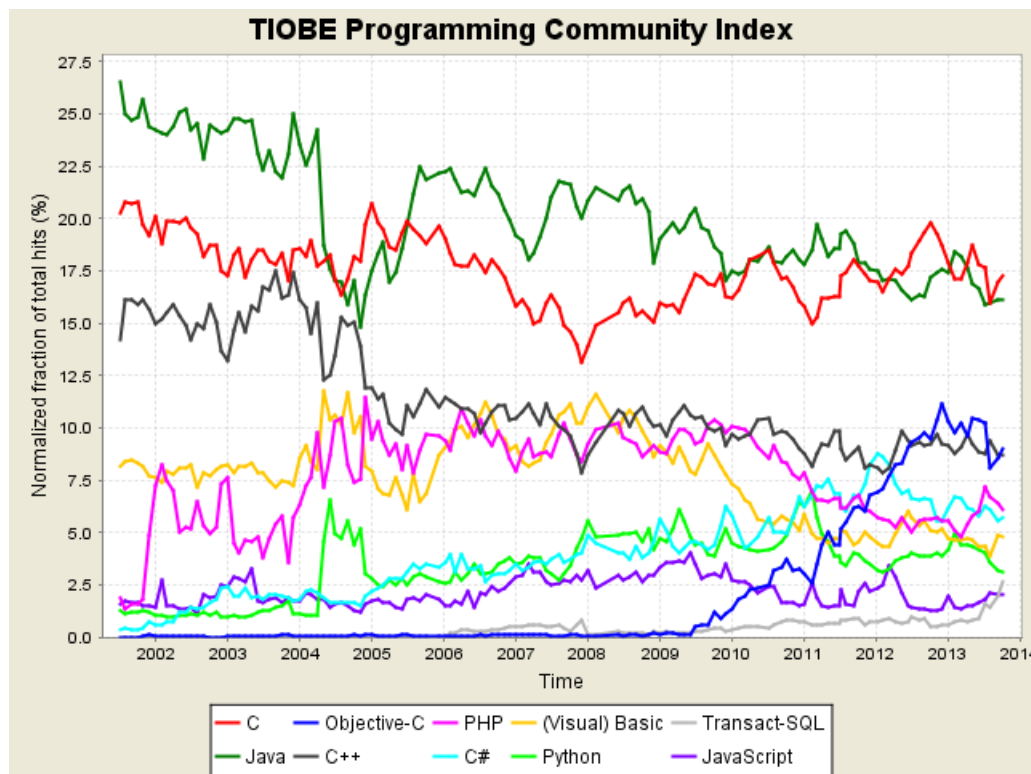


Python是著名的“龟叔”Guido van Rossum在1989年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。

现在，全世界差不多有600多种编程语言，但流行的编程语言也就那么20来种。如果你听说过TIOBE排行榜，你就能知道编程语言的大致流行程度。这是最近10年最常用的10种编程语言的变化图：



总的来说，这几种编程语言各有千秋。C语言是可以用来编写操作系统的贴近硬件的语言，所以，C语言适合开发那些追求运行速度、充分发挥硬件性能的程序。而Python是用来编写应用程序的高级编程语言。

当你用一种语言开始作真正的软件开发时，你除了编写代码外，还需要很多基本的已经写好的现成的东西，来帮助你加快开发进度。比如说，要编写一个电子邮件客户端，如果先从最底层开始编写网络协议相关的代码，那估计一年半载也开发不出来。高级编程语言通常都会提供一个比较完善的基础代码库，让你能直接调用，比如，针对电子邮件协议的SMTP库，针对桌面环境的GUI库，在这些已有的代码库的基础上开发，一个电子邮件客户端几天就能开发出来。

Python就为我们提供了非常完善的基础代码库，覆盖了网络、文件、GUI、数据库、文本等大量内容，被形象地称作“内置电池（batteries included）”。用Python开发，许多功能不必从零编写，直接使用现成的即可。

除了内置的库外，Python还有大量的第三方库，也就是别人开发的，供你直接使用的东西。当然，如果你开发的代码通过很好的封装，也可以作为第三方库给别人使用。

许多大型网站就是用Python开发的，例如YouTube、[Instagram](#)，还有国内的[豆瓣](#)。很多大公司，包括Google、Yahoo等，甚至NASA（美国航空航天局）都大量地使用Python。

龟叔给Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。

总的来说，Python的哲学就是简单优雅，尽量写容易看明白的代码，尽量写少的代码。如果一个资深程序员向你炫耀他写的晦涩难懂、动不动就几万行的代码，你可以尽情地嘲笑他。

那Python适合开发哪些类型的应用呢？

首选是网络应用，包括网站、后台服务等等；

其次是许多日常需要的小工具，包括系统管理员需要的脚本任务等等；

另外就是把其他语言开发的程序再包装起来，方便使用。

最后说说Python的缺点。

任何编程语言都有缺点，Python也不例外。优点说过了，那Python有哪些缺点呢？

第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。

但是大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载MP3的网络应用程序，C程序的运行时间需要0.001秒，而Python程序的运行时间需要0.1秒，慢了100倍，但由于网络更慢，需要等待1秒，你想，用户能感觉到1.001秒和1.1秒的区别吗？这就好比F1赛车和普通的出租车在北京三环路上行驶的道理一样，虽然F1赛车理论时速高达400公里，但由于三环堵车时的时速只有20公里，因此，作为乘客，你感觉的时速永远是20公里。



第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

这个缺点仅限于你要编写的软件需要卖给别人挣钱的时候。好消息是目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站和移动应用卖服务的模式越来越多了，后一种模式不需要把源码给别人。

再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux一样的开源代码，我们千万不要高估自己写的代码真的有非常大的“商业价值”。那些大公司的代码不愿意开放的更重要的原因是代码写得太烂了，一旦开源，就没人敢用他们的产品了。



当然，Python还有其他若干小缺点，请自行忽略，就不一一列举了。

因为Python是跨平台的，它可以运行在Windows、Mac和各种Linux/Unix系统上。在Windows上写Python程序，放到Linux上也是能够运行的。

要开始学习Python编程，首先就得把Python安装到你的电脑里。安装后，你会得到Python解释器（就是负责运行Python程序的），一个命令行交互环境，还有一个简单的集成开发环境。

2.x还是3.x

目前，Python有两个版本，一个是2.x版，一个是3.x版，这两个版本是不兼容的，因为现在Python正在朝着3.x版本进化，在进化过程中，大量的针对2.x版本的代码要修改后才能运行，所以，目前有许多第三方库还暂时无法在3.x上使用。

为了保证你的程序能用到大量的第三方库，我们的教程仍以2.x版本为基础，确切地说，是2.7版本。请确保你的电脑上安装的Python版本是2.7.x，这样，你才能无痛学习这个教程。

在Mac上安装Python

如果你正在使用Mac，系统是OS X 10.8或者最新的10.9 Mavericks，恭喜你，系统自带了Python 2.7。如果你的系统版本低于10.8，请自行备份系统并免费升级到最新的10.9，就可以获得Python 2.7。

查看系统版本的办法是点击左上角的苹果图标，选择“关于本机”：



在Linux上安装Python

如果你正在使用Linux，那我可以假定你有Linux系统管理经验，自行安装Python 2.7应该没有问题，否则，请换回Windows系统。

对于大量的目前仍在使用Windows的同学，如果短期内没有打算换Mac，就可以继续阅读以下内容。

在Windows上安装Python

首先，从Python的官方网站python.org下载最新的2.7版本，网速慢的同学请移步[国内镜像](#)。

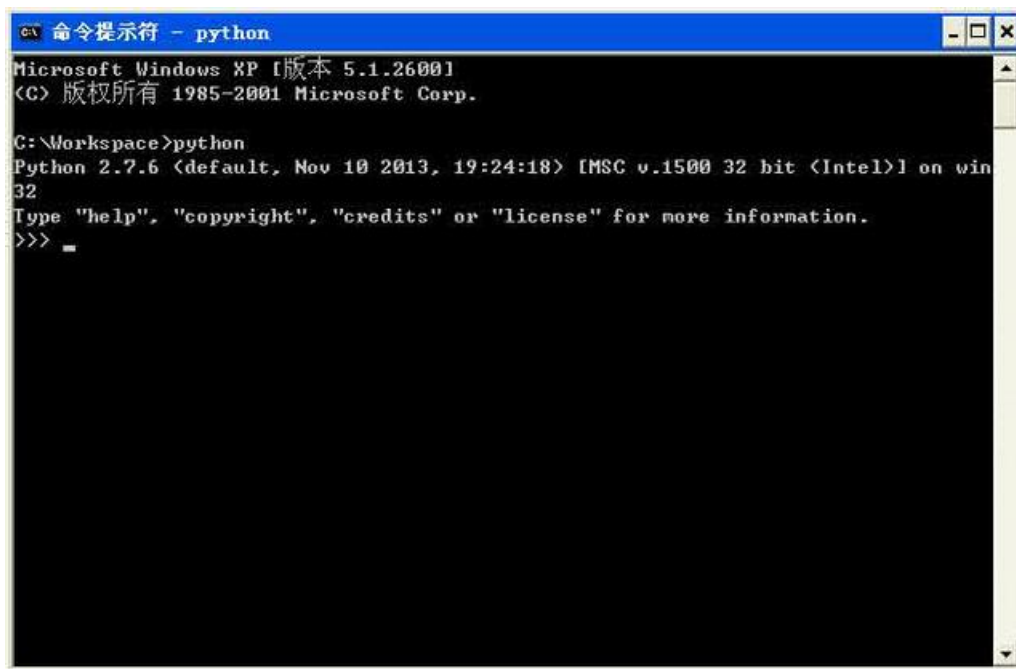
然后，运行下载的MSI安装包，在选择安装组件的一步时，勾上所有的组件：



特别要注意选上pip和Add python.exe to Path，然后一路点“Next”即可完成安装。

默认会安装到C:\Python27目录下，然后打开命令提示符窗口，敲入python后，会出现两种情况：

情况一：



看到上面的画面，就说明Python安装成功！

你看到提示符>>>就表示我们已经在Python交互式环境中了，可以输入任何Python代码，回车后会立刻得到执行结果。现在，输入exit()并回车，就可以退出Python交互式环境（直接关掉命令行窗口也可以！）。

情况二：得到一个错误：

'python'不是内部或外部命令，也不是可运行的程序或批处理文件。

这是因为Windows会根据一个Path的环境变量设定的路径去查找python.exe，如果没找到，就会报错。如果在安装时

漏掉了勾选Add python.exe to Path，那就要手动把python.exe所在的路径C:\Python27添加到Path中。

如果你不知道怎么修改环境变量，建议把Python安装程序重新运行一遍，记得勾上Add python.exe to Path。

小结

学会如何把Python安装到计算机中，并且熟练打开和退出Python交互式环境。

当我们编写Python代码时，我们得到的是一个包含Python代码的以.py为扩展名的文本文件。要运行代码，就需要Python解释器去执行.py文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

CPython

当我们从[Python官方网站](#)下载并安装好Python 2.7后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行python就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

IPython

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。

PyPy

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用[JIT技术](#)，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解[PyPy和CPython的不同点](#)。

Jython

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

IronPython

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

小结

Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

本教程的所有代码只确保在CPython 2.7版本下运行。请务必在本地安装CPython（也就是从Python官方网站下载的安装程序）。

此外，教程还内嵌一个IPython的Web版本，用来在浏览器内练习执行一些Python代码。要注意两者功能一样，输入的代码一样，但是提示符有所不同。另外，**不是所有代码都能在Web版本的IPython中执行**，出于安全原因，很多操作（比如文件操作）是受限的，所以有些代码必须在本地环境执行代码。

现在，了解了如何启动和退出Python的交互式环境，我们就可以正式开始编写Python代码了。

在写代码之前，请千万不要用“复制”-“粘贴”把代码从页面粘贴到你自己的电脑上。写程序也讲究一个感觉，你需要一个字母一个字母地把代码自己敲进去，在敲代码的过程中，初学者经常会敲错代码，所以，你需要仔细地检查、对照，才能以最快的速度掌握如何写程序。

在交互式环境的提示符>>>下，直接输入代码，按回车，就可以立刻得到代码执行结果。现在，试试输入100+200，看看计算结果是不是300：

```
>>> 100+200
300
```

很简单吧，任何有效的数学计算都可以算出来。

如果要让Python打印出指定的文字，可以用print语句，然后把希望打印的文字用单引号或者双引号括起来，但不能混用单引号和双引号：

```
>>> print 'hello, world'
hello, world
```

这种用单引号或者双引号括起来的文本在程序中叫字符串，今后我们还会经常遇到。

最后，用exit()退出Python，我们的第一个Python程序完成！唯一的缺憾是没有保存下来，下次运行时还要再输入一遍代码。

小结

在Python交互式命令行下，可以直接输入代码，然后执行，并立刻得到结果。

在Python的交互式命令行写程序，好处是一下就能得到结果，坏处是没法保存，下次还想运行的时候，还得再敲一遍。

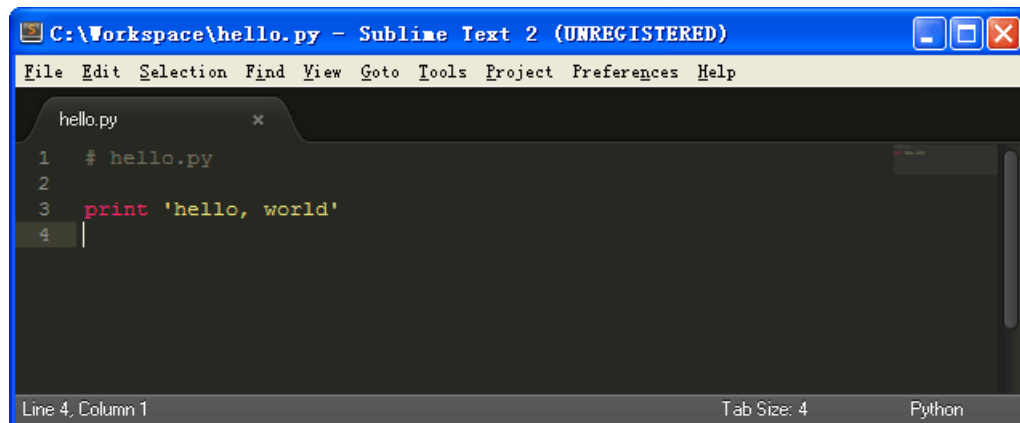
所以，实际开发的时候，我们总是使用一个文本编辑器来写代码，写完了，保存为一个文件，这样，程序就可以反复运行了。

现在，我们就把上次的'hello, world'程序用文本编辑器写出来，保存下来。

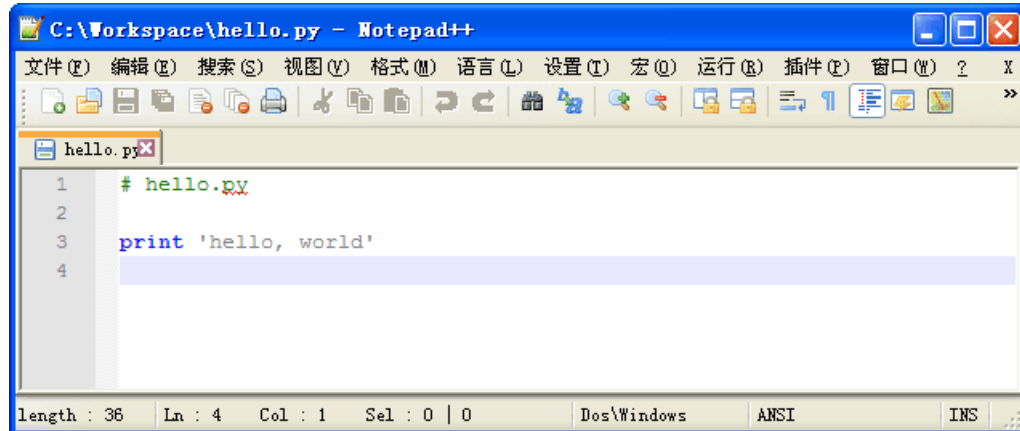
那么问题来了：文本编辑器到底哪家强？

推荐两款文本编辑器：

一个是[Sublime Text](#)，免费使用，但是不付费会弹出提示框：



一个是[Notepad++](#)，免费使用，有中文界面：



请注意，用哪个都行，但是**绝对不能用Word和Windows自带的记事本**。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

安装好文本编辑器后，输入以下代码：

```
print 'hello, world'
```

注意print前面不要有任何空格。然后，选择一个目录，例如C:\Workspace，把文件保存为hello.py，就可以打开命令行窗口，把当前目录切换到hello.py所在目录，就可以运行这个程序了：

```
C:\Workspace>python hello.py
hello, world
```

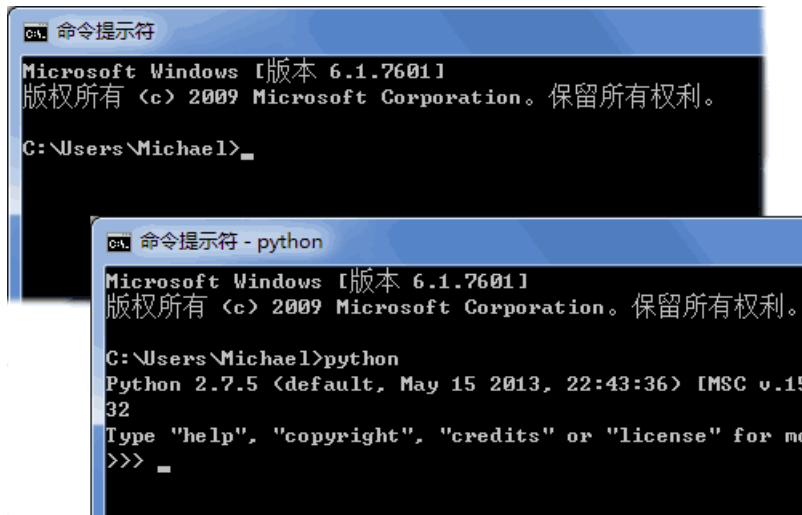
也可以保存为别的名字，比如abc.py，但是必须要以.py结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有hello.py这个文件，运行python hello.py就会报错：

```
python hello.py
python: can't open file 'hello.py': [Errno 2] No such file or directory
```

报错的意思就是，无法打开hello.py这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

请注意区分命令行模式和Python交互模式：



看到类似c:\>是在Windows提供的命令行模式，看到>>>是在Python交互式环境下。

在命令行模式下，可以执行python进入Python交互式环境，也可以执行python hello.py运行一个.py文件，但是在Python交互式环境下，只能输入Python代码执行。

直接运行py文件

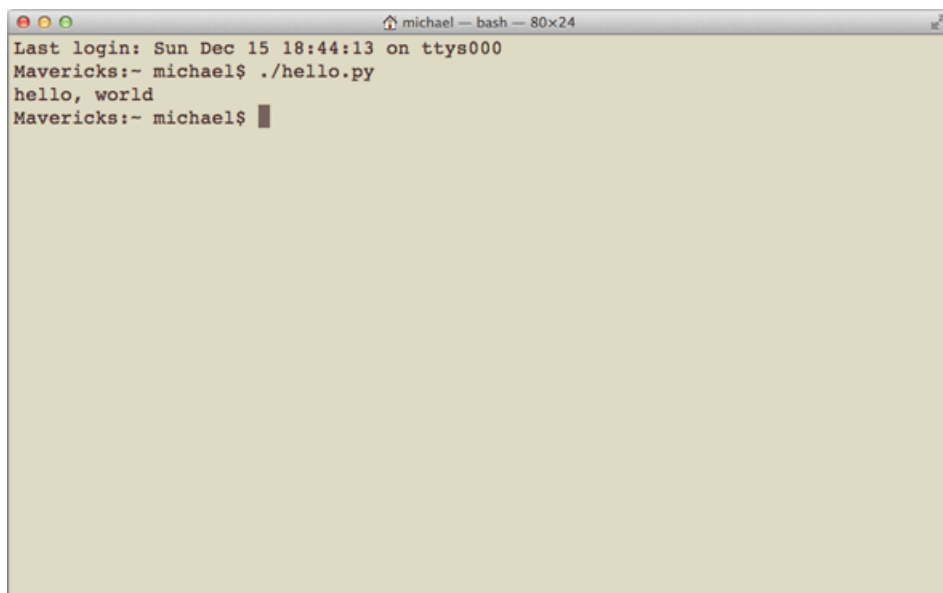
还有同学问，能不能像.exe文件那样直接运行.py文件呢？在Windows上是不行的，但是，在Mac和Linux上是可以的，方法是在.py文件的第一行加上：

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

然后，通过命令：

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

就可以直接运行hello.py了，比如在Mac下运行：

A terminal window titled 'michael — bash — 80x24'. The output shows a successful execution of a Python script. The text in the terminal is: 'Last login: Sun Dec 15 18:44:13 on ttys000', 'Mavericks:~ michael\$./hello.py', 'hello, world', and 'Mavericks:~ michael\$' followed by a cursor.

```
michael — bash — 80x24
Last login: Sun Dec 15 18:44:13 on ttys000
Mavericks:~ michael$ ./hello.py
hello, world
Mavericks:~ michael$
```

小结

用文本编辑器写Python程序，然后保存为后缀为.py的文件，就可以用Python直接运行这个程序了。

Python的交互模式和直接运行.py文件有什么区别呢？

直接输入python进入交互模式，相当于启动了Python解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行.py文件相当于启动了Python解释器，然后一次性把.py文件的源代码给执行了，你是没有机会输入源代码的。

用Python开发程序，完全可以一边在文本编辑器里写代码，一边开一个交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

输出

用`print`加上字符串，就可以向屏幕上输出指定的文字。比如输出'hello, world'，用代码实现如下：

```
>>> print 'hello, world'
```

`print`语句也可以跟上多个字符串，用逗号“,”隔开，就可以连成一串输出：

```
>>> print 'The quick brown fox', 'jumps over', 'the lazy dog'
The quick brown fox jumps over the lazy dog
```

`print`会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：

`print`也可以打印整数，或者计算结果：

```
>>> print 300
300
>>> print 100 + 200
300
```

因此，我们可以把计算`100 + 200`的结果打印得更漂亮一点：

```
>>> print '100 + 200 =', 100 + 200
100 + 200 = 300
```

注意，对于`100 + 200`，**Python**解释器自动计算出结果300，但是，`'100 + 200 ='`是字符串而非数学公式，**Python**把它视为字符串，请自行解释上述打印结果。

输入

现在，你已经可以用`print`输出你想要的结果了。但是，如果要从用户从电脑输入一些字符怎么办？**Python**提供了一个`raw_input`，可以让用户输入字符串，并存放到一个变量里。比如输入用户的名字：

```
>>> name = raw_input()
Michael
```

当你输入`name = raw_input()`并按下回车后，**Python**交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

输入完成后，不会有任何提示，**Python**交互式命令行又回到`>>>`状态了。那我们刚才输入的内容到哪去了？答案是存放到`name`变量里了。可以直接输入`name`查看变量内容：

```
>>> name
'Michael'
```

什么是变量？请回忆初中数学所学的代数基础知识：

设正方形的边长为`a`，则正方形的面积为`a x a`。把边长`a`看做一个变量，我们就可以根据`a`的值计算正方形的面积，比如：

若`a=2`，则面积为`a x a = 2 x 2 = 4`；

若`a=3.5`，则面积为`a x a = 3.5 x 3.5 = 12.25`。

在计算机程序中，变量不仅可以为整数或浮点数，还可以是字符串，因此，`name`作为一个变量就是一个字符串。

要打印出`name`变量的内容，除了直接写`name`然后按回车外，还可以用`print`语句：

```
>>> print name
Michael
```

有了输入和输出，我们就可以把上次打印'hello, world'的程序改成有点意义的程序了：

```
name = raw_input()
print 'hello,', name
```

运行上面的程序，第一行代码会让用户输入任意字符作为自己的名字，然后存入name变量中；第二行代码会根据用户的名字向用户说hello，比如输入Michael：

```
C:\Workspace> python hello.py
Michael
hello, Michael
```

但是程序运行的时候，没有任何提示信息告诉用户：“嘿，赶紧输入你的名字”，这样显得很不好。幸好，raw_input可以让你显示一个字符串来提示用户，于是我们把代码改成：

```
name = raw_input('please enter your name: ')
print 'hello,', name
```

再次运行这个程序，你会发现，程序一运行，会首先打印出please enter your name:，这样，用户就可以根据提示，输入名字后，得到hello, xxx的输出：

```
C:\Workspace> python hello.py
please enter your name: Michael
hello, Michael
```

每次运行该程序，根据用户输入的不同，输出结果也会不同。

在命令行下，输入和输出就是这么简单。

小结

任何计算机程序都是为了执行一个特定的任务，有了输入，用户才能告诉计算机程序所需的信息，有了输出，程序运行后才能告诉用户任务的结果。

输入是Input，输出是Output，因此，我们把输入输出统称为Input/Output，或者简写为IO。

raw_input和print是在命令行下面最基本的输入和输出，但是，用户也可以通过其他更高级的图形界面完成输入和输出，比如，在网页上的一个文本框输入自己的名字，点击“确定”后在网页上看到输出信息。

Python是一种计算机编程语言。计算机编程语言和我们日常使用的自然语言有所不同，最大的区别就是，自然语言在不同的语境下有不同的理解，而计算机要根据编程语言执行任务，就必须保证编程语言写出的程序决不能有歧义，所以，任何一种编程语言都有自己的一套语法，编译器或者解释器就是负责把符合语法的程序代码转换成CPU能够执行的机器码，然后执行。Python也不例外。

Python的语法比较简单，采用缩进方式，写出来的代码就像下面的样子：

```
# print absolute value of an integer:
a = 100
if a >= 0:
    print a
else:
    print -a
```

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号“:”结尾时，缩进的语句视为代码块。

缩进有利有弊。好处是强迫你写出格式化的代码，但没有规定缩进是几个空格还是Tab。按照约定俗成的管理，应该始终坚持使用4个空格的缩进。

缩进的另一个好处是强迫你写出缩进较少的代码，你会倾向于把一段很长的代码拆分成若干函数，从而得到缩进较少的代码。

缩进的坏处就是“复制—粘贴”功能失效了，这是最坑爹的地方。当你重构代码时，粘贴过去的代码必须重新检查缩进是否正确。此外，IDE很难像格式化Java代码那样格式化Python代码。

最后，请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在Python中，能够直接处理的数据类型有以下几种：

整数

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。

浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如， 1.23×10^9 和 12.3×10^8 是相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代， 1.23×10^9 就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。

整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

字符串

字符串是以"或'"括起来的任意文本，比如'abc'，"xyz"等等。请注意，"或'"本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。如果'本身也是一个字符，那就可以用'"括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

```
'I\'m \'OK\'!'
```

表示的字符串内容是：

```
I'm "OK"!
```

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print打印字符串看看：

```
>>> print 'I\'m ok.'
I'm ok.
>>> print 'I\'m learning\nPython.'
I'm learning
Python.
>>> print '\\n\\'
\
\
```

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

```
>>> print '\\t\\'
\
\
>>> print r'\\t\\'
\\t\\
```

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

```
>>> print '''line1
... line2
... line3'''
line1
line2
line3
```

上面是在交互式命令行内输入，如果写成程序，就是：

```
print '''line1
line2
line3'''
```

多行字符串'''...'''还可以在前面加上r使用，请自行测试。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False，在Python中，可以直接用True、False表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用and、or和not运算。

and运算是与运算，只有所有都为True，and运算结果才是True：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

or运算是或运算，只要其中有一个为True，or运算结果就是True：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
```

not运算是非运算，它是一个单目运算符，把True变成False，False变成True：

```
>>> not True
False
>>> not False
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:
    print 'adult'
else:
    print 'teenager'
```

空值

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在程序中就是用一个变量名表示了，变量名必须是大小写英文、数字和_的组合，且不能用数字开头，比如：

```
a = 1
```

变量a是一个整数。

```
t_007 = 'T007'
```

变量t_007是一个字符串。

```
Answer = True
```

变量Answer是一个布尔值True。

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

```
a = 123 # a是整数
print a
a = 'ABC' # a变为字符串
print a
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下（//表示注释）：

```
int a = 123; // a是整数类型变量
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
x = 10
x = x + 2
```

如果从数学上理解 $x = x + 2$ 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 $x + 2$ ，得到结果12，再赋给变量x。由于x之前的值是10，重新赋值后，x的值变成12。

最后，理解变量在计算机内存中的表示也非常重要。当我们写：

```
a = 'ABC'
```

时，Python解释器干了两件事情：

1. 在内存中创建了一个'ABC'的字符串；
2. 在内存中创建了一个名为a的变量，并把它指向'ABC'。

也可以把一个变量a赋值给另一个变量b，这个操作实际上是把变量b指向变量a所指向的数据，例如下面的代码：

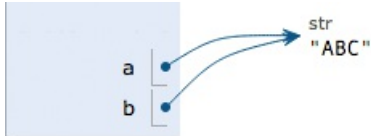
```
a = 'ABC'
b = a
a = 'XYZ'
print b
```

最后一行打印出变量b的内容到底是'ABC'呢还是'XYZ'？如果从数学意义上理解，就会错误地得出b和a相同，也应该是'XYZ'，但实际上b的值是'ABC'，让我们一行一行地执行代码，就可以看到到底发生了什么事：

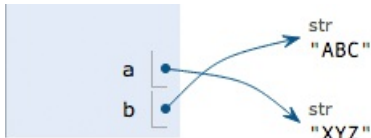
执行 `a = 'ABC'`，解释器创建了字符串 `'ABC'` 和变量 `a`，并把 `a` 指向 `'ABC'`：



执行 `b = a`，解释器创建了变量 `b`，并把 `b` 指向 `a` 指向的字符串 `'ABC'`：



执行 `a = 'XYZ'`，解释器创建了字符串 `'XYZ'`，并把 `a` 的指向改为 `'XYZ'`，但 `b` 并没有更改：



所以，最后打印变量 `b` 的结果自然是 `'ABC'` 了。

常量

所谓常量就是不能变的变量，比如常用的数学常数 π 就是一个常量。在 `Python` 中，通常用全部大写的变量名表示常量：

```
PI = 3.14159265359
```

但事实上 `PI` 仍然是一个变量，`Python` 根本没有任何机制保证 `PI` 不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量 `PI` 的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的，可以试试：

```
>>> 10 / 3
3
```

你没有看错，整数除法永远是整数，即使除不尽。要做精确的除法，只需把其中一个整数换成浮点数做除法就可以：

```
>>> 10.0 / 3
3.3333333333333335
```

因为整数除法只取结果的整数部分，所以 `Python` 还提供一个余数运算，可以得到两个整数相除的余数：

```
>>> 10 % 3
1
```

无论整数做除法还是取余数，结果永远是整数，所以，整数运算结果永远是精确的。

小结

`Python` 支持多种数据类型，在计算机内部，可以把任何数据都看成一个“对象”，而变量就是在程序中用来指向这些数据对象的，对变量赋值就是把数据和变量给关联起来。

字符编码

我们已经讲过了，字符串也是一种数据类型，但是，字符串比较特殊的是还有一个编码问题。

因为计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。

由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。

但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。

你可以想得到的是，全世界有上百种语言，日本把日文编到Shift_JIS里，韩国把韩文编到Euc-kr里，各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。



因此，Unicode应运而生。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。

Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。

现在，捋一捋ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

字母A用ASCII编码是十进制的65，二进制的01000001；

字符0用ASCII编码是十进制的48，二进制的00110000，注意字符'0'和整数0是不同的；

汉字中已经超出了ASCII编码的范围，用Unicode编码是十进制的20013，二进制的01001110 00101101。

你可以猜测，如果把ASCII编码的A用Unicode编码，只需要在前面补0就可以，因此，A的Unicode编码是00000000 01000001。

新的问题又出现了：如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的UTF-8编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

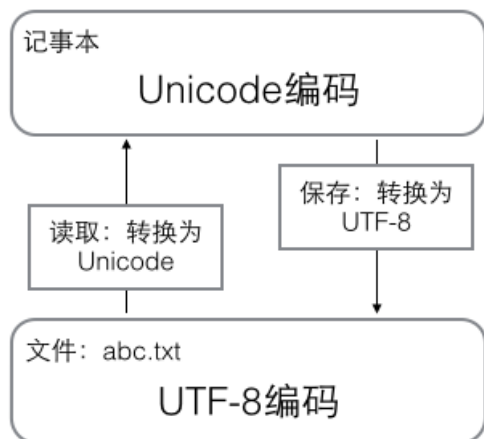
字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	x	01001110 00101101	11100100 10111000 10101101

从上面的表格还可以发现，UTF-8编码有一个额外的好处，就是ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在UTF-8编码下继续工作。

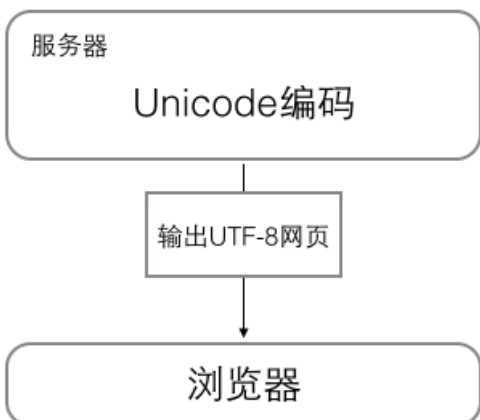
搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



所以你看很多网页的源码上会有类似<meta charset="UTF-8" />的信息，表示该网页正是用的UTF-8编码。

Python的字符串

搞清楚了令人头疼的字符编码问题后，我们再来研究Python对Unicode的支持。

因为Python的诞生比Unicode标准发布的时间还要早，所以最早的Python只支持ASCII编码，普通的字符串'ABC'在Python内部都是ASCII编码的。Python提供了ord()和chr()函数，可以把字母和对应的数字相互转换：

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Python在后来添加了对Unicode的支持，以Unicode表示的字符串用u'...'表示，比如：

```
>>> print u'中文'
中文
>>> u'中'
u'\u4e2d'
```

写u'中'和u'\u4e2d'是一样的，\u后面是十六进制的Unicode码。因此，u'A'和u'\u0041'也是一样的。

两种字符串如何相互转换？字符串'xxx'虽然是ASCII编码，但也可以看成是UTF-8编码，而u'xxx'则只能是Unicode编码。

把u'xxx'转换为UTF-8编码的'xxx'用encode('utf-8')方法：

```
>>> u'ABC'.encode('utf-8')
'ABC'
>>> u'中文'.encode('utf-8')
'\xe4\xb8\xad\xe6\x96\x87'
```

英文字符转换后表示的UTF-8的值和Unicode值相等（但占用的存储空间不同），而中文字符转换后1个Unicode字符将变为3个UTF-8字符，你看到的\xe4就是其中一个字节，因为它的值是228，没有对应的字母可以显示，所以以十六进制显示字节的数值。len()函数可以返回字符串的长度：

```
>>> len(u'ABC')
3
>>> len('ABC')
3
>>> len(u'中文')
2
>>> len('\xe4\xb8\xad\xe6\x96\x87')
6
```

反过来，把UTF-8编码表示的字符串'xxx'转换为Unicode字符串u'xxx'用decode('utf-8')方法：

```
>>> 'abc'.decode('utf-8')
u'abc'
>>> '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
u'\u4e2d\u6587'
>>> print '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
中文
```

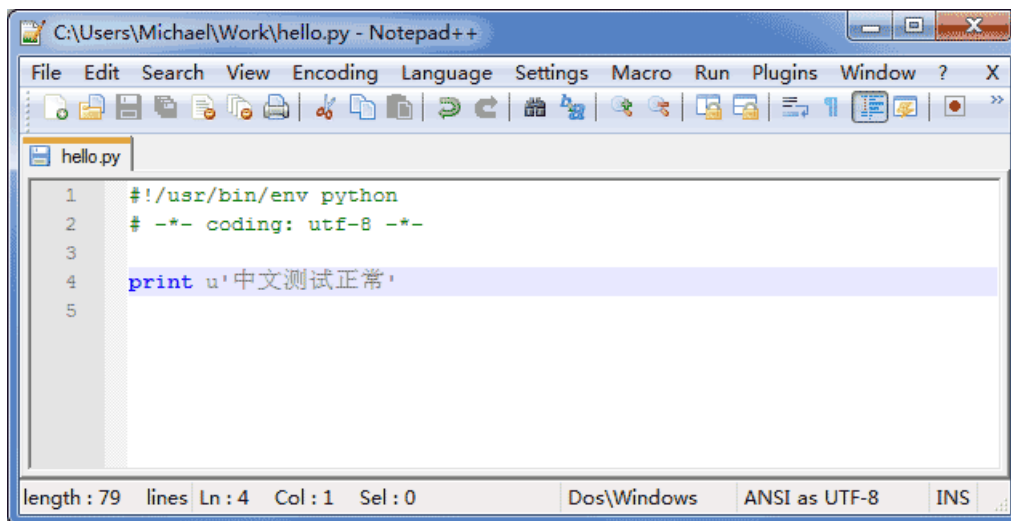
由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

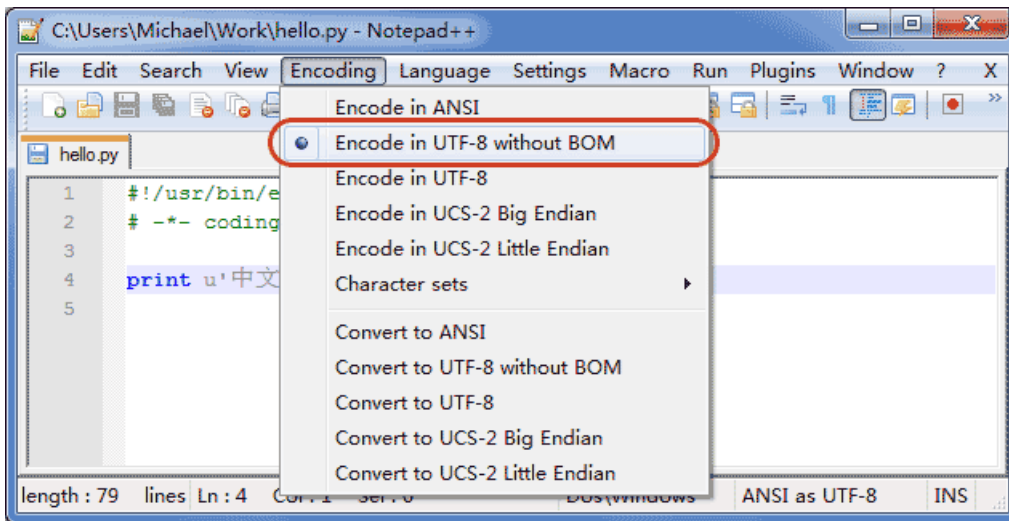
第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

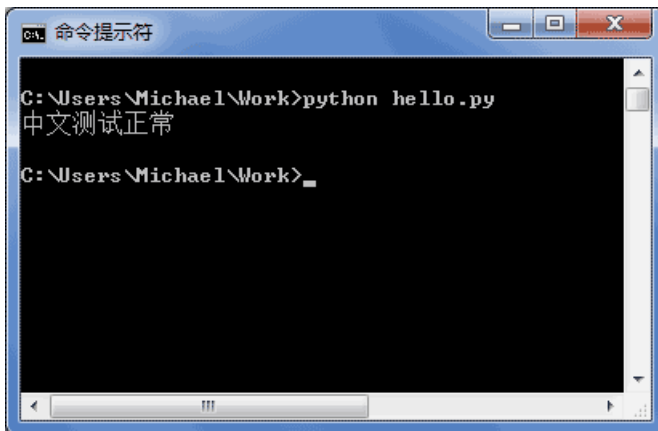
如果你使用Notepad++进行编辑，除了要加上# -*- coding: utf-8 -*-外，中文字符串必须是Unicode字符串：



声明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保Notepad++正在使用UTF-8 without BOM编码：

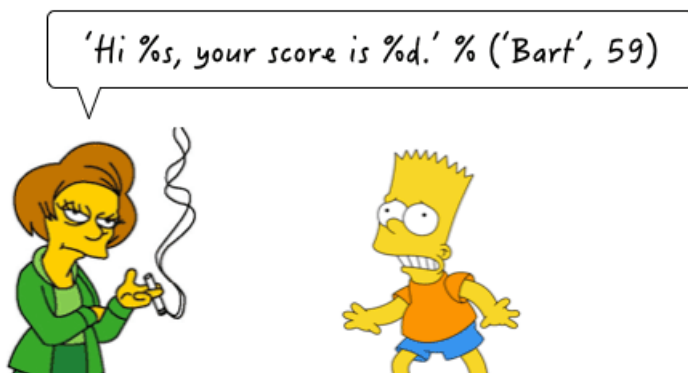


如果.py文件本身使用UTF-8编码，并且也声明了`# -*- coding: utf-8 -*-`，打开命令提示符测试就可以正常显示中文：



格式化

最后一个常见的问题是如何输出格式化的字符串。我们经常会输出类似'亲爱的xxx你好！你xx月的话费是xx，余额是xx'之类的字符串，而xxx的内容都是根据变量变化的，所以，需要一种简便的格式化字符串的方式。



在Python中，采用的格式化方式和C语言是一致的，用%实现，举例如下：

```
>>> 'Hello, %s' % 'world'
'Hello, world'
>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
'Hi, Michael, you have $1000000.'
```

你可能猜到了，%运算符就是用来格式化字符串的。在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?，括号可以省略。

常见的占位符有：

```
%d 整数
%f 浮点数
%s 字符串
%x 十六进制整数
```

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

```
>>> '%2d-%02d' % (3, 1)
' 3-01'
>>> '%.2f' % 3.1415926
'3.14'
```

如果你不太确定应该用什么，%s永远起作用，它会把任何数据类型转换为字符串：

```
>>> 'Age: %s. Gender: %s' % (25, True)
'Age: 25. Gender: True'
```

对于Unicode字符串，用法完全一样，但最好确保替换的字符串也是Unicode字符串：

```
>>> u'Hi, %s' % u'Michael'
u'Hi, Michael'
```

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

```
>>> 'growth rate: %d %%' % 7
'growth rate: 7 %'
```

小结

由于历史遗留问题，Python 2.x版本虽然支持Unicode，但在语法上需要'xxx'和u'xxx'两种字符串表示方式。

Python当然也支持其他编码方式，比如把Unicode编码成GB2312：

```
>>> u'中文'.encode('gb2312')
'\xd6\xd0\xce\xca'
```

但这种方式纯属自找麻烦，如果没有特殊业务要求，请牢记仅使用Unicode和UTF-8这两种编码方式。

在Python 3.x版本中，把'xxx'和u'xxx'统一成Unicode编码，即写不写前缀u都是一样的，而以字节形式表示的字符串则必须加上b前缀：b'xxx'。

格式化字符串的时候，可以用Python的交互式命令行测试，方便快捷。

list

Python内置的一种数据类型是列表：**list**。**list**是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个**list**表示：

```
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']
```

变量classmates就是一个**list**。用len()函数可以获得**list**元素的个数：

```
>>> len(classmates)
3
```

用索引来访问**list**中每一个位置的元素，记得索引是从0开始的：

```
>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是len(classmates) - 1。

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

```
>>> classmates[-1]
'Tracy'
```

以此类推，可以获取倒数第2个、倒数第3个：

```
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

当然，倒数第4个就越界了。

list是一个可变的有序表，所以，可以往**list**中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引号为1的位置：

```
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除**list**末尾的元素，用pop()方法：

```
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

要删除指定位置的元素，用`pop(i)`方法，其中`i`是索引位置：

```
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

```
>>> classmates[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

`list`里面的元素的数据类型也可以不同，比如：

```
>>> L = ['Apple', 123, True]
```

`list`元素也可以是另一个`list`，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
>>> len(s)
4
```

要注意`s`只有4个元素，其中`s[2]`又是一个`list`，如果拆开写就更容易理解了：

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
```

要拿到`'php'`可以写`p[1]`或者`s[2][1]`，因此`s`可以看成是一个二维数组，类似的还有三维、四维.....数组，不过很少用到。

如果一个`list`中一个元素也没有，就是一个空的`list`，它的长度为0：

```
>>> L = []
>>> len(L)
0
```

tuple

另一种有序列表叫元组：**tuple**。**tuple**和**list**非常类似，但是**tuple**一旦初始化就不能修改，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在，`classmates`这个**tuple**不能变了，它也没有`append()`，`insert()`这样的方法。其他获取元素的方法和**list**是一样的，你可以正常地使用`classmates[0]`，`classmates[-1]`，但不能赋值成另外的元素。

不可变的**tuple**有什么意义？因为**tuple**不可变，所以代码更安全。如果可能，能用**tuple**代替**list**就尽量用**tuple**。

tuple的陷阱：当你定义一个**tuple**时，在定义的时候，**tuple**的元素就必须被确定下来，比如：

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的**tuple**，可以写成`()`：

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有1个元素的**tuple**，如果你这么定义：

```
>>> t = (1)
>>> t
1
```


定义的不是`tuple`，是1这个数！这是因为括号`()`既可以表示`tuple`，又可以表示数学公式中的小括号，这就产生了歧义，因此，`Python`规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的`tuple`定义时必须加一个逗号`,`，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

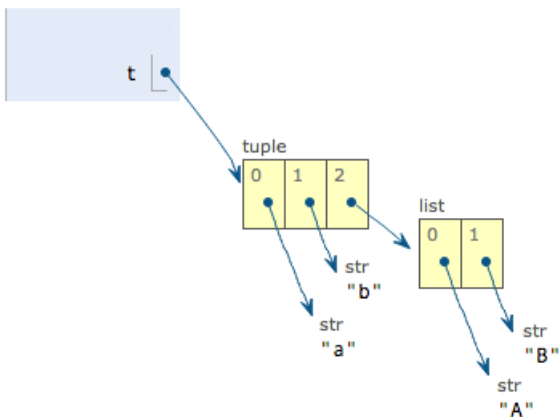
`Python`在显示只有1个元素的`tuple`时，也会加一个逗号`,`，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”`tuple`：

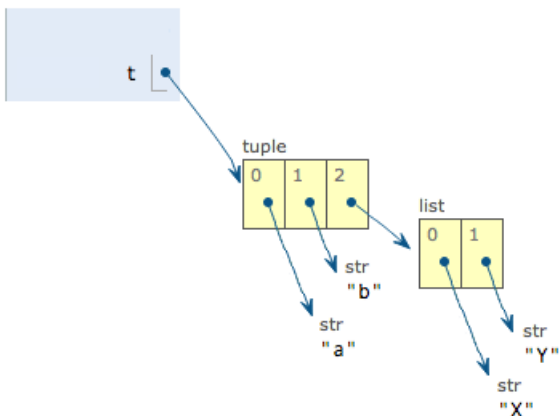
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

这个`tuple`定义的时候有3个元素，分别是`'a'`、`'b'`和一个`list`。不是说`tuple`一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候`tuple`包含的3个元素：



当我们把`list`的元素`'A'`和`'B'`修改为`'X'`和`'Y'`后，`tuple`变为：



表面上看，`tuple`的元素确实变了，但其实变的不是`tuple`的元素，而是`list`的元素。`tuple`一开始指向的`list`并没有改成别的`list`，所以，`tuple`所谓的“不变”是说，`tuple`的每个元素，指向永远不变。即指向`'a'`，就不能改成指向`'b'`，指向一个`list`，就不能改成指向其他对象，但指向的这个`list`本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的`tuple`怎么做？那就必须保证`tuple`的每一个元素本身也不能变。

小结

`list`和`tuple`是Python内置的有序集合，一个可变，一个不可变。根据需要来选择使用它们。

条件判断

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用if语句实现：

```
age = 20
if age >= 18:
    print 'your age is', age
    print 'adult'
```

根据Python的缩进规则，如果if语句判断是True，就把缩进的两行print语句执行了，否则，什么也不做。

也可以给if添加一个else语句，意思是，如果if判断是False，不要执行if的内容，去把else执行了：

```
age = 3
if age >= 18:
    print 'your age is', age
    print 'adult'
else:
    print 'your age is', age
    print 'teenager'
```

注意不要少写了冒号:。

当然上面的判断是很粗略的，完全可以用elif做更细致的判断：

```
age = 3
if age >= 18:
    print 'adult'
elif age >= 6:
    print 'teenager'
else:
    print 'kid'
```

elif是else if的缩写，完全可以有多个elif，所以if语句的完整形式就是：

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

if语句执行有个特点，它是从上往下判断，如果在某个判断上是True，把该判断对应的语句执行后，就忽略掉剩下的elif和else，所以，请测试并解释为什么下面的程序打印的是teenager：

```
age = 20
if age >= 6:
    print 'teenager'
elif age >= 18:
    print 'adult'
else:
    print 'kid'
```

if判断条件还可以简写，比如写：

```
if x:
    print 'True'
```

只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

循环

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print name
```

执行这段代码，会依次打印names的每一个元素：

```
Michael
Bob
Tracy
```

所以for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个sum变量做累加：

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print sum
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range()函数，可以生成一个整数序列，比如range(5)生成的序列是从0开始小于5的整数：

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

range(101)就可以生成0-100的整数序列，计算如下：

```
sum = 0
for x in range(101):
    sum = sum + x
print sum
```

请自行运行上述代码，看看结果是不是当年高斯同学心算出的5050。

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print sum
```

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

再议raw_input

最后看一个有问题的条件判断。很多同学会用raw_input()读取用户的输入，这样可以自己输入，程序运行得更有意思：

```
birth = raw_input('birth: ')
if birth < 2000:
    print '00前'
else:
    print '00后'
```

输入1982，结果却显示00后，这么简单的判断Python也能搞错？

当然不是Python的问题，在Python的交互式命令行下打印birth看看：

```
>>> birth
'1982'
>>> '1982' < 2000
False
>>> 1982 < 2000
True
```

原因找到了！原来从`raw_input()`读取的内容永远以字符串的形式返回，把字符串和整数比较就不会得到期待的结果，必须先用`int()`把字符串转换为我们想要的整型：

```
birth = int(raw_input('birth: '))
```

再次运行，就可以得到正确地结果。但是，如果输入`abc`呢？又会得到一个错误信息：

```
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'abc'
```

原来`int()`发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的[错误和调试](#)会讲到。

小结

条件判断可以让计算机自己做选择，Python的`if...elif...else`很灵活。

```
if salary >= 10000:
```

```
    print 
```

```
elif salary >= 5000:
```

```
    print 
```

```
else:
```

```
    print 
```

循环是让计算机做重复任务的有效的办法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用`Ctrl+C`退出程序，或者强制结束Python进程。

请试写一个死循环程序。

dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。

如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个dict如下：

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

为什么dict查找速度这么快？因为dict的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在list中查找元素的方法，list越大，查找越慢。

第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字，无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。

dict就是第二种实现方式，给定一个名字，比如'Michael'，dict在内部就可以直接计算出Michael对应的存放成绩的“页码”，也就是95这个数字存放的内存地址，直接取出来，所以速度非常快。

你可以猜到，这种key-value存储方式，在放进去的时候，必须根据key算出value的存放位置，这样，取的时候才能根据key直接拿到value。

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

```
>>> d['Adam'] = 67
>>> d['Adam']
67
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果key不存在，dict就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

要避免key不存在的错误，有两种办法，一是通过in判断key是否存在：

```
>>> 'Thomas' in d
False
```

二是通过dict提供的get方法，如果key不存在，可以返回None，或者自己指定的value：

```
>>> d.get('Thomas')
>>> d.get('Thomas', -1)
-1
```

注意：返回None的时候Python的交互式命令行不显示结果。

要删除一个key，用pop(key)方法，对应的value也会从dict中删除：

```
>>> d.pop('Bob')
75
>>> d
{'Michael': 95, 'Tracy': 85}
```

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而增加；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是不可变对象。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

```
>>> key = [1, 2, 3]
>>> d[key] = 'a list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

```
>>> s = set([1, 2, 3])
>>> s
set([1, 2, 3])
```

注意，传入的参数[1, 2, 3]是一个list，而显示的set([1, 2, 3])只是告诉你这个set内部有1, 2, 3这3个元素，显示的[]不表示这是一个list。

重复元素在set中自动被过滤：

```
>>> s = set([1, 1, 2, 2, 3, 3])
>>> s
set([1, 2, 3])
```

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果：

```
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
```

通过`remove(key)`方法可以删除元素：

```
>>> s.remove(4)
>>> s
set([1, 2, 3])
```

`set`可以看成数学意义上的无序和无重复元素的集合，因此，两个`set`可以做数学意义上的交集、并集等操作：

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
set([2, 3])
>>> s1 | s2
set([1, 2, 3, 4])
```

`set`和`dict`的唯一区别仅在于没有存储对应的`value`，但是，`set`的原理和`dict`一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证`set`内部“不会有重复元素”。试试把`list`放入`set`，看看是否会报错。

再议不可变对象

上面我们讲了，`str`是不变对象，而`list`是可变对象。

对于可变对象，比如`list`，对`list`进行操作，`list`内部的内容是会变化的，比如：

```
>>> a = ['c', 'b', 'a']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

而对于不可变对象，比如`str`，对`str`进行操作呢：

```
>>> a = 'abc'
>>> a.replace('a', 'A')
'Abc'
>>> a
'abc'
```

虽然字符串有个`replace()`方法，也确实变出了`'Abc'`，但变量`a`最后仍是`'abc'`，应该怎么理解呢？

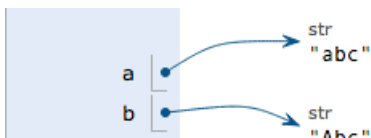
我们先把代码改成下面这样：

```
>>> a = 'abc'
>>> b = a.replace('a', 'A')
>>> b
'Abc'
>>> a
'abc'
```

要始终牢记的是，`a`是变量，而`'abc'`才是字符串对象！有些时候，我们经常说，对象`a`的内容是`'abc'`，但其实是指，`a`本身是一个变量，它指向的对象的内容才是`'abc'`：



当我们调用`a.replace('a', 'A')`时，实际上调用方法`replace`是作用在字符串对象`'abc'`上的，而这个方法虽然名字叫`replace`，但却没有改变字符串`'abc'`的内容。相反，`replace`方法创建了一个新字符串`'Abc'`并返回，如果我们用变量`b`指向该新字符串，就容易理解了，变量`a`仍指向原有的字符串`'abc'`，但变量`b`却指向新字符串`'Abc'`了：



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

小结

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

tuple虽然是不变对象，但试试把(1, 2, 3)和(1, [2, 3])放入dict或set中，并解释结果。

我们知道圆的面积计算公式为：

$$S = \pi r^2$$

当我们知道半径 r 的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 $3.14 * x * x$ 不仅很麻烦，而且，如果要把3.14改成3.14159265359的时候，得全部替换。

有了函数，我们就不再每次写 $s = 3.14 * x * x$ ，而是写成更有意义的函数调用 $s = \text{area_of_circle}(x)$ ，而函数 area_of_circle 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如： $1 + 2 + 3 + \dots + 100$ ，写起来十分不方便，于是数学家发明了求和符号 \sum ，可以把 $1 + 2 + 3 + \dots + 100$ 记作：

100

$$\sum_n$$

$n=1$

这种抽象记法非常强大，因为我们看到 \sum 就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

100

$$\sum (n^2+1)$$

$n=1$

还原成加法运算就变成了：

$$(1 \times 1 + 1) + (2 \times 2 + 1) + (3 \times 3 + 1) + \dots + (100 \times 100 + 1)$$

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数abs，只有一个参数。可以直接从Python的官方网站查看文档：

<http://docs.python.org/2/library/functions.html#abs>

也可以在交互式命令行通过help(abs)查看abs函数的帮助信息。

调用abs函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

调用函数的时候，如果传入的参数数量不对，会报TypeError的错误，并且Python会明确地告诉你：abs()有且仅有1个参数，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报TypeError的错误，并且给出错误信息：str是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而比较函数cmp(x, y)就需要两个参数，如果x<y，返回-1，如果x==y，返回0，如果x>y，返回1：

```
>>> cmp(1, 2)
-1
>>> cmp(2, 1)
1
>>> cmp(3, 3)
0
```

数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如int()函数可以把其他数据类型转换为整数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> unicode(100)
u'100'
>>> bool(1)
True
>>> bool('')
False
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量a指向abs函数
>>> a(-1) # 所以也可以通过a调用abs函数
1
```

小结

调用Python的函数，需要根据函数定义，传入正确的参数。如果函数调用出错，一定要学会看错误信息，所以英文很重要！

在Python中，定义一个函数要使用def语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用return语句返回。

我们以自定义一个求绝对值的my_abs函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

请自行测试并调用my_abs看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有return语句，函数执行完毕后也会返回结果，只是结果为None。

return None可以简写为return。

空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

```
def nop():
    pass
```

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

pass还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了pass，代码运行就会有语法错误。

参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出TypeError：

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes exactly 1 argument (2 given)
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试my_abs和内置函数abs的差别：

```
>>> my_abs('A')
'A'
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而我们定义的my_abs没有参数检查，所以，这个函数定义不够完善。

让我们修改一下my_abs的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数isinstance实现：

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
```

```
        return x
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续讲到。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的新的坐标：

```
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
```

这样我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print x, y
151.961524227 70.0
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print r
(151.96152422706632, 70.0)
```

原来返回值是一个tuple！但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

小结

定义函数时，需要确定函数名和参数个数；

如果有必要，可以先对参数的数据类型做检查；

函数体内部可以用return随时返回函数结果；

函数执行完毕也没有return语句时，自动return None。

函数可以同时返回多个值，但其实就是一个tuple。

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

默认参数

我们仍以具体的例子来说明如何定义函数的默认参数。先写一个计算 x^2 的函数：

```
def power(x):  
    return x * x
```

当我们调用power函数时，必须传入有且仅有的一个参数x：

```
>>> power(5)  
25  
>>> power(15)  
225
```

现在，如果我们要计算 x^3 怎么办？可以再定义一个power3函数，但是如果我们要计算 x^4 、 x^5怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把power(x)修改为power(x, n)，用来计算 x^n ，说干就干：

```
def power(x, n):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

对于这个修改后的power函数，可以计算任意n次方：

```
>>> power(5, 2)  
25  
>>> power(5, 3)  
125
```

但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码无法正常调用：

```
>>> power(5)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: power() takes exactly 2 arguments (1 given)
```

这个时候，默认参数就排上用场了。由于我们经常计算 x^2 ，所以，完全可以把第二个参数n的默认值设定为2：

```
def power(x, n=2):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

这样，当我们调用power(5)时，相当于调用power(5, 2)：

```
>>> power(5)  
25  
>>> power(5, 2)  
25
```

而对于 $n > 2$ 的其他情况，就必须明确地传入n，比如power(5, 3)。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入name和gender两个参数：

```
def enroll(name, gender):  
    print 'name:', name  
    print 'gender:', gender
```

这样，调用enroll()函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')  
name: Sarah  
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):  
    print 'name:', name  
    print 'gender:', gender  
    print 'age:', age  
    print 'city:', city
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')  
Student:  
name: Sarah  
gender: F  
age: 6  
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)  
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用enroll('Bob', 'M', 7)，意思是，除了name, gender这两个参数外，最后1个参数应用在参数age上，city参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用enroll('Adam', 'M', city='Tianjin')，意思是，city参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个END再返回：

```
def add_end(L=[]):  
    L.append('END')  
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])  
[1, 2, 3, 'END']
```



```
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用`add_end()`时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是`[]`，但是函数似乎每次都“记住了”上次添加了`'END'`后的`list`。

原因解释如下：

Python函数在定义的时候，默认参数`L`的值就被计算出来了，即`[]`，因为默认参数`L`也是一个变量，它指向对象`[]`，每次调用该函数，如果改变了`L`的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的`[]`了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用`None`这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计`str`、`None`这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例，给定一组数字`a, b, c, ……`，请计算 $a^2 + b^2 + c^2 + ……$ 。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把`a, b, c, ……`作为一个`list`或`tuple`传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个`list`或`tuple`：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义list或tuple参数相比，仅仅在参数前面加了一个*号。在函数内部，参数numbers接收到的的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个*号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

这种写法相当有用，而且很常见。

关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

```
def person(name, age, **kw):
    print 'name:', name, 'age:', age, 'other:', kw
```

函数person除了必选参数name和age外，还接受关键字参数kw。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=kw['city'], job=kw['job'])
```

```
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> kw = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **kw)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这4种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。

比如定义一个函数，包含上述4种参数：

```
def func(a, b, c=0, *args, **kw):
    print 'a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> func(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> func(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> func(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> func(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

最神奇的是通过一个tuple和dict，你也可以调用该函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'x': 99}
>>> func(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'x': 99}
```

所以，对于任意函数，都可以通过类似func(*args, **kw)的形式调用它，无论它的参数是如何定义的。

小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

*args是可变参数，args接收的是一个tuple；

**kw是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过*args传入：func(*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过**kw传入：func(**{'a': 1, 'b': 2})。

使用*args和**kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 $n! = 1 \times 2 \times 3 \times \dots \times n$ ，用函数fact(n)表示，可以看出：

$$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n = (n-1)! \times n = \text{fact}(n-1) \times n$$

所以, $\text{fact}(n)$ 可以表示为 $n \times \text{fact}(n-1)$, 只有 $n=1$ 时需要特殊处理。

二是, $\text{fact}(n)$ 用递归的方式写出来就是:

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>> fact(1)
>> fact(5)
20
>> fact(100)
3326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000L
```

如果我们计算 `fact(5)`，可以根据函数定义看到计算过程如下：

```
==> fact(5)
==> 5 * fact(4)
==> 5 * (4 * fact(3))
==> 5 * (4 * (3 * fact(2)))
==> 5 * (4 * (3 * (2 * fact(1))))
==> 5 * (4 * (3 * (2 * 1)))
==> 5 * (4 * (3 * 2))
==> 5 * (4 * 6)
==> 5 * 24
==> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试fact(1000)：

```
>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded
```

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，`return`语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的fact(n)函数由于return n * fact(n - 1)引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到, `return fact_iter(num - 1, num * product)` 仅返回递归函数本身, `num - 1` 和 `num * product` 在函数调用前就会被计算, 不影响函数调用。

fact(5)对应的fact_iter(5, 1)的调用如下:

```
==> fact_iter(5, 1)
==> fact_iter(4, 5)
==> fact_iter(3, 20)
==> fact_iter(2, 60)
==> fact_iter(1, 120)
==> 120
```

递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的fact(n)函数改成尾递归方式，也会导致栈溢出。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

掌握了Python的数据类型、语句和函数，基本上就可以编写出很多有用的程序了。

比如构造一个1, 3, 5, 7, ..., 99的列表，可以通过循环实现：

```
L = []
n = 1
while n <= 99:
    L.append(n)
    n = n + 2
```

取list的前一半的元素，也可以通过循环实现。

但是在Python中，代码不是越多越好，而是越少越好。代码不是越复杂越好，而是越简单越好。

基于这一思想，我们来介绍Python中非常有用的高级特性，一行代码能实现的功能，决不写5行代码。

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

取前3个元素，应该怎么做？

笨办法：

```
>>> [L[0], L[1], L[2]]
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

```
>>> r = []
>>> n = 3
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

如果第一个索引是0，还可以省略：

```
>>> L[:3]
['Michael', 'Sarah', 'Tracy']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

记住倒数第一个元素的索引是-1。

切片操作十分有用。我们先创建一个0-99的数列：

```
>>> L = range(100)
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前10个数：

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

后10个数：

```
>>> L[-10:]
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

```
>>> L[10:20]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
>>> L[:10:2]
[0, 2, 4, 6, 8]
```

```
>>> L[:5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

```
>>> (0, 1, 2, 3, 4, 5)[:3]
(0, 1, 2)
```

```
>>> 'ABCDEFGH'[:3]
'ABC'
>>> 'ABCDEFGH'[:2]
'AC'
>>> 'ABCDEFGH'[:2:2]
'ACEG'
```

有了切片操作，很多地方循环就不再需要了。**Python**的切片非常灵活，一行代码就可以实现很多行循环才能完成的操作。

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过for ... in来完成的，而很多语言比如C或者Java，迭代list是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {  
    n = list[i];  
}
```

可以看出，Python的for循环抽象程度要高于Java的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有了下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}  
>>> for key in d:  
...     print key  
...  
a  
c  
b
```

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.itervalues()，如果要同时迭代key和value，可以用for k, v in d.iteritems()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

```
>>> for ch in 'ABC':  
...     print ch  
...  
A  
B  
C
```

所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

```
>>> from collections import Iterable  
>>> isinstance('abc', Iterable) # str是否可迭代  
True  
>>> isinstance([1,2,3], Iterable) # list是否可迭代  
True  
>>> isinstance(123, Iterable) # 整数是否可迭代  
False
```

最后一个小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):  
...     print i, value  
...  
0 A  
1 B  
2 C
```

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:  
...     print x, y  
...  
1 1  
2 4
```


小结

任何可迭代对象都可以作用于`for`循环，包括我们自定义的数据类型，只要符合迭代条件，就可以使用`for`循环。

列表生成式即**List Comprehensions**，是Python内置的非常简单却强大的可以用来创建**list**的生成式。

举个例子，要生成**list** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用range(1, 11)：

```
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...     L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的**list**：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素x * x放到前面，后面跟for循环，就可以把**list**创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Work
```

for循环其实可以同时使用两个甚至多个变量，比如dict的iteritems()可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.iteritems():
...     print k, '=', v
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成**list**：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.iteritems()]
['y=B', 'x=A', 'z=C']
```

最后把一个**list**中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

小结

运用列表生成式，可以快速生成**list**，可以通过一个**list**推导出另一个**list**，而代码却十分简洁。

思考：如果**list**中既包含字符串，又包含整数，由于非字符串类型没有lower()方法，所以列表生成式会报错：

```
>>> L = ['Hello', 'World', 18, 'Apple', None]
>>> [s.lower() for s in L]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

使用内建的isinstance函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
False
```

请修改列表生成式，通过添加if语句保证列表生成式能正确地执行。