# 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：

-- 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。

1. 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。

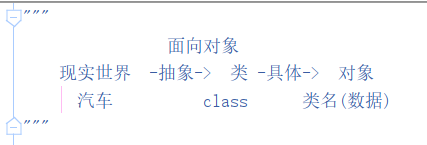
1. 缺点：学习曲线陡峭。

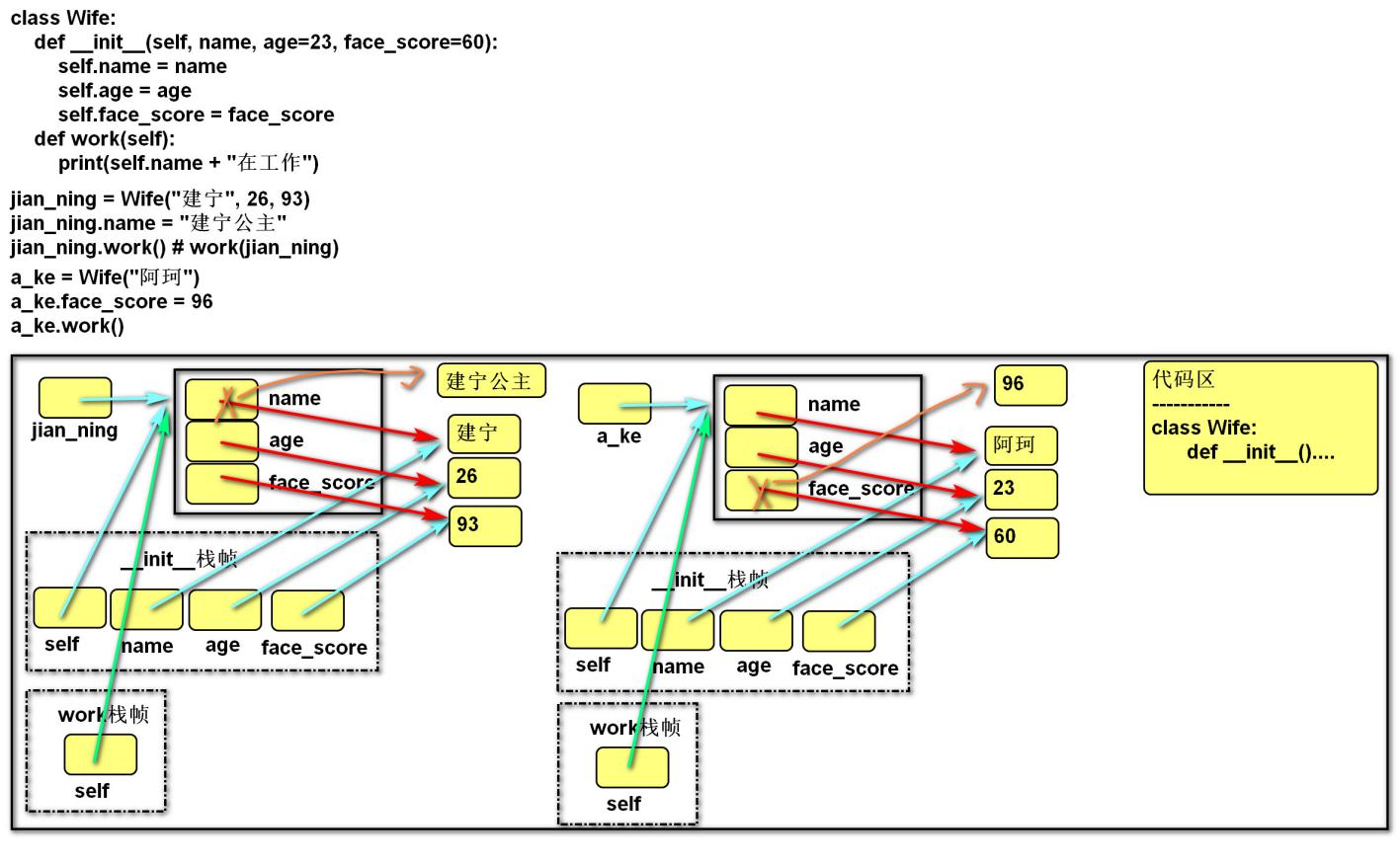
## 类和对象

1. 类：一个抽象的概念，即生活中的“类别”。(**抽象**)
2. 对象：类的具体实例，即归属于某个类别的“个体”。（**具体的**）
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。
2. 
3. 类和对象内存图



### 语法

#### 定义类

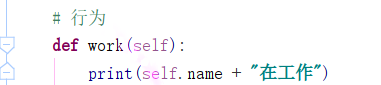
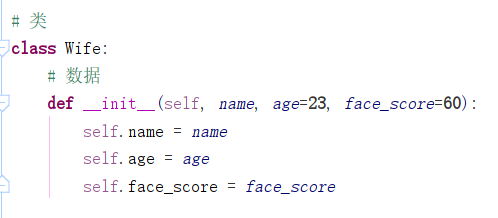
1. 代码

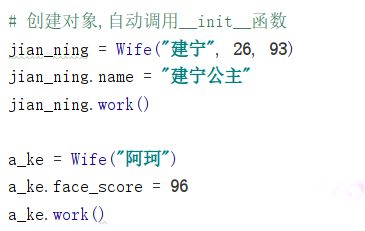
class 类名: #类名所有单词首字母大写,多个单词不用下划线隔开.

def \_\_init\_\_(self,参数列表):

self.**实例变量** = 参数 (**. 指CPU寻址**)

方法成员





1. 说明

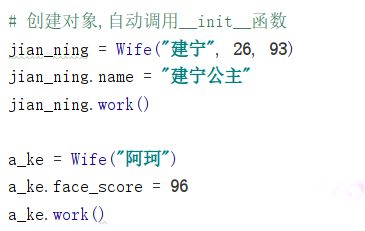
-- 类名所有单词首字母大写.

-- \_\_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 = 类名(参数列表)

（上面1.代码的图片）

练习：创建手机类

数据：品牌、价格、颜色

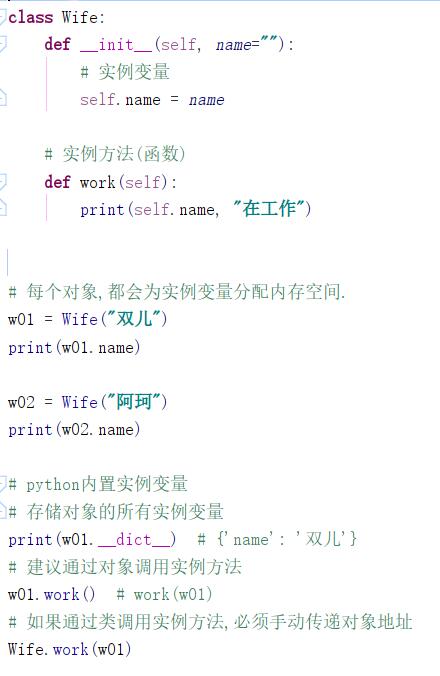
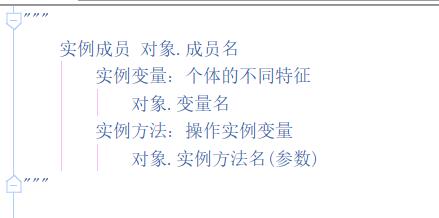
行为：通话

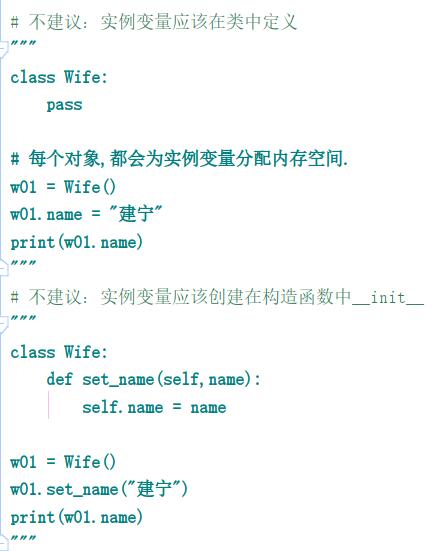
实例化两个对象并调用其函数

画出内存图

### 实例成员

#### 实例变量(自己的)





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：描述某个对象的数据。
3. \_\_dict\_\_：对象的属性，用于存储自身实例变量的字典。

练习1：创建狗类

数据：品种、昵称、身长、体重

行为：吃(体重增长1)

实例化两个对象并调用其函数

画出内存图

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

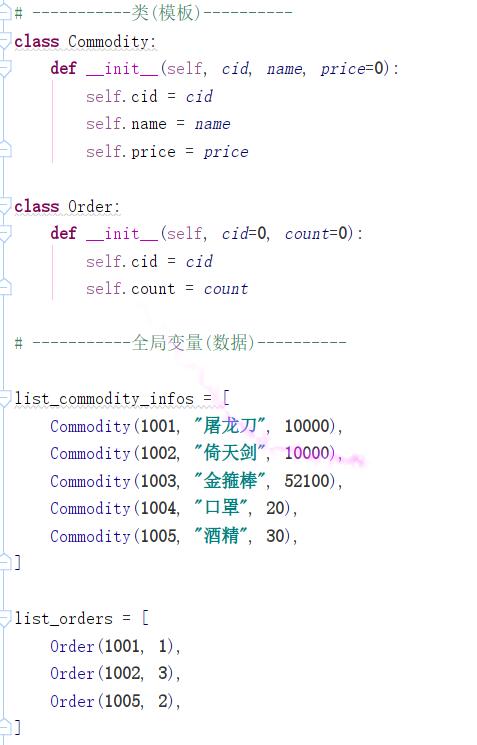
list\_orders = [

{"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']}")



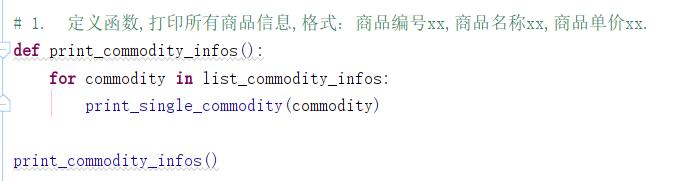
# 1. 定义函数,打印所有商品信息,格式：商品编号xx,商品名称xx,商品单价xx.

def print\_commodity\_infos():

for commodity in list\_commodity\_infos:

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

print\_single\_commodity(commodity)



# 2. 定义函数,打印商品单价小于2万的商品信息

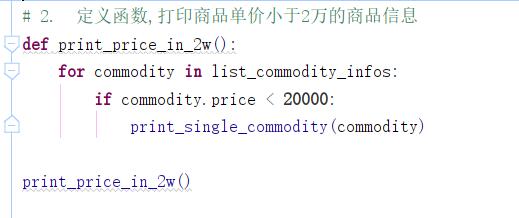
def print\_price\_in\_2w():

for commodity in list\_commodity\_infos:

if commodity["price"] < 20000:

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

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: # 遍历所有订单

# order["cid"] --> 1001 -->

for commodity in list\_commodity\_infos: # 遍历所有商品信息

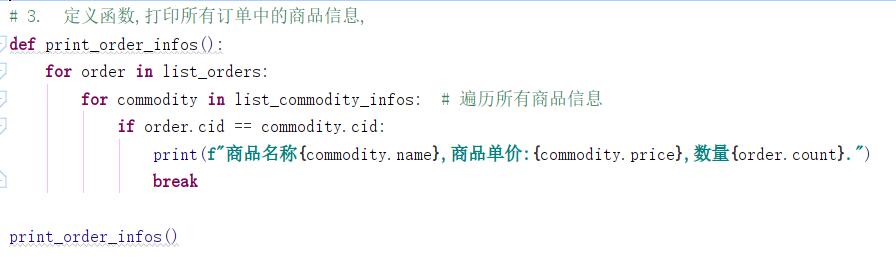
# commodity["cid"] --> 1001

# 使用订单中的商品编号 在 商品信息中查找(商品)

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"]:

# 使用更大的那个订单 替换 假设的订单

max\_value = list\_commodity\_infos[i]

return max\_value



# 5. 根据单价对商品列表降序排列

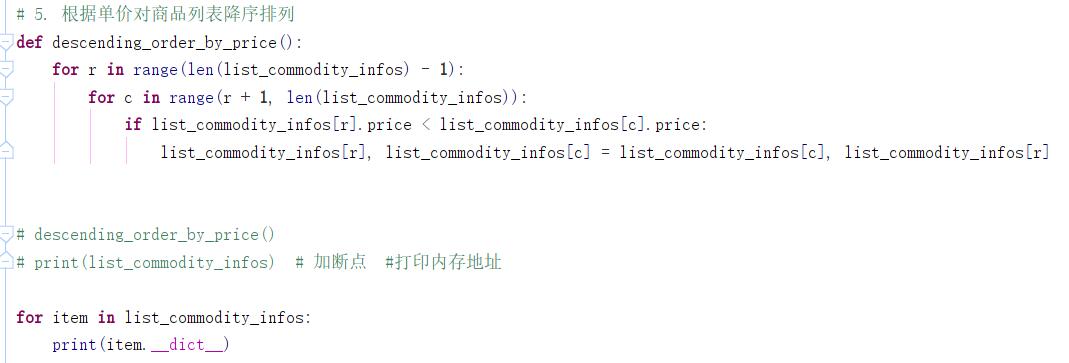
def descending\_order\_by\_price():

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]



#### 实例方法

1. 语法

(1) 定义： def 方法名称(**self**, 参数列表):

方法体

1. 调用： 对象地址.实例方法名(参数列表)

对象名 = 类名(数据)

对象名.实例变量

对象名.实例方法()

**不建议**通过类名访问实例方法

1. 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**。

1. 作用：表示对象行为。

### 类成员

#### 类变量（大家的）

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

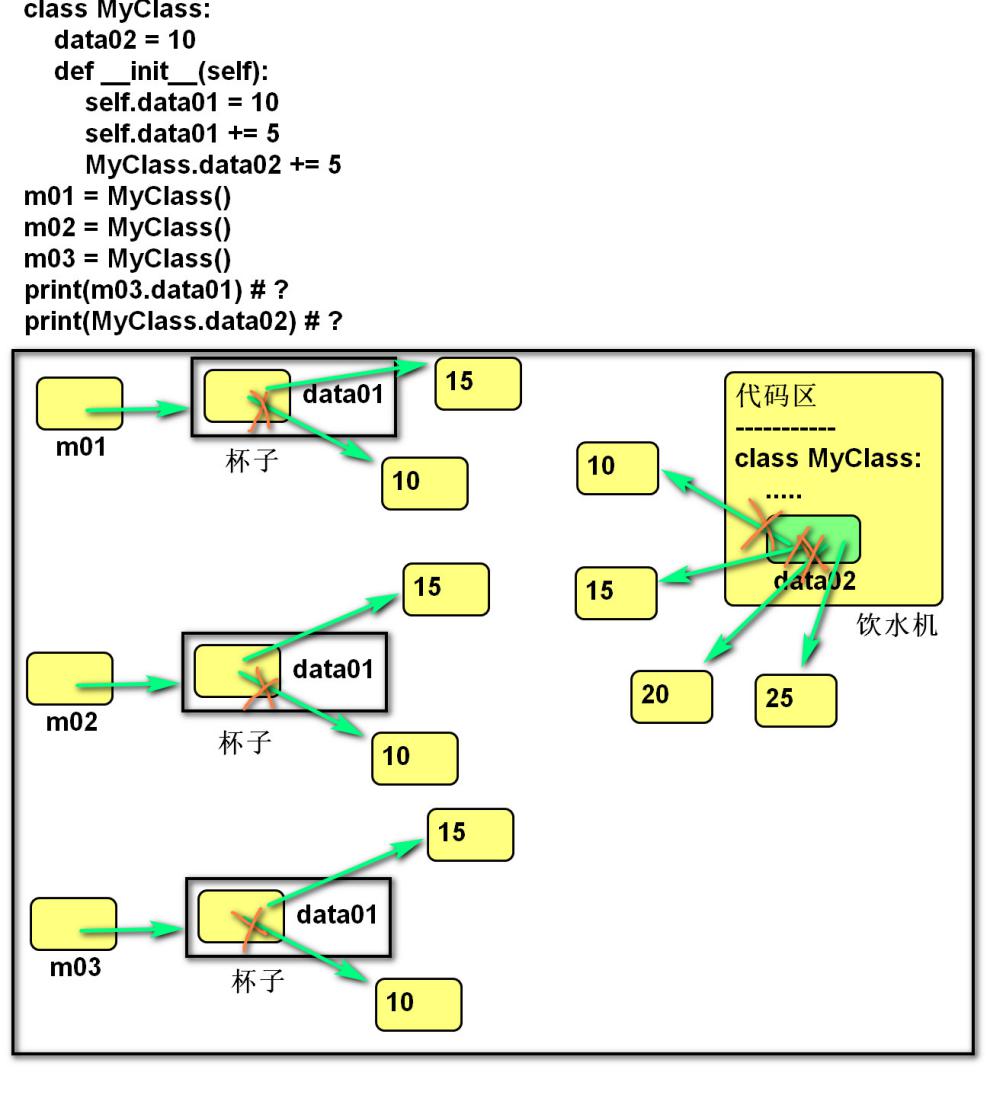
不建议通过对象访问类变量

1. 说明

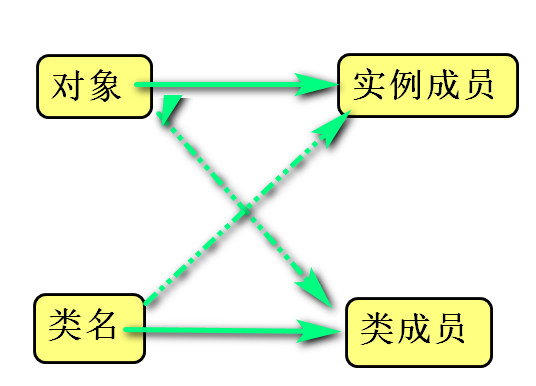
(1) 存储在类中。

(2) 只有一份，被所有对象共享。

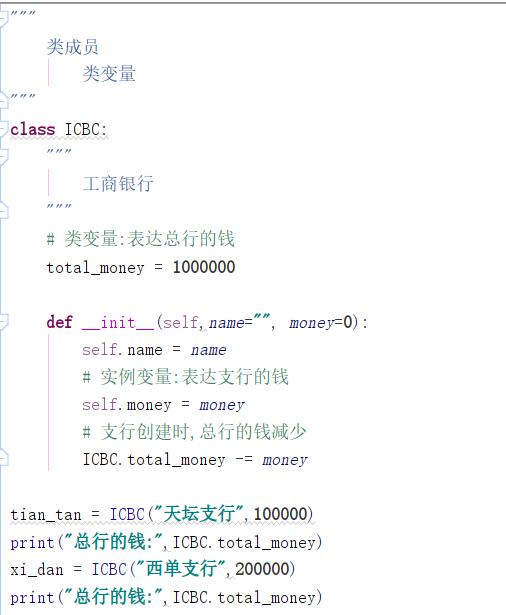
1. 作用：描述所有对象的共有数据。
2. 类变量内存图



1. 类成员与实例成员访问方式



1. 类变量的应用



#### 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

(1) 至少有一个形参，第一个形参用于**绑定类**，一般命名为'**cls**'

(2) 使用@classmethod修饰的目的是调用类方法时可以隐式传递类。

(3) 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。

练习1：对象计数器

统计构造函数执行的次数

使用类变量实现

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")

print(w05.count) # 5

Wife.print\_count()# 总共娶了5个老婆

### 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

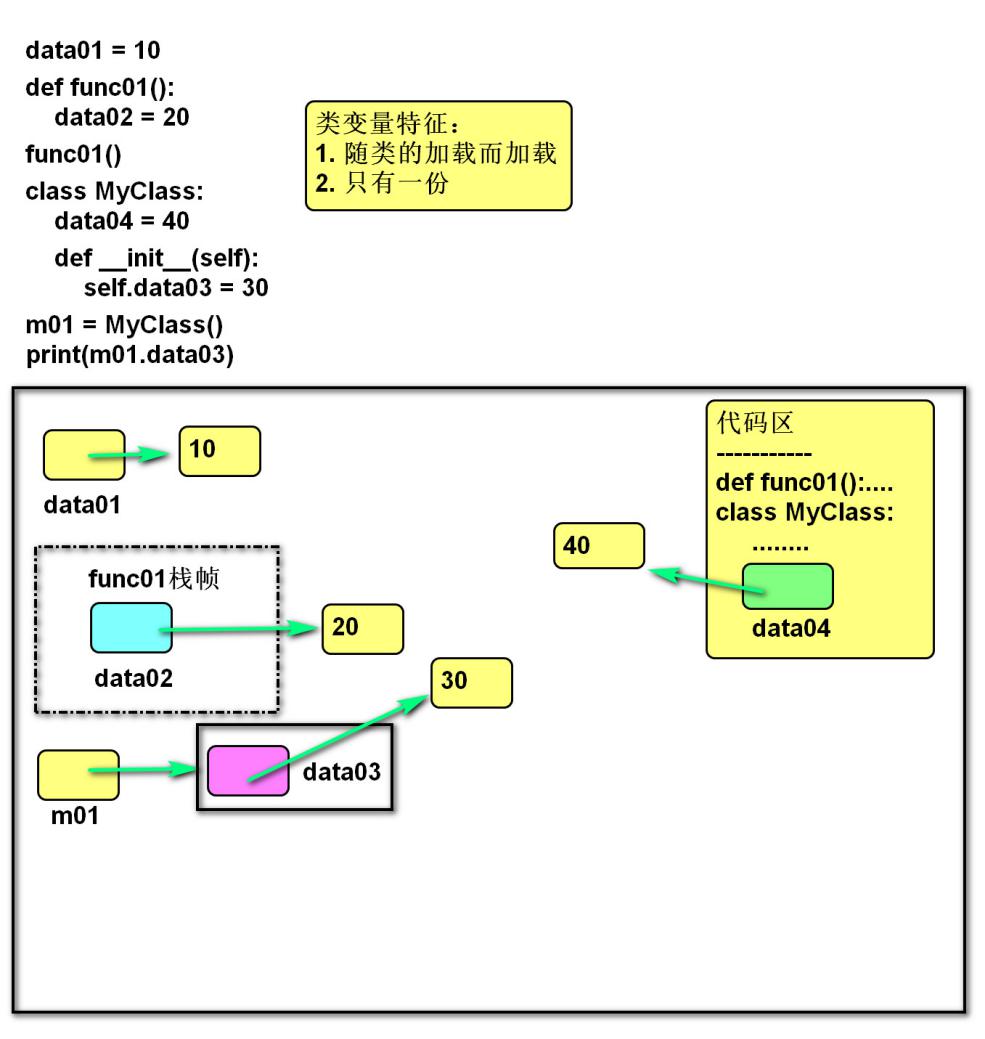
(1) 使用@ staticmethod修饰的目的是该方法不需要隐式传参数。

(2) 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。

**所有变量内存图**

图片：

内存图：

## 三大特征

### 封装

#### 数据角度讲

1. 定义：

将一些基本数据类型复合成一个自定义类型。

1. 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）。

#### 行为角度讲

1. 定义：

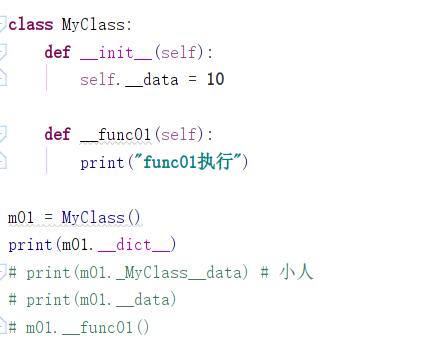
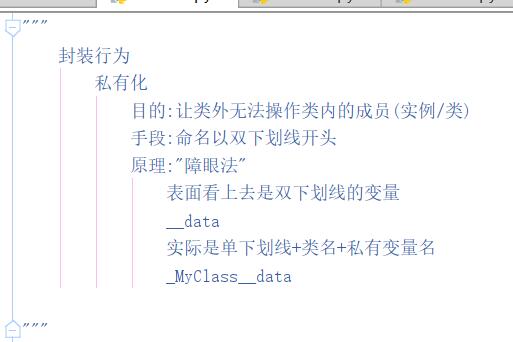
向类外提供必要的功能，隐藏实现的细节。

1. 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。

1. 私有成员：
2. 作用：无需向类外提供的成员，可以通过私有化进行屏蔽。
3. 做法：命名使用**双下划线**开头。
4. 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。



1. 属性@property：

将方法的使用方式像操作变量一样方便，从而保护实例变量。



**快捷方式：props + 回车**

1. 定义：

@property

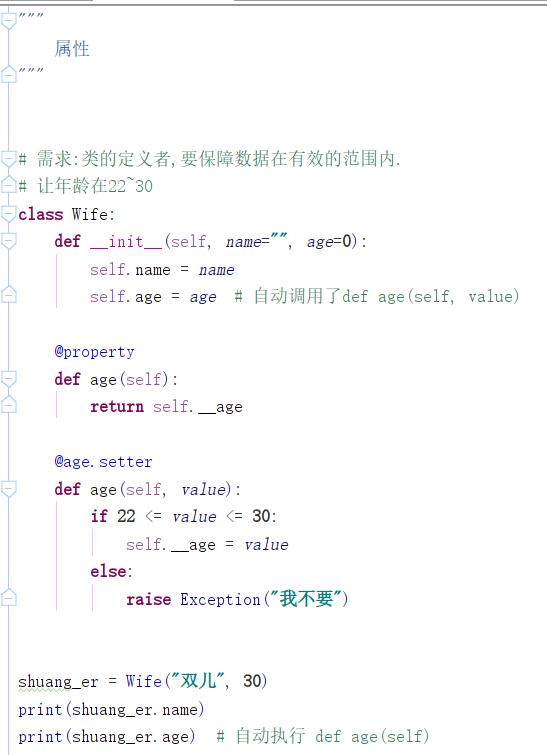
def 属性名(self):

return self.\_\_属性名

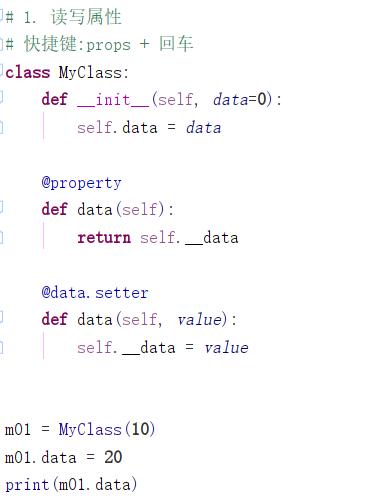
@属性名.setter

def 属性名(self, value):

self.\_\_属性名= value



**读写属性**



**只读属性**



1. 调用：

对象.属性名 = 数据

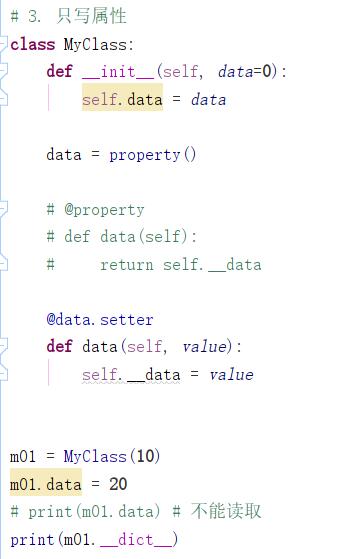
变量 = 对象.属性名

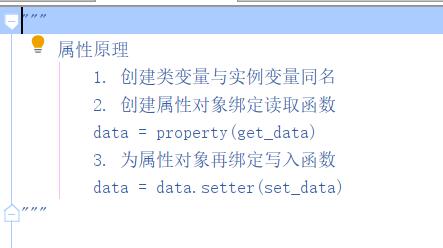
1. 说明：

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)



1. 属性原理：
2. 

练习1：创建敌人类，并保护数据在有效范围内

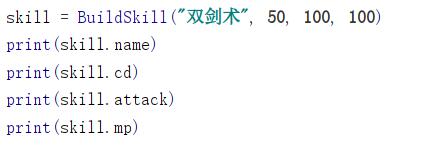
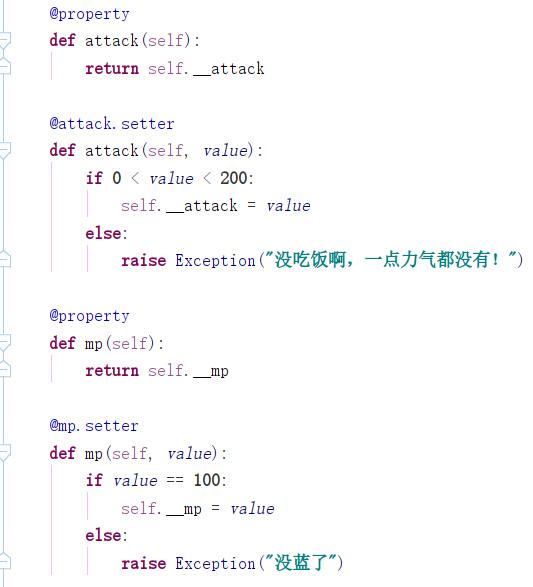
数据:姓名/ 攻击力/ 血量

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



#### 设计角度讲

1. 定义：

(1) 分而治之--->行为的不同

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。

(2) 变则疏之(寻找行为变化点)

变化的地方独立封装，避免影响其他类。

(3) 高 内 聚

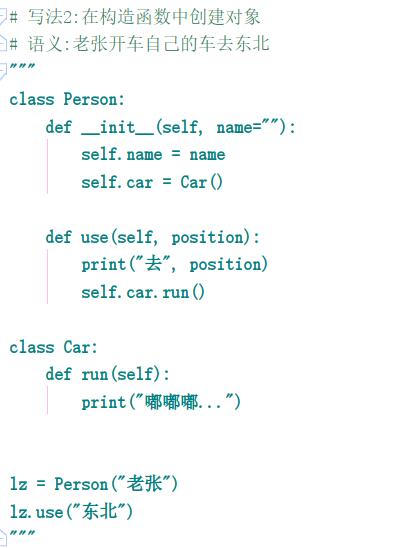
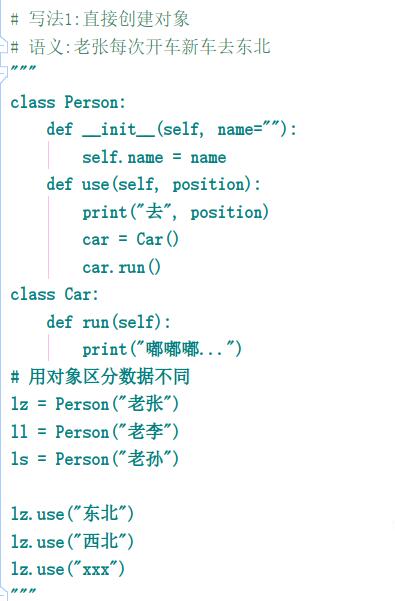
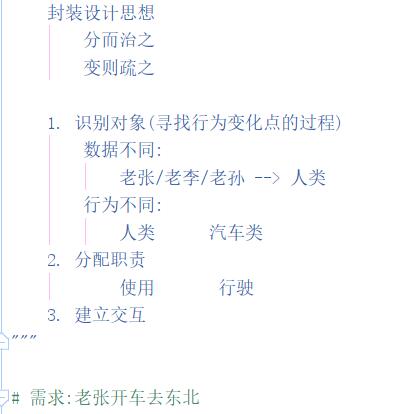
类中各个方法都在完成一项任务(单一职责的类)。

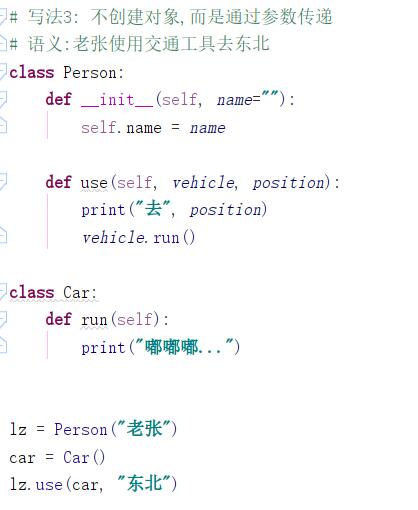
(4) 低 耦 合

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。

1. 优势：

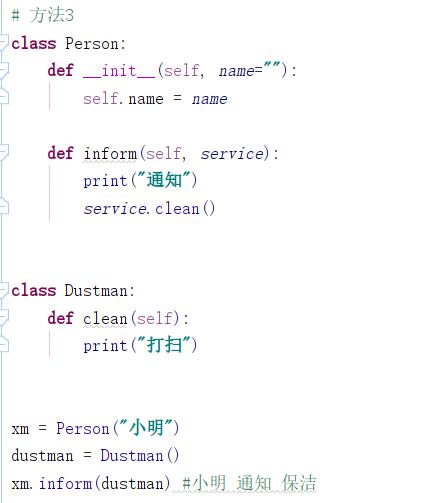
便于分工，便于复用，可扩展性强。





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

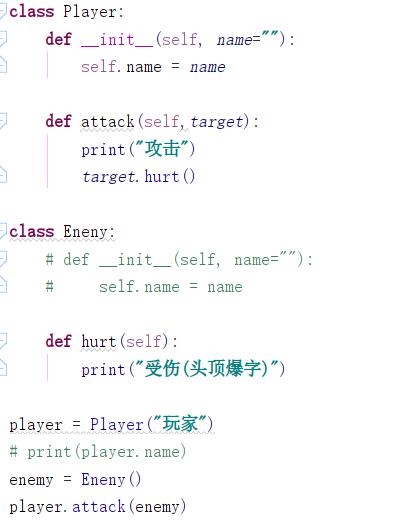


练习2：以面向对象的思想，描述以下情景:

小明在银行取钱（银行钱多了，小明钱少了）

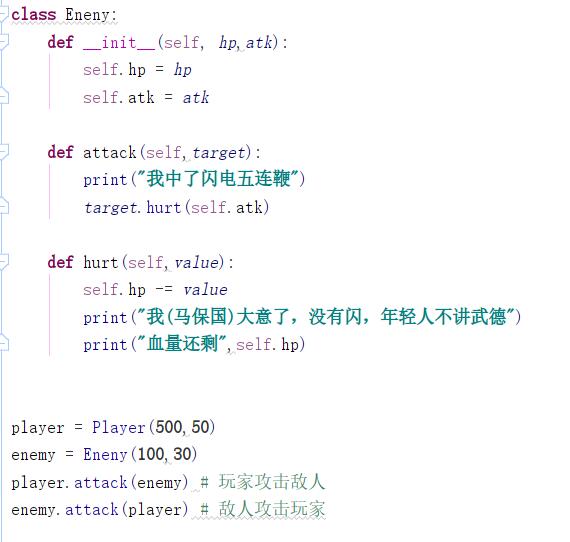
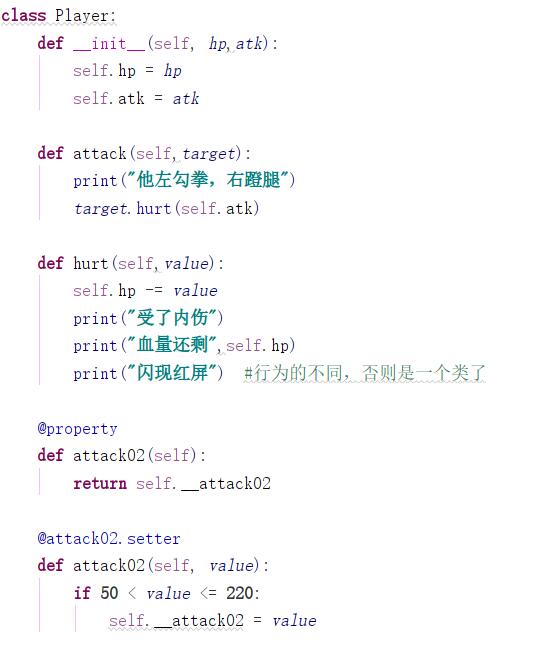
练习3：以面向对象思想,描述下列情景.

玩家攻击敌人,敌人受伤(头顶爆字).



练习4：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).



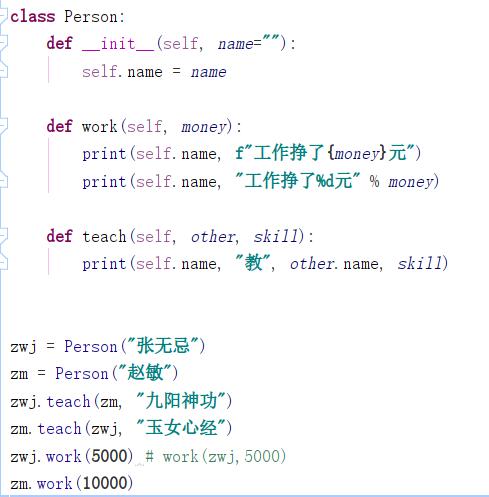
练习5：以面向对象思想,描述下列情景.

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



### 继承

#### 语法角度讲

##### 继承方法

1. 代码:

class 父类:

def 父类方法(self):

方法体

class 子类**(父类)**：

def 子类方法(self):

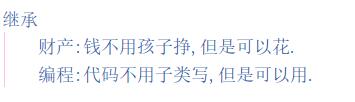
方法体

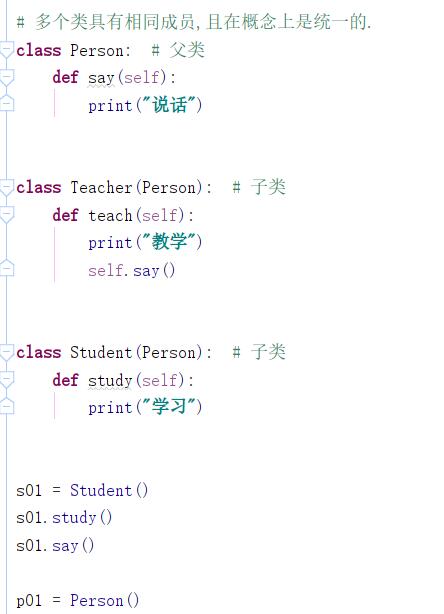
儿子 = 子类()

儿子.子类方法()

儿子.父类方法()

1. 说明：

子类直接拥有父类的方法.



##### 内置函数

**关系的判定**

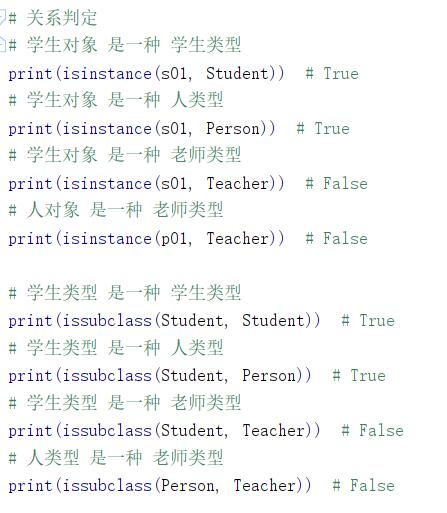
**isinstance**(**对象**, 类型) --->**有包含的成分**

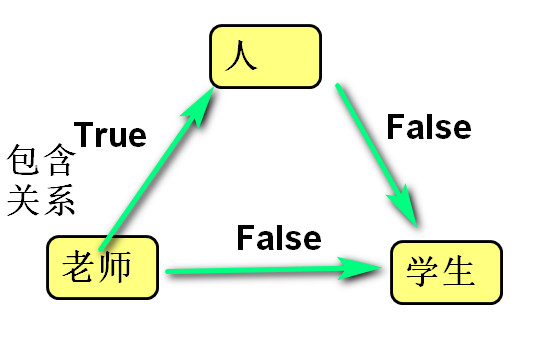
返回指定对象是否是某**个**类的**对象**。

**issubclass**(**类型**，类型)---->**有包含的成分**

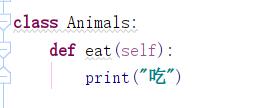
返回指定类型是否属于某**个类型**。

Type() == 类型 --->**没有包含的成分**

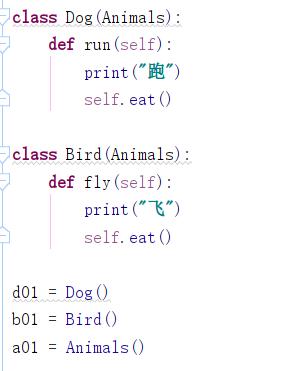




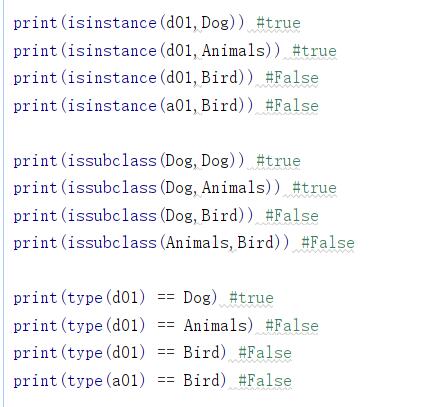
练习： 创建父类：动物(吃)



创建子类：狗(跑)，鸟类(飞)，体会子类复用父类方法



体会 isinstance 、issubclass 与 type 的作用.



##### 继承数据

1. 代码

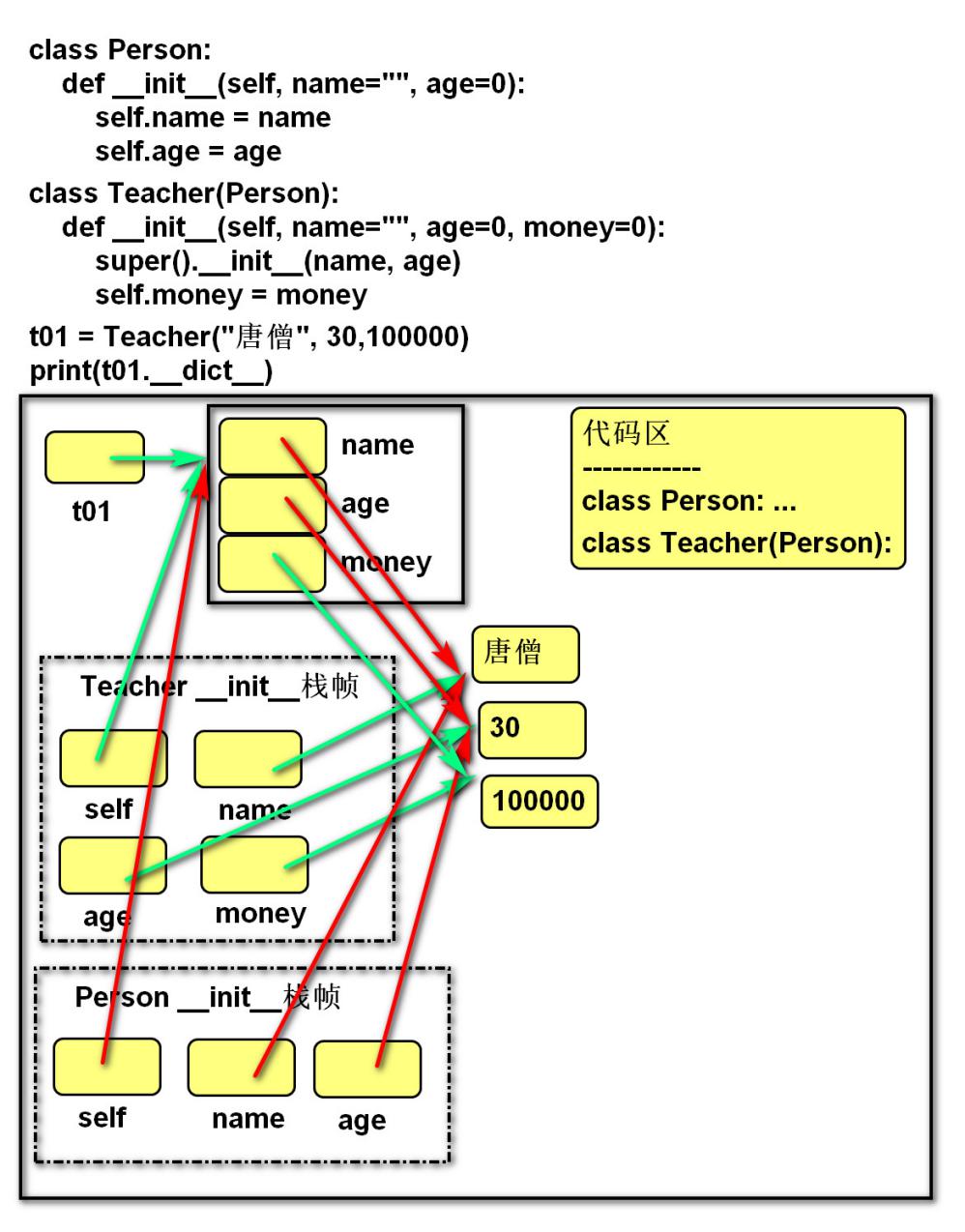
class 子类(父类):

def \_\_init\_\_(self,参数列表):

super().\_\_init\_\_(参数列表)

self.自身实例变量 = 参数

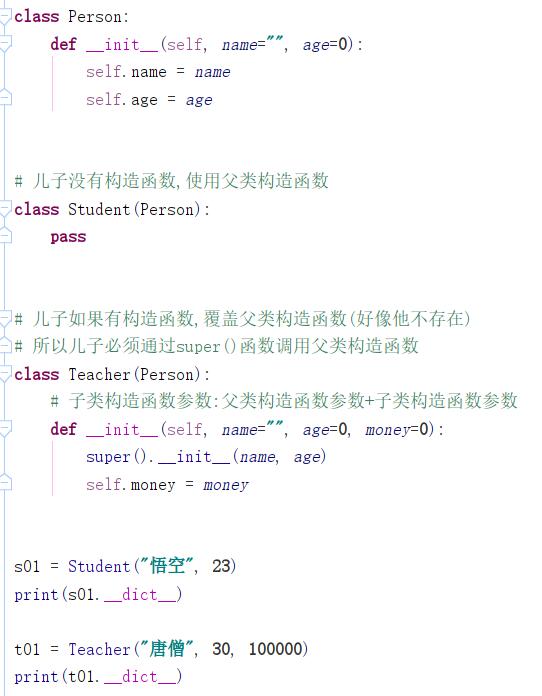
**继承关系内存图**



1. 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。

例题

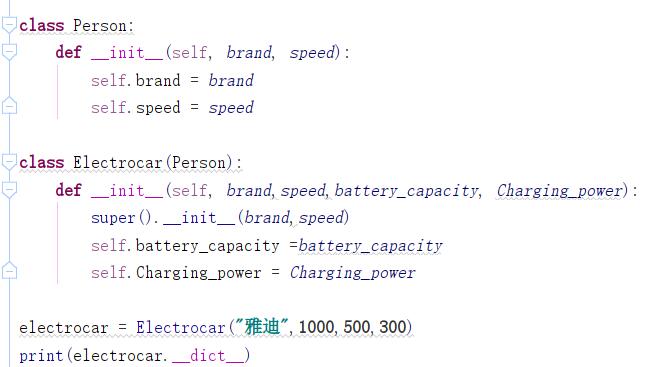


练习：

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。



##### 定义

重用现有类的功能，并在此基础上进行扩展。

说明：子类直接具有父类的成员（共性），还可以扩展新功能。

##### 优点

一种代码复用的方式。

##### 缺点

耦合度高：父类的变化，直接影响子类。

#### 设计角度讲

##### 定义

将相关类的共性进行抽象，统一概念，隔离变化。

##### 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#）。

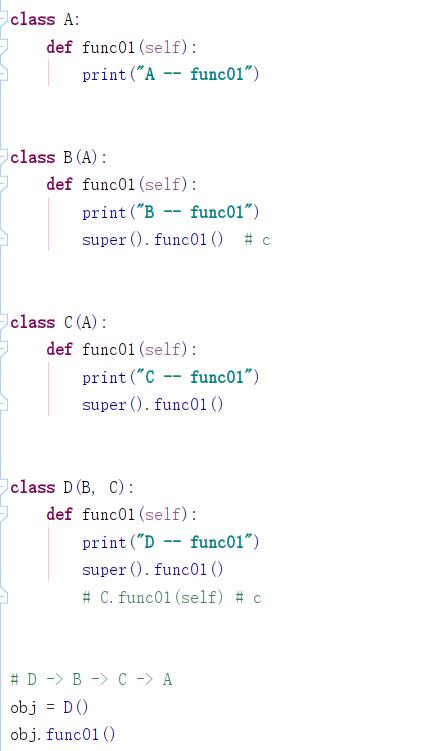
多继承：父类有多个（例如C++，Python）。

**Object类：任何类都直接或间接继承自 object 类**。

#### 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。

同名方法解析顺序:继承列表顺序(广度有限)  
 如果调用指定父类的同名方法:  
 父类名.同名方法名(self)



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

在项目中什么情况使用多继承：继承是在隔离某一维度上的变化，实现行为的统一。

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()

### 多态(个性)

#### 设计角度讲16

##### 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。

##### 作用

1. 在继承的基础上，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。

#### 语法角度讲

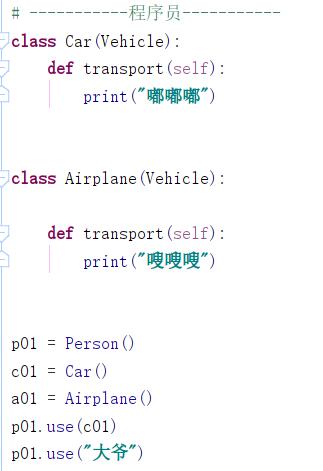
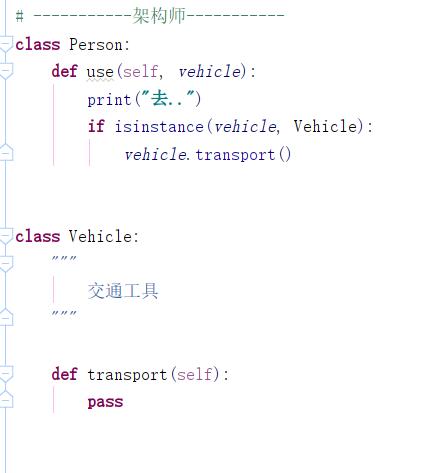
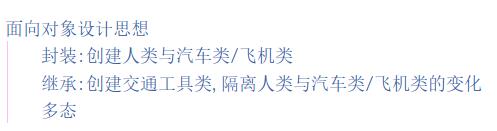
##### 重写

子类实现了父类中相同的方法（方法名、参数）。如果不写父类，默认调用object

在调用该方法时，**实际执行**的是**子类的方法**。

##### 快捷键

Ctrl + O



##### 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。

###### 转换字符串:把对象-->字符串

**\_\_str\_\_函数**：将对象转换为字符串(对人友好的)



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

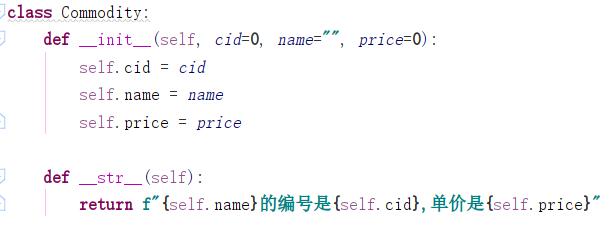
class Commodity:

def \_\_init\_\_(self, cid=0, name="", price=0):

self.cid = cid

self.name = name

self.price = price





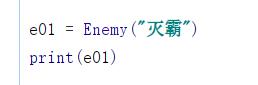
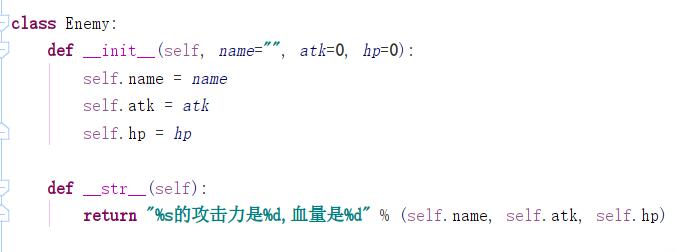
class Enemy:

def \_\_init\_\_(self, name="", atk=0, hp=0):

self.name = name

self.atk = atk

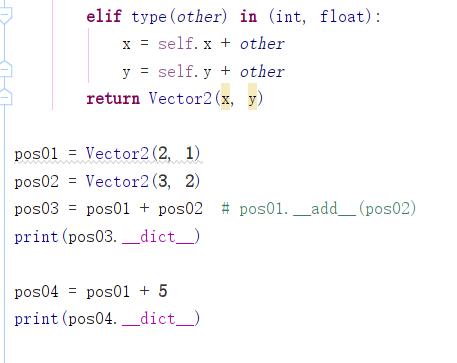
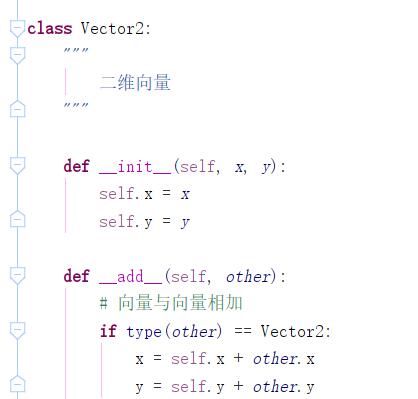
self.hp = hp



###### 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**



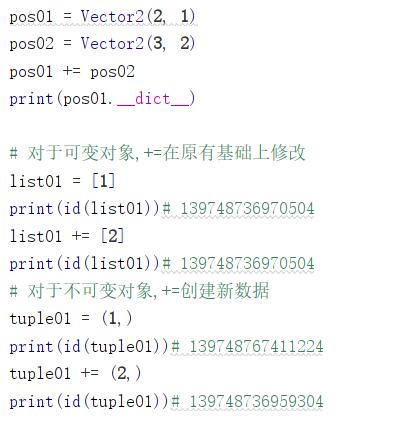
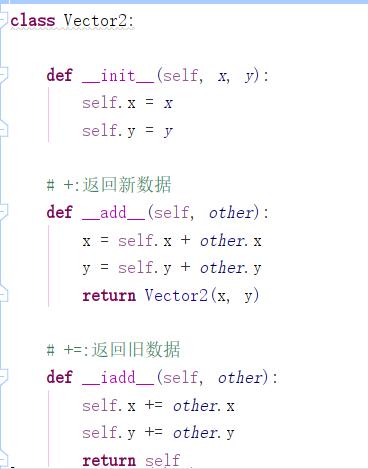
算数运算符



复合运算符重载



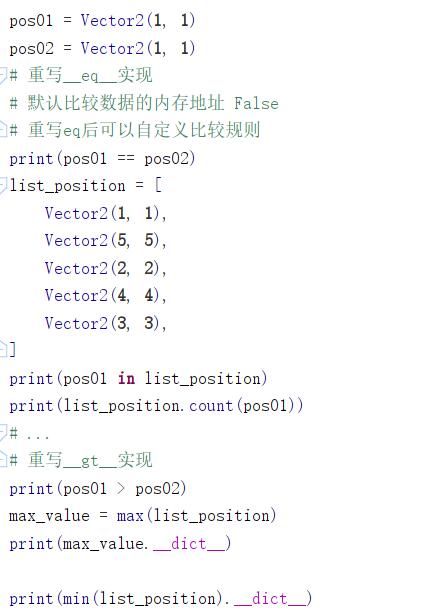
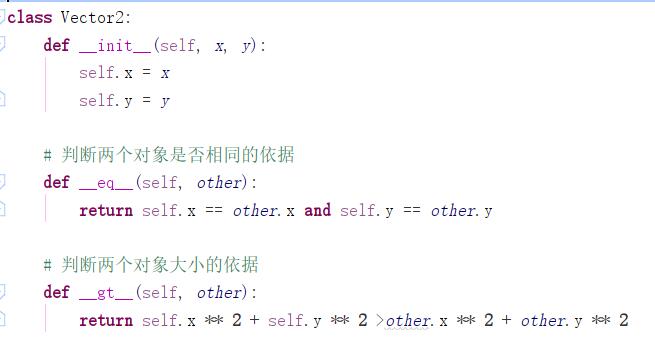
复合运算符重载



比较运算重载



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）-->对封装的解释

**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）-->引出了继承

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

客户端代码(调用的类)尽量依赖(使用)抽象。

抽象不应该依赖细节，细节应该依赖抽象。

### 组合复用原则（复用的最佳实践）-->调用类的方式（组合、继承)

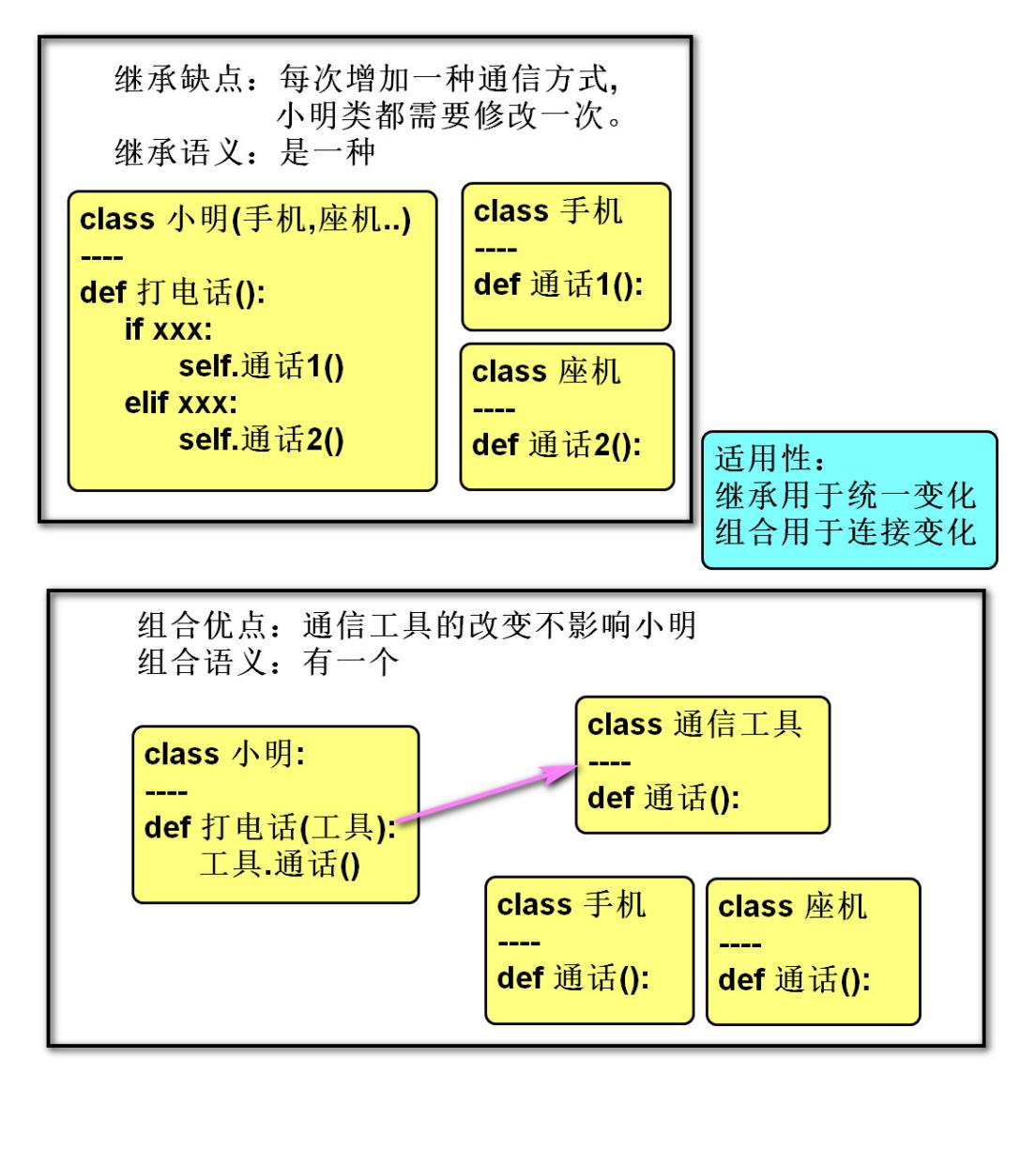
Composite Reuse Principle

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。

**优点**：抽象类的改变不会影响客户端代码(调用的类)

**适用性：（继承：用于统一变化）,组合：用于连接变化(“桥”)**

****

### 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。形参父，实参子

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。利用super().调用父类的功能，然后在此基础上扩展子类自己的功能。

### 迪米特法则（类与类交互的原则）-->软件设计思想，找“爹”隔离，也叫低耦合

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。

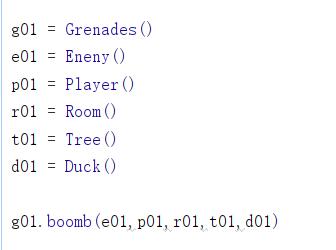
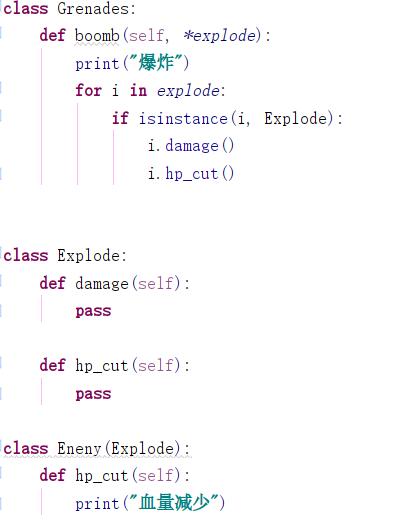
练习1：以面向对象思想，描述下列情景：

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

1. 记录多种图形（圆形、矩形....）

2. 提供计算总面积的方法.

满足：

开闭原则

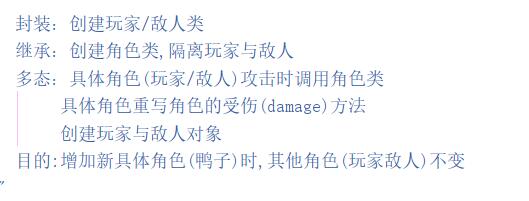
测试：

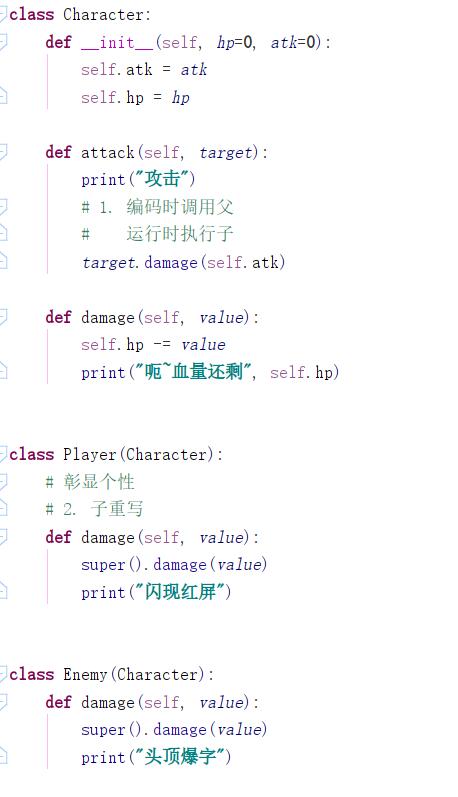
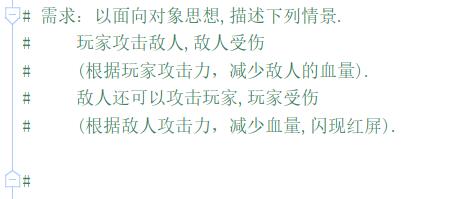
创建图形管理器，存储多个图形对象。

通过图形管理器，调用计算总面积方法.

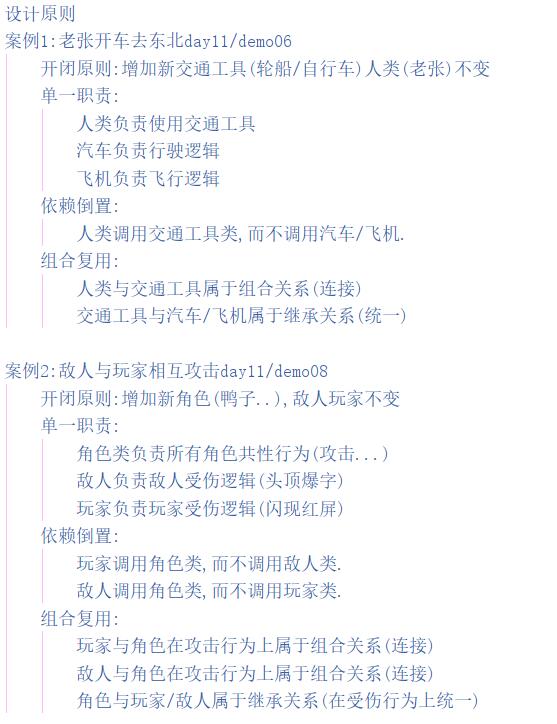
练习3：以面向对象思想，描述下列情景：

玩家攻击敌人,敌人受伤  
(根据玩家攻击力，减少敌人的血量).  
敌人还可以攻击玩家,玩家受伤  
(根据敌人攻击力，减少血量,闪现红屏).



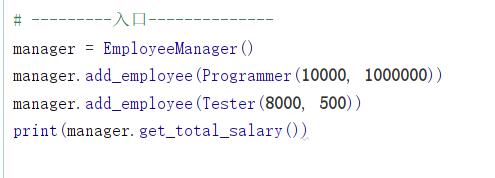
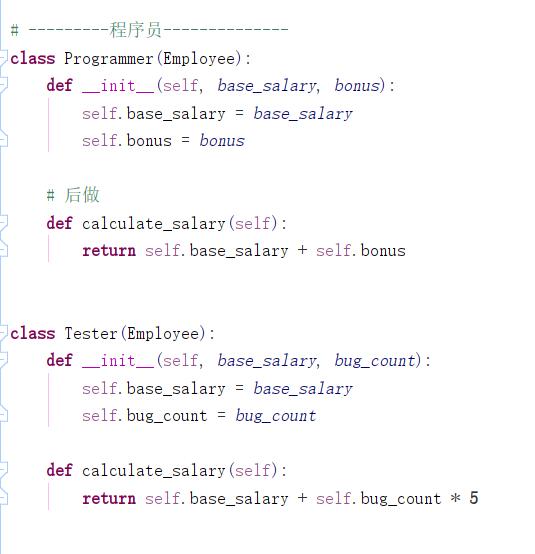
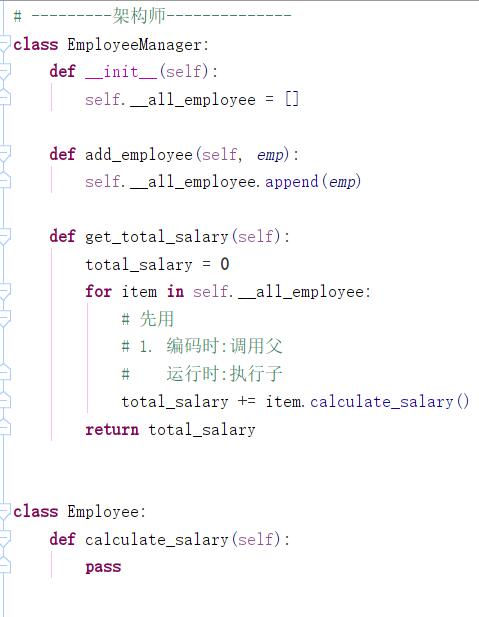
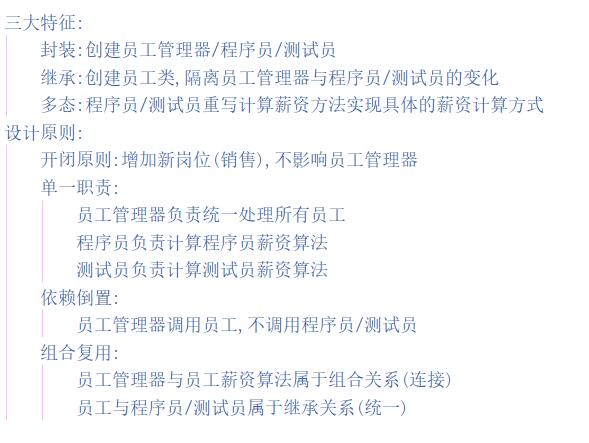


开闭原则，类的单一职责，依赖倒置，组合复用原则的运用



练习4：以面向对象思想，描述下列情景：

有一个公司有如下几种岗位  
 程序员:底薪 + 项目分红  
 测试员:底薪 + Bug \* 5  
 ...  
创建员工管理器  
 1. 记录所有员工  
 2. 提供计算总薪资的功能



练习5：学生信息管理系统  
  
 商品信息管理系统  
 1. 添加商品信息.  
 2. 存储商品信息.  
 3. 显示商品信息  
 4. 删除商品信息  
 5. 修改商品信息

