Table of Contents

软件测试**Python**课程

本阶段课程不仅可以帮助我们进入Python语言世界，同时也是后续UI自动化测试、接口自动化测试等课程阶段的语言基础。

> Life is short, you need Python! -- 人生苦短，我用Python！

课程大纲

序号	章节	知识点
1	Python基础	1. 认识Python 2. Python环境搭建 3. PyCharm 4. 注释、变量、变量类型、输入输出、运算符
2	流程控制结构	1. 判断语句 2. 循环
3	数据序列	1. 字符串 2. 列表 3. 元组 4. 字典
4	函数	1. 函数基础 2. 变量进阶 3. 函数进阶 4. 匿名函数
5	面向对象	1. 面向对象编程介绍 2. 类和对象 3. 面向对象基础语法 4. 封装、继承、多态 5. 类属性和类方法
6	异常、模块、文件操作	1. 异常 2. 模块和包 3. 文件操作
7	UnitTest框架	1. UnitTest基本使用 2. UnitTest断言 3. 参数化 4. 生成HTML测试报告

课程目标

1. 掌握如何搭建Python开发环境；
2. 掌握Python基础语法, 具备基础的编程能力；
3. 建立编程思维以及面向对象程序设计思想；
4. 掌握如何通过UnitTest编写测试脚本，并生成HTML测试报告。

面向对象

目标

1. Python类的定义及使用
2. Python对象和类的区别
3. 类中的初始化方法
4. 成员变量和成员方法的属性
5. Python中继承的含义、作用
6. Python中继承的实现
7. Python中多态的含义及实现
8. Python类属性及类方法

面向对象编程介绍

目标

1. 了解面向对象基本概念

1. 面向对象基本概念

> 面向对象编程 —— `Object Oriented Programming` 简写 `OOP`

- 我们之前学习的编程方式就是 面向过程 的
- 面相过程 和 面相对象，是两种不同的 编程方式
- 对比 面向过程 的特点，可以更好地了解什么是 面向对象

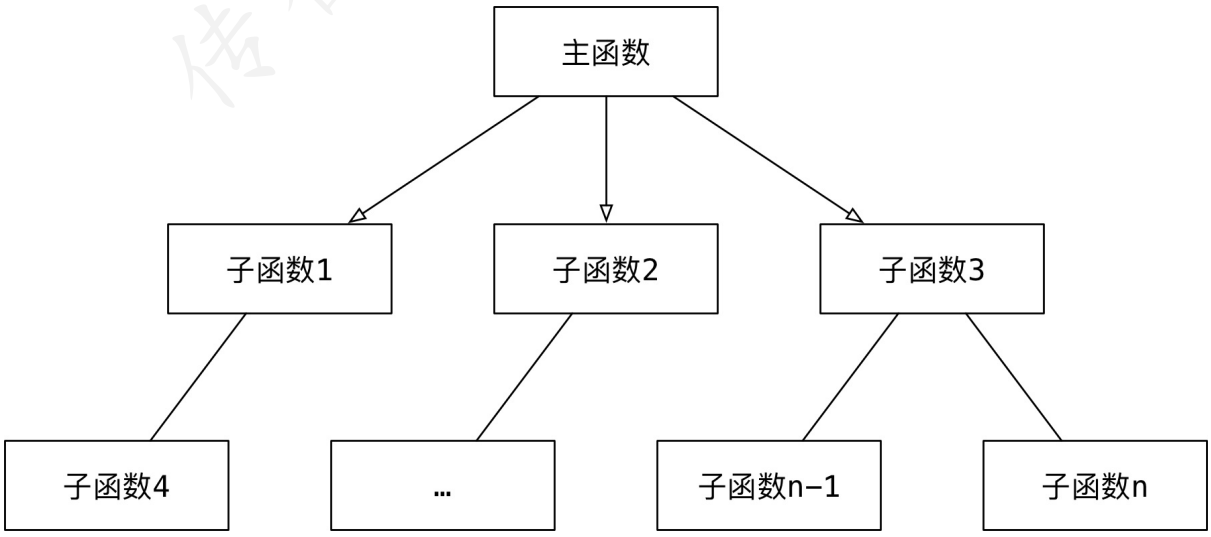
1.1 过程和函数（科普）

- 过程 是早期的一个编程概念
- 过程 类似于函数，只能执行，但是没有返回值
- 函数 不仅能执行，还可以返回结果

1.2 面相过程和面相对象基本概念

1) 面相过程 —— 怎么做？

1. 把完成某一个需求的 `所有步骤` `从头到尾` 逐步实现
2. 根据开发需求，将某些 功能独立 的代码 封装 成一个又一个 函数
3. 最后完成的代码，就是顺序地调用 不同的函数
4. 开发复杂项目，没有固定的套路，开发难度很大！



2) 面向对象 —— 谁来做？

> 相比较函数，面向对象 是 更大 的 封装，根据 职责 在 一个对象中 封装 多个方法

1. 在完成某一个需求前，首先确定 职责 —— 要做的事情（方法）
2. 根据 职责 确定不同的 对象，在 对象 内部封装不同的 方法（多个）
3. 最后完成的代码，就是顺序地让 不同的对象 调用 不同的方法
4. 更加适合应对复杂的需求变化，是专门应对复杂项目开发，提供的固定套路

向日葵
生命值
生产阳光() 摇晃()

豌豆射手
生命值
发射子弹()

冰冻射手
生命值
发射冰冻子弹()

普通僵尸
生命值
咬() 移动()

铁桶僵尸
生命值 铁桶
咬() 移动()

跳跃僵尸
生命值 竹竿
咬() 跳() 移动()

类和对象

目标

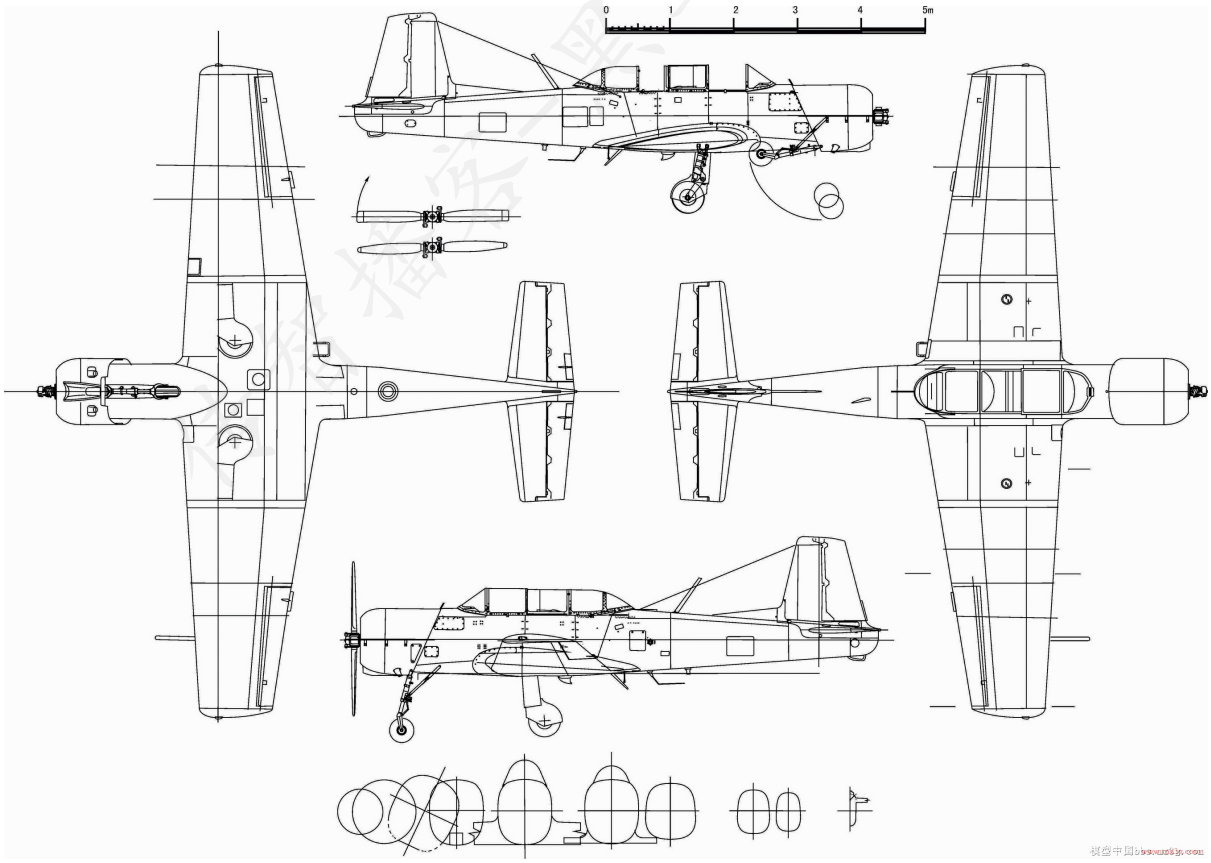
1. 掌握类和对象的概念
2. 掌握类和对象的关系
3. 掌握类的设计

1. 类和对象的概念

类 和 对象 是面向对象编程的两个核心概念

1.1 类

- 类 是对一群具有 相同 特征 或者 行为 的事物的一个统称，是抽象的，不能直接使用
  - 特征 被称为 属性
  - 行为 被称为 方法
- 类 就相当于制造飞机时的图纸，是一个 模板，是 负责创建对象的



1.2 对象

- 对象 是 由类创建出来的一个具体存在，可以直接使用

- 由 哪一个类 创建出来的 对象，就拥有在 哪一个类 中定义的：
  - 属性
  - 方法
- 对象 就相当于用 图纸 制造 的飞机

> 在程序开发中，应该 先有类，再有对象

2. 类和对象的关系

- 类是模板，对象 是根据 类 这个模板创建出来的，应该 先有类，再有对象
- 类 只有一个，而 对象 可以有很多个
  - 不同的对象 之间 属性 可能会各不相同
- 类 中定义了什么 属性和方法，对象 中就有什么属性和方法，不可能多，也不可能少

3. 类的设计

在使用面相对象开发前，应该首先分析需求，确定一下，程序中需要包含哪些类！

向日葵
生命值
生产阳光() 摇晃()

豌豆射手
生命值
发射子弹()

冰冻射手
生命值
发射冰冻子弹()

普通僵尸
生命值
咬() 移动()

铁桶僵尸
生命值 铁桶
咬() 移动()

跳跃僵尸
生命值 竹竿
咬() 跳() 移动()

在程序开发中，要设计一个类，通常需要满足一下三个要素：

1. 类名 这类事物的名字，满足大驼峰命名法
2. 属性 这类事物具有什么样的特征
3. 方法 这类事物具有什么样的行为

> 大驼峰命名法：`CapWords`
>
> 1. 每一个单词的首字母大写
> 2. 单词与单词之间没有下划线

3.1 类名的确定

名词提炼法 分析 整个业务流程，出现的 名词，通常就是找到的类

3.2 属性和方法的确定

- 对 对象的特征描述，通常可以定义成 属性
- 对象具有的行为（动词），通常可以定义成 方法

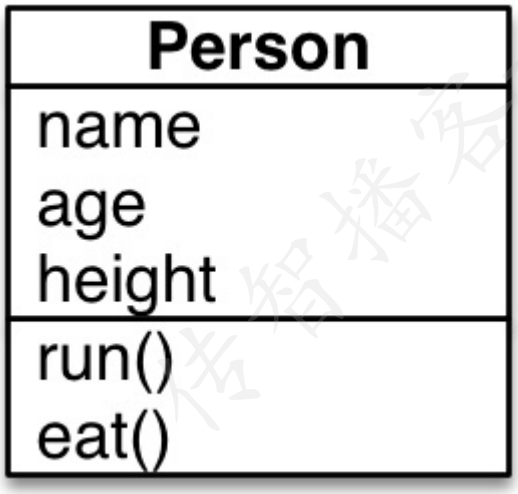
> 提示：需求中没有涉及的属性或者方法在设计类时，不需要考虑

4. 案例演练

4.1 练习1

需求：

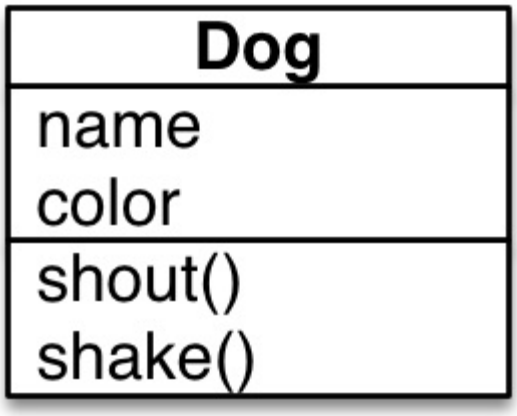
- 小明 今年 **18** 岁，身高 **1.75**，每天早上 跑 完步，会去 吃 东西
- 小美 今年 **17** 岁，身高 **1.65**，小美不跑步，小美喜欢 吃 东西



4.2 练习2

需求：

- 一只 黄颜色 的 狗狗 叫 大黄
- 看见生人 汪汪叫
- 看见家人 摇尾巴



面相对象基础语法

目标

1. 知道dir内置函数
2. 掌握定义简单的类（只包含方法）
3. 理解方法中的self参数
4. 掌握初始化方法的定义和使用
5. 了解内置方法和属性

1. dir内置函数（知道）

在Python中 对象几乎是无所不在的，我们之前学习的 变量、数据、函数 都是对象

在Python中可以使用以下两个方法验证：

1. 在 标识符 / 数据 后输入一个 `.`，在PyCharm中会提示该对象能够调用的 方法列表
2. 使用内置函数 `dir` 传入 标识符 / 数据，可以查看对象内的 所有属性及方法

提示：`__方法名__` 格式的方法是 `Python` 提供的 内置方法 / 属性，稍后会给大家介绍一些常用的 内置方法 / 属性

序号	方法名	类型	作用
1	`__init__`	方法	对象被初始化时，会被 自动 调用
2	`__del__`	方法	对象被从内存中销毁前，会被 自动 调用
3	`__str__`	方法	返回对象的描述信息，`print` 函数输出使用

> 提示: 利用好 `dir()` 函数，在学习时很多内容就不需要死记硬背了

2. 定义简单的类（只包含方法）

面向对象 是 更大 的 封装，在 一个类中 封装 多个方法，这样 通过这个类创建出来的对象，就可以直接调用这些方法了！

2.1 定义只包含方法的类

在 `Python` 中要定义一个只包含方法的类，语法格式如下：

```
class 类名:

    def 方法1(self, 参数列表):
        pass

    def 方法2(self, 参数列表):
        pass
```

- 方法 的定义格式和之前学习过的函数 几乎一样
- 区别在于第一个参数必须是 `self`，大家暂时先记住，稍后介绍 `self`

> 注意：类名 的 命名规则 要符合 大驼峰命名法

2.2 创建对象

- 当一个类定义完成之后，要使用这个类来创建对象，语法格式如下：

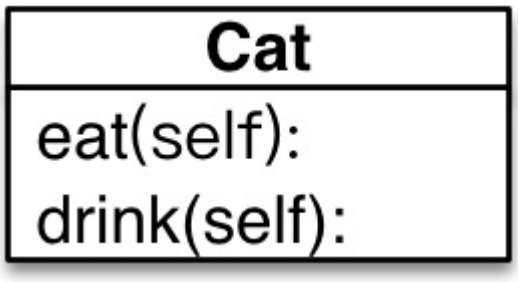
```
对象变量 = 类名()
```

2.3 第一个面向对象程序

需求：小猫 爱 吃 鱼，小猫 要 喝 水

分析：

1. 定义一个猫类 `Cat`
2. 定义两个方法 `eat` 和 `drink`
3. 按照需求 —— 不需要定义属性




```
class Cat:
    """这是一个猫类"""

    def eat(self):
        print("小猫爱吃鱼")

    def drink(self):
        print("小猫在喝水")

tom = Cat()
tom.drink()
tom.eat()
```

引用概念的强调

> 在面向对象开发中，引用的概念是同样适用的！

- 在 `Python` 中使用类 创建对象之后， `tom` 变量中 仍然记录的是 对象在内存中的地址
- 也就是 `tom` 变量 引用 了 新建的猫对象
- 使用 `print` 输出 对象变量，默认情况下，是能够输出这个变量 引用的对象 是 由哪一个类创建的对象，以及在内存中的地址（十六进制表示）

> 提示：在计算机中，通常使用 十六进制 表示 内存地址
>
> - 十进制 和 十六进制 都是用来表达数字的，只是表示的方式不一样
> - 十进制 和 十六进制 的数字之间可以来回转换

- `%d` 可以以 **10** 进制 输出数字

- `%x` 可以以 **16** 进制 输出数字

案例进阶 —— 使用 **Cat** 类再创建一个对象

```
lazy_cat = Cat()
lazy_cat.eat()
lazy_cat.drink()
```

> 提问：`tom` 和 `lazy_cat` 是同一个对象吗？

3. 方法中的 `self` 参数

3.1 案例改造 —— 给对象增加属性

- 在 `Python` 中，要 给对象设置属性，非常的容易，但是不推荐使用
  - 因为：对象属性的封装应该封装在类的内部
- 只需要在 类的外部的代码 中直接通过 `.` 设置一个属性即可

> 注意：这种方式虽然简单，但是不推荐使用！

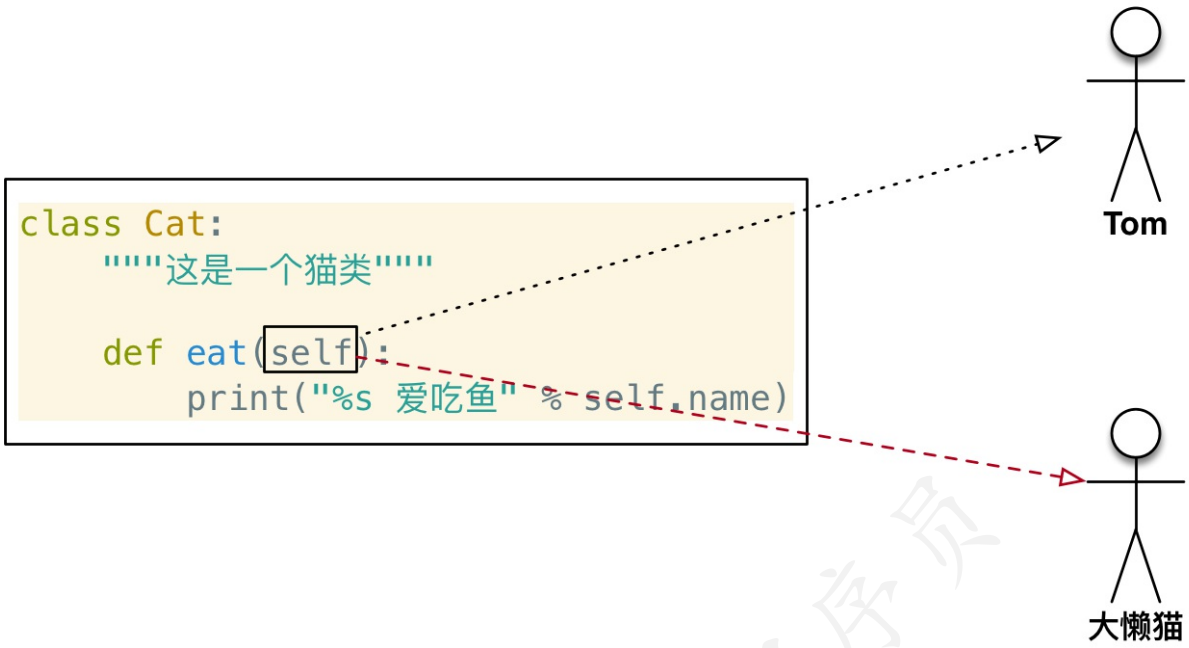
```
tom.name = "Tom"
...

lazy_cat.name = "大懒猫"
```

3.2 使用 `self` 在方法内部输出每一只猫的名字

> 由 哪一个对象 调用的方法，方法内的 `self` 就是 哪一个对象的引用

- 在类封装的方法内部，`self` 就表示 当前调用方法的对象自己
- 调用方法时，程序员不需要传递 `self` 参数
- 在方法内部
  - 可以通过 `self.` 访问对象的属性
  - 也可以通过 `self.` 调用其他的对象方法
- 改造代码如下：

```
class Cat:

    def eat(self):
        print("%s 爱吃鱼" % self.name)

tom = Cat()
tom.name = "Tom"
tom.eat()

lazy_cat = Cat()
lazy_cat.name = "大懒猫"
lazy_cat.eat()
```



- 在 类的外部，通过 `变量名.` 访问对象的 属性和方法
- 在 类封装的方法中，通过 `self.` 访问对象的 属性和方法

4. 初始化方法

4.1 之前代码存在的问题 —— 在类的外部给对象增加属性

- 将案例代码进行调整，先调用方法 再设置属性，观察一下执行效果

```
tom = Cat()
tom.drink()
tom.eat()
tom.name = "Tom"
print(tom)
```

- 程序执行报错如下：

```
AttributeError: 'Cat' object has no attribute 'name'
属性错误：'Cat' 对象没有 'name' 属性
```

提示

- 在日常开发中，不推荐在 类的外部 给对象增加属性
  - 如果在运行时，没有找到属性，程序会报错
- 对象应该包含有哪些属性，应该 封装在类的内部

4.2 初始化方法

- 当使用 `类名()` 创建对象时，会 自动 执行以下操作：
  1. 为对象在内存中 分配空间 —— 创建对象
  2. 为对象的属性 设置初始值 —— 初始化方法(`init`)
- 这个 初始化方法 就是 `__init__` 方法， `__init__` 是对象的内置方法

> `__init__` 方法是 专门 用来定义一个类 具有哪些属性的方法！

在 `Cat` 中增加 `__init__` 方法，验证该方法在创建对象时会被自动调用

```
class Cat:
    """这是一个猫类"""

    def __init__(self):
        print("初始化方法")
```

4.3 在初始化方法内部定义属性

- 在 `__init__` 方法内部使用 `self.属性名 = 属性的初始值` 就可以 定义属性
- 定义属性之后，再使用 `Cat` 类创建的对象，都会拥有该属性

```
class Cat:

    def __init__(self):

        print("这是一个初始化方法")

        # 定义用 Cat 类创建的猫对象都有一个 name 的属性
        self.name = "Tom"

    def eat(self):
        print("%s 爱吃鱼" % self.name)

# 使用类名()创建对象的时候，会自动调用初始化方法 __init__
tom = Cat()

tom.eat()
```

4.4 改造初始化方法 —— 初始化的同时设置初始值

- 在开发中，如果希望在 创建对象的同时，就设置对象的属性，可以对 `__init__` 方法进行 改造
  1. 把希望设置的属性值，定义成 `__init__` 方法的参数
  2. 在方法内部使用 `self.属性 = 形参` 接收外部传递的参数
  3. 在创建对象时，使用 `类名(属性1, 属性2...)` 调用

```
class Cat:

    def __init__(self, name):
        print("初始化方法 %s" % name)
        self.name = name
    ...

tom = Cat("Tom")
...

lazy_cat = Cat("大懒猫")
...
```

5. 内置方法和属性

序号	方法名	类型	作用

1	`__del__`	方法	对象被从内存中销毁前，会被 自动 调用
2	`__str__`	方法	返回对象的描述信息， `print` 函数输出使用

5.1 `__del__` 方法（知道）

- 在 `Python` 中
  - 当使用 `类名()` 创建对象时，为对象 分配完空间后，自动 调用 `__init__` 方法
  - 当一个 对象被从内存中销毁 前，会 自动 调用 `__del__` 方法
- 应用场景
  - `__init__` 改造初始化方法，可以让创建对象更加灵活
  - `__del__` 如果希望在对象被销毁前，再做一些事情，可以考虑一下 `__del__` 方法
- 生命周期
  - 一个对象从调用 `类名()` 创建，生命周期开始
  - 一个对象的 `__del__` 方法一旦被调用，生命周期结束
  - 在对象的生命周期内，可以访问对象属性，或者让对象调用方法

```
class Cat:

    def __init__(self, new_name):

        self.name = new_name

        print("%s 来了" % self.name)

    def __del__(self):

        print("%s 去了" % self.name)

# tom 是一个全局变量
tom = Cat("Tom")
print(tom.name)

# del 关键字可以删除一个对象
del tom

print("-" * 50)
```

5.2 `__str__` 方法

- 在 `Python` 中，使用 `print` 输出 对象变量，默认情况下，会输出这个变量 引用的对象 是 由哪一个类创建的对象，以及 在内存中的地址（十六进制表示）
- 如果在开发中，希望使用 `print` 输出 对象变量 时，能够打印 自定义的内容，就可以利用 `__str__` 这个内置方法了

> 注意： `__str__` 方法必须返回一个字符串

```
class Cat:

    def __init__(self, new_name):

        self.name = new_name

        print("%s 来了" % self.name)
```

```
    def __del__(self):

        print("%s 去了" % self.name)

    def __str__(self):
        return "我是小猫: %s" % self.name

tom = Cat("Tom")
print(tom)
```

面向对象封装案例

目标

1. 理解封装的概念
2. 掌握面向对象封装案例-小明爱跑步
3. 掌握面向对象封装案例-存放家具
4. 了解身份运算符的用法

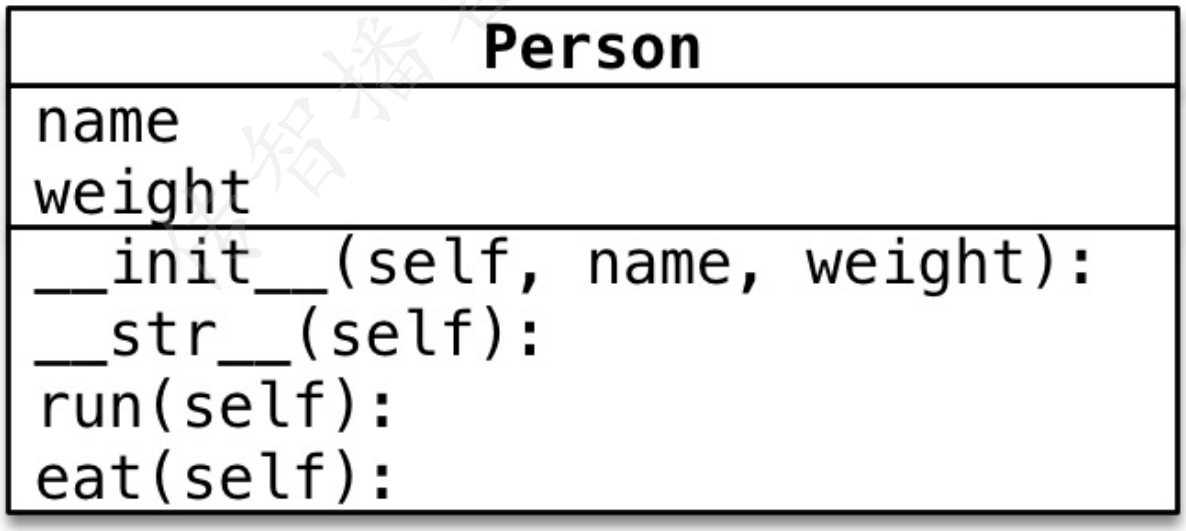
1. 封装

- 封装 是面向对象编程的一大特点
- 面向对象编程的 第一步 —— 将 属性 和 方法 封装 到一个抽象的 类 中
- 外界 使用 类 创建 对象，然后 让对象调用方法
- 对象方法的细节 都被 封装 在 类的内部

2. 封装案例一：小明爱跑步

需求：

1. 小明 体重 `75.0` 公斤
2. 小明每次 跑步 会减肥 `0.5` 公斤
3. 小明每次 吃东西 体重增加 `1` 公斤



> 提示：在 对象的方法内部，是可以 直接访问对象的属性 的！

代码实现：

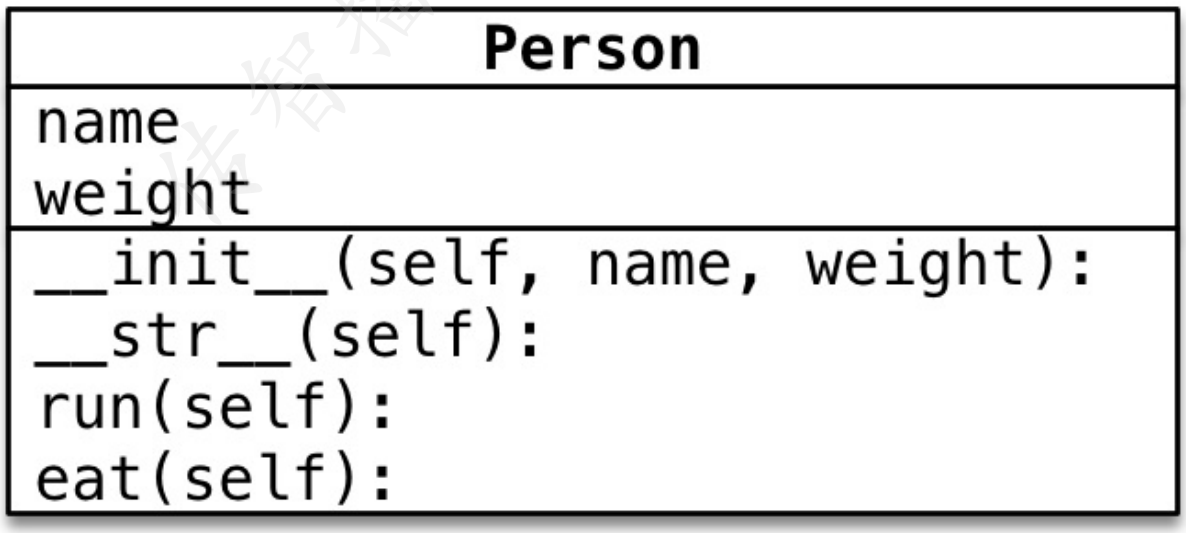
```
class Person:
    """人类"""

    def __init__(self, name, weight):
        self.name = name
```

```
        self.weight = weight

    def __str__(self):
        return "我的名字叫 %s 体重 %.2f 公斤" % (self.name, self.weight)

    def run(self):
        """跑步"""

        print("%s 爱跑步，跑步锻炼身体" % self.name)
        self.weight -= 0.5

    def eat(self):
        """吃东西"""

        print("%s 是吃货，吃完这顿再减肥" % self.name)
        self.weight += 1


xiaoming = Person("小明", 75)

xiaoming.run()
xiaoming.eat()
xiaoming.eat()

print(xiaoming)
```

2.1 小明爱跑步扩展--小美也爱跑步

需求：

1. 小明 和 小美 都爱跑步
2. 小明 体重 `75.0` 公斤
3. 小美 体重 `45.0` 公斤
4. 每次 跑步 都会减少 `0.5` 公斤
5. 每次 吃东西 都会增加 `1` 公斤



提示

1. 在 对象的方法内部，是可以 直接访问对象的属性 的
2. 同一个类 创建的 多个对象 之间，属性 互不干扰！

3. 封装案例二：摆放家具

需求：

1. 房子**(House)** 有 户型、总面积 和 家具名称列表
   - 新房子没有任何的家具
2. 家具**(HouseItem)** 有 名字 和 占地面积，其中
   - 席梦思**(bed)** 占地 `4` 平米
   - 衣柜**(chest)** 占地 `2` 平米
   - 餐桌**(table)** 占地 `1.5` 平米
3. 将以上三件 家具 添加 到 房子 中
4. 打印房子时，要求输出：户型、总面积、剩余面积、家具名称列表

HouseItem
name area
__init__(self, name, area): __str__(self):

House
house_type area free_area item_list
__init__(self, house_type, area): __str__(self): add_item(self, item):

剩余面积

1. 在创建房子对象时，定义一个 剩余面积的属性，初始值和总面积相等
2. 当调用 `add_item` 方法，向房间 添加家具 时，让 剩余面积 -= 家具面积

思考：应该先开发哪一个类？

答案 —— 家具类

1. 家具简单
2. 房子要使用到家具，被使用的类，通常应该先开发

3.1 创建家具

```
class HouseItem:

    def __init__(self, name, area):
        """

        :param name: 家具名称
        :param area: 占地面积
        """
        self.name = name
        self.area = area

    def __str__(self):
        return "[%s] 占地面积 %.2f" % (self.name, self.area)


# 1. 创建家具
bed = HouseItem("席梦思", 4)
chest = HouseItem("衣柜", 2)
table = HouseItem("餐桌", 1.5)

print(bed)
print(chest)
```

```
print(table)
```

小结

1. 创建了一个 家具类，使用到 `__init__` 和 `__str__` 两个内置方法
2. 使用 家具类 创建了 三个家具对象，并且 输出家具信息

3.2 创建房间

```
class House:

    def __init__(self, house_type, area):
        """

        :param house_type: 户型
        :param area: 总面积
        """
        self.house_type = house_type
        self.area = area

        # 剩余面积默认和总面积一致
        self.free_area = area
        # 默认没有任何的家具
        self.item_list = []

    def __str__(self):

        # Python 能够自动的将一对括号内部的代码连接在一起
        return ("户型：%s\n总面积：%.2f[剩余：%.2f]\n家具：%s"
                % (self.house_type, self.area,
                   self.free_area, self.item_list))

    def add_item(self, item):

        print("要添加 %s" % item)

...

# 2. 创建房子对象
my_home = House("两室一厅", 60)

my_home.add_item(bed)
my_home.add_item(chest)
my_home.add_item(table)

print(my_home)
```

小结

1. 创建了一个 房子类，使用到 `__init__` 和 `__str__` 两个内置方法
2. 准备了一个 `add_item` 方法 准备添加家具
3. 使用 房子类 创建了 一个房子对象
4. 让 房子对象 调用了三次 `add_item` 方法，将 三件家具 以实参传递到 `add_item` 内部

3.3 添加家具

需求：

1. 判断 家具的面积 是否 超过剩余面积，如果超过，提示不能添加这件家具

2. 将 家具的名称 追加到 家具名称列表 中
3. 用 房子的剩余面积 - 家具面积

```
def add_item(self, item):

    print("要添加 %s" % item)
    # 1. 判断家具面积是否大于剩余面积
    if item.area > self.free_area:
        print("%s 的面积太大，不能添加到房子中" % item.name)

        return

    # 2. 将家具的名称追加到名称列表中
    self.item_list.append(item.name)

    # 3. 计算剩余面积
    self.free_area -= item.area
```

3.4 小结

- 主程序只负责创建 房子 对象和 家具 对象
- 让 房子 对象调用 `add_item` 方法 将家具添加到房子中
- 面积计算、剩余面积、家具列表 等处理都被 封装 到 房子类的内部

4. 身份运算符

身份运算符用于 比较 两个对象的 内存地址 是否一致 —— 是否是对同一个对象的引用

运算符	描述	实例
is	is 是判断两个标识符是不是引用同一个对象	x is y，类似 id(x) == id(y)
is not	is not 是判断两个标识符是不是引用不同对象	x is not y，类似 id(a) != id(b)

> 在Python中针对 `None` 比较时，建议使用 `is` 判断

4.1 is 与 == 区别

- `is` 用于判断 两个变量 引用对象是否为同一个
- `==` 用于判断 引用变量的值 是否相等

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> b is a
False
>>> b == a
True
```

私有属性和私有方法

目标

1. 掌握私有属性和私有方法的应用场景及定义方式

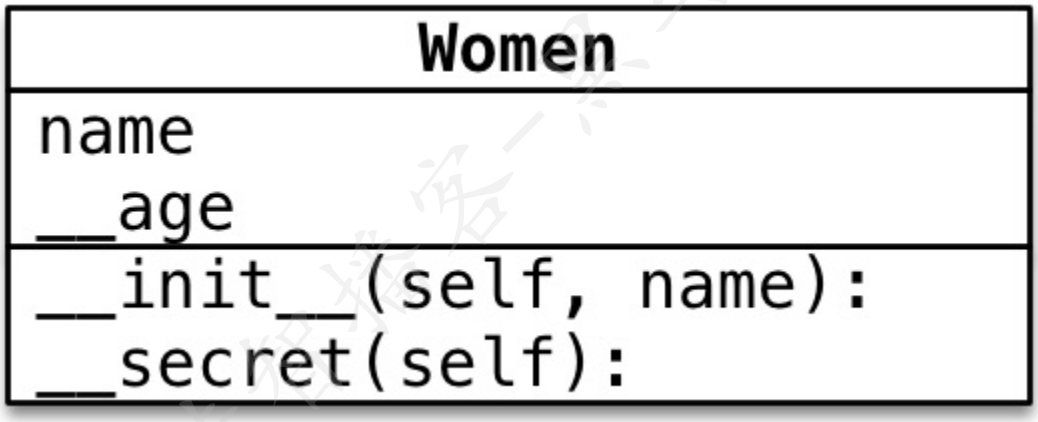
1. 私有属性和私有方法

1.1 应用场景

- 在实际开发中，对象 的 某些属性或方法 可能只希望 在对象的内部被使用，而 不希望在外部被访问到
- 私有属性 就是 对象 不希望公开的 属性
- 私有方法 就是 对象 不希望公开的 方法

1.2 定义方式

- 在 定义属性或方法时，在 属性名或者方法名前 增加 两个下划线，定义的就是 私有 属性或方法




```python
class Women:

    def __init__(self, name):

        self.name = name
        # 不要问女生的年龄
        self.__age = 18

    def __secret(self):
        print("我的年龄是 %d" % self.__age)


xiaofang = Women("小芳")
# 私有属性，外部不能直接访问
# print(xiaofang.__age)

# 私有方法，外部不能直接调用
# xiaofang.__secret()
```

2. 伪私有属性和私有方法（科普）

> 提示：在日常开发中，不要使用这种方式，访问对象的 私有属性 或 私有方法

Python中，并没有 真正意义 的 私有

- 在给 属性、方法 命名时，实际是对 名称 做了一些特殊处理，使得外界无法访问到
- 处理方式：在 名称 前面加上 `_类名` => `_类名__名称`

```
# 私有属性，外部不能直接访问到
print(xiaofang._Women__age)

# 私有方法，外部不能直接调用
xiaofang._Women__secret()
```

继承

目标

1. 掌握继承的概念和语法
2. 理解方法的重写

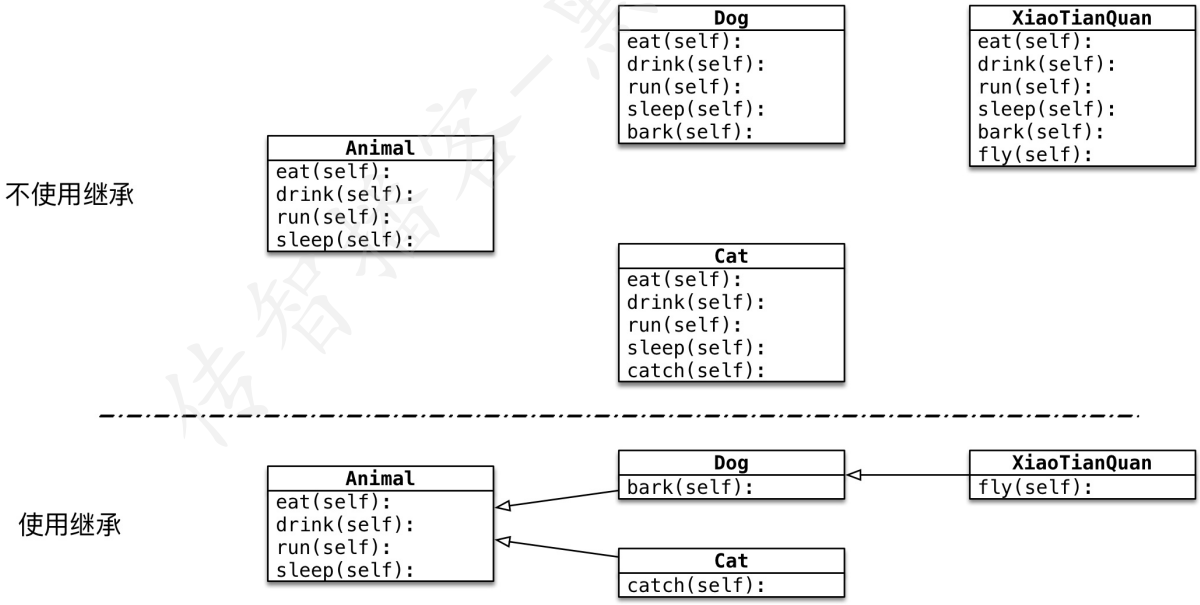
1. 面向对象三大特性

- 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，增加代码的灵活度

1. 继承

1.1 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

专业术语

- `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

1.2 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

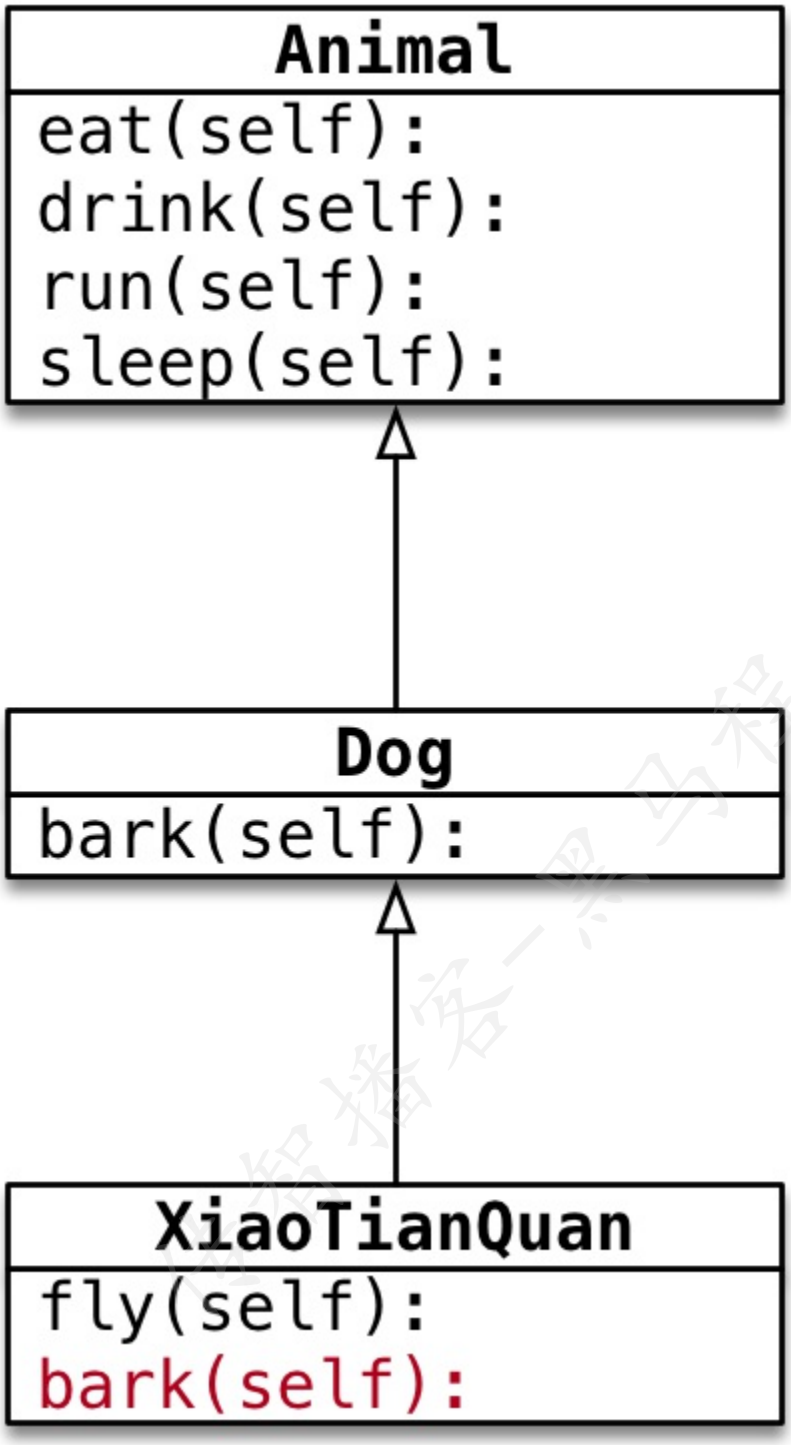
> 思考：哮天犬能够调用 `Cat` 类中定义的 `catch` 方法吗？
> 答案：不能，因为哮天犬和 `Cat` 之间没有继承关系

2. 方法的重写

- 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：

1. 覆盖 父类的方法
2. 对父类方法进行 扩展

2.1 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，就相当于在 子类中 定义了一个 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

2.2 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
  - 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

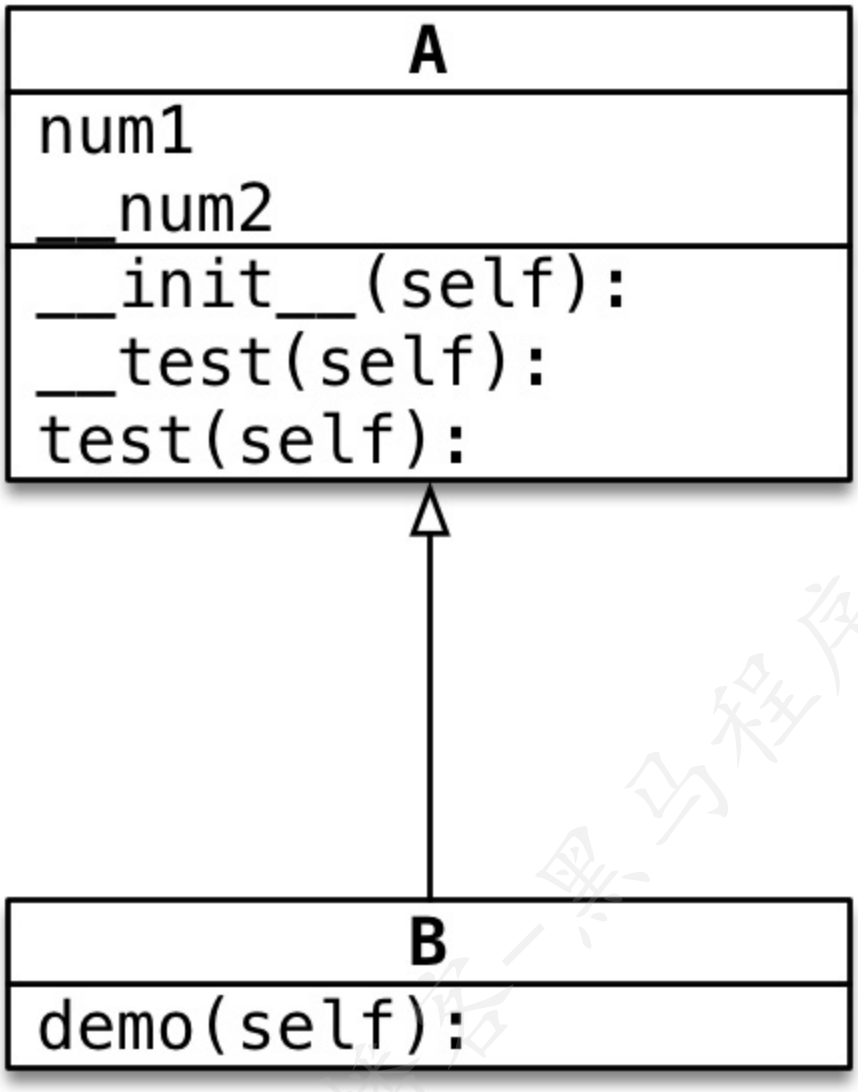
- 在 `Python` 中 `super` 是一个 特殊的类
- `super()` 就是使用 `super` 类创建出来的对象
- 最常 使用的场景就是在 重写父类方法时，调用 在父类中封装的方法实现

3. 父类的 私有属性 和 私有方法

1. 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
2. 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

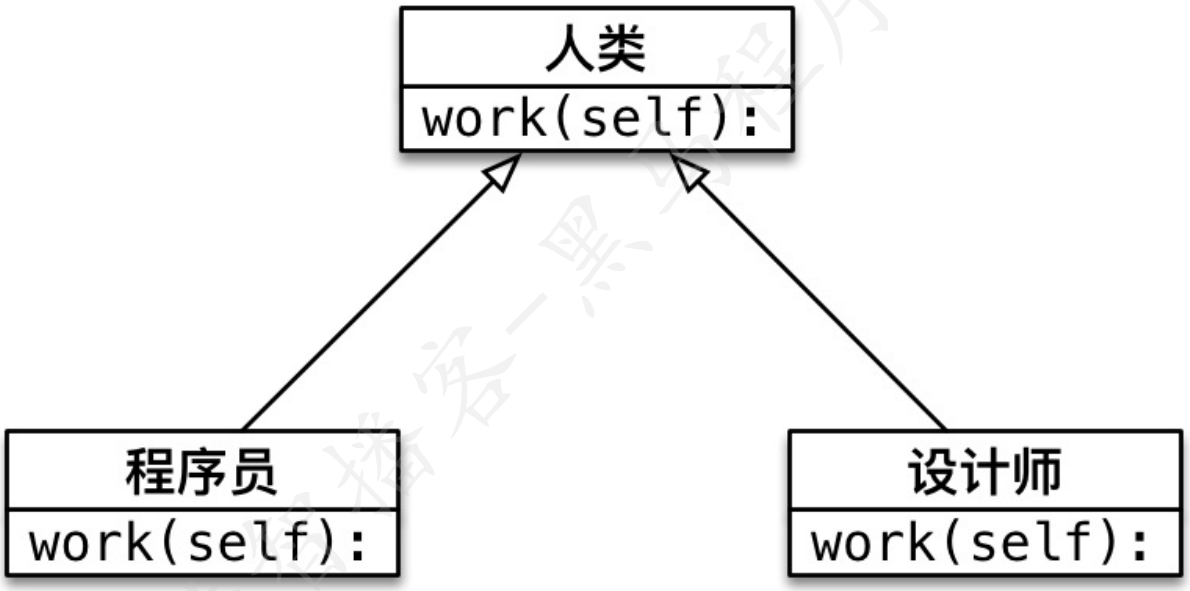
多态

目标

1. 理解多态的概念和作用

1. 多态

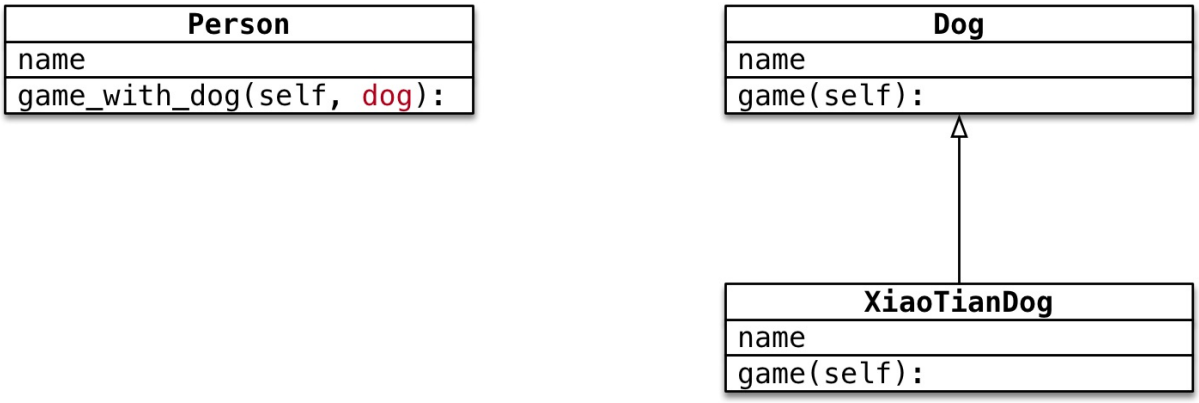
- 多态：不同的 子类对象 调用相同的 父类方法，产生不同的执行结果
- 多态 可以 增加代码的灵活度
- 以 继承 和 重写父类方法 为前提
- 是调用方法的技巧，不会影响到类的内部设计



2. 多态案例演练

需求

1. 在 `Dog` 类中封装方法 `game`
   - 普通狗只是简单的玩耍
2. 定义 `XiaoTianDog` 继承自 `Dog`，并且重写 `game` 方法
   - 哮天犬需要在天上玩耍
3. 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



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

> 多态 更容易编写出通用的代码，做出通用的编程，以适应需求的不断变化！

```
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)
```

类属性和类方法

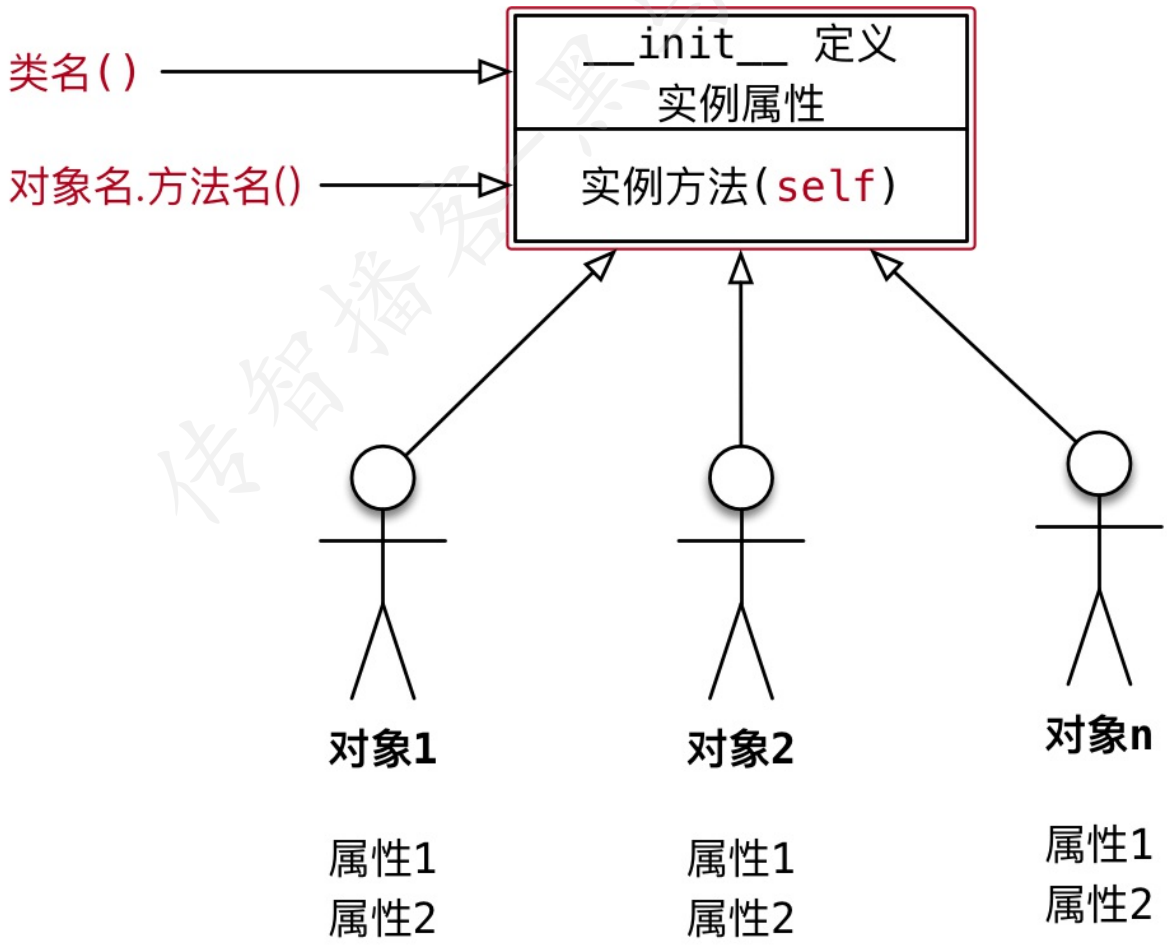
目标

1. 掌握类的结构
2. 掌握类属性和实例属性
3. 掌握类方法和静态方法

1. 类的结构

1.1 实例

1. 使用面相对象开发，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：

1. 创建出来的 对象 叫做 类 的 实例

2. 创建对象的 动作 叫做 实例化
3. 对象的属性 叫做 实例属性
4. 对象调用的方法 叫做 实例方法

在程序执行时：

1. 对象各自拥有自己的 实例属性
2. 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

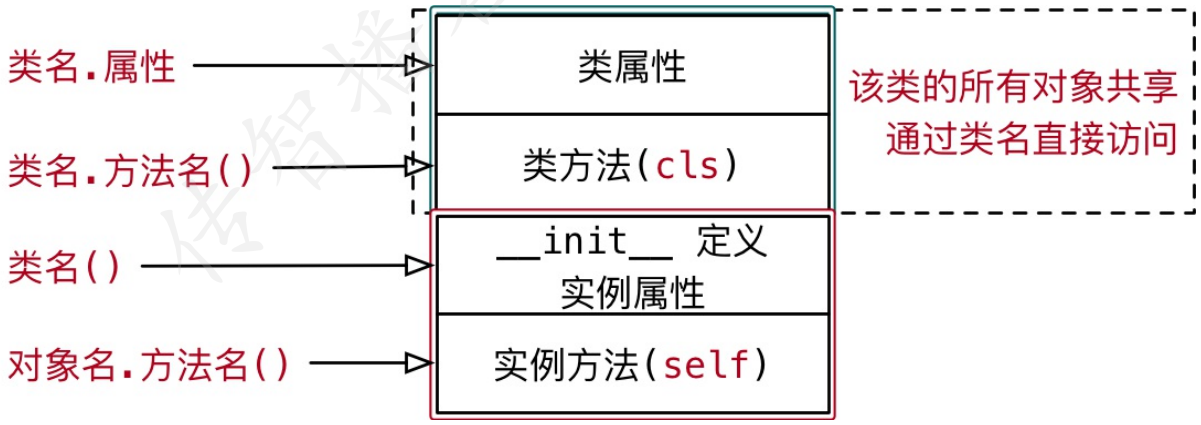
结论

- 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

1.2 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
  1. 类属性
  2. 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



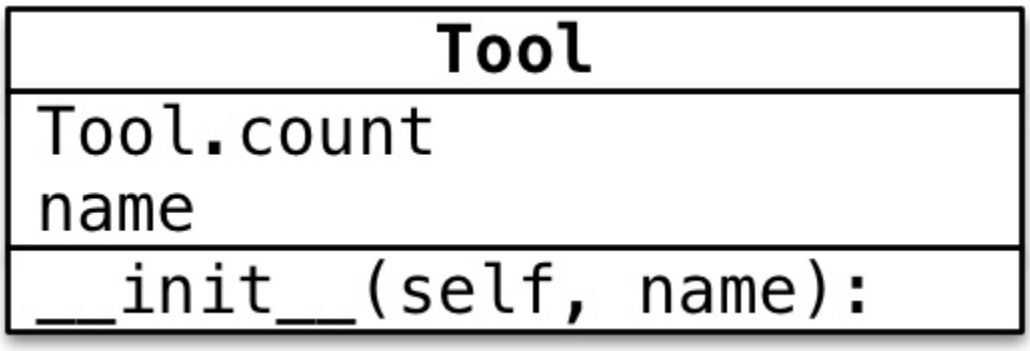
2. 类属性和实例属性

2.1 概念和使用

- 类属性 就是给 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

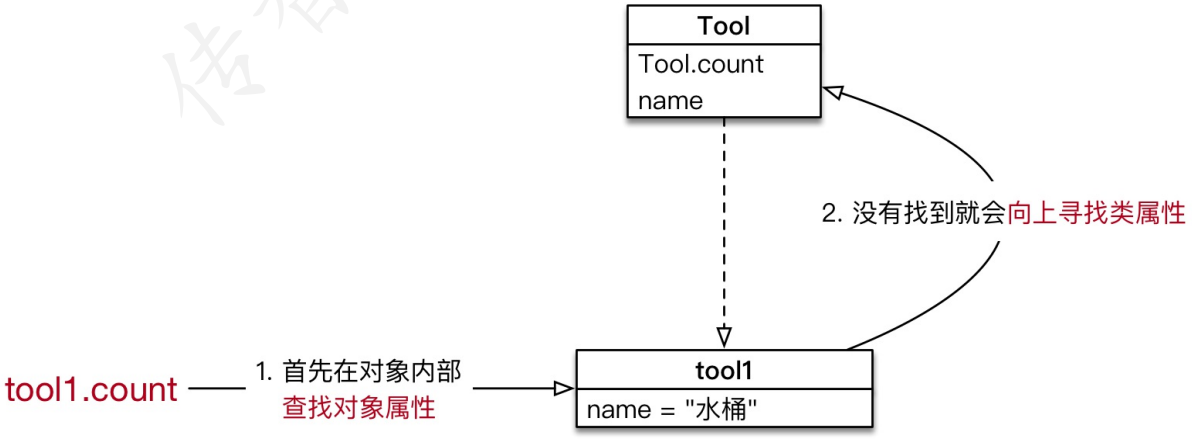
- 定义一个 工具类
- 每件工具都有自己的 `name`
- 需求 —— 知道使用这个类，创建了多少个工具对象？




```python
class Tool(object):
    # 使用赋值语句，定义类属性，记录创建工具对象的总数
    count = 0

    def __init__(self, name):
        self.name = name
        # 针对类属性做一个计数+1
        Tool.count += 1


# 创建工具对象
tool1 = Tool("斧头")
tool2 = Tool("榔头")
tool3 = Tool("铁锹")

# 知道使用 Tool 类到底创建了多少个对象?
print("现在创建了 %d 个工具" % Tool.count)
```

2.2 属性的获取机制（科普）

- 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：
  1. 类名.类属性
  2. 对象.类属性 （不推荐）

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

3. 类方法和静态方法

3.1 类方法

- 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

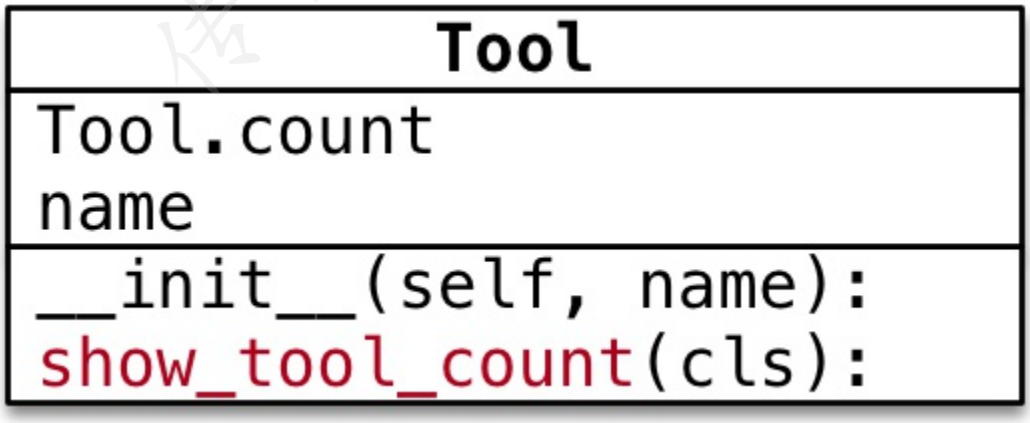
语法如下

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

- 类方法需要用 修饰器 `@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
  - 这个参数和 实例方法 的第一个参数是 `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

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




```
@classmethod
def show_tool_count(cls):
    """显示工具对象的总数"""
    print("工具对象的总数 %d" % cls.count)
```

> 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

3.2 静态方法

- 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
```

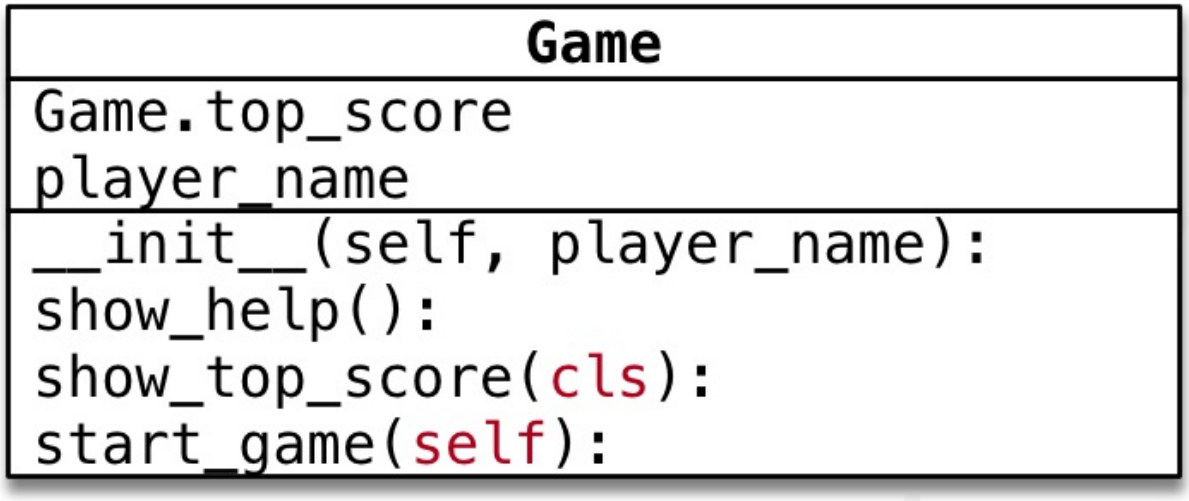
- 静态方法 需要用 修饰器 `@staticmethod` 来标识，告诉解释器这是一个静态方法
- 通过 类名.调用 静态方法

```
class Dog(object):
    # 狗类对象计数
    dog_count = 0

    @staticmethod
    def run():
        # 不需要访问实例属性也不需要访问类属性的方法
        print("狗在跑...")

    def __init__(self, name):
        self.name = name
```

3.3 方法综合案例

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



案例小结

1. 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
2. 类方法 —— 方法内部 只 需要访问 类属性
3. 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 即需要访问 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

```
class Game(object):

    # 游戏最高分，类属性
    top_score = 0

    @staticmethod
    def show_help():
        print("帮助信息：让僵尸走进房间")

    @classmethod
    def show_top_score(cls):
        print("游戏最高分是 %d" % cls.top_score)

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

    def start_game(self):
        print("[%s] 开始游戏..." % self.player_name)

        # 使用类名.修改历史最高分
        Game.top_score = 999

# 1. 查看游戏帮助
Game.show_help()

# 2. 查看游戏最高分
Game.show_top_score()
```

```
# 3. 创建游戏对象，开始游戏
game = Game("小明")

game.start_game()

# 4. 游戏结束，查看游戏最高分
Game.show_top_score()
```