Python程序设计实践

主讲：张琦

面向对象编程与高级数据结构

1. 面向对象编程
2. 高级数据结构
3. 正则表达式与网络编程
4. 数据库编程
5. 多线程与多进程编程

3.1 面向对象编程

- 类与对象的定义与使用
- 类的属性与方法
- 继承与多态
- 实践指导：编写面向对象的程序，练习类的定义与使用

类与对象的定义和使用

- **面向对象编程（Object-Oriented Programming，OPP）是一种编程范式，它使用“对象”来设计软件。**
  - **类（Class）**：定义了一组具有相同属性（数据元素）和行为（功能）的对象的蓝图或模板。类是创建对象的模板。
  - **对象（Object）**：类的实例。对象是根据类定义创建的实体，包含了类中定义的属性和方法。
  - **属性（Attribute）**：对象的数据部分，代表对象的状态或特征。在类中定义，每个对象都拥有这些属性的具体值。
  - **方法（Method）**：对象的行为部分，定义在类中的函数。方法可以访问和修改对象的属性。
  - **继承（Inheritance）**：允许一个类继承另一个类的属性和方法，使得代码复用成为可能。子类继承父类的特性，同时可以有自己的特性。
  - **多态（Polymorphism）**：指不同类的对象对同一消息的响应方式不同。通过继承和方法重写（Overriding）实现多态性，使得同一个接口可以有不同的实现。
  - **封装（Encapsulation）**：将对象的实现细节隐藏起来，只暴露有限的接口与外界交互。封装使得对象的内部表示对外部是不可见的，只能通过对象提供的方法来访问。

如何定义一个类

```python
# 定义一个名为Person的类
class Person:
    # 初始化方法，定义了两个属性：name和age
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # 一个方法，用于打印一个人的信息
    def print_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# 创建一个Person类的实例
person1 = Person("Alice", 30)

# 调用实例的方法
person1.print_info()
```

用代码展示继承、多态和封装

```python
# 定义一个基类 Animal
class Animal:
    def __init__(self, name):
        self.name = name  # 封装：将名称属性隐藏在类内部

    def speak(self):  # 多态：基类提供接口，具体实现由子类决定
        raise NotImplementedError("Subclass must implement abstract method")

# 定义两个派生类 Dog 和 Cat，它们继承自 Animal 类
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# 创建 Dog 和 Cat 的实例
dog = Dog("Buddy")
cat = Cat("Whiskers")

# 调用 speak 方法，展示多态
print(dog.speak())  # Buddy says Woof!
print(cat.speak())  # Whiskers says Meow!

# 这个例子展示了：
# 继承：Dog 和 Cat 类继承自 Animal 类
# 多态：Dog 和 Cat 类重写了 Animal 类的 speak 方法，实现了多态
# 封装：Animal 类封装了 name 属性，只能通过类的方法访问
```

实践指导

- **3.1 编写面向对象的程序，练习类的定义与使用**
  - 练习实践指导3中的3个习题，并且完成下面的作业。

任务描述：

设计一个简单的图书管理系统，要求如下：

1. **图书类（Book）**：
   - 属性：书名（title）、作者（author）、ISBN号（isbn）、价格（price）
   - 方法：
     - `__init__`：初始化图书信息
     - `display_info`：打印图书的详细信息
2. **图书馆类（Library）**：
   - 属性：图书集合（一个包含多本图书的列表）
   - 方法：
     - `__init__`：初始化图书馆，开始时图书集合为空
     - `add_book`：向图书馆添加一本图书
     - `remove_book_by_isbn`：通过ISBN号移除一本图书
     - `display_all_books`：打印图书馆中所有图书的信息

要求：

- 使用面向对象的方式实现上述需求。
- 对图书进行添加、移除操作后，能够正确显示图书馆中的图书信息。

附加挑战（可选）：

- 实现一个搜索功能，通过书名或作者搜索图书，并显示相关图书的信息。
- 对图书按价格进行排序并显示。

提示：

- 可以使用列表来存储图书馆中的图书对象。
- 在`Library`类中，遍历图书列表来实现显示和移除图书的功能。

3.2 高级数据结构

- 列表推导式与生成器
- 迭代器与装饰器
- 匿名函数与高阶函数
- 实践指导

- **列表推导式**

  - **定义**　列表推导式（List Comprehension）是Python中一种简洁而优雅的创建列表的方式。它通过在一个表达式中嵌入循环和条件来生成列表。

  - **语法**

```
[表达式 for 元素 in 可迭代对象 if 条件]
```

  - **示例**　创建一个包含1到10之间所有偶数的列表

```
even_numbers = [x for x in range(1, 11) if x % 2 == 0]
print(even_numbers)  # 输出: [2, 4, 6, 8, 10]
```

将一个字符串列表中的所有字符串转换为大写：

```
words = ["hello", "world", "python"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words)  # 输出: ['HELLO', 'WORLD', 'PYTHON']
```

- **列表推导式**
  - **嵌套列表推导式**　创建一个包含1到3之间所有组合的二维列表：

```
combinations = [(x, y) for x in range(1, 4) for y in range(1, 4)]
print(combinations)
# 输出: [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

- **生成式**

  - **定义**　生成式（Generator Expression）类似于列表推导式，但它不是一次性生成整个列表，而是返回一个生成器对象，按需生成元素。生成器在需要时才生成值，因此在处理大量数据时更高效。

  - **语法**

```
（表达式 for 元素 in 可迭代对象 if 条件）
```

  - **示例**　创建一个生成器对象，生成1到10之间所有偶数：

```
even_numbers_gen = (x for x in range(1, 11) if x % 2 == 0)

for num in even_numbers_gen:
    print(num)  # 输出: 2 4 6 8 10
```

- **示例**　将一个字符串列表中的所有字符串转换为大写生成器：

```
words = ["hello", "world", "python"]
uppercase_words_gen = (word.upper() for word in words)

for word in uppercase_words_gen:
    print(word)  # 输出: HELLO WORLD PYTHON
```

- **示例**　生成无限序列：

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_gen = fibonacci()

for _ in range(10):
    print(next(fib_gen))  # 输出前10个斐波那契数: 0 1 1 2 3 5 8 13 21 34
```

- **总结**
  - 列表推导式用于生成列表，语法简洁，适合处理小数据集。
  - 生成式用于生成生成器对象，按需生成数据，适合处理大数据集。
  - 列表推导式和生成式都可以嵌套使用，以实现复杂的数据生成逻辑。

通过掌握列表推导式和生成式，Python程序员可以编写出更加简洁、高效的代码，尤其是在数据处理和分析领域。

- **迭代器和生成器**
  - **定义**　迭代器（Iterator）是一个可以记住遍历位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。
  - **迭代器的基本方法**
    - **`__iter__()`**：返回迭代器对象本身。
    - **`__next__()`**：返回容器的下一个元素，如果没有元素则抛出 `StopIteration` 异常。
  - **示例**　创建一个简单的迭代器

```python
class MyIterator:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            result = self.data[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration

# 使用迭代器
my_iter = MyIterator([1, 2, 3, 4])
for item in my_iter:
    print(item)  # 输出: 1 2 3 4
```

- **内置迭代器**　Python内置了许多迭代器，例如 `range`、`map`、`filter` 等。

```python
# range 迭代器
for i in range(5):
    print(i)  # 输出: 0 1 2 3 4

# map 迭代器
squared = map(lambda x: x*x, [1, 2, 3, 4])
for num in squared:
    print(num)  # 输出: 1 4 9 16

# filter 迭代器
even_numbers = filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5])
for num in even_numbers:
    print(num)  # 输出: 2 4
```

- **生成器**　生成器是创建迭代器的一种简便方式。使用 yield 关键字可以创建生成器。

```
# 定义一个简单的生成器函数，生成0到n的平方
def square_numbers(n):
    for i in range(n):
        yield i ** 2

# 使用生成器
for square in square_numbers(5):
    print(square)
```

这个例子中， square_numbers是一个生成器函数，它逐个产生0到n-1的平方。使用yield语句可以将一个函数转换为生成器，这样它就会产生一个序列的值，而不是一次性返回所有值。这对于处理大量数据时节省内存非常有用。

- **装饰器**

  - **定义**　装饰器（Decorator）是一个用于修改函数或方法行为的高级函数。装饰器允许在不修改原函数代码的情况下，向函数添加功能。

  - **语法**　通过在函数定义前使用@decorator_name语法来应用装饰器

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# 输出:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

- **装饰器**

  - **带参数的装饰器**　装饰器可以接受参数，通过嵌套函数实现。

```python
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                func(*args, **kwargs)
        return wrapper
    return decorator_repeat

@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
# 输出:
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!
```

- **装饰器**
  - **装饰器的实际应用**
    - **日志记录：**记录函数调用的日志。
    - **性能计时：**计算函数执行时间。
    - **访问控制：**检查用户权限。
    - **示例：性能计时装饰器**

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function {func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    print("Function complete")

slow_function()
# 输出:
# Function complete
# Function slow_function took 2.0001 seconds
```

- **总结**
  - **迭代器**：用于遍历集合中的元素，提供 **`__iter__()`** 和 **`__next__()`** 方法。
  - **生成器**：一种创建迭代器的简便方式，使用 `yield` 关键字。
  - **装饰器**：用于修改函数或方法行为的高级函数，允许在不修改原函数代码的情况下，向函数添加功能。

通过掌握迭代器和装饰器，Python程序员可以编写更简洁、高效和可复用的代码，尤其是在数据处理和函数扩展方面。

- **匿名函数**

  - **定义**　匿名函数（Anonymous Function）是没有名字的函数。在Python中，匿名函数使用 `lambda` 关键字定义。由于匿名函数通常只用于简单的操作，因此它们通常只包含一个表达式。

  - **语法**

```
lambda 参数列表: 表达式
```

  - **示例**　定义一个简单的匿名函数，计算两个数的和：

```
add = lambda x, y: x + y
print(add(2, 3))  # 输出: 5
```

使用匿名函数对列表进行排序：

```
points = [(2, 3), (1, 2), (4, 1)]
points.sort(key=lambda point: point[1])
print(points)  # 输出: [(4, 1), (1, 2), (2, 3)]
```

- **匿名函数**

  - **匿名函数的应用**　匿名函数通常用于需要一个简单函数的场景，例如作为高阶函数的参数。

```python
# 使用匿名函数与map
squared = map(lambda x: x * x, [1, 2, 3, 4])
print(list(squared))  # 输出: [1, 4, 9, 16]

# 使用匿名函数与filter
even_numbers = filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
print(list(even_numbers))  # 输出: [2, 4]
```

- **高阶函数**

  - **定义**　高阶函数（Higher-Order Function）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。高阶函数的主要特点是能够操作其他函数。

  - **常见的高阶函数**

    - `map(function, iterable)`：对可迭代对象的每个元素应用函数，并返回一个迭代器。

    - `Filter(function, iterable)`：对可迭代对象的每个元素应用函数，并返回一个过滤后的迭代器

    - reduce(function, iterable)：对可迭代对象的元素进行累积计算。

- **高阶函数**

  - **示例** 使用 map高阶函数将列表中的每个元素平方：

```python
numbers = [1, 2, 3, 4]
squared = map(lambda x: x * x, numbers)
print(list(squared))  # 输出: [1, 4, 9, 16]
```

使用 filter高阶函数过滤列表中的偶数：

```python
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]
```

使用 reduce高阶函数计算列表中元素的累积和：

```python
from functools import reduce

numbers = [1, 2, 3, 4]
sum = reduce(lambda x, y: x + y, numbers)
print(sum)  # 输出: 10
```

- **高阶函数**

  - **高阶函数的应用** 高阶函数在函数式编程中广泛应用，能够提高代码的简洁性和可读性。

```python
# 定义一个高阶函数，接受另一个函数作为参数
def apply_function(func, value):
    return func(value)

# 使用高阶函数
result = apply_function(lambda x: x * x, 5)
print(result)  # 输出: 25
```

  - **函数作为返回值** 高阶函数可以返回一个函数：

```python
def multiplier(factor):
    def multiply_by_factor(number):
        return number * factor
    return multiply_by_factor

# 使用高阶函数返回的函数
double = multiplier(2)
print(double(5))  # 输出: 10

triple = multiplier(3)
print(triple(5))  # 输出: 15
```

- **总结**
  - **匿名函数**：使用 `lambda` 关键字定义的没有名字的函数，通常用于简单的操作。
  - **高阶函数**：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，常用于函数式编程。

通过掌握匿名函数和高阶函数，Python程序员可以编写更加简洁和高效的代码，尤其是在数据处理和函数操作方面。

实践指导

- **3.2 编写高级数据结构的练习题，使用生成器和装饰器优化代码**
  - 练习实践指导中的5个习题