第 5 章

函数

本章介绍函数。函数是带名字的代码块，用于完成特定的任务。要执行函数定义的特定任务，可调用该函数。使用函数，当需要在程序中多次执行同一任务时，就无须重复编写完成该任务的代码。通过使用函数，程序的编写、阅读、测试和修改都更容易。

使用 PyCharm 创建一个新的项目 ch05，用来验证本章示例。

5.1 定义函数

Python 的函数有以下 3 类。

- 内建函数：Python 语言自带了许多内建函数，如之前用过的 print()、type()等都是 Python 的内建函数，可以直接用。
- 自定义函数：编程人员可以自己创建函数，叫作自定义函数。设计并实现自定义函数是程序设计的主要工作。
- API 函数：别人创建的、封装好的函数。使用者并不知道它是怎样实现的，但可以直接使用，使用方法与前两类函数无本质区别。

本节主要介绍自定义函数。

1. 最简单的函数

下面示例用最简单的函数形式，说明函数定义与调用的过程。创建一个 Python 文件 test_def1.py，代码为

```
def say_hello():                #①
    """②显示问候语"""
    print("Hello!")             #③

say_hello()                     #④
```

执行结果：

```
Hello!
```

test_def1.py 中说明了最简单的函数结构。第①行使用 def 保留字告诉 Python 要定义一个函数，该行称为函数定义。def 保留字之后必须跟有函数名和包括形式参数的圆括号。本例的函数名是 say_hello。括号中可以向函数传递参数，也就是告诉函数完成任务需要什么样的信息；本例不需要参数，括号内为空。该行要以冒号结束。

函数体语句从函数定义的下一行开始，必须是缩进的。第②行的文本是被称为文档字符串(docstring)的注释，描述了函数是做什么的。文档字符串用三引号括起来，有些工具通过文档字符串自动生成在线的或可打印的文档；即使不生成文档，文档字符串也会让程序员之间的交流变得更加顺畅，在代码中包含文档字符串是一个好的习惯。

第③行是本例函数体唯一的一行执行代码，它的功能是调用 print()函数显示问候语。

要想使用这个函数，可以调用它，如本例的第④行。函数调用让 Python 执行函数中的代码。函数调用需要指明函数名，以及用括号括起来的实际参数。本例不需要参数，因此括号内为空。

2. 向函数传递参数

对 test_def1.py 进行修改，使其在显示问候语时加上用户名，代码为

```
def say_hello(username):
    """显示问候语"""
    print("Hello, "+username+"!")

say_hello("Zhang")
```

执行结果：

```
Hello, Zhang!
```

在函数定义的括号内增加了参数 username，通过该参数，函数可以接受调用时给出的任何值。本例在调用时指定了参数值为"Zhang"，函数的显示结果就是“Hello, Zhang!”。如果函数调用改为 say_hello("Xu")，函数的显示结果就是“Hello, Xu!”。

3. 实参和形参

前面笼统地使用术语“参数”，更深入的讨论需要将参数分为形参和实参。

在函数定义中，括号内要求的参数(如 username)被称为形式参数，简称形参，函数体中根据这些形参进行操作。当调用函数时，需要指定参数的实际值(如 say_hello("Zhang")中的"Zhang")，称为实际参数，简称实参。

5.2 返回值

5.1 节示例函数 say_hello()的功能只是单纯输出。在很多情况下，函数在经过处理之后，需要将处理结果返回给调用者，这时候就要用到返回值。

5.2.1 return 语句

在函数体中，由 return 语句指定函数的返回值。创建一个 Python 文件 test_return1.py，代码为

```
def max(x, y):
    a=x
    if x<y:
        a=y
    return a

m=max(6, 9)
print(m)
```

执行结果：

```
9
```

函数 max 接收两个形式参数，使用 if 语句，将两个参数中值比较大的一个赋值给变量 a，然后用 return 语句返回 a 的值。

语句 m＝max(6, 9)，先将实际参数 6 和 9 传递给函数 max，再将函数返回的结果赋值给变量 m。

需要说明的是，即使函数体中没有 return 语句，函数运行结束后也会隐含地返回一个 None 作为返回值，None 的说明见 2.2.3 节。

5.2.2 多分支 return

Python 函数使用 return 语句返回“返回值”，可以将返回值赋给其他变量作其他的用途。所有函数都有返回值，即使返回的是 None。

一个函数可以有多个 return 语句，遇到任何一个 return 语句，函数立即返回，函数中的其他语句都不再执行，也就是说最多只有一个 return 语句会被执行。如果执行到函数结束也没有遇到 return 语句，就隐式地执行 return None。如果有必要，也可以显式地执行 return None，明确返回一个 None 作为返回值，return None 可以简写为 return。

对 test_return1.py 进行修改，将相关代码修改为

```
def max(x, y):
    if x<y:
        return y
    else:
        return x
```

执行结果完全相同。

5.2.3 返回值类型

可以用 type()函数查看函数返回值的类型。

对 test_return1.py 进行修改，将调用部分代码改为

```
m=max(6, 9)
print(m)
print(type(max(6, 9)))
```

执行结果：

```
9
<class 'int'>
```

Python 的 return 语句只能返回单值，如果需要返回多个值，需要使用复合数据类型，如列表、元组等。

创建一个 Python 文件 test_return2.py，代码为

```
def show_list():
    return [1, 2, 3]


print(show_list())
print(type(show_list()))
```

执行结果：

```
[1, 2, 3]
<class 'list'>
```

可以看到返回的是列表类型。

还可以返回其他的复合类型，如元组。对 test_return2.py 的 return 语句进行修改，将代码改为

```
return (1, 2, 3)
```

执行结果：

```
(1, 2, 3)
<class 'tuple'>
```

继续对 test_return2.py 的 return 语句进行修改，将代码改为

```
return 1, 2, 3
```

执行结果仍然为

```
(1, 2, 3)
<class 'tuple'>
```

return 1，2，3 看似返回多个值，其实隐式地被 Python 封装成了一个元组返回，本例验证了 Python 函数只能返回单值。但利用元组的封装和拆封功能，可以使 Python 函数达到返回多个值的效果。例如：

```
def show_list():
    return 1, 2, 3
```

```
a, b, c=show_list()
print(a)
print(b)
print(c)
```

在本例中，return 语句先将多个值封装成元组，“a，b，c＝show_list()”一句再将函数返回的元组拆封成三个变量。

5.3 参数的传递方式

在函数定义中可以包含若干参数，称为形式参数（形参）。调用时将实际的参数值传入，称为实际参数（实参）。本节讨论实参是如何传递给形参的，也就是参数的传递方式。要理解 Python 的参数传递方式，最好与其他语言（如 C 语言）进行对照，并且借用 C 语言的术语。

C 语言采用的是值传递方式，即形参和实参分配不同的内存地址，在调用时将实参的值复制到形参，在这种情况下，在函数内部修改形参的值不会影响到实参。C++ 增加了“引用”这个概念，即在函数定义时，在形参前加一个 & 符号，表示传递参数的引用，实参与形参指向共同的内存地址，在函数内修改形参值会影响到实参，这种参数传递的方式被称为引用传递。那么，Python 是值传递还是引用传递呢？接下来看两个例子。

示例一：创建一个 Python 文件 test_tran1.py，代码为

```
def swap(x, y):
    """试图交换两个变量的值"""
    temp=x
    x=y
    y=temp
    print(f"函数内部: x={x},y={y}")

a, b=5, 9
print(f"交换之前: a={a},b={b}")
swap(a, b)
print(f"交换之后: a={a},b={b}")
```

执行结果：

```
交换之前: a=5,b=9
函数内部: x=9,y=5
交换之后: a=5,b=9
```

可以看到，在 swap 函数内部对形参 x 和 y 进行了交换，但在函数的外部，a 和 b 的值仍然是函数调用之前的值，没有改变。按照这个示例，看起来 Python 采用的是值传递方式。

示例二：创建一个 Python 文件 test_tran2.py，代码为

```
def proc_list(m):
```

```
    """对列表进行修改"""
    m.append(10)
    print(f"函数内部: {m}")

n=list(range(5))
print(f"调用之前: {n}")
proc_list(n)
print(f"调用之后: {n}")
```

执行结果：

```
调用之前: [0, 1, 2, 3, 4]
函数内部: [0, 1, 2, 3, 4, 10]
调用之后: [0, 1, 2, 3, 4, 10]
```

函数 proc_list 接收一个列表，并在列表的尾部追加一个值。在函数调用之后，外部传入的实参值发生了变化，也就是说，函数内外操作的是同一个列表。按照这个示例，看起来 Python 采用的是引用传递方式。

Python 的参数传递方式与内存使用机制有关，不能确定是值传递还是引用传递，随着版本的变化还可能改变。回顾 4.1.1 节关于 is 操作符及 id()函数的使用，可以帮助读者理解本节内容。如果在程序中适合的位置使用 id()函数，可以看到，在示例一中，a 和 x 的 id 不同，而在示例二中，n 和 m 的 id 相同。通常情况下，简单对象(如 int)采用值传递，复杂对象(如列表)采用引用传递。

5.4 参数类型

在使用 Python 的内置函数(如 print()函数)时，相信读者已经领略到 Python 参数的灵活性。与其他编程语言相比，Python 的一大优势就是其参数类型及传递形式丰富而灵活。Python 中参数的定义形式有多种，不但包括位置参数、默认值参数、关键字参数等一般形式，还可以使用元组参数、字典参数的封装与拆封进一步增强参数传递的灵活性。这些形式可以单独使用也可以混合使用。

5.4.1 位置参数

位置参数是参数定义的最基本形式，本章前面的函数定义采用的都是位置参数。

创建一个 Python 文件 test_para1.py，代码为

```
def f(a, b, c):
    print(a, b, c)

f(1, 2, 3)
```

执行结果：

```
1 2 3
```

位置参数必须以函数定义中的顺序来传递，如函数调用 f(1，2，3)中的 1、2 和 3 分别对应函数定义 f(a，b，c)中的 a、b 和 c。

5.4.2 默认值参数

可以为一个或多个参数指定默认值，这样，在调用时就可以传入比定义时更少的实际参数。创建一个 Python 文件 test_para2.py，代码为

```
def f(a, b=10, c=20):
    print(a, b, c)


f(1, 2, 3)
f(1, 2)
f(1)
f()    #调用时会报错
```

执行结果：

```
Traceback (most recent call last):
  File "E:/PyCharmProjects/ch04/test_para2.py", line 7, in <module>
    f()
TypeError: f() missing 1 required positional argument: 'a'
1 2 3
1 2 20
1 10 20
```

参数 a 是位置参数，调用时必须传入。参数 b 和 c 给出了默认值，调用时这些参数如果不指定实参，将采用默认值。这个函数可以通过以下几种不同的方式调用。

(1) 位置参数调用时必须传入：

```
f()#调用时会报错
```

(2) 只给出必要的参数：

```
f(1)
```

(3) 给出部分可选的参数：

```
f(1, 2)
```

(4) 给出所有的参数：

```
f(1, 2, 3)
```

需要注意的是，对于引用传递的参数(实参值在函数内部可能改变)，即使函数被多次调用，默认值也只会被赋值一次，参数值的改变可能在多次调用中累积。请看下面的示例，创建一个 Python 文件 test_para3.py，代码为

```
def f(a, li=[]):
    li.append(a)
```

```
    return li

print(f(1))
print(f(2))
print(f(3))
```

执行结果：

```
[1]
[1, 2]
[1, 2, 3]
```

如果不想让默认值在后续调用中累积，可以将函数改为如下形式：

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

这个示例的实质是将参数的引用传递改成值传递，这样，参数值的改变就不会在多次调用中累积。

5.4.3　关键字参数

函数调用时可以通过关键字参数的形式来传递参数，形如 keyword＝value。下面示例说明关键字参数的用法，创建一个 Python 文件 test_para4.py，代码为

```
def f(a, b=10, c=20):
    print(a, b, c)

#最一般的调用方式
f(1, 2, 3)
#其他正确的调用方式
f(a=1, b=2, c=3)
f(1, c=2)
f(a=1)
f(b=2, c=3, a=1)
#其他错误的调用方式
f(a=1, 2)        #关键字参数之后不能再使用非关键字参数
f(1, a=2)        #重复指定了参数 a 的值
f(b=2)           #参数 a 的值未指定
f(1, d=2)        #没有名称为 d 的参数
```

函数定义与前例相同，接收一个必选参数(a)和两个可选参数(b 和 c)。前面是正确的调用形式，执行结果：

```
1 2 3
```

```
1 2 3
1 10 2
1 10 20
1 2 3
```

正确的调用方式包括如下两种。

(1) 位置参数也可以用关键字参数的形式来指定值,f(a=1)是正确的。

(2) 指定关键字参数的顺序可以任意,f(b=2, c=3, a=1)是正确的。

错误的调用方式包括如下 3 种。

(1) 在函数调用中,关键字的参数必须跟随在位置参数的后面,所以 f(a=1, 2)是错的。

(2) 任何参数都不可以多次赋值,所以 f(1, a=2)是错的。

(3) 传递的所有关键字参数必须与函数接受的某个参数相匹配,所以 f(1, d=2)是错的。

关键字参数是 Python 程序员非常喜欢的一种参数传递方式,希望读者能够熟练掌握。

5.4.4 元组参数的封装与拆封

3.2 节介绍了元组封装与拆封的概念,5.2 节介绍了函数返回值如何使用元组的封装与拆封,本节介绍元组的封装与拆封如何用于参数传递。

在定义函数时,有时候并不知道调用的时候会传入多少个参数。这时候,采用形如“*name”的形式,可传递可变个数的参数,这些参数被封装进一个元组。

创建一个 Python 文件 test_para5.py,代码为

```
def func(*name):
    print(type(name))
    print(name)

func(1, 2, 3)
func(1, 2, 3, 4, 5, 6)
```

执行结果:

```
<class 'tuple'>
(1, 2, 3)
<class 'tuple'>
(1, 2, 3, 4, 5, 6)
```

可见,形参可被当成元组来操作,从而可以知道实参的数量及各参数的值。

这样传递的参数元组必须在位置参数和默认参数之后。将文件 test_para5.py 修改为

```
def func(a, b=2, *name):
    print(type(name))
    print(name)

func(1, 2, 3)
```

```
func(1, 2, 3, 4, 5, 6)
func(1)
```

执行结果：

```
<class 'tuple'>
(3,)
<class 'tuple'>
(3, 4, 5, 6)
<class 'tuple'>
()
```

可以看出，在前两个调用中，前两个实参被传递给了形参 a 和 b，剩余的参数被封装传递给形参 name。第三个调用只有一个实参，该实参传递给形参 a，形参 b 使用默认参数，而 name 只能接收到一个空元组。

注意：第一个调用显示的元组为(3,)，这是单元素元组的显示形式，在单元素的后面要加一个逗号。

在函数定义中，任何出现在 * name 后面的参数都被当成关键字参数。将文件 test_para5.py 修改为

```
def func(a, b=2, *name, last_para):
    print(type(name))
    print(name)
    print(last_para)


func(1, 2, 3, last_para='OK!')
func(1, 2, 3, 4, 5, 6, last_para='Yes!')
```

执行结果：

```
<class 'tuple'>
(3,)
OK!
<class 'tuple'>
(3, 4, 5, 6)
Yes!
```

上面示例说明了元组参数的封装，相反的过程就是元组参数的拆封。

创建一个 Python 文件 test_para6.py，代码为

```
def func(a, b, c):
    print(a, b, c)


args=(1, 2, 3)
func(*args)
```

执行结果：