Python语言入门

介绍

大学里面大家学过C语言，C++或者Java语言，甚至也有汇编，很多人大学里可能没有接触过python，其实python做为一门胶水语言，语法简洁清晰，具有丰富和强大的库，这篇文章试图通过一些基础的介绍和实战，使大家对python有一个整体的了解并在日常工作的使用中感受到它的简洁和高效。

基本概念

字面意义上的常量

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

变量

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

数

在Python中有4种类型的数——整数、长整数、浮点数和复数。

- 2是一个整数的例子。
- 长整数不过是大一些的整数。

- 3.23和52.3E-4是浮点数的例子。E标记表示10的幂。在这里，52.3E-4表示52.3 * 10-4。
- (-5+4j)和(2.3-4.6j)是复数的例子。

字符串

字符串是 字符的序列 。字符串基本上就是一组单词。

我几乎可以保证你在每个Python程序中都要用到字符串，所以请特别留心下面这部分的内容。下面告诉你如何在Python中使用字符串。

- 使用单引号（'）

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

- 使用双引号（"）

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

- 使用三引号（'''或"""）

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."
'''
```

标识符命名

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

- 标识符的第一个字符必须是字母表中的字母（大写或小写）或者一个下划线（‘ _ ’）。
- 标识符名称的其他部分可以由字母（大写或小写）、下划线（‘ _ ’）或数字（0-9）组成。
- 标识符名称是对大小写敏感的。例如，myname和myName不是一个标识符。注意前者中的小写n和后者中的大写N。
- 有效标识符名称的例子有i、*_my_name*、*name*23和a1b2_c3。
- 无效标识符名称的例子有2things、this is spaced out和my-name。

缩进

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。

逻辑行与物理行

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

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

默认地，Python希望每行都只使用一个语句，这样使得代码更加易读。

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

```
i = 5
print i
```

与下边这个相同：

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

同样也可以写成：

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

但我目前还没见到写python代码加分号的，关于这方便内容和上一小节提到的缩进建议查看官方语言规范。

控制流

if语句

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

```
#!/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
```

输出:

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done
$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done
$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

while语句

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

number = 23
running = True

while running:
    guess = int(raw_input('Enter an integer : '))

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

print 'Done'
```

输出:

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

for循环

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

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

输出:

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

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'
```

输出:

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

continue语句

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

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

输出:

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

数据结构

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

在Python中有三种内建的数据结构——列表、元组和字典。我们将会学习如何使用它们，以及它们如何使编程变得简单。

列表

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

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

对象与类的快速入门

尽管我一直推迟讨论对象和类，但是现在对它们做一点解释可以使你更好的理解列表。我们会在相应的章节详细探索这个主题。

列表是使用对象和类的一个例子。当你使用变量i并给它赋值的时候，比如赋整数5，你可以认为你创建了一个类（类型）int的对象（实例）i。事实上，你可以看一下help(int)以更好地理解这一点。

类也有方法，即仅仅为类而定义地函数。仅仅在你有一个该类的对象的时候，你才可以使用这些功能。例如，Python为list类提供了append方法，这个方法让你在列表尾添加一个项目。例如mylist.append('an item')列表mylist中增加那个字符串。注意，使用点号来使用对象的方法。

一个类也有域，它是仅仅为类而定义的变量。仅仅在你有一个该类的对象的时候，你才可以使用这些变量/名称。类也通过点号使用，例如mylist.field。

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

# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print 'I have', len(shoplist),'items to purchase.'

print 'These items are:', # Notice the comma at end of the line
for item in shoplist:
    print item,

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

print 'I will sort my list now'
shoplist.sort()
print 'Sorted shopping list is', shoplist

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

输出:

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

它如何工作

变量shoplist是某人的购物列表。在shoplist中，我们只存储购买的东西的名字字符串，但是记住，你可以在列表中添加 任何种类的对象 包括数甚至其他列表。

我们也使用了for..in循环在列表中各项目间递归。从现在开始，你一定已经意识到列表也是一个序列。序列的特性会在后面的章节中讨论。

注意，我们在print语句的结尾使用了一个 逗号 来消除每个print语句自动打印的换行符。这样做有点难看，不过确实简单有效。

接下来，我们使用append方法在列表中添加了一个项目，就如前面已经讨论过的一样。然后我们通过打印列表的内容来检验这个项目是否确实被添加进列表了。打印列表只需简单地把列表传递给print语句，我们可以得到一个整洁的输出。

再接下来，我们使用列表的sort方法来对列表排序。需要理解的是，这个方法影响列表本身，而不是返回一个修改后的列表——这与字符串工作的方法不同。这就是我们所说的列表是 可变的 而字符串是不可变的 。

最后，但我们完成了在市场购买一样东西的时候，我们想要把它从列表中删除。我们使用del语句来完成这个工作。这里，我们指出我们想要删除列表中的哪个项目，而del语句为我们从列表中删除它。我们指明我们想要删除列表中的第一个元素，因此我们使用del shoplist[0]（记住，Python从0开始计数）。

如果你想要知道列表对象定义的所有方法，可以通过help(list)获得完整的知识。

元组

元组和列表十分类似，只不过元组和字符串一样是 不可变的 即你不能修改元组。元组通过圆括号中用逗号分割的项目定义。元组通常用在使语句或用户定义的函数能够安全地采用一组值的时候，即被使用的元组的值不会改变。

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

zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)

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

输出:

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

它如何工作

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

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

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

含有0个或1个项目的元组。一个空的元组由一对空的圆括号组成，如myempty = ()。然而，含有单个元素的元组就不那么简单了。你必须在第一个（唯一一个）项目后跟一个逗号，这样Python才能区分元组和表达式中一个带圆括号的对象。即如果你想要的是一个包含项目2的元组的时候，你应该指明singleton = (2 ,)。

元组与打印语句

元组最通常的用法是用在打印语句中，下面是一个例子：

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

age = 22
name = 'Swaroop'

print '%s is %d years old' % (name, age)
print 'Why is %s playing with that python?' % name
```

输出:

```
$ python print_tuple.py
Swaroop is 22 years old
Why is Swaroop playing with that python?
```

它如何工作

print语句可以使用跟着%符号的项目元组的字符串。这些字符串具备定制的功能。定制让输出满足某种特定的格式。定制可以是%s表示字符串或%d表示整数。元组必须按照相同的顺序来对应这些定制。

观察我们使用的第一个元组，我们首先使用%s，这对应变量name，它是元组中的第一个项目。而第二个定制是%d，它对应元组的第二个项目age。

Python在这里所做的是把元组中的每个项目转换成字符串并且用字符串的值替换定制的位置。因此%s被替换为变量name的值，依此类推。

print的这个用法使得编写输出变得极其简单，它避免了许多字符串操作。它也避免了我们一直以来使用的逗号。

在大多数时候，你可以只使用%s定制，而让Python来提你处理剩余的事情。这种方法对数同样奏效。然而，你可能希望使用正确的定制，从而可以避免多一层的检验程序是否正确。

在第二个print语句中，我们使用了一个定制，后面跟着%符号后的单个项目——没有圆括号。这只在字符串中只有一个定制的时候有效。

字典

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

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

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

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

字典是dict类的实例/对象。

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

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

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

print "Swaroop's address is %s" % ab['Swaroop']

# Adding a key/value pair
ab['Guido'] = 'guido@python.org'

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

print '\nThere are %d contacts in the address-book\n' % len(ab)
for name, address in ab.items():
    print 'Contact %s at %s' % (name, address)

if 'Guido' in ab: # OR ab.has_key('Guido')
    print "\nGuido's address is %s" % ab['Guido']
```

输出:

```
$ python using_dict.py
Swaroop's address is swaroopch@byteofpython.info

There are 4 contacts in the address-book

Contact Swaroop at swaroopch@byteofpython.info
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org
Contact Guido at guido@python.org

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

它如何工作

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

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

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

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

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

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

序列

列表、元组和字符串都是序列，但是序列是什么，它们为什么如此特别呢？序列的两个主要特点是索引操作符和切片操作符。索引操作符让我们可以从序列中抓取一个特定项目。切片操作符让我们能够获取序列的一个切片，即一部分序列。

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

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

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

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

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

输出:

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

它如何工作

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

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

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

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

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

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

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

参考

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

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

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

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

del shoplist[0]

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

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

print 'shoplist is', shoplist
print 'mylist is', mylist
# notice that now the two lists are different
```

输出:

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

它如何工作

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

更多字符串的内容

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

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

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

name = 'Swaroop' # This is a string object

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

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

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

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

输出:

```
$ python str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

它如何工作

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

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

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
```

输出:

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

它如何工作

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

函数形参

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

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

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

def printMax(a, b):
    if a > b:
        print a, 'is maximum'
    else:
        print b, 'is maximum'

printMax(3, 4) # directly give literal values

x = 5
y = 7

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

输出

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

它如何工作

这里，我们定义了一个称为printMax的函数，这个函数需要两个形参，叫做a和b。我们使用if..else语句找出两者之中较大的一个数，并且打印较大的那个数。

在第一个printMax使用中，我们直接把数，即实参，提供给函数。在第二个使用中，我们使用变量调用函数。printMax(x, y)使实参x的值赋给形参a，实参y的值赋给形参b。在两次调用中，printMax函数的工作完全相同。

局部变量

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

```
#!/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

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

它如何工作

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

接下来，我们把值2赋给x。x是函数的局部变量。所以，当我们在函数内改变x的值的时候，在主块中定义的x不受影响。

在最后一个print语句中，我们证明了主块中的x的值确实没有受到影响。

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
```

它如何工作

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

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

默认参数值

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

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

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

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

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

输出

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

它如何工作

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。

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
```

它如何工作

名为func的函数有一个没有默认值的参数，和两个有默认值的参数。

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

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

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

return语句

return语句用来从一个函数 返回 即跳出函数。我们也可选从函数 返回一个值 。

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

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

print maximum(2, 3)
```

输出

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

它如何工作

maximum函数返回参数中的最大值，在这里是提供给函数的数。它使用简单的if..else语句来找出较大的值，然后 返回 那个值。

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

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

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

pass语句在Python中表示一个空的语句块。

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.
```

它如何工作

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

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

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

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

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

模块

你已经学习了如何在你的程序中定义一次函数而重用代码。如果你想要在其他程序中重用很多函数，那么你该如何编写程序呢？你可能已经猜到了，答案是使用模块。模块基本上就是一个包含了所有你定义的函数和变量的文件。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。

模块可以从其他程序 输入 以便利用它的功能。这也是我们使用Python标准库的方法。首先，我们将学习如何使用标准库模块。

使用sys模块

```
#!/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']
```

它如何工作

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

当Python执行import sys语句的时候，它在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。

sys模块中的argv变量通过使用点号指明——sys.argv——这种方法的一个优势是这个名称不会与任何在你的程序中使用的argv变量冲突。另外，它也清晰地表明了这个名称是sys模块的一部分。

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

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

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

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

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

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？）。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。

使用模块的name

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

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

输出

```
$ python using_name.py
This program is being run by itself

$ python
>>> import using_name
I am being imported from another module
>>>
```

它如何工作

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

Python 可以在模块级别暴露接口：

```
__all__ = ["foo", "bar"]
```

制造你自己的模块

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

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

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

version = '0.1'

# End of mymodule.py
```

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

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

输出

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

import mymodule

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

它如何工作

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

面向对象

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

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

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

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

类使用class关键字创建。类的域和方法被列在一个缩进块中。

类

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

class Person:
    pass # An empty block

p = Person()
print p
```

输出

```
$ python simplestclass.py
<__main__.Person instance at 0xf6fcb18c>
```

它如何工作

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

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

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

对象的方法

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

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

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

p = Person()
p.sayHi()

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

输出

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

它如何工作

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

init方法

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

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

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

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

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

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

输出

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

它如何工作

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

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

现在，我们能够在我们的方法中使用self.name域。这在sayHi方法中得到了验证。

类与对象的变量

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

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

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

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

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

class Person:
    '''Represents a person.'''
    population = 0

    def __init__(self, name):
        '''Initializes the person's data.'''
        self.name = name
        print '(Initializing %s)' % self.name

        # When this person is created, he/she
        # adds to the population
        Person.population += 1

    def __del__(self):
        '''I am dying.'''
        print '%s says bye.' % self.name

        Person.population -= 1

        if Person.population == 0:
```

```
            print 'I am the last one.'
        else:
            print 'There are still %d people left.' % Person.population

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

        Really, that's all it does.'''
        print 'Hi, my name is %s.' % self.name

    def howMany(self):
        '''Prints the current population.'''
        if Person.population == 1:
            print 'I am the only person here.'
        else:
            print 'We have %d persons here.' % Person.population

swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()

kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

swaroop.sayHi()
swaroop.howMany()
```

输出

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
I am the only person here.
(Initializing Abdul Kalam)
Hi, my name is Abdul Kalam.
We have 2 persons here.
Hi, my name is Swaroop.
We have 2 persons here.
Abdul Kalam says bye.
There are still 1 people left.
Swaroop says bye.
I am the last one.
```

它如何工作

这是一个很长的例子，但是它有助于说明类与对象的变量的本质。这里，population属于Person类，因此是一个类的变量。name变量属于对象（它使用self赋值）因此是对象的变量。

观察可以发现**init**方法用一个名字来初始化Person实例。在这个方法中，我们让population增加1，这是因为我们增加了一个人。同样可以发现，self.name的值根据每个对象指定，这表明了它作为对象的变量的本质。

记住，你只能使用self变量来参考同一个对象的变量和方法。这被称为 属性参考 。

在这个程序中，我们还看到docstring对于类和方法同样有用。我们可以在运行时使用Person.**doc**和Person.sayHi.**doc**来分别访问类与方法的文档字符串。

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

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

继承

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

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

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

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

使用这种方法有很多优点。如果我们增加/改变了SchoolMember中的任何功能，它会自动地反映到子类型之中。例如，你要为教师和学生都增加一个新的身份证域，那么你只需简单地把它加到SchoolMember类中。然而，在一个子类型之中做的改动不会影响到别的子类型。另外一个优点是你可以把教师和学生对象都作为SchoolMember对象来使用，这在某些场合特别有用，比如统计学校成员的人数。一个子类型在任何需要父类型的场合可以被替换成父类型，即对象可以被视作是父类的实例，这种现象被称为多态现象。

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

在上述的场合中，SchoolMember类被称为 基本类 或 超类 。而Teacher和Student类被称为 导出类 或 子类 。

现在，我们将学习一个例子程序。

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

class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: %s)' % self.name

    def tell(self):
        '''Tell my details.'''
        print 'Name:"%s" Age:"%s"' % (self.name, self.age),

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Salary: "%d"' % self.salary

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: %s)' % self.name

    def tell(self):
        SchoolMember.tell(self)
        print 'Marks: "%d"' % self.marks

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 22, 75)

print # prints a blank line

members = [t, s]
for member in members:
    member.tell() # works for both Teachers and Students
```

输出

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

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"22" Marks: "75"
```

它如何工作

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

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

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

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

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

Python语言进阶

Python标准库

Python标准库是随Python附带安装的，它包含大量极其有用的模块。熟悉Python标准库是十分重要的，因为如果你熟悉这些库中的模块，那么你的大多数问题都可以简单快捷地使用它们来解决。
我们已经研究了一些这个库中的常用模块。你可以在Python附带安装的文档的“库参考”一节中了解Python标准库中所有模块的完整内容。

核心模块

math模块

math 模块实现了许多对浮点数的数学运算函数. 这些函数一般是对平台 C 库
中同名函数的简单封装, 所以一般情况下, 不同平台下计算的结果可能稍微地
有所不同, 有时候甚至有很大出入. 如下展示了如何使用 math 模块.

```
import math
print "e", "=>", math.e
print "pi", "=>", math.pi
print "hypot", "=>", math.hypot(3.0, 4.0)
```

输出

```
e => 2.71828182846
pi => 3.14159265359
hypot => 5.0
```

完整函数列表请参阅 Python Library Reference .

string模块

string 模块提供了一些用于处理字符串类型的函数, 如下所示.

使用string模块

```
import string
text = "Monty Python's Flying Circus"
print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text,
"Java")
print "count", "=>", string.count(text, "n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

使用字符串方法替代 string 模块函数

```
text = "Monty Python's Flying Circus"
print "upper", "=>", text.upper()
print "lower", "=>", text.lower()
print "split", "=>", text.split()
print "join", "=>", "+".join(text.split())
print "replace", "=>", text.replace("Python", "Perl")
print "find", "=>", text.find("Python"), text.find("Perl")
print "count", "=>", text.count("n")
```

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Perl's Flying Circus
find => 6 -1
count => 3
```

使用 string 模块将字符串转为数字

```
#! usr/bin/python
#coding=utf-8

import string
print int("4711"),
print string.atoi("4711"),
print string.atoi("11147", 8), # octal 八进制
print string.atoi("1267", 16), # hexadecimal 十六进制
print string.atoi("3mv", 36) # whatever...
print string.atoi("4711", 0),
print string.atoi("04711", 0),
print string.atoi("0x4711", 0)
print float("4711"),
print string.atof("1"),
print string.atof("1.23e5")
```

```
4711 4711 4711 4711 4711
4711 2505 18193
4711.0 1.0 123000.0
```

time模块

time 模块提供了一些处理日期和一天内时间的函数. 它是建立在 C 运行时库的简单封装.
给定的日期和时间可以被表示为浮点型(从参考时间, 通常是 1970.1.1 到现在经过的秒数. 即 Unix 格式), 或者一个表示时间的 struct (类元组).

使用 time 模块获取当前时间

```
import time
now = time.time()
print now, "seconds since", time.gmtime(0)[:6]
print
print "or in other words:"
print "- local time:", time.localtime(now)
print "- utc:", time.gmtime(now)
```

```
1491569666.39 seconds since (1970, 1, 1, 0, 0, 0)

or in other words:
- local time: time.struct_time(tm_year=2017, tm_mon=4, tm_mday=7, tm_hour=20,
tm_min=54, tm_sec=26, tm_wday=4, tm_yday=97, tm_isdst=0)
- utc: time.struct_time(tm_year=2017, tm_mon=4, tm_mday=7, tm_hour=12,
tm_min=54, tm_sec=26, tm_wday=4, tm_yday=97, tm_isdst=0)
```

将时间值转换为字符串

```
import time
now = time.localtime(time.time())
print time.asctime(now)
print time.strftime("%y/%m/%d %H:%M", now)
print time.strftime("%a %b %d", now)
print time.strftime("%c", now)
print time.strftime("%I %p", now)
print time.strftime("%Y-%m-%d %H:%M:%S %Z", now)
# do it by hand...
year, month, day, hour, minute, second, weekday, yearday, daylight = now
print "%04d-%02d-%02d" % (year, month, day)
print "%02d:%02d:%02d" % (hour, minute, second)
print ("MON", "TUE", "WED", "THU", "FRI", "SAT", "SUN")[weekday], yearday
```

```
Fri Apr 07 20:57:09 2017
17/04/07 20:57
Fri Apr 07
04/07/17 20:57:09
08 PM
2017-04-07 20:57:09 中国标准时间
2017-04-07
20:57:09
FRI 97
```

更多标准模块

StringIO模块

StringIO 模块实现了一个工作在内存的文件
对象 (内存文件). 在大多需要标准文件对象的地方都可以使用它来替换.

使用 StringIO 模块从内存文件读入内容

```
import StringIO
MESSAGE = "That man is depriving a village somewhere of a computer
scientist."
file = StringIO.StringIO(MESSAGE)
print file.read()
```

```
That man is depriving a village somewhere of a computer scientist.
```

使用 StringIO 模块向内存文件写入内容

```
import StringIO
file = StringIO.StringIO()
file.write("This man is no ordinary man. ")
file.write("This is Mr. F. G. Superman.")
print file.getvalue()
```

```
This man is no ordinary man. This is Mr. F. G. Superman.
```

pstats模块

pstats 模块用于分析 Python 分析器收集的数据.如下所示：

```
import pstats
import profile
def func1():
      for i in range(1000):
            pass
def func2():
      for i in range(1000):
            func1()
p = profile.Profile()
p.run("func2()")
s = pstats.Stats(p)
s.sort_stats("time", "name").print_stats()
```

```
         2005 function calls in 0.065 seconds

   Ordered by: internal time, function name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.051    0.000    0.063    0.000
C:\Users\Administrator\Desktop\aaa.py:6(func1)
     1001    0.012    0.000    0.012    0.000 :0(range)
        1    0.002    0.002    0.065    0.065
C:\Users\Administrator\Desktop\aaa.py:9(func2)
        1    0.000    0.000    0.065    0.065 profile:0(func2())
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.065    0.065 <string>:1(<module>)
        0    0.000             0.000          profile:0(profiler)
```

getpass模块

getpass 模块提供了平台无关的在命令行下输入密码的方法. 如下所示.
getpass(prompt) 会显示提示字符串, 关闭键盘的屏幕反馈, 然后读取密码.

```
import getpass
usr = getpass.getuser()
pwd = getpass.getpass("enter password for user %s: " % usr)
print usr, pwd
```

```
enter password for user yangtze:
yangtze 123456
```

多线程

Python标准库模块(thread, threading)提供了对多线程的支持
// TODO

网络编程

网络编程涉及的socket套接字知识都相似的
// TODO

实战

内置HTTP协议服务器

Python内置的HTTP协议服务器SimpleHTTPServer本身的功能十分简单，只需要在命令行下面敲一行命令，一个HTTP服务器就起来了：

```
python -m SimpleHTTPServer 80
```

后面的80端口是可选的，不填会采用缺省端口8000。注意，这会将当前所在的文件夹设置为默认的Web目录，试着在浏览器敲入本机地址：`http://localhost:80`

如果当前文件夹有index.html文件，会默认显示该文件，否则，会以文件列表的形式显示目录下所有文件。这样已经实现了最基本的文件分享的目的，你可以做成一个脚本，再建立一个快捷方式，就可以很方便的启动文件分享了。如果有更多需求，完全可以根据自己需要定制。

学习开发Post工具

工作经历，开发维护了一段时间老东家的项目，模块通信使用的是HTTP+XML格式请求，时不时的要开发修改接口，联调之前都是post工具自验。

```
#!/usr/bin/env python
# Filename: post.py

import sys
def send(host,port,request):
    import socket
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect((host,port))
    s.sendall("POST / HTTP/1.1\r\nContent-Length:%d\r\n\r\n%s\r\n"%
(len(request),request))
    while 1:
        buf = s.recv(1024*8)
        if not len(buf):
            break;
        sys.stdout.write(buf)
    sys.stdout.write("\r\n")

#parse argv
n = len(sys.argv)
if n != 4:
    print "Usage: ./post.py IP_ADDRESS PORT post.xml"
    sys.exit()
else:
    send(sys.argv[1],int(sys.argv[2]),open(sys.argv[3]).read())
```

豆瓣爬虫项目

一个烂大街的爬虫小项目（大家这么喜爱抓豆瓣信息也是对豆瓣网页结构简洁的肯定吧），可以作为scrapy爬虫使用入门来介绍。Scrapy是Python开发的一个快速 web爬虫抓取框架，用于抓取web站点并从页面中提取结构化的数据。

1. 目标网站：豆瓣电影TOP250
2. 目标网址：http://movie.douban.com/top250
3. 目标内容：豆瓣电影TOP250中的电影信息，内容包括了：
    1. 电影名称

2. 电影信息
3. 电影引述

4. 输出结果：生成csv文件

安装Scrapy,执行`scrapy startproject douban`即可生成一个豆瓣爬虫项目，其中主要包括：

- items.py文件
- settings.py文件
- pipelines.py文件
- spider.py文件

items.py定义需要抓取并需要后期处理的数据。
settings.py文件配置Scrapy，从而修改user-agent，设定爬取时间间隔，设置代理，配置各种中间件等等。
pipeline.py用于存放执行后期数据处理的功能，从而使得数据的爬取和处理分开。
spider.py实现了一个Douban类，通过接口完成爬取URI和提取Item的功能。

items.py文件定义了我们想要抓取的数据结构，具体如下：

```
import scrapy

class DoubanItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    title = scrapy.Field()
    movieInfo = scrapy.Field()
    star = scrapy.Field()
    quote = scrapy.Field()
```

settings.py文件伪造user-agent，将爬取内容保存至当前目录csv文件中，具体如下：

```
BOT_NAME = 'douban'
SPIDER_MODULES = ['douban.spiders']
NEWSPIDER_MODULE = 'douban.spiders'

# Crawl responsibly by identifying yourself (and your website) on the user-
agent
#USER_AGENT = 'douban (+http://www.yourdomain.com)'
USER_AGENT = 'Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like
Gecko) Ubuntu Chromium/37.0.2062.120 Chrome/37.0.2062.120 '

FEED_URI = u'./douban.csv'
FEED_FORMAT = 'CSV'
```

这里我们没有做item列表的数据处理而是直接返回，pipeline.py文件具体如下：

```
class DoubanPipeline(object):
    def process_item(self, item, spider):
        return item
```

Douban类中实现了xpath解析网页，爬虫编写的脏活累活都在这儿啦，spider.py随着抓取的网页结构的变化需要做相应修改，具体如下：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy.contrib.spiders import CrawlSpider
from scrapy.http import Request
from scrapy.selector import Selector
from douban.items import DoubanItem

class Douban(CrawlSpider):
        name = "douban"
        redis_key = 'douban:start_urls'
        start_urls = ['http://movie.douban.com/top250']

        url = 'http://movie.douban.com/top250'

        def parse(self,response):
                #print response.body
                item = DoubanItem()
                selector = Selector(response)
                Movies = selector.xpath('//div[@class="info"]')
                for eachMovie in Movies:
                        title =
eachMovie.xpath('div[@class="hd"]/a/span/text()').extract()
                        fullTitle = ''
                        for each in title:
                                fullTitle += each
                        movieInfo =
eachMovie.xpath('div[@class="bd"]/p[@class=""]/text()').extract()
                        star =
eachMovie.xpath('div[@class="bd"]/div[@class="star"]/span[@class="rating_num"
]/text()').extract()[0]
                        quote =
eachMovie.xpath('div[@class="bd"]/p[@class="quote"]/span/text()').extract()
                        #quote可能为空，需要先进行判断
                        if quote:
                                quote = quote[0]
                        else:
                                quote = ''
                        item['title'] = fullTitle
                        item['movieInfo'] = ';'.join(movieInfo)
                        item['star'] = star
                        item['quote'] = quote
```

```
                        yield item
                nextLink =
selector.xpath('//span[@class="next"]/link/@href').extract()
                #第十页是最后一页，没有下一页的链接
                if nextLink:
                        nextLink = nextLink[0]
                        #print nextLink
                        yield Request(self.url +
nextLink,callback=self.parse)
```

运行`scrapy crawl douban`执行，爬虫执行完之后得到文件douban.csv，效果如下图：

表格 1

	movieInfo	title	star	quote
2	导演: 弗兰克·德拉邦特 Frank Darabont 主演: 蒂姆·罗宾斯 T 1994 / 美国 / 犯罪 剧情	肖申克的救赎 / The Shawshank Redemption / 月黑高飞(港) / 刺激	9.6	希望让人自由。
3	导演: 吕克·贝松 Luc Besson 主演: 让·雷诺 Jean Reno / 娜 1994 / 法国 / 剧情 动作 犯罪	这个杀手不太冷 / Léon / 杀手莱昂 / 终极追杀令(台)	9.4	怪蜀黍和小萝莉不得不说的故事。
4	导演: 陈凯歌 Kaige Chen 主演: 张国荣 Leslie Cheung / 张 1993 / 中国大陆 香港 / 剧情 爱情 同性	霸王别姬 / 再见，我的妾 / Farewell My Concubine	9.5	风华绝代。
5	导演: Robert Zemeckis 主演: Tom Hanks / Robin Wright F 1994 / 美国 / 剧情 爱情	阿甘正传 / Forrest Gump / 福雷斯特·冈普	9.4	一部美国近现代史。
6	导演: 罗伯托·贝尼尼 Roberto Benigni 主演: 罗伯托·贝尼尼 1997 / 意大利 / 剧情 喜剧 爱情 战争	美丽人生 / La vita è bella / 一个快乐的传说(港) / Life Is Beautiful	9.5	最美的谎言。
7	导演: 宫崎骏 Hayao Miyazaki 主演: 柊瑠美 Rumi Hîragi / 入 2001 / 日本 / 剧情 动画 奇幻	千与千寻 / 千と千尋の神隠し / 神隐少女(台) / Spirited Away	9.2	最好的宫崎骏，最好的久石让。
8	导演: 史蒂文·斯皮尔伯格 Steven Spielberg 主演: 连姆·尼森 1993 / 美国 / 剧情 历史 战争	辛德勒的名单 / Schindler's List / 舒特拉的名单(港) / 辛德勒名单	9.4	拯救一个人，就是拯救整个世界。
9	导演: 詹姆斯·卡梅隆 James Cameron 主演: 莱昂纳多·迪卡普 1997 / 美国 / 剧情 爱情 灾难	泰坦尼克号 / Titanic / 铁达尼号(港 / 台)	9.2	失去的才是永恒的。
10	导演: 克里斯托弗·诺兰 Christopher Nolan 主演: 莱昂纳多·迪 2010 / 美国 英国 / 剧情 动作 科幻 悬疑 冒险	盗梦空间 / Inception / 潜行凶间(港) / 全面启动(台)	9.2	诺兰给了我们一场无法盗取的梦。
11	导演: 安德鲁·斯坦顿 Andrew Stanton 主演: 本·贝尔特 Ben 2008 / 美国 / 喜剧 爱情 科幻 动画 冒险	机器人总动员 / WALL·E / 瓦力(台) / 太空奇兵·威E(港)	9.3	小瓦力，大人生。
12	导演: 朱塞佩·托纳多雷 Giuseppe Tornatore 主演: 蒂姆·罗斯 1998 / 意大利 / 剧情 音乐	海上钢琴师 / La leggenda del pianista sull'oceano / 声光伴我飞(港	9.2	每个人都要走一条自己坚定了的路，就算是粉身碎骨。
13	导演: 拉库马·希拉尼 Rajkumar Hirani 主演: 阿米尔·汗 Aami 2009 / 印度 / 剧情 喜剧 爱情 歌舞	三傻大闹宝莱坞 / 3 Idiots / 三个傻瓜(台) / 作死不离3兄弟(港)	9.1	英俊版憨豆，高情商版谢耳朵。
14	导演: 莱塞·霍尔斯道姆 Lasse Hallström 主演: 理查·基尔 Ric 2009 / 美国 英国 / 剧情	忠犬八公的故事 / Hachi: A Dog's Tale / 忠犬小八(台) / 秋田犬八千	9.2	永远都不能忘记你所爱的人。
15	导演: 克里斯托夫·巴拉蒂 Christophe Barratier 主演: 杰拉尔 2004 / 法国 瑞士 德国 / 剧情 音乐	放牛班的春天 / Les choristes / 歌声伴我心(港) / 唱诗班男孩	9.2	天籁一般的童声，是最接近上帝的存在。

项目样例的[代码地址]

思考和延伸：

- 将爬取的内容保存到数据库
- 数据分析（分析数据中的影片国家，上映年份，主演人员，影片类型）
- 如何做电影推荐?

学习掌握Redis实现将结果保存到MongoDB中，同时有很多的数据分析库可供使用，主流和常用的电影推荐策略比如按导演推荐（最近你看了《杀死比尔》，那么你可能还喜欢看昆汀其他的电影作品），按演员推荐（最近你看了一部李连杰主演的电影，那么你肯能还喜欢他的其他电影作品），按系列推荐（最近你看了星际迷航系列的一部电影，那么极有可能你会喜欢这一系列的其他影视作品），按类型推荐（最近你看了一部动作片，那么你可能还喜欢其他的高分动作电影）。

内容推荐和类型推荐的区别：假如你看动物世界里面一部有关狮子的视频，给你推荐其他团队拍摄的狮子的纪录片属于内容上的推荐（狮子作为影片内容主题未改变），给你推荐了老虎记录片则属于类型推荐（可能都是BBC拍摄的动物系列纪录片），另外门户最喜欢推送的时下热度高的视频内容，这在电影信息里面并不能体现，需要通过其他数据采集方案获取。

举个栗子，这是我在网上看到的一片实际应用，非官方版豆瓣电影可视化分析报告

模块发布

使用pip或easy_install管理和安装python的package包都是从pypi服务器中搜索和下载的，Pypi服务器允许我们将自己的代码也上传发布到服务器上，这样世界上的所有人都能使用pip或easy_install来下载使用我们的代码。

如果你已经编写了一个有用的库并想将它分享给他人，第一件事就是给它一个唯一的名字，并且清理它的目录结构。 例如，一个典型的函数库包会类似下面这样：

```
projectname/
    README.txt
    Doc/
        documentation.txt
    projectname/
        __init__.py
        foo.py
        bar.py
        utils/
            __init__.py
            spam.py
            grok.py
    examples/
        helloworld.py
        ...
```

要让你的包可以发布出去，首先你要编写一个 setup.py类似下面这样：

```
# setup.py
from distutils.core import setup

setup(name='projectname',
    version='1.0',
    author='Your Name',
    author_email='you@youraddress.com',
    url='http://www.you.com/projectname',
    packages=['projectname', 'projectname.utils'],
)
```

有关setup的详细格式参考文档,下一步就是创建一个 MANIFEST.in文件，列出所有在你的包中需要包含进来的非源码文件：

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

确保 setup.py和 MANIFEST.in文件放在你的包的最顶级目录中。 一旦你已经做了这些，你执行`python setup.py xxx`命令即可，其中xxx是打包格式的选项，如下：

```
python setup.py bdist_egg  # 生成easy_install支持的格式
python setup.py sdist      # 生成pip支持的格式
```

发布到pypi首先需要注册一个账号，然后进行如下两步：

- 注册package，输入`python setup.py register`
- 上传文件，输入`python setup.py sdist upload`

对于公司内部的项目，不方便放到外网上去，我们搭建的内网pypi源服务器，具有安全且拥有同样的舒适体验。

接下来学什么?

Python Web框架（Web2py，Flask，Django, Tornado）

游戏（pygame，cocos2d-python，Panda3D）

爬虫（scrapy, Beautiful Soup）

人工智能（NumPy, SciPy, tensorflow）

数据分析（Pandas，Matplotlib）

图形软件：PyQt PyGTK wxPython ...

Python 3.x ...

参考及说明

参考

本文大量引用《简明Python教程》一书

《python标准库》

《Python Network Programming Cookbook》

《Python Cookbook》第三版

极客学院Python爬虫入门系列

500lines

The Architecture of Open Source Applications

说明

示例运行于Python 2.7.10，Python 3.x版本可能有差异；

不保证所有内容的精确，可能会有错误和误导，水平有限只是做了些整理总结，望谅解 --!