如果你需要某个Python函数或语句的快速信息帮助，那么你可以使用内建的help功能。尤其在你使用带提示符的命令行的时候，它十分有用。比如，运行help(str)——这会显示str类的帮助。str类用于保存你的程序使用的各种文本（字符串）。类将在后面面向对象编程的章节详细解释

类似地，你可以获取Python中几乎所有东西的信息。使用help()去学习更多关于help本身的东西！

如果你想要获取关于如print那样操作符的帮助，那么你需要正确的设置PYTHONDOCS环境变量。这可以在Linux/Unix中轻松地通过**env**命令完成。

|  |
| --- |
| $ env PYTHONDOCS=/usr/share/doc/python-docs-2.3.4/html/ python Python 2.3.4 (#1, Oct 26 2004, 16:42:40) [GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2 Type "help", "copyright", "credits" or "license" for more information. >>> help('print') |

你应该注意到我特意在“print”上使用了引号，那样Python就可以理解我是希望获取关于“print”的帮助而不是想要它打印东西。

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

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

值得注意的一件事是，在一个字符串中，行末的单独一个反斜杠表示字符串在下一行继续，而不是开始一个新的行。例如：

|  |
| --- |
| "This is the first sentence.\ This is the second sentence." |

等价于"This is the first sentence. This is the second sentence."

如果你想要指示某些不需要如转义符那样的特别处理的字符串，那么你需要指定一个自然字符串。自然字符串通过给字符串加上前缀r或R来指定。例如r"Newlines are indicated by \n"。

Unicode是书写国际文本的标准方法。如果你想要用你的母语如北印度语或阿拉伯语写文本，那么你需要有一个支持Unicode的编辑器。类似地，Python允许你处理Unicode文本——你只需要在字符串前加上前缀u或U。例如，u"This is a Unicode string."。

记住，在你处理文本文件的时候使用Unicode字符串，特别是当你知道这个文件含有用非英语的语言写的文本

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

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

|  |
| --- |
| i = 5; print i; |

完全造价于：

|  |
| --- |
| i = 5 print i |

然而，我**强烈建议**你坚持**在每个物理行只写一句逻辑行**。仅仅当逻辑行太长的时候，在多于一个物理行写一个逻辑行。这些都是为了尽可能避免使用分号，从而让代码更加易读。事实上，我 *从来没有* 在Python程序中使用过或看到过分号。

下面是一个在多个物理行中写一个逻辑行的例子。它被称为**明确的行连接**。

|  |
| --- |
| s = 'This is a string. \ This continues the string.' print s |

类似地，

|  |
| --- |
| print \ i |

与如下写法效果相同：

|  |
| --- |
| print i |

空白在Python中是重要的。事实上**行首的空白是重要的**。它称为**缩进**。在逻辑行首的空白（空格和制表符）用来决定逻辑行的缩进层次，从而用来决定语句的分组。

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

你需要记住的一样东西是错误的缩进会引发错误。例如：

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

当你运行这个程序的时候，你会得到下面的错误：

|  |
| --- |
| File "whitespace.py", line 4     print 'Value is', i # Error! Notice a single space at the start of the line     ^ SyntaxError: invalid syntax |

注意，在第二行的行首有一个空格。Python指示的这个错误告诉我们程序的语法是无效的，即程序没有正确地编写。它告诉你， *你不能随意地开始新的语句块* （当然除了你一直在使用的主块）。

**表5.1 运算符与它们的用法**

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **名称** | **说明** | **例子** |
| + | 加 | 两个对象相加 | 3 + 5得到8。'a' + 'b'得到'ab'。 |
| - | 减 | 得到负数或是一个数减去另一个数 | -5.2得到一个负数。50 - 24得到26。 |
| \* | 乘 | 两个数相乘或是返回一个被重复若干次的字符串 | 2 \* 3得到6。'la' \* 3得到'lalala'。 |
| \*\* | 幂 | 返回x的y次幂 | 3 \*\* 4得到81（即3 \* 3 \* 3 \* 3） |
| / | 除 | x除以y | 4/3得到1（整数的除法得到整数结果）。4.0/3或4/3.0得到1.3333333333333333 |
| // | 取整除 | 返回商的整数部分 | 4 // 3.0得到1.0 |
| % | 取模 | 返回除法的余数 | 8%3得到2。-25.5%2.25得到1.5 |
| << | 左移 | 把一个数的比特向左移一定数目（每个数在内存中都表示为比特或二进制数字，即0和1） | 2 << 2得到8。——2按比特表示为10 |
| >> | 右移 | 把一个数的比特向右移一定数目 | 11 >> 1得到5。——11按比特表示为1011，向右移动1比特后得到101，即十进制的5。 |
| & | 按位与 | 数的按位与 | 5 & 3得到1。 |
| | | 按位或 | 数的按位或 | 5 | 3得到7。 |
| ^ | 按位异或 | 数的按位异或 | 5 ^ 3得到6 |
| ~ | 按位翻转 | x的按位翻转是-(x+1) | ~5得到-6。 |
| < | 小于 | 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。 | 5 < 3返回0（即False）而3 < 5返回1（即True）。比较可以被任意连接：3 < 5 < 7返回True。 |
| > | 大于 | 返回x是否大于y | 5 > 3返回True。如果两个操作数都是数字，它们首先被转换为一个共同的类型。否则，它总是返回False。 |
| <= | 小于等于 | 返回x是否小于等于y | x = 3; y = 6; x <= y返回True。 |
| >= | 大于等于 | 返回x是否大于等于y | x = 4; y = 3; x >= y返回True。 |
| == | 等于 | 比较对象是否相等 | x = 2; y = 2; x == y返回True。x = 'str'; y = 'stR'; x == y返回False。x = 'str'; y = 'str'; x == y返回True。 |
| != | 不等于 | 比较两个对象是否不相等 | x = 2; y = 3; x != y返回True。 |
| not | 布尔“非” | 如果x为True，返回False。如果x为False，它返回True。 | x = True; not x返回False。 |
| and | 布尔“与” | 如果x为False，x and y返回False，否则它返回y的计算值。 | x = False; y = True; x and y，由于x是False，返回False。在这里，Python不会计算y，因为它知道这个表达式的值肯定是False（因为x是False）。这个现象称为短路计算。 |
| or | 布尔“或” | 如果x是True，它返回True，否则它返回y的计算值。 | x = True; y = False; x or y返回True。短路计算在这里也适用。 |

**表5.2 运算符优先级**

|  |  |
| --- | --- |
| **运算符** | **描述** |
| lambda | Lambda表达式 |
| or | 布尔“或” |
| and | 布尔“与” |
| not x | 布尔“非” |
| in，not in | 成员测试 |
| is，is not | 同一性测试 |
| <，<=，>，>=，!=，== | 比较 |
| | | 按位或 |
| ^ | 按位异或 |
| & | 按位与 |
| <<，>> | 移位 |
| +，- | 加法与减法 |
| \*，/，% | 乘法、除法与取余 |
| +x，-x | 正负号 |
| ~x | 按位翻转 |
| \*\* | 指数 |
| x.attribute | 属性参考 |
| x[index] | 下标 |
| x[index:index] | 寻址段 |
| f(arguments...) | 函数调用 |
| (experession,...) | 绑定或元组显示 |
| [expression,...] | 列表显示 |
| {key:datum,...} | 字典显示 |
| 'expression,...' | 字符串转换 |

|  |
| --- |
| #!/usr/bin/python # Filename: expression.py  length = 5 breadth = 2 area = length \* breadth print 'Area is', area print 'Perimeter is', 2 \* (length + breadth) |

输出为：

|  |
| --- |
| $ python expression.py Area is 10 Perimeter is 14 |

注意Python如何打印“漂亮的”输出。尽管我们没有在'Area is'和变量area之间指定空格，Python自动在那里放了一个空格，这样我们就可以得到一个清晰漂亮的输出，而程序也变得更加易读（因为我们不需要担心输出之间的空格问题）。这是Python如何使程序员的生活变得更加轻松的一个例子。（手动添加一个空格反而会让它在is和10之间加了两个空格）

if语句

|  |
| --- |
| #!/usr/bin/python # Filename: if.py   number = 23 guess = int(raw\_input('Enter an integer : '))  if guess == number:     print 'Congratulations, you guessed it.' # New block starts here     print "(but you do not win any prizes!)" # New block ends here elif guess < number:     print 'No, it is a little higher than that' # Another block     # You can do whatever you want in a block ... else:     print 'No, it is a little lower than that'      # you must have guess > number to reach here  print 'Done' # This last statement is always executed, after the if statement is executed |

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

注意我们使用了缩进层次来告诉Python每个语句分别属于哪一个块。这就是为什么缩进在Python如此重要的原因。

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

While语句

|  |
| --- |
| #!/usr/bin/python # Filename: while.py  number = 23 running = **T**rue  while running:     guess = int(raw\_input('Enter an integer : '))      if guess == number:         print 'Congratulations, you guessed it.'          running = **F**alse # this causes the while loop to stop     elif guess < number:         print 'No, it is a little higher than that'      else:         print 'No, it is a little lower than that'  else:     print 'The while loop is over.'      # Do anything else you want to do here  print 'Done' |

True和False被称为布尔类型。你可以分别把它们等效地理解为值1和0。在检验重要条件的时候，布尔类型十分重要，它们并不是真实的值1。

else块事实上是多余的，因为你可以把其中的语句放在同一块（与while相同）中，跟在while语句之后，这样可以取得相同的效果。

For语句

|  |
| --- |
| #!/usr/bin/python # Filename: for.py  for i in range(1, 5):     print i else:     print 'The for loop is over' |

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

记住，else部分是可选的。如果包含else，它总是在for循环结束后执行一次，除非遇到[break](http://sebug.net/paper/python/ch06s05.html)语句。（把else删除，并把最后的一个print语句前面的空格删除，其实是与上面的代码是等价的）

**break语句**

|  |
| --- |
| #!/usr/bin/python # Filename: break.py  while True:     s = raw\_input('Enter something : ')     if s == 'quit'**:**         break     print 'Length of the string is', len(s) print 'Done' |

函数：

函数是重用的程序段。它们允许你给一块语句一个名称，然后你可以在你的程序的任何地方使用这个名称任意多次地运行这个语句块。这被称为 *调用* 函数。我们已经使用了许多内建的函数，比如len和range。

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

|  |
| --- |
| #!/usr/bin/python # Filename: function1.py  def sayHello():     print 'Hello World!' # block belonging to the function  sayHello() # call the function |

**局部变量**

|  |
| --- |
| #!/usr/bin/python # Filename: func\_local.py  def func(x):     print 'x is', x     x = 2     print 'Changed local x to', x  x = 50 func(x) print 'x is still', x |

输出为：

|  |
| --- |
| $ python func\_local.py x is 50 Changed local x to 2 x is still 50 |

**使用global语句**

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

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

|  |
| --- |
| #!/usr/bin/python # Filename: func\_global.py  def func():     global x      print 'x is', x     x = 2     print 'Changed local x to', x  x = 50 func() print 'Value of x is', x |

**输出**

|  |
| --- |
| $ python func\_global.py x is 50 Changed global x to 2 Value of x is 2 |

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

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

**默认参数值**

|  |
| --- |
| #!/usr/bin/python # Filename: func\_default.py  def say(message, times = 1):     print message \* times  say('Hello') say('World', 5) |

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

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

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

**关键参数**

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

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

|  |
| --- |
| #!/usr/bin/python # Filename: func\_key.py  def func(a, b=5, c=10):     print 'a is', a, 'and b is', b, 'and c is', c  func(3, 7) func(25, c=24) func(c=50, a=100) |

输出：

|  |
| --- |
| $ python func\_key.py a is 3 and b is 7 and c is 10 a is 25 and b is 5 and c is 24 a is 100 and b is 5 and c is 50 |

**return语句**

注意，没有返回值的return语句等价于return None。None是Python中表示没有任何东西的特殊类型。例如，如果一个变量的值为None，可以表示它没有值。

除非你提供你自己的return语句，每个函数都在结尾暗含有return None语句。通过运行print someFunction()，你可以明白这一点，函数someFunction没有使用return语句，如同：

def someFunction():  
    pass

pass语句在Python中表示一个空的语句块。

**DocStrings**

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

|  |
| --- |
| #!/usr/bin/python # Filename: func\_doc.py  def printMax(x, y):     '''Prints the maximum of two numbers.      The two values must be integers.'''     x = int(x) # convert to integers, if possible     y = int(y)      if x > y:         print x, 'is maximum'     else:         print y, 'is maximum'  printMax(3, 5) print printMax.\_\_doc\_\_ |

输出：

|  |
| --- |
| $ python func\_doc.py 5 is maximum Prints the maximum of two numbers.          The two values must be integers. |

在函数的第一个逻辑行的字符串是这个函数的 *文档字符串* 。注意，DocStrings也适用于[模块](http://sebug.net/paper/python/ch08.html)和[类](http://sebug.net/paper/python/ch11.html)，我们会在后面相应的章节学习它们。

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

你可以使用\_\_doc\_\_（注意双下划线）调用printMax函数的文档字符串属性（属于函数的名称）。请记住Python把 *每一样东西* 都作为对象，包括这个函数。我们会在后面的[类](http://sebug.net/paper/python/ch11.html)一章学习更多关于对象的知识。

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

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

**模块**

为了在其他程序中重用模块，模块的文件名**必须**以.py为扩展名。

|  |
| --- |
| #!/usr/bin/python # Filename: using\_sys.py  import sys  print 'The command line arguments are:' for i in sys.argv:     print i  print '\n\nThe PYTHONPATH is', sys.path, '\n' |

输出：

|  |
| --- |
| $ python using\_sys.py we are arguments The command line arguments are: using\_sys.py we are arguments   The PYTHONPATH is ['/home/swaroop/byte/code', '/usr/lib/python23.zip', '/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2', '/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload', '/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/gtk-2.0'] |

当Python执行import sys语句的时候，它在sys.path变量中所列目录中寻找sys.py模块。如果找到了这个文件，这个模块的主块中的语句将被运行，然后这个模块将能够被你 *使用* 。注意，初始化过程仅在我们 *第一次* 输入模块的时候进行。另外，“sys”是“system”的缩写。

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

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

**字节编译的.pyc文件**

输入一个模块相对来说是一个比较费时的事情，所以Python做了一些技巧，以便使输入模块更加快一些。一种方法是创建 *字节编译的文件* ，这些文件以.pyc作为扩展名。字节编译的文件与Python变换程序的中间状态有关（是否还记得[Python如何工作的介绍](http://sebug.net/paper/python/ch01s02.html#intermediate)？）。当你在下次从别的程序输入这个模块的时候，.pyc文件是十分有用的——它会快得多，因为一部分输入模块所需的处理已经完成了。另外，这些字节编译的文件也是与平台无关的。所以，现在你知道了那些.pyc文件事实上是什么了。

**from..import语句**

如果你想要直接输入argv变量到你的程序中（避免在每次使用它时打sys.），那么你可以使用from sys import argv语句。如果你想要输入所有sys模块使用的名字，那么你可以使用from sys import \*语句。这对于所有模块都适用。一般说来，应该避免使用from..import而使用import语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

**模块的\_\_name\_\_**

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

|  |
| --- |
| #!/usr/bin/python # Filename: using\_name.py  if \_\_name\_\_ == '\_\_main\_\_':     print 'This program is being run by itself' else:     print 'I am being imported from another module' |

|  |
| --- |
| $ python using\_name.py This program is being run by itself  $ python >>> import using\_name I am being imported from another module >>> |

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

**创建你自己的模块**

|  |
| --- |
| #!/usr/bin/python # Filename: mymodule.py  def sayhi():     print 'Hi, this is mymodule speaking.'  version = '0.1'#这个应当只是定义了一个变量，并不是一定要这样写  # End of mymodule.py |

|  |
| --- |
| #!/usr/bin/python # Filename: mymodule\_demo.py  import mymodule  mymodule.sayhi() print 'Version', mymodule.version |

输出：

|  |
| --- |
| $ python mymodule\_demo.py Hi, this is mymodule speaking. Version 0.1 |

下面是一个使用from..import语法的版本。

|  |
| --- |
| #!/usr/bin/python # Filename: mymodule\_demo2.py  from mymodule import sayhi, version # Alternative: # from mymodule import \*  sayhi() print 'Version', version |

**dir()函数**

你可以使用内建的dir函数来列出模块定义的标识符。标识符有函数、类和变量。

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

|  |
| --- |
| $ python >>> import sys >>> dir(sys) # get list of attributes for sys module ['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_name\_\_', '\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_', '\_getframe', 'api\_version', 'argv', 'builtin\_module\_names', 'byteorder', 'call\_tracing', 'callstats', 'copyright', 'displayhook', 'exc\_clear', 'exc\_info', 'exc\_type', 'excepthook', 'exec\_prefix', 'executable', 'exit', 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding', 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode', 'meta\_path','modules', 'path', 'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version', 'version\_info', 'warnoptions'] >>> dir() # get list of attributes for current module ['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'sys'] >>> >>> a = 5 # create a new variable 'a' >>> dir() ['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'sys'] >>> >>> del a # delete/remove a name >>> >>> dir() ['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'sys'] >>> |

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

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

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

关于del的一点注释——这个语句在运行后被用来 *删除* 一个变量/名称。在这个例子中，del a，你将无法再使用变量a——它就好像从来没有存在过一样。

# Python基础教程图书拾遗

|  |
| --- |
| x = [1,2,3,4]  x.reverse()  print(list(x)) |

输出：

|  |
| --- |
| C:\>python PythonTest.py  [4, 3, 2, 1] |

以下代码会达到一样的效果：

|  |
| --- |
| x = [1,2,3,4]  print(list(reversed(x))) |

reversed函数并不返回一个列表，而是返回一个迭代器