Python 的面向对象编程（OOP）是一种编程范式，它使用 **类（Class）** 和 **对象（Object）** 来组织代码，使其更易于管理、扩展和复用。OOP 主要包括以下几个概念：

- **类（Class）**：定义对象的模板
- **对象（Object）**：类的实例
- **属性（Attribute）**：对象的数据
- **方法（Method）**：对象的行为
- **封装（Encapsulation）**：隐藏对象的内部实现
- **继承（Inheritance）**：子类继承父类的属性和方法
- **多态（Polymorphism）**：不同类的对象可以使用相同的方法

1. 定义类和创建对象

```
class Person:
    def __init__(self, name, age):  # 构造函数
        self.name = name  # 属性
        self.age = age

    def greet(self):  # 方法
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# 创建对象
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

# 调用方法
p1.greet()
p2.greet()
```

2. 封装（Encapsulation）

封装是指隐藏对象的内部实现，使用私有变量（ __ 开头）限制访问。

```python
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # 私有属性

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}, new balance: {self.__balance}")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrawn {amount}, new balance: {self.__balance}")
        else:
            print("Insufficient funds!")

    def get_balance(self):  # 提供访问私有属性的方法
        return self.__balance

# 创建账户
account = BankAccount("Alice", 1000)
account.deposit(500)
account.withdraw(300)
print(account.get_balance())

# print(account.__balance)  # 这行会报错，因为 __balance 是私有的
```

3. 继承（Inheritance）

子类可以继承父类的属性和方法，并进行扩展。

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass  # 让子类实现

class Dog(Animal):
    def make_sound(self):
        print(f"{self.name} says: Woof!")

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

# 创建对象
dog = Dog("Buddy")
cat = Cat("Kitty")

dog.make_sound()
cat.make_sound()
```

4. 多态（Polymorphism）

不同的类可以使用相同的方法，达到灵活性。

```python
def animal_sound(animal):
    animal.make_sound()

animals = [Dog("Rex"), Cat("Mimi")]

for animal in animals:
    animal_sound(animal)  # 调用各自的方法
```

5. 类方法、静态方法

- **实例方法（Instance Method）**：默认第一个参数 `self`
- **类方法（Class Method）**：使用 `@classmethod`，第一个参数是 `cls`
- **静态方法（Static Method）**：使用 `@staticmethod`，不依赖 `self` 或 `cls`

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

    @classmethod
    def description(cls):
        return "This is a math utility class."

print(MathUtils.add(5, 10))
print(MathUtils.description())
```

6. Python 的特殊方法（魔法方法）

魔法方法是以 `__`（双下划线）包围的特殊方法，比如 `__init__`、`__str__`、`__len__` 等。

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} by {self.author}, {self.pages} pages"

    def __len__(self):
        return self.pages

book = Book("Python Programming", "John Doe", 350)
print(book)  # 调用 __str__
print(len(book))  # 调用 __len__
```

7. 组合（Composition）

组合是一种**比继承更灵活的方式**，可以在类中包含其他类的对象。

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self, brand):
        self.brand = brand
        self.engine = Engine()  # 组合

    def start(self):
        print(f"{self.brand} car is starting...")
        self.engine.start()

car = Car("Toyota")
car.start()
```

练习题：

1. 定义一个 `Student` 类，包括 `name`、`age` 和 `grades`（列表），并实现：
    - `add_grade(grade)` 方法添加成绩
    - `average_grade()` 计算平均分
    - `__str__()` 方法，打印学生信息
2. 创建 `Employee` 类，继承 `Person`，增加 `salary` 和 `work()` 方法。

你可以试着写代码，如果有问题，我可以帮你改进 😊