高级程序设计
---Python与深度学习
7. 类与对象
7. Class & OOP

李冰
副研究员
交叉科学研究院

类和面向对象

- 类
  - 定义
  - 实例化
  - 继承
- 类的特殊方法
- 类的运算符重载
- 编写一个类
  - 定义
  - 子类
  - 测试工具

身边的例子

- 硕士研究生
  - 硕士研究生考试成绩、本科成绩、本科所获得的奖励（属性）
  - 能做什么：写代码、公式推导、看论文、写论文、查资料 （方法）

身边的例子

- 硕士研究生
  - 硕士研究生考试成绩、本科成绩、本科所获得的奖励（属性）
  - 能做什么：写代码、公式推导、看论文、写论文、查资料 （方法）
- 孩子
  - 出生年月、父母、出生地（属性）
  - 哭、排泄、吃奶、睡觉、交流（方法）

身边的例子

- 硕士研究生
  - 硕士研究生考试成绩、本科成绩、本科所获得的奖励（属性）
  - 能做什么：写代码、公式推导、看论文、写论文、查资料 （方法）
- 孩子
  - 出生年月、父母、出生地（属性）
  - 哭、排泄、吃奶、睡觉、交流（方法）
- 在工业制造上，使用模具来铸造机壳和零件。
- 芯片制造中，我们根据版图来制造芯片，芯片完成很多功能。

类

- **类（class）：** 面向对象编程（Object Oriented Program, OOP）编程语言中一个重要概念。

类

- **类（class）：** 面向对象编程（Object Oriented Program, OOP）编程语言中一个重要概念。
  - 类是对象的蓝图，是具有相同属性和方法的对象的抽象。

类

- **类（class）：**面向对象编程（Object Oriented Program, OOP）编程语言中一个重要概念。
  - 类是对象的模板，是具有相同属性和方法的对象的抽象。
  - 对象只是数据（变量）和作用于这些数据的方法（函数）的集合。
    - 代码中，用变量表示数据，函数表示方法
    - “可变对象”，“不可变对象”
    - 对象（object，也被称为实例，instance）

类：定义语法

- 如何定义类？
- **class** 类名：
  执行语句...
  零个到多个**类变量**...
  零个到多个**方法**...

def

类：定义语法

- `class` 类名：
  执行语句...
  零个到多个类变量...
  零个到多个方法...

```
class Empty():
    pass
```

- Python 类所包含的最重要的两个成员就是变量和方法，
  - 类变量属于类本身，用于定义该类本身所包含的状态数据；
  - 实例变量则属于该类的对象，用于定义对象所包含的状态数据；
  - 方法则用于定义该类的对象的行为或功能实现

类：定义的例子

```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

类：定义的例子

类名

```
class FirstClass:          # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

类：定义的例子

```
class FirstClass:          # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance
    def display(self):
        print(self.data)         # self.data: per instance
```

方法

方法的第一个参数是self：调用自己的对象

类：定义的例子

```
class FirstClass:          # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

属性

类：实例化

```
# Make two instances, each is a new namespace
x = FirstClass()
y = FirstClass()
print(type(x))
print(x)
print(y)
```

类：实例化

```
# Make two instances, each is a new namespace
x = FirstClass()
y = FirstClass()
print(type(x))
print(x)
print(y)
```

!!括号

类：实例化

```
# Make two instances, each is a new namespace
x = FirstClass()
y = FirstClass()
print(type(x))
print(x)
print(y)
```

```
<class '__main__.FirstClass'>
<__main__.FirstClass object at 0x000002011AB975B0>
<__main__.FirstClass object at 0x000002011AB97940>
```

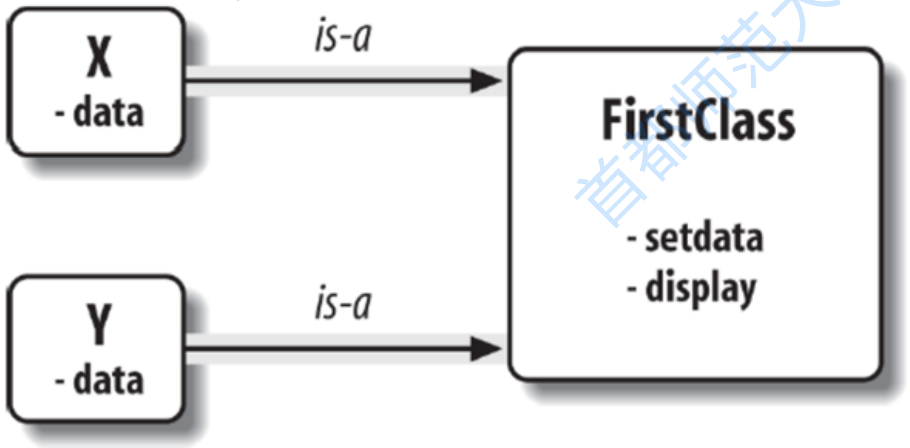
Python 为每个实例对象分配独立的内存空间。

（'__main__' 是顶层代码执行的作用域的名称）

类：实例化

```
# Make two instances, each is a new namespace
x = FirstClass()
y = FirstClass()
print(type(x))
print(x)
print(y)
```

```
<class '__main__.FirstClass'>
<__main__.FirstClass object at 0x000002011AB975B0>
<__main__.FirstClass object at 0x000002011AB97940>
```



- "data" 属性会在实例内找到
- "setdata" 和 "display" 是在类中找到。

类：实例化

```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

- 对实例以及类对象内的属性名称进行点号运算，Python 会通过继承搜索从类取得变量名。

```
x.setdata("King Arthur") # Call methods: self is x
y.setdata(3.14159)       # Runs: FirstClass.setdata(y, 3.14159)
```

类：实例化

```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

- 对实例以及类对象内的属性名称进行点号运算，Python 会通过继承搜索从类取得变量名。

```
x.setdata("King Arthur") # Call methods: self is x
y.setdata(3.14159)       # Runs: FirstClass.setdata(y, 3.14159)
```

```
x.display()      # self.data differs in each instance
y.display()      # Runs: FirstClass.display(y)
```

```
King Arthur
3.14159
```

实例x, y 的 data 成员存储了不同对象类型（字符串和浮点数）。

类：实例化

- 修改实例属性

```
x.display()
x.data = "New value" # Can get/set attributes
x.display()          # Outside the class too
```

```
King Arthur
New value
```

类：实例化

- 产生新的属性
  - Python允许对实例绑定新的属性，但很少这样做，容易引起混乱

```
x.anothername = "spam" # Can set new attributes here too!
print(y.anothername)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-143f08f14ea0> in <module>
      1 x.anothername = "spam" # Can set new attributes here too!
----> 2 print(y.anothername)

AttributeError: 'FirstClass' object has no attribute 'anothername'
```

类：构造函数__init__()

- 实例属性的初始化,在启动类时执行
- 将值赋给对象属性，或者在创建对象时需要执行的其他操作：
  - 在创建实例的时候，把一些属性强制加进去。
  - 我们通常使用它来初始化所有变量。

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Bill", 63)

print(p1.name)
print(p1.age)
```

```
Bill
63
```

类：构造函数__init__()

- 实例属性的初始化,在启动类时执行
- 将值赋给对象属性，或者在创建对象时需要执行的其他操作：
  - 在创建实例的时候，把一些属性强制加进去。
  - 我们通常使用它来初始化所有变量。
  - 当创建对象时自动执行的操作。

```
class Person:
    def __init__(self, name="Jack", age=20):
        self.name = name
        self.age = age
        self.sayhello()

    def sayhello(self):
        print("Hello, I'm "+self.name+", I'm {} years old.".format(self.age))

p1 = Person()
```

```
Hello, I'm Jack, I'm 20 years old.
Jack
20
```

类：继承

- 继承是面向对象编程中的一项强大功能。

```
class BaseClass:
  #基类主体
class DerivedClass(BaseClass):
  #派生类的主体
```

- 它指的是定义一个新类，而对现有类的进行很少修改或没有修改。
  - 新类称为派生（或子）类，而从其继承的新类称为基（或父）类。
  - 子类从父类继承一切属性和方法。
  - 运行父类和子类有相同名字的方法，子类相同方法有不同的工作，称为重载。
  - 当父类和子类的对象调用同样的方法，但是行为不同，这就是多态。
- Python 是一种动态编程语言，相比于以往的OOP编程语言有些新的特性

类的继承

```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance

class SecondClass(FirstClass): # Inherits setdata
    def display(self): # Changes display
        print('Current value = "{}"'.format(self.data))

z = SecondClass()
z.setdata(42)        # Finds setdata in FirstClass
z.display()          # Finds overridden method in SecondClass
```

setdata继承，display 方法重载

类的继承

```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance
```

```
class SecondClass(FirstClass): # Inherits setdata
    def display(self): # Changes display
        print('Current value = "{}"'.format(self.data))
```

```
z = SecondClass()
z.setdata(42)        # Finds setdata in FirstClass
z.display()          # Finds overridden method in SecondClass
```

```
Current value = "42"
```

类的继承

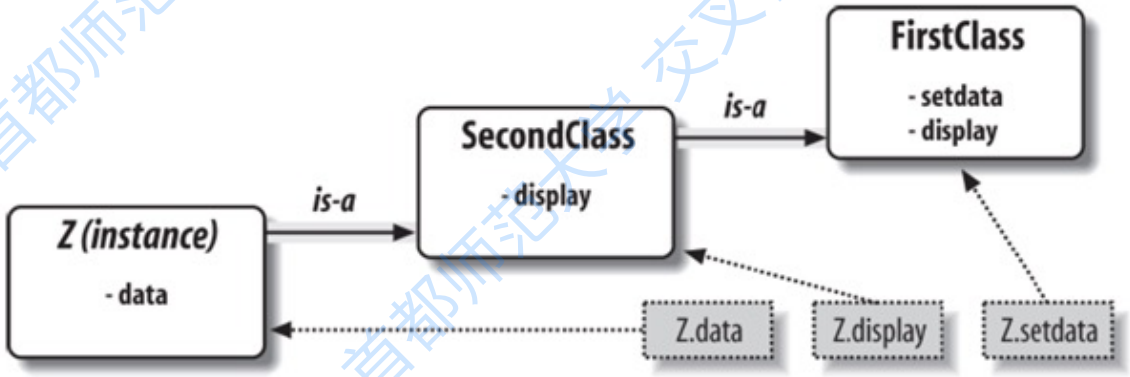
```
class FirstClass:        # Define a class object
    def setdata(self, value):    # Define class's methods
        self.data = value        # self is the instance

    def display(self):
        print(self.data)         # self.data: per instance

class SecondClass(FirstClass): # Inherits setdata
    def display(self): # Changes display
        print('Current value = "{}"'.format(self.data))
```

```
z = SecondClass()
z.setdata(42)        # Finds setdata in FirstClass
z.display()          # Finds overridden method in SecondClass
```

```
Current value = "42"
```



类的继承

- 构造函数的重载

```
class Person:
    def __init__(self, name="Jack", age=20):
        self.name = name
        self.age = age
        self.sayhello()

    def sayhello(self):
        print("Hello, I'm "+self.name+", I'm {} years old.".format(self.age))
class Student(Person):
    def __init__(self, score=90):
        self.score=score
```

```
s1 = Student()
dir(s1)
```

```
__
__
'sayhello',
'score']
```

```
p1 = Person()
dir(p1)
```

```
__
__
'age',
'name',
'sayhello']
```

类的继承

- 构造函数的重载
- 从父类继承属性的两种方法

```
class Student(Person):
    def __init__(self, score=90):
        Person.__init__(self)
        self.score=score
s1 = Student()
dir(s1)
```

```
Hello, I'm Jack, I'm 20 years old.
['__class__',
'__delattr__',
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'age',
'name',
'sayhello',
'score']
```

类的继承

```
class Person:
    def __init__(self, name="Jack", age=20):
        self.name = name
        self.age = age
        self.sayhello()
```

- 构造函数的重载
- 从父类继承属性的两种方法

```
class Student(Person):
    def __init__(self, score=90):
        super(Student,self).__init__()
        self.score=score
s2 = Student()
dir(s2)
```

```
Hello, I'm Jack, I'm 20 years old.

'age',
'name',
'sayhello',
'score']
```

通过使用 super() 函数，不必使用父类的名称，它将自动从父类继承方法和属性。

类和面向对象

- 类
  - 定义
  - 实例化
  - 继承
- 类的特殊方法
- 类的运算符重载
- 编写一个类
  - 定义
  - 子类
  - 测试工具

类的特殊方法

- object
  - 所有类的基类或父类
  - 如果一个类在定义中没有明确定义继承的基类，那么默认就会继承 object。

```
print(Empty.__mro__)  #Print Inheritance
```

```
(<class '__main__.Empty'>, <class 'object'>)
```

```
for i in dir(Empty):
    print(i)
```

```
__class__
__delattr__
__dict__
__dir__
__doc__
__eq__
__format__
__ge__
__getattribute__
__gt__
__hash__
__init__
__init_subclass__
__le__
__lt__
__module__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__
__weakref__
```

类的特殊方法

- __class__属性
  - __class__是是实例的一个属性，指向实例所属的类
- __doc__
  - 一个有效的属性，将返回所属类的文档字符
- __str__
  - 默认使用 print() 方法打印一个对象时，就是对它的调用
  - 可以重写这个函数还实现自定义类向字符串的转换。
- **逻辑运算**
  - __eq__，__ge__，__gt__，__le__，__lt__ 和 __ne__。
  - 默认使用对象的地址，也即 id 进行比较。

```
__class__
__delattr__
__dict__
__dir__
__doc__
__eq__
__format__
__ge__
__getattribute__
__gt__
__hash__
__init__
__init_subclass__
__le__
__lt__
__module__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__
__weakref__
```

类的特殊方法

```
class Empty():
    """This is an empty class"""
    pass
```

```
nobody =Empty()
nobody.__class__
```

```
__main__.Empty
```

```
nobody.__doc__
```

```
'This is an empty class'
```

```
print(nobody)
```

```
<__main__.Empty object at 0x7fb2ada44e10>
```

```
class Empty():
    """This is an empty class"""
    def __str__(self):
        return "{}".format("Empty Class!")
```

```
print(nobody)
```

```
Empty Class!
```

```
__class__
__delattr__
__dict__
__dir__
__doc__
__eq__
__format__
__ge__
__getattribute__
__gt__
__hash__
__init__
__init_subclass__
__le__
__lt__
__module__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__
__weakref__
```

类：运算符重载

- “运算符重载” 意思是在类方法中**拦截**内置操作的操作
- 当类的实例出现在内置操作中，Python自动调用你的方法，并且你的方法的返回值变成了相应操作的结果。
  - 比如print()方法， 初始化__init__
- 在数学中，运算符重载比较重要，例如，向量或矩阵类可以重载加法运算符。

类：运算符重载

```
class FirstClass:       # Defi
    def setdata(self, value):
        self.data = value

    def display(self):
        print(self.data)
```

```
class ThirdClass(FirstClass): # Inherit from SecondClass
    def __init__(self, value): # On "ThirdClass(value)"
        print('init {}'.format(value))
        self.data = value

    def __str__(self):       # On "print(self)", "str()"
        return '[ThirdClass: %s]' % self.data

    def __add__(self, other): # On "self + other"
        return ThirdClass(self.data + other)

    def mul(self, other): # In-place change: named
        self.data *= other
```

注意 __add__ 方法创建并返回这个类的新的实例对象。

```
a = ThirdClass('cnu') # __init__ called
```

```
init cnu
```

类：运算符重载

```
print(a)              # __str__: returns display string
b = a+'_math'         # __add__: makes a new instance
print(b)
```

```
[ThirdClass: cnu]
init cnu_math
[ThirdClass: cnu_math]
```

```
type(b)
```

```
__main__.ThirdClass
```

类：运算符重载

```
print(a)                # __str__: returns display string
b = a+'_math'           # __add__: makes a new instance
print(b)
```

```
[ThirdClass: cnu]
init cnu_math
[ThirdClass: cnu_math]
```

```
type(b)
```

```
__main__.ThirdClass
```

```
a.mul(3)                # mul: changes instance in place
print(a)
```

```
[ThirdClass: cnucnucnu]
```

类：运算符重载

- 下表列出其中一些常用的重载方法。

方法	重载	调用
__init__	构造函数	对象建立：X = Class（args）
__add__	运算符 +	X + Y, X += Y
__or__	运算符|（位OR）	X | Y, X |= Y
__str__	打印	print(X),
__lt__, __gt__, __ge____eq__, __ne__	特定比较	X<Y, X>Y, X<=Y, X>=Y, X == Y, X != Y
__ sub __	运算符-	X-Y
__ pow __	求幂（**）	p1 ** p2

具体例子

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

这段代码意味着当我们创建 Person 对象 时，需要给 name 传入值，但是 job 和 pay 是可选的

```
if __name__ == '__main__': # When run for testing only
    # self-test code
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

文件底部的测试代码

```
Bob Smith 0
Sue Jones 100000
```

具体例子：添加方法

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Behavior methods
        return self.name.split()[-1] # self is implied subject

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent)) # Must change here only
```

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(bob.lastName(), sue.lastName()) # Use the new methods
    print(sue.name, sue.pay)
    sue.giveRaise(.10) # instead of hardcoding
    print(sue.pay)
```

具体例子：运算符重载

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Behavior methods
        return self.name.split()[-1] # self is implied subject

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent)) # Must change here only
    def __str__(self):                  # Added method
        return '[Person: {}, {}]'.format(self.name, self.pay) # String to print
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
```

具体例子：定义子类

- Manager
  - Person的子类，它用一个特殊的版本替代了继承的 giveRaise 方法。
  - 当一个 Manager 要涨工资的时候，它像往常一样接受一个百分比，同时也会获得一份默认10%的额外奖金。

```
class Manager(Person): # Inherit Person attrs
    def giveRaise(self, percent, bonus=.10): # Redefine to customize
```

如何实现Manager中的giveRaise()

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        # Bad: cut and paste
        self.pay = int(self.pay * (1 + percent + bonus))
```

复制粘贴 Person 中的 giveRaise 代码，然后针对 Manager 进行修改

造成的问题是涨工资的方式发生变化，需要修改这两个地方的代码。

具体例子：定义子类

- 使用修改的参数来直接调用其最初的版本：

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        # Good: augment original
        Person.giveRaise(self, percent + bonus)
```

通过类来调用类方法

```
if __name__=="__main__":
    tom = Manager('Tom Jones', 'mgr', 50000)
    print(tom)            # Runs inherited __str__
    print(tom.lastName()) # Runs inherited method
    tom.giveRaise(.10)    # Runs custom version
    print(tom)            # Runs inherited __str__
```

```
[Person: Tom Jones, 50000]
Jones
[Person: Tom Jones, 60000]
```

可以看出，Manager 对象自动从 Person 继承了 __init__、lastName 和 __str__ 方法。

具体例子：定制构造函数

```
class Manager(Person):
    # Redefine constructor
    def __init__(self, name, pay):
        # Run original with 'mgr'
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

```
tom = Manager('Tom Jones', 50000) # Job name not needed
print(tom)
```

以这种重定义的方式调用父类构造函数，在 Python 中是一种很常见的编码模式。

- 可以指出哪些参数传递给父类的构造函数，也可以选择根本不调用它。
- 不调用父类的构造函数允许**替代**父类构造函数。

具体例子：测试工具

- 在测试代码中，我们希望看到实例的类别以及属性值
  - 首先，当打印 tom 的时候，我们希望知道他的类是什么。
  - 其次，打印的时候，我们希望知道实例的具体属性有哪些。
- 介绍过的特殊方法
  - instance.__class__，实例所述的类
  - instance.__dict__，一对字典值：属性和对象中属性的值

具体例子：测试工具

```
bob = Person('Bob Smith')
print(bob)
```

```
[Person: Bob Smith, 0]
```

```
bob.__class__  # Show bob's c
```

```
__main__.Person
```

```
bob.__class__.__name__
```

```
'Person'
```

```
for key in bob.__dict__:
    print(key, '=>', bob.__dict__[key]) # Index manually
```

```
name => Bob Smith
job => None
pay => 0
```

```
for key in bob.__dict__:
    print(key, '=>', getattr(bob, key))
```

```
name => Bob Smith
job => None
pay => 0
```

具体例子：测试工具

- 通用显示工具
  - 打印的时候，我们希望知道实例的类、属性及属性值。
- AttrDisplay 类， 重载了 __str__ 方法

```
class AttrDisplay:
    """
    Provides an inheritable display overload method that shows
    instances with their class names and a name=value pair for
    each attribute stored on the instance itself (but not attrs
    inherited from its classes). Can be mixed into any class,
    and will work on any instance.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __str__(self):
        return '[%s: %s]' % (self.__class__.__name__, self.gatherAttrs())
```

具体例子：测试工具

- 将顶层类 Person 修改为从 AttrDisplay 类继承而来，因此类 Person 和子类 Manager 的实例都会继承新的打印重载方法。

```
class Person(AttrDisplay): # Mix in a repr at this level
    """
    Create and process person records
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self): # Assumes last is last
        return self.name.split()[-1]
    def giveRaise(self, percent): # Percent must be 0..1
        self.pay = int(self.pay * (1 + percent))
```

具体例子：测试工具

- 现在再运行测试代码，将会输出对象的所有属性，而不只是最初的 __str__ 中直接编码的那些属性。

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
```

```
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
```

具体例子：测试工具

- 将顶层类 Person 修改为从 AttrDisplay 类继承而来，因此类 Person 和子类 Manager 的实例都会继承新的打印重载方法。

```
class Manager(Person):
    """
    A customized Person with special requirements
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay) # Job name
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

具体例子：测试工具

- 将顶层类 Person 修改为从 AttrDisplay 类继承而来，因此类 Person 和子类 Manager 的实例都会继承新的打印重载方法。

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)
```

```
[Person: job=None, name=Bob Smith, pay=0]
[Person: job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person: job=dev, name=Sue Jones, pay=110000]
Jones
[Manager: job=mgr, name=Tom Jones, pay=60000]
```

面向对象的思想

- 通过**定制**(customizing)来编程，而不是复制和修改已有的代码。
  - 通过继承，Manager 可以具有与Person 相同的行为，不用重写写代码。
  - 避免代码冗余和向后兼容的问题，对 Person 的修改能够在Manager中体现。
  - 通过定义自己的方法，避免修改Person造成它无法满足其他需求。
- 我们可以用类来构建**可定制层级结构**，通过编写新的子类来裁剪或扩展之前的工作，为那些将会随时间发展的软件提供一个更好的解决方案。

练习

- 1.Python 类中的 **self** 有什么意义？
- 2.实现学生选课系统
  - 学生类
    - *属性*：姓名、学号、电话、所选课程列表
    - *方法*：**查看**：显示该学生所有课程信息；**添加课程**：将选好的课程添加到课程列表中
  - 课程类
    - *属性*：课程编号、课程名称、教师名
    - *方法*：**查看**：显示该课程的全部信息；**设置教师**：给当前课程安排一个教师
  - 教师类：
    - *属性*：教师编号，教师名、电话、所教课程列表
    - *方法*：**查看**：查看该教师的所有课程
  - 完成以上三个类，并创建20名学生，6个课程，3名教师。给课程随机安排任课教师并给20名学生随机分配3个课程，最终显示这20名学生选课情况。