笔记

> 版权：智泊AI
>
> 作者：Jeff

一、封装

1.概念

> 广义的封装：函数和类的定义本身，就是封装的体现
>
> 狭义的封装：一个类的某些属性，在使用的过程 中，不希望被外界直接访问，而是把这个属性给作为私有的【只有当前类持有】，然后暴露给外界一个访问的方法即可【间接访问属性】
>
> 封装的本质：就是属性私有化的过程
>
> 封装的好处：提高了数据的安全性，提高了数据的复用性

2.属性私有化和方法私有化

> 如果想让成员变量不被外界直接访问，则可以在属性名称的前面添加两个下划线__,成员变量则被称为私有成员变量
>
> 私有属性的特点：只能在类的内部直接被访问，在外界不能直接访问
>
> 代码演示：

```
class Girl():
  def __init__(self,name,sex,height):
      self.name = name
      self.sex = sex
      self.height = height
      # 比如女孩的年龄是秘密,在外面不能轻易的访问,需要把年龄设置为私有属性
      self.__age = 18

   def say(self):
      print("帅哥,帮个忙呗!")


    # 私有方法
    # 接吻
    def __kiss(self):
        print("一吻定终身!")

    # 类中可以访问私有方法
    def love(self, relationship):
        if relationship == "情侣关系":
            self.__kiss()
        else:
            print("不能随便kiss,小心中毒!")
```

```
xiaohong = Girl("小红","美女",168)
print(xiaohong.name)
print(xiaohong.sex)
print(xiaohong.height)
# print(xiaohong.age)   # 将age设置为私有属性后,外部不能直接访问
xiaohong.say()
xiaohong.love("情侣关系")


'''
私有属性:
1.写法:在属性的前面加两个下划线    __age
2.用法:只能在类的内部访问,不能在类的外部访问   可以在类的内部设置一个外部访问的接口(这个接口一般会做各种条件判断,满足后才能访问),让外部获取私有属性的值

私有方法:
1.写法:在方法的前面加两个下划线    __kiss()
2.用法:只能在类的内部访问,不能在类的外部访问.   私有方法一般是用来在类的内部实现某些功能的,对于外部来说没有实质的意义.这种方法一般定义为私有方法.
'''
```

3.@property装饰器

装饰器的作用：可以给函数动态添加功能，对于类的成员方法，装饰器一样起作用

Python内置的@property装饰器的作用:将一个函数变成属性使用

代码演示：

```
# 第一种访问和设置 私有属性的方式
class Girl():
  def __init__(self,name,age):
      self.name = name
    self.__age = age

  # 通过装饰器@property 获取私有属性age
  @property
  def age(self):
      return self.__age

  # 通过装饰器设置私有属性    @ + 私有属性名 + setter
  @age.setter
  def age(self,age):
      self.__age = age

lan = Girl("小兰",21)
print(lan.name)
print(lan.age)  # 通过装饰器修访问私有属性,访问格式: 对象名.私有属性名
```

```
lan.age = 19   # 通过装饰器设置私有属性,格式: 对象名.私有属性名 = 值
print(lan.age)
```

二、类方法和静态方法

类方法：使用@classmethod装饰器修饰的方法，被称为类方法，可以通过类名调用，也可以通过对象调用，但是一般情况下使用类名调用

静态方法：使用@staticmethod装饰器修饰的方法，被称为静态方法，可以通过类名调用，也可以通过对象调用，但是一般情况下使用类名调用

代码演示：

```
class Animal():
  # 类属性
  name = "牧羊犬"

  # 对象属性
  def __init__(self,name,sex):
      self.name = name
      self.sex = sex

  '''
     类方法:
       1.通过@classmethod装饰器修饰的方法就是类方法
       2.类方法可以使用类名或者对象调用. 但是一般情况下使用类名调用类方法(节省内存)
       3.没有self,在类方法中不可以使用其他对象的属性和方法(包括私有属性和私有方法)
       4.可以调用类属性和其他的类方法,  通过cls来调用
       5.形参的名字cls是class的简写,可以更换,只不过是约定俗成的写法而已
       6.cls表示的是当前类
  '''
  @classmethod
  def run(cls):
      print("我是类方法")
      print(cls.name)
      print(cls == Animal) # cls表示的是当前类

  '''
    静态方法:
      1.通过@staticmethod装饰器修饰的方法就是静态方法
      2.通过类名或者对象名都可以调用静态方法  (推荐使用类名调用)
      3.静态方法形式参数中没有cls, 在静态方法中不建议调用(类属性\类方法\静态方法)
      4.静态方法一般是一个单独的方法,只是写在类中

  '''
  # 静态方法
  @staticmethod
  def eat():
```

```
        print("我是静态方法")


Animal.run()  # 类名调用类方法
Animal.eat()  # 类调用静态方法
# 创建对象
dog = Animal('中华土狗','公')
# dog.run()  # 对象调用类方法
```

三、类中的常用属性（了解）

```
__name__
   通过类名访问，获取类名字符串
   不能通过对象访问，否则报错

__dict__
   通过类名访问，获取指定类的信息【类方法，静态方法，成员方法】，返回的是一个字典
   通过对象访问，获取的该对象的信息【所有的属性和值】，，返回的是一个字典
```

代码演示：

```
class Animal(object):
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex

    def eat(self):
        print("吃")

animal = Animal("二哈","公狗")
# __name__ 通过类名访问获取当前类的类名,不能通过对象访问
print(Animal.__name__)   # Animal
# __dict__以字典的形式返回类的属性和方法 以及 对象的属性
print(Animal.__dict__)  # 以字典的形式显示类的属性和方法
print(animal.__dict__)  # 以字典的形式显示对象的属性


# 魔术方法: __str__() 和 __repr__()
class Person(object):
    def __init__(self,name,age):
        self.name = name
        self.age = age

    def swim(self):
```

```
        print("游泳的方法")

    # __str__() 触发时机：当打印对象的时候,自动触发.
    #       一般用它来以字符串的形式返回对象的相关信息,必须使用return返回数据
    '''
    def __str__(self):
        return f"姓名是:{self.name},年龄是:{self.age}"
        # print("姓名是:{self.name},年龄是:{self.age}")
    '''
    # __repr__()作用和 __str__()类似,若两者都存在,执行 __str__()
    def __repr__(self):
        return f"姓名是:{self.name},年龄是:{self.age}"
        # print("姓名是:{self.name},年龄是:{self.age}")

xiaohong = Person("小红",18)
print(xiaohong)
```

四、继承【extends】

1.概念

> 如果两个或者两个以上的类具有相同的属性或者成员方法，我们可以抽取一个类出来，在抽取的类中声明公共的部分
>
> 被抽取出来的类：父类，基类，超类，根类
>
> 两个或者两个以上的类：子类，派生类
>
> 他们之间的关系：子类 继承自 父类
>
> 父类的属性和方法子类可以直接使用。
>
> 注意：
>
> a. object是所有类的父类，如果一个类没有显式指明它的父类，则默认为object
>
> b.简化代码，提高代码的复用性

2.单继承

2.1使用

> 简单来说，一个子类只能有一个父类，被称为单继承
>
> 语法：
>
> 父类：
>
> class 父类类名(object):
>
> 类体【所有子类公共的部分】
>
> 子类：
>
> class 子类类名（父类类名）：

类体【子类特有的属性和成员方法】

说明：一般情况下，如果一个类没有显式的指明父类，则统统书写为object

代码演示：

最简单的继承

```
# 父类
class Person(object):
  def say(self):
      print("说话的方法")

# 子类
class Boy(Person): # 定义一个子类    将父类的类名传进去   子类就继承了父类
  def eat(self):
      print("子类自己的吃饭的方法")

boy = Boy()
boy.eat()   # 子类调用自己的方法
boy.say()   # 子类调用父类的方法


# 有构造函数的单继承
# 父类
class Animal(object):
  def __init__(self,name,sex):
      self.name = name
      self.sex = sex

  def eat(self):
      print("所有的动物都有捕食的技能")

# 子类
class Cat(Animal):
  def __init__(self,name,sex,tail): # 先继承父类的属性,再重构
      # 1.经典的写法
      # Animal.__init__(self,name,sex)  # 继承父类的构造方法

      # 2.隐式的继承父类的构造函数
      super(Cat,self).__init__(name,sex)
      self.tail = tail # 定义子类自己的属性

  def catchMouse(self):
      print("猫抓老鼠")

cat = Cat("波斯猫","母","揪尾巴")
print(cat.name)
print(cat.sex)
print(cat.tail)
cat.eat()
```

```
cat.catchMouse()
```

总结：

继承的特点：

a.子类对象可以直接访问父类中非私有化的属性

b.子类对象可以调用父类中非私有化的成员方法

c.父类对象不能访问或者调用子类 中任意的内容

继承的优缺点：

优点：

a.简化代码，减少代码的冗余

b.提高代码的复用性

c.提高了代码的可维护性

d.继承是多态的前提

缺点：

通常使用耦合性来描述类与类之间的关系，耦合性越低，则说明代码的质量越高

但是，在继承关系中，耦合性相对较高【如果修改父类，则子类也会随着发生改变】

3.多继承

一个子类可以有多个父类

语法：

class 子类类名(父类1，父类2，父类3.。。。)：

类体

代码演示：

```
# 父亲类
class Father(object):
  def __init__(self,surname):
      self.surname = surname

  def make_money(self):
      print("钱难挣, 屎难吃!")

# 母亲类
class Mother(object):
  def __init__(self,height):
      self.height = height
```

```
  def eat(self):
      print("一言不合, 就干饭!")

# 子类
class Son(Father,Mother): # 子类继承多个父类时,在括号内写多个父类名称即可
  def __init__(self,surname,height,weight):
      # 继承父类的构造函数
      Father.__init__(self,surname)
      Mother.__init__(self,height)
      self.weight = weight

  def play(self):
      print("就这这么玩游戏倍爽!")


son = Son("张", 178, 160)
print(son.surname)
print(son.height)
print(son.weight)
son.make_money()
son.eat()
son.play()
```