# 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。

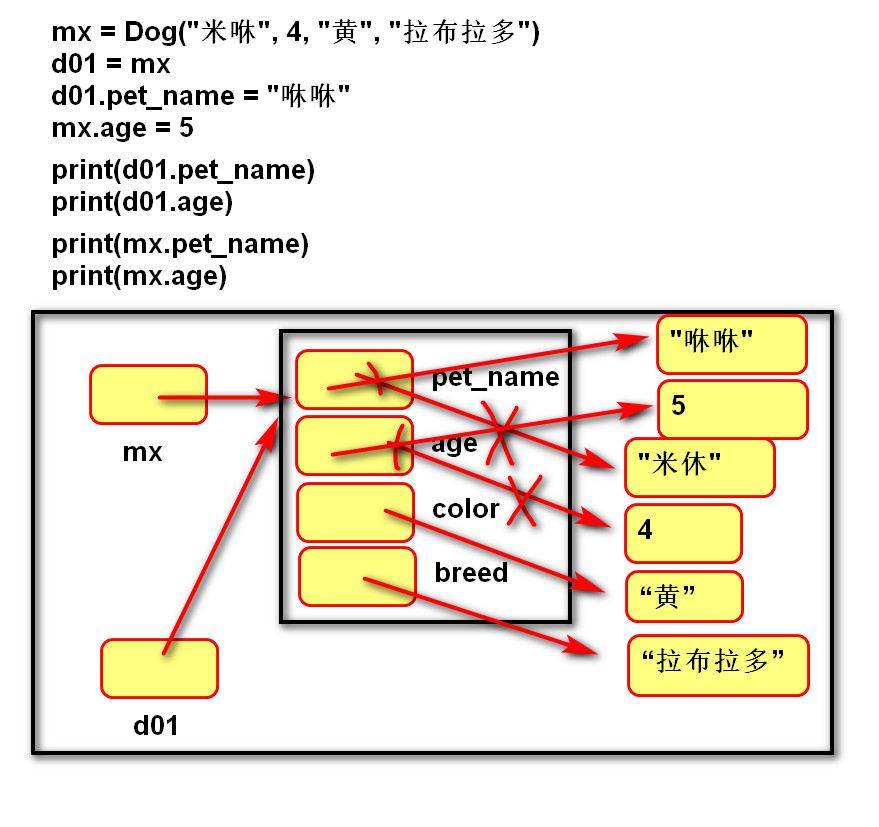
#### 创建对象(实例化)

变量 = 构造函数 (参数列表)

"""

类和对象

现实事物 -抽象化-> 类 -实例化-> 对象

 15：20

"""

class Wife:

"""

抽象的老婆

"""

# 数据：姓名、颜值、钱....

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

self.name = name

self.face\_score = face\_score

self.money = money

# 方法(函数)：玩...

def play(self):

print(self.name, "在玩耍")

jg = Wife("金刚",6,5000)

jg.play()

### 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()

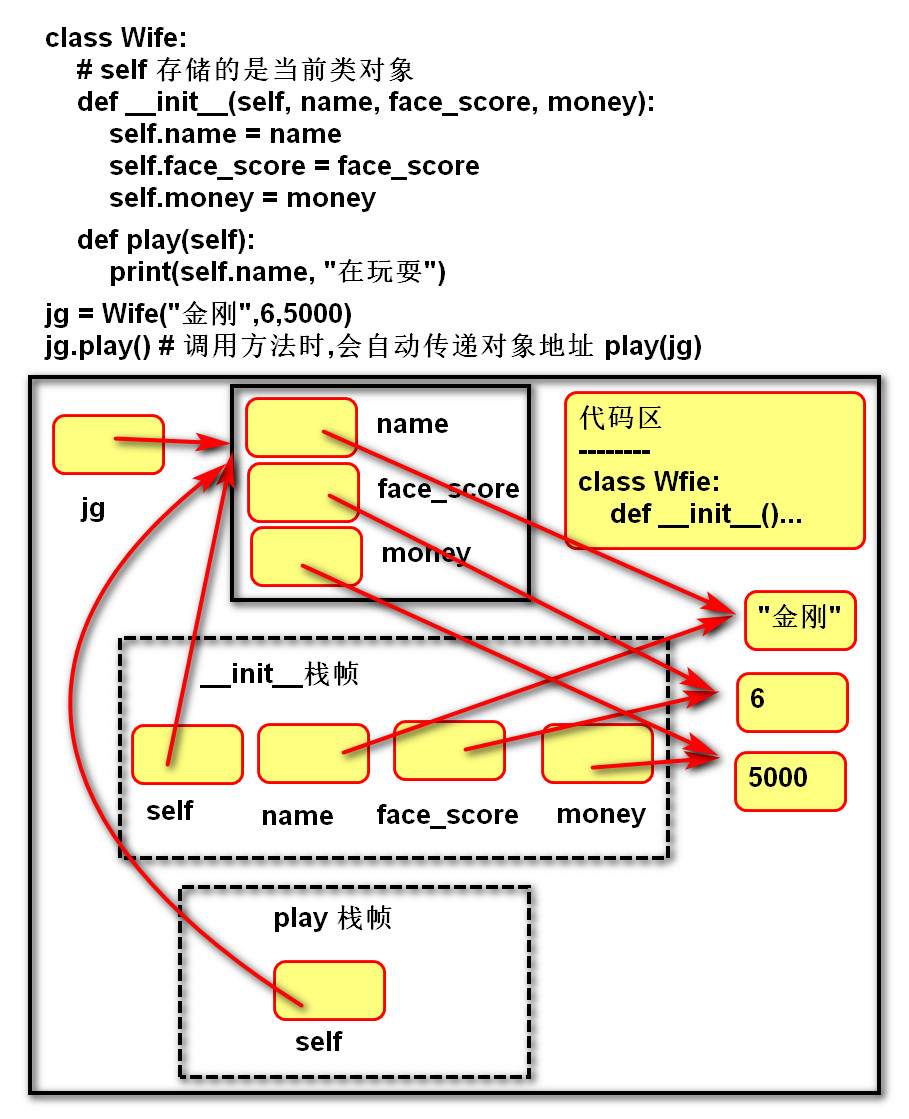
w01.name = “丽丽”

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

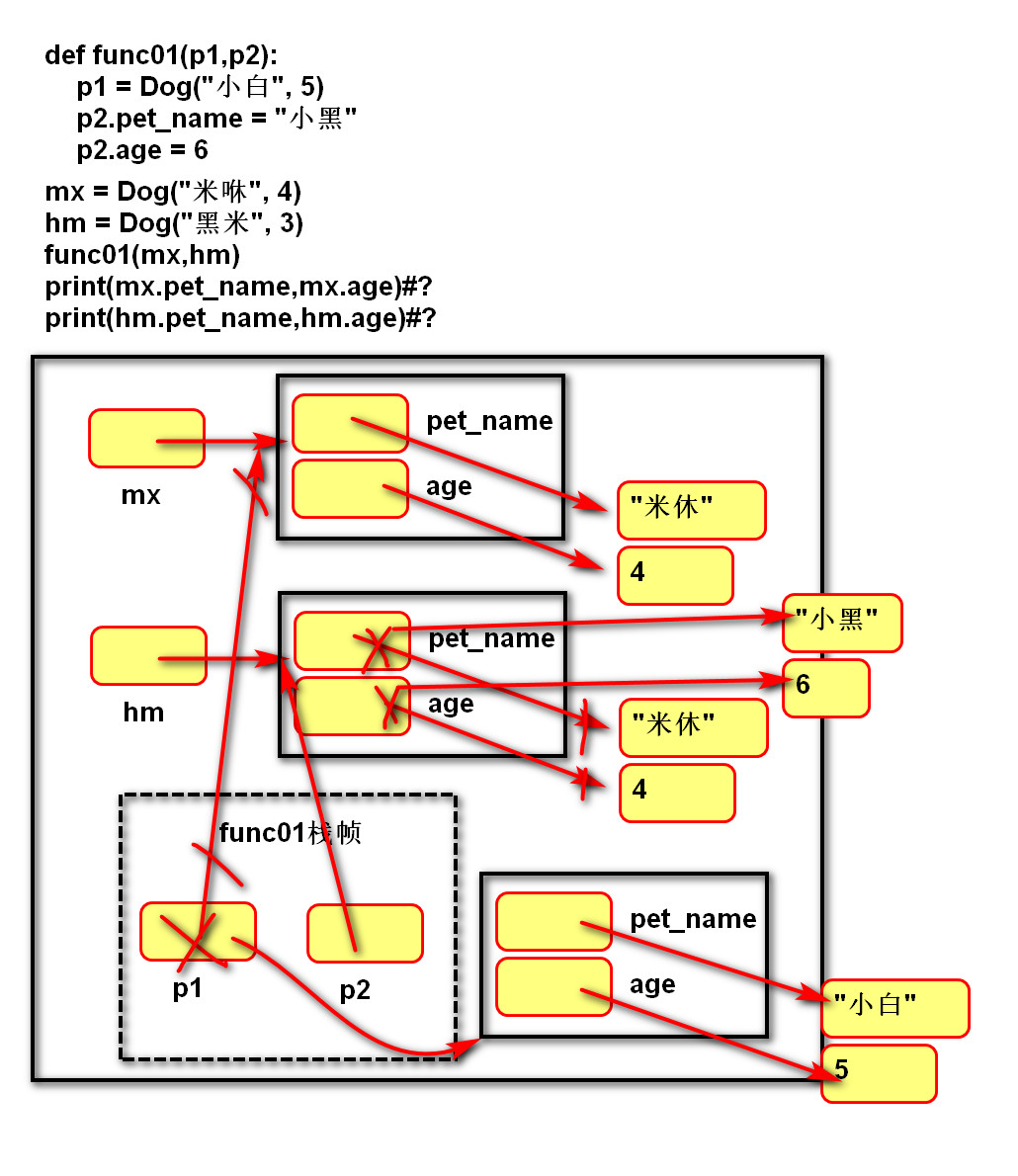
(2) 调用： 对象地址.实例方法名(参数列表)

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。

"""

实例成员

实例变量：对象不同的数据

实例方法：对象相同行为

核心逻辑：

对象.?

"""

# 1. 标准写法

class Wife:

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

# 创建实例变量：对象.变量名 = 数据

self.name = name

def print\_self(self):

print("我叫：", self.name)

def play(self):

print("在玩耍")

# 对象.方法名()

self.print\_self()

w01 = Wife("双儿")

w02 = Wife("阿珂")

# 读取实例变量：? = 对象.变量名

print(w01.name)

# 内置实例变量

# -- 获取当前对象所有实例变量

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

w01.play()

w02.play()

# 2. 不建议(实例变量应该由类的定义者决定)

# class Wife:

# pass

#

# w01 = Wife()

# # 创建实例变量：对象.变量名 = 数据

# w01.name = "双儿"

# # 读取实例变量：? = 对象.变量名

# print(w01.name)

# 3. 不建议(实例变量应该直接在init中定义)

# class Wife:

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

# pass

#

# def func01(self,name):

# # 创建实例变量：对象.变量名 = 数据

# self.name = name

#

# w01 = Wife()

# w01.func01("双儿")

# # 读取实例变量：? = 对象.变量名

# print(w01.name)

dict01 = {}

dict01["name"] = "双儿"

dict01["name"] = "双双"

### 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。

"""

* + - * 1. 类成员
        2. 类变量：大家的数据
        3. 类方法：大家的行为
        4. 核心逻辑：
        5. 类.?
        6. """
        7. class ICBC:
        8. # 总行的钱
        9. total\_money = 1000000
        10. @classmethod
        11. def print\_total\_money(cls):
        12. print("总行的钱：", cls.total\_money)
        13. def \_\_init\_\_(self, name="", money=0):
        14. # 实例变量
        15. self.name = name
        16. # 支行的钱
        17. self.money = money
        18. # 从总行扣除当前支行需要的钱
        19. ICBC.total\_money -= money
        20. i01 = ICBC("天坛支行", 100000)
        21. i02 = ICBC("陶然亭支行", 200000)
        22. # print("总行的钱：",ICBC.total\_money)
        23. ICBC.print\_total\_money() # print\_total\_money(ICBC)

### 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。

"""

1. 静态方法
2. 可以独立存在的工具函数
3. """
4. class Vector2:
5. """
6. 二维向量
7. """
8. def \_\_init\_\_(self, x, y):
9. self.x = x
10. self.y = y
11. @staticmethod
12. def get\_right():
13. return Vector2(0, 1)
14. @staticmethod
15. def get\_up():
16. return Vector2(-1, 0)
17. @staticmethod
18. def get\_left():
19. return Vector2(0, -1)
20. list01 = [
21. ["00", "01", "02", "03"],
22. ["10", "11", "12", "13"],
23. ["20", "21", "22", "23"],
24. ["30", "31", "32", "33"],
25. ]
26. # 位置
27. pos = Vector2(1, 2)
28. # 方向
29. # right = Vector2(0, 1)
30. right = Vector2.get\_right()
31. # 需求：沿着某个方向移动
32. pos.x += right.x
33. pos.y += right.y
34. print(pos.x,pos.y)
35. # 练习1:创建向上的静态方法
36. # 练习2:创建向左的静态方法
37. # 测试：让某个位置沿着该方向移动
38. up = Vector2.get\_up()
39. pos.x += up.x
40. pos.y += up.y
41. print(pos.x,pos.y)

## 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。

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)

"""

python 属性全部语法

核心逻辑：拦截

目标：保护数据

10:30

"""

# 1. 读 + 写属性 快捷键：props + 回车

# class Wife:

# def \_\_init\_\_(self, age=0):

# self.age = age

#

# @property

# def age(self):

# return self.\_\_age

#

# @age.setter

# def age(self, value):

# self.\_\_age = value

#

# w01 = Wife(25)

# print(w01.age)

# 2. 读 属性 快捷键：prop + 回车

# class Wife:

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

# # 数据从何而来,无所谓(可以从参数来、某个方法来、写死).

# self.\_\_age = 25

#

# @property

# def age(self):

# return self.\_\_age

#

# w01 = Wife()

# print(w01.age)

#

# w01.age = 222

# 3. 写 属性 快捷键：prop + 回车

# class Wife:

# def \_\_init\_\_(self, age=0):

# self.age = age

#

# age = property()

# @age.setter# age.setter(..)

# def age(self, value):

# self.\_\_age = value

class Wife:

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

self.age = age

def age(self, value):

self.\_\_age = value

age = property(None,age)

# w01 = Wife(25)

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

# print(w01.age)

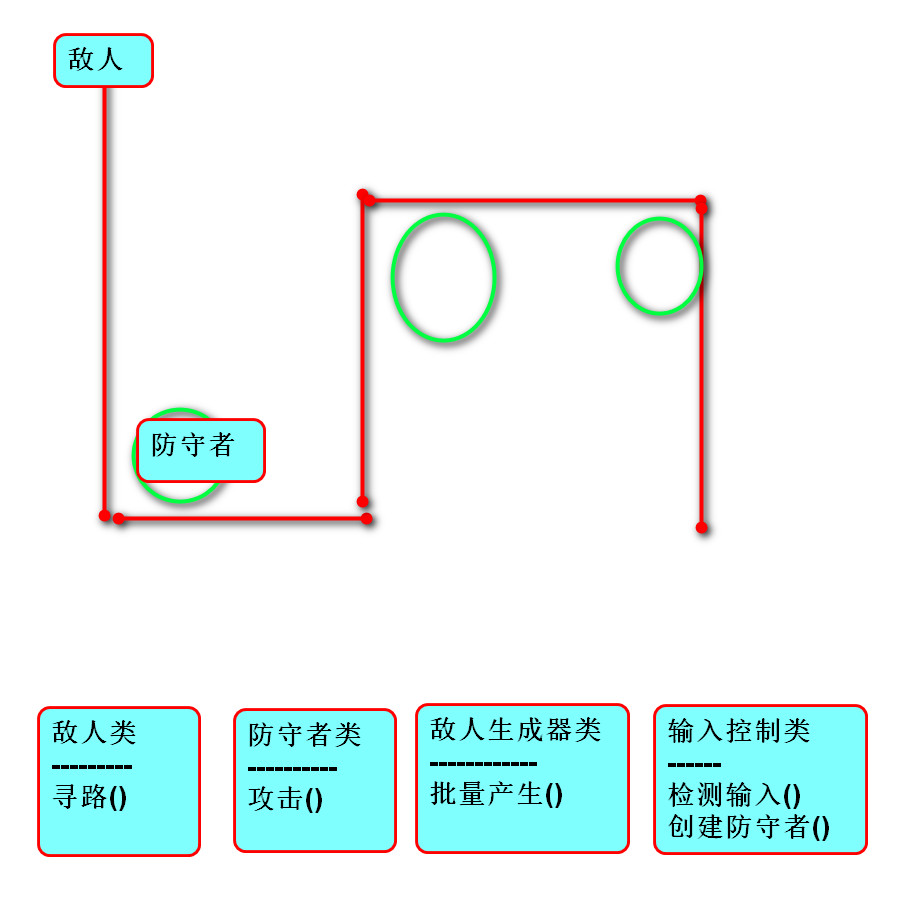
#### 设计角度讲

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

分而治之：划分为多个类

变则疏之：识别变化点（行为）

-------------

高内聚：类有且只要一个变化点(原因)

低耦合：类与类的关系松散

"""

# 类与类行为不同

# 对象与对象数据不同

# 需求：老张开车去东北

# 变化：老张、老李、老王... --> 数据不同

# 划船、骑车、飞机... --> 行为不同

# 西北、南北... --> 数据不同

# 写法1

# 语义：老张去东北创建一辆新车

#　　　老张无论去哪里，都会创建一辆新车

# class Person:

# def \_\_init\_\_(self, name=""):

# self.name = name

#

# def go\_to(self,position):

# print(self.name,"去",position)

# car = Car()

# car.run()

#

# class Car:

# # 实例成员：对象.?

# def run(self):

# print("汽车行驶")

#

# lz = Person("老张")

# lz.go\_to("东北")# 一辆新车

# lz.go\_to("家")# 又一辆新车

# 写法2

# 语义：老张开车自己的车去东北

# class Person:

# def \_\_init\_\_(self, name=""):

# self.name = name

# self.car = Car()

#

# def go\_to(self,position):

# print(self.name,"去",position)

# self.car.run()

#

# class Car:

# def run(self):

# print("汽车行驶")

# lz = Person("老张")

# lz.go\_to("东北")

# lz.go\_to("家")

# 写法3

# 语义：老张通过传递的参数,决定如何去东北.

class Person:

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

self.name = name

def go\_to(self,vehicle,position):

print(self.name,"去",position)

vehicle.run()

class Car:

def run(self):

print("汽车行驶")

lz = Person("老张")

bm = Car()

lz.go\_to(bm,"东北")

#### 案例:信息管理系统

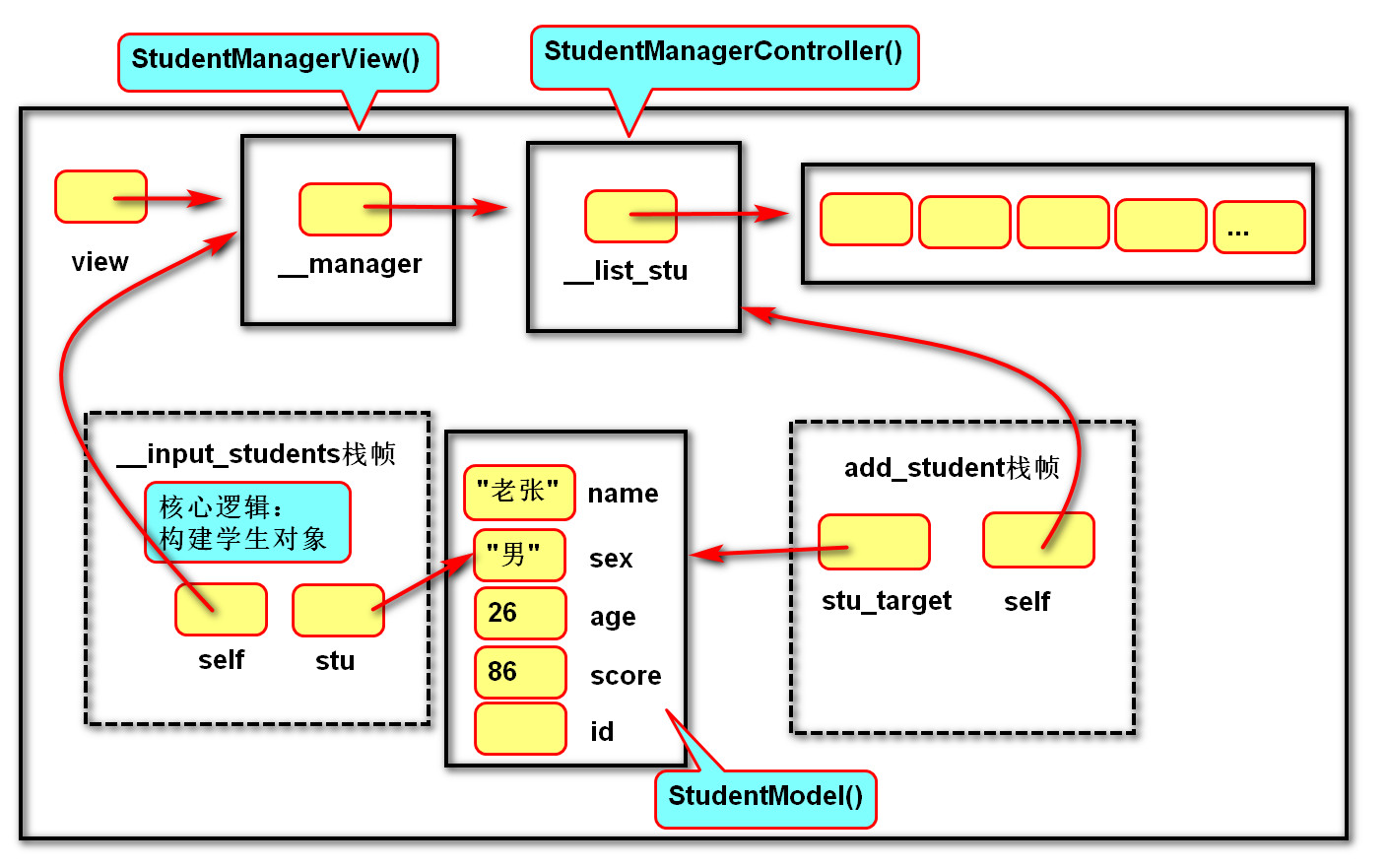


##### 需求

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

"""

封装

目标：保障数据有效性

property 核心逻辑：拦截

1. 创建实例变量

2. 提供读取与写入方法（数据验证）

3. 创建类变量(与实例变量名称相同),存储property对象

"""

class Wife:

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

self.name = name

self.age = age

@property # age = property(读取方法, None)

def age(self):

return self.\_\_age

@age.setter #

def age(self, value):

if 22 <= value <= 32:

self.\_\_age = value

else:

raise Exception("我不要")

w01 = Wife("双儿", 25)

print(w01.name)

# print(w01.get\_age())

print(w01.age)

### 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。

##### 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。

##### 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

"""

多继承

有多种变化需要隔离（不是代码复用）

１６：５０

"""

class A:

def func01(self):

print("A -- func01")

class B(A):

def func01(self):

print("B -- func01")

class C(A):

def func01(self):

print("C -- func01")

class D(C, B):

def func01(self):

print("D -- func01")

# 1. 如何调用某一个同名方法

B.func01(self)

d = D()

d.func01()

# 2. 同名方法解析顺序

print(D.mro())

### 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

##### 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。

"""

内置可重写函数

"""

class Wife:

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

self.name = name

self.face\_score = face\_score

self.money = money

# 重写\_\_str\_\_作用是：定义当前对象转换的字符串格式

# 显示对象时使用（没有格式限制）

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

return "臣妾%s,颜值%d,存款%f." % (self.name, self.face\_score, self.money)

# 重写\_\_repr\_\_作用是：定义当前对象转换的字符串格式

# 克隆当前对象（python语法格式）

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

return 'Wife("%s", %d, %f)' % (self.name, self.face\_score, self.money)

w01 = Wife("双儿", 100, 50000)

message = w01.\_\_str\_\_()

print(message) # <\_\_main\_\_.Wife object at 0x7ff8a3224cf8>

w02 = Wife("阿珂", 150, 20000)

print(w02)

str\_code = w01.\_\_repr\_\_()

# eval:将字符串作为python代码执行

w03 = eval(str\_code) # 克隆老婆对象

w01.name = "双双"

print(w03.name) #

###### 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

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。

"""

运算符重载

"""

class Vector2:

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

self.x = x

self.y = y

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

return "向量的x分量为%d,y分量为%d" % (self.x, self.y)

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

return Vector2(self.x + other.x, self.y + other.y)

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

self.x += other.x

self.y += other.y

return self

# 比较相同

def \_\_eq\_\_(self, other):

return self.x == other.x and self.y == other.y

# 比较大小

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

return self.x < other.x

pos = Vector2(10, 25)

dir = Vector2(1, 0)

print(pos + dir) # pos.\_\_add\_\_(dir)

pos += dir

print(pos)

# list01 = [10]

# list01 += [20]# 在原有对象中添加新元素（在原有可变对象上修改）

# print(list01) # [10, 20]

#

# list02 = [10]

# list02 = list02 + [20] # 创建了新对象

# print(list02) # [10, 20]

v01 = Vector2(100, 200)

v02 = Vector2(100, 200)

print(v01 == v02) #

list\_vectors = [

Vector2(200, 200),

Vector2(400, 400),

Vector2(100, 100),

Vector2(300, 300),

Vector2(500, 500),

]

print(list\_vectors.count(Vector2(100, 100)))

for item in sorted(list\_vectors):

print(item)

算数运算符



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。

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。

### 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。

"""

里氏替换

"""

class Granade:

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

self.atk = atk

def explode(self, target):

print("手雷爆炸啦")

target.damage(self.atk)

class Target:

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

self.hp = hp

def damage(self, value):

self.hp -= value

# ------------------------------------------

class Player(Target):

def damage(self, value):

super().damage(value)

print("玩家受伤,屏幕碎屏")

class Enemy(Target):

def damage(self, value):

super().damage(value)

print("敌人受伤,掉下装备")

g01 = Granade()

g01.explode(Player())

g01.explode(Enemy())

### 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。