函数

到目前为止，咱们写一些小程序是没有问题的了。但是我们经常会碰到一种情况，已经用程序实现好了的一个功能，希望在其它地方也能用到。这个时候，我们需要对这个功能进行抽象，进而加以命名定义，使其具有可复用性。尤其在写大型程序的时候，这种做法既灵活可靠又安全快捷，同时还可以使代码具有更高的可读性。这也就是函数的作用

定义

- 函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。
    - 函数能提高应用的模块性，和代码的重复利用率。其实我们已经接触过很多的Python提内建函数，比如print()。本章主要介绍如何自己创建函数，即自定义函数。
- 函数特性
    - 函数是一个可以重复调用的代码段
    - 函数可以有明确的输入和输出（当然也可以没有），从而实现一个功能
    - 函数要先定义后使用
    - 函数本身也可以作为参数
    - 函数可以被重复调用和声明，以最新一次申明为准

创建函数

语法：

```
def functionname( parameters ):

   ‘comment’

   function_body

   return [expression]
```

- 任何传入参数和自变量必须放在圆括号中间。参数的个数没有限制。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- Return[expression]结束函数，选择性地返回一个值给调用方。函数一旦return，后面的代码块将不再起作用。也就是说，除非你需要在函数中间return，否则return最好写在函数的

最后，这样容易避免很多问题

- 不带表达式的return相当于返回 None

```
>>> def hello():
...     print 'hello'
...     print 'world'
...
>>> hello()
hello
world
>>> hello()
hello
world
>>> def hello():
...     return 'hello world'
...
>>> print hello()
hello world
>>> print hello()
hello world
>>>
```

参数

函数通过它的参数来获取他需要的一系列值。参数本身可以理解为变量。函数调用的时候，我们就是在给这些变量赋值

我们在函数声明中参数变量通常叫做形参，而调用函数的时候提供的值是实参

```
>>> def hello(text):
...   print 'hello %s' % text
...
>>> hello('wd')
hello wd
>>> hello('pc')
hello pc
>>>
```

- 对于字符串（以及数字和元组）这种不可改变的参数，其本身是不会被改变的（也就是说只能用新的值覆盖）。所以在函数调用的时候，函数内为参数赋值不会影响这个参数本身的值

```
para = ‘subin’

def change(para):

…     para = ‘pc’

…     print para

…

change(para)

pc

print para

subin
```

可以看到，调用change函数的时候，把para赋值为pc并没有改变para本身的值

- 对于可以改变的数据结构（如list和dict），函数内为参数赋值则会影响这个参数本身的值。可以认为是一种引用传递

```
>>> listPara = ['subin', 'wd']
>>> def change(para):
...     para[1] = 'woniu'
...     print para
...
>>> change(listPara)
['subin', 'woniu']
>>> print listPara
['subin', 'woniu']
```

很明显，listPara的值已经被改变了（第二个值被重新赋值）。如果不想改变dict或者list本身，那么需要首先复制，然后将复制出来的值传入函数

```
listPara = [‘subin’, ‘wd’]

listPara1 = listPara[:]

def change(para):

…     para[1] = ‘woniu’

…     print para

…

change(listPara1)

[‘subin’, ‘woniu’]

print listPara

[‘subin’, ‘wd’]
```

位置参数与关键字参数和默认值

当我们定义的函数，有多个参数的时候（实际上这种情况经常遇到），情况可能会变得稍微有些复杂，比如我怎么知道我传入的实参赋给了正确的形参呢？

- 我们的传入参数个数，一定要和函数声明的参数个数相同
- 如果传入参数仅仅是具体的值，参数的位置是有意义的，即传入参数是按照位置赋值给对应函数参数的，跟参数名无关，这个叫位置参数

```
>>> def hello_world(name, word):
...     print '%s,%s' % (name,word)
...
>>> hello_world('wd','hello')
wd,hello
>>>
>>> def hello_world(word,name):
...     print '%s,%s' % (name,word)
...
>>> hello_world('wd','hello')
hello,wd
>>>
```

python提供了很好的办法可以忽略参数的位置，这在其它语言中不常见。是个很好的特性

- 我们在做函数调用的时候，如果指定参数赋值调用，则位置参数不再起作用，这个就是关键字参数。

```
def hello_world1(name,word):
…     print ‘%s,%s’ % (name,word)
…
def hello_world2(word,name):
…     print ‘%s,%s’ % (name,word)
…
hello_world1(name=’wd’,word=’hello’)
wd,hello
hello_world2(name=’wd’,word=’hello’)
wd,hello
```

可以灵活的实现参数赋值，而不用考虑参数本身在函数声明时候的顺序。也就是说，关键字参数调用，与参数本身的位置无关

- 在声明函数的时候，可以对参数进行赋值，这个时候，可以认为它是参数的默认值，这也是为了函数调用的方便

- 在设置默认值参数的时候，有默认值的参数应该放在后面
- 对于默认值参数来说，如果在调用的时候，有传入实参，则默认值被替换，否则使用这个默认值

```
>>> def hello_world(name='wd',word='hello'):
...     print '%s,%s' % (name,word)
...
>>> hello_world()
wd,hello
>>> hello_world('pc')
pc,hello
>>> hello_world('pc','hehe')
pc,hehe
>>> def hello_world(name='wd',word):
...     print '%s,%s' % (name,word)
...
  File "", line 1
SyntaxError: non-default argument follows default argument
```

可以看到，后面那个函数定义被报错了，也就是我们上面说的第二条

收集参数

- 可以使用*来实现任意数量的参数输入，这里叫收集参数。
- 这个输入的tuple，最好是没有嵌套的。如果有的话，有些运算可能没办法支持（因为对于这种参数，一般程序都是通过迭代处理的）

```
>>> def print_params(*params):
...     print params
...
>>> print_params()
()
print_params(1,2,3,4)>>> print_params(1,2,3,4)
(1, 2, 3, 4)
>>>
```

- 收集参数可以和位置参数混用，但是收集参数必须放在后面，意思说收集其余位置的参数
- *收集参数不能和关键字参数混用

```
>>> def print_params(title, *params):
...     print title
...     print params
...
>>> print_params('params:', 1, 2, 3)
params:
(1, 2, 3)
>>>
```

- 我们还可以用**来传递一个dict

```
>>> def print_params(**params):
...     print params
...
>>> print_params(age = 20)
{'age': 20}
>>> print_params(name = 'subin', age = 20)
{'name': 'subin', 'age': 20}
>>>
```

- 另外一个方面，收集参数不好的地方就是带来了输入参数的不确定性，程序需要处理。尤其是在位置参数和收集参数混合使用的时候
  - 用tuple或者dict变量去做* 或者 收集参数的话，调用的时候，参数前一定要加上 * 或，否则会被认为是单个值
  - 和位置参数一样，也可以先定义确认的参数，不确定的用 收集。 同时这里面的位置参数可以指定默认值，可以和关键字参数混用，但是可能会出现重复定义参数的问题

感受一下下面的例子

```
def print_params1(name = ‘subin’, *params):
…     print name
…     print params
…
print_params1()
subin
()
```

```
print_params1('wd')

wd

()

print_params1('wd', 1)

wd

(1,)

print_params1('wd', 1, 2)

wd

(1, 2)

print_params1(name = 'wd', 1, 2)

  File "", line 1

SyntaxError: non-keyword arg after keyword arg


def print_params(name, **params):

…     print name

…     print params

…

print_params(name = 'wd', age = 20)

wd

{'age': 20}

print_params(name = 'wd', age = 20, work = 'teacher')

wd

{'age': 20, 'work': 'teacher'}

print_params('wd', age = 20, work = 'teacher')

wd

{'age': 20, 'work': 'teacher'}
```

```
print_params(‘wd’, age = 20, work = ‘teacher’, name = ‘subin’)

Traceback (most recent call last):

  File ““, line 1, in

TypeError: print_params() got multiple values for keyword argument
‘name’

print_params(name = ‘wd’, age = 20, work = ‘teacher’, name = ‘subin’)

  File ““, line 1

SyntaxError: keyword argument repeated
```

参数作用域

参数的作用域即参数的可以见范围

- 一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。
- 变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。对于函数来说两种最基本的变量作用域如下：
  - 全局变量（需要用global声明）
  - 局部变量

```
x = 1

def change_global():

…     x = 2

…

change_global()

print x

1
```

可以看到，change_global()函数并没有影响函数体外面x的值，也就是说，change_global()函数内部对x的定义可见性仅限于函数内部。但是下面的例子就不一样了：

```
>>> x = 1
>>> def change_global():
...     global x
...     x = 2
...
>>> change_global()
>>> print x
2
```

这个函数定义和上面不同的地方在于，函数内部对x进行操作之前声明了这个变量是global的。这个时候，就可以影响函数体外定义的x值了，因为这两个x，是同一个x

- 函数的参数只能是局部变量（对于不可变元素），也就是说，对和函数的参数同名的传入参数，如果声明了全局变量，则会报错

```
def change_global(x):

…     global x

…     x = 2

…

  File ““, line 1

SyntaxError: name ‘x’ is local and global
```

理解变量的作用域非常重要,但同时我们也建议，慎用全局变量

特殊函数

函数本身也可以作为参数传递给函数，这里说的函数是函数本身，而不是函数return值。有了这个概念，我们夏明就可以介绍几个特殊的函数

- 递归函数

递归函数，可以认为是一个循环在调用自己的函数，但是一定要有一个终止条件。这种函数在数学计算中使用的比较多。基本定义可以为：

```
def recursion():
    return recursion()
```

大家可以尝试用这种方法去计算阶乘或者等差数列的和，非常方便

对下面介绍的函数，大家可以了解清楚其作用和使用原理，灵活使用他们很多时候会起到事半功倍的效果

- Map函数(映射)
  - 原型：map(function, sequence)，作用是将一个列表映射到另一个列表，
  - 使用方法：

```
def f(x):
    return x**2
l = range(1,10)
map(f,l)
Out[3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - 实现：

```
def map(func,seq):
    mapped_seq = []
    for eachItem in seq:
        mapped_seq.append(func(eachItem))
    return mapped_seq
```

- Reduce函数
  - 原型：reduce(function, sequence, startValue)，作用是将一个列表归纳为一个输出
  - 使用方法：

```
def f2(x,y):
    return x+y
reduce(f2,l)
Out[7]: 45
reduce(f2,l,10)
Out[8]: 55
```

  - 实现：

```
def reduce(bin_func, seq, initial=None):
    lseq = list(seq)
    if initial is None:
        res = lseq.pop(0)
    else:
        res = initial
    for eachItem in lseq:
        res = bin_func(res,eachItem)
    return res
```

- Filter函数
  - 原型：filter(function, sequence)，作用是按照所定义的函数过滤掉列表中的一些元素，

- 使用方法：

```
def f2(x):
    return x%2 != 0
filter(f2,l)
Out[5]: [1, 3, 5, 7, 9]
```

记住：这里的function必须返回布尔值。

- 实现

```
def filter(bool_func,seq):
    filtered_seq = []
    for eachItem in seq:
        if bool_func(eachItem):
            filtered_seq.append(eachItem)
    return filtered_seq
```

- Lambda函数（匿名函数）
  - 原型：lambda <参数>: 函数体，隐函数，定义一些简单的操作，
  - 使用方法：

```
f3 = lambda x: x**2

f3(2)

Out[10]: 4
```

还可以结合map、reduce、filter来使用，如：

```
map(f3,l)

Out[11]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- 列表推导式
  - 基本形式：[expression for item in sequence], 这里x表示对item的操作，
  - 使用方法：

```
[i**2 for i in l]

Out[12]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```