# 1. 面向对象三大特征--封装

## 1.1. 封装定义

面向对象的程序设计中，某个类把所需要的数据（也可以说是类的属性）和对数据的操作（也可以说是类的行为）全部都封装在类中，分别称为类的成员变量和方法（或成员函数）。这种把成员变量和成员函数封装在一起的编程特性称为封装。

## 1.2. 公有成员变量和私有成员变量

python中用成员变量的名字来区分是共有成员变量或者是私有成员变量。

python中，以两个下划线‘\_\_’开头的变量都是私有成员变量，而其余的变量都属于公有成员变量。

其中，私有的成员变量只能在类的内部访问，而共有的公有的成员变量可以在类的外部进行访问。

## 1.3. 公有方法和私有方法

如上面所说，类的方法是对类行为的封装。

类的方法也分为**公有方法**和**私有方法**。

类的私有方法只能通过对象名（在类内部也就是self）在类的内部进行访问。

而公有方法可以在类的外部通过对象名进行访问。

和属性不同的是，一般意义上的类方法**属于对象**，也就是说只有通过对象才可以进行调用，不能直接通过类名进行调用。

一般类方法的第一个参数必须是代指类对象本身的（一般我们常用self，实际上可以是任何自定义的名字，只不过self是大家约定俗成的用法，在下面介绍的类方法中，大家一般用cls，因为那里更多地标识的是一个类），可以通过self访问类对象的**成员方法和数据**。

同样，公有的成员方法和私有的成员方法也是通过名字来区分的，双下划线‘\_\_’开头的方法是私有成员方法。

****2.2.2 类方法和静态方法****

python中对象有两种：经典对象和新型对象。

经典对象就是不继承object父类定义出来的对象。

新型对象就是要继承object类定义出的对象。

新型对象和经典对象最大的区别就是，**新型对象中提供了对类方法和静态方法的支持**。

类方法就是被classmethod()函数处理过的函数，能够直接通过****类名进行调用****，也能够****被对象直接调用****。

静态方法相当于类层面的全局函数，可以被****类直接调用****，可以被****所有实例化对象共享****，通过staticmethod()定义静态方法，静态方法没有self参数。

# 私有方法

python默认的成员函数和成员变量都是公开的，[python 私有属性和方法](http://www.cnpythoner.com/post/295.html)没有类似别的语言的public,private等关键词来修饰。 在python中定义私有变量只需要在变量名或函数名前加上 "\_\_"两个下划线，那么这个函数或变量就会为私有的了

## 2.1. 私有方法

**声明该方法为私有方法，不能在类的外部调用**

代码

class Test(object):

#普通方法

def test(self):

print("普通方法test")

#普通方法

def \_test1(self):

print("普通方法\_test1方法")

#私有方法

def \_\_test2(self):

print("私有方法\_\_test2方法")

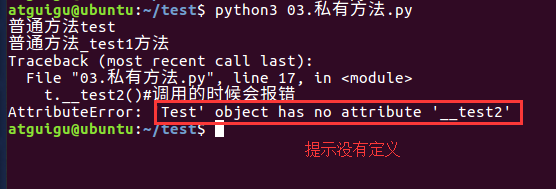
t = Test()

t.test()

t.\_test1()

#t.\_\_test2()#调用的时候会报错

执行效果



外界无法访问私有方法，但可以在类内部方法私有方法。

class Test(object):

#私有方法

def \_\_test2(self):

print("私有方法\_\_test2方法")

#普通方法

def test(self):

print("普通方法test")

#普通方法

def \_test1(self):

print("普通方法\_test1方法")

#可以在类内部调用私有方法

t.\_\_test2()#

self.\_\_test2()

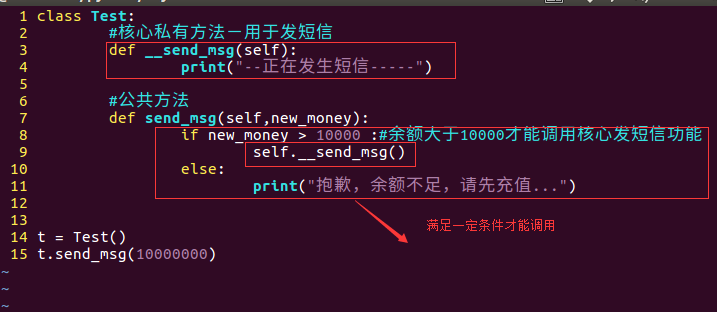
t = Test()

t.test()

t.\_test1()

#t.\_\_test2()和t.test2()#调用的时候会报错

## 2.2. 私有方法应用场景



执行效果



# **3. 私有属性**

如果有一个对象，当需要对其进行修改属性时，有2种方法

对象名.属性名 = 数据 ---->直接修改

对象名.方法名() ---->间接修改

为了更好的保存属性安全，即不能随意修改，一般的处理方式为

将属性定义为私有属性

添加一个可以调用的方法，供调用

## 3.1. 案例

class Person(object):

def \_\_init\_\_(self, name):

self.\_\_name = name

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

if len(new\_name) >= 5:

self.\_\_name = new\_name

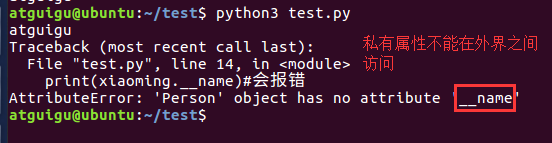
else:

print("error:名字长度需要大于或者等于5")

xiaoming = Person("atguigu")

print(xiaoming.get\_name())

#print(xiaoming.\_\_name)#会报错



class Person(object):

def \_\_init\_\_(self, name):

self.\_\_name = name

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

if len(new\_name) >= 5:

self.\_\_name = new\_name

else:

print("error:名字长度需要大于或者等于5")

p = Person("afu")

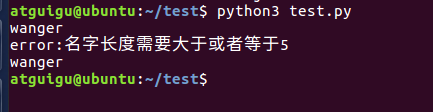
p.set\_name("wanger")

print(p.get\_name())

p.set\_name("lisi")

print(p.get\_name())

执行如图：



## 3.2. 总结

Python中没有像C++中public和private这些关键字来区别公有属性和私有属性

它是以属性命名方式来区分，如果在属性名前面加了2个下划线'\_\_'，则表明该属性是私有属性，否则为公有属性（方法也是一样，方法名前面加了2个下划线的话表示该方法是私有的，否则为公有的）。

## 3.3. 练习

练习，定义一个类Person，类中有私有方法和普通方法，私有属性和普通属性，并且都要调用普通方法和私有方法和私有属性。

class Test(object):

def test(self):

self.\_\_sex = "保密"

print("普通共有方法test")

#调用私有方法

self.\_\_test1()

def \_\_test1(self):

print("私有方法\_\_test1")

#调用私有属性

print("私有属性\_\_sex:",self.\_\_sex)

t = Test()

t.test()

# \_\_del\_\_()方法

创建对象后，python解释器默认调用\_\_init\_\_()方法。当删除一个对象时，python解释器也会默认调用一个方法，这个方法为\_\_del\_\_()方法。在python中，对于开发者来说很少会直接销毁对象(如果需要，应该使用del关键字销毁)。Python的内存管理机制能够很好的胜任这份工作。也就是说,不管是手动调用del还是由python自动回收都会触发\_\_del\_\_方法执行。

## 4.1. 创建多个对象的时候触发\_\_del\_\_方法

Python解释器释放实例对象的时候，调用该对象的\_\_del\_\_方法

各个对象被被Python解释器释放对象的时候调用\_\_del\_\_方法

class Person(object):

def \_\_init\_\_(self,name):

self.name = name

def \_\_del\_\_(self):

print("实例对象:%s"%self.name,id(self))

print("python解释器开始回收%s对象了" % self.name)

print("类对象",id(Person))

zhangsan = Person("张三")

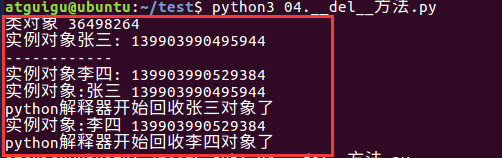
print("实例对象张三:",id(zhangsan))

print("------------")

lisi = Person("李四")

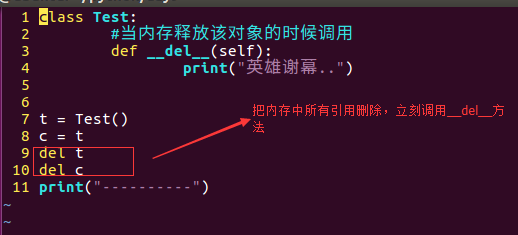
print("实例对象李四:",id(lisi))

执行效果



## 4.2. 使用del删除引用时的调用情况

当使用del 把内存的所有应用删除，立刻调用\_\_del\_\_方法



执行效果



创建对象后，python解释器默认调用\_\_init\_\_()方法；

当删除一个对象时，python解释器也会默认调用一个方法，这个方法为\_\_del\_\_()方法

## 4.3. 创建多个实例对象并且删除

import time

class Animal(object):

# 初始化方法

# 创建完对象后会自动被调用

def \_\_init\_\_(self, name):

print('\_\_init\_\_方法被调用')

self.\_\_name = name

# 析构方法

# 当对象被删除时，会自动被调用

def \_\_del\_\_(self):

print("\_\_del\_\_方法被调用")

print("%s对象马上被干掉了..."%self.\_\_name)

# 创建对象

dog = Animal("哈皮狗")

# 删除对象

del dog

cat = Animal("波斯猫")

cat2 = cat

cat3 = cat

print("---马上 删除cat对象")

del cat

print("---马上 删除cat2对象")

del cat2

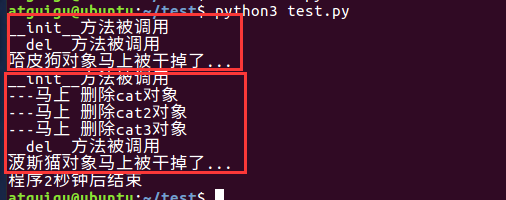
print("---马上 删除cat3对象")

del cat3

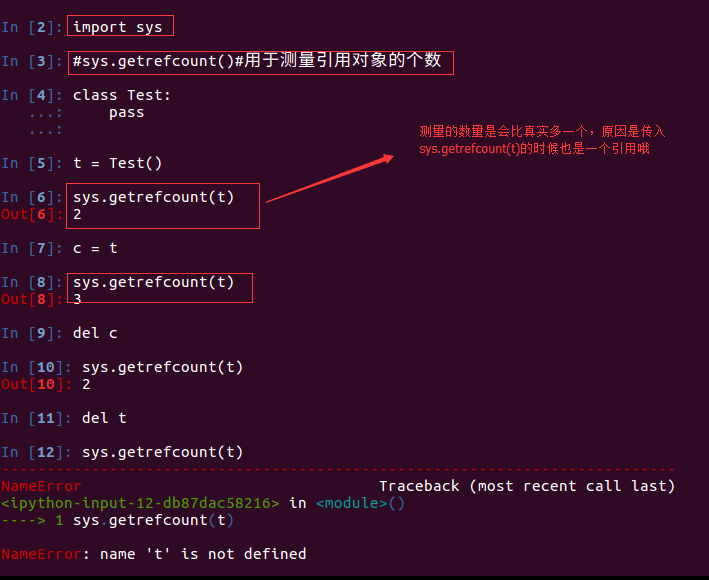
print("程序2秒钟后结束")

time.sleep(2)

结果:



## 4.4. 测量对应引用个数



## 4.5. 小结

当有1个变量保存了对象的引用时，此对象的引用计数就会加1。

当使用del删除变量指向的对象时，如果对象的引用计数不会1，比如3，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除

### 4.5.1. 注意

**del xxx 不会主动调用\_\_del\_\_方法，只有引用计数==0时，\_\_del\_\_()才会被执行，并且定义了\_\_del\_()的实例无法被Python的循环垃圾收集器收集，所以尽量不要自定义\_\_del\_\_()**。一般情况下，\_\_del\_\_() 不会破坏垃圾处理器。

# 5. 面向对象三大特征--继承

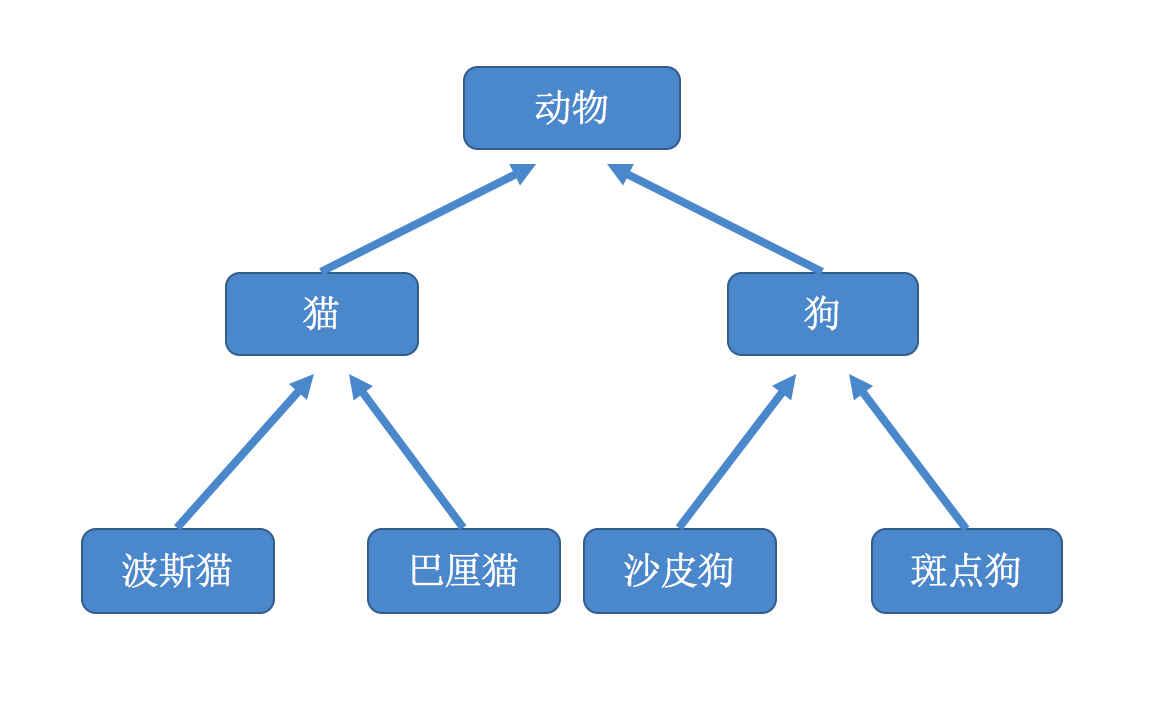
## **5.1. 继承的概念**

在现实生活中，继承一般指的是子女继承父辈的财产，如下图





在程序中，继承描述的是事物之间的所属关系，例如猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物；同理，波斯猫和巴厘猫都继承自猫，而沙皮狗和斑点狗都继承狗，如下如所示：



## **5.2. 继承示例**

### 5.2.1. 狗和猫都是属于动物

Dog类继承Animal类，当调用方法的时候，先调用Dog的方法，没有才去调用父类的方法，同理Cat类也一样；Dog类无法调用Cat类的方法，同理Cat类也无法调用Dog的方法

子类不能调用父类的私有方法和私有属性（类属性）。



### 5.2.2. 多层继承



**多层继承**

# 定义一个父类，如下:

class Cat(object):

def \_\_init\_\_(self, name, color="白色"):

self.name = name

self.color = color

def run(self):

print("%s--在跑"%self.name)

# 定义一个子类，继承Cat类如下:

class Bosi(Cat):

def set\_name(self, newName):

self.name = newName

def eat(self):

print("%s--在吃"%self.name)

bs = Bosi("印度猫")

print('bs的名字为:%s'%bs.name)

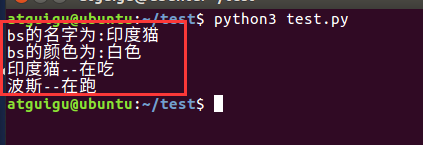
print('bs的颜色为:%s'%bs.color)

bs.eat()

bs.set\_name('波斯')

bs.run()

运行结果:



说明：

虽然子类没有定义\_\_init\_\_方法，但是父类有，所以在子类继承父类的时候这个方法就被继承了，所以只要创建Bosi的对象，就默认执行了那个继承过来的\_\_init\_\_方法

### 5.2.3. 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

私有属性（私有类属性）和私有方法不被继承。

## **5.3. 注意点-私有的属性和方法不会被继承**

class Animal(object):

def \_\_init\_\_(self, name='动物', color='白色'):

self.\_\_name = name

self.color = color

def \_\_test(self):

print(self.\_\_name)

print(self.color)

def test2(self):

print(self.\_\_name)

print(self.color)

class Dog(Animal):

def dog\_test1(self):

#print(self.\_\_name) #不能访问到父类的私有属性

print(self.color)

def dog\_test2(self):

#self.\_\_test() #不能访问父类中的私有方法

self.test2()

A = Animal()#print(A.\_\_name) #程序出现异常，不能访问私有属性

print(A.color)#A.\_\_test() #程序出现异常，不能访问私有方法

A.test2()

print("------分割线-----")

dog = Dog(name = "小花狗", color = "黄色")

dog.dog\_test1()

dog.dog\_test2()

私有的属性，不能通过对象直接访问，但是可以通过方法访问

**私有的方法，不能通过对象直接访问**

**私有的属性、方法，不会被子类继承，也不能被访问**

一般情况下，私有的属性、方法都是不对外公布的，往往用来做内部的事情，起到安全的作用

## 5.4. 重写父类方法与调用父类方法

### 5.4.1. 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

#coding=utf-8

class Cat(object):

def sayHello(self):

print("hello-----1")

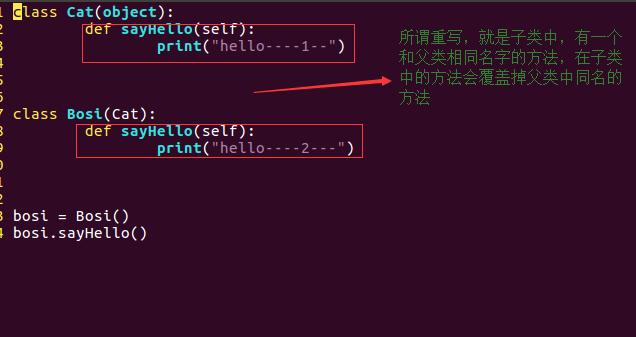
class Bosi(Cat):

def sayHello(self):

print("hello-----2")

bosi = Bosi()

bosi.sayHello()



执行效果

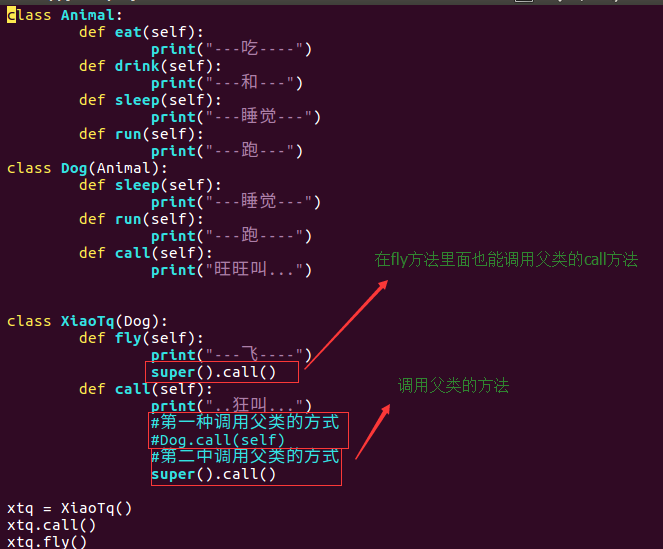


但是要注意的时候，子类的方法覆盖父类的方法，只要名字相同，但参数不一样也是覆盖，这个时候是否要传递参数这个使用根据子类的方法是否有参数。

### 5.4.2. 使用super调用父类的方法

super并不是一个函数，是一个类名，形如super(B, self)事实上调用了super类的初始化函数，产生了一个super对象。

super类的初始化函数并没有做什么特殊的操作，只是简单记录了类类型和具体实例；



#coding=utf-8

class Cat(object):

def \_\_init\_\_(self,name):

self.name = name

self.color = 'yellow'

class Bosi(Cat):

def \_\_init\_\_(self,name):

# 调用父类的\_\_init\_\_方法1(python2)

#Cat.\_\_init\_\_(self,name)

# 调用父类的\_\_init\_\_方法2

#super(Bosi,self).\_\_init\_\_(name)

# 调用父类的\_\_init\_\_方法3

super().\_\_init\_\_(name)

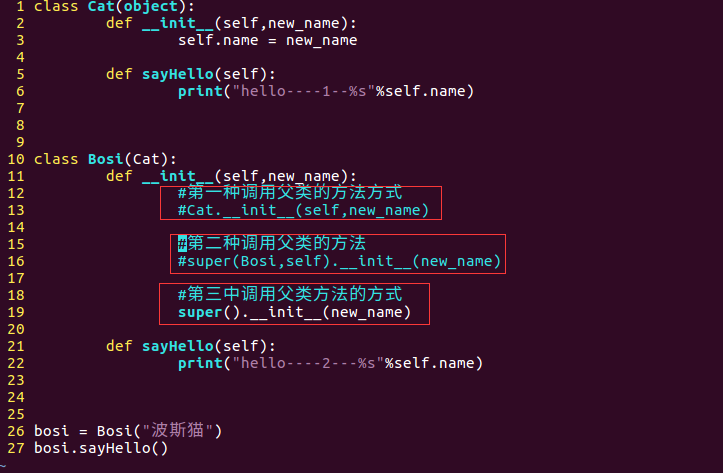
def getName(self):

return self.name

bosi = Bosi('xiaohua')

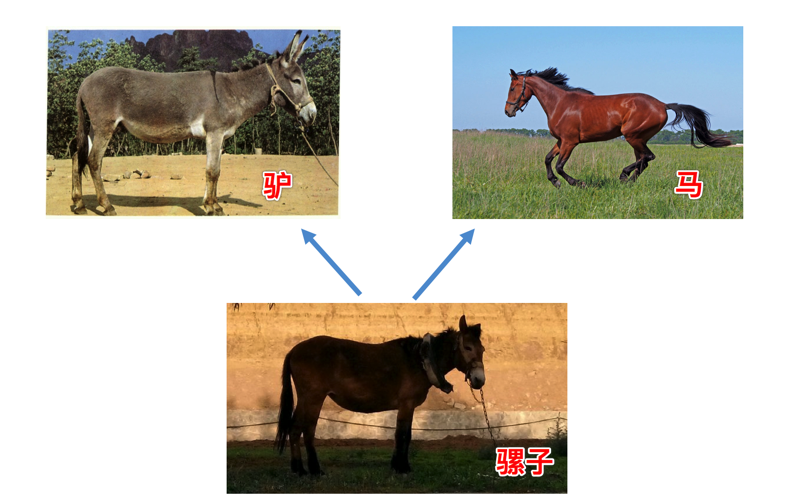
print(bosi.name)

print(bosi.color)



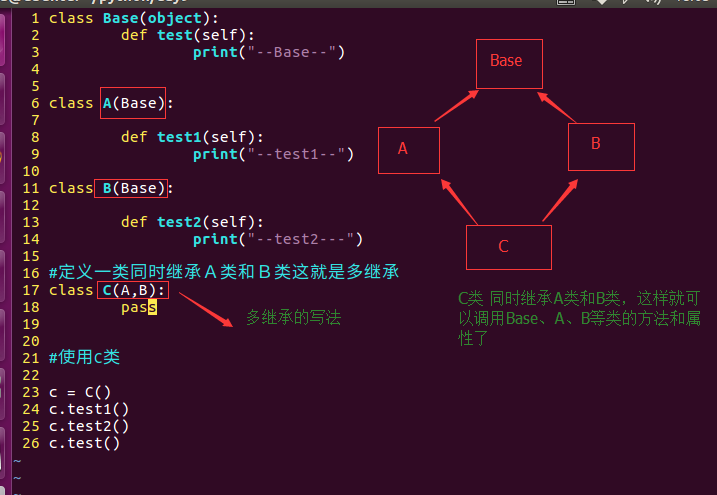
# 6. Python的多继承

## **6.1. 多继承概念**



从图中能够看出，所谓多继承，即子类有多个父类，并且具有它们的特征

Python中多继承的格式如下:



### 6.1.1. 案例

# 定义一个父类

class A:

def printA(self):

print('----A----')

# 定义一个父类

class B:

def printB(self):

print('----B----')

# 定义一个子类，继承自A、B

class C(A,B):

def printC(self):

print('----C----')

obj\_C = C()

obj\_C.printA()

obj\_C.printB()

运行结果:

----A----

----B----

**说明**

python中是可以多继承的

父类中的方法、属性，子类会继承

## **6.2. 注意点**

想一想:

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？

#coding=utf-8

class base(object):

def test(self):

print('----base test----')

class A(base):

def test(self):

print('----A test----')

# 定义一个父类

class B(base):

def test(self):

print('----B test----')

# 定义一个子类，继承自A、B

class C(A,B):

pass

obj\_C = C()

obj\_C.test()

print(C.\_\_mro\_\_) #可以查看C类的对象搜索方法时的先后顺序

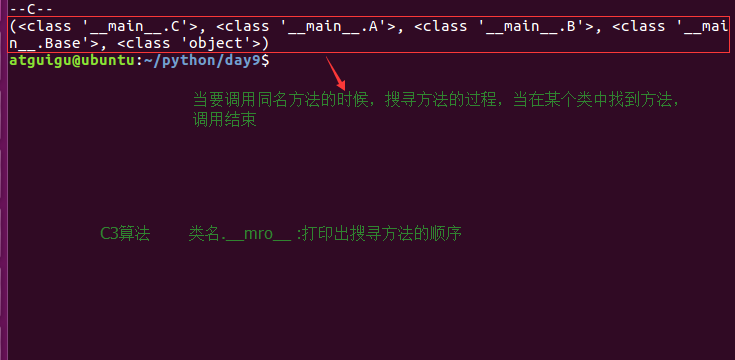
### 6.2.1. 子类方法和父类重名时调用子类方法结束





### 6.2.2. 查看某个类调用父类方法搜寻过程

### 6.2.3. 类名.\_\_mro\_\_方法的作用



# **7. 面向对象三大特征--多态**

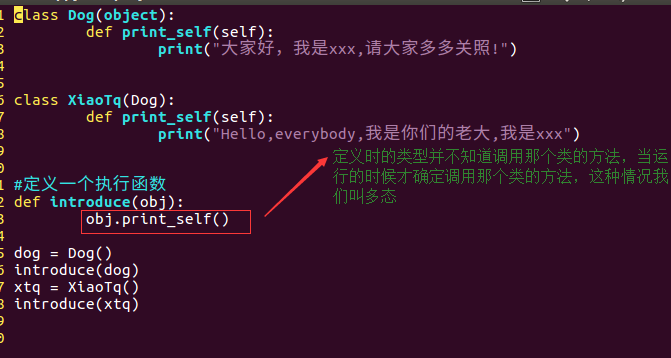
## 7.1. 多态定义

**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。

**多态**指的是一类事物有多种形态，（一个抽象类有多个子类，因而多态的概念依赖于继承）。

当子类和父类都存在相同的print\_self()方法时，我们说，子类的print\_self()覆盖了父类的print\_self()，在代码运行的时候，总是会调用子类的print\_self()。这样，我们就获得了继承的另一个好处：**多态**。

## 7.2. 多态的好处



Python伪代码实现Java或C#的多态

下面的并不是java代码

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'

# 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F1# 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 “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):**

**obj.show()**

s1\_obj = S1()

Func(s1\_obj)

s2\_obj = S2()

Func(s2\_obj)

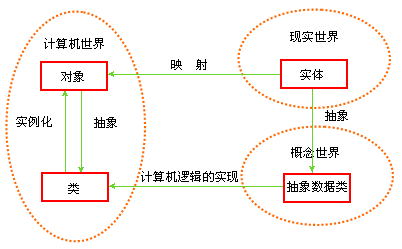
## 7.3. 面向对象四大特征

### 7.3.1. 抽象

忽略一个主题中与当前目标无关的东西,专注的注意与当前目标有关的方面.(就是把现实世界中的某一类东西,提取出来,用程序代码表示,抽象出来的一般叫做类或者接口).抽象并不打算了解全部问题,而是选择其中的一部分,暂时不用部分细节.抽象包括两个方面,一个数据抽象,而是过程抽象.

数据抽象 -->表示世界中一类事物的特征,就是对象的属性.比如鸟有翅膀,羽毛等(类的属性)

过程抽象 -->表示世界中一类事物的行为,就是对象的行为.比如鸟会飞,会叫(类的方法)



### 7.3.2. 封装

有些时候我们不希望把对象的属性公开，就可以把它设为私有，python并没有像其他语言对成员的权限控制系统，如private和protected，默认情况下，python的所有属性都是公有的，可以被访问到，要设成私有，则在前面加双下划线。

### 7.3.3. 继承

继承可以使子类拥有父类的属性和方法，并且可以重写这些方法，加强代码的复用性，python中子类可以有多个父类，但是不建议这样使用，一般会产生重复调用的问题，Super().方法名，可以调用父类的方法（不用传参，作用是调用父类的方法，传的是子类实例的值）

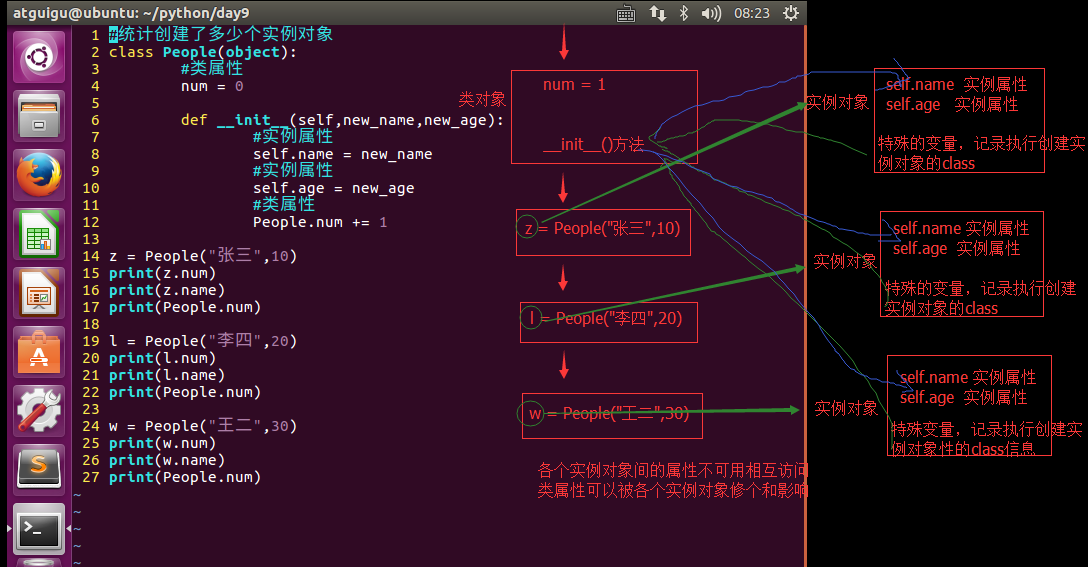
### 7.3.4. 多态

指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。

# **8. 类属性、实例属性**

在前面的例子中我们接触到的就是**实例属性**（对象属性），顾名思义，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，在内存中只存在一个副本，这个和C++中类的静态成员变量有点类似。对于公有的类属性，在类外可以通过类对象和实例对象访问

## **8.1. 类属性**



class Person(object):

name = 'Tom' #公有的类属性

\_\_age = 12 #私有的类属性

p = Person()

print(p.name) #正确

print(Person.name) #正确

#print(p.\_\_age) #错误，不能在类外通过实例对象访问私有的类属性

#print(Person.\_\_age) #错误，不能在类外通过类对象访问私有的类属性

## **8.2. 实例属性(对象属性)**

class Person(object):

address = '贵州' #类属性

def \_\_init\_\_(self):

self.name = 'xiaoyang' #实例属性

self.age = 20 #实例属性

p = Person()

p.age =12 #实例属性

print(p.address) #正确

print(p.name) #正确

print(p.age) #正确

print(Person.address) #正确

#print(Person.name) #错误

#print(Person.age) #错误

### **8.2.1. 通过实例(对象)去修改类属性**

class People(object):

country = 'china' #类属性

print(People.country)

p = People()

print("p.country:",p.country)

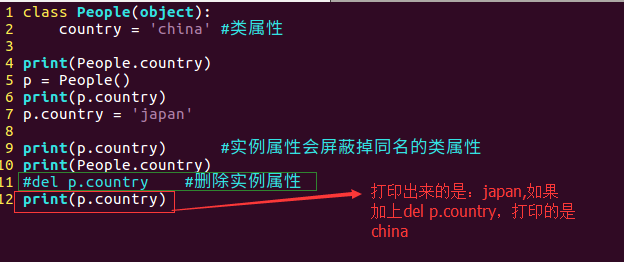
p.country = 'japan'

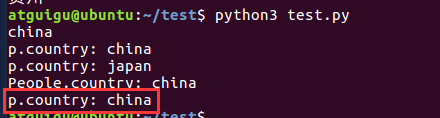
print("p.country:",p.country) #实例属性会屏蔽掉同名的类属性

print("People.country:",People.country)

del p.country #删除实例属性

print("p.country:",p.country)





### **8.2.2. 总结**

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。

# **静态方法和类方法**

Python的静态方法和类成员方法都可以被类或实例访问，两者概念不容易理清，但还是有区别的：

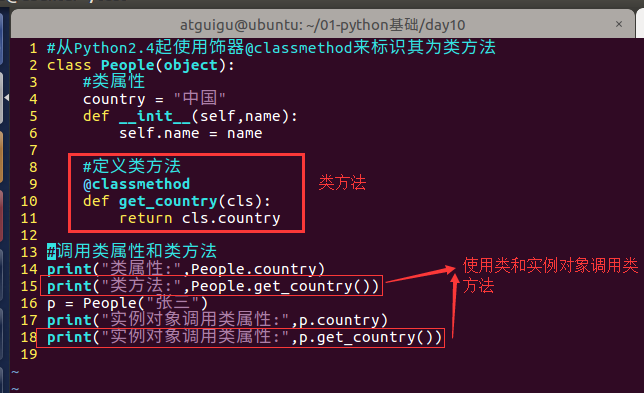
1）静态方法无需传入self参数，类成员方法需传入代表本类的cls参数；

2）从第1条，静态方法是无法访问实例变量的，而类成员方法(类方法)也同样无法访问实例变量，但可以访问类变量；

3）静态方法有点像函数工具库的作用，而类成员方法则更接近类似Java面向对象概念中的静态方法。

## **9.1. 类方法**

是类对象所拥有的方法，从Python2.4后需要用修饰器@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（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了），能够通过实例对象和类对象去访问。



#从Python2.4起使用饰器@classmethod来标识其为类方法

class People(object):

#类属性

country = "中国"

def \_\_init\_\_(self,name):

self.name = name

#定义类方法

@classmethod

def get\_country(cls):

return cls.country

#调用类属性和类方法

print("类属性:",People.country)

print("类方法:",People.get\_country())

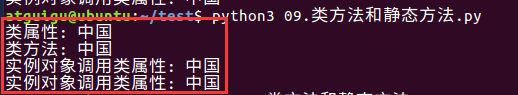
p = People("张三")

print("实例对象调用类属性:",p.country)

print("实例对象调用类属性:",p.get\_country())

注意也可以在类方法加上参数

执行结果：



类方法还有一个用途就是可以对类属性进行修改：

#从Python2.4起使用饰器@classmethod来标识其为类方法

class People(object):

#类属性

country = "中国"

def \_\_init\_\_(self,name):

self.name = name

#定义类方法

@classmethod

def get\_country(cls):

return cls.country

@classmethod

def set\_country(cls,country):

cls.country = country

#调用类属性和类方法

print("类属性:",People.country)

print("类方法:",People.get\_country())

p = People("张三")

#设置

p.set\_country("韩国")

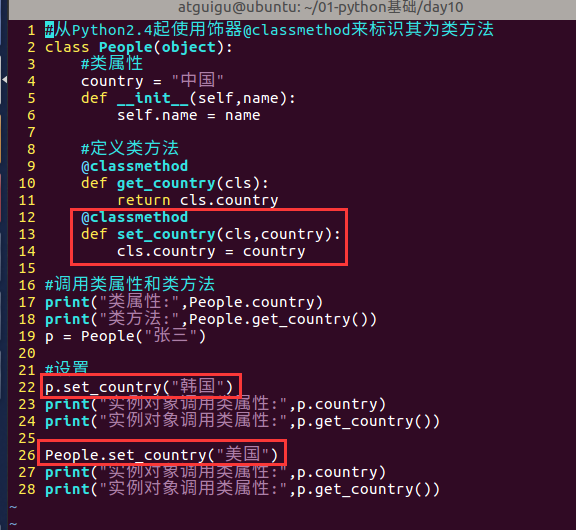
print("实例对象调用类属性:",p.country)

print("实例对象调用类属性:",p.get\_country())

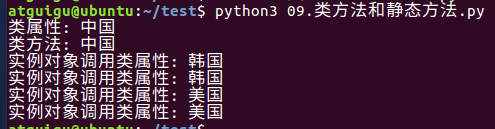
People.set\_country("美国")

print("实例对象调用类属性:",p.country)

print("实例对象调用类属性:",p.get\_country())



执行效果



结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变。

## **9.2. 静态方法**

从Python2.4后需要通过修饰器@staticmethod来进行修饰，静态方法不需要多定义参数，当调用的方法和类对象和实例对象没有关系的时候就可以用静态方法.



#从Python2.4起使用饰器@classmethod来标识其为静态方法

class People(object):

#类属性

country = "中国"

def \_\_init\_\_(self,name):

self.name = name

#静态方法

@staticmethod

def get\_country():

return People.country

@staticmethod

def print\_info(info):

print(info)

#调用静态方法

People.get\_country()

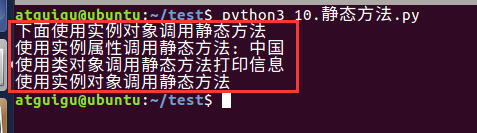
print("下面使用实例对象调用静态方法")

p = People("张三")

print("使用实例属性调用静态方法:",p.get\_country())

People.print\_info("使用类对象调用静态方法打印信息")

p.print\_info("使用实例对象调用静态方法")



## **9.3. 总结**

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，那么通过cls引用的必定是类对象的属性和方法；

而实例方法的第一个参数是实例对象self，那么通过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**。

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.**

**静态方法：**无法访问类属性、实例属性，相当于一个相对独立的方法，跟类其实没什么关系，换个角度来讲，其实就是放在一个类的作用域里的函数而已。

**类成员方法：**可以访问类属性，无法访问实例属性。

思考：类方法可以调用实例属性吗？

# 10.Python新式类与经典类(旧式类)的区别

## 10.1. 新式类都从object继承，经典类不需要

Python 2.x中默认都是经典类，只有显式继承了object才是新式类

Python 3.x中默认都是新式类,经典类被移除，不必显式的继承object

#coding=utf-8

class A:

def \_\_init\_\_(self):

print ('a',)

class B(A):

def \_\_init\_\_(self):

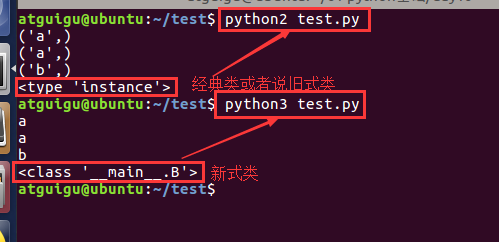
A().\_\_init\_\_()

print ('b',)

d=B()

print(type(d))

运行结果如图：



## 10.2. 经典类采用深度优先，新式类采用广度优先

新式类的MRO(method resolution order 基类搜索顺序)算法采用C3算法**广度优先搜索**，而旧式类的MRO算法是采用**深度优先搜索**。

新式类相同父类只执行一次构造函数，经典类重复执行多次。

下面是经典类

#coding=utf-8

class A(object):

def test(slef):

print("A中的test")

class B(A):

def test(slef):

print("B中的test")

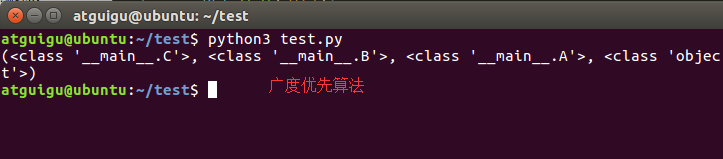
class C(B):

def test(slef):

print("C中的test")

#C是大写是类名

print(C.\_\_mro\_\_)



截止到python2.1，只存在旧式类。旧式类中，类名和type是无关的：如果x是一个旧式类，那么x.\_\_class\_\_定义了x的类名，但是type(x)总是返回<type 'instance'>。这反映了所有的旧式类的实例是通过一个单一的叫做instance的内建类型来实现的，这是它和类不同的地方。

新式类是在python2.2为了统一类和实例引入的。一个新式类只能由用户自定义。如果x是一个新式类的实例，那么type(x)和x.\_\_class\_\_是一样的结果（尽管这不能得到保证，因为新式类的实例的\_\_class\_\_方法是允许被用户覆盖的）。

#coding=utf-8

class A:

def test(slef):

print("A中的test")

class B(A):

def test(slef):

print("B中的test")

class C(B):

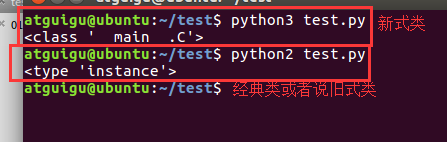
def test(slef):

print("C中的test")

c = C()

#C是大写是类名

print(type(c))



## 10.3. 新式类相同父类只执行一次构造函数，经典类重复执行多次

下面举个例子说明一下。有类A、B、C、D、E，B、C继承A，D继承B、C，E继承D、A。先从经典类说起，代码如下：

#coding=utf-8

class A:

def \_\_init\_\_(self):

print 'a',

class B(A):

def \_\_init\_\_(self):

A().\_\_init\_\_()

print 'b',

class C(A):

def \_\_init\_\_(self):

A().\_\_init\_\_()

print 'c',

class D(B,C):

def \_\_init\_\_(self):

B().\_\_init\_\_()

C().\_\_init\_\_()

print 'd',

class E(D,A):

def \_\_init\_\_(self):

D().\_\_init\_\_()

A().\_\_init\_\_()

print 'e',

d=D()

print '---'

e=E()

代码执行后打印如下：

a a b a a b a a c a a c d   
a a b a a b a a c a a c d a a b a a b a a c a a c d a a e



第一行应该按如下分组a a b 、a a b |a a c 、a a c| d。首先执行D的init函数，D的init包含B和C的init，都执行之后才会执行print d，至于为什么显示执行了两次构造函数，这个取决于类内部的call方法，之后介绍。

#coding=utf-8

class A(object):

def \_\_init\_\_(self):

print('a',)

class B(A):

def \_\_init\_\_(self):

super(B,self).\_\_init\_\_()

print('b',)

class C(A):

def \_\_init\_\_(self):

super(C,self).\_\_init\_\_()

print('c',)

class D(B,C):

def \_\_init\_\_(self):

super(D,self).\_\_init\_\_()

print('d',)

class E(D,A):

def \_\_init\_\_(self):

super(E,self).\_\_init\_\_()

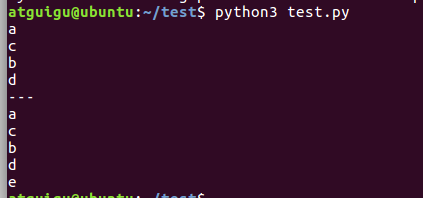
print('e',)

d=D()

print('---')

e=E()

结果打印如下:



可能有人会问，假如在子类里面不执行父类的init函数，不就不需要这么多次调用了么？但是，如果不执行父类init，怎么调用父类属性呢？一般而言调用父类的方法目的是初始化父类的参数，便于子类使用的时候，已经初始化好了。

## 10.4. 使用 \_\_bases\_\_属性，可以查看类上一层的父类是什么

