python 第三部分

class

定义

```
1  #encoding:utf-8
2
3  class Anjuke:
4      pass
```

```
1  #encoding:utf-8
2
3  class Anjuke:
4      print "This is anjuke"
```

```
1  #encoding:utf-8
2
3  class Anjuke:
4      departments = 1
5      def init(self):
6          self.departments += 1
```

公有属性

公有属性可以通过实例直接访问和修改

```
1   #encoding:utf-8
2
3   class Anjuke:
4       departments = ["user","mobile","salesmen"]
5       address = "东方路1217号"
6
7   if __name__ == '__main__':
8       a = Anjuke()
9       print a.departments
10      a.address = 123
11      print a.address
```

私有属性

私有属性不可以通过实例直接访问和修改
注意和公有属性在命名上的区别

```
1   #encoding:utf-8
2
3   class Anjuke:
4       __departments = ["user","mobile","salesmen"]
5       __address = "东方路1217号"
6
7   if __name__ == '__main__':
8       a = Anjuke()
9       print a.__departments
10      a.__address = 123
11      print a.__address
```

方法

```
1   #encoding:utf-8
2
3   class Anjuke:
4       departments = ["user","mobile","salesmen"]
5       address = "东方路1217号"
6
7       def print_address(self):
8           print self.address
9
10      def print_departments(self):
11          print self.departments
12
13  if __name__ == '__main__':
14      a = Anjuke()
15      a.print_address()
16      a.print_departments()
```

私有方法

```
1   #encoding:utf-8
2
3   class Anjuke:
4       __departments = ["user","mobile","salesmen"]
5       __address = "东方路1217号"
6
7       def __print_address(self):
8           print self.__address
9
10      def print_address(self):
11          self.__print_address()
12
13  if __name__ == '__main__':
14      a = Anjuke()
15      a.print_address()
```

纠结的self参数

方法总是将他们的第一个参数绑定到所属实例上

```
1   #encoding:utf-8
2
3   class Anjuke:
4       address = "东方路1217号"
5
6       def print_address(self):
7           print self.address
8
9   if __name__ == '__main__':
10      a = Anjuke()
11      b = Anjuke()
12      b.address = "13楼"
13      Anjuke.print_address(a)
14      Anjuke.print_address(b)
```

```
1   #encoding:utf-8
2
3   class Anjuke:
4       address = "东方路1217号"
5
6       def print_address(self):
7           print type(self),hex(id(self))
8
9   if __name__ == '__main__':
10      a = Anjuke()
11      b = Anjuke()
12      print 'a的地址是',hex(id(a))
13      print 'b的地址是',hex(id(b))
14      Anjuke.print_address(a)
15      Anjuke.print_address(b)
```

self的绑定并不依赖调用方式，请看如下例子

通过变量访问实例方法也是绑定到所属实例上

```
#encoding:utf-8

class Anjuke:
    address = "东方路1217号"

    def print_address(self):
        print type(self),hex(id(self))

if __name__ == '__main__':
    a = Anjuke()
    Anjuke.print_address(a)
    other = a.print_address
    other()
    print type(other)
```

常用魔法方法

- __init__()
- __del__()
- __getitem__()
- __setitem__()
- __len__()
- __getattr__()
- __setattr__()
- __delattr__()
- __iter__()
- __add__()

__init__()

```python
#encoding:utf-8

class Anjuke:
    address = "东方路1217号"
    depart = "用户端"
    def __init__(self,*args):
        if len(args) > 0:
            self.address = args[0]
            self.depart = args[1]

    def print_attr(self):
        print self.address
        print self.depart

if __name__ == '__main__':
    a = Anjuke()
    a.print_attr()
    b = Anjuke("陆家嘴","网站端")
    b.print_attr()
```

类实例化时自动调用

__del__()

```python
#encoding:utf-8

class Anjuke:
    address = "东方路1217号"
    depart = "用户端"
    def __init__(self,*args):
        if len(args) > 0:
            self.address = args[0]
            self.depart = args[1]

    def __del__(self):
        print "我被回收了"

    def print_attr(self):
        print self.address
        print self.depart

if __name__ == '__main__':
    a = Anjuke()
    a.print_attr()
    #del a
    b = Anjuke()
    a.address = "陆家嘴"
    b.print_attr()
```

实例被销毁时调用

python的垃圾回收机制是计算引用来进行的，把21行的注释去掉再看看效果

__getitem__()

```
1   #encoding:utf-8
2
3   class Anjuke:
4       l = []
5       def __init__(self,*args):
6           for i in args:
7               self.l.append(i)
8
9       def __getitem__(self,key):
10          return self.l[key]
11
12  if __name__ == '__main__':
13      a = Anjuke('a','b','c')
14      print a[2]
```

__setitem__()

```python
#encoding:utf-8

class Anjuke:
    l = []
    def __init__(self,*args):
        for i in args:
            self.l.append(i)

    def __getitem__(self,key):
        return self.l[key]

    def __setitem__(self,key,value):
        if key > len(self.l)-1:
            self.l.append(value)
        else:
            self.l[key] = value

if __name__ == '__main__':
    a = Anjuke('a','b','c')
    a[3] = 'string'
    a[1] = 'int'
    print a.l
```

getitem和setitem配合使用就可以使任何一个类模拟序列操作,既然是序列肯定可以应用到迭代上,下一节将演示__iter__()的用法

__iter__()

```python
#encoding:utf-8
class Anjuke:
    l = []
    count = -1
    def __init__(self,*args):
        for i in args:
            self.l.append(i)
    def __getitem__(self,key):
        return self.l[key]
    def __setitem__(self,key,value):
        if key > len(self.l)-1:
            self.l.append(value)
        else:
            self.l[key] = value
    def __iter__(self):
        return self
    def next(self):
        if Anjuke.count <= len(self.l):
            Anjuke.count += 1
            return self.l[Anjuke.count]
    def __len__(self):
        return len(self.l)
if __name__ == '__main__':
    a = Anjuke('a','b','c')
    a[3] = 'string'
    for i in a:
        if i != 'string':
            print i
        else:
            break
    print len(a)
```

这里有两个值得注意的地方

1.__len__()方法

2.使用类.属性名的访问方式

__getattr__()

```
1   #encoding:utf-8
2
3   class Anjuke:
4
5       def __init__(self):
6           self.address = "东方路1217号"
7
8       def __getattr__(self,name):
9           print name,"不存在"
10
11  if __name__ == '__main__':
12      a = Anjuke()
13      print a.address
14      a.depart
```

此方法在访问不存在的属性时调用

__setattr__()

```
1   #encoding:utf-8
2
3   class Anjuke:
4
5       def __init__(self):
6           self.address = "东方路1217号"
7
8       def __getattr__(self,name):
9           print name,"不存在"
10
11      def __setattr__(self,name,value):
12          self.__dict__[name] = value
13
14  if __name__ == '__main__':
15      a = Anjuke()
16      a.depart = 'user'
17      print a.__dict__
```

__add__()

```python
#encoding:utf-8

class Anjuke:

    def __init__(self,num):
        self.people = num

    def __add__(self,other):
        return self.people+other.people

if __name__ == '__main__':
    a = Anjuke(5)
    b = Anjuke(10)
    print a+b
```

此方法是许多运算方法中的一种，其余的还有很多:__eq__(),__abs__(),__sub__() .etc

静态方法和类方法

```python
#encoding:utf-8

class Anjuke:

    address = "东方路1217号"

    def __init__(self):
        self.depart = "哈哈"

    @staticmethod
    def print_address():
        print Anjuke.address

    @staticmethod
    def print_depart():
        print self.depart

if __name__ == '__main__':
    a = Anjuke()
    a.print_address()
    Anjuke.print_address()
```

```python
#encoding:utf-8

class Anjuke:

    address = "东方路1217号"

    def __init__(self):
        self.depart = "哈哈"

    @classmethod
    def print_depart(cls):
        print cls.depart

    @classmethod
    def print_address(cls):
        print cls.address

if __name__ == '__main__':
    Anjuke.print_address()
```

面向对象三大特性

- 封装
- 继承
- 多态

封装

```python
#encoding:utf-8

class Anjuke(object):

    def __init__(self):
        self.__depart = "user"

    def get_depart(self):
        if self.__depart == 'user':
            return self.__depart
        else:
            return '不是用户端，不输出'

    def set_depart(self,attr):
        self.__depart = attr

    depart = property(get_depart,set_depart)

if __name__ == '__main__':
    a = Anjuke()
    print a.depart
    a.depart = "mobile"
    print a.depart
```

```python
#encoding:utf-8

class Anjuke(object):

    def __init__(self):
        self.__depart = "user"

    @property
    def depart(self):
        return self.__depart

    @depart.setter
    def depart(self, value):
        self.__depart = value

    @depart.getter
    def depart(self):
        if self.__depart == 'user':
            return self.__depart
        else:
            return '不是用户端，不输出'
    @depart.deleter
    def depart(self):
        print '不能删除'

if __name__ == '__main__':
    a = Anjuke()
    a.depart = 'mobile'
    print a.depart
    del a.depart
```

继承

```python
#encoding:utf-8

class Person:
    def __init__(self):
        self.age = 20
        self.name = 'person'

    def say(self):
        print "I am %s" % self.age

class Anjuke(Person):

    def __init__(self):
        Person.__init__(self)

class AnjukeB(Person):

    def __init__(self):
        Person.__init__(self)
        self.name = 'AnjukeB'

    def say(self):
        print self.name

if __name__ == '__main__':
    a = Anjuke()
    print 'my name is',a.name
    a.say()
    b = AnjukeB()
    b.say()
```

```python
#encoding:utf-8
__metaclass__ = type
class Person:
    def __init__(self):
        self.age = 20
        self.name = 'person'

    def say(self):
        print "I am %s" % self.age

class Anjuke(Person):

    def __init__(self):
        super(Anjuke,self).__init__()

class AnjukeB(Person):

    def __init__(self):
        Person.__init__(self)
        self.name = 'AnjukeB'

    def say(self):
        print self.name

if __name__ == '__main__':
    a = Anjuke()
    print 'my name is',a.name
    a.say()
    b = AnjukeB()
    b.say()
```

新式类，注意第2行和第14行

多重继承

```python
#encoding:utf-8
__metaclass__ = type

class Person:
    def __init__(self):
        self.age = 20
        self.name = 'person'

    def say(self):
        print "I am %s" % self.age

class Employee:

    def __init__(self):
        self.name = 'Employee'

    def get_salary(self):
        print "I want salary"

    def say(self):
        print "I am Employee"

class Anjuke(Person,Employee):
    def __init__(self):
        super(Anjuke,self).__init__()


if __name__ == '__main__':
    a = Anjuke()
    print a.name
    a.say()
```

注意父类的顺序，比较代码的不同之处

```python
#encoding:utf-8
__metaclass__ = type

class Person:
    def __init__(self):
        self.age = 20
        self.name = 'person'

    def say(self):
        print "I am %s" % self.age

class Employee:

    def __init__(self):
        self.name = 'Employee'

    def get_salary(self):
        print "I want salary"

    def say(self):
        print "I am Employee"

class Anjuke(Employee,Person):
    def __init__(self):
        super(Anjuke,self).__init__()


if __name__ == '__main__':
    a = Anjuke()
    print a.name
    a.say()
```

多态

```python
#encoding:utf-8
__metaclass__ = type

class Anjuke:

    def count(self,arg):
        return arg.count('a')

if __name__ == '__main__':
    a = Anjuke()
    print a.count("abcdefa")
    print a.count(['a','b'])
```

python是动态语言，自身就带有多态特性，最直接的例子就是'+'运算符，可以作用于字符串，数值甚至是类实例上

一个显而易见的多态例子

```python
#encoding:utf-8
__metaclass__ = type

class A:
    def say(self):
        print "I am A"

class B(A):
    pass

class C:
    def say(self):
        print "I am C"

class Anjuke:

    def mul(self,obj):
        obj.say()

if __name__ == '__main__':
    an = Anjuke()
    an.mul(A())
    an.mul(B())
    an.mul(C())
```

```python
#encoding:utf-8
__metaclass__ = type

class User:
    def say(self):
        print "I am User"

class Web:
    def say(self):
        print "I am Web"

class Mobile:
    def say(self):
        print "I am Mobile"

class AnjukeFactory:

    def produce(self,type):
        if type == 'User':
            return User()
        elif type == 'Web':
            return Web()
        else:
            return Mobile()

if __name__ == '__main__':
    an = AnjukeFactory()
    an.produce('User').say()
    an.produce('Web').say()
    an.produce('Mobile').say()
```