第6章

函数

拓展阅读

学习目标

- 了解函数，能够简述函数的概念以及在程序中使用函数的好处
- 掌握函数的定义和调用，能够根据需求定义和调用函数
- 掌握函数参数的传递方式，能够通过各种方式向函数内部传递数据
- 熟悉函数的返回值，能够在程序中处理一个或多个返回值
- 掌握局部变量和全局变量，能够在程序中使用局部变量和全局变量
- 掌握 global 或 nonlocal 关键字，能够通过这两个关键字修改变量的作用域
- 掌握递归函数的使用，能够运用递归函数解决阶乘的问题
- 掌握匿名函数的使用，能够运用匿名函数简化简单函数的定义

随着程序功能的增加，程序开发的难度和程序的复杂度也逐渐增加，如果仍然按照前面介绍的编写代码的方式开发程序，那么程序的阅读和后期管理与维护会给开发人员带来不少困扰。为了解决这些问题，也为了提高代码的复用性、更好地组织代码的结构与逻辑，Python 引入了函数这一概念。本章将针对函数的相关知识进行讲解。

6.1 函数概述

在程序开发中，函数是组织好的、用于实现单一功能或相关联功能的代码段。在前面我们已经接触过一些函数，例如，用于向控制台输出内容的 print()函数、用于接收用户输入内容的 input()函数等。我们可以将函数视为一段有名字的代码，这类代码可以在需要的地方以“函数名()”的形式调用。

为了帮助读者更直观地理解使用函数的好处，下面以非函数和函数两种形式编写程序，分别输出由 2×2、3×3、4×4 个星号组成的正方形，具体如图 6-1 所示。

对比图 6-1（a）和图 6-1（b）可知，使用了函数的程序的结构更加清晰、代码更加精简。

试想一下，若程序希望再输出一个 5×5 个星号的正方形，应该如何实现呢？对于未使用函数的程序而言，需要复制图 6-1（a）中用于输出任一正方形的代码，将循环次数修改为 5，

如此，冗余代码增加；对于使用了函数的程序而言，只需要在调用用于输出正方形的函数时，将函数中的参数修改为 5 即可。

```
# 输出由2×2个星号组成的正方形
for i in range(2):
    for i in range(2):
        print('*', end=' ')
    print()
# 输出由3×3个星号组成的正方形
for i in range(3):
    for i in range(3):
        print('*', end=' ')
    print()
# 输出由4×4个星号组成的正方形
for i in range(4):
    for i in range(4):
        print('*', end=' ')
    print()
```

（a）未使用函数的程序

```
# 输出正方形的函数
def print_square(length):
    for i in range(length):
        for i in range(length):
            print('*', end=' ')
        print()

# 使用函数，输出由2×2个星号组成的正方形
print_square(2)
# 使用函数，输出由3×3个星号组成的正方形
print_square(3)
# 使用函数，输出由4×4个星号组成的正方形
print_square(4)
```

（b）使用了函数的程序

图6-1　未使用和使用了函数的程序

综上所述，相较于之前的编程方法，函数式编程将程序模块化，可减少冗余代码，使程序结构更为清晰，这样不仅可以提高开发人员的编程效率，而且方便后期的程序维护与扩展。

6.2　函数的定义和调用

函数的使用分为定义和调用两个部分，本节将针对函数的定义和调用进行详细讲解。

6.2.1　定义函数

前面使用的 print()和 input()都是 Python 的内置函数，这些函数由 Python 官方定义，可以供开发人员直接使用。另外，开发人员也可以根据自己的需求定义函数。在 Python 中使用关键字 def 定义函数，定义函数的语法格式如下：

```
def 函数名([参数列表]):
    ['''文档字符串''']
    函数体
    [return 语句]
```

以上语法格式的相关说明如下。

- 关键字 def：用于标记函数的开始。
- 函数名：函数的唯一标识，其命名遵循标识符的命名规则。
- 参数列表：负责接收传入函数中的数据，可以包含一个或多个参数，也可以为空。
- 冒号：用于标记函数体的开始。
- 文档字符串：由一对三引号包裹的、用于说明函数功能的字符串，可以省略。
- 函数体：实现函数功能的具体代码，由一行或多行语句构成。
- return 语句：用于将函数的处理结果返回给函数的调用方，同时也标记函数的结束。若函数没有返回值，则可以省略 return 语句。

如果在定义某函数时参数列表为空，则这个函数称为无参函数。例如，定义一个计算两

个数之和的函数，具体代码如下：

```
def add():
    result = 11 + 22
    print(result)
```

以上定义的 add()函数是一个无参函数，它只能计算 11 和 22 的和，具有很强的局限性。为了增强函数的灵活性，使函数能够计算任意两个数之和，这里可以定义一个带有参数的 add_modify()函数，使用该函数的参数接收从函数外部传入的数据，之后计算它们的和，示例代码如下：

```
def add_modify(a, b):
    result = a + b
    print(result)
```

6.2.2　调用函数

函数在定义完成后不会立刻执行，直到被程序调用时才会执行。调用函数的方式非常简单，其语法格式如下：

```
函数名([参数列表])
```

例如，调用 6.2.1 小节中定义的 add_modify()函数，代码如下：

```
add_modify(10, 20)
```

运行代码，结果如下所示：

```
30
```

实际上，程序在执行“add_modify(10, 20)”时经历了以下 4 个步骤。

（1）程序在调用函数的位置暂停执行，跳转到函数定义的位置。

（2）将数据传递给函数参数。在这个例子中，10 和 20 被分别传递给函数的参数 a 和 b。

（3）程序执行函数体中的语句。在这个例子中，函数体包含两条语句，用于计算并输出 a 与 b 的和。

（4）程序回到暂停处继续执行。

下面使用一张图来描述程序执行“add_modify(10, 20)”的整个过程，具体如图 6-2 所示。

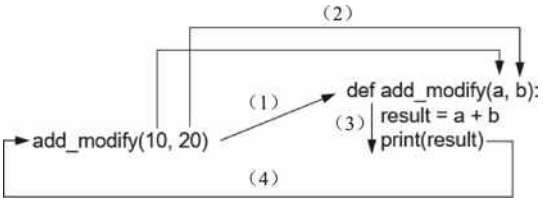

图6-2　程序执行“add_modify(10, 20)”的整个过程（1）

函数内部也可以调用其他函数，这称为函数的嵌套调用。例如，在 add_modify()函数内部增加调用 add()函数的代码，修改后的代码如下：

```
def add_modify(a, b):
    result = a + b
    add()                    # 在 add_modify()函数的内部调用 add()函数
    print(result)
```

运行代码，结果如下所示：

```
33
30
```

下面画图分析一下程序执行“add_modify(10, 20)”的整个过程，具体如图 6-3 所示。

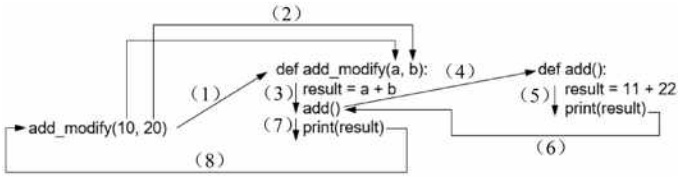

图6-3　程序执行“add_modify(10, 20)”的整个过程（2）

图 6-3 中的执行过程具体如下。

（1）程序在调用 add_modify()函数的位置暂停执行，跳转到该函数定义的位置。

（2）程序将数据 10 和 20 分别传递给函数的参数 a 和 b。

（3）程序执行 add_modify()函数的函数体。

（4）程序执行到调用 add ()函数的位置暂停执行，跳转到该函数定义的位置。

（5）程序执行 add ()函数的函数体。

（6）程序回到调用 add ()函数的位置继续执行。

（7）程序继续执行 add_modify()函数的函数体。

（8）程序回到调用 add_modify()函数的位置继续执行。

多学一招：函数的嵌套定义

在定义函数时可以在其内部嵌套定义另外一个函数，此时嵌套的函数称为外层函数，被嵌套的函数称为内层函数。例如，在 add_modify()函数中定义 test()函数，具体代码如下所示：

```
def add_modify(a, b):
    result = a + b
    print(result)
    def test():                                  # 在 add_modify()函数中定义函数 test()
        print("我是内层函数")
add_modify(10, 20)
```

运行代码，结果如下所示：

```
30
```

由以上运行结果可知，程序没有执行内层函数的输出语句，只执行了外层函数的输出语句。

需要说明的是，在函数外部无法直接调用内层函数，只能在外层函数中调用内层函数，从而执行两个函数的输出语句。例如，在 add_modify()函数中调用 test()函数，修改后的代码如下：

```
def add_modify(a, b):
    result = a + b
    print(result)
    def test():                                  # 在 add_modify()函数中定义函数 test()
        print("我是内层函数")
    test()                                       # 在 add_modify()函数中调用函数 test()
add_modify(10, 20)
```

运行代码，结果如下所示：

```
30
我是内层函数
```

6.3　函数参数的传递

通常情况下，定义函数时设置的参数称为形式参数，简称为形参；调用函数时传入的参数称为实际参数，简称为实参。函数参数的传递是指将实参传递给形参的过程。函数参数的传递可以分为位置参数的传递、关键字参数的传递、默认参数的传递、参数的打包与解包以及混合传递。本节将针对函数参数的这几种传递方式进行详细讲解。

6.3.1　位置参数的传递

当调用函数传递位置参数时，实参会按照位置顺序依次传递给对应的形参。也就是说，第 1 个实参会传递给第 1 个形参，第 2 个实参会传递给第 2 个形参，以此类推。注意，实参的数量和位置必须与函数定义中位置参数的数量和位置保持一致，否则会导致解释器运行时出现异常信息，或者错误的参数匹配。

例如，定义一个用于获取两个数之间较大值的函数 get_max()，调用 get_max()函数，通过位置参数的方式传递实参，示例代码如下：

```
def get_max(a, b):
    if a > b:
        print(a, "是较大的值！")
    elif a < b:
        print(b, "是较大的值！")
    else:
        print("两个值一样大！")
get_max(8, 5)
```

以上函数执行后会将第一个实参 8 传递给第一个形参 a，第二个实参 5 传递给第二个形参 b。

运行代码，结果如下所示：

```
8 是较大的值！
```

6.3.2　关键字参数的传递

当函数有较多参数时，想要记住每个参数的位置和含义可能会很困难，因此按照位置参数的方式传递参数并不是一个好的选择。此时，可以使用关键字参数的方式传递参数。关键字参数的传递中，通过使用“形参=实参”的格式将实参与对应的形参关联起来，根据具体的参数名传递参数，从而避免记忆参数位置的困扰。

例如，定义一个用于连接设备的函数 connect()，调用 connect()函数，通过关键字参数的方式传递实参，示例代码如下：

```
def connect(ip, port):
    print(f"设备{ip}:{port}连接！")
connect(ip="127.0.0.1", port=8080)
```

以上代码执行后会将"127.0.0.1"传递给与其关联的形参 ip，将 8080 传递给与其关联的形参 port。

运行代码，结果如下所示：

```
设备 127.0.0.1:8080 连接！
```

多学一招：仅限位置

有人此时可能会产生一个疑问，实参无论采用位置参数的方式传递，还是采用关键字参数的方式传递，每个形参都是有名称的，怎么区分采用哪种方式传递呢？Python 新增了仅限位置的语法，使用符号“/”来限定部分形参只能接收通过位置参数的传递方式传递的实参，示例代码如下：

```
def func(a, b, /, c):                    # 使用“/”限制其前面的形参 a、b
    print(a, b, c)
```

以上定义的 func()函数中，符号“/”之前的 a、b 只能接收通过位置参数的传递方式传递的实参；符号“/”之后的 c 为普通形参，可以接收采用位置参数的传递方式或关键字参数的传递方式传递的实参。

调用 func()函数，示例代码如下：

```
# 错误的调用方式
func(a=10, 20, 30)
func(10, b=20, 30)
# 正确的调用方式
func(10, 20, c=30)
```

运行代码，结果如下所示：

```
10 20 30
```

6.3.3 默认参数的传递

定义函数时可以指定形参的默认值，调用函数时可以选择是否给带有默认值的形参传值。若没有给带有默认值的形参传值，直接使用形参的默认值；若给带有默认值的形参传值，则使用实参的值覆盖默认值。

例如，定义一个用于连接设备的函数，在该函数中给一个参数指定默认值，对另一个参数不指定默认值，示例代码如下：

```
def connect(ip, port=8080):
    print(f"设备{ip}:{port}连接！")
```

接下来调用两次 connect()函数，分别使用默认值和不使用默认值，示例代码如下：

```
connect(ip="127.0.0.1")
connect(ip="127.0.0.1", port=3306)
```

以上代码中，第一次调用 connect()函数时，"127.0.0.1"会传递给与其关联的形参 ip，而形参 port 使用默认值 8080；第二次调用 connect()函数时，"127.0.0.1"会传递给与其关联的形参 ip，3306 会传递给与其关联的形参 port。

运行代码，结果如下所示：

```
设备 127.0.0.1:8080 连接！
设备 127.0.0.1:3306 连接！
```

6.3.4 参数的打包与解包

函数支持将实参以打包和解包的形式传递给形参，这样可以在不事先知道参数数量的情况下进行函数调用或参数传递，使函数的调用和参数的传递更加方便和灵活。关于打包和解包的介绍如下。

1. 打包

Python 中参数的打包是指将多个实参打包成一个元组或字典，在函数调用时将其作为一

个整体传递给形参。如果形参的前面加上“*”，那么它可以接收以元组形式打包的多个值；如果形参的前面加上“**”，那么它可以接收以字典形式打包的多个值。

定义一个形参为*args 的函数，示例代码如下：

```
def test(*args):
    print(args)
```

调用 test()函数时传入多个实参，多个实参会以元组形式打包并传递给形参。示例代码如下：

```
test(11, 22, 33, 44, 55)
```

运行代码，结果如下所示：

```
(11, 22, 33, 44, 55)
```

由以上运行结果可知，Python 解释器将传给 test()函数的所有值打包成元组后传递给了形参*args。

定义一个形参为**kwargs 的函数，示例代码如下：

```
def test(**kwargs):
    print(kwargs)
```

调用 test()函数时传入多个关联形参名的实参，示例代码如下：

```
test(a=11, b=22, c=33, d=44, e=55)
```

运行代码，结果如下所示：

```
{'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}
```

由以上运行结果可知，Python 解释器将传给 test()函数的所有实参打包成字典后传递给了形参**kwargs。

值得一提的是，虽然函数中添加“*”或“**”的形参名可以是符合命名规则的任意名称，但建议使用*args 和**kwargs。若函数没有接收到任何数据，参数*args 和**kwargs 为空，即它们分别为空元组和空字典。

2. 解包

Python 中参数的解包是指使用特殊语法将元组或字典拆分为多个值并赋给对应的形参。如果函数在调用时接收的实参是元组类型的数据，那么可以使用“*”将元组拆分成多个值，并按照位置参数的传递方式赋给对应的形参；如果函数在调用时接收的实参是字典类型的数据，那么可以使用“**”将字典拆分成多个键值对，并按照关键字参数的传递方式赋给与键名称对应的形参。

定义一个带有 5 个形参的函数，示例代码如下：

```
def test(a, b, c, d, e):
    print(a, b, c, d, e)
```

调用 test()函数时传入一个包含 5 个元素的元组，并使用“*”对该元组执行解包操作，示例代码如下：

```
nums = (11, 22, 33, 44, 55)
test(*nums)
```

运行代码，结果如下所示：

```
11 22 33 44 55
```

由以上运行结果可知，元组被解包成多个值。

调用 test()函数时传入一个包含 5 个元素的字典，并使用“**”对该字典执行解包操作，示例代码如下：

```
nums = {"a":11, "b":22, "c":33, "d":44, "e":55}
test(**nums)
```

运行代码，结果如下所示：

```
11 22 33 44 55
```

由以上运行结果可知，字典被解包成多个值。

6.3.5 混合传递

前面介绍的参数传递的方式在定义或调用函数时可以混合使用，但是需要遵循一定的优先级规则。函数参数的传递方式按优先级从高到低依次为位置参数的传递、关键字参数的传递、默认参数的传递、打包传递。

在定义函数时，带有默认值的参数必须在普通参数（不带有默认值或星号标识的参数）之后，带有“**”标识的参数必须在带有“*”标识的参数之后。

例如，定义一个有多种形参的函数，代码如下：

```
def test(a, b, c=33, *args, **kwargs):
    print(a, b, c, args, kwargs)
```

调用 test()函数，依次传入不同个数和形式的参数，示例代码如下：

```
test(1, 2)
test(1, 2, 3)
test(1, 2, 3, 4)
test(1, 2, 3, 4, e=5)
```

运行代码，结果如下所示：

```
1 2 33 () {}
1 2 3 () {}
1 2 3 (4,) {}
1 2 3 (4,) {'e': 5}
```

test()函数共有 5 个参数，以上代码多次调用 test()函数并传入不同个数和形式的参数，下面结合代码运行结果逐个说明函数调用过程中参数的传递情况。

第一次调用 test()函数时，该函数接收到实参 1、2，这两个实参分别被位置参数 a 和 b 接收；剩余 3 个形参 c、*args、**kwargs 没有接收到实参，都使用默认值，因此最后 3 个参数对应的输出的结果为 33、()和{}。

第二次调用 test()函数时，该函数接收到实参 1、2、3，前 3 个实参分别被位置参数 a、b、c 接收；剩余两个形参*args、**kwargs 没有接收到实参，都使用默认值，因此最后两个参数对应的输出的结果为()和{}。

第三次调用 test()函数时，该函数接收到实参 1、2、3、4，前 4 个实参分别被形参 a、b、c、*args 接收；形参**kwargs 没有接收到实参，因此最后一个参数对应的输出的结果为{}。

第四次调用 test()函数时，该函数接收到实参 1、2、3、4 和关联形参 e 的实参 5，所有的实参被相应的形参接收。

6.4 函数的返回值

函数中的 return 语句会在函数结束时将数据返回给程序，同时让程序回到函数被调用的位置继续执行。

例如，定义一个用于过滤敏感词的函数，代码如下：

```
def filter_sensitive_words(words):
    if "山寨" in words:
        new_words = words.replace("山寨", "**")
        return new_words
```

以上代码中的 filter_sensitive_words()函数会接收字符串，将该字符串中的"山寨"替换为"**"，并使用 return 语句返回替换后的字符串。

调用 filter_sensitive_words()函数，使用一个变量保存返回值，示例代码如下：

```
result = filter_sensitive_words("这个手机是山寨版吧！")
print(result)
```

运行代码，结果如下所示：

```
这个手机是**版吧！
```

以上定义的函数只返回了一个值，如果函数使用 return 语句返回了多个值，那么这些值将被保存到元组中。

下面定义一个用于控制游戏角色位置的函数，该函数使用 return 语句返回游戏角色目前所处位置的 x 坐标和 y 坐标，示例代码如下：

```
def move(x, y, step):
    nx = x + step
    ny = y - step
    return nx, ny  # 使用 return 语句返回多个值
result = move(100, 100, 60)
print(result)
```

运行代码，结果如下所示：

```
(160, 40)
```

6.5　变量作用域

Python 变量并不是在哪个位置都可以访问，具体的访问权限取决于变量定义的位置，变量的有效范围视为变量的作用域，作用域决定了在哪些位置能够访问变量。本节将针对变量作用域的相关知识进行详细讲解。

6.5.1　局部变量和全局变量

根据作用域的不同，变量可以分为局部变量和全局变量。下面分别对局部变量和全局变量进行介绍。

1. 局部变量

局部变量是指在函数内部定义的变量，它的作用域仅限于函数内部，只能在函数内部对它进行访问或使用。一旦函数执行结束，局部变量将无法被访问或使用。

例如，在 test_one()函数中定义一个局部变量 number，分别在该函数内部和函数外部访问局部变量 number，代码如下：

```
def test_one():
    number = 10                    # 局部变量
    print(number)                  # 在函数内部访问局部变量
```

```
test_one()
print(number)
```

运行代码，结果如下所示：

```
10
Traceback (most recent call last):
  File "E:\FastPrograms3\Chapter06\code04.py", line 5, in <module>
    print(number)
          ^^^^^^
NameError: name 'number' is not defined
```

结合代码运行结果分析，程序调用 test_one()函数后成功访问并输出了局部变量 number 的值，说明局部变量能够在函数内部使用；程序在调用 test_one()函数后继续在函数外部访问局部变量 number，这时出现“name 'number' is not defined”的错误信息，说明局部变量不能在函数外部使用。

不同函数内部可以包含同名的局部变量，这些局部变量的关系类似于不同目录下同名文件的关系，它们相互独立，互不影响。例如，在 test_one()函数中定义一个局部变量 number，在 test_two()函数中也定义一个局部变量 number，分别在每个函数的内部访问 number，代码如下：

```
def test_one():
    number = 10
    print(number)                          # 访问 test_one()函数的局部变量 number
def test_two():
    number = 20
    print(number)                          # 访问 test_two()函数的局部变量 number
test_one()
test_two()
```

运行代码，结果如下所示：

```
10
20
```

结合代码运行结果分析：程序在执行 test_one()函数时访问了局部变量 number，并且输出了 number 的值 10；程序在执行 test_two()函数时访问了局部变量 number，并且输出了 number 的值 20；由此可见，不同函数的局部变量之间互不影响。

2. 全局变量

全局变量是指在整个程序中都可以使用的变量，它们一般定义在函数外部，并且在整个程序运行期间占用存储单元。例如，在 test_one()函数的外部定义一个全局变量 number，分别在该函数的内部和外部访问全局变量 number，具体代码如下：

```
number = 10                                # 定义全局变量
def test_one():
    print(number)                          # 在函数内部访问全局变量
test_one()
print(number)                              # 在函数外部访问全局变量
```

运行代码，结果如下所示：

```
10
10
```

结合代码运行结果分析：程序在调用 test_one()函数时成功地访问了全局变量 number，并且输出了 number 的值；程序在执行完 test_one()函数后再次成功地访问了全局变量 number，并且输出了 number 的值；由此可知，全局变量可以在程序的任意位置被访问。

需要注意的是，全局变量在函数内部只能被访问，而无法被直接修改。下面对 test_one() 函数进行修改，在该函数中添加给全局变量 number 重新赋值的语句，修改后的代码如下：

```
number = 10                                    # 定义全局变量
def test_one():
    print(number)                              # 在函数内部访问全局变量
    number += 1                                # 在函数内部直接修改全局变量
test_one()
print(number)
```

运行代码，结果如下所示：

```
Traceback (most recent call last):
  File "E:\FastPrograms3\Chapter06\code04.py", line 5, in <module>
    test_one()
  File " E:\FastPrograms3\Chapter06\code04.py ", line 3, in test_one
    print(number)                                  # 在函数内部访问全局变量
          ^^^^^^
UnboundLocalError: cannot access local variable 'number' where it is not associated
with a value
```

在程序的开头明明已经定义了全局变量，但以上错误信息表示程序无法访问未声明的局部变量 number，这是为什么呢？这是因为函数内部的变量 number 被视为局部变量，而在执行“number+=1”这行代码之前并未声明过局部变量 number。由此可见，在函数内部只能访问全局变量，而无法直接修改全局变量。

多学一招：LEGB 原则

LEGB 是在程序中搜索变量时遵循的原则，该原则中的每个字母指代一种作用域，具体如下。

（1）L（Local）：局部作用域，例如，在函数内部定义的局部变量和形参的作用域。

（2）E（Enclosing）：嵌套作用域，例如，在嵌套函数中外层函数声明的变量的作用域。

（3）G（Global）：全局作用域，例如，在模块中定义变量的作用域。

（4）B（Built-in）：内置作用域，例如，Python 内置的模块或函数中声明的变量的作用域。

Python 在搜索变量时会按照“L→E→G→B”这个顺序依次在这 4 种作用域中搜索变量：若搜索到匹配的变量，则终止搜索并使用该变量；若搜索完 L、E、G、B 这 4 种作用域仍未找到变量，程序将抛出异常。

6.5.2　global 和 nonlocal 关键字

在函数内部无法直接修改全局变量或在嵌套函数的外层函数中声明的变量，但可以使用 global 或 nonlocal 关键字间接修改这些变量。下面分别介绍 global 和 nonlocal 关键字的用法。

1. global 关键字

global 关键字用于在函数内部声明一个全局变量，并允许在函数内部对该全局变量进行修改。global 关键字的语法格式如下：

```
global 变量名
```

下面对 6.5.1 小节的最后一个示例的代码进行修改，先在 test_one()函数中使用 global 关键字声明全局变量 number，然后在函数中重新给 number 赋值，最后输出修改后 number 的值，修改后的代码如下：

```
number = 10                              # 定义全局变量
def test_one():
    global number                        # 使用 global 声明变量 number 为全局变量
    number += 1
    print(number)
test_one()
print(number)
```

运行代码，结果如下所示：

```
11
11
```

由上述结果可知，程序能够正常运行，说明使用 global 关键字修饰后可以在函数中修改全局变量。

2. nonlocal 关键字

如果在嵌套函数内部访问和修改外层函数中的变量，而不是创建新的局部变量，可以使用 nonlocal 关键字。nonlocal 关键字的语法格式如下：

```
nonlocal 变量名
```

例如，在 test()函数嵌套的 test_in()函数中使用 nonlocal 关键字声明 number 变量是外部函数中定义的变量，之后修改这个变量的值，具体代码如下所示：

```
def test():
    number = 10                  # 定义变量
    def test_in():
        nonlocal number          # 使用 nonlocal 声明变量 number
        number = 20              # 在内部函数中修改变量 number
    test_in()
    print(number)
test()
```

以上定义的 test()函数中嵌套了函数 test_in()，在 test()函数中定义了一个变量 number，在 test_in()函数中使用 nonlocal 关键字声明变量 number，并修改了变量 number 的值，调用 test_in()函数后输出变量 number 的值。

运行代码，结果如下所示：

```
20
```

从程序的运行结果可以看出，程序在执行 test_in()函数时成功地修改了变量 number，并且输出了 number 的值。

6.6 实训案例

6.6.1 智能聊天机器人

智能聊天机器人

近年来，智能聊天机器人在国内广泛应用于企业客户服务、教育、医疗等多个领域，为人们提供了更加便捷、高效、智能的服务。本案例要求实现一个简易智能聊天机器人（以下简称机器人）——小智，用于帮助用户解答百科知识的问题，具体要求如下。

（1）机器人默认解答 5 个问题，这 5 个问题分别是诗仙是谁、中国第一个朝代是哪个朝代、三十六计的第一计是什么、“天府之国”是中国的哪个地方、中国第一长河是哪条河，

答案分别是李白、夏朝、瞒天过海、四川、长江。

（2）机器人有 3 项功能，分别是训练、对话和离开。若用户从键盘输入 t，说明用户想训练机器人，此时机器人需要记录训练的新问题及答案；若用户从键盘输入 c，说明用户想跟机器人对话，此时机器人需要回答用户提出的问题；若用户从键盘输入 l，说明用户想让机器人离开，此时机器人需要退出程序。

6.6.2　饮品自动售货机


饮品自动售货机

随着“无人零售”经济的兴起，商场、车站、大厦等很多场所都引入了饮品自动售货机，方便人们选购自己想要的饮品。购买者选择想要的饮品，通过投币或扫码的方式支付，支付成功后从出货口取出饮品。本案例要求编写代码，运用函数的知识实现饮品自动售货机的程序，模拟用户在饮品自动售货机选购饮品的流程，具体要求如下。

（1）用户开始使用时饮品自动售货机会展示饮品菜单，饮品菜单包括饮品名称及其价格，具体如下所示。

```
--------------------
可口可乐 : 2.5 元
百事可乐 : 2.5 元
冰红茶 : 3 元
脉动 : 3.5 元
果缤纷 : 3 元
绿茶 : 3 元
茉莉花茶 : 3 元
尖叫 : 2.5 元
--------------------
```

（2）用户可以根据自己的需要重复输入饮品名称和数量，如果用户输入的饮品名称不存在，则会输出饮品名称不正确的提示信息；如果用户输入的饮品数量为负数或者非整数，则会输出饮品数量不合法的提示信息；如果用户输入的饮品名称为 q，则会完成选择进入结算的流程。

（3）饮品自动售货机会根据饮品价格和数量计算总金额。

6.7　特殊形式的函数

除了前面按标准定义的函数之外，Python 还提供了两种具有特殊形式的函数：递归函数和匿名函数。本节将针对递归函数和匿名函数进行详细讲解。

6.7.1　递归函数

函数在定义时可以直接或间接地调用其他函数。若函数内部调用了自身，则这个函数被称为递归函数。递归函数通常用于解决结构相似的问题，它采用递归的方式，将一个复杂的大型问题转化为与原问题结构相似的、规模较小的若干子问题，之后对最小化的子问题求解，从而得到原问题的解。

递归函数在定义时需要满足两个基本条件：一个是有递归公式，另一个是有边界条件。

其中，递归公式描述如何通过解决子问题来解决原问题；边界条件是最小化的子问题，也是递归结束的条件。

递归函数的执行可以分为以下两个阶段。

（1）递推：递归的执行基于上一次的运算结果，将问题规模逐渐缩小，通过不断调用自身来求解子问题，这一阶段也称为递归调用。

（2）回溯：当递归达到边界条件时，递归开始，逐级返回函数调用过程中得到的结果，将得到的部分结果组合成最终的解。

递归函数的一般语法格式如下所示：

```
def 函数名([参数列表]):
    if 边界条件:
        return 结果
    else:
        return 递归公式
```

递归最经典的应用之一便是阶乘。在数学中，求正整数 $n!$（n 的阶乘）问题根据 n 的取值可以分为以下两种情况。

（1）当 n=1 时，所得的结果为 1。

（2）当 n>1 时，所得的结果为 $n\times(n-1)!$。

那么利用递归求解阶乘问题时，n=1 是边界条件，$n\times(n-1)!$是递归公式。

编写代码实现 $n!$的求解，示例代码如下：

```
def func(num):
    if num == 1:
        return 1
    else:
        return num * func(num - 1)
num = int(input("请输入一个整数："))
result = func(num)
print("5!=%d"%result)
```

运行代码，按提示输入整数 5，结果如下所示：

```
请输入一个整数：5
5!=120
```

func(5)的求解过程如图 6-4 所示。

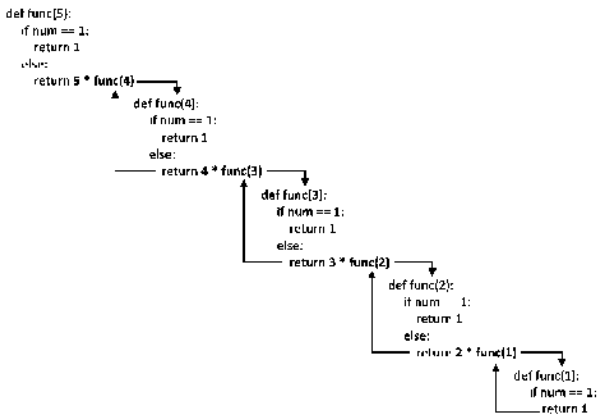

图6-4 func(5)的求解过程

结合图 6-4 分析 func(5)的求解过程可知：程序将求解 func(5)转化为求解 5*func(4)，想要得到 func(5)的结果，必须先得到 func(4)的结果；求解 func(4)又会被转换为求解 4*func(3)，同样地，想要得到 func(4)的结果必须先得到 func(3)的结果。以此类推，直到程序开始求解 2*func(1)，此时触发临界条件，func(1)的值可以直接计算，之后结果开始向上逐级传递，直到最终返回 func(5)的位置，求得 5!。

6.7.2　匿名函数

匿名函数是一类不需要命名的函数，它与普通函数一样可以在程序的任何位置使用。Python 中使用 lambda 关键字定义匿名函数，它的语法格式如下：

```
lambda <形参列表> :<表达式>
```

以上语法格式中，形参列表可以包含零个、一个或多个参数，多个参数使用英文逗号分隔，表达式定义匿名函数要执行的操作。

关于匿名函数与普通函数的主要区别如下。

（1）普通函数在定义时有函数名，而匿名函数没有函数名。

（2）普通函数的函数体中包含多条语句，而匿名函数的函数体只能是一个表达式。

（3）普通函数可以实现比较复杂的功能，而匿名函数可实现的功能比较简单。

定义好的匿名函数不能直接使用，最好使用一个变量保存它，以便后期可以随时使用这个函数。例如，定义一个计算数值平方的匿名函数，并将匿名函数返回的结果赋给一个变量，具体代码如下：

```
temp = lambda x : pow(x, 2)
```

此时，变量 temp 可以作为匿名函数的临时名称来调用函数，示例代码如下：

```
print(temp(10))
```

运行代码，结果如下所示：

```
100
```

6.8　实训案例

6.8.1　兔子数列


兔子数列

兔子数列又称斐波那契数列、黄金分割数列，它从兔子繁殖的例子中引出，故此得名。兔子繁殖的故事如下。

兔子一般在出生两个月之后就有了繁殖能力，每对兔子每月可以繁殖一对小兔子，假如所有的兔子都不会死，试问一年以后一共有多少对兔子？

本案例要求编写代码，利用递归函数实现根据月份计算兔子总数量的功能。

6.8.2　商品排序


商品排序

某电商平台统计了近一周内华为手机的销量等信息，具体如表 6-1 所示。

本实例要求编写程序，使用匿名函数定义排序规则，实现按销量对所有手机信息进行降序排列的功能。

表 6-1 华为手机的销量等信息

名称	价格/元	销量
华为 P60	4887.00	210
华为 Mate 40E Pro	4699.00	90
华为 nova 10 青春版	1698.00	102
华为 P50 Pro	3987.00	88
华为畅享 70	999.00	152

6.9 阶段案例——学生管理系统

阶段案例——学生管理系统

学生信息是高等院校的一项重要数据资源，具有数量庞大、更新频繁等特点，给管理人员带来了许多挑战。随着计算机应用的普及，人们设计了针对学生信息特点及实际需要的学生管理系统，减轻了管理人员的工作负担。本案例要求开发一个简易版学生管理系统，该系统的功能菜单如图 6-5 所示。

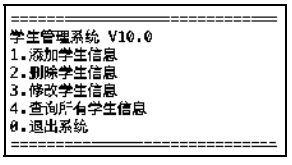

图6-5 学生管理系统的功能菜单

由图 6-5 可知，学生管理系统总共有 5 个功能，各个功能的说明如下。

（1）添加学生信息：用户按照系统提示依次输入学生的姓名、性别和手机号，输入完成后会收到系统的“添加成功!”提示信息。

（2）删除学生信息：用户按照系统提示输入学生的序号，输入完成后会收到系统的“删除成功!”提示信息。

（3）修改学生信息：用户按照系统提示先输入待修改学生的序号，再依次输入修改后的学生姓名、性别和手机号。若学生管理系统中还没有添加过学生信息，提示“学生信息为空”。

（4）查询所有学生信息：系统按照固定格式输出所有学生信息。

（5）退出系统：退出学生管理系统。

6.10 本章小结

本章主要讲解了函数的相关知识，包括函数概述、函数的定义和调用、函数参数的传递、函数的返回值、变量作用域、特殊形式的函数，并配套了实训案例。通过对本章的学习，读者能够深入理解函数的相关知识，并熟练地在实际开发中应用函数。

6.11 习题

一、填空题

1. Python 中使用关键字_____定义一个函数。
2. _____函数是一类不需要命名的函数。
3. 若函数内部调用了自身，则这个函数被称为_____。
4. Python 中使用_____关键字可以在函数内部声明全局变量。
5. _____变量是指在函数内部定义的变量，它只能在函数内部被使用。

二、判断题

1. 函数在定义完成后会立刻执行。（　　）
2. 变量在程序的任意位置都可以被访问。（　　）
3. 函数可以提高代码的复用性。（　　）
4. 在任何函数内部都可以直接访问和修改全局变量。（　　）
5. 当按照位置给函数传递参数时，多个参数之间有严格的位置关系。（　　）

三、选择题

1. 下列关于函数的说法中，错误的是（　　）。
   A. 函数可以减少重复的代码，使得程序更加模块化
   B. 不同的函数中可以使用相同名字的变量
   C. 调用函数时，实参的传递顺序与形参的传递顺序可以不同
   D. 匿名函数与使用关键字 def 定义的函数没有区别
2. Python 使用哪个关键字定义一个匿名函数？（　　）
   A. function　　B. func　　C. def　　D. lambda
3. Python 使用哪个关键字自定义一个函数？（　　）
   A. function　　B. func　　C. def　　D. lambda
4. 请阅读下面的代码：

```
num_one = 12
def sum(num_two):
    global num_one
    num_one = 90
    return num_one + num_two
print(sum(10))
```

运行代码，输出结果为（　　）。
   A. 102　　B. 100　　C. 22　　D. 12

5. 请阅读下面的代码：

```
def many_param(num_one, num_two, *args):
    print(args)
many_param(11, 22, 33, 44, 55)
```

运行代码，输出结果为（　　）。
   A. (11,22,33)　　B. (22,33,44)　　C. (33,44,55)　　D. (11,22)

四、简答题

1. 简述位置参数的传递、关键字参数的传递、默认参数的传递的区别。
2. 简述函数参数混合传递的规则。
3. 简述局部变量和全局变量的区别。

五、编程题

1. 编写函数，输出 1～100 中偶数之和。
2. 编写函数，计算 20×19×18×…×3 的结果。
3. 编写函数，判断用户输入的整数是否为回文数。回文数是一个正向和逆向都相同的整数，如 123454321、9889。
4. 编写函数，判断用户输入的 3 个数字是否能作为边长构成三角形。
5. 编写函数，计算两个正整数的最小公倍数。