A Byte of Python

v1.92(for Python 3.0)

Written by：Swaroop, C. H.

Translated by：Let it be!

e-mail：329974248@qq.com

Welcome to Exchange!

Date：2011.7.9

Python

《A Byte of Python》是一本关于用 Python 语言编程的书。可以作为初学这的入门教程。也可以供计算机相关人员参考。

翻译

有许多该书翻译的不同语言的版本，感谢那些不辞辛劳的志愿者。

如果你对阅读感兴趣或者志愿翻译这本书其他语言的翻译工作，请自行下载英文版《A Type of Python》参考“翻译”一节。

由于最新的《A Type of Python》已经更新至 Version1.92(for Python 3.0)，但作为唯一指定简体中文译本《简明 Python 教程》还未更新，处于本人的兴趣，翻译了本书。

本书的翻译参考中译本《简明 Python 教程》（Version:1.2）（沈洁元）。本书的翻译和排版的原则是结构清晰，突出重点。

目录

第 1 章 前言

Python 可能是极少数既简单有强大的编程语言中的一种。这对初学者和专家都是好事，更重要的是，用它来编程是非常快乐的事。本书的目标就是帮助你学习这门奇妙的语言，展示出如何方便快捷地完成任务 —— 实质上“为你的编程问题提供完美的解决方案”。

1.1 本书适合的读者

本书可作为 Python 编程语言的指导或辅导。主要是针对新手的，当然，对于有经验的程序员也很有用。

如果你所了解的计算机的知识就是如何保存文本文件，那么你就能从本书开始学习 Python 。如果你先前有编程经验，那么你也可以从本书来开始学习 Python 。

如果你有先前先前的编程经验，你将对 Python 和你喜欢的编程语言之间的差别感兴趣。顺便提醒一下， Python 会很快变成你喜欢的编程语言！

1.2 本书的来历

我最初接触 Python 是当我需要为我的软件钻石写一个方便安装过程的安装程序的时候。我得在 Python 和 Perl 语言中选择一个绑定 Qt 库。我在网上做了一些研究，偶然发现了一篇文章。那是 Eric S. Raymond ，一个著名而又受人尊敬的黑客，谈 Python 如何成为他最喜欢地编程语言的一篇文章。我同时发现 PyQt 绑定与 Perl-Qt 相比要出色得多，所以我选择了 Python 语言。之后我开始寻找一本关于 Python 的优秀书籍。我竟然找不到！虽然我找到了一些 O’Reilly 的书，不过它们不是太贵就是如同一本参考手册而不是一本指南。我最后使用了 Python 附带的文档，不过它太简略了。那个文档确实很好的给出了 Python 的概念，不过不够全面。尽管最后我根据我以前得编程经验掌握了那个文档，不过我觉得它完全不适合于新手。

大约在我首次使用 Python 语言的六个月之后，我安装了那时最新的 Red Hat 9.0 Linux 。在我玩弄 KWord 应用程序的时候，我突然想写一点关于 Python 的东西。很快我就写了 30 多页，然后我开始认真地想办法把它变成一本完整的书。经过多次的改进和重写，它终于成为了一本有用的完整的 Python 语言学习指南。我把本书贡献给开源软件者们。

本书来自于我个人学习 Python 的笔记，不过我尽力让它更加适合别人的口味。:）

在开源精神的鼓舞下，我收到了许多建设性的建议和批评以及来自热心读者的反馈，它们使这本书变得更加出色。

1.3 本书的状况

从 2005 年 3 月较多的修订以来，针对 Python 3.0 发行版（预期在 2008 年 8 月/9 月）许多变化也更新了。由于 Python 3.0 语言自身仍然未完成/发布，该书也在不断变化。

然而，在开源哲学“早发布，常发布”的精神鼓舞下，书也不断发布，不断更新。

本书需要想你一样的读者的帮助来指出书中不好的部分，最好不是能理解的或简单的错误。请给作者（http://www.swaroopch.com/contact/或者各个翻译者提出你们自己的意见和建议。

在初学者的需求和倾向于信息的完整性二者之间做衡量通常充满矛盾。如果读者能反馈关于该书所应有的深度方面的信息，对本书也将非常有益。

1.4 官方网站

本书的官方网站是 http://www.swaroopch.com/notes/Python ，在这里可以在线阅读整本书，下载本书的最新版，买到打印的版本，也可给我发回反馈信息。

1.5 许可

参照英文原版《A Type of Python》“preface”--“License”。

1.6 反馈

我尽了很大的力让这本书即生动又尽可能的准确。然而，如果你找到任何不太令你满意的地方或者错误，或者是需要改进的地方，请告诉我以便我改正它们。你可以通过用户页面给我反馈。

1.7 值得考虑的一些东西

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies.
有两种方式构建软件设计：一种是把软件做得很简单以至于明显找不到缺陷；另一种是把它做得很复杂以至于找不到明显的缺陷。

——C. A. R. Hoare

Success in life is a matter not so much of talent and opportunity as of concentration and perseverance.
获得人生中的成功需要的专注与坚持不懈多过天才与机会。

——C. W. Wendte

第 2 章 Python 介绍

2.1 简介

Python 语言是少有的一种可以称得上即简单又功能强大的编程语言。你将惊喜地发现 Python 语言是多么地简单，它注重的是如何解决问题而不是编程语言的语法和结构。

- Python 语言的官方介绍是：
  Python 是一种简单易学，功能强大的编程语言，它有高效率的高层数据结构，简单而有效地实现面向对象编程。 Python 简洁的语法和对动态输入的支持，再加上解释性语言的本质，使得它在大多数平台上的许多领域都是一个理想的脚本语言，特别适用于快速的应用程序开发。

  我会在下一节里详细地讨论 Python 的这些特点。

注释：

Python 语言的创造者 Guido van Rossum 是根据英国广播公司的节目“蟒蛇飞行马戏”命名这个语言的，并非他本人特别喜欢蛇缠起它们的长身躯碾死动物觅食。

2.2 Python 的特点

1. 简单
   Python 是一种代表简单主义思想的语言。阅读一个良好的 Python 程序就感觉像是在读英语一样，尽管这个英语的要求非常严格！ Python 的这种伪代码本质是它最大的优点之一。它使你能够专注于解决问题而不是去搞明白语言本身。

2. 易学
   就如同你即将看到的一样， Python 极其容易上手。前面已经提到了， Python 有极其简单的语法。

3. 免费、开源
   Python 是 FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。 FLOSS 是基于一个团体分享知识的概念。这是为什么 Python 如此优秀的原因之一 —— 它是由一群希望看到一个更加优秀的 Python 的人创造并经常改进着的。

4. 高层语言
   当你用 Python 语言编写程序的时候，你无需考虑诸如如何管理你的程序使用的内存一类的底层细节。

5. 可移植性
   由于它的开源本质， Python 已经被移植在许多平台上（经过改动使它能够工

作在不同平台上)。如果你小心地避免使用依赖于系统的特性，那么你的所有 Python 程序无需修改就可以在下述任何平台上面运行。
这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE 甚至还有 Pocket-PC!

6. 解释性
   这一点需要一些解释。
   一个用编译性语言比如 C 或 C++ 写的程序可以从源文件（即 C 或 C++ 语言）转换到一个你的计算机使用的语言（二进制代码，即 0 和 1)。这个过程通过编译器和不同的标记、选项完成。当你运行你的程序的时候，连接/转载器软件把你的程序从硬盘复制到内存中并且运行。
   而 Python 语言写的程序不需要编译成二进制代码。你可以直接从源代码运行程序。在计算机内部， Python 解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。事实上，由于你不再需要担心如何编译程序，如何确保连接转载正确的库等等，所有这一切使得使用 Python 更加简单。由于你只需要把你的 Python 程序拷贝到另外一台计算机上，它就可以工作了，这也使得你的 Python 程序更加易于移植。

7. 面向对象
   Python 即支持面向过程的编程也支持面向对象的编程。在面向过程的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在面向对象的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如 C++ 和 Java 相比， Python 以一种非常强大又简单的方式实现面向对象编程。

8. 可扩展性
   如果你需要你的一段关键代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用 C 或 C++ 编写，然后在你的 Python 程序中使用它们。

9. 可嵌入性
   你可以把 Python 嵌入你的 C/C++ 程序，从而向你的程序用户提供脚本功能。

10. 丰富的库
    Python 标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI（图形用户界面)、Tk 和其他与系统有关的操作。记住，只要安装了 Python ，所有这些功能都是可用的。这被称作 Python 的“功能齐全”理念。
    除了标准库以外，还有许多其他高质量的库，如 wxPython 、Twisted 和 Python 图像库等等。

Python 确实是一种十分精彩又强大的语言。它合理地结合了高性能与使得编写程序简单有趣的特色。

2.3 为什么不选 Perl?

也许你以前并不知道，Perl 是另外一种极其流行的开源解释性编程语言。

如果你曾经尝试过用 Perl 语言编写一个大程序，你一定会自己回答这个问题。在规模较小的时候， Perl 程序是简单的。它可以胜任于小型的应用程序和脚本，“使工作完成”。然而，当你想开始写一些大一点的程序的时候， Perl 程序就变得不实用了。我是通过为 Yahoo 编写大型 Perl 程序的经验得出这样的总结的!

与 Perl 相比， Python 程序一定会更简单、更清晰、更易于编写，从而也更加易懂、易维护。我确实也很喜欢 Perl，用它来做一些日常的各种事情。不过当我要写一个程序的时候，我总是想到使用 Python ，这对我来说已经成了十分自然的事。Perl 已经经历了多次大的修正和改变，遗憾的是，即将发布的 Perl 6 似乎仍然没有在这个方面做什么改进。我感到 Perl 唯一也是十分重要的优势是它庞大的 CPAN 库——综合 Perl 存档网络。就如同这个名字所指的意思一样，这是一个巨大的 Perl 模块集，它大得让人难以置信 —— 你几乎用这些模块在计算机上做任何事情。 Perl 的模块比 Python 多的原因之一是 Perl 拥有更加悠久的历史。当然，随着 Python 包索引的增加这也在不断变化。

2.4 为什么不选 Ruby?

也许你以前并不知道， Ruby 是另外一种极其流行的开源解释性编程语言。

如果你已喜欢并且已经使用 Ruby ，那么我明确地给你说应该继续使用它。

对那些没使用过 Ruby ，正试着判断该学 Python 还是 Ruby 的人来讲，我会推荐 Python ，纯粹是因为从易学的角度来考虑。就我个人而言，让我来喜欢 Ruby 是很困难的事情，但对理解 Ruby 的人来说，他们总是会赞赏 Ruby 的优美。非常不幸，我不是这种幸运儿。

2.5 程序员都说些什么

读一下像 ESR 这样的超级电脑高手谈 Python 的话，你会感到十分有意思:

- Eric S. Raymond 是《The Cathedral and the Bazaar》的作者、“开放源码”一词的提出人。他说 Python 已经成为了他最喜爱的编程语言。这篇文章也是促使我第一次接触 Python 的真正原动力。
- Bruce Eckel 著名的《Thinking in Java》和《Thinking in C++》的作者。他说没有一种语言比得上 Python 使他的工作效率如此之高。同时他说 Python 可能是唯一一种旨在帮助程序员把事情弄得更加简单的语言。请阅读完整的采访以获得更详细的内容。

- Peter Norvig 是著名的 Lisp 语言书籍的作者和 Google 公司的搜索质量主任（感谢 Guido van Rossum 告诉我这一点）。他说 Python 始终是 Google 的主要部分。事实上你看一下 Google 招聘的网页就可以验证这一点。在那个网页上，Python 知识是对软件工程师的一个必需要求。

2.6 关于 Python 3.0

Python 3.0 是该编程语言的新版。有时又被用为 Python 3000 或 Py3K 。

本版更新的主要原因就是为了除去过去几年积累的一些问题，并且使得本语言更清晰。

如果你已有大量的 Python 2.x 的代码，那么这儿有个工具能帮助你将 2.x 转换成 3.x 的源代码。（http://docs.python.org/dev/3.0/library/2to3.html）
更多细节：

- Guido van Rossum's introduction
  (http://www.artima.com/weblogs/viewpost.jsp?thread=208549)
- What's New in Python 2.6
  (http://docs.python.org/dev/whatsnew/2.6.html)
  (features significantly different from previous Python 2.x versions and most likely will be included in Python 3.0)
- What's New in Python 3.0
  (http://docs.python.org/dev/3.0/whatsnew/3.0.html)
- Python 2.6 and 3.0 Release Schedule
  (http://www.python.org/dev/peps/pep-0361/)
- Python 3000 (the official authoritative list of proposed changes)
  (http://www.python.org/dev/peps/pep-3000/)
- Miscellaneous Python 3.0 Plans
  (http://www.python.org/dev/peps/pep-3100/)
- Python News (detailed list of changes)
  (http://www.python.org/download/releases/3.0/NEWS.txt)

第 3 章 安装

如果你已经安装了 Python 2.x，那么就没必要卸载后再安装 Python 3.x，实际上，可以将它们同时安装在电脑里。

3.1 对于 Linux 和 BSD 用户

如果你正在使用一个 Linux 的发行版比如 Fedora 或者 Mandrake 或者其他（你的选择），或者一个 BSD 系统比如 FreeBSD，那么你可能已经在你的系统里安装了 Python。

要测试你是否已经随着你的 Linux 包安装了 Python，你可以打开一个 shell 程序（就像 konsole 或 gnome-terminal）然后输入如下所示的命令 python -V。

```
$ python -v
python 3.0b1
```

注释：

$ 是 shell 的提示符。根据你的操作系统的设置，它可能与你那个不同，因此我只用 $ 符号表示提示符。如果你看见向上面所示的那样一些版本信息，那么你已经安装了 Python 了。如果你得到像这样的消息：

```
$ python -V
bash: Python: command not found
```

那么 Python 还没有安装，这几乎不可能，只是极其偶尔才会遇到。

注释：

如果已经安装了 Python 2.x，那试一试 python3 -V。

在这种情况下，你有两种方法在你的系统上安装 Python。

- 你可以从源码编译 Python（http://www.python.org/dowload/releases/3.0）然后安装。在网上有编译的指令。
- [这个选项在 Python 3.0 之后才能使用] 利用你的操作系统附带的包管理软件安装二进制包，例如，在 Ubuntu/Debian 和以 Debian 为基础的 Linux 系统上，用 apt-get，在 FreeBSD 上用 pkg-add，等等。注意，使用给方法时需要联网。
  你也可从其它任何地方下载二进制包然后复制到你的电脑中进行安装。

3.2 对于 Windows 用户

访问 http://www.python.org 网站下载最新版，在写本书的时候是 3.0 beta 1。仅有 12.8MB，与大多其它的语言或软件相比，是非常紧凑的。安装与其它的 Windows 软件一样。

注意：

即便安装程序为你提供了不检查可选组件的选项，你也不要不作任何检查！有些组件对你很有用，特别是集成开发环境。

有趣的是，大多的 Python 下载是来自 Windows 用户的。当然，这并不能说明问题，因为几乎所有的 Linux 用户已经在安装系统的时候默认安装了 Python 。

3.3 DOS 命令提示符

如果想通过 Windows 命令行，如 DOS 命令提示符来使用 Python ，就需要正确设置环境变量 PATH 。

3.4 对于 Mac OS X 用户

Mac OS X 用户会发现已经在系统中安装了 Python 。打开 Terminal.app 运行 python -V ，接着参考上面关于 Linux 部分的建议。

3.5 概括

对于 Linux 系统，很可能你已经在你的系统里安装了 Python 。否则，你可以通过你的发行版附带的包管理软件安装 Python 。对于 Windows 系统，安装 Python 就是下载安装程序然后双击它那么简单。从现在起，我们将假设你已经在你的系统里安装了 Python 。

接下来，就写我们的第一个 Python 程序。

第 4 章 最初的步骤

4.1 简介

我们将看一下如何用 Python 编写运行一个传统的“Hello World”程序。通过它，你将学会如何编写、保存和运行 Python 程序。

有两种使用 Python 运行你的程序的方式 —— 使用交互式的带提示符的解释器或使用源文件。我们将学习这两种方法。

4.2 使用带提示符的解释器

在 shell 提示符下，键入 Python 命令启动解释器。

对 Windows 用户，如果你已经配置好了 PATH 变量，那么就可在命令行中启动解释器。

如果使用 IDLE，点击 开始 ⟶ 程序 ⟶ Python 3.0 ⟶ IDLE (Python GUI)。

键入 print('Hello World')，按回车键。将会看到 Hello World 字样的输出。

```
$ python
Python 3.0b2 (r30b2:65106, Jul 18 2008, 18:44:17) [MSC v.
   1500 32 bit(Intel)] on win32 Type "help", "copyright", "
   credits" or "license" for more information.
>>>print('Hello World')
Hello World
>>>
```

注意，Python 会在下一行立即给出你输出！你刚才键入的是一句 Python 语句。我们使用 print（不要惊讶）来打印你提供给它的值。这里，我们提供的是文本 Hello World，它被迅速地打印在屏幕上。

如何退出解释器提示符

如果你使用的是 Linux/BSD shell，那么按 Ctrl-d 退出提示符。如果是在 Windows 命令行中，则按 Ctrl-z 再按 Enter。

4.3 选择一个编辑器

用 Python 写程序源文件之前，需要一个编辑器。对与编辑器的选择确实非常重要。选择一个编辑器就像你买车一样。一个好的编辑器能帮助你容易地编写 Python 程序，使你的编程之旅更加舒适，容易、安全地达到你的目的（达到你的目标）。

最基本的要求就是能**语法高亮**，Python 程序不同的部分有不同的颜色，这样就能使你看清你的程序使其更形象。

如果你用的是 Windows，我建议你使用 IDLE。IDLE 有语法高亮，还有许多其他的功能，比如允许你在 IDLE 中运行你的程序。特别值得注意的是：**不要使用 Notepad** —— 它是一个糟糕的选择，因为它没有语法加亮功能，而且更加重要的

是，它不支持文本缩进。而我们将会看到文本缩进对于我们来说极其重要。一个好的编辑器，比如 IDLE（还有 VIM）将会自动帮助你做这些事情。

如果你使用的是 Linux/FreeBSD，那就有很多的选择。如果你是程序的初学者，你或许会选择 geany。有图形用户界面，有编译、运行按钮，很容易运行 Python 程序。

如果你是一个有经验的程序员，你一定已经会使用 Vim 或 Emacs。毋庸置疑，这是两个最强大的编辑器，用它们来编写 Python 程序时，你会从中受益。我个人使用 Vim 编写大多数程序。如果你是初学编程的人，可以使用 Kate，这也是我喜欢的编辑器之一。如果你想花时间学习使用 Vim 或 Emacs，我强烈推荐你学习二者之一，因为从长远考虑，它对你非常有用。

在这本书中，使用的是 IDLE，我们的 IDE 和选择的编辑器。IDLE 在 Windows 和 Mac OS X 中的 Python 安装程序中是默认安装的。对于 LInux 和 BSDs，在各自的库中，也能得到安装程序。

在下一节中会研究如何使用 IDLE。更多细节，请参考 IDLE 的文档。

如果你还想寻找一下其他可供选择的编辑器，可以看一下详尽的 Python 编辑器列表，然后作出你的选择。你也可以使用 Python 的 IDE（集成开发环境）。请看一下详尽的支持 Python 的 IDE 列表以获得详尽的信息。一旦你开始编写大型的 Python 程序，IDE 确实很有用。

我再一次重申，请选择一个合适的编辑器——它能使编写 Python 程序变得更加有趣、方便。

对于 Vim 用户

John M Anderson 有一篇关于如何将 Vim 变成一个强大的 Python IDE 的好文章（http://blog.sontek.net/2008/05/11/python-with-a-modular-ide-vim/）

对于 Emacs 用户

Ryan McGuire 有一篇关于如何将 Vim 变成一个强大的 Python IDE 的好文章（http://www.enigmacurry.com/2008/05/09/emacs-as-a-powerful-python-ide/）

4.4 使用源文件

现在，回到编程上来。有个惯例，就是当学习一门新语言的时候，第一个程序就是编写、运行‘Hello World’程序 —— 运行该程序的时候输出‘Hello World’。这如 Simon Cozens 提到的那样，“它是编程之神的传统咒语，可以帮助你更好地学习语言”:)。

启动你选择的编辑器，输入下面的程序保存为 helloworld.py。如果你使用的是 IDLE，点击 File → New Window 然后输入下面的程序。点击 File → Save。

```
1 #!/usr/bin/python
2 #Filename: helloworld.py
3 print('Hello World')
```

打开 shell（LInux 终端或 DOS 命令提示符），键入 python helloworld.py 来运行程序。

如果用的是 IDLE，使用菜单栏 Run → Run Module 或者快捷键 F5 来运行程序。

输出如下所示：

```
$ python helloworld.py
Hello World
```

如果你得到的输出与上面所示的一样，那么恭喜！—— 你已经成功地运行了你的第一个 Python 程序。

万一你得到一个错误，那么请确保你键入的程序准确无误，然后再运行一下程序。注意 Python 是大小写敏感的，即 print 与 Print 不一样 —— 注意前一个是小写 p 而后一个是大写 P。另外，确保在每一行的开始字符前没有空格或者制表符 —— 我们将在后面讨论为什么这点是重要的。

4.5 它是如何工作的？

让我们思考一下这个程序的前两行。它们被称作注释 —— 任何在 # 符号右面的内容都是注释。注释主要作为提供给程序读者的笔记。

Python 至少应当有第一行那样的特殊形式的注释。它被称作组织行 —— 源文件的头两个字符是 #!，后面跟着一个程序。这行告诉你的 Linux/Unix 系统当你执行你的程序的时候，它应该运行哪个解释器。这会在下一节做详细解释。注意，你总是可以通过直接在命令行指定解释器，从而在任何平台上运行你的程序。就如同命令 python helloworld.py 一样。

重要的：

在你的程序中合理地使用注释以解释一些重要的细节 —— 这将有助于你的程序的读者轻松地理解程序在干什么。记住，这个读者可能就是 6 个月以后的你！

跟在注释之后的是一句 Python 语句 —— 它只是打印文本“Hello World”。print 实际上是一个操作符，而“Hello World”被称为一个字符串 —— 别担心我们会在后面详细解释这些术语。

4.6 可执行的 Python 程序

这部分内容只对 Linux/Unix 用户适用，不过 Windows 用户可能也对程序的第一行比较好奇。首先我们需要通过 chmod 命令，给程序可执行的许可，然后运行程序。

```
$ chmod a+x helloworld.py
$ ./helloworld.py
```

```
3 Hello World
```

chmod 命令用来改变文件的模式，给系统中所有用户这个源文件的执行许可。然后我们可以直接通过指定源文件的位置来执行程序。我们使用./来指示程序位于当前目录。

为了更加有趣一些，你可以把你的文件名改成仅仅 helloworld，然后运行 ./helloworld。这样，这个程序仍然可以工作，因为系统知道它必须用源文件第一行指定的那个解释器来运行程序。

如果你不知道 Python 的位置该怎么办呢？于是，可以用在 Linux/Unix 上特殊的 env 程序。仅仅改变程序中的第一行：

```
1 #!/usr/bin/env python
```

env 程序会反过来寻找会运行程序的 Python 解释器。

到此，只要我们知道准确的路径，就已经能够运行程序了。如果想在任何地方都能运行程序该怎么办呢？你可以将这个程序保存在已经存在的环境变量 PATH 所列的路径中。不管在什么时候，运行程序的时候，系统会检查在 PATH 环境变量中所列的所有路径，然后运行该程序。我们可以将源文件拷贝到任何所列的环境变量 PATH 的路径中都能运行。

```
1 $ echo $PATH
2 /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
3 $ cp helloworld.py /home/swaroop/bin/helloworld
4 $ helloworld
5 Hello World
```

可以用 echo 命令来显示 PATH 变量，在变量名前附加 $ 来给 shell 表明我们需要该变量的值。可以看到 /home/swaroop/bing 是其中的一个路径，这里 swaroop 是我正在使用的系统的用户名。在你的系统中也会有以你的用户名命名的类似路径。当然，你也可以给你的 PATH 变量增加你自己选择的路径 —— 这可以通过运行 PATH=$PATH:/home/swaroop/mydir 命令完成，这儿‘/home/swaroop/mydir’是你想加入 PATH 变量的路径。

当你想要在任何时间、任何地方运行你的程序的时候，这个方法十分有用。它就好像创造你自己的指令，如同 cd 或其他 Linux 终端或 DOS 提示符命令那样。

注意：

对于 Python 来说，程序、脚本或者软件都是指同一个东西。

4.7 获得帮助

在 Python 中，如果你想得到任何关于函数或语句的快速信息帮助，就可以使用内置的 help 函数。当你使用解释器提示符的时候，这个方法尤其有用。例如，运

行 help(print)（译者注：给方法在 Python 2.x 不能用，在 Python 3.x 中能用） —— 这回显示 print 函数的帮助信息， print 是用来在屏幕上打印信息的函数。

注释：

按 q 来退出帮助。

类似地，可以在 Python 中得到关于几乎任何东西的信息。用 help() 得到关于关于帮助的更多信息。

如果你需要得到关于类似 return 操作符的帮助，需要在内部加上引号，help('return')，这样 Python 就能理解你到底想干什么。

4.8 概括

现在，你应该能很容易地写出、保存并且能运行 Python 程序了。既然你是一个 Python 的使用者，那就让我们来学习更多关于 Python 的概念。

第 5 章 基础

仅能打印‘Hello World’还远远不够，难道不是吗？你像做得更多 —— 想得到一些输入，对其进行操作，然后再得到一些输出。在 Python 中可以使用常量和变量来实现。

5.1 字面意义上的常量

一个字面意义上的常量的例子是如同 5 、 1.23 、 9.25e-3 这样的数，或者如同 'This is a string' 、 "It'sa string!" 这样的字符串。它们被称作字面意义上的，因为它们具备字面的意义 —— 你按照它们的字面意义使用它们的值。数 2 总是代表它自己，而不会是别的什么东西 —— 它是一个常量，因为不能改变它的值。因此，所有这些都被称为字面意义上的常量。

5.2 数

在 Python 中数的类型有三种 —— 整数、浮点数和复数。

- 2 是一个整数的例子。
- 3.23 和 52.3E-4 是浮点数的例子。 E 标记表示 10 的幂。在这里， 52.3E-4 表示 $52.3 * 10^{-4}$。
- (-5+4j) 和 (2.3-4.6j) 是复数的例子。

给有经验的程序员的注释：

在 Python 中不用区分‘long int’类型。默认的整数类型可以任意长。（译者注：长度应该与内存包括虚拟内存的大小有关）

5.3 字符串

字符串是字符的序列。字符串基本上就是一组单词。单词可以是英语或其它由 Unicode 标准支持的语言，其实这也意味着世界上几乎所有的语言（http://www.unicode.org/faq/basic_q.html#16）。

给有经验的程序员的注释：

没有仅仅使用 ASCII 的字符串，原因是 Unicode 是 ASCII 的超集。如果要严格使用 ASCII 编码的字节流，可用 str.encode("ascii") 。更多细节，请访问在 StackOverflow 上的相关讨论。

默认所有的字符串的编码是 Unicode 。

我保证在你写的几乎每个 Python 程序中都会用到用到字符串，所以注意一下下一部分中关于如何在 Python 中使用字符串的内容。

5.4 单引号

你可以用单引号指定字符串，如 'Quote me on this' 。所有的空白，即空格和制表符都照原样保留。

5.5 双引号

在双引号中的字符串与单引号中的字符串的使用完全相同，例如 "What's your name?" 。

5.6 三引号

利用三引号（"""or'''），你可以指示一个多行的字符串。你可以在三引号中自由的使用单引号和双引号。例如：

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

5.7 转义序列

假如你有一个字符串包含单引号（'），如何表示这个字符串呢？例如，字符串是 What's your name? 。你不能用 'What's your name?' 来表示，因为 Python 不知道字符串的起始和结束位置。所以应该将字符串中间的这个单引号指定为不表示字符串的结束。这可在称之为转义序列的协助下实现。可以讲单引号指定为 ´—— 注意反斜杠。现在，就能将字符串表示为 'What\'s your name?' 。

还有一种方式就是用双引号 "What's your name?" 。类似地，在用双引号的字符串中用双引号必须用转义符。还有，必须用转义符 \\ 来表示反斜杠。

如果你想指定两行字符串，该如何做呢？一种方式就是用前面提到的用三引号的字符串，或者可以用转义符 \n 表示新的一行的开始。例如 This is the first line\nThis is the second line 。另外一个有用的转义字符是 Tab 键 ——\ t。有许多转义序列，这儿仅仅提到了最有用的几个。

需要说明的是，在一个字符串中，在一行末尾的反斜杠仅仅表示下一行的字符串是上一行的继续。，但并不增加新的行。例如：

```
"This is the first sentence.\
This is the second sentence."
```

与 "This is the first sentence. This is the second sentence." 等价。

5.8 自然字符串

如果，你想指定一些不被特殊处理，例如像转义序列，那么，就需要通过在字符串前面附加 r 或 R 来指定自然字符串。例如，"Newlines are indicated by \n" 。

5.9 字符串是不可变的

这意味着一个字符串一旦创建，就不能在改变它。虽然这似乎是坏事，其实不是。在下面我们会看到在各种各样的程序中这其实并不是什么限制。

5.10 字符串按字面意义连接

如果你把两个字符串按字面意义相邻放着，他们会被 Python 自动级连。例如，'What\'s"your name?' 会被自动转为 "What's your name?" 。

给 C/C++ 程序员的注释：

在 Python 中没有单独的 char 数据类型。其实也没有必要，我确定你不会再考虑它。

给 Perl/PHP 程序员的注释：

记住单引号和双引号是一样的 —— 没有丝毫差异。

给正则表达式使用者的注释：

用正则表达式的时候请使用自然字符串。否则，可能会用到许多反斜杠。例如,，后向引用符可以写成 '\\1' 或 r'\1'。

5.11 format 方法

有时我们并不想用其他信息来构造字符串。这儿 format() 方法就很有用。

```
#!/usr/bin/python
# Filename: str_format.py
age = 25
name = 'Swaroop'
print('{0} is {1} years old'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

输出：

```
$ python str_format.py
Swaroop is 25 years old
Why is Swaroop playing with that python?
```

运行原理：

一个字符串能使用确定的格式，随后，可以调用 format 方法来代替这些格式，参数要与 format 方法的参数保持一致。

观察首次使用 0 的地方，这与 format 方法的第一个参变量 name 相一致。类似地，第二个格式 1 与 format 方法的第二个参变量 age 相一致。

注意，也可用字符串连接： name + ' is ' str(age) + ' years old' 来实现，但这似乎比较难看，容易出错。第二，转为字符串的操作由 format 自动完成，而不需要明确的转换。第三，用 format 方法的时候，不必处理用过的变量和 vice-versa 就能改变消息。

在 Python 中， format 方法就是用参变量的值代替格式符。更多细节如下：

```
>>> '{0:.3}'.format(1/3) # decimal (.) precision of 3 for float
'0.333'
>>> '{0:_^11}'.format('hello') # fill with underscores (_) with the
    text
centered (^) to 11 width
'___hello___'
>>> '{name} wrote {book}'.format(name='Swaroop', book='A Byte of
   Python')
# keyword-based
'Swaroop wrote A Byte of Python'
```

更多关于格式在 Python Enhancement Proposal No.3101 中有解释。

5.12 变量

仅仅使用字面意义上的常量很快就会引发问题 —— 我们需要一种既可以储存信息又可以对它们进行操作的方法。这是为什么要引入变量。变量就是我们想要的东西 —— 它们的值可以变化，即你可以使用变量存储任何东西。变量只是你的计算机中存储信息的一部分内存。与字面意义上的常量不同，你需要一些能够访问这些变量的方法，因此要给变量命名。

5.13 标识符的命名

变量是标识符的例子。标识符是用来标识某样东西的名字。在命名标识符的时候，你要遵循这些规则：

- 标识符的第一个字符必须是字母表中的字母（大写或小写）或者一个下划线（'_'）。
- 标识符名称的其他部分可以由字母（大写或小写）、下划线（'_'）或数字（0-9）组成。
- 标识符名称是对大小写敏感的。例如， myname 和 myName 不是一个标识符。注意前者中的小写 n 和后者中的大写 N 。
- 有效标识符名称的例子有 i 、 __my_name 、 name_23 和 a1b2_c3 。
- 无效标识符名称的例子有 2things 、 this is spaced out 和 my-name 。

5.14 数据类型

变量可以有不同类型的值，称之为**数据类型**。基本数据类型是数字和字符串，这两种已经讨论过了。在后面的章节中，会看到如何用类来创建我们自己的类型。

5.15 对象

记住， Python 将一切在程序中用到的东西都作为对象。这是从广义上来讲的。我们不会说“某某东西”，我们会说“某个对象”。

给面向对象程序使用者的注释：

Python 是完全面向对象的，在某种意义上，任何东西都被作为对象，包括数字、字符串和函数。

让我们来看看如何与文字意义上的常量一道使用变量。保存下面的例子，运行程序。

如何写 Python 程序

此后，保存、运行一个 Python 程序的标准过程如下：

1. 打开你喜欢的编辑器。
2. 键入例子中给出的程序代码。
3. 用注释中给的文件名来保存文件，我保持将所有 Python 程序保存成扩展名为.py 的习惯。
4. 用带命令行解释器运行.py 程序，或者用 IDLE 运行程序。也可以用前面提到的执行方式。

5.16 例子：使用变量和文字意义上的常量

```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)
s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

输出：

```
$ python var.py
5
6
This is a multi-line string.
This is the second line.
```

运行机理：

这里是这个程序是运行机理。首先，用 = 给变量 i 赋值 5 。这一行叫做一个语句，语句声明需要做某件事情，在这个地方我们把变量名 i 与值 5 连接在一起。接下来，我们用 print 语句打印 i 的值，就是把变量的值打印在屏幕上。 然后我们对 i 中存储的值加 1 ，再把它存回 i 。我们打印它时，得到期望的值 6 。

类似地，我们把一个字面意义上的字符串赋给变量 s 然后打印它。

给静态语言程序员的注释：

使用变量时只需要给它们赋一个值。不需要声明或定义数据类型。

5.17 逻辑行和物理行

物理行是你在编写程序时所看见的。逻辑行是 Python 看见的单个语句。 Python 假定每个物理行对应一个逻辑行。

逻辑行的例子如 print 'Hello World' 这样的语句 —— 如果它本身就是一行（就像你在编辑器中看到的那样），那么它也是一个物理行。

默认地， Python 希望每行都只使用一个语句，这样使得代码更加易读。

如果你想要在一个物理行中使用多于一个逻辑行，那么你需要使用分号（;）来特别地标明这种用法。分号表示一个逻辑行/语句的结束。例如：

```
1 i = 5
2 print i
```

与下面这个相同：

```
1 i = 5;
2 print i;
```

同样也可以写成：

```
1 i = 5; print i;
```

甚至可以写成：

```
1 i = 5; print i
```

然而，我强烈建议你坚持在每个物理行只写一句逻辑行。仅仅当逻辑行太长的时候，在多于一个物理行写一个逻辑行。这些都是为了尽可能避免使用分号，从而让代码更加易读。事实上，我从来没有在 Python 程序中使用过或看到过分号。下面是一个在多个物理行中写一个逻辑行的例子。它被称为明确的行连接。

```
1 s = 'This is a string. \
2 This continues the string.'
3 print s
```

它的输出：

```
1 This is a string. This continues the string.
```

类似地，

```
print \
i
```

与如下写法效果相同：

```
print i
```

有时候，有一种暗示的假设，可以使你不需要使用反斜杠。这种情况出现在逻辑行中使用了圆括号、方括号或波形括号的时候。这被称为暗示的行连接。你会在后面介绍如何使用列表的章节中看到这种用法。

5.18 缩进

在 Python 中空白非常重要。实际上，在每行开头的空白很重要。称之为缩进。在行首的主要的空白（空格键和制表符）用来决定逻辑行缩进的层次，从而来决定语句分组。

这意味着同一层次的语句必须有相同的缩进。每一组这样的语句称为一个块。我们将在后面的章节中看到有关块的用处的例子。

你需要记住的一样东西是错误的缩进会引发错误。例如：

```
i = 5
 print('Value is ', i) # Error! Notice a single space at the start
    of the line
print('I repeat, the value is ', i)
```

当你运行的时候，会得到下面的出错信息：

```
File "whitespace.py", line 4
 print('Value is ', i) # Error! Notice a single space at the start
    of the line
 ^
IndentationError: unexpected indent
```

注意在第二行的开头有一个空格。Python 给出的错误信息告诉我们程序的语法不正确，例如，程序的写法不正确。它告诉你，你不能随意地开始新的语句块（当然除了你一直在使用的主块）。何时你能够使用新块，将会在后面的章节，如控制流中详细介绍。

如何缩进

不要混合使用制表符和空格来缩进，因为这在跨越不同的平台的时候，无法正常工作。我强烈建议你在每个缩进层次使用单个制表符或两个或四个空格。选择这三种缩进风格之一。更加重要的是，选择一种风格，然后一贯地使用它，即只使用这一种风格。

给静态语言程序员的注释：

Python 总是使用缩进来代表代码块，不再使用括号。运行 from __future__ import braces 来获取更多信息。

5.19 概况

现在我们已经学习了很多详细的内容，我们可以开始学习更加令你感兴趣的东西，比如控制流语句。在继续学习之前，请确信你对本章的内容清楚明了。

第 6 章 操作符和表达式

6.1 简介

你写的许多语句（逻辑行）会包含表达式。表达式的最简单的例子是 2 + 3 。表达式可以被分解成操作符和操作数。

运算符的功能是完成某件事，它们由如 + 这样的符号或者其他特定的关键字表示。运算符需要数据来进行运算，这样的数据被称为操作数。在这个例子中，2 和 3 是操作数。

6.2 操作符

简要地看一下操作符及用法：

你可以交互地使用解释器来计算例子中给出的表达式。例如，为了测试表达式 2+3，使用交互式的带提示符的 Python 解释器：

```
>>>2 + 3
5
>>>3 * 5
15
>>>
```

Operator	Name	Explanation	Examples
+	Plus	Adds the two objects	3 + 5 gives 8. 'a' + 'b' gives 'ab'.
-	Minus	Either gives a negative number or gives the subtraction of one number from the other	-5.2 gives a negative number. 50 - 24 gives 26.
*	Multiply	Gives the multiplication of the two numbers or returns the string repeated that many times.	2 * 3 gives 6. 'la' * 3 gives 'lalala'.
**	Power	Returns x to the power of y	3 ** 4 gives 81 (i.e. 3 * 3 * 3 * 3)
/	Divide	Divide x by y	4 / 3 gives 1.333333333333.
//	Floor	Division Returns the floor of the quotient	4 // 3 gives 1.
%	Modulo	Returns the remainder of the division	8 % 3 gives 2. -25.5 % 2.25 gives 1.5.

< <	Left Shift	Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1)	2 < < 2 gives 8. 2 is represented by 10 in bits. left shifting by 2 bits gives 1000 which represents the decimal 8.
> >	Right Shift	Shifts the bits of the number to the right by the number of bits specified.	11 > > 1 gives 5. 11 is represented in bits by 1011 which when right shifted by 1 bit gives 101 which is the decimal 5.
&	Bitwise AND	Bitwise AND of the numbers	5 & 3 gives 1.
|	Bit-wise OR	Bitwise OR of the numbers	5 | 3 gives 7
^	Bit-wise XOR	Bitwise XOR of the numbers	5 ^ 3 gives 6
~	Bit-wise invert	The bit-wise inversion of x is -(x+1)	~5 gives -6
<	Less Than	Returns whether x is less than y. All comparison operators return True or False. Note the capitalization of these names.	5 < 3 gives False and 3 < 5 gives True. Comparisons can be chained arbitrarily: 3 < 5 < gives True.
>	Greater Than	Returns whether x is greater than y	5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.
<=	Less Than or Equal To	Returns whether x is less than or equal to y	x = 3; y = 6; x <= y returns True.
>=	Greater Than or Equal To	Returns whether x is greater than or equal to y	x = 4; y = 3; x >= 3 returns True.
==	Equal To	Compares if the objects are equal	x = 2; y = 2; x == y returns True. x = 'str'; y = 'stR'; x == y returns False. x = 'str'; y = 'str'; x == y returns True.
!=	Not Equal To	Compares if the objects are not equal	x = 2; y = 3; x != y returns True.
not	Boolean NOT	If x is True, it returns False. If x is False, it returns True.	x = True; not x returns False.

and	Boolean AND	x and y returns False if x is False, else it returns evaluation of y	x = False; y = True; x and y returns False since x is False. In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is False which implies that the whole expression will be False irrespective of the other values. This is called short-circuit evaluation.
or	Boolean OR	If x is True, it returns True, else it returns evaluation of y	x = True; y = False; x or y returns True. Short-circuit evaluation applies here as well.

6.3 数学运算和赋值的简便方法

对变量进行数学运算然后再将值赋给原来的变量较为常见，因此对这种表达式有渐变的方法：
可以将：

```
a = 2; a = a * 3
```

写成：

```
a = 2; a *= 3
```

注意到，变量 = 变量 操作符 表达式 变成了 变量 操作符 = 表达式。

6.4 优先级

如果你有一个如 2 + 3 * 4 那样的表达式，是先做加法呢，还是先做乘法？我们的中学数学告诉我们应当先做乘法 —— 这意味着乘法运算符的优先级高于加法运算符。

下面这个表给出 Python 的运算符优先级，从最低的优先级（最松散地结合）到最高的优先级（最紧密地结合）。这意味着在一个表达式中，Python 会首先计算表中较下面的运算符，然后在计算列在表上部的运算符。

下面这张表（与 Python 参考手册中的那个表一模一样）已经顾及了完整的需要。事实上，我建议你使用圆括号来分组运算符和操作数，以便能够明确地指出运算的先后顺序，使程序尽可能地易读。例如，2 + (3 * 4) 显然比 2 + 3 * 4 清晰。与此同时，圆括号也应该正确使用，而不应该用得过滥（比如 2 + (3 + 4)）。

Operator	Description
lambda	Lambda Expression
or	Boolean OR

and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
< <, > >	Shifts
+, -	Addition and subtraction
*, /, //, %	Multiplication, Division, Floor Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[index1:index2]	Slicing
f(arguments ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
key:datum, ...	Dictionary display

其中我们还没有接触过的运算符将在后面的章节中介绍。在表中列在同一行的运算符具有相同优先级。例如，+ 和 - 有相同的优先级。

6.5 改变优先级

为了使表达式更易于阅读，可以使用圆括号。例如，2 + (3 * 4) 要比 2 + 3 * 4 更易于理解，因为第二个表达式需要了解操作符的优先级。与任何其他的事情一样，圆括号应该合理地使用，而不应使其冗余（例如 2 + (3 + 4)）

6.6 结合顺序

运算符通常是从左往右结合的，例如，有相同优先级的运算符按照从左向右的顺序计算。例如，2+3+4 的顺序是 (2+3)+4。一些运算符，如赋值运算符有从右向左的结合顺序，例如 a=b=c 被当成 a=(b=c) 。

6.7 表达式

例子：

```
1 #!/usr/bin/python
```

```
# Filename: expression.py

length = 5
breadth = 2

area = length * breadth
print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

输出：

```
$ python expression.py
Area is 10
Perimeter is 14
```

它如何工作

矩形的长度与宽度存储在以它们命名的变量中。我们借助表达式使用它们计算矩形的面积和边长。我们表达式 length * breadth 的结果存储在变量 area 中，然后用 print 语句打印。在另一个打印语句中，我们直接使用表达式 2 * (length + breadth) 的值。

另外，注意 Python 如何打印“漂亮的”输出。尽管我们没有在 'Area is' 和变量 area 之间指定空格，Python 自动在那里放了一个空格，这样我们就可以得到一个清晰漂亮的输出，而程序也变得更加易读（因为我们不需要担心输出之间的空格问题）。这是 Python 如何使程序员的生活变得更加轻松的一个例子。

6.8 概括

前面看到了如何使用运算符，操作数和表达式 —— 这是任何程序的最为基本的构成。接下来，将会看到如何通过语句在程序中使用这些内容。

第 7 章 控制流

7.1 简介

在到目前为止我们所见到的程序中，总是有一系列的语句，Python 忠实地按照它们的顺序执行它们。如果你想要改变语句流的执行顺序，该怎么办呢？例如，你想要让程序做一些决定，根据不同的情况做不同的事情，例如根据时间打印“早上好”或者“晚上好”。你可能已经猜到了，这是通过控制流语句实现的。在 Python 中有三种控制流语句 ——if、for 和 while 。

7.2 if 语句

if 语句用来检验一个条件，如果条件为真，我们运行一块语句（称为 if-块），否则我们处理另外一块语句（称为 else-块）。else 子句是可选的。

例子：

```
#!/usr/bin/python
# Filename: if.py

number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    print('Congratulations, you guessed it.') # New block starts
        here
    print('(but you do not win any prizes!)') # New block ends here
elif guess < number:
    print('No, it is a little higher than that') # Another block
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guess > number to reach here
print('Done')
# This last statement is always executed, after the if statement is
    executed
```

输出：

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done

$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done

$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
```

```
(but you do not win any prizes!)
Done
```

如何工作：

在这个程序中，我们从用户处得到猜测的数，然后检验这个数是否是我们手中的那个。我们把变量 number 设置为我们想要的任何整数，在这个例子中是 23 。然后，我们使用 input() 函数取得用户猜测的数字。函数只是重用的程序段。我们将在下一章学习更多关于函数的知识。

我们为内建的 input 函数提供一个字符串，这个字符串被打印在屏幕上，然后等待用户的输入。一旦我们输入一些东西，然后按回车键之后，函数返回输入。对于 input 函数来说是一个字符串。我们通过 int 把这个字符串转换为整数，并把它存储在变量 guess 中。事实上， int 是一个类，不过你想在对它所需了解的只是它把一个字符串转换为一个整数（假设这个字符串含有一个有效的整数文本信息）。

接下来，我们将用户的猜测与我们选择的数做比较。如果他们相等，我们打印一个成功的消息。注意我们使用了缩进层次来告诉 Python 每个语句分别属于哪一个块。这就是为什么缩进在 Python 如此重要的原因。我希望你能够坚持“每个缩进层一个制表符”的规则。你是这样的吗？

注意 if 语句在结尾处包含一个冒号 —— 我们通过它告诉 Python 下面跟着一个语句块。

然后，我们检验猜测是否小于我们的数，如果是这样的，我们告诉用户它的猜测大了一点。我们在这里使用的是 elif 从句，它事实上把两个相关联的 if else-if else 语句合并为一个 if-elif-else 语句。这使得程序更加简单，并且减少了所需的缩进数量。

elif 和 else 从句都必须在逻辑行结尾处有一个冒号，下面跟着一个相应的语句块（当然还包括正确的缩进）。

你也可以在一个 if 块中使用另外一个 if 语句，等等 —— 这被称为嵌套的 if 语句。

记住， elif 和 else 部分是可选的。一个最简单的有效 if 语句是：

```
if True:
    print('Yes, it is true')
```

在 Python 执行完一个完整的 if 语句以及与它相关联的 elif 和 else 从句之后，它移向 if 语句块的下一个语句。在这个例子中，这个语句块是主块。程序从主块开始执行，而下一个语句是 print 'Done' 语句。在这之后， Python 看到程序的结尾，简单的结束运行。

尽管这是一个非常简单的程序，但是我已经在这个简单的程序中指出了许多你应该注意的地方。所有这些都是十分直接了当的（对于那些拥有 C/C++ 背景的用户来说是尤为简单的）。它们在开始时会引起你的注意，但是以后你会对它们感到熟悉、“自然”。

给 C/C++ 程序员的注释：

在 Python 中没有 switch 语句。你可以使用 if..elif..else 语句来完成同样的工作（在某些场合，使用字典会更加快捷。）

7.3 while 语句

只要在一个条件为真的情况下， while 语句允许你重复执行一块语句。 while 语句是所谓循环语句的一个例子。 while 语句有一个可选的 else 从句。

例子：

```
# !/usr/bin/python
# Filename while.py

number = 23
running = True

while running:
  guess=int(input("Enter an integer:"))

  if guess == number:
        print("Congratulation, you guessed it.")
        running=False #this causes the while loop to stop
  elif guess < number:
        print("No, it is a little higher.")
  else:
        print("No, it is a little lower.")
else:
  print("the while loop is over.")
  # Do anything else you want to do here

print("done")
```

输出：

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

如何工作：

在这个程序中，我们仍然使用了猜数游戏作为例子，但是这个例子的优势在于用户可以不断的猜数，直到他猜对为止 —— 这样就不需要像前面那个例子那样为每次猜测重复执行一遍程序。这个例子恰当地说明了 while 语句的使用。

我们把 input 和 if 语句移到了 while 循环内，并且在 while 循环开始前把 running 变

量设置为 True 。首先，我们检验变量 running 是否为 True ，然后执行后面的 while-块。在执行了这块程序之后，再次检验条件，在这个例子中，条件是 running 变量。如果它是真的，我们再次执行 while-块，否则，我们继续执行可选的 else-块，并接着执行下一个语句。

当 while 循环的条件为假时 —— 这或许在第一次检查条件的时候。如果 while 循环有 else 从句，它将始终被执行，除非你的 while 循环将永远循环下去不会结束！

Ture 和 False 称为布尔类型，你可以将它们看做 1 和 0 。

给 C/C++ 程序员的注释：

记得 while 循环可以有 else 语句。

7.4 for 循环

for..in 是另外一个循环语句，它在一序列的对象上迭代，即逐一使用序列中的每个项目。我们会在后面的章节中更加详细地学习序列。你现在需要知道的就是序列仅仅是一些项目的有序聚集。

例子：

```
#!/usr/bin/python
#  Filename:  for.py

for  i  in  range ( 1 ,  5 ):
    print (i)
else :
    print ( 'The  for  loop  is  over' )
```

输出：

```
$ python for.py
1
2
3
4
The for loop is over
```

如何工作：

在这个程序中，我们打印了一个序列的数。我们使用内建的 range 函数生成这个数的序列。

我们所做的只是提供两个数， range 返回一个序列的数。这个序列从第一个数开始到第二个数为止。例如， range(1,5) 给出序列 [1, 2, 3, 4]。默认地， range 的步长为 1。如果我们为 range 提供第三个数，那么它将成为步长。例如，range(1,5,2) 给出 [1,3]。记住，range 向上延伸到第二个数，即它不包含第二个数。

for 循环在这个范围内递归 ——for i in range(1,5) 等价于 for i in [1, 2, 3, 4]，这就如同把序列中的每个数（或对象）赋值给 i ，一次一个，然后以每个 i 的值执行这个程序块。在这个例子中，我们只是打印 i 的值。

记住， else 部分是可选的。如果包含 else ，它总是在 for 循环结束后执行一次，除非遇到 break 语句。

记住， for..in 循环对于任何序列都适用。这里我们使用的是一个由内建 range 函数生成的数的列表，但是广义说来我们可以使用任何种类的由任何对象组成的序列！我们会在后面的章节中详细探索这个观点。

给 C/C++/Java/C# 注释：

Python 的 for 循环从根本上不同于 C/C++ 的 for 循环。 C# 程序员会注意到 Python 的 for 循环与 C# 中的 foreach 循环十分类似。 Java 程序员会注意到它与 Java 1.5 中的 for (int i : IntArray) 相似。

在 C/C++ 中，如果你想要写 for (int i = 0; i < 5; i++) ，那么用 Python ，你写成 for i in range(0,5) 。你会注意到， Python 的 for 循环更加简单、明白、不易出错。

7.5 break 语句

break 语句是用来终止循环语句的，即哪怕循环条件没有变为 False 或序列还没有被完全迭代结束，也停止执行循环语句。

一个重要的注释是，如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。

例子：

```
#!/usr/bin/python
# Filename: break.py

while True:
    s = (input('Enter something : '))
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

输出：

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something : use Python!
Length of the string is 12
Enter something : quit
Done
```

如何工作：

在这个程序中，我们反复地取得用户地输入，然后打印每次输入地长度。我们提

供了一个特别的条件来停止程序，即检验用户的输入是否是 'quit' 。通过终止循环到达程序结尾来停止程序。

输入的字符串的长度可以用内置函数 len 来计算。

记住，break 语句也可以用在 for 循环语句中。

7.6 Swaroop's 诗意般的 Python

我在这里输入的是我所写的一段小诗，称为 Swaroop's 诗意般的 Python：

```
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

7.7 continue 语句

continue 语句被用来告诉 Python 跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

例子：

```
#!/usr/bin/python
# Filename: continue.py

#!/usr/bin/python
# Filename: continue.py

while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
    continue
    print('Input is of sufficient length')
    # Do other kinds of processing here...
```

输出：

```
$ python test.py
Enter something : a
Too small
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

如何工作：

在这个程序中，我们从用户处取得输入，但是我们仅仅当它们有至少 3 个字符长的时候才处理它们。所以，我们使用内建的 len 函数来取得长度。如果长度小于 3，我们将使用 continue 语句忽略块中的剩余的语句。否则，这个循环中的剩余语句将被执行，我们可以在这里做我们希望的任何处理。

注意，continue 语句对于 for 循环也有效。

7.8 概括

我们已经学习了如何使用三种控制流语句 —— if 、 while 和 for 以及与它们相关的 break 和 continue 语句。它们是 Python 中最常用的部分，熟悉这些控制流是应当掌握的基本技能。

接下来，我们将学习如何创建和使用函数。

第 8 章 函数

8.1 简介

函数是重用的程序段。它们允许你给一个语句块一个名称，然后你用这个名字可以在你的程序的任何地方，任意多次地运行这个语句块。这被称为调用函数。我们已经使用了许多内建的函数，比如 len 和 range 。

函数用关键字 def 来定义。def 关键字后跟一个函数的标识符名称，然后跟一对圆括号。圆括号之中可以包括一些变量名，该行以冒号结尾。接下来是一块语句，它们是函数体。下面这个例子将说明这事实上是十分简单的：

例子：

```
#!/usr/bin/python
# Filename: function1.py

def sayHello():
    print('Hello World!') # block belonging to the function
# End of function
sayHello() # call the function
sayHello() # call the function again
```

如何工作：

我们使用上面解释的语法定义了一个称为 sayHello 的函数。这个函数不使用任何参数，因此在圆括号中没有声明任何变量。参数对于函数而言，只是给函数的输入，以便于我们可以传递不同的值给函数，然后得到相应的结果。

8.2 函数的参数

函数取得的参数是你提供给函数的值，这样函数就可以利用这些值做一些事情。这些参数就像变量一样，只不过它们的值是在我们调用函数的时候定义的，而非在函数本身内赋值。

参数在函数定义的圆括号对内指定，用逗号分割。当我们调用函数的时候，我们以同样的方式提供值。注意我们使用过的术语 —— 函数中的参数名称为形参而你提供给函数调用的值称为实参。

例子：

```
#!/usr/bin/python
# Filename: func_param.py

def printMax(a,b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to',b)
    else:
        print(b, 'is maximum')
```

```

printMax(3,4) # directly give literal valuse

x = 5
y = 7

printMax(x,y) # give variables as arguments
```

输出：

```
$ python func_param.py
4 is maximum
7 is maximum
```

如何工作：

这里，我们定义了一个称为 printMax 的函数，这个函数需要两个形参，叫做 a 和 b 。我们使用 if..else 语句找出两者之中较大的一个数，并且打印较大的那个数。在第一个 printMax 使用中，我们直接把数，即实参，提供给函数。在第二个使用中，我们使用变量调用函数。printMax(x,y) 使实参 x 的值赋给形参 a，实参 y 的值赋给形参 b 。在两次调用中， printMax 函数的工作完全相同。

8.3 局部变量

当你在函数定义内声明变量的时候，它们与函数外具有相同名称的其他变量没有任何关系，即变量名称对于函数来说是局部的。这称为变量的作用域。所有变量的作用域是它们被定义的块，从它们的名称被定义的那点开始。

例子：

```
#!/usr/bin/python
# Filename: func_local.py

x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

输出：

```
$ python func_local.py
x is 50
Changed local x to 2
x is still 50
```

如何工作：

在函数中，我们第一次使用 x 的值的时候， Python 使用函数声明的形参的值。

接下来，我们把值 2 赋给 x 。 x 是函数的局部变量。所以，当我们在函数内改变 x 的值的时候，在主块中定义的 x 不受影响。在最后一个 print 语句中，我们证明了主块中的 x 的值确实没有受到影响。

8.4 使用全局语句

如果你想要为一个定义在函数外的变量赋值，那么你就得告诉 Python 这个变量名不是局部的，而是全局的。我们使用 global 语句完成这一功能。没有 global 语句，是不可能为定义在函数外的变量赋值的。

你可以使用定义在函数外的变量的值（假设在函数内没有同名的变量）。然而，我并不鼓励你这样做，并且你应该尽量避免这样做，因为这使得程序的读者会不清楚这个变量是在哪里定义的。使用 global 语句可以清楚地表明变量是在外面的块定义的。

例子：

```
#!/usr/bin/python
#Filename: func_global.py

x = 50

def func():
    global x

    print('x is',x)
    x = 2
    print('Changed global x to',x)

func()
print('Value of x is',x)
```

输出：

```
$ python func_global.py
x is 50
Changed global x to 2
Value of x is 2
```

如何工作：

global 语句被用来声明 x 是全局的 —— 因此，当我们在函数内把值赋给 x 的时候，这个变化也反映在我们在主块中使用 x 的值的时候。

你可以使用同一个 global 语句指定多个全局变量。例如 global x, y, z 。

8.5 使用非局部语句

上面给出了如何在局部和全局作用域内使用变量。还有一种作用域叫做“非局部”域，处于这两种作用域之间。

非局部作用域在你定义函数内的函数时会看到。

由于在 Python 中，任何事物是可执行的代码，你可以在任何地方定义函数。

例子：

```
#!/usr/bin/python
# Filename: func_nonlocal.py

def func_outer():
    x = 2
    print('x is',x)

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('Changed local x to',x)

func_outer()
```

输出：

```
$ python func_nonlocal.py
x is 2
Changed local x to 5
```

如何工作：

当在函数 func_inner 的内部时，在函数 func_outer 的第一行定义的 'x' 相对来讲既不是在局部范围也不是在全局的作用域中，使用这样的 x 称之为非局部 x，因此可以使用这个变量。

将非局部变量 x 变成全局变量 x，可以通过将语句本身移除，在这两个例子中观察不同。

8.6 默认参数值

对于一些函数，你可能希望它的一些参数是可选的，如果用户不想要为这些参数提供值的话，这些参数就使用默认值。这个功能借助于默认参数值完成。你可以在函数定义的形参名后加上赋值运算符（=）和默认值，从而给形参指定默认参数值。

注意，默认参数值应该是一个参数。更加准确的说，默认参数值应该是不可变的——这会在后面的章节中做详细解释。从现在开始，请记住这一点。

例子：

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
    print(message * times)

say('Hello')
say('World', 5)
```

输出：

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

名为 say 的函数用来打印一个字符串任意所需的次数。如果我们不提供一个值，那么默认地，字符串将只被打印一遍。我们通过给形参 times 指定默认参数值 1 来实现这一功能。

在第一次使用 say 的时候，我们只提供一个字符串，函数只打印一次字符串。在第二次使用 say 的时候，我们提供了字符串和参数 5，表明我们想要说这个字符串消息 5 遍。

重要的：

只有在形参表末尾的那些参数可以有默认参数值，即你不能在声明函数形参的时候，先声明有默认值的形参而后声明没有默认值的形参。这是因为赋给形参的值是根据位置而赋值的。例如，def func(a, b=5) 是有效的，但是 def func (a=5, b) 是无效的。

8.7 关键参数

如果你的某个函数有许多参数，而你只想指定其中的一部分，那么你可以通过命名来为这些参数赋值 —— 这被称作关键参数 —— 我们使用名字（关键字）而不是位置（我们前面所一直使用的方法）来给函数指定实参。

这样做有两个优势 —— 一、由于我们不必担心参数的顺序，使用函数变得更加简单了。二、假设其他参数都有默认值，我们可以只给我们想要的那些参数赋值。

例子：

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

输出：

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

如何工作：

名为 func 的函数有一个没有默认值的参数，和两个有默认值的参数。

在第一次使用函数的时候， func(3, 7)，参数 a 得到值 3，参数 b 得到值 7，而参数 c 使用默认值 10。

在第二次使用函数 func(25, c=24) 的时候，根据实参的位置变量 a 得到值 25。根据命名，即关键参数，参数 c 得到值 24。变量 b 根据默认值，为 5。

在第三次使用 func(c=50, a=100) 的时候，我们使用关键参数来完全指定参数值。注意，尽管函数定义中， a 在 c 之前定义，我们仍然可以在 a 之前指定参数 c 的值。

8.8 VarArgs 参数

要做的：

在还未讨论列表和字典之前，我在后面的章节应该写这些吗？

有时，你或许想定义一个能获取任意个数参数的函数，这可通过使用 * 号来实现。

```
#!/usr/bin/python
# Filename: total.py

def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count

print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

输出：

```
$ python total.py
166
```

如何工作：

当我们定义一个带星的参数，像 *param 时，从那一点后所有的参数被收集为一个叫做 'param' 的列表。（译者注：如在该例中，首先会给 initial 的值由 5 变成 10，然后 numbers 将 1,2,3，收集作为一个列表 numbers=(1,2,3)）。

类似地，当我们定义一个带两个星的参数，像 **param 时，从那一点开始的所有的关键字参数会被收集为一个叫做 'param' 的字典。（译者注：在该例子中，从 vegetables=50 后的所有参数收集为一个字典 keywords='vegetables': 50, 'fruits': 100）。

在后面的章节中，我们会讨论列表和字典。

8.9 Keyword-only 参数

如果想指定特定的关键参数为 keywor-only 而不是位置参数，可以在带星的参数后申明：

```
#!/usr/bin/python
# Filename: keyword_only.py

def total(initial=5, *numbers, vegetables):
    count = initial
    for number in numbers:
        count += number
    count += vegetables
    return count

print(total(10, 1, 2, 3, vegetables=50))
print(total(10, 1, 2, 3,))
# Raises error because we have not supplied a default argument
   value for 'vegetables'
```

输出：

```
$ python keyword_only.py
66
Traceback (most recent call last):
File "test.py", line 12, in <module>
print(total(10, 1, 2, 3))
TypeError: total() needs keyword-only argument vegetables
```

如何工作：

在带星号的参数后面申明参数会导致 keyword-only 参数。如果这些参数没有默认值，且像上面那样不给关键参数赋值，调用函数的时候会引发错误。

如果你想使用 keyword-only 参数，但又不需要带星的参数，可以简单地使用不适用名字的空星，如 def total(initial=5, *, vegetables)。

8.10 return 语句

return 语句用来从一个函数返回即跳出函数。我们也可选是否从函数返回一个值。

```
#!/usr/bin/python
```

```
# Filename: func_return.py

def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print(maximum(2, 3))
```

输出：

```
$ python func_return.py
3
```

如何工作：

maximum 函数返回参数中的最大值，在这里是提供给函数的数。它使用简单的 if..else 语句来找出较大的值，然后返回那个值。注意，没有返回值的 return 语句等价于 return None 。 None 是 Python 中表示没有任何东西的特殊类型。例如，如果一个变量的值为 None ，可以表示它没有值。除非你提供你自己的 return 语句，每个函数都在结尾暗含有 return None 语句。通过运行 print(someFunction()) ，你可以明白这一点，函数 someFunction 没有使用 return 语句，如同：

```
def someFunction():
    pass
```

pass 语句在 Python 中表示一个空的语句块。

注释：

有一个内置函数 max，已经完成了'find maximum' 函数的功能，当可能的时候，使用内置函数。

8.11 DocStrings

Python 有一个很奇妙的特性，称为文档字符串，它通常被简称为 docstrings 。DocStrings 是一个重要的工具，由于它帮助你的程序文档更加简单易懂，你应该尽量使用它。你甚至可以在程序运行的时候，从函数恢复文档字符串！

例子：

```
#!/usr/bin/python
# Filename: func_doc.py
def printMax(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    x = int(x) # convert to integers, if possible
    y = int(y)

    if x > y:
```

```
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

printMax(3, 5)
print(printMax.__doc__)
```

输出：

```
$ python func_doc.py
5 is maximum
Prints the maximum of two numbers.

    The two values must be integers.
```

如何工作：

在函数的第一个逻辑行的字符串是这个函数的文档字符串。注意，DocStrings 也适用于模块和类，我们会在后面相应的章节学习它们。

文档字符串的惯例是一个多行字符串，它的首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述。强烈建议你在你的函数中使用文档字符串时遵循这个惯例。

你可以使用 __doc__（注意双下划线）调用 printMax 函数的文档字符串属性（属于函数的名称）。请记住 Python 把每一样东西都作为对象，包括这个函数。我们会在后面的类一章学习更多关于对象的知识。

如果你已经在 Python 中使用过 help()，那么你已经看到过 DocStings 的使用了！它所做的只是抓取函数的 __doc__ 属性，然后整洁地展示给你。你可以对上面这个函数尝试一下 —— 只是在你的程序中包括 help(printMax)。记住按 q 退出 help。

自动化工具也可以以同样的方式从你的程序中提取文档。因此，我强烈建议你对你所写的任何正式函数编写文档字符串。随你的 Python 发行版附带的 pydoc 命令，与 help() 类似地使用 DocStrings。

8.12 注解

函数有另外一个高级特性叫做注解，这是一个获取每个参数和返回值附加信息的漂亮的方式。由于 Python 语言本身不以任何方式解释这些注解（也即函数功能留给第三方库以他们想要的方式去解释），在我们的讨论中就跳过这一特性。如果你对阅读注解感兴趣，请看 Python Enhancement Proposal No. 3107。

8.13 概括

我们已经学习了函数的很多方面的知识，不过注意还有一些方面我们没有涉及。然而，我们已经覆盖了大多数在日常使用中，你可能用到的 Python 函数知识。

接下来，我们将学习如何创建和使用 Python 模块。

第 9 章 模块

9.1 简介

你已经学习了如何在你的程序中定义一次函数而重用代码。如果你想要在其他程序中重用很多函数，那么你该如何编写程序呢？你可能已经猜到了，答案是使用模块。

编写模块有各种各样的方法，但最简单的方法就是创建以 .py 为扩展名的文件，在文件中包含函数和变量。

另外一个编写模块的方法就是用自然语言，就是以 Python 编译器本身的方式写。例如，你可以用 C 语言写模块（http://docs.python.org/extending/），当编译时，使用标准 Python 编译器时，可以在 Python 代码使用。

模块可以从另外一个程序导入来使用其函数的功能。这也是我们使用 Python 标准库的方式。首先，看一看如何使用标准库模块。

例子：

```
#!/usr/bin/python
# Filename: using_sys.py

import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

输出：

```
$ python using_sys.py we are arguments
The command line arguments are:
using_sys.py
we
are
arguments

The PYTHONPATH is ['', 'C:\\Windows\\system32\\python30.zip',
'C:\\Python30\\DLLs', 'C:\\Python30\\lib',
'C:\\Python30\\lib\\plat-win', 'C:\\Python30',
'C:\\Python30\\lib\\site-packages']
```

如何工作：

首先，用导入了 sys 模块。基本上，这句语句告诉 Python，我们想要使用这个模块。sys 模块包含了与 Python 解释器和它的环境有关的函数。

当 Python 执行 import sys 语句的时候，它会寻找 sys 模块。在这个例子中，它是内置模块，因此 Python 知道在哪里找到它。

如果它是未编译的模块，如用 Python 写的模块， Python 解释器会查找列在 sys.path 变量中的路径。如果模块找到了，就会运行那个模块主体中的语句，模块就是可以利用的了。注意，初始化过程仅在我们第一次导入模块的时候进行。

在 sys 模块中，argv 变量可以用 sys.argv 来引用。它很清楚地表明这个名字是 sys 模块中的一部分。这种方法的一个优势是这个名称不会与任何在你的程序中使用的 argv 变量冲突。另

sys.argv 变量是一个字符串的列表（列表会在后面的章节详细解释）。特别地，sys.argv 包含了命令行参数的列表，即使用命令行传递给你的程序的参数。

如果你使用 IDE 编写运行这些程序，请在菜单里寻找一个指定程序的命令行参数的方法。

这里，当我们执行 python using_sys.py we are arguments 的时候，我们使用 python 命令运行 using_sys.py 模块，后面跟着的内容被作为参数传递给程序。 Python 为我们把它存储在 sys.argv 变量中。

记住，脚本的名称总是 sys.argv 列表的第一个参数。所以，在这里，'using_sys.py' 是 sys.argv[0]、'we' 是 sys.argv[1]、'are' 是 sys.argv[2] 以及'arguments' 是 sys.argv[3]。注意，Python 从 0 开始计数，而非从 1 开始。

sys.path 包含输入模块的目录名列表。我们可以观察到 sys.path 的第一个字符串是空的 —— 这个空的字符串表示当前目录也是 sys.path 的一部分，这与 PYTHONPATH 环境变量是相同的。这意味着你可以直接输入位于当前目录的模块。否则，你得把你的模块放在 sys.path 所列的目录之一。

注意，当前目录是程序启动的目录。运行 import os; print(os.getcwd()) 来查询你的程序的当前目录。

9.2 按字节编译的 .pyc 文件

导入一个模块相对来说是一个比较费时的事情，所以 Python 做了一些技巧，以便使输入模块更加快一些。一种方法是创建按字节编译的文件，这些文件以.pyc 作为扩展名。字节编译的文件与 Python 变换程序的中间状态有关（是否还记得 Python 如何工作的介绍？）。当你在下次从别的程序输入这个模块的时候，.pyc 文件是十分有用的 —— 它会快得多，因为一部分输入模块所需的处理已经完成了。另外，这些字节编译的文件也是与平台无关的。所以，现在你知道了那些.pyc 文件事实上是什么了。

注释：

这些.pyc 文件通常与.py 文件相同的方式在相同路径被创建。如果 Python 没有写入当前路径的权限，.pyc 文件就不会被创建。

9.3 from...import... 语句

如果你想要直接输入 argv 变量到你的程序中（避免在每次使用它时打 sys.），那么你可以使用 from sys import argv 语句。如果你想要输入所有 sys 模块使用的名字，那么你可以使用 from sys import * 语句。这对于所有模块都适用。

一般说来，应该避免使用 from..import 而使用 import 语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

9.4 模块的 __name__

每个模块都有一个名称，在模块中可以通过语句来找出模块的名称。这在一个场合特别有用 —— 就如前面所提到的，当一个模块被第一次输入的时候，这个模块的主块将被运行。假如我们只想在程序本身被使用的时候运行主块，而在它被别的模块输入的时候不运行主块，我们该怎么做呢？这可以通过模块的 __name__ 属性完成。

例子：

```
#!/usr/bin/python
# Filename: using_name.py

if __name__ == '__main__':
    print('This program is being run by itself')
else:
    print('I am being imported from another module')
```

输出：

```
$ python using_name.py
This program is being run by itself
$ python
>>> import using_name
I am being imported from another module
>>>
```

如何工作：

每个 Python 模块都有它的 __name__，如果它是'__main__'，这说明这个模块被用户单独运行，我们可以进行相应的恰当操作。

9.5 创建自己的模块

创建你自己的模块是十分简单的，你一直在这样做！每个 Python 程序也是一个模块。你已经确保它具有.py 扩展名了。下面这个例子将会使它更加清晰。

例子：

```
#!/usr/bin/python
# Filename: mymodule.py

def sayhi():
    print('Hi, this is mymodule speaking.')

__version__ = '0.1'
# End of mymodule.py
```

上面是一个模块的例子。你已经看到，它与我们普通的 Python 程序相比并没有什么特别之处。我们接下来将看看如何在我们别的 Python 程序中使用这个模块。

记住这个模块应该被放置在我们输入它的程序的同一个目录中，或者在 sys.path 所列目录之一。

```
#!/usr/bin/python
# Filename: mymodule_demo.py

import mymodule

mymodule.sayhi()
print ('Version', mymodule.__version__)
```

输出：

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

如何工作：

注意我们使用了相同的点号来使用模块的成员。Python 很好地重用了相同的记号来，使我们这些 Python 程序员不需要不断地学习新的方法。

下面是一个使用 from..import 语法的版本。

```
#!/usr/bin/python
# Filename: mymodule_demo2.py

from mymodule import sayhi, __version__

sayhi()
print('Version', __version__)
```

mymodule_demo2.py 的输出与 mymodule_demo.py 完全相同。

注意如果已经在导入 mymodule 的模块中申明了一个 __version__ 的名字，这就会有冲突。这也是有可能的，因为从实际情况来看，每个模块会用这个名字来申明它的版本。因此，推荐选择使用 import 语句，虽然会导致程序稍微有点冗长。

你也可以这样使用：

```
from mymodule import *
```

这回导入像 sayhi 这样公用的名字，但不会导入 __version__，因为它是以双下划线开始的。

Python 之禅

Python 的一个指导原则是“直白优于含蓄”。运行 import this 来学习更多，看关于这个的讨论 (http://stackoverflow.com/questions/228181/zen-of-python)，，这儿列举了每个原则的例子。

9.6 dir 函数

你可以使用内建的 dir 函数来列出模块定义的标识符。标识符有函数、类和变量。

当你为 dir() 提供一个模块名的时候，它返回模块定义的名称列表。如果不提供参数，它返回当前模块中定义的名称列表。

例子：

```
$ python
>>> import sys # get list of attributes, in this case, for the sys
    module
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__package__', '__s
tderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_compact_freelists',
'_current_frames', '_getframe', 'api_version', 'argv',
'builtin_module_names', '
byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook'
    ,
'dllhandle'
, 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
'executable',
'exit', 'flags', 'float_info', 'getcheckinterval',
'getdefaultencoding', 'getfil
esystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof',
'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize',
'maxunicode
', 'meta_path', 'modules', 'path', 'path_hooks', '
    path_importer_cache',
'platfor
m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile',
'setrecursionlimit
', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version'
    ,
'version_in
fo', 'warnoptions', 'winver']
>>> dir() # get list of attributes for current module
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
>>> a = 5 # create a new variable 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']
>>> del a # delete/remove a name
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
>>>
```

如何工作：

首先，我们来看一下在输入的 sys 模块上使用 dir。我们看到它包含一个庞大的属性列表。

接下来，我们不给 dir 函数传递参数而使用它，默认地，它返回当前模块的属性列表。注意，输入的模块同样是列表的一部分。

为了观察 dir 的作用，我们定义一个新的变量 a 并且给它赋一个值，然后检验 dir，我们观察到在列表中增加了以上相同的值。我们使用 del 语句删除当前模块中的变量/属性，这个变化再一次反映在 dir 的输出中。

关于 del 的注释 —— 该语句用于删除变量/名字，在运行语句后，。在这个例子中，del a，你将无法再使用变量 a—— 它就好像从来没有存在过一样。

注意 dir() 函数能在任何对象上起作用。例如，运行 dri(print) 来了解关于 print 函数的属性，或者 dir(str) 来查看 str 类的属性。

9.7 包

到目前为止，你一定已经开始看到了组织你的程序的层次。变量通常在函数内部运行。函数和全局变量通常在模块内部运行。如果你想自己组织模块呢？那" 包 " 就会进入到你的视野中。

包是模块的文件夹，有一个特殊的 __init__.py 文件，用来表明这个文件夹是特殊的因为其包含有 Python 模块。

加入你想创建一个叫做'world' 的包，有子包'asia','africa' 等等，并且，这些子包又包含模块，如'india','madagascar' 等等。

这就是你像构造的文件夹：

```
- <some folder present in the sys.path>/
    - world/
        - __init__.py
        - asia/
            - __init__.py
            - india/
                - __init__.py
                - foo.py
            - africa/
                - __init__.py
                - madagascar/
                    - __init__.py
                    - bar.py
```

包仅仅是为了方便层次化地组织模块。你会看到在标准库中看到许多这样的实例。

9.8 概括

就像函数是在程序块中能在利用一样，模块是可在利用的程序。包是另一个组织模块的层次。伴随 Python 的标准库是这样一个包集和模块的例子。

已经看到了如何使用模块并且创建自己的模块。

接下来，来了解一些叫做数据结构的这一有趣概念。

第 10 章 数据结构

10.1 简介

数据结构基本上就是 —— 它们是可以处理一些数据的结构。或者说，它们是用来存储一组相关数据的。

在 Python 中有四种内建的数据结构 —— 列表、元组、字典和集合。我们将会学习如何使用它们，以及它们如何使编程变得简单。

10.2 列表

list 是处理一组有序项目的数据结构，即你可以在一个列表中存储一个序列的项目。假想你有一个购物列表，上面记载着你要买的东西，你就容易理解列表了。只不过在你的购物表上，可能每样东西都独自占有一行，而在 Python 中，你在每个项目之间用逗号分割。

列表中的项目应该包括在方括号中，这样 Python 就知道你是在指明一个列表。一旦你创建了一个列表，你可以添加、删除或是搜索列表中的项目。由于你可以增加或删除项目，我们说列表是可变的数据类型，即这种类型是可以被改变的。

10.3 对象和类的简要介绍

虽然到现在一般延迟对对象和类得讨论，但现在需要一点解释，这样可以更好地理解列表。我们会在它所在的章节中详细研究这一主题。

列表就是使用了对象和类得例子。当我们使用变量 i 给其赋值，例如赋上整数 5，你就能将其想作创建了一个类 int 的对象（即，实例）i。实际上，可以阅读有关 int 的帮助（help(int)）来更好地理解这些内容。

一个类也可以有方法（即，定义的函数）来使用，当然，方法是就该类而言的。仅当你已经定义了一个该类的对象时，你才能使用这些功能。例如，Python 给列表提供了允许你在列表的末尾来增加项目的 append 方法。例如，mylist.append('an item') 会给列表 mylist 增加一个字符串。注意用带点的记号来访问对象的方法。

类也有字段，就是对类而言的定义了变量来使用。你可以使用这些变量/名字当有一个该类的对象的时候。字段也可用带点的方法来访问，例如，mylist.field。

例子：

```
1  #!/usr/bin/python
2  # Filename: using_list.py
3
4  # This is my shopping list
5  shoplist = ['apple', 'mango', 'carrot', 'banana']
6
7  print('I have', len(shoplist), 'items to purchase.')
8
9  print('These items are:', end=' ')
10
```

```
for item in shoplist:
    print(item, end=' ')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)
print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

输出：

```
$ python using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana','rice
    ']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango','rice
    ']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

如何工作：

变量 shoplist 是将要去市场的人的一个购物单。在 shoplist 中，进存储了各项要买的东西的名字，但你也可以增加任意类型对象，包括数字甚至列表。

使用了 for...in 循环通过列表中的各项来进行重复。到现在，你一定明白了列表也是一个序列。序列的特殊性会在后一节讨论。

注意 print 函数的 end 关键参数来表示以空格结束输出，而不是通常的换行。

接下来，用 append 方法来给列表对象增加一项，像前面讨论的那样。然后，通过简单地将列表传送给 print 语句漂亮地将列表的内容输出，来检查的确给类表增加了那一项。

然后，用 sort 方法将列表进行排序。理解这一方法影响了列表本身，没有返回修改后的列表是重要的 —— 这与字符串起作用的方式不同。这就是我们所说的列表是易变的，字符串是不可变的。

接下来，当我们在市场买完了一项时，我们想将其从列表中移去。用 del 语句来实现。这儿，我们提到了想要从列表中移去那一项，用 del 语句将其移除。指定想要移除列表中的第一项因此用 del shoplist[0]（记住，Python 中从 0 开始计算）。

如果想知道通过列表对象定义的所有方法，用 help(list) 获取更多细节。

10.4 元组

元组用来将多样的对象集合到一起。元组和列表十分类似，只不过元组和字符串一样是不可变的即你不能修改元组。元组通过圆括号中用逗号分割的项目定义。

元组通常用在使语句或用户定义的函数能够安全地采用一组值的时候，即被使用的元组的值不会改变。

例子：

```
#!/usr/bin/python
# Filename: using_tuple.py

zoo = ('python', 'elephant', 'penguin') # remember the parentheses
    are optional
print('Number of animals in the zoo is', len(zoo))

new_zoo = ('monkey', 'camel', zoo)
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is',len(new_zoo)-1+len(
    new_zoo[2]))
```

输出：

```
$ python using_tuple.py
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python','elephant'
    , 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

它如何工作:

变量 zoo 是一个元组，我们看到 len 函数可以用来获取元组的长度。这也表明元组也是一个序列。

由于老动物园关闭了，我们把动物转移到新动物园。因此，new_zoo 元组包含了一些已经在那里的动物和从老动物园带过来的动物。回到话题，注意元组之内的元组不会失去它的特性。

我们可以通过一对方括号来指明某个项目的位置从而来访问元组中的项目，就像我们对列表的用法一样。这被称作索引运算符。我们使用 new_zoo[2] 来访问 new_zoo 中的第三个项目。我们使用 new_zoo[2][2] 来访问 new_zoo 元组的第三个项目的第三个项目。如果理解习惯的这相当简单。

圆括号：

虽然圆括号是随意的，我更喜欢写出它使其明显地表明它是一个元组，尤其因为它会避免歧义。

例如，print(1,2,3) 和 print((1,2,3)) 意义不同 —— 前一个打印三个数字，而后一个打印出一个元组（包含三个数）。

含有 0 个或 1 个项目的元组：

一个空的元组由一对空的圆括号组成，如 myempty = ()。然而，含有单个元素的元组就不那么简单了。你必须在第一个（唯一一个）项目后跟一个逗号，这样 Python 才能区分元组和表达式中一个带圆括号的对象。即如果你想要的是一个包含项目 2 的元组的时候，你应该指明 singleton = (2 ,)。

给 perl 程序员的注释：

列表之中的列表不会失去它的身份，即列表不会像 Perl 中那样被打散。同样元组中的元组，或列表中的元组，或元组中的列表等等都是如此。只要是 Python，它们就只是使用另一个对象存储的对象。

10.5 字典

字典类似于你通过联系人名字查找地址和联系人详细情况的地址簿，即，我们把键（名字）和值（详细情况）联系在一起。注意，键必须是唯一的，就像如果有两个人恰巧同名的话，你无法找到正确的信息。

注意，你只能使用不可变的对象（比如字符串）来作为字典的键，但是你可以把不可变或可变的对象作为字典的值。基本说来就是，你应该只使用简单的对象作为键。

键值对在字典中以这样的方式标记：d = key1 : value1, key2 : value2 。注意它们的键/值对用冒号分割，而各个对用逗号分割，所有这些都包括在花括号中。

记住字典中的键/值对是没有顺序的。如果你想要一个特定的顺序，那么你应该在使用前自己对它们排序。

字典是 dict 类的实例/对象。

例子：

```
#!/usr/bin/python
# Filename: using_dict.py

# 'ab' is short for 'a'ddress'b'ook

ab = {  'Swaroop' : 'swaroop@swaroopch.com',
          'Larry' : 'larry@wall.org',
      'Matsumoto' : 'matz@ruby-lang.org',
        'Spammer' : 'spammer@hotmail.com'
     }

print("Swaroop's address is", ab['Swaroop'])

# Deleting a key-value pair
del ab['Spammer']

print('\nThere are {0} contacts in the address-book\n'.format(len(
    ab)))
```

```
18
19 for name, address in ab.items():
20     print('Contact {0} at {1}'.format(name, address))
21
22 # Adding a key-value pair
23 ab['Guido'] = 'guido@python.org'
24 if 'Guido' in ab: # OR ab.has_key('Guido')
25     print("\nGuido's address is", ab['Guido'])
```

输出：

```
1 $ python using_dict.py
2 Swaroop's address is swaroop@swaroopch.com
3
4 There are 3 contacts in the address-book
5
6 Contact Swaroop at swaroop@swaroopch.com
7 Contact Matsumoto at matz@ruby-lang.org
8 Contact Larry at larry@wall.org
9
10 Guido's address is guido@python.org
```

如何工作：

我们使用已经介绍过的标记创建了字典 ab。然后我们使用在列表和元组章节中已经讨论过的索引操作符来指定键，从而使用键/值对。我们可以看到字典的语法同样十分简单。

我们可以使用索引操作符来寻址一个键并为它赋值，这样就增加了一个新的键/值对，就像在上面的例子中我们对 Guido 所做的一样。

我们可以使用我们的老朋友 l——del 语句来删除键/值对。我们只需要指明字典和用索引操作符指明要删除的键，然后把它们传递给 del 语句就可以了。执行这个操作的时候，我们无需知道那个键所对应的值。

接下来，我们使用字典的 items 方法，来使用字典中的每个键/值对。这会返回一个元组的列表，其中每个元组都包含一对项目 —— 键与对应的值。我们抓取这个对，然后分别赋给 for..in 循环中的变量 name 和 address 然后在 for 一块中打印这些值。

我们可以使用 in 操作符来检验一个键/值对是否存在，或者使用 dict 类的 has_key 方法。你可以使用 help(dict) 来查看 dict 类的完整方法列表。

关键字参数与字典：

如果换一个角度看待你在函数中使用的关键字参数的话，你已经使用了字典了！只需想一下 —— 你在函数定义的参数列表中使用的键/值对。当你在函数中使用变量的时候，它只不过是使用一个字典的键（这在编译器设计的术语中被称作符号表）。

10.6 序列

列表、元组和字符串都是序列，但是序列是什么，它们为什么如此特别呢？

序列的主要特点是索成员检验（例如，在和不在表达式中）和索引操作符，索引操作符让我们可以直接从序列中抓取一个特定项目。

上面提到的三种类型的序列 —— 列表、元组和字符串，也有切片操作，切片操作让我们取出序列的薄片，例如序列的部分。

例子：

```
#!/usr/bin/python
# Filename: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Indexing or 'Subscription' operation
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])

# Slicing on a list
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing on a string
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

输出：

```
$ python seq.py
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

如何工作：

首先，我们来学习如何使用索引来取得序列中的单个项目。这也被称作是下标操作。每当你用方括号中的一个数来指定一个序列的时候，Python 会为你抓取序列中对应位置的项目。记住，Python 从 0 开始计数。因此，shoplist[0] 抓取第一个项目，shoplist[3] 抓取 shoplist 序列中的第四个元素。

索引同样可以是负数，在那样的情况下，位置是从序列尾开始计算的。因此，shoplist[-1] 表示序列的最后一个元素而 shoplist[-2] 抓取序列的倒数第二个项目。

切片操作符是序列名后跟一个方括号，方括号中有一对可选的数字，并用冒号分割。注意这与你使用的索引操作符十分相似。记住数是可选的，而冒号是必须的。

切片操作符中的第一个数（冒号之前）表示切片开始的位置，第二个数（冒号之后）表示切片到哪里结束。如果不指定第一个数，Python 就从序列首开始。如果没有指定第二个数，则 Python 会停止在序列尾。注意，返回的序列从开始位置开始，刚好在结束位置之前结束。即开始位置是包含在序列切片中的，而结束位置被排斥在切片外。

这样，shoplist[1:3] 返回从位置 1 开始，包括位置 2，但是停止在位置 3 的一个序列切片，因此返回一个含有两个项目的切片。类似地，shoplist[:] 返回整个序列的拷贝。

你可以用负数做切片。负数用在从序列尾开始计算的位置。例如，shoplist[:-1] 会返回除了最后一个项目外包含所有项目的序列切片。

你也可以给切片规定第三个参数，就是切片的步长（默认步长是 1）。

```
>>> shoplist = ['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::1]
['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::2]
['apple', 'carrot']
>>> shoplist[::3]
['apple', 'banana']
>>> shoplist[::-1]
['banana', 'carrot', 'mango', 'apple']
```

注意当步长是 2 时，我们得到在位置 0,2,... 的项，当步长是 3 时，得到位置 0,3, 等等的项。

使用 Python 解释器交互地尝试不同切片指定组合，即在提示符下你能够马上看到结果。序列的神奇之处在于你可以用相同的方法访问元组、列表和字符串！

10.7 集合

集合是没有顺序的简单对象的聚集。当在聚集中一个对象的存在比其顺序或者出现的次数重要时使用集合。

使用集合，可以检查是否是成员，是否是另一个集合的子集，得到两个集合的交集等等。

```
>>> bri = set(['brazil', 'russia', 'india'])
```

```
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

如何工作：

例子非常明显，它涉及到了基本的结合理论，这些都在高等数学中学过。

10.8 引用

当你创建一个对象并给它赋一个变量的时候，这个变量仅仅引用那个对象，而不是表示这个对象本身！也就是说，变量名指向你计算机中存储那个对象的内存。这被称作名称到对象的绑定。

一般说来，你不需要担心这个，只是在引用上有些细微的效果需要你注意。这会通过下面这个例子加以说明。

例子：

```
#!/usr/bin/python
# Filename: reference.py

print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist is just another name pointing to the
    same object!

del shoplist[0] # I purchased the first item, so I remove it from
    the list

print('shoplist is', shoplist)
print('mylist is', mylist)
# notice that both shoplist and mylist both print the same list
    without
# the 'apple' confirming that they point to the same object

print('Copy by making a full slice')
mylist = shoplist[:] # make a copy by doing a full slice
del mylist[0] # remove first item

print('shoplist is', shoplist)
print('mylist is', mylist)
```

输出：

```
$ python reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

如何工作：

大多数解释已经在程序的注释中了。你需要记住的只是如果你想要复制一个列表或者类似的序列或者其他复杂的对象（不是如整数那样的简单对象），那么你必须使用切片操作符来取得拷贝。如果你只是想要使用另一个变量名，两个名称都引用同一个对象，那么如果你不小心的话，可能会引来各种麻烦。

给 Perl 程序员的注释：

记住列表的赋值语句不创建拷贝。你得使用切片操作符来建立序列的拷贝。

10.9 更多关于字符串的内容

我们已经在前面详细讨论了字符串。我们还需要知道什么呢？那么，你是否知道字符串也是对象，同样具有方法。这些方法可以完成包括检验一部分字符串和去除空格在内的各种工作。

你在程序中使用的字符串都是 str 类的对象。这个类的一些有用的方法会在下面这个例子中说明。如果要了解这些方法的完整列表，请参见 help(str)。

例子：

```
#!/usr/bin/python
# Filename: str_methods.py

name = 'Swaroop' # This is a string object

if name.startswith('Swa'):
    print('Yes, the string starts with "Swa"')

if 'a' in name:
    print('Yes, it contains the string "a"')

if name.find('war') != -1:
    print('Yes, it contains the string "war"')

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

输出：

```
$ python str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
```

```
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

它如何工作

这里，我们看到使用了许多字符串方法。startwith 方法是用来测试字符串是否以给定字符串开始。in 操作符用来检验一个给定字符串是否为另一个字符串的一部分。

find 方法用来找出给定字符串在另一个字符串中的位置，或者返回 -1 以表示找不到子字符串。str 类也有以一个作为分隔符的字符串 join 序列的项目的整洁的方法，它返回一个生成的大字符串。

10.10 概括

我们已经详细探讨了多种 Python 内建的数据结构。这些数据结构将是编写程序时至关重要的部分。

现在我们已经掌握了很多 Python 的基本知识，我们接下来将学习如何设计和编写一个实用的 Python 程序。

第 11 章 解决问题

我们已经探讨了 Python 语言的各个部分，现在就来看看如何通过设计和编写一个能做有用事情的程序来将这些部分结合到一起。思路就是学习如何写你自己的一个 Python 脚本。

11.1 问题

问题是“我想写一个能给我所有重要文件建立备份的程序”。

尽管这是一个简单的问题，但是问题本身并没有给我们足够的信息来解决它。进一步的分析是必需的。例如，我们如何确定该备份哪些文件？备份保存在哪里？我们怎么样存储备份？

在恰当地分析了这个问题之后，我们开始设计我们的程序。我们列了一张表，表示我们的程序应该如何工作。对于这个问题，我已经创建了下面这个列表以说明我如何让它工作。如果是你设计的话，你可能不会这样来解决问题 —— 每个人都有其做事的方法，这很正常。

1. 需要备份的文件和目录由一个列表指定。
2. 备份应该保存在主备份目录中。
3. 文件备份成一个 zip 文件。
4. zip 存档的名称是当前的日期和时间。
5. 我们使用标准的 zip 命令，它通常默认地随 Linux/Unix 发行版提供。Windows 用户可以从GnuWin32项目页安装。注意你可以使用任何存档命令，只要它有命令行界面就可以了，那样的话我们可以从我们的脚本中传递参数给它。

11.2 解决方案

由于程序的设计现在来说是相对稳定的，我们可以写出这些代码，这就是解决方案的实现。（译者注：下面的程序并不一定能直接运行，还需手动修改一些相关参数）。

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1. The files and directories to be backed up are specified in a
   list.
source = ['"C:\\My Documents"', 'C:\\Code']
# Notice we had to use double quotes inside the string for names
   with
spaces in it.

# 2. The backup must be stored in a main backup directory
```

```
target_dir = 'E:\\Backup' # Remember to change this to what you
    will be
using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + os.sep + time.strftime('%Y%m%d%H%M%S') + '.
    zip'

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

输出：

```
$ python backup_ver1.py
Successful backup to E:\Backup\20080702185040.zip
```

现在，我们已经处于测试环节了，在这个环节，我们测试我们的程序是否正确工作。如果它与我们所期望的不一样，我们就得调试我们的程序，即消除程序中的 bugs（错误）。

如果上面的程序没有工作，在 os.system 命令之前设置一条语句 print(zip_command)，然后运行程序。现在复制/粘贴打印出来的 zip_command 到 shell 提示符，看一看能否正常运行。如果命令失败，检查 zip 命令手册到底是什么出错了。如果命令正常，检查 Python 程序，看它是不是与上面写出的程序一致。

如何工作：

接下来你将看到我们如何把设计一步一步地转换为代码。

我们使用了 os 和 time 模块，所以我们输入它们。然后，我们在 source 列表中指定需要备份的文件和目录。目标目录是我们想要存储备份文件的地方，它由 target_dir 变量指定。zip 归档的名称是目前的日期和时间，我们使用 time.strftime() 函数获得。它还包括.zip 扩展名，将被保存在 target_dir 目录中。

注意 os.sep 变量的使用 —— 这回根据你的操作系统给出路径分割符，例如在 Linux、Unix 中用'/'，在 windows 中用'\\'，在 Mac OS 中用':'。使用 os.sep 而不是直接使用这些符号会使你的程序更简洁，并且能在这些系统下正常工作。

time.strftime() 函数需要我们在上面的程序中使用的那种定制。%Y 会被无世纪的年份所替代。% m 会被 01 到 12 之间的一个十进制月份数替代，其他依次类推。这些定制的详细情况可以在《Python 参考手册》中获得。

我们用添加操作（该操作多个字符串连接到一起，例如，将两个字符串练到一起然后返回返回一个新的串）创建了名为 target.zip 的文件。然后，创建了一个

zip_command 字符串，这个字符串包含了将要执行的命令。你可以通过在 shell 中（Linux 终端或 DOS 提示符）运行来检查命令的正确性。

使用的 zip 命令有一些选项和传递的参数。-q 选项被用来表明 zip 命令应该以快速的方式进行。-r 选项指定 zip 命令应该对每个目录重复执行，例如应该包含所有子目录和文件。这两个选项可以用 -qr 结合到一起。选项后面跟的是要备份到 zip 文档的文件和目录列表。我们用 join 方法将资源列表转换成一个字符串，join 方法在上面已经讲过如何使用。

然后，我们最后用 os.system 函数运行命令，该函数就好像我们在系统中直接运行命令一样，例如在 shell 中，如果命令运行成功则返回 0，否则返回错误代码。

根据命令的结果，我们打印适当的消息来说明备份失败还是成功。

好了，我们已尽建立了一个脚本来备份我们的重要文件。

给 windows 用户的注释：

不用双反斜杠转义字符串，你可以用自然字符串。例如，用'C:\\Documents' 或'D:\Documents'。但不要用'D:\Documents'，因为你以未知转义字符 \D 结束。

现在我们已经有了一个可以工作的备份脚本，我们可以在任何我们想要建立文件备份的时候使用它。建议 Linux/Unix 用户使用前面介绍的可执行的方法，这样就可以在任何地方任何时候运行备份脚本了。这被称为软件的实施环节或开发环节。

上面的程序可以正确工作，但是（通常）第一个程序并不是与你所期望的完全一样。例如，可能有些问题你没有设计恰当，又或者你在输入代码的时候发生了一点错误，等等。正常情况下，你应该回到设计环节或者调试程序。

11.3 第二版

第一个版本的脚本可以工作。然而，我们可以对它做些优化以便让它在我们的日常工作中变得更好。这称为软件的维护环节。

我认为优化之一是采用更好的文件名机制 —— 使用时间作为文件名，而当前的日期作为目录名，存放在主备份目录中。这样做的一个优势是你的备份会以等级结构存储，因此它就更加容易管理了。另外一个优势是文件名的长度也可以变短。还有一个优势是采用各自独立的文件夹可以帮助你方便地检验你是否在每一天创建了备份，因为只有在你创建了备份，才会出现那天的目录。

```
1  #!/usr/bin/python
2  # Filename: backup_ver2.py
3
4  import os
5  import time
6
7  # 1. The files and directories to be backed up are specified in a
       list.
8  source = ['"C:\\My Documents"', 'C:\\Code']
9  # Notice we had to use double quotes inside the string for names
       with spaces in it.
10
```

```
# 2. The backup must be stored in a main backup directory
target_dir = 'E:\\Backup' # Remember to change this to what you
    will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main
    directory
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print('Successfully created directory', today)

# The name of the zip file
target = today + os.sep + now + '.zip'

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

输出：

```
$ python backup_ver2.py
Successfully created directory E:\Backup\20080702
Successful backup to E:\Backup\20080702\202311.zip

$ python backup_ver2.py
Successful backup to E:\Backup\20080702\202325.zip
```

如何工作：

两个程序的大部分是相同的。改变的部分主要是使用 os.exists 函数检验在主备份目录中是否有以当前日期作为名称的目录。如果没有，我们使用 os.mkdir 函数创建。（译者注：注意 os.sep 变量的用法 —— 这会根据你的操作系统给出目录分隔符，即在 Linux、Unix 下它是'/'，在 Windows 下它是'\\'，而在 Mac OS 下它是':'。使用 os.sep 而非直接使用字符，会使我们的程序具有移植性，可以在上述这些系统下工作。）

11.4 第三版

第二个版本在我做较多备份的时候还工作得不错，但是如果有极多备份的时候，我发现要区分每个备份是干什么的，会变得十分困难！例如，我可能对程序或者演讲稿做了一些重要的改变，于是我想要把这些改变与 zip 归档的名称联系起来。这可以通过在 zip 归档名上附带一个用户提供的注释来方便地实现。

注释：

下面的程序不能运行，所以不要慌，请跟着一起因为这儿还有一课。

```
#!/usr/bin/python
# Filename: backup_ver3.py

import os
import time

# 1. The files and directories to be backed up are specified in a
    list.
source = ['"C:\\My Documents"', 'C:\\Code']
# Notice we had to use double quotes inside the string for names
    with spaces in it.

# 2. The backup must be stored in a main backup directory
target_dir = 'E:\\Backup' # Remember to change this to what you
    will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main
    directory
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = input('Enter a comment --> ')
if len(comment) == 0: # check if a comment was entered
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' +
    comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print('Successfully created directory', today)

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

输出：

```
$ python backup_ver3.py
    File "backup_ver3.py", line 25
      target = today + os.sep + now + '_' +
                                            ^
SyntaxError: invalid syntax
```

如何（不）工作：

这个程序不工作！Python 说有一个语法错误，这意味着脚本不满足 Python 可以识别的结构。当我们观察 Python 给出的错误的时候，它也告诉了我们它检测出错误的位置。所以我们从那行开始调试我们的程序。

通过仔细的观察，我们发现一个逻辑行被分成了两个物理行，但是我们并没有指明这两个物理行属于同一逻辑行。基本上，Python 发现加法操作符（+）在那一逻辑行没有任何操作数，因此它不知道该如何继续。记住我们可以使用物理行尾的反斜杠来表示逻辑行在下一物理行继续。所以，我们修正了程序。这被称为 bug 修订。

11.5 第四版

```
#!/usr/bin/python
# Filename: backup_ver3.py

import os
import time

# 1. The files and directories to be backed up are specified in a
   list.
source = ['"C:\\My Documents"', 'C:\\Code']
# Notice we had to use double quotes inside the string for names
   with spaces in it.

# 2. The backup must be stored in a main backup directory
target_dir = 'E:\\Backup' # Remember to change this to what you
   will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main
   directory
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = input('Enter a comment --> ')
if len(comment) == 0: # check if a comment was entered
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' +\
    comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print('Successfully created directory', today)

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))

```

```
# Run the backup
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

输出：

```
$ python backup_ver4.py
Enter a comment --> added new examples
Successful backup to E:\Backup\20080702\202836_added_new_examples.
   zip

$ python backup_ver4.py
Enter a comment -->
Successful backup to E:\Backup\20080702\202839.zip
```

如何工作：

这个程序现在工作了！让我们看一下版本三中作出的实质性改进。我们使用 input 函数得到用户的注释，然后通过 len 函数找出输入的长度以检验用户是否确实输入了什么东西。如果用户只是按了回车（比如这只是一个惯例备份，没有做什么特别的修改），那么我们就如之前那样继续操作。

然而，如果提供了注释，那么它会被附加到 zip 归档名，就在.zip 扩展名之前。注意我们把注释中的空格替换成下划线 (_)—— 这是因为处理这样的文件名要容易得多。

11.6 更多的提炼

对于大多数用户来说，第四个版本是一个满意的工作脚本了，但是它仍然有进一步改进的空间。比如，你可以在程序中包含交互程度 —— 你可以用 -v 选项来使你的程序更具交互性。

另一个可能的改进是使文件和目录能够通过命令行直接传递给脚本。我们可以通过 sys.argv 列表来获取它们，然后我们可以使用 list 类提供的 extend 方法把它们加到 source 列表中去。

最重要的提炼就是不用 os.system 方式来创建压缩文档，而是用 zipfile 或 tarfile 内置模块来创建压缩文档。它们是标准库的一部分，没有与你计算机中外部 zip 程序的依赖性，并且已经能够使用。

但是，在上面的例子中，我已经使用了 os.system 的方式创建备份，这纯粹是为教学的方便，以使得例子足够简单，能被每个人理解，但这已经足够用了。

你能尝试用zipfile 模块来替代 os.system 调用写出第四版吗？

11.7 软件开发过程

现在，我们已经走过了编写一个软件的各个环节。这些环节可以概括如下：

1. 什么（分析）
2. 如何（设计）
3. 编写（实施）
4. 测试（测试与调试）
5. 使用（实施或开发）
6. 维护（优化）

我们创建这个备份脚本的过程是编写程序的推荐方法 —— 进行分析与设计。开始时实施一个简单的版本。对它进行测试与调试。使用它以确信它如预期那样地工作。再增加任何你想要的特性，根据需要一次次重复这个编写－测试－使用的周期。记住“软件是长出来的，而不是建造的”。

11.8 概括

我们已经学习如何创建我们自己的 Python 程序/脚本，以及在编写这个程序中所设计到的不同的状态。你可以发现它们在创建你自己的程序的时候会十分有用，让你对 Python 以及解决问题都变得更加得心应手。

接下来，我们将讨论面向对象的编程。

第 12 章 面向对象编程

12.1 简介

到目前为止，在我们的程序中，我们都是根据操作数据的函数或语句块来设计程序的。这被称为面向过程的编程。还有一种把数据和功能结合起来，用称为对象的东西包裹起来组织程序的方法。这种方法称为面向对象的编程理念。在大多数时候你可以使用过程性编程，但是有些时候当你想要编写大型程序或是寻求一个更加合适的解决方案的时候，你就得使用面向对象的编程技术。

类和对象是面向对象编程的两个主要方面。类创建一个新类型，而对象是这个类的实例。这类似于你有一个 int 类型的变量，这存储整数的变量是 int 类的实例（对象）。

给静态语言程序员的注释：

注意，即便是整数也被作为对象（属于 int 类）。这和 C++、Java（1.5 版之前）把整数纯粹作为类型是不同的。通过 help(int) 了解更多这个类的详情。

C# 和 Java 1.5 程序员会熟悉这个概念，因为它类似与封装与解封装的概念。

对象可以使用普通的属于对象的变量存储数据。属于一个对象或类的变量被称为域。对象也可以使用属于类的函数来具有功能。这样的函数被称为类的方法。这些术语帮助我们把它们与孤立的函数和变量区分开来。域和方法可以合称为类的属性。

域有两种类型 —— 属于每个实例/类的对象或属于类本身。它们分别被称为实例变量和类变量。

类使用 class 关键字创建。类的域和方法被列在一个缩进块中。

12.2 self

类的方法与普通的函数只有一个特别的区别 —— 它们必须有一个额外的第一个参数名称，但是在调用这个方法的时候你不为这个参数赋值，Python 会提供这个值。这个特别的变量指对象本身，按照惯例它的名称是 self。

虽然你可以给这个参数任何名称，但是强烈建议你使用 self 这个名称 —— 其他名称都是不赞成你使用的。使用一个标准的名称有很多优点 —— 你的程序读者可以迅速识别它，如果使用 self 的话，还有些 IDE（集成开发环境）也可以帮助你。

给 C++/Java/C# 程序员的注释：

Python 中的 self 等价于 C++ 中的 self 指针和 Java、C# 中的 this 参考。

你一定很奇怪 Python 如何给 self 赋值以及为何你不需要给它赋值。举一个例子会使此变得清晰。假如你有一个类称为 MyClass 和这个类的一个实例 MyObject。当你调用这个对象的方法 MyObject.method(arg1, arg2) 的时候，这会由 Python 自动转为 MyClass.method(MyObject, arg1, arg2)—— 这就是 self 的原理了。

这也意味着如果你有一个不需要参数的方法，你还是得给这个方法定义一个 self 参数。

12.3 类

一个尽可能简单的类如下面这个例子所示。

```
#!/usr/bin/python
# Filename: simplestclass.py

class Person:
    pass # An empty block

p = Person()
print(p)
```

输出：

```
$ python simplestclass.py
<__main__.Person object at 0x019F85F0>
```

它如何工作：

我们使用 class 语句后跟类名，创建了一个新的类。这后面跟着一个缩进的语句块形成类体。在这个例子中，我们使用了一个空白块，它由 pass 语句表示。

接下来，我们使用类名后跟一对圆括号来创建一个对象/实例。（我们将在下面的章节中学习更多的如何创建实例的方法）。为了验证，我们简单地打印了这个变量的类型。它告诉我们我们已经在 __main__ 模块中有了一个 Person 类的实例。

可以注意到存储对象的计算机内存地址也打印了出来。这个地址在你的计算机上会是另外一个值，因为 Python 可以在任何空位存储对象。

12.4 对象的方法

我们已经讨论了类/对象可以拥有像函数一样的方法，这些方法与函数的区别只是一个额外的 self 变量。现在我们来学习一个例子。

```
#!/usr/bin/python
# Filename: method.py

class Person:
    def sayHi(self):
        print('Hello, how are you?')

p = Person()
p.sayHi()
# This short example can also be written as Person().sayHi()
```

输出：

```
$ python method.py
Hello, how are you?
```

如何工作：

这里我们看到了 self 的用法。注意 sayHi 方法没有任何参数，但仍然在函数定义时有 self。

12.5 __init__ 方法

在 Python 的类中有很多方法的名字有特殊的重要意义。现在我们将学习 __init__ 方法的意义。

__init__ 方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的初始化。注意，这个名称的开始和结尾都是双下划线。

例子：

```
#!/usr/bin/python
# Filename: class_init.py

class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.sayHi()

# This short example can also be written as Person('Swaroop').sayHi
    ()
```

输出：

```
$ python class_init.py
Hello, my name is Swaroop
```

如何工作：

这里，我们把 __init__ 方法定义为取一个参数 name（以及普通的参数 self）。在这个 __init__ 里，我们只是创建一个新的域，也称为 name。注意它们是两个不同的变量，尽管它们有相同的名字。点号使我们能够区分它们。

最重要的是，我们没有专门调用 __init__ 方法，只是在创建一个类的新实例的时候，把参数包括在圆括号内跟在类名后面，从而传递给 __init__ 方法。这是这种方法的重要之处。

现在，我们能够在我们的方法中使用 self.name 域。这在 sayHi 方法中得到了验证。（译者注：__init__ 方法相当于 C++，Java，C# 中的构造函数）。

12.6 类和对象变量

我们已经讨论了类与对象的功能部分，现在我们来看一下它的数据部分。事实上，它们只是与类和对象的名称空间绑定的普通变量，即这些名称只在这些类与对象

的前提下有效。

有两种类型的域 —— 类的变量和对象的变量，它们根据是类还是对象拥有这个变量而区分。

类的变量由一个类的所有对象（实例）共享使用。只有一个类变量的拷贝，所以当某个对象对类的变量做了改动的时候，这个改动会反映到所有其他的实例上。

对象的变量由类的每个对象/实例拥有。因此每个对象有自己对这个域的一份拷贝，即它们不是共享的，在同一个类的不同实例中，虽然对象的变量有相同的名称，但是是互不相关的。通过一个例子会使这个易于理解。

```
#!/usr/bin/python
# Filename: objvar.py

class Robot:
    '''Represents a robot, with a name.'''

    #A class variable, counting the number of robots
    population = 0

    def __init__(self,name):
        '''Initializes the data.'''
        self.name = name
        print('(Initialize {0})'.format(self.name))

        #When this person is created, the robot
        # adds to the population
        Robot.population += 1

    def __del__(self):
        '''I am dying.'''
        print('{0} is being destroyed!'.format(self.name))

        Robot.population -= 1

        if Robot.population == 0:
            print('{0} was the last one.'.format(self.name))
        else:
            print('There are still {0:d} robots working.'.format(
                Robot.population))

    def sayHi(self):
        '''Greeting by the robot.

        Yeah, they can do that.'''
        print('Greetings, my master call me {0}.'.format(self.name)
            )

    def howMany():
        '''Prints the current population.'''
        print('We have {0:d} robots.'.format(Robot.population))
    howMany = staticmethod(howMany)

droid1 = Robot('R2-D2')
droid1.sayHi()
```

```
Robot.howMany()

droid2 = Robot('C-3P0')
droid2.sayHi()
Robot.howMany()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")

del droid1
del droid2

Robot.howMany()
```

输出：

```
(Initialize R2-D2)
Greetings, my master call me R2-D2.
We have 1 robots.
(Initialize C-3P0)
Greetings, my master call me C-3P0.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3P0 is being destroyed!
C-3P0 was the last one.
We have 0 robots.
```

如何工作：

这是一个很长的例子，但有助于说明类和对象变量的本质。这儿，population 属于 Robot 类，因此是一个类变量。name 变量属于对象（用 self 给其赋值），因此是一个对象变量。

因此，我们使用 Robot.population 来引用 population 类变量，而不是用 self.population 来引用。我们在该对象的方法中用 self.name 来引用对象变量 name。记住类和对象变量之间这个简单的差别。也要注意一个与类变量有相同名字的对象变量会隐藏类变量！

howMany 实际上是属于类而不是对象的方法。这意味着我们或者可以定义类方法或者可以定义静态方法，这取决于我们是否需要知道我们是那个类的部分。既然我们不需要这样的信息，就需要将其定义为静态方法。

我们也能用如下的方式来实现。(http://www.ibm.com/developerworks/linux/library/l-cpdecor.html)

```
@staticmethod
def howMany():
```

```
3       '''Prints the current population.'''
4       print('We have {0:d} robots.'.format(Robot.population))
```

这种方法可以想成是调用显式方法的简便方法，正像这个例子中看到的那样。

注意到 __init__ 方法用一个名字来初始化 Robot 实例。在该方法中，给 population 自增 1 来表明又添加了一个 robot。也主要到 self.name 的值对每个对象而言是不同的，这也说明了对象变量的本质。

记住，你必须仅用 self 来引用同一对象的变量和方法。这叫属性参考。

在这个程序中，也看到了和方法一样在类中使用 docstrings。我们可以在运行的时候使用 Robot.__doc__ 来获得类的 docstring，用 Robot.sayHi.__doc__ 来获得方法的 docstring。

就如同 __init__ 方法一样，还有一个特殊的方法 __del__，它在对象消逝的时候被调用。对象消逝即对象不再被使用，它所占用的内存将返回给系统作它用。在这个方法里面，我们只是简单地把 Person.population 减 1。

当对象不再被使用时，__del__ 方法运行，但是很难保证这个方法究竟在什么时候运行。如果你想要指明它的运行，你就得使用 del 语句，就如同我们在以前的例子中使用的那样。

给 C++/Java/C# 程序员的注释：

Python 中所有的类成员（包括数据成员）都是公共的，所有的方法都是有效的。

只有一个例外：如果你使用的数据成员名称以双下划线前缀比如 __privatevar，Python 的名称管理体系会有效地把它作为私有变量。

这样就有一个惯例，如果某个变量只想在类或对象中使用，就应该以单下划线前缀。而其他的名称都将作为公共的，可以被其他类/对象使用。记住这只是一个惯例，并不是 Python 所要求的（与双下划线前缀不同）。

12.7 继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

假设你想要写一个程序来记录学校之中的教师和学生情况。他们有一些共同属性，比如姓名、年龄和地址。他们也有专有的属性，比如教师的薪水、课程和假期，学生的成绩和学费。

你可以为教师和学生建立两个独立的类来处理它们，但是这样做的话，如果要增加一个新的共有属性，就意味着要在这两个独立的类中都增加这个属性。这很快就会显得不实用。

一个比较好的方法是创建一个共同的类称为 SchoolMember 然后让教师和学生的类继承这个共同的类。即它们都是这个类型（类）的子类型，然后我们再为这些子类型添加专有的属性。

使用这种方法有很多优点。如果我们增加/改变了 SchoolMember 中的任何功能，它会自动地反映到子类型之中。例如，你要为教师和学生都增加一个新的身份证

域，那么你只需简单地把它加到 SchoolMember 类中。然而，在一个子类型之中做的改动不会影响到别的子类型。另外一个优点是你可以把教师和学生对象都作为 SchoolMember 对象来使用，这在某些场合特别有用，比如统计学校成员的人数。一个子类型在任何需要父类型的场合可以被替换成父类型，即对象可以被视作是父类的实例，这种现象被称为**多态现象**。

另外，我们会发现在重用父类的代码的时候，我们无需在不同的类中重复它。而如果我们使用独立的类的话，我们就不得不这么做了。

在上述的场合中，SchoolMember 类被称为基本类或超类。而 Teacher 和 Student 类被称为导出类或子类。

现在，我们将学习一个例子程序。

```
#!/usr/bin/python
# Filename: inherit.py

class SchoolMember:
    '''Represent any school member.'''
    def __init__(self,name,age):
        self.name = name
        self.age = age
        print('(Initialize SchoolMember:{0})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{0}" Age:"{1}"'.format(self.name,self.age),end
            ='')

class Teacher(SchoolMember):
    '''Represent a teacher.'''
    def __init__(self,name,age,salary):
        SchoolMember.__init__(self,name,age)
        self.salary = salary
        print('(Initialized Teacher:{0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary:"{0:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Represents a student'''
    def __init__(self,name,age,marks):
        SchoolMember.__init__(self,name,age)
        self.marks = marks
        print('(Initialized Student:{0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks:"{0:d}"'.format(self.marks))

t = Teacher('Mrs.Shrividya',30,30000)
s = Student('Swaroop',25,75)

print() #pirnts a blank line
```

```
41
42 members = [t,s]
43 for member in members:
44     member.tell() # work for both Teacher and Students
```

输出：

```
1 $ python inherit.py
2 (Initialize SchoolMember:Mrs.Shrividya)
3 (Initialized Teacher:Mrs.Shrividya)
4 (Initialize SchoolMember:Swaroop)
5 (Initialized Student:Swaroop)
6
7 Name:"Mrs.Shrividya" Age:"30"Salary:"30000"
8 Name:"Swaroop" Age:"25"Marks:"75"
```

如何工作：

为了使用继承，我们把基本类的名称作为一个元组跟在定义类时的类名称之后。然后，我们注意到基本类的 __init__ 方法专门使用 self 变量调用，这样我们就可以初始化对象的基本类部分。这一点十分重要 ——Python 不会自动调用基本类的 constructor，你得亲自显式调用它。

我们还观察到我们在方法调用之前加上类名称前缀，然后把 self 变量及其他参数传递给它。

注意，在我们使用 SchoolMember 类的 tell 方法的时候，我们把 Teacher 和 Student 的实例仅仅作为 SchoolMember 的实例。

另外，在这个例子中，我们调用了子类型的 tell 方法，而不是 SchoolMember 类的 tell 方法。可以这样来理解，Python 总是首先查找对应类型的方法，在这个例子中就是如此。如果它不能在导出类中找到对应的方法，它才开始到基本类中逐个查找。基本类是在类定义的时候，在元组之中指明的。

一个术语的注释 —— 如果在继承元组中列了一个以上的类，那么它就被称作多重继承。

12.8 概括

我们已经研究了类和对象的多个内容以及与它们相关的多个术语。通过本章，你已经了解了面向对象的编程的优点和缺陷。Python 是一个高度面向对象的语言，理解这些概念会在将来有助于你进一步深入学习 Python。

接下来，我们将学习如何处理输入/输出以及如何用 Python 访问文件。

第 13 章 输入输出

13.1 简介

会有这种情形就是你的程序必须与用户进行交互。例如，你想要从用户那里获取输入然后将一些结果打印。我们可以分别使用 input() 和 print() 函数来实现。

对于输入，我们也可以使用 str(string) 类的各种方法。例如，你能够使用 rjust 方法来得到一个按一定宽度右对齐的字符串。利用 help(str) 获得更多详情。

另一个常用的输入/输出类型是处理文件。创建、读和写文件的能力是许多程序所必需的，我们将会在这章探索如何实现这些功能。

13.2 用户输入

```
#!/usr/bin/python
# user_input.py

def reverse(text):
    return text[::-1]

def is_palindrome(text):
    return text == reverse(text)

something = input('Enter text:')
if (is_palindrome(something)):
    print("Yes, it is a palindrome")
else:
    print("No,it is not a palindrome")
```

输出：

```
$ python user_input.py
Enter text: sir
No, it is not a palindrome

$ python user_input.py
Enter text: madam
Yes, it is a palindrome

$ python user_input.py
Enter text: racecar
Yes, it is a palindrome
```

如何工作：

我们使用了切片特性来将文本逆转。我们已经知道了如何使用 seq[a:b] 代码从位置 a 到位置 b 来形成切片。我们可以给其提供第三个参数，这个参数决定了切片的步长。默认步长是 1，因为要返回文本的连续部分。给一个负的步长，如 -1 会将文本逆转。

input() 函数用一个字符串作为其参数，然后显示给用户。然后等待用户键入一些东西，按返回键。一旦用户键入，input() 函数就返回该文本。

我们获得文本将其逆转。如果原始文本和逆转后的文本是一样的，文本是 回文。

课外练习：

检查文本是不是回文也应该忽略、空格和大小写。例如，"Rise to vote,sir." 也是回文，但当前的程序并不能指明。你能改进上面的程序来检查出这是回文吗？

译者注：课外练习答案

```
#!/usr/bin/python
# user_input.py
import string

def reverse(text):
    return text[::-1]

def is_palindrome(text):

    text = text.lower()
    text = text.replace(' ','')
    for char in string.punctuation:
        text = text.replace(char,'')
    return text == reverse(text)

def main():
    something = input('Enter text:')
    if (is_palindrome(something)):
        print("Yes, “{0}” is a palindrome".format(something))
    else:
        print("No “,{0}” is not a palindrome".format(something))

if __name__ == '__main__':
    main()
else:
    print("user_input.py was imported!")
```

13.3 文件

你可以通过创建一个 file 类的对象来打开一个文件，分别使用 file 类的 read、readline 或 write 方法来恰当地读写文件。对文件的读写能力依赖于你在打开文件时指定的模式。最后，当你完成对文件的操作的时候，你调用 close 方法来告诉 Python 我们完成了对文件的使用。

例子：

```
#!/usr/bin/python
# Filename: using_file.py

poem = '''\
Programming is fun
When the work is done
```

```
if you wanna make your work also fun:
    use Python!
'''

f = open('poem.txt', 'w') # open for 'w'riting
f.write(poem) # write text to file
f.close() # close the file

f = open('poem.txt') # if no mode is specified, 'r'ead mode is
   assumed by default
while True:
    line = f.readline()
    if len(line) == 0: # Zero length indicates EOF
        break
    print(line, end='')
f.close() # close the file
```

输出：

```
$ python using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

如何工作：

首先，我们通过指明我们希望打开的文件和模式来创建一个 file 类的实例。模式可以为读模式（'r'）、写模式（'w'）或追加模式（'a'）。事实上还有多得多的模式可以使用，你可以使用 help(file) 来了解它们的详情。默认情况下，open() 认为文件是't'ext 文件，且用'r'ead 模式打开。

在我们的例子中，我们首先用写模式打开文件，然后使用 file 类的 write 方法来写文件，最后我们用 close 关闭这个文件。

接下来，我们再一次打开同一个文件来读文件。如果我们没有指定模式，读模式会作为默认的模式。在一个循环中，我们使用 readline 方法读文件的每一行。这个方法返回包括行末换行符的一个完整行。所以，当一个空的字符串被返回的时候，即表示文件末已经到达了，于是我们停止循环。

默认情况下，print() 函数将文本和自动生成的新行打印到屏幕。我们指定 end=" 来制约新行的产生，因为从文件中读出的行已经用了换行符。最后，关闭文件。

现在，来看一下 poem.txt 文件的内容来验证程序读写的确都是正常的。

13.4 pickle 模块

Python 提供了一个叫做 pickle 的标准模块，使用该模块你可以将任意对象存储在文件中，之后你又可以将其完整地取出来。这被称为持久地存储对象。

例子：

```
#!/usr/bin/python
# Filename: pickling.py

import pickle

# the name of the file where we will store the object
shoplistfile = 'shoplist.data'
# the list of things to buy
shoplist = ['apple','mango','carrot']

# Write to the file
f = open(shoplistfile,'wb')
pickle.dump(shoplist, f) #dump the object to a file
f.close()

del shoplist # detroy the shoplist variable

# Read back from the storage
f = open(shoplistfile,'rb')
storedlist = pickle.load(f) # load the object from the file
print(storedlist)
```

输出：

```
$ python pickling.py
['apple', 'mango', 'carrot']
```

如何工作：

为了在文件里储存一个对象，首先以写，二进制模式打开一个 file 对象，然后调用 pickle 模块的 dump 函数，这个过程称为 pickling 。

接下来，我们用 pickle 模块的返回对象的 load 函数重新取回对象。这个过程称之为 unpickling 。

13.5 概括

我们讨论了输入/输出的各种类型，也讨论了文件处理和使用 pickle 模块。

接下来，我们会探讨异常的概念。

第 14 章 异常

14.1 简介

当你的程序中出现某些异常的状况的时候，异常就发生了。例如，当你想要读某个文件的时候，而那个文件不存在。或者在程序运行的时候，你不小心把它删除了。上述这些情况可以使用异常来处理。

假如你的程序中有一些无效的语句，会怎么样呢？Python 会引发并告诉你那里有一个错误，从而处理这样的情况。

14.2 错误

考虑一个简单的 print 语句。假如我们把 print 误拼为 Print，注意大写，这样 Python 会引发一个语法错误。

```
>>> Print('Hello World')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    Print('Hello World')
NameError: name 'Print' is not defined
>>> print('Hello World')
Hello World
```

我们可以观察到有一个 NameError 被引发，并且检测到的错误位置也被打印了出来。这是这个错误的错误处理器所做的工作。

14.3 异常

我们尝试读取用户的一段输入。按 Ctrl-d，看一下会发生什么。

```
>>> s = input('Enter something --> ')
Enter something -->
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s = input('Enter something --> ')
EOFError: EOF when reading a line
```

Python 引发了一个称为 EOFError 的错误，这个错误基本上意味着它发现一个不期望的文件尾（由 Ctrl-d 表示）（译者注：Python3.2 中没有此种错误发生）

14.4 处理异常

我们可以使用 try..except 语句来处理异常。我们把通常的语句放在 try-块中，而把我们的错误处理语句放在 except-块中。

```
#!/usr/bin/python
```

```
# Filename: try_except.py

try:
        text = input('Enter something --> ')
except EOFError:
        print('Why did you do an EOF on me?')
except KeyboardInterrupt:
        print('You cancelled the operation.')
else:
        print('You entered {0}'.format(text))
```

输出：

```
$ python try_except.py
Enter something --> # Press ctrl-d
Why did you do an EOF on me?

$ python try_except.py
Enter something --> # Press ctrl-c
You cancelled the operation.

$ python try_except.py
Enter something --> no exceptions
You entered no exceptions
```

如何工作：

我们把所有可能引发错误的语句放在 try 块中，然后在 except 从句/块中处理所有的错误和异常。except 从句可以专门处理单一的错误或异常，或者一组包括在圆括号内的错误/异常。如果没有给出错误或异常的名称，它会处理所有的错误和异常。

注意：对于每个 try 从句，至少都有一个相关联的 except 从句。

如果某个错误或异常没有被处理，默认的 Python 处理器就会被调用。它会终止程序的运行，并且打印一个消息，我们已经看到了这样的处理。

你还可以让 try..except 块关联上一个 else 从句。当没有异常发生的时候，else 从句将被执行。

我们还可以得到异常对象，从而获取更多有个这个异常的信息。这会在下一个例子中说明。

14.5 引发异常

你可以使用 raise 语句引发异常。你还得指明错误/异常的名称和伴随异常触发的异常对象。

你可以引发的错误或异常应该分别是一个 Error 或 Exception 类的直接或间接导出类。

```
class ShortInputException(Exception):
        '''A user-defined exception class'''
        def __init__(self, length,atleast):
```

```
                Exception.__init__(self)
                self.length = length
                self.atleast = atleast

try:
        text = input('Enter something-->')
        if len(text) < 3:
                raise ShortInputException(len(text),3)
        #other work can continue as usual here
except EOFError:
        print('Why did you do an EOF on me')
except ShortInputException as ex:
        print('ShortInputException The input was {0} long, excepted
            \
        atleast {1}'.format(ex.length, ex.atleast))
else:
        print('No exception was raised.')
```

输出：

```
$ python raising.py
Enter something --> a
ShortInputException: The input was 1 long, expected at least 3

$ python raising.py
Enter something --> abc
No exception was raised.
```

如何工作：

这里，我们创建了我们自己的异常类型，其实我们可以使用任何预定义的异常/错误。这个新的异常类型是 ShortInputException 类。它有两个域——length 是给定输入的长度，atleast 则是程序期望的最小长度。

在 except 从句中，我们提供了错误类和用来表示错误/异常对象的变量。这与函数调用中的形参和实参概念类似。在这个特别的 except 从句中，我们使用异常对象的 length 和 atleast 域来为用户打印一个恰当的消息。

14.6 Try..Finally

假如你在读一个文件的时候，希望在无论异常发生与否的情况下都关闭文件，该怎么做呢？这可以使用 finally 块来完成。注意，在一个 try 块下，你可以同时使用 except 从句和 finally 块。如果你要同时使用它们的话，需要把一个嵌入另外一个。

```
#!/usr/bin/python
# Filename: finally.py

import time

try:
        f = open('poem.txt')
        while True:     # our usual file-reading idiom
```

```
                line = f.readline()
                if len(line) == 0:
                        break
                print(line, end = '')
                time.sleep(2) # To make sure it runs for a while
except KeyboardInterrupt:
        print('!! You cancelled the reading from the file.')
finally:
        f.close()
        print('(Cleanig up: closed the file)')
```

输出：

```
$ python finally.py
Programming is fun
When the work is done
if you wanna make your work also fun:
!! You cancelled the reading from the file.
(Cleaning up: Closed the file)
```

如何工作：

我们进行通常的读文件工作，但是我有意在每打印一行之前用 time.sleep 方法暂停 2 秒钟。这样做的原因是让程序运行得慢一些（Python 由于其本质通常运行得很快）。在程序运行的时候，按 Ctrl-c 中断/取消程序。

我们可以观察到 KeyboardInterrupt 异常被触发，程序退出。但是在程序退出之前，finally 从句仍然被执行，把文件关闭。

14.7 with 语句

在 try 块中获得资源，随后又在 finally 块中释放资源，这是一种常见的模式。今后，也有一种 with 语句能以清晰的方式完成这样的功能。

```
#!/usr/bin/python
# Filename: using_with.py

with open("poem.txt") as f:
        for line in f:
                print(line,end='')
```

如何工作：

输出与前面例子的输出应该一样。这里的区别就是用 with 语句使用 open 函数 —— 用 with open 就能使得在结束的时候自动关闭文件。

在屏幕后面发生的事情就是 with 语句使用了一种协议。获得了 open 语句返回的对象，就叫做'thefile' 好了。

在启动代码块之前，在后台总会调用 thefile.__enter__ 函数，在代码块结束后又会调用 thefile.__exit__ 函数。

所以我们用 finally 块写的代码应该自行注意 __exit__ 方法。这就能帮助我们避免反复使用显式的 try..finally 语句。

更多关于这一主题的讨论已经超出了本书的范围，所以请参考 PEP 343获取综合性的说明。

14.8 概括

我们已经讨论了 try..except 和 try..finally 语句的用法。我们还学习了如何创建我们自己的异常类型和如何引发异常。

接下来，我们将探索 Python 标准库。

第 15 章 标准库

15.1 简介

Python 标准库是随 Python 附带安装的，它包含大量极其有用的模块。熟悉 Python 标准库是十分重要的，因为如果你熟悉这些库中的模块，那么你的大多数问题都可以简单快捷地使用它们来解决。

我们将会研究一些这个库中的常用模块。你可以在 Python 附带安装的文档的“库参考”一节中了解 Python 标准库中所有模块的完整内容。

让我们来研究一些有用的模块。

注释：

如果你发现这一章太高级了，你可以跳过这一章。但是，我强烈建议你，当你感觉用 Python 写程序更舒适时，你应回到这一章。

15.2 sys 模块

sys 模块包含了系统指定的函数功能。我们已经看到 sys.argv 列表包含命令行参数。

设想我们想要查看我们所使用 Python 命令行的版本，也就是说，我们想要保证我们正在使用的 Python 版本至少为 3。sys 模块给我们这样的功能。

```
>>> import sys
>>> sys.version_info
(3, 0, 0, 'beta', 2)
>>> sys.version_info[0] >= 3
True
```

如何工作：

sys 模块有一个 version_info 元组，它能给出版本的信息。第一个是主要的版本号。我们可以来检查这一条来保证程序运行在 Python3.0 或以上。

```
#!/usr/bin/python
# Filename: versioncheck.py

import sys, warnings

if sys.version_info[0] < 3:
        warnings.warn("Need Python 3.0 for this program to run",
            RuntimeWarning)
else:
        print('Proceed as normal')
```

输出：

```
$ python2.5 versioncheck.py
```

```
versioncheck.py:6: RuntimeWarning: Need Python 3.0 for this program
    to run
RuntimeWarning)

$ python3 versioncheck.py
Proceed as normal
```

如何工作：

我们使用了标准库中的另一个叫做 warning 的模块，来给最终用户显示警告信息。如果 Python 版本号少于 3，我们我们显示相应的警告。

15.3 logging 模块

如果你想得到一些调试信息或重要信息并将其存储在某个地方，用来检查你的程序程序运行是否是你期望的那样，该怎么做呢？如何处理存储在某处的这些信息呢？可以用 logging 模块来实现。

```
#!/usr/bin/python
# Filename: use_logging.py

import os, platform, logging

if platform.platform().startswith('Windows'):
        logging_file = os.path.join(os.getenv('HOMEDRIVE'),
                                                    'test.log')
else:
        logging_file = os.path.join(os.getenv('HOME'),'test.log')

logging.basicConfig(
                                level = logging.DEBUG,
                                format = '%(asctime)s : %(levelname
                                    )s : %(message)s',
                                filename = logging_file,
                                filemode = 'w',
                                )

logging.debug("Start of the program")
logging.info("Doing something")
logging.warning("Dying now")
```

输出：

```
$python use_logging.py
Logging to C:\Users\swaroop\test.log
```

如果检查 test.log 的内容，会有这样一些内容：

```
2008-09-03 13:18:16,233 : DEBUG : Start of the program
2008-09-03 13:18:16,233 : INFO : Doing something
2008-09-03 13:18:16,233 : WARNING : Dying now
```

如何工作：

我们使用了标准库中的三个模块 ——os 模块用来和操作系统交互，platform 模块用来得到平台的信息，例如像操作系统平台，logging 模块用来记录信息。

首先，我们用 platform.platform()（更多信息，查看 import platform;help(platform)）返回的字符串来检查操作系统的类型。如果是 windows，我们取出主盘符，主文件夹和要存储信息的文件名。将这些信息放在一起，就得到了文件的完整地位置。对于其他平台，我们只需要知道用户的主文件夹，然后得到文件的完整位置。

我们使用 os.path.join() 函数将这三部分放在一起。我们使用专门的函数而不是仅仅将三个字符串相加是因为函数能保证得到操作系统对应的文件位置格式。我们配置 logging 模块来将所有的信息以特定的格式写入指定的文件中。

最后，可以放入信息，可以是关于调试，通知，警告或者是临界消息。一旦程序运行，我们可以检查这个文件，了解到程序中发生了什么情况，尽管运行的程序没有任何信息显示给用户。

15.4 urllib 和 json 模块

如果你写的程序能得到从网页中的搜索结果，那会多么令人兴奋。现在就让我们做一番探索。

这可以用一些模块来实现。首先就是 urllib 模块，他可以用来从 internet 上取来一些网页。我们将使用 Yahoo! 搜索引擎来得到搜索结果，它们能给出叫做 JSON 格式的结果，我们就能很容易用在标准库中内建的模块来解析。

要做的事：

这个程序还不能工作，这似乎是 Python 3.0 beta 2 的一个 bug(http://bugs.python.org/issue3763)。

```
#!/usr/bin/python
# Filename: yahoo_search.py

import sys

if sys.version_info[0] != 3:
        sys.exit('This program needs Python 3.0')

import json
import urllib, urllib.parse, urllib.request, urllib.response

#Get your own APP ID at http://developer.yahoo.com/wsregapp/
YAHOO_APP_ID = '
   jl22psvV34HELWhdfUJbfDQzlJ2B57KFS_qs4I8D0Wz5U5_yCI1Awv8.lBSfPhwr
   '
SEARCH_BASE = 'http://search.yahooapis.com/WebSearchService/V1/
   webSearch'

class YahooSearchError(Exception):
        pass

```

```
# Taken from http://developer.yahoo.com/python/python-json.html
def search(query, results=20, start=1,**kwargs):
        kwargs.update({
                                'appid':YAHOO_APP_ID,
                                'query':query,
                                'results':results,
                                'start':start,
                                'output':'json',
                                })
        url = SEARCH_BASE + '?' + urllib.parse.urlencode(kwargs)
        result = json.load(urllib.request.urlopen(url))
        if 'Error' in result:
                raise YahooSearchError(result['Error'])
        return result['ResultSet']

query = input('What do you want to search for?')
for result in search(query)['Result']:
        print("{0}:{1}".format(result['Title'], result['Url']))
```

输出：

要做的事：

如何工作：

我们可以通过给出我们要搜索的特殊格式的文本从特殊的网站得到搜索结果。我们必须指定许多选项，我们用 key1 = value1 & key2 = value2 格式表示，这就可以用 urllib.parse.urlencode() 函数处理。

例如，在你的浏览器中打开这个链接，你会得到 20 个结果，从第一个结果"byte of python"，我们要求的输出结果是 JSON 格式的。

我们用 urllib.request.urlopen() 函数连接到这个 URL，将文件句柄传递给 json.load()，它会读取其中的内容，同时转换为 Python 对象。然后在这些结果上一次次循环，将其展示给终端用户。

15.5 Week 系列模块

在标准库中还有很多值得探究，如调试，命令行选项处理，正则表达式得等。

进一步探究标准库最好的方式是阅读 Doug Hellmann's excellent Python Moduleof the Week series。

15.6 概括

我们已经探究了在 Python 标准库中的许多模块的一些功能。强烈建议你浏览 Python 标准库文档来得到所有模块的概念。

接下来，我们会覆盖 Python 的各个方面，让我们的 Python 之旅更加完整。

第 16 章 更多内容

16.1 简介

到目前为止，我们已经学习了绝大多数常用的 Python 知识。在这一章中，我们将要学习另外一些方面的 Python 知识，从而使我们对 Python 的了解更加完整。

16.2 传送元组

你曾想过在一个函数中返回两个不同的值吗？你可以。要做的就是使用元组。

```
>>> def get_error_details():
... return (2, 'second error details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'second error details'
```

注意 a, b = <some expression> 可以解释为有两个值的元组表达式的结果。

如果想将结果解释为 (a, <everything else>)，你需要用星来表示，就如在函数参数中的那样。

```
>>> a, *b = [1, 2, 3, 4]
>>> a
1
>>> b
[2, 3, 4]
```

在 Python 中这也意味着交换两个变量的最快的方式：

```
>>> a = 5; b = 8
>>> a, b = b, a
>>> a, b
(8, 5)
```

16.3 特殊方法

有一些如 __init__ 和 __del__ 这样的确定方法，对类来说有重要意义。

特殊的方法用来模仿内置类型的确定动作。例如，如果你想对类使用 x[key] 角标操作（就像你在列表和元组中那样），那么你只需要实现 __getitem__() 方法就可以了。想一下，Python 就是对 list 类这样做的！

下面这个表中列出了一些有用的特殊方法。如果你想要知道所有的特殊方法，你可以在 参考手册 中找到一个庞大的列表。

Name	Explanation
__init__(self, ...)	This method is called just before the newly created object is returned for usage.
__del__(self)	Called just before the object is destroyed
__str__(self)	Called when we use the print function or when str() is used.
__lt__(self, other)	Called when the less than operator (<) is used. Similarly, there are special methods for all the operators (+, >, etc.)
__getitem__(self, key)	Called when x[key] indexing operation is used.
__len__(self)	Called when the built-in len() function is used for the sequence object.

16.4 单语句块

现在，你已经很深刻地理解了每一个语句块是通过它的缩进层次与其它块区分开来的。然而这在大多数情况下是正确的，但是并非 100% 的准确。如果你的语句块只包含一句语句，那么你可以在条件语句或循环语句的同一行指明它。下面这个例子清晰地说明了这一点：

```
>>> flag = True
>>> if flag: print 'Yes'
...
Yes
```

就如你所看见的，单个语句被直接使用而不是作为一个独立的块使用。虽然这样做可以使你的程序变得小一些，但是除了检验错误之外我强烈建议你不要使用这种缩略方法。不使用它的一个主要的理由是一旦你使用了恰当的缩进，你就可以很方便地添加一个额外的语句。

16.5 Lambda 形式

lambda 语句被用来创建新的函数对象，并且在运行时返回它们。

```
#!/usr/bin/python
# Filename: lambda.py

def make_repeater(n):
        return lambda s: s*n

twice = make_repeater(2)

print(twice('word'))
print(twice(5))
```

输出：

```
$ python lambda.py
wordword
10
```

如何工作：

这里，我们使用了 make_repeater 函数在运行时创建新的函数对象，并且返回它。lambda 语句用来创建函数对象。本质上，lambda 需要一个参数，后面仅跟单个表达式作为函数体，而表达式的值被这个新建的函数返回。注意，即便是 print 语句也不能用在 lambda 形式中，只能使用表达式。

要做的事：

你能通过用 lambda 创建的比较函数来完成 list.sort() 吗？

```
points = [ { 'x' : 2, 'y' : 3 }, { 'x' : 4, 'y' : 1 } ]
# points.sort(lambda a, b : cmp(a['x'], b['x']))
```

16.6 列表综合

通过列表综合，可以从一个已有的列表导出一个新的列表。例如，你有一个数的列表，而你想要得到一个对应的列表，使其中所有大于 2 的数都是原来的 2 倍。对于这种应用，列表综合是最理想的方法。

```
#!/usr/bin/python
# Filename: list_comprehension.py

listone = [2,3,4]
listtwo = [2*i for i in listone if i > 2]
print(listtwo)
```

输出：

```
$ python list_comprehension.py
[6, 8]
```

如何工作：

这里我们为满足条件（if i > 2）的数指定了一个操作（2*i），从而导出一个新的列表。注意原来的列表并没有发生变化。

使用列表综合的好处就是他会生成与循环生成每一个元素然后在一个新的列表中存储它一样的工作。

16.7 在函数中接收元组和列表

当要使函数接收元组或字典形式的参数的时候，有一种特殊的方法，它分别使用 * 和 ** 前缀。这种方法在函数需要获取可变数量的参数的时候特别有用。

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to specified
   power.'''
...     total = 0
...     for i in args:
...         total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2,10)
100
```

因为在 args 变量前面用了 * 前缀，所有传递给函数的多余的参数都被存储在元组 args 中。如果用 ** 前缀，额外的参数都会被以键/值的方式存储在字典中。

16.8 exec 和 eval 语句

exec 语句用来执行储存在字符串或文件中的 Python 语句。例如，我们可以在运行时生成一个包含 Python 代码的字符串，然后使用 exec 语句执行这些语句：

```
>>> exec('print("Hello World")')
Hello World
```

类似地，eval 函数用来执行存储在字符串中的 Python 表达式。简单的例子如下：

```
>>> eval('2*3')
6
```

16.9 assert 语句

assert 语句用来声明某个条件是真的。例如，如果你非常确信某个你使用的列表中至少有一个元素，而你想要检验这一点，并且在它非真的时候引发一个错误，那么 assert 语句是应用在这种情形下的理想语句。当 assert 语句失败的时候，会引发一个 AssertionError。

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> mylist
[]
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AssertionError
>>>
```

当然应该明智地使用 assert 语句。多数情况下，最好用来捕获异常，而不是处理问题或者给用户显示出错的消息然后退出。

16.10 repr 函数

repr 函数用来取得对象的规范字符串表示。注意，在大多数时候有 eval(repr(object)) == object。

```
>>> i = []
>>> i.append('item')
>>> i
['item']
>>> repr(i)
"['item']"
>>> eval(repr(i))
['item']
>>> eval(repr(i)) == i
True
```

基本上，repr 函数用来获取对象的可打印的表示形式。你可以通过定义类的 __repr__ 方法来控制你的对象在被 repr 函数调用的时候返回的内容。

16.11 概括

在这一章中，我们又学习了一些 Python 的特色，然而你可以肯定我们并没有学习完 Python 的所有特色。不过，到目前为止，我们确实已经学习了绝大多数你在实际中会使用的内容。这些已经足以让你去创建任何程序了。

接下来，我们会讨论一下如何进一步深入探索 Python。

第 17 章 近一步

如果你已经完全读完了这本书并且也实践着编写了很多程序，那么你一定已经能够非常熟练自如地使用 Python 了。你可能也已经编写了一些 Python 程序来尝试练习各种 Python 技能和特性。如果你还没有那样做的话，那么你一定要快点去实践。现在的问题是“接下来学习什么？”。

我会建议你先解决这样一个问题：

创建你自己的命令行地址簿程序。在这个程序中，你可以添加、修改、删除和搜索你的联系人（朋友、家人和同事等等）以及它们的信息（诸如电子邮件地址和/或电话号码）。这些详细信息应该被保存下来以便以后提取。

思考一下我们到目前为止所学的各种东西的话，你会觉得这个问题其实相当简单。如果你仍然希望知道该从何处入手的话，那么这里也有一个提示。

提示（先不要读）：

建一个类来表示人的信息。用字典来存储人的对象，可以将名字作为其键值。用 pickle 模块将对象永久地存在磁盘上。用字典的内置方法实现增加，删除和修改联系人的信息。

一旦你完成了这个程序，你就可以说是一个 Python 程序员了。现在，请立即寄一封信给（http://www.swaroopch.com/contact/）我感谢我为你提供了这本优秀的教材吧。是否告知我，如你所愿，但是我确实希望你能够告诉我。也可以考虑考虑捐助，帮助改进或者志愿做翻译工作来支持本书的继续发展。

如果你认为上面的程序很容易，这还有一个：

完成替换 (replace) 命令。这个命令完成给出的一列文件中字符串的替换。

替换命令要多简单有多简单，要多复杂有多复杂，可以是一个简单的串替换，也可以是寻找模式（正则表达式）。

完成之后，这有一些来继续你的 Python 之旅：

17.1 将代码作为例子

学习程序语言的最好的方式是多些，多读。

- ✓ The PLEAC project
- ✓ Rosetta code repository
- ✓ Python examples at java2s
- ✓ Python Cookbook 关于用 Python 如何解决确定种类的问题，收集了很有价值的巧妙方法和建议，

17.2 问题和答案

- ✓ Official Python Dos and Don’ts
- ✓ Official Python FAQ
- ✓ Norvig’s list of Infrequently Asked Questions

- ✓ Python Interview Q & A
- ✓ StackOverflow questions tagged with python

17.3 提示和技巧

- ✓ Python Tips & Tricks
- ✓ Advanced Software Carpentry using Python
- ✓ Charming Python is an excellent series of Python-related articles by David Mertz.

17.4 书籍，论文，辅导，视频

在本书后合理的下一步学习就是阅读 Mark Pilgrim 的书《Dive Into Python》，本书也可完全在线阅读。《Dive Into Python》详细探讨了关于正则表达式，XML 处理，web services，单元测试等内容。

其他有用的资源：

- ✓ ShowMeDo videos for Python
- ✓ GoogleTechTalks videos on Python
- ✓ Awaretek's comprehensive list of Python tutorials
- ✓ The Effbot's Python Zone
- ✓ Links at the end of every Python-URL! email
- ✓ Python Papers

17.5 讨论

如果你被一个 Python 问题卡住了，但又不知道要问谁，那么 comp.lang.python 讨论组就是最好的提问的地方。

当然首先要确保你做了你的作业，并且你自己已经尝试了如何去解决。

17.6 新闻

如果想了解 Python 的最新消息，那就时刻关注Official PythonPlanet 和/或者 Unofficial Python Planet。

17.7 安装库

大量的开源库在 Python Package Index,，这些可以用在你的程序中。

要安装这些库，你可以使用 Philip J. Eby's 的EasyInstall 工具。

17.8 图形软件

如果你想用 Python 创建你自己的图形软件。可以用 GUI(Graphical User Interface) 库和它们的绑定。绑定是允许你用 Python 写程序，使用库，但它们本身使用 C 或 C++ 或其他语言写的。

使用 Python，这有许多可供选择的 GUI:

✓ **PyQt**

这是 Qt 工具包的 Python 绑定。Qt 工具包是构建 KDE 的基石。Qt，特别是配合 Qt Designer 和出色的 Qt 文档之后，它极其易用并且功能非常强大。你可以在 Linux 下免费使用它，但是如果你在 Windows 下使用它需要付费。使用 PyQt，你可以在 Linux/Unix 上开发免费的（GPL 约定的）软件，而开发具产权的软件则需要付费。一个很好的 PyQt 资源是《使用 Python 语言的 GUI 编程：Qt 版》请查阅官方主页以获取更多详情。

✓ **PyGTK**

这是 GTK+ 工具包的 Python 绑定。GTK+ 工具包是构建 GNOME 的基石。GTK+ 在使用上有很多怪癖的地方，不过一旦你习惯了，你可以非常快速地开发 GUI 应用程序。Glade 图形界面设计器是必不可少的，而文档还有待改善。GTK+ 在 Linux 上工作得很好，而它的 Windows 接口还不完整。你可以使用 GTK+ 开发免费和具有产权的软件。请查阅官方主页以获取更多详情。

✓ **wxPython**

这是 wxWidgets 工具包的 Python 绑定。wxPython 有与它相关的学习方法。它的可移植性极佳，可以在 Linux、Windows、Mac 甚至嵌入式平台上运行。有很多 wxPython 的 IDE，其中包括 GUI 设计器以及如 SPE（Santi's Python Editor）和 wxGlade 那样的 GUI 开发器。你可以使用 wxPython 开发免费和具有产权的软件。请查阅官方主页以获取更多详情。

✓ **TkInter**

这是现存最老的 GUI 工具包之一。如果你使用过 IDLE，它就是一个 TkInter 程序。在 PythonWare.org 上的 TkInter 文档是十分透彻的。TkInter 具备可移植性，可以在 Linux/ Unix 和 Windows 下工作。重要的是，TkInter 是标准 Python 发行版的一部分。

要获取更多选择，请参阅 Python.org 上的 GUI 编程 wiki 页。

GUI 工具概括

不幸的是，并没有单一的标准 Python GUI 工具。我建议你根据你的情况在上述工具中选择一个。首要考虑的因素是你是否愿意为 GUI 工具付费。其次考虑的是你是想让你的程序运行在 Linux 下、Windows 下还是两者都要。第三个考虑因素根据你是 Linux 下的 KDE 用户还是 GNOME 用户而定。

跟多细节和综合分析，请看 The Python Papers, Volume, Issue 1的第 26 页。

17.9 多方面的补充

一种程序设计语言通常还有两部分 —— 语言和软件。语言是如何写一些东西。软件是实际什么在运行你的程序。

我们已经使用 CPython 软件来运行我们的程序。被归为 CPython，因为是用 C 语言写成的，就是我们的经典 Python 解释器。

还有其他的软件可以运行你的 Python 程序：

- ✓ Jython

  用 Java 语言实现的 Python 解释器。这意味着你可以用 Python 语言编写程序而同时使用 Java 库！Jython 是一个稳定成熟的软件。如果你也是一个 Java 程序员，我强烈建议你尝试一下 Jython。

- ✓ IronPython

  是用 C# 语言实现的 Python 解释器！这意味着你可以在 Python 语言内使用.NET 库和类或在.NET 语言中使用 Python。

- ✓ PyPy

  用 Python 写的 Python 解释器的实现！这是一个研究项目为了使得更快、更容易地改善编译器，是因为编译器本身使用动态语言写成的。（与用上面给出的三种静态语言如 C、Java 或 C# 的解释器相反）

- ✓ Stackless Python

  指定为以线程为基础的 Python 解释器。

还有其它的如 CLPython—— 用 Lisp 语言写成的 Python 解释器，和 IronMonkey，是 IronPython 的接口运行在 JavaScript 解释器之上，意味着你可以使用 Python(代替 JavaScript) 来写你的网页 -浏览器（"Ajax"）程序。

每一种解释器都有其特殊的应用领域，在这些地方非常有用。

17.10 概括

现在，我们已经来到了本书的末尾，但是就如那句名言，这只是开始的结束！你现在是一个满怀渴望的 Python 用户，毫无疑问你准备用 Python 解决许多问题。你可以使你的计算机自动地完成许多先前无法想象的工作或者编写你自己的游戏，以及更多别的什么东西。所以，请出发吧！

附录 1：FLOSS

自由/开放源码软件 (FLOSS)

FLOSS 基于社区的概念，而它本身基于共享，特别是知识共享的概念。FLOSS 可以免费使用、修改和再发行。

如果你已经读了本书，那么你一定熟悉 FLOSS，因为你一直在使用 Python，Python 就是开源软件！

这有一些 FLOSS 的例子，给出了社区分享和增加可以创造的一种东西的概念。

Linux.	Ubuntu.	OpenOffice.org.
Mozilla Firefox.	Mono.	Apache web server.
MySQL.	VLC Player.	GeexBox.

上面这个列表只是希望给你一个大概的印象 —— 还有很多别的优秀 FLOSS，比如 Perl 语言、PHP 语言、Drupal 网站内容管理系统、PostgreSQL 数据库服务器、TORCS 赛车游戏、KDevelop IDE、Xine—— 电影播放器、VIM 编辑器、Quanta+ 编辑器、Banshee 视频播放器，GIMP 图像编辑程序... 这个列表可以一直继续下去。

要获知 FLOSS 世界的最新进展，请访问下述网站：

◇ linux.com ◇ LinuxToday

◇ NewsForge ◇ DistroWatch

访问下述网站以获取更多 FLOSS 信息：

◇ SourceForge ◇ FreshMeat

那么，现在就出发去探索广博、免费、开放的 FLOSS 世界了吧！

附录 2：关于本书

请参考英文原版教程《A Byte of Python》。

附录 3：版本历史

请参考英文原版教程《A Byte of Python》。

附录 4：Python 3000 的更新

请参考英文原版教程《A Byte of Python》。