

思考 Python

像计算机科学家一样思考

Version 1.1.22

思考 Python

像计算机科学家一样思考

Version 1.1.22

Allen Downey

Green Tea Press
Needham, Massachusetts

Copyright © 2008 Allen Downey.

Printing history:

2002 四月: 第一版像计算机科学家一样思考.

2007 八月: 大幅改动, 把标题改为像 (Python) 程序员一样思考.

2008 六月: 大幅改动, 把标题改为思考 Python: 像计算机科学家一样思考.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and with no Back-Cover Texts.

The GNU Free Documentation License is available from www.gnu.org or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

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

The \LaTeX source for this book is available from <http://www.thinkpython.com>

前言

0.1 本书的奇怪历史

1999 年一月份的时候，我准备用 **Java** 教一门介绍性的编程课。在那之前，我已经教了三次，而且每次我都很失望。这门课的挂课率非常之高，尽管对那些通过的学生来说，整体的水平也是很低的。

我认为问题的根源之一是教科书。教科书太厚了，掺杂着大量不必要的 **Java** 细节内容，并且没有足够高水平的引导去指导学生如何编程。学生们深陷“陷阱门”：他们起步很轻松，逐步的学习，突然，大约在第五章的某个位置，困难出现了。学生必须快速的学习大量的新内容。结果，我不得不把剩下的学期花在挑选一些片段来教学。

课程开始的前两周，我决定自力更生 --- 自己编写书。我的目标是：

- 尽量简短. 对学生来说，阅读十页比阅读五十页要好。
- 注意词汇量。我尽量减少使用术语，而且在使用前必须先定义。
- 逐步学习。为了避免陷阱门，我把最难的部分分解成一系列的小步骤。
- 把重心放在编程，而不是编程语言。我采用最少的有用的 **Java** 语言的语法，忽略其他的。

我需要一本书名，所以我就临时地把它叫做《像计算机科学家一样思考》

第一版很粗糙，但是很成功。学生们很乐意看它，并且能很好理解我在课堂上讲的难点，趣点和让他们实践的内容 (这个最重要)。

我用 GNU 自由文档许可证发布了这本书，读者们可以自由的复制，修改，发布这本书。

接下来发生的事儿极其的有趣。**Jeff Elkner**, 居住在弗尼亚的高中老师，改变了我的书，把它翻译成了 **Python**。他给我寄了份他翻译的副本，于是乎我就有了一段不寻常的学习 **Python** 的经历 -- 通过阅读我自己的书。

Jeff 和我随后修订了这本书，加入了 **Chris Meyers** 提供的一个案例学习。在 2001 年，我们共同发布了《像计算机科学家一样思考：Python 编程》，当然同样是用 GNU 自由文档许可

证。通过 Gree Tea Press, 我出版了这本书, 并且开始在亚马逊和大学书店卖纸质书。Gree Tea Press 出版的书可以从这儿获得greenteapress.com

2003 年, 我开始在 Olin College 教书。第一次, 我开始教 Python。和教授 Java 的情况相反, 学生们不再陷入泥潭, 学到了更多, 参与了很多有趣的项目, 越学越带劲。

在过去的五年里, 我一直继续完善这本书, 改正错误, 提过某些例子的质量, 加入一些其他的材料, 特别是练习。在 2008 年, 我开始重写这本书 --- 同时, 剑桥大学出版社的编辑联系到了我, 他想出版本书的下一板。美妙的时刻!

结构就出现了现在的这本书, 不过有了一个简洁的名字《思考 Python》。变化有:

- 在每一章末尾加了点调试的部分。这些部分提供了发现和避免 bug 的通用技巧, 也对 Python 的陷阱提出了警告。
- 删除了最后几章关于列表和树实现的内容。虽然, 我万分不舍, 但是考虑到和本书余下的部分不协调, 只能忍痛割爱。
- 增加了一些案例学习 --- 提供了练习, 答案和相关讨论的大例子。一些东西是基于 Swampy, 这是我为了教学而设计的 Python 程序。Swampy, 代码实例和部分答案可以从这儿获得thinkpython.com.
- 扩展了关于程序构建计划和基本的设计模式的讨论。
- Python 运用的更加地道。虽然这本书仍然是讨论编程的, 而不是 Python 本身, 但是现在我不得不承认这本书深受 Python 浸染。

我希望读者们可以享受这本书, 也希望帮助你学习程序设计和像计算机科学家一样思考, 哪怕是一丁点儿。

Allen B. Downey
Needham MA

Allen Downey 是 Olin College 大学计算机科学与技术系的副教授。

致谢

首先也是最重要的, 我要感谢 Jeff Elkner 将我的 Java 教材译为 Python 版本, 为此项目奠基并且向我介绍了 Python 语言, 它现在已经是我的首选编程语言。

感谢 Chris Meyers, 向如何像一个计算机科学家思考一书也就是本书的前身贡献了若干内容。

感谢自由软件基金会开发了 GNU Free Document License 这一授权协议, 使我与 Jeff, Chris 的共事得以成为现实。

感谢 Lulu 的编辑们为本书前身如何像一个计算机科学家思考付出的努力。

感谢所有为此书早期版本做出努力的学生与所有向我寄送更正与建议的贡献者 (在后面列出)。

更感谢我妻子 Lisa 为此书, 还有绿茶出版, 以及其他所有事情做出的工作。

贡献者列表

100 多位目光锐利，头脑灵活的读者在过去的几年中寄给我许多建议与更正。他们的贡献和热忱对项目建功颇多。

如果你有建议或者更正，也请寄送 email 至 feedback@thinkpython.com。如果我基於你的反馈做出了修改，你将被列於此贡献者列表（除非你自己申明愿意被忽略）。

如果能够附上出现错误的句子，哪怕是一部分，将有助我进行搜索。页码与章节序号亦可，不过句子更方便。谢谢。

- Lloyd Hugh Allen 提交了第 8.4 节的一个更正。
- Yvon Boulianne 寄给我第 5 章的一个更正。
- Fred Bremmer 提交了第 2.1 节的一个更正。
- Jonah Cohen 撰写了将 LaTeX 源代码转换为漂亮的 HTML 文件的 Perl 脚本。
- Michael Conlon 提交了第 2 章的一个语法更正与第 1 章的一个文风改善意见，还发起了关于解释器的技术层面的讨论。
- Benoit Girard 提交了第 5.6 节的一个比较滑稽错误的更正。
- Courtney Gleason 与 Katherine Smith 撰写了 `horsebet.py`，用作此书早期版本的一个参考案例。此程序可以在网上找到。
- Lee Harr 提交了很多更正，因为空间有限无法一一列出。因为贡献较大，其实他可以称作此书的重要编辑之一。
- James Kaylin 是一个使用此教材的学生，提交了大量的更正。
- David Kershaw 修正了第 3.10 节的 `catTwice` 函数。
- Eddie Lam 提交了第 1, 2, 3 章的大量更正。他也修正了 `Makefile`，使得在第一次运行时可以生成一个索引。本书的版本计划也由他发起。
- Man-Yong Lee 提交了第 2.4 节的一个例题代码的一个更正。
- David Mayo 指出第 1 章的一个用词不当。
- Chris McAloon 提交了第 3.9 节与第 3.10 节的若干更正。
- Matthew J. Moelter 曾长期关注此书，提交了大量的更正与建议。
- Simon Dicon Montford 报告了第 3 章的一个缺失函数定义错误与若干拼写错误。亦指出第 13 章中 `increment` 函数的错误。
- John Ouzts 更正了第 3 章中的关于返回值的定义。
- Kevin Parks 针对如何提高此书的发行质量提交了宝贵的评论与建议。
- David Pool 提交了第 1 章的术语表中一个拼写错误，而且给我们打气。
- Michael Schmitt 提交了关于文件于异常章节中的一个更正。
- Robin Shaw 指出了 13.1 中的一个错误（一个范例中使用 `printTime` 函数但是没有事先定义）。
- Paul Sleight 找出了第 7 章的一个错误，且发现了 Jonah Cohen 用来从 LaTeX 生成 HTML 页面的 Perl 脚本中的一个 bug。
- Craig T. Snyder 于 Drew 大学实践此教材。他贡献了数处有用的建议与更正。
- Ian Thomas 与他的学生在一个编程课程中使用此教材。他们是此书后面一半章节最早的实践者，也提出了大量的更正与建议。
- Keith Verheyden 提交了第 3 章的一个更正。
- Peter Winstanley 给我们指出了第 3 章中一个长期存在的错误。
- Chris Wrobel 贡献了若干关于文件 I/O 与异常章节中的更正。

- Moshe Zadka 对此项目做出了难能可贵的贡献. 除了为此书开头部分撰写了辞典范例相关内容, 在整个此书的早期阶段他一直提供指导。
- Christoph Zwerschke 提交了数处更正与教学考虑上的建议, 并向我们解释了 `gleich` 与 `selbe` 之间的差别。
- James Mayer 向我们提交了一大堆的拼写与录入错误, 包含这个贡献者列表中的两个。
- Hayden McAfee 指出了两个范例中可能存在的令人困惑的不一致。
- Angel Arnal 是本书的西班牙文版本翻译组成员之一. 他亦找出了英文版本的数处错误。
- Tauhidul Hoque 与 Lex Berezhny 创建了第 1 章插图并提高了其他数处插图的质量。
- Dr. Michele Alzetta 找出了第 8 章的一个错误并且针对 Fibonacci 与 Old Maid 这两个范例提交了一些从教学方面的评论与建议。
- Andy Mitchell 找出了第 1 章的一个拼写错误与第 2 章的某范例的阙漏之处。
- Kalin Harvey 建议对第 7 章的一处概念澄清并且提交了一些拼写错误。
- Christopher P. Smith 找出了数处拼写错误并且协助我们将此书针对 Python 2.2 做了更新。
- David Hutchins 找出了序中的一个拼写错误。
- Gregor Lingl 於奥地利维也纳的某高中教授 Python, 他正在从事此书的德文翻译并且找出了第 5 章的若干重要错误。
- Julie Peters 找出了前言中的一个拼写错误。
- Florin Oprina 提交了 `makeTime` 中的一个改善, 一个 `printTime` 中的更正, 与一个拼写错误。
- D. J. Webre 建议了第 3 章的一个概念澄清。
- Ken 找出了第 8, 9, 11 章中的大量错误。
- Ivo Wever 找出了第 5 章中的一个拼写错误并且建议了第 3 章中的一个概念澄清。
- Curtis Yanko 建议了第 2 章中的一个概念澄清。
- Ben Logan 提交了大量的拼写错误与若干将此书转换为 HTML 格式的错误。
- Jason Armstrong 指出了第 2 章一处语法阙漏。
- Louis Cordier 指出第 16 章某处代码与文字不相符合。
- Brian Cain 建议了数处第 2 章与第 3 章的若干概念澄清。
- Rob Black 提交了大量更正, 包含某些针对 Python 2.2 的修改。
- Jean-Philippe Rey 来自於巴黎的 Ecole Centrale, 提交了大量的补丁, 包含某些针对 Python 2.2 的修改与一些有想法的改善。
- Jason Mader 来自於 George Washington 大学提交了大量的有用的建议与更正。
- Jan Gundtofte-Bruun 指出一个语法错误。
- Abel David 与 Alexis Dinno 提醒我们 `matrix` 的复数形式为 `matrices`, 而非 `matrixes`。此错误存在於本书中有数年之久, 但是两位读者在同一天向我们指出了, 不可不谓奇。
- Charles Thayer 指出了数处语法错误。
- Roger Sperberg 指出了第 3 章某处逻辑有误。
- Sam Bull 指出了第 2 章某处令人困惑的段落。
- Andrew Cheung 指出了两处使用之后才定义的错误。
- C. Corey Capel 指出了一个语法错误与一个拼写错误。
- Alessandra 帮助我们澄清了某些概念。
- Wim Champagne 在辞典范例中指出一处错误。
- Douglas Wright 指出了 `arc` 中的一个除法相关错误。
- Jared Spindor 指出一些语法错误。
- Lin Peiheng 提交了大量的非常有益的建议。

- Ray Hagtvedt 提交了两处错误与一处似是而非的地方。
- Torsten Hübsch 指出了 Swampy 范例中某处不一致。
- Inga Petuhhov 更正了第 14 章中的某个一个范例。
- Arne Babenhauserheide 提交了数处有益的更正。
- Mark E. Casida 非常擅长於给我们指出重复用词的地方。
- Scott Tyler 提交了大量更正。
- Gordon Shephard 给我们发送了多封信件，提交了数处更正。
- Andrew Turner 指出了第 8 章的一个错误。
- Adam Hobart 修正了 `arc` 中的一个除法相关错误。
- Daryl Hammond 与 Sarah Zimmerman 指出我过早地提到了 `math.pi`。Zim 也指出了一个拼写错误。
- George Sass 在调试章节指出了一个错误。
- Brian Bingham 建议了练习题 ??。
- Leah Engelbert-Fenton 指出了我使用 `tuple` 作为变量名, 与之前的我自己的建议背道而驰。之后又发现了大量拼写错误与一处“定义之前使用”。
- Joe Funke 指出了一个拼写错误。
- Chao-chao Chen 发现了 Fibonacci 范例中的某处不一致。
- Jeff Paine 指出了一处语法错误。
- Lubos Pintes 提交了一个拼写错误。
- Gregg Lind 与 Abigail Heithoff 建议了练习题 ??。
- Max Hailperin 提交了大量的更正与建议。Max 是著名的 *Concrete Abstractions* 一书的作者之一, 你读完此书再把那本书找来阅读学习效果可能会更好。。
- Chotipat Pornavalai 指出了一个错误信息中的错误。
- Stanislaw Antol 提交了大量的非常有益的建议。
- Eric Pashman 提交了第 4--11 章的大量的更正。
- Miguel Azevedo 指出若干拼写错误。
- Jianhua Liu 提交了大量的更正。
- Nick King 指出一处语法漏洞。
- Martin Zuther 提交了大量的建议。
- Adam Zimmerman 指出数处错误。
- Ratnakar Tiwari 建议了一个解释退化三角形的脚注。
- Anurag Goel 建议了若干更正。
- Kelli Kratzer 指出一处拼写错误。
- Mark Griffiths 指出了第 3 章的范例中某令人困惑之处。
- Roydan Ongie 指出关于 Newton 法的一个错误。
- Patryk Wolowiec 协助我修正了关于 HTML 版本的一个错误。
- Mark Chonofsky 告诉我 Python 3.0 的一个新的关键词。
- Russell Coleman 在几何学方面协助过我。
- Wei Huang 指出数处录入错误。
- Karen Barber 指出书中最古老的拼写错误之一。
- Nam Nguyen 指出一个拼写错误并且指出了我使用了 Decorator 模式但是没有提到它的名字。
- Stéphane Morin 提交了数处更正与建议。
- Paul Stoop 更正了一个拼写错误。
- Eric Bronner 指出了操作符优先级讨论中的一处困惑之处。
- Alexandros Gezerlis 指出了大量的高质量的建议，我们相当感激！
- Gray Thomas 指出了一处错误。

Contents

前言	v
0.1 本书的奇怪历史	v
1 编程的方式	1
1.1 Python 编程语言	1
1.2 什么是程序	2
1.3 什么是调试?	3
1.4 语义错误	3
1.5 正式语言和自然语言	4
1.6 第一个程序	5
1.7 调试	6
1.8 术语表	6
1.9 练习	7
2 变量、表达式和语句	9
2.1 变量和数据类型	9
2.2 变量	10
2.3 变量命名和关键字	11
2.4 程序语句	12
2.5 运算符和运算数	12
2.6 表达式	13
2.7 运算符的优先级	13
2.8 字符串操作	13
2.9 注释	14
2.10 调试	15
2.11 术语表	15
2.12 练习	16

3	函数	17
3.1	类型转换函数	17
3.2	数学函数	18
3.3	创建	19
3.4	编写新的函数	19
3.5	定义和使用	20
3.6	执行流	21
3.7	形参和实参	21
3.8	变量和参数是局部的	22
3.9	Stack diagrams 堆栈示意图	23
3.10	“结果”函数和 void 函数	24
3.11	为什么要函数	25
3.12	调试	25
3.13	术语表	25
3.14	练习	26
4	实例学习：接口设计	29
4.1	TurtleWorld	29
4.2	简单重复	30
4.3	练习	31
4.4	封装	31
4.5	泛化	32
4.6	接口设计	32
4.7	重构	33
4.8	一个开发者的计划	34
4.9	文档字符串	34
4.10	调试	35
4.11	术语表	35
4.12	练习	36

5	条件语句和递归	37
5.1	模操作符	37
5.2	布尔表达式	37
5.3	逻辑运算符	38
5.4	条件执行	38
5.5	选择执行	39
5.6	链式条件	39
5.7	嵌套的条件语句	40
5.8	递归	40
5.9	递归函数的堆栈图	42
5.10	无穷递归	42
5.11	键盘输入	43
5.12	Debugging 调试	43
5.13	术语表	44
5.14	练习	45
6	“结果” 函数	47
6.1	返回值	47
6.2	增量开发	48
6.3	Composition	50
6.4	布尔函数	50
6.5	更多递归	51
6.6	信心的飞跃	53
6.7	另一个例子	53
6.8	类型检查	53
6.9	调试	54
6.10	术语	55
6.11	练习	56

7	迭代器	59
7.1	多重赋值	59
7.2	更新变量	60
7.3	<code>while</code> 语句	60
7.4	<code>break</code> 语句	61
7.5	平方根	62
7.6	算法	63
7.7	调试	64
7.8	术语表	64
7.9	练习	64
8	字符串	67
8.1	字符串是一个序列	67
8.2	<code>len</code>	67
8.3	使用 <code>for</code> 循环遍历	68
8.4	字符串切片	69
8.5	字符串是不变的	70
8.6	搜索	70
8.7	循环和记数	70
8.8	<code>string</code> 方法	71
8.9	<code>in</code> 运算符	72
8.10	字符串比较	72
8.11	调试	73
8.12	术语	74
8.13	练习	75
9	实例学习：字符处理	77
9.1	读取单词表	77
9.2	练习	78
9.3	搜索	78
9.4	使用索引循环	80
9.5	调试	81
9.6	术语表	81
9.7	练习	82

10 列表	83
10.1 列表是一个序列	83
10.2 列表是可改变的	83
10.3 遍历列表	85
10.4 列表操作	85
10.5 列表切片	85
10.6 列表方法	86
10.7 映射, 筛选和归并	87
10.8 删除元素	88
10.9 列表和字符串	88
10.10 对象和值	89
10.11 别名	90
10.12 列表参数	91
10.13 调试	92
10.14 术语	93
10.15 练习	94
11 字典	95
11.1 把字典作为计数器	96
11.2 循环和字典	97
11.3 颠倒查询	98
11.4 字典和列表	99
11.5 备忘录	100
11.6 全局变量	101
11.7 长整数	103
11.8 调试	103
11.9 术语表	104

12 元组	107
12.1 元组是不可变的	107
12.2 元组赋值	108
12.3 元组作为返回值	109
12.4 变长参数元组	109
12.5 列表和元组	110
12.6 字典和元组	111
12.7 元组比较	112
12.8 序列的序列	113
12.9 调试	114
12.10 术语	115
12.11 练习	115
13 实例学习：数据结构的实例	117
13.1 单词频率分析	117
13.2 随机数字	117
13.3 单词频率直方图	118
13.4 最常用单词	119
13.5 可变参数	120
13.6 字典减法	121
13.7 随机单词	121
13.8 马尔可夫分析	122
13.9 数据结构	123
13.10 调试	124
13.11 术语表	125
13.12 练习	125
14 文件	127
14.1 持久性	127
14.2 读取和写入	127
14.3 格式运算符	128
14.4 文件名和路径	129

14.5	捕获异常	130
14.6	数据库	131
14.7	Pickling	131
14.8	管道	132
14.9	编写模块	133
14.10	调试	134
14.11	术语	134
14.12	练习	135
15	类和对象	137
15.1	自定义类型	137
15.2	属性	138
15.3	矩形	139
15.4	实例作为返回值	140
15.5	对象是可变的	140
15.6	复制	141
15.7	调试	142
15.8	术语表	143
15.9	练习	143
16	类和函数	145
16.1	时间	145
16.2	纯函数	145
16.3	修改函数	147
16.4	原型与计划	147
16.5	调试	148
16.6	术语	149
16.7	练习	149

17 类和方法	151
17.1 面向对象特点	151
17.2 Printing objects	152
17.3 另外一个例子	153
17.4 更复杂的一个例子	153
17.5 初始方法	154
17.6 __str__方法	154
17.7 运算符重载	155
17.8 基于类型的调度	155
17.9 多态	157
17.10 调试	157
17.11 术语表	158
17.12 练习	158
18 继承	161
18.1 扑克对象	161
18.2 类属性	162
18.3 比较卡片	163
18.4 纸牌	164
18.5 打印纸牌	164
18.6 添加, 删除, 洗牌, 理牌	165
18.7 继成	166
18.8 类图	167
18.9 调试	168
18.10 术语	169
18.11 练习	169
19 实例学习: Tkinter	171
19.1 GUI	171
19.2 按钮和回调	172
19.3 画布控件	173
19.4 坐标序列	174

19.5	更多的控件	174
19.6	排列控件	175
19.7	菜单和可召唤的	177
19.8	Binding 绑定	178
19.9	调试	180
19.10	术语表	181
19.11	练习	181
A	调试	183
A.1	语法错误	183
A.2	运行时错误	184
A.3	语义错误	187

Chapter 1

编程的方式

这本书的目的是教会大家如何像计算机科学家一样思考。计算机科学用严谨的语言来表明思想,尤其是计算。像工程师,他们设计,把各个组件装配成系统并且在可选方案中评估出折中方案。像科学家,他们研究复杂系统的性能,做出假定并且测试假设。

对一个计算机科学家来说最重要的是解决问题。解决问题意味着清晰明确的阐述问题,积极思考问题答案,并且清楚正确的表达答案的能力。实践证明:学习如何编程是一种很好的机会来练习解决问题的技巧。这也是为什么把这章叫做“编程的方式”。

一方面,你将学习编程,一个非常有用的技巧。另一方面你将会把编程作为一种科技。随着我们的深入学习,这点会渐渐明晰。

1.1 Python 编程语言

我们将要学习的编程语言是 Python。Python 仅是高级语言中的一种,你可能也听说过其他的高级编程语言,比如 C,C++,Perl, 和 Java。

也有一些低级语言,有时也被称为机器语言或者汇编语言。一般来说,计算机只能执行用低级语言编写的程序。所以,用高级语言编写的程序在执行前必须做相应的处理。这会花费一定的时间,同时这也是高级语言的“缺点”。

然而,高级语言的优点也是无限的。首先,用高级语言编程是一件非常容易的。用高级语言编程通常花费的时间比较少,同时编写的程序简短,易读,易纠错。第二,高级语言是可移植的,这意味着他们可以不加修改(或者修改很少)地运行在不同的平台上。低级语言编写的程序只能在一个机器上运行,如果想要运行在另外一台机器上,必须得重写。

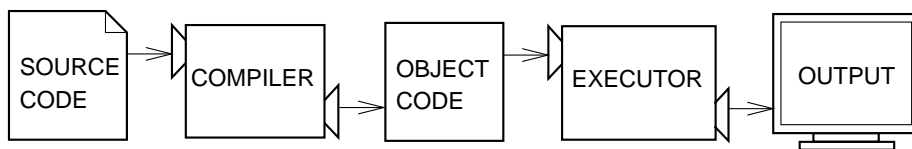
基于这些优点,几乎所有的程序都是用高级语言编写的。低级语言统称仅仅用在一些专门的应用程序中。

有两种程序把高级语言“处理”成低级语言:解释器和编译器。解释器读取源程序,解释执行,也就是按照程序表达的意思"做事"。解释器一次解释一点,或者说,一次读取一行,

然后执行。



编译器读取程序，完全转换之。在这种情况下，高级语言程序叫做源码，编译后的程序叫做目标代码或者叫可执行代码。一旦程序被编译，就可以直接执行，无须再编译。



一般地，我们把 `python` 当作是解释型语言，因为用 `Python` 编写的程序是通过解释器执行的。有两种使用解释器的方式：交互模式和脚本模式。在交互模式下，你可以输入 `Python` 程序，然后解释器输出结果：

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

锯齿符，`>>>`，是提示符，解释器用它来表明自己已经准备好了，如果你输入 `1 + 1`，解释器显示 `2`。

另外地，我们可以把代码存储在一个文件里，使用解释器执行文件，此时这个文件被称作脚本。习惯上，`Python` 脚本的扩展名为 `.py`。

如果要执行 `Python` 脚本，我们必须提供给解释器脚本的文件名。在 `UNIX` 命令窗口，可以输入 `python dinsdale.py`。在其他开发环境中，会有些细节方面的差别。可以在 `Python` 官网上 (python.org) 找到相应的指导。

在交互模式下工作很容易测试一小段代码，因为可以随时输入，并且立刻执行。但如果代码量较大，我们必须把代码存放在脚本里，这样方便我们以后修改执行。

1.2 什么是程序

程序就是指令集合，这些指令说明了如何执行计算。计算可能是数学上的，例如解决等式组或者计算多项式的平方根。但是也可以是符号计算，比如搜索替换文件的文本或者 (很奇怪) 编译一个程序。

不同的语言有一些细节上的差异。但是他们有一些共有的指令：

输入：从键盘获取数据，文件，或者从其他设备。

输出：在显示器上显示数据或者把数据输出到文件或其他设备。

数学运算：做基本的数学操作像加法和乘法。

条件执行：检查条件，然后执行正确的语句。

循环：重复执行一些动作，通常有些变化。

信不信由你，就是这样。我们用过的任何一个软件，无论多么复杂，基本上都是由与这些相似的指令组成。所以，我们可以这么理解：编程就是把复杂庞大的任务分解为一系列的小任务，知道这些小任务简单到可以用这些基本的指令表示。

这个有点模糊，但是当我们讲到算法的时候，我们再回过头来聊这个话题。

1.3 什么是调试?

有三种错误经常在程序中出现：语法错误，运行时错误和语义错误。为了能够快速的跟踪捕捉到他们，区分他们之间的诧异还是很有好处的。

1.3.1 语法错误

Python 只能执行语法正确的程序；否则，解释器就会报错。语法指的是程序的结构和结构的规则。比如，括号必须是成对出现，所以 `(1 + 2)` 是合法的，但 `8)` 就是语法错误。

在英语中，读者可以忍受大多数语法错误，这就是为什么我们玩味 E. E 康明思的诗歌，而没有提出任何错误信息的原因。**Python** 不会这么仁慈。如果你程序的某个地方出现了哪怕是一个语法错误，**Python** 也会显示错误信息然后退出，你也不能再继续执行程序。在你初学编程的几周里，你很可能会花费大量的时间追踪，捕捉语法错误。一旦你有经验了，你犯的错误就更少，并且也能很快的发现他们。

1.3.2 运行时错误

第二中错误是运行时错误，之所以这么命名是因为从这种错误知道程序开始运行才会出现。这些错误也叫做异常，因为他们通常表明异常的事情发生了。

运行时错误在前几章的简短的代码中比较少见，因此你可能会有一段时间才会遇到。

1.4 语义错误

第三中错误是语义错误。如果有语义错误，程序会成功运行（即计算机不会产生任何的错误信息），但是它却没有做对！计算机做了另外的事。确切的说，计算机确实做了你告诉他的指令。

1.4.1 试验性的调试

你必须拥有的一条技能是调试。尽管在这个过程中，你可能很受伤，但，调试是编程中最具有挑战，最有意思，最能考验智力的一部分。

某种程度上，调试就像是侦探。你面对着很多线索，必须推断导致你看到的结果的过程和事件。

调试也像是一个科学实验。一旦你意识到错误的地方，改正她，再尝试。如果你的假想是正确的，你就可以预测出改变带来的结果，你也就离能够执行的程序更近一步了。如果你的猜想是错误的，你不得不提出一个新的。正如 Sherlock Holmes 指出的，“当你移除了不可能的，留下来的无论是什么，也不论多么不可能，都是真理。(A. Conan Doyle, The Sign of Four)”

对某些人来说，编程和调试是同时完成的。也就是，编程是不断调试，直到看到想要结果的过程。理念就是：你必须以一个能够工作的程序开始，然后做些小改动，随着进度不断调试他们，这样就总是有一个可工作的程序。

比如：Linux 是一个包含成千上万行代码的操作系统，但它也是从一个 Linux Torvalds 用来研究 Intel 80386 芯片的小程序开始的。按照 Larry Greenfield 的说法，“Linus 的早期项目就是一个在打印 AAAA 和 BBBB 之间切换的程序。”(The Linux Users' Guide Beta Version 1)。

接下来的章节将介绍更多的调试建议还有其他的编程经验。

1.5 正式语言和自然语言

自然语言是人们日常说的语言，比如英语，西班牙语和法语。他们不是人民设计的（尽管人们努力的强加一些规则）；他们是自然发展的。

正式语言是人们为了特别的应用而设计的语言。比如，数学家使用的符号就是一门正式语言，它很擅长揭示数字和符号之间的联系。化学家用正式语言代表分子的化学结构。最重要的是：

编程语言是正式语言，是被设计来表达计算的。

正式语言倾向于有严谨的语法规则。比如， $3 + 3 = 6$ 是语法争取的数学语句。但是 $3+ = 3\$6$ 不是。 H_2O 是语法正确的化学分子式，但 $_2Zz$ 不是。

语法规则涉及到两个方面：标记和结构。标记是语言的最基本元素，比如字，数字和化学元素。 $3+ = 3\$6$ 的一个问题是 $\$$ 不是一个合法的数学标记（至少据我所知）。相似的， $_2Zz$ 不合法是因为没有元素的缩写是 Zz 。

第二种语法错误涉及到语句的结构，也就是，标记被安排的方式。语句 $3+ = 3\$6$ 是非法的因为尽管 $+$ 和 $=$ 是合法的标记，但我们不能把两个相连。同样的，在化学分子式中，下标必须在元素之后，不是前面。

Exercise 1.1 写一个结构正确的英语句子，同时标记也必须合法。然后写一个结构不合理但是标记合法的句子。

当阅读一个英文句子或者正式语言的一个语句，必须明确句子的结构（尽管对于自然语言来说，这个是潜意识的）。这个过程叫做句法分析。

比如，当你听到一个句子，“一便士硬币掉了”，你理解“一便士硬币”是主语，“掉了”是谓语。一旦你分析了这个句子，你就明确句子的意思。假如你知道一个便士是什么，并且

什么是掉了，你就会明白这个句子的一般含意。

尽管正式语言和自然语言有很多共同点 --- 标记，结构，语法和语义 --- 也存在一些不同点：

二义性： 自然语言充满了二义性（模糊性），人们利用上下文来区分。正式语言被设计成几乎没有二义性，这也意味着每个语句都有明确的意思，无论上下文。

冗余性： 为了弥补二义性和减少误解，自然语言设置了很多冗余。因此自然语言是冗长的。自然语言更简短，精确。

无修饰性： 自然语言充满了习语和隐喻。如果我说“一便士硬币掉了”，也许根本没有便士也没有东西掉了¹。正式语言表达了是精确的意思。

成长过程中，说自然语言的人 --- 每个人 --- 通常在调整自己适应正式语言的过程中都会经历痛苦。某种程度上，正式语言和自然语言之间的区别就像诗歌和散文²之间的区别，甚至更多：

诗歌： 单词的运用既是为了语义的需要，也是为了音韵的需要，整首诗创造了一种情感共鸣。二义性不仅很常见，而且常常是故意安排的。

散文： 单词的字面意思更加重要，结构也表达了更多的意思。散文比诗歌更容易分析，但是仍然具有二义性。

程序： 计算机程序是无二义性。可以通过分析标记和结构完全理解。

这里给些读程序时候的一些建议（包括其他正式语言）。第一，记住正式语言是比自然语言要晦涩的，所以要花长时间阅读。其次，结构也是非常重要的，所以，从头到位，从左到右阅读通常不是一个好的办法，可以学习在大脑中分析程序，识别标记的意思，然后解释结构。最后，细节也很重要。一些拼写和标点上细小的错误（在自然语言中可以忽略的），有时会在正式语言中掀起大浪。

1.6 第一个程序

通常，学习新语言的第一个程序就是"hello world"，应为所做的就是显示单词，"Hello, World!"。在 Python 中，看起来是：

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

这是一个 `print` 语句的例子³，没有真正在纸上打印东西。它在显示器上显示了一个值。在这种情况下，结果是单词

Hello, World!

程序中的引号标志了要被显示的文本的开始和结束，他们不会出现在结果中。

一些人通过"Hello, World!" 程序的简洁程度来判断编程语言的好坏。按照这个标准，Python 确实非常好！

¹这个习语意思是某人困惑之后恍然大悟。

²译者：这里的散文不是诗化的散文，像余光中老前辈开启的诗化散文

³在 Python 3.0 中，`print` 是一个函数，不是一个语句了，所以语法是 `print("Hello, World!")`。我们不久就要接触到函数了！译注：在本书翻译时 python 2.7 和 python 3.1 已经发布，python 3.2 的 release 版也即将发布

1.7 调试

坐在电脑前面看这本书是个不错的方法，你可以随时尝试书中的例子。你可以在交互模式下运行大多数的程序，但是如果你把代码放在一个脚本里，也是很容易尝试改变一些内容的。⁴

无论何时，尝试一个新的特点的时候，你应该故意的犯些错误。比如，在"Hello, World!"程序中，如果忽略了双引号其中之一，会发生什么？如果把两个引号都忽略了，又会怎样？如果拼错了 `print` 了呢？

这种实验能够有效的帮助你记住你看的内容，同时也对调试有好处，因为你知道了错误信息的意思了。现在故意的犯错误总比以后猝不及防的犯错误要好的多。

编程，特别是调试，有时带来很强的情绪。你在一个困难的 `bug` 里苦苦挣扎，你可能变得怒不可遏，苦恼不堪，甚至羞愧不已。

有证据表明，人们很容易把电脑当成人来对待⁵。当电脑工作正常，我们把它们当作是队友，当电脑不给力时，我们把它们当成粗鲁顽固的人。

为这些反应作准备也许会帮助你合理的处理。一个方法是把电脑当作一个员工，他既拥有一定力量，比如速度和精度，也会有特别的缺点，比如缺少默契，没有能力理解大的图片。

你的工作就是做一个好的经理：发掘有效的方法扬长补短。并且寻找方法利用你的情绪来投入到解决问题中，不要让你的（不良）反应干扰你工作的能力。

学习调试是令人沮丧的，但是一种宝贵的技巧，在编程的其他领域也是大有裨益的。在每章的末尾，都有一个调试段落，像这个一样，是我调试经验的总结。我希望他们对你有帮助！

1.8 术语表

problem solving 问题解决： 表述问题，发现解，表达解的过程。

high-level language 高级语言： 像 `Python` 一样的程序设计语言，被设计让人们易读易写程序。

low-level language 低级语言： 设计让计算机容易执行的程序设计语言；也叫做“机器语言”或者“汇编语言”。

portability 可移植性： 程序可以在一台或多台电脑执行的属性。

interpret 解释： 逐行逐行解释执行用高级语言编写的程序。

compile 编译： 把用高级语言编写的程序转换成低级语言。

⁴译者注：我的理解是，可以很方便的改动某些变量或者语句，然后执行

⁵参看 Reeves 和 Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*.

source code 源码： 未编译的高级语言编写的程序。

object code 目标代码： 编译器转换程序后的输出。

executable 可执行代码： 目标代码的别名，可以被执行。

executable 可执行代码

prompt 提示符： 解释器显示的字符，表明做好准备让用户输入。

script 脚本： 存储在文件中的程序。

interactive mode 交互模式： 一种通过输入命令和表达式的使用 **python** 解释器的方式。

script mode 脚本模式： 一种使用 **Python** 解释器的方式，**Python** 解释器读取脚本中的语句执行。

program 程序： 指明计算的指令集合。

algorithm 算法 求解一类问题的通用过程。

bug: 程序的错误。

debugging 调试： 发现，去除程序错误的过程。

syntax 语法 程序的结构。

syntax error 语法错误： 使程序不能正确解析的错误。

exception 异常： 程序在运行时发现的错误。

semantics 语义： 程序的含意。

semantics error 语义错误： 程序中的错误，使计算机执行另外的程序。

natural language 自然语言： 人们日常交流用的语言，自然发展的。

formal language 正式语言： 人民为了某种特殊目的设计的语言，比如，代表数学思想或者计算机程序，所有的程序设计语言都是正式语言。

token 标记： 程序语法结构的最基本元素，类似于自然语言的单词。

parse 句法分析： 检查程序，分析语法结构。

print statement **print** 语句： 一条指示 **Python** 解释器显示一个值的指令。

1.9 练习

Exercise 1.2 打开浏览器浏览 **Python** 官网 python.org. 这个页面包含了 **Python** 的一些信息，还有和 **Python** 相关的连接。你可以查看 **Python** 官方文档。

比如，在搜索框里输入 **print**，第一个链接就是 **print** 语句的文档。此时，并不是所有的信息对你都有意义，但是知道它们在哪里总是有好处的。

Exercise 1.3 启动 Python 的解释器，输入 `help()` 启动在线帮助工具。或者你也可以输入 `help('print')` 获得关于 `print` 语句的信息。

如果没有成功，你或许需要安装额外的 Python 官方文档，或者设置环境变量。这个依赖于你使用的操作系统和 Python 解释器版本。

Exercise 1.4 打开 Python 解释器，我们暂且把它作为计算器。关于数学操作的语法，Python 和标准的数学符号很相似。比如，符号 `+`、`-` 和 `/` 表示加减，除。乘法的符号是 `*`。

如果 43 分钟 30 秒，跑了 10 公里，每英里花费的时间是多少？你的平均速度是多少英里每小时？（**Hint:** 一英里等于 1.61 公里）。

Chapter 2

变量、表达式和语句

2.1 变量和数据类型

变量是程序处理的最基本的事物，就像一个字母或者一个数字。至今为止我们见过的变量有 1, 2 以及 'Hello, World!'。

这些变量属于不同数据类型：2 是一个整数，而 'Hello, World!' 是一个字符串，之所以这么称呼是因为它包含一“串”字母。因为被引号包围，读者（以及解释器）可以将它们识别为字符串。

print 命令同样适用于整数。

```
>>> print 4
4
```

如果你不确定一个变量的数据类型，解释器会告诉你。

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

显然，字符串属于类型 `str`，整数属于类型 `int`。同样，带小数点的数字属于 `float` 类型，因为这些数字通过一种叫做 `floating-point` 的数据类型来表示。

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

那么对于像 '17' 和 '3.2' 呢？它们虽然看起来像数字，实际上由于它们在引号中，所以是字符串。

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

它们是数字。

当你输入一个大整数时，你会习惯性地 3 个数子中间加上逗号，如 1,000,000。对于 Python 来说这不是一个有效的数字，但是这样的表示是合法的：

```
>>> print 1,000,000
1 0 0
```

当然，这不是我们所期待的。Python 将 1,000,000 解释为一个用逗号划分的数字序列，在输出时用空格区分。

这是我们见到的第一个语义错误的例子：代码可以运行，没有报出错信息，但是没有做“正确”的事情。

2.2 变量

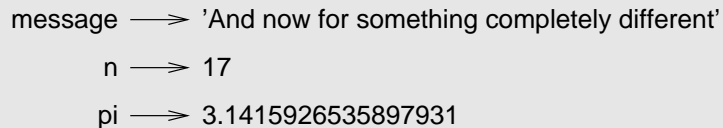
对变量的操作是编程语言最强大的特征之一。一个变量是一个对应值的名称。

通过赋值语句创建新的变量，并对它们赋值：

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

这个例子包含三条赋值语句。第一条将一个字符串赋值给一个叫做 `message` 的变量；第二条将整数 17 赋值给变量 `n`；第三条将 π 的近似值赋值给变量 `pi`。

变量通常用表示为变量名用一个肩头指向变量的值。这样的图称作状态图表，因为它给出了每个变量所处的状态（想象各个变量处在各自的状态）。以下的图表给出了之前例子中的结果：



```
message —> 'And now for something completely different'
      n —> 17
      pi —> 3.1415926535897931
```

你可以使用一个 `print` 语句来显示变量的值：

```
>>> print n
17
>>> print pi
3.14159265359
```

一个变量的数据类型是这个变量对应值的数据类型。

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

Exercise 2.1 如果你在输入一个整数时以 0 开头，你会得到一个奇怪的错误：

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

其他的数字也许可以工作，但是结果是诡异的：

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

你能指出这是为什么吗？提示：打印数值 01，010，0100 和 01000。

2.3 变量命名和关键字

程序员通常选取有意义的名字作为变量名——变量名说明了变量的作用。

变量名可以任意长。它们可以同时包含字母和数字，但必须以一个字母开头。大写字母的使用是合法的，但是一个好的习惯是使用小写字母作为变量的首字母（后面会说明原因）。

下划线符号 (`_`) 可以出现在一个变量名中，通常使用在有多个单词的名字中，如 `my_name` 或者 `airspeed_of_unladen_swallow`。

如果你赋予了一个变量非法的名字，你会得到这样的一个语法错误：

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

`76trombones` 是非法的，因为它不是有字母开始。`more@` 是非法的，因为它包含了非法字符 `@`。但是 `class` 哪里错了呢？

事实上 `class` 属于 Python 的一个关键字。解释器使用这些关键字来识别程序的结构，因此它们不能被用做变量名。

Python 共有 31 个关键字¹：

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

你可以放一份关键字列表在手边。如果解释器报错说变量名有误，而你有不知道原因，看看这个变量名是否在关键字列表上。

¹在 Python 3.0 中，`exec` 不再是一个关键字，但 `nonlocal` 是。

2.4 程序语句

程序语句是一组 Python 解释器可以执行的代码。我们已经遇到过两种程序语句：`print` 和 `assignment`。

当你在交互模式下输入一条程序语句，解释器会执行该条语句，并显示结果，如果有结果的话。

一个脚本通常包含一系列程序语句。如果有多于一条程序语句，每执行一条语句将会显示对应的结果。

例如，脚本

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

会产生输出

```
1
2
```

赋值语句不产生输出。

2.5 运算符和运算数

运算符是表示计算的特殊符号，如加法和乘法。运算符作用的数据被称为运算数。

运算符 `+`、`-`、`*`、`/` 和 `**` 实现加法、减法、乘法、除法和指数运算，如下面的示例：

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

在一些其他的编程语言中，`^` 被用做指数运算，但是在 Python 中这是一个位操作，称为 XOR。在本书中将不会涉及位操作的知识，你可以在 wiki.python.org/moin/BitwiseOperators 中阅读相关知识。

除法运算符有可能不按照你期望的运行：

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

`minute` 的值是 59，在传统的算术中，59 除以 60 的结果是 0.98333，而不是 0。产生偏差的原因是因为 Python 执行了下取整除法²。

当两个运算数都是整数，结果同样是整数；下取整除法舍去了小数部分，因此在这个例子中结果为 0。

如果任意一个运算数是一个浮点数，Python 执行浮点除法，此时结果为浮点数：

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

²在 Python 3.0 中，这个除法的结果是一个浮点数。新的运算符 `//` 执行取整除法。

2.6 表达式

表达式是数值、变量和运算符的组合。一个数值是一个表达式，一个变量也是一个表达式，因此一下都是合法的表达式（假设变量 `x` 已经被赋值）：

```
17
x
x + 17
```

如果你在交互模式下输入一个表达式，解释器立即对它赋值并显示结果：

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

但在脚本中，一个表达式本身不做任何事情！初学者通常对此会产生困惑。

Exercise 2.2 在交互模式下输入一下表达式，观察结果：

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

现在将这些表达式放在一个脚本文件里并运行。此时输出是什么？修改脚本，在每个表达式前添加 `print` 语句，然后观察输出。

2.7 运算符的优先级

当一个表达式中出现多于一个运算符时，求值的顺序由优先级规则决定。Python 遵从数学运算符的约定。PEMDAS 是一个有效的方法来记忆这些规则：

- **Parentheses**（括号）具有最高的优先级，可以强迫表达式按照指定的顺序求值。由于在括号中的表达式先被求值，`2 * (3-1)` 的结果是 4，`(1+1)**(5-2)` 的结果是 8。你可以用括号来使得表达式更容易阅读，如 `(minute * 100) / 60`，即便这并不改变最终的结果。
- **Exponentiation**（指数运算）有次高的优先级，因此 `2**1+1` 的结果是 3，而不是 4，`3*1**3` 的结果是 3，而不是 27。
- **Multiplication**（乘法）和 **Division**（除法）具有相同的优先级，并优于 **Addition**（加法）和 **Subtraction**（减法），它们同样具有相同的优先级。因此 `6+4/2` 的结果是 8，而不是 5。
- 具有相同优先级的运算符是从左到右进行求值的。因此在表达式 `degrees / 2 * pi` 中，首先计算除法，然后在结果上乘以 `pi`。如果要除以 2π ，你需要使用括号，或者使用 `degrees / 2 / pi`。

2.8 字符串操作

一般情况下，你不能使用算术运算符作用于字符串，即使字符串看起来像数字，因此以下是非法的：

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

运算符 `+` 可以作用于字符串，但未必按照你期望的方式工作：它执行的是级联，即将字符串首尾相连。例如：

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

程序的输出是 `throatwarbler`。

运算符 `*` 同样可以作用于字符串，它的功能是重复字符串。如 `'Spam'*3` 的结果是 `'SpamSpamSpam'`。如果一个运算符是字符串，那么另一个必须是一个整数。

运算符 `+` 和 `*` 在字符串的使用类似加法和乘法，如 `4*3` 相当于 `4+4+4`，因此我们期望 `'Spam'*3` 相当于 `'Spam'+'Spam'+'Spam'`，事实如此。当然，字符串的级联和重复相比整数的加法和乘法还是有很大的区别的。你可以找出一个加法有而字符串级联没有的性质吗？

2.9 注释

当程序变得越来越长，越来越复杂，它们变得更难读懂。正是的变成语言是密集的，通常很难通过阅读一小段代码来指出它是做什么的，或者为这么这么写。

由于这个理由，在程序旁添加标记，使用自然语言来说明这段程序是做什么的是一个良好的想法。这些标记被称为注释，它们由 `#` 符号开头：

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

在这个例子中，注释以整行的方式出现。你也可以在一行的末尾添加注释：

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

`#` 符号后到一行结束中的任何内容都被忽略 --- 它们将不影响程序的运行。

当说明代码中不明显的特点时，注释非常有效。我们可以假设阅读代码的人可以指出代码做了什么；更有必要的是解释为什么这么做。

以下的注释是冗余而无用的：

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

以下的注释包含了代码中没有的有用信息：

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

好的变量名可以减少对注释的依赖，但长的变量名会使得表达式难以阅读，因此这是一个折衷。

2.10 调试

目前位置你遇到的语法错误大多为非法变量名，如关键字 `class` 和 `yield`，或者包含非法字符，如 `odd~job` 和 `US$`。

如果你在一个变量名中加入一个空格，Python 会认为这是两个没有运算符的运算数：

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

对于语法错误，错误消息没有很大的帮助。通常的消息是 `SyntaxError: invalid syntax` 和 `SyntaxError: invalid token`，它们都不包含很多信息量。

运行时错误通常由于“未定义而使用”，即试图使用一个没有赋值的变量。这会在你拼写变量名错误时发生：

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

变量名是大小写敏感的，因此 `LaTeX` 和 `latex` 是不同的。

语义错误通常发生在运算符的顺序上。例如，求值 $\frac{1}{2\pi}$ ，你也许会写成：

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

但是首先执行了除法，因此你会得到 $\pi/2$ ，并不是想要的结果！Python 没有办法知道你写的是什么含义，因此这种情况下你不会得到一个错误消息；你得到的只是错误的结果。

2.11 术语表

数值： 程序操作的基本数据单元，如一个数字或字符串。

类型： 数据的分类。目前为止我们见过的类型有整数 (`int`)，浮点数 (`float`)，和字符串 (`str`)。

整数： 表示所有整数的类型。

浮点数： 表示具有小数部分的数的类型。

字符串： 表示一串字符的数据类型。

变量： 指向一个数值的名称。

语句： 一段代表命令或行为的代码。目前为止我们见过的语句有赋值语句和打印语句。

赋值： 将数值赋值给变量的语句。

状态图： 表示变量和对应数值的图。

关键字： 一些编译器用来对程序进行语法分析的保留单词，你不能使用例如 `if`，`def` 和 `while` 作为变量名。

运算符： 表示一种简单运算的特殊符号，如加法，乘法，或者字符串级联。

运算数： 运算符作用的数值。

下取整除法：两数相除并舍去小数部分的运算。

表达式：表示单一结果的变量、运算符和数值的组合。

求值 通过执行运算符化简表达式得到一个单一结果。

优先级规则 定义运算符和运算数求值顺规的规则。

级联：将两个运算数首尾现连。

注释：程序中提供给程序员（或者任何阅读程序的人）信息，但不影响程序执行的内容。

2.12 练习

Exercise 2.3 假设我们执行了如下的赋值语句：

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

对于下列表达式，写出表达式的数值和数据类型（表达式的数值）。

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

使用 Python 解释器来验证你的结果。

Exercise 2.4 练习使用 Python 解释器作为计算器。

1. 一个半径为 r 的球的体积为 $\frac{4}{3}\pi r^3$ 。一个半径为 5 的球体的体积是多少？提示：392.6 是错误的！
2. 假设一本书的封面价格是 \$24.95，书店提供 40% 的折扣，对第一本书的运输费用为 \$3，往后每增加一本需要 75 分，如果购买 60 本书总共需要多少？
3. 如果我在 6:52 离开我家，用轻松的步伐跑了 1 英里（8:15 每英里），然后保持节奏跑了 3 英里（7:12 每英里），最后用轻松的步伐跑了 1 英里，我什么时候回到家吃早饭？

Chapter 3

函数

在程序设计中，函数是带有函数名的一系列执行计算的语句，当定义一个函数，我们指定一个函数名和一系列的语句。然后，就可以通过函数名调用函数。我们其实已经看到一个函数调用的例子。

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

函数名是类型，小括号里的表达式称作函数的形式参数 (argument). 结果是形参的类型。

通常我们这么说：函数接受一个参数，返回一个结果（叫做返回值）。

3.1 类型转换函数

Python 提供一些内置函数用来把一种类型的值转换成另一类型。

`int` 函数接受一个值，如果可以，就把它转换成整数，否则就会“抱怨”。

```
>>> int('32')
32
>>> int('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` 函数可以把浮点数转换为整数，但是不能向上取整，只能截掉小数部分：

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

`float` 函数把整数和字符串转换成浮点数：

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

1

最后，`str` 把参数转换为字符串：

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

3.2 数学函数

Python 带有一个数学模块，提供了大多数我们熟悉的数学函数。模块是一个包含一系列有联系函数的文件。

在我们使用模块之前，必须导入它。

```
>>>import math
```

这个语句创建了一个叫做 `math` 的模块对象。如果你尝试打印模块对象，你会得到如下的信息：

```
>>>print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

模块对象包含了定义在该模块内的函数和变量。要访问这些函数，你必须指定模块名和函数名，中间用 “.” 分隔。这种格式叫做点记法。

```
>>>ratio = signal_poewr / noise_poewr
>>>decibels = 10 * math.log10(ratio)
```

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

第一个例子计算以 10 为底，信噪比的对数。`math` 模块也提供了 `log` 来计算以 `e` 为底的对数。

第二个例子计算 `radians` 的正弦值。变量名提示 `sin` 和其他的三角函数 (`cos`, `tan` 等等) 接受弧度作为参数。度除以 360，再乘以 2π ，就得到弧度。

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

2

表达式 `math.pi` 从 `math` 模块获得变量 `pi`。变量 `pi` 的值近似等于 π ，精度大约达到 15 个位。

如果了解三角学，你可以拿上面的结果和二分之根号二比较，看看是否相等：

¹在译者的机器上 `float('3.14159')` 的输出为:3.1415899999999999(解释器 Python2.5 和 2.6); 3.14159 (解释器 Python3.1)。

²在译者的机器上输出为:0.70710678118654746

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

```
3
```

3.3 创建

迄今为止，我们已经见到了程序的部分元素 --- 变量，表达式，语句 --- 并没有把他们结合起来，而只是孤立的涉及到。

程序设计语言最强大的一个特色就是能够把小块的程序块结合起来，创建一个程序块。比如，函数的参数可以是任何表达式，包括数学运算符：

```
x = math.sin(degrees / 360 * 2 * math.pi)
```

甚至包括函数调用：

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

可以这么说，可以接受值的地方，几乎都可以放置一个表达式。有一个例外：赋值语句的左面必须是一个变量名。⁴。其他的任何表达式放在左面都会引起语法错误⁵。

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

3.4 编写新的函数

直到现在，我们只是使用 Python 自带的函数，当然，我们也可以编写自己的新函数。函数定义指明了当函数被调用时，新函数的名字和一系列的语句集合。

这儿有个例子：

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

def 是一个关键字，表明这是一个函数定义。函数名是 `print_lyrics`。函数名的命名规则和变量名一样：字母，数字和一些标点符号是合法的，但是首字符不可以是数字。我们也不可以用关键字作为函数的名字，并且也要尽量避免函数和变量使用相同的名字。

函数名后面的空的小括号表明，函数不带有参数。

函数定义的第一行叫做函数头；其余的部分叫做函数体。函数头必须要以一个分号结尾，函数体必须要有缩进。通常，缩进四个空格（参看 Section ??）。函数体可以包含任意数量

³在译者的机器上，两者只能是近似相等，计算机对浮点数的处理都会涉及到精度问题

⁴在 c/c++ 中，叫做左值

⁵我们即将看到这样的异常

的语句。

`print` 语句的字符串是用双引号引起来的。单引号好双引号效果是一样的；大多数的人使用单引号，除了单引号也出现在字符串中。

如果在交互模式下输入函数，解释器输出省略号 (...) 让我们获悉函数定义还没有结束：

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

要结束一个函数定义，必须留有一个空行（在脚本模式下无须若此）。

定义一个函数，也就创建了同名的变量。

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

`print_lyrics` 的值是一个函数对象，类型是 `'function'`。

调用自建的函数和内置函数的语法是相同的：

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

一旦定义了一个函数，就可以在其他函数中使用它。比如，要重复前面的打印的话，我们可以写一个函数，叫 `repeat_lyrics`：

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

再调用之：

```
>>> 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.
```

但是，真实的歌儿可不是这么唱的～～。

3.5 定义和使用

组合一个前面的代码片段，真个的程序看上去是这样的：


```
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()
```

这个程序包含了两个函数定义：`print_lyrics` 和 `repeat_lyrics`。函数定义的执行就像其他语句一样，只是创建的是一个函数对象。函数体里的语句知道函数被调用的时候才执行，函数调用也不产生任何输出。

正如期待的，在使用前，我们必须创建一个函数。还句话说，函数定义在第一次调用前必须被执行。

Exercise 3.1 把上面程序最后以行移到顶部，这样，函数调用在定义前出现。运行程序，看看有什么错误信息输出。

Exercise 3.2 再把函数调用移到底部，并且把 `print_lyrics` 函数定义移到 `repeat_lyrics` 后面。再运行程序，看看有什么输出？

3.6 执行流

为了保证函数在第一次使用前定义，我们必须知道语句的执行顺序，叫做执行流。

执行通常是从程序的第一个语句开始的。语句一次执行一句，自顶向下。

函数调用就像一次执行流的迂回。不是执行下一个语句，执行流跳转到函数体里，执行那里的所有语句，然后再会到上次跳转的地方。

这听起来很简单，但是还记得我们说过函数也是可以调用其他函数的？当执行流在一个函数的中间时，程序可能不得不执行另外一个函数的语句。当执行那个新的函数时，程序可能不得不执行其他另外一个函数！

幸运的是，**Python** 很擅长追踪执行流在哪里，所以，每次函数执行完成时，程序回到调用它的函数的调用点。当到达程序末尾时，执行流终止。

这个“混乱不堪”的叙述的寓意是什么呢？当你读程序的时候，不必要从上至下。有时候，按着执行流读，甚至更有意义。

3.7 形参和实参

我们已经见到一些内置函数需要接受实参。比如，当调用 `math.sin`，我们传递一个数字作为实参。一些函数接受不止一个参数：`math.pow` 接受两个，底和幂：

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

这个函数，把实参赋值给形参 `bruce`。当函数被调用时，打印形参的值两次（无论是什么）。

函数接受任意可被打印的值。

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

适用于内置函数的创建规则同样也适用于用户自定义函数，由此，我们可以使用任何表达式作为 `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
```

实参在函数调用开始前被计算，所以，在例子中，表达式 `'Spam '*4` 和 `math.cos(math.pi)` 只被计算一次。

也可以用一个变量作为实参：

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

我们作为实参传递给函数的变量名（`michael`）和形参（`bruce`）没有任何关系。每个值的名称是无所谓的（在调用函数中），在 `print_twice`，我们把每个人叫做 `bruce`。

3.8 变量和参数是局部的

当在函数中创建一个变量，它是局域，意味着它只存在于函数中。比如：

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

函数接受两个参数，连接它们，打印结果两次。请看下面的实例：

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

当 `cat_twice` 结束的时候，变量 `cat` 就毁灭了。如果我们尝试打印它，我们会得到一个异常：

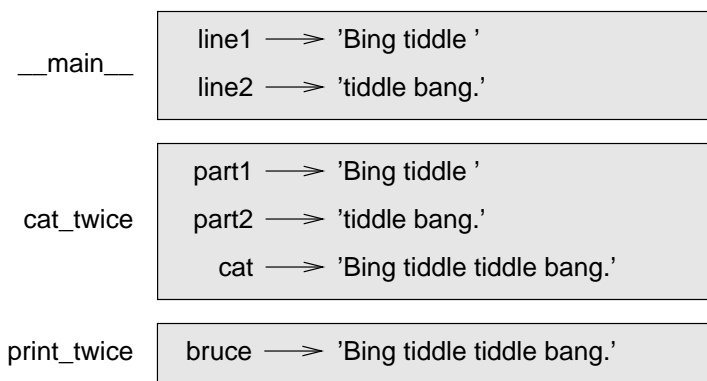
```
>>> print cat
NameError: name 'cat' is not defined
```

形参也是局域的。比如，在 `print_twice` 外，没有 `bruce` 这个东东。

3.9 Stack diagrams 堆栈示意图

为了跟踪哪个变量可以在哪里使用，有时，画堆栈示意图是非常有效的。像状态图一样，堆栈图显示了每个变量的值，同时也显示了变量所属的函数。

每个框图代表一个函数。一个框图就是一个盒子，旁边是函数名，里面是函数形参和变量。我们可以把前面的例子表示成框图：



框图被安排在堆栈中，表明哪个函数调用哪个函数。在这个例子中，`print_twice` 被 `cat_twice` 调用，`cat_twice` 被 `__main__` 调用。

每一个形参指向和他相关联的实参的值。所以，`part1` 和 `line1` 拥有相同的值，`part2` 和 `line2` 拥有相同的值，`bruce` 和 `cat` 拥有相同的值。

如果在函数调用过程中出现错误，Python 打印函数的名称，调用它的函数的函数名，和上一个调用它的函数，一知道 `__main__` 函数。比如，如果尝试在 `print_twice` 函数里访问 `cat`，将会得到一个 `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
```

这样列举函数叫做 **traceback**(追踪)。它告诉我们错误发生在哪个文件，哪行，当时哪个函数在执行。也显示了引起错误的行号。

在 **traceback** 中的函数顺序和在堆栈图中的框图是一样的。正在运行的函数在最底部。

3.10 “结果” 函数和 void 函数

一些我们使用的函数，比如 **math** 函数，产生结果；找不到一个更好的名字，我姑且称他为“结果函数” (**fruitful functions**)。其他函数，像 **print_twice**，实施一个动作，但是没有返回一个值。他们叫做“虚无函数” (**void function**)⁶。

当调用一个“结果函数”时，你大多数情况下希望对返回结果做些什么；比如，你可能把它赋给一个变量或者把它作为一个表达式的一部分：

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

当你在交互模式调用一个函数，Python 显示如下结果：

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

但在脚本中，如果，调用仅仅调用一个“结果函数”，返回值就永远的丢失了！

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

脚本计算 5 的平方根，但是因为它不存储或者显示结果，所以没有多大的价值⁷。

虚无函数 (**Void 函数**) 在屏幕上显示某些东西，或者产生其他的效果，但是他们没有返回值。如果尝试着把结果赋给一个变量，我们将会得到一个特殊的值 **None**。

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

None 的值和字符串 '**None**' 是截然不同的概念。它有一个特殊的类型：

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

截至目前，我们写的函数都是“虚无函数”，再过几章，我们就开始编写“结构函数”。

⁶译注：这些只是作者自己的命名

⁷译注：可以作为一条语句执行，这就是它的价值

3.11 为什么要函数

可能大家还不是很清楚，为什么我们要花大力气去把程序分割成函数。有以下几个原因：

- 创建一个新函数，让我们有机会去给一组语句命名，这使得程序易读和易调试。
- 通过消除重复的代码，函数可以使程序变得精巧。以后，如果想做个改动，只需要改变一个地方。
- 把一个规模庞大的程序分解成函数使得我们一次调试一个地方，然后把他们组合起来，成为一个可工作的整体。
- 设计良好的函数在很多程序中都会有用武之地。一旦写了一个，调试无误，就可以重用⁸。

3.12 调试

如果使用编辑器编辑脚本，可能会遇到空格符和制表符的问题。避免这些问题最好的方式就是只使用空格（不用制表）。大多数的编辑器（对 Python 支持）默认使用这个方式，但是有一些不是。

制表符和空格符一般都是不可见的，这使得他们很难去调试，所以，最好使用能够自动为你产生缩进的编辑器⁹。

另外，别忘了在运行程序之前，保存它。有些开发环境自动保存，但是有些不¹⁰。如果没有保存，你在编辑器里看到的程序和你运行的可能不是一样的。

一定要确保你看到的代码就是你要运行的代码。如果不确定的话，在程序开始加入一些语句，比如 `print 'hello'`，重新运行之。如果你没有看到 `hello`，你运行的就不是你想要的程序。

3.13 术语表

function 函数： 带有名字的，执行有意义的操作的语句集合。函数可以接受也可以不接受参数和产生结果。

function definition 函数定义： 一个创建新函数的语句，指明了函数名，参数，和执行的语句。

function object 函数对象： 由函数定义常见的值。函数名就是一个指向函数对象的变量。

header 函数头： 函数定义的第一行。

body 函数体： 函数定义里面的一系列语句。

parameter 形参： 函数里指向传递过来的实参值的一个名字。

⁸译注：reuse，有的书上建议翻译成复用

⁹译注：译者最喜欢的编辑器 Vim 和 Geany 对 Python 的支持都是非常好的

¹⁰译注：一般，编辑器都是可以设置成自动保存的

function call 函数调用： 一条执行函数的语句。由函数名和一个参数列表组成。

argument 实参： 函数被调用时，提供给函数的值。这个值被赋给函数里相应的形参。

local variable 局域变领： 定义在函数里的变量。只能在函数体里使用。

return value 返回值： 函数的结果。如果函数调用被用作表达式，返回值是表达式的值。

fruitful function 结果函数： 有返回值的函数。

void function 虚无函数： 没有返回值的函数。

module 模块： 包含一系列函数和其他定义的文件。

import statement **import** 语句： 读取模块文件，创建模块对象的语句。

module object 模块对象： 由 **import** 语句创建的值，提供了访问定义在模块里值的能力。

dot notation 点记法： 调用另外一个模跨函数的语法，具体的是模块名后加一个点，让后是函数名。

composition 创建： 用一个表达式作为更大表达式的一部分，或者一个语句作为更大语句的一部分。

flow of execution 执行流： 程序运行期间，语句的执行顺序。

stack diagram 堆栈图： 函数，及其变量和制的图式化堆栈表示。

frame 框图： 堆栈图中的盒子，用来表示一个函数调用。包含了函数的局部变量和形式参数。

traceback: 异常发生时，打印的正在执行的函数表。

3.14 练习

Exercise 3.3 Python 提供了内置函数 **len**，返回字符串的长度，**len('allen')** 的值是 5。

编写一个名为**right_justify** 的函数，接受一个名为 **s** 的参数，打印该字符串，使得打印的字符串的最后一个字符在第 70 列。

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

Exercise 3.4 函数对象是一个值，所以你可以把它赋给一个变量，或者把它作为参数传递。比如，**do_twice** 函数接受一个函数对象作为参数，然后调用两次：

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

这里是另外一个例子，使用**do_twice** 调用函数**print_spam** 两次。

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

do_twice(print_spam)
```

1. 把上面的例子输入脚本，测试一下。
2. 修改`do_twice`函数，让它接受两个参数 --- 一个函数对象和一个普通的值，调用函数两次，把普通值作为实参。
3. 编写一个更一般版本的`print_spam`，调用`print_twice`两次，传递`'spam'`作为实参。
4. 定义一个新的函数`do_four`，就手一个函数低相和一个普通值，调用函数四次，传递普通值给函数，作为参数。函数体里只能有两个语句，不能有四个。

你可以参考我提供的答案thinkpython.com/code/do_four.py。

Exercise 3.5 这个练习¹¹可以只用我们迄今为止学过的语句和其他语言特点实现。

1. 编写一个函数，打印如下的网格：

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

提示：要在一行打印多个值，可以打印一个逗号分隔的语句序列：

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

如果序列以逗号结尾，Python 会认为此句没有结束，所以打印的值在同一行。

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

语句的输出是`'+-'`。

`print` 语句自动结束本行，进入下一行。

2. 是哦嗯先前的函数打印类似的网格，要求 4 行 4 列。

可以参考我的答案：thinkpython.com/code/grid.py。

¹¹基于 Oualline 的一个练习，Practical C Programming, Third Edition, O'Reilly(1997)

Chapter 4

实例学习：接口设计

4.1 TurtleWorld

为了完成这本书，我写了一个叫做 **Swamp** 的模块。其中的一个就是 **TurtleWorld**，它提供了一系列在屏幕上绘制移动的乌龟的函数。

你可以从 thinkpython.com/swampy 下载 **Swamp**，然后根据说明将 **Swampy** 安装到你的系统上。

打开含有 **TurtleWorld.py** 的文件夹，创建一个 **polygon.py** 的文件，然后输入以下代码：

```
from TurtleWorld import *
```

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

```
wait_for_user()
```

第一行是我们先前见过的 **import** 语句的变种；它直接从模块中导入函数，而不是创建一个模型模块对象，这样你可以直接访问函数，而不需要点符号。

接下来的几行创建了一个 **TurtleWorld** 并赋值给 **world**，创建了一个 **Turtle** 并赋值给 **bob**，然后打印 **bob**：

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

输出表明 **bob** 指向一个在模块 **TurtleWorld** 中定义的 **Turtle** 的实例。在这里，“实例”指一个集合中的一个成员；这个 **Turtle** 是集合中可能存在的 **Turtles** 中的一个。

wait_for_user 告诉 **TurtleWorld** 等待用户进行某些操作，在这个例子中，用户除了关闭窗口没有其他的操作。

TurtleWorld 提供了多个移动乌龟的函数：**fd** 和 **bk** 用来前进和后退，**lt** 和 **rt** 用来左转和右转。此外，每只乌龟携带了一支笔，笔或者提起或者放下；如果笔是放下的，乌龟将在它经过的地方留下轨迹。函数 **pu** 和 **pd** 代表“提起笔”和“放下笔”。

要画一个直角，下程序中添加以下代码（在创建 **bob** 之后，调用 **wait_for_user** 之前）：

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

第一行告诉 bob 向前移动 100 步。第二行告诉它左转。

当你运行这个程序是，你可以看见 bob 先向东移动，然后向南移动，并留下两条轨迹。

现在修改程序，绘制一个正方形。在未完成之前请不要继续下去！

4.2 简单重复

你肯能会写出类似这样的代码（忽略创建 TurtleWorld 和等待用户的）：

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

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

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

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

我们可以使用 for 语句使得程序更简洁。在 polygon.py 中添加如下代码然后运行：

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

你会看到这样的结果：

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

这是 for 语句最简单的用法；以后我们会看到更多的例子。这些已经足够帮助你重写画正方形的程序了。请在完成后继续阅读。

这里给出使用 for 语句画正方形的代码：

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

for 语句的语法类似一个函数的定义。它有一个以冒号结尾的首部以及缩进过的主体部分。主题部分可以包含任意多的语句。

for 常常被称作循环，因为程序执行时穿过主体部分，然后循环回到顶部。在这个例子中，主体部分循环了 4 次。

这个版本的程序事实上和先前的画正方形的程序有所不同，因为在画最后一条边后多了一次左转。额外的一次旋转仅仅花费很少的时间，但是代码得以简化，如果我们每次都重复循环中的操作。这个版本还让乌龟回到起始位置的同时朝向初始的方向。

4.3 练习

以下是一系列使用 TurtleWorld 的练习。它们被设计的更有趣，同时含有关键点。当你在练习时，思考关键点是什么。

接下来的章节有练习的解答，所以在你完成前（至少尝试前）不要去翻阅它们。

1. 写一个叫做 `square` 的函数，读取一个叫做 `t` 的参数，参数是一个乌龟。函数使用这个乌龟画一个正方形。
写一个函数调用将 `bob` 作为参数传给 `square`，然后运行程序。
2. 在 `square` 中新增一个叫做 `length` 的参数。修改程序主体使得边长等于 `length`，然后修改函数调用，增加第二个参数。运行程序。使用一系列的 `length` 测试你的程序。
3. 函数 `lt` 和 `rt` 默认转 90 度，但是你可以提供第二个参数来指定旋转的角度。例如，`lt(bob, 45)` 令 `bob` 向左转 45 度。
复制 `square` 并命名为 `polygon`。添加一个新的参数 `n`，并修改函数使得函数绘制一个 `n` 边正多边形。提示：一个 `n` 边正多边形的外角为 $360.0/n$ 度。
4. 写一个 `circle` 函数，输入为一个乌龟 `t` 和半径 `r` 作为参数，使用的边长和边数调用 `polygon` 函数画一个多边形来近似圆。使用一系列 `r` 来测试你的程序。
提示：计算出圆的周长，使得 `length * n = circumference`。
额外提示：如果你觉得 `bob` 移动得太慢，你可以设置 `bob.delay`，它决定了两次移动的时间间隔。`bob.delay = 0.01` 应该会让它看起来在运动。
5. `circle` 的一个更一般化的版本是 `arc`，它接收一个额外的参数 `angle`，指定了绘制圆周的多少部分。`angle` 使用角度作为单位，因此当 `angle=360` 时，`arc` 将会绘制一个完整的圆周。

4.4 封装

第一个练习要求你将画正方形的代码装入一个函数定义，然后传递一个乌龟作为参数，调用该函数。如下所示：

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

```
square(bob)
```

最深处的代码 `fd` 和 `lt` 被缩进两次，因此它们处在函数定义中的 `for` 循环中。在接下来的行中，`square(bob)` 左对齐到初始位置，代表着 `for` 循环和函数定义的结束。

在函数中，`t` 和 `bob` 指向同一个顾伟，因此 `lt(t)` 和 `lt(bob)` 具有相同的效果。那么为什么不直接使用参数 `bob` 呢？主要是考虑到 `t` 可以被用来代表任意的乌龟，不仅仅是 `bob`，你可以创建一个新的乌龟，然后把它作为参数传给 `square`：

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

将一段代码打包在一个函数的过程叫做封装。封装的一个好处是给代码附上一个名称，起着文档的作用。另一个好处是你可以重用代码，调用一个函数比复制黏贴代码更明智！

4.5 泛化

下一步是在 `square` 函数中添加 `length` 参数，如下所示：

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

```
square(bob, 100)
```

给一个函数添加一个变量称为泛化，因为它使得函数使用更广泛的情况：在先前的版本中，画的正方形总是相同的大小，在这个版本中可以是任意大小。

下一步仍是泛化。`polygon` 可以画一个任意边数的正多边形，不仅仅是正方形。如下所示：

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

```
polygon(bob, 7, 70)
```

这段代码花了一个边长为 70 的正七变形。如果函数有较多的参数，人们容易忘记它们分别是什么，或者应该以什么顺序给出。因此将参数名包含在参数列表中是合法的，有事也很有用：

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

这些包括变量名作为“参数”，被称作关键字参数（不要和 Python 关键字，如 `while` 或 `def` 混淆）。

这样的语法使得程序的可读性增强。同时提醒数值和参数是如何工作的：当你调用一个函数，数值将被赋值给参数。

4.6 接口设计

下一步是写函数 `circle`，它读取半径 `r` 作为参数。下面给出一个示例，通过调用 `polygon` 画正 50 边形：

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

第一行使用公式 $2\pi r$ 计算半径为 `r` 的圆的周长。由于使用了 `math.pi`，我们需要包含 `math`。通常 `import` 语句放在脚本的开头。

`n` 是用来近似圆的多边形的边数，`length` 是每段边长的长度。因此 `polygon` 画一个边长为 `r` 的 50 边形来近似一个圆。

这个方法有个限制，由于 `n` 是一个常数，当圆非常大的时候，每条边长将变得很长。而对于小的圆，画这些小的线段又浪费了太多的时间。一个解决方法是将函数泛化，增加一个参数 `n`。这样给用户（调用 `circle` 的人）更多的控制，但是函数接口就显得复杂了。

一个函数的接口 是函数如何使用的概述：参数是什么？函数做了什么？返回值是什么？如果一个接口“尽可能的简洁，但不简单（Einstein）”，我们称这个接口是“干净的”。

在这个例子中，`r` 属于接口，因为它决定了所画圆的大小。`n` 相对而言不适合作为接口，因此它属于函数如何画圆的细节。

与其将接口弄乱，根据周长选择一个合适的 `n` 是一个更好的方法。

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

现在多边形的边数大约是周长/3，因此每条边的长度大约为 3，即使圆足够好看，又保证绘画的效率，因此适合所有尺寸的圆。

4.7 重构

当我写 `circle` 时，我可以重用 `polygon` 的代码，因为多边形对应和许多代码和圆的代码很相似。但是 `arc` 并没有那么相似，我们不能使用 `polygon` 或者 `circle` 来画一段圆弧。

一个解决方法是复制一个 `polygon` 拷贝，并将其改为 `arc`，结果看起来应该如下：

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

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

函数代码的第二部分和 `polygon` 非常相似。但是如果不修改接口，我们不能重用 `polygon` 的代码。我们可以泛化 `polygon`，让它读入一个角度作为第三个参数，但此时已经不适合用 `polygon` 作为函数名了！因此我们称这个泛化的函数为 `polyline`：

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

现在我们使用 `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，我们使用 `arc` 重写 `circle`：

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

这个重新安排程序以优化程序接口改善程序复用的过程被称为重构。在本例中，我们注意到 `arc` 和 `polygon` 代码中有许多类似的部分，于是我们将这部分“提取”并作为 `polyline`。

如果我们在一开始的时候设计过，我们会首先写 `polyline` 来避免重构，但通常在工程开始的时候你不肯能了解所有的细节并设计所有的接口。当你开始编程后，你对工程会有更深入的了解。有时候重构是你学到某些东西的标志。

4.8 一个开发者的计划

开发计划是写程序的一个过程。在本例中我们使用的过程是“封装和泛化”。这个过程包括以下步骤：

1. 开始阶段写一个小的没有函数定义的程序。
2. 当程序可以工作后，将其封装成函数，并赋予函数名。
3. 通过适当添加函数参数泛化函数。
4. 重复步骤 1—3，直到你有了一系列可以工作的函数。复制黏贴代码以避免重复输入（和重复调试）。
5. 寻找机会通过重构改善代码。例如，你在不同地方使用了相似的代码，可以考虑将它们重构到一个更一般化的函数中。

这个过程包含一些缺点，我们会在以后介绍改进方法，当当你在一开始不知道如何将程序划分为函数时这是一个有效的方法，让你可以继续你的设计。

4.9 文档字符串

文档字符串是在函数头部解释接口的字符串（“doc”是“document”的缩写），以下是一个例子：

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

文档字符串是一个用引号三次包含的多行字符串，三重引号允许字符串跨越多行。

文档字符串虽然简洁，但对于需要使用这个函数的人来说包含了必要的信息。它准确的解释了函数做了什么（而不涉及是怎么做的）。它解释了每个参数对函数的影响以及每个参数的数据类型（如果不是很明显）。

这类文档的撰写时接口设计中一个重要的部分。一个良好设计的接口可以用简洁的语言来解释。如果你在解释某个函数时遇到了困难，也许就是接口有待改进的标记。

4.10 调试

接口好比函数和调用者间的一个协议。调用者同意提供一定的参数，函数同意完成一定的工作。

例如，`polyline` 需要四个参数。第一个参数是一个乌龟。第二个参数是一个数，理论上这应该是一个正数，事实上即使不是函数也能正常工作。第三个参数是一个整数，否则 `range` 会报错（报错信息取决于你运行的 Python 版本）。第四个参数是一个数，决定绘图的角度。

这写要求被称作先决条件，在程序执行前首先需要满足这些条件。相对的，在函数尾部的条件被称为后决条件。后决条件包括函数需要的功能（如画线段）以及附带的效果（如移动乌龟和对环境的其他操作）。

调用者需要对先决条件负责。如果调用者违反了（合理设计的）先决条件导致函数工作异常，这个错误是调用者的，而不是函数的。

4.11 术语表

实例： 一个集合中的一员。本章中 `TurtleWorld` 是 `TurtleWorlds` 集合中的一员。

循环： 程序中可以重复执行的部分。

封装： 将一系列语句包装成一个函数的过程。

泛化： 将某些无需确定的（如一个数字）东西用一些通用的东西（如变量或参数）代替的过程。

关键字参数： 将“关键字”作为名字的参数。

接口： 如何使用一个函数的描述，包括参数名、参数描述和返回值。

开发计划： 一个编写程序的过程。

文档字符串： 在函数定义处的字符串，用来给出函数接口的文档。

先决条件： 调用者在调用函数前需要满足的条件。

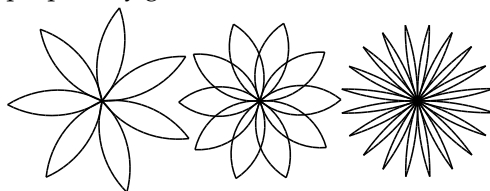
后决条件： 函数在返回前需要满足的条件。

4.12 练习

Exercise 4.1 下载本章的代码 thinkpython.com/code/polygon.py.

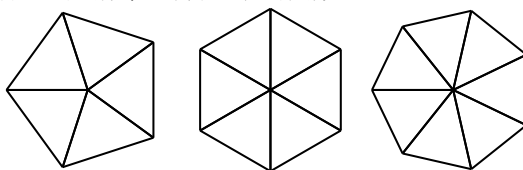
1. 给 `polygon`, `arc` 和 `circle` 编写合适的文档字符串。
2. 绘制栈图, 给出 `circle(bob, radius)` 执行时程序的状态。你可以手工计算其中的算术, 或者添加 `print` 语句。
3. 章节??中的 `arc` 的版本不是非常精确, 近似的直线总是在实际圆的外围, 因此乌龟最终停止的位置偏离正确的位置。我的程序给出了一个减小误差的方法。阅读代码并理解。如果你尝试画图, 你可以看到它是如何工作的。

Exercise 4.2 Write an appropriately general set of functions that can draw flowers like this:



You can download a solution from thinkpython.com/code/flower.py.

Exercise 4.3 写一些列合适的函数, 绘制以下的图形:



你可以从这里下载一个解决方案 thinkpython.com/code/pie.py。

Exercise 4.4 字母可以由一些基本元素组成,, 如垂直或水平的线和曲线。使用最少的基本元素设计一种字体, 并调用函数绘制字母。

你可以为每个字母写一个函数, 命名为`draw_a`, `draw_b` 等, 然后把你的函数放在 `letters.py` 的文件里。你可以从thinkpython.com/code/typewriter.py 下载一个“乌龟打字机”来帮助你测试你的代码。

你可以从这里下载一个解决方案 thinkpython.com/code/letters.py。

Chapter 5

条件语句和递归

5.1 模操作符

模操作符用于两个整数，第一个操作数除以第二个操作数产生余数。在 Python 中，模操作符是一个百分号 (%)。语法的格式和其他的操作符相同。

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

7 除以 3 等于 2 余 1。

模操作符是非常有用的，比如，你可以查看一个数是否可以被另一个数整除 --- 如果 `x % y` 是 0，`x` 就可以被 `y` 整除。

你也可以用模运算来提取整数的最右边的数字。比如，`x % 10` 得到 `x` 的最右面的一个数字¹（以十为底）。类似地，`x % 100` 得到最后的两位数字²。

5.2 布尔表达式

布尔表达式的结果要么是真 (`true`)，要么为假 (`false`)。下面的例子是使用 `==` 运算符，比较两个操作数，如果相等则结果为 `True`，否则为 `False`：

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

¹译注：个位数

²十位和个位的数字

`True` 和 `False` 是两个特殊的值，属于 `bool` 类型；他们不是字符串：

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

`==` 运算符是关系运算符中的一个，其他的还有：

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

尽管你可能很熟悉这些运算符，他们在 `Python` 中的表示方法和数学中的有很大的不同。一个常见的错误是只使用一个 `=` 号，而不是两个 `==` 号。记住 `=` 是赋值操作符，`==` 是关系运算符。而且，`Python` 中没有这样的符号 `=<` 或者 `=>`³。

5.3 逻辑运算符

有三个逻辑运算符：`and`、`or` 和 `not`。这些操作符的意思和在英语中的意思差不多。比如，`x > 0 and x < 10` 为真，仅当 `x` 大于 0 小于 10⁴。

如果 `n % 2 == 0 or n % 3 == 0` 有一个条件语句为真，则表达式的值就为真，亦即 `n` 可以被 2 或 3 整除。

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

严格来说，逻辑运算符的操作数只能是布尔表达式，但是 `Python` 对此可没什么严格要求。任何不为 0 的整数也被解释成 `True`⁵

```
>>> 17 and True
True
```

这个灵活性是很有用处的，但是可能会产生一些微妙的问题。我们要尽可能的避免他们（除非知道自己在做什么）。

5.4 条件执行

考虑到要写一些有用的程序，我们几乎总是需要检查条件，并改变相应的改变程序的行为。条件语句给了我们这个能力。最简单的要属 `if` 语句了：

³在 FP(functional programming) 中可能会遇到这个符号

⁴译注：在 `Python` 中，更 `pythonic` 的写法是 `0 < x < 10`。这样的符号对于 `c/c++` 背景的程序员来说，有点陌生，在 `c/c++` 等值的分别是 `&&`、`||`、`!`。

⁵这个也可扩展到任何其他类型，比如后面要涉及到的 `list`、`tuple`、`dict`、`set` 还有 `str`。

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

`if` 语句后面的布尔表达式叫做条件。如果条件为真，则下面缩进的语句就被执行。反之，则什么也不发生。⁶

`if` 语句和函数定义有着相同的结构：一个头，后面跟着一个缩进的语句块。这样的语句叫做复合语句。

虽然对复合语句里面可以含有的语句数量不限，但是必须至少有一条⁷。偶然地，可能在语句体里暂时不需要语句（通常作为一个占位符）。在这种情况下，我们可以使用 `pass` 语句，它什么也不做。

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

5.5 选择执行

`if` 语句的第二种形式是选择执行，此时，有两种可能性，条件决定了哪一个可能性被执行。语法是这样：

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

如果 `x` 除以 2 的余数是 0，我们可以判定 `x` 是偶数，程序就输出这个效果。如果条件为假，第二个语句就被执行。因为条件必须为真或假，其中一个必定会被执行。选择项叫做分支，因为它们是执行流的分支。

5.6 链式条件

有时，可能会有不止两种可能性，我们就需要更多的分支。一种方式是使用链式条件语句。

```
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”的缩写形式。再次说明一下，只有一条语句被执行。`elif` 语句的数目也是没有限制的。如果要写 `else` 语句，必须是在链式条件的最后，但是如果没有，也是允许的。

⁶这是针对本例而言，因为本例只有一条语句。在其他的情况下，可能会有诸如 `else` 之类的语句。

⁷C/C++ 中没有这样的限制

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

每个条件按顺序被检查。如果第一个为假，下一个就被检查，如此如此。如果有一个为真，相应的分支就被执行，链式语句也就终止。尽管可能有多个条件为真，也只有第一个为真的分支被执行。

5.7 嵌套的条件语句

一条条件语句也可以嵌套在另一个语句之中。我们写一个典型的例子：

```
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'
```

外层的条件包两个分支。第一个分支包含一个简单语句。第二个分支包含例外一个 `if` 语句，同时，这个 `if` 语句也有两个分支。这两个简单的分支都是简单语句，尽管他们本也可能是条件语句。

尽管缩进使得代码结构清晰，嵌套语句还是难以快速的理解。一般来说，尽可能的避免使用嵌套条件语句。

逻辑运算符可以简化嵌套条件语句。比如，下面的代码可以只用一条条件语句：

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

只有当我们“通过”了两个条件时，`print` 语句才会被执行，所以，我们可以用 `and` 运算符达到同样的效果。

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

5.8 递归

函数调用⁸另外一个函数是合法的；函数调用他自身也是合法的。很难一眼看出这样做有什么好处⁹，但实践证明，这是程序能做的最具有魔力的事情之一。比如，看下面的函数：

⁸译注：台湾的书籍一般翻译为呼叫，这个很形象

⁹译者注：我猜，这种情况就像是自己把自己提起来一样～～

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

如果 n 是非正数，程序输出，“Blastoff! ”，否则，输出 n ，然后调用 `countdown` 函数 ---也就是它自己 ---同时把 $n-1$ 当作参数传递给它。

如果我们调用这个函数，究竟发生了什么？

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

`countdown` 从 $n=3$ 开始执行， n 此时大于 0，于是输出 3，接着调用自身。。。。。。

`countdown` 从 $n=2$ 开始执行， n 此时大于 0，于是输出 2，接着调用自身。。。。。。

`countdown` 从 $n=1$ 开始执行， n 此时大于 0，于是输出 1，接着调用自身。。。。。。

`countdown` 从 $n=0$ 开始执行， n 此时不大于 0，输出 “Blastoff!” 然后返回。

接受 $n=1$ 的 `countdown` 返回。

接受 $n=2$ 的 `countdown` 返回。

`countdown` 接受 $n=3$ 的函数返回。

然后，我们就会到 `__main__` 里了。整个输出如下：

```
3
2
1
Blastoff!
```

调用自身的函数称作递归函数；调用的过程叫做递归。

另外一个例子，我们写一个打印一个字符串 n 次的函数。

```
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。执行流立刻返回到调用者，剩余的部分就不再被执行了。

函数的剩余部分和 `countdown` 还书相似：如果 n 大于 0，输出 s ，然后调用自身显示 s $n-1$ 次。所以，输出的行数是 $1 + (n-1)$ ，也就是 n 次。

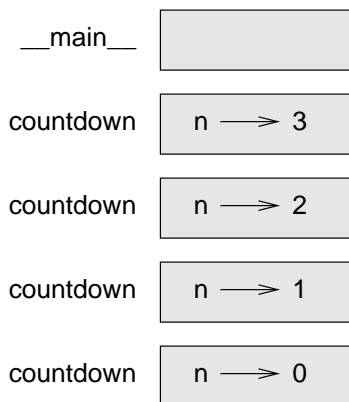
这样简单的例子，其实可以很容易用一个 `for` 循环来实现。但我们以后将会看到很难写成 `for` 循环形式，但是很容易用递归实现的例子，我们现在就开始认识递归是有好处的。

5.9 递归函数的堆栈图

在??部分，我们使用堆栈图代表程序在函数调用过程中的状态。同样的方法也可以帮助我们理解递归函数。

每次函数调用的时候，Python 创建一个新的函数框图，里面包含了函数的局部变量和参数。对于递归函数来说，可能会有不止一个框图同时出现在堆栈图里。

下图显示了 `n = 3` 时 `countdown` 函数的堆栈图。



像通常的一样，栈顶是 `__main__` 的卡框图。由于我们没有在 `__main__` 里创建任何的变量或传递任何的参数给它，所以它是空的。

四个 `countdown` 框图拥有不同的 `n` 值。栈底，`n = 0`，叫做终止条件 (`base case`)。不执行任何的递归调用，所以就没有更多的框图了。

画 `print_n` 的堆栈图，其中，`s = 'Hello'` `n = 2`。

编写 `do_n` 函数，接受一个函数对象，和一个数字，`n` 作为参数，调用传递过来的函数 `n` 次。

5.10 无穷递归

如果递归函数没有终止条件，就会无止境的递归¹⁰。当达到最大递归深度时，Python 就会报告错误信息。

```

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
  
```

这个追踪比我们上一章看到的要大很多。当错误产生时，在栈中有 1000 张 `recurse` 框图！

¹⁰译者：直到消耗完资源，或者操作系统终止它

5.11 键盘输入

迄今为止，我们编写的程序对用户来说有点“不礼貌”---不接受来自用户的输入。每次只是做同样的事。

Python 提供了一个内置的函数`raw_input` 获取用户的输入¹¹ 当`raw_input` 函数被调用时，程序停下来等待用户输入些东西。当用户敲击 `Return` 或者 `Enter`，程序恢复运行，`raw_input` 把用户输入的东西作为字符串返回。

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

在用户输入之前，最好能够输出一个提示，告诉用户输入什么。`raw_input` 可以接受一个提示作为参数。

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

提示后面的`n` 代表一个换行，这是一个特殊字符，产生一个断行。这也是为什么用户的输入出现在提示下面的原因。

如果希望用户输入一个整数，可以尝试把返回值转换为 `int`。

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

但是，如果用户输入的不是数字组成的字符串，就会得到一个错误：

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

我们以后将会看到如何处理这样的错误。

5.12 Debugging 调试

当错误发生，Python 输出的跟踪信息，包含大量的信息，但是也很容易让人“眼花缭乱”，特别是栈中有很多框图的时候。最有用的部分通常是：

¹¹在 Python3.0 中，这个函数叫做 `input`。译注：在 Python3.x 中都是如此

- 什么类型的错误
- 在什么地方发生的

通常，语法错误很容易发现，但是也有些微妙的东西¹²。“空格”错误可能很微妙，因为空格和制表是不可见的，而且我们习惯上忽略它们¹³。

```
>>> x = 5
>>> y = 6
      File "<stdin>", line 1
        y = 6
        ^
```

SyntaxError: invalid syntax

这个例子中，问题发生在第二行有了一个空格缩进。但是错误指向了 `y`，这个是一个误导。一般，错误信息提示问题发生的地方，当时实际的错误可能在提示的前面一点，有时在前一行。

对于运行时错误，也是如此。假设，你想用分贝计算信噪比。公式是 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$ 。你可能写出如下的 Python 代码：

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

但是，当你运行时，你会得到一个错误信息¹⁴。

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

错误信息提示错误发生在第五行，但是那行根本没有错误。为了发掘出真正的错误，打印 `ratio` 的值可能会有帮助，结果显示是 `0`。问题出现在第四行，因为两个整数的除法实施的是地板除（floor division）¹⁵。解决方案是用浮点数来表示信号功率和噪声功率。

总的来书，错误信息告诉我们出现问题的地方，但通常不是问题发生的根本所在。

5.13 术语表

modulus operator 模运算符： 一个操作符，记法为百分号（%）。作用在两个整数之间，产生余数。

boolean expression 布尔表达式： 值要么为 `True` 要么为 `False` 的表达式。

relational operator 关系运算符： 比较操作数的的运算符：`==`，`t != , > , < , >=` 和 `<=`。

¹²原文是:but there are a few gotchas

¹³特别是在不同的机器上，或者不同的工具上大开源码的时候

¹⁴在 Python3.0 中，不会得到错误信息；除法运算符执行浮点除，尽管操作数是整数，这个和实际很贴近

¹⁵译注：Python 核心编程的中译本中，把它翻译成地板除

logical operator 逻辑运算符： 连接布尔表达式的运算符： `and`, `or` 和 `not`。

conditional statement 条件语句： 依靠一些条件控制执行流的语句。

condition 条件： 条件语句中的布尔表达式， 决定分支的执行。

compound statement 复合语句： 包含头和体的语句。头一 `(:)` 结尾， 体依据头， 缩进。

body 体： 符合语句中的一系列语句。

branch 分支： 条件语句中的可选择执行的语句（序列）。

chained conditional 链式条件语句： 拥有一系列的选择分支的条件语句。

nested conditional 嵌套条件语句： 出现在条件语句中分支中的条件语句。

recursion 递归： 调用函数自身的过程。

base case 终止条件： 递归函数里的一个条件分支， 终止递归调用。

infinite recursion 无穷递归： 没有终止条件的递归， 最终无穷递归产生一个运行时错误。

5.14 练习

Exercise 5.1 费马最后定理这么表述： 不存在这样的整数 a, b 和 c 使得对于 n 大于 2,

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

。

1. 编写 `check_fermat` 函数， 接受 4 个参数 --- a, b, c 和 n ---验证费马定理是否正确。 如果 n 大于 2,

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

是成立的， 程序输出， ``Holy smokes, Fermat was wrong!``, 否则， 输出， ``No, that doesn't work.``

2. 编写一个函数提示用户输入 a, b, c and n 的值， 把他们转换成整数， 使用 `check_fermat` 验证是否违背费马定理。

Exercise 5.2 给你三根木棒， 你也许能， 也许不能组成一个三角形。 比如， 其中一根木棒 12 英尺长， 其他的两根是 1 英尺长， 很明显， 不能使短木棒在长木棒的中间相遇。 对于三个任意长度， 可以用一个简单的测试在检测是否能够构成一个三角形。

“如果三个长度的任意一个大于其他两个之和， 就不能构成一个三角形。 否则， 就可以¹⁶”

1. 编写 `is_triangle` 函数， 接受 3 个整数作为参数， 依据能不能构成三角形， 输出要么是 “Yes”， 要么是 “or”。
2. 编写一个函数提示用户输入木棒的长度， 转换成整数， 然后调用 `is_triangle` 检测是否可以构成一个三角形。

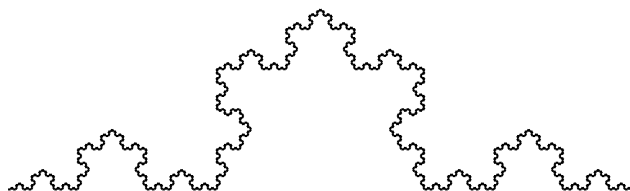
¹⁶如果两个长度之和等于第三个， 他们形成退化的三角形。

接下来的练习使用 4 章节的 `TurtleWorld`。

Exercise 5.3 阅读下面的函数，看看它的功能是什么。然后运行它（查看 4 章节的例子）。

```
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)
```

Exercise 5.4 柯霍曲线是一个分形体，看起来像这样：



画长度为 x 的柯霍曲线，你所需要做得就是

1. 画一个长度为 $x/3$ 的柯霍曲线
2. 左转 60 度。
3. 画一个长度为 $x/3$ 的柯霍曲线
4. 右转 120 度
5. 画一个长度为 $x/3$ 的柯霍曲线
6. 左转 60 度。
7. 画一个长度为 $x/3$ 的柯霍曲线

唯一的例外是如果 x 小于 3，此时，就直接画一个长度为 x 的直线。

1. 编写 `koch` 函数，接受一个 `turtle` 和长度作为参数，使用 `turtle` 画一个给定长度的柯霍曲线。
2. 编写 `snowflake` 函数，画 3 个柯霍曲线，形成雪花的轮廓。
可以查看我的答案 thinkpython.com/code/koch.py。
3. 柯霍曲线可以用多种方式一般化。查看 wikipedia.org/wiki/Koch_snowflake 例子，并实现自己喜欢的雪花。

Chapter 6

“结果” 函数

6.1 返回值

我们使用的一些内建函数，如数学函数，会返回结果。调用这些函数的过程中会产生一个数值，通常我们将它赋值给一个变量，或者作为表达式的一部分。

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

目前为止我们所写的函数都是空的：它们打印一些东西或者移动乌龟，但它们的返回值都是空。

在本章中，我们开始编写卓有成效的函数。第一个例子是 `area`，它返回给定半径的圆的面积：

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

我们先前见过 `return` 语句，但在卓有成效的函数中，`return` 语句包含一个表达式。返回语句的含义为：“立即从函数返回，并将表达式的值作为返回值”。表达式可以任意复杂，因此我们可以这样更简洁的写这个函数：

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

同时，临时变量 如 `temp` 通常使得调试更为方便。

有时一个函数会有多个返回语句，每一条位于一个条件分支中：

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

由于 `return` 语句位于一个选择的条件中，只有一条会被执行。

当一条返回语句被执行时，函数立刻返回，而不再执行后面的语句。在 `return` 语句后的代码，或者存在于任何程序流执行不到的地方的代码，称为死区代码。

在一个卓有成效的函数中，确保每个可能的路径都对应一个 `return` 语句是一个良好的习惯。例如：

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

这个程序是错误的，因为当 `x` 恰为 0 的时候，没有一个条件语句为真，函数结束时没有遇到 `return` 语句。如果一个程序执行到函数的末尾，它将返回一个 `None` 值，而不是绝对值 0。

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

顺便一提，Python 提供了一个内建函数 `abs`，可以计算一个数的绝对值。

Exercise 6.1 编写比较 函数，函数返回 1 如果 `x > y`，返回 0 如果 `x == y`，返回 -1 如果 `x < y`。

6.2 增量开发

当你编写越来越大的函数时，你发现在调试上会花更多的时间。

对于越加复杂的程序，你可以尝试增量开发的方法。增量开发是通过每次测试一小部分代码的方法避免过长的调试过程。

例如，假设你需要计算给定坐标 (x_1, y_1) 和 (x_2, y_2) 两点间的距离。根据毕达哥拉斯理论，距离为：

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

首先考虑 `distance` 函数在 Python 中应该是如何的。换言之，函数的输入（参数）和输出（返回值）应该是什么？

在本例中，输入是两个点，你可以用 4 个数来表示。返回值是距离，可以用一个浮点数来表示。

现在你可以写出函数的框架：

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

显然，这个版本并没有计算距离；它总是返回零。但是它的语法是正确的，可以正常运行，这意味着你在把它修改的更复杂前可以测试它。

为了测试新函数，我们使用样例参数来调用它：

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

我选择的两个点水平距离为 3，垂直距离为 4，因此两点距离为 5（斜边为 3-4-5 的三角形）。当测试一个函数时，直到正确答案是很有用的。

现在我们确定函数的语法是正确的，我们可以开始添加代码。下一步是计算 $x_2 - x_1$ 和 $y_2 - y_1$ 的差，下一个版本将这两个值储存在临时变量中，并打印。

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

如果函数可以工作，它应该显示 'dx is 3' 以及 'dy is 4'。如果没错，我们知道函数得到了正确的参数，并正确的执行了第一步计算。如果有错，我们只需要检查几行代码。

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
```

同样，你可以运行程序，并检查结果（应该是 25）。最后你可以使用 `math.sqrt` 来计算并返回结果。

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

如果运行正确，你就完成了这个程序。否则，你需要在返回语句前打印 `result`。

函数的最终版本在执行过程中并不显示任何内容。`print` 语句在调试的过程中非常有用，但是一旦程序工作正常了，你需要删除它们。这样的代码被称为脚手架代码，它们对编写程序有帮助，但并不是最终结果的一部分。

当你刚开始时，你应该每次只添加几行代码。当你变得熟练时，你发现你可以调试大段的代码。通常，增量开发可以节约你很多调试的时间。

这个过程的关键在于：

1. 编写程序时每次只做少量修改，在任何时刻如果出错，你可以很方便的定位错误。
2. 使用临时变量记录中间过程值，这样你可以显示并检查它们。
3. 当程序正常工作后，你可能需要删除一些脚手架代码或者合并多个语句为一个复合语句。注意保持程序的可读性。

Exercise 6.2 使用增量开发编写一个斜边函数，读取三角形两条直角边作为参数，返回斜边长度。记录下开发过程中的每一个阶段。

6.3 Composition

现在你应该可以预料，你可以在一个函数中调用另一个函数。这种能力被称为复合。

例如，我们将写一个函数，它读入两个点作为参数，一个是圆心，另一个是圆周上的一个点，该函数将计算圆周的面积。

假设圆心对应的点保存在变量 `xc` 和 `yc`，圆周上的点的坐标为 `xp` 和 `yp`。第一步是计算圆的半径，即两点之间的距离。我们刚写了函数 `distance`，它实现了这个功能：

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

第二部是通过半径计算圆的面积，我们刚才也实现了：

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

将这两步封装在一个函数，我们得到：

```
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` 用来开发和调试，但是当程序正常工作后，我们可以通过复合这些函数调用来使得程序看起来更简洁：

```
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，这样便于隐藏函数中复杂的细节。例如：

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

通常布尔函数的函数名类似 `yes/no` 的问题；`is_divisible` 根据 `x` 是否可以被 `y` 整除，返回 `True` 或 `False`。

下面给出一个例子：

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

运算符 `==` 的结果返回一个布尔值，所以我们可以直接返回结果，这样更加简洁：

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

布尔函数通常用在条件语句中：

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

也许会写成类似如下的代码：

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

但是多余的比较是不必要的。

Exercise 6.3 写一个函数 `is_between(x, y, z)`，如果 $x \leq y \leq z$ 则返回 `True`，否则返回 `False`。

6.5 更多递归

我们仅仅覆盖了 `Python` 的一小部分，但也许你会对这是一个完备的编程语言集感兴趣，即任何可以被计算的东西都可以用这用语言来表示。任何现有的程序都可以用你现在学到的语言特征重写（事实上你还需要一些命令来控制键盘、鼠标、磁盘等设备，但仅此而已）。

这种说法首先由早期计算机科学家 **Alan Turing** 提出（有人会争论他是一个数学家，但许多早期的计算机科学家是数学家起家的），被称为图灵理论。如果你想对图灵理论有更完全（更精确）的讨论，我推荐 **Michael Sipser's** 的书 *Introduction to the Theory of Computation*。

为了让你认识到你可以用到目前为止学到的工具做什么，我们来计算一些递归定义的数学函数。递归定义类似循环定义，定义的本身包含对所定义的东西的引用。一个真正的循环定义不是很有用：

frabjous: 一个描述 **frabjous** 东西的形容词。

如果在字典里看到这样的解释，你也许会感到恼怒。然而，如果你查看阶乘的定义（表示为 $!$ ），你会的到类似这样的东西：

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

这个定义规定 0 的阶乘是 1 ，对于其他的任何数 n 的阶乘，其值为 n 乘以 $n-1$ 的阶乘。

因此 $3!$ 的阶乘是 3 乘以 $2!$ ， $2!$ 的阶乘是 2 乘以 $1!$ ， $1!$ 的阶乘是 1 乘以 $0!$ 。将这些放在一起， $3!$ 等于 3 乘以 2 乘以 1 乘以 1 ，结果为 6 。

如果你可以写出某个东西的递归定义，你通常可以通过写一个 `Python` 程序来进行求值。第一步是决定参数。很明显在本例中 `factorial` 读取一个整数作为参数：

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

如果参数等于 0 ，我们所要做的就是返回 1 ：

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

否则，将开始有趣的部分，我们递归的调用阶乘函数 $n-1$ 然后乘以 n ：

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

程序的执行流程类似章节 ?? 中 `countdown`。如果我们使用参数 3 调用 `factorial`：

由于 3 不是 0，我们得到第二个计算 $n-1$ 阶乘的分支...

由于 2 不是 0，我们得到第二个计算 $n-1$ 阶乘的分支...

由于 1 不是 0，我们得到第二个计算 $n-1$ 阶乘的分支...

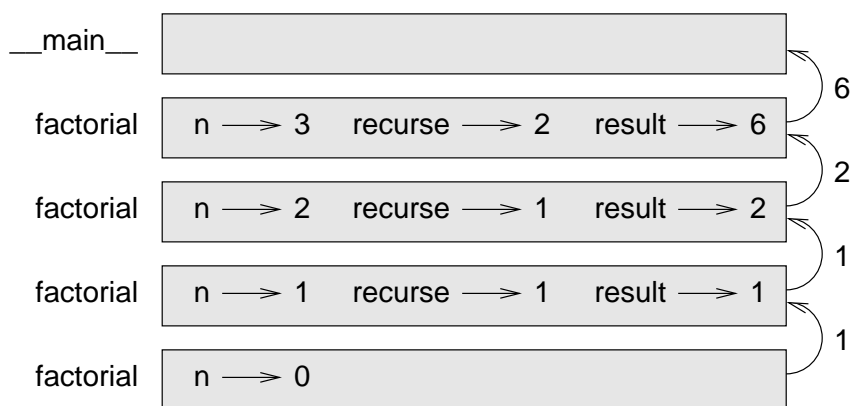
由于 0 等于 0，我们得到第一个分支，返回结果 1，不再进行更多的递归调用。

返回值 (1) 被乘以 n ，对应 1，然后结果返回。

返回值 (1) 被乘以 n ，对应 2，然后结果返回。

返回值 (2) 被乘以 n ，对应 3，结果为 6，这是整个函数最终返回的结果。

下面给出这一系列函数调用对应的栈图：



如图所示，返回值通过栈向上传递。在每一帧中，返回值是 `result` 的值，即 n 和 `recurse` 的乘积。

在最后一帧中，局部变量 `recurse` 和 `result` 并不存在，因为该分支并没有创建它们。

6.6 信心的飞跃

按照程序执行流程是阅读程序的一个方法，但这么做很容易走入迷宫。这里给出另一个方法，我称之为“信心的飞跃”。当你遇到一个函数调用，你假设函数执行无误，并返回正确的结果，而不是追溯程序执行流程。

事实上，当你调用内建函数时，你已经体验过信心的飞跃。当你调用 `math.cos` 和 `math.exp` 时，你并不检查这些函数的主体。你仅仅假设它们是由优秀的程序员编写的，可以正常的工作。

这同样适用于你自己写的函数。例如，在章节 6.4 中，我们编写了一个叫做 `is_divisible` 的函数，判断一个数是否可以被另一个数整除。一旦我们通过检查测试函数确信它可以正常工作，我们可以直接调用该函数，而不再去查看函数主体。

以上思想同样适用于递归程序。当你遇到递归调用，你应该假设递归调用工作正常（并返回正确结果），而不是跟着程序执行流程。你可以这么问你自己，“假设我可以得到 $n-1$ 的阶乘，我应该怎么计算 n 的阶乘？”很明显，通过乘以 n 就可以了。

当然，假设程序工作正常，而事实上你还没有完成这个函数，这里有一些诡异，但这就是为什么称之为信心的飞跃！

6.7 另一个例子

除了阶乘外，最常见的递归定义的数学函数是斐波纳契函数，其定义如下：¹：

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

翻译成 Python 为：

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

如果你试图跟随执行过程，即使很小的数 n ，你的脑袋也会爆炸。但是根据信心的飞跃，假设两个递归调用工作正确，就很容易看出通过将两者相加你将得到正确的结果。

6.8 类型检查

如果我们调用 `factorial` 函数时提供的参数是 1.5 会怎么样？

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

¹参见 wikipedia.org/wiki/Fibonacci_number。

看起来这将会是一个无穷的递归。这事怎么发生的？递归函数有一个初始情况——当 `n == 0`。但是如果 `n` 不是一个整数，我们将错过 初始状态并不断的递归。

在第一次递归调用中，`n` 的值是 0.5。在下次调用时，它变为 -0.5。接着，它变得越来越小（更朝负方向），永远不会为 0。

我们有两种选择。我们可以推广 `factorial` 函数使之可以工作在浮点数，或者我们可以让 `factorial` 检查参数的数据类型。第一个方法可以参考 `gamma` 函数²，超出了本书的范围。让我们看看第二个方法。

我们可以使用内建函数 `isinstance` 来验证参数的数据类型。同时我们可以确保参数的正的：

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

第一部分处理非整数；第二部分捕获负数。在这两种情况下程序将打印错误信息，并返回 `None` 表示有错误发生：

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is only defined for positive integers.
None
```

如果通过了两个检查，我们知道 `n` 是一个正整数，可以保证递归会终止。

这个程序展示了一个称为监护人的模式。前两个条件扮演了监护人的角色，防止错误的参数造成代码运行的错误。监护代码保证了程序的正确性。

6.9 调试

将大的程序拆成小的函数的过程给出了合理的调试测试点。如果一个函数工作不正常，有三种情况需要考虑：

- 函数得到的参数有误，即先决条件没有满足。
- 函数本身有误，即后决条件没有满足。
- 返回之有误，或者使用方式不正确。

²参考 wikipedia.org/wiki/Gamma_function。

为了排除第一种可能性，你可以在函数开始部分添加 `print` 语句，打印参数值（数据类型）。或和你可以直接编写代码检查先决条件。

如果参数没有问题，在每条 `return` 语句前添加 `print` 语句，打印返回值。可能的话手工检查结果。试图使用容易检查的参数来调用函数（参见章节 6.2）。

如果函数工作正常，检查函数调用，确保返回值被正确使用（或被使用！）。

在函数开始和结尾添加打印语句可以使程序流程更加明显。例如，下面给出带打印语句的 `factorial` 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` 是空格的字符串，用来缩进输出结果。下面给出 `factorial(5)` 的输出结果：

```
                factorial 5
            factorial 4
        factorial 3
    factorial 2
factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
```

如果你不清楚程序执行的流程，这样的输出信息很有帮助。搭建脚手架会花费一些时间，但会帮助节省很多调试时间。

6.10 术语

临时变量： 在复杂的计算过程中用来记录中间值的变量。

死区代码： 程序中无法执行到的代码，通常因为出现在 `return` 语句的后面。

None： 一个特殊的函数返回值，表明函数没有返回语句，或者返回语句没有参数。

增量开发： 一种程序开发的方法，通过每次添加测试少量代码来避免程序的调试。

脚手架： 在程序开发过程中使用的代码，并不出现在最终版本中。

守护人： 使用条件语句检查错误并处理可能出错的情况的编程模式。

6.11 练习

Exercise 6.4 画下面程序的栈图。程序执行时会打印什么内容？

```
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):
    sum = x + y + z
    pow = b(sum)**2
    return pow

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

Exercise 6.5 Ackermann 函数, $A(m, n)$, 定义为³:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \quad (6.1)$$

编写一个 `ack` 的函数来计算 Ackerman's 公式。使用你的函数来计算 `ack(3, 4)`, 结果应该是 125。如果选择大的 m 和 n 会发生什么？

Exercise 6.6 回文是一个从前往后和从后往前拼写相同的单词, 如 “noon” 和 “redivider”。从递归的角度来看, 如果一个单词第一个字母和最后一个字母相同, 且中间是回文的, 那么这个单词是回文的。

下面的函数读取一个字符串作为参数, 分别返回第一个、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]
```

我们会在章节 ?? 学习它们是如何工作的。

1. 在 `palindrome.py` 文件中输入这些函数并测试。如果你输入两个字母, 调用 `middle` 会返回什么? 一个字母呢? 空字符串呢 (使用 `''` 输入, 不包含任何字母)?

³See wikipedia.org/wiki/Ackermann_function.

2. 编写函数`is_palindrome`，读取一个字符串作为参数，如果是回文则返回 `True`，否则返回 `False`。你可以使用内建的 `len` 函数来检查字符串的长度。

Exercise 6.7 数 a 称为数 b 的幂，如果 a 可以被 b 整除，同时 a/b 是 b 的幂。编写函数 `is_power`，读取 a 和 b 作为参数，如果 a 是 b 的幂则返回 `True`。

Exercise 6.8 最大公约数（GCD）是可以整除 a 和 b 最大的数⁴。

寻找两个数的 GCD 的一个方法是欧几里得算法，算法指出如果 r 是 a 除以 b 的余数，那么 $\text{gcd}(a, b) = \text{gcd}(b, r)$ 。对于基本情况，我们考虑 $\text{gcd}(a, 0) = a$ 。

编写函数 `gcd`，参数为 a 和 b ，返回值为两者的最大公约数。如果你需要帮助，参考 wikipedia.org/wiki/Euclidean_algorithm。

⁴本练习基于 Abelson and Sussman's *Structure and Interpretation of Computer Programs* 的习题。

Chapter 7

迭代器

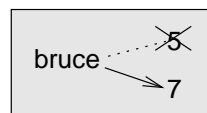
7.1 多重赋值

你可能已经发现，给一个变量多次赋值是合法的。一次新的赋值使得已存在的变量指向一个新值（当然也就不指向原来的值）。

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

程序的输出是 5 7，因为第一次 `bruce` 被输出时，它的值是 5，第二次是 7。第一个 `print` 语句末尾的逗号抑制了换行，这也是为什么两个输出在同一行的原因。

下图是多重赋值的状态图：



对于多重赋值，很有必要分清赋值操作符和关系运算符中的等号。因为 `Python` 使用等于号 (`=`) 来表示赋值。很容易，误把这样的语句 `a = b` 当作判断相等的语句，实际不是的！

第一，相等是对称关系，赋值不是。比如，在数学中，如果 $a = 7$ ，那么 $7 = a$ 。但在 `Python` 中，语句 `a = 7` 是合法的，`7 = a` 是非法的。

另外，在数学中，相等语句总是要么为真要么为假。如果， $a = b$ ，则， a 总是等于 b 。在 `Python` 中，赋值语句可以使得两个变量相等，但是他们不总是保持相等：

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

第三行改变了 `a` 的值，但是没有改变 `b` 的值。所以他们不再相等。

尽管多重赋值通常是有益的，使用时也要小心。如果变量的值经常改变，代码会变得很脑阅读和调试。

7.2 更新变量

多重赋值的最常见的形式之一就是更新（update），变量的新值依赖于原有值。

```
x = x+1
```

含义是：获取 `x` 的当前值，加一，然后用新值更新变量 `x`。

如果试着更新一个不存在的变量，将会得到一个错误，因为 Python 在把值赋给 `x` 之前会计算右边的值。

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

在更新一个变量之前，必须得初始化（initialize）它，通常的做法就是一个简单的赋值。

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

仅仅通过加一来更新变量叫做增量（increment）；减一叫做减量（decrement）。

7.3 while 语句

计算机通常被用来自动完成重复性的任务。计算机很擅长于重复相同或相似的任务。人类却恰恰相反¹。

我们已经看到两个程序，`countdown` 和 `print_n`，它们使用递归实现重复，这也可以乘坐迭代 `iteration`。因为迭代是如此的常见，以致于 Python 提供了几个特有的方式来简化使用。其中之一就是我们在 4.2 部分看到的 `for` 语句。我们不久将回来重新研究它。

另外一个就是 `while` 语句。这里是一个使用 `while` 语句的 `countdown` 版本。

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

我们几乎可以把 `while` 语句当成英语来读了。含义是：当 `n` 大于 0 时，显示 `n` 的值，并把 `n` 的值减 1。当 `n` 的值为 0 的时候，显示 `Blastoff!`。

更正式地，下面的是 `while` 语句的执行流。

1. 计算条件的值，产生结果 `True` 或者 `False`。
2. 如果条件为假，退出 `while` 循环，继续执行下一条语句。
3. 如果条件为真，执行语句体里的语句，然后回到步骤一。

¹太枯燥了，回顾一下小学时，老师天天要求抄生字.....

执行流的类型称作是循环 (loop) 的原因是因为第三步循环返回至第一步。

循环体应该改变一个或多个变量的值, 使得最终条件为假, 循环终止。否则, 循环将永远重复, 也就产生了无限循环 (infinite loop)。计算机科学家的一个永远的谈资就是看到洗发水的说明" 泡沫, 漂洗, 重复", 是一个无限循环。

在 `countdown` 的例子中, 我们可以证明循环一定会终止, 因为我们知道 `n` 的值是有限的, 并且每一次循环 `n` 的值都会减小, 最终, `n` 的值肯定是 0。在其他情况下, 就不一定这么容易辨别了:

beforeverb

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

这个循环的条件是 `n != 1`, 所以循环会一直执行到 `n` 是 1, 此时条件为假。

每一次循环, 程序输出 `n` 的值, 然后检查是否为偶数或奇数。如果是偶数 `n` 就除以 2。如果为奇数, `n` 的值就被 `n*3+1` 代替。比如, 如果传递 3 给 `sequence`, 产生的结果是 3, 10, 5, 16, 8, 4, 2, 1。

因为 `n` 时增时减, 没有一个明显的办法确定 `n` 是否会为 1, 也就是程序是否会正常终止。对于某些特别的 `n`, 我们可以证明终止。比如, 如果 `n` 的值是 2 的倍数, 每次循环时 `n` 的值, 都是偶数直到为 1。前面的例子从 16 开始都是这种情况。

困难的是我们是否可以证明对所有的正整数, 程序都能终止。迄今为止²没有人可以证明可以, 也没有人证明不可以。

Exercise 7.1 重写??部分的`print_n`函数, 要求使用迭代器, 而不是递归。

7.4 break 语句

有时, 直到执行到循环体里面的时候, 才直到需要跳出循环。此时, 我们可以使用 `break` 语句跳出循环。

比如, 假设一直想从用户那里得到输入, 直到用户输入 `done`。我们可以这么写:

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

print 'Done!'
```

²参看wikipedia.org/wiki/Collatz_conjecture.

循环条件为 `True`，也就是永远为真，所以循环直到遇到 `break statement` 才终止执行。

每次循环，用尖括号提示用户。如果用户输入 `done`, `break` 语句终止了循环。否则，程序输出用户输入的内容，会到循环的顶部。下面是一个例子：

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

这种使用 `while` 循环的方式很常见，因为我们可以循环的任何地方检查条件（不仅仅是在顶部），同时也积极的表达了结束的条件（当这个发生时，终止），而不是消极地（"一直运行，直到这个发生"）。

7.5 平方根

循环经常用在计算数值的程序中，通常都是以一个相近的值开始，然后迭代，逐渐提高。

比如，有一种计算平方根的算法叫牛顿方法。假设，我们想得到 a 的平方根。如果以任意猜测的一个值开始，我们可以用下面的公式，计算一个更好的猜测值：

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

比如，如果 a 是 4， x 是 3：

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

结果已经接近正确答案了（ $\sqrt{4} = 2$ ）。如果我们重复这个过程，就更接近了：

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

经过几次更新，猜测值基本上等于精确值了：

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

一般来说，我们实现并不知道经过多少步才能得到正确的结果，但是我们知道什么时候得到正确的结果，应为猜测值成定值了。

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

当 `y == x`，我们就可以停止了。下面是一个循环，从一个初始值开始，然后逐步逼近，直到猜测值为定值。

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

对于大多数的 `a`，这个都适用。但是，一般说来，测试浮点数是否相等是危险地。浮点值只是近似相等：大多数有理数，像 $1/3$ ，和無理数，像 $\sqrt{2}$ ，不能用一个浮点数精确的表示。

与其检查 `x` 和 `y` 是否精确相等，不如安全的采用内置的函数 `abs` 计算差的绝对值：

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

这里，`epsilon` 是一个极小值，像 `0.0000001`，表示了什么样的接近才是非常接近了。

Exercise 7.2 把这个循环封装在函数 `square_root` 里，接受 `a` 为参数，选择一个合适的 `x`，返回 `a` 的近似平方根。

7.6 算法

牛顿方法是算法的一个例子：它是解决一类问题的方法（在这个例子里是计算平方根）。

定义一个算法可不是件容易的事。从不是算法的方面入手或许会有帮助。当你学习单位数相乘时，你背了乘法表。事实上，你记住了 100 个具体的解法。这种知识不是算法。

但是，如果你比较“懒”，你可能作些小弊。比如，计算 n 和 9 的乘积，你可以把十位写成 $n-1$ ，个位写成 $10-n$ 。这个就是解决任何单位数乘以 9 的一般方法³。对了，这就是算法。

类似地，我们学到的技巧，比如加法进位，减法借位，长除法都是算法。这些算法的共有的特点就是他们不需要任何的智力来实施。他们是机械的过程，每一步都依靠简单的规则。

从我的观点来看，人们花费大量的时间在学校里学习 ---不夸张地说，不需要任何智力的算法是非常不值得的。

³译注：比如 $8*9 = 72 = (8-1)(10-8)$

从另一方面来说，设计算法的过程确实有趣的，挑战智力的，也是程序设计的中心内容。

有些事情，人们作起来很自然，没有困难也无须苦思冥想，但却是最难用算法表达的。理解自然语言是一个好的例子。我们都有这种能力，但是至今没有人能解释为什么我们可以，至少我们不能以算法的形式表达出来。

7.7 调试

当我们开始编写大型的程序时，就会发现调试将会花费我们大量的时间。代码越多，意味着犯错的机会就越大，隐藏 `bug(s)` 的地方就越多。

减少调试时间的一种方法就是“二分法”。例如，程序有 100 行代码，每次检查一个，需要 100 步。

换一种思路，把问题分成两半。查看程序的中间部分，或者接近中间部分，寻找一个中间值来检查。添加一个 `print` 语句（或者其他的能产生验证效果的语句），然后执行程序。

如果中间检查有问题，那么必定是程序的前半部分有问题。反之，则在第二部分。

每次按照这样检查的话，我们缩减了需要检查的代码。至少理论上来说，六步以后，（这远远小于 100），我们就可以缩减到一两行代码了。

实际中，很难界定程序的中间部分是哪里，也不总可能检查它。数代码的行数然后计算精确的中间点是没有任何意义的。相反，仔细想象什么地方最有可能出现错误，什么地方容易插入一个语句来检查。然后，选择一处你认为 `bug` 出现在检查点前后几率近似相等的地方。

7.8 术语表

multiple assignment 多重赋值： 在程序的执行过程中给同一个变量赋多次值。

update 更新： 变量的新值依赖原来值的赋值。

initialization 初始化： 给一个将被更新的变量初始值的赋值。

increment 增量： 增加一个变量的更新（通常是 1）。

decrement 减量： 减小一个变量的更新（通常是 1）。

iteration 迭代： 使用递归或者循环的重复性执行的语句集合。

infinite loop 无限循环 终止条件永远不满足的循环。

7.9 练习

Exercise 7.3 测试这章的平方根算法，你可以把结果和 `math.sqrt` 的结果比较。写一个函数 `test_square_root` 打印一个如下的表：

```

1.0 1.0          1.0          0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0          2.0          0.0
5.0 2.2360679775 2.2360679775 0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0          3.0          0.0

```

第一列是数字 a , 第二列是用 7.2 计算的 a 的平方根, 第三列是用 `math.sqrt` 计算的平方根, 第四列是两个结果的差值。

Exercise 7.4 内置函数 `eval` 接受一个字符串, 然后调用 `python` 解释器计算字符串的值, 例如:

```

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

```

编写一个函数 `eval_loop` 重复的提示用户, 接受用户输入, 然后调用 `eval` 计算值, 并打印结果。

程序必须知道用户舒服 'done' 才结束循环, 然后返回最后计算的表达式的值。

Exercise 7.5 杰出的数学家 Srinivasa Ramanujan 发现了无穷序列⁴可以用来产生近似的 π 值。

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

编写函数 `estimate_pi`, 函数使用上面的公式计算 π 值, 并返回之, 函数应该使用一个 `while` 循环计算项的和知道最后一项小于 `1e-15` (Python 里的记法为 `10-15`)。你可以拿它和 `math.pi` 比较一下。

可以参看我的解答 thinkpython.com/code/pi.py。

⁴参看 wikipedia.org/wiki/Pi.

Chapter 8

字符串

8.1 字符串是一个序列

字符串是字符的序列。你可以使用括号运算符每次访问一个字符。

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

第二条语句从 `fruit` 中读取第一个字符，并赋值给 `letter`。

在括号中的表达式称为下标。下标指示了序列中对应的字符。

但是你却也许没有得到你期望的：

```
>>> print letter
a
```

对于大多数人来说，`'banana'` 是 `b`，而不是 `a`。但是对计算机科学家来说，下标是从字符串开始位置的偏移量，第一个字符的偏移量是零。

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

所以 `b` 是 `'banana'` 中第零个字符，`a` 是第一个字符，`n` 是第二个字符。

你可以使用包含变量和运算符的表达式作为下标，但下标值必须是一个整数，否则你将得到：

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

8.2 len

`len` 是一个内建函数，返回一个字符串的长度：

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

要得到一个字符串最有一个字符，你可以这么做：

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

下表错误的原因是因为 'banana' 中没有字符下标为 6。下标从 0 开始计数，6 个字符的下标分别是 0 到 5。要得到最后一个字符，你要对 `length` 减 1：

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

另一个方式是使用负数下标，它将从字符串尾部计数。表达式 `fruit[-1]` 返回最后一个字符，`fruit[-2]` 返回倒数第二个字符，以此类推。

8.3 使用 for 循环遍历

许多计算包含对字符串字母逐一操作。通常从字符串头开始，每次选择一个字符进行操作，直到字符串尾。这个过程称为遍历。遍历的一个方法是使用 `while` 循环：

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

这个循环遍历字符串并在每行打印一个字母。循环条件是 `index < len(fruit)`，当 `index` 等于字符串的长度，条件为假，循环的主体将不再执行。最后一个被访问的字符的下标是 `len(fruit)-1`，对应字符串的最后一个字符。

Exercise 8.1 编写一个函数，读取一个字符串作为参数，逆序打印字符串，每行一个字符。

遍历的另一个方法是使用 `for` 循环：

```
for char in fruit:
    print char
```

对于每一次循环，字符串中的下一个字符被赋值给变量 `char`。当所有字符都赋值过后，循环结束。

下面的例子演示了如何使用连接（字符串相加）和 `for` 循环来生成一个字母表（即字母顺序）。在 Robert McCloskey 的 *Make Way for Ducklings* 中，小鸭子们的名字为 Jack, Kack, Lack, Mack, Nack, Ouack, Pack 和 Quack。

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

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


输出为:

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

当然这不完全正确, 因为” Ouack “和” Quack “拼写错误了。

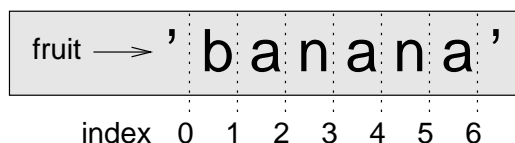
Exercise 8.2 修改程序以修复错误。

8.4 字符串切片

字符串的一段称为切片。选择字符串的一个切片和选择一个字符类似:

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

运算符 `[n:m]` 返回字符串中第 `n` 到第 `m` 个字符, 包括第 `n` 个字符, 但不包括第 `m` 个字符。这种行为是违反直觉的, 但是我们可以通过想象在在字符间放置标号来帮助理解, 如下图所示:



如果你忽略第一个下标 (在冒号之前), 切片开始于字符串的头部。如果你忽略第二个下标, 切片将结束于字符串的尾部:

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

如果第一个下标大于等于第二个下标, 结果将为一个空字符串, 用两个引号表示:

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

空字符串不包含任何字符且长度为零, 除此以外和其他字符串相同。

Exercise 8.3 假设 `fruit` 是一个字符串, `fruit[:]` 是什么?

8.5 字符串是不变的

下面尝试使用 `[]` 运算符作为赋值语句的左值，试图改变字符串中的字符：

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

在本例中”对象“是字符串，”项目“是你试图赋值的字符。现在，一个”对象“等同于一个数值，以后我们会重新进行定义。一个”项目“是序列中的一个值。

错误的原因是因为字符串是不可变的，即你不可以修改一个已经存在的字符串。你可以做的是新建一个字符串并在原来的基础上修改：

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

例子中将新的第一个字母和 `greeting` 的一个切片连接，这个操作对原来字符串没有影响。

8.6 搜索

以下函数实现了什么功能？

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

从某种角度来说，`find` 是 `[]` 运算符的反操作。它读取字符，寻找字符串中对应的下标。如果字符没有找到，函数返回 `-1`。

这是我们要看到的第一个 `return` 语句在循环体中的例子。如果 `word[index] == letter`，函数中断跳出循环，并立即返回。

如果字符串中不存在该字符，程序正常退出循环并返回 `-1`。

这种遍历序列寻找我们需要的内容的计算模式称为搜索。

Exercise 8.4 修改 `find` 函数，增加第三个参数 `word`，指示开始搜索的位置。

8.7 循环和记数

下面的程序统计字符 `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
```

程序演示了另一种计算模式记数。变量 `count` 首先被初始化为 0，然后每次发现 `a` 就增加 1。当循环退出后，`count` 记录了 `a` 总共出现的次数。

Exercise 8.5 将代码封装成一个名叫 `count` 的函数，推广函数是得它可以接收字符串和字符作为参数。

Exercise 8.6 重写函数，使用 3 个参数的 `find` 版本，而不是遍历字符串。

8.8 string 方法

方法类似函数，它读取参数，并返回一个值，但两者语法有区别。例如，方法 `upper` 读取一个字符串并返回一个全大写的新字符串：

相比函数的语法 `upper(word)`，方法的语法为 `word.upper()`。

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

这样的点符号的形式指定了方法的名称 `upper`，以及方法作用的字符串名 `word`。空的括号指示这个方法不读取参数。

一个方法的调用称为调用；在本例中，我们称对 `word` 调用了 `upper` 方法。

事实上，字符串有一个方法 `find` 十分类似我们写的函数：

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

在这个例子中，我们对 `word` 调用了 `find` 方法，并传入我们需要寻找的字符作为参数。

事实上，`find` 方法比我们的函数适用更广，它不仅可以查找字符，也可以查找子串：

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

它可以读取第二个参数来指定开始查找的位置：

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

还有第三个参数，指定查找结束的位置：

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

这次查找失败，因为 `b` 在查找范围 1 到 2 中不存在（不包含 2）。

Exercise 8.7 字符串有个方法称为 `count`，类似先前练习中的函数。阅读这个方法的文档，编写方法调用，统计 `a` 在 `'banana'` 中出现的次数。

8.9 in 运算符

`in` 是一个布尔运算符，它读取两个字符串作为参数，如果第一个字符串是第二个字符串的子串，则返回 `True`：

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

例如，下面的函数打印在 `word1` 和 `word2` 中同时出现的字符：

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

如果函数名是精心挑选的，Python 有时读起来想英语。你可以这么读一个循环：“for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

下面是你比较 `apples` 和 `oranges` 的输出：

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

8.10 字符串比较

关系运算符可以作用在字符串上。看两个字符串是否相等：

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

其他关系运算符可以用来将单词按字母表排序：

```
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 不像人们一样处理大小写字母。所有的大写字母在小写字母之前，所以：

Your word, Pineapple, comes before banana.

一个常用的解决这个问题的方法是在比较前将字符串转化为一个标准的格式，如全小写。

8.11 调试

当使用下标遍历序列的值时，很容易将遍历的开始位置和结束位置搞错。这里给出了一个函数，它试图比较两个单词，如果一个单词是另一个的逆序则返回 `True`，函数中有两个错误：

```
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
```

第一个 `if` 语句检查两个单词是否长度相同。如果不相同，我们可以立即返回 `False`。否则，我们可以保证两个单词长度相同。这是章节 ?? 中一个守护人模式的例子。

`i` 和 `j` 是下标，`i` 从前往后遍历 `word1`，而 `j` 从后往前遍历 `word2`。如果我们发现两个字符不同，我们可以立即返回 `False`。如果我们完成整个循环，即所有字符都匹配，我们返回 `True`。

如果我们用 “pots” 和 “stop” 来测试这个函数，我们期望返回值为 `True`，然而我们得到下标错误：

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

为了调试这类错误，我第一步是在错误的行前面打印相应的值：

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

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

现在再次运行程序，我得到更多的信息：

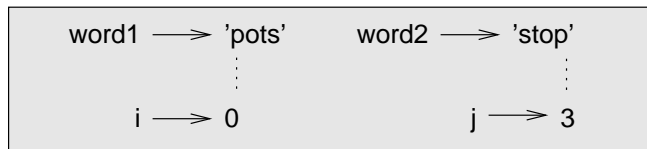
```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

在第一次穿越循环后，`j` 的值是 4，超过了 'pots' 的长度。最后一个字符对应的下标是 3，所以 `j` 的初始值应该为 `len(word2)-1`。

我修正了这个错误，再次运行程序得到：

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

这次我们得到正确的结果，但看起来循环只运行了 3 次，十分可疑。为了弄清楚是怎么回事，一个很有效的方法是画状态图。在第一次迭代中，`is_reverse` 的第一帧如下：



我用虚线标记出变量 `i` 和 `j` 对应 `word1` 和 `word2` 中的字符。

Exercise 8.8 从这个图开始，在纸上运行程序，在每次迭代中修改 `i` 和 `j` 的值。找出并修改函数中的错误。

8.12 术语

对象： 变量可以指向的东西。到目前，你可以互换的使用“对象”和“值”。

序列： 一个有序的集合，每一个集合中的数对应一个整数下标。

项目： 序列中的一个项目。

下标： 序列中用于选择项目的一个整数值，如字符串中的一个字符。

切片： 有一定下标确定的字符串的一部分。

空字符串： 一个没有字符的长度为 0 的字符串，使用两个引号表示。

不可变： 序列中的项目不可改变的性质。

遍历： 迭代访问序列中的每个项目，并执行类似的操作。

查找： 遍历的一种模式，当找到要找的内容时停止。

计数器： 用来计数的变量，通常初始化为 0，然后递增。

方法： 一个和对象关联的函数，应用点符号来调用。

调用： 调用方法的语句。

8.13 练习

Exercise 8.9 字符串的切片可以读取第三个参数“步长”，它决定了相邻字符之间的间隔。步长为 2 意味着每隔一个字符，步长为 3 意味着隔两个字符，以此类推。

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

步长为 -1 对应从后往前遍历单词，因此[::-1] 将产生一个翻转的字符串。

使用这个写法编写一个一行版本的 `is_palindrome` 函数，类似练习 12.4。

Exercise 8.10 阅读字符串的方法 docs.python.org/lib/string-methods.html。你需要使用部分函数来确保你理解它们是如何工作的。`strip` 和 `replace` 特别有用。

文档使用的语法可能令人困惑。比如，在 `find(sub[, start[, end]])` 中，括号表示参数是可选的。因此 `sub` 是必须的，而 `start` 是可选的，如果你包含了 `start`，那么 `end` 是可选的。

Exercise 8.11 下面的函数都试图检查一个字符串是否包含小写字母，但至少它们中的一些是错误的。对于每个函数，描述它们实际上做了什么（假设参数是一个字符串）。

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

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

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

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

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

Exercise 8.12 ROT13 是一种弱的加密方式，它将单词中的每个字母“旋转”13个单位¹。旋转一个字母意味着按字母顺序移动，有必要时绕回开始的位置，因此'A'移动3个单位是'D'，'Z'移动1个单位是'A'。

编写函数`rotate_word`，读取一个字符串和一个整数作为参数，返回一个新的字符串，它是在原来的字符串的字符“旋转”给定的量。

比如，“cheer”旋转7个单位是“jolly”，“melon”旋转-10个单位是“cubed”。

你也许会用到内建函数 `ord`，它将字符转换为一个数字代码，以及函数 `chr`，将数字代码转换为一个字符。

网上一些攻击性的玩笑有时候使用 ROT13 来加密的，如果你不是轻易被冒犯的，寻找并解码它们。

¹参见 wikipedia.org/wiki/ROT13。

Chapter 9

实例学习：字符处理

9.1 读取单词表

我们本章的联系需要一个英语单词表。在网络上有数以万计的单词表，但是最适合我们的一个是贡献给公共域的单词表，它是由 Grady Ward 搜集整理作为 Moby 词典工程的一部分¹。它由 113,809 个官方纵横组合字谜的单词组成，也就是在纵横组合字谜中和其他字谜游戏中存在的单词组成。在 Moby 的搜集中，文件名是 113089.fic；我复制了这个文件，命名为 words.txt，并把它包含在了 Swampy 里了。

这个文件是纯文本文件，你可以用一个编辑器打开它，当然，你也可以用 python 来读取它。内置函数 open 接受一个文件名作为参数，返回一个文件对象，用它可以读取文件。

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

fin 是赋给用来输入的文件对象的常见名称。大开模式 'r' 表明文件以只读方式大开（和 'w' 以只写模式打开相反）。

文件对象提供了多种方式来读取文件，其中就包括 readline，该函数从文件中读取字符知道遇到换行符，然后以字符串的形式返回结果：

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

这个单词表的第一个单词是 "aa," 是一种熔岩的名称。序列 \r\n 代表两个空白字符，回车和换行，它们把这个单词下一行分隔开来。

文件对象自己跟踪达到了文件的什么地方，所以当再次调用 readline 函数时，就会得到下一个单词：

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

下一个单词是 "aah," ---绝对合法的一个单词，不要以那么怪的眼神看我²。如果那个 whitespace 看上去很不给力，我们使用 strip 函数剔除它：

¹wikipedia.org/wiki/Moby_Project.

²译者：我没有笑话你的意思哦

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

也可以在 `for` 循环中使用文件对象。下面的程序对去 `word.txt`，打印每一个单词，一行一个：

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

Exercise 9.1 编写一个程序，读取 `words.txt`，打印包含超过 20 个字符的单词（不包括空白）。

9.2 练习

下个部分有这些练习的答案。你至少得在参看答案之前尝试做一做每一道题。

Exercise 9.2 1939 年，Ernest Vincent Wright 发表了一本 50,000 字的小说 *Gadsby*，在这本书中不包含字母 “e”。“e” 在英语中是非常常见的字母，所以这件事很难做到。

事实上，如果不使用最常见的符号是很难想象那样的情况的。开始进展比较缓慢，但是谨慎地训练几个小时之后，你就会逐步的掌握要领。

好的，进入正题！

编写一个函数 `has_no_e`，如果给定的单词不含有字母 “e”，返回 `True`。

修改上一部分的程序，打印不含有 “e” 的单词，并计算不含有 “e” 的单词的百分比。

Exercise 9.3 编写一个函数 `avoids` 接受一个单词和一串 “禁止” 字母，如果单词没有使用禁止字母中的任何一个，返回 `True`

修改你的程序提示用户输入一串禁止字母，然后打印不含有它们中任意一个的单词数目。你能找出由 5 个 “禁止” 字母组成的单词，不包括最小数目的单词吗？

Exercise 9.4 编写一个函数 `uses_only`，接受一个单词和一串字母，如果单词仅仅包含列表字符串里的字母，返回 `True`。除了 “Hoe alfalfa” 你能用 `acefhlo` 里的字母造一个句子吗？

Exercise 9.5 编写一个函数 `uses_all`，接受一个单词和一串字母，如果单词使用一串字母里的所有字母（至少一次），返回 `True`。有多少单词包含了全部的元音 `aeiou`？`aeiouy` 呢？

Exercise 9.6 编写一个函数 `is_abecedarian`，如果单词里的字母以字典顺序出现，返回 `True`。有多少 `abecedarian` 式的单词？

9.3 搜索

前一部分的所有联系都有一个共同点：他们都可以通过一个可搜索模式来解决，我们在??部分遇到过。最简单的例子是：

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

for 循环遍历 word 中的每一个字母；如果不匹配，就进入下一个字母。如果我们正常退出循环，就意味着我们没有找到“e”，于是返回 True。

avoids 是一个更一般的has_no_e 版本，但是有着相同的结构：

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

如果发现一个“禁止”字母，我们就返回 False；如果我们达到了循环的结尾，就返回 True。

uses_only 和它类似，出了条件的意思是相反的。

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

除了使用一串“禁止”字母，我们也可以使用可获得字母（available）。如果我们在 word 中发现一个字母不再可获得字母中，就返回 False。

uses_all 函数也类似，出了我们掉换了 word 和一串字母的角色。

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

我们没有遍历 word 中的字母，循环遍历了“要求”的字母，如果任意一个“要求”字母没有出现在 word 中，我们返回 False。

如果你想像一个计算机科学家一样思考，你可能已经认出uses_all 是前面已经解决过的问题的实例，你就会编写：

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

这是程序设计方法中的一个实例 ---叫做问题识别，含义是你认识到现在解决的问题是已经解决问题的一个实例，然后修改应用以前的解法。

9.4 使用索引循环

我用 `for` 循环编写以前的函数，因为我只需要用到字符串里的字符；我没有必要使用索引来解决问题。

对于 `is_abecedarian` 我们必须要比较相邻的字母，`for` 循环就有点力不从心了。

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

另外一种方法就是使用递归：

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

也还可以使用 `while` 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

循环从 `i = 0` 开始，以 `i = len(word) - 1` 收尾。每次循环时，比较第 `i`（你可以把它当成当前的字符）和 `i+1` 字符（你可以把它当作第二个字符）。

如果下一个字符小于当前的字符（字典顺序是在前），字母次序就被打断了，我们返回 `False`。

如果我们到达了循环的末尾并且没有发现错误，这个 `word` “通过”了测试。为了让自己信服循环正确的结束了，考虑这样一个例子 `'flossy'`。这个单词的长度是 `6`，所以最后一次循环执行的时候 `i` 是 `4`---正好是倒数第二个字符的索引。在最后一次迭代中，程序比较倒数第二个字符和最后一个字符 ---这就是我们希望的。

下面是一个 `is_palindrome` 的版本（参看练习 12.4），函数使用了两个索引；一个从头部开始，逐步递增，另外一个从尾部开始，逐步递减。

```
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，你也注意到了这又是已解决问题的一个实例，你或许会这么写：

```

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

```

我认为你做了练习 8.8。

9.5 调试

测试程序是一种“折磨”。本章的函数相对来说比较容易测试，因为你可以手动的检查输出结果。尽管如此，某些地方还是很难甚至不可能选择一个合适的单词表来测试所有可能的错误。

拿 `has_no_e` 做例子，有两种明显的情况需要测试：含有“e”的单词应该返回 `False`；不含有的返回 `True`。对于这两种情况，应该都没有任何错误。

在这两种情况下又有一些不太明显的子情况。在含有“e”的单词中，我们应该测试“e”在单词前，在单词尾，在中间的某个地方。也应该测试长单词，短单词，和非常短的单词，像空字符串。空字符串是特殊情况中的一个例子，也是非常不明显的情况，而错误恰恰经常藏身此处。

除了测试你自己想出的情况，你也可以用单词表来测试，比如使用 `word.txt`。通过查看输出，你就可以抓住错误，但是小心：你或许能抓住一种错误（本不该包括的单词被包括了）但不是另外一种（本该包含的单词没有被包含）。

一般说来，测试能帮助我们发现 `bugs`，但是产生好的测试案例不是一件容易的事，而且即使你测试了，你也不能保证你的程序就是正确的。

一位传奇式的计算机科学家这么说过：

```

    程序测试能够发现 bugs 是存在的，但是永远发现不了 bugs 不存在。
    ---Edsger W.Dijkstra

```

9.6 术语表

file object 文件对象：代表打开文件的数值。

problem recognition 问题识别：通过把问题阐释成已经解决问题的一个实例来解决问题的方法。

special case 特殊情况：非典型的或不明显的测试例子（很难完美解决）。

9.7 练习

Exercise 9.7 这个问题是基于广播节目 Car Talk 播发的一个难题³:

给我一个含有三个连续的双字母单词，我能给你几个几乎符合条件的单词，但是不是完全符合。比如，单词 `committee`, `c-o-m-m-i-t-t-e-e`。如果没有了 `i` 在那里，会更完美；`Mississippi` 也是 `M-i-s-s-i-s-s-i-p-p-i`。如果能够去掉这些 `i`，结果就很完美。有一个单词有三个连续的双字母，据我所知，只有这一个。当然，也有可能 有 500 多个，但是我只能想到这一个。那么这个单词是什么？

编写一个程序，寻找它。你可以参看我的答案 thinkpython.com/code/cartalk.py。

Exercise 9.8 下面的也是一个 Car Talk 难题⁴:

“有一天，我驾车在高速公路上行使，我偶然看到了我的里程表。像大多数的里程表一样，它显示了六个数字，全部是里数。比如，如果我的小汽车行使额 300,000 里，我将会看到 3-0-0-0-0-0。

“现在，我那天看到的却是非常的有意思。我注意到后四个数字是回文的；也就是说从前往后读和从后往前读是一样的。比如，5-4-4-5 是回文，所以我的里程表可能显示 3-1-5-4-4-5。

“行使了一里之后，后五位数是回文的了。比如，可能是 3-6-5-4-5-6。再行使一里后中间的四位又是回文的了。你或许猜到了，一公里后，六位数字是回文的了！

“问题是，当我第一次看里程表的时候，显示的是什么呢？”

编写一个 Python 程序，测试所有的六位数，打印任何满足条件的数字。你可以参看我的答案 thinkpython.com/code/cartalk.py。

Exercise 9.9 下面的又是一个 Car Talk 难题，你可以用搜索来解决它⁵:

“最近，我拜访了妈妈。我们认识到组成我年龄的两位数倒过来时是她的年龄。比如，如果她是 73，我就是 37。我们想知道这种事发生的频率是多少？但是我们转移到了另外一个话题，我们也就没有得到答案。

“当我会到家的时候，我计算出组成我们年龄的两位数已经有六次是像上面那样了。我也计算出，如果幸运的话，几年后又会发生，如果我们真的幸运的话，在那之后，还会一次。也就是说，总共会有 8 次。现在的问题是，我今年多大了？”

编写一个 Python 程序，搜索这个难题的答案。Hint: 你或许会发现字符串方法 `zfill` 对你有些帮助。

可以参看我的答案 thinkpython.com/code/cartalk.py。

³www.cartalk.com/content/puzzler/transcripts/200725.

⁴www.cartalk.com/content/puzzler/transcripts/200803.

⁵www.cartalk.com/content/puzzler/transcripts/200813

Chapter 10

列表

10.1 列表是一个序列

类似字符串，列表一个序列的值。在字符串中，每个值是字符；在一个列表中可以是任何数据类型。列表中的数值称为元素，有时也称为项目。

有多种方法可以创建一个新的列表；最简单的方法是用方括号（`[]`）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。第二个例子是包含 3 个字符串的列表。一个列表中的元素不需要是相同数据类型。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]]
```

在一个列表中的列表称为嵌套。

没有任何元素的列表称为空列表。你可以使用空的括号 `[]` 创建一个空列表。

正如你可能期望的，列表的值可以被赋值给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 列表是可改变的

访问列表中元素的语法和访问字符串中字符的语法相同，都是通过括号运算符实现的。括号中的表达式指定了下标。记住下标从 0 开始：

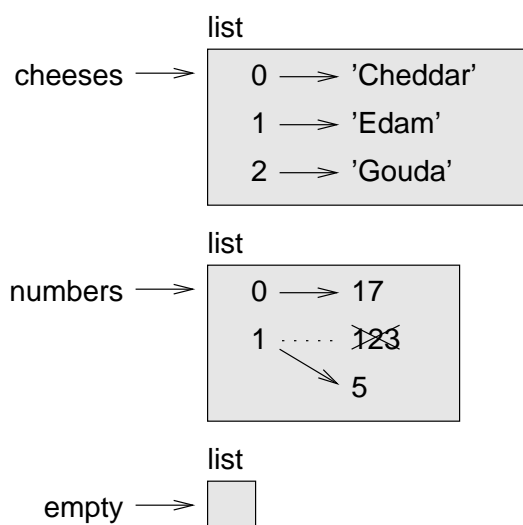
```
>>> print cheeses[0]
Cheddar
```

与字符串不同，列表是可以改变的。当括号运算符出现在赋值语句的左边，它指向列表中将被赋值的元素。

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

`numbers` 中的第一个元素，原来是 123，现在是 5。used to be 123, is now 5.

你可以将列表看成下标和元素的对应关系。这种关系成为映射。每个下标“对应”一个元素。这里给出 `cheeses`, `numbers` 和 `empty` 的状态图：



列表用外部标有“list”的盒子表示，内部是列表中的元素。`cheeses` 是一个有 3 个元素的列表，下标分别是 0, 1 和 2。`numbers` 包含 2 个元素。状态图显示了第二个元素原来是 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的相同：

- 任何整数表达式可以作为下标。
- 试图读写一个不存在的元素将得到下标错误。
- 下标可以取负数，它将从后往前访问列表。

`in` 运算符同样使用列表。

```
>>> 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。语法类似字符串：

```
for cheese in cheeses:
    print cheese
```

这种写法适用于只读列表中的元素。如果你需要写或者更新元素，你需要通过下标访问。一个常用的做法是结合 `range` 和 `len` 函数：

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

这个循环遍历列表并更新每个元素。`len` 函数返回列表中元素个数。`range` 函数返回一个从 0 到 $n-1$ 的下标的列表，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` 读取老的值并赋值新的值。

对于一个空列表的 `for` 循环将不会执行循环的主体：

```
for x in empty:
    print 'This never happens.'
```

虽然一个列表可以包含另一个列表，被嵌套的列表作为单独的一个元素。以下列表的长度为 4：

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

10.4 列表操作

运算符 `+` 连接列表：

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

类似的，运算符 `*` 给定次数地重复列表：

```
>>> [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 `[0]` 4 次。第二个例子重复列表 `[1, 2, 3]` 3 次。

10.5 列表切片

切片运算符同样适用于列表：

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

如果你忽略第一个下标，切片从列表头开始。如果你忽略第二个，切片到列表尾部结束。因此如果你忽略两个，切片为整个列表的拷贝。

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

由于列表是可改变的，有必要在折叠、旋转或切断操作前复制列表。

赋值语句左边的切片运算符可以更新多个元素：

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

10.6 列表方法

Python 提供了一个列表的方法。例如，`append` 方法将新的元素添加到列表尾部：

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

`extend` 方法读取一个列表作为参数，并附加其中所有的元素：

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

这个例子中 `t2` 没有改变。

`sort` 方法从小到大对列表中的元素进行排序：

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

列表的方法都是空的，它们对列表进行修改并返回 `None`。如果你写了 `t = t.sort()`，你不会得到你想要的结果。

10.7 映射，筛选和归并

对列表中所有元素求和，你可以这么使用循环：

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

`total` 被初始化为 0。每次经过循环，`x` 从列表中读取一个元素。运算符 `+=` 提供一个快捷的更新变量的方法。这是增量赋值语句：

```
total += x
```

相当于：

```
total = total + x
```

当循环执行时，`total` 记录了元素的和。这样的变量称为累加器。

对列表中元素求和是一个普通的操作，Python 提供了内建函数 `sum`：

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

类似将一个序列的元素合并到一个单一的数值的操作称为归并。

有时你需要遍历一个列表来构建另一个列表。例如，下面的函数读取一个字符串列表作为参数，返回大写后的新列表：

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

`res` 被初始化为一个空的列表。每次循环中，我们附加下一个元素。因此 `res` 是另一种累加器。

类似 `capitalize_all` 有时被称为映射，因为它对序列中的每个元素“映射”某个函数（在本例中是方法 `capitalize`）。

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` 是一个字符串的方法，如果字符串仅含有大写字母，则返回 `True`。

类似 `only_upper` 的操作被称为筛选，它选出一部分元素，而过滤其他元素。

绝大多数列表操作可以表示为映射、筛选和归并的组合。由于这些操作很常用，Python 提供了语言特点的支持，包括内建函数 `map` 和一个称为“列表理解”的运算符。

Exercise 10.1 编写函数，读取一个数字列表作为参数，返回累积求和值。即第 i 个元素是原列表中前 $i+1$ 个元素的和。例如，`[1, 2, 3]` 的累积求和为 `[1, 3, 6]`。

10.8 删除元素

有多种方法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)
>>> print t
['a', 'c']
>>> print x
b
```

`pop` 修改列表，并返回被删除的元素。如果你不提供下标，它将删除最后一个元素。

如果你不需要被删除的值，你可以使用 `del` 运算符：

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

如果你知道你要删除的元素（但不知道下标），你可以使用 `remove`：

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

`remove` 的返回值是 `None`。

要删除多于一个元素，你可以对 `del` 使用切片下标：

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

照常，切片选择从第一个下标到第二个下标（不包括第二个下标）中的所有元素。

10.9 列表和字符串

字符串是字符的序列，列表是值的序列，但是字符的列表不同于字符串。你可以使用 `list` 将字符串转化为字符的列表：

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

由于 `list` 是内建函数名，你应该避免将它作为变量名。我同样避免使用 `l` 因为它看起来像 `1`。于是我用了 `t`。

`list` 函数将字符串分割成单独的字符。如果你要将字符串分割成单词，你可以使用 `split` 方法：

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

一个可选的参数称为分割符，它指定了什么字符作为分界线。下面的例子使用连字符作为分割符：

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` 功能和 `split` 相反。它将一个列表字符串连接起来。`join` 是一个字符串方法，因此你需要在一个分割符上调用它，并传递一个列表作为参数：

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

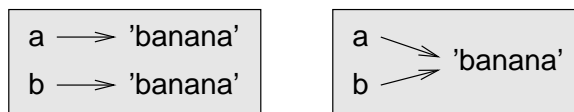
在这个例子中分割符是一个空格，`join` 在每个单词中间添加一个空格。如果不需要使用空格连接，你可以使用一个空的字符串 `''` 作为分割符。

10.10 对象和值

如果我们执行以下的赋值语句：

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

我们知道 `a` 和 `b` 都指向一个字符串，但我们不知道他们是否指向一个相同的字符串。有些两种可能：



在第一种情况中，`a` 和 `b` 指向两个有相同值的不同对象。在第二种情况中，它们指向同一个对象。

你可以使用 `is` 运算符检查两个变量是否指向同一个对象：

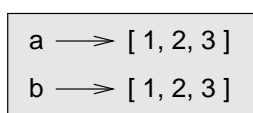
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

在这个例子中，Python 只产生了一个字符串对象，a 和 b 都指向它。

但是如果你创建两个列表，你得到两个对象：

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

状态图看起来是这样的：



在本例中，我们称这两个列表是相等的，因为它们有相同的元素，但不是相同的，因为它们不是同一个对象。如果两个对象是相同的，它们也是相等的，但是如果它们是相等的，它们不一定相同。

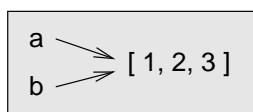
目前为止，我们可以交换地使用“对象”和“值”，但更精确地说是对象包含一个值。如果你执行 `[1,2,3]`，你会得到一个整数序列的对象。如果另一个列表有相同的元素，我们称它有相同的值，但它不是相同的对象。

10.11 别名

如果 a 指向一个对象，然后你进行赋值 `b = a`，那么两个变量都指向同一个对象：

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

状态图如图所示：



一个变量和一个对象的关联称为引用。在这个例子中，同一个对象有两个引用。

如果一个对象有多于一个引用，我们称这个对象是有别名的。

有别名的对象是可改变的，对一个别名的改动会影响另一个：

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

这个行为虽然很有用，但容易造成错误。通常，对于可改变的对象避免使用别名相对更安全。

对于不可改变的对象，如字符串，使用别名没有任何问题。例如：

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

使用 `a` 或者 `b` 指向同一个字符串基本上没有任何区别。

10.12 列表参数

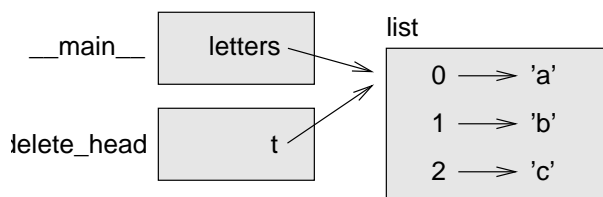
当你将一个列表作为参数传给一个函数，函数将得到这个列表的一个引用。如果函数对这个列表参数进行了修改，调用者会看见变动。例如，`delete_head` 删除列表的第一个元素：

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

它是这么使用的：

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

参数 `t` 和变量 `letters` 是同一个对象的别名。栈图如下：



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)
>>> print t1
[1, 2, 3]
>>> print t2
None
```

```
>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

如果你要编写函数修改列表，这个区别就很重要。例如，下面函数没有删除列表的第一个元素：

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

切片操作创建一个新的列表，并使 `t` 指向它。但这些操作对作为参数的列表都没有影响。

一个替代的写法是创建并返回一个新的列表。例如，`tail` 返回不包含第一个元素的列表：

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

这个函数并不修改原来的列表。下面给出如何使用这个函数：

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

Exercise 10.2 编写函数 `chop`，读取一个列表并进行修改，删除第一个和最后一个元素，并返回 `None`。

再编写函数 `middle`，读取一个列表作为参数，返回一个包含除了第一个和最后一个元素的新列表。

10.13 调试

粗心的使用列表（以及其他可改变的对象）会导致长时间的调试。下面给出一些常见的陷阱以及避免它们的方法：

1. 记住大多数的列表的方法对参数进行修改，然后返回 `None`。字符串的方法则相反，它们保留原始的字符串并返回一个新的字符串。

如果你习惯编写处理字符串的代码，如：

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

你可能会写出下面的代码：

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

由于 `sort` 返回 `None`，你对 `t` 的下一个操作可能会失败。

在使用列表的方法和运算符前，你应该仔细阅读文档，并在交互模式下测试。列表和其他序列（如字符串）共有的方法和运算符的文档在docs.python.org/lib/typesseq.html。可改变的序列独有的方法和运算符的文档在docs.python.org/lib/typesseq-mutable.html。

2. 养成自己的编码习惯。

列表中的一个问题是有太多的途径做相同的事。例如，要删除列表中的一个元素，你可以使用 `pop`，`remove`，`del` 甚至切片赋值。

要添加一个元素，你可以使用 `append` 方法或 `+` 运算符。但记住什么是正确的：


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

以下是错误的:

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

在交互模式下测试每一个例子，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，其他的都是合法的，但做了错误的事情。

3. 复制拷贝，避免别名。

如果你要使用类似 `sort` 的方法来修改参数，但同时有要保留原列表，你可以复制一个拷贝。

```
orig = t[:]
t.sort()
```

在这个例子中你还可以使用内建函数 `sorted`，它将返回一个新的已排序的列表，原列表将保持不变。注意你需要避免使用 `sorted` 作为变量名！

10.14 术语

列表： 一个序列的值。

元素： 列表（或序列）中的一个值，也称为项目。

下标： 对应列表中的元素的整数。

嵌套列表： 列表中的元素是另一个列表。

列表遍历： 对列表中的元素按顺序访问。

映射： 一个集合中的元素和另一个集合中的元素的对应关系。例如，列表是下标到元素的映射。

累加器： 循环中用于相加或累积的变量。

增量赋值： 更新变量的语句，使用类似 `+=` 的运算符。

归并： 遍历序列，将所有元素求和为一个值的处理模式。

映射： 遍历序列，对每个元素执行操作的处理模式。

筛选： 遍历序列，选出满足一定标准的处理模式。

对象： 变量可以指向的东西。一个对象有其数据类型和值。

相等： 有相同的值。

相同： 是用一个对象（隐含相等）。

引用： 变量和值间的关联。

别名： 两个或两个以上的变量指向同一个对象。

分割符： 用于指示字符串分割位置的字符或者字符串。

10.15 练习

Exercise 10.3 编写函数 `is_sorted`，读取一个列表作为参数，如果列表是升序排序的，则返回 `True`，否则返回 `False`。你可以假设（作为先决条件）列表中的元素可以用关系运算符如 `<`，`>` 等比较。

例如，`is_sorted([1,2,2])` 将返回 `True`，`is_sorted(['b','a'])` 将返回 `False`。

Exercise 10.4 两个单词是回文的，如果你可以重新排列一个的字符后可以拼写出另一个。编写函数 `is_anagram`，读取两个字符串，如果它们是回文的则返回 `True`。

Exercise 10.5 这也被称为生日悖论：

1. 编写函数 `has_duplicates`，读取一个列表作为参数，如果任何元素出现超过一次，则返回 `True`。函数不能改变原列表。
2. 如果你班级上有 23 个学生，2 个学生生日相同的概率是多少？你可以通过随即产生 23 个生日并检查匹配来估计概率。提示：你可以使用 `random` 模块中的 `randint` 函数来生成随即生日。

你可以在 wikipedia.org/wiki/Birthday_paradox 了解这个问题，你可以在 thinkpython.com/code/birthday.py 找到我的程序。

Exercise 10.6 编写函数 `remove_duplicates`，参数为一个列表，返回一个新的列表，其中只包含原列表中唯一的元素。提示：列表中的元素不一定按照原来的顺序。

Exercise 10.7 编写函数，读取文件 `words.txt`，建立一个列表，每个单词为一个元素。编写两个版本函数，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？

你可以在 thinkpython.com/code/wordlist.py 中找到我的程序。

Exercise 10.8 检查一个单词是否在单词表中，你可以使用 `in` 运算符，但这很慢，因为它按顺序查找单词。

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。

每次查找，你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 步来找到这个单词，或者知道单词不存在。

编写函数 `bisect`，参数为一个已排序的列表和一个目标值，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！

Exercise 10.9 两个单词被称为是“反转词对”，如果一个另一个的反转。编写函数，找出单词表中所有的反转词对。

Exercise 10.10 两个单词被称为是“连锁词”，如果交替的从两个单词中取出字符将组成一个新的单词¹。例如，“shoe”和“cold”连锁后成为“schooled”。

1. 编写程序，找出所有的连锁词。提示：不用列举所有的单词对。
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。

¹这个练习来自 puzzlers.org 中的一个例子。

Chapter 11

字典

字典像列表一样，但是更一般。列表的索引必须是整数，但字典的索引（键）几乎可以是任何类型。

可以把字典当作是索引集合（关键字）和值集合之间的映射。每一个关键字对应一个值。关键字和对应的值称为键-值对，或者项。

我们将构造一个英语单词及其对应西班牙语单词的字典，关键字和关键字值都是字符串。

`dict` 函数创建一个空字典。由于 `dict` 是内建函数名，所以，我们应该避免使用它作为变量名。

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

大括号`{}`，代表空字典。向字典中添加一个项，可以使用方括号：

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

这行代码创建了一个从关键字`'one'`到值`'uno'`的映射。如果再次输出字典，可以看到关键字-值对，和他们之间的冒号：

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

上面输出的格式，也是输入格式。比如，可以创建一个拥有三项的字典：

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

但是如果输出 `eng2sp`，你可能会感到惊讶：

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

关键字-值对的顺序和输入的不一样。事实上，如果在读者的机器上尝试这个例子，也可能得到不同的结果。一般来说，字典项的顺序是随机的。

但这个也不会有什么問題，因为字典的元素是不是通过索引来获取的。可以使用关键字来查询对应的值：

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

关键字 'two' 总是对应值 'dos'，所以项的顺序没有什么关系。

如果关键字不在字典中，就会抛出异常：

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

`len` 函数对于字典也是适用的。它返回键-值对的数目：

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

`in` 运算符对字典也同样使用。它显示某个键是否在字典中作为关键字（。

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

如果想查看某个值是否在字典中，可以使用方法 `values`，返回包含关键字值的列表，然后使用 `in` 运算符：

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

`in` 运算符操作列表和字典时使用不同的算法。对于列表，使用搜索算法，参考??部分。随着列表变长，搜索时间成比例增加；对于字典，Python 使用散列算法，带来一个很显著的效果：无论字典有多少项，`in` 运算符花费近乎同样的时间。在这里，我不对此做出更多的解释，详情参考 wikipedia.org/wiki/Hash_table。

Exercise 11.1 编写函数，读取 `words.txt` 文件里的单词，把他们作为字典键存储在字典里，关键字值随便是什么。然后，使用 `in` 运算符查看某个字符串是否在字典里。

如果做了练习??，可以比较一下这个实现和列表 `in` 运算符与二分搜索的速度。

11.1 把字典作为计数器

假设给定一个字符串，统计每个字母出现的次数。可以有好几种：

1. 创建 26 个变量，每个代表一个字母。然后遍历字符串，对每一个字母，增加对应对应的计数器，可以使用链条件语句。

2. 创建一个 26 元素的列表。然后把每个字符变换为数字（使用内建 `ord` 函数），把数字作为列表的索引，增加相应的计数器。
3. 创建一个字典，字母作为关键字，计数器作为对应的值。第一次遇到衣蛾字符，把它加入字典。然后可以增加相应项的值。

以上的每个方法实现同样的计算，但是实现的方法不同。

实现是实施计算的一种方法，存在某些实现比其他的要好。比如，用字典实现的好处是我们不必要实现知道哪个字母会出现在字符串里，我们只需为出现的字母分配空间。

下面是字典实现的代码：

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

函数名是 `histogram`，是统计学术语，用来直观表示频率。

函数的第一行，创建了一个空字典。`for` 循环遍历字符串。每次循环，如果字符 `c` 不在字典里，我们创建一个键为 `c`，初值为 1 的项。如果 `c` 已经在字典里，我们增加 `d[c]` 的值。

看看它是如何工作的：

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

直方图 (`histogram`) 显示，字母 'a' 和 'b' 出现一次，'o' 出现两次，等等。

Exercise 11.2 字典有一个 `get` 函数，接受一个键和一个缺省值。如果键在字典里，`get` 返回对应的值；否则返回缺省值。比如：

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

使用 `get` 编写一个精巧的 `histogram` 函数。应该不使用 `if` 语句就可以实现。

11.2 循环和字典

如果在 `for` 语句中使用字典，程序遍历字典的关键字。比如，`print_hist` 输出每个关键字和对应的值：

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

下面是输出结果：

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

可以再一次看到，关键字是无序的。

Exercise 11.3 字典有一个方法 `keys`，以列表形式返回字典的关键字。

修改`print_hist` 以字典顺序¹打印关键字和对应的值。

11.3 颠倒查询

给定一个字典 `d` 和关键字 `k`，可以很容易的查询对应的值 `v = d[k]`。这个操作叫做查询。

但是如果，给定一个 `v`，想查找对应的 `k` 该怎么办？有两个问题：第一，可能多个关键字映射到同一个值 `v`。根据实际应用，可能需要选择其中一个，也可能创建一个列表来容纳所有的关键字。第二，没有一个简单的语法来实施颠倒查询，必须使用搜索。

下面是一个函数，接受一个值，返回第一个对应于该值的关键字：

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

这个函数也是搜索的一个例子，但是使用了一个我们没有见过的特性，`raise`。 `raise` 语句引发一个异常；此处，引发一个 `ValueError` 异常，一般意味着，参数的值出现了问题。

如果我们到达了循环的末尾，意味着 `v` 没有出现在字典关键字值里，所以我们引发一个异常。

下面是一个成功颠倒查询的例子：

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

一个查询失败的例子：

¹译注：此处的字典顺序指的是字母顺序，因为英文的字典是按照字母的顺序排列的。

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

手动引发一个异常和 Python 引发异常是相同的：输出回溯路径和错误信息。

`raise` 语句接受一个详细的错误信息作为可选参数。比如：

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: value does not appear in the dictionary
```

颠倒查询比正常查询慢很多。如国必须经常颠倒查询，或者当字典变的很大时，程序的表现会很糟糕。

Exercise 11.4 修改`reverse_lookup`, 让它一列表形式返回一个对应于值 `v` 的所有关键字。 , 如果没有, 返回一个空的列表。

11.4 字典和列表

列表可以作为字典的关键字值。比如，给定一个从字母到频率映射的字典，可能想反转它，也就是说，创建一个从频率到字母映射的字典。因为可能有好几个字母的频率是一样的，所以，反转字典的关键字值应该表示成由字母构成的列表。

下面是一个反转字典的函数：

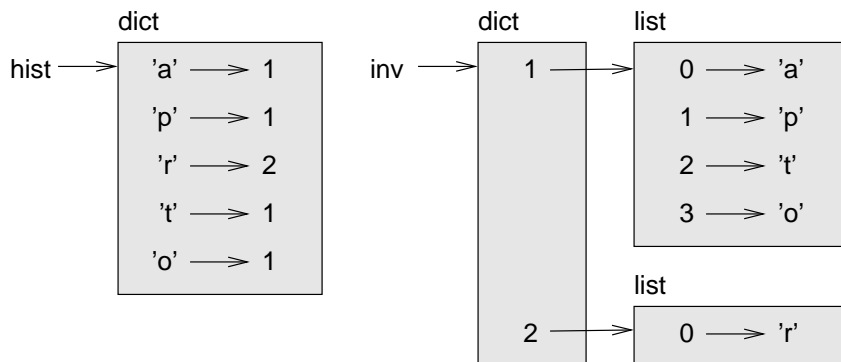
```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

每次循环，`key` 从 `d` 得到一个关键字，`val` 得到对应的值。如果 `val` 不在 `inv` 里，意味着，我们之前还没有见过它，所以我们创建一个新的项，用包含一个值的列表初始化它。否则，我们把对应的关键字追加到列表里。

下面是一个例子：

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

这里有个图表，显示了 `hist` 和 `inv` 的变化：



字典用盒子来表示，类型 `dict` 在盒子上方，键-值对在盒子里面。如果值为整型，浮点型或者字符串，通常画在盒子里面，如果是列表，则画在盒子外面，这样做仅仅是为了保持图的简洁。

列表可以作为字典的键值，正如上面的例子演示的。但是，不能作为字典的键。请看下面：

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

我在之前提到过，字典是用散列方法实现的，这就意味着，关键字必须是可散列的。

散列函数，接受一个任何类型的值，返回一个整数。字典使用这些整数（散列值）存储，查询键-值对。

如果关键字是不可变的，则一切正常。但是，如果关键字是可变的，比如列表，麻烦来了。比如，当创建一个键-值对，`Python` 散列关键字，并且把它存放在相应的位置。如果修改关键字，然后再一次散列，就会散列到另外一个位置。这种情况下，将会得到两个有着相同键的项，或者可能无法获取关键字。无论那种情况，字典都不能正常工作。

这就是为什么关键字必须是可散列的，可变数据类型像列表不能作为关键字的原因。最简单的解决方式是使用元组，我们在下一章将会遇到。

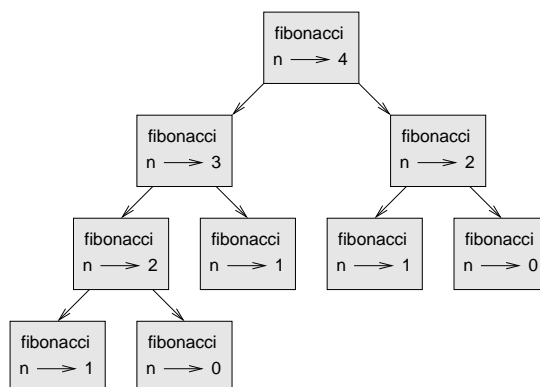
虽然字典是可变的，不能作为关键字，但是可以作为键值。

Exercise 11.5 查阅字典方法 `setdefault` 的文档，用它编写一个更简洁的 `invert_dict`。

11.5 备忘录

如果尝试过??部分的 `fibonacci` 函数，可能会注意到，提供的参数越大，函数运行花费的时间越长。确切地说，运行时间增长长的很快。

为了一探究竟，看看这个调用图，其中 `n = 4`：



调用图包含了一系列函数框图，每个框图及其调用的函数框图用直线连接。在调用图的顶部， $n=4$ 的 `fibonacci` 调用 $n=3$ 的 `fibonacci` 和 $n=2$ 的 `fibonacci`。依次地， $n=3$ 的 `fibonacci` 调用 $n=2$ 和 $n=1$ 的 `fibonacci` 函数.....

计算一下 `fibonacci(0)` 和 `fibonacci(1)` 分别被调用了多少次。可以看出，这不是解决这个问题的高效方式，并且随着参数变大，效率会更低。

另外一种方法就是跟踪已经被计算的值 --- 把它存储在字典里。存储已经计算的留作后用叫做备忘录²。下面是使用备忘录实现的 `fibonacci`:

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

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

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

`known` 是一个字典，跟踪我们已经计算出的 Fibonacci 数字。起始项为：0 映射到 0, 1 映射到 1。

无论何时调用 `fibonacci`，函数检查 `known` 字典。如果字典包含需要的结果，函数立刻返回，否则函数计算新值，并把它加入字典，然后返回。

Exercise 11.6 运行此版本的 `fibonacci` 函数和原先的版本，多传递几个参数，然后比较运行的时间。

11.6 全局变量

在以往的例子中，我们在函数外面创建 `known`，所以它属于特殊的框图 `__main__`。`__main__` 内的变量有时称为全局变量，因为任何函数都可以访问它们。不像局部、变量，在函数返回时销毁，全局变量在函数调用过程中，都是存在的。

把全局变量作为标记来使用是很常见的，也就是说，布尔变量（“标记”）表明条件是否为真。比如，有些程序使用 `verbose` 标记控制输出的层次：

²参考 wikipedia.org/wiki/Memoization。

```

verbose = True

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

```

如果试着给一个全局变量重新赋值³，我们可能会很惊讶。下面的例子跟踪函数是否被调用：

```

been_called = False

def example2():
    been_called = True          # WRONG

```

如果运行程序，将会看到`been_called` 的值并没有改变。问题在于 `example2` 创建了一个新的局部变量`been_called`。局部变量随着函数的终结而销毁，而对全局变量没有任何影响。

如果要在函数体里为全局变量重新赋值，我们必须在使用前声明全局变量：

```

been_called = False

def example2():
    global been_called
    been_called = True

```

`global` 语句告诉解释器“在这个函数里，当我说`been_called`，它就是全局变量，不要创建一个局部变量了。”

下面是一个更新全局变量值的例子：

```

count = 0

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

```

如果运行它，会得到：

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

Python 认为 `count` 是一个局部变量，这也意味着在写之前必须读取它。解决的方法是，声明 `count` 为全局变量。

```

beforeverb

def example3():
    global count
    count += 1

```

如果全局变量是可变的，就可以不声明而修改它：

```

known = {0:0, 1:1}

def example4():
    known[2] = 1

```

³译注：这个地方使用的术语是不准确的。我们知道 Python 中使用的是引用，所以不存在赋值，我们可以说重新绑定 (rebind)

所以，我们可以添加，删除，替换全局列表和全局字典的元素，但是如果想要给变量重新赋值，必须声明为全局变量：

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

11.7 长整数

如果计算 `fibonacci(50)`，我们得到：

```
>>> fibonacci(50)  
12586269025L
```

结果尾部的 `L` 表明该数是一个长整型⁴。

`int` 类型的值是有范围的；长整型值可以是任意大，但是值越大，消耗的空间和时间就越大。

基本数学运算符对长整型适用，`math` 模块的函数也如此，所以，一般来说，`int` 的代码也同样适用于 `long`。

任何时候，计算结果太大，而整型无法表示的时候，`Python` 自动把结果转换为长整型：

```
>>> 1000 * 1000  
1000000  
>>> 100000 * 100000  
10000000000L
```

第一种情况下，结果是 `int` 型；第二种情况，是 `long` 型。

Exercise 11.7 大整数幂是一般公钥加密算法的基础。阅读维基百科中 `RSA` 算法页面⁵，然后编写一个函数加密解密信息。

11.8 调试

随着程序数据的增加，通过手动输出并检查数据变得越来越笨拙。下面是关于此种情况的一些建议：

缩小输入数据量： 如果可以，尽量减少数据量。比如，如果程序想要读取一个文本文档，就可以仅仅读取开始的 `10` 行，或者其他你认为比较小的部分。可以修改文件本身，或者（最好这样）修改程序，让程序只读取文件的前 `n` 行。

如果出现错误，可以缩小 `n` 的值，到出现错误的地方，然后逐渐增加它，直到找到并更正错误。

检查概要和类型： 不必要输出检查全部数据，考虑输出数据的概要：比如，字典项的数目或者列表中数字的个数。

一个常见的运行时错误是数据类型错误。调试这种粗无，通常输出值的类型就足够了。

⁴在 `Python 3.0` 中，类型 `long` 不存在了；所有整数，即使非常大，也是 `int` 类型，或者 `long` 类型

⁵wikipedia.org/wiki/RSA.

编写自我测试：有时，可以编写代码自动检查错误。比如，计算列表中数字的平均数，可以检查结果应该不大于列表的最大元素，不大于最小元素。这个叫做“健康测试”，因为测试发现不健康的结果。

另外一种测试比较两个不同的计算结果，看看他们是否连贯。这叫做“连贯性测试”。

精巧的打印输出：格式化调试输出可以更容易发现粗无。参看 6.9 部分。pprint 模块提供 pprint 函数，以人类可读格式显示内置类型。

再次提醒：花费在搭建“脚手架”的时间越多，用来调试的时间就越少。

11.9 术语表

dictionary 字典：键集合到对应值的映射。

key-value pair 键值对：键 -- 值映射的表示。

item 项：键值对的别名。

key 键：字典中，键值对的第一个部分。

value 关键字值：字典中，键值对的第二部分。比我们以前使用的单词“值”，更特殊。

implementation 实现：计算的方式。

hashtable 散列表：实现 Python 字典的一种算法。

hash function 散列函数：计算键位置的函数。

hashable 散列体：拥有散列函数的数据类型。不可变类型，像整型，浮点型，字符串都是散列体；可变类型，比如列表和字典不是。

lookup 查询：通过关键字查找关键字值的字典操作。

reverse lookup 颠倒查询：通过关键字值查找对应关键字的字典操作。

singleton 独子体 只有一个元素的线性数据结构。

call graph 调用框图：显示在程序执行时，从调用函数到被调用函数框图的箭头的图表。

histogram 直方图：计数器的集合。indexhistogram 直方图

memo 备忘录：存储已经计算出的值，留作后用的方法。

global variable 全局变量：函数体外定义的变量。全局变量可以从任何函数中访问。

flag 标记：表明条件是否成立的布尔变量。

declaration 声明：像 global 语句一样，告诉解释器关于变量的一些信息。

section 练习

Exercise 11.8 如果做了练习??，已经编写了一个has_duplicates 函数，它接受一个列表作为参数，如果有一个对象在列表中出现多次，就返回 True。

使用字典编写一个更快，更简洁的has_duplicates 函数。

Exercise 11.9 两个单词，如果“旋转”其中一个可以得到第二个就称为“旋转对”（参看练习 8.12 `rotate_word`）

编写一个程序，读取单词列表，发现所有的旋转对。

Exercise 11.10 下面又是一个难题，来自 Car Talk⁶：

这个难题来自于 Dan O'Leary。他遇到一个常见的由一个音节，五个字母构成的单词。这个单词具有如下的性质：当移除第一个字母，剩下的字母组成了一个与原来单词发音形同的单词。如果去掉第二个字母，其他字母不变，剩下的又是一个同音词。问这个单词是什么？

现在给个不成功的例子。我们随手举个五个字母的单词为例，“wreck”。W-R-A-C-K，之，比如 `wreck with pain'。如果去掉第一个字母，剩下'R-A-C-K'。就像，`Holy cow, did you see the rack on that buck! It must have been a nine-pointer!'。它确实是一个很完美的同音词。如果去掉'r'，剩下 `wack'，它确实是一个单词，但是不是同音词了。

但是确实至少有一个符合这个条件的单词 --- 去掉前两个字母中的一个都可以构成原来单词的同音词。问题是，这个单词是什么？

可以使用练习 11.1 的字典，检查字符串是否在字典列表里。

如果要检查两个单词是否是同音词，可以使用卡内基·梅隆大学发音辞典。可以从这里下载：www.speech.cs.cmu.edu/cgi-bin/cmudict 或者从这里 thinkpython.com/code/c06d 也可以下载 thinkpython.com/code/pronounce.py，这个模块提供了函数 `read_dictionary`，读取发音辞典，返回 Python 字典，提供了从单词到对应发音（以字符串表示）的映射。

编写程序，列举能够解决这个难题的所有单词。参看我的解答 thinkpython.com/code/homophone.py。

⁶www.cartalk.com/content/puzzler/transcripts/200717.

Chapter 12

元组

12.1 元组是不可变的

元组是一组序列的值。元组中的值可以是任何数据类型，使用整数作为下标，在这个方面元组很像列表。但是一个主要的区别是元组是不可改变的。

从语法构成上来看，元组是用逗号隔开的值的序列：

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

通常用括号包含元组，虽然这不是必要的：

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

要创建只含一个元素的元组，你需要包含最后的逗号：

```
>>> t1 = 'a',  
>>> type(t1)  
<type 'tuple'>
```

在括号中的值不是元组：

```
>>> t2 = ('a')  
>>> type(t2)  
<type 'str'>
```

创建元组的另一个方式是使用内建函数 `tuple`。当没有参数时，函数创建一个空的元组：

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

如果参数是一个序列（字符串、列表或元组），返回结果是使用序列中的元素构成的元组：

```
>>> t = tuple('lupins')  
>>> print t  
( 'l', 'u', 'p', 'i', 'n', 's' )
```

由于 `tuple` 是一个内建函数的名字，你应该避免使用它作为变量名。

大多数的列表运算符适用于元组。括号运算符可以索引一个元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

切片运算符选择一个范围内的元素。

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

但是如果你试图修改元组中的元素，你会得到错误：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

你不能修改一个元组的元素，但是你可以使用另一个元组代替原来的元组：

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

12.2 元组赋值

我们经常会用到两个变量间的值的交换。对于传统的赋值，你需要使用一个临时变量。例如，要交换 `a` 和 `b`：

```
>>> temp = a
>>> a = b
>>> b = temp
```

这个方法显得笨拙，元组赋值就优雅许多：

```
>>> a, b = b, a
```

左侧是变量组成的元组，右侧是表达式组成的元组。每个值被赋值给对应的变量。右侧所有的表达式被计算后再赋值。

左侧的变量个数必须等于右侧值的个数：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

更一般的，右侧可以是任何序列（字符串、列表或元组）。例如，要分离一个 `email` 地址的用户名和域，你可以这么写：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

`split` 的返回值是两个元素的列表。第一个元素被赋值给 `uname`，第二个被赋值给 `domain`。


```
>>> print uname
monty
>>> print 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，但是如果这个值是一个元组，等效于返回多个值。例如，如果你要计算两个整数的除法并得到商和余数，分别计算 x/y 和 $x\%y$ 的效率太低，一个好的方法是同时计算这两个值。

内建函数 `divmod` 读取两个参数，返回有两个值的元组，分别是商和余数。你可以将结果保存在一个元组中：

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

或者使用元组赋值将元素分开保存：

```
>>> quot, rem = divmod(7, 3)
>>> print quot
2
>>> print rem
1
```

下面给出一个返回元组的函数的例子：

```
def min_max(t):
    return min(t), max(t)
```

`max` 和 `min` 是内建函数，分别查找序列中最大和最小的元素，`min_max` 计算两者并返回包含两个值的元组。

12.4 变长参数元组

函数可以读取一个变长的参数。一个以 `*` 开头的参数将参数聚集为一个元组。例如，`printall` 可以接收任意个数的参数，并打印它们：

```
def printall(*args):
    print args
```

聚集的参数可以取任何你喜欢的名字，但是习惯上使用 `args`。下面给出函数是如何工作的：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

和聚集相对应的是散布。如果你有一个值的序列要作为多个参数传给一个函数，你可以使用 `*` 运算符。例如，`divmod` 需要两个参数，而不是一个元组：

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

但是如果你散布元组，它将工作正常：

```
>>> divmod(*t)
(2, 1)
```

Exercise 12.1 许多内建函数使用变长参数元组。例如，`max` 和 `min` 可以读取任意个数的参数：

```
>>> max(1,2,3)
3
```

但是 `sum` 函数不是这样。

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

编写函数 `sumall`，可以接收任意多个参数，并返回它们的和。

12.5 列表和元组

`zip` 是一个内建函数，参数为两个或两个以上的序列，并将它们“拉链”成一个元组的列表，每个元组包含每个序列中的一个元素¹。

下面是一个字符串和一个列表的拉链的例子：

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

结果是一个元组的列表，每个元组包含字符串中的一个字符和列表中对应的元素。

如果序列的长度不同，结果的长度和较短的序列相同。

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

你可以在 `for` 循环中使用元组赋值来遍历一个元组的列表：

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

对于每次循环，Python 选择列表中下一个元组并将元素赋值给 `letter` 和 `number`。循环的输出是：

```
0 a
1 b
2 c
```

¹在 Python 3.0 中，`zip` 返回一个元组的迭代器，但是对于大多数情况，迭代器表现的像一个列表。

如果你结合 `zip`, `for` 和元组赋值, 你得到一个同时遍历两个 (或多个) 序列的常用写法。例如, `has_match` 读取两个序列, `t1` 和 `t2`, 如果有下标 `i` 使得 `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
```

如果你要遍历一个序列中的元素和它们的下标, 你可以使用内建函数 `enumerate`:

```
for index, element in enumerate('abc'):
    print index, element
```

循环的输出为:

```
0 a
1 b
2 c
```

12.6 字典和元组

字典有个方法称为 `items`, 返回一个元组的列表, 每个元组是键 - 值对²。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

正如你对字典所期望的, 列表中的项目没有固定的顺序。

相反的, 你可以使用一个元组列表来初始化一个字典:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

结合 `dict` 和 `zip` 给出了一个简洁的创建字典的方法:

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

字典的另一个方法 `update` 读取一个元组列表, 将它作为键 - 值对加入现有的字典。

结合 `items`, 元组赋值和 `for` 循环, 你可以得到遍历字典的键 - 值对的常用写法:

```
for key, val in d.items():
    print val, key
```

²在 Python 3.0 中稍有不同。

循环的输出为:

```
0 a
2 c
1 b
```

通常使用元组作为字典的键（主要因为你不能使用列表）。例如，电话簿是姓名对到电话号码的映射。假设我们已经定义了 `last`, `first` 和 `number`，我们可以这么写：

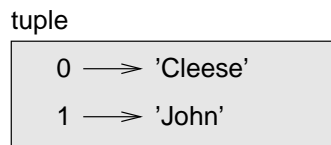
```
directory[last,first] = number
```

括号中的表达式是一个元组。我们可以使用元组赋值来遍历字典。

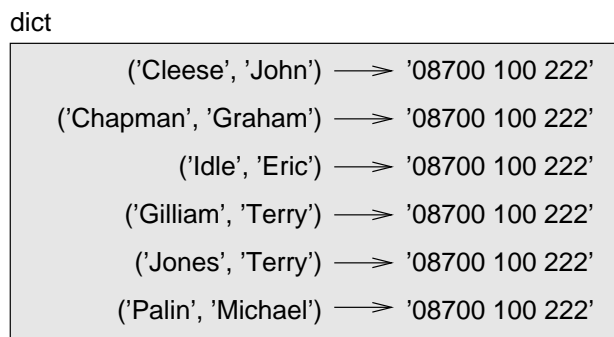
```
for last, first in directory:
    print first, last, directory[last,first]
```

这个循环遍历 `directory` 中作为键的元组。将每个元组中的元素赋值给 `last` 和 `first`，然后打印姓名和对应的电话号码。

有两种在状态图中标示元组的方式。下面的更详细的版本给出了类似列表的下标和元素。例如，元组 `('Cleese', 'John')` 将会标示成这样：



但是在一个更大的图中你也许希望忽略细节。例如，一个电话簿的状态图会是这样：



这里元组根据 Python 的语法以图形速记的形式给出。

图中的电话号码是 BBC 的投诉电话，请不要拨打。

12.7 元组比较

关系运算符使用于元组和其他序列。Python 从每个序列的第一个元素开始比较。如果它们相同，则继续比较下一个元素，依次类推，直到找到有区别的元素，以后的元素将不被考虑（即使它们很大）。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

`sort` 函数的工作原理类似，它根据第一个元素排序，如果第一个元素相同，则对第二个元素进行排序，依次类推。

这个特点对应 `DSU` 模式：

`Decorate` 装饰序列，生成元组列表，将一个或多个排序关键字放在元素的最前面。

`Sort` 对元组列表排序

`Undecorate` 通过从已排序的序列中抽出元素来还原。

例如，假设你有列单词，你需要从最长到最短将它们排序：

```
def sort_by_length(words):
    t = []
    for word in words:
        t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
        res.append(word)
    return res
```

第一个循环构造了一个元组列表，每个元组有单词长度和单词组成。

`sort` 函数首先比较第一个元素，即单词长度，仅当单词长度相同时才考虑第二个元素。关键字参数 `reverse=True` 告诉 `sort` 使用降序排序。

第二个循环遍历元组列表，构建一个按长度递减排列的单词列表。

Exercise 12.2 在这个例子中，长度相同的单词是按照字母顺序排序的。对于一些其他的应用，你也许需要它们是随机排序的。修改例子是长度相同的单词随机排序。提示：参考 `random` 模块中的 `random` 函数。

12.8 序列的序列

我主要使用元组的列表，事实上几乎本章所有的例子都还可以使用列表的列表，元组的元组和列表的元组来实现。为了避免枚举可能的组合，有时说成序列的序列更为方便。

在很多情况下，不同的序列（字符串，列表和元组）可以交换的使用。那么该如何选择，为什么这么选择呢？

显而易见的是，字符串相比其他序列受到更多的限制，因为元素必须是字符。同时它们是不可改变的。如果你需要能够修改字符串中的字符（而非创建一个新的字符串），你需要使用字符的列表。

列表比元组更为常用，主要因为它们是可改变的。但有一些情况你或许会更倾向于元组：

1. 在某些情况下，如 `return` 语句，语法上创建一个元组比创建一个列表更方便。在其他情况下，你也许更倾向列表。

2. 如果你要使用一个类似字典关键字的序列，你必须使用类似元组或字符串的不可改变的数据类型。
3. 如果你将序列作为函数参数，使用元组会减小潜在的因为别名而造成的意外的行为。

由于元组是不可改变的，它们不提供类似 `sort` 或 `reverse` 等修改列表的方法。但是 Python 提供内建函数 `sorted` 和 `reversed`，它们读取任何序列作为参数，并返回一个新的排序后的列表。

12.9 调试

列表、字典和元组通常被成为数据结构。本章中我们开始接触复合数据结构，如元组的列表、以元组作为键列表作为值的字典。复合数据结构十分有用，但它们容易造成我习惯称呼的形状错误，即由于数据结构含有错误的类型、大小或符合而造成的错误。例如，你期望得到一个只含一个整数的列表，而我给你的是一个整数（不是一个列表），这将导致错误。

为了帮助调试此类错误，我编写了一个叫做 `structshape` 的模块，提供一个 `structshape` 函数，它可以读取任何数据结构作为参数，并返回一个描述该结构的字符串。你可以在 thinkpython.com/code/structshape.py 下载。

下面给出一个简单列表的结果：

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```

更好的程序也许会输出 `list of 3 ints`，但不考虑复数相对简单。下面是一个列表的列表：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print structshape(t2)
list of 3 list of 2 int
```

如果列表中的元素不是同一数据类型，`structshape` 按类型聚合它们：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

下面是一个元素的列表：

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> print structshape(lt)
list of 3 tuple of (int, str)
```

下面是有 3 项从整数到字符串映射的字典。

```
>>> d = dict(lt)
>>> print structshape(d)
dict of 3 int->str
```

如果你在跟踪数据结构时遇到了问题，`structshape` 可以帮助你。

12.10 术语

元组：不可改变的元素的序列。

元组赋值：一个序列在右、一个元组变量在左的赋值。右边首先计算值，然后将其元素赋值给左边的变量。

聚集：对变长参数元组的集合操作。

散布：将序列作为参数列表的操作。

DSU：`decorate-sort-undecorate` 的简称，一个构建元组列表、排序、提取结果的模式。

数据结构：相关数据的集合，通常组织为列表、字典、元组等。

形状（数据结构的）：对数据结构类型、大小和组成的概述。

12.11 练习

Exercise 12.3 编写函数`most_frequent`，参数为一个字符串，以频率降序打印字符出现的次数。从不同语言的测试文本中寻找频率的不同。将你的结果和wikipedia.org/wiki/Letter_frequencies 中的表格作比较。

Exercise 12.4 更多关于回文的练习！

1. 编写函数，从文件中读取一个单词表（参考章节 ??），打印所有回文的单词。

下面的例子给出可能的输出结果：

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

提示：你也许需要构建一个字典满足从字母的集合到这些字母可以组成的单词列表的映射。问题是你如何表示这个字母集合使得它们可以作为字典的键？

2. 修改之前的程序，使程序按照结果集合的大小从大到小输出。
3. 在拼字游戏中，如果你手头的 7 个字母和桌面上的 1 个字母组成一个 8 个字母的单词，你实现了 “bingo”。哪 8 个字母组成的集合有最大的概率实现 “bingo”？提示：有 7 组。
4. 我们定义两个单词为 “置换对”，如果你能通过交换字母顺序将一个单词转换为另一个单词³。例如，“converse” 和 “conserve” 是一对 “置换对”。提示：不要测试所有的单词对，也不要测试所有的交换。

你可以在thinkpython.com/code/anagram_sets.py 下载一个解答。

Exercise 12.5 下面是另一个 Car Talk 难题⁴：

³这个练习受 puzzlers.org 中一个例子的启发。

⁴www.cartalk.com/content/puzzler/transcripts/200651。

如果每次你从一个单词中删除一个字母它仍是一个有效的英语单词，在英语中满足这样条件的最长的单词是什么？

删除的字母可以位于两端或者中间，但是你不能重新排列字母。每次你删除一个字母，你得到另一个英语单词。最终你得到一个只有一个字母的单词。我要知道最长的单词是什么，它有多少字母？

我要给出一个例子：**Sprite**。你从 **sprite** 开始删除字母，首先删除 **r**，我们得到单词 **spite**，接着删除 **e**，我们得到 **spit**，再删除 **s**，我们得到 **pit**，**it** 和 **I**。

编写程序，找出可以按这中方法缩小的所有单词，并找出最长的一个。

这个练习比以往的都更有挑战性，所以给出一些建议：

1. 你也许需要编写一个函数，参数为一个单词，函数找出所有删除一个字母后仍合法的单词，即原单词的“孩子”。
2. 递归的看，一个单词是可缩小的，如果它所有的孩子都是可缩小的。作为基本状态，你可以认为空字符串是可以缩小的。
3. 我提供的单词表 `words.txt` 中不包括单字母的单词。所以你需要添加 “I”，“a” 和空字符串。
4. 为了提高你的程序的效率，你需要记住已知的可缩小的单词。

你可以在thinkpython.com/code/reducible.py 下载我的解答。

Chapter 13

实例学习：数据结构的实例

13.1 单词频率分析

按照惯例，在参考我的答案之前，最好尝试自己做下面的练习。

Exercise 13.1 编写程序，读取文件，把每行分解为一个个单词，并从单词中去除掉空白和标点符号。然后把单词转换成大写形式。

Hint:`string` 模块提供 `whitespace` 字符串，包含了空格，制表，换行等等字符，`punctuation` 字符串包含标点符号。让我们看看，是否可以让 Python “诅咒”：

```
>>> import string
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

并且，可能会使用字符串方法 `strip`, `replace` 和 `translate`。

Exercise 13.2 打开古腾堡工程的首页 (gutenberg.net), 下载你最喜欢的过了版权期的文本书籍。

修改程序，读取下载的书籍，略过文件开头的头信息，像以前一样处理剩余的单词。

然后修改程序，统计书中单词的数目，和每个单词出现的次数。

输出书中不同单词的数目。比较不同时期，不同作者的书籍。哪个作者使用的词汇量最大？

Exercise 13.3 修改前一个练习的程序，输出书中使用频率最大的 20 个单词。

Exercise 13.4 修改前面程序，读取单词列表（参看 9.1 部分），输出书中所有不在单词列表的单词。他们中有多少是排版错误？有多少是常见的单词，有多少是生僻单词？

13.2 随机数字

给定同样的输入，大多数的计算机程序每次产生同样的输出，所以，他们是确定的。确定性通常是一件好事儿，因为我们希望同样的计算产生同样的结果。对于某些程序，我们却希望产生不可确定的结果。游戏就是一个很显然的例子，但是远远不止这一个例子。

编写一个真正具有不确定性的程序不是一件容易的事儿。但是，有一些方法可以使它看上去是不确定的。一种是使用产生伪随机数算法。伪随机数字并不是真正的随即，因为他们是通过确定的计算产生的，但是如果仅仅观察数字，是不可能把他们和真正随机数区分开来。

`random` 模块提供了可以产生伪随机数的函数（我们一下都用“随机数”来代称）。

`random` 函数返回一个从 0.0 到 1.0 的随机浮点数（包括 0.0，但是不包括 1.0）。每次，调用 `random`，就会的到一个随机数。看下面的循环：

```
import random

for i in range(10):
    x = random.random()
    print x
```

`randint` 函数接受一个 `low` 和 `high` 参数，返回 `low` 和 `high` 之间的一个整数（包括两者）。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

要想随机地从一个序列中选取元素，可以使用 `choice`：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

`random` 模块还提供了从连续分布中产生随机数函数，包括，高斯函数，指数函数， γ 分布，等等。

Exercise 13.5 编写函数 `choose_from_hist`，接受一个直方图（在 11.1 部分定义的），然后从直方图里随机返回一个数，和对应的频率。比如：

```
>>> t = ['a', 'a', 'b']
>>> h = histogram(t)
>>> print h
{'a': 2, 'b': 1}
```

函数必须是 'a' 及频率 2/3, 和 'b' 及频率 1/3。

13.3 单词频率直方图

下面的程序读取文件，建立文件中单词的频率直方图。

```
import string

def process_file(filename):
    h = dict()
```

```

    fp = open(filename)
    for line in fp:
        process_line(line, h)
    return h

def process_line(line, h):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        h[word] = h.get(word, 0) + 1

hist = process_file('emma.txt')
```

程序读取 `emma.txt`---是简·奥斯汀的小说《爱玛》。

`process_file` 循环遍历获取文件的每行，然后传递给 `process_line`。直方图 `h` 作为累加器使用。

`process_line` 先使用字符串方法 `replace` 把连字符替换为空格，然后使用 `split` 把每行分隔为字符串列表。接着，遍历单词列表，使用 `strip` 和 `lower` 剔除标点并把字符串转换为小写形式。（这里说“转换”是简略的说法。记得我们说过字符串是不可变的，所以方法 `strip` 和 `lower` 等返回新的字符串。）

最后，`process_line` 通过创建新的项或者增加已存在项的值，更新直方图，

要计算文件中单词的总数，可以计算直方图的频数之和：

```
def total_words(h):
    return sum(h.values())
```

单词量等于字典项的数目：

```
def different_words(h):
    return len(h)
```

下面是输出结果的代码：

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

结果为：

```
Total number of words: 161073
Number of different words: 7212
```

13.4 最常用单词

若要找出最常用的单词，我们可以应用 `DSU` 模式；`most_common` 接受一个直方图，返回（单词—频数）列表，频数由小到大顺序排序。

```
def most_common(h):  
    t = []  
    for key, value in h.items():  
        t.append((value, key))  
  
    t.sort(reverse=True)  
    return t
```

下面是通过循环实现输出最常用的十个单词代码：

```
t = most_common(hist)  
print 'The most common words are:'  
for freq, word in t[0:10]:  
    print word, '\t', freq
```

《爱玛》中，最常用的十个单词是：

```
The most common words are:  
to      5242  
the     5204  
and     4897  
of      4293  
i       3191  
a       3130  
it      2529  
her     2483  
was     2400  
she     2364
```

13.5 可变参数

我们已经遇到接受可变参数¹的内置函数和方法。我们也是可以自定义带有可变参数的函数。比如，下面是输出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[0:num]:  
        print word, '\t', freq
```

第一个参数是必须的；第二个参数是可选的。`num` 的缺省值是 10。

如果只提供一个参数：

```
print_most_common(hist)
```

`num` 就是缺省值。如果提供两个参数：

```
print_most_common(hist, 20)
```

`num` 使用传递来的参数。换句话说，可选参数覆盖了缺省值。

如果函数拥有可变参数，所有必需参数必须首先赋值，最后才能是可选参数。

¹译注：英文为 *optional argument*，直译为可选择的参数，英文侧重于每一个参数，而中文翻译侧重为参数整体

13.6 字典减法

从书中找出不在单词列表 `words.txt` 的单词，可以看做是集合减法运算；也就是说，我们从一个集合（书中单词）找出不在另一个集合的单词（单词表里）。

`subtract` 接受 `d1` 和 `d2` 两个字典，返回一个包含所有不在 `d2` 里的 `d1` 中的单词为关键字构成的字典。这里，我们不必关心关键字值大小，所以我们把关键字值设为 `None`。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

发现书中不在 `words.txt` 中单词，我们可以使用 `process_file` 为 `words.txt` 建立一个直方图，然后相减：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

下面是处理《爱玛》后的结果：

```
The words in the book that aren't in the word list are:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

有些单词是事物（人，动物，地名）名字。有些，像“`rencontre`”不是很常见。但是有些常见的单词没有包含在单词列表中。

Exercise 13.6 Python 提供了 `set` 数据结构，拥有一些常见的集合原算。阅读官方文档（docs.python.org/lib/types-set.html），编写程序使用集合减法，查找书中不在单词列表里的单词。

13.7 随机单词

要从直方图中随机选取一个单词，最简单的算法是依据每个单词的频数，创建一个列表，列表中每个单词出现的次数等于频数。

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

表达式 `[word] * freq` 创建一个列表，列表包含了 `freq` 个字符串 `word`。`extend` 方式和 `append` 方法类似，除了它的参数是一个序列。

Exercise 13.7 这个算法是有效的，但是不是很高效。每次，随机选择一个单词，就会重新创建一个列表（近乎和原书一样大小）。一个显著的改进方案是只创建一次列表，然后多次选择，然而列表还是很大。

一个替代方案：

1. 使用 `keys` 获取书中单词列表。
2. 创建一个包含单词频数累计和的列表（参看练习??）。最后一项是书中单词的总数目， n 。
3. 随机选择 1 到 n ，内的任意整数。使用二分搜索（参看练习??），查找插入随机数的索引。
4. 使用索引查找单词列表里对应的单词。

编写程序使用此算法从书中随机生成一个单词。

13.8 马尔可夫分析

如果从书中随机选择单词，只能得到词汇，但不能得到句子。

`this the small regard harriet which knightley's it most things`

连续的随机单词没有什么意义，因为这些连续的单词之间并没有什么语义上的关系。比如，在真正的句子中，“the”后面通常跟着形容词或名词，不会跟着动词或副词。

检测这些联系的方法之一是马尔可夫分析²，它刻划了，对于给定的一系列单词的下一个单词的可能性。比如，歌曲 *Eric, the Half a Bee*:

```
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?
```

在歌词中，“bee”，总是跟在短语“half the”后面，但是短语“the bee”的后面跟着“has”或“is”。

马尔可夫分析的结果是从前缀（像“half the”和“the bee”）到所有可能的后缀（像“has”和“is”）的映射。

给定这个映射，就可以产生一篇随即文本 --- 从任何前缀开始，然后随机选取可能的后缀。接着，可以结合当前前缀的后部分和后缀形成一个新的前缀，如此反复。

²这个实例分析源自 Kernighan and Pike, *The Practice of Programming*, 1999 的一个例子

比如，从前缀“Half a”开始，下一个单词必须是“bee”，因为前缀仅仅在歌词中出现一次。下一个前缀是“a bee”，所以下一个后缀可能是“philosophically”，“be”或者“due”。

在这个例子中，前缀的长度总是为 2，但是你可以使用任何长度的前缀。前缀的长度称为，分析的“阶”。

Exercise 13.8 马尔可夫分析：

1. 编写程序读取文档内容，然后进行马尔可夫分析。结果为从前缀到可能后缀集合³的映射(字典)。集合可以是列表，元组或者字典，这个完全由你自己做主。可以使用长度为 2 的前缀测试程序，但是必须编写程序使得很容易尝试其他长度的前缀。
2. 向之前的程序添加一个函数，依据马尔可夫分析随机生成一段文本。下面是一个使用前缀长度为 2 的例子：

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.

对于这个例子，我把标点附加在单词后面。从句法上看，结果大多正确，但是不是全部符合规范。从语义上看，情况亦如此。

如果增加前缀的长度，情况会如何？随机生成的文本会更有意义吗？

3. 一旦程序可以工作，或许想尝试一下混合的效果：如果分析两本或者更多的书籍，随机产生的文本将会以一种有趣的现象混合了不同作者的词汇和短语。

13.9 数据结构

使用马尔可夫分析生成随机文本非常有意思，但是这个练习有一个要点：数据结构的选择。前面的练习，必须选择：

- 如何表示前缀。
- 如何表示可能后缀的集合⁴。
- 如何表示从前缀到后缀集合的映射。

恩，最后一个是很简单的，我们至今学到的映射数据类型是字典，所以自然选择它了。

对于前缀，最明显的选择就是使用字符串，字符串列表，或者字符串元组。对于后缀，一种选择是列表，另外一种直方图（字典）。

应该如何选择数据结构呢？第一步，思考要为数据结构实现的操作。对于前缀，我们需要能够从前部删除单词，添加到后部。比如，如果当前的前缀是“Half a”，下面一个单词是“bee”，需要能够形成下一个前缀“a bee”。

第一个想到的可能是列表，因为列表很容易实现添加和删除元素，但是我们需要能够把前缀作为字典的键字使用，所以排除列表。对于元组，不能实现添加和删除，但是可以使用“+”运算符构建新的元组。

³译注：collection，不是 set

⁴译注：collection

```
def shift(prefix, word):  
    return prefix[1:] + (word,)
```

`shift` 接受单词元组，`prefix` 和一个字符串 `word`，创建一个新的元组，包含 `prefix` 中除第一个以外的全部单词，`word` 被添加到新元组的尾部。

对于后缀集合，我们需要的操作包括添加一个新后缀（或者增加已存在后缀的频数），和随机选择一个后缀。

添加一个新后缀对于列表实现或者直方图实现都是很容易的。从列表中随机选择一个元素也是很容易的；但从直方图中选择，就不是很高效率（参看练习 13.7）。。

迄今为止，我们已经谈论的大多是容易实现，但是还有其他因素在选择数据结构时。一种是运行时间。有时，理论上，希望一种数据结构比其他的要快。比如，我提到，`in` 操作符，至少在元素数目很大时，用于字典比列表要快。

但是，通常我们之前并不知道哪种实现快？一种方式是实现两种，测试一下哪个更好。这种方式叫做基准程序。一个更实用的方法是选择最容易实现的数据结构，看看对于要实现的应用程序是否足够快。如果是，没有必要再实现其它数据结构。如果不是，有其他的程序，像 `profile` 模块，可以确定程序中花费时间最长的部分。

另外一种需要考虑的因素是存储空间。比如，使用直方图存储后缀集合需要极小的空间，因为只需要存储每个单词一次，无论它在文本中出现多少次。某种情况下，节省存储空间也可使得运行速度变快。极端地说，如果你使用完内存，程序根本无法运行。但是，对于很多应用程序，空间在运行后是次要考虑因素。

最后思考一下：此探讨中，我隐示地说明，在分析和生成阶段使用同一个数据结构。但是，因为是分离的阶段，所以，可能使用一种数据结构分析，然后转换为另外一种数据结构生成。如果生成的时间超过转换的时间，这种方式也是不错的选择。

13.10 调试

当调试程序时，特别是调试一个极难“对付”的 `bug` 时，我们可以尝试下面的四种方法：

阅读： 重新检查代码，确保程序表达了自己的意愿。

运行： 修改程序，运行不同版本的程序。通常，如果在适当的位置，输出了预计的东西，问题就很明显，但可能有时候，需要花些时间构建脚手架。

反思： 花些时间反思！到底是出现了什么错误：语法，运行时，还是语义粗我？可以从错误信息中或者程序输出得到什么信息？什么错误可能导致遇到错误？问题出现前，最后一次改动是什么？

“退后”： 有时候，最好的办法就是恢复改变，直到程序能工作，并且很容易理解。然后，就可以重建构建程序。

初级程序员有时只专注于一种方法，忽略了其他的方法。每种方法都有失效的时候。

比如，如果问题粗出在印刷错误上，阅读程序会有帮助，但如果是概念性误解，就毫无帮助了。如果自己都不理解自己的程序，就算是读上百遍，也看不到什么错误。因为错误在大脑里。

做实验是很有效的，特别是运行简单的测试。但是，如果在实验时，不加思考，不阅读自己的程序，很可能会落入“瞎猫碰死耗子编程”模式⁵---随意做出改变，直到程序得出正确的结果的过程。毋需说，“瞎猫碰死耗子编程”是费时，费力的。

必须花费时间思考。调试就像做科学实验。必须首先假设出现了什么问题。如果有两种或多种可能性，那就测试一下，排除一个。

适当的休息一下，有助于思考。探讨也是。如果向其他人（甚至是自己）解释问题，有时就会在在问问题的过程中，得到答案。

但是，如果程序有太多的错误，或者要调试的代码太大，太复杂，甚至最好的调试技巧都会失败。有时，最好的办法就是退后，简化程序，直到程序能够运行，并且很容易理解（至少自己能够理解）。

初级程序员通常不情愿退后，因为，他们不能忍受删除一行代码（尽管它是错误的）。如果实在心有不舍，把程序复制一份，然后退后。这样，随后就可以一点点的粘贴回来

发现一个隐藏的 `bug` 需要阅读，运行，反思，有时还需要退后。如果有一种方式行不通，使用其它的方法。

13.11 术语表

deterministic 确定的： 给定相同的输入，程序每次运行时，做同样的事情，给出同样的输出。

pseudorandom 伪随机： 由确定性程序生产，表面上是随机的数字序列。

default value 缺省值： 如果没有参数提供，由可选参数提供的值。

override 覆盖： 使用参数代替缺省值。

benchmarking 基准程序： 实现候选的数据结构，给定一定的输入进行测试的过程。

13.12 练习

Exercise 13.9 单词的“等级”就是在依照频数排序的单词列表里的位置：最常见的单词是得等级 1，此常见单词是等级 2，等等。

Zipf 法则用自然语言描述了单词等级和频数之间的关系⁶。特别地，它预测了等级为 r 的单词的 f 频数：

$$f = cr^{-s}$$

s 和 c 是依赖于语言和文本的参数。如果，对等式两边取对数，得到：

$$\log f = \log c - s \log r$$

如果绘制 $\log f$ -- $\log r$ ，图像，就会得到一条斜率为 $-s$ ，截距为 $\log c$ 的直线。

编写程序，读取文件，计算单词频数，按频数降序排列，每行打印一个单词，和对应 $\log f$ 和 $\log r$ 。使用绘图程序绘制结果，并且检查看看是否形成直线。你能够估测 s 的值吗？

⁵译注：random walk programming

⁶参看 wikipedia.org/wiki/Zipf's_law.

Chapter 14

文件

14.1 持久性

目前我们所见到的的大多数的程序都是瞬态的，即它们在短时间内运行并输出一些结果，当它们结束时，数据也就消失了。如果你再次运行程序，它将从一个初始的状态开始。

另一类程序被称为是持久的，它们长时间运行（或者时刻运行），至少将一部分数据记录在永久储存设备（如硬盘）上，当程序关闭并重新启动时，它们可以恢复结束前的状态以便继续运行。

一个持久的程序的例子是操作系统，当一台电脑开机后操作系统在绝大多数时间都在运行，对于一个接受网络请求的 **web** 服务器，操作系统时刻在运行。

程序维护数据最简单的方法是读写文本文件。我们已经接触过读取文本文件的程序，在本章中我们将接触写入文本文件的程序。

记录程序状态的另一个方式是使用数据库。在本章节中我给出一个简单的数据库和模块 **pickle**，该模块简化了存储数据的过程。

14.2 读取和写入

文本文件是储存在类似硬盘、闪存、CD-ROM 等永久介质上的字符序列。我们在章节 ?? 中接触了文件的打开和读取。

要写入一个文件，你需要在打开文件时添加第二个参数 **'w'**：

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

如果文件已经存在，以写的模式打开该文件将会清空原来的数据并从新的开始，所以要小心！如果文件不存在，那么将创建一个新的文件。

write 方法将数据写入文件。

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

文件对象将跟踪位置，如果你再次调用 `write`，它将在尾部写入新的数据。

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

当你完成写入，你可以关闭文件。

```
>>> fout.close()
```

14.3 格式运算符

`write` 的参数必须是字符串，如果我们要将其他值写入文件，我们需要将它们转换为字符串。最简单的方法是使用 `str`：

```
>>> x = 52
>>> f.write(str(x))
```

另一个方法是使用格式运算符`%`。当作用于整数，`%` 是取模运算符，而当第一个运算数是字符串时，`%` 是格式运算符。

第一个运算数是格式字符串，它包含一个或多个格式序列，它们指定了第二个运算数是如何格式化的。结果为一个字符串。

例如，格式序列`'%d'` 意味着第二个运算数应该被格式化为一个整数（`d` 代表 “decimal”）：

```
>>> camels = 42
>>> '%d' % camels
'42'
```

结果是字符串`'42'`，需要和整数值 `42` 区分开来。

格式序列可以出现在字符串中的任何位置，所以你可以将值嵌入到语句中：

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

如果字符串中有多于一个格式序列，第二个参数必须为一个元组。每个格式序列按次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'` to format a floating-point number (don't ask why), and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

元组中元素的个数必须等于字符串中格式序列的个数。同时，元素的类型必须符合对应的格式序列。

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

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。

格式运算符十分强大，但它很难使用。你可以在docs.python.org/lib/typeseq-strings.html 阅读更多有关的内容。

14.4 文件名和路径

文件以目录（也称为“文件夹”）的形式管理。每个运行的程序有一个“当前目录”，它是许多操作的默认目录。例如，当你打开一个文件来读取数据，Python 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()
>>> print cwd
/home/dinsdale
```

cwd 代表“current working directory”，即当前工作目录。在本例中结果是/home/dinsdale，它是用户名为 dinsdale 的主目录。

类似 cwd 能够确定一个文件的字符串称为路径。相对路径从当前目录开始，绝对路径从文件系统的根目录开始。

我们现在看到的路径都是简单的文件名，因此它们是相对当前目录的。要得到一个文件的绝对目录，你可以使用 os.path.abspath：

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

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，os.path.isdir 检查它是否是一个目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

类似的，os.path.isfile 检查是否是一个文件。

os.listdir 返回给定目录下的文件（以及其他目录）：

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

为了演示这些函数，下面的例子“遍历”一个目录，打印所有文件的名称，并对所有目录递归地调用自身。

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)

        if os.path.isfile(path):
            print path
        else:
            walk(path)
```

`os.path.join` 读取一个目录名和一个文件名，并将两者合并为一个完整路径。

Exercise 14.1 修改 `walk` 函数，使之返回文件名列表，而不是打印这些信息。

Exercise 14.2 `os` 模块提供了与我们的 `walk` 函数类似的函数，但功能更丰富。阅读文档，使用该函数打印给定目录下的文件和子目录。

14.5 捕获异常

当你尝试读写文件时，很多地方会发生错误。如果你试图打开一个不存在的文件，你会得到一个输入输出错误：

```
>>> 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')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

如果你试图读取一个目录，你会得到：

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

为了避免这些错误，你可以使用类似 `os.path.exists` 和 `os.path.isfile` 的检查函数，但是检查所有可能的错误会占用很多时间和代码（如果“Errno 21”是一个错误信息，那么至少有 21 种出错情况）。

更好的方法是当问题出现了才去处理，即 `try` 语句所做的。它的语法类似 `if` 语句：

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Python 从 `try` 语句开始执行，如果一切正常，那么 `except` 将被跳过。如果发生异常，则跳出 `try` 语句块，执行 `except` 中的代码。

使用 `try` 语句处理异常被称为捕获异常。在本例中，`except` 语句块中的代码仅仅打印了错误信息。通常，捕获异常给了你修补问题的机会，你可以继续尝试，或者至少可以优雅地结束程序。

14.6 数据库

数据库是用于存储数据的文件。大多数的数据库以字典的形式组织，即将键映射为值。数据库是保存在磁盘（或其他永久存储设备）上，因此即使程序结束它们仍然存在。

模块 `anydbm` 提供了创建和跟新数据库文件的接口。作为一个例子，我将创建一个包含图片文件标题的数据库。

打开数据库和其他文件类似：

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

模式 `'c'` 代表如果文件不存在则创建文件。返回结果是一个数据库对象，它可以像字典一样被使用（对于大多数的操作）。如果你创建一个新的项目，`anydbm` 将更新数据库文件。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

当你访问某个项目是，`anydbm` 将读取文件：

```
>>> print db['cleese.png']
Photo of John Cleese.
```

如果你对已有的键再次进行赋值，`anydbm` 将替代旧的值：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

许多字典的方法，如 `keys` 和 `items`，同样适用于数据库对象，包括使用 `for` 语句实现的迭代：

```
for key in db:
    print key
```

和其他文件一样，当你完成操作后你需要关闭文件：

```
>>> db.close()
```

14.7 Pickling

`anydbm` 模块的一个限制在于键和值必须是字符串。如果你尝试使用其他数据类型，你会得到一个错误。

`pickle` 模块可以解决这个问题。它能将任何类型的对象翻译成适合在数据库中储存的字符串，同时能将字符串还原成对象。

`pickle.dumps` 读取一个对象作为参数，并返回一个表征字符串（`dumps` 是“dump string”（转储字符串）的缩写）：

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

这个格式对人类读者来说不是很好理解，但是对 `pickle` 来说很好解释。`pickle.loads` (“载入字符串”) 可以重建对象：

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

虽然新的对象和老的对象有相同的值，它们（通常）不是同一个对象：

```
>>> t1 == t2
True
>>> t1 is t2
False
```

换言之，`pickling` 然后 `unpickling` 等效于复制一个对象。

你可以使用 `pickle` 在数据库中存储一个非字符串对象。事实上，这个组合非常常用，并有一个已经封装好的模块 `shelve`。

Exercise 14.3 如果你做了章节 12.4 中的练习，修改你的方案，创建一个数据库，将列表中的单词映射为使用同样字母的单词的列表。

编写另一个程序，读取数据库并以人类适宜阅读的格式打印内容。

14.8 管道

大多数的操作系统提供一个命令行的接口，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` 来启动一个网页浏览器。

任何你在 `shell` 中可以启动的程序也可以在 `Python` 中通过使用管道来启动。一个管道是一个表示活动进程的对象。

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)
```

参数是一个包含 `shell` 命令的字符串。返回值是一个对象，就像打开一个文件一样。你可以使用 `readline` 每次从输出读取一样，或者使用 `read` 一次读取所有内容：

```
>>> res = fp.read()
```

当你完成操作后，你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` 表示它正常结束（没有错误）。

管道的一个常用用法是增量的读取一个压缩文件，即不需要一次解压整个文件。下面的函数读取一个压缩文件名作为参数，使用 `gunzip` 来解压文件，并返回一个管道：

```
def open_gunzip(filename):
    cmd = 'gunzip -c ' + filename
    fp = os.popen(cmd)
    return fp
```

如果你每次从 `fp` 读取一行，你不需要将解压的文件保存在内存或磁盘上。

14.9 编写模块

任何包含 Python 代码的文件可以作为模块被导入。例如，假设你有文件 `wc.py`，内容如下：

```
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，并打印行数，结果是 7。你也可以这样导入模块：

```
>>> import wc
7
```

现在你有一个模块对象 `wc`：

```
>>> print wc
<module 'wc' from 'wc.py'>
```

这里提供了一个称为 `linecount` 的函数：

```
>>> wc.linecount('wc.py')
7
```

以上就是如何编写 Python 模块。

这个例子中唯一的问题在于当你导入模块后，最后的测试代码被执行。通常当你导入一个模块，它定义新的函数，但并不执行它们。

作为模块的程序通常写成一下结构：

```
if __name__ == '__main__':
    print linecount('wc.py')
```

`__name__` 是一个程序开始时设置的内置变量。如果程序以脚本的形式运行，`__name__` 的值为 `__main__`，在这种情况下，测试代码将被执行。否则，如果是作为模块被导入，测试代码将被忽略。

Exercise 14.4 将例子输入到文件 `wc.py` 中，并以脚本形式运行。如果在 Python 解释器中运行 `import wc`，当模块被导入后，`__name__` 的值是什么？

警告：当你再次导入一个已经导入的模块，Python 将什么也不做。它不会重新读取文件，即使文件发生了改变。

如果你要重载一个模块，你可以使用内建函数 `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，你也许会遇到空白带来的问题。由于空格符、`tab` 符和换行符通常是不可见的，这样的错误很难调试：

```
>>> 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。它读取一个对象作为参数，并返回一个表示这个对象的字符串。对于字符串，它将空白符号用反斜杠序列表示：

```
>>> print repr(s)
'1 2\t 3\n 4'
```

这个对调试很有用。

你也许会遇到另一个问题，不同的操作系统使用不同的符号作为换行符。有的系统使用 `\n`，有的使用 `\r`，有的使用两者。如果你在不同系统中使用，这些差异会导致问题。

大多数系统提供了格式转换的程序。你可以在 wikipedia.org/wiki/Newline 中找到（并阅读更多相关内容）。当然你可以自己编写一个转换程序。

14.11 术语

持久性： 一个长期运行、并至少将一部分数据保存在永久性的储存设备上的程序。

格式运算符： 运算符 `%`，参数为一个格式字符串和一个元组，生成一个按格式字符串规定的元组中元素的值的字符串。

格式字符串： 使用格式运算符，包含格式序列的字符串。

格式序列： 格式字符串中的字符序列，类似 `%d`，指定了一个值的格式。

文本文件： 保存在类似硬盘的永久储存设备上的字符序列。

目录： 一个命名的文件的集合，也称为文件夹。

路径： 用于识别文件的字符串。

相对路径： 从当前目录开始的路径。

绝对路径： 从文件系统顶部开始的路径。

捕获： 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： 一个类似字典使用键对应值的文件。

14.12 练习

Exercise 14.5 `urllib` 模块提供了操作 URL 和从互联网下载信息的方法。下面的例子从 `thinkpython.com` 下载并打印一条秘密信息：

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn.fp:
    print line.strip()
```

运行程序，运行结果将给你下一步指令。

Exercise 14.6 在一个有很多 MP3 文件的收藏中，有可能同一首歌有多个拷贝，以不同的名字保存在不同的目录下。这个练习的目的是找出这些拷贝。

1. 编写程序，递归地搜索一个目录和所有子目录，返回一个完整路径的后缀给定的（如 `.mp3`）的文件名列表。提示：`os.path` 提供了几个有用的操作文件和路径名的函数。
2. 为了识别拷贝，你可以使用哈希函数，读取文件并生成一个针对内容的简短的概述。例如，MD5 (Message-Digest algorithm 5) 读取一个任意长的“消息”并返回一个 128 比特的“校验和”。两个不同文件返回相同的校验和的概率非常小。

你可以在 wikipedia.org/wiki/Md5 了解有关 MD5 的知识。在一个 Unix 系统上你可以使用 `md5sum` 程序和 Python 中的管道来计算校验和。

Exercise 14.7 网络电影数据库 (IMDb) 是一个在线的电影信息收集的网站。它们的数据库可以以纯文本的格式获得，便于 Python 的读取。在这个练习中，你需要文件 `actors.list.gz` 和 `actresses.list.gz`，它们可以从 www.imdb.com/interfaces#plain 下载。

我编写了一个程序，可以解析这些文件并分割为演员名、电影标题等。你可以在 thinkpython.com/code/imdb.py 下载。

如果你已脚本方式运行 `imdb.py`，程序将读取 `actors.list.gz` 并在每行答应演员 - 电影对。或者你可以 `import imdb`，调用函数 `process_file` 来处理这些文件。参数为一个文件名，一个函数对象和一个可选的指定处理行数的数。下面给出一个例子：

```
import imdb

def print_info(actor, date, title, role):
    print actor, date, title, role

imdb.process_file('actors.list.gz', print_info)
```

当你调用 `process_file`，它打开 `filename`，读取内容，并对文件中的每行调用 `print_info`。`print_info` 读取演员、日期、电影标题和角色作为参数，并打印这些信息。

1. 编写函数，读取 `actors.list.gz` 和 `actresses.list.gz`，使用 `shelve` 来构建一个数据库，将每个演员映射到他或她的电影列表。
2. 两个演员称为是“合演”的，如果他们至少一起演过一部电影。基于上一步建立的数据库，构建第二个数据库，将每个演员映射到他或她“合演”的演员列表。

3. 编写程序，实现 “Six Degrees of Kevin Bacon”，你可以在wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon 中了解相关信息。这个问题的挑战在于你需要在关系图上找到最短路径。你可以在wikipedia.org/wiki/Shortest_path_problem 里阅读最短路径算法相关的资料。

Chapter 15

类和对象

15.1 自定义类型

我们已经使用了很多 Python 的内置数据类型；现在我们将要定义自己的数据类型。作为一个例子，我们将要创建一个 `Point` 类型，代表二维空间的一个点。

在数学记法中，点通常用写在括号里的坐标表示，中间用逗号隔开。比如， $(0,0)$ 代表原点， (x,y) 代表距离原点右边为 x ，上面为 y 的点。

在 Python 中，我们有多种表示点的方式。

- 我们可以把两个坐标存储在两个变量里，`x` 和 `y`。
- 我们可把坐标存储在列表或者元组里。
- 我们可以创建新的数据类类型来代表点。

创建型的数据类型相比于其他方式来说有些许困难，但是优势也即将显现。

用户¹自定义类型也叫做类。类的定义是这样：

```
class Point(object):  
    """represents a point in 2-D space"""
```

类头表明新的类是 `Point`，它也是一种对象。对象也是一种内置的数据类型。

类体是文档字符串（`docstring`），解释了 `Point` 类的作用。我们可以在类里定义变量和函数，我们一会儿就会涉及到。

定义一个 `Point` 类，也就创建了一个类对象²。

```
>>> print Point  
<class '__main__.Point'>
```

因为 `Point` 是在顶级定义的，他的“全名”是 `__main__`。

类对象就像是“制造”对象的工厂。为了创建一个类，我们可以像函数一样调用 `Point`。

¹译注：这里的用户指的是程序员，而不是终极用户。一般情况下用户的意思可以通过上下文来辨别。

²译注：class object, 类对象，不是类的对象

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

返回值是一个 `Point` 对象的引用，我们把它赋给 `blank` 变量。创建一个新的对象叫实例化，对象是类的一个实例。

当打印一个对象，`Python` 告诉我们它是属于哪个类的，还有它存储的内存地址（前缀 `ox` 表示后面的数字是十六进制的）。

15.2 属性

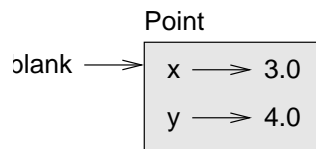
我们可以使用点记法实例赋值。

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

使用的语法和从模块里选择一个变量类似，就像 `math.pi` 或者 `string.whitespace`。这里，我们给实例的元素赋值。这些元素叫做属性。

作为一个名词，“AT-trib-ute” 的第一个音节要发重音，和作为动词时 “a-TRIB-ute,” 相反。

下面的图表显示了赋值后的结果。显示对象及其属性的状态图乘坐对象图。



`blank` 变量指向一个包含两个属性的点的对象。每一个属性指向一个浮点数。

我们可以通过同样的语法读取属性的值。

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

表达式 `blank.x` 意思是，“到 `blank` 指向的对象去，获取 `x` 的值。此时，我们把取得的值赋给变量 `x`。变量 `x` 和属性 `x` 没有冲突。

我们可以在任何表达式中使用点记法。比如：

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print distance
5.0
```

也可以把一个实例作为参数传递。比如：

```
def print_point(p):
    print '(%g, %g)' % (p.x, p.y)
```

`print_point` 接受一个点作为参数，并用数学记法显示它。我们可以把 `blank` 传递给它：

```
>>> print_point(blank)
(3.0, 4.0)
```

在函数里，`p` 是 `blank` 的别名，所以如果函数改变了 `p`, `blank` 也改变了³。

Exercise 15.1 编写一个函数 `distance` 接受两个点作为参数，返回两个点之间的距离。

15.3 矩形

有时，很明显就可以看出对象需要什么属性，但是有时候，必须劳神费心一番。比如，想象一下，你设计一个类来代表矩形。你会用什么属性来指定矩形的位置和大小？忽略角度，也为了简单，假设举行是水平和竖直的。

至少有两种可能：

- 你可以指定矩形的一角（或者中心），宽度和高度。
- 也可以指定对立的两个角。

此时，很难说明两种方式孰优孰劣。那我们就实现第一种把，仅仅作为一个例子。

下面是一个类的定义：

```
class Rectangle(object):
    """represent a rectangle.
       attributes: width, height, corner.
    """
```

文档字符串列举了所有的属性：`width` 和 `height` 是数字，`corner` 是一个类的对象，代表了左下角。

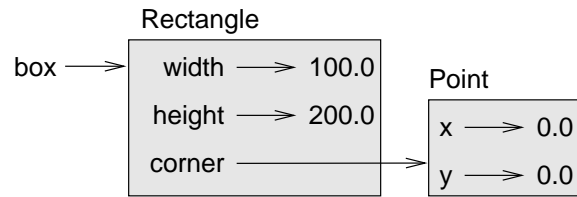
为了代表一个矩形，我们必须实例化一个矩形的对象，给属性赋值：

```
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` 意思是“到 `box` 指向的对象那里，选择一个属性 `corner`；然后再到那个对象那里，选择 `x` 属性。”

图表显示了对象的状态：

³译注：Python 中，只有引用



一个对象是另外一个对象的属性叫做嵌套。

15.4 实例作为返回值

函数可以返回一个实例。比如，`find_center` 接受一个 `Rectangle` 作为参数，返回包含 `Rectangle` 中心点坐标的 `Point`。

beforeverb

```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

下面是一个例子：把 `box` 作为参数，把返回的点赋给变量 `center`：

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

15.5 对象是可变的

我们可以通过给对象的一个属性来改变对象的状态。比如，改变矩形的大小，但是不改变位置，我们可以修改 `width` 和 `height` 的值：

```
box.width = box.width + 50
box.height = box.width + 100
```

我们也可以编写函数来改变对象。比如，`grow_rectangle` 接受一个矩形对象，和两个值，`dwidth` 和 `dheight`，分别把 `dwidth` 和 `dheight` 加到矩形的宽和高上：

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

下面的例子，演示了效果：

```
>>> print box.width
100.0
>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
```



```
150.0
>>> print box.height
300.0
```

在函数体里，`rect` 是 `box` 的别名，所以 `rect` 改变时，`box` 也改变了。

Exercise 15.2 编写一个函数 `move_rectangle` 接受一个矩形和两个数字 `dx` 和 `dy`。函数通过把 `dx` 和 `dy` 的值分别加到 `corner` 点的 `x` 和 `y`。

15.6 复制

别名可以令程序难以阅读因为在一处改变了某些值可能会在其他某处产生不可预料的影响。很难跟踪所有指向给定对象的变量。

选择复制对象是别名的一种替代方法。`copy` 模块包含了一个函数 `copy` 复制任意的对象：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` 和 `p2` 包含了同样的数据，但是他们不是同一个点。

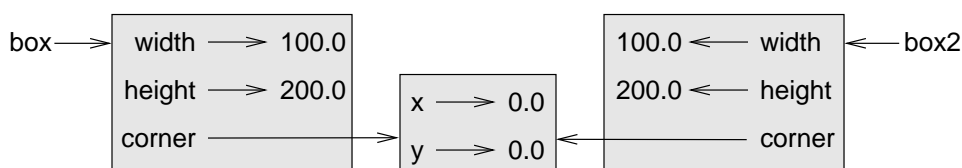
```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

`is` 操作符表明 `p1` 和 `p2` 不是同一个对象，这是我们希望。但是你或许希望 `==` 运算符产生 `True`，因为这两个点包含了同样的数据。在这种情况下，你可能会失望的了解到对于实例，`==` 缺省的行为和 `is` 操作符是相同的；它检查对象的身份而不是对象数据是否相等。我们也可以改变这种行为 --- 我们不久就会看到。

我们可以使用 `copy.copy` 复制长方形，我们将发现它复制了矩形对象，但是没有复制嵌套的点。

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

下面是对象图：



这种操作叫做浅拷贝，因为它拷贝了对象和它包含的引用，但是不包括嵌套的对象。

对于大多数的应用来说，这不是你想要的。此时，调用`grow_rectangle` 不会影响其他的其他的矩形。但是调用`move_rectangle` 会影响双方！这种行为很容易令人迷惑，也很容易产生错误。

幸运的是，`copy` 模块包含了另外一个方法 `deepcopy`，不仅复制对象本身，也复制对象指向的东西及其指向的东西等等。

你讲不会对这种操作叫做深拷贝而惊讶。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` 和 `box` 是两个完全分离的对象。

Exercise 15.3 编写另外一种版本`move_rectangle` 创建并返回一个新的矩形，不要改变原来的矩形。

15.7 调试

当使用对象的时候，我们很可能会遇到一些新的异常。如果你访问一个不存在的属性，将会得到 `AttributeError` 异常：

```
>>> p = Point()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

如果不能确定对象的类型是什么，可以这样：

```
>>> type(p)
<type '__main__.Point'>
```

如果不确定对象是否拥有一个属性，可以使用内置的函数 `hasattr`：

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一个参数可以是任意对象，第二个参数是一个字符串包含要查询的属性。

15.8 术语表

class 类: 用户自定义类型。类的定义创建了一个类对象。

class object 类对象: 包含用户自定义类型信息的对象。类对象可以用来产生一个实例。

instance 实例: 属于类的对象。

attribute 属性: 一个和对象相关的命名的值。

embedded (object) 嵌套对象: 作为属性存储在其他对象里的对象。

shallow copy 浅拷贝: 拷贝对象的结构, 包括指向嵌套对象的引用; 通过 `copy` 模块的 `copy` 函数实现。

deep copy 深拷贝: 拷贝对象的结构嵌套对象和包含在嵌套对象里的对象, 如此如此; 通过 `copy` 模块的 `deepcopy` 函数实现。

object daigram 对象图: 显示对象及其书香和属性值的图表。

15.9 练习

Exercise 15.4 Swampy(参看 4) 有一个模块, `World.py`, 包含了自定义的类定义 `World`。你可以这样导入它:

```
from World import World
```

这个版本的 `import` 语句从 `World` 模块导入了 `World` 类。下面的代码创建了一个 `World` 对象, 并且调用了 `mainloop` 方法, 等待用户。

```
world = World()
world.mainloop()
```

应该出现了一个带有框和空白区域的窗口。我们将要使用这个窗口画点, 长方形还有其他的图形。把下面的代码加到 `mainloop` 的前面, 运行这个程序。

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

你应该看到了一个边是黑色的绿色矩形, 代码的第一行创建了一个画布, 以空白区域填充窗口。画布对象提供了一些方法像 `rectangle` 来绘制不同的图形。

`bbox` 是序列, 代表了矩形的边界。第一个坐标对代表了矩形左下角, 第二个坐标代表了右上角。

你可以这么样来绘制一个圆:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

第一个参数是圆心的坐标; 第二个参数是半径。

如果把这条代码加入到程序中, 将会看到类似孟加拉国国旗的图形。(参看 wikipedia.org/wiki/Gallery_of_sovereign-state_flags)。

1. 编写一个函数 `draw_rectangle`, 接受一个画布和矩形为参数, 在画布上绘制矩形。

2. 在你的矩形对象里增加一个 `color` 属性，修改`draw_rectangle` 使得用颜色属性作为填充颜色。
3. 编写一个函数`draw_point`, 接受一个画布和点作为参数，在华布上画一个点。
4. 定义一个类 `Circle`，自己定义适当的属性，实例化一些 `Circle` 对象。编写函数`draw_cicle` 在华布上画圆。
5. 编写一个程序绘制捷克共和国的国旗。Hint: 可以这样绘制一个多边形：

```
points = [[-150,-100], [150, 100], [150, -100]]  
canvas.polygon(points, fill='blue')
```

我已经编写了一些程序，列出了可供使用的颜色；可以从这儿下载thinkpython.com/code/color_list.py.

Chapter 16

类和函数

16.1 时间

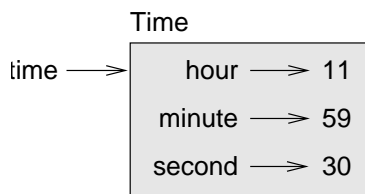
作为用户定义类型的另一个例子，我们将定义一个 `Time` 类，记录当前时间，类的定义如下：

```
class Time(object):  
    """represents the time of day.  
        attributes: hour, minute, second"""
```

我们可以创建一个新的 `Time` 对象，并对时、分和秒进行赋值：

```
time = Time()  
time.hour = 11  
time.minute = 59  
time.second = 30
```

`Time` 对象的状态图如下：



Exercise 16.1 编写函数 `print_time`，参数为一个时间对象，以时：分：秒的格式打印时间。提示：格式字符串 `'%.2d'` 使用至少两位打印一个整数，如果需要则在前面添零。

Exercise 16.2 编写布尔函数 `is_after`，读取两个时间对象 `t1` 和 `t2`，如果 `t1` 在 `t2` 之后则返回 `True`，否则返回 `False`。挑战：不使用 `if` 语句。

16.2 纯函数

在下面几个章节中，我们将编写两个函数，实现时间相加的功能。它们将展示两种函数：纯函数和修改。同时将给出一个我称为原型和补丁的开发计划，即对于一个复杂的问题，从简单的原型开始，增量地处理其中的复杂问题。

下面给出`add_time` 的一个简单原型:

```
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
```

这个函数创建一个新的 `Time` 对象, 初始化其属性并作为引用返回给一个新的对象。这被称为纯函数, 因为它不改变任何作为参数的对象, 除了返回一个值它没有其他作用, 类似显示一个值或读取用户输入。

我创建了两个时间对象来测试这个函数, `start` 包含了一个电影开始的时间, 如 *Monty Python and the Holy Grail*, `duration` 包含了电影的时间长度, 是 1 小时 35 分钟。

`add_time` 给出电影结束的时间。

```
>>> 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
```

10:80:00 不是你所想要的结果。问题在于这个函数没有处理分钟和秒钟加起来超过 60 的情况。当这个情况发生时, 我们需要将多余的秒钟“进位”到分钟, 将多余的分钟“进位”到小时。

下面给出一个改进的版本:

```
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
```

虽然这个函数是正确的，但是它开始变得冗长。我们之后会看见一个精简的版本。

16.3 修改函数

有时让函数修改参数对象是很有用的。这中情况下，修改对调用者是可见的。这样工作的函数被称为修改函数。

`increment` 是将一定秒数加到一个时间对象，可以写成一个修改函数。下面是一个草稿：

```
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
```

第一行执行基本的操作，后面几行处理我们之前遇到过的特殊情况。

这个函数对吗？如果参数 `seconds` 大于 60 会怎么样？

在这种情况下，进位一次是不够的，我们需要不断进位直到 `time.second` 小于 60。一个解决方案是使用 `while` 语句替换 `if` 语句。这能是函数工作正常，但不是很有效率。

Exercise 16.3 编写一个正确的 `increment`，不使用任何循环。

任何修改函数可以做的都可以使用纯函数来实现。事实上有的编程语言只允许纯函数。有些证据证明使用纯函数的程序相比使用修改函数的程序开发更快捷，错误更少。但是修改函数更加方便使用，而函数的编程效率相对较低。

通常，我推荐你使用纯函数，除非修改函数有明显的优势。这个称为函数式编程风格。

Exercise 16.4 编写纯函数版本的 `increment`，创建一个新的时间对象并返回，而不是修改参数。

16.4 原型与计划

我在展示的开发计划被称为“原型和补丁”。对于每个函数，我编写实现基本功能的原型并进行测试，并对错误打补丁。

这个方法会很有效率，尤其是对问题没有一个深入的认识。但是增量的修改会使得代码变得不必要的复杂，因为需要处理不同的特殊情况，同时由于你很难知道是否找到了所有的错误，代码也不可靠。

另一种是有计划的开发，从高层次分析问题将会简化程序的设计。在这个例子中，对问题的分析在于认识到时间对象是 3 个 60 进制的数（参考 wikipedia.org/wiki/Sexagesimal。）！秒是“第 1 列”，minute 是“第 60 列”，小时是“第 360 列”。

当我们编写 `add_time` 和 `increment`，我们完成了基 60 的加法，这也是为什么我们需要从一列到另一列进位。

这个观察给出了解决整个问题的另一个方法，我们可以将时间对象转换为整数，并利用计算机进行整数计算。

下面的函数将时间转换为整数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

下面的函数将整数转换为时间（回忆 `divmod` 将第一个参数除以第二个参数，并返回商和余数的元组）。

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

你也许需要一些思考，并运行一些测试来确保这些函数工作正常。一个测试方法是对许多 `x` 值检查 `time_to_int(int_to_time(x)) == x`。这是一个强壮型检查的例子。

当你确信它们是正确的，你可以使用它们重写 `add_time`：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

这个版本比原来的简洁，同时也更容易验证。

Exercise 16.5 使用 `time_to_int` 和 `int_to_time` 重写 `increment`。

有时候，60 进制和 10 进制的相互转换比处理时间更难。基数转换相对更抽象，我们的直觉更擅长处理时间。

但是如果我们将时间看成 60 进制的数，并编写转换函数（`time_to_int` 和 `int_to_time`），我们使得程序更简短，更适合阅读和调试，以及更可靠。

同时也方便以后增加新的特性。例如，想象将两个时间相减，得到两者之间的间隔。最直观的方法是实现借位减法。使用转换函数可以更简单，也更容易正确。

讽刺的是有时候将问题复杂化（或普遍化）实际简化了问题（因为特殊情况变少，同时出错概率减小）。

16.5 调试

一个时间对象被称为是良好组织的，如果分钟和秒钟位于 0 到 60（包括 0 但不包括 60），`hours` 是正的，小时和分钟是整数，但我们可以允许秒钟有小数部分。

类似这些要求被称为约束，它们应该始终为真。换言之，如果它们非真，则有些地方就有错误。

编写程序检查约束可以帮助你检查错误并找出原因。例如，你可以编写函数`valid_time`，读取一个时间对象作为参数，如果违反了约束则返回 `False`：

```
def valid_time(time):
    if time.hours < 0 or time.minutes < 0 or time.seconds < 0:
        return False
    if time.minutes >= 60 or time.seconds >= 60:
        return False
    return True
```

在每个函数的开头你可以检查参数来保证它们是有效的：

```
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，它将检查一个给定的约束，如果检查失败则会发出一个异常错误。

```
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` 语句很有用，它们区分普通的条件判断和异常检查。

16.6 术语

原型和补丁：一种开发计划，包括编写程序的草稿、测试、修改发现的错误。

有计划的开发：一种开发计划，包括从高层次对程序进行分析，相对增量开发或原型开发有更多的计划。

纯函数：不修改作为参数的对象的函数。

修改函数：修改一个或多个作为参数的对象的函数。

函数式编程风格：一种程序设计模式，将大多数函数设计为纯函数。

约束：在程序执行时必须始终为真的条件。

16.7 练习

Exercise 16.6 编写函数`mul_time`，参数为一个时间对象和一个数，返回一个新的时间对象，其值是原时间和数的乘积。

使用`mul_time`编写一个函数，参数为一个时间对象和一个数值，时间对象表示完成一个比赛所用的时间，数值表示距离，返回一个时间对象，其意义是平均速度（英里每单位时间）。

Exercise 16.7 编写日期对象的类定义，含有属性日，月和年。编写函数`increment_date`，参数为一个日期对象 `date` 和一个整数 `n`，返回值为一个新的日期对象，对应 `date` 后的 `n` 天。提示：“2 月没有 30 天...” 挑战：你的函数在闰年工作正常吗？参考wikipedia.org/wiki/Leap_year。

Exercise 16.8 `datetime` 模块提供了类似本章节中的日期和时间对象，`date` 和 `time`，它们提供了丰富的方法和运算符，阅读docs.python.org/lib/datetime-date.html 中的文档。

1. 使用 `datetime` 模块编写程序，读取一个日期，打印这个日期所在的周。
2. 编写程序，读如一个生日，打印用户的年龄，以及多少天、小时、分钟和秒钟后是下一个生日。

Chapter 17

类和方法

17.1 面向对象特点

Python 是一门面向对象的编程语言，意味着它提供很多特点支持面向对象编程。

完整的定义面向对象编程不是一件容易的事，但是我们已经看到它的一些特点：

- 程序由对象定义和函数定义组成，大多数的计算都在操作对象过程中进行。
- 每个对象定义都和现实世界的一些对象或者概念符合，操作对象的函数和现实世界对象的交互方式相一致。

比如，在??章节的 **Time** 类和人们记录日期的方式相同，我们定义的函数和人们处理时间的方式相一致。类似地，**Point** 和 **Rectangle** 类和数学意义上的点和矩形相一致。

迄今为止，我们没有很好的利用 **Python** 提供的特点进行面向对象编程。这些特点不是强制使用；大多数特点为我们已经解决的问题提供了可供选择的语法。但是杂很多情况下，可替代的方案更精简准确地表示程序的结构。

比如，在 **Time** 程序中，类定义和紧跟其后的函数定义没有什么明显的联系。仔细看看，就会发现，每个函数都接受了至少一个 **Time** 对象作为参数。

这个发现激发了方法的出现；方法就是和一个特定的类结合在一起的函数。我们已经于遇到了字符串，列表，字典和元组的方法。这一章，我们为自定义类型定义方法。

方法在语义上就是函数，但是有来嗯个语法上的不同：

- 方法定义在类体里，使得方法和类的关系更为明显。
- 调用方法的语法和函数不同。

在接下来的几个部分里，我们把前两章的函数拿过来，把它们转换成方法。转换的方法是机械的，仅仅跟着一系列的步骤做就行了。如果，你很熟连把一种形式转换成另外一种形式，你将会更好的选择你想要的方式。

17.2 Printing objects

在??章，我们定义了一个 `Time` 类，在练习 16.1，你写了一个函数 `print_time`：

```
class Time(object):
    """represents the time of day.
       attributes: hour, minute, second"""

def print_time(time):
    print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

调用这个函数，你必须给函数传递一个 `Time` 对象作为参数：

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

要把 `print_time` 转换成方法，我们所要做的就是将函数定义移到类定义里。注意缩进。

```
class Time(object):
    def print_time(time):
        print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

现在有两种方式调用 `print_time`。地一种方式是使用函数语法（不常见）：

```
>>> Time.print_time(start)
09:45:00
```

这种方法里，`Time` 是类名，`print_time` 是方法名。`start` 作为参数被传递。

第二种（更简洁）方式是使用方法语法：

```
>>> start.print_time()
09:45:00
```

在这种方式里，`print_time` 是方法名，`start` 是方法被调用的对象，叫做主体。就像句子的主语是句子的主体一样，方法调用的主体方法的主体。

在方法内部，主体赋给第一个参数，这个例子里是 `start` 被赋给 `time`。

习惯上，方法的第一个参数是 `self`，所以更常见的是把 `print_time` 写成这样：

```
class Time(object):
    def print_time(self):
        print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

这个习惯的原因是一个隐喻：

- 函数调用的语法，`print_time(start)`，意味着函数是主体。好像这样，“嗨，`print_time`！这里有一个对象需要你输出。”
- 在面向对象编程中，对象是主体。方法调用 `start.print_time()` 是这样的，“嗨，`start`！请打印自己。”

这个方式上的转变更加的礼貌，但是，你看不出有什么更有利的地方，在我们遇到的例子里，确实是这样的。但是，有时候，把函数的责任调到对象身上使得函数更加实用，也更容易维护和复用。

Exercise 17.1 把`time_to_int` 函数 (??部分) 改成方法。把`int_to_time` 函数改成方法是不恰当的，至少我们清楚要调用对象。

17.3 另外一个例子

下面是 `increment` (16.3) 被改写为方法的一个版本:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

这个版本假定`time_to_int`(练习??) 也是被写成了方法。必须要提一下，这个是一个纯函数，不是一个改变版。

下面的是调用 `increment` 的方式:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

主体 `start` 被赋给第一个参数 `self`。参数 `1337`，被赋给第二个参数 `seconds`。

这个方法是令人迷惑的，特别是当发生错误时。比如，如果传递两个参数调用 `increment`，会得到:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

错误信息乍看起来是令人迷惑的，因为在括号里明明只有 2 个参数。但是主体也是被认为是参数，所有一共有 3 个参数。

17.4 更复杂的一个例子

`is_after`(联系 16.2) 稍微有点复杂，因为它接受两个 `Time` 对象作为参数。此时，习惯上把第一个参数当作哦 `self`，第二个参数作为其他:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

使用这个方法，必须使用一个对象作为主体，另外一个对象作为参数:

```
>>> end.is_after(start)
True
```

有趣的是这个几乎完全可以像英语一样读: “end is after start?”

17.5 初始方法

初始方法 (简称“initialization”) 是一个特殊的方法当一个对象实例化的时候。全名是 `__init__` (两个下划线, 然后是 `init`, 最后又是两个下划线)。 `Time` 类的初始方法可以这么写:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

`__init__` 的参数和类的属性拥有相同的名称是很常见的。语句

```
    self.hour = hour
```

存储了 `hour` 的值作为 `self` 的属性。

参数是可选的, 如果调用 `Time`。而不指定参数, 就会得到一个缺省值。

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

如果提供了一个参数, 就会覆盖 `hour`:

```
>>> time = Time(9)
>>> time.print_time()
09:00:00
```

如果提供两个参数, 就会覆盖 `hour` 和 `minute`。

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

如果提供三个参数, 就会覆盖所有的三个缺省值。

Exercise 17.2 为 `Point` 类编写一个初始化方法, 接受 `x` 和 `y` 作为可选择参数, 并且把他们赋给对应的属性。

17.6 `__str__` 方法

`__str__` 像 `__init__` 一样, 是一个特殊的方法, 返回一个字符串化的对象。

比如, 这里是 `Time` 对象的一个 `str` 方法:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

当 `print` 一个对象时, Python 调用 `str` 方法:

```
>>> time = Time(9, 45)
>>> print time
09:45:00
```

当我编写新类的时候, 我几乎都要从编写 `__init__` (使得更容易实例化对象) 和 `__str__` (使得调试更方便) 方法开始。

Exercise 17.3 为 `Point` 编写一个 `str` 方法。创建一个 `Point` 对象, 并打印它。

17.7 运算符重载

通过定义其他的特殊方法, 可以指明用户自定义类型的操作符的行为。比如, 你为 `Time` 类定义了一个方法 `__add__` 方法, 就可以在 `Time` 对象间使用 `+` 运算符。

下面是可能的定义:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

下面演示的是如何使用它:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

当在 `Time` 对象间应用 `+` 运算符时, Python 调用 `__add__`。当打印结果的时候, Python 调用 `__str__`。安静的表面蕴藏着无限的内容!

改变运算符的行为, 适应自定义类型叫做运算符重载。Python 中的每一个运算符都有一个对应的特殊方法, 像 `__add__`。欲之详情参阅 docs.python.org/ref/specialnames.html。

Exercise 17.4 为 `Point` 类编写一个 `add` 方法。

17.8 基于类型的调度

前一部分, 我们相加了两个对象, 但我们也想把一个整数加到 `Time` 对象上。下下面是 `__add__` 的一个版本, 检查 `other` 的类型, 然后调用 `add_time` 或 `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)
```

内置函数 `isinstance` 接受一个值和一个类对象，如果值是类的对象，返回在 `True`。

如果 `other` 是一个时间对象，`__add__` 调用 `add_time`，否则，函数认为参数是一个整数，调用 `increment`。这种操作叫做基于类型的调度，因为程序依据参数的类型分派不同的操作。

下面是一个使用 `+` 运算符的例子，传递了不同类型的参数。

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

不幸的是，这个加法的实现不是可交换的，如果第一个操作数是整数，会得到：

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

问题在于，不是要求 `Time` 对象去加一个整数，Python 要求整数加一个 `Time` 对象，当然，Python 不知道如何做。有一个很巧妙的解决办法：特殊方法 `__radd__`，代表“右边相加”。当 `Time` 对象出现在 `+` 的右边时，这个方法被调用。下面是定义：

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

下面是使用方式：

```
>>> print 1337 + start
10:07:17
```

Exercise 17.5 为 `Point` 类编写一个函数 `add`，要求可以适用于操作数是 `Point` 对象或者是元组：

- 如果第二个操作数是点，方法返回新的点，坐标 `X` 等于操作数的 `X` 坐标之和，`y` 也是。

- 如果第二个操作数是元组，方法把元组的第一个元素加到 x 上，第二个加到 y 上，返回新的点。

17.9 多态

基于类型的调度必要时是非常有用的，但是并不总是很有必要。通常不必要根据不同的类型编写不同的函数。

我们为字符串编写的很多函数都是适用于任何线性数据结构。比如，在 11.1 部分，我们使用 `histogram` 统计每个字符在单词中出现的次数。

```
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` 的元素是散乱的，他们就可以作为 `d` 的关键字¹。

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

能够适用于不同类型的函数叫做多态。多态促进了代码的复用。比如，内置函数 `sum`，求序列元素的和，对于元素支持相加的序列都是适用的。

`Time` 对象也提供了 `add` 方法，所以 `sum` 函数也使用。

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

总的来说，如果函数里定义的操作适用于给定的类型，那么函数就使用于那种类型。

最好的多态就是不自主的而产生的 --- 你发现你写的函数适用于你没有打算适用的类型！

17.10 调试

在程序运行时，给对象增加属性是合法的，但是如果你是一个类型论的坚持者，使相同类型的对象拥有不同的属性是不可靠的习惯。最好的是在初始化方法里初始化所有的对象属性。

如果不确定对象时候拥有一个特定的属性，可以使用内置函数 `hasattr`（参看 15.7）。

¹译注：这句话有待商榷，关键字必须是 `immutable`

另外一种访问对象属性的方式是通过 `__dict__` 方法，以字典的形式显示属性名（作为字符串）和值。

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

从调试的角度看，经常使用它是个不错的调试方式。

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

`print_attribute` 遍历对象字典的所有项，打印对应的名称及其值。

内置函数 `getattr`，接受一个对象和属性名，返回属性的值。

17.11 术语表

object-oriented language 面向对象语言：提供用户自定义类型和方法语法等特点的语言，有利于面向对象编程。

object-oriented programming 面向对象编程：一种编程风格，倡导数据和操作封装在类及其方法里。

method 方法：定义在类里的函数，通过类的实例调用。

subject 主体：方法被调用的对象。

operator overloading 运算符重载：改变像 `+` 之类运算符的行为，使得适用于用户自定义类型。

type-based dispatch 基于类型的调度：一种编程模式，检查操作数类型，然后调用不同的函数。

polymorphism 多态：函数适用于不同类型的数据。

17.12 练习

Exercise 17.6 这个练习是对 Python 中最常见也是最难发现的错误之一的一个警告。

1. 编写一个类 `Kangaroo`，要求拥有下面的方法：

- (a) `__init__` 方法，初始化一个属性 `pouch_contents` 为空列表。
- (b) `put_in_pouch`，接受一个任意类型的对象，加到 `pouch_contents` 上。
- (c) `__str__` 方法，返回字符串化的 `Kangaroo` 对象和袋鼠袋子里的东西。

测试你的代码：创建两个 `Kangaroo` 对象，分别赋给 `kanga` 和 `roo`，把 `roo` 加到 `kanga` 的袋子里。

2. 下载 thinkpython.com/code/BadKangaroo.py。解答里包含了前一个问题的解答，但是有一个很大的 bug。找出并改正它。

如果你卡壳了，可以下载 thinkpython.com/code/GoodKangaroo.py，它解释了问题的所在并且给出了完整的解答。

Exercise 17.7 `Visual` 是 Python 的一个模块，提供了 3D 图形。通常在安装 Python 的时候，默认是不安装它的，你可以从软件仓库里下载，如果那里没有，从这儿 vpython.org。

下面的例子，创建了一个 3D 空间，宽，长，高均为 256 单元，中心被设定在点 (128,128,128)，绘制了一个蓝色的地球。

```
from visual import *

scene.range = (256, 256, 256)
scene.center = (128, 128, 128)

color = (0.1, 0.1, 0.9)          # mostly blue
sphere(pos=scene.center, radius=128, color=color)

color 是 RGB 元组，也就是是哦，元组的元素是从 0.0--1.0 的 RGB 值（参看 wikipedia.org/wiki/RGB\_color\_model）。
```

如果运行这个代码，你将会看到一个黑色背景和蓝色地球的窗口。如果上下拖拉中间的按钮，可以放大缩小图像。也拖拉右边的按钮旋转图形，但是只能显示一个地球，很难找出区别：

```
t = range(0, 256, 51)
for x in t:
    for y in t:
        for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. 把代码存储在脚本里，确保能够运行。
2. 修改程序，使得立方里的每个地球的颜色和它的位置相对应。注意：坐标的范围是 0--255, RGB 元组的范围是 0.0--1.0。
3. 下载 thinkpython.com/code/color_list.py。使用函数 `read_colors`，产生你系统上可使用的颜色列表 --- 颜色名和对应的 RGB 值。对于每一个颜色，在与 RGB 值对应的位置绘制一个地球。

可以参看我的解答 thinkpython.com/code/color_space.py。

Chapter 18

继承

在本章中我们将设计一个来玩扑克的类。如果你不玩扑克，你可以在wikipedia.org/wiki/Poker 中阅读相关信息，但你不必这么做，做练习时我会告诉你一些所需要的信息。

如果你不熟悉如何玩扑克，你可以阅读wikipedia.org/wiki/Playing_cards 中的内容。

18.1 扑克对象

一副牌有 52 张牌，每一张属于 4 个花色和 13 个等级。4 中花色是黑桃、红桃、草花和方块。13 个等级是 Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q 和 K。对于不同的游戏，Ace 可能比 2 大，也可能比 2 小。

如果我们要定义一个纸牌的对象，显然其属性有等级和花色，但是属性的数据类型应该如何选择？一个方法是使用字符串，如用 'Spade' 描述花色，'Queen' 描述等级。这么做的一个问题在于实现花色或等级的比较不是很容易。

另一个方法是使用整数对等级和花色编码。在这里，“编码”指我们将定义一个数字和花色以及数字和等级之间的映射。这种映射不是秘密的（区别“加密”）。

例如，下面的表格给出了花色和对应整数编码：

Spades	↦	3
Hearts	↦	2
Diamonds	↦	1
Clubs	↦	0

这个编码简化了卡片的比较，由于高的花色映射为大的数，我们可以通过比较编码来比较花色。

等级的映射相对更明显，每个数字等级映射为对应的整数，对于其他卡片：

Jack	↦	11
Queen	↦	12
King	↦	13

我使用 \mapsto 符号来突出这些映射不是 Python 程序实现的。它们属于程序的设计，但它们不直接的表现代码中。

Card 类的定义如下：

```
class Card(object):
    """represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Init 方法有两个可选参数，默认的卡片是草花 2。

如果要创建一个卡片，你可以调用 Card，并传入你要的花色和等级。

```
queen_of_diamonds = Card(1, 12)
```

18.2 类属性

为了以适合人类阅读的格式打印卡片对象，我们需要整数代码到对应花色和等级的映射。一个自然的方法是使用字符串列表，我们将这两个列表赋值给类属性：

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                              Card.suit_names[self.suit])
```

定义在类内部任何函数外的变量，如suit_names 和rank_names，被称为类属性，因为它们和类对象 Card 像关联。

该术语区别类似 suit 和 rank 的变量，它们被称为实例属性，因为它们和一个特定的实例相关联。

两种属性都通过点符号访问。例如，在__str__ 里，self 是一个卡片对象，self.rank 是它的等级。类似的，Card 是一个类对象，Card.rank_names 是一个和类关联的字符串列表。

每个卡片都有自己的花色和等级，但是有一个suit_names 和rank_names 的备份。

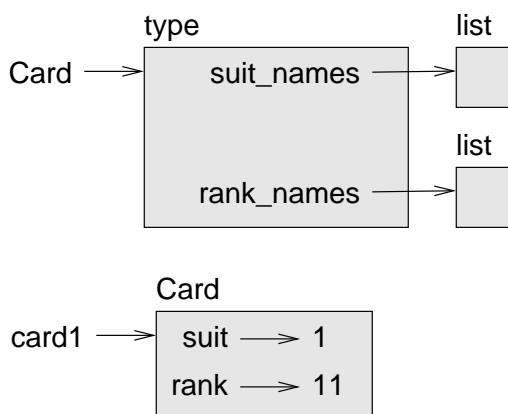
将这些结合起来，表达式Card.rank_names[self.rank] 意思是“从 self 对象中使用属性 rank 作为下标，访问 Card 类中的rank_names 列表，选择对应的字符串”。

rank_names 中的第一个元素是 None，因为没有等级为零的卡片。通过引入 None 作为占位符，我们很好的将下标 2 映射为字符串'2'，以此类推。若不使用这种方法，我们可以使用一个字典，而不是一个列表。

基于我们已有的方法，我们可以创建并打印卡片：

```
>>> card1 = Card(2, 11)
>>> print card1
Jack of Hearts
```

下面给出 `Card` 类对象和一个卡片实例的图：



`Card` 是一个类对象，它的数据类型为 `type`。`card1` 的类型为 `Card`。（为了节省空间，我没有画出 `suit_names` 和 `rank_names` 的内容）。

18.3 比较卡片

对于内建类型，关系运算符（如 `<`，`>`，`==` 等）可以比较它们的值。对于用户定义的类型，我们可以通过提供 `__cmp__` 函数来重载内建运算符的行为。

`__cmp__` 接受两个参数，`self` 和 `other`，如果第一个对象大则返回一个正数，如果第一个对象小则返回一个负数，如果两者两等则返回 `0`。

卡片的正确顺序不是很明显。例如一个草花 3 和一个方块 2 哪个大？一个等级更高，而另一个花色更高。为了比较卡片，你需要确定等级和花色哪个更重要。

答案取决于你玩的是什么游戏。为了简单起见，我们任意的选择花色更为重要，因此所有的黑桃都大于任意方块，依此类推。

当这个决定后，我们可以编写 `__cmp__`：

```
# inside class Card:

def __cmp__(self, other):
    # check the suits
    if self.suit > other.suit: return 1
    if self.suit < other.suit: return -1

    # suits are the same... check ranks
    if self.rank > other.rank: return 1
    if self.rank < other.rank: return -1

    # ranks are the same... it's a tie
    return 0
```

你可以使用元组比较编写更简洁的程序:

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

内建函数 `cmp` 和方法 `__cmp__` 有相同的接口: 它读取两个值, 如果第一个更大则返回一个正数, 第二个更大则返回一个负数, 如果相等则返回 0。

Exercise 18.1 为时间对象编写方法 `__cmp__`。提示: 你可以使用元组比较, 也可以考虑使用整数减法。

18.4 纸牌

现在我们有卡片对象, 下一步是定义纸牌。由于纸牌是由卡片组成的, 自然的想法是每个纸牌将一个卡片的列表作为它的属性。

下面是纸牌的类定义。初始化方法创建卡片属性, 并生成标准的 52 张卡片:

```
class Deck(object):

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

生成纸牌最简单的方法是使用一个嵌套的循环。外层循环枚举 0 到 3 的花色, 内层循环枚举 1 到 13 的等级。每次迭代创建一个对应当前花色和等级的新卡片, 并附加在 `self.cards` 后。

18.5 打印纸牌

下面是纸牌的 `__str__` 方法:

```
#inside class Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

这个方法演示了汇聚大字符串的有效的方法: 建立一个字符串列表, 然后使用 `join` 方法。内建函数 `str` 对每个卡片调用 `__str__` 方法, 并返回表征的字符串。

由于我们在新行符号上调用 `join`, 卡片通过新行符分割。下面是结果:


```
>>> 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
```

虽然结果看起来是 52 行, 实际上它是一个包含换行符的一个长字符串。

18.6 添加, 删除, 洗牌, 理牌

对于卡片, 我们需要一个从纸牌中删除一张卡片并返回改纸牌的方法。列表的 `pop` 方法提供了一个便捷的实现方式:

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

由于 `pop` 删除列表中最后一张卡片, 我们是纸牌底部进行操作。在现实生活中从底部操作是不可取的¹, 但在这里也是行得通的。

为了添加一张卡片, 我们可以使用列表的 `append` 方法:

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

类似这种调用其他函数, 本身不做很多具体工作的方法有时被称为饰面。这个词来自木工行业, 通常人们将一片很薄的好木材黏贴在一块便宜的木材的表面。

在本例中我们定义个一个“单薄”的方法, 使列表的操作适用于纸牌。

作为另一个例子, 我们可以编写纸牌方法 `shuffle`, 调用 `random` 模块中的 `shuffle` 函数:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

不要忘了导入 `random` 模块。

Exercise 18.2 编写纸牌方法 `sort`, 通过调用列表方法 `sort` 对纸牌中的卡片进行排序。`sort` 适用我们定义的 `__cmp__` 方法来决定排序顺序。

¹参考wikipedia.org/wiki/Bottom_dealing

18.7 继成

和面向对象编程最相关的语言特点是继成。继成是通过修改已有的类来创建新的类的能力。

之所以称为“继成”是因为新的类继成了已有类的方法。换言之，已有类被称为父类，新创建的类被称为子类。

例如，我们要创建一个类来表示“手牌”，即一个玩家手中持有的牌。手牌类似纸牌：它们都是卡片的集合，都需要类似添加和删除卡片的操作。

手牌同时区别于纸牌：我们需要手牌有一些操作，但这些操作对纸牌来说没有意义。例如，在扑克中我们需要比较两个手牌来决定谁获胜。在桥牌中，我们需要计算手牌的分数，以便要价。

两个类的相似关系和不同点导致了它们的继成关系。

子类的定义和其他类定义相同，但父类的名字出现在括号中：

```
class Hand(Deck):
    """represents a hand of playing cards"""
```

该定义表示 `Hand` 继成 `Deck`，这意味着我们可以想纸牌一样对手牌使用类似 `pop_card` 和 `add_card` 的方法。

`Hand` 同时继成了 `Deck` 的 `__init__`，但这并不是我们想要的：手牌的 `cards` 属性应该被初始化为一个空的列表，而不是塞满 52 张新牌。

如果我们给 `Hand` 提供一个初始化方法，它将重载 `Deck` 中的方法：

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

因此当你创建了一个手牌对象，Python 调用这个初始化方法：

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

但是其他方法从 `Deck` 继成，所以我们可以使用 `pop_card` 和 `add_card` 来处理卡片：

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

自然的下一步是将这些代码封装成 `move_cards` 方法：

```
#inside class Deck:
```

```
def move_cards(self, hand, num):
    for i in range(num):
        hand.add_card(self.pop_card())
```

`move_cards` 读取两个参数，一个为手牌对象，一个为要处理的卡片数。它同时修改 `self` 和 `hand`，并返回 `None`。

在有的游戏中，卡片在不同手牌中移动，或者从手牌移动到台面牌。你可以使用 `move_cards` 来进行任何这些操作：`self` 可以是手牌或者桌面牌，`hand` 同样可以是桌面牌，虽然它名字叫手牌。

Exercise 18.3 编写纸牌方法 `deal_hands`，读取两个参数，分别是手牌的个数和每个手牌的卡片个数，并根据这个数字创建手牌对象，返回一个手牌对象列表。

继成是一个有用的特性。通过使用继成可以将一些重复行的程序写得更优雅。继成使得代码重用更容易，你可以自定义父类的行为，而不需要去改变它。在某些情况下，继成结构反应了现实中问题的结构，使得问题容易理解。

另一方面，继成会使得程序难以阅读。当一个方法被调用时，有时方法的定义的位置不是很清楚。相关的代码可能散布在多个模块中。同时，很多可以用继成做的事情同样可以不用继成来实现，甚至做得更好。

18.8 类图

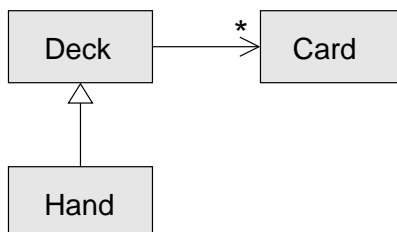
目前位置我们见过了栈图，它现实了程序的状态，对象图，它现实了对象的属性和值。这些图是程序执行过程中的快照，因此它们随程序的执行而改变。

同时它们非常详细，有时对某些应用太详细了。类图是个相对抽象的表示程序结构的图，它显示了类的结构和类之间的关系，而不是显示每一个对象。

存在以下几种类的关系：

- 类中的一个对象包含另一个类的对象的引用。例如，每个长方形包含一个点的应用，每个纸牌包含多个卡片的引用。这类关系被称为包含，正如“长方形包含一个点”。
- 一个类是另一个类的继成。这种关系被称为是，正如“手牌是纸牌的一种”。
- 一个类也许决定于另一个类，即一个类的改动要求其他类的改动。

类图是图形化的表示这些关系的图²。例如，下面的图显示了 `Card`，`Deck` 和 `Hand` 的关系。



²我在这里使用的图类似 UML (参考wikipedia.org/wiki/Unified_Modeling_Language)

空心三角箭头表示“属于关系”，在本例中表示手牌继承了纸牌。

标准箭头表示“有关系”，在本例中纸牌有卡片的引用。

在箭头头部的星号（*）表示多态。它指示了纸牌中有多少张卡片。多态可以是一个简单的数字，如 52，一个范围，如 5..7，或一个新号，表示纸牌可以有任何多的卡片。

更详细的图将显示纸牌事实上又一个卡片列表，但是通常类似列表和字典的内建类型不出现在类图中。

Exercise 18.4 阅读 `TurtleWorld.py`, `World.py` 和 `Gui.py`，绘制类图来显示它们之间的关系。

18.9 调试

继承使得调试成为一个挑战，因为当你对一个对象调用方法是，你也许不知道哪个方法被调用。

假设你编写了一个手牌对象的函数，你希望它能适用于任何手牌，如扑克手牌，桥牌手牌等。假设你调用了类似 `shuffle` 的方法，你也许调用的是定义在纸牌中的方法，但是如果某个子类重载了这个方法，你会得到那个版本的方法。

当你不确定程序执行的流程，最简单的方法是在相关的方法的开始部分添加打印语句。如果 `Deck.shuffle` 打印类似 `Running Deck.shuffle` 的语句，那么当程序执行时它将追踪执行的流程。

另一个方法是使用下面的函数，它接受一个对象和一个方法名（以字符串的形式），返回提供该方法定义的类：

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

下面给出一个例子：

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

因此这个手牌的 `shuffle` 方法来自纸牌。

`find_defining_class` 使用 `mro` 方法得到用于查找方法的类对象（类型）列表。“MRO”指“method resolution order”，即方法解析顺序。

以下是程序设计的建议：每当你重载一个方法，新方法的接口应该和原方法相同。它们应使用相同的参数，返回相同的类型，符合相同的先决条件和后决条件。如果你遵守这些规则，你会发现任何使用于超类实例（如纸牌）的函数同样使用于子类实例（如手牌或扑克手牌）。

如果你违反了这些规则，你的代码很肯会崩溃。

18.10 术语

编码： 通过建立映射使用一个集合的值表示另一个集合的值。

类属性： 和类对象关联的属性。类属性定义在类定义内部，方法定义外部。

实例属性： 和类的实例相关的属性。

饰面： 为另一个函数提供不同的接口而不进行许多计算的方法或函数。

继承： 通过修改先前定义的类来创建新的类的能力。

父类： 被子类继承的类。

子类： 通过继承已有的类而创建的新的类。

属于关系： 子类和父类之间的关系。

包含关系： 两个类之间的关系，其中的一个类包含另一个类的引用。

类图： 表示程序中的类以及类之间的关系的图。

多态： 类图中的标记，对于包含关系，该标记说明了对其他类的引用的数量。

18.11 练习

Exercise 18.5 下面是扑克中肯能的手牌，以大小升序排列（概率降序）：

对子： 等级相同的两张牌。

双对： 等级相同的两对对子。

相同的 3 个： 等级相同的 3 张卡片。

顺子： 5 张等级连续的卡片（aces 可以为高或低，因此 Ace-2-3-4-5 是个顺子，10-Jack-Queen-King-Ace 也是顺子，但是 Queen-King-Ace-2-3 不是）。

同花： 5 张花色相同的卡片。

3 张相同和 2 张相同的牌： 3 张牌等级相同，另 2 张牌等级相同。

相同的 4 个 等级相同的 4 张牌。

同花顺： 5 张花色相同的顺子。

本练习的目的是估计抽到不同的手牌的概率。

1. 从thinkpython.com/code 下载下列文件：

`Card.py` : 本章中 `Card`, `Deck` 和 `Hand` 的完整版本。

`PokerHand.py` : 未完成的手牌的类，包含一些测试代码。

2. 如果你运行 `PokerHand.py`，它将抽取 7 张手牌，并检查是否包含同花顺。在你继续前请仔细阅读代码。
3. 在 `PokerHand.py` 添加方法 `has_pair`, `has_twopair` 等，根据相关准则进行判断并返回 `True` 或者 `False`。你的代码应使用于任意张数的手牌（虽然 5 和 7 是最常见的数量）。

4. 编写方法 `classify`，找出手牌中值最大的分类，并记录在 `label` 属性中。例如，一个 7 张的手牌可能有一个顺子和一对，那么应该标记为“顺子”。
5. 当你确信你的分类方法工作正常，下一步是估计不同手牌出现的概率。在 `PokerHand.py` 中编写函数，对一副牌进行洗牌，然后分为若干手牌，对手牌分类，统计不同分类出现的次数。
6. 打印分类和概率的表。多次运行程序，直到输出稳定到一个合理的精度。将你的结果和 wikipedia.org/wiki/Hand_rankings 比较。

Exercise 18.6 本练习使用章节 4 的 `TurtleWorld`。你将编写代码，让乌龟玩贴标签的游戏。如果你不熟悉游戏规则，参考 [wikipedia.org/wiki/Tag_\(game\)](http://wikipedia.org/wiki/Tag_(game))。

1. 下载 thinkpython.com/code/Wobbler.py 并运行。你将看到乌龟世界里有 3 只乌龟。如果你按下 `Run` 按钮，乌龟将随机移动。
2. 阅读代码，了解它是如何工作的。闲逛者类继承乌龟类，这意味着乌龟的方法 `lt`, `rt`, `fd` 和 `bk` 同样适用于闲逛者。
`step` 方法由乌龟世界调用。它调用 `steer`，令乌龟朝向指定的方向。`wobble` 根据乌龟的笨拙程度随机的转向，`move` 根据乌龟的速度向前移动一定的距离。
3. 创建 `Tagger.py`，导入 `Wobbler`，定义类 `Tagger`，继承 `Wobbler`。以 `Tagger` 的类对象作为参数的调用 `make_world`。
4. 在 `Tagger` 中添加 `steer` 方法，重载 `Wobbler` 中的函数。作为起步练习，编写始终将乌龟指向原点的版本。提示：使用数学函数 `atan2` 和乌龟属性 `x`, `y` 和 `heading`。
5. 修改 `steer`，将乌龟约束在边界内。为了调试，你可以使用 `Step` 按钮，它将对每个乌龟调用 `step`。
6. 修改 `steer`，令每只乌龟朝向离它最近的邻居。提示：乌龟有属性 `world`，是它们所在的乌龟世界的引用，而乌龟世界有属性 `animals`，对应所有乌龟的列表。
7. 修改 `steer` 让乌龟玩贴标签游戏。你可以在 `Tagger` 上添加方法，并重载 `steer` 和 `__init__`，但你不能修改或重载 `step`, `wobble` 或 `move`。另外 `steer` 允许改变乌龟的方向，但不能改变位置。
修改规则和你的 `steer` 方法，增加游戏的质量。例如，慢的乌龟应该有机会给快的乌龟贴上标签。

你可以在 thinkpython.com/code/Tagger.py 下载到我的程序。

Chapter 19

实例学习：Tkinter

19.1 GUI

我们迄今为止编写的程序都是字符界面下的，现在很多程序使用图形用户接口（**graphical user interfaces**，简称 **GUIs**）。

Python 提供了多种选择来编写 GUI 程序，包括 **wxPython**，**Tkinter** 和 **Qt**¹。每种框架都优缺参半。这也就是为什么 Python 在图形库上没有形成一个标准的原因。

这一章，我要讲述的是 **Tkinter**，因为我认为它是最容易入手的。本章的很多概念同样适用于其他 GUI 模块。

有一些关于 **Tkinter** 的书和网站。网上最好的资料是 **Fredrik Lundh** 写的 *An Introduction to Tkinter*。

我也写了一个模块 **Gui.py**，包含在 **Swampy** 里。它提供了 **Tkinter** 里函数和类的简单接口。这章的例子就是以这个模块为基础。

下面是一个创建显示 **Gui** 的简单例子：

要创建一个 GUI，必须导入 **Gui** 模块，并且实例化一个 **Gui** 对象：

```
from Gui import *

g = Gui()
g.title('Gui')
g.mainloop()
```

当运行这段代码，会出现一个窗口，窗口由灰色的区域和标题 **Gui** 组成。**mainloop** 运行事件循环，等待用户操作，然后做出相应的反应。这是一个无限循环，一直运行到用户关闭窗口，或者按下 **Control-C**，或者其他能使程序终止的操作。

这个 **Gui** 没有做什么事情，因为它没有任何的控件。控件是构成 GUI 的元素，包括：

Button 按钮： 包含文本或图像的控件，当被按压时，产生一个动作。

Canvas 画布： 能够显示直线，矩形，圆和其他图形的区域。

¹译注：还有一个非常流行的就是 **pyGtk**

Entry 输入框: 用户可以输入文本的区域。

Scrollbar 滚动条: 控制其他空间可见部分的空间。

Frame 框: 容纳其他控件的容器, 通常是不可见的。

创建 **Gui** 时的空白灰色区域就是一个框。当新建一个控件, 就会被加到这个框。

19.2 按钮和回调

bu 方法创建一个按钮空间:

```
button = g.bu(text='Press me.')
```

bu 方法的返回值是按钮对象。框里的按钮是这个对象的图形化显示; 可以通过调用按钮的方法控制按钮。

bu 可以通过 32 个参数控制按钮的外形和功能, 这些参数叫做选项。除了提供全部的 32 个选项, 你可以提供关键的参数, 像 `text='Press me.'`, 指定你需要的选项即可, 其他的使用缺省值。

当向一个框添加控件时, 框就变被”收缩包裹”了, 也就是框收缩成按钮般大小。如果想添加更多的空间, 框自动增长来安排他们。

la 方法创建标签控件:

```
label = g.la(text='Press the button.')
```

缺省情况下, **Tkinter** 从上至下安排空间, 然后使其居中。我们不久将会看到如何覆盖这种行为。

如果按下按钮, 你会看到它也没有做什么事情。那是因为你还没有“启动”它, 也就是说, 你还没有告诉它做什么!

控制一个按钮行为的选项是 **command**。**command** 的值是一个函数, 它在按钮按下时候执行。比如, 下面是一个函数创建一个新的标签:

```
def make_label():  
    g.la(text='Thank you.')
```

现在我们可以创建一个按钮, 把这个函数作为 **command**:

```
button2 = g.bu(text='No, press me!', command=make_label)
```

当按下这个按钮, 程序执行 **make_label**, 一个新的标签就会出现。

command 的值是一个函数对象, 也叫回调函数, 因为当你调用 **bu** 创建一个按钮, 用户按下按钮时, 执行流回调了。

这种控制流是事件驱动编程的特点。用户动作, 像按下按钮和击键, 叫做事件。在事件驱动编程中, 执行流是由用户动作控制而不是程序员。

事件驱动编程的最大挑战在于为任何的用户动作建立一系列的控件及回调函数 (至少更产生适当的错误信息)。

Exercise 19.1 编写一个程序，创建只有一个按钮的 GUI。当按钮被按下时，程序创建第二个按钮。当第二个按钮被按下时，应该创建一个标签，显示"Nice job!"。

如果按了多次按钮，会出现什么情况？

可以参看我的解答thinkpython.com/code/button_demo.py

19.3 画布控件

画布是用途最多的空间之一，创建一个可以绘制直线，圆，和其他形状的区域。如果你做了练习??，你已经熟悉了画布。

ca 创建了一个新的画布：

```
canvas = g.ca(width=500, height=500)
```

width 和 height 是用像素表示的画布尺度。

在创建了一个控件之后，仍然可以通过 config 方法来修改选项的值。比如，bg 选项改变背景颜色：

```
canvas.config(bg='white')
```

bg 的值是颜色名。不同的 Python 拥有不同的合法颜色名，但是所有的 python 实现都会提供至少如下的颜色名：

```
white    black
red      green    blue
cyan     yellow   magenta
```

画布上的图形叫做项。比如，画布方法 circle 绘制（你猜是这样）一个圆：

```
item = canvas.circle([0,0], 100, fill='red')
```

第一个参数是一个坐标对，指明了圆心的位置；第二个是半径。

Gui.py 提供了一个标准的笛卡尔坐标系，原点在画布的中央，y 正半轴向上。这个其他的图像系统不一样，他们的原点在左上角，y 正半轴向下。

fill 选项指明圆应该用红色来填充。

circle 的返回值是一个项对象，提供了修改画布上项的方法。蔽日，可以使用 config 改变圆的任意一个选项：

```
item.config(fill='yellow', outline='orange', width=10)
```

width 是用像素表示的轮廓厚度；outline 是颜色。

Exercise 19.2 编写一个程序创建一个画布和按钮。当用户按下按钮，应该在画布上画一个圆。

19.4 坐标序列

`rectangle` 方法接受坐标序列指明对顶角的位置。下面这个例子绘制了一个绿色的矩形, 坐下角在原点, 右上角在 (200,100):

```
canvas.rectangle([[0, 0], [200, 100]],  
                 fill='blue', outline='orange', width=10)
```

这种指定角的方法叫做界定盒子, 因为, 两个点界定了一个矩形。

`oval` 接受一个界定的盒子, 在矩形里绘制一个椭圆。

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

`line` 接受坐标序列, 绘制线段连接各个点。下面这个例子绘制了三角形的两个边:

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

`polygon` 接受同样的参数, 但是它绘制了最后一条边 (如果有必要), 并且填充它:

```
canvas.polygon([[0, 100], [100, 200], [200, 100]],  
               fill='red', outline='orange', width=10)
```

19.5 更多的控件

`Tkinter` 提供了两个控件让用户输入文本: 一个输入框, 只能单行输入, 和文本控件, 可以输入多行。

`en` 创建一个输入框:

```
entry = g.en(text='Default text.')
```

`text` 选项允许你在输入框创建时把文本放进输入框。`get` 方法返回文本框的内容 (可能被用户改变了):

```
>>> entry.get()  
'Default text.'
```

`te` 创建一个文本控件:

```
text = g.te(width=100, height=5)
```

`width` 和 `height` 是空间中字符数和行数的大小。

`insert` 把文本插入文本控件:

```
text.insert(END, 'A line of text.')
```

`END` 是一个特别的索引, 代表文本框里的最后一个字符。

你可以指定用点索引 (`dotted index` 来插入字符, 像 1.1, 小数点前面的数字代表行数, 后面的代表列数。下面的例子把 'nother' 加到第一行第一个字符之后。

```
>>> text.insert(1.1, 'nother')
```

`get` 方法从文本控件读取文本；接受起始索引作为参数。下面的例子返回文本控件的所有内容，包括换行符：

```
>>> text.get(0.0, END)
'Another line of text.\n'
```

`delete` 方法从文本控件里移除文本；下面的例子删除除了开头的两个字符：

```
>>> text.delete(1.2, END)
>>> text.get(0.0, END)
'An\n'
```

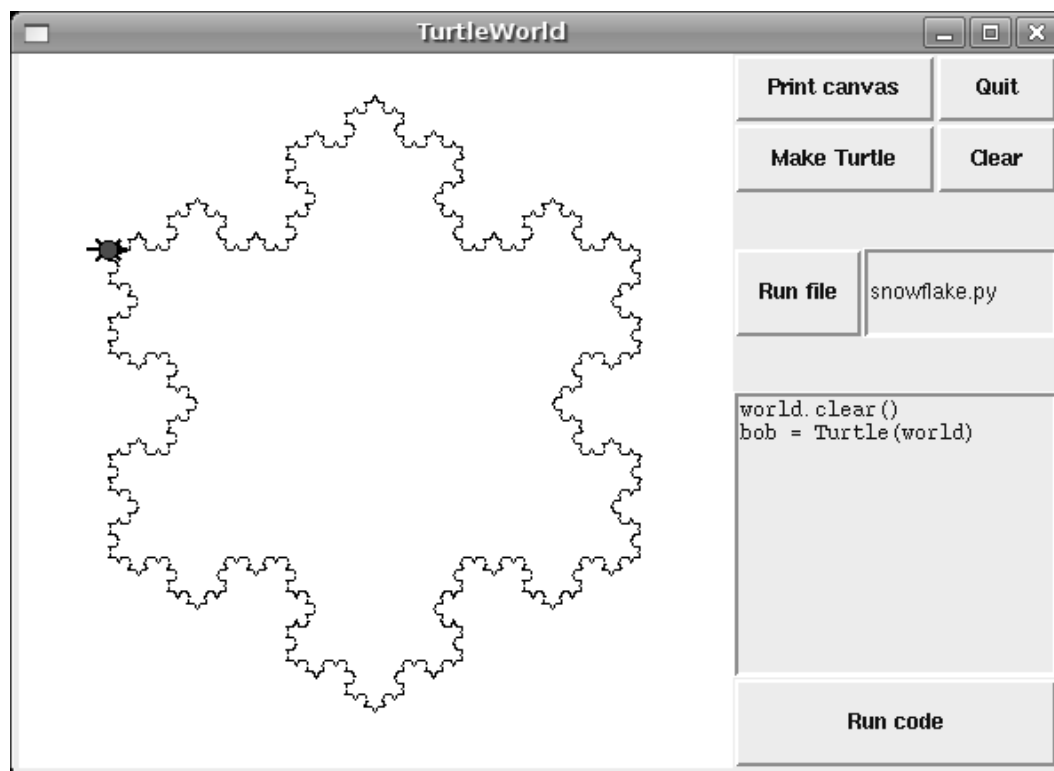
Exercise 19.3 修改练习 19.2 的解答，增加一个输入框和按钮。当用户按下新增加的按钮时，程序从输入框读取一个颜色名，用它改变圆的填充色。用 `config` 修改圆，不要新建一个圆。

你的程序应该能处理没有无颜色名或者颜色名不合法的情况。

参看我的解答 thinkpython.com/code/circle_demo.py.

19.6 排列控件

迄今，我们只是上下安排控件，但是大多数的 GUIs 的布局都是很复杂的。比如，下面是一个稍微有点复杂的 `TurtleWorld` (参看 4 章节)。



这个部分给出了创建这个 GUI 的代码，分解成一系列的步骤。可以下载完整的程序 thinkpython.com/code/SimpleTurtleWorld.py。

在顶级, 这个 GUI 包含了两个控件 --- 一个画布和一个框 --- 并行排列着。所以第一步是创建行。

```
class SimpleTurtleWorld(TurtleWorld):
    """This class is identical to TurtleWorld, but the code that
       lays out the GUI is simplified for explanatory purposes."""

    def setup(self):
        self.row()
        ...
```

`setup` 是创建并布局控件的函数。布局控件叫做包装。

`row` 行创建了一个行框, 使它成为当前框。知道这个框被关闭或者新的框被船舰, 所有后创建的控件都被包装在这行里。

下面是创建画布和列框, 容纳其他控件的代码:

```
self.canvas = self.ca(width=400, height=400, bg='white')
self.col()
```

列框里的第一个控件是格子框, 它包含了四个两两相邻的按钮。

```
self.gr(cols=2)
self.bu(text='Print canvas', command=self.canvas.dump)
self.bu(text='Quit', command=self.quit)
self.bu(text='Make Turtle', command=self.make_turtle)
self.bu(text='Clear', command=self.clear)
self.endgr()
```

`gr` 创建网格; 参数是列的数目。网格里的控件是按照从左至右, 从上到下安排的。

第一个按钮使用 `self.canvas.dump` 作为回调函数; 第二个使用 `self.quit`。有三种绑定方法 (bound methods), 意味着他们和一个特定的对象绑定在一起。当他们被调用, 他们是作用在该对象上。

列的下一个控件是行框, 包含了一个按钮和输入框。

```
self.row([0,1], pady=30)
self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5)
self.endrow()
```

`row` 的第一个参数是引力列表, 决定了控件之间多余的空间如何分配。`[0,1]` 表明所有的空间都分配给第二个控件, 这里是输入框。如果你运行这段代码, 改变窗口大小, 将会看到输入框变大而按钮不变。

选项 `pady` 在 `y` 轴方向填充, 上下分别增加 30 像素的空间。

`endrow` 结束添加控件, 所以以后创建的可空间会被包装在列框。`Gui.py` 保存了框的栈:

- 当使用 `row, col`, 或者 `gr` 创建一个框, 程序进入栈顶, 并且变为当前框。
- 当使用 `endrow, endcol` 或 `endgr` 来关闭一个框, 程序弹出栈顶, 把前一个框置为当前框。

`run_file` 方法读取输入框的内容，把它作为文件名，读取文件，传递给`run_code`。`self.inter` 是一个解释器对象，能够接受一个字符串，并且把它作为 Python 代码执行。

```
def run_file(self):
    filename = self.en_file.get()
    fp = open(filename)
    source = fp.read()
    self.inter.run_code(source, filename)
```

最后两个控件是文本控件和按钮：

```
self.te_code = self.te(width=25, height=10)
self.te_code.insert(END, 'world.clear()\n')
self.te_code.insert(END, 'bob = Turtle(world)\n')

self.bu(text='Run code', command=self.run_text)
```

`run_text` 和`run_file` 相似，除了它从文本控件接受代码，而不是从文件：

```
def run_text(self):
    source = self.te_code.get(1.0, END)
    self.inter.run_code(source, '<user-provided code>')
```

不幸的是，控件布局的方式和其他语言是有差异的，甚至在 Python 的不同模块里也是有差异的。Tkinter 自己也提供了 3 种不同的布局控件的方式。这些方法叫做几何管理器。这部分我演示的是“grid”几何管理器；其他的两种叫做“组装”和“放置”。

幸运的是，这部分的大多数概念对其他 GUI 模块和其他语言也是适用的。

19.7 菜单和可召唤的

菜单按钮是一个看起来像按钮的控件，但是当按下它时，它弹出菜单。用户选择一个项以后，菜单消失。

下面是创建一个颜色选择菜单按钮的代码（可以从这个[下载](http://thinkpython.com/code/menubutton_demo.py)）：

```
g = Gui()
g.la('Select a color:')
colors = ['red', 'green', 'blue']
mb = g.mb(text=colors[0])
```

`mb` 创建一个菜单按钮。开始，按钮上的文本是缺省颜色名。下面的循环为每个颜色创建了一个菜单项：

```
for color in colors:
    g.mi(mb, text=color, command=Callable(set_color, color))
```

`mi` 的第一个参数是联系这些项在一起的菜单按钮。

`command` 选项是可调对象，这个是个新内容。迄今为止，我们已经看到函数和绑定方法作为回调函数，如果不需要传递参数给函数，这个工作的很好。否则，你必须创建一个包含函数（像`set_color`）及其参数（像 `color`）的可调对象。

可调用对象存储了函数的引用和参数作为属性。然后，当用户点击菜单项的时候，回调函数调用函数并且传递存储的参数。

下面是 `set_color` 函数：

```
def set_color(color):
    mb.config(text=color)
    print color
```

当用户选择一个菜单项时，`set_color` 被调用，它重新配置了菜单按钮显示新选择的颜色，同时也打印了颜色。如果你试着运行这个例子，你可以确定当你选择一个项时 `set_color` 被调用，当创建可调用对象时没有被调用。

19.8 Binding 绑定

绑定就是控件，事件和回调函数之间的联系：当一个事件（像按下按钮）在一个控件上发生时，回调函数被调用。

很多控件都有缺省的绑定。比如，当你按下按钮，缺省的绑定改变了按钮的样子，看起来像被压平了一样。当释放按钮，绑定恢复了按钮的样子，然后调用回调函数（由 `command` 选项指定）

可以使用 `bind` 方法覆盖缺省的绑定，或者增加一个新的绑定。比如，下面的代码为画布创建了一个新的绑定（可以从这儿下载 thinkpython.com/code/draggable_demo.py）：

```
ca.bind('<ButtonPress-1>', make_circle)
```

第一个参数是一个事件字符串；这个事件在用户按下鼠标的左键时，发出。其他的鼠标事件包括 `ButtonMotion`，`ButtonRelease` 和 `Double-Button`。

第二个参数是事件处理器。事件处理器是一个函数或者绑定方法，像回调函数一样，但是有一个重要的不同就是事件处理器接受一个事件对象作为参数。下面是一个例子：

```
def make_circle(event):
    pos = ca.canvas_coords([event.x, event.y])
    item = ca.circle(pos, 5, fill='red')
```

事件对象包含了事件类型和一些细节等信息，像鼠标指针的坐标。这个例子中我们需要的信息是鼠标点击的位置。这些值都是以像素坐标保存的，由底层的图像系统定义。`canvas_coords` 方法把他们转换成 "Canvas coordinates"，这个值才能在画布方法中使用，像 `circle`。

对于输入框来是哦，绑定 `<Return>` 事件是很常见的，当用户按下 `Return` 键时，就会发射。比如，下面的例子创建了一个按钮和输入框：

```
bu = g.bu('Make text item:', make_text)
en = g.en()
en.bind('<Return>', make_text)
```

用户在输入框里输入时，当按下按钮或者用户敲击 `Return` `make_text` 就被调用。为了使这个能公正常工作，我们需要一个能够被调用的函数作为 `command`（无参数）或者作为事件处理器（`Event` 作为参数）：

```
beforeverb
```

```
def make_text(event=None):
    text = en.get()
    item = ca.text([0,0], text)
```

`make_text` 获取输入框的内容，并把文本显示在画布上。

也可以给画布项创建绑定。下面是类 `Item` 子类 `Draggable` 的定义。`Item` 提供了绑定，能够实现拖放。

```
class Draggable(Item):

    def __init__(self, item):
        self.canvas = item.canvas
        self.tag = item.tag
        self.bind('<Button-3>', self.select)
        self.bind('<B3-Motion>', self.drag)
        self.bind('<Release-3>', self.drop)
```

初始化方法接受一个 `Item` 作为参数。它复制了 `Item` 的属性，然后为三个事件创建绑定：按下按钮，按钮移动和按钮释放。

事件处理器 `select` 存储当前事件的坐标和项的原先颜色，然后把颜色改成黄色：

```
def select(self, event):
    self.dragx = event.x
    self.dragy = event.y

    self.fill = self.cget('fill')
    self.config(fill='yellow')
```

`cget` 代表“得到配置”，它接受选项名，然后返回选项的值。

`drag` 计算相对于起点移动的距离，更新存储的坐标，然后移动项。

```
def drag(self, event):
    dx = event.x - self.dragx
    dy = event.y - self.dragy

    self.dragx = event.x
    self.dragy = event.y

    self.move(dx, dy)
```

计算是在像素坐标中进行，没有必要转换成画布坐标。

最后 `drop` 恢复项的原先颜色：

```
def drop(self, event):
    self.config(fill=self.fill)
```

你可以使用 `Draggable` 类为存在的项添加拖放功能。比如，下面是一个改编的 `make_circle`，使用 `circle` 创建一个项，并且使用 `Draggable` 使得他可以拖拉：

```
def make_circle(event):  
    pos = ca.canvas_coords([event.x, event.y])  
    item = ca.circle(pos, 5, fill='red')  
    item = Draggable(item)
```

这个例子演示了继承的好处：你可以修改父类的能力而不修改它的定义。如果你想改变定义在模块里但还没有编写的行为，这招非常有效。

19.9 调试

GUI 编程的一个挑战是跟踪什么事情在 GUI 创建时发生，什么事情在响应用户事件发生。

举一个例子，当你设置一个回调函数，很常见的一个错误就是调用函数而不是传递它的引用。

```
def the_callback():  
    print 'Called.'  
  
g.bu(text='This is wrong!', command=the_callback())
```

如果你运行这段代码，你将会看到它立即调用 `the_callback`，然后，创建一个按钮。当按下按钮，什么事情也不发生，因为 `the_callback` 的返回值是 `None`。通常，当你创建 GUI 的时候，你不想调用一个回调函数；它只在随后响应用户事件中调用。

GUI 编程的另外一个挑战是你无法控制执行流。哪部分程序执行和他们的执行的顺序由用户动作决定。也就是说，必须设计程序能够对任何的事件进行正确的处理。

比如，练习 `circle2` 的 GUI 有两个控件：一个创建 `Circle` 控件，另外一个改变 `Circle` 的颜色。如果用户创建圆，并且改变了个改变了圆的颜色，没有问题！但是如果用户改变了不存在的圆的颜色怎么办？或者创建了多个圆？

随着控件数目的增加，就更难想出所有可能的事件了。一种处理的方式是把系统的状态封装在一个对象里，然后考虑：

- 可能的状态是什么？在 `Circle` 例子中，我们考虑两种状态：用户创建第一个圆的前后状态。
- 在每个状态里，什么事件会发生？在例子中，用户要么按下按钮，要么退出。
- 对于每个状态—事件对，期待的结果是什么？因为有两种状态和两个按钮，有四种状态—事件对需要考虑。
- 什么可以导致从一个状态到另一个状态的转变。这种情况下，当用户创建第一个圆时，发生了第一个转变。

你也许会发现定义检查包含所有的事件不变量是一个有用的方法。

这种方式对于 GUI 编程能够帮助你写出正确的代码，而不需要花费事件测试每一种可能的用户事件。

19.10 术语表

GUI: 用户图形接口。[GUI]

widget 控件: 组成 GUI 的元素之一, 包括按钮, 菜单, 文本输入域等等。

option 选项: 控制控件外表或者功能的值。

keyword argument 关键参数: 表明参数是函数调用的参数。

bound method 绑定方法: 和特定的实例联系在一起的方法。

event-driven programming 事件驱动编程: 执行流由用户动作决定的编程方式。

event 事件: 一个用户动作, 比如, 鼠标点击或者击键, 致使 GUI 发生反应。

event loop 事件循环: 等待用户动作的无限循环。

item 项: 画布控件上的图形元素。

bouding box 界定盒子: 占据着一定位置的矩形, 通常指明了对顶角的位置。

pack 包装: 安排显示 GUI 元素。

geometry manager 集合管理器: 包装控件的系统。

binding 绑定: 控件, 事件和事件处理器的联系。当事件发生时, 事件处理器被调用。

19.11 练习

Exercise 19.4 这个联系要求你编写一个图像查看器。下面是一个简单的例子:

beforeverb

```
g = Gui()
canvas = g.ca(width=300)
photo = PhotoImage(file='danger.gif')
canvas.image([0,0], image=photo)
g.mainloop()
```

PhotoImage 读取文件并返回一个 Tkinter 可以显示的 PhotoImage 对象。Canvas.image 把图像放置在画布上, 按照给定的坐标居中显示。也可以把图像放在标签, 按钮和其他的控件上:

```
g.la(image=photo)
g.bu(image=photo)
```

PhotoImage 只能处理为数不多的图像格式, 像 GIF 和 PPM。但是我们可以使用 Python Imaging Library(PIL) 读取文件。

PIL 模块的名字是 Image, 但是 Tkinter 定义了一个同样的名字。为了避免冲突, 可以使用 import...as 语句:

beforeverb

```
import Image as PIL
import ImageTk
```

第一行, 导入了 `Image`, 赋给了它一个局部名字 `PIL`。第二行导入了 `ImageTk`, 可以把 `PIL` 图形转换成 Tkinter 的 `PhotoImage`。下面是一个例子:

```
image = PIL.open('allen.png')
photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. 从 thinkpython.com/code 下载 `image_demo.py`, `danger.gif` and `allen.png`。运行 `image_demo.py`。你可能需要安装 `PIL` 和 `ImageTk`。他们很可能已经包含在软件仓库里, 但是如果没有, 从这儿获得 pythonware.com/products/pil/。
2. 在 `image_demo.py` 里, 把第二个 `PhotoImage` 从 `photo2` 改成 `photo`, 重新运行程序。你将会看到第二个 `PhotoImage`, 但是看不到第一个。

问题在于, 当你重新给 `photo` 复制时, 就覆盖掉了第一个 `PhotoImage` 的引用, 随后它就消失了。同样的问题也会出现在你把 `PhotoImage` 赋给一个局部变量; 函数结束时, 它就销毁了。

为了避免这个问题, 你必须存储指向每一个 `PhotoImage` 的引用。你可以使用一个全局变量或者把 `PhotoImage` 存储在一个数据结构里, 或者作为一个对象的属性存在。

这种方式可能很令人沮丧, 这也就是我为什么警告你的原因 (也是为什么例子图片显示 “Danger!”)。

3. 从这个例子开始, 编写一个程序, 接受一个目录作为参数, 循环遍历所有文件, 显示所有 `PIL` 认为是图片的文件。你可以使用 `try` 语句抓住 `PIL` 不认识的文件。
当用户点击图形, 程序必须显示下一个图形。
4. `PIL` 提供很多方法操作图片。可以参考 pythonware.com/library/pil/handbook。作为一个小小的挑战, 选用一些方法, 在 GUI 中应用到图片。

可以下载一个简单的解答 thinkpython.com/code/ImageBrowser.py。

Exercise 19.5 矢量图编辑器是一个允许用户在屏幕上拖拉编辑图形并且能够以矢量格式 (比如 `Postscript` 和 `SVG`) 输出图形的程序²。

用 `Tkinter` 编写一个简单的矢量图编辑器。至少: 它能允许用户画直线, 圆和矩形, 必须使用 `Canvas.dump` 输出 `Postscript` 格式的图形。

作为挑战, 你可以允许用户选择和调整画布上项的大小。

Exercise 19.6 使用 `Tkinter` 编写一个简单的网页浏览器。要求必须有一个文本空间, 这样用户可以输入 `URL`, 还有一个画布显示网页的内容。

你可以使用 `urllib` 模块下载文件 (参看练习 14.5), 使用 `HTMLParser` 模块分析 `HTML` 标签。(参看 docs.python.org/lib/module-HTMLParser.html)。

至少: 你的浏览器能够处理纯文本文件和超连接。作为一个挑战, 你可以处理背景颜色, 文件格式化标签和图像。

²参考 wikipedia.org/wiki/Vector_graphics_editor。

Appendix A

调试

程序中会出现不同的错误，加以区分有助于加快对错误的跟踪。

- 语法错误是 **Python** 在将源代码转换为字节码时产生的。它们通常说明程序的语法有错误。例如：在 `def` 语句后忽略冒号会产生类似 `SyntaxError: invalid syntax` 的信息。
- 运行时错误由解释器在程序运行出错时产生。大多数的运行时错误消息包含错误发生的位置和正在执行的函数。例如，一个无穷递归的函数最终导致运行时错误 `“maximum recursion depth exceeded”`。
- 语义错误指程序执行过程中没有产生出错信息，但程序没有做正确的工作。例如，一个表达式没有按照你期望的顺序进行求值，导致错误的结果。

调试的第一步是找出你遇到了什么类型的错误。虽然下面的章节根据错误类型组织的，但一些方法使用于多种情况。

A.1 语法错误

当你找到原因后语法错误一般很容易修正。不幸的是，错误消息常常不是很有帮助。最常见的消息是 `SyntaxError: invalid syntax` 和 `SyntaxError: invalid token`，它们都不包含很多信息量。

另一方面，错误消息告诉你问题发生在程序中的什么位置。事实上，当 **Python** 遇到问题时它将告诉你，但这不一定是错误的地方。有时错误在错误消息位置的前面，通常是前几行。

如果你增量的开发程序，你会清楚错误在哪里。它就是最新添加的代码。

如果你从书中复制一段代码，那么从仔细地检查你的代码和书中的代码开始。检查每一个字符。同时记住书本也会有错误，如果你遇到类似语法错误，也许它就是。

下面给出一些避免常见语法错误的方法：

1. 避免使用 **Python** 关键字作为变量名。
2. 确保你在每个复合语句头的尾部加上了冒号，包括 `for`，`while`，`if` 和 `def`。

3. 确保代码中的字符串都有匹配的引号。
4. 如果你用三重引号标记多行字符串，确保你正确的结束该字符串。一个未终止的字符串会导致程序尾部 `invalid token` 错误，或者程序接下来的部分都被认为字符串，直到下一个字符串。在第二种情况，Python 可能不会产生一个错误消息！
5. Python 将未封闭的运算符 `--(`，`{`，或 `[`---的下一行作为当前语句的一部分。通常第二行将产生错误。
6. 在条件语句中使用传统的 `=` 而不是 `==`。
7. 确保每行缩进正确。Python 可以处理空格和 `tabs`，但如果你混淆它们会产生错误。避免这个问题最好的方法是使用适合 Python 编辑、会自动缩进的文本编辑器。

如果这些没有效，继续阅读下一章节...

A.1.1 我做了修改但是没有任何区别

如果解释器报错，但你找不到错误，有可能你和解释器看到不是相同的代码。检查你的编程环境，确保你正在编辑的文件是 Python 将要运行的。

如果你不确定，在程序的开始位置添加明显的故意的语法错误，再次运行，如果解释器没有找到这个错误，说明你不在运行新的程序。

下面是一些可能的出错原因：

- 你编辑了文件，运行前忘记了保存。有的编程环境会为你自动保存，有的不会。
- 你修改了文件的名称，但仍然运行老的名字。
- 你的开发环境配置有误。
- 如果你在编写模块并使用 `import`，确保你写的模块名不同于 Python 标准模块名。
- 如果你使用 `import` 读取一个模块，记得重启解释器或使用 `reload` 来读取一个修改过的文件。如果你再次导入模块，解释器将不做任何事。

如果你陷入困境找不到出错的原因，一个方法是从一个类似“Hello, World!”的新程序开始，确保你从一个已知的可运行的程序开始。然后逐步添加原程序中的代码到新程序。

A.2 运行时错误

当你的程序没有语法错误，Python 可以编译并运行。这时可能发生什么错误呢？

A.2.1 我的程序什么也没做

这个问题通常发生在你的文件包含函数和类，但没有任何调用执行的语句。也许你故意这样，因为你仅仅打算导入这个模块来提供函数和类。

如果这不是故意的，确保你调用一个函数来开始执行，或从一个交互的命令提示行下执行。参见下面的“执行流程”章节。

A.2.2 我的程序挂起了

如果一个程序停止，但上去什么也没做，我们称“挂起”。通常意味着程序陷入一个无限循环或者无线递归。

- 如果你怀疑某个循环引起这个问题，在循环开始时添加 `print` 语句打印“进入循环”，在循环结束时打印“`exiting the loop`”退出循环。
运行程序，如果你的到第一条消息，但没有第二条消息，你得到一个无限循环。参见“无限循环”章节。
- 大多数时候，无限循环会令程序运行一段时间，然后产生“`RuntimeError: Maximum recursion depth exceeded`”的错误。如果是这样，参考下面的“无限递归”章节。
如果你没有得到这样的错误信息，但你怀疑某个递归方法或函数有问题，你仍可以使用“无限递归”章节中提到的方法。
- 如果这些方法都没有，尝试测试其他的循环、递归的函数和方法。
- 如果还是没有效果，有可能你不清楚程序执行的流程。参考下面的“执行流程”章节。

无限循环

如果你有一个无限循环并且你认为这个循环导致了问题，在循环体结束的地方添加 `print` 语句打印条件语句中的变量值和条件值。

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，每个循环你都会看到 3 行输出。对于最后一次循环，条件应该为 `false`。如果循环不断执行，你可以看到 `x` 和 `y` 的值，也许能够发现为什么它们没有被正确的更新。

无限递归

大多数时候，无限循环会令程序运行一段时间，然后产生“`RuntimeError: Maximum recursion depth exceeded`”的错误。

如果你怀疑一个函数或方法导致了无限递归，首先检查存在一个基本状态。换言之，应该有一个状态使函数或方法直接返回，而不是再次递归调用。如果不是，你需要重新思考算法，并设计一个基本状态。

如果存在一个基本状态，但程序似乎没有运行到这个状态，你可以在函数或方法开始的部分添加 `print` 语句打印参数。现在当你运行程序，每次函数或方法被调用时，你将看到几行关于参数的输出。如果参数没有向基本状态靠拢，你也许会发现为什么这样。

执行流程

如果你不确定程序的执行流程，在每个函数开始的地方加入 `print` 语句，打印类似“entering function foo”的语句，其中 `foo` 是函数的名字。

现在当你运行程序，它将打印每个函数被调用的追踪。

A.2.3 当我运行程序，我得到一个异常

有时程序运行时出错，Python 将打印异常的名字、问题发生的位置和回溯。

回溯识别当前运行的函数，然后识别调用该函数的函数，以此类推。换言之，它追踪了你到达这个错误所经过的函数调用。同时它也包括这些调用在文件中的位置。

第一步是检查代码中对应的出错的位置，或许你能发现错误。下面是常见的运行时错误：

名字错误：你试图使用一个不在当前环境下的变量。记住局部变量是局部的，你不能在定义它们的函数的外部引用它们。

类型错误：可能的几个原因是：

- 你试图不适当的使用值。例如：使用一个非整数的值作为字符串、列表或元组的下标。
- 格式字符串和用于转换的对象不匹配。这发生在或者数量不相同，或者调用了一个非法的转换。
- 你调用函数或方法时传递的参数个数有误。对于方法，检查方法定义，确保第一个参数是 `self`，然后检查方法调用，确保你对一个对象调用方法，并正确的提供其他参数。

键错误：你试图使用字典中不存在的键访问对应的元素。

属性错误：你试图访问一个不存在的属性或方法。检查拼写！你可以使用 `dir` 来列举存在的属性。

如果属性错误指出一个对象是空类型，及它是空的。一个常见的原因是函数结束时忘记返回值。如果在返回尾部没有 `return` 语句，函数将返回一个 `None`。另一个原因是使用类似列表中的 `sort` 方法，它返回 `None`。

下标错误：用来访问列表、字符串或元组的下标超过长度减 1。在错误发生的前一行使用 `print` 语句显示下标值和序列的长度，检查两个值是否正确。

Python 调试器 (pdb) 允许你在错误前检查程序状态，有助于追踪异常。你可以在 docs.python.org/lib/module-pdb.html 阅读有关 `pdb` 的内容。

A.2.4 我添加了太多的 `print` 语句，输出令我应接不暇

使用 `print` 语句调试的一个问题是你会被大量的输出信息掩埋。有两个处理方法：简化输出或简化程序。

简化输出可以删除或注释不用的 `print` 语句，或将它们合并，或格式化输出使得它们容易理解。

简化程序你可以做一下几件事。首先缩小程序处理的问题的规模。例如，如果你在搜索一个列表，搜索一个小的列表。如果程序从用户读取输入，输入最简单的参数。

第二，整理程序，删除死区代码，使程序易读。例如，如果你认为问题深嵌在程序中，试图用简单的结构重写那部分。如果你怀疑一个大函数有错误，试图将它分割成小函数，然后分别测试。

通常寻找最小测试案例的过程让你找到错误。如果你发现一个程序对于一个情况适用，对另一个情况不使用，这将给你一些线索。

同样，重写一段代码有助于发现一些微小的错误。如果你做了一个你认为不会影响程序的修改，但事实上影响了，这个方法可以给你警戒。

A.3 语义错误

在某种程度上，语义错误时最难调试的，因为解释器不能提供任何错误信息。你所知道的只有程序应该怎么做。

第一步是建立程序代码和你见到的行为之间的联系。你需要假设程序实际上做了什么。一个主要困难在于计算器运行的太快了。

你常希望你可以降低程序的速度，是人类可以跟上，通过使用调试器，你可以实现这步。但是在关键地方加入一些 `print` 语句所用的时间通常短于建立调试器，加入删除断点，然后“逐步”运行程序直到错误发生。

A.3.1 我的程序不工作

你需要问自己这些问题：

- 有没有什么程序应该做却没有发生？找到执行该函数的代码段，确保程序被执行。
- 有没有什么不应该发生的发生了？找到执行该函数的代码段，查看它是否执行了？
- 有没有代码的执行效果与你期望的不同？确保你理解有问题的代码，尤其是包含调用其他 **Python** 模块中的函数或方法。阅读你调用的函数的文档，用一些简单的例子进行测试。

在编程时，你心中需要有一个关于程序如何工作的模型。如果你的程序没有按照你期望的工作，很可能问题不在于程序，而在于你心中的模型。

修正你心中的模型的最好的方法是将程序分割为不同部分（通常是函数和方法），并分别测试。一旦你发现了模型和现实的差异，你就可以解决问题。

当然，在开发的过程中你需要建立并测试组件。如果你遇到了问题，只有一小部分新的代码是不确定正确性的。

A.3.2 我写了一个很长的表达式，它没有按照我期望的工作

编写复杂的表达式是合理的，如果它们可读。但是它们调试起来很困难。通常我们将一个复杂的表达式分割成一系列的临时变量的赋值。

例如：

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

可以重写为:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

这个明晰的版本更适于阅读, 因为变量名提供了额外的文档, 同时调试也更容易, 因为你可以检查中间变量的类型和它们的值。

大的表达式的另一个问题是计算的顺序不一定是你期望的。例如, 如果你将表达式 $\frac{x}{2\pi}$ 翻译为 Python, 你也许会写成:

```
y = x / 2 * math.pi
```

这是不正确的, 因为乘法和除法有相同的优先级, 因此是从左往右计算的, 这个表达式计算的是 $x\pi/2$ 。

调试表达式的一个好的方法是添加括号, 使得计算顺序简洁明了:

```
y = x / (2 * math.pi)
```

当你不确定计算优先级是, 使用括号。不仅程序将工作正常 (按你的要求执行), 同时也让那些没有记住优先级规则的人阅读起来更方便。

A.3.3 我的函数或方法没有按照我期望的返回

如果你的 `return` 语句包含一个复杂的表达式, 你没有机会在返回前打印返回值。同样你可以使用临时变量。例如: 对于

```
return self.hands[i].removeMatches()
```

你可以写成:

```
count = self.hands[i].removeMatches()
return count
```

现在你在返回前可以打印 `count` 的值。

A.3.4 我实在是卡住了, 我需要帮助

第一, 尝试离开电脑几分钟。电脑辐射会对大脑产生影响, 导致下列几种症状:

- 沮丧和愤怒
- 迷信的人认为“电脑讨厌我”, 并神奇的相信“程序仅当我向后戴着帽子时才工作正常”。
- 随机漫步编程 (用各种可能的方法编程, 并选择工作正常的那个)。

如果你发现你有以上任意一种症状, 站起来走一走。当你心绪平静时, 思考一下程序。它是做什么的? 什么肯能造成了这种行为? 上次可以工作的程序是什么时候? 下一步做什么?

有时找到一个错误很费时间。我常常在我离开电脑, 让思维游荡的时候找到错误。一些找到错误最好的地方有火车上, 浴室里, 以及临睡前。

A.3.5 不，我真的需要帮助

即使最好的程序员也会卡住。有时你在一个程序上工作了太长的时间，因此你难以发现错误。而他人可能一眼就发现问题。

在你向其他人寻求帮助前，你需要做好准备。你的程序需要尽可能简洁，你需要最少的输入来重现错误。你需要在合适的位置加入 `print` 语句，同时输出应可理解。你需要能够以简洁的语言描述问题。

当你想某人求助，你需要提供足够的信息：

- 是否有出错消息？它是什么？指向程序的哪部分？
- 错误出现前你做的最后一步是什么？你写的最后几行是什么？什么新的测试导致了错误？
- 你做了哪些尝试？你学到了什么？

当你找到了错误，花时间想一想你怎么能更快的定位它。下一次你遇到类似的问题，你就可以更快的找到问题。

记住，目标不仅仅是让程序工作，而是学会如何让程序工作。

Index

- 0, 下标开始于, 84
- , 181
- 泛化, 35
- abecedarian, 78
- accumulator
 - histogram, 119
- Ackerman 函数, 56
- add method add 方法, 155
- addition with carrying, 63
- algorithm, 7
 - RSA, 103
 - square root, 64
- algorithm 算法, 63, 122
- aliasing 别名, 139, 141, 158
- alternative executive 选择执行, 39
- ambiguity 二义性, 5
- and operator and 运算符, 38
- anydbm 模块, 131
- append 方法, 86, 91, 94, 164, 165
- argument
 - keyword, 172
 - optional, 99
- argument 参数, 19
- argument 实参, 21, 22, 26
- argument 形式参数, 17
- assert 语句, 149
- assignment
 - multiple, 64
 - multiple 多次, 102
- attribute
 - __dict__, 157
 - initializing, 157
 - instance, 138, 143
- AttributeError, 142
- Austin, Jane, 119
- available colors, 144, 159
- Bacon, Kevin, 135
- Bangladesh, national flag 孟加拉国, 国旗, 143
- base case, 45
- base case 终止条件, 42
- benchmarking 基准程序, 124, 125
- binding 绑定, 178, 181
- bingo, 115
- bisection, debugging by, 64
- bisect 模块, 94
- body, 45
- body 体, 19, 61
- body 函数体, 25
- bool type bool 类型, 38
- boolean expression 布尔表达式, 37, 44
- borrowing, subtraction with, 63
- bouding box, 174
- bouding box 界定盒子, 181
- bound method 绑定方法, 176
- bound method 绑定方法, 181
- bounding box , 143
- bracket
 - squiggly, 95
- branch, 45
- branch 分支, 39
- break statement break 语句, 61
- bug, 7
 - worst, 158
 - worst ever, 182
- Button widget 按钮控件, 172
- calculator 计算器, 8
- calculator, 16
- call graph, 101
- call graph 调用框图, 104
- callable object 可调用对象, 177
- callback 回调, 172, 176--178
- callback 回调函数, 180
- Canvas coordinate 画布坐标, 173, 179
- Canvas item 画布项, 173
- Canvas object, 143
- Canvas widget 画布控件, 173
- Car Talk, 82, 105, 115
- Car Talk 汽车谈话, 82

- Card 类, 162
- carrying, addition with, 63
- chained conditional 链式条件, 39
- chained conditional 链式条件语句, 45
- choice function choice 函数, 118
- class, 143
 - Kangaroo, 158
 - Point, 137, 154
 - Rectangle, 139
 - SimpleTurtleWorld, 176
- class definition 类定义, 137
- class object 类对象, 137, 143
- class 类, 137
- close 方法, 128, 131, 132
- __cmp__ 方法, 163
- cmp 函数, 164
- Collatz conjecture Collatz 猜想, 61
- colon 分号, 19
- color list, 144, 159
- commutativity 可交换, 156
- compile 编译, 6
- comple, 1
- composition 创建, 22
- compositon 创建, 26
- compound statement, 45
- compound statment, 39
- concatenation 连接, 22
- condition 条件, 39, 45, 61
- conditional
 - chained, 39, 45
 - nested, 40, 45
- conditional execution 条件执行, 38
- conditional statement 条件语句, 38, 45
- config method, 173
- consistency check 连贯性测试, 104
- conversion
 - type 转换
类型, 17
- coordinate
 - Canvas, 173
 - pixel, 179
- coordinate sequence 坐标序列, 174
- coordinate ! Canvas, 179
- copy
 - deep, 142
 - shallow, 142
- copy module copy 模块, 141
- copying objects 复制对象, 141
- counter 计数器, 96, 102
- crosswords 纵横组合字谜, 77
- cummings, e. e. 康明思, 3
- Czech Republic, national flag 捷克共和国,
国旗, 144
- data structure 数据结构, 123
- datetime 模块, 150
- Date 类, 150
- debugging
 - by bisection, 64
 - emotional response 调试
情绪反应, 6
 - experimental 调试
实验, 3
- debugging 调试, 6, 7, 25, 43, 81, 103, 124, 142,
157, 180
- declaration 声明, 102, 104
- decrement, 64
- decrement 减量, 60
- deep copy 深拷贝, 142, 143
- deepcopy function deepcopy 函数, 142
- def keyword def 关键字, 19
- default value
 - avouding mutable, 158
- default value 缺省值, 120, 125, 154
- definition
 - class, 137
 - function 定义
函数, 19
- del 运算符, 88
- deterministic, 117
- deterministic 确定的, 125
- development plan
 - problem recognition, 79, 81
 - random walk programming, 125
- diagram
 - call graph 调用框图, 104
 - object, 138, 139, 141, 143
 - stack, 23
 - state, 59, 100, 138, 139, 141
- __dict__ attribute, 157
- dict function dict 函数, 95
- dictionary
 - invert, 99
 - lookup, 98
 - looping with, 97
 - reverse lookup, 98
 - subtraction, 121
 - traversal, 158
- dictionary 字典, 95, 104
- Dijkstra, Edsger, 81

- dispatch
 - type-based, 157
- dispatch ,type-based, 156
- divisibility 整除, 37
- division
 - floor, 44
- divmod, 109, 148
- docstring 文档字符串, 137
- documentation 文档, 7
- dot notation 点记法, 18, 26, 138, 152
- double letters , 82
- Doyle, Arthur Conan, 4
- drag-and-drop 拖放, 179
- DSU pattern, 119
- DSU 模式, 113, 115
- duplicate, 104

- Einstein, Albert, 33
- elif keyword elif 关键字, 39
- ellipses 省略号, 20
- else keyword else 关键字, 39
- email 地址, 108
- embedded object 嵌套对象, 158
- embeded object, 140
 - copying, 141
- embededed object 嵌套对象, 143
- emotional debugging 情绪调试, 6
- encapsulation 封装, 63
- encryption 加密, 103
- Entry widget 输入框, 174
- enumerate 函数, 111
- epsilon , 63
- error
 - runtime, 42, 44
 - runtime 错误
 - 运行时, 3
 - semantic 错误
 - 语义, 3
- error message 错误信息, 3, 6
- eval function eval 函数, 65
- event, 181
- event handler 事件处理, 178
- event loop 事件循环, 171, 181
- Event object, 178
- event string 事件字符串, 178
- event-driven programming, 181
- event-driven programming 事件驱动编程, 180
- event-driven programming 时间驱动编程, 172

- exception
 - AttributeError, 142
 - KeyError, 96
 - NameError, 23
 - OverflowError, 44
 - RuntimeError, 42
 - SyntaxError, 19
 - TypeError, 100, 153
 - UnboundLocalError, 102
 - ValueError, 43, 98
- exception 异常, 3, 7
- exists 函数, 129
- experimental debugging 实验性的调试, 3
- experimental debugging 实验性调试, 124
- expression
 - boolean, 37, 44
- extend 方法, 86

- False special value False 特殊值, 38
- Fermat's Last Theorem 费马最后定理, 45
- fibonacci function fibonacci 函数, 100
- file object 文件对象, 77
- file object 文件对象, 81
- find 函数, 70
- flag 标记, 101, 104
- float function float 函数, 17
- floating-point 浮点, 63
- floor division 地板除, 44
- flow of execution 执行流, 26, 60, 180
- flow of execution 执行流 j, 21
- for 循环, 30
- formal language 正式语言, 4, 7
- for 循环, 68, 85, 111
- frabjous, 51
- frame, 101
- Frame widget, 176
- frame 图, 42
- frame 框图, 26
- frame 框架, 23
- Free Document License, GNU, vi
- Free Documentation License,GNU, v
- frequency
 - word, 125
 - word 单词, 117
- frequency 频率, 97
- fruitful fucntion 结果函数, 26
- fruitful function 卓有成效的函数, 24
- fucntion
 - choice, 118
- fucntion syntax 函数语法, 152

- function
 - deepcopy, 142
 - dict, 95
 - eval, 65
 - fibonacci, 100
 - float 函数
 - float, 17
 - getattr, 158
 - hasattr, 157
 - int 函数
 - int, 17
 - isinstance, 156
 - len, 26, 96
 - log 函数
 - log, 18
 - open, 77, 78
 - randint, 118
 - random, 118
 - raw_input, 43
 - recursive, 41
 - sqrt 函数
 - sqrt, 18
 - str 函数
 - str, 18
 - type, 142
- function argument 函数实参, 21
- function call 函数调用, 17, 26
- function definition 函数定义, 19, 20, 25
- function frame 函数图, 42
- function frame 函数框, 23
- function frame 函数框图, 26, 101
- function object 函数对象, 20, 25, 26
- function parameter 函数形参, 21
- function 函数, 19, 25, 151
- function, fruitful, 24
- function, math 函数, 数学, 18
- function, reason for, 25
- function, trigonometric 函数, 三角, 18
- function, void, 24
- gamma 函数, 54
- GCD (最大公约数), 57
- generalization, 79
- geometry manager, 181
- geometry manager 几何管理器, 177
- get method get 方法, 97
- getattr function getattr 函数, 158
- getcwd 函数, 129
- global statement 全局语句, 102
- global variable
 - update 更新, 102
- global variable 全局变量, 101, 104
- GNU Free Document License, vi
- GNU Free Documentation License, v
- graphical user interface 图形用户接口, 171
- grid 网格, 27
- GUI, 171
- Gui module, 171
- gunzip (Unix 命令), 133
- Hand 类, 166
- hasattr function hasattr 函数, 157
- hash function 散列函数, 100, 104
- hashable 可散列的, 100
- hashable 散列体, 104
- hashtable 散列, 96
- hashtable 散列表, 104
- header 函数头, 25
- header 头, 19
- Hello, World, 5
- help utility 帮助工具, 8
- hexadecimal 十六进制, 138
- histogram
 - random choice, 118, 121
 - word frequencies, 118
- histogram 直方图, 97
- Holems, Sherlock, 4
- homophone 同音词, 105
- HTMLParser module, 182
- hyperlink 超连接, 182
- if statement if 语句, 38
- Image module, 181
- image viewer 图像查看器, 181
- IMDb (网络电影数据库), 135
- immutability 不可变性, 100
- implementation 实现, 97, 104, 123
- import statement import 语句, 26
- import 语句, 133
- in operator in 运算符, 96
- increment, 64, 147
- increment 增量, 60
- increment 增量, 153
- indentation 缩进, 19, 152
- index
 - looping with, 80
- index 索引, 95
- infinite loop 无限循环, 61, 64, 171
- infinite recursion 无穷递归, 42, 45
- init method init 方法, 157

- init method 初始方法, 154
- initialization
 - variable, 64
- initialization (before update) (更新前) 初始化, 60
- init 方法, 162, 164, 166
- instance
 - as argument, 138
 - as return value, 140
- instance attribute 实例属性, 138, 143
- instance 实例, 138, 143
- instantiation 实例化, 138
- int function int 函数, 17
- integer
 - long 长, 103
- interactive mode 交互模式, 7, 24
- interpret, 1
- invariant 不变量, 180
- invert dictionary 反转字典, 99
- in 运算符, 72, 84
- is operator is 运算符, 141
- isinstance function isinstance 函数, 156
- isinstance 函数, 54
- is 运算符, 89
- item
 - Canvas, 173, 181
 - dictionary 字典, 104
- items 方法, 111
- iteration 迭代, 60, 64
- iteration 迭代器, 59
- join 方法, 89, 164
- Kangaroo class Kangaroo 类, 158
- Kevin Bacon Game, 135
- key 关键字, 95
- key 键, 104
- key-value pair 关键字-值对, 95
- key-value pair 键值对, 104
- keyboard input 键盘输入, 43
- KeyError, 96
- keys method keys 方法, 98
- keyword
 - def 关键字
 - def, 19
 - elif, 39
 - else, 39
- keyword argument, 181
- keyword argument 关键参数, 172
- koch curve 柯霍曲线, 46
- Label widget, 172
- language
 - formal 语言
 - 正式, 4
 - natural 语言
 - 自然, 4
 - safe 语言
 - 安全, 3
- len function len 函数, 96
- len 函数, 67
- letter rotation , 105
- Linux, 4
- lipogram 避讳, 78
- list
 - 函数, 88
- literalness 无修饰性, 5
- local variable 局域变量, 22, 26
- log function log 函数, 18
- logarithm 对数, 125
- logical operator 逻辑运算符, 37, 38
- long integer 长整数, 103
- lookup 查询, 104
- lookup,dictionary 查询, 字典, 98
- loop
 - event, 171
 - infinite, 61, 171
 - while, 60
- loop 循环, 61
- looping
 - with dictionaries, 97
 - with indices, 80
- ls (Unix 命令), 132
- mapping 映射, 122
- Markov analysis 马尔可夫分析, 122
- mash-up 混合, 123
- math function 数学函数, 18
- max 函数, 109, 110
- McCloskey, Robert, 68
- MD5 算法, 135
- membership
 - dictionary, 96
 - set, 96
- memo 备忘录, 101, 104
- Menubutton widget 菜单按钮, 177
- metaphor, method invocation 隐喻, 方法调用, 152
- method
 - __str__, 154
 - add, 155

- config, 173
- get, 97
- init, 154
- keys, 98
- radd, 156
- setdefault, 100
- strip, 77, 117
- translate, 117
- values, 96
- method syntax 方法语法, 152
- method 方法, 151, 158
- method, bound, 176
- methods
 - readline, 77
- min 函数, 109, 110
- module
 - copy, 141
- module
 - Gui, 171
 - HTMLParser, 182
 - Image, 181
 - pprint, 104
 - profile, 124
 - random, 118
 - string, 117
 - urllib, 182
 - Visual, 159
 - vpython, 159
 - World, 143
- module object 模块对象, 18
- module 模块, 18, 26
- modulus operator 模操作符, 37
- modulus operator 模运算符, 44
- Monty Python and the Holy Grail, 146
- MP3, 135
- mro 方法, 168
- mutability 可变, 102, 140
- mutable object, as default value, 158
- mutiple assignement 多重赋值, 64
- mutiple assignment 多重赋值, 59, 102
- NameError, 23
- natural language 自然语言, 4, 7
- nested conditional 嵌套条件语句, 45
- nested conditional 嵌套的条件语句, 40
- newline 换行, 43, 59
- Newton's method 牛顿方法, 62
- None special value None 特殊值, 24
- None 特殊值, 48, 55
- not operator not 运算符, 38
- number 数字, random 随机, 117
- object
 - Callable, 177
 - Canvas, 143
 - class, 137
 - copying, 141
 - embedded, 143, 158
 - embeded, 140
 - Event, 178
 - file, 77, 81
 - function, 26
 - function 对象
函数, 20
 - mutable, 140
 - printing, 152
- object code 目标代码, 7
- object diagram 对象图, 138, 139, 141, 143
- object 对象, 137
- object-oriented language 面向对象语言, 158
- object-oriented programming 面向对象编程, 151, 158
- odometer 里程表, 82
- oeprn function open 函数, 77
- open function open 函数, 78
- open 函数, 127, 131
- open 海曙, 130
- operator
 - and, 38
 - in, 96
 - is, 141
 - logical, 37, 38
 - modulus, 44
 - modulus 操作符
模, 37
 - not, 38
 - or, 38
 - overloading, 158
 - relational, 38
- operator overloading 运算符重载, 155
- option, 172
- option 选项, 181
- optional argument, 99
- optional parameter 可变参数, 120
- optional parameter 可选择参数, 154
- or operator or 运算符, 38
- os 模块, 129
- other (parameter name), 153
- OverflowError, 44
- overloading 重载, 158

- override 覆盖, 120, 125, 154
- packing widgets , 181
- packing widgets 包装控件, 176
- palindrome 回文, 80, 82
- parameter
 - optional, 154
 - optional 可选的, 120
 - self, 152
- parameter 形参, 21, 23, 25
- Parentheses
 - parameters in, 22
- parentheses
 - empty 小括号空, 19
 - matching 括号匹配, 3
 - parameters in, 21
- parenthess
 - argument in, 17
- parse 句法分析, 4, 7
- pass statement, 39
- patameter
 - other, 153
- pattern
 - DSU, 119
 - search, 78, 98
- pdb (Python 调试器) , 186
- PEMDAS, 13
- pi, 18, 65
- pickle 模块, 127, 131
- pickling, 131
- pie, 36
- PIL (Python Imaging Library), 181
- pixel coordinate 像素坐标, 179
- plain text 纯文本, 182
- plain text 纯文本文件, 77, 117
- poetry 诗歌, 5
- point ,mathematical 点, 数学, 137
- Point class Point 类, 137, 154
- polymorphism 多态, 157, 158
- popen 函数, 132
- pop 方法, 88, 165
- pprint module pprint 模块, 104
- prefix 前缀, 122
- pretty print 精巧的输出, 104
- print statement print 语句, 5, 7
- print statement print 语句, 155
- print 语句, 186
- problem recognition 问题识别, 79, 81
- profile module profile 模块, 124
- program testing 程序测试, 81
- program 程序, 7
- Project Gutenberg, 117
- prompt 提示, 43
- prompt 提示符, 7
- prose 散文, 5
- pseudorandom 伪随机, 117
- pseudorandom 伪随机, 125
- Puzzler 难题, 82
- Puzzler 难题, 82, 105
- Python 3.0, 5, 12, 43, 103, 110
- Python Imaging Library(PIL), 181
- python.org, 7
- Python 调试器 (pdb) , 186
- quotation mark 引号, 5
- radd method radd 方法, 156
- radian 弧度, 18
- raise statement 引发异常, 98
- raise 语句, 149
- Ramanujan, Srinivasa, 65
- randint function randint 函数, 118
- randint 函数, 94
- random function random 函数, 118
- random module random 模块, 118
- random number 随机数字, 117
- random text , 122
- random walk programming 瞎猫碰死耗子编程, 125
- random 函数, 113
- random 模块, 94, 113, 165
- raw_input function raw_input 函数, 43
- readline 方法, 132
- read 方法, 132
- Rectangle class Rectangle 类, 139
- recursion
 - base case, 42
 - infinite, 42
- recursion 递归, 40, 41, 45
- redline method readline 方法, 77
- reducible world 可化简单词, 105
- redundancy 冗余性, 5
- relational operator 关系运算符, 38
- reload 函数, 134, 184
- remove 方法, 88
- replace method replace 方法, 117
- representation 代表, 137, 139
- repr 函数, 134

- return statement return 语句, 41
- return value 返回值, 17, 26, 140
- return 语句, 188
- reverse lookup ,dictionary 颠倒查询, 字典, 104
- reverse lookup,dictionary 颠倒查询, 字典, 98
- reversed 函数, 114
- rotation
 - letters, 105
- RSA algorithm RSA 算法, 103
- running pace 跑步速度, 8
- runtime error 运行时错误, 3, 42, 44
- RuntimeError, 42
- safe language 安全语言, 3
- sanity check 健康测试, 104
- scaffolding 脚手架, 104
- script mode 脚本模式, 7, 24
- script 脚本, 7
- search, 98
- search pattern 搜索模式, 78
- self (parameter name), 152
- semantics error 语义错误, 7
- semantics errors 语义错误, 3
- semantics 语义, 3, 7, 151
- sequence
 - coordinate, 174
- set membership, 96
- set 集合, 121
- setdefault method setdefault 方法, 100
- shallow copy, 143
- shallow copy 浅拷贝, 142
- shell, 132
- shelve 模块, 132, 135
- shuffle 函数, 165
- SimpleTurtleWorld class, 176
- sine function sin 函数, 18
- singleton , 99
- singleton 独子体, 104
- slice
 - update, 86
- sorted 函数, 114
- sort 方法, 86, 92, 112, 165
- specail value
 - True, 38
- special case 特殊情况, 81
- special case 特殊情况, 81
- special value
 - False, 38
 - None, 24
- split 方法, 89, 108
- sqrt function sqrt 函数, 18
- square root, 62
- squiggly bracket 大括号, 95
- stack diagram 堆栈图, 26, 42
- stack diagram 堆栈示意图, 23
- stack diagrams 堆栈示意图, 23
- state diagram, 59
- state diagram 状态图, 100, 138, 139
- state diagrm 状态图, 141
- statement
 - assignment, 59
 - break, 61
 - conditional, 38, 45
 - global 全局, 102
 - if, 38
 - import, 26
 - pass, 39
 - print, 155
 - print 语句
 - 打印, 5, 7
 - raise, 98
 - return, 41
 - while, 60
- statment
 - compound, 39
- str function str 函数, 18
- __str__方法, 154
- __str__ 方法, 164
- string module string 模块, 117
- string representation, 154
- strip method strip 方法, 77, 117
- structshape 模块, 114
- structure 结构, 4
- subject 主体, 152, 158, 176
- subtraction
 - dictionary, 121
 - with borrowing, 63
- suffix 后缀, 122
- sum 函数, 110
- SVG, 182
- Swampy, 29, 77, 143, 171
- syntax, 183
- syntax error, 7
- syntax 语法, 3, 7, 151
- SyntaxError 语法错误, 19
- testing
 - and absence of bugs, 81

- is hard, 81
- text
 - plain, 77, 182
 - plain 完全的, 117
 - random, 122
- Text widget 文本空间, 174
- Time 类, 145
- Tkinter, 171
- token 标记, 4, 7
- traceback, 26
- traceback 回溯, 99
- traceback 跟踪, 42, 43
- traceback 追踪, 24
- translate method translate 方法, 117
- traversal, 119
 - dictionary, 158
- traversal 遍历, 79, 97
- triangle 三角形, 45
- trigonometric function 三角函数, 18
- True special value True 特殊值, 38
- try 语句, 130
- tuple
 - as key in dictionary, 123
- tuple 函数, 107
- Turing, Alan, 51
- TurtleWorld, 29, 46, 170
- type
 - bool, 38
 - dict, 95
 - long 长, 103
 - set 集合, 121
 - user-defined, 137
- type conversion 类型转换, 17
- type function type 函数, 142
- type-based dispatch 基于类型的调度, 157, 158
- type-based dispatchch 基于类型的调度, 156
- TypeError, 100, 153
- typographical error 印刷错误, 124
- UML, 167
- UnboundLocalError, 102
- Unix 命令
 - gunzip, 133
 - ls, 132
- update
 - coordinate, 179
 - global variable 全局变量, 102
 - histogram, 119
 - slice, 86
- update 更新, 60, 62, 64
- update 方法, 111
- URL, 135, 182
- urllib module, 182
- urllib 模块, 135
- use before def 定义前使用, 21
- user-defined type 自定义类型, 137
- value
 - default 缺省, 120
- value 关键字值, 104
- ValueError, 98
- ValueError, 43
- values method values 方法, 96
- variabl
 - global 全局, 101
- variable
 - local 变量
 - 局域, 22
 - updating, 60
- vector graphics 矢量图, 182
- Visual module, 159
- void function void 函数, 24
- void function 虚无函数, 26
- vpython module, 159
- while loop while 循环, 60
- whitespace 空格, 44, 78
- whitespace 空白, 25
- widget, 181
 - Button, 172
 - Canvas, 173
 - Entry, 174
 - Frame, 176
 - Label, 172
 - Menubutton, 177
 - Text, 174
- widget 控件, 171
- widget, packing, 176
- word frequency 单词频数, 125
- word frequency 单词频率, 117
- word, reducible 单词, 可化简, 105
- World module World 模块, 143
- worst bug
 - ever, 182
- worst bug 最糟糕的 bug, 158
- Zipf's law, 125
- zip 函数, 110
 - 和 dict 一同使用, 111

- 三重引用字符串, 35
- 下划线字符, 11
- 下取整除法, 12, 16
- 下标, 67, 73, 74, 83, 93, 186
 - 从 0 开始, 84
 - 从零开始, 67
 - 切片, 69, 85
 - 循环, 85
 - 负数, 68
- 下标错误, 68, 73, 84, 186
- 不可变, 70, 74
- 不可改变, 91, 113
- 不可改变的, 107
- 两分法搜索, 94
- 临时变量, 47, 55, 188
- 乌龟打字机, 36
- 二进制搜索, 94
- 交互模式, 2, 12
- 交换性, 14
- 交换模式, 108
- 优先级, 16, 188
- 优先级规则, 13, 16
- 位操作, 12
- 低级语言, 1
- 信心的飞跃, 53
- 修改函数, 147, 149

- 借位减法, 148
- 借位, 减法, 148
- 值, 89, 90
 - 元组, 109
- 值错误, 108
- 元素, 83, 93
- 元素删除, 88
- 元组, 107, 109, 113, 115
 - 作为字典中的键, 112
 - 切片, 108
 - 单一, 107
 - 在括号中, 112
 - 比较, 112, 164
 - 赋值, 108
- 元组赋值, 109, 110, 115
- 先决条件, 35, 55, 94, 168
- 八进制, 11
- 六十进制, 147
- 关系运算符, 163
- 关键字, 11, 15, 183
- 关键字参数, 32, 35, 113
- 冒号, 183
- 函数
 - ack, 56
 - cmp, 164
 - enumerate, 111
 - exists, 129
 - find, 70
 - getcwd, 129
 - isinstance, 54
 - len, 67
 - list, 88
 - max, 109, 110
 - min, 109, 110
 - open, 127, 130, 131
 - popen, 132
 - randint, 94
 - random, 113
 - reload, 134, 184
 - repr, 134
 - reversed, 114
 - shuffle, 165
 - sorted, 114
 - sum, 110
 - tuple, 107
 - zip, 110
 - 圆, 31
 - 多边形, 31
 - 开根, 49
 - 弧, 31
 - 斐波纳契, 53
 - 比较, 48
 - 绝对值, 48
 - 阶乘, 51
- 函数复合, 50
- 函数式编程风格, 147
- 函数时编程风格, 149
- 函数类型
 - 修改, 147
 - 纯, 146
- 函数, 元组作为返回值, 109
- 分割服, 89
- 分割符, 93
- 切片, 74
 - 元组, 108
 - 列表, 85
 - 复制, 69, 86
 - 字符串, 69
- 切片运算符, 69, 75, 85, 92, 108
- 列表, 83, 88, 93, 113
 - 下标, 84
 - 作为参数, 91
 - 元素, 83

- 元组的, 110
- 切片, 85
- 复制, 86
- 对象的, 164
- 嵌套, 83, 85
- 成员, 84
- 操作, 85
- 方法, 86
- 理解, 88
- 空, 83
- 连接, 85, 91, 94
- 遍历, 85, 93
- 重复, 85
- 创建, 19
- 删除, 列表中的元素, 88
- 别名, 89, 90, 93
 - 复制以避免, 93
- 加密, 161
- 包含关系, 167, 169
- 单一, 107
- 单词统计, 133
- 单词, 可缩小的, 116
- 压缩
 - 文件, 133
- 原型和补丁, 145, 147, 149
- 参数, 91
 - 关键字, 32, 35, 113
 - 列表, 91
 - 变长元组, 109
 - 可选, 71, 89
 - 聚集, 109
- 参数散布, 109
- 反转词对, 94
- 变量, 9, 10, 15
 - 临时, 47, 55, 188
- 变长参数元组, 109
- 可变, 70
- 可执行代码, 2
- 可改变, 86, 90, 113
- 可改变的, 84, 107
- 可缩小的单词, 116
- 可选参数, 71, 89
- 名字错误, 186
- 后决条件, 35, 55, 168
- 和校验, 135
- 哈希表, 112
- 唯一, 94
- 回文, 56, 75, 94
- 回文集合, 115, 132
- 回溯, 186
- 图
 - 对象, 145, 163
 - 栈, 91
 - 状态, 74, 84, 90, 112, 145, 163
 - 类, 167, 169
- 图灵完备语言, 51
- 图灵理论, 51
- 图表
 - 状态, 10
- 圆函数, 31
- 增量开发, 55, 183
- 增量赋值, 87, 93
- 复制
 - 切片, 69, 86
- 复合, 50
- 多态, 168
- 多态 (在类图中), 168, 169
- 多行, 184
- 多行字符串, 35
- 多边形函数, 31
- 大小写敏感, 变量名, 15
- 大而复杂的表达式, 187
- 头部, 183
- 子类, 166, 169
- 字典, 111, 186
 - 初始化, 111
 - 遍历, 111
- 字典方法
 - anydbm 模块, 131
- 字母, 36
- 字母表, 68
- 字母频率, 115
- 字符, 67
- 字符串, 9, 15, 88, 113
 - 三重引用, 35
 - 不可变, 70
 - 切片, 69
 - 多行, 35
 - 操作, 13
 - 方法, 71
 - 比较, 72
 - 汇聚, 164
 - 空, 89
- 字符串方法, 75
- 字符串类型, 9
- 字符串表示, 134
- 字符创
 - 多行, 184
- 字符旋转, 76
- 守护人模式, 54, 55, 73

- 定义
 - 循环, 51
 - 递归, 116
- 实例, 29, 35
- 实例属性, 162, 169
- 对象, 70, 74, 89, 90, 93
 - 模块, 133
- 对象图, 145, 163
- 导入指令, 29
- 封装, 31, 35, 50, 71, 166
- 属于关系, 167, 169
- 属性
 - 实例, 162, 169
 - 类, 162, 169
- 属性错误, 186
- 嵌套列表, 83, 85, 93
- 工作目录, 129
- 布尔函数, 50, 145
- 布尔运算符, 72
- 带情绪的调试, 188
- 帧, 52
- 序列, 67, 74, 83, 88, 107, 113
- 开发方案
 - 原型和补丁, 145
- 开发计划, 35
 - 原型和补丁, 147
 - 增量, 48, 183
 - 封装和泛化, 34
 - 有计划的, 147
 - 随机漫步编程, 188
- 开根, 49
- 异常, 15, 183, 186
 - 下标错误, 68, 73, 84, 186
 - 值错误, 108
 - 名字错误, 186
 - 属性错误, 186
 - 类型错误, 67, 70, 108, 109, 128, 186
 - 输入输出错误, 130
 - 键错误, 186
- 异常, 捕获, 130
- 引号标记, 9, 35, 69, 184
- 引用, 90, 91, 93
 - 别名, 90
- 弧函数, 31
- 强壮型检查, 148
- 归并模式, 87, 93
- 形状, 115
- 形状错误, 114
- 循环, 30, 35, 110
 - for, 30, 68, 85
 - 下标, 85
 - 关于字符串, 70
 - 嵌套, 164
 - 无限, 185
 - 条件, 185
 - 遍历, 68
- 循环和记数, 70
- 循环定义, 51
- 心中的模型, 187
- 愤怒, 188
- 成员
 - 两分法搜索, 94
 - 二进制搜索, 94
 - 列表, 84
- 扑克, 161, 169
- 打字机, 乌龟, 36
- 执行流程, 53, 55, 168, 186
- 括号
 - 元组位于, 107
 - 压倒优先级, 13
 - 父类在, 166
 - 空, 71
- 括号运算符, 67, 83, 108
- 拷贝, 135
- 拼字游戏, 115
- 持久性, 127, 134
- 挂起, 185
- 捕获, 134
- 接口, 33, 35, 168
- 提示符, 2
- 搜索模式, 70
- 搜索, 两分法, 94
- 搜索, 二进制, 94
- 散布, 109, 115
- 数值, 15
- 数学运算符, 12
- 数据库, 131, 134, 135
- 数据类型, 9
 - 字符串, 9
 - 整数, 9
 - 浮点数类型, 9
- 数据结构, 114, 115
- 整数, 15
- 整数类型, 9
- 文件, 127
 - 压缩, 133
 - 权限, 130
 - 读取和写入, 127
- 文件名, 129

- 文件夹, 129
- 文本
 - 纯, 135
- 文本文件, 134
- 文档字符串, 34, 35
- 斐波纳契函数, 53
- 斜边, 49
- 新行, 164
- 方法, 71, 74
 - __cmp__, 163
 - __str__, 164
 - append, 86, 91, 164, 165
 - close, 128, 131, 132
 - extend, 86
 - init, 162, 164, 166
 - items, 111
 - join, 89, 164
 - mro, 168
 - pop, 88, 165
 - read, 132
 - readline, 132
 - remove, 88
 - sort, 86, 92, 112, 165
 - split, 89, 108
 - update, 111
 - 字符串, 75
 - 空, 86
 - 记数, 72
- 方法 append, 94
- 方法解析顺序, 168
- 方法, 列表, 86
- 旋转, 字符, 76
- 无穷递归, 54
- 无限循环, 185
- 无限递归, 185
- 映射, 84, 93, 161
- 映射模式, 87, 93
- 普遍化, 148
- 更新
 - 数据库, 131
 - 项目, 85
- 更新运算符, 87
- 最大公约数 (GCD), 57
- 有计划的开发, 147, 149
- 未定义而使用, 15
- 权限, 文件, 130
- 条件, 184, 185
- 条件语句, 51
- 查找模式, 74
- 栈图, 36, 52, 56, 91
- 格式字符串, 128, 134
- 格式序列, 128, 134
- 格式运算符, 128, 134, 186
- 模块
 - anydbm, 131
 - bisect, 94
 - datetime, 150
 - os, 129
 - pickle, 127, 131
 - random, 94, 113, 165
 - reload, 184
 - shelve, 132, 135
 - structshape, 114
 - urllib, 135
 - 重载, 134
- 模块对象, 133
- 模块, 编写, 133
- 模型, 心中的, 187
- 模式
 - DSU, 113
 - 交换, 108
 - 守护人, 55, 73
 - 归并, 87, 93
 - 搜索, 70
 - 映射, 87, 93
 - 查找, 74
 - 监护人, 54
 - 筛选, 87, 93
 - 装饰 -排序 -还原, 113
- 欧几里得算法, 57
- 步长, 75
- 死区代码, 48, 55, 186
- 比较
 - 元组, 112, 164
 - 字符串, 72
- 比较函数, 48
- 毕达哥拉斯理论, 48
- 汇聚
 - 字符串, 164
- 沮丧, 188
- 沼泽, 170
- 泛化, 32
- 注释, 14, 16
- 测试
 - 交互模式, 2
 - 信心的飞跃, 53
 - 增量开发, 48
 - 最小测试案例, 187
 - 直到答案, 49
- 测试案例, 最小, 187

- 浮点数, 15
- 浮点数类型, 9
- 浮点除法, 12
- 源码, 2
- 点符号, 71, 162
- 父类, 166, 169
- 特殊值
 - None, 48, 55, 86, 88
- 特殊值 None, 86, 88
- 特殊情况, 147
- 状态图, 15, 74, 84, 90, 112, 145, 163
- 状态图表, 10
- 玩扑克, Anglo-American, 161
- 生日, 150
- 生日悖论, 94
- 用户定义类型, 145
- 目录, 129, 134
 - 工作, 129
 - 遍历, 129
- 目标, 2
- 相同, 90, 93
- 相对路径, 129, 134
- 相等, 90, 93
- 相等与赋值, 59
- 秘密练习, 135
- 移植性, 1
- 程序, 2
- 程序语句, 12
- 空列表, 83
- 空字符串, 74, 89
- 空方法, 86
- 空白, 134
- 空白符, 184
- 等级, 161
- 筛选模式, 87
- 筛选模式 i, 93
- 算法, 3
 - MD5, 135
 - 欧几里得, 57
- 管道, 132, 135
- 类
 - Card, 162
 - Date, 150
 - Hand, 166
 - Time, 145
 - 子, 166, 169
 - 父, 166
 - 纸牌, 164
- 类图, 167, 169
- 类型, 15
 - 元组, 107
 - 列表, 83
 - 文件, 127
 - 用户定义, 145
- 类型数据, 9
- 类型检查, 53
- 类型错误, 67, 70, 108, 109, 128, 186
- 类属性, 162, 169
- 累加器, 93
 - 列表, 87
 - 和, 87
- 累积求和, 88
- 约束, 148, 149
- 级联, 14, 16
- 纯函数, 146, 149
- 纯文本, 135
- 纸牌类, 164
- 纸牌, 玩, 161
- 纸牌, 玩卡片, 164
- 练习, 秘密, 135
- 组成, 164
- 绝对值函数, 48
- 绝对路径, 129, 134
- 继成, 166
- 继承, 169
- 编码, 161, 169
- 编程语言, 1
- 缩进, 184
- 网络电影数据库 (IMDb) , 135
- 置换, 115
- 聚集, 109, 115
- 脚手架, 55
- 脚手架代码, 49
- 脚本, 2
- 脚本模式, 2, 12
- 臭虫, 3
- 花, 36
- 花色, 161
- 行结束符号, 134
- 表示, 161
- 表达式, 12, 13, 16
 - 大而复杂, 187
- 装饰 -排序 -还原模式, 113
- 解决问题, 1
- 解析, 135
- 计数器, 74
- 记数, 70
- 记数和循环, 70

- 记数方法, 72
- 访问, 83
- 语义错误, 10, 15, 74, 183, 187
- 语句, 15
 - assert, 149
 - for, 30, 68, 85
 - import, 133
 - print, 186
 - raise, 149
 - return, 188
 - try, 130
 - 导入, 29
 - 条件, 51
 - 赋值语句, 10
 - 返回, 47
- 语法错误, 3, 15, 183
- 语言
 - 低级, 1
 - 图灵完备, 51
 - 编程, 1
 - 高级, 1
- 调用, 71, 74
- 调试, 3, 15, 35, 54, 73, 92, 114, 134, 148, 168, 183
 - 情绪反应, 188
 - 迷信, 188
- 调试器 (pdb), 186
- 负数下标, 68
- 贡献者, vii
- 赋值, 13, 15, 83, 94
 - 以避免别名, 93
 - 元组, 108--110, 115
 - 增量, 87, 93
 - 项目, 70, 84, 108
- 赋值语句, 10
- 超类, 166
- 跑步比赛, 16
- 跑步速度, 149
- 跟踪者, 170
- 路径, 129, 134
 - 相对, 129
 - 绝对, 129
- 输入输出错误, 130
- 运算符
 - 关系, 163
- 运算数, 12, 15
- 运算符, 15
 - del, 88
 - in, 72, 84
 - is, 89
 - 切片, 69, 75, 85, 92, 108
 - 字符串, 13
 - 布尔, 72
 - 括号, 67, 83, 108
 - 按位, 12
 - 更新, 87
 - 格式, 128, 134, 186
- 运算符优先级, 15, 188
- 运算符的优先级, 13
- 运算符重载, 163
- 运算符, 数学, 12
- 运行时错误, 15, 53, 183, 186
- 返回值, 47
 - 元组, 109
- 返回语句, 47
- 进位, 加法, 146, 148
- 连接, 68, 70, 89
 - 列表, 85, 91, 94
- 连锁词, 94
- 迷信的调试, 188
- 递归, 51, 53
 - 无穷, 54
 - 无限, 185
- 递归定义, 51, 116
- 遍历, 68, 70, 73, 74, 87, 93, 110, 111, 113
 - 列表, 85
 - 字典, 111
- 遍历, 目录, 129
- 重复, 30
 - 列表, 85
- 重构, 33, 34
- 重载, 163, 166, 168
- 错误
 - 形状, 114
 - 编译时, 183
 - 语义, 15, 74, 183, 187
 - 语义错误, 10
 - 语法, 3, 15, 183
 - 运行时, 15, 183
- 错误信息, 3, 10
- 错误检查, 53
- 错误消息, 15, 183
- 键-值对, 111
- 键错误, 186
- 阶乘函数, 51, 53
- 除法
 - 下取整, 12
 - 浮点, 12
- 随机漫步编程, 188

难题, 115
集合
 回文, 115, 132
零, 下标开始, 67
面向对象编程, 166
项目, 74, 83
项目更新, 85
项目赋值, 70, 84, 108
频率
 字母, 115
饰面, 165, 169
高级语言, 1

Index

- 0, 下标开始于, 84
- , 181
- 泛化, 35
- abecedarian, 78
- accumulator
 - histogram, 119
- Ackerman 函数, 56
- add method add 方法, 155
- addition with carrying, 63
- algorithm, 7
 - RSA, 103
 - square root, 64
- algorithm 算法, 63, 122
- aliasing 别名, 139, 141, 158
- alternative executive 选择执行, 39
- ambiguity 二义性, 5
- and operator and 运算符, 38
- anydbm 模块, 131
- append 方法, 86, 91, 94, 164, 165
- argument
 - keyword, 172
 - optional, 99
- argument 参数, 19
- argument 实参, 21, 22, 26
- argument 形式参数, 17
- assert 语句, 149
- assignment
 - multiple, 64
 - multiple 多次, 102
- attribute
 - __dict__, 157
 - initializing, 157
 - instance, 138, 143
- AttributeError, 142
- Austin, Jane, 119
- available colors, 144, 159
- Bacon, Kevin, 135
- Bangladesh, national flag 孟加拉国, 国旗, 143
- base case, 45
- base case 终止条件, 42
- benchmarking 基准程序, 124, 125
- binding 绑定, 178, 181
- bingo, 115
- bisection, debugging by, 64
- bisect 模块, 94
- body, 45
- body 体, 19, 61
- body 函数体, 25
- bool type bool 类型, 38
- boolean expression 布尔表达式, 37, 44
- borrowing, subtraction with, 63
- bouding box, 174
- bouding box 界定盒子, 181
- bound method 绑定方法, 176
- bound method 绑定方法, 181
- bounding box , 143
- bracket
 - squiggly, 95
- branch, 45
- branch 分支, 39
- break statement break 语句, 61
- bug, 7
 - worst, 158
 - worst ever, 182
- Button widget 按钮控件, 172
- calculator 计算器, 8
- calculator, 16
- call graph, 101
- call graph 调用框图, 104
- callable object 可调对象, 177
- callback 回调, 172, 176--178
- callback 回调函数, 180
- Canvas coordinate 画布坐标, 173, 179
- Canvas item 画布项, 173
- Canvas object, 143
- Canvas widget 画布控件, 173
- Car Talk, 82, 105, 115
- Car Talk 汽车谈话, 82

- Card 类, 162
- carrying, addition with, 63
- chained conditional 链式条件, 39
- chained conditional 链式条件语句, 45
- choice function choice 函数, 118
- class, 143
 - Kangaroo, 158
 - Point, 137, 154
 - Rectangle, 139
 - SimpleTurtleWorld, 176
- class definition 类定义, 137
- class object 类对象, 137, 143
- class 类, 137
- close 方法, 128, 131, 132
- __cmp__ 方法, 163
- cmp 函数, 164
- Collatz conjecture Collatz 猜想, 61
- colon 分号, 19
- color list, 144, 159
- commutativity 可交换, 156
- compile 编译, 6
- comple, 1
- composition 创建, 22
- compositon 创建, 26
- compound statement, 45
- compound statment, 39
- concatenation 连接, 22
- condition 条件, 39, 45, 61
- conditional
 - chained, 39, 45
 - nested, 40, 45
- conditional execution 条件执行, 38
- conditional statement 条件语句, 38, 45
- config method, 173
- consistency check 连贯性测试, 104
- conversion
 - type 转换
 - 类型, 17
- coordinate
 - Canvas, 173
 - pixel, 179
- coordinate sequence 坐标序列, 174
- coordinate! Canvas, 179
- copy
 - deep, 142
 - shallow, 142
- copy module copy 模块, 141
- copying objects 复制对象, 141
- counter 计数器, 96, 102
- crosswords 纵横组合字谜, 77
- cummings, e. e. 康明思, 3
- Czech Republic, national flag 捷克共和国, 国旗, 144
- data structure 数据结构, 123
- datetime 模块, 150
- Date 类, 150
- debugging
 - by bisection, 64
 - emotional response 调试
 - 情绪反应, 6
 - experimental 调试
 - 实验, 3
- debugging 调试, 6, 7, 25, 43, 81, 103, 124, 142, 157, 180
- declaration 声明, 102, 104
- decrement, 64
- decrement 减量, 60
- deep copy 深拷贝, 142, 143
- deepcopy function deepcopy 函数, 142
- def keyword def 关键字, 19
- default value
 - avouding mutable, 158
- default value 缺省值, 120, 125, 154
- definition
 - class, 137
 - function 定义
 - 函数, 19
- del 运算符, 88
- deterministic, 117
- deterministic 确定的, 125
- development plan
 - problem recognition, 79, 81
 - random walk programming, 125
- diagram
 - call graph 调用框图, 104
 - object, 138, 139, 141, 143
 - stack, 23
 - state, 59, 100, 138, 139, 141
- __dict__ attribute, 157
- dict function dict 函数, 95
- dictionary
 - invert, 99
 - lookup, 98
 - looping with, 97
 - reverse lookup, 98
 - subtraction, 121
 - traversal, 158
- dictionary 字典, 95, 104
- Dijkstra, Edsger, 81

- dispatch
 - type-based, 157
- dispatch ,type-based, 156
- divisibility 整除, 37
- division
 - floor, 44
- divmod, 109, 148
- docstring 文档字符串, 137
- documentation 文档, 7
- dot notation 点记法, 18, 26, 138, 152
- double letters , 82
- Doyle, Arthur Conan, 4
- drag-and-drop 拖放, 179
- DSU pattern, 119
- DSU 模式, 113, 115
- duplicate, 104
- Einstein, Albert, 33
- elif keyword elif 关键字, 39
- ellipses 省略号, 20
- else keyword else 关键字, 39
- email 地址, 108
- embedded object 嵌套对象, 158
- embeded object, 140
 - copying, 141
- embededed object 嵌套对象, 143
- emotional debugging 情绪调试, 6
- encapsulation 封装, 63
- encryption 加密, 103
- Entry widget 输入框, 174
- enumerate 函数, 111
- epsilon , 63
- error
 - runtime, 42, 44
 - runtime 错误
 - 运行时, 3
 - semantic 错误
 - 语义, 3
- error message 错误信息, 3, 6
- eval function eval 函数, 65
- event, 181
- event handler 事件处理, 178
- event loop 事件循环, 171, 181
- Event object, 178
- event string 事件字符串, 178
- event-driven programming, 181
- event-driven programming 事件驱动编程, 180
- event-driven programming 时间驱动编程, 172
- exception
 - AttributeError, 142
 - KeyError, 96
 - NameError, 23
 - OverflowError, 44
 - RuntimeError, 42
 - SyntaxError, 19
 - TypeError, 100, 153
 - UnboundLocalError, 102
 - ValueError, 43, 98
- exception 异常, 3, 7
- exists 函数, 129
- experimental debugging 实验性的调试, 3
- experimental debugging 实验性调试, 124
- expression
 - boolean, 37, 44
- extend 方法, 86
- False special value False 特殊值, 38
- Fermat's Last Theorem 费马最后定理, 45
- fibonacci function fibonacci 函数, 100
- file object 文件对象, 77
- file object 文件对象, 81
- find 函数, 70
- flag 标记, 101, 104
- float function float 函数, 17
- floating-point 浮点, 63
- floor division 地板除, 44
- flow of execution 执行流, 26, 60, 180
- flow of execution 执行流 j, 21
- for 循环, 30
- formal language 正式语言, 4, 7
- for 循环, 68, 85, 111
- frabjous, 51
- frame, 101
- Frame widget, 176
- frame 图, 42
- frame 框图, 26
- frame 框架, 23
- Free Document License, GNU, vi
- Free Documentation License, GNU, v
- frequency
 - word, 125
 - word 单词, 117
- frequency 频率, 97
- fruitful fuction 结果函数, 26
- fruitful function 卓有成效的函数, 24
- fucntion
 - choice, 118
- fucntion syntax 函数语法, 152

function

- deepcopy, 142
- dict, 95
- eval, 65
- fibonacci, 100
- float 函数
 - float, 17
- getattr, 158
- hasattr, 157
- int 函数
 - int, 17
- isinstance, 156
- len, 26, 96
- log 函数
 - log, 18
- open, 77, 78
- randint, 118
- random, 118
- raw_input, 43
- recursive, 41
- sqrt 函数
 - sqrt, 18
- str 函数
 - str, 18
- type, 142

function argument 函数实参, 21

function call 函数调用, 17, 26

function definition 函数定义, 19, 20, 25

function frame 函数图, 42

function frame 函数框, 23

function frame 函数框图, 26, 101

function object 函数对象, 20, 25, 26

function parameter 函数形参, 21

function 函数, 19, 25, 151

function, fruitful, 24

function, math 函数, 数学, 18

function, reason for, 25

function, trigonometric 函数, 三角, 18

function, void, 24

gamma 函数, 54

GCD (最大公约数), 57

generalization, 79

geometry manager, 181

geometry manager 几何管理器, 177

get method get 方法, 97

getattr function getattr 函数, 158

getcwd 函数, 129

global statement 全局语句, 102

global variable

- update 更新, 102
- global variable 全局变量, 101, 104
- GNU Free Document License, vi
- GNU Free Documentation License, v
- graphical user interface 图形用户接口, 171
- grid 网格, 27
- GUI, 171
- Gui module, 171
- gunzip (Unix 命令), 133

Hand 类, 166

hasattr function hasattr 函数, 157

hash function 散列函数, 100, 104

hashable 可散列的, 100

hashable 散列体, 104

hashtable 散列, 96

hashtable 散列表, 104

header 函数头, 25

header 头, 19

Hello, World, 5

help utility 帮助工具, 8

hexadecimal 十六进制, 138

histogram

- random choice, 118, 121
- word frequencies, 118

histogram 直方图, 97

Holems, Sherlock, 4

homophone 同音词, 105

HTMLParser module, 182

hyperlink 超连接, 182

if statement if 语句, 38

Image module, 181

image viewer 图像查看器, 181

IMDb (网络电影数据库), 135

immutability 不可变性, 100

implementation 实现, 97, 104, 123

import statement import 语句, 26

import 语句, 133

in operator in 运算符, 96

increment, 64, 147

increment 增量, 60

increment 增量, 153

indentation 缩进, 19, 152

index

- looping with, 80

index 索引, 95

infinite loop 无限循环, 61, 64, 171

infinite recursion 无穷递归, 42, 45

init method init 方法, 157

- init method 初始方法, 154
- initialization
 - variable, 64
- initialization (before update) (更新前) 初始化, 60
- init 方法, 162, 164, 166
- instance
 - as argument, 138
 - as return value, 140
- instance attribute 实例属性, 138, 143
- instance 实例, 138, 143
- instantiation 实例化, 138
- int function int 函数, 17
- integer
 - long 长, 103
- interactive mode 交互模式, 7, 24
- interpret, 1
- invariant 不变量, 180
- invert dictionary 反转字典, 99
- in 运算符, 72, 84
- is operator is 运算符, 141
- isinstance function isinstance 函数, 156
- isinstance 函数, 54
- is 运算符, 89
- item
 - Canvas, 173, 181
 - dictionary 字典, 104
- items 方法, 111
- iteration 迭代, 60, 64
- iteration 迭代器, 59
- join 方法, 89, 164
- Kangaroo class Kangaroo 类, 158
- Kevin Bacon Game, 135
- key 关键字, 95
- key 键, 104
- key-value pair 关键字—值对, 95
- key-value pair 键值对, 104
- keyboard input 键盘输入, 43
- KeyError, 96
- keys method keys 方法, 98
- keyword
 - def 关键字
 - def, 19
 - elif, 39
 - else, 39
- keyword argument, 181
- keyword argument 关键参数, 172
- koch curve 柯霍曲线, 46
- Label widget, 172
- language
 - formal 语言
 - 正式, 4
 - natural 语言
 - 自然, 4
 - safe 语言
 - 安全, 3
- len function len 函数, 96
- len 函数, 67
- letter rotation , 105
- Linux, 4
- lipogram 避讳, 78
- list
 - 函数, 88
- literalness 无修饰性, 5
- local variable 局域变量, 22, 26
- log function log 函数, 18
- logarithm 对数, 125
- logical operator 逻辑运算符, 37, 38
- long integer 长整数, 103
- lookup 查询, 104
- lookup,dictionary 查询, 字典, 98
- loop
 - event, 171
 - infinite, 61, 171
 - while, 60
- loop 循环, 61
- looping
 - with dictionaries, 97
 - with indices, 80
- ls (Unix 命令), 132
- mapping 映射, 122
- Markov analysis 马尔可夫分析, 122
- mash-up 混合, 123
- math function 数学函数, 18
- max 函数, 109, 110
- McCloskey, Robert, 68
- MD5 算法, 135
- membership
 - dictionary, 96
 - set, 96
- memo 备忘录, 101, 104
- Menubutton widget 菜单按钮, 177
- metaphor, method invocation 隐喻, 方法调用, 152
- method
 - __str__, 154
 - add, 155

- config, 173
- get, 97
- init, 154
- keys, 98
- radd, 156
- setdefault, 100
- strip, 77, 117
- translate, 117
- values, 96
- method syntax 方法语法, 152
- method 方法, 151, 158
- method, bound, 176
- methods
 - readline, 77
- min 函数, 109, 110
- module
 - copy, 141
- module
 - Gui, 171
 - HTMLParser, 182
 - Image, 181
 - pprint, 104
 - profile, 124
 - random, 118
 - string, 117
 - urllib, 182
 - Visual, 159
 - vpython, 159
 - World, 143
- module object 模块对象, 18
- module 模块, 18, 26
- modulus operator 模操作符, 37
- modulus operator 模运算符, 44
- Monty Python and the Holy Grail, 146
- MP3, 135
- mro 方法, 168
- mutability 可变, 102, 140
- mutable object, as default value, 158
- mutiple assignement 多重赋值, 64
- mutiple assignment 多重赋值, 59, 102
- NameError, 23
- natural language 自然语言, 4, 7
- nested conditional 嵌套条件语句, 45
- nested conditional 嵌套的条件语句, 40
- newline 换行, 43, 59
- Newton's method 牛顿方法, 62
- None special value None 特殊值, 24
- None 特殊值, 48, 55
- not operator not 运算符, 38
- number 数字, random 随机, 117
- object
 - Callable, 177
 - Canvas, 143
 - class, 137
 - copying, 141
 - embedded, 143, 158
 - embeded, 140
 - Event, 178
 - file, 77, 81
 - function, 26
 - function 对象
函数, 20
 - mutable, 140
 - printing, 152
- object code 目标代码, 7
- object diagram 对象图, 138, 139, 141, 143
- object 对象, 137
- object-oriented language 面向对象语言, 158
- object-oriented programming 面向对象编程, 151, 158
- odometer 里程表, 82
- oeprn function open 函数, 77
- open function open 函数, 78
- open 函数, 127, 131
- open 海曙, 130
- operator
 - and, 38
 - in, 96
 - is, 141
 - logical, 37, 38
 - modulus, 44
 - modulus 操作符
模, 37
 - not, 38
 - or, 38
 - overloading, 158
 - relational, 38
- operator overloading 运算符重载, 155
- option, 172
- option 选项, 181
- optional argument, 99
- optional parameter 可变参数, 120
- optional parameter 可选择参数, 154
- or operator or 运算符, 38
- os 模块, 129
- other (parameter name), 153
- OverflowError, 44
- overloading 重载, 158

- override 覆盖, 120, 125, 154
- packing widgets , 181
- packing widgets 包装控件, 176
- palindrome 回文, 80, 82
- parameter
 - optional, 154
 - optional 可选的, 120
 - self, 152
- parameter 形参, 21, 23, 25
- Parentheses
 - parameters in, 22
- parentheses
 - empty 小括号空, 19
 - matching 括号匹配, 3
 - parameters in, 21
- parenthess
 - argument in, 17
- parse 句法分析, 4, 7
- pass statement, 39
- patameter
 - other, 153
- pattern
 - DSU, 119
 - search, 78, 98
- pdb (Python 调试器) , 186
- PEMDAS, 13
- pi, 18, 65
- pickle 模块, 127, 131
- pickling, 131
- pie, 36
- PIL (Python Imaging Library), 181
- pixel coordinate 像素坐标, 179
- plain text 纯文本, 182
- plain text 纯文本文件, 77, 117
- poetry 诗歌, 5
- point ,mathematical 点、数学, 137
- Point class Point 类, 137, 154
- polymorphism 多态, 157, 158
- popen 函数, 132
- pop 方法, 88, 165
- pprint module pprint 模块, 104
- prefix 前缀, 122
- pretty print 精巧的输出, 104
- print statement print 语句, 5, 7
- print statement print 语句, 155
- print 语句, 186
- problem recognition 问题识别, 79, 81
- profile module profile 模块, 124
- program testing 程序测试, 81
- program 程序, 7
- Project Gutenberg, 117
- prompt 提示, 43
- prompt 提示符, 7
- prose 散文, 5
- pseudorandom 伪随机, 117
- pseudorandom 伪随机, 125
- Puzzler 难题, 82
- Puzzler 难题, 82, 105
- Python 3.0, 5, 12, 43, 103, 110
- Python Imaging Library(PIL), 181
- python.org, 7
- Python 调试器 (pdb) , 186
- quotation mark 引号, 5
- radd method radd 方法, 156
- radian 弧度, 18
- raise statement 引发异常, 98
- raise 语句, 149
- Ramanujan, Srinivasa, 65
- randint function randint 函数, 118
- randint 函数, 94
- random function random 函数, 118
- random module random 模块, 118
- random number 随机数字, 117
- random text , 122
- random walk programming 瞎猫碰死耗子编程, 125
- random 函数, 113
- random 模块, 94, 113, 165
- raw_input function raw_input 函数, 43
- readline 方法, 132
- read 方法, 132
- Rectangle class Rectangle 类, 139
- recursion
 - base case, 42
 - infinite, 42
- recursion 递归, 40, 41, 45
- redline method readline 方法, 77
- reducible world 可化简单词, 105
- redundancy 冗余性, 5
- relational operator 关系运算符, 38
- reload 函数, 134, 184
- remove 方法, 88
- replace method replace 方法, 117
- representation 代表, 137, 139
- repr 函数, 134

- return statement return 语句, 41
- return value 返回值, 17, 26, 140
- return 语句, 188
- reverse lookup ,dictionary 颠倒查询, 字典, 104
- reverse lookup,dictionary 颠倒查询, 字典, 98
- reversed 函数, 114
- rotation
 - letters, 105
- RSA algorithm RSA 算法, 103
- running pace 跑步速度, 8
- runtime error 运行时错误, 3, 42, 44
- RuntimeError, 42
- safe language 安全语言, 3
- sanity check 健康测试, 104
- scaffolding 脚手架, 104
- script mode 脚本模式, 7, 24
- script 脚本, 7
- search, 98
- search pattern 搜索模式, 78
- self (parameter name), 152
- semantics error 语义错误, 7
- semantics errors 语义错误, 3
- semantics 语义, 3, 7, 151
- sequence
 - coordinate, 174
- set membership, 96
- set 集合, 121
- setdefault method setdefault 方法, 100
- shallow copy, 143
- shallow copy 浅拷贝, 142
- shell, 132
- shelve 模块, 132, 135
- shuffle 函数, 165
- SimpleTurtleWorld class, 176
- sine function sin 函数, 18
- singleton , 99
- singleton 独子体, 104
- slice
 - update, 86
- sorted 函数, 114
- sort 方法, 86, 92, 112, 165
- specail value
 - True, 38
- special case 特殊情况, 81
- special case 特殊情况, 81
- special value
 - False, 38
 - None, 24
- split 方法, 89, 108
- sqrt function sqrt 函数, 18
- square root, 62
- squiggly bracket 大括号, 95
- stack diagram 堆栈图, 26, 42
- stack diagram 堆栈示意图, 23
- stack diagrams 堆栈示意图, 23
- state diagram, 59
- state diagram 状态图, 100, 138, 139
- state diagrm 状态图, 141
- statement
 - assignment, 59
 - break, 61
 - conditional, 38, 45
 - global 全局, 102
 - if, 38
 - import, 26
 - pass, 39
 - print, 155
 - print 语句
 - 打印, 5, 7
 - raise, 98
 - return, 41
 - while, 60
- statment
 - compound, 39
- str function str 函数, 18
- __str__方法, 154
- __str__ 方法, 164
- string module string 模块, 117
- string representation, 154
- strip method strip 方法, 77, 117
- structshape 模块, 114
- structure 结构, 4
- subject 主体, 152, 158, 176
- subtraction
 - dictionary, 121
 - with borrowing, 63
- suffix 后缀, 122
- sum 函数, 110
- SVG, 182
- Swampy, 29, 77, 143, 171
- syntax, 183
- syntax error, 7
- syntax 语法, 3, 7, 151
- SyntaxError 语法错误, 19
- testing
 - and absence of bugs, 81

- is hard, 81
- text
 - plain, 77, 182
 - plain 完全的, 117
 - random, 122
- Text widget 文本空间, 174
- Time 类, 145
- Tkinter, 171
- token 标记, 4, 7
- traceback, 26
- traceback 回溯, 99
- traceback 跟踪, 42, 43
- traceback 追踪, 24
- translate method translate 方法, 117
- traversal, 119
 - dictionary, 158
- traversal 遍历, 79, 97
- triangle 三角形, 45
- trigonometric function 三角函数, 18
- True special value True 特殊值, 38
- try 语句, 130
- tuple
 - as key in dictionary, 123
- tuple 函数, 107
- Turing, Alan, 51
- TurtleWorld, 29, 46, 170
- type
 - bool, 38
 - dict, 95
 - long 长, 103
 - set 集合, 121
 - user-defined, 137
- type conversion 类型转换, 17
- type function type 函数, 142
- type-based dispatch 基于类型的调度, 157, 158
- type-based dispatchch 基于类型的调度, 156
- TypeError, 100, 153
- typographical error 印刷错误, 124
- UML, 167
- UnboundLocalError, 102
- Unix 命令
 - gunzip, 133
 - ls, 132
- update
 - coordinate, 179
 - global variable 全局变量, 102
 - histogram, 119
 - slice, 86
- update 更新, 60, 62, 64
- update 方法, 111
- URL, 135, 182
- urllib module, 182
- urllib 模块, 135
- use before def 定义前使用, 21
- user-defined type 自定义类型, 137
- value
 - default 缺省, 120
- value 关键字值, 104
- ValueError, 98
- ValueError , 43
- values method values 方法, 96
- variabl
 - global 全局, 101
- variable
 - local 变量
 - 局域, 22
 - updating, 60
- vector graphics 矢量图, 182
- Visual module, 159
- void function void 函数, 24
- void function 虚无函数, 26
- vpython module, 159
- while loop while 循环, 60
- whitespace 空格, 44, 78
- whitespace 空白, 25
- widget, 181
 - Button, 172
 - Canvas, 173
 - Entry, 174
 - Frame, 176
 - Label, 172
 - Menubutton, 177
 - Text, 174
- widget 控件, 171
- widget,packing, 176
- word frequency 单词频数, 125
- word frequency 单词频率, 117
- word, reducible 单词, 可化简, 105
- World module World 模块, 143
- worst bug
 - ever, 182
- worst bug 最糟糕的 bug, 158
- Zipf's law, 125
- zip 函数, 110
 - 和 dict 一同使用, 111

- 三重引用字符串, 35
- 下划线字符, 11
- 下取整除法, 12, 16
- 下标, 67, 73, 74, 83, 93, 186
 - 从 0 开始, 84
 - 从零开始, 67
 - 切片, 69, 85
 - 循环, 85
 - 负数, 68
- 下标错误, 68, 73, 84, 186
- 不可变, 70, 74
- 不可改变, 91, 113
- 不可改变的, 107
- 两分法搜索, 94
- 临时变量, 47, 55, 188
- 乌龟打字机, 36
- 二进制搜索, 94
- 交互模式, 2, 12
- 交换性, 14
- 交换模式, 108
- 优先级, 16, 188
- 优先级规则, 13, 16
- 位操作, 12
- 低级语言, 1
- 信心的飞跃, 53
- 修改函数, 147, 149

- 借位减法, 148
- 借位, 减法, 148
- 值, 89, 90
 - 元组, 109
- 值错误, 108
- 元素, 83, 93
- 元素删除, 88
- 元组, 107, 109, 113, 115
 - 作为字典中的键, 112
 - 切片, 108
 - 单一, 107
 - 在括号中, 112
 - 比较, 112, 164
 - 赋值, 108
- 元组赋值, 109, 110, 115
- 先决条件, 35, 55, 94, 168
- 八进制, 11
- 六十进制, 147
- 关系运算符, 163
- 关键字, 11, 15, 183
- 关键字参数, 32, 35, 113
- 冒号, 183
- 函数
 - ack, 56
 - cmp, 164
 - enumerate, 111
 - exists, 129
 - find, 70
 - getcwd, 129
 - isinstance, 54
 - len, 67
 - list, 88
 - max, 109, 110
 - min, 109, 110
 - open, 127, 130, 131
 - popen, 132
 - randint, 94
 - random, 113
 - reload, 134, 184
 - repr, 134
 - reversed, 114
 - shuffle, 165
 - sorted, 114
 - sum, 110
 - tuple, 107
 - zip, 110
 - 圆, 31
 - 多边形, 31
 - 开根, 49
 - 弧, 31
 - 斐波纳契, 53
 - 比较, 48
 - 绝对值, 48
 - 阶乘, 51
- 函数复合, 50
- 函数式编程风格, 147
- 函数时编程风格, 149
- 函数类型
 - 修改, 147
 - 纯, 146
- 函数, 元组作为返回值, 109
- 分割服, 89
- 分割符, 93
- 切片, 74
 - 元组, 108
 - 列表, 85
 - 复制, 69, 86
 - 字符串, 69
- 切片运算符, 69, 75, 85, 92, 108
- 列表, 83, 88, 93, 113
 - 下标, 84
 - 作为参数, 91
 - 元素, 83

- 元组的, 110
- 切片, 85
- 复制, 86
- 对象的, 164
- 嵌套, 83, 85
- 成员, 84
- 操作, 85
- 方法, 86
- 理解, 88
- 空, 83
- 连接, 85, 91, 94
- 遍历, 85, 93
- 重复, 85
- 创建, 19
- 删除, 列表中的元素, 88
- 别名, 89, 90, 93
 - 复制以避免, 93
- 加密, 161
- 包含关系, 167, 169
- 单一, 107
- 单词统计, 133
- 单词, 可缩小的, 116
- 压缩
 - 文件, 133
- 原型和补丁, 145, 147, 149
- 参数, 91
 - 关键字, 32, 35, 113
 - 列表, 91
 - 变长元组, 109
 - 可选, 71, 89
 - 聚集, 109
- 参数散布, 109
- 反转词对, 94
- 变量, 9, 10, 15
 - 临时, 47, 55, 188
- 变长参数元组, 109
- 可变, 70
- 可执行代码, 2
- 可改变, 86, 90, 113
- 可改变的, 84, 107
- 可缩小的单词, 116
- 可选参数, 71, 89
- 名字错误, 186
- 后决条件, 35, 55, 168
- 和校验, 135
- 哈希表, 112
- 唯一, 94
- 回文, 56, 75, 94
- 回文集合, 115, 132
- 回溯, 186
- 图
 - 对象, 145, 163
 - 栈, 91
 - 状态, 74, 84, 90, 112, 145, 163
 - 类, 167, 169
- 图灵完备语言, 51
- 图灵理论, 51
- 图表
 - 状态, 10
- 圆函数, 31
- 增量开发, 55, 183
- 增量赋值, 87, 93
- 复制
 - 切片, 69, 86
- 复合, 50
- 多态, 168
- 多态 (在类图中), 168, 169
- 多行, 184
- 多行字符串, 35
- 多边形函数, 31
- 大小写敏感, 变量名, 15
- 大而复杂的表达式, 187
- 头部, 183
- 子类, 166, 169
- 字典, 111, 186
 - 初始化, 111
 - 遍历, 111
- 字典方法
 - anydbm 模块, 131
- 字母, 36
- 字母表, 68
- 字母频率, 115
- 字符, 67
- 字符串, 9, 15, 88, 113
 - 三重引用, 35
 - 不可变, 70
 - 切片, 69
 - 多行, 35
 - 操作, 13
 - 方法, 71
 - 比较, 72
 - 汇聚, 164
 - 空, 89
- 字符串方法, 75
- 字符串类型, 9
- 字符串表示, 134
- 字符创
 - 多行, 184
- 字符旋转, 76
- 守护人模式, 54, 55, 73

- 定义
 - 循环, 51
 - 递归, 116
- 实例, 29, 35
- 实例属性, 162, 169
- 对象, 70, 74, 89, 90, 93
 - 模块, 133
- 对象图, 145, 163
- 导入指令, 29
- 封装, 31, 35, 50, 71, 166
- 属于关系, 167, 169
- 属性
 - 实例, 162, 169
 - 类, 162, 169
- 属性错误, 186
- 嵌套列表, 83, 85, 93
- 工作目录, 129
- 布尔函数, 50, 145
- 布尔运算符, 72
- 带情绪的调试, 188
- 帧, 52
- 序列, 67, 74, 83, 88, 107, 113
- 开发方案
 - 原型和补丁, 145
- 开发计划, 35
 - 原型和补丁, 147
 - 增量, 48, 183
 - 封装和泛化, 34
 - 有计划的, 147
 - 随机漫步编程, 188
- 开根, 49
- 异常, 15, 183, 186
 - 下标错误, 68, 73, 84, 186
 - 值错误, 108
 - 名字错误, 186
 - 属性错误, 186
 - 类型错误, 67, 70, 108, 109, 128, 186
 - 输入输出错误, 130
 - 键错误, 186
- 异常, 捕获, 130
- 引号标记, 9, 35, 69, 184
- 引用, 90, 91, 93
 - 别名, 90
- 弧函数, 31
- 强壮型检查, 148
- 归并模式, 87, 93
- 形状, 115
- 形状错误, 114
- 循环, 30, 35, 110
 - for, 30, 68, 85
 - 下标, 85
 - 关于字符串, 70
 - 嵌套, 164
 - 无限, 185
 - 条件, 185
 - 遍历, 68
- 循环和记数, 70
- 循环定义, 51
- 心中的模型, 187
- 愤怒, 188
- 成员
 - 两分法搜索, 94
 - 二进制搜索, 94
 - 列表, 84
- 扑克, 161, 169
- 打字机, 乌龟, 36
- 执行流程, 53, 55, 168, 186
- 括号
 - 元组位于, 107
 - 压倒优先级, 13
 - 父类在, 166
 - 空, 71
- 括号运算符, 67, 83, 108
- 拷贝, 135
- 拼字游戏, 115
- 持久性, 127, 134
- 挂起, 185
- 捕获, 134
- 接口, 33, 35, 168
- 提示符, 2
- 搜索模式, 70
- 搜索, 两分法, 94
- 搜索, 二进制, 94
- 散布, 109, 115
- 数值, 15
- 数学运算符, 12
- 数据库, 131, 134, 135
- 数据类型, 9
 - 字符串, 9
 - 整数, 9
 - 浮点数类型, 9
- 数据结构, 114, 115
- 整数, 15
- 整数类型, 9
- 文件, 127
 - 压缩, 133
 - 权限, 130
 - 读取和写入, 127
- 文件名, 129

- 文件夹, 129
- 文本
 - 纯, 135
- 文本文件, 134
- 文档字符串, 34, 35
- 斐波纳契函数, 53
- 斜边, 49
- 新行, 164
- 方法, 71, 74
 - cmp--, 163
 - str--, 164
 - append, 86, 91, 164, 165
 - close, 128, 131, 132
 - extend, 86
 - init, 162, 164, 166
 - items, 111
 - join, 89, 164
 - mro, 168
 - pop, 88, 165
 - read, 132
 - readline, 132
 - remove, 88
 - sort, 86, 92, 112, 165
 - split, 89, 108
 - update, 111
 - 字符串, 75
 - 空, 86
 - 记数, 72
- 方法 append, 94
- 方法解析顺序, 168
- 方法, 列表, 86
- 旋转, 字符, 76
- 无穷递归, 54
- 无限循环, 185
- 无限递归, 185
- 映射, 84, 93, 161
- 映射模式, 87, 93
- 普遍化, 148
- 更新
 - 数据库, 131
 - 项目, 85
- 更新运算符, 87
- 最大公约数 (GCD), 57
- 有计划的开发, 147, 149
- 未定义而使用, 15
- 权限, 文件, 130
- 条件, 184, 185
- 条件语句, 51
- 查找模式, 74
- 栈图, 36, 52, 56, 91
- 格式字符串, 128, 134
- 格式序列, 128, 134
- 格式运算符, 128, 134, 186
- 模块
 - anydbm, 131
 - bisect, 94
 - datetime, 150
 - os, 129
 - pickle, 127, 131
 - random, 94, 113, 165
 - reload, 184
 - shelve, 132, 135
 - structshape, 114
 - urllib, 135
 - 重载, 134
- 模块对象, 133
- 模块, 编写, 133
- 模型, 心中的, 187
- 模式
 - DSU, 113
 - 交换, 108
 - 守护人, 55, 73
 - 归并, 87, 93
 - 搜索, 70
 - 映射, 87, 93
 - 查找, 74
 - 监护人, 54
 - 筛选, 87, 93
 - 装饰 -排序 -还原, 113
- 欧几里得算法, 57
- 步长, 75
- 死区代码, 48, 55, 186
- 比较
 - 元组, 112, 164
 - 字符串, 72
- 比较函数, 48
- 毕达哥拉斯理论, 48
- 汇聚
 - 字符串, 164
- 沮丧, 188
- 沼泽, 170
- 泛化, 32
- 注释, 14, 16
- 测试
 - 交互模式, 2
 - 信心的飞跃, 53
 - 增量开发, 48
 - 最小测试案例, 187
 - 直到答案, 49
- 测试案例, 最小, 187

- 浮点数, 15
- 浮点数类型, 9
- 浮点除法, 12
- 源码, 2

- 点符号, 71, 162
- 父类, 166, 169
- 特殊值
 - None, 48, 55, 86, 88
- 特殊值 None, 86, 88
- 特殊情况, 147
- 状态图, 15, 74, 84, 90, 112, 145, 163
- 状态图表, 10
- 玩扑克, Anglo-American, 161
- 生日, 150
- 生日悖论, 94
- 用户定义类型, 145
- 目录, 129, 134
 - 工作, 129
 - 遍历, 129
- 目标, 2
- 相同, 90, 93
- 相对路径, 129, 134
- 相等, 90, 93
- 相等与赋值, 59
- 秘密练习, 135
- 移植性, 1
- 程序, 2
- 程序语句, 12
- 空列表, 83
- 空字符串, 74, 89
- 空方法, 86
- 空白, 134
- 空白符, 184
- 等级, 161
- 筛选模式, 87
- 筛选模式 i, 93
- 算法, 3
 - MD5, 135
 - 欧几里得, 57
- 管道, 132, 135
- 类
 - Card, 162
 - Date, 150
 - Hand, 166
 - Time, 145
 - 子, 166, 169
 - 父, 166
 - 纸牌, 164
- 类图, 167, 169
- 类型, 15
 - 元组, 107
 - 列表, 83
 - 文件, 127
 - 用户定义, 145
- 类型数据, 9
- 类型检查, 53
- 类型错误, 67, 70, 108, 109, 128, 186
- 类属性, 162, 169
- 累加器, 93
 - 列表, 87
 - 和, 87
- 累积求和, 88
- 约束, 148, 149
- 级联, 14, 16
- 纯函数, 146, 149
- 纯文本, 135
- 纸牌类, 164
- 纸牌, 玩, 161
- 纸牌, 玩卡片, 164
- 练习, 秘密, 135
- 组成, 164
- 绝对值函数, 48
- 绝对路径, 129, 134
- 继成, 166
- 继承, 169
- 编码, 161, 169
- 编程语言, 1
- 缩进, 184
- 网络电影数据库 (IMDb), 135
- 置换, 115

- 聚集, 109, 115
- 脚手架, 55
- 脚手架代码, 49
- 脚本, 2
- 脚本模式, 2, 12
- 臭虫, 3
- 花, 36
- 花色, 161
- 行结束符号, 134
- 表示, 161
- 表达式, 12, 13, 16
 - 大而复杂, 187
- 装饰 -排序 -还原模式, 113
- 解决问题, 1
- 解析, 135
- 计数器, 74
- 记数, 70
- 记数和循环, 70

- 记数方法, 72
- 访问, 83
- 语义错误, 10, 15, 74, 183, 187
- 语句, 15
 - assert, 149
 - for, 30, 68, 85
 - import, 133
 - print, 186
 - raise, 149
 - return, 188
 - try, 130
 - 导入, 29
 - 条件, 51
 - 赋值语句, 10
 - 返回, 47
- 语法错误, 3, 15, 183
- 语言
 - 低级, 1
 - 图灵完备, 51
 - 编程, 1
 - 高级, 1
- 调用, 71, 74
- 调试, 3, 15, 35, 54, 73, 92, 114, 134, 148, 168, 183
 - 情绪反应, 188
 - 迷信, 188
- 调试器 (pdb), 186
- 负数下标, 68
- 贡献者, vii
- 赋值, 13, 15, 83, 94
 - 以避免别名, 93
 - 元组, 108--110, 115
 - 增量, 87, 93
 - 项目, 70, 84, 108
- 赋值语句, 10
- 超类, 166
- 跑步比赛, 16
- 跑步速度, 149
- 跟踪者, 170
- 路径, 129, 134
 - 相对, 129
 - 绝对, 129
- 输入输出错误, 130
- 运算符
 - 关系, 163
- 运算数, 12, 15
- 运算符, 15
 - del, 88
 - in, 72, 84
 - is, 89
 - 切片, 69, 75, 85, 92, 108
 - 字符串, 13
 - 布尔, 72
 - 括号, 67, 83, 108
 - 按位, 12
 - 更新, 87
 - 格式, 128, 134, 186
- 运算符优先级, 15, 188
- 运算符的优先级, 13
- 运算符重载, 163
- 运算符, 数学, 12
- 运行时错误, 15, 53, 183, 186
- 返回值, 47
 - 元组, 109
- 返回语句, 47
- 进位, 加法, 146, 148
- 连接, 68, 70, 89
 - 列表, 85, 91, 94
- 连锁词, 94
- 迷信的调试, 188
- 递归, 51, 53
 - 无穷, 54
 - 无限, 185
- 递归定义, 51, 116
- 遍历, 68, 70, 73, 74, 87, 93, 110, 111, 113
 - 列表, 85
 - 字典, 111
- 遍历, 目录, 129
- 重复, 30
 - 列表, 85
- 重构, 33, 34
- 重载, 163, 166, 168
- 错误
 - 形状, 114
 - 编译时, 183
 - 语义, 15, 74, 183, 187
 - 语义错误, 10
 - 语法, 3, 15, 183
 - 运行时, 15, 183
- 错误信息, 3, 10
- 错误检查, 53
- 错误消息, 15, 183
- 键-值对, 111
- 键错误, 186
- 阶乘函数, 51, 53
- 除法
 - 下取整, 12
 - 浮点, 12
- 随机漫步编程, 188

难题, 115
集合
 回文, 115, 132
零, 下标开始, 67
面向对象编程, 166
项目, 74, 83
项目更新, 85
项目赋值, 70, 84, 108
频率
 字母, 115
饰面, 165, 169
高级语言, 1