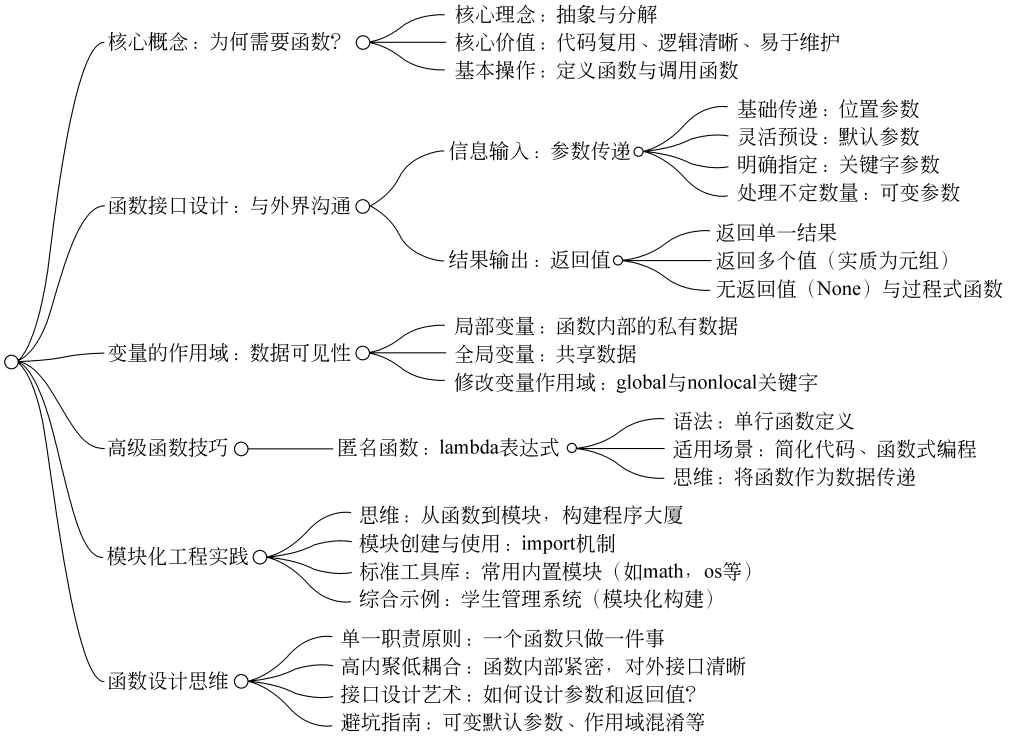
第5章　函　　数

在前面的章节中介绍了 Python 的基本语法、数据类型、控制结构等内容。随着程序功能的增加，代码也变得越来越长，越来越复杂。如果把所有代码都写在一起，不仅难以阅读和理解，而且当需要修改某个功能时，也很难快速定位到相应的代码位置。更重要的是，当程序中需要多次执行相同或相似的操作时，重复编写相同的代码不仅效率低下，而且容易出错。

函数(function)是解决这些问题的有效工具。函数是组织好的、可重复使用的代码块，用来实现单一或相关联的功能。通过函数，可以将复杂的程序分解成若干功能模块，每个模块负责完成特定的任务。这种模块化的编程方式不仅使代码结构更加清晰，便于理解和维护，还能大大提高代码的复用性。

事实上，在前面的学习中已经接触过许多 Python 内置函数，如 print()、input()、len()、range()等。这些函数帮助我们完成了输出、输入、计算长度、生成序列等各种操作。本章将带领读者学习如何定义和使用自己的函数，掌握函数参数、返回值、作用域等核心概念，并了解 lambda 表达式和函数的高级应用。通过本章的学习，相信读者能够编写出结构更加清晰、功能更加强大的 Python 程序。本章内容的思维导图如下：



5.1　函数的基本概念

5.1.1　什么是函数

函数是一段具有特定功能的、可重复使用的代码块。在数学中，我们学过函数的概念：给定一个输入值，函数会根据特定的规则产生一个输出值。编程中的函数与此类似，它接收输入数据（称为参数），经过一系列处理后，产生输出结果（称为返回值）。

从本质上讲，函数是对一组操作的封装。可以给这组操作起一个有意义的名字，然后在需要时通过函数名来调用它，而不必关心函数内部的具体实现细节。这种“封装”的思想是程序设计中非常重要的概念。

例如，当使用 print()函数时，只需要知道它能将内容输出到屏幕上，而不需要了解它是如何与操作系统交互、如何控制显示器显示字符的。这就是函数封装的好处，即隐藏复杂的实现细节，提供简洁的使用接口。

5.1.2　为什么需要函数

在实际编程中，使用函数有以下几个重要的优势。

1. 提高代码复用性

当程序中需要多次执行相同的操作时，如果每次都重复编写相同的代码，不仅工作量大，而且一旦需要修改，就要在多处进行修改，容易遗漏或出错。使用函数后，只需编写一次代码，就可以在程序的任何地方调用。

例如，假设我们需要在程序的多个地方计算圆的面积。如果不使用函数，代码可能是这样的：

```
#计算第一个圆的面积
radius1 =5
area1 =3.14159 * radius1 * radius1
print("圆 1 的面积:", area1)
#计算第二个圆的面积
radius2 =8
area2 =3.14159 * radius2 * radius2
print("圆 2 的面积:", area2)
#计算第三个圆的面积
radius3 =3
area3 =3.14159 * radius3 * radius3
print("圆 3 的面积:", area3)
```

这段代码中，计算圆面积的公式被重复写了 3 次。如果使用函数，代码会简洁很多：

```
def circle_area(radius):
    return 3.14159 * radius * radius
```

```
print("圆 1 的面积:", circle_area(5))
print("圆 2 的面积:", circle_area(8))
print("圆 3 的面积:", circle_area(3))
```

2. 提高程序的可读性和可维护性

通过将程序分解为多个函数,每个函数完成一个特定的任务,程序的结构会更加清晰。函数名本身就说明了这段代码的功能,使程序更容易理解。同时,当需要修改某个功能时,只需要修改对应的函数即可,不会影响程序的其他部分。

3. 降低程序的复杂度

复杂的问题可以分解为若干简单的子问题,每个子问题用一个函数来解决。这种“分而治之”的思想使得解决复杂问题变得更加容易。程序员可以专注于每个小功能的实现,而不必一次性处理整个复杂的逻辑。

4. 便于团队协作

在团队开发中,不同的成员可以负责开发不同的函数模块。只要约定好函数的接口(参数和返回值),各个模块就可以独立开发和测试,最后组装在一起形成完整的程序。

5.1.3 函数的定义与调用

1. 函数的定义

在 Python 中,使用 def 关键字来定义函数,基本语法格式如下:

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

接下来通过一个具体的例子来理解函数的定义:

```
def greet():
    """这是一个问候函数"""
    print("你好,欢迎学习 Python!")
```

这个函数的各个组成部分说明如下:

(1) def:定义函数的关键字;

(2) greet:函数名,应该使用有意义的名称,遵循标识符命名规则;

(3) ():参数列表,即使没有参数也要保留括号;

(4) :冒号表示函数定义的开始;

(5) """函数文档字符串""":可选的函数说明,描述函数的功能,用三引号括起来;

(6) print("你好,欢迎学习 Python!"):函数体,即函数要执行的具体代码,必须缩进,这个函数没有 return 语句,表示不返回任何值。

下面是一个带参数和返回值的函数示例：

```
def add(a, b):
    """计算两个数的和"""
    result =a +b
    return result
```

这个函数接收两个参数 a 和 b，计算它们的和，并通过 return 语句返回结果。

2. 函数的调用

定义函数只是创建了一个可执行的代码块，要让函数真正执行，需要调用它。函数调用的语法格式为

```
函数名(实际参数)
```

如调用前面定义的 greet()函数：

```
greet()    #输出：你好，欢迎学习 Python!
```

调用 add()函数：

```
result =add(3, 5)
print(result)      #输出：8
```

在这个例子中，3 和 5 是传递给函数的实际参数，函数执行后返回的结果被赋值给变量 result。

需要注意的是，函数必须先定义后调用。如果在函数定义之前调用函数，Python 会报错：

```
#错误示例
hello()       #报错：NameError: name 'hello' is not defined
def hello():
    print("Hello!")
```

正确的做法是先定义函数，再调用：

```
def hello():
    print("Hello!")
hello()    #正确：输出 Hello!
```

下面通过一个完整的例子来演示函数的定义和调用。

例 5-1　定义并调用一个函数，计算矩形面积。

```
def calculate_rectangle_area(length, width):
    """
    计算矩形的面积
    参数：
        length：矩形的长
        width：矩形的宽
    返回值：
        矩形的面积
    """
```

```
    area = length * width
    return area
#调用函数
rect_area = calculate_rectangle_area(10, 5)
print("矩形的面积是:", rect_area)                                    #输出: 矩形的面积是: 50
#直接在 print 中调用函数
print("另一个矩形的面积是:", calculate_rectangle_area(8, 6)) #输出: 另一个矩形的
                                                                    #面积是: 48
```

在这个例子中,定义了一个计算矩形面积的函数,并演示了两种常见的函数调用方式:一种是将返回值赋给变量后使用,另一种是直接在其他语句中调用函数。

通过学习本节,可以了解函数的基本概念、使用函数的好处,以及如何定义和调用函数。在接下来的章节中,将深入学习函数的参数、返回值等更多内容。

5.2 函数的参数

参数是函数与外部进行数据交互的重要途径。通过参数,可以向函数传递不同的数据,使函数能够处理各种情况。如果把函数比作一个加工车间,那么参数就是车间的原材料输入口,不同的原材料经过同样的加工流程,可以产出不同的产品。Python 提供了多种参数类型,使函数的使用更加灵活和强大。

在函数定义时写在括号内的参数称为形式参数(形参),它们只是一些符号,用于说明函数需要接收什么样的数据。而在调用函数时传递给函数的具体数值称为实际参数(实参)。函数执行时,实参的值会传递给对应的形参,供函数内部使用。理解形参和实参的区别对于正确使用函数参数非常重要。

5.2.1 位置参数

位置参数是最基本、最常用的参数类型。它的特点是:参数的传递顺序非常重要,调用函数时传递的实际参数会按照位置顺序依次赋值给函数定义中的形式参数。这就像排队买票,第一个人拿到 1 号票,第二个人拿到 2 号票,顺序不能乱。

1. 基本用法

例 5-2 使用位置参数传递用户信息并打印。

```
def introduce(name, age, city):
    """介绍个人信息"""
    print(f"我叫{name},今年{age}岁,来自{city}。")
#调用函数时,参数按位置传递
introduce("张三", 20, "北京")      #输出: 我叫张三,今年 20 岁,来自北京。
```

在这个例子中,"张三"对应 name,20 对应 age,"北京"对应 city。参数的顺序非常重要,如果顺序错误,会导致结果不符合预期:

```
introduce("北京", 20, "张三")      #输出: 我叫北京,今年 20 岁,来自张三。
```

虽然程序不会报错,但结果显然是错误的。这说明使用位置参数时,必须清楚地记住每个参数的含义和顺序。

2. 位置参数的数量必须匹配

调用函数时,传递的实际参数数量必须与函数定义的形式参数数量一致,否则会报错。这是 Python 的强制要求,目的是确保函数能够获得所有必需的数据来完成其功能,例如:

```
introduce("李四", 22)                          #报错: 缺少 1 个必需的位置参数'city'
introduce("王五", 21, "上海", "学生")          #报错: 接收了 4 个位置参数,但只定义了 3 个
```

第一个调用缺少了 city 参数,Python 无法知道应该为 city 赋什么值;第二个调用传递了多余的参数,Python 也不知道应该如何处理。这两种情况都会导致程序抛出 TypeError 异常。

3. 位置参数的应用场景

位置参数适用于那些参数含义明确、数量固定、调用频繁的函数。例如,数学运算函数、几何图形计算函数等,这些函数的参数含义清晰,使用位置参数既简洁又高效。

5.2.2　默认参数

默认参数(也称为缺省参数)是指在定义函数时为参数设置默认值。如果调用函数时没有为该参数传值,就使用默认值;如果传值了,则使用传递的值。这种机制大大增强了函数的灵活性,使得函数既可以处理常见的简单情况,也可以应对复杂的特殊需求。

1. 基本用法

例 5-3　使用默认参数简化问候语函数调用。

```
def greet(name, greeting="你好"):
    """问候函数,greeting 有默认值"""
    print(f"{greeting},{name}!")
#使用默认值
greet("小明")                  #输出: 你好,小明!
#指定新的值
greet("小红", "早上好")        #输出: 早上好,小红!
```

在这个例子中,greeting 参数有一个默认值"你好"。当只传递一个参数时,函数使用默认的问候语;当传递两个参数时,第二个参数会覆盖默认值。这种设计使得函数在保持简洁的同时,又具备了足够的灵活性。

2. 默认参数的优势

默认参数的主要优势在于简化函数调用。在实际编程中,很多函数的某些参数在大多数情况下使用相同的值,只在少数特殊情况下需要修改。如果没有默认参数机制,每次调用

函数都必须显式地传递所有参数，即使它们的值是重复的。使用默认参数后，只需在特殊情况下传递非默认值，大幅减少了代码量。

例 5-4 使用默认参数实现幂运算，默认求平方。

```
def power(base, exponent=2):
    """计算幂,默认计算平方"""
    return base * * exponent
print(power(3))            #输出: 9(3 的平方)
print(power(3, 3))         #输出: 27(3 的立方)
print(power(2, 5))         #输出: 32(2 的五次方)
```

在这个幂运算函数中，由于求平方是最常见的操作，可以将 exponent 的默认值设为 2。这样在计算平方时只需传递一个参数，代码更加简洁明了。而当需要计算其他次方时，只需传递第二个参数即可。

3. 默认参数的定义规则

在定义函数时，默认参数必须放在非默认参数(位置参数)之后，否则会报错。这个规则的原因是：如果默认参数在前，Python 在解析参数时就无法判断传递的参数应该对应哪个形参。

```
#错误示例
def wrong_function(a=1, b):          #报错: 非默认参数不能出现在默认参数之后
    return a +b
#正确示例
def correct_function(b, a=1):
    return a +b
```

想象一下，如果允许 wrong_function 这样的定义，当调用 wrong_function(5)时，Python 无法判断 5 应该赋给 a 还是 b。因此，Python 强制要求所有默认参数必须在非默认参数之后，这样在解析参数时就不会产生歧义。

4. 默认参数的陷阱

需要特别注意的是，默认参数的值在函数定义时就已经确定，而不是在函数调用时。这个特性对于不可变对象(如整数、字符串、元组)来说没有问题，但当默认参数是可变对象(如列表、字典)时，可能会出现意想不到的结果。

例 5-5 演示默认参数为可变对象时的陷阱及修复方法。

```
def add_item(item, item_list=[]):
    """向列表中添加元素"""
    item_list.append(item)
    return item_list
#第一次调用
print(add_item("苹果"))              #输出: ['苹果']
#第二次调用
print(add_item("香蕉"))              #输出: ['苹果', '香蕉']注意: 不是['香蕉']!
```

这个现象产生的原因在于，默认参数列表“item_list=[]”在函数定义时就创建了一个

列表对象，这个对象在内存中只有一份。每次调用函数时如果不传递 item_list 参数，都会使用同一个列表对象。因此，第二次调用时列表中已经包含了第一次添加的元素。

这种行为在某些特定场景下可能是有用的，但在大多数情况下不是我们想要的。正确的做法是使用 None 作为默认值，然后在函数内部创建新的可变对象：

```
def add_item(item, item_list=None):
    """向列表中添加元素"""
    if item_list is None:
        item_list =[]
    item_list.append(item)
    return item_list
print(add_item("苹果"))              #输出:['苹果']
print(add_item("香蕉"))              #输出:['香蕉']
```

这样，每次调用函数时如果没有传递 item_list 参数，都会创建一个新的空列表，避免了多次调用之间的相互影响。

5.2.3　关键字参数

关键字参数是指在调用函数时，通过“参数名＝值”的形式明确指定每个参数的值。使用关键字参数可以不必关心参数的顺序，只需要知道参数的名称即可。这种方式使函数调用更加清晰明确，特别是当函数有多个参数时，关键字参数能够显著提高代码的可读性。

1. 基本用法

例 5-6　使用关键字参数传递用户信息，忽略顺序。

```
def student_info(name, age, major, grade):
    """显示学生信息"""
    print(f"姓名: {name}")
    print(f"年龄: {age}")
    print(f"专业: {major}")
    print(f"年级: {grade}")
#使用关键字参数调用,顺序可以任意
student_info(age=19, name="李华", grade="大一", major="计算机科学")
#输出结果:
姓名: 李华
年龄: 19
专业: 计算机科学
年级: 大一
```

在这个例子中，尽管参数的传递顺序与函数定义时的顺序不同，但由于使用了参数名进行明确指定，Python 能够正确地将每个值赋给对应的参数。这种方式特别适合参数较多的函数，避免了因参数顺序错误而导致的逻辑错误。

2. 混合使用位置参数和关键字参数

在实际使用中，可以同时使用位置参数和关键字参数，但必须遵循以下规则：位置参数

必须在关键字参数之前。这个规则确保了参数解析的明确性和一致性。

```
#正确示例
student_info("王明", 20, major="软件工程", grade="大二")
#错误示例
student_info(name="王明", 20, major="软件工程", grade="大二")    #报错
```

混合使用的优势在于：对于那些位置固定、含义明确的参数可以使用位置方式传递，这样更简洁；而对于那些可选的或容易混淆的参数则使用关键字方式传递，这样更清晰。这种混合方式在实际编程中非常常见。

3. 关键字参数的优势

使用关键字参数有以下几个显著的好处。

(1) 提高代码可读性。当函数有多个参数时，关键字参数能够明确指出每个参数的含义。读代码的人不需要去查看函数定义就能理解每个参数的用途。例如，create_user(username="admin", email="admin@example.com", age=25)比 create_user("admin", "admin@example.com", 25)更容易理解。

(2) 避免参数顺序错误。使用位置参数时，如果参数类型相同(如都是字符串或都是数字)，很容易因为顺序错误而导致逻辑错误，且这种错误往往难以发现。关键字参数通过明确的参数名避免了这个问题。

(3) 与默认参数配合使用更灵活。当函数有多个默认参数时，使用关键字参数可以只修改需要改变的参数，而不必按顺序传递所有参数。

例 5-7 混合使用位置参数和关键字参数创建用户。

```
def create_user(username, email, age=18, city="未知", is_active=True):
    """创建用户信息"""
    return {
        "username": username,
        "email": email,
        "age": age,
        "city": city,
        "is_active": is_active
    }
#只修改需要改变的参数
user1 =create_user("zhangsan", "zhangsan@example.com")
print(user1)
user2 =create_user("lisi", "lisi@example.com", city="上海", age=25)
print(user2)
```

在这个例子中，第二次调用只修改了 city 和 age，而 is_active 保持默认值。如果使用位置参数，必须写成 create_user("lisi", "lisi@example.com", 25, "上海")，这就要求我们必须记住所有参数的顺序，并且无法跳过中间的参数。

5.2.4 可变参数

在某些情况下，无法预先确定函数需要接收多少个参数。例如，一个求和函数可能需要

计算 2 个数的和，也可能需要计算 10 个数的和；一个数据记录函数可能需要记录不同数量的信息项。Python 的*args 和**kwargs 提供了可变参数机制，允许函数接收任意数量的参数，使函数的适用范围更加广泛。

1. *args

args 用于接收任意数量的位置参数，这些参数在函数内部被组织成一个元组(tuple)。 是 Python 中的解包操作符，它告诉 Python 将所有的位置参数收集到一个元组中。参数名 args 是约定俗成的写法，代表"arguments"(参数)的缩写，可以使用其他名称，但为了代码的可读性和一致性，建议使用 args。

例 5-8　使用*args 实现任意数量数字求和。

```
def sum_numbers(*numbers):
    """计算任意多个数字的和"""
    total =0
    for num in numbers:
        total +=num
    return total
print(sum_numbers(1, 2, 3))          #输出: 6
print(sum_numbers(10, 20, 30, 40))   #输出: 100
print(sum_numbers(5))                #输出: 5
print(sum_numbers())                 #输出: 0
```

在这个例子中，参数*numbers 可以接收 0 个、1 个或多个参数，它们在函数内部被当作元组 numbers 来处理。函数内部通过遍历这个元组来累加所有的数值。这种设计使得同一个函数可以处理不同数量的参数，大幅提高了函数的通用性。

可变参数特别适合那些操作对象数量不固定的场景。例如，打印多条信息、查找多个元素、合并多个数据等。

```
def print_info(*items):
    """打印所有传入的信息"""
    for item in items:
        print(item)
print_info("Python", "Java", "C++", "JavaScript")
```

需要注意的是，*args 必须放在普通位置参数和默认参数之后，因为它会“吃掉”所有剩余的位置参数。如果*args 放在前面，后面的参数就无法通过位置方式传递了。

2. **kwargs

kwargs 用于接收任意数量的关键字参数，这些参数在函数内部被组织成一个字典(dictionary)。 同样是解包操作符，它告诉 Python 将所有的关键字参数收集到一个字典中，其中参数名作为键(Key)，参数值作为值(Value)。参数名 kwargs 是 keyword arguments(关键字参数)的缩写。

例 5-9　使用**kwargs 接收任意关键字参数并打印。

```
def show_person_info(**info):
    """显示个人信息"""
```