第 5 章　函数与模块

在编程中，函数和模块是重要的代码组织方式。函数是将一段具有特定功能的、需要重复使用的代码段封装成有命名的代码块，函数通过执行这个命名的语句序列实现代码块的运行。模块来源于将复杂任务分解为多个子任务进行处理，这些子任务具有尽量彼此独立、简单可维护、可重复使用的特点，这些模块化的子任务就是模块，模块通过导入已有的模块文件进行使用。同时函数和模块也代表了一种封装，通过封装隐藏操作细节，使用者只需关注如何使用和结果，提高了编程效率。

本章学习目标：

- 能够熟练运用函数定义和调用语法。
- 能够熟练运用模块定义和调用语法。
- 掌握函数不同形式的参数传递的区别和用法。
- 理解函数和模块对代码组织和程序设计的意义。
- 了解构建复杂问题中函数、模块和包的使用。

5.1　函数和模块的定义

5.1.1　内置函数和内置模块

前面使用的 print()、type()函数是 Python 的内置(build-in)函数。Python 开发者定义了解决常见问题的函数集，内置在 Python 中，这些函数不需要我们预先定义就可以使用，被称为内置函数。

在之前的内容中我们已经见过内置函数直接调用的示例代码如下：

```
>>>type(10)
<type  'int'>
```

这个函数的名称是 type，括号里面的表达式为这个函数的参数，参数可以是值或者变量，作为函数的输入，type 函数的作用是显示参数的类型。通常函数会接收参数作为输入，然后返回一个结果，而这个结果被称为返回值。

例如，len 函数为一个内置函数，它能返回参数的长度。如果 len 函数的参数是一个字符串，那么该函数返回这个字符串的字符个数。len 函数不仅可得出字符串的长度，也可以操作其他数据结构类型。示例代码如下：

```
>>>s ='Hello world'
>>>len(s)
11
```

需要注意的是，可以将内置函数名类比为关键字，尽量避免另作他用，比如不要使用 len 作为变量名。内置函数可以通过帮助文档查阅获取具体信息。

模块是包含一组相关的函数文件。Python 中除了内置函数，还可以直接调用模块中已经定义好的函数。Python 中有一个数学计算的 math 模块，提供了大部分常见的数学函数，而更常见的一些数学函数则直接归入到内置函数中，示例代码如下：

```
>>>max('Hello world')
'w'
>>>abs(-2)
2
```

要使用 math 模块中的数学函数，需要先使用 import 语句导入该模块，示例代码如下：

```
>>>import math
>>>math
<moudle 'math' (built-in)>
```

math 为模块名字，访问 math 模块中的某个函数，需要同时指定模块名称和函数名称，用句点 . 分隔表示。例如，对三角函数的使用，示例代码如下：

```
>>>degrees =45
>>>radians =degrees / 180.0 * math.pi
>>>math.sin(radians)
0.707106781187
```

math.pi 是从 math 模块获取 π 的浮点近似值，大约精确到 15 位数字。math.sin (radians)是从 math 模块调用 sin()函数计算 radians 的正弦值，其中 degrees 代表角度值，radians 为弧度值。

Python 中除了 math 模块，还有其他提供各种不同功能集的模块，统称为 Python 语言内置标准模块库，详见 Python 标准库参考手册。这些模块已经预先安装，使用这些模块中的函数的步骤，和使用 math 模块的函数的步骤一致。而第三方模块库需要安装后才能使用。

5.1.2 自定义函数

除了使用 Python 自带的函数，我们也可以定义一个由自己设计的函数，这种函数叫作用户自定义函数。

定义一个函数，形式化表达为：

```
def functionname( 参数列表 ):
    """函数文档字符串"""
    函数体
    return 返回值
```

就是使用以下规则：函数代码块以 def 关键词开头，后接函数名和()；任何传入的参数

须放入()中;函数第一行语句可以写入文档字符串作为函数接口说明,函数比较简单时可以不写;函数内容以冒号起始,缩进;使用 return 表达式结束函数,返回一个值。当返回表达式为 return None,表示返回一个空值即无返回值,此时该表达式可以隐藏不写。

一般把函数定义的第一行称为函数头,其余部分为函数体。

【例 5-1】 举一个简单的例子。

```
def print_double():
    print("I don't like bug.")
    print("I like bug, but don't like debug.")
    return None            #表示无返回值,可以不写
```

def 为关键词,进行函数定义。这个函数名为 print_double,其中函数的命名规则和变量名相同。函数名后面的圆括号是空的,表示函数无任何参数输入。

如果在交互模式下输入函数定义,每空一行解释器就会打印三个句点,提示你定义并没有结束。但是函数定义尽量写在脚本文件中:

```
>>>def print_double():
...      print("I don't like bug.")
...      print("I like bug, but don't like debug.")
...
```

此时输入一个空行可以结束函数定义。定义一个函数会创建一个函数对象,类型为 function:

```
>>>type(print_double)
<class 'function'>
```

自定义函数创建结束,运行脚本,就可以在命令行交互模型下调用该函数,示例代码如下:

```
>>>print_double()
I don't like bug.
I like bug, but don't like debug.
```

再来看个例子,假设给定一个字符串,构建一个函数去统计这个字符串中每个字母出现的次数。具体实现的方法,可以建立一个字典,以字符为键,以统计计数为对应的值。当第一次遇到某个字符时,在字典中添加对应的键值对数据项,值为 1。如果字符已经存在,则键值对的数据项的值累加。对于这个函数来说,字符串是要传入处理的值,统计结果存放在字典中,因而函数返回值为该字典。

【例 5-2】 统计字符串中字符出现的次数。

```
def histogram(s):
    d =dict()
    for c in s:
```

```
        if c not in d:
            d[c] =1
        else:
            d[c] +=1
    return d
```

在函数的第一行创建一个空字典，for 循环遍历字符串，每次迭代中如果字符 c 不在字典中，新建一个键值对。如果 c 在字典中，c 键的值 d[c]增加 1。示例代码如下：

```
>>>h =histogram('brontosaurus')
>>>h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

字典有个 get 方法，接收一个键以及一个默认值。如果键出现在字典中，get 返回对应值；否则返回默认值。示例代码如下：

```
>>>h =histogram('a')
>>>h
{'a': 1}
>>>h.get('a', 0)
1
>>>h.get('b',0)
0
```

【例 5-3】 使用 get 方法可以优化例 5-2 的代码。

```
def histogram(s):
d =dict()
for c in s:
     d[c] =d.get(c,0) +1
return d
```

这里利用 get 方法消除了 if 语句。

5.1.3 自定义模块

Python 中既可以使用内置模块，也可以安装使用第三方模块，它们也称为标准库和第三方库，如果没有特指可统称为库。用户还可以自定义模块。自定义模块就是创建一个自定义的 .py 文件，这个文件就是模块，就是说任何包含 Python 代码的文件都可以作为模块导入使用。模块中的定义可以导入到其他模块中去使用。

例如，下面创建一个 fibo.py 模块，其中包含两个函数，函数 fib 产生一个最大值在 n 以内的斐波那契数列，并打印输出；函数 fib2 将产生的最大值 n 以内的斐波那契数列用返回值 result 返回。

【例 5-4】 创建 fibo 模块。

```
def fib(n):
    a, b =0, 1
    while a <n:
        print(a, end=' ')
        a, b =b, a+b
    print()
def fib2(n):
    result =[]
    a, b =0, 1
    while a <n:
        result.append(a)
        a, b =b, a+b
    return result
```

然后可以通过导入模块 fibo,实现对模块内定义的内容的访问,示例代码如下:

```
>>>import fibo
>>>fibo.fib(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>>f =fibo.fib2(100)
>>>print(f)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

另外,模块导入还可以使用 from 语句导入某个模块的部分内容。比如上面对模块 fibo 中的 fib 函数的导入,示例代码如下:

```
>>>from fibo import fib
>>>fib(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

也可以通过 from 导入模块的全部内容,使用 * 替代具体模块的内容。示例代码如下:

```
>>>from math import *
```

还可以为导入的对象取别名,示例代码如下:

```
>>>from fibo import fib as f
```

这样可以用 f 替代 fib 使用。

通过上面的例子,我们发现自定义模块定义完成后,导入使用的方式和前面内置模块是一样的。无论哪种模块,都是统一的导入使用方式,这样方便理解和使用。

5.1.4 模块内置属性和搜索路径

Python 为模块提供内置属性,完善模块相关功能支持,它们在__builtin__模块中定义。其中__name__和__all__属性比较常用,模块所有属性、方法列表可以使用 dir()函数查看。

__name__属性可以获取模块的名称，当程序启动时就会被设置。如果程序作为脚本执行，__name__的值为'__main__'；如果程序作为模块被导入，__name__的值为模块名。利用__name__的这个特性，可以设置模块的测试代码，而不担心模块调用产生的影响。

【例 5-5】 创建 wc.py 模块。

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

    print(linecount('wc.py'))
```

程序判断文件行数，最后一句作为测试，读取自身的内容，打印出文件的行数，结果为 7。但是当导入这个模块的时候，测试代码也会执行。比如：

```
>>> import wc
7
```

将最后一句改为如下模式：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

再次执行

```
>>> import wc
>>>
```

此时导入模块时不会执行测试代码，但是直接运行该脚本，测试代码会被运行。

这里经常会出现一个问题，比如 wc.py 模块文件保存在 C:\ds\python 目录下，不运行该模块而直接导入该模块：

```
>>> import wc
```

会提示 No module named 'wc'的错误信息，提示 wc 模块找不到。这是由于 Python 解释器在执行模块导入语句时，会去指定目录列表中搜索模块信息。可以通过调用标准库 sys 模块的 path 属性获取指定目录列表。当找不到模块时，可以将模块所在的目录添加到 sys.path 列表中，示例代码如下：

```
>>> import sys
>>> sys.path.append(r'C:\ds\python)
>>> import wc
>>>
```

就不会出现错误了，此时可以对比添加前后 sys.path 的输出结果，目录列表最后加入了 C:\ds\python 目录。当 import 导入模块找不到时，可以使用这种方法来解决。

__all__属性可用于模块导入时的限制，和 from <module> import * 一起连用时，如果定义了__all__属性，则只有__all__属性指定的属性、方法等会被导入。反之则全部导入。示例代码如下：

```
import math
def circle(r):
    return math.pi * r * r
__all__=['circle']
```

使用 from 导入代码如下：

```
>>>from md import *
>>>circle(4)
50.26548245743669
>>>print(math.pi)
```

代码运行结果表示无法找到 math 库：

```
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(math.pi)
NameError: name 'math' is not defined
```

此时执行 print(math.pi)出错，使用__all__属性定义后，这里只会导入 md 模块的 circle 方法，其他的不可见(下画线开头的成员除外)。可以接着调用 dir()函数查看当前的属性、方法列表：

```
>>>dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__
package__', '__spec__', 'circle']
```

这里面 math 没有出现，该结果不一定和上述一致，但是 math 不会出现(除非之前执行了 import math)。

5.2 函数详解

5.2.1 函数调用

程序代码的执行流程总是从第一句开始，自顶向下，逐次执行一条语句。其中函数的定义部分也会被执行，但是其作用仅限于创建函数对象。而函数内部语句在函数被调用之前，不会被执行。

那么真正要使用自定义函数，也就是需要让函数内部语句被执行，必须通过函数调用实

现函数体被执行。函数调用前必须先定义这个函数。也就是说，函数定义必须在函数第一次调用前。

函数调用使得执行流程绕了一个弯，执行流程此时会跳入函数体，首先开始执行函数体语句，然后再回到原来离开的位置，接着再执行下一条语句，如图 5-1 所示。

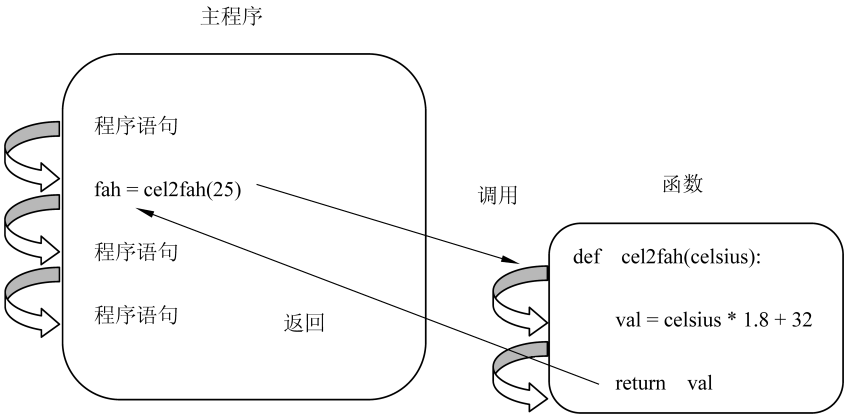

图 5-1　函数调用执行流程

Python 记录程序执行流程的位置，在执行函数调用完成时，程序会回到原来的位置。因此，阅读程序的时候，最好跟着执行流程阅读比较合理。

5.2.2　形参和实参

函数的使用分为两个部分：函数的定义和函数的调用。函数的参数有两种，函数定义里的为形参，调用函数时传入的为实参。形参和实参是有关联的，在函数体内部，实参会赋值给形参变量。也就是函数调用时会把实参的值赋给形参。

【例 5-6】 定义一个有两个形参的函数。

```
def add(x,y):
    s =x +y * 4
    return s
```

这个函数调用时，会将两个实参的值赋给形参 x 和 y，即形参主要是函数接收函数外部值，然后传入函数体内去处理，是函数的数据输入接口。函数调用时，所有能支持函数体语句执行的参数都是实参。示例代码如下：

```
>>>add(2,9)
38
>>>add('t','uo')
'tuouououo'
>>>x =5
>>>y =4.3
>>>add(x,y)
22.2
```

以上示例中，add 函数调用时，数值 2 和 9，字符串't'和'uo'，变量 x 和 y 都是实参。由于实参为某个值或者变量，所以也可以使用表达式来作为实参。示例代码如下：

```
>>>add(math.sin(math.pi),math.cos(math.pi))
-4.0
```

实参无论以何种方式或名字表达，其和形参名没有任何关系，实参传入函数后，函数体内部只会看到形参。

其中需要注意的是，实参是按实参对象的引用调用来传递，当实参如果是可变对象时，参数传递过程中，形参会变为实参对象的别名，则别名的变化会影响原实参。例如 ratio 函数是计算列表中各个数值的占比。

【例 5-7】 实参为可变对象。

```
def ratio(t):
    s =sum(t)
    for i in range(len(t)):
        t[i] =t[i] / s
```

下面使用该函数，

```
>>>ls =[1, 2, 3, 4]
>>>ratio(ls)
>>>ls
[0.1, 0.2, 0.3, 0.4]
```

这时函数 ratio 调用执行后，把实参 ls 也做了修改。如果希望 ls 的值不轻易被改动，则需要设法在函数内创建新列表替代原有列表，下面为其中一个方法，在 ratio 函数内通过切片复制一个新的列表 ct 进行修改。

【例 5-8】 对例 5-7 进行修改。

```
def ratio(t):
    ct =t[: ]
    s =sum(ct)
    for i in range(len(ct)):
        ct[i] =ct[i] / s
    return ct
```

5.2.3 函数的作用域和命名空间

函数本质上是一个新的程序，函数在开始执行时，建立了自己的命名空间。在函数内部创建的对象，将在函数的命名空间中命名，而不是在程序的命名空间中命名。作用域是相对于命名空间而言的，是命名空间在程序中的应用范围，不同的命名空间可以作为定义变量的作用域。只有在当前的作用域内的变量，才能在执行过程中引用。

函数在实参向形参的传递过程中，会受到不同作用域的影响，实参来自于函数外的程序

命名空间，而形参在函数不同的命名空间中，即使变量名一致的两个变量，也是完全不相同的。其中函数的命名空间是局部命名空间，函数中的变量为局部变量。比如在上一节的函数调用 add(x,y)中的实参 x 和 y 以及 add 函数定义中的形参 x 和 y，是不同的作用域中的变量。

函数中的变量是局部的，只能存在于函数内部。

【例 5-9】 函数局部变量示例。

```
def add2sin(num1,num2):
    num =num1 +num2
    print(math.sin(num))
```

然后调用函数，示例代码如下：

```
>>>n1 =5
>>>n2 =50
>>>add2sin(n1,n2)
-0.9997551733586199
>>>print(num)
NameError: name 'num' is not defined
```

当函数 add2sin 调用结束时，变量 num 被销毁，这里试图打印 num 会出错。

5.2.4 函数返回值

在以前的一些例子中，很多函数在调用结束后会返回一个值，例如 math 模块的数学函数，这类函数称为有返回值函数。而其他的一些函数，执行了一个打印动作但不返回任何值，这类函数称为无返回值函数。

当调用一个有返回值的函数时，可以把返回结果用在表达式里，或者赋值语句里。函数的返回值是通过 return 语句实现的。如以下的示例代码：

```
import math
x =math.sin(rs)
golden =(math.sqrt(5) +1)/2
```

在交互模式下调用有返回值函数，Python 解释器会马上显示结果：

```
>>>math.sqrt(34)
5.830951894845301
```

但是在脚本中，会没有任何结果显示。

无返回值函数会执行某个过程，类似打印输出或者其他操作，但是它们并不是没有返回值。无返回值函数实际上会返回一个 None 值。