译序

<<a byte of python>>原作者为 Swaroop C H
原作官网 http://www.swaroopch.com/notes/Python
此版中译为 I_NBFA, 仅供学习交流之用, 其它行为后果自负.
==========
不必多说，<<a byte of python>>，中文版<<python 简明教程>>，老牌经典 python 教程.
早升级到 py3k 了但中文版一直没动静，另外 py3k 的资料本来就少上眼的教程更少.
因此我决定自己翻译它. 如果有童鞋看了我的中译本果断"叛变"到 python 阵营，本人将不胜荣幸.

<<a byte of python>>有很多版本，我采用的是最新版支持 python3.0
这里有个历史原因，因为写书的时候 python 3.0 还没正式发布，作者采用的是 3.0b1
所以里面有些内容看起来象是逆向穿越，但是表担心它们同样适用于其后版本。
注意，这绝不是本人第一次翻译文章，第二次是也，
本人英文阅读能力比较强大，与之形成鲜明对比书写能力比较凄惨(但与翻译没多大关系).
欢迎各路神仙批评指正，鄙人谢过。

ps. 表误会, 翻译仍在进行中...

<<a byte of python>>for python 3
I_NBFA 中译版 目录

目录

python 或许是少数几种简单又强大的语言之一，使其既适合新手又能满足老鸟．更重要的是使用 python 编程会非常有趣．
本书旨在帮助大家学习这门奇妙的语言，并示范如何快捷方便的完成任务．
事实上它就象理想的抗蛇毒血清对抗编程中出现的各种问题(注: python 是条大蛇嘛)

本书的读者

本书作为学习指南或 python 程序设计教程主要针对初学者，但对有经验的程序员同样有用．
即使您只会使用电脑保存文件，也可以通过本书学习 python．何况有编程经验的人自不必说．
如果你是后者很可能对 python 与你所喜爱的程序设计语言之间的差异感兴趣，本书突出显示了很多这样的差异．
另外表怪我没提醒你哦，随着学习的深入 python 很快就会成你最爱不释手的语言拉，嘿嘿．

本书的历史

第一次使用 python 语言是因为我曾编写的一个被称为"钻石"的程序需要一个简单的安装程序．
我必须在 python 和 perl 中择其一调用 Qt 库编写界面(注:一个著名的跨平台图形开发框架)．
为此我在网上搜索相关资料的过程中找到一篇黑客大牛 Eric S. Raymond 的文章，其中谈到
他是如何爱上 python 的．并且我也发现 PyQt 比 Perl_Qt 更加成熟．于是我决定选择 python．
可搜索 python 好书的结果竟然是无功而返!
我确实找到一些 O'Reilly 出版的书，但不是价钱太贵就是更象参考书而不是我需要的入门指南．
最后没办法，我只能去学习 python 自带的官方文档．不过这些资料过于精简概括性强但不够详尽．
幸好我有一些程序设计经验好歹啃明白了，但显然它们非常不适合初学者．
在首次接触 python 的 6 个月后，我安装了当时最新版的 Red Hat 9.0 linux 并且兴奋的玩起了 KWord(注:一个字处理软件)
突然我有一个想法，何不用 KWord 记下一些 python 资料？开始我只写了 10 页但很快就超过了 30 页．
既如此我决定将其认真酝酿成一本书，经过大量重写这个目标最终实现了，一本有用的 python 学习指南．
并且我将它贡献给开源社区．
本书始于我的个人 python 笔记，现在我同样这样看待它．不过我已付出很多努力让其更符合他人的阅读口味:)

另外在开源的精神下，我收到大量建设性的意见，批评和热心读者的反馈，他们帮助我对本书进行了极大的改善.

本书近况

上一个主要修订在 2005 年 3 月，其后的内容变化主要针对 python3.0 的发布(预计 2008 年 7 或 8 月份).
因为 python3.0 自身仍在完善中，因此本书也将不断改变. 不过本着开源哲学"早发布，常发布"的精神.
更新的即是发布的，而发布的也将被不断更新.
本书欢迎读者(比如您)指出不足，难以理解或者是犯了简单错误的地方. 请将您的意见和建议直接发给本人
(http://www.swaroopch.com/contact/)或是联系各地的译者.
另外本书一直存在着平衡初学着需求与信息完整性的问题. 读者的反馈将有助于本书应该定位在何种深度.

官方网站

本书官网 http://www.swaroopch.com/notes/Python，在这里可以在线完整阅读本书，下载本书最新版本，
买到其印刷版(http://www.swaroopch.com/buybook)或是提供反馈意见.

本书许可证

(注：比较敏感的东西不翻译了，如果只作学习之用尽管放心阅读，否则还是把许可证看懂了吧)

2. Attribution must be shown by linking back to http:/ / www. swaroopch. com/ notes/
Python and clearly indicating that the original text can be fetched from this location.
3. All the code/scripts provided in this book is licensed under the 3-clause BSD License
(http:/ / www. opensource. org/ licenses/ bsd-license. php) unless otherwise noted.
4. Volunteer contributions to this original book must be under this same license and the
copyright must be assigned to the main author of this book.

反馈

我已经付出大量努力使得本书尽可能有趣并精确.
但是如果您发现了错误或者前后矛盾的地方, 又或只是希望改进本书请通知我, 我将做出合适的修改.(可以在我的用户页找到我.)

购买本书

如果您希望支持本书的后续编写, 可以考虑购买其印刷版
(http://www.swaroopch.com/buybook)或进行捐赠.

一些思考

有两种构建软件的方式:
一种是将其设计的很简单, 这样明显不会有缺陷.
另一种是将其设计的很复杂, 因此不会看出有明显的缺陷.
--C.A.R.Hoare
生活中的成功是专注和坚持不懈的问题, 而不是天才与机遇的问题.
--C.W.Wendte

python 简介

python 是少数几种有实力宣称自己是集简便与强大于一体的语言之一。
使用 python 你会惊奇的发现能够很轻松的将精力集中到解决的问题上而不是语言的语法与结构上。
python 语言的官方介绍如下:

python 是一门易于学习，功能强大的程序设计语言。它具有高效的高级数据结构与简单但有效的面向对象

编程机制。python 那优雅的语法，动态类型与解释特性使其成为大多数平台上理想的脚本和快速开发语言
在下一节我将详细介绍其中的大多数特性。

笔记

python 语言的创始人 Guido van Rossum 以 BBC 播出的"巨蟒飞行马戏团"(Monty Python's Flying Circus)命名 python.

而他本人并不喜欢那些为了食物卷起长长的身体并压碎猎物的蛇.

python 特色

简单

python 是一门极简主义语言。阅读 python 程序感觉就象是在阅读英语一样，虽然这门英语具有严格的语法规定!

python 这种类似伪代码的特性是它最棒的优点之一。使得你能够集中精力到所解决的问题上而不是语言本身.

易学

你即将看到，python 具有非常简单的语法，很容易上手.

免费并且开源

python 是 FLOSS(Free/Fibre and Open Source Software)的一个范例，简单的说你可以自由分发这个软件，

阅读代码，进行修改，或是将其部份的用到其他开源软件中。FLOSS 基于社区应该彼此分享知识的理念，

这也是 python 如此优秀的原因之一 -- 从它诞生以来就不断的被一个社区改进，这个社区希望它可以变的更好.

高级语言

当你使用 python 编程时无需顾虑底层细节，比如存储器管理.

可移植

由于 python 的开源特性，它已经被移植到很多平台。只要注意避免平台相关调用你的 python 程序就可以运行在所有这些平台之上。

其中包括 Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390,

z/OS, Palm OS, ONX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE

甚至 PocketPC!

解释性

这里需要说明一下，由编译型语言诸如 C/C++编写的源代码会被编译器转换为机器可以执行的语言(机器语言，0 和 1 的组合),

其中编译器拥有一些列的标志与选项控制它的编译方式．当你运行程序时链接/加载器会将程序从磁盘拷贝到内存然后执行之。

与之相反 python 程序不需要被编译成二进制码，程序直接在源代码级"运行".

但在内部，python 会将源码转换为被称为字节码的中间码形式之后再将其翻译为所在计算机的本地码最后运行之。

如此，使用 python 就非常简便了，因为省去了编译过程也用不着考虑如何连接/加载正确的程序库.

这也使得 python 程序拥有非常好的可移植性，只要将程序拷贝到其他的电脑然后运行就 OK 了!

面向对象

python 既支持面向过程也支持面向对象编程。在面向过程的语言中，程序围绕可复用的过程或者函数创建。

而在面向对象的语言中，程序围绕可以组合数据与相应操作的对象创建。

python 以一种非常强大但又简明的方式支持 OOP(注：面向对象编程)，与那些庞大的语言相比(诸如 c++/java)这点更为突出.

可扩展

如果你需要让关键代码运行的更快或是希望算法的某部分避免公开，可以使用 c 或 c++编写它们并在 python 中进行调用。

可嵌入

可以将 python 嵌入到你的 C/C++程序中给予用户使用脚本的能力.

丰富的程序库

python 标准库已经非常庞大，它可以帮助你实现各种任务，包括正则表达式，文档生成，单元测试，多线程，

数据库, web 浏览器, CGI, FTP，电子邮件, XML, XML-PRC, HTML, wav 文件，加密, GUI(图形用户界面), Tk

以及系统相关功能．而这一切只需安装 python 即可拥有，此被称作 python 的"Batteries Included"哲学.

除此之外, python 还有许多高质量的第三方库可供使用,

wxPython(http://www.wxpython.org),

Twisted(http://www.twistedmatrix.com/products/twisted),

Python Imaging Library(http://www.pythonware.com/products/pil/index.htm)

等等等等...

python 如此强大另人兴奋，它将性能与语言特性良好的组合到一起使得编程成为一件简单轻松的事情.

为什么不选择 Perl?

听说过 Perl 吗? 它是另一门非常流行的开源的解释型程序设计语言.

如果你曾试图用它编写大型程序, 我想你自己已经有了这个问题的答案.
换言之, perl 擅长小型程序, 这时用起来比较方便. 但是在编写稍大的程序时 perl 很快就会变的难以使用.
我可不是信口开河, 这是我在 Yahoo!编写大型 perl 程序的亲身体会.
而与之相比, python 更加简单, 清晰, 容易编写, 因此也更容易理解与维护.
我个人也是喜欢 perl 的并且用它解决各种日常问题, 但在编写前我总是优先考虑 python, 可见它已经成为我的条件反射
了. perl 经历了那么多的修改与变更, 感觉就象一个超大号补丁语言. 遗憾的是即将到来的 perl 6 似乎并没有对此作出任
何改善. 在我看来 perl 目前唯一的重大优势就是它的 CPAN 了(http://cpan.perl.org). perl 资料大全网(Comprehensive Perl Archive Network), 顾名思义一个丰富无比的 perl 模块网, 其深广度另人难以置信,

你几乎可以利用这些资源完成任何计算机任务. perl 比 python 拥有更多库的原因之一是它拥有更悠久的历史.
不过随着 python 包索引(Python Package Index)(http://pypi.python.org/pypi)的增长, 这一优势正在改变.

为什么不使用 Ruby?

也许你还不知道, ruby 同样是一个非常流行的开源的解释型程序设计语言.
如果你已经喜欢上 ruby, 我会明确建议你继续使用下去. 而对于那些没有用过 ruby 并且正在犹豫是学 python 还是 ruby 的同学.

纯粹以简单易学的角度我会推荐 python. 就个人来说我发现自己真的很难对 ruby 感冒, 但那些懂 ruby 的人全都赞叹这门语言的美丽,

很遗憾, 我没这么幸运.

看看程序员门怎么说

你可能会对象 ESR 这样伟大的黑客们关于 python 的评论感兴趣:
• Eric S. Raymond 是"大教堂与集市"(The Cathedral and the Bazaar)的作者也是"开源"一此的创造者.
　他说 python 已经成为他非常喜爱的程序设计语言
(http://www.linuxjournal.com/article.php?sid=3882)
　也正是这篇文章另我心动, 决定一试 python
• Bruce Eckel 是著作"java 编程思想"(Thinking in Java)和"C++编程思想"(Thinking in C++)的作者
　他说没有任何语言令他的工作效率如此高. 他还说也许 python 是唯一一个旨在让程序员更简便完成任务的语言.

更详尽的访问内容(http://www.artima.com/intv/aboutme.html)

• Peter Norvig 即著名的 Lisp 语言作者并担任 Google 的搜索质量主管(感谢 Guido van Rossum 指出这点).

他说 python 已经成为 Google 不可或缺的一部分.

事实上你可以通过 Google Jobs(http://www.google.com/jobs/index.html)证实这点.

里面指出具有 python 知识是成为 Google 软件工程师的必要条件.

关于 **Python 3.0**

python 3.0 是 python 语言的新版本, 有时它被称做 Python 3000 或 Py3K.

升级的主要原因在于移除一些小问题并将那些累积了数年的毛病解决掉, 使得语言更加整洁.

如果你有很多 python 2.x 代码, 这个工具有助于将其转到 3.x 代码 (http://docs.python.org/dev/3.0/library/2to3.html)

安装 Python

如果你已经安装 python 2.x，不必卸载，python 3.0 可以与之共存.

Linux 和 BSD 用户

如果你正在使用某个 Linux 发行版，诸如 Ubuntu, Fedora, OpenSUSE 或{请把你的系统写在这里}, BSD 系统如 FreeBSD.
那么系统很可能已经自带 python 了。
为了验证这点打开一个 shell(比如 konsole 或 gnome-terminal)输入命令 *python -V,*

$ python -V
Python 3.0b1

注意

*$*是 shell 提示符，根据你的系统设置它会发生变化．本书一律使用$

如果你看到类似上面的 python 版本信息则你的系统已经安装 Python3．反之如果你看到类似下面的信息:

$ python -V
bash: Python: command not found

这证明没有安装 Python．虽然这种可能性非常小.

注意

如果你已经安装 python 2.x，则输入 *python3 -V.*

这种情况下你有两种方式在你的系统上安装 python.

- 可以从这里下载 python 代码然后自己编译并安装
(http://www.python.org/download/releases/3.0/)
- [此方法只有等到 python 3.0 最终发布后才有效](注：早发布了...)
通过你的系统自带的包管理软件安装 python2 进制包(例如 Ubuntu/基于 Debian linux 的 apt-get, Fedora linux 的 yum,

FreeBSD 的 pkg_add 等等...)
注意这种安装方式需要上网．不过你也可以从别的地方下载并拷贝到你的机器上安装.(注：真聪明...)

Windows 用户

访问 http://www.python.org/download/releases/3.0/下载 python 最新版,
当我写作本文的时候最新版为 3.0 beta 1(http://www.python.org/ftp/python/3.0/python-3.0b1.msi)
其大小只有 12.8MB 相较其它语言或软件，压缩率非常高了．安装方式与其它 windows 软件无异.

警告

不要改变预设选项，很多"可选"项对你很有用，尤其对 IDLE 更有用.

一个有趣的现象是 python 下载者中大多数都是 Windows 用户，当然很多 linux 系统已经自己带 python 了.

DOS 提示符(注：指的是 windows 命令行)

如果你想在 Windows 命令行即 DOS 提示下运行 python，则需适当设置 PATH 环境变量.
Windows 2000, XP, 2003 用户，点击控制面板 -> 系统 -> 高级 -> 环境变量.
点中"系统变量"中的 PATH 节选择"编辑"后加入";你的 python 安装目录"(注，无引号有分号),
比如你的 python 安装目录为"*C:\Python30*"则加入"*;C:\Python30*".
对于较老的 windows 系统，在 *C:\AUTOEXEC.BAT* 中加入"*PATH=%PATH%;你的安装目录*"(注：无引号).
Windows NT 则使用 *AUTOEXEC.NT* 文件.

Mac OS X 用户

Mac OS X 用户会发现系统已经安装 Python 了，打开 Terminal.app 并运行 *python -V*，其它参考上面的 Linux 用户节.

小结

Linux 系统很可能已经预装 Python 了，否则可以利用发行版的包管理软件安装 python.
Windows 系统下安装 python 非常简单，下载并双击即可.
从现在起，我们假设你的系统已经安装好 python. 接下来，即将编写我们的第一个 python 程序

简介

我们先来看看如何用 python 运行万年不变雷打不动的"Hello World"程序，这将教会你如何编写，保存，运行 python 程序.
有两种方式运行 python 程序 - 使用交互解释器或者执行源文件，现在我们就来分别看看这两种方式.

使用交互解释器

在命令行中输入"*python*"即可启动交互解释器.
如果 Windows 用户希望使用 IDLE，点击开始 -> 程序 -> Python 3.0 -> IDLE(Python GUI)即可.
现在输入 *print("Hello World")*，回车会你应该看到屏幕输出 Hello World 了.

```
$ 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 立即给出输出行！ 而你刚才输入的是一个 python 语句。
我们使用 print 打印任何提供给它的东西. 这里提供的是 Hello World 然后立即被打印到屏幕上。

如何退出交互解释器?

IDLE 或 Linux/BSD shell 下输入 ctrl-d 及可退出。Windows 命令行下则需输入 ctrl-z 并回车.

选择一个 python 编辑器

在编写 python 源文件之前我们需要一个文本编辑器。选择一个好用的编辑器非常关键，就象你去选购一辆车重要。
一个好的编辑器可以让你轻松的编写 python 程序，使得你的旅途既轻松又能快速而安全的到达目的地.
选择的基础条件之一是语法高亮，python 程序不同的部分将会赋予不同的颜色，因此你能够更容易的识别代码，

更容易在脑中形象化它的运行过程。
如果你是 Windows 系统，我建议使用 IDLE. 它不仅提供语法高亮还拥有许多其他特性，比如在 IDLE 中运行你的程序.
一个值得特别注意的地方: **不要使用记事本** - 这是一个糟糕的选择因为它不支持语法高亮，更重要的不支持自动文本

缩进(后面你就会看到这个功能很重要). 而好的编辑器例如 IDLE(或 VIM)会自动帮你做到这点.
Linux/FreeBSD 下有很多选择, 如果你是编程新手你也许会想要使用 geany. 它提供了图形用户界面并有按钮执行编译和

运行功能 python 程序的功能. 对于经验丰富的程序员, 一定已经使用 Vim 或 Emacs 了. 毋需多言这是两个超强的编辑器,

使用它们编写 python 程序肯定另你受益不少吧. 我个人就使用 vim 编写我的大多数程序. 如果你是菜鸟程序员, 可以使用 Kata 它也是我的最爱之一.

假如你乐于花点时间学习 Vim 或 Emacs, 我强烈建议你学习其中的一个, 从长远看这将对你非常有益.
在本书中既可以使用 IDLE, 也可以使用集成开发环境(IDE)或是自己中意的编辑器.
IDLE 在 Windows 和 Mac OS X 的 python 安装包中已经默认安装,
Linux 同样可以得到它的安装包
(http://love-python.blogspot.com/2008/03/install-idle-in-linux.html)
BSD 系统的 IDLE 分别在各自的资料库中(repositories)
我们将在下节探究 IDLE 的用法, 更具体的资料详见 IDLE 文档
(http://www.python.org/idle/doc/idlemain.html).
如果你还希望研究下其他可供选择的编辑器, 请参阅 python 编辑器列表
(http://www.python.org/cgi-bin/moinmoin/PythonEditors)
另外你也可以在选择 python IDE, 详见支持 python 的 IDE 列表

(http://www.python.org/cgi-bin/moinmoin/IntegratedDevelopmentEnvironments)
一但你开始编写大型 python 程序, IDE 真的非常有用.
最后我再重申一遍, 请选择一个合适的编辑器, 它能让 python 程序的编写更加有趣和容易.

使用源文件

现在让我们回到程序编写, 有一个传统, 当学习一个新的程序设计语言时, 第一个编写并运行的程序程序称做"Hello World",

它全部的功能只是简单的打印"Hello World". 正如 Simon Cozens 所说它是编程帝的咒语可以帮助人们更好的学习语言:)
启动你的编辑器, 输入下面的程序并将其保存为文件 *helloworld.py*.
IDLE 用户则点击文件(File) → 新建窗口(New Window)然后输入下面的程序并点击文件(File) → 保存(Save)

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

print('Hello World')
```

启动一个 shell(Linux 终端或 DOS 提示符(注:即 windows 命令行)输入命令 *python*

helloworld.py
IDLE 同学点击运行(Run) → 运行模块(Run Module)或者使用键盘快捷键 F5.

程序输出如下.

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

如果你得到上面的输出，恭喜恭喜！你已经成功的运行了你的第一个 python 程序.
否则如果出错了，请按照上面的代码仔细核对你的输入并再次运行程序.
另外要注意是 python 是大小写敏感的，也就是说 print 和 Print 是不同的，后者是大写而前者是小写.
还要确保每行的第一个字符前不要有空格或 tab，我会在后面解释这点的重要性.

程序如何工作

程序的前两行为注释 - 符号#右面的任何字符都会被当作注释，注释的主要功能是作为程序阅读者的笔记.
python 注释是可选的但第一行是个例外，它被称做事务行(shebang line) - 以*#!*开头后跟程序运行的位置.
在 linux/unix 下这会告诉系统本程序将在这个指定的 python 解释器下运行. 这点在下节会有详细解释.
另外注意通过在命令行直接指定 python 解释器，你总是可以让程序运行在任何平台上，就象命令 *python helloworld.py*

重点

合理运用注释说明程序的重要细节对于代码的阅读者非常有益，如此他们就可以轻松的理解程序的工作流程了.

记住，这个读者可能就是 6 个月后的你!

跟在注释后的是一个 python 语句. 我们调用的 print 是个函数，在这里它仅仅打印文本 "Hello World".
下一章我们会学习函数，你现在需要明白的是你在小括号中提供什么，print 就会在屏幕输出什么.
本例中，我们提供的是'Hello World'，它被称作字符串 - 不明白"字符串"是什么？没关系后面会详细研究这个术语.

执行 **python** 程序

本部分内容只适用 linux/unix 用户不过 windows 用户可能会好奇程序的第一行.
首先我们必须通过 chmod 命令为程序赋予可执行属性，然后才能运行它

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

这里，chmod 命令改变文件的模式使得系统中所有的用户都可以运行这个文件.
随后我们通过指定源文件的位置执行程序，./代表我们的程序在当前目录中.
为了更有趣一些，将文件重命名为 helloworld 并运行它. 你会发现程序仍然可以工作，

这是因为系统知道它现在必须使用源文件第一行指定的解释器运行这个程序.
要是你不知道 python 的位置呢? 那也可以在 linux/unix 上使用 env 程序. 修改代码的第一行为:

#!/usr/bin/env python

这样 env 程序就会依次查找将要运行这个程序的 python 解释器了.
迄今为止, 我们只要指定精确的路径就可以运行我们的程序了. 但是如果我们想让程序在任意位置运行怎么办?
当你运行任何程序时, 系统会查找环境变量 PATH 中指定的目录, 找到后执行之. 所以将程序拷贝到 *PATH* 中的某个目录下即可做到这点.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
$ cp helloworld.py /home/swaroop/bin/helloworld
$ helloworld
Hello World
```

我们可以通过 echo 命令显示 *PATH*, 前缀$告诉 shell 我们需要显示 PATH 的值.
可以看到*/home/swaroop/bin* 是 *PATH* 中的一个目录, 其中 swaroop 是我在我的系统中使用的用户名,

而在你的系统中也会有一个类似的目录, 当然用户名就是你自己的了.
作为另一种选择, 你也可以将一个新的目录加入 *PATH*, 这可以通过执行 *PATH=$PATH:/home/swaroop/mydir* 做到.
其中'*/home/swaroop/mydir*'是被添加的目录.
如果你需要让程序在任何时间任何地方运行, 这个方法会非常有效.

这就象是你自己创建的命令, 好比在 linux 终端或命令行下的 cd 或其他命令一样.

提示

在 python 中, 程序脚本或者软件的概念是相同的.

获取帮助

如果你需要获取 python 中任何函数或语句的快速信息, 可以使用内建的 help 功能.
当使用交换解释器的时候这会非常有用. 例如运行 *run(print)* - 将显示内建 print 函数的帮助.

提示

按 q 退出帮助.

通过 help 你几乎可以获取 python 中的任何信息, 而输入 *help()*还可以得到 help 自身的帮助!
假如你需要 python 运算符的帮助信息, 例如 *return*. 这时你需要将其用引号括起来 *help('return').*

小结

现在你应该有能力轻松的编写, 保存并运行 python 程序了, 既然你已经成为 python 用户那么接下来让我们学习更多的 python 概念吧.

你肯定不满足于只打印"Hello World"吧? 你想要的更多 - 你希望得到一些输入，操纵它后再从中得到某些东西.
我们可以使用 python 中的常量和变量实现这些功能。

字面常量(literal constant)

字面常量的一个例子是数字诸如 *5, 1.23, 9.25e-3* 或字符串 *This is a string', "It's a string!".*
顾名思义，字面常量的重点在于"字面", 你直接以字面的意义使用它们。数字 *2* 永远是数字 2 绝不会是别的东西。
而常量代表它们永远不会被改变，因此它们全被称为字面常量

数字

python3 拥有 3 种数字类型 - 整数,浮点和复数(注: python3 的整数类型已经被统一了，再没有短整数和长整数的分别)

- 例如，*2* 是一个整数.
- 浮点数的例子如 *3.23, 52.3E-4. E* 代表 10 的幂，在这里 *52.3E-4* 等于 *52.3 * 10-4*
- *-5+4j, 2.3 - 4.6j* 属于复数

编程老鸟请注意

python3 没有"long int"类型，事实上 python 的整数可以任意大(注：只是理论上的，再咋地存储器也不是无限的)

字符串

字符串是一个字符序列，经常用来表示字词. 字词可以是英语也可以是 Unicode 标准支持下的任何语言.
而 unicode 支持世界上绝大多数语言
(http://www.unicode.org/faq/basic_q.html#16).

编程老鸟请注意

在 python3 中不存在纯 ASCII 字符串，因为 Unicode 是 ASCII 的父集。
如果你真的需要一个纯 ASCII 字符串，则需调用 *字符串.encode("ascii")*方法.
详见 StackOverflow 中的相关讨论
(http://stackoverflow.com/questions/175240/how-do-i-convert-a-files-format-from-unicode-to-ascii-using-python#175270).
(注：没必要看了过时了，调用 encode 后产生一个 bytes 对象，它不是字符串对象，python3 中的所有字符串都是 unicode)
我几乎可以保证在你编写的每个程序中都会用到字符串，下面将教你如何使用字符串，你要特别留心点阿.

单引号
你可以将字符串放到单引号中，所有空白字符即空格和 tab 都回原封保留。

双引号
双引号和单引号的效果完全相同。

三引号
在指定多行字符串的时候可以利用三引号 - (”””或'''), 在三引号中还能自由的使用单引号和双引号. 一个例子:

```
'''This is a multi
This is the second
"What's your name?
He said "Bond, Jam
'''
```

转义字符
假设你想要一个带单引号(')的字符串，你会怎么写呢？例如 What's your name?，
你不能写成*'What's your name?'*, 因为 python 会搞不清字符串是从哪开始又是从哪结束.
所以你需要告诉 python 字符串中间的单引号并不是字符串的结束标志。利用转义字符可以做到这点.
将单引号替换成 \' - 注意反斜杠，这样字符串就变成*'What\'s your name?'*了.
另一个办法是使用双引号*"What's your name?"*,

不过当你需要双引号的时候和单引号的情况类似，必须加上反斜杠 \", 而反斜杠也一样必须表示成\\
如果你需要一个双行字符串呢? 一个办法是使用前面提到的三引号或者使用换行的转义字符*\n* 开始新的一行,
还有一个有用的转义字符*\t* 表示 tab, 转义字符太多了我只说了常用的.
另一个值得注意的地方是在一个字符串末尾的反斜杠表示这个字符串将被合并到下一行, 例如:

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

上面的字符串等价于*"This is the first sentence. This is the second sentence."*.

原始字符串
如果你不希望 python 对字符串进行特别处理, 比如取消转义字符, 可以为字符串加上前缀 r 或 R.
例如 r"Newlines are indicated by \n".

字符串是不可变类型
这意味着创建一个字符串对象后就无法改变它了. 这听起来很糟糕, 其实非也, 后面见分解.

字面字符串连接
如果你将两个字面字符串相邻, 它们会被 python 自动合并到一起.

例如*'What\'s ' 'your name?'*会变为*"What's your name?"*

C++程序员请注意

python 没有单独的字符类型，也没这个必要我保证你不会为此烦恼.

Perl/PHP 程序员请注意

单引号和双引号字符串完全相同，没有任何区别

正则表达式用户请注意

永远使用原始字符串编写正则表达式，否则会需要大量的反斜杠，例如反向引用可以表示为*'\\1'*或 *r'\1'*.

字符串的 format 方法

有时我们需要使用额外信息构造字符串，这时 format 就很有用了.

```
#!/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))
```

Output:

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

format 如何工作?

一个 string 可以含有某些格式说明符，随后调用的 format 将用你提供给它的参数替换对应的格式说明符.

观察上面的例子，*{0}*对应变量 *name* 它也是 format 的第 1 个参数. 与之类似*{1}*对应 format 的第 2 个参数 *age*.

注意我们也可以使用字符串连接达到同样的目的: *name + ' is ' + str(age) + ' years old'*.

但是这种方式太乱啊，很容易出错. 而且需要手动将变量转换为字符串而 format 可以为我们代劳.

最后，使用 format 我们可以改变最终生成的字符串而不必修改传给 format 的变量，反之一样.

format 的本质就是将其参数替换到字符串中的格式说明符，下面是一些更复杂的使用方式:

```
>>> '{0:.3}'.format(1/3) # 小数点后保留 3 位
'0.333'
>>> '{0:_^11}'.format('hello') # 以下划线填充字符串到 11 位长, hello 中间对齐
'___hello___'
>>> '{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python')
# 关键字格式化
'Swaroop wrote A Byte of Python'
```

关于格式说明符的具体信息见 Python 增强提议 No.3101(http://www.python.org/dev/peps/pep-3101/).

变量

只使用字面常量很快就会感到无聊 - 我们需要某种方式存储并操作任何信息，变量就是其主要手段。
正如其名 - 变量的值可以变化，即可以用它们存储任何数据。
变量实际上只是计算机存储器的一部分，用于存储你给出的信息。
与字面常量不同，你需要访问这些变量，因此需要你给它们起名字。

标识符命名

变量是标识符的一个例子，而标识符用于标识某些东西的名字。下面是一些命名标识符识必须遵守的规则:

- 标识符的第一个字符必须是字母(包括大小写 ASCII 字母或 unicode 字符)或下划线('_')(注：没错，用中文试试哈)
- 后面的字符可以由字母(大小写 ASCII 字母或 unicode 字符)，下划线('_')和数字组成.
- 标识符是大小写敏感的，例如 *myname* 和 *myName* 是不同的，前者小写后者大写.
- 一些合法标识符的例子 *i*, *__my_name*, *name_23*, *a1b2_c3*, resumÃƒÆ'Ã†â€™Ãƒâ€ Ã¢â‚¬â„¢Ã¢â‚¬Å¡ÃƒÆ'Ã¢â‚¬Å¡Ã‚Â©_count.(注：不知道乱码还是某国语言...)
- 非法标识符的例子, *2things*, *my-name*, *"this_is_inquotes"*

数据类型

变量能容纳不同类型的值，被称为数据类型. 基本数据类型包括我们刚才讨论过的数字和字符串.
在后面的章节中我们将看到如何利用类创建我们自己的数据类型。

对象

记住，在 python 中一切皆对象，因此从广义上我们不说"某物"而说"某对象"

面向对象编程用户请注意

python 是很彻底的面向对象语言，它的一切包括数字，字符串和函数全是对象。

如何编写 **python** 程序

从现在开始，保存和运行一个 python 程序的过程如下:

1. 打开你钟爱的编辑器.
2. 输入本书的范例代码.
3. 用源文件注释中给出的名字保存文件。并且我遵循惯例为搜索 python 文件加上 py 后缀名.
4. 以命令 *python program.py* 启动解释器执行程序或者使用 IDLE. 当然你也可以使用前文介绍过的执行方法。

范例：使用变量和字面常量

```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)
s = '''This is a multi-line s
This is the second line.'''
print(s)
#输出:
    $ python var.py
    5
    6
    This is a multi-line stri
    This is the second line.
```

范例工作流程:

首先，我们使用赋值运算符(=)为变量 i 赋值字面常量 5，这行被称做一条语句，因为它表示需要完成某种操作，

这里是为变量名 *i* 绑定数值 *5*. 接下来我们使用 *print* 语句打印 i 的值.

(注: py2k 中 print 是条语句，说成"print 语句"可以接受，但 py3k 中 print 是个地道的函数，因此说"调用 print 函数"更为恰当)

然后我们为 *i* 的值加 *1* 并存回 *i*, 当打印它的值时得到意料中的 *6*。

与之类似，我们又为变量 *S* 赋值字面字符串然后打印它。

静态语言程序员请注意

变量被赋值后即可使用，这里无需变量声明或数据类型定义.

逻辑行和物理行

你所看到的代码中的行即为物理行，而 python 将一条语言当作一个逻辑行。

python 假设一个物理行对应一个逻辑行。

逻辑行的例子是形如 *print('Hello World')*的单条语句，如果它独占一行(就象你在编辑器看到的)，那么它也是一个物理行.

python 本身鼓励每条语句占一行，这样可读性更强。

如果你希望在一个物理行包含多个逻辑行，则必须使用分号(;)显式一个逻辑行/语句的结束.

例,

```
i = 5
print(i)
```

与下面的等效

```
i = 5;
print(i);
```

这样写的效果也一样

```
i = 5; print(i);
```

甚至可以这样写

i = 5; print(i)

虽然有多种写法，但我**强烈建议你一个物理行只包含一个逻辑行**．不过当逻辑行太长，也可以把它写成多个物理行。

这些办法都是为了尽可能避免分号，使得代码的可读性更强。事实上我**从不使用甚至都**没在 python 程序中看到过分号。

下面是一个将逻辑行扩展为多个物理行的例子，它被称做**显式行合并**(explicit line joining)

```
s = 'This is a string. \
This continues the string.'
print(s)
```

相应的输出为:

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

与之类似的，

```
pring\(i)
```

相当于

```
print(i)
```

另外某些情况下会导致**隐式行合并**(implicit line joining)，比如逻辑行使用了大中小括号，此时无需反斜杠。

在后面的章节我们会使用列表编写程序，届时你会看到这种用法。

缩进

在 python 中空白字符是非常重要的，更具体的说是**每行开头的空白字符十分重要**。这被称作**缩进**。

逻辑行开头的空白字符(空格和 tab)用于确定逻辑行的缩进级别，依此按顺序将语句分组。也就是说同组的语句**必须**拥有相同的缩进级别，而这些语句组被称作**块**。下一章我们会看到块的重要性。

现在你需要牢记的是错误的缩进会导致程序出错，举个例子:

```
i = 5
 print('Value is ', i) # 错误! 注意本行的开头多了一个空格.
print('I repeat, the value is ', i)
```

运行上面代码会得到下面的错误信息:

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

你可能注意到了程序的第 2 行的开头多了一个空格．错误信息告诉我们程序的语法是非法的，即程序的编写有问题．

这也意味着你不能随便开始新的语句块(除了主块，我们一直在主快内编写)．那么要如何开始新的块呢？后面的章节会有详解，比如控制流．

如何缩进

不要混用空格和 tab，否则在不同的平台可能无法正常工作．

我强烈建议你使用 tab 或 4 个空格进行缩进，谨记两者则其一然后**一直**用下去．

静态语言程序员请注意

python 总是使用缩紧表达语句块，这里没有括号啥事．运行代码 Run from __future__

import braces 了解更多
(注：表运行了，过时了无视吧，不过前句话没过时啊!)

小结

本章涉及到很多核心知识，下面会接触到更有趣的东东，比如控制流语句.
在此之前请确定你已经对本章的内容有足够的理解了.

简介

你编写的大多数语句(即逻辑行)都会包含表达式。例如 *2 + 3*。
一个表达式可以被分解为运算符和运算数。运算符用来完成某些操作，并以特殊的关键字形式表示之，
如符号+，同时运算符还需要操纵某些数据，这些数据被称为运算数，在 *2 + 3* 中，*2* 和 *3* 就是两个操作数。

运算符

我们先简短的浏览一下运算符即其用法:
注意，你可以通过 python 解释器交互式的计算示例中的表达示，例如测试表达示 *2 + 3*

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

(注: 后面的 x 和 y 分别代表第一操作数和第二操作数，单目运算符则只有 x)

运算符	名称	说明	示例
+	加法	两个操作数相加	3 + 5 得 8。‘a’ + ‘b’等于 ab。
-	减法	两个操作数相减或取负	-5.2 取负。50 – 24 得 26。
*	乘法	两个操作数相乘或重复字符串	2 * 3 得 6。 ‘la’ * 3 等于’lalala’。
**	幂运算	返回 x 的 y 次方	3 ** 4 得 81(即 3 * 3 * 3 * 3)。
/	除法	x 除 y	4 / 3 得 1.3333333333333333。
//	取整除	只返回商的整数部分	4 // 3 等于 1。
%	取模	只返回除法的余数	8 % 2 得 2。 -25.5 % 2.25 得 1.5。
<<	左移	将 x 的二进制位左移第 2 个操作数次 (每个数在内存中都以二进制数 0 和 1 的组合表示之)	2 << 2 得 8。2 的二进制表示为 10，左移 2 位 后等于 1000，1000 等十二进制数 8。
>>	右移	将 x 的二进制右移 y 次	11 > 1 得 5。 1 的二进制表示为 1011，右移 1 位 后等于 101，正好是十进制数 5。
&	按位与	x 与 y	5 & 3 等于 1。
|	按位或	x 或 y	5 | 3 等于 7。

^	按位异或	x 异或 y	5 ^ 3 等于 6。
~	按位取反	按位取反 x，等于-(x + 1)	~5 等于-6。
<	小于	判断 x 是否 y。 所有比较运算符都回返回 True(真)或 False(假)。 注意它们首字母是大写的	5 < 3 等于 False 而 3 < 5 等于 True。 比较运算符可以被随意连接起来： 3 < 5 < 7 得 True。
>	大于	判断 x 是否大于 y。	5 > 3 等于 True。 如果俩个操作数都是数字， 它们首先会被转化为相同的类型。 否则永远返回 False。
<=	小于或等于	判断 x 是否小于等于 y	x = 3; y = 6; x <= y 返回 True。
>=	大于或等于	判断 x 是否大于等于 y	x = 4; y = 3; x >= y 返回 True。
==	等于	判断 x 是否等于 y	x = 2; y = 2; x == y 返回 True。 x = ‘str’; y = ‘stR’; x == y 返回 False。 x = ‘str’; y = ‘str’; x == 返回 True。
!=	不等于	判断 x 是否不等于 y	x = 2; y = 3; x != y; 返回 True。
not	布尔非	如果 x 为 True 则返回 False, 如果 x 为 False 则返回 True	x = True; not x 返回 False。
and	布尔与	如果 x 为 False 则返回 False， 否则返回 y 的值	x = False; y = True; x and y 返回 False。 x 为 False 时 python 不会继续求 y， 因为 and 的左操作数为假则结果必定为 False， 这叫做”短路求值”。
or	布尔或	如果 x 为 True 则返回 True， 否则返回 y 的值	x = True; y = False; x or y 返回 True，布尔或同样应用 短路求值。

(注: 作者在这里的描述不够严谨，严格的讲 python 的 and 和 or 返回的是操作数, x and y 如果 x 为 False 则返回 x 否则返回 y, x and y 如果 x 为假则返 y 否则 x)

组合算数运算与赋值运算

对一个变量执行算数运算后在再把结果存回这个变量是很常见的，所以 python 提供了一种更简便的方式完成这类操作:

*a = 2; a = a * 3*

相当于

*a = 2; a *= 3*

注意这里 变量 = 变量 的形式变成了 变量 操作符= 表达式。

求值顺序

对于表达式 *2 + 3 * 4*，是先求加法呢还是先算乘法？
我们的中学数学知识告诉我们应该先算乘法(注：中学才乘法=.=)，
也就是说乘法拥有比加法更高的运算优先级。
下面的表给出了 python 运算符的优先级，顺序从低(最后集合)到高(最先集合)，
这意味着在一个表达式中，
python 会先计算表中较靠上的运算符然后再计算较靠下的运算符。
下表来自于 python 官方参考手册
(http://docs.python.org/dev/3.0/reference/expressions.html#evaluation-order)
所以其内容是完备的。
(注：原已地址已 out, 看我发的这个
http://docs.python.org/py3k/reference/expressions.html#evaluation-order).
另外适当的运用小括号将运算符和运算数分组以便更加明确的指出运算优先级
对程序的可读性大有好处。详见后面**改变求值顺序**一节。

操作符	描述
lambda	Lambda 表达式
or	布尔或
and	布尔与
not x	布尔非
in，not in	从属关系测试
is，is not	同一性测试(注: 比较两个变量是否引用相同对象)
<，<=，>，>=，!=，==	比较
|	按位或
^	按位异或
&	按位与
<<，>>	移位
+，-	加和减
+x，-x	正负号
~x	按位非
**	求幂
x.attribute	属性引用
x[index]	下标操作
x[index1:index2]	分片操作
f(arguments…)	函数调用
(expressions, …)	绑定或元组显示
[expressions, …]	列表显示
{key:datum, …}	字典显示

以上有些运算符我们还没有接触到，在后面的章节会有解释。
相同优先级的运算符被列在了同一列，比如+和-拥有同等的运算级。

改变运算符求值顺序

为了增强表达式可读性，我们可以使用小括号。
例如 *2 + (3 * 4)*明显比 *2 + 3 * 4* 更容易理解，而后者需要有运算符优先级的知识.
不过凡事都不能过度，适当的使用小括号增强可读性很好, 但是类似这样的表达式 *2 + (3 + 4)*就是滥用了。
另外使用小括号还有另一个好处，它可以帮助我们改变求值顺序。
比如你想在一个表达式中先求加法而后求乘法，可以这样写*(2 + 3) * 4*.
(注: 鄙人窃以为小括号的发明和主要用途就是改变求值顺序，作者有点颠倒…)

结合性

大多数情况下运算符从左到右结合，即拥有相同运算级的运算符从左到右求值，例如
2 + 3 + 4 相当与*(2 + 3) + 4*。
但有一些运算符，比如赋值运算符从右向左结合，*a = b = c* 相当与 *a = (b = c)*

表达式

举个例子:

```
#!/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
```

如何工作的：
矩形的长和宽被分别存入变量 *length* 和 *breadth*，并利用表达式分别求出面积和周长。
表达式 *length * breadth* 的结果被存入 *area*, 之后利用 *print* 函数进行打印。
但在第二个 print 调用中我们直接利用表达式 *2 * (length + breadth)*的结果。
另外注意 python 是如何”漂亮的”进行输出的，尽管我们没有在”*Area is*”和变量 *area*
之间指定空格，python 却为我们代劳了。因此我们得到一个清爽的输出并且可读性变

的更强(因为我们无需劳心被输出字符串间的空格问题)。这也是 python 如何让程序员
的生活变得更轻松的例子之一。

小结

我们已经学到如何使用操作符，操作数和表达式 – 它们都是任何程序的基础。接下来我们将会看到如何在语句中利用它们。

简介

迄今为止我们见到的所有程序总是含有一连串语句并且 python 忠实的顺序执行它们。
那么如何改变它们的执行顺序呢？例如你希望程序根据不同情况作出不同反应，按照当前时间分别
打印出'Good Morning'或'Good Evening'?
也许你已经猜到了，这需要使用控制流程语句，python 拥有 3 种此类语句，分别为 *if*，*for* 和 *while*。

if 语句

if 语句用来检查一个条件，如果条件为*真*则执行一个语句块(被称作 *if 块*)，否则执行另一个语句块(被称作 *else 块*)。
其中 else 分支是可选的。
范例：

```
#!/usr/bin/python
# Filename: if.py
number = 23
guess = int(input('Enter an integer : '))
if guess == number:
    print('Congratulations, you guessed it.') # 新块开始处
    print('(but you do not win any prizes!)') # 新块结束处
elif guess < number:
    print('No, it is a little higher than that') # 另一个块
    # 你可以在一个块里做任何你想做的。。。
else:
    print('No, it is a little lower than that')
    # 只有 guess > number 才会执行到此处
print('Done')
# if 语句执行完毕后，最后这条语句永远会被执行
```

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

程序流程：

本程序将用户猜测的数对比被猜的数(注: 23)。
我们可以将被猜数定为任何整数。然后利用 *input()*函数得到用户猜测的数。
函数只是可复用的程序片段，下章我们会学到更多关于函数的知识。
我们为内建 *input* 函数提供一个字符串，函数会将它打印到屏幕，一但我们输入了什么并按下回车，
input 就会将输入当作一个字符串返回，之后利用 *int* 我们将其转换为一个整数并存入变量 *guess* 中。
实际上 *int* 是一个类，现在你只需知道它能将一个字符串转换为整数(假设字符串包含一个合法的文本
形式的整数)。
接下来我们比较用户猜测的数与被猜的数，如果它们相等我们就打印一个成功信息。注意我们
使用缩进告诉 python 语句分别属于哪些块，这就是为什么缩进对于 python 非常重要。
我也希望你能做到前文提到的"缩进一致性"规则，你可以吗?
同时也要注意到 if 语句的结尾包含一个分号 – 它指示 python 分号其后将跟随一个语句块。
紧接着我们检查猜测的数是否小于被猜数，如果是，则告诉用户猜测的数大了一些。
这里我们使用的是 *elif* 分支，事实上它把两个相关的 *if else-if else* 语句组合成一个 *if-elif-else* 语句。
这样做不仅使得程序更加简洁也降低了缩进数量。
同样，*elif* 和 *else* 语句必须在逻辑行的结尾写上冒号，其后是与之对应的语句块(当然还要有相应的缩进)
最后你也可以在 if 语句中插入另一个 *if-block* 块，这叫做嵌套的 if 语句。
上面说过 elif 和 else 是可选的。一个最简单合法的 if 语句如下：

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

当 python 执行完 if 语句及其相关的 elif 和 else 分支，控制权会转移到包含此 if 语句的语句块的下一条语句。
本例中，这个块是主块，程序从此块开始执行，下一个条语句为 *print('Done')*。
随后 python 看到程序的结尾并简单的结束运行。
尽管程序非常简单，我也指出了这个简单程序中你所有应该注意的地方。
这一切十分易懂(有 C/C++背景更是如此)，但在最初你还是要引起注意，等到用熟以后就会感到自然，顺手了。

C/C++程序员请注意

python 没有 switch 语句，你可以使用 *if...elif...else* 语句达到同样的目的(有时用字典代替会更加快捷)。

while 语句

只要条件为真，while 语句允许你不断的重复执行语句块。
while 语句是所谓循环语句的一个例子，它还可以拥有一个可选的 else 分支。
范例：

```
#!/usr/bin/python
# Filename: while.py
number = 23
running = True
while running:
    guess = int(input('Enter an integer : '))
    if guess == number:
        print('Congratulations, you guessed it.')
        running = False # this causes the while loop to stop
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
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
```

工作流程：
本例中我们继续玩猜数字游戏，不过更先进的是用户可以不断的猜直到猜对为止 – 如此就不会象
上一个范例那样用户每猜一次都运行一遍程序了。这正好可以恰当的示范 while 语句的使用。
我们将 input 和 if 语句移进 while 循环内，并在循环开始前将变量 *running* 设为 *True*。
最初我们检查变量 running 是否为 True 并进入 while 块执行之。while 块执行完毕后继续检查条件，即 running 是否为真。
如果是则再次执行 while 块，否则可选的 else 块将被执行。完毕后再执行下一条语句。

只有 while 语句的条件为 False 时 else 才会被执行 – 就算条件第一次被检查时也是如此。
因此除非使用 *break* 跳出循环，否则 while 循环的 else 分支肯定会被执行。
True 和 *False* 被称为 *Boolean* 类型，你可以分别把它们看作是值 1 和 0。
(注：只是"看作"而已，不要认为它们完全等于 0 和 1)
写给 C/C++程序员
记住，while 循环可以拥有 else 分支

for 循环

for...in 是另一种循环语句，用来遍历序列对象，也就是说遍历序列中的每个元素。
至于序列对象，你现在只要记住序列只是元素的集合就可以了。
范例：

```
#!/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
```

执行流程：
在这个程序中，我们打印了一个数字序列。而这个序列由内建 *range* 函数产生。
根据我们提供的两个数字，range 返回一个开始于第一个数字而结束于第二个数的序列，例如
*range(1, 5)*产生序列[1, 2, 3, 4]。默认的 range 步长为 1。如果我们传给它第三个数字，则相当于设置步长。
例如 range(1, 5, 2)产生[1, 3]。记住 range 函数止步于我们提供的第 2 个数字，即不包含第 2 个数。
然后 for 循环对其进行迭代 – *for i in range(1, 5)*等价于 for I in [1, 2, 3, 4]就象将序列中的每个数
字一次一个的(或对象)赋给 i，而每次赋值都会执行一遍 for 语句块。本例中我们仅仅打印 i 的值。
记住，else 部分同样是可选的，除非使用 break 语句跳出循环否则它总是在循环结束时执行一次。
还应记住，*for...in* 可以工作于任何序列，这里我们使用的是内建函数 range 产生的数字列表，
但我们也可以使用任何种类的对象组成的任何种类的序列！后面的章节会有具体解释。

C/C++/Java/C#请注意

python 的 for 循环完全不同于 C/C++的 for 循环。

C#程序员应该已经注意到它类似于 C#中的 *foreach* 循环。

而 Java 程序员也应该注意到了 Java1.5 的 *for(int i : IntArray)*与之很相似。

如果你想实现 C/C++中的 *for(int i = 0; i< 5; i++)*，python 中只需编写 *for i in range(0, 5)*。

如你所见，python 的 for 循环更简单，更富于表达力也更不容易出错。

break 语句

break 语句用于跳出循环，即停止循环语句的执行，即使循环条件还没有变为 False 或者序列的遍历尚未完成。

一个需要特别注意的地方是如果你使用 break 跳出 for 或 while 循环，那么相关的 else 块不会被执行。

范例

```
#!/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
```

工作流程：

这个程序不断的取得用户输入，并打印每次输入的长度。通过一个特殊条件判断用户

输入是否等于’quit’，如果是则使用 break 跳出循环，随后来到程序尾则程序终止。

而输入字符串的长度可以通过内建函数 len 得到。

记住 for 循环同样可以使用 break 语句。

Swaroop Poetic Python

我在范例中输入的是一首自赋的小诗，它的名字叫 Swaroop's Poetic Python：

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

(注: Swaroop 是本文作者，不翻译了，各位用个人独有的悠悠情愫”淫”起来吧，啊。。。。啊。。。。)

continue 语句

语句 *continue* 告诉 python 跳过当前循环语句块的剩余部分执行下次迭代。

范例：

```
#!/usr/bin/python
# Filename: continue.py
while True:
    s = 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
```

工作流程：

本例接收用户输入，只有当输入至少为 3 个字符长时才会进行处理。

因此我们使用内建 len 函数取得输入长度，当小于 3 时利用 continue 语句跳过程序的余下部分。

否则程序的余下部分会被执行，在这里可以进行我们希望做的任何种类的处理。

注意，continue 语句同样能够配合 for 循环使用。

小结

我们已经知道如何使用 python 中的 3 个控制流语句了 - if, while, python 及其相关的 break, continue 语句。

它们是 python 中常用的部分，因此有必要熟练掌握。

接下来，我们会看到如何创建和使用函数

简介

函数是程序的可复用片段，允许你为语句块赋予名字之后在程序的任何地方运行它们任意次，这称做函数调用。
我们已经使用过一些内建函数，例如 len 和 range 等。
函数也许是任何有意义的软件中最重要的构件，所以我们将在本章探究函数的方方面面。
函数以关键字 *def* 定义，其后紧跟函数名，由一对小括号闭合的形参，最后以冒号结束定义行，
定义行下面的是函数体，它是一个语句块。
听着有点复杂，其实定义起来是很简单的，见下面的例子：
范例：

```
#!/usr/bin/python
# Filename: function1.py
def sayHello():
   print('Hello World!') # 语句块也就是函数体
# 结束函数定义
sayHello() # 调用函数
sayHello() # 再次调用
```

输出:

```
   $ python function1.py
   Hello World!
   Hello World!
```

工作流程：
我用使用上面讲解的函数定义语法定义了一个名叫 *sayHello* 的函数，这个函数没有形参因此小括号里也就没有
定义变量。函数形参只是提供给函数的输入，所以我们可以为函数传递不同的形参而后得到相应的结果。
注意我们调用了相同的函数两次，这也意味着我们无需多次编写相同的代码(注：函数体)。

函数形参

一个函数可以拥有形参，它们是你提供给函数的值，因此函数就可以利用这些值进行一些操作了。
函数形参非常类似变量，只是这些变量是在我们调用函数时定义的并在函数运行前被赋值。
形参在函数定义中的小括号内指定，并以逗号分隔。当我们调用函数时以同样的方式提供形参值。
注意下面两个术语的区别 – 函数定义中给定的参数叫做*形参*，而在函数调用提供的值叫做 – *实参*
范例：

```
#!/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) # 直接提供字面值
x = 5
y = 7
printMax(x, y) # 以变量作为实参
```

输出：

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

工作流程：

我们定义的函数叫做 *printMax*，它需要两个形参，分别叫做 *a* 和 *b*。
函数利用简单的 *if...else...*语句找出两者中较大的数并将其打印。
第一次调用 printMax 时我们直接使用数字作为实参，而第二次调用时我们使用变量。
*printMax(x, y)*促使实参 *x* 的值赋给形参 *a*, *y* 赋给 *b*。
两次调用中，pringMax 的工作方式完全相同。

局部变量

在函数内声明的变量与在函数外的同名变量没有任何关系，即变量名对于函数是局部的。
这被称作变量的*作用域*，变量的作用域开始于它们所在块中定义它们的定义点处。

范例：

```
#!/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：

在函数内部，我们首先使用变量 *x* 的值时，python 引用的是函数形参 x 的值。接下来，我们为 x 赋值 *2*，因为 x 是这个函数的局部名字，所以当我们在函数中使用 x 的时候，
不会影响到主块中的 x ，在最后的 print 调用中我们打印了主块中 x 的值也证明了这点。

使用 global 语句

当你希望为程序的顶级名字赋值时(没有定义在任何其他作用域中的变量，比如函数或类作用域)，
(注: python 有名字空间的概念，名字空间建立起名字与实际对象(/数据)的映射关系，
这里的”名字”指的是名字所对应的对象)。这时你必须告诉 python，名字不是局部而是全局的。
通过 *global* 语句可以做到这点，否则是不可能对一个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(但要假设函数内没有同名变量)，不过并不鼓励这种用法而是应该
尽量避免，这会让代码的读者难以弄清变量到底定义在哪？

范例：

```
#!/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
```

工作流程：

global 语句用于声明 *x* 是一个全局变量 – 因此我们在函数内为 x 赋值后变化也会反映在主块中对 x 的使用中。
你也可以使用一个 global 语句指定多个全局变量，比如 *global x, y, z*。

使用 nonlocal 语句

我们已经看到如何存取局部和全局变量。还有另一类叫做"*非局部*"的作用域，它介于局部和全局之间。
当在函数内定义函数时你会注意到非局部作用域。
因为 python 中的一切只是可执行代码，所以你可以在任何地方定义函数。让我们看一个例子：

```
#!/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
```

工作流程：
当我们处于 *func_inner* 函数中时，*func_outer* 第一行定义的变量 *x* 既不是局部也不是全局变量。
这时通过 *nonlocal x* 我们声明这个 *x* 是非局部的，所以我们才能够存取它。
试着将 *nonlocal x* 改为 *global x* 观察这两种用法有什么不同。

默认实参值

对于一些函数，你可能希望它们的形参是可选的，并当用户没有为这些形参提供值的时候给它们一个默认值。
这需要借助默认实参值。默认实参值在函数定义时通过为形参名赋一个默认值实现。
注意默认实参值应该是一个常量，更确切的应该是一个不可变类型 – 后面的章节会有具体解释，现在只要记住这
点就可以了。
范例：

```
#!/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
```

工作流程：

函数 *say* 用于以指定次数打印指定字符串. 如果我们没有指定次数则默认的字符串只被打印 *1* 次。
我们通过为形参 *times* 指定一个默认实参值 *1* 实现这个功能。
第一次调用 *say* 时，我们只提供了被打印的字符串然后函数只打印一次。
而第二次调用时，我们不仅提供了被打印的字符串还提供了实参 *5* 以表明我们需要字符串被打印 5 次。

重点：
只有在实参列表靠后的参数才能拥有默认实参值，即你不能先声明带有默认实参值的形参再声明不带有默认实参值的参数。
这是因为实参值是根据形参的位置赋给形参的，例如：*def func(a, b = 5)*合法，但 *def func(a = 5, b)*就非法了。

关键实参

如果你有一些函数拥有许多参数，但你只想使用其中的几个，这时你可以通过形参名为其赋值。
这被称做*关键实参* - 使用形参名(关键字)为函数指定实参而不是我们一直使用的通过位置指定实参。
这样做有两个优点，首先函数用起来更简单，因为我们不用操心实参的顺序了。
其次，可以只为我们感兴趣的形参赋值，如果其它参数带有默认实参值的话。
范例：

```
#!/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
```

工作流程：
函数 func 拥有一个不带有默认实参值的参数，后跟两个带有默认实参值的参数。
第一次调用 *func(3, 7)*，形参 *a* 得到值 3，*b* 为 *7*，*c* 为默认值 *10*。
第二次调用 *func(25, c=24)*, 因为 *a* 在参数表中所处的位置，它得到 *25*。

而 c 通过引用其名字得到 *24* 也就是关键字实参的功能。*b* 为默认值 *5*。
第三次调用 f*unc(c=50, a=100)*, 我们只使用关键字实参指定形参值。
注意尽管在函数定义中 a 比 c 更早定义，但我们仍然可以先指定 c 再指定 a。

可变参数(VarArgs)

TODO
因为我们还没有讨论列表和字典，我是不是应该将这部分放到后面的章节？

有时你可能希望编写一个可以接受任意多个形参的函数，使用星号可以帮你做到：

```
#!/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
```

工作流程：
当我们以星号声明一个形参比如*param，那么这个参数点之后的所有实参会被收集成一个列表，
本例中这个列表叫做 param。与之类似如果我们以双星号声明一个形参，它会被收集成一个关键字实参字典。
后面的章节我们会研究列表和字典。

只能以关键字赋值的形参(Keyword-only Parameters)

(注：为方便理解和翻译，以后直接使用英文术语 keyword-only)
如果我们希望某些关键字形参只能通过关键字实参得到而不是按照实参的位置得到，可以将其声明在星号形参后面：

```
#!/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))
# 引发错误，因为我们没有为 vegetables 提供默认实参值
```
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
```
工作流程：

在星号形参后面声明的形参导致它成为 keyword-only 实参。

如果没有为这些实参提供一个默认值，那么必须在调用函数时以关键字实参为其赋值，否则将引发错误。

如果你只需要 keyword-only 实参但不需要星号实参，那么可以简单的省略星号实参的实参名。

例如 *def total(initial=5, *, vegetables)*。

return 语句

return 用于从函数返回，即跳出函数。也可以利用 return 语句从函数返回一个值。

范例：
```
#!/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
```
工作流程：

maximum 函数用于返回参数中的最大值，本例中我们以数字作为参数调用它。

函数使用一个简单的 *if...else* 语句比较出最大值并使用 return 语句将其返回。

注意一个不带有返回值的 return 语句相当于返回 *return None*。

None 是 python 的一个特殊类型，代表空。例如如果一个变量的值为 None 则代表它不存在值。

每个函数的末尾都隐含的包含一个 return None 语句除非你编写了自己的 return 语句。

你可以通过 *print(someFunction())*证实这点，someFunction 是一个没有显式使用 *return* 语句的函数。例如：
```
def someFunction():
    pass
```
其中 *pass* 语句用来指示一个空语句块。

提示

python 已经包含了一个被称作 max 的内建函数，它的功能即是寻找最大值，所以尽可能的使用这个函数吧。

DocStrings

python 拥有一个俏皮的特性被称作文档字符串，通常它被简称为 docstrings。文档字符串是一个你应该利用的重要的工具，因为它帮助你更好的注释程序使得程序更易于理解。

更神奇的是你甚至可以在程序运行时取得文档字符串！

范例：

```
#!/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.
```

工作流程：

一个函数的第一个逻辑行的字符串将成为这个函数的文档字符串。

注意类和模块同样拥有文档字符串，在后面相应的章节我们会学到它们。

根据惯例，文档字符串是一个多行字符串，其中第一行以大写字母开头，并以句号结尾。

接下来的第二行为空行，从第三行开始为详细的描述。

我强烈建议你在你的正规函数中遵循这个编写文档字符串的惯例。

我们可以通过使用函数的*__doc__*属性(注意双下划线)存取 *printMax* 的文档字符串。

记住 python 中的一切都是对象，其中也包括函数。在后面的类一章我们会学到更多。

如果你在 python 使用过 help()，其实你已经看到过文档字符串的应用了！

help()只是取出函数的*__doc__*属性，然后以一种整洁的方式显示给你。

你可以用上面的函数作个实验 – 在你的程序中包含 *help(printMax)*即可，记住按 *q* 键退出 help。

函数注解(Annotations)

函数还拥有另一个被称作*函数注解*的高级特性，对于附加额外的形参和返回值信息非常有用。
因为 python 语言本身并不提供这样的功能（甩给了第三方库，具体实现方式第三方库说了算）。
所以在我们的讨论中决定跳过这一特性。如果你对函数注解有兴趣可以参见 python 增强提议 No.3107
(http://www.python.org/dev/peps/pep-3107/)(注：python3 已经支持这个特性了)

小结

我们已经看到了函数的方方面面，但注意这些不是函数的所有方面。
不过现有的关于 python 函数的知识已经足够应付日常应用了。
接下来我们将学习如何使用和创建 pthon 模块。

简介

现在你已经知道通过定义函数可以在你的程序中复用代码。但当你想在你编写的其他程序中复用大量函数怎么办呢？
也许你可以猜到了，办法就是利用模块。
有各种编写模块的方式，但最简单的方式是创建一个以.py 为后缀的文件并包含所需的函数与变量。
另一种方式是以编写 python 解释器的本地语言编写模块。
例如 C 语言编写的模块被编译后可供运行于标准 python 解释器上的 python 代码使用。
模块可以被其它程序导入以使用其提供的功能。这也是为什么我们可以使用 python 标准库。
我们先来看看如何使用标准库模块。
范例：

```
#!/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']
```

工作流程：
首先我们使用 *import* 语句导入 *sys* 模块。本质上这告诉 python 我们希望使用这个模块。
sys 模块包含 python 解释器与其工作环境(即系统)相关的功能。
当 python 执行 *import sys* 语句时，它将查找 sys 模块。本例中 sys 是内建模块之一，因此 python 知道在哪能找到它。
如果导入的不是一个编译模块，即不是用 python 编写的模块，python 解释器会在变量 *sys.path* 中列出的目录中查找它。
(注：if it was not a compiled module i.e. a module written in Python)

如果模块被找到，这个模块中的语句将被执行然后你就可以使用它了(注: 只有顶级语句才会执行，也就是主块中的语句)。
注意一个模块只有在第一次导入时会被初始化。
sys 模块中的 *argv* 通过点号引用即 *sys.argv*。它清晰的指出这个名字是 sys 模块中的一部分。
这种语法的另一个优势是不会和你的程序中的同名 argv 变量发生冲突。
变量 sys.argv 是一个字符串列表(后章会详细解释列表)。
具体说 sys.argv 是一个包含命令行参数的列表，也就是使用命令行传递给你的程序的参数。
如果你在使用 IDE 编写程序，请在菜单中查找为程序指定命令行参数的方法。
这里，当我们执行 *python using_sys.py we are arguments* 时，我们以 *python* 命令运行 *using_sys.py* 模块，其后的内容是传递给程序的参数。
python 将它们存到 sys.argv 以供我们使用。
记住，被运行脚本的脚本名永远是 sys.argv 的第一个参数。
所以本例中 *sys.argv[0]*为*'using_sys.py'*，*sys.argv[1]*为*'we'*，*sys.argv[2]*是*'are'*, *argv[3]*是*'arguments'*。注意 python 下标从 *0* 开始而非 *1*。
sys.path 包含一个目录名列表指示从哪里导入模块。
观察程序输出，sys.path 的第一个字符串为空 – 其指出当前目录也是 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导入当前目录下的模块，否则你就必须将你的模块放到 sys.path 列出的目录中的一个了。
注意程序在哪个目录运行的，这个目录就是这个程序的当前目录。运行 *import os; print(os.getcwd())* 可以看到你的程序的当前目录。
(注：windows 下 sys.path[0]可能不为空，而是显式指出当前路径)

字节编译文件 .pyc

导入模块是一个相对昂贵的操作，所以 python 使用了一些技巧加速这个过程。
一个办法是创建后缀为**.pyc** 的字节编译文件用于将程序转换为中间格式。(还记得介绍 python 如何工作的那一节吗？)
当你下次从其他文件导入模块时 pyc 文件会非常有用 – 它将大大增加导入速度，因为导入模块的部分操作已经预先完成了。
并且这个字节编译文件仍然是平台无关的。

注意
.pyc 文件一般被创建在与其对应的.py 文件所在的相同目录下。如果 python 没有这个目录的写权限，则.pyc 文件不会被创建。

from…import…语句

如果你希望将变量 *argv* 直接导入到你的程序中（避免每次输入 *sys.*），那么可以使用 *from sys import argv* 语句。
如果希望导入 sys 模块中的所有名字，则 *from sys import* *可以做到。此语句可以用于任何模块。

通常你应该避免使用这个语句并用 *import* 语句代替之，因为使用后者可以避免名字冲突，程序的可读性也更好。

模块的__name__属性

每个模块都有一个名字，并且通过模块中的某些语句可以得到这个模块名。
在一些想要搞清模块是独立运行还是被导入的情况下，这会非常方便。
如前所述，当模块第一次导入时模块中的代码会被执行。我们可以据此改变模块独立执行时的行为方式。
这可以通过模块的*__name__*属性做到。(注：独立运行是指程序最开始运行的那个脚本文件(/模块))

范例：

```
#!/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
   >>>
```

工作流程：

每个 python 模块都有自己的*__name__*定义，如果它是*'__main__'*则暗示模块为独立运行，我们可以进行一些适当的处理。

制作你自己的模块

创建你自己的模块很简单，其实你一直在这样做！因为每个 python 脚本都是一个模块。你只需确保它带有*.py* 扩展名即可。
下面的例子会让你对其有一个清晰的认识：

范例：

```
#!/usr/bin/python
# Filename: mymodule.py
def sayhi():
   print('Hi, this is mymodule speaking.')
__version__ = '0.1'
# 结束 mymodule.py 编写
```

上面就是一个简单的模块，如你所见，这和我们平时的 python 程序相比没有什么特别之处。

记住模块应该放到导入它的那个程序所在的目录下，或者放到 *sys.path* 列出的目录之一中。

```
#!/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：

注意我们同样使用点号访问模块成员。

python 很好的重复利用了相同的符号，带来独特的’Pythonic’感受，这样我们就不必学习更多的语法知识了。

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__)Python en:Modules 59
```

mymodule_demo2.py 与 *mymodule_demo* 的输出完全相同。

注意，如果导入 *mymodule* 的模块中已经存在同名的*__version__*，则将发生名字冲突。

事实上这很可能发生，因为每个模块都用*__version__*声明它的版本是一种常见的做法。

因此建议你优先考虑 *import* 语句，虽然它可能会让你的程序变的更长一些。

你同样可以使用：

*from mymodule import **

这将导入模块的所有公有名字，例如 *sayhi*，但是不会导入*__version__*因为它以双下划线开头。

Python 之禅

python 的一个指导原则是”清晰的好过隐晦的”. 执行 *import this* 可以看到完整内容。

这里的讨论列出了每个原则的范例

(http://stackoverflow.com/questions/228181/zen-of-python)

dir 函数

你可以使用 *dir* 函数列出一个对象定义的所有标识符。例如对于一个模块，标识符包括函数，类，变量。

当你为 *dir()*函数提供一个模块名，它将返回定义在其中的所有名字。

当 *dir()*的参数为空时，返回定义在当前模块中所有名字。

范例：

```
$ python
>>> import sys # 得到 sys 模块的属性列表
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__package__', '__stderr__', '__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', 'platform',
'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit',
'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'warnoptions',
'winver']
>>> dir() # 得到当前模块的属性列表
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
>>> a = 5 # create a new variable 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']
>>> del a # 删除名字 a
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

工作流程：

首先，我们通过被导入的 *sys* 模块应用 *dir* 函数. 可以看到 *sys* 包含巨多的属性。接下来，我们不为 dir 函数提供参数。默认的它返回当前模块的属性列表。注意被导入模块列表也是当前模块列表的一部分。

为了观察到 dir 确定起作用了，我们定义一个新变量 *a* 并为其赋值然后检验 dir 的返回值，我们发现在返回的列表中确实出现

了一个与变量 *a* 同名的值。在我们使用 *del* 语句删除当前模块的变量/属性后，改变再次反映到了 dir 函数的输出上。

del 注解 – 这个语句用于删除一个变量/名字，在本例中，*del a* 之后你就无法访问变量 *a* 了 – 就像它从来没有存在过一样。

注意 *dir()*函数可用于任何对象。例如执行 *dir (print)*学习更多关于 *print* 函数的属性，或是 *dir(str)*列出 *str* 类的属性。

包

如今，你必须开始留心组织你的程序层次了。

变量在函数内部，函数和全局变量通常在模块内部。那么如何组织模块呢？这就轮到*包*登场了。

*包*仅仅是包含模块的文件夹，并带有一个特殊的文件*__init__.py* 用于指示 python 这个文件夹是特殊的，因为它包含 python 模块。

让我们假设你需要创建一个叫做*'world'*的包，里面包括诸如*'asia'*，*'africa'*等的子包。
下面告诉你应该如何组织文件夹结构：

```
- <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
```

包只是用来有层次的组织模块。你会在标准库中看到它的很多应用。

小结

就象函数是程序的可复用部分一样，模块是可复用的程序。
包用于组织模块的层次结构。python 自带的标准库正是一组包和模块的范例。
我们已经看到如何使用这些模块，并且知道如何创建自己的模块。
接下来，我们将学习一个比较有趣的概念 – 数据结构。

简介

数据结构基本上就是 – 可以将一些数据结合到一起的结构，换言之用于存储一组相关的数据。
python 拥有 4 种内建数据结构 – 列表，元组(tuple)，字典和集合。
我们将看到如何它们，它们又是怎样使我们的编程生涯变的惬意~

列表

列表是一种用于保存有序元素集合的数据结构，即你可以在列表中存储元素序列。
考虑一个购物清单，上面有你需要购买的物品列表，只不过你可能希望以行分隔它们而到了 python 变成了逗号。
这样想来就容易理解列表了吧。
列表元素应该被封闭在方括号中，这样 python 才会明白你指定的是一个列表。
一但列表创建完毕，你可以对其元素进行添加，删除和搜索。
正因为可以执行添加和删除操作，我们将列表称作可变类型，即这种类型可以被修改。

对象和类快速简介

尽管我一直推迟讨论对象和类，但现在需要对其进行少量的说明好让你更好的理解列表。后面会在相应的章节深入研究类和对象。
列表是使用对象和类的一个例子。当我们为变量 *i* 赋值时，例如赋值 *5*，这相当于创建一个 *int* 类(类型)的对象(实例)*i*。
事实上你可以阅读 help(int)的输出更好的理解它。
一个类同样可以拥有方法，即函数，而且它们只能应用于这个类。并且只有当你拥有一个类的对象时才能使用这些功能。
例如，python 为列表类提供了一个 *append* 方法允许你将新的元素添加到列表尾。举个例子，*mylist.append('an item')*将字符串添加到列表 *mylist* 的尾部。注意要使用点号访问对象的方法。
一个类还可以拥有字段，而字段只不过是专门应用于一个类的变量而已。当你拥有对应类的对象时就能使用这些变量/名字了。
字段同样利用点号访问，例如 *mylist.field*。
范例：

```
#!/usr/bin/python
# Filename: using_list.py
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']
print('I have', len(shoplist), 'items to purchase.')
print('These items are:', end=' ')
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']
```

工作流程：

变量 *shoplist* 是某个购物人的购物清单。

我们只在 shoplist 中存储被购买物品的名字的字符串，但你也可以为列表增加任何其它种类的对象，包括数字甚至是其它列表。

我们通过 *for...in* 迭代列表元素，现在你一定意识到一个列表也是一个序列了吧。有关序列的特点我们会在后节讨论。

注意 *print* 的 *end* 关键字实参，它指定我们希望以空格结束输出而不是通常的换行。

接下来我们使用列表对象的 *append* 方法为列表添加一个新的元素。

为了确定元素真的被添加进去了，我们简单的将列表传给 print 函数，print 函数整洁的将列表内容打印出来。

随后我们使用列表的 *sort* 方法对列表进行排序，紧记 sort 会影响列表本身而不是返回一个被修改后的列表。

这与字符串的工作方式不同。这也是为什么说类标是可变类型而字符串是不可变类型的原因。

然后当在市场购买一样东西后，我们希望将其从列表中删除，*del* 语句正是用武之地。

在这里我们指出希望删除列表中的哪个元素，del 就将这个元素从列表中删除。

我们指定的是希望删除列表的第一个元素，因此我们使用 *del shoplist[0]*(回想一下，python 的索引从 *0* 开始)。

如果你想知道 list 对象的所有方法，详见 *help(list)*。

元组

元组用于保存各种各样的对象。它与列表很相似，但它缺少列表提供的大量功能。列表的一个主要特点就象字符串一样，它是不可变类型，也就是说你不可以修改元组。

元组通过一组以逗号分隔的元素定义，并以一个可选的小括号闭合。

元组通常用于这样的情形，一个语句或一个用户定义的函数能够安全的假设其使用的一组值(即元组值)不会发生改变。

范例：

```
#!/usr/bin/python
# Filename: using_tuple.py
zoo = ('python', 'elephant', 'penguin') # 注意小括号是可选的
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
```

代码如何工作：

变量 *zoo* 引用一个元组。我们看到 *len* 函数可以得到元组的长度。这也表明元组同样是一个序列类型。

因为老动物园歇菜了，于是我们将这些动物转移到一个新的动物园。因此元组 *new_zoo* 既包含已有的动物又包含从老动物园转移过来的新动物。

言归正传，注意一个包含在其它元组内的元组并不会丢失它的身份。

(注：包含元组会引用被包含元组，即在包含元组内对被包含元组的操作会反应到被包含元组自身之上，有点绕口。。。)

像列表一样，我们可以通过一对方括号指定元素的位置访问这个元素。这叫做索引操作符。

我们通过 *new_zoo[2]*访问 *new_zoo* 的第三个元素，通过 *new_zoo[2][2]*访问 *new_zoo* 的第三个元素的第三个元素。

一但你理解这种语言风格，这样的操作太安逸了。

小括号

虽然小括号是可选的，但我强烈建议你坚持使用小括号，这样一眼就能看出它是个元组，尤其还能避免出现歧义。
例如，*print(1, 2, 3)*和 *print((1, 2, 3))*是不同的 – 前者打印 *3* 个数字而后者打印一个元组(包含 *3* 个数字)。

拥有 0 个或 1 个元素的元组
一个空元组通过空小括号创建，例如 *myempty = ()*。
不过，指定一个单元素元组就不那么直观了。你必须在其仅有的一个元素后跟随一个逗号，这样 python 才能区分出
你要的是元组而不是一个被小括号包围的对象的表达式。例如你想要一个包含值为 *2* 的单元素元组，则必须写成 *singleton = (2,)*

perl 程序员请注意（注：对不起 perl 程序员，我是 perl 盲。。。不知道说的啥，以后可能补充翻译）
A list within a list does not lose its identity i.e. lists are not flattened as in Perl. The same applies to a tuple within a tuple, or a tuple within a list, or a list within a tuple, etc. As far as Python is concerned, they are just objects stored using another object, that's all.

字典

字典就像通讯录，只要知道联系人的名字就能找到他的地址或详细信息。即我们将*键*(名字)与*值*(相关信息)联系到一起。
注意键必须是唯一的，这就像如果两个人同名你就没法找到正确的信息了。
还有字典的键必须是不可变对象（比如字符串），但字典的值可以是可变或不可变对象。基本上这意味着只能将简单的对象作为键。
字典中的键值对使用语法 *d = {key1 :value1, key2: value2}*指定。
其中键和值由分号分隔而所有的键值对用逗号分隔，并且它们被括在一对大括号内。
记住字典中的键值对是无序的。如果你希望按照特定的顺序排列它们，你只能在使用前自己排序。
而你实际使用的字典是 dict 类的对象/实例。
范例：

```
#!/usr/bin/python
# Filename: using_dict.py
# 'ab'是'a'ddress'b'ook 的缩写
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'])
# 删除一个键值对
del ab['Spammer']
```

```
print('\nThere are {0} contacts in the address-book\n'.format(len(ab)))
for name, address in ab.items():
    print('Contact {0} at {1}'.format(name, address))
# 添加一个键值对
ab['Guido'] = 'guido@python.org'
if 'Guido' in ab: # OR ab.has_key('Guido')
    print("\nGuido's address is", ab['Guido'])
```

Output:

```
$ python using_dict.py
Swaroop's address is swaroop@swaroopch.com

There are 3 contacts in the address-book

Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org

Guido's address is guido@python.org
```

代码如何工作：

我们使用先前介绍的语法创建字典 *ab*。然后使用在列表和元组部分讨论过的索引操作符指定字典键访问键值对。多简单的语法阿。

我们的老朋友 *del* 语句可以帮助我们删除键值对。只需简单的为索引操作符指定被删除的键，再将其传给 *del* 语句就哦了。

执行删除操作时我们无需理会键所对应的值。

接下来我们使用字典的 *items* 方法访问字典的键值对，它会返回一个包含键值对元组的列表 – 值跟在键后面。

在 *for...in* 循环中我们检索每个键值对并将它们分别赋给变量 *name* 和 *address*，之后在循环体中打印它们。

利用索引操作符访问一个键并对其赋予一个值我们可以增加一个新的键值对，就象本例中的 *Guido* 那样。

通过 *dict* 类的 *has_key* 可以检查字典中是否存在某个键值对。你可以执行 *help(dict)* 找到字典所有方法的列表。

关键字实参与字典

如果你已经在函数中使用过关键字实参，那么你也已经使用过字典了！

你可以这样理解 – 你在函数定义时的形参列表中指定了键值对，当你在函数中访问这些变量的时候只不过是在访问一个字典

(在编译器设计的术语中这被称作符号表)

序列

列表，元组和字符串都是序列的例子，但到底序列是啥呢？为什么它对我们的意义如此特别？

序列最主要的特点在于支持成员从属测试（即，表达式中的 *in* 和 *not in* 操作）和索引操作。
其中索引操作允许我们直接地获取序列中的指定元素。
以上说到的三种序列类型 – *lists*，*tuples*，*strings* 还支持一种 *切片*操作，允许我们得到序列的一个切片，即序列的部分。
范例：

```
#!/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[:])
```

Output:

```
$ 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

代码如何工作：

首先我们看看如何使用索引得到序列的单个元素。这也被称作 *下标*操作。

正如上面的代码，每当你在序列旁的方括号中指定一个数字的时候，python 会获取这个索引所对应的序列元素。

回想一下，python 的索引从 *0* 开始计算。因此 *shoplist[0]*获取序列 *shplist* 的第一个元素，而 *shoplist[3]*获取第四个元素。

索引也可以是负数，这时候位置将从序列尾开始计算。所以，*shoplist[-1]*引用序列的最后一个元素，*shoplist[-2]*为倒数第二个。

切片操作的使用方法是先指定序列名后跟一对方括号，其中包含一对可选的由分号分隔的数字。

注意这与你至今使用的索引操作非常相似。记住数字是可选的，但分号不可以省略。

切片操作中的第一个数字（分号前）指出切片的开始位置而第二个数字（分号后）指定将在哪个位置结束。

如果省略第一个数字则 python 将以序列的起点为开始处，而省略第二个数字时切片会停止在序列的结尾处。

注意切片将在开始处开始，结束于结尾处之前，即包括开始处但不包括结尾处。（注：比如 *a[1:10]*，返回的是 a[1]到 a[9]不包括 *a[10]*）。

因此，*shoplist[1:3]*开始于索引 *1*，包括索引 *2* 但止于索引 *3*，即返回一个包含两个元素的切片。与之类似 *shoplist[:]*将返回整个序列的拷贝。

你还能以负索引切片。负数代表从序列的末尾开始反向计算位置。例如 *shooplist[:-1]*返回整个序列，但不包括末末的元素。

另外你还可以为切片提供第三个实参，它代表步长（默认为 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']
```

注意当步长为 *2* 时，我们得到索引为 *0*，*2*...的元素，步长为 *3* 时得到 *0*，*3*...，以此类推。

用 python 交互解释器(这样你能立即看到结果)尝试切片的各种用法吧。

序列类型最棒的地方在于你能够以相同的方式访问元组，列表，字符串！

集合

集合是简单对象的无序集合，适合当更关心集合中的元素是否存在而不是它们的顺序或是它们出现的次数的时候。

使用集合，你可以测试从属关系，是否一个集合是另一个集合的子集，或是寻找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'}
```

代码如何工作：

代码几乎是自说明的，因为它涉及到的基础集合论知识我们已经在学校学过了。

引用

当你创建一个对象并将其赋给一个变量的时候，变量只是引用了这个对象，而变量并不代表这个对象本身！

换言之，变量名指向你的计算机内存的一部分，而这部分内存用于存储实际的对象。这叫做名字到对象的绑定。

通常你不用关心这些，但你应该知道由于引用造成的一些微妙的影响。

范例：

```
#!/usr/bin/python
# Filename: reference.py
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist 只是指向相同对象的另一个名字
del shoplist[0] # 我购买了第一个水果，所以把它从清单中删除
print('shoplist is', shoplist)
print('mylist is', mylist)
# 注意列表 shoplist 和 mylist 打印了相同的内容，其中都不包括’apple’，因为它们指向的是相同的对象。
print('Copy by making a full slice')
mylist = shoplist[:] # 以全切片创造一个列表的完整拷贝
del mylist[0] # 删除第一个元素
print('shoplist is', shoplist)
print('mylist is', mylist)
# 注意现在两个列表指向不同的对象（注：回忆一下，切片操作会返回一个新的对象！）
```

Output:

```
$ 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']
```

代码如何工作：

大多数的解释已经包含在注释中了。

记住，如果你想创建一个诸如列表这样的序列或复杂对象(不是象整数那样的简单对象)的拷贝，必须使用切片操作。

如果你只是简单的用变量名指向另一个变量名，两者实际上将引用相同的对象，如果你不注意这点将会招来麻烦！

Perl 程序员请注意

记住对于列表的赋值语句并不会创建一个拷贝。必须使用分片操作创建序列的拷贝。

(注：实际上切片操作不是唯一的选择，内见的工厂函数比如 list，typle, set 等都能达到同样的目的)

关于字符串的更多知识

前面我们已经详细讨论过字符串了。在这里我们还会了解到什么呢？

呵呵，你知道字符串同样是一种对象并拥有很多方法吗? 从检查字符串的一部分到删除其中的空格应有尽有！

你在程序中使用的所有字符串都是 *str* 类的对象。下面的例子会演示 str 类中的一些有用的方法。全部方法的列表，参见 *help(str)*。

范例：

```
#!/usr/bin/python
# Filename: str_methods.py
name = 'Swaroop' # 这是一个字符串对象
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))
```

Output:

```
$ 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
```

代码如何工作：

在这里我们看到了许多字符串方法的用法。
startswith 方法用于确定字符串是否以指定的字符串开头。而 *in* 操作检查一个字符串是否是另一个字符串的一部分。
find 方法用来寻找给定的字符串在字符串中的位置，如果没找到对应的子串则返回*-1*。
str 类还有一个简洁的连接序列中每个字符串并返回连接后的字符串的方法 *join*，其中每个字符串都将以指定的字符串分隔。

小结

我们已经详细研究了 python 中的各种内建数据结构。这些数据结构对于编写合理规模大小的程序是必须可少的。
现在我拥有了许多的 python 的基础知识，下一步我们会看到如何设计和编写一个实用的 python 程序。

我们已经探究了 python 语言的方方面面，现在我们将通过设计编写一个有用的程序将这些内容有机的结合起来。
主要目标是让大家有能力独自编写程序。

问题

我们要解决的问题是”希望编写一个程序，用于创建所有重要文件的备份”。
尽管这个问题很简单，但并没有给出足够多的直观信息用以创建解决方案。所以进行少量的分析还是必须的。
例如，如何指定哪些文件需要备份？如何存储？存在哪？
适当的分析过问题后，我们开始设计程序。我们创建一个用于指明程序应该如何工作的列表。
在本例中，我已经创建了一个我希望程序如何工作的列表。
如果换作你来设计，你可能不会和我一样分析问题，毕竟每个人都有自己解决问题的思路，这很正常.
1．需要备份的文件和目录由一个列表指定。
2．备份必须存在一个主备份目录中。
3．文件会被备份为一个 zip 文件。
4．这个 zip 文件以当前的日期和时间命名。
5．我们使用任何标准 linux/unix 发行版中默认的标准 *zip* 命令创建 zip 文件。
Windows 用户可以从 GnuWin32 工程页下载安装之，并将 C:\Program Files\GnuWin32\bin 添加到你的系统环境变量 PATH 中。
GnuWin32 工程页: http://gnuwin32.sourceforge.net/packages/zip.htm
zip 命令下载: http://gnuwin32.sourceforge.net/downlinks/zip.php
注意你可以使用任何希望的存档命令，只要它拥有一个命令行接口。因此我们可以通过脚本为它传送参数。

解决方案

现在我们的程序已经设计妥当，下一步可以着手编写实现我们的解决方案的代码了。

```
#!/usr/bin/python
# Filename: backup_ver1.py
import os
import time
# 1. 需要备份的文件和目录由一个列表指定
source = ['"C:\\My Documents"', 'C:\\Code']
# 注意我们必须在字符串内部使用双引号将带有空格的名字括起来。
# 2. 备份必须存在一个主备份目录中
target_dir = 'E:\\Backup' # 记住改变这里即可改变你想要使用的主目录

# 3. 文件会被备份为一个 zip 文件。
```

```
# 4. 这个 zip 文件以当前的日期和时间命名。
target = target_dir + os.sep + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. 我们使用 zip 命令将文件归档成一个 zip
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))
# 执行备份
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，我们正在测试阶段测试我们的程序是否正确工作，如果与预期不符，那么我不得不对程序 debug，即消除程序中的 bug(错误)。
如果上面的程序在你那无法工作，那么在调用 *os.system* 前添加一句 *print(zip_command)*并运行程序。
现在将打印出的 *zip_command* 的值拷贝/复制到命令行看看它是否能正确运行。
如果这个命令还是运行失败，查阅 zip 命令手册吧确定是哪里出了问题。
相反如果命令运行成功，请仔细核对你输入的代码与上面的程序是否相同。

代码如何工作：

接下来你将注意到我们是如何将我们的设计一步一步转换为代码的。
我们将使用 *os* 和 *time* 模块，所以最先导入了它们。然后我们在 *source* 列表中指定需要备份的文件和目录。
目标目录用来存储所有的备份文件，其由 *target_dir* 变量指定。
我们将要创建的 zip 归档文件的文件名为当前的日期和时间，这通过 *time.strftime()* 函数得到。
归档文件还拥有*.zip* 后缀名并被存储到 target_dir 指定的目录中。
注意 *os.sep* 变量的使用 - 这给予目录分隔符以系统无关性，即它在 linux, unix 为'/'，在 windows 为'\\'而在 Mac OS 是':'。
使用 os.sep 代替直接使用这些字符将使我们的程序可移植的跨多系统工作。
time.strftime()函数需要一个类似上面程序中使用的说明符。
说明符*%Y* 会被带有世纪部分的年份替换(注: 2010 即带有世纪部分，而 10 则不带有，此处原文是"%Y 不带有世纪部分"这是错误的，%y 才不带有)。
%m 会被以为十进制数 01 到 12 表示的月份替换，后面的说明符以此类推。
完整的说明符列表可以在 python 参考手册中找到。
(http://docs.python.org/dev/3.0/library/time. html#time. strftime)。
(注：这个地址好像挂了，不然得翻墙？查本地手册是一样一样地)
我们使用加法运算符连接字符串以创建目标 zip 文件名，这里的加法运算符用于将两个字符串连接起来并返回这个连接后的字符串。
然后我们创建字符串 *zip_command*，其中包括我们将要执行的命令。你可以在 shell(linux 终端或 dos 命令行)中运行这个命令检查它是否正常工作。
我们使用的 *zip* 命令带有一些选项和参数。*-q* 选项指示 zip 将以安静模式工作。*-r* 指定命令将递归的压缩目录，即包括指定目录的所有子目录和文件。

两个选项可以简写组合到一起*-qr*。选项后面紧跟被创建的 zip 归档文件的文件名和需要备份的文件与目录。
我们通过字符串的 *join* 方法将列表 source 合并成一个字符串，join 的使用方法我们已经讲过了。
最后我们使用 *os.system* 函数运行命令，这就好像从系统(shell)运行命令一样 – 如果命令成功则返回 0，否则返回错误号。
根据命令运行的结果，我们打印出适当的信息提示备份是否成功。
至此，我们终于完成了这个备份重要文件的脚本！

写给 windows 用户
作为反斜杠转义序列的替代，你可以使用原始字符串。例如*'C:\\Documents'*或者*r'C:\Documents'*。
但是不要使用*'C:\Documents'*因为这样做实际上是在使用一个未定义的转义序列*\D*。

既然我们已经拥有一个可用的备份脚本，那么当我们需要备份文件的时候都可以使用它。
建议 linux/unix 用户使用前面介绍过的执行方法，这样就可以在任何时间任何地点运行备份脚本了。这被称作软件的操作阶段或部署阶段。
上面程序可以正确运行，但通常第一版程序不会完全达到你的预期。
例如如果你没有正确的设计程序或是输入了错误代码等等都会造成问题。这时你将不得不返回设计阶段或是为程序 debug。

第二版程序

我们的第一版程序可以工作。不过仍有一些改良空间使得它在日常使用中更好的工作。这称作软件的维护阶段。
其中我认为比较有用的一个改良是提供更好的文件命名机制 – 在主备份目录中，以时间作为文件名而以日期作为目录名。
这样做的优点之一是你的备份被分级存放，因此变得更容易管理。第二文件名变的更短。
第三个优点是分离各个目录将帮助你轻松的检查当天是否创建了备份，因为只有当天创建了备份相应的目录才会被创建。

```
#!/usr/bin/python
# Filename: backup_ver2.py
import os
import time
# 1. 需要备份的文件和目录由一个列表指定
source = ['"C:\\My Documents"', 'C:\\Code']
# 注意我们必须在字符串内部使用双引号将带有空格的名字括起来。
# 2. 备份必须存在一个主备份目录中
target_dir = 'E:\\Backup' # 记住改变这里即可改变你想要使用的主目录
# 3. 文件会被备份为一个 zip 文件。
# 4. 主目录中的子目录将以当前日期命名
```

```
today = target_dir + os.sep + time.strftime('%Y%m%d')
# zip 归档文件将以当前时间命名
now = time.strftime('%H%M%S')
# 如果子目录不存在则创建之
if not os.path.exists(today):
    os.mkdir(today) # make directory
    print('Successfully created directory', today)
# 组成 zip 文件名
target = today + os.sep + now + '.zip'
# 5. 我们使用 zip 命令将文件归档成一个 zip
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))
# 执行备份
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
```

代码如何工作：

程序的大部分与第一版相同。主要改变是我们使用 *os.path.exists* 函数检查主备份目录中否存在一个与当前日期同名的子目录。

如果不存在则利用 *os.mkdir* 函数创建之。

第三版程序

我用第二版程序做了许多备份，它工作的很好，但当备份太多时我发现很难彼此区分它们！

例如，我可能对一个程序或演示稿做了某些重要修改，并希望将这些改变关联到 zip 归档文件名上。

这可以通过为 zip 归档文件名附加一个用户注释轻松做到。

注意

下面的程序并不能工作，所以不要疑惑请阅读下去，在此它只是一个演示。

```
#!/usr/bin/python
# Filename: backup_ver3.py
import os
import time
# 1. 需要备份的文件和目录由一个列表指定
source = ['"C:\\My Documents"', 'C:\\Code']
```

```
# 注意我们必须在字符串内部使用双引号将带有空格的名字括起来
# 2. 备份必须存在一个主备份目录中
target_dir = 'E:\\Backup' # 记住改变这里即可改变你想要使用的主目录
# 3. 文件会被备份为一个 zip 文件
# 4. 主目录中的子目录将以当前日期命名
today = target_dir + os.sep + time.strftime('%Y%m%d')
# zip 归档文件将以当前时间命名
now = time.strftime('%H%M%S')
# 让用户输入一个注释以便创建 zip 文件名
comment = input('Enter a comment --> ')
if len(comment) == 0: # 检查是否输入注释
   target = today + os.sep + now + '.zip'
else:
   target = today + os.sep + now + '_' +
      comment.replace(' ', '_') + '.zip'
# 如果子目录不存在则创建之
if not os.path.exists(today):
   os.mkdir(today) # 创建目录
   print('Successfully created directory', today)
# 5. 我们使用 zip 命令将文件归档成一个 zip
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))
# 运行脚本
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
```

代码如何运行：

记住这个程序无法工作！python 说它遇到一个语法错误，这表示脚本不符合 python 预期的语法结构。

当我们留意错误信息时还发现 python 同时告诉我们错误出在哪里。所以我们从这个错误行开始 debug。

仔细观察后我们注意到一个逻辑行被分割成两个物理行，但我们并指出这两个物理行是连一起的。

基本上，python 发现了加法运算符(+)但没有在这个逻辑行上找到需要的运算数，因此 python 不知道该如何继续下去。

记住在物理行末尾使用反斜杠我们可以指定逻辑行将延续到下个物理行。

因此我们修正这个错误，这叫做错误修正(bug fixing)。

第四版程序

```
#!/usr/bin/python
# Filename: backup_ver4.py
import os
import time
# 1. 需要备份的文件和目录由一个列表指定
source = ['"C:\\My Documents"', 'C:\\Code']
# 注意我们必须在字符串内部使用双引号将带有空格的名字括起来
# 2. 备份必须存在一个主备份目录中
target_dir = 'E:\\Backup' # 记住改变这里即可改变你想要使用的主目录
# 3. 文件会被备份为一个 zip 文件.
# 4. 主目录中的子目录将以当前日期命名
today = target_dir + os.sep + time.strftime('%Y%m%d')
# zip 归档文件将以当前时间命名
now = time.strftime('%H%M%S')
# 让用户输入一个注释以便创建 zip 文件名
comment = input('Enter a comment --> ')
if len(comment) == 0: # 检查是否输入注释
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' + \
        comment.replace(' ', '_') + '.zip'
# 如果子目录不存在则创建之
if not os.path.exists(today):
    os.mkdir(today) # 创建目录
    print('Successfully created directory', today)
# 5. 我们使用 zip 命令将文件归档成一个 zip
zip_command = "zip -qr {0} {1}".format(target, ' '.join(source))
# 执行脚本
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
```

代码如何工作：

程序现在可以工作了！让我们看看在第三版中所做的实质性增强吧。
我们使用 *input* 函数得到用户注释，然后检查用户是否真的输入了什么，这可以通过 *len* 函数判断输入长度做到。
如果用户什么都没输入(也许用户只是需要一个常规备份或没有对文件进行什么特别的修改)，则程序就像老版本那样工作。
相反，如果用户提供了注释，则注释会被附加到 zip 归档文件的扩展名.zip 之前。注意我使用下划线替换注释中的空格 – 这样文件名管理起来更简单。

更多改进

第四版程序对于大多数用户已经是个令人满意的脚本了，但程序永远存在可以改进的空间。
例如，你可以为程序增加一个冗言层(verbosity level)并以*-v* 选项启动，使得你的程序更多嘴(注：也就是让程序的交互性更强)。
另一个可能的改进是在命令行直接将文件和目录传给脚本。这些文件和目录名可以通过 *sys.argv* 列表得到并使用 list 类的 extend 方法将它们添加到 source 列表。
而最重要的一个改进可能就是用内建模块 *zipfile* 或 *tarfile* 代替 *os.system* 创建归档文件了。它们是标准库的一部分而且无需依赖你的计算机安装外部 zip 程序。
无论如何，在上面的示例中出于教学目的我完全使用 os.system 创建备份，这很简单明了足以让任何人理解但并不是令人满意的实现方式。
那么你能不能使用 zipfile 模块代替 os.system 实现程序的第五版呢？(注：在官方文档里查 zipfile 的使用方法)

软件开发过程

现在我们已经走过了编写软件的各个阶段。这些阶段总结如下：
1. 需要什么功能(分析)
2. 如何实现它们(设计)
3. 着手实现它们(实现)
4. 测试这些功能(测试和 debug)
5. 使用(实施或部署)
6. 维护程序(改良)

我们创建这个备份脚本的步骤就是编写一个程序时被推荐的方式 - 首先分析和设计。然后实现一个简单版本。
测试并 debug 确定它能够如期工作。最后增加你需要的新特性并重复*编写-测试-使用*的过程直到你满意为止。
记住，软件是扩展起来的，而不是建造起来的。

小结

我们已经看到如何编写自己的 python 程序/脚本，并见到编写一个类似程序所牵涉到的各个步骤。

你会发现这些知识对于创建自己的程序会非常有用，因此你不仅掌握了 python 也掌握了解决问题的办法。
接下来，我们将讨论面向对象编程。

简介

(注: OOP 代表面向对象编程，OO 代表面向对象，以后全部使用英文缩写)
迄今为止我们编写的所有程序都是围绕函数创建的，函数即操纵数据的语句块。这称作面向过程编程。
除此之外还有另一种组织程序的方法，将数据与功能组合到一起封装进被称为对象的东西中。这叫做 OOP。
大多数时候你可以使用过程性编程，但当编写大型程序或问题更倾向以 OO 方式解决时，你还可以使用 OOP 技术。
类和对象是 OOP 的两个重要特征。类用于创建新的类型，而对象是类的实例。这就象你创建 int 类型的变量，这些变量就是 int 类的实例(对象)

静态语言程序员请注意
注意在 python 中就算整数也被当作对象(int 类的对象)。
这与 C++和 Java(版本 1.5 之前)是不同的，它们中的整数属于原始本地类型。具体详见 *help(int)*。
而 C#和 java 1.5 程序员会发现这与装箱(boxing)和拆箱(unboxing)的概念类似。

对象可以使用属于这个对象的变量存储数据。属于对象或类的变量称作字段(fields)。
对象同样可以利用属于类的函数实现所需的功能。这些函数被称作类的方法(method)。
这些术语非常重要，因为它们帮助我们将独立的函数和变量与属于类或对象的函数和变量区分开来。
字段与方法统称为类属性。
字段分为两种类型 – 它们既可以属于类的实例/对象也可以属于类本身，两者分别称为实例变量与类变量。
一个类通过关键字 *class* 创建。字段与方法被列在类的缩进块里。

self

类方法与普通函数只有一个特殊区别 – 类方法必须增加一个额外的形参，而且它必须处于第一个形参的位置，
但是在调用类方法时不要为这个额外的形参传值，python 会自动代劳。这个特别的变量引用对象本身，按照惯例它被命名为 *self*。
尽管你可以随便为这个形参取名字，但我强烈建议你使用 *self* – 其它名字会让人皱眉头的。
使用标准名字是有很多好处的 – 任何你的代码的读者都会立即明白它代表什么，甚至当你使用标准名字时专业的 IDE 都会更好的帮助你。

写给 C++/Java/C#程序员
python 中的 self 相当于 C++中的 this 指针和 java/C#中的 this 引用

你一定感到疑惑 python 是如何为 self 赋值的，为什么你无需亲力亲为呢？举个例子可以让事情变的明朗。
假设你有一个叫做 *MyClass* 的类与一个它的实例 *myobject*。 当你调用这个对象的方法时 *myobject.method(arg1, arg2)*，
python 会自动将其转换为 *myobject.method(myobject, arg1, arg2)* – 这就是关于 self 的内幕。
这也意味着如果你有一个无需参数的方法，你也仍然需要一个 *self* 实参。

类

下面的例子可以是一个最简单的类了。

```
#!/usr/bin/python
# Filename: simplestclass.py
class Person:
   pass # 空语句块
p = Person()
print(p)
```

输出：

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

代码如何工作：
我们使用 *class* 语句和一个类名创建了一个新类，其后紧跟一个定义类体的缩进的语句块。本例中 *pass* 语句代表一个空语句块。
接下来，我们创建这个类的对象/实例，方法是类名后跟一对小括号(后节我们会学到更多关于实例的知识)。
为了验证变量类型，我们简单的打印它。它告诉我们在__main__模块我们拥有了一个 *Person* 类的实例。
注意存储这个对象的计算机内存地址也被打印出来了。而地址值在你的计算机中可能会不同，因为 python 只要找到可用空间就会把对象存进去。

对象方法

我们已经讨论过对象/类可以拥有类似函数一样的方法，只不过这些函数必须加上额外的 *self* 变量。现在我们就来看个例子。

```
#!/usr/bin/python
# Filename: method.py
class Person:
   def sayHi(self):
      print('Hello, how are you?')
p = Person()
p.sayHi()
# 在这个简短的例子中同样可以写成 Person().sayHi()
```

输出：

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

代码如何工作：

本利中我们使用了 *self*，注意方法 *sayHi* 虽无需参数但在函数中仍然要写上 *self*。

__init__方法

在 python 的类中，有很多方法名拥有特殊意义。我们现在看看*__init__*方法的特别之处。

*__init__*方法在类对象被实例化时立即执行。此方法专用于初始化对象。另外注意方法名中的双下划线前后缀。

范例：

```
#!/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()
# 在这个简短的例子中同样可以写成 Person().sayHi()
```

输出：

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

范例如何工作：

这里我们定义一个带有名为 *name* 形参的*__init__*方法(除了寻常的 self)。

然后我们又创建了一个名为 *name* 的字段。注意它们是两个不同的变量尽管都叫’name’。点号允许我们将它们区分开来。

重点是，我们并没有显式的调用*__int__*方法，而是在创建类实例时在类名后面的小括号中指定实参。这就是*__init__*的特殊之处。

初始化之后，我们就可以在方法中使用 *self.name* 字段了，方法 *sayHi* 演示了这点。

类和对象变量

我们已经讨论了类与对象的功能(方法)部分，现在就来学习数据部分。

数据即字段，只不过是绑定在类和对象的名字空间中的普通变量。这意味着这些名字只在其所属类和对象的上下文中合法有效。这就是它们被称作名字空间的原因。

有两种字段类型 – 类变量和对象变量，它们根据所有者是类还是对象而区分开来。

类对象是共享的 – 它们可以被一个类的所有实例存取。即一个类对象在类中只存在一个拷贝，任何对象对其的修改都会反映到所有对象上。

而每个独立的类对象/实例都拥有自己的对象变量。这样每个对象都有属于自己的字段拷贝即这些字段非共享，不同对象中的同名对象变量彼此没有任何关联。举个例子将更容易理解：

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

class Robot:
   '''Represents a robot, with a name.'''
   # 一个类变量用于记录机器人的数量
   population = 0

   def __init__(self, name):
      '''Initializes the data.'''
      self.name = name
      print('(Initializing {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 masters 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-3PO')
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：

```
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
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-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

代码如何工作：

这个例子很长但有助于演示类和对象变量的本质。

population 属于 *Robot* 类因此它是个类变量。变量 *name* 属于对象(使用 self 赋值)因此它是个对象变量。

于是乎，我们使用 *Robot.population* 引用类变量 *populatin* 而不是 *self.population.*。而在方法中引用对象变量 *name* 时使用 *self.name* 语法。

记住这个类变量和类对象中简单的差异吧。还要注意对象变量会隐藏同名的类变量！

howMany 实际上是一个属于类的方法而不是对象。这意味着我们可以将及其定义为 *classmethod* 也可以定义为 *staticmethod*，

这取决于我们是否需要知道我们是哪个类的一部分。因为我们无需类似信息，所以我们使用 staticmethod。

另外我们还可以通过装饰符(decorators)达到同样的效果：

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

装饰符可以被想象成调用一条显式语句的捷径，就像在这个例中看到的一样。

观察__init__方法，它用于以一个指定的名字初始化 Robot 实例。其内部对 population 累加 1，因为我们又添加了一个机器人。
同时观察 self.name，它的值特定于每个对象，这也指出了类对象的本质。
记住，你只能使用 self 引用相同对象的变量和方法。这被称作属性引用。
本例中，我们还在类与方法中使用了文档字符串。我们可以在运行时使用 *Robot.__doc__*和 *Robot.sayhi.__doc__*分别访问类和方法的文档字符串。
就像*__init__*方法，这里还有另一个特殊方法*__del__*，当对象挂掉的时候将被调用。
对象挂掉是指对象不再被使用了，它占用的空间将返回给系统以便重复使用。
在__del__中我们只是简单的将 Robot.population 减 1。
当对象不再被使用时__del__方法将可以被执行，但无法保证到底啥时执行它。
如果你想显式执行它则必须使用 *del* 语句，就象本例中做的那样。（注：本例中 del 后对象的引用计数降为 0）。

C++/Java/C#程序员请注意
python 中所有类成员（包括数据成员）全部为 public，并且所有方法都为 virtual。
只有一个例外：如果你使用的数据成员，其名字带有双下划线前缀例如 *__privatevar*，则 python 将使用名字混淆机制有效的将其作为 private 变量。
另外存在一个惯例，任何只在类或对象内部使用的变量应该以单下划线为前缀，其他的变量则为 public 可以被其它类/变量使用。
记住这只是一个惯例而不是强迫(不过双下划线前缀例外).

继承

OOP 的主要好处就是代码重用，而达此目的的方法之一是利用继承机制。最好将继承想象为实现类与类之间的类型与子类型关系。
假设你想要编写一个程序用来追踪大学里的师生情况。师生有很多共同的特征比如名字，年龄和地址等。
他们还拥有一些不同的特征例如教师的薪水课程假期，学生的成绩学费等。
你可以分别为师生创建两个独立的类并分别处理之，但增加一个共同的特征意味两个类都要增加。很快这种设计方式就会变的笨重臃肿。
更好的办法是创建一个称为 *SchoolMember* 的通用类，然后让教师和学生类分别继承它即成为 *SchoolMember* 的子类型。之后再增加各自的专有特征。
这样做有很多优点，我们为 SchoolMember 增加或修改任何功能都会自动反映到它的子类型中。
例如，你可以为教师和学生类增加一个 ID 卡字段，这只需简单的将其加入 SchoolMember 即可。不过子类型中的改变不会影响到其他子类型
另一个好处是如果你能够以 SchoolMember 对象引用教师或学生对象，这对于统计学校人数之类的情形很有用。
这被称作多态，在任何需要父类型的地方其子类型都可以被替换成父类型，即对象可以被当做父类的实例。
另外我们可以复用父类代码而无需在不同的类中重复这些代码，但使用我们先前讨论的独立类时就不得不重复了。

本例中的 SchoolMember 类被称为基类或超类。*Teacher* 和 *Student* 类叫做派生类或子类。
现在我们就来看看这个范例吧。

```
#!/usr/bin/python
# Filename: inherit.py
class SchoolMember:
   '''Represents any school member.'''
   def __init__(self, name, age):
      self.name = name
      self.age = age
      print('(Initialized 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):
   '''Represents 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', 40, 30000)
s = Student('Swaroop', 25, 75)
print() # 打印一个空行
members = [t, s]
for member in members:
   member.tell() # 在 Teacher 和 Student 上都能工作
```

输出：

```
   $ python inherit.py
   (Initialized SchoolMember: Mrs. Shrividya)
```

```
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```

范例如何工作：

为了使用继承，我们在类定义的类名后的一个元组中指定基类名。

然后我们注意到使用 *self* 变量作为参数显式的调用了基类的*__init__*方法，这样我们就能初始化对象的基类部分了。

牢记这点非常重要 – python 不会自动调用基类的构造函数，你必须自己显式的调用它。

我们同样注意到可以通过增加类名前缀调用基类方法，然后为其传送 self 变量和其他任何实参。

注意当我们使用 SchoolMember 类的 *tell* 方法时，可以将 *Teacher* 或 *Student* 的实例当做 *SchoolMember* 的实例。

同时应该注意到程序调用的是子类型的 *tell* 方法二不是 *SchoolMember* 类的 *tell* 方法。

理解这个机制的一个思路是 python 永远先在实际类型中查找被调用的方法，这个例子中就是如此。

如果 python 没有找到被调用方法，则会在基类中逐个查找，查找顺序由类定义时在元组中指定的基类顺序决定。

另外解释一个术语 – 如果一个类继承了多个基类，则被称做多重继承。

小结

我们已经研究了类和对象的方方面面，还包括与其相关的各种术语。同样也看到了 OOP 的优点与陷阱。

python 是一个高度 OO 语言，认真理解这些概念将对你大有裨益。

接下来，我们将学习如何在 python 中处理输入/输出与文件存取。

简介

一些情况下你不得不让程序与用户进行交互。例如，你需要从用户处得到输入然后输出计算结果。我们可以分别通过 *input()*和 *print()*函数做到这些。
对于输出，我们还可以使用 *str*(string)类的各种方法。例如 *rjust* 方法可以得到一个指定宽度的右对齐字符串。详见 *help(str)*。
另一种常见的输入/输出类型为文件处理。对于很多程序拥有创建，读写文件的能力是必不可少的，我们会在这节探究这些内容。

得到用户输入

```
#!/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
```

范例如何工作：
范例中我们使用切片操作反转文本。之前我们已经学过如何通过 *seq[a:b]*(从 a 开始止于 b)对序列切片。
对于切片操作我们还可以指定第三个参数步长，步长默认为 *1* 将返回文本的一个连续部分。而给定一个负步长-1 将返回反转后的文本。
*input()*函数接收一个字符串实参并将其打印给用户，然后函数等待用户输入一些东西，一但用户按下回车键则输入结束，input 函数将返回输入的文本。

之后我们反转文本，如果反转后的文本与原文本相同，则代表它是一个回文(http://en.wiktionary.org/wiki/palindrome)。

练习题：
检测一个文本是否为回文应该忽略标点，空格和大小写。
例如*"Rise to vote, sit."*同样是一个回文，但是我们当前的例子无法识别它。你能改善这个例子让它做都这点吗？

文件

通过创建 *file* 类对象，使用其 *read, readline* 或 *write* 方法你可以对文件进行读写。具体读或写的方式依赖于你打开文件时指定的模式。
最后当你完成文件操作时调用 *close* 关闭文件。
范例：

```
#!/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') # 写模式打开
f.write(poem) # 写文件
f.close() # 关闭文件

f = open('poem.txt') # 如果没有提供打开模式, 则默认假设为读模式
while True:
   line = f.readline()
   if len(line) == 0: # 长度为 0 代表 EOF(注: end of file 即文件尾)
      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!
```

范例如何工作：
首先，我们通过内建函数 open 打开一个文件，在函数中我们指定了被打开文件的文件名与希望使用的打开模式。

其中打开模式可以为读模式(‘r’)，写模式(‘w’)或追加模式(‘a’)。另外我们也可以处理文件文件(‘t’)和二进制文件(‘b’)。
实际上还有很多模式可用，详见 *help(open)*。默认的 *open* 将文件对待为文本文件’t’，并以读模式’r’打开。
在范例中，我们首先以写文本模式打开文件，使用文件对象的 *write* 方法写文件，并调用 *close* 将其关闭。
然后我们再次打开相同的文件用于读取。这里我们无需指定打开模式因为’读文本文件’是 open 的默认模式。
在循环中我们使用 *readline* 方法读取文件的每一行。这个方法返回一整行文本其中包括末尾的换行符。
当返回一个空字符串时，意味着我们已经来到文件尾，因此使用 *break* 跳出循环。
默认的，print()函数将自动打印一个换行。因为从文件读出的文本行末尾已经包含一个换行，所以我们指定参数 *end=’’*抑制换行。
最后我们关闭文件。
现在，检查 *poem.txt* 文件内容以确定程序真的写入并读取了文件。

Pickle

python 提供了一个名为 *pickle* 的标准模块用于将任意 python 对象存入文件或从文件中读出。这被称做永久性存储对象(persistently)。
范例：

```
#!/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) # 转储对象到文件
f.close()

del shoplist # 销毁 shoplist 变量

# 从文件找回对象
f = open(shoplistfile, 'rb')
storedlist = pickle.load(f) # 从文件加载对象
print(storedlist)
```

输出：

```
   $ python pickling.py
```

['apple', 'mango', 'carrot']

范例如何工作：

为了将对象存储到文件，我们必须首先*'wb'*写二进制文件模式打开文件然后调用 *pickle* 模块的 *dump* 函数。这个过程叫做封藏(pickling)对象。

接下来我们使用 pickle 的 *load* 函数重新找回对象。这个过程叫做解封(unpickling)对象。

小结

我们已经讨论了各种形式的输入/输出，和利用 pickle 模块进行文件处理。

接下来，我们将会学习异常处理。

简介

当程序发生意外情况时则产生异常。
例如你需要读一个文件而这个文件并不存在会咋样？又或者是程序运行时你把它误删除了呢？
上述情形通过异常进行处理。
类似的，如果你的程序存在一些非法语句会发生什么呢？这时 python 会举手告诉你存在一个错误。

错误

考虑一个简单的 print 函数调用。如果我们把 *print* 错拼成 *Print*(注意大小写)，这时 python 将引发一个语法错误。

```
>>> 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
```

我们看到一个 *NameError* 被引发并且发生错误的位置也被打印出来。这就是一个错误处理器(error handler)为这个错误所进行的处理。

异常

我们尝试从用户读取输入，看看当键入 *ctrl-d*(注：windows 用户输入 ctrl-z)时会发生什么。

```
>>> 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
```

(注：也许你看到的信息会所有不同，但只要引发了 EOFError 即可)
可以看到 python 引发了一个被称作 *EOFError* 的异常，一般这意味着它遇到了一个非期望的文件尾(end of file)符号(注：windows 下为 ctrl-z)

处理异常

利用 *try...except* 语句使得我们可以处理异常。通常我们将语句放入 *try* 块而将错误处理放到 *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
```

代码如何工作：

我们将可能引起异常/错误的所有语句放入 *try* 块，然后将适当的错误/异常处理器放进 *except* 块/从句中。

except 从句可以处理一个单一的指定的错误或异常，或者一组括在小括号中的异常/错误。

如果没有给出异常或错误，则 except 会处理所有的错误和异常。

注意每个 try 至少要关联一个 except 从句，否则只存在 try 块有啥意义呢？

任何没有被处理的错误或异常都会导致 python 默认处理器的调用，它的作用仅仅是终止程序运行并将错误信息打印出来。前面我们已经见识过了。

你还可以为 *try..except* 块关联一个 *else* 从句。如果没有异常发生则 else 块被执行。

下面的例子中，我们将看到如何得到异常对象获取额外的异常信息。

引发异常

通过 *raise* 语句你可以引发异常。为 raise 语句提供错误/异常名后异常对象会被抛出。

你抛出的错误或异常必须是一个间接或直接派生自 Exception 类的类。

```
#!/usr/bin/python
# Filename: raising.py
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, expected at
least {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.
```

范例如何工作：

这里，我们创建了自己的异常类型。这个新的异常类型被称作 *ShortInputException*。

ShortInputException 拥有两个字段 – *length* 指出给定输入的长度，而 *atleast* 为程序希望输入的最小长度。

在 except 从句中，我们给定异常类并将其对象存储为一个变量。这就类似于函数调用中的形参与实参。

在这个特定的 except 从句中，我们利用异常对象的 length 和 atleast 字段向用户打印出适当的提示信息。

try...finally

假设你的程序正在读取一个文件。如何保证无论是否发生异常文件对象都能被适当的关闭？这可以通过 *finally* 块做到。

注意你可以同时为 try 块关联 except 和 finally 块。如果你希望同时使用两者则必须将一个嵌入另一个中。

```
#!/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('(Cleaning 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)
```

代码如何工作：

我们执行一个常见的读文件操作，但故意在打印每行后利用 *time.sleep* 函数让程序休眠 *2* 秒，因此程序会运行的比较慢。

当程序运行时输入 *ctrl-c* 将中断/取消程序的运行。

注意 ctrl-c 会导致抛出 *KeyboardInterrupt* 异常，随后程序结束。但在程序结束前 *finally* 会被执行因此文件对象永远都会被关闭。

with 语句

在 try 块中获得资源后在 finally 块中释放之是很常见的设计方式。因此 python 提供 *with* 语句给予更简洁的实现方式。

```
#!/usr/bin/python
# Filename: using_with.py
with open("poem.txt") as f:
    for line in f:
        print(line, end='')
```

代码如何工作：

程序的输出应该和上面的范例相同。程序的不同之处在于我们在 *with* 语句中使用 *open* 函数 – 如此 with 语句就会自动关闭文件了。

在幕后 with 与用户有一个协议。它将读取 open 语句返回的对象，这里我们将这个对象称为*"thefile"*

在 with 块的开始处，with 永远都会调用 *thefile.__enter__*方法而在块结束处又会调用 *thefile.__exit__*方法。

因此我们在 finally 块中的代码被委托给*__exit__*方法了。这将帮助我们避免反复的使用 *try..finally* 块。

关于此主题更多的讨论已经超出本书范围，详见
(http://www.python.org/dev/peps/pep-0343/) (注：看本地文档也行)

小结

我们已经讨论了 try...except 和 try...finally 的用法。并了解到如何创建自己的异常类型与引发异常。
下面，我们将研究 python 标准库。

简介

python 标准库作为 python 标准安装的一部分，其自身包含数量庞大的实用模块，因此熟悉 python 标准库非常重要，因为很多问题都能利用 python 标准库快速解决。
下面我们将研究标准库中的一些常用模块。完整的标准库模块列表可以在安装 python 时附带的文档中的*'Library Reference'*一节找到。
现在就让我们来看看这些模块吧。

提示
如果你感觉本章内容对于你过于超前，那么可以跳过本章。但是当你熟悉 python 编程后我强烈建议你把这章补上。

sys 模块

sys 模块包含一些系统相关的功能。先前我们已经见识过 sys.argv 列表，它包括命令行参数。
假设我们想要检查所使用的 python 命令行的版本，比方说我们需要确定正在使用的版本不低于 *3*。
诸如此类的功能正是 sys 模块所提供的。

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

代码如何工作：
sys 模块含有一个 *version_info* 元组用于提供版本信息。其第一个元素为主版本。因此我们可以通过检查它确保程序只会运行在 python 3.0 和 3.0 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

代码如何工作：

这里我们使用标准库中另一个名为 *warnings* 的模块，用于向最终用户显示警告信息。

如果 python 版本号小于 *3*，则显示相应的警告。

loggin 模块

如果你希望得到存储在某处的重要信息或调试信息，以便检查程序是否如期运行时该咋办呢？你如何将这些信息存储在某处呢？

这些可以通过 *logging* 模块做到。

```
#!/usr/bin/python
# Filename: use_logging.py
import os, platform, logging
if platform.platform().startswith('Windows'):
    logging_file = os.path.join(os.getenv('HOMEDRIVE'),
os.getenv('HOMEPATH'), '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
If we check the contents of test.log, it will look something like this:
    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
```

代码如何工作：

我们使用了 3 个标准库模块 – *os* 模块与系统交互，*platform* 模块取得平台信息，即操作系统信息。而 *logging* 模块用于记录日志信息。

首先，我们通过 *platform.platform*(详见 *import platform; help(platform)*)返回的字符串检测操作系统类型。

如果为 windows 系统，则分别计算出主驱动器，主目录与文件名，这个文件用于存储相关信息。然后将这三部分合并得到文件的全路径。

对于其他平台，我们只需得到用户的主目录就能计算出文件的全路径了。

我们之所以没有简单的使用字符串连接合并这三部分而是利用 *os.path.join*，原因在于这个特殊的函数可以确保路径格式符合特定系统的规范。
之后我们配置 *logging* 模块，指示在我们指定的文件中以特殊的格式写入所有信息。
最后，我们就能写入信息了，它们可以是调试信息，警告信息甚至是危机信息(critical messages)。
一但程序开始运行，我们就可以检查这个文件以了解程序发生了什么，而用户并不会看到这些信息。

urllib 与 json 模块

如果我们让自己编写的程序在 web 上获得搜索结果是不是很有趣呢？我们现在就来研究下。
这个功能可以通过少量模块实现。第一 *urllib* 模块使得我们可以访问 internet 上的任何网页。
这里我们准备利用雅虎搜索获得搜索结果，它恰好以一种被称作 *JSON* 的格式为我们提供搜索结果。
这种格式分析起很方便，因为我们使用的是标准库中的内建 *json* 模块。

```
#!/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']))
```

代码如何工作：

(注：这个例子目前有错误，暂且跳过)。

Module of the Week 系列

标准库中还有更多内容有待探索，例如
debug(http://docs.python.org/dev/library/pdb.html)，
处理命令行选项(http://docs.python.org/py3k/library/getopt.html)，
正则表达式(http://www.diveintopython.org/regular_expressions/index.html)等等。
而进一步研究标准库的最好办法就是阅读 Python Module of the Week 系列了
(http://www.doughellmann.com/projects/PyMOTW/)。

小结

我们已经探索了许多标准库模块中的功能。强烈推荐浏览 python 标准库文档以便对所有模块有个概念性的认识(http://docs.python.org/py3k/library/)
接下来，我们将概览 python 的方方面面让我们的 python 之旅更加完整

简介

迄今为止我们已经学习了 python 中的大多数常用知识。本章中我们会接触到更多的知识，使得我们更全面的掌握 python。

传递元组

你是否希望过从函数返回两个不同的值？做到这点使用元组即可。

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

注意 *a, b = <某些表达式>*的使用，它会将表达式的结果解释为带有两个值的元组。

如果你希望将结果解释成*(a, <其它值>)*的形式，那么你要做的就象在函数形参中的那样：

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

这种语法也暗示出在 python 中快速交换两个变量值的方法：

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

特殊方法

有一些诸如*__intit__*和*__del__*的方法在类中拥有特殊的含义。特殊方法用于模拟某些内建类型的行为。

例如，你希望为你的类使用 *x[key]*索引操作(就像在列表和元组中那样)，那么你仅仅需要实现*__getitem__*方法就可以了。

顺便思考一下，python 正是这样实现 list 类的！

一些有用的特殊方法列在下表中。如果你想了解所有的特殊方法，详见 (http://docs.python.org/py3k/reference/datamodel.html#special-method-names)。

方法名	解释
__init__(self, ...)	在对象被返回以变的可用前调用

__del__(self)	在对象被销毁前调用
__str__(self)	在使用 print 函数或 str()时调用
__lt__(self, other)	在使用小于运算符时(<)调用。 类似的其它运算符也存在对象的特殊方法(+, > 等)
__getitem__(self, key)	当使用 x[key]索引操作时调用
__len__(self)	当使用内建 len()函数时调用。

单语句块

我们已经看到每个语句块都根据它的缩进级别将彼此区分开。不过有一个例外。如果某语句块只包含单条语句，你可以把它放到同一行，例如条件语句或循环语句。

下面的例子清楚的说明了这点：

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

注意上面的单条语句被放置到同一行而没有作为单独的块。

虽然你利用这点可以让程序变的更短，但我强烈建议你避免之(除了错误检测)，主要原因是使用适当的缩进可以更方便的添加额外的语句。

Lambda 表达式

lambda 语句用于在运行时创建并返回新的函数对象。

```
#!/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
```

代码如何工作：

在运行时我们利用函数 *make_repeater* 创建一个新的函数对象并返回它。其中一条 lambda 语句用于创建函数对象。

本质上这条 lambda 需要一个参数后跟一个相当于函数体的单表达式，这个表达式的值将成为函数的返回值。

注意就算 *print* 这样的语句也不能出现在 *lambda* 中，只能是表达式。(注：py3k 中 print 是个函数，作者 out 了)。

思考一下

我们能否利用 lambda 创建一个比较函数并将其提供给 *list.sort()*?

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

列表解析(List Comprehension)

列表解析用于从一个现有的列表派生出一个新的列表。

假设你有一个数字列表，你想让其中所有大于 *2* 的元素乘以 *2* 并组成一个新的列表。

类似问题正是使用列表解析的理想场合。

```
#!/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]
```

代码如何工作：

当某些条件满足时(*if i > 2*)我们执行某些操作(*2 * i*)，由此产生一个新列表。注意原始列表并不会被改变。

使用列表解析的好处在于当我们使用循环遍历元素并将其存储到新列表时可以减少样板代码量。

函数接收元组和列表

这里有一种特殊的方法可以将函数的形参当做元组或字典，那就是分别使用*和**前缀。

当需要在函数内得到可变数量的实参时这个方法很有用。

```
>>> 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 *args* 变量带有*前缀，因此所有额外的实参都会被当做一个元组存入 args 中并传给函数。

如果这里的*换成**，则所有额外的形参都会被当做一个字典的键/值对。

exec 和 eval

exec 函数用于执行 python 语句，不过这些语句储存在字符串或文件中而不是程序自身中。
例如，我们可以在运行时产生一个包含 python 代码的字符串，然后利用 exec 执行之。

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

与之类似，*eval* 函数用于执行合法的存储在字符串中的 python 表达式。下面是一个简单的例子。

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

assert 语句

assert 用于断言一个表达式为真。
例如，你需要确保正在使用的列表至少有一个元素，否则引发一个错误，这正是使用 assert 的理想场合。
当 assert 语句断言失败，则引发一个 *AssertError*。

```
>>> 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
```

assert 应当慎重使用。多数时候用于捕获异常，处理问题或是向用户显示错误后随即终止程序。

repr 函数

repr 函数用于获得对象的正规字符串表示。有趣的是多数时候 *eval(repr(object))* 等于 *object*。

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

True
基本上，repr 函数用来获得一个对象的可打印形式。你可以通过在类中定义 *__repr__*方法控制 repr 的返回值。

小结

本章我们介绍了更多的 python 特性，虽然没有引入 python 的所有特性但足以应付实践中的大多数应用了。
接下来我们考虑如何进一步学习 python。

接下来做什么

如果你有认真通读本书之前的内容并且实践其中包含的大量例程，那么你现在一定可以熟练使用 python 了。
同时你可能也编写了一些程序用于验证 python 特性并提高你的 python 技能。如果还没有这样做的话，你应该去试试。
现在的问题是**接下来应该做什么**？
我建议你先解决下面的问题：

创建你自己的命令行版本的**通讯录**程序，利用它你可以浏览修改删除或搜索诸如朋友，家人，同事等联系人和

他们的 email 地址/或电话号码等信息。这些信息必须存起来以便需要时提取。
思考下我们已经学到的各种知识，这个问题其实相当简单。
如果你感觉还是不好下手的话，这有一些提示。
提示(其实最好不要阅读这个提示)

创建一个表示联系人(persion)信息的类。使用字典存储联系人对象并以人物的名字作为字典键。

然后利用 *pickle* 模块把这些对象永久存储到你的硬盘中。

最后通过字典的内建方法 *add, delete* 和 *modify* 分别增加删除修改联系人。
只要你有能力完成这个程序，你就可以自信的说你是一个 python 程序员了。
那么现在马上给我发送 mail(http://www.swaroopch.com/contact/)好感谢我编写了如此强大的教程吧:-)
当然这步是可选的但我还是希望你发过来。
同时，也请考虑下捐赠，提供改进意见建议或是自愿翻译本书，以支持本书的持续发展。(注：我翻译的我翻译的我翻译的...)
如果你觉得上面的程序太简单，这还有另一个：

实现 replace 命令（http://unixhelp.ed.ac.uk/CGI/man-cgi?replace）

此命令用于在给定的文件列表中的所有文件中替换指定的字符串。
replace 命令可以简单的执行字符串替换也可以复杂的进行模式查找(正则表达式)，这取决于你的意愿。
下面是一些继续学习 python 的方法：

实例代码

学习程序设计最好的办法就是编写阅读大量代码：

- PLEAC 项目(http://pleac.sourceforge.net/pleac_python)
- Rosetta 代码资料库(Rosetta code repository)(http://rosettacode.org/wiki/Category:Python)
- java2s 网的 python 范例 (http://www.java2s.com/Code/Python/CatalogPython.htm)
- Python Cookbook (http://code.activestate.com/recipes/langs/python/)

对于某些种类的问题 Python Cookbook 提供了许多解决问题的珍贵技巧和诀窍。此网是每个 python 用户都必读的。

问题与解答

- 官方 Python Dos and Don'ts (http://docs.python.org/dev/howto/doanddont.html)(注：Dos and Don’ts 是 可为与不可为 的意思)
- 官方 Python 问与答 (http://docs.python.org/faq/general)
- Norvig 的宝贵的已解答问题列表(http://norvig.com/python-iaq.html)
- Python 面试问答 (http://dev.fyicenter.com/Interview-Questions/Python/index.html)
- StackOverflow 网的 python 相关问题 (http://stackoverflow.com/questions/tagged/python)

技巧和诀窍

- Python 技巧和诀窍(http://www.siafoo.net/article/52)
- 使用 python 的高级软件木工（http://ivory.idyll.org/articles/advanced-swc/)
- 引人入胜的 python(Charming Python) (http://gnosis.cx/publish/tech_index_cp.html)是一系列优秀的 python 相关的文章，作者 David Mertz.

书籍，文章，教程，视频

逻辑上看完本书应该读读 Mark Pilgrim 那超棒的 *Dive Into Python* 一书 (http://www.diveintopython.org/)，你可以在线完整阅读。
此书详细的探索了诸如正则表达式，XML 处理，web 服务，单元测试等内容。
其它有用的资源：

- ShowMeDo 的 python 视频 （http://showmedo.com/videotutorials/python）
- GoogleTechTalks 的 python 视频 (http://youtube.com/results?searchquery=googletechtalks+python)
- Awaretek 的 python 教程的综合列表(http://www.awaretek.com/tutorials.html)
- Effbot 的 Python Zone (http://effbot.org/zone/)
- 每个 Python-URL!邮件尾的链接 (http://groups.google.com/group/comp.lang.python.announce/t/37de95ef0326293d)
- Python Papers (http://pythonpapers.org)

讨论组

如果你被某个问题难住了，也不知道找谁求助，那么 *comp.lang.python* 讨论组是个提问的好地方。
(http://groups.google.com/group/comp.lang.python/topics)
记住尽量自己解决问题，不行再去发问。

新闻

如果你想了解 python 的最新动态，请关注 Official Python Planet (http://planet.python.org) 和 Unofficial Python Planet (http://www.planetpython.org).

安装库

python 包索引(Python Package Index)拥有数量巨大的开源库，你可以在自己的程序中使用它们。(http://pypi.python.org/pypi)
安装和使用这些库，你可以使用 Philip J. Eby 的优秀的 EasyInstall 工具。(http://peak.telecommunity.com/DevCenter/EasyInstall#using-easy-install)。

图形软件

如果你想使用 python 创建自己的图形程序。那么可以使用已绑定到 python 上的 GUI(图形用户界面)库。
绑定允许你在自己的程序中使用这些库，而库本身是用 C/C++或其它语言编写的。
使用 python 你可以选择很多种 GUI 库：

PyQt

这是绑定到 python 的 Qt 工具包，它是创建 KDE 的基石。
Qt 非常易用，功能又很强大，尤其是仰仗于它的 Qt Designer 与出色的 Qt 文档。
如果你在创建开源软件(GPL'ed)则 PyQt 是免费的, 相反创建私营闭源软件的用户就要掏银子买它了。
从 Qt4.5 开始你同样可以用它创建非 GPL 软件。
作为入门可以阅读 PyQt 教程(http://zetcode.com/tutorials/pyqt4/)或者 PyQt book (http://www.qtrac.eu/pyqtbook.html).

PyGTK

GTK+工具包的 python 绑定。它是 GNOME 的基础。
GTK+含有很多奇怪的用法，不过一旦熟悉它你就能够快速创建 GUI 应用了。其中 Glade 图形界面设计器是必不可少的。
GTK+的文档仍然完善中。GTK+在 linux 上工作的很好，但其 windows 实现仍未完成。
另外使用 GTK+你既可以创建开源也可以创建私营软件。
入门可以阅读 PyGTK 教程(http://www.pygtk.org/tutorial.html)

wxPython

这是绑定到 python 的 wxWidgets 工具包。
wxPython 有一定的学习曲线。但是具有很强的可移植性，可以运行在 linux，windows，Mac 甚至是嵌入式平台之上。
wxPython 拥有很多可用的 IDE，其中包括 GUI 设计器和诸如 SPE(Stani 的 python 编辑器)(http://spe.pycs.net)和
wxGlade(http://wxglade.sourceforge.net/)的开发工具。
入门可以阅读 wxPython 教程(http://zetcode.com/wxpython/)

Tkinter

这是现存最老的 GUI 工具包之一。如果你用过 IDLE，它就是一个使用 Tkinter 编写的程序。
Tkinter 没有什么不错的视觉外观，因为它是个守旧派。
Tkinter 是可移植的能够运行在 Linux/Unix 和 Windows 上。更重要的 Tkinter 是标准 python 发布版的一部分。
入门可以阅读 Tkinter 教程(http://www.pythonware.com/library/tkinter/troduction/)。
更多的 GUI 库选择，见 python 官网的 GuiProgramming 维基页面 (http://wiki.python.org/moin/GuiProgramming)

GUI 工具小结

很不幸，python 没有一个标准 GUI 工具。我建议根据你的情况选择上面的工具。
考虑的第一歌因素是你是否愿意付费使用 GUI 工具。
第二你是否希望程序只运行在 windows 或 mac 或 linux 还是希望都能运行。
第三对于 linux 平台，你是一个 KDE 还是一个 GNOME 用户呢。
更详细广泛的分析，见 Python Papers 第 26 页卷 3 问题 1(http://archive.pythonpapers.org/ThePythonPapersVolume3Issue1.pdf)

各种 python 实现

一个程序设计语言通常包含两部分 – 语言和软件。语言指出如何编写程序。而软件用来运行我们的程序。
我们一直在用 *CPython* 运行我们的程序，之所以称为 CPython 是因为它是用 C 语言实现的并且为标准 python 解释器。
另外还有其它的软件也可以运行 python 程序：

Jython (http://www.jython.org)

一个运行在 java 平台的 python 实现。这意味着你可以在 python 语言内部使用 java 库和类，反之亦然。

IronPython

(http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython)

一个运行在.NET 平台的 python 实现。即你可以在 python 语言内部使用 java 库和类，反之亦然

PyPy (http://codespeak.net/pypy/dist/pypy/doc/home.html)

一个用 python 写的 python 实现！

这是一个研究项目，用于使之可以快而容易的改进解释器，因为解释器本身就是用动态语言编写的。（而不是类似上面的 C, java 或 C#等静态语言）

Stackless Python (http://www.tackless.com)

一个专用于基于线程性能的 python 实现。

除此之外还有 CLPython(http://common-lisp.net/project/clpython/)一个 Common Lisp 编写的 python 实现。
IronMonkey(https://wiki.mozilla.org/Tamarin:IronMonkey)是一个运行在 JavaScript 解释器之上的 IronPython 的接口，
这可能意味着你可以使用 python(替代 JavaScript)编写 web 浏览器程序(“Ajax”)。

以上的每个实现都有自己的擅长领域。

小结

现在我们已经来到本书的结尾了。不过据说，结束意味着另一个开始！
你现在是一个满腔热切的 python 用户，很可能摩拳擦掌准备利用 python 解决大量问题。
现在你可以让计算机自动完成许多以前无法想象的事情或是编写游戏或是更多更多。
既然如此！那就行动起来大干一场吧！