面向对象基础中

1、析构方法

1.1、当一个对象被删除或者被销毁时，python 解释器也会默认调用一个方法，这个方法为__del__()方法，也称为析构方法

程序执行结束自动调用 del 方法

```
class Animal(object):
    def __init__(self,name):
        self.name = name
        print('__init__ 方法被调用')

    # 析构方法，当对象被销毁时 python 解析器会自动调用
    def __del__(self):
        print('__del__ 方法被调用')
        print('%s 对象被销毁' % self.name)
dog = Animal('旺财')
```

输出结果：程序执行完之后 python 解析器自动调用 del 方法。

```
C:\Users\Administrator\Desktop\test\venv\Scripts\python.exe C:/Users/Administrator/Desktop/test/test.py
__init__ 方法被调用
__del__ 方法被调用
旺财 对象被销毁

Process finished with exit code 0
```

对象被删除时也会自动调用 del 方法，利用 del 手动删除 dog 对象，用 input 函数让程序等待。执行结果 **del** 方法也是会被调用。

```python
class Animal(object):
    def __init__(self,name):
        self.name = name
        print('__init__ 方法被调用')

    # 析构方法，当对象被销毁时 python 解析器会自动调用
    def __del__(self):
        print('__del__ 方法被调用')
        print('%s 对象被销毁' % self.name)
dog = Animal('旺财')

del dog

input('程序等待中......')
```

输出结果：

```
C:\Users\Administrator\Desktop\test\venv\Scripts\python.exe C:/Users/Administrator/Desktop/test/test.py
__init__ 方法被调用
__del__ 方法被调用
旺财 对象被销毁
程序等待中.....
```

1.2、析构函数一般用于资源回收，对象被销毁后利用 del 方法回收内存等资源。

2、单继承

2.1、继承

在现实生活中，继承一般指的是子女继承父辈的财产。在面向对象中同样有继承。

例如：

猫的方法：喵喵叫、吃、喝

狗的方法：汪汪叫、吃、喝

如果给猫和狗都创建一个类，那么猫和狗的所有方法都要写。

```
class 猫:
    def 喵喵叫(self):
        print('喵喵叫')
    def 吃(self):
        pass
    def 喝(self):
        pass
class 狗:
    def 旺旺叫(self):
        print('旺旺')
    def 吃(self):
        pass
    def 喝(self):
        pass
```

上面的代码中我们可以发现，吃，喝方法时一样的，但是写了两遍。

如果用继承的思想，我们可以这样：

动物有吃，喝的方法，猫和狗都是动物。

```
class 动物(object):
    def 吃(self):
        pass
    def 喝(self):
        pass


class 猫(动物):  # 继承动物类
    def 喵喵叫(self):
        print('喵喵叫')


class 狗(动物):  # 继承动物类
    def 旺旺叫(self):
        print('旺旺')
```

所以，对于面向对象的继承来说，其实就是将多个类共有的方法提取到父类中，子类仅需继承父类而不必一一实现每个方法

```
class Animal(object):
    def eat(self):
        print('正在吃饭')
class Cat(Animal): # Cat 类继承 Animal 类,Cat 是子类，也称派生类，Animal 类是父类，
也称为基类
    pass
class Dog(Animal):
    pass

cat  = Cat()
cat.eat()
dog = Dog()
dog.eat()

#输出两次正在吃饭
```

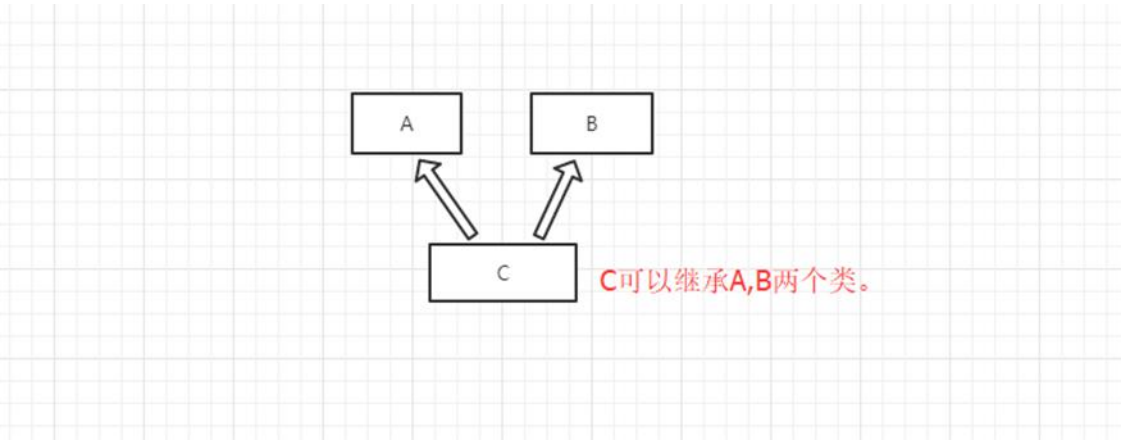
上图中子类没有写任何方法，但是直接调用 eat 方法没有报错，这说明子类继承了父类的方法。

总结：

子类在继承的时候，在定义类时，小括号()中为父类的名字 父类的属性、方法，会被继承给子类。

3、多继承

3.1、继承可以继承一个父类，那是否可以继承两个父类或多个呢？答案是可以的，这就是多继承。



C 可以将 A,B 中的方法继承过来，C 拥有 A，B 的方法属性。

```python
class A(object):
    def a(self):
        print('A 类的方法 a 输出')

class B(object):
    def b(self):
        print('B 类的方法 b 输出')

# C 类继承 A,B
# 在小括号里有多个父类名字就是多继承
class C(A,B):
    pass
c = C()
c.a()
c.b()

# 执行输出
# A 类的方法 a 输出
# B 类的方法 b 输出
```

在 C 类中并没有写方法，只继承了 A，B 两个父类。C 可以调用两个父类的方法。

3.2、如果在上面的多继承例子中，如果父类 A 和父类 B 中，有一个同名的方法，那么通过子类去调用的时候，调用哪个？

```
class Base(object):
    def test(self):
        print('------Base test---- ')
class Base1(object):
    def test(self):
        print('------Base1 test---------')

# A 继承 Base
class A(Base):
    def test(self):
        print('----A  test---------')
# B 继承 Base1
class B(Base1):
    def test(self):
        print('----B test---------')

# 定义一个子类，继承自 A，B
class C(A,B):
    pass
c = C()
c.test()

print(C.__mro__)     # 可以查看 C 类的对象搜索方法是的先后顺序，也就是继承的顺序
```

3.3、查找方法的顺序可以用 mro 查看。上图中代码查找顺序为 C->A->Base->B>Base1，一旦找到，则寻找过程立即中断，便不会再继续找了

4、继承的传递

4.1、想一想：

在现实中遗产继承，爷爷的遗产可以被父亲继承，儿子可以继承父亲的。这样看来是不是儿子也是有继承到爷爷的遗产。在面向对象中的继承呢？子类是否能继承父类的父类的方法？

4.2、看看下面的继承关系，Son 类继承 Father 类，Father 类并没有提供 eat 方法，但是父类又继承了 Grandfather 类。Son 的对象调用 eat 方法可以正常执行，运行结果得出，Son 类也继承了

Granderfather 类的方法。这就是继承的传递性。

```
class GrandFather(object):

    def eat(self):
        print('吃饭')
# 继承 GrandFather 类
class  Father(GrandFather):S
    pass

# 继承 Father 类
class Son(Father):
    pass

lg = Son()
lg.eat()
```

4.3、父类又称为基类，子类又称为派生类，在有些资料中会将子类称之为派生类，父类称之为基类。

5、重写和调用父类方法

5.1、重写父类方法

所谓重写，就是子类中，有一个和父类相同名字的方法，在子类中的方法会覆盖掉父类中同名的方法 伪代码示例：

```
class 父类:
    def 抽烟(self):
        print('抽芙蓉王')
    def 喝酒(self):
        print('喝二锅头')

class 子类(父类):
    # 与父类的（抽烟）同名方法，这就是重写父类方法
    def 抽烟(self):
        print('抽中华')    # 儿子比他爹有出息，抽好烟
```

重写父类方法后，子类调用父类的方法时将调用的是子类的方法。

5.2、调用父类方法

如果在子类中有一个方法需要父类的功能，并且又要添加新的功能。如果直接重写父类方法，那么就要重复写很多代码。那么这就要调用父类方法

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


class Cat(Animal):
    def __init__(self,name):
        #第一种方法
        #super(Cat, self).__init__(name)
        #第二种方法
        #super().__init__(name)
        #第三种方法
        #Animal.__init__(name)
```

调用父类方法有三种方式：

```
#第一种方法
#super(Cat, self).__init__(name)
#第二种方法
#super().__init__(name)
#第三种方法
#Animal.__init__(name)
```

6、多态

6.1、所谓多态：定义时的类型和运行时的类型不一样，此时就成为多态。

Pyhon 不支持 Java 和 C#这一类强类型语言中多态的写法，但是原生多态，其 Python 崇尚“鸭子类型”。 利用 python 伪代码实现 Java 和 C#的多态

```python
class F1(object):
    def show(self):
        return 'F1.show'
class S1(F1)
    def show(self):
        return 'S1.show'
class S2(F1):
    def show(self):
        return 'S2.show'
# 由于在Java或C#中定义函数参数时，必须指定参数的类型
# 为了让Func函数既可以执行S1对象的show方法，又可以执行S2对象的show方法，
# 所以，定义了一个S1和S2类的父类
# 而实际传入的参数是：S1对象和S2对象

def Func(F1 obj):
    """Func函数需要接收一个F1类型或者F1子类的类型"""

    print obj.show()

s1_obj = S1()
Func(s1_obj) # 在Func函数中传入S1类的对象 s1_obj，执行 S1 的show方法，结果：S1.
show

s2_obj = S2()
Func(s2_obj) # 在Func函数中传入Ss类的对象 ss_obj，执行 Ss 的show方法，结果：S2.
show
```

Python 天生就是支持多态，因为他是弱类型语言，不需要指定类型。 Python“鸭子类型”。

```
class F1(object):
    def show(self):
        print 'F1.show'

class S1(F1):

    def show(self):
        print 'S1.show'

class S2(F1):

    def show(self):
        print 'S2.show'

def Func(obj):
    print obj.show()

s1_obj = S1()
Func(s1_obj)

s2_obj = S2()
Func(s2_obj)
```

7、类属性和实例属性

7.1、类属性和实例属性

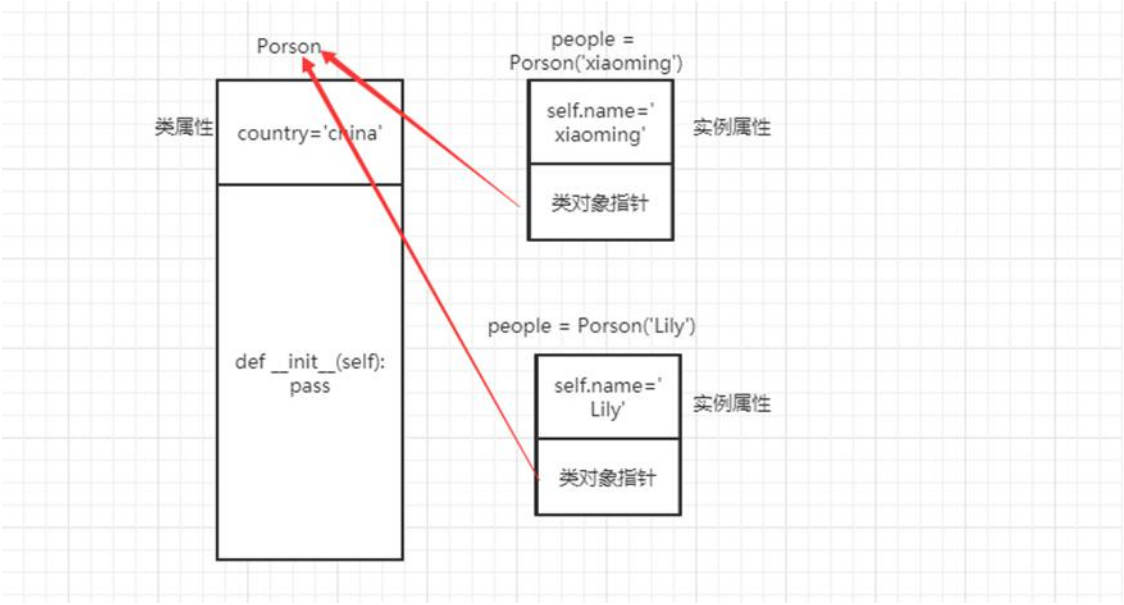
类属性：就是类对象所拥有的属性，它被所有类对象的实例对象所共有，类对象和实例对象可以访问。

实例属性：实例对象所拥有的属性，只能通过实例对象访问。

```
class Person(object):
    country = 'china'  # 类属性
    def __init__(self,name):
        self.name = name
people = Person('xiaoming')
print(people.country)  # 可以访问类属性
print(Person.country)# 可以访问类属性

# print(Person.name)  # 类对象不可以访问实例属性
print(people.name)  # 实例对象可以访问实例属性
```

类属性类对象可以访问，实例对象也可以访问，这与内存中保存的方式有关。



上图中可以看出，所有实例对象的类对象指针指向同一类对象。实例属性在每个实例中独有一份，而类属性是所有实例对象共有一份。

访问修改类属性：用实例对象修改

```
class Person(object):
    country = 'china'  # 类属性
    def __init__(self,name):
        self.name = name

people = Person('xiaoming')
people.country = 'japan'    # 通过实例对象修改类属性
print(Person.country)       # 再次访问类属性并没有修改，实际上是生成了一个实例属性
# 输出 'china'
print(people.country)    # 访问实例属性
```

类对象引用修改：

```
class Person(object):
    country = 'china'  # 类属性
    def __init__(self,name):
        self.name = name
people = Person('xiaoming')
Person.country = 'japan'    # 通过类对象修改类属性
print(Person.country)       # 再次访问类属性发现已经类属性已经修改
# 输出 'japan'
```

如果需要在类外修改类属性，必须通过类对象去引用然后进行修改。如果通过实例对象去引用，会产生一个同名的实例属性，这种方式修改的是实例属性，不会影响到类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是实例属性，除非删除了该实例属性。

8、类方法和静态方法

8.1、类方法

类对象所拥有的方法，需要用装饰器@classmethod 来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以 cls 作为第一个参数，类方法可以通过

类对象，实例对象调用。

```
class Person(object):
    country = 'china'  # 类属性
    def __init__(self,name):
        self.name = name

    # 类方法，用装饰器 classmethod 装饰
    @classmethod
    def get_country(cls):
        print(cls.country)

people = Person('xiaoming')
result = Person.get_country() # 获取类属性
# 打印出 'china'
```

类方法主要可以对类属性进行访问，修改。

静态方法： 类对象所拥有的方法，需要用@staticmethod 来表示静态方法，静态方法不需要任何参数。

```
class Person(object):
    country = 'china'  # 类属性
    def __init__(self,name):
        self.name = name

    # 静态方法，用装饰器 staticmethod 装饰
    @staticmethod
    def get_country():  # 静态方法不用传任何参数
        print(Person.country)

people = Person('xiaoming')
result = Person.get_country() # 获取类属性
# 输出 'china'
```

8.2 、类方法、实例方法、静态方法对比

1、类方法的第一个参数是类对象 cls，通过 cls 引用的类对象的属性和方法

2、实例方法的第一个参数是实例对象 self，通过 self 引用的可能是类属性、也有可能是实例属性(这个需要具体分析)，不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。

3、静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用。