课堂笔记

课程之前

复习和反馈

```
print: 只是打印输出,将内容显示到控制台, 代码中大多数的 print 都是为了验证结果
return: 可以将函数中的数据返回到函数外部,可以进行其他的操作(print, ....)
将来的代码 如果存在数据,大多数都是 return

类()  --> 参数 __init__ 方法
函数()  ---> 参数, 定义

对象 = 类名()
对象.方法名()
```

作业

今日内容

```
面向对象
-   继承(重点)
-   多态(了解)
-   权限划分(公有 和 私有)
-   对象(实例对象 和 类对象)
-   属性(实例属性 和 类属性)
-   方法(实例方法 类方法 静态方法)
-   案例

文件操作(使用代码操作文件,对文件进行读写, 重点是读)
```

继承

```
1, 继承描述的是类与类之间的关系  is ... a
2, 继承的好处: 减少代码冗余,重复代码不需要多次书写, 提高编程效率
```

语法

```python
# class 类A(object):
# class 类A():
class 类A:  # 默认继承 object 类，object 类 Python 中最原始的类
    pass

class 类B(类A):  # 就是继承，类 B 继承 类 A
    pass


# 类 A：父类  或 基类
# 类 B：子类  或 派生类
```

子类继承父类之后，子类对象可以直接使用父类中的属性和方法

```python
# 1. 定义动物类，动物有姓名和年龄属性，具有吃和睡的行为
class Animal:
    """动物类"""
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def eat(self):
        """吃"""
        print(f'{self.name} 吃东西')

    def sleep(self):
        """睡"""
        print(f"{self.name} 睡觉")


# 2. 定义猫类，猫类具有动物类的所有属性和方法，并且具有抓老鼠的特殊行为
class Cat(Animal):
    """猫类"""
    def catch(self):
        print(f"{self.name} 会抓老鼠...")


# 3. 定义狗类，狗类具有动物类的所有属性和方法，并且具有看门的特殊行为
class Dog(Animal):
    """狗类"""
    def look_the_door(self):
        """看门"""
        print(f"{self.name} 正在看家....")


# 4. 定义哮天犬类，哮天犬类具有狗类的所有属性和方法，并且具有飞的特殊行为
class XTQ(Dog):
    """哮天犬类"""
    def fly(self):
        """飞"""
        print(f"{self.name} 在天上飞...")
```

```
if __name__ == '__main__':
    ani = Animal('佩奇', 2)
    ani.eat()
    ani.sleep()
    cat = Cat('黑猫警长', 10)
    cat.eat()  # 调用 父类 animal 中的方法
    cat.sleep()  # 调用 父类 animal 中的方法
    cat.catch()  # 调用自己类的方法
    dog = Dog('旺财', 3)
    dog.eat()
    dog.sleep()
    dog.look_the_door()
    xtq = XTQ('哮天犬', 100)
    xtq.eat()
    xtq.sleep()
    xtq.look_the_door()
    xtq.fly()
```

```
继承具有传递性: C 继承 B, B 继承 A,  C 可以使用 A 类中的属性和方法

对象调用方法的顺序:  对象.方法名()
1, 会现在自己的类中查找, 找到直接使用
2, 没有找到 去父类中查找, 找到直接使用
3, 没有找到, 在父类的父类中查找, 找到直接使用
4, 没有找到, ...
5, 直到 object 类, 找到直接使用, 没有找到,报错
```

重写(override)

```
1, 什么是重写?
    重写是在子类中定义了和父类中名字一样的方法.
2, 重写的原因? 为什么重写?
    父类中的代码不能满足子类对象的需要
3, 重写的方式
3.1  覆盖式重写
3.2  扩展式重写
```

覆盖式重写

```
父类中的功能全部不要.
直接在子类中定义和父类中方法名字一样的方法接口, 直接书写新的代码.
```

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

```
class XTQ(Dog):
    # 需要哮天犬 嗷嗷嗷叫, 父类中的 bark 方法,不能满足子类对象的需要, 覆盖式重写
    def bark(self):
        print('嗷嗷嗷叫.....')
    pass


if __name__ == '__main__':
    xtq = XTQ()
    xtq.bark()
```

扩展式重写

父类中的功能还需要,只是添加了新的功能
方法:
1. 先在子类中定义和父类中名字相同的方法
2. 在子类的代码中 使用 super().方法名() 调用父类中的功能
3. 书写新的功能

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


class XTQ(Dog):
    # 需要哮天犬 嗷嗷嗷叫, 父类中的 bark 方法,不能满足子类对象的需要, 覆盖式重写
    def bark(self):
        # 调用父类中的功能
        super().bark()
        print('嗷嗷嗷叫.....')
        print('嗷嗷嗷叫.....')


if __name__ == '__main__':
    xtq = XTQ()
    xtq.bark()
```

多态

多态: 调用代码的技巧

多态:不同的子类对象调用相同的方法, 产生不同的执行结果

```
class Dog:
```

```
    def game(self):
        print('普通狗简单的玩耍...')


class XTQ(Dog):
    def game(self):
        print('哮天犬在天上玩耍...')


class Person:
    def play_with_dog(self, dog):
        """dog 是狗类或者其子类的对象"""
        print('人和狗在玩耍...', end='')
        dog.game()


if __name__ == '__main__':
    dog1 = Dog()
    xtq = XTQ()
    xw = Person()
    xw.play_with_dog(dog1)
    xw.play_with_dog(xtq)
```

私有和公有

在Python 中,定义类的时候，可以给 属性和方法设置 访问权限，即规定在什么地方可以使用.
权限一般分为两种：公有权限 私有权限

公有权限

- 定义

  直接定义的属性和方法就是公有的

- 特点

  可以在任何地方访问和使用，只要有对象就可以访问和使用

私有权限

- 定义

  1，只能在类内部定义(class 关键字的缩进中)
  2，只需要在 属性名 或者方法名 前边 加上两个 下划线，这个方法或者属性就变为私有的

- 特点

> 私有 只能在当前类的内部使用. 不能在类外部和子类直接使用

- 应用场景

> 一般来说,定义的属性和方法 都为公有的.
> 某个属性 不想在外部直接使用, 定义为私有
> 某个方法,是内部的方法(不想在外部使用), 定义为私有

```
"""定义人类, name 属性 age 属性(私有)"""


class Person:
    def __init__(self, name, age):
        self.name = name  # 公有
        self.__age = age  # 公有-->私有, 在属性名前加上两个下划线

    def __str__(self):  # 公有方法
        return f"{self.name}, {self.__age}"

    def set_age(self, age):  # 定义公有方法,修改私有属性
        if age < 0 or age > 120:
            print('提供的年龄信息不对')
            return
        self.__age = age


if __name__ == '__main__':
    xw = Person('小王', 18)
    print(xw)
    xw.__age = 10000  # 添加一个公有属性 __age
    print(xw)
    xw.set_age(10000)
    print(xw)
```

对象 属性 方法

对象分类

> python 中一切皆对象.

类对象

```
类对象 就是 类，就是使用 class 定义的类
在代码执行的时候，解释期会自动的创建.
作用:
1，使用类对象创建 实例对象
2，存储一些类的特征值(类属性)
```

实例对象

```
1，创建对象也称为实例化，所以 由类对象(类) 创建的对象 称为是 实例对象，简称实例
2，一般来说,没有特殊强调,我们所说的对象 都是指  实例对象(实例)
3，实例对象 可以保存实例的特征值 (实例属性)
4，就是使用 类名()  创建的对象
```

属性的划分

```
使用 实例对象.属性  访问 属性的时候，会先在 实例属性中查找,如果找不到,去类属性中查找，找到就使用，找不到,就报错
即:  每个实例对象 都有可能访问类属性值(前提,实例属性和类属性不重名)
```

实例属性

- 概念

  ```
  是每个实例对象 具有的特征(属性)，每个实例对象的特征
  ```

- 定义

  ```
  一般都是在 init 方法中,使用 self.属性名 = 属性值  来定义
  ```

- 特征(内存)

  ```
  每个实例对象 都会保存自己的 实例属性，即内存中存在多份
  ```

- 访问和修改

  ```
  # 可以认为是通过 self
  实例对象.属性 = 属性值  # 修改
  实例对象.属性  # 访问
  ```

类属性

- 概念

是类对象具有的 特征，是整个类的特征

- 定义

一般 在类的内部(class 缩进中)，方法的外部(def 的缩进外部) 定义的变量

- 特征(内存)

只有类对象保存一份，即在内存中只有一个

- 访问和修改

```
# 即通过类名
类对象.属性 = 属性值
类对象.属性
```

> 什么时候 定义类属性?
>
> 代码中 使用的属性 基本上 都是 实例属性,即都通过 self 定义.
>
> 当 某个属性值描述的信息是整个类的特征(这个值变动,所有的这个类的对象这个特征都会发生变化)

案例

```
1. 定义一个 工具类
2. 每件工具都有自己的 name
3. 需求 — 知道使用这个类，创建了多少个工具对象?
```

```
类名: Tool
属性: name(实例属性)  count(类属性)
方法: init 方法
```

```python
class Tool:
    # 定义类属性 count,记录创建对象的个数
    count = 0

    def __init__(self, name):
        self.name = name  # 实例属性，工具的名字
        # 修改类属性的值
        Tool.count += 1


if __name__ == '__main__':
    # 查看 创建对象的个数

    print(Tool.count)   # 查看类属性
```

```
tool1 = Tool('锤子')
print(Tool.count)
tool2 = Tool('扳手')
print(Tool.count)
print(tool2.count)  # 先找实例属性 count，找不到，找类属性 count，找到,使用
```

方法的划分

实例方法

- 定义时机

  > 如果方法中 需要使用 实例属性，则这个方法 **必须** 定义为实例方法

- 定义

```
# 直接定义的方法就是实例方法
class 类名:
    def 方法名(self):
        pass
```

- 参数

  > 参数一般写作 self，表示的是 实例对象

- 调用

  > 实例对象.方法名()

类方法

- 定义时机

  > 如果 方法中 不需要使用 实例属性，但需要使用 类属性，则这个方法 **可以** 定义为 类方法(建议)

- 定义

```
# 定义类方法,需要在方法名上方 书写 @classmethod  ，即使用 @classmethod 装饰器装饰
class 类名:
    @classmethod
    def 方法名(cls):
        pass
```

- 参数

> 参数 一般写作 cls，表示类对象，即 类名，同样不需要手动传递,Python 解释器会自动传递

- 调用

```
# 方法一
类名.方法名()

# 方法二
实例对象.方法名()
```

静态方法(了解)

- 定义时机

> 方法中即不需要使用 实例属性，也不需要使用 类属性，**可以** 将这个方法定义为 静态方法

- 定义

```
# 定义静态方法，需要使用 装饰器 @staticmethod  装饰方法

class 类名:
    @staticmethod
    def 方法名():
        pass
```

- 参数

> 静态方法,对参数没有要求，一般没有

- 调用

```
# 方法一
类名.方法名()

# 方法二
实例对象.方法名()
```

```
class Tool:
    # 定义类属性 count,记录创建对象的个数
    count = 0

    def __init__(self, name):
        self.name = name  # 实例属性，工具的名字
        # 修改类属性的值
        Tool.count += 1
```

```
    @classmethod
    def show_tool_count(cls):  # cls 就是类对象，类名
        return cls.count


if __name__ == '__main__':
    # 查看 创建对象的个数
    print(Tool.show_tool_count())
    tool1 = Tool('锤子')
    print(Tool.show_tool_count())
    tool2 = Tool('扳手')
    print(tool2.show_tool_count())
```

案例

需求:
1. 设计一个 Game 类
2. 属性:
• 定义一个 top_score 类属性 -> 记录游戏的历史最高分
• 定义一个 player_name 实例属性 -> 记录当前游戏的玩家姓名
3. 方法:
• 静态方法 show_help() -> 显示游戏帮助信息
• 类方法 show_top_score() -> 显示历史最高分
• 实例方法 start_game() -> 开始当前玩家的游戏
- ① 使用随机数 生成 10-100 之间数字 作为本次游戏的得分
- ② 打印本次游戏等分 : 玩家 xxx 本次游戏得分 ooo
- ② 和历史最高分进行比较，如果比历史最高分高，修改历史最高分
4. 主程序步骤: main
    1 查看帮助信息
    2 查看历史最高分
    3 创建游戏对象，开始游戏
    4 再一次游戏

类名: Game
属性:
    top_score = 0 类属性
    player_name 实例属性 init

```
import random


class Game:
    # 定义类属性，保存历史最高分
    top_score = 0

    def __init__(self, name):

        self.play_name = name  # 实例属性
```

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

    # 类方法
    @classmethod
    def show_top_score(cls):
        print(f'历史最高分为: {cls.top_score}')

    def start_game(self):
        print(f'玩家 {self.play_name} 开始游戏.....')
        score = random.randint(10, 100)  # 本次游戏得分
        print(f" 玩家 {self.play_name} 本次游戏得分为 {score}")
        if score > Game.top_score:
            Game.top_score = score


if __name__ == '__main__':
    Game.show_help()
    Game.show_top_score()
    player = Game('小王')
    player.start_game()
    Game.show_top_score()
    player.start_game()
    Game.show_top_score()
```

文件

文件操作，使用代码 来读写文件
1，Game 案例,最高分不能保存的，可以将最高分保存到文件中，
2，自动化，测试数据在文件中保存的，从文件中读取测试数据,进行自动化代码的执行

文件的介绍

文件：可以存储在长期存储设备(硬盘，U盘)上的一段数据即为文件

1，计算机只认识 二进制数据(0 和 1)
2，所有的文件在计算机中存储的形式都是 二进制即 0 和 1 ,打开文件看到的是文字不是 0 和 1 ,原因是打开文件的软件会自动的将二进制转换为 文字

文件的分类(根据能否使用文本软件(记事本)打开文件):
- 文本文件
- 二进制文件

- 文本文件

    - 可以使用记事本软件打开
    - `txt, py, md, json`
- 二进制文件
    - 不能使用 记事本软件打开
    - 音频文件 `mp3`
    - 视频文件 `mp4 ....`
    - 图片 `png, jpg, gif, exe`

文件操作的步骤

```
1, 打开文件
2, 读写文件
3, 关闭文件(保存)
```

打开文件 open()

```
open(file, mode='r', encoding=None)   # 将硬盘中的文件 加载到内存中
- file: 表示要操作的文件的名字,可以使用相对路径 和绝对路径
    - 绝对路径, 从根目录开始书写的路径 C:/  D:/
    - 相对路径, 从当前目录开始书写的路径  ./   ../
- mode: 打开文件的方式
    - r , 只读打开  read, 如果文件不存在,会报错
    - w , 只写打开, write, 如果文件存在, 会覆盖原文件
    - a , 追加打开, append, 在文件的末尾写入新的内容
- encoding: 编码格式, 指 二进制数据 和 汉字 转换的规则的
    - utf-8(常用) : 将一个汉字转换为 3 个字节的二进制
    - gbk: 将一个汉字转换为 2 个字节的二进制
返回值: 文件对象, 后续对文件的操作, 都需要这个文件对象
```

关闭文件 close()

```
文件对象.close()  # 关闭文件, 如果是 写文件, 会自动保存, 即将内存中的数据同步到硬盘中
```

读文件 read()

```
变量 = 文件对象.read()
返回值: 返回读取到文件内容, 类型是字符串
```

```
# with open('a.txt', encoding='utf-8') as f:
#     buf = f.read()
#     print(buf)

f = open('a.txt', encoding='utf-8')
data = f.read()
print(data)

f.close()
```

写文件 write()

```
文件对象.write()
# 参数：写入文件的内容，类型 字符串
# 返回值：写入文件中的字符数，字符串的长度，一般不关注
```

```
# 1，打开文件
f = open('a.txt', 'w', encoding='utf-8')
# 2，写文件
# f.write('hello python!')
f.write('好好学习\n天天向上')
# 3，关闭文件
f.close()
```

文件打开的另一种写法(推荐)

```
with open(file, mode, encoding) as 变量:  # 变量 就是文件对象
    pass

# 使用这种写法打开文件，会自动进行关闭,不用手动书写关闭的代码
# 出了 with 的缩进之后，文件就会自动关闭
```

```
with open('a.txt', 'a', encoding='utf-8') as f:
    f.write('good good study\n')
```