

Python 面向对象编程

Python 从设计之初就已经是一门面向对象的语言，正因为如此，在 Python 中创建一个类和对象是很容易的。

面向对象编程（OOP）不是单纯的语法技巧，而是以“**对象**”为核心的代码组织思维——把现实世界的实体抽象成“属性（数据）+方法（行为）”的类，通过封装、继承、多态让代码更贴合现实逻辑，同时解决复用、维护、扩展的问题。

Python 作为纯面向对象语言，“一切皆对象”的特性让 OOP 的实现更灵活，更强调**约定优于强制**，这既是优势也要求开发者有更好的代码规范意识。

一、类与对象

1. 类是“模板”，对象是“模板实例化的具体产物”

类定义了某一类事物的**公共特征和行为**（类属性、类方法），对象则是具备**专属特征的个体**（实例属性、实例方法）。

比如 Dog 类定义了所有狗的共同属性 species="Canis familiaris"，而 dog1 = Dog("旺财", 3) 则是拥有专属 name 和 age 的具体对象。

类属性的“共享性”需要注意——修改类属性会影响所有实例，这一特性适合做全局配置，但如果误将实例专属的属性定义为类属性，会引发意想不到的 bug（比如多个员工实例共享同一个 salary 类属性）。

2. 类与实例的绑定关系：self 和 cls

- `self`: 实例方法的第一个参数，指向**当前实例对象**，是实例属性和方法的“入口”——通过 `self.xxx` 才能访问实例的专属数据，Python 会自动传递该参数，无需手动传入。
- `cls`: 类方法的第一个参数，指向**类本身**，用于访问类属性或创建实例（替代构造器），同样由 Python 自动传递。

思考：为什么实例方法必须带 `self`？

本质是 Python 的**方法绑定机制**——未绑定 `self` 的方法只是类中的普通函数，无法关联到具体实例，这一点和 Java 等语言的隐式 `this` 不同，是 Python OOP 的关键细节。

二、三大特性

1. 封装：“可控访问”

封装的核心是**将数据和操作数据的方法封装为一个整体**，并对外部访问做限制，目的

是隐藏内部实现细节，避免外部随意修改数据导致的逻辑混乱。

Python 没有 public/private 关键字，通过**命名约定**实现封装：

- 公有：直接命名（如 name、bark()），外部可自由访问；
- 受保护：单下划线开头（如 _age），约定“类内部+子类可访问，外部不建议访问”；
- 私有：双下划线开头（如 __gender），通过**名称重整**（变成类名 __gender）实现强制隐藏，外部无法直接访问。

关键思考：

1. Python 的私有不是“绝对隐藏”，外部可通过类名__属性强行访问，这说明 Python 的封装是**给遵守规则的人设计的**，而非防黑客；
2. 封装的最佳实践不是单纯用下划线，而是**通过@property 装饰器实现属性的可控读写**——既隐藏了属性的直接操作，又能在设置属性时做校验（比如年龄不能为负数），这才是封装的真正价值。

2. 继承

继承是让子类复用父类的属性和方法，同时可扩展自己的专属功能，Python 支持**单继承、多继承、多层继承**，核心语法是 class 子类(父类 1, 父类 2)：。

核心要点

1. `super()` 函数：不是简单“调用父类方法”，而是按照**方法解析顺序 (MRO)** 调用上一级类的方法，解决了多继承中的方法调用混乱问题，是继承中必用的工具；
2. 方法重写：子类可重新定义父类的方法，实现个性化逻辑（如 Animal 的 `make_sound()` 在 Dog 中重写为“汪汪”），重写时可通过 `super()` 保留父类逻辑，再扩展子类逻辑；
3. 属性访问顺序：实例自身属性 → 子类类属性 → 父类类属性 → 祖先类类属性，找不到则抛 `AttributeError`。

思考：

1. 多继承的“钻石问题”：多个父类继承自同一个基类时，可能导致基类方法被多次调用，Python 通过**MRO 算法**（深度优先+广度优先结合）解决，可通过类名.`__mro__` 查看方法调用顺序，开发中尽量减少多继承的使用；
2. 继承的原则：**组合优于继承**——继承是“is-a”关系（如 Dog is a Animal），组合是“has-a”关系（如 Bird has a Wings、Bird has a Legs）。当需要复用的功能不是“父子关系”时，用组合（将其他类的实例作为当前类的属性）更灵活，可避免继承的层级臃肿和耦合过高。

3. 多态

多态的核心是**不关注对象的具体类型，只关注对象的行为**——不同类的对象，只要拥

有相同的方法名，就能对同一消息做出不同的响应，实现“接口统一，实现各异”。

比如 Dog、Cat、Duck 都有 speak() 方法，无需判断对象类型，直接调用 animal_sound(animal) 就能得到对应的叫声。

思考：Python 的多态是“动态的”（鸭子类型）——不像 Java 需要实现接口或继承抽象类，Python 只要对象拥有对应的方法，就可以被当作该类型使用，这让多态的实现更简洁，也更符合 Python 的“灵活”特性。

核心价值：降低代码耦合，提高扩展性——新增一个 Pig 类，只要实现 speak() 方法，无需修改 animal_sound() 函数，就能直接使用，符合“开闭原则”。

三、高级特性

1. 三大方法：实例方法、类方法、静态方法

易混淆这三种方法，核心区别在于是否依赖类/实例的状态，适用场景清晰后就不会用错：

方法类型	装饰器	第一个参数	依赖状态	适用场景
实例方法	无	self	依赖实例属性	操作实例的专属数据（如对象的增删改查）
类方法	@classmethod	cls	依赖类属性	操作类属性、创建替代构造器（如 from_string() 从字符串生成实例）
静态方法	@staticmethod	无	不依赖类/实例	与类相关的工具函数（如数学计算、格式校验），仅为代码组织

思考：静态方法本质是“放在类里的普通函数”，如果去掉类的包裹，函数依然能独立运行，这是判断是否该用静态方法的关键——不要为了“凑类的结构”而强行用静态方法。

2. 抽象基类（ABC）：强制子类实现指定方法

抽象基类是不能被实例化的基类，通过 `abc.ABC` 和 `@abstractmethod` 装饰器定义，要求子类必须实现所有抽象方法，否则无法实例化。

比如 `Shape` 抽象类定义了 `area()` 和 `perimeter()` 抽象方法，`Rectangle` 和 `Circle` 必须实现这两个方法才能创建对象。

思考：抽象基类的核心作用是定义接口规范，让子类遵循统一的方法名，避免因方法名不一致导致的多态失效。它不是用来实现代码复用的，复用应该用普通的父类。

3. 特殊方法

特殊方法是以双下划线开头和结尾的方法（如 `__init__`、`__add__`、`__str__`），Python 会在执行原生操作时自动调用，让自定义对象能像内置对象（如列表、整数）一样使用 `+`、`print()`、`len()` 等操作。

常用特殊方法及场景

- `__init__`: 构造方法，实例化时初始化属性；
- `__str__`/`__repr__`: 自定义对象的字符串表示，`__str__` 用于 `print()`（面向用户），`__repr__` 用于调试（面向开发者）；
- `__add__`/`__sub__`: 运算符重载，让对象支持 `+`/`-` 等运算；
- `__len__`: 让对象支持 `len()` 函数；
- `__eq__`: 定义 `==` 的比较规则，默认是比较对象的内存地址，重写后可按属性比较。

思考：运算符重载要遵循“直觉性”——比如 `Point` 类的 `__add__` 应该实现坐标相加，而不是无意义的数值计算，否则会让代码难以理解。

4. slots: 优化内存，限制动态属性

Python 的普通类实例会用字典存储属性，字典有额外的内存开销，当创建大量实例时，内存占用会很高。`__slots__` 可以指定实例只能拥有的属性，让 Python 用固定大小的数组替代字典，大幅减少内存占用，同时禁止动态给实例添加属性。

思考： `__slots__` 是“空间换时间”的优化手段，仅适用于需要创建大量实例的场景（如数据处理、爬虫），普通场景无需使用，否则会失去 Python 动态添加属性的灵活性。另外，`__slots__` 仅对当前类的实例生效，子类需要重新定义才能生效。

四、开发原则

1. 五大设计原则 (SOLID)

- 单一职责原则：一个类只做一件事，只有一个引起变化的原因（如 `User` 类只处理用户信息，不处理订单逻辑）；
- 开闭原则：对扩展开放，对修改关闭（新增功能通过扩展子类实现，而非修改父类代码）；

- 里氏替换原则：子类可以替换父类，且不会影响程序的正常运行（子类不能破坏父类的方法逻辑）；
- 接口隔离原则：避免大而全的接口，拆分为小而专的接口（Python 用抽象基类实现，避免子类实现无关方法）；
- 依赖倒置原则：依赖抽象（如抽象基类），而非具体实现（如具体的子类），降低耦合。

2. 常见坑点与解决思路

1. **子类重写`__init__`忘记调用父类`__init__`**: 导致父类的属性未初始化，解决：用`super().__init__()`调用父类构造方法，保留父类逻辑；
2. **过度使用继承**: 导致类的层级过深、耦合过高，解决：优先用组合替代继承，将独立的功能抽离为单独的类，通过实例化的方式组合；
3. **滥用类属性**: 将实例专属的属性定义为类属性，导致多个实例共享数据，解决：实例属性在`__init__`中用`self.xxx`定义；
4. **直接修改实例属性**: 导致数据校验缺失，解决：用`@property`装饰器封装属性，通过`setter`方法做校验；
5. **多继承的方法冲突**: 解决：查看`__mro__`确认方法调用顺序，尽量减少多继承，或用组合替代。