第5章

Python 函数和模块

在 Python 编程中，函数和模块起着至关重要的作用。函数可以帮助我们组织代码，提高代码的可复用性，并使程序更易于维护；模块则允许我们将相关功能组织在一起，提供了封装性和命名空间的概念，有助于代码结构的清晰和模块化开发。本章将深入探讨 Python 函数的定义、参数传递、返回值以及模块的导入和使用方式，帮助读者更好地理解如何利用函数和模块来构建健壮的 Python 应用程序。

本章学习目标

一、知识目标

1. 理解 Python 函数的基本概念，包括函数的定义、调用和作用域。
2. 掌握参数和返回值的使用方法，以实现函数输入/输出的基本机制。
3. 学习不同类型的函数，包括内置函数、用户定义函数以及匿名函数（lambda 表达式）。
4. 了解递归函数的概念及其在解决问题中的应用。

二、技能目标

1. 能够熟练定义和调用函数，解决特定问题。
2. 能够使用参数、返回值和作用域，编写灵活且高效的代码。
3. 能够利用递归函数解决复杂问题，如数据结构的遍历和排序算法。
4. 能够阅读和理解他人编写的函数，以及在必要时对代码进行调试和优化。

三、情感态度与价值目标

1. 培养解决问题的积极态度，通过编写和调试函数来提升逻辑思维和分析能力。
2. 重视代码的可读性和可维护性，学习编写清晰、简洁和高效的函数。
3. 激发创新思维，鼓励学生尝试不同的方法去解决问题，以发现最优解。
4. 培养团队合作精神，通过小组项目和代码分享，学习如何在团队环境中共同解决问题。

5.1 函数的定义与使用

函数用模块化的思想来实现单一或相关联功能的代码段，模块化思想在人们的生活中是非常常见的，大到国家的治理，各个机构组织都有自己专门负责的领

域,相互协调,共同配合,使一切井然有序地运行;小到整理自己的桌面,将不同的东西分门别类地放置;这是一种高效的思想。

在 Python 中,定义函数的基础格式如下所示:

```
def my_function():
    """
    This is a docstring that provides a brief description of the function.
    It explains what the function does,
    what arguments it takes (if any), and what it returns (if any).
    """
    #Function code goes here
    pass
```

- **def**:这个关键字告诉 Python 要定义一个函数。
- **my_function**:这是编程者要创建的函数的名称,在调用函数时使用。
- **函数的文档字符串**:这是一个可选的字符串,用于介绍函数的注释或文档字符串。
- **函数体**:指从函数内部执行的一系列 Python 语句。
- **pass**:占位符,表示函数体中至少有一个语句;在实际开发中,可能需要在这里添加实际的代码。

例 5.1 有一台机器人,它专门负责每天给大家送上美好的问候。请写一个函数,每天通过调用这个简单的函数享受机器人的温馨问候。

为实现上述问题,可以使用 Copilot 来编写相关代码,使用以下指令让 Copilot 来实现“机器人问候”函数的代码。

```
//编写一个函数,可以每天通过调用这个简单的函数享受机器人的问候
```

引导 Copilot 生成的代码如下所示:

```
import datetime
def greet():
    current_time = datetime.datetime.now()
    if current_time.hour<12:                    #如果当前时间小于 12 点
        print("早上好! 祝您有个美好的一天!")      #打印早上好的问候语
    elif current_time.hour< 18:                 #如果当前时间小于 18 点
        print("下午好! 祝您度过愉快的时光!")      #打印下午好的问候语
    else:
        print("晚上好! 祝您有个宁静的夜晚!")      #打印晚上好的问候语
#调用函数
greet()
```

例 5.1 通过 def 关键字定义了一个名为 greet 的函数,用于根据当前时间向用户发送问候。在函数内部,首先使用 datetime.datetime.now()函数获取当前的日期和时间,并将其赋值给 current_time。然后,根据 current_time 的小时数(通过 current_time.hour 获取)来判断当前是早上、下午还是晚上,并打印出相应的问候语。

如果当前的小时数小于 12,那么就打印“早上好!祝您有个美好的一天!”;如果当前的小时数小于 18,那么就打印“下午好!祝您度过愉快的时光!”;否则打印“晚上好!祝您有个宁静的夜晚!”。在代码的最后,调用 greet()函数执行其内部的代码,也就是打印出根据当前时间选择的问候语。

5.2　函数的参数传递

函数定义处的参数称为形式参数，函数调用处的参数称为实际参数。Python 中的函数可以接收不同类型的参数，包括位置参数、关键字参数、默认值参数和可变参数。下面介绍这些不同类型的参数传递方式。

5.2.1　位置参数

位置参数是最常用的参数传递方式，它要求调用函数时参数的顺序和定义函数时的参数列表顺序相匹配。类似给朋友传递一串糖果，位置参数的顺序是关键，如下所示：

```
def greet(name,greeting):                          #所有的形参都是位置参数
    print(greeting,name)
    #使用位置参数调用函数
    greet("Python","Hello")
    greet("Python","Hello","Good morning")  #位置参数的个数必须和形参的个数一致
```

在上述代码中，name 和 greeting 就是位置参数，需要按照它们在函数中的位置依次传递值。

5.2.2　关键字参数

与位置参数不同，关键字参数允许在调用函数时通过参数名来指定参数值，传递的参数顺序可以与定义时的参数顺序不同，好比在给朋友送礼物时在包装上写上“送给”和祝福语，不再依赖礼物的位置。这种形式的参数传递使得函数调用更加清晰易懂，如下所示：

```
def greet(name, greeting):                          #定义函数
    print(greeting, name)                           #打印问候语
#使用关键字参数调用函数
greet(greeting="Hi", name="wang" )
```

5.2.3　默认值参数

在 Python 中，可以在函数定义中为参数提供默认值。当在函数调用中没有为参数提供值时，该参数将使用其默认值，这可以避免在函数调用中总是需要提供所有参数，如下所示：

```
def greet(name,greeting="Hello"):     #greeting 是默认值参数
    print(greeting,name)
    #使用默认值参数调用函数
greet( "wang")
```

5.2.4　可变参数

在 Python 中，可变参数在函数定义中使用星号（*）或双星号（**）来传递多个参数。这允许在调用函数时传递任意数量的参数，可变参数允许函数接收不定数量的参数，使得函

数更加灵活。

星号(*)用于传递非关键字的可变参数列表:当使用星号作为参数时,所有位置参数都会被收集到一个元组中,如下所示:

```
def my_function(*args):        # *args 表示可变数量的参数
    """
    接收可变数量的参数并打印它们
    """
    for arg in args:
        print(arg)
#调用函数
my_function(1, 2, 3)           #输出: 1 2 3
my_function('a', 'b', 'c')     #输出: a b c
```

双星号(**)用于传递关键字的可变参数字典:当使用双星号作为参数时,所有关键字参数都会被收集到一个字典中,如下所示:

```
def my_function(**kwargs):          #**kwargs 表示可变数量的关键字参数
    """
    接收关键字参数并打印它们
    """
    for key, value in kwargs.items(): #items()返回一个包含关键字参数的元组列表
        print(key, value)
#调用函数
my_function(name="Alice", age=25, city="New York")
#输出: name Alice age 25 city New York
```

上述代码定义了一个名为 my_function 的函数,它接收任意数量的关键字参数(通过 **kwargs 表示)。在函数内部,使用 for 循环遍历这些关键字参数,并打印出每个参数的键和值。在代码的最后,调用了 my_function()函数,并传入了 3 个关键字参数:name、age 和 city。函数将打印出这些参数的键和值。

5.3 函数的返回值

在 Python 中,函数可以返回一个值,这个返回的值可以通过 return 语句来指定。当函数执行完毕时,return 语句会返回一个值,这个值会作为函数的结果。如果函数没有 return 语句,或者 return 语句没有赋值,那么函数将返回 None。

例 5.2 有一台可以制作冰淇淋的神奇机器。该机器有一个按钮,按下按钮就能制作出不同口味的冰淇淋。这个按钮类似一个“制作冰淇淋”函数,接收想要的口味,然后产生对应的冰淇淋。

我们可以使用以下指令让 Copilot 实现“制作冰淇淋”函数的代码。

```
//实现一个函数,机器有一个按钮,按下按钮就能制作出不同口味的冰淇淋,接收想要的口味,然后
  产生对应的冰淇淋
```

Copilot 生成的代码如图 5-1 所示。

```
def make_ice_cream(flavor):
    ice_cream_recipes = {
        "巧克力": "巧克力冰淇淋配方",
        "草莓": "草莓冰淇淋配方",
        "香草": "香草冰淇淋配方",
        # 添加更多口味和对应的配方
    }

    if flavor in ice_cream_recipes:
        recipe = ice_cream_recipes[flavor]
        print(f"制作{flavor}口味的冰淇淋：{recipe}")
    else:
        print(f"对不起，暂时没有{flavor}口味的冰淇淋配方。")

# 测试
make_ice_cream("巧克力")
make_ice_cream("草莓")
make_ice_cream("抹茶")
```

图 5-1　“制作冰淇淋”函数

输出结果如下：

```
制作巧克力口味的冰淇淋：巧克力冰淇淋配方
制作草莓口味的冰淇淋：草莓冰淇淋配方
对不起,暂时没有抹茶口味的冰淇淋配方
```

例 5.2 定义了一个名为 make_ice_cream 的函数，该函数接收一个参数 flavor，表示冰淇淋的口味。函数内部首先定义了一个字典 ice_cream_recipes，字典的键是冰淇淋的口味，值是对应口味的冰淇淋配方。然后，函数检查传入的口味 flavor 是否在 ice_cream_recipes 字典中。如果在，那么获取对应的冰淇淋配方，打印出一条消息，表示正在制作该口味的冰淇淋；如果不在，那么打印出一条消息，表示暂时没有该口味的冰淇淋配方。

在代码的最后，调用了 make_ice_cream 函数，并传入了"巧克力""草莓""抹茶"作为参数，所以函数将分别打印出制作这三种口味冰淇淋的消息。对于 "巧克力" 和 "草莓"，因为字典中有对应的配方，所以会打印出制作的消息；对于 "抹茶"，因为字典中没有对应的配方，所以会打印出没有配方的消息。

5.4　变量作用域

变量作用域是指一个变量在程序中的可见性和有效性。在不同的上下文环境中，变量可能具有不同的作用域。在 Python 中，根据范围作用的大小分为局部变量和全局变量，函数作用域决定了变量的可见性和访问权限。类比社会组织，函数可以看作一个小型的组织，而作用域则是这个组织的界限。

5.4.1　局部变量的定义和使用

局部变量是在函数内部定义的变量，其作用范围仅限于函数体内。这就好比在一个小房间里存放的物品，只有在这个房间里才能直接访问和使用。函数执行结束，局部变量的生命周期也结束。

定义局部变量:局部变量的定义非常简单,只需要在函数内部使用等号(=)进行赋值即可,如下所示:

```
def function():
#定义局部变量
    local_variable = "l am a local variable"
    print(local_variable)
#调用函数
function()
```

在上述示例中,local_variable 就是一个局部变量,它只能在 function 函数内部被直接访问。

局部变量的作用范围从变量定义的地方开始,直到函数结束。一旦函数执行完毕,局部变量就会被销毁,无法在函数外部直接访问,如下所示:

```
def function( ):
    #定义局部变量
    local_variable = "I am a local variable"
    print(local_variable)
#调用函数
function()
print(local_variable)        #报错,因为局部变量只能在函数内部使用
```

5.4.2 全局变量的定义和使用

全局变量是在整个程序中都可以访问的变量,其作用范围不仅限于单个函数,而是覆盖了整个代码。这就好比是在城市中建立的一个标志性建筑物,任何人都可以看到并使用它。

定义全局变量:全局变量通常在函数外部或模块级别进行定义,这使得全局变量在整个程序中都能被访问和修改,如下所示:

```
#定义全局变量
global_variable = "I am a global variable"
def function_using_global_variable():
    #在函数内部使用全局变量
    print(global_variable)
#调用函数
function_using_global_variable()
#输出: I am a global variable
```

在上述示例中,定义了一个全局变量 global_variable,并赋值为"I am a global variable"。全局变量是在函数外部定义的变量,可以在程序的任何地方使用。然后,定义了一个名为 function_using_global_variable 的函数。在这个函数内部,使用 print 语句打印出全局变量 global_variable 的值。

需要注意的是,全局变量的使用应谨慎,因为它们可以被程序中的任何地方修改,可能导致不可预测的行为。应避免在函数内部直接修改全局变量,除非有充分的理由。更好的做法是通过函数的参数和返回值来传递和获取信息。

当我们谈论“变量作用域”时,实际上是在谈论 Python 如何在不同的上下文中确定变量之间的关系和可见性。理解 Python 的作用域规则对于编写正确和可读的代码是非常重要的。

例 5.3　有一顶魔法帽子，里面有不同颜色的球。每个人都可以从帽子里抽一个球，每抽一次便输出对应的颜色，帽子里的球相应地减少。

我们可以使用以下指令让 Copilot 实现这个例题的代码。

```
#有一顶魔法帽子，里面有不同颜色的球，每个人都可以从帽子里抽一个球。使用 Python 中的函数模拟这个过程，通过全局变量和局部变量实现这个函数
```

Copilot 生成的代码如图 5-2 所示。

```
import random
# 全局变量，表示帽子里的球
hat = ['红色', '蓝色', '绿色', '黄色']

def draw_ball():
    ball = random.choice(hat)  # 局部变量，表示抽到的球
    hat.remove(ball)  # 从帽子里移除抽到的球
    return ball

# 测试函数
for _ in range(4):
    print(draw_ball())
```

图 5-2　“魔法帽子”函数

输出结果如下：

```
蓝色
红色
绿色
黄色
```

例 5.3 定义了一个名为 draw_ball 的函数，用于从全局变量 hat 中随机抽取一个球，并将其从 hat 中移除。在函数内部，首先使用 random.choice(hat)随机选择 hat 中的一个元素，将其赋值给局部变量 ball。这里的 random.choice 是 Python 的 random 模块提供的一个函数，用于从列表中随机选择一个元素。然后，使用 hat.remove(ball)将抽取到的球从 hat 中移除。这里的 remove 是 Python 列表的一个方法，用于移除列表中的一个元素。最后，函数返回抽取到的球。

这个函数可以用于模拟从一个装有不同颜色的球的帽子中随机抽球的过程。在每次抽球后，被抽到的球都会从帽子中移除，所以每次抽球的结果都会影响后续的抽球结果。

5.5　匿名函数 lambda

匿名函数又称为 lambda 函数，是一种没有具体名称的小型、临时性的函数，通常用于需要一个简单函数的场景，而不必显式地定义一个完整的函数。匿名函数通常只有一行，语法简洁，适用于简单的操作。

匿名函数的定义：使用 lambda 关键字可以创建匿名函数，语法如下所示：

```
#格式：lambda 参数列表：表达式
add = lambda x, y: x + y
```

在上述代码中,“lambda x,y: x+y”就是一个匿名函数,它接收两个参数 x 和 y,并返回它们的和。

匿名函数是 Python 编程中的一把利器,善用它可以使代码更加简洁、灵活。通过灵活应用匿名函数,读者可以在不增加过多函数定义的情况下完成许多任务。

例 5.4 在一个奇妙的餐馆中,顾客可以通过匿名点餐的方式来定制不同的菜品。我们可以使用匿名函数来创建特殊的点餐规则。

引导 Copilot 生成代码的指令如下:

```
//通过匿名点餐的方式来定制不同的菜品,创建特殊的点餐规则
```

Copilot 生成的代码如图 5-3 所示。

```
# 匿名函数 - 定制点餐规则
customize_order = lambda dish, customization: f"{dish}（{customization}）"

def place_order(dish, customization_function):
    # 调用匿名函数
    customized_dish = customization_function(dish)
    print(f"点餐成功: {customized_dish}")

# 定制点餐规则
spicy_order = lambda dish: customize_order(dish, "辣味")
vegan_order = lambda dish: customize_order(dish, "素食")

# 点餐
place_order("牛肉面", spicy_order)
place_order("披萨", vegan_order)
```

图 5-3 定制点餐规则

输出结果如下:

```
点餐成功: 牛肉面(辣味)
点餐成功: 比萨(素食)
```

在例 5.4 中,customize_order 是一个匿名函数,接收两个参数 dish(菜品)和 customization(定制要求),返回一个带有定制信息的字符串。spicy_order 和 vegan_order 是两个匿名函数,分别用于为菜品添加辣味和标记为素食。place_order 函数接收菜品名称和一个定制函数作为参数,调用匿名函数,生成定制后的菜品信息,并打印出点餐成功的信息。

5.6 递归函数

递归函数是一种在函数内部调用自身的函数,它通过不断地调用自身来解决问题,是一种强有力的编程技巧,能够简化复杂问题的解决过程。在现实生活中,我们也应该如此,当我们遇到困难挫折时,首先要做的是挖掘自己的潜力(递归函数调用自己),向内寻求力量,先尽全力解决问题,实在解决不了时再向他人寻求帮助,而不是一碰到困难挫折就向他人寻求帮助。只有不断地挖掘自身潜力,才能不断提高自己,使自己变得越来越强大。

5.6.1 递归函数的定义与调用

递归函数的基本原理是将一个大问题划分为一个或多个相同或相似的小问题来解决,

并将解决小问题的过程自动化。递归函数通常包含两部分：基本情况和递归步骤。

- **基本情况（终止条件）**：定义一个或多个基本情况，当满足基本情况时，不再调用自身，直接返回结果。
- **递归调用**：在函数内部调用自身，将原问题转化为更小的子问题，并通过递归调用解决子问题。

递归函数的定义必须有终止条件（基本情况），确保递归能在某个条件下停下来，避免陷入无限循环，下面通过斐波那契数列来实现递归函数的定义。

例 5.5　斐波那契数列的递归函数定义。

引导 Copilot 生成代码的指令如下：

```
//斐波那契数列
```

引导 Copilot 生成的代码如图 5-4 所示。

```
def fibonacci(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        fib = [0, 1]
        for i in range(2, n):
            fib.append(fib[i-1] + fib[i-2])
        return fib

n = int(input("请输入斐波那契数列的长度："))
fib_seq = fibonacci(n)
print(fib_seq)
```

图 5-4　斐波那契数列

当输入长度为 10 时，输出结果如下：

```
[0,1,1,2,3,5,8,13,21,34]
```

例 5.5 定义了一个名为 fibonacci 的递归函数，用于生成斐波那契数列。这个函数接收一个参数 n，表示要生成的斐波那契数列的长度。在函数内部，首先定义了递归的终止条件：如果 n 小于或等于 0，则返回空列表；如果 n 等于 1，则返回包含一个元素 0 的列表；如果 n 等于 2，则返回包含两个元素（0 和 1）的列表。

如果 n 大于 2，则函数将调用自身，传入的参数是 n－1，也就是生成长度为 n－1 的斐波那契数列。然后，计算斐波那契数列的下一个数（数列中最后两个数的和），并将其添加到数列的末尾。最后，返回生成的斐波那契数列。在代码的最后，调用了 fibonacci 函数，传入的参数是 10，所以函数将生成长度为 10 的斐波那契数列，并将其打印出来。

5.6.2　递归函数的应用与注意事项

递归函数在问题分解方面具有强大的应用能力。通过将一个大问题划分为更小、相似的子问题，递归函数可以更容易地解决复杂的任务。

- **数据结构操作**:递归常用于对树、图等数据结构进行操作。例如,在树的遍历、搜索或构建过程中,递归可以提供一种简洁而有效的方法。
- **编程语言解释器实现**:递归也是一些编程语言解释器实现的基本原理之一,例如函数调用栈就是通过递归的方式来管理的。
- **数学计算**:数学中的一些问题,如计算阶乘、斐波那契数列等,天然地适合使用递归来表达和解决。
- **分治算法**:递归函数可以用于分治算法的实现,如归并排序和快速排序等。

尽管递归函数非常强大,但也需要注意一些事项。

- **基本情况**:递归函数必须包含一个或多个基本情况,否则函数将无限循环并导致堆栈溢出。
- **递归深度**:递归函数的递归深度越大,函数调用栈的大小也会越大。如果递归深度过大,可能会导致堆栈溢出。在使用递归函数时,需要评估问题的大小和系统的限制。
- **性能问题**:递归函数可能在某些情况下比迭代循环慢,因为每次递归调用都会有额外的开销。在解决问题之前,需要评估性能要求,并根据实际情况考虑使用递归或迭代。
- **空间复杂度**:递归函数的空间复杂度可能比较高,因为每次递归调用都会在堆栈中保存一些信息。如果问题的规模较大,递归函数可能需要大量的内存空间。
- **代码可读性**:递归函数可以提供一种简洁的解决方案,但有时可能更难理解和调试。在编写递归函数时,应确保代码易于理解和维护。

5.7 常见的内置函数

在 Python 中,内置函数是由解释器提供的、可供开发者直接使用的函数。这些函数构成了 Python 的核心库,涵盖各种功能,包括基本的数学运算到高级的数据结构操作。

5.7.1 数据类型转换函数

数据类型转换是 Python 中常用的操作之一,可以将一个数据类型转换为另一种数据类型。Python 内置了多个数据类型转换函数,如表 5-1 所示。

表 5-1 常见数据类型转换函数

函数名称	描述说明	函数名称	描述说明
int(x)	将 x 转换为整数类型	list(x)	将 x 转换为列表类型
float(x)	将 x 转换为浮点数类型	tuple(x)	将 x 转换为元组类型
str(x)	将 x 转换为字符串类型	set(x)	将 x 转换为集合类型
bool(x)	将 x 转换为布尔类型		

下面逐个介绍这些函数的用法和示例。

引导 Copilot 生成的代码如下所示: