第 5 章 函 数

函数(function)是执行计算的命名语句序列。将一段代码封装为函数并在需要的位置进行调用,不仅可以实现代码的重复利用,更重要的是可以保证代码的完全一致。

教学课件

5.1 函数的定义和使用

1. 函数的定义

函数定义(definition)的语法形式:

```
def 函数名([形式参数列表]):                #[]表示可选,即一个函数可以没有形式参数
    ''' docstring①'''                      #函数的功能说明,也就是文档字符串
    函数体中的语句
```

Python 使用关键字 def 定义函数。关键字 def 后面有一个空格,然后是函数名,接下来是一对圆括号,圆括号里面是形式参数(formal parameter),圆括号后面是一个冒号和换行,最后是必要的注释以及函数代码。定义函数时需要注意如下几个问题:

(1) 一个函数即使不需要接收任何参数,也必须保留一对圆括号;

(2) 括号后面的冒号必不可少;

(3) 函数体(包括注释部分)相对于 def 关键字必须向右缩进一定数量的空格。

下面定义一个 add()函数,该函数接收两个形式参数 x1 和 x2。

```
def add(x1, x2):                           #函数头
    '''Return the sum of x1 and x2.'''
    return x1 +x2
```

在上述定义的 add()函数中,其函数体的第 1 行是注释,也就是文档字符串 docstring。在 Python 语言中有些工具利用 docstring 自动生成联机文档,使得用户可以方便地、交互式地浏览程序代码,如图 5-1 所示。编写代码时额外添加文档字符串 docstring 是一种好的编程实践,请注意培养这样的好习惯。

---

① docstring 代表文档字符串(documentation string)。

```
>>> def add(x1, x2):                                # 函数头
        '''Return the sum of x1 and x2.'''          # 注意缩进
        return x1 + x2

>>> add.__doc__
'Return the sum of x1 and x2.'
>>> help(add)
Help on function add in module __main__:

add(x1, x2)
    Return the sum of x1 and x2.

>>> add(
      (x1, x2)
      Return the sum of x1 and x2.
```

图 5-1　函数定义中的文档字符串

在图 5-1 中,使用 add()函数的__doc__属性可以查看在该函数的定义中添加的文档字符串,也可以使用 Python 的内置函数 help()查看 add()函数的使用帮助。另外,在调用 add()函数时输入左括号,IDLE 等集成开发环境就会立即弹出该函数的使用帮助信息。

2. 函数的调用(call)

函数定义完毕并不能自动运行,只有被调用时才能运行。下面的代码用整数 1 和 2 调用 add()函数,该函数的返回值被赋值给变量 result。

```
result =add(1, 2)
print(result)
```

程序的运行结果:

```
3
```

上述调用 add()函数时使用的整数 1 和 2 是实际参数(actual parameter),简称实参;而在函数头使用的参数是形式参数,简称形参。形参没有具体的值,形参的值来自实参。

3. 函数的返回值(return value)

通常,定义一个函数是希望它能够返回一个或多个计算结果,这在 Python 语言中是通过关键字 return 来实现的。无论 return 语句出现在函数的什么位置,一旦被执行,它都会立即结束函数的执行过程。如果在函数的定义中没有出现 return 语句或者执行了不返回任何值的 return 语句,Python 解释器就认为该函数以 return None 语句结束,即返回一个空值(None)。下面定义的 3 个函数 demo(x, y),它们的返回值都是 None。

```
def demo(x, y):
    x +y                                #没有 return 语句

def demo(x, y):
    Return                              #return 语句不返回任何值

def demo(x, y):
    return None
```

下面定义一个 fib()函数,该函数能够输出任意指定范围内的斐波那契(Fibonacci)数列[①]:

```
>>>def fib(n):
    '''Print a Fibonacci series up to n.'''
    a, b =0, 1
    while a <n:
        print(a, end=' ')                    #输出 a 的值后,接着输出一个空格
        a, b =b, a+b                         #元组赋值
    print()                                  #输出一个空行
>>>fib(20)                                   #将 20 作为实参调用 fib()函数
```

上述代码的输出结果:

```
0 1 1 2 3 5 8 13
```

可以将一个函数名赋值给另外一个变量,使得该变量也可以作为函数使用:

```
>>>f =fib                                    #这是一种通用的重命名机制
>>>f(20)
0 1 1 2 3 5 8 13
```

由于上述定义的 fib()函数中没有 return 语句,因此 Python 解释器会自动返回一个空值。

```
>>>print(fib(5))                             #注意对比 fib(5)与 print(fib(5))的输出结果
0 1 1 2 3
None                                         #fib(5)不输出此行
```

再看一个例子:

```
>>>def demo():
    pass
>>>print(demo())
None
```

程序源码

修改上述定义的 fib()函数,使其返回一个由斐波那契数组成的列表,而不是在该函数中打印该数列:

```
>>>def fib2(n):
    '''Return a list containing the Fibonacci series up to n.'''
    result =[]
    a, b =0, 1
```

① 斐波那契数列的前两项为 0 和 1,从第 3 项开始,每一项都等于前两项之和,如 0,1,2,3,5,…。

```
    while a <n:
        result.append(a)
        a, b =b, a+b
return result
>>>f100 =fib2(100)                                    #调用 fib2()函数
>>>f100                                               #输出结果
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

5.2　函数的参数类型

在函数的定义中,形参列表的一般形式为

```
<必选参数>, …, <可选参数>=<默认值>, …
```

1. 必选参数

没有给出默认值(default value)的参数都是必选参数。在定义函数时,必选参数必须出现在可选参数的前面。例如,在下面定义的 demo()函数中,参数 x 就是一个必选参数。在调用函数时,必须给必选参数赋值。

```
def demo(x, y=5):
    pass
```

2. 可选参数

带有默认值的参数都是可选参数。在定义函数时已经给可选参数指定了默认值。因此在调用一个函数时,如果没有给可选参数提供值,那么该函数就会使用其默认值。例如,在上面定义的 demo()函数中,y 就是可选参数,因此在调用 demo()函数时,可以不给可选参数 y 提供新值。下面再定义一个函数 person()。

```
def person(name, gender='male', age=20):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
```

可以使用以下三种方式调用上述定义的 person()函数:

(1) 仅给必选参数 name 赋值,如 person('Tim');

(2) 给必选参数 name 和可选参数 gender 赋值,如 person('Tim', 'female');

(3) 给所有参数赋值,如 person('Smith', 'male', 30)。

3. 关键字参数

拥有参数名的参数都是关键字参数(这里所说的关键字不是指 Python 解释器内部定义并使用的关键字,如 if 等)。在 person()函数的定义中出现的 3 个参数都是关键字参数。使用关键字参数可以很方便地给形参赋值,而无须记住各个参数在函数定义中出现的先后顺序。下面是调用 demo()函数的各种方式。

```
demo(3)                          #正确,x 的取值 3,y 的取值 5
demo(x=3)                        #正确,x 的取值 3,y 的取值 5
demo(3, y=4)                     #正确,x 的取值 3,y 的取值 4
demo(y=4, x=3)                   #正确,x 的取值 3,y 的取值 4
demo(y=4, 3)                     #错误
```

4. 可变长度参数

在 Python 语言中,可变长度参数有两种类型:一种是用单星操作符“*”定义的;另一种是用双星操作符“**”定义的。拥有第一种可变长度参数的函数能够接收任意数量的实参,这些实参被封装成一个元组:

```
def multiply(*args):             #arg 代表 argument(参数)
    z =1
    for arg in args:
        z *=arg
    print(z)
```

下面调用 multiply()函数。

```
multiply(4, 5)                   #输出结果为 20
multiply(10, 9)                  #输出结果为 90
multiply(2, 3, 4)                #输出结果为 24
```

在可变长度参数的前面,可能会出现必选参数:

```
def demo(name, *args):           #可变长度参数 args 前有一个必选参数 name
    print(name, end=', ')
    for arg in args:
        print(arg, end=' ')
demo('Jim', 'nice', 'to', 'meet', 'you!')
```

在上一行的函数调用中,name 的取值为 Jim,args 的值为元组('nice', 'to', 'meet', 'you!')。程序的输出结果如下。

```
Jim, nice to meet you!
```

再看一个例子:

```
>>>def concat(*args, sep="/"):
    return sep.join(args)
```

上述代码中的参数 sep 为可选参数,也是关键字参数。

```
>>>concat("a", "b", "c")
'a/b/c'
```

在上述代码中，可选参数 sep 取默认值“/”，可变长度参数 args 的值为元组("a", "b", "c")。

```
>>>concat("one", "two", "three", sep=".")
'one.two.three'
```

在上一行的函数调用中，可变长度参数 args 的值为元组("one", "two", "three")，并且为可选参数 sep 指定了新值“.”。

如果在函数形参列表的最后有一个用双星操作符“* *”定义的可变长度参数，那么该函数能够接收任意数量的实参，而且这些实参被封装成一个字典：

程序源码

```
def print_values(* * kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))

print_values(my_name="Tom", your_name="Tim")
```

上述代码的执行结果：

```
The value of my_name is Tom
The value of your_name is Tim
```

注意：在函数的定义中，双星操作符“* *”必须出现在单星操作符“*”的后面。

下面定义一个 demo()函数。

```
def demo(*args, * * kwargs):            # * * kwargs18 必须出现在 * args 的后面
    for arg in args:
        print(arg, end=' ')
    print()                             #输出一个回车换行
    for key, value in kwargs.items():
        print(key, '=>', value)
```

下面调用 demo()函数。

```
demo('hello', 'world', name='Tom', age=30)
```

程序的输出结果如下。

```
hello world
name =>Tom
age =>30
```

上述四种参数类型相互之间并不是互斥的关系，如可选参数同时也是关键字参数。

5. 函数参数的赋值方式

为方便读者阅读，再次给出 person()函数的定义：

```
def person(name, gender='male', age=20):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
```

在调用函数时,如果不使用参数名给形参赋值,那么将按照实参出现的顺序依次给对应位置上的形参赋值,这种方式叫作**按位置赋值**。如下面的函数调用。

```
person('Tim')
```

执行上述代码后,形参 name 得到的值为 Tim,gender 和 age 取默认值,也就是 gender 的取值为 male,age 的取值为 20。再看下面几种函数调用。

```
person('Tim', 'female')              #name 为 Tim,gender 为 female,age 取默认值 20
person('Smith', 'male', 30)          #name 为 Smith,gender 为 male,age 为 30
```

再强调一次:必须给必选参数指定参数值。综上可知:按位置给形参赋值时,实参的出现顺序非常重要。除了按位置给形参赋值外,还可以通过指定参数名的方式给形参赋值,这种方式叫作**按关键字赋值**。如下面的函数调用。

```
person(name='Tim', gender='female')
person(gender='female', name='Tim')
person(age=22, gender='female', name='Tim')
```

很显然,按关键字赋值的优点是不用考虑实参出现的先后顺序。

也可以将这两种赋值方式混合使用:

```
person('Sue', gender='female')
```

实参 Sue 按位置给形参 name 赋值,而实参 female 按关键字给 gender 赋值。

注意:当混合使用这两种赋值方式时,一定要保证按位置赋值出现在按关键字赋值的前面。下面的函数调用是错误的。

```
person(gender='female', 'Sue')
```

在上述的函数调用中,按位置赋值应出现在前面,即 person('Sue', gender='female')。

再看两个错误的函数调用方法:

```
person('Gorge', name='Gorge')            #为同一个参数指定重复的值
person(weight=150)                       #使用未知的形参 weight
```

有时只能采用按关键字赋值的方式给形参赋值,如下面定义的函数。

```
def demo(*args, name):                   #可变长度参数 args 后面有一个必选参数 name
    print(name, end=', ')
```

```
    for arg in args:
        print(arg, end=' ')
```

假如按位置赋值,也就是采用下面的方式调用 demo()函数,读者可以试一试能否得到期望的输出结果。

```
demo('nice', 'to', 'meet', 'you!', 'Jim')
```

此时应该按关键字赋值:

```
demo('nice', 'to', 'meet', 'you!', name='Jim')
```

5.3　参数解包

有时实参已存储在列表、元组等数据容器中,但是函数调用却需要单独的位置参数,这时就需要使用单星操作符"*"将实参从这些数据容器中解包(unpacking)出来:

```
>>>def demo(x, y, z):
    print(x+y+z)

>>>lt =[1, 2, 3]                          #列表 lt
>>>demo(*lt)
6
>>>tu =(1, 2, 3)                          #元组 tu
>>>demo(*tu)
6
>>>st ={1, 2, 3}                          #集合 st
>>>demo(*st)
6
实参为字典时:
>>>dt ={1:'a', 2:'b', 3:'c'}
>>>demo(*dt)                              #默认使用字典的键
6
>>>demo(*dt.values())                     #明确指定使用字典的值
abc
>>>list(range(3, 6))
[3, 4, 5]
>>>args =[3, 6]
>>>list(range(*args))                     #解包列表,得到实参 3 和 6
[3, 4, 5]                                 #执行结果相同
```

类似地,字典可以使用双星操作符"**"进行解包以便传递关键字参数:

```
>>>def person(name, gender='male', age=20):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)

>>>dt ={"name": "Tom", "gender": "male", "age": 40}
>>>person( * * dt)
name: Tom
gender: male
age: 40
```

5.4 递归函数

简单地说,算法是解决问题的方法与步骤。递归算法(recursive algorithm)的核心思想是分治策略。分治,是“分而治之”(divide and conquer)的意思。分治策略将一个复杂问题反复分解为两个或更多个相同的,或相似的子问题,直到这些子问题可以直接求解,最后将子问题的解合并起来,就能得到原问题的解,如图 5-2 所示。

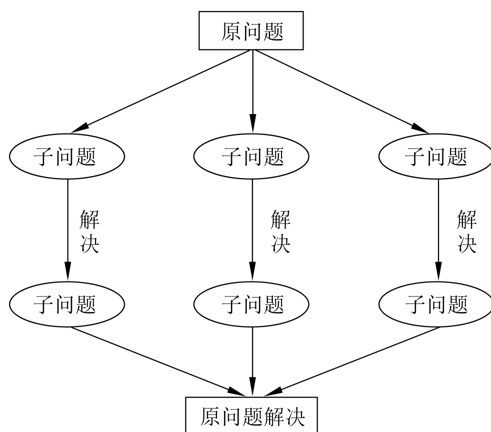

图 5-2 分治策略

在 Python 语言中,分治策略是通过递归函数实现的。一个函数在其函数体内调用它自身,这种函数叫作递归函数。递归函数由终止条件和递归条件两部分构成。下面定义一个计算阶乘的函数 factorial(n)。

```
def factorial(n):                          #factorial 是阶乘的意思
if n <=1:                                  #终止条件
return 1
    else:
        return n * factorial(n -1)         #递归条件
print(factorial(5))
```

上述代码的输出结果：

```
120
```

调用上述定义的 factorial(n)函数计算 5 的阶乘，执行过程如下所示。

```
factorial(5)  =  5 * factorial(4)
              =  5 * 4 * factorial(3)
              =  5 * 4 * 3 * factorial(2)
              =  5 * 4 * 3 * 2 * factorial(1)
              =  5 * 4 * 3 * 2 * 1
              =  120
```

定义计算斐波那契数列的递归函数 fib(n)：

```
def fib(n):
    """ 计算斐波那契数列，参数 n 为数列的第 n 项 """
    if n in [0, 1]:
        return n
    else:
        return fib(n-1) +fib(n-2)
for i in range(10):
    print(fib(i), end=" ")
```

上述代码的执行结果：

```
0 1 1 2 3 5 8 13 21 34
```

5.5　lambda 函数

可以使用 lambda 关键字创建小型匿名函数，如：

```
lambda x, y: x +y
```

该函数返回两个参数之和，可以像普通函数那样使用 lambda 函数。注意，lambda 函数在形式上只能是一个表达式。

```
>>>add_one =lambda x: x +1
>>>add_one(1)
2
```

用普通函数实现上述匿名函数的功能：

```
def add_one(x):
    return x +1
```