Class ：类

Object ：对象 #python3可忽略不写

Self ：自己

Init ：初始化

类的定义：

基本形式：

class ClassName(object): #(object),python3可忽略不写

1.class定义类的关键字

2.ClassName类名，类名的每个单词的首字母大写。

3.object是父类名，object是一切类的基类。

class Fruits(object): #创建Fruits(水果)这样一个类

Pass #占位

类的初始化： \_\_init\_\_

定义类时，这种方法可以使类的对象实例按某种特定的模式生产出来。

后面的参数中第一个参数我们约定俗成的为self参数名，self代表的是在类实例化后这个实例对象本身。

初始化函数除了有self这个参数表示实例对象本身之外，其他的参数的定义也遵循函数的必备参数和默认参数一样的原则，必备参数就是在实例化是一定要传入的参数，默认参数就是在定义时可以给这个参数一个初始值。

class Fruits(object):

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

self.name = name

self.color = color

self.weight = weight

类的实例化：

一、基本形式：

实例对象名 = 类名(参数)

在实例化的过程中，初始化中self代表的就是这个实例对象自己。 #即self = fr

fr = Fruits('apple','red',90)

二、实例化时会把类名后面接的参数传进去赋值给实例，这样传进去的参数就成为了这个实例对象的属性。 下面案例紫色部分。

#创建类

class Fruits(object):

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

self.name = name

self.color = color

self.weight = weight

#将类实例化

>>> fr = Fruits('apple','red',90)

>>> fr.name

'apple'

>>> fr.color

'red'

>>> fr.weight

90

三、实例化的过程遵循函数调用的原则：

在实例化时也必须个数和顺序与定义时相同（使用关键字参数可以改变传参的顺序）。

当初始化函数定义时使用了默认参数时，在实例化时默认参数可以不传参，这时这个实例对象就会使用默认的属性，如果传了参数进去则会改变这参数值，使实例化对象的属性就为你传进来的这个参数。

class Fruits(object):

def \_\_init\_\_(self,name,color,weight=90):

self.name = name

self.color = color

self.weight = weight

>>> fr = Fruits('apple','red')

>>> fr.weight

90

>>> fr = Fruits('apple','red',100)

>>> fr.weight

100

四、判断实例名是否在类里

isinstance(实例名，类的实例)

判断一个实例是不是这个类的实例。

>>> a = 12

>>> isinstance(a,int)

True

>>> b = 'python'

>>> isinstance(b,int)

False

类的属性和实例化属性：

类属性

1.类属性是可以直接通过“类名.属性名”来访问和修改。

2.类属性是这个类的所有实例对象所共有的属性，任意一个实例对象都可以访问并修改这个属性（私有隐藏除外）。

3.对类属性的修改，遵循基本数据类型的特性：列表可以直接修改，字符串不可以，所以当类属性是一个列表时，通过任意一个实例对象对其进行修改。但字符串类型的类属性不能通过实例对象对其进行修改。

class Fruits(object):

list1 = [] #这时list1是类的属性,他是共有的

def \_\_init\_\_(self,name,color,weight=90): #这里是实例化属性，是私有的

self.name = name

self.color = color

self.weight = weight

self.list2 = []

>>> fr1 = Fruits('apple','red')

>>> fr2 = Fruits('banana','yellow')

>>> fr1.list1.append('mmmmm')

>>> fr2.list2.append('aaaaaa')

>>> fr1.list1 #调用fr1.list1和fr2.list1里面的东西都是共有的

['mmmmm']

>>> fr1.list2

[]

>>> fr2.list1

['mmmmm']

>>> fr2.list2

['aaaaaa']

实例化属性

1.在属性前面加了self标识的属性为实例的属性。

2.在定义的时候用的self加属性名字的形式，在查看实例的属性时就是通过实例的名称.属性名来访问实例属性。

例如：fr1.color

方法属性

定义属性方法的内容是函数，函数的第一个参数是self，代表实例本身。

class Fruits(object):

kind = 'fruits' #类的属性

list1 = []

def \_\_init\_\_(self,name,color,weight=90):

self.name = name ┐

self.\_color = color } #实例属性，数据属性

self.\_\_weight = weight ┘

def show(self):

print ('fruits........') #方法属性

>>> fr1 = Fruits('banana','yellow') #先定义

>>> fr1.show() #再调用，方法属性要加()

Fruits........

>>> fr1.kind

'fruits'

一些说明：

1.数据属性会覆盖同名的方法属性。减少冲突，可以方法使用动词，数据属性使用名词。

2.数据属性可以被方法引用。

3.一般，方法第一个参数被命名为self,，这仅仅是一个约定，self没有特殊含义，程序员遵循这个约定。

4.查看类中的属性和实例属性可以调用\_\_dict\_\_方法返回属性组成的字典。

class Fruits(object): #定义类，类名首字母大写

def \_\_init\_\_(self,name,color,weight=90): #类的初始化函数，self代表实例本身

self.name = name #实例属性，数据属性

self.color = color

self.weight = weight

self.show = []

def show(self): #方法属性

print ('fruits........')

>>> fr1 = Fruits('banana','yellow')

>>> fr1.show #数据属性宇方法属性同名 数据属性覆盖方法属性

[]

>>> fr1.\_\_dict\_\_ #用\_\_dict\_\_方法返回属性组成的字典

{'name': 'banana', 'color': 'yellow', 'weight': 90, 'show': []}

数据隐藏和名字重整：

输入“实例名.”后不能按Tab查看出来，需用\_调用。

1、以一个下划线开头的命名（无论是函数，方法或数据成员），都会被隐藏起来。但直接将命名完整的写下还是一样可以访问到。

2、以两个下划线开头的命名，一样会被隐藏，只能在内部访问和修改，不能在外部访问，

因为它变成了“\_类名\_\_属性”的形式。在外部只能通过这种形式才能访问和修改。

1. 数据隐藏的作用就是确保外部代码不能随意修改对象内部的状态。(修改是可以修改，前提是先找到它在进行修改)

class Fruits(object):

def \_\_init\_\_(self,name,color,weight=90):

self.name = name

self.\_color = color

self.\_\_weight = weight

>>> fr1 = Fruits('banana','yellow')

>>> fr1.\_color

'yellow'

>>> fr1.\_Fruits\_\_weight

90

数据封装：

在类里面数据属性和行为用函数的形式封装起来，访问时直接调用，不需知道类里面具体的实现方法。

继承：

一、在定义类时，可以从已有的类继承，被继承的类称为基(父类)，新定义的类称为派生类(子类)。

class Fruits(object):

def \_\_init\_\_(self,name,color,weight=110):

self.name = name

self.color = color #父类(基类)

self.weight = weight

self.list2 = []

---------------------------------------------------------------------------------------------------------------------------------

class Apple(Fruits): #子类(派生类)

def \_\_init\_\_(self,color,weight=90): #self后面只有color一个必备函数所以实例化只

需要输入一个，name初始的时候可以给他一个值

所以这里不用写，直接写在下面

Fruits.\_\_init\_\_(self,'apple',color,weight) #这句绿色继承上面绿色

>>> a1 = Apple('red')

>>> a1.name #只继承了’apple’

'apple'

>>> a1.color #’color,weight’ 自己有不用继承

'red'

>>> a1.weight

90

二、在类中找不到调用的属性时就搜索基类，如果基类是从别的类派生而来，这个规则会递归的应用上去。反过来不行。如果派生类中的属性与基类属性重名，那么派生类的属性会覆盖掉基类的属性，包括初始化函数。

自己没有：

class Fruits(object): #父类(基类)

def \_\_init\_\_(self,name,color,weight=90):

self.name = name

self.color = color

self.weight = weight

self.list2 = []

def show(self):

print ('fruits........')

---------------------------------------------------------------------------------------------------------------------------------

class Apple(Fruits): #子类(派生类)

def \_\_init\_\_(self,color,weight=90):

Fruits.\_\_init\_\_(self,'apple',color,weight)

>>> a1 = Apple('red')

>>> a1.show() #子类没有show调用父类show

Fruits........

===============================================================================

自己有：

class Fruits(object): #父类(基类)

def \_\_init\_\_(self,name,color,weight=90):

self.name = name

self.color = color

self.weight = weight

self.list2 = []

def show(self):

print ('fruits........')

---------------------------------------------------------------------------------------------------------------------------------

class Apple(Fruits): #子类(派生类)

def \_\_init\_\_(self,color,weight=90):

Fruits.\_\_init\_\_(self,'apple',color,weight)

def show(self):

print ('Apple........')

>>> a1 = Apple('red') #子类有show调用自己的show

>>> a1.show()

Apple........

1. 派生类在初始化函数中需要继承和修改初始化过程，使用’类名+\_\_init\_\_(arg)’来实现继承和私有特性,也可以使用super()函数。 #与二用法一样只是用super内置函数

class Fruits(object): #父类(基类)

def \_\_init\_\_(self,name,color,weight=100):

self.name = name

self.color = color

self.weight = weight

self.list2 = []

def show(self):

print ('fruits........')

class Orange(Fruits):

def \_\_init\_\_(self,color,weight):

super(Orange,self).\_\_init\_\_('orange',color,weight)

>>> o1 = Orange('orange1',80) #这里是给上面(self,color,weight)赋值

>>> o1.name

'orange'

>>> o1.color

'orange1'

>>> o1.weight

80

>>> o1.list2

[]

>>> o1.show()

fruits........

四、issubclass(类名1，类名2)

判断类1是否继承了类2

多态：

多态是基于继承的一个好处。当派生类重写了基类的方法时就实现了多态性。

super 是用来解决多重继承问题的，直接用类名调用父类方法在使用单继承的时候没问题，但是如果使用多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等种种问题。

普通继承：

class FooParent(object):  
 def \_\_init\_\_(self):  
 self.parent = 'I\'m the parent.'  
 print 'Parent'  
  
 def bar(self, message):  
 print message, 'from Parent'  
  
  
class FooChild(FooParent):  
 def \_\_init\_\_(self):  
 FooParent.\_\_init\_\_(self)  
 print 'Child'  
  
 def bar(self, message):  
 FooParent.bar(self, message)  
 print 'Child bar function.'  
 print self.parent  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 a = FooParent()  
 print a.parent  
 a.bar('aaa')

Parent

I'm the parent.

aaa from Parent

b = FooChild()  
 b.bar('HelloWorld')

Parent

Child

HelloWorld from Parent

Child bar function.

I'm the parent.

Super继承：

class FooParent(object):  
 def \_\_init\_\_(self):  
 self.parent = 'I\'m the parent.'  
 print 'Parent'  
  
 def bar(self, message):  
 print message, 'from Parent'  
  
  
class FooChild(FooParent):  
 def \_\_init\_\_(self):  
 super(FooChild, self).\_\_init\_\_()  
 print 'Child'  
  
 def bar(self, message):  
 super(FooChild, self).bar(message)  
 print 'Child bar fuction'  
 print self.parent  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 a = FooParent()  
 print a.parent  
 a.bar('aaa')

Parent

I'm the parent.

aaa from Parent

b = FooChild()  
 b.bar('HelloWorld')

Parent

Child

HelloWorld from Parent

Child bar fuction

I'm the parent.

从运行结果上看，普通继承和super继承是一样的。但是其实它们的内部运行机制不一样，这一点在多重继承时体现得很明显。在super机制里可以保证公共父类仅被执行一次，至于执行的顺序，是按照mro进行的（E.\_\_mro\_\_）。