像计算机
科学家那样思考

（中文版）

目　录

贡献者名单

为阐述自由软件基金会的哲学，这本书是自由的，就像自由言论（free speech）一样，不过不一定像免费披萨（free pizza）一样免费。本书经由合作产生，如果没有 GNU Free Documentation License，这种类型的合作就不可能发生。所以我们想要感谢自由软件基金会发展这个授权，而且让我们得以取用。

我们同时也想感谢一百多个眼睛锐利而且体贴的读者，他们在过去数年间传送给我们建议和修正。依照自由软件的精神，我们决定用贡献者名单的形式表示我们的感谢。可惜的是，这份名单并不完整，但我们会尽最大努力使其包含最新讯息。如果要包含所有仅传送一两个错误的人员，这份名单会变得过于庞大。你除了有我们的感谢，还得到了个人满足感，因为对你和所有使用本书的人来说，你让这本对你有用的书更完善了。第二版名单新增的部分是那些目前仍做出贡献的人。

如果你有机会检视这份名单的话，你应该明瞭这里的每个人仅借着通知我们，就让你和后面的读者不会因技术上的错误，或不甚清楚明白的解释而疑惑。

虽然经过这么多次修正后，似乎不太可能再出错，不过本书中可能仍有些错误。如果你看到其中一个，我们希望你能花点时间联络我们。电子邮件地址是： jeff@elkner.net。如果因你的建议而做出实质改变，你将会被加入下一版本的贡献者名单中（除非你要求略去姓名）。谢谢你！

0.1.1 第二版

- 特别感谢 Jeff 2007 - 2008 学年 HB-Woodlawn 课程中计算机科学班级的学生：James Crowley、Joshua Eddy、Eric Larson、Brian McGrail 和 Iliana Vazuka。你们愿意作为新章节写好时的 beta 测试者，并忍受随着你们的响应而经常进行的修订，在在都证明是无价的。也由于你们，这本书有真正*由学生测试过的*文字。
- Ammar Nabulsi 送来了关于第一章和二章为数众多的修正。
- Aldric Giacomoni 指出了第五章我们关于斐波纳契数列的一个错误。
- Roger Sperberg 送来数个拼字修正，并指出第三章中一个逻辑混乱的段落。

- Adele Goldberg 在 PyCon 2007 和 Jeff 坐下来谈了一会，并给了他一份关于整本书的建议和修正清单。
- Ben Bruno 送来关于第四、五、六和第七章的修正。
- Carl LaCombe 指出我们在第六章错用了 commutative 这个词，其实 symmetric 才是正确的词。
- Alessandro Montanile 送来在第三、十二、十五、十七、十八、十九和二十章中，程序范例和文字错误的修正。
- Emanuele Rusconi 在第四、八和十五章找到错误。
- Michael Vogt 回报第六章中一个范例的缩排错误，并送来第一章关于 shell vs. script 一节的建议，以改善理解度。

0.1.2 第一版

- Lloyd Hugh Allen 送来 8.4 节的一个修正。
- Yvon Boulianne 送来第五章中一个语义错误的修正。
- Fred Bremmer 提交了一个 2.1 节的修正。
- Jonah Cohen 写了一个 Perl 脚本，将本书的 LaTeX 原始档转换成美丽的 HTML 码。
- Michael Conlon 送来第二章中一个文法修正，并改善第一章中的写作风格，另外他也发起了关于编译器技术观点的讨论。
- Benoit Girard 送来对于 5.6 节一个可笑错误的修正。
- Courtney Gleason 和 Katherine Smith 写了 `horsebet.py`，它被用来当成本书早期版本中的实例探讨。他们的程序现在可以在网站上找到。
- Lee Harr 所提交的错误多到我们这里的空间不足以一一列出，他其实应该被列为本文主要编辑之一。
- James Kaylin 是个使用本文的学生。他提交了许多修正。
- David Kershaw 修好了 3.10 节中坏掉的 `catTwice` 函数。
- Eddie Lam 送来了许多关于第一、二和第三章的修正。他同时也修好了 Makefile，让它可以在第一次执行时，建立一个索引，并且帮助我们设定了版本管理系统。
- Man-Yong Lee 送来 2.4 节中范例程序代码的一个修正。
- David Mayo 指出第一章中 unconsciously 这个字应该更正为 subconsciously。

- Chris McAloon 送来 3.9 和 3.10 节中的数个修正。
- Matthew J. Moelter 是个长期贡献者，他送来对本书为数众多的修正和建议。
- Simon Dicon Montford 回报了第三章中遗漏的函数定义和数个拼字错误。他同时也找到了第十三章中 `increment` 函数中的错误。
- John Ouzts 修正了第三章中 return value 的定义。
- Kevin Parks 送来关于如何推广本书的评论和建议。
- David Pool 送来第一章术语中的一个拼字错误，同时也用他亲切的文字鼓励我们。
- Michael Schmitt 送来关于档案和例外章节的一个修正。
- Robin Shaw 指出 13.1 节的一个错误，printTime 函数被使用在范例中，却没有定义。
- Paul Sleigh 发现了一个第七章中的错误，并且在 Jonah Cohen 从 LaTeX 生成 HTML 的 Perl 脚本中找到一个臭虫。
- Craig T. Snydal 在德鲁大学中的一个课程中测试本文。他贡献了数个珍贵的建议和修正。
- Ian Thomas 和他的学生在一堂程序设计课程中使用本文。他们是首批测试本书后半章节的人员，而且他们做出了为数众多的修正和建议。
- Keith Verheyden 送来一个关于第三章的修正。
- Peter Winstanley 让我们知道第三章中的拉丁文里一直存在的错误。
- Chris Wrobel 修改了档案 I/O 和例外章节中的程序代码。
- Moshe Zadka 对这个计划做出了无法估计的贡献。除了撰写 Dictionaries 章节的初稿外，他还在本书的早期阶段提供了不间断的指导。
- Christoph Zwerschke 送来数个修正和教学法上的建议，并解释了 *gleich* 和 *selbe* 之间的不同。
- James Mayer 送来许多拼字和排版错误，包含贡献者名单中的两个。
- Hayden McAfee 在两个范例间抓到了一个有可能导致迷惑的不一致之处。
- Angel Arnal 是国际翻译团队的一员，致力于本书的西班牙文版本。他也同时找到了英文版中的数个错误。
- Tauhidul Hoque 和 Lex Berezhny 绘制了第一章插图，并改善了许多其它的插图。
- Dr. Michele Alzetta 在第八章抓出一个错误，并送来一些关于斐波那契与老处女（Old Maid）的有趣教学法评论和建议。
- Andy Mitchell 找出第一章中的一个拼字错误和第二章中失效的范例。
- Kalin Harvey 对于第七章提出了增进理解度的建议，并抓出一些拼字错误。

- Christopher P. Smith 抓出了数个拼字错误，并帮忙我们准备升级本书到 Python 2.2 版。
- David Hutchins 在前言中找出一个拼字错误。
- Gregor Lingl 在奥地利维也纳的高中里教授 Python。他正进行本书的德文翻译，他在第五章中抓出一些不良的错误。
- Julie Peters 在序言中抓出一个拼字错误。

0.1.3 正体中文版

- 正体中文版由自由软件铸造场同仁于 2008 年开始翻译，中文书名译为 －『如何像计算机科学家一样思考』，其中：
- 张凯庆翻译本书正体中文版第一到八章。
- 徐孟逵校订本书正体中文版第一到八章，并翻译第九到十四章。

前言

David Beazley 著

作为教育家、研究者及书籍作者，我很高兴看到这本书的完成。Python 是个有趣并且非常易用的程序语言，在过去几年里，Python 逐渐地受到欢迎。Guido van Rossum 在十多年前开发了 Python，其简单的语法与整体感觉则是得自于 ABC，这个 1980 年代发展的教学语言。然而，Python 也被设计成可以解决真实的问题，并且也从其它程序语言如 C++、Java、Modula-3 和 Scheme 等，借用了各式各样的特征。正因如此，Python 最显而易见的特色之一就是它广泛吸引了专业软件开发者、科学工作者、研究者、艺术家及教育家。

虽然 Python 吸引了不同社群的人士，你可能还是会怀疑 为何选用 Python？，或者 为甚么教导用 Python 撰写程序？ 回答这些问题可不简单---特别是当大众的意见都站在，如 C++ 和 Java 这些自讨苦吃的选择这一边的时候。然而，我想最直接的答案就是用 Python 写程序可以单纯地得到许多乐趣，并且更有生产力。

当我教授计算机科学课程时，我希望教学题材涵盖重要概念，又能对学生来说有趣且迷人。可惜的是，程序设计入门课程目前倾向投注过多焦点在数学抽象概念上，且让学生因恼人问题而沮丧，如语法、编译以及看起来晦涩难解的规则等低阶细节。尽管这些抽象及形式对专业软件工程师和计划继续学习计算机科学的学生来说，非常重要，在一个入门性的课程采取这种方式大部分只会使计算机科学更无趣。教课的时候，我可不想面对一整间无精打采的学生。我宁愿看到他们透过探索不同的想法、采取有创意的方式、打破成规并从错误中学习，以尝试解决有趣的问题。这么做的原因是，我不想浪费半个学期试着厘清含糊的语法问题、难以理解的编译器错误讯息，或是一个程序可能造成一般性保护错误的数百种方式。

我喜欢 Python 的原因之一，就是它在实做与概念上取得了很好的平衡。既然 Python 是直译的，初学者几乎可以立刻学会这个语言，并做些美妙的事，而不会迷失在编译与连结的问题中。更有甚者，Python 随附大型的模块库，能够应用在各种工作上，包括网络程序设计到图形处理等。具有如此实用的重点是吸引学生注意力的绝佳方式，并且能让他们完成重要的项目。然而，Python 也可以作为介绍重要计算机科学概念的优良基础。由于 Python 完整支持程序和类别，便可逐步引导学生认识程序抽象化、数据结构，以及对象导向程序设计等课题，这些全都可以应用在往后 Java 或 C++ 的课程上。Python 还向函数型程序语言借了许多功能，可以用来介绍这些以往是在 Scheme 及 Lisp 的课堂上详细介绍的概念。

读了 Jeffrey 的序文，我被他的评论所感动，Python 使他看见更高层次的成就以及较低层次的挫折，而他可以工作得更快且获得较佳结果。尽管这些评论是指他的入门课程，我有时会以同样的理由，将 Python 应用在芝加哥大学的进阶研究所计算机科学课程。在这些课程里，我不断地面对将大量困难的课程资料，包含在极短的九星期学季中这种让人气馁的工作。虽然我必定可以承受使用像 C++ 这种语言所带来的大量痛苦及折磨，我常常发现这种方式有着不良的后果，特别是这些课程的主题不只是关于程序设计时。我发现使用 Python 让我能够较为集中在实际的主题上，同时也使学生能够完成重要的课程作业。

虽然 Python 仍是年轻、发展中的语言，我相信它在教育上会有耀眼的未来。这本书在这方向上是重要的一步。

David Beazley，芝加哥大学，Python Essential Reference 作者

序言

By Jeffrey Elkner

本书的存在归功于网络及自由软件运动所实现的合作方式。它的三位作者---一位大学教授、一位高中老师，以及一位专业程序设计师---尚未见过面，但是我们已经能够紧密合作，并且受到许多愿意付出时间与心力的杰出人士帮助，使得这本书更臻完善。

我们认为本书证明了此种合作方式的优点以及未来的可能性，而这样的合作架构已经由 Richard Stallman 和自由软件基金会运作多年。

我如何及为甚么使用 **Python**

1999 年，大学委员会的「先修大学计算机科学课程」（Advanced Placement Computer Science）考试，第一次用 C++。和全国各地的许多高中一样，维吉尼亚州阿灵顿郡约克敦高中的计算机科学课程也直接受到变更语言的决定影响，我就在此所高中任教。在此之前，Pascal 是我们一年级及先修课程的教学语言。为了和过去让学生有两年的时间接触相同语言的作法一样，我们决定在 1997 到 1998 学年的一年级课程改用 C++ 教学，如此我们便能衔接大学委员会对下年度先修计算机科学课程的改变。

两年后，我确信使用 C++ 为学生介绍计算机科学，不是个适当的选择。虽然它的确是个非常强大的程序语言，但它也是个极度难以学习与教导的语言。我发现我自己不断地对抗 C++ 困难的语法以及处理相同事务的多重方式，结果让我无谓地失去了许多学生。我确信一定会有更适合一年级课程的语言，于是开始寻找一个可以替代 C++ 的选择。

我需要一个能够运作在我们 GNU/Linux 实验室语言，但同时也要能够运作在大部分学生家里的 Windows 及 Macintosh 平台。我希望它是个自由软件，好让学生不管收入多少，都能够在家使用。我想要一个专业程序设计师使用的语言，并且有个活跃的开发社群。它必须同时支持程序式及对象导向程序设计。最重要的是，它必须容易学习和教授。当我以这些条件研究过许多选择后，Python 脱颖而出成为此项工作的最佳候选人。

我请一位聪明的约克敦高中学生，Matt Ahrens，尝试使用 Python。他不但在两个月内学会这个语言，而且写了一个叫做 pyTicket 的应用程序，让我们的人员可以透过网络回报技术

问题。我知道 Matt 无法在这么短的时间用 C++ 完成如此规模的应用程序，这项成就和 Matt 对 Python 的正面评价，显示 Python 是我所寻找的解决方案。

寻找教科书

我决定在下年度的两个计算机科学概论课程都使用 Python 后 ，最迫切的问题是缺少一本可用的教科书。

自由文件解决了这个问题。Richard Stallman 在今年稍早向我介绍了 Allen Downey 这个人。我们两个都写信向 Richard 表达发展自由教育资料的兴趣。Allen 已经写了本一年级计算机科学的教科书，*如何像计算机科学家一样思考*。当我读了这本书，我立即知道我想将它应用在我的课程中。它是我见过最清楚且最有帮助的计算机科学教科书。它突显出在写程序时的思考过程，而非特定语言的功能。阅读它使我马上变成了更好的老师。

*如何像计算机科学家一样思考*不只是一本出色的书，并且它是以 GNU 公众授权发布，这就是说它可以自由使用，并可依使用者需要修改。一旦我决定使用 Python 后，我想到我可以转换 Allen 这本书的原始 Java 版本到新的语言。也许我本来无法自己撰写一本教科书，但从 Allen 的书改写使我可以达成这个工作，这同时也证明了在软件上运作良好的协力发展模式，也可以使用在教学资料上。

过去两年来编写这本书对学生及我都受益匪浅，而我的学生在此过程中扮演了一个重要的角色。既然我够在有人发现拼写错误或是阅读困难的章节时立即修正，因此我在课文因他们的建议而修正时，为他们加分，以鼓励他们寻找本书的错误。这么做有着双重的好处，一是鼓励他们更加小心地阅读课文，另一个好处则是让课文由它最重要的评论者---使用它学习计算机科学的学生---澈底审查。

针对本书后半部关于对象导向程序设计的部分，我知道我需要一个比我更有实际程设经验的人来使它更为正确。本书在大部分时间处于未完成的状态，直到开放源码社群再次提供完成它所需要的工具。

我收到一封 Chris Meyers 寄来的电子邮件，表达他对本书的兴趣。Chris 是一位专业程序设计师，他去年在奥勒岗州尤金市的 Lane 小区大学开始使用 Python 教授程序设计课程。教授这门课程让 Chris 找到这本书，而且他立即开始帮忙本书的编撰。在学年结束前，他在我们的网站 http://openbookproject.net 设置了一个指南手册计划 ，叫做 *Python for Fun*，他并担任指导教师，与我最顶尖的学生一起合作，带领他们进入超出我能力之外的领域。

用 Python 介绍程序设计

过去的两年里，转换及使用*如何像计算机科学家一样思考*的过程已经证实 Python 适于教导初学的学生。Python 极其简化程序设计范例，并且让重要的程序设计观念较容易教授。

课文中的第一个范例说明了这一点。传统的 Hello, world 程序，在本书的 C++ 版本看来如下：

```
#include <iostream.h>

void main()
{
   cout << "Hello, world." << endl;
}
```

在 Python 版本中，它变成了：

```
print "Hello, World!"
```

虽然这是个平凡的例子，Python 的优点脱颖而出。约克敦高中的「计算机科学第一级」没有先修课程，所以许多看到此例的学生是第一次看到一个程序。他们之中有些人听说计算机程序设计难以学习后，无疑地有一点紧张。本书的 C++ 版本总是强迫我在两个令人不满意的选项中选择：不是冒着一开始就使某些学生感到困惑或恐惧的风险，解释#include、void main()、{ 和 } 陈述，就是顶着相同的风险告诉他们 现在不必担心这些东西，我们稍后会再讲到这些陈述。课程此时的教学目标是介绍学生程序设计陈述的概念，并让他们撰写第一个程序，从而介绍程序设计环境。Python 程序正好拥有进行这些事所需的所有条件，既不多也不少。

比较本书不同版本对此程序的解释文字，更进一步说明了这对初学的学生有什么样的意义。C++ 版本对于 Hello, world! 有着十三段的解释文字；在 Python 版本中只有两段。最重要的是，缺少的那十一段文字并非讲述计算机程序设计中的重要观念，而是述说 C++ 语法的琐碎细节。我发现整本书都发生同样的事情。Python 更清楚的语法使整个段落显得不必要，而直接消失在 Python 版本的课文中。

使用像 Python 这种非常高阶的语言，让教师得以延后谈论关于机器的低阶细节，直到学生拥有所需的背景知识，以进一步理解这些细节。它因此使教学能够先解决重要的事。关于这点的绝佳范例之一便是 Python 处理变量的方式。在 C++ 中，变量是一个用来容纳东西的位置的名称。变量至少有部份必须宣告型态，因为必须先决定它们所指涉之位置的大小。如此一来，变量的概念就与机器硬件紧密结合在一起。变量强大且基础的概念对初学的学生来说，已经十分困难（同时就计算机科学及代数两个层面来看）。字节及地址对问题没有帮

助。在 Python 中，变数是指涉某个东西的名字。这对初学的学生来说，是个更为直觉的概念，并且也更为接近他们在数学课中学到的变量定义。今年我教导变量比过去少了很多困难，而且我花费较少的时间指导使用它们有问题的学生。

Python 有助于教导和学习程序设计的另一个范例是它的函数语法。我的学生对理解函数总是有着极大的困难。主要问题集中在函数定义和函数呼叫间的差别，以及与其相关的参数和自变量的区别上。Python 以十分美丽的语法解决了这个问题。函数定义由关键词 def 开始，所以我只要告诉我的学生：当你定义一个函数，由 def 开始，后面加上你所定义函数的名称；当你呼叫函数时，只要呼叫（输入）它的名称即可。参数和定义一起使用；自变量则与呼叫一起使用。没有传回型态、参数型态，或是参考及数值参数干扰，所以现在我可以用比之前少一半的时间教导函数，学生也理解得更好。

使用 Python 改善了我们的计算机科学课程对所有学生的成效。与我教授 C++ 的两年相较，我发现平均成功标准提高了，同时也降低了挫败的程度。我前进得更快，并且能得到更好的成果。有更多的学生结束这门课程时，能够设计有意义的程序，并且对这门课所产生的程序设计经验带着正面的看法。

建立社群

我收到全球使用这本书学习或教导程序语言的人寄来的电子邮件。一个使用者社群已经开始兴起，而且有许多人寄来可应用在附属网站 http://openbookproject.net/pyBiblio的数据，为这个计划做出贡献。

随着 Python 的持续发展，我预期这个使用者社群会持续并加速成长。这个使用者社群的出现，及它对类似的教育者合作计划所提出的可能性，对我来说是执行这个计划最令人兴奋的部份。藉由共同工作，我们能够增进可用数据的质量，并节省宝贵的时间。我邀请你参加我们的社群并期待你的来信。来信请寄至 jeff@elkner.net。

Jeffrey Elkner

约克敦高中

阿灵顿郡，维吉尼亚州

1 程序之道

这本书的目标是要教你像计算机科学家一样思考。这样的思考方式结合数学、工程学及自然科学一些最优良的特色。计算机科学家像数学家，使用形式语言表达概念（特别是在计算方面）。他们也像工程师般设计东西，组合原件成系统，然后从中评估成本效益。他们还像自然科学家一样，观察复杂系统的变化，提出假说，并且测试所预期的结果。

计算机科学家最重要的一项能力就是**解决问题**。解决问题的意思是指能够系统式地阐述问题，思考解决方法时别具创意，并且清楚正确地表达解决方法。结果证明，学习程序设计的过程是个练习问题解决技巧的绝佳机会。这也是为甚么这一章叫做：程序之道。

从一方面来说，你将学会设计程序这个有用的技巧。对另一方面而言，你将利用程序设计作为工具，达成目标。随着我们的教学，这目标会越来越清楚。

1.1 Python 程序语言

你将学习程序语言是 Python。Python 是**高级语言**的一种，你可能听过的其它高级语言有 C++、PHP 及 Java。

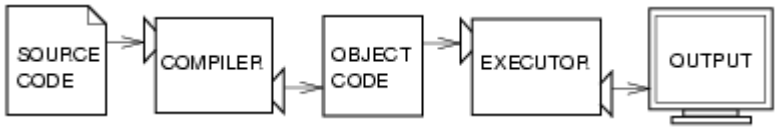
你可以从高级语言这个名字猜测到，也有所谓的**低级语言**，有时也称为机器语言或是汇编语言。广义来说，计算机只能执行用低级语言写的程序。因此，用高级语言写的程序在执行前必须先经过处理。这种额外的处理需要一些时间，这是高级语言的一个小缺点。

但其优点为数众多。首先，用高级语言撰写程序更容易。撰写高级语言程序所需的时间更少，程序更短也更容易阅读，而且正确性更高。其次，高级语言是**可移植的**，这表示它可以在经过些许修改后，运作在其它计算机架构上，有时甚至完全不需修改。低级语言程序只能运作在特定种类的计算机上，并且必须重写，才能在其它的计算机上执行。

由于这些优点，几乎所有的程序都是由高级语言来写的。低级语言只用在一些专业应用程序上。

有两种程序可以让高级语言转换成低级语言的，分别是**直译器**和**编译器**。直译器读取高级语言程序并执行它，这意味着直译器直接执行程序的指示。它一次处理一点程序，交错地读取程序代码和执行计算。

编译器则在程序开始执行前读取程序并将其完全翻译。在这种情况下，高级语言程序称为**原始码**，编译后的程序则叫做**目标码**或是**执行文件**。一旦程序编译完成，你可以重复地执行，而不需进一步翻译。



许多现代的语言同时使用两种程序。它们先编译到一种称为**位码**的较低级语言，然后使用一个称为**虚拟机器**的程序直译。Python 同时使用两种程序，但因其与程序设计师互动的方式，它通常被认为是一种直译式许言。

有两种方式使用 Python 直译器：*shell 模式*及*脚本模式*。在 shell 模式下，你将 Python 陈述输入 **Python shell** 中，直译器会立刻印出结果来：

```
$ python
Python 2.5.1 (r251:54863, May  2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

这个例子的第一行，是在 Unix 命令提示符下启动 Python 直译器的命令。下三行是直译器提供的讯息。第四行以 >>> 起始，这是 Python 提示符。直译器使用这个指示符表示它已准备好接受指示。我们输入 print 1+1，直译器回应结果为 2。

另外，你也可以将程序写入一个档案中，然后利用直译器执行档案内容。这样的档案就叫做**脚本**。例如，我们使用文字编辑器建立一个名为 firstprogram.py 档案，并包含下列内容：

```
print 1 + 1
```

传统上，包含 Python 程序的档案有着以 .py 结尾的名称。

要执行这个程序，我们必须要告诉直译器脚本的名称：

```
$ python firstprogram.py
2
```

这些例子说明 Python 运作在 Unix 命令列的情形。在其它程序开发环境中，程序执行的细节可能会不一样。另外，大多数的程序比这个例子有趣许多。

本书范例同时使用 Python 直译器和脚本。你将可以分辨该使用哪一种方式，因为 shell 模式范例永远以 Python 提示符起始。

在 shell 模式中工作可以方便测试简短的程序代码，因为你可以获得立即回馈。请将它想成用来帮你找出问题答案的便条纸。任何长于数行的程序都应该放到脚本中。

1.2 程序是什么？

程序是一连串具体说明如何计算的指令。这种计算可能是数学的，像是找到方程组的解或是多项式的根，也可能是一种象征性的计算，就像在文件中搜寻并取代文字，或（说来也奇怪）编译一个程序。

不同程序语言的详细情况看起来都不一样，但有一些基本的指令，几乎在每种程序语言中都可以发现：

输入：

从键盘、档案或是其它装置取得数据；

输出：

在屏幕上显示数据，或着是将数据传送到档案或是其它装置；

数学：

执行基本的数学运算，如加法和乘法；

条件执行：

检查特定条件，并执行适当的陈述序列；

重复：

反复执行某些动作，通常会有些变化。

不管你相信与否，就是这么多了。你曾使用过的每个程序，不论有多复杂，都由或多或少类似的指令组成。因此，我们可以把程序设计当成是一种拆解的过程，将大型、复杂的任务，逐步分离成愈来愈小的子任务，直到这些子任务简单到能使用这些基本指令执行为止。

这可能听起来有点糊，当我们稍后讨论到**算法**时会再回到这个主题。

1.3 除虫是什么？

程序设计是一个复杂的过程，而且因为是由人类完成，所以常导致错误。为了一些奇怪的理由，程序设计的错误称为**虫**，而追踪这些虫，并修正的过程称为**除虫**。

程序里有三类可能发生的错误：句型错误、执行错误以及语意错误。为了更快速地追踪它们，清楚分辨这三种错误是很有用的。

1.4 语法错误（Syntax errors）

Python 只能够执行语法正确的程序，否则程序就会执行失败，并传回错误讯息。**语法**就是指程序的结构，以及结构的规则。举例来说，在英文中，句子一定是以大写字母起始，并以句点为结束。下列例句就犯了英文的**语法错误**：「this sentence contains a syntax error.」。此句亦有语法错误，「So does this one」。

对大多数读者来说，一些语法错误并不是个严重问题，这是为甚么我们读康明思（e. e. cummings）的诗却不会吐出错误讯息。Python 则没有这么宽容。如果程序中任何地方有一个语法错误，Python 会印出错误讯息并结束程序。你将无法执行你的程序。在你程设生涯中最初的几个星期，你大概会花很多时间追踪语法错误。然而，当你的经验增长，你将犯较少错，并能更快地找到错误。

1.5 执行错误（Runtime errors）

第二种类型的错误叫做执行错误，会这么命名是因为这种错误直到执行的时候才会出现。这种错误也称为**异常**，因为它们通常表示某种异常（而且不好的）事情发生。

执行错误很少发生在头几章简单的程序里，所以你可能要等好一会才会遇到。

1.6 语意错误（Semantic errors）

第三种类型错误叫做**语意错误**。如果有语意错误在你的程序里，程序仍会顺利的执行，因此计算机不会产生任何的错误讯息，但是程序不会做正确的事情。程序还是会执行另一些事情，特别是你叫程序执行的事。

问题是你写的程序并非是你想要写的程序。程序的意义（它的语意）是错的。判定语意错误可能是困难的，因为需要你检视程序的输出，并尝试找出程序正在执行的事，以回溯你的工作。

1.7 实验性除虫

除虫是你会获得的重要技术之一。虽然除虫工作可能令人沮丧，它却是程序设计中最富机智、最具挑战性以及最有趣的部份。

在某些方面，除虫就像侦探工作一样。你要依据得到的线索，推论出会导致所见结果的程序与情况。

除虫也像是实验科学。一旦你知道什么出错了，你修改你的程序并且再测试一次。如果你的假说是对的，你就可以预期修改的结果，并能更接近可以运作的程序。但如果你的假说是错的，你就必须想出一个新的假说。就像夏洛克．福尔摩斯指出的：当你排除不可能的，剩下的即使不太可能，那也必定是真相。（柯南．道尔，*四个人的签名*）

对有些人来说，程序设计与除虫是同一件事。也就是说，程序设计就是逐步除虫的过程，直到程序执行你想要的事。这个概念是说，你应该以一个可执行*某件事*的程序开始，然后在除虫的过程中进行些微修改，好让你永远有个可运作的程序。

举例来说，Linux 是个包含着数千行程序代码的操作系统，但是它最初只是个 Linus Torvalds 用来探索 Intel 80386 芯片的简单程序。依据 Larry Greenfield 所说，Linus 早期的计划是个将屏幕印出的 AAAA 变成 BBBB 的程序。这个程序稍后演变成为 Linux。（*The Linux Users' Guide* Beta Version 1）

往后的章节会提供更多除虫及其它程设实务的建议。

1.8 形式语言及自然语言

自然语言是人们所讲的语言，如英语、西班牙语和法语。它们并非是由人所设计的（虽然人们尝试将某种规则套用在它们上面），而是自然演变而成。

形式语言是人们为特定应用所设计的语言。举例来说，数学家所使用的标记法就是一种形式语言，这种语言特别适合表示数字与符号间的关系。化学家也使用一种形式语言表现分子的化学结构。最重要的是：

程序语言是设计来呈现计算的形式语言。

形式语言对于语法有严格的规则。例如，3+3=6 是一个语法正确的数学陈述，但 3=+6$ 不是。H_2O 是语意正确的化学名称，而 $_2Zz$ 则否。

语法规则分为两种，分别属于 **标记**与结构。标记是程序语言的基本组件，就像字词、数字和化学元素等。3=+6$ 的问题之一是 $ 并非数学的正确标记（至少就我们所知道的来说）。同样的，$_2Zz$ 也不正确，因为没有一种元素的缩写是 Zz。

第二种语法规则属于陈述的结构---也就是说，标记的排列方法。3=+6$ 这个陈述的结构不正确，因为等号不能马上接加号。同样的，分子式的下标必须在元素名称的后方，而非前方。

当你阅读英文句子或是形式语言的陈述，你必须去理解句子的结构是什么（虽然在自然语言里，你会下意识地理解）。这种过程称为**分析**。

举例来说，当你听到 The other shoe fell 这个句子，你就能瞭解 the other shoe 是主词，而 fell 是动词。一旦你分析了一个句子，你可以理解它的意义是什么，或说是这个句子的语意。假设你知道「shoe」是指什么以及「fall」的意义，你就能了解这个句子大致上的含意。

虽然形式语言和自然语言有许多相同特征□□□标记、结构、语法和语意□□□它们依然有许多相异之处：

歧义性：

自然语言充满着歧义，人们总是利用上下文的线索和其它信息来理解。形式语言设计成近乎完全或是完全消除歧义性，这就是说不管上下文的内容为何，每个陈述都正好只有一个意义。

冗赘性：

为了弥补歧义以及减少误解，自然语言使用了大量赘词。结果使得自然语言往往非常冗长。形式语言则有较少赘词，并且更简洁。

字面性：

自然语言充满成语和隐喻。如果有人说，The other shoe fell，则可能没有鞋子也没有任何东西掉落。形式语言的字义则精确地相符。

在成长时使用自然语言的的人们---也就是说，每个人---常在适应形式语言时遭遇困难。在某些方面，形式语言和自然语言间的差别就像诗及散文，更进一步来说：

诗歌：

文字使用音义并重，整首诗合起来产生一种意义或着是情绪呼应 。歧义性不仅普遍，而且常是刻意营造。

散文：

字面意义较为重要，但是文章结构提供了更多含意。散文比诗歌更容易解析，但仍常有歧义。

程序：

计算机程序不会有歧义，可以照字面了解，并且能够经由分析标记与结构而完全理解。

以下是阅读程序（及其它形式语言）的建议。首先，记得形式语言的结构比自然语言更为紧密，因此需花较多时间阅读。另外，结构是非常重要的，所以从头读到尾，从左读到右不见得是个好方法。相反地，你要学着在心里分析程序，也就是辨认标记并且转译结构。最后，细节是非常重要的。像是拼写或标点错误这种小事，在你使用自然语言的时候能够略过，但在形式语言中可能造成极大的不同。

1.9 第一个程序

传统上，使用一个新语言所撰写的第一个程序称为 Hello, World!，因为这程序只会显示 Hello, World! 文字在屏幕上。在 Python 中，这个程序看起来像这样：

```
print "Hello, World!"
```

这是一个 print 陈述的例子，它并不会真的印什么东西在纸上，而是在屏幕上印出一个数值。在这个例子中，结果就是下列文字：

```
Hello, World!
```

程序里的两个双引号标示该数值的起迄点，它们不会出现在结果中。

有些人以 Hello, World! 程序的简洁性，来判断一种程序语言的品质。以这个标准来看，Python 几乎做到了尽善尽美。

1.10 术语

算法（algorithm）：

解决一种问题的大致步骤。

臭虫（bug）：

程序里的错误。

位码（byte code）：

介于原始码与目标码的中介语言。许多现代程序语言会先编译原始码到位码，然后使用一个称为 *虚拟机器* 的程序直译位码。

编译（compile）：

把用高阶程序语言写的程序整批翻译成低级语言，以利稍后执行。

除虫（debugging）：

找到及移除三种程序设计错误的过程。

异常（exception）：

执行错误的另一个名称。

执行档（executable）：

可执行目标码的另一个名称。

形式语言（formal language）：

人们为了特定目标而设计的任何语言，如呈现数学想法或是计算机程序；所有的程序语言都是形式语言。

高级语言（high-level language）：

一种设计成让人容易读写的程序语言，如 Python。

直译（interpret）：

以一次翻译一行程序代码的方式执行高级语言程序。

低级语言（low-level language）：

一种设计成让计算机容易执行的程序语言，也称为 机器语言 或 汇编语言。

自然语言（natural language）：

人类所讲的任何一种自然演变的语言。

目标码（object code）：

编译器翻译程序后的输出结果。

分析（parse）：

检查程序并分析语法结构。

移植性（portability）：

一种能运作在多种计算机的程序特性。

print 陈述（print statement）：

一个使 Python 直译器在屏幕上显示数值的指令。

解决问题（problem solving）：

阐述问题，找到解决方法，并表达解决方法的过程。

程序（program）：

一连串详细说明计算机活动及执行计算的指令。

Python shell：

Python 直译器的交互式使用者接口。Python shell 的使用者在提示符号（>>>）后输入命令，按下 return 键直接传送这些命令到直译器处理。

执行错误（runtime error）：

直到程序开始执行时才发生的错误，这种错误会阻止程序继续执行。

脚本（script）：

储存在一个档案中的程序（通常是直译式的程序）。

语意错误（semantic error）：

一种程序中的错误，使该程序执行程序撰写者预期外的事情。

语意（semantics）：

程序的意义。

原始码（source code）：

在编译前的高级语言程序。

语法（syntax）：

程序的结构。

语法错误（syntax error）：

一种使程序不能分析的错误（也因而不能直译）。

标记（token）：

程序语法结构中的基本元素，与自然语言中的字词类似。

1.11 练习

1. 写出一个语法错误，但语意可理解的英文句子。写出另一个语法正确，但有语意错误的句子。

2. 启动 Python shell。键入 1 + 2 按下 return 键。Python *计算*这个*表达式*，印出结果，接着印出另一个提示符号。* 是 *乘法运算子*, 而 ** 是 *乘幂运算子*。试着输入不同的表达式，并且记录 Python 直译器印出什么。如果你用 / 运算子会发生什么事情？结果是你预期的吗？解释看看。

3. 输入 1 2 并按下 return 键。Python 试着计算这个表达式，因为这个表达式语法不正确，因此无法计算。取而代之的，Python 会印出错误讯息：

```
  File "<stdin>", line 1
    1 2
      ^
SyntaxError: invalid syntax
```

在许多情况下，Python 会指出语法错误发生的地方，但并非每次都是正确的，而且它并不会给你多少关于该错误的讯息。因此，大部分都要靠你学习语法规则解决。

在这个例子中，Python 抱怨的原因是两个数字中间没有运算符号。

写下另外三个输入 Python 后，会产生错误讯息的字符串。解释各个例子为何不是有效的 Python 语法。

4. 键入 print ’hello'。Python 执行这个陈述，结果是将字母 h、e、l、l、o 印出。请注意你用来包住字符串的双引号，并不是输出结果的一部分。

现在，输入 print ’″hello″’，然后描述并解释你的结果。

5. 输入不包含双引号的 print cheese。输出结果会像是这样：

```
Traceback (most recent call last):
   File ″<stdin>″, line 1, in ?NameError: name ’cheese’ is not defined
```

这是一个执行错误，确切地说，这是个 NameError，说得再具体一点，错误的原因是 cheese 这个名称没有定义。如果你还不知道这是什么意思，你很快地就会瞭解原因。

6. 在 Python 提示符后输入 ’This is a test...’ 并按下 enter。记下结果。现在用以下内容建立一个名为 test1.py 的 Python 脚本（请确定在你尝试执行前已储存档案）：

```
’This is a test...’
```

当你执行这个脚本时发生了什么事？现在改变脚本的内容如下：

```
print ’This is a test...’
```

再执行一次。这次发生了什么？

无论何时在 Python 提示符后输入 *表达式*，Python 都会 *计算* 该表达式，然后把结果印在下一行。’This is a test...’ 是一个求取 ’This is a test...’ 数值的表达式（就像 42 是个求取 42 数值的表达式）。然而，在一个脚本中，表达式的计算不会传送至输出结果，所以必须将它清楚地印出。

2 变数、表达式及陈述

2.1 数值与型态

数值，如一个字母或是一个数字，是程序处理的基础事项之一。到目前为止我们所看过的数值有 2(1 + 1 的相加结果)，和 "Hello, World!" 。

这些数值分属于不同的**型态**：2 是*整数*，"Hello, World!" 则是*字符串* ，其名称是因字符串包含一连串字母而来。你（和直译器）都能标识符串，因为它们包含在引号中。

print 陈述也能用于整数。

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

如果你不能确定一个数值的型态，直译器可以告诉你。

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

不出所料，字符串属于 **str** 型态，而整数属于 **int** 型态。要注意的是，带有小数点的数字属于一种称为 **float** 的型态，因为这些数字是用一种称为 *浮点数（floating-point）*的格式表示。

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

那像是 "17" 及 "3.2" 的数值呢？它们看起来像数字，不过它们像字符串一样包含在引号中。

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

它们是字符串。

Python 里的字符串可以使用单引号（'）包住，也可以用双引号（"）。

```
>>> type('This is a string.')
<type 'str'>
>>> type("And so is this.")
<type 'str'>
```

使用双引号的字符串中可以包含单引号，如 "Bruce's beard"，而使用单引号的字符串中可以包含双引号，如 'The knights who say "Ni!"'。

当你输入很大的整数时，你可能会想要用逗点分组，就像 1,000,000 一般。这在 Python 中虽然不是个合法整数，但却可以顺利执行：

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

这完全不是我们所预期的结果！Python 会把 1,000,000 当成要打印的三个项目列表。所以记得不要放逗点在你的整数之中。

2.2 变数

程序语言最强大的特色之一就是处理**变量**的能力。变量就是代表一个数值的名称。

指派陈述可以建立新的变量，并给予它们数值：

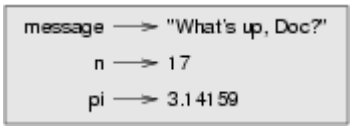
```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

这个例子执行了三个指派任务。第一个任务是将 "What's up, Doc?" 指派到名为 message 的新变数中。第二个任务是将整数 17 指派到 n，第三个任务则是将浮点数 3.14159 指派到 pi。

不要将**指派运算子**，=，和等号搞混（虽然它使用了同一个字符）。指派运算子连结左手边的*名称*和右手边的*数值*。这就是你输入下列式子后，会得到错误讯息的原因：

```
>>> 17 = n
```

在纸上表示变量的常用方法是写下变量名称后，画个箭头指向该变量的数值。这种图表就称为**状态图**，因为它标示出每个变量处于哪种状态（请将它想成变量的心理状态）。此状态图显示了指派陈述的结果：



print 陈述也能使用在变量上。

```
>>> print message
What's up, Doc?>>> print n
17
>>> print pi
3.14159
```

每个范例的结果都是印出变量所指的数值。变量一样具有型态，我们可以用前述方法询问直译器它们所属的型态。

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

变量数值的型态就是变量的型态。

2.3 变量名称与关键词

程序设计师通常会为变量选择一个有意义的名称，它们记录了该变量的用途。

变量名称可以为任意长度，也可以同时包含字母与数字，但必须以字母开头。虽然也可以使用大写字母，但通常我们不如此做。如果你同时使用大小写字母，请记住大小写有分别。如 Bruce 与 bruce 是不同的变数。

底线符号（_）可以出现在变量名称中。它通常用于多字的的名称中，例如 my_name 或 price_of_tea_in_china。

如果你给予变量一个不合法的名称，就会发生语法错误：

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

76trombones 因为不是以字母开头而不合法。more$ 则是因为使用了一个不合法的字母：美金符号。那 class 这个名称又有什么错误呢？

这是因为 class 是 Python 的**关键词**之一。关键词用来定义程序语言的规则与结构，不能当成变量名称。

Python 有 31 个关键词：

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

def for lambda try

你也许会想将此表放在手边。当 Python 直译器向你抱怨你的变量名称之一，而且你不知道为什么的时候，检查一下它是否在这张表上。

2.4 陈述

陈述是个 Python 直译器可以执行的指令。我们已经见过两种陈述：print 与指派。

当你在命令列输入陈述时，Python 就会执行它并且在有结果时显示执行的成果。Print 陈述的结果就是一个数值。而指派陈述则不会产生任何结果。

一个脚本通常包含一连串的陈述。如果脚本包含一个以上的陈述，结果会随着陈述的执行一个个显示。

举例来说，下列脚本

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

会产生下列输出

```
1
2
```

我们可以再次看到，指派陈述不会产生结果。

2.5 表达式求值

表达式是数值、变量以及运算符号的组合。如果你在命令列中输入表达式，直译器就会**求取**它的值，并且显示结果：

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

*表达式的求值*会产生一个数值，这也是为甚么表达式可以出现在指派陈述的右手边。一个单独的数值就是一个简单的表达式，变量也有相同的特性。

```
>>> 17
17
>>> x
2
```

令人迷惑的是，求取表达式的值和打印一个数值不完全相等。

```
>>> message = "What's up, Doc?"
>>> message
"What's up, Doc?"
>>> print message
```

```
What's up, Doc?
```

当 Python shell 显示表达式的数值时，它会使用你输入一个数值时所用的格式。在字符串的例子中，这表示它会包含引号。然而 print 陈述会印出表达式的数值，在这个例子中，就是字符串的内容。

在脚本里，一个单独的表达式是个合法的陈述，但是没有任何功能。以下的脚本

```
17
3.2
"Hello, World!"
1 + 1
```

丝毫不会有任何输出。你要如何更改这个脚本以显示这四个表达式的数值呢？

2.6 运算子与操作数

运算子是一些用来呈现加减等计算的特殊符号。运算子所使用的数值称为**操作数**。

下列皆是意义大致上清楚的合法 Python 表达式：

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

加号 +、减号 -、除号 /，以及用来分组的括号 () 在 Python 中的意义和其数学意义一样。星号（*）则为相乘符号，两个连续的星号 ** 则是幂号。

当一个变量名称出现在操作数的位置，则在执行计算前，变量的数值将会取代操作数

加法、减法、乘法及乘幂的结果都与你的预期一致，不过除法的结果可能会让你讶异。下列运算会呈现出非预期的结果：

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

变量 minute 的数值为 59，59 除以 60 应该是 0.98333，而不是 0。会有这种偏差的原因是 Python 在此例中使用了**整数除法**。

当两个操作数都是整数时，运算结果也必定是整数，按照惯例，整数除法总是会*舍去*小数点后的数值，即使在类似此例的情况下，运算结果非常接近下一位整数时亦然。

这个问题的可能解决方法是使用百分比计算，而不是用分数计算：

```
>>> minute*100/60
98
```

计算结果又再一次舍去小数点后的数值，但至少近似于正确答案。另一种方法是利用浮点数除法。我们将会在第四章学到如何将整数及变量转换为浮点数。

2.7 运算的次序

当表达式中有一个以上的运算子时，则求值次序由**优先规则**决定。Python 对数学运算子采取和数学中相同的优先规则。PEMDAS 这个缩写字是记住运算顺序的好方法：

1. 小括号（**P**arentheses）具有最高优先权，而且可以用来强迫表达式依照你所希望的顺序求值。因为小括号内的表达会先求值，所以 2 * (3-1) 是 4，(1+1)**(5-2) 则是 8。你也可以用小括号使表达式更容易阅读，就像 (minute * 100) / 60 一样，结果并不会改变。
2. 乘幂（**E**xponentiation）具有次高的优先权，所以 2**1+1 是 3，不是 4，而 3*1**3 也是 3，并不是 27。
3. 乘法（**M**ultiplication）和除法（**D**ivision）的优先权相同，高于加法（**A**ddition）与减法（**S**ubtraction），而加法与减法的优先权亦相等。所以 2*3-1 产生 5 而不是 4，2/3-1 则是 -1 而非 1（记得 2/3=0 是整数除法）。
4. 具有相同优先权的运算子则从左到右进行计算。所以在 minute*100/60 这个表达式中，乘法先做，于是产生 5900/60，然后得到 98 的结果。如果这个运算是由右至左求值，结果会是 59*1，然后得到 59 这个错误的答案。

2.8 字符串的运算

一般来说，你不能够执行字符串的数学运算，尽管字符串看起来像数字。下列皆为不合法的范例（假设 message 的型态是字符串）：

```
message-1   "Hello"/123   message*"Hello"   "15"+2
```

有趣的是，+ 运算子可以在字符串上运作，虽然它并非完全依照你的期待运作。在字符串中，+ 运算子代表的是**连接**，也就是说是将两个操作数首尾连接组合起来。例如：

```
fruit = "banana"
baked_good = " nut bread"
print fruit + baked_good
```

这个程序会输出 banana nut bread。在 nut 之前的空格是字符串的一个必需部份，它可以在连接字符串之间制造出一个空格。

* 运算子也可用于字符串，它可以重复字符串。举例来说，'Fun'*3 的结果是'FunFunFun'。其中一个操作数必须是字符串，说另一个则必须为整数。

就某方面来说，我们可以将 + 及 * 的意义模拟为加法和乘法。就如同 4*3 等于 4+4+4，我们预期 "Fun"*3 与 "Fun"+"Fun"+"Fun" 一样，而结果也是如此。从另一方面来说，字符串的连接和重复与整数加法和乘法有一个极大的不同之处。你能够想出一个加法和乘法的性质是字符串连接和重复所没有的吗？

2.9 输入

Python 有两个内建函数可以取得键盘输入的数据：

```
n = raw_input("请输入您的姓名： ")
print n
n = input("请输入一个数字的表达式：")
print n
```

这个脚本的执行实例看起来如下：

```
$ python tryinput.py
请输入您的姓名：嬴政，中国史上第一个皇帝
嬴政，中国史上第一个皇帝
请输入一个数字的表达式：7 * 3
21
```

这两个函数都可以在小括号内输入*提示文字*。

2.10 组合

到目前为止，我们已经分别看过程序的组成要素---变量、表达式以及陈述---却尚未讨论如何组合这些要素。

程序语言最有用的特色之一就是它们**组合**小型基础组件的能力。举例来说，我们知道如何将数字相加，也知道如何印出数值，结果证明我们可以同时执行这两件事：

```
>>>  print 17 + 3
20
```

实际上，加法必须发生在打印之前，所以两者并不是真的发生于同一时间。重点是任何包含数字、字符串及变量的表达式可以用在 print 陈述中。你已经见过这样的范例：

```
print "午夜过了几分钟：", hour*60+minute
```

你也可以将任何表达式放置在指派陈述的右手边：

```
percentage = (minute * 100) / 60
```

这个能力现在看起来也许毫不起眼，不过你以后可以在其它范例中看到组合的能力可以用利落且精确的方式表示复杂的运算。

警告：这种表达式的使用位置也有一些限制。例如，指派陈述的左手边必须是*变量*名称，不能是表达式。所以，像 minute+1 = hour 就是不合法的。

2.11 批注

当程序逐渐变得庞大且复杂的时候，也变得难以阅读。形式语言繁复难解，要观察一段程序代码，并瞭解其功能和目的经常是很困难的。

正因如此，在你的程序中加入注记，用自然语言解释程序的功能是个好主意。这样的笔记称为**批注（comment）**，并用井字号 # 作为标记：

```
# 计算一小时中流逝的时间百分比
percentage = (minute * 100) / 60
```

在上面的例子中，批注单独成行。你也可以将批注放在一行的结尾处：

```
percentage = (minute * 100) / 60      # 警告：整数除法
```

从 # 到行尾的全部内容都会被忽略---它对程序没有任何影响。这些讯息是为了程序设计师或未来可能使用这段程序代码的程序设计师而准备的。在上述范例中，它提醒读者整数除法总是出人意表的行为。

2.12 术语

数值（value）：

可以储存在变量中或可以在表达式中计算的数字或字符串（或稍后会列出的其它事物）。

型态（type）：

数值的集合。数值的型态决定它在表示式中的使用方式。目前你已经见过的型态有整数（int）、浮点数（float）以及字符串（string）。

整数（int）：

Python 的一种数据型态，包括正值与负值的整数。

字符串（str）：

Python 的一种数据型态，包含一连串的字符。

浮点数（float）：

Python 储存*浮点*数的一种数据型态。浮点数分成两个部份储存在这个型态内部：*基底（base）*和*指数（exponent）*。当浮点数以标准格式打印时，看起来就像一般的小数。你使用浮点数时要注意舍去的错误，并记住它们仅是近似值。

变数（variable）：

一个指涉某个数值的名称。

指派陈述（assignment statement）：

指派数值到某个名称（变量）的陈述。在指派运算子（=）左边的是名称。指派运算子右边则是经 Python 直译器计算并指派到该名称的表达式，指派陈述左右两边的不同之处常常使初出茅庐的程序设计师混淆。在下列的指派中：

```
n = n + 1
```

n 在 = 的左右两边扮演了非常不同的角色。在右边的 n 是一个*数值*，并且是*表达式*的一部份，Python 直译器会先计算这个表达式，再将其指派到左边的名称。

指派运算子（assignment operator）：

Python 的指派运算子是 =，别将它和使用相同符号的数学比较运算子搞混了。

状态图（state diagram）：

一组变量与其对应的数值之图形化呈现。

变量名称（variable name）：

给予一个变量的名称。在 Python 里，变量名称由字母（a…z、A…Z 和 _）和数字（0…9）序列组成，开头必须是字母。在较佳的程序设计实务中，变量名称的选择应该要可以描述它们在程序中的功能，让程序可以*自行记录*。

关键词（keyword）：

编译器用来剖析程序的保留字，你不能使用 if、def 或 while 等关键词作为变量名称。

陈述（statement）：

Python 直译器可以执行的指令。陈述的范例包含指派陈述以及 print 陈述。

表达式（expression）：

用以表示单一数值数结果的变量、运算子及数值组合。

求值（evaluate）：

执行计算简化表达式以产生单一数值。

运算子（operator）：

用来表示简单计算，如加法、乘法或是字符串连接的特殊符号。

操作数（operand）：

运算子计算的数值之一。

整数除法（integer division）：

两个整数相除，所得的商也是整数。整数除法会产出分子可被分母分割的完整次数，余数则被舍去。

优先规则（rules of precedence）：

在包含多个运算子及操作数的表达式中，用来管理计算顺序的规则。

连接（concatenate）：

将两个操作数首尾相连。

组合（composition）：

将简单表达式及陈述组合成复合陈述及表达式的能力，它可以精准地表示复杂的计算。

批注（comment）：

在程序中意图让其它程序设计者（或任何原始码的读者）知道的讯息，它对程序的执行没有任何影响。

2.13 练习

1. 请记下当你打印一个指派陈述时，发生什么事：

```
>>> print n = 7
```

那么这一个呢？

```
  >>> print 7 + 5
```

或是这个呢？

```
  >>> print 5.2, "this", 4 - 2, "that", 5/2.0
```

你能够想到一个符合 print 陈述的普遍规则吗？执行 print 陈述会传回什么结果呢？

2. 将下列例句的每个字储存在不同的变量中，然后将句子打印到同一行：All work and no play makes Jack a dull boy.。

3. 将 6 * 1 - 2 这个表达式中加入小括号，使结果从 4 变成 -6。

4. 在以前可执行的程序代码中，将批注加至其中一行的行首，并记录再次执行时发生什么事。

5. input 与 raw_input 的不同，在于 input 会求取输入字符串的值，而 raw_input 则否。在直译器中尝试下列程序代码，并记录发生什么事：

```
>>> x = input()
3.14
>>> type(x)
```

```
>>> x = raw_input()
3.14
>>> type(x)

>>> x = input()
'The knights who say "ni!"'
>>> x
```

如果你拿掉上述范例的引号并尝试执行，会发生什么事？

```
>>> x = input()
The knights who say "ni!"
>>> x

>>> x = raw_input()
'The knights who say "ni!"'
>>> x
```

请试着描述并解释每一个结果。

6. 启动 Python 直译器，并在提示符号后输入 bruce + 4。你会看到一个错误讯息：

```
NameError: name 'bruce' is not defined
```

指派一个数值到 bruce，使 bruce + 4 等于 10。

7. 写一个名为 madlib.py 的程序（Python 脚本），要求使用者输入一系列的名词、动词、形容词、副词、复数名词和过去式动词等，然后产生一个语法正确，但是语意滑稽的文章段落（可参考 http://madlibs.org）。

3 函数

3.1 函数定义及用法

在程序设计的范畴，**函数**是个有名称的陈述序列，用来执行所需的运算。这个运算在**函数定义**中指定。在 Python 里，函数定义的语法为：

```
def NAME( LIST OF PARAMETERS ):
    STATEMENTS
```

你可以为你建立的函数取任意名称，但是你不能使用与 Python 关键词相同的名称。其中，LIST OF PARAMETERS （参数列表）用来指定使用新的函数前，你是否必须提供任何信息。

函数里的陈述数量没有限制，但是这些陈述必须在 def 关键词之后缩排。在这本书的范例之中，我们都将使用缩排四个空格的标准方式。函数定义是我们将会看到的**复合陈述**之一，它们全都有相同的形态：

1. 一个**标头**，以关键词作为起始，冒号作为结束。
2. 一个**主体**，由一个或多个 Python 陈述组成，每一个陈述都须从标头缩排*四个空格* -- 这是 Python 的标准缩排。

在函数定义之中，标头中的关键词是 def，接着是函数名称以及在括号内的*参数*列表。参数列表可以是空的，也可以包含任意数量的参数。不论何者，括号都是必需的。

我们一开始撰写的两个函数并不包含参数，因此语法如下：

```
def new_line():
    print           # 一个没有自变量的 print 陈述会印出一个空行
```

这个函数的名称是 new_line。空的括号表示不包含任何参数。它的主体只包含单一陈述，用来输出一个新行符号。（这就是你使用一个不带有自变量的 print 命令时会发生的事。）

定义新的函数并不会执行它。我们必须使用**函数呼叫**的方式执行函数。函数呼叫包含执行函数的名称，后头接着一连串的值，称为*自变量（arguments）*，自变量会被指派给函数定义里的参数。我们第一个例子并没有参数，所以函数呼叫并没有接收任何自变量。然而，必须注意的是，*函数名称中必需有括号*：

```
print "First Line."
new_line()
```

```
print "Second Line."
```

这个程序的输出结果如下：

```
First line.

Second line.
```

在两行中的空白行就是 new_line 函数呼叫的执行结果。如果我们想要在两行中间加入更多空白行呢？我们可以重复呼叫相同的函数：

```
print "First Line."
new_line()
new_line()
new_line()
print "Second Line."
```

或着我们可以写一个叫做 three_lines 的新函数，可以印出三个空白行：

```
def three_lines():
    new_line()
    new_line()
    new_line()

print "First Line."
three_lines()
print "Second Line."
```

这个函数包含三个陈述，全都缩排四个空格。因为下一个陈述没有缩排，Python 知道它不属于函数的一部份。

这个程序有几点你应该要注意：

- 你能够重复呼叫相同程序。事实上，这种作法是很普遍而且有用的。
- 你也可以用一个函数去呼叫其它的函数，在这个例子中，three_lines 呼叫 new_line。

到目前为止，你可能还不是很清楚为什么要花工夫建立这些新函数，事实上，这有很多理由，不过这个例子中只说明了其中两个：

1. 建立新函数让你有机会命名一组陈述。函数可以将复杂的计算隐藏在单一指令后，并且用英文单字来代替神秘的程序代码，以达到简化程序的目的。
2. 建立新函数能够减少重复的程序代码，那程序更小。例如：印出连续九个空白行的快捷方式就是呼叫三次 three_lines 函数。

将前面部份的程序片段 储存在名为 tryme1.py 的脚本中，整个程序看起来如下：

```
def new_line():
    print

def three_lines():
```

```
    new_line()
    new_line()
    new_line()

print "First Line."
three_lines()
print "Second Line."
```

这个程序包含两个函数定义，new_line 及 three_lines。函数定义和其它陈述的执行方式一样，但其效果是建立新的函数。在呼叫函数前，函数中的陈述不会被执行，而函数定义也不会产出任何输出。

就如你所预期的，你必须执行函数前先建立函数。也就是说，函数定义必须在第一次呼叫前执行。

3.2 执行流程

为了确保函数在第一次使用前先定义，你就得要弄清楚陈述在其中执行的次序，这叫做**执行流程**。

程序的第一个陈述永远最先执行。程序由上到下一次执行一个陈述。

函数定义并不会改变执行流程，函数内的陈述直到函数被呼叫后才会执行。虽然你可以在函数内定义另一个函数，但这并不普遍，在这种情况下，外层的函数被呼叫后才会执行内层的函数定义。

函数呼叫就像在执行流程内绕道而行。它并非接着执行下一个陈述，执行的流程跳到被呼叫函数的第一行，执行该处所有的陈述，然后再回原处继续执行。

直到你想起函数可以呼叫另一个函数之前，这听起来够简单了。在执行某个函数之间，程序也许必须执行其它函数里的陈述，但是在执行这个新函数时，程序也许又必须执行另一个不同的函数！

幸好 Python 熟悉追踪程序执行的足迹，所以每当一个函数执行完，程序就会回到原函数中它被呼叫的地方继续执行。当执行到程序的结尾，程序就会终止。

这个悲惨故事后寓意是什么？当你读一个程序，不要从上读到下。而是要顺着执行流程。

3.3 参数、自变量以及 import 陈述

大多的函数都需要自变量，也就是用来控制函数如何进行工作的值。举例来说，如果你想要找到某数的绝对值，你就必须指出某数是多少。Python 有个计算绝对值的内建函数：

```
>>> abs(5)
5
>>> abs(-5)
5
```

在这个例子中，abs 函数的自变量分别是 5 及 -5。

有些函数可以接受一个以上的自变量，例如内建函数 pow 就需要两个自变量，底数与指数。在这个函数内，被指派至变量的数值称为**参数**。

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

另一个接受一个以上自变量的内建函数是 max。

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3*11, 5**3, 512-9, 1024**0)
503
```

内建函数 max 可以接受用逗号分开的任意数量自变量，并且传回所接收值中最大的值。自变量可以是简单的数值或是表达式，在最后的例子中，回传的结果是 503，因为它大于 33、125 及 1。

这里有个只拥有一个参数的自订函数范例：

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

这个函数接受单一**自变量**，并且指派给名为 bruce 的参数。参数的数值（在这里我们无法得知该值为何）会印出两次，并紧接着一个空白行。选择 bruce 这个名称是暗示你可以自行决定给予参数的名称，但一般来说，你会想用一个比 bruce 更有意义的名称。

交互式的 Python shell 提供一个简便的方式测试我们的函数。我们可以使用 **import 陈述**将我们在脚本中定义的函数汇入直译式程序中。如果要看这个功能如何运作，我们先假设 print_twice 函数被定义在一个名为 chap03.py 的脚本中。我们现在可以将它*汇入*到我们的 Python shell 程序中，以互动的方式测试。

```
>>> from chap03 import *
>>> print_twice('Spam')
Spam Spam
>>> print_twice(5)
5 5
>>> print_twice(3.14159)
3.14159 3.14159
```

在一个函数呼叫中，自变量的数值会被指派到函数定义里对应的参数。 从实际的功能来看，如果 bruce = 'Spam' 在 print_twice('Spam') 被呼叫时执行，在 print_twice(5) 中 bruce = 5，而在 print_twice(3.14159) 中 bruce = 3.14159。

任何可以被印出的自变量形态都可以传送至 print_twice 函数中，在第一个函数呼叫中，自变量是字符串。在第二个函数呼叫中是整数。而 I 第三个则是浮点数。

和内建函数一样，我们可以在 print_twice 函数中使用表达式：

```
>>> print_twice('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
```

'Spam'*4 会先求出 'SpamSpamSpamSpam' 的值，接着传送至 print_twice 作为自变量。

3.4 组合

就像数学的函数一样，Python 的函数可以**组合**，这意思是说你可以使用一个函数的结果作为另一个函数的输入值。

```
>>> print_twice(abs(-7))
7 7
>>> print_twice(max(3, 1, abs(-11), 7))
11 11
```

在第一个例子中，abs(-7) 的结果是 7，然后变成 print_twice 的自变量。第二个例子则存在两层的组合关系，abs(-11) 先求出 11 的值，接着在 max(3, 1, 11, 7) 中得到的值为 11 ，最后 print_twice(11) 显示结果。

我们也可以用变量作为自变量：

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

请注意这里有一件非常重要的事。我们当成自变量传送的变量名称（michael）和参数名称（bruce）完全无关。无论该值原来（在函数呼叫中）称为什么都无关紧要，在 print_twice 里，我们把每个值都称为 bruce。

3.5 区域的概念：变数及参数

当你在函数里建立一个**区域变量**，它就只存在这个函数里头，你不能用于这个函数以外的地方。例如：

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

这个函数接受两个自变量并连接它们，然后印出结果两次。我们能够用两个字符串来呼叫这个函数：

```
>>> chant1 = "Pie Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> cat_twice(chant1, chant2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

当 cat_twice 终止后，变数 cat 也随即消失。如果我们尝试印出这个变量，我们会得到错误讯息：

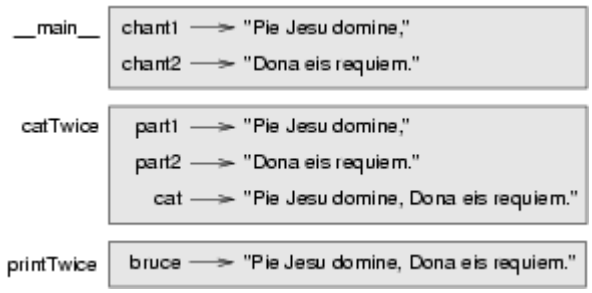
```
>>> print cat
NameError: name 'cat' is not defined
```

参数也具有区域性。举例来说，在函数 print_twice 的范围以外，并不存在 bruce 这个名称。如果你尝试使用它，Python 就会抱怨。

3.6 堆栈图

为了掌握变量可以用在何处，有时画出**堆栈图**非常有用。堆栈图就像状态图一样，标示出每个变量的数值，并同时显示每个变量所属的函数。

每个函数用一个**框架**表示。框架就是一个旁边标示函数名称的长方形，而其中标示出参数与变量。上述范例的堆栈图如下：



框架的次序表示执行流程。__main__ 是给予最顶层的函数的专有名称，它先呼叫 cat_twice，然后 cat_twice 再呼叫 print_twice。当你建立一个不属于任何函数的变量时，这个变量就属于 __main__。

每一个参数与其对应自变量指涉相同的数值。所以，part1 与 chant1 有相同的数值，part2 数值则与与 chant2 相同，而 bruce 与 cat 的数值亦相同。

如果函数呼叫时发生错误，Python 就会印出被呼叫函数的名称、和呼叫它的函数名称及*再上一层*的函数名称，一直回溯到最顶层函数名称为止。

如果想看实际执行状况，建立一个名为 tryme2.py 的 Python 脚本如下：

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

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

chant1 = "Pie Jesu domine, "
chant2 = "Dona eis requim."
cat_twice(chant1, chant2)
```

我们将 print cat 陈述加入 print_twice 函数，但是变量 cat 在该处并未定义。执行这个脚本会产生以下错误讯息：

```
Traceback (innermost last):
  File "tryme2.py", line 11, in <module>
    cat_twice(chant1, chant2)
  File "tryme2.py", line 7, in cat_twice
    print_twice(cat)
  File "tryme2.py", line 3, in print_twice
    print cat
NameError: global name 'cat' is not defined
```

这种函数列表称为**异常追踪（traceback）**。它告诉你错误在哪一个程序档案中发生，并指出错误发生时执行到哪一行及哪一个函数。它也会显示造成错误的程序代码。

要注意到异常追踪与堆栈图之间的相似度。这并不是一个巧合。事实上，异常追踪的另一个常用名称就是 *推迭追踪（stack trace）*。

3.7 术语

函数（function）：

一个有名称的陈述序列，用来执行一些有用的运算。函数可以接受参数或产出结果，而反之亦然。

函数定义（function definition）：

一个建立新函数的陈述，具体地指定新建函数的名称、参数以及执行何种陈述。

复合陈述（compound statement）：

由两个部份组成的陈述：

1. 标头 - 以决定陈述种类的关键词起始，并且以冒号作为结束。

2. 主体 - 包含一个或多个从标头缩排相同距离的陈述组成。

复合陈述的语法如下：

```
keyword expression :
    statement
    statement ...
```

标头（header）：

复合陈述的第一个部份。标头由关键词起始，并以冒号（:）结束。

主体（body）：

复合陈述的第二个部份，主体由一系列从标头起始处缩排相同距离的陈述组成。Python 社群所使用的标准缩排量是四个空格。

函数呼叫（function call）：

一个执行函数的陈述。它由函数名称和其后包含于括号中的自变量列表所组成。

执行流程（flow of execution）：

在程序运行时，陈述执行的顺序。

参数（parameter）：

一个函数中，用来代表自变量数值的名称。

汇入（import）：

一种陈述，可以将一个 Python 脚本中定义的函数和变量，带入另一个脚本或运行中的 Python shell 环境中。

举例来说，假设下列程序代码是一个名为 tryme.py 的脚本：

```
def print_thrice(thing):
    print thing, thing, thing


n = 42
s = "And now for something completely different..."
```

现在从 tryme.py 所在的目录下启动 Python shell：

```
$ ls
tryme.py  <and other stuff...>
$ python
>>>
```

在 tryme.py 中定义了三个名称，分别是： print_thrice、n 及 s。如果我们没有先汇入这些名称就尝试使用它们，我们会得到错误讯息：

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

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

不过，如果我们汇入 tryme.py 中所有程序代码，我们就能够使用所有定义于其中函数或变量。

```
>>> from tryme import *
>>> n
42
>>> s
'And now for something completely different...'
>>> print_thrice("Yipee!")
Yipee!Yipee!Yipee!>>>
```

请注意你在 import 陈述中，不需要包含 .py 扩展名。

自变量（argument）：

呼叫函数时提供给函数的值。这个值被指派到函数中的对应参数中。

函数组合（function composition:）：

使用一个函数呼叫的输出作为另一个函数的输入。

区域变量（local variable）：

一个函数中定义的变量，一个区域变量只能使用在所属函数中。

堆栈图（stack diagram）：

一种显示许多函数、函数中的变量和变量值的图像化表示法。

框架（frame）：

堆栈图中用来表示函数呼叫的长方形图标。它包含该函数的区域变量及参数。

异常追踪（traceback）：

当执行错误发生时，所列出的执行中函数列表。. 异常追踪亦常被称为 *堆栈追踪（stack trace）*，因为它会依照函数在 执行堆栈（runtime stack）中的顺序列出函数。

3.8 练习

1. 使用文字编辑器建立一个名为 tryme3.py 的脚本。写一个利用 three_lines 印出九个空白行的 nine_lines 函数。然后再建立一个名为 clear_screen 的函数，可以印出二十五个空白行。你的程序的最后一行应该*呼叫* clear_screen 函数。

2. 将 tryme3.py 的最后一行移动到程序最上方，使函数 clear_screen 的*呼叫*出现在*函数定义*之前。执行这个程序，然后记下你得到的错误讯息。你能够说出一个规则，描述*函数定义*与*函数呼叫*在一个程序中出现位置的相互关系吗？

3. 拿一个可运作的 tryme3.py，将其中 new_line 的定义移到 three_lines 的定义之后。然后记下执行这个程序时发生什么事。现在再把 new_line 的定义移动 到three_lines()的呼叫之后。解释一下为什么这是你在上一个练习中描述规则的范例。

4. 将下面 cat_n_times *函数定义*的*主体*完成，让这个函数可以印出 s 字符串 n 次：

```
def cat_n_times(s, n):
    <fill in your code here>
```

将这个函数储存在名为 import_test.py 的脚本中。现在在 unix 提示列下，确定你在 import_test.py 所存放的目录下（ls 指令应该会显示 import_test.py）。启动 Python shell 然后尝试下列指令：

```
>>> from import_test import *
>>> cat_n_times('Spam', 7)
SpamSpamSpamSpamSpamSpamSpam
```

如果一切顺利，你执行的结果应该如上所示。尝用用其它的值呼叫 cat_n_times 函数，到你熟悉它如何运作为止。

4 条件式

4.1 余数运算子

余数运算子运作在整数（及整数表达式）中，并在第一个操作数除以第二个操作数时，得出余数。在 Python 里，余数运算子用百分比符号（%）表示。语法则与其它运算子相同：

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

所以 7 除以 3 得到的商是 2，余数为 1。

余数运算子的运算结果出人意料地有用。例如，你可以用来检查一个数是否能被另一个数整除---如果 x % y 的结果是 0，x 就能被 y 整除。

你也可以用它抽取出一个数值中的最右侧的数字。例如，x % 10 可得出 x 最右侧的数字（在十进制时）。同样地，而 x % 100 则会产生最右侧的两位数字。

4.2 布尔值与表达式

用来存放真假值的 Python 型态称为 bool，依英国数学家 George Boole 而命名。George Boole 创造了*布尔代数*，这是所有现代计算机计算的基础。

布尔值只有两种：True 与 False。大写在此非常重要，因为 true 和 false 并非布尔值。

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

布尔表达式（boolean expression）是用来求取布尔值的表达式。== 运算子会比较两个数值，然后产生一个布尔值：

```
>>> 5 == 5
True
```

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

第一个陈述中两个操作数是相等的，所以表达式的结果为 True；而在第二个陈述中，5 与 6 并不相等，所以我们得到 False。

== 运算子是**比较运算子**之一，其它还有：

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

虽然你可能熟悉这些运算，Python 符号却与数学符号不大相同。一个常见错误就是使用单一等号（=）而非两个连续等号（==），请记得 = 是指派运算子，而 == 则是比较运算子。同样的，=< 或 => 这种符号亦是错误的。

4.3 逻辑运算子

逻辑运算子有三个，分别是 and、or 和 not。这些运算子的意义与其英文意思相似。例如，只有在 x 大于 0 *而且*小于 10 的时候，x > 0 and x < 10 才成立。

在 n % 2 == 0 or n % 3 == 0 中的两个条件之一成立时，其表达式即成立，也就是 n 可以被 2 整除*或是*被 3 整除。

最后，not 运算子则用来否定一个布尔表达式，也就是说 (x > y) 不成立时，not(x > y) 即成立，亦即 x 小于或等于 y。

4.4 条件执行

为了撰写有用的程序，我们几乎永远需要检查条件，并依其改变程序行为的能力。**条件陈述**赋予我们这种能力。其中最简单的型式就是 **if 陈述**：

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

在 if 陈述之后的布尔表达式就称为**条件**。如果该条件成立，其后缩排的陈述就会执行；反之则否。

if 陈述的语法如下：

```
if 布尔表达式：
    陈述
```

和上一章的函数定义及其它复合陈述一样，if 陈述包含一个标头和一个主体。标头以关键词 if 开始，后接 *布尔表达式*，最后则以冒号（:）结束。

其后缩排的陈述称为**区块**，而第一个没有缩排的陈述则标示了该区块的结尾。复合陈述中的陈述区块称为陈述**主体**。

如果布尔表达式求出的值为 True，每个在主体里的陈述就会依序执行。而若布尔表达式求出的值为 False 时，就会跳过整个区块。

if 陈述不限制出现在主体中的陈述个数，但是至少要有一个。有时主体不放陈述是很有用的（通常是为了空出尚未写好的程序代码位置）。在这种情况下，你可以利用 pass 陈述，它不会执行任何动作。

```
if True:           # 这永远会是真
    pass           # 所以这也永远会被执行，但不会执行任何动作
```

4.5 替代执行

if 陈述的第二个型态为替代执行，在这个型态中有两个可能性，条件的成立与否则选择要执行哪一个可能性。语法如下：

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

如果 x 除以 2 的余数是 0，我们就知道 x 是偶数，于是程序显示相应的结果。而若该式不成立，则执行第二组陈述。既然条件非真即假，那一定会执行两组陈述之一。这两组陈述称为**分流（branches）**，因为它们是从执行流程分化出来的。

此外，如果你时常需要检查数值的奇偶性（检查数值为偶数或是奇数），你可以把这段程序代码包进函数里：

```
def print_parity(x):
    if x % 2 == 0:
        print x, "is even"
    else:
        print x, "is odd"
```

不论 x 数值为何，print_parity 都会显示对应的讯息。当你呼叫这个函数，你可以用任何整数式作为自变量。

```
>>> print_parity(17)
17 is odd.
>>> y = 41
>>> print_parity(y+1)
```

```
42 is even.
```

4.6 炼状条件式

有时候会有多于两种可能性，因此我们需要两个以上的分流。用来表示这种计算的方式之一是使用**炼状条件式**：

```
if x < y:
    print x, "is less than", y
elif x > y:
    print x, "is greater than", y
else:
    print x, "and", y, "are equal"
```

elif 是 else if 的缩写，同样地，炼状条件式一定只会执行一个分流。elif 的使用数量并没有限制，但是只能有一个 else 陈述，并且需为陈述中最后一个分流。

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
elif choice == 'c':
    function_c()
else:
    print "Invalid choice."
```

Python 会依序检查每个修件。如果第一个条件不成立，则检查第二个，依此类推。如果其中一个条件成立，就执行对应的分流，然后这个陈述就结束了。就算有多个条件成立，但是只会执行第一个成立的分流。

4.7 巢状条件式

一个条件式可以**巢居**在另一个条件式中。我们可以写个三分法范例如下：

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

外层的条件式包含了两个分流，第一个分流包括一个简单的输出陈述。第二个分流则内含另一个有两个分流的 if 陈述。这里的两个分流皆为输出陈述，不过它们也可以用条件陈述取代，为你的程序加入另一层条件。

虽然陈述缩排的方式让结构显而易见，巢状条件式还是难以迅速解读，一般来说，最好尽可能避免使用它。

逻辑运算子通常可以简化巢状条件式。例如，我们可以用单一条件式改写以下程序代码：

```
if 0 < x:
    if x < 10:
        print "x is a positive single digit."
```

只有在两个条件式都成立的时候，print 陈述才会被执行，所以我们可以运用 and 运算子：

```
if 0 < x and x < 10:
    print "x is a positive single digit."
```

这种条件式非常普遍，因此 Python 提供了另一种与数学表示法相似的语法：

```
if 0 < x < 10:
    print "x is a positive single digit."
```

这个条件式的语意，跟复合布尔表达式和巢状条件式相同。

4.8 return 陈述

return 陈述让你可以在抵达函数结尾前，停止函数的执行。当你侦测到错误状况时，你可以利用它：

```
def print_square_root(x):
    if x <= 0:
        print "Positive numbers only, please."
        return

    result = x**0.5
    print "The square root of x is", result
```

上述 print_square_root 函数有个名为 x 参数，这个函数所做的第一件事就是检查 x 是否小于 0 或等于 0，如果条件成立，则会显示一个错误讯息，并使用 return 退出这个函数。执行流程立刻回到原本呼叫的地方，而函数中其余的程序代码不会被执行。

4.9 键盘输入

在第二章中，我们学到了可取得键盘输入信息的 Python 内建函数：raw_input 及 input。现在让我们进一步看看这两个函数。

当呼叫这两个函数之一时，程序就会暂停，并等待使用者输入一些讯息。当使用者按下 Return 键或 Enter 键时，程序就会再度执行，raw_input 则将使用者输入的数据当作字符串传回：

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

在呼叫 raw_input 之前，最好先印出一个讯息，告诉使用者需输入什么信息。这样的讯息就称为**提示**。我们可以利用提示作为 raw_input 的自变量：

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

请注意提示属于字符串，所以必须用引号包起来。

如果预期的响应为整数，就可以使用 input 函数，它会将响应当成 Python 表达式运算：

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

如果使用者输入一串数字，就会转换成整数，并且指派给 speed 变量。但是，如果使用者输入不符合 Python 表达式的文字，程序就会当掉：

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?What do you mean, an African or a European swallow?...
SyntaxError: invalid syntax
```

在上述最后一个例子中，如果使用者将响应包在引号中，使其符合 Python 表达式，则不会有错误讯息。

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?"What do you mean, an African or a European swallow?"
>>> speed
'What do you mean, an African or a European swallow?'
>>>
```

为了避免此种错误，最好使用 raw_input 取得一个字符串，然后再使用转换指令转成其它型态。

4.10 型态转换

每一种 Python 型态都有对应的内建指令，可以将其它型态的数值转换成该种型态。例如，int(自变量) 能够尝试将任何数值转换成整数，如果不能转换，则显示错误讯息：

```
>>> int("32")
32
```

```
>>> int("Hello")
ValueError: invalid literal for int() with base 10: 'Hello'
```

int 也可以将浮点数转换成整数，不过请记得小数点后的部份都会省去：

```
>>> int(-2.3)
-2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

float(自变量) 指令则可以将整数和字符串转换为浮点数：

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

Python 将整数 1 和浮点数 1.0 当成不同的数字，这对使用者来说可能有些奇怪。他们可能表示相同的数字，但却属于不同的型态。这是因为计算机对整数及浮点数采取不同表示方法的缘故。

str(自变量) 指令可以将任何自变量转换成字符串型态：

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

str(自变量) 可以用在任何数值，并将其转换成字符串。如同之前所提，True 是个布尔值，而 true 则不是。

就布尔值而言，情况特别有意思：

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool("Ni!")
True
>>> bool("")
```

```
False
>>> bool(3.14159)
True
>>> bool(0.0)
False
```

Python 对任何型态的数值都会指定一个布尔值。对数字型态，如整数与浮点数，来说，0 是假，非 0 就是真。而就字符串型态来说，空字符串为假，非空字符串则为真。

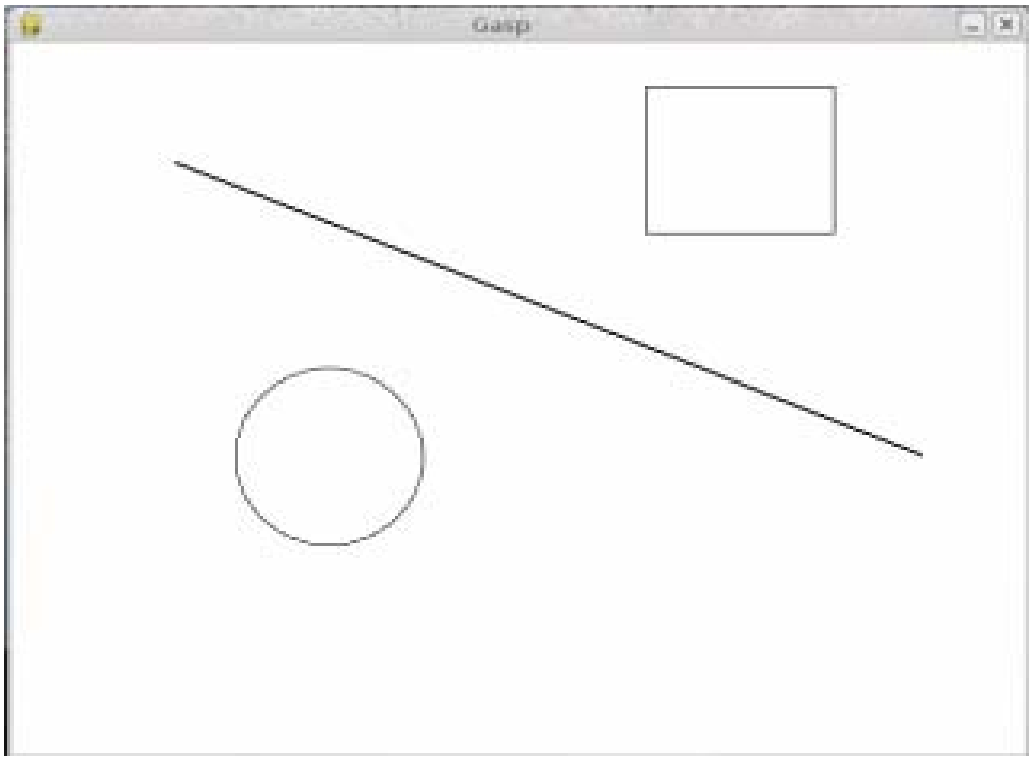
4.11 Gasp

GASP（**G**raphics **A**PI for **S**tudents of **P**ython，中文是「给学生的 Python 应用程序图形接口」）可以让我们撰写包含图形的程序。

要启动 gasp，请试试下列程序代码：

```
>>> from gasp import *
>>> begin_graphics()
>>> Circle((200, 200), 60)
Circle instance at (200, 200) with radius 60
>>> Line((100, 400), (580, 200))
Line instance from (100, 400) to (590, 250)
>>> Box((400, 350), 120, 100)
Box instance at (400, 350) with width 120 and height 100
>>> end_graphics()
>>>
```

最后一个陈述关闭绘图帆布前，你应该可以看到一个图形窗口如下：



从现在开始，我们将会用 GASP 来勾勒（双关）计算机程序设计的概念，并增加学习的乐趣。

4.12 术语

余数运算子（modulus operator）：

使用百分比符号（%）表示的运算子，它可以使用在整数中，并取得一个数值除以另一个数所得的余数。

布尔值（boolean value）：

布尔值只有两种：True 及 False。Python 直译器运算布尔表达式时，即产生布尔值。布尔值的型态名称为 bool。

布尔表达式（boolean expression）：

一种表达式，求出的值非真即假。

比较运算子（comparison operator）：

可用来比较数值的运算子，包括 ==、!=、>、<、>= 和 <=。

逻辑运算子（logical operator）：

用来结合布尔表达式的运算子，包含 and、or 及 not。

条件陈述（conditional statement）：

依据条件决定执行流程的陈述。在 Python 中，if、elif 和 else 关键词被用来表示条件陈述。

条件（condition）：

在条件陈述中可以决定执行哪个分流的布尔表达式。

区块（block）：

一组有相同缩排的连续陈述。

主体（body）：

在复合陈述中，紧接着标头的陈述区块。

分流（branch）：

由条件执行所决定的执行流程可能路径之一。

炼状条件式（chained conditional）：

一种条件式分流，其中包含多于两种可能的执行流程。在 Python 中，炼状条件式是由 if ... elif ... else 陈述构成。

巢状（nesting）：

程序结构中包含着另一个结构，就像条件式陈述里的分流还包含另一个条件式陈述。

提示（prompt）：

告诉使用者输入数据的视觉线索。

型态转换（type conversion）：

一种明确的陈述，可以取得一种型态的数值，并得出另一种型态的对应数值。

4.13 练习

1. 试着在心中计算下列数字表示式，然后用 Python 直译器检查结果：

a) >>> 5 % 2
b) >>> 9 % 5
c) >>> 15 % 12
d) >>> 12 % 15
e) >>> 6 % 6
f) >>> 0 % 7
g) >>> 7 % 0

最后一个例子发生了什么事情？为甚么？如果你可以正确预期最后一个范例之外的计算机响应，请继续其它练习。如果不行，现在请花点时间想出你自己的范例。请继续研究余数运算子，直到你有自信完全瞭解它如何运作为止。

2.

```
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"
```

将这些程序代码*包*进 compare(x, y) 函数中，呼叫 compare 三次：并分别使第一个自变量小于、大于和等于第二个自变量。

3. 要进一步了解布尔表达式，建构真值表非常有用。两个布尔表达式只有在拥有相同的真值表时，*才会在逻辑上相等*。

下列 Python 脚本会以 p 和 q 两个变数，印出任何布尔表达式的真值表。

```
expression = raw_input("Enter a boolean expression in two variables, p and q:
")

print " p      q      %s"  % expression
length = len( " p      q      %s"  % expression)
print length*"="

for p in True, False:
    for q in True, False:
        print "%-7s %-7s %-7s" % (p, q, eval(expression))
```

你将会在下一章中学到这个脚本如何运作。现在，你则是要用它来学习布尔表达式。拷贝这些程序代码到名为 p_and_q.py 的档案中，在命令列中执行，并在提示输入布尔表达式时，键入 p or q。你应该会得到如下的结果：

```
 p      q      p or q
====================
True    True    True
True    False   True
False   True    True
False   False   False
```

现在我们知道它如何运作了，请将它包进一个函数中，使其更容易使用：

```
def truth_table(expression):
    print " p      q      %s"  % expression
    length = len( " p      q      %s"  % expression)
    print length*"="
```

```
        for p in True, False:
            for q in True, False:
                print "%-7s %-7s %-7s" % (p, q, eval(expression))
```

我们可以将它汇入 Python shell 中，并利用一个包含上述 p or q 布尔表达式的字符串作为自变量，呼叫 truth_table 函数：

```
>>> from p_and_q import *
>>> truth_table("p or q")
 p      q      p or q
====================
True    True    True
True    False   True
False   True    True
False   False   False
>>>
```

以下列布尔表达式作为自变量，使用 truth_table 函数，并记下每次产生的真值表：

a. not(p or q)
b. p and q
c. not(p and q)
d. not(p) or not(q)
e. not(p) and not(q)

哪些布尔表达式在逻辑上相等？

4. 将下列的表达式依序输入 Python shell：

```
True or False
True and False
not(False) and True
True or 7
False or 7
True and 0
False or 8
"happy" and "sad"
"happy" or "sad"
"" and "sad"
"happy" and ""
```

分析这些结果。对于这些不同型态的数值和逻辑运算子，你可以观查到什么样的结果？你是否可以将观察到的结果写成简单的*规则*，以表示 and 及 or 表达式？

5.

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
```

```
elif choice == 'c':
    function_c()
else:
    print "Invalid choice."
```

将这些程序代码包进一个名为 dispatch(choice) 的函数。然后定义 function_a、function_b 和 function_c，让它们可以印出表示它们被呼叫的讯息。例如：

```
def function_a():
    print "function_a was called..."
```

将四个函数（dispatch、function_a、function_b 和 function_c）放进名为 ch4prob4.py 的脚本中。在脚本的底部，加入一个 dispatch('b') 呼叫，你的执行结果应如下：

```
function_b was called...
```

最后，修改这个脚本让使用者能够输入「a」或「b」或「c」。然后将其汇入 Python Shell 测试。

6. 撰写一个名为 is_divisible_by_3 的函数，用单一整数作为自变量，如果该数值可以被 3 整除，印出「This number is divisible by three.」，反之则印出「This number is not divisible by three.」。

现在再写一个名为 is_divisible_by_5 的类似函数。

7. 扩展上一个练习中你所写的函数，让它可以用两个整数当作自变量，然后印出第一个数是否可被第二个数整除，并将这个函数命名为 is_divisible_by_n(x, n)。将其储存成一个名为 ch04e06.py 的档案。然后汇入 Python Shell 测试。你的程序应该看起来如下：

```
>>> from ch04e06 import *
>>> is_divisible_by_n(20, 4)
Yes, 20 is divisible by 4
>>> is_divisible_by_n(21, 8)
No, 21 is not divisible by 8
```

8. 下列程序代码的输出为何？

```
if "Ni!":
    print 'We are the Knights who say, "Ni!"'
else:
    print "Stop it!No more of this!"

if 0:
    print "And now for something completely different..."
else:
    print "What's all this, then?"
```

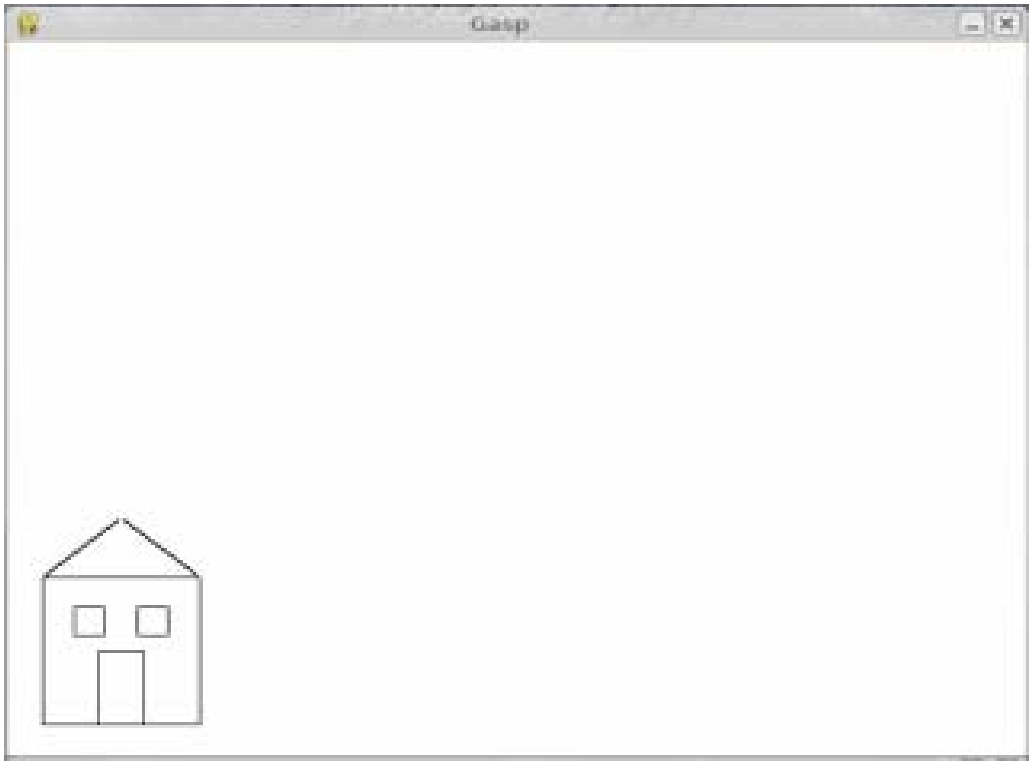
请解释会发生什么事及其原因。

9. 下面的 GASP 脚本名为 house.py，可以在 GASP 帆布上画出简单的房子：

```
from gasp import *              # 从 gasp 函数库中引入所有功能
```

```
begin_graphics()                  # 打开绘图帆布

Box((20, 20), 100, 100)           # 房子
Box((55, 20), 30, 50)             # 门
Box((40, 80), 20, 20)             # 左边的窗户
Box((80, 80), 20, 20)             # 右边的窗户
Line((20, 120), (70, 160))        # 左边的屋顶
Line((70, 160), (120, 120))       # 右边的屋顶

pause()                           # 保持帆布直到按下某个按键
end_graphics()                    # 关闭帆布（这样直接会关闭，
                                  # 虽然程序到这里就结束了，但最好还是
                                  # 明确表示）
```

- 执行这个脚本，并确定你得到一个看起来如下的窗口：



- 将这些程序代码包进一个名为 draw_house() 的函数里。
- 现在执行这个脚本。你看到房子了吗？为甚么没有？
- 增加一个 draw_house() 呼叫在脚本底部，让房子回到屏幕上。
- 为这个函数*制定* x 及 y 两个参数 -- 标头应该变成 def draw_house(x, y):，让你可以输入房子在帆布上的位置。
- 利用 draw_house 在帆布上不同的地方画出五间房子。

5 多效函数

5.1 传回值

我们已经用过的内建函数，如 abs、pow 和 max，都会产生结果。这些函数每个在被呼叫时都会产生一个数值，我们通常会将这些数值指派给一个变量，或是作为表达式的一部份。

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

但是到目前为止，我们自己写的函数都还没传回一个数值。

在这一章里头，我们要写些会传回数值的函数，我们将这些函数称之为**多效函数（fruitful functions）**，因为没有更好的名字。第一个例子就是 area 函数，它会根据所给予的半径算出圆面积：

```
def area(radius):
    temp = 3.14159 * radius**2
    return temp
```

我们之前已经见过 return 陈述，但是在多效函数中 return 陈述还包含着**传回值（return value）**。这就是说：「立即由这个函数传回，并用其后的表达式当作传回值。」所提供的表达式要多复杂都可以，于是我们就能够把上面的函数写得更简洁：

```
def area(radius):
    return 3.14159 * radius**2
```

但以另一方面来讲，像 temp 这种 **暂存变量（temporary variables）**常常可以让除错更容易些。

有时在条件句中的每个分流都使用一个 return 陈述是非常有用的。我们已经见过内建的 abs 函数，现在我们来看看如何撰写自己的函数：

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

既然这些 return 位于替代条件句中，那就只有一个 return 陈述会被执行。一旦执行了某个 return 陈述，函数随即终止，之后的陈述都不会被执行。

另外一个撰写上述函数的方法是省去 else，仅在 if 条件后接上第二个 return 陈述。

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

好好想一想这个版本，并让自己确信它和第一个版本的功能相同。

程序代码如果出现在 return 陈述后面，或是任何执行流向永远不会到达的地方，就称为**死码（dead code）**。

在一个多效函数中，最好能确认程序中每个可能的路径都有一个 return 陈述。下列版本的 absolute_value 函数就没有做到：

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

这个版本不正确是因为当 x 刚好为 0 时，两个条件都不会为真，因此函数就会在没有遇到一个 return 陈述的情况下结束。在这个情况下，传回值就会是一个称为 None 的特别值：

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

None 是 NoneType 这个型态中唯一的数值：

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

只要 Python 的函数没有传回任何其它的数值，就会传回 None。

5.2 程序开发

在这个时候，你应该已经可以观察完整的函数，并分办它们是做什么的。同样的，如果你一直都有做练习，你就已经写了一些小型函数。当你开始写较为大型的函数时，你可能会开始遇到更多困难，特别是在执行错误和语意错误部分。

为了处理日益复杂的程序，我们建议一个名为**渐进式开发（incremental development）**的技术，渐进式开发的目标是藉由一次只加入和测试少量程序代码，避免冗长的除虫时间。

举例来说，假设你想要找到（x_1, y_1）及（x_2, y_2）两点间的距离，由勾股定理，距离如下：

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

一步便是考虑 Python 中的 distance 函数看起来应该像是怎样， 就是说，输入（参数）是什么，输出（传回值）又会是什么呢？

在这个例子中，两点坐标便是输入，我们可以用四个参数来表达。传回值便是距离，这会是浮点数型态的数值。

我们已经可以写出这个函数的雏型：

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

然这个版本的函数无法计算距离，它总是会传回数值 0。 而他的语法正确，可以执行，这就是说我们在使它更复杂之前，就可以测试它。

要测试这个新函数，我们用简单的数值来呼叫它：

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

我们用的两个坐标是水平距离 3，垂直距离 4，所以两点距离会是 5（这是 3-4-5 三角形的斜边），在测试函数时，知道正确的答案是很有用的。

这时我们已经确定目前的函数语法正确，然后我们就可以开始增加几行程序代码。每次渐进的改变后，我们都再次测试这个函数，如果在任何时候产生了错误，我们就会知道错误一定在哪里---就在我们所加入的最后一行程序代码。

计算上的第一步便是找到 $x_2 - x_1$ 及 $y_2 - y_1$ 的差，我们将这两个差储存到名为 dx 与 dy 的暂存变量中，然后印出来。

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

如果这个函数正确执行，输出应该会是 3 与 4。如果这样的话，我们就知道这个函数得到正确的参数并正确执行第一个计算，而结果不是这样的话，就只有几行程序代码需要检查。

接下来我们计算 dx 及 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
```

请注意我们移除了上一步所写的 print 陈述，这样的程序代码就叫做 **支架码（scaffolding）**，因为它有助于建立程序，但并不是最后成品的一部分。

们会在这个阶段再次执行这个程序，检查它的结果（应为 25）。

最后，我们用分数指数 0.5 找到平方根，我们可以计算并传回结果：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result
```

如果这运作正确，你也就完成这个距离函数。不然的话，你也许会想要在 result 陈述前印出结果的数值。

开始的时候，你应该一次只增加一两行程序代码，而当经验愈多时，你大概会发现你自己可以一次处理更多程序代码。不论如何，渐进式开发的过程可以节省你大量的除虫时间。

这个过程的关键观点如下：

1. 开始就用可以顺利执行的程序，然后渐进地改变它。在任何时候，只要发生了一个错误，你就可以清楚知道错误发生的地方。
2. 利用暂存变量储存计算过程的中间值，让你能够输出并检查这些数值。
3. 一旦程序可以顺利运作，你可以移除一些骨架码，或是把多个陈述融合成复合表达式，只要不会让程序难以阅读。

5.3 函数的组合

你现在应该已经想到，你可以在一个函数中呼叫另一个函数，这种能力就叫做**组合（composition）**。

我们会写个需要圆心与圆周上一点坐标的函数，然后计算圆的面积以作为范例。

假设圆心坐标储存在 xc 与 yc 两个变量中，圆周上一点的坐标则是储存在 xp 及 yp。第一步便是找到圆的半径，也就是两点间的距离。幸运的是，我们才刚写了 distance 函数，可以执行这个计算，所以这时我们要做就是使用它：

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

第二步 0 是利用半径找到圆的面积并传回，我们会再次使用到我们先前写好的函数：

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

将这些包进一个函数里，我们得到：

```
def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
```

```
    return result
```

我们称呼这个函数为 area2 来区别之前定义的 area ，在已知模块内函数只能有一个特定名称的函数。

暂存变量 radius 和 result 对开发和除虫非常有用，但一旦程序可以正确执行，我们可以组合函数呼叫，让程序更简洁：

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

5.4 布尔函数

布尔函数就是可以回传布尔值的函数，它通常便于在函数中隐藏复杂的测试，举例来说：

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

这个函数的名称叫做 is_divisible。常给予**布尔函数**的名称看起来就像是非题，is_divisible 回传的不是 True 就是 False ，藉此来指出 x 是或不是被 y 整除。

们可以利用 if 陈述本身就是布尔表达式的情况，使程序看起来更简洁， 我们可以直接传回这个函数，完全舍弃 if 陈述：

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

这部份显示了这个新函数运作的样子：

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

布尔函数常使用在条件句中：

```
if is_divisible(x, y):
    print "x is divisible by y"
else:
    print "x is not divisible by y"
```

你也许会想这样写：

```
if is_divisible(x, y) == True:
```

其实额外的比较是没有必要的。

5.5 函数（function）型态

和 int、float、str、bool 与 NoneType 一样，函数是 Python 中另一个型态。

```
>>> def func():
...     return "function func was called..."
...
>>> type(func)
<type 'function'>
>>>
```

也如同其它的型态，函数能被当作其它函数的自变量：

```
def f(n):
    return 3*n - 6

def g(n):
    return 5*n + 2

def h(n):
    return -2*n + 17

def doto(value, func):
    return func(value)

print doto(7, f)
print doto(7, g)
print doto(7, h)
```

函数 doto 被呼叫了三次。7 每次都作为要计算的自变量，而函数 f、g 和 h 依次被传入 func 中。这个脚本的输出会是：

```
15
37
3
```

这个范例有点不自然，然而我们稍后会见到将一个函数传入另一个函数的有用范例。

5.6 有型的程序设计

对程序设计师来讲，程序的可读性是相当重要的，因为实作上阅读与修改程序会比撰写来的更频繁。这本书里的所有程序代码范例都参照 *Python Enhancement Proposal 8*（PEP 8），这是 Python 社群所发展的风格指南。

随着程序变得更复杂，我们将会有更多关于风格的说明，不过先知道以下观点会很有用：

- 用四个半角空格来缩排
- import 陈述应该要放在档案的最前面
- 不同的函数定义用两个空白行来区隔

- 把函数定义放在一起
- 将最顶层的陈述，包含函数呼叫，一起放在脚本的最下方

5.7 三引号字符串

除了我们在第二章第一次看到的单引号及双引号字符串外，Python 也提供了*三引号字符串（triple quoted strings）*：

```
>>> type("""This is a triple quoted string using 3 double quotes.""")
<type 'str'>
>>> type('''This triple quoted strings uses 3 single quotes.''')
<type 'str'>
>>>
```

三引号字符串里可以包含单引号及双引号：

```
>>> print '''"Oh no", she exclaimed, "Ben's bike is broken!"'''
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
```

最后，三引号字符串可以跨越多行：

```
>>> message = """This message will
... span several
... lines."""
>>> print message
This message will
span several
lines.
>>>
```

5.8 用 doctest 做单元测试

近年来，在软件开发中包含对原始程序代码的自动的**单元测试（unit testing）**通常是最佳作法 。单元测试提供一种自动确认的方式，检查个别部份的程序代码，如函数，是否能正确执行。这使得我们可以在后期变更一个函数的效果，并快速地测试它是否仍然可以完成它该做的工作。

Python 有一个内建的 doctest 模块，可以做简单的单元测试。octest 可以写在三引号字符串里面，放在函数主体或是脚本的*第一行*， 它们由直译器阶段范例组成，而这些范例包含了一系列在 Python 提示符下的输入，并紧接着预期从 Python 直译器得到的输出。

doctest 模块会自动执行任何由 >>> 开始的陈述，并且比对下一行程序代码与直译器所输出的结果。

要看这是如何运作的，将下列内容放在名为 myfunctions.py 的脚本中：

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
    """


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

最后三行程序代码使 doctest 得以执行，将它们放在任何包含 doctest 的档案底部。我们将会在第十章提到模块时一并解释它们是如何运作的。

执行这个脚本会产生如下输出：

```
$ python myfunctions.py
**********************************************************************
File "myfunctions.py", line 3, in __main__.is_divisible_by_2_or_5
Failed example:
    is_divisible_by_2_or_5(8)
Expected:
    True
Got nothing
**********************************************************************
1 items had failures:
   1 of   1 in __main__.is_divisible_by_2_or_5
***Test Failed*** 1 failures.
$
```

这是一个*失败测试*的例子，这个测试说：「如果你呼叫 is_divisible_by_2_or_5(8)，结果应为 True。」既然如上的 is_divisible_by_2_or_5 并没有传回任何东西，测试就失败了，并且 doctest 告诉我们它预期得到 True 这个结果，但是没有得到任何东西。

我们可以直接传回 True 值以通过这个测试：

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
    """
    return True


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

如果我们现在执行这个脚本并不会有任何输出，这表示这个测试通过了。再次注意 doctest 模块的字符串必须直接放在函数定义的标头之后，这样才能执行。

要看更多的输出细节，使用 -v 命令列选项呼叫这个脚本：

```
$ python myfunctions.py -v
Trying:
    is_divisible_by_2_or_5(8)
Expecting:
    True
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.is_divisible_by_2_or_5
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
$
```

虽然测试通过了，我们的测试组却显然是不适当的，因为不论将什么自变量传入 is_divisible_by_2_or_5 都会传回 True。这里有一个包含更完备的测试组和程序代码的完整版本，可以通过这个测试：\

```
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
      >>> is_divisible_by_2_or_5(7)
      False
      >>> is_divisible_by_2_or_5(5)
      True
      >>> is_divisible_by_2_or_5(9)
      False
    """
    return n % 2 == 0 or n % 5 == 0


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

用 -v 命令列选项执行这个脚本，然后看看你得到什么。

5.9 术语

多效函数（fruitful function）：

产生传回值的函数。

传回值（return value）：

作为一个函数呼叫结果的数值。

暂存变数（temporary variable）：

复杂计算中用来储存中间值的变量。

死码（dead code）：

程序中永远不会被执行到的部分，通常是因为它出现在 return 陈述之后。

None：

有 return 陈述的函数或没有自变量的 return 陈述所传回的 Python 特别数值。None 是 NoneType 中唯一的数值。

渐进式开发（incremental development）：

借着一次只增加和测试少量程序代码，以避免除虫的程序开发计划。

支架码（scaffolding）：

在程序开发过程中用到却不包含在最后版本中的程序代码。

布尔函数（boolean function）：

传回布尔数值的函数。

函数的组合（composition of functions）：

在一个函数主题中呼叫另一个函数，或使用一个函数的传回值作为引号，呼叫另一个函数。

单元测试（unit testing）：

用来验证程序代码个别单元是否运作正常的自动机制，Python 中内建 doctest 可以完成这个目的。

5.10 练习

有的练习都应该加入 ch05.py 的档案中，并且在底部放入下面的程序代码：

```
if __name__ == '__main__':

    import doctest
    doctest.testmod()
```

依次完成每一个练习后，执行这个程序，以确认你的新函数通过 doctest 测试。

1. 写一个比较（compare）函数，如果 a > b 传回 1，a == b 为 0，a < b 则 -1。

```
def compare(a, b):
```

```
    """
      >>> compare(5, 4)
      1
      >>> compare(7, 7)
      0
      >>> compare(2, 3)
      -1
      >>> compare(42, 1)
      1
    """
    #  你的函数主体应由此开始。
```

完成函数主体以通过 doctest 测试。

2. 利用渐进式开发写一个名为 hypotenuse 的函数，并且以直角三角形的两股作为参数，让这个函数传回斜边长。同时记下你在渐进式开发过程中的每一步。

```
def hypotenuse(a, b):
    """
      >>> hypotenuse(3, 4)
      5.0
      >>> hypotenuse(12, 5)
      13.0
      >>> hypotenuse(7, 24)
      25.0
      >>> hypotenuse(9, 12)
      15.0
    """
```

当你完成后，将你的函数和 doctest 一起加入 ch05.py 中，并确认通过 doctest 测试。

3. 写一个 slope(x1, y1, x2, y2) 函数，传回通过 (x1, y1) 及 (x2, y2) 两点直线的斜率，并且确认 slope 的结果能够通过下列 doctest 测试：

```
def slope(x1, y1, x2, y2):
    """
      >>> slope(5, 3, 4, 2)
      1.0
      >>> slope(1, 2, 3, 2)
      0.0
      >>> slope(1, 2, 3, 3)
      0.5
      >>> slope(2, 4, 1, 2)
      2.0
    """
```

然后在新函数 intercept(x1, y1, x2, y2) 中呼叫 slope ，使其传回通过 (x1, y1) 及 (x2, y2) 两点直线的 y 截距。

```
def intercept(x1, y1, x2, y2):
```

```
    """
      >>> intercept(1, 6, 3, 12)
      3.0
      >>> intercept(6, 1, 1, 6)
      7.0
      >>> intercept(4, 6, 12, 8)
      5.0
    """
```

intercept 应该可以通过上面的 doctest 测试。

4. 写一个用整数作为自变量的 is_even(n) 函数，让它在自变量是**偶数**时传回 True，是**奇数**时传回 False，并将你自己的 doctest 测试加入这个函数。

5. 现在写一个 is_odd(n) 的函数，当 n 是奇数时传回 True，反之则传回 False。在你撰写时，加入给这个函数的 doctest 测试。最后，修改这个函数，使它呼叫 is_even 函数来判断它的自变量是否为一个奇数的整数。

6.

```
def is_factor(f, n):
    """
      >>> is_factor(3, 12)
      True
      >>> is_factor(5, 12)
      False
      >>> is_factor(7, 14)
      True
      >>> is_factor(2, 14)
      True
      >>> is_factor(7, 15)
      False
    """
```

替 is_factor 加入主体让它通过 doctest 测试。

7.

```
def is_multiple(m, n):
    """
      >>> is_multiple(12, 3)
      True
      >>> is_multiple(12, 4)
      True
      >>> is_multiple(12, 5)
      False
      >>> is_multiple(12, 6)
      True
      >>> is_multiple(12, 7)
      False
```

"""

替 is_multiple 加入主体让它通过 doctest 测试。你能找出一个将 is_factor 用在 is_multiple 定义里的方法吗？

8.

```
def f2c(t):
    """
      >>> f2c(212)
      100
      >>> f2c(32)
      0
      >>> f2c(-40)
      -40
      >>> f2c(36)
      2
      >>> f2c(37)
      3
      >>> f2c(38)
      3
      >>> f2c(39)
      4
    """
```

替 f2c 函数的函数定义写出主体的部份，让它传回和已知华氏温度最相近的摄氏温度整数值。*提示*：你可能会想要用内建函数 round，尝试在 Python shell 中印出 round.__doc__，并且用 round 实验，直到你可以流畅地使用为止。）

9.

```
def c2f(t):
    """
      >>> c2f(0)
      32
      >>> c2f(100)
      212
      >>> c2f(-40)
      -40
      >>> c2f(12)
      54
      >>> c2f(18)
      64
```

```
    >>> c2f(-48)
    -54
    """
```

为 c2f 加入函数主体，让它转换摄氏温度到华氏漫度。

6 重复

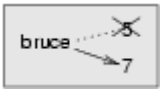
6.1 多重指派

你可能已经发现，指派多个值到相同的变量是允许的。新的指派使已经存在的变量指向新数值（并且停止指向旧数值）。

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

这个程序的输出是 5 7 ，因为第一次印出 bruce 时，它的数值是 5，而第二次印出时，它的数值是 7。第一个 print 陈述最后的逗点使输出后不会产生新的一行，这就是为什么两个输出发生在同一行。

以下是**多重指派**在状态图里看起来的样子：



在多重指派中，区别指派运作及相等陈述之间的不同是非常重要的。因为 Python 用等号（=）表示指派，所以如 a = b 的陈述常会被认为是相等陈述。其实它不是！

首先，相等是对称的，而指派不是对称的。举例来说，在数学上若 a = 7 则 7 = a。但是在 Python 中，a = 7 的陈述是合法的，但 7 = a 则否。

此外，数学中相等的陈述永远是真。如果现在 a = b，就是说 a 永远都与 b 相等。在 Python 中，指派陈述能够让两个变量相等，但是它们不一定永远都是这样：

```
a = 5
b = a     # a 和 b 现在相等
a = 3     # a 和 b 已不再相等
```

第三行改变了 a 的数值，但是并没有改变 b 的数值，所以它们已不再相等。（在某些程序语言，指派使用是不同符号以避免混淆，例如 <- 或 :=。

6.2 更新变数

更新是多重指派最普遍的型式，也就是说变量的新数值由旧数值得出。

```
x = x + 1
```

这个意思是拿 x 目前的数值，加上 1，然后用新数值更新 x。

如果你尝试更新一个不存在的变量，你就会得到错误，因为 Python 在指派结果数值到指派运算子左边的名称之前，会先求取右边的表达式：

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

在你更新变量之前，你必须**初始化**这个变量，通常是用一个简单的指派达成：

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

用增加 1 更新变量称为**递增**；减 1 则称为**递减**.

6.3 while 陈述

计算机常用来自动执行反复的工作。反复地执行相同或类似的工作而不出错，是件计算机做得很好，而人类做不好的事。

反复执行一组陈述就叫做**重复**。因为重复太普遍了，Python 提供了几个语言特征使它更容易。我们要看的第一个特征就是 while 陈述。

这里有个名为 countdown 的函数，示范了 while 陈述的用法：

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

你几乎能够把 while 陈述当成英文来读。它的意思是当 n 大于 0，就继续显示 n 的数值，然后将 n 的数值减去 1。当你达到 0 的时候，就显示 Blastoff!的文字

更正式地说，这是 while 陈述的执行流程：

1. 求出条件，产生 False 或 True。
2. 如果不符合条件，就离开 while 陈述，继续执行下一个陈述。
3. 如果符合条件，执行在主体内每个陈述，然后回到步骤 1。

主体由标头下方有相同缩排的所有陈述组成。

这种流程的型态叫做**循环**，因为在步骤 3 会返回一开始的步骤。请注意如果循环第一次检查时，不符合条件，循环内的陈述就永远不会执行。

循环的主体应该改变一个或多个变量的数值，使其最后变得不符合条件，而让循环结束。否则循环将会一直重复，这就叫做**无穷循环**。 计算机科学家一直都认为洗头发的指示非常有趣，洗发，冲洗，重复是一个无穷循环。

在 countdown 的例子中，我们能够证明循环结束是因为我们知道数值 n 是有限的，然后我们可以看到数值 n 每次通过循环就变小，所以最后我们就会得到 0。在其它的例子中，就不见得容易分辨了：

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

这个循环的条件是 n != 1 ，所以循环将会持续到 n 是 1 ，使其不符合条件为止。

每一次通过这个循环，程序输出 n 的数值，然后检查它是偶数或是奇数。如果是偶数，n 的数值就会除以 2。如果是奇数，数值就会被 n * 3 + 1 代换。举例来说，如果开始的数值（传给序列的自变量）是 3，结果序列便会是 3、10、5、16、8、4、 2、1。

既然 n 有时增加，有时减少，便没有明显的证明 n 就竟会不会到达 1，或是这个程序会结束。某些特定 n 的数值，我们能够证明终止。举例来说，如果开始的数值是 2 的次方，因而每一次 n 的数值通过循环都将会是偶数直到变成 1。之前例子便是以 16 起始的序列结束。

特定的数值撇开一旁，有趣的问题是我们能不能证明这个程序会被 *所有* n 的数值结束。到目前为止，没有人已经能够证明这是对的*或是*错的。

6.4 追踪一个程序

为了写有效率的计算机程序，程序设计师需要发展**追踪**计算机程序执行的能力。追踪包括成为计算机，并且藉由执行简单的程序，跟随执行流程，记录所有变量的状态以及程序执行每一项指令后产生的任何输出。

要了解这个过程，让我们追踪上一节对 sequence(3) 的呼叫。在追踪开始时，我们有一个初始值为 3 的区域变量 n （参数）。既然 3 并不等于 1，while 循环的主体就会执行。3 会被印出来，而且求取 3 % 2 == 0 的值。因为它求出的值为 False，else 分流就会执行，并且求取 3 * 3 + 1 的值，并指派到 n。

在你人工追踪一个程序时，为了记录以上所有的流程，请在一张纸画一栏给程序执行时所产生的每一个变量，并将输出的值记录在另一个字段中。我们的追踪到目前为止看起来这样：

```
  n              output
 ---             ------
  3                 3
  10
```

既然 10 != 1 求出 True 的值，循环主体会再次执行，然后印出 10。10 % 2 == 0 为真，所以会执行 if 分流，然后 n 变成 5。到追踪的结尾，我们得到：

```
  n              output
 ---             ------
  3                 3
  10                10
  5                 5
  16                16
  8                 8
  4                 4
  2                 2
  1
```

追踪可以是冗长乏味且易于犯错的（这就是为什么我们原先会用计算机做这种事！），但这是程序设计师不可或缺的技能。我们能够从这个追踪学习到很多关于程序代码运作的方式。举例来说，我们可以观察到 n 一旦变成 2 的次方，这个程序将会需要执行循环主体 $\log_2(n)$ 次才可以完成。我们也能够看到最后的 1 将不会被当成输出值印出。

6.5 计算数字

下面的函数计算十进制正整数的位数，并以十进制格式表示：

```
def num_digits(n):
    count = 0
    while n:
        count = count + 1
        n = n / 10
    return count
```

呼叫 num_digits(710) 将会传回 3。追踪这个函数呼叫的执行以使你自己确定它运作的方式。

这个函数示范了另一个叫做**计数器（counter）**的计算模式。变量 count 被初始化为 0，然后在每一次执行循环主体时递增。当离开循环时，count 就包含着结果 -- 循环主体被执行的总次数，这与位数是相同的。

如果我们希望只计算 0 或 5 的个数，可以计数器递增之前加入一个条件句：

```
def num_zero_and_five_digits(n):
    count = 0
    while n:
        digit = n % 10
        if digit == 0 or digit == 5:
            count = count + 1
        n = n / 10
    return count
```

证实 num_zero_and_five_digits(1055030250) 传回 7。

6.6 缩写指派

递增一个变量是太普遍了，因此 Python 替它提供了一个缩写语法：

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
>>>
```

count += 1 是 count = count + 1 的缩写。递增的数值不须为 1。

```
>>> n = 2
>>> n += 5
>>> n
7
>>>
```

-=、*=、/= 和 %= 也有缩写：

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
>>> n
6
>>> n /= 2
>>> n
3
>>> n %= 2
>>> n
1
```

6.7 表格

循环适合的其中一件事情便是产生列表式的数据。在计算机容易获得之前，人们必须手动计算对数、正弦、余弦及其它数学的函数。为了方便，数学书籍中包含着记载这些函数数值的长表格。创建这些表格费时而无聊，并且它们常充满错误。

当计算机出现后，最早期的反应之一就是太棒了！我们能够利用计算机来产生这些表格，就不会有错误。这个想法后来证明（大部分）是真的，但缺乏远见。不久之后，计算机及计算器普及到使表格变成过时的东西。

呃，几乎啦。对一些运算，计算机利用数值的表格取得一个近似的答案，然后执行计算以改进这个近似值。在一些例子中，基本的表格一直存在着错误，其中最知名的是 Intel Pentium 用来执行浮点数除法的表格。

虽然对数表格已不如过去有用，它仍然是一个重复的好例子。下列的程序会在左栏输出一个数值序列，而右栏则为 2 的左栏数值次方：

```
x = 1
while x < 13:
    print x, '\t', 2**x
    x += 1
```

字符串 '\t' 代表一个 **tab** 字符。在「\t」中的反斜线指出这是**逸出序列**的开始。跳脱序列用来表示不可见的字符，像是 tab 和新行。\n 序列就表示一个**新行**。

跳脱序列能够出现在字符串的任何地方；在这个例子中，tab 跳脱序列是这个字符串里唯一的内容。你觉得要如何在字符串中表示反斜线呢？

当字符和字符串被展示在屏幕上，一个叫做**光标**的隐形标记会记录下一个字符会在哪里。在 print 陈述之后，游标通常会到下一行的开头。

tab 字符会往右移动光标直到它抵达 tab 字符的停止点之一。Tab 有助于文字直行排列，就像在前述程序的输出中一样：

```
1       2
2       4
3       8
4       16
5       32
6       64
7       128
8       256
9       512
10      1024
11      2048
```

```
12      4096
```

因为 tab 字符在两行之间，第二行的位置并不取决于第一行的位数。

6.8 二维表格

二维表格是一种要你读取列与行交点数值的表格。乘法表是一个好例子。假设你想要印出数值从 1 到 6 的乘法表。

开始的好方法是写一个循环，使它在同一行印出 2 的倍数：

```
i = 1
while i <= 6:
    print 2*i, '   ',
    i += 1
print
```

第一行初始化一个名为 i 的变量，它扮演了计数器或是**循环变量**的角色。当这个循环执行的时候，i 的数值由 1 递增到 6。当 i 等于 7 时，循环就结束了。每一次通过循环，它就显示 2*i 的数值并紧接着三个空格。

此外，在 print 陈述中的逗点压抑了新行的输出。循环完成后，第二个 print 陈述开始新的一行。

程序的输出是：

```
2     4     6     8     10    12
```

到目前为止都还不错。下一步是封装（encapsulate）和一般化（generalize）。

6.9 封装与一般化

封装是把一段程序代码包裹进函数的过程，这让你可以充分利用函数的所有优点。你已经看过两个封装的例子：第四章的 print_parity 及第五章的 is_divisible。

一般化是说使某种特定的程序代码更具一般性，例如使印出 2 的倍数的程序代码印出任意整数的倍数。

下列函数封装前述的循环，并使它一般化印出 n 的倍数：

```
def print_multiples(n):
    i = 1
    while i <= 6:
        print n*i, '\t',
        i += 1
    print
```

要封装这段程序代码，我们只要增加第一行，它宣告了函数名称及参数列。要一般化这段程序代码，我们只要用参数 n 代换数值 2。

如果我们用 2 作为自变量呼叫这个函数，我们会得到跟之前相同的结果。而用 3 作为自变量，输出会是：

```
3       6       9       12      15      18
```

用 4 作为自变量，输出会是：

```
4       8       12      16      20      24
```

你现在大概猜出如何印出乘法表了---利用不同自变量重复呼叫 print_multiples。事实上，我们可以运用另一个循环：

```
i = 1
while i <= 6:
    print_multiples(i)
    i += 1
```

请注意这个循环跟 print_multiples 内的循环的相似度。我们只是用函数呼叫置换 print 陈述。

这个程序的输出是一份乘法表：

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

6.10 更多的封装

为了再一次示范封装，让我们把上一节的程序代码包裹在函数里：

```
def print_mult_table():
    i = 1
    while i <= 6:
        print_multiples(i)
        i += 1
```

这样的过程是一种常见的**开发计划**。我们藉由在任何函数外撰写程序代码，或是将它们键入直译器中开发程序。一旦我们取得可运作的程序代码，我们便将它抽出来，然后包进函数里。

如果在你开始写程序时，不知道如何分割程序为不同的函数，这样的开发计划对你就特别有用。这样的方法让你一边进行，一边规划。

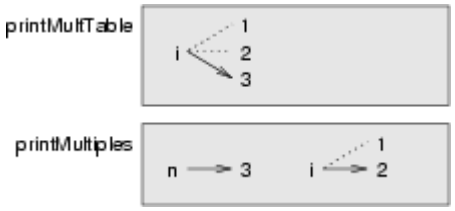
6.11 区域变量

你可能会好奇我们如何能在 print_multiples 与 print_mult_table 两个函数中，使用相同的变量 i。当其中一个函数改变变量的数值，不会发生问题吗？

答案是不会的，因为在 print_multiples 中的 i 以及在 print_mult_table 中的 i 并*不是*相同的变数。

在函数定义里产生的变量是区域的；你不能在母函数之外存取区域变量。这是说你可以自由使用多个相同名称的变量，只要他们不是出现在同一个函数中。

程序的堆栈图显示出两个名为 i 的变量不是相同的变量。它们能够代表不同的数值，并且改变一个不会影响到另一个。



i 的数值在 print_mult_table 从 1 跑到 6。在这张图里它刚好跑到 3。它通过下一次循环就会变成 4。每一次通过循环，print_mult_table 用目前 i 的数值当作自变量呼叫 print_multiples。那个数值被指派到参数 n。

在 print_multiples 里，i 的数值从 1 跑到 6。在这张图里，它刚好跑到 2。改变这个变量并不会影响到在 print_mult_table 里的 i 的数值。

使用相同名称的区域变量是相当普遍而合法的。特别是像 i 和 j 的名称经常用来当作循环变量的名称。如果你因为在某个地方用过了，而避免在一个函数中使用它们，你可能会让程序更难阅读。

6.12 更多的一般化

作为另一个一般化的例子，想象你希望程序可以印出任何大小的乘法表，不只是 6 × 6 的表格。你可以在 print_mult_table 中加入一个参数：

```
def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i)
        i += 1
```

我们用参数 high 换掉了数值 6。如果我们用自变量 7 呼叫 print_mult_table，它会展示：

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

还不错，除了我们可能希望表格是方的---行与列都有相同的数量。要做到那样，我们在 print_multiples 增加另一个参数，以指定表格的行数。

我们就讨人厌地称参数为 high 好了，用来说明不同的函数能够有相同名称的参数（就像区域变量）。以下是完整的程序：

```
def print_multiples(n, high):
    i = 1
    while i <= high:
        print n*i, '\t',
        i += 1
    print

def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i, high)
        i += 1
```

注意当我们增加新的参数，我们必须改变函数的第一行（函数的标头），然后我们也必须改变在 print_mult_table 里的呼叫位置。

如同预期的，这个程序产生了一个 7 × 7 的表格：

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

当你适当地一般化函数时，你通常会得到不在计划中的程序性质。举例来说，你可能会注意到因为 ab = ba，导致在表格中所有的条目都出现了两次。你可以只印出一半表格以节省油墨。要那样做，你只需要改变 print_mult_table 中的一行。改变

```
print_multiples(i, high)
```

为

```
print_multiples(i, i)
```

然后你得到

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

6.13 函数

现在我们已经提过几次函数的所有优点。你现在也许会想知道这些东西到底是什么。这里列出了一些：

1. 给予一序列的陈述一个名称，这使你的程序更容易阅读及除错。
2. 分割一个很长的程序为多个函数，这让你可以分开程序的组成部分，个别除错，然后再组合他们为一个整体。
3. 函数使重复使用更容易。
4. 良好设计的函数对许多程序时常常很有用。一旦你写好并且也除错好了，你就能够重复利用它。

6.14 牛顿勘根法

循环在程序中时常藉由从近似的答案开始，反复改进它，以计算数值型结果。

举例来说，计算平方根的一个方法就是牛顿勘根法。假设你想要知道 n 的平方根。如果你几乎可能从任何一个近似值开始，你能够借助下面的公式计算更好的近似值：

```
better =  (approx + n/approx)/2
```

我们可以藉由重复应用这个公式，直到较佳的近似值等于之前那个近似值为止，来写一个计算平方根的函数：

```
def sqrt(n):
    approx = n/2.0
    better = (approx + n/approx)/2.0
    while better != approx:
        approx = better
        better = (approx + n/approx)/2.0
    return approx
```

试着用 25 作为自变量呼叫这个函数，以确保它会回传 5.0。

6.15 算法

牛顿勘根法是**算法**的一个例子：这是解决特定问题的机械式步骤（在这里就是计算平方根）。

定义算法并不容易。也许由非算法的事情开始会有所帮助。当你学到个位数相乘的时候，你可能背过乘法表。事实上，你背了一百个特定的解法。这样的知识并不是算法。

但如果你很懒，你大概会藉由一些技巧作弊。举例来说，要找到 n 和 9 的乘积，你可以用 n - 1 当作第一个数字，然后 10 - n 作为第二个数字。这样的技巧就是个位数字乘以 9 的一般解法。那就是算法！

一样的道理，你所学过加法的进位、减法的借位以及长除法都是算法。算法的一个特征就是不需要智力完成。算法是按照简单法则一步接一步的机械式步骤。

依我们的看法，人类在学校花太多时间练习执行实际上不需智力的算法，是很丢脸的。

另一方面来说，设计算法的过程是有趣而且考验智力的，而其核心部份就是我们所说的程序设计。

有一些人们做起来很自然、毫无困难或知觉的事是最难用算法表达的。理解自然语言就是个好例子。我们所有人都说话，但直到现在没有人可以解释我们*如何*说话，至少不是用算法的型式。

6.16 术语

多重指派（multiple assignment）：

在程序执行的时候对相同变量做超过一次的指派。

初始化（变数）[initialization (of a variable)]：

初始化是说给予变量初始值，通常发生在多重指派的内容之中。因为 Python 中的变量要被指派数值后才会存在，于是它们创建时就已经初始化。其它的程序语言里，情况并非如此，而且变量创建时可以不被初始化，在这种例子中，他们有默认值或是*垃圾*值。

递增（increment）

递增的意思是说增加 1，是名词也是动词。

递减（decrement）

减少 1。

重复（iteration）：

重复的执行一组程序的陈述。

循环（loop）：

一个陈述或一群陈述重复执行直到满足结束的条件。

无穷循环（infinite loop）：

一个永远不会满足结束条件的循环。

追踪（trace）：

用手动跟随执行流程，记录变量状态的改变以及输出的结果。

计数器（counter）

用来计算某些东西的变量，通常初始化为 0 ，并且在循环的主体内递增。

主体（body）：

循环内的陈述。

循环变量（loop variable）：

作为循环结束条件一部分的变量。

tab：

使光标移动到同一行中下一个 tab 停止点的特别符号。

新行（newline）：

使光标移动到下一行开头的特别符号。

游标（cursor）：

记录下一次在哪印出字符的的隐形标记。

跳脱序列（escape sequence）：

一个跳脱字符「\」紧接着一个或多个可印出的字符，以表示不可印出的字符。

封装（encapsulate）：

将大型复杂程序分成不同组件（像函数），并且互相隔离（例如使用区域变量）。

一般化（generalize）：

把某些不须特别指定的东西（像是常数）用某些合适的一般性东西代替（像是变数或是参数）。一般化使程序代码更有弹性，像是可以重复使用，有时甚至更容易撰写。

开发计划（development plan）：

开发一个程序的过程。在这一章中，我们举例说明了基于发展程序代码做简单、特定的工作，然后封装、一般化的开发风格。

算法（algorithm）：

逐步解决特定问题的过程。

6.17 练习

1. 写单独一个字符串印出

```
produces
this
output.
```

2. 给 6.14 的 sqrt 函数加入一个 print 陈述，在每一次计算时印出 better 的数值。用 25 作为自变量呼叫你修改过的函数，然后记下结果。

3. 追踪 print_mult_table 最后一个版本的执行，然后想出它是如何运作的。

4. 写一个 print_triangular_numbers(n) 函数印出前 n 个三角数。呼叫 print_triangular_numbers(5) 会产生以下的输出：

```
1       1
2       3
3       6
4       10
5       15
```

（提示：利用网络搜寻找三角数的解释。）

5. 打开一个叫做 ch06.py 的档案，然后加入以下的内容：

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

写一个以单一整数自变量的 is_prime 函数，当自变量为**质数**时回传 True，若非则回传 False。在你开发的时候加入 doctests 到你的函数中。

6. num_digits(0) 将会回传什么？修改这个例子使它回传 1。为甚么呼叫 num_digits(-24) 结果会是无穷循环（*提示： -1/10 计算出 -1*）？修改 num_digits 以便任何整数都能正确运作。

加入下面的内容到之前你所创建的 ch06.py 档案中：

```
def num_digits(n):
    """
      >>> num_digits(12345)
      5
      >>> num_digits(0)
      1
      >>> num_digits(-12345)
      5
    """
```

加入函数主体到 num_digits ，然后确信它能通过 doctests。

7. 将下面的内容加入到 ch06.py：

```
def num_even_digits(n):
    """
      >>> num_even_digits(123456)
      3
      >>> num_even_digits(2468)
      4
      >>> num_even_digits(1357)
      0
      >>> num_even_digits(2)
      1
      >>> num_even_digits(20)
      2
    """
```

替 num_even_digits 写出主体以便让它如预期地运作。

8. 将下面的内容加入到 ch06.py：

```
def print_digits(n):
    """
      >>> print_digits(13789)
      9 8 7 3 1
      >>> print_digits(39874613)
      3 1 6 4 7 8 9 3
      >>> print_digits(213141)
      1 4 1 3 1 2
    """
```

替 print_digits 写出主体以使它能通过已知的 doctests。

9. 写一个计算整数每个位数数字平方合的函数 sum_of_squares_of_digits。举例来说，sum_of_squares_of_digits(987) 应该传回 194，因为 9**2 + 8**2 + 7**2 == 81 + 64 + 49 == 194。

```
def sum_of_squares_of_digits(n):
    """
      >>> sum_of_squares_of_digits(1)
      1
      >>> sum_of_squares_of_digits(9)
      81
      >>> sum_of_squares_of_digits(11)
      2
      >>> sum_of_squares_of_digits(121)
      6
      >>> sum_of_squares_of_digits(987)
      194
    """
```

7 对照上面的 doctests 检查你的答案。

7. 字符串

7.1 复合数据型态

到目前为止我们已经看过五种型态：int、float、bool、NoneType 及 str。字符串性质上不同于其它四个，因为他们是由更小的片段所构成的---字符。

由小片段组成的型态叫做**复合数据型态**。依据我们所做的事，我们可能会想要将复合数据型态视为单一的整体，或者我们也可能想要存取它的一部分。这种模棱两可的情况是有用的。

中括号运算子从字符串选取单一的字符：

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

表达式 fruit[1] 从 fruit 选取第一个字符。变量 letter 指向这个结果。当我们展示 letter，我们会感到奇怪：

```
a
```

"banana" 的第一个字符不是 a，除非你是计算机科学家。为了许多反常的理由，计算机科学家总是从 0 开始计算。"banana" 第 0 个字母（zero-eth）是 b。第 1 个字母（one-eth）是 a， 然后第 2 个字母（two-eth）是 n。

如果你想要字符串中第 0 个字母，你只需要放置 0 或是任何计算出 0 的表达式到中括号中：

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

中括号里的表达式叫做**索引**。索引指定一个有序集合的成员，在这个例子是字符串中字符的集合。索引*指出*那一个成员是你想要的，也就是就是成员的名称。它可以是任何的整数表达式。

7.2 长度

len 函数回传字符串里字符的数量：

```
>>> fruit = "banana"
```

```
>>> len(fruit)
6
```

为了取得字符串的最后一个字母，你也许会尝试如下的内容：

```
length = len(fruit)
last = fruit[length]       # ERROR!
```

那样并不能运作。它会产生执行错误 IndexError: string index out of range。理由是没有第六个字母在 "banana" 之中。既然我们从 0 开始计数，六个字母便由 0 到 5 编号。要得到最后一个字符，我们必须从 length 减掉 1 ：

```
length = len(fruit)
last = fruit[length-1]
```

或着我们可以用负数的索引，这会从字符串的最后倒数过来。表达式 fruit[-1] 产生最后一个字母， fruit[-2] 会产生倒数第二个等等。

7.3 走访以及 for 循环

很多的计算包括处理字符串一次一个字符。通常是从头开始，依次选取每一个字符，对其做一些事情，然后持续到末端。这种处理的模式叫做 **走访**。其中一个编码走访的方式就是用 while 陈述：

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

循环走访这个字符串，同时自动的一行展示一个字母。循环的条件是 index < len(fruit)，所以当 index 与字符串的长度相等时，条件便是假，循环的主体就不会再执行。最后一个存取的索引是 len(fruit)-1，这是字符串的最后一个字符。

使用索引走访一组数值是如此的普遍，所以 Python 提供了另一个更可替代，而且更简单的句型 --- for 循环：

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

每一次通过循环，字符串里的下一个字符被指派给变量 char。循环持续直到没有字符为止。

下面的例子表现出如何用连接及 for 循环产生学习字母的学生的序列。学习字母的学生所指的是依字母顺序排列的字母或表格。举例来说，在 Robert McCloskey's 的书 *Make Way for Ducklings*，小野鸭的名字是 Jack、Kack、Lack、Mack、Nack、Ouack、Pack 及 Quack。下面的循环依次序输出这些名字：

```
prefixes = "JKLMNOPQ"
suffix = "ack"

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

这个程序的输出是：

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

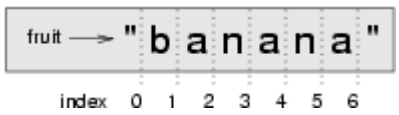
当然，这并不全然是对的，因为 Ouack 与 Quack 是拼错的。你将在最后练习的部份修正这个循环。

7.4 字符串切片

字符串的子字符串叫做 **切片**。选取切片如同选取字符：

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

运算子 [n:m] 回传从 n-eth 字符到 m-eth 字符的字符串部份，包括第一个，但是不包括最后一个。这种行为是违反直觉的；如果想象索引标在*介于*字符间，你便会更有感觉，如同下面的图：



如果你省略了第一个索引（在冒号之前），切片将会从字符串的起点开始。如果你省略了第二个索引，切片便会以字符串的末端作为结束。像这样：

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

你想 s[:] 的意义会是什么呢？

7.5 字符串比较

比较运算子一样可以运作在字符串上。为了确认两个字符串是否相等：

```
if word == "banana":
    print  "Yes, we have no bananas!"
```

其它有用的比较运算是把词按照字母顺序排列：

```
if word < "banana":
    print "Your word," + word + ", comes before banana."
elif word > "banana":
    print "Your word," + word + ", comes after banana."
else:
    print "Yes, we have no bananas!"
```

虽然你应该知道 Python 处理大、小写字母并不是和人一样。所有的大写字母出现在所有的小写字母之前。就像这样的结果：

```
Your word, Zebra, comes before banana.
```

解决这个问题的一个普遍方法是执行比较之前转换字符串到标准格式，例如全部小写。另一个比较困难的问题是让程序认识 zebras 不是水果。

7.6 字符串是不可变的

为了改变字符串里的字符，在指派的左边用 [] 运算子是很吸引人的。举例来说：

```
greeting = "Hello, world!"
greeting[0] = 'J'             # ERROR!
print greeting
```

并非产生 Jello, world! 的输出，程序代码产生执行错误 TypeError: 'str' object doesn't support item assignment。

字符串是**不可变的**，这个意思是说你不能改变已存在的字符串内容。最好的做法是建立一个新的字符串，作为原本字符串的对照：

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

这里的解决方法是在 greeting 的切片之上连接新的第一个字母。这样的运作并不会影响原始的字符串。

7.7 in 运算子

in 运算子测试一个字符串是否为另一个字符串的子字符串：

```
>>> 'p' in 'apple'
True
>>> 'i' in 'apple'
False
>>> 'ap' in 'apple'
True
>>> 'pa' in 'apple'
False
```

注意到一个字符串是它自己本身的子字符串：

```
>>> 'a' in 'a'
True
>>> 'apple' in 'apple'
True
```

结合 in 运算子与字符串连接运算子 +，我们能够写出从字符串中移除所有元音字母的函数：

```
def remove_vowels(s):
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    for letter in s:
        if letter not in vowels:
            s_without_vowels += letter
    return s_without_vowels
```

测试这个函数确信它做到我们想要它做的。

7.8 A find 函数

下面的函数会做什么呢？

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

在某个意义上，find 与 [] 运算子正好相反。并非是取得索引然后抽取出相对应的字符，它取得某个字符然后找到字符出现的索引位置。如果这个字符没有被找到，函数就回传-1。

这是我们看到 return 陈述在循环中的第一个例子。如果 strng[index] == ch，函数就会草率的直接跳出循环回传。

如果这个字符并不出现在字符串中，程序就会正常的结束并且回传 -1。

这种计算模型有时被称为 eureka 走访，因为一旦我们找到我们所找的，我们就能够大叫 Eureka，然后停止查看。

7.9 循环及参数

下面的程序计数字母 a 在字符串中出现的次数，并且是第六章介绍计数器模型的另一个例子：

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

7.10 选择性参数

为了找到字符在字符串中第二次或第三次出现的位置，我们可以修改 find 函数，增加在搜寻字符串起始位置的第三个参数：

```
def find2(strng, ch, start):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

呼叫 find2('banana', 'a', 2) 现在回传 3，就是第二个 'a' 在 'banana' 中的索引。

还有更好的做法是我们可以用**选择性参数**结合 find 与 find2：

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

find('banana', 'a', 2) 的呼叫对于 find 的版本就好比是 find2，当在 find('banana', 'a') 呼叫的时候，start 将会被设定成 0 的 **默认值**。

增加另一个选择性参数给 find，使它可以向前与向后的搜寻：

```
def find(strng, ch, start=0, step=1):
    index = start
```

```
    while 0 <= index < len(strng):
        if strng[index] == ch:
            return index
        index += step
    return -1
```

用数值 -1 给 step 传递将会使它朝向字符串的开始搜寻，而非结束的地方。注意到我们需要在 while 循环中检查索引的下界，这就和容纳改变的上界一样。

7.11 string 模块

string 模块处理字符串有用的函数。照例，我们在使用之前必须引入它：

```
>>> import string
```

为了看它里头有些什么，利用 dir 函数，以模块名称作为自变量。

```
>>> dir(string)
```

这将会回传 string 模块中项目的串行：

```
['Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__',
'__name__', '_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re',
'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'atof', 'atof_error',
'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize', 'capwords', 'center',
'count', 'digits', 'expandtabs', 'find', 'hexdigits', 'index', 'index_error',
'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase', 'lstrip',
'maketrans','octdigits','printable','punctuation','replace','rfind','rindex',
'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase',
'translate', 'upper', 'uppercase', 'whitespace', 'zfill']
```

为了发现更多关于串行里的项目，我们可以利用 type 命令。我们需要明载模块的名称，然后利用**小数点表示法**紧随着项目名称。

```
>>> type(string.digits)
<type 'str'>
>>> type(string.find)
<type 'function'>
```

既然 string.digits 是字符串，我们就可以印出它，看看它包含什么：

```
>>> print string.digits
0123456789
```

不令人意外，它包含每一个十进制数。

string.find 跟我们写过的函数所做的事情是一样的。为了发现更多关于它的内容，我们可以印出它的 **docstring**， __doc__， 这包含函数的说明文件：

```
>>> print string.find.__doc__
find(s, sub [,start [,end]]) -> in

    Return the lowest index in s where substring sub is found,
    such that sub is contained within s[start,end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

中括号里的参数是选择性参数。虽然我们可以利用 string.find，但是我们做了我们自己的 find：

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
```

这个例子示范了一个模块的优点---他们帮助避免内建函数与使用者自订函数在名称上的冲突。藉由小数点表示法我们能够指明哪一个 find 的版本是我们想要的。

事实上，string.find 比我们的版本更全面。它不只能找到字符，更能够找到子字符串：

```
>>> string.find("banana", "na")
2
```

就像我们的版本，它取得一个指定索引开始位置的附加自变量：

```
>>> string.find("banana", "na", 3)
4
```

却又不像我们的版本，第二个自变量指定搜寻应该停止的位置：

```
>>> string.find("bob", "b", 1, 2)
-1
```

在这个例子中，搜寻失败是因为字母 *b* 并不出现在索引从 1 到 2 的范围（不包括 2）。

7.12 字符的分类

检查字符、测试大小写或是判断是字符还是数字，这些常常是有帮助的。string 模块提供数个有助于这些目的的常数。其中一个，我们已经见过的是 string.digits。

字符串 string.lowercase 包含所有系统认为是小写的字母。相似的，string.uppercase 包含所有大写字母。尝试下面的陈述然后观察你得到什么：

```
print string.lowercase
print string.uppercase
print string.digits
```

我们可以利用这些常数，并且让 find 分类字符。举例来说，如果 find(lowercase, ch) 回传不是 -1 的数值， ch 就必定是小写字母：

```
def is_lower(ch):
    return string.find(string.lowercase, ch) != -1
```

或者，我们能利用 in 运算子：

```
def is_lower(ch):
    return ch in string.lowercase
```

另一个选择，我们可以使用比较运算子：

```
def is_lower(ch):
    return 'a' <= ch <= 'z'
```

如果 ch 介于 *a* 与 *z* 之间，它就必定是小写字母。

另一个定义在 string 模块的常数当你把它印出来时，大概会让你感到讶异：

```
>>> print string.whitespace
```

Whitespace 字符移动光标却没印出东西。他们建立介于可见字符的空白间隔（至少在白纸上）。常数 string.whitespace 内含所有 whitespace 的字符，包括空格、tab（\t）及新行（\n）。

在 string 模块中还有其它有用的函数，但是这本书并不打算作为参考手册。另一方面，*Python Library Reference* 才是。连同其它丰富的文件，你都能够从 Python 官网 http://www.python.org 取得。

7.13 字符串格式化

在 Python 里编排字符串最简洁及强大的方式是用*字符串格式化运算子*（ %），并且作为 Python 的字符串格式化运算。为了看这是如何运作的，让我们由几个例子开始：

```
>>> "His name is %s." % "Arthur"
'His name is Arthur.'
>>> name = "Alice"
>>> age = 10
>>> "I am %s and I am %d years old." % (name, age)
'I am Alice and I am 10 years old.'
>>> n1 = 4
>>> n2 = 5
>>> "2**10 = %d and %d * %d = %f" % (2**10, n1, n2, n1 * n2)
'2**10 = 1024 and 4 * 5 = 20.000000'
>>>
```

字符串格式化运算的句型看似这样：

```
"<格式>" % (<数值>)
```

由包含字符序列的*格式*以及*转换规格*开始。转换规格由 % 运算子开始。格式字符串后跟随的是单一个 %，然后是数值的序列，在小括号内由逗点分开，*每个数值对应一种转换规格*。如果只有一个数值，小括号就是任意的。

上面的第一个例子，只有一个转换规格，%s，这指的是字符串。这个数值，"Arthur"，对应到转换规格，并且没有包在小括号中。

在第二个例子，name 具有字符串的数值，"Alice"，而 age 则有整数数值 10。这些分别对应到两种转换规格，%s 及 %d。第二个转换规格中的 d 所指的是十进制整数。

第三个例子中 n1 及 n2 两个变量各别具有整数数值 4 与 5。在格式字符串中有四个转换规格：三个 %d 与一个 %f。f 指的是数值应该被表示成浮点数。四个分别对应到这四个转换规格的数值是：2**10、n1、n2 与 n1 * n2。

s、d 与 f 全都是我们在这本书中所需要的转换型态。浏览 Python Library Reference 的 String Formatting Operations 部份，可以看到完整的一览表。

下面的例子说明了字符串格式化的实用之处：

```
i = 1
print "i\ti**2\ti**3\ti**5\ti**10\ti**20"
while i <= 10:
    print i, '\t', i**2, '\t', i**3, '\t', i**5, '\t', i**10, '\t', i**20
    i += 1
```

这个程序印出从数字 1 到 10 的各种次方。在现在的形式中，排列数值的栏依赖 tab 字符（\t），但是当表格里的数值超过 8 个 tab 字符宽度时，就会失败：

```
i       i**2    i**3    i**5    i**10   i**20
1       1       1       1       1       1
2       4       8       32      1024    1048576
3       9       27      243     59049   3486784401
4       16      64      1024    1048576         1099511627776
5       25      125     3125    9765625         95367431640625
6       36      216     7776    60466176        3656158440062976
7       49      343     16807   282475249       79792266297612001
8       64      512     32768   1073741824      1152921504606846976
9       81      729     59049   3486784401      12157665459056928801
10      100     1000    100000  10000000000     100000000000000000000
```

可能的解决方法便是改变 tab 的宽度，但是第一栏的空格已经超过所需。最佳的解决方法会是每一栏个别设定宽度。就想你现在大概已经猜到，字符串格式化提供了这个解决方法：

```
i = 1
print "%-4s%-5s%-6s%-8s%-13s%-15s" %
      ('i', 'i**2', 'i**3', 'i**5', 'i**10', 'i**20')
while i <= 10:
    print "%-4d%-5d%-6d%-8d%-13d%-15d" % (i, i**2, i**3, i**5, i**10, i**20)
    i += 1
```

执行这个版本的方法产生如下的输出：

```
i   i**2 i**3  i**5    i**10         i**20
1   1    1     1       1             1
```

2	4	8	32	1024	1048576
3	9	27	243	59049	3486784401
4	16	64	1024	1048576	1099511627776
5	25	125	3125	9765625	95367431640625
6	36	216	7776	60466176	3656158440062976
7	49	343	16807	282475249	79792266297612001
8	64	512	32768	1073741824	1152921504606846976
9	81	729	59049	3486784401	12157665459056928801
10	100	1000	100000	10000000000	100000000000000000000

转换规格中每一个 % 之后的 - 指的是向左对齐的数量。数字指定最小的长度，所以 %-13d 是指向左对齐至少 13 字符的宽度。

7.14 索引

复合数据型态：

数值由成份或元素组成组成的数据型态，这些成份也是数值。

索引：

常于选取有序集合成员的变量或数值，就像是字符串中的字符。

走访：

反复通过集合的元素，对每个元素执行相似的运算。

切片：

字符串的部份（子字符串）由索引的范围指定。一般来说，Python 中任何序列型态的子序列都可以用切片运算子建立（*序列*[*起始位置*:*结束位置*]）。

不可变的：

复合数据型态的元素不能被指派新的数值。

选择性参数：

写在函数标头用默认值指派的参数，函数呼叫时若没有相对应的自变量就会以默认值代入。

默认值：

选择性参数被给予的数值，函数呼叫时若没有相对应的自变量就会以默认值代入。

小数点表示法：

使用**小数点运算子**，.，存取模块内的函数。

docstring：

在函数或是模块定义中的第一行字符串常数（我们稍后将会同样在类别与方法定义中看到）。Docstring 提供一个便捷的方法用程序代码连结说明文件。Docstrings 也被用在 doctest 模块做自动的测试。

whitespace：

移动光标有印出可见字符的字符。常数 string.whitespace 内含有所有印出空白间隔的字符。

7.15 练习

1. 修改下面的程序吗

```
prefixes = "JKLMNOPQ"
suffix = "ack"

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

以便让 Ouack 和 Quack 拼写正确。

2. 封装下面的程序代码

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

到名为 count_letters 的函数中，并且将其一般化以便使它接受字符串与字母作为自变量。

3. 现在重写 count_letters 函数，使它用第三个选择性参数指定字母开始计数的位置，重复的呼叫 find （8.10 的版本）以便代替走访整个字符串。

4. 你想哪一个 is_lower 的版本会是最快的？你能想到除了速度的其它原因，而选一个版本吗？

5. 建立一个名为 stringtools.py 的档案，然后将下面的程序代码放到档案中：

```
def reverse(s):
    """
      >>> reverse('happy')
      'yppah'
      >>> reverse('Python')
      'nohtyP'
      >>> reverse("")
```

```
      ''
      >>> reverse("P")
      'P'
    """


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

加入函数主体到 reverse 进行 doctests。

6. 加入 mirror 函数到 stringtools.py 中。

```
def mirror(s):
    """
      >>> mirror("good")
      'gooddoog'
      >>> mirror("yes")
      'yessey'
      >>> mirror('Python')
      'PythonnohtyP'
      >>> mirror("")
      ''
      >>> mirror("a")
      'aa'
    """
```

给这个函数写主体，让它如同 doctests 所说的方式运作。

7. 将 remove_letter 函数也纳入 stringtools.py 里。

```
def remove_letter(letter, strng):
    """
      >>> remove_letter('a', 'apple')
      'pple'
      >>> remove_letter('a', 'banana')
      'bnn'
      >>> remove_letter('z', 'banana')
      'banana'
      >>> remove_letter('i', 'Mississippi')
      'Mssssppi'
    """
```

给这个函数写主体，让它如同 doctests 所说的方式运作。

8. 最后，依次加入主体到下面的函数之中。

```
def is_palindrome(s):
    """
      >>> is_palindrome('abba')
      True
      >>> is_palindrome('abab')
```

```
    False
    >>> is_palindrome('tenet')
    True
    >>> is_palindrome('banana')
    False
    >>> is_palindrome('straw warts')
    True
  """

def count(sub, s):
  """
    >>> count('is', 'Mississippi')
    2
    >>> count('an', 'banana')
    2
    >>> count('ana', 'banana')
    2
    >>> count('nana', 'banana')
    1
    >>> count('nanan', 'banana')
    0
  """

def remove(sub, s):
  """
    >>> remove('an', 'banana')
    'bana'
    >>> remove('cyc', 'bicycle')
    'bile'
    >>> remove('iss', 'Mississippi')
    'Missippi'
    >>> remove('egg', 'bicycle')
    'bicycle'
  """

def remove_all(sub, s):
  """
    >>> remove_all('an', 'banana')
    'ba'
    >>> remove_all('cyc', 'bicycle')
    'bile'
    >>> remove_all('iss', 'Mississippi')
    'Mippi'
    >>> remove_all('eggs', 'bicycle')
    'bicycle'
  """
```

直到所有的函数都通过 doctests。

9. 在 Python shell 尝试下面每一个格式化字符串，然后记下结果：

    a. "%s %d %f" % (5, 5, 5)
    b. "%-.2f" % 3
    c. "%-10.2f%-10.2f" % (7, 1.0/2)
    d. print " $%5.2f\n $%5.2f\n $%5.2f" % (3, 4.5, 11.2)

10. 下面的格式化字符串有错误。修正他们：

    e. "%s %s %s %s" % ('this', 'that', 'something')
    f. "%s %s %s" % ('yes', 'no', 'up', 'down')
    g. "%d %f %f" % (3, 3, 'three')

8 案例研究： Catch

8.1 起步

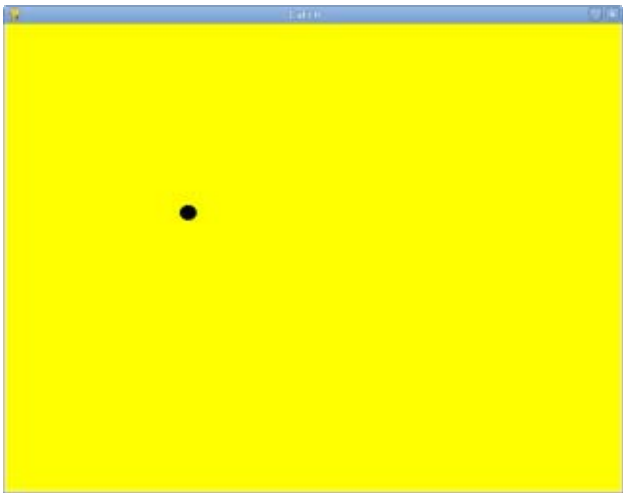
在我们第一个案例研究中，我们即将用 GASP 套件中的工具建立一个小型电动。游戏会从窗口的左到右射出一颗球，然后你要操作右边的手套接住这颗球。

8.2 用 while 移动球

while 陈述与 gasp 可以用来在程序中增加动作。下面的程序移动一颗黑色的球穿过 800 x 600 画素的绘图画布。请将这些程序代码加入名为 pitch.py 的档案。

```
from gasp import *

begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

ball_x = 10
ball_y = 300
ball = Circle((ball_x, ball_y), 10, filled=True)
dx = 4
dy = 1

while ball_x < 810:
    ball_x += dx
    ball_y += dy
    move_to(ball, (ball_x, ball_y))
    wait()

end_graphics()
```

当球通过屏幕，你将会看到像是如下的图形窗口：



追踪程序前几个重复动作，以确定你了解变量 x 与 y 发生了什么事。

从这个例子可以学到有关 GASP 的一些新用法：

- begin_graphics 能够取得绘图画布关于宽、高、标题及背景色的自变量。.
- set_speed 能够取得每秒的画面更新率。
- 将 filled=True 的属性加入到 Circle(...) 中，使结果的圆圈成实心状。
- 利用 ball = Circle 将圆圈储存到名为 ball 变量中（我们稍后将会讨论实际上圆圈是什么），使得它能在稍后沿用。
- GASP 的 move_to 函数允许程序设计师传递形状（这个例子是球）及位置，然后移动形状到那个位置。
- wait 函数被用来延迟 gasp 程序中的动作，直到发生特定事件。根据预设它会等待下一个画面，等待时间由 set_speed 所设定的画面更新率所决定。

8.3 程度的调整

为了使我们的游戏更有趣点，我们希望可以改变球的速度及方向。GASP 有个函数，random_between(low, high) 回传介于 low 与 high 之间的**随机**整数。要看这是如何运作的，请执行下面的程序：

```
from gasp import *

i = 0
while i < 10:
    print random_between(-5, 5)
    i += 1
```

每依次这个函数被呼叫，就会选取约介于 -5 到 5 的随机整数。当我们执行这个程序，我们会得到：

```
-2
-1
-4
1
-2
3
-5
-3
4
-5
```

你大概会得到一个不同的数字序列。

让我们用 random_between 改变球的方向。在指派 1 到 y 的 pitch.py 中置入这一行：

```
dy = 1
```

指派一个介于 -4 到 4 的随机数字：

```
dy = random_between(-4, 4)
```

8.4 使球弹回

执行新版的程序，你会注意到球常常还没到右边，就会从窗口的上方或下方的边缘离开。为了防止这种情况，让我们使这颗球从边缘弹回来，藉由改变 dy 的符号，然后把球送回相反的垂直方向。

将下面的程序代码加入 pitch.py 中 while 循环主体的第一行：

```
if ball_y >= 590 or ball_y <= 10:
    dy *= -1
```

多执行这个程序几次，然后看它如何表现。

8.5 break 陈述

break 被用来直接离开循环的主体。下面的程序设置了简单的猜数字游戏：

```
from gasp import *

number = random_between(1, 1000)
guesses = 1
guess = input("Guess the number between 1 and 1000: ")

while guess != number:
    if guess > number:
        print "Too high!"
    else:
        print "Too low!"
```

```
    guess = input("Guess the number between 1 and 1000: ")
    guesses += 1

print "\n\nCongratulations, you got it in %d guesses!\n\n" % guesses
```

使用 break 陈述，我们能够重写这个程序，减少重复两次的 input 陈述。

```
from gasp import *

number = random_between(1, 1000)
guesses = 0

while True:
    guess = input("Guess the number between 1 and 1000: ")
    guesses += 1
    if guess > number:
        print "Too high!"
    elif guess < number:
        print "Too low!"
    else:
        print "\n\nCongratulations, you got it in %d guesses!\n\n" % guesses
        break
```

这个程序利用了数学的三一律（已知实数 a、b，a 与 b 之间的关系若不是 a>b 或 a<b，那就是 a=b）。虽然这个程序的两个版本都是十五行的长度，不过第二个版本常被认为有较清楚的逻辑。

请将这些程序代码放入名为 guess.py 的档案中。

8.6 响应键盘

下面的程序建立一个会响应键盘输入的圆圈（或是手套）。按 j 或 k 两键分别移动手套往上或朝下移动。加入这些程序代码到名为 mitt.py 的档案中：

```
from gasp import *

begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

mitt_x = 780
mitt_y = 300
mitt = Circle((mitt_x, mitt_y), 20)

while True:
    if key_pressed('k') and mitt_y <= 580:
        mitt_y += 5
    elif key_pressed('j') and mitt_y >= 20:
```

```
        mitt_y -= 5

    if key_pressed('escape'):
        break

    move_to(mitt, (mitt_x, mitt_y))
    wait()

end_graphics()
```

执行 mitt.py，按 j 和 k 在屏幕上下之间移动。

8.7 检查碰撞

下面的程序中使两颗球从屏幕相对的两边朝彼此移动。当他们碰撞的时候，两颗球同时消失，然后程序终止：

```
from gasp import *

def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

ball1_x = 10
ball1_y = 300
ball1 = Circle((ball1_x, ball1_y), 10, filled=True)
ball1_dx = 4

ball2_x = 790
ball2_y = 300
ball2 = Circle((ball2_x, ball2_y), 10)
ball2_dx = -4

while ball1_x < 810:
    ball1_x += ball1_dx
    ball2_x += ball2_dx
    move_to(ball1, (ball1_x, ball1_y))
    move_to(ball2, (ball2_x, ball2_y))
    if distance(ball1_x, ball1_y, ball2_x, ball2_y) <= 20:
        remove_from_screen(ball1)
        remove_from_screen(ball2)
        break
```

```
    wait()

wait(event=ELAPSED_TIME, delay=1)
end_graphics()
```

将这个程序放到名为 collide.py 的档案中，然后执行它。

8.8 组合这些片段

为了结合移动的球、移动的手套，以及碰撞的预防，我们需要单一的 while 循环依次做这些事情：

```
from gasp import *

def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

ball_x = 10
ball_y = 300
ball = Circle((ball_x, ball_y), 10, filled=True)
dx = 4
dy = random_between(-4, 4)

mitt_x = 780
mitt_y = 300
mitt = Circle((mitt_x, mitt_y), 20)

while True:
    # move the ball
    if ball_y >= 590 or ball_y <= 10:
        dy *= -1
    ball_x += dx
    if ball_x > 810:            # the ball has gone off the screen
        break
    ball_y += dy
    move_to(ball, (ball_x, ball_y))

    # check on the mitt
    if key_pressed('k') and mitt_y <= 580:
        mitt_y += 5
    elif key_pressed('j') and mitt_y >= 20:
        mitt_y -= 5
```

```
    if key_pressed('escape'):
        break

    move_to(mitt, (mitt_x, mitt_y))

    if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:  # ball is caught
        remove_from_screen(ball)
        break

    wait()

end_graphics()
```

将这个程序放到名为 catch.py 的档案中，并且多执行几次。请务必在其中几次接到球，其它则漏接球。

8.9 显示文字

这个程序在图形屏幕上展示游戏者与计算机分别所得的分数。它产生 0 或 1 的随机数（像是翻转硬币），如果是 1 则游戏者加 1 分，0 则计算机加 1 分。然后更新显示在屏幕上。

```
from gasp import *

begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

player_score = 0
comp_score = 0

player = Text("Player: %d Points" % player_score, (10, 570), size=24)
computer = Text("Computer: %d Points" % comp_score, (640, 570), size=24)

while player_score < 5 and comp_score < 5:
    wait(event=ELAPSED_TIME, delay=1)
    winner = random_between(0, 1)
    if winner:
        player_score += 1
        remove_from_screen(player)
        player = Text("Player: %d Points" % player_score, (10, 570), size=24)
    else:
        comp_score += 1
        remove_from_screen(computer)
        computer = Text("Computer: %d Points" % comp_score, (640, 570), size=24)
```

```
if player_score == 5:
    Text("Player Wins!", (340, 290), size=32)
else:
    Text("Computer Wins!", (340, 290), size=32)

wait(event=ELAPSED_TIME, delay=4)

end_graphics()
```

将这个程序放到名为 scores.py 的档案中，然后执行它。

我们现在可以修改 catch.py ，显示谁是赢家。紧接在 if ball_x > 810: 条件句后，加入下面的内容：

```
Text("Computer Wins!", (340, 290), size=32)
wait(event=ELAPSED_TIME, delay=2)
```

当玩家为赢家时的显示留在后面当作练习。

8.10 抽象化

我们的程序变得有一点复杂。更糟糕的是，我们准备增加它的复杂度。下一个阶段的发展需要**巢状循环**。外层的循环将会重复操控游戏的回合，直到玩家或是计算机达到胜利分数为止。内层的循环将会是我们写过的其中一个程序，发球接球的单一回合，然后决定发生的是接到或是漏接。

研究指出对我们处理认知事物有明显的极限（见 George A. Miller 的 The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information）。程序变得越复杂，即使有经验的程序设计师也越难发展及维护。

为了处理复杂度的增加，我们可以把相关的陈述包进函数中，利用**抽象化**隐藏程序的细节。这使我们得以对待一组程序的陈述如同单一的概念，为进一步的工作释放心灵的频宽。使用抽象化的能力在计算机程序设计中是一个最有威力的观念。

这里是 catch.py 的完整版本：

```
from gasp import *

COMPUTER_WINS = 1
PLAYER_WINS = 0
QUIT = -1


def distance(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5
```

```
def play_round():
    ball_x = 10
    ball_y = random_between(20, 280)
    ball = Circle((ball_x, ball_y), 10, filled=True)
    dx = 4
    dy = random_between(-5, 5)

    mitt_x = 780
    mitt_y = random_between(20, 280)
    mitt = Circle((mitt_x, mitt_y), 20)

    while True:
        if ball_y >= 590 or ball_y <= 10:
            dy *= -1
        ball_x += dx
        ball_y += dy
        if ball_x >= 810:
            remove_from_screen(ball)
            remove_from_screen(mitt)
            return COMPUTER_WINS
        move_to(ball, (ball_x, ball_y))

        if key_pressed('k') and mitt_y <= 580:
            mitt_y += 5
        elif key_pressed('j') and mitt_y >= 20:
            mitt_y -= 5

        if key_pressed('escape'):
            return QUIT

        move_to(mitt, (mitt_x, mitt_y))

        if distance(ball_x, ball_y, mitt_x, mitt_y) <= 30:
            remove_from_screen(ball)
            remove_from_screen(mitt)
            return PLAYER_WINS

        wait()


def play_game():
    player_score = 0
    comp_score = 0

    while True:
```

```
        pmsg = Text("Player: %d Points" % player_score, (10, 570), size=24)
        cmsg = Text("Computer: %d Points" % comp_score, (640, 570), size=24)
        wait(event=ELAPSED_TIME, delay=3)
        remove_from_screen(pmsg)
        remove_from_screen(cmsg)

        result = play_round()

        if result == PLAYER_WINS:
            player_score += 1
        elif result == COMPUTER_WINS:
            comp_score += 1
        else:
            return QUIT

        if player_score == 5:
            return PLAYER_WINS
        elif comp_score == 5:
            return COMPUTER_WINS


begin_graphics(800, 600, title="Catch", background=color.yellow)
set_speed(40)

result = play_game()

if result == PLAYER_WINS:
    Text("Player Wins!", (340, 290), size=32)
elif result == COMPUTER_WINS:
    Text("Computer Wins!", (340, 290), size=32)
wait(event=ELAPSED_TIME, delay=4)

end_graphics()
```

从这个例子我们可以学到一些新的观念：

遵遁良好的组织习惯使程序更容易阅读。在程序中使用下列的组织：

- 引进
- 全域常数
- 函数定义
- 程序主体

- 符号**常数**像 COMPUTER_WINS、PLAYER_WINS 及 QUIT 可以被用来加强程序的可读性。习惯上用全部大写字母替常数命名。既然 Python 中没有提供简单的方式强

制指派新数值到常数（许多其它的程序语言却是如此），所以是否要这样做由程序设计师决定。

- 我们拿在 8.8 节发展的程序包进名为 play_round() 的函数之中。play_round 利用程序顶端定义的常数。记得 COMPUTER_WINS 比任意指派的数值都还要容易。
- 新的函数，play_game()，建立 player_score 与 comp_score 两个新的变量。使用 while 循环重复的呼叫 play_round，藉以检查每一次呼叫的结果并且恰当的更新分数。最后游戏者或计算机任一达到五分后，play_game 回传赢家到程序的主体，然后显示谁是赢家，接着结束程序。
- 有两个名为 result 的变量---一个在 play_game 函数，另一个则是在程序的主体。虽然他们有相同的名称，但是他们分别属于不同的*名称空间*，相互之间毫无关系。每一个函数建立属于自己本身的函数空间，函数主体所定义的名称并不可见于函数主体之外。名称空间将会在下一章详细的讨论。

8.11 术语

随机（random）：

不具有特定的模式，同时是不可预期的。然而计算机被设计成可预期的，所以不可能从计算机得到真正的随机数。特定的函数产生出数值的序列，他们的出现就好像是随机的，其实这些是我们从 Python 得到的*拟随机*数。

三一律（trichotomy）：

已知任意实数 a 与 b ，只有下列三种关系中的一种会成立：$a < b$、$a > b$ 或 $a = b$。于是当你能够证实其中两个关系为假，你就可以推论剩下的一个为真。

巢状循环（nested loop）：

在另一个循环主体中的循环。

抽象化（abstraction）：

其为藉由减少概念信息内容的*一般化*。Python 中的函数能够收集若干数量的程序陈述，以单一的名称除去细节，并且使程序更容易被理解。

常数（constant）：

程序执行的过程中不会改变的数值。习惯上常用全部大写字母的名称作为常数，由于没有语言的机制支持 Python 中真正的常数，所以 Python 程序设计师仍依赖训练才能依循这个习惯。

8.12 练习

1. 当执行 mitt.py 时，若你按下 <Escape> 键会发生什么事情？从程序找出执行这个动作的两行程序代码，并且解释他们是如何运作的。

2. 在 guess.py 中的作为计数的变量是那一个？猜到正确的数字在适当的策略之下，最多的次数应该为 11。这个策略是什么？

3. 当 mitt.py 里的手套到达图形窗口的顶端或底部的时候发生了什么事情？从程序中找出控制这些行为的程序代码，然后*详细的*解释他们的运作方式。

4. 改变在 collide.py 中 ball1_dx 的数值为 2。程序的表现有什么不同呢？现在再将 ball1_dx 改回 4，然后设定 ball2_dx 为 -2。*详细的*解释这些改变如何影响程序的行为。

5. 将 collide.py 的 break 陈述改成批注（在陈述前加上 # 符号）。. 你注意到程序的行为有何改变吗？现在也将 remove_from_screen(ball1) 陈述改成批注。现在发生了什么事？实验将 remove_from_screen 陈述与 break 陈述改成批注，或是改成不是批注，直到你能够*清楚的*描述这些一起运作的陈述如何在程序中产生预期的行为。

6. 你可以在哪里将下面的两行：

```
Text("Player Wins!", (340, 290), size=32)
wait(event=ELAPSED_TIME, delay=2)
```

加入 8.8 节中的 catch.py，以使得球被接到后，程序就会显示如上的讯息。

7. 追踪 catch.py 最后版本的执行流向，而当执行到 play_round 时，你按下了 <Escape> 键。当你按下这个按键的时候发生了什么事？为什么？

8. 列出 catch.py 最后版本的程序主体。*详细的*描述每一行程序代码做些什么事情。那一行陈述呼叫启动游戏的函数呢？

9. 辨认显示球及手套所负责的函数。有什么其它的效果可以提供给这个函数？

10. 哪一个函数追踪分数？这个函数也是显示分数的吗？藉由讨论程序代码建置这些效果的部份证明你的答案。

8.13 专题： pong.py

Pong 是最早的商业电动游戏之一。大写字母 P 是注册商标，但是 pong 被用来表示像拍子及球的桌球电动游戏。

catch.py 已经包含所有发展我们自己版本的 pong 所需要的程序设计工具。这个专题的目标是要渐进的从 catch.py 变成 pong.py，这使得你将会达成下面系列的练习：

1. 拷贝 catch.py 到 pong1.py 然后用 Box 而非 Circle 把球变成拍子。你可以参考附录 A 获得关于 Box 更多的信息。做一些让拍子保持在屏幕上必需的调整。

2. 拷贝 pong1.py 到 pong2.py。用布尔函数 hit(bx, by, r, px, py, h) 取代 distance 函数，使布尔函数在球的垂直坐标（by）介于拍子之间时，回传 True，同时球的水平位置（bx）要小于或等于半径（r），也就是远离拍子的正面。使用 hit 决定当球击中球拍，然后当 hit 回传 True 时使球弹回水平的相反方向。你的完整函数应该要通过这些 doctests：

```
def hit(bx, by, r, px, py, h):
    """
      >>> hit(760, 100, 10, 780, 100, 100)
      False
      >>> hit(770, 100, 10, 780, 100, 100)
      True
      >>> hit(770, 200, 10, 780, 100, 100)
      True
      >>> hit(770, 210, 10, 780, 100, 100)
      False
    """
```

最后，改变得分规则使球离开屏幕的左边时，游戏者得到一分。

3. 拷贝 pong2.py 到 pong3.py。在屏幕的左边增加一个新的拍子，当按下 'a' 往上移动，按下 's' 则往下。改变发球位置到屏幕的中央（400,300），然后在每一回合随机的向左或向右发球。

9 Tuple

9.1 可变性与 tuple

到目前为止，你已经看过了两种复合型态：字符串，它是由文字组成；和列表，它是由任何型态的元素组成。其中一个我们所注意到的不同之处是一个列表中的元素得以变更，但一个字符串中的文字则否。也就是说，字符串是**不可变的**，而列表则是**可变的**。

Python 中还有另外一种称为 **tuple** 的型态，它和列表非常类似，只不过它是不可变的。就语法上来说，一个 tuple 是一系列用逗号分隔的值：

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```

虽然并非必要，但我们习惯上会用括号包住 tuple：

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

要建立只有一个元素的 tuple 时，我们必须在最后面加上一个逗号：

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

如果没有这个逗号，Python 会将 ('a') 视为在括号中的字符串：

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

先不论语法问题，tuple 的运算和列表的运算是一样的。索引运算子（index operator）会从 tuple 中选择一个元素。

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

```
>>> tuple[0]
'a'
```

而切片运算子（slice operator）会选择一系列的元素。

```
>>> tuple[1:3]
('b', 'c')
```

但是如果我们尝试变更这个 tuple 中的元素，我们会得到错误讯息：

```
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```

当然，即使我们不能变更一个 tuple 中的元素，我们还是可以用一个不同的 tuple 取代它。

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```

9.2 Tuple 指派

有的时候，交换两个变数的值是很有用的。在传统的指派陈述中，我们必须使用一个暂时变量。举例来说，要交换 a 和 b 的话：

```
>>> temp = a
>>> a = b
>>> b = temp
```

如果我们必须经常这样做，这种方式会变得很麻烦。Python 提供了一种 **tuple 指派**的型式可以利落地解决这个问题：

```
>>> a, b = b, a
```

在左边的 tuple 代表变量，而在右边的 tuple 则表示值。每个值都被指派到各自的变数上。在右边所有的表达式会在指派前运算。这个功能让 tuple 指派非常有弹性。

自然地，在左边变量的个数和右边值的数量必须相同。

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

9.3 Tuple 做为传回值

函数可以传回 tuple 作为传回值。举例来说，我们可以写一个交换两个参数的函数：

```
def swap(x, y):
  return y, x
```

然后我们可以指派这个传回值到一个包含两个变量的 tuple 中：

```
a, b = swap(a, b)
```

在这个例子中，将 swap 写成一个函数并没有多大的益处。事实上，将 swap 封装起来是有点危险的，就像下列常犯错误一样：

```
def swap(x, y):       # 不正确版本
  x, y = y, x
```

如果我们像这样呼叫这个函数：

```
swap(a, b)
```

则 a 和 x 会是相同值的不同代号。变更 swap 中的 x 使得 x 代表一个不同的值，但它对 __main__ 中的 a 没有作用。同样地，变更 y 也不会对 b 产生作用。

这个函数执行时不会产生错误讯息，但它并没有做我们想要的事。这是一个语义错误的例子。

作为练习，请为这个函数画一个状态图，让你可以了解为什么它行不通。

9.4 随机数字

大多数计算机程序在每次执行时都执行相同的工作，所以它们被认为是**决定论的**。决定论通常是个好事，因为我们希望同样的运算能产生相同的结果。然而，在一些应用程序里，我们希望计算机是不可预测的。游戏是个很明显的例子，不过还有其它更多范例。

要让一个程序真正的不具有决定论并不是那么容易的，但仍有其它方式让它至少看起来像是不具有决定论的样子。方法之一是产生随机数字，并用它们决定程序的输出。Python 内建一个产生**伪随机**数字的函数，这些数字在数学上来说并不是真正随机的，不过它们依然可以满足我们的目标。

这个随机模块包含了一个称为 random 的函数，它会传回一个 0.0 到 1.0 之间的浮点数。每次你呼叫 random 的时候，你会得到一长串数字中的下一个数字。要看实例的话，执行这个循环：

```
import random


for i in range(10):
  x = random.random()
  print x
```

要产生一个介于 0.0 和上限值 high 之间的随机数字，可以用 x 乘以 high。

作为练习，请产生一个介于 low 和 high 之间的随机数字。

作为额外的练习，请产生一个介于 low 和 high 之间的随机整数，范围包含两个介值。

9.5 随机数字列表

我们第一步要先产生一个随机值的列表。randomList 可以取一个整数自变量并传回指定长度的随机数字列表。它以一个有 n 个零的列表开始。每一次通过循环，它会用一个随机数字取代其中一个元素。传回值是一个对完整列表的参照：

```
def randomList(n):
  s = [0] * n
  for i in range(n):
    s[i] = random.random()
```

```
  return s
```

我们将用一个包含八个元素的列表测试这个函数。为了除虫，最好先由小处着手。

```
>>> randomList(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

由 random 产生的数字应该会均匀地散布，也就是说每一个值出现的可能性要相同。

如果我们将可能值的范围等量分开，并计算一个随机值落入每个分区的次数，我们应该会在每个分区得到约略相同的数字。

我们可以写一个程序将范围等量分开，并计算每个分区中值的数量，来测试这个理论。

9.6 计算

一个解决这种问题的好方式是将问题分成数个子问题，并寻找那些符合你之前见过的计算模式的子问题。

在这个例子中，我们希望横跨一系列数字，并计算一个值落在特定范围中的次数。听起来很耳熟。在 7.8 节 中，我们写了一个程序，它会横跨一个字符串，并计算特定字母出现的次数。

因此，我们可以从复制这个旧程序，并依现在的问题改写开始。原来的程序是：

```
count = 0
for char in fruit:
  if char == 'a':
```

```
    count = count + 1
print count
```

第一步要将 fruit 用 t 取代，并将 char 取代成 num 。这并没有改变这个程序；这只是让它更具可读性。

第二步是要改变这个测试。我们对找出字母没有兴趣。我们想要知道 num 是否在特定值 low 和 high 之间。

```
count = 0
for num in t:
  if low < num < high:
    count = count + 1
print count
```

最后一步是要将这份程序代码包在一个称为 inBucket 的函式之中。参数是这个列表和 low 与 high 的值。

```
def inBucket(t, low, high):
  count = 0
  for num in t:
    if low < num < high:
      count = count + 1
  return count
```

藉由复制和变更现有的程序，我们得以快速地写出这个函数，并节省很多除虫的时间。这种开发计划称为**模式匹配（pattern matching）**。如果你发现自己在正处理一个你之前已经解决过的问题，请重复利用之前的解决。

9.7 许多分区

随着分区的数量增加，inBucket 使用上就会开始显得不够方便。只有两个分区时，它还不差：

```
low = inBucket(a, 0.0, 0.5)

high = inBucket(a, 0.5, 1)
```

不过当分区有四个时，它就开始麻烦了起来。

```
bucket1 = inBucket(a, 0.0, 0.25)

bucket2 = inBucket(a, 0.25, 0.5)

bucket3 = inBucket(a, 0.5, 0.75)

bucket4 = inBucket(a, 0.75, 1.0)
```

其中包含了两个问题。一是我们必须为每个结果取一个新的变量名称。其二是我们必须计算每一个分区的范围。

我们会先解决第二个问题。如果分区的数量是 numBuckets，那么每个分区的宽度就是 1.0 / numBuckets。

我们会使用一个回围计算每一个分区的范围。循环变量 i 会从 0 开始计算到 numBuckets-1：

```
bucketWidth = 1.0 / numBuckets

for i in range(numBuckets):

  low = i * bucketWidth

  high = low + bucketWidth

  print low, "to", high
```

要计算每个分区的下限，我们可以将循环变量乘以分区宽度。如此一来，上限就在一个 bucketWidth 之外了。

当 numBuckets = 8 的时候，输出会是：

```
0.0 to 0.125

0.125 to 0.25

0.25 to 0.375

0.375 to 0.5
```

```
0.5 to 0.625

0.625 to 0.75

0.75 to 0.875

0.875 to 1.0
```

你可以确认到每个分区具有相同的宽度，而且不会重迭，并加起来能够覆盖到整个从 0.0 到 1.0 的范围。

现在回到第一个问题。我们需要一个可以储存八个整数的方式，这些整数是经由循环变量一次指定一个而来。现在你应该想着「列表」！

我们必须在循环外建立分区列表，因为我们只想要做它一次。在这个循环中，我们会重复呼叫 inBucket 并更新列表中第 i 个元素：

```
numBuckets = 8

buckets = [0] * numBuckets

bucketWidth = 1.0 / numBuckets

for i in range(numBuckets):

  low = i * bucketWidth

  high = low + bucketWidth

  buckets[i] = inBucket(t, low, high)

print buckets
```

当处理有 1000 个值的列表时，这份程序代码会产出下列分区列表：

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

这些数字非常接近 125，这是我们期望的数字。至少，它们接近到我们可以相信随机数字产生器运作正常。

作为练习，用一些更长的列表测试这个函数，并看看是否在每个分区中值的数量有等量的倾向。

9.8 一个单程的解决方法

虽然这个程序可以运作，它并没有发挥它的最大效率。每一次它呼叫 inBucket 的时候，它就会横跨整个列表。随着分区的数量增长，横跨的次数也多得不得了。

如果可以制作一个只要单程跨过列表，就可以为每个值计算它落入的分区索引是最好的。然后我们就可以增加一个适当的计数器了。

在前一节中，我们把索引 i 乘以 bucketWidth 以找出特定分区的下限。现在我们想要取一个介于 0.0 到 1.0 范围中的值，并找出它落入的分区索引。

既然这个问题和前一个问题刚好相反，我们可以猜到我们应该把索引除以 bucketWidth 而不是乘上它。这个猜测是正确的。

既然 bucketWidth = 1.0 / numBuckets，除以 bucketWidth 就等于乘以 numBuckets。如果我们把一个介于 0.0 到 1.0 范围的数字乘以 numBuckets，我们会得到一个介于 0.0 到 numBuckets 范围的数字。如果我们将这个数字无条件舍去整数，我们就会刚好得到我们正在找的东西 — 一个分区索引：

```
numBuckets = 8

buckets = [0] * numBuckets

for i in t:

  index = int(i * numBuckets)

  buckets[index] = buckets[index] + 1
```

我们使用 int 函数将一个浮点数转换成整数。

这个计算是否可能产出一个超出范围（不论是负数还是大于 len(buckets)-1）的索引呢？

像 buckets 这种列表，会计算在每个范围中值的数量，就称为 **长条图（histogram）**。

作为练习，写一个称为 *histogram* 的函数，使其取一个列表和分区数量为自变量，并传回特定分区数量的长条图。

9.9 术语

不可变型态（immutable type）

在这个型态中，元素不可以变更。对元素的指派或分割不可变型态都会造成错误。

可变型态（mutable type）

在这个数据型态中，元素可以变更。所有的可变型态都是复合型态。列表和字典是可变数据型态；字符串和 tuple 则不是。

tuple

一种很类似列表的序列型态，只不过它是不可变的。Tuple 可用在任何需要一个不可变型态的地方，例如在一个在字典中的键值。

tuple 指派（tuple assignment）

使用单一指派陈述，就可以指派一个 tuple 中所有元素的指派。Tuple 指派是同时发生而非依序发生，使它适合用来交换数值。

决定论的（deterministic）

一个程序在每一次被呼叫时执行相同的事。

伪随机（pseudorandom）

一系列看起来像是随机的数字，但事实上是一个决定论的计算的结果。

长条图（histogram）

一个整数列表，其中每一个元素都计算某事发生的次数。

模式匹配（pattern matching）

一种程序开发计划，其中包含识别一个熟悉的计算模式，和复制相似问题的解决方式。

10 缺少

11 档案和例外

当一个程序运作时，它的数据会在内存里。当这个程序结束，或计算机关机时，在内存中的数据就会消失。要永久储存数据，你必须把它放入一个**档案**里面。档案通常储存在硬盘、软盘或光盘中。

当档案数量庞大时，它们常会被整理到一个**目录**（也称「数据夹」）中。每个档案是由一个独有的名称，或是档案和目录名称的独特组合来辨视。

借着读取和赛户档案，程序可以在彼此间交换讯息，并产生可打印的格式，如 PDF 。

使用档案和使用一本书很像。要使用一本书，你必须打开它。当你结束时，你必须合起它。当书打开时，你可以在里面写东西，或读它里面的东西。不论何者，你知道你在那本书的哪个位置。大部分时间，你依序读取整本书，不过你也可以跳着读。

这些都可以适用在档案上。要打开一个档案，你要指定它的名称，并指示你想要读取或是写入它。

开启一个档案会建立一个档案对象。在这个例子中，f 变量代表了新的档案对象。

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

开启函数包含了两个自变量。第一个是文件名称，而第二个是模式（mode）。"w" 模式意味着我们要写入这个档案。

如果没有称为 test.dat 的档案，则会自动建立一个。如果已经这个档案，它将被我们写入的档案所取代。

当我们印出档案对象时，我们会看到档案的名称、模式和对象的位置。

要将数据放进档案中，我们要激发在档案对象上的 write 方法：

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
```

关闭这个档案告诉系统我们结束写入了，并使这个档案可以读取。

```
>>> f.close()
```

现在我们可以再次开启这个档案，这次是要读取它，并将内容读取到一个字符串中。这时，模式自变量是 "r" ，代表了读取：

```
>>> f = open("test.dat","r")
```

如果我们尝试开启一个不存在的档案，我们会得到错误讯息：

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

显而易见地，read 方法会从档案中读取数据。没有自变量时，它会读取这个档案的全部内容：

```
>>> text = f.read()
>>> print text
Now is the timeto close the file
```

在 time 和 to 之间没有空格符，那是因为我们在字符串之间不会写入空格符。

read 也可以导入指示读取多少字符的自变量：

```
>>> f = open("test.dat","r")
>>> print f.read(5)
Now i
```

如果档案中的字符不够，read 会传回剩余的字符。当我们达到档案结尾时，read 会传回空白字符串：

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()


>>>
```

下列函数会复制一个档案，而且一次读取并写入最多五十个字符。第一个自变量是源文件的名称；第二个是新档案的名称：

```
def copyFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.read(50)
    if text == "":
      break
    f2.write(text)
  f1.close()
  f2.close()
  return
```

我们还没有看过 break 陈述。执行它会打断循环；执行流程会移动到循环之后的第一个陈述。

在这个例子中， while 循环是个无穷循环，因为 True 值永远为真。要跳脱这个循环*唯一的*办法就是执行 break，这会在 text 为空字符串时发生，也就是当我们到达档案结尾时。

11.1 文字文件

一个**文字文件**是一个包含排列成行的可打印字符和空格符，并由新行字符分隔的档案。既然 Python 特别设计来处理文字文件，它提供了让这个工作容易的方法。

为了展示这个特点，我们会建立一个文字文件，其中包含三行文字，并由新行字符分隔：

```
>>> f = open("test.dat","w")
>>> f.write("line one\nline two\nline three\n")
>>> f.close()
```

readline 方法会读取直到下个新行符号之前的所有字符，包含该新行符号。

```
>>> f = open("test.dat","r")
>>> print f.readline()
line one


>>>
```

readlines 会把剩余的所有行当成一个字符串列表传回：

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

在这个情况下，会以列表格式输出，这表示字符串会带有引号，而新行字符则以逸出序列
012 表示。

在该档案结尾，readline 会传回空白字符串，而 readlines 会传回空白列表：

```
>>> print f.readline()


>>> print f.readlines()
[]
```

以下是个处理行的程序例子。filterFile 会拷贝 oldFile 并略去任何以 #开头的行：

```
def filterFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.readline()
    if text == "":
      break
```

```
    if text[0] == '#':
      continue
    f2.write(text)
  f1.close()
  f2.close()
  return
```

continue 陈述结束目前的循环循环，但会继续循环。执行流程会移到循环顶端，检查条件，并依其进行。

因此，如果 text 是个空白字符串，就会循环就会退出。如果 text 的第一个字符是个井字号，执行流程会回到循环顶端。只有在同时不通过两个条件时，我们才会将 text 复制到新的档案中。

11.2 写入变数

write 的自变量必须是一个字符串，所以如果我们想要将其它值放入一个档案，我们必须先将它们转换成字符串。最简单的方法是使用 str 函数：

```
>>> x = 52
>>> f.write (str(x))
```

另一个方式则是使用**格式运算子（format operator）** %。当应用在整数上时，% 是指模数运算子。不过当第一个运算域是个字符串时，% 是格式运算子。

第一个运算域是**格式字符串**，而第二个运算域是一组表达式。结果会是一个包含表达式的值的字符串，并依据格式字符串格式化。

以下是个简单的例子，**格式序列** "%d" 的意思是在 tuple 中第一个表达式应该格式化为一个整数。在这里字母 *d* 的意思是「十进制的」：

```
>>> cars = 52
>>> "%d" % cars
'52'
```

结果会是这个 '52' 字符串，不要把他和整数值 52 搞混了。

一个格式序列可以出现在格式字符串中的任何地方，所以我们可以在一个句子中嵌入一个值：

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
'In July we sold 52 cars.'
```

格式序列 "%f" 会将 tuple 中下一个对象格式化为浮点数，而 "%s" 会将下一个对象格式化为字符串。

```
>>> "In %d days we made %f million %s." % (34,6.1,'dollars')
'In 34 days we made 6.100000 million dollars.'
```

依照预设，浮点数格式会印出小数点后六位。

在 tuple 中表达式的数量必须符合字符串中的格式序列数量。表达示的型态也必须符合格式序列：

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

在第一个范例中，没有足够的表达式；在第二个，表达式的格式错误。

要进一步控制数字的格式，我们可以在格式序列中的一部分指定位数：

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

在百分比符号后的数字代表数字可以使用的最少位数。如果我们提供的值使用更少位位数，就会在它前面加入空白位数。如果位数的数字为负数，就会在它后面加入空白位数：

```
>>> "%-6d" % 62
'62    '
```

对浮点数，我们也可以指定小数点后的位数。

```
>>> "%12.2f" % 6.1
'        6.10'
```

在这个例子中，结果会使用十二个位数，包含小数点后两个位数。这种格式对印出小数点对齐的金钱数量很有用。

举例来说，想象一个以学生名字为键值，并以时薪为值的 dictionary。以下这个函数会将这个 dictionary 的内容印成一个格式化报告：

```
def report (wages) :
  students = wages.keys()
  students.sort()
  for student in students :
    print "%-20s %12.2f" % (student, wages[student])
```

要测试这个函数，我们会建立一个小型 dictionary，并印出内容：

```
>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}
>>> report (wages)
joe                          5.45
joshua                       4.25
mary                         6.23
```

借着控制每一个值的宽度，只要名字小于二十一个字母，而薪水小于一小时十亿美元，我们就可以确保每一列都会对齐。

11.3 目录

当你藉由开启并写入一个档案来建立一个新档案时，这个新档案会归于当前的目录下（当你在执行程序时所在的目录）。同样地，当你要打开并读取一个档案时，Python 会在当前目录中寻找它。

如果你想开启其它地方的档案，你必须指定档案的**路径（path）**，也就是档案所在处的目录（或数据夹）名称：

```
>>>    f = open("/usr/share/dict/words","r")
>>>    print f.readline()
Aarhus
```

上述例子会打开一个名为 words 的档案，它位于一个名为 dict 的目录之下，dict 位于 share 之中，而 share 位于 usr 之中，usr 则位于系统最上层的目录 / 之中。

你没办法用 / 当成文件名称的一部分，它被保留来当成一个目录和文件名称之间的定义符号（delimiter）。

/usr/share/dict/words 这个档案包含了一个依字母排列的单字列表，第一个单字是一所丹麦大学的名字。

11.4 腌制

为了要将值放了一个档案，你必须将它们转换为字符串。你已经看过如果用 str 执行这个工作：

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

问题是当你读回这个值时，你会得到一个字符串。原始的型态信息已经遗失了。事实上，你甚至无法辨别一个值结束而下一个值开始的地方：

```
>>>    f.readline()
'12.3[1, 2, 3]'
```

解决的方法就是 **pickling（腌制）**会这么称呼它是因为它「保留」了数据。pickle 模块包含了必要的命令。要使用它，汇入 pickle 然后以正常方式开启档案：

```
>>> import pickle
>>> f = open("test.pck","w")
```

要储存一个数据结构，使用 dump 方法，然后以正常方式关闭这个档案：

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

然后我们可以开启并读取这个档案，而且加载我们转存出来的数据结构：

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

每一次我们使用 load 时，我们会从这个档案得到单一值，这个值附有它的原始型态。

11.5 例外

每当一个运作错误发生时，它会建立一个**例外**。通常，这个程序会停止，然后 Python 会印出一个错误讯息。

举例来说，一个数字除以 0 会产生一个例外：

```
>>> print 55/0

ZeroDivisionError: integer division or modulo
```

存取一个不存在的列表对象也会：

```
>>> a = []

>>> print a[5]

IndexError: list index out of range
```

或是存取一个不在 dictionary 中的键值：

```
>>> b = {}

>>> print b['what']

KeyError: what
```

或试着开启一个不存在的档案：

```
>>> f = open("Idontexist", "r")

IOError: [Errno 2] No such file or directory: 'Idontexist'
```

在每一个情况，错误讯息都有两个部分：逗点前是错误的型态，逗点后是错误的详细信息。通常 Python 也会印出一个程序执行到何处的追溯标记，不过我们在范例中将它略去了。

有时候我们会想要执行一个会导致例外的操作，但我们不希望程序停止。我们可以用 try 和 except 陈述**处理**例外。

举例来说，我们可能会提示使用者输入一个档案的名称，然后试着开启它。如果这个档案不存在，我们不希望程序当掉；我们想要处理这种例外：

```
filename = raw_input('Enter a file name: ')

try:

  f = open (filename, "r")

except IOError:

  print 'There is no file named', filename
```

try 陈述会执行第一个区块中的陈述。如果没有例外发生，它会忽略 except 陈述。如果发生一个型态为 IOError 的例外，它会执行 except 分支中的陈述，然后继续下去。

我们可以将这个能力封装在一个函数中： exists 会接受一个文件名称，然后在该档存在时传回 true，不存在时则传回 false：

```
def exists(filename):
  try:
    f = open(filename)
    f.close()
    return True
  except IOError:
    return False
```

你可以使用多个 except 区块来处理不同种类的例外。*Python Reference Manual* 中有更多详情。

如果你的程序侦测到一个错误情况，你可以让它**提出**一个例外。以下范例会接受使用者输入，并检查 17 这个值。假设 17 因为某个理由不是个效的输入，我们提出一个例外。

```
def inputNumber () :
  x = input ('Pick a number: ')
  if x == 17 :
    raise ValueError, '17 is a bad number'
  return x
```

raise 陈述会接受两个自变量：例外型态和错误的详细信息。ValueError 是 Python 在多种场合提供的例外型态之一。另一个范例包含了 TypeError、KeyError 和我的最爱 NotImplementedError。

如果呼叫 inputNumber 的函数可以处理这个错误，那么这个程序可以继续；反之，Python 会印出错误讯息并退出：

```
>>> inputNumber ()
```

```
Pick a number: 17

ValueError: 17 is a bad number
```

这个错误讯息包含例外型态，和你提供的额外讯息。

作为练习，写一个使用 inputNumber 的函数，让它可以从键盘输入一个数字，并处理 ValueError 例外。

11.6 术语

档案（file）：

一个已命名的实体，通常储存在硬盘、软盘或 CD-ROM 上，内含一连串的字符。

目录（directory）：

一个已命名的档案集合，也称为数据夹。

路径（path）：

一系列目录名称，指定档案的确切位置。

文字文件（text file）：

一个包含排列成行的可打印字符，并由新行字符分隔的档案。

break 陈述（break statement）：

一个导致执行流程退出循环的陈述。

继续陈述（continue statement）：

一个导致循环现有循环中止的陈述。执行流程会回到循环顶端，评估条件，并依其执行。

格式运算子（format operator）：

% 运算子接受一个格式化的字符串和一组表达式，产生一个包含表达式的字符串，该表达式会依据格式字符串格式化。

格式字符串（format string）：

一个字符串，包含可打印的字母和指示如果格式化值的格式序列。

格式序列（format sequence）：

一序列的字母，以 % 开始并指示如何格式化一个值。

腌制（pickle）：

将一个数据的值写入一个档案，并包含它的型态信息，让它可以在稍后复原。

例外（exception）：

在执行时发生的错误。

处理（handle）：

用 try 和 except 陈述防止一个例外结束程序。

提出（raise）：

使用 raise 陈述标志一个例外。

12 类别与对象

12.1 使用者设定的复合型态

在使用过一些 Python 内建的型态后，我们准备好要建立一个使用者定义的型态了：Point。

从数学上点的概念来看。在二维时，一个点是一并被视为单一对象的两个数字（坐标）。以数学符号表示时，点通常会使用括号包起来，并用一个逗号分隔坐标。例如，（0，0）代表原点，（x，y）就代表从原点向右 x 单位、向上 y 单位的点。

在 Python 中要表示一个点的自然方式就是使用两个浮点数。那么，问题在于如何把这两个值组合成一个复合对象。快速却杂乱的解决方式是使用一个 tuple 清单，不过对某些程序来说，这可能是最好的选择。

另一种方式是定义一个使用者定义的复合型态，也称为**类别**。这种方式得花更多力气，不过它的好处马上就会显露出来。.

一个类别定义看起来如下：

```
class Point:

  pass
```

类别定义可以出现在程序中任何一处，不过它们通常位于靠近开头的地方（在 import 陈述之后）。类别定义的语法规则和其它复合陈述一样（请见 4.4 节）。

这个定义建立了一个称为 Point 的新类别。**pass** 陈述并没有任何作用：不过它却是必要的，因为一个复合陈述的主体一定要有东西。

借着建立 Point 类别，我们也建立了一个新型态，它的名称也是 Point。这个型态的组成部分称为这个型态或是**对象**的**实例（instances）** 。建立一个新的范例则称为 **实例化（instantiation）**。要实例化一个 Point 对象，我们可以呼叫一个称为（你猜对了）Point 的函数：

```
blank = Point()
```

变量 blank 为新的 Point 对象指定了一个参照。一个像 Point 会建立一个新对象的函数就称为**建构子（constructor）**。

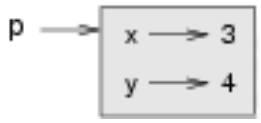
12.2 属性

我们可以用句点符号将新数据加入一个实例：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

这个语法和从一个模块中选择一个变量的句法相似，例如 math.pi 或 string.uppercase。不过在这里，我们是要从一个实例中选择一个数据项。这些有名称的项目就称为**属性（attributes）**。

下列状态图显示了这个指派的结果：



变量 blank 指的是一个 Point 对象，其中包含了两个属性。每一个属性指的是一个浮点数。

我们可以用相同的语法来读取一个属性的值：

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

blank.x 这个表达式表示「到 blank 所指的对象取得 x 值。」在这里我们指派这个值到一个称为 x 的变数。变量 x 和属性 x 之间并没有冲突。句点符号的目的是清楚明白地辨视你指的是哪一个变数。

你可以使用句点符号作为任何表达式的一部分，所以下列陈述是合法的：

```
print '(' + str(blank.x) + ', ' + str(blank.y) + ')'
distanceSquared = blank.x * blank.x + blank.y * blank.y
```

第一行会输出 (3.0, 4.0)；第二行则计算出 25.0 的值。

你可能会试着印出 blank 本身的值：

```
>>> print blank
<__main__.Point instance at 80f8e70>
```

这个结果表示 blank 是 Point 类别的实例，并在 __main__ 定义。80f8e70 是这个对象的唯一标示符，以十六进制法（base 16）表示。这大概不是表示一个 Point 对象最具信息性的方式。. 你稍后会看到如何改变它。

作为一个练习，建立并印出一个 Point 对象，然后使用 id 印出这个对象的唯一标示符。将十六进制格式转换成小数格式，并确认它们是相符的。

12.3 以实例作为自变量

你可以用平常的方法将一个实例当成一个自变量传递。举例来说：

```
def printPoint(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

printPoint 接受一个点作为自变量，并以标准格式显示它。如果你呼叫 printPoint(blank)，输出会是 (3.0, 4.0)。

作为一个练习，重新撰写 5.2 节 中的函数 distance ，让它可以接受两个 Point 作为自变量，而非四个数字。

12.4 相同性

到你深入考虑为止，「相同」一词的意思似乎十分明确，然后你会理解到，它的意义不止是如此。

举例来说，如果你说：「克里斯和我有一样的车。」你指的是他的车和你的车是同一厂牌和同一型号，不过你们有着不同的车。如果你说：「克里斯和我有同一个母亲。」你指的是他的母亲和你的是同一个人。* 注意： 所以「相同性」这个概念会依上下文不同。

当你谈到物件时，也有相似的歧异性。举例来说，如果两个 Point 是相同的，是否就意味着它们包含相同的数据（坐标）或它们是同一个对象呢？

要找到相同对象是否有两个参照，请使用 is 运算子。举例来说：

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 is p2
False
```

虽然 p1 和 p2 包含相同的坐标，他们却不是同一个对象。如果我们将 p1 指派到 p2，那么这两个变数会是同一个对象的别称：

```
>>> p2 = p1
>>> p1 is p2
True
```

这种相等称为浅层相等（shallow equality），因为它只比较参照，而不是对象的内容。

要比较对象的内容，—— 深层相等（deep equality） ——，我们可以写一个名为 samePoint 的函数：

```
def samePoint(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

现在如果我们建立了两个包含相同数据的不同对象，我们可以使用 samePoint 找出它们是否代表相同的点。

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
```

```
>>> p2.y = 4

>>> samePoint(p1, p2)

True
```

当然，如果这两个变量指的是同一个对象，它们就同时拥有浅层和深层相等。

12.5 矩形

假设我们想要一个表示一个矩形的类别。问题是，我们必须提供什么信息才能说明一个矩形呢？最简单的方法是，假设这个矩形不是垂直就是水平摆放，不会倾斜。

则有几个可能性：我们应该说明这个矩形的中心点（两个坐标）和它的大小（宽和高）；或者我们应该说明四角中的一角和大小；我们也可以说明两个相对的角。传统方法是指定矩形左上方的角落和大小。

我们会再一次指定一个新的类别：

```
class Rectangle:

  pass
```

And instantiate it:

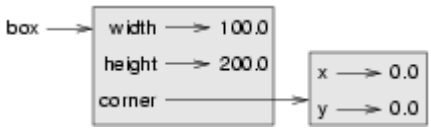
```
box = Rectangle()

box.width = 100.0

box.height = 200.0
```

这段程序代码会以两个浮点数属性，建立一个新的 Rectangle 对象。要指定左上方角落，我们可以在一个对象中埋入另一个物件！

```
box.corner = Point()

box.corner.x = 0.0

box.corner.y = 0.0
```

点运算子就会构成该角落。表达式 box.corner.x 的意思是：「到 box 所指的对象，并选择名称为 corner 的属性，然后再到该对象并选择名称为 x 的属性」。

下图表示这个对象的状态：



12.6 以实例作为传回值

函数可以传回实例。举例来说，findCenter 会以一个 Rectangle 作为自变量，并传回一个 Point ，其中包含了 Rectangle 中心的坐标：

```
def findCenter(box):
  p = Point()
  p.x = box.corner.x + box.width/2.0
  p.y = box.corner.y - box.height/2.0
  return p
```

要呼叫这个函数，并将 box 作为一个自变量传递，然后设派结果到一个变量里：

```
>>> center = findCenter(box)
>>> printPoint(center)
(50.0, -100.0)
```

12.7 物件是可变的

我们可以用指派属性的方式，改变一个对象的状态。举例来说，要改变一个矩形的大小，却不改变它的位置，我们可以变更 width 和 height 的值：

```
box.width = box.width + 50
box.height = box.height + 100
```

我们可以将这段程序代码包成一个 method，并使它可以任意放大矩形：

```
def growRect(box, dwidth, dheight) :
  box.width = box.width + dwidth
  box.height = box.height + dheight
```

变量 dwidth 和 dheight 表示矩形在每个方向应该放大多少。启动这个 method 有着变更自变量 Rectangle 的效果。

举例来说，我们可以建立一个新的 Rectangle，并称为 bob 然后把它传到 growRect：

```
>>> bob = Rectangle()
>>> bob.width = 100.0
>>> bob.height = 200.0
>>> bob.corner = Point()
>>> bob.corner.x = 0.0
>>> bob.corner.y = 0.0
>>> growRect(bob, 50, 100)
```

当 growRect 在运作时，参数 box 是 bob 的代称。任何对 box 的变更也会影响到 bob。

作为练习，写一个名为 moveRect 函数，并让它以一个 Rectangle 和两个名为 dx 和 dy 的参数作为自变量。应该会以把 dx 加到 corner 的 x 坐标，并将 dy 加到 corner 的 y 坐标的方式，改变矩形的位置。

12.8 复制

使用代称可能会使一个程序难以阅读，因为在一个地方所做的改变可能在另一个地方会有意想不到的效果。要追踪指涉一个特定对象的所有变量是很困难的。

复制一个对象常常是避免使用代称的另一个方法。copy 模块包含一个称为 copy 的函数，它可以复制任何对象：

```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
```

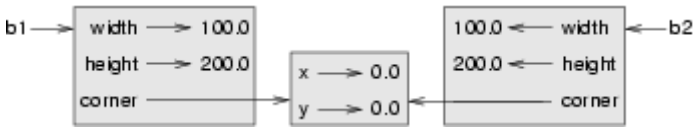
```
False
>>> samePoint(p1, p2)
True
```

一旦我们汇入 copy 模块后，我们可以使用 copy method 来建立一个新的 Point。p1 和 p2 不是同一个点，不过它们包含相同的数据。

要复制一个像 Point 一样没有嵌入其它对象的简单对象， copy 就够用了。这称为**浅层复制（shallow copying）**。

而对像 Rectangle 的东西来说，它包含了一个对 Point 的参照，copy 则没办法正常运作。它会复制对这个 Point 对象的参照，所以旧的 Rectangle 和新的都会指涉同一个 Point。

如果我们以平常的方式使用 copy，建立一个名为 b1 的矩形，然后复制它，并称为 b2，结果的状态图会如下：



这几乎可以确定不是我们所想要的。在这个情况下，在其中一个 Rectangles 启动 growRect 不会影响到另一个，不过在任何一方中启动 moveRect 则会影响到两者！这个行为会使人迷惑而且容易出错。

幸运的是，copy 模块包含了一个名为 deepcopy 的 method ，它不但会复制一个对象，还会复制任何嵌入的对象。我想你不会惊讶这种操作称为**深层复制（deep copy）**。

```
>>> b2 = copy.deepcopy(b1)
```

现在 b1 和 b2 是两个完全不同的对象。

我们可以使用 deepcopy 重写 growRect 让它不是变更一个现有的 Rectangle，而是在原来的地方建立一个大小不同的新 Rectangle。

```
def growRect(box, dwidth, dheight) :
  import copy
  newBox = copy.deepcopy(box)
  newBox.width = newBox.width + dwidth
```

```
  newBox.height = newBox.height + dheight

  return newBox
```

作为一个练习，重写 *moveRect*，让它建立并传回一个新的 *Rectangle*，而不是变更旧的矩形。

12.9 术语

类别（class）

一个使用者定义的复合型态。一个类别也可以想成是该类别实例对象的样版。

实例化（instantiate）

建立一个类别的实例。

实例（instance）

一个属于一个类别的对象。

物件（object）

一个复合数据型态，常用来表现真实世界的一个事物或概念。

建构子（constructor）

一个用来建立新对象的 method。

属性（attribute）

形成一个实例的已命名数据项之一。

浅层相等（shallow equality）

参照的相等，或两个指向同一个物件的参照。

深层相等（deep equality）

值的相等，或两个指向对象的参照有相同的值。

浅层复制（shallow copy）

复制一个对象的内容，包含任何对嵌入对象的参照；以 `copy` 模块中的 `copy` 函数执行。

深层复制（deep copy）

复制一个对象的内容和嵌入对象，和任何嵌入对象中的对象，以此类推；以 `copy` 模块中的 `deepcopy` 函数执行。

13 类别与函数

13.1 Time 类别

作为另一个使用者定义类型的范例，我们将定义一个名为 Time 的类别，它可以记录日期和时间。这个类别定义看起来如下：

```
class Time:
  pass
```

我们可以建立一个新的 Time 对象，并为小时、分钟和秒指派属性。

```
time = Time()
time.hours = 11
time.minutes = 59
time.seconds = 30
```

Time 对象的状态图看起来会像这样：

作为练习，请撰写一个 *printTime* 函数，让它可以接受 *Time* 对象作为自变量，并以 *小时:分钟:秒* 的格式列出结果。

作为第二个练习，请撰写一个布尔函数 *after*，让它可以接受两个 *Time* 对象 *t1* 和 *t2* 作为自变量，并于 *t1* 在时间上在 *t2* 之后时，传回 *True*，反之则传回 *False*。

13.2 纯函数

在下一节里，我们会撰写一个名为 addTime 的函数的两个版本，它会计算两个 Time 的总和。它们会展示两种函数：纯函数（pure functions）和修饰子（modifiers）。

下面是 addTime 的粗略版本：

```
def addTime(t1, t2):
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds
  return sum
```

这个函数会建立一个新的 Time 对象，初始化它的属性，并传回这个新对象的参照。这称为一个**纯函数**，因为它不会变更任何传给它作为自变量的对象，而且没有副作用，例如显示一个值或是取得使用者输入。

这里有个如何使用这个函数的范例。我们将建立两个 Time 对象：currentTime 包含了目前的时间；而 breadTime 包含了一台制面包机制作面包所需的时间。然后我们会使用 addTime 来计算什么时候面包会做好。如果你还没写好 printTime，请在尝试前看一下 14.2 节：

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds =  30

>>> breadTime = Time()
>>> breadTime.hours =  3
>>> breadTime.minutes =  35
>>> breadTime.seconds =  0

>>> doneTime = addTime(currentTime, breadTime)
>>> printTime(doneTime)
```

这个程序的输出会是 12:49:30，这是正确的。另一方面来说，有些情况下结果不会是正确的。你能想出一种结果不正确的情况吗？

问题在于这个函数不能处理秒数或分钟数加起来大于六十的情况。在这种情况下，我们得「搬运」多余的秒数到分钟栏里，或是将多余的分钟数搬到小时栏中。

以下是这个函数秒数修正后的版本：

```
def addTime(t1, t2):
  sum = Time()
  sum.hours = t1.hours + t2.hours
  sum.minutes = t1.minutes + t2.minutes
  sum.seconds = t1.seconds + t2.seconds

  if sum.seconds >= 60:
    sum.seconds = sum.seconds - 60
    sum.minutes = sum.minutes + 1

  if sum.minutes >= 60:
    sum.minutes = sum.minutes - 60
    sum.hours = sum.hours + 1

  return sum
```

虽然这个函数是正确的，不过它开始变得很庞大。稍后我们会介绍另一个可以产生较少程序代码的方法。

13.3 修饰子

有时候，一个函数可以变更一个或多个它接收作为自变量的对象是很有用的。通常，呼叫者会保留它所传出对象的参照，所以呼叫者可以看到任何函数所进行的变更。这种函数就称为**修饰子**。

increment 会在 Time 对象中加上特定秒数，用来写成修饰子是再自然不过的了。这个函数的粗略版本看起来如下：

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds

  if time.seconds >= 60:
    time.seconds = time.seconds - 60
    time.minutes = time.minutes + 1

  if time.minutes >= 60:
    time.minutes = time.minutes - 60
    time.hours = time.hours + 1
```

第一行程序代码进行基础运算；其余的部分则处理我们之前看过的特殊情况。

这个函数是否是正确的呢？如果参数 seconds 大于六十很多，会发生什么事呢？在这种情况下，搬运一次是不过的；我们必须持续搬运到秒数小于六十的时候。有一种解决方法是将 if 陈述取代为 while 陈述：

```
def increment(time, seconds):
  time.seconds = time.seconds + seconds

  while time.seconds >= 60:
    time.seconds = time.seconds - 60
    time.minutes = time.minutes + 1
```

```
while time.minutes >= 60:
    time.minutes = time.minutes - 60
    time.hours = time.hours + 1
```

这个函数现在是正确的了，不过这不是最有效率的解决方法。

作为练习，将这个函数重写为不包含任何循环的版本。

作为第二个练习，请将 increment 重写为纯函数，并写一个函数同时呼叫两个版本。

13.4 哪一个比较好？

任何可以用修饰子完成的事都可以用纯函数来完成。事实上，一些程序语言只充许纯函数。证据显示使用纯函数的程序比使用修饰子的程序开发速度较快，错误也较少。不过，修饰子有时是很方便的，在某些情况下，函数程序则较没有效率。

总体来说，我们建议你在合理时撰写纯函数，只在有利时才采用修饰子。这种方法可以称为**函数式程序风格（functional programming style）。**

13.5 原型开发 V.S. 计划

在这章中，我们展示了一个称为**原型开发（prototype development）**的的程序开发方法。针对每一个情况，我们撰写了一个粗略版本（或称原型），让它可以执行基本运算，然后我们在一些情况下测试，并在找到暇疵时更正它。

虽然这种方法可以很有效率，不过却可能导致程序代码不必要的复杂 —— 因为它处理许多特别的情况——而且也不太可靠——因为你很难知道你是否已经找到所有的错误。

另一种方法则是**计划式开发（planned development）**，在这种开发方式中，对问题的高度理解可以使开发更为容易。在这种情况下，我们理解到 Time 对象实际上是一个 60 进位 的三位数数字！秒数组件是「一」的字段， 分钟组件是「六十」的字段，而小时组件则是「三千六百」的字段。

当我们在撰写 addTime 和 increment 时，我们实际上以六十进制增加，这也是为什么我们必须从一个字段搬运到另一个字段。

这个观察暗示了另一种解决全部问题的方法——我们可以将 Time 对象转换成单一数字，然后利用计算机知道如何进行数字计算的特点。下列函数将一个 Time 对象转换成一个整数：

```
def convertToSeconds(t):
  minutes = t.hours * 60 + t.minutes
  seconds = minutes * 60 + t.seconds
  return seconds
```

现在，我们所需要的只是一个将整数转换为 Time 对象的方法：

```
def makeTime(seconds):
  time = Time()
  time.hours = seconds // 3600
  time.minutes = (seconds%3600) // 60
  time.seconds = seconds%60
  return time
```

你可能得深入考虑一下，才能确信这个函数是正确的。假设你已经确信了，你可以用它和 convertToSeconds 来重新撰写 addTime：

```
def addTime(t1, t2):
  seconds = convertToSeconds(t1) + convertToSeconds(t2)
  return makeTime(seconds)
```

这个版本比原来的版本短太多了，而且也比较容易展示它是正确的。

作为练习，请以相同方法重新撰写 increment。

13.6 一般化

在某些情况下，从 60 进位转换到 10 进位再转换回来比纯粹处理时间更难理解。进位转换较为抽象；用我们的直觉来处理时间较好。

不过如果我们能够理解如何用六十进制数来看待时间，并花时间撰写转换函数（convertToSeconds 和 makeTime），我们可以得到一个更短、更容易阅读和除虫也更可靠的程序。

它也比较容易在稍后增加功能。例如，想象我们要用两个 Time 相减，以找出它们之间的间隔。天真的方法会用借位的方法来实行相减。使用转换函数则较简单，而且比较有可能是正确的。

讽刺地是，有时使程序较为困难（也就是更为一般化）会使得它更为简单（因为有较少的特殊情况和较少错误的机会）。

13.7 算法

相对于为单一问题撰写一个特定解决方法，当你为一种类别的问题撰写一般化的解决方式时，你就写出了一个**算法（algorithm）**。我们在之前已经提过这个词，不过并没有特意定义它。它并不容易定义，所以我们会尝试一些方法。

首先，思考一下什么东西不算是算法。当你学到个位数相乘的时候，你可能背过乘法表。事实上，你背了一百个特定的解法。这样的知识并不是算法。

但如果你很「懒」，你大概会藉由一些技巧作弊。举例来说，要找到 n 和 9 的乘积，你可以用 n-1 当作第一个数字，然后 10-n 作为第二个数字。这样的技巧就是个位数字乘以 9 的一般解法。那就是算法！

一样的道理，你所学过加法的进位、减法的借位以及长除法都是算法。算法的一个特征就是不需要智力完成。算法是按照简单法则一步接一步的机械式步骤。

依我们的看法，人类在学校花太多时间练习执行实际上不需智力的算法，是很丢脸的。

另一方面来说，设计算法的过程是有趣而且考验智力的，而其核心部份就是我们所说的程序设计。

有一些人们做起来很自然、毫无困难或知觉的事是最难用算法表达的。理解自然语言就是个好例子。我们所有人都说话，但直到现在没有人可以解释我们 *如何*说话，至少不是用算法的型式。

13.8 术语

纯函数（pure function）

一种函数，不会变更任何它接收作为自变量的对象。大部分纯函数是很有生产力的。

修饰子（modifier）

一种函数，会变更一个或多个它接收作为自变量的对象。大部分修饰子生产力不佳。

函数式程序风格（functional programming style）

一种程序设计的风格，其中大部分的函数都是纯函数。

原型开发（prototype development）

一种开发程序的方法，以一个原型开始，逐步测试和改善。

计划式开发（planned development）

一种开发程序的方法，其中包含对问题的高度理解，而且比起增加式开发或原型开发有更多计划。

算法（algorithm）

一组指示，其利用机械化、非智能性的流程来解决一种类别的问题。

14 类别与方法

14.1 对象导向功能

Python 是一种**对象导向程序语言**，这表示它提供了支持 **对象导向程序设计**的功能。

要定义对象导向程序设计并不容易，不过我们已经看过它的一些特色：

- 程序是由对象定义和函数定义组成，而且大多数的计算以对象的操作来表示。
- 每一个对象定义都与一些真实世界中的对象或概念相应，而作用于该对象的函数则与真实对象的互动方式相应。

与例来说，在 十三章中定义的 Time 类别和人们记录时间的方式相应，而我们所定义的函数则和人们使用时间的方式相应。同样地，Point 和Rectangle 类别和数学生点与矩形的概念相应。

到目前为止，我们还没有利用 Python 提供的功能来支持对象导向程序设计。严格说起来，这些功能并不是必要的。在大多数时候，它们为我们之前已经做好的事提供另一种语法，但是在许多情况下，这种替代语法更为简洁而且能与精准地传达程序的结构。

例如，在 Time 程序中，在类别定义和其后的函数定义简，并没有明显的关联性。在经过一些检查后，很明显地，每一个函数至少会使用一个 Time 对象当成自变量。

这种观察就是**方法**的原点。我们已经看过一些方法，例如在讲到 dictionaries 时所使用的 keys 和 values。每个方法都与一个类别相关，而且常在该类别的范例中使用。

方法就像函数一样，不过有两个不同：

- 方法是定义在一个类别定义中，好让类别与方法之间的关系清楚明白。
- 使用一个方法的语法和呼叫一个函数的语法不同。

在以下几节中，我们会将前两章的函数转换成方法。这种转换非常机械化；你只要按照顺序或步骤来进行即可。如果你能得心应手地从一种形式转换到另一种，无论你要做什么，你将能够选择最好的形式。

14.2 printTime

在 第十三章 中，我们定义了一个名为 Time 的类别，而且你写了一个名为 printTime 函数，它看起来应该像这样：

```
class Time:
  pass


def printTime(time):
  print str(time.hours) + ":" + \
        str(time.minutes) + ":" + \
        str(time.seconds)
```

要呼叫这个函数，我们会传送一个 Time 对象作为自变量：

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
```

要把 printTime 做成一个方法，我们只要将这个函数定义移到类别定义中即可。请注意要改变缩排方式。

```
class Time:
  def printTime(time):
    print str(time.hours) + ":" +  \
          str(time.minutes) + ":" +  \
          str(time.seconds)
```

现在我们可以利用句点符号来使用 printTime 了。

```
>>> currentTime.printTime()
```

一如以往，使用方法的对象出现在句点前面，而方法的名称则出现在句点的后现。

要使用方法的对象会指派到第一个参数，所以在这里 currentTime 被指派为参数 time。

一般来说，一个方法的第一个参数称为 self。原因有点复杂，不过是基于一个有用的隐喻。

一个呼叫函数的语法，printTime(currentTime)，表示这个函数是个活性剂。它表示：「嘿，printTime！这里有个对象让你印出。」

在对象导向程序设计中，对象就是活性剂。一个像 currentTime.printTime() 的使用方法则说「嘿！currentTime！请印出你自己！」

观点上的的改变可能较为礼貌，不过不明显的是它也是很有用的。在我们已经看过的范例中可能不是这样。不过有时间将责任从函数移转到对象上，让我们可能写出更有弹性的函数来，而且使得维持和重新使用程序代码更为容易。

14.3 另一个范例

让我们把 increment（第 13.3 节中的）转换成一个方法。为了节省空间，我们会排除之前已经定义好的方法，不过你应该在自己的版本中保留它们：

```
class Time:
  #previous method definitions here...


  def increment(self, seconds):
    self.seconds = seconds + self.seconds


    while self.seconds >= 60:
      self.seconds = self.seconds - 60
      self.minutes = self.minutes + 1


    while self.minutes >= 60:
      self.minutes = self.minutes - 60
      self.hours = self.hours + 1
```

这个转换是纯粹机械化的动作——我们将方法定义移到类别定义中，然后改变第一个参数的名称。

现在，我们可以使用 increment 作为一个方法了。

```
currentTime.increment(500)
```

另外，要使用方法的对象会指派到一个参数 self。第二个参数，seconds，则得到 500 的值。

作为练习，请将 convertToSeconds（在 13.5 节中的）转换成一个 Time 类别中的方法。

14.4 一个更为复杂的范例

after 函数则较为复杂，因为它在两个 Time 对象上执行，而不是一个。我们只能将一个参数转换成 self；另一个则保持原状：

```
class Time:
  #previous method definitions here...


  def after(self, time2):
    if self.hour > time2.hour:
      return 1
    if self.hour < time2.hour:
      return 0


    if self.minute > time2.minute:
      return 1
    if self.minute < time2.minute:
      return 0
```

```
    if self.second > time2.second:
      return 1
    return 0
```

我们在一个对象上使用这个方法，然后传送另一个当成自变量：

```
if doneTime.after(currentTime):
  print "The bread is not done yet."
```

你几乎可以把这个使用方法当成英文来读：「如果 done.time 在 current.time 之后，则……」。

14.5 选择性自变量

我们已经看过一些内建函数可以接受不同数量的自变量。举例来说，string.find 可以接受两个、三个或四个自变量。

要撰写一个有选择性自变量列表的使用者定义函数是可能的。举例来说，我们可以把我们自己的 find 版本升级到和 string.find 的功能一样。

以下是 7.7 节中的原始版本：

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

而这是新的改良版本：

```
def find(str, ch, start=0):
  index = start
```

```
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

第三个参数，start 是选择性的，因为我们提供了一个默认值 0。如果我们只利用两个自变量来使用 find，它会使用默认值，并从字符串的最前面开始：

```
>>> find("apple", "p")
1
```

如果我们提供了第三个自变量，它会**复盖掉**默认值：

```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```

作为练习，请加入第四个参数，*end*，来指定搜寻停止的地方。

警告：这个练习需要一点技巧。*end* 的默认值应该为 *len(str)*，不过这样没办法运作。默认值是在函数定义时计算的，而不是当它被呼叫时才计算的。当我们定义 *find* 的时候，*str* 尚未存在，所以你无法找到它的长度。

14.6 初始化方法

初始化方法是一种特别的方法，它是在一个对象被创造时使用的。这个方法的名称是 __init__（两个底线符号，后面接着 init，然后再接两个底线符号）。Time 类别中的初始化方法看起来像这样：

```
class Time:
  def __init__(self, hours=0, minutes=0, seconds=0):
```

```
    self.hours = hours
    self.minutes = minutes
    self.seconds = seconds
```

属性 self.hours 和参数 hours 之间并没有冲突。句点符号则指定我们要的是哪一个变量。

当我们使用 Time 建构子时，我们提供的自变量会一起传送到 init：

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
9:14:30
```

因为这些自变量是选择性的，所以我们可以忽略它们：

```
>>> currentTime = Time()
>>> currentTime.printTime()
0:0:0
```

或者只提供第一个：

```
>>> currentTime = Time (9)
>>> currentTime.printTime()
9:0:0
```

或前两个：

```
>>> currentTime = Time (9, 14)
>>> currentTime.printTime()
9:14:0
```

最后，我们可以明确地命名它们，好一次指派一组参数：

```
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
```

9:0:30

14.7 重新审视 Points

让我们重新撰写 12.1 节中的 Point 类别，让它更具对象导向的风格。

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

初始化方法接受 x 和 y 值作为选择性的参数；这两个参数的默认值皆为 0。

下一个方法，__str__ 会传回一个代表一个 Point 对象的字符串。如果一个类别提供一个名为 __str__ 的方法，它会复盖掉 Python 内建的 str 函数的行为模式。

```
>>> p = Point(3, 4)
>>> str(p)
'(3, 4)'
```

印出一个 Point 对象就暗示了在这个对象上使用 __str__，所以定义__str__ 的同时也改变了 print 的行为模式：

```
>>> p = Point(3, 4)
>>> print p
(3, 4)
```

当我们撰写一个新的类别时，我们几乎总是以撰写 __init__ 和 __str__开始，前者让说明对象更容易，后者则几乎总是对除虫非常有帮助。

14.8 运算子多载

一些语言可以让你在内建运算子运用在使用者定义型态时，改变它们的定义。这种功能称为**运算子多载（operator overloading）**。这在定义数学型态时特别有用。

举例来说，为了覆盖加法运算子 +，我们提供一个名为 __add__ 的方法：

```
class Point:

  # previously defined methods here...


  def __add__(self, other):

    return Point(self.x + other.x, self.y + other.y)
```

一如以往，第一个参数是使用这个方法的对象。第二个参数则方便地命名为 other ，好和 self 不同。为了加上两个 Point，我们建立和传回一个新的 Point，其中包含 x 坐标的总和与 y 坐标的总合。

现在，当我们将 + 运算子运用在 Point 对象上时，Python 会使用__add__：

```
>>>   p1 = Point(3, 4)

>>>   p2 = Point(5, 7)

>>>   p3 = p1 + p2

>>>   print p3

(8, 11)
```

表达式 p1 + p2 与 p1.__add__(p2) 相等，但是很明显地更为优雅。

作为练习，加入一个 __sub__(self, other) 方法，让它多载减法运算子，并试用这个方法。

我们有多种方法可以多载乘法运算子的行为方式：我们可以定义一个名为 __mul__ 或 __rmul__ 的方法，或同时使用两者。

在 * 左侧的操作数是一个 Point，Python 使用 __mul__ 方法，它会假设另一个操作数也同样是个 Point。它会计算以线性代数规则定义的两个点的**点积（dot product）**。

```
def __mul__(self, other):
```

```
  return self.x * other.x + self.y * other.y
```

如果 * 的左侧操作数是一个原始形态，而右侧操作数是个 Point，Python 会使用 __rmul__，它会执行**纯量乘法（scalar multiplication）**：

```
def __rmul__(self, other):
  return Point(other * self.x,  other * self.y)
```

结果会是一个新的 Point，它的坐标会是原有坐标的倍数。如果 other 是一个无法乘以浮点数的形态，那么 __rmul__ 就会产生错误讯息：

以下是两种乘法的范例：

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

如果我们试着计算 p2 * 2 会发生什么事呢？因为第一个操作数是一个 Point，Python 使用__mul__ 方法，并以 2 作为第二个自变量。在 __mul__方法中，这个程序试着存取 other 的 x 坐标，不过因为一个整数没有属性而失败。

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

不幸的是，这个错误讯息有点难以理解。这个范例展示了一些对象导向程序设计的困难。有时候光是要判断哪段程序代码在运作已经很难了。

关于运算子多载更复杂的的范例，请参阅 附件 B。

14.9 多形（Polymorphism）

我们之前撰写的方法大多数只能运用在一个特定形态。当你建立了一个新的对象，你会撰写运作在该形态之上的方法。

不过你会希望某些操作能应用在多种形态上，例如前一节中的四则运算操作。如果有多种形态支持同一组操作，你可以撰写一组可以在任何这些形态中运作的函数。

举例来说， multadd 运算（这在线性代数很常见）会接受三个自变量；它会将前两个自变量相乘，并加上第三个自变量。我们可以在 Python 中这么撰写它：

```
def multadd (x, y, z):
  return x * y + z
```

这个方法可以运作在任何 x 和 y 可以相乘的值，和任何 z 可以和积相加的值。

我们可以使用在数值上：

```
>>> multadd (3, 2, 1)
7
```

也可以用在 Point 上：

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print multadd (2, p1, p2)
(11, 15)
>>> print multadd (p1, p2, 1)
44
```

在第一个情况下，Point 乘以一个纯量，然后加到另一个 Point。在第二个情况下，点积会产出一个数值，所以第三个自变量也必须是一个数值。

一个像这样可以接受不同形态自变量的函数就称为**多形（polymorphic）**。

我们来看另一个范例，想一下 frontAndBack 这个方法，它会印出一个列表两次，第一次正向，第二次反向：

```
def frontAndBack(front):
  import copy
  back = copy.copy(front)
```

```
  back.reverse()

  print str(front) + str(back)
```

因为 reverse 方法是一个修饰子，我们在反转列表以前做了一个备份。这样，这个方法就不会更改它接受作为自变量的列表。

以下是一个应用 frontAndBack 在一个列表上的范例：

```
>>>   myList = [1, 2, 3, 4]

>>>   frontAndBack(myList)

[1, 2, 3, 4][4, 3, 2, 1]
```

当然，我们就是要应用这个函式到列表上，所以它可以成功运作也就不让人讶异了。如果我们可以将它用在一个 Point 上才会令人讶异。

为了要确定一个函数是否可以用在一个新形态上，我们会应用多形的基本规则。

如果函数中所有运算都可以应用在这个形态上，这个函数就可以应用在这个形态上。

这个方法中的运算包含 copy、reverse、和 print。

copy 可以在任何对象作运作，而我们已经为 Point 撰写了一个 __str__方法，所以我们所需要的只是一个在 Point 类别下的 reverse 方法：

```
def reverse(self):

  self.x , self.y = self.y, self.x
```

然后我们就可以传送 Point 到 frontAndBack 中：

```
>>>   p = Point(3, 4)

>>>   frontAndBack(p)

(3, 4)(4, 3)
```

最好的多形种类是那种无意中造成的，你发现你已经写好的一个函数可以运用在你从未计划好的形态上。

14.10 术语

对象导向语言（object-oriented language）

一种程序语言，提供使用者定义类别和继承等有利于对象导向程序设计的功能。

对象导向程序设计（object-oriented programming）

一种程序设计风格，在其中数据和操作数据的运算都组织成类别和方法。

方法（method）

一个定义在类别定义中并使用在该类别范例上的函数，

覆盖（override）

取代一个默认值。范例包含用一个特定自变量取代默认值，和提供一个有相同名称的新方法来取代预设的方法。

初始化方法（initialization method）

一种在一个新对象建立时自动执行的的特别方法，它会初始化该对象的属性。

运算子多载（operator overloading）

延伸内建运算子（+、-、*、>、< 等），使它们可以在使用者定义型态中运作。

点积（dot product）

一种在线性代数中定义的运算，这种运算会将两点相乘，并产生一个数值。

纯量乘法（scalar multiplication）

一种在线性代数中定义的运算，它会将一个点的坐标各乘以一个数值。

多形（polymorphic）

一个可以在多于一种形态上运作的函数。如果一个函数中的所有运算可以在一种形态中运作，那么这个函数就可以在这种形态中运作。