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
Walter Lewis

Green Tea Press
Needham, Massachusetts

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

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，阅读十页比阅读无十页要好。
- 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。

我需要一个书名，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。

声明

首先，也是最重要的，我要感谢 Jeff Elkner, 是他把我的 Java 书翻译成了 Python，也由此开启了这项“工程“，也把我领进了我最爱的编程语言大门。

我也要感谢 Chris Meyers, 他贡献了《像计算机科学家一样思考》的部分内容。

感谢 FSF 制定的 GNU 自由文档许可证，使我和 Jeff 和 Chris 的合作成为可能。

我也要感谢所以使用以前版本的学生和所有的贡献者，他们提供了宝贵的更正和建议 2。

感谢我的妻子，Lisa 为她在这本书上所花费的努力，还有 Gree Tea Press, 和其他的一切。

贡献者名单

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

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

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

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word ``unconsciously'' in Chapter 1 needed to be changed to ``subconsciously''.
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of ``return value'' in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.

- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have made numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between gleich and selbe.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and helped us update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.
- Florin Oprina sent in an improvement in `makeTime`, a correction in `printTime`, and a nice typo.
- D. J. Webre suggested a clarification in Chapter 3.
- Ken found a fistful of errors in Chapters 8, 9 and 11.
- Ivo Wever caught a typo in Chapter 5 and suggested a clarification in Chapter 3.
- Curtis Yanko suggested a clarification in Chapter 2.
- Ben Logan sent in a number of typos and problems with translating the book into HTML.
- Jason Armstrong saw the missing word in Chapter 2.
- Louis Cordier noticed a spot in Chapter 16 where the code didn't match the text.
- Brian Cain suggested several clarifications in Chapters 2 and 3.
- Rob Black sent in a passel of corrections, including some changes for Python 2.2.
- Jean-Philippe Rey at Ecole Centrale Paris sent a number of patches, including some updates for Python 2.2 and other thoughtful improvements.
- Jason Mader at George Washington University made a number of useful suggestions and corrections.
- Jan Gundtofte-Bruun reminded us that ``a error'' is an error.
- Abel David and Alexis Dinno reminded us that the plural of ``matrix'' is ``matrices'', not ``matrixes''. This error was in the book for years, but two readers with the same initials reported it on the same day. Weird.

- Charles Thayer encouraged us to get rid of the semi-colons we had put at the ends of some statements and to clean up our use of ``argument'' and ``parameter''.
- Roger Sperberg pointed out a twisted piece of logic in Chapter 3.
- Sam Bull pointed out a confusing paragraph in Chapter 2.
- Andrew Cheung pointed out two instances of ``use before def.''
- C. Corey Capel spotted the missing word in the Third Theorem of Debugging and a typo in Chapter 4.
- Alessandra helped clear up some Turtle confusion.
- Wim Champagne found a brain-o in a dictionary example.
- Douglas Wright pointed out a problem with floor division in `arc`.
- Jared Spindor found some jetsam at the end of a sentence.
- Lin Peiheng sent a number of very helpful suggestions.
- Ray Hagtvedt sent in two errors and a not-quite-error.
- Torsten Hübsch pointed out an inconsistency in Swampy.
- Inga Petuhhov corrected an example in Chapter 14.
- Arne Babenhauserheide sent several helpful corrections.
- Mark E. Casida is is good at spotting repeated words.
- Scott Tyler filled in a that was missing. And then sent in a heap of corrections.
- Gordon Shephard sent in several corrections, all in separate emails.
- Andrew Turner `spot`ted an error in Chapter 8.
- Adam Hobart fixed a problem with floor division in `arc`.
- Daryl Hammond and Sarah Zimmerman pointed out that I served up `math.pi` too early. And Zim spotted a typo.
- George Sass found a bug in a Debugging section.
- Brian Bingham suggested Exercise ??.
- Leah Engelbert-Fenton pointed out that I used `tuple` as a variable name, contrary to my own advice. And then found a bunch of typos and a ``use before def.''
- Joe Funke spotted a typo.
- Chao-chao Chen found an inconsistency in the Fibonacci example.
- Jeff Paine knows the difference between space and spam.
- Lubos Pintes sent in a typo.
- Gregg Lind and Abigail Heithoff suggested Exercise ??.
- Max Hailperin has sent in a number of corrections and suggestions. Max is one of the authors of the extraordinary Concrete Abstractions, which you might want to read when you are done with this book.
- Chotipat Pornavalai found an error in an error message.
- Stanislaw Antol sent a list of very helpful suggestions.
- Eric Pashman sent a number of corrections for Chapters 4--11.
- Miguel Azevedo found some typos.
- Jianhua Liu sent in a long list of corrections.
- Nick King found a missing word.
- Martin Zuther sent a long list of suggestions.

- Adam Zimmerman found an inconsistency in my instance of an ``instance'' and several other errors.
- Ratnakar Tiwari suggested a footnote explaining degenerate triangles.
- Anurag Goel suggested another solution for `is_abecedarian` and sent some additional corrections. And he knows how to spell Jane Austen.
- Kelli Kratzer spotted one of the typos.
- Mark Griffiths pointed out a confusing example in Chapter 3.
- Roydan Ongie found an error in my Newton's method.
- Patryk Wolowiec helped me with a problem in the HTML version.
- Mark Chonofsky told me about a new keyword in Python 3.0.
- Russell Coleman helped me with my geometry.
- Wei Huang spotted several typographical errors.
- Karen Barber spotted the the oldest typo in the book.
- Nam Nguyen found a typo and pointed out that I used the Decorator pattern but didn't mention it by name.
- Stéphane Morin sent in several corrections and suggestions.
- Paul Stoop corrected a typo in `uses_only`.
- Eric Bronner pointed out a confusion in the discussion of the order of operations.
- Alexandros Gezerlis set a new standard for the number and quality of suggestions he submitted. We are deeply grateful!

Contents

Chapter 1

编程的方式

这本书的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“。

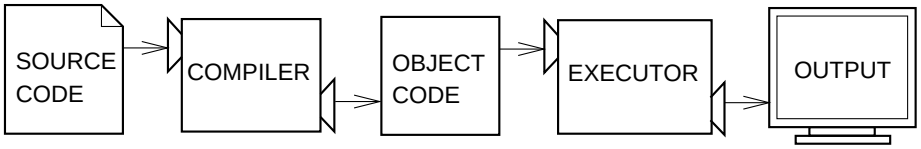
然而，高级语言的优点也是无限的。首先，用高级语言编程是一件非常容易的。用高级语言编程通常花费的时间比较少，同时编写的程序简短，易读，易纠错。第二，高级语言是可移植的，这意味着他们可以不加修改（或者修改很少）地运行在不同的平台上。低级语言编写的程序只能在一个机器上运行，如果想要运行在另外一台机器上，必须得重写。

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 (非常奇怪) 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。如果在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 Holmoes 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。正式语言被设计成近乎没有二义性，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 Python3.0 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 python3.1 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。我希望他们对你有益！

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

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。
t 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_poewe / 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，函数不带有任何参数。

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:

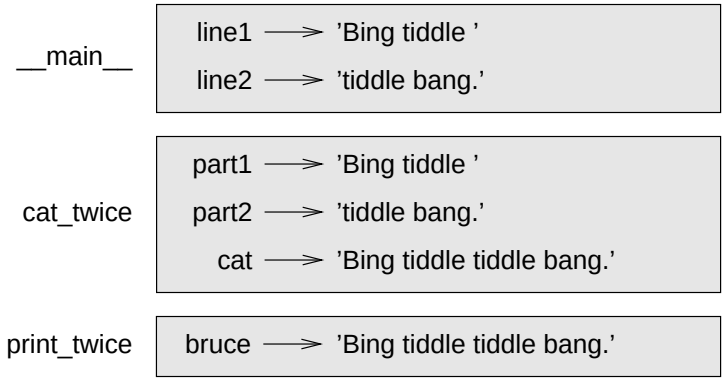
```
>>> print cat
NameError: name 'cat' is not defined
```

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 卓有成效的函数和 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。

functon 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`。

1

[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

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:

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

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 10[4]。

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的类型，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。
[7]c/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。这两个简单的分之都是简单语句，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 <= 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。

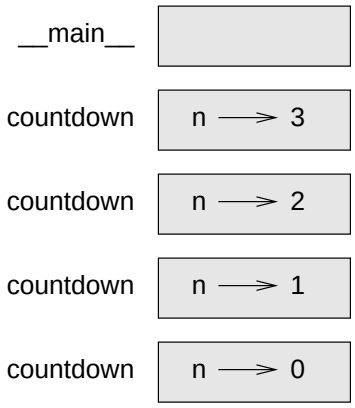
这样简单的例子，其实可以很容易用一个 for 循环来实现。但我们以后将会看到很难写成 for 循环形式，但是很容易用递归实现的例子，我们现在就开始认识递归是有好处的。

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 键盘输如

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之的条件语句。

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

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。

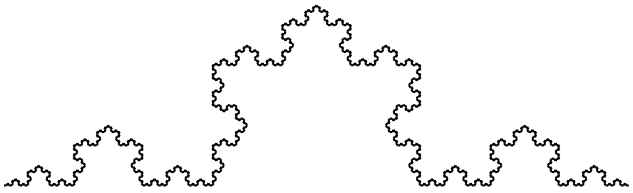
[16]如果两个长度之和等于第三个，他们形成退化的三角形。

接下来的练习使用??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??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

卓有成效的函数

Chapter 7

迭代器

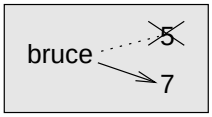
7.1 多重赋值

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?? 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 coutdown 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 的例子里，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，因为我们可以在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 $1/3$，和无理数，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 *pi* 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 *pi* 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鱼 `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

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：

[1] wikipedia.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 (avaliable)。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??) ，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??。

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。

在这两中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?”

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，几年后又会发生了，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`.

[3] `www.cartalk.com/content/puzzler/transcripts/200725`.
[4] `www.cartalk.com/content/puzzler/transcripts/200803`.
[5] `www.cartalk.com/content/puzzler/transcripts/200813`

Chapter 10

列表

Chapter 11

字典

Chapter 12

元组

Chapter 13

实例学习：数据结构的实例

Chapter 14

文件

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的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 0x 表示后面的数字是十六进制的）。

15.2 属性

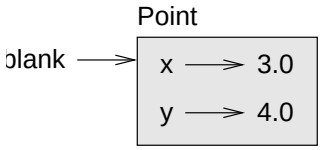
我们可以使用点记法实例赋值。

```
>>> 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 heigth 是数字，corner 是一个类的对象，代表了左下角。

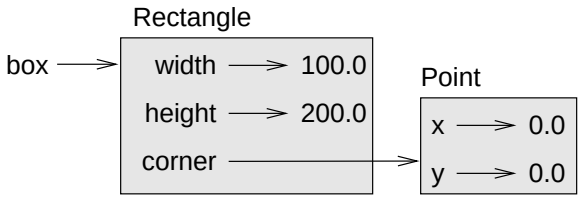
为了代表一个矩形，我们必须实例化一个矩形的对象，给属性赋值：

```
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 cornet；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 `heigth` 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。

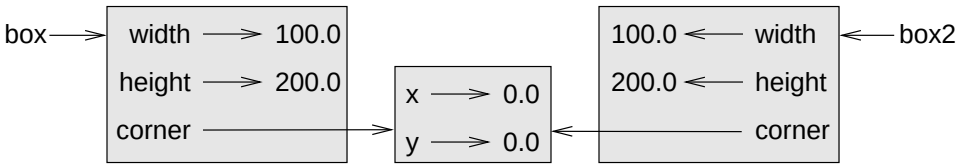
```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

is 操作符表明 p1 和 p2 不是同一个对象，这是我们希望的。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` 模括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。

embededed (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??) 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` 是序列表，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

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??，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，我们所要做的就是把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，也更容一维护和复用。

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（??）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，所有一用有 3 个参数。

17.4 更复杂的一个例子

is_after(联系??）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

基于类型的调度当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??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 unmutable

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_contens 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。使用汉说 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

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，它在按钮按下的时候执行。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，应该在华布上画一个圆。

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'
```

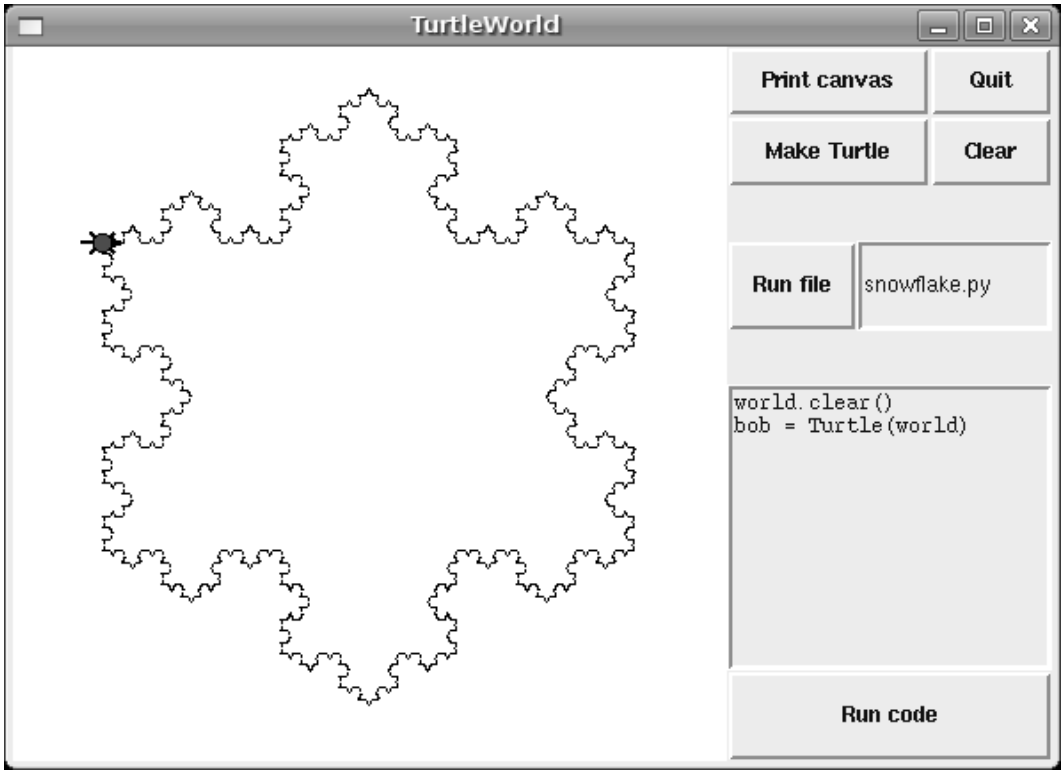
Exercise 19.3 修改练习 19.2的解答，增加一个输入框和按钮。当用户按下新增加的按钮时，程序从输入框读取一个颜色名，用它改变圆的填充色。用 config 修改圆，不要新建一个圆。

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??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的 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米（可以从这个下载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 选项是可调用对象，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）的可调用对象。

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 方法覆盖缺省的帮定，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 Returnmake_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： 表明参数明是函数调用的参数。

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起。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??），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