Python面向对象

1. 概述

1.1 面向过程

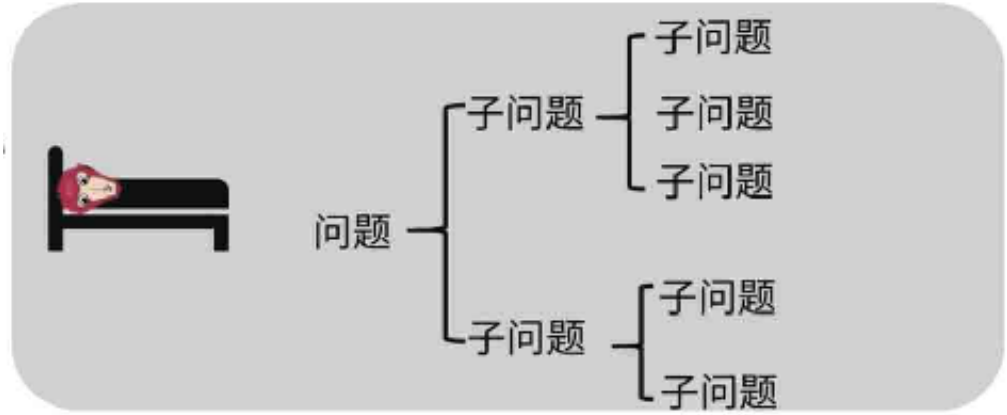
(1) 定义：分析出解决需求的步骤(1234)，然后逐步实现。

> 例如：婚礼筹办
>
> 1. 请柬（选照片、措词、制作）
> 2. 宴席（场地、找厨师、准备桌椅餐具、计划菜品、购买食材）
> 3. 仪式（定婚礼仪式流程、请主持人）

(2) 公式：程序 = 算法 + 数据结构

(3) 优点：所有环节、细节自己掌控。

缺点：考虑所有细节，工作量大。



1.2 面向对象

(1) 定义：找出解决问题的人，然后分配职责。

> 例如：婚礼筹办
>
> -- 发请柬：找摄影公司（拍照片、制作请柬）
>
> -- 宴席：找酒店（告诉对方标准、数量、挑选菜品）
>
> -- 婚礼仪式：找婚庆公司（对方提供司仪、制定流程、提供设备、帮助执行）

(2) 公式：程序 = 对象 + 交互的动作方法

(3) 优点：

a. 思想层面：

- 可模拟现实情景，更接近于人类思维。
- 有利于梳理归纳、分析解决问题。

b. 技术层面：

- 高复用：对重复的代码进行封装，提高开发效率。
- 高扩展：增加新的功能，不修改以前的代码。
- 高维护：代码可读性好，逻辑清晰，结构规整。

(4) 缺点：学习曲线陡峭。

1.3 对比

1. **面向过程**：实现小功能/在一个函数的内部，将需求划分为多个步骤,逐一实现（直接做）

```
思考流程：
    获取数据
    逻辑处理
    显示结果
```

2. **面向对象**：设计软件架构, 将需求分配给多个人,建立交互.（谁?干嘛?）

```
思考流程：
            现实世界            虚拟世界
      出租车 ----抽象化->类 ----具体化-> 对象
                          车牌号          京c007
                          品牌            奔驰
                          颜色            白色
```

2. 类和对象

(1) 类：一个抽象的概念，即生活中的”类别”。

(2) 对象：类的具体实例个体。类是创建对象的”模板”。

(3)类的成员：

抽象: 从具体事物中抽离出共性，本质

- 数据成员：名词类型的"状态"。
- 方法成员：动词类型的"行为"。

> 手机:
>
> 数据：品牌，价格，颜色
>
> 行为： 通话，发短信

(4) 类与类行为不同，对象与对象数据不同。

2.1 语法

2.1.1 定义类

```
class 类名:
  """
    文档说明
  """
  def __init__(self,参数):
    self.实例变量 = 参数

  # 方法成员
  def work():
```

- 类名所有单词首字母大写.
- init 也叫构造函数，创建对象时被自动调用。
- self 变量绑定的是被创建的对象，名称可以随意。

2.1.2 实例化对象

(1) 代码

```
变量 = 类名(参数)
```

(2) 说明

- 变量存储的是实例化后的对象**地址**
- 类名后面的参数按照构造函数(init)的形参传递

(3) 演示

```
```

```python
class Wife:
    """
        自定义老婆类
    """
    # 初始化对象数据
    def __init__(self, name, age, sex):
        self.name = name
        self.age = age
        self.sex = sex

    # 行为(方法=函数)
    def play(self):
        print(self.name, "玩耍")

# 调用构造函数(__init__)
shang_er = Wife("双儿", 26, "女")
# 操作对象的数据
shang_er.age += 1
print(shang_er.age)
# 调用对象的函数
# 通过对象地址调用方法,会自动传递对象地址.
shang_er.play()
print(shang_er)# <__main__.Wife object at 0x7f390e010f28>
```

> 练习：创建手机类，实例化两个对象并调用其函数，最后画出内存图。

数据：品牌、价格、颜色

行为：通话

2.2 实例成员

2.2.1 实例变量

(1) 语法

- 定义：对象.变量名
- 访问：对象.变量名

(2) 说明

a. 首次通过对象赋值为创建，再次赋值为修改.

```python
lili = Wife()
lili.name = "丽丽"
lili.name = "莉莉"
```

b. 通常在构造函数(__init_)中创建

```python
lili = Wife("丽丽",24)
print(lili.name)
```

(3) 每个对象存储一份，通过对象地址访问

(4) 作用：描述某个对象的数据

(5) `__dict__`：对象的属性，用于存储自身实例变量的字典。

```
print(jian_ning) # <__main__.Wife object at 0x7f49179c0f98>
print(jian_ning.__dict__) # {'name': '建宁公主', 'height': 170, 'face_score': 95}
```

2.2.2 实例方法

(1) 定义

```
def 方法名称(self, 参数):
    方法体
```

(2) 调用：

```
对象.方法名称(参数)
```

(3) 说明

- 至少有一个形参，第一个参数绑定调用这个方法的对象,一般命名为**self**
- 无论创建多少对象，**方法只有一份**，并且被所有对象**共享**

(4) 作用：表示对象行为

> 示例

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

    def print_self(self):
        print("我是: ", self.name)

lili = Wife("丽丽")  # dict01 = {"name":"丽丽"}
lili.name = "莉莉"  # dict01["name"] = "莉莉"
print(lili.name)  # print(dict01["name"])
lili.print_self()
print(lili.__dict__)  # {"name":"丽丽"}

"""
# 支持动态创建类成员
# 类中的成员应该由类的创造者决定
class Wife:
    pass

w01 = Wife()
w01.name = "莉莉"
print(w01.name)#对象.变量名
```

```
"""

"""
# 实例变量的创建要在构造函数中__init__
class Wife:
    def set_name(self,name):
        self.name = name

w01 = Wife()
w01.set_name("丽丽")
print(w01.name)
"""
```

> 练习1：创建狗类，实例化两个对象并调用其函数，画出内存图。

数据：品种、昵称、身长、体重

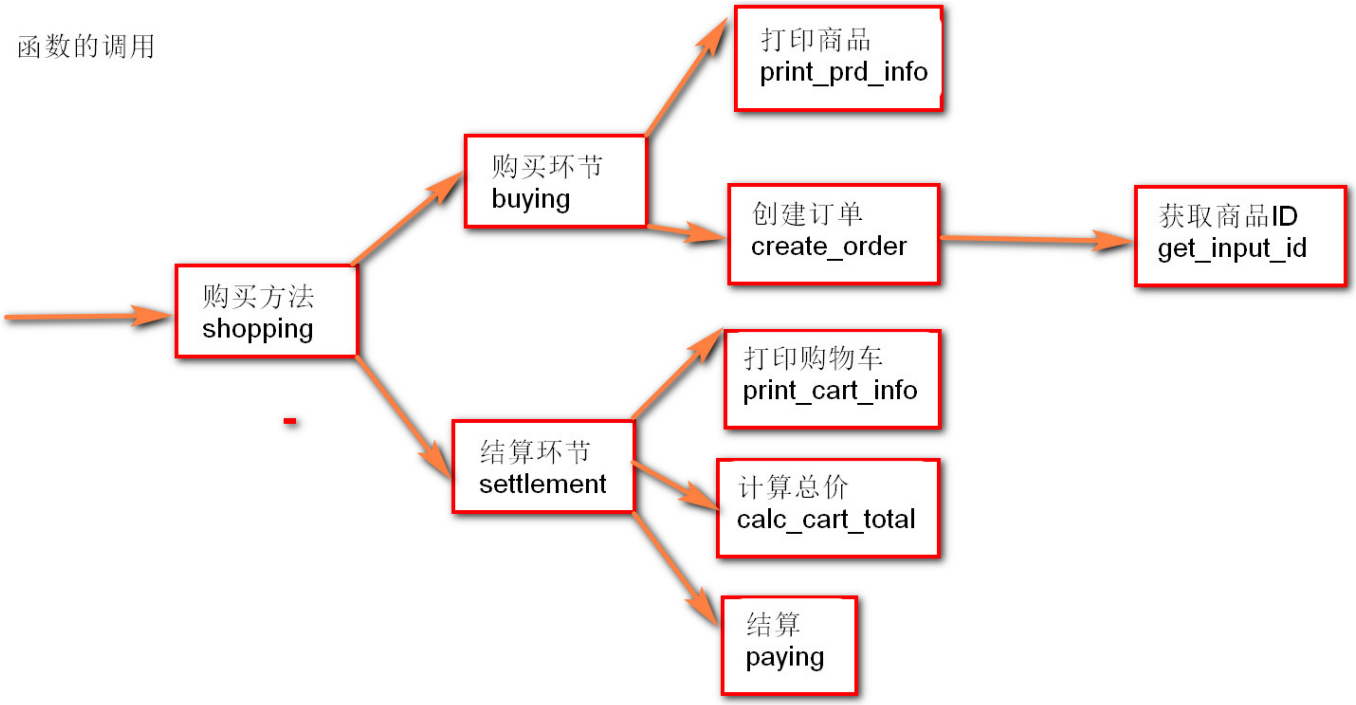
行为：吃(体重增长1)

> 练习2：将面向过程代码改为面向对象代码

```
list_commodity_infos = [
  {"cid": 1001, "name": "屠龙刀", "price": 10000},
  {"cid": 1002, "name": "倚天剑", "price": 10000},
  {"cid": 1003, "name": "金箍棒", "price": 52100},
  {"cid": 1004, "name": "口罩", "price": 20},
  {"cid": 1005, "name": "酒精", "price": 30},
]

# 订单列表
list_orders = [
  {"cid": 1001, "count": 1},
  {"cid": 1002, "count": 3},
  {"cid": 1005, "count": 2},
]

def print_single_commodity(commodity):
  print(f"编号:{commodity['cid']},商品名称:{commodity['name']},商品单价:
{commodity['price']}")

# 1. 定义函数,打印所有商品信息,格式: 商品编号xx,商品名称xx,商品单价xx.
def print_commodity_infos():
  for commodity in list_commodity_infos:
      print_single_commodity(commodity)

# 2. 定义函数,打印商品单价小于2万的商品信息
def print_price_in_2w():
  for commodity in list_commodity_infos:
      if commodity["price"] < 20000:
      print_single_commodity(commodity)

```

```python
# 3. 定义函数,打印所有订单中的商品信息,
def print_order_infos():
  for order in list_orders:
    for commodity in list_commodity_infos:
        if order["cid"] == commodity["cid"]:
            print(f"商品名称{commodity['name']},商品单价:{commodity['price']},数量{order['count']}.")
                break # 跳出内层循环

# 4. 查找最贵的商品(使用自定义算法,不使用内置函数)
def commodity_max_by_price():
  max_value = list_commodity_infos[0]
  for i in range(1, len(list_commodity_infos)):
      if max_value["price"] < list_commodity_infos[i]["price"]:
        max_value = list_commodity_infos[i]
  return max_value

# 5. 根据单价对商品列表降序排列
def descending_order_by_price():
  for r in range(len(list_commodity_infos) - 1):
    for c in range(r + 1, len(list_commodity_infos)):
        if list_commodity_infos[r]["price"] < list_commodity_infos[c]["price"]:
      list_commodity_infos[r], list_commodity_infos[c] = list_commodity_infos[c], list_commodity_infos[r]
```

函数的调用



2.2.3 跨类调用(注入)

```python
# 写法1：method中直接创建对象
# 语义：老张“每次创建”一辆新车去
class Person:
    def __init__(self, name=""):
        self.name = name

    def go_to(self,position):
        print("去",position)
        car = Car()
        car.run()

class Car:
    def run(self):
        print("跑喽～")

lz = Person("老张")
lz.go_to("东北")
```

```python
# 写法2：在init构造函数中创建对象
# 语义：老张开“自己的”车去
class Person:
    def __init__(self, name=""):
        self.name = name
        self.car = Car()

    def go_to(self,position):
        print("去",position)
        self.car.run()

class Car:
    def run(self):
        print("跑喽～")

lz = Person("老张")
lz.go_to("东北")
```

```python
# 方式3：通过参数传递，外部传object进去
# 语义：老张用交通工具去
class Person:
    def __init__(self, name=""):
        self.name = name

    def go_to(self,vehicle,position):
        print("去",position)
        vehicle.run()

class Car:
    def run(self):
        print("跑喽～")
```

```
14
15  lz = Person("老张")
16  benz = Car()
17  lz.go_to(benz,"东北")
```

> 练习1：以面向对象思想,描述下列情景.
>
> 小明请保洁打扫卫生

```
1   # 语义:小明每次预约保洁服务(保洁员/扫地机器人)，依赖于抽象了
2   class Client:
3       def __init__(self, name=""):
4           self.name = name
5
6       def notify(self, server):
7           print("发出通知")
8           server.cleaning()
9
10  class Cleaner: # 可扩展继承!
11      def cleaning(self):
12          print("打扫卫生")
13
14  xm = Client("小明")
15  cleaner = Cleaner()
16  xm.notify(cleaner)
17  # xm.notify("保洁")
```

> 练习2：以面向对象思想,描述下列情景.
>
> 玩家攻击敌人,敌人受伤(头顶爆字).
>
> 玩家攻击敌人,敌人受伤(根据玩家攻击力，减少敌人的血量).

```
1   # 玩家攻击目标(敌人)
2   class Player:
3       def attack(self,target):
4           print("发起攻击")
5           target.damage()
6
7   class Enemy:
8       def damage(self):
9           print("头顶爆字")
10
11  p = Player()
12  e = Enemy()
13  p.attack(e)
```

2.3 类成员

2.3.1 类变量

(1) 定义：在类中，方法外。

```
class 类名:
    变量名 = 数据
```

(2) 调用：

```
类名.变量名
```

(3) 特点：

- 随类的加载而加载， 存在优先于对象
- 只有一份，被所有对象共享。互相影响
- 不建议通过对象访问类变量

(4) 作用：描述所有对象的共有数据

2.3.2 类方法

(1) 定义：

```
@classmethod
def 方法名称(cls,参数):
    方法体
```

(2) 调用：

```
类名.方法名(参数)
```

(2) 说明

- 至少有**一个**形参，第一个形参用于绑定类，一般命名为'cls'
- 使用**@classmethod**修饰的目的是调用类方法时可以隐式传递类
- 类方法中**不能**访问实例成员，实例方法中可以访问类成员
- 不建议通过对象访问类方法

(3) 作用：操作类变量

> 演示：支行与总行钱的关系

```
class ICBC:
    """
        工商银行
    """
    # 类变量：总行的钱
    total_money = 1000000
    # 类方法：操作类变量
    @classmethod
    def print_total_money(cls):
```

```
            # print("总行的钱: ", ICBC.total_money)
            print("总行的钱: ", cls.total_money)

        def __init__(self, name,money=0):
            self.name = name
            # 实例变量: 支行的钱
            self.money = money
            # 总行的钱因为创建一家支行而减少
            ICBC.total_money -= money

    ttzh = ICBC("天坛支行",100000)
    xdzh = ICBC("西单支行",200000)
    # print("总行的钱: ", ICBC.total_money)
    ICBC.print_total_money()
```

> 练习：创建对象计数器，统计构造函数执行的次数，使用类变量实现并画出内存图。

```
class Wife:
    pass

w01 = Wife("双儿")
w02 = Wife("阿珂")
w03 = Wife("苏荃")
w04 = Wife("丽丽")
w05 = Wife("芳芳")
Wife.print_count()# 总共娶了5个老婆
```

2.3.3 静态方法

(1) 定义：

```
@staticmethod
def 方法名称(参数):
    方法体
```

(2) 调用：

```
类名.方法名称(参数)
```

(3) 说明

- 使用@ staticmethod修饰的目的是该方法**不需要**隐式传参数。
- 静态方法**不能**访问实例成员和类成员
- 不建议通过对象访问静态方法

(4) 作用/何时：定义常用的工具函数,然后类名访问

2.5 简化类dataclass

如果只需要一个简单的类，可以使用 Python 的 **dataclass**：

```python
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    gender: str

# 创建一个实例
person2 = Person(name="Bob", age=25, gender="Male")

# 访问属性
print(person2.name)   # 输出: Bob
print(person2.age)    # 输出: 25
print(person2.gender) # 输出: Male
```

3. 三大特征

3.1 封装

3.1.1 class角度

(1) 定义：将一些基本数据类型复合成一个自定义类型(class)。

(2) 优势：

- 将数据与对数据的操作相关联(封装后再增删改查)。
- 代码可读性更高（类是对象的模板）。

3.1.2 私有成员+属性

(1) 定义： 向类外提供必要的功能，隐藏实现的细节。

(2) 优势： 简化编程，使用者不必了解具体的实现细节，只需要调用对外提供的功能。

(3) 私有成员：

- 作用：无需向类外提供的成员，可以通过私有化进行屏蔽。
- 做法：命名使用 双下划线开头 。
- 本质：障眼法，实际也可以访问。
- 私有成员的名称被修改为：_类名__*成员名，可以直接通过* __dict__属性查看。

> 演示

```python
class MyClass:
    def __init__(self, data):
        self.__data = data

    def __func01(self):
        print("func01执行了")

m01 = MyClass(10)
# print(m01.__data) # 无法访问
print(m01._MyClass__data)
print(m01.__dict__)  # {'_MyClass__data': 10}

# m01.__func01() # 无法访问
m01._MyClass__func01()
```

3.1.3 保护属性

作用：保护实例变量，外部以为自己能直接修改

定义：

```python
@property
def 属性名(self):
  return self.__属性名

@属性名.setter
def 属性名(self, value):
  self.__属性名= value
```

调用：

```
修改：对象.属性名 = 数据
获得：变量 = 对象.属性名
```

> 练习1：创建敌人类，并保护数据在有效范围内

数据：姓名、攻击力、血量

0-100　0-500

> 练习2：创建技能类，并保护数据在有效范围内

数据：技能名称、冷却时间、攻击力度、消耗法力(只读)

0 -- 120　0 -- 200　100 -- 100

2种形式：

```python
# 1. 读取属性
#       在实例变量读写过程中进行控制(数据验证,修改)
# 快捷键:props + 回车
```

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

    @property
    def radius(self):
        """获取半径"""
        return self._radius

    @radius.setter
    def radius(self, value):
        """设置半径"""
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

# 创建实例
c = Circle(5)

# 读取属性
print(c.radius)  # 输出: 5

# 修改属性
c.radius = 10
print(c.radius)  # 输出: 10

# 尝试设置负值（会报错）
# c.radius = -5  # ValueError: Radius cannot be negative

```

```python
# 2. 只读属性
#        为私有变量提供读取功能
# 快捷键:prop + 回车
class Circle:
    def __init__(self, radius):
        self._radius = radius  # 使用私有变量存储值

    @property
    def radius(self):
        """只读属性: 获取半径"""
        return self._radius

# 创建实例
c = Circle(5)

# 访问只读属性
print(c.radius)  # 输出: 5

# 尝试修改只读属性（会报错）
# c.radius = 10  # AttributeError: can't set attribute
```

3.2 继承

3.2.1 继承方法

(1) 语法:

```
class 父类:
   def 父类方法(self):
     方法体

class 子类(父类):
   def 子类方法(self):
     方法体

儿子 = 子类()
儿子.子类方法()
儿子.父类方法()
```

(2) 说明：子类直接拥有父类的方法.

(3) 适用性/何时：多个类型,代码上有共性且概念上统一

> 演示：

```
class Person:
    def say(self):
        print("说话")

class Teacher(Person):
    def teach(self):
        self.say()
        print("教学")

class Student(Person):
    def study(self):
        self.say()
        print("学习")

qtx = Teacher()
qtx.say()
qtx.teach()

xm = Student()
xm.say()
xm.study()
```

3.2.2 内置函数比较对象

(1) isinstance(对象, 类型) : 返回指定对象是否是某个类的对象。

(2) issubclass(类型，类型) : 返回指定类型是否属于某个类型。

> 演示

```
# 子类对象既能访问自身也能访问父类成员
qtx = Teacher()
qtx.teach()
qtx.say()  # 类外通过对象名调用父类方法

zs = Person()# 但是，父类对象只能访问自身成员
zs.say()

# 1.对象 是一种 类型： isinstance(对象,类型)
# 老师对象 是一种 老师类型
print(isinstance(qtx, Teacher))  # True
# 老师对象 是一种 人类型
print(isinstance(qtx, Person))  # True
# 老师对象 是一种 学生类型
print(isinstance(qtx, Student))  # False
# 人对象 是一种 学生类型
print(isinstance(p, Student))  # False

# 2.类型 是一种 类型： issubclass(类型,类型)
# 老师类型 是一种 老师类型
print(issubclass(Teacher, Teacher))  # True
# 老师类型 是一种 人类型
print(issubclass(Teacher, Person))  # True
# 老师类型 是一种 学生类型
print(issubclass(Teacher, Student))  # False
# 人类型 是一种 学生类型
print(issubclass(Person, Student))  # False

# 3.是的关系(绝对判定!)
# 老师对象的类型 是 老师类型
print(type(qtx) == Teacher)  # True
# 老师对象的类型 是 人类型
print(type(qtx) == Person)  # False
```

> 练习：

创建子类：狗(跑)，鸟类(飞)

创建父类：动物(吃)

体会子类复用父类方法

体会 isinstance 、issubclass 与 type 的作用.

3.2.3 继承数据

(1) 语法

```
class 子类(父类):
 def __init__(self,父类参数,子类参数):
   super().__init__(参数) # 调用父类构造函数
   self.实例变量 = 参数
```

(2) 说明: 子类如果没有构造函数，将自动执行父类的，但如果有构造函数将覆盖父类的。此时必须通过super()函数调用父类的构造函数，以确保父类实例变量被正常创建。

> 演示

```
class Person:
    def __init__(self, name="", age=0):
        self.name = name
        self.age = age

# 子类有构造函数,不会使用继承而来的父类构造函数
class Student(Person):
    # 子类构造函数: 父类构造函数参数,子类构造函数参数
    def __init__(self, name, age, score):
        # 调用父类构造函数
        super().__init__(name, age)
        self.score = score

ts = Person("唐僧",22)
print(ts.name)
kw = Student("悟空", 23, 100)
print(kw.name)
print(kw.score)
```

> 练习:

创建父类：车(品牌，速度)

创建子类：电动车(电池容量,充电功率)

创建子类对象并画出内存图。

3.2.4 定义

(1) 概念： 重用现有类的功能，并在此基础上进行扩展or改变行为。

(2) 说明：子类直接具有父类的成员（共性），还可以扩展新功能。

(3) 相关知识

- 父类（基类、超类）、子类（派生类）。
- 父类相对于子类更抽象，范围更宽泛；子类相对于父类更具体，范围更狭小。
- 单继承：父类只有一个（例如 Java，C#）。

  多继承：父类有多个（例如C++，Python）。

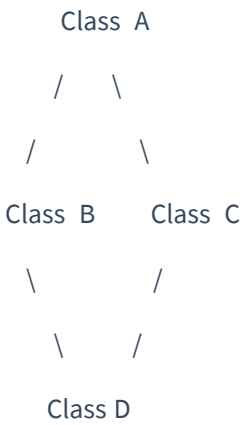
- Object类：任何类都直接或间接继承自 object 类。

3.2.5 多继承

(1) 定义：一个子类继承两个或两个以上的基类，父类中的属性和方法同时被子类继承下来。

(2) 同名方法解析顺序（MRO， Method Resolution Order）：

类自身 --> 父类继承列表（由左至右）--> 再上层父类



> 练习：写出下列代码在终端中执行效果

```
class A:
  def func01(self):
      print("A")
      super().func01()

class B:
    def func01(self):
      print("B")

class C(A,B):
  def func01(self):
      print("C")
      super().func01()

```

```python
class D(A, B):
    def func01(self):
      print("D")
      super().func01()

class E(C,D):
    def func01(self):
      print("E")
      super().func01()

e = E()
e.func01() # E C D A B
```

3.3 多态

多态：重写是多态, 同一个方法在不同子类中有不同的实现

3.3.1 重写内置函数

(1) 定义：Python中，以双下划线开头、双下划线结尾的是系统定义(内置)的成员。我们可以在自定义类中进行**重写**，从而改变其行为

本质： 重写内置函数！改变底层逻辑，方便调用简单！

何时： 想想底层逻辑需要到什么，然后我们去overwrite什么方法

1. __str__ 函数

定义： 将对象转换为字符串, 对人友好的print出来

> 演示：

```python
class Person:
    def __init__(self, name="", age=0):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}的年龄是{self.age}"

wk = Person("悟空", 26)
print(wk)
# 1.一般情况：<__main__.Person object at 0x7fbabfbc3e48>
# 2.用了这个方法：悟空的年龄是26
```

> 练习：

直接打印商品对象: xx的编号是xx,单价是xx

直接打印敌人对象: xx的攻击力是xx,血量是xx

```python
class Commodity:
    def __init__(self, cid=0, name="", price=0):
        self.cid = cid
        self.name = name
        self.price = price


    def __str__(self):
        return f"{self.name}的编号是{self.cid},单价是{self.price}"

printer=Commodity(1,'打印机',999)
print(printer)
```

=>简化

dataclass

```python
from dataclasses import dataclass

@dataclass
class Commodity:
    cid:int
    name:str
    price:float

    def __str__(self):
        return f"{self.name}的编号是{self.cid},单价是{self.price}"

printer=Commodity(cid=1,name='打印机',price=999)
print(printer)
```

2. 算数运算符

方法名	运算符和表达式	说明
`__add__(self, rhs)`	self + rhs	加法
`__sub__(self, rhs)`	self - rhs	减法
`__mul__(self, rhs)`	self * rhs	乘法
`__truediv__(self, rhs)`	self / rhs	除法
`__floordiv__(self, rhs)`	self // rhs	地板除
`__mod__(self, rhs)`	self % rhs	取模(求余)
`__pow__(self, rhs)`	self ** rhs	幂

演示

```python
class Vector2:
    """
        二维向量
    """

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

    # 决定两个自定义对象相加的逻辑
    def __add__(self, other):
        # 判断传入的数据other是向量还是数值??
        if type(other) == Vector2:
            # 向量+向量
            x = self.x + other.x
            y = self.y + other.y
        else:
            # 向量+数值
            x = self.x + other
            y = self.y + other
        return Vector2(x, y)


pos01 = Vector2(1, 2)
pos02 = Vector2(3, 4)
pos03 = pos01 + pos02  # 相当于pos01.__add__(pos02)
pos04 = pos01 + 10
print(pos03.__dict__) # 4 6
print(pos04.__dict__) # 11 12
```

练习：创建颜色类，数据包含r、g、b、a，实现颜色对象相加

```python
class Color:
    def __init__(self, r, g, b, a=255):
        # 初始化颜色，确保颜色值在有效范围内
        self.r = self._clamp(r)
        self.g = self._clamp(g)
        self.b = self._clamp(b)
        self.a = self._clamp(a)

    def _clamp(self, value):
        # 确保值在 0 到 255 之间
        return max(0, min(255, value))

    def __add__(self, other):
        if not isinstance(other, Color):
            raise TypeError("Can only add another Color object.")
        # 两个颜色相加，值超出 255 会被截断
```

```
            return Color(
                self.r + other.r,
                self.g + other.g,
                self.b + other.b,
                self.a + other.a
            )

        def __str__(self):
            # 提供可读的字符串表示
            return f"Color(r={self.r}, g={self.g}, b={self.b}, a={self.a})"


    # 测试代码
    color1 = Color(100, 150, 200, 255)
    color2 = Color(50, 100, 60, 128)

    result = color1 + color2
    print(result)  # 输出: Color(r=150, g=250, b=255, a=255)
```

3. 复合运算重写(改自己)

方法名	运算符和复合赋值语句	说明
`__iadd__(self, rhs)`	self += rhs	加法
`__isub__(self, rhs)`	self -= rhs	减法
`__imul__(self, rhs)`	self *= rhs	乘法
`__itruediv__(self, rhs)`	self /= rhs	除法
`__ifloordiv__(self, rhs)`	self //= rhs	地板除
`__imod__(self, rhs)`	self %= rhs	取模(求余)
`__ipow__(self, rhs)`	self **= rhs	幂

> 演示

```
class Vector2:
    """
        二维向量
    """

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

```
        self.y = y

    def __str__(self):
        return "x是:%d,y是:%d" % (self.x, self.y)

    # + 创建新
    def __add__(self, other):
        return Vector2(self.x + other.x, self.y + other.y)

    # += 在原有基础上修改(自定义类属于可变对象)
    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

v01 = Vector2(1, 2)
v02 = Vector2(2, 3)
print(v01 + v02) # 新对象了
print(id(v01))
v01 += v02
print(id(v01)) # 还是原对象了
print(v01)
```

4. 比较运算重写

方法名	运算符和复合赋值语句	说明
`__lt__(self, rhs)`	self < rhs	小于
`__le__(self, rhs)`	self <= rhs	小于等于
`__gt__(self, rhs)`	self > rhs	大于
`__ge__(self, rhs)`	self >= rhs	大于等于
`__eq__(self, rhs)`	self == rhs	等于
`__ne__(self, rhs)`	self != rhs	不等于

> 演示

```
class Vector2:
    """
        二维向量
    """

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

```
        self.y = y

    # 1.决定相同的依据
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
        #return self.__dict__ == other.__dict__

    # 2.决定大小的依据
    def __lt__(self, other):
        return self.x < other.x


v01 = Vector2(1, 1)
v02 = Vector2(1, 1)
print(v01 == v02)  # True 比较两个对象内容(__eq__决定)
print(v01 is v02)  # False 比较两个对象地址

list01 = [
    Vector2(2, 2),
    Vector2(5, 5),
    Vector2(3, 3),
    Vector2(1, 1),
    Vector2(1, 1),
    Vector2(4, 4),
]

# 1.必须重写 eq
# 查找时候
print(v01 == v02) # True
print(Vector2(1, 1) in list01) # True
print(list01.count(Vector2(2, 2))) # 1
list01.remove(Vector2(3, 3))

# 2.必须重写 lt
list01.sort()
print(list01)
for item in list01:
    print(item.__dict__)
```

> 练习：创建颜色列表，调用时实现in、count、max、sort运算。

总结：如果一个内置函数底层用到了比较运算符，我们需要重写比较运算

3.3.2 重写自定义函数

(1) 子类实现了父类中相同的方法（方法名、参数），在调用该方法时，实际执行的是子类的方法。

(2) 快捷键：重写ctrl + o

(3) 作用

- 在继承的基础上，体现类型的个性（一个行为有不同的实现）。
- 增强程序灵活性。

```python
# 设计理论：依赖于抽象的父class，low-level module
class Character:
    def __init__(self, hp=0, atk=0):
        self.hp = hp
        self.atk = atk

    def attack(self, target):
        print("发起攻击")
        target.damage(self.atk)  # 注入其他对象

    def damage(self, value): # 被攻击了后续逻辑
        self.hp -= value
        if self.hp <= 0:
            self.death()

    def death(self):
        pass

class Player(Character):
    def damage(self, value): # 被攻击了
        print("碎屏")
        super().damage(value)

    def death(self):
        print("充值")

class Enemy(Character):
    def damage(self, value): # 被攻击了
        print("头顶爆字")
        super().damage(value)

    def death(self):
        print("加分")


# --------------测试--------------
p = Player(200, 50)
e = Enemy(100, 10)
p.attack(e)
e.attack(p)
```

> 练习：以面向对象思想，描述下列情景：
>
> 情景：手雷爆炸，可能伤害敌人(头顶爆字)或者玩家(碎屏)。
>
> 变化：还可能伤害房子、树、鸭子....

> 要求：增加新事物，不影响手雷.
>
> 画出架构设计图

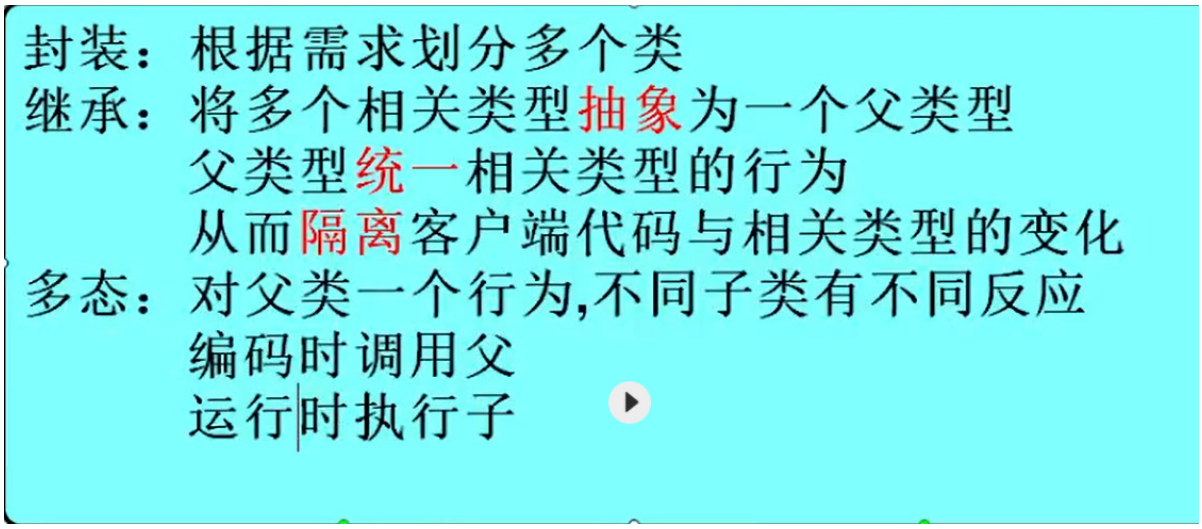
新手雷

手雷.attack(对象)

对象.damage(self.atk)

>

3.4 封装继承多态总结



1.Open-Close Principle理论，即对扩展开放、对修改封闭

- **使用抽象和接口**：在设计时通过抽象类或接口定义功能，具体实现可以在后续扩展中添加。
- **使用继承和多态**：通过继承已有类并扩展其功能，而不修改原有类的代码。

> 练习：创建图形管理器
>
> -- 存储多种图形（圆形、矩形....）
>
> -- 提供计算总面积的方法.
>
> 要求：增加新图形，不影响图形管理器.

```
from abc import ABC, abstractmethod
import math

# 1. 定义抽象基类，表示通用的图形
class Shape(ABC):
    @abstractmethod
    def area(self):
        """计算图形面积
        强制子类实现特定方法：通过定义抽象方法，可以确保所有子类都必须提供这些方法的具体实现
```

```
        当多个类需要遵循相同的接口时，可以使用抽象基类来定义接口
        无需修改这个基类了，后面的subclass直接overwrite就好
        """
        pass

# 2. 创建具体的图形类
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

# 3. 图形管理器类
class ShapeManager:
    def __init__(self):
        self.shapes = []

    def add_shape(self, shape):
        if isinstance(shape, Shape):
            self.shapes.append(shape)
        else:
            raise TypeError("Only objects of type Shape can be added")

    def total_area(self):
        return sum(shape.area() for shape in self.shapes)

# 4. 测试代码

if __name__ == "__main__":
    manager = ShapeManager()

    # 添加各种图形
    manager.add_shape(Circle(5))
    manager.add_shape(Rectangle(4, 6))
    manager.add_shape(Triangle(3, 4))
```

```
62
63      print(f"总面积: {manager.total_area():.2f}")
```

抽象类，接口可以为纯抽象类

- 抽象类一般为基类，可以有一些通用的具体方法, 只能单一继承抽象类
- 接口一般为一个功能，完全抽象方法，多重继承时用

```python
from abc import ABC, abstractmethod

class Flyable(ABC):
    @abstractmethod
    def fly(self):
        pass

class Swimmable(ABC):
    @abstractmethod
    def swim(self):
        pass

class Duck(Flyable, Swimmable):
    def fly(self):
        print("Duck is flying.")

    def swim(self):
        print("Duck is swimming.")

duck = Duck()
duck.fly()   # 输出: Duck is flying.
duck.swim()  # 输出: Duck is swimming.
```

2.Liskov Substitution Principle理论

- 在使用基类的地方，可以用子类来替代
- 当子类的行为与基类不一致时，需要重新设计类层次，避免强行继承导致的问题，比如子类没有这个行为方法

```python
class Bird:  # 更加原始的基类
    pass

class FlyingBird(Bird):
    def fly(self):
        return "I'm flying!"

class Sparrow(FlyingBird):
    def fly(self):
        return "Sparrow is flying!"

class Penguin(Bird):
```

```
    def swim(self):
        return "Penguin is swimming!"

def let_bird_fly(bird: FlyingBird):
    print(bird.fly())

# 使用 FlyingBird 子类
sparrow = Sparrow()
let_bird_fly(sparrow)    # 输出: Sparrow is flying!

# Penguin 不会传递给 let_bird_fly 函数，因此不会出现异常
penguin = Penguin()
print(penguin.swim())    # 输出: Penguin is swimming!
```

3.多态：dependency inversion理论

- 高层模块不依赖于低层模块，两者都依赖于抽象。
- 抽象不依赖于具体实现，具体实现依赖于抽象。

```
from abc import ABC, abstractmethod

# 定义抽象接口
class Database(ABC):
    @abstractmethod
    def connect(self):
        pass

# 具体实现 MySQLDatabase
class MySQLDatabase(Database):
    def connect(self):
        print("Connecting to MySQL database...")

# 具体实现 PostgreSQLDatabase
class PostgreSQLDatabase(Database):
    def connect(self):
        print("Connecting to PostgreSQL database...")

# UserService 依赖于抽象接口 Database
class UserService:
    def __init__(self, database: Database):
        self.database = database

    def get_user(self):
        self.database.connect()
        print("Fetching user data...")

# 使用 UserService，并指定不同的数据库实现
mysql_service = UserService(MySQLDatabase())
mysql_service.get_user()
```

```
31
32  postgres_service = UserService(PostgreSQLDatabase())
33  postgres_service.get_user()
```

4.MVC练习

4.1 学生信息管理系统

4.1.1 需求

实现对学生信息的增加、删除、修改和查询。

4.1.2 分析

界面可能使用控制台，也可能使用Web等等。

(1) 识别对象：界面视图类　逻辑控制类　数据模型类

(2) 分配职责：

-- 界面视图类：负责处理界面逻辑，比如显示菜单，获取输入，显示结果等。

-- 逻辑控制类：负责存储学生信息，处理业务逻辑。比如添加、删除等

-- 数据模型类：定义需要处理的数据类型。比如学生信息。

(3) 建立交互：

界面视图对象 <----> 数据模型对象 <----> 逻辑控制对象

4.1.3 设计

(1) 数据模型类：StudentModel

-- 数据：编号 id,姓名 name,年龄 age,成绩 score

(2) 逻辑控制类：StudentManagerController

-- 数据：学生列表 __stu_list

-- 行为：获取列表 stu_list,添加学生 add_student，删除学生remove_student，修改学生update_student，根据成绩排序order_by_score。

(3) 界面视图类：StudentManagerView

-- 数据：逻辑控制对象

-- 行为：显示菜单__display_menu，选择菜单项__select_menu_item，入口逻辑main，

输入学生__input_students，输出学生__output_students，删除学生__delete_student，

修改学生信息__modify_student

4.2 书籍管理系统

```
"""
MVC架构  ==》 设计模式
分层:  各司其职
M层  Model 模型层  数据  存储 访问 验证 操作
V层  View  视图层  展示信息
C层  Controller 控制器 视图层和模型层的中介  存储和计算等核心逻辑

餐厅
接待   厨师   传菜员
"""

import sys


# M层  造数据
class BookModel:
    def __init__(self, name="", isbn=""):
        self.name = name
        self.isbn = isbn

    # 这里也可以有打印方法


# V层
class BookView:
    def __init__(self):
        self.controller = BookController()

    def display_menu(self):
        print("按1键添加书籍信息")
        print("按2键显示书籍信息")
        print("按3键删除书籍信息")
        print("按4键修改书籍信息")
        print("按5键退出系统")

    def select_menu(self):
        number = input("请输入数字: ")
        if number == "1":
            print("添加书籍")
            self.input_book()
        elif number == "2":
            print("显示书籍")
            self.display_book()
        elif number == "3":
            print("删除书籍")
            self.delete_book()
        elif number == "4":
```

```
            print("修改书籍")
            self.modfiy_book()
        elif number == "5":
            print("退出系统")
            sys.exit()

    def input_book(self):
        # 实例化M层获取书籍对象
        model = BookModel(
            input("请输入书籍名称: "),
            input("请输入书籍isbn: ")
        )

        # 把对象存储在list_book中 ==》C层的列表
        # 弊端:  每次都用新对象使用新列表
        # 期望:每次都能使用旧的对象  ==》 构造函数中去实例化
        # controller = BookController()
        # controller.list_book.append(model)
        # print(controller.list_book)

        # 存储动作 ==》 业务逻辑  ==》 不应该出现在V层，而是在C层
        # self.controller.list_book.append(model)
        # print(self.controller.list_book)

        # V负责显示，C负责存储等业务逻辑
        self.controller.add_book(model)
        print(f"self{self.__dict__}")
        print(f"self.controller{self.controller.__dict__}")
        print(f"list_book{self.controller.list_book[0].__dict__}")

    def display_book(self):

        for item in self.controller.list_book:
            print(f"书名: {item.name}, isbn为{item.isbn}")

    def delete_book(self):
        name = input("请输入要删除的书籍名称: ")
        if self.controller.remove_book(name):
            print("删除成功~")
        else:
            print("删除失败~")

    def modfiy_book(self):
        name = input("请输入要修改的书的名称: ")
        newModel = BookModel(
            isbn=input("请输入isbn号: ")
        )
        if self.controller.update_book(name, newModel):
            print("修改成功~")
        else:
            print("修改失败~")

```

```python
# C层 存储数据
class BookController:
    def __init__(self):
        self.list_book = []

    def add_book(self, bookModelObj):
        self.list_book.append(bookModelObj)
        print(self.list_book)

    def remove_book(self, name):
        for i in range(len(self.list_book)):
            if self.list_book[i].name == name:
                del self.list_book[i]
                return True
        return False

    def update_book(self, name, newBookModelObj):
        print(newBookModelObj.__dict__)
        for item in self.list_book:
            if item.name == name:
                item.isbn = newBookModelObj.isbn
                return True
        return False


view = BookView()
while True:
    view.display_menu()
    view.select_menu()

"""
假设 对象 用 {}  代替 数据结构如下:
view = {
    controller: {
        list_book: [
            {name: "活着", isbn: 122},
            {name: "不活着", isbn: 666},
            {name: "还是活着吧", isbn: 999},
        ]
    }
}
"""
```

4.3 电影信息管理系统

```
"""
MVC架构
电影信息管理系统
    一、录入电影信息
        view:显示菜单、选择菜单、录入信息
        model:封装电影名称、主演、类型、指数
        controller:电影列表、添加信息
    二、显示电影
        view:打印列表中的元素
        model:定义变量的显示格式
    三、删除电影
       view:录入电影名称,显示成败
       controller:在列表中移除元素
    四、修改电影
       view:录入电影名称,新数据(电影名称、主演、类型、指数),显示成败
       controller:在列表中更新元素
"""

# M层 数据封装
import sys

class MovieModel:
    def __init__(self, name="", actor="", type="", index=0):
        self.name = name
        self.actor = actor
        self.type = type
        self.index = index

    # 也可以在M层来显示电影的信息
    # def display(self):
    #     print(f"电影：{self.name}，主演是：{self.actor}，类型是：{self.type}，热映指数：
{self.index}")


# V层 信息显示
class MovieView:
    def __init__(self):
        self.controller = MovieController()

    def display_menu(self):
        print("按1键添加电影信息")
        print("按2键显示电影信息")
        print("按3键删除电影信息")
        print("按4键修改电影信息")
        print("按5键退出系统")

    def select_menu(self):
        number = input("请输入数字：")
        if number == "1":
            print("添加电影")
```

```
            self.input_movie()
        elif number == "2":
            print("显示电影")
            self.display_movie()
        elif number == "3":
            print("删除电影")
            self.delete_movie()
        elif number == "4":
            print("修改电影")
            self.modfiy_movie()
        elif number == "5":
            print("退出系统")
            sys.exit()

    def input_movie(self):
        # 跨类调用:  每次用新的
        model = MovieModel(
            input("请输入电影名: "),
            input("请输入主演: "),
            input("请输入类型: "),
            int(input("请输入指数: ")),
        )
        # 跨类调用:  每次用旧的
        self.controller.add_movie(model)

    def display_movie(self):
        for item in self.controller.list_movie:
            print(f"电影: {item.name}, 主演是: {item.actor}, 类型是: {item.type}, 热映指数: {item.index}")
            # item.display()

    def delete_movie(self):
        name = input("请输入删除的电影名称: ")
        if self.controller.remove_movie(name):
            print("亲~橙啦! ")
        else:
            print("亲~失败啦! ")

    def modfiy_movie(self):
        name = input("请输入要修改的电影名称: ")
        newModel = MovieModel(
            input("请输入新的电影名: "),
            input("请输入主演: "),
            input("请输入类型: "),
            int(input("请输入指数: "))
        )
        if self.controller.update_movie(name, newModel):
            print("亲!  橙啦~")
        else:
            print("亲!  完啦~")


```

```python
# C层 核心业务逻辑
class MovieController:
    def __init__(self):
        self.list_movie = []

    def add_movie(self, newModel):
        self.list_movie.append(newModel)

    def remove_movie(self, name):
        for i in range(len(self.list_movie)):
            if self.list_movie[i].name == name:
                del self.list_movie[i]
                return True
        return False

    def update_movie(self, name, newModel):
        for item in self.list_movie:
            if item.name == name:
                item.__dict__ = newModel.__dict__
                return True
        return False


view = MovieView()

while True:
    view.display_menu()
    view.select_menu()
```

4.4 餐厅信息管理系统(优化)

```
"""
餐厅信息管理系统
    一、录入餐厅信息
        view:显示菜单、选择菜单、录入信息
        model:封装名称,城市,点评人数,人均消费
        controller:餐厅列表、添加信息
    二、显示餐厅
        view:打印列表中的元素
        model:定义变量的显示格式
    三、删除餐厅
       view:录入地区,显示成败
       controller:在列表中移除元素
    四、修改餐厅
       view:录入旧地区,新数据(名称,城市,点评人数,人均消费),显示成败
       controller:在列表中更新元素
    要求使用封装的思想
"""

```

```python
# M层
import sys


class RestaurantModel:
    def __init__(self, name, city, count, consume):
        self.name = name
        self.city = city
        self.count = count
        self.consume = consume


# V层
class RestaurantView:
    def __init__(self):
        self.__controller = RestaurantController()

    # 公开的方法
    def main(self):
        while True:
            self.__display_menu()
            self.__select_menu()

    def __display_menu(self):
        print("按1键添加餐厅信息")
        print("按2键显示餐厅信息")
        print("按3键删除餐厅信息")
        print("按4键修改餐厅信息")
        print("按5键退出系统")

    def __select_menu(self):
        number = input("请输入服务数字: ")
        if number == "1":
            print("添加")
            self.__input_rest()
        elif number == "2":
            print("显示")
            self.__display_rest()
        elif number == "3":
            print("删除")
            self.__delete_rest()
        elif number == "4":
            print("修改")
            self.__modfiy_rest()
        elif number == "5":
            print("退出系统~")
            sys.exit()
        else:
            print("无此项服务~")

    def __input_rest(self):
        model = RestaurantModel(
```

```
                input("请输入餐厅名称："),
                input("请输入餐厅城市："),
                int(input("请输入餐厅点评人数：")),
                float(input("请输入餐厅人均消费："))
            )
            # 列表  ==》 存储  ==》C层
            self.__controller.add_rest(model)

        def __display_rest(self):
            for item in self.__controller.list_rest:
                print(vars(item))
                print(f"{item.name}餐厅所的城市是{item.city}，有{item.count}人点评，人均消费{item.consume}")

        def __delete_rest(self):
            name = input("请输入删除的餐厅名称：")
            if self.__controller.remove_rest(name):
                print("删除成功~")
            else:
                print("删除失败~")

        def __modfiy_rest(self):
            name = input("请输入删除的餐厅名称：")
            model = RestaurantModel(
                input("请输入新的餐厅名称："),
                input("请输入新的餐厅城市："),
                int(input("请输入餐厅新的点评人数：")),
                float(input("请输入餐厅新的人均消费："))
            )
            if self.__controller.update_rest(name, model):
                print("修改成功~")
            else:
                print("修改失败~")


    # C层
    class RestaurantController:
        def __init__(self):
            self.list_rest = []

        def add_rest(self, newModel):
            self.list_rest.append(newModel)
            print(self.list_rest)

        def remove_rest(self, name):
            for item in self.list_rest:
                if item.name == name:
                    self.list_rest.remove(item)
                    return True
            return False

        def update_rest(self, name, model):
```

```
        for item in self.list_rest:
            if item.name == name:
                item.__dict__ = model.__dict__ # 关键：属性复制一下给他
                return True
        return False


# 程序开始的地方
view = RestaurantView()
view.main()
```

4.5 博客和评论系统

```
"""
基于MVC的博客管理系统
可以完成博客的创建、查看（单个）、编辑、删除、显示所有博客列表操作，同时也可以给某篇博客添加评论

1. M层
   POST文章类
   文章id  标题 内容 作者 评论信息(列表)

   Comment 评论类
   评论id  内容   评论者

2. V层
   run()  调用后进入功能选择页面
          1. 创建              create_post
          2. 查看（单个）       view_post
          3. 编辑              edit_post
          4. 删除              delete_post
          5. 添加评论          add_comment
          6. 显示所有博客列表   list_posts
          7.esc退出

3. C层
    创建              create_post
    编辑              edit_post
    删除              delete_post
    添加评论          add_comment
"""

# M层  封装数据
import sys


class PostModel:
    # 类属性
    next_id = 1
```

```
36
37      def __init__(self, title, content, author):
38          self.id = PostModel.next_id
39          PostModel.next_id += 1
40          self.title = title
41          self.content = content
42          self.author = author
43          self.comments = []
44
45
46  class CommentModel:
47      # 类属性
48      next_id = 1
49
50      def __init__(self, content, reviewer):
51          self.id = CommentModel.next_id
52          CommentModel.next_id += 1
53          self.content = content
54          self.reviewer = reviewer
55
56
57  # V层  输入输出
58  class BlogView:
59      def __init__(self):
60          self.__controller = BlogController()
61
62          # 字典映射  数字和方法名的映射
63          self.menu_dict = {
64              "1": self.__input_post,
65              "2": self.__display_post,
66              "3": self.__modify_post,
67              "4": self.__delete_post,
68              "5": self.__input_comment,
69              "6": self.__list_posts,
70              "7": sys.exit
71          }
72          self.list_lastid = []
73
74      def run(self):
75          while True:
76              self.__display_menu()
77              self.__select_menu()
78
79      # 显示菜单
80      def __display_menu(self):
81          print("按1键添加博客信息")
82          print("按2键显示博客信息")
83          print("按3键修改博客信息")
84          print("按4键删除博客信息")
85          print("按5键添加博客评论")
86          print("按6键显示所有博客信息")
87          print("按7键退出系统")
```

```
88
89      # 选择菜单
90      def __select_menu(self):
91          number = input("请输入数字: ")
92
93          if number in self.menu_dict:
94              self.menu_dict[number]()
95          else:
96              print("暂无此项服务~")
97
98      # 添加
99      def __input_post(self):
100         postModel = PostModel(
101             input("请输入标题: "),
102             input("请输入内容: "),
103             input("请输入作者: ")
104         )
105         self.__controller.add_post(postModel)
106
107     # 显示
108     def __display_post(self):
109         # 如何真正的实现通过id值来查看对象的文章的信息
110         id = int(input("请输入要查看的文章ID: "))
111
112         for item in self.__controller.list_posts:
113             if item.id == id:
114                 print(f"ID为: {item.id}的文章标题是:{item.title},内容为: {item.content},作者
    是: {item.author}")
115
116     # 修改
117     def __modify_post(self):
118         id = int(input("请输入要修改的文章ID: "))
119         post_dict = {
120             "title": input("请输入标题: "),
121             "content": input("请输入内容: "),
122             "author": input("请输入作者: ")
123         }
124         if self.__controller.update_post(id, post_dict):
125             print(f"修改ID为: {id}文章成功")
126         else:
127             print("修改失败")
128
129     # 删除
130     def __delete_post(self):
131         id = int(input("请输入要修改的文章ID: "))
132         if self.__controller.remove_post(id):
133             print(f"删除id为{id}的文章成功")
134         else:
135             print("删除失败")
136
137     # 添加评论
138     def __input_comment(self):
```

```
            pid = int(input("输入想评论的文章的id: "))

            for item in self.__controller.list_posts:
                if item.id == pid:
                    if pid in self.list_lastid: # 评论过
                        CommentModel.next_id = self.pid
                        model = CommentModel(
                            input("请输入评论的内容: "),
                            input("请输入评论的作者: "),
                        )
                        self.__controller.add_comment(pid, model)
                    else:
                        CommentModel.next_id = 1
                        model = CommentModel(
                            input("请输入评论的内容: "),
                            input("请输入评论的作者: "),
                        )
                        self.__controller.add_comment(pid, model)
                        self.list_lastid.append(pid)
                        self.pid = CommentModel.next_id


        # 显示所有信息
        def __list_posts(self):
            for item in self.__controller.list_posts:
                print(f"ID:{item.id}文章的标题为: {item.title},内容是: {item.content},作者是: {item.author}")
                for comment in item.comments:
                    print(f"ID为{item.id}文章, 评论第{comment.id}条: 内容是: {comment.content},作者是: {comment.reviewer}")


    # C层 核心业务逻辑
    class BlogController:
        def __init__(self):
            self.list_posts = []

        def add_post(self, model):
            self.list_posts.append(model)

            for item in self.list_posts:
                print(item.__dict__)

        def update_post(self, id, post_dict):
            for item in self.list_posts:
                if item.id == id:
                    if post_dict['title'].strip():
                        item.title = post_dict['title']

                    if post_dict['content'].strip():
                        item.content = post_dict['content']

```

```
                if post_dict['author'].strip():
                    item.author = post_dict['author']
                return True
        return False

    def remove_post(self, id):
        for item in self.list_posts:
            if item.id == id:
                self.list_posts.remove(item)
                return True
        return False

    def add_comment(self, pid, model):
        for item in self.list_posts:

            if item.id == pid:
                item.comments.append(model)
                for comment in item.comments:
                    print(comment.__dict__)
                    # return True
            # return False
            print(item.__dict__)


# 梦开始的地方
view = BlogView()
view.run()
```