

[笔记][黑马 Python 之 Python 面向对象 - 2]

Python

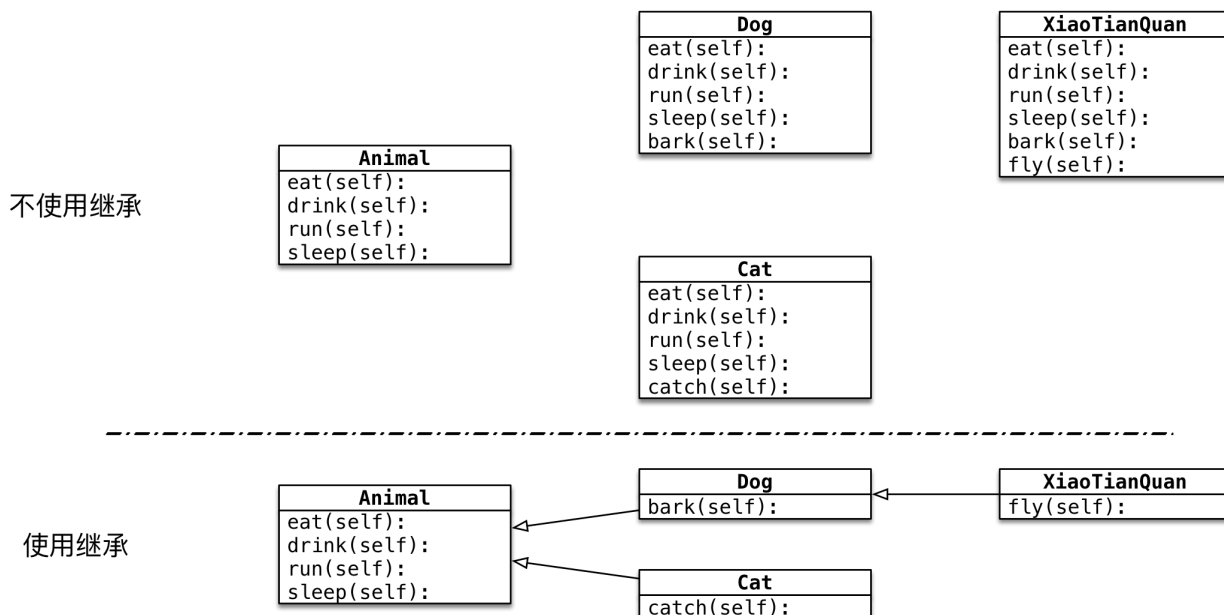
[笔记][黑马 Python 之 Python 面向对象 - 2]

- 031. 单继承-01-问题的抛出-单纯封装可能会出现重复的代码
- 032. 单继承-02-继承的概念和语法
- 033. 单继承-03-继承的相关术语-继承和派生
- 034. 单继承-04-继承的传递性
- 035. 单继承-05-继承传递性的注意事项
- 036. 方法的重写-01-覆盖父类方法，重写子类方法实现
- 037. 方法的重写-02-扩展父类方法，super对象调用父类方法
- 038. 方法的重写-03-使用父类方法名调用父类方法
- 039. 私有属性和方法-01-子类对象不能直接访问
- 040. 私有属性和方法-02-通过父类方法间接访问
- 041. 多继承-01-概念、语法和基本演练
- 042. 多继承-02-注意父类之间不要有重名方法或属性
- 043. 多继承-03-MRO方法搜索顺序
- 044. 多继承-04-新式类和经典类
- 045. 多态-01-基本概念
- 046. 多态-02-案例演练
- 047. 类属性-01-创建对象的过程以及实例的概念
- 048. 类属性-02-类是一个特殊的对象
- 049. 类属性-03-类属性的定义及使用
- 050. 类属性-04-属性查找机制-向上查找
- 051. 类属性-05-使用对象名.类属性的赋值语句会创建实例属性
- 052. 类方法-01-基本语法
- 053. 类方法-02-案例演练
- 054. 静态方法-01-应用场景和定义方式
- 055. 方法综合-01-案例分析
- 056. 方法综合-02-案例演练
- 057. 方法综合-03-确定方法类型的套路
- 058. 单例-01-设计模式和单例设计模式概念
- 059. 单例-02-new方法的作用
- 060. 单例-03-重写new方法
- 061. 单例-04-单例设计模式思路分析
- 062. 单例-05-单例设计模式代码实现
- 063. 单例-06-初始化动作只执行一次

031. 单继承-01-问题的抛出-单纯封装可能会出现重复的代码

面向对象三大特性

- **封装** 根据 **职责** 将 **属性** 和 **方法** **封装** 到一个抽象的 **类** 中
- **继承** 实现代码的重用，相同的代码不需要重复的编写
- **多态** 不同的对象调用相同的方法，产生不同的执行结果，增加代码的灵活性



032. 单继承-02-继承的概念和语法

继承的概念：子类拥有父类的所有方法和属性

继承的语法

```
class 类名(父类名):  
  
    pass
```

- 子类继承自父类，可以直接享受父类中已经封装好的方法，不需要再次开发
- 子类中应该根据职责，**封装子类特有的属性和方法**

033. 单继承-03-继承的相关术语-继承和派生

- `Dog` 类是 `Animal` 类的子类，`Animal` 类是 `Dog` 类的父类，`Dog` 类从 `Animal` 类继承
- `Dog` 类是 `Animal` 类的派生类，`Animal` 类是 `Dog` 类的基类，`Dog` 类从 `Animal` 类派生

034. 单继承-04-继承的传递性

- `C` 类从 `B` 类继承，`B` 类又从 `A` 类继承
- 那么 `C` 类就具有 `B` 类和 `A` 类的所有属性和方法

子类拥有父类以及父类的父类中封装的所有属性和方法

```
class Animal:
    def eat(self):
        print('吃---')

    def drink(self):
        print('喝---')

    def run(self):
        print('跑---')

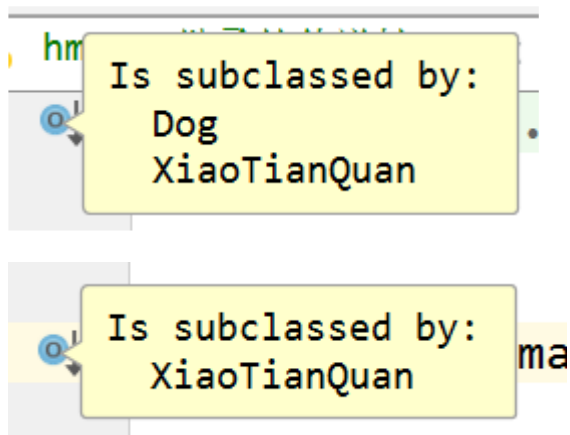
    def sleep(self):
        print('睡---')

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

class XiaoTianQuan(Dog):
    def fly(self):
        print('我会飞')

# 创建一个哮天犬的对象
xtq = XiaoTianQuan()
xtq.fly()
xtq.bark()
xtq.eat()
```

PyCharm 会有子类的智能提示



035. 单继承-05-继承传递性的注意事项

提问

哮天犬 能够调用 `Cat` 类中定义的 `catch` 方法吗？

答案

不能，因为 哮天犬 和 `Cat` 之间没有 继承 关系

```
class Animal:
    def eat(self):
        print('吃---')

    def drink(self):
        print('喝---')

    def run(self):
        print('跑---')

    def sleep(self):
        print('睡---')

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

class XiaoTianQuan(Dog):
    def fly(self):
        print('我会飞')

class Cat(Animal):
    def catch(self):
        print('抓老鼠')
```

```
# 创建一个哮天犬的对象
xtq = XiaoTianQuan()
xtq.fly()
xtq.bark()
xtq.eat()

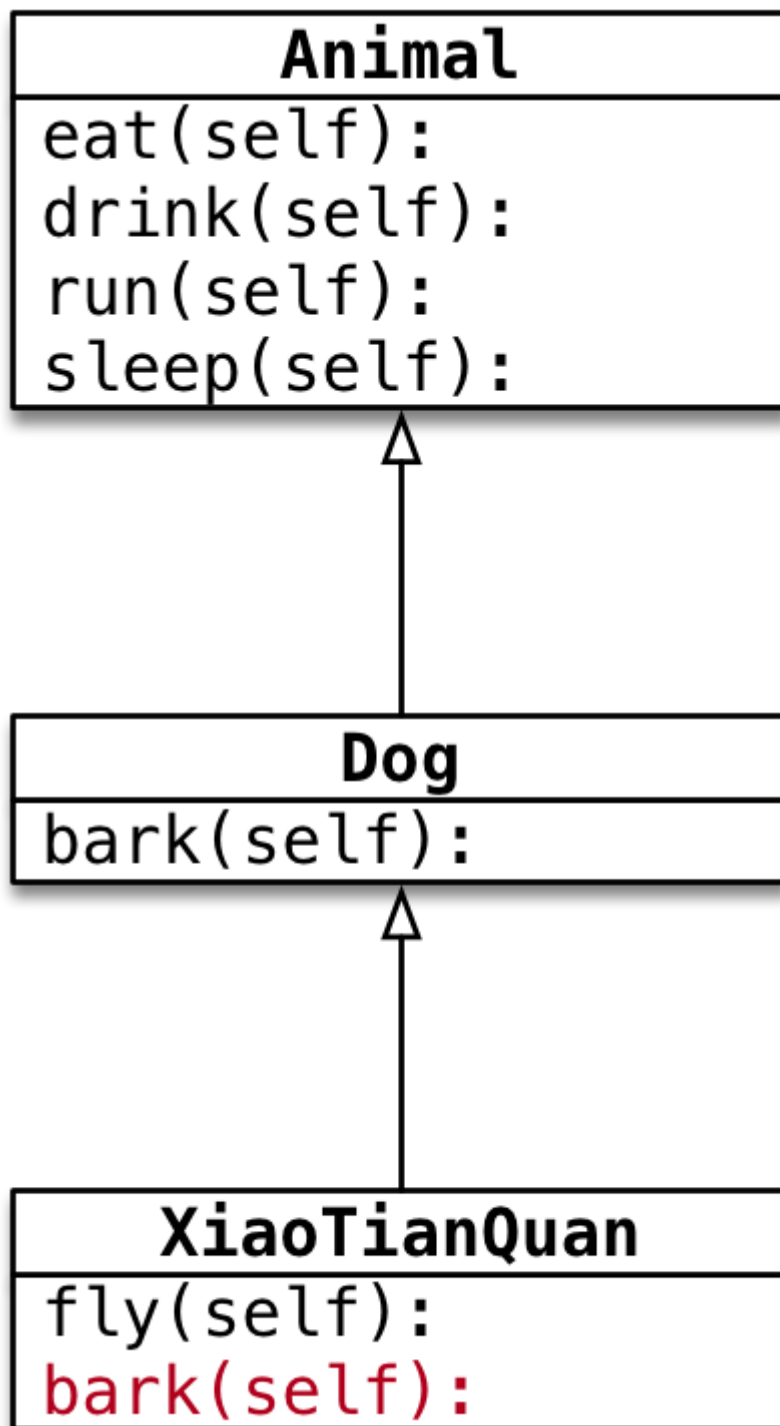
# 没有继承关系
# 哮天犬的对象不能调用猫类方法
# xtq.catch()
```

036. 方法的重写-01-覆盖父类方法，重写子类方法实现

- **子类** 拥有 **父类** 的所有 **方法** 和 **属性**
- **子类** 继承自 **父类**，可以直接享受父类中已经封装好的方法，不需要再次开发

应用场景

- 当 **父类** 的方法实现不能满足子类需求时，可以对方法进行 **重写**(**override**)



重写 父类方法有两种情况：

- **覆盖** 父类的方法
- 对父类方法进行 **扩展**

覆盖父类的方法

- 如果在开发中，**父类的方法实现** 和 **子类的方法实现**，**完全不同**
- 就可以使用 **覆盖** 的方式，在子类中 **重新编写** 父类的方法实现

具体的实现方式：**在子类中定义一个和父类同名的方法并且实现**

重写之后，在运行时，只会调用 **子类中重写的方法**，而不再会调用 **父类封装的方法**

037. 方法的重写-02-扩展父类方法，super对象调用父类方法

对父类方法进行扩展

- 如果在开发中，子类的方法实现中 **包含** 父类的方法实现
即**父类原本封装的方法实现** 是 **子类方法的一部分**
- 就可以使用 **扩展** 的方式
 1. 在子类中 **重写** 父类的方法
 2. 在需要的位置使用 `super().父类方法` 来调用父类方法的执行
 3. 代码其他的位置针对子类的需求，编写 **子类特有的代码实现**

关于 `super`

- 在 `Python` 中 `super` 是一个 **特殊的类**
- `super()` 就是使用 `super` 类创建出来的对象
- 最常使用的场景就是 **在重写父类方法时，调用在父类中封装的方法实现**

【!】注意

`Python 2.x` 使用 `super` 需要写成 `super(子类名, self).xxx`
而 `Python 3.x` 只要写成 `super().xxx` 即可

```
class Animal:
    def eat(self):
        print('吃---')

    def drink(self):
        print('喝---')

    def run(self):
        print('跑---')

    def sleep(self):
        print('睡---')

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

```

class XiaoTianQuan(Dog):
    def fly(self):
        print('我会飞')

    def bark(self):
        # 1. 针对子类特有的需求, 编写代码
        print('神一样的叫唤...')
        # 2. 使用 super(). 调用原本在父类中封装的方法
        super().bark()
        # 3. 增加其他子类的代码
        print('#$%*#@#$%*#@')

xtq = XiaoTianQuan()
# 如果子类 重写override 了父类方法
# 在使用子类对象调用方法时, 会调用子类中重写的方法
xtq.bark()

```

038. 方法的重写-03-使用父类方法名调用父类方法

在 Python 2.x 的早期版本没有 `super`

调用父类方法的另外一种方式

- 在 Python 2.x 时, 如果需要调用父类的方法, 还可以使用以下方式:
`父类名.方法(self)`
- 这种方式, 目前在 Python 3.x 还支持这种方式
- 这种方法 **不推荐使用**, 因为一旦 **父类发生变化**, 方法调用位置的 **类名** 同样需要修改

提示

- 在开发时, **父类名** 和 `super()` 两种方式不要混用
- 如果使用 **当前子类名** 调用方法, 会形成递归调用, **出现死循环**

```

class Animal:
    def eat(self):
        print('吃---')

    def drink(self):
        print('喝---')

    def run(self):
        print('跑---')

```



```

def sleep(self):
    print('睡---')

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

class XiaoTianQuan(Dog):
    def fly(self):
        print('我会飞')

    def bark(self):
        # 1. 针对子类特有的需求, 编写代码
        print('神一样的叫唤...')
        # 2.1 使用 super(). 调用原本在父类中封装的方法
        # super().bark()
        # 2.2 使用 父类名.方法(self) 来调用父类方法
        Dog.bark(self)
        # 3. 增加其他子类的代码
        print('##$*#@#$*#@')

xtq = XiaoTianQuan()
# 如果子类 重写override 了父类方法
# 在使用子类对象调用方法时, 会调用子类中重写的方法
xtq.bark()

```

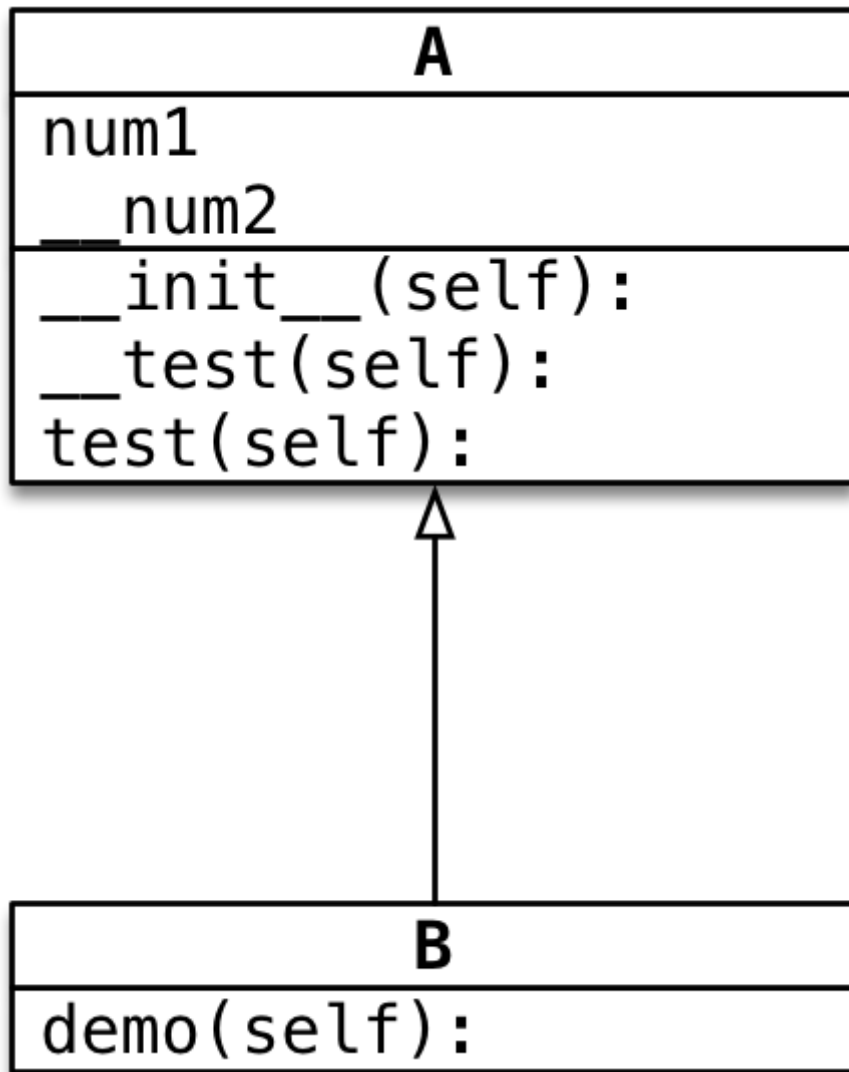
039. 私有属性和方法-01-子类对象不能直接访问

父类的 **私有属性** 和 **私有方法**

- 子类对象 **不能** 在自己的方法内部, 直接访问父类的私有属性或私有方法
- 子类对象可以通过父类的公有方法**间接**访问到私有属性或私有方法

私有属性、方法是对象的隐私, 不对外公开, 外界以及子类都不能直接访问
私有属性、方法 通常用于做一些内部的事情

示例



- B 的对象不能直接访问 `__num2` 属性
- B 的对象不能在 `demo` 方法内访问 `__num2` 属性
- B 的对象可以在 `demo` 方法内，调用父类的 `test` 方法
- 父类的 `test` 方法内部，能够访问 `__num2` 属性和 `__test` 方法

040. 私有属性和方法-02-通过父类方法间接访问

```
class A:
    def __init__(self):
        self.num1 = 100
        self.__num2 = 200

    def __test(self):
        print('私有方法 %d %d' % (self.num1, self.__num2))

    def test(self):
        print('父类的公有方法 %d' % self.__num2)
```

```

self.__test()

class B(A):
    def demo(self):
        # 1. 在子类的对象方法中，不能访问父类的私有属性
        # print('访问父类的私有属性 %d' % self.__num2)

        # 2. 在子类的对象方法中，不能调用父类的私有方法
        # self.__test()

        # 3. 访问父类的共有属性
        print('子类方法 %d' % self.num1)

        # 4. 调用父类的公有方法
        self.test()

# 创建一个子类对象
b = B()
print(b)
# 在外界不能直接访问对象的私有属性/调用私有方法
# print(b.__num2)
# b.__test()
b.demo()

# 在外界访问父类的公有属性/调用公有方法
print(b.num1)
b.test()

```

041. 多继承-01-概念、语法和基本演练

概念

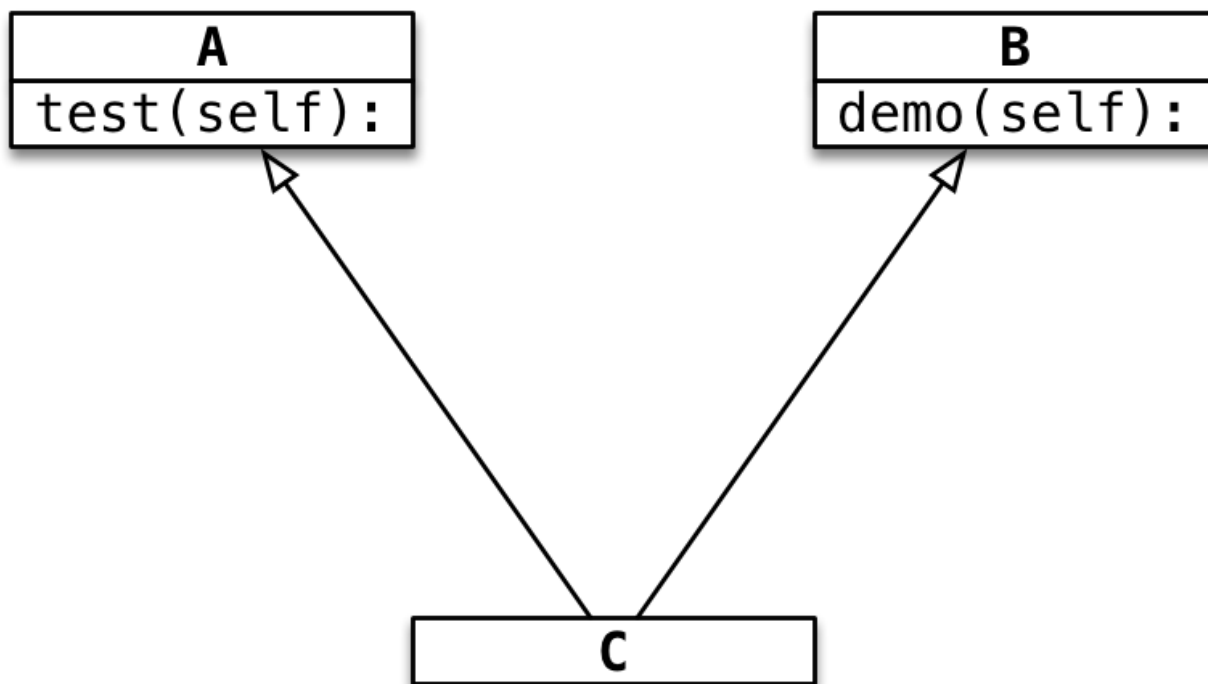
- **子类** 可以拥有 **多个父类**，并且具有 **所有父类** 的 **属性** 和 **方法**
例如：**孩子** 会继承自己 **父亲** 和 **母亲** 的 **特性**

语法

```

class 子类名(父类名1, 父类名2...):
    pass

```



```
class A:
    def test(self):
        print('test 方法')

class B:
    def demo(self):
        print('demo 方法')

class C(A, B):
    """多继承可以让子类对象，同时具有多个父类的属性和方法"""
    pass

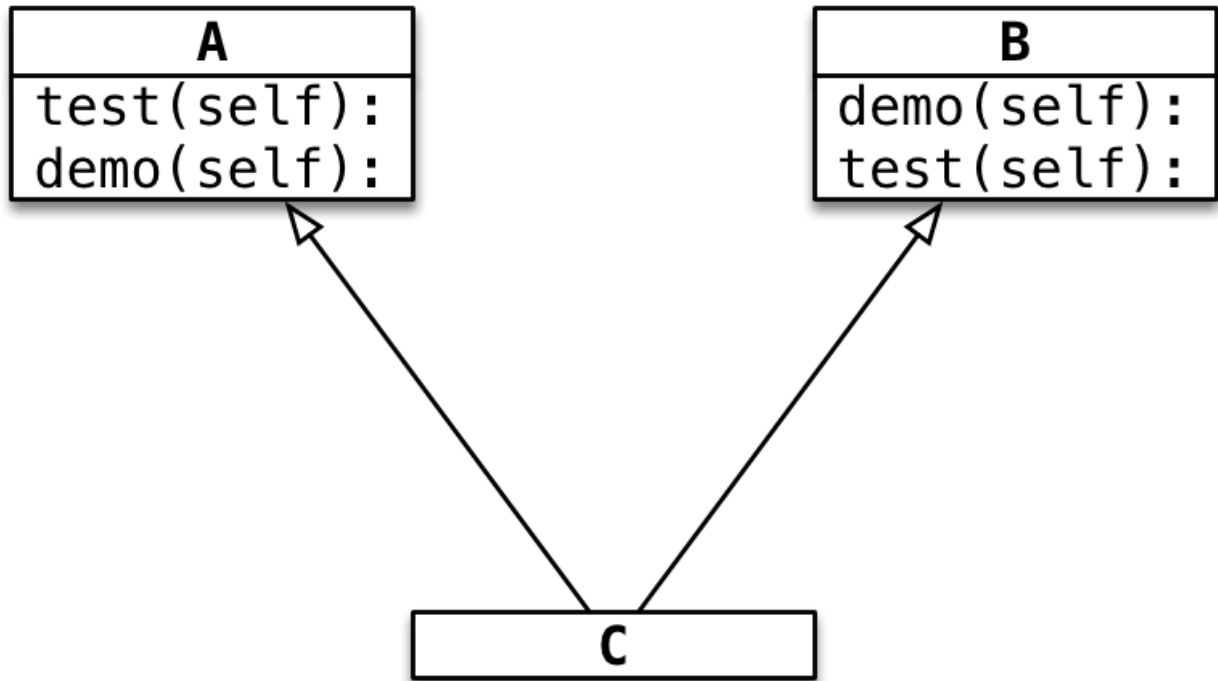
# 创建子类对象
c = C()
c.test()
c.demo()
```

042. 多继承-02-注意父类之间不要有重名方法或属性

问题的提出

如果不同的父类中存在同名的方法，子类对象在调用方法时，会调用哪一个父类中的方法呢？

提示：开发时，应该尽量避免这种容易产生混淆的情况！——如果父类之间存在 **同名的属性或者方法**，应该 **尽量避免使用多继承**，因为可读性较差。



```
class A:
    def test(self):
        print('A --- test 方法')

    def demo(self):
        print('A --- demo 方法')

class B:
    def test(self):
        print('B --- test 方法')

    def demo(self):
        print('B --- demo 方法')

class C(A, B):
    """多继承可以让子类对象，同时具有多个父类的属性和方法"""
    pass

# 创建子类对象
c = C()
c.test()
c.demo()
```

043. 多继承-03-MRO方法搜索顺序

- Python 中针对 类 提供了一个 内置属性 `__mro__` 可以查看 方法搜索顺序，它是一个 元组
 - MRO 是 `method resolution order`，主要用于在多继承时判断方法、属性的调用路径
- 【!】笨办法学Python里面叫做方法解析顺序

```
print(C.__mro__)
```

输出结果

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

- 在搜索方法时，是按照 `__mro__` 的输出结果 从左至右 的顺序查找的
- 如果在当前类中 找到方法，就直接执行，不再搜索
- 如果 没有找到，就查找下一个类中是否有对应的方法，如果找到，就直接执行，不再搜索
- 如果找到最后一个类，还没有找到方法，程序报错

`object` 类是所有类的基类

044. 多继承-04-新式类和经典类

新式类与旧式（经典）类

`object` 是 Python 为所有对象提供的 基类，提供有一些内置的属性和方法，可以使用 `dir` 函数查看

- 新式类：以 `object` 为基类的类，推荐使用
- 经典类/旧式类：不以 `object` 为基类的类，不推荐使用
- 在 Python 3.x 中定义类时，如果没有指定父类，会默认使用 `object` 作为该类的 基类—— Python 3.x 中定义的类 都是 新式类
- 在 Python 2.x 中定义类时，如果没有指定父类，则不会以 `object` 作为 基类

新式类 和 经典类 在多继承时 —— 会影响到方法的搜索顺序

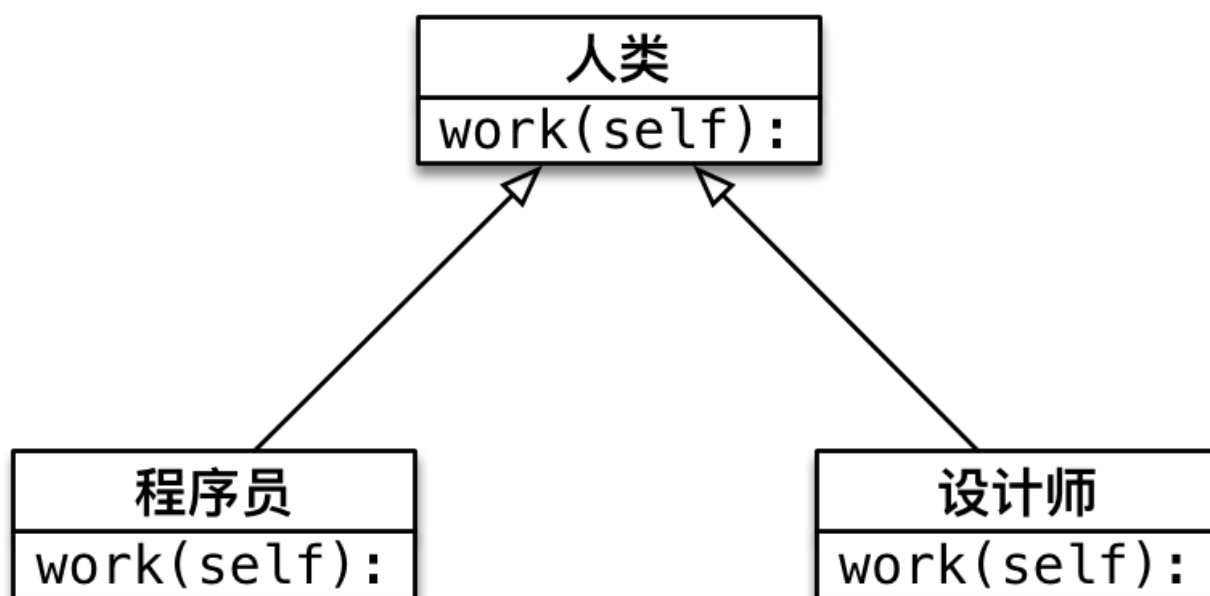
为了保证编写的代码能够同时在 Python 2.x 和 Python 3.x 运行！
今后在定义类时，如果没有父类，建议统一继承自 `object`

```
class 类名(object):  
    pass
```

045. 多态-01-基本概念

面向对象三大特性

1. **封装**：根据职责将属性和方法封装到一个抽象的类中
 - 定义类的准则
2. **继承**：实现代码的重用，相同的代码不需要重复的编写
 - 设计类的技巧
 - 子类针对自己特有的需求，编写特定的代码
3. **多态**：不同的子类对象调用相同的父类方法，产生不同的执行结果
 - 多态可以**增加代码的灵活度**
 - 以**继承和重写父类方法**为前提
 - 是调用方法的技巧，**不会影响到类的内部设计**

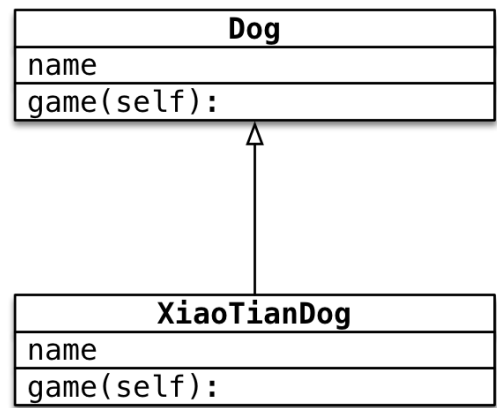


046. 多态-02-案例演练

需求

- 在 `Dog` 类中封装方法 `game`
普通狗只是简单的玩耍
- 定义 `XiaoTianDog` 继承自 `Dog`，并且重写 `game` 方法
哮天犬需要在天上玩耍
- 定义 `Person` 类，并且封装一个 **和狗玩** 的方法
在方法内部，直接让 **狗对象** 调用 `game` 方法

Person
name
game_with_dog(self, dog):



```

class Dog(object):
    def __init__(self, name):
        self.name = name

    def game(self):
        print('%s 蹦蹦跳跳的玩耍...' % self.name)

class XiaoTianDog(Dog):
    def game(self):
        print('%s 飞到天上去玩耍...' % self.name)

class Person(object):
    def __init__(self, name):
        self.name = name

    def game_with_dog(self, dog):
        print('%s 和 %s 快乐的玩耍...' % (self.name, dog.name))
        # 让狗玩耍
        dog.game()

# 1. 创建一个狗对象
# wangcai = Dog('旺财')
wangcai = XiaoTianDog('飞天旺财')

# 2. 创建一个小明对象
xiaoming = Person('小明')

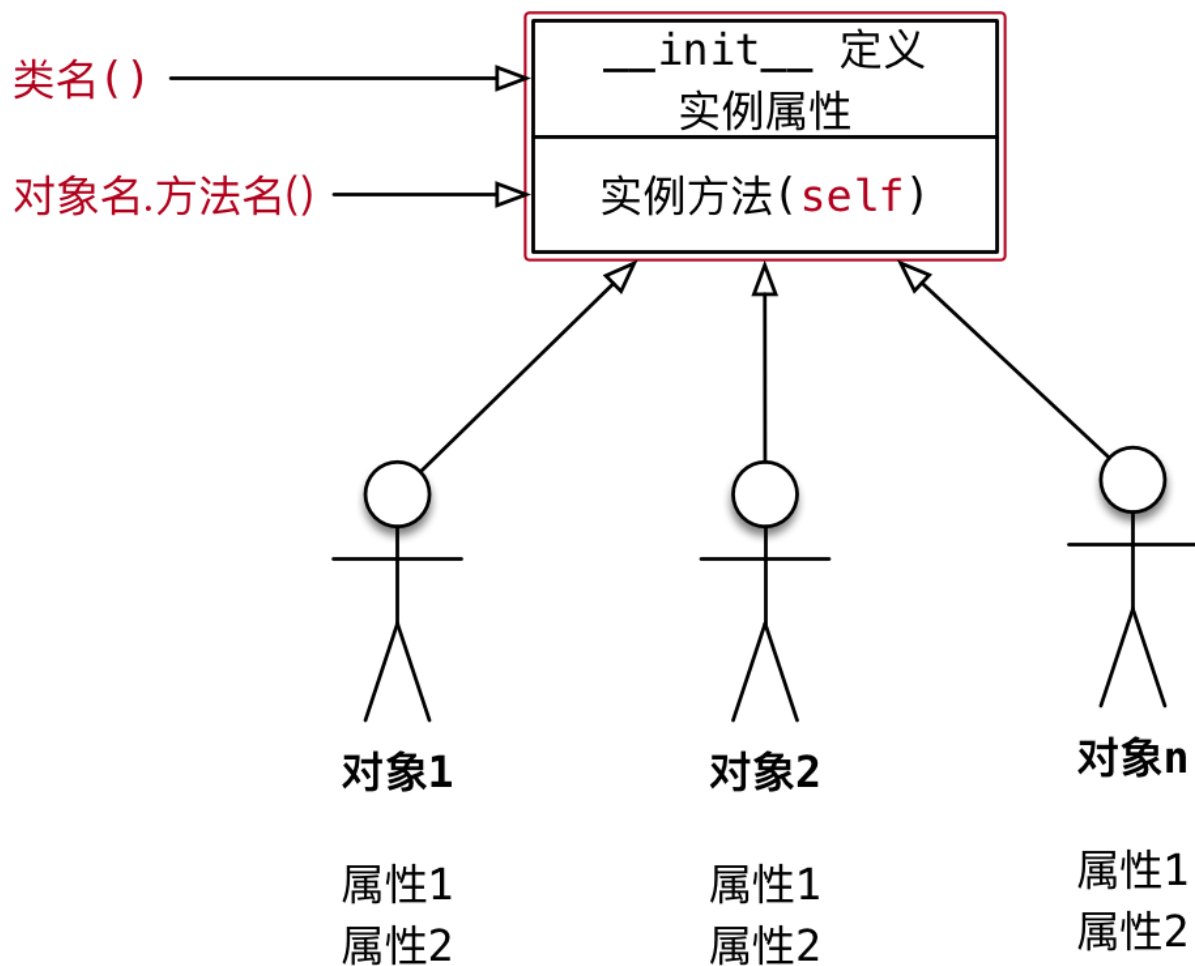
# 3. 让小明调用和狗玩的方法
xiaoming.game_with_dog(wangcai)
  
```

案例小结

Person 类中只需要让 **狗对象** 调用 **game** 方法，而不关心具体是 **什么狗**
game 方法是在 **Dog** 父类中定义的
 在程序执行时，传入不同的 **狗对象** 实参，就会产生不同的执行效果

047. 类属性-01-创建对象的过程以及实例的概念

1. 使用面向对象开发，第 **1** 步是**设计类**
2. 使用 **类名()** 创建对象，**创建对象** 的动作有两步：
 - 1) 在内存中为对象 **分配空间**
 - 2) 调用初始化方法 **`__init__`** 为 **对象初始化**
3. 对象创建后，**内存** 中就有了一个对象的 **实实在在** 的存在 —— **实例**



因此，通常也会把：

- 创建出来的 **对象** 叫做 **类的实例**
- 创建对象的 **动作** 叫做 **实例化**
- 对象的属性叫做 **实例属性**
- 对象调用的方法叫做 **实例方法**

在程序执行时：

- 对象各自拥有自己的 **实例属性**

- 调用对象方法，可以通过 `self.`
 - 访问自己的属性
 - 调用自己的方法

结论

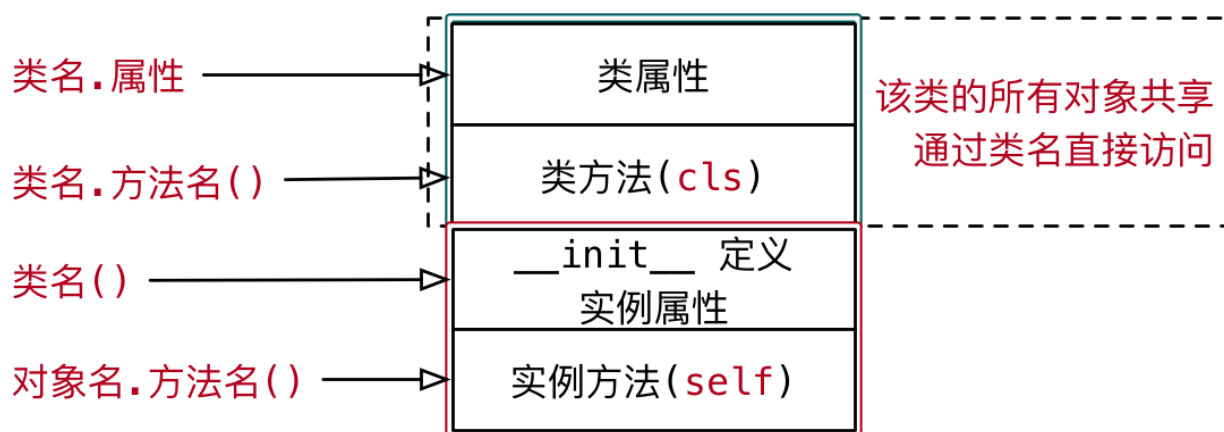
- 每一个对象都有自己 **独立的内存空间**，保存 **各自不同的属性**
- **多个对象的方法**，在内存中只有一份，在调用方法时，需要把对象的引用传递到方法内部

048. 类属性-02-类是一个特殊的对象

Python 中 **一切皆对象**：

- `class AAA:` 定义的类属于 **类对象**
- `obj1 = AAA()` 属于 **实例对象**

- 在程序运行时，类 同样会被加载到内存
- 在 Python 中，类是一个特殊的对象 —— **类对象**
- 在程序运行时，**类对象在内存中只有一份**，使用一个类可以创建出很多个对象实例
- **除了封装实例的属性和方法外**，类对象还可以拥有自己的属性和方法
 - 类属性
 - 类方法
- 通过 `类名.` 的方式可以 **访问类的属性** 或者 **调用类的方法**



049. 类属性-03-类属性的定义及使用

- **类属性** 就是在类对象中定义的属性
- 通常用来记录 **与这个类相关** 的特征
- **类属性** 不会用于记录 **具体对象的特征**

示例需求

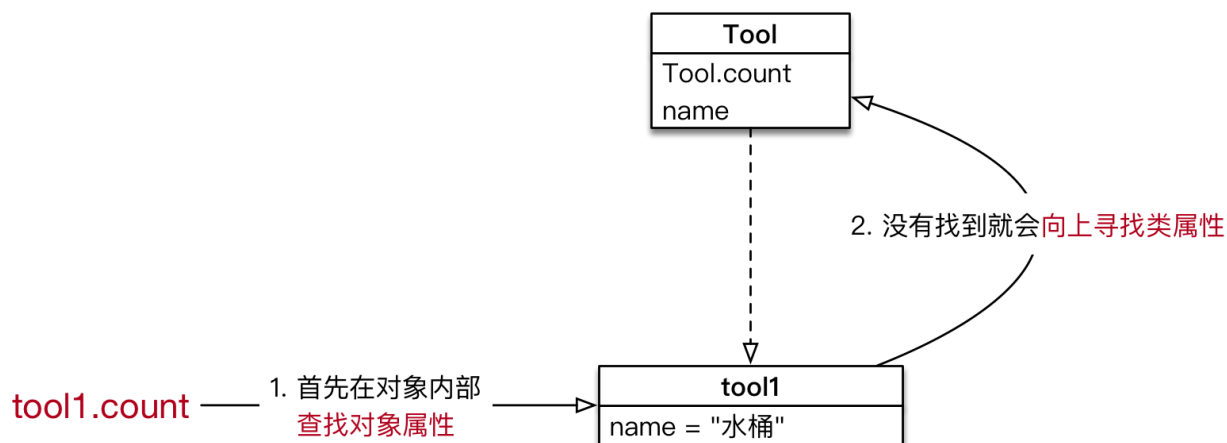
- 定义一个 **工具类**
- 每件工具都有自己的 **name**
- **需求** —— 知道使用这个类，创建了多少个工具对象？

Tool
Tool.count name
<code>__init__(self, name):</code>

```
class Tool(object):  
    # 使用赋值语句定义类属性，记录所有工具对象的数量  
    count = 0  
  
    def __init__(self, name):  
        self.name = name  
        # 让类属性的值+1  
        Tool.count += 1  
  
# 1. 创建工具对象  
tool1 = Tool('斧头')  
tool2 = Tool('榔头')  
tool3 = Tool('水桶')  
  
# 2. 输出工具对象的总数  
print(Tool.count)
```

050. 类属性-04-属性查找机制-向上查找

- 在 **Python** 中 **属性的获取** 存在一个 **向上查找机制**



- 因此，要访问类属性有两种方式：
 - 类名.类属性
 - 对象名.类属性 (不推荐)

051. 类属性-05-使用对象名.类属性的赋值语句会创建实例属性

注意

- 如果使用 `对象.类属性 = 值` 赋值语句，只会给对象添加一个属性，而不会影响到类属性的值

```
class Tool(object):
    # 使用赋值语句定义类属性，记录所有工具对象的数量
    count = 0

    def __init__(self, name):
        self.name = name
        # 让类属性的值+1
        Tool.count += 1

# 1. 创建工具对象
tool1 = Tool('斧头')
tool2 = Tool('榔头')
tool3 = Tool('水桶')

# 2. 输出工具对象的总数
tool3.count = 99
print('工具对象总数 %d' % tool3.count)
print('====> %d' % Tool.count)
```

052. 类方法-01-基本语法

- **类属性** 就是针对 **类对象** 定义的属性
- 使用 **赋值语句** 在 `class` 关键字下方可以定义 **类属性**
- **类属性** 用于记录 **与这个类相关** 的特征
- **类方法** 就是针对 **类对象** 定义的方法
- 在 **类方法** 内部可以直接访问 **类属性** 或者调用其他的 **类方法**

语法如下

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

- 类方法需要用 **装饰器** `@classmethod` 来标识，告诉解释器这是一个类方法
- 类方法的 **第一个参数** 应该是 `cls`
- 由 **哪一个类** 调用的方法，方法内的 `cls` 就是 **哪一个类的引用**
即调用这个类方法的类引用
- 这个参数和 **实例方法** 的第一个参数 `self` 类似

提示：使用其他名称也可以，不过习惯使用 `cls`

- 通过 **类名** 调用 **类方法**，调用方法时，不需要传递 `cls` 参数
- 在方法内部
 - 可以通过 `cls.` 访问类的属性
 - 也可以通过 `cls.` 调用其他的类方法

053. 类方法-02-案例演练

示例需求

- 定义一个 **工具类**
- 每件工具都有自己的 `name`
- **需求** —— 在类封装一个 `show_tool_count` 的类方法，输出使用当前这个类，创建的对象个数

Tool
Tool.count name
<code>__init__(self, name):</code> <code>show_tool_count(cls):</code>

在类方法内部，可以直接使用 `cls` 访问类属性 或者 调用类方法

```
class Tool(object):
    # 使用赋值语句定义类属性，记录所有工具对象的数量
    count = 0

    @classmethod
    def show_tool_count(cls):
        print('工具对象的数量 %d' % cls.count)

    def __init__(self, name):
        self.name = name
        # 让类属性的值+1
        Tool.count += 1

# 创建工具对象
tool1 = Tool('斧头')
tool2 = Tool('榔头')

# 调用类方法
Tool.show_tool_count()
```

054. 静态方法-01-应用场景和定义方式

- 在开发时，如果需要在类中封装一个方法，这个方法：
 - 既 **不需要** 访问 **实例属性** 或者调用 **实例方法**
 - 也 **不需要** 访问 **类属性** 或者调用 **类方法**
- 这个时候，可以把这个方法封装成一个 **静态方法**

语法如下

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

- **静态方法** 需要用 **装饰器** `@staticmethod` 来标识，告诉解释器这是一个静态方法
- 通过 `类名.` 调用 **静态方法**

```
class Dog(object):
    @staticmethod
    def run():
        # 不访问实例属性/类属性
        print('小狗要跑...')

# 通过 类名. 调用静态方法 - 不需要创建对象
Dog.run()
```

调用静态方法，不需要创建对象，直接通过 `类名.` 就可以了

055. 方法综合-01-案例分析

需求

1. 设计一个 `Game` 类
2. 属性：
 - 定义一个 **类属性** `top_score` 记录游戏的 **历史最高分**
 - 定义一个 **实例属性** `player_name` 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
4. 主程序步骤
 - 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

Game
Game.top_score player_name
<code>__init__(self, player_name):</code> <code>show_help():</code> <code>show_top_score(cls):</code> <code>start_game(self):</code>

056. 方法综合-02-案例演练

```
class Game(object):
    # 历史最高分
    top_score = 0

    def __init__(self, player_name):
        self.player_name = player_name

    @staticmethod
    def show_help():
        print('帮助信息: 让僵尸进入大门')

    @classmethod
    def show_top_score(cls):
        print('历史记录 %d' % cls.top_score)

    def start_game(self):
        print('%s 开始游戏啦...' % self.player_name)

# 1. 查看游戏的帮助信息
Game.show_help()

# 2. 查看历史最高分
Game.show_top_score()

# 3. 创建游戏对象
game = Game('小明')
game.start_game()
```


057. 方法综合-03-确定方法类型的套路

案例小结

- **实例方法** —— 方法内部 **需要访问实例属性**
实例方法 内部可以使用 **类名.** 访问类属性
- **类方法** —— 方法内部 **只需要访问类属性**
- **静态方法** —— 方法内部，**不需要访问实例属性和类属性**

提问

如果方法内部既需要访问 **实例属性**，又需要访问 **类属性**，应该定义成什么方法？

答案

应该定义 **实例方法**

因为，类只有一个，在 **实例方法** 内部可以使用 **类名.** 访问类属性

058. 单例-01-设计模式和单例设计模式概念

设计模式

- **设计模式** 是前人工作的总结和提炼，通常，被人们广泛流传的设计模式都是针对 **某一特定问题的成熟的解决方案**
- 使用 **设计模式** 是为了可重用代码、让代码更容易被他人理解、保证代码可靠性

单例设计模式

- 目的 —— **让类创建的对象，在系统中只有唯一的一个实例**
- 每一次执行 **类名()** 返回的对象，内存地址是相同的

单例设计模式的应用场景

- **音乐播放** 对象
- **回收站** 对象
- **打印机** 对象

.....

059. 单例-02-new方法的作用

__new__ 方法

以 `__方法名__` 命名的方法是内置方法

- 使用 `类名()` 创建对象时，Python 的解释器首先会调用 `__new__` 方法为对象分配空间
- `__new__` 是一个由 `object` 基类提供的内置的静态方法，主要作用有两个：
 - 1) 在内存中为对象分配空间
 - 2) 返回对象的引用
- Python 的解释器获得对象的引用后，将引用作为第一个参数，传递给 `__init__` 方法

重写 `__new__` 方法的代码非常固定！

- 重写 `__new__` 方法一定要 `return super().__new__(cls)`
- 否则 Python 的解释器得不到分配了空间的对象引用，就不会调用对象的初始化方法
- 注意：`__new__` 是一个静态方法，在调用时需要主动传递 `cls` 参数



060. 单例-03-重写new方法

```
class MusicPlayer(object):  
    def __new__(cls, *args, **kwargs):  
        # 1. 创建对象时，new方法会被自动调用  
        print('创建对象，分配空间')  
        # 2. 为对象分配空间（直接调用父类方法）  
        instance = super().__new__(cls)  
        # 3. 返回对象的引用  
        return instance  
  
    def __init__(self):  
        print('播放器初始化')
```

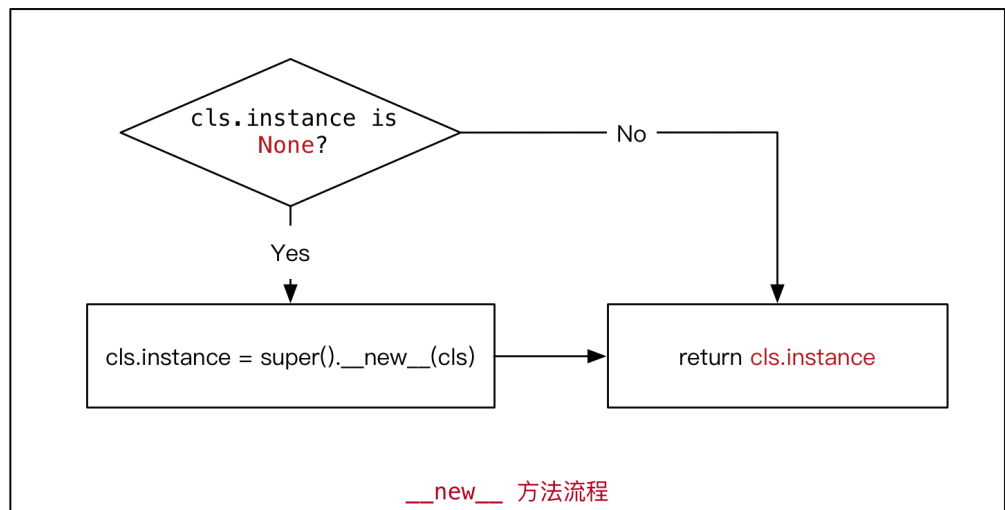
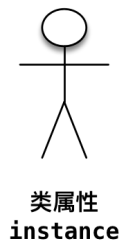
```
# 创建播放器对象  
player = MusicPlayer()
```

```
print(player)
```

061. 单例-04-单例设计模式思路分析

单例 —— 让类创建的对象，在系统中只有唯一的一个实例

1. 定义一个 类属性，初始值是 `None`，用于记录 单例对象的引用
2. 重写 `__new__` 方法
3. 如果 类属性 `is None`，调用父类方法分配空间，并在类属性中记录结果
4. 返回类属性中记录的对象引用



062. 单例-05-单例设计模式代码实现

```
class MusicPlayer(object):
    # 记录第一个被创建对象的引用
    instance = None

    def __new__(cls, *args, **kwargs):
        # 1. 判断类属性是否为空对象
        if cls.instance is None:
            # 2. 调用父类的方法，为第一个对象分配空间
            cls.instance = super().__new__(cls)
            # 3. 返回类属性保存的对象引用
            return cls.instance
```

创建多个对象

```
player1 = MusicPlayer()
print(player1)

player2 = MusicPlayer()
print(player2)
```

063. 单例-06-初始化动作只执行一次

只执行一次初始化工作

- 在每次使用 `类名()` 创建对象时，`Python` 的解释器都会自动调用两个方法：
 - `__new__` 分配空间
 - `__init__` 对象初始化
- 在上一小节对 `__new__` 方法改造之后，每次都会得到 **第一次被创建对象的引用**
- 但是：**初始化方法还会被再次调用**

需求

- 让 **初始化动作** 只被 **执行一次**

解决办法

1. 定义一个类属性 `init_flag` 标记是否 **执行过初始化动作**，初始值为 `False`
2. 在 `__init__` 方法中，判断 `init_flag`，如果为 `False` 就执行初始化动作
3. 然后将 `init_flag` 设置为 `True`
4. 这样，再次 **自动** 调用 `__init__` 方法时，**初始化动作就不会被再次执行** 了

```
class MusicPlayer(object):
    # 记录第一个被创建对象的引用
    instance = None
    # 记录是否执行过初始化动作
    init_flag = False

    def __new__(cls, *args, **kwargs):
        # 1. 判断类属性是否为空对象
        if cls.instance is None:
            # 2. 调用父类的方法，为第一个对象分配空间
            cls.instance = super().__new__(cls)
            # 3. 返回类属性保存的对象引用
            return cls.instance

    def __init__(self):
        # 1. 判断是否执行过初始化动作
        if MusicPlayer.init_flag:
```

```
        return

# 2. 如果没有执行过，再执行初始化动作
print('初始化播放器')

# 3. 修改类属性的标记
MusicPlayer.init_flag = True

# 创建多个对象
player1 = MusicPlayer()
print(player1)

player2 = MusicPlayer()
print(player2)
```

完成于 201810081430