

Think Python

如何像计算机科学家一样思考

Allen Downey

译者: EarlGrey et al

Green Tea Press

Needham, Massachusetts

0	译者序	3
0.1	为什么想要翻译一本书?	3
0.2	为什么选择《Think Python》	3
0.3	为什么翻译 Python 3 版?	4
0.4	贡献者	4
0.5	声明	4
1	前言	5
1.1	本书与众不同的历史	5
1.2	Acknowledgments	6
1.3	Contributor List	7
2	第一章：程序之道	13
2.1	什么是程序?	13
2.2	运行 Python	14
2.3	第一个程序	14
2.4	算术运算符	15
2.5	值和类型	16
2.6	形式语言和自然语言	16
2.7	调试	18
2.8	术语表	18
2.9	练习题	20
3	第二章：变量、表达式和语句	21
3.1	赋值语句	21
3.2	变量名	21
3.3	表达式和语句	22
3.4	脚本模式	23
3.5	运算顺序	24
3.6	字符串运算	24
3.7	注释	25
3.8	调试	25
3.9	术语表	26
3.10	练习题	27

4	第三章：函数	29
4.1	函数调用	29
4.2	数学函数	30
4.3	组合	31
4.4	新建函数	31
4.5	定义和使用	32
4.6	执行流程	33
4.7	形参和实参	33
4.8	变量和形参都是局部的	34
4.9	堆栈图	35
4.10	有返回值函数和无返回值函数	36
4.11	为什么写函数？	37
4.12	调试	37
4.13	术语表	37
4.14	练习题	39
5	第四章：案例研究：接口设计	41
5.1	turtle 模块	41
5.2	简单的重复	42
5.3	练习	43
5.4	封装	44
5.5	泛化	44
5.6	接口设计	45
5.7	重构	46
5.8	开发方案	47
5.9	文档字符串	47
5.10	调试	47
5.11	术语表	48
5.12	练习题	49
6	第五章：条件和递归	51
6.1	地板除和求余	51
6.2	布尔表达式	52
6.3	逻辑运算符	52
6.4	有条件的执行	53
6.5	二选一执行	53
6.6	链式条件	53
6.7	嵌套条件	54
6.8	递归	55
6.9	递归函数的堆栈图	56
6.10	无限递归	56
6.11	键盘输入	57
6.12	调试	58
6.13	术语表	59
6.14	练习题	60
7	第六章：有返回值的函数	63
7.1	返回值	63
7.2	增量式开发	64
7.3	组合	66

7.4	布尔函数	66
7.5	再谈递归	67
7.6	信仰之跃	69
7.7	再举一例	69
7.8	检查类型	69
7.9	调试	70
7.10	术语表	71
7.11	练习题	72
8	第七章：迭代	75
8.1	重新赋值	75
8.2	更新变量	76
8.3	while 语句	76
8.4	break	77
8.5	平方根	78
8.6	算法	79
8.7	调试	80
8.8	术语表	80
8.9	练习题	81
9	第八章：字符串	83
9.1	字符串是一个序列	83
9.2	len	84
9.3	使用 for 循环遍历	84
9.4	字符串切片	85
9.5	字符串是不可变的	86
9.6	搜索	86
9.7	循环和计数	87
9.8	字符串方法	87
9.9	in 运算符	88
9.10	字符串比较	89
9.11	调试	89
9.12	术语表	91
9.13	练习题	91
10	第九章：文字游戏	95
10.1	读取单词列表	95
10.2	练习	96
10.3	搜索	97
10.4	使用索引进行循环	98
10.5	调试	99
10.6	术语表	100
10.7	练习题	100
11	第十章：列表	103
11.1	列表是一个序列	103
11.2	列表是可变的	103
11.3	遍历列表	105
11.4	列表操作	105
11.5	列表切片	105
11.6	列表方法	106

11.7	映射、筛选和归并	107
11.8	删除元素	108
11.9	列表和字符串	108
11.10	对象和值	109
11.11	别名	110
11.12	列表参数	111
11.13	调试	113
11.14	术语表	114
11.15	练习题	115
12	第十一章：字典	119
12.1	字典即映射	119
12.2	字典作为计数器集合	120
12.3	循环和字典	121
12.4	逆向查找	122
12.5	字典和列表	123
12.6	备忘录	124
12.7	全局变量	125
12.8	调试	127
12.9	术语表	127
12.10	练习题	129
13	第十二章：元组	131
13.1	元组是不可变的	131
13.2	元组赋值	132
13.3	元组作为返回值	133
13.4	可变长度参数元组	134
13.5	列表和元组	134
13.6	字典和元组	136
13.7	序列嵌套	137
13.8	调试	138
13.9	术语表	139
13.10	练习题	139
14	第十三章：案例研究：数据结构选择	143
14.1	词频分析	143
14.2	随机数	144
14.3	单词直方图	145
14.4	最常用单词	146
14.5	可选形参	147
14.6	字典差集	147
14.7	随机单词	148
14.8	马尔科夫分析	149
14.9	数据结构	150
14.10	调试	151
14.11	术语表	152
14.12	练习题	153
15	第十四章：文件	155
15.1	持久化	155
15.2	读取和写入	155

15.3	格式化运算符	156
15.4	文件名和路径	157
15.5	捕获异常	158
15.6	数据库	159
15.7	序列化	160
15.8	管道	160
15.9	编写模块	161
15.10	调试	162
15.11	术语表	163
15.12	练习题	164
16	第十五章：类和对象	165
16.1	程序员自定义类型	165
16.2	属性	166
16.3	矩形	167
16.4	实例作为返回值	168
16.5	对象是可变的	168
16.6	复制	169
16.7	调试	170
16.8	术语表	171
16.9	练习题	172
17	第十六章：类和函数	175
17.1	时间	175
17.2	纯函数	176
17.3	修改器	177
17.4	原型 vs. 方案	178
17.5	调试	179
17.6	术语表	179
17.7	练习题	180
18	第十七章：类和方法	183
18.1	面向对象的特性	183
18.2	打印对象	184
18.3	再举一例	185
18.4	一个更复杂的例子	185
18.5	init 方法	186
18.6	__str__ 方法	187
18.7	运算符重载	187
18.8	类型分发 (type-based dispatch)	188
18.9	多态性	189
18.10	接口和实现	190
18.11	调试	190
18.12	术语表	191
18.13	练习题	191
19	第十八章：继承	193
19.1	卡牌对象	193
19.2	类属性	194
19.3	比较卡牌	195
19.4	一副牌	196

19.5	打印一副牌	196
19.6	添加, 移除, 洗牌和排序	197
19.7	继承	198
19.8	类图	199
19.9	数据封装	200
19.10	调试	201
19.11	术语表	202
19.12	练习题	203
20	第十九章: 进阶小技巧	205
20.1	条件表达式	205
20.2	列表推导式	206
20.3	生成器表达式	207
20.4	any 和 all	207
20.5	集合	208
20.6	计数器	209
20.7	defaultdict	210
20.8	命名元组	211
20.9	汇集关键字实参	212
20.10	术语表	213
20.11	练习题	214
21	第二十章: 调试	215
21.1	语法错误	215
21.2	运行时错误	216
21.3	语义错误	219
22	第二十一章: 算法分析	223
22.1	增长量级	224
22.2	Python 基本运算操作分析	225
22.3	搜索算法分析	226
22.4	哈希表	227
22.5	术语表	230

Contents:

前言

0.1 本书与众不同的历史

1999 年 1 月, 我正准备使用 Java 教一门编程入门课程。我之前已经开了三次课, 但是却感到越来越沮丧。课程的不及格率太高, 即使对于及格的学生, 他们整体的收获也太低。

我看到的问题之一是教材。

它们都太厚重了, 写了太多关于 Java 的不必要细节, 却缺乏如何编程的上层指导 (high-level guidance)。这些教材都陷入了陷阱门效应 (trap door effect): 开始的时候简单, 逐渐深入, 然后大概到了第五章左右, 基础差的学生就跟不上了。学生们看的材料太多, 进展太快, 最后, 我在接下来的学期里都是在收拾残局 (pick up the pieces)。

所以, 在开始上课前两周, 我决定自己写一本书。我的目标是:

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

我需要一本书名, 所以一时兴起, 我选择了《如何像计算机科学家一样思考》。

这本书的第一版很粗糙, 但是却起了作用。学生们读了它之后, 对书中内容理解的很好, 因此我才可以在课堂上讲授那些困难、有趣的主题, 并让学生们动手实践 (这点最重要)。

我将此书以 GNU 自有文档许可的形式发布, 允许用户拷贝、修改和传播此书。

有趣的是接下来发生的事。弗吉尼亚一所高中的教师 Jeff Elkne 采用了我的教材, 并改为使用 Python 语言。他将修改过的书发给了我一份, 就这样, 我读着自己的书学会了 Python。2001 年, 通过 Green Tea Press, 我出版了本书的第一个 Python 版本。

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

此后, 我一直致力于改善本书, 纠正错误, 改进一些示例, 新增教学材料, 尤其是练习题。

最后的结果, 就是此书。现在的书名没有之前那么浮夸, 就叫《Think Python》。下面是一些变化:

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

《Think Python》第二版还有以下新特点：

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

我希望你能使用该书愉快的工作，也希望它能帮助你学习编程，学会像计算机科学家一样思考，至少有那么一点像。

Allen B. Downey
Olin College

0.2 Acknowledgments

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

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

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

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

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

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

0.3 Contributor List

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

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

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

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word “unconsciously” in Chapter 1 needed to be changed to “subconsciously”.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13.
- John Ouzts corrected the definition of “return value” in Chapter 3.

- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the `printTime` function was used in an example without being defined.
- Paul Sleight found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snyder is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.

- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D.ăJ.ăWebre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that “a error” is an error.
- Abel David and Alexis Dinno reminded us that the plural of “matrix” is “matrices”, not “matrixes”. This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.
- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of “argument” and “parameter”.
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of “use before def”.
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.

- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner spotted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in arc.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up math.pi too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exerciseā[exrotatepairs].
- Leah Engelbert-Fenton pointed out that I used tuple as a variable name, contrary to my own advice. And then found a bunch of typos and a “use before def” .
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exerciseā[checksum].
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary *Concrete Abstractions*, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4-11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.
- Adam Zimmerman found an inconsistency in my instance of an “instance” and several other errors.

- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.
- Russell Coleman helped me with my geometry.
- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.
- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!
- Gray Thomas knows his right from his left.
- Giovanni Escobar Sosa sent a long list of corrections and suggestions.
- Alix Etienne fixed one of the URLs.
- Kuang He found a typo.
- Daniel Neilson corrected an error about the order of operations.
- Will McGinnis pointed out that `polyline` was defined differently in two places.
- Swarup Sahoo spotted a missing semi-colon.
- Frank Hecker pointed out an exercise that was under-specified, and some broken links.
- Animesh B helped me clean up a confusing example.
- Martin Caspersen found two round-off errors.
- Gregor Ulm sent several corrections and suggestions.
- Dimitrios Tsirigkas suggested I clarify an exercise.
- Carlos Tafur sent a page of corrections and suggestions.
- Martin Nordsletten found a bug in an exercise solution.

- Lars O.D. Christensen found a broken reference.
- Victor Simeone found a typo.
- Sven Hoexter pointed out that a variable named input shadows a build-in function.
- Viet Le found a typo.
- Stephen Gregory pointed out the problem with cmp in Python 3.
- Matthew Shultz let me know about a broken link.
- Lokesh Kumar Makani let me know about some broken links and some changes in error messages.
- Ishwar Bhat corrected my statement of Fermat's last theorem.
- Brian McGhie suggested a clarification.
- Andrea Zanella translated the book into Italian, and sent a number of corrections along the way.
- Many, many thanks to Melissa Lewis and Luciano Ramalho for excellent comments and suggestions on the second edition.
- Thanks to Harry Percival from PythonAnywhere for his help getting people started running Python in a browser.
- Xavier Van Aubel made several useful corrections in the second edition.

第一章：程序之道

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

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

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

1.1 什么是程序？

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

不同编程语言中，程序的具体细节也不一样，但是有一些基本的指令几乎出现在每种语言当中：

输入（*input*）：

从键盘、文件、网络或者其他设备获取数据。

输出（*output*）：

在屏幕上显示数据，将数据保存至文件，通过网络传送数据，等等。

数学（*math*）：

执行基本的数学运算，如加法和乘法。

有条件执行（*conditional execution*）：

检查符合某个条件后，执行相应的代码。

重复 (*repetition*):

重复执行某个动作, 通常会有一些变化。

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

1.2 运行 Python

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

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

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

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

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

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

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

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

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

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

1.3 第一个程序

根据传统, 你用一门新语言写的第一个程序叫做 “Hello, World!”, 因为它的功能只不过是显示单词 “Hello, World!”。在 Python 中, 它看起来是这样:

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

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

```
Hello, World!
```

程序中的单引号标记了被打印文本的首尾；它们不会出现在结果中。

括号说明 `print` 是一个函数。我们将在第三章介绍函数。在 Python 2 中，`print` 是一个语句；不是函数，所以不需要使用括号。

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

很快你就会明白二者之间的区别，现在知道这些就足够了。

译者注：Python 核心开发者 Brett Cannon 详细解释了 [为什么 print 在 Python 3 中变成了函数](#)。

1.4 算术运算符

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

运算符 `+`、`-` 和 `*` 分别执行加法、减法和乘法，详见以下示例：

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

运算符 `/` 执行除法运算：

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

你可能会问，为什么结果是 42.0，而不是 42。在下节中，我会进行解释。

最后，运算符 `*` 执行乘方运算；也就是说，它将某个数字乘以自身相应的次数：

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

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

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

我不打算在本书中介绍位运算符，但是你可以阅读 [Python 官方百科](#)，了解相关内容。

1.5 值和类型

值 (value) 是程序处理的基本数据之一，比如说一个单词或一个数字。我们目前已经接触到的值有：2，42.0，和 'Hello World!'。

这些值又属于不同的 **类型 (types)**：2 是一个 **整型数 (integer)**，42.0 是一个 **浮点数 (floating point number)**，而 'Hello, World!' 则是一个 **字符串 (string)**，之所以这么叫是因为其中的字符被串在了一起 (*strung together*)。

如果你不确定某个值的类型是什么，解释器可以告诉你：

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

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

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

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

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

它们其实是字符串。

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

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

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

1.6 形式语言和自然语言

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

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

编程语言是被设计用于表达计算的形式语言。

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

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

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

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

译者注：上面两句英文都是不符合语法的，一个包含非法标记，另一个结构不符合语法。

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

虽然形式语言和自然语言有很多共同点——标记、结构和语法，它们也有一些不同：

歧义性：

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

冗余性：

为了弥补歧义性并减少误解，自然语言使用很多冗余。结果，自然语言经常很长。形式语言则冗余较少，更简洁。

字面性：

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

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

诗歌：

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

散文：

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

程序：

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

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

1.7 调试

程序员都会犯错。由于比较奇怪的原因，编程错误被称为 **故障**（译者注：英文为 **bug**，一般指虫子），追踪错误的过程被称为 **调试**（**debugging**）。

编程，尤其是调试，有时会让人动情绪。如果你有个很难的 **bug** 解决不了，你可能会感到愤怒、忧郁抑或是丢人。

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

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

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

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

1.8 术语表

解决问题：

将问题形式化、寻找并表达解决方案的过程。

高级语言 (*high-level language*)：

像 Python 这样被设计成人类容易阅读和编写的编程语言。

低级语言 (*low-level language*):

被设计成计算机容易运行的编程语言；也被称为“机器语言”或“汇编语言 (*assembly language*)”。

可移植性：

程序能够在多种计算机上运行的特性。

解释器：

读取另一个程序并执行该程序的程序。

提示符：

解释器所显示的字符，表明已准备好接受用户的输入。

程序：

说明一个计算的一组指令。

打印语句：

使 Python 解释器在屏幕上显示某个值的指令。

运算符:

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

值:

程序所处理数据的基本元素之一，例如数字或字符串。

类型:

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

整型数:

代表整数的类型。

浮点数:

代表一个有小数点的数字的类型。

字符串:

代表一系列字符的类型。

自然语言:

任意一种人们日常使用的、自然演变而来的语言。

形式语言:

任意一种人类为了某种目的而设计的语言，例如用来表示数学概念或者电脑程序；所有的编程语言都是形式语言。

记号:

程序语法结构中的基本元素之一，与自然语言中的单词类似。

语法:

规定了程序结构的规则。

解析:

阅读程序，并分析其语法结构的过程

故障:

程序中的错误。

调试:

寻找并解决错误的过程。

1.9 练习题

1.9.1 习题 1-1

你最好在电脑前阅读此书，因为你可以随时测试书中的示例。

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

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

1. 在打印语句中，如果你去掉一个或两个括号，会发生什么？
2. 你想打印一个字符串，如果你去掉一个或两个引号，会发生什么？
3. 你可以使用减号创建一个负数，如-2。如果你在一个数字前再加上个加号，会发生什么？`2++2` 会得出什么结果？
4. 在数学标记中，前导零（leading zeros）没有问题，如 02。如果我们在 Python 中这样做，会发生什么？
5. 如果两个值之间没有运算符，又会发生什么？

1.9.2 习题 1-2

启动 Python 解释器，把它当计算器使用。

1. 42 分 42 秒一共有多少秒？
2. 10 公里可以换算成多少英里？提示：一英里等于 1.61 公里。
3. 如果你花 42 分 42 秒跑完了 10 公里，你的平均配速（pace）是多少（每英里耗时，分别精确到分和秒）？你每小时平均跑了多少英里（英里/时）？

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

贡献者

1. 翻译：@bingjin
2. 校对：@bingjin
3. 参考：@carfly

第二章：变量、表达式和语句

编程语言最强大的特性之一，是操作**变量**的能力。变量是指向某个值的名称。

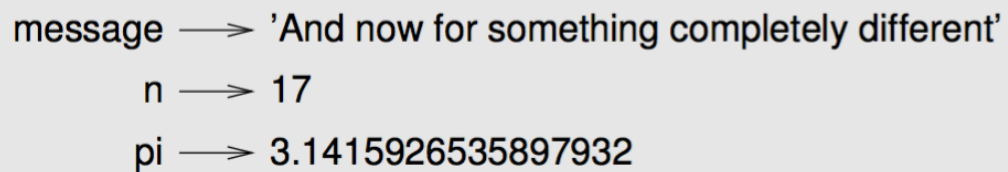
2.1 赋值语句

赋值语句（**assignment statement**）会新建变量，并为这个变量赋值。

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

这个例子进行了三次赋值。第一次将一个字符串赋给一个叫 `message` 的新变量；第二次将整型数 17 赋给 `n`；第三次将 π 的（近似）值赋给 `pi`。

在纸上表示变量的一个常见方法，是写下变量名，并用箭头指向变量的值。这种图被称为**状态图（state diagram）**，因为它展示了每个变量所处的状态（可以把其看成是变量的心理状态）。图 2-1 展示了前面例子的结果。



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

图 2.1: 图 2-1: 状态图。

2.2 变量名

程序员通常为变量选择有意义的名字—它们可以记录该变量的用途。

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

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

如果你给了变量一个非法的名称，解释器将抛出一个语法错误：

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

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

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

Python 3 有以下关键词：

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

你没有必要背诵上面的关键词。大部分的开发环境会用不同的颜色区别显示关键词；如果你不小心使用关键词作为变量名，你肯定会发现的。

2.3 表达式和语句

表达式（**expression**）是值、变量和运算符的组合。值自身也被认为是一个表达式，变量也是，因此下面都是合法的表达式：

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

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

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

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

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

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

2.4 脚本模式

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

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

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

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

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

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

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

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

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

这个行为开始可能有些令人费解。

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

例如，以下脚本

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

产生的输出结果是

```
1
2
```

赋值语句不产生输出。

在 Python 解释器中键入以下的语句，看看他们的结果是否符合你的理解：

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

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

2.5 运算顺序

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

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

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

2.6 字符串运算

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

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

但有两个例外，+ 和 *。

加号运算符 + 可用于**字符串拼接（string concatenation）**，也就是将字符串首尾相连起来。例如：

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

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

+ 和 * 的这个用法，类比加法和乘法也讲得通。就像 $4*3$ 与 $4+4+4$ 等价一样，我们也会期望 `'Spam'*3` 和 `'Spam'+'Spam'+'Spam'` 等价，而事实上的确如此。另一方面，字符串拼接和重复与整数的加法和乘法也有很大的不同。你能想出来一个加法具有而字符串拼接不具有的特性么？

2.7 注释

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

因此，在你的程序中用自然语言做笔记，解释程序做什么通常是比较好的办法。这些标注被称为**注释（comments）**，以 `#` 符号开始。

```
# 计算百分比
percentage = (minute * 100) / 60
```

此例中，注释独立一行。你也可以将注释放在行尾：

```
percentage = (minute * 100) / 60 # 计算百分比
```

从 `#` 开始到行尾的所有内容都会被解释器忽略——其对程序执行没有影响。

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

下面这个注释只是重复了代码，没有什么用：

```
v = 5 # 5 就是 v
```

下面的注释包括了代码中没有的有用信息：

```
v = 5 # 22222222/2
```

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

2.8 调试

程序中可能会出现下面三种错误：语法错误（**syntax error**）、运行时错误（**runtime error**）和语义错误（**semantic error**）。我们如果能够分辨出三者区别，有助于快速追踪这些错误。

语法错误：

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

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

运行时错误：

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

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

语义错误:

第三类错误是“语义”错误, 即与程序的意思的有关。如果你的程序中有语义错误, 程序在运行时不会产生错误信息, 但是不会返回正确的结果。它会返回另外的结果。严格来说, 它是按照你的指令在运行。

识别语义错误可能是棘手的, 因为这需要你反过来思考, 通过观察程序的输出来搞清楚它在做什么。

2.9 术语表

变量:

变量是指向某个值的名称。

赋值语句:

将某个值赋给变量的语句。

状态图:

变量及其所指的值的图形化表示。

关键字:

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

运算数 (operand):

运算符所操作的值之一。

表达式:

变量、运算符和值的组合, 代表一个单一的结果。

计算 (evaluate):

通过执行运算以简化表达式, 从而得出一个单一的值。

语句:

代表一个命令或行为的一段代码。目前为止我们接触的语句有赋值语句和打印语句。

执行:

运行一个语句, 并按照语句的指令操作。

交互式模式:

通过在提示符中输入代码, 使用 Python 解释器的一种方式。

脚本模式:

使用 Python 解释器从脚本中读取代码, 并运行脚本的方式。

脚本:

保存在文件中的程序。

运算顺序：

有关多个运算符和运算数时计算顺序的规则。

拼接：

将两个运算数首尾相连。

注释：

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

语法错误：

使得程序无法进行解析（因此无法进行解释）的错误。

异常：

只有在程序运行时才发现的错误。

语义：

程序中表达的意思。

语义错误：

使得程序偏离程序员原本期望的错误。

2.10 练习题

2.10.1 习题 2-1

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

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

2.10.2 习题 2-2

继续练习将 Python 解释器当做计算器使用：

1. 半径为 r 的球体积是 $\frac{4}{3}\pi r^3$ 。半径为 5 的球体积是多少？

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

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

2.10.3 贡献者

1. 翻译：@bingjin
2. 校对：@bingjin
3. 参考：@carfly

第三章：函数

在编程的语境下，**函数（function）**指的是一个有命名的、执行某个计算的语句序列（sequence of statements）。在定义一个函数的时候，你需要指定函数的名字和语句序列。之后，你可以通过这个名字“调用（call）”该函数。

3.1 函数调用

我们已经看见过一个**函数调用（function call）**的例子。

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

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

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

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

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

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

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

`float` 可以将整型数和字符串转换为浮点数：

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

最后, `str` 可以将其实参转换成字符串:

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

3.2 数学函数

Python 中有一个数学模块 (`math`), 提供了大部分常用的数学函数。模块 (**module**) 指的是一个含有相关函数的文件。

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

```
>>> import math
```

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

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

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

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

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

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

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

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

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

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

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

3.3 组合

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

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

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

甚至是函数调用：

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

几乎任何你可以放值的地方，你都可以放一个任意类型的表达式，只有一个例外：赋值语句的左侧必须是一个变量名。左侧放其他任何表达式都会产生语法错误（后面我们会讲到这个规则的例外）。

```
>>> minutes = hours * 60                # [OK]
>>> hours * 60 = minutes                  # [Error]
SyntaxError: can't assign to operator
```

3.4 新建函数

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

下面是一个示例：

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

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

函数名后面的圆括号是空的，表明该函数不接受任何实参。

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

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

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

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

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

为了结束函数定义，你必须输入一个空行。

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

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

调用新函数的语法，和调用内建函数的语法相同：

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

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

```
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.
```

不过，这首歌的歌词实际上不是这样的。

3.5 定义和使用

将上一节的多个代码段组合在一起，整个程序看起来是这样的：

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

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

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

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

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

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

3.6 执行流程

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

执行流程总是从程序的第一条语句开始，自顶向下，每次执行一条语句。

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

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

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

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

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

3.7 形参和实参

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

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

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

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

该函数对任意能被打印的值都有效。

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

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

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

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

你也可以用变量作为实参：

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

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

3.8 变量和形参都是局部的

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

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

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

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

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


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

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

3.9 堆栈图

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

每个函数用一个**栈帧**（**frame**）表示。一个栈帧就是一个线框，函数名在旁边，形参以及函数内部的变量则在里面。前面例子的堆栈图如图 3-1 所示。

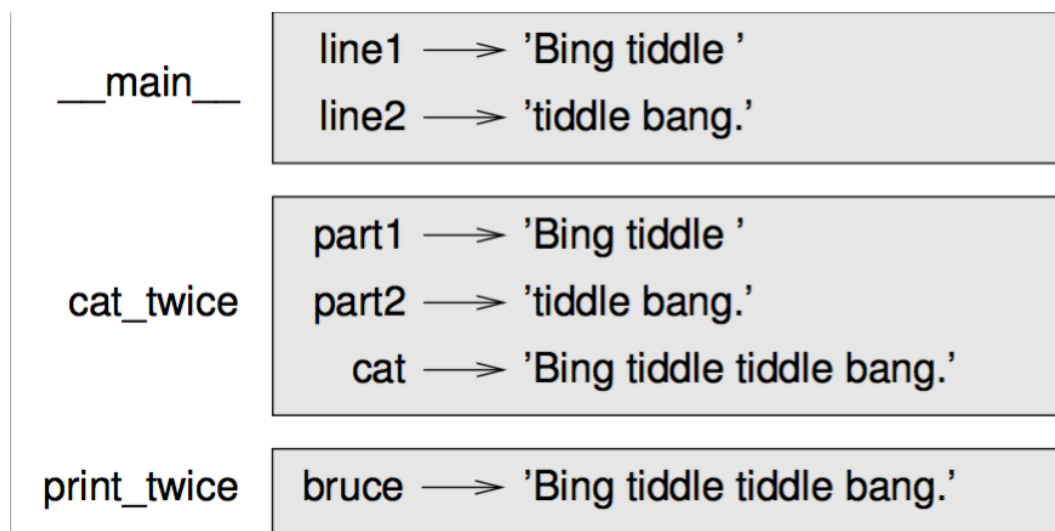


图 3.1: 图 3-1: 堆栈图。

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

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

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

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

```
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
```

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

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

3.10 有返回值函数和无返回值函数

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

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

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

当你在交互模式下调用一个函数时，Python 解释器会马上显示结果：

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

但是在脚本中，如果你单调用一个有返回值函数，返回值就永远丢失了！

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

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

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

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

None 这个值和字符串 `'None'` 不同。这是一个自己有独立类型的特殊值：

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

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

3.11 为什么写函数？

你可能还不明白为什么值得将一个程序分解成多个函数。原因包括以下几点：

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

3.12 调试

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

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

调试也像是一门实验性科学。一旦你猜到大概哪里出错了，你可以修改程序，再试一次。如果你的假设是正确的，那么你就可以预测到修改的结果，并且离正常运行的程序又近了一步。如果你的假设是错误的，你就不得不再提一个新的假设。如夏洛克·福尔摩斯所指出的，“当你排除了所有的不可能，无论剩下的是什么，不管多么难以置信，一定就是真相。”（阿瑟·柯南·道尔，《四签名》）

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

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

3.13 术语表

函数（function）：

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

函数定义（function definition）：

创建一个新函数的语句，指定了函数名、形参以及所包含的语句。

函数对象（function object）：

函数定义所创建的一个值。函数名是一个指向函数对象的变量。

函数头 (header):

函数定义的第一行。

函数体 (body):

函数定义内部的语句序列。

形参 (parameters):

函数内部用于指向被传作实参的值的名字。

函数调用 (function call):

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

实参 (argument):

函数调用时传给函数的值。这个值被赋给函数中相对应的形参。

局部变量 (local variable):

函数内部定义的变量。局部变量只能在函数内部使用。

返回值 (return value):

函数执行的结果。如果函数调用被用作表达式, 其返回值是这个表达式的值。

有返回值函数 (fruitful function):

会返回一个值的函数。

无返回值函数 (void function):

总是返回 None 的函数。

None:

无返回值函数返回的一个特殊值。

模块 (module):

包含了一组相关函数及其他定义的文件。

导入语句 (import statement):

读取一个模块文件, 并创建一个模块对象的语句。

模块对象 (module object):

导入语句创建的一个值, 可以让开发者访问模块内部定义的值。

点标记法 (dot notation):

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

组合 (composition):

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

执行流程（flow of execution）：

语句执行的顺序。

堆栈图（stack diagram）：

一种图形化表示堆栈的方法，堆栈中包括函数、函数的变量及其所指向的值。

栈帧（frame）：

堆栈图中的一个栈帧，代表一个函数调用。其中包含了函数的局部变量和形参。

回溯（traceback）：

当出现异常时，解释器打印出的出错时正在执行的函数列表。

3.14 练习题

3.14.1 习题 3-1

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

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

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

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

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

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

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

do_twice(print_spam)
```

1. 将这个示例写入脚本，并测试。
2. 修改 `do_twice`，使其接受两个实参，一个是函数对象，另一个是值。然后调用这一函数对象两次，将那个值传递给函数对象作为实参。
3. 从本章前面一些的示例中，将 `print_twice` 函数的定义复制到脚本中。
4. 使用修改过的 `do_twice`，调用 `print_twice` 两次，将 'spam' 传递给它作为实参。

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

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

注意：这一习题只能使用我们目前学过的语句和特性来完成。

3.14.2 习题 3-2

1. 编写一个能画出如下网格（grid）的函数：



提示：你可以使用一个用逗号分隔的值序列，在一行中打印出多个值：

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

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

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

这两个语句的输出结果是 `'+ -'`。

一个没有传入实参的 `print` 语句会结束当前行，跳到下一行。

2. 编写一个能够画出四行四列的类似网格的函数。

答案: <http://thinkpython2.com/code/grid.py>。致谢：这个习题基于 *Practical C Programming, Third Edition* 一书中的习题改编，此书由 O'Reilly 出版社于 1997 年出版。

3.14.3 贡献者

1. 翻译: @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第四章：案例研究：接口设计

本章将通过一个案例研究，介绍如何设计出相互配合的函数。

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

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

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

4.1 turtle 模块

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

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

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

新建一个名叫 `mypolygon.py` 的文件，输入以下代码：

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

`turtle` 模块（小写的 `t`）提供了一个叫作 `Turtle` 的函数（大写的 `T`），这个函数会创建一个 `Turtle` 对象，我们将其赋值给名为 `bob` 的变量。打印 `bob` 的话，会输出下面这样的结果：

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

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

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

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

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

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

`fd` 方法的实参是像素距离，所以实际前进的距离取决于你的屏幕。

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

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

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

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

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

现在修改程序，画一个正方形。在没有成功之前，不要继续往下看。

4.2 简单的重复

很有可能你刚才写了像下面这样的程序：

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

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

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

bob.fd(100)
```

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

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

你应该会看到如下输出：

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


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

下面是一个画正方形的 `for` 语句：

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

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

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

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

4.3 练习

下面是一系列学习使用 **Turtle** 的练习。这些练习虽说是为了好玩，但是也有自己的目的。你在做这些练习的时候，想一想它们的目的是什么。

译者注：原文中使用的还是 **TurtleWorld**，应该是作者忘了修改。

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

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

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

2. 给 `square` 增加另一个名为 `length` 的形参。修改函数体，使得正方形边的长度是 `length`，然后修改函数调用，提供第二个实参。重新运行程序。用一系列 `length` 值测试你的程序。
3. 复制 `square`，并将函数改名为 `polygon`。增加另外一个名为 `n` 的形参并修改函数体，让它画一个正 `n` 边形（`n-sided regular polygon`）。提示：正 `n` 边形的外角是 $360/n$ 度。
4. 编写一个名为 `circle` 的函数，它接受一个海龟 `t` 和半径 `r` 作为形参，然后以合适的边长和边数调用 `polygon`，画一个近似圆形。用一系列 `r` 值测试你的函数。

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

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

4.4 封装

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

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

square(bob)
```

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

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

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

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

4.5 泛化

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

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

square(bob, 100)
```

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

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

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

polygon(bob, 7, 70)
```

这个示例代码画了一个边长为 70 的七边形。

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

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

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

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

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

4.6 接口设计

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

```
import math

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

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

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

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

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

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

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

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

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

4.7 重构

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

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

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

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

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

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

现在, 我们可以用 `polyline` 重写 `polygon` 和 `arc`:

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

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

最后, 我们可以用 `arc` 重写 `circle`:

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

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

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

4.8 开发方案

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

1. 从写一个没有函数定义的小程序开始。
2. 一旦该程序运行正常，找出其中相关性强的部分，将它们封装进一个函数并给它一个名字。
3. 通过增加适当的形参，泛化该函数。
4. 重复 1-3 步，直到你有一些可正常运行的函数。复制粘贴有用的代码，避免重复输入（和重新调试）。
5. 寻找机会通过重构改进程序。例如，如果在多个地方有相似的代码，考虑将它分解到一个合适的通用函数中。

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

4.9 文档字符串

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

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

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

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

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

4.10 调试

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

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

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

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

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

4.11 术语表

方法（**method**）：

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

循环（**loop**）：

程序中能够重复执行的那部分代码。

封装（**encapsulation**）：

将一个语句序列转换成函数定义的过程。

泛化（**generalization**）：

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

关键字实参（**keyword argument**）：

包括了形参名称作为“关键字”的实参。

接口（**interface**）：

对如何使用一个函数的描述，包括函数名、参数说明和返回值。

重构（**refactoring**）：

修改一个正常运行的函数，改善函数接口及其他方面代码质量的过程。

开发计划（**development plan**）：

编写程序的一种过程。

文档字符串（**docstring**）：

出现在函数定义顶部的一个字符串，用于记录函数的接口。

先决条件（**preconditions**）：

在函数运行之前，调用者应该满足的要求。**ends.**

后置条件（**postconditions**）：

函数终止之前应该满足的条件。

4.12 练习题

4.12.1 习题 4-1

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

1. 画一个执行 `circle(bob, radius)` 时的堆栈图 (stack diagram)，说明程序的各个状态。你可以手动进行计算，也可以在代码中加入打印语句。
2. “重构”一节中给出的 `arc` 函数版本并不太精确，因为圆形的线性近似 (linear approximation) 永远处在真正的圆形之外。因此，`Turtle` 总是和正确的终点相差几个像素。我的答案中展示了降低这个错误影响的一种方法。阅读其中的代码，看看你是否能够理解。如果你画一个堆栈图的话，你可能会更容易明白背后的原理。

4.12.2 习题 4-2

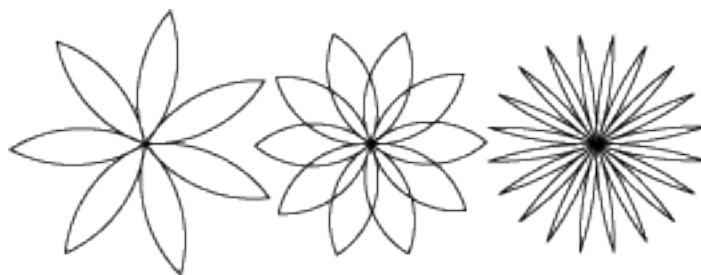


图 4.1: 图 4-1: 使用 Turtle 绘制的花朵。

编写比较通用的一个可以画出像图 4-1 中那样花朵的函数集。

答案: <http://thinkpython2.com/code/flower.py> , 还要求使用这个模块 <http://thinkpython2.com/code/polygon.py>.

4.12.3 习题 4-3

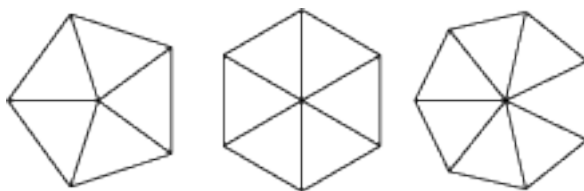


图 4.2: 图 4-2: 使用 Turtle 画的饼状图。

编写比较通用的一个可以画出图 4-2 中那样图形的函数集，。

答案: <http://thinkpython2.com/code/pie.py> 。

4.12.4 习题 4-4

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

你应该为每个字母写一个函数，起名为 `draw_a`, `draw_b` 等等，然后将你的函数放在一个名为 `letters.py` 的文件里。你可以从<http://thinkpython2.com/code/typewriter.py> 下载一个“海龟打字员”来帮你测试代码。

你可以在 <http://thinkpython2.com/code/letters.py> 中找到答案；这个解法还要求使用 <http://thinkpython2.com/code/polygon.py>。

4.12.5 习题 4-5

前往<http://en.wikipedia.org/wiki/Spiral> 阅读螺旋线（spiral）的相关知识；然后编写一个绘制阿基米德螺旋线（或者其他种类的螺旋线）的程序。

答案：<http://thinkpython2.com/code/spiral.py>。

4.12.6 贡献者

1. 翻译：@bingjin
2. 校对：@bingjin
3. 参考：@carfly

第五章：条件和递归

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

5.1 地板除和求余

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

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

但是，用小时做单位的时候，我们通常并不写出小数部分。地板除丢弃除法运算结果的小数部分，返回整数个小时：

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

如果你希望得到余数，你可以从除数中减去一个小时也就是 60 分钟：

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

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

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

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

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

如果你正在使用 Python 2, 那么除法就会和前面的介绍有点不同。除法运算符 `/` 在被除数和除数都是整数的时候，会进行地板除，但是当被除数和除数中任意一个是浮点数的时候，则进行浮点数除法。（译者注：在 Python3 中，无论任何类型都会保持小数部分）

5.2 布尔表达式

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

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

`True` 和 `False` 是属于 `bool` 类型的特殊值；它们不是字符串。

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

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

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

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

5.3 逻辑运算符

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

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

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

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

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

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

5.4 有条件的执行

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

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

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

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

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

```
if x < 0:
    pass           # 占位符
```

5.5 二选一执行

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

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

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

5.6 链式条件

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

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

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

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

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

5.7 嵌套条件

一个条件可以嵌到另一个里面。我们可以这样写前一节的例子：

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

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

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

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

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

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

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

对于这样的条件，Python 提供了一种更加简洁的写法。

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

5.8 递归

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

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

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

如果我们像这样调用该函数会发生什么呢？

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

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

`countdown` 开始以 $n=2$ 执行，由于 n 大于 0，它输出值 2，然后调用它自己...

`countdown` 开始以 $n=1$ 执行，既然 n 大于 0，它输出值 1，然后调用它自己...

`countdown` 开始以 $n=0$ 执行，由于 n 不大于 0，它输出单词 “Blastoff!”，然后返回。

获得 $n=1$ 的 `countdown` 返回。

获得 $n=2$ 的 `countdown` 返回。

获得 $n=3$ 的 `countdown` 返回。

然后你回到 `__main__` 中。因此整个输出类似于：

```
3
2
1
Blastoff!
```

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

再举一例，我们可以写一个函数，其打印一个字符串 n 次。

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

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

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

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

5.9 递归函数的堆栈图

在堆栈图一节中，我们用堆栈图表示了一个函数调用期间程序的状态。这种图也能帮我们理解递归函数。

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

图 5-1: 堆栈图展示了一个以 $n = 3$ 调用 `countdown` 的堆栈图。

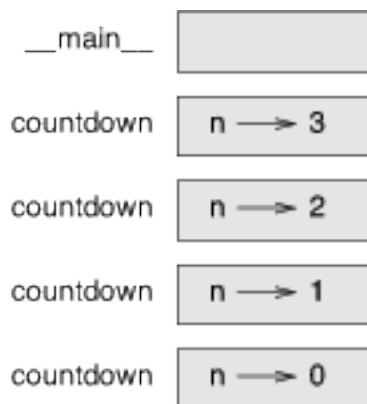


图 5.1: 图 5-1: 堆栈图

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

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

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

5.10 无限递归

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

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

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

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

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

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

5.11 键盘输入

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

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

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

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

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

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

如果你期望用户键入一个整型数，那么你可以试着将返回值转化为 `int`：

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

```
42
>>> int(speed)
42
```

但是，如果用户输入不是数字构成的字符串，你会获得一个错误：

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

我们后面将介绍处理这类错误的方法。

5.12 调试

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

- 是哪类错误，以及
- 在哪儿出现。

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

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

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

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

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

但是，当你运行它的时候，你却获得一个异常。

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


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

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

5.13 术语表

地板除:

一个操作符, 用 `//` 表示, 表示对两个数做除法同时向 0 取整。

求余运算符:

一个运算符, 用百分号 `%` 表示, 返回两个数相除的余数

布尔表达式:

一个值要么为真要么为假的表达式。

关系运算符:

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

逻辑运算符:

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

条件语句:

一段根据某个条件决定程序执行流程的语句。

条件:

决定哪个分支会被执行的布尔表达式

复合语句:

由语句头和语句体组成的语句。语句头以: 结尾, 语句体相对语句头缩进。

分支:

条件语句中的选择性语句序列。

链式条件:

由一系列替代分支组成的条件语句。

嵌套条件:

出现另一个条件语句某个分支中的条件语句。

返回语句: 结束函数执行并且将结果返回给调用者的语句。

递归:

调用正在执行的函数本身的过程。

基本情形:

在递归函数中，不进行递归调用的条件分支。

无限递归：

没有基本情形或者无法出现基本情形的递归函数。最终无限递归会导致运行时错误。

5.14 练习题

5.14.1 习题 5-1

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

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

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

5.14.2 习题 5-2

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

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

对于任何大于 2 的 n 成立。

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

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

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

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

5.14.3 习题 5-3

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

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

1. 写一个名为 `is_triangle` 的函数，其接受三个整数作为形参，能够根据给定的三个长度的棍子能否构成三角形来打印 “Yes” 或 “No”。
2. 写一个函数，提示用户输入三根棍子的长度，将它们转换成整型数，然后使用 `is_triangle` 检查给定长度的棍子能否构成三角形。

5.14.4 习题 5-4

下面程序的输出是什么？画出展示程序每次打印输出时的堆栈图。

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

recurse(3, 0)
```

1. 如果你这样调用函数：`recurse(-1,0)`，会有什么结果？
2. 请写一个文档字符串，解释调用该函数时需要了解的全部信息（仅此而已）。

5.14.5 习题 5-5

后面的习题要用到第四章中的 Turtle：

阅读如下的函数，看看你能否看懂它是做什么的。然后运行它（见第四章的例子）。

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

5.14.6 习题 5-6

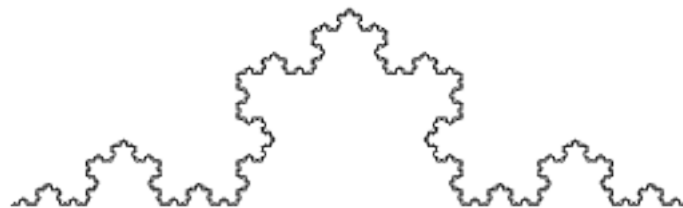


图 5.2: 图 5-2: 科赫曲线 (Koch Curve)。

科赫曲线 (Koch Curve) 是一个看起来类似图 5-2 的不规则碎片几何体 (fractal)。要画一个长度为 x 的科赫曲线，你只需要：

1. 画一个长度为 $x/3$ 的科赫曲线。
2. 左转 60 度。

3. 画一个长度为 $x/3$ 的科赫曲线。
4. 右转 120 度。
5. 画一个长度为 $x/3$ 的科赫曲线。
6. 左转 60 度。
7. 画一个长度为 $x/3$ 的科赫曲线。

例外情况是 x 小于 3 的情形：此时，你只需要画一道长度为 x 的直线。

1. 写一个名为 `koch` 的函数，接受一个海龟和一个长度作为形参，然后使用海龟画一条给定长度的科赫曲线。
 2. 写一个名为 `snowflake` 的函数，画出三条科赫曲线，构成雪花的轮廓。
- 答案：<http://thinkpython.com/code/koch.py>。
3. 科赫曲线能够以多种方式泛化。点击http://en.wikipedia.org/wiki/Koch_snowflake 查看例子，并实现你最喜欢的那种方式。

5.14.7 贡献者

1. 翻译：@iphyer
2. 校对：@bingjin
3. 参考：@carfly

第六章：有返回值的函数

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

6.1 返回值

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

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

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

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

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

之前我们已经见过 `return` 语句了，但是在一个有返回值的函数中，`return` 语句包含一个表达式。这条语句的意思是：“马上从该函数返回，并使用接下来的表达式作为返回值”。此表达式可以是任意复杂的，因此我们可以将该函数写得更简洁些：

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

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

有时，在条件语句的每一个分支内各有一个返回语句会很有用，。

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

因为这些 `return` 语句在不同的条件内，最后只有一个会被执行。

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

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

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

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

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

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

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

6.2 增量式开发

随着你写的函数越来越大，你在调试上花的时间可能会越来越多。

为了应对越来越复杂的程序，你可能会想尝试一种叫作 **增量式开发（incremental development）** 的方法。增量式开发的目标，是通过每次只增加和测试少量代码，来避免长时间的调试。

举个例子，假设你想计算两个给定坐标点 (x_1, y_1) 和 (x_2, y_2) 之间的距离。根据勾股定理（the Pythagorean theorem），二者的距离是：

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

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

本例中，输入是可以用 4 个数表示的两个点。返回值是距离，用浮点数表示。

现在你就可以写出此函数的轮廓了：

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

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

用样例实参调用它来进行测试。

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

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

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

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

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

下一步我们计算 `dx` 和 `dy` 的平方和。

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

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

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

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

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

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

这种开发方式的关键是：

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

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

6.3 组合

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

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

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

下一步是用得到的半径计算圆面积；我们也刚写了这样的函数：

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

将这些步骤封装在一个函数中，可以得到下面的函数：

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

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

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

6.4 布尔函数

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

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

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

请看下面的示例：

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

`==` 运算符的结果是布尔值，因此我们直接返回它，让代码变得更简洁。


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

布尔函数通常被用于条件语句中：

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

很容易写出下面这样的代码：

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

但这里的比较是多余的。

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

6.5 再谈递归

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

阿兰·图灵第一个证明了这种说法的正确性，这可是一项非凡的工作。他是首批计算机科学家之一（一些人认为他是数学家，但很多早期的计算机科学家也是出身于数学家）。相应地，这被称为图灵理论。关于图灵理论更完整（和更准确）的讨论，我推荐 Michael Sipser 的书《*Introduction to the Theory of Computation*》。

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

漩涡状： 一个用以描述漩涡状物体的形容词。

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

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

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

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

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

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

如果实参刚好是 0，我们就返回 1：

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

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

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

程序的执行流程和第五章[递归](#)一节中的 `countdown` 类似。如果我们传入参数的值是 3：

由于 3 不等于 0，我们执行第二个分支并计算 $n-1$ 的阶乘...

由于 2 不等于 0，我们执行第二个分支并计算 $n-1$ 的阶乘...

由于 1 不等于 0，我们执行第二个分支并计算 $n-1$ 的阶乘...

由于 0 等于 0，我们执行第一个分支并返回 1，不再进行任何递归调用。

返回值 1 与 n （其为 1）相乘，并返回结果。

返回值 1 与 n （其为 2）相乘，并返回结果。

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

图 6-1：堆栈图显示了该函数调用序列的堆栈图看上去是什么样子。

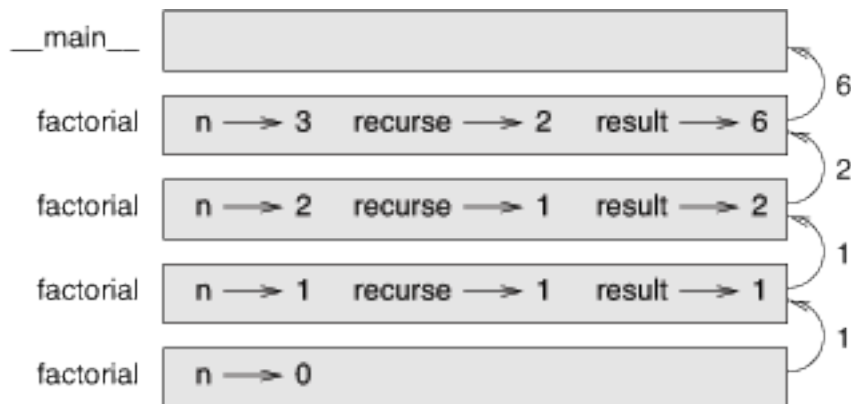


图 6.1: 图 6-1: 堆栈图

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

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

6.6 信仰之跃

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

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

当你调用一个自己写的函数时也是一样。例如，在布尔函数一节中，我们写了一个 `is_divisible` 函数来判断一个数能否被另一个数整除。通过对代码的检查，一旦我们确信这个函数能够正确运行，我们就能不用再查看函数体而直接使用了。

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

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

6.7 再举一例

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

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

翻译成 Python，看起来就像这样：

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

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

6.8 检查类型

如果我们将 1.5 作为参数调用阶乘函数会怎样？

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

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

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

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

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

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

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

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

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

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

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

6.9 调试

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

- 该函数获得的实参有些问题，违反先决条件。

- 该函数有些问题，违反后置条件。
- 返回值或者它的使用方法有问题。

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

如果形参看起来没问题，就在每个 `return` 语句之前增加一条 `print` 语句，来打印返回值。如果可能，手工检查结果。考虑用一些容易检查的值来调用该函数（类似在增量式开发一节中那样）。

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

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

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

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

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

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

6.10 术语表

临时变量（temporary variable）：

一个在复杂计算中用于存储过度值的变量。

死代码（dead code）：

程序中永远无法执行的那部分代码，通常是因为其出现在一个返回语句之后。

增量式开发 (incremental development):

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

脚手架代码 (scaffolding):

程序开发中使用的代码, 但并不是最终版本的一部分。

监护人 (guardian):

一种编程模式, 使用条件语句来检查并处理可能引发错误的情形。

6.11 练习题

6.11.1 习题 6-1

画出下面程序的堆栈图。这个程序的最终输出是什么?

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

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

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

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

6.11.2 习题 6-2

Ackermann 函数 $A(m, n)$ 的定义如下:

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

查看http://en.wikipedia.org/wiki/Ackermann_function。编写一个叫作“ack”的函数来计算 Ackermann 函数。使用你的函数计算 ack(3, 4), 其结果应该为 125。如果 m 和 n 的值较大时, 会发生什么? 答案: <http://thinkpython2.com/code/ackermann.py>。

6.11.3 习题 6-3

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

下面的函数接受一个字符串实参, 并返回第一个、最后一个和中间的字母:

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

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

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

在第八章中我们将介绍他们是如何工作的。

1. 将它们录入到文件 `palindrome.py` 中并测试。当你用一个两个字母的字符串调用 `middle` 时会发生什么? 一个字母的呢? 空字符串呢? 空字符串这样 `''` 表示, 中间不含任何字母。
2. 编写一个叫 `is_palindrome` 的函数, 接受一个字符串作为实参。如果是回文词, 就返回 `True`, 反之则返回 `False`。记住, 你可以使用内建函数 `len` 来检查字符串的长度。

答案: http://thinkpython2.com/code/palindrome_soln.py。

6.11.4 习题 6-4

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

6.11.5 习题 6-5

a 和 b 的最大公约数 (GCD) 是能被二者整除的最大数。

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

6.11.6 习题 6-6

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

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

6.11.7 贡献者

1. 翻译: @theJian
2. 校对: @bingjin
3. 参考: @carfly

第七章：迭代

本章介绍迭代，即重复运行某个代码块的能力。我们已经在[递归](#)一节接触了一种利用递归进行迭代的方式；在[简单的重复](#)一节中，接触了另一种利用 `for` 循环进行迭代的方式。在本章中，我们将讨论另外一种利用 `while` 语句实现迭代的方式。不过，首先我想再多谈谈有关变量赋值的问题。

7.1 重新赋值

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

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

第一次打印 `x` 时，它的值为 5；第二次打印时，它的值是 7。

图 7-1 状态图展示了 **重新赋值** 在状态图中看起来是什么样子。

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

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

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

```
>>> a = 5
>>> b = a    # a 5 b 55555
>>> a = 3    # a 3 b 55555
>>> b
5
```

第三行改变了 `a` 的值，但是没有改变 `b` 的值，所以它们不再相等了。

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

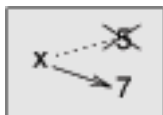


图 7.1: 图 7-1 状态图

7.2 更新变量

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

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

这个语句的意思是，“获得 x 的当前值，加 1，然后将 x 的值更新为新的值。”

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

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

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

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

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

7.3 while 语句

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

所以 Python 提供了使其更容易实现的语言特性。其中之一就是我们在 [简单的重复](#) 一节看到的我们已经见过两个利用递归来迭代的函数：`countdown` 和 `print_n`。由于迭代如此普遍，所以 Python 提供了使其更容易实现的语言特性。其中之一就是我们在 [简单的重复](#) 一节看到的 `for` 语句。后面我们还会继续介绍。

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

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

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

更正式地说，`while` 语句的执行流程如下：

1. 首先判断条件为真还是为假。
2. 如果为假，退出 `while` 语句，然后执行接下来的语句；

3. 如果条件为真，则运行 `while` 语句体，运行完再返回第一步；

这种形式的流程叫做循环（loop），因为第三步后又循环回到了第一步。

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

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

有些其他循环，可能就没那么好理解了。例如：

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

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

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

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

难点是能否证明程序对于所有的正整数 `n` 都会终止的。目前为止，还没有人证明或者证伪该命题。（见：http://en.wikipedia.org/wiki/Collatz_conjecture。）

我们做个练习，利用迭代而非递归，重写之前递归一节中的 `print_n` 函数。

7.4 break

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

例如，假设你想从用户那里获取输入，直到用户键入“done”。你可以这么写：

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

print('Done!')
```

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

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

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

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

7.5 平方根

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

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

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

例如，假定 a 是 4， x 是 3：

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

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

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

再通过多几次的运算，这个估算可以说已经是很精确了。

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

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

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

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

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

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

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

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

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

7.6 算法

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

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

不过，如果你比较“懒”，你可能就会找到一些诀窍。比如说为了计算 n 和 9 的乘积，你可以把 $n - 1$ 作为乘积的第一位数，再把 $10 - n$ 作为第二位数，从而得到它们的乘积。这个诀窍是将任意个位数与 9 相乘的普遍解法。这就是一种算法。

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

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

人们轻轻松松或者下意识自然而然做的一些事情，往往是最难用算法来表达的。理解自然语言就是个很好的例子。我们每个人都听得懂自然语言，但是目前还没有人能够解释我们是怎么做到的，至少不是以算法的形式解释。

7.7 调试

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

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

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

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

每次都这样检查，就可以将需要搜索的代码行数减少一半。经过 6 步之后（这比 100 小多了），你将会找到那或者两行出错的代码，至少理论上是这样。

在实践中，可能并不能很好的确定程序的“中间部分”是什么，也有可能并不是那么好检查。计算行数并且取其中间行是没有意义的。相反，多考虑下程序中哪些地方比较容易出问题，或者哪些地方比较容易进行检查。然后选定一个检查点，在这个断点前后出现 **bug** 的概念差不多。

7.8 术语表

重新赋值 (reassignment):

给已经存在的变量赋一个新的值。

更新 (update):

变量的新值取决于旧值的一种赋值方法。

初始化 (initialize):

给后面将要更新的变量一个初始值的一种赋值方法。

递增 (increment):

通过增加变量的值的方式更新变量（通常是加 1）。

递减 (decrement):

通过减少变量的值的方式来更新变量。

迭代 (iteration):

利用递归或者循环的方式来重复执行代一组语句的过程。

无限循环 (infinite loop):

无法满足终止条件的循环。

算法 (algorithm):

解决一类问题的通用过程。

7.9 练习题

7.9.1 习题 7-1

复制平方根一节中的循环，将其封装进一个叫 `mysqrt` 的函数中。这个函数接受 `a` 作为形参，选择一个合适的 `x` 值，并返回 `a` 的平方根估算值。

为测试上面的函数，编写一个名为 `test_sqrt_root` 的函数，打印出如下表格：

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

其中第一列是 `a` 的值；第二列是通过 `mysqrt` 计算得到的 `a` 的平方根；第三列是用 `math.sqrt` 计算得到的平方根；第四列则是这两个平方根之差的绝对值。

7.9.2 习题 7-2

内置函数 `eval` 接受一个字符串，并使用 Python 解释器来计算该字符串。例如：

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

编写一个名为 `eval_loop` 的函数，迭代式地提示用户输入，获取输入的内容，并利用 `eval` 来计算其值，最后打印该值。

该程序应持续运行，知道用户输入 'done'，然后返回它最后一次计算的表达式的值。

7.9.3 习题 7-3

数学家斯里尼瓦瑟·拉马努金 (Srinivasa Ramanujan) 发现了一个可以用来生成 $1/\pi$ 近似值的无穷级数 (infinite series)：

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

编写一个名为 `estimate_pi` 的函数，利用上面公式来估算并返回 π 的值。这个函数应该使用 `while` 循环来计算所有项的和，直到最后一项小于 $1e-15$ (Python 中用于表达 10^{-15} 的写法) 时终止循环。你可以将该值与 `math.pi` 进行比较，检测是否准确。

答案: <http://thinkpython2.com/code/pi.py>。

7.9.4 贡献者

1. 翻译: @lroolle
2. 校对: @bingjin
3. 参考: @carfly

第八章：字符串

字符串不像整数、浮点数和布尔型。字符串是一个 **序列 (sequence)**，这就意味着它是其他值的一个有序的集合。在这章中，你将学习怎么去访问字符串里的字符，同时你也会学习到字符串提供的一些方法。

8.1 字符串是一个序列

字符串是由字符组成的序列。你可以用括号运算符一次访问一个字符：

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

第 2 条语句从 `fruit` 中选择索引为 1 的字符并将它赋给 `letter`。

括号中的表达式被称作 **索引 (index)**。索引指出在序列中你想要哪个字符 (因此而得名)。

但是你可能不会获得你期望的东西：

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

对于大多数人，`'banana'` 的第一个字母是 `b` 而不是 `a`。但是对于计算机科学家，索引是从字符串起点开始的位移量 `offset`，第一个字母的位移量就是 0。

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

所以 `b` 是 `'banana'` 的第 0 个字母 (“zero-eth”)，`a` 是第一个字母 (“one-eth”)，`n` 是第二个字母 (“two-eth”)。

你可以使用一个包含变量名和运算符的表达式作为索引：

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

索引值必须使用整数。否则你会得到:

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

8.2 len

`len` 是一个内建函数，其返回字符串中的字符数量:

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

为了获得一个字符串的最后一个字符，你可能会尝试像这样操作:

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

出现 `IndexError` 的原因，是在 `'banana'` 中没有索引为 6 的字母。由于我们从 0 开始计数，六个字母的编号是 0 到 5。为了获得最后一个字符，你必须将 `length` 减一:

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

或者你可以使用负索引，即从字符串的结尾往后数。表达式 `fruit[-1]` 返回最后一个字母，`fruit[-2]` 返回倒数第二个字母，以此类推。

8.3 使用 for 循环遍历

许多计算中需要一个字符一个字符地处理字符串。通常计算从字符串的头部开始，依次选择每个字符，对其做一些处理，然后继续直到结束。这种处理模式被称作 **遍历 (traversal)**。编写遍历的方法之一是使用 `while` 循环:

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

该循环遍历字符串并在每行显示一个字符串。该循环的条件是 `index < len(fruit)`，所以当 `index` 和字符串的长度相等时，条件为假，循环体不被执行。被访问的最后一个字符的索引为 `len(fruit)-1`，这也是字符串的最后一个字符。

我们做个练习，编写一个函数，接受一个字符串作为实参，按照从后向前的顺序显示字符，每行只显示一个。

编写遍历的另一种方法是使用 `for` 循环:

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

每次循环时，字符串中的下一个字符被赋值给变量 `letter`。循环继续，直到没有剩余的字符串了。

下面的例子演示了如何使用拼接（字符串相加）和 `for` 循环生成一个字母表序列（即按照字母表顺序排列）。在 Robert McCloskey 的书《*Make Way for Ducklings*》中，小鸭子的名字是 Jack、Kack、Lack、Mack、Nack、Ouack、Pack 和 Quack。此循环按顺序输出这些名字：

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

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

输出是：

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

当然，输出并不完全正确，因为“Ouack”和“Quack”拼写错了。我们做个练习，修改这个程序，解决这个问题。

8.4 字符串切片

字符串的一个片段被称作 **切片 (slice)**。选择一个切片的操作类似于选择一个字符：

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

`[n:m]` 操作符返回从第 `n` 个字符到第 `m` 个字符的字符串片段，包括第一个，但是不包括最后一个。这个行为违反直觉，但是将指向两个字符之间的索引，想象成图 8-1：切片索引中那样或许有帮助。

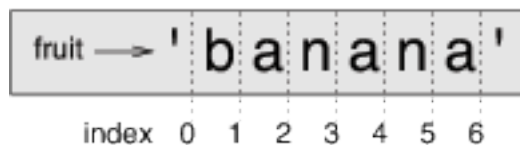


图 8.1: 切片索引

如果你省略第一个索引 (冒号前面的值), 切片起始于字符串头部。如果你省略第二个索引, 切片一直到字符串结尾:

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

如果第一个索引大于或等于第二个, 结果是空字符串 (**empty string**), 用两个引号表示:

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

一个空字符串不包括字符而且长度为 0, 但除此之外, 它和其它任何字符串一样。

继续这个例子, 你认为 `fruit[:]` 的结果是什么? 尝试运行看看。

8.5 字符串是不可变的

你会很想在赋值语句的左边使用 `[]`, 来改变字符串的一个字符。例如:

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

错误信息中的 “object (对象)” 是那个字符串, “item(元素)” 是你赋值给它的字符。目前, 我们认为对象 (object) 和值是同样的东西, 但是我们后面将改进此定义 (详见 “对象与值” 一节)。

出现此错误的原因是字符串是 **不可变的 (immutable)**, 这意味着你不能改变一个已存在的字符串。你最多只能创建一个新的字符串, 在原有字符串的基础上略有变化:

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

上面的示例中, 我们将一个新的首字母拼接到 `greeting` 的一个切片上。它不影响原字符串。

8.6 搜索

下面的函数起什么作用?

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

```

    index = index + 1
    return -1

```

在某种意义上，`find` 和 `[]` 运算符相反。与接受一个索引并提取相应的字符不同，它接受一个字符并找到该字符所在的索引。如果没有找到该字符，函数返回 `-1`。

这是我们第一次在循环内部看见 `return` 语句。如果 `word[index] == letter`，函数停止循环并马上返回。

如果字符没出现在字符串中，那么程序正常退出循环并返回 `-1`。

这种计算模式——遍历一个序列并在找到寻找的东西时返回——被称作 **搜索 (search)**。

我们做个练习，修改 `find` 函数使得它接受第三个参数，即从何处开始搜索的索引。

8.7 循环和计数

下面的程序计算字母 `a` 在字符串中出现的次数：

```

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

```

此程序演示了另一种被称作 **计数器 (counter)** 的计算模式。变量 `count` 初始化为 `0`，然后每次出现 `a` 时递增。当循环结束时，`count` 包含了字母 `a` 出现的总次数。

我们做一个练习，将这段代码封装在一个名为 `count` 的函数中，并泛化该函数，使其接受字符串和字母作为实参。

然后重写这个函数，不再使用字符串遍历，而是使用上一节中三参数版本的 `find` 函数。

8.8 字符串方法

字符串提供了可执行多种有用操作的方法 (**method**)。方法和函数类似，接受实参并返回一个值，但是语法不同。例如，`upper` 方法接受一个字符串，并返回一个都是大写字母的新字符串。

不过使用的不是函数语法 `upper(word)`，而是方法的语法 `word.upper()`。

```

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

```

点标记法的形式指出方法的名字，`upper`，以及应用该方法的字符串的名字，`word`。空括号表明该方法不接受实参。

这被称作 **方法调用 (invocation)**；在此例中，我们可以说是在 `word` 上调用 `upper`。

事实上，有一个被称为 `find` 的字符串方法，与我们之前写的函数极其相似：

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

此例中，我们在 `word` 上调用 `find`，并将我们要找的字母作为参数传入。

事实上，`find` 方法比我们的函数更通用；它还可以查找子字符串，而不仅仅是字符：

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

`find` 默认从字符串的首字母开始查找，它还可以接受第二个实参，即从何处开始的索引。

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

这是一个 **可选参数 (optional argument)** 的例子；`find` 也可以接受结束查找的索引作为第三个实参：

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

此次搜索失败，因为 `'b'` 没有出现在索引 1-2 之间（不包括 2）。一直搜索到第二个索引，但是并不搜索第二个索引，这使得 `find` 跟切片运算符的行为一致。

8.9 in 运算符

单词 `in` 是一个布尔运算符，接受两个字符串。如果第一个作为子串出现在第二个中，则返回 `True`：

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

例如，下面的函数打印所有既出现在 `word1` 中，也出现在 `word2` 中的字母：

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

变量名挑选得当的话，Python 代码有时候读起来像是自然语言。你可以这样读此循环，“对于 (每个) 在 (第一个) 单词中的字母，如果 (该) 字母 (出现) 在 (第二个) 单词中，打印 (该) 字母”。

如果你比较 `'apples'` 和 `'oranges'`，你会得到下面的结果：

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

8.10 字符串比较

关系运算符也适用于字符串。可以这样检查两个字符串是否相等：

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

其它的关系运算符对于按字母序放置单词也很有用：

```
if word < 'banana':
    print('Your word☹ ' + word + '☹ comes before banana.')
```

```
elif word > 'banana':
    print('Your word☹ ' + word + '☹ comes after banana.')
```

```
else:
    print('All right☹ bananas.')
```

Python 处理大写和小写字母的方式和人不同。所有的大写字母出现在所有小写字母之前，所以：

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

解决此问题的常见方式是，在执行比较之前，将字符串转化为标准格式，例如都是小写字母。请牢记这点，万一你不得不防卫一名手持菠萝男子的袭击呢。

8.11 调试

当你使用索引遍历序列中的值时，正确地指定遍历的起始和结束点有点困难。下面是一个用来比较两个单词的函数，如果一个单词是另一个的倒序，则返回 `True`，但其中有两个错误：

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

    i = 0
    j = len(word2)

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

    return True
```

第一条 `if` 语句检查两个单词是否等长。如果不是，我们可以马上返回 `False`。否则，在函数其余的部分，我们可以假定单词是等长的。这是[检查类型](#)一节中提到的监护人模式的一个例子。

`i` 和 `j` 是索引：`i` 向前遍历 `word1`，`j` 向后遍历 `word2`。如果我们找到两个不匹配的字母，我们可以立即返回 `False`。如果我们完成整个循环并且所有字母都匹配，我们返回 `True`。

如果我们用单词 “pots” 和 “stop” 测试该函数，我们期望返回 `True`，但是却得到一个 `IndexError`：

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

为了调试该类错误，我第一步是在错误出现的行之前，打印索引的值。

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

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

现在，当我再次运行该程序时，将获得更多的信息：

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

第一次循环时，`j` 的值是 4，超出字符串 'post' 的范围了。最后一个字符的索引是 3，所以 `j` 的初始值应该是 `len(word2)-1`。

如果我解决了这个错误，然后运行程序，将获得如下输出：

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

这次我们获得了正确的答案，但是看起来循环只运行了三次，这很奇怪。画栈图可以帮我们更好的理解发生了什么。在第一次迭代期间，`is_reverse` 的栈帧如图 8-2：堆栈图所示。

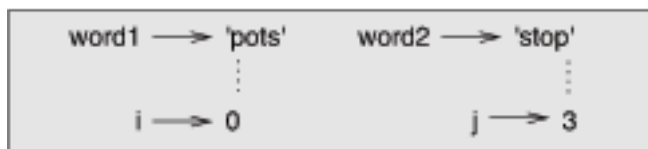


图 8.2: 图 8-2: 堆栈图

我对堆栈图做了些调整，重新排列了栈帧中的变量，增加了虚线来说明 `i` 和 `j` 的值表示 `word1` 和 `word2` 中的字符。

从这个堆栈图开始，在纸上运行程序，每次迭代时修改 `i` 和 `j` 的值。查找并解决这个函数的中第二个错误。

8.12 术语表

对象 (object):

变量可以引用的东西。现在你将对象和值等价使用。

序列 (sequence):

一个有序的值的集合，每个值通过一个整数索引标识。

元素 (item):

序列中的一个值。

索引 (index):

用来选择序列中元素（如字符串中的字符）的一个整数值。在 Python 中，索引从 0 开始。

切片 (slice):

以索引范围指定的字符串片段。

空字符串 (empty string):

一个没有字符的字符串，长度为 0，用两个引号表示。

不可变 (immutable):

元素不能被改变的序列的性质。

遍历 (traversal):

对一个序列的所有元素进行迭代，对每一元素执行类似操作。

搜索 (search):

一种遍历模式，当找到搜索目标时就停止。

计数器 (counter):

用来计数的变量，通常初始化为 0，并以此递增。

方法调用 (invocation): 执行一个方法的声明。

可选参数 (optional argument):

一个函数或者一个方法中不必要指定的参数。

8.13 练习题

8.13.1 习题 8-1

点击如下链接，阅读字符串方法的文档 <http://docs.python.org/3/library/stdtypes.html#string-methods>。为了确保你理解他们是怎么工作的，可以尝试使用其中的一些方法。`strip` 和 `replace` 尤其有用。

文档中使用了可能会引起困惑的句法。例如，在 `find(sub[start:end])` 中，方括号意味着这是可选参数。所以，`sub` 是必填参数，但是 `start` 是可选的，而且如果你提供了 `start`，也不一定必须提供 `end`。

8.13.2 习题 8-2

有一个字符串方法叫 `count`，它类似于之前[循环和计数](#)一节中的 `counter`。阅读这个方法的文档，写一个计算 `'banana'` 中 `a` 的个数的方法调用。

8.13.3 习题 8-3

一个字符串切片可以接受指定步长的第三个索引；也就是连续字符间空格的个数。步长为 2，意味着每隔一个字符；步长为 3，意味着每隔两个字符，以此类推。

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

步长为-1就是从单词的尾部开始进行，所以切片 `[::-1]` 生成一个倒序的字符串。

利用这个惯用法 (idiom)，将习题 6-3 中 `is_palindrome` 函数改写为一行代码版。

8.13.4 习题 8-4

下面这些函数，都是用于检查一个字符串是否包含一些小写字母的，但是其中至少有一些是错误的函数。检查每个函数，描述这个函数实际上做了什么 (假设形参是字符串)。

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

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

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

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

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

8.13.5 习题 8-5

凯撒密码是一种弱加密方式，它将每一个字母偏移固定的位置。偏移一个字母，指的是按着字母表偏移，如果需要的话再从尾部跳转至首字母，所以“A”偏移三个位置即为“D”，“Z”偏移一个位置是“A”。

要偏移一个单词，可以将其中每一个字母偏移相同的量。例如，“cheer”偏移 7 个位置后变成了“jolly”，“melon”偏移-10 个位置变成了“cubed”。在电影 *《2001：太空奥德赛》（2001: A Space Odyssey）* 中，飞船上的电脑叫做 HAL，也就是 IBM 偏移 1 个位置后的单词。

编写一个叫 `rotate_word` 的函数，接受一个字符串和一个整数作为形参，并返回原字符串按照给定整数量偏移后得到的一个新字符串。

你可能想用内置函数 `ord`，它可以将字符转化成数值代码，还有 `chr`，它可以将数值代码转化成字符。字母表的字母以字母表顺序编码，例如：

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

因为 'c' 是字母表中的第二个字母。但是请注意：大写字母的数值代码是不同的。

网上一些可能冒犯人的笑话有时以 ROT13 编码，即以 13 为偏移量的凯撒密码。如果你不是很容易就被冒犯，那么可以找些这样的笑话，并解码。答案：<http://thinkpython2.com/code/rotate.py>。

8.13.6 贡献者

1. 翻译：@xpgeng
2. 校对：@bingjin
3. 参考：@carfly

第九章：文字游戏

这一章将介绍第二个案例研究，即通过查找具有特定属性的单词来解答字谜游戏。例如，我们将找出英文中最长的回文单词，以及字符按照字符表顺序出现的单词。另外，我还将介绍另一种程序开发方法：简化为之前已解决的问题。

9.1 读取单词列表

为了完成本章的习题，我们需要一个英语单词的列表。网络上有许多单词列表，但是最符合我们目的列表之一是由 Grady Ward 收集并贡献给公众的列表，这也是 Moby 词典项目的一部分（见：http://wikipedia.org/wiki/Moby_Project）。它由 113,809 个填字游戏单词组成，即在填字游戏以及其它文字游戏中被认为有效的单词。在 Moby 集合中，该列表的文件名是 113809of.fic；你可以从<http://thinkpython.com/code/words.txt> 下载一个拷贝，文件名已被简化为 words.txt。

该文件是纯文本，因此你可以用一个文本编辑器打开它，但是你也可以从 Python 中读取它。内建函数 `open` 接受文件名作为形参，并返回一个 **文件对象（file object）**，你可以使用它读取该文件。

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

`fin` 是输入文件对象的一个常用名。该文件对象提供了几个读取方法，包括 `readline`，其从文件中读取字符直到碰到新行，并将结果作为字符串返回：

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

在此列表中，第一个单词是“aa”，它是一种岩浆。序列 `\r\n` 代表两个空白字符，回车和换行，它们将这个单词和下一个分开。

此文件对象跟踪它在文件中的位置，所以如果你再次调用 `readline`，你获得下一个单词：

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

下一个单词是“aah”，它是一个完全合法的单词，所以不要那样看我。或者，如果空格困扰了你，我们可以用字符串方法 `strip` 删掉它：

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

你也可以将文件对象用做 for 循环的一部分。此程序读取 `words.txt` 并打印每个单词，每行一个：

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

9.2 练习

下一节给出了这些习题的答案。在你看这些答案之前，应该至少试着解答一下。

9.2.1 习题 9-1

编写写一个程序，使得它可以读取 `words.txt`，然后只打印出那些长度超过 20 个字符的单词 (不包括空格)。

9.2.2 习题 9-2

1939 年，Ernest Vincent Wright 出版了一本名为《*Gadsby*》的小说，该小说里完全没有使用字符“e”。由于“e”是最常用的英文字符，因此这并不容易做到。

事实上，不使用这个最常用的符号 (字符 e) 来构建一个孤立的想法是很难的。开始进展缓慢，但是经过有意识的、长时间的训练，你可以逐渐地熟练。

好啦，不再说题外话了（让我们开始编程练习）。

写一个叫做 `has_no_e` 的函数，如果给定的单词中不包含字符“e”，其返回 `True`。

修改上一节中的程序，只打印不包含“e”的单词，并且计算列表中不含“e”单词的比例。

9.2.3 习题 9-3

编写一个名为 `avoids` 的函数，接受一个单词和一个指定禁止使用字符的字符串，如果单词中不包含任意被禁止的字符，则返回 `True`。

修改你的程序，提示用户输入一个禁止使用的字符，然后打印不包含这些字符的单词的数量。你能找到一个 5 个禁止使用字符的组合，使得其排除的单词数目最少么？

9.2.4 习题 9-4

编写一个名为 `uses_only` 的函数，接受一个单词和一个字符串。如果该单词只包括此字符串中的字符，则返回 `True`。你能只用 `acefhlo` 这几个字符造一个句子么？除了“*Hoe alfalfa*”外。

9.2.5 习题 9-5

编写一个名为 `uses_all` 的函数，接受一个单词和一个必须使用的字符组成的字符串。如果该单词包括此字符串中的全部字符至少一次，则返回 `True`。你能统计出多少单词包含了所有的元音字符 `aeiou` 吗？如果换成 `aeiouy` 呢？

9.2.6 习题 9-6

编写一个名为 `is_abecedarian` 的函数，如果单词中的字符以字符表的顺序出现（允许重复字符），则返回 `True`。有多少个具备这种特征的单词？

9.3 搜索

前一节的所有习题有一个共同点：都可以用在[搜索](#)一节中看到的搜索模式解决。举一个最简单的例子：

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

`for` 循环遍历 `word` 中的字符。如果我们找到字符“e”，那么我们可以马上返回 `False`；否则我们必须检查下一个字符。如果我们正常退出循环，就意味着我们没有找到一个“e”，所以我们返回 `True`。

你也可以用 `in` 操作符简化上述函数，但是我之所以一开始写成这样，是因为它展示了搜索模式的逻辑。

`avoid` 是一个更通用的 `has_no_e` 函数，但是结构是相同的：

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

一旦我们找到一个禁止使用的字符，我们返回 `False`；如果我们到达循环结尾，我们返回 `True`。

除了检测条件相反以外，下面 `uses_only` 函数与上面的函数很像：

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

这里我们传入一个允许使用字符的列表，而不是禁止使用字符的列表。如果我们在 `word` 中找到一个不在 `available` 中的字符，我们就可以返回 `False`。

除了将 `word` 与所要求的字符的角色进行了调换之外，下面的 `uses_all` 函数也是类似的。

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

该循环遍历需要的字符，而不是遍历 `word` 中的字符。如果任何要求的字符没出现在单词中，则我们返回 `False`。

如果你真的像计算机科学家一样思考，你可能已经意识到 `uses_all` 是前面已经解决的问题的一个实例，你可能会写成：

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

这是一种叫做简化为之前已解决的问题（**reduction to a previously solved problem**）的程序开发方法的一个示例，也就是说，你认识到当前面临的问题是之前已经解决的问题的一个实例，然后应用了已有的解决方案。

9.4 使用索引进行循环

前一节我用 `for` 循环来编写函数，因为我只需要处理字符串中的字符；我不必用索引做任何事情。

对于下面的 `is_abecedarian`，我们必须比较邻接的字符，用 `for` 循环来写的话有点棘手。

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

一种替代方法是使用递归：

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

另一中方法是使用 `while` 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```


循环起始于 $i=0$ ， $i=\text{len}(\text{word})-1$ 时结束。每次循环，函数会比较第 i 个字符（可以将其认为是当前字符）和第 $i+1$ 个字符（可以将其认为是下一个字符）。

如果下一个字符比当前的小（字符序靠前），那么我们在递增趋势中找到了断点，即可返回 `False`。

如果到循环结束时我们也没有找到一点错误，那么该单词通过测试。为了让你相信循环正确地结束了，我们用 `'flossy'` 这个单词来举例。它的长度为 6，因此最后一次循环运行时， i 是 4，这是倒数第 2 个字符的索引。最后一次迭代时，函数比较倒数第二个和最后一个字符，这正是我们希望的。

下面是 `is_palindrome` 函数的一种版本（详见习题 6-3），其中使用了两个索引：一个从最前面开始并往前上，另一个从最后面开始并往下走。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i < j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

或者，我们可以把问题简化为之前已经解决的问题，这样来写：

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

使用图 8-2：堆栈图中描述的 `is_reverse`。

9.5 调试

程序测试很困难。本章中介绍的函数相对容易测试，因为你可以手工检查结果。即使这样，选择一可以测试所有可能错误的单词集合，是很困难的，介于困难和不可能之间。

以 `has_no_e` 为例，有两个明显的用例需要检查：含有 ‘e’ 的单词应该返回 `False`，不含的单词应该返回 `True`。你应该可以很容易就能想到这两种情况。

在每个用例中，还有一些不那么明显的子用例。在含有 “e” 的单词中，你应该测试 “e” 在开始、结尾以及在中间的单词。你还应该测试长单词、短单词以及非常短的单词，如空字符串。空字符串是一个**特殊用例（special case）**，及一个经常出现错误的不易想到的用例。

除了你生成的测试用例，你也可以用一个类似 `words.txt` 中的单词列表测试你的程序。通过扫描输出，你可能会捕获错误，但是请注意：你可能捕获一类错误（包括了不应该包括的单词）却没能捕获另一类错误（没有包括应该包括的单词）。

一般来讲，测试能帮助你找到错误，但是生成好的测试用例并不容易，并且即便你做到了，你仍然不能保证你的程序是正确的。正如一位传奇计算机科学家所说：

程序测试能用于展示错误的存在，但是无法证明不存在错误！

—Edsger W. Dijkstra

9.6 术语表

文件对象 (file object) :

代表打开文件的变量。

简化为之前已经解决的问题:

通过把未知问题简化为已经解决的问题来解决问题的方法。

特殊用例 (special case) :

一种不典型或者不明显的测试用例 (而且很可能无法正确解决的用例)。

9.7 练习题

9.7.1 习题 9-7

这个问题基于广播节目《Car Talk》(<http://www.cartalk.com/content/puzzlers>) 上介绍的一个字谜:

找出一个包含三个连续双字符的单词。我将给你一系列几乎能够符合条件但实际不符合的单词。比如, committee 这个单词, c-o-m-m-i-t-t-e-e。如果中间没有 i 的话, 就太棒了。或者 Mississippi 这个单词: M-i-s-s-i-s-s-i-p-p-i。假如将这些 i 剔除出去, 就会符合条件。但是确实存在一个包含三个连续的单词对, 而且据我了解, 它可能是唯一符合条件的单词。当然也可能有 500 多个, 但是我只能想到一个。那么这个单词是什么?

编写一个程序, 找到这个单词。答案: <http://thinkpython2.com/code/cartalk1.py>。

9.7.2 习题 9-8

下面是另一个来自《Car Talk》的谜题 (<http://www.cartalk.com/content/puzzlers>) :

“有一天, 我正在高速公路上开车, 我偶然注意到我的里程表。和大多数里程表一样, 它只显示 6 位数字的整数英里数。所以, 如果我的车开了 300,000 英里, 我能够看到的数字是:3-0-0-0-0-0。

我当天看到的里程数非常有意思。我注意到后四位数字是回文数; 也就是说, 正序读和逆序读是一样的。例如, 5-4-4-5 就是回文数。所以我的里程数可能是 3-1-5-4-4-5。

一英里后, 后五位数字变成了回文数。例如, 里程数可能变成了是 3-6-5-4-5-6。又过了一英里后, 6 位数字的中间四位变成了回文数。你相信吗? 一英里后, 所有的 6 位数字都变成了回文数。

那么问题来了, 当我第一次看到里程表时, 里程数是多少?”

编写一个程序, 测试所有的 6 位数字, 然后输出所有符合要求的结果。答案: <http://thinkpython2.com/code/cartalk2.py>。

9.7.3 习题 9-9

还是《Car Talk》的谜题 (<http://www.cartalk.com/content/puzzlers>)，你可以通过利用搜索模式解答：

“最近我探望了我的妈妈，我们忽然意识到把我的年纪数字反过来就是她的年龄。比如，如果她 73 岁，那么我就是 37 岁。我们想知道过去这些年来，发生了多少次这样的巧合，但是我们很快偏离到其他话题上，最后并没有找到答案。

回到家后，我计算出我的年龄数字有 6 次反过来就是妈妈的年龄。同时，我也发现如果幸运的话，将来几年还可能发生这样的巧合，运气再好点的话，之后还会出现一次这样的巧合。换句话说，这样的巧合一共会发生 8 次。那么，问题来了，我现在多大了？”

编写一个查找谜题答案的 Python 函数。提示：字符串的 `zfill` 方法特别有用。答案：<http://thinkpython2.com/code/cartalk3.py>。

9.7.4 贡献者

1. 翻译：@iphyer
2. 校对：@bingjin
3. 参考：@carfly

第十章：列表

本章介绍 Python 中最有用的内置类型之一：列表（list）。你还将进一步学习关于对象的知识以及同一个对象拥有多个名称时会发生什么。

10.1 列表是一个序列

与字符串类似，**列表**是由多个值组成的序列。在字符串中，每个值都是字符；在列表中，值可以是任何数据类型。列表中的值称为 **元素（element）**，有时也被称为 **项（item）**。

创建新列表的方法有多种；最简单的方法是用方括号（`[` 和 `]`）将元素包括起来：

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

第一个例子是包含 4 个整数的列表。第二个是一个包含 3 个字符串的列表。一个列表中的元素不需要是相同的数据类型。下面的列表包含一个字符串、一个浮点数、一个整数和另一个列表：

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

一个列表在另一个列表中，称为 **** 嵌套（nested）列表 ****。

一个不包含元素的列表被称为空列表；你可以用空的方括号 `[]` 创建一个空列表。

正如你想的那样，你可以将列表的值赋给变量：

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

10.2 列表是可变的

访问列表中元素的语法，与访问字符串中字符的语法相同，都是通过方括号运算符实现的。括号中的表达式指定了元素的索引。记住，索引从 0 开始：

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

和字符串不同的是，列表是可变的。当括号运算符出现在赋值语句的左边时，它就指向了列表中将被赋值的元素。

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

`numbers` 中索引为 1 的元素，原来是 123，现在变成了 5。

图 10-1: 状态图 是 `cheeses`、`numbers` 和 `empty` 的状态图。

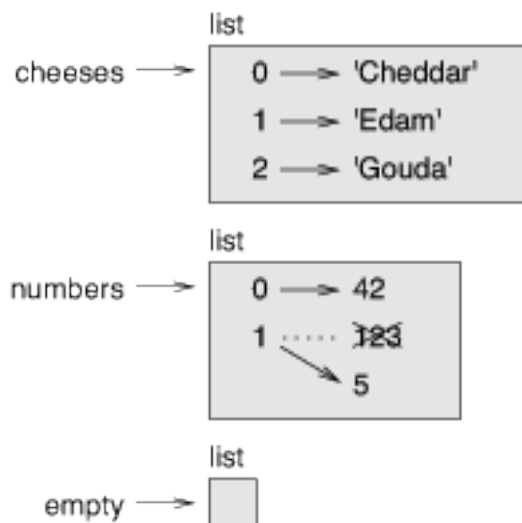


图 10.1: 图 10-1: 状态图

列表用外部标有“list”的盒子表示，盒子内部是列表的元素。`cheeses` 指向一个有 3 个元素的列表，3 个元素的下标分别是 0、1、2。`numbers` 包含两个元素；状态图显示第二个元素原来是 123，被重新赋值为 5。`empty` 对应一个没有元素的列表。

列表下标的工作原理和字符串下标相同：

- 任何整数表达式都可以用作下标。
- 如果你试图读或写一个不存在的元素，你将会得到一个索引错误 (`IndexError`)。
- 如果下标是负数，它将从列表的末端开始访问列表。

`in` 运算符在列表中同样可以使用。

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

10.3 遍历列表

最常用的遍历列表的方式是使用 `for` 循环。语法和字符串遍历类似：

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

如果你只需要读取列表中的元素，这种方法已经足够。然而，如果你想要写入或者更新列表中的元素，你需要通过下标访问。一种常用的方法是结合内置函数 `range` 和 `len`：

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

这个循环将遍历列表并更新每个元素。`len` 返回列表中的元素个数。`range` 返回一个包含从 0 到 $n - 1$ 下标的列表，其中 n 是列表的长度。每次循环中，`i` 得到下一个元素的下标。循环主体中的赋值语句使用 `i` 读取该元素的旧值，并赋予其一个新值。

对一个空列表执行 `for` 循环时，将不会执行循环的主体：

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

尽管一个列表可以包含另一个列表，嵌套的列表本身还是被看作一个单个元素。下面这个列表的长度是 4：

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

10.4 列表操作

+ 运算符拼接多个列表：

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

运算符 * 以给定次数的重复一个列表：

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

第一个例子重复 4 次。第二个例子重复了那个列表 3 次。

10.5 列表切片

切片（slice）运算符同样适用于对列表：

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

如果你省略第一个索引, 切片将从列表头开始。如果你省略第二个索引, 切片将会到列表尾结束。所以如果你两者都省略, 切片就是整个列表的一个拷贝。

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

由于列表是可变的, 通常在修改列表之前, 对列表进行拷贝是很有用的。

切片运算符放在赋值语句的左边时, 可以一次更新多个元素:

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

10.6 列表方法

Python 为列表提供了一些方法. 例如, `append` 添加一个新元素到列表的末端:

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

`extend` 将接受一个列表作为参数, 并将其其中的所有元素添加至目标列表中:

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

这个例子中 `t2` 没有改动。

`sort` 将列表中的元素从小到大进行排序:

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

大部分的列表方法都是无返回值的; 它们对列表进行修改, 然后返回 `None`。如果你意外的写了 `t.sort()`, 你将会对结果感到失望的。

10.7 映射、筛选和归并

你可以这样使用循环，对列表中所有元素求和：

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

`total` 被初始化为 0。每次循环时, `x` 从列表中获取一个元素。运算符 `+=` 提供了一个快捷的更新变量的方法。这个 **增量赋值语句 (augmented assignment statement)**

```
total += x
```

等价于

```
total = total + x
```

当循环执行时, `total` 将累计元素的和；一个这样的变量有时被称为 **累加器 (accumulator)**。

把一个列表中的元素加起来是一个很常用的操作，所以 Python 将其设置为一个内建内置函数 `sum`：

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

一个像这样的将一系列的元素合并成一个单一值的操作有时称为 **归并 (reduce)**。

有时，你在构建一个列表时还需要遍历另一个列表。例如，下面的函数接受一个字符串列表作为参数，返回包含大写字符的新列表：

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

`res` 被初始化为一个空列表；每次循环时，我们添加下一个元素。所以 `res` 是另一种形式的累加器。

类似 `capitalize_all` 这样的操作有时被称为 **映射 (map)**，因为它“映射”一个函数（在本例中是方法 `capitalize`）到序列中的每个元素上。

另一个常见的操作是从列表中选择一些元素，并返回一个子列表。例如，下面的函数读取一个字符串列表，并返回一个仅包含大写字符串的列表：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` 是一个字符串方法，如果字符串仅含有大写字母，则返回 `True`。

类似 `only_upper` 这样的操作被称为 **筛选（filter）**，因为它选中某些元素，然后剔除剩余的元素。

大部分常用列表操作可以用映射、筛选和归并这个组合表示。

10.8 删除元素

有多种方法可以从列表中删除一个元素。如果你知道元素的下标，你可以使用 `pop`：

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

`pop` 修改列表，并返回被移除的元素。如果你不提供下标，它将移除并返回最后一个元素。

如果你不需要被移除的元素，可以使用 `del` 运算符：

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

如果你知道要删除的值（但是不知道其下标），你可以使用 `remove`：

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

`remove` 的返回值是 `None`。

要移除多个元素，你可以结合切片索引使用 `del`：

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

同样的，切片选择到第二个下标（不包含第二个下标）处的所有元素。

10.9 列表和字符串

一个字符串是多个字符组成的序列，一个列表是多个值组成的序列。但是一个由字符组成的列表不同于字符串。可以使用 `list` 将一个字符串转换为字符的列表：

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

由于 `list` 是内置函数的名称，你应避免将它用作变量名。我同样避免使用 `l`，因为它看起来很像 `1`。这就是为什么我用了 `t`。

`list` 函数将字符串分割成单独的字符。如果你想将一个字符串分割成一些单词，你可以使用 `split` 方法：

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

可以提高一个叫做 **分隔符（delimiter）** 的可选参数，指定什么字符作为单词之间的分界线。下面的例子使用连字符作为分隔符：

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

`join` 的功能和 `split` 相反。它将一个字符串列表的元素拼接起来。`join` 是一个字符串方法，所以你需要在一个分隔符上调用它，并传入一个列表作为参数：

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

在这个例子中，分隔符是一个空格，所以 `join` 在单词之间添加一个空格。如果不使用空格拼接字符串，你可以使用空字符串 `''` 作为分隔符。

10.10 对象和值

如果我们执行下面的赋值语句：

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

我们知道 `a` 和 `b` 都指向一个字符串，但是我们不知道是否他们指向 同一个字符串。这里有两种可能的状态，如图 10-2：状态图所示。

一种情况是，`a` 和 `b` 指向两个有相同值的不同对象。第二种情况是，它们指向同一个对象。

为了查看两个变量是否指向同一个对象，你可以使用 `is` 运算符。



图 10.2: 图 10-2: 状态图

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

在这个例子中，Python 仅生成了一个字符串对象，`a` 和 `b` 都指向它。但是当你创建两个列表时，你得到的是两个对象：

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

所以状态图如图 10-3: 状态图所示。

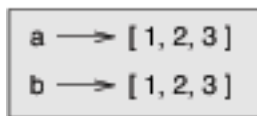


图 10.3: 图 10-3: 状态图

在这个例子中，我们称这两个列表是 **相等 (equivalent)** 的，因为它们有相同的元素。但它们并不 **相同 (identical)**，因为他们不是同一个对象。如果两个对象 **相同**，它们也是相等的，但是如果它们是相等的，它们不一定是相同的。

之前，我们一直在等价地使用“对象”和“值”，但是更准确的说，一个对象拥有一个值。如果你对 `[1, 2, 3]` 求值，会得到一个值为整数序列的列表对象。如果另一个列表有同样的元素，我们说它们有相同的值，但是它们并不是同一个对象。

10.11 别名

如果 `a` 指向一个对象，然后你赋值 `b = a`，那么两个变量指向同一个对象：

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

状态图如图 10-4: 状态图所示。

变量和对象之间的关联称为 **引用 (reference)**。在这个例子中，有两个对同一个对象的引用。

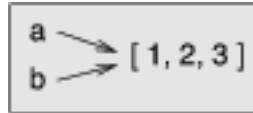


图 10.4: 图 10-4: 状态图

如果一个对象有多于一个引用，那它也会有多个名称，我们称这个对象是 **有别名的**（**aliased**）。

如果一个有别名的对象是可变的，对其中一个别名（**alias**）的改变会影响到其它的别名：

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

尽管这个行为很有用，但是容易导致出现错误。通常，避免对于可变对象使用别名相对更安全。

对于像字符串这样的不可变对象，使用别名没有什么问题。例如：

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

a 和 **b** 是否指向同一个字符串基本上没有什么影响。

10.12 列表参数

当你将一个列表作为参数传给一个函数，函数将得到这个列表的一个引用。如果函数对这个列表进行了修改，会在调用者中有所体现。例如，`delete_head` 删除列表的第一个元素：

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

这样使用这个函数：

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

参数 **t** 和变量 **letters** 是同一个对象的别名。其堆栈图如图 10-5: 堆栈图所示。

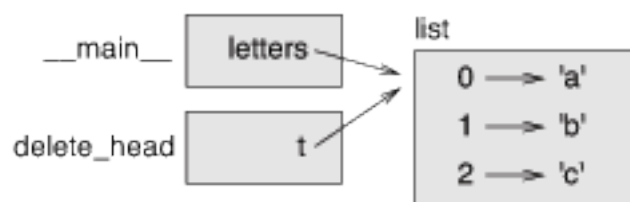


图 10.5: 图 10-5: 堆栈图

由于列表被两个帧共享，我把它画在它们中间。

需要注意的是修改列表操作和创建列表操作间的区别。例如，`append` 方法是修改一个列表，而 `+` 运算符是创建一个新的列表：

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

`append` 修改列表并返回 `None`。

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

运算符 `+` 创建了一个新列表，而不改变原始的列表。

如果你要编写一个修改列表的函数，这一点就很重要。例如，这个函数 不会删除列表的第一个元素：

```
def bad_delete_head(t):
    t = t[1:]
```

切片运算符创建了一个新列表，然后这个表达式让 `t` 指向了它，但是并不会影响原来被调用的列表。

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

在 `bad_delete_head` 的开始处，`t` 和 `t4` 指向同一个列表。在结束时，`t` 指向一个新列表，但是 `t4` 仍然指向原来的、没有被改动的列表。

一个替代的写法是，写一个创建并返回一个新列表的函数。例如，`tail` 返回列表中除了第一个之外的所有元素：

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

这个函数不会修改原来的列表。下面是函数的使用方法：

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

10.13 调试

粗心地使用列表（以及其他可变对象）会导致长时间的调试。下面列举一些常见的陷阱以及避免它们的方法：

1. 大多数的列表方法会对参数进行修改，然后返回 `None`。这和字符串方法相反，后者保留原始的字符串并返回一个新的字符串。

如果你习惯这样写字符串代码：

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

那么你很可能会写出下面的列表代码：

```
t = t.sort() # 错误
```

因为 `sort` 返回 `None`，所以你的下一个对 `t` 执行的操作很可能会失败。

在使用 `list` 方法和操作符之前，你应该仔细阅读文档，然后在交互模式下测试。

2. 选择一种写法，坚持下去。

列表的一个问题就是有太多方法可以做同样的事情。例如，要删除列表中的一个元素，你可以使用 `pop`、`remove`、`del` 甚至是切片赋值。

要添加一个元素，你可以使用 `append` 方法或者 `+` 运算符。假设 `t` 是一个列表，`x` 是一个列表元素，以下这些写法都是正确的：

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

而这些是错误的：

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

在交互模式下尝试每一个例子，保证你明白它们做了什么。注意只有最后一个会导致运行时错误；其他的都是合乎规范的，但结果却是错的。

3. 通过创建拷贝来避免别名。

如果你要使用类似 `sort` 这样的方法来修改参数，但同时有要保留原列表，你可以创建一个拷贝。

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

在这个例子中，你还可以使用内置函数 `sorted`，它将返回一个新的已排序的列表，原列表将保持不变。

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

10.14 术语表

列表 (list) :

多个值组成的序列。

元素 (element) :

列表 (或序列) 中的一个值, 也称为项。

嵌套列表 (nested list) :

作为另一个列表的元素的列表。

累加器 (accumulator) :

循环中用于相加或累积出一个结果的变量。

增量赋值语句 (augmented assignment) :

一个使用类似 `+=` 操作符来更新一个变量的值的语句。

归并 (reduce) :

遍历序列, 将所有元素求和为一个值的处理模式。

映射 (map) :

遍历序列, 对每个元素执行操作的处理模式。

筛选 (filter) :

遍历序列, 选出满足一定标准的元素的处理模式。

对象 (object) :

变量可以指向的东西。一个对象有数据类型和值。

相等 (equivalent) :

有相同的值。

相同 (identical) :

是同一个对象 (隐含着相等)。

引用 (reference) :

一个变量和它的值之间的关联。

别名使用:

两个或者两个以上变量指向同一个对象的情况。

分隔符 (delimiter) :

一个用于指示字符串分割位置的字符或者字符串。

10.15 练习题

你可以从 http://thinkpython2.com/code/list_exercises.py 下载这些练习题的答案。

10.15.1 习题 10-1

编写一个叫做 `nested_sum` 的函数，接受一个由一些整数列表构成的列表作为参数，并将所有嵌套列表中的元素相加。例如：

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

10.15.2 习题 10-2

编写一个叫做 `cumsum` 的函数，接受一个由数值组成的列表，并返回累加和；即一个新列表，其中第 i 个元素是原列表中前 $i + 1$ 个元素的和。例如：

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

10.15.3 习题 10-3

编写一个叫做 `middle` 的函数，接受一个列表作为参数，并返回一个除了第一个和最后一个元素的列表。例如：

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

10.15.4 习题 10-4

编写一个叫做 `chop` 的函数，接受一个列表作为参数，移除第一个和最后一个元素，并返回 `None`。例如：

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

10.15.5 习题 10-5

编写一个叫做 `is_sorted` 的函数，接受一个列表作为参数，如果列表是递增排列的则返回 `True`，否则返回 `False`。例如：

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

10.15.6 习题 10-6

如果可以通过重排一个单词中字母的顺序, 得到另外一个单词, 那么称这两个单词是变位词。编写一个叫做 `is_anagram` 的函数, 接受两个字符串作为参数, 如果它们是变位词则返回 `True`。

10.15.7 习题 10-7

编写一个叫做 `has_duplicates` 的函数, 接受一个列表作为参数, 如果一个元素在列表中出现了不止一次, 则返回 `True`。这个函数不能改变原列表。

10.15.8 习题 10-8

这个习题与所谓的生日悖论有关。你可以在 http://en.wikipedia.org/wiki/Birthday_paradox 中了解更多相关的内容。

如果你的班级上有 23 个学生, 2 个学生生日相同的概率是多少? 你可以通过随机产生 23 个生日, 并检查匹配来估算概率。提示: 你可以使用 `random` 模块中的 `randint` 函数来生成随机生日。

你可以从 <http://thinkpython2.com/code/birthday.py> 下载我的答案。

10.15.9 习题 10-9

编写一个函数, 读取文件 `words.txt`, 建立一个列表, 其中每个单词为一个元素。编写两个版本, 一个使用 `append` 方法, 另一个使用 `t = t + [x]`。那个版本运行得慢? 为什么?

答案: <http://thinkpython2.com/code/wordlist.py>。

10.15.10 习题 10-10

你可以使用 `in` 运算符检查一个单词是否在单词表中, 但这很慢, 因为它是按顺序查找单词。

由于单词是按照字母顺序排序的, 我们可以使用两分法 (也称二进制搜索) 来加快速度, 类似你在字典中查找单词的方法。你从中间开始, 如果你要找的单词在中间的单词之前, 你查找前半部分, 否则你查找后半部分。

不管怎样, 你都会将搜索范围减小一半。如果单词表有 113,809 个单词, 你只需要 17 步就可以找到这个单词, 或者得出单词不存在的结论。

编写一个叫做 `in_bisect` 的函数, 接受一个已排序的列表和一个目标值作为参数, 返回该值在列表中的位置, 如果不存在则返回 `None`。

或者你可以阅读 `bisect` 模块的文档并使用它!

答案: <http://thinkpython2.com/code/inlist.py>。

10.15.11 习题 10-11

两个单词中如果一个是另一个的反转，则二者被称为是“反转词对”。编写一个函数，找出单词表中所有的反转词对。

解答：http://thinkpython2.com/code/reverse_pair.py。

10.15.12 习题 10-12

如果交替的从两个单词中取出字符将组成一个新的单词，这两个单词被称为是“连锁词”。例如，“shoe”和“cold”连锁后成为“schooled”。

答案：<http://thinkpython2.com/code/interlock.py>。致谢：这个练习的灵感来自网站<http://puzzlers.org>的一个示例。

1. 编写一个程序，找出单词表中所有的连锁词。提示：不要枚举所有的单词对。
2. 你能够找到三重连锁的单词吗？即每个字母依次从 3 个单词得到。

10.15.13 贡献者

1. 翻译：[@obserthinker](#)
2. 校对：[@bingjin](#)
3. 参考：[@carfly](#)

第十一章：字典

本章介绍另一个内建数据类型：字典 (dictionary)。字典是 Python 中最优秀的特性之一；许多高效、优雅算法即以此为基础。

11.1 字典即映射

字典与列表类似，但是更加通用。在列表中，索引必须是整数；但在字典中，它们可以是（几乎）任何类型。

字典包含了一个索引的集合，被称为 **键 (keys)**，和一个值 (values) 的集合。一个键对应一个值。这种一一对应的关联被称为 **键值对 (key-value pair)**，有时也被称为 **项 (item)**。

在数学语言中，字典表示的是从键到值的 **映射**，所以你也可以说每一个键“映射到”一个值。举个例子，我们接下来创建一个字典，将英语单词映射至西班牙语单词，因此键和值都是字符串。

`dict` 函数生成一个不含任何项的新字典。由于 `dict` 是内建函数名，你应该避免使用它来命名变量。

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

```

花括号 `{}` 表示一个空字典。你可以使用方括号向字典中增加项：

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

```

这行代码创建一个新项，将键 `'one'` 映射至值 `'uno'`。如果我们再次打印该字典，会看到一个以冒号分隔的键值对：

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

```

输出的格式同样也是输入的格式。例如，你可以像这样创建一个包含三个项的字典：

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

但是，如果你打印 `eng2sp`，结果可能会让你感到意外：

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

键-值对的顺序和原来不同。同样的例子在你的电脑上可能有不同的结果。通常来说，字典中项的顺序是不可预知的。

但这没有关系，因为字典的元素不使用整数索引来索引，而是用键来查找对应的值：

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

键 `'two'` 总是映射到值 `'dos'`，因此项的顺序没有关系。

如果键不存在字典中，会抛出一个异常：

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

`len` 函数也适用于字典；它返回键值对的个数：

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

`in` 操作符也适用于字典；它可以用来检验字典中是否存在某个键（仅仅有这个值还不够）。

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

想要知道字典中是否存在某个值，你可以使用 `values` 方法，它返回值的集合，然后你可以使用 `in` 操作符来验证：

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

`in` 操作符对列表和字典采用不同的算法。对于列表，它按顺序依次查找目标，如[搜索](#)一节所示。随着列表的增长，搜索时间成正比增长。

对于字典，Python 使用一种叫做 **哈希表 (hashtable)** 的算法，这种算法具备一种了不起的特性：无论字典中有多少项，`in` 运算符搜索所需的时间都是一样的。我将在第二十一章的哈希表一节中具体解释背后的原理，但是如果你不再多学习几章内容，现在去看解释的话可能很难理解。

11.2 字典作为计数器集合

假设给你一个字符串，你想计算每个字母出现的次数。有多种方法可以使用：

1. 你可以生成 26 个变量，每个对应一个字母表中的字母。然后你可以遍历字符串，对于每个字符，递增相应的计数器，你可能会用到链式条件。
2. 你可以生成具有 26 个元素的列表。然后你可以将每个字符转化为一个数字（使用内建函数 `ord`），使用这些数字作为列表的索引，并递增适当的计数器。
3. 你可以生成一个字典，将字符作为键，计数器作为相应的值。字母第一次出现时，你应该向字典中增加一项。这之后，你应该递增一个已有项的值。

每个方法都是为了做同一件事，但是各自的实现方法不同。

实现是指执行某种计算的方法；有的实现更好。例如，使用字典的实现有一个优势，即我们不需要事先知道字符串中有几种字母，只要在出现新字母时分配空间就好了。

代码可能是这样的：

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

函数名叫 `histogram`（直方图），是计数器（或是频率）集合的统计术语。

函数的第一行生成一个空字典。`for` 循环遍历该字符串。每次循环，如果字符 `c` 不在字典中，我们用键 `c` 和初始值 1 生成一个新项（因为该字母出现了一次）。如果 `c` 已经在字典中了，那么我们递增 `d[c]`。

下面是运行结果：

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

`histogram` 函数表明字母 'a' 和 'b' 出现了一次，'o' 出现了两次，等等。

字典类有一个 `get` 方法，接受一个键和一个默认值作为参数。如果字典中存在该键，则返回对应值；否则返回传入的默认值。例如：

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

我们做个练习，试着用 `get` 简化 `histogram` 函数。你应该能够不再使用 `if` 语句。

11.3 循环和字典

在 `for` 循环中使用字典会遍历其所有的键。例如，下面的 `print_hist` 会打印所有键与对应的值：

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

输出类似：

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

重申一遍，字典中的键是无序的。如果要以确定的顺序遍历字典，你可以使用内建方法 `sorted`：

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

11.4 逆向查找

给定一个字典 `d` 以及一个键 `t`，很容易找到相应的值 `v = d[k]`。该运算被称作 **查找** (**lookup**)。

但是如果你想通过 `v` 找到 `k` 呢？有两个问题：第一，可能不止一个的键其映射到值 `v`。你可能可以找到唯一一个，不然就得用 `list` 把所有的键包起来。第二，没有简单的语法可以完成 **逆向查找** (**reverse lookup**)；你必须搜索。

下面这个函数接受一个值并返回映射到该值的第一个键：

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

该函数是搜索模式的另一个例子，但是它使用了一个我们之前没有见过的特性，`raise`。`raise` 语句能触发异常，这里它触发了 `ValueError`，这是一个表示查找操作失败的内建异常。

如果我们到达循环结尾，这意味着字典中不存在 `v` 这个值，所以我们触发一个异常。

下面是一个成功逆向查找的例子：

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


和一个失败的例子：

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

你触发的异常和 Python 触发的产生效果一样：都打印一条回溯和错误信息。

raise 语句接受一个详细的错误信息作为可选的实参。例如：

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

逆向查找比正向查找慢得多；如果你频繁执行这个操作或是字典很大，程序性能会变差。

11.5 字典和列表

在字典中，列表可以作为值出现。例如，如果你有一个从字母映射到频率的字典，而你想倒转它；也就是生成一个从频率映射到字母的字典。因为可能有些字母具有相同的频率，所以在倒转字典中的每个值应该是一个字母组成的列表。

下面是一个倒转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

每次循环，key 从 d 获得一个键和相应的值 val。如果 val 不在 inverse 中，意味着我们之前没有见过它，因此我们生成一个新项并用一个 **单元素集合 (singleton)**（只包含一个元素的列表）初始化它。否则就意味着之前已经见过该值，因此将其对应的键添加至列表。

举个例子：

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

图 11-1：状态图是关于 hist 与 inverse 的状态图。字典用标有类型 dict 的方框表示，方框中是键值对。如果值是整数、浮点数或字符串，我就把它们画在方框内部，但我通常把列表画在方框外面，目的只是为了不让图表变复杂。

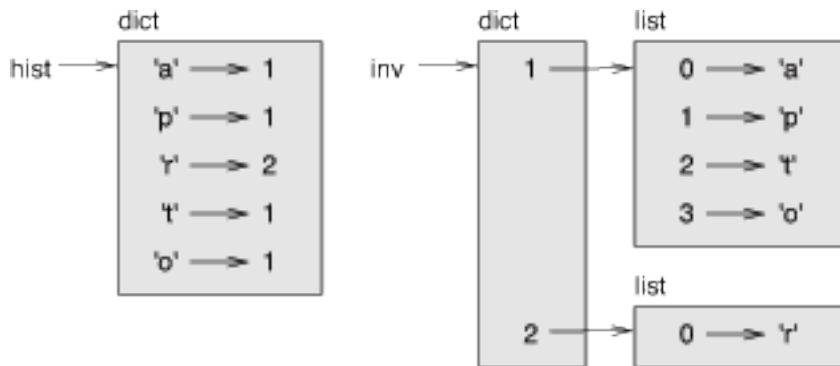


图 11.1: 图 11-1: 状态图

如本例所示，列表可以作为字典中的值，但是不能是键。下面演示了这样做的结果：

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

我之前提过，字典使用哈希表实现，这意味着键必须是 **可哈希的（hashable）**。

哈希（hash） 函数接受一个值（任何类型）并返回一个整数。字典使用被称作哈希值的这些整数，来存储和查找键值对。

如果键是不可变的，那么这种实现可以很好地工作。但是如果键是可变的，如列表，那么就会发生糟糕的事情。例如，当你生成一个键值对时，Python 哈希该键并将其存储在相应的位置。如果你改变键然后再次哈希它，它将被存储到另一个位置。在那种情况下，对于相同的键，你可能有两个值，或者你可能无法找到一个键。无论如何，字典都不会正确的工作。

这就是为什么键必须是可哈希的，以及为什么如列表这种可变类型不能作为键。绕过这种限制最简单的方法是使用元组，我们将在下一章中介绍。

因为字典是可变的，因此它们不能作为键，但是 **可以用作值**。

11.6 备忘录

如果你在**再举一例**一节中接触过 `fibonacci` 函数，你可能注意到输入的实参越大，函数运行就需要越多时间。而且运行时间增长得非常快。

要理解其原因，思考 **图 11-2: 调用图**，它展示了当 `n=4` 时 `fibonacci` 的 **调用图（call graph）**：

调用图中列出了一系列函数栈帧，每个栈帧之间通过线条与调用它的函数栈帧相连。在图的顶端，`n=4` 的 `fibonacci` 调用 `n=3` 和 `n=2` 的 `fibonacci`。接着，`n=3` 的 `fibonacci` 调用 `n=2` 和 `n=1` 的 `fibonacci`。以此类推。

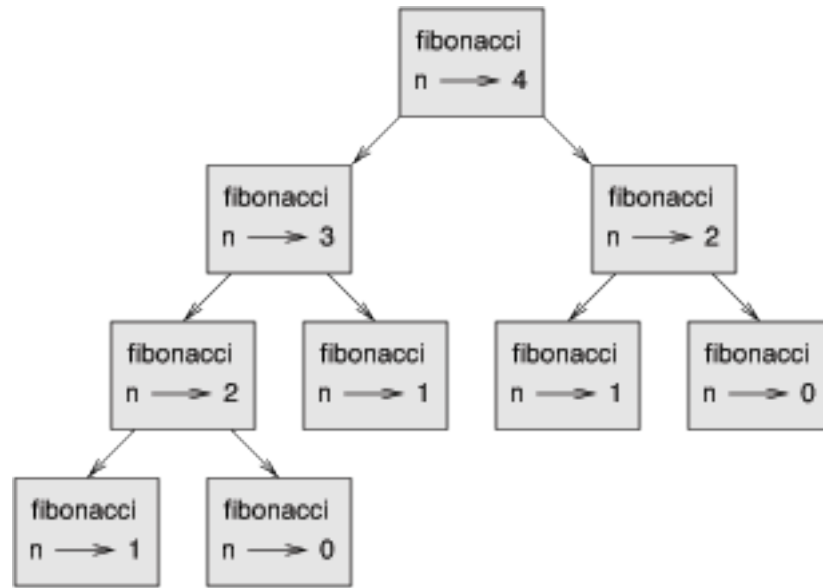


图 11.2: 调用图

函数 `fibonacci(0)` 和 `fibonacci(1)` 总共被调用了几次。对该问题，这不是一个高效的解，并且随着实参的变大会变得更糟。

一个解决办法是保存已经计算过的值，将它们存在一个字典中。存储之前计算过的值以便今后使用，它被称作 **备忘录 (memo)**。下面是使用备忘录 (memoized) 的 `fibonacci` 的实现：

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

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

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

`known` 是一个字典，记录了我们已经计算过的斐波纳契数字。它一开始包含两个项：0 映射到 0，1 映射到 1。

当 `fibonacci` 被调用时，它先检查 `known`。如果结果存在，则立即返回。否则，它必须计算新的值，将其加入字典，并返回它。

将两个版本的 `fibonacci` 函数比比看，你就知道后者快了很多。

11.7 全局变量

在前面的例子中，`known` 是在函数的外部创建的，因此它属于被称作 `__main__` 的特殊帧。因为 `__main__` 中的变量可以被任何函数访问，它们也被称作 **全局变量 (global)**。与函数结束时就会消失的局部变量不同，不同函数调用时全局变量一直都存在。

全局变量普遍用作 **标记 (flag)**；也就是说 (标记) 一个条件是否为真的布尔变量。例如，一些程序使用一个被称作 `verbose` 的标记来控制输出的丰富程度：

```
verbose = True

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

如果你试图对一个全局变量重新赋值，结果可能出乎意料。下面的例子本应该记录函数是否已经被调用过了：

```
been_called = False

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

但是如果你运行它，你会发现 `been_called` 的值并未发生改变。问题在于 `example2` 生成了一个新的被称作 `been_called` 的局部变量。当函数结束的时候，该局部变量也消失了，并且对全局变量没有影响。

要在函数内对全局变量重新赋值，你必须在使用之前 **声明 (declare)** 该全局变量：

```
been_called = False

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

global 语句告诉编译器，“在这个函数里，当我说 `been_called` 时，我指的是那个全局变量，别生成局部变量”。

下面是一个试图更新全局变量的例子：

```
count = 0

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

一旦运行，你会发现：

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

Python 默认 `count` 是局部变量，在这个假设下，你这是在未写入任何东西前就试图读取。解决方法还是声明 `count` 是全局变量。

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

如果全局变量是可变的，你可以不加声明地修改它：

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

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

因此你可以增加、删除和替代全局列表或者字典的元素，但是如果你想对变量重新赋值，你必须声明它：

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

全局变量有时是很有用的，但如果你的程序中有很多全局变量，而且修改频繁，这样会增加程序调试的难度。

11.8 调试

当你操作较大的数据集时，通过打印并手工检查数据来调试很不方便。下面是针对调试大数据集的一些建议：

缩小输入：

如果可能，减小数据集的大小。例如，如果程序读入一个文本文件，从前 10 行开始分析，或是找到更小的样例。你可以选择编辑读入的文件，或是（最好）修改程序使它只读入前 *n* 行。

如果出错了，你可以将 *n* 缩小为会导致该错误的最小值，然后在查找和解决错误的同时，逐步增加 *n* 的值。

检查摘要和类型：

考虑打印数据的摘要，而不是打印并检查全部数据集：例如，字典中项的数目或者数字列表的总和。

运行时错误的一个常见原因，是值的类型不正确。为了调试此类错误，打印值的类型通常就足够了。

编写自检代码：

有时你可以写代码来自动检查错误。例如，如果你正在计算数字列表的平均数，你可以检查其结果是不是大于列表中最大的元素，或者小于最小的元素。这被称作“合理性检查”，因为它能检测出“不合理的”结果。

另一类检查是比较两个不同计算的结果，来看一下它们是否一致。这被称作“一致性检查”。

格式化输出：

格式化调试输出能够更容易定位一个错误。我们在[调试](#)一节中看过一个示例。`pprint` 模块提供了一个 `pprint` 函数，它可以更可读的格式显示内建类型（`pprint` 代表“pretty print”）。

重申一次，你花在搭建脚手架上的时间能减少你花在调试上的时间。

11.9 术语表

映射（mapping）：

一个集合中的每个元素对应另一个集合中的一个元素的关系。

字典 (dictionary):

将键映射到对应值的映射。

键值对 (key-value pair):

键值之间映射关系的呈现形式。

项 (item):

在字典中, 这是键值对的另一个名称。

键 (key):

字典中作为键值对第一部分的对象。

值 (value):

字典中作为键值对第二部分的对象。它比我们之前所用的“值”一词更具体。

实现 (implementation):

执行计算的一种形式。

哈希表 (hashtable):

用来实现 Python 字典的算法。

哈希函数 (hash function):

哈希表用来计算键的位置的函数。

可哈希的 (hashable):

具备哈希函数的类型。诸如整数、浮点数和字符串这样的不可变类型是可哈希的; 诸如列表和字典这样的可变对象是不可哈希的。

查找 (lookup):

接受一个键并返回相应值的字典操作。

逆向查找 (reverse lookup):

接受一个值并返回一个或多个映射至该值的键的字典操作。

raise 语句:

专门引发异常的一个语句。

单元素集合 (singleton):

只有一个元素的列表 (或其他序列)。

调用图 (call graph):

绘出程序执行过程中创建的每个栈帧的调用图, 其中的箭头从调用者指向被调用者。

备忘录 (memo):

一个存储的计算值，避免之后进行不必要的计算。

全局变量（global variable）：

在函数外部定义的变量。任何函数都可以访问全局变量。

global 语句：

将变量名声明为全局变量的语句。

标记（flag）：

用于说明一个条件是否为真的布尔变量。

声明（declaration）：

类似 global 这种告知解释器如何处理变量的语句。

11.10 练习题

11.10.1 习题 11-1

编写一函数，读取 `words.txt` 中的单词并存储为字典中的键。值是什么无所谓。然后，你可以使用 `in` 操作符检查一个字符串是否在字典中。

如果你做过习题 10-10，可以比较一下 `in` 操作符和二分查找的速度。

11.10.2 习题 11-2

查看字典方法 `setdefault` 的文档，并使用该方法写一个更简洁的 `invert_dict`。

答案：http://thinkpython2.com/code/invert_dict.py。

11.10.3 习题 11-3

将习题 6-2 中的 Ackermann 函数备忘录化（memoize），看看备忘录化（memoization）是否可以支持解决更大的参数。没有提示！

答案：http://thinkpython2.com/code/ackermann_memo.py。

11.10.4 习题 11-4

如果你做了习题 10-7，你就已经写过一个叫 `has_duplicates` 的函数，它接受一个列表作为参数，如果其中有某个对象在列表中出现不止一次就返回 `True`。

用字典写个更快、更简单的版本。

答案：http://thinkpython2.com/code/has_duplicates.py。

11.10.5 习题 11-5

两个单词如果反转其中一个就会得到另一个, 则被称作“反转对”(参见习题 8-5 中的 `rotate_word`)。

编写一程序, 读入单词表并找到所有反转对。

答案: http://thinkpython2.com/code/rotate_pairs.py。

11.10.6 习题 11-6

下面是取自 *Car Talk* 的另一个字谜题 (<http://www.cartalk.com/content/puzzlers>):

这是来自一位名叫 Dan O’Leary 的朋友的分享。他有一次碰到了一个常见的单音节、有五个字母的单词, 它具备以下独特的特性。当你移除第一个字母时, 剩下的字母组成了原单词的同音词, 即发音完全相同的单词。将第一个字母放回, 然后取出第二个字母, 结果又是原单词的另一个同音词。那么问题来了, 这个单词是什么?

接下来我给大家举一个不满足要求的例子。我们来看一个五个字母的单词“*wrack*”。W-R-A-C-K, 常用短句为“*wrack with pain*”。如果我移除第一个字母, 就剩下了一个四个字母的单词“*R-A-C-K*”。可以这么用, “*Holy cow, did you see the rack on that buck! It must have been a nine-pointer!*” 它是一个完美的同音词。如果你把“*w*”放回去, 移除“*r*”, 你得到的单词是“*wack*”。这是一个真实的单词, 但并不是前两个单词的同音词。

不过, 我们和 Dan 知道至少有一个单词是满足这个条件的, 即移除前两个字母中的任意一个, 将会得到两个新的由四个字母组成的单词, 而且发音完全一致。那么这个单词是什么呢?

你可以使用习题 11-1 中的字典检查某字符串是否出现在单词表中。

你可以使用 CMU 发音字典检查两个单词是否为同音词。从 <http://www.speech.cs.cmu.edu/cgi-bin/cmudict> 或 <http://thinkpython2.com/code/c06d> 即可下载。你还可以下载 <http://thinkpython2.com/code/pronounce.py> 这个脚本, 其中提供了一个名叫 `read_dictionary` 的函数, 可以读取发音字典, 并返回一个将每个单词映射至描述其主要梵音的字符串的 Python 字典。

编写一个程序, 找到满足字谜题条件的所有单词。

11.10.7 贡献者

1. 翻译: @theJian & @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第十二章：元组

本章介绍另一个内置类型：元组，同时说明如何结合使用列表、字典和元组。我还将介绍一个有用的特性，即可变长度参数列表，以及汇集和分散操作符。

说明：“tuple”并没有统一的发音，有些人读成“tuh-ple”，音律类似于“supple”；而在编程的语境下，大部分读成“too-ple”，音律类似于“quadruple”。

12.1 元组是不可变的

元组是一组值的序列。其中的值可以是任意类型，使用整数索引，因此从这点上看，元组与列表非常相似。二者不同之处在于元组的不可变性。

语法上，元组是用逗号隔开一系列值：

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

虽然并非必须，元组通常用括号括起来：

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

使用单一元素创建元组时，需要在结尾处添加一个逗号：

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

将值放置在括号中并不会创建元组：

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

另一个创建元组的方法是使用内建函数 `tuple`。在没有参数传递时，它会创建一个空元组：

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

如果实参是一个序列（字符串、列表或者元组），结果将是一个包含序列内元素的元组。

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

因为 `tuple` 是内建函数名，所以应该避免将它用作变量名。

列表的大多数操作符同样也适用于元组。方括号运算符将索引一个元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

切片运算符选取一个范围内的元素：

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

但是，如果你试图元组中的一个元素，会得到错误信息：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

因为元组是不可变的，你无法改变其中的元素。但是可以使用其他元组替换现有元组：

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

这个语句创建了一个新元组，然后让 `t` 引用该元组。

关系型运算符也适用于元组和其他序列；Python 会首先比较序列中的第一个元素，如果它们相等，就继续比较下一组元素，以此类推，直至比值不同。其后的元素（即便是差异很大）也不会再参与比较。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 元组赋值

两个变量互换值的操作通常很有用。按照传统的赋值方法，你需要使用一个临时变量。例如，为了交换 `a` 和 `b` 的值：

```
>>> temp = a
>>> a = b
>>> b = temp
```

这个方法很繁琐；通过**元组赋值**来实现更为优雅：

```
>>> a, b = b, a
```

等号左侧是变量组成的元组；右侧是表达式组成的元组。每个值都被赋给了对应的变量。变量被重新赋值前，将先对右侧的表达式进行求值。

左侧的变量数和右侧值的数目必须相同：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

一般来说，右侧可以是任意类型（字符串、列表或者元组）的序列。例如，将一个电子邮箱地址分成用户名和域名，你可以这样做：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

`split` 函数返回的对象是一个包含两个元素的列表；第一个元素被赋给了变量 `uname`，第二个被赋给了 `domain`。

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3 元组作为返回值

严格地说，一个函数只能返回一个值，但是如果这个返回值是元组，其效果等同于返回多个值。例如，你想对两个整数做除法，计算出商和余数，依次计算出 x/y 和 $x\%y$ 的效率并不高。同时计算出这两个值更好。

内建函数 `divmod` 接受两个实参，返回包含两个值的元组：商和余数。你可以使用元组来存储返回值：

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

或者使用元组赋值分别存储它们：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

下面是一个返回元组作为结果的函数例子：

```
def min_max(t):
    return min(t), max(t)
```

`max` 和 `min` 是用于找出一组元素序列中最大值和最小值的内建函数。`min_max` 函数同时计算出这两个值，并返回二者组成的元组。

12.4 可变长度参数元组

函数可以接受可变数量的参数。以 “*” 开头的形参将输入的参数 汇集到一个元组中。例如, `printall` 可以接受任意数量的参数, 并且将它们打印出来:

```
def printall(*args):
    print(args)
```

汇集形参可以使用任意名字, 但是习惯使用 `args`。以下是这个函数的调用效果:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

与汇集相对的, 是 **分散 (scatter)**。如果你有一个值序列, 并且希望将其作为多个参数传递给一个函数, 你可以使用运算符 `*`。例如, `divmod` 只接受两个实参; 元组则无法作为参数传递进去:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

但是如果你将这个元组分散, 它就可以被传递进函数:

```
>>> divmod(*t)
(2, 1)
```

许多内建函数使用了可变长度参数元组。例如, `max` 和 `min` 就可以接受任意数量的实参:

```
>>> max(1, 2, 3)
3
```

但是 `sum` 不行:

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

我们做个练习, 编写一个叫做 `small` 的函数, 使它能够接受任何数量的实参并返回它们的和。

12.5 列表和元组

`zip` 是一个内建函数, 可以接受将两个或多个序列组, 并返回一个元组列表, 其中每个元组包含了各个序列中相对位置的一个元素。这个函数的名称来自名词拉链 (`zipper`), 后者将两片链齿连接拼合在一起。

下面的示例对一个字符串和列表使用 `zip` 函数:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

输出的结果是一个 **zip 对象**, 包含了如何对其中元素进行迭代的信息。`zip` 函数最常用于 `for` 循环:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

`zip` 对象是迭代器的一种，即任何能够按照某个序列迭代的对象。迭代器在某些方面与列表非常相似，但不同之处在于，你无法通过索引来选择迭代器中的某个元素。

如果你想使用列表操作符和方法，你可以通过 `zip` 对象创建一个列表：

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

结果就是一个包含若干元组的列表；在这个例子中，每个元组又包含了字符串中的一个字符和列表中对应的一个元素。

如果用于创建的序列长度不一，返回对象的长度以最短序列的长度为准。

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

您可以在 `for` 循环中使用元组赋值，遍历包含元组的列表：

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

每次循环时，Python 会选择列表中的下一个元组，并将其内容赋给 `letter` 和 `number`。循环的输出是：

```
0 a
1 b
2 c
```

如果将 `zip`、`for` 循环和元组赋值结合起来使用，你会得到一个可以同时遍历两个（甚至多个）序列的惯用法。例如，`has_match` 接受两个序列 `t1` 和 `t2`，如果存在索引 `i` 让 `t1[i] == t2[i]`，则返回 `True`：

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

如果需要遍历一个序列的元素以及其索引号，您可以使用内建函数 `enumerate`：

```
for index, element in enumerate('abc'):
    print(index, element)
```

`enumerate` 的返回结果是一个枚举对象（`enumerate object`），可迭代一个包含若干个对的序列；每个对包含了（从 0 开始计数）的索引和给定序列中的对应元素。在这个例子中，输出结果是：

```
0 a
1 b
2 c
```

和前一个示例的结果一样。

12.6 字典和元组

字典有一个叫做 `items` 的方法，它返回由多个元组组成的序列，其中每个元组是一个键值对。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

其结果是一个 `dict_items` 对象，这是一个对键值对进行迭代的迭代器。你可以在 `for` 循环中像这样使用它：

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

由于是字典生成的对象，你应该也猜到了这些项是无序的。

另一方面，您可以使用元组的列表初始化一个新的字典：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

将 `dict` 和 `zip` 结合使用，可以很简洁地创建一个字典：

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

字典的 `update` 方法也接受元组列表，并将其作为键值对添加到已有的字典中去。

在字典中使用元组作为键（主要因为无法使用列表）的做法很常见。例如，一个电话簿可能会基于用户的姓-名对，来映射至号码。假设我们已经定义了 `last`、`first` 和 `number` 三个变量，我们可以这样实现映射：

```
directory[last, first] = number
```

方括号中的表达式是一个元组。我们可以通过元组赋值来遍历这个字典：

```
for last, first in directory:
    print(first, last, directory[last,first])
```

该循环遍历电话簿中的键，它们其实是元组。循环将元组的元素赋给 `last` 和 `first`，然后打印出姓名和对应的电话号码。

在状态图中有两种表示元组的方法。更详细的版本是，索引号和对应该元素就像列表一样存放在元组中。例如，元组 (`'Cleese'`, `'John'`) 可像图 12-1：状态图中那样存放。

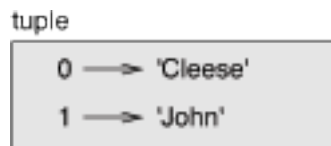


图 12.1: 图 12-1：状态图

在更大的图表中，你不会想要再描述这些细节。例如，该电话簿的状态图可能如图 12-2：状态图所示。

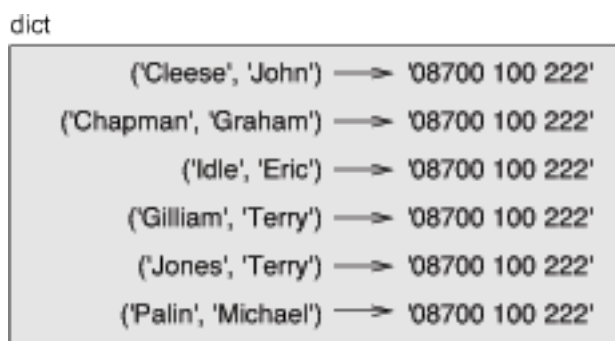


图 12.2: 图 12-2：状态图

在上图中，为了方便起见，使用 Python 语法表示元组。此图中的电话号码是 BBC 的投诉热线，请不要拨打它。

12.7 序列嵌套

我已经介绍了包含元组的列表，但本章几乎所有示例也适用于列表嵌套列表、元组嵌套元组，以及元组嵌套列表。为了避免穷举这类可能的嵌套组合，介绍序列嵌套有时更简单一些。

在很多情况下，不同类型的序列（字符串、列表、元组）可以互换使用。因此，我们该如何选用合适的序列呢？

首先，显而易见的是，字符串比其他序列的限制更多，因为它的所有元素都必须是字符，且字符串不可变。如果你希望能够改变字符串中的字符，使用列表嵌套字符或许更合适。

列表比元组更常用，主要是因为它们是可变的。但是有些情况下，你可能更倾向于使用元组：

1. 在一些情况下（例如 `return` 语句），从句式上生成一个元组比列表要简单。

2. 如果你想使用一个序列作为字典的键，那么你必须使用元组或字符串这样的不可变类型。
3. 如果你向函数传入一个序列作为参数，那么使用元组可以降低由于别名而产生的意外行为的可能性。

由于元组的不可变性，它们没有类似（`sort`）和（`reverse`）这样修改现有列表的方法。然而 Python 提供了内建函数 `sorted` 和 `reversed`，前者可以接受任意序列，并返回一个正序排列的新列表，后者则接受一个序列，返回一个可逆序迭代列表的迭代器。

12.8 调试

列表、字典和元组都是数据结构（**data structures**）；本章中，我们开始接触到复合数据结构（**compound data structures**），如：列表嵌套元组，以及使用元组作为键、列表作为值的字典。复合数据结构非常实用，但是使用时容易出现所谓的形状错误（*shape errors*），也就是由于数据结构的类型、大小或结构问题而引发的错误。例如，当你希望使用一个整数组成的列表时，我却给了你一个纯粹的整数（没有放在列表中），就会出现错误。

为了方便调试这类错误，我编写了一个叫做 `structshape` 的模块，它提供了一个同名函数，可以接受任意类型的数据结构作为实参，然后返回一个描述它形状的字符串。你可以从 <http://thinkpython2.com/code/structshape.py> 下载该模块。

下面是用该模块调试一个简单列表的示例：

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

更完美的程序应该显示“list of 3 ints”，但是忽略英文复数使程序变得简单的多。我们再看一个列表嵌套的例子：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

如果列表内的元素不是相同类型，`structshape` 会按照类型的顺序进行分组：

```
>>> 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)'
```

下面是一个元组列表的例子：

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

下面是一个字典的例子，其中包含三个将整数映射至字符串的项：

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

如果你在追踪数据结构的类型上遇到了困难，可以使用 `structshape` 来帮助分析。

12.9 术语表

元组 (tuple):

一个由多个元素组成的不可变序列。

元组赋值 (tuple assignment):

一种赋值方式，等号右侧为一个序列，等号左侧为一个变量组成的元组。右侧的表达式先求值，然后其元素被赋值给左侧元组中对应的变量。

汇集 (gather):

组装可变长度实参元组的一种操作。

分散 (scatter):

将一个序列变换成一个参数列表的操作。

zip 对象:

使用内建函数 `zip` 所返回的结果；它是一个可以对元组序列进行迭代的对象。

迭代器 (iterator):

一个可以对序列进行迭代的对象，但是并不提供列表操作符和方法。

数据结构 (data structure):

一个由关联值组成的数据集合，通常组织成列表、字典、元组等。

形状错误 (shape error):

由于某个值的形状出错，而导致的错误；即拥有错误的类型或大小。

12.10 练习题

12.10.1 习题 12-1

编写一个名为 `most_frequent` 的函数，接受一个字符串，并按字符出现频率降序打印字母。找一些不同语言的文本样本，来试试看不同语言之间字母频率的区别。将你的结果和 http://en.wikipedia.org/wiki/Letter_frequencies 页面上的表格进行比较。

答案: http://thinkpython2.com/code/most_frequent.py。

12.10.2 习题 12-2

再来练习练习易位构词:

1. 编写一个程序，使之能从文件中读取单词列表（参考读取单词列表一节），并且打印出所有属于易位构词的单词组合。

下面是一个输出结果的示例:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

提示：你也许应该创建一个字典，用于映射一个字母集合到一个该集合可异位构词的词汇集合。但是问题是，你怎样表示这个字母集合才能将其用作字典的键呢？

2. 改写前面的程序，使之先打印异位构词数量最多的列表，第二多的次之，依次按异位构词的数量排列。
3. 在 Scrabble 拼字游戏中，游戏胜利（“bingo”）指的是你利用手里的全部七个字母，与图版上的那个字母一起构成一个 8 个字母的单词。哪八个字母能够达成最多的“bingo”？提示：最多有 7 种胜利方式。

答案：http://thinkpython2.com/code/anagram_sets.py。

12.10.3 习题 12-3

如果两个单词中的某一单词可以通过调换两个字母变为另一个，这两个单词就构成了“换位对（metathesis pair）”；比如，“converse”和“conserve”。编写一个程序，找出字典里所有的“换位对”。

提示：不用测试所有的单词组合，也不用测试所有的字母调换组合。致谢：这道习题受 <http://puzzlers.org> 上的案例启发而来。

答案：<http://thinkpython2.com/code/metathesis.py>。

12.10.4 习题 12-4

又是一个来自 Car Talk 的字谜题（<http://www.cartalk.com/content/puzzlers>）：

如果你每一次从单词中删掉一个字母以后，剩下的字符仍然能构成一个单词，请问世界上符合条件的最长单词是什么？

注意，被删掉的字母可以位于首尾或是中间，但不允许重新去排列剩下的字母。每次移除一个字母后，你会得到一个新单词。这样一直下去，最终你只剩一个字母，并且它也是一个单词——可以在字典中查到。我想知道，符合条件的最长单词是什么？它由多少个字母构成？

我先给出一个短小的例子：“Sprite”。一开始是 `sprite`，我们可以拿掉中间的‘r’从而获得单词 `spite`，然后拿掉字母‘e’得到 `spit`，再去掉‘s’，剩下 `pit`，依次操作得到 `it`，和 `I`。

编写一个程序，找到所有能够按照这种规则缩减的单词，然后看看其中哪个词最长。

这道题比大部分的习题都要难，所以我给出一些建议：

1. 你可以写一个函数，接受一个单词，然后计算所有“子词”（即拿掉一个字母后所有可能的单词）组成的列表。
2. 递归地看，如果单词的子词之一也可缩减，那么这个单词也可被缩减。我们可以将空字符串视作也可以缩减，视其为基础情形。

3. 我提供的词汇列表中，并未包含诸如 ‘I’、‘a’ 这样的单个字母词汇，因此，你可能需要加上它们，以及空字符串。
4. 为了提高程序的性能，你可能需要暂存（memoize）已知可被缩减的单词。

答案：<http://thinkpython2.com/code/reducible.py>。

12.10.5 贡献者

1. 翻译：@SeikaScarlet
2. 校对：@bingjin
3. 参考：@carfly

第十三章：案例研究：数据结构选择

目前为止，你已经学完了 Python 的核心数据结构，同时你也接触了利用到这些数据结构的一些算法。如果你希望学习更多算法知识，那么现在是阅读第二十一章的好时机。但是不必急着马上读，什么时候感兴趣了再去读即可。

本章是一个案例研究，同时给出了一些习题，目的是启发你思考如何选择数据结构，并练习数据结构使用。

13.1 词频分析

和之前一样，在查看答案之前，你至少应该试着解答一下这些习题。

13.1.1 习题 13-1

编写一个程序，读取一个文件，将每一行转换成单词列表，删掉单词中的空格和标点，然后将它们转换为小写字母。

提示：string 模块提供了名为 `whitespace` 的字符串，其包括空格、制表符、新行等等，以及名为 `punctuation` 的字符串，其包括标点字符。试试能否让 Python 说脏话：

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

同时，你可以考虑使用字符串方法 `strip`、`replace` 和 `translate`。

13.1.2 习题 13-2

前往古腾堡项目 (<http://gutenberg.org>)，以纯文本格式下载你喜欢的已无版权保护的图书。

修改前面习题的程序，读取你下载的书，跳过文件开始的头部信息，像之前那样处理其余的单词。

然后修改程序，计算书中单词的总数，以及每个单词使用的次数。

打印该书使用单词的总数。比较不同年代、不同作者写的书。哪个作者使用的词汇量最大？

13.1.3 习题 13-3

修改上一个习题中的程序，打印书中最常使用的 20 个单词。

13.1.4 习题 13-4

修改上一个习题中的程序，读取一个单词列表（见[读取单词列表](#)一节），然后打印书中所有没有出现在该单词表中的单词。它们中有多少是拼写错误的？有多少是词表中应该包括的常用词？有多少是生僻词？

13.2 随机数

给定相同的输入，大多数计算机程序每次都会生成相同的输出，它们因此被称作**确定性的**（**deterministic**）。确定性通常是个好东西，因为我们期望相同的计算产生相同的结果。然而，对于有些应用，我们希望计算机不可预知。游戏是一个明显的例子，但是不限于此。

让程序具备真正意义上的非确定性并不容易，但是有办法使它至少看起来是不确定的。其中之一是使用生成**伪随机**（**pseudorandom**）数的算法。伪随机数不是真正的随机数，因为它们由一个确定性的计算生成，但是仅看其生成的数字，不可能将它们和随机生成的相区分开。

`random` 模块提供了生成伪随机数（下文中简称为“随机数”）的函数。

函数 `random` 返回一个 0.0 到 1.0 之间的随机浮点数（包括 0.0，但是不包括 1.0）。每次调用 `random`，你获得一个长序列中的下一个数。举个例子，运行此循环：

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

函数 `randint` 接受参数 `low` 和 `high`，返回一个 `low` 和 `high` 之间的整数（两个都包括）。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

你可以使用 `choice`，从一个序列中随机选择一个元素：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

`random` 模块提供的函数，还可以生成符合高斯、指数、伽马等连续分布的随机值。

13.2.1 习题 13-5

编写一个名为 `choose_from_hist` 的函数，其接受一个如字典作为计数器集合一节中定义的 `histogram` 对象作为参数，并从该对象中返回一个随机值，其选择概率和值出现的频率成正比。例如：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

你的函数返回 'a' 的概率应该是 2/3，返回 'b' 的概率应该是 1/3。

13.3 单词直方图

在继续下面的习题之前，你应该尝试完成前面的练习。你可以从http://thinkpython2.com/code/analyze_book1.py 下载我的答案。你还需要下载<http://thinkpython2.com/code/emma.txt>。

下面这个程序将读取一个文件，并建立文件中单词的直方图：

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

该程序读取 `emma.txt`，其包括 Jane Austen 写的《Emma》的文本。

`process_file` 循环读取文件的每行，依次把它们传递给 `process_line`。直方图 `hist` 被用作一个累加器。

在使用 `split` 将一行文件切分成一个字符串列表之前，`process_line` 使用字符串的 `replace` 方法将连字符替换成空格。它会遍历单词列表，并使用 `strip` 和 `lower` 来删除标点以及将单词转换为小写。（“转换”只是一种简略的说法；记住，字符串是不可变的，所以类似 `strip` 和 `lower` 这样的方法其实返回的是新字符串。）

最后，`process_line` 通过生成一个新的项或者递增一个已有的项来更新直方图。

我们可以通过累加直方图中的频率，来统计文件中的单词总数：

```
def total_words(hist):  
    return sum(hist.values())
```

不同单词的数量恰好是词典中项的数目:

```
def different_words(hist):  
    return len(hist)
```

这是打印结果的代码:

```
print('Total number of words:', total_words(hist))  
print('Number of different words:', different_words(hist))
```

结果是:

```
Total number of words: 161080  
Number of different words: 7214
```

13.4 最常用单词

为了找到最常用的单词, 我们可以使用元组列表, 其中每个元组包含单词和它的频率, 然后排序这个列表。

下面的函数接受一个直方图并且返回一个单词-频率的元组列表:

```
def most_common(hist):  
    t = []  
    for key, value in hist.items():  
        t.append((value, key))  
  
    t.sort(reverse=True)  
    return t
```

每一个元组中, 频率在前, 所以这个列表是按照频率排序。下面是输出最常用的十个单词的循环:

```
t = most_common(hist)  
print('The most common words are:')  
for freq, word in t[:10]:  
    print(word, freq, sep='\t')
```

这里我通过关键词参数 `sep`, 让 `print` 使用一个制表符 (Tab) 而不是空格键作为分隔符, 所以第二行将对齐。下面是对小说 *《Emma》* 的分析结果:

```
The most common words are:  
to      5242  
the     5205  
and     4897  
of      4295  
i       3191  
a       3130  
it      2529  
her     2483
```



```
was      2400
she      2364
```

当然，这段代码也可以通过 `sort` 函数的参数 `key` 进行简化。如果你感兴趣，可以阅读 <https://wiki.python.org/moin/HowTo/Sorting>。

13.5 可选形参

我们已经见过接受可变数量实参的函数和方法了。程序员也可以自己定义具有可选实参的函数。例如，下面就是一个打印直方图中最常见单词的函数。

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

第一个形参是必须的；第二个是可选的。`num` 的默认值（**default value**）是 10。

如果你只提供了一个实参：

```
print_most_common(hist)
```

`num` 将使用默认值。如果你提供两个实参：

```
print_most_common(hist, 20)
```

`num` 获得实参的值。换句话说，可选实参覆盖（**overrides**）了默认值。

如果一个函数同时有必选和可选两类形参，则所有的必选形参必须首先出现，可选形参紧随其后。

13.6 字典差集

从书中找到所有没出现在词表 `words.txt` 中的单词，可以认为是一个差集问题；也就是，我们应该从一个集合中（书中的单词）找到所有没出现在另一个集合中（列表中的单词）的单词。

`subtract` 接受词典 `d1` 和 `d2`，并返回一个新的词典，其包括 `d1` 中的所有没出现在 `d2` 中的键。由于并不真正关心值是什么，我们将它们都设为 `None`。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

为了找到书中没有出现在 `words.txt` 中的单词，我们可以使用 `process_file` 来为 `words.txt` 构建一个直方图，然后使用 `subtract`：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff.keys():
    print(word, end=' ')
```

这是针对小说《Emma》的部分运行结果：

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

这些单词中，一些是名字和名词所有格。如“rencontre”这样的其他单词已经不常使用了。但是有一些真的应该包括在列表中！

13.6.1 习题 13-6

Python 提供了一个叫做集合（set）的数据结构，支持许多常见的集合操作。你可以前往第十九章阅读相关内容，或者在官网上阅读文档 <http://docs.python.org/3/library/stdtypes.html#types-set>。

编写一个程序，使用集合的差集操作来找出一本书中不在 `word list` 中的单词。

答案：http://thinkpython2.com/code/analyze_book2.py。

13.7 随机单词

如果想从直方图中随机选择一个单词，最简单的算法是创建一个列表，其中根据其出现的频率，每个单词都有相应个数的拷贝，然后从该列表中选择单词：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

表达式 `[word] * freq` 创建一个具有 `freq` 个 `word` 字符串拷贝的列表。除了它的实参要求是一个序列外，`extend` 方法和 `append` 方法很像。

该算法能够满足要求，但是效率不够高；每次你选择一个随机单词，它都重建列表，这个列表和原书一样大。一个明显的改进是，创建列表一次，然后进行多次选择，但是该列表仍然很大。

一个替代方案是：

1. 使用 `keys` 来获得该书中单词的列表。
2. 创建一个包含单词频率累积和的列表（见习题 10-2）。此列表的最后一项是书中单词的数目 n 。

3. 选择一个从 1 到 n 的随机数。使用二分搜索（见习题 10-10）找到该随机数应该被在累积和中插入的索引。
4. 使用该索引从单词列表中找到相应的单词。

13.7.1 习题 13-7

编写一个使用该算法从书中选择一个随机单词的程序。

答案：http://thinkpython2.com/code/analyze_book3.py。

13.8 马尔科夫分析

如果你从书中随机选择单词，那么你会大致了解其使用的词汇，但可能不会得到一个完整的句子：

```
this the small regard harriet which knightley's it most things
```

一系列随机单词很少有意义，因为相邻的单词之间没有关系。例如，在一个真实的句子中，你可能期望“the”这样的冠词后面跟着的是一个形容词或者名词，而大不可能会是一个动词或者副词。

衡量相邻单词关系的方法之一是马尔科夫分析法，对于一个给定的单词序列，马尔科夫分析法将给出接下来单词的概率。例如，歌曲 *Eric, the Half a Bee* 的开头是：

```
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D' you see?
But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?
```

在此文本中，短语“half the”后面总是跟着单词“bee”，但是短语“the bee”则可能跟着“has”或者“is”。

马尔科夫分析的结果是从每个前缀（如“half the”和“the bee”）到所有可能的后缀（如“has”和“is”）的映射。

给定此映射，你能够通过以任意前缀开始并从可能的后缀中随机选择一个的方法，来生成一个随机文本。接下来，你可以将前缀的结尾和新的后缀组合成下一个前缀，并重复下去。

例如，如果你以前缀“Half a”开始，然后下一个但是必须是“bee”，因为此前缀在文本中仅出现一次。下一个前缀是“a bee”，所以下一个后缀可能是“philosophically”，“be”或“due”。

此例中，前缀的长度总是 2，但是你可以以任意前缀长度进行马尔科夫分析。前缀的长度被称作此分析的“阶”。

13.8.1 习题 13-8

马尔科夫分析：

1. 编写一个程序，从一个文件中读取文本并执行马尔科夫分析。结果应该是一个字典，即从前缀映射到一个可能的后缀集合。此后缀集合可以是一个列表、元组或字典；你需要做出合适的选择。你可以用长度为 2 的前缀测试程序，但是在编写程序时，应确保其很容易支持其它长度。
2. 在前面的程序中添加一个函数，基于马尔科夫分析生成随机文本。下面是使用《Emma》执行前缀为 2 的马尔科夫分析生成的示例：

```
He was very clever, be it sweetness or be angry, ashamed or only amused,  
at such a stroke. She had never thought of Hannah till you were never  
meant for me?" "I cannot make speeches, Emma:" he soon cut it all himself.
```

在此例中，我保留了附在词后面的标点符号。从语法上看，结果几乎是正确的，但不完全。语义上讲，它几乎有意义，但也不完全。

如果你增加前缀的长度，会发生什么？随机文本更有意义是么？

3. 一旦程序正常运行，你可以想尝试一下混搭：如果你组合两本或更多书中的文本，你生成的随机文本将以有趣的方式混合这些书中的词汇和短语。

致谢：此案例研究基于 Kernighan 与 Pike 所著的《The Practice of Programming》一书中的示例。

在继续阅读之前，你应该尝试解决该习题；你可以从<http://thinkpython2.com/code/markov.py> 下载我的答案。你还需要下载<http://thinkpython2.com/code/emma.txt>。

13.9 数据结构

使用马尔科夫分析生成随机文本很有趣，但是上面那道习题的目的是：学习数据结构选择。在解答上述习题时，你不得不选择：

- 如何表示前缀。
- 如何表示可能后缀的集合。
- 如何表示从前缀到可能后缀集合的映射。

最后一个选择很简单：明显应该选择字典作为键至对应值的映射。

对于前缀，最明显的选择是字符串、字符串列表或者字符串元组。

对于后缀，一个选择是列表；另一个是直方图（字典）。

你如何选择呢？第一步是考虑对每个数据结构你需要实现的操作。对于前缀，我们需要能从头部删除单词，并在结尾处加入单词。例如，如果当前的前缀是“Half a”，下一个词是“bee”，你需要能构成下一个前缀“a bee”。

你的第一个选择可能是列表，因为它能很容易的增加和删除元素，但是我们也需要让前缀作为字典的键，这就排除了列表。使用元组，你不能追加或删除元素，但是你能使用加号运算符来形成一个新的元组：

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` 接受一个单词元组 `prefix` 和一个字符串 `word`，并形成一个新的元组，其具有 `prefix` 中除第一个单词外的全部单词，然后在结尾增加 `word`。

对于后缀的集合，我们需要执行的运算包括增加一个新的后缀（或者增加一个已有后缀的频率），并选择一个随机后缀。

对于列表或者直方图，增加一个新的后缀一样容易。从列表中选择一个随机元素很容易；在直方图中选择的难度更大（见习题 13-7）。

目前为止，我们主要讨论实现的难易，但是选择数据结构时还要考虑其它因素。一个是运行时间。有时，一个数据结构比另一个快有理论依据；例如，我提到过 `in` 运算符对于字典比对列表要快，至少当元素的数目很大的时候。

但是通常你事先不知道哪个实现更快。一个选择是两个都实现，然后再看哪个更快。此方法被称作**基准测试（benchmarking）**。另一个更实际的选择是选择最容易实现的数据结构，然后看它对于拟定的应用是否足够快。如果是的话，就不需要继续了。如果不是，可以使用一些工具，如 `profile` 模块，识别程序中哪处最耗时。

另一个要考虑的因素是存储空间。例如，使用直方图表示后缀集合可能用更少的空间，因为无论一个单词在文本中出现多少次，你只需要存储它一次。在一些情况下，节省空间也能让你的程序更快，极端情况下，如果内存溢出，你的程序可能根本不能运行。但是对于许多应用，空间是运行时间之后的第二位考虑。

最后一点：在此讨论中，我暗示了我们应该使用一种数据结构同时进行分析 and 生成。但是既然这些是独立的步骤，使用一种数据结构进行分析，然后采用另一种结构进行生成也是可能的。如果生成节省的时间超过了转化花费的时间，这也会提高程序的性能。

13.10 调试

在调试一个程序的时候，特别是调试一个很难的错误时，应该做到以下五点：

细读：

检查你的代码，仔细地阅读，并且检查是否实现了你的期望。

运行：

通过修改和运行不同的版本来不断试验。通常，如果你在程序中正确的地方打印了正确的东西，问题会变得很明显，但是有时你不得不搭建一些脚手架。

思考：

花些时间思考！错误的类型是什么：语法、运行时、语义？你从错误信息或者程序的输出中能获得什么信息？什么类型的错误能引起你看到的问题？问题出现前，你最后的修改是什么？

小黄鸭调试法（rubberducking）：

如果将你的问题解释给别人听，有时你会发现在解释完问题之前就能找到答案。你通常并不需要真的去问另外一个人；你可以对着一个小黄鸭说。这就是著名的小黄鸭调试法（**rubber duck debugging**）的由来。这可不是我编造的，你可以看看这个维基页面：https://en.wikipedia.org/wiki/Rubber_duck_debugging。

回退：

有时候，最好的做法是回退，撤销最近的修改，直到你回到一个能运行并且你能理解的程序。然后你可以开始重建。

初级程序员有时陷入这些步骤之一，忘记了还可以做其他的事情。事实上，每种方法都有失败的可能。

例如，如果程序是一个排版错误，读代码可能有帮助，但是如果问题是概念理解错误，则未必是这样。如果你不理解程序要做什么，可能读 100 遍程序都不会发现错误，因为错误在你的头脑中。

试验可能会有帮助，特别是如果你运行简单短小的测试。但是，如果你不思考或者阅读你的代码，就直接进行实验，你可能陷入一种我称为“随机游走编程”的模式。这指的是随机修改，直到程序通过测试。不用说，随机游走编程会花费很长的时间。

你必须花时间思考。调试就像是一门实验科学。你应该至少有一个关于问题是什么的假设。如果有两个或者更多的可能，试着考虑利用测试消除其中一个可能。

但是，如果有太多的错误，或者你正试图修复的代码太大、太复杂，即使最好的调试技巧也会失败。有时，最好的选择是回退，简化程序，直到你获得一个正常运行并且能理解的程序。

初级程序员经常不愿意回退，因为他们舍不得删除一行代码（即使它是错误的）。如果能让你好受些，在你开始精简之前，可以将你的代码拷贝到另一个文件中。然后你再把修改后的代码一块一块地拷回去。

发现一个错误，需要阅读、运行、沉思、和时而的回退。如果其中某个步骤没有进展，试一下其它的。

13.11 术语表

确定性的 (deterministic)：

指的是给定相同的输入，一个程序每次运行的结果是一样的。

伪随机 (pseudorandom)：

指的是一串数字看上去是随机的，但是实际是由一个确定性程序生成的。

默认值：

没有提供实参时，赋给可选形参的值。

覆盖：

用实参替代默认值。

基准测试 (benchmarking)：

通过可能的输入样本对使用不同数据结构的实现进行测试，从而选择数据结构的过程。

小黄鸭调试法 (rubberducking)：

通过向小黄鸭这样的非生物体解释你的问题来进行调试。清晰地陈述问题可以帮助你解决问题，即使小黄鸭并不懂 Python。

13.12 练习题

13.12.1 习题 13-9

单词的“秩”是指它在按照单词频率排序的列表中的位置：出现频率最高的单词，它的秩是 1，频率第二高的单词，它的秩是 2，以此类推。

Zipf 定律 (<http://en.wikipedia.org/wiki/Zipf's Law>) 描述了自然语言中秩和单词出现频率的关系。特别是，它预测对于秩为 r 的单词，其出现的频率 f 是：

$$f = cr^{-s}$$

其中， s 和 c 是依赖于语言和文本的参数。如果在上述等式两边取对数的话，你可以得到：

$$\log f = \log c - s \log r$$

因此，如果绘出 $\log f$ 和 $\log r$ 的图像，你可以得到一条以 $-s$ 为斜率、以 c 为截距的直线。

编写一个程序，从文件中读取文本，计算单词频率，倒序输出每个单词，一个单词一行，同时在这行输出对应的 $\log f$ 和 $\log r$ 。使用你喜欢的绘图程序，画出结果并检查是不是形成一条直线。你可以估算出 s 的值吗？

答案：<http://thinkpython2.com/code/zipfpy>。如果希望运行我的答案，你需要安装绘图模块 `matplotlib`。当然如果你安装了 `Anaconda`，你就已经有了 `matplotlib`；否则你需要安装它。

13.12.2 贡献者

1. 翻译：@iphycr
2. 校对：@bingjin
3. 参考：@carfly

第十四章：文件

本章将介绍“持久（persistent）”程序的概念，即永久储存数据的程序，并说明如何使用不同种类的永久存储形式，例如文件和数据库。

14.1 持久化

目前我们所见到的的大多数程序都是临时的（transient），因为它们只运行一段时间并输出一些结果，但当它们结束时，数据也就消失了。如果你再次运行程序，它将以全新的状态开始。

另一类程序是持久（persistent）的：它们长时间运行（或者一直在运行）；它们至少将一部分数据记录在永久存储（如一个硬盘中）；如果你关闭程序然后重新启动时，它们将从上次中断的地方开始继续。

持久程序的一个例子是操作系统，在一台电脑开机后的绝大多数时间系统都在运行。另一个例子是网络服务器，不停地在运行，等待来自网络的请求。

程序保存其数据的一个最简单方法，就是读写文本文件。我们已经接触过读取文本文件的程序；在本章，我们将接触写入文本的程序。

另一种方法是使用数据库保存程序的状态。本章我将介绍一个简单的数据库，以及简化存储程序数据过程的 pickle 模块。

14.2 读取和写入

文本文件是储存在类似硬盘、闪存、或者 CD-ROM 等永久介质上的字符序列。我们在[读取单词列表](#)一节中接触了如何打开和读取文件。

要写入一个文件，你必须在打开文件时设置第二个参数来为 'w' 模式：

```
>>> fout = open('output.txt', 'w')
```

如果该文件已经存在，那么用写入模式打开它将会清空原来的数据并从新开始，所以要小心！如果文件不存在，那么将创建一个新的文件。

open 会返回一个文件对象，该对象提供了操作文件的方法。write 方法将数据写入文件。

```
>>> line1 = "This here's the wattle,\n">>> fout.write(line1)24
```

返回值是被写入字符的个数。文件对象将跟踪自身的位置，所以下次你调用 `write` 的时候，它会在文件末尾添加新的数据。

```
>>> line2 = "the emblem of our land.\n">>> fout.write(line2)24
```

完成文件写入后，你应该关闭文件。

```
>>> fout.close()
```

如果你不关闭这个文件，程序结束时它才会关闭。

14.3 格式化运算符

`write` 的参数必须是字符串，所以如果想要在文件中写入其它值，我们需要先将它们转换为字符串。最简单的法是使用 `str`：

```
>>> x = 52>>> fout.write(str(x))
```

另一个方法是使用 **格式化运算符 (format operator)**，即 `%`。作用于整数时，`%` 是取模运算符，而当第一个运算数是字符串时，`%` 则是格式化运算符。

第一个运算数是 **格式化字符串 (format string)**，它包含一个或多个 **格式化序列 (format sequence)**。格式化序列指定了第二个运算数是如何格式化的。运算结果是一个字符串。

例如，格式化序列 `'%d'` 意味着第二个运算数应该被格式化为一个十进制整数：

```
>>> camels = 42>>> '%d' % camels'42'
```

结果是字符串 `'42'`，需要和整数值 `42` 区分开来。

一个格式化序列可以出现在字符串中的任何位置，所以可以将一个值嵌入到一个语句中：

```
>>> 'I have spotted %d camels.' % camels'I have spotted 42 camels.'
```

如果字符串中有多个格式化序列，那么第二个参数必须是一个元组。每个格式化序列按顺序和元组中的元素对应。

下面的例子中使用 `'%d'` 来格式化一个整数，`'%g'` 来格式化一个浮点数，以及 `'%s'` 来格式化一个字符串：

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')'In 3 years I have spotted 0.1 camels.'
```

元组中元素的个数必须等于字符串中格式化序列的个数。同时，元素的类型也必须符合对应的格式化序列：

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

在第一个例子中，元组中没有足够的元素；在第二个例子中，元素的类型错误。

可以在 <https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting> 中了解关于格式化运算符的更多信息。一个更为强大的方法是使用字符串的 `format` 方法，可以前往 <https://docs.python.org/3/library/stdtypes.html#str.format> 中了解。

14.4 文件名和路径

文件以 **目录（directory）**（也称为“文件夹（folder）”）的形式组织起来。每个正在运行的程序都有一个“当前目录（current directory）”作为大多数操作的默认目录。例如，当你打开一个文件来读取时，Python 会在当前目录下寻找这个文件。

`os` 模块提供了操作文件和目录的函数（“os”代表“operating system”）。`os.getcwd` 返回当前目录的名称：

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` 代表“current working directory”，即“当前工作目录”。在本例中，返回的结果是 `/home/dinsdale`，即用户名为 `dinsdale` 的主目录。

类似 `'/home/dinsdale'` 这样的字符串指明一个文件或者目录，叫做 **路径（path）**。

一个简单的文件名，如 `memo.txt`，同样被看做是一个路径，只不过是 **相对路径（relative path）**，因为它是相对于当前目录而言的。如果当前目录是 `/home/dinsdale`，那么文件名 `memo.txt` 就代表 `/home/dinsdale/memo.txt`。

一个以 `/` 开头的路径和当前目录无关，叫做 **绝对路径（absolute path）**。要获得一个文件的绝对路径，你可以使用 `os.path.abspath`：

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` 还提供了其它函数来对文件名和路径进行操作。例如，`os.path.exists` 检查一个文件或者目录是否存在：

```
>>> os.path.exists('memo.txt')
True
```

如果存在，可以通过 `os.path.isdir` 检查它是否是一个目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

类似的, `os.path.isfile` 检查它是否是一个文件。

`os.listdir` 返回给定目录下的文件列表 (以及其它目录)。

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

接下来演示下以上函数的使用。下面的例子“遍历”一个目录, 打印所有文件的名字, 并且针对其中所有的目录递归的调用自身。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` 接受一个目录和一个文件名, 并把它们合并成一个完整的路径。

`os` 模块提供了一个叫做 `walk` 的函数, 和我们上面写的类似, 但是功能更加更富。作为练习, 阅读文档并且使用 `walk` 打印出给定目录下的文件名和子目录。你可以从 <http://thinkpython2.com/code/walk.py> 下载我的答案。

14.5 捕获异常

试图读写文件时, 很多地方可能会发生错误。如果你试图打开一个不存在的文件夹, 会得到一个输入输出错误 (`IOError`):

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

如果你没有权限访问一个文件:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

如果你试图打开一个目录来读取, 你会得到:

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

为了避免这些错误, 你可以使用类似 `os.path.exists` 和 `os.path.isfile` 的函数来检查, 但这将会耗费大量的时间和代码去检查所有的可能性 (从 “Errno 21” 这个错误信息来看, 至少有 21 种可能出错的情况)。

更好的办法是在问题出现的时候才去处理, 而这正是 `try` 语句做的事情。它的语法类似 `if...else` 语句:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python 从 `try` 子句（clause）开始执行。如果一切正常，那么 `except` 子句将被跳过。如果发生异常，则跳出 `try` 子句，执行 `except` 子句。

使用 `try` 语句处理异常被称为是 **捕获（catching）** 异常。在本例中，`except` 子句打印出一个并非很有帮助的错误信息。一般来说，捕获异常后你可以选择是否解决这个问题，或者继续尝试运行，又或者至少优雅地结束程序。

14.6 数据库

数据库是一个用来存储数据的文件。大多数的数据库采用类似字典的形式，即将键映射到值。数据库和字典的最大区别是，数据库是存储在硬盘上（或者其他永久存储中），所以即使程序结束，它们依然存在。

`dbm` 模块提供了一个创建和更新数据库文件的接口。举个例子，我接下来创建一个包含图片文件标题的数据库。

打开数据库和打开其它文件的方法类似：

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

模式 `'c'` 代表如果数据库不存在则创建该数据库。这个操作返回的是一个数据库对象，可以像字典一样使用它（对于大多数操作）。

当你创建一个新项时，`dbm` 将更新数据库文件。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

当你访问某个项时，`dbm` 将读取文件：

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

返回的结果是一个 **字节对象（bytes object）**，这就是为什么结果以 `b` 开头。一个字节对象在很多方面都和一个字符串很像。但是当你深入了解 Python 时，它们之间的差别会变得很重要，但是目前我们可以忽略掉那些差别。

如果你对已有的键再次进行赋值，`dbm` 将把旧的值替换掉：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

一些字典方法，例如 `keys` 和 `items`，不适用于数据库对象，但是 `for` 循环依然适用：

```
for key in db:
    print(key, db[key])
```

与其它文件一样，当你完成操作后需要关闭文件：

```
>>> db.close()
```

14.7 序列化

dbm 的一个限制在于键和值必须是字符串或者字节。如果你尝试去用其它数据类型，你会得到一个错误。

pickle 模块可以解决这个问题。它可以将几乎所有类型的对象转化为适合在数据库中存储的字符串，以及将那些字符串还原为原来的对象。

pickle.dumps 读取一个对象作为参数，并返回一个字符串表示（dumps 是 “dump string” 的缩写）：

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

这个格式对人类来说不是很直观，但是对 pickle 来说很容易去解释。pickle.loads (“load string”) 可以重建对象：

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

尽管新对象和旧对象有相同的值，但它们（一般来说）不是同一个对象：

```
>>> t1 == t2
True
>>> t1 is t2
False
```

换言之，序列化然后反序列化等效于复制一个对象。

你可以使用 pickle 将非字符串对象存储在数据库中。事实上，这个组合非常常用，已经被封装进了模块 shelve 中。

14.8 管道

大多数的操作系统提供了一个命令行的接口，也被称为 **shell**。shell 通常提供浏览文件系统和启动程序的命令。例如，在 Unix 系统中你可以使用 `cd` 改变目录，使用 `ls` 显示一个目录的内容，通过输入 `firefox`（举例来说）来启动一个网页浏览器。

任何可以在 shell 中启动的程序，也可以在 Python 中通过使用 **管道对象（pipe object）** 来启动。一个管道代表着一个正在运行的程序。

例如，Unix 命令 `ls -l` 将以详细格式显示当前目录下的内容。你可以使用 `os.popen` 来启动 `ls`：

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

实参是一个包含 shell 命令的字符串。返回值是一个行为类似已打开文件的对象。你可以使用 `readline` 来每次从 `ls` 进程的输出中读取一行，或者使用 `read` 来一次读取所有内容：

```
>>> res = fp.read()
```

当你完成操作后，像关闭一个文件一样关闭管道：

```
>>> stat = fp.close()
>>> print(stat)
None
```

返回值是 `ls` 进程的最终状态。`None` 表示正常结束（没有出现错误）。

例如，大多数 Unix 系统提供了一个叫做 `md5sum` 的命令，来读取一个文件的内容并计算出一个“校验和（checksum）”。你可以在 <http://en.wikipedia.org/wiki/Md5> 中了解更多 MD5 的信息。不同的内容产生相同校验和的概率非常小（也就是说，在宇宙坍塌之前是不可能的）。

你可以使用一个管道来从 Python 中运行 `md5sum`，并得到计算结果：

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 编写模块

任何包含 Python 代码的文件，都可以作为模块被导入。例如，假设你有包含以下代码的文件 `wc.py`：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

如果你运行这个程序，它将读取自身并打印文件的行数，结果是 7。你也可以这样导入模块：

```
>>> import wc
7
```

现在你有了一个模块对象 `wc`：

```
>>> wc
<module 'wc' from 'wc.py'>
```

这个模块对象提供了 `linecount` 函数：

```
>>> wc.linecount('wc.py')
7
```

以上就是如何编写 Python 模块的方法。

这个例子中唯一的问题在于，当你导入模块后，它将自动运行最后面的测试代码。通常当导入一个模块时，它将定义一些新的函数，但是并不运行它们。

作为模块的程序通常写成以下结构：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` 是一个在程序开始时设置好的内建变量。如果程序以脚本的形式运行，`__name__` 的值为 `__main__`，这时其中的代码将被执行。否则当被作为模块导入时，其中的代码将被跳过。

我们做个练习，将例子输入到文件 `wc.py` 中，然后以脚本形式运行它。接着，打开 Python 解释器并导入 `wc`。当模块被导入后，`__name__` 的值是什么？

警示：如果你导入一个已经被导入了的模块，Python 将不会做任何事情。它并不会重新读取文件，即使文件的内容已经发生了改变。

如果你要重载一个模块，可以使用内建函数 `reload`，但它可能会出错。因此最安全的方法是重启解释器，然后重新导入模块。

14.10 调试

当你读写文件时，可能会遇到空白带来的问题。这些问题会很难调试，因为空格、制表符和换行符通常是看不见的：

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2 3
4
```

内建函数 `repr` 可以用来解决这个问题。它接受任意一个对象作为参数，然后返回一个该对象的字符串表示。对于空白符号，它将用反斜杠序列表示：

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

这个对于调试会很有用。

另一个你可能会遇到的问题是，不同的系统使用不同的符号来表示一行的结束。有些系统使用换行符 `\n`，有的使用返回符号 `\r`，有些两者都使用。如果你在不同的系统中移动文件，这些差异会导致问题。

对大多数的系统，有一些转换不同格式文件的应用。你可以在 <http://en.wikipedia.org/wiki/Newline> 中找到这些应用的信息（并阅读更多相关内容）。当然，你也可以自己编写一个转换程序。

14.11 术语表

持久性 (persistent):

用于描述长期运行并至少将一部分自身的数据保存在永久存储中的程序。

格式化运算符 (format operator):

运算符%。读取一个格式化字符串和一个元组，生成一个包含元组中元素的字符串，按照格式化字符串的要求格式化。

格式化字符串 (format string):

一个包含格式化序列的字符串，和格式化运算符一起使用。

格式化序列 (format sequence):

格式化字符串中的一个字符序列，例如%d，指定了一个值的格式。

文本文件 (text file):

保存在类似硬盘的永久存储设备上的字符序列。

目录 (directory):

一个有命名的文件集合，也叫做文件夹。

路径 (path):

一个指定一个文件的字符串。

相对路径 (relative path):

从当前目录开始的路径。

绝对路径 (absolute path):

从文件系统顶部开始的路径。

捕获 (catch):

为了防止程序因为异常而终止，使用 `try` 和 `except` 语句来捕捉异常。

数据库 (database):

一个内容结构类似字典的文件，将键映射至对应的值。

字节对象 (bytes object):

和字符串类的对象。

shell:

一个允许用户输入命令，并通过启用其它程序执行命令的程序。

管道对象 (pipe object):

一个代表某个正在运行的程序的对象，允许一个 Python 程序去运行命令并得到运行结果。

14.12 练习题

14.12.1 习题 14-1

编写一个叫做 `sed` 的函数，它的参数是一个模式字符串（`pattern string`），一个替换字符串和两个文件名。它应该读取第一个文件，并将内容写入到第二个文件（需要时创建它）。如果在文件的任何地方出现了模式字符串，就用替换字符串替换它。

如果在打开、读取、写入或者关闭文件时出现了错误，你的程序应该捕获这个异常，打印一个错误信息，并退出。

答案：<http://thinkpython2.com/code/sed.py>。

14.12.2 习题 14-2

如果你从 http://thinkpython2.com/code/anagram_sets.py 下载了习题 12-2 的答案，你会看到答案中创建了一个字典，将从一个由排序后的字母组成的字符串映射到一个可以由这些字母拼成的单词组成的列表。例如，`'opst'` 映射到列表 `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`。

编写一个模块，导入 `anagram_sets` 并提供两个新函数：函数 `store_anagrams` 在将 `anagram` 字典保存至 `shelf`，函数 `read_anagrams` 查找一个单词，并返回它的 `anagrams` 列表。

答案：http://thinkpython2.com/code/anagram_db.py。

14.12.3 习题 14-3

在一个很大的 MP3 文件集合中，或许会有同一首歌的不同拷贝，它们存放在不同的目录下或者有不同的名字。这个练习的目的是检索出这些拷贝。

1. 编写一个程序，搜索一个目录和它的所有子目录，并返回一个列表，列表中包含所有的有给定后缀（例如 `.mp3`）的文件的完整路径。提示：`os.path` 提供了一些可以操作文件和路径名的函数。
2. 为了识别出重复的文件，你可以使用 `md5sum` 来计算每个文件的“校验和”。如果两个文件的校验和相同，它们很可能有相同的内容。
3. 你可以使用 Unix 命令 `diff` 再确认一下。

答案：http://thinkpython2.com/code/find_duplicates.py。

14.12.4 贡献者

1. 翻译：[@obserthinker](#)
2. 校对：[@bingjin](#)
3. 参考：[@carfly](#)

第十五章：类和对象

目前你已经知道如何使用函数来组织你的代码，同时用内置的类型来管理数据。下一步我们将学习“面向对象编程”，即使用程序员定义的类来组织代码和数据。面向对象编程是一个很大的话题，讲完需要一些章节。

本章的示例代码可以在<http://thinkpython2.com/code/Point1.py> 获取；练习题的答案可以在http://thinkpython2.com/code/Point1_soln.py 获取。

15.1 程序员自定义类型

我们已经使用过了许多 Python 的内置类型；现在我们要定义一个新类型。举个例子，我们来创建一个叫做 `Point` 的类型，代表二维空间中的一个点。

在数学记法中，点通常被写成在两个小括号中用一个逗号分隔坐标的形式。例如 $(0,0)$ 代表原点， (x,y) 代表原点向右 x 个单位，向上 y 个单位的点。

在 Python 中，有几种表示点的方法：

- 我们可以将坐标存储在两个独立的变量， x 和 y 中。
- 我们可以将坐标作为一个列表或者元组的元素存储。
- 我们可以创建一个新类型将点表示为对象。

创建一个新类型比其他方法更复杂，但是它的优势一会儿会显现出来。

程序员自定义类型 (A programmer-defined type) 也被称作类 (class)。像这样定义一个对象：

```
class Point:
    """Represents a point in 2-D space."""
```

头部语句表明新类的名称是 `Point`。主体部分是文档字符串，用来解释这个类的用途。你可以在一个类的定义中定义变量和函数，稍后会讨论这个。

定义一个叫做 `Point` 的类将创建了一个类对象 (class object)。

```
>>> Point
<class '__main__.Point'>
```

由于 `Point` 是定义在顶层的，所以它的“全名”是 `__main__.Point`。

类对象就像是一个用来创建对象的工厂。要创建一个点，你可以像调用函数那样调用 `Point`。

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

返回值是一个 `Point` 对象的引用，我们将它赋值给 `blank`。

创建一个新对象的过程叫做**实例化**（**instantiation**），这个新对象叫做这个类的一个**实例**（**instance**）。

当你试图打印一个实例，Python 会告诉你它属于哪个类，以及它在内存中的存储地址（前缀 `0x` 代表紧跟后面的数是以十六进制表示的）。

每一个对象都是某种类的实例，所以对象和实例可以互换。但是在这章我用“实例”来表示我在讨论程序员自定义类型。

15.2 属性

你可以使用点标记法向一个实例进行赋值操作：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

这个语法类似于从一个模块中使用变量的语法，比如 `math.pi` 和 `string.whitespace`。不过在这个例子中，我们是给一个类中已命名的元素赋值。这类元素叫做**属性**（**attributes**）。

作为名词的时候，“属性”的英文“**AT-trib-ute**”的重音在第一个音节上，作为动词的时候，“**a-TRIB-ute**”重音在第二个音节上。

下面这张图展示了这些赋值操作的结果。说明一个对象及其属性的状态图叫做**对象图**（**object diagram**）；见图 15-1：对象图。

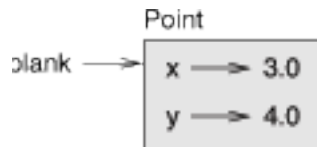


图 15.1: 图 15-1: 对象图

变量 `blank` 引用了一个 `Point` 类，这个类拥有了两个属性。每个属性都引用了一个浮点数。

你可以使用相同的语法读取一个属性的值：

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表达式 `blank.x` 的意思是，“前往 `blank` 所引用的对象并且获取 `x` 的值”。在这个例子中，我们将获取到的值赋值给了一个叫做 `x` 的变量。变量 `x` 和属性 `x` 并不会冲突。

你可以在任何表达式中使用点标记法。例如：

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

你可以将一个实例作为参数传递。例如：

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` 接受一个点作为参数，打印出其在数学中的表示方法。调用它的时候，你可以将 `blank` 作为参数传递：

```
>>> print_point(blank)
(3.0, 4.0)
```

在这个函数内部，`p` 是 `blank` 的别名，所以，如果函数修改了 `p`，`blank` 也会随之改变。

我们做个联系，编写一个叫做 `distance_between_points` 的函数，它接受两个 `Point` 作为参数，然后返回这两个点之间的距离。

15.3 矩形

有时候，一个对象该拥有哪些属性是显而易见的，但有时候你需要好好考虑一番。比如，你需要设计一个代表矩形的类。为了描述一个矩形的位置和大小，你需要设计哪些属性呢？角度是可以忽略的；为了使事情更简单，我们假设矩形是水平或者竖直的。

至少有两种可能的设计：

- 你可以指定矩形的一个角（或是中心）、宽度以及长度。
- 你可以指定对角线上的两个角。

这个时候还不能够说明哪个方法优于哪个方法。我们先来实现前者。

下面是类的定义：

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

文档字符串中列出了属性：`width` 和 `height` 是数字；`corner` 是一个 `Point` 对象，代表左下角的那个点。

为了描述一个矩形，你需要实例化一个 `Rectangle` 对象，并且为它的属性赋值：

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表达式 `box.corner.x` 的意思是，“前往 `box` 所引用的对象，找到叫做 `corner` 的属性；然后前往 `corner` 所引用的对象，找到叫做 `x` 的属性。”

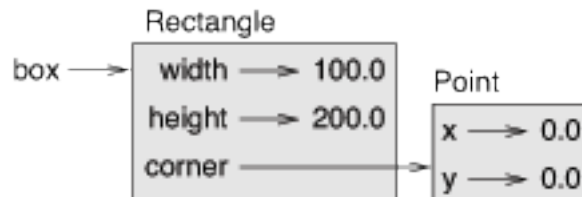


图 15.2: 图 15-2: 对象图

图 15-2: 对象图展示了这个对象的状态。一个对象作为另一个对象的属性叫做嵌套 (embedded)。

15.4 实例作为返回值

函数可以返回实例。例如，`find_center` 接受一个 `Rectangle` 作为参数，返回一个 `Point`，代表了这个 `Rectangle` 的中心坐标：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

下面这个例子将 `box` 作为参数传递，然后将返回的 `Point` 赋值给 `center`：

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 对象是可变的

你可以通过给一个对象的属性赋值来改变这个对象的状态。例如，要改变一个矩形的大小而不改变它的位置，你可以修改 `width` 和 `height` 的值：

```
box.width = box.width + 50
box.height = box.height + 100
```

你也可以编写函数来修改对象。例如，`grow_rectangle` 接受一个 `Rectangle` 对象和两个数字，`dwidth` 和 `dheight`，并将其加到矩形的宽度和高度上：

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

下面的例子展示了具体效果：

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

在函数内部，`rect` 是 `box` 的一个别名，所以如果函数修改了 `rect`，则 `box` 也随之改变。

我们做个练习，编写一个叫做 `move_rectangle` 的函数，接受一个 `Rectangle` 以及两个数字 `dx` 和 `dy`。它把 `corner` 的 `x` 坐标加上 `dx`，把 `corner` 的 `y` 坐标加上 `dy`，从而改变矩形的位置。

15.6 复制

别名会降低程序的可读性，因为一个地方的变动可能对另一个地方造成预料之外的影响。跟踪所有引用同一个对象的变量是非常困难的。

通常用复制对象的方法取代为对象起别名。`copy` 模块拥有一个叫做 `copy` 的函数，可以复制任何对象：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` 和 `p2` 拥有相同的数据，但是它们并不是同一个 `Point` 对象。

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

正如我们预期的，`is` 运算符显示了 `p1` 和 `p2` 并非同一个对象。不过你可能会认为 `==` 运算的结果应该是 `True`，因为这两个点的数据是相同的。然而结果并不如你想象的那样，`==`

运算符的默认行为和 `is` 运算符相同；它检查对象的标识（identity）是否相同，而非对象的值是否相同。因为 Python 并不知道什么样可以被认为相同。至少目前不知道。

如果你使用 `copy.copy` 来复制一个 `Rectangle`，你会发现它仅仅复制了 `Rectangle` 对象，但没有复制嵌套的 `Point` 对象。

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

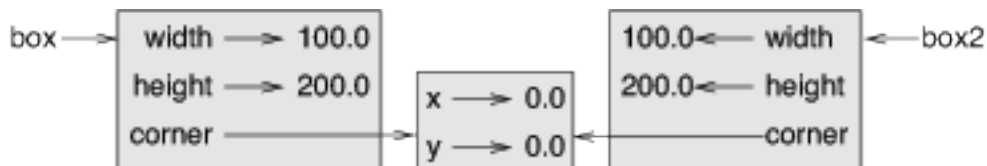


图 15.3: 图 15-3: 对象图

图 15-3: 对象图展示了相应的对象图。这个操作叫做浅复制（shallow copy），因为它仅复制了对象以及其包含的引用，但未复制嵌套的对象。

对大多数应用来说，这并非是你想要的结果。在这个例子中，对其中一个 `Rectangle` 对象调用 `grow_rectangle` 并不会影响到另外一个，然而当对任何一个 `Rectangle` 对象调用 `move_rectangle` 的时候，两者都会被影响！这个行为很容易带来疑惑和错误。

幸运的是，`copy` 模块拥有一个叫做 `deepcopy` 的方法，它不仅可以复制一个对象，还可以复制这个对象所引用的对象，甚至可以复制这个对象所引用的对象所引用的对象，等等。没错！这个操作叫做深复制（deep copy）。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` 和 `box` 是完全互不相干的对象。

我们做个练习，编写另一个版本的 `move_rectangle`，函数创建并返回一个新的 `Rectangle` 对象而非修改原先的那个。

15.7 调试

当你开始学习对象的时候，你可能会遇到一些新的异常。如果你访问一个不存在的属性，你会得到 `AttributeError` 的错误提示：

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```


如果你不确定一个对象的类型，你可以询问：

```
>>> type(p)
<class '__main__.Point'>
```

你也可以用 `isinstance` 来检查某个对象是不是某个类的实例。

```
>>> isinstance(p, Point)
True
```

如果你不确定一个对象是否拥有某个属性，你可以使用内置函数 `hasattr` 检查：

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一个参数可以是任何对象；第二个参数是一个字符串，代表了某个属性的名字。

你也可以使用 `try` 语句来检查某个对象是不是有你需要的属性：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

这个方法可以让你更容易编写出可以适应多种数据结构的函数。你可以在 [polymorphism] 节查看更多内容。

15.8 术语表

类 (class)：

一种程序员自定义的类型。类定义创建了一个新的类对象。

类对象 (class object)：

包含程序员自定义类型的细节信息的对象。类对象可以被用于创建该类型的实例。

实例 (instance)：

属于某个类的对象。

实例化 (instantiate)：

创建新的对象。

属性 (attribute)：

和某个对象相关联的有命名的值。

嵌套对象 (embedded object)：

作为另一个对象的属性存储的对象。

浅复制 (shallow copy)：

在复制对象内容的时候, 只包含嵌套对象的引用, 通过 `copy` 模块的 `copy` 函数实现。

深复制 (deep copy) :

在复制对象内容的时候, 既复制对象属性, 也复制所有嵌套对象及其中的所有嵌套对象, 由 `copy` 模块的 `deepcopy` 函数实现。

对象图 (object diagram) :

展示对象及其属性和属性值的图。

15.9 练习题

15.9.1 习题 15-1

定义一个叫做 `Circle` 的类, 类的属性是圆心 (`center`) 和半径 (`radius`), 其中, 圆心 (`center`) 是一个 `Point` 类, 而半径 (`radius`) 是一个数字。

实例化一个圆心 (`center`) 为 (150,100), 半径 (`radius`) 为 75 的 `Circle` 对象。

15.9.2 习题 15-2

编写一个名称为 `point_in_circle` 的函数, 该函数可以接受一个圆类 (`Circle`) 对象和点类 (`Point`) 对象, 然后判断该点是否在圆内。在圆内则返回 `True`。

15.9.3 习题 15-3

编写一个名称为 `rect_in_circle` 的函数, 该函数接受一个圆类 (`Circle`) 对象和矩形 (`Rectangle`) 对象, 如果该矩形是否完全在圆内或者在圆上则返回 `True`。

15.9.4 习题 15-4

编写一个名为 `rect_circle_overlap` 函数, 该函数接受一个圆类对象和一个矩形类对象, 如果矩形有任意一个角落在圆内则返回 `True`。或者写一个更具有挑战性的版本, 如果该矩形有任何部分落在圆内返回 `True`。

答案:<http://thinkpython2.com/code/Circle.py>.

15.9.5 习题 15-5

编写一个名为 `draw_rect` 的函数, 该函数接受一个 `Turtle` 对象和一个 `Rectangle` 对象, 使用 `Turtle` 画出该矩形。参考 [turtlechap] 章中使用 `Turtle` 的示例。

15.9.6 习题 15-6

编写一个名为 `draw_circle` 的函数, 该函数接受一个 `Turtle` 对象和 `Circle` 对象, 并画出该圆。

答案:<http://thinkpython2.com/code/draw.py>.

15.9.7 贡献者

1. 翻译: @iphyer
2. 校对: @bingjin
3. 参考: @carfly

第十六章：类和函数

现在我们已经知道如何去定义一个新的类型，下一步就是编写以自定义对象为参数的函数，并返回自定义对象作为结果。在本章中，我还将介绍“函数式编程风格”和两种新的编程开发方案。

本章的代码示例可以从 <http://thinkpython2.com/code/Time1.py> 下载。练习的答案可以从 http://thinkpython2.com/code/Time1_soln.py 下载。

16.1 时间

再举一个程序员自定义类型的例子，我们定义一个叫 `Time` 的类，用于记录时间。这个类的定义如下：

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

我们可以创建一个新的 `Time` 类对象，并且给它的属性 `hour`、`minutes` 和 `seconds` 赋值：

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

`Time` 对象的状态图类似于图 16-1：对象图。

我们做个练习，编写一个叫做 `print_time` 的函数，接收一个 `Time` 对象并用“时: 分: 秒”的格式打印它。提示：格式化序列 `'%.2d'` 可以至少两位数的形式打印一个整数，如果不足则在前面补 0。

编写一个叫做 `is_after` 的布尔函数，接收两个 `Time` 对象，`t1` 和 `t2`，若 `t1` 的时间在 `t2` 之后，则返回 `True`，否则返回 `False`。挑战：不要使用 `if` 语句。

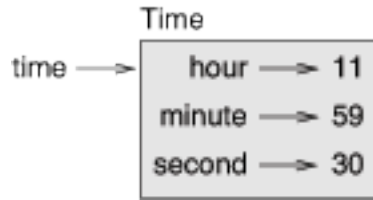


图 16.1: 图 16-1: 对象图

16.2 纯函数

在下面几节中,我们将编写两个用来增加时间值的函数。它们展示了两种不同的函数: 纯函数 (pure functions) 和修改器 (modifiers)。它们也展示了我所称的 **原型和补丁 (prototype and patch)** 的开发方案。这是一种处理复杂问题的方法,从简单的原型开始,逐步解决复杂情况。

下面是一个简单的 `add_time` 原型:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

这个函数创建了一个新的 `Time` 对象,初始化了对象的属性,并返回了这个对象的引用。我们把这个函数称为 **纯函数 (pure function)**,因为它除了返回一个值以外,并不修改作为参数传入的任何对象,也没有产生如显示一个值或者获取用户输入的影响。

为了测试这个函数,我将创建两个 `Time` 对象: `start` 用于存放一个电影 (如 *Monty Python and the Holy Grail*) 的开始时间, `duration` 用于存放电影的放映时长,这里时长定为 1 小时 35 分钟。

`add_time` 将计算电影何时结束。

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

这个结果 10:80:00 可能不是你所希望得到的。问题在于这个函数并没有处理好秒数和分钟数相加超过 60 的情况。当发生这种情况时,我们要把多余的秒数放进分钟栏,或者把多余的分钟加进小时栏。

下面是一个改进的版本:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

这个函数虽然正确，但是它开始变得臃肿。我们会在后面看到一个较短的版本。

16.3 修改器

有时候用函数修改作为参数传入的对象是很有用的。在这种情况下，这种改变对调用者来说是可见的。这种方式工作的函数称为 **修改器（modifiers）**。

函数 `increment` 给一个 `Time` 对象增加指定的秒数，可以很自然地用修改器来编写。下面是一个初稿：

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

第一行进行基础操作；其余部分的处理则是我们之前看到的特殊情况。

这个函数正确吗？如果 `seconds` 比 60 大很多会发生什么？

在那种情况下，只进位一次是不够的；我们要重复执行直到 `seconds` 小于 60。一种方法是用 `while` 语句代替 `if` 语句。这样能够让函数正确，但是并不是很高效。

我们做个练习：编写正确的 `increment` 函数，不能包含任何循环。

任何能够用修改器实现的函数同样能够用纯函数实现。事实上，一些编程语言只允许用纯函数。一些证据表明用纯函数实现的程序比用修改器实现的函数开发更快、更不易出错。但是有时候修改器是很方便的，而函数式程序效率反而不高。

通常来说，我推荐只要是合理的情况下，都使用纯函数方式编写，只有在有完全令人信服的原因下采用修改器。这种方法可以称为 **函数式编程风格（functional programming style）**。

我们做个练习，编写一个纯函数版本的 `increment`，创建并返回一个 `Time` 对象，而不是修改参数。

16.4 原型 vs. 方案

我刚才展示的开发方案叫做 **原型和补丁 (prototype and patch)**。针对每个函数，我编写了一个可以进行基本运算的原型并对其测试，逐步修正错误。

这种方法在你对问题没有深入理解时特别有效。但增量修正可能导致代码过度复杂，因为需要处理许多特殊情况。也并不可靠，因为很难知道你是否已经找到了所有的错误。

另一种方法叫做 **设计开发 (designed development)**。对问题有高层次的理解能够使开发变得更容易。这给我们的启示是，`Time` 对象本质上是一个基于六十进制的三位数（详见 <http://en.wikipedia.org/wiki/Sexagesimal>。）！属性 `second` 是“个位”，属性 `minute` 是“六十位”，属性 `hour` 是“360 位数”。

当我们编写 `add_time` 和 `increment` 时，其实是在基于六十进制累加，所以我们需要把一位进位到下一位。

这个观察意味着我们可以用另一种方法去解决整个问题——我们可以把 `Time` 对象转换为整数，并利用计算机知道如何进行整数运算的这个事实。

下面是一个把 `Time` 对象转成整数的函数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

下面则是一个把整数转换为 `Time` 对象（记得 `divmod` 是用第一个参数除以第二个参数并以元组的形式返回商和余数）。

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

你可能需要思考一下，并运行一些测试，以此来说服自己这些函数是正确的。一种测试方法是对很多的 `x` 检查 `time_to_int(int_to_time(x)) == x` 是否正确。这是一致性检查的例子。

一旦你确信它们是正确的，你就能使用它们重写 `add_time`：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

这个版本比先前的要更短，更容易校验。我们再做个练习，使用 `time_to_int` 和 `int_to_time` 重写 `increment` 函数。

从某个方面来说，六十进制和十进制相互转换比处理时间更难些。进制转换更加抽象；我们解决时间值的想法是更好的。

但如果我们意识到把时间当作六十进制，并预先做好编写转换函数（`time_to_int` 和 `int_to_time`）的准备，我们就能获得一个更短、更易读、更可靠的程序。

这让我们日后更加容易添加其它功能。例如，试想将两个 `Time` 对象相减来获得它们之间的时间间隔。最简单的方法是使用借位来实现减法。使用转换函数则更容易，也更容易正确。

讽刺的是，有时候把一个问题变得更难（或更加普遍）反而能让它更加简单（因为会有更少的特殊情况和更少出错的机会）。

16.5 调试

如果 `minute` 和 `second` 的值介于 0 和 60 之间（包括 0 但不包括 60），并且 `hour` 是正值，那么这个 `Time` 对象就是合法的。`hour` 和 `minute` 应该是整数值，但我们可能也允许 `second` 有小数部分。

这样的要求称为 **不变式（invariants）**。因为它们应当总是为真。换句话说，如果它们不为真，肯定是某些地方出错了。

编写代码来检查不变式能够帮助检测错误并找到出错的原因。例如，你可能会写一个 `valid_time` 这样的函数，接受一个 `Time` 对象，并在违反不变式的条件下返回 `False`：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

在每个函数的开头，你可以检查参数，确认它们是否合法：

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

或者你可以使用 **assert 语句**，检查一个给定的不变式并在失败的情况下抛出异常：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` 语句非常有用，因为它们区分了处理普通条件的代码和检查错误的代码。

16.6 术语表

原型和补丁（prototype and patch）：

一种开发方案，编写一个程序的初稿，测试，发现错误时修正它们。

设计开发 (designed development):

一种开发方案, 需要对问题有更深层次的理解, 比增量开发或原型开发更有计划性。

纯函数 (pure function):

一种不修改任何作为参数传入的对象的函数。大部分纯函数是有返回值的 (fruitful)。

修改器 (modifier):

一种修改一个或多个作为参数传入的对象的函数。大部分修改器没有返回值; 即返回 `None`。

函数式编程风格 (functional programming style):

一种程序设计风格, 大部分函数为纯函数。

不变式 (invariant):

在程序执行过程中总是为真的条件。

断言语句 (assert statement):

一种检查条件是否满足并在失败的情况下抛出异常的语句。

16.7 练习题

本章的代码示例可以从 <http://thinkpython2.com/code/Time1.py> 下载; 练习的答案可以从 http://thinkpython2.com/code/Time1_soln.py 下载。

16.7.1 习题 16-1

编写一个叫做 `mul_time` 的函数, 接收一个 `Time` 对象和一个数, 并返回一个新的 `Time` 对象, 包含原始时间和数的乘积。

然后使用 `mul_time` 编写一个函数, 接受一个表示比赛完赛时间的 `Time` 对象以及一个表示距离的数字, 并返回一个用于表示平均配速 (每英里所需时间) 的 `Time` 对象。

16.7.2 习题 16-2

`datetime` 模块提供的 `time` 对象, 和本章的 `Time` 对象类似, 但前者提供了更丰富的方法和操作符。可以在 <http://docs.python.org/3/library/datetime.html> 阅读相关文档。

1. 使用 `datetime` 模块来编写一个程序, 获取当前日期并打印当天是周几。
2. 编写一个程序, 接受一个生日作为输入, 并打印用户的年龄以及距离下个生日所需要的天数、小时数、分钟数和秒数。
3. 对于两个不在同一天出生的人来说, 总有一天, 一个人的出生天数是另一个人的两倍。我们把这一天称为“双倍日”。编写一个程序, 接受两个不同的出生日期, 并计算他们的“双倍日”。

4. 再增加点挑战, 编写一个更通用的版本, 用于计算一个人出生天数是另一个人 n 倍的日子。

答案: <http://thinkpython2.com/code/double.py>。

16.7.3 贡献者

1. 翻译: @cxyfreedom
2. 校对: @bingjin
3. 参考: @carfly

第十七章：类和方法

虽然我们已经在使用部分 Python 面向对象的特性，前两个章节中的程序并不是真正面向对象的，因为它们没有呈现出程序员自定义类型与对其进行操作的功能（**functions**）之间的关系。下一步，我们将会把这些函数转换成明显突出这一关系的方法（**methods**）。

本章代码可以从<http://thinkpython2.com/code/Time2.py> 获取，练习题的答案位于http://thinkpython2.com/code/Point2_soln.py。

17.1 面向对象的特性

Python 是一门 **** 面向对象的编程语言 ****，这意味它提供了能够支持面向对象编程的特性。面向对象编程具有以下特征：

- 程序包含类和方法定义。
- 大部分计算以对象上的操作表示。
- 对象通常代表现实世界的物体，方法对应现实世界中物体交互的方式。

例如，第 16 章中定义的 **Time** 类对应人们用来记录一天中的时间，其中定义的各种函数对应人们使用时间的方式。类似的，第 15 章中的 **Point** 类和 **Rectangle** 类对应数学中点和矩形的概念。

到目前为止，我们还没有利用 Python 提供的支持面向对象编程的特性。这些特性严格来说并不是必须的；大部分提供的是我们已经实现的功能的替代语法。但在很多情况下，这些替代语法更加简洁，更准确地表达了程序的结构。

例如，在 **Time1.py** 中，类定义与之后的函数定义之间没有明显的联系。仔细检查之后，才会发现每个函数都至少接受一个 **Time** 对象作为参数。

从这个观察中我们发现了**方法**：方法是一个与特定的类相关联的函数。我们已经接触了字符串、列表、字典和元组的方法。在这章中，我们将会定义程序员自定义类型的方法。

方法和函数的语义相同，但是有两处句法的不同：

- 方法在一个类定义内部声明，为的是显示地与类进行关联。
- 调用方法的语法和调用函数的语法不同。

在接下来的几节中，我们会把前面两章中的函数转化为方法。这个转化是纯机械式的；你可以通过一系列步骤完成。如果你能够轻松地将一种形式转换成另一种形式，就可以选择最适合目前需求的形式。

17.2 打印对象

在第 16 章中, 我们定义了一个名叫 `Time` 的类, 在时间一节中, 你编写了一个叫做 `print_time` 的函数:

```
class Time:
    """Represents the time of day."""

    def print_time(time):
        print('%02d:%02d:%02d' % (time.hour, time.minute, time.second))
```

想要调用这个函数, 你必须把一个 `Time` 对象作为一个参数传递给函数。

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

将 `print_time` 变成一个方法, 我们只需要将函数定义移到类定义里面即可。注意缩进的变化。

```
class Time:
    def print_time(time):
        print('%02d:%02d:%02d' % (time.hour, time.minute, time.second))
```

现在有两种方法可以调用 `print_time`。第一种 (也是不常用的) 是使用函数的语法:

```
>>> Time.print_time(start)
09:45:00
```

在这个点标记法的用法中, `Time` 是类的名字, `print_time` 是方法的名字。 `start` 是传递的参数。

第二种语法 (也更简洁) 是使用方法语法:

```
>>> start.print_time()
09:45:00
```

在这个点标记法的用法中, `print_time` 是方法的名称, 然后 `start` 是调用方法的对象, 被称为主语 (**subject**)。就像一个句子的主语是句子的核心, 方法的主语也是方法作用的主要对象。

在方法中, 主语被赋值为第一个参数, 所以在这里 `start` 被赋值给 `time` 上了。

根据约定, 方法的第一个参数写作 `self`, 所以 `print_time` 写成这样更常见:

```
class Time:
    def print_time(self):
        print('%02d:%02d:%02d' % (self.hour, self.minute, self.second))
```

使用该约定原因在于一种暗喻:

- 在函数调用的语法中, `print_time(start)` 表示函数是一个活跃的代理。就像是在说 “Hi, `print_time`! 这有一个对象需要你打印”。

- 在面向对象编程中，对象是活跃的代理。一个类似 `start.print_time()` 的方法调用，就像是在说“Hi start! 请打印你自己”。

视角的转换似乎让语气变得更文雅些了，但很难看出其好处。在前面的例子中，的确如此。但是将职责从函数上面转移到对象上，可以更加容易地写出多样化的函数（或方法），并且代码将更加容易维护和复用。

我们做个练习，将 `time_to_int`（见原型 *vs.* 方案）重写为方法。你或许也想将 `int_to_time` 改写为方法，但是那样做并没有什么意义，因为没有调用它的对象。

17.3 再举一例

下面是 `increment`（见修改器）改写为方法后的代码版本：

```
# inside class Time:

def increment(self, seconds):
    seconds += self.time_to_int()
    return int_to_time(seconds)
```

这个版本假设 `time_to_int` 已经改成了方法。另外，注意这是一个纯函数，不是修改器。

下面是调用 `increment` 的方法：

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

主语 `start` 被赋值给第一个形参 `self`。实参 `1337` 被赋值给第二个形参 `seconds`。

这个机制有时会把人弄晕，尤其是你犯错的时候。例如，如果你使用两个实参调用 `increment`，你会得到：

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

错误信息一开始让人很难理解，因为在括号内只有两个实参。但是主语也被认为是一个实参，所以加在一起共有三个实参。

另外，位置参数是没有形参名的参数；也就是说，它不是一个关键字参数。在下面这个函数调用中：

```
sketch(parrot, cage, dead=True)
```

`parrot` 和 `cage` 是位置参数，而 `dead` 是一个关键字参数。

17.4 一个更复杂的例子

重写 `is_after`（见时间一节）要更加复杂一些，因为它接受两个 `Time` 对象作为参数。在这个例子中，惯用的做法是将第一个形参命名为 `self`，第二个形参命名为 `other`：

```
# inside class Time:

def is_after(self, other):
    return self.time_to_int() > other.time_to_int()
```

要使用该方法的话，你必须在某个对象上调用它，并传入 `other` 的实参：

```
>>> end.is_after(start)
True
```

这个语法有一个好处，就是它读起来很像英语：“`end` 是出现在 `start` 之后吗？”

17.5 `init` 方法

`init` 方法（“initialization”的简称）是一个特殊的方法，当一个对象初始化的时候调用。它的全名是 `__init__`（两个下划线后加 `init` 再加两个下划线）。一个 `Time` 类的 `init` 方法看起来像是这样的：

```
# inside class Time:

def __init__(self, hour=0, minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second
```

通常 `__init__` 方法的参数和属性的名称一样。

```
self.hour = hour
```

上面的语句把 `hour` 参数的值储存为 `self` 的一个属性。

参数是可选的，所以如果你不带参数的调用 `Time`，你会得到默认值。

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

如果你提供一个参数，它会覆盖 `hour`：

```
>>> time = Time(9)
>>> time.print_time()
09:00:00
```

如果你提供两个参数，他们会覆盖 `hour` 和 `minute`。

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

如果你提供三个参数，它们会覆盖三个默认值。

我们做个练习，为 `Point` 类写一个 `init` 方法，使用 `x` 和 `y` 作为可选参数，然后赋值给对应的属性。

17.6 __str__ 方法

`__str__` 是一个和 `__init__` 方法类似的特殊方法，返回一个对象的字符串表现形式。

例如，下面是一个 `Time` 对象的 `str` 方法：

```
# inside class Time:

def __str__(self):
    return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

当你打印一个对象，Python 调用 `str` 方法：

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

写一个新类时，我总是从 `__init__` 开始，使得更容易实例化对象，接着就是写 `__str__` 方法，方便调试。

我们做个练习，为 `Point` 类写一个 `str` 方法。然后创建一个 `Point` 对象并打印。

17.7 运算符重载

通过定义其它的一些特殊方法，你可以在程序员自定义类型上指定运算符的行为。例如，如果你为 `Time` 类定义了一个叫 `__add__` 的方法，你就可以在 `Time` 对象上使用 `+` 运算符。

可以大致像这样定义：

```
# inside class Time:

def __add__(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
```

下面是使用方式：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

当你在 `Time` 对象上应用 `+` 运算符，Python 会调用 `__add__`。当你打印结果时，Python 会调用 `__str__`。所以实际上后台发生了很多有趣的事情！

改变一个运算符的行为，使其兼容程序员自定义类型，这被称为**运算符重载**（**operator overloading**）。对于每一个运算符，Python 有一个类似 `__add__` 的对应的特殊方法。更多的详情，请参考 <http://docs.python.org/3/reference/datamodel.html#specialnames>。

我们做个练习，为 `Point` 类编写一个 `add` 方法。

17.8 类型分发 (type-based dispatch)

在上一节中, 我们将两个 `Time` 对象相加, 但是你还想要将一个整数与 `Time` 对象相加。下面这个版本的 `__add__` 会检查 `other` 的类型, 并相应地调用 `add_time` 或者 `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)
```

内建函数 `isinstance` 接受一个值和一个类对象, 如果值是这个类的实例则返回 `True`。

如果 `other` 是一个 `Time` 对象, `__add__` 调用 `add_time`。否则它假设参数是一个数字然后调用 `increment`。这个操作被称为类型分发 (type-based dispatch), 因为它根据参数的类型将计算任务分发给不同的方法。

下面是一些在不同类型上使用 `+` 运算符的例子:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

不幸的是, 这个加法的实现没有交换性 (commutative)。如果第一个运算数是一个整数, 你会得到:

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

问题在于, 我们不是让一个 `Time` 对象去加一个整数, 而是让一个整数去加一个 `Time` 对象, 但是 Python 不知道怎样去做。不过这个问题有一个优雅的方案: 特殊方法 `__radd__`, 表示“右手加法”。当一个 `Time` 对象在 `+` 运算符的右边出现时, 调用这个方法。下面是定义:

```
# inside class Time:

def __radd__(self, other):
    return self.__add__(other)
```

接着是使用方法:

```
>>> print(1337 + start)
10:07:17
```

我们做个练习，为 `Points` 编写一个 `add` 方法，使其既适用 `Point` 对象，也适用元组：

- 如果第二个运算数是一个 `Point`，该方法将返回一个新的 `Point`，其 x 坐标是两个运算数的 x 的和， y 以此类推。
- 如果第二个运算数是一个元组，该方法将把元组的第一个元素与 x 相加，第二个元素与 y 相加，然后返回以相关结果为参数的新的 `Point`。

17.9 多态性

类型分发在必要的时候非常有用，但是（幸运的是）它不是绝对必须的。通常，你可以通过编写对不同参数类型都适用的函数，来避免这种情况。

许多我们为字符串写的函数，实际上也适用于其他序列类型。例如，在[字典作为计数器集合](#)一节中，我们使用 `histogram` 计算了单词中每个字母出现的次数。

```
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
```

这个函数也适用于列表、元组甚至是字典，只要 `s` 的元素是可哈希的，你就可以把它用作 `d` 的键。

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

适用于多种类型的函数，被称为**多态函数**。多态性有助于代码复用。例如，内建函数 `sum` 对一个序列的元素求和，只要序列中的元素支持加法即可。

因为 `Time` 对象提供了一个 `add` 方法，`sum` 也可以应用于该对象：

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

通常，如果一个函数内所有的操作都适用于一个类型，那这个函数就能适用该类型。

最好的多态性是无心成柳柳成荫的，就是你发现你已经写的一个函数，在你没有预计的类型上也能使用。

17.10 接口和实现

面向对象设计的一个目标是使得软件更容易维护，这意味着当系统的其它部分改变时程序还能正常运行，你可以修改程序满足新的需求。

有助于实现该目标的一个设计原则是，接口和实现分离。对于对象，就意味着一个类提供的方法不应该依赖属性的形式。

例如，在本章中，我们设计了一个表示一天中时间的类。这个类提供的方法包括 `time_to_int`, `is_after` 和 `add_time`。

我们有多种方式可以实现这些方法。实现的细节取决于我们如何表示时间。在本章中，`Time` 对象的属性是 `hour`, `minute` 和 `second`。

另一种方式是，我们用一个整数表示从零点开始的秒数，来替代这些属性。这个实现会使得一些方法（如 `is_after`）更容易编写，但也让编写其他方法变得更难。

在你完成一个新类后，你可能会发现有一个更好的实现。如果程序其他部分使用了你的类，再来改变接口需要很多时间，而且容易出错。

但是如果你细心设计好接口，你可以改变实现而保持接口不变，这样程序的其它部分都不用改变。

17.11 调试

在程序执行的任何时间，为一个对象添加属性都是合法的，但是如果相同类型的对象拥有不同的属性，就会很容易出现错误。通常一个好的做法是在 `init` 方法中初始化一个对象的所有属性。

如果你不确定一个对象是否应该有某个属性，你可以使用内建函数 `hasattr`（参见调试一节）。

另一种访问对象属性的方法是使用内建函数 `vars`，它接受一个对象，并返回一个将属性名称（字符串形式）到对应值的字典：

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

定义下面这段代码，可能对调试非常有用：

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` 遍历一个对象的字典，然后打印每个属性的名称和对应的值。

内建函数 `getattr` 接受一个对象和一个属性名称（字符串）作为参数，然后返回该属性的值。

17.12 术语表

面向对象的语言 (object-oriented language):

提供有助于面向对象编程特性的语言, 如程序员自定义类型和方法。

面向对象编程 (object-oriented programming):

一种编程风格, 数据和处理数据的操作被组织成类和方法。

方法 (method):

在类定义内部定义的一个函数, 必须在该类的实例上调用。

主语 (subject):

方法在该对象上调用。

位置参数 (positional argument):

不包括形参名的实参, 所以不是关键字实参。

运算符重载 (operator overloading):

改变类似 + 的运算符, 使其可以应用于程序员自定义类型。

类型分发 (type-based dispatch):

一种检查运算符的类型, 并根据类型不同调用不同函数的编程模式。

多态的 (polymorphic):

描述一个可应用于多种类型的函数。

信息隐藏 (information hiding):

对象提供的接口不应依赖于其实现的原则, 尤其是其属性的表示形式。

17.13 练习题

17.13.1 习题 17-1

可以从 <http://thinkpython2.com/code/Time2.py> 下载本章的代码。修改 `Time` 类的属性, 使用一个整数代表自午夜零点开始的秒数。然后修改类的方法 (和 `int_to_time` 函数), 使其适用于新的实现。你不用修改 `main` 函数中的测试代码。

完成之后, 程序的输出应该和之前保持一致。答案: http://thinkpython2.com/code/Time2_soln.py。

17.13.2 习题 17-2

这道习题中包含了 Python 中最常见、最难找出来的错误。编写一个叫 `Kangaroo` 的类, 包含以下方法:

1. 一个 `__init__` 方法, 初始化一个叫 `pouch_contents` 的属性为空列表。

2. 一个叫 `put_in_pouch` 的方法，将一个任意类型的对象加入 `pouch_contents`。
3. 一个 `__str__` 方法，返回 `Kangaroo` 对象的字符串表示和 `pouch` 中的内容。

创建两个 `Kangaroo` 对象，将它们命名为 `kanga` 和 `roo`，然后将 `roo` 加入 `kanga` 的 `pouch` 列表，以此测试你写的代码。

下载<http://thinkpython2.com/code/BadKangaroo.py>。其中有一个上述习题的答案，但是有一个又大又棘手的 `bug`。找出并修正这个 `bug`。

如果你找不到 `bug`，可以下载 <http://thinkpython2.com/code/GoodKangaroo.py>，里面解释了问题所在并提供了一个解决方案。

17.13.3 贡献者

1. 翻译: @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第十八章：继承

最常与面向对象编程联系在一起的语言特性就是 **继承**。继承指的是在现有类的基础上进行修改，从而定义新类的能力。在本章中，我会用表示卡牌（`playing cards`）、一副牌（`deck of hands`）和牌型（`poker hands`）的类，来展示继承这一特性。

如果你不玩扑克牌，你可以阅读 <http://en.wikipedia.org/wiki/Poker> 了解一下，但这不是必须的；我会告诉你完成练习所需要了解的知识点。

本章的代码示例可以从 <http://thinkpython2.com/code/Card.py> 下载。

18.1 卡牌对象

一副牌有 52 张牌，每一张属于 4 种花色中的一个和 13 个等级的一个。4 种花色是黑桃（`Spades`），红心（`Hearts`），方块（`Diamonds`），梅花（`Clubs`），以桥牌中的逆序排列。13 个等级是 A、2、3、4、5、6、7、8、9、10、J、Q、K。根据你玩的游戏的不同，A 可能比 K 大或者比 2 小。

如果我们定义一个新的对象来表示卡牌，明显它应该有 `rank`（等级）和 `suit`（花色）两个属性。但两个属性的类型不太明显。一个可能是使用字符串类型，如 `'Spade'` 表示花色，`'Queen'` 表示等级。这种实现的一个问题是，不是那么容易比较牌的大小，看哪张牌的等级或花色更高。

另外一种方法，是使用一个整型来 **编码** 等级和花色。在这里，“编码”表示我们要定义一个数字到花色或数字到等级的映射。但是这里的编码并不是为了保密（那就成了“加密”）。

例如，下面的表格列出了花色和对应的整数码：

Spades	↦	3
Hearts	↦	2
Diamonds	↦	1
Clubs	↦	0

整数码使得很容易比较牌的大小；因为更高的花色对应更高的数字，我们可以通过比较数字，来判断花色的的大小。

等级的映射类型选择就显而易见；每个数字等级对应相应的整数，然后对于 J, K, Q:

Jack	↦	11
Queen	↦	12
King	↦	13

这里，我使用 \mapsto 符号来清楚的表示，这些不是 Python 程序的一部分。它们属于程序设计的一部分，但是不会出现在代码中。

Card 的类定义如下：

```
class Card:
    """ 22222222 """

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

通常，init 方法接受针对每个属性的可选形参。默认的卡牌是梅花 2。

可以使用你需要的花色和等级调用 Card，创建一个 Card 对象。

```
queen_of_diamonds = Card(1, 12)
```

18.2 类属性

为了让大家能够轻松看懂的方式来打印卡牌对象，我们需要一个从整数码到对应的等级和花色的映射。一种直接的方法是使用字符串列表。我们把这些列表赋值到**类属性**：

```
# 2 Card 222:

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])
```

像 suit_names 和 rank_names 这样的变量，是定义在类内部但在方法之外，被称为**类属性**。因为他们是被关联到 Card 类对象上的。

这个术语将它们同 suit 和 rank 这样的变量区分开来，后者被称为**实例属性**，因为他们被关联到了特定的实例。

这两种属性都使用点标记法来访问。例如，在 __str__ 中，self 是一个卡牌对象，self.rank 是它的等级。同样的，Card 是一个类对象，Card.rank_names 是一个和类关联的字符串列表。

每一张卡牌都有自己的花色和等级，但是这里只有一份 suit_names 和 rank_names 拷贝。

综合来说，表达式 Card.rank_names[self.rank] 表示“使用 self 对象中的 rank 属性作为 Card 类的 rank_names 列表的索引下标，然后获取相应的字符串。”

`rank_names` 的第一个元素是 `None`，因为没有卡牌的等级是 0。通过使用 `None` 作为占位符，我们可以很好地将索引 2 映射到字符串 '2'，等等。为了避免使用这种小技巧，我们也可以使用一个字典来代替列表。

利用现有的方法，我们可以创建和打印卡牌：

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

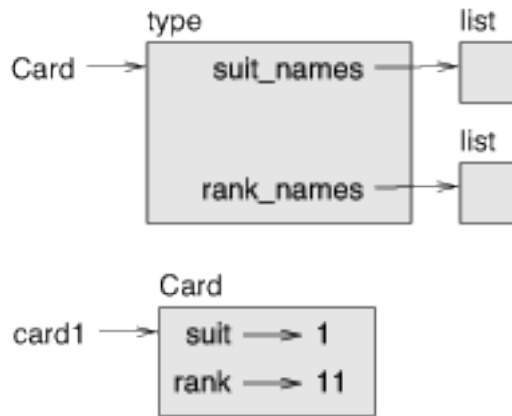


图 18.1: 图 18-1: 对象图

图 18-1: 对象图是 `Card` 类对象和一个 `Card` 实例的图示。`Card` 是一个类对象；它的类型是 `type`。`card1` 是 `Card` 的一个实例，因此它的类型是 `Card`。为了节省空间，我没有画出 `suit_names` 和 `rank_names` 的内容。

18.3 比较卡牌

对于内建类型，有关系运算符 (`<`, `>`, `==`, 等等) 可以比较值，判断哪一个是大于、小于或等于另外一个。对于程序员自定义的类型，我们可以通过提供一个叫 `__lt__`（代表“小于”）的方法，来覆盖内建运算符的行为。

`__lt__` 接受 2 个参数, `self` 和 `other`，如果 `self` 比 `other` 的值要小则返回 `True`。

卡牌的正确顺序并不明显。例如，梅花 3 和方块 2 哪个更高？一个等级更高，另一个花色更高。为了比较卡牌，你必须决定等级还是花色更重要。

答案可能根据你玩的是什么游戏而不同，但是简洁起见，我们将规定花色更重要，所以所有的黑桃大于任何方块卡牌，以此类推。

定好了这个规则后，我们可以编写 `__lt__` 了：

```
# Card 类:

def __lt__(self, other):
    # ???
    if self.suit < other.suit: return True
    if self.suit > other.suit: return False
```

```
# 返回...
return self.rank < other.rank
```

你可以使用元组比较来使得代码更加简洁:

```
# Card 比较:

def __lt__(self, other):
    t1 = self.suit, self.rank
    t2 = other.suit, other.rank
    return t1 < t2
```

我们做个练习, 编写一个 Time 对象的 `__lt__` 方法。你可以使用元组比较, 也可以考虑比较整数。

18.4 一副牌

现在我们有 Card 类了, 下一步是定义完整的一副牌 (Deck) 了。因为一副牌由许多牌组成, 自然地每一个 Deck 都有一个卡牌列表作为属性。

下面是一个 Deck 的类定义。初始化方法创建了 `cards` 属性, 然后生成了由 52 张牌组成一副标准卡牌。

```
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)
```

生成一副牌的最简单方法是使用嵌套循环。外层循环枚举 0 到 3 的花色。内层循环枚举 1 到 13 的等级。每一个迭代都用当前的花色和等级创建一张新的牌。然后放入 `self.cards` 中。

18.5 打印一副牌

下面是为 Deck 定义的 `__str__` 方法:

```
# Deck 打印

def __str__(self):
    res = []
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

这个方法展示了累积大字符串的高效方法: 建立一个字符串列表然后使用字符串方法 `join`。内建函数 `str` 会调用每个卡牌上的 `__str__` 方法, 并返回它们的字符串表示。

由于我们是在一个换行符上调用的 `join`, 卡牌之间被换行符分隔。下面是结果示例:

```
>>> 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
```

虽然这个结果有 52 行，但他实际上是包含换行符的一个长字符串。

18.6 添加，移除，洗牌和排序

为了发牌，我们需要一个可以把卡牌从一副牌中移除并返回的方法。列表的 `pop` 方法提供了一个便捷的实现：

```
# 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.

由于 `pop` 移除列表的最后一张卡牌，所以我们从牌底开始发牌。

我们可以使用列表的 `append` 方法，添加一张卡牌：

```
# Deck [?]

def add_card(self, card):
    self.cards.append(card)
```

像上面这样利用别的方法（method），自己却没有做太多处理的方法，有时候被称为 **伪装方法（veneer）**。这个隐喻来源于木工行业，他们通常用一片高质量的木质薄层粘贴在一块便宜木材的表面，改善外观形象。

在这里，`add_card` 是一个“瘦”方法，以卡牌的术语来表述一个列表操作。它改善了实现的外观，或者说接口。

再举一个例子，我们可以用 `random` 模块中的 `shuffle` 函数，给 `Deck` 写一个叫 `shuffle` 的方法。

```
# Deck [?]

def shuffle(self):
    random.shuffle(self.cards)
```

不要忘记了导入 `random`。

我们做个练习，用列表的 `sort` 方法来写一个 `Deck` 的 `sort` 方法，给卡牌排序。`sort` 使用我们定义的 `__cmp__` 来决定排序顺序。

18.7 继承

继承指的是在现有类的基础下进行修改，从而定义新类的能力。例如，假设我们想定义一个类来代表手牌（hand），即玩家目前手里有的牌。手牌与一副牌（deck）类似：二者都由卡牌组成，都要求支持添加和移除卡牌的操作。

但二者也有区别；有些我们希望手牌具备的操作，对于 deck 来说并不合理。例如，在扑克牌中，我们可能需要比较两个手牌，比较哪方赢了。在桥牌中，我们可能需要计算手牌的得分，才好下注。

类之间有相似之处，但也存在不同，这时就可以用上继承了。你只需要在定义新类时，将现有类的名称放在括号里，即可继承现有类：

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

这个定义表明，Hand 继承自 Deck；这意味着我们也可以对 Hands 使用 Deck 的 pop_card 和 add_card 方法。

当一个新类继承自一个现有类时，现有类被称为 **父类**，新类被称为 **子类**。

在此例中，Hand 继承了 Deck 的 __init__ 方法，但是它并没有满足我们的要求：init 方法应该为 Hand 初始化一个空的 cards 列表，而不是往手牌里添加 52 张新牌。

如果我们提供一个 Hand 的 init 方法，它会覆盖从 Deck 类继承来的同名方法。

```
# Hand
def __init__(self, label=''):
    self.cards = []
    self.label = label
```

当你创建一个 Hand 时，Python 会调用这个 init 方法，而不是 Deck 中的同名方法。

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

其它方法是从 Deck 继承来的，所以我们可以使用 pop_card 和 add_card 来发牌：

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

很自然地，下一步就是把这些代码封装进一个叫 move_cards 的方法：

```
# Deck
def move_cards(self, hand, num):
    for i in range(num):
        hand.add_card(self.pop_card())
```

`move_cards` 接受两个参数，一个是 `Hand` 对象，另一个是发牌的数量。它会同时修改 `self` 和 `hand`，然后返回 `None`。

在有些游戏里面，卡牌从一个手牌移动到另外一个手牌，或者从手牌退到牌堆里面。任何这些操作都可以使用 `move_cards`：`self` 可以是一个 `Deck` 或者一个 `Hand`，而且尽管名字叫 `hand`，它也可以是一个 `Deck`。

继承是一个非常有用的特性。有了继承，一些重复性的代码可以写得非常的优雅。继承有助于代码重用，因为你可以不修改父类定义的前提下，就改变父类的行为。在有些情况下，继承的结构反映了真实问题的结构，使得程序更易于理解。

另一方面，继承又有可能使得程序更加难读。当调用一个方法时，有时候搞不清楚去哪找它的定义。相关的代码可能被分散在几个模块之中。而且，许多用继承能完成的事情，不用继承也可以完成，有可能还完成得更好。

18.8 类图

到目前为止我们已经了解过栈图，它显示的是一个程序的状态；以及对象图，它显示的是一个对象的属性及其值。这些图代表了程序执行中的一个快照，所以它们随着程序的运行而变化。

它们也十分的详细；但有些时候显得过于详细了。类图是程序结构的一种更加抽象的表达。它显示的是类和类之间的关系，而不是每个独立的对象。

类之间有如下几种关系：

- 一个类中的对象可以包含对另外一个类的对象的引用。例如，每一个矩形包含对点的引用，每一个 `Deck` 包含对许多 `Card` 的引用。这种关系被称为组合 (**HAS-A**)，可以类似这样描述：“一个矩形有一个 (has a) 点”。
- 一个类可能继承自另外一个类。这种关系被称为继承 (**IS-A**)，可以类似这样描述：“`Hand` is a kind of `Deck`”。
- 一个类可能强赖另一个类，因为前者中的对象接受后者中的对象作为参数，或者使用后者中的对象作为计算的一部分。这种关系被称为 **依赖**。

类图是这些关系的图形化表示。例如，图 18-2：类图标明了 `Card`，`Deck` 和 `Hand` 之间的关系。

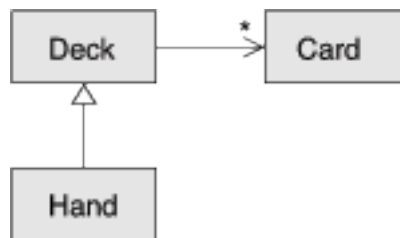


图 18.2: 图 18-2: 类图

带空心三角的箭头表示 **IS-A** 的关系；这里它表示 `Hand` 继承自 `Deck`。

标准箭头表示 **HAS-A** 的关系；这里表示 `Deck` 包含对 `Card` 对象的引用。

箭头旁边的星号是一个复数（**multiplicity**）表达；它表示 Deck 包含多少个 Card。一个复数表达可以是一个简单的数字（如 52），一个范围（如 5..7）或者是 *，表示有任意数量的 Card。

上图中没有标出依赖关系。这种关系通常使用虚线箭头表示。或者，如果有很多依赖关系的话，有时候会省略。

一个更详细的类图可能会显示 Deck 实际包含了一个由 Cards 组成的列表，但是通常类图中不会包含 list 和 dict 等内建类型。

18.9 数据封装

前面几章中描述了一种可以称为“面向对象设计”的开发计划。我们确定所需要的对象——如“Point”、Rectangle 和 Time——然后定义代表它们的类。对于每个类来说，这个类对象和真实世界（或至少是数学世界）中的某种实体具有明显的对应关系。

但是有时有很难界定你需要的对象以及它们如何交互。在这个时候，你需要一个不同的开发计划。之前我们通过封装和泛化来编写函数接口，我们同样可以通过**数据封装**来编写类接口。

马尔科夫分析一节中介绍的马尔科夫分析就是一个很好的例子。如果你从<http://thinkpython2.com/code/markov.py> 下载我的代码，你会发现它使用了两个全局变量——`suffix_map` 和 `prefix`，它们被多个函数进行读写。

```
suffix_map = {}
prefix = ()
```

因为这些变量是全局的，我们一次只能运行一个分析。如果我们读取了两个文本，它们的前缀和后缀会被加入相同的数据结构（会使得输出文本混乱）。

如果想同时运行多个分析，并且保持它们的相互独立，我们可以把每个分析的状态封装到一个对象中。下面是一个示例：

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

下一步，我们把这些函数转换为方法。例如：下面是 `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)
```

像这样改变一个程序——改变设计而保持功能不变——是代码重构的另一个例子（参见 [重构](#) 一节）。

下面的例子给出了一种设计对象和方法的开发计划：

1. 首先编写读取全局变量的函数（如有必要）。
2. 一旦你让程序跑起来了，开始查找全局变量和使用它们的函数的联系。
3. 封装相关的变量作为一个对象的属性。
4. 转换相关函数为新类的方法。

我们做个练习，从 <http://thinkpython2.com/code/markov.py> 下载我的马尔科夫分析代码，然后按照上面所述的步骤，将全局变量封装为新类 `Markov`（注意 `M` 为大写）的属性。

18.10 调试

继承会使得调试变得更加复杂，因为你可能不知道实际调用的是哪个类的方法。

假设你在写一个处理 `Hand` 对象的函数。你可能会想让它可以处理所有种类的 `Hand`，如 `PokerHands`，`BridgeHands`，等等。如果你调用类似 `shuffle` 这样的方法，你可能会得到 `Deck` 中定义的那个，但是如果有任何一个子类覆盖了这个方法。你实际上得到的是子类的那个方法。这个行为通常是一件好事，但是容易让人混淆。

只要你不确定程序的执行流程，最简单的方法是在相关方法的开始处添加 `print` 语句。如果 `Deck.shuffle` 打印一条如像 `Running Deck.shuffle` 的消息，那么随着程序的运行，它会追踪执行的流程。

另外一种方法是使用下面的函数，它接受一个对象和一个方法的名字（字符串格式）作为参数，然后返回提供这个方法定义的类：

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

例如：

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

所以 `Hand` 的 `shuffle` 方法是来自于 `Deck` 的。

`find_defining_class` 使用 `mro` 方法获得将类对象（类型）的列表，解释器将会从这里依次搜索哪个类提供了这个方法。“MOR”是“method resolution order”的简称，指的是 Python “解析”方法名时将搜索的一个类序列。

我提一个对程序设计的建议：当你覆盖一个方法时，新方法的接口应该与旧方法保持一致。它们应该接受相同的参数，返回相同的类型，遵守相同的先决条件和后置条件。如果你遵循这个原则，你会发现：任何你设计的函数，只要能用于一个父类的对象（如 `Deck`），就能够用于任何子类的实例（如 `Hand` 和 `PokerHand`）。

如果你违背这条规则（该原则被称为“里氏代换原理”，英文为：Liskov substitution principle），你的代码逻辑就会变得乱七八糟。

18.11 术语表

编码 (encode):

利用另一组值代表一组值, 方法时构建二者之间的映射。

类属性 (class attribute):

与类对象相关联的属性。类属性定义在类定义的内部, 但在方法的外部。

实例属性 (instance attribute):

与类的实例相关联的属性。

伪装方法 (veneer):

提供另一个函数的不同接口, 但不做太多计算的函数或方法。

继承 (inheritance):

在此前定义的类的基础上进行修改, 从而定义一个新类的能力。

父类 (parent class):

子类所继承自的类。

子类 (child class):

通过继承一个现有类创建的新类。

IS-A 关系:

子类和父类之间的关系。

HAS-A 关系:

两个类之中, 有一个类包含对另一个类的实例的引用的关系。

依赖 (dependency):

两个类之中, 一个类的实例使用了另一个类的实例, 但没有将其保存为属性的关系。

类图 (class diagram):

表明程序中包含的类及其之间的关系的图示。

复数 (multiplicity):

类图中的一种标记, 表明在 HAS-A 关系中, 某个对包含了多少个对另一个类实例的引用。

数据封装 (data encapsulation):

一种程序开发计划, 包括首先编写一个使用全局变量的原型, 然后再讲全局变量变成实例属性的最终版代码。

18.12 练习题

18.12.1 习题 18-1

针对以下程序，画一个 UML 类图，说明其中包含的类及其之间的关系。

```
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)
```

18.12.2 习题 18-2

为 Deck 编写一个叫 `deal_hands` 的方法，接受两个参数：手牌的数量以及每个手牌的卡牌数。它应该创建相应数量的 `Hand` 对象，给每个手牌发放相应数量的卡牌，然后返回一个 `Hands` 列表。

下面是扑克牌中可能的手牌（牌型），越往下值越大，几率越低：

对牌：

两张相同牌面的牌

两对牌：

两对相同牌面的牌

三条：

三张等级相同的牌

顺子：

五张连续的牌（A 可高可低。如 A-2-3-4-5 是一个顺子,10-J-Q-K-A 也是。但是 Q-K-A-2-3 就不是）

同花：

五张花色一样的牌

三代二:

三张等级一样的牌, 另外两张等级一样的牌

四条:

四张牌面一样的牌

同花顺:

五张花色相同的等级连续的牌

18.12.3 习题 18-3

下面这些习题的目的, 是估算抽到不同手牌的几率。

1. 从<http://thinkpython2.com/code> 页面下载以下文件:

Card.py: 本章中完整版本的 `Card`, `Deck` 和 `Hand` 类。

PokerHand.py: 代表 `poker hand` 的不完整的实现, 和一些测试代码。

2. 如果你运行 `PokerHand.py`, 它会发放 7 张牌的 `poker hand`, 检查是否含有顺子。仔细阅读代码, 再继续下面的内容。
3. 往 `PokerHand.py` 文件中添加叫做 `has_pair`、`has_twopair` 等方法, 这些方法根据手牌是否满足相应的标准来返回 `True` 或 `False`。你的代码应该可以正确地处理包含任意卡牌数量 (虽然 5 和 7 是最常见的数量) 的手牌。
4. 写一个叫 `classify` 的方法, 计算出一个手牌的最高值分类, 然后设置对应的 `label` 属性。例如, 一个 7 张牌的手牌可能包含一个顺子和一个对子; 那么它应该标注为 “顺子”。
5. 确信你的分类方法是正确的之后, 下一步是估算这些不同手牌出现的几率。在 `PokerHand.py` 中编写一个函数, 完成洗牌, 分牌, 对牌分类, 然后记录每种分类出现的次数。
6. 打印每种分类和对应频率的表格。运行你的程序, 不断增加手牌的卡牌数量, 直到输出的值保持在足够准确的范围。将你的结果和http://en.wikipedia.org/wiki/Hand_rankings 页面中的值进行比较。

答案: <http://thinkpython2.com/code/PokerHandSoln.py>。

18.12.4 贡献者

1. 翻译: @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第十九章：进阶小技巧

我在写这本书时的一个目标，就是尽量少教些 Python。如果有两种实现方法，我会挑其中之一讲解，避免再提另一种方法。有时候可能会将第二种方法放在练习题里。

现在我想回过头来讲一些之前没有涉及的内容。Python 提供的特性中，有一些其实并不是必须的——没有它们你也能写出好的代码——但是有了它们之后，有时候你能写出更简洁、可读性更高或者效率更高的代码，有时候甚至三个好处都有。

19.1 条件表达式

在[有条件的执行](#)一节中，我们学习了条件语句。条件语句通常用于在两个值之间进行二选一；例如：

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

这个语句检测 x 是否是正值。如果是，它将计算它的 `math.log`。如果不是，`math.log` 会抛出 `ValueError`。为了避免程序出错，我们生成一个“NaN”，这是一个代表“非数字”的特殊浮点值。

我们可以使用 **条件表达式** 简化这个语句：

```
y = math.log(x) if x > 0 else float('nan')
```

这条语句读起来很像英语：“y gets log-x if x is greater than 0; otherwise it gets NaN”（如果 x 大于 0， y 的值则是 x 的 `log`；否则 y 的值为 NaN）。

有时候也可以使用条件表达式改写递归函数。例如，下面是阶乘函数的递归版本：

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

我们可以像这样重写：

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

条件表达式的另一个用处是处理可选参数。例如，下面是习题 17-2 中 GoodKangaroo 类的 init 方法：

```
def __init__(self, name, contents=None):  
    self.name = name  
    if contents == None:  
        contents = []  
    self.pouch_contents = contents
```

我们可以像这样重写：

```
def __init__(self, name, contents=None):  
    self.name = name  
    self.pouch_contents = [] if contents == None else contents
```

一般来说，如果条件语句的两个分支中均为简单的表达式，不是被返回就是被赋值给相同的变量，那么你可以用条件表达式替换该条件语句。

19.2 列表推导式

在映射、筛选和归并一节中，我们学习了映射和筛选模式。例如，下面这个函数接受一个字符串列表，将字符串方法 capitalize 映射至元素，并返回一个新的字符串列表：

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

我们可以使用 **列表推导式** 简化该函数：

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

方括号操作符表示，我们正在构造一个新列表。方括号中的表达式指定列表中的元素，for 子句表示我们要遍历的序列。

列表推导式的语法有点奇怪，因为此例中的循环变量 `s` 在定义之前就出现了。

列表推导式也可以用于筛选。例如，这个函数只选择 `t` 中为大写的元素，并返回一个新列表：

```
def only_upper(t):  
    res = []  
    for s in t:  
        if s.isupper():  
            res.append(s)  
    return res
```

我们可以使用列表推导式重写这个函数：

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

列表推导式非常简洁、易读，至少对简单的表达式是这样的。而且通常比对应的 `for` 循环要更快，有时要快很多。所以，如果你恨我之前没介绍，我可以理解。

但是，我这么做也是有原因的，列表推导式的调试难度更大，因为你不能在循环中添加打印语句。我建议你只在计算足够简单、第一次就能写出正确代码的前提下使用。不过对初学者来说，第一次就写对几乎不可能。

19.3 生成器表达式

生成器表达式与列表推导式类似，但是使用的是圆括号，而不是方括号：

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

结果是一个表达式对象，该对象知道如何遍历一个值序列。但与列举推导式不同的是，它不会一次性计算出所有的值；而是等待求值请求。内建函数 `next` 从生成器获取下一个值：

```
>>> next(g)
0
>>> next(g)
1
```

抵达序列的末尾时，`next` 会抛出 `StopIteration` 异常。你还可以使用 `for` 循环遍历这些值：

```
>>> for val in g:
...     print(val)
4
9
16
```

生成器对象会记录其在序列中的位置，因此 `for` 循环是从 `next` 结束的地方开始的。一旦生成器被消耗完，它会抛出 `StopException`。

```
>>> next(g)
StopIteration
```

生成器表达式常与 `sum`、`max` 和 `min` 等函数一起使用：

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any 和 all

Python 提供了一个内建函数 `any`，它接受一个布尔值序列，如果其中有任何一个值为 `True` 则返回 `True`。它也适用于列表：

```
>>> any([False, False, True])
True
```

但是它通常用于生成器表达式:

```
>>> any(letter == 't' for letter in 'monty')
True
```

上面这个例子不是很有用, 因为它的功能和 `in` 操作符一样。但是我们可以使用 `any` 重写[搜索](#)一节中的部分搜索函数。例如, 我们可以像这样编写 `avoids` 函数:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

上面的函数读取来和英语没什么区别: “word avoids forbidden if there are not any forbidden letters in word.” (如果某个词中没有任何禁用字母, 那么该词就算避免了使用禁用词。)

将 `any` 与生成器表达式结合使用的效率较高, 因为它只要一遇到真值就会终止, 所以不会对整个序列进行计算。

Python 还提供了另一个内建函数 `all`, 如果序列中的每个元素均为 `True` 才会返回 `True`。我们做个练习, 使用 `all` 重写[搜索](#)一节中 `uses_all` 函数。

19.5 集合

在[字典差集](#)一节中, 我使用字典对那些在文档中但不在单词列表里的单词进行了查找。我写的那个函数接受参数 `d1` 和 `d2`, 分别包含文档中的单词 (作为键使用) 和单词列表。它返回不在 `d2` 中但在 `d1` 里的键组成的字典。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

在上面的字典中, 所有键的值均为 `None`, 因为我们不会使用这些值。后果就是会浪费一些存储空间。

Python 提供了另一个叫做集合的内建类型, 它的行为类似没有值的字典键集合。往集合中添加元素是非常快的; 成员关系检测也很快。另外, 集合还提供了计算常见集合操作的方法和操作符。

例如, 集合差集就有一个对应的 `difference` 方法, 或者操作符 `-`。因此, 我们可以这样重写 `subtract` 函数:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

结果是一个集合, 而不是字典, 但对于像迭代这样的操作而言, 二者是没有区别的。

如果使用集合来完成本书中的部分练习题, 代码会比较简洁、高效。例如, 下面是[习题 10-7](#)中 `has_duplicates` 函数的一种使用字典的实现:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

当某个元素首次出现时，它被添加至字典中。如果同样的元素再次出现，函数则返回 True。

如果使用集合，我们可以像这样重写该函数：

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

一个元素在集合中只能出现一次，因此如果 `t` 中的某个元素出现次数超过一次，那么集合的大小就会小于 `t`。如果没有重复的元素，集合和 `t` 的大小则相同。

我们还可以使用集合完成第九章：文字游戏中的部分练习题。例如，下面是使用循环实现的 `uses_only` 函数：

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

`uses_only` 检查 `word` 中的所有字符也在 `available` 中。我们可以像这样重写该函数：

```
def uses_only(word, available):
    return set(word) <= set(available)
```

操作符 `<=` 检查某个集合是否是另一个集合的子集或本身，包括了二者相等的可能性。如果 `word` 中所有的字符都出现在 `available` 中，则返回 True。

接下来做个练习，使用集合重写 `avoids` 函数。

19.6 计数器

计数器（Counter）类似集合，区别在于如果某个元素出现次数超过一次，计数器就会记录其出现次数。如果你熟悉数学中的 **多重集** 概念，计数器就是用来表示一个多重集的自然选择。

计数器定义在叫做 `collections` 的标准模块中，因此你必须首先导入该模块。你可以通过字符串、列表或任何支持迭代的数据结构来初始化计数器：

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

计数器的行为与字典有很多相似的地方：它们将每个键映射至其出现的次数。与字典一样，键必须是可哈希的。

与字典不同的是，如果你访问一个没有出现过的元素，计数器不会抛出异常，而只是返回 0：

```
>>> count['d']
0
```

我们可以使用计数器重写习题 10-6 中的 `is_anagram` 函数：

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

如果两个单词是变位词，那么它们会包含相同的字符，而且字符的计数也相同，因此它们的计数器也是等价的。

计数器提供了执行类似集合操作的方法和操作符，包括集合添加、差集、并集和交集。另外，还提供了一个通常非常有用的方法 `most_common`，返回一个由值-频率对组成的列表，按照频率高低排序：

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict

`collections` 模块中还提供了一个 `defaultdict`，它类似字典，但是如果你访问一个不存在的键，它会临时生成一个新值。

在创建 `defaultdict` 时，你提供一个用于创建新值的函数。这个用于创建对象的函数有时也被称为 **工厂**。用于创建列表、集合和其他类型的内建函数也可以用作工厂：

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

请注意，这里的实参是 `list`，它是一个类对象，而不是 `list()`，后者是一个新列表。你提供的函数只有在访问不存在的键时，才会被调用。

```
>>> t = d['new key']
>>> t
[]
```

新列表 `t` 也被添加至字典中。因此如果我们修改 `t`，改动也会出现在 `d` 中。

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

如果你要创建一个列表组成的字典，通常你可以使用 `defaultdict` 来简化代码。在习题 12-2 的答案（可从 http://thinkpython2.com/code/anagram_sets.py 处获取）中，我创

建的字典将排好序的字符串映射至一个可以由这些字符串构成的单词列表。例如，'opst' 映射至列表 ['opts', 'post', 'pots', 'spot', 'stop', 'tops']。

下面是代码：

```
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
```

这个函数可以使用 `setdefault` 进行简化，你可能在习题 11-2 中也用到了：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

这种方案有一个缺点，即不管是否需要，每次都会创建一个新列表。如果只是创建列表，这问题你不大，但是如果工厂函数非常复杂，就可能会成为一个大问题。

我们可以使用 `defaultdict` 来避免这个问题，同时简化代码：

```
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
```

习题 18-3 的答案（可从 <http://thinkpython2.com/code/PokerHandSoln.py> 下载）中，`has_straightflush` 函数使用了 `setdefault`。这个答案的缺点就是每次循环时都会创建一个 `Hand` 对象，不管是否需要。我们做个练习，使用 `defaultdict` 改写这个函数。

19.8 命名元组

许多简单对象基本上就是相关值的集合。例如，第十五章：类和对象中定义的 `Point` 对象包含两个数字 `x` 和 `y`。当你像下面这样定义类时，你通常先开始定义 `init` 和 `str` 方法：

```
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)
```

但是编写了这么多代码，却只传递了很少的信息。Python 提供了一个更简洁的实现方式：

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

第一个实参是你希望创建的类的名称。第二个实参是 `Point` 对象应该具备的属性列表，以字符串的形式指定。`namedtuple` 的返回值是一个类对象：

```
>>> Point
<class '__main__.Point'>
```

这里的 `Point` 自动提供了像 `__init__` 和 `__str__` 这样的方法，你没有必须再自己编写。

如果想创建一个 `Point` 对象，你可以将 `Point` 类当作函数使用：

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

`init` 方法将实参赋值给你提供的属性。`str` 方法打印 `Point` 对象的字符串呈现及其属性。

你可以通过名称访问命令元组的元素：

```
>>> p.x, p.y
(1, 2)
```

但是你也可以把命名元组当作元组使用：

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

命名元组是定义简单类的一种便捷方式。缺点是这些简单类不会一成不变。之后你可能会发现想要给命名元组添加更多的方法。在这种情况下，你可以定义一个继承自命名元组的新类：

```
class Pointier(Point):
    # add more methods here
```

或者使用传统的类定义方式。

19.9 汇集关键字实参

在可变量数参数元组一节中，我们学习了如何编写一个将实参汇集到元组的函数：

```
def printall(*args):
    print(args)
```

你可以使用任意数量的位置实参（即不带关键字的参数）调用该函数：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

不过 * 星号操作符无法汇集关键字参数：

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

如果要汇集关键字参数，你可以使用 ** 双星号操作符：

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

你可以给关键字汇集形参取任意的名称，但是 `kwargs` 是常用名。上面函数的结果是一个将关键字映射至值的字典：

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

如果你有一个有关键字和值组成的字典，可以使用分散操作符（scatter operator）** 调用函数：

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

如果没有分散操作符，函数会将 `d` 视为一个位置实参，因此会将 `d` 赋值给 `x` 并报错，因为没有给 `y` 赋值：

```
>>> 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'
```

在处理有大量形参的函数时，通常可以创建指定了常用选项的字典，并将其传入函数。

19.10 术语表

条件表达式（conditional expression）：

根据条件在两个值中二选一的表达式。

列表推导式（list comprehension）：

位于方括号中带 `for` 循环的表达式，最终生成一个新列表。

生成器表达式（generator expression）：

位于圆括号中带 for 循环的表达式，最终生成一个生成器对象。

多重集 (multiset):

一个数学概念，表示一个集合的元素与各元素出现次数之间的映射。

工厂 (factory):

用于创建对象的函数，通常作为形参传入。

19.11 练习题

19.11.1 习题 19-1

下面是一个递归计算二项式系数 (binomial coefficient) 的函数。

```
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
```

使用嵌套条件表达式重写函数体。

注意：这个函数不是特别高效，因为它最后在不断地重复计算相同的值。你可以通过备忘录模式 (memoizing, 也可理解为缓存) 来提高效率 (参见备忘录一节)。不过你会发现，如果使用条件表达式，进行缓存的难度会更大。

19.11.2 贡献者

1. 翻译: @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第二十章：调试

在调试时，你应该区别不同类别的错误，才能更快地追踪定位：

- 语法错误是 Python 将源代码翻译成字节代码的时候产生的，说明程序的结构有一些错误。例如：省略了 `def` 语句后面的冒号会产生看上去有点重复的错误信息 `SyntaxError: invalid syntax`。
- 运行时错误是当程序在运行时出错，解释器所产生的错误。大多数运行时错误会包含诸如错误在哪里产生和正在执行哪个函数等信息。例如：一个无限递归最终会造成 `maximum recursion depth exceeded`（“超过递归最大深度”）的运行时错误。
- 语义错误是指一个程序并没有抛出错误信息，但是没有做正确的事情。例如：一个表达式可能因为没有按照你预期的顺序执行，因此产生了错误的结果。

调试的第一步是弄清楚你正在处理哪种错误。虽然下面的各节是按照错误类型来组织的，有些技巧实际上适用于多种情形。

20.1 语法错误

通常一旦找出是哪种语法错误，就容易修正。不幸的是，抛出的错误消息通常没什么帮助。最常见的错误消息是 `SyntaxError: invalid syntax` 和 `SyntaxError: invalid token`，都没有提供很多信息。

另一方面，这些错误消息会告诉你程序的哪里出现了错误。实际上，它告诉你 Python 是在哪里发现的问题，但这并不一定就是出错的地方。有时，错误出现在错误消息出现的位置之前，通常就在前一行。

如果你是一点一点地增量式地写的代码，你应该能够知道错误在哪里。一般就在我最后添加的那行代码里。

如果你是从书上复制的代码，那请仔细地从头和书中的代码对照。一个一个字母地比照。同时，记住也可能是书上就错了，所以如果你发现看上去像语法错误的地方，那可能就是了。

下面是避免大部分常见语法错误的一些方法：

1. 确保你没有使用 Python 的关键字作为变量名称。
2. 检查你在每个复合语句首行的末尾都加了冒号，包括 `for`，`while`，`if`，和 `def` 语句。

3. 确保代码中的字符串都有匹配地引号。确保所有的引号都是“直引号”，而不是“花引号”。
4. 如果你有带三重引号的多行字符串，确保你正确地结束了字符串。一个没有结束的字符串会在程序的末尾产生 `invalid token` 错误，或者它会把剩下的程序看作字符串的一部分，直到遇到下一个字符串。第二种情况下，可能根本不会产生错误！
5. 一个没有关闭的操作符（`(`，`{` 以及 `[`）使得 Python 把下一行继续看作当前语句的一部分。通常下一行会马上提示错误消息。
6. 检查条件语句里面的 `==` 是不是写成了 `=`。
7. 确保每行的缩进是符合要求。Python 能够处理空格和制表符，但是如果混用则会出错。避免该问题的最好方法是使用一个了解 Python 语法、能够产生一致缩进的纯文本编辑器。
8. 如果代码中包含有非 ASCII 字符串（包括字符串和注释），可能会出错，尽管 Python 3 一般能处理非 ASCII 字符串。从网页或其他源粘贴文本时，要特别注意。

如果上面的方法都不想，请接着看下一节...

20.1.1 我不断地改代码，但似乎一点用都没有。

如果解释器说有一个错误但是你怎么也看不出来，可能是因为你和解释器看的不是同一个代码。检查你的编码环境，确保你正在编辑的就是 Python 试图要运行的程序。

如果你不确定，试着在程序开始时制造一些明显、故意的语法错误。再运行一次。如果解释器没有提示新错误，说明你没有运行新修改的代码。

有可能是以下原因：

- 你编辑了文件，但是忘记了在运行之前保存。有一些编程环境会在运行前自动保存，有些则不会。
- 你更改了文件的名称，但是你仍然在运行旧名称的文件。
- 开发环境的配置不正确。
- 如果你在编写一个模块，使用了 `import` 语句，确保你没有使用标准 Python 模块的名称作为模块名。
- 如果你使用 `import` 来载入一个模块，记住你必须重启解释器或者使用 `reload` 才能重新载入一个修改了的文件。如果你导入一个模块两次，第二次是无效的。

如果你依然解决不了问题，不知道究竟是怎么回事，有一种办法是从一个类似“Hello, World!”这样的程序重头开始，确保你能运行一个已知的程序。然后逐渐地把原来程序的代码粘贴到新的程序中。

20.2 运行时错误

一旦你的程序语法正确，Python 就能够编译它，至少可以正常运行它。接下来，可能会出现哪些错误？

20.2.1 我的程序什么也没有做。

在文件由函数和类组成，但并没有实际调用函数执行时，这个问题是最常见的。你也可能是故意这么做的，因为你只打算导入该模块，用于提供类和函数。

如果你不是故意的，确保你调用了函数来开始执行，请确保执行流能够走到函数调用处（参见下面“执行流”一节）。

20.2.2 我的程序挂死了。

如果一个程序停止了，看起来什么也没有做，这就是“挂死”了。通常这意味着它陷入了无限循环或者是无限递归。

- 如果你怀疑问题出在某个循环，在该循环之前添加一个打印语句，输出“进入循环”，在循环之后添加一个打印“退出循环”的语句。

运行程序。如果打印了第一条，但没有打印第二条，那就是进入了无线循环。跳到下面“无限循环”一节。

- 大多数情况下，无限递归会造成程序运行一会儿之后输出“RuntimeError: Maximum recursion depth exceeded”错误。如果发生了这个错误，跳到下面“无限递归”一节。

如果没有出现这个错误，但你怀疑某个递归方法或函数有问题，你仍可以使用“无线递归”一节中的技巧。

- 如果上面两种方法都没用，开始测试其他的循环和递归函数或方法是否存在问题。
- 如果这也没有用，那有可能你没有看懂程序的执行流。跳到下面“执行流”一节。

无限循环

如果你认为程序中有一个无限循环，并且知道是哪一个循环，在循环的最后添加一个打印语句，打印条件中各个变量的值，以及该条件的值。

例如：

```
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))
```

现在，当你运行程序时，你可以看到每次循环都有 3 行输出。最后一次循环时，循环条件应该是 `False`。如果循环继续走下去，你能够看到 `x` 和 `y` 的值，这时你或许能弄清楚到为什么它们的值没有被正确地更新。

无限递归

大多数情况，无限递归会造成程序运行一会儿之后输出“RuntimeError: Maximum recursion depth exceeded”错误。

如果你怀疑一个函数造成了无限递归, 确保函数有一个基础情形。也就是存在某种条件能够让函数直接返回值, 而不会再递归调用下去。如果没有, 你需要重新思考算法, 找到一个初始条件。

如果有了基础情形了但是程序还是没有到达它, 在函数的开头加入一个打印语句来打印参数。现在当你运行程序时, 每次递归调用你都能看到几行输出, 你可以看到参数的值。如果参数没有趋于基础情形, 你会大致明白其背后的原因。

执行流

如果你不确定程序执行的过程, 在每个函数的开始处添加打印语句, 打印类似“进入函数 `foo`”这样的信息, `foo` 是你的函数名。

现在运行程序时, 就会打印出每个函数调用的轨迹。

20.2.3 运行程序时产生了异常。

如果在运行时出现了问题, Python 会打印出一些信息, 包括异常的名称、产生异常的行号和一个回溯 (traceback)。

回溯会指出正在运行的函数、调用它的上层函数以及上上层函数等等。换言之, 它追踪进行到目前函数调用所调用过的函数, 包括每次函数的调用所在的行号。

第一步是检查程序中发生错误的位置, 看你能不能找出问题所在。下面是一些常见的运行时错误:

命名错误 (NameError):

你正在使用当前环境中不存在的变量名。检查下名称是否拼写正确, 或者名称前后是否一致。还要记住局部变量是局部的。你不能在定义它们的函数的外面引用它们。

类型错误 (TypeError):

有几种可能的原因:

- 值的使用方法不对。例如: 使用除整数以外的东西作为字符串、列表或元组的索引下标。
- 格式化字符串中的项与传入用于转换的项之间不匹配。如果项的数量不同或是调用了无效的转换, 都会出现这个问题。
- 传递给函数的参数数量不对。如果是方法, 查看方法定义是不是以 `self` 作为第一个参数。然后检查方法调用; 确保你在一个正确的类型的对象上调用方法, 并且正确地提供了其它参数。

键错误 (KeyError):

你尝试用字典没有的键来访问字典的元素。如果键是字符串, 记住它是区分大小写的。

属性错误 (AttributeError):

你尝试访问一个不存在的属性或方法。检查一下拼写！你可以使用内建函数 `dir` 来列出存在的属性。

如果一个属性错误表明一个对象是 `NoneType`，那意味着它就是 `None`。因此问题不在于属性名，而在于对象本身。

对象是 `None` 的一个可能原因，是你忘记从函数返回一个值；如果程序执行到函数的末尾没有碰到 `return` 语句，它就会返回 `None`。另一个常见的原因是使用了列表方法的结果，如 `sort`，这种方法返回的是 `None`。

索引错误 (`IndexError`):

用来访问列表、字符串或元组的索引要大于访问对象长度减一。在错误之处的前面加上一个打印语句，打印出索引的值和数组的长度。数组的大小是否正确？索引值是否正确？

Python 调试器 (`pdb`) 有助于追踪异常，因为它可以让你检查程序出现错误之前的状态。你可以阅读 <https://docs.python.org/3/library/pdb.html> 了解更多关于 `pdb` 的细节。

20.2.4 我加入了太多的打印语句以至于输出刷屏。

使用打印语句来调试的一个问题，是你可能会被泛滥的输出所淹没。有两种途径来处理：简化输出或者是简化程序。

为了简化输出，你可以移除或注释掉不再需要的打印语句，或者合并它们，或者格式化输出便于理解。

为了简化程序，有几件事情可以做的。首先，缩减当前求解问题的规模。例如，如果你在检索一个列表，使用一个小列表来检索。如果程序从用户获得输入，给一个会造成问题的最简单的输入。

其次，清理程序。移除死代码，并且重新组织程序使其易于理解。例如，如果你怀疑问题来自程序深度嵌套的部分，尝试使用简单的结构重写它。如果你怀疑是一个大函数的问题，尝试分解它为小函数并分别测试。

通常，寻找最小化测试用例的过程能够引出 `bug`。如果你发现一个程序在一种条件下运行正确，在另外的条件下运行不正确，这能够给你提供一些解决问题的线索。

类似的，重写代码能够让你发现难找的 `bug`。如果你做了一处改变，认为不会影响程序但是却事实证明相反，这也可以给你线索。

20.3 语义错误

在某些程度上，语义错误是最难调试的，因为解释器不能提供错误的信息。只有你知道程序本来应该是怎么样做的。

第一步是在程序代码和你看到的表现之间建立连接。你需要首先假设程序实际上干了什么事情。这种调试的难处之一，是电脑运行的太快了。

你会经常希望程序能够慢下来好让你能跟上它的速度，通过一些调试器 (`debugger`) 就能做到这点。但是有时候，插入一些安排好位置的打印语句所需的时间，要比你设置好调试器、插入和移除断点，然后“步进”程序到发生错误的地方要短。

20.3.1 我的程序不能工作。

你应该问自己下面这些问题：

- 是不是有我希望程序完成的但是并没有出现的东西？找到执行这个功能的代码，确保它是按照你认为的方式工作的。
- 是不是有些本不该执行的代码却运行了？找到程序中执行这个功能的代码，然后看看它是不是本不应该执行却执行了。
- 是不是有一些代码的效果和你预期的不一样？确保你理解了那部分的代码，特别是当它涉及调用其它模块的函数或者方法。阅读你调用的函数的文档。尝试写一些简单的测试用例，来测试他们是不是得到了正确的结果。

在编程之前，你需要先建立程序是怎样工作的思维模型。如果你写出来的代码并非按照你预期的工作，问题经常不是在程序本身，而是你的思维模型。

纠正思维模型最好的方法，是把程序切分成组件（就是通常的函数和方法），然后单独测试每个组件。一旦你找到了模型和现实的不符之处，你就能解决问题了。

当然，你应该在写代码的过程中就编写和测试组件。如果你遇到了一个问题，那只能是刚写的一小段代码才有可能出问题。

20.3.2 我写了一个超大的密密麻麻的表达式，结果它运行得不正确。

写复杂的表达式是没有问题的，前提是可读，但是它们很难调试。通常把复杂的表达式打散成一系列临时变量的赋值语句，是一个好做法。

例如：

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

这可以重写成：

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

显示的版本更容易读，因为变量名提供了额外的信息，也更容易调试，因为你可以检查中间变量的类型和值。

超长表达式的另外一个问题是，计算顺序可能和你想得不一樣。例如如果你把 $\frac{x}{2\pi}$ 翻译成 Python 代码，你可能会写成：

```
y = x / 2 * math.pi
```

这就不正确了，因为乘法和除法具有相同的优先级，所以它们从左到右进行计算。所以表达式计算的是 $x\pi/2$ 。

调试表达式的一个好办法，是添加括号来显式地指定计算顺序：

```
y = x / (2 * math.pi)
```

只要你不太确定计算的顺序，就用括号。这样不仅能确保程序正确（按照你认为的方式工作），而且对于那些记不住优先级的人来说更加易读。

20.3.3 有一个函数没有返回我期望的结果。

如果你的 `return` 语句是一个复杂的表达式，你没有机会在返回之前打印出计算的结果。不过，你可以用一个临时变量。例如，与其这样写：

```
return self.hands[i].removeMatches()
```

不如写成：

```
count = self.hands[i].removeMatches()
return count
```

现在，你就有机会在返回之前显示 `count` 的值了。

20.3.4 我真的是没办法了，我需要帮助。

首先，离开电脑几分钟吧。电脑发出的辐射会影响大脑，容易造成以下症状：

- 焦躁易怒
- 迷信（“电脑就是和我作对”）和幻想（“只有我反着带帽子程序才会正常工作”）。
- 随机漫步编程（试图编写所有可能的程序，选择做了正确的事情的那个程序）。

如果你发现你自己出现上述的症状，起身走动走动。当你冷静之后，再想想程序。它在做什么？它异常表现的一些可能的原因是什么？上次代码正确运行时什么时候，你接下来做了什么？

有时，找到一个 `bug` 就是需要花很长的时间。我经常都是在远离电脑、让我的思绪飞扬时才找到 `bug` 的。一些寻找 `bug` 的绝佳地点是火车上、洗澡时、入睡之前在床上。

20.3.5 我不干了，我真的需要帮助。

这个经常发生。就算是最好的程序员也偶尔被难住。有时你在一个程序上工作的时间太长了，以至于你看不到错误。那你该是休息一下双眼了。

当你拉某人来帮忙之前，确保你已经准备好了。你的程序应该尽量简单，你应该应对造成错误的最小输入。你应该在合适的地方添加打印语句（打印输出应该容易理解）。你应该对程序足够理解，能够简洁地对其进行描述。

当你拉某人来帮忙时，确保提供他们需要的信息：

- 如果有错误信息，它是什么以及它指出程序的错误在哪里？
- 在这个错误发生之前你最后做的事情是什么？你写的最后一行代码是什么，或者失败的新的测试样例是怎样的？
- 你至今都尝试了哪些方法，你了解到了什么？

你找到了 `bug` 之后，想想你要怎样才能更快的找到它。下次你看到相似的情况时，你就可以更快的找到 `bug` 了。

记住，最终目标不是让程序工作，而是学习如何让程序正确工作。

贡献者

1. 翻译: @bingjin
2. 校对: @bingjin
3. 参考: @carfly

第二十一章：算法分析

本附录摘自 Allen B. Downey 的 *Think Complexity* 一书，也由 O'Reilly Media (2011) 出版。当你读完本书后，也许你可以接着读读那本书。

算法分析 (**Analysis of algorithms**) 是计算机科学的一个分支，着重研究算法的性能，特别是它们的运行时间和资源开销。见 http://en.wikipedia.org/wiki/Analysis_of_algorithms。

算法分析的实际目的是预测不同算法的性能，用于指导设计决策。

2008 年美国总统大选期间，当候选人奥巴马 (Barack Obama) 访问 Google 时，他被要求进行即时分析。首席执行官 Eric Schmidt 开玩笑地问他“对一百万个 32 位整数排序的最有效的方法”。显然有人暗中通知了奥巴马，因为他很快回答，“我认为不应该采用冒泡排序法”。详见 http://www.youtube.com/watch?v=k4RRi_ntQc8。

是真的：冒泡排序概念上很简单，但是对于大数据集来说速度非常慢。Schmidt 所提问题的答案可能是“基数排序 (http://en.wikipedia.org/wiki/Radix_sort)”¹。

算法分析的目的是在不同算法间进行有意义的比较，但是有一些问题：

- 算法的相对性能依赖于硬件的特性，因此一个算法可能在机器 A 上比较快，另一个算法则在机器 B 上比较快。对此问题一般的解决办法是指定一个 机器模型 (machine model) 并且分析一个算法在一个给定模型下所需的步骤或运算的数目。
- 相对性能可能依赖于数据集的细节。例如，如果数据已经部分排好序，一些排序算法可能更快；此时其它算法运行的比较慢。避免该问题的一般方法是分析 最坏情况。有时分析平均情况性能也可，但那通常更难，而且可能不容易弄清该对哪些数据集进行平均。
- 相对性能也依赖于问题的规模。一个对于小列表很快的排序算法可能对于长列表很慢。此问题通常的解决方法是将运行时间（或者运算的次数）表示成问题规模的函数，并且根据各自随着问题规模的增长而增加的速度，将函数分成不同的类别。

此类比较的好处是有助于对算法进行简单的分类。例如，如果我知道算法 A 的运行时间与输入的规模 n 成正比，算法 B 与 n^2 成正比，那么我可以认为 A 比 B 快，至少对于很大的 n 值来说。

这类分析也有一些问题，我们后面会提到。

¹ 但是，如果你面试中被问到这个问题，我认为更好的答案是，“对上百万个整数进行最快排序的方法就是用你所使用的语言的内置排序函数。它的性能对于大多数应用而言已优化的足够好。但如果最终我的应用运行太慢，我会用性能分析器找出大量的运算时间被用在了哪儿。如果采用一个更快的算法会对性能产生显著的提升，我会试着找一个基数排序的优质实现。”

21.1 增长量级

假设你已经分析了两个算法，并能用输入计算量的规模表示它们的运行时间：若算法 A 用 $100n + 1$ 步解决一个规模为 n 的问题；而算法 B 用 $n^2 + n + 1$ 步。

下表列出了这些算法对于不同问题规模的运行时间：

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

当 $n = 10$ 时，算法 A 看上去很糟糕，它用了 10 倍于算法 B 所需的时间。但当 $n = 100$ 时，它们性能几乎相同，而 n 取更大值时，算法 A 要好得多。

根本原因是对于较大的 n 值，任何包含 n^2 项的函数都比首项为 n 的函数增长要快。首项 (leading term) 是指具有最高指数的项。

对于算法 A，首项有一个较大的系数 100，这是为什么对于小 n ，B 比 A 好。但是不考虑该系数，总有一些 n 值使得 $an^2 > bn$ ， a 和 b 可取任意值。

同样推论也适用于非首项。即使算法 A 的运行时间为 $n + 1000000$ ，对于足够大的 n ，它仍然比算法 B 好。

一般来讲，我们认为具备较小首项的算法对于规模大的问题是一个好算法，但是对于规模小的问题，可能存在有一个交叉点 (crossover point)，在此规模以下，另一个算法更好。交叉点的位置取决于算法的细节、输入以及硬件，因此在进行算法分析时它通常被忽略。但是这并不意味着你可以忘记它。

如果两个算法有相同的首项，很难说哪个更好；答案还是取决于细节。所以对于算法分析来说，具有相同首项的函数被认为是相当的，即使它们具有不同的系数。

增长量级 (order of growth) 是一个函数集合，集合中函数的增长行为被认为是相当的。例如 $2n$ 、 $100n$ 和 $n + 1$ 属于相同的增长量级，可用大 O 符号 (Big-Oh notation) 写成 $O(n)$ ，而且常被称作线性级 (linear)，因为集合中的每个函数随着 n 线性增长。

首项为 n^2 的函数属于 $O(n^2)$ ；它们被称为二次方级 (quadratic)。

下表列出了算法分析中最通常的一些增长量级，按照运行效率从高到低排列²。

Order of Growth	Name	
$O(1)$	constant	
$O(\log_b n)$	logarithmic	
$O(n)$	linear	
$O(n \log_b n)$	linearithmic	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(c^n)$	exponential	

对于对数级，对数的基数并不影响增长量级。改变基数等价于乘以一个常数，其不改变增长量级。相应的，所有的指数级数都属于相同的增长量级，而无需考虑指数的基数大小。指数函数增长量级增长的非常快，因此指数级算法只用于小规模问题。

² constant: 常数级; logarithmic: 对数级; linear: 线性级; linearithmic: 线性对数级; quadratic: 二次方级; cubic: 三次方级; exponential: 指数级

21.1.1 习题 21-1

访问 http://en.wikipedia.org/wiki/Big_O_notation，阅读维基百科关于大 O 符号的介绍，并回答以下问题：

1. $n^3 + n^2$ 的增长量级是多少？ $1000000n^3 + n^2$ 和 $n^3 + 1000000n^2$ 的增长量级又是多少？
2. $(n^2 + n) \cdot (n + 1)$ 的增长量级是多少？在开始计算之前，记住你只需要考虑首项即可。
3. 如果 f 的增长量级为 $O(g)$ ，那么对于未指定的函数 g ，我们可以如何描述 $af + b$ ？
4. 如果 f_1 和 f_2 的增长量级为 $O(g)$ ，那么 $f_1 + f_2$ 的增长量级又是多少？
5. 如果 f_1 的增长量级为 $O(g)$ ， f_2 的增长量级为 $O(h)$ ，那么 $f_1 + f_2$ 的增长量级是多少？
6. 如果 f_1 的增长量级为 $O(g)$ ， f_2 的增长量级为 $O(h)$ ，那么 $f_1 \cdot f_2$ 的增长量级是多少？

关注性能的程序员经常发现这种分析很难忍受。他们的观点有一定道理：有时系数和非首项会产生巨大的影响。有时，硬件的细节、编程语言以及输入的特性会造成很大的影响。对于小问题，渐近的行为没有什么影响。

但是，如果你牢记这些注意事项，算法分析就是一个有用的工具。至少对于大问题，“更好的”算法通常更好，并且有时要好的多。相同增长量级的两个算法之间的不同通常是一个常数因子，但是一个好算法和一个坏算法之间的不同是无限的！

21.2 Python 基本运算操作分析

在 Python 中，大部分算术运算的开销是常数级的；乘法会比加减法用更长的时间，除法更长，但是这些运算时间不依赖被运算数的数量级。非常大的整数却是个例外；在这种情况下，运行时间随着位数的增加而增加。

索引操作——在序列或字典中读写元素——的增长量级也是常数级的，和数据结构的大小无关。

一个遍历序列或字典的 `for` 循环通常是线性的，只要循环体内的运算是常数时间。例如，累加一个列表的元素是线性的：

```
total = 0
for x in t:
    total += x
```

内建函数 `sum` 也是线性的，因为它做的是相同的事情，但是它要更快一些，因为它是一个更有效的实现；从算法分析角度讲，它具有更小的首项系数。

根据经验，如果循环体内的增长量级是 $O(n^a)$ ，则整个循环的增长量级是 $O(n^{a+1})$ 。如果这个循环在执行一定数目循环后退出则是例外。无论 n 取值多少，如果循环仅执行 k 次，整个循环的增长量级是 $O(n^a)$ ，即便 k 值比较大。

乘上 k 并不会改变增长量级，除法也是。因此，如果循环体的增长量级是 $O(n^a)$ ，而且循环执行 n/k 次，那么整个循环的增长量级就是 $O(n^{a+1})$ ，即使 k 值很大。

大部分字符串和元组运算是线性的，除了索引和 `len`，它们是常数时间。内建函数 `min` 和 `max` 是线性的。切片运算与输出的长度成正比，但是和输入的大小无关。

字符串拼接是线性的；它的运算时间取决于运算对象的总长度。

所有字符串方法都是线性的，但是如果字符串的长度受限一个常数—例如，在单个字符上的运算—它们被认为是常数时间。字符串方法 `join` 也是线性的；它的运算时间取决于字符串的总长度。

大部分的列表方法是线性的，但是有一些例外：

- 平均来讲，在列表结尾增加一个元素是常数时间。当它超出了所占用空间时，它偶尔被拷贝到一个更大的地方，但是对于 n 个运算的整体时间仍为 $O(n)$ ，所以我每个运算的平均时间是 $O(1)$ 。
- 从一个列表结尾删除一个元素是常数时间。
- 排序是 $O(n \log n)$ 。

大部分的字典运算和方法是常数时间，但有些例外：

- `update` 的运行时间与作为形参被传递的字典（不是被更新的字典）的大小成正比。
- `keys`、`values` 和 `items` 是常数时间，因为它们返回迭代器。但是如果你对迭代器进行循环，循环将是线性的。

字典的性能是计算机科学的一个小奇迹之一。在 [哈希表](#) 一节中，我们将介绍它们是如何工作的。

21.2.1 习题 21-2

访问 http://en.wikipedia.org/wiki/Sorting_algorithm，阅读维基百科上对排序算法的介绍，并回答下面的问题：

1. 什么是“比较排序”？比较排序在最差情况下的最好增长量级是多少？别的排序算法在最差情况下的最优增长量级又是多少？
2. 冒泡排序法的增长量级是多少？为什么奥巴马认为是“不应采用的方法”
3. 基数排序 (radix sort) 的增长量级是多少？我们使用它之前需要具备的前提条件有哪些？
4. 排序算法的稳定性是指什么？为什么它在实际操作中很重要？
5. 最差的排序算法是哪一个（有名称的）？
6. C 语言使用哪种排序算法？Python 使用哪种排序算法？这些算法稳定吗？你可能需要谷歌一下，才能找到这些答案。
7. 大多数非比较算法是线性的，因此为什么 Python 使用一个增长量级为 $O(n \log n)$ 的比较排序？

21.3 搜索算法分析

搜索 (search) 算法，接受一个集合以及一个目标项，并判断该目标项是否在集合中，通常返回目标的索引值。

最简单的搜索算法是“线性搜索”，其按顺序遍历集合中的项，如果找到目标则停止。最坏的情况下，它不得不遍历全部集合，所以运行时间是线性的。

序列的 `in` 操作符使用线性搜索；字符串方法 `find` 和 `count` 也使用线性搜索。

如果元素在序列中是排序好的，你可以用 二分搜索 (bisection search)，它的增长量级是 $O(\log n)$ 。二分搜索和你在字典中查找一个单词的算法类似（这里是指真正的字典，不是数据结构）。你不会从头开始并按顺序检查每个项，而是从中间的项开始并检查你要查找的单词在前面还是后面。如果它出现在前面，那么你搜索序列的前半部分。否则你搜索后一半。如论如何，你将剩余的项数分为一半。

如果序列有 1,000,000 项，它将花 20 步找到该单词或判断出其不在序列中。因此它比线性搜索快大概 50,000 倍。

二分搜索比线性搜索快很多，但是它要求已排序的序列，因此使用时需要做额外的工作。

另一个检索速度更快的数据结构被称为 哈希表 (hashtable) —它可以在常数时间内检索出结果—并且不依赖于序列是否已排序。Python 中的字典就通过哈希表技术实现的，因此大多数的字典操作，包括 `in` 操作符，只花费常数时间就可完成。

21.4 哈希表

为了解释哈希表是如何工作以及为什么它的性能如此优秀，我们从实现一个简单的映射 (map) 开始并逐步改进它，直到其成为一个哈希表。

我们使用 Python 来演示这些实现，但在现实生活中，你用不着用 Python 写这样的代码；你只需用内建的字典对象就可以了！因此在接下来的内容中，你就当字典对象并不存在，你希望自己实现一个将键映射到值的数据结构。你必须实现的操作包括：

`add(k, v):`

增加一个新的项，其从键 `k` 映射到值 `v`。如果使用 Python 的字典 `d`，该运算被写作 `d[k] = v`。

`get(k):`

查找并返回相应键的值。如果使用 Python 的字典 `d`，该运算被写作 `d[k]` 或 `d.get(k)`。

现在，假设每个键只出现一次。该接口最简单的实现是使用一个元组列表，其中每个元组是一个键-值对。

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` 向项列表追加一个键-值元组，其增长量级为常数时间。

`get` 使用 `for` 循环搜索该列表：如果它找到目标键，则返回相应的值；否则触发一个 `KeyError`。因此 `get` 是线性的。

另一个方案是保持列表按键排序。那么，`get` 可以使用二分搜索，其增长量级为 $O(\log n)$ 。但是在列表中间插入一个新的项是线性的，因此这可能不是最好的选择。有其它的数据结构能在对数级时间内实现 `add` 和 `get`，但是这仍然不如常数时间好，那么我们继续。

另一种改良 `LinearMap` 的方法是将键-值对列表分成小列表。下面是一个被称作 `BetterMap` 的实现，它是 100 个 `LinearMap` 组成的列表。正如一会儿我们将看到的，`get` 的增长量级仍然是线性的，但是 `BetterMap` 是迈向哈希表的一步。

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` 会生成一个由 `n` 个 `LinearMap` 组成的列表。

`add` 和 `get` 使用 `find_map` 查找往哪一个列表中添加新项，或者对哪个列表进行检索。

`find_map` 使用了内建函数 `hash`，其接受几乎任何 Python 对象并返回一个整数。这一实现的一个限制是它仅适用于可哈希的键。像列表和字典等可变类型是不能哈希的。

被认为是相等的可哈希对象返回相同的哈希值，但是反之不是必然成立：两个具备不同值的对象能够返回相同的哈希值。

`find_map` 使用求余运算符将哈希值包在 0 到 `len(self.maps)` 之间，因此结果是该列表的合法索引值。当然，这意味着许多不同的哈希值将被包成相同的索引值。但是如果哈希函数散布相当均匀（这是哈希函数被设计的初衷），那么我们预计每个 `LinearMap` 会有 $n/100$ 项。

由于 `LinearMap.get` 的运行时间与项数成正比，那么我们预计 `BetterMap` 比 `LinearMap` 快 100 倍。增长量级仍然是线性的，但是首项系数变小了。这样很好，但是仍然不如哈希表好。

下面是使哈希表变快的关键：如果你能保证 `LinearMap` 的最大长度是有上限的，则 `LinearMap.get` 的增长量级是常数时间。你只需要跟踪项数并且当每个 `LinearMap` 的项数超过阈值时，通过增加更多的 `LinearMap` 调整哈希表的大小。

以下是哈希表的一个实现：

```

class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items():
                new_maps.add(k, v)

        self.maps = new_maps

```

每个 HashMap 包含一个 BetterMap。__init__ 开始仅有两个 LinearMap，并且初始化 num，用于跟踪项的数量。

get 仅仅用来调度 BetterMap。真正的操作发生于 add 内，其检查项的数量以及 BetterMap 的大小：如果它们相同，每个 LinearMap 的平均项数为 1，因此它调用 resize。

resize 生成一个新的 BetterMap，是之前那个的两倍大，然后将像从旧表“重新哈希”至到新的表。

重新哈希是必要的，因为改变 LinearMap 的数目也改变了 find_map 中求余运算的分母。这意味着一些被包进相同的 LinearMap 的对象将被分离（这正是我们希望的，对吧？）。

重新哈希是线性的，因此 resize 是线性的，这可能看起来很糟糕，因为我保证 add 会是常数时间。但是记住，我们不必每次都调整，因此 add 通常是常数时间，只是偶尔是线性的。运行 add n 次的整体操作量与 n 成正比，因此 add 的平均运行时间是常数时间！

为了弄清这是如何工作的，考虑以一个空的 HashTable 开始并增加一系列项。我们以两个 LinearMap 开始，因此前两个 add 操作很快（不需要调整大小）。我们假设它们每个操作花费一个工作单元。下一个 add 需要进行一次大小调整，因此我们必须重新哈希前两项（我们将其算成两个额外的工作单元），然后增加第 3 项（又一个工作单元）。增加下一项的花费一个单元，所以目前为止添加四个项共需要 6 个单元。

下一个 add 花费 5 个单元，但是之后的 3 个操作每个只花费 1 个单元，所以前八个 add 总共需要 14 个单元。

下一个 add 花费 9 个单元，但是之后在下次调整大小之前，可以再增加七个，所以前 16 个 add 总共需要 30 个单元。

进行 32 次 add 之后，总共花费了 62 个单元，我希望你开始看到规律。 n 次 add 后，其中 n 是 2 的倍数，总花费是 $2n - 2$ 个单元，所以平均每个 add 操作只花费了少于 2 个单元。

当 n 是 2 的倍数时，那是最好的情况。对于其它的 n 值，平均花费稍高一点，但是那并不重要。重要的是其增长量级为 $O(1)$ 。

图 21-1：哈希表中 `add` 操作的成本形象地说明了其工作原理。每个区块代表一个工作单元。每列显示每个 `add` 所需的单元，按从左到右的顺序排列：前两个 `add` 花费 1 个单元，第三个花费 3 个单元，等等。

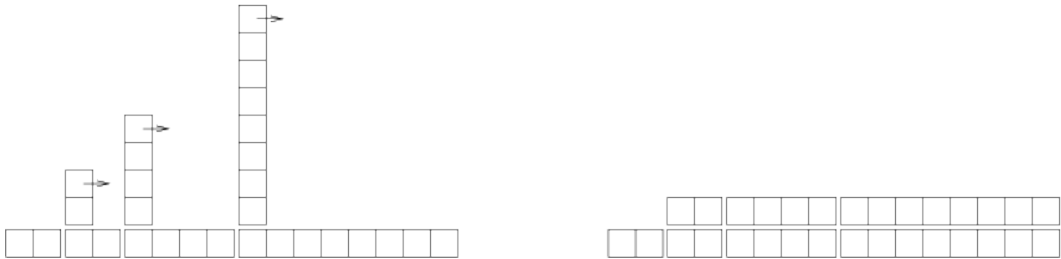


图 21.1: 图 21-1: 哈希表中 `add` 操作的成本

重新哈希的额外工作，表现为一系列不断增高的高塔，各自之间的距离越来越大。现在，如果你打翻这些塔，将大小调整的代价均摊到所有的 `add` 上，你会从图上看到 n 次 `add` 的整个花费是 $2n - 2$ 。

该算法一个重要的特征是，当我们调整 `HashTable` 的大小时，它呈几何级增长；也就是说，我们用常数乘以表的大小。如果你按算术级增加大小——每次增加固定的数目——每个 `add` 的平均时间是线性的。

你可以从 <http://thinkpython2.com/code/Map.py> 下载到 `HashMap` 的实现代码，你不必使用它；如果你想要一个映射数据结构，只要使用 Python 中的字典即可。

21.5 术语表

算法分析 (algorithm analysis):

比较不同算法间运行时间和资源占用的分析方法。

机器模型 (machine model):

用于描述算法（性能）的简化计算机表示。

最坏情况 (worst case):

使得给定算法运行时间最长（或占用做多资源）的输入。

首项 (leading term): 在多项式中，拥有指数最高的项。

交叉点 (crossover point):

使得两个算法需要相同运行时间或资源开销的问题大小。

增长量级 (order of growth):

一个函数集合，从算法分析的角度来看其中的函数的增长视为等价的。例如，线性递增的所有的函数都属于同一个增长量级。

大 O 记法 (Big-Oh notation):

代表一个增长量级的记法；例如， $O(n)$ 代表线性增长的函数集合。

线性级（linear）：

算法的运行时间和所求解问题的规模成正比，至少对大的问题规模如此。

二次方级（quadratic）

算法的运行时间和求解问题的规模的二次方 (n^2) 成正比， n 用于描述问题的规模。

搜索（search）：

在一个集合（如列表或字典）中定位某个元素位置的问题，或者判断其不在集合中。

哈希表（hashtable）：

代表键-值对集合的一种数据结构，执行搜索操作只需常数时间。

21.5.1 贡献者

1. 翻译：@SeikaScarlet
2. 校对：@bingjin
3. 参考：@carfly

译者序

我是一名自学 Python 的编程爱好者。之所以说是爱好者，是因为我不是计算机专业毕业的。我的第一个单位是一家媒体，因为工作关系对当时新闻界流行的数据可视化十分感兴趣，从《鲜活的数据》一书中了解到 Python 编程语言。而我使用的第一本教材，就是 Allen Downey 的《Think Python》，副标题叫“如何像计算机科学家一样思考”。

后来，我到了一家互联网公司，接触运营工作。我也利用业余时间开发出了自己的网站——编程派，而网站的定位就是专注 Python 编程，这也是网站名称和网站域名的由来：“编程派”中的“派”字，取自 Python 前两个字母的发音；codingpy，则是 coding python 的缩略。了解 Python 的朋友都知道，Python 脚本的默认文件名后缀就是‘.py’。

搭建“编程派”网站（后来还开通了“编程派”微信订阅号和今日头条号）的目的，就是为了能够让更多的人了解 Python 语言，更快地获取国外的 Python 教程和资源，一起交流、学习 Python。因为我相信，每个人都值得学习一点编程知识，而在我看来，最好的入门语言就是 Python。

22.1 为什么想要翻译一本书？

呃，其实我是语言类专业出身的，所以难免会有想要翻译一本书的冲动，可惜还没有出版社请我去翻书。所以只有自己上啦。

另外，这四个多月来通过网站和微信号，已经和大家分享了近百篇原创编译的 Python 教程，很多都得到了各大技术网站的转载。虽然大家在阅读文章的时候，可能并不太会注意“编程派”这个名字，但能够帮助到更多的人，还是不错的。但是这些原创编译的教程存在一个问题，就是各自是不关联的，而且也有一定的难度，对于完全零基础的人来说，根本没有什么卵用。

所以，我想成系列地与大家分享文章，而我最想帮助的人，就是那些零编程基础的朋友。因此，最适合的系列文章莫过于一本 Python 入门教材啦。

22.2 为什么选择《Think Python》

选择《Think Python》，一是因为它是我自己入门使用的第一本教材；二是，它确实非常浅显易懂，把很多概念用非常朴实的话进行了解释。至少，当时零基础的我，看这本书时没有大的障碍。

这和作者在设计、编写此书时的出发点密不可分。你可以在本书的“序言”部分（今天微信推送的另一篇文章），看到更加详细的解释。

22.3 为什么翻译 Python 3 版？

此书已经有了 Python 2 版，为什么还要更新到 Python 3 呢？

（本书最新版与此前版本的区别，请看“序言”部分。）

这是因为 Python 3 才是 Python 的未来。Python 之父 Guido van Rossum 早就坚决指出，不会再延长 Python 2 的支持协议。更具体一点说，就是到 2020 年时，Python 核心开发团队就会停止对 Python 2.7 的开发、维护工作。而且，他们也早就不再往 2.7 版中添加新特性了。

虽然现在很多公司还在使用 2.7，从 Python 2 迁移到 Python 3 的过程也非常的艰难，但是可以肯定的是，Python 3 才是 Python 的未来！

所以，我们需要 Python 3 版的《Think Python》。（另外一个原因，当然是因为 Python 2 版的已经有中译本，而且已经出版啦。。本书在翻译过程中，会参考 [车万翔老师组织翻译的 Python 2 版](<https://github.com/carfly/thinkpython-cn>)。)

22.4 贡献者

感谢下面这些朋友的贡献：

1. ipyher
2. theJian
3. lroolle
4. xpgeng
5. obserthinker
6. SeikaScarlet
7. cxyfreedom

尤其感谢 SeikaScarlet 辛苦制作精校 PDF 版。

22.5 声明

本书的原作是一本自由书籍（Free Book），以知识共享协议（[Creative Commons Attribution-NonCommercial 3.0 Unported License](<https://creativecommons.org/licenses/by-nc/3.0/us/>）发布。因此，该中译版将以同样的协议发布，不会用于商业用途。你可以自由分享、修改、复制该中译版，但请不要用于商业用途。

此书原著版权归 [Allen Downey](<http://greenteapress.com/wp/think-python-2e/>) 所有，原作链接如下：[\[https://github.com/AllenDowney/ThinkPython2\]](https://github.com/AllenDowney/ThinkPython2)(<https://github.com/AllenDowney/ThinkPython2>)。

该中译版在翻译过程中，参考了车万翔老师组织翻译的 Python 2 版：[\[https://github.com/carfly/thinkpython-cn\]](https://github.com/carfly/thinkpython-cn)(<https://github.com/carfly/thinkpython-cn>)。