第二章 函数、类与数据

Lianghai Xiao

https://github.com/styluck/mlb

作业邮箱：alswfx@126.com

创建一个简单函数

- def my_function(a, b, c):
- print(a, b, c)

- def my_function(name, age, city):
- print(f"{name} is {age} years old and lives in {city}.")

利用生成器定义函数

- def my_generator():
- 	yield 1
- 	yield 2
- 	yield 3
- gen = my_generator()

- for value in gen:
- 	print(value)

*arg命令

- def my_function(*args):
- for arg in args:
- print(arg)
- # 可以将任意数量的参数传递给 *args，这里被打包成一个元组
- 。my_function(1, 2, 3)
- my_function("a", "b", "c")

反向打包

- def my_function(a, b, c):
- print(a, b, c)

- values = (1, 2, 3)
- my_function(*values) # 解包元组

反向打包

- def my_function(name, age, city):
- print(f"{name} is {age} years old and lives in {city}.")

- info = {"name": "Alice", "age": 25, "city": "New York"}
- my_function(**info) #解包字典

*args与**kwargs

- ***args** 将额外的**位置**参数收集为**元组**。
- ****kwargs** 将额外的**关键字**参数收集为**字典**。

- def my_function(*args, **kwargs):
- print("Positional arguments:", args)
- print("Keyword arguments:", kwargs)

- my_function(1, 2, 3, name="Alice", age=25)

*args与**kwargs

```
def describe_pet(animal_type, pet_name, *args, **kwargs):
    print(f"\nI have a {animal_type} named {pet_name}.")
    if args:
        print("Additional info:")
        for info in args:
            print(f"- {info}")
    if kwargs:
        print("Extra details:")
        for key, value in kwargs.items():
            print(f"{key}: {value}")
```

*args与**kwargs

- # 使用混合参数调用函数
- describe_pet(
- "dog",
- "Buddy",
- "Loves to play fetch",
- "Very friendly",
- age=5,
- color="brown"
-)

*args与**kwargs

- def make_pizza(size, *toppings, crust="regular"):
- print(f"\nMaking a {size}-inch pizza with {crust} crust.")
- if toppings:
- print("Toppings:")
- for topping in toppings:
- print(f"- {topping}")

- # Calling the function
- make_pizza(12, "pepperoni", "mushrooms", crust="thin")

装饰器函数

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something before the function")

        result = func(*args, **kwargs)

        print("Something after the function")

        return result
    return wrapper
```

创建一个类(class)

```
class student:
 def __init__(self, name, age, **kwargs):
        self.name = name
        self.age = age
        for key, value in kwargs.items():
            setattr(self, key, value)

    def __call__(self):
        print(f"My name is {self.name}.")

def show_age(self):
        print(f"{self.name} is {self.age} years old.")
```

类的用法

- Peter = student(‘peter’, 21)
- Peter
- Peter.__str__()
- Peter()
- Peter.__dict__
- Peter.show_age()

上下文管理器

- 上下文管理器定义了在一段代码块开始前和结束后要做的事（进入/清理）。
- 典型用途：资源管理（文件/网络连接/锁/事务等），保证异常也能正确清理，相当于把 try/finally 写进了对象内部。
- with open("file.txt", "r") as file:
- content = file.read()

- with open("example.txt", "w") as file:
- file.write("Hello, World!")

模块化

- 把代码按职责拆分为**可复用、低耦合**的单元（模块/包），每个单元只关注一类功能。
- 复用、测试方便、多人协作清晰、替换升级容易、降低复杂度。

- 模块（module）：单个 .py 文件。
- 包（package）：含 __init__.py 的目录。

__init__.py

- 标记目录为普通包。
- 执行一次：首次 import mypkg 时会运行其中代码。
- 定义包的对外 API（重导出子模块/对象），以及包级元数据（如 __version__）。
- 可做少量初始化（注册插件、轻量校验等），但要避免副作用和沉重依赖的即时导入。

__init__.py

- # mypkg/__init__.py
- """mypkg: 简要说明（1~2 行）"""

- # 版本号（打包后可用 importlib.metadata 读取）
- __version__ = "0.1.0"

- # 公开 API（从子模块重导出）
- from .core import Model, train
- from .io import load_data, save_data

- # 控制 from mypkg import * 的导出名单
- __all__ = ["Model", "train", "load_data", "save_data", "__version__"]

模块化

- if __name__=='__main__':
- 编写 Python 脚本时，我们希望某些代码只在该脚本作为主程序运行时执行，而不是在被其他模块导入时执行。上述语句就可以用来确定当前模块是否是主程序入口点。
- 在开发过程中，可以在模块内部编写一些测试代码或者示例代码，并将它们放在if __name__=='__main__'条件块内。这样，当你运行该模块作为主程序时，这些测试代码将被执行，而当该模块被其他模块导入时，这些测试代码不会干扰其他模块的正常运行。