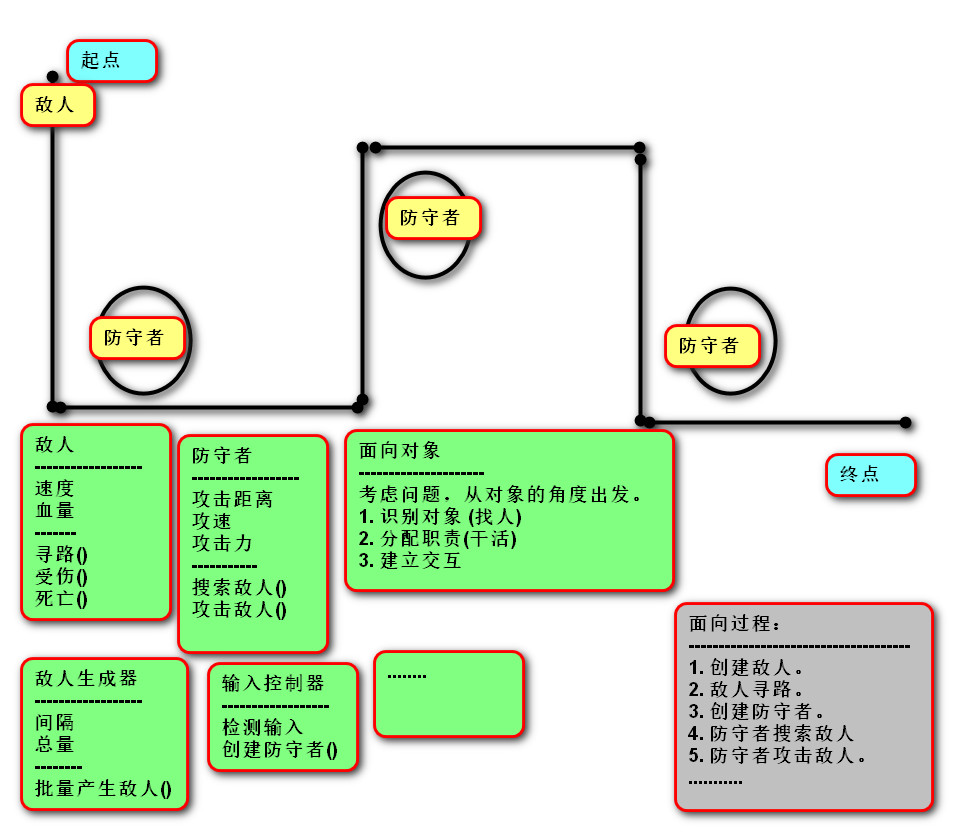
# 面向对象 Object Oriented

## 概述

面向对象概述内存图:



### 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。

#### 创建类与对象案例

##### 案例一:

**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*

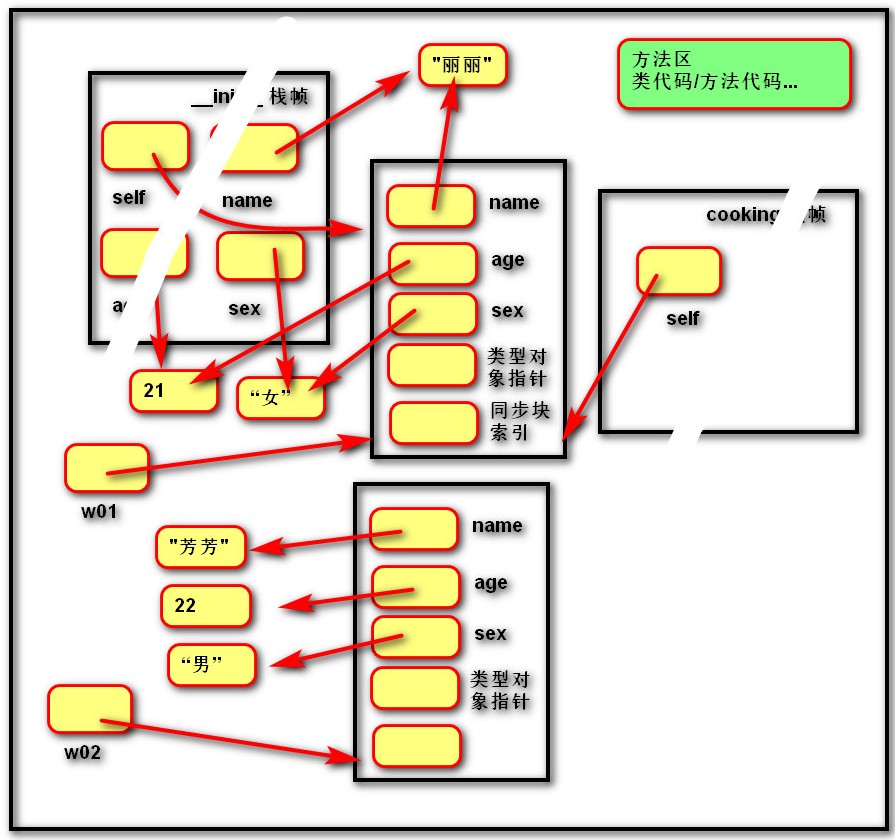
w01.cooking()

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．*

**内存图:**

##### 案例二:

*(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) # ? 孙悟空*

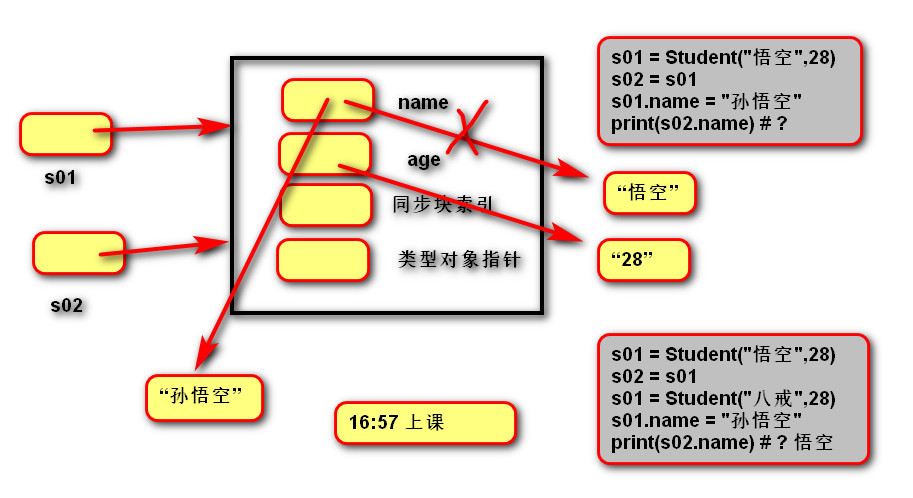
s01 = Student(**"悟空"**,28)

s02 = s01

s01 = Student(**"八戒"**,28) *#* 表示又另开了一块内存区域

s01.name = **"孙悟空"**

print(s02.name) *# ? 悟空*

**内存图:**

##### 案例三:

**class** Enemy:

*"""*

*敌人*

*"""*

**def** \_\_init\_\_(self,name=**""**,hp=0,atk=0,atk\_speed=0): *#名字,血量,攻击力,攻击速度*

self.name = name

self.hp = hp

self.atk = atk

self.atk\_speed = atk\_speed

**def** print\_self(self): *#将敌人列表输出*

print(self.name,self.hp,self.atk,self.atk\_speed)

*# 1. 在控制台中输入３个敌人，存入列表．*

*# 2. 将敌人列表输出（调用print\_self）到控制台*

e01 = Enemy(**"zs"**,100,1000,5)

list\_enemy = []

**for** i **in** range(3):

e = Enemy()

e.name = input(**"请输入姓名："**)

e.hp = int(input(**"请输入血量："**))

e.atk = float(input(**"请输入攻击力："**))

e.atk\_speed = float(input(**"请输入攻击速度："**))

list\_enemy.append(e)

**for** item **in** list\_enemy:

item.print\_self()

##### 案例四:

*# 练习３：定义函数，在敌人列表中，根据姓名查找敌人对象．*

*# e01 = Enemy("zs",100,10,2)*

*# e02 = Enemy("ls",200,3,5)*

*# e03 = Enemy("ww",300,30,5)*

*#*

*# 提示:list\_enemy = [e01,e02,e03]*

**def** get\_enemy\_for\_name(list\_enemy,name):

*# 遍历敌人列列表*

**for** item **in** list\_enemy:

*# 如果有指定名称的敌人对象*

**if** item.name == name:

*# 则返回对象地址*

**return** item

list01 = [

Enemy(**"zs"**,100,10,2),

Enemy(**"ls"**, 200, 5, 3),

Enemy(**"ww"**,300,8,5)

]

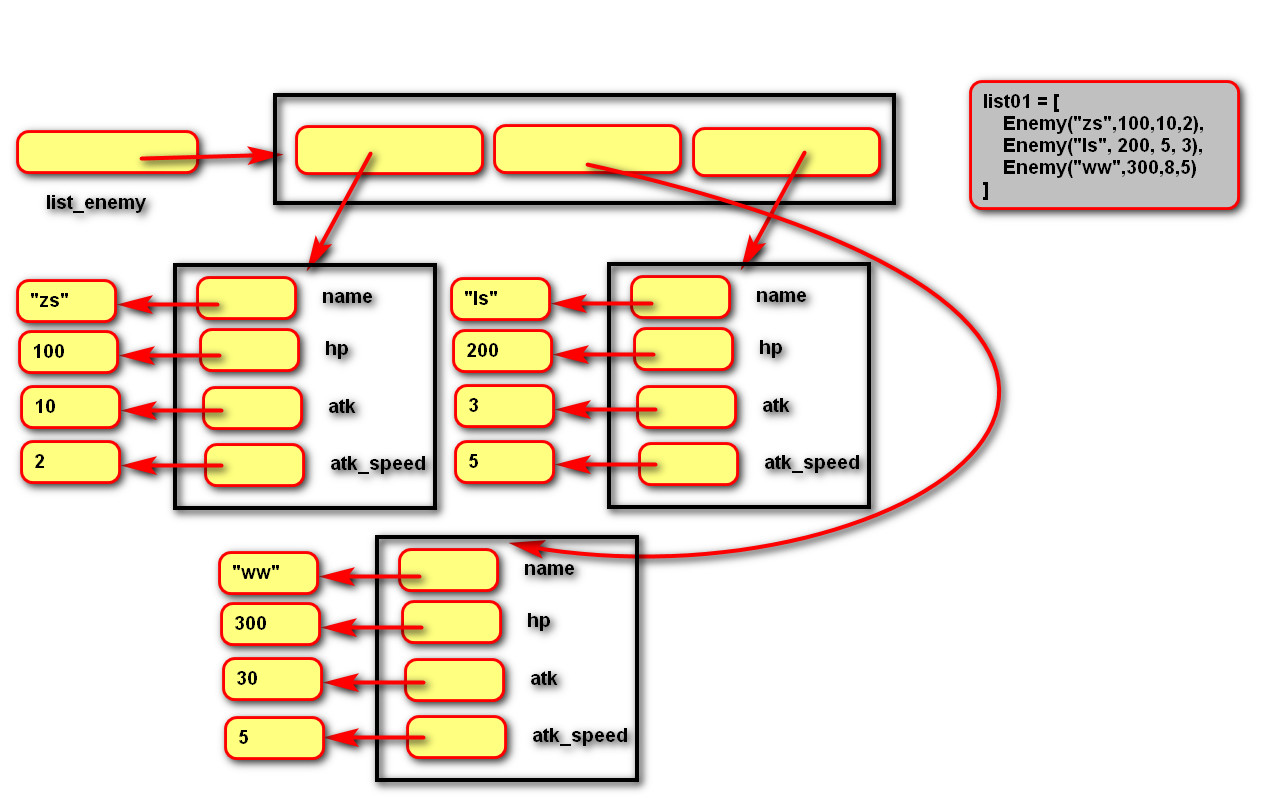
re = get\_enemy\_for\_name(list01,**"ls"**)

**if** re != **None**:

re.print\_self()

**else**:

print(**"没有找到"**)

**内存图:**

## 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。

##### 实例变量案例:

**class** Wife02:

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

self.name = name

w01 = Wife02(**"丽丽"**) *#* 首次创建

w01.name = **"莉莉" *#*** 再次修改

print(w01.\_\_dict\_\_)*# 此时实例变量是：{'name': '莉莉'}*

print(w01.name)

w01 = Wife02(**"丽丽"**)

print(w01.\_\_dict\_\_)*# 此时实例变量是：{'name': '丽丽'}*

*# 创建实例成员，可以不在\_\_init\_\_中．（在实际项目中，仍然会在\_\_init\_\_方法中）*

**class** Wife03:

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

self.name = name

**def** fun01(self):

self.age = 10

print(**"fun01有执行"**)

w01 = Wife03(**"丽丽"**)

*# 通过对象调用实例方法，会自动传递对象地址．*

w01.fun01() *# fun01有执行*

*# 通过类名调用实例方法，*

Wife03.fun01(w01) *# fun01有执行*

print(w01.age) *# 10*

#### 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。

### 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**(自动)。

-- 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。

#### 类成员案例:

**代码：**

**class** ICBC:

*"""*

*工商银行*

*"""*

*# 类变量(被所有对象共享,在类中，方法外定义变量。)*

moneys = 9999999 *#工商总行的钱*

*# 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) ----->不建议通过对象访问类变量*

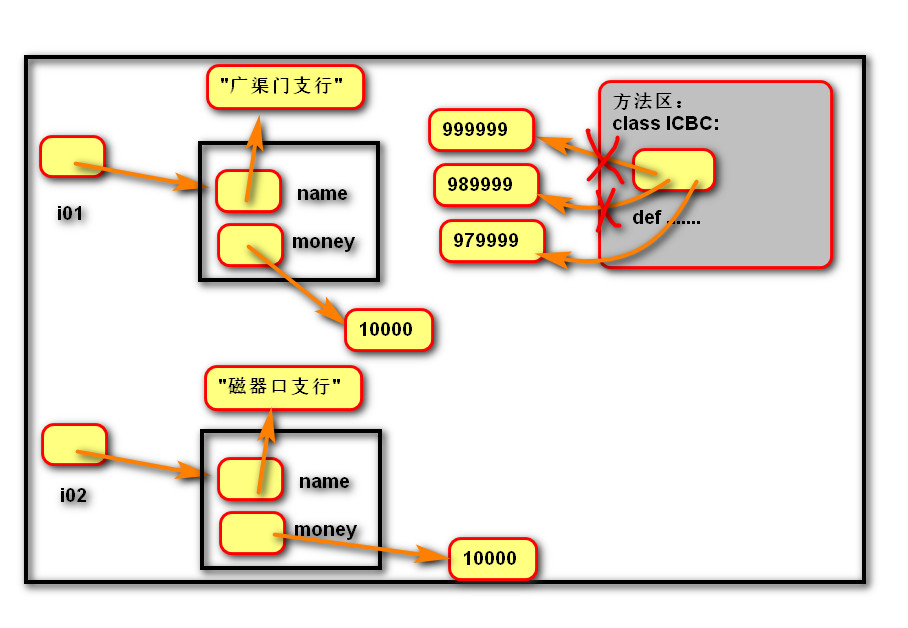
*# 调用类方法(类名.变量名)，此时会自动传递类名进入方法*

ICBC.print\_total\_moneys() *# 9899999*

i02 = ICBC(**"磁器口支行"**,100000)

*# print("总行金额：",ICBC.moneys)*

ICBC.print\_total\_moneys() *# 9799999*

**内存图：**

### 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。

## 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。(每个变化点单独做成一个类)

(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。（隔离）

[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。

**分而治之：**

案例一：

*"""*

*需求: 老张开去车东北.*

*分而治之* *-- 分解*

*变化点*

*练习:exercise01*

*"""*

#需求: 老张开去车东北.

class Person:

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

self.name = name

def go\_to(self, type, str\_pos):

type.run(str\_pos)

class Car:

def run(self, str\_pos):

print("行驶到", str\_pos)

p01 = Person("老张")

c01 = Car()

p01.go\_to(c01, "东北")

案例二：

# 练习: 小明在招商银行取钱.

class Person:

def \_\_init\_\_(self, name, money=0):

self.name = name

self.money = money

class Bank:

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

self.name = name

self.total\_money = money

# 考虑:取钱逻辑,应该由银行决定.所以取钱方法,定义在了银行.

def draw\_money(self, person, value):

if self.total\_money >= value:

self.total\_money -= value

person.money += value

print(person.name, "取钱成功")

else:

print("取钱失败")

p01 = Person("小明")

b01 = Bank("招商银行", 100000)

b01.draw\_money(p01, 10000000)

b01 = Bank("招商银行", 100000)

b01.draw\_money(p01, 10000000)

### 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。

#### 案例：

*"""*

*\_\_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)

#### 属性封装案例:

**class** Enemy:

*"""*

*敌人*

*"""*

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

self.name=name

self.hp=hp

@property *#读取name的变量*

**def** name(self):

**return** self.\_\_name

@name.setter *#写入name的变量*

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

self.\_\_name=value

@property

**def** hp(self):

**return** self.\_\_hp

@hp.setter

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

**if** 0<= value <= 100:

self.\_\_hp=value

**else**:

self.\_\_hp=0 *#当赋予值不在范围内,赋予它一个默认值(不然会报错)*

print(value,**"当前赋值不在范围内"**)

e01=Enemy(**"哥斯拉"**,200)

print(e01.name,e01.hp)

### 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。

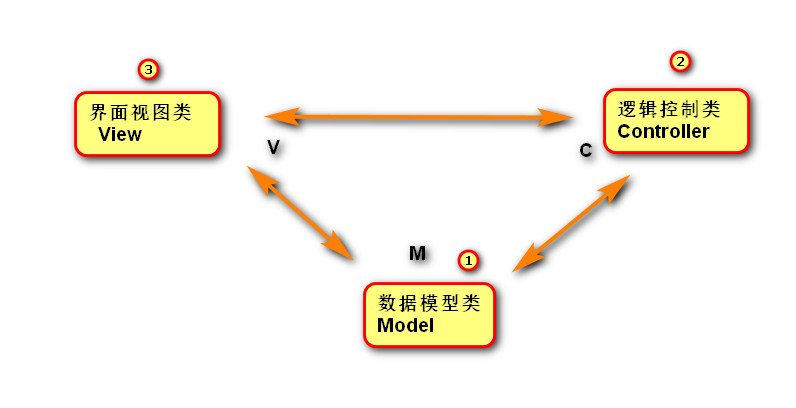
逻辑控制类：负责存储学生信息，处理业务逻辑。比如添加、删除等

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

内存图：



#### 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

#### 代码：

#### *"""*

*学生管理器系统*

*"""*

**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: 成绩*

*"""*

self.id = id

self.name = name

self.age = age

self.score = score

@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:

*"""*

*学生逻辑控制器*

*"""*

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

self.\_\_list\_stu = []

@property

**def** list\_stu(self):

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

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

*"""*

*添加新学生*

**:param** *stu: 需要添加的学生对象*

*"""*

stu.id = self.\_\_generate\_id()

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

**def** \_\_generate\_id(self):

*# 生成编号的需求:新编号,比上次添加的对象编号多1.*

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

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

*# else:*

*# id = 1*

*# return id*

**return** self.\_\_list\_stu[-1].id + 1 **if** len(self.\_\_list\_stu) > 0 **else** 1

**def** order\_by\_score(self):

*"""*

*根据成绩升叙排列*

**:return***:*

*"""*

*# 创建新列表(目的:不影响原有列表)*

new\_list = self.\_\_list\_stu[:]

**for** r **in** range(len(new\_list)-1):

**for** c **in** range(r+1,len(new\_list)):

**if** new\_list[r].score > new\_list[c].score:

new\_list[r],new\_list[c] = new\_list[c],new\_list[r]

**return** new\_list

**def** remove\_student(self,id):

*"""*

*删除学生*

**:param** *id:*

**:return***:*

*"""*

**for** stu **in** self.list\_stu:

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

self.list\_stu.remove(stu)

**return True** *# 表示删除成功*

**return False** *# 表示删除失败*

*# controller = StudentManagerController()*

*# controller.add\_student(StudentModel("zs",18,85))*

*# controller.add\_student(StudentModel("zs",18,85))*

*# for item in controller.list\_stu:*

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

**class** StudentManagerView:

*"""*

*界面视图类*

*"""*

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

*# 创建逻辑控制器对象*

self.\_\_manager = StudentManagerController()

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

*# 1. 在控制台中录入学生信息,存成学生对象StudentModel.*

stu = StudentModel()

stu.name = input(**"请输入姓名:"**)

stu.age = int(input(**"请输入年龄:"**))

stu.score = int(input(**"请输入成绩:"**))

*# 2. 调用逻辑控制器的add\_student方法*

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

print(self.\_\_manager)

**def** \_\_output\_students(self,list\_target):

*"""*

*显示学生列表信息*

**:return***:*

*"""*

*# for stu in self.\_\_manager.list\_stu:*

**for** stu **in** list\_target:

print(**"%d -- %s -- %d -- %d"**%(stu.id,stu.name,stu.age,stu.score))

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

*"""*

*根据成绩显示所有学生信息*

**:return***:*

*"""*

list\_target = self.\_\_manager.order\_by\_score()

self.\_\_output\_students(list\_target)

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

id = int(input(**"请输入需要删除的学生编号:"**))

**if** self.\_\_manager.remove\_student(id):

print(**"删除成功"**)

**else**:

print(**"删除失败"**)

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

*"""*

*显示菜单*

**:return***:*

*"""*

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) 按照成绩降序排列"**)

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

*"""*

*选择菜单*

**:return***:*

*"""*

number = input(**"请输入选项:"**)

**if** number == **"1"**:

self.\_\_input\_students()

**elif** number == **"2"**:

self.\_\_output\_students(self.\_\_manager.list\_stu)

**elif** number == **"3"**:

self.\_\_delete\_student()

**elif** number == **"4"**:

**pass**

**elif** number == **"5"**:

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

**def** main(self):

*"""*

*界面入口方法*

**:return***:*

*"""*

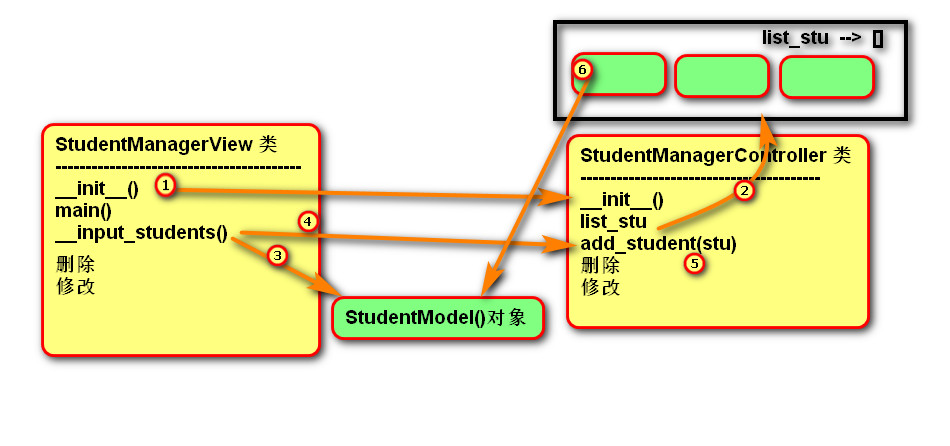
**while True**:

self.\_\_display\_menu()

self.\_\_select\_menu()

view = StudentManagerView()

view.main()

内存图：

## 继承

### 语法

1. 代码

class 子类(父类):

def \_\_init\_\_(self,参数列表):

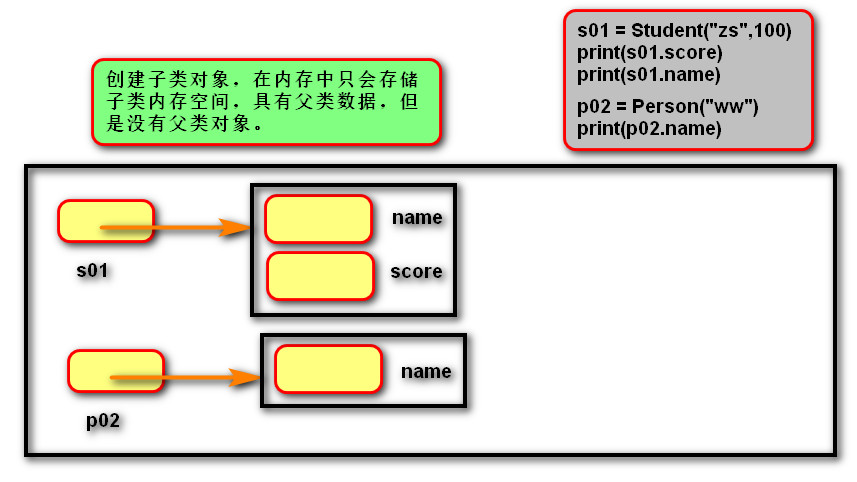
super().\_\_init\_\_(参数列表)

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。



#### 继承案例：

*# 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*

#### 继承语法：

**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

案例：

*"""*

*多继承*

*"""*

**class** A:

**def** m(self):

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

**class** B(A):

**def** m(self):

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

**class** C(A):

**def** m(self):

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*

*# 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())

### 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。

## 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。

　　（调用父类，执行子类）

### 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。

（通过重写执行不同的变化点）

### 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。

#### 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)

#### 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



## 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。

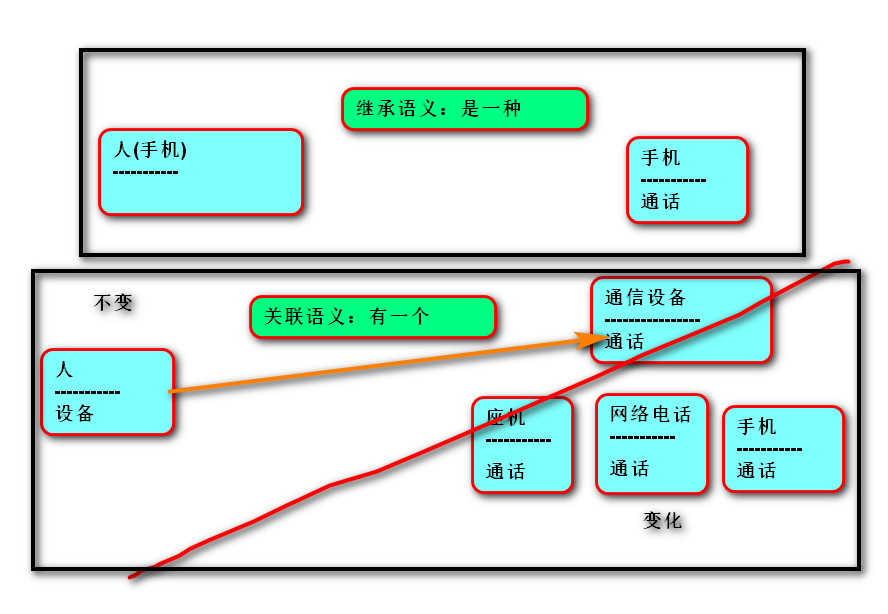
抽象的是稳定的。实现是多变的。

### 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。

## 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（变量）。

**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。

# 练习

## 每日课堂练习

### 四月十六（day11）

#### １.练习:手雷爆炸

手雷爆炸了,可能伤害敌人,玩家.还有可能伤害未知事物(鸭子,树,房子).

要求:如果增加了新的事物,手雷代码不变.

**class** Grenade:

*"""*

*手雷*

*"""*

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

self.atk = atk

**def** explode(self, \*args):*#　单星号元组形参*

*"""*

*爆炸*

**:return***:*

*"""*

**for** item **in** args:

**if not** isinstance(item, Damageable):

print(**"类型不兼容"**)

**return**

item.damage(self.atk)

**class** Damageable:

*"""*

*可以受伤*

*"""*

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

self.hp = hp

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

*# 约束子类必须具有当前方法*

*# raise NotImplementedError()*

self.hp -= value

**class** Player(Damageable):

**def** damage(self, value=0):

*# self.hp -= value*

super().damage(value)

print(**"碎屏"**)

**class** Enemy(Damageable):

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

*# self.hp -= value*

super().damage(value)

print(**"播放动画"**)

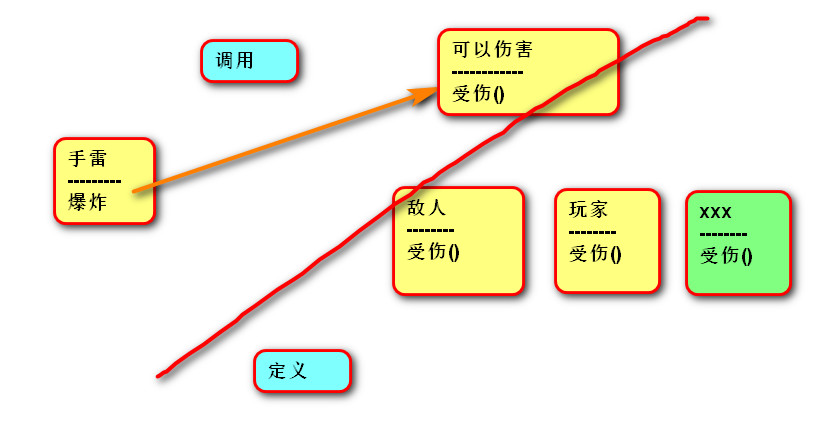
g01 = Grenade(10)

p02 = Player(100)

e03 = Enemy(50)

g01.explode(p02, e03)

内存图：



#### ２.练习:计算图形面积

*"""*

*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:

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

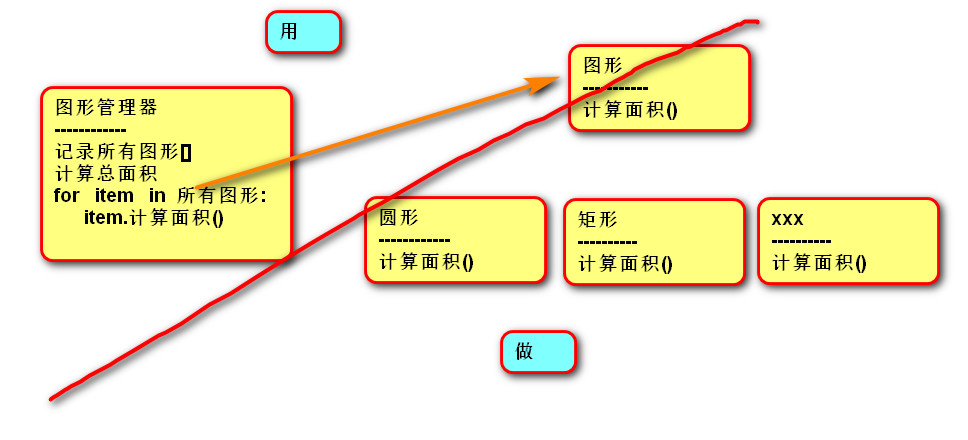
manager = GraphicManager()

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

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

manager.add\_graphic(Circle(5))

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

内存图：

#### 3 .老张开车去东北　　继承 -- 设计思想

*"""*

*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***:*

*"""*

*# 如果传入的对象,不是交通工具,则退出.*

**if not** isinstance(vehicle,Vehicle):

print(**"传入的不是交通工具"**)

**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(),确保父类，子类都有这个方法*

**raise** NotImplementedError()

*# print("儿子们,必须有这个方法啊")*

**class** Car(Vehicle):

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

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")*

print(p01.name)

p01.go\_to(Airplane(),**"东北"**)

内存图：（使用继承思想，不用面面俱到）

### 四月十七（day12）

##### 1.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.

**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

**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

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

print(lw.calculate\_salary())

*# 转岗*

lw.job = Programmer(8000,100000)

print(lw.calculate\_salary())

**内存图：**

##### ２.内置可重写函数

**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:"**,w01)*# 将对象转换为字符串 # w01:奴家叫:金莲,年芳:25*

*#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:"**,w02) *# w02:奴家叫:金莲,年芳:25*

w01.name =**"莲莲"**

print(w02.name) *# 金莲*

##### 3.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)

##### ４.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:*

*"""*

*学生数据模型类*

*"""*

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

*"""*

*创建学生对象*

**:param** *id: 编号*

**:param** *name: 姓名*

**:param** *age: 年龄*

**:param** *score: 成绩*

*"""*

self.id = id

self.name = name

self.age = age

self.score = score

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

**return "编号：%d,名字：%s,年龄：%d,成绩：%d"**%(self.id,self.name,self.age,self.score)

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

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

s01=StudentModel(1,**"老杜"**,18,60)

print(s01) *# 编号：1,名字：老杜,年龄：18,成绩：60*

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

print(s02) *# 编号：1,名字：老杜,年龄：18,成绩：60*

list01=[s01,s02]

print(list01) *# [StudentModel(1,"老杜",18,60), StudentModel(1,"老杜",18,60)]*

## 每日课后作业

### 四月十五（day10）

##### １.使用面对对象思想编程

# 1. 使用面向对象思想,写出下列场景:

# 玩家(攻击力)攻击敌人,敌人受伤(血量)后掉血,还可能死亡(播放动画).

# 敌人(攻击力)攻击力攻击玩家,玩家(血量)受伤后碎屏,还可能死亡(游戏结束).

# 程序调试,画出内存图.

**class** Player:

*"""*

*玩家类*

*"""*

**def** \_\_init\_\_(self,hp,atk):

self.atk = atk

self.hp = hp

**def** attack(self,enemy):

print(**"打死你"**)

*# 调用敌人受伤方法(敌人负责定义受伤逻辑)*

enemy.damage(self.atk)

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

self.hp -= value

print(**"玩家受伤啦,屏幕碎啦"**)

**if** self.hp <= 0:

self.\_\_death()

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

print(**"玩家死亡,游戏结束"**)

**class** Enemy:

**def** \_\_init\_\_(self,hp,atk):

self.hp = hp

self.atk = atk

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

self.hp -= value

print(**"受伤啦"**)

**if** self.hp <= 0:

self.\_\_death()

**def** attack(self,player):

print(**"打死你"**)

player.damage(self.atk)

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

print(**"死啦,播放动画"**)

p01 = Player(100,50)

e01 = Enemy(60,10)

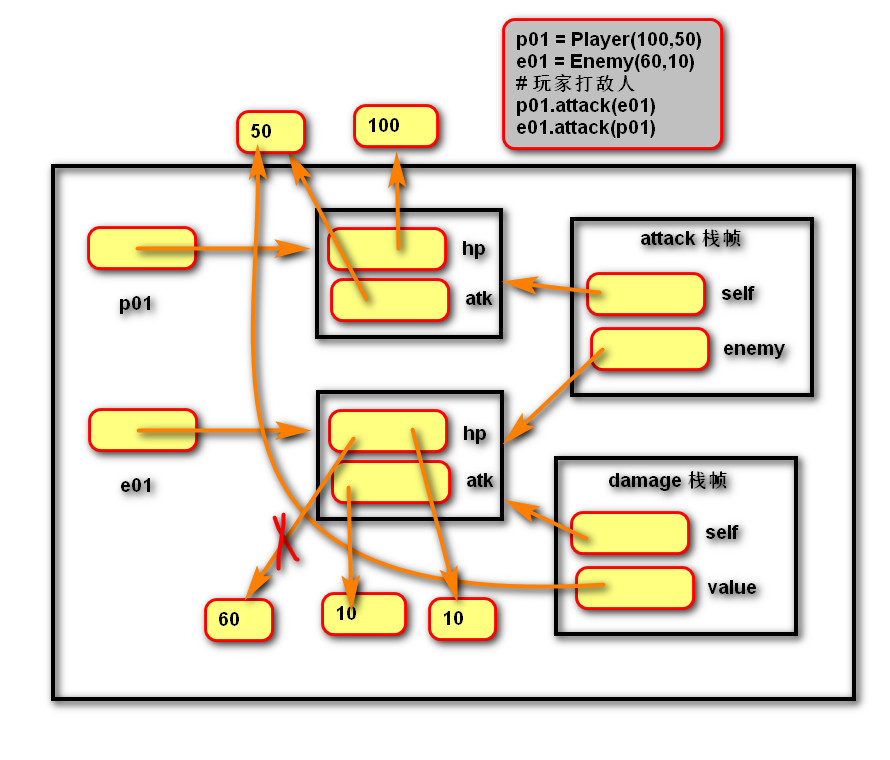
*# 玩家打敌人*

p01.attack(e01)

*# p01.attack(e01)*

e01.attack(p01)

内存图：

每创建一个对象都会开辟一块内存空间，每次调用方法会开辟一块临时的栈帧

### 四月十六（day11）

##### 1.定义武器

定义父类:武器,数据:攻击力,行为:购买(所有子类都一样).攻击(不知道怎么攻击)

定义子类:枪,数据:射速,行为:攻击

定义子类:手雷,数据:爆炸范围,行为:攻击

创建相应对象,调用相应方法.

画出内存图

**class** Weapon:

*"""*

*武器*

*"""*

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

self.atk = atk

**def** buy(self):

print(**"购买武器"**)

**def** attack(self):

*# 子类如果没有当前方法,就会报错*

**raise** NotImplementedError()

**class** Gun(Weapon):

*"""*

*枪*

*"""*

**def** \_\_init\_\_(self,atk,speed):

super().\_\_init\_\_(atk)

self.att\_speed = speed

**def** attack(self):

print(**"开枪啦"**)

**class** Grenade(Weapon):

*"""*

*手雷*

*"""*

**def** \_\_init\_\_(self, atk, range):

super().\_\_init\_\_(atk)

self.explode\_range = range

**def** attack(self):

print(**"爆炸啦"**)

g01 = Gun(10,0.1)

g01.buy()

g01.attack()

grenade01 = Grenade(50,10)

grenade01.buy()

grenade01.attack()

##### 2.员工管理器

一家公司,有如下几种岗位:

普通员工:底薪

程序员:底薪 + 项目分红

销售员:底薪 + 销售额

定义员工管理器,记录所有员工,提供计算总薪资方法.

要求:增加新岗位,员工管理器代码不做修改.

体会:依赖倒置

**class** EmployeeManager:

*"""*

*员工管理器*

*"""*

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

self.\_\_all\_employee = []

**def** add\_employee(self, employee):

**if not** isinstance(employee, Employee):

**return**

self.\_\_all\_employee.append(employee)

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

*"""*

*计算总薪资*

**:return***:*

*"""*

total\_salary = 0

**for** item **in** self.\_\_all\_employee:

*# 编码期间:item 认为是员工*

*# 运行期间:item 实际是具体员工*

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

**return** total\_salary

**class** Employee:

*"""*

*员工类*

*作用:代表具体员工,隔离员工管理器与具体员工的变化.*

*"""*

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

self.name = name

self.base\_salary = salary

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

**return** self.base\_salary

**class** Programmer(Employee):

*"""*

*程序员*

*"""*

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

super().\_\_init\_\_(name, 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(Employee):

*"""*

*销售员*

*"""*

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

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

self.sale\_value = sale\_value

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

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

manager = EmployeeManager()

manager.add\_employee(Employee(**"zs"**,3000))

manager.add\_employee(Programmer(**"xp"**,4000,10))

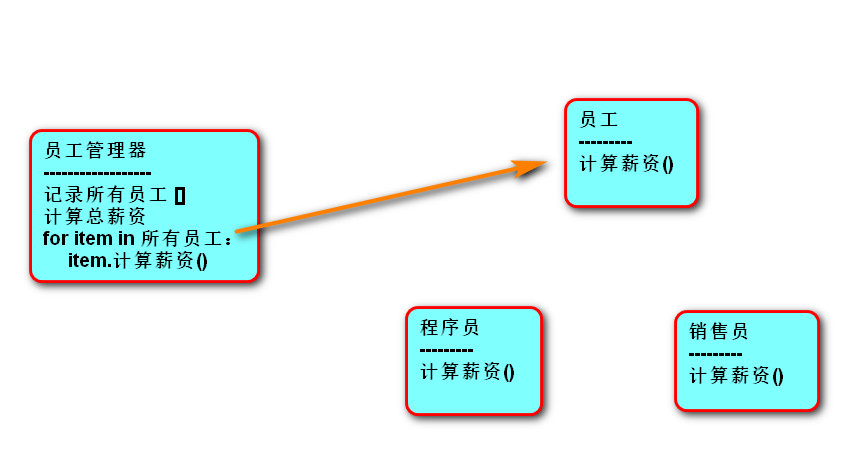
manager.add\_employee(Programmer(**"xx"**,99999,6135138))

manager.add\_employee(Salesmen(**"pp"**,3000,500))

re = manager.get\_total\_salary()

print(re)

**内存图：**



## 每日早上复习

### 四月十三（day09）

*"""*

*day09 复习*

*1. 实例成员*

*2. 类成员*

*3. 静态方法*

*4. 属性*

*"""*

*# 1.实例成员:对象的成员(创建对象,就会开辟空间,存储的数据)*

*# 作用:表示对象的信息,与对象的行为.*

**class** A:

**def** \_\_init\_\_(self, 参数):

self.实例变量 = 参数

**def** 实例方法名称(self):

print(self.实例变量)

a01 = A(10)

print(a01.实例变量)

a01.实例方法名称() *# 自动传递对象地址*

*# 2.类成员:属于类级别的成员*

*# 表示所有对象共享的成员*

**class** B:

*# 无论创建多少对象,类变量只有一份,被所有对象所共享.*

类变量 = 10

*# 类方法:操作类变量*

*# 因为没有对象地址,所以不能操作实例成员.*

@classmethod

**def** 类方法名称(cls):

print(cls.类变量)

print(B.类变量)

B.类方法名称() *# 自动传递类名*

*# 函数*

*# def fun01():*

*# print("函数执行啦")*

*# fun01()*

**class** C:

*# 3.静态方法*

@staticmethod

**def** fun01():

print(**"方法执行啦"**)

C.fun01()

*# 4. 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)

### 四月十五（day10）

*"""*

*day10 复习*

*封装:*

*1. 封装数据:将多个基本类型复合为一个自定义类型.*

*-- 优势:复合人类的思考方式*

*体现对数据的操作方式*

*2. 封装功能:对外提供必要的功能,隐藏实现细节.*

*-- 模块化的编程思想*

*3. 分而治之, 封装变化, 高内聚, 低耦合.*

*分解 变化点 类职责单一 类与类的关系松散*

*MVC*

*"""*

**class** StudentManagerController:

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

print(**"添加学生"**)

**class** StudentManagerView:

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

self.\_\_manager = StudentManagerController()

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

*# 调用逻辑控制器的实例方法*

*# 语法:控制器对象地址.实例方法*

self.\_\_manager.add\_student()

### 四月十六（day11）

*"""*

*day11 复习*

*继承*

*"""*

**class** 爸爸:

**def** \_\_init\_\_(self, 数据):

self.爸爸数据1 = 数据

**def** 方法1(self):

print(**"方法1"**)

**class** 儿子(爸爸):

*# 子类没有构造函数,则自动调用父类构造函数*

*# 子类有构造函数,不再自动调用父类构造函数*

**def** \_\_init\_\_(self, 爸爸数据, 儿子数据):

super().\_\_init\_\_(爸爸数据)

self.子儿子数据 = 儿子数据

**def** 方法2(self):

print(**"方法2"**)

c01 = 儿子(10, 20)

c01.方法1()

c01.方法2()

*# 判断对象与类型*

print(isinstance(c01, 爸爸))

*# 判断类型与类型*

print(issubclass(儿子, 爸爸))

print(c01.爸爸数据1)

# 案例：学生管理系统

*"""*

*学生管理器系统*

*"""*

**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: 成绩*

*"""*

self.id = id

self.name = name

self.age = age

self.score = score

@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:

*"""*

*学生逻辑控制器*

*"""*

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

self.\_\_list\_stu = []

@property

**def** list\_stu(self):

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

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

*"""*

*添加新学生*

**:param** *stu: 需要添加的学生对象*

*"""*

stu.id = self.\_\_generate\_id()

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

**def** \_\_generate\_id(self):

*# 生成编号的需求:新编号,比上次添加的对象编号多1.*

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

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

*# else:*

*# id = 1*

*# return id*

**return** self.\_\_list\_stu[-1].id + 1 **if** len(self.\_\_list\_stu) > 0 **else** 1

**def** order\_by\_score(self):

*"""*

*根据成绩升叙排列*

**:return***:*

*"""*

*# 创建新列表(目的:不影响原有列表)*

new\_list = self.\_\_list\_stu[:]

**for** r **in** range(len(new\_list) - 1):

**for** c **in** range(r + 1, len(new\_list)):

**if** new\_list[r].score > new\_list[c].score:

new\_list[r], new\_list[c] = new\_list[c], new\_list[r]

**return** new\_list

**def** remove\_student(self, id):

*"""*

*删除学生*

**:param** *id:*

**:return***:*

*"""*

**for** stu **in** self.list\_stu:

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

self.list\_stu.remove(stu)

**return True** *# 表示删除成功*

**return False** *# 表示删除失败*

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

*"""*

*更新学生信息*

**:return***:*

*"""*

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

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

item.name = stu.name

item.score = stu.score

item.age = stu.age

**return True**

**return False**

*# controller = StudentManagerController()*

*# controller.add\_student(StudentModel("zs",18,85))*

*# controller.add\_student(StudentModel("zs",18,85))*

*# for item in controller.list\_stu:*

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

**class** StudentManagerView:

*"""*

*界面视图类*

*"""*

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

*# 创建逻辑控制器对象*

self.\_\_manager = StudentManagerController()

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

*# 1. 在控制台中录入学生信息,存成学生对象StudentModel.*

stu = StudentModel()

stu.name = input(**"请输入姓名:"**)

stu.age = int(input(**"请输入年龄:"**))

stu.score = int(input(**"请输入成绩:"**))

*# 2. 调用逻辑控制器的add\_student方法*

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

print(self.\_\_manager)

**def** \_\_output\_students(self, list\_target):

*"""*

*显示学生列表信息*

**:return***:*

*"""*

*# for stu in self.\_\_manager.list\_stu:*

**for** stu **in** list\_target:

print(**"%d -- %s -- %d -- %d"** % (stu.id, stu.name, stu.age, stu.score))

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

*"""*

*根据成绩显示所有学生信息*

**:return***:*

*"""*

list\_target = self.\_\_manager.order\_by\_score()

self.\_\_output\_students(list\_target)

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

id = int(input(**"请输入需要删除的学生编号:"**))

**if** self.\_\_manager.remove\_student(id):

print(**"删除成功"**)

**else**:

print(**"删除失败"**)

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

*"""*

*修改学生信息*

**:return***:*

*"""*

stu = StudentModel()

stu.id = int(input(**"请输入需要修改的学生编号:"**))

stu.name = input(**"请输入姓名:"**)

stu.age = int(input(**"请输入年龄:"**))

stu.score = int(input(**"请输入成绩:"**))

**if** self.\_\_manager.update\_student(stu):

print(**"修改成功"**)

**else**:

print(**"更新失败"**)

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

*"""*

*显示菜单*

**:return***:*

*"""*

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) 按照成绩降序排列"**)

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

*"""*

*选择菜单*

**:return***:*

*"""*

number = input(**"请输入选项:"**)

**if** number == **"1"**:

self.\_\_input\_students()

**elif** number == **"2"**:

self.\_\_output\_students(self.\_\_manager.list\_stu)

**elif** number == **"3"**:

self.\_\_delete\_student()

**elif** number == **"4"**:

self.\_\_modify\_student()

**elif** number == **"5"**:

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

**def** main(self):

*"""*

*界面入口方法*

**:return***:*

*"""*

**while True**:

self.\_\_display\_menu()

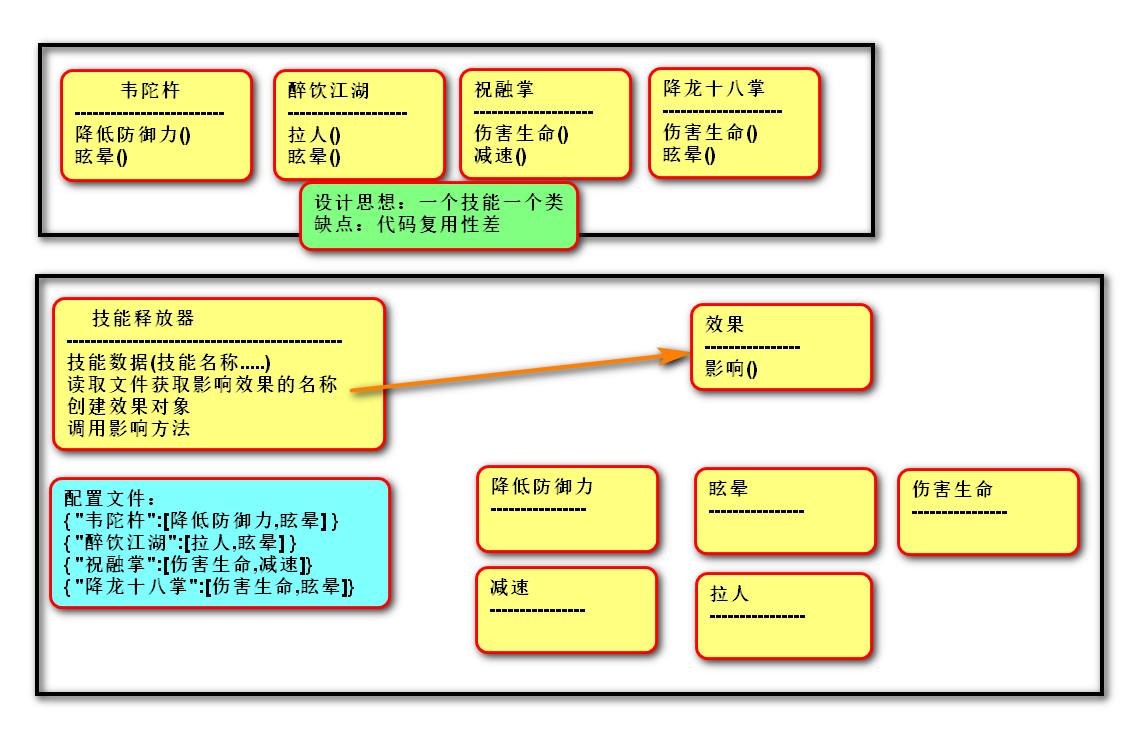
self.\_\_select\_menu()

view = StudentManagerView()

view.main()

# 天龙八部技能设计

**内存图：**

**代码：**

*"""*

*天龙八部技能系统*

*设计原则:*

*"""*

**class** ImpactEffect:

*"""*

*影响效果*

*隔离技能释放器 与 具体的影响效果*

*"""*

**def** impact(self):

*# 要求子类必须实现,否则报错*

**raise** NotImplementedError()

**class** LowerDefense(ImpactEffect):

*"""*

*降低防御力*

*"""*

**def** \_\_init\_\_(self,distance,ratio):

*# 距离*

self.distance = distance

*# 比例*

self.ratio = ratio

**def** impact(self):

print(**"降低%d米内,目标的防御力为%d."**%(self.distance,self.ratio))

**class** LowerSpeed(ImpactEffect):

*"""*

*降低速度*

*"""*

**def** \_\_init\_\_(self, time, ratio):

*# 时间*

self.time = time

*# 比例*

self.ratio = ratio

**def** impact(self):

print(**"降速为%.1f.时间是:%d"** % (self.ratio, self.time))

**class** Damage(ImpactEffect):

*"""*

*伤害生命*

*"""*

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

*# 伤害值*

self.value = value

**def** impact(self):

print(**"伤害%d生命"** % (self.value))

**class** SkillDeployer:

*"""*

*技能释放器*

*"""*

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

self.name = name

*# 配置释放器,存储当前技能具有的所有影响效果对象*

self.\_\_list\_impact = self.\_\_config\_deployer()

**def** \_\_config\_deployer(self):

*"""*

*配置释放器*

**:return***:*

*"""*

*#\*. 定义配置*

*#1. 读取相应的影响效果*

*#2. 创建影响效果对象*

*# 依赖注入*

dict\_skill\_config = {

**"韦陀杵"**:[**"LowerDefense(10,0.5)"**,**"Damage(30)"**],

**"降龙十八掌"**: [**"LowerSpeed(5,0.2)"**, **"Damage(80)"**]

}

*# ["LowerDefense(10,0.5)","Damage(30)"]*

*# 根据键(技能名称)获取值(影响效果列表)*

list\_impact\_name = dict\_skill\_config[self.name]

*# list\_impact = []*

*# for item in list\_impact\_name:*

*# # 创建影响效果对象,并加入到列表中*

*# list\_impact.append(eval(item))*

**return** [eval(item) **for** item **in** list\_impact\_name]

**def** generate\_skill(self):

*"""*

*生成技能*

**:return***:*

*"""*

print(self.name,**"释放啦!"**)

*# 执行所有影响效果*

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

*# 编码期间:认为调用的是影响效果(父类ImpactEffect)*

*# 运行期间:由于创建的是子类对象(伤害生命Damage...),所以执行的是子类方法.*

item.impact()

*#--------测试---------------*

*# 创建技能对象*

wei\_tuo\_chu = SkillDeployer(**"韦陀杵"**)

*# 释放技能*

wei\_tuo\_chu.generate\_skill()

xiang\_long\_18\_zhang = SkillDeployer(**"降龙十八掌"**)

xiang\_long\_18\_zhang.generate\_skill()