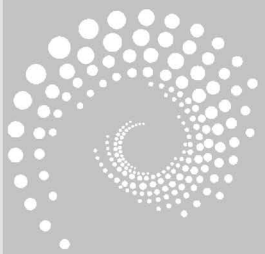
第5章

函数与模块

CHAPTER 5



本章学习目标

- 掌握函数的定义、使用和函数的返回值。
- 掌握 lambda 表达式及其使用方法。
- 理解函数的参数传递、变量的作用域。
- 理解函数的递归及其使用方法。
- 了解模块、包及程序的模块化。

函数是一段可以重复使用的代码段，用来独立地完成某个功能，它可以接收用户传递的数据，也可以不接收。接收用户数据的函数在定义时要指明参数，不接收用户数据的不需要指明，由此可以将函数分为有参函数和无参函数。将代码段封装成函数的过程叫作函数定义。本章主要介绍函数定义与调用、参数传递、参数类型、变量作用域、递归函数、函数应用、模块与包等内容。掌握这些内容，可以把一些完成特定功能的模块编写成自定义函数，然后通过调用这些函数来完成相应功能，提高编程效率。

5.1 函数定义与调用

函数是实现模块化程序设计的基本构成单位。在 Python 中，函数分为内置函数、标准库函数、第三方提供的函数和自定义函数等 4 类。

(1) 内置函数，这类函数在程序中可以直接使用，内置函数运行速度快，建议程序设计时应尽量使用，所有内置函数可以在 IDLE 环境中使用 dir()函数来查看。

(2) 标准库函数，这类标准库函数非常庞大，Windows 版本的 Python 安装程序通常包含整个标准库，所以不需要单独安装，使用前用 import 命令导入相应模块即可使用。常用的标准库有 math 库、random 库、os 库和 datetime 库等。

(3) 第三方提供的函数，第三方为 Python 提供了很多扩展库，这些库中的函数或模块在使用前需要下载安装，正确安装之后才可以使用。

(4) 自定义函数，在程序设计过程中，用户可以将一些常用计算定义为函数，一旦定义该函数对象后，就可以反复调用，既提高了软件复用的效率，又提高了代码的质量。Python 本身就是函数式编程，在程序设计过程中建议多使用自定义函数。

5.1.1 函数定义

在 Python 中，函数要先定义后使用。定义一个函数要使用 def 语句，依次写出函数名、一对圆括号、圆括号中的参数以及冒号；然后，在缩进块中编写函数体，函数返回值用 return 语句返回。

函数定义的语法格式：

```
def 函数名(参数 1,参数 2,…):
    函数体语句块
    [return[表达式 1[,表达式 2[, …]]]]
```

说明：

(1) def 为定义函数的关键字，不可缺少。

(2) “[]”表示该项为可选项。

(3) 函数名为符合命名规则的标识符，由用户自定义。

(4) 函数名后紧跟一对圆括号“()”，其中可以有若干个参数，参数间用“,”分隔；也可以没有参数。这里的参数称为形式参数，简称为形参。

(5) 冒号“:”表示函数体的开始，不可缺少。这里为英文半角冒号。

(6) 函数体中的语句要缩进，通常是 4 个字符。注意缩进要一致，即左对齐。

(7) return 语句为可选项，其作用是：函数执行到 return 语句时，停止本函数的执行，并返回到调用程序；其中的表达式也可以是具体值。在 Python 中，一条 return 语句可以同时返回 0 到多个值。

【例 5-1】 自定义一个 fib(n)函数，生成并打印输出斐波那契数列前 n 项。

分析：斐波那契数列(Fibonacci Sequence)又称黄金分割数列，是数学家列昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例而引入的，故又称为“兔子数列”，指的是数

列 1、1、2、3、5、8、13、21、34、55、89、144、……，即前两项都是 1，从第三项开始，每一项都等于前两项之和。

数学上递推定义：F(1)＝1，F(2)＝1，F(n)＝F(n－1)＋F(n－2)(n≥3)。

代码如下：

```
def fib(n):
    a,b = 1,1
    print(a,b,end = " ")
    for m in range(3,n + 1):
        a,b = b,a + b
        print(b,end = " ")
```

5.1.2　函数调用

当在任务中每次需要完成某一步的功能时，如果已有该功能函数的定义，则只需调用一次对应的函数即可。

函数调用的语法格式：

```
函数名(实参 1,实参 2,…)
```

说明：

(1) 函数名是已经定义的函数名。

(2) 圆括号中是参数列表，可以有若干个参数，用逗号分隔，也可以没有参数。这里的参数称为实际参数，简称为实参。

下面的例子是对已经定义过的 myinput()函数的调用。

```
>>> def myinput():
...     num = input("请输入一个整数:")
...     print(f'您输入的数是:{num}')
>>> myinput()
请输入一个整数: 5
您输入的数是: 5
```

要先对 myinput()函数进行定义，才能对其进行调用。该程序的执行过程为：首先执行函数调用语句 myinput()，然后转到 myinput()函数的定义处开始执行函数体语句，获取输入并输出；函数体语句执行完成后，本次函数调用结束，回到函数调用语句处：myinput()，程序运行结束。函数定义用来说明函数的参数、功能及返回值，如果函数没有被调用，函数就不会被执行。

【例 5-2】 定义一个函数，计算 1～100 所有整数的和并输出结果。

分析：按题目要求，通过函数来计算 1 到 100 的整数和，运用函数定义、调用的相关知识即可实现。代码如下：

```
def calSum():                    # 函数定义
    start = 1
    end = 100
```

```
    sum = 0
    while start <= end:
        sum += start
        start += 1
    print(f'1 + 2 + … + 100 = {sum}')
calSum()                    # 函数调用语句,调用 calSum()函数
```

运行结果：

```
 ================================
1 + 2 + … + 100 = 5050
```

凡是需要计算 1～100 的整数和并输出结果的地方都可以调用 calSum()函数，可达到一次定义、多次使用的目的。程序首先执行第 9 行的函数调用语句，转而执行第 1 行开始的函数体语句，第 8 行执行完后，函数调用过程结束，回到第 9 行结束程序运行。

5.1.3 函数返回值

在程序中，一个函数结束运行前，要返回一个值给调用程序，调用程序根据被调用函数的返回值作出相应的处理。

Python 中，函数的返回值是通过 return 语句完成的。

格式：

```
return[表达式 1[,表达式 2[,…]]]
```

说明：

(1) “[]”表示该项为可选项。

(2) 使用 return 语句结束当前函数的执行，返回到调用程序。

(3) 在一条 return 语句中可返回 0 到多个值给调用程序。

(4) 一个函数中若无 return 语句，则无返回值，函数结果为 None。

(5) 一个函数中若有 return 语句，且有返回值的表达式，则有返回值，函数结果就是表达式的值。

(6) 一个函数中若有 return 语句，但无表达式，则无返回值，函数结果为 None。

函数返回值有以下两种常用用法。

1. return 结束函数执行

此种用法常用于函数需要在某个位置结束执行，但不需要返回数据给调用程序时。例如：

```
>>> def f2():
...     a = 1
...     b = 2
...     print(f'{a} * {b} = {a * b}')
...     return                          #此处结束函数执行,返回
```

```
...     print(f'{a} * {b} = {a * b}')     # 该语句不会得到执行
>>> f2()
1 * 2 = 2
```

2. return 结束函数执行并返回值

在函数需要将值返回给调用程序的地方使用此种方式，调用程序可以用一个变量接收该返回值，这是较为常用的一种返回方式。例如：

```
>>> def f3():
...     a = 1
...     b = 2
...     ret = a * b
...     return ret                    # 函数定义结束
>>> res = f3()                        # 调用 f3()函数，变量 res 用于接收 f3()函数的返回值
>>> print(res)                        # 输出 res 的值
2
```

【例 5-3】 定义一个函数，计算 1～100 的和，将计算结果返回。

分析：例题要求通过函数返回计算结果，要使用 return 语句将 1～100 的和返回。

代码如下：

```
def getSum():
    start = 1
    end = 100
    sum = 0
    while start <= end:
        sum += start
        start += 1
    return sum                         # 循环结束，返回 sum 的值
res = getSum()                         # 调用 getSum()函数，res 接收函数返回值
print(f'1 + 2 + … + 100 = {res}')      # 输出 res 的值
```

运行结果：

```
================================
1 + 2 + … + 100 = 5050
```

5.1.4　匿名函数

在 Python 中，不仅可以定义普通的函数，即用 def 关键字定义的函数，也可以定义匿名函数。所谓匿名函数，就是没有函数名称的函数。

Python 中的匿名函数是通过 lambda 表达式实现的，常用来表示内部仅包含 1 行表达式的函数。当函数比较简单时，可以使用 lambda 表达式进行简洁表示，以便提高程序的性能。

格式：

```
lambda 参数列表:表达式
```

说明：

(1) 匿名函数没有函数名称。

(2) 使用 lambda 关键字创建匿名函数。

(3) 匿名函数冒号后面的表达式有且只有一个。

(4) 匿名函数自带 return 语句，而 return 语句返回的结果就是表达式的计算结果。

例如，用如下普通函数实现的功能：

```
def f(x,y):                          #定义 f()函数,有 x 和 y 两个参数
    z = x * y                        #计算 x * y 的结果
    return z                         #返回计算结果
```

可以用匿名函数实现，具体如下：

```
>>> lambda x,y:x * y                 #x、y 部分对应参数列表,冒号后 x * y 为表达式,即返回值
```

相比普通函数而言，匿名函数借助 lambda 表达式实现，可以省去定义函数的过程，使代码更加简洁。同时，对于不需要多次复用的函数，使用 lambda 表达式可以在用完之后立即释放，提高程序执行的性能。在使用匿名函数时，可以把 lambda 表达式赋给一个变量，此变量是一个函数对象，相当于匿名函数指定了一个函数名。例如：

```
>>> s = lambda x,y:x * y             #lambda 表达式赋给变量 s,s 是一个函数对象
>>> s(2,3)                           #通过变量 s 调用匿名函数获取结果
6
```

在使用匿名函数时，允许使用默认值参数和可变长度参数。例如：

```
>>> b = lambda x,y = 2:x + y         #参数 y 的默认值为 2
>>> b(1)
3
>>> b(1,3)
4
>>> b = lambda  * z:z                #参数 z 为可变长度参数
>>> b(10,'test')
(10, 'test')
```

【例 5-4】 用匿名函数实现对传入的参数求平方。

分析：通过 lambda 表达式实现对匿名函数求平方的功能，调用匿名函数时输入待计算的数据，使用 print()函数输出结果。代码如下：

```
f = lambda x:x ** 2
print(f(5))
```

运行结果：

```
 ================================
25
```

5.1.5　嵌套函数

Python 支持嵌套函数。嵌套函数是指函数内定义了另外一个函数，内层函数不能被外部直接使用，只能在外层定义它的函数中使用，否则会抛出异常。

内层函数可以访问外层函数中定义的变量，但不能重新赋值。

嵌套函数示例 1 ：

```
>>> def func1():
…       def func2():
…           print("this is func2")
…       return func2          # 调用 func1()函数将 func2()函数对象返回给调用者
>>> res = func1()             # 调用 func1()函数，返回了 func2()函数的返回对象，res = func2
>>> res()                     # 调用的就是 func2()函数
this is func2                 # 打印结果
```

嵌套函数示例 2：

```
>>> def fdemo1(x,y):
...      def fdemo2(z):
...          return x + y * z
...      return fdemo2        # 调用 fdemo1()函数将 fdemo2()函数对象返回给调用者
>>> abc = fdemo1(10,20)      # 调用 fdemo1()函数，返回了 fdemo2()函数的返回对象，abc = fdemo2
>>> abc(30)                  # 调用的就是 fdemo2()函数
610
```

5.2　参数传递

5.2.1　形式参数和实际参数

定义函数时所声明的参数，即为形式参数，简称为形参。调用函数时，提供函数所需参数的实际值，即为实际参数，简称为实参。

函数定义时圆括号内为若干个用逗号分隔的形参。一个函数可以没有形参，但是圆括号必须保留，表示该函数是无参函数，不接受参数。

函数调用时向形参传递对应的实参，也就是将实参的值或引用传递给对应的形参。实参值默认按位置顺序依次传递给形参。如果实参个数不对，会产生错误。

函数定义时表明的形式参数，等同于函数体中的局部变量，在函数体中的任何位置都可以使用。局部变量和形式参数变量的区别在于，局部变量在函数体中绑定到某个对象；而形式参数变量则绑定到函数调用代码传递的对应实际参数对象。

Python 参数传递方法是传递对象引用，而不是传递对象的值。传递对象引用又可分为传递不可变对象的引用和传递可变对象的引用。

5.2.2 传递不可变对象的引用

函数调用时，如果传递的是不可变对象（例如数字型、字符串、元组等），且函数体中修改不可变对象值，其实质是创建一个新的不可变对象。

【例 5-5】 传递不可变对象的引用示例。

```
x = 100
y = 200
def abc(m,n):
    m += n
    n += m
    return n
y = abc(x,y)
print(x,y)
```

运行结果：

```
================================
100 500
```

本示例中，x 的初始值为 100，y 的初始值为 200；调用 abc(x,y)函数后，在函数体内执行“x＋＝200”，函数体内 x 的值为 300，执行“y＋＝300”后，函数体内 y 的值为 500，并返回函数值 500；abc(x,y)函数调用结束后，不可变对象 x 的值仍为 100；不可变对象 y 被重新赋值为 500，等价于新建一个不可变对象 y。

5.2.3 传递可变对象的引用

函数调用时，如果传递的是可变对象（例如列表、集合、字典等）的引用，则函数体中可以直接修改可变对象的值。

【例 5-6】 传递可变对象的引用示例。

```
list1 = [1,2,3,4,5,6]
def exchange(lst,m,n):
    lst[m],lst[n] = lst[n],lst[m]
exchange(list1,2,4)
print(list1)
```

运行结果：

```
================================
[1, 2, 5, 4, 3, 6]
```

函数调用时，如果传递给函数的是可变序列，并且在函数内部使用下标或可变序列自身的方法增加、删除元素或修改元素时，修改后的结果是可以反映到函数之外的，实参也得到相应的修改。

5.2.4　序列解包参数传递

函数调用时，若为多个形参传递参数时，可以使用 Python 元组、列表、集合、字典等可迭代对象作为实参，并在实参前加一个星号，Python 解释器自动将其解包，然后传递给多个形参。字典对象作为实参时，默认使用字典的“键”；如果需要使用字典中“键-值”作为参数，则需要使用字典 item()方法；如果需要使用字典中“值”作为参数，则需要使用字典 values()方法。

【例 5-7】 序列解包参数传递示例。

```
def fdemo(x,y,z):print(x,y,z);print(x+y+z)
list1=[1,2,3]
fdemo(*list1)4
dict1={"x":"AA","y":"BB","z":"CC"}
fdemo(*dict1)7
fdemo(*dict1.values())9
fdemo(*dict1.items())
```

运行结果：

```
 ================================
1 2 3
6
x y z
xyz
AA BB CC
AABBCC
('x', 'AA') ('y', 'BB') ('z', 'CC')
('x', 'AA', 'y', 'BB', 'z', 'CC')
```

5.3　参数类型

Python 中，函数参数可以分为位置参数、关键参数、默认参数、可变参数等。

Python 在函数定义时不需要指定形参的类型，其类型是由函数调用传递的实参类型以及 Python 解释器的理解和推断来决定的，类似于函数重载。

Python 函数定义时也不需要指定函数的数据类型，其类型由函数中的 return 语句来决定，如果没有 return 语句或 return 语句没有得到执行，则认为返回空值 None。

5.3.1　位置参数

位置参数是指函数调用时，实参默认按位置顺序传递形参。位置参数是较常用的参数类型，调用函数时实参和形参的顺序必须严格一致，并且实参与形参的数量必须相同。其格式如下：

```
函数定义：def 函数名(形参 1,形参 2,…,形参 n):
函数调用：函数名(实参 1,实参 2,…,实参 n)
```

说明：

(1) 要求函数调用时实参个数要与形参个数相同。

(2) 对应参数的顺序要一致。要传递给形参 1 的实参只能放在实参 1 的位置，要传递给形参 2 的实参只能放在实参 2 的位置，以此类推。

【例 5-8】 位置参数示例。

```
def fdemo(x,y,z):
    print(x,y,z)
fdemo(1,2,3)
fdemo("66","55","44")
```

运行结果：

```
================================
1 2 3
66 55 44
```

调用 fdemo(1,2)函数和 fdemo(1,2,3,4)函数都会显示出错信息。

5.3.2 关键参数

在 Python 中，解释器可以根据参数名找到传递过来的参数值，函数调用时按形参的名字传递实参，即为关键参数。使用关键参数可以更灵活地传递参数，不要求实参与形参的顺序一致。其格式如下：

```
函数定义：def 函数名(形参 1,形参 2,…,形参 n):
函数调用：函数名(形参 1 = 实参 1,形参 2 = 实参 2,…,形参 n = 实参 n)
```

说明：

(1) 关键参数传递参数时，要求实参个数与形参个数一致。

(2) 通过形参名指定要为哪个形参传递值。

(3) 通过关键参数，实参顺序可以和形参顺序不一致，但不影响传递结果，避免用户需要牢记位置参数顺序的麻烦。

【例 5-9】 关键参数示例。

```
def fdemo(x,y,z):
    print(x,y,z)
fdemo(x = 3,z = 5,y = 4)
fdemo(z = "AA",x = "BB",y = 123)
```

运行结果：

```
================================
3 4 5
BB 123 AA
```

5.3.3　默认参数

默认参数是指函数定义时为形参设置默认值。

带有默认参数的函数定义语法格式：

```
def 函数名(形参 1[ = 默认值 1],形参 2[ = 默认值 2]…,形参 n[ = 默认值 n]):
    函数体
```

说明：

(1) “[]”表示该项为可选项。

(2) 在形参处给出默认值。

(3) 形参默认值的设置应当遵循由后向前的顺序设置。即默认参数必须出现在函数参数列表的最右端，且任何一个默认参数右边不能有非默认参数。

例如，deffdemo(x=3,y,z=10)和 def fdemo(x=3,y)都会导致函数定义失败。

(4) 调用带有默认参数的函数时，可以不给默认参数传递实参。若是没有传递实参，则使用定义时的默认值；若是传递了实参，则使用传递的实参值。

【例 5-10】 默认参数示例。

```
def fdemo(x,y,z = 10):
    print(x,y,z)
fdemo(1,2)
fdemo(1,2,3)
```

运行结果：

```
 ================================
1 2 10
1 2 3
```

【例 5-11】 设计一个函数，能够计算任意给出的两个整数 s 和 e(s <= e)之间(包括 s 和 e)所有整数之和。如果未给出第 1 个整数 s，则使用默认值 1；如果未给出第 2 个整数 e，则使用默认值 100。

分析：函数需要两个参数，并且均需要指定默认值，最后函数返回两个参数之间的所有整数之和。代码如下：

```
def sum_default(s = 1,e = 100):          #给形参 s 设置默认值为 1,形参 e 设置默认值为 100
    sum = 0                              #sum 变量初始化为 0
    while s < = e:                       #while 循环及其循环条件 s < = e
        sum += s                         #sum 的值加上循环变量 s 的当前值
        s += 1                           #s 的值加 1,为下一次循环做准备
    return sum                           #返回 sum 的值,即从 s 到 e 的总和
st = 2
ed = 101
#调用函数,未给出实参,则参数分别使用默认值 1 和 100
res = sum_default()                      #变量 res 接收函数返回值
print(res)
#调用函数,将实参 st 的值 2 传递给形参 s,实参 ed 的值 101 传递给形参 e
```

```
res = sum_default(st,ed)                    # 变量 res 接收函数返回值
print(f"{st} + {st + 1} + … + {ed} = {res}")
```

运行结果：

```
 ================================
5050
2 + 3 + … + 101 = 5150
```

5.3.4 可变参数

可变参数是指函数定义时标识带星(*)的参数，从而函数调用时允许向函数传递可变数量的实参。

带有可变参数的函数定义语法格式：

```
def 函数名([形参 1,形参 2,…,] * 形参 n):
```

说明：

(1)“[]”表示该项为可选项。

(2)声明一个参数为可变参数需要在变量名前用“*”表示。

(3)可变参数可以看成系统根据实参个数自动生成一个元组。

(4)可变参数只能是参数列表中最后一个参数。

可变参数主要包括以下两种形式。

(1) * parameter：接收多个实参并存放在一个元组中。

(2) ** parameter：接收多个关键参数并存放到一个字典中。

【例 5-12】 可变参数示例。

```
>>> def fdemo( * p):
…      print(p)
>>> fdemo(1,2,3)
(1, 2, 3)

>>> def fdemo( ** p):
…      print(p)
>>> fdemo(x = 1,y = 2,z = 3)
{'x': 1, 'y': 2, 'z': 3}

>>> def fdemo(m,n = "AA", * p, ** q):
…      print(m,n)
…      print(p)
…      print(q)
>>> fdemo("AA","BB",11,22,33,x = "11",y = "22",z = "33")
AA BB
(11, 22, 33)
{'x': '11', 'y': '22', 'z': '33'}
```

注意：调用函数时如果对可迭代对象实参使用一个星号(*)进行序列解包，则这些解包后的实参将被作为普通位置参数对待，并且在关键参数和使用两个星号(**)可变参数进行序列解包的参数之前进行处理。

5.4　变量作用域

Python 中，一个变量除了数据类型和取值外，还有一个重要的属性就是其作用域。所谓变量作用域，就是变量的有效范围，即变量可以使用的范围。

5.4.1　Python 作用域

Python 作用域一共包括 4 种：局部作用域、嵌套父级函数的局部作用域、全局作用域、内置作用域。

(1) 局部作用域：作用于函数定义所在范围。

(2) 嵌套父级函数的局部作用域：作用于嵌套的父级函数定义所在范围，即作用于包含此函数的上级函数定义所在范围，是一种局部作用域。内层函数引用外层函数的变量，形成闭包。

(3) 全局作用域：作用于函数定义所在模块范围。

(4) 内置作用域：作用于 Python 内置模块范围。

在作用域中搜索变量的优先级顺序为：局部作用域＞嵌套父级函数的局部作用域＞全局作用域＞内置作用域。

一个变量在函数外部定义和在函数内部定义，其作用域是不同的。函数定义中，变量按其作用域，可分为全局变量和局部变量两种。

局部变量的引用比全局变量速度快，应优先考虑使用。

5.4.2　局部变量

局部变量是指在函数内部定义的变量(包括形参)，其有效范围(作用域)为函数体(函数内部)。当函数执行结束后，局部变量自动删除，不可以再使用。

【例 5-13】 局部变量示例。

```
>>> def fdemo(x,y):                    #定义函数,形参 x 和 y 是局部变量
...     z = x + y                      #z 是函数内部局部变量
...     print(x,y,z)
...     print(locals())
>>> fdemo(100,200)                     #调用函数
100 200 300
{'x': 100, 'y': 200, 'z': 300}
>>> x                                  #x 是 fdemo()函数中的局部变量,函数执行结束后自动删除
NameError: name 'x' is not defined
```

5.4.3　全局变量

全局变量是指模块在函数定义之外声明的变量，可在全局作用域中使用，即在模块中所

有函数定义外使用。

如果在一个函数中定义的局部变量(包括形参)与全局变量重名,则局部变量优先,即函数中定义的变量是指局部变量,而不是全局变量。

【例 5-14】 全局变量示例。

```
>>> z = 100                          # z 是全局变量
>>> def fdemo(x,y):                  # 定义函数,形参 x 和 y 是局部变量
...     z = x + y                    # z 是同名的局部变量
...     print(x,y,z)

>>> fdemo(200,300)                   # 调用函数
200 300 500                          # 优先返回局部变量值

>>> z                                # 函数调用结束,z 为全局变量
100                                  # 返回全局变量值
```

5.4.4 全局语句 global

如果想要在函数内部给一个定义在函数外的变量赋值,那么这个变量就不能是局部的,其作用域必须为全局的,能够同时作用于函数外,称为全局变量,可以通过 global 来定义。

global 是 Python 中的一个保留字,用来显式声明一个变量为全局变量。

global 定义全局变量时,可分为以下两种情况。

(1) 一个变量已在函数外定义,如果在函数内需要为这个变量赋值,并要将这个赋值结果反映到函数外,可以在函数内用 global 声明这个变量,将其声明为全局变量。

(2) 一个变量在函数外没有声明,在函数内部直接将一个变量声明为全局变量,该函数执行后,将增加为新的全局变量。

【例 5-15】 全局语句 global 示例。

```
>>> def fdemo(x,y):                  # 定义函数
...     global z                     # 显式定义 z 为全局变量
...     z = x + y
...     print(x,y,z)
>>> fdemo(200,300)                   # 调用函数
200 300 500
>>> z                                # z 为全局变量
500
```

5.4.5 非局部语句 nonlocal

在函数体中,可以定义嵌套函数。在嵌套函数中,如果要为定义在上级函数整体的局部变量赋值,可使用 nonlocal 语句,表明变量不是所在块的局部变量,而是在上级函数体中定义的局部变量。nonlocal 语句可以指定多个非局部变量。例如,nonlocalx,y,z。

【例 5-16】 非局部语句 nonlocal 示例。

```
>>> def outf():                      # 定义 outf()函数
...     s = 100                      # s 是 outf()函数的局部变量
```

```
…      print(s)
…      def inf():          #定义 inf()函数
…          nonlocal s      #显式定义 s 为非 inf()函数局部变量,是上级 outf()函数定义的局部变量
…          s = 200
…          print(s)
…      inf()
…      print(s)
>>> outf()                 #调用 outf()函数
100
200
200
>>> s                      #s 是 outf()函数的局部变量,函数执行结束后自动删除
NameError: name 's' is not defined
```

5.5　递归函数

函数内部可以调用其他函数,如果一个函数在内部调用自身,则该函数是递归函数。

5.5.1　递归函数的定义

递归函数即自调用函数,在函数体内部直接或间接地自己调用自己,即函数的嵌套调用是函数本身。

【例 5-17】 计算 n!的递归函数。

分析:正整数的阶乘(factorial)是所有小于和等于该数的正整数的乘积。自然数 n 的阶乘写作 n!,n!=1×2×3×…×n。阶乘的递归定义:0!=1,n!=(n−1)!×n。代码如下:

```
def factorial(n):                       #函数定义
    if n == 0:
        return 1
    return factorial(n - 1) * n
for x in range(10):
    print(x,"! = ",factorial(x))        #调用函数
```

运行结果:

```
================================
0 ! = 1
1 ! = 1
2 ! = 2
3 ! = 6
4 ! = 24
5 ! = 120
6 ! = 720
7 ! = 5040
8 ! = 40320
9 ! = 362880
```

5.5.2 递归函数的原理

递归函数执行过程中将反复调用其自身，每调用一次就进入新的一层。因此，递归函数必须有结束条件，当函数在一直递推，直到遇到墙后返回，这个墙就是结束条件。

递归函数包括两个要素：终止条件与递推关系。

(1) 终止条件。终止条件用于结束递归，返回函数值。例 5－17 中 factorial()函数的终止条件是 n＝＝0。缺少终止条件的递归函数，将会导致无限递归函数调用，其最终结果是系统会耗尽内存，抛出 RecursionError。

【例 5-18】 对于用户输入的字符串 s，输出反转后的字符串。

```
def reverse(s):
    return reverse(s[1:]) + s[0]
print(reverse("ABC"))
```

运行结果：

```
 ================================
Traceback (most recent call last):
  File "E:/python 程序/例 5 - 18.py", line 3, in < module >
    print(reverse("ABC"))
  File "E:/python 程序/例 5 - 18.py", line 2, in reverse
    return reverse(s[1:]) + s[0]
  File "E:/python 程序/例 5 - 18.py", line 2, in reverse
    return reverse(s[1:]) + s[0]
  File "E:/python 程序/例 5 - 18.py", line 2, in reverse
    return reverse(s[1:]) + s[0]
  [Previous line repeated 1022 more times]
RecursionError: maximum recursion depth exceeded
```

错误表明该函数没有终止条件，递归层数超过了系统允许的最大深度。该函数体的完善代码如下：

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]
print(reverse("ABC"))
```

运行结果：

```
 ================================
CBA
```

在 Python 中递归的层数默认限制在 1000 层，虽然可以将递归的层数修改得大一些，但是建议在程序中不要使用太深的递归层数。

(2) 递推关系。相邻两次递归步骤之间有紧密的联系，前一次要为后一次做准备，也就是将第 n 步的参数值的函数与第 n－1 步的参数值的函数关联。而且每次进入更深一层递

归时，问题规模相比上次递归都应有所减少，保证收敛。否则，也会导致无限递归函数调用。

递归调用的执行过程分为递推过程和回归过程两部分。这两个过程由递归终止条件控制，即逐层递推，直至递归条件终止，然后逐层回归。递归调用同普通的函数调用一样利用了先进后出的栈结构来实现。每次调用时，在栈中分配内存单元保存返回地址以及参数和局部变量；而与普通的函数调用不同的是，由于递推的过程是一个逐层调用的过程，因此存在一个逐层连续的参数入栈过程，调用过程每调用一次自身，把当前参数压栈，每次调用时都首先判断递归终止条件。直到达到递归终止条件为止；接着回归过程不断从栈中弹出当前的参数，直到栈空返回到初始调用处为止。

例如，递归调用 3!的执行过程，如图 5-1 所示。

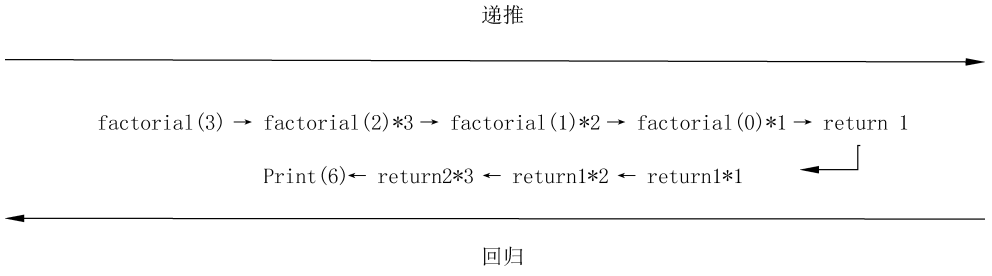


图 5-1　递归调用 3!的执行过程

5.5.3　递归函数实例

【例 5-19】　计算最大公约数的递归函数。

分析：最大公约数是指两个或多个整数共有约数中的最大数，又称最大公因、最大公因子。a 和 b 的最大公约数记为 gcd(a,b)，a、b 和 c 的最大公约数记为 gcd(a,b,c)，多个整数的最大公约数也有同样的记号。最大公约数有多种求解方法，常见的包括辗转相除法、质因数分解法、短除法和更相减损法。其中，辗转相除法又称欧几里得算法，主要依赖于定理：gcd(a,b)＝gcd(b,a mod b)。

计算最大公约数的递归函数：

(1) 终止条件：当 b＝0 时，gcd(a,b)＝a。

(2) 递推关系：gcd(b,a%b)。

代码如下：

```
def gcd(x,y):
    if y == 0: return x
    return gcd(y,x % y)
```

运行结果：

```
 ================================
gcd(36,60)
12
```

【例 5-20】　实现汉诺塔的递归函数。

分析：汉诺塔问题源于印度一个古老传说的益智玩具。该问题描述的是一张桌面上有

三个柱子 X、Y 和 Z。X 柱子上套有 64 个大小不等的圆盘，大的在下，小的在上。汉诺塔问题示意图如图 5-2 所示。

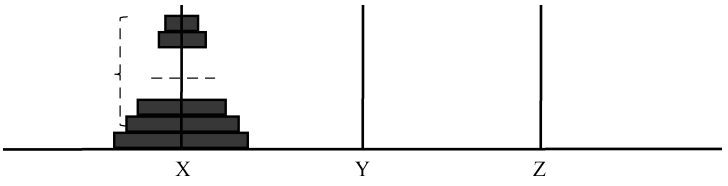


图 5-2　汉诺塔问题示意图

汉诺塔问题要求把这 64 个圆盘从 X 柱移动到 Z 柱上，每次只能移动一个圆盘，移动可以借助 Y 柱进行。但在任何时候，任何柱上的圆盘都必须保持大盘在下，小盘在上的特点。

实现汉诺塔的递归函数：

(1) 终止条件：当 n＝＝1 时，ht(n,x,y,z)＝ht(1,x,y,z)。如果起始柱 X 只有一个圆盘，则可以直接将其移动到目标柱 Z 上。

(2) 递推关系：ht(n,x,y,z)可以分解成 ht(n－1,x,z,y)、ht(1,x,y,z)和 ht(n－1,y,x,z)三个步骤。

代码如下：

```
def ht(n,x,y,z):
    if n == 1:
        print(x," ==>",z)
    else:
        ht(n - 1,x,z,y)
        ht(1,x,y,z)
        ht(n - 1,y,x,z)
n = int(input("请输入汉诺塔的圆盘数:"))
ht(n,"X","Y","Z")
```

运行结果：

```
================================
请输入汉诺塔的圆盘数: 3
X ==> Z
X ==> Y
Z ==> Y
X ==> Z
Y ==> X
Y ==> Z
X ==> Z
```

5.6　函数应用

【例 5-21】 定义一个函数，能够接收任意多个实数，并能返回一个元组，其中第一个元素为所有参数的平均值，其他元素为所有参数中大于平均值的实数。

```
def fdemo( * para):
    avg = sum(para)/len(para)
    tavg = [n for n in para if n > avg]
    return(avg,) + tuple(tavg)
print(fdemo(10,20,30,40,50))
```

运行结果：

```
 ================================
(30.0, 40, 50)
```

【例 5-22】 定义一个函数，能够接收字符串参数，返回一个列表，其中第一个元素为大写字母个数，第二个元素为小写字母个数，第三个元素是数字个数。

```
def fdemo(str1):
    result = [0,0,0]
    for ch in str1:
        if "A"< = ch < = "Z":
            result[0] += 1
        elif "a"< = ch < = "z":
            result[1] += 1
        elif ch.isdigit():
            result[2] += 1
    return result
print(fdemo("ABCDabcde123456"))
```

运行结果：

```
 ================================
[4, 5, 6]
```

【例 5-23】 定义一个函数，能够接收包含 15 个整数的列表 list1 和一个整数 n 作为参数。处理规则：列表 list1 中下标 n 之前的元素逆序打印输出；下标 n 及其之后的元素逆序打印输出；整个列表 list1 中的所有元素再逆序返回。

```
def fdemo(list1,n):
    x = list1[:n]
    x.reverse()
    print(x)
    y = list1[n:]
    y.reverse()
    print(y)
    z = list1
    z.reverse()
    return z
list2 = list(range(1,16))
print(fdemo(list2,6))
```

运行结果：

```
 ================================
[6, 5, 4, 3, 2, 1]
```

```
[15, 14, 13, 12, 11, 10, 9, 8, 7]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

【例 5-24】 定义一个函数，能够接收整数参数 n，返回斐波那契数列中大于 n 的第一个数前的数列，以及斐波那契数列中大于 n 的第一个数。

```
def fdemo(n):
    a,b = 1,1
    print(a,b,end = " ")
    while b < = n:
        a,b = b,a + b
        print(b,end = " ")
    else:
        return b
print(fdemo(50))
```

运行结果：

```
 ================================
1 1 2 3 5 8 13 21 34 55 55
```

【例 5-25】 定义一个函数，能够接收整数参数 n，判断 n 是否为素数。

```
import math
def isprime(n):
    if n == 1: return(str(n) + "非素数!")
    for x in range(2,int(math.sqrt(n)) + 1):
        if n % x == 0: return(str(n) + "非素数!")
    return(str(n) + "是素数!")
print(isprime(17))
print(isprime(27))
```

运行结果：

```
 ================================
17 是素数!
27 非素数!
```

【例 5-26】 定义一个函数，能够接收一个大于 2 的正偶数为参数，输出两个素数，并且这两个素数之和等于原来的正偶数(哥德巴赫猜想)。如果存在多组符合条件的素数，则全部输出。

```
import math
def isprime(n):
    m = int(math.sqrt(n)) + 1
    for x in range(2,m):
        if n % x == 0:
            return False
    return True
def gt(n):
```

```
    if isinstance(n,int) and n>2 and n%2==0:
        for x in range(3,int(n/2)+1):
            if x%2==1 and isprime(x) and isprime(n-x):
                print(n,"=",x,"+",n-x)
gt(100)
```

运行结果：

```
================================
100 = 3 + 97
100 = 11 + 89
100 = 17 + 83
100 = 29 + 71
100 = 41 + 59
100 = 47 + 53
```

5.7　模块与包

当编写的程序中类和函数较多时，就需要对它们进行有效的组织，在 Python 中，模块和包都是组织的方式。复杂度较低时可以使用模块进行管理，复杂度高时还要使用包进行管理。

5.7.1　模块的概念

模块就是一个包含 Python 定义和域名的脚本文件(.py)，通过这个文件把一组相关的函数、类或代码组织到一个文件中，实现代码复用。

1. 模块的分类

Python 中，模块被看成一个独立的文件存在，其存在的目的是被其他程序或者解释器调用。模块一般可以分为以下几种：

(1) 内置模块，如之前调用过的 time、math 等模块，都是 Python 语言开发者为大家内置的一些模块，完成一些简单的功能。

(2) 第三方模块，这部分模块需要通过额外安装才能够调用。如通过 pip 或者 conda 进行安装的模块。

(3) 自定义模块，有时为了完成特定的任务，但又没有现有的内置模块或第三方模块，只能自行编写模块完成任务，这一类模块就是自定义模块。这里需要注意，编写自定义模块应当和内置模块以及第三方模块命名不冲突，否则可能会造成麻烦。

2. 使用模块的优点

模块就像是大家搭建积木城堡时的每一小块积木。一个大型的程序就是由一个个模块组成的，在编写大型程序时使用模块有以下优点。

(1) 极大程度上提高代码的可维护性。

(2) 模块可以被复用,也就是一个模块编写完成后,其他程序可以直接调用,无须再次编写,减少代码书写量,符合程序设计规范。

(3) 使用模块可有效地避免变量名的重复,因为在不同模块中可以使用相同的名字进行编程,但是它们之间不会相互干扰。

3. 命名空间

讲到模块,就需要提到"命名空间"(namespace)这个概念。在 Python 中有三类命名空间——局部命名空间、全局命名空间和内置命名空间。解释器在运行时,会为每一个模块新建立一个命名空间,相当于给每个模块提供一间屋子,因此在这个屋子中,函数和变量的命名互相不受影响,同样在不同的模块中也可以有相同名称的变量和函数;在不同的函数中也可以有相同的变量。

当在函数内部声明一个变量时,解释器会将这个变量放到局部变量的命名空间中去;当在模块中声明一个变量时,解释器会将这个变量放入全局变量的命名空间中去。其嵌套关系是:全局变量的命名空间包含着局部变量的命名空间。因此,就会出现,在函数中可以使用函数中定义的局部变量,同时也可以使用模块中定义的全局变量。但是在全局变量的命名空间是无法使用函数内部定义的局部变量的。

解释器寻找程序中所需要的变量过程:解释器依次查找三个命名空间,由局部变量的命名空间开始查找,再到全局变量的命名空间,最后到内置变量的命名空间。当解释器找到所需变量后,就会停止查找。如果在任何一个命名空间中都没有查找到,解释器将会报出错误。

5.7.2 模块的导入

想要使用模块,就要先导入模块。在 Python 中,模块的导入方式有使用 import 导入和使用 from…import…导入两种。

1. 使用 import 导入模块

语法格式 1:

```
import 模块 1,模块 2,…
```

语法格式 2:

```
import 模块 1 as 别名
```

说明:可一次导入一个模块,也可一次导入多个模块。

```
>>> import random                    #导入一个模块
>>> import time,pygame               #导入多个模块
>>> import random as rd              #导入 random 模块并以 rd 作为别名
```

模块导入之后便可以通过"模块名.函数名"的方式使用模块中的功能。下面以 random 模块为例说明模块的导入与使用。

```
>>> import random                    #导入 random 模块
>>> random.randint(10,20)
13
>>> import random as rd              #别名方式使用
>>> rd.randint(10,20)                #产生指定范围内的随机数
15
>>> rd.seed(10)                      #指定随机数种子 10
>>> rd.randint(1,100)                #产生指定范围内的随机数
74
>>> rd.randint(1,100)
5
>>> rd.randint(1,100)
55
>>> rd.seed(10)                      #再次指定随机数种子 10,用于产生可重复的随机数序列
>>> rd.randint(1,100)
74
>>> rd.randint(1,100)
5
>>> rd.randint(1,100)
55
```

2. 使用 form…import…导入模块

语法格式：

```
from 模块名 import 函数
```

说明：使用 form…import…方式导入模块之后，无须添加前缀，即可像使用当前程序中的函数一样使用模块中的内容。form…import…也支持一次导入多个函数、类、变量等，函数与函数之间使用逗号隔开。

例如，导入 random 模块中的 randint 函数和 uniform 函数后便可直接使用。

```
>>> from random import randint,uniform
>>> uniform(10,20)
14.825616745508558
```

利用通配符 * 可使用 form…import…导入模块中的全部函数。例如，导入 random 模块中的全部函数，并直接使用 randint 函数。

```
>>> from random import *
>>> randint(10,20)
12
```

模块是逻辑上有关系的变量、函数以及类的集合，运行这些内容的目的是初始化模块，并且这个初始化的过程只执行一次，执行时机是首次导入的时候。当模块多次被导入时，将不会再次进行初始化，而是从内存中读取已经导入的模块内容。

例如：

```
# test_import.py                    建立模块
print("test_import.py")
```

```
# test.py
#导入时需保证 test_import.py 和 test.py 位于同一目录下,并且运行 test.py
import test_import
import test_import
```

运行结果：

```
================================
test_import.py
```

5.7.3 包的使用

1. 包的概念和结构

当程序中的模块非常多时,需要再进行组织。将功能类似的模块放到一起,形成“包”。本质上,“包”就是一个必须有__init__.py 的文件夹。包的典型结构如图 5-3 所示。

包下面可以包含模块(module),也可以再包含子包(subpackage)。就像文件夹下面可以有文件,也可以有子文件夹一样。包含子包的结构如图 5-4 所示。

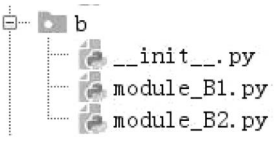


图 5-3 包的典型结构

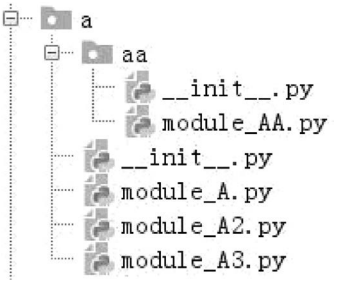


图 5-4 包含子包的结构

2. 包的导入

在 Python 中,包的导入与模块的导入方式相同,可以使用 import 导入和使用 from…import…导入两种。

语法格式 1：

```
from package import item
```

说明：item 可以是包、模块,也可以是函数、类、变量。

语法格式 2：

```
import item1.item2
```

说明：item 必须是包或模块。

例如在图 5-3 结构中,需要导入 module_AA.py,方式如下：

```
import a.aa.module_AA                    #必须加完整名称来引用
from a.aa import module_AA               #可以直接使用模块名
from a.aa.module_AA import fun_AA        #可以直接使用函数名
```

3. __init__.py

__init__.py 是包的标志性文件，Python 通过一个文件夹下是否有__init__.py 文件，来识别该文件夹是否为包文件。不同包下包含相同名称的模块时，为了区分，通过“包名.模块名”路径来指定模块，这个路径叫作命名空间。

导入包的本质是导入了包的“__init__.py”文件。也就是说，“import pack1”意味着执行了包 pack1 下面的__init__.py 文件。这样，可以在__init__.py 中批量导入需要的模块，而不再需要一个一个地导入。

__init__.py 具有以下三个特征：

(1) 作为包的标识，不能删除。

(2) 可以实现模糊导入。

(3) __init__.py 内容可以为空，也可以写入一些包执行时的初始化代码。

本章习题

一、填空题

1. 定义函数时使用的关键字是_____。

2. _____表达式可以用来创建只包含一个表达式的匿名函数。

3. Python 中函数参数可以分为_____、_____、_____和可变参数等。

4. Python 作用域可以分为_____、嵌套父级函数的局部作用域、_____和_____。

5. 在函数内部可以通过关键字_____来声明或定义全局变量。

6. 递归函数包括两个要素：_____与_____。

二、判断题

1. 函数是代码复用的一种方式。(　　)

2. 一个函数如果带有默认值参数，那么必须所有参数都设置默认值。(　　)

3. 定义 Python 函数时，如果函数中没有 return 语句，则默认返回空值 None。(　　)

4. 在函数内部，可以使用 global 来声明使用外部全局变量，也可以使用 global 直接定义全局变量。(　　)

5. 在函数内部没有办法定义全局变量。(　　)

6. 不同作用域中的同名变量之间互相不影响，也就是说，在不同的作用域内可能定义同名的变量。(　　)

7. 在调用函数时，可以通过关键参数的形式进行传值，从而避免必须记住函数形参顺

序的麻烦。（　　）

三、程序阅读题

1. 下面程序的执行结果是_____。

```
def Sum(a,b = 3,c = 5):
    return sum([a,b,c])
print(Sum(a = 8,c = 2))
print(Sum(8))
print(Sum(8,2))
```

2. 下面程序的执行结果是_____。

```
def demo():
    x = 5
x = 3
demo()
print(x)
```

3. 下面程序的执行结果是_____。

```
def f1(x,y):
    if y == 0:
        return x
    else:
        return x % y
print(f1(18,5))
```

4. 下面程序的执行结果是_____。

```
s = map(lambda x:x ** 2,[1,2,3])
for n in s:
    print(n,end = " ")
```

5. 下面程序的执行结果是_____。

```
def f2(p1, * p2):
    print(p1,p2)
f2(1,2,3,4)
```

6. 下面程序的执行结果是_____。

```
def f3(p1, ** p2):
    print(p1)
    print(p2)
f3(1,x = 2,y = 3,z = 4)
```

7. 下面程序的执行结果是_____。

```
n = 1
m = 0
```

```
def f4():
    global n
    for x in [1,2,3]:n += 1
    m = 10
    print(n,m)
f4()
print(n,m)
```

四、程序设计题

1. 自定义一个 sumfib(n)函数，返回斐波那契数列前 n 项之和。

2. 自定义一个 jc(n)函数，返回 n!。

3. 自定义一个 f1()函数，可以接收任意多个整数并输出其中的最大值和所有整数之和。

4. 自定义一个 hw(ch)函数，判断 ch 是否返回字符串。