面向对象特性

- 封装

1. 对象都有明确的边界，把属性保护在边界之内，内部的变化对外部的变化没有影响
2. 封装的粒度：
    粒度过大：导致对象过于复杂，不利于各司其职
    粒度过小：造成对象过于简单，让过程过于复杂

- 继承

1. 类和类之间的关系
    动物类---马类，猫类，鸟，鱼类
    父类-----子类
2. 继承：一定要满足 子类 is a 父类
3. 父类是子类的共性的抽象
4. 父类---子类 ： 一般---特殊
5. 语法：
    class 子类名（父类名）：
    父类：基类，超类
    object：是所有类的父类---根类

- 继承的语法规则

1. 父类拥有的成员，子类可以继承
2. 子类不能继承父类的私有成员
3. Python中的继承是多继承
4. 继承具有可扩展性
    父类扩展了子类
5. 如果子类没有创建任何初始化方法，则调用父类的初始化方法
    调用父类的初始化方法：
        1. 父类类名.__init__(self)
        2. super().__init__()
6. 子类的修改，不会影响其他子类

```
class Father:
    __money=1000000
    def football(self):
        print('football play good')
    def __init__(self):
        print('this is your father')

class AfterFather:#干爹
    money=100000
    def __init__(self):
        print('this is your afterfather')

class Children(Father,AfterFather):
    def __init__(self):  # self=c
        print('this is your son')
        # Father.__init__(self)
        # AfterFather.__init__(self)
```

```
        super().__init__()

c=Children()  # c
print(c.money)
c.football()
```

- 方法覆盖

1. 父类中拥有的方法（非私有），子类必定拥有，子类也实现了同名的方法，此时创建对象后，调用该方法，只会调用子类的方法（遮蔽）
2. 子类的特殊实现，遮蔽了父类的一般实现

```
class Animal:
    def eat(self):
        print('animal can eat')

class Cat(Animal):
    def eat(self):
        print('cat can eat fish')

class Dog(Animal):
    def eat(self):
        print('dog can eat bone')

dog=Dog()
cat=Cat()
```

```
dog.eat()
cat.eat()
```

- 接口

```
1. 接口就是标准
2. 父类中的方法和属性就是一种标准（接口）
3. 作用：指导，规定所有继承于该父类的子类，应该有的属性和方
4. 接口回调：
        某个标准还尚未完成时，已经可以调用该标准（需要一个预先定好的接口名）
```

```
def fun(n):  # 先使用了hehe 属性
    print(n.hehe) # n 应该有hehe属性---标准（接口）



class A:
    hehe='alksjdlaksj'

a=A()
fun(a)
```

多态

- 多态

1. 一种事物具有多种形态
2. 没有继承就没有多态
3. 类和类之间的关系

```
class Animal(object):
    def eat(self):
        print('animal can eat')

class Dog(Animal):
    pass

class Cat(Animal):
    pass

class BigOrange(Cat): # BigOrange---Cat---Animal---object
    pass

def fun(animal): # 参数应该是一个动物，或动物的子类
    # 为了保证调用的正确性，必须传入一个Animal的子类，
    # Animal本身就拥有eat（）方法,子类必定也有eat（）方法
    animal.eat()   # 接口

dog=Dog()
cat=Cat()
bo=BigOrange()
```

```
# fun(dog)
# fun(cat)
fun(bo)
```

- 多态性

1. 向不同的对象发送同一条消息，不同的对象会有不同的行为
2. 对象和对象的关系
3. 和继承无关

```
class Duck: # 鸭子    鸭子类型
    def shut(self):
        print('duck can duck')

    def swim(self):
        print('duck can swim')

    fur='鸭毛'

class Bird:
    def shut(self):
        print('bird can biubiu')

    def swim(self):
        print('bird can swim')

    fur='鸟毛'
```

```
class Person:
    def shut(self):
        print('person can duck')

    def swim(self):
        print('person can swim')

    fur='羽绒服'


def fun(n):
    print(n.fur)
    n.swim()
    n.shut()
    print('**********************')


d=Duck()
b=Bird()
p=Person()

fun(d)
fun(b)
fun(p)
```

- 优势

1. 多态：
        增加了程序的可扩展性
2. 多态性：
        增加了程序的灵活性

补充：

- 钻石继承问题

1. Python是多继承
2. 也称之为：菱形继承
3. 问题描述：
        一个类如果有多个子类，多个子类又拥有相同的子类，如果使用最后一级的子类，高级的父类会被多次创建，大量的浪费空间
4. 解决方案：
        使用super类
        super().__init__()
5. 原理：
        super（）的底层，使用的就是mro
        mro：继承链式关系
        mro：所有的类都会出现，且只出现一次
        机理：mro的继承关系，每个类只有一个，如果使用mro调用__init__不会造成资源浪费

```
class A:
    def __init__(self):
```

```
        print('A')

class B(A):
    def __init__(self):
        # A.__init__(self)
        super().__init__()
        print('B')

class C(A):
    def __init__(self):
        # A.__init__(self)
        super().__init__()
        print('C')

class D(B,C):
    def __init__(self):
        # B.__init__(self)
        # C.__init__(self)
        super().__init__()
        print('D')

D()
print(D.mro())
print(D.__mro__)
```

- 内置函数

1. issubclass(cls1,cls2 or tuple)
        判断cls1是否是cls2的子类或cls1是否是元组中的类的子类（只要有一个父类，就为真）
2. isinstance(obj,cls or tuple)
        判断obj，是否是cls的实例对象，或是否是元组中的类的实例对象
3. hasattr(obj,name)
        判断，obj对象中，是否有name这个属性
4. getattr(object, name[, default])
        获得object中的name属性，如果没有则返回default
5. setattr(obj, name, value)
        给obj设置name属性，值为value
6. delattr(obj, name)
        删除obj中的name属性

```
class A:pass
class B(A):pass
class C(A):pass
class D(B):pass
class E:pass

print(issubclass(B,A))
print(issubclass(B,(A,C,D,E)))
print(issubclass(B,E))
print(issubclass(B,B))

print(isinstance(B(),A))
print(isinstance(B(),B))
```

```
print(isinstance(B(),E))
print(isinstance(B(),(A,C,D,E)))

class  F:
    age=18

f=F()
print(hasattr(f,'age'))
print(hasattr(f,'age2'))

print(getattr(f,'age'))
print(getattr(f,'age2','error'))

setattr(f,'age2',20)
print(getattr(f,'age2'))

delattr(f,'age2')
print(getattr(f,'age2','error'))
```