在本节中我们构建一组类来做一些具体的事情 将会创建两个类 Person Manager

1. 创建一个 Person 类，并编写一个构造函数

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name=name
        self.job = job
        self.pay = pay
```

2.添加行为方法

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name=name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        return self.pay * (1 + percent)

if __name == '__main__':
```

```
    x = Person(name='liu shuo', pay = 100)
    print x.lastName()
    print x.giveRaise(0.2)
```

3 运算符重载

在类中编写一个运算符重载的方法，这个方法在实例上运行的时候 方法截获并处理内置操作

打印操作 运算符重载 __str__ （可能是第二常用的运算符重载方法）

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name=name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = self.pay * (1 + percent)
    def __str__(self):
        return '[Person %s %s]' %(self.name, self.pay)

x = Person(name='liu shuo', pay = 100)
print x.lastName()
```

print x.giveRaise(0.2)

print x

4.通过子类来定制行为

目标 定义一个子类 manager 当其实例要涨工资的时候 默认加多 10%的额外奖金 因为 Manager 类相对于类树处于底层，所以他可以覆盖父类中的方法

写法有两种

class Manager(Person):

    def giveRaise(self, percent, other=0.1):

        self.pay = self.pay * (1 + percent + other) ##不好的写法

不好的原因： 任何时候当你复制代码的时候， 几乎都会使未来的维护工作倍增，假如用这个方式如果一旦改变了工资增加的方式(如不再是 1+percent)，此时我们将修改两个地方而不是一个地方的代码

    def giveRaise(self, percent, other=0.1):

        Person.giveRaise(self, percent + other) ##好的写法

bob = Person('bob smitch')

sue = Person('sue', job='dev', pay=10000)

tom= Manager('tom', 'mgr', 50000)

```
for j in [bob, sue, tom]:
    j.giveRaise(0.1)
    print j
```

结果

[Person bob smitch 0.0]

[Person sue 11000.0]

[Person tom 60000.0]

多态的作用

bob sue tom 对象是一个 Person 或 Manager, Python 自动运行相应的 giveRaise 针对 bob sue 使用的是 Person 中的版本 针对 TOM 使用的是 Manager 中的版本，这称之为多态， 根据所传递的对象的类型将会自动运行相应的版本

通过继承扩展父类

在子类中可以添加父类中没有的方法

```
class Manager(Person):
#     def giveRaise(self, percent, other=0.1):
#         return self.pay * (1 + percent + other)
```

```
    def giveRaise(self, percent, other=0.1):
        Person.giveRaise(self, percent + other)
    def doSomething(self):
        return   'func dosomthing'
```

如果不使用继承会怎样?

1. 我们可以从头编写一个 Manager 类，它是全新并且独立的代码 但是这样会导致代码的冗余 增加未来对代码维护的工作

2. 可以在原处修改 Person 类变成 Manager，但是这样可能会使原来需要 Person 行为的地方无法满足需求

5 定制构造函数

示例代码:

父类 A

```
class A(object):
    def __init__(self, name):
        self.name=name
        print "name:", self.name
    def getName(self):
        return 'A ' + self.name
```

子类不重写__init__，实例化子类时，会自动调用父类定义的__init__

```
```

```
class B(A):
    def getName(self):
        return 'B '+self.name

if __name__=='__main__':
    b=B('hello')
    print b.getName()
```

执行

```
$python lei2.py
name: hello
B hello
```

但重写了__init__时，实例化子类，就不会调用父类已经定义的__init__

```
class A(object):
    def __init__(self, name):
        self.name=name
        print "name:", self.name
    def getName(self):
        return 'A ' + self.name

class B(A):
    def __init__(self, name):
        print "hi"
        self.name =  name
    def getName(self):
        return 'B '+self.name

if __name__=='__main__':
    b=B('hello')
    print b.getName()
```

执行

```
$python lei2.py
hi
B hello
```

结论

1.子类不重写__init__，实例化子类时，会自动调用父类定义的__init__

2.但重写了__init__时，实例化子类，就不会调用父类已经定义的__init__

3.为了能使用或扩展父类的行为，最好显示调用父类的__init__方法

在创建 Manager 类的对象的时候，必须为它提供一个 mgr 的工作名称，这看起来是不必要的，我们可以通过重写构造函数的方式自动填入这个值

第一种方式

```
class Manager(Person):
    def __init__(self, name, pay, age):  ##子类的构造函数相对于父类进行了重写
        self.age = age
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, other=0.1):
        Person.giveRaise(self, percent + other)
    def doSomething(self):
        return  'func dosomthing'
    def __str__(self):
        return '%s|%s|%s|%s' %(self.name, self.age, self.job, self.pay)
```

注意： 通过 Person.__init__ 来重定义构造函数，如果在构造的时候要运行更高的__init__方法，需要通过这种参数传递给超类的构造函数的方式

第二种方式

```
class c3(c1):
    def __init__(self, name, age):
        super(c3, self).__init__(name=name)
        self.age = age
    def display(self):
        print 'c3 myname is %s and age %s' %(self.name, self.age)
```