第九章 - 函数

张建章

阿里巴巴商学院

杭州师范大学

2024-09

抽象是计算机科学中的核心概念，旨在隐藏复杂的实现细节，突出核心功能，从而提高程序的可读性、可维护性和扩展性。

在编程实践中，函数抽象是实现这一目标的主要手段之一。通过将重复的逻辑封装在函数中，开发者可以减少代码冗余，提升代码的模块化程度。

例如，考虑一个需要计算多个矩形面积的场景。如果不使用函数，可能会多次编写相同的计算逻辑：

```
# 计算第一个矩形的面积
width1 = 5
height1 = 10
area1 = width1 * height1

# 计算第二个矩形的面积
width2 = 3
height2 = 7
area2 = width2 * height2
# 计算第三个矩形的面积
```

上述代码存在明显的重复，增加了维护难度。通过定义一个计算矩形面积的函数，可以有效地抽象出重复的逻辑：

```
def calculate_area(width, height):
    return width * height

# 使用函数计算面积
area1 = calculate_area(5, 10)
area2 = calculate_area(3, 7)
area3 = calculate_area(6, 9)
```

通过这种方式，计算面积的逻辑被封装在 `calculate_area` 函数中，调用者只需提供不同的参数即可复用该逻辑。这不仅减少了代码冗余，还提高了代码的清晰度和可维护性。

函数的定义使用 `def` 关键字，后跟函数名和括号中的参数列表。函数内部的代码块通常由缩进的语句组成，而 `return` 语句用于返回函数的结果。

1. 函数的定义与调用

函数的基本定义格式如下：

```
def function_name(parameters):
    # 执行的代码块
    return result
```

在定义函数时，`def` 后接函数名，再由圆括号包围的参数列表，可以为函数的参数指定默认值。如果函数没有参数，可以省略。如果函数需要返回一个结果，可以使用 `return` 语句将计算结果返回给调用者。

2. 参数的传递

Python 支持多种参数传递方式，包括位置参数、关键字参数和混合方式。位置参数 (调用时不写参数名字) 是最常见的类型，调用时传入的值按照位置匹配到相应的参数。

```
def add(a, b):
    return a + b

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

除了位置参数，Python 还支持使用关键字参数 (调用时写参数名字) 来进行函数调用，这使得传递参数时不受位置顺序的限制。

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(animal_type="dog", pet_name="Buddy")
```

3. 混合方式

混合使用位置参数和关键字参数进行函数调用时，位置参数必须在关键字参数前面。

```
def multiply(a, b):
    return a * b

print(multiply(4, b = 5))   # 输出 20
print(multiply(a = 4, 5))   # 错误，位置参数必须在关键字参数前面
```

4. 返回值的使用

函数可以返回值，`return` 语句用于指定返回值。如果函数没有 `return` 语句，它会默认返回 `None`。返回的值可以用于后续的计算。

```
def multiply(a, b):
    return a * b

result = multiply(4, 5)
print(result)  # 输出 20
```

5. 示例代码：带参数的函数

以下是一个包含多个参数、返回值以及默认参数的完整示例：

```python
def calculate_area(length, width=1):
    """ 计算矩形的面积，宽度参数有默认值"""
    return length * width

# 使用位置参数
area1 = calculate_area(5, 3)
print(f"Area 1: {area1}")  # 输出 Area 1: 15

# 使用默认参数
area2 = calculate_area(5)
print(f"Area 2: {area2}")  # 输出 Area 2: 5
```

在此示例中，`width` 参数有默认值，因此在调用 `calculate_area(5)` 时，`width` 会自动取默认值 `1`。

6. 文档字符串

文档字符串（docstring）和函数注解（function annotation）是提高代码可读性和可维护性的关键工具。文档字符串用于描述模块、类或函数的功能和用法，而函数注解用于为函数的参数和返回值提供类型提示。

文档字符串是位于模块、类或函数定义内部的字符串字面量，通常使用三重引号（`"""  """`）包裹。文档字符串应简洁明了，首行应为简短的描述，后续可包含更详细的说明。

```python
def add(a, b):
    """
    返回两个数的和。

    参数:
    a (int): 第一个加数。
    b (int): 第二个加数。

    返回:
    int: 两个数的和。
    """
    return a + b
```

在上述示例中，函数 `add` 的文档字符串清晰地描述了函数的功能、参数和返回值。这有助于用户快速理解函数的用途和使用方法。

7. 函数注解（Function Annotation）

函数注解是 Python 3 引入的特性，用于为函数的参数和返回值添加元数据，通常用于类型提示。注解的语法是在参数名后使用冒号加类型提示，返回值注解则在参数列表后使用箭头加类型提示。函数注解仅用于提供信息，不会影响函数的实际行为。

```
def add(a: int, b: int) -> int:
    return a + b
```

在此示例中，函数 `add` 的参数 `a` 和 `b` 以及返回值均被注解为整数类型。这为阅读代码的人提供了关于参数和返回值类型的有用信息。

1. 形式参数与实际参数

函数的定义和调用涉及两个关键概念：形式参数（formal parameters）和实际参数（actual parameters）。形式参数是在函数定义时指定的变量，用于接收传入的数据；实际参数则是在函数调用时提供的具体值或表达式。

```
def multiply(a, b):
    return a * b

result = multiply(4, 7)
print(result)  # 输出: 28
```

在此示例中，`a` 和 `b` 是形式参数，`4` 和 `7` 是实际参数。调用 `multiply(4, 7)` 时，实际参数 `4` 和 `7` 被传递给形式参数 `a` 和 `b`，函数返回它们的乘积 `28`。

2. 默认参数（Default Arguments）

默认参数是函数定义时为参数指定的默认值。如果在调用时没有为该参数提供值，函数将使用默认值。

```
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("Bob")  # 使用默认值
greet("Alice", 30)  # 使用提供的值
```

在这个例子中，`age`参数有一个默认值`25`。如果在调用函数时没有传入`age`的值，默认值`25`将被使用。

注意：定义函数时，有默认值的参数在后，没默认值的参数在前。

3. 不定长参数（Arbitrary Arguments）

当不确定传入函数的参数个数时，可以使用不定长参数。Python 提供了 `*args` 和 `**kwargs` 两种方式来处理这种情况。

- `*args` 用于传递任意数量的位置参数。
- `**kwargs` 用于传递任意数量的关键字参数。

```python
def sum_numbers(*numbers):
    print(type(numbers))
    print(numbers)
    total = sum(numbers)
    print(f"Sum: {total}")

sum_numbers(1, 2, 3, 4)  # 传入多个位置参数
```

在这个例子中，`*numbers` 接受了多个位置参数并将它们放入一个元组 `numbers` 中，之后通过 `sum()` 函数计算它们的和。

```python
def display_info(**info):
    print(type(info))
    print(info)
    for key, value in info.items():
        print(f"{key}: {value}")

display_info(name="David", age=45, job="Engineer")
```

在这个例子中，`**info` 接受了多个关键字参数并将它们存储为一个字典 `info`，可以通过 `items()` 方法遍历字典中的每一对键值。

在 Python 编程中，变量的作用域决定了变量的可访问范围。主要分为局部变量和全局变量两种类型。

1. 局部变量（Local Variables）

局部变量是在函数内部定义的变量，其作用域仅限于该函数内部。当函数被调用时，局部变量被创建；函数执行结束后，局部变量被销毁。局部变量在函数外部无法访问。

```
def example_function():
    local_var = "I am a local variable"
    print(local_var)

example_function()
print(local_var)  # 试图在函数外部访问局部变量
```

上例中，`local_var` 是在函数 `example_function` 内部定义的局部变量。在函数内部打印该变量时，输出正常。然而，当尝试在函数外部访问 `local_var` 时，Python 抛出 `NameError`，提示未定义该变量。

2. 全局变量（Global Variables）

全局变量是在函数外部定义的变量，其作用域覆盖整个模块 (整个.ipynb 文件或.py 文件)。全局变量可以在函数内部和外部访问。然而，在函数内部如果需要修改全局变量的值，必须使用 `global` 关键字声明，否则 Python 会将其视为新的局部变量。

```
global_var = "I am a global variable"

def example_function():
    print(global_var)  # 在函数内部访问全局变量

example_function()
print(global_var)  # 在函数外部访问全局变量
```

在上例中，`global_var` 是在函数外部定义的全局变量。在函数 `example_function` 内部和外部均可访问该变量，且输出结果一致。

3. 在函数内部修改全局变量

如果需要在函数内部修改全局变量的值，必须使用 `global` 关键字声明该变量。否则，Python 会在函数内部创建一个同名的局部变量，而不会影响全局变量的值。

```
global_var = "I am a global variable"

def example_function():
    global global_var
    global_var = "I have been modified"
    print(global_var)

example_function()
print(global_var)  # 检查全局变量是否被修改
```

上例中，使用 `global` 关键字声明 `global_var`，表示在函数内部对全局变量进行修改。因此，函数内部和外部的 `global_var` 值均被修改。

函数式编程（Functional Programming）是一种编程范式，强调使用纯函数进行计算，避免副作用，并鼓励函数作为一等公民。在 Python 中，尽管其主要是面向对象的，但也提供了对函数式编程的支持。

1. 纯函数

纯函数是指在相同输入下总是产生相同输出且没有副作用的函数。这意味着函数的执行不依赖于外部状态，也不改变外部状态。

```
def add(a, b):
    return a + b
```

上述函数 `add` 在相同的输入 `a` 和 `b` 下，总是返回相同的结果 `a + b`，且不影响外部状态，因此是一个纯函数。

2. lambda 表达式

lambda 表达式用于创建匿名函数，即没有名称的简短函数。其语法形式为：

```
lambda 参数列表: 表达式
```

参数列表可以包含多个参数，表达式是对这些参数进行操作并返回结果。lambda 表达式常用于需要简短函数且不想正式定义函数的场景。

```
# 定义一个 lambda 表达式，将输入值加 10
add_ten = lambda x: x + 10
print(add_ten(5))  # 输出: 15
```

上例定义了一个 lambda 表达式并将其赋值给变量 `add_ten`，接受一个参数 `x`，返回 `x` 加 10 的结果。调用 `add_ten(5)` 时，输出为 15。

3. 高阶函数

高阶函数是指接受一个或多个函数作为参数，或返回一个函数作为结果的函数。Python 内置的 `map`、`filter` 和 `reduce` 函数就是高阶函数的典型例子。

(1) `map` 函数

`map` 函数用于将指定的函数依次作用于可迭代对象的每个元素，返回一个包含结果的迭代器。其语法为：

```
map(function, iterable, ...)
```

- `function`：应用于每个元素的函数；
- `iterable`：一个或多个可迭代对象。

```python
# 将列表中的每个元素平方
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))  # 输出: [1, 4, 9, 16, 25]
```

在此示例中，`map` 函数将匿名函数 `lambda x: x**2` 应用于列表 `numbers` 的每个元素，返回其平方值的迭代器。使用 `list` 函数将其转换为列表后，输出结果为 `[1, 4, 9, 16, 25]`。

`map` 可接受多个可迭代对象，如下例：

```python
weights = [7, 9, 10, 5, 8, 4, 2, 1, 6, 3, 7, 9, 10, 5, 8, 4, 2]
ids = '11011319870212l018'
sum(map(lambda x,y:x*int(y),weights,ids)) # 169
```

(2) filter 函数

filter 函数用于过滤可迭代对象中的元素，保留使指定函数返回 True 的元素，返回一个迭代器。其语法为：

```
filter(function, iterable)
```

- function：用于判断每个元素是否保留的函数，返回布尔值。
- iterable：可迭代对象。

```
# 过滤出列表中的偶数
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # 输出: [2, 4, 6]
```

在上例中，filter 函数将匿名函数 lambda x: x % 2 == 0 应用于列表 numbers 的每个元素，保留偶数元素。返回一个 filter 类型迭代器。

(3) reduce 函数

reduce 函数用于对可迭代对象中的元素进行累积操作，依次将前两个元素传递给指定函数，得到结果后，再与下一个元素继续进行累积，直到处理完所有元素，返回最终结果。在 Python 3 中，reduce 函数位于 functools 模块中，需要先导入。其语法为：

```
from functools import reduce
reduce(function, iterable[, initializer])
```

- function：用于累积操作的函数，接受两个参数。
- iterable：可迭代对象。
- initializer（可选）：初始值，若提供，则首先与可迭代对象的第一个元素进行累积。

```python
from functools import reduce

# 计算列表元素的累积乘积
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 输出: 120
# 初始值为 2
product = reduce(lambda x, y: x * y, numbers,2)
print(product) # 输出: 240
```

在此示例中，`reduce` 函数将匿名函数 `lambda x, y: x * y` 依次应用于列表 `numbers` 的元素，计算其累积乘积。最终结果为 `120`。当初始值为 2 时，累积乘积为 240.

递归是一种通过重复调用自身来解决问题的方法，适用于具有自相似结构的问题，即可以分解为若干规模缩小的相似子问题。递归函数是一类通过调用自身来解决问题的函数，特别适用于以下几类问题：

- **数学归纳类问题**：像阶乘、斐波那契数列、幂运算等问题，具有明显的数学递归定义，因此使用递归实现代码简洁明了。
- **分治问题**：这类问题可以划分为若干小规模的同类型子问题，并通过递归解决后合并结果。例如，归并排序和快速排序都通过递归分解排序任务，分别解决子数组的排序问题。
- **树形或图形结构遍历**：树结构的遍历（如二叉树的前序、中序和后序遍历）天然适合递归。通过递归，函数可以直接处理每个节点并对其子节点继续递归调用。图结构的遍历如深度优先搜索（DFS）也可通过递归实现。
- **回溯问题**：对于路径选择和解空间搜索问题，递归是实现回溯算法的自然方式。例如，解决数独问题、生成排列组合以及八皇后问题等，递归函数能更清晰地表达解空间的遍历过程。

递归函数的基本结构通常如下所示：

```
def recursive_function(parameters):
    if base_case_condition:  # 基线条件
        return base_case_value  # 返回基线值
    else:
        return recursive_function(modified_parameters)  # 递归调用
```

为避免无限循环和栈溢出，递归函数必须定义基线条件和递归条件。

- **基线条件：**基线条件是递归的终止条件，当问题规模足够小且可直接解决时，递归停止并返回结果。没有基线条件，递归会一直进行，导致栈溢出。
- **递归条件：**递归条件指函数在某些情况下调用自身，解决更小规模的子问题，直到达到基线条件。

示例 1：计算阶乘

阶乘是一个经典的递归问题，定义为：

- $0! = 1$
- $n! = n \times (n-1)!$，其中 $n > 0$

递归实现阶乘函数的代码如下：

```python
def factorial(n):
    if n <= 1:  # 基线条件
        return 1
    else:  # 递归条件
        return n * factorial(n - 1)
```

在这个实现中：

- **基线条件**是当 $n \leq 1$ 时，返回 1。
- **递归条件**是返回 $n \times factorial(n-1)$，每次将问题规模减小 1。

示例 2：斐波那契数列

斐波那契数列是另一个经典的递归问题，其定义为：

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$，其中 $n > 1$

递归实现斐波那契数列的代码如下：

```
def fibonacci(n):
    if n <= 0:  # 基线条件
        return 0
    elif n == 1:  # 基线条件
        return 1
    else:  # 递归条件
        return fibonacci(n - 1) + fibonacci(n - 2)
```

- **基线条件**分别是当 $n = 0$ 时返回 0，当 $n = 1$ 时返回 1。
- **递归条件**是返回 $fibonacci(n-1) + fibonacci(n-2)$，通过递归逐步减小问题规模。

使用递归函数的注意事项

1. **基线条件（终止条件）**：确保函数定义了正确的基线条件，避免无限递归。基线条件是递归终止的必要条件，缺少基线条件可能导致栈溢出。
2. **递归深度和性能**：递归会占用调用栈，每次递归调用都将使用额外的栈空间。因此，递归深度过大会导致栈溢出。Python 默认的递归深度是有限的，深度递归会触发 `RecursionError`。可以考虑通过迭代、缓存或动态规划来替代深度递归。
3. **重复计算和效率优化**：某些递归算法可能导致大量重复计算，例如在计算斐波那契数列时，可对已计算的结果进行缓存（如通过 `functools.lru_cache` 实现）来提升效率。这种优化技术称为记忆化（Memoization），能显著降低时间复杂度。
4. **可读性和平衡性**：在某些情况下，递归函数虽然简洁，但若过度依赖递归，可能会降低代码的可读性。编写时应权衡递归和迭代，选择更直观、效率更高的实现方式。

THE END