1. 概述

1.1 面向过程

(1) 定义：分析出解决问题的步骤，然后逐步实现。

例如：婚礼筹办

-- 请柬（选照片、措词、制作）

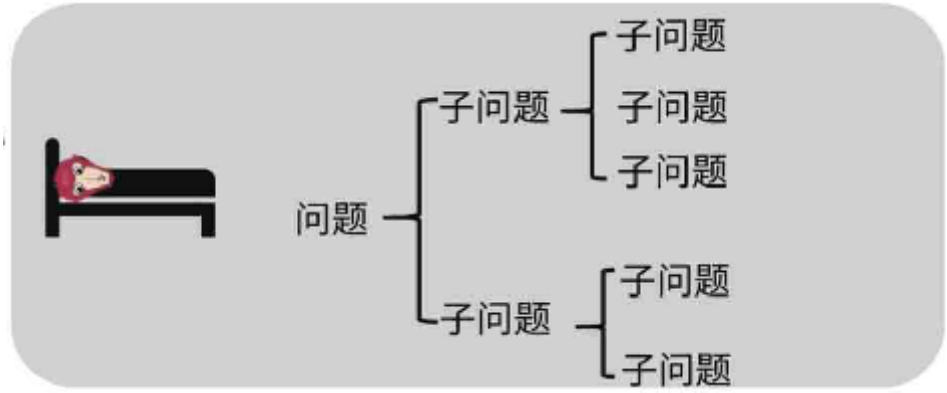
-- 宴席（场地、找厨师、准备桌椅餐具、计划菜品、购买食材）

-- 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。

(4) 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

(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。

2. 类和对象

(1) 抽象：从具体事物中抽离出共性、本质，舍弃个别、非本质过程。

具体

抽象

数据：品牌、价格、颜色..

行为：通话..

(2) 类：一个抽象的概念，即生活中的"类别"。

(2) 对象：类的具体实例，即归属于某个类别的"个体"。

(3) 类是创建对象的"模板"。

-- 数据成员：名词类型的状态。

-- 方法成员：动词类型的行为。

2.1 语法

2.1.1 定义类

(1) 代码

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

     方法成员
```

(2) 说明

-- 类名所有单词首字母大写.

-- init 也叫构造函数，创建对象时被调用，也可以省略。

-- 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的形参传递

(3) 演示

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

```
shang_er.play()# 通过对象地址调用方法,会自动传递对象地址.
# play(shanger)
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

a. 定义：对象.变量名

b. 调用：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.

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

b. 通常在构造函数(__init_)中创建

```
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。

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)
    # 不建议通过类名访问实例方法
```

(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。

(5) 演示

```python
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

```python
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)

# 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]
```

2.2.3 跨类调用

```
# 写法1：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")
```

```
# 写法2：在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")
```

```
# 方式3：通过参数传递
# 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～")

lz = Person("老张")
benz = Car()
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

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).

练习3：以面向对象思想,描述下列情景.

玩家攻击敌人,敌人受伤(根据玩家攻击力，减少敌人的血量).

练习4：以面向对象思想,描述下列情景.

张无忌教赵敏九阳神功

赵敏教张无忌玉女心经

张无忌工作挣了5000元

赵敏工作挣了10000元

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
    # 不建议通过对象访问类变量
```

(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。

(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)
  # 不建议通过对象访问类方法
```

(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。

(3) 作用：操作类变量。

(4) 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
```

2.4 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)
    # 不建议通过对象访问静态方法
```

(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

(4) 作用：定义常用的工具函数。

3. 三大特征

3.1 封装

3.1.1 数据角度

(1) 定义：将一些基本数据类型复合成一个自定义类型。

(2) 优势：

-- 将数据与对数据的操作相关联。

-- 代码可读性更高（类是对象的模板）。

3.1.2 行为角度

(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，可以通过__dict__属性查看。

-- 演示

```
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()
```

(4) 属性@property：

-- 作用：保护实例变量

-- 定义：

```
@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：

```
对象.属性名 = 数据
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

0 -- 120 0 -- 200 100 -- 100

-- 三种形式：

```
# 1. 读取属性
class MyClass:
    def __init__(self,data):
        self.data = data

    @property
    def data(self):
        return self.__data

    @data.setter
    def data(self, value):
        self.__data = value

m01 = MyClass(10)
print(m01.data)
```

```python
# 2. 只读属性
class MyClass:
    def __init__(self):
        self.__data = 10

    @property
    def data(self):
        return self.__data


m01 = MyClass()
# m01.data = 20# AttributeError: can't set attribute
print(m01.data)
```

```python
# 3. 只写属性
class MyClass:
    def __init__(self, data):
        self.data = data

    # data = property()

    # @data.setter
    # def data(self, value):
    #     self.__data = value

    def data(self, value):
        self.__data = value

    data = property(fset=data)


m01 = MyClass(10)
print(m01.data)  # AttributeError: unreadable attribute
m01.data = 20
```

3.1.3 案例:信息管理系统

3.1.3.1 需求

实现对学生信息的增加、删除、修改和查询。

3.1.3.2 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

3.1.3.3 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__manager

-- 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

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) 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

(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。

(3) 演示

```
# 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

# 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

# 是的关系
# 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
```

(4) 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。

(3) 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[子覆盖了父方法,好像它不存在]
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(wk.name)
print(wk.score)
```

(4) 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。

(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）。

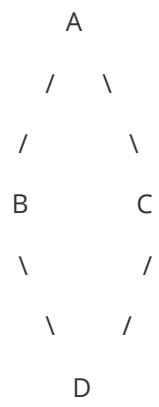
-- Object类：任何类都直接或间接继承自 object 类。

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



(3) 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()

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()
```

3.3 多态

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的成员。我们可以在自定义类中进行重写，从而改变其行为。

(2) __str__ 函数：将对象转换为字符串(对人友好的)

-- 演示

```
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)
# <__main__.Person object at 0x7fbabfbc3e48>
# 悟空的年龄是26
print(wk)
# message = wk.__str__()
# print(message)

```

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

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

class Enemy:
  def __init__(self, name="", atk=0, hp=0):
    self.name = name
    self.atk = atk
    self.hp = hp
```

(3) 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

```
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)

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

v01 = Vector2(1, 2)
v02 = Vector2(2, 3)
print(v01 + v02)  # v01.__add__(v02)
```

-- 练习：创建颜色类，数据包含r、g、b、a，实现颜色对象相加。

(4) 复合运算符重载

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(id(v01))
v01 += v02
print(id(v01))
print(v01)
```

-- 练习：创建颜色类，数据包含r、g、b、a，实现颜色对象累加。

(5) 比较运算重载

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

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

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

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

    # 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),
]

# 必须重写 eq
print(Vector2(5, 5) in list01)
print(list01.count(Vector2(1, 1)))

# 必须重写 lt
list01.sort()
print(list01)
```

-- 练习：创建颜色列表，实现in、count、index、max、sort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：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。

练习1：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

练习2：创建图形管理器

-- 记录多种图形（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.

测试：

创建图形管理器，存储多个图形对象。

通过图形管理器，调用计算总面积方法.