类的继承

基本概念

面向对象三要素之一，继承Inheritance

人类和猫类都继承自动物类。
个体继承自父母，继承了父母的一部分特征，但也可以有自己的个性。
在面向对象的世界中，从父类继承，就可以直接拥有父类的属性和方法，这样可以减少代码、多复用。子类可以定义自己的属性和方法。

看一个不用继承的例子

```
class Animal:
    def shout(self):
        print('Animal shouts')

a = Animal()
a.shout()

class Cat:
    def shout(self):
        print('Cat shouts')

c = Cat()
c.shout()
```

上面的2个类虽然有关系，但是定义时并没有建立这种关系，而是各自完成定义。
动物类和猫类都有吃，但是它们的吃有区别，所以分别定义。

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

    def shout(self): # 一个通用的吃方法
        print('{} shouts'.format(self.__class__.__name__))

    @property
    def name(self):
```

```
        return self._name

a = Animal('monster')
a.shout()

class Cat(Animal):
    pass

cat = Cat('garfield')
cat.shout()
print(cat.name)

class Dog(Animal):
    pass

dog = Dog('ahuang')
dog.shout()
print(dog.name)
```

上例可以看出，通过继承，猫类、狗类不用写代码，直接继承了父类的属性和方法。

继承
class Cat(Animal) 这种形式就是从父类继承，括号中写上继承的类的列表。
继承可以让子类从父类获取特征（属性和方法）

父类
Animal就是Cat的父类，也称为基类、超类。

子类
Cat就是Animal的子类，也称为派生类。

定义

格式如下

```
class 子类名(基类1[,基类2,...]):
    语句块
```

如果类定义时，没有基类列表，等同于继承自object。在Python3中，object类是所有对象的根基类。

```
class A:
    pass
# 等价于
class A(object):
    pass
```

注意，上例在Python2中，两种写法是不同的。

Python支持多继承，继承也可以多级。

查看继承的特殊属性和方法有

特殊属性和方法	含义	示例
__base__	类的基类	
__bases__	类的基类元组	
__mro__	显示方法查找顺序，基类的元组	
mro()方法	同上	int.mro()
__subclasses__()	类的子类列表	int.__subclasses__()

继承中的访问控制

```
class Animal:
    __COUNT = 100
    HEIGHT = 0

    def __init__(self, age, weight, height):
        self.__COUNT += 1
        self.age = age
        self.__weight = weight
        self.HEIGHT = height

    def eat(self):
        print('{} eat'.format(self.__class__.__name__))

    def __getweight(self):
        print(self.__weight)
```

```
    @classmethod
    def showcount1(cls):
        print(cls.__COUNT)

    @classmethod
    def __showcount2(cls):
        print(cls.__COUNT)

    def showcount3(self):
        print(self.__COUNT)

class Cat(Animal):
    NAME = 'CAT'
    __COUNT = 200


# c = Cat() # __init__函数参数错误
c = Cat(3, 5, 15)
c.eat()
print(c.HEIGHT)
# print(c.__COUNT) #私有的不可访问
# c.__showweight() #私有的不可访问
c.showcount1()
# c.__showcount2() #私有的不可访问
c.showcount3()
print(c.NAME)

print("{}".format(Animal.__dict__))
print("{}".format(Cat.__dict__))
print(c.__dict__)
print(c.__class__.mro())
```

从父类继承，自己没有的，就可以到父类中找。
私有的都是不可以访问的，但是本质上依然是改了名称放在这个属性所在类的了__dict__中。知道这个新名称就可以直接找到这个隐藏的变量，这是个黑魔法技巧，慎用。

总结
继承时，公有的，子类和实例都可以随意访问；私有成员被隐藏，子类和实例不可直接访问，当私有变量所在的类内的方法中可以访问这个私有变量。
Python通过自己一套实现，实现和其它语言一样的面向对象的继承机制。

属性查找顺序
实例的__dict__ 》 类__dict__ 如果有继承==》 父类 __dict__
如果搜索这些地方后没有找到就会抛异常，先找到就立即返回了。

方法的重写、覆盖override

```
class Animal:
    def shout(self):
        print('Animal shouts')

class Cat(Animal):
    # 覆盖了父类方法
    def shout(self):
        print('miao')

a = Animal()
a.shout()
c = Cat()
c.shout()

print(a.__dict__)
print(c.__dict__)
print(Animal.__dict__)
print(Cat.__dict__)

# Animal shouts
# miao
```

Cat中能否覆盖自己的方法吗？

```
class Animal:
    def shout(self):
        print('Animal shout')

class Cat(Animal):
    # 覆盖了父类方法
    def shout(self):
        print('miao')
    # 覆盖了自身的方法，显式调用了父类的方法
```

```
    def shout(self):
        print(super())
        print(super(Cat, self))
        super().shout()
        super(Cat, self).shout() # 等价于super()
        self.__class__.__base__.shout(self) # 不推荐

a = Animal()
a.shout()
c = Cat()
c.shout()

print(a.__dict__)
print(c.__dict__)
print(Animal.__dict__)
print(Cat.__dict__)
```

super()可以访问到父类的属性，其具体原理后面说。

那对于类方法和静态方法呢？

```
class Animal:
    @classmethod
    def class_method(cls):
        print('class_method_animal')

    @staticmethod
    def static_method():
        print('static_method_animal')

class Cat(Animal):
    @classmethod
    def class_method(cls):
        print('class_method_cat')

    @staticmethod
    def static_method():
        print('static_method_cat')

c = Cat()
c.class_method()
```

```
c.static_method()
```

这些方法都可以覆盖，原理都一样，属性字典的搜索顺序。

继承中的初始化

先看下面一段代码，有没有问题

```
class A:
    def __init__(self, a):
        self.a = a

class B(A):
    def __init__(self, b, c):
        self.b = b
        self.c = c

    def printv(self):
        print(self.b)
        print(self.a) # 出错吗？

f = B(200,300)
print(f.__dict__)
print(f.__class__.__bases__)
f.printv()
```


上例代码可知：
如果类B定义时声明继承自类A，则在类B中__bases__中是可以看到类A。
但是这和是否调用类A的构造方法是两回事。
如果B中调用了A的构造方法，就可以拥有父类的属性了。如何理解这一句话呢？
观察B的实例f的__dict__中的属性。

```
class A:
    def __init__(self, a, d=10):
        self.a = a
        self.__d = d


class B(A):
```

```
    def __init__(self, b, c):
        A.__init__(self, b + c, b - c)
        self.b = b
        self.c = c

    def printv(self):
        print(self.b)
        print(self.a) #

f = B(200,300)
print(f.__dict__)
print(f.__class__.__bases__)
f.printv()
```

作为好的习惯，如果父类定义了__init__方法，你就该在子类的__init__中调用它。

那子类什么时候自动调用父类的__init__方法呢？

示例1

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        self.__a2 = 'a2'
        print('A init')

class B(A):
    pass

b = B()
print(b.__dict__)
```

B实例的初始化会自动调用基类A的__init__方法

示例2

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        self.__a2 = 'a2'
        print('A init')
```

```
class B(A):
    def __init__(self):
        self.b1 = 'b1'
        print('B init')

b = B()
print(b.__dict__)
```

B实例的初始化__init__方法不会自动调用父类的初始化__init__方法，需要手动调用。

```
class A:
    def __init__(self):
        self.a1 = 'a1'
        self.__a2 = 'a2'
        print('A init')

class B(A):
    def __init__(self):
        self.b1 = 'b1'
        print('B init')
        A.__init__(self)

b = B()
print(b.__dict__)
```


如何正确初始化

```
class Animal:
    def __init__(self, age):
        print('Animal init')
        self.age = age

    def show(self):
        print(self.age)

class Cat(Animal):
    def __init__(self, age, weight):
        print('Cat init')
        self.age = age + 1
        self.weight = weight
```

```
c = Cat(10, 5)
c.show()
```

上例我们前面都分析过，不会调用父类的__init__方法的，这就会导致没有实现继承效果。所以在子类的__init__方法中，应该显式调用父类的__init__方法。

```
class Animal:
    def __init__(self, age):
        print('Animal init')
        self.age = age

    def show(self):
        print(self.age)

class Cat(Animal):
    def __init__(self, age, weight):
        # 调用父类的__init__方法的顺序决定着show方法的结果
        super().__init__(age)
        print('Cat init')
        self.age = age + 1
        self.weight = weight
        # super().__init__(age)

c = Cat(10, 5)
c.show()
```

注意，调用父类的__init__方法，出现在不同的位置，可能导致出现不同的结果。
那么，直接将上例中所有的实例属性改成私有变量呢？

```
class Animal:
    def __init__(self, age):
        print('Animal init')
        self.__age = age

    def show(self):
        print(self.__age)

class Cat(Animal):
    def __init__(self, age, weight):
```

```
        # 调用父类的__init__方法的顺序决定着show方法的结果
        super().__init__(age)
        print('Cat init')
        self.__age = age + 1
        self.__weight = weight
        # super().__init__(age)

c = Cat(10, 5)
c.show()

print(c.__dict__)
```

上例中打印10，原因看__dict__就知道了。因为父类Animal的show方法中__age会被解释为_Animal__age，因此显示的是10，而不是11。
这样的设计不好，Cat的实例c应该显示自己的属性值更好。

解决的办法：一个原则，自己的私有属性，就该自己的方法读取和修改，不要借助其他类的方法，即使是父类或者派生类的方法。