# 面向对象 Object Oriented

## 概述

### 面向过程

1. 分析出解决问题的步骤，然后逐步实现。

例如：婚礼筹办

-- 发请柬（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）

-- 婚礼仪式（定婚礼仪式流程、请主持人）

1. 公式：程序 = 算法 + 数据结构
2. 优点：所有环节、细节自己掌控。
3. 缺点：考虑所有细节，工作量大。

### 面向对象

1. 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）

1. 公式：程序 = 对象 + 交互
2. 优点
3. 思想层面：

-- 更接近于人的思维方式。

-- 有利于梳理归纳、分析解决问题。

(2) 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。

## 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，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。

1. 类与类行为不同，对象与对象数据不同。

例如：(1)学生student是一个类，具有姓名，年龄等数据；

具有学习study，工作work等行为。

对象：悟空同学，28岁。

八戒同学，29岁。

(2)车 car是一个类，具有类型type，速度speed等数据；

启动start，停止stop，行驶run等行为。

对象：宝马，180.

比亚迪，100.

(3)狗dog是一个类，具有类型，姓名，重量weight等数据，

拉臭臭shit，玩play等行为。

对象：拉布拉多，米咻。

金毛，赵金多。

(4)字符串str是一个类，”abc”是一个对象。

### 语法

#### 定义类

1. 代码

class 类名:

“””文档说明”””

def \_init\_(self,参数列表):

self.实例变量 = 参数

方法成员

1. 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。

#### 创建对象(实例化)

变量 = 构造函数 (参数列表)

### 实例成员

#### 实例变量

1. 语法
2. 定义：对象.变量名
3. 调用：对象.变量名
4. 说明
5. 首次通过对象赋值为创建，再次赋值为修改.

w01 = Wife()

w01.name = “丽丽”

w01.name = “莉莉”

1. 通常在构造函数(\_init\_)中创建。

w01 = Wife(“丽丽”,24)

print(w01.name)

1. 每个对象存储一份，通过对象地址访问。
2. 作用：描述所有对象的共有数据。
3. \_\_dict\_\_：对象的属性，用于存储自身实例变量的字典。

#### 实例方法

1. 语法

(1) 定义： def 方法名称(self, 参数列表):

方法体

(2) 调用： 对象地址.实例方法名(参数列表)

不建议通过类名访问实例方法

2. 说明

(1) 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"。

(2) 无论创建多少对象，方法只有一份，并且被所有对象共享。

3. 作用：表示对象行为。

*"""*

*面向对象：考虑问题，从对象的角度出发．*

*类：模板　　抽象*

*对象：具体*

*"""*

**class** Wife:

*"""*

*老婆*

*"""*

*# 1.数据成员 姓名　年龄　性别　．．．*

**def** \_\_init\_\_(self, name, age, sex):

*# self "自己"，调用当前方法的对象*

print(id(self))

self.name = name

self.age = age

self.sex = sex

*# 2.方法成员 做饭　．．．*

**def** cooking(self):

print(id(self))

print(self.name + **"做饭"**)

*# 创建对象(实例化)*

*# 调用* *\_\_init\_\_(self,name,age,sex) 方法*

w01 = Wife(**"丽丽"**, 21, **"女"**)

print(id(w01))

*# 调用对象的方法* *w01 将自身传入方法*

w01.cooking()

w02 = Wife(**"芳芳"**, 22, **"男"**)

w02.cooking()

print(id(w02))

*# 在内存中，方法只有一份．而对象有多份．*

*"""*

*(1)学生student是一个类，具有姓名，年龄等数据；*

*具有学习study，工作work等行为。*

*对象：悟空同学，28岁。*

*八戒同学，29岁。*

*"""*

**class** Student:

*"""*

*学生类*

*"""*

**def** \_\_init\_\_(self,name,age):

self.name = name

self.age = age

**def** study(self):

print(str(self.age) + **"学习"**)

**def** work(self):

print(self.name+**"工作"**)

*# s01 悟空对象的地址*

s01 = Student(**"悟空"**,28)

s02 = Student(**"八戒"**,29)

*# 通过对象地址，调用对象方法，会自动传递对象地址．*

s01.study()

s02.work()

*# s01 = Student("悟空",28)*

*# s02 = s01*

*# s01.name = "孙悟空"*

*# print(s02.name) # ?*

s01 = Student(**"悟空"**,28)

s02 = s01

s01 = Student(**"八戒"**,28)

s01.name = **"孙悟空"**

print(s02.name) *# ? 悟空*

### 类成员

#### 类变量

1. 语法
2. 定义：在类中，方法外定义变量。

class 类名:

变量名 = 表达式

1. 调用：类名.变量名

不建议通过对象访问类变量

1. 说明

-- 存储在类中。

-- 只有一份，被所有对象共享。

1. 作用：描述所有对象的共有数据。

#### 类方法

1. 语法
2. 定义：

@classmethod

def 方法名称(cls,参数列表):

方法体

1. 调用：类名.方法名(参数列表)

不建议通过对象访问类方法

1. 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。

1. 作用：操作类变量。

*"""*

*类成员*

*练习:exercise01*

*"""*

**class** ICBC:

*"""*

*工商银行*

*"""*

*# 类变量 相当于被大家共享的"饮水机",*

moneys = 9999999

*# 类方法*

@classmethod

**def** print\_total\_moneys(cls):

*# print(ICBC.moneys)*

print(**"总行金额："**,cls.moneys)

*# 实例方法*

**def** \_\_init\_\_(self,name,money):

*# 实例变量：相当于每个人的"杯子"*

self.money = money

self.name = name

*# 从总行中，扣除当前支行的现金*

ICBC.moneys -= money

i01 = ICBC(**"广渠门支行"**,100000)

*# 调用类变量*

*# print("总行金额：",ICBC.moneys)*

*# 调用类方法，此时会自动传递类名进入方法*

ICBC.print\_total\_moneys()

i02 = ICBC(**"磁器口支行"**,100000)

*# print("总行金额：",ICBC.moneys)*

ICBC.print\_total\_moneys()

### 静态方法

1. 语法
2. 定义：

@staticmethod

def 方法名称(参数列表):

方法体

1. 调用：类名.方法名(参数列表)

不建议通过对象访问静态方法

1. 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

1. 作用：定义常用的工具函数。

*"""*

*静态方法*

*操作二维列表工具*

*"""*

**class** Vector2:

*"""*

*向量*

*"""*

**def** \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

*# 将函数转移到类中，就是静态方法．*

@staticmethod

**def** right():

**return** Vector2(0, 1)

@staticmethod

**def** up():

**return** Vector2(-1, 0)

@staticmethod

**def** left():

**return** Vector2(0, -1)

@staticmethod

**def** down():

**return** Vector2(1, 0)

@staticmethod

**def** right\_up():

**return** Vector2(-1, 1)

**class** DoubleListHelper:

*"""*

*二维列表助手类*

*定义：在开发过程中，所有对二维列表的常用操作．*

*"""*

@staticmethod

**def** get\_elements(list\_target, v\_pos, v\_dir, count):

result = []

**for** i **in** range(count):

v\_pos.x += v\_dir.x

v\_pos.y += v\_dir.y

result.append(list\_target[v\_pos.x][v\_pos.y])

**return** result

*# 测试．．．．．．．．．．．．．*

list01 = [

[**"00"**, **"01"**, **"02"**, **"03"**],

[**"10"**, **"11"**, **"12"**, **"13"**],

[**"20"**, **"21"**, **"22"**, **"23"**],

]

*# 10 向右　　　　　３　　　　－－> 11 12 13*

re01 = DoubleListHelper.get\_elements(list01, Vector2(1, 0), Vector2.right(), 3)

print(re01)

*# 练习１：在二维列表中，获取23位置，向左，３个元素．*

re02 = DoubleListHelper.get\_elements(list01, Vector2(2, 3), Vector2.left(), 3)

*# 练习２：在二维列表中，获取02位置，向下，２个元素．*

re02 = DoubleListHelper.get\_elements(list01, Vector2(0, 2), Vector2.down(), 2)

*# 练习３：在二维列表中，获取20位置，右上，２个元素．*

re02 = DoubleListHelper.get\_elements(list01, Vector2(2, 0), Vector2.right\_up(), 2)

print(re02)

## 封装

### 定义

1. 数据角度讲，将一些基本数据类型复合成一个自定义类型。
2. 行为角度讲，向类外提供功能，隐藏实现的细节。
3. **设计角度讲**：

（1）分而治之

-- 将一个大的需求分解为许多类，每个类处理一个独立的功能。

-- 拆分好处：便于分工，便于复用，可扩展性强。

(2) 封装变化

-- 变化的地方独立封装，避免影响其他类。

(3) 高 内 聚

-- 类中各个方法都在完成一项任务(单一职责的类)。

(4) 低 耦 合

-- 类与类的关联性与依赖度要低(每个类独立)，让一个类的改变，尽少影响其他类。

[例如：硬件高度集成化，又要可插拔]

最高的内聚莫过于类中仅包含1个方法，将会导致高内聚高耦合。

最低的耦合莫过于类中包含所有方法，将会导致低耦合低内聚。

### 作用

1. 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。
2. 松散耦合，降低了程序各部分之间的依赖性。
3. 数据和操作相关联，方法操作的是自己的数据。

### 私有成员

1. 作用：无需向类外提供的成员，可以通过私有化进行屏蔽。
2. 做法：命名使用双下划线开头。
3. 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\_属性或dir函数查看。

### \_\_slots\_\_

1. 作用：限定一个类创建的实例只能有固定的实例变量，不能再额外添加。
2. 语法：

在类中定义

\_\_slots\_\_ = (“变量名1”,”变量名2”…..)

1. 说明：含有\_\_slots\_\_属性的类所创建的对象没有\_\_dict\_\_属性, 即此实例不用字典来存储对象的实例属性。
2. 优点：访止用户因错写属性的名称而发生程序错误。
3. 缺点：丧失了动态语言可以在运行时为对象添加变量的灵活性。

*"""*

*封装数据优势：*

*１．符合人类思考方式*

*２．将数据与对数据的操作封装起来。*

*使用方法封装变量*

*"""*

*# class Wife01:*

*# def \_\_init\_\_(self, name, age):*

*# self.name = name*

*# # 缺点：缺乏对象数据的封装，外界可以随意赋值．*

*# self.age = age*

*#*

*#*

*# w01 = Wife01("芳芳", 26)*

*# w02 = Wife01("铁锤", 86)*

*# print(w02.age)*

*# w02.age = 87*

*# print(w02.age)*

*# 注意：通过两个方法，读写私有变量．*

*# 练习：定义敌人类(姓名，攻击力，攻击速度(０－１０)，血量(０－－１００))*

**class** Wife02:

**def** \_\_init\_\_(self, name = **""**, age = 0):

self.set\_name(name)

*# 私有成员：障眼法(解释器会改变双下划线开头的变量名)*

*# self.\_\_age = age*

self.set\_age(age)

**def** get\_name(self):

**return** self.\_\_name

**def** set\_name(self,value):

self.\_\_name = value

**def** get\_age(self):

**return** self.\_\_age

**def** set\_age(self,value):

**if** 20 <= value <= 30:

self.\_\_age = value

**else**:

print(**"我不要"**)

w01 = Wife02(**"铁锤"**,86)

*# 找不到双下划线开头的数据*

*# print(w01.\_\_age)*

*# 通过下划线　＋　类名 可以访问双下划线开头的数据*

*# print(w01.\_Wife02\_\_age)*

w01.set\_age(50)

*# print(w01.get\_age())*

print(w01.\_\_dict\_\_)

*"""*

*\_\_slots\_\_ 属性*

*"""*

**class** SkillData:

*# 限制当前类,创建的对象,只能具有的实例变量.*

\_\_slots\_\_ = (**"\_\_name"**)

**def** \_\_init\_\_(self, name):

self.name = name

@property

**def** name(self):

**return** self.\_\_name

@name.setter

**def** name(self, value):

self.\_\_name = value

s01 = SkillData(**"技能名称"**)

*# s01.name = "降龙十八掌"*

print(s01.name)

*# 为当前对象,添加实例变量*

*# s01.time = 5*

*# print(s01.time)*

*# print(s01.\_\_dict\_\_) # 因为使用了\_\_slots\_\_属性,所以不是使用\_\_dict\_\_*

### 属性@property

公开的实例变量，缺少逻辑验证。私有的实例变量与两个公开的方法相结合，又使调用者的操作略显复杂。而属性可以将两个方法的使用方式像操作变量一样方便。

1. 定义：

@property

def name(self):

return self.\_\_name

@name.setter

def name(self, name):

self.\_\_name = name

1. 调用：

对象.属性名 = 数据

变量 = 对象.属性名

1. 说明：

-- 通常两个公开的属性，保护一个私有的变量。

-- @property 负责读取，@属性名.setter 负责写入

-- 只写：属性名= property(None, 写入方法名)

*"""*

*使用属性封装变量*

*"""*

*# 练习：修改Enemy类，使用属性封装变量*

**class** Wife:

**def** \_\_init\_\_(self, name=**""**, age=0):

self.name = name *# 调用* *@name.setter 修饰的方法*

self.age = age *# 调用* *@age.setter 修饰的方法*

@property *# 拦截读取变量的操作*

**def** name(self): *# get\_name()*

**return** self.\_\_name

@name.setter *# 拦截写入变量的操作*

**def** name(self, value): *# set\_name()*

self.\_\_name = value

@property

**def** age(self):

**return** self.\_\_age

@age.setter

**def** age(self, value):

**if** 20 <= value <= 30:

self.\_\_age = value

**else**:

self.\_\_age = 0

print(**"我不要"**)

w01 = Wife(**"铁锤"**, 26)

print(w01.name)

print(w01.age)

*# 属性*

**class** D:

**def** \_\_init\_\_(self,name):

*# 写入操作*

self.name = name

*# 拦截读取操作*

*# 本质:创建property对象,name 存储对象地址.*

*# 注意: 创建对象时,会指定读取方法*

*# 相当于: name = property(读取方法,None)*

@property

**def** name(self):

*# 逻辑判断*

**return** self.\_\_name

*# 拦截写入操作*

@name.setter *# 本质:name.setter(写入方法)*

**def** name(self,value):

*# 逻辑判断*

self.\_\_name = value

d01 = D(**"张三"**)

d01.name = **"老三"** *# 写入操作*

print(d01.name) *# 读取操作*

*# 只读属性*

**class** E:

**def** \_\_init\_\_(self, a):

self.\_\_a = a

*# 拦截读取操作*

@property

**def** a(self):

*# 逻辑判断*

**return** self.\_\_a

e01 = E(100)

*# e01.a = 10 # 错误*

print(e01.a)

*# 属性本质*

**class** F:

**def** \_\_init\_\_(self,a):

*# 写入操作*

self.a = a

**def** get\_a(self):

print(**"读取变量喽"**)

**return** self.\_\_a

**def** set\_a(self, value):

print(**"设置变量喽"**)

self.\_\_a = value

*# 拦截对变量a的读写操作*

*# 创建property对象,a存储的是对象地址.*

*# 注意:创建对象时,需要传递读写方法*

a = property(get\_a,set\_a)

f01 = F(10)

f01.a = 200 *# 写入*

print(f01.a) *# 读取*

*# 只写属性*

**class** G:

**def** \_\_init\_\_(self,a):

*# 写入操作*

self.a = a

**def** set\_a(self, value):

print(**"设置变量喽"**)

self.\_\_a = value

*# 拦截对变量a的写入操作*

a = property(**None**,set\_a)

g01 = G(100)

g01.a = 200

*# print(g01.a) # 错误*

print(g01.\_G\_\_a)

### 案例:学生管理系统



#### 需求

实现对学生信息的增加、删除、修改和查询。

#### 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。

1. 建立交互：

界面视图对象 <----> 数据模型对象 <----> 逻辑控制对象

#### 设计

数据模型类：StudentModel

数据：编号 id,姓名 name,年龄 age,成绩 score

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。

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，按成绩输出学生\_\_output\_student\_by\_score

*"""*

*学生管理系统*

*"""*

*# 练习1：创建学生数据模型类*

*# 练习2：创建逻辑控制类*

*# 数据：学生列表*

*# 行为：获取列表的只读属性stu\_list,添加学生方法*

**class** StudentModel:

*"""*

*学生数据模型类*

*"""*

**def** \_\_init\_\_(self,name=**''**,age=0,score=0,id=0):

*"""*

*创建学生对象*

**:param** *id: 编号*

**:param** *name: 姓名*

**:param** *age: 年龄*

**:param** *score: 分数*

*"""*

self.id=id

self.name=name

self.age=age

self.score=score

*# self.id+=1*

@property

**def** id(self):

**return** self.\_\_id

@id.setter

**def** id(self, value):

self.\_\_id = value

@property

**def** name(self):

**return** self.\_\_name

@name.setter

**def** name(self,value):

self.\_\_name=value

@property

**def** age(self):

**return** self.\_\_age

@age.setter

**def** age(self, value):

self.\_\_age = value

@property

**def** score(self):

**return** self.\_\_score

@score.setter

**def** score(self, value):

self.\_\_score = value

**class** StudentManagerController:

**def** \_\_init\_\_(self):

self.\_\_stu\_list=[]

*# 只读*

@property

**def** stu\_list(self):

**return** self.\_\_stu\_list

*# 添加学生信息*

**def** add\_student(self,student):

**if** len(self.\_\_stu\_list)>0:

student.id=self.\_\_stu\_list[-1].id+1

**else**:

student.id=1

self.\_\_stu\_list.append(student)

**def** delete\_student(self):

id=int(input(**"请输入要删除的学生学号："**))

**for** i **in** range(len(self.stu\_list)):

**if** self.stu\_list[i].id==id:

**del** self.stu\_list[i]

**break**

**def** updata\_student(self,stu):

**for** item **in** self.\_\_stu\_list:

**if** item.id==stu.id:

item.name=stu.name

item.age = stu.age

item.score = stu.score

**return True**

**return False**

*# s01=StudentModel("小明",16,88)*

*# s02=StudentModel("小黑",15,68)*

*# stu=StudentManagerController()*

*# stu.add\_student(s01)*

*# stu.add\_student(s02)*

*#*

*# for item in stu.stu\_list:*

*# print(item.id,item.name,item.age,item.score)*

**class** StudentManagerView:

**def** \_\_init\_\_(self):

self.\_\_managers = StudentManagerController()

**def** \_\_dispay\_stu\_list(self,targe\_list):

**for** item **in** targe\_list:

print(item.id,item.name,item.age,item.score)

**def** \_\_output\_student\_by\_score(self):

temp\_list=self.\_\_managers.stu\_list[:]

**for** i **in** range(len(temp\_list)-1):

**for** n **in** range(i+1,len(temp\_list)):

**if** temp\_list[i].score>temp\_list[n].score:

temp\_list[i],temp\_list[n]=temp\_list[n],temp\_list[i]

self.\_\_dispay\_stu\_list(temp\_list)

**def** \_\_input\_student(self):

stu=StudentModel()

stu.name=input(**"请输入学生姓名："**)

stu.age=int(input(**"请输入学生年龄："**))

stu.score=float(input(**"请输入学生成绩："**))

self.\_\_managers.add\_student(stu)

**def** \_\_output\_students(self):

id=int(input(**"请输入要显示的学生学号："**))

*# smc=StudentManagerController()*

*# temp=smc.stu\_list()*

**for** item **in** self.\_\_managers.stu\_list:

**if** item.id==id:

print(item.id,item.name,item.age,item.score)

**def** \_\_mpdify\_student(self):

stu = StudentModel()

stu.id=int(input(**"请输入修改的学生编号："**))

stu.name = input(**"请输入学生姓名："**)

stu.age = int(input(**"请输入学生年龄："**))

stu.score = float(input(**"请输入学生成绩："**))

self.\_\_managers.updata\_student(stu)

**def** \_\_select\_menu(self):

print(**"1.添加学生"**)

print(**"2.显示学生"**)

print(**"3.删除学生"**)

print(**"4.修改学生"**)

print(**"5.按学生成绩从低到高显示学生信息"**)

print(**"6.打印全部学生信息"**)

**def** \_\_select\_menu\_item(self):

number=int(input(**"请选择菜单："**))

**if** number==1:

self.\_\_input\_student()

**elif** number==2:

self.\_\_output\_students()

**elif** number==3:

self.\_\_managers.delete\_student()

**elif** number==4:

self.\_\_mpdify\_student()

**elif** number==5:

self.\_\_output\_student\_by\_score()

**elif** number==6:

self.\_\_dispay\_stu\_list(self.\_\_managers.stu\_list)

**def** main(self):

**while True**:

self.\_\_select\_menu()

self.\_\_select\_menu\_item()

start=StudentManagerView()

start.main()

## 继承

### 语法

1. 代码

class 子类(父类):

def \_\_init\_\_(self,参数列表):

super().\_\_init\_\_(参数列表)

self.自身实例变量 = 参数

1. 说明

-- 子类拥有父类的所有成员。

-- 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，以确保父类属性被正常创建。

*"""*

*继承语法* *-- 数据*

*# 练习: 定义父类--宠物, 数据:名字*

*# 定义子类--狗, 数据:工作*

*# 创建相应对象,画出内存图*

*"""*

**class** Person:

**def** \_\_init\_\_(self,name):

self.name = name

**class** Student(Person):

**def** \_\_init\_\_(self,name,score):

*# 通过函数,调用父类方法*

super().\_\_init\_\_(name)

self.score = score

**class** Teacher(Person):

**def** \_\_init\_\_(self,name,salary):

super().\_\_init\_\_(name)

self.salary = salary

s01 = Student(**"zs"**,100)

print(s01.score)

print(s01.name)

p02 = Person(**"ww"**)

print(p02.name)

### 多继承

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。

同名方法的解析顺序（MRO， Method Resolution Order）:

类自身 --> 父类继承列表（由左至右）--> 再上层父类

A

/ \

/ \

B C

\ /

\ /

D

*"""*

*多继承*

*"""*

**class** A:

**def** m(self):

print(**"a -- m"**)

**class** B(A):

**def** m(self):

super().m()

print(**"b -- m"**)

**class** C(A):

**def** m(self):

super().m()

print(**"c -- m"**)

*# 多继承*

**class** D(C,B):

**def** m(self):

*# 调用父级同名方法,执行顺序(继承列表顺序)*

super().m()*# C*

super().m()

*# 通过类名调用实例方法,传递对象地址.*

*# B.m(self)*

print(**"d -- m"**)

d01 = D()

d01.m()

*# print(D.mro())*

*# d02=B()*

*# d02.m()*

### 定义

1. 重用现有类的功能与概念，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。

-- 事物具有一定的层次、渊源，继承可以统一概念。

例如：公司组织架构

老板

行政中心 营销中心 技术中心

人力资源 行政部 销售部 策划部 研发部 产品部

### 优点

1. 一种代码复用的方式。
2. 以层次化的方式管理类。

### 缺点

耦合度高

### 作用

隔离客户端代码与功能的实现方式。

### 适用性

多个类在概念上是一致的，且需要进行统一的处理。

### 相关概念

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。

### 内置函数

isinstance(obj, class\_or\_tuple)

返回这个对象obj 是否是某个类的对象,或者某些类中的一个类的对象。

*"""*

*继承语法* *-- 方法*

*财产*

*皇位*

*#练习1:定义父类--宠物, 行为:吃*

*# 定义子类--狗, 行为:防守xx*

*# 定义子类--鸟, 行为:飞*

*#创建相应对象,调用相应方法.测试isinstance,issubclass函数*

*"""*

*# 学生 与 老师 在某种概念上是统一的*

*# 学生是 人*

*# 老师是 人*

*#*

**class** Person:

**def** say(self):

print(**"说"**)

**class** Student(Person):

**def** study(self):

print(**"学习"**)

**class** Teacher(Person):

**def** teach(self):

print(**"教"**)

s01 = Student()

s01.study()

*# 可以直接使用父类成员*

s01.say()

p02 = Person()

p02.say()

*# 父类不能使用子类成员*

*# p02.study()*

t03 = Teacher()

t03.teach()

*# 不能使用"兄弟"类成员*

*# t03.study()*

*# 判断一个对象是否"兼容"一个类型*

print(isinstance(s01,Person)) *# True*

print(isinstance(s01,Student)) *# True*

print(isinstance(s01,object)) *# True*

print(isinstance(s01,Teacher)) *# False*

print(isinstance(p02,Student)) *# False*

*# 判断一个类是否"兼容"一个类型*

print(issubclass(Student,Person)) *# True*

print(issubclass(Student,Teacher)) *# False*

print(issubclass(Student,(Teacher,Person))) *# True*

print(issubclass(Student,Student)) *# True*

## 多态

### 定义

父类的同一种动作或者行为，在不同的子类上有不同的实现。

**class** Person:

**def** \_\_init\_\_(self,name):

self.name=name

**def** go\_to(self,vehicle,place):

print(self.name)

vehicle.run(place)

**class** Vehicle:

**def** run(self,place):

**raise** NotImplementedError(**"儿子没有这个方法"**)

**class** Car(Vehicle):

**def** run(self,place):

print(**"开车去%s"**%place)

**class** Train(Vehicle):

**def** run(self,place):

print(**"乘车去%s"**%place)

**class** Plane(Vehicle):

**def** run(self,place):

print(**"坐飞机去%s"**%place)

p01=Person(**'老王'**)

p02=Person(**'老张'**)

c=Car()

t=Train()

p=Plane()

p02.go\_to(c,**'东北'**)

### 作用

1. 继承将相关概念的共性进行抽象，多态在共性的基础上，体现类型的个性化（一个行为有不同的实现）。
2. 增强程序扩展性，体现开闭原则。

### 重写

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，实际调用的是子类的方法。

*"""*

*有若干个图形(圆形,矩形........)*

*每种图形,都可以计算面积.*

*定义图形管理器,记录所有图形,提供计算总面积的方法.*

*要求:增加新的图形,不改变图形管理器代码.*

*"""*

**class** GraphicManager:

**def** \_\_init\_\_(self):

*# 记录所有图形*

self.\_\_graphics = []

**def** add\_graphic(self,g):

**if not** isinstance(g,Graphic):

**return**

self.\_\_graphics.append(g)

**def** get\_total\_area(self):

*"""*

*计算总面积*

**:return***:*

*"""*

total\_area = 0

**for** item **in** self.\_\_graphics:

*# 调用:父类*

*# 执行:子类*

total\_area += item.get\_area()

**return** total\_area

**class** Graphic:

*"""*

*图形*

*"""*

**def** get\_area(self):

**pass**

**class** Circle(Graphic):

*"""*

*圆形*

*"""*

**def** \_\_init\_\_(self,radius):

self.radius = radius

**def** get\_area(self):

**return** self.radius \*\* 2 \* 3.14

**class** Rectangle(Graphic):

*"""*

*矩形*

*"""*

**def** \_\_init\_\_(self, length,width):

self.length = length

self.width = width

**def** get\_area(self):

**return** self.length \* self.width

manager = GraphicManager()

manager.add\_graphic(**"ok"**)

manager.add\_graphic(Rectangle(2,3))

manager.add\_graphic(Circle(5))

*# 加断点,调试*

print(manager.get\_total\_area())

#### 内置可重写函数

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。

\_\_str\_\_函数：将对象转换为字符串(对人友好的)

\_\_repr\_\_函数：将对象转换为字符串(解释器可识别的)

*"""*

*内置可重写函数*

*练习:exercise01.py*

*"""*

**class** Wife:

**def** \_\_init\_\_(self,name,age):

self.name = name

self.age = age

**def** \_\_str\_\_(self):

*# 返回给人看*

**return "奴家叫:%s,年芳:%d"**%(self.name,self.age)

**def** \_\_repr\_\_(self):

*# 返回给解释器看*

**return 'Wife("%s",%d)'**%(self.name,self.age)

w01 = Wife(**"金莲"**,25)

print(w01)*# 将对象转换为字符串*

*# re = eval("1 + 5")*

*# print(re)*

*# w02 = eval('Wife("金莲",25)')*

*# w03 = eval(input(""))*

*#创建了新对象*

w02 = eval(w01.\_\_repr\_\_())

print(w02)

w01.name =**"莲莲"**

print(w02.name)

*# 练习: 重写* *StudentModel 类* *\_\_str\_\_ 方法 与\_\_repr\_\_方法*

*# 创建学生对象,创建学生对象列表.分别print*

**class** StudentModel:

**def** \_\_init\_\_(self, name=**""**, age=0, score=0, id=0):

self.id = id

self.name = name

self.age = age

self.score = score

**def** \_\_str\_\_(self):

*# return*

**return "我的编号是:%d,姓名是:%s,年龄是:%d,成绩是:%d."**%(self.id,self.name,self.age,self.score)

**def** \_\_repr\_\_(self):

**return 'StudentModel("%s",%d,%d)'**%(self.name,self.age,self.score)

s01 = StudentModel(**"zs"**,24,100)

s02 = eval(s01.\_\_repr\_\_())

print(s02)

print(s01)

list01 = [s01,s02]

print(list01)

#### 运算符重载

定义：让自定义的类生成的对象(实例)能够使用运算符进行操作。

##### 算数运算符



##### 反向算数运算符重载



##### 复合运算符重载



##### 比较运算重载



*"""*

*运算符重载*

*"""*

*# print("a" + "b")*

**class** Vector:

*"""*

*向量*

*"""*

**def** \_\_init\_\_(self, x):

self.x = x

**def** \_\_str\_\_(self):

**return "向量的x变量是:%s"**%self.x

*# 对象* *+*

**def** \_\_add\_\_(self, other):

**return** Vector(self.x + other)

*# # + 对象*

*# def \_\_radd\_\_(self, other):*

*# return Vector(self.x + other)*

*#*

*# # 累加:在原有对象基础上进行操作,不创建新对象.*

*# def \_\_iadd\_\_(self, other):*

*# self.x += other*

*# return self*

*#*

*# def \_\_lt\_\_(self, other):*

*# return self.x < other*

v01 = Vector(10)

v02 = v01 + 5

print(v02)

*# # 练习:实现向量类 与 整数 做 减法/乘法运算* *17:25*

*# v03 = 5 + v01*

*# print(v03)*

*# v04 = v01 + v02*

*# print(v04)*

*# 练习:实现整数 与 向量 做 减法/乘法运算*

*# 向量 向量*

*# list01 = [1]*

*# list02 = list01 + [2]*

*# print(list01)*

*# list01 += [2]*

*# print(list01)*

*#*

*# print(id(v01))*

*# # 累加含义:在原有对象基础上,增加新值*

*# v01 += 1*

*# print(v01)*

*# print(id(v01))*

*# # 练习:实现向量类 与 整数 做 累计减法/累计乘法运算*

*# print(v01 < 2)*

## 设计原则

### 开-闭原则（目标、总的指导思想）

**O**pen **C**losed **P**rinciple

对扩展开放，对修改关闭。

增加新功能，不改变原有代码。

### 类的单一职责（一个类的定义）

**S**ingle **R**esponsibility **P**rinciple

一个类有且只有一个改变它的原因。

### 依赖倒置（依赖抽象）

**D**ependency **I**nversion **P**rinciple

客户端代码(调用的类)尽量依赖(使用)抽象的组件。

抽象的是稳定的。实现是多变的。

### 组合复用原则（复用的最佳实践）

Composite Reuse Principle

如果仅仅为了代码复用优先选择组合复用，而非继承复用。

组合的耦合性相对继承低。

### 里氏替换（继承后的重写，指导继承的设计）

**L**iskov **S**ubstitution **P**rinciple

父类出现的地方可以被子类替换，在替换后依然保持原功能。

子类要拥有父类的所有功能。

子类在重写父类方法时，尽量选择扩展重写，防止改变了功能。

### 迪米特法则（类与类交互的原则）

Law of Demeter

不要和陌生人说话。

类与类交互时，在满足功能要求的基础上，传递的数据量越少越好。因为这样可能降低耦合度。

*"""*

*面向对象:考虑问题,要从对象的角度出发.*

*谁?干嘛?*

*主要思想:*

*识别对象/分配职责/建立交互*

*封装变化/隔离变化/执行变化*

*封装 继承 多态*

*特征:*

*封装:*

*1) 数据:将多个基本类型,合成一个自定义类型.*

*优势:复合人类的思考方式*

*比如:学生,向量(x,y)*

*2) 行为:对外提供简单的必要的功能,隐藏实现的细节.*

*优势:模块化开发,简化编程*

*比如:学生管理系统* *-- 逻辑控制器,供界面调用.*

*3) 设计:*

*分而治之:分解需求,让多个类协同完成.*

*封装变化:每个变化点单独做成一个类.*

*-------------------------------*

*高内聚:类的内部处理一个变化点*

*低耦合:类与类的关系,尽量做到互不影响.*

*继承:重用现有类的概念,并在此基础上进行扩展.*

*(子类的共性) (子类相比父类更加具体)*

*作用:隔离客户端代码与实现方式(隔离用与做)*

*比如:交通工具隔离了人与飞机/火车...的变化*

*图形隔离了图形管理器与圆形/矩形...的变化*

*多态:调用父一个方法,执行子类方法,不同实现方式不一样,所以表现形态就不一样.*

*作用:通过重写执行不同变化点*

*比如:人调用交通工具的运输方法,执行的是飞机/火车的运输方法*

*图形管理器调用图形的计算面积方法,执行的是圆形/矩形的计算面积方法.*

*原则:*

*开闭原则:增加新功能,不修改客户端代码.*

*比如:某个技能,增加新影响效果,只要创建新的效果类,不需要修改其他代码.*

*单一职责:每个类有且只有一个改变的原因.*

*比如:技能系统中,每个类的职责明确.*

*依赖倒置:使用抽象(父),而不是用具体(子)*

*比如:技能释放器,调用的是影响,而不是伤害生命/降低防御力..*

*组合复用:使用关联关系,代替继承关系.*

*比如:技能释放器与影响效果,使用了关联关系.*

*里氏替换:父类出现的地方,可以被子类替换*

*替换后,保持原有功能.*

*比如:技能释放器使用影响效果,但可以被所有具体效果替换.*

*迪米特法则:低耦合*

*比如:影响效果的变化,不影响释放器.*

*每种效果之间,互不影响.*

*"""*

*"""*

*继承* *--- 设计思想*

*面向对象设计原则*

*练习:exercise01.py*

*1. 开闭原则*

*开放 关闭*

*对扩展 对修改*

*允许增加新功能 不允许改变(增加/删除/修改)以前的代码*

*2. 依赖倒置(抽象)*

*使用抽象(父类),而不使用具体(子类).*

*"""*

*# 老张开车去东北*

*# 变化:飞机*

*# 火车*

*# 汽车*

*# .......*

**class** Person:

**def** \_\_init\_\_(self, name):

self.name = name

*# def go\_to(self, str\_type, str\_pos):*

*# if str\_type == "汽车":*

*# Car().run(str\_pos)*

*# elif str\_type =="飞机":*

*# Airplane().fly(str\_pos)*

*# # elif xxxxx:*

**def** go\_to(self, vehicle, str\_pos):

*"""*

*写代码期间:*

*使用的是交通工具的,而不是汽车,飞机等*

*所以无需判断具体类型*

*运行期间:*

*传递具体的对象(汽车,飞机)*

**:param** *vehicle:*

**:param** *str\_pos:*

**:return***:*

*"""*

vehicle.transport(str\_pos)

*# class Car:*

*# def run(self, str\_pos):*

*# print("行驶到", str\_pos)*

*#*

*#*

*# class Airplane:*

*# def fly(self, str\_pos):*

*# print("飞到", str\_pos)*

**class** Vehicle:

*"""*

*交通工具*

*"""*

**def** transport(self, str\_pos):

*# 人为创造一个错误()*

**raise** NotImplementedError()

*# print("儿子们,必须有这个方法啊")*

**class** Car(Vehicle):

**def** transport(self, str\_pos):

*# super().transport()*

print(**"行驶到"**, str\_pos)

**class** Airplane(Vehicle):

**def** transport(self, str\_pos):

print(**"飞到"**, str\_pos)

*# ....*

p01 = Person(**"老张"**)

*# p01.go\_to("汽车", "东北")*

p01.go\_to(Car(),**"东北"**)

## 类与类的关系

**泛化：**子类与父类的关系，概念的复用，耦合度最高；

B类泛化A类，意味B类是A类的一种；

做法：B类继承A类

**关联(聚合/组合)：**部分与整体的关系，功能的复用，变化影响一个类；

A与B关联，意味着B是A的一部分；

做法：在A类中包含B类型成员。

**依赖：**合作关系，一种相对松散的协作，变化影响一个方法；

A类依赖B类，意味A类的某些功能靠B类实现；

做法：B类型作为A类中方法的参数，并不是A的成员。

*"""*

*有若干个图形(圆形,矩形........)*

*每种图形,都可以计算面积.*

*定义图形管理器,记录所有图形,提供计算总面积的方法.*

*要求:增加新的图形,不改变图形管理器代码.*

*"""*

**class** GraphicManager:

**def** \_\_init\_\_(self):

*# 记录所有图形*

self.\_\_graphics = []

**def** add\_graphic(self,g):

**if not** isinstance(g,Graphic):

**return**

self.\_\_graphics.append(g)

**def** get\_total\_area(self):

*"""*

*计算总面积*

**:return***:*

*"""*

total\_area = 0

**for** item **in** self.\_\_graphics:

*# 调用:父类*

*# 执行:子类*

total\_area += item.get\_area()

**return** total\_area

**class** Graphic:

*"""*

*图形*

*"""*

**def** get\_area(self):

**pass**

**class** Circle(Graphic):

*"""*

*圆形*

*"""*

**def** \_\_init\_\_(self,radius):

self.radius = radius

**def** get\_area(self):

**return** self.radius \*\* 2 \* 3.14

**class** Rectangle(Graphic):

*"""*

*矩形*

*"""*

**def** \_\_init\_\_(self, length,width):

self.length = length

self.width = width

**def** get\_area(self):

**return** self.length \* self.width

manager = GraphicManager()

manager.add\_graphic(**"ok"**)

manager.add\_graphic(Rectangle(2,3))

manager.add\_graphic(Circle(5))

*# 加断点,调试*

print(manager.get\_total\_area())

**class** Employee:

*"""*

*员工类*

*"""*

**def** \_\_init\_\_(self, name,job):

self.name = name

*# 成员变量的类型是岗位*

self.job = job # 类作为属性

**def** calculate\_salary(self):

*"""*

*使用岗位,计算薪资.*

**:return***:*

*"""*

**return** self.job.get\_salary()

**class** Job:

*"""*

*岗位*

*"""*

**def** \_\_init\_\_(self,salary):

self.base\_salary = salary

**def** get\_salary(self):

**return** self.base\_salary

**class** Programmer(Job):

*"""*

*程序员*

*"""*

**def** \_\_init\_\_(self,salary, bonus):

super().\_\_init\_\_(salary)

self.bonus = bonus

**def** get\_salary(self):

*# return self.base\_salary + self.bonus*

*# 扩展重写*

**return** super().get\_salary() + self.bonus

**class** Salesmen(Job):

*"""*

*销售员*

*"""*

**def** \_\_init\_\_(self,salary, sale\_value):

super().\_\_init\_\_(salary)

self.sale\_value = sale\_value

**def** get\_salary(self):

**return** super().get\_salary() + self.sale\_value \* 0.05

*#练习:老王转岗*

*# 销售* *--> 程序员*

*# 继承关系*

*# lw = Salesmen("老王",3000,500)*

*# lw = Programmer("老王",8000,100000)*

*# 重新创建新对象,替换引用.好比是开除"老王",招聘新"老王"*

*# 要求:对象部分改变,而不是全部改变.*

lw = Employee(**"老王"**,Salesmen(3000,500))

print(lw.calculate\_salary())

*# 转岗*

lw.job = Programmer(8000,100000)

print(lw.calculate\_salary())