类代码编写基础

类对象和实例对象

类对象提供默认的行为 是实例对象的工厂 来自于语句 class …

实例对象是程序处理的实际对象 各自都有独立的命名空间 来自于调用 x = c1()

Python 类的主要特点

1. class 语句创建类对象并将其赋值给变量的名字

2.class 内中的赋值语句创建了类的属性

3.类属性提供了对象的状态和行为

Python 类实例的特点

1.像函数那样调用类对象会创建新的实例对象

2.每个实例对象继承了类的属性并获取自己的命名空间

3.在方法内对 self 属性做赋值运算会产生每个实例自己的属性

在类的方法函数内 第一个参数 self 会引用正处理的实例对象 对 self 的属性做赋值运算 会创建或修改实例内的数据而不是类的数据

第一个例子

```
class FirstClass:
    def setName(self, value):
```

```
        self.name = value
    def getName(self):
        print self.name
    def display(self):
        print self.name


x = c1()
y= c1()
x.setName('togogo')
x.getName()
y.setName('cisco')
y.getName()
```

第二个例子(继承)

除了作为工厂来生成多个实例对象之外　类也可以引入子类来进行扩展，而不对当前的父类进行修改 这就是 Python 类的层次结构 在阶层较低的地方覆盖现有的属性 让行为特定化 实际上向层次的下端越深入 软件就会变的越特定

例:

```
class SecondClass(FirstClass):
    def display(self):
```

```
        print 'currnet value = %s' %self.name


z = SecondClass()
z.setName('yeslab_python')
z.display()
```

在上面的例子中 SecondClass 创建了其实例对象， setdata 依然是执行 FirstClass 中的版本 但是 display 使用的是 SecondClass 中的版本，两个类中的 display 方法打印的内容不相同，可见我们不是修改 FirstClass 而是对它进行了定制

在模块中导入类

```
[root@yeslab python]# cat demo11.py     ##类也是模块内的属性
from demo10 import FirstClass
x = FirstClass()
x.setName('demo11')
x.getName()
```

运算符重载初步

它可以让类截获并响应在内置类型上的运算 如数学运算 切片 打印和点号运算

1.以双下划线命名的方法 __x__， 这种特殊的命名方法 用来拦截运算 每种运算和特殊的命名方法之间存在一个固定不变的映射关系

2.当实例出现内置运算时（如 + -）这种类方法会自动调用

如果实例对象继承了一个__add__方法，当实例使用运算符+时，这个__add__方法将会被调用，这个方法的返回值就变成了相应表达式的结果

3.运算符重载几乎可以截获并实现内置类型的所有运算

4.最常用的运算符重载方法是 __init__

第三个例子(运算符重载)

```
class ThirdClass(SecondClass):
    def __init__(self, value):
        self.name=value
    def __add__(self, other):
        print self.name + other
    def __str__(self):
        return self.name

x = ThirdClass('yeslab')
x.display()
x + 'cisco'
```

类的命名空间

```
#-*-coding:utf8-*-
#Python 实例有自己的__dict__ 它对应的类也有自己的__dict__
#打印类的__dict__属性时 列出了所包含的属性 包括一些类内置属性和类变量


class cls:
    clsvar = 1
    def __init__(self):
        self.invar = 2
ins1 = cls()
ins2 = cls()

print cls.__dict__
print ins1.__dict__

"""
{'clsvar': 1, '__module__': '__main__', '__doc__': None, '__init__': <function __init__ at 0x10d8cc1b8>}
在类的__dict__中列出了类 cls 所包含的属性 包括一些内置属性 类变量 以及构造方法


{'invar': 2} 实例变量包含在对象 ins1 的__dict__属性中一个对象的属性查找
遵循先找实例对象自己 然后是类 然后是父类
"""

print '#' * 10
ins1.clsvar = 20
print cls.clsvar  #1
print ins1.clsvar  #20
print ins2.clsvar #1
print cls.__dict__
print ins1.__dict__

"""
{'clsvar': 1, '__module__': '__main__', '__doc__': None, '__init__': <function __init__ at 0x10d8cc1b8>}
```

```
{'invar': 2, 'clsvar': 20}
"""


print '#' * 20
cls.clsvar = 10

print cls.clsvar
print ins1.clsvar
print ins2.clsvar
print cls.__dict__
print ins1.__dict__

"""
{'clsvar': 10, '__module__': '__main__', '__doc__': None, '__init__': <function __init__ at 0x10d8cc1b8>}
{'invar': 2, 'clsvar': 20}
"""

print '#' * 10
ins1.x = 11
print ins1.x
print ins1.__dict__ #{'invar': 2, 'x': 11, 'clsvar': 20}
print '#' * 10

cls.m = 21
print cls.m
print cls.__dict__ #{'clsvar': 10, '__module__': '__main__', 'm': 21, '__doc__': None, '__init__': <function __init__
at 0x1101b41b8>}
print '#' * 10

ins3 = cls()
print ins3.__dict__  #{'invar': 2}
print ins3.invar  #2
print ins3.clsvar # 10
print ins3.m  # 21

#print ins3.x  #没有 x 属性



#对类的__init__进行批量赋值,避免去写类似这样的代码


"""
```

```
class c1:
    def __init__(self, dict):
        self.a = dict.get('a', 0)
        self.b = dict.get('b', 0)
        self.c = dict.get('c', 0)

"""



print '##' * 10

x = {'a': 'apple', 'b': 'banana'}

class c1():
    def __init__(self, directory):
        self.__dict__.update(directory)

i1 = c1(x)

print i1.__dict__
print i1.a
print i1.b

#print i1.c  #报错因为没有 C 这个类属性
```