python基础8_面向对象2

一、面向对象：继承、属性、方法

1. 【记忆】 私有权限

- 公有权限和私有权限的区别：
    - 2个下划线__开头的属性和方法，为私有属性和方法，否则，则为公有的属性和方法
    - 私有权限：只能本类的内部直接访问，不能在类外面直接访问
    - 公有权限：类内部和外部都能直接访问
- 私有属性:

```
"""
```

```
私有属性:
    1. __(2个下划线)开头的属性，就是私有属性
    2. 只能在本类的内部访问，在类的外面无法直接访问
"""


class Dog(object):
    def __init__(self):
        # 私有属性  __(2个下划线)开头的属性，就是私有属性
        self.__baby_count = 0
        # 公有属性
        self.age = 1

    def print_info(self):
        # 私有属性 只能在本类的内部访问，在类的外面无法直接访问
        print(self.__baby_count)


dog1 = Dog()

# 私有属性 只能在本类的内部访问，在类的外面无法直接访问
```

```
# print(dog1.__baby_count)    #
AttributeError: 'Dog' object has no
attribute '__baby_count'
# 公有属性 在类的内外都能访问
print(dog1.age)

dog1.print_info()   # print_info(dog1)
```

- 私有方法

```
"""
私有方法:
    1. __(2个下划线)开头的方法，就是私有方法
    2. 只能在本类的内部访问，在类的外面无法直
接访问
    3. 在类的内部调用实例方法的语法格式:
self.方法名()
"""


class Dog(object):
    def __init__(self):
        self.__baby_count = 0 # 私有属
性，以__(2个下划线)开头的属性
        self.age = 1
```

```
    def print_info(self):
        print(self.__baby_count)
        # 在类的内部调用私有方法的语法格式: self.方法名()
        # 私有方法 只能在本类的内部访问，在类的外面无法直接访问
        self.__leave()

    # 定义一个私有方法：__(2个下划线)开头的方法，就是私有方法
    def __leave(self):
        print('休产假了')


dog1 = Dog()
dog1.print_info()
# AttributeError: 'Dog' object has no attribute '__leave'
# dog1.__leave() # err, 外部不能访问私有方法
```

2. 继承

2.1【记忆】继承介绍

1. 继承的作用：解决**代码重用**问题，提高开发效率
2. 继承的语法格式：

```
class 子类名(父类名):
    pass
```

```
"""
继承：复用代码，继承过来的东西可以复用
格式：
class 子类名(父类名):
    pass

# 父类，也叫基类
# 子类，也叫派生类
"""


# 定义一个父类
class Father(object):
    def __init__(self):
        self.money = 9999999

    def print_info(self):
        print(self.money)
```

```
# 定义一个子类，继承与Father
class Son(Father):
    pass


# 子类创建对象
s = Son()
print(s.money)  # 子类可以使用父类的属性
s.print_info()  # 子类可以使用父类的方法
```

2.2【重点】单继承和多层继承

- 单继承：子类只继承一个父类
- 多层继承：继承关系为多层传递，如生活中的`爷爷、父亲、儿子`

```
"""
单继承：只有一个父类
"""


# 定义一个父类，  Animal
class Animal(object):
```

```
    def eat(self):
        print("吃东西")


# 定义一个子类，只有一个父类 （单继承）
class Dog(Animal):
    pass


# 创建一个子类对象
dog1 = Dog()
dog1.eat()
```

```
"""
生活中的多层继承：几代人关系，爷爷、父亲、儿子
"""


# 定义一个爷爷类， Animal
class Animal(object):
    def eat(self):
        print("吃东西")


# 定义一个父亲类
```

```
class Dog(Animal):
    def drink(self):
        print("喝东西")


# 定义一个儿子类
class Son(Dog):
    pass


# 创建对象
s1 = Son()
s1.eat()
s1.drink()
```

3.【记忆】重写父类方法

3.1【记忆】子类重写父类同名方法：

- 在子类中定义了一个和父类同名的方法(参数也一样)
    - 子类对象调用同名方法，默认只会调用子类的方法

```
"""
```

```
1. 父类的方法不能满足子类的需要，可以对父类的方法重写，重写父类方法的目的是为了给他扩展功能
2. 在子类中定义了一个和父类同名的方法(参数也一样)，即为对父类的方法重写
3. 子类对象调用同名方法，默认只会调用子类的方法
"""


# 定义一个父类，  Animal
class Animal(object):
    def __init__(self):
        print("Animal的初始化")
        self.type = "动物"

    def print_type(self):
        print("Animal类中的print_type:", self.type)


# 定义一个子类，继承与Animal
# class Dog(Animal):    # 子类没有实现自己的__init__和print_type,那么使用父类的.
#     pass

class Dog(Animal):
    # 子类写了一个和父类同名的方法，重写父类方法
```

```
    # 使用子类创建对象会默认调用这个__init__()
    def __init__(self):
        print("Dog类的初始化方法")
        self.type = "可爱的小狗"

    # 子类写了一个和父类同名的方法，重写父类方法
    # 使用子类对象调用print_type，默认调用这个print_type
    def print_type(self):
        print("Dog类中的print_type: ", self.type)


# 定义一个子类对象
dog1 = Dog()
dog1.print_type()
```

3.2【记忆】子类调用父类同名方法：

- 推荐使用：`super().同名方法(形参1, ……)`

```
"""
1. 父类的方法不能满足子类的需要，可以对父类的方法重写，重写父类方法的目的是为了给他扩展功能
```

```
2. 在子类中定义了一个和父类同名的方法(参数也一样)，即为对父类的方法重写
3. 子类调用同名方法，默认只会调用子类的
4. 子类调用父类的同名方法（三种方法）
    4.1 父类名.同名方法(self, 形参1, ……)
    4.2 super(子类名, self).同名方法(形参1, ……)
    4.3 super().同名方法(形参1, ……) # 是 4.2 方法的简写 (推荐使用这种)
"""


# 定义一个父类， Animal
class Animal(object):
    # 添加一个type属性
    def __init__(self):
        print('Animal类中的__init__')
        self.type = '动物'

    # 设计一个方法，打印属性
    def print_type(self):
        print('Animal类中的print_type = ', self.type)


# 定义一个子类，继承与Animal
```

```
class Dog(Animal):
    # __init__和父类的同名，重写父类同名方法
    def __init__(self):
        print('Dog类中的__init__')
        self.type = '可爱的小狗'

    # print_type和父类的同名，重写父类同名方法
    def print_type(self):
        print('Dog类中的print_type = ', self.type)
        print('='*20)

        # 子类中调用父类同名函数的三种方法
        # 方法1： 父类名.同名方法(self, 形参1, ……)
        Animal.__init__(self)
        Animal.print_type(self)
        print('=' * 20)

        # 方法2：super(子类名, self).同名方法(形参1, ……)
        super(Dog, self).__init__()
        super(Dog, self).print_type()
        print('=' * 20)
```

```
        # 方法3：super().同名方法(形参1, ……) # 是 4.2 方法的简写
        # 推荐使用的方法
        super().__init__()
        super().print_type()


# 定义一个子类对象
dog1 = Dog()  # 调用子类的__init__
dog1.print_type()  # 调用子类的print_type()
```

4. 【重点】多继承

4.1【重点】多继承

- 多继承特点：子类有多个父类
- 多继承语法格式：

```
class 子类名(父类1, 父类2, ……):
    pass
```

```
# 多继承：多个父类
# 格式：  class 子类名(父类1,  父类2, ...):


# 定义2个类，它们没有继承关系，是平级的
```

```
# 再定义一个类
class SmallDog(object):
    def eat(self):
        print('吃小东西')


class BigDog(object):
    def drink(self):
        print('大口喝水')


# 定义一个子类，多继承于上面2个父类 （多继承）
class SuperDog(SmallDog, BigDog):
    pass


# 定义子类对象，调用方法
dog1 = SuperDog()
dog1.eat()
dog1.drink()
```

4.2【重点】查看继承顺序

- `类名.__mro__`

```
# 查看类的继承顺序
```

```
# 定义2个类，它们没有继承关系，是平级的
# 再定义一个类
class SmallDog(object):
    def eat(self):
        print("小口吃东西")


class BigDog(object):
    def drink(self):
        print("大口喝水")


# 定义一个子类，多继承于上面2个父类
class SuperDog(SmallDog, BigDog):
    pass


# 查看类的继承顺序 类名.__mro__
# (<class '__main__.SuperDog'>, <class
'__main__.SmallDog'>, <class
'__main__.BigDog'>, <class 'object'>)
print(SuperDog.__mro__)
```

- 默认调用顺序：按照继承顺序找方法

```python
# 定义2个类，它们没有继承关系，是平级的

# 再定义一个类
class SmallDog(object):
    def eat(self):
        print('吃小东西')


class BigDog(object):
    def eat(self):
        print('啃大骨头')


# 定义一个子类，多继承于上面2个父类
class SuperDog(SmallDog, BigDog):
    pass


# 定义子类对象，调用方法
dog1 = SuperDog()
# 子类没有实现eat方法，会按照__mro__继承顺序查询eat方法，
#    由于继承顺序表中SuperDog的下一个类是SmallDog,所以调用SmallDog的eat方法
```

```
# # (<class '__main__.SuperDog'>, <class
'__main__.SmallDog'>, <class
'__main__.BigDog'>, <class 'object'>)
dog1.eat()  # 吃小东西
```

4.3【重点】子类调用父类同名方法：

1. `父类名.同名方法(self, 形参1, ……)`：调用指定的父类
2. `super(类名, self).同名方法(形参1, ……)`：调用继承顺序中类名的下一个类的同名方法
3. `super().同名方法(形参1, ……)`：调用先继承父类的同名方法

```
"""
子类调用父类同名方法：
1. 父类名.同名方法(self, 形参1, ……)：调用指定的父类
2. super(类名, self).同名方法(形参1, ……)：调用继承顺序中类名的下一个类的同名方法
3. super().同名方法(形参1, ……) ：调用先继承父类的同名方法
"""
```

```python
class SmallDog(object):
    def eat(self):
        print('吃小东西')


class BigDog(object):
    def eat(self):
        print('啃大骨头')


# 定义一个子类，多继承于上面2个父类
class SuperDog(SmallDog, BigDog):
    def eat(self):
        print("吃蟠桃")
        print("=" * 20)

        # 多继承中子类调用父类同名方法：
        # 1. 父类名.同名方法(self, 形参1, ……)：调用指定的父类
        SmallDog.eat(self)
        print("=" * 20)

        # 2. super(类名, self).同名方法(形参1, ……)：调用继承顺序中类名的下一个类的同名方法
```

```
        # (<class '__main__.SuperDog'>,
<class '__main__.SmallDog'>, <class
'__main__.BigDog'>, <class 'object'>)
        super(SmallDog, self).eat()  #
BigDog.eat()
        print("=" * 20)

        # 3. super().同名方法(形参1, ……) ：调
用先继承父类的同名方法
        # super(SuperDog, self).eat()
        super().eat()


dog1 = SuperDog()
dog1.eat()
```

5.【知道】私有和继承

私有方法、属性不能直接继承使用

```
# 私有和继承：私有属性和方法不能直接继承使用


# 定义一个父类，  Animal
class Animal(object):
```

```
    def __init__(self):
        # 定义一个私有属性
        self.__type = "动物"

    # 定义一个私有方法
    def __leave(self):
        print("休产假3个月")

    def print_info(self):
        # 通过父类的公有方法可以间接访问父类的私有属性和私有方法
        print(self.__type)
        self.__leave()


# 定义一个子类
class Dog(Animal):
    def test(self):
        # 父类的私有属性和私有方法不能直接继承使用
        # print(self.__type) # AttributeError: 'Dog' object has no attribute '_Dog__type'
        # self.__leave() # AttributeError: 'Dog' object has no attribute '_Dog__leave'
        pass
```

```
# 创建子类对象
dog1 = Dog()
dog1.test()
dog1.print_info()
```

6.【了解】多态

- 多态：多种形态，调用同一个函数，不同表现
- 因为Python是动态语言，站在用户的角度，本身就是多态，不存在非多态的情况

- 实现多态的步骤:
  - 实现继承关系
  - 子类重写父类方法
  - 通过对象调用该方法

```
"""
1. 多态：多种形态，调用同一个函数，不同表现

2. 实现多态的步骤：
```

```
    1. 实现继承关系
    2. 子类重写父类方法
    3. 通过对象调用该方法

"""


# 定义一个父类,  Animal
class Animal(object):
    def eat(self):
        print("吃东西")


# 定义一个子类Dog，继承于Animal
class Dog(Animal):
    def eat(self):
        print("啃骨头")


# 定义一个子类Cat，继承于Animal
class Cat(Animal):
    def eat(self):
        print("吃小鱼")


# 定义一个函数，用于测试多态
```

```
def func(temp):
    temp.eat()


# 创建子类对象
d = Dog()
c = Cat()
# 调用同一个函数，不同表现
func(d)
func(c)
```

7. 【记忆】属性

7.1. 【记忆】类属性定义

- 类属性的定义方式：定义在**类里面，类方法外面**的变量就是**类属性**
- 类属性可以使用 **类名** 或 **实例对象** 访问，**推荐使用类名访问**

```
# 定义类
class 类名(object):
    类属性变量 = 数值1

    def __init__(self):
        pass
```

```
"""
实例属性:
    1. 通过在__init__方法里面给实例对象添加的属性
    2. 在类的外面，直接通过实例对象添加的属性
    3. 实例属性 必须通过 实例对象 才能访问
类属性:
    1. 定义在 类里面，类方法外面 的变量就是 类属性
    2. 类属性可以使用 类名 或 实例对象 访问，推荐使用类名
"""


# python下万物皆对象
# def foo():
#     pass
#
# class Dog(object):
```

```
#     pass
#
#
# # python下万物皆对象
# print(Dog)
# print(id(Dog))
# print(type(Dog))    # <class 'type'> 类在python底层实现中,它也是一个对象
# dog1 = Dog()
# print(id(dog1))
# print(type(dog1))   # <class '__main__.Dog'>
# print(foo)
# print(type(foo))  # <class 'function'>
# print(id(foo))
# print(type(20))  # <class 'int'>


# 实例属性:
#     1. 通过在__init__方法里面给实例对象添加的属性
#     2. 在类的外面, 直接通过实例对象添加的属性
#     3. 实例属性 必须通过 实例对象 才能访问

class Dog(object):
    def __init__(self, _name):
```

```
        # 1. 通过在__init__方法里面给实例对象添加的属性
        # 添加实例属性 : self就是当前对象(实例)本身,self.属性就是实例属性
        self.name = _name


dog1 = Dog("旺财")
# 添加实例属性: 2. 在类的外面, 直接通过实例对象添加的属性
dog1.age = 2

# 3.  实例属性必须通过 实例对象 才能访问
print(dog1.name)
print(dog1.age)


# 类属性:
#     1. 定义在 类里面, 类方法外面 的变量就是 类属性
#     2. 类属性可以使用 类名 或 实例对象 访问, 推荐使用类名

class Cat(object):
    # 1. 定义在 类里面, 类方法外面 的变量就是 类属性
```

```
    # 类属性
    count = 0

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


# 2. 类属性可以使用 类名 或 实例对象 访问，推荐使用类名
print(Cat.count)
cat1 = Cat("小黑")

print(cat1.count)
```

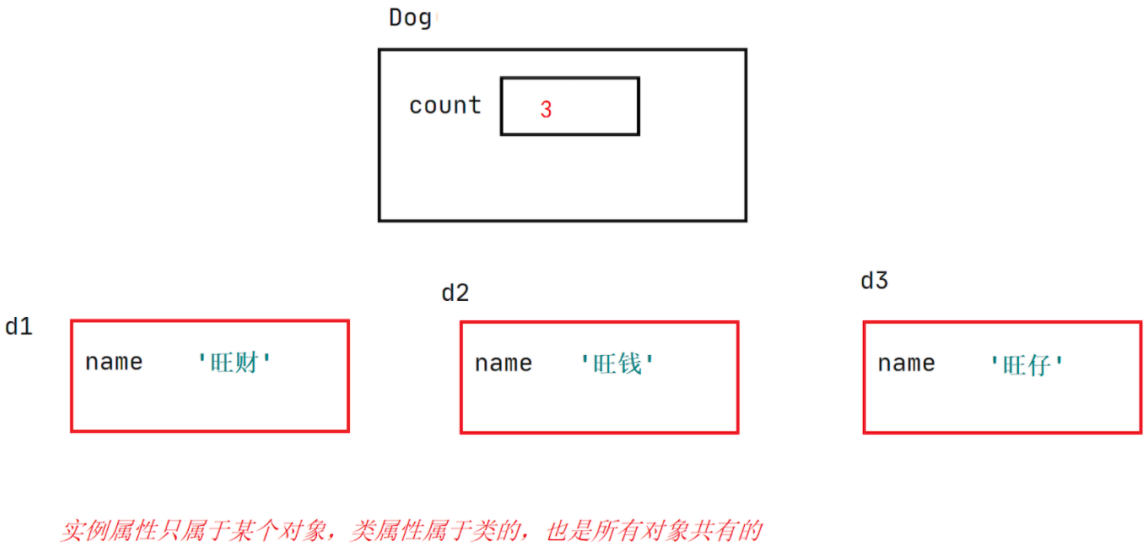
7.2. 【记忆】类属性和实例属性的区别

- 类属性属于类的，类所有对象**共享**
- 实例属性只属于某个实例对象，实例属性只能通过实例对象名访问

```
"""
1. 定义一个类属性count，用于记录实例对象初始化的次数
```

```
2. __init__添加实例属性name，每初始化1次，类属性count加1
"""


class Dog(object):
    # 定义类属性：统计使用这个Dog类创建了多少对象，是所有实例对象公用的
    count = 0

    def __init__(self, _name):
        # 实例属性：每一个实例对象特有的
        self.name = _name
        # 每次调用__init__，count计数+1
        Dog.count += 1


print(Dog.count)
dog1 = Dog("旺财")
print(dog1.name, Dog.count)

dog2 = Dog("旺钱")
print(dog2.name, Dog.count)

dog3 = Dog("旺仔")
print(dog3.name, Dog.count)
```

```
# 3 3 3 类属性是所有实例对象共有的.
print(dog1.count, dog2.count, dog3.count)
# 旺财 旺钱 旺仔 实例属性是每一个实例对象特有的.
print(dog1.name, dog2.name, dog3.name)
```



实例属性只属于某个对象，类属性属于类的，也是所有对象共有的

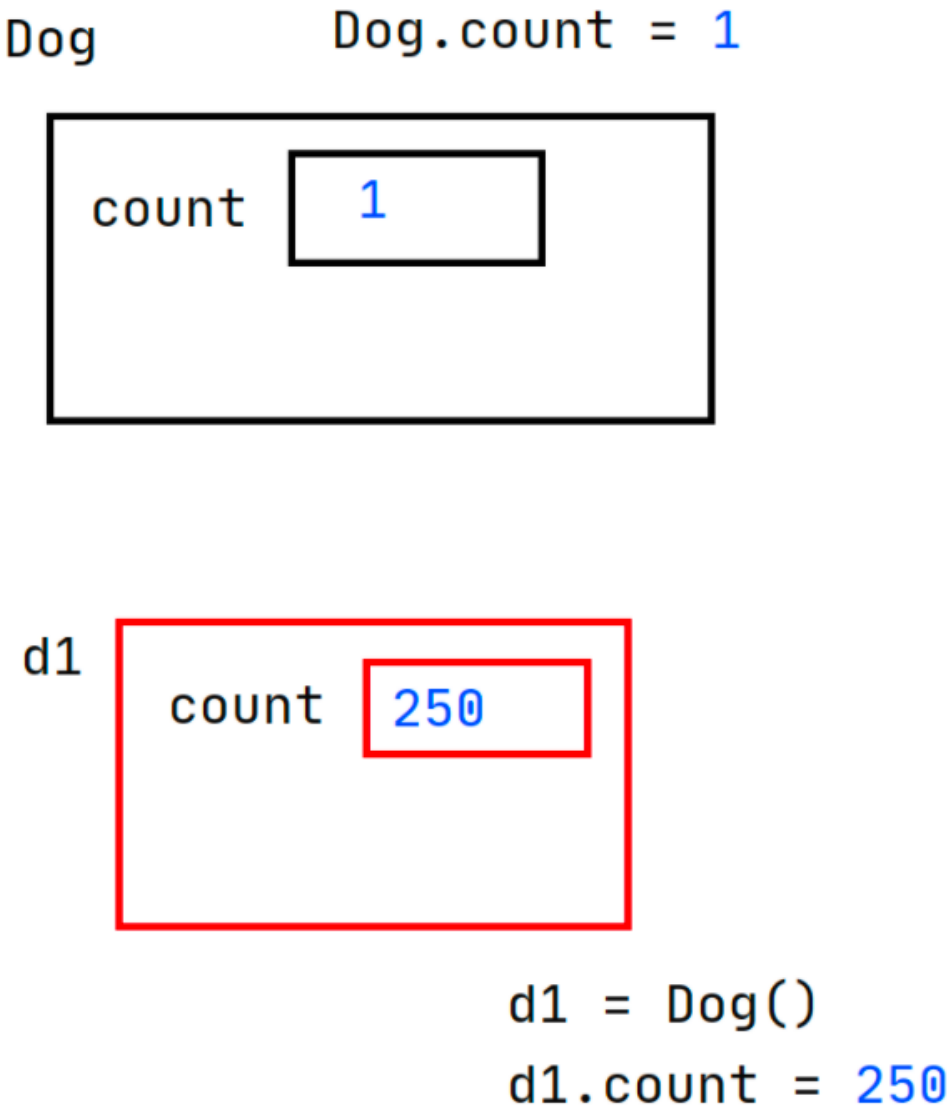
7.3. 修改和访问类属性注意点:

- 修改类属性注意:
  - 类属性修改，**只能通过类名修改**，不能通过对象名修改

```
# 类属性修改，只能通过类名修改，不能通过对象名修改
# 对象名.变量 = 数据 默认操作给实例对象添加实例属性，已经不能操作类属性
```

```python
# 如果类属性名字和实例属性名字相同，实例对象名只能操作实例属性

class Dog(object):
    # 类属性
    count = 0


d1 = Dog()
print(Dog.count)     # 0 使用类名访问类属性
print(d1.count)      # 0 使用实例对象访问类属性

Dog.count = 1    # 只能使用类名修改类属性
print(Dog.count)     # 1
print(d1.count)      # 1

d1.count = 250   # 使用实例对象添加一个实例属性,只不过这个实例属性的名字和类属性名字一样.
print(Dog.count)  # 1
print(d1.count)  # 250 由于类属性和实例属性名字一样,所以通过实例对象只能访问实例属性, 无法访问到类属性
```

-



- 访问建议
  - 建议:**类属性使用类名访问**, 实例属性使用实例对象访问.

```
# 如果类属性和实例属性同名，实例对象名只能操作实例属性


class Dog(object):
```

```
    # 类属性
    count = 666

    def __init__(self):
        # 实例属性
        self.count = 250


d1 = Dog()
print(d1.count, Dog.count)
#       250          666
```

7.4. 私有类属性

```
class Dog(object):
    # 私有的类属性, 不能在类的外部访问, 只能在类的内部访问
    __count = 0

    def print_count(self):
        print(Dog.__count)


# AttributeError: type object 'Dog' has no attribute '__count'
```

```
# print(Dog.__count)

d1 = Dog()
d1.print_count()
```

8. 【记忆】方法

8.1【记忆】类方法定义

- 类方法: 为了方便处理类属性

```
"""
类方法：为了方便处理类属性. 在没有创建实例对象时,也可以调用类方法操作类属性.
    1. 用装饰器 @classmethod 来标识其为类方法
    2. 一般以 cls 作为第一个参数，代表当前这个类，这个参数不用人为传参，解释器会自动处理
    3. 类方法调用:
        3.1 类名.类方法()    推荐用法
        3.2 实例对象名.类方法()
"""


class Dog(object):
    # 类属性
    count = 0
```

```
    # 实例方法：创建实例对象后才能调用的方法
    # def print_count(self):

    # 1. 用装饰器 @classmethod 来标识其为类方法
    @classmethod
    # 2. 一般以 cls 作为第一个参数，代表当前这个类，这个参数不用人为传参，解释器会自动处理
    def print_count(cls):
        # print(Dog.count)
        # print(cls)
        print(cls.count)


# 3. 类方法调用：
# 3.1 类名.类方法()    推荐用法
Dog.print_count()  # print_count(Dog)
# 3.2 实例对象名.类方法()
# d1 = Dog()
# d1.print_count()   #
print_count(type(d1))
```

8.2 【记忆】静态方法定义

- **取消不需要的参数传递，有利于 减少不必要的内存占用和性能消耗**

```
"""
静态方法:
    1. 需要通过装饰器@staticmethod来进行修饰默认情况下
    2. 既不传递类对象也不传递实例对象（形参没有self/cls）
    3. 静态方法调用:
        3.1 类名.静态方法()     推荐用法
        3.2 实例对象名.静态方法()
"""


class Dog(object):
    # 1. 需要通过装饰器@staticmethod来进行修饰默认情况下
    @staticmethod
    # 2. 既不传递类对象也不传递实例对象（形参没有self/cls）
    def foo():
        # 实例属性: self.属性
        # 类属性: cls.属性
```

```
        print("一个与实例属性和类属性无关的函数")


#     3. 静态方法调用:
#           3.1 类名.静态方法()     推荐用法
Dog.foo()
#           3.2 实例对象名.静态方法()
# d1 = Dog()
# d1.foo()
```

8.3 【记忆】类方法、实例方法、静态方法的区别

- 定义方法区别

```
class 类名(object):
    # 实例方法定义
    def 实例方法名(self):
        pass

    # 类方法
    @classmethod
    def 类方法名(cls):
        pass
```

```
    # 静态方法
    @staticmethod
    def 静态方法名():
        pass
```

- 调用方法区别
  - 实例方法必须通过**实例对象名**调用：创建完实例对象后，通过实例对象调用
  - 类方法、静态方法通过 **实例对象** 和 **类对象(类名)** 调用，推荐使用类名
- 定义原则:
  - 当方法中需要使用实例属性，定义成实例方法。
  - 当方法中需要使用类属性，定义成类方法。
  - 当方法中不需要使用类属性和实例属性，定义成静态方法。
  - 当方法中需要使用类属性和实例属性，定义成实例方法。