第 5 章　函数设计与使用

在实际开发中，有很多操作是完全相同或者是非常相似的，仅仅是要处理的数据不同而已，因此，经常会在不同的位置多次执行相似甚至完全相同的代码块。从软件设计和代码复用的角度来讲，直接将该代码块复制到多个相应的位置然后进行简单修改绝对不是一个好主意。虽然这样使得多份复制的代码可以彼此独立地进行修改，但这样不仅增加了代码量，使得程序文件变大，也增加了代码理解和代码维护的难度，更重要的是为代码测试和纠错带来很大的困难。一旦被复制的代码块在将来某天被发现存在问题需要修改，必须对所有的复制都做同样正确的修改，这在实际中是很难完成的一项任务。由于代码量的大幅增加，导致代码之间的关系更加复杂，很可能在修补旧漏洞的同时又引入新错误。因此，应尽量避免使用直接复制代码块的方式来实现复用。

解决上述问题一种常用的方式是设计和编写函数；另一种是设计和编写面向对象程序设计中的类。本章介绍函数设计与使用，第 6 章介绍面向对象程序设计。将可能需要反复执行的代码封装为函数，并在需要执行该段代码功能的地方进行调用，不仅可以实现代码的复用，更重要的是可以保证代码的一致性，只需要修改该函数代码则所有调用位置均得到体现。当然，在实际开发中，需要对函数进行良好的设计和优化才能充分发挥其优势。在编写函数时，有很多原则需要参考和遵守，例如，不要在同一个函数中执行太多的功能，尽量只让其完成一个高度相关且大小合适的功能，以提高模块的内聚性。另外，尽量减少不同函数之间的隐式耦合，例如，减少全局变量的使用，使得函数之间仅通过调用和参数传递来显式体现其相互关系。

在编写函数时，函数体中代码的编写与前面章节介绍的内容基本一样，只是对代码进行了封装并增加了函数调用、传递参数、返回计算结果等外围接口，这也正是本章讲解的重点。由于 Python 程序是解释执行的，如果编写的函数或代码有问题，只有在被调用和执行时才可能被发现，甚至包括某些语法错误。另外，还有可能传递某些类型的参数时执行正确，而传递另一些类型的参数时却出现错误。出现这样的情况有多种可能的原因，例如，不同的参数值可能会使得函数执行不同的路径，或者不同的参数类型所支持的操作和运算符不同，等等。所以，在进行代码测试时一定要注意，一次或几次运行正常并不表示编写的代码没有问题，必须进行尽可能完全的测试，尽量满足各种覆盖性要求，对所有的执行路径都要测试，在代码发布之前发现和解决更多的潜在问题。

5.1 函数定义与调用

在 Python 中，定义函数的语法如下：

```
def 函数名([形参列表]):
    '''注释'''
    函数体
```

在 Python 中首先使用 def 关键字定义函数，然后是一个空格和函数名，接下来是一对圆括号，在圆括号内是形参列表，如果有多个参数则使用逗号分隔，圆括号之后是一个冒号和换行，最后是必要的注释和函数体代码。定义函数时需要注意以下事项。

（1）函数形参不需要声明其类型，也不需要指定函数返回值类型，解释器会自动推断。

（2）即使该函数不需要接收任何参数，定义和调用时也必须保留一对空的圆括号。

（3）圆括号后面的冒号必不可少。

（4）函数体相对于 def 关键字必须保持一定的空格缩进。

（5）Python 允许嵌套定义函数，可参见 5.8 节的讨论。

例如，下面的函数用来计算斐波那契数列中小于参数 n 的所有值：

```
def fib(n):                                    #n 为形参
    a, b = 1, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

该函数的调用方式为

```
fib(1000)                                      #1000 为实参
```

在定义函数时，开头部分的注释并不是必需的，但是如果为函数的定义加上一段注释，可以为用户提供友好的提示和使用帮助。例如，把上面生成斐波那契数列的函数定义修改为下面的形式，在函数开头加上一段注释。在调用该函数时，输入左侧圆括号，立刻会得到该函数的使用说明，如图 5-1 所示。

```
>>> def fib(n):
    '''accept an integer n.
       return the numbers less than n in Fibonacci sequence.'''
    a, b = 1, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()


>>> fib(
(n)
accept an integer n.
return the numbers less than n in Fibonacci sequence.
```

图 5-1 使用注释为用户提示函数使用说明

在定义函数时，可以声明形参类型和函数返回值类型，但并不真正约束和检查，可以接收任意类型的实参，也可以返回任意类型的值。例如：

```
def func(i:int, j:int) -> int:
    return i + j
print(func(3, 5))
```

如果在函数体中有调用该函数自身的代码，这称作递归函数。在编写递归函数时，应保证每次递归时问题性质不变但规模越来越小，并且当问题规模小到一定程度时可以直接解决而不需要继续递归。例 5-17 和 7.5 节遍历目录树的代码演示了递归函数的用法。

5.2 形参与实参

函数定义时圆括号内是使用逗号分隔的形参（parameters）列表，一个函数可以没有形参，但是定义和调用时一对圆括号必须有，表示这是一个函数并且不接收参数。函数调用时向其传递实参（arguments），将实参的引用传递给形参。

在定义函数时，对参数个数并没有限制，如果有多个形参，则需要使用逗号进行分隔。例如，下面的函数用来接收两个参数，输出其中的最大值，模拟内置函数 max()的功能。

```
def printMax(a, b):
    if a >= b:
        print(a)
    else:
        print(b)
```

当然，这里只是为了演示，而忽略了一些细节，如果输入的参数不支持比较运算，则会出错，可以参考第 8 章中介绍的异常处理结构来解决这个问题。


5.2

在函数内直接修改形参的值不会影响实参。例如：

```
>>> def addOne(a):
    print(a)
    a += 1                              #这里实际是修改了形参 a 的引用,a=a + 1
    print(a)
>>> a = 3
>>> addOne(a)
3
4
>>> a                                   #实参的值和引用不变
3
```

从运行结果可以看出，在函数内修改了形参 a 的值，但是当函数运行结束以后，实参 a 的值并没有被修改，可以参考 5.5 节中关于变量作用域的讨论。在有些情况下，可以通过一定的方式在函数内修改实参的值，例如下面的代码：

```
>>> def modify(v):                     #使用下标修改列表元素值
    v[0] = v[0] + 1                     #没有修改形参的引用
>>> a = [2]
>>> modify(a)
>>> a
[3]
>>> def modify(v, item):               #使用原地操作的 append()方法为列表增加元素
    v.append(item)                      #没有修改形参的引用
```

```
>>> a = [2]
>>> modify(a, 3)
>>> a
[2, 3]
>>> def modify(d):                          #使用下标修改字典元素值或为字典增加元素
    d['age'] = 38                           #没有修改形参的引用
>>> a = {'name':'Dong', 'age':37, 'sex':'Male'}
>>> modify(a)
>>> a
{'age': 38, 'name': 'Dong', 'sex': 'Male'}
```

也就是说，如果传递给函数的是 Python 可变对象，并且在函数内使用下标或对象自身原地操作的方法为可变对象增加、删除元素或修改元素值时，修改后的结果是可以反映到函数之外的，实参也得到相应的修改。

5.3 参数类型

在 Python 中，函数参数的形式有很多种，主要可以分为普通位置参数、默认值参数、关键参数、可变长度参数等。Python 函数的定义也非常灵活，在定义函数时不需要指定参数的类型，形参的类型完全由调用者传递的实参类型及 Python 解释器的理解和推断来决定；同样，也不需要指定函数的返回值类型。函数的返回值类型由 return 语句返回值的类型来决定，如果函数中没有 return 语句或者没有执行到 return 语句而返回或者执行了不带任何值的 return 语句，函数都默认为返回空值 None。

没有任何特殊说明的参数为位置参数，实参按顺序依次传递给形参，要求实参和形参的数量和顺序都一致。

5.3.1 默认值参数

在定义函数时，Python 支持默认值参数，即在定义函数时为形参设置默认值。在调用带有默认值参数的函数时，可以不用为设置了默认值的形参传递实参，此时函数将会直接使用函数定义时设置的默认值。默认值参数与 5.3.3 节介绍的可变长度参数可以实现类似于函数重载的目的。带有默认值参数的函数定义语法如下：

```
def 函数名(…, 形参名=默认值):
    函数体
```

调用带有默认值参数的函数时，可以不对默认值参数进行赋值，也可以通过显式赋值来替换其默认值，具有较大的灵活性。如果需要，可以使用“函数名.__defaults__”随时查看函数所有默认值参数的当前值，其返回值为一个元组，其中的元素依次表示每个默认值参数的当前值。例如下面的函数定义：

```
>>> def say(message, times=1):
    print((message+' ') * times)
>>> say.__defaults__
(1,)
```

调用该函数时，如果只为第一个参数传递实参，则第二个参数使用默认值 1；如果为第

二个参数传递实参，则不再使用默认值 1，而是使用调用者显式传递的值。

```
>>> say('hello')
hello
>>> say('hello', 3)
hello hello hello
>>> say('hi', 7)
hi hi hi hi hi hi hi
```

在定义带默认值参数的函数时，默认值参数必须全部出现在位置参数右侧，任何一个默认值参数右侧都不能再出现没有默认值的普通位置参数。例如下面的示例，前两个函数不符合这一要求，从而导致函数定义失败，如图 5-2 所示。

```
>>> def f(a=3,b,c=5):
        print a,b,c

SyntaxError: non-default argument follows default argument
>>> def f(a=3,b):
        print a,b

SyntaxError: non-default argument follows default argument
>>> def f(a,b,c=5):
        print a,b,c


>>>
```

图 5-2　带默认值参数的函数定义

默认值参数的值是在函数定义时确定的，然后默认值参数的引用不再变化，调用函数且不给默认值参数传递实参时将一直使用这个引用。对于列表、字典这样可变类型的默认值参数，这一点可能会导致很严重的逻辑错误，而这种错误或许会耗费较多的精力来定位和纠正。例如：

```
def demo(newitem, old_list=[]):
    old_list.append(newitem)
    return old_list
print(demo('5', [1, 2, 3, 4]))
print(demo('aaa', ['a', 'b']))
print(demo('a'))
print(demo('b'))
```

运行上面的代码，仔细看看结果，是否能发现问题呢？然后把代码修改为下面的样子，再运行，看看区别在哪里。仔细阅读本节前面的内容，应该会发现答案。

```
def demo(newitem, old_list=None):
    if old_list is None:
        old_list = []
    new_list = old_list[:]
    new_list.append(newitem)
    return new_list

print(demo('5', [1, 2, 3, 4]))
print(demo('aaa', ['a', 'b']))
print(demo('a'))
print(demo('b'))
```

下面代码再一次演示了函数参数默认值是在函数定义时确定的。

```
>>>i = 3
>>>def f(n=i):                          #参数 n 的值仅取决于 i 的当前值
    print(n)

>>>f()
3
>>>i = 5                                #函数定义后修改 i 的值不影响参数 n 的默认值
>>>f()
3
```

5.3.2 关键参数

通过关键参数可以按参数名传递实参，实参顺序可以和形参顺序不一致，但不影响参数的传递结果，避免了用户需要牢记参数位置和顺序的麻烦，使得函数的调用和参数传递更加灵活方便。

```
>>> def demo(a, b, c=5):
    print(a, b, c)
>>> demo(3, 7)                          #位置参数 a 和 b，参数 c 使用默认值
3 7 5
>>> demo(c=8, a=9, b=0)                 #关键参数
9 0 8
```

对于 Python 3.8 及更高版本，在定义函数时，可以使用单个斜线或单个星号作为参数，这两个符号不是真正的参数，只用于对其他参数进行约束。其中，单个斜线表示前面的所有参数都必须以位置参数的形式进行传递，单个星号表示后面的所有参数都必须以关键参数的形式进行传递。

```
>>>def func(a, /, *, b, c):            #参数 a 必须以位置参数的形式传递
                                       #参数 b 和 c 必须以关键参数的形式传递
    return a+b+c
>>>func(a=3, b=4, c=5)                 #参数 a 不能使用关键参数的形式传递，出错
TypeError: func() got some positional-only arguments passed as keyword arguments: 'a'
>>>func(3, 4, 5)                       #参数 b 和 c 不能使用位置参数的形式传递，出错
TypeError: func() takes 1 positional argument but 3 were given
>>>func(3, b=4, c=5)                   #符合要求，成功调用函数
12
```

5.3.3 可变长度参数


5.3.3

可变长度参数在定义函数时主要有两种形式：*parameter 和**parameter，前者用于接收任意多个位置实参并将其放在一个元组中，后者接收多个关键参数并将其放入字典中。

下面的代码演示了第一种形式可变长度参数的用法，无论调用该函数时传递了多少位置实参，一律将其放入元组中，元组长度由实参个数确定。

```
>>> def demo(*p):
    print(p)
>>> demo(1, 2, 3)
(1, 2, 3)
```

```
>>> demo(1, 2, 3, 4, 5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
```

下面的代码演示了第二种形式可变长度参数的用法，在调用该函数时自动将接收的关键参数转换为字典，字典长度由实参个数确定。

```
>>> def demo(**p):
    for item in p.items():
        print(item)
>>> demo(x=1, y=2, z=3)                          #关键参数
('x', 1)
('y', 2)
('z', 3)
```

下面的代码演示了定义函数时几种不同形式的参数混合使用的用法。虽然 Python 完全支持这样做，但是除非真的很必要，否则不要这样用，因为这会使得代码非常混乱而严重降低可读性，并导致程序查错非常困难。另外，一般而言，一个函数如果可以接收很多参数，很可能是函数设计得不好，例如，函数功能过多。需要进行必要的拆分和重新设计，以满足高内聚的要求，同时也利于代码阅读和维护。

```
>>> def func_4(a, b, c=4, *aa, **bb):
    print((a, b, c))
    print(aa)
    print(bb)
>>> func_4(1, 2, 3, 4, 5, 6, 7, 8, 9, xx='1', yy='2', zz=3)
(1, 2, 3)
(4, 5, 6, 7, 8, 9)
{'xx': '1', 'yy': '2', 'zz': 3}
```


5.3.4

5.3.4 参数传递时的序列解包

为含多个形参的函数传递参数时，可以使用 Python 列表、元组、集合、字典以及其他可迭代对象作为实参，并在实参名前加一个星号，Python 解释器将自动进行解包，然后传递给多个位置形参。如果使用字典对象作为实参，则默认使用字典的“键”；如果需要将字典中“键:值”对作为参数，则需要使用 `items()`方法；如果需要将字典的“值”作为参数，则需要调用字典的 `values()`方法。最后，务必保证实参中元素个数与形参个数相等，否则将出现错误。

```
>>> def demo(a, b, c):
    print(a+b+c)
>>> seq = [1, 2, 3]
>>> demo(*seq)
6
>>> tup = (1, 2, 3)
>>> demo(*tup)
6
>>> dic = {1:'a', 2:'b', 3:'c'}
>>> demo(*dic)
6
>>> demo(*dic.values())
abc
>>> Set = {1, 2, 3}
>>> demo(*Set)
6
```

如果使用字典作为函数实参，在前面使用两个星号进行解包时，会把字典解包成为关键参数进行传递，字典的“键”作为参数名，字典的“值”作为参数的值。

```
>>> def demo(a, b, c):
    print(a+b+c)
>>> demo(**{'a':97, 'b':98, 'c':99})          #每个“键”都必须在形参列表中
294
```

5.4 return 语句

return 语句用于结束函数的执行，同时还可以通过 return 语句从函数中返回一个任意类型的值。不论 return 语句出现在函数的什么位置，一旦得到执行将直接结束函数。如果函数没有 return 语句、有 return 语句但没有执行或者执行了不返回任何值的 return 语句，Python 将认为该函数以 return None 结束，即返回空值。

在调用函数和方法（见第 6 章）时，一定要注意有没有返回值，以及是否会对参数的值进行修改。例如第 2 章介绍过的列表对象方法 sort()属于原地操作，没有返回值；而内置函数 sorted()返回排序后的列表，并不对原列表做任何修改。

```
>>> a_list = [1, 2, 3, 4, 9, 5, 7]
>>> print(sorted(a_list))                   #返回排序后的新列表
[1, 2, 3, 4, 5, 7, 9]
>>> print(a_list)                           #不影响原列表的内容
[1, 2, 3, 4, 9, 5, 7]
>>> print(a_list.sort())                    #原地排序，没有返回值
None
>>> print(a_list)
[1, 2, 3, 4, 5, 7, 9]
```

5.5 变量作用域

变量起作用的代码范围称为变量的作用域，不同作用域内同名变量之间互不影响。

在 Python 中，主要有局部变量、nonlocal 变量和全局变量这三类，范围依次从近到远。在访问一个变量时，首先会使用局部变量，如果没有同名的局部变量则尝试使用外层函数中的 nonlocal 变量，如果不存在外层函数或者同名的 nonlocal 变量则尝试使用全局变量，如果全局变量也不存在则再尝试使用内置命名空间中的标识符，如果仍不存在则提示错误。本书重点介绍局部变量和全局变量。

一个变量在函数外定义和在函数内定义，其作用域是不同的，函数内使用赋值语句直接创建的变量为局部变量，在函数外创建的变量为全局变量。一般而言，局部变量的引用速度比全局变量快，应优先考虑使用。除非真的有必要，否则应尽量避免使用全局变量，因为全局变量会增加不同函数之间的隐式耦合度，从而降低代码可读性，同时也使得代码测试和纠错变得很困难。

在函数内定义的普通变量只在该函数内起作用，称为局部变量。当函数运行结束后，在该函数内定义的局部变量被自动删除而不可访问。在函数内使用关键字 global 定义的全局变量当函数结束以后仍然存在并且可以访问。

如果想在函数内修改一个在函数外定义的变量的值，那么这个变量就不能是局部的，其

作用域必须为全局的，能够同时作用于函数内外，称为全局变量，可以通过 global 声明或定义。这分两种情况。

（1）一个变量已在函数外定义，如果在函数内需要修改这个变量的值，可以在函数内用关键字 global 声明使用这个全局变量，明确声明要使用已定义的同名全局变量。

（2）在函数内直接使用 global 关键字将一个变量声明为全局变量，如果在函数外没有定义该全局变量，在调用这个函数之后，将自动增加新的全局变量，应避免这样做。

或者说，也可以这么理解：在函数内如果只引用某个变量的值而没有为其赋新值，该变量为（隐式的）全局变量；如果在函数内任意位置有为变量赋值的操作，该变量即被认为是（隐式的）局部变量，除非在函数内显式地用关键字 global 进行声明。

下面的示例代码演示了局部变量和全局变量的用法。

```
>>> def demo():
    global x                                #声明或创建全局变量
    x = 3                                   #修改全局变量的值
    y = 4                                   #局部变量
    print(x, y)
>>> x = 5                                   #在函数外定义全局变量 x
>>> demo()                                  #本次调用修改了全局变量 x 的值
3  4
>>> x
3
>>> y                                       #局部变量在函数运行结束之后自动删除
NameError: name 'y' is not defined
>>> del x                                   #删除了全局变量 x
>>> x
NameError: name 'x' is not defined
>>> demo()                                  #本次调用创建了全局变量
3  4
>>> x
3
>>> y                          #局部变量在函数调用和执行结束后自动删除，在函数外不可访问
NameError: name 'y' is not defined
```

在函数内任意位置只要有为变量赋值的语句，那么在整个函数内该变量都是局部变量。在这条赋值语句之前不能有引用变量值的操作，否则会引发代码异常，除非在函数开始处使用关键字 global 声明该变量为全局变量。

```
>>> x = 3
>>> def f():
    print(x)                                #本意是先输出全局变量 x 的值，但是不允许这样做
    x = 5                                   #有赋值操作，因此在整个作用域内 x 都是局部变量
    print(x)
>>> f()                                     #略去异常的详细信息
UnboundLocalError: local variable 'x' referenced before assignment
```

如果局部变量与全局变量具有相同的名字，那么该局部变量会在自己的作用域内隐藏同名的全局变量，例如下面的代码：

```
>>> def demo():
    x = 3                                    #创建了局部变量,并自动隐藏了同名的全局变量
>>> x = 5
>>> demo()                                   #不会影响全局变量的值
>>> x
5
```

5.6 lambda 表达式

lambda 表达式常用于声明匿名函数,即没有函数名的临时使用的小函数。lambda 表达式只可以包含一个表达式,不允许使用选择结构、循环结构、函数定义等语法,但在表达式中可以调用其他函数,并支持默认值参数和关键参数,该表达式的计算结果就是函数的返回值。

lambda 表达式属于可调用对象之一,常用于内置函数 sorted()、max()、min()和列表方法 sort()的 key 参数,内置函数 map()、filter()和标准库函数 reduce()的第一个参数,以及其他可以使用函数的地方。

下面的代码演示了不同情况下 lambda 表达式的应用。

```
>>> f = lambda x, y, z: x+y+z                          #可以给 lambda 表达式起名字
>>> print(f(1, 2, 3))                                  #可以像普通函数一样调用
6
>>> g = lambda x, y=2, z=3: x+y+z                      #含默认值参数
>>> print(g(1))
6
>>> print(g(2, z=4, y=5))                              #调用时使用关键参数
11
>>> L = [(lambda x: x**2), (lambda x: x**3), (lambda x: x**4)]
>>> print(L[0](2), L[1](2), L[2](2))                   #使用没有名字的 lambda 表达式
4 8 16
>>> D = {'f1':(lambda: 2+3), 'f2':(lambda: 2*3), 'f3':(lambda: 2**3)}
>>> print(D['f1'](), D['f2'](), D['f3']())
5 6 8
>>> L = [1, 2, 3, 4, 5]
>>> print(map((lambda x: x+10), L))                    #使用没有名字的 lambda 表达式
[11, 12, 13, 14, 15]
>>> L
[1, 2, 3, 4, 5]
>>> def demo(n):
    return n*n
>>> demo(5)
25
>>> a_list = [1, 2, 3, 4, 5]
>>> list(map(lambda x: demo(x), a_list))      #包含函数调用并且没有名字的 lambda 表达式
[1, 4, 9, 16, 25]
>>> data = list(range(20))
>>> import random
>>> random.shuffle(data)
```