# 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。

站在软件制高点上，俯视一切，不谈细节

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：从具体事物中抽离出共性、本质，舍弃个别、非本质过程。

1. 对象：类的具体实例，即归属于某个类别的”个体”。
2. 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)

"""

面向对象的思考步骤：

现实事物 -抽象化-> 类 -具体化-> 对象



现实事物与对象对应，描述现实

# int 类的对象

a = 10

# str 类的对象

b = "悟空"

# list 类的对象

c = [1,2,3]

语法：

class 类名:

def \_\_init\_\_(self, 参数):

self.数据 = 参数

def 方法名称(self,参数):

方法体

变量 = 类名(参数) # 标准：构造函数(参数)

"""

class Wife:

# 数据：名词性的描述

def \_\_init\_\_(self, name, face\_score, money=0.0):

self为自动生成，行业规范，self后面可以5种形参任选

self.name = name 参数name可以替换为其他任意，但一般与属性名字相同

self.money = money

self.face\_score = face\_score

# 行为：动词性的功能

def work(self): def叫方法，面向过程叫函数，面向对象叫方法

print(self.name, "工作")

# 创建对象(自动调用\_\_init\_\_函数)

w01 = Wife("双儿",97,2000) 魔法方法，不写方法名()，写类名（)；

自动调用\_\_init\_\_（self）构造函数，self可不填

# 自动传递对象 work(w01)

地图的截图

描述已自动生成w01.work()

cvvvlo白板上写着字

描述已自动生成

### 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”（赋值） 不建议写法

w01.name = “莉莉”（修改） 不建议写法

1. 通常在构造函数(\_init\_)中创建。

w01 = Wife(“丽丽”,24) 建议写法

print(w01.name)

1. 每个对象存储一份，通过对象地址访问。（**最大价值是每个数据可以不一样**）
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:对象.成员名（实例成员就是讲对象，核心逻辑一定离不开对象）**

**实例变量**

**对象.变量名**

**实例方法 （操作实例变量）**

**对象.方法()**

练习:exercise01

"""

class Wife:

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

# 局部变量：存储在栈帧中

a = 10

# 实例变量：存储在对象中

self.name = name

self.list = [ ] 补充写法1:创建空列表 不写参数

self.b = B() 补充写法2:直接创建对象 也不写参数

def work(self):

# print(a) 不能访问其他方法的局部变量

# 可以访问自身对象的实例变量

print(self.name + "在工作")

w01 = Wife("双儿")

w02 = Wife("建宁")

print(w01.name)

print(w02.name)

# 存储了对象所有的实例变量

print(w01.\_\_dict\_\_) # {'name': '双儿'}

w01.work() # 自动传递对象地址 work(w01) 建议写法

w02.work() # 自动传递对象地址 work(w01)

Wife.work(w01) # 手动传递对象地址 类名.方法名(对象地址) 不建议写法

"""

**不建议写法1：**

class Wife:

pass

w01 = Wife()

w01.name = "建宁" # 创建实例变量

w01.name = "双儿" # 修改实例变量

print(w01.name)

dict01 = {}

dict01["a"] = "A"

dict01["a"] = "B"

print(dict01["a"]) # B

"""

**不建议写法2：**

# class Wife:

# def set\_name(self,name):

# # 建议创建实例变量在\_\_init\_\_中

# self.name = name

#

# w01 = Wife()

# w01.set\_name("双儿")

# print(w01.name) #

### 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 类成员离不开类，围着类转**

**类变量**

**创建：在类中**

**使用：用类名 类名.类变量**

**类方法**

练习:exercise02

"""

class ICBC:

# 类变量：总行的钱

total\_money = 1000000

@classmethod

def print\_total\_money(cls):

# print("总行的钱是", ICBC.total\_money)

print("总行的钱是", cls.total\_money) cls简化类名的写法

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

self.name = name

# 实例变量：支行的钱

self.money = money

# 总行的钱减少

ICBC.total\_money -= money

tt = ICBC("天坛支行", 100000)

trt = ICBC("陶然亭支行", 200000)

# print("总行的钱：", ICBC.total\_money)

ICBC.print\_total\_money()# print\_total\_money(ICBC)

"""

**总结Python所有变量**

"""

# 全局变量:存储文件中

a = 10

def func01():

# 局部变量:存储栈帧中

b = 20

class MyClass:

# 类变量:存储类中 【大家的变量：饮水机】

d = 40

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

# 实例变量:存储对象中

self.c = 30 # 【自己的变量：杯子】

手机屏幕截图

描述已自动生成

### 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。

name face\_score age height weight --> Wife

备注：把类做出来，让一个类型包装多个类型，以前[]取数，以后.取数

1. 优势：

将数据与对数据的操作相关联。

备注：def \_\_init\_\_ 与def work() 数据抽象之后与行为合在一起

代码可读性更高（类是对象的模板）。

class Commodity:

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

self.cid = cid

self.name = name

self.price = price

将数据抽象成一个类，就是数据封装，用实例变量表达信息。

类是具体数据模板

list\_commodity\_infos = [

Commodity(1001, "屠龙刀", 10000),

Commodity(1002, "倚天剑", 10000),

Commodity(1003, "金箍棒", 52100),]

#### 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。 （核心是隐藏）

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。

"""

封装行为

需求：类的定义者保障数据的有效性

年龄： 25 ～ 30

练习：exercise04

"""

class Wife:

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

self.name = name

self.age = age # 2

self.name赋值时自动执行设置函数，无需调用

@property

执行读取函数

def age(self):

return self.\_\_age

@age.setter

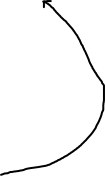


def age(self, value): # 3

if 25 <= value <= 30:

self.\_\_age = value

else:



raise Exception("不行")

w01 = Wife("双儿", 25) # 1

print(w01.name)

print(w01.age) #

"""

私有成员:以双下划线开头的成员

类外无法访问,类内可以访问（不要访问私有成员，尊重创作者）

本质：障眼法

看着是双下划线命名 \_\_data

实际是单下划线类名+双下划线命名 \_MyClass\_\_data

"""

class MyClass:

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

self.\_\_data = data

def \_\_func01(self):

print("func01执行喽")

c01 = MyClass(10)

print(c01.\_\_dict\_\_)

# print(c01. \_\_data)

# c01.\_\_func01()

# 不要试图访问私有成员

# print(c01.\_MyClass\_\_data)

# c01.\_MyClass\_\_func01()

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)

"""

property 属性（没有属性，所有操作都对实例变量，有了属性变为属性）

价值：保护实例变量

1. 属性名与实例变量名称相同（拦截）

2. 属性中操作私有变量（需要被保护）

核心：拦截

**“属性名不保存数据，私有变量存储数据”**

"""

class Wife:

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

self.name = name

# 属性

self.age = age

# age = property(读取函数)

# (1) 创建属性对象property()

# (2) 将下面的函数作为参数property(读取函数)

# (3) 将属性对象交给变量名关联age

@property

def age(self):# 读取函数 属性不保存数据

return self.\_\_age 私有变量存储数据

# age.setter(设置函数)

# (1) 调用属性的setter函数

# (2) 将下面的函数作为参数setter(设置函数)

@age.setter

def age(self, value): # 设置函数

if 25 <= value <= 30:

self.\_\_age = value

else:

raise Exception("不行")

w01 = Wife("双儿", 25) # 1

print(w01.name)

print(w01.age) #

"""

属性各种写法

"""

"""

# 写法1：读写(能够对外提供读取和设置功能)

# 快捷键：props + 回车

class Wife:

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

self.age = age# 执行设置函数

@property

def age(self): # 读取函数

return self.\_\_age

@age.setter

def age(self, value): # 设置函数

self.\_\_age = value

w01 = Wife(25)

print(w01.age)# 执行读取函数

"""

"""

# 写法2：只读(只对外提供读取功能,类外不能修改)

# 快捷键：prop + 回车

class Wife:

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

self.\_\_age = age# 为私有变量赋值

@property

def age(self):

return self.\_\_age

w01 = Wife(25)

print(w01.age)# 执行读取函数

# w01.age = 10

"""

# 写法3：只写(只能够对外提供设置功能)

# 快捷键： 无

class Wife:

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

self.age = age # 执行设置函数

age = property()

@age.setter

def age(self, value): # 设置函数

self.\_\_age = value

w01 = Wife(25)

# print(w01.age) # 执行读取函数

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

#### 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。 （大功能拆小块）

(2) 变则疏之

变化的地方独立封装，避免影响其他类。 （把每个变化点分到一个类，经常变的拆出来）

(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。

大学生对封装认知为：把一些数据包装成一个类型，在一个类中体现数据的核心行为

对外提供必要的功能，隐藏实现的细节

达内培训需掌握封装设计思想：分而治之 变则疏之 高内聚 低耦合

重点1：划分类的依据

重点2：类与类的相互调用语法

"""

封装设计思想（1划分类的依据）

需求：老张开车去东北

类：承担行为 （区分行为不同，分而治之，考虑行为不同）

对象：承担数据 （区分数据不同）

1. 识别对象

老张 车

2. 分配职责

去() 行驶()

3. 建立交互

老张调用车

练习:exercise01

"""

# lz = Person("老张")

# ls = Person("老孙")

"""

# 写法1：直接创建对象

# 语义： 老张去东北用一辆新车

class Person:

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

self.name = name

def go\_to(self,position):

print(self.name,"去",position)

car = Car() 用实例方法，必须创建对象

car.run() 对象.方法名（）

class Car:

# 实例方法

def run(self):

print("汽车在行驶")

lw = Person("老王")

lw.go\_to("东北")

lw.go\_to("西北")

"""

"""

# 写法2：在构造函数中创建对象

# 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)

# 第一次点.: 从栈帧到人对象(self.car = Car())

# 第二次点.: 从人到车对象(self.car.run())

self.car.run()

class Car:

# 实例方法

def run(self):

print("汽车在行驶")

lw = Person("老王")

lw.go\_to("东北")

lw.go\_to("西北")

"""

# 写法3：通过参数传递对象(不创建对象)

# 语义：人通过交通工具(参数传递而来)去东北（参数传递的是一个交通工具对象）

class Person:

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

self.name = name

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

print(self.name,"去",position)

vehicle.run()

class Car:

# 实例方法

def run(self):

print("汽车在行驶")

lw = Person("老王")

c01 = Car()

lw.go\_to("东北",c01)

"""

总结

类与类调用语法

"""

# 函数互相调用：通过函数名直接调用

# def func01():

# func02()

# def func02():

# print("func02")

# func01()

# 类与类调用写法1：直接创建对象调

# class A:

# def func01(self):

# b = B()

# b.func02()

# class B:

# def func02(self):

# print("func02")

# g01 = A()

# g01.func01()

# 类与类调用写法2：构造函数创建对象调

# class A:

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

# self.b = B()

#

# def func01(self):

# self.b.func02()

# class B:

# def func02(self):

# print("func02")

# g01 = A()

# g01.func01()

# 类与类调用写法3：通过参数传递对象（不创建对象）

class A:

def func01(self, c):

c.func02()

class B:

def func02(self):

print("func02")

g01 = A()

g02 = B()

g01.func01(g02)

### 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.

"""

继承

财产：钱不用儿子挣，但是儿子能花

皇位：江山不用太子打，但是太子能登基

编程：代码不用子类写，但是子类能使用

练习：exercise03

"""

# 从思想讲：先有子再有父 父难写：抽象

# 从编码讲：先有父再有子 子容易：具体

# 多个类有代码的共性，且属于一个共同概念。

class Person:

def say(self):

print("说话")

class Student(Person): 类名（父类）

def study(self): 函数名（参数）

self.say() 继承调用比封装调用简单很多，但是要慎重使用

print("学习")

class Teacher(Person):

def teach(self):

print("教学")

# 创建子类对象,可以调用父类方法和子类方法

s01 = Student()

s01.say()

s01.study()

# 创建父类对象,只能调用父类方法

p01 = Person()

p01.say()

##### 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。

适用性：实参与形参判定

# isinstance(对象,类型) 判断关系

# 学生对象 是一种 学生类型

print(isinstance(s01, Student)) # True

# 学生对象 是一种 人类型

print(isinstance(s01, Person)) # True

# 学生对象 是一种 老师类型

print(isinstance(s01, Teacher)) # False

# 人对象 是一种 学生类型

卡通人物

描述已自动生成print(isinstance(p01, Student)) # False

# issubclass(类型,类型) 判断关系

# 学生类型 是一种 学生类型

print(issubclass(Student, Student)) # True

# 学生类型 是一种 人类型

print(issubclass(Student, Person)) # True

# 学生类型 是一种 老师类型

print(issubclass(Student, Teacher)) # False

# 人类型 是一种 学生类型

print(issubclass(Person, Student)) # False

# Type

# type(对象) == 类型 相等/相同/一模一样

# 学生对象的类型 是 学生类型

print(type(s01) == Student) # True

# 学生对象的类型 是 人类型

print(type(s01) == Person) # False

# 学生对象的类型 是 老师类型

print(type(s01) == Teacher) # False

# 人对象的类型 是 学生类型

print(type(p01) == Student) # False

##### 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 儿子(爸爸):

def \_\_init\_\_(self, 爸爸构造函数参数,儿子构造函数参数):

super().\_\_init\_\_(爸爸构造函数参数)

self.数据 = 儿子构造函数参数

练习：exercise04

"""

class Person:

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

self.name = name

self.age = age

class Student(Person):

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

# 通过supe r()调用父类实例成员

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

self.score = score

# 1. 子类没有构造函数,可以直接使用父类的

# s01 = Student()

# 2. 子类有构造函数,会覆盖父类构造函数(好像他不存在)

（指父子构造函数参数重复的情况下会覆盖）

# 所以子类必须通过super()调用父类构造函数

s01 = Student("小明", 24, 100)

print(s01.name)

print(s01.age)

print(s01.score)

地图的截图

描述已自动生成

##### 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**

**继承不是代码的复用方式（继承是隔离变化）**

"""

class A:

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

self.a = a

def func01(self):

print("A -- func01,实例变量:",self.a)

class B:

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

self.b = a

def func02(self):

print("B -- func01实例变量:",self.b)

# C 需要使用AB的函数

class C(A, B):

def func03(self):

super().func01()

super().func02()

print("C -- func01")

# 创建C对象,使用的是A构造函数,没有执行B构造函数.

c = C("a")

c.func03() # 因为B对象没有执行构造函数,所以不能正常工作

地图上有字

描述已自动生成

没有代码复用

"""

**多继承**

**同名方法解析顺序**

**类.mro()**

面试回答：通过类.mro（）方法去判定

"""

手机屏幕的截图

描述已自动生成

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(B, C):

def func01(self):

super().func01() # 继承列表第一个父类

print("D -- func01")

d = D()

d.func01()

# 解析顺序：类.mro() 解析顺序定义：当调用D的方法时，到底去哪里找

# [<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

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。

"""

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。

步骤：

1. 调用父-先用 （编码时调用父，运行时执行子，写1时，3还没有）

2. 子重写-后做 （重写时多态的一个步骤）

3. 创建子

目的：

彰显子类个性(不同/变化/具体)

体现开闭原则(目标)

"""

class A:

def func01(self):

pass

class B(A):

# 2. 子重写

def func01(self):

print("B -- func01")

class C(A):

def func01(self):

print("C -- func01")

def func02(a):

# 1. 调用父

a.func01()

b = B()

c = C()

# 3. 创建子

func02(c)

#### 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。

解释：编码时调用父，执行时运行子

父类体现共性

子类彰显个性，重写（名一样，执行的是自己的）

##### 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

\_\_str\_\_函数：将对象转换为字符串(对人友好的)

适用性：呈现自定义对象时，决定对象展示的风格 return不是print

\_\_repr\_\_函数：将对象转换为字符串(解释器可识别的)

适用性：拷贝自定义对象

"""

内置可重写函数（核心是时机）

"""

# 任何一个类,都直接或间接继承自object类(万类之祖).

# class Dog(object):

class Dog:

# 时机：创建对象时 自动执行

def \_\_init\_\_(self, variety, name, age, weight=0.0):

self.variety = variety

self.name = name

self.age = age

self.weight = weight

# 时机：打印对象时 自动执行子类

def \_\_str\_\_(self):# 没有语法限制，return后自行定义

将对象转换为字符串 - - >对人类友好

return f"我是{self.name},品种{self.variety},今年{self.age}岁了,体重{self.weight}斤"

# 时机：拷贝对象时 （对比切片拷贝容器，repr拷贝对象）

def \_\_repr\_\_(self): # return返回的字符串必须满足Python语法格式要求

将对象转换为字符串 - - >解释器可识别

通过eval方法可以让机器执行该字符串

return f'Dog("{self.variety}", "{self.name}", {self.age}, {self.weight})'

d01 = Dog("拉布拉多", "米咻", 5, 70)

#　我是米咻,品种拉布拉多,今年5岁了,体重70斤

print(d01) 自定义对象打印的结果是类名+地址<main. Dog at 0x7fc6fff..>

# message = d01.\_\_str\_\_() print原理1 <main. Dog at 0x7fc6fff..>

# print(message) print原理2

# 将字符串作为代码执行

# eval(字符串) --> eval(input()) 将"无所不能"

d02 = eval(d01.\_\_repr\_\_()) 将下方的字符串作为代码执行，拷贝对象

f'Dog("{self.variety}", "{self.name}", {self.age}, {self.weight})'

应用场景：代码或数据库需要定期备份，防误删，数据丢失

d01.name = "咻咻" 因为拷贝生成两个对象，d01改变不影响d02的结果

print(d02) "米咻"

###### 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



"""

运算符重载（Python无重载只有重写） 开发时不用,但可以了解本质。

算数运算符

任何语言本质无加减，只有基于面向对象思想设计的对象和方法

加法的本质是调用\_\_add\_\_(self, rhs)，时机：两个对象相加时自动调用

各种类型如int str list 均内置重写了add，相加时调用add

"""

class Vector2:

"""

二维向量

"""

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

self.x = x

self.y = y

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

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

pos01 = Vector2(1, 2)

pos02 = Vector2(3, 4)

pos03 = pos01 + pos02 # pos01.\_\_add\_\_(pos02) 计数机的本质

print(pos03.\_\_dict\_\_) 或者用str重写打印结果

复合运算符重载



"""

运算符重载（比较+=与+的区别） 开发时不用，重点调试返回新/原有

增强运算符 += -= \*= /= ...

add + 无论可变还是不可变对象,都创建新对象

iadd += 对于可变对象,在原有基础上进行修改 (return self)

对于不可变对象,创建新对象

练习:exercise02

"""

class Vector2:

"""

二维向量

"""

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

self.x = x

self.y = y

# += 累加

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

# 因为自定义类是可变对象

# 所以返回原有对象self,不是创建新对象

self.x += other.x

self.y += other.y

return self 与add区别的关键是self，返回自身

pos01 = Vector2(1, 2)

print(id(pos01)) ID1

pos01 += Vector2(3, 4)

print(id(pos01)) ID2

在有add的情况下，ID1与ID2不同，在有iadd的情况下，ID1与ID2相同

"""可变对象与不可变对象累加的区别

# += 对于可变对象,在原有基础上进行修改

list01 = [1]

print(id(list01))# 139887136870152

list01 += [2]

print(id(list01))# 139887136870152

# += 对于不可变对象,创建新对象

tuple01 = (1,)

print(id(tuple01))# 139887167310872

tuple01 += (2,)

print(id(tuple01))# 139887136858376

比较运算重载

"""

核心：自定义对象的列表,如果需要使用内置函数就需要重写比较运算符。

重写比较运算符



\_\_eq\_\_ 定义相同依据 内置函数index count in

\_\_lt\_\_ 定义大小依据 内置函数 sort max

"""

class Employee:

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

self.eid = eid

self.did = did

self.name = name

self.money = money

# 员工列表 list\_employees = [

Employee(1001, 9002, "师父", 60000),

Employee(1001, 9002, "师父", 60000),

Employee(1001, 9002, "师父", 60000),

Employee(1002, 9001, "孙悟空", 50000),

Employee(1005, 9001, "小白龙", 15000),]

# 定义员工对象的相同依据 重写eq（根据eid判断）

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

return self.eid == other.eid

# 员工对象的大小依据 重写lt（根据money判断）

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

return self.money < other.money

e01 = Employee(1001, 9002, "师父", 60000)

e02 = Employee(1001, 9002, "师父", 60000)

# 内部调用\_\_eq\_\_

# ==比较内容是否相同

print(e01 == e02) # true 编号相同(未重写eq时，默认比较地址is，结果为false)

# is比较地址是否相同

print(e01 is e02) # false 两个员工对象地址不同

print(list\_employees.count(Employee(1001)))

#未重写eq时，结果为0，自定义类count失效；重写eq后，结果为3

print(list\_employees.index(Employee(1005)))

#未重写eq时，报错，自定义类index失效；重写eq后，结果为4

# 内部在循环调用员工的\_\_lt\_\_方法

list\_employees.sort() #Python自带排序，开发时可以直接使用，但要懂排序算法

print(list\_employees) # 打印时，重写\_\_repr\_\_

"""

面向对象设计思想 （作用：设计软件结构，不是做具体功能）

需求：老张开车去东北

变化：增加飞机、自行车.... 现实中变化很多

封装：划分类 人类 车类 分

继承：隔离 人类 具体交通工具 隔

抽象：父类是如何来的，根据变化点抽象出相关类的共性

统一：父类要干什么，要统一（约束）子类行为（约束函数名、参数、返回值， 只有函数体不约束）

隔离：父类约束的价值/目的是隔离变化 （隔离用与做）

语法上代码复用的缺点：紧耦合，失去灵活性，违背开闭原则，所以千万不 要认为继承是代码的复用方式，而是解耦、隔离

多态（重写）：彰显子类个性 具体交通工具重写交通工具的方法

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。

步骤：

1. 调用父-先用 （编码时调用父，运行时执行子，写1时，3还没有）

2. 子重写-后做 （重写时多态的一个步骤）

3. 创建子

目的：

彰显子类个性(不同/变化/具体)

体现开闭原则(目标)

情景：手雷爆炸，可能伤害敌人或者玩家的生命。

变化：还可能伤害房子、树、鸭子....

要求：增加新事物，不影响手雷.

体会：开闭原则

画出架构设计图

"""

--------------架构师------------

class Person: #1

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

self.name = name

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

print(self.name, "去", position)

vehicle.transport()

class Vehicle: #2 增加继承父类

"""

交通工具虽然没有具体功能代码

但是在隔离人与具体交通工具

"""

def transport(self):

pass

--------------程序员------------

class Car(Vehicle): #3 重写 CRTL + O

def transport(self):

print("汽车在行驶")

class Airplane(Vehicle): #4 重写 CRTL + O

# ctrl+o

def transport(self):

print("飞机在飞行")

--------------测试------------ 不在面向对象设计范围之内

lw = Person("老王")

c01 = Car()

a01 = Airplane()

lw.go\_to("东北", c01)

手机屏幕截图

描述已自动生成

面向对象设计思想是拥抱变化、应对变化

当下关注变化、违背面向对象设计思想

代码缺点：**违反开闭原则**

增加飞机，还要修改人的代码

引出继承



第一阶段三板斧之三 软件架构设计图 开发顺序1->2->3->4

封装：人类、 车类

继承：抽象 统一 隔离

抽象：父类是如何来的，根据变化点抽象来的

统一：父类要干什么，统一（约束）子类行为 （约束函数名、参数、返回值， 只有函数体不约束）

隔离：父类约束的价值/目的是隔离变化 （隔离用与做）

手机屏幕的截图

描述已自动生成 重写：彰显子类个性 具体交通工具重写交通工具的方法

print与object是一起写的

隔离30年

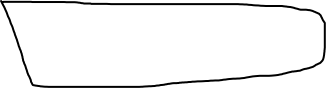
1991年与2020年

父类

继承

调用object

day13 作业1



重写

## 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。 （与函数小而精思想一致，对封装补充）

### 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。 （对继承的补充，引出继承）

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。 （组合：我与你们，小明与通讯工具，手机，座机等）

手机屏幕截图

描述已自动生成

桥

抽象

右侧变化：增加新通话方式

左侧小明要跟随变化

用变量去调用：

小明有一个手机

语义不通：

小明是一种手机

### 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。（父只提供客户端代码必要功能）

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。

具体做法：A要调用B的功能2，为了减少传参量，用C隔离，C只放功能2，C是B的父类

C 功能2



A

B 功能1、功能2、功能3、功能4

"""

**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.

三大特征

封装：创建GraphicManager、Circle、Rectanle

继承：创建Graphic图形(抽象/统一/隔离 具体图形)

多态:GraphicManager调用Graphic

Circle、Rectanle重写Graphic

向GraphicManager添加Circle、Rectanle对象

六大原则：

开闭：增加新图形,图形管理器不变.

单一职责：

GraphicManager管理所有图形

Circle 计算圆形面积

Rectanle 计算矩形面积

依赖倒置：GraphicManager使用父Graphic

组合复用：GraphicManager与图形组合（父与子都包括在内）

手机屏幕的截图

描述已自动生成"""

class GraphicManager:

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

self.all\_graphic = []

#里氏替换1： graphic 类型是父类图形，不是圆形、矩形，但是传递的是圆形、矩形（看见父，传递子）

def add\_graphic(self,graphic):

self.all\_graphic.append(graphic)

def calculate\_total\_area(self):

total\_area = 0

for graphic in self.all\_graphic:

# 使用所有图形的统一行为

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

return total\_area

class Graphic:

def get\_area(self):

"""

计算该图形面积

:return: 数值类型,图形的面积

"""

pass

# 父类在约束所有所有子类在某一行为上达到统一

class Circle(Graphic):

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

self.r = r 听爸爸的话，儿子自己解决参数问题

def get\_area(self): （self,r）r父类不能给你参数，否则就面向变化了

# 里氏替换2：扩展重写，建议先调用super().get\_area()，再加点新功能

# super().get\_area()

return 3.14 \* self.r \*\* 2

class Rectanle(Graphic):

def \_\_init\_\_(self, l,w):

self.l = l

self.w = w

def get\_area(self):

return self.l \* self.w

manager = GraphicManager()

manager.add\_graphic(Circle(5))

manager.add\_graphic(Rectanle(5, 6))

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

"""

创建员工管理器

1. 记录多个员工（程序员、测试员....）

2. 提供计算总薪资的方法.

程序员：底薪 + 项目分红

测试员: 底薪 + Bug数 × 5

满足：

开闭原则

测试：

创建员工管理器，存储多个员工对象。

通过员工管理器，调用计算总薪资方法.

三大特征

封装：创建EmployeeManager、Programmer、Tester（分）

继承：创建Employee（隔）

多态:EmployeeManager调用Employee （做）

Programmer、Tester重写Employee

向EmployeeManager添加的是Programmer、Tester对象

六大原则：

开闭：增加新岗位的员工，EmployeeManager不改变

单一职责：

EmployeeManager操作所有员工

Programmer负责实现程序员的薪资算法

Tester负责实现测试员的薪资算法

依赖倒置：

EmployeeManager使用Employee

不使用Programmer、Tester

组合复用：

EmployeeManager和员工薪资算法

里氏替换：

Programmer、Tester重写时先调用父类方法

迪米特法则：

Employee隔离EmployeeManager与Programmer、Tester的变化

手机屏幕的截图

描述已自动生成"""

class EmployeeManager:

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

self.all\_employee = []

def add\_employee(self, emp):

self.all\_employee.append(emp)

def calculate\_total\_money(self):

total\_money = 0

for emp in self.all\_employee:

total\_money += emp.get\_money()

return total\_money

class Employee:

def get\_money(self):

pass

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

class Programmer(Employee):

def \_\_init\_\_(self, base\_salary, bonus):

self.base\_salary = base\_salary

self.bonus = bonus

def get\_money(self):

super().get\_money()

return self.base\_salary + self.bonus

class Tester(Employee):

def \_\_init\_\_(self, base\_salary, bug\_count):

self.base\_salary = base\_salary

self.bug\_count = bug\_count

def get\_money(self):

super().get\_money()

return self.base\_salary + self.bug\_count \* 5

manager = EmployeeManager()

manager.add\_employee(Programmer(8000, 100000))

manager.add\_employee(Tester(5000, 500))

print(manager.calculate\_total\_money())

**代码进一步优化**

"""

创建员工管理器

1. 记录多个员工（程序员、测试员....）

2. 提供计算总薪资的方法.

程序员：底薪 + 项目分红

测试员: 底薪 + Bug数 × 5

满足：

开闭原则

测试：

创建员工管理器，存储多个员工对象。

通过员工管理器，调用计算总薪资方法.

"""

class EmployeeManager:

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

# 建议将使用的数据私有化

self.\_\_all\_employee = []

def add\_employee(self, emp):

# 如果 emp 是一种 员工类型

if isinstance(emp, Employee):

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

def calculate\_total\_money(self):

total\_money = 0

for emp in self.\_\_all\_employee:

total\_money += emp.get\_money()

return total\_money

class Employee:

def \_\_init\_\_(self, base\_salary):

self.base\_salary = base\_salary

def get\_money(self):

return self.base\_salary

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

class Programmer(Employee):

def \_\_init\_\_(self, base\_salary, bonus):

super().\_\_init\_\_(base\_salary)

self.bonus = bonus

def get\_money(self):

# 先通过爸爸的方法获取底薪

base\_salary = super().get\_money()

return base\_salary + self.bonus

class Tester(Employee):

def \_\_init\_\_(self, base\_salary, bug\_count):

super().\_\_init\_\_(base\_salary)

self.bug\_count = bug\_count

def get\_money(self):

return super().get\_money() + self.bug\_count \* 5

manager = EmployeeManager()

manager.add\_employee(Programmer(8000, 100000))

manager.add\_employee(Tester(5000, 500))

manager.add\_employee("二大爷")

print(manager.calculate\_total\_money())

## 面向对象总结

**复习面向对象**

**面向对象：考虑问题从对象的角度出发**

**识别对象 分配职责 建立交互**

**三个特征：**

**封装：分而治之,变则疏之 [分]**

**继承：抽象、统一、隔离变化 [隔]**

**多态：体现子类个性(变化) [做]**

**六个原则：**

**开闭原则：能够增加新功能,不修改客户端代码.**

**单一职责：小而精,有且只有一个改变的原因**

**依赖倒置：使用抽象(爸爸),不适用具体(儿子)**

**组合复用：优先使用组合关系,不是继承关系.**

**继承：统一变化(交通工具约束火车汽车在运输的行为上是一致的)**

**组合：连接变化(人通过变量调用交通工具)**

**里氏替换：形参是父,实参可以是各种子类。**

**建议扩展重写**

**迪米特：通过抽象隔离调用(低耦合)**

## MVC

手机屏幕截图

描述已自动生成

"""

基本的操作：增删改查

分配职责：

界面视图类View：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Controller：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Model：定义需要处理的数据类型。比如学生信息。

"""

list01 = []

# 增加

list01.append(10)

list01.append(20)

# 修改

list01[0] = 100

# 查询

print(list01[0])

# 删除

list01.remove(10)

del list01[0]

"""

基本调用

"""

class XXView:

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

self.controller = XXController()

def func01(self):

self.controller.func02()

class XXController:

def func02(self):

print("func02执行喽")

view = XXView()# 内部创建Controller

view.func01() # 内部调用func02

"""

学生信息管理系统MVC

"""

# 2.数据模型

class StudentModel:

"""

学生数据模型

对具体学生信息进行抽象

"""

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

self.name = name

self.age = age

self.score = score

# 学生编号：对数据进行唯一标识(全球唯一标识符)

self.sid = sid # 自增长1001 1002 1003

# 对某个数据进行有效性验证

@property

def score(self):

return self.\_\_score

@score.setter

def score(self, value):

if value < 0:

value = 0

elif value > 100:

value = 100

self.\_\_score = value

# 3.界面逻辑

class StudentView:

"""

学生视图：负责处理界面逻辑

"""

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

self.\_\_controller = StudentController()

def \_\_display\_menu(self):

print("1) 添加学生信息")

print("2) 显示学生信息")

print("3) 删除学生信息")

# ...

def \_\_select\_menu(self):

item = input("请输入选项：")

if item == "1":

# 先写调用,再快捷键生成定义函数代码

# atl + 回车

self.\_\_input\_student()

elif item == "2":

self.\_\_show\_students()

elif item == "3":

self.\_\_delete\_student()

def main(self):

"""

入口函数

"""

while True:

self.\_\_display\_menu()

self.\_\_select\_menu()

def \_\_input\_student(self):

stu = StudentModel()

stu.name = input("请输入学生姓名：")

stu.age = int(input("请输入学生年龄："))

stu.score = int(input("请输入学生成绩："))

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

def \_\_show\_students(self):

for stu in self.\_\_controller.list\_students:

print(f"{stu.name}的编号是{stu.sid}年龄是{stu.age}成绩是{stu.score}")

def \_\_delete\_student(self):

sid = int(input("请输入需要删除的学生编号："))

if self.\_\_controller.remove\_student(sid):

print("删除成功")

else:

print("删除失败")

# 4.业务逻辑

class StudentController:

"""

学生控制器

负责处理业务逻辑

"""

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

self.\_\_list\_students = []

self.\_\_start\_sid = 1001

# 只读属性

@property

def list\_students(self):

return self.\_\_list\_students

def add\_student(self, stu):

"""

添加学生

:param stu: 需要添加的学生对象

"""

stu.sid = self.\_\_start\_sid

self.\_\_start\_sid += 1

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

def remove\_student(self, sid):

"""

删除学生

:param sid: int类型的学生编号

:return: bool类型,是否删除成功

"""

for student in self.\_\_list\_students:

if student.sid == sid:

self.\_\_list\_students.remove(student)

return True # 删除成功

return False # 删除失败

# 1.入口

view = StudentView()

view.main()

"""

Python语言的设计理念：

代码简洁自由,开发效率高.

其他语言：

int number = 100;

number = "007"; x 错误

python:

number = 100

number = "007"

类型标注：

number:int = 100

缺点：

不利于开发大型项目。

制作

def 函数名(参数名称):

...

使用

函数名(?)

Typing 类型标注：

python3.5 新功能,可以为变量增加类型标注(标记注释).

作用：

易于理解

类型检查

方便开发

语法：

变量名:类型

-> 返回值类型

# type:类型

typing：

List[类型] 标注列表元素的类型

Union[类型1,类型2] 标注可以选择的多种类型

Optional[类型1,类型2] 相当于Union[类型1,类型2,None]

"""

# 自定义类型

**from typing import List, Union, Optional**

class EpidemicInformationModel: # 套餐

"""

疫情信息模型

"""

def \_\_init\_\_(self, region, confirmed, cure, dead=0):

"""

创建疫情信息对象

:param region:地区

:param confirmed:确诊人数

:param cure:治愈人数

:param dead:死亡人数

"""

self.region = region # type:str

self.confirmed = confirmed # 确诊人数 小吃

self.cure = cure # 治愈人数 冷饮

self.dead = dead # 死亡人数 冰激凌

class EpidemicInformationManager: # 麦当劳

"""

疫情信息管理器

"""

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

# 疫情列表

self.\_\_list\_epidemics = [] # type:List[EpidemicInformationModel]

def add\_epidemic\_info(self, info: EpidemicInformationModel):

self.\_\_list\_epidemics.append(info)

# def get\_epidemic\_by\_region(self, region) -> Union[EpidemicInformationModel,None]:

def get\_epidemic\_by\_region(self, region) -> Optional[EpidemicInformationModel]:

"""

根据地区获取疫情信息

:param region:地区

:return:疫情信息对象

"""

for item in self.\_\_list\_epidemics:

if item.region == region:

return item

if \_\_name\_\_ == '\_\_main\_\_':

manager = EpidemicInformationManager()

# manager.add\_epidemic\_info(EpidemicInformationModel("湖北", 66337, 28993))

manager.add\_epidemic\_info(EpidemicInformationModel("四川", 538, 351, 3))

# manager.add\_epidemic\_info("四川")

epidemic = manager.get\_epidemic\_by\_region("四川")

print(epidemic.region, epidemic.confirmed, epidemic.cure, epidemic.dead)