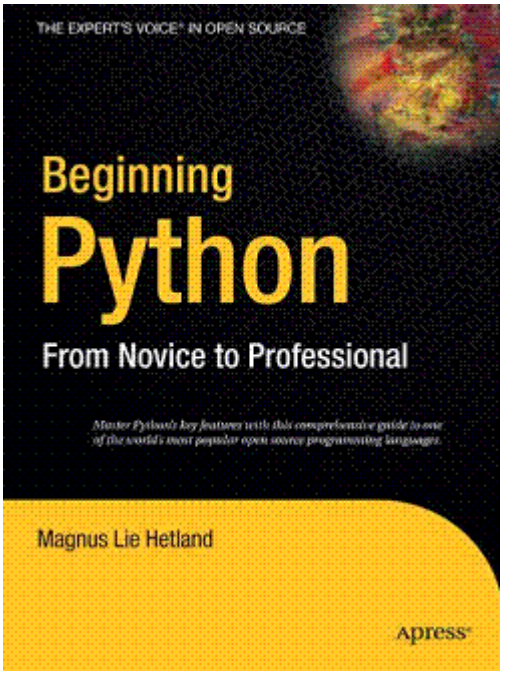
Beginning Python:

From Novice To Professional

中文版测试版(1-10章)



翻译：

siwei/www.siwei.org

第一、三、四、五、六、七、八和十章

junwei/www.laozeng.net

第二、九章

排版：siwei

有关翻译：

1. 翻译本书是为了练习英文阅读和理解，还有就是学习 Python。质量可以说比较差，但是应该比市面上很多滥竽充数的废纸书强。
2. 由于 PDF->DOC->PDF 的原因，排版的很差，我们尽量保持原书的格式。在页面大小上，保持了原书的尺寸（177.8x288.6mm），而没有使用 A4 尺寸。
3. 原书是在 Python2.4 时代写成，但是所有的代码在 Python2.5 中都可以使用。我们对所有出现的代码都进行了测试，以保证可以在 2.5 的 IDLE 中运行成功。
4. 鉴于我们的烂水平，问题肯定多多，希望大家不吝赐教，在这里我们先谢过了。

关于一些术语的翻译：

由于目前 Python 在中国还没有形成类似 C/C++/Java/C#这样普遍的学习氛围，所以一些在其他语言中没有出现过的术语，以及出现过但是含义不同的术语会有很多种翻译方式。在这本书的翻译中，我们约定以《Python Tutorial》中出现的术语翻译为标准。如果《PT》中没有出现的，则在互联网上寻找以定夺。如果实在没有找到，我们会以自己的翻译写出，并且详细注明。

List	列表	Sequence	序列
Tuple	元组	Dictionary	字典
Set	集合		
Attribute	特性	Property	属性
Type	类型	Reference/Refer to	参考
Function	函数	Method	方法
expression	表达式	list comprehesion	列表推导式
Iterate	迭代	Enumerate	枚举
Operator	运算符		

第一章

黑客进行时：基础

Instant Hacking

现在是黑客[1]进行时。本章中，你将学会如何借助于说一门计算机听得懂的语言——Python——来对其进行控制。这没什么可难的，如果你懂些计算机的基本操作，就能跟着例子自己试着去做。我会贯穿基础知识于其中，以让人发狂的简单开始。不过，Python是一门非常强大的语言，你不久就可以做一些高级的事情了。

首先，我会告诉你需要什么软件。然后是一点点算法和主要的构成、表达式和语句。这些部分过后，会有一些小例子（大多数只用了简单的算法），你可以自己在Python的交互解释器（参看本章后面的“交互式解释器”部分）。你会学习到变量（Variable）、函数（Function）和模块（Module），在掌握这些主题后，我会告诉你如何书写和执行大型的程序。最后，我会说一些几乎是每个Python程序中的重要方面——字符串（String）的事情。

安装Python
Installing Python

在开始编程前，你需要一些新软件。下面就是一些关于如何下载、安装Python的描述。如果你想直接跳到安装部分而不看详细的指导，可以直接去http://python.org/download下载最新版本的Python。

Windows

要在Windows中安装Python，按照下面步骤：

1．打开浏览器，访问http:/www.python.org；
2．点击“Download”链接。
3．你应该能看到一些链接，比如“Python 2.4”和“Python 2.4 Windows installer”之类。点击“Windows installer”链接——你应该能直接获得安装文件了。现在跳到第五步。如果你找不到类似的链接，那么点击版本号最高的链接，比如“Python 2.4”。你应该能最先看到这类链接。对于Python2.4，你也可以直接访问http://www.python.org/2.4
4．按照对于Windows用户的提示，然后下载一个叫做python-2.4.msi（或者类似）的文件，2.4所在的位置应该是最新版本的版本号。
5．将Windows Installer文件放在电脑的任何位置均可，比如C:\download\python-2.4.msi。（只要建立一个你之后能找到的文件夹就好）
6．双击运行文件。然后会出现Python的安装向导，然后就简单了，只需要接受默认设

置，等到安装程序完成，你就能用了！

1.“黑客”（Hacker/Hacking）并不同于“骇客”（Cracker/Cracking），后者描述的是计算机犯罪行为。两者经常被混为一谈。“Hacking”意味着在编程时获得乐趣。更多的信息，请看Eric Raymond的文章“How to Become a Hacker”（如何成为黑客），地址为http://www.catb.org/~esr/faqs/hacker-howto.html

■**注意** Python的Microsoft Windows版本以Windows Installer文件的方式分发，需要你的Windows版本至少能支持Windows Installer 2.0（或以后版本）。如果你没有安装Windows Installer，可以下载针对Windows 95、98、Me、NT4.0和2000的版本。Windows XP已经包含了Windows Installer，很多旧及其也一样。Python的下载页面上也有下载Installer的步骤。或者你可以访问Microsoft的网站：http://www.microsoft.com/downloads，搜索“Windows Installer”（或者直接从下载菜单里面选择），选择针对你所用平台的最新版本，然后按照步骤下载。如果你不知道你是否安装了Windows Installer，只要简单的执行上面提示中的第六步。如果你能看到安装向导，证明一切正常。访问http://www.python.org/2.4/msi.html查看更多和Python安装程序相关Windows Installer的高级特性。

假设安装一切正常，现在你应该能在Windows的开始菜单中找到新的程序。按照开始->程序->Python2->IDLE(Python GUI)的步骤运行Python Integrated Development Environment（IDLE，Python综合开发环境）

你应该能看到一个类似于图1.1的窗口。如果你感觉有点犯晕，那么在菜单里面选择Help->IDLE Help，你会看到一个简单的关于菜单和基本使用的帮助。更多关于IDLE的文档，访问http://www.python.org/idle（这里你也能看到除Windows外其他平台上运行IDLE的信息）。如果你按下F1，或者选择Help->Python Docs，你会得到完整的Python文档。（这个文档最好用来作为“库参考”），所有的文档都是可搜索的。

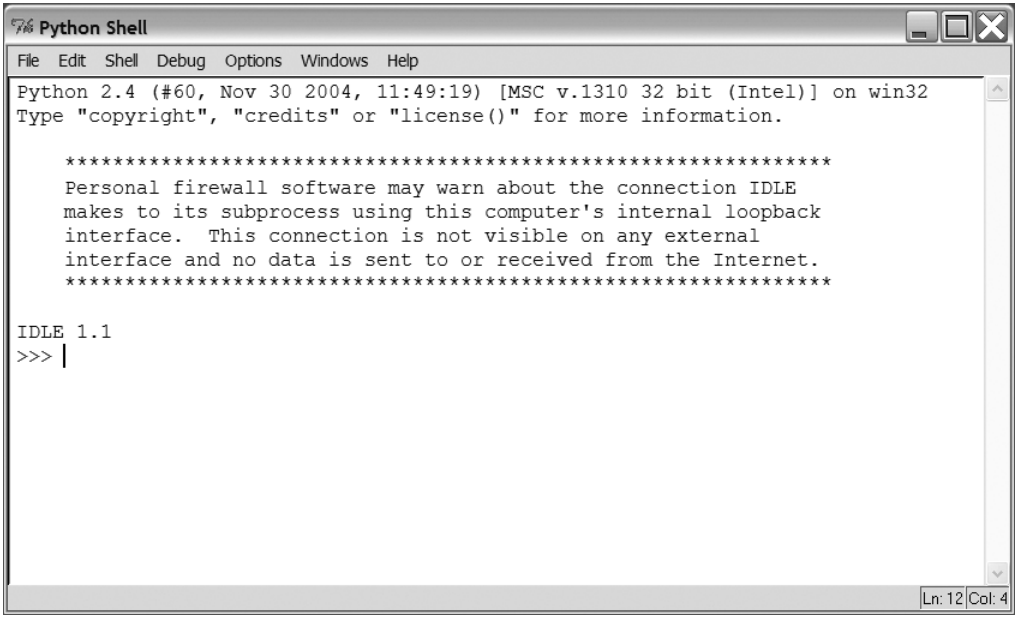


图 1-1 *IDEL交互式Python外壳*

一旦IDLE交互式Python外壳可以运行，接下来你就可以阅读本章后面的“交互式解释器”部

分。

2. 这个菜单选项可能包含你的Python版本，比如Python 2.4。

Linux和UNIX
Linux and UNIX

在很多——如果不是大多数的话——Linux和UNIX的安装过程中，Python的解释器就已经存在了。你可以在命令提示符下输入python command进行验证，如下：

```
$ python
```

执行这个命令应该会启动交互式Python解释器，同时会有如下输出：

```
Python 2.4 (#1, Dec  7  2004, 09:18:58)
[GCC 3.4.1] on sunos5
Type "help",  "copyright", "credits" or "license" for  more information.
>>>
```

■ **注意** 退出交互式解释器，使用Ctrl+D。

如果Python解释器没有被安装，你可能会得到如下的错误信息：

```
bash: python: command not found
```

这种情况下，你需要自己安装Python，下面会讲到。

使用RPM

如果你正在运行包含有RPM包管理器的Linux，那么按照下面的步骤安装Python RPM包：

1) 访问下载页（参考在Windows上安装Python步骤的前两步）
2) 选择版本号最新的链接，比如“Python 2.4”（不要选择带有“sources”字样的链接）。你应该能最先看到这个链接。对于Python2.4而言，你可以直接访问http://www.python.org/2.4。之后按照对于Fedora用户的指导来做：点击“RPMs”链接。
3) 下载所有的二进制PRM包，并保存在临时的位置（比如~/rmps/python）。
4) 确保你以系统管理员（root）身份登录，并且位于之前存储RPM的位置。确保此目录中没有其他RPM。
5) 执行rpm --install *.rpm命令安装所有的包。如果你已经安装了旧版本的Python，希望进行升级，你应该使用rpm --upgrade *.rpm命令。

■**警告** 前面说到的命令会安装当前目录下所有的PRM文件。确保你位于正确的目录，并且目录中只有你希望安装的包。如果你希望小心一些，你可以分别指定每个包的名字。有关RPM的更多信息，请查看man页。

现在你应该能运行Python了。有可能你会遇到一些无法解决的问题——你可能会缺少安装Python的包。你可以访问诸如http://www.rpmfind.net一类的搜索引擎定位这些包。

有些时候二进制的RPM包会针对一个Linux发行版设定（比如Red Hat Linux），无法顺利在其他的Linux版本（比如Mandrake Linux）上运行。如果你发现二进制包无法正常使用，试着下载RPM的源文件（文件名类似packagename.src.rpm）。之后你可以利用下面的命令建立一套二进制包：

```
rpm --rebuild packagename.src.rpm
```

上面的packagename.src.rpm应该是你需要重建的包的名字。完成之后，你应该会得到一系列新的RPM文件，你可以按照之前的步骤进行安装。

■**注意** 你必须以root（管理员帐户）身份登录才能使用RPM安装程序。如果你没有root权限，你应该自己编译Python，本章后面的“从源文件编译”部分会讲到。

除了rpm之外，还有一些其他的针对Linux的包系统和安装方式、如果你正在运行包含某种形式包管理器（Package Manager）的Linux，也可以用它安装Python。

■**注意** 你可能需要拥有管理员特权（root帐户）来使用包管理器在Linux中安装Python。

举例来说，如果你正在使用Debian Linux，你应该可以用下面的命令来安安装Python：

```
$ apt-get install python2.4
```

如果是Gentoo Linux的话，便用：

```
$ emerge python
```

两例中，$是提示符。用你所安装的版本号替换上面的2.4字样。

从源文件编译

如果你没有包管理器，或者不愿意用的话，你可以自己编译Python。如果你正在使用UNIX系统却没有root权限（安装特权）的话这应该是个可选方法。本方法非常灵活，可以让你在任何地方安装Python，包括你自己的主目录（home directory）。编译并安装Python，可以按照以下步骤：

1．访问下载网页（参考在Windows上安装Python步骤的前两步）

2．按照指导下载源代码。
3．下载扩展名为tgz的文件。将其保存在临时位置，比如说你想将Python安装在自己的主目录，你可以将它放置在类似于~/python的目录中，进入目录（比如使用cd ~/python命令）。
4．使用tar -xzvf Python-2.4.tgz（2.4是你所安装的代码的版本号）解压缩文件。如果你是用的tar版本不支持z选项，你可以使用gunzip先解压缩，然后使用tar -xvf命令。如果解压缩过程中出错，试着重新下载。下载时候有时会出错。

5．进入解压缩好的文件夹：
```
$ cd Python-2.4
```
6．现在你应该能执行下面的命令：
```
./configure --prefix=$(pwd)
make
make install
```
7．最后你应该能在当前文件夹内得到一个名为python的可执行文件（如果没用的话，请参看包含在下载文件中的README文件）。将当前文件夹的地址放置到你的系统变量PATH中，大功告成。
8．查看其它配置指示，执行：
```
./configure --help
```

苹果机
Macintosh

1) 如果你正在使用苹果机，按照以下步骤：
2) 访问下载页面（参考在Windows上安装Python步骤的前两步）
3) 点击Macintosh OS X installer所在的链接，应该会跳转到MacPython的下载页面，上面会有更多信息。MacPython页面也有针对于Mac OS的旧版本Python。

■**注意** 对于Python2.4版本而言，OS X版本安装文件还是2.3版本。

其他版本
Other Distributions

现在你已经安装了标准Python分发版。除非你对于其他选择有特殊的兴趣，现在的应该已足够。如果你好奇（并且，可能还有些胆量），那么继续读下去……
Python还有其他的分发版本，最有名的算是ActivePython。另外相对来说不太出名但是很有趣的版本是Stackless Python。这些分发版都是基于由C语言书写的Python标准版本的。Jython和IronPython则是两个有不同用途的分法办。如果你对于IDLE以外的其他开发环境有兴趣，表1-1里面有些选择：

表1-1 一些针对Python的完整IDE

环境	描述	下载地址．．．
IDLE	标准Python环境	http://www.python.org/idle
Pythonwin	面向Windows的环境	http://www.python.org/windows
ActivePython	特性包：包含Pythonwin IDE	http://www.activestate.com
Komodo	商业性IDE	http://www.activestate.com

Wingware	商业性IDE	http://www.wingware.com
BlackAdder	商业性IDE 并且含有Qt GUI生成器	http://www.thekompany.com
Boa Constructor	免费的IDE和GUI生成器	http://boa-constructor.sf.net
Anjuta	Linux/UNIX上的万能IDE	http://anjuta.sf.net
ArachnoPython	商业性IDE	http://www.python-ide.com
Code Crusader	商业性IDE	http://www.newplanetsoftware.com
Code Forge	商业性IDE	http://www.codeforge.com
Eclipse	流行、灵活并且开源的IDE	http://www.eclipse.org
eric	免费的Qt IDE	http://eric-ide.sf.net KDevelop
	针对KDE的跨语言IDE	http://www.kdevelop.org
VisualWx	免费GUI生成器	http://visualwx.altervista.org
wxDesigner	商业性GUI生成器	http://www.roebling.de
wxGlade	免费GUI生成器	http://wxglade.sf.net

ActivePython是由ActiveState（http://www.activestate.com）发行的Python版本。这个版本的内核同标准的Windows版本Python相同。最大的不同点就是它包含了许多额外的分离可用的工具（模块）。如果你正在使用Windows的话值得一看。

Stackless Python是Python的重实现版本，基于原始内核，但是有一些重要的内部改变。对于入门用户来说没有多大区别，标准的版本反而更有用。Stackless Python的最大优点就是允许深层次的递归和更加有效的多线程使用。刚才提到过了，两者都是高级特性，一般用户并不需要，你可以在http://www.stackless.com下载Stackless Python。

Jython（http://www.jython.org）和IronPython（http://www.ironpython.com）则大有不同——他们都是其他语言实现的Python。Jython利用Java实现，运行于Java虚拟机，而IronPython利用C#实现，运行于公共语言运行时（Common Language Runtime）的.NET和MONO执行器上。本书撰写时，Jython相当稳定，但是延迟于Python——目前Jython的版本为2.1，Python为2.4。两个版本的语言具有相当的不同。IronPython非常新，并且还在试验阶段。而且它的确可用，一些测试表明它的运行速度比普通Python要快。

时常关注，保持更新
Keeping In Touch and Up to Date

Python语言还在不断发展。python.org网站对于查找最新更新和相关工具来说是无价之宝。访问相应的发布版本查看新的特性，比如http://python.org/2.4针对2.4版本。然后你就能看到Andrew Kuchling对于新版本新特性的详细描述。对于2.4版本来说地址为http://python.org/doc/2.4/whatsnew。如果在本书出版后还有新的版本发布，你可以访问这些网页查看新的特性。

如果你想跟随最新发布的第三方Python模块和软件的脚步，你可以查看Python的邮件列表python-announce-list，对于Python的一般讨论可以使用python-list，但是我们在此敬告：此列表会有很多流量产生。这些列表都能在http://mail.python.org找到。如果你是Usenet用户，这两个邮件列表也能分别在comp.lang.python.announce和comp.lang.python找到。如果你有些迷惑，那么可以试试python-help邮件列表（和前面两个列表位置一样），或者直接发邮件到help@python.org。在此前你应该看看自己的问题是不是经常会有人遇到，访问http://python.org/doc/faq查看Python的FAQ，或者直接在网络上搜索。

交互式解释器
The Interactive Interpreter

当你启动Python时候，你可能会看到跟下面相似的提示：

```
Python 2.4 (#1, Dec  7  2004, 09:18:58)
[GCC 3.4.1] on sunos5
Type "help",  "copyright", "credits" or "license" for  more information.
>>>
```

■**注意** 解释器的准确外观和错误信息取决你所使用的版本。

看起来不是很有趣，但相信我——会有趣的。这是你进入黑客大殿的门——是你控制电脑的第一步。更实际说来，这是交互式Python解释器。试着输入下面的命令看看是否有用：

```
>>> print "Hello, world!"
```

When you press the Enter key, the following output appears:
当你按下回车后，会有下面的输出：

```
Hello, world!
>>>
```

■**注意** 如果你熟悉其他的计算机语言，你可能会习惯于每行以分号结束。Python则不用，一行就是一行，不管多少。如果你喜欢的话可以加上分毫，但是不会有任何作用（除非同一行还有更多代码），这也不是普遍的做法。

发生了什么？那个>>>就是提示符。你可以在这里写点什么，比如print "Hello, world!"。如果你按下回车，Python解释器会打印出"Hello, world!"字符串，下面又会有一个新的提示符。

■**注意** 本章中的术语“打印"意为在屏幕上写出文本，并非在打印机上输出真实拷贝。

如果你写点完全不一样的呢？试试看，比如：

```
>>> The Spanish Inquisition
SyntaxError: invalid   syntax
>>>
```

显然，解释器不明白你的输入。（如果你使用除IDLE外的其他解释器，比如Linux的命令行版本，错误信息应该会明显不同）。解释器指出什么地方错了：它会在“Spanish”上加上红色背景进行强调（在命令行版本中，使用转义符号^）。

如果你喜欢，可以和解释器多玩点什么。（提示一下，可以在提示符后输入help然后按回车。前文说过，你可以按下F1获得IDLE的帮助）。或者我们继续往下。毕竟你不知道怎么跟解释器交流的话还是挺没意思的，不是吗？

---

3. 毕竟，没有人期待西班牙宗教裁判所(Spanish Inquisition)……

算法……什么？
Algo . . . What?

在你开始认真开始编程之前，我会试着告诉你什么是计算机编程。那么，它是什么？它就是告诉计算机做什么事。计算机可以做很多事情，但是不太擅长自己思考。它们需要被告知具体的细节。你需要用计算机明白的语言“喂”给它算法。“算法”不过是“程序”（procedure）或者“菜谱”（recipe）——如何做某事的详细描述——的冠冕堂皇的说法，考虑下面的文字：

> 某某以及某某、某某、鸡蛋和
> 某某：首先，拿一些某某
> 然后加入一些某某、某某和鸡蛋
> 如果需要特殊的辣味某某，加入一些某某
> 一直煮直到完成——每十分钟检查一次

这个菜谱可能不是非常有趣，但是它的结构则很有趣。它包括一系列有顺序的指令。有些指令可以直接完成（拿一些某某），有些则需要考虑一下（如果需要特殊的辣味某某），还有些则必须重复数次（每十分钟检查一次）。

厨房和方法都包括一些成分（ingredients）（对象、物品），以及指令（语句）。本例中，“某某”和“鸡蛋”就是因素，指令则包括添加某某、按照给定时间烹调等等。那么

让我们从一些非常简单的Python成分开始，看看你能用它们做些什么。

数值和表达式
Numbers and Expressions

交互式Python解释器可以当作非常强大的计算器，试试下面这个：

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

应该会给你答案为4。好像不是很难，那么看看这个：

```
>>> 53672 + 235253
288925
```

还没让你吃惊？不能否认这是非常普通的事情（我假定你曾经用过计算器，知道1+2*3和(1+2)*3的区别）。常用的运算符和平常使用方法相同——绝大多数。这里有个潜在的陷阱，就是整数除法（在3.0前的Python是如此，不过3.0也在短时间不会出现）。

```
>>> 1/2
0
```

发生了什么？一个整数（无小数部分的数）被另外一个除，算出的结果被截成了整数。有些时候这个行为比较有用，但是不是经常，你需要普通的除法。那么你需要怎么做？有两个可用的解决方案：你要用实数（带有十进制小数点的数）而不是整数，或者你可以告诉Python改变除法的方式。

实数在Python中被称为浮点数（Float，或者Float-point Number），如果除法过程中有一个数为浮点数，结果亦为浮点数：

```
>>> 1.0 / 2.0
0.5

>>> 1/2.0
0.5

>>> 1.0/2
0.5

>>> 1/2.
0.5
```

如果你特别希望Python能执行合适的除法，你可以在你的程序（后面会讲到书写完整的程序）前加上下面的语句，或者直接在解释器里面执行：

```
>>> from __future__ import division
```

还有另外一个方法，如果你在命令行（比如Linux）下面运行Python，那么使用命令-Qnew。两例中，除法都应该更合理了：

```
>>> 1 / 2
0.5
```

当然，单斜线不再用做前面说到的整数除法，但是还有一个专门给你准备的操作符——双斜线：

```
>>> 1 // 2
0
```

就算是浮点数，双斜线也会执行整数除法：

```
>>> 1.0 // 2.0
0.0
```

在“回到__future__”部分中，会对__future__作深入的解释。
现在已经见识过基本的运算符了（加减乘除），但是还有些非常有用的运算符：

```
>>> 1 % 2
1
```

这是余数（模除）运算符——x % y会得出x除以y的余数，比如：

```
>>> 10 / 3
3
>>> 10 % 3
1
>>> 9 / 3
3
>>> 9 % 3
0
>>> 2.75 % 0.5
0.25
```

这里10/3得3是因为结果被截去了小数。但是三三得九，所以你得到余数为1。当你计算9/3时，结果就是3，没有截取。那么余数就是0。当你在做一些类似本章前面“每十分钟”的菜谱例子的事情时这就非常有用了。你可以直接检查 时间%10 的结果是否为0。（关于如何做此事的描述，参看本章后面的“管窥：if语句”部分）上例中可以看到，余数运算符对于浮点数的运算也很正常。

最后一个运算符就是幂（乘方）运算符：

```
>>> 2 ** 3
8
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

注意，幂运算符比符号（减号）同数字连接的紧密，所以-3**2等同于-(3**2)。如果你想计算(-3)**2，就需要明确说明。

长整数
Large Integers

Python可以处理非常大的整数：

```
>>> 1000000000000000000
1000000000000000000L
```

发生了什么？数字后面被加了个L。

■**注意** 如果你正在使用2.2版本以下的Python，会看到如下的情况：

```
>>> 1000000000000000000
OverflowError: integer   literal too large
```

新版本的Python则在处理大数时候更加灵活

普通整数不能大于2147483647（或者不能小于-2147483648），如果你真的需要大数，需要使用长整型。长整形数书写方法和普通整数一样，但是结尾有个L。（理论上讲你用小写l也可以，但是看起来有点像1，所以我建议你不要用小写）。

前面的尝试中，Python把整数转换为了长整型，但是你也能自己完成，试试下面的大数：

```
>>> 10000000000000000000L
10000000000000000000L
```

当然，这在不能处理大数的旧版本Python中很有用。

那么你能不能对这些天文数字进行运算呢？当然可以，看看下面的：

```
>>> 198716398716398163918 6L * 19876398172639182 6L + 23
394976626432005567613000143784791693659L
```

可以看到，你可以在混合使用长整型数和普通整数。同样你也不用担心长整型数和整数的区别，除非你要检查类型，我们会在第七章中讲到——这事你几乎从来不用做。

十六进制和八进制
Hexadecimals and Octals

对于本部分做个总结，我应该告诉你十六进制数应该像这样写：

```
>>> 0xAF
175
```

八进制：

```
>>> 010
8
```

首数字都是0（如果你不知道这是什么，那么闭上眼跳到下一部分吧——你没错过任何重要的东西）。

■**注意** 附录B是Python数值类型和操作符的总结。

变量
Variables

另外一个你需要熟知的概念就是变量（Variable）。如果数学让你抓狂的话，别着急，Python之中的变量很好理解。变脸基本上说来就是代表（或者关联到）某数的名字。举例来说，你可能希望用x代表3，那么你就要执行下面语句：

```
>>> x = 3
```

这被称为赋值（assignment）。我们把值3赋给了变量x。另外一个说法就是我们把变量x绑定到了值（或者对象）3上面。在变量被复制后，你可以在表达式中使用变量。

```
>>> x * 2
6
```

注意你在使用变量之前已经对其赋值。毕竟使用没有值的变量也没意义，不是吗？

■**注意** 变量名称可以包括字母、数字和下划线（_）。变量不能以数字开头，所以Plan9是合法变量名，而9Plan不合法。

语句
Statements

到现在每只我们几乎已经完全说完了表达式，也就是菜谱的成分。但是语句——也就是指令呢？

事实上，我骗了你了。我已经介绍了两类语句：print语句和赋值语句。那么语句和表达式之间有什么区别？表达式**是**某事，而语句**做**某事（换句话说就是告诉计算机做什么）。比如2*2**是**4，而print 2*2**打印**4。你可能会问，那么区别呢。他们非常相像，看看下面的：

```
>>> 2*2
4
>>> print 2*2
4
```

当你在解释器中执行两个语句，结果是一样的。但是这是因为解释器总是把所有表达式的值打印出来（都使用repr表现，参看本章中后面字符串表现的部分）。这不是Python的一般表现。本章后面你会看到如何不用交互式提示编程，只在程序中写2*2是做不了什么事情的[4]。而写上print 2*2则会打印出4.

语句和表达式之间的区别在赋值时候会表现的更加明显一些。因为它们不是表达式，没有可以用交互式解释器打印出的值：

```
>>> x = 3
>>>
```

你可以看到，下面立刻出现了新的提示符。有些东西变化了，x被绑定了值3。

这也是语句的一般定义：它们改变事物。比如赋值语句改变变量，print语句改变你的屏幕显示。
赋值语句可能是任何计算机程序设计语言中最重要的语句类型，现在难以说清它们的重要性。变量就像临时“储备”（就像厨房菜谱中的锅碗瓢盆一样），但是变量的强大之处就在于，你不需要知道它们存储了什么值而就能进行操作[5]。比如和想知道x*y的乘积，但是你不知道x和y到底是多少。所以你的程序不需要知道变量到底是多少，但是可以有很多种方法进行使用，它们最终会在程序运行时候绑定（或关联到）值。

获取用户输入
Getting Input from the User

---

4. 你有些奇怪吧，是的，它的确还是做了些事情的，它会计算2*2的结果。但是结果不显示给用户，它在运算器本身之外没有任何边界效应（*side effects*）。
5. 注意“储备”一词的引号。值并不**储存**在变量中——他们储存在计算机内存的深处，**关联到**变量上。为了让你在阅读中更加明了，一个以上的变量可以同时关联一个值。

你已经看到，你能写不需要知道变量值的程序了。当然，解释器最终还是得知道变量的值。那么我们不赋值又会如何？编辑器只知道我们告诉它的，不是吗？

不必如此。你可以写个程序，让别人用。你不能预测他们会提供给程序什么值。那么让我们看看非常有用的input函数吧（马上我就会说到更多关于函数的事情）。

```
>>> input("The meaning   of life:
")The meaning of life: 42
42
```

当第一行的input(...)在解释器里面被执行后发生了什么？它在打印出了字符串“The meaning of life:”作为新的提示符，我输入了42然后按下回车。结果值就是我输入的，它被自动在最后一行被打印出来。不太有用，但是看看下面的：

```
>>> x = input("x: ")
x: 34
>>> y = input("y: ")
y: 42
>>> print x * y
1428
```

Python提示符（>>>）后面的语句可以算一个完整程序的部分了，输入的值（34和42）由用户提供。你的程序应该会得出结果1428，是这两个数的乘积。你不用知道用户输入到程序里面的数是多少，不是吗？

■**注意** 当你想把自己的程序存为其他用户可以执行的单独文件时非常有用。本章后面你会在“保存和执行你的程序”部分学到。

管窥：IF语句

为了更有趣一些，我会偷偷给你看一些你在第五章之前真的不该学到的东西：if语句。if语句可以让你在给定条件为真的情况下做出反应（另外的语句）。一类条件是相等性测试，使用相等运算符==（是的，是两个等号。一个等号的运算符是用来赋值的，还记得吗？）。

你可以简单的把这个条件放在“if”后面，然后用冒号将其和后面的语句隔开：

```
>>> if 1 == 2: print 'One equals    two'
...
>>> if 1 == 1: print 'One equals    one'
...
One equals one
>>>
```

你可以看到，当条件为假（false）时什么都没发生。当它为真的时候，后面的语句（本例中为print语句）被执行了。注意，如果在交互式编译器内使用if语句，需要按两次回车才能执行。（原因会在第五章内说明——现在别担心）。
所以，如果变量time绑定到当前时间的分钟数上，你可以使用下面的语句检查是不是“到了整点”。

```
if time % 60    == 0: print 'On the    hour!'
```

函数
Functions

在数和表达式部分内我曾经用幂运算符（**）来计算乘方，事实上你可以利用函数（function）来代替，调用pow函数：

```
>>> 2**3
8
>>> pow(2,3)
8
```

函数就像可以用来表现特殊行为的小程序一样。Python有很多能做奇妙事情的函数。事实上，你也可以自己定义函数（后面会讲），所以我们通常会把类似pow的普通函数叫做“内置”（built-in）函数。

上例中我使用函数的方式叫做“调用”（calling）函数。你可以给它提供参数

（parameter，本例中的2和3），它会“返回”（return）值给你。因为它返回了值，函数的调用就是另外一类表达式（expression），就像在本章前面讨论的算数表达式一样。事实上，你可以联合使用函数调用和操作符创建复杂的表达式。

---

6. 如果你忽略返回值，函数调用也可以用作语句。

```
>>> 10 + pow(2, 3*5)/3.0
10932.666666666666
```

■**注意** 具体的十进制值会依你所使用的Python版本而不同。

还有很多可以像这样用作数值表达式的内置函数。比如abs可以得到数的绝对值，round函数会把浮点数截取为最近的整数值：

```
>>> abs(-10)
10
>>> 1/2
0
>>> round(1.0/2.0)
1.0
```

注意最后两个表达式的不同。整数除法总是会截取结果的小数部分，而round函数截取到最近的整数（译注：即四舍五入）。但是如果你想截取一个给定的数字呢？比如说你知道一个人是32.9岁——但是你想给它截取为32，因为她还没到33。Python有这样的函数（叫做floor）——直接使用是不行的。它和很多有用的函数在一起，可以在模块中找到。

模块
Modules

你可以把模块想像成导入到Python内增加其能力的扩展。你可以使用特殊的命令import导入模块。前部分中我们需要的函数floor就在一个叫做math的模块中：

```
>>> import math
>>> math.floor(32.9)
32.0
```

注意它是怎么起作用的：我们是用import导入了模块，然后按照模块.函数的格式使用这个模块的函数。
如果你想要年龄化为整数（32）而不是浮点数（32.0），你可以使用int函数[7]。

```
>>> int(math.floor(32.9))
```

■**注意** 也有类似的转换到其它类型的函数（比如long和float）。事实上，它们并不是完全是普通函数——他们是**类型对象**（type object）。以后我会对类型对象详述。相对于floor的函数是ceil（ceiling（天花板）的简写），可以将所给数值转换成为大于它的最小的整数。

如果你肯定自己不会导入一个以上的确定名字的函数（对于不同模块来说），你可能并不想每次在调用函数时候还要写上模块名字。你可以使用import命令的一个变形：

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

在使用了from模块import函数之后，你可以不用模块前缀就使用函数。

■**提示** 事实上，你可以使用变量来关联函数（或者Python之中大多数事物）。比如你可以使用foo=math.sqrt进行赋值，然后使用foo来计算平方根：

foo(4)生成2。

---

7. int函数/类型在转换成正是时候就自动截取了小数，所以转换成整数时，math.floor是多余的，你可以直接用int(32.9)

cmath和复数
cmath and Complex Numbers cmath

sqrt函数用于计算一个数的平方根。让我们看看如果给它个负数会如何：

```
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent  call last):
          File "<pyshell#23>",  line 1, in ?
        sqrt(-1)
        ValueError: math domain error
```

还是可以理解的。你不能求负数的平方根——真的不可以吗？当然可以：负数的平方根是虚数（这是标准的数学概念——如果你感觉有些绕不过弯来，跳过即可）。那么你为什么不能使用sqrt？因为它只能处理浮点数，而虚数（以及复数，为实数和虚数之和）是完全不同的——这也是为什么它们被放在另外一个叫做cmath（即complex math，复数）的模块内。

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

注意，我没有使用from...import...语句。如果我用了，那么我就会丢失普通的sqrt函数。这类名字冲突很讨厌，所以除非你真想使用the from version，你应该坚持使用普通的import。

1j是个虚数，虚数均以j（或者J）结尾，就像长整型数使用L一样。我们不深究复数的理论，让我们看看具体的使用方法：

```
>>> (1+3j) *  (9+4j)
(-3+31j)
```

可以看到，对于复数的支持是内建在语言中的。

■**注意** Python中没有单独的叙述类型。它们被当作实数部分为0的复数。

__回到__future__
Back to the _future_

有传言说Guido van Rossum（Python之父）拥有时间机器，因为当有人请求这门语言的新特性时候，这个特性就已经实现了。当然，我们是不允许进入这架时间机器的，但是Guido已经将它以魔力模块__future__的形式建立为Python的一部分。通过它我们可以导入目前没有，但是在以后会成为标准Python一部分的特性。你已经在数和表达式部分见识过了，而你在这本书内也会一次又一次的看到。

保存和执行你的程序
Saving and Executing Your Programs

交互式解释器是Python的强项之一。它可以让你实时测试解法和试验这门语言。如果你想知道某些语句如何用，那么就试试看吧！但是，你在解释器里面输入的一切都会在退出的时候丢失。你真正想做的事情是写你和其他人能运行的程序。本部分里面，你会学到如何去做。

首先，你需要一个文字编辑器，最好是专门用来编程的（如果你使用Microsoft Word，保证你的代码以纯文本保存）。如果你已经在用IDLE，那很幸运：用File->New Window方式建

立个新编辑器就好了。会出现另外一个窗口——没有交互式提示符，哇！

```
print "Hello, world!"
```

现在选择File->Save保存程序（其实就是纯文本）。保证放在你以后能找到的地方。你可能会专门建立一个放Python工程的文件夹，比如Windows系统的C:\Python。UNIX环境下，你可能会建立一个类似~/python的文件夹。给你的文件起个有意义的名字，比如hello.py。以.py为扩展名很重要。

■**注意** 如果你照着本章前面的安装指导进行了安装，那么你已经把Python安装在了~/python，但是因为它会自己建立子文件夹（比如~/python/Python-2.4），那么也就不会出现问题了。如果你想要把文件放在自己需要的地方，那么放心的放在诸如~/my python programs之类的文件夹。

搞定了？先别关闭有你程序的窗口，如果已经关了，那么再打开（File->Open）。现在你应该能用Edit->Run来运行，或者按下Ctrl+F5键（如果你没有使用IDLE，请查看下一部分，关于如何在命令提示符下运行你的程序）。

发生了什么？Hello, world!被打印在解释器窗口内，这就是我们想要的。解释器提示符没了，但是你可以按下回车召它回来（在交互器窗口）。

```
name = raw_input("What    is your  name? ") print
"Hello, " + name    + "!"
```

■**注意** 不要担心input和raw_input的区别——我会讲到的。

如果你运行这个程序（记得先保存），你应该会在解释器窗口看到下面的提示：

```
What is your    name?
```

输入你的名字（比如Gumby）然后按下回车。应该会看到：

```
Hello, Gumby!
```

有趣，不是吗？

利用命令提示符运行你的Python代码
Running Your Python Scripts from a Command Prompt

事实上，运行你程序的方法有很多。首先让我假设你的Windows有DOS，或者有UNIX下的提示符，而且Python的可执行文件（Windows用python.exe，UNIX使用python）已经放置在环境变量的PATH中[8]。而且你在前例中的代码已经在当前文件夹内了。那么你可以在Windows下使用这个命令运行你的代码：

```
C:\>python hello.py
```

或者在UNIX下：

```
$ python hello.py
```

可以看到，命令是一样的，仅仅是系统提示符不同。

■**注意** 如果你不想跟什么环境变量打交道，你可以直接在解释器里面明确python的完整路径。Windows下面应该类似：

```
C:\>C:\Python24\python hello.py
```

让你的代码像正常程序一样
Making Your Scripts Behave Like Normal Programs

有些时候你希望像运行其他程序（比如浏览器、文本编辑器）一样运行Python程序（也叫做代码）。UNIX下面有个标准的方法：就是在代码首行前面加上#!（叫做poundbang或者shebang），然后是解释代码的程序的完整路径（我们用Python）。甚至说如果你不太明白的话，直接把下面那行放在自己代码的第一行即可：

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

这样就会运行程序，不管Python在哪里。

■**注意** 有些系统内如果你安装了最新版本的Python（比如2.4）的同时还有旧版的（比如1.5.2），有些程序需要所以不能卸载。这种情况下/usr/bin/env技巧就不好用了，旧版Python可能会去运行你的程序。你需要找到新版本Python可执行文件（可能叫做python或者python2）的具体位置，然后用完整路径进行替换，比如：

```
#/usr/bin/python2
```

在你运行之前，必须让它可执行：

```
$ chmod a+x hello.py
```

现在就能这样运行了（假设你的路径中包含有当前文件夹）。

```
$ hello.py
```

8. 如果你不太明白这个句子的意思，或许你应该跳过部分。你不是真的需要这部分知识。

■**注意** 如果不起作用的话，试试./hello.py，就算你的当前文件夹不在可执行路径内也可以。

如果你喜欢，你可以重命名你的文件，然后去掉py扩展名让它看起来像个一般程序一样。

双击怎么样？
What About Double-Clicking?

Windows系统下，扩展名.py可以让你的代码运行的像普通程序一样。试着双击你在上一部分存好的
hello.py。如果Python安装正确，会出现一个DOS窗口，里面有“What is your name”的提示，很酷吧？（你可以让你的程序看起来更棒，比如有按钮、菜单，后面会提到）。

你的程序运行时候可能有个问题。一旦你输入了你的名字，程序窗口在你看到结果前就关闭了。程序运行完毕窗口也就关闭。试着改改你的代码，在最后加上这行：

```
raw_input("Press <enter>")
```

现在运行程序输入名字，会出现个有下面内容的DOS窗口：

```
What is your    name? Gumby
Hello, Gumby! Press
<enter>
```

一旦你按下回车，窗口关闭（因为程序运行结束）。现在把你的文件改名为
hello.pyw（Window专用），像刚才一样双击。发生了什么？什么都没有！怎么会呢？本书后面我会告诉你——我保证。

注释
Comments

井号符号（#）在Python内有些特殊。当你在代码中输入它的时候，它右面的一切都会被忽略（这也是刚才为什么解释器不会在/usr/bin/env行“憋住”的原因），比如：

```
# Print the circumference of the    circle:
print 2 * pi    * radius
```

这里的第一行被称为注释（comment），在程序设计时候会让程序更易懂——对其他人和对你自己重看旧代码都一样。据说程序员的第一戒律就是“汝应注释（Thou Shalt Comment）”（尽管很多没良心的程序员的座右铭是“如果难写，就该难读”）。保证你的注释说的都是重要事情，不要把代码上一眼能看出的东西重说一遍。无用的、多余的代码还不如没有。举例来说，下例中的注释就不好：

```
# Get the user's name:
user_name = raw_input("What is your name?")
```

让你的代码易读永远是个好主意，就算是没注释也一样。幸运的是，Python在
书写易读的程序方面是个出色的语言。

字符串

那么，raw_input和"Hello,"+name+"!"这些到底什么意思？我们按下raw_input这部分暂且不表，先说"Hello"这部分。

```
print "Hello, world!"
```

在编程教程中，习惯上都会以这样一个程序开始——问题就是我还没有真正的解释它是怎么工作的。不过你已经了解了关于print语句的基本知识（随后我会介绍更多），但是"Hello, world!"呢？它被称为“字符串”（String）。字符串几乎在所有的可用的、现实世界的Python程序中都会存在，并且有多种用法，也是显示类似感叹句"Hello, world!"这样字符的主要因素。

单引号字符串和转义引用
Single-Quoted Strings and Escaping Quotes

字符串是值，就像数字一样：

```
>>> "Hello, world!"
'Hello, world!'
```

本例中有一个地方可能会让你小吃一惊：当Python打印出我们的字符串时候，是用单引号括起来的，而我们用的是双引号。有什么区别吗？事实上，没有区别。

```
>>> 'Hello, world!'
'Hello, world!'
```

这里我们也用了单引号，结果是一样的。那么为什么两个都可以用？因为有些时候会派上用场：

```
>>> "Let's go!"
"Let's go!"
>>> '"Hello,  world!" she said'
'"Hello, world!" she said'
```

上面的代码中，第一段字符串包含了单引号（或者叫做省略符号，我们这里也称为单引号），这时候我们就不能用单引号括起来。如果我们这么做了，那么解释器会抱怨道（也是恰当的）：

```
>>> 'Let's go!'
SyntaxError: invalid  syntax
```

在这里字符串为'Let'，Python不太明白后面的s（以及再后面的一串文字）是什么意思。

第二个字符串中，我们使用了双引号，所以就需要用单引号把字符串括起来，原因和刚才说的一样。或者
我们并不**一定**要这样。另外一个选择就是使用反斜线（\）来在字符串中进行转义：

```
>>> 'Let\'s go!'
```

"Let's go!"

Python会明白中间的单引号是个在字符串**中**的字符，而不是字符串的**结束**（就算是这样，Python也会在打印出字符串的时候使用双引号）。双引号中也是一样，你可能会希望像这样：

```
>>> "\"Hello, world!\"    she said"
'"Hello, world!" she said'
```

这样的转义引用很有用，有些时候还是必须。举例来说，如果你想打印一个包含单双引号的字符串，不用反斜线的话你能怎么办？比如字符串'Let\'s say "Hello, world!"'？

■**注意** 厌烦了反斜线？本章后面你会看到，你可以用长字符串和自然字符串（可以联合使用）避免绝大多数的反斜线。

字符串的连接
Concatenating Strings

继续我们刚才的例子，我给你看个写同样字符串的不同方法：

```
>>> "Let's say " '"Hello, world!"'
'Let\'s say "Hello, world!"'
```

我只是写了两个字符串而已，一个跟在另一个后面。Python自动连接了它们（合为一个字符串）。这个机制用的不多，但是有时非常有用。不过它仅仅你同时写下两个字符串时候才有用，要一个紧接着另一个：

```
>>> x = "Hello, "
>>> y = "world!"
>>> x y
SyntaxError: invalid   syntax
```

换句话说，这仅仅是书写字符串的一个特殊方式，并不是连接字符串的一般方法。但是，你怎么连接字符串呢？就像加数字一样：

```
>>> "Hello, " + "world!"
'Hello, world!'
>>> x = "Hello, "
>>> y = "world!"
>>> x + y
'Hello, world!'
```

字符串表达，str和repr
String Representations, str and repr

通过前面的例子你可能注意到了，所有通过Python打印的字符串还是被引号括起

来的。这是因为Python打印值的时候会保持在Python代码中的状态，而不是你希望用户所看到的。如果你使用print语句，结果就不一样了：

```
>>> "Hello, world!"
'Hello, world!'
>>> 10000L
10000L
>>> print "Hello, world!"
Hello, world!
>>> print 10000L
10000
```

你可以看到，长整数10000L被转换成了数10000，在显示给用户的时候也如此显示。但是当你想知道一个变量的值是多少，你可能会对它是整型还是长整型有兴趣。

我们讨论的实际上是值被转换为字符串的两种机制。两种你都可以自己使用，通过str函数，它会把值转换为用户可以理解的合理形式的字符串。而repr会创建同合法的Python表达式一样的字符串的表现形式[9]。

```
>>> print repr("Hello,   world!")
'Hello, world!'
>>> print repr(10000L)
10000L
>>> print str("Hello, world!")
Hello, world!
>>> print str(10000L)
10000
```

repr(x)的功能可以用`x`实现（注意，`是反引号，而不是单引号）。当你想打印一个包含数字的句子时候会很有用，比如：

```
>>> temp = 42
>>> print "The temperature is " + temp
  Traceback (most recent   call last):
    File "<pyshell#61>",   line 1, in ?
      print "The temperature is " + temp
  TypeError: cannot add type "int"   to string
>>> print "The temperature is " + `temp` The
temperature is 42
```

第一个语句并不起作用，因为你不能把字符串加到数字上。第二个则工作正常，因为我已经把temp的值用反引号转换为字符串“42”。

9. 事实上，str和int、long一样，是一种类型。而repr仅仅是函数。

（当然，我也可以使用repr，也能得到同样结果，但是这样可能更清楚一些。事实上，本例中，我也可以使用str。现在对此不用过多担心）

简单来说：str、repr和反引号是转换Python值到字符串的三种方式。函数str让字符串可看，repr（和反引号）试图让结果字符串转换为合法的Python表达式。

input vs. raw_input

现在你已经知道"Hello, " + name + "!"是什么意思了。但是raw_input呢？input函数不够好吗？让我们试试看。在另外一个代码文件中输入下面语句：

```
name = input("What is your name? ")
print "Hello, " + name + "!"
```

完美且合法的程序，但是你不久就会发现，它有点不太实际，试试看：

```
What is your name? Gumby
Traceback (most recent call last):
  File "C:/python/test.py", line 2, in ?
    name = input("What is your name? ")
  File "<string>", line 0, in ?
    NameError: name 'Gumby' is not defined
```

问题在于input会假设你输入的是合法的Python表达式（或多或少有些repr逆过来的意思）。如果你以字符串作为输入的名字，没有问题：

```
What is your name? "Gumby"
Hello, Gumby!
```

但是要求用户带着引号输入他或者她的名字有点过分，那么我就要用raw_input了，它会把所有的输入当作自然数据（raw data），然后将其转换为字符串：

```
>>> input("Enter a number: ") Enter a
number: 3
3
>>> raw_input("Enter a number: ") Enter a
number: 3
'3'
```

除非你对input有特殊需要，你应该使用raw_input。

长字符串、自然字符串和Unicode
Long Strings, Raw Strings, and Unicode

在结束本章之前，我会在告诉你一些其他的书写字符串的方法。这些“替补”的字符串语

法在你需要多行字符串、或者有特殊字符的字符串时候非常有用。

长字符串

如果你想要写一个非常非常长的字符串，需要跨多行，你可以使用三引号代替普通引号。

```
print '''This is a very long string. It
continues here.
And it's not  over yet.
"Hello, world!"
Still here.'''
```

你也可以使用三双引号，"""像这样"""。注意，因为这种与众不同的引用方式，字符串之中可以包含单引号和双引号，而不是反斜线进行转义。

■**提示** 普通字符串也可以跨行。如果一行之中最后一个字符是反斜线，那么这行就会截断自己“转义”[10]，然后忽略，比如：

```
print "Hello, \
world!"
```

这句会打印`Hello, world!`。对于表达式和语句也是一样：

```
>>> 1 + 2 + \
        4 + 5
12
>>> print \
        'Hello, world'
Hello, world
```

自然字符串
Raw Strings

自然字符串对于反斜线不会吹毛求疵，有些时候这就派上用场了。对于普通字符串来说，反斜线有特殊身份：它会**转义**，可以让你在字符串中加入不能直接加入的东西。比如新行符号写为\n，可以放在字符串中：

```
>>> print 'Hello,\nworld!'
Hello,
world!
```

看起来不错，但是有些时候不会如你所愿。如果你想要在字符串里面包含反斜线加上n怎么办？你可能会想打印DOS路径“C:\nowhere”为字符串，比如：

---

10. 尤其是在书写正则表达式时候。第十章会有更多信息。

```
>>> path = 'C:\nowhere'
>>> path
'C:\nowhere'
```

看起来正确，直到你打印的时候：

```
>>> print path
C:
owhere
```

不是我们想要的，不是吗？那么怎么办？我可以对反斜线本身进行转义：

```
>>> print 'C:\\nowhere'
C:\nowhere
```

看起来不错，但是对于长路径，你可能会需要很多反斜线：

```
path = 'C:\\Program Files\\fnord\\foo\\bar\\baz\\frozz\\bozz'
```

自然字符串在这些时候就派上用场了。它们不会把反斜线作为特殊字符对待。每个你作为自然字符串输入的字符都会按照你写的方式保持原样：

```
>>> print r'C:\nowhere'
C:\nowhere
>>> print r'C:\Program    Files\fnord\foo\bar\baz\frozz\bozz'
C:\Program Files\fnord\foo\bar\baz\frozz\bozz
```

你可以看到，自然字符串以r开头。看起来你可以在自然字符串里面放任何东西，这么说是基本正确的。当然，引号也要像平常一样被转义，意味着你需要在最终字符串里面加上反斜线：

```
>>> print r'Let\'s go!'
Let\'s go!
```

你在自然字符串里面不能输入的是结尾的反斜线。换句话说，自然字符串的结尾不能是反斜线。继续上例，看起来很明显：如果最后一个字符串（在结束引号前的那个）是个反斜线，Python就会不清楚是否结束字符串：

```
>>> print r"This is illegal\"
SyntaxError: invalid    token
```

好了，这样还是合理的。但是如果你需要最后一个字符是反斜线怎么办？（有可能是DOS路径的最后一个字符）。那么，我会在这里告诉你一个解决问题的技巧，基本上来说就是你需要把反斜线作为单独的字符串。简单的方法类似于：

```
>>> print r'C:\Program    Files\foo\bar' '\\'
C:\Program Files\foo\bar\
```

注意看，你现在可以同时使用单双引号以及自然字符串。就算是三引号字符串也能是自然字符串。

Unicode字符串

Unicode Strings

字符串常量的最后一个类型就是Unicode字符串（或者为Unicode对象——和字符串不同属一类）。如果你不知道什么是Unicode，那么你也不用了解这些（如果你希望得到更多信息，可以去Unicode的网站，http://www.unicode.org）。Python之中的普通字符串都在内部存储为9位的ASCII码，Unicode则存储为16位Unicode字符。这样就允许字符串种类更多，包括世界上大多数语言的特殊字符。我将前面的字符串重新限定为Unicode字符串：

```
>>> u'Hello,  world!'
u'Hello, world!'
```

可以看到，Unicode字符串使用u前缀，就像自然字符串使用r一样。

快速总结
A Quick Summary

本章的内容非常具体。让我们看看你在继续下一章之前都学到了什么：

算法。算法是告诉你具体做什么的菜谱。当你编程时，你基本上就是在以计算机能懂得的语言书写算法，比如Python。这类对机器友好的描述就叫做程序（program），包含有表达式（expression）和语句（statement）。

表达式。表达式是计算机程序的一部分，它用于表现值。举例来所，2+2是表达式，表现出值4。简单的表达式用文字值建立（比如2或者"Hello"），使用运算符（operator）（比如+或者%）和函数（function）（比如pow）。更复杂的表达式可以由联合的简单表达式建立。（比如 (2+2)*(3-1)）表达式也可以包含变量。

变量。变量是表现值的名字。新的值可以通过类似x=2的赋值（assignment）赋予变量。赋值也是一类语句。

语句。语句是告诉计算机**做**什么事情的指令。它可能包含改变变量（通过赋值）、向屏幕打印东西（比如print "Hello, world!"）、导入模块或者其他的事情。

函数。Python中的函数就像数学中的函数：它们可以带有参数，并且返回值（在返回值前可以做很多有趣的事情）。你可以在第六章里面学习如何书写自己的函数。

模块。模块是可以导入到Python内、延展其功能的扩展。比如很多有用的数学函数都在math模块中。

程序。你已经见过如何书写、保存和运行Python程序了。

字符串。字符串非常简单——就是一块块的文本。字符串有很多相关知识。本章内你已经知书写他们的很多方法，第三章内还有很多使用它们的方式。

本章内的新函数
New Functions in This Chapter

函数	描述
abs(number)	返回数的绝对值
cmath.sqrt(number)	开平方根，负数可用
float(object)	转换字符串和数为浮点数
help()	提供交互式帮助
input(prompt)	获取用户输入
int(object)	转换字符串和数为整数
long(object)	转换字符串和数为长整数
math.ceil(number)	以浮点数值返回数的下舍整数
math.floor(number)	以浮点数值返回数的上入整数
float math.sqrt(number)	平方根，不适用于负数
pow(x, y[, z])	x的y次方（模z） raw_input(prompt)
	以字符串获取用户输入 repr(object)
	以字符串形式返回值
round(number[, ndigits])	四舍五入
str(object)	转换值为字符串

现在学什么？
What Now?

现在你已经掌握了表达式的基础知识，让我们深入一点：数据结构。除了和简单值（比如数）打交道外，你还能将它们联合存储在复杂的结构中，比如列表（list）和字典（dictionary）。另外，你还会深入字符串。第五章中，你会学习更多关于语句的知识。之后你就能写你自己的程序了。

第二章

列表和元组
Lists and Tuples

本章会介绍一个新的内容：数据结构（*data structures*.）。数据结构是一些数据元素的集合（比如数字或者字母——甚至可以是其它的数据结构）。数据结构通过一些方法构成，比如，通过量化特地的元素。在Python中最基本的数据结构是序列（*sequence*）。序列中的每个元素被分配一个数字——元素的位置，也叫索引。第一个索引是0，第二个索引是1，如此递推。

■**注意** 当你在日常生活中数数时，你可能从1开始数。Python 中使用的编码制可能看起来很奇怪，但这种方法实际上更正常。这么做的原因之一是：正如你在本章后面将要看到的你能从最后的元素开始计数；序列中的最后一个条目被标记为 -1，倒数第二是-2，如此递推。这就意味着你能从第一个元素向前数或者向后数。相信我，你会习惯这样的。

Python有六个内置的序列类型，但让我们集中注意力于其中的最常用的——列表和元组。这两种类型的主要区别是你能改变一个列表但你不能改变一个元组。这意味着如果你要根据你的要求来添加元素，列表可能会更有用；当你出于某些原因而不能让序列改变时元组很适合使用。后者的理由经常是技术性的的，为了满足Python内部的运行方式。这就是为什么你可能看到内置的函数返回元组。在你自己的程序中，几乎在所有的情况下你能使用列表来替代元组（唯一的一个需要注意的例外将在第4章把元组作为字典的键来使用时讨论。在那里列表不被允许，因为你不能修改键）

■**注意** 其他的内置序列类型是字符串（我将在下一章再次讨论），Unicode字符串（Unicode strings），buffer对象和xrange对象。

序列在你要用一个值的集合时很有用。你可能用一个序列代表数据库中的一个人员信息的条目——这个条目的第一个字段是姓名，第二个字段是年龄。写一个列表（组成列表的条目被逗号分隔，并且被包含在方括号中）就像这样：

```
>>> edward =   ['Edward Gumby', 42]
```

但序列也能包含其他的序列，因此你能生成如下的一个关于人员信息的列表，列表的结果

是database：

```
>>> edward =   ['Edward Gumby', 42]
>>> john = ['John Smith', 50]
>>> database   = [edward, john]
>>> database
[['Edward Gumby', 42],   ['John Smith', 50]]
```

这章会用一些对所有的序列（包括列表和元组）都通用的操作来开始。这些操作同样对字符串（会被一些例子中使用的）有效，尽管你必须等到下一章你才能看到对字符串操作的全面讨论。

在处理完基本的东西后我们开始使用列表，看看有什么新的特性。在讨论完列表后，我们接着讨论元组，元组和列表很像，除了你不能改变它们。

■**注意** Python有一种数据结构叫做容器（container），这是一个基本的概念，它意味着任何一个对象能包含其他的对象。有两种主要的容器，一种是序列（就像列表和元组），一种是映射（mappings，就像字典 dictionaries）。两者的区别是：序列中的每个元素都被编号而映射中的每个元素则有一个名字（也叫键 key）。在第四章你会学到更多的关于映射的知识。对于一个既不是序列也不是映射的容器类型的例子，请看第10章对集合（sets）的讨论。

序列的共有操作

Common Sequence Operations

有一些操作可以对所有的序列用。这些操作包括：索引（indexing），切片（slicing），加（adding），乘（multiplying）以及对成员数量的检查。此外，Python已经

■**注意** 迭代（iteration）作为一个重要的操作没有被包含在这里。在一个序列上迭代的意思是对序列中的元素重复的做出一些操作。可以看特定的章节了解更多的信息。

索引

Indexing

在序列中的所有元素都被编号----从零开始开始。你能通过编号来访问那些元素，就像：

```
>>> greeting   = 'Hello'
>>> greeting[0]
'H'
```

■**注意** 一个字符串就是一个由字符组成的序列。编号0引用的是第一个元素，在这个例子中

是字母H。

这就是索引---你为每个元素编制索引。所有序列都能通过这种方式被索引。当你使用一个负数索引时，Python的语法是从右边开始计数，也就是从最后一个元素开始。最后一个元素的位置编号是-1（不是-0，因为那会和第一个元素重合）

```
>>> greeting[-1]
'o'
```

字符串能够直接使用索引，而不需要一个变量引用它们。作用是一样的。

```
>>> 'Hello'[1]
'e'
```

如果一个函数调用返回一个序列，你能对它使用索引。例如，如果你只是简单的的对用户输入的年份的第四个数字感兴趣，你能像这么做：

```
>>> fourth =   raw_input('Year: ')[3] Year: 2005
>>> fourth
'5'
```

范例

列表2-1包含了一个例子程序，程序要求你输入年月（1到12）日（1到31），然后打印出相应的月份名称等等。程序的一个会话例子可能如下：

```
Year: 1974
Month (1-12): 8
Day (1-31): 16
August 16th,   1974
```

最后一行是程序的输出。

Liesting2-1 索引示例

```
# Print out a date, given year, month, and   day as numbers

months = [
    'January',
    'February',
    'March',
    'April',
    'May',
    'June',
```

```
    'July',
    'August',
    'September',
    'October',
    'November',
    'December'
]


# A list with one ending for each number from 1 to 31
endings = ['st', 'nd',   'rd'] + 17 * ['th']   \
          + ['st', 'nd',   'rd'] +   7 * ['th']   \
          + ['st']


year       = raw_input('Year: ')
month      = raw_input('Month (1-12): ')
day        = raw_input('Day (1-31):
') month_number =   int(month)
day_number =   int(day)

# Remember to subtract   1 from month and day to get a
correct index month_name =   months[month_number-1]
ordinal = day + endings[day_number-1]


print month_name + ' '   + ordinal   + ', ' + year
```

切片
Slicing

就像你用索引来访问单个的元素，你能使用切片来访问一定范围的元素。你通过被分号隔开的两个索引号来实现切片。

```
>>> tag = '<a href="http://www.pythonorg">Python web site</a>'
>>> tag[9:30]
'http://www.Python.org'
>>> tag[32:-4]
'Python web site'
```

就像你能看到的，切片对于提取一个序列的一部分很有用。量化在这里显得很重要。第一个索引号是你要提取的元素中的第一个元素的编号。然而，最后的索引号是你切片后剩下

的第一个元素的编号。参考如下的代码：

```
>>> numbers = [1, 2, 3, 4, 5, 6,  7, 8, 9, 10]
>>> numbers[3:6]
[4, 5, 6]
>>> numbers[0:1]
[1]
```

总体来讲，你要为你的切片提供两个索引号作为限制，其中第一个是被包含在切片内的，第二个是在切片外的。

捷径
A Nifty Shortcut

假设你要访问最后的三个元素（根据先前的例子）。你能很清楚做到：

```
>>> numbers[7:10]
[8, 9, 10]
```

现在，索引号10指向的是11号元素----并不存在此元素，但它却是你所要的最后一个元素后面的第一个元素，懂了么？

```
>>> numbers[-3:-1]
[8, 9]
```

这样看起来你也能按这种方式访问最后的元素。那么使用索引号为0来切片会怎么样？是一样的效果吗？

```
>>> numbers[-3:0]
[]
```

不是设想的结果，实际上，只要切片中左边的索引号如果比右边的靠后（在这个例子中是倒数第三个比第一个靠后），那么结果就是一个空的序列。幸运的是，你能使用一个技巧：如果切片持续到序列的结尾，你可以简单的置空后一个索引号（leave out the last index）：

```
>>> numbers[-3:]
[8, 9, 10]
```

这种方法同样适应于从开始切片。

```
>>> numbers[:3]
[1, 2, 3]
```

实际上，如果你复制整个序列，你可以置空两个索引号。

```
>>> numbers[:]
[1, 2, 3, 4,   5, 6, 7, 8, 9, 10]
```

范例

Listing2-2 包含一个提示你输入URL的小程序，（假设它的形式像是 http:// www.somedomainname.com）然后提取域名。这里有个程序运行的示例：

```
Please enter   the URL: http://www.pythonorg
Domain name:   python
```

Listing2-2 切片示例：

```
# Split up a   URL of the form http://www.something.com

url = raw_input('Please enter the URL: ')
domain = url[11:-4]

print "Domain name: " + domain
```

更深的了解

Longer Steps

切片时，你指定（不管是直接还是间接）切片的开始和结束点。其他的参数（在python.3中被加入内置类型）----经常被暗中丢弃，指出步长（step length）。在一个正常的切片中，步长指的是从一个元素移动到下一个元素的过程中开始和结束点之间的元素个数。

```
>>> numbers[0:10:1]
[1, 2, 3, 4,   5, 6, 7, 8, 9, 10]
```

在这个例子中，你能看到切片中包含了其他的数字。也许你已经猜测出来了，这就步长的明确表达。如果步长值被设置为比1大，那么有的元素就会被跳过，比如，步长为2的切片将只是包含从开始点到结束点的所有元素中的间隔的元素。

```
>>> numbers[0:10:2]
[1, 3, 5, 7,   9]
numbers[3:6:3]
```

```
[4]
```

你能继续使用先前提及的技巧；如果你需要一个间隔4个元素的切片，你只要设置步长为4：

```
>>> numbers[::4]
    [1, 5, 9]
```

自然地，步长不能为0---那不会有什么结果---但步长能为**负数**，这就意味着从右到左提取元素。

```
>>> numbers[8:3:-1]
[9, 8, 7, 6,  5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
>>> numbers[0:10:-2]
[]
>>> numbers[::-2]
[10, 8, 6, 4, 2]
>>> numbers[5::-2]
[6, 4, 2]
>>> numbers[:5:-2]
[10, 8]
```

在这里要使切片的结果正确需要动些脑筋。正如你所看到的，开始点的元素（最左边元素）是被包含的，而结束点的元素（最右边的元素）是在切片之外的。当使用一个负数作为步长，你必须让开始点高于结束点。这可能会在你不明确指定开始和结束点时带来一些混淆，python这种情况下会做正确的事；对于一个正值步长，是从开始点向结束点移动，对于负值步长则是从结束点向开始点移动。

增加序列
Adding Sequences

序列能够用加号连接:

```
>>> [1, 2, 3] + [4, 5,  6]
[1, 2, 3, 4,  5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> [1, 2, 3] + 'world!'
Traceback (innermost
```

```
last):
  File "<pyshell#2>", line 1, in  ?
    [1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "string") to list
```

正如你能在错误信息里看到的,你不能把一个列表和字符串连接在一起,尽管它们都是序列.总体来说,你只能连接两种相同类型的序列。

乘法
Multiplication

用一个数字----X，去乘以一个序列能产生一个新的序列，在新的序列中原来的序列被重复X次

```
>>> 'python  * 5
'pythonPythonPythonPythonPython
>>> [42] * 10
[42, 42, 42,  42, 42, 42, 42, 42,  42, 42]
```

None,空列表和初始化
None, Empty Lists, and Initialization

一个空的列表的形式是两个中括号（[]）——中间没有东西。但如果你想创建一个占了十个元素空间但却不包含任何有用的东西的列表，那又怎么办呢？你能像前面那样使用[42]*10，或者更实际地用[0]*10。如果这么做你就得到了一个包含十个零的列表。然而，有的时候你可能想要一个值来代表空值（nothing）——意味着你没有放置任何东西在里面。这个时候你就需要使用None，None 是一个python的值，它的确切意思是：“这里什么也没有”。所以如果你想初始化一个长度为10的列表，你可以按下面的做：

```
>>> sequence  = [None] * 10
>>> sequence
[None, None,  None, None, None, None, None,  None, None, None]
```

列表2—3包含一个向屏幕打印一个由字符组成的盒子。输出的内容在屏幕上居中而且能根据用户输入的句子自动调整。下面是例子的运行情况：

Sentence: He's a very naughty boy!

```
+————————————————————
————————————————————
———————+
```

```
                    |
                    |
| He's a very naughty boy!
                    |
                    |
                    |

          +————————————————————
          ————————————————————
                    ————+
```

代码可能看起来很复杂，但它的复杂只是基于算法——计算出有多少空格，破折号，然后你需要把那些放置到合适的地方。

Listing 2-3. *Sequence (String) Multiplication Example*

```
# Prints a sentence in  a centered "box" of  correct width


# Note that the integer division  operator (//) only works in python
# 2.2 and newer. In earlier versions, simply use plain division (/)


sentence = raw_input("Sentence: ")


screen_width =  80
text_width  = len(sentence)
box_width  = text_width + 6
left_margin    = (screen_width - box_width) // 2


print
print ' ' * left_margin + '+'    + '-' * (box_width-2)  +    '+'
print ' ' * left_margin + '|   ' + ' ' * text_width                +  '
|' print ' ' * left_margin + '|   ' +          sentence          +  '
|' print ' ' * left_margin + '|   ' + ' ' * text_width                +  '
|' print ' ' * left_margin + '+'    + '-' * (box_width-2)  +    '+'
print
```

成员关系
Membership

为了查看一个值是否在序列中你需要使用in运算符。这个运算符和之前已经讨论过的运算符（比如+， * ）有一点不同。这个运算符查看是不是真值（true），然后根据是true还是

false返回 true 或者false。这样的运算符叫做布尔运算符（Boolean operatior），真值被叫做布尔值Boolean values）你会在第五章的条件语句部分了解到更多的关于布尔语句的内容

```
>>> permissions = 'rw'
>>> 'w' in permissions
True
>>> 'x' in permissions
False
>>> users = ['mlh', 'foo', 'bar']
>>> raw_input('Enter your user name: ') in    users
Enter your user name: mlh
True
>>> subject = '$$$ Get    rich now!!! $$$'
>>> '$$$' in    subject
True
```

头两个例子使用了成员关系测试来查看是否‘W’和‘X’能在字符串permissions中单独被找到。这能在UNIX机器中作为一个查看文件可写和可执行权限的脚本。下一个例子查看一个被替换的用户名（mln）是否在一个关于用户的列表中。如果你的程序要加强一些安全策略这个就很有用。（在这种情况下你可能也想要得密码）最后一个例子能成为垃圾邮件过滤器的一部分，比如：它查看是否字符串对象包含字符窜‘$$$’。

■**注意** 最后一个例子不同于其他的例子。总体来说，in运算符查看一个对象是否是一个序列（或者是其他的数据集合（collection））的成员（元素）。然而一个字符串唯一的成员或者元素就是它的字符。因此，下面的例子就展示了完整的内涵：

```
>>> 'P' in
'python True
```

实际上，在python早期的版本中这个市是唯一能被用于字符串成员关系查看的——查看一个字符是否在一个字符串中。如果你尝试去查看一个比字符更长的子字符串比如‘$$$’你会得到一个错误信息（内容是类型错误）。要实现这个功能你必须使用一个字符串方法。你将在第三章了解更多。但从python .3开始，你能使用in 运算符来实现。

列表 2 — 4 展示了一个去数据库（实际上是一个列表）查看用户输入的用户名和PIN码是否存在的程序。

用户名和PIN如果在数据库中存在那么就打印'Access granted'。（if语句在第一章被提及并将在第五章被全面描述）

Listing 2-4. *Sequence Membership Example*

```
# Check a user name and PIN code

database = [
    ['albert',   '1234'],
    ['dilbert', '4242'],
    ['smith',    '7524'],
    ['jones',    '9843']
]

username = raw_input('User name:   ')
pin = raw_input('PIN code: ')

if   [username, pin] in database: print 'Access granted'
```

长度，最小值和最大值
Length, Minimum, and Maximum

内部函数len，min和max非常的有用。Len函数 返回序列中包含的元素的数量，min和max则分别返回序列中最大和最小的元素 （你会在第五章的比较运算符部分了解更多的对象比较方面的 内容）。

```
>>> numbers = [100, 34, 678]
>>> len(numbers)
3
>>> max(numbers)
678
>>> min(numbers)
34
>>> max(2, 3)
3
>>> min(9, 3, 2, 5)
2
```

应该对前面的解释进行澄清特别是对最后两个语句，在这里max和min的参数不是一个序列，多个数字作为参数被直接传了进去。

在前面的例子中我使用了很多次列表 。你已经见识了列表有多有用，但这个部分要处理的是让元组和字符串不同于列表和mutable的地方——你能改变后两者的内容并且后两者有更多有用的特殊方法 。

list函数
The list Function

因为字符串不能像列表一样被修改，所以我们经常从一个字符串创建一个列表。你能用list函数[1]做这个工作：

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

注意list函数能用于所有的序列而不单单的是字符串。

■**提示** 你能用下面的语句把一个由字符组成的列表转回为一个字符串：
''.join(somelist) Chapter 3.

在这里，somelist是你需要转换的列表。为了了解这里正真的含义请参考第三章关于join的部分。

基本的列表运算符
Basic List Operations

改变列表：项目赋值
Changing Lists: Item Assignments

你能在列表中使用所有的序列运算符，比如索引，切片，连接和乘法。但有趣的是列表是能被修改的。在这个部分，你能看到改变一个列表的一些方法：元素赋值，元素删除，元素切片以及list方法（请注意并不是所有的list方法都会实际地改变列表）。

改变一个列表是很容易的。你只需使用在第一章提及的普通赋值语句就行了。然而为了代替写一些像x=2之类的语句，你使用索引符号来表示一个单独并且存在的位置，如x[1]=2。

```
>>> x = [1, 1, 1]
>>> x[1] = 2
>>> x
[1, 2, 1]
```

■**注意** 你不能表示一个不存在的位置。如果你的列表长度为2，你不能赋值给索引为100的位置。如果要那样做，就必须建立一个长度为101（或者更多）的列表。请参考本章前面关于“None，空的列表和初始化”的部分 。

---

1. 它实际上是一种类型而不是一个函数，但在这里两者的区别不重要。

删除元素
Deleting Elements

在列表中删除元素也很容易；你能使用del语句来做。

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
>>> del names[2]
>>> names
['Alice', 'Beth', 'Dee-Dee', 'Earl']
```

注意Cecil是如何彻底的消失并且列表的长度从5变到了4。除了删除列表中的元素，del语句还能被用于删除其他的东西。它能被用于字典（请参考第四章）甚至能被用在变量，关于这方面的详细信息请参考第五章。

赋值时切片
Assigning to Slices

切片是一个强大的特征，当你使用赋值时切片则更加强大。

```
>>> name = list('Perl')
>>> name
['P', 'e', 'r', 'l']
>>> name[2:]  = list('ar')
>>> name
['P', 'e', 'a', 'r']
```

你可以一次赋值多个位置。你可能会奇怪这有什么大不了的，我就不能一次一次地赋值吗？当然可以，但是当你使用切片赋值时，你可能用和原序列不等长的序列替换了切片：

```
>>> name = list('Perl')
>>> name[1:]  = list('ython')
>>> name
['P', 'y', 't', 'h', 'o', 'n']
```

切片赋值语句能被用于插入元素而不需要代替任何原来的元素。

```
>>> numbers = [1, 5]
>>> numbers[1:1] = [2,  3, 4]
>>> numbers
```

```
[1, 2, 3, 4,    5]
```

在这里，我只是“替换”了一个空的切片，其实发生的是插入了一个序列。相反的，你能通过切片删除元素。

```
>>> numbers
[1, 2, 3, 4,    5]
>>> numbers[1:4] = []
>>> numbers
[1, 5]
```

正如你可能猜测的，最后一个例子和删除[1:4]的效果一样。

列表的方法
List Methods

你已经遇到过函数，但现在是了解函数密切相关的方法的时候了。

注意 你在第七章能了解到关于这个方法的实质的更多的详细信息。

方法是一个与一些对象有紧密联系的函数，对象可能是一个数字，一个字符串或者其他的什么。总体来说，一个方法能这样被调用：

```
object.method(arguments)
```

就像你看到的，一个方法的调用就像一个函数的调用除了对象被放到了方法名称的前面，并且在两者之间用一个点隔开。列表有许多方法让你检查或者修改其中的内容。

append

append方法被用来追加一个对象到列表的末尾。

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

你可能想知道为什么我选择如此恶心的名字作为我的列表名而不是选择list来作为名字，我的确能那样做但你可能记得list是一个内部函数[2]。如果我用list作为名字我就不能调用那个函数了。你能为一个应用找到更好的名字，一个名字就像lst并没有告诉你任何事。如果你的列表是关于价格的，你就应该使用prices, prices_of_eggs, 或者 pricesOfEggs作为名字。

注意，下面的很重要：append方法和一些其他的方法一样只是在原来的列表上进行修改。

这意味着这些方法不是简单的返回一个新的修改后的列表而是在原来的列表上修改再返回原来的列表，但这有时候会产生问题。在这章的稍后的部分我在说明排序时会回头再看看这个话题。

count

Count方法统计一个列表中的元素的个数。

```
>>> ['to', 'be', 'or',  'not', 'to', 'be'].count('to')
2
>>> x = [[1,  2], 1, 1,  [2, 1, [1, 2]]]
>>> x.count(1)
2
>>> x.count([1, 2])
1
```

extend

extend方法允许你向一个序列一次追加另一个序列中的多个值。换句话说，你原来的列表被其他的列表扩展了。

---

2. 实际上，从python.2开始列表就作为一个类型而不是一个函数。（元组和字符串（str）也是这样） 如果要查看关于这个的整个说明请参见第九章的子类化（Subclassing）列表，字典（dict）和字符串。

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4,  5, 6]
```

这个看起来很像连接，但重要的不同之处是被扩展的列表（在这个例子中是a）是在原来的列表的基础上被修改了，而在连接时则是返回一个新的列表。

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4,  5, 6]
>>> a
```

```
[1, 2, 3]
```

如你所见，使用连接的返回结果和前一个例子中使用extend方法的结果是一样的，然而列表a却没有被修改。这是因为常规的连接操作是创建了一个包含了a和b的新的列表，如果你想的到下面所示的效果使用连接的效率是不能和使用extend相比的。

```
>>> a = a + b
```

同样的，在这里也不是一个原位置操作（in-place operation）—并没有修改原来的。

我们能使用切片来达到extend方法的效率。

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a[len(a):] = b
>>> a
[1, 2, 3, 4,    5, 6]
```

然而这么做，代码的可读性不如extend方法。

Index

index方法被用来在列表中查找一个值第一次出现的索引位置。

```
>>> knights = ['We', 'are', 'the', 'knights', 'who',    'say', 'ni']
>>> knights.index('who')
4
>>>
knights.index('herring')
Traceback (innermost
last):
   File "<pyshell#76>",    line 1, in ?
     knights.index('herring')
ValueError: list.index(x): x not    in list
```

当你搜索单词“who”，你的到的结果是它在索引号为4的位置。

```
 >>> knights[4]
 'who'
```

然而当你搜索“herring”时会得到一个异常（exception），因为这个词没有被找到。

insert

insert方法被用来向列表插入一个对象。

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers.insert(3, 'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

就像使用extend，你能用切片赋值来完成插入。

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers[3:3] = ['four']
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

这有点搞笑，但它的可读性绝对不如insert

pop

pop方法移除列表中的一个元素（默认是最后一个）并且放回元素的值。

```
>>> x = [1, 2, 3]
>>> x.pop()
3
>>> x
[1, 2]
>>> x.pop(0)
1
>>> x
[2]
```

■**注意** pop方法是唯一修改一个列表并且返回一个值（而不是None）的列表方法。

通过使用pop方法，你能实现一个被成为堆栈的常见数据结构。这样的一个堆栈就像堆盘子那样工作。你能在顶部放一个盘子也能顶部拿走一个盘子。最后被放入堆栈的最先被移除（这个原则被称为后进先出，LIFO）

压入（push）和弹出（pop）是被广为接受的两个堆栈操作（放入和移出）的名字。python没有push方法，但你能使用append方法来代替。pop和append方法的操作结果恰好相反，如果你压入（或者追加）你刚刚弹出的值，最后的到的结果还是原来的堆栈。

```
>>> x = [1, 2, 3]
>>> x.append(x.pop())
>>> x
```

```
[1, 2, 3]
```

■**提示** 如果你想要一个先进先出（FIFO）的队列（queue），你能使用inert(0,…)来代替append方法。当然，你也可以继续使用append方法但必须用pop(0)来代替pop()。一个更好的解决方案是使用集合模型（collection module）的双端队列（deque）。详细信息请参见第十章。

---

remove

remove方法被用来移除第一次出现的某个值。

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
['to', 'or',   'not', 'to', 'be']
>>> x.remove('bee')
Traceback (innermost
last):
  File "<pyshell#3>", line 1, in   ?
    x.remove('bee')
ValueError: list.remove(x): x not in list
```

正如你所看到的，只有第一次出现的值被移除了，同时你也没法移除一些（比如“bee”）不存在于列表中的值。值得注意的是这个方法是一个没有返回值的原位置改变的方法。和pop方法相反remove方法修改了列表但什么也不返回。

reverse

reverse方法将列表中的元素倒置存放（我猜你不是很吃惊吧）。

```
>>> x = [1, 2, 3]
>>> x.reverse()
>>> x
[3, 2, 1]
```

请注意该方法也是改变列表但不返回值（就像remove和sort）。

■**提示** 如果你要在一个序列上进行反向迭代（iterate）你能使用reversed函数。这个函数不是返回一个列表而是返回一个迭代器（iterator）（你能在第九章找到更多关于迭代器的内容）尽管如此，你仍然可以把返回的对像转换成一个列表。

```
>>> x = [1, 2, 3]
>>> list(reversed(x))
[3, 2, 1]
```

Sort

sort方法被用来在原地[3]（in place）对列表进行排序。在原地排序意味着改变原来的列表以便其中的元素能按一定的顺序排列而不是简单的返回一个排序了的列表副本。

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort()
>>> x
[1, 2, 4, 6,  7, 9]
```

你已经遇到了几个改变列表却不返回值的方法，在大多数情况下这样的行为方式是很合常理的（比如append方法）。但因为很多人被弄糊涂了，所以我要在排序中强调这样的行为方式。当使用者要对一个列表的副本排序而不管原来的列表时往往会被弄糊涂。一个本能的做法（是错误的）如下所示：

---

3. 假如你感兴趣：从Python.3开始，sort方法使用一个固定的排序算法。

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x.sort() # Don't do this!
>>> print y
None
```

因为sort方法修改了x但放回的是空值，你把x进行了排序但y却包含的是None。一个正确的方法是首先把y绑定（bind）到x的一个副本上，然后对y排序，如下所示：

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x[:]
>>> y.sort()
>>> x
[4, 6, 2, 1,  7, 9]
>>> y
```

[1, 2, 4, 6, 7, 9]调用x[:]得到的是x中的所有元素，这是一中很有效率的复制整个列表的方法。简单的把x赋值给y不起作用，因为这样做就让x和y都指向同一个列表。

```
>>> y = x
```

```
>>> y.sort()
>>> x
[1, 2, 4, 6,  7, 9]
>>> y
[1, 2, 4, 6,  7, 9]
```

另一个得到已排序的列表副本的方法是使用sorted函数。

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = sorted(x)
>>> x
[4, 6, 2, 1,  7, 9]
>>> y
[1, 2, 4, 6,  7, 9]
```

这个函数能被用于任何序列，但是总是返回一个列表[4]。

```
>>> sorted('python)
['P', 'h', 'n', 'o', 't', 'y']
```

如果你想把一些元素按相反的方式排列，你能使用sort（或者sorted），同时调用reverse方法，否则你就要使用下面要描述的reverse参数。

4．sorted函数能被用于其它的能被迭代的对象（iterable object）。关于迭代对象的详细信息请参见第九章。

高级排序

Advanced Sorting

如果你要让元素按一种特殊的方式（降序排列而不是sort函数默认的方式）来排列。你能自己定义一个类似于compare(x,y)的比较函数（comparison function），compare(x,y)会在x小于y时返回一个负值，而当x大于y时返回一个整数，如果x=y则放回0（根据你自己的定义）。然后你能使用自定义的函数进行排序。内部函数cmp提供了默认的方式。

```
>>> cmp(42, 32)
1
>>> cmp(99, 100)
-1
>>> cmp(10, 10)
0
>>> numbers = [5, 2, 9, 7]
>>> numbers.sort(cmp)
```

```
>>> numbers
[2, 5, 7, 9]
```

sort方法有两个参数——键（key）和reverse。如果你要使用它们就要通过名字来指定（这叫做关键字参数(key word arguments)，更多的信息请参见第六章）。参数key和cmp的参数类似——你必须提供一个能用于排序过程的函数。然而，不是用来决定对象的大小而是为每个元素创建一个键，然后所有的元素根据键来排序。比如：如果你要根据元素的长度排序，你就使用len作为键函数（key function）

```
>>> x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']
>>> x.sort(key=len)
>>> x
['add', 'acme', 'aerate', 'abalone', 'aardvark']
```

另一个关键字参数reverse是简单的真值（true或者是false；你能在第五章了解详细信息）用来指示列表是否要反向排序。

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort(reverse=True)
>>> x
[9, 7, 6, 4,  2, 1]
```

The cmp, key, and reverse arguments are available in the sorted function as well. In many cases, using custom functions for cmp or key will be useful—you learn how to define your own functions in Chapter 6.

cmp，键，reverse参数在sorted函数中也是存在的。在许多情况下，使用为cmp或键自定义的的函数更有用——你将在第六章学会如何定义自己的函数。

■**提示** 如果你想得到关于排血的更多内容，你能查看Andrew Dalke的“Sorting Mini-HOWTO,” 你能在http://pythonorg/doc/howto找到。

元组：常量序列

Tuples: Immutable Sequences

元组和列表一样也是一种序列。唯一的不同是元组不能被修改[5]。

```
>>> 1, 2, 3
(1, 2, 3)
```

如你所见，元组也（经常）用圆括号括起：

```
>>> (1, 2, 3)
(1, 2, 3)
```

空元组表示为无内容的两个圆括号。

```
>>> ()
()
```

因此你可能想知道怎样写包含一个值的元组，在这里有点特殊——你必须使用一个逗号，甚至在这里只有一个值。

```
>>> 42
42
>>> 42,
(42,)
>>> (42,)
(42,)
```

最后的两个例子生成了一个长度为一的元组，第一个例子不是元组。逗号是很重要的。只是简单的添加括号是没有用的：(42)和42时一样的。一个逗号却能彻底的改变语句的值。

```
>>> 3*(40+2)
126
>>> 3*(40+2,)
(42, 42, 42)
```

---

5. 元组和列表在技术实现上有一些不同，但你在实际使用时可能不会注意到。并且元组没有像列表一样的方法。

元组函数
The tuple Function

元组函数(tuple)和列表函数（list）基本上是一样的：tuple以一个序列作为参数并把它转换成一个元组[6]。如果参数是一个元组那么就被原样返回。

```
>>> tuple([1, 2, 3])
(1, 2, 3)
```

```
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

基本元组操作
Basic Tuple Operations

就像你已经了解的，元组不是很复杂——你能按访问其他的序列中的元素的方式去访问元组中的元素。

```
>>> x = 1, 2, 3
>>> x[1]
2
>>> x[0:2]
(1, 2)
```

正如你所看到的，元组的切片还是元组，就像列表的切片还是切片一样。

那么，什么是重点？
So What's the Point?

现在你可能想知道为什么有人要一个像元组似的常量序列。难道你不能坚持只用列表并在不希望其中的内容被改变时只要不管列表就行了？基本上，这是可行的，但是有两个很重要的理由你需要了解元组：

它们能在映射(mappings)中用作键（key）——列表不能。（你可能记得映射在导读那章提及过，你能在第四章了解更多的关于映射的信息）

它们能作为很多内部函数和方法的返回值，这就意味着你必须处理它们。只要你不要尝试修改它们，处理它们的意思经常是指像列表那样对待它们。（除了你需要一些元组没有的方法比如索引（index）和计数（count））

大体上说，列表能满足你关于序列所有的需要。

---

6 就像我对列表函数(list)提及的，元组函数(tuple)也不是一个真正的函数——而是一个类型。但对于列表来说你能很安全的忽视这点

。

快速总结
A Quick Summary

让我们回顾一些在本章提及得非常重要的内容：

- 序列。序列是一种其包含的元素都被编号的数据结构。典型的序列有列表，字符串和元组。在这几种序列中，列表是可以被修改的，但元组和字符串是不能修改的（一旦被创建，它们就是固定的）。序列的一部分能通过切片被访问，其中切片支持两个索引号来指出切片的开始和结束的位置。为了改变一个列表，你要把对相应的位置赋值，或者使用赋值语句覆盖整个切片。

- 成员关系。一个值是否存在于一个序列（或者其他的容器）是用in操作符来查看的。在字符串中使用in是一个特例——它能让你寻找子字符串。

- 方法。一些内部类型（比如列表和字符串，元组不在其中）有很多有用的方法。这些方法除了和一个特定的值联系在一起外有些像函数。方法是面向对象编程的一个重要的内容，我们会在第七章讨论面向对象程序设计。

本章内的新函数
New Functions in This Chapter

函数	描述
cmp(x, y)	比较两个值
len(seq)	返回一个序列的长度
list(seq)	把一个序列转换成列表
max(args)	返回一个序列或者参数中的最大值
min(args)	返回一个序列或者参数中的最小值
reversed(seq)	让你再一个序列上反向迭代
sorted(seq)	返回一个seq中元素的排序后的列表
tuple(seq)	把一个序列转化成元组。

现在学什么？
What Now?

你现在对序列很熟了，让我们前进到字符组成的序列，也叫字符串。

第三章

使用字符串

Working with Strings

本章前你应该已经见识过字符串，并且知道如何创建它们了。而且你也看过如何利用索引（indexing）和切片（slicing）访问字符串中的字符。本章中，你会学到如何使用字符串格式化其他的值（比如为了打印），并且会简略的了解利用字符串的splitting、joining、searching等方法能做的有用的事情。

基本字符串操作
Basic String Operations

所有字符串的普通队列操作（索引、切片、复制、成员、长度、最小和最大）你已经在上一章中见到过了。记住，字符串都是不可变（immutable）的，类似这样的切片是不合法的：

```
>>> website = 'http://www.python.org'
>>> website[-3:] = 'com'
Traceback (most recent  call last):
  File "<pyshell#19>",  line 1, in ?
    website[-3:] = 'com'
TypeError: object doesn't support slice assignment
```

字符串格式化：一带而过
String Formatting: The Short Version

字符串格式化可以用格式化操作符——百分号%——实现。

注意 如你所知，%也可以用作模运算（余数）操作符。

左侧放置字符串（格式化的），右侧放置你希望格式化的值。你也可以使用单独的值，比如字符串或者数字，也能使用值的元组（如果你希望格式化多个）。下一章里面我会介绍，你也能使用字典（dictionary）。元组版本的最一般例子：

```
>>> format =  "Hello, %s. %s enough for ya?"
```

```
>>> values =  ('world',  'Hot')
>>> print format % values
Hello, world. Hot enough for ya?
```

■**注意** 如果你使用列表（list）或者其他队列代替元组，那么所用的队列必须转换为单值。只有元组和字典（第四章讨论）允许你使用一个以上的值。

格式化字符串的%s部分叫做转换说明（conversion specifier）。它们对值要代替的位置进行了标记。s表示值如果是字符串的话应该被格式化——如果他们不是，那么会被str转换为字符串。对于大多数值都可以用这个方法，对于列表以及其他特殊类型，参见本章后面的表3-1

■**注意** 要在格式化字符串里面包括百分号，你应该使用%%，这样Python因为第一个转换说明就不会错误地将%转换。

如果你格式化实数（浮点数）的话，可以使用f这个说明类型，加上作为精度的点号“.”，后面跟上你希望保留地小数位数的十进制。格式化说明应总以类型字符结束，所以你应在之前放置精度要求：

```
>>> format =  "Pi with three decimals: %.3f"
>>> from math import pi
>>> print format % pi
Pi  with three decimals: 3.142
```

模板字符串

string模块提供另外一种格式化值的方法：模板字符串。它的工作方式类似于很多UNIX Shell里的变量替换，使用传递到substitute方法的foo的关键字参数替换$foo：

```
>>> from string import  Template
>>> s = Template('$x, glorious $x!')
>>> s.substitute(x='slurm')
'slurm, glorious slurm!'
```

如果替换域是单词的一部分，名字就必须用括号括起，使得结束处变得清晰：

```
>>> s = Template("It's  ${x}tastic!")
>>> s.substitute(x='slurm')
"It's slurmtastic!"
```

插入美元符号，使用$$

```
>>> s = Template("Make  $$ selling $x!")
>>> s.substitute(x='slurm')
'Make $ selling slurm!'
```

除了关键字参数之外，你还可以提供位于字典中的值-名称对（value-name

pair)

```
>>> s = Template('A $thing must never $action.')
>>> d = {}
>>> d['thing'] = 'gentleman'
>>> d['action'] = 'show his socks'
>>> s.substitute(d)
'A gentleman must never show his socks.'
```

方法sfe_substitute不会因缺少值或者不正确使用$字符而出错。请参见Python库参考(http:// python.org/doc/lib/node108.html)的4.1.2部分“模板字符串”。

字符串格式化：长篇大论
String Formatting: The Long Version

格式化操作符的右侧参数可以是任何东西。如果是元组或者字典一类的映射（mapping）的话，将会收到特别对待。我们先不看映射（类似于字典），首先关注一下元组。我们会在第四章中讲到映射的格式化，目前不赘述。如果右侧参数是元组的话，它之中的每一个元素都会被单独格式化，你需要为每个值准备个转换说明。

■**注意** 如果你把需要转换的元组写作转换表达式的一部分，你必须用圆括号将其括起，避免Python出错：

```
>>> '%s plus   %s equals   %s' % (1,   1, 2)
'1   plus 1 equals 2'
>>> '%s plus   %s equals   %s' % 1, 1, 2 # Lacks parentheses!
 Traceback (most recent   call last):
   File "<stdin>", line   1, in
TypeError: not enough arguments for format   string
```

在下面的阅读材料里面，我会带你一起对转换说明做个总结：

剖析转换说明

基本的转换说明（相对于完整的转换说明来说，完整的会包括映射键，参见第四章）包括跟随的项目。注意，在这里顺序是至关重要的。

- **%字符**。标记转换说明的起始。
- **转换标志（可选）**。可选的有“-”：指示左对齐；“+”，指示修改后值的优先报纸。“ ”（空白）：指示正数之前的空格；“0”，指示转换后应该用0填充。

- **最小域宽（可选）**。转换后的字符串至少应该有此值宽。如果为*的话，宽度会从元组中读出。

- **点（.）和精度值（可选）**。如果转换的是实数，后面的十进制精度值应该写出。如果是字符串，那么后面的数字表示最大域宽。如果是值为*，那么精度将会从元组中读出。

- **转换类型**：参见表3-1。

简单转换
Simple Conversion

简单的转换只要写出转换类型，用起来很容易：

```
>>> 'Price of eggs: $%d' % 42
'Price of eggs: $42'
>>> 'Hexadecimal price  of eggs: %x' % 42
'Hexadecimal  price of eggs: 2a'
>>> from math import pi
>>> 'Pi: %f...' % pi
'Pi: 3.141593...'
>>> 'Very inexact estimate of pi: %i' % pi
'Very inexact estimate  of pi: 3'
>>> 'Using str: %s' % 42L
'Using str: 42'
>>> 'Using repr: %r' %  42L
'Using repr:  42L'
```

查看所有的转换类型，参见表3-1

表 3-1. *字符串格式化转换类型*

转换类型	意义
d, i	带符号的十进制整数
o	不带符号的八进制
u	不带符号的十进制
x	不带符号的十六进制（小写）
X	不带符号的十六进制（大写）
e	允许带指数的浮点数（小写）
E	允许带指数的浮点数（大写）
f, F	十进制浮点数
g	如果指数大于-4或者小于精度值则和e相同
f 或 G	如果指数大于-4或者小于精度值则和E相同

F或 c	单字符（接受证书或者单字符字符串）
r	字符串（由任何Python对象使用repr转换）

域宽和精度
Width and Precision

转换说明可以包括域宽和精度。域宽是为转换后的值所保留的最小字符个数，而精度（对于数字转换来说）是结果中应该包含的小数位数，或者（对于字符串转换来说）是转换后的值所能包含的最大字符个数：

```
>>> '%10f' %   pi          #   Field width 10
'    3.141593'
>>> '%10.2f'   % pi       #   Field width 10, precision 2
'          3.14'
>>> '%.2f' %   pi          #   Precision   2
'3.14'
>>> '%.5s' %   'Guido van Rossum'
'Guido'
```

你可以使用*（星号）作为域宽或者精度（或者两者），数值会从元组参数中读出：

```
>>> '%.*s' %   (5, 'Guido van Rossum')
'Guido'
```

符号、对齐和0填充
Signs, Alignment, and Zero-Padding

在域宽和精度值之前你还可以放置一个“标志”，里面可以有零、加号、减号和空格。零表示数字将会用0进行填充。

```
>>> '%010.2f' % pi
'0000003.14'
```

注意，在010中开头的那个0并不意味着“域宽说明”为八进制数，它只是个普通的Python数值。当你在域宽说明内使用010时候，表示域宽为10，并且用0进行填充，而不是说域宽为8：

```
>>> 010
8
```

减号（-）用来左对齐数值：

```
>>> '%-10.2f' % pi
'3.14      '
```

你可以看到，在数的右侧多出了额外的空格。

而空白（“ ”）意味着在正数前加空白。在你需要对齐正负数时候会很有用：

```
>>> print ('% 5d' % 10) + '\n' +   ('% 5d' %   -10)
    10
   -10
```

最后说说加号（+），它表示不管是正数还是负数都应该标示出符号（不管是正还是负）。（在对齐时候也很有用）：

```
>>> print ('%+5d' % 10) + '\n' +   ('%+5d' %   -10)
  +10
  -10
```

下面的例子中，我使用星号域宽说明格式化表格中的水果价格，用户则可输入表格的总宽度。因为是由用户提供信息，我就不能在转换说明时候将域宽定死。而使用星号，我就可以从转换元组中读出域宽。源代码列出于列表3-1：

列表 3-1. *字符串格式化范例*

```
# Print a formatted price list with a given width

width = input('Please enter width: ')


price_width = 10
item_width =   width - price_width
```

```
header_format = '%-*s%*s'
format  = '%-*s%*.2f'

print '=' * width

print header_format % (item_width, 'Item',   price_width, 'Price')

print '-' * width

print format   % (item_width, 'Apples', price_width, 0.4)
print format   % (item_width, 'Pears', price_width, 0.5)
print format   % (item_width, 'Cantaloupes',   price_width, 1.92)
print format   % (item_width, 'Dried Apricots (16 oz.)', price_width, 8)
print format   % (item_width, 'Prunes (4 lbs.)', price_width, 12)

print '=' * width
```

The following is a sample run of the program:
下面是程序运行时候的范例：

```
Please enter   width: 35
===================================
======= Item
```

```
Price
_____________________________
_____________________________
Apples
0.40
Pears                          0.50
Cantaloupes                    1.92
Dried Apricots (16 oz.)        8.00
Prunes (4 lbs.)                12.00
==================================
```

字符串方法
String Methods

你已经在列表那章见过很多方法了。字符串则拥有更加丰富的方法，这是因为字符串从string模块中“继承”了很多方法，而在早期的Python中，这些方法都是作为函数出现的（如果你真的需要的话，还是能找到的）。

但是字符串未死

尽管字符串方法完全来源于string模块，但是在这个模块中还包括一些字符串方法不能作为字符串方法使用的常量和函数。maketrans函数就是其中之一，后面我将会拿它和translate方法一起介绍。表格3-2列出了一些字符串可用的有用常量。对于此模块的更多介绍，请参看Python库参考手册（http://python.org/doc/lib/module-string.html）的4.1部分。

表 3-2. *string模块中有用的值*

常量	描述
string.digits	A string containing the digits 0–9包含有数字0-9的字符串
string.letters	A string containing all letters (upper- and lowercase) 包含有所有字母（大小写）的字符串
string.lowercase	A string containing all lowercase letters 包含所有小写的字符串
string.printable	A string containing all printable characters 包含有所有可打印字符的字符串
string.punctuation	A string containing all punctuation characters 包含有所有标点的字符串
string.uppercase	A string containing all uppercase letters包含有所有大写字母的字符串

因为字符串的方法实在太多，在这里只介绍一些有用的。全部参考请参看附录B。在字符串方法的描述中，可以在本章找到关联到其他方法的参考（用“请参看”标记），或请参看附录B。

find

find方法会在一个较长的字符串中查找子串。它返回字串所在位置的最左端索引。如果没有找到则返回-1

```
>>> 'With a moo-moo here, and a moo-moo there'.find('moo')
7
>>> title = "Monty Python's Flying Circus"
>>> title.find('Monty')
0
>>> title.find('Python')
6
>>> title.find('Flying')
15
>>> title.find('Zirquss')
-1
```

在第二章中我们初识了成员资格（Membership），我们在主题中使用了“$$$”表达式建立了一个垃圾邮件过滤器。我们也可以用在find方法中（Python2.3中也可用，但是只能用来查找字符串中的单个字符的成员属性：

```
>>> subject = '$$$ Get    rich now!!! $$$'
>>> subject.find('$$$')
0
```

注意 string的find方法并不返回布尔值。如果返回的是0，则证明找到了字串，索引值为0。

你也可以给你的搜索提供起始点，同样可选的还有终止点：

```
>>> subject = '$$$ Get    rich now!!! $$$'
>>> subject.find('$$$')
0
>>> subject.find('$$$', 1) # Only supplying the start
20
>>> subject.find('!!!')
16
>>> subject.find('!!!', 0, 16) #    Supplying    start and    end
-1
```

注意对于起始和终止值的范围说明（第二和第三个参数）包含第一个索引，但不包含第二个。这在Python中是个惯例。

join

非常重要的字符串方法，join是split方法的逆方法，用来在队列中添加元素：

```
>>> seq = [1, 2, 3, 4,  5]
>>> sep = '+'
>>> sep.join(seq) # Trying to join a list of numbers
Traceback (most recent  call last):
  File "<stdin>", line  1, in ?
TypeError: sequence item 0: expected string, int found
>>> seq = ['1', '2', '3', '4', '5']
>>> sep.join(seq) # Joining a list of strings
'1+2+3+4+5'
>>> dirs = '', 'usr', 'bin', 'env'
>>> '/'.join(dirs)
'/usr/bin/env'
>>> print 'C:' + '\\'.join(dirs)
C:\usr\bin\env
```

你可以看到需要添加的队列元素都必须是字符串。注意最后两个例子，我使用了字典的列表，而在格式化时候，UNIX和DOS/Windows下面的分隔符号是不同的（在DOS版本中还增加了DOS版本）。
请参看：split。

lower

lower方法返回字符串的小写字母版本。

```
>>> 'Trondheim Hammer Dance'.lower()
'trondheim hammer dance'
```

如果你想要书写“大小写不敏感”的代码的话，这个方法就派上用场了——代码会忽略大小写字幕。举例来说，你可以在列表中查找一个用户名是否存在。如果你的列表包含字符串“Gumby”，而用户输入自己的名字为“Gumby”，你就找不到了：

```
>>> if 'Gumby' in ['gumby', 'smith', 'jones']: print  'Found it!'
...
>>>
```

如果你存储的是“Gumby”而用户输入“gumby”甚至是“GUMBY”，结果也是一样的。解决方法就是在存储和搜索时候把所有名字都转换为小写。代码像下面这样：

```
>>> name = 'Gumby'
>>> names = ['gumby', 'smith', 'jones']
>>> if name.lower() in  names: print 'Found  it!'
...
Found it!
>>>
```

请参看：translate
附录B： islower, capitalize, swapcase, title, istitle, upper, isupper.

replace

replace方法返回一个被替换过字符的字符串。

```
>>> 'This is    a test'.replace('is', 'eez')
'Theez eez a    test'
```

如果你曾经用过文字处理程序中的“查找并替换”功能的话，就不会质疑这个方法的用处了。

请参看：translate。
附录B： expandtabs.

split

非常重要的字符串方法，split是join的逆方法，用来将字符串分割成队列。

```
>>> '1+2+3+4+5'.split('+')
['1', '2', '3', '4', '5']
>>> '/usr/bin/env'.split('/')
['', 'usr', 'bin', 'env']
>>> 'Using      the      default'.split()
['Using', 'the', 'default']
```

注意，如果不提供任何分隔符，程序会把所有空白作为分隔符（空格、缩进、换行以及其他）。

请参看：join
附录B： rsplit, splitlines

strip

strip方法返回去除两侧空格的字符串：

```
>>> '          internal whitespace is kept                '.strip()
'internal whitespace is kept'
```

和lower一起使用的话就可以很方便的对比输入值和存储的值。让我们回到lower部分中的那个例子，假设有人在输入名字时候加上了空格：

```
>>> names = ['gumby', 'smith', 'jones']
>>> name = 'gumby '
>>> if name in names: print 'Found it!'
...
>>> if name.strip() in    names: print 'Found    it!'
...
Found it!
>>>
```

你还可以指定需要去除的字符，将它们列为参数即可。

```
>>> '*** SPAM * for * everyone!!! ***'.strip(' *!')
'SPAM * for * everyone'
```

这个方法只会去除两侧的字符，所以字符串中的星号没有被去掉。
附录B： lstrip, rstrip

translate

和replace一样，translate会替换字符串中的某些部分，但是和前者不同的是，translate只处理单个字符。它的威力在于可以同时进行多个替换，有些时候比replace有用的多。

这个方法的使用手段有很多（比如替换换行符或者其他有平台依赖性的特殊字符）。但是让我们考虑一个简单的例子（尽管有些傻）：假设你想要把英文文本转换为带有德国口音的版本。为了达到你要的效果，需要把字符“c”替换为“k”、“s”替换为“z”。在开始前，你需要有个翻译表。翻译表中是以某字符替换某字符的说明。因为这个表（事实上是字符串）有256个项目，你不用自己写出来，使用string模块里面的maketrans函数就好。

maketrans函数带有两个参数：两个等长的字符，表示第一个字符串中的每个字符都用第二个字符串中相同位置的字符替换。明白了没？看看例子吧：

```
>>> from string import  maketrans
>>> table = maketrans('cs', 'kz')
```

翻译表中都有什么？

```
A translation table is a string containing one replacement letter
for each of the 256 characters in the ASCII
character set:
翻译表是包含有替换ASCII256个字符的说明的字符串。

>>> table = maketrans('cs', 'kz')
>>> len(table)
256
>>> table[97:123]

'abkdefghijklmnopqrztuvwxyz'
>>> maketrans('', '')[97:123]
'abcdefghijklmnopqrstuvwxyz'
```

你可以看到，我已经把小写字母部分的表提取出来了。注意这个字母表和空的翻译表。空的翻译表包含有一个普通的字母表，而在上一个例子中，字母c和s分别为替换为k和z。

一旦你拿到这个表，你可以将它用作tranlate方法的参数，进行字符串的翻译。

```
>>> 'this is  an incredible test'.translate(table)
'thiz iz an inkredible  tezt'
```

translate的第二个参数可选，用来标明需要删除的字符。如果你想要得到一个语速超快的德国英语文本，可以删除所有空格：

```
>>> 'this is    an incredible test'.translate(table, ' ')
'thizizaninkredibletezt'
```

■**提示** 有些时候类似于lower的字符串方法并不能如你所愿的工作，比如你想使用一个非英语字母的字母表。就拿大写挪威字母举例吧：
“BØLLEFRØ”转换为小写：

```
>>> print 'BØLLEFRØ'.lower()
bØllefrØ
```

没有成功。因为Python不把“Ø”当作真正的字幕。本例中，你可以使用translate方法完成转换：

```
>>> table = maketrans('ÆØA', 'æøå')
>>> word = 'KÅPESØM'
>>> print word.lower()
kÅpesØm
>>> print word.translate(table)
KåPESøM
>>> print word.translate(table).lower()
kåpesøm
```

请参看：replace，lower

快速总结
A Quick Summary

本章中，你学到了有关string的两种非常重要的使用方式。

字符串格式化。模除操作符（%）可以用来将其他值转换为字符串，亦可带有诸如%s之类的转换标志进行转换。你可以用它来对字符串进行不同方式的格式化，包括左右对齐、设定域宽以及精度值，增加符号（正负号）或者左填充0等。

字符串方法。字符串有一大堆方法。有些非常有用（比如split和join），有些则用的很少（比如istitle或者capitalize）。

本章内的新函数
New Functions in This Chapter

函数	描述
string.maketrans(from, to)	创造用于转换的翻译表

现在学什么？
What Now?

列表、字符串和字典是Python中最重要的三种数据类型。你已经见识过列表和字符串，那么下面是什么呢？下一章中，你将会学到字典是如何支持索引以及其他方式的索引键（比如字符串和元组）的。字典也提供了一些方法，但是没有字符串多。

第四章

字典：当索引不好用时

Dictionaries:When Indices Won't Do

你已经了解到，当你想把数值分组为结构，并且将每个值赋予数字索引时候，列表（List）是很有用的。本章中，你将学习一种依名字索引数值的数据结构。这类数据结构被称为映射（Mapping），Python之中唯一内建的映射类型就是字典（Dictionary）。字典中的值并没有特殊的顺序，但是都按照关键字（Key）进行存储，关键字可以是数字、字符串，甚至是元组（Tuple）。

但它用来做什么？
But What Are They For?

很多情况下字典比列表更加适于使用：那么“字典”这个名字或许能让你有点头绪：一本普通的书是用来从头看到尾的。如果你愿意，也可以很快的打开到任何给定的页。这有点像Python的列表。而字典（不管是真实的还是Python中的）被设计成可以轻松查找特殊词语（关键字）以找到其定义（值）。Python字典有些很特殊的用途：

- 表现游戏棋盘，每个关键字都是坐标值构成的元组；
- 储存文件修改时间，用文件名作为关键字；
- 电话/住址簿

比方说你有如下包含人名的列表：

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
```

如果你想要创建一个小型的可以储存这些人电话号码的数据库的话——你会怎么办呢？一个方法就是建立另外一个列表。让我们假设只需储存四位的号码。那么就会有这样一个列表：

```
>>> numbers = ['2341',  '9102', '3158', '0142', '5551']
```

■**注意** 你可能会奇怪为什么我用字符串表示电话号码——为什么不用整数？考虑一下Dee-Dee的电话号码：

```
>>> 0142
98
```

并不是我们想要的结果，不是吗？就像在第一章中简略地提到的那样，八进制数字均以0开

头。十进制数字是不可能表示类似那样形式的。

```
>>> 0912
File "<stdin>", line  1
          0912
          ^
SyntaxError: invalid  syntax
```

这件事情告诉我们：电话号码（以及其他可能开头为0的数字）应该表示为字符串，而不是整数类型。

一旦你建立了这些列表，你可以像这样查找Cecil的电话：

```
>>> numbers[names.index('Cecil')]
3158
```

工作正常，但是有些不切实际。你**真正**想要的效果是：

```
>>> phonebook['Cecil']
3158
```

猜到了吗？如果电话簿用字典来做，你就能像上面那样做了。

字典的语法
Dictionary Syntax

字典的语法如下：

```
phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

字典中包含很多对（称为条目items）键（keys）和它们相对应的值（values）。在前例中，名字是键，而电话号码就是值。每个键和值之间用冒号（:）隔开，条目之间以逗号（,）相隔，而整个字典是由一对花括号括起。空字典（没有任何项目）由两个花括号组成，像这样：{}。

■ **注意** 字典中的键是唯一的（其他类型的映射也是如此），而值则不然。

字典函数
The dict Function

你可以用字典函数从其他映射（比如其他字典）或者（键，值）的序列建立字典。

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d
{'age': 42, 'name': 'Gumby'}
>>> d['name']
```

'Gumby'

它可以带有关键词参数使用，如下：

```
>>> d = dict(name='Gumby', age=42)
>>> d
{'age': 42, 'name': 'Gumby'}
```

然而这可能并不是dict函数最有用的功能，你还能利用它和映射参数建立和映射中具有同样项目的字典（在没有带有参数使用时，它返回一个空字典，就像其他类似的函数list、tuple或者str一样）。如果另外的那个映射也是字典（毕竟这是唯一内建的映射类型），你可以使用字典方法来进行拷贝，后面我们会说到。

■**注意** dict函数并不真的是一个函数。它是个类型，就像list、tuple和str一样。

基本字典操作
Basic Dictionary Operations

字典最基本的行为和序列（sequence）类似：len(d)返回d中条目（键-值）的数量，d[k]返回关联到键k上的值，d[k]=v将值v关联到关键字k上，del d[k]删除拥有键k的条目，k in d检查d中是否含有键k。尽管有很多特性相同，两者还是有重要的不同的：

- 字典的键不一定为整数（可能会是），应为不可变类型，比如浮点数（Float number）、字符串（String）或者元组（Tuple）。
- 你可以关联一个值到一个键上面，就算那个键不存在于字典中，这种情况下新的条目会建立。你不能将值关联到列表范围内不存在的索引（index）上。
- 表达式k in d（d为字典）用来查找**键**，而不是**值**。表达式v in l（l为列表）则用来查找**值**，而不是**索引**。这样看起来有些不太一致，但是当你使用的时候就会感觉非常自然了。毕竟，如果字典含有给出的键，查找相应的值也就很简单了。

■**提示** 检查键在字典中的成员资格比检查值在列表中的要高效得多，数据结构的规模越大，区别越明显。

第一点——键可以为任何不变类型——是字典最强项，第二点也很重要。看看下面的区别：

```
>>> x = []
>>> x[42] = 'Foobar'
Traceback (most recent    call last):
          File "<stdin>", line    1, in ?
          IndexError: list assignment index out of range
>>> x = {}
```

```
>>> x[42] = 'Foobar'
>>> x
{42: 'Foobar'}
```

首先，我试图将字符串'Foobar'关联到一个空列表的42号位置——显然是不可能的，因为那个位置根本不存在。为了将其变为可能，我必须要用43个'None'或者其它什么初始化x，而不能仅仅使用[]。下一个尝试中，却一切完美。我将'Foobar'关联到空字典的42键上，没问题！新的条目已经添加到字典中，我达到目的了。

示例

列表4-1．字典示例

```
people = {

    'Alice':  {
        'phone': '2341',
        'addr': 'Foo drive 23'
    },

    'Beth': {
        'phone': '9102',
        'addr': 'Bar street 42'
    },

    'Cecil':  {
        'phone': '3158',
        'addr': 'Baz avenue 90'
    }

}

# Descriptive labels for the phone number and address. These will be used
# when printing the output.
labels = {
    'phone':  'phone number',
    'addr': 'address'
}

name = raw_input('Name: ')

# Are we looking for a  phone number or an address?
request = raw_input('Phone number (p) or address (a)? ')

# Use the correct key:
if  request == 'p': key  = 'phone'
```

```
if   request == 'a': key   = 'addr'

# Only try to print information if the name is a valid key in   our dictionary:
if   name in people: print "%s's %s is %s." % (name, labels[key], people[name][key])
```

用字典格式化字符
String Formatting with Dictionaries

第三章中，你已经看到如何使用字符串格式化功能来格式化元组中的值。如果你使用字典（只有字符串作为键）而不是元组来做这个工作，甚至还会更酷一些。在每个转换说明（conversion specifier）中的%字母后面，你可以加上键（用圆括号括住），后面跟上其他说明元素。

```
>>> phonebook
{'Beth': '9102', 'Alice': '2341', 'Cecil':   '3258'}
>>> "Cecil's   phone number is %(Cecil)s." %   phonebook
"Cecil's phone number is 3258."
```

除了添加字符串键，转换字串还是像以前一样工作。当像这样使用字典时候，只要所有给出的键都在字典中找到，你可能获得任意数量的所需字符。这类字符串格式化在模板系统中非常有用（本例中使用HTML）。

```
>>> template   = '''<html>
<head><title>%(title)s</title></head>
<body>
<h1>%(title)s</h1>
<p>%(text)s</p>
</body>'''
>>> data = {'title': 'My Home Page', 'text': 'Welcome to my home page!'}
>>> print template % data
<html>
<head><title>My Home Page</title></head>
<body>
<h1>My Home Page</h1>
<p>Welcome to my home page!</p>
</body>
```

■**注意** 字符串模板类（第三章中提到过）也是此类非常有用的工具

字典方法
Dictionary Methods

就像其他内建类型一样，字典也有方法。这些方法非常有用，你可能不会像使用列表或者

字符串方法那样频繁的使用。你可能会想跳过此章，对于哪些方法可用有个大概印象，然后再返回头来查看具体哪个方法怎么用。

clear

clear方法清除字典中所有的条目。这是个内部（in-place）操作（类似于list.sort），所以无返回值。

```
>>> d = {}
>>> d['name'] = 'Gumby'
>>> d['age']   = 42
>>> d
{'age': 42, 'name': 'Gumby'}
```

```
>>> returned_value = d.clear()
>>> d
{}
>>> print returned_value
None
```

为什么这个方法有用？考虑一下下面的情况，注意两种行为间的不同。
情况1：

```
>>> x = {}
>>> y = x
>>> x['key']   = 'value'
>>> y
{'key': 'value'}
>>> x = {}
>>> y
{'key': 'value'}
```

情况2：

```
>>> x = {}
>>> y = x
>>> x['key']   = 'value'
>>> y
{'key': 'value'}
>>> x.clear()
>>> y
{}
```

两种情况中，x和y关联于同一个字典。情况1中，我用关联一个新字典的方法“清空”了x，对于y一点影响也没有，它还关联到原先的字典。这可能是你所需要的行为，但是如果你真的想清空**原始**字典中所有的元素，你必须使用clear方法。你可以看到在情况2中，y随后也被清空了。

copy

copy方法返回一个具有相同键-值配对的字典（为浅（shallow）拷贝，因为值本身就是**相同**的，而不是拷贝。）

```
>>> x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
>>> y = x.copy()
>>> y['username'] = 'mlh'
>>> y['machines'].remove('bar')
>>> y
{'username':   'mlh', 'machines': ['foo', 'baz']}
>>> x
{'username':   'admin', 'machines': ['foo', 'baz']}
```

你可以看到，当你在拷贝中替换值的时候，原始字典不受影响，**但是**，如果你**修改**了某个值（内部修改，而不是替换），原始的字典也会改变，因为同样的值也存储在原字典中（就像上面的"mechines"列表一样）

■**提示** 避免这个问题的方法就是使用深（deep）拷贝，拷贝其所有包含的值。你可以使用copy模块的deepcopy函数：

```
>>> from copy import deepcopy
>>> d = {}
>>> d['names'] = ['Alfred', 'Bertrand']
>>> c = d.copy()
>>> dc = deepcopy(d)
>>> d['names'].append('Clive')
>>> c
{'names': ['Alfred', 'Bertrand',   'Clive']}
>>> dc
{'names': ['Alfred', 'Bertrand']}
```

---

fromkeys

fromkeys方法使用给定的键建立新的字典，每个键默认对应的值为None。

```
>>> {}.fromkeys(['name', 'age'])
{'age': None, 'name': None}
```

刚才的例子中首先建立了一个空字典，然后使用fromkeys方法，建立**另外**一个词典——有些多余。你可以直接在所有字典的类型dict上面调用方法。（关于类型和类的概念在第七章中会深入讨论）

```
>>> dict.fromkeys(['name', 'age'])
{'age': None, 'name': None}
```

如果你不想使用None作为默认值，也可以自己提供默认值。

```
>>> dict.fromkeys(['name', 'age'], '(unknown)')
{'age': '(unknown)', 'name': '(unknown)'}
```

get

get方法是个更宽松的访问字典条目的方法。一般来说，如果你试图访问字典中没有的条目时会出错：

```
>>> d = {}
>>> print d['name']
Traceback (most recent    call last):
     File "<stdin>",         line    1, in
  ? KeyError: 'name'
```

而用get就不会：

```
 >>> print d.get('name') None
```

可以看到，当你试图访问一个不存在的键时，没有任何异常，反过来你得到了None值。你可以自定义“默认”值，替换None：

```
 >>> d.get('name', 'N/A')
 'N/A'
```

如果键存在，get用起来就像普通的字典查询一样：

```
 >>> d['name'] = 'Eric'
 >>> d.get('name')
 'Eric'
```

has_key

has_key方法可以检查字典中是否含有给出的键。表达式d.has_key(k)相当于k in d。使用哪个方式很大程度上取决于自己的喜好。这里有一个你可能会使用has_key的例子：

```
>>> d = {}
>>> d.has_key('name')
0
>>> d['name'] = 'Eric'
>>> d.has_key('name')
1
```

items和iteritems

items方法将所有的字典条目以列表方式返回，每个元素的形式为（键，值）。但是条目的返回并没有特殊的顺序。

```
>>> d = {'title': 'Python Web Site', 'url': 'http://www.python.org', 'spam': 0}
>>> d.items()
[('url', 'http://www.python.org'), ('spam', 0), ('title', 'Python Web Site')]
```

iteritems方法使用方法也类似，但是会返回一个迭代（iterator）对象而不是列表。

```
>>> it = d.iteritems()
>>> it
<dictionary-iterator object at 169050>
>>> list(it)   # Convert   the iterator to a list
[('url', 'http://www.python.org'), ('spam', 0), ('title', 'Python Web Site')]
```

在很多情况下使用iteritems更加有效率（尤其是你想要迭代结果的情况下）。查看关于迭代器的更多信息，请参看第九章。

keys和iterkeys

keys方法将字典中的键以列表方式返回，iterkeys则返回迭代形式的键。

pop

pop方法用来获得对应于给定键的值，然后将这对键值从字典中移除。

```
>>> d = {'x': 1, 'y': 2}
>>> d.pop('x')
1
>>> d
{'y': 2}
```

popitem

popitem方法类似于list.pop。但是和后者不同的是，popitem弹出随机的条目，因为字典并没有“最后的元素”或者其他顺序的概念。当你想一个个移除条目时候，这个方法就非常有效（不用首先获取键的列表）。

```
>>> d
{'url': 'http://www.python.org',   'spam': 0, 'title':   'Python Web Site'}
>>> d.popitem()
('url', 'http://www.python.org')
>>> d
{'spam': 0, 'title': 'Python Web   Site'}
```

尽管popitem和列表的pop方法很类似，但字典中没有对应的append方法。因为字典是无序的，类似于append的方法是没有任何意义的。

setdefault

setdefault方法某种程度上类似于get，除了后者的基本功能外，setdefault还能在字典中不含有给定键的情况下**设定**相应的键值。

```
>>> d = {}
>>> d.setdefault('name', 'N/A')
'N/A'
>>> d
```

```
{'name': 'N/A'}
>>> d['name'] = 'Gumby'
>>> d.setdefault('name', 'N/A')
'Gumby'
>>> d
{'name': 'Gumby'}
```

你可以看到，当键不存在的时候，setdefault返回默认值并且更新字典。如果键存在，那么就返回其对应值，字典不改变。默认值是可选的，这点和get一样。如果不设定，会默认使用None。

```
>>> d = {}
>>> print d.setdefault('name')
None
>>> d
{'name': None}
```

update

update方法可以利用一个字典的条目更新另外一个字典：

```
>>> d = {
        'title': 'Python Web Site',
        'url': 'http://www.python.org',
        'changed': 'Mar    14 22:09:15 MET 2005'
        }
>>> x = {'title': 'Python Language Website'}
>>> d.update(x)
>>> d
{'url': 'http://www.python.org',    'changed': 'Mar 14 22:09:15 MET 2005',
'title': 'Python Language Website'}
```

提供条目的字典会被添加到旧的字典中，若有相同的键则会进行覆盖。

update方法可以用dict函数（或者类型构造器）进行调用，这点在本章前面已经讨论。这就意味着update可以和映射、拥有（键，值）对的队列（或者其他可迭代的对象）以及关键字参数一起调用。

values和itervalues

values方法以列表的形式返回字典中的值（itervalues返回值的迭代）。和keys不同，以列表形式返回的值可以有重复：

```
>>> d = {}
>>> d[1] = 1
>>> d[2] = 2
>>> d[3] = 3
>>> d[4] = 1
>>> d.values()
[1, 2, 3, 1]
```

修改上个例子

列表4-2演示了一个列表4-1程序的修改版本，其使用get方法访问“数据库”实体。接着是一个程序执行的例子。注意get方法添加值的灵活性，它使得程序在用户输入我们并未准备的值时也能做出有用的反应。

```
Name:Gumby
Phone number    (p) or address (a)?    batting average
Gumby's batting average is not available.
```

Listing 4-2. *Dictionary Method Example*

```
# A simple database using get()

# Insert database (people) from Listing 4-1 here.

labels = {
      'phone':    'phone number',
      'addr': 'address'
}

name = raw_input('Name: ')

# Are we looking for a    phone number or an address?
request = raw_input('Phone number (p) or address (a)? ')

# Use the correct key:
key = request # In case the request is neither 'p' nor 'a'
if    request == 'p': key    = 'phone'
if    request == 'a': key    = 'addr'

# Use get to    provide default values:
person = people.get(name, {})
label = labels.get(key, key)
result = person.get(key, 'not available')

print "%s's %s is %s."    % (name, label, result)
```

快速总结
A Quick Summary

本章中，你学习了：

映射。映射可以让你以任何不可变对象标识元素。最常用的类型是字符串和元组。Python唯一内建的映射类型是字典。

利用字典格式化字符串。你可以使用字典的字符串格式化操作对内部的名字（键）进行特定格式化。当对元组进行字符格式化时候，你还需要对元组中每一个元素都设定“格式化说明”（formatting specifier）。在使用字典时，你所用的说明可以比字典中的条目少。

字典的方法。字典有很多方法，调用的方式和列表以及字符串方法的方式相同。

本章内的新函数
New Functions in This Chapter

函数	描述
dict(seq)	用（键，值）对建立字典

现在学什么？
What Now?

现在你对于Python的基本数据结构已经有了很多了解，并且知道怎么在表达式中使用它们。回想一下第一章，计算机程序还有另外一个重要的因素——语句（statements）。下掌中我们会对它们进行详细的讨论。

第五章

条件、循环和其他语句

Conditionals、Loops and Some Other Statements

估计你学到这里都有点不耐烦了，没错——这些数据结构什么的看起来都不错，但是你还是用它们做不了什么事，不是么？

那我们就加快进度好了。你已经见过一些类型的语句（print语句、import语句、赋值语句）。在将语句分为条件语句（conditionals）和循环语句（loops）前，我们先看看更多使用他们的方法。之后你会看到类似于循环和条件语句的列表推导式（list comprehensions），尽管它们本身是表达式。最后你会学到pass、del和exec的用法。

关于print和import的更多信息
More About print and import

随着学习关于Python的更多知识，你可能会注意到有些你认为自己学会Python知识点，还藏着一些让你惊讶的特性。让我们看看print和import的特性吧。

和逗号一起输出
Printing with Commas

你已经学会如何使用print来打印表达式——不管是字符串还是其他类型自动转换的字符串。但是事实上你可以打印一个以上的字符串，只要用逗号隔开就好：

```
>>> print 'Age:', 42
Age: 42
```

可以看到，每个参数之间都插入了一个空格符。

■**注意** print的参数**并不**组成一个元组：

```
>>> 1, 2, 3
(1, 2, 3)
>>> print 1,  2, 3
1 2 3
>>> print (1, 2, 3)
```

```
(1, 2, 3)
```

如果你想要同时输出文本和变量，却又不想强大的字符串格式化功能的话，这个特性就有用了：

```
>>> name = 'Gumby'
>>> salutation = 'Mr.'
>>> greeting   = 'Hello,'
>>> print greeting, salutation, name
Hello, Mr. Gumby
```

■**注意** 如果greeting字符串不带逗号，那么你在结果中怎么能得到逗号呢？你不能使用

```
print greeting, ',', salutation,   name
```

因为上面的语句会在逗号前加入空格。一个解决方案如下：

```
print greeting + ',', salutation, name
```

这样，在问候语后面就加上了逗号。

如果你在结尾处加上逗号，那么你接下来打印的语句会紧跟前一行，例如：

```
print 'Hello,',
print 'world!'
```

```
print out Hello, world!
```

把某某作为某某导入
Importing Something As Something Else

从模块导入的时候，你可以使用

```
import somemodule
```

或者

```
from somemodule import   somefunction
```

或者

```
from somemodule import   *
```

有在你确定自己想要从给定模块导入**所有**功能时候才应该使用最后一个语句。但是如果你有两个模块都有一个叫做open的函数——那又该怎么办？你可以像第一

个语句那样导入，然后像下面这样使用：

```
module1.open(...)
module2.open(...)
```

但是还有一个选择：你可以在语句末尾提供你想要使用的名字，可以是模块名字：

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0
```

也可以是函数名：

```
>>> from math import sqrt as foobar
>>> foobar(4)
2.0
```

对于open函数，你可以像下面这样使用：

```
from module1    import open as open1
from module2    import open as open2
```

赋值魔法
Assignment Magic

就算是低调的赋值语句也有一些技巧：

序列解包
Sequence Unpacking

你已经见过赋值语句的例子，包括变量和数据结构部件的（比如list的位置和切片以及字典的位置），但是还有些东西要说。你可以**同时**进行多个赋值：

```
>>> x, y, z = 1, 2, 3
>>> print x,  y, z
1 2 3
```

很有用吧？你还能用它交换两个（或更多个）变量：

```
>>> x, y = y, x
>>> print x,  y, z
2 1 3
```

事实上，我在这里做的事情叫做“序列解包（sequence unpacking）”——一些值组成序列，然后我把它们解包到变量的序列中。说更直接一点就是：

```
>>> values =  1, 2, 3
```

```
>>> values
(1, 2, 3)
>>> x, y, z = values
>>> x
1
```

当你函数或者方法返回元组（或者其他序列和迭代对象）时候这个特性尤其有用。打个比方，你想要接受（和删除）字典中任意的键-值对。你可以使用popitem方法，这个方法将键-值作为元组返回。你可以把这个元组直接赋值到两个变量：

```
>>> scoundrel = {'name': 'Robin', 'girlfriend': 'Marion'}
>>> key, value = scoundrel.popitem()
>>> key
'girlfriend'
>>> value
'Marion'
```

它允许函数返回一个以上的值并且打包成元组，然后通过一个赋值语句进行访问。你所解包的序列中的元素数量必须和你放置在赋值符号=左边的变量数量完全一致，否则Python会在赋值时候抛出异常。

链式赋值
Chained Assignments

链式赋值（Chained assignments）是你绑定很多变量到一个值时候的捷径。看起来有些像上节中的同时赋值，不过这次面对的是一个值：

```
x = y = somefunction()
```

和下面语句的效果是一样的：

```
y = somefunction()
x = y
```

注意前面的语句**不能**写成：

```
x = somefunction()
y = somefunction()
```

更多的信息，请参看本章后面关于标识运算符（is, Identity Operator）的一节。

扩张赋值
Augmented Assignments

除了写成x=x+1之外，你还能把表达式运算符放置在赋值符号（=）左边，写成x+=1。这种写法叫做扩张赋值（Augmented Assignments），对于*、/、%等标准运算符都适用：

```
>>> x = 2
>>> x += 1
>>> x *= 2
>>> x
6
```

对于其他数据结构也适用：

```
>>> fnord = 'foo'
>>> fnord +=  'bar'
>>> fnord
'foobar'
```

扩张赋值可以让你的代码更加紧凑和简练，很多情况下会更易读。

■**提示** 一般来说，对于字符串不应该使用+=，尤其是你想要在循环中建立一个长字符串的时候（更多关于循环的信息请参看本章后面的循环部分）。每次的自增和复制都需要建立一个新的字符串，消耗时间，让程序变得较慢。较好的方法是把小字符串附加到列表，然后使用字符串的join方法建立长字符串。

语句块：缩排的乐趣
Blocks: The Joy of Indentation

下面要讲的不是某种语句，但是是你在掌握下面两个部分之前应该了解的。

语句块是在条件为真（条件语句）时执行或者执行多次（循环语句）的**一组**语句，语句块由代码缩进制造，也就是在语句前放置空格。

■**注意** 你也可以使用tab字符缩进语句块。Python将一个tab解释为到下一个tab位置的移动，而一个tab位置为8个空格，但是标准且推荐的方式是只用空格，尤其是在每个缩进需要4个空格的时候。

块中的每行都应该缩进同样的量。下面的伪代码（并未真正Python代码）演示了缩进：

```
this is a line
this is another line:
    this is another block
    continuing the same block
    the last  line of this block
phew, there we escaped  the inner  block
```

很多语言中，一个特殊单词或者字符（比如begin或{）用来开始一个语句块，其他的（比如end或者}）的单词或者字符用来结束。Python中，冒号（:）用来标识语句块的开始，块中的每一个语句都是缩进的（同样的缩进量）。当你回退到和已经闭合的块一样的缩进量时，你就明白当前块已经结束了。

现在我能肯定你很好奇块是怎么使用的了，所以废话不多说，继续学下去。

条件和条件语句
Conditions and Conditional Statements

目前为止你写的程序都是语句一条条顺序执行的。这部分中会介绍让你的程序选择是否执行语句块的方法。

这就是布尔变量的作用
So That's What Those Boolean Values Are For

你需要**事实值**（也叫做布尔值，这个名字根据在事实值上做过大量研究的George Boole命名），马上你就要不停的接触到它了。

■**注意** 如果你仔细注意过的话就会发现在第一章的“管窥：if语句”中就已经描述过if语句。到目前为止我还没正式介绍这个语句。马上你会发现它的内容比我目前告诉你的要多。

对于解释器来说，下面的值都为**假**（false）：

```
False      None      0      ""      ()      []      {}
```

换句话说，也就是所有的False和None标准数值、所有类型的数字0（包括浮点、长整型和其他类型）、空的序列（比如空字符串、元组和列表）以及空的字典都是false值。**其他的一切**都被解释为true，包括特殊值True。Laura Creighton称其为辨别**一些东西**和**没有东西**的区别，而不是真和假的区别。

明白了没？这也就意味着Python中的所有值都能被解释为事实值，初识的时候你可能会有些搞不明白，但是这点的确非常有用。“标准的”事实值为True和False。在老版本的Python中，标准的事实值为0（表示假）和1（表示真）。事实上，True和False只不过是0和1的一种“华丽”的说法而已，看起来不同，用起来一样。

```
>>> True
True
>>> False
False
>>> True == 1
True
```

```
>>> False ==  0
True
>>> True + False + 42
43
```

那么，如果你看到某个逻辑表达式返回1或0（在老版本Python中），你就知道它实际的意思是True和False。

布尔值True和False属于类型bool，可以用来（和list、str以及tuple一样）转换其他值。

```
>>> bool('I think, therefore I am') True
>>>
bool(42)
True
>>> bool('')
False
>>> bool(0)
 False
```

因为所有值都可以用作布尔值，你几乎不用需要对它们进行显示转换（explicit conversion）。

条件执行和 if 语句
Conditional Execution and the if Statement

事实值可以联合使用（一会你会看到），但是让我们先看看它们能做什么。试着运行下面的代码：

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
    print 'Hello, Mr. Gumby'
```

这就是if语句，可以让你进行**条件执行**。这就意味着如果**条件**（在if和冒号之间的部分）判定为**真**（前面说过了），那么后面的语句块（本例中是简单的print语句）就会被执行。如果条件为**假**，语句块就**不会**被执行（恐怕你已经猜到了吧？）。

■**注意** 在第一章的“管窥：if语句”中，所有语句都写在一行里面了。那种书写方式和上例是等价的。

---

else 子句
else Clauses

前一节的例子中，如果你输入了以“Gumby”为结尾的名字，那么name.endswith就会返回真，使得if会执行语句块中的问候语。如果你愿意的话也可以用else子句（之所以叫做**子句**是因为它不能单独使用，只能作为if语句的一部分）对另外一种情况进行处理。

```
name = raw_input('What is your name? ')
if name.endswith('Gumby'):
    print 'Hello, Mr. Gumby'
else:
    print 'Hello, stranger'
```

如果第一个语句块没有被执行（因为条件被判定为假），你就会转入第二个语句块，可以

看到，阅读Python代码很容易，不是吗？大声把代码读出来，听起来就像正常（可能也不是**很**正常）的句子一样。

elif子句
elif Clauses

如果你想要检查数个条件，就可以使用elif，它是“else if”的简写。它是if和else子句的联合使用——也就是具有条件的else子句。

```
num = input('Enter a number: ')
if   num > 0:
     print 'The number is positive'
elif num < 0:
     print 'The number is negative'
else:
     print 'The number is zero'
```

缩进代码块
Nesting Blocks

让我们来点有意思的。你可以在if语句里面使用if语句，就像这样：

```
name = raw_input('What   is your name? ')
if name.endswith('Gumby'):
     if name.startswith('Mr.'):
          print 'Hello, Mr. Gumby'
     elif name.startswith('Mrs.'):
          print 'Hello, Mrs. Gumby'
     else:
          print 'Hello, Gumby'
else:
     print 'Hello, stranger'
```

如果名字是以“Gumby”结尾的话，还要检查名字的开头——在第一个语句块中又有if语句。注意这里elif的使用。最后一个条件中（else子句）没有条件——如果其他的条件都不满足就使用最后一个。你可以把任何一个else放在语句块外。如果把里面的else子句放在外面，名字不以“Mr.”或者“Mrs.”开头的的都被忽略掉了（但都是以“Gumby”结尾的）。如果你不写最后一个else子句，陌生人就被忽略掉了。

更复杂的条件
More Complex Conditions

这就是if语句的所有知识了。下面我们回到条件，因为条件执行的部分还是蛮有趣的。

比较运算符
Comparison Operators

用在条件中的最基本的运算符就是比较运算符（comparison operators）了，它们被用来比较其他东西。比较运算符已经总结在表5-1中。

表 5-1. *Python中的比较运算符*

Expression表达式	Description描述
x == y	x 等于 y.
x < y	x 小于 y.
x > y	x 大于 y.
x >= y	x 大于等于 y.
x <= y	x 小于等于 y.
x != y	x 不等于 y.
x is y	x 和 y是同一个对象.
x is not y	x 和 y是不同的对象
x in y	x 是 y容器（比方说序列）的成员.
x not in y	x 不是 y容器（比方说序列）的成员

如果你在其他地方见过x<>y这样的表达式的话，你应该知道它就是x!=y。<>运算符是不赞成使用的，应该避免它。在Python中比较运算和赋值运算一样是可以链接的——你可以把几个运算符连在一起，比如：0<age<100。

■ **提示** 比较东西的时候，你可以使用第二章中介绍的内建的cmp函数。

有些运算符值得特别关注一下，下面的章节中会有介绍。

相等运算符
The Equality Operator

如果你想要知道两个东西是否相等，应该使用相等运算符，即两个等号，==：

```
>>> "foo" ==
"foo" True
>>> "foo" ==
"bar" False
```

两个等号？为什么不像数学里面一样，只用一个呢？我想你已经悟到是为什么了，但是还是看一下：

```
>>> "foo" = "foo"
SyntaxError:  can't assign to literal
```

单个相等运算符是赋值运算符，是用来**改变**东西的，而**不能**用来比较。

IS：同一性运算符
is: The Identity Operator

这个运算符比较有趣。它看起来和==一样，事实上却不是：

```
>>> x = y = [1, 2, 3]
>>> z = [1, 2, 3]
>>> x == y
True
>>> x == z
True
>>> x is y
True
>>> x is z
False
```

到最后一个例子之前，一切看起来都很好，但是最后一个结果很奇怪，x和z相等却不等同，为什么呢？因为is是判定**同一性**的，而不是**等同性**。变量x和y都被绑定到**同一个**列表上，而z被绑定在另外一个具有相同数值和顺序的列表上。它们的值可能相等，但是却不是同一个对象。

看起来有些不可理喻？看看这个例子：

```
>>> x = [1, 2, 3]
>>> y = [2, 4]
>>> x is not  y
True
>>> del x[2]
>>> y[1] = 1
>>> y.reverse()
```

本例中，我以两个不同的列表x和y开始。可以看到x is not y，这个你已经知道了。之后我改动了以下列表，尽管它们的值相等了，但是还是两个不同的列表。

```
>>> x == y
True
>>> x is not y
True
```

显然，两个列表值等但是不本身不同。

总结一下：使用==来判定两个对象是否**相等**，使用is判定两者是否**同一**（同一个对象）。

■**警告** 避免对于类似数值和字符串之类的基本、不可变值使用 is。由于 Python内部操作这些对象的方式，使用的结果是不可以预测的。

---

in：成员资格运算符
in: The Membership Operator

我已经介绍过in运算符了（在第二章的“成员资格”部分）。它可以在条件中使用，就像其他的比较运算符一样。

```
name = raw_input('What  is your name? ')
if  's' in name:
    print 'Your name contains the letter "s".'
else:
    print 'Your name does not contain the letter "s".'
```

比较字符串和序列
Comparing Strings and Sequences

字符串是可比的，因为它们可以按照字母顺序排列进行比较。

```
>>> "alpha" < "beta"
True
```

如果你在里面加入了大写字母，结果就会有点乱。（事实上，字符是按照本身的顺序值排列的。一个字母的顺序值可以用ord查到，ord的反函数为chr）。为了避免产生大小写字母的歧义，可以使用字符串方法转换为大写或小写字母。

```
>>> 'FnOrD'.lower() ==  'Fnord'.lower()
True
```

同理，其他的序列也是可比的，不过比的不是字符而是其他类型元素。

```
>>> [1, 2] <  [2, 1]
True
```

如果一个序列中有其他序列作为元素，比较规则也同样应用于序列元素。

```
>>> [2, [1, 4]] < [2, [1, 5]] True
```

布尔运算符
Boolean Operators

现在你已经见过很多返回事实值的东西了（事实上，所有值都可以转换为事实值，**所有的表达式**也都返回事实值）。但是你可能还想要使用一个以上的条件。打个比方，你想要写个读取数字并且判断他是否位于1-10之间的程序，你**可以**这样做：

```
number = input('Enter a number between 1 and 10: ')
if number  <= 10:
    if number >= 1:
        print 'Great!'
    else:
        print 'Wrong!'
else:
    print 'Wrong!'
```

没问题，但是方法太笨了。笨在你需要写两次“Wrong”。在复制上费精力可不是好事。那么怎么办？很简单：

```
if  number <=  10 and number >= 1:
    print 'Great!'
```

```
else:
    print 'Wrong!'
```

■**注意** 本例中，你还能（或者说应该）使用更简单的方法，即连接比较：1 <= number <= 10

---

and运算符就是传说中的布尔运算符。它连接两个事实值，并且在两者都为真时候返回真，反之亦然。与它同类的还有两个运算符，or和not。这三个运算符就能让你随自己的愿随意联合事实值。

```
if   ((cash > price) or customer_has_good_credit) and not out_of_stock:
    give_goods()
```

短路逻辑

布尔运算符有个很有趣的特性：它们只计算需要的部分。举例来说，表达式x and y需要都为真才为真，所以如果x为假，表达式就会立刻返回false，而不管y的值。事实上，如果x为假，表达式会返回x的值——否则它就返回y的值（你能明白它是怎么达到预期效果的吗？）这种行为被称为短路逻辑（short-circuit logic）：布尔运算符经常被叫做逻辑运算符，你也看到了，第二个值有时候“被短路了”。对于or来说也是一样。在表达式x or y中，x为真时，它直接返回x值，否则返回y值。（应该明白什么意思吧？）

那么这有什么用呢？让我们假设用户可以输入他/她的名字，也可以选择什么都不输入，这个时候你要设定一个默认值“<unknown>”。你可以使用if语句，但是可以使用很简洁的方式：

```
name = raw_input('Please enter your name: ') or '<unknown>'
```

换句话说，如果raw input的返回值为真（不是空字符串），那么它的值就会赋到name上，否则将“<unknown>”赋给name。

这类短路逻辑可以用来实现C和Java中的“三重运算符（条件运算符）”。

更深入的解释，请参看Alex Mertelli的Python CookBook（http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/5231）。

断言
Assertions

这是和if语句有关的非常有用的知识，工作方式类似于（伪代码）：

```
if not   condition:
    crash program
```

那么，究竟为什么要出现上面这种情况呢？简单说就是因为相比较错误条件在之后的时间浮现来说，现在直接让程序崩溃更好。最基本的，你需要某个肯定为真的东西。在语句中的关键字就是断言。

```
>>> age = 10
>>> assert 0   < age < 100
>>> age = -1
>>> assert 0   < age < 100
Traceback (most recent   call last):
   File "<stdin>", line   1, in ?
AssertionError
```

如果你确定程序中的某某一定为真才能让程序工作正常的话，断言就有用了，它可以作为断点便用。

条件后可以添加字符串，用来解释断言：

```
>>> age = -1
>>> assert 0   < age < 100, 'The age must be   realistic'
Traceback (most recent   call last):
   File "<stdin>", line   1, in ? AssertionError:
The age must be realistic
```

循环
Loops

现在你知道如果某条件为真（或假）时候如何做事了，但是你知道怎么才能让某事重复做几次呢？举例来说，你想要做个每月提醒你付房租的程序，但是就我们目前学到的知识而言，你得这样写程序（伪代码）：

```
发邮件
等一个月
发邮件
等一个月
发邮件
等一个月
（继续下去……）
```

但是如果你想让程序继续执行直到你停止它呢？比如你想这样（还是伪代码）：

```
当我们没有停止时：
        发邮件
        等一个月
```

或者换个简单些的例子。假设你想要打印1-100的数字，你就得再次用这个笨方法：

```
print 1
print 2
print 3
```

……等等等等。但是如果你想用这种笨方法也就不会学Python，对吧？

while循环
while Loops

为了避免伪代码中演示的讨厌的代码，你可以像这样做：

```
x = 1
while x <= 100:
    print x
    x += 1
```

那么Python里面应该如何写呢？你应该猜到了：就像上面那样。不是很复杂吧？你可以用一个循环确保用户输入了名字：

```
name = ''
while not name:
    name = raw_input('Please enter your name: ')
print 'Hello, %s!' % name
```

运行这个程序看看，然后在程序问你名字时候按下回车，程序会再次询问你的名字，因为name还是空字符串，被判定为false。

■**提示** 如果你至输入一个空格作为名字又会如何？试试看。程序会接受这个名字，因为有一个空格作为内容的字符串并不是空的，所以不会判定为假。我们的小程序中因此出现了瑕疵，但是改正也很简单：只需要把`while not name`改为`while not name or name.isspace()`即可，或者可以使用`while not name.strip()`。

---

for循环
for Loops

while语句非常灵活。它可以用来在**任何条件**为真的情况下重复执行一块代码。一般情况下这样就够了，但是有些时候你还得量体裁衣。比如要为一个集合（序列和其他可迭代对象）的每个元素都执行一遍代码，这个时候你可以使用for语句：

```
words = ['this', 'is',  'an', 'ex', 'parrot']
for word in words:
    print word
```

或者：

```
numbers = [0, 1, 2, 3,  4, 5, 6, 7, 8, 9]
for number in numbers:
    print number
```

因为迭代（“循环”的另外一种说法）某范围的数字是常见情况，所以有个内建的范围函数供你使用：

```
>>> range(0,  10)
[0, 1, 2, 3,  4, 5, 6, 7, 8, 9]
```

范围的工作方式类似于切片。他们都包含第一个数（本例中为0），但是不包含最后一个数（本例中为10）。一般你要从0开始的时候就可以使用一个限制数（作为最后一个数）：

```
>>> range(10)
[0, 1, 2, 3,  4, 5, 6, 7, 8, 9]
```

■**提示** 还有个叫做xrange的函数在循环中类似于range函数，但是range函数一次创建整个序列，而xrange一次只创建一个数。当你想要迭代一个**巨大**的序列时候xrange会很有效，不过一般情况下你不需要过多关注它。

下面的程序会打印1到100的数字：

```
for number in range(1,101):
    print number
```

它比之前的while循环更简洁。

■**提示** 如果能使用for循环，就尽量不用while循环。

迭代字典元素
Iterating Over Dictionaries

一个简单的语句就能循环字典的所有键，就像处理序列一样：

```
d = {'x': 1,  'y': 2, 'z': 3}
for key in d:
    print key, 'corresponds to',  d[key]
```

Python2.2之前，你还需要字典方法来获得键（因为直接迭代字典是不允许的）。如果你只需要值，你可以使用d.values替代d.keys。你可能会记得d.items将键值对作为远足返回，循环的一大好处就是你可以使用序列进行解包：

```
for key, value in d.items():
    print key, 'corresponds to',  value
```

你可以使用iterkeys（等同于loop）、itervalues或者iteritems方法让迭代变得更有效。（这些方法不返回列表，而是迭代器，第九章内会解释）。

■**注意** 字典元素的顺序通常是没有定义的。换句话说，迭代的时候，字典中的键和值都能保证被处理，但是顺序就不一定了。如果顺序很重要的话，你可以将键值保存在单独的列表中，在迭代前进行排序。

一些迭代功能

Some Iteration Utilities

迭代序列（或者其他可迭代对象）时候有些函数是很有用的。有些函数位于itertools模块（第九章介绍），但是有些内建函数也非常有用。

平行迭代
Parallel Iteration

有些时候你可能想同时迭代两个序列。比如有两个列表：

```
names = ['anne', 'beth', 'george', 'damon']
ages = [12, 45, 32, 102]
```

如果你想要打印名字和对应的年龄，你**可以**这样做：

```
for i in range(len(names)):
    print names[i], 'is', ages[i], 'years old'
```

这里我使用i作为loop循环索引的变量名。

而内建的zip函数可以用来做平行迭代，可以把两个序列“压缩（zip）”在一起，然后返回一个元组的列表：

```
>>> zip(names, ages)
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
```

现在我们可以在循环中解包：

```
for name, age in zip(names, ages):
    print name, 'is', age, 'years old'
```

zip函数也可以对其他很多序列使用。关于它很重要的一点是zip可以应付不等长的序列：当短序列“用完”的时候就会停止：

```
>>> zip(range(5), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

我不推荐在上面的代码中用range替换xrange——尽管我们只需要前五个数字，range会把计算所有的数字，要花费很长的时间。而xrange就没这个问题了，它只会计算前五个数字。

数值迭代
Numbered Iteration

有些时候你想要迭代序列中的对象，同时还要获取当前对象的索引。举例来说，你要在一个字符串列表中替换所有包含“xxx”的字符串。实现的方法可能有很多，所以你想要这样做：

```
for string in strings:
    if 'xxx' in string:
        index = strings.index(string)
        strings[index] = '[censored]'
```

没问题，但是在替换前要搜索给定的字符串似乎没必要。如果你不替换的话，搜

索的步骤还会返回给你错误的索引（之前同一个词的索引）。一个比较好的版本如下：

```
index = 0
for string in strings:
    if 'xxx' in string:
        strings[index] = '[censored]'
    index += 1
```

方法有些笨，不过可以接受。还有个解决方法，可以使用内建的enumerate函数：

```
for index, string in enumerate(strings):
    if 'xxx' in string:
        strings[index] = '[censored]'
```

这个函数可以让你迭代键值对，且自动提供索引。

翻转和排序迭代
Reversed and Sorted Iteration

让我们看看另外两个有用的函数：reversed和sorted。他们和列表的reverse和sort（sorted和sort采用同样的参数）方法类似，但是可以用在可迭代对象上，它们不会修改对象，而是返回翻转和排序后的版本：

```
>>> sorted([4, 3, 6, 8, 3])
[3, 3, 4, 6, 8]
>>> sorted('Hello, world!')
['  ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
>>> list(reversed('Hello, world!'))
['!', 'd', 'l', 'r', 'o', 'w', '  ', ',', 'o', 'l', 'l', 'e', 'H']
>>> ''.join(reversed('Hello, world!'))
'!dlrow ,olleH'
```

可以看到，尽管sorted返回列表，reversed却返回一个奇怪的可迭代对象。它们具体的含义你就不用担心了，你大可使用循环以及join这类方法，没有问题。不过你不能直接使用索引、切片以及调用list方法，所以你可以使用list类型进行转换，例子中已经给出。

阻断循环
Breaking Out of Loops

一般来说，循环会一直执行到条件为假时，或者到序列元素用完时。但是有些时候你可能会想要打断一个循环，进行新的迭代（新一“轮”的代码执行），或者仅仅就是想结束循环。

break

结束（跳出）循环可以使用break。假设你想要找到100内的最大平方数，那么你开始从100往下迭代到0.当你找到一个平方数时候就不需要继续了，所以你可以跳出循环：

```
from math import sqrt
for n in range(99, 0, -1):
```

```
root = sqrt(n)
if root == int(root):
    print n
    break
```

如果你执行这个程序的话，会打印出81，然后程序停止。注意，我给range函数增加了第三个参数——表示**步数（step）**，给定负值的话就会像例子中一样负向迭代。它也可以用来跳过数字：

```
>>> range(0,  10, 2)
[0, 2, 4, 6,  8]
```

continue

continue语句比break用的要少得多。它会让当前的迭代结束，“跳”到另外一个循环的开始。它的最基本的意思是“跳过剩余的循环体，但是不结束循环”。当你的循环体很大而且复杂的时候会很有用，有些时候你可能会要跳过它——这个时候你可以使用continue：

```
for x in seq:
    if condition1: continue
    if condition2: continue
    if condition3: continue

    do_something()
    do_something_else()
    do_another_thing()
    etc()
```

很多时候，只要使用if就好：

```
for x in seq:
    if not (condition1  or condition2 or condition3):
        do_something()
        do_something_else()
        do_another_thing()
        etc()
```

尽管continue非常有用，它却不是最基本和实质性的。 break语句却是你应该使用的，因为在while True语句中你会经常使用它。下一节会介绍：

while True/break习语
The while True/break Idiom

Python中的while和for循环非常灵活，但是一旦使用while你就会碰到一个难题。比如你想要处理用户输入的单词，并且在用户部署入世后结束程序。一个方法可能象下面这样：

```
word = 'dummy'
while word:
    word = raw_input('Please enter a word:  ')
    # do something with the word:
```

```
    print 'The word was ' + word

    Here is an example
session: Please enter   a word:
 first The word was   first
Please enter   a word: second
The word was   second
Please enter   a word:
```

用起来和你想象的差不多。（大概你还能做些比直接打印出单词来更多的工作）。但是你也看到了，代码有些丑。使用哑值（dummy value）就是工作没有尽善尽美的标志。让我们使者避免：

```
word = raw_input('Please enter a   word: ')
while word:
    # do something with the word:
    print 'The word was ' + word
    word = raw_input('Please enter a word:   ')
```

哑值不见了。但是我重复了一个语句两次（这样也不好）：我要用一样的赋值语句在两个地方调用两次raw_input。能不能避免呢？我可以使用while True/break惯用语句：

```
while True:
    word = raw_input('Please enter a word:   ')
    if not word: break
    # do something with the word:
    print 'The word was ' + word
```

■**注意** 习语（idiom）是指知道某语言的人们假定了解的做某事的通常方法

---

while True的部分给你一个永远不会自己停止的循环。但是你可以在loop内部加入if语句，在条件满足时候调用break语句。这样一来你就可以在循环内部任何地方而不是在开头（像普通的while循环一样）终止循环。if/break语句自然地将循环分为两部分：第一部分关注于设定和处理（可以直接从一般的循环语句中拷过来），其他的部分用于使用在第一部分内初始化好的数据，提供给循环条件真值。

尽管你应该对过多使用break关注（因为可能会让你的循环难以阅读），这个特殊的技术在大多数Python程序员（包括你）中间使用的非常普遍，并且会让你称心如意。

循环中的else 子句
else Clauses in Loops

当你在循环内使用break语句，通常是因为你“找到”了某物或者因为某事“发生”了。你要跳出时候做点什么是很简单的（比如print n），但是有些时候你想要在**没有**跳出时候做些事情。那么你怎么判断出来呢？你可以使用布尔变量，在循环前设定为False，然后当你跳出后设定为True。然后你再使用if语句查看循环是否跳出了：

```
broke_out =
```

```
False for x in seq:
    do_something(x)
    if condition(x):
        broke_out = True
        break
    do_something_else(x)
if not  broke_out:
    print "I  didn't break out!"
```

更简单的方式是在循环中增加一个else子句——它仅在你没有调用break时候执行。让我们用这个方法把刚才的例子重写：

```
from math import sqrt
for n in range(99, 81,  -1):
    root = sqrt(n)
    if root == int(root):
        print n

break else:
    print "Didn't find  it!"
```

注意我改变了第二个限制为81，以测试else子句。如果你执行程序的话，它会打印出“Didn't find it!”，因为（就像你在break那节看到的一样）100内最大的平方数是81。for和while循环中你都可以使用continue、break和else子句。

列表推导式——轻量级循环
List Comprehension—Slightly Loopy

列表推导式（List comprehension）是又其他列表创建列表（类似于集合推导式，数学名词）。它的工作方式类似于循环，也很简单：

```
>>> [x*x for  x in range(10)]
[0, 1, 4, 9,  16, 25, 36, 49, 64,  81]
```

列表由x*x for each x in range(10)这句组成。太容易了？如果你只想打印出那些能被3整除的平方数呢？那么你可以使用模除运算符——y%3，当数字可以被3整除时候返回0。（注意x*x只有在x能被3整除时候才可以被3整除）你可以将这个语句作为推导式一部分加进去：

```
>>> [x*x for  x in range(10) if x  % 3 == 0]
[0, 9, 36, 81]
```

也可以增加更多部分：

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0,  1), (0, 2), (1, 0),  (1, 1), (1, 2), (2,  0), (2, 1), (2, 2)]
```

也可以和if子句联合使用，像这样：

```
>>> girls = ['alice', 'bernice',  'clarice']
```

```
>>> boys = ['chris', 'arnold', 'bob']
>>> [b+'+'+g  for b in boys for g  in girls if b[0] ==  g[0]]
['chris+clarice', 'arnold+alice', 'bob+bernice']
```

这样你就得到了那些名字首字母相同的男孩和女孩。

更优方案

男孩女孩配对的例子其实效率不高，因为它会检查所有的可能配对。Python有很多解决这个问题的方法，下面的是Alex Martelli推荐的：

```
girls = ['alice', 'bernice', 'clarice'] boys =
['chris', 'arnold', 'bob'] letterGirls = {}
for girl in girls:
    letterGirls.setdefault(girl[0], []).append(girl)
print [b+'+'+g for b in boys for  g in letterGirls[b[0]]]
```

这个程序建造了一个叫做letterGirl的字典，每个条目都把单字母作为键，以及女孩名字组成的列表作为值。（setdefault字典方法在前章中已介绍）在字典建立后，列表推导式循环整个男孩集合，并且查找那些和当前男孩名字首字母相同的女孩集合。这样列表推到时就不用常识所有的男孩女孩的自核，检查每一个首字母配对。

三人行
And Three for the Road

作为本章的结束，让我们走马观花的看二个语句：pass、del和exec。

什么都没发生！
Nothing Happened!

有些时候你需要什么都不做。这种情况不多，但是一旦出现，你就要庆幸还有pass语句：

```
>>> pass
>>>
```

似乎没什么动静。

那么你用一个什么都不做的语句究竟要干什么呢？它可以在你的代码中做占位只用。比如你想写个if循环，然后测试，但是缺少其中一个语句块的代码，考虑下面的情况：

```
if  name == 'Ralph Auldus Melish':
    print 'Welcome!'
elif name ==  'Enid':
    # Not finished yet... elif
name ==  'Bill Gates':
```

```
    print 'Access Denied'
```

代码不会执行，因为Python的空块是非法的。解决方法就是加上一个pass：

```
if   name == 'Ralph Auldus Melish':
    print 'Welcome!'
elif name ==   'Enid':
    # Not finished yet...
    pass
elif name ==   'Bill Gates':
    print 'Access Denied'
```

■**注意** 注释和pass语句的替代方案是加入一个字符串。对于那些没有完成的函数（见第六章）和类（见第七章）来说这个方法尤其有用，因为它们会扮演类似“**文档（docstring）**”的作用（第六章中会有解释）。

使用del删除
Deleting with del

一般来说，Python删除名字（或者数据结构的一部分）时候，你不需要做更多事情：

```
>>> scoundrel = {'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = scoundrel
>>> scoundrel
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> scoundrel = None
>>> robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
>>> robin = None
```

首先，robin和scoundrel都被绑定到同一个字典上。所以当我设定scoundrel为None的时候，字典通过robin还是可用的。但是当我把robin也设定为None的时候，字典就“漂”在内存里面了，没有任何名字绑定到上面。我没有任何方法获取和使用它，所以Python解释器（以其无穷的智慧）直接删除了那个字典（这个行为称为**垃圾收集（garbage collection）**。注意，我也可以使用None之外的其他值。字典也会“消失不见”。

另外一个方法就是使用del语句（我们在第二章和第四章里面用来删除序列和字典元素的语句，记得吗？），它不仅会移除一个对象的参考，也会移除那个名字本身。

```
>>> x = 1
>>> del x
>>> x
Traceback (most recent    call last):
```

```
  File "<pyshell#255>", line 1, in ?
    x
NameError: name 'x' is  not defined
```

看起来很简单，但是要理解就有些难度了。下面的例子中，x和y都指向同一个列表：

```
>>> x = ["Hello", "world"]
>>> y = x
>>> y[1] = "Python"
>>> x
['Hello', 'Python']
```

你可能认为删除x后y也就没了，但是并**不是**这样：

```
>>> del x
>>> y
['Hello', 'Python']
```

怎么搞的？x和y都是同一个列表的参考，但是删除x并不影响y。原因就是你删除的是**名称**，而不是列表本身（值）。事实上，在Python中是没有办法删除值的（你也不需要过多考虑，因为在你不用某个值得时候它会自己搞定的）。

使用 exec 和 eval 执行和估值字符串
Executing and Evaluating Strings with exec and eval

有些时候你可能想“凭空”创造Python代码然后将其作为表达式执行，这可能会触及“黑色魔法”的边缘——考虑给自己个警告吧。

■ **警告** 本节中，你会学到如何执行储存在字符串中的Python代码。这其中会有很严重的潜在安全漏洞。如果你将用户提供的一段内容中的一部分字符串作为代码执行，你将会失去对代码执行的控制，在网络应用程序——比如CGI中尤其危险，这部分内容会在第十五章介绍。

---

exec

这条语句会执行一段字符串中的语句：

```
>>> exec "print 'Hello, world!'"
Hello, world!
```

但是，使用如此简单的exec语句绝不是好事，很多情况下你想要提供程序**命名空间**，也就是它可以放置变量的地方。你要保证代码不会干扰**你的**命名空间（也就是改变你的变量），比如说，下面的语句中使用了名称sqrt：

```
>>> from math import sqrt
>>> exec "sqrt = 1"
>>> sqrt(4)
Traceback (most recent  call last):
  File "<pyshell#18>",  line 1, in ?
    sqrt(4)
TypeError: object is not callable: 1
```

你说，你会写这样的代码吗？exec语句有用的地方在于你可以凭空书写代码。如果字符串是从其他地方获得的——很有可能是用户——你几乎不能保证其中包含什么。所以为了安全起见，你可以给它个字典，起到命名空间的作用。

■**注意** 命名空间的概念，或称为作用域（scope），是非常重要的知识。下一章中你会深入学习到，但是现在你可以把它想像成保存你的变量的地方，类似于不可见的字典。所以当你执行类似x=1的语句时，你把键x和值1放在**当前的命名空间**内，一般都是全局命名空间（目前为止绝大多数时间都是如此），但是这不是必须的。

你可以将其放置在<scope>中，<scope>就是起到放置你代码字符串命名空间作用的字典。

```
>>> from math import sqrt
>>> scope = {}
>>> exec 'sqrt = 1' in  scope
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

你可以看到，潜在的破坏性代码并不会覆盖sqrt函数，原来的函数工作的好好的，而从exec中执行的变量sqrt只在它的作用域内有效。

注意，如果你想要打印作用域的话，你会看到其中包含很多东西，因为内建的名为__builtins__的字典自动包含所有的内建函数和值：

```
>>> len(scope)
2
>>> scope.keys()
['sqrt', '__builtins__']
```

eval

eval是类似于exec的内建函数。exec会执行一系列的Python**语句**，而eval会计算并且返回Python**表达式**（以字符串形式书写）的的值。（exec并不返回任何东西，因为它本身就是语句）。举例来说，你可以使用下面的代码创造个Python计算器：

```
>>> eval(raw_input("Enter an arithmetic expression: "))
Enter an arithmetic expression: 6 + 18 * 2
42
```

■**注意** 表达式eval(raw_input(...))事实上等同于input(...)。

你可以给eval提供一个命名空间，就想给exec提供一样。尽管表达式几乎不像语句一样重绑定值（事实上，你可以给eval提供**两个**命名空间，一个全局的一个局部的。全局的是字典，但是局部的可以是任何映射形式）。

■**警告** 尽管**作为惯例**来说表达式一般不重绑定变量，它们也是可以（比如调用函数重绑定全局变量）做到这一点的。所以使用eval对付一些不可信任的代码并不比exec安全。目前，在Python内没有任何执行不可信任代码的安全方式。一个可选的方案是使用Python的扩展版本，比如Jython（见第十七章），以及使用一些本地机制，比如Java的sandbox。

初探作用域

当你给exec或者eval提供命名空间时，你还可以在真正使用命名空间前放置一些值进去：

```
>>> scope = {}
>>> scope['x'] = 2
>>> scope['y'] = 3
>>> eval('x * y', scope)
6
```

同理，对于用于exec或者eval的作用域也能在另外一个上面使用：

```
>>> scope = {}
>>> exec 'x = 2' in scope
>>> eval('x*x', scope)
4
```

事实上，exec和eval并不常用，但是它们可以作为你“屁兜”里的得力工具（当然，仅仅是比喻）。

快速总结
A Quick Summary

本章中，你学习了几类语句：

打印。你可以使用print语句打印由逗号隔开的数个值。如果语句以逗号结尾，后面的打印语句会在同一行内继续打印。

导入。有些时候你不喜欢所导入函数的名字——可能会重名。你可以使用import...as...语句，进行函数的局部重命名。

赋值。你已经见识过如果解包和联合赋值了，你可以一次对多个变量赋值，通过扩张赋值可以即刻改变变量。

块。块的意思是缩排的一组语句。他们可以在条件以及循环语句中使用，后面的章节中你会看到函数和类中也可以和其他一些东西一起使用。

条件。条件语句会执行或者不执行一个块，取决于条件（布尔表达式）。几个条件可以用if/elif/else联合使用。

断言。断言简单来说就是肯定某事（布尔表达式）为真，也可在后面跟上为什么这么认为的解释。如果表达式为假，断言就会让你的程序崩溃（事实上是产生异常——第八章会介绍）。比起你的程序不知道什么时候崩溃而言，更好的方法是尽早找到错误。

循环。你可以为序列中每一个元素都执行一块语句（比如一列数字），或者在条件为真的时候继续执行。跳出语句和继续下一次迭代可以使用continue，跳出循环使用break。或者你还可以在循环结尾加上else子句，当你没有执行break循环的时候便会执行。

列表推导式。其实它不是语句——它是看起来像循环的表达式，这也是我把它归到循环语句的原因。通过列表推导式，你可以从旧列表中产生新的，对元素执行函数，过滤掉不要的元素等等。这个技术很强大，但是很多例子中会直接使用循环和条件语句（工作也能完成），程序也会更易读。

pass、del、exec和eval。pass语句什么都不做，可以作为占位语句使用。del语句用来删除变量，或者数据结构的一部分，但是不能用来删除值。exec语句用来执行内容为Python代码的字符串。内建的eval函数对写在字符串中的表达式进行计算并且返回结果。

本章内的新函数
New Functions in This Chapter

函数	描述
chr(n)	返回n所代表的字符，(0 ≤ n < 256)
eval(source[, globals[, locals]])	将字符串作为表达式计算，并且返回值
enumerate(seq)	产生(索引,值)用于迭代
ord(c)	返回单字符字符串的int值
range([start,] stop[, step])	创造整数的列表
reversed(seq)	产生序列中值的反向版本，用于迭代
sorted(seq[, cmp][, key][, reverse])	返回排序后的列表
order xrange([start,] stop[, step])	创造xrange对象用于迭代
zip(seq1, seq2,...)	创造用于平行迭代的新序列

现在学什么？
What Now?

现在你已经完全学完基本知识了。你可以写自己能想到的任何算法，也能读取参数并且打印结果。下面几章中，你将会学到帮助你创建较大程序，而不用长篇大论的知识。也就是我们所说的**抽象（abstraction）**。

第六章

抽象
Abstraction

本章内你将会学习如何把语句组织成函数，从而使得计算机可以听从你的指令做事，而且一次通知你。你不用一次次的给它信息。本章也会介绍参数（parameters）和作用域（scopes），你将会了解什么是递归并且用它书写你的程序，你还会看到函数本身是怎么像数值、字符串和其它对象一样用作参数的。

懒惰即美德
Laziness Is a Virtue

目前我们写的程序都很小，不过如果你想要做大程序的话，不久就会陷入麻烦之中。考虑下面的情况：你在一处写的代码也要在另外一处使用。举例来说吧，假设你想要写一个计算菲波纳契数列（某数字等于前两个数字之和）的程序：

```
fibs = [0, 1]
for i in range(8):
    fibs.append(fibs[-2] + fibs[-1])
```

After running this, fibs contains the first ten Fibonacci numbers:

```
>>> fibs
[0, 1, 1, 2,  3, 5, 8, 13, 21, 34]
```

如果你想要一次计算10个数字的话，没有问题。你可以使用用户输入的数字作为动态范围的长度，从而改变循环：

```
fibs = [0, 1]
num = input('How many Fibonacci numbers do  you want?  ')
for i in range(num-2):
    fibs.append(fibs[-2] + fibs[-1])
print fibs
```

■**注意** 如果你要读取字符串，可以使用raw input，之后用int函数转换为整数。

但是如果你想把数字作为他用呢？你可以在需要的时候改变循环，但是如果写个复杂一些的程序——比如下载一系列网页并且计算词频——这种时候呢？你还会写很多次代

码，每段代码应付每次的需要吗？当然不，真正的程序员不会这么做。真正的程序员都很懒。但是不是用错误的方式犯懒，换句话说就是他们不做无用的功。
那么真正的程序员怎么做？他们会让自己的程序**抽象**一些，你可以把上面的程序改写为抽象一些的版本：

```
num = input('How many numbers do   you want?   ')
print fibs(num)
```

对于这个程序的具体细节已经写的很清楚了（读入数值，然后打印结果）。事实上计算菲波纳契数列是由抽象的方式完成的：你只需要告诉计算机去做就好。你不用特别说明应该怎么做。你创建了叫做fibs的函数，然后在计算菲波纳契数列的小程序中所需要的地方使用。如果你要用很多次的话，它会省下你很多精力。

抽象和结构
Abstraction and Structure

抽象可以省很多劳力，但是它的作用还要更重要。它是让计算机程序为人类可读的关键（也是最基本的，不管你是读还是写程序）。计算机对于处理精确和具体的数据的工作是很欢迎的，但是人可就不同了。如果你问我去电影院怎么走，估计你不会希望我这么回答：“前走10步，左转90度，再走10步，右转45读，走123步。”，弄不好你就迷路了，对吧？

现在，如果我告诉你“一直沿着街走，过桥，电影院就在左手边”，这样就明白多了吧。关键在于你已经了解如何沿街走路以及过桥，不需要明确指令来指导你去做这些事。

同理于你组织计算机程序。你的程序应该是非常抽象的，就像“下载网页、计算频率，打印每个单词的频率”一样易懂。事实上，我们现在就能把这段指令翻译成Python程序：

```
page = download_page()
freqs = compute_frequencies(page)
for word, freq in freqs:
    print word, freq
```

读完之后你就知道程序干什么的了。但是你还是不能明确的说出它是**怎么**做的。你只需要告诉计算机下载网页并且计算词频，这些操作的具体细节会在其它地方写出——在独立的**函数定义**中。

创建你的函数
Creating Your Own Functions

函数是你可以调用（可能包含参数，也就是放在圆括号中的值），做出某种行动且返回值的东西。一般来说，你可以用内建的callable函数定义某物是否可调用：

```
>>> import math
>>> x = 1
>>> y = math.sqrt
>>> callable(x)
```

```
0
>>> callable(y)
1
```

如你在前一节所知，创建函数是组织程序的关键。那么你怎么定义函数呢？使用def语句：

```
def hello(name):
    return 'Hello, ' +  name + '!'
```

运行这段程序，你就得到了一个叫做hello的函数，它可以返回一个将你输入的参数作为名字的问候语。你可以像使用内建函数一样使用它：

```
>>> print hello('world')
Hello, world!
>>> print hello('Gumby')
Hello, Gumby!
```

很精巧吧？那么想想看怎么能写个返回一列表菲波纳契数字的函数吧。简单！只需要使用刚才的代码，把从用户获取数字改为作为参数接受数字：

```
def fibs(num):
    result =  [0, 1]
    for i in  range(num-2):
        result.append(result[-2]  + result[-1])
    return result
```

执行这段语句后，你就告诉编译器如何计算菲波纳契数列了——所以现在你就不用管细节，只要用函数fibs就行：

```
>>> fibs(10)
[0, 1, 1, 2,  3, 5, 8, 13, 21, 34]
>>> fibs(15)
[0, 1, 1, 2,  3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

本例中中的名字和结果都是随便找的，但是return语句非常重要。return语句是用来从函数返回东西的（前例中hello函数）。

■**提示** 你的函数可以返回多于一个值——将其归于元组并且返回即可。

函数文档
Documenting Functions

如果你想要给函数写文档，让后来人能明白自己的程序的话，可以加入注释（以#开头）。另外一个方式就是直接写上字符串。这类字符串在其它地方可能会有用，比如在def语句后面（以及在模块或者类的开头——本书后面几章你会看到）。如果你在函数的开头写下字符串，它就会作为函数的一部分进行存储，称为“**文档字符串（docstring）**”。下面代码演示了如何给函数添加文档字符串：

```
def square(x):
    'Calculates the square of the number x.'
    return x*x
```

The docstring may be accessed like this:

```
>>> square.__doc__
'Calculates the square    of the number x.'
```

■**注意**　__doc__ 是**函数属性**。第七章中你会学到关于属性的更多知识。属性名中的双下划线表示他它是个特殊属性。这类特殊和“魔法”属性会在第九章内讨论。

有个内建的叫做help的函数非常有用。如果你在交互式解释器中使用，你可以得到函数以及它的文档字符串的信息：

```
>>> help(square)
Help on function square in module __main__:

square(x)
    Calculates the square of the    number x.
```

你会在第十章中再次看到help函数

函数并不真是函数
Functions That Aren't Really Functions

函数是数学术语，一般用于计算参数后返回值。Python内有些函数并不返回任何东西。其它语言中（比如Pascal），这类函数可能有其它名字（比如**过程procedure**），但是Python内的函数就是函数，即便它在学术上并不是。不返回任何东西的函数没有return语句。或者他们**有**return语句但是在后面没有任何单词：

```
def test():
    print 'This is printed'
    return
    print 'This is not'
```

这里的return语句只起到结束函数的作用：

```
>>> x = test()
This is printed
```

可以看到，第二行print语句被跳过了。（类似于循环中的break语句，不过这里是跳出函数）但是如果test不返回任何东西，那么x又参考到哪里呢？让我们看看：

```
>>> x
>>>
```

没东西，再仔细看看：

```
>>> print x
None
```

好熟悉啊：None。所以所有的函数的确都返回了东西：当你不告诉它们返回什么

的时候，它们就返回个None。看来我刚才的“有些函数并不真的是函数”的说法有些不公平了。

参数魔法
The Magic of Parameters

函数易懂，创建起来也不复杂。而参数用起来就有些“魔法”在里面了。首先还是来最基本的。

值从哪来？
Where Do the Values Come From?

有些时候你定义了个函数，可能会奇怪它们从哪里获取值。一般来说你不用担心，写函数只是给需要它的程序（也可能是其它程序）提供服务，你的人物就是保证函数在被提供可接受参数的时候工作正常就行，参数错误的话则显然会导致失败（一般来说这时候用断言和异常，第八章会介绍异常）。

■**注意** 你写在def语句中定义的函数后面的变量通常叫做函数的**形式参数（formal parameters）**，当你调用函数的时候提供的值是**实际参数（actual parameters）**，或者称参量。一般来说，我对两者的区别并不吹毛求疵。如果的确重要的话，我会调用实参“值”以区别形参。

我能改变参数吗？
Can I Change a Parameter?

你的函数通过自己的参数获得一系列值。那么你能改变它们吗？如果你改变了又会怎样？参数只是变量而已，所以它们会表现得和你想象的一样。在函数内赋予参数新的值不会改变外面任何东西：

```
>>> def try_to_change(n):
        n = 'Mr. Gumby'

>>> name = 'Mrs. Entity'
>>> try_to_change(name)
>>> name
'Mrs. Entity'
```

在try_to_change内，参数n获得了新值，但是你可以看到，它没有影响到name参数。毕竟它是个完全不同的变量，具体的工作方式类似于这样：

```
>>> name = 'Mrs. Entity'
>>> n = name   # 基本上等于传参时候发生的事情
>>> n = 'Mr.   Gumby' #函数内部的改变
>>> name
'Mrs. Entity'
```

结果是显而易见的。当变量n改变的时候，变量name不变。同样的，当你在函数内部把参数重绑定（赋值）的时候，函数外的变量不会受到影响。

■**注意** 参数储存在被**局部作用域（local scope）**内，本章后面会介绍。

字符串（以及数值和元组）是**不可变**的，你不能修改它们。所以它们做参数的时候没什么好说的。但是考虑一下如果你用可变的数据结构，比如列表用作参数的时候：

```
>>> def change(n):
        n[0]    = 'Mr. Gumby'

>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> change(names)
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

本例中，参数被改变了。这就是本例和前面例子中至关重要的区别。前面的例子中，我们给局部变量赋了新值，但是这个例子中我们真的**改变了**变量names所绑定的列表。有些奇怪？其实不太怪，让我们不用函数调用再做一次：

```
>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> n = names # Again pretending    to pass names as a parameter
>>> n[0] = 'Mr. Gumby'    # Change the list
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

之前你已经见过很多次这类亲故康乐。当两个变量同时参考到一个列表上的时候，它们就……同时参考到一个列表。就是这么简单。如果你想避免这种情况的话，可以做一个列表的**拷贝**。当你在序列中做切片的时候，返回的切片总是一个拷贝。所以你可以做**整个列表**的切片以便获得拷贝：

```
>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> n = names[:]
```

现在n和name包括两个**独立**（不同）的列表，值是相等的：

```
>>> n is names
0
>>> n == names
1
```

如果你现在改变n（就像你在函数中做的一样），不会影响到names：

```
>>> n[0] = 'Mr. Gumby'
>>> n
['Mr. Gumby', 'Mrs. Thing']
>>> names
['Mrs. Entity', 'Mrs. Thing']
```

再用change试一下：

```
>>> change(names[:])
>>> names
```

```
['Mrs. Entity', 'Mrs. Thing']
```

现在参数n包含一个拷贝，而你原始的列表是安全的。

■**注意** 你可能会有些疑惑：函数的局部名称——包括参数在内——并不和外面的函数名称（全局的）冲突。关于作用域的更多信息，本章后面讨论到。

为什么我想要修改参数？
Why Would I Want to Modify My Parameters?

使用函数改变数据结构（比如列表和字典）是将你的程序抽象化的好方法。假设你要写一个存储名字并且能用首、中、末名查找联系人的程序，你可以能会使用下面的数据结构：

```
ABSTRACTION storage = {}
        storage['first'] = {}
        storage['middle'] = {}
        storage['last'] = {}
```

这个数据结构的存储方式是带有三个键“first”、“middle”和“last”的字典。每个键下面都又存储一个字典。子字典中，你可以使用名字（首中末）作为键，插入联系人列表作为值。比如要把我加入这个数据结构，你可以这么做：

```
>>> me = 'Magnus Lie Hetland'
>>> storage['first']['Magnus'] =   [me]
>>> storage['middle']['Lie'] = [me]
>>> storage['last']['Hetland'] =   [me]
```

每个键下面你都存储了一列表的人。本例中，列表中只有我。

现在如果你想得到中名为Lie的人，可以这么做：

```
>>> storage['middle']['Lie']
['Magnus Lie   Hetland']
```

你也看到了，把人加到列表中的步骤比较枯燥乏味，尤其是有很多有相同名字的人要加入的时候，因为你需要扩展已经存储了那些名字的列表。举例来说，下面加入我的姐姐，而且假设你不知道数据库中已经存储了什么：

```
>>> my_sister = 'Anne Lie Hetland'
>>> storage['first'].setdefault('Anne', []).append(my_sister)
>>> storage['middle'].setdefault('Lie', []).append(my_sister)
>>> storage['last'].setdefault('Hetland', []).append(my_sister)
>>> storage['first']['Anne']
['Anne Lie Hetland']
>>> storage['middle']['Lie']
['Magnus   Lie Hetland', 'Anne   Lie Hetland']
```

想象一下如果要写个大程序来这样更新列表——很快程序就臃肿笨拙不堪了。

抽象的要点就是隐藏更新的具体细节，你可以用函数实现。下面我们就写个初始化数据结构的函数：

```
    def init(data):
        data['first'] = {}
        data['middle'] = {}
        data['last'] = {}
```

上面的代码中，我只是把初始化语句放到了函数中，你可以这样使用：

```
>>> storage = {}
>>> init(storage)
>>> storage
{'middle': {}, 'last':   {}, 'first': {}}
```

看到了吧，函数包办了初始化的工作，让代码更易读。

注意 字典的键并没有具体的顺序，所以当字典打印出来的时候，顺序是不一定的。如果在你的解释器打印出了不同的顺序，请不要担心。

在写存储名字的函数前，先写个获得名字的函数：

```
def lookup(data, label, name):
    return data[label].get(name)
```

你可以使用标签（比如“middle”）以及名字（比如“Lie”）来用lookup函数，并且会获得包含全名的列表。换句话说，如果我的名字已经存储进去了，你可以这样做：

```
>>> lookup(storage, 'middle', 'Lie')
['Magnus Lie   Hetland']
```

注意，返回的列表和储存在数据结构中的列表是相同的，所以如果你改变了这个列表，也会影响数据结构（没有查询人的时候就问题不大了：返回的是None）。

```
def store(data, full_name):
    names = full_name.split()
    if len(names) == 2: names.insert(1, '')
    labels =   'first', 'middle', 'last'
    for label, name in   zip(labels, names):
        people = lookup(data, label, name)
        if people:
            people.append(full_name)
        else:
            data[label][name] = [full_name]
```

store函数包含以下步骤：

一、你带着参数data和full_name进入函数，两个参数是你在函数外获得的一些值。
二、你的到一个被切分的全名的列表。
三、如果names的长度为2（只有首名和末名），你插入一个空字符串作为中名。
四、在标签中将字符串“first”、“middle”和“last”作为元组存储（也可以使用列表。不用括号要方便一些）。

五、你使用zip函数联合标签和函数，对于每一个（标签，名字）对，你进行以下处理：1.获得属于给定名字和标签的列表；2.将full_name添加到列表中，或者增加一个需要的新列表。

让我们试试看：

```
>>> MyNames = {}
>>> init(MyNames)
>>> store(MyNames, 'Magnus Lie Hetland')
>>> lookup(MyNames, 'middle', 'Lie')
['Magnus Lie  Hetland']
```

看起来管用，再试试：

```
>>> store(MyNames, 'Robin Hood')
>>> store(MyNames, 'Robin Locksley')
>>> lookup(MyNames, 'first', 'Robin')
['Robin Hood', 'Robin Locksley']
>>> store(MyNames, 'Mr. Gumby')
>>> lookup(MyNames, 'middle', '')
['Robin Hood', 'Robin Locksley',  'Mr. Gumby']
```

可以看到，如果某些人有相同的首中末名，你看可以一起获得他们。

■**注意** 这类程序很适合面向对象程序设计，我们将在下一章中讨论到。

如果我的参数不可变呢？
What If My Parameter Is Immutable?

有些语言（比如C++、Pascal和Ada）中，重绑定参数并且影响到函数外的变量是家常便饭的事情。Python中这是不可能的：你只能修改参数对象本身。但是如果你的参数不可变——比如数字——又怎么办呢？

不好意思，没法办。你应该做的就是返回所有需要返回的值（如果多于一个的话就作为元组）。举例来说，增加数值的函数可以这样写：

```
>>> def inc(x): return  x + 1
...

>>> foo = 10
>>> foo = inc(foo)
>>> foo
11
```

如果我真想改变值怎么办？

如果你真的想改变参数，你可以使用一点小技巧将值放置在列表中：

```
>>> def inc(x): x[0] =  x[0] + 1
...
>>> foo = [10]
>>> inc(foo)
>>> foo
[11]
```

这样就会返回新值，方法还很干净。

关键字参数和默认参数
Keyword Parameters and Defaults

目前我们所使用的参数都叫做**位置参数（positional parameters）**，因为它们的位置很重要——事实上比它们的名字更加重要。考虑下面的两个函数：

```
def hello_1(greeting, name):
    print '%s, %s!' % (greeting,  name)

def hello_2(name, greeting):
    print '%s, %s!' % (name, greeting)
```

他们都做**完全**一样的事情，只是参数名字反过来了：

```
>>> hello_1('Hello', 'world')
Hello, world!
>>> hello_2('Hello', 'world')
Hello, world!
```

有些时候（尤其是你有很多参数的时候），参数顺序是很难记住的。为了让事情简单些，你可以提供参数的**名字**：

```
>>> hello_1(greeting='Hello', name='world')
Hello, world!
```

这样顺序就完全没影响啊了：

```
>>> hello_1(name='world', greeting='Hello')
Hello, world!
```

但：

```
>>> hello_2(greeting='Hello', name='world')
world, Hello!
```

这类使用参数名提供的参数叫做**关键字参数（key parameters）**。它的用处在于可以辩明

每个参数的身份。而不用使用某些奇怪且迷惑的调用：

```
>>> store('Mr. Brainsample', 10,  20, 13, 5)
```

你可以使用：

```
>>> store(patient='Mr.  Brainsample', hour=10, minute=20, day=13, month=5)
```

尽管这样以来打的字就多了些，但是显然更清楚了。而且你还能随意改变参数顺序，也没有影响。

关键字参数最厉害的地方再就是你可以在函数中给参数提供默认值：

```
def hello_3(greeting='Hello', name='world'):
    print '%s, %s!' % (greeting,  name)
```

当参数具有默认值的时候，你在调用的时候就不用提供参数了！你可以不提供、或者提供一些以及所有的参数：

```
>>> hello_3()
Hello, world!
>>> hello_3('Greetings')
Greetings, world!
>>> hello_3('Greetings', 'universe')
Greetings, universe!
```

可以看到，对于位置参数这个方法不错，除了你要提供name时候就要greeting参数。但是如果你**只**想提供名字，而让greeting使用默认值呢？我敢肯定你已经猜到了：

```
>>>
hello_3(name='Gumby')
Hello, Gumby!
```

很简洁吧？还没完。你可以联合使用位置和关键字参数。只需要把位置参数放置在前面就行。如果不这样做，解释器会不知道它们到底是那个参数（也就是它们所处的位置）。

注意 除非你能确定自己在做什么，否则应该避免混合使用各种参数。一般来说只有在强制要求的参数比可修改的具有默认值的参数少的时候才使用。

举例来说，我们的hellow函数可能需要名字，但是允许我们（可选的）名且问候语和标点：

```
def hello_4(name, greeting='Hello', punctuation='!'):
    print '%s, %s%s' %  (greeting, name, punctuation)
```

函数可以用很多方式调用，这是其中一些：

```
>>> hello_4('Mars')
Hello, Mars!
>>> hello_4('Mars', 'Howdy')
Howdy, Mars!
```

```
>>> hello_4('Mars', 'Howdy', '...')
Howdy, Mars...
>>> hello_4('Mars', punctuation='.')
Hello, Mars.
>>> hello_4('Mars', greeting='Top of the morning to ya') Top
of the morning to ya, Mars!
>>> hello_4()
Traceback (most recent  call last):
  File "<pyshell#64>",  line 1, in ?
    hello_4()
TypeError: hello_4() takes at least 1 argument (0 given)
```

■**注意** 如果我给name个默认值，那么最后一个语句就不会产生异常。

很灵活吧？我们也不需要做多少工作。下一节中我们可以做的**更**灵活。

收集参数
Collecting Parameters

有些时候让用户提供任意个数参数是很有用的。比如在名字存储程序（本章前面写到的）中，你每次只能存一个名字。如果能像这样存储多一些名字就更好了：

```
>>> store(data, name1,  name2, name3)
```

你可以提供任意多的参数给函数。事实上，这也是可能的。

试着这样定义函数：

```
def print_params(*params):
    print params
```

这里我只明确了一个参数，但是前面加上了个星号。这是什么意思？让我们用一个参数调用函数看看：

```
>>> print_params('Testing')
('Testing',)
```

可以看到，结果作为远足打印出，因为里面有个逗号（长度为1的元组有些怪……）。所以使用在参数定义前使用星号就能打印出元组？那么用多个参数看看：

```
>>> print_params(1, 2,  3)
(1, 2, 3)
```

参数前的星号将所有值放置在同一个元组中。收集，然后使用。不知道能不能用来联合普通参数呢……让我们再写个函数：

```
def print_params_2(title, *params):
    print title
```

```
    print params
```

试试看：

```
>>> print_params_2('Params:', 1,  2, 3)
Params:
(1, 2, 3)
```

没问题！所以星号的意思就是“收集除了位置参数外的所有参数”。我打赌如果不提供任何供收集的元素，参数就会是个空元组：

```
>>> print_params_2('Nothing:')
Nothing:
()
```

的确如此。多有用啊！那么能不能处理关键字参数（也是参数）呢？

```
>>> print_params_2('Hmm...', something=42)
Traceback (most recent  call last):
  File "<pyshell#60>",  line 1, in ?
    print_params_2('Hmm...', something=42)
TypeError: print_params_2() got an unexpected keyword argument 'something'
```

看来不行。所以我们需要另外一个能处理关键字参数的操作了。你认为会是什么？可能是**？

```
def print_params_3(**params):
    print params
```

至少解释器没有发牢骚……那么调用看看：

```
>>> print_params_3(x=1, y=2, z=3)
{'z': 3, 'x': 1, 'y': 2}
```

哈，我们得到个字典而不是元组。放一起用看看：

```
def print_params_4(x, y, z=3, *pospar, **keypar):
    print x,  y, z
    print pospar
    print keypar
```

工作起来像我们期望的一样：

```
>>> print_params_4(1, 2, 3, 5, 6, 7, foo=1, bar=2)
1 2 3


(5, 6, 7)
{'foo': 1, 'bar': 2}
>>> print_params_4(1, 2)
1 2 3
()
{}
```

联合使用这些技术，你可以做的事就多了。你可能会奇怪为什么几种技术联合起来还能够工作（或者说为什么允许这么做），试试看！（下一节中，你会看

到*和**是怎么用来进行函数外的调用的）。

现在回到原来的问题上：你怎么实现多名字存储。解决方案如下：

```
def store(data, *full_names):
    for full_name in full_names:
        names = full_name.split()
        if len(names) == 2: names.insert(1, '')
        labels = 'first', 'middle', 'last'
        for label, name in zip(labels, names):
            people = lookup(data, label, name)
            if people:
                people.append(full_name)
            else:
                data[label][name] = [full_name]
```

使用这个函数就像上一节中的只接受一个名字的函数一样简单：

```
>>> d = {}
>>> init(d)
>>> store(d,  'Han Solo')
```

但是现在你可以这样：

```
>>> store(d,  'Luke Skywalker', 'Anakin Skywalker')
>>> lookup(d, 'last', 'Skywalker')
['Luke Skywalker', 'Anakin Skywalker']
```

反转过程
Reversing the Process

现在你已经学会了如何将参数收集为元组合字典，但是事实上使用*和**的话，过程反过来也是可以的。那么函数收集的逆过程是什么样？假设我们有以下的函数：

```
def add(x, y): return x + y
```

■**注意** 你可以在operator模块中找到此函数的更有效的版本。

比如说你有个包含两个要相加数字组成的元组：

```
params = (1,  2)
```

这个过程或多或少有点像我们上一节中介绍的方法的逆过程。除了收集参数外，我们还能**分配**它们。使用星号运算符就简单了，不过是在调用而不是在定义时候使用：

```
>>> add(*params)
3
```

对于参数列表来说工作正常，因为所期望的就是列表。你可以使用同样的技术对付字典——使用双星号。假设你之前定义了hello_3，你可以这样使用：

```
>>> params =  {'name': 'Sir Robin', 'greeting': 'Well  met'}
>>> hello_3(**params)
 Well met, Sir Robin!
```

在定义或者调用时候使用星号（或者双星号）都能传递元组和字典，所以你可能没遇到什么麻烦：

```
>>> def with_stars(**kwds):
        print kwds['name'], 'is', kwds['age'], 'years old'

>>> def without_stars(kwds):
        print kwds['name'], 'is', kwds['age'], 'years old'

>>> args = {'name': 'Mr. Gumby',  'age': 42}
>>> with_stars(**args) Mr.
Gumby is  42 years old
>>> without_stars(args)
Mr. Gumby is  42 years
old
```

你可以看到，在with_stars中，我在调用和定义时候都使用了星号。而在without_starts中两处我都没用，达到了一样的效果。所以星号只在你定义函数（允许不定数目参数）**或者**调用（“分割”字典或者序列）时候才有用。

Example范例

有了这么多种提供和接受参数的方法，你很容易犯晕。所以让我们放在一起举个例子。首先，我定义了一些函数：

```
def story(**kwds):
    return 'Once upon a time, there was a ' \
           '%(job)s called %(name)s.' % kwds

def power(x,  y, *others):
    if others:

print 'Received redundant parameters:', others
 return pow(x, y)

def interval(start, stop=None, step=1):
 'Imitates range() for step >  0'
 if stop is None:            # 如果提供了stop
     start, stop = 0, start  # 使用参数
 result =  []
 i = start                   # 计算start索引
 while i < stop:             # 直到计算到stop的索引
     result.append(i)        # ...将索引加到result内...
     i +=  step              # ..用step增加索引i
 return result
```

让我们用一下：

```
>>> print story(job='king', name='Gumby')
Once upon a time, there was a king called Gumby.
>>> print story(name='Sir Robin', job='brave knight')
Once upon a time, there was a brave knight  called Sir Robin.
>>> params =  {'job': 'language',  'name': 'Python'}
>>> print story(**params)
Once upon a time, there was a language called Python.
>>> del params['job']
>>> print story(job='stroke of genius', **params)
Once upon a time, there was a stroke of genius called Python.
>>> power(2,3)
8
>>> power(3,2)
9
>>> power(y=3,x=2)
8
>>> params =  (5,) * 2
>>> power(*params)
3125
>>> power(3,  3, 'Hello, world')
Received redundant parameters: ('Hello, world',)
27
>>> interval(10)
[0, 1, 2, 3,  4, 5, 6, 7, 8, 9]
>>> interval(1,5)
[1, 2, 3, 4]
>>> interval(3,12,4)
[3, 7, 11]
>>> power(*interval(3,7))
Received redundant parameters: (5, 6)
81
```

你应该尽量试验这些函数和自己的函数，直到你自信能掌握它们的用法。

作用域
Scoping

到底什么**是**变量？你可以把它们想像成参考到值的名字。所以在x=1赋值语句后，名字x参考到值1.这就像用字典一样，键参考到值，当然你用的是个“不可见”的字典。事实上，这个比喻和真实情况很近了。内建的vars函数可以返回这个字典：

```
>>> x = 1
>>> scope = vars()
>>> scope['x']
1
```

```
>>> scope['x'] += 1
>>> x
2
```

■**警告** 一般来说，你不应该修改vars所返回的字典，因为根据官方的Python文档，结果是未定义的。换句话说，你可能得不到想要的结果。

这类“不可见字典”叫做**命名空间（namespace）**或者**作用域（scope）**。那么到底有多少个命名空间？出了全局作用域外，每个函数都会创建一个：

```
>>> def foo(): x = 42
...
>>> x = 1
>>> foo()
>>> x
1
```

这里的foo函数改变（重绑定）了变量x，但是当你看到最后的时候，x并没有变。这是因为当你调用foo的时候，**新的**命名空间就被创建了，它作用于foo**内**的块。赋值语句x=42只在内部作用域（局部命名空间）起作用，所以它并不影响外在（全局）的作用域。函数内的变量被称为**局部变量（local variables）**。参数的工作原理类似于局部变量，所以用全局变量的名字作为参数明并没有问题。

```
>>> def output(x): print x
...
>>> x = 1
>>> y = 2
>>> output(y)
2
```

目前为止一切正常。但是如果你想要在函数内部获取全部变量怎么办呢？而且你只想**读取**变量的值（也就是说你不想重绑定），一般来说是没有问题的：

```
>>> def combine(parameter): print parameter + external
...
>>> external    = 'berry'
>>> combine('Shrub')
 Shrubberry
```

遮蔽（Shaowing）的问题

读取全局变量一般来说并不是问题，但是还是有个会出麻烦的事情。如果局部变量或者参数的名和你想要读取的全局变量相同的话，你就不能直接获取了。全局变量会被局部变量**遮蔽（shadow）**。

如果的确需要的话，你还是能用galobals函数获取全局变量值的，另外一个就是vars，可以返回全局变量的字典（locals返回局部变量的字典）。举例来说，如

果前例中你有个叫做parameter的变量，你就不能使用combine函数，因为你的参数也叫这个名字。不过你能使用globals()['parameter']获取：

```
>>> def combine(parameter):
        print parameter + globals()['parameter']
...
>>> parameter = 'berry'
>>> combine('Shrub') Shrubberry
```

重绑定全局变量
Rebinding Global Variables

重绑定全局变量（将其参考到其他新值上）是另外一个要关注的话题。如果你在函数内部将值赋予一个变量，它会自动成为局部变量，除非你向Python进行声明。那么你觉得怎么才能告诉Python这是一个全局变量呢？

```
>>> x = 1
>>> def change_global():
        global x
        x = x + 1

>>> change_global()
>>> x
2
```

小菜一碟吧！

■**注意** 只有在需要的时候才使用全局变量。它们会让你的代码变得较不易读和不灵活。局部变量可以让你的代码更加抽象，因为他们是在函数中“隐藏”的。

嵌套作用域

Python的作用域可以（从Python2.2以后）被嵌套使用。也就意味着你可以（使用其他功能）像下面这样书写代码：

```
def multiplier(factor):
    def multiplyByFactor(number):
        return number*factor
    return multiplyByFactor
```

一个函数位于另外一个里面，外层函数返回里层函数。每次外层函数被调用时，里层函数都会被重定义，同时变量factor也会具有新值。使用嵌套作用域，外层局部作用域（multiplier）的变量就可以随后被里层函数调用，就像下面这样：

```
>>> double =   multiplier(2)
```

```
>>> double(5)
10
>>> triple =  multiplier(3)
>>> triple(3)
9
>>> multiplier(5)(4)
20
```

类似multiplyByFactor函数存储子封闭作用域的行为叫做封装（closure）。
如果因为某些原因你还在使用Python2.1，你需要在程序开头使用下面的语句：

```
from __future__ import  nested_scopes
```

在老版本的Python中，外层嵌套作用域的变量是不可用的，你会得到类似下面的错误：

```
>>> double =  multiplier(2)
>>> double(2)
Traceback (innermost last):
  File "<stdin>", line  1, in ?
  File "<stdin>", line  3, in multiplyByFactor
NameError: factor
```

因为老版本的Python只有拒不和全局作用域，而multiplyByFactor中的factor并不是局部变量，Python会假定它为全局变量。不过他并不是全局变量，所以你会得到异常。从封闭作用域中存储变量，你可以用默认值方法进行存储：

```
def multiplier(factor):
  def multiplyByFactor(number,  factor=factor):
      return number*factor
  return multiplyByFactor
```

这个方法起作用了，因为默认值在函数定义时候是“封冻”的。

递归
Recursion

你已经学习了很多关于创建和调用函数的知识。你还知道函数也可以调用另外的函数。如果函数调用**自己**的话**会**发生什么呢？

如果你之前没接触过这类问题，你可能会奇怪这个词什么意思啊。简单说来就是参考（或者调用）自身的意思。一个有点幽默的定义：

> *re•cur•sion \ri-'k&r-zh&n\ n: see recursion.*
> *（递归（名词）：见递归）*

递归的定义（包括递归函数定义）包括它们自身定义内容的参考。而根据你经历多少的不同，递归可能让你大伤脑筋，也可能是小菜一碟。为了深入理解它，你应该

买本计算机科学方面的好书，但是多用用Python解释器也能帮助你。

一般来说，你不会希望像“递归”的幽默定义一样进递归定义，因为那样你什么也做不了。你查找递归的意思，结果它告诉你请参看递归，无穷尽也。一个类似的函数定义如下：

```
def recursion():
    return recursion()
```

显然它**做**不了任何事情——和刚才那个递归的假定义一样傻。但是如果你运行它会发生什么事情？欢迎尝试：不一会儿，程序直接就崩溃了（发生异常）。理论上讲，它应该永远运行下去。然后每次函数调用都会用掉一点内存，在足够的函数调用发生后（在之前的调用返回后），空间就不够了，而程序以一个到达最大递归深度的错误信息结束。

这类递归叫做**无穷递归（infinite recursion）**，类似于以while True开始的无穷循环，中间没有break或者return语句。因为（理论上讲）它永远不结束，那么递归又有什么用呢。有用的递归程序包含以下几部分：

- 当函数直接返回值时有基本实例（最小可能性问题）
- 递归实例，包括一个或者多个调用问题最小部分的递归

这里的关键就是将问题解析为小部分，递归不能永远继续下去，因为希望以最小可能性问题结束，而它又是存储在基本实例中的。

所以才会让函数调用自身。但是怎么将其实现呢？做起来没有看起来这么怪。就像我刚才说的那样，每次函数被调用时候，针对这个调用的新命名空间都会创建，意味着当函数调用“自身”时候，你实际上运行的是两个不同的函数（或者说是同一个函数具有两个命名空间）。你可以将它想像成一个某种类物体和同种类的另外一个物体的对话。

两个经典：阶乘和幂
Two Classics: Factorial and Power

本节中，我们会看到两个经典的递归函数。首先，假设你想要计算数n的阶乘。n的阶乘定义为$n \times (n-1) \times (n-2) \times \ldots \times 1$。很多数学程序中都会用到它（比如计算将n个人排为一行的方式）。那么怎么计算呢？你可以使用循环：

```
def factorial(n):
    result =  n
    for i in  range(1,n):
        result *= i
    return result
```

这个方法可行而且直截了当。事实上它的过程是：首先，将result赋到n上，然后result被从1到n-1的数乘一遍，最后返回return。但是如果你喜欢的话可以用不同的方法。关键在于阶乘的数学定义，下面就是：

- 1的阶乘是1
- 大于1的数n的阶乘是n乘n-1的阶乘

你可以看到，这个定义完全符合本章开头的那个。

现在考虑如果将定义实现为函数。一旦你理解了定义本的话，事实上很简单：

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

这就是定义的直接实现。只要记住函数调用factrial(n)是和调用factorial(n-1)不同的实体就行。

考虑另外一个例子。假设你想要计算幂，就像内建的pow函数或者**运算符一样。你可以用很多种方法定义一个数的（整数）幂。让我们来个简单的吧：幂(x,n)（x为n的幂次）是x自乘n-1的结果（所以x用作乘数n次）。所以幂(2,3)是2乘自身两次：2 × 2 × 2 = 8。

实现很简单：

```
def power(x,  n):
    result =  1
    for i in  range(n):
        result *= x
    return result
```

程序很小也挺好，但是如果你想要改编为递归版本：

- 对于任何数，幂(x,0)是1
- 对于任何大于0的数，幂(x,n)是x乘幂(x,n-1)的结果

你又看到了，这个定义完全符合我们的需要。理解定义是最困难的部分——实现起来就简单了：

```
def power(x,  n):
    if n == 0:
        return 1
    else:
        return x * power(x, n-1)
```

我又一次将自己的文字描述的定义转换为了（Python）程序。

提示 如果含说着算法很复杂而且南通的话，在实现前用自己的话定义一下是很有帮助的。这类“准程序语言”被称为**伪代码（pseudocode）**。

那么递归有什么用？就不能用循环代替吗？事实上——是的，你可以在大多数情况下用循环，而且大多数例子中还会更有效率。但是很多例子中，递归更加易读——有时候**更更**易读——尤其当读程序的人懂得递归函数的定义时候。尽管你可以避开写递归程序，作为程序员你还是要理解递归算法以及其他人写的递归程序，至少要做到这点。

另外一个经典：二元搜索
Another Classic: Binary Search

作为递归练习的最后例子，让我们看看叫做**二元搜索（binary search）**的算法例子。你可能知道那个问二十个问题然后猜别人在想什么的游戏。对于大多数问题，你都可以将可能性（或多或少）减半。比如你知道答案是个人，那么你可以问“你是不是在想一个女人？”你不会上来就问”你是不是在想John Cleese？”除非你会读心术。这个例子的数学版本就是猜数字。举例来说，你的搭档可能在想一个1到100间的数字，你需要猜中它。当然，你可以猜100次，但是你真正需要多少次？

答案就是你只需要问7个问题。第一个类似于“数字是否大于50？”，如果大于，你就问“是否大于75？”，继续问下去，直到找到正确答案。这个不需考虑就能解答出来。

很多其他问题上也能用同样的方法。一个很普遍的问题就是查找一个数字是否存在于一个（排过序）的序列中，还要找到具体位置。你还可以使用同样的过程。“这个数字是否在序列正中间的右边？”如果不是的话，“那么是否在第二个四分之一范围内（左侧靠右）？”然后这样继续下去。你对数字可能存在的位置上下限心里有数，然后用每个问题继续切分可能的距离。

这个算法的本身就是递归的定义，亦可用递归实现。让我们首先重看定义，保证我们直到自己在做什么：

- 如果上下限相同，那么就是数字所在位置，返回
- 否则找到两者中点，查找数字是否在左侧或者右侧，继续查找所在的一半范围。

这个递归例子的关键就是顺序，所以当你找到中间元素的时候，你只需要将它和你所找的数字相比，如果你的数字较大，那么一定在右侧，反之在左侧。递归部分就是“继续查找所在的一半范围”，因为搜索的具体实现可能会和定义中完全相同。（注意搜索的算法返回的是数字**应该**在的位置——如果它本身不再序列中，那么所返回位置上的其实是其他数字）

```
def search(sequence, number, lower, upper):
    if lower    == upper:
        assert number == sequence[upper]
        return upper
    else:
        middle = (lower + upper)    // 2
        if number > sequence[middle]:
            return search(sequence, number, middle+1, upper)
        else:
            return search(sequence, number, lower, middle)
```

完全符合定义。如果lower==upper（下限==上限），那么返回upper，也就是上限。注意你假设（断言）你所查找的数字一定会被找到（number==sequence[upper]）。如果你没有到达基本实例的话，会继续查找中间，检查你的数字是在左边还是在右边，然后使用新的上下限继续递归调用。你可以将限制设为可选以便易用。只要在函数定义的加入判断语句即可：

```
def search(sequence, number, lower=0, upper=None):
    if upper    is None: upper = len(sequence)-1
    ...
```

现在如果你不提供限制，程序会自动设定查找范围为整个序列，看看能不能行：

```
>>> seq = [34, 67, 8, 123, 4, 100, 95]
>>> seq.sort()
>>> seq
[4, 8, 34, 67, 95, 100, 123]
>>> search(seq, 34)
2
>>> search(seq, 100)
5
```

但是你可能会问：为什么老是这么麻烦？对于一个东西来说，你可以直接使用list的index方法，如果想要自己实现的话，只要从头到尾循环迭代一下直到找到数字就行了。
当然可以，使用index没问题。但是使用循环可能没什么效率。记得我说过你需要7个问题查找100内的一个数（位置）吗？用循环的话，在最糟糕的情况下要问100个问题。没什么大不了的，你可能会这样想。但是如果列表有100,000,000,000,000,000,000,000,000,000,000,000个元素，要循环这么多次（可能对于Python的列表来说这个大小有些不现实），就“有什么大不了的”了。二元查找法只需要117个问题。很有效吧？

■**提示** 标准库中的bisect模块可以非常有效地实现二元查找。

函数到处放
Throwing Functions Around

到现在为止，你可能像使用其他对象（字符串、数值、序列等等）一样习惯来使用函数了，你以赋给它们变量、传参以及从其他函数返回它们。有些编程语言（比如Scheme或者LISP）中使用函数完成几乎所有的事情，尽管在Python（你经常会创建自己类型的对象——下一章会讲到）中你不用那么倚重函数，你也**可以**那么做。本节中会介绍一些对于“函数型编程”很有用的函数，它们是：map、filter、reduce和apply。

LAMBDA 表达式

下面的章节中，有时我会使用lambda表达式。它们是小型的匿名函数，只包含一个表达式并且返回值。lambda表达式像这样书写：

```
lambda x, y, z: x + y + z
```

第一个单词lambda是保留词（关键字），紧跟着是参数以及冒号（:），最后是表达式。
尽管lambda表达式可能有些时候很有用，你最好还是写个完整的函数出来，这是因为函数名会标识函数的作用。

map

map函数会给每个元素应用函数，把一个序列“映射（map）”到另一个上面（同等长度）。举例来说，你有一列表的数字，你希望创造另外一个每个元素值都是现在元素2倍的新列表：

```
>>> numbers = [72, 101, 108, 108, 111, 44,  32, 119, 111, 114,  108, 100,  33]
>>> map(lambda n: 2*n,  numbers)
[144, 202, 216, 216, 222, 88, 64, 238, 222, 228, 216, 200, 66]
```

1. 名称“lambda”从希腊字母λ的发音而来，在数学中用于标识匿名函数

你不需要使用lambda表达式——用命名寒素也可以：

```
>>> map(chr,  numbers)
['H', 'e', 'l', 'l', 'o', ',', '  ', 'w', 'o', 'r', 'l', 'd', '!']
```

内建的chr函数使用数字作为唯一的参数，返回在ASCII表中对应的字符。chr的反函数是ord：

```
>>> map(ord,  'Hello, world!')
[72, 101, 108, 108, 111, 44, 32,  119, 111,  114, 108,  100, 33]
```

因为字符串就是字符的序列，所以你可以直接使用map。注意返回值是列表而不是字符串。参看下面一节中关于map、filter和list的提醒。

filter

filter函其它回一个不包含你想要过滤掉的元素的列表，或者说会返回你所想要的。你可以给filter提供一个返回布尔值（事实值）的函数作为筛选现有元素的一句。如果函数返回真，那么元素就包含在返回序列中，如果返回假，元素就不会包含在其中（原始序列不会被修改）。举例来说，你想要在一个数字列表中获得偶数：

```
>>> numbers = [72, 101, 108, 108, 111, 44,  32, 119, 111, 114,  108, 100,  33]
>>> filter(lambda n: n  % 2 == 0,  numbers)
[72, 108, 108, 44, 32,  114, 108,  100]
```

lambda表达式指检查给定数字除以2后的余数是否为0（换句话说就是数字是否偶数）。

现在map和filter可以派上用场了，但是它们都是在列表推导式之前加入到Python中的，所以它们的工作都能用列表推导式完成：

```
>>> [chr(n) for n in numbers] # characters  corresponding to numbers
['H', 'e', 'l', 'l', 'o', ',', '  ', 'w', 'o', 'r', 'l', 'd', '!']
>>> [ord(c) for c in 'Hello, world!'] # numbers corresponding  to characters
[72, 101, 108, 108, 111, 44, 32,  119, 111,  114, 108,  100, 33]
```

```
>>> [n for n    in numbers if n % 2    == 0] # filters out    the odd numbers
[72, 108, 108, 44, 32,    114, 108,    100]
```

在我看来，使用列表推导式的代码在大多数情况下都比使用map和filter的易读。我当然不是让你**总用**列表推导式，关键看自己的兴趣，以及具体程序任务的需要。

■**注意** 如果你追求的是速度，你应该使用map和filter。内建函数的速度要比列表推导式快。

reduce

那么第三个函数reduce呢？这个函数比较有技巧性，我承认我很少用它。但是习惯于函数型编程的人会习惯用它。它会将序列的前两个元素用给定的函数处理，并且将它们的返回值和第三个元素继续处理直到整个序列都处理完毕，剩下一个结果。举例来说，你要计算一个序列的数字之后，你可以使用reduce函数加上lambda x,y:x+y（继续使用相同的数字）[2]：

2. 事实上，除了lambda函数外，你还能在operator模块引入针对每个内建运算符的函数。使用operator模块通常比用自己的函数更有效率。

```
>>> numbers = [72, 101, 108, 108, 111, 44,    32, 119, 111, 114,    108, 100,    33]
>>> reduce(lambda x, y: x+y, numbers)
1161
```

本例中，所有的数字都被加在一起了。这个过程类似于：

```
sum = 0
for number in numbers:
    sum = sum + number
```

原始的例子中，reduce会打理好有关求和和循环的事情，而lambda充当了num+number的角色。让我们看看到底发生了什么。下面是一个数字求和并且打印参数的函数定义：

```
def peek_sum(x, y):
    print 'Adding', x,    'and', y
    return x    + y

    Let's use this with reduce:

>>> reduce(peek_sum, [1, 2, 3, 4, 5])
Adding 1 and    2
Adding 3 and    3
Adding 6 and    4
Adding 10 and 5
15
```

当reduce处理1和2之后，将结果再加3，知道所有元素都被加过。最后打印出整个过程，以及返回的值15。

■**注意** 内建的sum函数可以从列表返回数字之和（不能用于连结字符串）。

看看另外一个例子，假设你只要两个参数中大的那个（事实上会遍历整个序列），并且用于序列中，你可以使用reduce。

```
>>> reduce(max, numbers)
119
```

max函数用于返回两个数中的大数，reduce这次的工作不是求和求到底，而是比大小比到底。让我们再看看在内部的事情：

```
def peek_max(x, y):
    print 'Finding max    of', x, 'and', y
    return max(x, y)
```

就像pekk_sum和peek_max在运行时打印出自己的参数一样，我们用reduce实现：

```
>>> reduce(peek_max, [3, 5, 2, 6, 9, 2])
Finding max of 3 and 5
Finding max of 5 and 2
Finding max of 5 and 6
Finding max of 6 and 9
Finding max of 9 and 2
9
```

可以看到，左边的参数总是目前为止的最大数，右侧参数是序列中的下一个要比较的数。

■**注意** 可以看到reduce可以替代for循环，但是它不能简化为列表推导式，因为它不返回列表。

---

apply

在结束函数型编程的话题之前，我会再介绍个内建函数apply。它使用函数作为参数并且调用函数。作为可选参数，你可以以元组方式提供位置参数，以及以字典方式提供关键词参数。如果你的参数存在字典或者元组中并且想调用函数的时候可以使用apply。

```
>>> def rectangleArea(width, height):
        return width *    height


>>> rectangle = 20, 30
>>> apply(rectangleArea, rectangle)
600
```

但是这个函数已经过时，你可以使用星号来解包参数（本章前面的“收集参数”一节中已经介绍）：

```
>>> rectangleArea(*rectangle)
600
```

尽管你基本上不会用到apply，它已经在旧程序中广泛使用，你永远不知道在读别人的代码时候会不会碰到它。

快速总结
A Quick Summary

本章内，你学习了关于抽象的一般知识，以及函数：

抽象。抽象是隐藏多余细节的艺术。你可以用定义操作细节的函数让你的程序更抽象。

函数定义。函数使用def语句定义。它们是由语句组成的块，可以从“外部世界”接受值（参数），也可以返回一个或者多个值作为运算的结果。

参数。函数从参数中收到需要的信息——也就是函数调用时候的设定的变量。Python中有两类参数：位置参数和关键字参数。参数在给定默认值时候是可选的。

作用域。变量存储在作用域（也叫做命名空间）中。Python中有两类作用域——全局作用域和拒不作用于。作用域可以嵌套。

递归。函数可以调用自身——如果它这么做了就叫做递归。一切你用递归实现的功能都可以用循环实现，但是有些时候递归函数更易读。

函数型编程。Python有一些进行函数型编程的机制。包括lambda表达式以及map、filter和reduce函数。

本章内的新函数
New Functions in This Chapter

函数	描述
map(func, seq [, seq, ...])	接受函数作为参数应用于序列中每个元素
filter(func, seq)	返回在函数中验证为真的元素的列表
reduce(func, seq [, initial])	等同于func(func(func(seq[0], seq[1]), seq[2]), ...)
sum(seq)	返回seq中所有元素的和
apply(func[, args[, kwargs]])	调用函数，可以提供参数。

现在学什么？
What Now?

下一章把抽象提升到一个新高度。通过**面向对象程序设计（object-oriented programming）**你会学到如何创建自己对象的类型（或者说类），和Python提供的机制（比如字符串、列表和字典）一起使用，还会学到如何书写更优的程序。一旦你学完下一章，你将能写些真·**大**程序而不用迷失在代码中。

第七章

更加抽象

More Abstraction

前几章中你学过了Python主要的内建对象类型（数值、字符串、列表、元组和字典），你还初识了内建函数和标准库的威力，你甚至创建了自己的函数。现在只剩下一个东西了——创建自己的对象。本章讲的就是这个内容。

你可能奇怪它有什么用。建立自己的对象类型可能很酷，但是用来做什么呢？有了字典、序列、数值和字符串，不就能让函数起作用了么？当然可以。但是创建自己的对象（尤其是叫做**类（class）**的对象）是Python的中心概念——非常中心，事实上，Python被称为**面向对象（object-oriented）**的语言（和SmallTalk、C++、Java以及其他语言一样）。本章内，你将会学到如何创建对象，以及多态、封装、方法、特性、超类以及继承的概念——你会学到很多。那么我们开始吧。

■**注意** 如果你已经熟悉面向对象程序设计的概念，你可能会了解**构造器（constructor）**。构造器在本章内不会被提到。关于它的完整讨论，请参看第九章。

对象的魔力

The Magic of Objects

面向对象程序设计中的术语**对象（object）**基本上意味着数据（特性）以及一系列可以存取、操作这些数据的方法所组成的集合。我们使用对象替代全局变量和函数的原因可能有很多，其中最重要的是对象有以下重要的益处：

- 多态（Polymorphism）
- 封装（Encapsulation）
- 继承（Inheritance）

简略来说，这几个术语意味着你可以对不同类的对象使用同样的操作，它们会像被“施了魔法一般”（多态），你可以隐藏对象是如何在外部世界工作的细节（封装），你还可以从一个普通对象建立特殊的类（继承）。

许多关于面向对象程序设计的介绍中，这几个概念的顺序是不同的。封装何继承会被首先介绍，因为他们类似于现实世界中的对象。这种方法不错，但是在我看来，面向对象程序设计最有趣的特性就是多态。（以我的经历来看）它也是大多数人犯晕的地方。所以我会

以多态开始，并且展示这一个概念就足够让你喜欢面向对象程序设计了。

多态
Polymorphism

术语多态从希腊语而来，意思是“有多种形式”。基本上来说意味着你不知道变量所参考的对象类型是什么，的对象类型是什么，你还是能对它进行操作，而根据它类型的不同也会做出不同的行为。举例来说，假设你为一个买食品的商业网站创建了一个在线支付系统。你的程序会从系统的其它部分（或者以后可能会设计的其他类似系统）收到一“购物车”的物品——你要做的就是将它们的价格求和然后使用信用卡支付。

你首先想到的可能是当程序收到货物后如何具体的表现它们。比如你想要将它们作为元组接收，像这样：

```
('SPAM', 2.50)
```

如果你需要的是描述性标签和价格的话，这样就够了。但是还是不够灵活。让我们假设有些聪明为网站做了个拍卖服务——价格在某人买下货物前都会变动。如果用户能够把对象放入他/她的购物车然后处理结帐（你负责的系统部分），然后等着价格到了满意的程序后按下“支付”按钮就好了。

但是这样一来简单的元组就不能满足需要了。为了实现这个功能，在你的代码每次询问对象的时候，它都需要检查当前的价格（通过网络的某些功能）——不能固定在元组中。你也能解决它，只要写个函数：

```
# 不要这样做
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    else:
        return magic_network_method(object)
```

■**注意** 这里用isinstance进行类型/类检查是为了说明一点——类型检查一般说来并不是什么好法子。如果能避免就尽量不用类型检查。

函数isinstance在本章后面“检查继承”一节会有介绍。

前面的代码中，我使用函数以及isinstance函数查看对象是否为元组。如果是的话，就返回它的第二个元素，否就调用一些网络方法。

假设网络部分已经存在，你已经解决了这个问题——目前为止。但是还是不是很灵活。如果某些聪明的程序员决定用十六进制数的字符串作为价格，然后存储在字典中的键“price”下面呢？没问题，你只要更新函数：

```
# 不要这样做
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    elif isinstance(object, dict):
```

```
        return int(object['price'])
    else:
        return magic_network_method(object)
```

现在是不是考虑到了所有的可能性？但是如果某些人希望加入个price存储在其他键下面的新字典呢？你又会怎么做？你可以再次更新getPrice，但是你还能做多长时间这种工作？每次有人要实现对象的不同功能时候，你都要实现在自己的模块中。但是如果你已经卖出了自己的模块并且放在了其他更酷的工程中——那么客户怎么办？显然这是个不灵活而且不切实际的解决多种行为的编码方式。

那么你应该怎么办那？你可以让对象自己进行操作。听起来很明白，但是想象事情会容易多少。每个新的类型都可以接受和计算自己的价格并且返回给你值——你要做的就是问它价格。

这时候多态（还有封装）就要出场了。你收到了个对象，不知道它是怎么实现的——它可能有多种“形式”。你所知道的就是问它价格，这就够了，这种方式你应该熟悉了：

```
>>> object.getPrice()
2.5
```

绑定到对象特性上面的函数叫做**方法（method）**。你已经在字符串、列表和字典中见过方法了。而且也见识过多态：

```
>>> 'abc'.count('a')
1
>>> [1, 2, 'a'].count('a')
1
```

如果给你个变量x，你不知道它是字符串还是列表，它调用count方法时候——不会管是什么类型（同时你提供了一个字符作为参量）。

让我们做个实验吧。标准库random包含叫做choice的函数，可以从序列中随机选出元素。让我们给你的变量赋值：

```
>>> from random import  choice
>>> x = choice(['Hello, world!',  [1, 2, 'e', 'e', 4]])
```

运行后，x可能会包含字符串“Hello, world!”，也有可能包含列表[1, 2, 'e', 'e', 4]——你不知道，也不用关心到底是什么。你要关心的就是在x里面e出现多少次，而不管x是字符串还是列表。使用刚才的count函数，你可以得到：

```
>>> x.count('e')
2
```

本例中，看来是列表胜出了。但是关键点在于你不需要检测类型：你只需要知道x有个叫做count的函数，带有一个字符作为参量，并且返回正式。如果其他人创建了自己对象的类也有个count方法，跟你是没关系的——你只要像用字符串和列表一样用他们的对象。

多态的多种形式
Polymorphism Comes in Many Forms

多态在你不知道对象到底是什么类型而又能“做点什么”时候起作用。这不仅是对方法而

言——我们已经在很多运算符和函数中使用过多台了，考虑这个例子：

```
>>> 1+2
3
>>> 'Fish'+'license'
'Fishlicense'
```

这里的加运算符对于数字（本例中为整数）和字符串（其它类型的序列也一样）都能起作用。假设你要写个叫做add的函数，可以将两个东西加在一起。你可以直接将其定义成上面的形式（功能等同但比operator模块中的add函数效率低些）。

```
def add(x, y):
    return x+y
```

对于很多类型的参量都可以用：

```
>>> add(1, 2)
3
>>> add('Fish', 'license')
'Fishlicense'
```

看起来有些傻，但是我要说的是参量可以是**任何支持加法的东西**[1]。如果你想写个打印对象长度消息的函数，就需要对象**有**个长度（len函数就可用了）。

```
def length_message(x):
    print "The length of", repr(x), "is", len(x)
```

■**注意** 第一章中说过，repr可以将Python中的值表现为字符串。

可以看到，函数使用了repr，repr是多态大户——可以对任何东西使用。让我们看看：

```
>>> length_message('Fnord')
The length of 'Fnord' is 5
>>> length_message([1,  2, 3])
The length of [1, 2, 3] is 3
```

很多函数和运算符都是多态的——可能绝大多数你用的都是，尽管你并不打算让他们支持多态的。只要使用多态函数和运算符，多态就实现了。事实上，唯一能够毁掉多态的就是使用函数显示地检查类型，比如type、isinstance以及issubclass函数等。如果可能的话，你应该尽力避免使用这些毁掉多态的方式。真正紧要的是你要对象表现如何，不管它是正确的类型（或者类）抑或不是。

■**注意** 这里所讨论的多态的形式是Python式编程的中枢，也是被称为“鸭子类型（duck typing）”的东西。这个词从俗语“如果像鸭子一样呱呱大叫（If it quacks like a duck……”

---

1. 注意这类对象只支持同类的加法。调用 add(1,'license')不会起作用。

封装
Encapsulation

封装是对全局作用域中其他区域隐藏多余信息的原则。听起来有些像多态——使用对象而不用知道其内部细节的概念很类似，因为他们都是抽象的原则——他们都会帮助你处理程序组件而不用过多关心多余细节，就像函数做的一样。

但是封装并不等同于多态。多带可以让你对于不知道是什么类（对象类型）对象进行方法调用，而封装让你可以不用关心对象内部如何建立而进行使用。听起来还是有些相似？让我们用多态而不用封装写个例子，假设你有个叫做OpenObject的类（本章后面你会学到如何写类）：

```
>>> o = OpenObject() #    This is how we create objects...
>>> o.setName('Sir Lancelot')
>>> o.getName()
'Sir Lancelot'
```

你创建了一个对象（像用函数一样调用类），然后将其绑定到变量上。你可以使用setName 和getName方法（假设已经由 OpenObject类提供）。一切看起来都很完美，但是假设o将它的名字存储在全局变量globalName 上：

```
>>> globalName
 'Sir Lancelot'
```

这就意味着你在使用OpenObject类的实例时候，不用关心globalName的内容。事实上，你需要保证没人会更改它：

```
>>> globalName = 'Sir Gumby'
>>> o.getName()
'Sir Gumby'
```

如果你创建了多个OpenObject实例的话就会出现问题，因为你会因为相同的变量而搞混：

```
>>> o1 = OpenObject()
>>> o2 = OpenObject()
>>> o1.setName('Robin Hood')
>>> o2.getName()
'Robin Hood'
```

可以看到，设定一个名字后，其他的名字也就自动设定了。这可不是你想要的结果。

你需要将对象抽象地对待。当你调用方法的时候你不用关心其他的东西，比如是否干扰了全局变量。所以你能把名字“封装”在对象内吗？没问题。你可以将其作为**特性（attribute）**。特性是和方法一样作为对象一部分存在的变量，事实上方法更像是绑定到函数上的特性（在本章的“特性、函数和方法”一节中你会看到它们重要的不同点）。

那么你可以用特性而不是全局变量重写类，并且重命名为ClosedObject，像这样：

```
>>> c = ClosedObject()
>>> c.setName('Sir Lancelot')
>>> c.getName()
```

```
'Sir Lancelot'
```

目前为止还不错。但是你也知道，值可能还是存储在全局变量中的。让我们创建另一个对象：

```
>>> r = ClosedObject()
>>> r.setName('Sir Robin')
r.getName()
'Sir Robin'
```

我们可以看到新的对象有自己的set属性。这也是我们期望的。但是第一个对象怎么了呢？

```
>>> c.getName()
'Sir Lancelot'
```

名字还在！我所做的事情就是给了对象它自己的**状态（state）**。一个对象的状态由它的特性（比如name）描述。一个对象的方法可以改变它的特性。所以就像是将一大堆函数（方法）捆在一起，并且给予访问变量（特性）的权利，它们可以在函数调用之间保持值。

私有化
In Private

但是还不过。事实上，你可以从外部访问一个对象的特性：

```
>>> c.name
'Sir Lancelot'
>>> c.name =   'Sir Gumby'
>>> c.getName()
'Sir Gumby'
```

有些程序员觉得可以这样做，但是有些人（比如SmallTalk之父，SmallTalk的对象特性只允许由同一个对象的方法访问）觉得这样就打破了封装的原则。他们认为对象的状态对于外部应该是完全隐藏（不可访问）的。你可能会奇怪为什么他们会站在如此极端的立场上。每个对象管理自己的特性不就够了吗？为什么还要从外部世界中隐藏呢？毕竟如果你直接使用ClosedObject的名字特性的话你就不用使用setName和getName方法了。

关键在于其他程序员可能不知道（可能也不应该知道）的是对象内部的事情。举例来说，ClosedObject可能会在其他对象更改自己的名字的时候，给一些管理员发送带有自己名字的邮件。这应该是setName的工作。但是如果直接用c.name设定名字呢？那就什么都没发生，Email也没发出去。为了避免这样的事情，你应该使用**私有（private）**特性，这是外部对象无法访问但是类似getName和setName这类**访问器（accessor）**能够访问的特性。

■**注意** 第九章中，你将会学到关于属性（properties）的知识，它是访问器的好选择。

并不直接支持私有方式，而是靠程序员自己掌握什么时候在外部进行特性的修改。毕竟在用一个对象前你应该知道如何使用。但是，可以用一些小技巧达到

私有特性的效果。

为了让方法或者特性变为私有（从外部无法访问），只要在它的名字前面加上双下划线：

```
class Secretive:
    def __inaccessible(self):
        print "Bet you  can't see  me..."
    def accessible(self):
        print "The secret message is:"
        self.__inaccessible()
```

现在__inaccessible从外界是无法访问的，而内部还能使用（比如从accessible）访问：

```
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent  call last):
  File "<pyshell#112>", line 1, in ?
    s.__inaccessible()
AttributeError: Secretive instance has no attribute '__inaccessible'
>>> s.accessible()
The secret message is:
Bet you can't see me...
```

尽管双下划线有些奇怪，但是看起来像是其他语言中的标准的私有方法。真正发生事情才是不标准的。类的内部定义中，所有以双下划线开始的名字都被“翻译”成前面加上下划线和类名的形式。

```
>>> Secretive._Secretive__inaccessible
<unbound method Secretive.__inaccessible>
```

如果你知道这些幕后的事情，那么你还是能在类外访问这些私有方法，尽管你不应该这么做：

```
>>> s._Secretive__inaccessible()
Bet you can't see me...
```

简而言之，你不能确保其他人不会从对象外访问函数和特性，但是这类“名称变化术”就是他们不应该访问这些函数或者特性的强有力信号。

如果你不想要用这个方法但是还想让其他对象靠边站，你可以使用单下划线。这仅仅是为了方便，而且有实际效果。举例来说，前面有下划线的名字都不会被带星号的imports语句（from module import *）导入。

■**注意** 有些语言支持多种成员变量（特性）的私有性。比如Java就有四种级别。尽管单双下划线可以让你拥有两个级别的私有性，但Python并没有真正的私有性支持。

继承
Inheritance

继承是另外一个懒惰（褒义）的行为。程序员不想把同一段代码写好几次。我们之前用函数避免了这种情况，但是我现在会提出个更微妙的问题。如果你已经有了一个类，而又想建立一个类似的呢？可能只是个有几个方法的呢？当你写新类的时候，你又不想把旧类的代码全都拷过去。比如说你有个Shape类，可以用来在屏幕上画图。

现在你想要写个叫做Rectangle的类，**也**是在屏幕上画图的，但是还能计算自己的面子。你不想把Shape里面已经写好的draw方法再写一次。那么你该怎么办？你可以让Rectangle从Shape类**继承（inherit）**方法。你可以调用Rectangle对象然后自动从Shape类调用方法。本章后面我会写出这个细节。

类和类型
Classes and Types

现在你可能对什么是类有了大概感觉——或者你已经有些不耐烦听这些东西了。在开始之前，让我们什么是类，它和类型又有什么不同（和相同点）。

类到底是什么？
What *Is* a Class, Exactly?

我已经多次提到“类”这个词，将它或多或少的视为“种类”或者“类型”的同义词。很多情况下这就是类——一种对象。所有的对象都**属于**某一个类，被称为类的**实例（instance）**。

所以举例来说，你现在往窗外看，鸟就是“鸟类”的实例。这就是一个有很多子类的一般（抽象）类：你看到的鸟可能属于子类“larches”。你可以将“鸟类”想像成所有鸟的集合，而“larches类”是其中的一个子集。当一个对象所属的类是另外一个对象所属类的子集时，前者就被称为后者的**子类（subclass）**，所以“larches类”是“鸟类”的子类。相反地，“鸟类”是“larches类”的**超类（superclass）**。

■**注意** 每天的谈话中，我们经常用复数来描述对象的类，比如“birds”或者“larches”。Python中，习惯上都是单数，并且用首字母大写来表示，比如Bird和Larch。

这样一比喻，子类和超类就容易理解了。但是在面向对象程序设计内，子类的关系是隐式的，因为一个类的定义取决于它所支持的方法。类的所有实例都会包含这些方法，所以所有**子类**的所有实例**都**有这些方法。定义子类只是个定义更多（也有可能是重载已经存在）方法的事情。

举例来说，Bird可能支持fly方法，而Penguin（Bird的子类）可能会增加个eatFish方法。当创建penguin类时，你可能会想要**重载（override）**超类的fly方法，对于Penguin实例来说，这个方法应该什么都不错，或者产生异常（见第八章），因为penguin（企鹅）不会fly（飞）。

类型和类的前世今生

旧版本的Python中，类和类型之间有很明显的区别。内建的对象有类型，你自己的对象有类。你可以创建类但是不能创建类型。最近版本的Python中，事情有了些变化。在基本类型和类之间的界限开始模糊了。你可以创建内建类型的子类（或子类型），而这些类型的行为类似于类。除非你更熟悉这门语言否则你不会注意到这一点。如果你感兴趣的话，第九章中会有关于这方面的更多信息。

创建自己的类
Making Your Own Classes

终于来了！你可以创建自己的类了！已经够热情了……让我们冷静一下——这里是个简单的类：

```
class Person:
    def setName(self, name):
        self.name = name
    def getName(self):
        return self.name
    def greet(self):
        print "Hello, world! I'm  %s." % self.name
```

这个例子包含三个方法定义，除了它们是写在class里面外一切都像是函数定义。Person当然是类的名字。class语句会像函数一样创建自己的命名空间（参看本章后面“类的命名空间”）一节。一切看起来都挺好，但是你可能会对那个self参数感到奇怪。它是对于对象自身的参考。那么是什么对象？让我们创建一些实例看看：

```
>>> foo = Person()
>>> bar = Person()
>>> foo.setName('Luke Skywalker')
>>> bar.setName('Anakin Skywalker')
>>> foo.greet()
Hello, world! I'm Luke  Skywalker.
>>> bar.greet()
Hello, world! I'm Anakin Skywalker.
```

好了，例子一目了然，应该会说明白self的用处了。当我调用foo的setName以及greet函数时候，foo自动将自己作为第一个参数进行传递——也就是传说中的self。对它你可能会有自己的叫法，但是因为它总是函数自身，所以方便起见总是被叫做self（自身）。

显然这就是self的用处和存在的必要性。没有它的话，也就没有方法会访问应该进行特性操作的对象本身。

和之前一样，外部是可以访问特性的：

```
>>> foo.name
'Luke Skywalker'
>>> bar.name  = 'Yoda'
```

```
>>> bar.greet()
Hello, world! I'm Yoda.
```

特性、函数和方法
Attributes, Functions, and Methods

self参数事实上区别了方法和函数。方法（更学术一点可以叫做**绑定**方法）可以讲它们的第一个参数绑定到所属的实例上，你自己不用提供这个参数。所以你可以将特性绑定到一个普通函数上，这样就不会有特殊的self参数了：

```
>>> class Class:
        def method(self):
            print 'I have a self!'

>>> def function():
        print "I don't..."

>>> instance   = Class()
>>> instance.method() I have a self!
>>> instance.method = function
>>> instance.method() I don't...
```

注意，self参数并不取决于调用方法的方式，目前我使用实例调用方法，你可以随意使用其它变量参考到同一个方法上：

```
>>> class Bird:
        song   = 'Squaawk!'
        def sing(self):
            print self.song

>>> bird = Bird()
>>> bird.sing() Squaawk!
>>> birdsong   = bird.sing
>>> birdsong() Squaawk!
```

尽管最后一个方法看起来有些像函数调用，但是变量birdsong是参考到绑定方法bird.sing上的，也就意味着这还是对于self参数的访问。

■**注意** 第九章中，你将会看到类是如何调用超类的方法的（特殊一点来说是超类的**构造器**）。这些方法直接通过**类**调用，它们没有绑定自己的`self`参数到任何东西上，所以叫做**非绑定**方法。

方法到处放
Throwing Methods Around

前一节中，我给你展示了绑定方法，它们就像是没有self参数的函数一样。这就意味着你可以像之前使用函数一样用各种方式使用方法，比如用map、filter和reduce（参见第六章中的“函数到处放”）。本节中，我会给你一些这方面的例子，都是很好懂的。

让我们先创建个类：

```
class FoodExpert:
```

```
def init(self):
    self.goodFood = []

def addGoodFood(self, food):
    self.goodFood.append(food)

def likes(self, x):
    return x in self.goodFood

def prefers(self, x, y):
    x_rating = self.goodFood.index(x)
    y_rating = self.goodFood.index(y)
    if x_rating > y_rating:
        return y
    else:
        return x
```

这个类比前一个例子中的代码更多，但是还是很简单。它表现为一些只喜欢某些类型食物的美食家。

init方法使用包含有空列表的，叫做goodFood的特性来初始化对象。addGoodF方法为列表增加食物，第一个增加的是美食家的最爱，下一个就是第二选择，依此类推。likes方法检查美食家是否喜欢当前的食物（是否已经添加到goodFood中），而prefers方法对给定的两类食物（都是必须是喜欢的）进行对比并且返回更喜欢的那类（取决于它们在goodFood中的位置）。

现在开始玩玩这个类吧，下面是个例子，名为FoodExpert的类被创建，并且已经初始化：

```
>>> f = FoodExpert()
>>> f.init()
>>> map(f.addGoodFood,  ['SPAM', 'Eggs', 'Bacon', 'Rat', 'Spring Surprise'])
[None, None,  None, None, None]
```

代码前两行创建FoodExpert的实例并且进行初四化，并且将其赋到f上。map使用self参数调用addGoodFood方法以绑定到f。因为这个方法并不返回任何东西，所以返回的结果是个由None组成的列表。但是f已经被更新了：

```
>>> f.goodFood
['SPAM', 'Eggs', 'Bacon', 'Rat',  'Spring Surprise']
```

让我们的美食家给大家个推荐表：

```
>>> menu = ['Filet Mignon', 'Pasta', 'Pizza', 'Eggs', 'Bacon', 'Tomato', 'SPAM']
>>> rec = filter(f.likes, menu)
>>> rec
['Eggs', 'Bacon', 'SPAM']
```

我这里做的只不过是用f.likes作为menu的筛选器，美食家不喜欢的都被丢弃了。但是如果你想知道美食家喜欢什么呢？我这里再次使用了值得信赖的reduce：

```
>>> reduce(f.prefers, rec)
'SPAM'
```

这里做的事情和第六章（“reduce”一节）里面使用reduce和max的例子基本相同。.

如果我使用了不同的美食家，以不同的爱好进行初始化，我当然会得道不同的结果。当然，方法定义都是一样的。这也是标准的函数型编程和这类使用绑定函数的准函数型编程之间的区别，方法会访问可以“个性化”它们的状态。

■**注意** 你可以使用嵌套作用域进行函数传递状态，我们在前一章中已经讨论过。

类的命名空间
The Class Namespace

下面的两个语句（几乎）等价：

```
def foo(x): return x*x
foo = lambda  x: x*x
```

两者都创建了返回参数平方的函数，而且都把变量foo绑定到函数上。foo可以由全局（模块）范围进行定义，也可是局部的函数和方法进行定义的。当你定义类的时候同样的事情也会发生，所有位于class语句中的代码都被执行为特殊的命名空间——**类命名空间（class namespace）**。这个命名空间可由类内所有成员访问。并不是所有Python程序员都知道类的定义其实就是执行过的代码范围，但是这个知识点可以用做有用的信息。比如你可以不受限制使用def语句：

```
>>> class C:
        print 'Class C  being defined...'

Class C being defined...
>>>
```

简单吧，但是看看下面的：

```
class MemberCounter:
    members = 0
    def init(self):
        MemberCounter.members +=  1

>>> m1 = MemberCounter()
>>> m1.init()
>>> MemberCounter.members
1
>>> m2 = MemberCounter()
>>> m2.init()
>>> MemberCounter.members
2
```

上面的代码中，类作用域内有一个变量被定义，类内所有成员（实例）都可以进行访问，本例中是计算类的成员数量。注意init用来初始化所有实例，第九章内我会讲到它。

对于所有实例，它们和方法一样都是可访问类作用域变量的：

```
>>> m1.members
2
>>> m2.members
2
```

那么当你在示例中重绑定成员特性呢？

```
>>> m1.members = 'Two'
>>> m1.members
'Two'
>>> m2.members
2
```

新成员值被写到了m1的特性中，重载了类范围内的变量。这个行为反映出局部和全局变量的关系。

明确超类
Specifying a Superclass

就像本章前面我们讨论的一样，子类可以延展找类的定义。你可以在类名后面将其他类名写在括号内以指定超类：

```
class Filter:
    def init(self):
        self.blocked = []
    def filter(self, sequence):
        return [x for x in sequence if x not in self.blocked]

class SPAMFilter(Filter): # SPAMFilter is a subclass of Filter
    def init(self): # Overrides init method from Filter superclass
        self.blocked = ['SPAM']
```

Filter是个一般的用于过滤序列而类，事实上它不能过滤任何东西：

```
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

Filter类的用处在于它可以用作其他类的基类（超类），比如SPAMFilter类，可以将序列中的“SPAM”过滤出去。

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(['SPAM', 'SPAM', 'SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM'])
['eggs', 'bacon']
```

注意SPAMFilter定义的两个要点：

- 我用提供新定义的方式重载（override）了Filter的init定义。
- filter方法的定义是从Filter类中拿过来（继承）的，所以你不用重写它的定义。

第二个要点揭示了为什么继承很有用：我可以写一大堆不同的过滤类，全都从Filter继承，每一个我都可以使用已经实现的filter方法。这就是我说的有用的懒惰……

调查继承
Investigating Inheritance

如果你想要查看一个类是否是另一个的子类，你可以使用内建的issubclass函数：

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter,  SPAMFilter)
False
```

如果你有个类想要知道它的基类（们），你可以直接使用它的特殊__bases属性：

```
>>> SPAMFilter.__bases__
(<class __main__.Filter at 0x171e40>,)
>>> Filter.__bases__
()
```

同样的，你还能用使用isinstance方法知道一个对象是否是一个类的实例：

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

提示 就像以前说过的一样，isinstance在大多数时候最好不要用。依赖于多态更好。

你可以看到，s是SPAMFilter类的（直接）成员，但是它也是Filter类的（间接）成员，因为SPAMFilter是Filter的子类。另外一个说法就是SPAMFilter类就是Filter类。你可以在最后一个例子中看到，isinstance对于类型也起作用，比如字符串类型（str）。

如果你只想知道一个对象属于那个类，你可以使用__class特性：

```
>>> s.__class__
<class __main__.SPAMFilter at 0x1707c0>
```

多超类
Multiple Superclasses

我敢肯定你在上一节里面注意到了有些奇怪的地方：也就是__bases__的复数形式。我也跟你说过你可以找到一个类的的基类（们），也就暗示它的基类可能会多于一个。事实上就是这样，让我们建立几个新的类：

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)

class Talker:
    def talk(self):
        print 'Hi, my value is',  self.value

class TalkingCalculator(Calculator, Talker):
    pass
```

子类（TalkingCalculator）自己不做任何事，它从自己的超类集成所有的行为。它从Calculator继承calculate和从Talker继承talk，这样它就成了会说话的计算器（talking calculator）。

```
>>> tc = TalkingCalculator()
>>> tc.calculate('1+2*3')
>>> tc.talk()
Hi, my value  is 7
```

这种行为叫做**多继承（multiple inheritance）**，是个非常有用的工具。

■**注意** 当使用多继承时，有个需要你注意的东西。如果一个方法从多个超类继承，你必须要关注一下超类的顺序（在class语句）：早一些集成的类中的方法会重载晚一些的。所以如果前例中Calculator类也有个叫做talk的方法，他就会重载（使其不可访问）Talker的talk方法。如果把它们的顺序掉过来，像这样：

```
class TalkingCalculator(Talker, Calculator): pass
```

这样Talker的talk方法就可以访问了。一般来说多继承的处理要有个“实质性（substantial）”的基类，并且增加**混合（mix-in）**类实现一些方法“修改”继承关系。如果混合类重载了基类的一些方法，那么进行重载的理由一定要是充分的。如果超类们共享一个超类，那么超类在查找给定方法或者特性时候的顺序的机制称为方法解决顺序（Method Resolution Order，MRO），这个算法相当复杂。幸运的是，它工作的很好，所以你不用过多关心。

接口和内省
Interfaces and Introspection

“接口”的概念相关于多态。当你处理多态对象时候，你只要关心它的接口（或称“协议”）——公开的方法和特性。Python中，你不用显示地指定那个方法是对象需要作为参数接收的。举例来说，你不用显示的写接口（像在Java中一样），你可以在使用对象，的

时候假定它可做何事。如果它不能的话程序就会失败。

■**注意** 这里提到了一些Python中的显示接口机制。关于这方面的更多信息，可以参考Python Enhancement Proposal number 245 (http://www.python.org/peps/pep-0245.html)。

一般来说你只需要让对象符合当前的接口（换句话说就是实现当前方法），但是如果你希望的话，你还可以更灵活一些：

除了调用方法然后期待一切顺利之外，你还能检查所需方法是否已经存在——如果答案为否的话就需要做些其他事情：

```
>>> hasattr(tc, 'talk')
True
>>> hasattr(tc, 'fnord')
False
```

上面的代码中，你会发现tc（TalkingCalculator，本章前面例子中的类）有个叫做talk的特性（包含一个方法），但是并没有fnord特性。如果你需要的话，你甚至还能检查talk特性是否可调用：

```
>>> callable(getattr(tc, 'talk',  None)) True
>>> callable(getattr(tc, 'fnord', None)) False
```

注意，我使用了getattr，而没有在if语句内使用hasattr直接存储特性，getattr允许我提供默认值（本例中为None），以便在特性不存在时使用。然后我使用callable返回对象。

■**注意** getattr的反函数是setattr，可以用来设定特性到对象：

```
>>> setattr(tc, 'name', 'Mr. Gumby')
>>> tc.name
'Mr. Gumby'
```

如果你要查看对象内所有存储的值，你可以使用__dict__特性。如果你**真的**想要找到对象是由什么创建的，你可以查看inspect模块，这意味着对于那些想要制作对象浏览器（可以让你以图形方式浏览Python对象的程序）以及其他有类似机制的程序的高级用户来说才用得到。关于探索对象和模块更多的信息，你可以参看第十章里面的“探索模块”一节。

一些关于面向对象设计的思考
Some Thoughts on Object-Oriented Design

关于面向对象设计的书籍已经很多，尽管这并不是本章所关注的主题，我还是给你一些建议：

- 归类属于一起的。如果一个函数操作一个全局变量，那么两者最好都在类内，作为特性和方法出现。
- 不要让对象过于“亲密”。方法只应该关系到特性自己的实例上。让实例管理自己的状态。
- 简单就好。让你的方法小巧。作为摘要来说，它应该在30秒内被读完（并且理解）。

当考虑你需要什么类以及类要有什么方法时，你应该尝试：

1. 写下问题的描述（程序要做什么？），把所有名词、动词和形容词加下划线。
2. 对于所有名词，用作可能的类。
3. 对于所有动词，用作可能的方法。
4. 对于所有形容词，用作所有的特性。
5. 吧所有方法和特性分配到类

现在你已经有了**面向对象模型**的草图了。你还可以考虑类之间的关系（比如继承）和要有的对象，你可以用以下步骤精炼你的模型：

6. 写下（或者想象）一系列的使用实例——也就是你的程序应用时候的场景，试着重载所有的功能。
7. 一步步考虑每个应用，保证你的模型包括所有你需要的东西。如果有些遗漏的话就添加进来。如果某处不太正确则改成。继续，直到自己满意。

当你认为已经有了可以应用的模型时，你就可以开工了。你也可以审校自己的模型——或者是程序的一部分。幸运的是，在Python中你不用过多关心这方面的事情，只要投入进去就行（如果你需要面向对象程序设计方面的指导，请参看第十九章推荐的书表）。

快速总结
A Quick Summary

本章提供给你更多关于Python语言的信息，并且介绍了几个你可能完全陌生的概念。让我帮你总结一下：

对象。对象包括特性和方法。特性只是作为对象的一部分的变量，方法则是存储在特性内的函数。（绑定）方法和其他函数的区别在于方法总是将对象作为自己的第一个参数，这个参数一般称为self。

类。类表现为对象的集合（或一类对象），每个对象（实例）都有一个类。类的主要任务是定义它的实例会用到的方法。

多态。多态是实现将不同类别和类的对象进行同样对待的特性——你不需要知道对象属于哪个类就能调用方法。

封装。对象可以在它们的内部状态内隐藏（或者说封装）。一些语言中对象的状态（特性）只对自己的方法可用。Python中，所有的特性都是公开可用的，但是程序员总是应该对直接访问对象状态谨慎对待，因为这些特性可能是不可写的。

继承。一个类可以是一个或者更多个类的子类。子类从超类继承所有方法。你可以使用多于一个的超类，这个特性可以用来调整功能的相交部分。普通的实现方式是使用

核心的超类和一个或者多个的混合类。

接口和内省。一般来说，你不需要深入对象过深。你可以靠多态调用自己需要的方法。不过如果你想要对象到底有什么方法和特性，也有些可以帮你完成工作的函数。

面向对象设计。关于如何（或者说是否应该进行）面向对象设计有很多的观点。不管你持什么观点，理解这个问题，并且创建容易理解的设计是很重要的，

本章内的新函数
New Functions in This Chapter

函数	描述
callable(object)	确定对象是否可调用（比如函数或者方法）
getattr(object, name[, default])	确定特性的值，可选提供默认值
hasattr(object, name)	确定对象是否有给定的特性
isinstance(object, class)	确定对象是否是否是类的实例
issubclass(A, B)	确定A是否为B的子类
random.choice(sequence)	从非空列表中随机选择元素
setattr(object, name, value)	设定object的给定attribute为value
type(object)	返回对象的类型

现在学什么？
What Now?

你已经学习了许多关于创建自己的对象以及它们的用处的知识了。在轻率地进军Python特殊方法的魔法阵（第九章）之前，让我们先喘口气，看看关于异常处理的一小章。

第八章

异常

Exceptions

你写程序的时候通常会考虑正常事件和意外（非正常）的情况。这类异常事件可能是错误（比如试图除0），或者是你不希望经常发生的事情。为了处理这些异常事件，你可能在所有可能发生这类事件的地方都需要用条件语句处理（比如你的程序可以检查除数是否除零）。但是，这么做可能不仅会没效率和不灵活，而且还会让程序难以阅读。你可能试图避免这些异常，希望它们不会发生，但是Python会提供给你非常强大的解决方法。

什么是异常？
What Is an Exception?

nPython用异常对象（exception object）来表现异常情况。如果异常对象并未被处理，程序会带着回溯（Traceback，错误信息）一起终止

```
>>> 1/0
Traceback (most recent   call last):
   File "<stdin>", line   1, in ?
ZeroDivisionError: integer division or modulo by zero
```

如果这些错误信息就是你能用异常做的事情，那么异常也就没意思了。事实上，每隔一场都是一些类（本例中是ZeroDivisionError）的实例，这些实例可以被升出（raise），并且可以用很多种方法进行捕捉，使得你可以捉住错误并且对其做些事情，而不是让整个程序崩溃。

本节中，你将会学到如何创建和升出你自己的异常。在下面几节中，你可以学到如何用几种方法处理异常。

有关警告

异常可以用来表现你程序中异常或者非法的状态（比如试图除零或者从不存在的文件中读取数据），而且除非你捕捉到它，否则它会终止程序。另一方面来说，警告（warning）是温和一些的错误信息，它们会提示你有些东西不太对，但是你的程序会继续运行，举例来说，试着导入regex模块：

```
>>> import regex
```

```
__
main__:1: DeprecationWarning: the regex module is deprecated;
please use the re module
>>> regex
<module 'regex' (built-in)>
```

显然解释器不认这个：regex模块已经过时，你应该使用re模块（你会在第十章中减到re模块）。但是因为有很多代码已经使用了regex模块，**要求**使用re就显得没有道理：旧代码就不能用了。所以作为替代，这里使用了警告。

如果基于某些原因，你坚持使用regex模块，你可以无视这个警告（尽管你**应该**重写你的代码）。你甚至可以把它过滤掉（使用filterwarnings函数），这样它就不会被打印出来了。

```
>>> from warnings import filterwarnings
>>> filterwarnings('ignore')
>>> import regex
```

如果你希望学习关于警告的更多知识，你可以在标准库文档中查看warnings模块的介绍：http://www.python.org/doc/lib。

让事情按你的方式出错
Making Things Go Wrong . . . Your Way

异常可以在某些东西出错时候自动被升出。在学习如何处理异常之前，让我们先看你自己如何升出异常——甚至可以创建自己类型的异常。

raise语句
The raise Statement

为了升出异常，你可以带有一个类或者实例参数使用raise语句。当使用类时，会自动创建实例，你可也可以提供在类后面提供字符串参数，使用逗号隔开。下面是一些简单的例子，使用了内建的Exception类：

```
>>> raise Exception
Traceback (most recent  call last):
  File "<stdin>", line  1, in ?
Exception
>>> raise Exception, 'hyperdrive  overload'
Traceback (most recent  call last):
  File "<stdin>", line  1, in ?
Exception: hyperdrive overload
>>> raise Exception('hyperdrive overload')
Traceback (most recent  call last):
  File "<stdin>", line  1, in ?
Exception: hyperdrive overload
```

注意 事实上，raise还有两个使用方式。参数可以是字符串，或者你可以不用参数调

用raise。使用字符串参数已经过时，而不用参数会在本章后面的“老妈，看，没参数！”一节讲到。

第一个例子（raise Exception）中升出了一个没有任何错误信息的一般异常。最后两个例子中，我增加了hyperdrive overload错误信息。你可以看到，raise class,message和raise class(message)这两种方式是等价的，他们都会升出一个带有错误信息的异常。

内建的异常类有很多。你可以在Python库参考手册的“Built-in Exceptions”一节中找到关于它们的描述。你还能自己用交互式解释器探索它们，他们都可以在exceptions模块找到。将模块的内容列表可以使用dir函数，在第十章中会讲到：

```
>>> import exceptions
>>> dir(exceptions)
['ArithmeticError', 'AssertionError', 'AttributeError', ...]
```

你的解释器中，这个名单可能要长得多——为了易读我删除了大部分名字。所有这些异常都可以用在你的raise语句中：

```
>>> raise ArithmeticError
Traceback (most recent  call last):
  File "<stdin>", line  1, in ?
ArithmeticError
```

表8-1描述了一些最重要的内建异常：

表 8-1. *一些内建异常*

类名	描述
Exception	所有异常的基类
AttributeError	当在属性参考或者复制时 被升出
IOError	在试图打开不存在文件（包括其他情况）时升出
IndexError	在使用队列中不存在的索引时升出
KeyError	在使用映射中不存在的键时升出
NameError	在找不到名字（变量）时升出
SyntaxError	在代码为错误形式时升出
TypeError	在内建操作或者函数应用于错误类型的对象时升出
ValueError	在内建操作或者函数应用于正确类型的对象，但是不合适的值时升出
ZeroDivisionError	在除法或者模除操作的第二个参数为 0时升出

自定义异常类
Custom Exception Classes

尽管内建的异常类已经包括了大部分的情况，而且在对于很多要求都已经足够了，有些时候你还是需要创建自己的异常类。比如在hyperdrive overload的例子中，如果能有个特别的HyperDriveError类来表现hyperdrive的错误状况是不是更自然一些呢？错误信息是足够了，但是你会在下一节（“捕捉异常”）中看到你可以根据异常所在的类选择性的处理当前类型的异常。所以如果你想要使用特殊的错误处理代码处理hyperdrive的错误，你就需要一个和exceptions模块独立的类。

那么如何创建自己的异常类呢？就像其他类一样——但是确保是从Exception类继承（不管是间接的或者是直接的，也就是说继承其他的内建异常类也是可以的）。那么书写一个自定义异常类差不多就像这样：

```
class SomeCustomException(Exception): pass
```

还个能做太多事，对吧？

捕捉异常
Catching Exceptions

前面我曾经提到过，关于异常的最有意思的地方就是你可以处理它们（通常叫做**诱捕（trap）**或者**捕捉（catch）**异常）。你可以使用try/except来实现。假设你创建了一个让用户输入两个数然后进行相除的程序，像这样：

```
x = input('Enter the first number: ') y =
input('Enter the second number: ') print
x/y
```

程序工作正常，直到如果用户输入0作为第二个数

```
Enter the first number: 10
Enter the second number: 0
Traceback (most recent    call last):
  File "exceptions.py", line 3, in ?
    print x/y
ZeroDivisionError: integer division or modulo by zero
```

为了捕捉异常并且做出一些错误处理（本例中只是打印了友好一些的错误信息），你可以这样重写程序：

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print "The second number can't be zero!"
```

看起来用if语句检查y值会更简单一些，本例中这样做的确是个更好的方案。但是如果你想要给程序加入更多除法，你可能就得给每个除法加个if语句。而使用try/except的话只需要一个错误处理。

老妈，看，没参数！

Look, Ma, No Arguments!

如果你捕捉到了异常，但是想要重升出（比如要传递），你可以不使用参数调用raise（你还能在捕捉到异常时显式地提供一场，在本章后面“捕捉对象”一节中会解释）。

举个可以说明这么做多有用的例子吧，考虑一下一个能“抑制”ZeroDivisionErrors（除零错误）的计算类。如果这个行为被激活，那么计算器就会打印错误信息，而不是让异常发飙。如果是让用户在互动状态下使用那么就游泳了，但是如果是在程序内部使用，升出异常会更好些。那么“抑制”机制（muffling）就可以被关掉了，下面是这样一个类的代码：

```
class MuffledCalculator:
    muffled = 0
    def calc(self, expr):
        try:
            return eval(expr)
        except ZeroDivisionError:
            if self.muffled:
                print 'Division by zero is illegal'
            else:
                raise
```

注意 如果除零行为发生而抑制机制被打开，那么calc方法会（隐式地）返回None。换句话说，如果你打开了抑制机制，你就不应依赖于返回值。

下面是这个类的使用例子，分别打开和关闭了抑制：

```
>>> calculator = MuffledCalculator()
>>> calculator.calc('10/2')
5
>>> calculator.calc('10/0') # No muffling
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "MuffledCalculator.py", line 6, in calc
    return eval(expr)
  File "<string>", line 0, in ?
ZeroDivisionError: integer division or modulo by zero
>>> calculator.muffled = 1
>>> calculator.calc('10/0')
Division by zero is illegal
```

你可以看到，当计算器没有打开抑制时，ZeroDivisionError被捕捉到但是被传递了。

不止一个except子句
More Than One except Clause

如果你运行上一节的例子并且在提示符后面输入非数值，另外一个异常就会发生：

```
Enter the first number: 10
Enter the second number: "Hello,  world!"
Traceback (most recent  call last):
  File "exceptions.py", line 4, in ?
    print x/y
TypeError: unsupported  operand type(s) for  /: 'int' and 'str'
```

因为except子句只寻找ZeroDivisionError异常，这次的错误就溜过了检查并且导致程序停止。为了捕捉这个异常，你可以直接在同一个try/except后面加个except子句：

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print "The second number can't be zero!"
except TypeError:
    print "That wasn't  a number,  was it?"
```

这次用if语句实现可就复杂了。你怎么检查一个值是否能被用在除法中？方法很多，但是目前最好的方式是直接将值用来除一下看看是否能奏效。

你还应该注意异常处理并不会将搞乱原来的代码，而增加一大堆if语句检查可能的错误情况会让代码相当难度。

用一个块捕捉两个异常
Catching Two Exceptions with One Block

如果你想要用一个块捕捉一个类型以上的异常，你可以将它们作为元组列出，像下面这样：

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except (ZeroDivisionError, TypeError):
    print 'Your numbers were bogus...'
```

上面的代码中，如果用户输入字符串或者其它什么而不是数字，抑或第二个数为0，都会打印同样的错误信息。当然，只打印一个错误信息并没什么帮助。另外一个方案就是继续询问数字直到除法可以应用。我会在本章后面的“万事大吉”一节中展示给你如何做到这点。

注意，except子句中异常周围的圆括号很重要，一个共同的错误会忽略它们，你也就不会得到自己需要的。关于这方面的解释，查看下一节“捕捉对象”。

捕捉对象

Catching the Object

如果你想要在except子句中访问一个异常，你可以使用两个参数而不是一个（注意，就算你捕捉到了N个异常，你只能提供给except语句一个参数——一个元组）。这个功能很有用，（比如）如果你想让程序继续运行，但是你又想记录下错误（比如只是打印给用户看）。下面的例子程序会打印异常（如果发生的话），但是程序会继续运行：

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except (ZeroDivisionError, TypeError), e:
    print e
```

这个小程序中except子句再次捕捉两种异常，但是因为你是显示捕捉对象本身的，你可以打印出来，用户就能看到发生了什么（本章后面你会看到一个有用得多的程序，它位于“万事大吉”一节）。

真正的全捕捉
A Real Catchall

就算程序能处理好几种异常，有些异常还会从你眼皮底下溜走。比如还用那个除法程序距离，在提示符下面直接按回车，不输入任何东西，你会得到一个类似下面这样的堆栈追踪：

```
Traceback (most recent  call last):
  File 'exceptions.py', line 3, in ?
    x = input('Enter the first number: ')
  File '<string>', line 0

    ^
SyntaxError:  unexpected EOF while parsing
```

这个异常逃过了try/except语句的检查——很正常。你无法预测会发生什么，也不能对其进行准备。这些情况中程序立刻崩溃比用并不是捕捉这些try/except语句进行隐藏更好。

但是如果你**真的**想要用一段代码捕捉**所有**异常，你可以在except子句中忽略所有的异常类：

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x
except:
    print 'Something wrong happened...'

    Now you can do practically whatever you want:

Enter the first number: "This" is *completely* illegal 123
```

```
Something wrong happened...
```

■**警告** 像这样捕捉所有异常是危险的，因为它会隐藏所有你未想到并且未做好准备处理的错误。它同样会捕捉用户终止执行的Ctrl+C企图，以及你用sys.exit函数终止程序的企图，等等。用except Exception会更好些，或者用except object,e进行一些检查。

万事大吉
When All Is Well

有些情况中，**除非**某些坏事发生，执行一段代码是很有用的，对于条件和循环语句来说，你可以加个else子句：

```
try:
    print 'A simple task'
except:
    print 'What? Something went wrong?'
else:
    print 'Ah...It went as planned.'
```

If you run this, you get the following output:

```
A simple task
Ah...It went as planned.
```

使用else子句，你可实现在本章中前面“用一个块捕捉两个异常”一节中的循环提示：

```
while 1:
    try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        value = x/y
        print 'x/y is', value
    except:
        print 'Invalid input. Please try again.'
    else:
        break
```

这里的循环只在没有异常升出的情况下被破坏（被else子句中的break子句破坏）。换句话说，就算有些什么错误情况发生，程序会继续问你要新输入。下面是运行起来的例子：

```
Enter the first number: 1
Enter the second number: 0
Invalid input. Please try again.
Enter the first number: 'foo' Enter
the second number: 'bar' Invalid
input. Please try again. Enter the
```

```
first number: baz Invalid input.
Please try again. Enter the first
number: 10
Enter the second number: 2
x/y is 5
```

之前我提到过，使用空except子句的替代方案是捕捉所有Exception类的异常（也会捕捉所有所有子类的异常）。你不能百分之百肯定你会捕捉到所有的异常，因为在你try/except语句中的代码可能会出问题，比如过时的字符串异常或者自定义的不继承Exception的异常就无法捕捉。不过如果你想要用excpet Exception的话，你可以使用“捕捉对象”一节中的技巧在你的除法程序中打印更加有用的错误信息：

```
while 1:
    try:
        x = input('Enter the first number:  ')
        y = input('Enter the second number: ')
        value = x/y
        print 'x/y is', value
    except Exception, e:
        print 'Invalid  input:', e
        print 'Please try again'
    else:
        break
```

The following is a sample run:

```
Enter the first number: 1
Enter the second number: 0
Invalid input: integer  division or modulo by zero
Please try again
Enter the first number: 'x' Enter the second number: 'y'
Invalid input: unsupported operand type(s)  for /: 'str' and 'str' Please try again
Enter the first number: foo

Invalid input: name 'foo' is not  defined
Please try again
Enter the first number: 10
Enter the second number: 2
x/y is 5
```

最后……

And Finally . .

最后，是Finally子句。你可以用它在可能的异常后做清理。它和try子句（但是不和except子句）一起使用：

```
x=None
try:
 x = 1/0
        finally:
            print 'Cleaning up...'
            del x
```

上面的代码中，finally子句中的语句肯定会被执行，不管try子句中是否发生异常（try子句之前初始化x的原因是如果不这样做，因为ZeroDivisionError的存在，x就永远不会被赋值。这样就会导致在finally子句中删除它的时候产生异常，这个异常你是**无法**捕捉的）。

如果你运行这段代码，在程序崩溃之前x清理就完成了：

```
Cleaning up...
Traceback (most recent    call last):
  File "C:\python\div.py", line 4, in ?
    x = 1/0
ZeroDivisionError: integer division or modulo by zero
```

异常和函数
Exceptions and Functions

异常和函数能很自然地一起工作。如果异常在函数内升出并且不被处理，它就会传播（起泡）至函数调用的地方。如果在那里也没有处理异常，它就会继续起泡，直到到达主程序（全局作用域）。如果那里还是没有异常处理，程序会带着错误消息和关于哪里出错的消息（堆栈追踪）中止。让我们看个例子：

```
>>> def faulty():
...     raise Exception('Something is wrong')
...
>>> def ignore_exception():
...     faulty()
...
```

```
>>> def handle_exception():
...     try:
...         faulty()
...     except:
...         print 'Exception handled'
...
>>> ignore_exception()
Traceback (most recent    call last):
  File '<stdin>', line    1, in ?
  File '<stdin>', line    2, in ignore_exception
  File '<stdin>', line    2, in faulty
Exception: Something is wrong
>>> handle_exception() Exception handled
```

你可以看到，异常通过faulty和ignore_exception不完善的传播升出了，最终导致了堆栈追踪。同样的，它也传播通过了handle_exception，但是这个函数中有try/except语句。

异常之禅
The Zen of Exceptions

异常处理并不是很复杂。如果你知道某段代码可能会导致某种异常，你也不希望程序中止，出现堆栈追踪，那么你就需要添加try/except或者try/finally语句。

有些时候，你可以以条件语句完成和异常处理同样的工作，但是条件语句可能在自然性和可读性上差些。而从另一方面来看，某些程序中使用if/else实现会比try/except雅号。让我们看几个例子。

假设你写了个字典并希望打印存储在特定键下的值——如果那个键存在的话。如果它不存在，你则不希望做任何事情。代码可能像这样写：

```
def describePerson(person):
    print 'Description    of', person['name']
    print 'Age:', person['age']
    if 'occupation' in    person:
        print 'Occupation:', person['occupation']
```

如果给程序提供包含名字Throatwobbler Mangrove和年龄42（没有职业）的字典，你会得到如下输出：

```
Description of Throatwobbler Mangrove
Age: 42
```

如果你添加了职业“camper”，你会得到如下输出：

```
Description of Throatwobbler Mangrove
Age: 42
Occupation: camper
```

代码非常直观，但是有点没效率（尽管我们这里主要关心的是代码的简洁性）。程序会两次寻找“occupation”键——一次用来查看键是否存在（条件语句中），另外一次获得值（打印）。另外一个解决方案是：

```
def describePerson(person):
    print 'Description    of', person['name']
    print 'Age:', person['age']
    try: print 'Occupation:', person['occupation']
    except KeyError: pass
```

这个程序只假定“occupation”键存在。如果你假定了一般情况，那么就会省下一些力：值会被去除然后打印——不用额外的检查它是否存在。如果键不存在，KeyError

异常会被升出，可以被except子句捕捉到。

在查看对象是否存在特定特性时候，try/except也很有用。假设你想要查看某对象是否有write特性，那么你可以这样写：

```
try: obj.write
except AttributeError:
    print 'The object is not writeable'
else:
    print 'The object is writeable'
```

这里的try子句知识访问特性而不用它做其他有用的事情。如果AttributeError异常升出，就证明对象没有这个特性，反之它就有。这是实现第七章中介绍的getattr（“接口和内省”）功能的替代方法，也是你会更喜欢的方法。其实getattr在内部也是用的这个方法：它试着访问特性并且捕捉可能升出的AttributeError异常。

注意，我们这里所获得的效率并不高（微乎其微），一般来说（除非你的程序有性能问题），你不用过多担心这类优化问题。我们在很多情况下使用try/except语句比使用if/else会更自然一些（更“Pythonic”），你应该习惯在能使用它的地方就使用。

■**提示** try/except语句在Python中的表现可以用Grace Hopper的妙语解释：“请求宽恕易于请求许可（It's easier to ask forgiveness than permission。”试着做某事并处理可能发生的错误而不是在之前做一大堆检查，这个策略可以总结为习语“看前就跳（Leap Before You Look）”。

快速总结
A Quick Summary

本章的主题如下：

异常对象。异常情况（比如发生错误）可以用异常对象表示。它们可以用几种方法处理，但是如果忽略的话就会中止你的程序。

警告。警告类似于异常，但是（一般来说）仅仅打印错误信息。

升出异常。你可以使用raise语句升出异常。它接受异常类或者异常实例作为参数。你还能提供两个参数（异常和错误信息）。如果你在except子句中不使用参数调用raise，它就会“重升出”捕捉到的异常。

自定义异常类：你可以用继承Exception类的方法创建自己的异常类

捕捉异常。你可以使用try语句的except子句捕捉异常。如果你在except子句中不特别指定异常，那么所有的异常都会被捕捉。你可以将异常放在元组中实现一个以上异常的捕捉。如果你提供except两个参数，第二个参数就会绑定到异常对象上。你可以在一个try/except语句中包含多个except子句，用来分别处理不同的异常。

else子句。你可以使用else子句作为except的补充。如果主try块中没有异常升出，else里面的语句会被执行。

finally。如果你向确保某些代码不管是否有异常升出都要执行（比如清理代码），可以使用try/finally语句。这些代码可以放置在finally子句中。注意，你不能在一个try语句中同时使用except和finally子句——但是你能将一个放置在另一个中。

异常和函数。当你在函数内升出异常时，它就会起跑到函数调用的地方（对于方法也是一样）。

本章内的新函数
New Functions in This Chapter

函数	描述
warnings.filterwarnings(action, ...)	用于过滤错误

现在学什么？
What Now?

如果你觉得这章的内容有些意外（exceptional），那么下一章的内容会很不可思议，嗯……**几乎**不可思议。

第九章

魔法方法，属性和迭代器
Magic Methods, Properties, and Iterators

在Python中,有的名字会在前面和后面都加上两个下划线，这样拼写起来很特别。你已经遇到过其中的一些（比如__future__）这样的拼写符号有特殊的标记作用，不要在你的程序中使用那样的名字。这些名字中很著名的一部分是这门语言中的一些特殊方法的名字。如果你的对象实现了那些方法中的某几个，那么那些方法会在特殊的情况下（确切地说是根据名字）被Python调用。几乎没有任何直接调用它们的需要。这章会详细的讨论一些重要的特殊方法（最引人注目的是__init__方法和一些处理对象访问的方法，这些方法允许你创建自己的序列或者是映射）。这章还包括两个相关的主题：属性（先讨论特殊方法，接着处理属性函数）和迭代器（使用特殊方法__iter__来允许迭代器在for循环中使用）你会在这章的末尾找到一些使用到那时为止你要学到的东西来解决现实中的复杂问题的例子。

在我们开始之前……
Before We Begin . . .

在Pytong2.2中，对象的工作方式改变了很多。我在在一个边角处简短的提及过这个，位置是第七章的“类型和类的过去和现在”，在这章的后面我会再次提及（在“子类化列表，字典和字符串”）。这种改变产生了一些影响，但在你是一个刚使用Python来编程的程序员来说它们是可以忽略的1。其中的一个影响是，尽管你可能在使用新版的Python，但一些特性（比如属性和**超级函数（super function）**）不会在老式的类上起作用。为了确保你的类是新型的，你应该（直接或者间接）子类化内部类（build—in class）对象。请参看下面的两个类。

```
class NewStyle(object):
more_code_here

class OldStyle:
more_code_here
```

在这两个类中，NewStyle是新式的类，OldStyle是旧式的类。

1.在Alex Martelli所著的《Python技术手册》（*Python in a Nutshell*）(O'Reilly & Associates, March 2003)的第八章有关于旧式和新式类之间区别的深入讨论。

■**注意** 子类化一个内部类的更重要的作用是能确保你正在使用元类（metalclass）就是相应的内部类使用的。元类是关于类的类——一个更高深的主题。但是你使用被称为类型的内部元类来创建一个新式类的话就更容易了。不管你是把下面的赋值语句放置到你的模块的作用域（top—level scope）还是在你的类的作用域。

```
__metaclass__=type
```

把它放置在一个模块的开头能很容易的让你的类是新式类。要了解关于元类的更多信息你可以查看Guido van Rossum写的叫做《Unifying types and classes in Python 2.2》的（技术性的）文章（http://python.org/2.2/descrintro.html）。或者你在互联网上搜索"python metaclasses."

在这本书中，我只在需要的时候（因为在Pyton2.2之前的版本没有对象这个概念）才采取传统的子类化对象。如果你要没有兼容之前旧版Python的需要，我建议你让你的类成为新式的类，并且不断是使用超级函数（会在这章的"使用超级函数"部分讨论）这样的特性。

构造函数
Constructors

我们要讨论的第一个特殊的方法是构造函数。如果你以前没有听过"构造函数"这个词，我说明一下：构造函数是一个很奇特的名字，它代表着类似于我以前在例子中使用过的那种初始化方法的方法，但构造函数是在一个对象被创建后马上被自动调用的。因此，它能替代我到目前为止所做的那些：

```
>>> f = FooBar()
>>> f.init()
```

构造函数能让它简化成：

```
>>> f = FooBar()
```

在Python中创建一个构造函数很容易。只要把init方法的名字从简单的init修改为特殊的版本__init__：

```
class FooBar:
```

```
    def __init__(self):
        self.somevar =   42

>>> f = FooBar()
>>> f.somevar
42
```

现在就很好了。但你可能想知道如果你给构造函数传几个何时参数会有情况。参见下面的：

```
class FooBar:
    def __init__(self,   value=42):
        self.somevar =   value
```

你可能不相信你能这么用，因为参数是可选的，你可能发现什么也没发生。但如果你要使用参数（或者你不让参数是可选的）的时候会发生什么？我相信你已经猜到了，但我还是给你看看结果：

```
>>> f = FooBar('This is a constructor argument')
>>> f.somevar
'This is a constructor   argument'
```

在 Python 所有的特殊函数中，__init__再大部分情况下是你使用最多的一个。

■**注意** Python 中有一个特殊方法叫做__del__，也就是析构函数。它在对象就要被被垃圾回收之前被调用。但你不能知道发生调用的具体时间。

重载构造函数
Overriding the Constructor

在第七章，你会学到继承。每个类都可能有一个或者多个超类（superclasses），它们从超类那里继承行为方式。如果一个方法在B类的一个实例中被调用，但在B类中没有找到，那么就会去它的超类A找。参考下面的两个类：

```
class A:
  def hello(self):
        print "Hello, I'm A."

class B(A):
  pass
```

A类定义了一个叫做hello的方法，然后A类被B类继承。这里有一个说明类是如何工作的例

子：

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hello, I'm A.
>>> b.hello()
Hello, I'm A.
```

因为 B 没有定义自己 hello 方法，所以当 b.hello 被调用原始的信息被打印出来。B 也能重载（override）这个方法。比如下面的例子中 B 的定义被修改了。

```
class B(A):
    def hello(self):
        print "Hello, I'm B."

    Using this definition, b.hello() will give a different result:
```

使用这个定义，b.hello()能产生一个不同的结果。

```
>>> b = B()
>>> b.hello()
Hello, I'm B.
```

总体来讲，重载是继承机制中的一个重要内容。但你更可能在处理构造函数而不是普通的函数的重载时遇到特别的问题。如果你朝能够在类的构造函数，你需要调用超类的构造函数否则就要冒着对象不能被正确初始化的风险。

参考一下 Bird 类：

```
class Bird:
    def __init__(self):
        self.hungry = 1
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = 0
        else:
            print 'No,  thanks!'
```

这个类定义所有鸟都具有的一些最基本的能力：eating。这里有一个你如何使用它的例子：

```
>>> b = Bird()
>>> b.eat()
Aaaah...
>>> b.eat()
No, thanks!
```

就像你能在这个例子中看到的，一旦鸟吃过，它就不再饥饿。现在考虑子类 songBird 。我为

它添加了唱歌的的能力。

```
class SongBird(Bird):
    def __init__(self):
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

SongBird 类就和 Bird 类一样容易使用：

```
>>> sb = SongBird()
>>> sb.sing()
 Squawk!
```

因为 SongBird 是 Bird 的一个子类，它继承了 eat 方法，但你如果调用 eat 你发现一个问题：

```
>>> sb.eat()
Traceback (most recent   call last):
  File "<stdin>", line   1, in ?
  File "birds.py", line 6, in eat
    if self.hungry:
AttributeError: SongBird instance has no attribute 'hungry'
```

异常和清楚地说明了错误：SongBird 没有 hungry 属性。为什么是这样：在 SongBird 中构造函数被重载，但新的构造函数没有任何关于初始化 hungry 属性的代码。为了达到预期的效果 SongBird 的构造函数必须调用超类 Bird 的构造函数来确保基本的初始化。有两种方法能达到这个目的：调用超类构造函数的未邦定版本，或者使用函数 super。在下一部分我会解释两者。

■**注意** 尽管现在的讨论集中在构造函数的重载上，事实上这个技术可以应用于所有方法。

调用未绑定超类构造器
Calling the Unbound Superclass Constructor

如果你认为这个部分有难度，请放松。调用超类的构造函数实际上很容易（也很有用）。我会用提供前一部分末尾提出的问题的解决方法来开始讲解。

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squawk!'
    def sing(self):
```

```
        print self.sound
```

在 SongBird 类中只添加了一行代码——Bird.__init__(self)。在我解释这句的真实含义之前让我来演示一下执行的效果。

```
>>> sb = SongBird()
>>> sb.sing()
 Squawk!
>>> sb.eat()
 Aaaah...
>>> sb.eat()
 No, thanks!
```

为什么会有这样的结果？当你调用一个实例的方法时，参数 self 会被自动的绑定到实例（这样被称为绑定方法）。你已经看到几个那样的例子了。但，如果你直接调用类的方法（比如 Bird.__init__），就没有实例会被绑定。这样你就能自由的提供你想要的 self 参数。这样的方法被称为 未绑定（unbound）,这也说明了这部分的标题。

通过提供为一个未绑定方法的 self 参数提供一个当时（current）的实例，songbird 能够完成对它的超类的初始化 （这就是说属性 hungry 能被设置）。

这项技术在大多数情况下运行良好，知道如何像这样使用未绑定方法是很重要的。如果你使用的新式的类，你应该使用其它的选择：超级函数。

使用超级函数
Using the super Function

超级函数在能在新式类中使用。当时的类和对象能作为在超级函数被调用作为它的参数，用函数返回的对象调用任何方法都是调用超类的而不是当时类的方法（current class）。因此，__init__方法能以一个普通的（绑定）方式被调用。

下面的例子是对 bird 例子的更新。注意 Bird 是 object 的子类，这样它就成为了一个新式类。

```
class Bird(object):
    def __init__(self):
        self.hungry = 1
    def eat(self):
        if self.hungry:
            print 'Aaaah...'
            self.hungry = 0
        else:
            print 'No, thanks!'

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
```

```
        self.sound = 'Squawk!'
    def sing(self):
        print self.sound
```

这个新式的版本的运行结果和旧式的一样：

```
>>> sb = SongBird()
>>> sb.sing() Squawk!
>>> sb.eat() Aaaah...
>>> sb.eat() No, thanks!
```

为什么超级函数这么“超级”？

在我看来， 超级函数比在超类中直接调用未绑定方法更直观。但这并不是它为一个优点。超级函数实际上是很智能的，因此即使你已经继承多个超类，你也只需要使用一次超级函数（但要确保所有的超类的构造函数都使用了超级函数）。一些含糊的情况下使用旧式类会很诡异（比如你的两个超类一起继承一个超类）但能被新式类用超级函数自动的处理。你不需要明白内部的工作机理，但你必须清楚地知道：在大多数情况下，调用超类的未绑定的构造函数（或者其他的方法）是很更高级的。

那么超级函数返回什么？一般来说你不用担心，你就假装它返回你需要的超类。它真正返回的是超类对象，它可以帮你搞定方法解析。当你访问它的特性时候，它会在所有的超类（和超超类，等等……）中寻找特性（或者升出 AttributeError 异常）。

成员访问
Item Access

尽管__init__是目前为止你遇到的最重要的方法，但还有一些其他的能让你得到很酷的效果。特殊方法中有一个在本部分讨论，它能让你创建像序列和映射一样的对象。

实现基本的序列和映射的原则（protocol）很简单，但如果要实现全部的功能就需要实现很多特殊函数。幸运的是，有一些捷径，我将讨论它们。

■**注意** 原则在 Python 中使用的很多，它被用来描述管理行为的一些格式。这有点类似于前面提过的接口（inerface）的概念。原则说明了你应该实现的方法和这些方法应该如何运行。因为 Python 中的多态（polymorphism）是基于对象的行为（而不是基于像它所在的类或者它的基类这样的祖先（ancestry））。这是一个重要的概念：在其他的语言中对象可能被要求属于某一个类，或者被要求实现一些接口，但 Python 中只是简单的要求它遵守几个给定的原则。因此，为了成为一个序列，你所需要做的只是遵循序列的原则。

基本的序列和映射原则
The Basic Sequence and Mapping Protocol

序列和映射是对象的集合.为了实现它们基本的行为(原则),如果你的对象时不能改变的,你需要使用两个特殊函数,如果是能改变的则需要使用四个特殊函数.

__len_(self)：这个方法应该返回集合中所含的对象的数量。对于一个序列来说，这就是元素的个数；对于一个映射，则是键—值对的数量。如果__len__返回 0（并且你没有重载实现__nonzero__），对象会被当作一个布尔变量中的 false 值（这在空的列表，元组，字符串和字典也一样）。

__getitem__(self,key): 这个方法是返回所给键对应的值。对于一个序列，键应该是一个从 0 到 n-1 的整数（或者像后面所说的那样是负数），n 是序列的长度；对于一个映射，你能使用任何一种键。

__setitem__(self,key,value)：这个方法应该按一定的规则存储和键相关的值，它和__getitem__是反向的。当然，你只是为可以修改的对象定义这个方法。

__delitem__(self,key):这个方法在一部分对象上使用 del 语句时被调用，同时必须删除和元素相关的键。这个方法也是为可修改的对象定义的（并不是删除全部的对象而是一些你想要移除的元素）。

一些在这些方法上附加的其他的要求：

- 对于一个序列，如果键时一个负整数，那么要从末尾开始计数。换句话说就是 x[-n]和 x[len(x)-n]是一样的。
- 如果键是不合适的类型，会出现一个 TypeError
- 如果序列的索引是正确的类型，但超出了范围，应该出现一个 IndexError

让我们实践一下——看看如果创建一个无穷序列：

```
def checkIndex(key):
```

所给的键是能接受的索引吗？

为了能被接受，键应该是一个非负的整数。如果它不是一个整数，会出现 TypeError；如果是负的，则出现 IndexError

```
    if not isinstance(key, (int,  long)): raise TypeError
    if key<0: raise IndexError

class ArithmeticSequence:
```

```
    def __init__(self,   start=0, step=1):
        """
```

初始化算术序列：

start—序列中的第一个值

step—两个相近值的不同

changed—用户修改的值的集合。

```
        self.start = start                          # Store the   start value
        self.step = step                            # Store the   step value
        self.changed =   {}                         # No items have been modified

    def __getitem__(self, key):
        """
        Get an item from the arithmetic sequence.
        """
        checkIndex(key)

        try:    return self.changed[key]            # Modified?
        except KeyError:                            # otherwise...
            return self.start + key*self.step       # ...calculate the value

    def __setitem__(self, key, value):
        """
```

修改算术序列中的一个值

```
        """
        checkIndex(key)

        self.changed[key] = value                   # Store the   changed value
```

这实现的是一个算术序列，它其中的值是后者比前者大一个常数。第一个值是由构造函数 start（默认为 0 ）给出的，而值与值之间的间隔是 step 给出的。（默认为 1）。用户能通过 changed 来使用存储在字典中的修改规则来修改存储的值，如果元素没有被修改那就计算 start+key*step。

下面是如何使用这个类的例子：

```
>>> s = ArithmeticSequence(1, 2)
>>> s[4]
9
```

```
>>> s[4] = 2
>>> s[4]
2
>>> s[5]
11
```

注意因为我没有实现__del__所以删除元素是非法的：

```
>>> del s[4]
Traceback (most recent    call last):
  File "<stdin>", line    1, in ?
AttributeError: ArithmeticSequence instance has no attribute '__delitem__'
```

这个类没有__len__方法，因为它是无限长的。

如果使用了一个非法的索引，就会出现 TypeError，如果索引的格式是正确的但超出了范围（在最后两个例子中的负数）则会出现 IndexError：

```
>>> s["four"]
Traceback (most recent    call last):
  File "<stdin>", line    1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 10, in checkIndex
    if not isinstance(key, int):    raise TypeError
TypeError
>>> s[-42]
Traceback (most recent    call last):
  File "<stdin>", line    1, in ?
  File "arithseq.py", line 31, in __getitem__
    checkIndex(key)
  File "arithseq.py", line 11, in checkIndex
    if key<0: raise IndexError
IndexError
```

索引检查是由我为此写的 checkIndex 函数实现的。

为什么不进行类型检查？

有一件事可能会让你吃惊，即checkIndex函数是isinstance的应用（你很少使用它，因为类型检查和Python中的多态目标背道而驰）。我使用它是因为Python的语言规范上说明索引必须是整数（包括长整数）。遵守标准是使用类型检查的很少的正确原因之一。

■**注意** 只要你愿意你也能模拟切片。实现切片需要支持方法__getitem__。（切片对象在Python 库参考（http://python.org/doc/lib）的 2.1 节“Built-in Functions”中 slice 函数部分有介绍。

子类化列表，字典和字符串
Subclassing list, dict, and str

到目前为止已经向你介绍了四个基本的序列/映射原则，官方语言规范要求其他的特殊方法和普通方法必须被实现（参见《Python Reference Manual》中的“Emulating Container Types”部分 http://www.python.org/doc/ref/sequence-types.html），包括__iter__方法，我在这章稍后的迭代器那部分说明。实现所有的这些方法（为了让你自己的对象和列表或者字典一样具有多态）是一件繁重的工作并且很难做的完全正确。如果你只要让客户使用一个功能，那么其他的方法就不用处理。这就是程序员的懒散（也叫公共心理（commen sense））。

那么你要做什么呢？继承是一个很神奇的词。为什么在你继承的时候不重实现（reimplement）继承来的所有的事情？标准库有三个关于序列和映射原则（参见“UserList, UserString, and UserDict”部分的工具条）的准备好被使用的（ready-to-use）实现，在更新版本的 Python 中，你能子类化内部类型。（注意如果你的类的行为和默认的很接近这就很有用，如果你重实现大部分方法写一个新的类可能更容易。）

USERLIST, USERSTRING和USERDICT

标准库包含三种模型分别是：UserList, UserString, and UserDict，每个都包含和模型名一样的类。这些类满足所有的序列和映射的原则。UserList
和UserString 是和普通的列表和字符串一样的定制序列，UserDict 是和普通的字典一样的定制映射。在 Python 2.2 之前，在你创建自己的映射和函数时这些都是最好的选择。在 Python 2.2 中添加了子类化内部类的功能，减弱了三个模型的作用

因此，如果你要实现一个和内部列表类型很相似的序列，你可以轻易的使用子类化列表。

■**注意** 当你子类化一个内部类型，比如列表，你就间接的子类化了对象。因此你的类就自动成为新式类，这就意味着像超级函数这样的特性能被使用。

让我们看看例子——一个带有访问计数的列表。

```
class CounterList(list):
```

```
def __init__(self, *args):
    super(CounterList,  self).__init__(*args)
    self.counter = 0
def __getitem__(self, index):
    self.counter += 1
    return  super(CounterList, self).__getitem__(index)
```

CounterList 类和它的超类的行为很相似。CounterList 类没有重载任何的方法（就像 append, extend, index）都能被直接使用。在两个被重载的方法中，super 被用来调用相应的超类的方法。

counter 为属性添加了所需的初始化，并在__getitem__更新了 counter 属性。

■**注意** 重载__getitem__不是防止用户访问的方式，因为其他的访问列表内容的途径，比如通过 pop 方法

这里是 CounterList 如何使用的例子：

```
>>> cl = CounterList(range(10))
>>> cl
[0, 1, 2, 3,  4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6,  5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3,  2, 1, 0]
>>> cl.counter
0
>>> cl[4] + cl[2]
9
>>> cl.counter
2
```

正如你看到的，CounterList 在大多数时候和列表的效果一样，但它有一个会在列表元素每次被访问或增加的 counter 特性（被初始化为 0 ），在运行了 cl[4] + cl[2]后，特性值变为 2 。

更多魔力
More Magic

对于很多目的——我在先前所演示的只是所有可能中的一小部分——都有特殊的名字。大部分的特殊方法都是很高级的用法，所以我不会在这里详细讨论。但如果你有兴趣，我举几个例子——可以让对象像函数那样被调用，影响对象的比较等等。关于特殊函数的更多内容请参考《 the Python Reference Manual》中的“Special Method Names”(http://www.python.org/doc/ref/specialnames.html).

属性
Properties

在第七章我提过 accessor 方法。 Accessor 是一个简单的方法，他能够通过 像 getHeight，setHeight 这样的名字来得到或者设置一些特性（ attribute）（可能是类的私有特性——具体请看第七章的相关部分） 。

如果在访问给定的特性时必须要采取一些行动那么像这样的封装变量（特性）就很重要。比如下面例子中的 Rectangle 类

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
```

下面的例子是如何使用这个类：

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height  = 5
>>> r.getSize()
(10, 5)
>>> r.setSize((150, 100))
>>> r.width
150
```

在上面的例子中 getSize 和 setSize 方法是一个假象的 size 类的 accessor 方法，size 类似于包害了高度和宽度的元组。这些代码不是错误的，但却是有缺陷的。当程序员使用这个类时不应该还要考虑它是怎么实现的（封装）。如果有一天你要改变类的实现，把 size 变成一个特性，这样宽和高就不会被计算，那么你就要把他们放到一个 accessor 中去，并且任何使用这个类的程序都必须改写。

客户的代码（使用你代码的代码） 应该能够用同样的方式对待你的所有特性。那么怎么解决呢？把所有的特性都放到 accessor 中去？当然那是可行的。但是果果你有很多简单的特性那样做就很不现实（愚蠢的一种表现）。如果那样做你就得写很多除了返回或者设置特性就不做任何事的 accessor。

复制—粘贴式的编程方式很明显是很糟糕的（尽管是在一些语言中对于这个特定问题很普遍）。幸运的是， Python 能为你隐藏 accessor，让你的所有特性看起来一样。这些在 accessor 中定义的特性被称为属性。实际上在 Python 中有两中机制创建属性。我会主要是讨论新的机制—只在新式类中使用的 property 函数，然后我会简单的说明一些如何使用特殊方法实现属性。

property 函数
The property Function

令人高兴的是使用 property 函数很简单。如果你已经写了一个像前面部分的 Rectangle 那样的类，你只要增加一行代码：

```
class Rectangle(object):
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
    size = property(getSize, setSize)
```

在这个新版的 Rectangle 中，一个属性被用 property 函数创建 ，其中 accessor 函数被用作参数（先是 getter 然后是 setter），这个属性命名为 size。这样，你不再需要担心是怎么实现的了，可以用同样的方式处理宽，高和尺寸。

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height   = 5
>>> r.size
(10, 5)
>>> r.size =   150, 100
>>> r.width
150
```

很明显，size 属性仍然在计算时关系到了 getSize 和 setSize。但看起来就像普通的特性一样。

■**提示** 如果你的属性有很奇怪的行为，那么要查看你的类是不是子类化对象（要

么是直接要么是间接的—或者是直接设置元类（metaclass））；如果没有，那么虽然类的大部分属性还是能工作，但有的却不能工作，这很让人迷惑。

实际上，property 函数能用 0，1，2，3 或者是 4 个参数来调用。如果没有参数，产生的属性是只读的。第三个参数（可选）是一个用于删除属性的方法（它不要参数）。第四个参数（可选）是一个说明性的字符串。Property 的四个参数分别被叫做 fget,fset,fdel 和 doc—你能使用他们作为关键字参数，如果你想要一个属性是只写的，并且有一个说明字符串。

尽管这个部分很短（只是对 porperty 函数的简单说明），但它却十分的重要。理论上说，你在新式类中应该使用属性而不是 accessor。

它是如何工作的？

有的读者很想知道 property 是如何实现它的功能的，我在这里解释一下，不感兴趣的读者可以跳过。

实际上 property 不是一个真正的函数—它是一个拥有很多特殊函数的类，是那些特殊函数完成了所有的工作。问题中的方法是__get__,__set__和__delete__。和在一起，这三个方法定义了描叙符原则（descriptor protocol）。一个实现了其中任何一个的就叫描述符（descriptor）。关于描述符的特殊之处是它们怎么被访问的。比如，读取一个属性时，如果返回的对象实现了__get__，那么就会调用__get__（结果值也会被返回），而不只是简单的返回对象。实际上这就是属性的机理，即绑定方法，静态的和类成员方法（下一部分有更多的信息）还有 super。你能在 Python 语言规范（http://python.org/doc/ref/ descriptors.html）中找到关于描述符原则的简单说明。一个更全面的信息源是 Raymond Hettinger 的《How-To Guide for Descriptors》（http://users.rcn.com/python/download/Descriptor.htm）

静态方法和类成员方法
Static Methods and Class Methods

在讨论实现属性的旧方法前，让我们稍微绕道看看另一对实现方法和新式的实现属性方法类似的特征。静态方法和类成员方法分别在创建时被放入静态方法类型（staticmethod type）和类成员类型（classmethod type）的对象中。静态方法在定义的时候没有 self 参数，能够被类本身直接的调用。类成员方法在定义时要有一个被叫做 cls 的 self 参数，你能用类的具体对象直接调用类成员方法。但 cls 参数是自动被绑定到类的，请看例子：

```
__metaclass__ = type

class MyClass:
```

```
def smeth():
    print 'This is  a static method'
smeth = staticmethod(smeth)

def cmeth(cls):
    print 'This is  a class method of',  cls
cmeth = classmethod(cmeth)
```

手动包装和替换方法的技术看起来有点单调。在 Python2.4 中，一个关于包装方法（wrapping methond）的新的语法被引入，叫做 decorators（它能对任何对象起作用，既能够用于方法也能用于函数）你能用@operator 指定一个或者更多的 decrators（多个 decrator 在应用时是按相反的顺序）。

```
__metaclass__ = type

class MyClass:

    @staticmethod
    def smeth():
        print 'This is  a static method'


    @classmethod
    def cmeth(cls):
        print 'This is  a class method of',  cls
```

Once you've defined these methods, they can be used like this (that is, without instantiating the class):

一旦你定义了这些方法,就能像下面的例子那样使用(例子中没有实例化类)

```
>>> MyClass.smeth()
This is a static method
>>> MyClass.cmeth()
This is a class method  of <class  '__main__.MyClass'>
```

静态方法和类成员方法在Python中并不是向来都很重要,主要的原因是大部分情况下你能使用函数或者邦定方法代替。在早期的版本中没有得到支持也是一个原因。因此你可能看不到两者在现存的代码中的大量应用。静态方法和类成员方法有自己的应用（比如 厂函数（factory function）），你可以好好的考虑一下使用新的技术。

__getattr__,__setattr__,和友元
__getattr__, __setattr__, and Friends

我们也能够使用旧式类来实现属性，但你必须使用特殊方法而不是property函数。下面的四个方法提供了你需要的功能（在旧式类中你只需要后三个）

__getattribute__(self,name)：当属性名被访问时自动被调用。（只能在新式类中使用）

__getattr__(self,name): 当属性名被访问，且对象没有相应的属性时被自动调用。

__setattr__(self,name,value)：当试图给属性赋值时会被自动调用。

__delattr__(self, name):当试图删除属性名时被自动调用。

尽管和使用property相比有点复杂（而且效率更低）但这些特殊方法是很强大的，因为你能在其中的一个方法中未很多属性写代码。（如果你有选择的机会，我还是坚持推荐property）

下面是`Rectangle`的例子，但这次使用的是特殊方法：

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattr__(self, name):
        if name == 'size':
            return self.width, self.height
        else:
            raise AttributeError
```

正如你看到的，这个版本的类需要注意增加的管理细节。当你思考这个例子时，请对下面的几点引起重视：

- __setattr__方法在询问的属性不是size时也会被调用。因此，这个方法必须把两方面都考虑进去：如果属性是size，那么就像前面那样表现，否则特殊方法__dict__就要被使用，该特殊方法包含一个字典，字典里面是所有的对象的属性。为了避免__setattr__方法被再次调用（这样会使程序陷入死循环），__dict__方法被用来代替普通的属性赋值操作。
- __getattr__方法在普通的属性没有被找到的时候调用，这就是说在例子中如果使用了一个不存在的属性，这个方法会引发一个AttributeError。如果你想类和hasattr或者是getattr这样内部函数一起正确的工作，__gerattr__方法就很重要。如果被使用的是size属性，那么就会使用在前面的实现中找到的语句。

其它的陷阱

就像有一个死循环和__setattr__有关系一样，还有一个陷阱和__getattribute__有关系。因为__getattribute__影响所有属性的访问（在新式类中），他也会影响对__dict__的访问！在用__getattibute__访问属性时，使用超类的__getattribute__方法（用 super）是唯一的安全途径。

迭代器
Iterators

在这个部分，我只讨论一个特殊方法——__iter__，这个方法是迭代器原则（iterattor protocol）的基础。

迭代器原则
The Iterator Protocol

迭代的意思是重复一些事很多次——就像你在循环中做的。到现在为止我只是在for循环中对序列和字典进行迭代，但实际上你能对其他的对象进行迭代：对象必须要实现了__iter__方法。

__iter__方法返回一个迭代器（iterator），所谓的迭代器就是实现了next方法（这个方法不需要任何参数）的对象。当你调用next方法，迭代器会返回它的下一个值。如果next方法被调用但迭代器没有值可以返回，就会出现一个StopIteration的异常。

你也许会问，关键是什么？为什么不就使用列表?因为列表的威力太大。如果你有一个能一个接一个的计算值的函数，你可能只是需要一个接一个的计算——而不是一次性的存在列表里。如果有很多值，那么列表就会占用太多的内存。但还有其他的理由：使用迭代器更通用，更简单。让我们看看一个你不会去使用列表的例子，因为要用在这里列表的长度必须无限；

我们的“列表”是一个Fibonacci（一种数列）。一个迭代器在下面被使用：

```
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1
    def next(self):
        self.a, self.b    = self.b,    self.a+self.b
        return self.a
    def __iter__(self):
        return self
```

注意，迭代器实现了__iter__方法，这个方法实际上是返回迭代器本身。在很多情况下，你会把__iter__放到其他的你会在for循环中使用的对象中。那样做后就能返回你要的迭代器。有一点需要强调：在迭代器实现自己的__iter__方法后就能直接在for循环中使用迭代其本身了。

■**注意** 正式说来，一个实现了__iter__方法的对象是可迭代（iterable）的，一个实现了next方法的对象则是迭代器。

首先，产生一个Fibs对象：

```
>>> fibs = Fibs()
```

你能在for循环中使用——比如去查找在Fibonacci中比1000大的数中的最小的数：

```
>>> for f in fibs:
        if f > 1000:
            print f
            break
...
1597
```

在这里循环停止了，因为我设置了break，如果我没有设置，那么for循环就不会停止。

■**提示** 内部函数iter能被用来从一个可迭代的对象获得迭代器。

从迭代器得到序列
Making Sequences from Iterators

除了在迭代器上迭代（这是你经常做的），你还能把它们转换为序列。在大部分能使用序列的情况下（除了像indexing或者是slicing的操作），你能使用迭代器作为替换。关于这个的一个很有用的例子是使用列表的构造函数把一个迭代器转化为一个列表。

```
>>> class TestIterator:
        value = 0
        def next(self):
            self.value += 1
            if self.value > 10: raise StopIteration
            return self.value
        def __iter__(self):
            return self
...
>>> ti = TestIterator()
>>> list(ti)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

生成器（Generators）
Generators

生成器（因为历史原因也叫简单生成器）是新引入Pythong的。它和迭代器可能是近几年来引入的最强大的特征。生成器是一种用普通的函数语法定义的迭代器。它的工作方式通过例子能很到的展现。让我们先看看怎么创建和使用然后再了解一下它的机理。

创建一个生成器
Making a Generator

创建一个创建生成器很简单。就像创建一个函数。我相信你现在已经厌烦了Fibonacci序列的例子，因此我们另外做些事。我会创建一个平行嵌套列表的函数。参数是一个列表，和这个很像：

```
nested = [[1, 2], [3, 4], [5]]
```

换句话说就是一个列表的列表。我的函数应该按顺序告诉我列表中数字。解决的办法如下：

```
def flatten(nested):
    for sublist in nested:
        for element in  sublist:
            yield element
```

这个函数的大部分是简单的。首先是迭代nested中的所有子列表；然后按顺序迭代子列表中的元素。如果最后一行是打印元素的话，这个函数就容易理解，不是吗？

因此这里的yield语句是新的东西。任何包含yield语句的函数被叫做生成器。生成器不只是叫个新名字，它的行为和普通的函数有很大的不同。不同之处在于它不是像你用return那样返回值而是你每次产生一个值。每次一个值被产生，函数被冻结（freezes）：即函数停在那点等待被激活（reawakened）。函数被激活后就从停止的那点开始执行。

我能使用通过在生成器上迭代来使用所有的值。

```
>>> nested =  [[1, 2], [3, 4], [5]]
>>> for num in flatten(nested):
        print num
...
1
2
3
4
```

```
5

or

>>> list(flatten(nested))
[1, 2, 3, 4,  5]
```

递归生成器
A Recursive Generator

我在上一个部分创建的生成器只能处理两层嵌套，为了处理嵌套使用了两个for循环。如果你要处理任意层的嵌套该怎么办？可能你要使用树结构来表示。（我也能特定的tree类来做，但原理是一样的）你需要给每层一个for循环，但因为你不知道有几层嵌套，所以你必须把你的解决方案改得更灵活。现在是了解递归（recursion）的时候了。

```
def flatten(nested):
    try:
        for sublist in  nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

当flatten被调用，有两种可能性（处理递归时大部分都是有两种情况）：基本的或者是需要递归的情况。在基本的情况中，函数被告知flatten只有一个元素（比如一个数），这种情况会产生一个TypeError（因为你试图对一个数进行迭代），生成器会产生一个元素。

如果你被告知flatten是一个列表（或者其他的可迭代的），你就要做一些事来处理。你必须遍历所有的子列表（一些可能不是列表）并对它们调用flatten。然后你通过使用for循环来产生子列表元素。这可能看起来有点奇怪，但却能起作用。

```
>>> list(flatten([[[1],2],3,4,[5,[6,7]],8]))
[1, 2, 3, 4,  5, 6, 7, 8]
```

把它变得更安全
Making It Safer

这么做只有一个问题。如果nested是一个类似于字符串的对象（string，Unicode，UserString等等），因为是一个序列所以不会有TpyeError，但是你不想对这样的对象进行迭代。

■**注意** 关于不应该在`flatten`中对类似于字符串的对象进行迭代有两个主要的理由。首先，你想要的是把类似于字符串的对象当成自动值（`atomic  values`），而不是当成序列。第二，

对它们进行迭代实际上会导致无穷递归，因为一个字符串的第一个元素是另一的长度为一的字符串，而那个后者的第一个元素就是字符串本身（！）。

为了处理这种情况，你必须在生成器的开始处添加一个检查。试着把传入的对象和一个字符串连接看看是不是有TypeError，这是检查一个对象谁不是类似于字符串的最简单和最快速的方法。[2] 下面是加入了检查的生成器：

```
def flatten(nested):
    try:
        # Don't iterate over string-like objects:
        try:    nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in    nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

正如你所看到的,如果语句nested+产生了一个TypeError,对象就会被忽略.然而如果没有产生TypeError,try中的else子句就会产生一个TypeError.这就会让类似于字符串的对象被产生(在except字句的外面)，了解了么？

这里有一个例子展示了这个版本的类能用于字符串：

```
>>> list(flatten(['foo', ['bar',    ['baz']]]))
['foo', 'bar', 'baz']
```

注意在这里没有执行类型检查。我没有测试nested是否是一个字符串（我能使用is实例完成检查），而只是检查nested的行为是不是像一个字符串。（通过是不是能和字符串连接来检查）

通用生成器
Generators in General

如果你看懂了目前的所有例子，你已经或多或少的知道如何使用生成器了。让我来替你描述一下：生成器是一个包含yield的函数。当它被调用时，在函数体中的代码不会被执行，而会返回一个迭代器。每次一个值被请求，生成器中的代码就会被执行直到遇到一个yield或者return。Yield的意味着应该创建一个值。Return意味着生成器要停止执行（不在创建任何事，return在一个生成器中使用时只能进行无参数调用）。

换句话来说，生成器是由两部分元素组成：生成器的函数（generator-function）和生成器的迭代器(generator-iterator)。生成器的函数是被def语句定义的包含yield的部分，生成器的

迭代器是这个函数返回的部分。按一种不是很准确的说法，两个实体经常被当作一个，合起来被叫做生成器。

2.感谢Alex Martelli指出了这个习惯用法和在这里使用的重要性

```
>>> def simple_generator():
        yield 1
...
>>> simple_generator
<function simple_generator at 153b44>
>>> simple_generator()
<generator object at 1510b0>
```

被生成器的函数返回迭代器能像其他的迭代器那样被使用。

避免生成器
Avoiding Generators

如果你使用的是一个老版本的Python，生成器是没法使用的。下面介绍的就是如何使用普通的函数模拟生成器。

```
result = []
```

如果代码已经使用了result这个名字，你应该用其他名字代替（使用一个更有描述性的名字是一个好注意）然后，用result.append(some_expression)替换所有的yield语句。

最后，在函数的末尾，添加return result

尽管这个可能不能用于所有的生成器，但能用于绝大多数。（比如，它不能用于一个无限的生成器，这个当然不能把它的值放入列表中）

下面是flatten生成器用普通的函数重写的版本：

```
def flatten(nested):
    result =  []
    try:
        # Don't iterate over string-like objects:
        try:  nested + ''
        except TypeError: pass
        else: raise TypeError
```

```
        for sublist in  nested:
            for element in flatten(sublist):
                result.append(element)
    except TypeError:
        result.append(nested)
    return result
```

八皇后问题
The Eight Queens

现在你已经学习了这章所有的内容，是把它们用于实践的时候了。在这部分，你会看到怎么使用生成器解决经典的编程问题。

生成器是逐渐产生结果的复杂递归算法的理想实现工具。没有生成器，那些算法经常要求你传递一个让递归能建立在其上的额外的参数，这个参数是一个已经完成一半的的解决方案。如果使用生成器，所有的递归调用只要创建自己的部分。我在前一个版本的**flatten**中就是用的后一种做法，你能使用相同的策略传递图和树型结构。

图和树

如果你没有听过图和树，那么你应该尽快的学习它们。它们在计算机科学，编程，离散数学，数据结构，算法中都是很重要的概念。你能从下面的链接的网页中得到简单的定义：

- http://mathworld.wolfram.com/Graph.html
- http://mathworld.wolfram.com/Tree.html
- http://www.nist.gov/dads/HTML/tree.html
- http://www.nist.gov/dads/HTML/graph.html

在网上搜索会获得更多信息

回溯
Backtracking

在一些应用程序中，你不会立即得到答案，你必须做很多次选择。你不只是在一个层面上作选择，你必须在你递归的每个层面上做出选择。为了和真实的生活做类比，想象你有一个很重要的会要参加。然而你不知道在哪儿开会，但在你面前有两扇门，开会的地点就在其中的

一扇门后面，你挑了左边的进入，然后又发现两扇门。你选了左边结果却错了，于是你回溯，并选择右边的门，结果还是错的，于是你再次回溯，直到你回到了你的开始点，再在那里选择右边的门。

这样的回溯策略在解决需要你尝试每个相关单元的问题时很有用。这样的问题能被这样解决：

```
# Pseudocode
for each possibility at level 1:
    for each    possibility at level 2:
        ...
            for each possibility    at level n:
                is it viable?
```

为了实现for循环你必须知道你会遇到的具体层数，如果你没法知道就使用递归。

问题
The Problem

这是一个被广为喜爱的科学迷题：你有一个棋盘和8个要放到上面的皇后。唯一的要求是没有皇后能威胁到其他。也就是说，你必须把它们放置成每个皇后都不能吃掉对方。怎样才能做到？皇后们要被放置在哪里？

这是一个典型的回溯问题：你尝试放置第一个皇后（在第一行 ），然后放置第二个，如此进行。如果你发现你不能放置下一个皇后，就回溯到放置前一个皇后到其他的位置。最后你或者尝试万所有的可能或者找到解决方案。

在被描述的问题中你被提示，只有8个皇后，但我们假设有任意数目的皇后（这样就更合实际的回溯问题更相似），你怎么解决？如果你要自己解决，你现在应该停止阅读，因为我就要给你解决方案。

■**注意** 实际上对于这个问题有更高效的解决方案，如果你想了解更多的细节，去网上搜索能得到很多有价值的信息。一个关于各种解决方案的简单历史可以在 http://www.cit.gu.edu.au/~sosic/nqueens.html找到

状态表示
State Representation

为了表示一个可能的解决方案（或者方案的一部分），你能使用元组（或者列表）。每个元组中元素都指示相应行的皇后的位置（就是列）
如果state[0]==3，那么你就知道在第一行的皇后是在第四列（记得么，我们是从0开始计数

的）。当在某一个递归的层面时你只能直到上一行皇后的位置，因此你可能有一个状态元组的长度小于8（或者是皇后的数目）。

■**注意** 我也能使用列表来代替元组来表示状态。在这里只是一个习惯的问题，一般来说，如果序列很小而且是静态的，元组是一个好的选择。

寻找冲突
Finding Conflicts

让我们从做一些简单地抽象开始。为了找到一种没有冲突的设置（在里面没有皇后会被其他的皇后吃掉），你首先必须定义冲突是什么。为什么不在你定义的时候把它定义成一个函数？

已知的皇后的位置被传递给冲突函数（以元组的形式），然后函数判断下一个的皇后的位置会不会有新的冲突。

```
def conflict(state, nextX):
 nextY = len(state)
 for i in    range(nextY):
       if abs(state[i]-nextX) in (0, nextY-i):
            return True
 return False
```

参数nextX是代表下一个皇后的水平位置， nextY是代表垂直位置。这个函数对以前的每个皇后的位置做一个检查，如果下一个皇后和以前的一个皇后有同样的水平位置，或者是在一条对角线上，一个冲突就发生了， true就会被返回。如果没有这样的冲突发生， false就被返回，奇怪的是下面的语句：

```
abs(state[i]-nextX) in   (0, nextY-i)
```

如果下一个皇后和前一个的水平距离为0或者是等于垂直距离(这样就在一条对角线上)就返回true,否则就返回false.

基本情况
The Base Case

八皇后问题的实现需要有一点小技巧,但如果使用生成器就没什么难的了.如果你没有习惯于递归,我也不期望你自己完成解决方案.需要注意的是这个解决方案不是很有效率,因此如果是一个很大数目的皇后就会有点慢..

让我们从基本的情况开始: 最后一个皇后.你想要它怎么做?假设你可能是要找出所有可能的位置;那样,你可能期望它能根据其他皇后的位置生成它自己能占据的所有位置(可能没有).你能把这样的情况简单的画出。

```
def queens(num, state):
    if len(state) == num-1:
        for pos in range(num):
            if not conflict(state, pos):
                yield pos
```

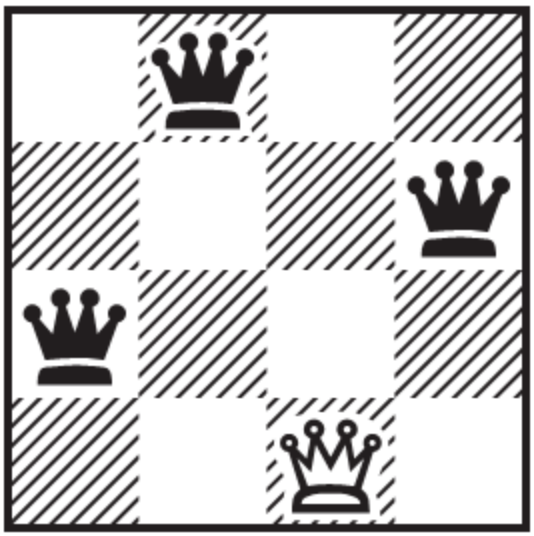


图9-1 *在一个4x4的棋盘上放4个皇后*

用人类的语言来描述它的意思是：如果只剩一个皇后没有放置，遍历最后一个的所有可能的位置并且返回没有冲突发生的位置．Num参数是皇后的总数目．State参数是存放前面皇后的位置信息的元组．比如，假设你有 4 个皇后，前三个分别被放置在 1 ， 3 ， 0 号位置，如图9-1所示. (不要在意第四行的白色皇后)

正如你在图中看到的,每个皇后占据了一行,并且位置的标号已经到了最大(在Python中都是从0开始的)：

```
>>> list(queens(4, (1,3,0)))
[2]
```

代码看起来就像一个魔咒。使用列表来让生成器创建列表中的所有值。在这种情况下，只有一个位置是可行的。白色皇后被放置在了图9-1（注意颜色不是代表什么特殊的意思，并且这不是程序的一部分）。

需要递归的情况
The Recursive Case

现在，让我们看看解决方案中的递归部分。当你完成你的基本情况时，递归的情况可能被

假设成了所有的来自低层（有更高编号的皇后）的结果都是正确的。因此你需要做的就是为在先前的代码判断state的if语句增加else子句。

你想从递归调用中得到什么结果？你想要的是所有前面皇后的位置，对吗？ 假设位置信息作为一个元组返回。在这种情况下，你可能必须修改你的基本情况来 返回一个元组-我稍后会那么做。

这样，你从前面的皇后得到了包含位置信息的元组，并且你要为后面的皇后提供当前皇后的每种可能的位置信息。为了让程序继续运行下去你需要做的就是把当前的位置信息添加到元组中并传给后面的皇后。

```
    ...
    else:
        for pos in range(num):
            if not conflict(state, pos):
                for result in queens(num, state + (pos,)):
                    yield (pos,)   + result
```

for pes和if not conflict部分是和前面的代码差不多的，因此你可以稍微修改一下就行了。让我们添加一些参数：

```
def queens(num=8, state=()):
    for pos in range(num):
        if not conflict(state, pos):
            if len(state) == num-1:
                yield (pos,)
            else:
                for result in queens(num, state + (pos,)):
                    yield (pos,)   + result
```

如果你认为代码很难理解，那就把代码做的事用你自己的语言来叙述，这样能有所帮助。（你还记得在（pos）中的逗号必须被设置为元组而不是简单地插入值吗？）

生成器queens能返回给你所有的解决方案（那就是所有的能合法放置皇后的地方）：

```
>>> list(queens(3))
[]
>>> list(queens(4))
[(1, 3, 0, 2), (2, 0, 3, 1)]
>>> for solution in queens(8):
...         print solution
...
(0, 4, 7, 5,   2, 6, 1, 3)
(0, 5, 7, 2,   6, 3, 1, 4)
```

```
...
(7, 2, 0, 5,  1, 4, 6, 3)
(7, 3, 0, 2,  5, 1, 6, 4)
>>>
```

如果你用8个皇后做参数来运行queens，你会看到很多解决方案闪过，我们看看有多少：

```
>>> len(list(queens(8)))
92
```

包装
Wrapping It Up

在离开queens前，让我们让输出更容易理解一点。清理输出总是一个好的习惯因为这让找出bug的具体位置更容易。

```
def prettyprint(solution):
    def line(pos, length=len(solution)):
        return '. ' * (pos) + 'X  ' + '. ' * (length-pos-1)
    for pos in solution:
        print line(pos)
```

注意我在prettyprint里面创建了一个能帮点小忙的函数。我把它放置在那里是因为我假设我在外面的任何地方都不会用到它。下面我打印出一个随机的解决方案来看看函数是不是正确的工作。

```
>>> import random
>>> prettyprint(random.choice(list(queens(8))))
. . . . . X . .
. X . . . . . .
. . . . . X . X . .
. . . . .
. . . X . . . .
. . . . . . . X
. . . . X . . .
. . X . . . . .
```

对应的图示为图9-2。用Python是不是很爽?

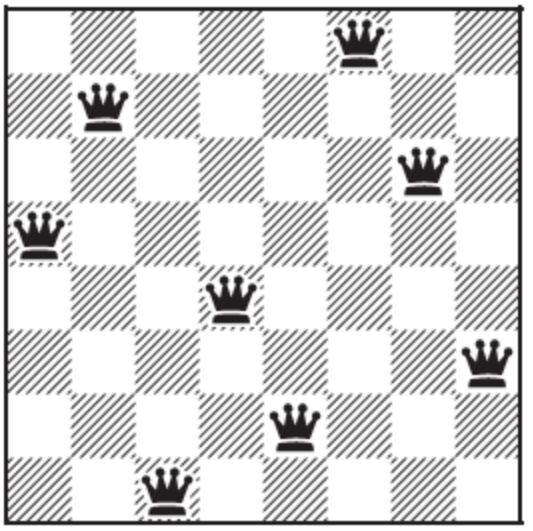


图 9-2. *八后问题多种解决方案中的一个*

快速总结
A Quick Summary

这章里有很多东西，我们来总结一下：

新式类：Python中类的工作方式发生了改变。在Python2.2中，新式类被引入，新式类提供了以些新的特征（比如super和属性）。为了创建一格新式类你必须子类化对象，不管是直接的还是间接的都行，或者设置__metaclass__属性也可以。

特殊方法：在Python中有一些特定的方法（名字是以“__”开始和结束的）。这些方法和函数只有很小的不同，但其中的大部分是在某些情况下被Python自动调用的。（比如__init__在对象被创建后调用）

构造函数：这是面向对象的语言共有的，你可能要为你自己写的每个类实现构造函数。构造函数被命名为__init__并且在对象被创建后自动调用。

重载：一个类能通过重新实现来覆盖它的超类中的方法和属性。如果新方法要调用老版本的方法，可以从超类（旧式类）直接的使用未绑定的版本或者是使用super函数（新式类）

序列和映射：创建自己的序列或者映射需要实现所有的序列的或是映射的原则，包括__getitem__和__setitem__这样特殊函数。通过子类化列表（或者用户自定义列表）和字典（或者是用户自定义字典）你能省不少事。

迭代器：迭代器是一个有next方法的简单的对象。迭代器能在一系列的值上

进行迭代。当没有值可供迭代时next方法就会发生StopIteration异常。可迭代对象有一个返回迭代器的__iter__方法，它能像序列那样在for循环中使用。经常一个迭代器自己是可迭代的，即迭代器有返回它自己的next方法

生成器：生成器函数（或者方法）是包含了关键字yield的函数。但被调用时，生成器函数返回一个生成器（一种特殊的迭代器）。

八皇后问题：八皇后问题在计算机科学中很有名，我们能很容易的使用生成器来实现。问题描述的是如何在棋盘上放置8个皇后而不让它们中的任何一个能被其他的吃掉。

本章内的新函数
New Functions in This Chapter

函数	说明描述
iter(obj)	从一个可迭代的对象得到迭代器
property(fget, fset, fdel, doc)	返回一个属性，所有的参数都是可选的
super(class, obj)	返回一个超类的邦定实例。

请注意，iter和super可能被用其它未在这里说明的参数调用。要了解更多的信息，请参看《Standard Python Documentation》（http://python.org/doc）。

现在学什么？
What Now?

你现在已经了解很Python这门语言的很多东西了。那么为什么还剩有很多章？因为还有很多需要学习，很多是关于Python怎么通过多种方法和外部世界联系。然后你有一些项目.....因此我们现在还没完。

第十章

充电时刻

Batteries Included

现在你已经懂得了大部分Python语言的基础。Python的核心语言非常强大，可以提供给你更多值得一玩的工具。标准的安装方式包括一些模块的集合，它们被称为**标准库（Standard Library）**。你已经见识过它们中的一些（math和cmath，包括用于计算实数和复数的数学函数），但是还有更多你没有见过。本章将会让你小窥这些模块的工作方式，并且探索它们，学习它们可以提供的功能。之后你会对标准库进行纵览，着重于一部分有用的模块。

模块
Modules

你已经学会如何创建自己的程序（或者说脚本）并且执行它们了。你还见识过怎么用import将函数从外部模块抓到你的程序中来：

```
>>> import math
>>> math.sin(0)
0.0
```

让我们看看怎么才能写自己的模块。

模块是程序
Modules Are Programs

任何Python程序都可以作为模块导入。假设你写了一个列表10-1的程序，并且将它存为hello.py（名字很重要）。

列表 10-1. *一个简单的模块*

```
# hello.py
print "Hello, world!"
```

你在哪里保存它也很重要。下一节中你会了解更多这方面的知识，但是现在假设你存在C:\python（Windows）或者~/python（UNIX）目录中。那么你就可以执行下面的代码告诉解释器在哪里寻找模块（以Windows目录为例）：

```
>>> import sys
>>> sys.path.append('c:/python')
```

注意 在上面例子中的目录路径中，我使用了正斜线。Windows中反斜线才是标准。两种斜线都是合法的，但是因为反斜线用来书写特殊字符（比如新行\n），正斜线使用起来安全一些。如果你使用反斜线，可以使用自然字符串方式（r'c:\python')或者将反斜线转义（'c:\\python'）。

我这里所做的只是告诉解释器除了平常经常寻找的目录，它也应该在目录c:\python中寻找模块。完成这个步骤之后，你就能导入自己的模块了（存储在c:\python\hello.py的文件，记得吗？）。

```
>>> import hello
Hello, world!
```

注意 当你导入模块时候，可能会注意到出现了新文件——本例中为c:\python\hello.pyc。这个以pyc为扩展名的文件是经过处理（编译）的，已经翻译成Python能够更加有效处理的文件。如果你之后导入了同一个模块，Python会导入pyc文件而不是py文件，除非py文件被改变过。那种情况下，新的pyc文件会生成。删除pyc文件不会损害程序（因为有同样效果的py文件存在）——当需要的时候，新的pyc文件又会被创建。

如你所见，当你导入模块的时候，其中的代码被执行了。不过如果你试图再次导入，就什么都不会发生：

```
>>> import hello
>>>
```

为什么这次没用了？因为模块并不意味着在被导入的时候去**做**事情（比如打印文本）。它们主要用于**定义**事情，比如变量、函数、类等等。而且因为只需要定义这些东西一次，多次导入模块和导入一次的效果是一样的。

为什么只是一次？

这种“只导入一次”（import-only-once）的行为在大多数情况下是一种优化，尤其在一种情况下非常重要：如果两个模块互相导入。

很多情况中，你可能会写两个互相访问函数和类的模块。举例来说，你可能创建了两个模块——clientdb和billing——包括用于客户端数据库和结帐系统的代码。你的客户端数据可能包含需要调用结帐系统的功能（比如每月自动将帐单发送给客户），而结帐系统可能也需要访问客户端数据，以保证帐单正确。

如果每个模块都可以导入数次，那么就出问题了。模块clientdb会导入billing，而billing又导入clientdb，然后clientdb又……你应该能想到这种情况。这个时候导入就成了无穷循环（无穷递归，记得吗？）。但是，因为在第二次到如模块时候什么都不会发生，循环就被打破了。

如果你坚持重载入模块，那么你可以使用内建的reload函数。它带有一个参数（你要导入的模块），并且返回重载入的模块。如果在你更改模块并且想要在程序运行时将这些更改反映出来时候这个功能会比较有用。重载入hello模块（只包含print语

句），你可以这么做：

```
>>> hello = reload(hello) Hello, world!
```

在这里我假设hello已经被导入过（一次）。将reload函数的返回值赋给hello，我就可以用冲载入的版本替换原先版本的模块。如你所见，问候语已经打印出来了，我在这里又导入了一次模块。

注意，如果你已经用给定的模块创建了对象，然后进行重载入，你的对象是不会重创建的。如果你想要对象基于重载入的模块。，你需要重新创建一个新的对象。

模块用来定义事情
Modules Are Used to Define Things

所以，模块在第一次导入到程序时候被执行。看起来有点用——但是并不是很有用。真正的用处在于它们（像类一样）可以保持自己的作用域。这就意味着你所定义的任何类和函数，以及赋值后的变量都成为模块的特性。听起来挺复杂的，用起来很简单。假设你写了个类似列表10-2的模块，并且将它存储为hello.py。你将它放置到Python解释能够找到的地方——可以使用前一节中的sys.path方法，也可以用后面“让你的模块可用”一节中的方法：

列表 10-2. *包括函数的简单模块*

```
# hello2.py
def hello():
    print "Hello, world!"
```

你可以这样导入：

```
>>> import hello2
```

模块会被执行，意味着hello函数在模块的作用域内被定义，也意味着你可以这样使用函数：

```
>>> hello2.hello()
Hello, world!
```

任何在模块的全局作用域中定义的名称都可以用同样方法使用。

干嘛自找麻烦？

你可能会奇怪，为什么要这么做呢。为什么不在主函数中定义一切呢？主要原因是代码**重用（code reuse）**，如果你把代码放在模块中，你就可以一个以上的程序中使用。以为这如果你写了一个非常棒的数据库客户端程序，并且将它放在叫做clientdb的模块中，你就可以在结帐系统发送垃圾邮件（尽管我不希望你这么做）时候使用，也可以用在任何需要访问你的客户数据的程序中。如果没有将这段代码放在单独的模块中，你就需要在每个程序中重写代码。所以请记住：为了让你的代码可重用，将它存为模块（是的，这也当然关乎于抽象）！

`if __name__ == '__main__'`

模块用来定义函数、类和其他一些东西，但是有些时候（事实上是经常），在模块中添加测试是否工作正常的代码是很有用的。举例来说，假如你想要确保hello函数工作正常，你可能会将hello2重写为hello3，如列表10-3：

列表 10-3. *带有问题的测试代码模块：*

```
# hello3.py
def hello():
    print "Hello, world!"

# A test:
hello()
```

看起来是合理的——如果你将它作为普通程序运行，你会看到它工作正常。但是如果你将它作为模块导入，然后在其他程序中使用hello函数，测试代码就会被执行，就像本章开头的第一个hello模块一样：

```
>>> import hello3
Hello, world!
>>> hello3.hello()
Hello, world!
```

这可不是你想要的。避免这种情况的关键在于“告诉”模块它是作为程序运行还是导入到其它程序。你需要使用__name__变量：

```
>>> __name__
'__main__'
>>> hello3.__name__
'hello3'
```

如你所见，“主程序”（包括解释器的交互式提示符在内），变量__name__有值'__main__',当作为模块导入时，这个值就被设定为模块的名字。你可以让模块的测试代码更加好用，将其放置在if语句中，就像列表10-4中的代码：

列表 10-4. *使用条件测试代码的模块*

```
# hello4.py

def hello():
    print "Hello, world!"

def test():
    hello()

if __name__ == '__main__': test()
```

如果将它作为程序运行，hello函数会被执行。而作为模块导入时，它就会像个普通模块一样：

```
>>> import hello4
>>> hello4.hello()
Hello, world!
```

如你所见，我将测试代码放在了test函数中。我也可以直接将它们放置在if语句中，但是放置在独立的test函数中可以让你在将模块导入其他程序的时候也可以进行测试：

```
>>> hello4.test()
Hello, world!
```

■**注意** 如果你写了更完整的测试代码，将其放置在单独程序中更好。关于书写测试代码的更多信息，参见第十六章。

让你的模块可用
Making Your Modules Available

前面的例子中，我改变了sys.path，其中包含了一列表的解释器应该寻找模块的目录（作为字符串）。然而一般来说可能你不详真么做。理想情况下sys.path会包含正确的目录（包括你的模块的目录）。有两种方式可以做到这点：

方案1：将模块放置在正确位置
Solution 1: Putting Your Module in the Right Place

将你的模块放置在正确位置（或者说**某个**正确位置，因为会有多种可能性）。这就是Python解释器在哪里你的模块的问题了，找到后你就可以把你的文件放在那里。

■**注意** 如果你机器上面的Python解释器是由管理员安装的，而你有没有管理员许可，你可能无法将你的模块存储在任何Python使用的目录中。这种情况下请直接参看方案2。

你可能记得，那些（称为搜索路径的）目录的列表可以在sys模块中的path变量中找到：

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
['C:\\Python24\\Lib\\idlelib',
 'C:\\WINDOWS\\system32\\python24.zip',
 'C:\\Python24',
 'C:\\Python24\\DLLs',
 'C:\\Python24\\lib',
 'C:\\Python24\\lib\\plat-win',
 'C:\\Python24\\lib\\lib-tk',
 'C:\\Python24\\lib\\site-packages']
```

■**提示** 如果你的数据结构过大，不能在一行打印完，可以使用pprint模块中的pprint函数替代普通的print语句。pprint是个相当好的打印函数，更加智能。

这是Windows上Python2.4的标准路径。你的结果可能会不同。关键在于每个字符串都提供了一个位置，也就是你想让编译器寻找你的模块时可以放置的位置。尽管这些位置都可以使用，site-packages目录是最佳选择，因为它就是用来做这些事情的。查看你的sys.path，找到site-packages目录，将列表10-1的模块存储其中，记得改名，比如改成another_hello.py，然后测试：

```
>>> import another_hello
>>> another_hello.hello()
Hello, world!
```

因为你的模块放置在了类似site-packages的目录中，所有你的程序就都能导入它了。

方案2：告诉编译器去哪里找
Solution 2: Telling the Interpreter Where to Look

方案1可能由于下面几种情况不适合你：

- 你不希望因为自己的模块而扰乱Python解释器的目录
- 你没有在Python解释器目录中存储文件的许可
- 你想将模块放在其他地方

最后一个原因是“你想将模块放在其他地方”，那么你就要告诉解释器去哪里找。你之前已经看到了一种方法，就是编辑sys.path，但是这不是**一般**的方法。标准方法是在PYTHONPATH变量中包含你的模块所在的目录。

环境变量

环境变量并不是Python解释器的一部分——它们是操作系统的一部分。它们基本上相当于Python变量，但是是在Python解释器外设定的。关于设置的方法你应该参考你的系统文档，但是我在这里提几句：

UNIX系统内，你可以在一些你每次登录都要执行的shell文件内设置环境变量。如果你用类似bash一类的shell，那么要修改的就是 .bashrc，可以在你的主目录中找到。将下面的命令添加到那个文件中，以增加~/python为你的PYTHONPATH：

```
export PYTHONPATH=$PYTHONPATH:~/python
```

注意，多个路径以冒号分隔。其他的shell可能会有不同的语法，所以你应该参考相应的文档。

Windows系统内，你可以使用控制面板（对于新版的Windows，比如XP、2000和NT应如此，旧版本的，比如Windows 98就不行了，你需要修改autoexec.bat文件，下一段中会讲到）。依次点击开始菜单->设置->控制面板。进入后，双击“系统”图标。在打开的对话框中选择“高级”选项卡，点击“系统变量”按钮。这时会弹出一个有两个标签的对话框：其中一个是你的用户变量，另外一个是系统变量。你要修改的是用户变量。如果你看到其中已经有PYTHONPATH项，那么选中它，单击“编辑”进行编辑。如果没有，单击“新建”，然后使用PYTHONPATH作为“变量名”，输入你的目录地址作为“变量值”。注意，多个目录以分号分隔。

如果上面的方法不行，你还能修改autoexec.bat文件，你可以在C盘的根目录下找到（假设你是标准安装Windows）。用记事本（或者IDLE编辑器）打开它，增加一行设

置PYTHONPATH的内容。如果你想要增加C:\python，可以这样做：

set PYTHONPATH=%PYTHONPATH%;C:\python

有关更多在Mac OS上设置系统变量的信息，参看MacPython的网页：
http://cwi.nl/~jack/macpython。

不同的操作系统中PYTHONPATH的值也不同（参看上面的阅读资料），但是基本上和sys.path里面的目录列表一样。

■**提示** 你不需要使用PYTHONPATH来更改sys.path。路径配置文件可以让Python替你完成工作的捷径。路径配置文件是以.pth为扩展名的文件，包括应该增加到sys.path的目录信息。空行和以#开头的行都会被忽略。以import开头的文件会被执行。执行路径配置文件的话需要将其放置在可以找到的地方。对于Windows来说，使用sys.prefix中的名字（可能类似于C:\Python22），UNIX中使用site-packages目录（更多信息可以参看Python库参考中的site模块，这个模块在Python解释器初始化时自动导入。

命名模块
Naming Your Module

你可能注意到了，包含模块代码的文件的名字要和模块名一样——再加上.py扩展名。Windows内，你也可以使用.pyw扩展名。有关文件扩展名的更多信息请参看第十二章。

包
Packages

为了组织模块，你可以将它们分组为**包（package）**。包基本上就是另外一类模块。包有趣的地方是他们能包含其他模块。当模块存储在文件中时（扩展名.py），包就是模块所在目录。为了让Python将其作为包对待，它必须包含一个命名为__init__.py的文件（模块）。如果将它作为模块导入的话，文件的内容就是包的内容。比如你有个命名为constants的包，文件constants/__init__.py包括语句PI=3.14，那么你就可以这么做：

```
import constants
print constants.PI
```

为了将模块放置在包内，直接把模块放在包目录内即可。

比如你需要个叫做drawing的包，包括叫做shapes和colors的模块，你就需要列在表10-1中的文件（UNIX路径名）：

表 10-1. *简单的包设计*

File/Directory文件/目录	Description描述
~/python/	PYTHONPATH中的目录
~/python/drawing/	包目录（drawing包）
~/python/drawing/__init__.py	包代码（drawing模块）
~/python/drawing/colors.py	colors模块
~/python/drawing/shapes.py	shapes模块

表10-1中的内容是在你将目录~/python放置在PYTHONPATH的前提下的。Windows系统中，只要用c:\python替换~/python即可，并且将正斜线换为反斜线。

依照这个设置，下面的语句都是合法的：

```
import drawing                  # (1)
Imports the drawing package import drawing.colors           # (2)
Imports the colors module from drawing   import shapes   # (3)
Imports the shapes module
```

在语句（1）中，drawing中__init__模块的内容是可用的，drawing和colors模块则不可用。在语句（2）后，colors模块可用，但是只能用drawing.colors来使用。在语句（3）后，shapes模块可用，可以使用短名（shapes）使用。注意，这些语句知识例子。你不用像我这里一样，在导入包的模块前导入包本身。第二个语句可以独立使用，第三个也一样。你可以在包之间进行嵌套。

探索模块
Exploring Modules

在应付标准库模块前，我会告诉你如何自己探索模块。这个技能极有价值，因为作为Python程序员，你会在自己的职业生涯中遇到很多有用的模块，我又不能在这里将其一一讲到。目前的标准库已经大到可以出本书了（事实上已经有这类书了）——而且他还在增长。每次新的模块发布都会增加进标准库，一些模块也在变化和改进。而且你还能在网上找到些有用的模块，“神入”它们可以很快很易地让你的编程变得更加享受。

模块中有什么？
What's in a Module?

探究模块最直接的方式就是在Python解释器中调查它们。当然，你要做的第一件事就是导入它。假设你听说有个叫做copy的模块：

```
>>> import copy
```

没有异常——所以它是存在的。但是它能做什么？它又有什么？

使用dir
Using dir

查看模块包含的内容可以使用dir函数，它会将对象（函数、类、变量以及模块）的所有特性列表。如果你想要打印出dir(copy)的内容，你会得到一长串名字（试试看）。一些名字以下划线开始，——提示（约定俗成）它们并不能在模块外部使用。所以让我们用列表推导式（如果你不知道这是什么，请参看第五章中关于列表推导式的章节）过滤掉它们：

```
>>> [name for name in dir(copy) if name[0]   != '_']
['Error', 'PyStringMap', 'copy',   'deepcopy', 'error']
```

这个列表推导式是个包含所有dir(copy)中不以下划线开头的名字的列表。这个列

表比完整列表要清楚些。

1.术语“神入”（grok）是“黑客语言（hackerspeak）”，意思是“完全理解”，从Robert A.Heinlein的小说《异乡异客（Stranger in a Strange Land）》中而来（Ace Books，1995年重新发行）。

译注：这本小说相当好看，可以说是R.A.海茵莱茵的代表作，也是科幻界的不朽巨著，获得过雨果奖和星云奖。推荐对科幻有兴趣的朋友们阅读 ：）。目前市面上有《科幻世界》出版社出版的中文译本，质量也不错。可惜没有电子版的。

■**提示** 如果你喜欢用tab完成，你应该查看库参考中的readline和rlcompleter模块。它们在探索模块时候很有用。

---

all变量
all variable

我在上一节中列表推导式中所做的事情是推测我**可能**会在copy模块中看到什么。但是你可以从列表本身获得正确答案。在完整的dir(copy)列表中，你可能会注意到__all__这个名字。这是个包含有类似于我刚才的列表推导式的内容的列表——除了这个列表位于模块本身。让我们看看它都有什么：

```
>>> copy.__all__
['Error', 'error', 'copy', 'deepcopy']
```

我的猜测还不太差。我只得到了一个额外的不是我有意要用的名字（PyStringMap）。但是__all__列表从哪来，为什么会存在？第一个问题容易回答。他在copy模块内部，像这样（从copy.py直接复制）：

```
__all__ = ["Error", "error", "copy", "deepcopy"]
```

那么为什么存在呢？它定义了模块的**公有接口（public interface）**。更准确地说，它告诉解释器它要从模块导入所有名字。所以如果你使用：

```
from copy import *
```

你只会得到__all__变量中的四个函数。导入PyStringMap的话你就要显式地使用imprort copy然后copy.PyStringMap，或者用from copy import PyStringMap。

类似的__all__是书写模块时候的有用技巧。因为你的模块中可能会有一大堆其他程序不需要或不想要的变量、函数和类，__all__“客气地”将它们过滤了出去。如果你没有设定__all__，用import *语句会输出的所有模块模块中不以下划线开头的全局名称。

用 help 获取帮助
Getting Help with help

目前为止，你已经利用自己的灵巧和Python的函数以及特殊特性探索了copy模块。交互式编译器是个非常强大的用于这类探索的工具，因为你对于语言的掌握只取决于你探索模块

有多深。但是，还有个可以给你所需所有信息的标准函数。这个函数叫做help，让我们先试试copy函数：

```
>>> help(copy.copy)
Help on function copy in module copy:

copy(x)
  Shallow copy operation on arbitrary Python objects.

  See the module's __doc__ string for more info.

>>>
```

很有趣：他告诉你copy带有一个参数x，并且是“浅拷贝操作”。但是它还提到了模块的__doc__字符串。这是什么？你可能记得我在第六章中提到的文档字符串。它是你写在函数开头以“归档”函数的字符串。这个字符串存储在函数的__doc__特性中。就像你从上面的帮助文本中所理解到的一样，模块也可以有文档字符串（写在模块开头），类也一样（写在类开头）。

事实上，前面的帮助文本是从函数的文档字符串中取出的。

```
>>> print copy.copy.__doc__
Shallow copy operation on arbitrary Python objects.

See the module's __doc__ string for more info.
```

像这样使用help直接检查文档字符串的好处是你会获得更多信息，比如函数签名（也就是带有什么参数）。试着调用help(copy)（在模块本身）看看得到什么。它会打印出很多信息，包括对于copy和deepcopy的彻底讨论（deepcopy(x)对存储在x中的值作为特性进行拷贝，而copy(x)只是拷贝x，将x中的值绑定到拷贝版本的特性上）。

文档
Documentation

有关模块的信息的自然来源当然是文档。我延后了文档的讨论，是因为你自己检查模块总是快一些。举例来说，你可能会问“range的参数是什么？”。除了在Python书籍或者标准Python文档中寻找range的描述外，你还可以直接查看：

```
>>> print range.__doc__
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers. range(i, j)
returns [i, i+1, i+2,..., j-1]; start (!) defaults to 0. When step is  given, it
specifies  the increment (or decrement).
For example,  range(4) returns [0, 1, 2, 3].  The end  point is  omitted!
These are exactly the valid indices for a list of 4 elements.
```

这样你就获得了关于range的精确描述，因为你可能已经运行了Python解释器（在你编程时经常会有这样的问题），这样获得信息花不了几秒钟。

但是，并不是每个模块和函数都有不错的文档字符串的（尽管应该有），有些时候你可能需要更多彻底的描述。大多数你从网上下载的模块都有相关文档。在我看来，对于学习Python编程来说最有用的莫过于Python库参考（Python Library Reference），它对于所有标准库中的模块都有描述。如果我想要查看Python的知识，十有八九我会去找它。库参考可以在线（http://python.org/doc/lib）或者下载浏览，其他一些标准文档（比如Python Tutorial或者Python Language Reference）也如此。所有这些文档都可以在Python网站（http://python.org/doc）找到。

使用源代码
Use the Source

目前为止我们讨论的探索技术在大多数情况下已经够用。但是希望真正理解Python语言的人想了解的关于模块的东西，是不能脱离阅读源代码而得到的。阅读源代码，事实上是学习Python最好的方式——除了自己编码外。

真正的阅读不是问题，但是问题在于源代码在哪？假设你想阅读标准模块copy的代码。你从哪去找呢？一个方案是检查sys.path，自己找，就像解释器做的一样。另外一个快捷方式是检查模块的__file__属性：

```
>>> print copy.__file__
C:\Python24\lib\copy.py
```

■**注意** 如果文件名以.pyc结尾，只要查看相应的以.py结尾的文件即可

就在那！你可以在你的代码编辑器中打开copy.py（比如IDLE），然后查看它是如何工作的。

■**警告** 当这样在文本编辑器中打开标准库文件时，你也承担着意外修改它的风险。这样做可能会破坏它，所以在你关闭文件的时候，确保你没有保存任何可能做出的修改。

注意，一些模块不包含任何你能阅读的Python源代码。他们可能被构造进解释器内（比如sys模块），或者可能是使用C程序语言写成（C代码也是可用的，但是已经超出了本书的范围，请查看第十七章获得更多关于用C扩展Python的信息）。

标准库：一些最爱
The Standard Library: A Few Favorites

有可能你会对本章的题目不知所云。这个短语最开始由Frank Stajano创造，用于描述Python丰富的标准库。安装Python后，你就“免费”获得了很多有用的模块，因为获得这些模块信息的方式很多（在本章的第一部分已经解释），我不会在这里列出所有的参考（因为要花上很大篇幅），但是我会对一些我最喜欢的进行说明，以打开你关于探索模块的胃口。你会在“工程章节”（第二十章及以后）碰到更多的标准模块。模块的描述并不完全，但是会覆盖每个模块比较有趣的特性。

sys

这个模块提供你访问和Python解释器联系紧密的变量和函数，其中一些列在表10-2中。

表 10-2. *sys模块中一些重要的函数和变量*

函数/变量	描述
argv	命令行参数，包括脚本名称
exit([arg])	退出当前的程序，可选参数为给定的值或者错误信息
modules	映射模块名字和载入模块的字典
path	模块所在目录的列表
platform	包括类似sunos5或者win32的平台标识符
stdin	标准输入流——文件类似（file-like）对象
stdout	标准输出流——文件类似（file-like）对象
stderr	标准错误流——文件类似（file-like）对象

变量sys.argv包括传递到Python解释器的参数，包括脚本名称。

函数sys.exit可以退出当前程序（如果和try/finally块一起调用，finally子句的内容会被执行）。你可以提供一个整数作为参数，标识程序是否成功运行——UNIX的管理。大多数情况下使用默认就可以（也就是0，表示成功）。或者你可以提供字符串参数，用于错误信息，对于用于查看程序为什么崩溃会很有用。程序会提供错误信息和错误标识码然后退出。

sys.modules将模块名字映射到实际存在的模块上。它只提供目前导入的模块。

sys.path模块变量在本章前面讨论过，它是一个字符串列表，每个字符串都是在import语句执行时，编译器寻找模块的目录名。

sys.platform模块变量（字符串）是解释器正在运行的“平台”。它可能是标识操作系统的名字（比如sunos5或者win32），也可能是其他的一些平台，如果你正在运行Jython的话，就是Java虚拟机（比如java1.4.0）。

sys.stdin、sys.stdout和sys.stderr模块变量是文件类似的**流（stream）**对象。它们表示标准UNIX概念的标准输入、输出和错误。简单来说，Python利用sys.stdin获得输入（比如函数input和raw_input），利用sys.stdout输出。你会在第十一章中学到更多关于文件（以及这三个流）的知识。

范例

反序打印参数。当你从命令行调用Python脚本时，可能会在后面加上一些参数——这就是**命令行参数（command-line argument）**。这些参数会放置在sys.argv列表中，脚本的名字存储在sys.argv[0]中。反序打印这些参数很简单，你可以在列表10-5中看到：

列表 10-5. *反序打印命令行参数*

```
# reverseargs.py
import sys
args = sys.argv[1:]
args.reverse()
print ' '.join(args)
```

如你所见，我对sys.argv进行了拷贝。你可以修改原始的列表，但是一般来说安全起见还是不要那么做，因为程序的其它部分可能也需要sys.argv包括的原始参数。注意，我跳过了sys.argv的第一个元素——脚本的名字。我使用args.reversee()方法进行反向排序，但是这样是不能打印出结果的——这是个返回None的即时修改操作。最后，为了保证输出得更好，我使用了字符串的join方法。让我们是试试看结果如何（假设使用UNIX Shell，但是在MS-DOS提示符下也是一样）：

```
$ python reverseargs.py this is a test
test a is this
```

OS

os模块提供给你访问操作系统服务的功能。os模块很大，列表10-1中只是其中很少一部分最有用的函数和变量。另外，os和它的子模块os.path包括一些检查、构造、删除目录和文件的函数。关于它的更多信息，请参看标准库文档。

表 10-3. *os模块中一些重要函数和变量*

函数/变量	描述
environ	对环境变量进行映射
system(command)	在子shell中执行操作系统命令
sep	路径中的分隔符
pathsep	分隔路径的分隔符
linesep	行分隔符('\n', '\r', or '\r\n')
urandom(n)	返回n Bytes的加密强随机数据

■**提示** os.walk是很有用的浏览目录的函数。请在Pythno库参考中查看。

os.environ映射包含本章前秒描述过的系统变量。比如要访问系统变量PYTHONPATH，你可以使用os.environ['PYTHONPATH']。这个映射也可以用来更改系统变量，不过不是所有系统都支持。

os.system函数用于运行外部程序。也有一些函数可以执行外部程序，包括execv，它可以关闭Python解释器，提供对于被执行程序的控制；以及popen，可以对程序生成文件类似的连接。关于这些函数的更多信息，请参看标准库文档。

■**提示** subprocess模块是目前才加入到Python中的，它包括了os.system、execv和popen的功能。

os.sep模块变量是用于路径名中的分隔符。UNIX中的标准分隔符是“/”，Windows中的是“\\”（Python的语法），而Mac OS中是“:”（有些平台上，os.altsep包括可选的路径分隔符，比如Windows中的“/”）。

你可以在组织路径的时候使用os.pathsep，就像在PYTHONPATH中的一样。pathsep用于分割路径名：UNIX使用“:”，Windows使用“;”，Mac OS使用“::”。

模块变量os.linesep是用于文本文件的字符串分隔符。UNIX中为新行字符（\n），Mac OS中是（\r），而Windows则是两者的合并（\r\n）。

urandom函数提供依赖于系统的“真”（至少是足够强度加密的）随机数。如果你的平台不支持它，会得到NotImplementedError异常。

范例

启动网络浏览器。系统命令可以用来执行程序，在类似于UNIX类的可以从命令行执行程序的环境中很有用，你可以用它发送Email，等等。但是对于图形界面中启动程序也很有用，比如网络浏览器。UNIX中，你可以使用下面的代码（假设你的/user/bin/firefox路径下有浏览器）：

```
os.system('/usr/bin/firefox')
```

Windows版本的可以是（也同样假设你这个路径下有浏览器）：

```
os.system(r'c:\"Program Files"\"Mozilla Firefox"\firefox.exe')
```

注意，我小心地用引号括起了Program FIles和Mozilla Firefox。否则DOS就会在空格处停下来（对于PYTHONPATH这点也同样重要）。而且你需要使用反斜线，因为DOS对于正斜线会犯糊涂。如果你运行程序，你会注意到浏览器会试图打开叫做File”、Mozilla……的网站——也就是在空牌后面的部分。最后，如果你试图在IDLE中运行程序，你会看到DOS窗口出现，直到你关闭这个DOS窗口前浏览器是不会出现的。这还不是完美的解决方法。

另外一个可以更好解决问题的函数是Windows特有函数os.startfile：

```
os.startfile(r' c:\Program Files\Mozilla Firefox\firefox.exe')
```

可以看到，os.startfile接受一般路径，就算包含空格也没问题（也就是不用像os.system例子中一样用引号括起来）。注意，在Windows内，你的Python程序在外部程序由os.system（或者os.startfile）启动后会继续运行，而UNIX内，你的程序会等待到直到os.system完成。

更好的方法：WEBBROWSER

os.system函数很有用，但是对于启动浏览器这样详细的任务来说还有更好的解决方法：webbrowser模块。它包括叫做open的函数，你可以用它来自动打开浏览器访问给定URL。举例来说，如果你想要你的程序在浏览器中打开Python的网站（或者打开新浏览器或者使用已经在运行的），你可以用：

```
import webbrowser
webbrowser.open('http://www.python.org')
```

网页就会弹出来了，很简洁吧？

fileinput

你会在第十一章中学到很多读写文件的知识，但是现在咱们先小看一下。fileinput模块可以让你轻松的地迭代文本文件的所有行。如果你这样调用你的脚本（假设在UNIX命令行下）：

```
$ python some_script.py file1.txt file2.txt file3.txt
```

你就可以顺序迭代file1.txt到file3.txt的所有行。你还能迭代标准输入（sys.stdin，记得吗？）的给定行。比如在UNIX中（使用标准UNIX命令行）：

```
$ cat file.txt | python some_script.py
```

If you use fileinput, this way of calling your script (with cat in a UNIX pipe) works just as well as the previous one (supplying the file names as command-line arguments to your script). The most important functions of the fileinput module are described in Table 10-4.

如果你使用fileinput，你调用脚本的方式（使用UNIX pipe中的cat）就像上一个例子中一样（将文件名作为参数提供给脚本）。fileinput模块最重要的函数列在表10-4中。

表 10-4. *fileinput模块中重要的函数*

函数	描述
input([files[, inplace[, backup]])	迭代多个输入流的行。
filename()	返回当前文件的名称
lineno()	返回当前（累计）的行数
filelineno()	返回当前文件的行数
isfirstline()	检查当前行是否是文件中第一行
isstdin()	检查最后一行是否是sys.stdin中的
nextfile()	关闭当前文件，移动到下一个
close()	关闭序列

函数fileinput.input是这些函数中最重要的。它会返回你可以在for循环中迭代的对象。如果你不想使用默认行为（fileinput查找要迭代的文件），你可以给函数提供

（序列形式的）一个或多个文件名。你还能将inplace参数设为真（inplace=True）以打开即时处理。对于访问的每一行，你需要打印出替代的，以返回到当前的输入文件中。可选的backup参数给定从原始文件进行备份的文件的扩展名（也就是包括你正在处理的行的文件）。

fileinput.filename函数返回你正在处理的文件名。

fileinput.lineno返回当前行的行数。计数是累计的，所以当你完成一个文件的处理并且开始下一个时，行数并不会复位，而是从上一个文件的最后行数开始计数。

fileinput.filelineno函数返回当前处理文件的行数。每次你处理完一个文件并且开始下一个时，行数都会复位为1.

fileinput.isfirstline函数在当前行是当前文件第一行时返回真，反之返回假。

fileinput.nextfile函数会关闭当前文件，跳到下一个。你跳过的行并不计入行数。当你知道你已经处理完文件时候这个函数就比较有用——比如每个文件都包含排序后的单词，而你正在寻找某个词。如果你已经在排序中已经找到了这个词的位置，你就能放心地跳到下一个文件。

fileinput.close函数关闭整个文件链，结束迭代。

范例

计算Python脚本行数。假设你写了一个Python脚本，想要计算行数。因为你想要程序在完成后继续起作用，所以你要在每一行的右侧加上作为注释的行号。你可以使用字符串格式化完成这个工作。让我们规定每个程序行最多有40个字符，然后把行号注释加在后面。列表10-6的程序展示了使用fileinput以及inplace参数的简单方式：

列表 10-6. *为Python脚本添加行号*

```
# numberlines.py

import fileinput

for line in fileinput.input(inplace=True):
    line = line.rstrip()
    num  = fileinput.lineno()
    print '%-40s # %2i' % (line,  num)
```

如果你这样运行程序：

```
$ python numberlines.py numberlines.py
```

你会得到类似于列表10-7的程序。注意，程序本身已经被更改，如果你这样运行多次，你最终会在每一行中添加多个行号。重新调用rstrip方法可以返回字符串的拷贝，右侧的空格都被去除（请参看第三章中的“字符串方法”一节，以及附录B中的表B-6）。

列表 10-7. *加入行号的Python程序*

```
# numberlines.py                    # 1
```

```
                                                    # 2
import fileinput                        # 3
                                                    # 4
for line in fileinput.input(inplace=1): # 5
  line = line.rstrip()                  # 6
  num  = fileinput.lineno()             # 7
  print '%-40s # %2i' % (line,  num)    # 8
```

警告 小心使用inplace参数——它很容易回调文件。你应该小心地测试自己的程序，而**不用**inplace设置（这样只会打印结果），确保程序工作正常后再修改文件。

关于另外的使用fileinput的范例，请参看本章后面关于random函数的一节。

集合、堆和双端队列
Sets, Heaps, and Deques

Python中有很多有用的数据结构，而且Python支持更一般的类型。其中类似字典（或者哈希表）、列表（或者动态数组）就是语言的一部分。其它的虽然不常用，有些时候也是能派上用场的。

集合
Sets

集合（Sets）在Python2.3中加入，Set类位于sets模块中。尽管你可以在现在的代码中创建Set实例，但是除非你想要兼容以前的程序，否则你很少用它。Python2.3的集合通过set类型实现为语言的一部分。目前你不需要引入sets模块了——你可以直接创建集合。

```
>>> set(range(10))
set([0, 1, 2, 3, 4, 5,  6, 7, 8, 9])
```

集合是由序列（或者其他可迭代对象）创建的。它们主要用于检查成员关系，所以副本是被忽略的。

```
>>> set([0, 1, 2, 3, 0, 1, 2, 3,  4, 5])
set([0, 1, 2, 3, 4, 5])
```

和字典一样，集合元素的顺序是任意的：

```
>>> set(['fee', 'fie',  'foe'])
set(['foe', 'fee', 'fie'])
```

检查成员关系时，你可以使用标准的集合操作（可能你在数学上学过），比如并和交，和进行整数的位操作是一样的（参见附录B）。比如你想要找两个集合的交集，可以使用union方法或者OR运算符“|”：

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
>>> a.union(b)
set([1, 2, 3, 4])
```

```
>>> a | b
set([1, 2, 3, 4])
```

这里列出了一些方法和对应的运算符，方法的名称已经清除地表明用途：

```
>>> c = a & b
>>> c.issubset(a)
True
>>> c <= a
True
>>> c.issuperset(a)
False
>>> c >= a
False
>>> a.intersection(b)
set([2, 3])
>>> a & b
set([2, 3])
>>> a.difference(b)
set([1])
>>> a -
b set([1])
>>> a.symmetric_difference(b)
set([1, 4])
>>> a ^ b
set([1, 4])
>>> a.copy()
set([1, 2, 3])
>>> a.copy()   is a
False
```

还有一些运算符和对应的方阿飞，比如基本方法add和remove。关于这方面更多的信息，请参看Python库参靠关于集合类型的一节（http://python.org/doc/lib/types-set.html）。

■**提示** 如果你需要个查看并且打印两个集合焦急的函数，你可以使用union方法的未绑定版本。这个方法很有用，比如用reduce：

```
>>> mySets =   []
>>> for i in   range(10):
...         mySets.append(set(range(i,i+5)))
...
>>> reduce(set.union, mySets)
set([0, 1, 2, 3, 4, 5,   6, 7, 8, 9, 10, 11,   12, 13])
```

集合是可变的，所以不能用做字典中的键。另外一个问题就是集合本身只能包含不可变（可哈希的）值，所以也就不能包含其它集合。因为集合的集合是经常使用的，所以这就是个问题了……很幸运，还有个frozenset类型，用来表现不可变（可哈希）的集合：

```
>>> a = set()
>>> b = set()
>>> a.add(b)
Traceback (most recent  call last):
  File "<stdin>", line  1, in ? TypeError:
set objects  are unhashable
>>> a.add(frozenset(b))
```

frozenset构造器创建给定集合的副本，不管你是想要将集合作为其他集合成员还是字典的键，frozenset都很有用。

堆

Heaps

另外一个地球人都知道的数据结构是堆（heap），它是优先队列的一种。优先队列让你可以任何顺序增加对象，并且能在任何时间（可能在增加同时）找到（也可能是移除）最小的元素。它比使用list的min方法要有效率得多。

事实上，Python中没有独立出来的堆——只有一个包含一些堆操作函数的模块。这个模块叫做heapq（q表示queue，队列），包括六个函数，其中前四个和堆操作直接相关。你必须用列表来实现堆。

heappush函数用于增加堆的项目。注意，你不能将它用于任何之前的列表中——只能用于使用各种堆操作建立的列表中。原因是元素的顺序很重要（尽管看起来是随意排列，元素并不是严密排序的）。

```
>>> from heapq import *
>>> from random import  shuffle
>>> data = range(10)
>>> shuffle(data)
>>> heap = []
>>> for n in  data:
...      heappush(heap,  n)
>>> heap
[0, 1, 3, 6,  2, 8, 4, 7, 9, 5]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 3, 6, 1, 8, 4, 7, 9, 5,  2]
```

■**注意** 元素的顺序并不像看起来那么随意。它们虽然不是严格排序的，但是也是有规则的：位于i位置的元素总比i//2位置的大（反过来说i位置的元素总比2*i以及2*i+1的元素小）。这是理解堆算法的基础。这个特性被称为**堆属性（heap property）**。

---

heappop函数弹出最小的元素——一般来说都是在索引0处的元素，并且会确保剩余元素中最小的占据这个位置（保持刚才提到的堆属性）。尽管弹出列表的第一个元素一般来说并不是很有效率，但是在这里不是问题，因为heappop在“幕后”会做一些精巧的混合：

```
>>> heappop(heap)
0
```

```
>>> heappop(heap)
0.5
>>> heappop(heap)
1
>>> heap
[2, 5, 3, 6,  9, 8, 4, 7]
```

heapify函数使用任意列表作为参数，在最小可能的混合量下，将其格式化为合法的堆（事实上是应用了刚才提到的堆属性）。如果你没有用heappush建立堆，在你使用heappush和heappop前应该使用这个函数。

```
>>> heap = [5, 8, 0, 3, 6, 7, 9,  1, 4, 2]
>>> heapify(heap)
>>> heap
[0, 1, 5, 3,  2, 7, 9, 8, 4, 6]
```

heapreplace函数一般来说不像其他的一样常用。它弹出堆的最小元素，并且将新元素推入。这样作比heappop之后再heappush要有效率。

```
>>> heapreplace(heap, 0.5)
0
>>> heap
[0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
>>> heapreplace(heap, 10)
0.5
>>> heap
[1, 2, 5, 3,  6, 7, 9, 8, 4, 10]
```

heapq模块中剩下的两个函数nlargest(n,iter)和nsmallest(n,iter)用来寻找任何可迭代对象iter中第n大或第n小的元素。你可以使用排序（比如使用sorted函数）和切片来完成这个工作，但是堆算法更快而且更有效地使用内存（而且更易用）。

双端队列（以及其他收集器）
Deques (and Other Collections)

双端队列（Double-ended queue，或称Deque）在你想要按照元素增加的顺序移除时非常有用。Python 2.4中增加了collections模块，包括了deque类型。

■**注意** Python2.4中的collections模块**只**包括deque类型，未来可能会加入二叉树（B-Tree）和菲波纳契堆（Fibonacci heap）。

双端队列从可迭代对象（比如集合）创建，而且有些非常有用的方法：

```
>>> from collections import deque
>>> q = deque(range(5))
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0,  1, 2, 3, 4, 5])
>>> q.pop()
5
```

```
>>> q.popleft()
6
>>> q.rotate(3)
>>> q
deque([2, 3,  4, 0, 1])
>>> q.rotate(-1)
>>> q
deque([3, 4,  0, 1, 2])
```

双端队列好用的原因是它允许有效率地从开头（左侧）增加和弹出元素，这和列表大相径庭。你还能有效地**旋转（rotate）**元素（也就是将它们从左到右替换）。双端队列对象还有extend和extendleft方法，extend和列表的extend方法差不多，extendleft则类似于appendleft。注意，extendleft使用的元素会反序出现在双端队列中。

time模块

time

time模块包括用于获得当前时间、操作时间日期、从字符串读取时间以及格式化时间为字符串的函数。日期可以用实数（从“新纪元”的1月1日0点开始到现在的秒数，对于UNIX来说是1970年），或者是包含有九个整数的元组。这些整数的意义列在了表10-5中，比如，元组：

```
(2002, 1, 21, 12, 2, 56, 0, 21, 0)
```

表示2002年1月21日12点2分56秒，星期一，并且是当年的第21天（无夏令时）。

表 10-5. *Python日期元组的组成*

索引	意义	值
0	年	比如2000，2001，等等
1	月	1–12
2	日	1–31
3	时	0–23
4	分	0–59
5	秒	0–61
6	周	0–6，周一为0
7	日期位置	1–366
8	夏令时	0，1，或–1

有些值是要解释一下的：秒的范围是0-61是为了应付闰秒和双闰秒。夏令时的数字是布尔值（真或假），但是如果你使用-1，mktime（利用从新纪元年到现在秒数转换元组到时间戳的函数）就会工作正常。time模块中最重要的函数已经列在表10-6中。

表 10-6. *time模块中最重要的函数*

函数	描述

asctime([tuple])	转换日期元组为字符串
localtime([secs])	转换秒数为日期元组，本地时间
mktime(tuple)	转换元组到本地时间
sleep(secs)	休眠（不做任何事件）secs秒
strptime(string[, format])	将字符串转换为日期元组
time()	当前时间（自从新纪元的秒数，UTC）

函数time.asctime将时间格式化为字符串，比如：

```
>>> time.asctime()
'Fri Dec 21 05:41:27 2001'
```

如果你不想获得当前时间，你还可以提供日期元组（比如从本地时间创建的）（有关更多精细格式化的内容，请参看标准文档中的strftime函数）。

time.localtime函数将实数（秒数）转换为本地时间的日期元组。如果你想获得全球统一时间，可以使用gmttime。

函数time.mktime将日期元组转换为秒数，它是localtime的反函数。

函数time.sleep让解释器等待给定的秒数。

函数time.strprtime将asctime格式化过的字符串转换为日期元组（可选的格式化参数和strftime的中的相同，请参看标准文档）。

函数time.time使用秒数格式返回当前（全球统一）时间，尽管每个系统的新纪元年可能不同，你可以使用它获得在某事件（比如函数调用）前后的时间，然后计算差值。这些函数的实例，请参看下一节中的random模块。

其他日期函数

表10-6列出的函数知识time模块的一部分。大多数函数和本节中的都类似或者相关。如果你需要这里没有介绍的函数的性能系你应该参看标准库参考中关于time模块的部分（http://python.org/doc/lib/module-time.html），你会找到自己想找的详细信息。

还有一些模块和time相关：datetime和timeit。你也能在库参考中找到它们的信息，timeit也会在第十六章中进行介绍。

random

random模块包括返回随机数的函数，可以用于模拟或者任何产生随机输出的程序。

■**注意** 事实上，所产生的数字都是伪随机数（pseudo-random）。也就是说他们看起是完全随机的，但是实际上有个可预测的系统作为基础。不过因为这个系统模块非常优秀可以制造随机，所以你也就不用过多担心（除非由于强加密原因使用的数字，这个时候它们就显得不够“强”以抵抗攻击——但是如果你用的是强加密的话，也就不用我来解释这些几出问题了。如果你需要真的随机数，你应该使用os模块的urandom函数。random模块内的SystemRandom类也是基于同一机制，可以让你的

数据接近真正的随机性。

一些这个模块中的重要函数已经列在表10-7中。

表 10-7. *random模块中一些重要的函数*

函数	描述
random()	返回0 ≤ *n* < 1之间的随机数n
getrandbits(n)	使用长整型形式返回n个位
uniform(a, b)	返回a ≤ *n* < b之间的随机实数n
randrange([start], stop, [step])	返回范围range(start,stop,step)内的随机数
choice(seq)	seq从序列seq中返回随意元素
shuffle(seq[, random])	在seq内对元素进行混排
sample(seq, n)	从序列seq中选择n个随机且独立的元素。

random.random函数是最基本的随机函数，他只是返回0到1之间位随机数n。除非这就是想要的，你应该使用其它提供额外功能的函数。random.getrandbits使用长整型形式返回给定数目的位（二进制数）。如果你处理的是真正的随机事务（比如加密），这个函数尤其有用。

random.uniform在提供给两个数值参数a和b的情况下返回在a到b范围内的随机（均一分部的）实数n，比如你需要随机角度，你可以使用uniform(0,360)。

random.randrange是标准的在给定范围内产生随机整数的函数。比如想要获得1到10（包括10）内的随机数，你可以使用randrange(1,11)（或者使用randrange(10)+1），如果你想要获得小于20的随机正奇数，可以使用randrange(1,20,2)。

random.choice从给定序列中（均一地）选择随机元素。

random.shuffle函数将给定（可变）序列的元素大乱，每种排列的可能性都是近似相等的。

random.sample函数从给定序列中选择给定数目的元素，确保元素互不相同。

■**注意** 对于统计学来说，还有些近似uniform的函数返回其他形式分布的样本随机数，比如betavariate、expoential、Gaussian等等。

范例

在给定范围内生成随机日期。下面的例子中，我使用了一些之前讲过的time模块中的函数。首先让我们获得时间间隔（2005年）限制的实数。你可以将日期作为元组表达（使用-1表示周几、第几天和夏令时，以便让Python自己计算），并且对这些元组调用mktime：

```
from random import *
from time import *
date1 = (2005, 1, 1, 0, 0, 0, -1, -1, -1)
time1 = mktime(date1)
date2 = (2006, 1, 1, 0, 0, 0, -1, -1, -1)
time2 = mktime(date2)
```

然后你就能再这个范围内均一地生成随机数（包括上限）：

```
>>> random_time = uniform(time1, time2)
```

然后，你可以将数字转换合法日期：

```
>>> print asctime(localtime(random_time)) Mon Jun 24 21:35:19 2005
```

创建电子投色子机。本例中，假设我们要求用户选择投掷几个色子以及每个色子应该有多少面。投色子机制可以用`randrange`和`for`循环实现：

```
from random import randrange
num   = input('How many dice? ')
sides = input('How many sides per die? ')
sum = 0
for i in range(num): sum += randrange(sides) + 1
print 'The result is', sum
```

如果你将代码放置在脚本文件中并且执行，那么你会得到如下的交互：

```
How many dice? 3
How many sides per die? 6
The result is 10
```

创建运势预测程序。假设你将运势分行放在了文本文件中，你就可以使用刚才说道的`fileinput`模块将运势都存入列表，再进行随机选择：

```
# fortune.py
import fileinput, random
fortunes = list(fileinput.input())
print random.choice(fortunes)
```

```
In UNIX, you could test this on the standard dictionary file
/usr/dict/words to get a random word:
```
UNIX下，你可以使用标准字典文件`/usr/dict/words`获取随机单词进行测试。

```
$ python fortune.py /usr/dict/words
dodge
```

创建自动发牌机。你想要自己的程序能够在每次敲击回车的时候处理一张牌。而且你要确保你不会获得同一张牌多于一次。首先你要创建“一副牌”——字符串列表：

```
>>> values =  range(1, 11) + 'Jack Queen King'.split()
>>> suits = 'diamonds clubs hearts spades'.split()
>>> deck = ['%s of %s'  % (v, s) for v in values for s in suits]
```

你创建牌目前不太适合进行游戏，让我们看看现在的牌：

```
>>> from pprint import   pprint
>>> pprint(deck[:12])
['1 of diamonds',
 '1 of clubs',
 '1 of hearts',
 '1 of spades',
 '2 of diamonds',
 '2 of clubs',
 '2 of hearts',
 '2 of spades',
 '3 of diamonds',
 '3 of clubs',
 '3 of hearts',
 '3 of spades']
```

太整齐了，不是吗？很容易解决；

```
>>> from random import   shuffle
>>> shuffle(deck)
>>> pprint(deck[:12])
['3 of spades',
 '2 of diamonds',
 '5 of diamonds',
 '6 of spades',
 '8 of diamonds',
 '1 of clubs',
 '5 of hearts',
 'Queen of diamonds',
 'Queen of hearts',
 'King of hearts',
 'Jack of diamonds',
 'Queen of clubs']
```

注意，我为了省地方只打印了前12张牌。你可以自己看看整幅牌。

最后，为了让Python在你每次按回车且还有剩余牌的时候给你一张牌，你只需要创建一个while循环。假设你将建立牌的代码放在了程序文件中，你只需要将下面这行加进去就可以：

```
while deck: raw_input(deck.pop())
```

■**注意** 如果你在交互式解释器中尝试这里的while循环，你会注意到每次你按下回车的时候都会打印出一个空字符串，因为raw_input返回你输入的内容（也就是空白），并且将其打印出来。在一般的程序中，从raw_input返回的值都会忽略掉。为了能够交互地“忽略”它，你只要把raw_input的值赋到一些你不用的变量上就可以，并且将其命名为ignore这类名字。

---

shelve

下一章中，你讲学会如何在文件中存储数据，但是如果你期望一个简单的存储防范，shelve模块可以满足你大部分的需要。你要做的就是提供给它文件名。shelve中唯一有用的模块是open。当你（使用文件名）调用它的时候，它会返回一个Shelf对象，你可以用它来存储东西，你可以把它看作一般的字典（除了键一定要为字符串，当你存储完成时（并且希望存储到磁盘中），可以调用close方法。

潜在的陷阱
A Potential Trap

意识到shelve.open返回的对象并不是普通的映射是很重要的，就像下面这个例子中所演示的：

```
>>> import shelve
>>> s = shelve.open('test.dat')
>>> s['x'] =  ['a', 'b', 'c']
>>> s['x'].append('d')
>>> s['x']
['a', 'b', 'c']
```

'd'去哪了？

解释很简单：当你在shelf对象中查找元素时，这个对象都会从已经存储的版本中重建，当你将元素赋到键上时，它就被存储了。上个例子中发生的事情是这样的：

1. 列表['a', 'b', 'c']存储在键x下。
2. 存储的版本被重新获得，并且创建新的列表，而'd'被增加到拷贝中。修改的版本还没有被存储！
3. 最终，原始版本被重新获得——没有'd'。

为了正确地使用shelve模块修改存储的对象，你必须将临时变量绑定到重获的靠背上，并且在它被修改后重新存储这个拷贝：

```
>>> temp = s['x']
>>> temp.append('d')
>>> s['x'] =  temp
>>> s['x']
['a', 'b', 'c', 'd']
```

感谢Luther Blissett指出这个问题。

Python2.4以前的版本还有个解决方法：设定open函数的writeback参数为True。如果这样做了，所有你读取或者赋值到shelf的数据结构都会保存在内存（缓存）中，并且只在你关闭shelf时候写回到磁盘中。如果你处理的数据不大，并且你不想考虑这些问题，那么writeback设为True（确保你在最后关闭了shelf）的方法还是不错的。

范例

列表10-8演示了使用shelve模块的简单数据库程序。

列表 10-8. *简单数据库程序*

```
# database.py
import sys, shelve

def store_person(db):
    """
    Query user for data and store it in the shelf object
    """

    pid = raw_input('Enter unique ID number: ')
    person =  {}
    person['name']  = raw_input('Enter name: ')
    person['age']   = raw_input('Enter age:  ')
    person['phone'] = raw_input('Enter phone number:  ')

    db[pid] = person

def lookup_person(db):
    """
    Query user for ID and desired field, and fetch the corresponding data from
    the shelf object
    """
    pid = raw_input('Enter ID number: ')
    field = raw_input('What would you like  to know? (name, age, phone)    ')
    field = field.strip().lower()
    print field.capitalize() + ':', \
          db[pid][field]

def print_help():
    print 'The available commands are:'
    print 'store  : Stores information about a person'
    print 'lookup : Looks up a person from  ID number'
    print 'quit   : Save changes  and exit'
    print '?      : Prints this message'

def enter_command():
    cmd = raw_input('Enter command (? for help): ')
    cmd = cmd.strip().lower()
    return cmd

def main():
    database  = shelve.open('C:\\database.dat')
    try:
          while True:
                cmd = enter_command()
                if   cmd == 'store':
           store_person(database)
                elif cmd == 'lookup':
           lookup_person(database)
```

```
            elif cmd == '?':
        print_help()
            elif cmd == 'quit':
        return
    finally:
        database.close()

if __name__ == '__main__': main()
```

警告 如你所见，程序明确了文件名为C:\database.dat。如果你偶然有个shelve模块可以使用的同名的数据库，数据库就会被修改。所以保证你的数据库使用的文件名还没有被用过。运行这个程序后，所指定的文件就会出现

列表10-8的程序有一些很有意思的特性：

- 我把一切都放在了函数中，让程序更加结构化（一个改进是将函数组织为类的方法）。
- 我把主程序放在了main函数中，使用if __name__=='__main__'调用。意味着你可以从其它的程序中将这个程序作为模块导入，并且能调用main函数。
- 我在main函数中打开数据库（shelf），然后将其作为参数传给需要的函数。我也可以使用全局变量，但是在大多数情况下最好避免用全局变量——除非你有充足的理由。
- 在读取了一些值后，我对修改后的版本调用strip和lower函数，因为如果提供的键在数据库中已经存储的话，那么它们两个应该完全一样。如果你总是在用户的输入中使用strip和lowser，你可以让他/她随意输入大小写和添加空格。注意，我在打印打印域名称时候使用了capitalize函数。
- 我使用try/finally确保数据库被正确关闭。你永远不知道什么时候某地方会出错（获得异常），如果程序在没有正确关闭数据库情况下被终止，你可能会得到一个损坏的数据库文件，这样也就没用了。使用try/finally可以避免这种情况。

那么让我们测试一下这个数据库吧。下面是一些程序和我之间的交互：

```
Enter command (? for help): ? The available commands   are:
store   : Stores information about a person lookup : Looks up a person from ID number quit    :
Save changes and exit
?           : Prints this message
Enter command (? for help): store
Enter unique   ID number: 001
Enter name: Mr. Gumby
Enter age: 42
Enter phone number: 555-1234
Enter command (? for help): lookup
Enter ID number: 001
What would you like to   know? (name, age, phone) phone
Phone: 555-1234
Enter command (? for help): quit
```

交互的过程并不是十分有趣。我用普通的字典也能获得和shelf对象一样的效果。但是现在我退出了程序，让我们看看在我重新打开它的时候发生了什么？

```
Enter command (? for help): lookup
Enter ID number: 001
What would you like to    know? (name, age, phone) name
Name: Mr. Gumby
Enter command (? for help): quit
```

可以看到，程序读出了我第一次创建的文件，而Mr.Gumby还在！

你可以随意实验这个程序，看看你是否还能扩展功能和增加它的用户友好性。有没可能想要做个自己用的版本？做个你的唱片收集的数据库呢？或者帮助你记录那些借你书的朋友名单的数据库（估计我会用这个版本）？

---

re

有些人，当面临一个问题的时候，会想“我知道，我可以使用正则表达式。”。现在他们有两个问题了。

—Jamie Zawinski

re模块包含对**正则表达式（regular expression）**的支持。如果你之前听说过正则表达式，可能就知道它有多强大。如果你没有，做好吃惊的准备吧。

但是你应该注意，在开始掌握正则表达式时候会有点困难（好吧，其实是很难）。学习它们的关键是——（在文档中）查看你对于所明确任务的需要。你不用死记它们。本章将会对re模块主要特性和正则表达式的进行描述，可以让你开个头。

> **提示** 除了标准文档外，Andrew Kuchling的“Regular Expression HOWTO”（http://amk.ca/python/howto/regex/）也是学习Python中正则表达式的有用资源。

什么是正则表达式？
What Is a Regular Expression?

正则表达式（也叫做正则，regex或regexp）是可以匹配文本片段的模式。最简单的正则表达式就是纯字符串，可以匹配其自身。换句话说，正则表达式“python”会匹配字符串“python”。你可以用这种匹配行为搜索文本，并且用计算后的值替换一些当前模式，或者将文本切分成数片。

通配符
The Wildcard

正则表达式可以匹配多于一个字符串，你可以使用一些特殊字符创建这类模式。比如点号（.）可以匹配任何字符（除了新行符号），所以正则表达式'.ython'可以匹配字符串'python'和'jython'。它还能匹配'qython'、'+ython'或者' ython'（第一个字母是空格），但是不会匹配类似'cpython'或者'ython'的字符串，因为点号会匹配一个字母，而不是两个或零个。

因为它可以匹配“任何字符串”（除新行外的任何字符），点号就被叫做**通配符（wildcard）**。

对特殊字符进行转义
Escaping Special Characters

当你使用特殊字符时，理解当你将它们用作普通字符时所产生的问题是很重要的。比如假设你要匹配字符串'python.org'，你就直接用'python.org'？你可以这么做，但是这样也会匹配'pythonzorg'，这可不是你期望的（点号可以匹配除新行外任何字符）。为了让特殊字符表现得像普通字符一样，你可以对它进行**转义（escape）**，就像我在第一章中对引号进行转义所做的一样。你可以在它前面加上反斜线。本例中，你可以使用'python\\.org'，这样只会匹配'python.org'。

■**注意** 为了获得re模块需要的单斜线，你需要在字符串中使用双反斜线——为了从解释器中进行转义。所以你需要两个级别的转义：（1）从解释器中转义，（2）从re模块中转义（事实上，有些情况下你可以使用单斜线，解释器会自动转义，但是别指望这个）。如果你厌烦了使用双斜线，那么可以使用自然字符串，比如`r'python\.org'`。

字符集合
Character Sets
匹配任何字符可能很有用，但是有些时候你希望获得更多控制。你可以使用中括号创建字符集合（character set）。字符集合会匹配任何具包括的字符，所以'[py]ython'会匹配'python'和'jython'而不是其它的。你可以使用范围，比如'[a-z]'会匹配任何从a-z的字符，你可以在一个后面写另外一个以联合使用，比如'[a-zA-Z0-9]'会匹配任何大小写字幕和数字（注意字符集合只能匹配**一个**这样的字符）。

为了反转字符集合，可以在开头使用^字符，比如'[^abc]'会匹配任何**除了**a、b和c之外的字符。

字符集合中的特殊字符

一般来说，如果你想要类似点号、星号和问号的特殊字符在模式中用作文本字符时，需要用反斜线转义，而不是用作正则运算符。在字符集合内，对这些字符进行转义一般来说是没必要的（尽管是完全合法的）。你应该时刻记住下面的准则：

你需要对出现在字符集合开头的脱字符号（^）进行转义，除非你希望它用作否定运算符（换句话说，不要将它放在开头，除非你就是准备那么用）。

类似地，右中括号（]）和横线（-）应该放在字符集合的开头或者用反斜线转义（事实上，如果你希望的话横线也能放在末尾）。

替代方案和子模式
Alternatives and Subpatterns

当你独立地匹配每个字符的时候字符集合很好用，但是如果想匹配字符串'python'和'perl'呢？你就不能使用字符集合或者通配符进行详细模式的指定。不过你还有特殊字符可以选择：“管形符号”（|）。所以你的模式可以写成'python|perl'。

但是，有些时候你不想对**整个**模式使用选择运算符——只想用一部分。这时你需要用圆括号括起需要的部分，或称**子模式（subparttern）**。前例可以写成'p(ython|erl)'。（注意，术语字模式也可以用于单字符）

可选项和重复子模式
Optional and Repeated Subpatterns

在子模式后面加上问号，它就变成了可选项。它可能出现在匹配字符串中，但是不是严格必需的。举例来说，（稍微有点难懂的）模式：

```
r'(http://)?(www\.)?python\.org'
```

会匹配下列字符串（而不会匹配其它的）：

```
'http://www.python.org'
'http://python.org'
'www.python.org'
'python.org'
```

有些值得一记的东西：

- 我对点号进行了转义，避免它被作为通配符使用。
- 我使用自然字符串减少反斜线的用量。
- 每个可选子模式都用圆括号括起。
- 可选子模式出现与否均可，而且互相独立。

问号表示子模式可以出现一次或者根本不出现。还有些让你的模式重复多于一次的运算符：

(pattern)*	模式重复0次或以上
(pattern)+	模式重复1次或以上
(pattern){m,n}	模式重复m到n次

举例来说，r'w*\.python\.org'会匹配'www.python.org'，也会匹配'.python.org'、'ww.python.org'和'wwwwwww.python.org'。类似地，r'w+\.python\.org'匹配'w.python.org'但不匹配'.python.org'，而r'w{3,4}\.python\.org'只匹配'www.python.org'和'wwww.python.org'。

■**注意** 这里使用的术语“匹配”（match）表示模式匹配整个字符串。而接下来要说到的match函数只要匹配字符串的开始。

字符串的开始和结尾
The Beginning and End of a String

目前为止，你所看到的模式匹配都是针对整个字符串的，但是你也能寻找匹配模式的子字符串，比如在字符串'www.python.org'中的子字符串'www'会和模式'w+'匹配。当你寻找这样的子字符串时，确定子字符串位于整个字符串的开始还是结尾是很有用的。比如你只想在字符串的开头匹配'ht+p'，你就可以使用脱字符（^）标记开始：'^ht+p'会匹配'http://python.org（以及'htttttp://python.org'但是不匹配'www.http.org'。类似地，字符串结尾用美元符号（$）表示。

■**注意** 正则运算符的完全列表可以参看标准库参考的“Regular Expression Syntax”一节（http://python.org/doc/lib/re-syntax.html）。

---

re模块的内容
Contents of the re Module

如果你不知道如何应用，了解如何书写正则表达式还是不够。re模块包含一些有用的正则函数。其中最重要的一些已经列在表10-8中：

表 10-8. *re模块中一些重要的函数*

函数	描述
compile(pattern[, flags])	从正则字符串创建模式对象
search(pattern, string[, flags])	在字符串中寻找模式
match(pattern, string[, flags])	在字符串开始匹配模式
split(pattern, string[, maxsplit=0])	以模式的出现分割字符串
findall(pattern, string)	将字符串中所有模式的出现作为列表返回
sub(pat, repl, string[, count=0])	将字符串中出现的pat模式的匹配用repl替换
escape(string)	将字符串中所有特殊正则字符转义。

函数re.compile将正则表达式（以字符串书写）转换为模式对象，可以用于更优效率的匹配。如果你使用字符串表示正则表达式，而调用search或者match函数，它们也会在内部将字符串转换为正则对象。使用compile完成一次转换之后，在每次使用模式的时候就不用进行转换。模式对象自己也有查找/匹配的函数作为方法，所以re.search(pat,string)（pat表示写为正则表达式的字符串）等价于pat.search(string)（pat为使用compile创建的模式对象）。使用过的compile的正则对象也能用于普通的re函数。

函数re.search会在给定字符串中寻找第一个匹配给定正则表达式的子字符串。一点找到，就会返回MatchObject（值为True），否则返回None（值为False）。因为返回值等同于事实值，所以可以像这样用于条件语句：

```
if re.search(pat,   string):
    print 'Found it!'
```

但是如果你需要更多关于被匹配子字符串的信息，你可以检查返回的MatchObject对象（会在下一节说道）。

函数re.match会在给定字符串的开头对正则表达式进行匹配。所以match('p','python'）返回True，而match('p', 'www.python.org')返回False（返回值和search返回值相同）。

■**注意** match函数在匹配到字符串开头的子字符串时会进行报告，模式并**不**需要匹配整个字符串。如果你想这么做，可以在模式的结尾加上美元符号。美元符号会对字符串的末尾进行匹配，然后“延展”整个匹配。

函数resplit会根据模式的出现分割字符串。它类似于字符串方法split，不过是用正则表达式代替了固定的字符。比如字符串方法split可以让你用字符串','的出现分割字符串，而

re.split可以让你用任意逗号队列的出现分割字符串：

```
>>> some_text = 'alpha, beta,,,,gamma     delta'
>>> re.split('[, ]+', some_text)
['alpha', 'beta', 'gamma', 'delta']
```

■**注意** 如果模式包含小括号，那么括起来的组会散布在分割后的子字符串之间。

如你所见，返回值是子字符串的列表。maxsplit参数表示最多返回的分割后字符串：

```
>>> re.split('[, ]+', some_text,    maxsplit=2)
['alpha', 'beta', 'gamma     delta']
>>> re.split('[, ]+', some_text,    maxsplit=1)
['alpha', 'beta,,,,gamma     delta']
```

函数re.findall返回给定模式在字符串中的所有出现的列表。比如要在字符串中查找所有的单词，你可以这么做：

```
>>> pat = '[a-zA-Z]+'
>>> text = '"Hm... Err    -- are you sure?" he said, sounding insecure.'
>>> re.findall(pat, text)
['Hm', 'Err', 'are', 'you', 'sure', 'he', 'said', 'sounding',    'insecure']
```

或者你可以查找标点符号：

```
>>> pat = r'[.?\-",]+'
>>> re.findall(pat, text)
['"', '...',    '--', '?"', ',', '.']
```

注意，横线（-）被转义了，所以Python不会将其解释为字符串范围的一部分（比如a-z）。

函数re.sub用于以给定替换内容替换最左端不重叠的模式的出现。考虑下面的例子：

```
>>> pat = '{name}'
>>> text = 'Dear {name}...'
>>> re.sub(pat, 'Mr. Gumby', text)
'Dear Mr. Gumby...'
```

请参看本章后面“在替换中使用组号和函数”一节，获取关于如何让这个函数更有效率的信息：

函数re.escape用于对字符串中所有可能被解释为正则运算符的字符进行转义。如果你的字符串很长还有很多特殊字符，而你又不想输入一大堆反斜线的时候，可以使用这个函数。或者你的字符串来自于用户（比如通过raw_input函数），且要用作正则表达式的一部分。下面是个例子：

```
>>> re.escape('www.python.org')
'www\\.python\\.org'
>>> re.escape('But where is the ambiguity?')
'But\\ where\\ is\\ the\\ ambiguity\\?'
```

■**注意** 你可能会注意到表10-8中有些函数有叫做flags的可选参数。这个参数用做表示正则表达式如何被解释的标志。关于它的更多信息，你可以参看标准库参考的re模块的一节：http://python.org/doc/lib/module-re.html。这个标志在“Module Contents”部分有介绍。

匹配对象和组
Match Objects and Groups

所有对于字符串某区域匹配模式的re函数都会在匹配达成的时候返回MatchObject对象。这些对象包括匹配到模式的子字符串的信息。他们还包含了哪个模式匹配了子字符串哪部分的信息——这些“部分”叫做**组（group）**。

组简单说来就是放置在括号内的子模式。组的序号取决于左侧的括号数。组0就是整个模式，所以在下面的模式中：

```
'There (was a (wee) (cooper)) who (lived in Fyfe)'
```

有这些组：

```
0    There was   a wee cooper who lived in Fyfe
1    was a wee   cooper
2    wee
3    cooper
4    lived in Fyfe
```

如果组中包含诸如通配符或者重复运算符之类的特殊字符的话，你可能会感兴趣是哪个给定组达成了匹配，比如在下面的模式中：

```
r'www\.(.+)\.com$'
```

组0包含整个字符串，而组1包含位于'www'和'.com'之间的一切。像这样创建模式的话，你就可以取出字符串中感兴趣的部分。

关于re MatchObject的重要方法已列在表10-9中：

表 10-9. *re MatchObject的重要方法*

方法	描述
group([group1, ...])	获取给定子模式（组）的出现
start([group])	返回给定组出现的位置
end([group])	返回给定组的结束（和切片一样，是不包括限制数在内的）位置
span([group])	返回一个组的开始和结束位置

group方法返回匹配模式中给定组的（子）字符串。如果没有设定组号，组0会默认私用。如果给定一个组号（或者你只用默认的0），会返回单个字符串。否则会将给定数目的字

符串作为元组返回。

■**注意** 除了整体匹配外（组0），你只能拥有99个组，范围从1-99。

start方法返回给定组出现的开始索引（默认为0，整个模式）。

方法end类似于start，但是返回结束索引加1。

方法span返回给定组出现的（开始，结束）位置的元组（默认为0，整个模式）。

```
>>> m = re.match(r'www\.(.*)\..{3}', 'www.python.org')
>>> m.group(1)
'python'
>>> m.start(1)
4
>>> m.end(1)
10
>>> m.span(1)
(4, 10)
```

在替换中使用组号和函数
Using Group Numbers and Functions in Substitutions

在使用re.sub的第一个例子中，我知识把一个字符串用其他的什么替换掉了。我用replace这个字符串方法（在第三章中的“字符串方法”一节介绍）也能轻松达到同样效果。当然，正则表达式很有用，因为它们允许你以更灵活的方式搜索，而且它们也允许你进行功能更强大的替换。

最简单的实践re.sub的强大功能的方式就是在替换字符串中使用组号。任何以在替换内容中以'\\n'形式出现的转义需略都会被匹配模式中组n的字符串替换掉。举例来说，假设你把“*某某”用“某某”替换掉，原来的形式是在文本文档（比如Email）中进行强调的方式，而替换的内容是相应的HTML代码（用于网页）。让我们首先建立正则表达式：

```
>>> emphasis_pattern =   r'\*([^\*]+)\*'
```

注意，正则表达式很容易变得难读，所以为了让某人（包括你自己在内！）读懂代码，使用有意义的变量名（或者加上一两句注释）是很重要的。

■**提示** 让你的正则表达式更加易读的方式是在re函数中使用VERBOSE标志。它允许你在模式中增加空白（空格、tab、新行等等），re则会忽略它们——除非你将其放在字符类或者用反斜线转义。你还能在VERBOSE正则式中添加注释。下面的模式对象等价于刚才写的模式，但是使用了VERBOSE标志：

```
>>> emphasis_pattern =   re.compile(r'''
...                 \*          #   Beginning   emphasis tag -- an asterisk
...                 (           #   Begin group for capturing phrase
...                 [^\*]+      #   Capture anything except asterisks
...                 )           #   End group
```

```
...                 \*          #   Ending emphasis tag
...                 '''', re.VERBOSE)
...
```

现在我的模式已经搞定，我可以使用re.sub进行替换：

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', 'Hello,   *world*!')
'Hello, <em>world</em>!'
```

如你所见，我成功的将文本转换为了HTML。

但是你可以用函数作为替换内容让你的替换行为更加强大。函数作为MatchObject的唯一参数，返回的字符串将会用作替换内容。换句话说，你可以对匹配到的子字符串做任何事，并且可以将处理替换内容的过程细化。这个功能用在什么地方呢，你可能会问。一旦你开始使用正则表达式，肯定会找到用它的地方。你可以在下面的“范例”中看到一个这样的程序。

贪婪和非贪婪模式

重复运算符默认是**贪婪（greedy）**的。意味着它会尽可能多的进行匹配。比如假设我重写了刚才用到下面模式的程序：

```
>>> emphasis_pattern =   r'\*(.+)\*'
```

它会匹配星号加上一个或多个字母然后又是一个星号的字符串。看起来完美吧？但是实际上不是：

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '*This*   is *it*!')
'<em>This* is *it</em>!'
```

如果你所见，模式匹配了从开始星号到最后星号的所有内容——包括中间的两个星号！也就意味着它是贪婪的：将所有的据为己有。
本例中，你当然不希望出现这种贪婪行为。前面的解决方案（使用字符结合匹配任何**不是**星号的内容）在你知道某个特殊字母是不合法的时候工作正常。但是假设另外一种情况：如果你使用了“**某某**”进行着重强调呢？现在在所强调的不分钟包括单星号已经不是问题了，但是如果避免过于贪婪？
事实上非常简单，你只要使用重复运算符的非贪婪版本即可。所有的重复运算符都可以在后面加上一个问号变成非贪婪版本：

```
>>> emphasis_pattern =   r'\*\*(.+?)\*\*'
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '**This** is **it**!')
'<em>This</em> is <em>it</em>!'
```

这里我用+?运算符代替了+，意思是模式会对一个或者多个符合通配符要求的出现内容进行匹配。但是它会尽可能少的进行匹配，因为它是非贪婪的。它仅会匹配达到下一个模式末尾——也就是'**'发生处的最小需要。可以看到，工作正常。

范例

找出Email的发信人。你有没有把Email存为文本文件过？如果有的话，你会看到文件包含一大堆基础的不可阅读的信息，类似于列表10-9中列出的：

列表 10-9. *一组（编造的）Email首部*

```
From foo@bar.baz  Thu Dec 20 01:22:50 2004
Return-Path: <foo@bar.baz>
Received: from xyzzy42.bar.com (xyzzy.bar.baz [123.456.789.42])
          by frozz.bozz.floop (8.9.3/8.9.3) with ESMTP  id BAA25436
          for <magnus@bozz.floop>;  Thu, 20 Dec 2004 01:22:50 +0100 (MET)
 Received: from [43.253.124.23] by bar.baz
            (InterMail vM.4.01.03.27 201-229-121-127-20010626) with ESMTP
            id  <20041220002242.ADASD123.bar.baz@[43.253.124.23]>;
            Thu, 20 Dec 2004 00:22:42 +0000


User-Agent: Microsoft-Outlook-Express-Macintosh-Edition/5.02.2022
Date: Wed, 19 Dec 2004  17:22:42 -0700
Subject: Re:  Spam
From: Foo Fie <foo@bar.baz>
To: Magnus Lie Hetland  <magnus@bozz.floop> CC: <Mr.Gumby@bar.baz>
Message-ID: <B8467D62.84F%foo@baz.com>
In-Reply-To: <20041219013308.A2655@bozz.floop> Mime-version: 1.0
Content-type: text/plain; charset="US-ASCII" Content-transfer-encoding: 7bit
Status: RO
Content-Length: 55
Lines: 6

So  long, and  thanks for all the spam!


Yours,

Foo Fie
```

让我们找出这封Email是谁发的。如果你直接看文本的话我能肯定你可以指出本例中（尤其在你看邮件本身的末尾时）的发信人。但是你能找出一般模式吗？你怎么能发发信人的名字取出而不带着Email地址？或者你能把首部中所有出现的Email地址列表吗？让我们先处理第一个任务。

包含有发信人的一行以字符串'From: '开始，以放置在尖括号（< 和 >）中的Email地址结尾。你要的文本夹在中间。如果你使用fileinput模块的话就简单了。解决这个问题的程序列在列表10-10中：

■**注意** 如果你希望的话也可以不使用正则表达式解决这个问题。你可以使用email模块。

列表 10-10. *寻找Email发信人的程序*

```
# find_sender.py import fileinput, re
pat = re.compile('From: (.*?) <.*>$')
for line in fileinput.input():
  m = pat.match(line)
  if m: print m.group(1)
```

你可以像这样运行程序（假设Email内容存储在文本文件message.eml）中：

```
$ python find_sender.py message.eml
Foo Fie
```

关于这个程序，你应该注意到：

- 我用compile函数处理了正则表达式，以便更有效率。
- 我将需要取出的子模式放在圆括号中作为组
- 我使用非贪婪模式匹配名字，因为我想要在我第一次到达左尖括号（或者说是前面的空格）时停止匹配。
- 我使用美元符号表明我要匹配整行
- 我使用if语句确保在我试图从特定组中取出匹配内容时，匹配的确存在。

为了列出首部总所有的Email地址，你需要建立只匹配Email的正则表达式。然后可以使用fineall方法寻找每行中所有Email的出现。为了避免收入副本，你可以将地址保存在集合（本章前面介绍过）中。最后，你取出所有的键，排序，打印：

```
import fileinput, re
pat = re.compile(r'[a-z\-\.]+@[a-z\-\.]+',    re.IGNORECASE)
addresses = set()
for line in fileinput.input():
      for address in pat.findall(line):
            addresses.add(address)
for address in sorted(addresses):
      print address
```

运行程序的时候会输出如下结果（以刚才的Email文件作为输入）：

```
Mr.Gumby@bar.
baz
foo@bar.baz
foo@baz.com
magnus@bozz.floop
```

注意，在排序的时候，大写字母要比小写字母靠前。

■**注意** 我在这里没有严格地照着问题描述去做。问题的内容是要在首部找出Email地址，但是这个程序找出了整个文件中的地址。为了避免这种情况，在有空行的情况下，你可以调用fileinput.close()，因为首部不包含空行，到了空行就证明工作完成了。或者你可以使用fileinput.nextfile()开始处理下一个文件——如果文件多于一个的话。

创建模板系统

Making a template system

模板是你可以放入具体值以获得某类完成文本的文件。比如你可以做一个只需要插入收件人姓名的邮件模板。Python已经有此类先进的模板机制：字符串格式化。但是使用正则表达式可以让你的系统更加先进。假设你需要把所有'[somethings]'（域）替换为something在Python中计算值的结果，所以字符串

```
'The sum of 7 and 9 is  [7 + 9].'
```

应该被翻译为：

```
'The sum of 7 and 9 is  16.'
```

你还可以在域内进行赋值，所以字符串

```
'[name="Mr. Gumby"]Hello, [name]'
```

应该被翻译为：

```
'Hello, Mr. Gumby'
```

看起来像是复杂的工作，但是让我们重看一下可用的工具：

- 你可以使用正则表达式匹配域，提取内容。
- 你可以用eval计算字符串值，提供包含作用域的字典。你可以在try/except语句内进行这些工作。如果并出了SyntaxError异常，你可能要用exec执行出现异常的语句（比如赋值）。
- 你可以用exec执行赋值字符串（和其他语句），在字典内保存模板的作用域。
- 你可以使用re.sub将估值的结果替换为处理后的字符串。

提示 如果你的看起来复杂的可怕，将其拆解为小一些的部分总是会有用的。还有，要总结出解决问题所使用的工具。

列表10-11是一个简单的实现：

列表 10-11. *一个模板系统*

```
# templates.py

import fileinput, re

# 匹配位于中括号中的域
field_pat = re.compile(r'\[(.+?)\]')


        # 用这个变量收集
        scope = {}

        # 用re.sub
        def replacement(match):
            code = match.group(1)
            try:
```

```
            # 如果域可以估值，就返回
            return str(eval(code, scope))
        except SyntaxError:
            # 否则执行相同作用域内的语句
            exec  code in scope
            # 返回空字符串
            return ''

    # 将所有文本作为一个字符串
    # 还有其他的方法，见第十一章
    lines = []
    for line in fileinput.input():
        lines.append(line)
    text = ''.join(lines)

    # 将所有的出现用模式替换
    print field_pat.sub(replacement,  text)
```

简单来说，程序做了下面的事情：

1. 定义了匹配域的模式
2. 创建用于模板的用做作用域的字典
3. 定义具有下列功能的函数：
   a. 将组1从匹配中取出，放入code；
   b. 试着对code用作用域字典（命名空间）进行估值，将结果转换为字符串返回。如果成功的话，域就是个表达式，一切正常。否则（比如升出了SyntaxError），跳至步骤c
   c. 执行在相同命名空间（作用域字典）内的域，返回空字符串（因为赋值语句不能估值）。
4. 使用fileinput读取所有可能的行，将其放入列表，联合成一个大字符串。
5. 将所有field_pat出现的地方用re.sub替换，打印结果。

■**注意** 将所有行放入列表在最后联合比用这种方法更有效率：

```
# 不要这样做
text = ''
for line in fileinput.input():
    text += line
```

尽管看起来简洁，但是每个赋值语句都要创建新的由就字符串和新增加的组成的字符串，这样就会造成严重的资源浪费，让你的程序运行缓慢。切记不要这样做。如果你希望有读取文件所有文本的更有效率的方法，请参看第十一章。

我用15行代码（不包含空行和注释）就创建了一个强大的模板系统。希望你已经认识到当你使用标准库时候Python有多么强大。让我们用测试这个模板系统的例子来结束。试着用程序来运行列表10-12中的文件：

列表 10-12. *简单的模板例子*

```
[x =  2]
[y =  3]
The sum of [x] and [y]  is [x + y].

You should see this:


The sum of 2  and 3 is 5.
```

■**注意** 虽然看起来不显然，但是上面的输出中是有三个空行的——两个在文本上方，一个在下方。尽管前两个域已经被替换为空字符串，跟随的空行还是留在那里的。而且print语句增加了新行，也就是末尾的空行。

但是等等，它还能更好！因为使用了fileinput，我可以轮流处理几个文件。意味着我可以使用一个文件为变量定义值，其他的文件作为插入这些值的地方。比如有个类似列表10-13的文件，名字为magnus.txt，以及列表10-14中叫做template.txt的文件：

列表 10-13. *一些模板定义*

```
[name       = 'Magnus Lie Hetland'  ]
[email      = 'magnus@foo.bar'      ]
[language = 'python'                ]
```

Listing 10-14. *一个模板*

```
[import time]
Dear [name],

I would like  to learn how to program. I hear you use the
[language] language a lot --  is it something I should
consider?

And, by the way, is [email] your  correct email address?


Fooville, [time.asctime()]

Oscar Frozzbozz
```

import time并不是赋值语句（我设定不处理的语句类型），但是因为我还不是过分吹毛求疵的人，所以只用了try/except语句，使得程序支持任何可以用eval和exec语句和表达式，你可以像这样运行程序（在UNIX命令行下）：

```
$ python templates.py magnus.txt  template.txt
```

你会获得类似于列表10-15的输出：

列表10-15 *模板系统的示例输出*

```
Dear Magnus Lie Hetland,

I would like    to learn how to program. I hear you use the
python language a lot -- is it something I
should consider?

And, by the way, is magnus@foo.bar your correct email address?


Fooville, Wed Apr 24 20:34:29 2004

Oscar Frozzbozz
```

尽管这个模板系统可以进行功能非常强大的替换，它还是有些瑕疵的。比如如果你能用更灵活的方式书写定义文件会更好。如果使用`execfile`执行文件，你可以使用正常的`Python`语句。这样也会解决输出中顶部出现空行的问题。

你能想到改进的方法吗？你能想到其他的可以用在程序中的概念吗？（在我看来）精通任何程序设计语言的方法是实践——尝试它的极限，探索它的威力。看看你能不能重写这个程序让它工作的更好并且满足你的需要。

■**注意** 事实上，在标准库的`string`模块中已经有了一个完美的模板系统。你可以试试`Template`类。

其他有趣的模块
Other Interesting Standard Modules

尽管这章里面已经包括了很多模块，但这知识冰山一角。为了让你能更快的深入，我在下面列出了一些很酷的库：

difflib。这个库让你可以计算两个序列的相似程度。还能让你从一些序列（从一列表的可能性中）找出和你提供的原始序列“最像”的。difflib可以用于创建简单的搜索程序。

md5和sha。这两个模块可以从字符串计算小“签名”（数值）。如果你从两个不同的字符串计算出了签名，就可以确保这两个签名完全不同。你可以将其应用于大文本文件。这两个模块在加密和安全性方面有很多用途

csv。CSV是Comma-Separated Values（逗号分隔值）的简写，这是一种很多程序（比如很多表格和数据库程序）都可以用来制表格式。它主要用于在不同程序间交换数据。csv模块让你可以轻松读写CSV文件，显然，它在格式化方面有很多精巧的部分。

timeit、profile和trace。timeit模块（以及它的命令行脚本）是测量代码运行时间的工具。它可有很多功能，你可以用它来代替time模块进行性能测试。profile模块（和伴随模块pstats）可以用于代码效率的全面分析。trace模块

（和程序）可以提供给你覆盖性的分析（也就是你的代码哪部分执行了，哪部分没执行）。当你在写测试代码的时候很有用。

datetime。如果time模块在时间时间追踪需要方面还不够，那么datetime可能会够用。它支持特殊的日期和事件对象，允许你用很多方法对它们建立和联合。它的接口在很多方面比time的要更加适合。

itertools。它用来创建和联合迭代器（或者其他迭代对象），还包括迭代链接、创建无限返回连续整数（和range类似，但是没有上线）的迭代器，以及重复轮轮询代对象的函数等。

logging。打印出你的程序在做什么的语句。如果你想进行跟踪但是不想打印debug输出，你可能会想到把信息写入日志文件。这个模块提供给你管理一个或者多个日志、多优先层次日志信息和其它一些功能。

getopt和optparse。在UNIX内，命令行程序经常使用不同的选项（option）或者开关（switches）运行（Python解释器就是个典型的例子）。这些东西都可以在sys.argv中找到，但是自己正确处理它们远没有这么简单。getopt库试图解决这个问题，optparse则更新、更强大和更易用。

cmd。这个模块可以让你书写命令行解释器，就像Python的交互式解释器一样。你可以定义用户能在提示符后执行的命令。可能你会想用这个创建你程序的用户界面？

快速总结
A Quick Summary

本章内，你学习了模块的知识：如何创建、如何探索以及如何使用标准Python库中的模块。

模块。模块基本说来是主函数用于**定义事情**的子程序，这些事情包括函数、类和变量。如果模块包含测试代码，那么应该将这部分代码放置在检查__name__=='__man__'是否为真的if语句中。PYTHONPATH中的模块可以导入。你可以用语句import foo导入存储在foo.py文件中的模块。

包。包是包含有其他模块的模块。包使用包含有__init__.py文件的目录实现。

探索模块。在你在交互式编辑器导入模块后，你可以用很多方法进行探索。比如dir、检查__all__变量以及使用help函数。文档和源代码和信息和内部机制的极好来源。

标准库。Python包括了一些模块，叫做标准库。本章内提到的包括：

- sys：让你访问和Python解释器连接紧密的一些变量和函数的模块。
- os：让你可以访问和操作系统连接紧密的一些变量和函数的模块。
- fileinput：让你轻松迭代数个文件和流中的行的模块。
- sets、heapq和deque：这三个模块提供你三个有用的数据结构。集合也以内建的类型set存在。
- time：获取当前时间，并且进行时间日期操作和格式化的模块。
- random：产生随机数、从序列中选取随机元素以及打乱列表元素的模块。
- shelve：创建将持续性映射内容保存在给定文件名的数据库中的模块。

如果你想要了解更多模块，我再次建议你浏览Python库参（http://python.org/doc/lib），读起来很有意思。

本章内的新函数
New Functions in This Chapter

函数	描述
dir(obj)	返回以字母排序的特性名称的列表
help([obj])	提供关于特定对象的互动帮助信息
reload(module)	返回已经导入模块的重载入版本

现在学什么？
What Now?

如果你多少从本章中汲取了一些概念，你的Python功力也会大涨。使用你你手头的标准库可以让Python从很强大变为无比强大。以你到目前学习的知识，已经能写很大范围的程序了。下一张中，你将会学习如何用Python和外部世界——文件以及网络——进行交互，应付范围扩大后的问题。