Day07 课堂笔记

课程之前

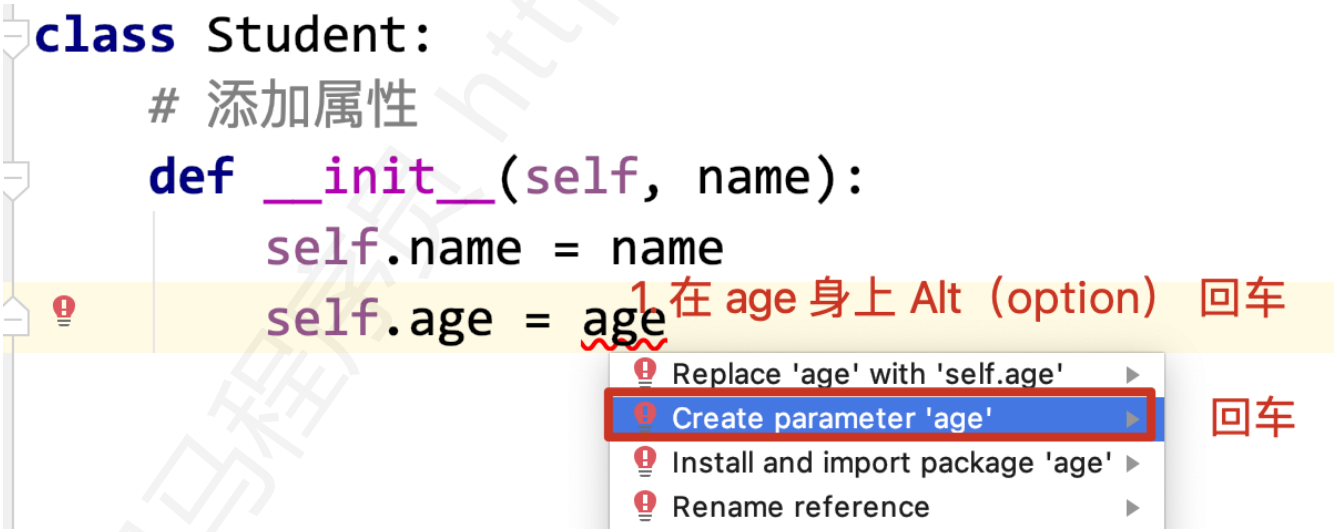
复习和反馈

```
魔法方法:
1. 方法什么时候自动调用，调用时机
2. 方法做什么用，用在哪
3. 注意事项
------
__init__ 初始化方法，
1. 创建对象后
2. 给对象添加属性的
3. 不要写错，如果存在 self 之外的形参,创建对象时,需要传递实参

__str__ 方法
1. print(对象)  会自动调用
2. 想要打印输出对象的时候，输出对象的属性信息
3. 必须返回一个字符串

dir(对象)  获取对象所有的方法
```

作业



- 作业三

```
类名: 学生类 Student
属性: 姓名 name, 年龄 age
方法: 吃饭 eat  睡觉 sleep  过年year , __str__ , __init__
```

```python
class Student:
    # 添加属性
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"姓名: {self.name}, 年龄: {self.age} 岁"

    def eat(self):
        print(f'{self.name} 要吃饭')

    def sleep(self):
        print(f'{self.name} 要睡觉')

    def year(self):
        self.age += 1


# 创建对象
xm = Student('小明', 18)
xh = Student('小红', 17)
print(xm)
print(xh)
xm.eat()
xm.sleep()
xm.year()
print(xm)
```

- 作业 4

```
类名: 电脑类 Computer
属性: 品牌 brand , 价格 price  [movie 电影的名字]
方法: play_movie
```

```python
class Computer:
    def __init__(self, brand, price):
        self.brand = brand
        self.price = price

    def play_movie(self, movie):
        print(f'{self.brand} 播放电影 {movie}')


# 创建对象
mi = Computer('小米', 4999)
mac = Computer('Mac', 16999)
mi.play_movie('葫芦娃')
mac.play_movie('变形金刚')
mi.play_movie('西游记')

```

今日内容

- 封装(定义类的过程)
  - 案例(存放家具)
- 继承
- 多态
- 封装的补充
  - 私有和公有权限
  - 属性的分类(实例属性, 类属性)
  - 方法的分类(实例方法, 类方法, 静态方法)

封装案例

```
类名: 房子类 House
属性: 户型 name, 总面积 total_area, 剩余面积 free_area = total_area
    家具名称列表 item_list = []
方法: __init__, __str__
添加家具方法
def add_item(self, item):  #item 家具对象
    先判断房子的剩余面积和总面积的关系
        修改房子的剩余面积
        修改房子的家具名称列表
```

```
类名: 家具类 HouseItem
属性: 名字 name, 占地面积 area
方法: __init__, __str__
```

```python
# 定义家具类
class HouseItem:
    """家具类"""
    def __init__(self, name, area):
        """添加属性的方法"""
        self.name = name
        self.area = area

    def __str__(self):
```

```
        return f'家具名字{self.name}, 占地面积 {self.area} 平米'


class House:
    """房子类"""
    def __init__(self, name, area):
        self.name = name  # 户型
        self.total_area = area  # 总面积
        self.free_area = area   # 剩余面积
        self.item_list = []     # 家具名称列表

    def __str__(self):
        return f"户型: {self.name}, 总面积:{self.total_area}平米, 剩余面积:
{self.free_area} 平米, " \
               f"家具名称列表: {self.item_list}"

    def add_item(self, item):  # item  表示的家具的对象
        # 判断房子的剩余面积(self.free_area)和家具的占地面积(item.area)之间的关系
        # self 表示的 房子对象, 缺少一个家具对象使用传参解决
        if self.free_area > item.area:
            # 添加家具, ---> 向列表中添加数据
            self.item_list.append(item.name)
            # 修改剩余面积
            self.free_area -= item.area
            print(f'{item.name} 添加成功')
        else:
            print('剩余面积不足, 换个大房子吧')


# 创建家具对象
bed = HouseItem('席梦思', 4)
chest = HouseItem('衣柜', 2)
table = HouseItem('餐桌', 1.5)
print(bed)
print(chest)
print(table)

# 创建房子对象
house = House('三室一厅', 150)
print(house)
# 添加 床
house.add_item(bed)
print(house)

```

案例 2

```
需求：某 web 项目登录页面包含：用户名，密码，验证码，登录按钮 和登录的方法
书写代码实现以上功能，登录方法中使用 print 输出即可

类名：LoginPage
属性：用户名(username)，密码(password)，验证码(code)，登录按钮(button)
方法：登录(login)   __init__
```

```python
class LoginPage:
    def __init__(self, username, password, code):
        self.username = username
        self.password = password
        self.code = code
        self.btn = '登录'

    def login(self):
        print(f'1. 输入用户名 {self.username}')
        print(f'2. 输入密码 {self.password}')
        print(f'3. 输入验证码 {self.code}')
        print(f"4. 点击按钮 {self.btn}")


login = LoginPage('admin', '123456', '8888')
login.login()

```

私有和公有

```
1. 在 Python 中定义的方法和属性，可以添加访问控制权限(即在什么地方可以使用这个属性和方法)
2. 访问控制权限分为两种，公有权限，私有权限
3. 公有权限
    > 直接书写的方法和属性，都是公有的
    > 公有的方法和属性，可以在任意地方访问和使用
4. 私有权限
    > 在类内部，属性名或者方法名 前边加上两个 下划线 ，这个属性或者方法 就变为 私有的
    > 私有的方法和属性，只能在当前类的内部使用

5. 什么时候定义私有
    > 1. 某个属性或者方法,不想在类外部被访问和使用，就将其定义为私有即可
    > 2. 测试中,一般不怎么使用，直接公有即可
    > 3. 开发中,会根据需求文档，确定什么作为私有

6. 如果想要在类外部操作私有属性，方法是，在类内部定义公有的方法，我们通过这个公有方法去操作


# 补充:
# 对象.__dict__   魔法属性，可以将对象具有的属性组成字典返回
```

- 案例

```
定义一个 Person 类，属性 name，age(私有)
```

- 代码

```python
class Person:
    def __init__(self, name, age):
        self.name = name   # 姓名
        # 私有的本质，是 Python 解释器执行代码,发现属性名或者方法名前有两个_，会将这个名字重命名
        # 会在这个名字的前边加上 _类名前缀,即 self.__age ===> self._Person__age
        self.__age = age  # 年龄，将其定义为私有属性，属性名前加上两个 _

    def __str__(self):  # 在类内部可以访问私有属性的
        return f'名字: {self.name}, 年龄: {self.__age}'


xm = Person('小明', 18)
print(xm)
# 在类外部直接访问 age 属性
# print(xm.__age)  # 会报错，在类外部不能直接使用私有属性
# 直接修改 age 属性
xm.__age = 20  # 这个不是修改私有属性，是添加了一个公有的属性 __age
print(xm)  # 名字: 小明, 年龄: 18
print(xm._Person__age)  # 能用但是不要用  18
xm._Person__age = 19
print(xm)  # 名字: 小明, 年龄: 19

```

继承

```
1. 继承描述的是类与类之间的关系
2. 继承的好处：减少代码的冗余(相同的代码不需要多次重复书写)，可以直接使用
```

语法

```python
# class A(object):
class A: # 没有写父类,但也有父类，object，object 类是 Python 中最顶级(原始)的类
    pass

class  B(A):  # 类 B，继承类 A
    pass
```

```
术语：
1. A 类，称为是 父类(基类)
2. B 类，称为是 子类(派生类)

单继承：一个类只继承一个父类,称为单继承
继承之后的特点：
    > 子类(B)继承父类(A)之后，子类的对象可以直接使用父类中定义的公有属性和方法


```

- 案例

```
1. 定义一个 动物类，吃
2. 定义一个 狗类，继承动物类，吃，叫
3. 定义一个 哮天犬类，继承 狗类
```

- 代码

```python
# 1. 定义一个 动物类，吃
class Animal:
    def eat(self):
        print('要吃东西')


# 2. 定义一个 狗类，继承动物类，吃，叫
class Dog(Animal):
    def bark(self):
        print('汪汪汪叫....')


# 3. 定义一个 哮天犬类，继承 狗类
class XTQ(Dog):
    pass


# 创建 动物类的对象
# ani = Animal()
# ani.eat()
# 创建狗类对象
# dog = Dog()
# dog.eat()  # 调用父类中的方法
# dog.bark()   # 调用自己类中方法
# 创建哮天犬类对象
xtq = XTQ()
xtq.bark()  # 调用 父类 Dog 类的方法
xtq.eat()   # 可以调用 父类的父类中的方法


```

- 结论

```
python 中 对象.方法() 调用方法
1. 现在自己的类中的去找有没有这个方法 如果有,直接调用
2. 如果没有去父类中 查找， 如果有,直接调用
3. 如果没有， 去父类的父类中查找， 如果有直接调用
4 ...
5. 如果 object 类中有,直接调用， 如果没有,代码报错
```

重写

```
重写: 在子类中定义了和父类中名字相同的方法， 就是重写
重写的原因: 父类中的方法,不能满足子类对象的需求,所以重写
重写之后的特点: 调用子类字节的方法， 不再调用父类中的方法
重写的方式:
    >1. 覆盖(父类中功能完全抛弃,不要,重写书写)
    >2. 扩展(父类中功能还调用,只是添加一些新的功能) (使用较多)
```

覆盖

```
1. 直接在子类中 定义和父类中名字相同的方法
2. 直接在方法中书写新的代码
```

```python
class Dog:
    def bark(self):
        print('汪汪汪叫.....')


class XTQ(Dog):
    # XTQ 类bark 方法不再是汪汪汪叫， 改为 嗷嗷嗷叫
    def bark(self):
        print('嗷嗷嗷叫...')


xtq = XTQ()
xtq.bark()
```

扩展父类中的功能

```
1. 直接在子类中 定义和父类中名字相同的方法
2. 在合适的地方调用 父类中方法  super().方法()
3. 书写添加的新功能
```

```python
class Dog:
    def bark(self):
        print('汪汪汪叫.....')
        print('汪汪汪叫.....')


```

```python
class XTQ(Dog):
    # XTQ 类bark 方法不再是汪汪汪叫，改为
    # 1. 先 嗷嗷嗷叫(新功能) 2，汪汪汪叫(父类中功能)  3. 嗷嗷嗷叫 (新功能)
    def bark(self):
        print('嗷嗷嗷叫...')
        # 调用父类中的代码
        super().bark()  # print() 如果父类中代码有多行呢?
        print('嗷嗷嗷叫...')


xtq = XTQ()
xtq.bark()
```

多态[了解]

```
1. 是一种写代码,调用的一种技巧
2. 同一个方法，传入不同的对象，执行得到不同的结果，这种现象称为是多态
3. 多态 可以 增加代码的灵活度

--------
哪个对象调用方法，就去自己的类中去查找这个方法，找不到去父类中找
```

属性和方法

```
Python 中一切皆对象.
即 使用 class 定义的类 也是一个对象
```

对象的划分

实例对象(实例)

```
1. 通过 类名() 创建的对象，我们称为实例对象,简称实例
2. 创建对象的过程称为是类的实例化
3. 我们平时所说的对象就是指 实例对象(实例)
4. 每个实例对象，都有自己的内存空间，在自己的内存空间中保存自己的属性(实例属性)
```

类对象(类)

```
1. 类对象 就是 类，或者可以认为是 类名
2. 类对象是 Python 解释器在执行代码的过程中 创建的
3. 类对象的作用: ① 使用类对象创建实例 类名()， ② 类对象 也有自己的内存空间，可以保存一些属性值信息(类属性)
4. 在一个代码中，一个类 只有一份内存空间
```

属性的划分

实例属性

- 概念: 是实例对象 具有的属性
- 定义和使用

```
在 init 方法中, 使用 self.属性名 = 属性值 定义
在方法中是 使用 self.属性名 来获取(调用)
```

- 内存

```
实例属性,在每个实例中 都存在一份
```

- 使用时机

```
1. 基本上 99% 都是实例属性,即通过 self 去定义
2. 找多个对象,来判断这个值是不是都是一样的, 如果都是一样的, 同时变化,则一般定义为 类属性, 否则定义为 实例属性
```

类属性

- 概念: 是 类对象 具有的属性
- 定义和使用

```
在类内部,方法外部,直接定义的变量 ,就是类属性

使用:  类对象.属性名 = 属性值  or  类名.属性名 = 属性值
类对象.属性名  or  类名.属性名
```

- 内存

```
只有 类对象 中存在一份
```

方法的划分

```
方法, 使用 def 关键字定义在类中的函数就是方法
```

实例方法(最常用)

- 定义

```python
# 在类中直接定义的方法 就是 实例方法
class Demo:
    def func(self):   # 参数一般写作 self,表示的是实例对象
        pass
```

- 定义时机(什么时候用)

```
如果在方法中需要使用实例属性(即需要使用 self), 则这个方法必须定义为 实例方法
```

- 调用

```
对象.方法名()   # 不需要给 self 传参
```

类方法(会用)

- 定义

```python
# 在方法名字的上方书写 @classmethod 装饰器(使用 @classmethod 装饰的方法)
class Demo:
    @classmethod
    def func(cls):  # 参数一般写作 cls, 表示的是类对象(即类名) class
        pass
```

- 定义时机(什么时候用)

```
1. 前提, 方法中不需要使用 实例属性(即 self)
2. 用到了类属性, 可以将这个方法定义为类方法,(也可以定义为实例方法)
```

- 调用

```python
# 1. 通过类对象调用
类名.方法名()  # 也不需要给 cls 传参, python 解释器自动传递
# 2. 通过实例对象调用
实例.方法名()  # 也不需要给 cls 传参, python 解释器自动传递
```

静态方法(基本不用)

- 定义

```python
# 在方法名字的上方书写 @staticmethod 装饰器(使用 @staticmethod 装饰的方法)
class Demo:
    @staticmethod
    def func():   # 一般没有参数
        pass
```

- 定义时机(什么时候用)

```
1. 前提，方法中不需要使用 实例属性(即 self)
2. 也不使用 类属性，可以将这个方法定义为 静态方法
```

```
class Company:
    def show_work(self, worker):  # worker 参数需要传入一个对象,只要这
        """wo                                  对象"""
        worke
        在方法中没有使用 self
```

Method 'show_work' may be 'static'
方法 可能是静态的
Make static More actions...

- 调用

```
# 1. 通过类对象调用
类名.方法名()
# 2. 通过实例对象调用
实例.方法名()
```

练习

练习 1

```
定义一个 Dog 类，定义一个类属性 count,用来记录创建该类对象的个数. (即每创建一个对象,count 的值就要加1)实例属性 name
```

```
class Dog:
    # 定义类属性
    count = 0

    # 定义实例属性，init 方法中
    def __init__(self, name):
        self.name = name   # 实例属性
        # 因为每创建一个对象,就会调用 init 方法，就将个数加 1 的操作,写在 init 方法中
        Dog.count += 1


# 在类外部
# 打印输出目前创建几个对象
print(Dog.count)  # 0
# 创建一个对象
dog1 = Dog('小花')
# 打印输出目前创建几个对象
print(Dog.count)  # 1
dog2 = Dog   # 不是创建对象，个数不变的
dog3 = dog1  # 不是创建对象，个数不变的
print(Dog.count)  # 1

dog4 = Dog('大黄')  # 创建一个对象 ，个数 + 1
print(Dog.count)  # 2
dog5 = Dog('小白')
print(Dog.count)  # 3

```

```python
# 补充，可以使用 实例对象.类属性名 来获取类属性的值 （原因，实例对象属性的查找顺序，先在实例属性中找，找到直接使用
# 没有找到会去类属性中 找，找到了可以使用，没有找到 报错）
print(dog1.count)  # 3
print(dog4.count)  # 3
print(dog5.count)  # 3

```

题目 2

定义一个游戏类 `Game`，包含实例属性 玩家名字(`name`)

1. 要求记录游戏的最高分(`top_score` 类属性),
2. 定义方法: `show_help` 显示游戏的帮助信息 输出`这是游戏的帮助信息`
3. 定义方法: `show_top_score`, 打印输出游戏的最高分
4. 定义方法: `start_game`, 开始游戏, 规则如下
    1. 使用随机数获取本次游戏得分 范围 `(10 - 100 )`之间
    2. 判断本次得分和最高分之间的关系
        - 如果本次得分比最高分高,
            - 修改最高分
        - 如果分数小于等于最高分,则不操作
    3. 输出本次游戏得分
5. 主程序步骤

```python
# 1）创建一个 Game 对象  小王
# 2）小王玩一次游戏，
# 3）查看历史最高分
# 4）小王再玩一次游戏
# 5）查看历史最高分
# 6）查看游戏的帮助信息
```

- 基本版本

```python
import random


class Game:
    # 类属性，游戏的最高分
    top_score = 0

    def __init__(self, name):
        # 定义实例属性 name
        self.name = name

```

```
    def show_help(self):
        print('这是游戏的帮助信息')

    def show_top_score(self):
        print(f'游戏的最高分为 {Game.top_score}')

    def start_game(self):
        print(f'{self.name} 开始一局游戏，游戏中 ...,', end='')
        score = random.randint(10, 100)  # 本次游戏的得分
        print(f'本次游戏得分为 {score}')
        if score > Game.top_score:
            # 修改最高分
            Game.top_score = score


xw = Game('小王')
xw.start_game()
xw.show_top_score()
xw.start_game()
xw.show_top_score()
xw.show_help()

```

- 优化（使用 类方法和静态方法）

```
    @staticmethod
    def show_help():
        print('这是游戏的帮助信息')

    @classmethod
    def show_top_score(cls):
        print(f'游戏的最高分为 {cls.top_score}')
```

补充

```
哈希（hash）：是一个算法， 可以对数据产生一个唯一的值（指纹）

is 可以用来判断两个对象是不是同一个对象，即 两个对象的引用是否相同
a is b   ===> id(a) == id(b)

面试中可能会问： is 和 == 的区别？
==  只判断数据值是否相同， is 判断引用是否相同
```