第二章 高阶函数

考虑问题 ① 内存（开销、释放） ② 效率 ③ 易读

2.1 函数参数

2.1.1 函数概要

简介 函数是编程中的基本模块之一，能够将重复使用的代码块打包并通过函数名调用，简化代码的重复劳动，提高可读性和可维护性。在 Python 中，函数是使用 `def` 关键字来定义的。函数的核心概念包括**输入参数、函数体、返回值**等。

2.1.2 函数参数

概要 函数可以接受多个参数，这些参数可以是**位置参数、默认参数、关键字参数或者可变参数**。

位置参数 最常见的形式，按照位置进行传递

默认参数 可以为参数提供默认值，调用时可以省略该参数

可变参数 ① ***args** 用于传递可变数量的**位置参数**

② ****kwargs** 用于传递可变数量的**关键字参数**

```python
def info(*args,**kwargs):
    print("位置参数",args)
    print("关键字参数",kwargs)
info(1,2,3,4,5,6,name="zhangsan",age=18,sex="man")
```

输出注意

1. 位置参数实际上**传入元组**，可以按照顺序读取 `(1, 2, 3, 4, 5, 6)`
2. 关键字参数实际上**传入的是字典** `{'name': 'zhangsan', 'age': 18, 'sex': 'man'}`
3. 推荐使用关键字参数传递，输入参数不要写死。比如画图时，`plt(x,y,color='black',lw=3,...)`关键字可以放在任何位置，这种传递方式能让程序健壮，可读性高

返回值 函数可以返回一个值或多个值(元组形式返回)，如果不使用 `return`，默认返回 `None`

函数不定义数据类型，编写时需要明确数据类型（整形、双精度等），比如注意浮点相减两者差值小于一个足够小的数。

2.2 匿名函数 Lambda

概述 可以使用 `lambda` 关键字定义匿名函数。常用于需要简单函数排序、过滤等。

用于创建**简短的、临时的小函数**，无需使用 def 定义。常用于需要函数对象但不希望命名的场景，比如作为高阶函数的参数。

语法 `lambda` **参数 1,参数 2,……：表达式**（即返回结果）

参数可以有多个，用逗号分隔，表达式部分计算结果即为返回值。

特点 代码行数必须限制在单行内，适用于简单操作。

实例

```python
#两个数的求和
add = lambda x,y:x+y
print(add(1,2))
```

```python
#列表排序
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
print(pairs)    #输出：[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

作用域　函数中定义的变量为**局部变量**。如果想在函数中修改全局变量，可以用 `global` 定义参数为全局变量
建议少用 global 进行全局传递。

```
#全局变量
global_var = 1
def modify_var():
    global global_var    #设定为全局变量，可以监测执行全过程中参数变化
    global_var = 2
    print("修改后的全局变量值为：", global_var)
```

2.3 map 函数

用途　map 函数用于将指定的函数**依次作用于一个或多个可迭代对象**（如列表、元组等）的每个元素，并返回一个**迭代器**，**效率比 for 循环高很多**
例如，画图时横轴数字后增加特殊尾缀，可以直接使用 map 增加。

语法　`map(function,iterable,…)`
`function`　**应用到每个元素的函数**。如果提供多个可迭代对象，则函数必须接收相应数量的参数。
`iterable`　一个或多个可迭代对象。

返回值　其返回结果是 **map 对象**，可以在前增加 `list()`将类型转换为列表

实例　① 使用 map 将列表中的**元素平方**

```
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))    #map 用于遍历列表，是个迭代器，效率比 for 循环高
print(squares)  #输出：[1, 4, 9, 16, 25]
#另一种实现方式
def square(x):
    return x ** 2
nums=[1,2,3,4,5]
squares = list(map(square,nums))
```

② 画图时横轴数字后增加尾缀，可以直接使用 map 增加

```
x = [1, 2, 3, 4, 5]
x_with_suffix = list(map(lambda x: str(x) + "st", x))
```

2.4 filter 函数

用途　用于**过滤可迭代对象中的元素**。它接收一个函数作为条件，该函数**返回 True 的元素会被保留下来**，返回 False 的元素将被过滤掉。

语法　`filter(function,iterable,…)`
`function`　判断条件函数，返回 True 或 False。若为 None，则直接判断元素的真假值。
`iterable`　一个或多个可迭代对象。

返回值　其返回结果是 **filter 对象**，可以在前增加 `list()`将类型转换为列表

实例　① 使用 `filter` **过滤出偶数**

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  #输出：[2, 4]
```

```
#另一种实现方式
def is_even(n):
    return n % 2 == 0
nums=[1,2,3,4,5]
even_nums=filter(is_even,nums)
```

2.5 reduce 函数

用途　用于**对可迭代对象中的所有元素进行累积计算**，将其归约为一个单一的值。常用于求和、求积等操作

语法　`reduce(function,iterable[,initializer])`

`function`　接收两个参数的函数，返回值会作为下一次调用的第一个参数。

`iterable`　一个可迭代对象。

`initializer`　（可选）初始值，如果提供，则第一次调用时会将该值作为第一个参数传入。

注意　在 Python 3 中，`reduce` 函数需要从 `functools` 模块中导入。

实例　① 计算列表元素之乘积

```python
from functools import reduce
nums = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, nums)      # 输出：120
```

2.6 递归函数

定义　在定义中调用自身的函数，常用于解决分治问题或自然分级问题，如求阶乘。

```python
#递归函数，计算阶乘
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5)) #输出：120
```

```python
#拷贝函数，将 A 复制到 B，A 如果是个目录，则反复调用
import os
def copy_directory(src, dst):
    for item in os.listdir(src):
        src_item = os.path.join(src, item)
        dst_item = os.path.join(dst, item)
        if os.path.isdir(src_item):
            os.makedirs(dst_item, exist_ok=True)
            copy_directory(src_item, dst_item)
        else:
            shutil.copy2(src_item, dst_item)
```

2.7 装饰器 Decorator

2.7.1 装饰器简介

用途　用于**扩展函数功能的高级特性**。它可以在**不改变原函数的情况下**增加新的功能。

通常用于日志记录、性能测试、权限验证等场景。

案例　例如，读取文件有多个函数：对 html、csv、二进制、grib 等文件分别的函数，可以直接使用装饰器增加检验文件是否存在的功能；把读取功能和判断功能(外围功能)执行分离，便于阅读。

此外，绘制各类图时，轴须、颜色映射等功能可以写在装饰器中，只需更改绘图的核心部分。

2.7.2 写法说明

定义　**函数声明**：定义一个名为 `file_exists` 的函数，该函数**接收一个参数** `func`，即将要被装饰的**目标函数**

包装函数：在 `file_exists` 内部定义了一个名为 `wrapper` 的函数。这个包装函数**接收与目标函数相同的参数**（这里至少包含 `filepath` 以及可变参数`*args`, `**kwargs`）。

保留元数据　`functools.wraps`　使用`@functools.wraps(func)`装饰 `wrapper` 函数。其作用是**将目标函数 `func` 的元数据**（例如函数名称、文档字符串等）**复制到 `wrapper` 中**。这样，即使函数经过装饰后，依然能够保留原有的信息，有助于调试和文档生成。

增加功能　**功能增强**：在 `wrapper` 函数内，首先检查传入的 `filepath` 指定的文件是否存在。如果文件不存在，则打印错误信息并返回 None，避免进一步执行目标函数。

调用原函数：如果文件存在，则调用原函数 `func`，并将所有参数传递过去，返回原函数的执行结果。

装饰器应用　**语法糖**：使用`@file_exists` 语法将 `read_file` 函数进行装饰。

等价于执行 read_file = file_exists(read_file)。这样，每次调用 read_file 时，实际执行的是 wrapper 函数，从而在执行原函数前加入了文件存在性的检查。

```python
import os
import functools
#装饰器
def file_exists(func):
    @functools.wraps(func)  #原函数信息保留
    def wrapper(filepath, *args, **kwargs):  #装饰器一般用
        if not os.path.exists(filepath):
            raise FileNotFoundError(f"File '{filepath}' does not exist.")
        return func(filepath,*args, **kwargs)   #如果存在，仍然执行原函数
    return wrapper
```

```python
# 增加装饰器，没有改变下方函数的结构，却拓展了功能
@file_exists
def read_file(file_path):
    with open(file_path, 'r') as file:
        content = file.read()
    return content
```

实例

装饰器用于记录函数的调用，优化程序时程序效率的监控

```python
import time
import functools
def timing_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"{func.__name__} took {execution_time:.4f} seconds to execute.")
        return result
    return wrapper
```

```python
@timing_decorator
def my_function():
    time.sleep(2)
my_function()
```

2.8 其他高级用法

2.5.1 闭包 closure

用途

在**一个函数内部定义另一个函数**，这个函数不仅可以使用自身的局部变量，还能捕获并使用其外部函数作用域中的变量，即使外部函数已经执行完毕，这种机制使得内部函数拥有记忆功能，可以访问并保留外部状态。

实例

```python
def greet_creator(greeting):
    # greeting 是外部函数的变量
    def greet(name):
        # 内部函数使用了外部变量 greeting
        return f"{greeting}, {name}!"
    return greet
```

```python
# 使用闭包创建一个问候函数
say_hello = greet_creator("Hello")
print(say_hello("Alice"))  # 输出: Hello, Alice!

# 也可以创建其它不同问候方式的函数
say_hi = greet_creator("Hi")
print(say_hi("Bob"))  # 输出: Hi, Bob!
```

2.5.2 函数注解

用途

用于**说明参数和返回值的类型**。这些注解不会影响代码的执行，但可以用于文档生成和类型检查工具。

说明

① **参数注解**：a: int 和 b: int 表示参数 a 和 b 被建议应为 int 类型

② **返回值注解**：-> int 表示函数建议返回值类型为 int。

实例

```python
def add(a: int, b: int) -> int: #不显式指定每个数据类型， Python 不会根据注解自动调整类型
    return a + b
print(add(3,5))
```