# 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”。
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。

### 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。

#### 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)

实例成员用对象地址去访问

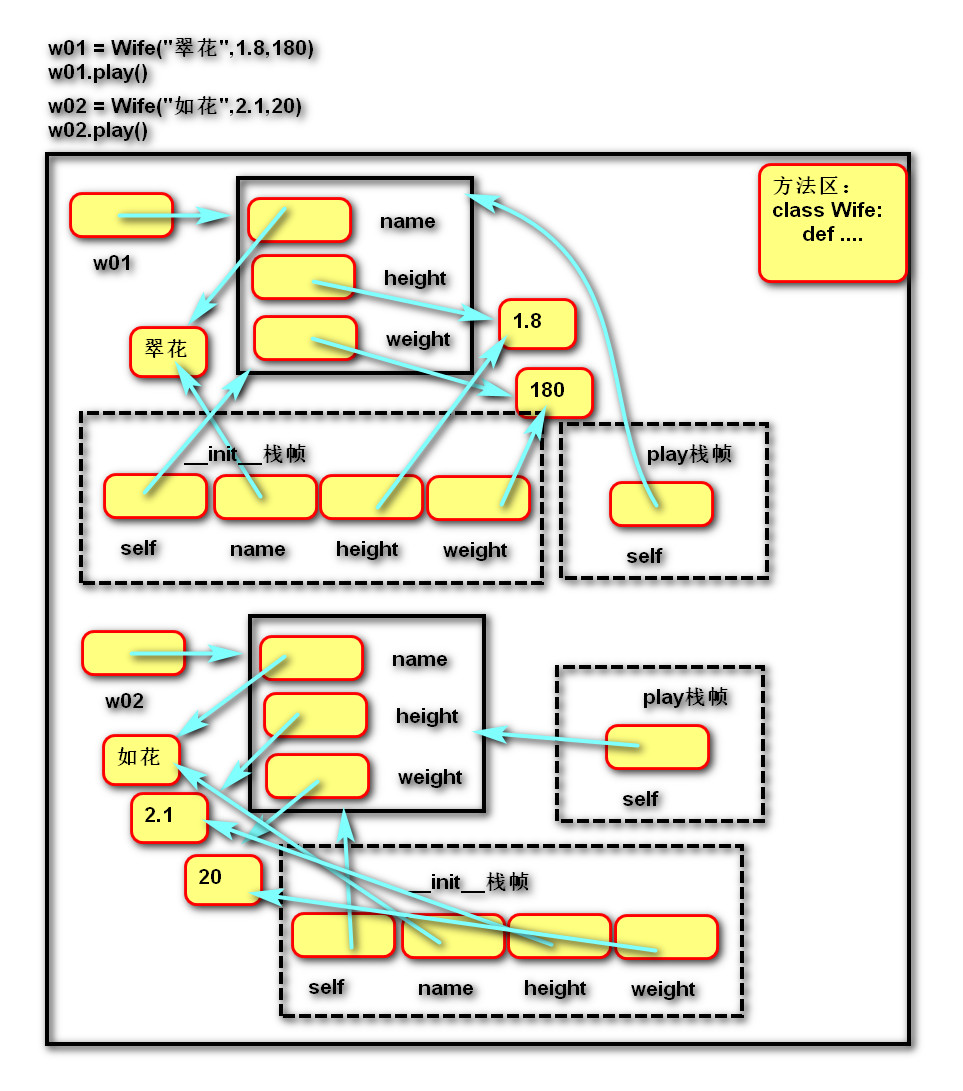
不建议通过类名访问实例方法，一定要用实例去访问

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:抽象的概念 类别  
 水果类成员:  
 数据(名词)成员:重量/水分/味道/价格...  
 行为(动词)成员:生长/腐烂...  
  
 狗类成员:  
 数据(名词)成员:重量/体味/高度...  
 行为(动词)成员:尿尿/吃/叫...  
  
 对象:具体的实例 个体  
 水果类对象:香蕉/苹果/哈密瓜  
 狗类对象:拉布拉多/金毛  
  
 类与类行为不同,对象与对象数据不同.  
  
"""*class Wife:  
 *"""  
 老婆类 ---- 抽象  
 """* # 数据  
 def \_\_init\_\_(self, name, height, weight):  
 self.height = height  
 self.weight = weight  
 self.name = name  
  
 # 行为  
 def play(self):  
 # 方法可以访问数据  
 print(self.name + "在玩耍")  
  
  
# 创建对象  
# 类名(参数...) 调用\_\_init\_\_方法  
w01 = Wife("翠花", 1.8, 180)  
w01.play() # 通过对象地址调用方法,会自动传递对象地址.  
  
w02 = Wife("如花", 2.1, 20)  
w02.play()  
# 同一个方法,可以访问不同的对象数据(方法中self指向了不同对象).

*"""  
 实例成员  
 记住一句话:实例成员,使用对象地址访问.  
 练习:exercise01.py  
 练习:exercise02.py  
"""*class Wife:  
 pass  
  
w01 = Wife()  
# 定义实例变量: 对象.变量名 = ?  
w01.name = "赵敏"  
print(w01.name)  
  
w02 = Wife()  
# print(w02.name)# 错误.因为w02指向的对象,没有创建过实例变量name  
print(w01.\_\_dict\_\_) # 通过\_\_dict\_\_获取当前对象的所有实例变量  
print(w02.\_\_dict\_\_)  
  
class Wife2:  
 def \_\_init\_\_(self, name, height):  
 self.name = name  
 self.height = height  
  
 # 实例方法  
 def print\_self(self):  
 print(self.name, self.height)  
  
w01 = Wife2("灭绝", 2.3)  
w02 = Wife2("金毛狮王", 3.3)  
# 建议:实例方法,通过对象地址访问.  
w01.print\_self()  
# 不建议:Wife2.print\_self()# 没有传递对象地址,实例方法不能正确访问对象数据.  
Wife2.print\_self(w02)

### 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

(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。

#### 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

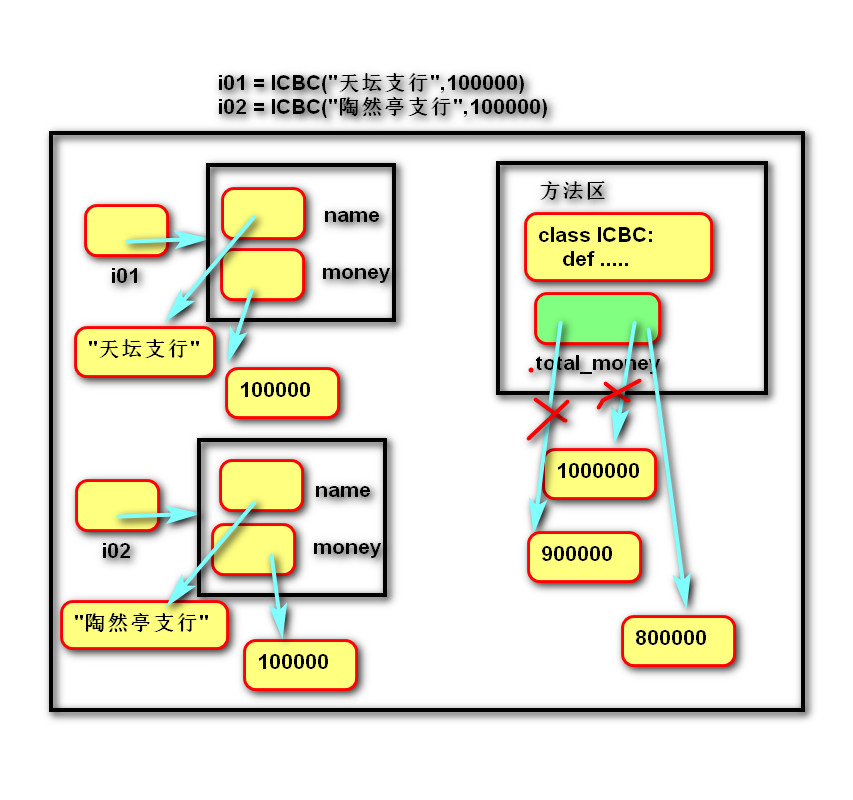
(1) 至少有一个形参，第一个形参用于绑定类，一般命名为'cls'

(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。

*"""  
 类成员  
 15:25  
"""*class ICBC:  
 # 类变量:总行的钱  
 total\_money = 1000000  
  
 # 类方法  
 @classmethod  
 def print\_total\_money(cls):  
 print(id(cls), id(ICBC))  
 # cls : 存储当前类的地址  
 # print("当前总行金额:",ICBC.total\_money)  
 print("当前总行金额:", cls.total\_money)  
  
 def \_\_init\_\_(self, name, money):  
 self.name = name  
 self.money = money  
 # 从总行扣除当前支行的钱  
 ICBC.total\_money -= money  
  
  
i01 = ICBC("天坛支行", 100000)  
i02 = ICBC("陶然亭支行", 100000)  
# 主流:通过类访问类成员  
ICBC.print\_total\_money()  
print(ICBC.total\_money)  
# 非主流:通过对象访问类成员  
# print(i02.total\_money)  
# i02.print\_total\_money()



### 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。

## 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：

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。

*"""  
 封装  
 数据: 老婆(名字,高度,体重) 学生(姓名,成绩,性别,年龄)  
 字典(名称,单价...)  
 行为: 必要  
"""*class Wife:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 # 私有成员:以双下划线开头  
 # self.\_\_age = age  
 self.set\_age(age)  
  
 def get\_age(self):  
 return self.\_\_age  
  
 def set\_age(self, value):  
 if 20 <= value <= 50:  
 self.\_\_age = value  
 else:  
 raise ValueError("我不要")  
  
  
w01 = Wife("小乔", 25)  
# 不能访问私有变量  
# print(w01.\_\_age)  
# w01.set\_age(30)  
print(w01.get\_age())

*"""  
 封装  
 行为: property  
 练习:exercise05.py  
"""*class Wife:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 # self.set\_age(age)  
 self.age = age  
  
 def get\_age(self):  
 return self.\_\_age  
  
 def set\_age(self, value):  
 if 20 <= value <= 50:  
 self.\_\_age = value  
 else:  
 raise ValueError("我不要")  
  
 # 类变量  
 # property 属性: 在拦截对age的读写操作  
 age = property(get\_age, set\_age)  
  
  
w01 = Wife("小乔", 25)  
# print(w01.get\_age())  
print(w01.age)

1. 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 属性名(self):

return self.\_\_属性名

@属性名.setter

def 属性名(self, value):

self.\_\_属性名= value

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)

*"""  
 封装  
 行为: 标准属性  
 练习:exercise06.py  
 练习:exercise07.py  
"""*# 读写属性age  
class Wife:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 @property # 拦截读取 age = property(读取方法,None)  
 def age(self):  
 return self.\_\_age  
  
 @age.setter # 拦截写入 age.setter(写入方法)  
 def age(self, value):  
 if 20 <= value <= 50:  
 self.\_\_age = value  
 else:  
 raise ValueError("我不要")  
  
  
w01 = Wife("小乔", 25)  
# print(w01.get\_age())  
print(w01.age)  
  
  
# ---------------------------  
# 只读属性age  
class Wife2:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.\_\_age = 23  
  
 @property # 拦截读取 age = property(读取方法,None)  
 def age(self):  
 return self.\_\_age  
  
  
w02 = Wife2("大桥")  
  
  
# w02.age = 30  
# ---------------------  
# 只写属性age  
class Wife3:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.age = 23  
  
 def set\_age(self, value):  
 if 20 <= value <= 50:  
 self.\_\_age = value  
 else:  
 raise ValueError("我不要")  
  
 age = property(None, set\_age)  
  
  
w03 = Wife3("大桥")  
w03.age = 30  
# print(w03.age)# 不能读取

#### 设计角度讲

1. 定义：

(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。

(2) 变则疏之

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)。

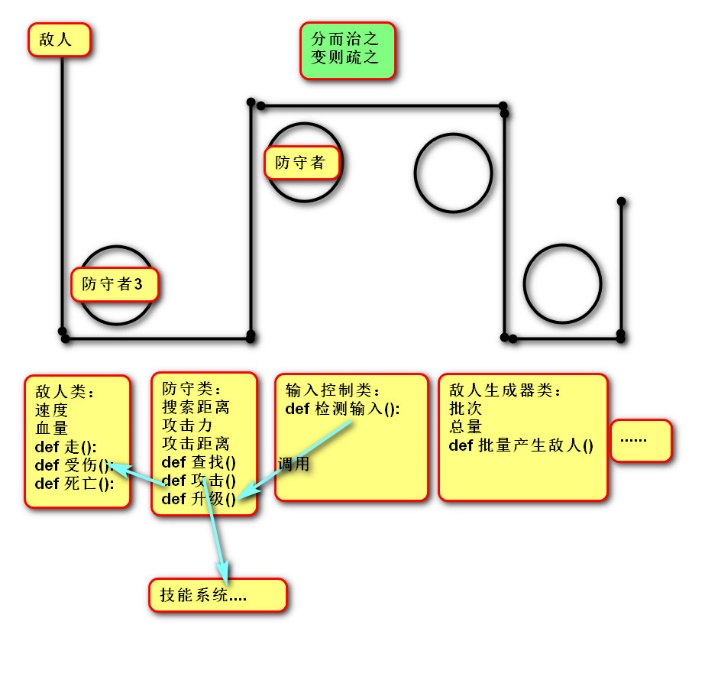
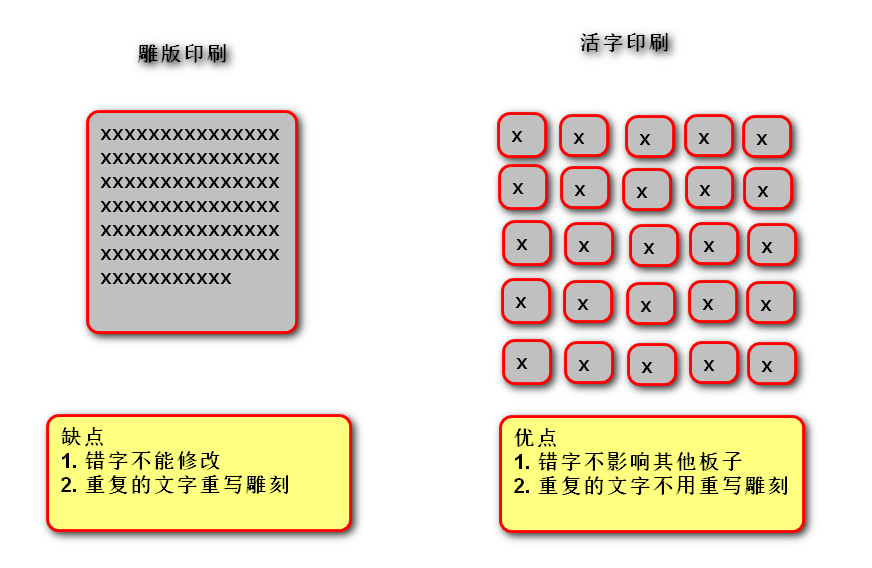
(4) 低 耦 合

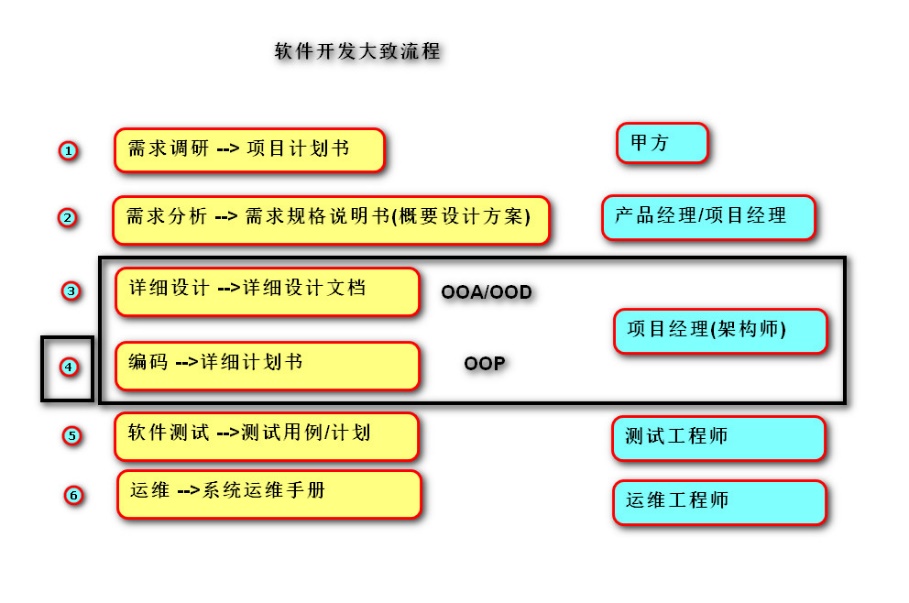
类与类的关联性与依赖度要低(每个类独立)，让一个类的改变，尽少影响其他类。

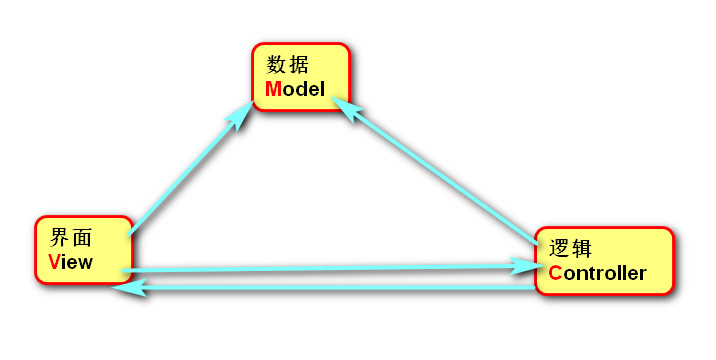
1. 优势：

便于分工，便于复用，可扩展性强。

*"""  
 封装  
 设计思想  
"""*# 以面向对象的思想,描述下列场景:  
# 老张开车去东北.  
class Person:  
 def \_\_init\_\_(self, name=""):  
 self.name = name  
  
 # 需求:人的go\_to方法,调用车的run方法  
 # 技术:实例方法 -- 通过对象调用  
 def go\_to(self, str\_position, type):  
 print("去:", str\_position)  
 type.run()  
  
  
class Car:  
 def run(self):  
 print("走你....")  
  
  
c01 = Car()  
lz = Person("老张")  
lz.go\_to("东北", c01)







#### 案例:信息管理系统



##### 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

### 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

isinstance(对象, 类型)

返回指定对象是否是某个类的对象。

issubclass(类型，类型)

返回指定类型是否属于某个类型。

注意与Type()区分

*"""  
 继承  
 财产:钱不用儿子挣,但是可以花.  
 皇位:江山不用太子打,但是可以坐.  
 编程:代码不用子类写,但是可以用.  
"""*class Person:  
 def say(self):  
 print("说话")  
  
class Student(Person):  
 def study(self):  
 print("学习")  
  
class Teacher(Person):  
 def teach(self):  
 print("讲课")  
  
# 创建父类型对象,只能访问父类型成员  
p01 = Person()  
p01.say()  
  
# 创建子类型对象,能访问父类型成员,还能访自身成员  
s01 = Student()  
s01.say()  
s01.study()  
  
# "是不是 实例"  
# s01的对象 是 一种 Student类型  
print(isinstance(s01, Student)) # True  
print(isinstance(s01, Person)) # True  
print(isinstance(s01, Teacher)) # False  
  
# s01的类型 等于 Student类型  
print(type(s01) == Student) # True  
print(type(s01) == Person) # False  
  
# Student 类型 是一种 Person类型  
print(issubclass(Student, Person)) # True  
print(issubclass(Student, Student)) # True  
print(issubclass(Student, Teacher)) # False  
  
# 练习:定义父类(动物,行为--吃)  
# 定义子类(狗,行为-- 跑)  
# 定义子类(鸟,行为-- 飞)  
# 分别创建父类子类对象,调用其行为.  
# 体会isinstance type issubclass三者之间的关系

##### 继承数据

1. 代码

class 子类(父类):

def \_\_init\_\_(self,参数列表):

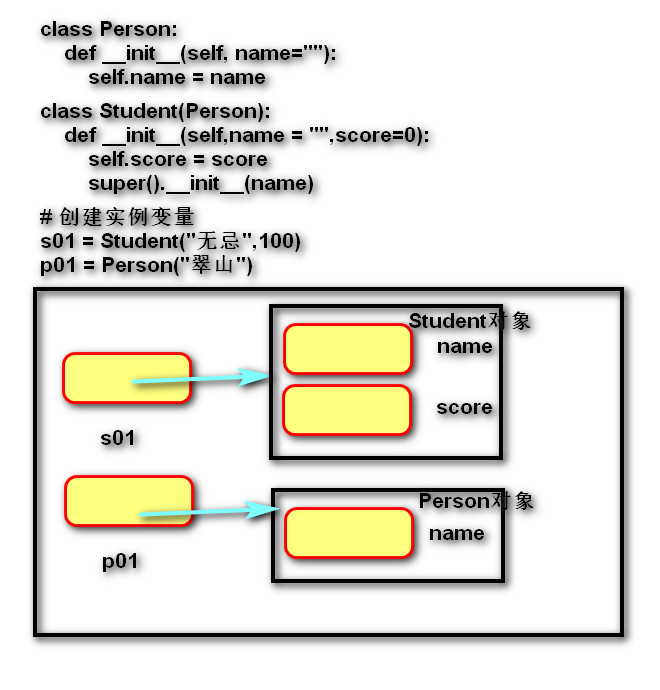
super().\_\_init\_\_(参数列表)

self.自身实例变量 = 参数

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。

*"""  
 继承 -- 数据  
  
"""*class Person:  
 def \_\_init\_\_(self, name=""):  
 self.name = name  
  
class Student(Person):  
 def \_\_init\_\_(self, name="", score=0):  
 self.score = score  
 # self.name = name  
 # 如果子类没有构造函数,使用父类构造函数  
 # 如果子类有构造函数,必须铜通过super()调用父类构造函数,  
 # 否则会覆盖父类(不执行)的.  
 super().\_\_init\_\_(name)  
  
  
# 创建实例变量  
s01 = Student("无忌", 100)  
print(s01.name)  
print(s01.score)  
p01 = Person("翠山")  
print(p01.name)



##### 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

### 多态

#### 设计角度讲

##### 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

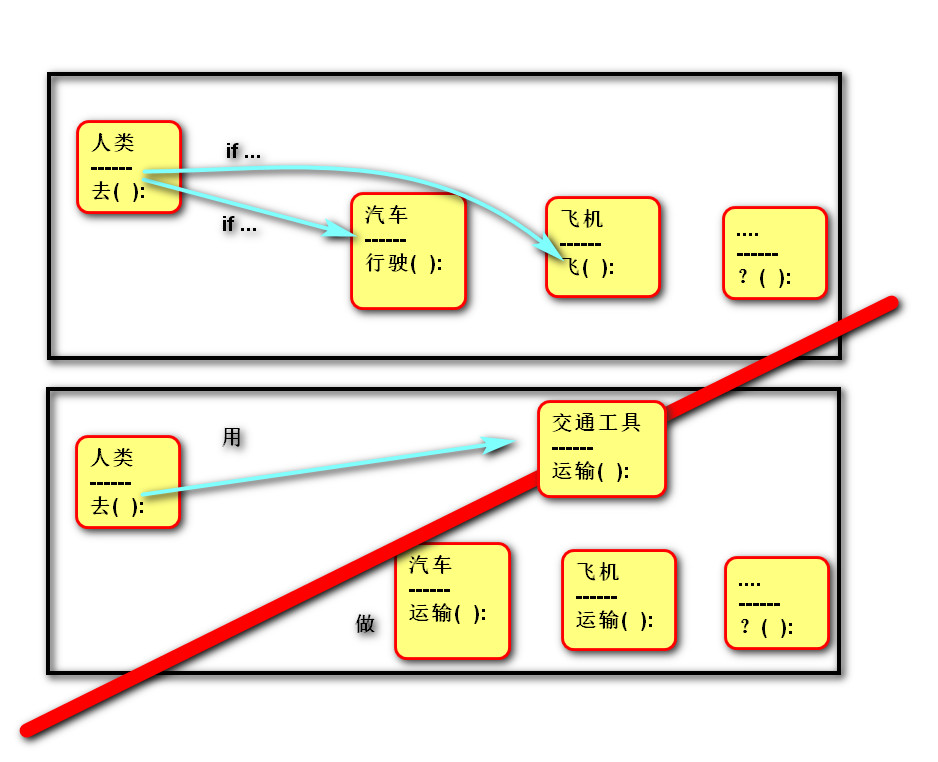
子类实现了父类中相同的方法（方法名、参数）。

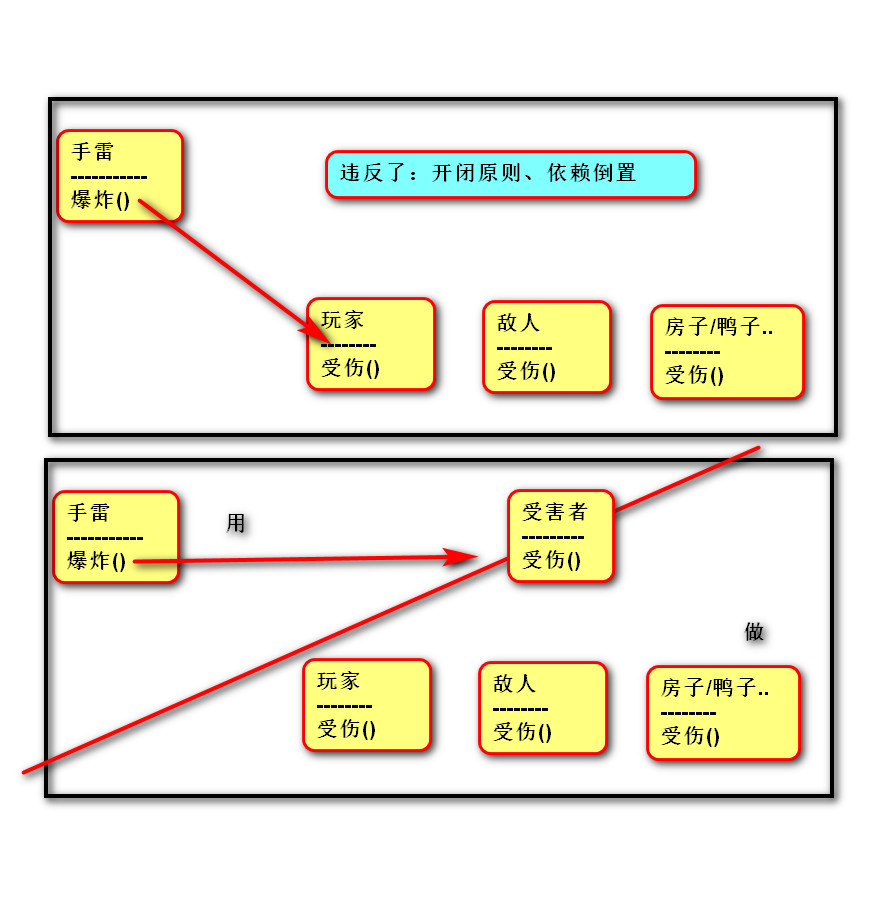
在调用该方法时，实际执行的是子类的方法。

##### 快捷键

Ctrl + O

*"""  
 继承 -- 设计(2)  
 练习:exercise02  
"""*# 老张开车去东北.  
# 需求变化:还可能坐飞机,坐火车......  
  
class Person:  
 def \_\_init\_\_(self, name=""):  
 self.name = name  
  
 def go\_to(self, str\_position, vehicle):  
 print("去:", str\_position)  
 vehicle.transport()  
  
class Vehicle:  
 *"""  
 抽象的  
 交通工具  
 """* def transport(self):  
 pass  
  
# -------------------------------------  
class Car(Vehicle):  
  
 def transport(self):  
 print("走你...")  
  
class Airplane(Vehicle):  
  
 def transport(self):  
 print("嗖嗖嗖.")  
  
  
c01 = Car()  
a01 = Airplane()  
lz = Person("老张")  
lz.go\_to("东北", a01)





##### 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

\_\_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(重写)

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。

*"""  
 继承 -- 设计(1)  
"""*# 老张开车去东北.  
# 需求变化:还可能坐飞机,坐火车......  
  
# 违背了面向对象设计原则:  
# 开闭原则 / 依赖倒置  
class Person:  
 def \_\_init\_\_(self, name=""):  
 self.name = name  
  
 # 对方法增加语句,删除语句,修改语句都视为改变代码.  
 def go\_to(self, str\_position, vehicle):  
 print("去:", str\_position)  
 if type(vehicle) == Car:  
 vehicle.run()  
 elif type(vehicle) == Airplane:  
 vehicle.flay()  
  
  
class Car:  
 def run(self):  
 print("走你....")  
  
  
class Airplane:  
 def flay(self):  
 print("嗖嗖...")  
  
  
c01 = Car()  
a01 = Airplane()  
lz = Person("老张")  
lz.go\_to("东北", c01)

### 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。

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）

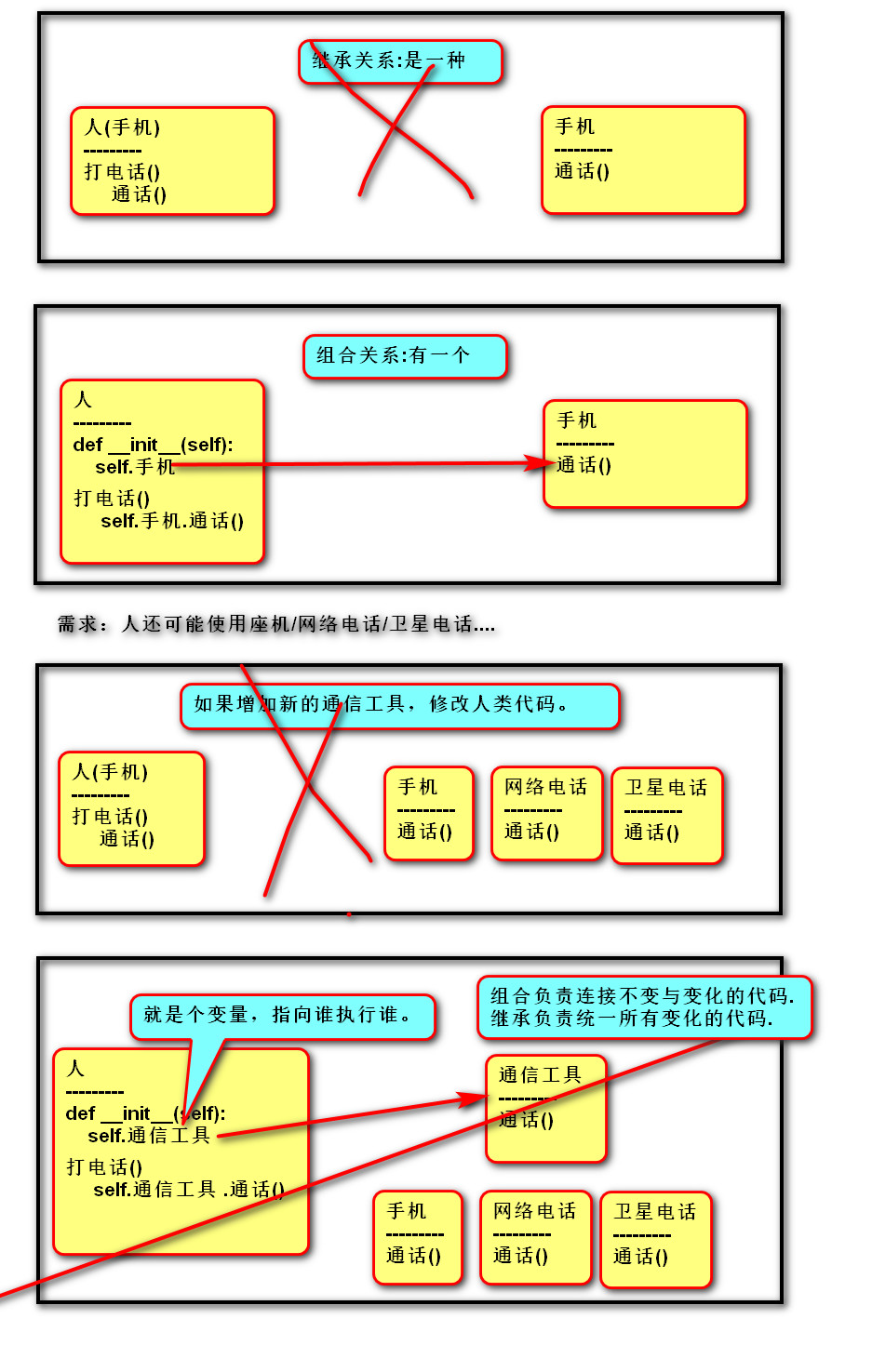
Composite Reuse Principle

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。

不变的事物与变的事物之间的链接

继承负责统一所有变化的代码



### 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。

如果A类要调用B类中的某一个方法，应该将B类加以抽象得到C类，A类直接调用抽象类C里的那个功能。

不变和变化之间用父类隔离。

复习：

"""

10:50

面向对象三大特征:

封装:

语法:

数据:用一个类将多个变量包装起来.

class 类名:

def \_\_init\_\_(self,参数):

self.数据1 = 参数

self.数据2 = 参数

self.数据3 = 参数

def 方法1(self):

操作数据的逻辑

def \_\_方法2(self):

操作数据的逻辑

变量 = 类名(参数)

变量.数据1

变量.数据2

变量.数据3

变量.方法1()

行为: 对外隐藏实现功能的细节

注意:对外要提供必要的功能

设计:

分而治之,变则疏之.

继承:

语法:

class 爸爸:

def \_\_init\_\_(self,参数1):

self.数据1 = 参数1

功能1

class 儿子(爸爸):

def \_\_init\_\_(self,参数1,参数2):

super().\_\_init\_\_(参数1)

self.数据2 = 参数2

功能2

变量 = 儿子()

变量.功能1()

变量.功能2()

设计:

抽象具体事物,统一概念(约束子类),隔离变化

多态:

语法:依赖重写实现

重写:儿子具有和爸爸相同的方法,实际执行的是儿子的方法(覆盖)

做法:重写 + 创建子类对象 --> 调用父执行子

class 爸爸:

def 功能1(self):

.....

class 儿子(爸爸):

def 功能1(self):

.....

# 创建子类对象

变量 = 儿子()

变量.功能1() # 执行儿子的功能

# 不是多态(创建父类对象)

变量 = 爸爸()

变量.功能1() # 执行爸爸的功能

设计:调用父类方法,执行子类方法(创建子类对象).

# 做函数使用父类

def 函数(爸爸):

爸爸.功能1()

# 用函数使用儿子

函数(儿子())# 传递到函数中的是子类对象

面向对象设计原则:

开闭原则:允许增加新功能,不能修改以前的代码.

单一职责:一个类处理一个变化点

依赖倒置:调用父,而不调用子.

组合复用:通过变量(参数/实例变量)调用,而不是通过继承调用.

案例1:

老张开车/坐飞机...去东北

# 违背了:开闭原则,依赖倒置

class 人:

def 去(飞机):

if 是飞机:

....

elif 是车:

...

------------架构师---------------

class 人:

def 去(交通工具):

# 现象:调用父类,执行子类.

交通工具.运输()

class 交通工具:

def 运输():

...

------------程序员---------------

# 做法1:重写

class 飞机(交通工具):

def 运输():

...

class 车:

def 运输():

...

------------测试---------------

变量1 = 人()

# 做法2:创建的是子类对象

变量2 = 飞机()

变量1.去(变量2)

案例2:

手雷爆炸伤害玩家/敌人.

封装:手雷/玩家/敌人

继承:受害者隔离了手雷爆炸的逻辑与玩家或者敌人受伤的逻辑

多态:手雷爆炸依赖受害者,实际执行玩家与敌人.

开闭:增加/减少新受害者,手雷不用改变.

单一:手雷(爆炸) 受害者(隔离变化) 玩家(受伤逻辑) 敌人(受伤逻辑)

依赖倒置:手雷依赖受害者,不依赖玩家/敌人.

组合复用:手雷的爆炸方法,通过参数"受害者"访问玩家/敌人"受伤"方法

------------架构师---------------

class 手雷:

def 爆炸(受害者):

受害者.受伤()

class 受害者:

def 受伤():

...

------------程序员---------------

class 玩家(受害者):

def 受伤():

...

class 敌人(受害者):

def 受伤():

...

------------测试---------------

变量1 = 手雷()

变量2 = 玩家()

变量1.爆炸(变量2)

案例3:

图形管理器管理圆形/矩形.

封装:图形管理器/圆形/矩形

继承:图形隔离了图形管理器计算总面积的逻辑与圆形/矩形获取面积的逻辑

多态:图形管理器依图形,实际执行圆形,矩形.

开闭:增加/减少新图形,图形管理器不用改变.

单一:图形管理器(管理图形) 图形(隔离变化) 圆形(获取面积) 矩形(获取面积)

依赖倒置:图形管理器依赖图形,不依赖圆形/矩形.

组合复用:图形管理器的计算总面积方法,通过实例变量"所有图形"访问圆形/矩形"获取面积"方法

------------架构师---------------

class 图形管理器:

def \_\_init\_\_():

self.所有图形 = []

def 计算总面积():

for 图形 in self.所有图形:

累加 图形.获取面积()

class 图形:

def 获取面积():

...

------------程序员---------------

class 圆形(图形):

def 获取面积():

...

class 矩形(图形):

def 获取面积():

...

------------测试---------------

变量1 = 图形管理器()

变量2 = 圆形()

变量1.添加图形(变量2)

变量1.计算总面积()

"""