。

元素 (item): 序列中的一个值.

索引 (index): 一个整数值，用来选择序列中的元素，比如选择字符串中某个字符。Python 中的索引都是从 0 开始。

切片 (slice): 字符串中一部分，通过索引范围确定。

空字符串 (empty string): 没有字符并且长度为 0, 同时用两个引号表示的字符串。

不可变 (immutable): 序列中元素不可改变的特性。

遍历 (traverse): 迭代序列中每个元素，并执行相似操作的过程。

查找 (search): 寻找预期目标的遍历模式。

计数器 (counter): 用来计数的变量，一般起于 0，而不断递增。

调用 (invocation): 运行方法的语句.

可选参数 (optional argument): 函数或者方法中的不必需参数。

8.13 习题集

Exercise 8.1. 阅读文档*http://docs.python.org/3/library/stdtypes.html#string-methods* 中的字符串方法，可能你会想试试其中一些方法，尽量弄明白它们的工作原理。strip 和 replace 特别有用。

文档中的语法可能难以理解。比如find(sub[, start[, end]]) 方法，方括号标识了可选参数。sub 是必需的，但是 start 是可选的，如果包含了 start，end 便是可选的。

Exercise 8.2. 有个 count 方法，和章节 *8.7*中的函数很相似。阅读此方法文档，编写调用此方法的代码，实现对'banana' 中 a 个数的统计.

Exercise 8.3. 字符串切片也可以有第三个参数，叫做“步长”; 也就是，在连续字符中，字符间的间距。步长为 *2* 表示每隔一个字符取一个; 步长为 *3* 表示每第三个取一个，以此类推。

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

步长为*-1*, 则表示倒序读取，所以 `[::-1]` 切片，便会产生一个倒序的字符串。

用这个神奇魔法，将习题 *6.3*中的 `is_palindrome` 修改为一行版本吧。

Exercise 8.4. 下面的函数都是试图检验字符串中是否包含小写字母，但是肯定有函数存在问题。仔细分析每个函数并明了其用途*(*假设入参都是字符串*).*

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercise 8.5. 凯撒加密 *(Caesar cypher)* 是一种通过对每个字母进行特定数值的“移位”操作，而实现的简单加密方案。对字母移位，也就是按照字母顺序，进行移动，如果直到末尾，位数不足，则从头继续。所以*'A'* 移位 *3*，得到*'D'* ，*'Z'* 移位 *1,* 得到*'A'*。

对一个单词移位，就是针对每个字母，都进行相同的移位数量。例如 *“cheer”* 移位 *7,* 则为 *“jolly”*，*“melon”* 移位*-10,* 则为 *“cubed”*。在电影 2001: A Space Odyssey 中，飞船电脑名叫 *HAL*，就是 *IBM* 移位*-1* 得到的。

编写`rotate_word` 函数，以字符串和整数为参数，对字符串中的字符进行数值移位，得到新字符串，并返回。

你可能会想用内置的 `ord` 函数，此函数可以将字符转为数字码，而 `chr` 则可以将数字码转回字符。字母表中的字母会按顺序进行编码，比如*:*

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

因'c' 在字母表中是第 2 个 (从 0 开始)，所以结果为 2。但要注意：大写字母的数字码和小写的不同。

网络上一些嘲弄都是采用了 *ROT13* 进行加密，也就是移位值 *13* 的凯撒加密。如果你不会太介意，试试解密它们吧。参阅: *http://thinkpython2.com/code/rotate.py*.

第 9 章　案例学习: word play

本章学习第二个案例，主要研究如何通过搜索特定词汇，进行猜迷。比如，查找最长回字文，以及寻找按照字母表顺序排列的单词。同时，我将介绍一种新的程序开发模式: 降低复杂度，仍然能够解决以前的问题。

9.1　读取单词列表

本章的练习，需要准备一个英文单词列表。网上有很多可用的单词列表，但是对我们来说，最理想的莫过于 Grady Ward 收集整理，作为 Moby 词典项目，分享给公告领域的单词列表 (详见`http://wikipedia.org/wiki/Moby_Project`)。这是包含 113,809 个游戏填词的单词列表; 也就是说，这些单词，已经被填词游戏和其他单词游戏证明了有效。在 Moby 项目中，这个文件名为 `113809of.fic`; 你可以从`http://thinkpython2.com/code/words.txt` 下载一个副本，其名字简称 `words.txt`。

此文件为纯文本文件，你可以用文本编辑器打开，但你也可以用 Python 读取。内置函数 `open`，以文件名为入参，返回一个**文件对象**，可以用来读取文件内容。

```
>>> fin = open('words.txt')
```

`fin` 是表示输入的文件对象的通用名称。文件对象针对读取提供了多个方法，包括 `readline`，此方法会读取文件的一整行字符，并作为字符串返回:

```
>>> fin.readline()
'aa\n'
```

单词列表中第一个单词是 “aa”，这是一种岩浆。后面的`\n` 是换行符，用来断行。

文件对象会跟踪目前读到哪里了，从而，再次运行 `readline`，会得到下面单词:

```
>>> fin.readline()
'aah\n'
```

下一个单词是 “aah”，这是个绝对正确的单词，所以别用异样眼光看我。另外，如果换行符令你厌烦，可以用字符串方法 `strip` 移除:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

你也可以将文件对象置于 `for` 循环中。这样程序便会读取 `words.txt`，然后逐行输出每个单词:

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

9.2 练习

下一节有这些习题的答案，但尽量在看答案之前尽力一试吧。
Exercise 9.1. 编写程序，读取 words.txt，同时打印多于 *20* 个字符的单词*(*不包括空格*)*。
Exercise 9.2. *1939* 年 *Ernest Vincent Wright* 出版了一部 *50,000* 单词的小说，名叫 Gadsby，本书不包括字母 *"e"*。而英文中最常用的便是 *"e''*，所以太难得了。

事实上，若不考虑通用的字符，甚至难以独立思考。不过，开始虽然进展缓慢，但是，训练几个小时，你便会慢慢习惯。

好了，闲言少叙。

编写 has_no_e 函数，如果输入单词不包括 *"e''*，则返回 True。

编写程序，读取 words.txt，只打印不包含 *"e''* 的单词。统计列表中，不包含 *"e''* 的单词的比例。
Exercise 9.3. 编写 avoids 函数，以单词和禁用字母字符串为输入，当单词不包含任何禁用字母时，返回 True。

编写程序，令用户输入禁用字母字符串，统计不含有这些字母的单词数量。看看你是否可以找出一个包含 *5* 个禁用字母的组合，从而排除的单词数量最少*?*
Exercise 9.4. 编写函数uses_only，以一个单词和一串字母组合为输入，如果单词的字母都在这个字母组合中，则返回 True. 你是否可以只用 acefhlo 这些字母，构造出句子来？换成 *"Hoe alfalfa''* 这些字母呢？
Exercise 9.5. 编写函数uses_all，输入为一个单词和一个必需字母的字符串，如果单词对必需字母，都至少使用了一次，则返回 True。看看有多少单词同时包含 aeiou？又有多少同时包含 aeiouy 呢*?*
Exercise 9.6. 编写函数is_abecedarian，如果单词中的字母是按照字母表顺序排列，则返回 True*(*字母相同，视为顺序*)*。看看有多少这种单词*?*

9.3 检索

上一章节中，所有的练习，有个共同之处; 它们都可以采用章节 8.6中的检索方法来解决。举个简单例子:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

for 循环会遍历 word 中所有字母。如果遇到 "e''，则即刻返回 False; 否则继续下一个字母。如果循环正常结束，也就是说没有遇到 "e''，则返回 True.

你也可以使用 in 运算符来精简程序，我先介绍此上述版本，主要是阐述清楚检索方法的逻辑。

avoids 函数相比 has_no_e 版本，功能更加通用，但结构相同:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

在此函数中，一旦遇到禁止字母，即刻返回 False，如果循环终了，则返回 True。

uses_only 与之极为相似，无非条件相反:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

这里不再是禁用字母列表了，而是可用字母列表。如果 word 中出现了不在可用列表中的字母，返回 False.

uses_all 和上述函数也较为相似，不同之处在于，我们交换了单词和字母组合的角色:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

不再是遍历 word 中的字母，而是遍历必需字母表。如果字母列表中有字母未出现在单词中，返回 False.

如若你已像计算机科学家一样去思考，你会发现 uses_all 不过是以前一个已解决的问题的另一种表现形式，你可能会这么写:

```
def uses_all(word, required):
    return uses_only(required, word)
```

此案例便是一个**抽离纷扰，回归已知**的程序开发模式的实践，这种模式是将遇到的问题，映射为已解决的问题，从而用已有的解决方案来解决当前问题。

9.4 索引循环

前一章节中，我用 for 循环编写了大量函数，这是因为我只需字符串中的字符; 所以无需关注索引。

但is_abecedarian 函数中，我们不得不比较相邻的字母，用 for 循环便不太方便:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

用递归来实现:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

还可以使用 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
```

此循环始于 i=0，终于 i=len(word)-1。每次循环都会比较第 i 个字符 (可以看作当前字符) 和第 $i+1$ 个字符 (可以看作后一个字符)。

如果后一个字符小于 (在字母表顺序中前于) 当前字符，我们会发现这和要求不符，便返回 False.

如果直到循环末尾，仍然无异常，则单词合格。为确信循环正常结束，可以试试'flossy'。单词长度为 6, 所以 i 为 4 时，便是最后一次循环，因为这是倒数第二个字符的索引。最后一次循环中，比较了倒数第二个和倒数第一个字符，这恰恰符合了预期。

下面是is_palindrome 函数 (参看习题 6.3) 的另一个版本，其使用了两个索引; 一个从开头到结尾，顺序前进; 另一个从结尾到开头，倒序进行.

```
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
```

或者，我们映射为前述问题，从而构建解决方案，进行重写:

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

此处使用了章节 8.11中的 is_reverse 函数.

9.5 调试

程序测试很难。本章的函数进行测试，相对简单，因为你可以手算来检验结果。但是如果选择一批单词，然后检验所有可能错误，那不仅困难，甚至难以完成。

以has_no_e 为例，需要检验两种情景: 含有 'e' 的单词，返回 False，以及不含有 'e' 的单词，返回 True。对此，你应该很容易验证任何一种情况。

对于每种情景，又有一些不太明显的子用例。对于包含 "e" 的单词，你又要检验 "e" 在开头的单词，在结尾的，甚至中间某些位置的单词。你要检验长单词，短单词，甚至非常短的单词，比如空字符串。空字符串是**特例**的一种, 也就是那种容易被忽视，却往往是错误频发之处的用例。

除了自己构建测试用例，你也可以用类似 `words.txt` 的单词列表检验程序。通过扫描输出，你可能会发现错误，但要小心：你可能发现了某种错误 (包含了不应包含的单词)，而忽略了其他错误 (没有包括应该包含的单词)。

通常，程序测试可以帮助你发现异常，但是制作出好的测试用例，往往很难，即使通过了这些测试，你也不能百分百地确认你的程序正确。一位传奇的计算机科学家说过:

> 测试只能表明代码异常的存在，却永远无法证明其不存在!
>
> — Edsger W. Dijkstra

9.6 术语表

文件对象 (file object): 代表一个打开的文件的值.

抽离纷绕，回归已知 (reduction to a previously solved problem): 一种通过映射以前类似的问题，从而参考已有方案，进而解决问题的方式.

特例 (special case): 非典型或者不明显的测试用例 (往往容易犯错的地方).

9.7 习题集

Exercise 9.7. 这个问题源于一个在广播节目 Car Talk 中出现的迷题 (*http://www.cartalk.com/content/puzzlers*):

> 给我一个存在三个连续的双字母的单词。我会给你一些看似符合，其实不符的单词。例如，单词 *committee*，*c-o-m-m-i-t-t-e-e*。如果没有 *'i'* 在中间，就很完美。又或者 *Mississippi: M-i-s-s-i-s-s-i-p-p-i.* 如果将其中的 *i* 都移除，就完美了。但是有一个词恰好含有三个连续的双字母，而且据我所知，这可能是仅有的一个这样的单词。当然，实际上有可能有 *500* 多个这样的单词，但我只能想到一个。是哪个单词呢?

编写程序来寻找一下吧。答案: *http://thinkpython2.com/code/cartalk1.py*.

Exercise 9.8. 这也是一个 Car Talk 的迷题 (*http://www.cartalk.com/content/puzzlers*):

> *''*某天我在高速上开车，碰巧注意到里程表。同多数里程表一样，有 *6* 个数字，只能表示整里数。所以，如果我的车跑了 *300,000* 英里，那显示的就是 *3-0-0-0-0-0.*
>
> *''*现在，我看到的很有意思。最后 *4* 个数字是回文;也就是说，从前往后读和从后往前读都一样。比如 *5-4-4-5* 便是个回文，所以我的里程表可能显示的是 *3-1-5-4-4-5.*
>
> *''*一英里后，最后的 *5* 个数字也是回文。例如显示为 *3-6-5-4-5-6*。然后又跑了 *1* 英里，*6* 个数字中间的 *4* 位是回文了。准备好玩这个了吗？那又跑了 *1* 英里，所有 *6* 个数字是回文了!
>
> *''*问题来了，我开始在里程表上看到的数字是什么?*"*

编写个 *Python* 程序，检验所有的 *6* 位数字，然后输出满足上述要求的任意数字。参阅: *http://thinkpython2.com/code/cartalk2.py*.

Exercise 9.9. 再来一个 Car Talk 上的迷题，你可以用检索法来解决 (*http://www.cartalk.com/content/puzzlers*):

> “最近我去看望母亲，我们发现我的年龄倒过来，正好是母亲的年龄。比如，若她是 *73*, 我是 *37*. 我们想知道在过去这些年，有多少次这种情况发生，但后来我们岔开了话题，没有得到答案。
>
> “我回到家后，我发现到目前为止，我们的年龄已经相逆了 *6* 次。同时，我也意识到，如果我们幸运的话，稍后几年，我们又会年龄互逆一次，如果我们特别幸运，就还会有一次机会。换句话说，这样的情况，我们一共会遇到 *8* 次。所以，请问，我现在多少岁了?”

编写 *Python* 程序，寻找这个迷题的答案。提示: 你可能会用到字符串的 `zfill` 方法。

参阅: *http://thinkpython2.com/code/cartalk3.py*.

第 10 章　列表

本章讲述 Python 最有用的内置类型，列表。你还会接触到对象，了解一个对象对应多个名称时的神奇景象。

10.1　列表即序列

同字符串一样，**列表** 也是值的序列。字符串中，值是字符串; 在列表中，值可以是任何类型。列表中的值，叫做**元素**，有时也叫**列表项**.

创建列表的方式很多; 最简单的就是将元素用方括号包起来 ([and]):

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

第一个列表由四个整数构成，第二个列表则由三个字符串构成。列表中的元素类型可以不同。下面的列表，便同时包括了字符串，浮点数，整数和另一个列表:

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

列表中包含列表，叫做**嵌套列表**.

不包含任何元素的列表，叫做空列表; 你可以用[] 创建空列表.

如你所想，列表也可以赋给变量:

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

10.2　列表可变

从列表中获取元素的语法和从字符串中获取字符的语法一样--方括号操作符。括号中的表达式确定了索引值。要注意，索引从 0 开始:

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

和字符串不同，列表是可变的，当括号操作符出现在赋值语句左侧，就会将对应的列表元素重新赋值。

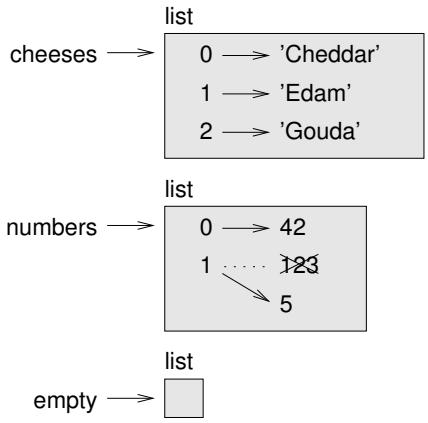

图 10.1: 状态图.

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

numbers 中的地一个元素，原来是 123, 现在变成了 5.

图 10.1 是 cheeses, numbers 和 empty 的状态图.

列表用单词 "list" 在外，元素在内的箱体图表示。cheeses 指向索引为 0,1 和 2 的三个元素的列表。numbers 包括两个元素; 图中也展现了第二个元素从 123 被赋值为 5 的过程。empty 指向了一个空列表。

列表索引和字符串索引的作用是一样的:

- 索引可以是任意整型表达式.
- 如果试图通过索引读写不存在的晕素，会得到 IndexError.
- 如果索引为负值，则从列表末尾倒序计数。

in 运算符也可用于列表。

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

10.3 遍历列表

遍历列表中元素的最常用方法便是 for 循环。语法同字符串遍历:

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

这种方法对于仅读取列表中元素很方便，但是如果你想写入或者更新元素，那便需要索引了。通常把内置函数 range 和 len 结合使用:

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

此循环会遍历列表并更新每个元素。len 方法会返回列表中元素数量。range 则返回从 0 到 $n-1$ 的索引值列表，其中 n 是列表的长度。每次循环，i 都会获得下一个元素的索引。循环体中的赋值语句则会通过 i 获取元素的旧值，并赋予新值。

空列表的 for 循环，永远不会运行到循环体:

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

虽然列表可以包含其他列表，但是嵌入的列表仍表示一个呀 unsu。下面列表的长度为四:

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

10.4 列表操作

+ 运算符可以拼接列表:

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

* 运算符会将列表复制一定次数:

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

第一个示例复制了 [0] 四次。第二个，则复制了 [1, 2, 3] 三次。

10.5 列表切片

切片运算符同样适用于列表:

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

如果省略第一位索引，则从头开始，进行切片。如果省略第二位索引，则切片止于末尾。如果两位都省略，则切片便等同于复制整个列表。

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

因为列表可变，所以通常都会在修改列表操作前，复制一份，以防万一。

如果切片运算符在赋值号左侧，也可以同时更新多个元素:

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

10.6 列表方法

Python 提供了大量操作列表的方法。例如，append 可以在列表末尾添加新元素:

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

extend 会以列表为参数，并将其中所有元素添加到执行列表中:

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

这个例子中，t2 没有被修改。

sort 会将列表元素，从低到高排序:

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

多数列表方法都没有返回值; 这些方法修改列表然后返回 None。如果你意外写了 t = t.sort()，结果会令你失望。

10.7 Map, filter 和 reduce

若想对列表中所有数字求和，可以用如下循环实现:

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

total 初始为 0. 每循环一次 x 会从列表中获取一个元素。+= 操作符，是一种更新运算的简写。也是一种**扩展赋值语句**。

```
    total += x
```

等同于

```
    total = total + x
```

随着循环运行，total 会累积元素求和; 这样的变量有时也称为**累加器**。对列表元素求和，Python 提供了更加方便的操作，便是内置函数 sum:

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

像这样将一系列元素合为一个值的操作，一般称为 **reduce**.

有时，你会遍历列表，创建新列表。例如，下述函数会接收字符串列表，并返回一个将所有字符串都变为首字母大写字符串的列表:

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

res 始于一个空列表; 每次循环，都会向其中添加下一个元素。所以 res 也可认为是另一种累加器。

类似capitalize_all 这样的操作，一般叫做 **map**，因为此操作会将函数 (此处为 capitalize)"应用" 到序列中每个元素上。

另一种常用操作是，从列表中筛选部分元素，返回子列表。例如，下面函数会接收一个字符串列表，返回仅包含大写字母的字符串列表:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper 是字符串方法，如果字符串中字母全部为大写，则返回 True。

如only_upper 一样的操作，叫做 **filter**，因为此操作会筛选出某些元素，过滤掉其他元素。

大部分的列表操作都可以由 map，filter 以及 reduce 组合构成。

10.8 移除元素

从列表中移除元素有多种方法。如果你知道所要移除元素的索引，可以用 pop 方法:

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

pop 会改变列表，并返回被移除的元素。如果未提供索引，则会删除并返回最后一个元素。

如果你不需要移除元素，则可以用 del 元算符:

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

如果你知道要删除元素的值 (但不知道索引位置)，你可以用 remove 方法:

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

remove 方法的返回值是 None.

如果要移除多个元素，可以用 del 配合切片索引实现:

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

通常，切片含头不含尾，即到第二个索引值（不包括）为止。

10.9 列表和字符串

字符串是一系列的字符，而列表是一系列的值，但是字符列表不同于字符串。采用 list 方法，可以将字符串转换为字符列表:

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

因 list 是一个内置函数的名字，所以尽量避免将其作为变量名。我通常也不建议使用 l，因为和 1 太像了。这便是我用 t 作为变量名的缘由。

list 函数可以将字符串拆解为单独的字母。如果你想将字符串分拆为一个个单词，则可以使用 split 方法:

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

此处的可选参数是**分隔符**，也就是定义单词边界的字符。下面的示例使用连字符作为分隔符:

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

join 和 split 的作用正好相反。它会接收字符串列表，然后拼接所有元素。join 是一个字符串方法，所以你需要在分隔符上调用此方法，然后将列表作为参数进行操作:

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

此例中，分隔符是一个空格，join 会在每个单词之间放置一个空格。若想不用空格拼接，则可以用空字符串'' 作为分隔符.

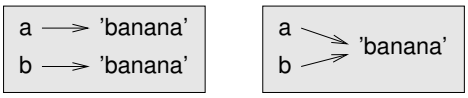

图 10.2: 状态图.

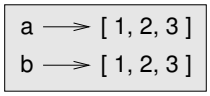

图 10.3: 状态图.

10.10 对象和值

如果运行下面的赋值语句:

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

我们知道 `a` 和 `b` 都指向了字符串，但我们不知道，他们是否指向了*相同的* 字符串。所以，便有了两种可能状态，如图 10.2.

第一种情况，`a` 和 `b` 指向了两个不同对象，这两个对象有相同的值。第二种情况，它们指向了同一个对象。

若想判断两个变量是否指向了同一个对象，可以用 `is` 运算符.

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

此例中，Python 仅创建了一个字符串对象，然后 `a` 和 `b` 都指向了此对象。但是当你创建两个列表时，你得到的就是两个对象:

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

此时，状态图便是图 10.3的样子.

这种情况下，可以说这两个列表是**相等的**, 因为它们有相同的元素，但它们不是**相同的**，因为不是同一个对象。两个对象如果相同，那么它们必然相等，但是它们如果相等，却未必相同。

到目前为止，我们一直混用“对象”和“值”, 但更准确地说，一个对象有一个值。若创建 `[1, 2, 3]`, 你会得到一个对象，而这个对象的值是个整数序列。如果另外一个列表也有相同的元素，我们便说它们拥有相同的值，但它们不是相同对象。

10.11 别称

如果 `a` 指向一个对象，同时令 `b = a`, 那么，两个变量都指向了同一个对象:

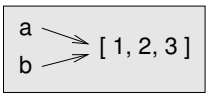

图 10.4: 状态图.

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

此时状态图如图 10.4所示.

变量和对象的这种关系，叫做**引用**. 上例中，两个引用指向了同一个对象。

如此，一个对象可以有不止一个引用，也就会有不止一个命名，所以，我们说，对象是有**别称的**.

如果一个别名对象是可变的，那么一个别名所做的修改会影响另一个:

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

虽然此特性很有用，但也很容易出错。通常，对于可变对象，避免使用别称，会安全很多。

而对于像字符串这样的不可变对象，别称使用往往不是问题。如下所示:

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

所以 `a` 和 `b` 是否指向同一个对象，已无关紧要。

10.12 列表参数

当给函数传递列表时，函数收到的是对该列表的引用。如果函数修改了列表，那么调用者也会观察到。例如，`delete_head` 函数删除了列表中第一个元素:

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

操作如下:

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

形参 `t` 和变量 `letters` 是同一个变量的别称。栈图如图 10.5所示.

因两个框共用一个列表，所以我把列表画在了它们中间。

区分修改列表操作和新建列表操作，是非常重要的。例如，`append` 方法是修改列表，而 `+` 运算符则会新建列表.

下面是 `append` 的一个使用示例:

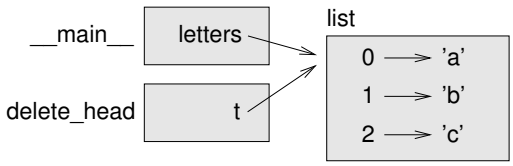

图 10.5: 栈图.

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

append 的返回值是 None.

下面是使用 + 运算符的示例:

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

操作结果是个新列表，但是原列表并没有改变.

在为修改列表而编写函数时，此差异尤要重视。例如，下面函数没有删除列表的第一个元素:

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

此切片运算符会新建列表，同时赋值号令 t 指向了此新列表，但是这个操作并不会影响调用者.

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

bad_delete_head 函数开始运行时，t 和 t4 指向了同一个列表。函数结束时，t 指向了新的列表，但是 t4 仍然指向了原来的列表，也就是没有修改原列表.

一种替代方案是，编写创建并返回新列表的函数。比如，tail 函数会返回列表中除首元素的所有元素列表:

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

此函数依然会保持原列表不变。下面是用法:

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

10.13 调试

若使用列表 (和其他可变对象) 时不够小心谨慎，很容易导致长达数小时的调试跟踪。这里有一些常见陷阱以及如何避免:

1. 多数的列表方法都是传入参数，返回 None. 这和字符串方法恰恰相反，字符串方法一般返回新字符串，同时保持原始值不变。

   如果你惯于编写下面这样的代码:

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

   那么你会写出下面这样的列表代码:

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

   因为 sort 返回的是 None, 所以后续对 t 的操作多会失败.

   在使用列表方法前，你应该认真阅读文档，并在交互模式下测试一下。

2. 确定原则，坚定执行.

   列表使用中面临的部分问题在于，有多条道路通罗马。比如，从列表中移除原色，你可以用 pop, remove, del, 甚至用切片重新赋值。

   若想给列表添加元素，可以用 append 方法，或者 + 运算符。假设 t 是列表，x 是列表元素，下面的操作都正确:

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

   而下面的代码错误:

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

   在交互模式下测试每个示例，确保理解其作用。你会注意到，只有最后一个会报运行时异常; 其他三个语句合法，但并不会得到想要的效果。

3. 复制列表，避免别称.

   你若想使用 sort 方法修改参数，同时又想保留原列表，那你可以复制一份。

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

   此例中，你也可以使用内置函数 sorted，此函数可以返回新的排好序的列表，同时又保留了原始列表。

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

10.14 术语表

列表 (list): 由值构成的序列。

元素 (element): 列表或序列中的值，也可以称为列表项.

嵌套列表 (nested list): 列表中的元素是其他列表。

累加器 (accumulator): 在循环中用于累加或累积结果的变量.

增强赋值语句 (augmented assignment): 使用 += 这样的操作符更新变量值的赋值语句.

reduce: 一种遍历序列，并累加元素为一个元素的处理模式.

map: 一种遍历序列，对每个元素都执行同一操作的处理模式.

filter: 一种遍历列表，并筛选满足特定条件的元素的处理模式.

对象 (object): 变量之指向. 对象拥有特定类型以及值.

相等 (equivalent): 拥有相同的值.

相同 (identical): 指向同一个对象 (也就意味着相等).

引用 (reference): 变量和其指向的值之间的关系.

别称 (aliasing): 多个变量指向同一个对象的情况.

分隔符 (delimiter): 用来界定长字符串分隔位置的字符和字符串.

10.15 习题集

大家可以从http://thinkpython2.com/code/list_exercises.py 下载这些习题的答案.

Exercise 10.1. 编写 nested_sum 函数，接收包含整数列表的一个列表，然后将所有内嵌列表的元素相加求和。比如:

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

Exercise 10.2. 编写函数 cumsum，使其接收一个数字列表，返回累加之和; 也就是，第 i 个元素是原列表中的第 $i+1$ 个元素之前所有元素之和。比如:

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

Exercise 10.3. 编写middle 函数，接收一个列表，返回包含掐头去尾后所有元素的新列表。比如:

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

Exercise 10.4. 编写函数 chop，使其接收一个列表，然后移除首尾元素，返回 None. 例如:

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

Exercise 10.5. 编写 is_sorted 函数，接收一个列表，如果列表是升序排列，则返回 True，否则返回 False. 例如:

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

Exercise 10.6. 如果一个单词，通过重新排列字母，得到另一单词，则称两个词为字母异位词。编写is_anagram 函数，接收两个字符串，如果两者为字母异位词，返回 True.

Exercise 10.7. 编写has_duplicates 函数，其以一个列表为输入，如果列表中存在重复出现的元素，返回 True. 注意不要修改原始列表.

Exercise 10.8. 此题属于生日悖论，你可以参考*http://en.wikipedia.org/wiki/Birthday_paradox* 了解更多.

如果你的班上有 23 名学生，那其中两人的生日相同的概率是多少？你可以生成 23 个随机的生日样本，检查其是否存在相同，从而估计概率值。提示：你可以通过 random 模块中的 randint 函数来制造随机生日。

可以从 *http://thinkpython2.com/code/birthday.py* 下载我的代码.

Exercise 10.9. 编写函数读取 words.txt 内容，并将其中每个单词放入列表中。写两个版本，一种用 append 方法，另一种用 t = t + [x] 实现。哪种耗时较长？为什么？

代码参见: *http://thinkpython2.com/code/wordlist.py*.

Exercise 10.10. 若要检验一个单词是否在列表中，可以用 in 运算符，但是速度会慢，因为它是每次都从头开始，顺序搜索。

由于这些单词通常是按照字母表顺序排列，我们可以通过对折查找(也叫二分查找)，提高速度。此方法和你通过字典(此处指书籍，不是数据结构)查找单词的方式很像。你一般会先翻到字典中间，然后看单词在之前，还是之后。如果在前面，则继续此方法查找，如果在后面，同样。

无论怎样，你都可以将搜索区间减半。如果单词列表有 *113,809* 个单词，大约 *17* 次，你就可以找到此单词，或者确定其不在其中.

编写函数in_bisect，使其接收一个有序列表，以及一个目标值，当单词在列表中时，返回 True，否则返回 False。

你也可以阅读 bisect 模块相关文档，进而使用它！可以参看代码: *http://thinkpython2.com/code/inlist.py*.

Exercise 10.11. 如果两个单词拼写顺序相反，则称其为“逆序对”. 编写代码，查找单词列表中所有逆序对. 参见代码: *http://thinkpython2.com/code/reverse_pair.py*.

Exercise 10.12. 如果从两个单词交替获取字母，构成新单词，便称其“连锁”。比如“*shoe*”和“*cold*”，交替获取字母，构成了“*schooled*”。代码样例: *http://thinkpython2.com/code/interlock.py*. 致谢: 此习题源于*http://puzzlers.org*的一个例子.

1. 编写函数，寻找所有连锁单词对。提示：不要枚举所有单词对!
2. 你能找到三路连锁的单词吗;也就是从三个单词，第一个，第二个，第三个，每次获取三个字母，从而构成新单词?

第11章　字典

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

11.1 A dictionary is a mapping

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

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

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

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

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

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

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

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

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

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

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

But if you print `eng2sp`, you might be surprised:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.2 Dictionary as a collection of counters

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

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

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

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

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

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

Here is what the code might look like:

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

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

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

Here's how it works:

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

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

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

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

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

11.3 Looping and dictionaries

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

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

Here's what the output looks like:

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

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

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

11.4 Reverse lookup

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

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search.

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

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

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

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

Here is an example of a successful reverse lookup:

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

And an unsuccessful one:

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

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

When you raise an exception, you can provide a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

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

11.5 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

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

Here is an example:

```
>>> hist = histogram('parrot')
>>> hist
```

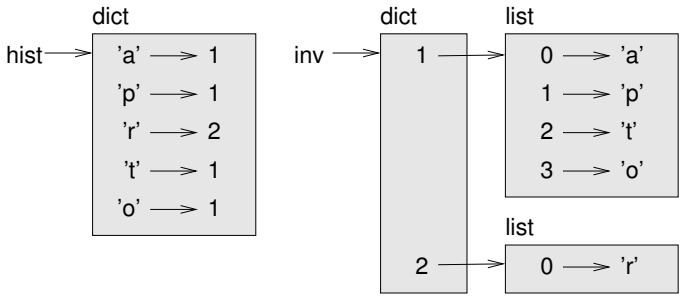

图 11.1: State diagram.

```
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

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

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

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

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

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

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

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

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

11.6 Memos

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

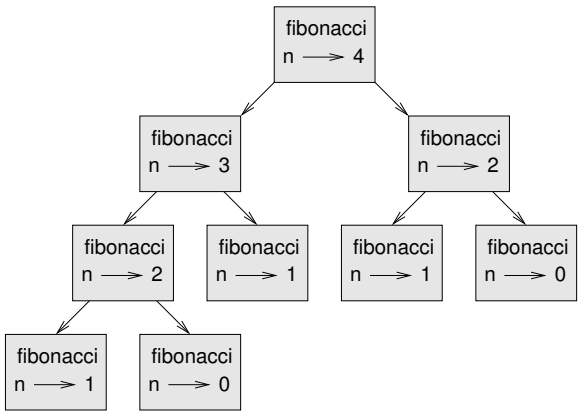

图 11.2: Call graph.

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

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

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

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

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

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

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

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

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

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

11.7 Global variables

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

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

```
verbose = True

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

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

```
been_called = False

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

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

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

```
been_called = False

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

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

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

```
count = 0

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

If you run it you get:

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

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

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

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

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

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

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

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

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

11.8 Debugging

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

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

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

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

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

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

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

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

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

11.9 Glossary

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

dictionary: A mapping from keys to their corresponding values.

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

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

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

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

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

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

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

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

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

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

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

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

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

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

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

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

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

11.10 Exercises

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

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

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

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

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

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

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

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

Exercise 11.6. *Here's another Puzzler from* Car Talk *(*`http://www.cartalk.com/content/puzzlers`*):*

> *This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?*
>
> *Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack.' W-R-A-C-K, you know like to 'wrack with pain.' If I remove the first letter, I am left with a four-letter word, 'R-A-C-K.' As in, 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' It's a perfect homophone. If you put the 'w' back, and remove the 'r,' instead, you're left with the word, 'wack,' which is a real word, it's just not a homophone of the other two words.*
>
> *But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?*

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

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

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

第 12 章　Tuples

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

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

12.1　Tuples are immutable

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.

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

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

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

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

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

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

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

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

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

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

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

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

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

And the slice operator selects a range of elements.

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

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

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

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

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

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

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).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 Tuple assignment

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`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

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

```
>>> 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.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> 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:

```
>>> 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`.

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3 Tuples as return values

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.

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:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Here is an example of a function that returns a tuple:

```
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.

12.4 Variable-length argument tuples

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:

```
def printall(*args):
    print(args)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
>>> 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:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
>>> divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

```
>>> max(1, 2, 3)
3
```

But `sum` does not.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

As an exercise, write a function called `sum_all` that takes any number of arguments and returns their sum.

12.5 Lists and tuples

`zip` is a built-in function that takes two or more sequences and interleaves them. The name of the function refers to a zipper, which interleaves two rows of teeth.

This example zips a string and a list:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

The result is a **zip object** that knows how to iterate through the pairs. The most common use of `zip` is in a `for` loop:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.

If you want to use list operators and methods, you can use a zip object to make a list:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
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:

```
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]`:

```
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`:

```
for index, element in enumerate('abc'):
    print(index, element)
```

The result from `enumerate` is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 a
1 b
2 c
```

Again.

12.6 Dictionaries and tuples

Dictionaries have a method called `items` that returns a sequence of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

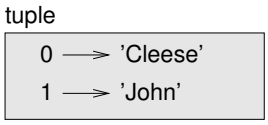

图 12.1: State diagram.

The result is a `dict_items` object, which is an iterator that iterates the key-value pairs. You can use it in a `for` loop like this:

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

As you should expect from a dictionary, the items are in no particular order.

Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

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:

```
directory[last, first] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
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.

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.

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.

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

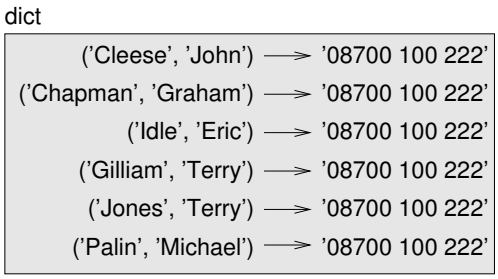

图 12.2: State diagram.

12.7 Sequences of sequences

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.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

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.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in function `sorted`, which takes any sequence and returns a new list with the same elements in sorted order, and `reversed`, which takes a sequence and returns an iterator that traverses the list in reverse order.

12.8 Debugging

Lists, dictionaries and tuples are examples of **data structures**; in this chapter we are starting to see compound data structures, like lists of tuples, or dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call **shape errors**; that is, errors caused when a data structure has the wrong type, size, or structure. For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

To help debug these kinds of errors, I have written a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as an argument and returns a string that summarizes its shape. You can download it from `http://thinkpython2.com/code/structshape.py`

Here's the result for a simple list:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

A fancier program might write "list of 3 ints", but it was easier not to deal with plurals. Here's a list of lists:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

If the elements of the list are not the same type, `structshape` groups them, in order, by type:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here's a list of tuples:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

And here's a dictionary with 3 items that map integers to strings.

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

If you are having trouble keeping track of your data structures, `structshape` can help.

12.9 Glossary

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: An operation that collects multiple arguments into a tuple.

scatter: An operation that makes a sequence behave like multiple arguments.

zip object: The result of calling a built-in function `zip`; an object that iterates through a sequence of tuples.

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.

12.10 Exercises

Exercise 12.1. *Write a function called* `most_frequent` *that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at* `http://en.wikipedia.org/wiki/Letter_frequencies`. *Solution:* `http://thinkpython2.com/code/most_frequent.py`.

Exercise 12.2. *More anagrams!*

1. *Write a program that reads a word list from a file (see Section 9.1) and prints all the sets of words that are anagrams.*

   Here is an example of what the output might look like:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Hint: you might want to build a dictionary that maps from a collection of letters to a list of words that can be spelled with those letters. The question is, how can you represent the collection of letters in a way that can be used as a key?

2. *Modify the previous program so that it prints the longest list of anagrams first, followed by the second longest, and so on.*

3. *In Scrabble a "bingo" is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What collection of 8 letters forms the most possible bingos?*

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

Exercise 12.3. *Two words form a "metathesis pair" if you can transform one into the other by swapping two letters; for example, "converse" and "conserve". Write a program that finds all of the metathesis pairs in the dictionary. Hint: don't test all pairs of words, and don't test all possible swaps. Solution:* `http://thinkpython2.com/code/metathesis.py`. *Credit: This exercise is inspired by an example at* `http://puzzlers.org`.

Exercise 12.4. *Here's another Car Talk Puzzler (*`http://www.cartalk.com/content/puzzlers`*):*

> *What is the longest English word, that remains a valid English word, as you remove its letters one at a time?*
>
> *Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?*

> *I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.*

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. *You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.*
2. *Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.*
3. *The wordlist I provided,* `words.txt`*, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.*
4. *To improve the performance of your program, you might want to memoize the words that are known to be reducible.*

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

第13章 Case study: data structure selection

At this point you have learned about Python's core data structures, and you have seen some of the algorithms that use them. If you would like to know more about algorithms, this might be a good time to read Chapter B. But you don't have to read it before you go on; you can read it whenever you are interested.

This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

13.1 Word frequency analysis

As usual, you should at least attempt the exercises before you read my solutions.
Exercise 13.1. *Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.*

Hint: The `string` *module provides a string named* `whitespace`*, which contains space, tab, newline, etc., and* `punctuation` *which contains the punctuation characters. Let's see if we can make Python swear:*

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Also, you might consider using the string methods `strip`, `replace` *and* `translate`.
Exercise 13.2. *Go to Project Gutenberg (*`http://gutenberg.org`*) and download your favorite out-of-copyright book in plain text format.*

Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.

Then modify the program to count the total number of words in the book, and the number of times each word is used.

Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?
Exercise 13.3. *Modify the program from the previous exercise to print the 20 most frequently used words in the book.*

Exercise 13.4. *Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that* should *be in the word list, and how many of them are really obscure?*

13.2 Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be **deterministic**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

The function `randint` takes parameters `low` and `high` and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Exercise 13.5. *Write a function named* `choose_from_hist` *that takes a histogram as defined in Section 11.2 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

your function should return `'a'` *with probability* 2/3 *and* `'b'` *with probability* 1/3.

13.3 Word histogram

You should attempt the previous exercises before you go on. You can download my solution from `http://thinkpython2.com/code/analyze_book1.py`. You will also need `http://thinkpython2.com/code/emma.txt`.

Here is a program that reads a file and builds a histogram of the words in the file:

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

This program reads `emma.txt`, which contains the text of *Emma* by Jane Austen.

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `hist` is being used as an accumulator.

`process_line` uses the string method `replace` to replace hyphens with spaces before using `split` to break the line into a list of strings. It traverses the list of words and uses `strip` and `lower` to remove punctuation and convert to lower case. (It is a shorthand to say that strings are "converted"; remember that strings are immutable, so methods like `strip` and `lower` return new strings.)

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```
def total_words(hist):
    return sum(hist.values())
```

The number of different words is just the number of items in the dictionary:

```
def different_words(hist):
    return len(hist)
```

Here is some code to print the results:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

And the results:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Most common words

To find the most common words, we can make a list of tuples, where each tuple contains a word and its frequency, and sort it.

The following function takes a histogram and returns a list of word-frequency tuples:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the ten most common words:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

I use the keyword argument `sep` to tell `print` to use a tab character as a "separator", rather than a space, so the second column is lined up. Here are the results from *Emma*:

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

This code can be simplified using the `key` parameter of the `sort` function. If you are curious, you can read about it at `https://wiki.python.org/moin/HowTo/Sorting`.

13.5 Optional parameters

We have seen built-in functions and methods that take optional arguments. It is possible to write programmer-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The **default value** of `num` is 10.

If you only provide one argument:

```
print_most_common(hist)
```

`num` gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

`num` gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

13.6 Dictionary subtraction

Finding the words from the book that are not in the word list from `words.txt` is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

`subtract` takes dictionaries `d1` and `d2` and returns a new dictionary that contains all the keys from `d1` that are not in `d2`. Since we don't really care about the values, we set them all to None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

To find the words in the book that are not in `words.txt`, we can use `process_file` to build a histogram for `words.txt`, and then subtract:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

Here are some of the results from *Emma*:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Some of these words are names and possessives. Others, like "rencontre", are no longer in common use. But a few are common words that should really be in the list!

Exercise 13.6. *Python provides a data structure called* `set` *that provides many common set operations. You can read about them in Section 19.5, or read the documentation at* `http://docs.python.org/3/library/stdtypes.html#types-set`.

Write a program that uses set subtraction to find words in the book that are not in the word list. Solution: `http://thinkpython2.com/code/analyze_book2.py`.

13.7 Random words

To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

The expression `[word] * freq` creates a list with `freq` copies of the string `word`. The `extend` method is similar to `append` except that the argument is a sequence.

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

An alternative is:

1. Use `keys` to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see Exercise 10.2). The last item in this list is the total number of words in the book, n.
3. Choose a random number from 1 to n. Use a bisection search (See Exercise 10.10) to find the index where the random number would be inserted in the cumulative sum.
4. Use the index to find the corresponding word in the word list.

Exercise 13.7. *Write a program that uses this algorithm to choose a random word from the book. Solution:* `http://thinkpython2.com/code/analyze_book3.py`.

13.8 Markov analysis

If you choose words from the book at random, you can get a sense of the vocabulary, but you probably won't get a sentence:

```
this the small regard harriet which knightley's it most things
```

A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.

One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the words that might come next. For example, the song *Eric, the Half a Bee* begins:

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

In this text, the phrase "half the" is always followed by the word "bee", but the phrase "the bee" might be followed by either "has" or "is".

The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").

Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

For example, if you start with the prefix "Half a", then the next word has to be "bee", because the prefix only appears once in the text. The next prefix is "a bee", so the next suffix might be "philosophically", "be" or "due".

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

Exercise 13.8. *Markov analysis:*

1. *Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length two, but you should write the program in a way that makes it easy to try other lengths.*

2. *Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from* Emma *with prefix length 2:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.*

   For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

   What happens if you increase the prefix length? Does the random text make more sense?

3. *Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.*

Credit: This case study is based on an example from Kernighan and Pike, The Practice of Programming, *Addison-Wesley, 1999.*

You should attempt this exercise before you go on; then you can download my solution from `http://thinkpython2.com/code/markov.py`. You will also need `http://thinkpython2.com/code/emma.txt`.

13.9 Data structures

Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:

- How to represent the prefixes.
- How to represent the collection of possible suffixes.
- How to represent the mapping from each prefix to the collection of possible suffixes.

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is "Half a", and the next word is "bee", you need to be able to form the next prefix, "a bee".

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can't append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` takes a tuple of words, `prefix`, and a string, `word`, and forms a new tuple that has all the words in `prefix` except the first, and `word` added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see Exercise 13.7).

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is run time. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the `in` operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don't know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is no need to go on. If not, there are tools, like the `profile` module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no

matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after run time.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

13.10 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are five things to try:

Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.

Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.

Ruminating: Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

Rubberducking: If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up; see `https://en.wikipedia.org/wiki/Rubber_duck_debugging`.

Retreating: At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

13.11 Glossary

deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom: Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

default value: The value given to an optional parameter if no argument is provided.

override: To replace a default value with an argument.

benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

rubber duck debugging: Debugging by explaining your problem to an inanimate object such as a rubber duck. Articulating the problem can help you solve it, even if the rubber duck doesn't know Python.

13.12 Exercises

Exercise 13.9. *The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.*

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages (`http://en.wikipedia.org/wiki/Zipf's_law`). Specifically, it predicts that the frequency, f, of the word with rank r is:

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with $\log f$ *and* $\log r$*. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of* s*?*

Solution: `http://thinkpython2.com/code/zipf.py`*. To run my solution, you need the plotting module* `matplotlib`*. If you installed Anaconda, you already have* `matplotlib`*; otherwise you might have to install it.*

第 14 章　Files

This chapter introduces the idea of "persistent" programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2 Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file. The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

14.3 Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value `42`.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

For more information on the format operator, see `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`. A more powerful alternative is the string format method, which you can read about at `https://docs.python.org/3/library/stdtypes.html#str.format`.

14.4 Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` stands for "current working directory". The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.

A string like `'/home/dinsdale'` that identifies a file or directory is called a **path**.

A simple filename, like `memo.txt` is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is `/home/dinsdale`, the filename `memo.txt` would refer to `/home/dinsdale/memo.txt`.

A path that begins with `/` does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` provides other functions for working with filenames and paths. For example, `os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

The `os` module provides a function called `walk` that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from `http://thinkpython2.com/code/walk.py`.

14.5 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does. The syntax is similar to an `if...else` statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

14.6 Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like `keys` and `items`, don't work with database objects. But iteration with a `for` loop works:

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7 Pickling

A limitation of `dbm` is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The `pickle` module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

`pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for `pickle` to interpret. `pickle.loads` ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

14.8 Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen`[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

[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.

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a "checksum". You can read about MD5 at `http://en.wikipedia.org/wiki/Md5`. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

14.10 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at `http://en.wikipedia.org/wiki/Newline`. Or, of course, you could write one yourself.

14.11 Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

14.12 Exercises

Exercise 14.1. *Write a function called* `sed` *that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.*

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution: `http://thinkpython2.com/code/sed.py`.

Exercise 14.2. *If you download my solution to Exercise 12.2 from* `http://thinkpython2.com/code/anagram_sets.py`*, you'll see that it creates a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example,* `'opst'` *maps to the list* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Write a module that imports `anagram_sets` *and provides two new functions:* `store_anagrams` *should store the anagram dictionary in a "shelf";* `read_anagrams` *should look up a word and return a list of its anagrams. Solution:* `http://thinkpython2.com/code/anagram_db.py`.

Exercise 14.3. *In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.*

1. *Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like* `.mp3`*). Hint:* `os.path` *provides several useful functions for manipulating file and path names.*

2. *To recognize duplicates, you can use* `md5sum` *to compute a "checksum" for each files. If two files have the same checksum, they probably have the same contents.*

3. *To double-check, you can use the Unix command* `diff`.

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

第15章　Classes and objects

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn "object-oriented programming", which uses programmer-defined types to organize both code and data. Object-oriented programming is a big topic; it will take a few chapters to get there.

Code examples from this chapter are available from `http://thinkpython2.com/code/Point1.py`; solutions to the exercises are available from `http://thinkpython2.com/code/Point1_soln.py`.

15.1 Programmer-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this:

```
class Point:
    """Represents a point in 2-D space."""
```

The header indicates that the new class is called `Point`. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a **class object**.

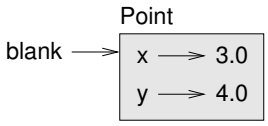

图 15.1: Object diagram.

```
>>> Point
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

Every object is an instance of some class, so "object" and "instance" are interchangeable. But in this chapter I use "instance" to indicate that I am talking about a programmer-defined type.

15.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

Figure 15.1 is a state diagram that shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**.

The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In the example, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

As an exercise, write a function called `distance_between_points` that takes two Points as arguments and returns the distance between them.

15.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

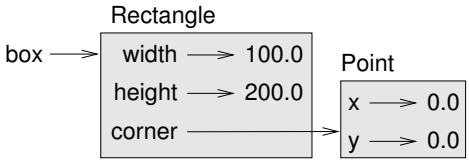

图 15.2: Object diagram.

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Here is an example that passes `box` as an argument and assigns the resulting Point to `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, `rect` is an alias for `box`, so when the function modifies `rect`, `box` changes.

As an exercise, write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.

15.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

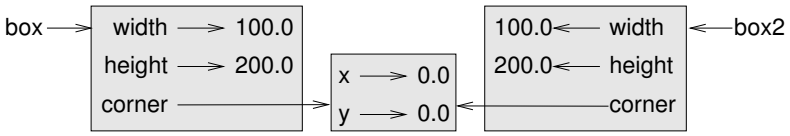

图 15.3: Object diagram.

```
>>> 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 provides a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

As an exercise, write a version of `move_rectangle` that creates and returns a new Rectangle instead of modifying the old one.

15.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<class '__main__.Point'>
```

You can also use `isinstance` to check whether an object is an instance of a class:

```
>>> isinstance(p, Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

You can also use a `try` statement to see if the object has the attributes you need:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

15.8 Glossary

class: A programmer-defined type. A class definition creates a new class object.

class object: An object that contains information about a programmer-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

instantiate: To create a new object.

attribute: One of the named values associated with an object.

embedded object: An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

15.9 Exercises

Exercise 15.1. *Write a definition for a class named* `Circle` *with attributes* `center` *and* `radius`*, where* `center` *is a Point object and* `radius` *is a number.*

Instantiate a Circle object that represents a circle with its center at $(150, 100)$ *and radius 75.*

Write a function named `point_in_circle` *that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.*

Write a function named `rect_in_circle` *that takes a Circle and a Rectangle and returns True if the Rectangle lies entirely in or on the boundary of the circle.*

Write a function named `rect_circle_overlap` *that takes a Circle and a Rectangle and returns True if any of the corners of the Rectangle fall inside the Circle. Or as a more challenging version, return True if any part of the Rectangle falls inside the Circle.*

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

Exercise 15.2. *Write a function called* `draw_rect` *that takes a Turtle object and a Rectangle and uses the Turtle to draw the Rectangle. See Chapter 4 for examples using Turtle objects.*

Write a function called `draw_circle` *that takes a Turtle and a Circle and draws the Circle.*

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

第16章 Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present "functional programming style" and two new program development plans.

Code examples from this chapter are available from `http://thinkpython2.com/code/Time1.py`. Solutions to the exercises are at `http://thinkpython2.com/code/Time1_soln.py`.

16.1 Time

As another example of a programmer-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

The state diagram for the `Time` object looks like Figure 16.1.

As an exercise, write a function called `print_time` that takes a Time object and prints it in the form `hour:minute:second`. Hint: the format sequence `'%.2d'` prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called `is_after` that takes two Time objects, `t1` and `t2`, and returns `True` if `t1` follows `t2` chronologically and `False` otherwise. Challenge: don't use an `if` statement.

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 develop-

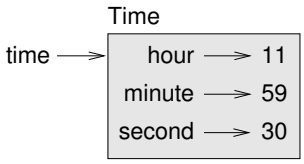

图 16.1: Object diagram.

ment plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: `start` contains the start time of a movie, like *Monty Python and the Holy Grail*, and `duration` contains the run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, `10:80:00` might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
```

```
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

16.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if `seconds` is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of `increment` that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

As an exercise, write a "pure" version of `increment` that creates and returns a new Time object rather than modifying the parameter.

16.4 Prototyping versus planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see `http://en.wikipedia.org/wiki/Sexagesimal`). The `second` attribute is the "ones column", the `minute` attribute is the "sixties column", and the `hour` attribute is the "thirty-six hundreds column".

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that `time_to_int(int_to_time(x)) == x` for many values of `x`. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite `increment` using `time_to_int` and `int_to_time`.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

16.5 Debugging

A Time object is well-formed if the values of `minute` and `second` are between 0 and 60 (including 0 but not 60) and if `hour` is positive. `hour` and `minute` should be integer values, but we might allow `second` to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like `valid_time` that takes a Time object and returns `False` if it violates an invariant:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an **assert statement**, which checks a given invariant and raises an exception if it fails:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

16.6 Glossary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return `None`.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

assert statement: A statement that check a condition and raises an exception if it fails.

16.7 Exercises

Code examples from this chapter are available from `http://thinkpython2.com/code/Time1.py`; solutions to the exercises are available from `http://thinkpython2.com/code/Time1_soln.py`.

Exercise 16.1. *Write a function called* `mul_time` *that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.*

Then use `mul_time` *to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).*

Exercise 16.2. *The* `datetime` *module provides* `time` *objects that are similar to the Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at* `http://docs.python.org/3/library/datetime.html`.

1. *Use the* `datetime` *module to write a program that gets the current date and prints the day of the week.*

2. *Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.*

3. *For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birth dates and computes their Double Day.*

4. *For a little more challenge, write the more general version that computes the day when one person is n times older than the other.*

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

第17章　Classes and methods

Although we are using some of Python's object-oriented features, the programs from the last two chapters are not really object-oriented because they don't represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

Code examples from this chapter are available from `http://thinkpython2.com/code/Time2.py`, and solutions to the exercises are in `http://thinkpython2.com/code/Point2_soln.py`.

17.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

For example, the `Time` class defined in Chapter 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 in Chapter 15 correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in `Time1.py` there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for **methods**; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

17.2 Printing objects

In Chapter 16, we defined a class named `Time` and in Section 16.1, you wrote a function named `print_time`:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, "Hey `print_time`! Here's an object for you to print."
- In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says "Hey `start`! Please print yourself."

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite `time_to_int` (from Section 16.4) as a method. You might be tempted to rewrite `int_to_time` as a method, too, but that doesn't really make sense because there would be no object to invoke it on.

17.3 Another example

Here's a version of `increment` (from Section 16.3) rewritten as a method:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

`parrot` and `cage` are positional, and `dead` is a keyword argument.

17.4 A more complicated example

Rewriting `is_after` (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter `self` and the second parameter `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

17.5 The init method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An init method for the `Time` class might look like this:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
        self.hour = hour
```

stores the value of the parameter `hour` as an attribute of `self`.

The parameters are optional, so if you call `Time` with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

As an exercise, write an init method for the Point class that takes x and y as optional parameters and assigns them to the corresponding attributes.

17.6 The __str__ method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

For example, here is a str method for Time objects:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

As an exercise, write a str method for the Point class. Create a Point object and print it.

17.7 Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

Here is what the definition might look like:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

When you apply the + operator to Time objects, Python invokes `__add__`. When you print the result, Python invokes `__str__`. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see `http://docs.python.org/3/reference/datamodel.html#specialnames`.

As an exercise, write an `add` method for the Point class.

17.8 Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of `__add__` that checks the type of `other` and invokes either `add_time` or `increment`:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function `isinstance` takes a value and a class object, and returns `True` if the value is an instance of the class.

If `other` is a Time object, `__add__` invokes `add_time`. Otherwise it assumes that the parameter is a number and invokes `increment`. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method `__radd__`, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

As an exercise, write an `add` method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

17.9 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used `histogram` to count the number of times each letter appears in a word.

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

This function also works for lists, tuples, and even dictionaries, as long as the elements of `s` are hashable, so they can be used as keys in `d`.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function `sum`, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an `add` method, they work with `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

17.10 Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 15.7).

Another way to access attributes is the built-in function `vars`, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

17.11 Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include `time_to_int`, `is_after`, and `add_time`.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a `Time` object are `hour`, `minute`, and `second`.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like `is_after`, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

17.12 Glossary

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

operator overloading: Changing the behavior of an operator like + so it works with a programmer-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

17.13 Exercises

Exercise 17.1. *Download the code from this chapter from* `http://thinkpython2.com/code/Time2.py`*. Change the attributes of* `Time` *to be a single integer representing seconds since midnight. Then modify the methods (and the function* `int_to_time`*) to work with the new implementation. You should not have to modify the test code in* `main`*. When you are done, the output should be the same as before. Solution:* `http://thinkpython2.com/code/Time2_soln.py`*.*

Exercise 17.2. *This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named* `Kangaroo` *with the following methods:*

1. *An* `__init__` *method that initializes an attribute named* `pouch_contents` *to an empty list.*
2. *A method named* `put_in_pouch` *that takes an object of any type and adds it to* `pouch_contents`*.*
3. *A* `__str__` *method that returns a string representation of the Kangaroo object and the contents of the pouch.*

Test your code by creating two `Kangaroo` *objects, assigning them to variables named* `kanga` *and* `roo`*, and then adding* `roo` *to the contents of* `kanga`*'s pouch.*

Download `http://thinkpython2.com/code/BadKangaroo.py`*. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.*

If you get stuck, you can download `http://thinkpython2.com/code/GoodKangaroo.py`*, which explains the problem and demonstrates a solution.*

第 18 章　Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

If you don't play poker, you can read about it at `http://en.wikipedia.org/wiki/Poker`, but you don't have to; I'll tell you what you need to know for the exercises.

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

18.1 Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like `'Spade'` for suits and `'Queen'` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

I am using the $\mapsto$ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for `Card` looks like this:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

18.2 Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to **class attributes**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance.

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`.

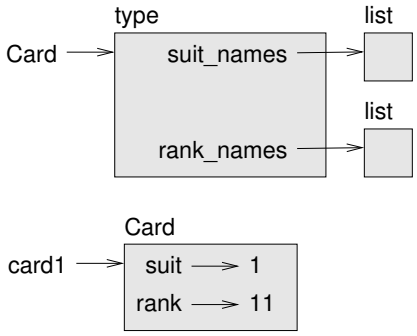

图 18.1: Object diagram.

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; its type is `type`. `card1` is an instance of `Card`, so its type is `Card`. To save space, I didn't draw the contents of `suit_names` and `rank_names`.

18.3 Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for "less than".

`__lt__` takes two parameters, `self` and `other`, and returns `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
# inside class Card:

    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

As an exercise, write an `__lt__` method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

18.4 Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for `Deck`. The init method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to `self.cards`.

18.5 Printing the deck

Here is a `__str__` method for `Deck`:

```
# inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
```

```
            res.append(str(card))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

18.6 Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that:

```
# inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Since `pop` removes the *last* card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method `append`:

```
# inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a **veneer**. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case `add_card` is a "thin" method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named `shuffle` using the function `shuffle` from the `random` module:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import `random`.

As an exercise, write a Deck method named `sort` that uses the list method `sort` to sort the cards in a `Deck`. `sort` uses the `__lt__` method we defined to determine the order.

18.7 Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

In this example, `Hand` inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize `cards` with an empty list.

If we provide an init method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

When you create a Hand, Python invokes this init method, not the one in `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

The other methods are inherited from `Deck`, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
# inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a `Deck`.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8 Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in, "a Rectangle has a Point."
- One class might inherit from another. This relationship is called **IS-A**, as in, "a Hand is a kind of a Deck."
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

A **class diagram** is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between `Card`, `Deck` and `Hand`.

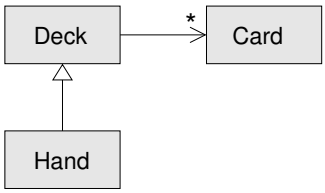

图 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (`*`) near the arrow head is a **multiplicity**; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like `52`, a range, like `5..7` or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a *list* of Cards, but built-in types like list and dict are usually not included in class diagrams.

18.9 Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like `shuffle`, you might get the one defined in `Deck`, but if any of the subclasses override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Here's an example:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class '__main__.Deck'>
```

So the `shuffle` method for this Hand is the one in `Deck`.

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order", which is the sequence of classes Python searches to "resolve" a method name.

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.

If you violate this rule, which is called the "Liskov substitution principle", your code will collapse like (sorry) a house of cards.

18.10 Data encapsulation

The previous chapters demonstrate a development plan we might call "object-oriented design". We identified objects we needed—like `Point`, `Rectangle` and `Time`—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from Section 13.8, provides a good example. If you download my code from `http://thinkpython2.com/code/markov.py`, you'll see that it uses two global variables—`suffix_map` and `prefix`—that are read and written from several functions.

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here's `process_word`:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return
```

```
        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from `http://thinkpython2.com/code/markov.py`, and follow the steps described above to encapsulate the global variables as attributes of a new class called `Markov`. Solution: `http://thinkpython2.com/code/markov2.py`.

18.11 Glossary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

instance attribute: An attribute associated with an instance of a class.

veneer: A method or function that provides a different interface to another function without doing much computation.

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass".

IS-A relationship: A relationship between a child class and its parent class.

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

class diagram: A diagram that shows the classes in a program and the relationships between them.

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.

18.12 Exercises

Exercise 18.1. *For the following program, draw a UML class diagram that shows these classes and the relationships among them.*

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Exercise 18.2. *Write a Deck method called* `deal_hands` *that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.*

Exercise 18.3. *The following are the possible hands in poker, in increasing order of value and decreasing order of probability:*

pair: *two cards with the same rank*

two pair: *two pairs of cards with the same rank*

three of a kind: *three cards with the same rank*

straight: *five cards with ranks in sequence (aces can be high or low, so* `Ace-2-3-4-5` *is a straight and so is* `10-Jack-Queen-King-Ace`*, but* `Queen-King-Ace-2-3` *is not.)*

flush: *five cards with the same suit*

full house: *three cards with one rank, two cards with another*

four of a kind: *four cards with the same rank*

straight flush: *five cards in sequence (as defined above) and with the same suit*

The goal of these exercises is to estimate the probability of drawing these various hands.

1. *Download the following files from* `http://thinkpython2.com/code`:

   `Card.py` *: A complete version of the* `Card`, `Deck` *and* `Hand` *classes in this chapter.*

   `PokerHand.py` *: An incomplete implementation of a class that represents a poker hand, and some code that tests it.*

2. *If you run* `PokerHand.py`*, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.*

3. *Add methods to* `PokerHand.py` *named* `has_pair`, `has_twopair`*, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for "hands" that contain any number of cards (although 5 and 7 are the most common sizes).*

4. *Write a method named* `classify` *that figures out the highest-value classification for a hand and sets the* `label` *attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled "flush".*

5. *When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in* `PokerHand.py` *that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.*

6. *Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at* `http://en.wikipedia.org/wiki/Hand_rankings`.

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

第 19 章　利器

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

19.1 Conditional expressions

We saw conditional statements in Section 5.4. Conditional statements are often used to choose one of two values; for example:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

This statement checks whether `x` is positive. If so, it computes `math.log`. If not, `math.log` would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a **conditional expression**:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: "`y` gets log-`x` if `x` is greater than 0; otherwise it gets NaN".

Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of `factorial`:

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

We can rewrite it like this:

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

Another use of conditional expressions is handling optional arguments. For example, here is the init method from `GoodKangaroo` (see Exercise 17.2):

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

We can rewrite this one like this:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

19.2 List comprehensions

In Section 10.7 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method `capitalize` to the elements, and returns a new list of strings:

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

We can write this more concisely using a **list comprehension**:

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

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the `for` clause indicates what sequence we are traversing.

The syntax of a list comprehension is a little awkward because the loop variable, `s` in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of `t` that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

19.3 Generator expressions

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function `next` gets the next value from the generator:

```
>>> next(g)
0
>>> next(g)
1
```

When you get to the end of the sequence, `next` raises a StopIteration exception. You can also use a `for` loop to iterate through the values:

```
>>> for val in g:
...     print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the `for` loop picks up where `next` left off. Once the generator is exhausted, it continues to raise `StopIteration`:

```
>>> next(g)
StopIteration
```

Generator expressions are often used with functions like `sum`, `max`, and `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 `any` and `all`

Python provides a built-in function, `any`, that takes a sequence of boolean values and returns `True` if any of the values are `True`. It works on lists:

```
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the `in` operator. But we could use `any` to rewrite some of the search functions we wrote in Section 9.3. For example, we could write `avoids` like this:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, "`word` avoids `forbidden` if there are not any forbidden letters in `word`."

Using `any` with a generator expression is efficient because it stops immediately if it finds a `True` value, so it doesn't have to evaluate the whole sequence.

Python provides another built-in function, `all`, that returns `True` if every element of the sequence is `True`. As an exercise, use `all` to re-write `uses_all` from Section 9.3.

19.5 Sets

In Section 13.6 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes `d1`, which contains the words from the document as keys, and `d2`, which contains the list of words. It returns a dictionary that contains the keys from `d1` that are not in `d2`.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

In all of these dictionaries, the values are `None` because we never use them. As a result, we waste some storage space.

Python provides another built-in type, called a `set`, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

For example, set subtraction is available as a method called `difference` or as an operator, `-`. So we can rewrite `subtract` like this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to `has_duplicates`, from Exercise 10.7, that uses a dictionary:

```
def has_duplicates(t):
    d = {}
    for x in t:
```

```
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns `True`.

Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in `t` appears more than once, the set will be smaller than `t`. If there are no duplicates, the set will be the same size as `t`.

We can also use sets to do some of the exercises in Chapter 9. For example, here's a version of `uses_only` with a loop:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` checks whether all letters in `word` are in `available`. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The `<=` operator checks whether one set is a subset of another, including the possibility that they are equal, which is true if all the letters in `word` appear in `available`.

As an exercise, rewrite `avoids` using sets.

19.6 Counters

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a **multiset**, a Counter is a natural way to represent a multiset.

Counter is defined in a standard module called `collections`, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

```
>>> count['d']
0
```

We can use Counters to rewrite `is_anagram` from Exercise 10.6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent.

Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, `most_common`, which returns a list of value-frequency pairs, sorted from most common to least:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict

The `collections` module also provides `defaultdict`, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a **factory**. The built-in functions that create lists, sets, and other types can be used as factories:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notice that the argument is `list`, which is a class object, not `list()`, which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

```
>>> t = d['new key']
>>> t
[]
```

The new list, which we're calling `t`, is also added to the dictionary. So if we modify `t`, the change appears in `d`:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using `defaultdict`. In my solution to Exercise 12.2, which you can get from `http://thinkpython2.com/code/anagram_sets.py`, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, `'opst'` maps to the list `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Here's the original code:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
```

```
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

This can be simplified using `setdefault`, which you might have used in Exercise 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

We can avoid this problem and simplify the code using a `defaultdict`:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

My solution to Exercise 18.3, which you can download from `http://thinkpython2.com/code/PokerHandSoln.py`, uses `setdefault` in the function `has_straightflush`. This solution has the drawback of creating a `Hand` object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a defaultdict.

19.8 Named tuples

Many simple objects are basically collections of related values. For example, the Point object defined in Chapter 15 contains two numbers, `x` and `y`. When you define a class like this, you usually start with an init method and a str method:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from `namedtuple` is a class object:

```
>>> Point
<class '__main__.Point'>
```

`Point` automatically provides methods like `__init__` and `__str__` so you don't have to write them.

To create a Point object, you use the Point class as a function:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes.

You can access the elements of the named tuple by name:

```
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

19.9 Gathering keyword args

In Section 12.4, we saw how to write a function that gathers its arguments into a tuple:

```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but `kwargs` is a common choice. The result is a dictionary that maps from keywords to values:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat `d` as a single positional argument, so it would assign `d` to `x` and complain because there's nothing to assign to `y`:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

19.10 Glossary

conditional expression: An expression that has one of two values, depending on a condition.

list comprehension: An expression with a `for` loop in square brackets that yields a new list.

generator expression: An expression with a `for` loop in parentheses that yields a generator object.

multiset: A mathematical entity that represents a mapping between the elements of a set and the number of times they appear.

factory: A function, usually passed as a parameter, used to create objects.

19.11 Exercises

Exercise 19.1. *The following is a function that computes the binomial coefficient recursively.*

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Rewrite the body of the function using nested conditional expressions.

One note: this function is not very efficient because it ends up computing the same values over and over. You could make it more efficient by memoizing (see Section 11.6). But you will find that it's harder to memoize if you write it using conditional expressions.

第 A 章　Debugging

When you are debugging, you should distinguish among different kinds of errors in order to track them down more quickly:

- Syntax errors are discovered by the interpreter when it is translating the source code into byte code. They indicate that there is something wrong with the structure of the program. Example: Omitting the colon at the end of a `def` statement generates the somewhat redundant message `SyntaxError: invalid syntax`.

- Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error "maximum recursion depth exceeded".

- Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

A.1　Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.
3. Make sure that any strings in the code have matching quotation marks. Make sure that all quotation marks are "straight quotes", not "curly quotes".
4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an `invalid token` error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
5. An unclosed opening operator—`(`, `{`, or `[`—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
6. Check for the classic = instead of == inside a conditional.
7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.
8. If you have non-ASCII characters in the code (including strings and comments), that might cause a problem, although Python 3 usually handles non-ASCII characters. Be careful if you paste in text from a web page or other source.

If nothing works, move on to the next section...

A.1.1 I keep making changes and it makes no difference.

If the interpreter says there is an error and you don't see it, that might be because you and the interpreter are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run.

If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run it again. If the interpreter doesn't find the new error, you are not running the new code.

There are a few likely culprits:

- You edited the file and forgot to save the changes before running it again. Some programming environments do this for you, but some don't.
- You changed the name of the file, but you are still running the old name.

- Something in your development environment is configured incorrectly.
- If you are writing a module and using `import`, make sure you don't give your module the same name as one of the standard Python modules.
- If you are using `import` to read a module, remember that you have to restart the interpreter or use `reload` to read a modified file. If you import the module again, it doesn't do anything.

If you get stuck and you can't figure out what is going on, one approach is to start again with a new program like "Hello, World!", and make sure you can get a known program to run. Then gradually add the pieces of the original program to the new one.

A.2 Runtime errors

Once your program is syntactically correct, Python can read it and at least start running it. What could possibly go wrong?

A.2.1 My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke a function to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure there is a function call in the program, and make sure the flow of execution reaches it (see "Flow of Execution" below).

A.2.2 My program hangs.

If a program stops and seems to be doing nothing, it is "hanging". Often that means that it is caught in an infinite loop or infinite recursion.

- If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop".

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

  If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

- If neither of those steps works, start testing other loops and other recursive functions and methods.
- If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `False`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

Infinite Recursion

Most of the time, infinite recursion causes the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function is causing an infinite recursion, make sure that there is a base case. There should be some condition that causes the function to return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function that prints the parameters. Now when you run the program, you will see a few lines of output every time the function is invoked, and you will see the parameter values. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

If you are not sure how the flow of execution is moving through your program, add `print` statements to the beginning of each function with a message like "entering function `foo`", where `foo` is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

A.2.3 When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

The traceback identifies the function that is currently running, and then the function that called it, and then the function that called *that*, and so on. In other words, it traces the

sequence of function calls that got you to where you are, including the line number in your file where each call occurred.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError: You are trying to use a variable that doesn't exist in the current environment. Check if the name is spelled right, or at least consistently. And remember that local variables are local; you cannot refer to them from outside the function where they are defined.

TypeError: There are several possible causes:

- You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
- There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
- You are passing the wrong number of arguments to a function. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError: You are trying to access an element of a dictionary using a key that the dictionary does not contain. If the keys are strings, remember that capitalization matters.

AttributeError: You are trying to access an attribute or method that does not exist. Check the spelling! You can use the built-in function `vars` to list the attributes that do exist.

If an AttributeError indicates that an object has `NoneType`, that means that it is `None`. So the problem is not the attribute name, but the object.

The reason the object is none might be that you forgot to return a value from a function; if you get to the end of a function without hitting a `return` statement, it returns `None`. Another common cause is using the result from a list method, like `sort`, that returns `None`.

IndexError: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

The Python debugger (`pdb`) is useful for tracking down exceptions because it allows you to examine the state of the program immediately before the error. You can read about `pdb` at `https://docs.python.org/3/library/pdb.html`.

A.2.4 I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are searching a list, search a *small* list. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think shouldn't affect the program, and it does, that can tip you off.

A.3 Semantic errors

In some ways, semantic errors are the hardest to debug, because the interpreter provides no information about what is wrong. Only you know what the program is supposed to do.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "stepping" the program to where the error is occurring.

A.3.1 My program doesn't work.

You should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves functions or methods in other Python modules. Read the documentation for the functions you call. Try them out by writing simple test cases and checking the results.

In order to program, you need a mental model of how programs work. If you write a program that doesn't do what you expect, often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

A.3.2 I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression $\frac{x}{2\pi}$ into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes $x\pi/2$.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the order of operations.

A.3.3 I've got a function that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the result before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].removeMatches()
```

you could write:

```
count = self.hands[i].removeMatches()
return count
```

Now you have the opportunity to display the value of `count` before returning.

A.3.4 I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these symptoms:

- Frustration and rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

A.3.5 No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. You need a fresh pair of eyes.

Before you bring someone else in, make sure you are prepared. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have `print` statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

第B章　Analysis of Algorithms

This appendix is an edited excerpt from *Think Complexity*, by Allen B. Downey, also published by O'Reilly Media (2012). When you are done with this book, you might want to move on to that one.

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`.

The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions.

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`.

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].

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

- 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.
- 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.

[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."

- Relative performance also depends on the size of the problem. A sorting algorithm that is fast for small lists might be slow for long lists. The usual solution to this problem is to express run time (or number of operations) as a function of problem size, and group functions into categories depending on how quickly they grow as problem size increases.

The good thing about this kind of comparison is that it lends itself to simple classification of algorithms. For example, if I know that the run time of Algorithm A tends to be proportional to the size of the input, n, and Algorithm B tends to be proportional to n^2, then I expect A to be faster than B, at least for large values of n.

This kind of analysis comes with some caveats, but we'll get to that later.

B.1 Order of growth

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.

The following table shows the run time of these algorithms for different problem sizes:

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	100 010 001

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.

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.

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small n. But regardless of the coefficients, there will always be some value of n where $an^2 > bn$, for any values of a and b.

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.

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.

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.

An **order of growth** is a set of functions whose growth behavior is considered equivalent. For example, $2n$, $100n$ and $n+1$ belong to the same order of growth, which is written $O(n)$ in **Big-Oh notation** and often called **linear** because every function in the set grows linearly with n.

All functions with the leading term n^2 belong to $O(n^2)$; they are called **quadratic**.

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

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 $af + b$, where a and b are constants?*

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, order of growth is irrelevant.

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the "better" algorithm 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!

B.2 Analysis of basic Python operations

In Python, most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases with the number of digits.

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

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:

```
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.

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if you can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n, then the loop is in $O(n^a)$, even for large k.

Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k.

Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

String concatenation is linear; the run time depends on the sum of the lengths of the operands.

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time. The string method `join` is linear; the run time depends on the total length of the strings.

Most list methods are linear, but there are some exceptions:

- Adding an element to the end of a list is constant time on average; when it runs out of room it occasionally gets copied to a bigger location, but the total time for n operations is $O(n)$, so the average time for each operation is $O(1)$.
- Removing an element from the end of a list is constant time.
- Sorting is $O(n \log n)$.

Most dictionary operations and methods are constant time, but there are some exceptions:

- The run time of `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.
- `keys`, `values` and `items` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear.

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section B.4.

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 Python use an $O(n \log n)$ comparison sort?*

B.3 Analysis of search algorithms

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.

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.

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

If the elements of the sequence are in order, you can use a **bisection search**, which is $O(\log n)$. Bisection search is similar to the algorithm you might use to look a word up in a dictionary (a paper dictionary, not the data structure). Instead of starting at the beginning and checking each item in order, you start with the item in the middle and check whether the word you are looking for comes before or after. If it comes before, then you search the first half of the sequence. Otherwise you search the second half. Either way, you cut the number of remaining items in half.

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.

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

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.

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:

`add(k, v)`: Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.

`get(k)`: Look up and return the value that corresponds to key `k`. With a Python dictionary, `d`, this operation is written `d[k]` or `d.get(k)`.

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.

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` appends a key-value tuple to the list of items, which takes constant time.

`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.

An alternative is to keep the list sorted by key. Then `get` could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures that can implement `add` and `get` in log time, but that's still not as good as constant time, so let's move on.

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:

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())
```

```
    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` makes a list of `n` `LinearMaps`.

`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` 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.

Hashable objects that are considered equivalent return the same hash value, but the converse is not necessarily true: two objects with different values can return the same hash value.

`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.

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.

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.

Here is an implementation of a hashtable:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
```

```
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

`__init__` creates a `BetterMap` and initializes `num`, which keeps track of the number of items.

`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`.

`resize` make a new `BetterMap`, twice as big as the previous one, and then "rehashes" the items from the old map to the new.

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in `find_map`. That means that some objects that used to hash into the same LinearMap will get split up (which is what we wanted, right?).

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!

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.

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.

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.

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 $O(1)$.

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 unit each, the third costs 3 units, etc.

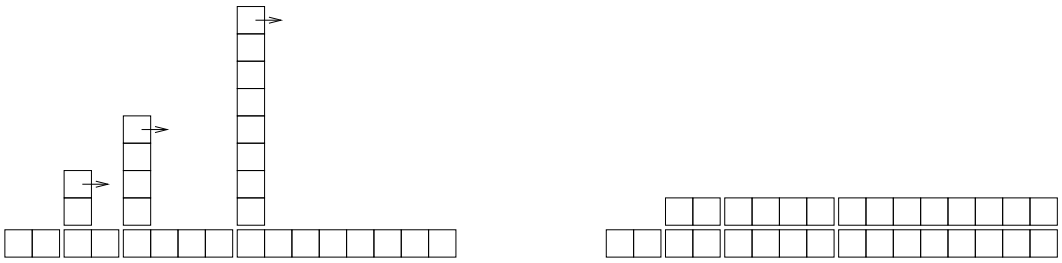

图 B.1: The cost of a hashtable add.

The extra work of rehashing appears as a sequence of increasingly tall towers with increasing space between them. Now if you knock over the towers, spreading the cost of resizing over all adds, you can see graphically that the total cost after n adds is $2n - 2$.

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://thinkpython2.com/code/Map.py`, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

B.5 Glossary

analysis of algorithms: A way to compare algorithms in terms of their run time and/or space requirements.

machine model: A simplified representation of a computer used to describe algorithms.

worst case: The input that makes a given algorithm run slowest (or require the most space).

leading term: In a polynomial, the term with the highest exponent.

crossover point: The problem size where two algorithms require the same run time or space.

order of growth: A set of functions that all grow in a way considered equivalent for purposes of analysis of algorithms. For example, all functions that grow linearly belong to the same order of growth.

Big-Oh notation: Notation for representing an order of growth; for example, $O(n)$ represents the set of functions that grow linearly.

linear: An algorithm whose run time is proportional to problem size, at least for large problem sizes.

quadratic: An algorithm whose run time is proportional to n^2, where n is a measure of problem size.

search: The problem of locating an element of a collection (like a list or dictionary) or determining that it is not present.

hashtable: A data structure that represents a collection of key-value pairs and performs search in constant time.