第 5 章　函数与模块

函数和模块是 Python 编程中的高级概念，它们可以帮助程序员更好地组织代码，提高代码的复用性和可维护性。一个复杂的任务或问题通常采用“分而治之”的思路解决，即将大任务分解为多个易于解决的小任务，最终解决较大的复杂任务。

本章将从函数的基本概念讲起，详细介绍函数的声明、调用和返回值。随后，讲解参数的传递方式，包括默认参数、可变长参数和关键字参数。此外，本章还将介绍变量的作用域，包括局部变量和全局变量的使用。最后，本章将讲解模块的概念及其创建和使用方法，帮助读者掌握如何将代码组织成模块，提高代码的复用性和可维护性。通过本章的学习，读者将掌握 Python 程序设计中的高级技巧，能够编写更加高效和优雅的代码。

5.1　函数

函数是封装特定逻辑、实现代码模块化与复用的核心工具。它将一组具有特定功能的代码片段进行封装，通过一个名称进行调用，从而有效地解决代码冗余、降低维护成本等问题。函数不仅提高了代码的复用性和可读性，还能让复杂的程序设计变得更加模块化与清晰化。

函数的使用主要包括 3 个核心环节：声明（定义）函数、调用函数、处理函数的返回值。声明函数是程序设计的第一步，定义了函数的功能和接收的参数。调用函数是将函数纳入主程序流程的重要操作，触发函数执行其定义的任务。返回值处理决定函数如何将计算结果反馈给主程序，函数通过 return 语句将结果返回，供后续代码使用。下面详细介绍这三个核心环节。

5.1.1　函数的定义

在 Python 中，定义（声明）一个函数的语法格式为：

```
def <函数名>([<参数列表>]):
    [<函数体>]
    return <返回值>
```

1. def 关键字

函数使用 def 关键字进行声明，这是单词“define”的缩写。函数名必须是一个有效的标识符，即由字母、数字和下画线组成，并且不能以数字开头。

2. 函数名与参数列表

函数名后面跟括号括起的参数列表，如果函数不需要参数，则可以将参数列表省略，但括号不能省略。

3. 函数体

函数体是缩进的代码块，包含函数的具体逻辑。Python 不使用 begin 和 end 或花括号标记代码块的开始与结束，而是通过缩进划分代码结构。

4. 返回值

可以省略，使用 return 关键字返回函数的结果。return 后面可以跟一个表达式，返回该表达式的值。函数一旦执行到 return，就会立即退出并返回值。如果没有 return 语句，则默认返回 None。

下面是一个简单的函数声明示例：

```
def say_hello():
    print('hello, world!')
```

“def say_hello()”声明了函数 say_hello，它没有参数；冒号“:”表示函数体的开始。缩进的语句“print('hello, world!')”是函数体，表示函数的内容。函数名“say_hello()”后面的括号和冒号不能省略。

要调用上面声明的函数，直接使用函数名加括号即可。例如：

```
say_hello()
```

运行结果：

```
hello, world!
```

5.1.2 函数的参数

传入参数的作用是：在函数执行时，接收外部（调用时）提供的数据，使得函数能够根据不同的输入进行计算和操作。通过传入参数，函数会更加灵活和通用。

有如下函数，完成了两个数字相加的功能：

```
def add():
    return 1 + 2
```

这段代码只会计算 1＋2。每次调用 add()时，它都返回相同的结果。假如想要每次都能够计算用户指定的两个数字，而不是固定的 1＋2，该如何实现呢？答案是使用函数的传入参数功能。

通过在函数的定义中使用参数，可以在每次函数调用时动态地传入不同的值。有如下函数定义：

```
def add(x, y):
    result = x + y
    print(f"{x} + {y}的结果是{result}")
```

现在，函数 add(x,y)将不再局限于计算固定的 1＋2，而是可以计算用户提供的任意两个数字。例如：

```
#调用函数
add(5, 6)            #运行结果是 5 + 6 的结果 11
```

在定义和调用函数时，会用到“形式参数”和“实际参数”这两个概念。

(1) 形式参数(形参)。

在定义函数时，使用的 x 和 y 就是形式参数。形式参数表示函数声明时将要使用的占位符，它们并不包含实际的数值，只是告诉 Python 该函数需要两个参数。

(2) 实际参数(实参)。

在调用函数时，传递给函数的 5 和 6 就是实际参数。实际参数是函数执行时实际使用的值。

在调用函数时，提供的数据(实参)会被传入函数定义对应的形式参数，函数根据这些数据执行计算。

5.1.3　函数的返回值

在 Python 中，函数可以通过 return 语句将计算结果返回给调用者。返回值为程序提供了丰富的数据交互能力，不仅可以是单个数值，也可以是字符串、列表、字典，甚至是复合数据结构。当函数执行到 return 语句时，会立即退出，并将 return 后的值传回给调用者。如果函数没有显式的 return 语句，则默认返回 None。

【例 5-1】　使用 return 返回较大值。

程序代码：

```
def maximum(x,y):
    if x>y:
        return x
    else:
        return y
#主函数
print(maximum(2,3))
```

运行结果：

```
3
```

【代码分析】　上面的例子定义了一个名为 maximum()的函数，用于比较两个数 x 和 y。当 x＞y 时，通过 return x 将较大值 x 返回；否则返回 y。使用 print(maximum(2,3))直接输出函数返回的结果。

函数的返回值为程序提供了数据交互能力，不仅可以返回单一结果，也可以返回多个数据。通过 return 语句，函数能够将计算结果反馈给调用者，并终止函数的执行。通常，返回值可以是任意数据类型，如整数、字符串、列表，甚至是复合数据结构。

当需要返回多个结果时，可以使用 return a,b,c 语法，调用者通过 x,y,z＝function()的

形式接收这些返回值。此外，return 语句还能用来提前终止函数的执行，忽略其后的语句。合理使用返回值可以让程序逻辑更加高效与灵活。

5.1.4 函数的文档与注释

函数文档(docstring)用于说明函数功能，可以通过 help()函数或__doc__属性查看，良好的文档能提高代码的可读性和可维护性。

【例 5-2】 查阅 calculate_area()的文档注释。

程序代码：

```
def calculate_area(radius):
    """
    计算圆的面积
    参数:
    radius (float): 圆的半径,必须为正数
    返回:
    float: 圆的面积,计算公式为 π×radius²
    异常:
    ValueError: 当半径为负数时抛出
    """
    if radius < 0:
        raise ValueError("半径不能为负数")
    return 3.14159 * radius * * 2

#查看函数文档
print(help(calculate_area))          #使用 print(calculate_area.__doc__)亦可
```

运行结果：

```
计算圆的面积
    参数:
        radius (float): 圆的半径,必须为正数
    返回:
        float: 圆的面积,计算公式为 π×radius²
    异常:
        ValueError: 当半径为负数时抛出
```

【代码分析】 通过内置的 help()函数或 print(calculate_area.__doc__)获取定义函数的文档。

5.2 函数参数类型与应用

在 Python 中，函数的参数传递方式是非常灵活的，能够根据实际需求选择不同的传递方式。我们已经了解了函数参数最基本的传递方式——按位置传递参数。当调用函数时，传入的参数值会按顺序匹配函数定义中的形式参数。也就是说，参数的传递是根据它们在函数定义中的位置决定的，传递的顺序必须与定义中的顺序一致。

然而，随着程序复杂性的增加，往往需要更加灵活的参数传递方式。Python 提供了几种更为强大的参数传递机制——默认参数、关键字参数和可变长参数。接下来详细探讨这些参数类型，它们不仅能提高代码的可读性和灵活性，还能够帮助开发者编写更具通用性的函数。

5.2.1　默认参数

在 Python 中，默认参数允许为函数的形参设置一个默认值。如果调用函数时没有传递对应参数，则默认值将会生效。这种设计大幅提高了函数的灵活性和适用性。实现默认参数的方法是在函数声明中为形参指定默认值，语法为“形参＝默认值”。需要注意的是，默认参数通常应为不可变类型，以避免潜在的副作用。

注意：默认参数是一个不可变的参数。

【例 5-3】 使用默认参数。

```
def say(message, times=1):
    print(message * times)
#主函数
say("Hello")                    #默认参数 times 为 1
say('World',5)                  #覆盖默认参数，指定 times=5
```

运行结果：

```
Hello
WorldWorldWorldWorldWorld
```

【注意事项】 默认参数必须位于所有非默认参数之后，否则会导致语法错误。如果参数类型为可变对象（如列表或字典），则需要谨慎处理，以避免出现意外行为。

5.2.2　关键字参数

在 Python 中，函数的多个参数值一般默认从左到右依次传入。但是，Python 提供了更灵活的传参方式，即关键字参数（Keyword Arguments），它允许通过指定参数名为函数传递值，从而可以不按照形参的顺序传递参数。

关键字参数具有以下特点。

（1）可以改变传递参数的顺序，通过参数名指定值可以避免顺序错误。

（2）关键字参数通常与默认参数结合使用，以提供更灵活的函数调用方式。

【例 5-4】 使用关键字参数。

```
def func(a,b=5,c=10):
    print("a is",a,"and b is",b,"and c is",c)
#主函数
func(3,7)
func(25,c=24)
func(c=50,a=100)
```

运行结果：

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

【代码分析】 func(3,7)中的参数按默认顺序传递；func(25,c=24)使用了关键字参数，将 c 赋值为 24，顺序不变；func(c=50,a=100)使用关键字参数，改变了传参顺序。

5.2.3 可变长参数

在 Python 中，可变长参数用于处理函数参数数量不固定的情况。根据参数的传递方式，可变长参数可以分为两类。

(1) 位置参数元组：通过一个星号(*)表示，它可以接收任意数量的位置参数，并将这些参数存储为一个元组。通常，这些参数以 *args 的形式传递，但 args 只是一个常见的命名约定，实际上可以使用任何名称，只要以“*”开头即可。

(2) 关键字参数字典：通过两个星号(**)表示，它可以接收任意数量的关键字参数，并将这些参数存储为一个字典。通常，这些参数以**kwargs 的形式传递，kwargs 只是一个常见的命名约定，可以使用任何名称，只要以“**”开头即可。

这两种可变长参数可以同时出现在同一个函数中。函数参数的顺序必须遵循以下规则：

(1) 首先是普通的位置参数；

(2) 其次是可变长位置参数(*args)；

(3) 可变长关键字参数(**kwargs)应放在最后。

【例 5-5】 可变长参数的使用。

```
def foo(x, *y, **z):
    print(f"x: {x}")
    print(f"y: {y}")
    print(f"z: {z}")
```

【代码分析】 函数 foo 首先接收一个普通的参数 x，然后接收可变长位置参数 *y 和可变长关键字参数**z。

【调用示例 1】 仅传入一个位置参数。

```
foo(1)
```

运行结果：

```
x: 1
y: ()
z: {}
```

【结果分析】

(1) x 接收传入的值 1。

(2) y 是一个元组,包含所有的额外位置参数,但由于只传入了一个参数,因此 y 是一个空元组()。

(3) z 是一个字典,可以接收所有的关键字参数,但由于没有传入任何关键字参数,因此 z 是一个空字典{}。

【调用示例 2】 传入多个位置参数。

```
foo(1, 2, 3, 4)
```

运行结果:

```
x: 1
y: (2, 3, 4)
z: {}
```

【结果分析】

(1) x 依然接收第一个位置参数 1。

(2) y 包含剩余的所有位置参数,作为一个元组传递,结果为(2, 3, 4)。

(3) z 依然是一个空字典,因为没有传入任何关键字参数。

【调用示例 3】 传入位置参数和关键字参数。

```
foo(1, 2, 3, a="a", b="b")
```

运行结果:

```
x: 1
y: (2, 3)
z: {'a': 'a', 'b': 'b'}
```

【结果分析】

(1) x 依然接收第一个位置参数 1。

(2) y 包含位置参数(2,3),即所有除 x 外的位置参数。

(3) z 包含关键字参数 a="a"和 b="b",作为一个字典传递,结果为{'a': 'a', 'b': 'b'}。

【调用示例 4】 只传入关键字参数。

```
foo(1, a="a", b="b")
```

运行结果:

```
x: 1
y: ()
z: {'a': 'a', 'b': 'b'}
```

【结果分析】

(1) x 接收位置参数 1。

(2) y 没有额外的顺序位置参数,结果是一个空元组()。

(3) z 包含关键字参数 a="a"和 b="b",结果是一个字典{'a': 'a', 'b': 'b'}。

5.3 变量作用域

变量作用域指变量有效可用的范围，Python 与大多数程序语言一样有局部变量和全局变量，变量的声明通过首次赋值产生，超出变量作用范围时会自动消亡。

5.3.1 局部变量

局部变量指定义在函数体内的变量，它只能被本函数使用，与函数外具有相同名称的其他变量没有任何关系。

【例 5-6】 局部变量举例。

```
def func(x):
    print("x is",x)
    x=2
    print("changed local x to",x)
#主程序
x=50            #全局变量
func(x)
print("x is still",x)
```

运行结果：

```
x is 50
changed local x to 2
x is still 50
```

【代码分析】

(1) 主函数中给 x 赋值为 50。

(2) func 函数中，x 是函数的局部变量，给 x 赋值为 2。

(3) 返回主函数，print()语句中 x 的值没有受到 func()函数中对 x 值的改变，说明主函数中的 x 不受影响。

5.3.2 全局变量

全局变量指定义在函数体外的变量，也称为公用变量，可在其他模块和函数中使用，若需要在函数内部修改全局变量，则应用 global 关键字声明(否则会被视为局部变量)。可以使用 global 语句指定多个全局变量，如 global x,y,z。

【例 5-7】 全局变量举例。

```
def func():
    global x
    print("x is",x)
    x=2
    print("Changed global x to",x)
#主函数
```

```
x=50
func()
print("Value of x is",x)
```

运行结果：

```
x is 50
Changed global x to 2
Value of x is 2
```

【代码分析】 global 语句用来声明 x 是全局变量，在 func()函数内改变 x 的值为 2 时，这个变化也反映在主函数中 x 的值上。

5.3.3　nonlocal 关键字

nonlocal 是一个关键字，用于在嵌套函数中声明一个变量，表明这个变量不是局部变量，而是来自外层(非全局)作用域的变量。它允许开发者在内部函数中修改外部函数中的变量。

【例 5-8】 nonlocal 关键字举例。

```
def outer():
    count = 0                #外层函数的变量
    def inner():
        nonlocal count       #声明: count 不是局部变量,而是来自外层作用域
        count += 1           #修改 count
        print(f"Count is: {count}")
    inner()
    inner()
    return count
result = outer()
print(f"Final result: {result}")
```

运行结果：

```
Count is: 1
Count is: 2
Final result: 2
```

【代码分析】 nonlocal 修饰的变量 count 在外层函数定义，内层函数 inner()对 count 进行加 1 操作，外层变量 count 被修改。

5.3.4　作用域链

当在函数中访问变量时，Python 会按“局部(Local)→外层(Enclosing)→全局(Global)→内置(Built-in)”的顺序查找，形成作用域链(LEGB)。其中，内置作用域包含 Python 内置的函数和异常名，如 print、len、str、ValueError 等。这些名称在任何地方都可以直接使用，因为它们存在于内置模块__builtins__中。

【例 5-9】 作用域链举例。

```
global_var = "我是全局变量"
def outer_function():
    #3. 外层函数作用域(嵌套函数的外层)
    outer_var = "我是外层函数变量"
    def inner_function():
        #4. 局部作用域(当前函数内部)
        inner_var = "我是局部变量"
        #查找变量时,遵循作用域链: 先找局部,再找外层,再找全局,最后找内置
        print("在 inner_function 中访问: ")
        print(inner_var)              #局部作用域: 直接找到
        print(outer_var)              #外层函数作用域: 局部没有,向外层找
        print(global_var)             #全局作用域: 外层也没有,向全局找
        print(len([1, 2, 3]))         #内置作用域: 全局也没有,找内置函数
    #调用内层函数,触发变量查找
    inner_function()
#调用外层函数
outer_function()
```

运行结果:

```
在 inner_function 中访问:
我是局部变量
我是外层函数变量
我是全局变量
3
```

【代码分析】 inner_var 是 inner_function 内部定义的变量,优先被访问,outer_var 不在 inner_function 中,但在其外层函数 outer_function 中,因此会向外层查找,global_var 不在嵌套函数中,而是在模块级别定义,因此继续向全局查找。len()是 Python 自带的内置函数,当前三层作用域都没有时,最终会找到内置作用域中的定义。

5.4 嵌套调用与递归调用

5.4.1 函数的嵌套调用

Python 允许函数在其内部调用其他函数,这种方法称为函数的嵌套调用。通过这种方式,我们可以将一个复杂的任务分解为多个小的子任务,从而使代码更简洁、易维护。

【例 5-10】 比较两个数并返回较大值。

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

【代码分析】 在上面的 max()函数中,我们定义了一个简单的函数,它接收两个参数 x 和 y,并返回它们中的较大值。该函数使用了 Python 中的条件表达式 x if x>y else y。