# 第5章 函　　数

在软件开发过程中，经常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。很显然，从软件设计和代码复用的角度来讲，直接将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。虽然这样可以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，也增加了代码阅读、理解和维护的难度，为代码测试和纠错带来很大的困难。一旦被复制的代码块将来某天被发现存在问题而需要修改，必须对所有的复制都做同样的正确修改，这在实际中是很难完成的一项任务。更糟糕的情况是，由于代码量的大幅度增加，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了新漏洞，维护成本大幅度增加。因此，**应尽量减少使用直接复制代码的方式来实现复用。**解决这个问题的有效方法是设计函数（function）和类（class）。本章介绍函数的设计与使用，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，然后在需要该功能的地方调用封装好的函数，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，只需要修改该函数的代码则所有调用位置均得到体现。同时，把大任务拆分成多个函数也是分治法的经典应用，复杂问题简单化，使得软件开发像搭积木一样简单。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，并不是使用了函数就万事大吉了。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，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。再就是设计函数时应尽量减少副作用，只实现指定的功能就可以了，不要做多余的事情。最后，在实际项目开发中，往往会把一些通用的函数封装到一个模块中，并把这个通用模块文件放到顶层文件夹中，这样更方便管理。

## 5.1　函数的定义与使用

### 5.1.1　基本语法

5.1.1 节

在 Python 中，定义函数的语法如下：

```
def 函数名 ([ 参数列表 ]):
    ''' 注释 '''
    函数体
```

在 Python 中使用 def 关键字来定义函数，然后是一个空格和函数名称，接下来是一对括号，在括号内是形式参数列表，如果有多个参数则使用逗号分隔开，括号之后是一个冒号和换行，最后是注释和函数体代码。定义函数时在语法上需要注意的问题主要有：①函数形参不需要声明其类型（也可以声明，但实际不起作用），也不需要指定函数的返回值类型；②即使该函数不需要接收任何参数，也必须保留一对空的括号；③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。

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):                                  # 定义函数，括号里的 n 是形参
    '''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()
```

该函数的调用方式为

```
fib(1000)                                    # 调用函数，括号里的 1000 是实参
```

如果代码本身不能提供非常好的可读性，那么最好加上适当的注释来说明。在定义函数时，开头部分的注释（称作文档字符串）并不是必需的，但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，可以使用内置函数 help() 来查看函数的使用帮助，并且在调用该函数时输入左侧圆括号之后，立刻就会得到该函数的使用说明，如图 5-1 所示。

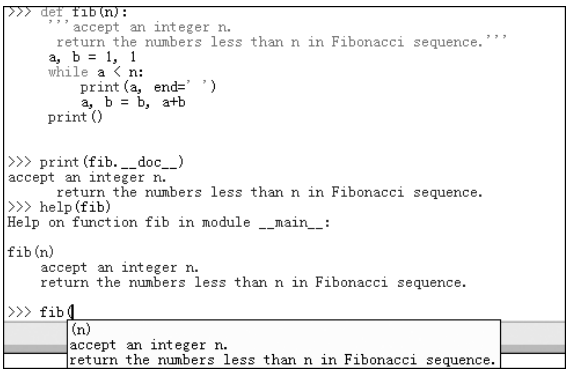


图 5-1 使用注释来为用户提示函数使用说明

在 Python 中，定义函数时也不需要声明函数的返回值类型，而是使用 return 语句结束函数执行的同时返回任意类型的值，**函数返回值类型与 return 语句返回表达式的类型一致**。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的执行。**如果函数没有 return 语句、有 return 语句但是没有执行到或者执行了不返回任何值的 return 语句，解释器都会认为该函数以 return None 结束，即返回空值。**

在编写函数时，应尽量减少副作用，尽量不要修改参数本身，除非接口规范中有明确要求。另外，应充分利用 Python 函数式编程的特点，让自己定义的函数尽量符合纯函数式编程的要求，如保证线程安全、可以并行运行等。

## 5.1.2　函数嵌套定义、可调用对象与修饰器（选讲）

5.1.2 节

### 1. 函数嵌套定义

Python 允许函数的嵌套定义，在函数内部可以再定义另外一个函数。下面的函数利用函数嵌套定义和递归实现帕斯卡公式 C(n, i) = C(n-1, i) + C(n-1, i-1)，进行组合数 C(n, i) 的快速求解。

```
def f2(n, i):                                  # 定义外层函数
    cache2 = dict()

    def f(n, i):                               # 定义内层函数
        if n==i or i==0:
            return 1
        elif (n,i) not in cache2:
            cache2[(n,i)] = f(n-1,i) + f(n-1,i-1)
        return cache2[(n,i)]
    return f(n, i)                             # 调用内层函数，返回其返回值
```

尽管函数嵌套定义使用很方便，也很灵活，但并不提倡过多使用，因为这样会导致内部的函数反复定义而影响执行效率。

### 2. 可调用对象

函数属于 Python 可调用对象之一，由于构造方法的存在，类也是可调用的。像 list()、tuple()、dict()、set() 这样的用法实际上都是调用了类的构造方法。另外，任何包含 __call__() 方法的类的对象也是可调用的。下面的代码使用函数的嵌套定义实现了可调用对象的定义：

```
def linear(a, b):
    def result(x):
        return a*x + b
    return result                                  # 返回内层函数
```

下面的代码演示了可调用对象类的定义：

```
class linear:
    def __init__(self, a, b):                     # 面向对象程序设计见第 6 章
```

```
        self.a, self.b = a, b
    def __call__(self, x):                    # 这里是关键
        return self.a*x + self.b
```

使用上面的嵌套函数和类这两种方式中的任何一种，都可以通过以下方式来创建一个可调用对象：

```
taxes = linear(0.3, 2)
```

然后通过以下方式来调用该对象：

```
taxes(5)
```

### 3．修饰器

修饰器（decorator）是函数嵌套定义的另一个重要应用。**修饰器本质上也是一个函数，只不过这个函数接收其他函数作为参数并对其进行一定的修饰之后返回新函数。**后面第 6 章中的静态方法、类方法、属性等都是通过修饰器实现的，Python 中还有很多这样的用法。下面的代码演示了修饰器的定义与使用方法，定义其他函数调用之前或之后需要执行的通用代码，可作用于其他任何函数，提高代码复用度。

```
def before(func):                          # 定义修饰器,func 是将要被修饰的函数
    def wrapper(*args, **kwargs):          # wrapper 的参数就是 func 的参数
        print('Before function called.')
        return func(*args, **kwargs)       # 调用被修饰的原函数
    return wrapper                         # 返回修饰后的新函数

def after(func):                           # 定义修饰器
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('After function called.')
        return result
     return wrapper

@before
@after
def test():                          # 同时使用两个修饰器改造函数,距离近的先起作用
    print(3)
# 调用被修饰的函数
test()
```

上面代码的运行结果为

```
Before function called.
3
After function called.
```

### 5.1.3　函数递归调用


5.1.3 节

函数的递归调用是函数调用的一种特殊情况，函数调用自己，自己再调用自己，自己再调用自己……，当某个条件得到满足时就不再调用了，然后再一层一层地返回，直到该函数的第一次调用，如图 5-2 所示。

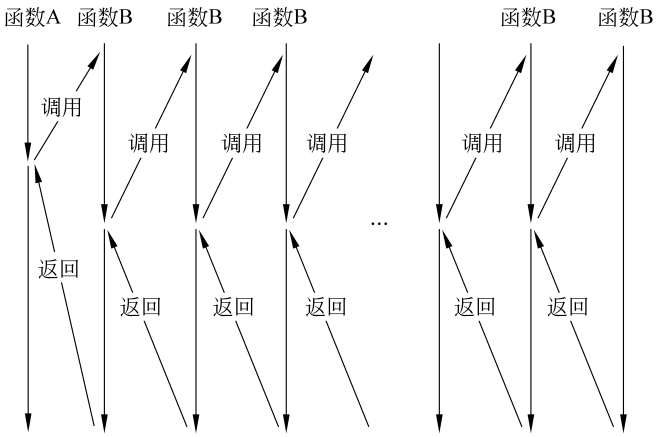


图 5-2　函数递归调用示意图

函数递归通常用来把一个大型的复杂问题层层转化为一个与原来问题性质相同但规模很小、很容易解决或描述的问题，只需要很少的代码就可以描述解决问题过程中需要的大量重复计算。下面的代码使用递归计算列表中所有元素之和，尽管在 Python 中没有这样做的必要。

```
def recursiveSum(lst):
    if len(lst) == 1:
        return lst[0]
    return lst[0] + recursiveSum(lst[1:])
```

而下面的代码使用递归实现了整数的因数分解，函数执行结束后，fac 中包含了整数 num 因数分解的结果。

```
from random import randint

def factors(num):
    # 每次都从 2 开始查找因数
    for i in range(2, int(num**0.5)+1):
        # 找到一个因数
        if num%i == 0:
            facs.append(i)
            # 对商继续分解，重复这个过程
            factors(num//i)
            # 注意，这个 break 非常重要
```

```
                break
        else:
            # 不可分解了，自身也是个因数
            facs.append(num)

facs = []
n = randint(2, 10**8)
factors(n)
result = '*'.join(map(str, facs))
if n == eval(result):
    print(n, '='+result)
```

最后，从图 5-2 可以看出，每次调用函数必须记住离开的位置才能保证函数运行结束以后回到正确的位置，这个过程称为保存现场，这需要一定的栈空间。另外，调用一个函数时会为该函数分配一个栈帧，用来存放参数和函数内部局部变量的值，这个栈帧会在函数调用结束后自动释放。而在函数递归调用的情况中，一个函数执行尚未结束就又调用了自己，原来的栈帧还没释放又分配了新栈帧，会占用大量的栈空间。所以，递归深度如果太大，可能会导致栈空间不足进而导致程序崩溃。

5.2 节

## 5.2 函数参数

函数定义时括号内是使用逗号分隔开的形参列表 (parameters)，函数可以有多个参数，也可以没有参数，但定义和调用时一对括号必须有，表示这是一个函数并且不接收参数。调用函数时向其传递实参 (arguments)，**将实参的引用传递给形参**。**定义函数时不需要声明参数类型**，解释器会根据实参的类型自动推断形参类型，在一定程度上类似于函数重载和泛型函数的功能。

**在函数内部直接修改形参的值，实际是修改引用，不会影响实参**，例如：

```
>>> def addOne(a):
    a += 1                          # 这条语句会得到一个新的变量 a
>>> a = 3
>>> addOne(a)
>>> a                               # 实参的值没有受到影响
3
```

从运行结果可以看出，在函数内部修改了形参 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。然而，列表、字典、集合这样的可变序列类型作为函数参数时，如果在函数内部通过下标或列表、字典或集合对象自身的方法修改参数中的元素时，同样的作用也会体现到实参上。

```
>>> def modify(v):                          # 修改列表元素值
    v[0] = v[0] + 1
>>> a = [2]
```

```
>>> modify(a)
>>> a
[3]
>>> def modify(v, item):                    # 为列表增加元素
    v.append(item)
>>> a = [2]
>>> modify(a, 3)
>>> a
[2,3]
>>> def modify(d):                          # 修改字典元素值或为字典增加元素
    d['age'] = 38
>>> a = {'name': 'Dong', 'age': 37, 'sex': 'Male'}
>>> modify(a)
>>> a
{'name': 'Dong', 'age': 38, 'sex': 'Male'}
>>> def modify(s, v):                       # 为集合添加元素
    s.add(v)
>>> s = {1, 2, 3}
>>> modify(s, 4)
>>> s
{1, 2, 3, 4}
```

也就是说，**如果传递给函数的是列表、字典、集合或其他自定义的可变序列，并且在函数内部使用下标或序列自身支持的方法为可变序列增加、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**，即实参也得到了相应的修改。

### 5.2.1　位置参数

位置参数（positional arguments）是比较常用的形式，调用函数时实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同，**实参按位置逐个传递给形参。**

```
>>> def demo(a, b, c):                      # 所有形参都是位置参数
    print(a, b, c)
>>> demo(3, 4, 5)                           # 实参按位置逐个传递给形参
3 4 5
>>> demo(3, 5, 4)
3 5 4
>>> demo(1, 2, 3, 4)                        # 实参与形参的数量必须相同
TypeError:demo() takes 3 positional arguments but 4 were given
```

### 5.2.2　默认值参数

Python 支持默认值参数，在定义函数时可以为形参设置默认值。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，也可以通过显式传递实参来替换其默认值。也就是说，**在调用函数时是否为默认值参数传递实参是可选的**，具有较大的灵活性，在一定程度上类似于函数重载的功能，同时还能在为函数增加新的参数和功能时通过为新参数设置默认值来保证向后兼容而不影响老用户的使用。需要注意的是，在定义带有默认值参

数的函数时，**任何一个默认值参数右边都不能再出现没有默认值的普通位置参数**，否则会提示语法错误。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
```

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。

```
>>> 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
```

**多次调用函数并且不为默认值参数传递值时，默认值参数只在函数定义时进行一次解释和初始化**，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大量精力来定位和纠正。

```
>>> def demo(newitem, old_list=[]):
    old_list.append(newitem)
    return old_list

>>> print(demo('5', [1,2,3,4]))
[1, 2, 3, 4, '5']
>>> print(demo('aaa', ['a','b']))
['a', 'b', 'aaa']
>>> print(demo('a'))
['a']
>>> print(demo('b'))                              # 注意这里的输出结果
['a', 'b']
```

上面的函数使用列表作为默认参数，由于其可记忆性，连续多次调用该函数而不给该参数传值时，再次调用将保留上一次调用的结果。一般来说，**要避免使用列表、字典、集合或其他可变序列作为函数参数默认值**，对于上面的函数，更建议使用下面的写法。

```
def demo(newitem, old_list=None):
    if old_list is None:
        old_list = []
```

```
    old_list.append(newitem)
    return old_list
```

如果在定义函数时某个参数的默认值为另一个变量的值，那么参数的默认值只依赖于函数定义时该变量的值，或者**说函数的默认值参数是在函数定义时确定值的**，只会被初始化一次，例如：

```
>>> 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
>>> def f(n=i):                    # 重新定义函数
   print(n)
>>> f()
5
```

## 5.2.3　关键参数

通过关键参数可以按参数名字传递值，明确指定哪个值传递给哪个参数，**实参顺序可以和形参顺序不一致**，避免了用户牢记参数位置和顺序的麻烦。

```
>>> def demo(a, b, c=5):
    print(a, b, c)
>>> demo(3, 7)                                  # 按位置传递参数
3 7 5
>>> demo(c=8, a=9, b=0)                         # 关键参数
9 0 8
```

## 5.2.4　可变长度参数

可变长度参数在定义函数时主要有两种形式：*parameter 和 **parameter，前者用来接收任意多个位置实参并将其放在一个元组中，后者接收类似于关键参数一样显式赋值形式的多个实参并将其放入字典中。

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实参，一律放入元组中，元组长度由实参数量决定：

```
>>> 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，即在调用该函数时将接收的关键参数转换为字典，字典长度由关键参数的数量决定：

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

Python 定义和调用函数时可以同时使用位置参数、关键参数、默认值参数和可变长度参数，但是除非真的很必要，否则不要这样做，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，如果一个函数可以接收很多不同类型的参数，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，以满足模块高内聚的要求。

### 5.2.5 实参序列解包

调用含有多个位置参数（positional arguments）的函数时，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，并在实参名称前加一个星号，Python 解释器将自动进行解包，然后把序列中的值作为位置参数分别传递给多个单变量形参。

```
>>> def demo(a, b, c):
    print(a+b+c)
>>> seq = [1, 2, 3]
>>> demo(*seq)                          # 对列表进行解包
6
>>> tup = (1, 2, 3)
>>> demo(*tup)                          # 对元组进行解包
6
>>> dic = {1: 'a', 2: 'b', 3: 'c'}
>>> demo(*dic)                          # 对字典的“键”进行解包
6
>>> demo(*dic.values())                 # 对字典的“值”进行解包
abc
>>> Set = {1, 2, 3}
>>> demo(*Set)                          # 对集合进行解包
6
```

如果实参是字典，可以使用两个星号 ** 对其进行解包，会把字典元素转换成关键参数的形式进行参数传递。对于这种形式的序列解包，要求实参字典中的所有键都必须是函数的形参名称，或者与函数中两个星号的可变长度参数相对应。

```
>>> p = {'a': 1, 'b': 2, 'c': 3}        # 要解包的字典
>>> def f(a, b, c=5):                   # 带有位置参数和默认值参数的函数
```