Python程序设计与实践

第九课：类与面向对象编程

2025. 5

Today

- 掌握类、对象与OOP核心机制，实现模块化与可扩展编程
- 理解类的构建方式、对象的使用方法，以及OOP的基本思想
- 理解 Dunder 方法（双下划线方法）的本质与用法
- 熟练设计复合对象类、重载常见运算符、进行类型检查

现实世界中的对象及其属性与方法

- 我们生活中充满了“对象”：如“电梯” “员工” “订单” “顾客队列”等
- 每个对象都有状态（属性）和行为（方法）：
  - 电梯：状态（所在楼层、是否有人），行为（上升、下降）
  - 员工：状态（姓名、工资），行为（出勤、交报告）
  - 队列：状态（等待的人列表），行为（入队、出队）
- 如果我们希望在程序中模拟这些实体并实现其逻辑，应该使用什么方式？
  - 这正是“类”的作用
  - 定义“类”就是在针对真实场景建模

什么是对象（Object）

- 类是对一组具有相同属性和方法的对象的抽象描述，而对象是类的具体实例
- 在Python中，“一切皆对象”
- 基本类型如整数、浮点数、字符串、列表、字典等，本质上都是某个类的对象实例
- 每个对象都具有：
  - 内部数据结构（由属性表示，如整数的二进制表示、列表的链式结构）
  - 一组方法，用于与对象交互（如.append()、.upper()）
- 对象不仅仅是数据，还包含对这些数据的操作方式，是数据+行为的封装体

```
x = 3          # int对象，支持加减乘除等操作
s = "hi"       # str对象，支持拼接、切片、查找等
L = [1, 2]     # list对象，支持增删查改
```

■ 面向对象编程（OOP）

- OOP 是一种编程范式，它将程序组织为“对象”的集合，封装状态和行为
- Python 是支持 OOP 的语言，提供关键机制：
  - 类（class）、对象（object）、封装（encapsulation）、继承（inheritance）、多态（polymorphism）
- OOP的优势：
  - 提升程序结构清晰度
  - 支持模块化与复用
  - 降低耦合度，便于维护与扩展

对象与类的基本概念

- 类（class）是一个用户定义的数据类型，是对象的蓝图
- 对象（object）是类的实例，是在程序中实际存在的实体
- 类中可以定义数据属性（存储状态）、方法（定义行为）
- 类是抽象的，只有通过实例化后对象才具有实际意义
- Python具有自动内存管理机制，对象无需手动释放，垃圾回收器（GC）会处理

类的设计动机与生活类比

- 编写大型程序时，数据与操作往往是紧密关联的
- 类的设计将数据与操作方法打包成一个整洁的单元
- 类 = 模板；对象 = 实例
  - 图纸是“类”，具体建成的房子是“对象”
  - 人的DNA是“类”，我们每个人是“对象”
- 使用类可以反复创建多个“功能相似但状态不同”的对象

用Python定义类——class关键字

- Python中使用 class 关键字定义类

```
class Coordinate:
    pass  # 暂不定义内容
```

- 语法说明：
  - 类名采用驼峰式命名法（如 MyClass、StudentRecord）
  - ： 号表示类体开始，类体内部通过缩进定义属性与方法
- 类定义本身不会创建任何对象，只有调用类名时才会实例化

```
c = Coordinate() # 创建对象，遵循Coordinate类的模板
```

■ 创建类的实例（Instantiation）

- 实例化 = 使用类创建对象，即“类名 + 括号参数”调用构造方法

```
# 由同一个类创建两个不同的实例
c1 = Coordinate(3, 4)
c2 = Coordinate(0, 0)
```

- 每个实例会独立存储一份属性副本

```
print(c1.x)   # 输出 3
print(c2.x)   # 输出 0
```

- 实例化时Python自动调用 `__init__` 构造方法，不需要显式传入 `self`
  - 定义class时必须含有__init__方法
  - 方法的参数在创建对象时给出具体值

■ 方法（Method）定义与使用

- 方法是类内部定义的函数，用于描述对象的行为
- 在类中定义方法时，第一个参数永远是 `self`，代表当前实例对象
- 示例：定义计算两点距离的方法

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        return ((self.x - other.x)**2 + (self.y - other.y)**2)**0.5
c1 = Coordinate(3, 4)
c2 = Coordinate(0, 0)
print(c1.distance(c2))  # 5.0 c1.distance(c2) → self = c1, other = c2
```

方法的调用机制详解

- 调用对象方法时，本质上是调用类中定义的函数，并自动将调用者作为第一个参数传入
- 以下两种方式是等价的

```
c1.distance(c2)
# 等价于
Coordinate.distance(c1, c2)
```

- Python内部机制：
  - 通过 . 查找 distance 方法
  - 自动绑定 self = c1，传入其他参数
  - 执行方法体

面向对象的三大特性

- Python支持OOP的三大核心特性
  - **封装（Encapsulation）**：数据+方法打包到对象中，隐藏内部实现
  - **继承（Inheritance）**：子类可以复用父类的代码并扩展
  - **多态（Polymorphism）**：相同方法名适用于不同对象，行为表现不同
  - 调用对象方法时，本质上是调用类中定义的函数，并自动将调用者作为第一个参数传入

默认封装方法：__str__

- 默认打印对象只显示内存地址：如 <__main__.Coordinate object at 0x...>
- 可通过重写 __str__ 方法定义自定义字符串表示

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})”
```

- 使用 print(c1) 时自动调用 __str__ 方法，输出更加直观

■ 类中的私有方法与公有接口

- 类中的方法也可以使用 _ 或 __ 作为前缀区分
  - _helper()：内部使用方法（受约定保护）
  - __secure_calc()：避免外部或子类直接访问

- 公有方法用于定义“接口”
- 私有方法和属性用于“实现细节”

```
class Bank:
    def __init__(self, balance):
        self._balance = balance

    def deposit(self, amt):
        self._balance += amt

    def _log(self):
        print(f"当前余额：{self._balance}")
```

封装原则与命名规范

- Python采用命名约定实现属性私有化（区分对外接口和对内实现，没有严格私有机制）：
  - _var：建议仅类内部访问
  - __var：类名重整机制（name mangling），防止子类访问

```
class BankAccount:
    def __init__(self, name, balance):
        self.name = name
        self._balance = balance  # 内部使用

    def deposit(self, amount): # 外部通过接口访问内部变量
        self._balance += amount

    def get_balance(self):
        return self._balance
```

实例：定义一个简单的学生类

- 目标：定义一个 Student 类，实现以下功能
  - 初始化学生姓名、成绩
  - 添加分数、计算平均成绩
  - 实现 __str__ 方法输出信息

```
class Student:
    def __init__(self, name):
        self.name = name
        self.scores = []

    def add_score(self, score):
        self.scores.append(score)

    def average(self):
        return sum(self.scores)/len(self.scores)

    def __str__(self):
        return f"{self.name}, 平均分:{self.average()}"
```

```
s = Student("小明")
s.add_score(85)
s.add_score(90)
print(s)
```

对象之间的组合（Composition）

- 类与类之间不仅可以继承，还可以组合，即一个类的属性是另一个类的实例
- 示例：每个学生对象包含一个地址对象

- 组合的优势：
  - 模块化：每个类负责一个职责
  - 可重用：Address类可用于多个实体

```
class Address:
    def __init__(self, city, street):
        self.city = city
        self.street = street

class Student:
    def __init__(self, name, address):
        self.name = name
        self.address = address

addr = Address("上海", "中山路99号")
s = Student("李雷", addr)
print(s.address.city)  # 输出：上海
```

■ 构造函数中的默认参数与可选项

- 构造方法 `__init__` 可使用默认参数简化对象创建过程

```
class Student:
    def __init__(self, name, scores=None):
        self.name = name
        self.scores = scores if scores else []

s1 = Student("小红")  # scores 默认为空列表
s2 = Student("小明", [90, 95])  # 初始化已给分数
```

- 不要用`[]`等可变对象作为默认值

  (创建对象时仅在创建别名，列表会被多个类的对象共享)

- 推荐写成 `scores=None`，再手动判断处理

类中的属性

- 类中的变量叫做“属性”，用于表示对象的状态
- 属性分为两种：
  - 实例属性：每个对象自己的属性（如每个点的坐标值，可在实例化时指定）
  - 类属性：所有对象共享的属性（如所有点都在二维空间，类定义时直接给出）
- 例子：为Coordinate类增加属性

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- self 代表当前对象本身，属性通过 self.属性名 赋值
- 每个对象有不同的x和y

■ 类属性与实例属性的区别

- 类属性：定义在类体中，所有实例共享
- 实例属性：定义在 `__init__` 或 `self.` 中，每个实例独有

- 修改类属性会影响所有实例
- 但实例属性互不干扰

```
class Dog:
    species = "Canine"  # 类属性

    def __init__(self, name):
        self.name = name  # 实例属性

d1 = Dog("Tommy")
d2 = Dog("Lucky")
print(d1.species, d2.species)  # Canine Canine
print(d1.name, d2.name)        # Tommy Lucky
```

■ 示例：动物类建模练习

- 目标：设计一个简单的 `Animal` 类，并基于它扩展出 `Cat` 和 `Dog` 子类
- 要求：
  - `Animal`：具有名字、体重属性，定义通用方法 `speak()`
  - 子类 `Dog` 和 `Cat` 分别重写 `speak()`，实现“汪汪”和“喵喵”

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

    def speak(self):
        return "发出声音"
```

```
class Dog(Animal):
    def speak(self):
        return "汪汪"

class Cat(Animal):
    def speak(self):
        return "喵喵"
```

■ 类的继承（Inheritance）

- 子类可以继承父类的属性和方法，避免重复代码
- 使用语法：`class 子类名(父类名):`
- 子类可以使用父类的所有“公有”方法/属性

```
class Animal:
    def speak(self):
        return "发出声音"

class Dog(Animal):
    pass

d = Dog()
print(d.speak())  # 输出：发出声音
```

■ 方法重写（Overriding）

- 子类可以定义与父类同名的方法以实现不同行为，这称为“重写”

- 运行时调用哪个方法取决于对象的真实类型，而不是变量类型
- 这正是多态的基础

```
class Dog(Animal):
    def speak(self):
        return "汪汪"

class Cat(Animal):
    def speak(self):
        return "喵喵"
```

使用 super() 调用父类方法

- `super()` 用于在子类中调用父类的方法，避免重复代码
- 应用场景：
  - 子类扩展父类功能
  - 多重继承中安全调用方法链（只需写一次`super`）

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)    # 调用父类构造函数
        self.breed = breed
```

■ 多态性（Polymorphism）与统一接口设计

- 多态：相同方法调用作用于不同类型的对象，表现出不同的行为
- 优点：
  - 调用代码与具体类解耦
  - 便于统一操作多个类

```
def animal_speak(animal):
    print(animal.speak())

animal_speak(Dog("Lucky", 10))   # 汪汪
animal_speak(Cat("Kitty", 5))    # 喵喵
```

示例：类层次设计（员工管理系统）

- 目标：建立一个基础的员工系统，包含通用员工（Employee）和两种特殊员工
  - 工程师（Engineer）、销售（Sales）
- 设计要点：
  - Employee：姓名、基本工资、计算年薪方法 get_annual_salary()
  - Engineer：年薪为基本工资 * 13
  - Sales：年薪为基本工资 * 12 + 销售提成

■ 示例：类层次设计（员工管理系统）

```
class Employee:
    def __init__(self, name, base_salary):
        self.name = name
        self.base_salary = base_salary

    def get_annual_salary(self):
        return self.base_salary * 12

class Engineer(Employee):
    def get_annual_salary(self):
        return self.base_salary * 13

class Sales(Employee):
    def __init__(self, name, base_salary, bonus):
        super().__init__(name, base_salary)
        self.bonus = bonus

    def get_annual_salary(self):
        return self.base_salary * 12 + self.bonus
```

示例：图书管理系统

- 场景：构建一个图书馆系统，管理书籍与借阅信息
- 类设计思路：
  - Book 类：包含书名、作者、ISBN等属性
  - LibraryUser 类：包含用户姓名、已借书籍列表、借书、还书方法

■ 示例：图书管理系统

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

class LibraryUser:
    def __init__(self, name):
        self.name = name
        self.borrowed_books = []

    def borrow(self, book):
        self.borrowed_books.append(book)

    def return_book(self, book):
        self.borrowed_books.remove(book)
```

面向对象模块化结构建议

- 建议一个中型项目中，类的组织结构为：

```
/project
    ├── main.py                 # 程序入口
    ├── student.py              # Student类定义
    ├── course.py               # Course类定义
    ├── utils.py                # 工具函数
    └── data/
        └── students.csv        # 学生数据
```

- 优点：
  - 每个类或职责分布在独立模块，清晰可维护
  - 避免类之间耦合
  - 便于单元测试与重用

引用项目中的模块：

```
from student import Student
```

回顾类的基本使用方式与深度设计视角

- 面向对象有两个视角：
  - **使用类**：即实例化、调用方法、访问属性
  - **设计类**：定义属性、构造方法、封装行为
- 使用 vs 实现：
- 从“调用类”转向“设计类”视角
- 如何让我们的类更像“内建对象”

```
# 使用角度
c1 = Coordinate(3, 4)
print(c1.distance(Coordinate(0, 0)))

# 实现角度（类定义）
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        ...
```

使用类构建类 —— 类的组合设计

- 类与类之间可以“组合”，即某个类的属性是另一个类的对象
- 使用场景举例：
  - Circle 类的中心是 Coordinate 对象，表示二维空间中的圆

```
class Circle:
    def __init__(self, center, radius):
        self.center = center  # Coordinate 实例
        self.radius = radius
```

- 好处：
  - 提高可重用性
  - 设计上更贴近现实世界

■ 输入类型检查与异常机制

- 在 __init__ 中可以使用 isinstance() 判断参数是否符合要求

```
class Circle:
    def __init__(self, center, radius):
        if not isinstance(center, Coordinate):
            raise ValueError("center必须是Coordinate类型")
        if not isinstance(radius, int):
            raise ValueError("radius必须是整数")
        self.center = center
        self.radius = radius
```

- 好处：
  - 提高程序健壮性
  - 明确接口契约

为已有的类设计方法

- 实现一个实例方法 is_inside(point)
- 判断某个点是否在圆内（欧几里得距离 < 半径）

```
class Circle:
    def __init__(self, center, radius):
        …
    def is_inside(self, point):
        return self.center.distance(point) < self.radius


c = Coordinate(2, 2)
p = Coordinate(1, 1)
circle = Circle(c, 2)
print(circle.is_inside(p))   # 输出 True
```

■ 类中的特殊方法（Dunder Methods）

- Python中很多操作（如 +、print()、len()）本质上是方法调用
- 特殊方法以双下划线（double underscore）开头和结尾
  - __init__：构造方法
  - __str__：打印时使用
  - __add__：+ 运算
  - __eq__：== 比较
  - __len__：len() 函数
- 通过定义这些方法，可以让自定义类表现得“像内置类型”一样自然

自定义 __str__() 打印方法

- 默认打印一个对象只显示内存地址，不直观

```
f = Fraction(3, 4)
print(f)  # 输出: <__main__.Fraction object at 0x...>
```

- 自定义 __str__ 提供人类友好的输出

```
class Fraction:
    def __init__(self, n, d):
        self.num = n
        self.denom = d
    def __str__(self):
        return f"{self.num}/{self.denom}"

f = Fraction(3, 4)
print(f)  # 输出: 3/4
```

实现 __add__() 运算符重载

- 想实现 f1 + f2，需要重载 __add__() 方法
- Python将 f1 + f2 转换为 f1.__add__(f2)

```
class Fraction:
    ...
    def __add__(self, other):
        top = self.num * other.denom + other.num * self.denom
        bottom = self.denom * other.denom
        return Fraction(top, bottom)

f1 = Fraction(1, 3)
f2 = Fraction(1, 6)
f3 = f1 + f2
print(f3)  # 输出: 9/18
```

■ 实现 __mul__() 与类型转换 __float__()

- __mul__()：实现乘法运算

```
def __mul__(self, other):
    return Fraction(self.num * other.num, self.denom * other.denom)
```

- __float__()：用于转换为浮点数

```
def __float__(self):
    return self.num / self.denom


f = Fraction(3, 4)
print(float(f))  # 输出：0.75
```

■ 继续为以上类拓展核心方法

- 为 Fraction 类添加 reduce() 方法用于将分数化简

```
def reduce(self):
    def gcd(a, b): # 找最大公约数
        while b != 0:
            a, b = b, a % b
        return a
    g = gcd(self.num, self.denom)
    return Fraction(self.num // g, self.denom // g)

f = Fraction(10, 20)
print(f.reduce())  # 输出: 1/2
```

- 注意：返回的是新对象，不改变原对象

■ 实现 __eq__() 方法 —— 自定义等号比较

- 默认情况下，Python使用对象的内存地址来判断是否相等

```
f1 = Fraction(1, 2)
f2 = Fraction(2, 4)
print(f1 == f2)  # False (尽管数学上相等)
```

- 自定义 __eq__() 让比较基于值

```
def __eq__(self, other):
    return self.num * other.denom == self.denom * other.num

f1 = Fraction(1, 2)
f2 = Fraction(2, 4)
print(f1 == f2)  # True
```

■ 对象类型检查

- 在定义类方法时，检查参数是否是合法类型是一种好习惯

```
def __add__(self, other):
    if not isinstance(other, Fraction):
        raise TypeError("只能加Fraction类型")
    ...
```

- 使用 isinstance() 比较而不是 type(x) == ...，因为支持继承链判断

```
print(isinstance(3, int))          # True
print(isinstance(True, int))       # True (因为bool是int子类)
print(isinstance("abc", str))      # True
```

■ 方法调用 与 运算符背后的机制

- 运算符其实只是语法糖，方法调用才是本质

```
a = Fraction(1, 2)
b = Fraction(2, 3)

# 三种写法本质等价:
print(a * b) # 语法糖
print(a.__mul__(b))
print(Fraction.__mul__(a, b))
```

- 推荐使用“语法糖”的写法，提高可读性

练习：设计一个 Vector2D 向量类

- 目标：实现一个 2D 向量类，支持以下操作：
  - 向量加法（+）
  - 向量打印（__str__）
  - 向量长度（magnitude()）

■ 练习：设计一个 Vector2D 向量类

```
import math

class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"<{self.x}, {self.y}>"

    def magnitude(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)
```

```
v1 = Vector2D(1, 2)
v2 = Vector2D(3, 4)
print(v1 + v2)              # 输出: <4, 6>
print(v1.magnitude())       # 输出: 2.236...
```

类中特殊方法协同使用

- 当一个类组合了另一个类对象时，可以将方法调用“转发”
- 例如 Circle 类中调用 Coordinate 的 字符串输出 方法

```
class Circle:
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def __str__(self):
        return f"Circle(center={self.center}, radius={self.radius})"
```

■ 类中特殊方法协同使用

- 当一个类组合了另一个类对象时，可以将方法调用“转发”
- 例如 Circle 类中调用 Coordinate 的 distance() 方法
- 前提是 Coordinate 实现了 __str__() 方法

```
class Coordinate:
    def __str__(self):
        return f"<{self.x}, {self.y}>"

c = Coordinate(3, 4)
circle = Circle(c, 5)
print(circle)  # 输出: Circle(center=<3, 4>, radius=5)
```

■ 实现__float__()：支持浮点数转换

- 自定义 __float__() 使对象可被强制转换为浮点数

```
class Fraction:
    def __float__(self):
        return self.num / self.denom

f = Fraction(3, 4)
print(float(f))  # 输出: 0.75
```

- 常用于科学计算、与内建库集成、排序等

■ 保持封装一致性：返回对象还是值

- 例如，Fraction类中的reduce() 方法可以有两种设计：
  - 返回新的 Fraction 对象（推荐，保留原始对象不变）
  - 或直接返回简化值，如整数（不推荐，会打破封装）

```
def reduce(self):
    g = gcd(self.num, self.denom)
    return Fraction(self.num // g, self.denom // g)
```

- 设计选择建议：
  - 如果是表示“对象自身的变化” → 使用 self 修改
  - 如果是“返回简化或计算后的新实体” → 返回新的对象

完整示例：分数类 Fraction（全功能）

- 支持功能：
  - 支持 +, *, float(), print()
  - 支持 == 判断
  - 实现 reduce() 简化分数
  - 分母为1时 __str__() 输出整数形式

■ 完整示例：分数类 Fraction（全功能）

```
class Fraction:
    def __init__(self, n, d):
        self.num = n
        self.denom = d

    def __str__(self):
        return str(self.num) if self.denom == 1 else f"{self.num}/{self.denom}"

    def __add__(self, other):
        top = self.num * other.denom + other.num * self.denom
        bot = self.denom * other.denom
        return Fraction(top, bot)

    def __eq__(self, other):
        return self.num * other.denom == self.denom * other.num
```

```
def reduce(self):
    def gcd(a, b):
        while b != 0:
            a, b = b, a % b
        return a
    g = gcd(self.num, self.denom)
    return Fraction(self.num // g, self.denom // g)
```

案例设计：账单金额建模类 Money

- 设计一个 Money 类，支持以下功能：
  - 属性：yuan（元），jiao（角），fen（分）
  - 方法：加法 __add__，字符串表示 __str__，转换为元金额 __float__

■ 案例设计：账单金额建模类 Money

```
class Money:
    def __init__(self, yuan, jiao, fen):
        self.total_fen = yuan * 100 + jiao * 10 + fen

    def __add__(self, other):
        return Money(0, 0, self.total_fen + other.total_fen)

    def __str__(self):
        y, remainder = divmod(self.total_fen, 100)
        j, f = divmod(remainder, 10)
        return f"{y}元{j}角{f}分"

    def __float__(self):
        return self.total_fen / 100
```

```
m1 = Money(3, 5, 6)
m2 = Money(2, 3, 4)
print(m1 + m2)        # 输出：5元9角0分
print(float(m1 + m2)) # 输出：5.9
```

抽象建模：数据结构的最小设计原则

- 在设计类时，需思考：
  - 最少需要哪些属性来表达这个对象
  - 哪些方法是与该对象强关联的

```
class Fraction:
    def __init__(self, num, denom=1):
        self.num = num # 分子
        self.denom = denom # 分母
```

- 最小设计的好处：
  - 降低耦合，提高重用
  - 更容易维护和扩展

■ 类的可拓展性与模块间协同设计

- 现实场景：多个类之间必须协同工作，例如：
  - User 类 与 Order 类
  - Book 类 与 Library 类
- 设计思路：尽量通过传入对象，而不是仅传原始数据

```
class User:
    def __init__(self, name):
        self.name = name
        self.orders = []

    def place_order(self, order):
        self.orders.append(order)
```

```
class Order:
    def __init__(self, item, price):
        self.item = item
        self.price = price


u = User("Alice")
o = Order("Python书籍", 88)
u.place_order(o)
```

继承中的方法重载与方法复用

- 父类方法可被子类重写（override），也可复用（通过 super()）

```
class Employee:
    def __init__(self, name):
        self.name = name
    def get_salary(self):
        return 3000

class Engineer(Employee):
    def get_salary(self):
        return super().get_salary() + 2000
```

- 使用 super() 可以调用父类中被重写的方法
- 继承结构设计与功能差异处理

■ 练习：构建一组有继承关系的类

- 设计一个交通工具体系，包括：
  - 父类：Vehicle，具有属性speed和方法sound()
  - 子类：Car, Bike, Plane，分别重写sound()方法

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed
    def sound(self):
        return "一般声音"
```

```
class Car(Vehicle):
    def sound(self):
        return "轰轰"

class Bike(Vehicle):
    def sound(self):
        return "铃铃"

class Plane(Vehicle):
    def sound(self):
        return "嗡嗡"
```

```
vehicles = [Car(120), Bike(30), Plane(600)]
for v in vehicles:
    print(v.sound())
```

■ 拓展设计：抽象基类（面向多态的统一接口）

- 当你希望多个子类都必须实现某一组方法，可用“接口类”或“抽象基类”
- Python中可以使用 abc 模块中的 ABC 与 @abstractmethod：

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r
```

- 优势：
  - 明确每个子类必须实现的核心方法
  - 保证接口一致，便于统一处理

■ 实战案例：学生与课程管理系统（支持运算）

- 目标建模：
  - 每个学生（Student）可选多门课程（Course）
  - 每门课程有学分与成绩
  - 支持计算 GPA、课程总学分等

■ 实战案例：学生与课程管理系统（支持运算）

```
class Course:
    def __init__(self, name, credit, score):
        self.name = name
        self.credit = credit
        self.score = score

    def __float__(self):
        return self.score
```

■ 实战案例：学生与课程管理系统（支持运算）

```
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []

    def add_course(self, course):
        self.courses.append(course)

    def gpa(self):
        total = sum(c.credit * c.score for c in self.courses)
        total_credit = sum(c.credit for c in self.courses)
        return total / total_credit
```

■ 实战案例：学生与课程管理系统（支持运算）

```
s = Student("小明")
s.add_course(Course("数学", 3, 90))
s.add_course(Course("英语", 2, 80))
print(s.gpa())  # 输出: 86.0
```

■ 实战案例：学生与课程管理系统（支持运算）

- 目录结构推荐：

```
/school/
    ├── __init__.py
    ├── student.py      # Student类
    ├── course.py       # Course类
    └── main.py         # 业务逻辑/测试代码
```

- 使用方式：

```
from student import Student
from course import Course
```

- 每个类职责清晰、便于维护
- 有利于单元测试与团队协作

■总结

- 类和对象的核心概念
- 构造函数、属性、方法
- 封装、继承、重写与多态
- 实战设计类、组织模块、真实建模案例
- 深入理解了 Python 类中的特殊方法（Dunder Methods）
- 学习了如何使自定义类像内建类型一样自然地工作
- 掌握了类的组合设计、继承重载、多态性运用
- 构建了多个面向真实问题的类结构示例

Reading and QA Time

See you next week !