class :

- (类)Animal -((类)Cat, (类)Dog,) - （实例）fentiao tom harry
- (类)zhiwu -

定义一个类 ¶

In [ ]:

```
# def 函数名(形参):
#     函数执行的内容

# 类的格式
# class 类名(父类):
#     类的内容
```

In [1]:

```
# 类名后面有括号的类，称为新式类；
# 括号里面的内容是父类的名称；程序中，所有类的父类都是object;
class Animals(object):
    pass

print Animals
```

```
<class '__main__.Animals'>
```

类的数据属性

In [2]:

```
class Animals(object):
    # 类的数据属性
    name = "fentiao"
    age = 12

print Animals

# 访问类的数据属性
print Animals.name
print Animals.age
```

```
<class '__main__.Animals'>
fentiao
```

类的方法

In [10]:

```
class Animals(object):
    name = "fentiao"
    age = 12
    weight = 10

    # 类的方法====函数
    # 在类中定义的函数叫做方法;
    # 类的方法中，python解释器要求第一个形参必须是self；与java中的this类似；
    # self实质上是类实例化后的对象本身;
    def eat(self):
        print "eating......"
        print self

# 类的实例化产生的就是对象； 把抽象的类创造出实际存在的事物；
# object: 对象
fentiao = Animals()
print fentiao

# 调用类的方法
fentiao.eat()
```

```
<__main__.Animals object at 0x2c2f490>
eating......
<__main__.Animals object at 0x2c2f490>
```

面向对象的三大特性： 封装， 继承， 多态

封装

封装实际上是把数据封装到某个地方， 以后再去调用被封装在某处的内容或者数据;

- 封装数据
- 调用封装数据
    - 通过对象直接调用;
    - 通过self间接调用

In [21]:

```
class Animals(object):

    # 构造方法
    # 当类实例化时会自动调用__init__构造方法;
    # name, age, weight是必选参数，当实例化是必须要传参，否则报错;
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    # eat方法
    def eat(self):
        print "%s eating......" %(self.name)
        self.weight += 2

     # drink方法
    def drink(self):
        print "%s is drinking....." %(self.name)
        self.weight += 1




# 对象可以实例化多个；
fentiao = Animals("fentiao", 5, 12)
tom = Animals("tom", 3, 5)
# self实质上就是类的实例化对象
print fentiao.name
print fentiao.age
print fentiao.weight
fentiao.eat()
print fentiao.weight
fentiao.drink()
print fentiao.weight


tom.drink()
print tom.weight
```

```
fentiao
5
12
fentiao eating......
14
fentiao is drinking.....
15
tom is drinking.....
6
```

应用练习1

创建一个类People，拥有的方法为砍柴,娶媳妇，回家；

- 实例化对象，执行相应的方法
- 显示如下:

  老李，18岁，男，开车去娶媳妇
  校思浩，22岁，男，上山去砍柴
  唐浩，10岁，女，辍学回家

- 提示:
  - 属性:name,age,gender
  - 方法：goHome(), kanChai(),quXiFu()

In [ ]:

```
#! /usr/bin/env python
# coding:utf-8

class People():
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def huiJia(self):
        print "%s,%d,%s,辍学回家" %(self.name,self.age,self.gender)
    def quXiFu(self):
        print "%s,%d,%s,开车去娶媳妇" %(self.name,self.age,self.gender)
    def kanChai(self):
        print "%s,%d,%s,上山砍柴" %(self.name,self.age,self.gender)

Laoli = People('laoli',43,'男')
Laoli.quXiFu()
zhangsan = People('zhangsan',22,'女')
zhangsan.huiJia()
lisi = People('lisi',11,'男')
lisi.kanChai()
```

应用练习2：栈的数据结构

class Stack:

栈的方法:

- 入栈(push)， 出栈（pop）, 栈顶元素(top),
- 栈的长度(lenght), 判断栈是否为空(isempty)
- 显示栈元素(view)

操作结果:

- 栈类的实例化
- 入栈2次
- 出栈1次
- 显示最终栈元素

In [ ]:

```
# coding: utf-8
class Stack(object):
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if not self.isempty():
            return self.stack.pop()
        else:
            return None

    def top(self):
        if not self.isempty():
            return self.stack[-1]
        else:
            return None

    def length(self):
        return len(self.stack)

    def view(self):
        for i in self.stack:
            print i,

    def isempty(self):
        # return self.stack # []的bool值为False； [1,2,3]的bool值为True
        return self.stack == []
```

应用实例3： 队列的数据结构

class Queue:

队列的方法:

- 入队(enqueue)， 出队（dequeue）, 队头元素(head), 队尾元素(tail),
- 队列的长度(lenght), 判断队列是否为空(isempty)
- 显示队列元素(view)

操作结果:

- 队列类的实例化
- 入队5次
- 出栈1次
- 显示最终队列元素

In [ ]:

```

```

继承

- 父类和子类; 基类和派生类;

- 注意: 类的属性名和方法名不同相同;
- 建议:
  - 属性名用名词；eg:name, age, weight;
  - 方法名建议用动词; eg: eat, drink, get_weight;

In [24]:

```python



# Animals是父类/基类;
class Animals(object):
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 2

    def drink(self):
        print "%s is drinking....." % (self.name)
        self.weight += 1


    def get_weight(self):
        pass

# Dog是Animal的子类/派生类;
class Dog(Animals):
    # 类里面的方法第一个参数必须是self
    def jiao(self):
        print "%s wang wang ......." % (self.name)

# Cat是Animal的子类/派生类;
class Cat(Animals):
    def jiao(self):
        print "%s miao miao miao........" % (self.name)



tom = Dog("tom", 12, 10)
tom.eat()
tom.jiao()
```

```
tom eating......
tom wang wang .......
```

重写父类的构造函数

- 父类名.__init__(self,形参)

- super(自己类的名称, self).__init__(形参)
  - 不需要明确告诉父类的名称；
  - 如果父类改变，只需修改class语句后面的继承关系即可;

In [38]:

```

# Animals是父类/基类;
class Animals(object):
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 2


# Dog是Animal的子类/派生类;
class Dog(Animals):   # name, age, weight, dogid
    def __init__(self, name, age, weight, dogid):
        # 第一种重写父类构造方法;
        #self.name = name
        #self.age = age
        #self.weight = weight

        # 第二种重写父类构造方法:
        # 让Dog的父类执行它的__init__方法;
        #Animals.__init__(self, name, age, weight)

        # 第三种重写父类构造函数的方法;
        super(Dog, self).__init__(name, age, weight)

        self.dogid = dogid

# Cat是Animal的子类/派生类;
class Cat(Animals):   # name, age, weight, food
    def __init__(self, name, age, weight, food):

#         self.name = name
#         self.age = age
#         self.weight = weight
#         Animals.__init__(self, name, age, weight)
        super(Cat, self).__init__(name, age, weight)
        self.food =  food

tom = Dog("tom", 3, 10, '001')
print tom.dogid


harry = Cat("harry", 2, 5, "fish")
print harry.food
# print harry.dogid
```

```
001
fish
```

新式类和经典类

- python2.x里面支持经典类和新式类；
- python3.x里面仅支持新式类；

In [41]:

```
# - 经典类
class Book1:
    pass


# - 新式类
class Book2(object):
    pass



class Book3:
    pass


b = Book3()
print b
```

```
<__main__.Book3 instance at 0x2ce5a28>
```

多重继承

In [ ]:

```
# 对于新式类来说， 多重继承的算法是广度优先;
class D(object):
    def test(self):
        print "D test"


class C(D):
    pass
    # def test(self):
    #     print  "C test"

class B(D):
    pass
    # def test(self):
    #     print "B test"

# A继承B和C；
class A(B,C):
    pass
    # def test(self):
    #     print "A test"

# A - B  - D
#   - C  - D

a = A()
a.test()
```

In [ ]:

```
# 对于经典类来说， 多重继承的算法是深度优先;
class D:
    pass
    # def test(self):
    #     print "D test"


class C(D):
    # pass
    def test(self):
        print  "C test"

class B(D):
    pass
    # def test(self):
    #     print "B test"

# A继承B和C ;
class A(B,C):
    pass
    # def test(self):
    #     print "A test"

# A - B  - D
#   - C  - D

a = A()
a.test()
```

多态

- 当父类和子类有相同的方法时， 调用优先执行子类的方法;

In [43]:

```
# Animals是父类/基类;
class Animals(object):
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 2


class Cat(Animals):
    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 1


class Dog(Animals):
    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 3


tom = Dog("tom", 12, 12)
tom.eat()
print tom.weight
```

```
tom eating......
15
```

特殊的类属性

In [51]:

```
class Base(object):
    pass

class Animals(object):
    """
    父类Animals：

    Attritube:
        name:
        age:
        weight:

    """
    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight = weight

    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 2


class Cat(Animals):
    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 1


class Dog(Animals, Base):
    def eat(self):
        print "%s eating......" % (self.name)
        self.weight += 3



# print Animals.__name__
# print Animals.__doc__

# # 打印类的所有父类,以元组类型返回;
# print Animals.__bases__
# print Dog.__bases__


# 以字典的方式返回类的方法和属性;
# print Animals.__dict__


# 如果类不是被导入的， 显示为__main__;
# 如果类是被import导入的， 则显示类所在的模块名
print Animals.__module__
```

```
__main__
```

类属性

In [54]:

```
class Info(object):
    a = 1
    # 类属性，在内存中只存一份；
    country="china"

    def __init__(self, name):
        # 构造函数里面的属性存的份数取决于你的对象个数;
        self.name = name


p1 = Info("p1")
print p1.country

p2 = Info("p2")
print p2.country


# 后面有1W条用户信息录入
```

```
china
china
```

实现查看类实例化对象的个数

In [62]:

```
class Info(object):
    count = 0

    # 构造函数
    def __init__(self):
        Info.count += 1



    # 析构函数, 在删除对象时自动调用;
    def __del__(self):
        Info.count -= 1


a = Info()
b = Info()
c = Info()
print Info.count


del a

print  Info.count
```

```
3
2
```

类方法和静态方法

In [ ]:

```
class Info(object):
    # 普通方法，第一个参数必须是self(对象);
    def eat(self):
        print self

    # 类方法，第一个参数是cls(类);
    @classmethod
    # cls是class的缩写；
    def drink(cls):
        print cls

    # 静态方法，不需要加特殊的第一个参数;
    @staticmethod
    def run():
        print "run"


a = Info()
# a.eat()
a.drink()
```

类方法和静态方法的应用

In [63]:

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def echo_date(self):
        print """
        Year:%s
        Month:%s
        Day:%s
        """%(self.year, self.month, self.day)

    @classmethod
    def str_date(cls, s):
        year, month, day = map(int, s.split('-'))
        d = cls(year, month, day)
        return d

    # s='2018-1-4'
    @staticmethod
    def is_date_legal(s):
        year, month, day = map(int, s.split('-'))
        return year > 0 and 0 < month <= 12 and 0 < day <= 31




d = Date.str_date('2018-1-4')
d.echo_date()

print "legal" if Date.is_date_legal("2018-1-4") else "illegal"
```

```
        Year:2018
        Month:1
        Day:4

legal
```

属性property

In [80]:

```python

class Price(object):
    def __init__(self, old_price, discount):
        self.old_price = old_price
        self.discount = discount

    @property
    def price(self):
        new_price = self.old_price * self.discount
        return new_price

    @price.setter
    def price(self, value):
        self.old_price = value

    @price.deleter
    def price(self):
        print "%d is delete...." % (self.old_price)
        del self.old_price

    def __del__(self):
        print "deleteing....."


p = Price(100, 0.8)
print p.price
p.price = 200
print p.price
del p.price
# print p.price
```

```
80.0
160.0
200 is delete....
```

特殊的实例方法

- **call**当调用对象时自动执行;
- **str**