第5章 chapter 5

函　数

函数可以视为由语句构成的、能完成某种功能的程序段。函数内的代码只有调用时才被运行。函数定义时可以指定要传入的参数为形式参数，即形参；调用函数时对应地传入参数，称为实参。函数的参数要放在圆括号内。函数运行结束后返回调用者，并能返回设定的结果。前面几章中已经介绍了一些 Python 的内置(builtin)函数，如求绝对值的 abs()、打开文件的 open()等，还有一些和内置对象有关的函数，如列表对象的 sort、append 方法、字符串的分割 split、大小写转换等。当然用户也可以自己定义函数。函数是可以多次调用的程序段，使用函数可以避免重复代码编写，还可将复杂任务分解，并提高代码的重用性。本章主要介绍 Python 中函数的定义和调用方法、变量的作用域和函数参数传递等问题。

学习目标

- 掌握函数定义的方法；
- 熟悉函数参数的传递方法，调用函数时的参数分配情况；
- 了解函数变量的作用域，区分局部变量和全局变量；
- 学习匿名函数 lambda 的使用。

5.1　函数的定义和调用

用户自定义函数使用 def 语句，def 语句可创建一个函数对象，并将其赋值给指定的函数名。def 语句的一般格式如下：

```
def 函数名 (形参 1,形参 2,…,形参 n):
    函数内的语句块
    [return 返回值]
```

def 语句首行包括函数名和形参表，以“:”结尾。函数名根据变量命名方法和编程规范，通常用字母的小写形式，可以包括字母、数字和下划线等，如 test_func1、factorical 等都是合法的函数名。形参即形式参数，是定义函数时需要的参数名，函数调用时形参的位置将被实际参数替代。根据情况，形参可以是 0 个或多个，多个形参之间用逗号分开。

注意：函数内部的语句块和 def 语句要有一定的缩进。函数主体可以包括 return 语句，return 后面是函数的返回值列表，表示函数结束并将这些值返回调用者。一个函数也可以没有返回值和 return 语句，这样系统会自动给调用者返回一个 None 对象。return 语句可以出现在函数主体的任何位置，不一定在函数的最后。

调用函数的格式如下：

函数名(实参表)

调用时传入的实参将替代函数定义时的形参。实参的个数、位置都应该和定义时的形参一一对应。即使函数没有形参，调用时也不能将括号缺省。函数调用可以作为语句，如果函数有返回值也可以作为表达式的一部分，甚至可以作为另一个函数的实参出现。

不过，一个函数在调用前一定要先有定义；否则出现运行错误。下面给出在交互环境下，一个简单的函数定义和调用使用的例子。

```
>>>def square(x):              #x 为形参
       return x * x

>>>square(5)                   #函数调用,出入实参
25
>>>10-square(3)                #函数调用作为表达式的一部分
1
>>>print(square(-3.2) )        #函数调用作为另一个函数的实参
10.240000000000002
>>>square('a')

Traceback (most recent call last):
  File "<pyshell#21>", line 1, in<module>
    square('a')
  File "<pyshell#18>", line 2, in square
    return x * x
TypeError: can't multiply sequence by non-int of type 'str'
```

例子中的函数是一个实现平方运算的函数，但是定义函数时并没有对参数的类型予以限制，可以传入整数，也可以是浮点数。两次调用尽管传入的实参类型不同，但函数都能正常进行平方运算，是因为乘法运算链接的可以是整数，也可以是浮点数，而乘法的结果取决于传入参数的类型，因此返回不同类型的结果。这种依赖类型的行为也就是所谓的多态。但如果给函数传入的是一个字符串，因为 Python 内置的乘法运算不能实现两个字符串相乘，因此函数调用时出现异常并结束。可见，实参和形参一定要一致，这里的一致包括了类型一致和位置对应。当然假如通过运算符重载定义了字符串乘的操作就不会出现异常。关于多态和运算符重载的知识将在面向对象程序设计(OOP)一章中介绍。

定义函数时同时给出函数的说明文档是个好的设计习惯，说明文档用于描述函数的

功能和使用方式。说明文档就是字符串常量，可以直接在 def 语句之后写，它将作为函数的一部分进行存储。当文档比较长时，可用三引号括起来，以便多行书写。使用 help 查询这个函数功能时就能显示这部分说明文字，或者直接调用函数的__doc__方法也可以显示说明文字。

```
>>>def square(x):
        'Calculats the square of a number x.'
        return x * x

>>>help(square)
Help on function square in module __main__:

square(x)
    Calculats the square of a number x.
>>>print  square.__doc__
Help on function square in module __main__:

square(x)
    Calculats the square of a number x.
```

5.2 参数传递

通过上面的例子已经初步认识了定义函数的形参和调用函数的实参之间的关系。如果把函数看做是完成某个任务的机器，那么参数就是给机器送入的要加工的材料，返回值则可以看做加工后的结果，当然不是所有函数都要求有返回值。本节讨论如何给函数传递参数。

5.2.1 参数传递的两种模式

Python 中参数的传递模式和其他高级语言很相似，也有两种情况：如果参数是不可变类型，如数值、字符串、元组等，由于参数的不可改变特性，实际需要创建一份参数的副本再传递，这一点相当于通过值传递参数，即传值；如果参数是可变类型，如列表、字典等，这些参数是可以原地修改的，函数对于这样的参数，实际传入的是对象的引用，也就是在函数中如果修改了这些对象，调用者中的原始对象也将受到影响，可见，这种参数传递方法相当于通过指针传递参数，传递的是引用，又称传址。

```
>>>def unchange(unch):
      unch+=1
      print unch

>>>unchange(110)        #赋值数值传入
    111
```

```
>>>def changed(chang):
      chang[0]+=1
      print chang

>>>changed([0,2,4])   #传入列表的引用
    [1, 2, 4]
```

通过引用进行参数传递使得不必创建多个参数的副本实现参数的更新。有时如果不希望可变参数在原地被修改，此时可以创建一个明确的对象副本传递给函数。仍然使用上例中 changed 函数的定义，但本次是将列表的一个副本传入，因此不会影响原来的列表。

```
>>>L=[-1,0,1]
>>>changed(L[:])   #明确地创建一个列表的副本作为传入参数,L 就不受影响
[0, 0, 1]
>>>L
[-1, 0, 1]
```

当给函数传递参数的引用时还应格外注意，如果在函数内部将可变类型的形参赋值给其他变量，对其他变量的修改依然会影响到引用对象。在上面的函数 changed 中，如果将传入的参数赋值给新的局部变量，尽管没有针对形参的具体操作，但是对新变量的修改依然影响原来的参数。

```
>>>def changed(chang):
       inner_var=chang
       inner_var[0]+=1
       print inner_var

>>>changed(L)
    [0, 0, 1]
```

inner_var 是函数内部定义的变量，称为局部变量，它只在该函数内有效。关于局部变量和全局变量的讨论可参见 5.3 节。

5.2.2 参数的匹配

调用函数时将实际参数传入，传入的参数要和定义函数时的参数进行匹配。Python 中有两种参数匹配方式：一种是最常用的位置参数，也就是根据参数的先后次序将实参和形参进行匹配；另一种为关键字参数，是根据参数名进行匹配的，通过 name＝value 的形式向函数传入参数，关键字参数对参数的次序没有要求。

1. 位置参数的匹配

位置参数要求函数调用时传入的实参和函数定义时的形参在位置上一一对应，默认情况下是根据位置从左到右进行严格匹配。如果出现多余的数据或者有的形参未被赋

值，就会触发类型异常。

```
>>>def max3(x,y,z):          #一个含有 3 个形参的函数
       if x>y>z or x>z>y:
           return x
       elif y>x>z or y>z>x:
            return y
       else:
            return z

>>>max3(1,3,4)               #调用函数，也传入 3 个值，分别为 3 个形参传值。正常匹配
   4
>>>max3(-2,0)                #传入值的数目少于形参数目，调用函数出现异常

   Traceback (most recent call last):
     File "<pyshell#51>", line 1, in<module>
       max3(-2,0)
   TypeError: max3() takes exactly 3 arguments (2 given)
>>>max3(-1,0,2,4)            #传入值多于形参，调用函数也出现异常

   Traceback (most recent call last):
     File "<pyshell#52>", line 1, in<module>
       max3(-1,0,2,4)
   TypeError: max3() takes exactly 3 arguments (4 given)
>>>var=[6,7,8]
>>>max3(*var)                #序列解包传递参数
   8
```

可见，位置参数要求函数调用者熟悉函数定义时的形参含义和次序，不仅参数的数目要相同，还要注意参数的次序问题，防止出错。因此定义函数时，往往需要对函数的参数含义、数目和类型等加以说明，这是函数文档的重要组成内容。当函数参数比较多时，对函数调用者而言，要准确匹配这些位置参数不是很容易，不留意就会出错。

调用时传递位置参数还可通过解包序列实现。如果实参是序列对象，使用前加 * 的变量就可以对序列解包后传入。不过解包序列也应和位置参数对应才能匹配。上面最后一个例子就是解包一个列表实现参数匹配的。

除了这种常规的根据位置的匹配方法实现实参和形参的匹配，Python 还提供了关键字匹配的方法。Python 3.0 版后的参数匹配方法变得更加灵活和丰富。

2. 关键字参数匹配和参数的默认值

函数参数都有一定的意义，为了明确参数的作用，可以为参数命名，而带有参数名的参数称为关键字参数。关键字形式传入函数的参数，可以让调用者不必关心参数的次序问题，通过关键字为参数赋值即可。在函数定义和调用时，都要使用形式关键字命名方式 name=value。参数传递时，系统通过关键字进行匹配而不是参数的位置。通过下面的例子来认识一下关键字参数。

函数 person 有 4 个形参，每个参数在定义时利用 name＝value 的形式给出了该参数的默认值。当函数调用时，如果该关键字参数没有传入值，则使用定义时给定的这个值作为默认值。调用函数时，传入参数也要求用 name＝value 的形式，即为关键字参数传入新的值，新的值将覆盖函数定义时的默认值。关键字参数使得函数参数的意义更为明确，尤其适合函数参数比较多的情况。

```
>>>def persons(name='xxx',gender='F',hobby='music',age=30):    #定义函数
        print 'The information is:',name,gender,age,hobby

>>>persons(name='Jenny',age=25,hobby='bike') #调用函数,对关键字参数次序没有要求
The information is: Jenny F 25 bike
>>>persons()                          #调用时不传入参数,使用函数定义时的默认值
The information is: xxx F 30 music
>>>person_data={'name':'Lee', 'age':39,'gender':'F','hobby':'Boxing'}
>>>persons(**person_data)      #利用字典解包传递关键字参数
The information is: Lee F 39 Boxing
```

最后一个例子中利用了字典解包的方式传递关键字参数，字典变量前要加 * * ，字典的关键字要和函数的形参关键字对应；否则无法通过解包传入关键字参数。

关键字参数在函数定义时如果没有初始值，可以设为空。但是需要注意，关键字参数的默认值只能使用一次，如果在函数中对关键字参数做了改变，尤其是可变类型的对象，要注意函数调用后对参数的影响。来看下面这个例子：L 作为一个关键字参数，函数定义时默认为[]，但是在函数内部对其进行了修改。调用函数时尽管都使用了 L 的默认值，但由于 L 被函数原地修改了，因此再次调用时如果默认 L 的值，L 将不再是函数定义时的默认值了。

```
>>>def f(a,L=[]):
       L.append(a)
       return L

>>>print f(1)
[1]
>>>print f(2)
[1, 2]
>>>print f(3)
[1, 2, 3]
```

如果希望在后面的多次调用中关键字参数的默认值依然有效，可以通过添加条件测试语句。例如，上面的例子，函数可以定义如下：

```
>>>def f(a,L=None):
       if L is None:
          L=[]
          L.append(a)
```

```
        return L

>>>print f(1)
[1]
>>>print f(2)
[2]
>>>print f(3)
[3]
```

关键字参数清晰地给出了参数名和值的对应，因此实际调用时即使参数次序发生了改变也不受影响。

实际应用中，函数的关键字参数可以和位置参数联合使用，但是位置参数要放在关键字参数前；否则系统报语法错误。参数匹配时，系统首先做位置关系的配对，然后再根据关键字匹配参数。关键字参数方式另一个优点是可以给出参数的默认值，正如例子显示的那样，定义函数时为关键字参数赋值。当调用函数时，如果该参数缺省，就自动使用函数定义时的值作为默认值。下面是联合使用位置参数和关键字参数的例子。

```
>>>def persons(name,hobby,age, gender='F'):
       print 'The information is:',name,gender,age,hobby

>>>persons('Lily','music',20,gender='M')
The information is: Lily M 20 music
>>>persons('Wu','chess',40)      #关键字参数没有传入值,使用默认值
The information is: Wu F 40 chess
```

3. 参数收集

参数收集机制在函数定义时提供了更灵活的参数匹配方案。定义时如果不能确定参数的数目，可以利用参数收集方式定义形参。针对位置参数和关键字参数，收集分两种。第一种，以 * 开头的参数名可以收集位置不匹配的参数，收集的参数形成一个包含位置信息的元组。 * 的含义就是收集多余的位置参数。如果是关键字参数形式，则使用**开头的变量收集额外的关键字参数，也就是第二种收集方法。使用**开头的参数收集的结果是一个字典变量，其中字典的键就是关键字参数名，值就是关键字的参数值。

```
>>>def f(*args):       #*args 为位置参数收集
       print args

>>>f()
()
>>>f(0)                #传入一个值,收集的元组包含一个元素
(0,)
>>>f(1,2,3)            #传入多个值,收集的元组包含多个元素
(1, 2, 3)
>>>def ff(**args):     #关键字参数收集
```

```
        print args

>>>ff()
{}
>>>ff(a=1,b=2)          #关键字参数收集得到一个字典
{'a': 1, 'b': 2}
```

在定义函数时，位置参数、关键字参数和参数收集策略可以混合使用，以应对函数复杂的参数情况：比如，在只能确定部分的位置参数或不知道参数名称的时候，就可以混合使用这些方式。需注意的是，如果混合使用，位置参数、关键字参数和参数收集有排序的规则，要求位置参数排在最前，之后是位置参数收集或关键字参数，最后是关键字参数收集，这样系统能够明确参数的对应关系，传递参数时才能正确匹配。下面是函数参数中各种方式混合的例子，注意函数定义时的形参写法以及函数调用时实参写法，最后是形成的匹配关系。

```
>>>def f(a,b=0,*c):        #位置参数,关键字参数和位置参数收集混合
        print 'a=',a,'b=',b,'c=',c

>>>f(9,-1,0,1)             #调用时都是位置参数的形式,匹配按次序进行,剩余的被收集
a=9 b=-1 c=(0, 1)

>>>def f(a,b=0,**c):       #位置参数,关键字参数和关键字参数收集
        print 'a=',a,'b=',b,'c=',c

>>>f(1,2,e=5,f=8)          #首先位置参数匹配,然后是关键字参数,最后被关键字收集
a=1 b=2 c={'e': 5, 'f': 8}
```

Python 3.0 版后这种混合策略更灵活，其他复杂参数传递情况可参见 Python 手册的说明。

5.3 变量的作用域

一个函数中可以定义变量和函数，这些名称在函数内部定义，它们的影响范围和在函数外定义的变量是不同的。函数内部的变量称为局部变量或本地(local)变量，局部变量的作用域在函数内部，在函数外无法使用。在一个模块文件中、所有函数外定义的变量称为全局(global)变量，全局变量的作用域是整个模块，也就是说，在该模块所有函数外和函数内都可以使用全局变量。先通过下面的例子初步认识变量的作用域的概念。该例是在交互环境下实现的。

```
>>>X=99                   #函数外定义,全局变量
>>>def func():
        X=0               #函数内定义,局部变量
        print (X)         #函数内部首先使用局部变量
```

```
>>>func()
0
>>>print (X)              #在函数外使用的是全局变量
99
>>>def f():
      print (X)           #全局变量可在函数内使用

>>>f()
99
```

在第一个函数 func 外定义了变量 X，在函数 func 内部也定义了一个同名的 X，func 内部的 print 函数使用了 X。当 func 函数调用时，内部的 print 打印出的是函数内部 X 的值，而不是外部的 X。说明内部的 X 覆盖了外部的 X。因为函数在调用时生成了自己内部变量构成的一个命名空间，遇到变量名 X 时，print 函数首先在函数自己的命名空间内查找 X 变量，如果有则首先使用函数内部定义的变量，而不管函数外部有无该变量名称。换句话说，函数内部变量和外部变量的名字并不会产生冲突，如果有同名的变量，函数内部变量将覆盖外部的同名变量。当函数调用结束时，该函数的命名空间也被释放了。因此，在交互环境下再次 print X 时，打印的是函数外部的 X 了。这就是说，函数内部的局部变量的作用域仅在函数内部，这也是将其称为局部变量的原因。

第二个例子定义了 f 函数，f 函数内部没有定义 X 变量，调用 f 执行 print X 时，由于 X 不在函数的命名空间中，就到函数外部找 X。因此，打印了函数外部定义的全局变量 X。也就是说，全局变量的作用域是整个模块。

通过上面的例子可以看出，局部变量和全局变量有不同的作用域，函数调用时动态地形成了由局部变量构成的命名空间，可以和函数外部的变量区分开。

在赋值语句部分已经介绍过，Python 的变量使用前无须事先声明，只要赋值之后就可以使用，赋值的同时就定义了变量。因此，变量赋值的位置决定了变量的性质及作用域范围。变量赋值的位置有两种：一种是在模块（也包括交互环境下）中定义的变量；另一种在函数内部或表达式内部定义的。模块中定义的变量作用域是在整个模块，而函数或表达式中定义的仅限于函数及表达式内部使用，不能被函数和表达式之外引用。再通过下面这个例子具体说明哪些变量是局部变量。

```
def intersect(seq1,seq2):
    res=[]
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

例子中局部变量包括：

- 函数内部通过赋值语句显式赋值的变量，如 res。
- 作为形参传入的变量，如 seq1 和 seq2。

• for 循环结构将元素值赋给的变量，如 x。

变量的作用域严格地区分了函数内部和外部的空间。变量的不同作用域有助于函数内变量的本地化，使得函数能够独立于调用者。下面是 Python 的变量作用域的一般法则。

(1) 模块文件中定义的变量是全局变量，其作用范围为全部模块代码空间。模块是分隔变量的顶层结构，一个模块的变量需要正确导入该模块后才能够执行和使用。

(2) 函数内部变量默认为局部变量，作用域是函数内部，在函数外无效。在每次函数调用时创建局部变量的命名空间。如果想在函数内定义一个全局变量，需要明确声明，即通过 global 语句来说明变量是全局的。全局变量在声明它的函数外也可以使用。也就是在函数内定义的全局变量作用域也是整个模块。

```
>>>X=88
>>>def func():
        global X
        X=100

>>>func()
>>>X          #在函数中对全局变量进行修改
100
```

(3) 原地修改变量不会改变变量的作用域，但是对变量名重新赋值可以。例如，有个全局列表 L，在函数内部调用增加元素的方法 L.append()，并不能把 L 变为函数的局部变量，L 仍然是全局变量。但是如果给 L 赋予一个新值 L=X，这时的 L 就变为在函数内定义的新的局部变量了。

Python 遇到一个未知函数变量时，变量名解析的次序符合 LEGB 法则。LEGB 分别代表了 4 个不同的作用域：首先查看变量是否在本地作用域(L)中，如果不在，再看是否存在嵌套函数，如果存在就到上一层函数作用域中找(E)，如果没有，之后是到全局作用域(G)中找，最后是查找 Python 内置作用域(B)，内置作用域包括内置的函数 print、zip 等。

一般函数和外界沟通的方式是传入参数和返回值，而全局变量的作用域是整个模块，如果函数内部使用全局变量，就相当于增大了函数和外界的联系，从而破坏了函数的独立性，因此设计中不建议在函数中过多地使用全局变量。

```
>>>X=99
>>>def func(Y):
        Z=X+Y
        return Z

>>>func(1)
100
```

在上面这个例子中，全局变量是 X 和函数 func，Y 和 Z 是局部变量。函数内部使用