返回目录

第七章　类与对象

本章导读

对象是Python对数据的抽象。在Python程序中，所有数据都表现为对象或者对象之间的关系。每个对象都有确定的身份以及其值和类型。在Python中，对象一旦被创建，其身份就不会改变，可以将其简单理解为内存中的地址。is运算符用于比较两个两个对象是否相同，id()函数返回一个整数，代表对象的身份， 如同对象在程序中的身份证号码。

学习目标：

1. 理解类-对象的基本概念；
2. 理解类的封装与继承；
3. 掌握类-对象在Python的应用；

本章目录

第一节　基本概念

类是对象的抽象，对象是类的实例。在现实生活中，“马”我们可以称之为类，是对所有马的抽象，白马黑马是马的实例，是具体的某匹马。在例程7-1中定义了一个名为UpDown的类，是对分数的抽象，myUD和hisUD是UpDown类的实例。

例程7-1

```
第1行  #class的定义及其使用
第2行
第3行  class UpDown():
第4行      def __init__(self,u,d):#UpDown对象初始化
第5行          self.Up=u
第6行          self.Down=d
第7行
第8行      def printUD(self):#printUD()函数用于打印输出
第9行          print(self.Up,'/',self.Down,sep='')
第10行
第11行 myUD=UpDown(1,3)
第12行 myUD.printUD()
```

```
第13行
第14行  hisUD=myUD+UpDown(1,2)
第15行  hisUD.printUD()
第16行
第17行  #eof
```

在面向对象体系中，类将描述类的数据以及处理该数据的函数集成在一起。描述类的数据称之为属性(attribute)或者数据成员(data member)，处理数据的函数称之为方法(method)或者函数成员(function member)。例程7-1中，UpDown类的属性为Up和Down，通过__init__()函数设定，printUD()和__init__()则称之为方法。一个类可以有多个数据成员也就是属性，也可以有多个函数成员也就是方法。在Python中，所有成员函数的第一个参数都为self，表示对象自身，在调用时，无需也不能传入该参数。

例程7-1中UpDown类的__init__()是比较特殊的方法，可以被称之为构造函数，是对象实例化时自动被执行的函数。与__init__对应的还有析构函数，其方法名称为__del__()。通过例程7-2就可以观察到__init__()和__del__()被调用过程。

例程7-2

```
第1行   #init和del的使用
第2行
第3行   class UpDown():
第4行       def __init__(self,u,d):#UpDown对象初始化
第5行           print("In init!",end=' ')
第6行           self.Up=u
第7行           self.Down=d
第8行       def __del__(self):
第9行           print("In del!",end=' ')
第10行
第11行  def Test():
第12行      tmp=UpDown(1,2)
第13行      return tmp
第14行
第15行  myUD=UpDown(1,3)
第16行  Test()
第17行
第18行  tmp=Test()
第19行
第20行  #全部输出为：In init! In init! In del! In init! In del! In del!
第21行   #eof
```

第二节　访问限制

观察例程7-3中第11-13行，可以发现UpDown的属性可以直接被访问。在Python语言中，class的属性如没有特别限定，可以认为其访问控制默认为public，可以直接访问其相关属性。观察第15行发现类可以增加新的属性。这样的特性，体现Python的动态语言特征，和Java以及C++等有明显的不同。

例程7-3

```
第1行   #class的访问控制
第2行
第3行   class UpDown():
第4行       def __init__(self,u,d):#UpDown对象初始化
第5行           self.Up=u
第6行           self.Down=d
```

```
第7行     def printUD(self):
第8行         print(self.Up,"/",self.Down,sep='')
第9行
第10行    myUD=UpDown(1,3)
第11行    myUD.Up=10#直接可以修改属性值
第12行    myUD.Down=11
第13行    myUD.printUD()#输出：10/11
第14行
第15行    myUD.value=myUD.Up/myUD.Down#增加新属性
第16行
第17行    print(myUD.value)#输出：0.9091
第18行
第19行     #eof
```

class的属性可以限定为private，其属性也可以限定为不允许增加，如例程7-4所示，在属性名称前增加两个下划线(__)即可设定该属性为private属性【注：两个下划线也是属性名称的组成部分】，不允许在类外访问该属性。通过两个下划线不仅仅可以限制属性名称为private，也可以限制方法也就是成员函数为privae，如例程第10行所示。

例程7-4

```
第1行    #class的访问控制
第2行
第3行    class UpDown():
第4行        __slots__=('__Up','__Down')#本类属性仅限于__Up和__Down，
第5行        def __init__(self,u,d):#UpDown对象初始化
第6行            self.__Up=u
第7行            self.__Down=d
第8行        def printUD(self):
第9行            gcd=self.__GCD()
第10行           print(self.__Up//gcd,"/",self.__Down//gcd,sep='')
第11行       def __GCD(self):
第12行           lastNum=min(self.__Up,self.__Down)
第13行           while lastNum>1:
第14行               if self.__Up % lastNum==0 and self.__Down%lastNum==0:
第15行                   return lastNum
第16行               else:
第17行                   lastNum-=1
第18行           else:
第19行               return 1
第20行
第21行   myUD=UpDown(20,30)
第22行   myUD.printUD()#输出：10/11
第23行
第24行   #eof
```

限制在类外增加新属性的方法如代码第4行所示，注意：属性名称两侧的引号不能省略。

第三节　运算符函数

假如有一个UpDown对象的List，能对其使用sorted()函数吗？答案是否定的。不管是哪种类型的排序(sort)，都将比较两个数的大小，一般默认为小于比较(less than)。对于UpDown而言，如何进行比较呢？对于一个新定义的类而言，不可能自动知道如何比较。例程7-5第9-10行定义了一个__lt__()方法，则第12、14行可以正确执行。UpDown类中的__lt__()方法实际上是对<的重载也就是针对该类的定义。

Python有很多运算符，都可以进行重载，运算符及其对应的方法如表7-1

例程7-5

```
第1行  #运算符重载
第2行
第3行  class UpDown(object):
第4行      def __init__(self,u,d):#UpDown对象初始化
第5行          self.Up=u
第6行          self.Down=d
第7行      def printUD(self):
第8行          print(self.Up,"/",self.Down,sep='',end='\t')
第9行      def __lt__(self,other):
第10行         return self.Up*other.Down<other.Up*self.Down
第11行
第12行 print(UpDown(2,3)<UpDown(4,5))#输出：true
第13行 listUD_A=[UpDown(1,2),UpDown(1,6),UpDown(1,3),UpDown(2,5)]
第14行 listUD_B=sorted(listUD_A)
第15行 for ud in listUD_B:
第16行     ud.printUD()#输出：1/6 1/3 2/5 1/2
第17行
第18行 #eof
```

表7-1：Python运算符方法名称

类别	函数原型	含义	示例	备注
算术运算符	object.__add__(self, other)	+	加法运算符	
	object.__sub__(self, other)	-		
	object.__mul__(self, other)	*		
	object.__matmul__(self, other)			
	object.__truediv__(self, other)			
	object.__floordiv__(self, other)			
	object.__mod__(self, other)			
	object.__divmod__(self, other)			
	object.__pow__(self, other[, modulo])			
	object.__lshift__(self, other)			
	object.__rshift__(self, other)			
关系运算符	object.__lt__(self, other)	<		小于
	object.__le__(self, other)	<=		小于等于
	object.__gt__(self, other)	>		大于
	object.__ge__(self, other)	>=		大于等于
	object.__eq__(self, other)	==		相等

类别	函数原型	含义	示例	备注
	object.__ne__(self, other)	!=、<>		不等于
逻辑运算符	object.__not__(self)	!		逻辑非
	object.__and__(self, other)	and		逻辑与
	object.__or__(self, other)	or		逻辑或
	object.__xor__(self, other)	xor		逻辑异或

第四节　特殊方法函数

对于UpDown而言，printUD()方法函数是一个普通方法函数，由用户命名。在Python类中，还有很多特殊方法函数，其命名和参数个数都已经由系统约定，如：__init__()、__and__()等。实际上，运算符重载等等，也是特殊方法函数，除此以外，Python还提供了其他特殊方法函数。

1、__str__(self)、__repr__(self)

在例程7-6第13行定义了一个List，其中数据为UpDown类型，第14-15行是遍历该List并输出每个成员的值。虽然UpDown提供了printUD()方法函数，但显然不及Python之print()函数灵活并符合习惯。为实现该目标，只要在类中定义一个特殊的方法函数__str__()即可，相当于将UpDown的对象转换为str类型。

例程7-6

```
第1行  #__str__()方法函数
第2行
第3行  class UpDown(object):
第4行      def __init__(self,u,d):#UpDown对象初始化
第5行          self.Up=u
第6行          self.Down=d
第7行      def printUD(self):
第8行          print(self.Up,"/",self.Down,sep='',end='\t')
第9行      def __str__(self):
第10行         return "("+str(self.Up)+"/"+str(self.Down)+")"
第11行     __repr__=__str__
第12行
第13行 listUD=[UpDown(1,2),UpDown(1,5),UpDown(2,3),UpDown(4,3)]
第14行 for ud in listUD:
第15行     print(ud,end="\t")#输出：(1/2) (1/5) (2/3) (4/3)
第16行
第17行 #eof
```

__repr__()方法函数与__str__()功能相似，__repr__()更多是用于Python交互式环境，如：执行UpDown(2,3)时，显示内容由__repr__()定义。由于__repr__()与__str__()经常相同，可以直接写为__repr__=__str__，如例程7-6第11行所示。

Python还提供了__int__(self)、__float__(self)、__round(self[,n])以及__complex(self)__用于类型转换，其用法与__str__(self)类似。

2、__iter__(self)、__next__(self)

如一个类想用于for ... in场景，如同list或tuple，其前提是实现一个__iter__()方法，该方法返回self，然后，for循环会不断调用该对象__next__()方法函数，直到遇到StopIteration异常时退出循环。

例程7-7

```
第1行  #__iter__()、__next__()方法函数的示例
第2行
第3行  class Fibo():
```

```
第4行     def __init__(self):
第5行         print("__init__")
第6行         self.First,self.Second=0,1
第7行     def __iter__(self):
第8行         print("__iter__")
第9行         return self
第10行    def __next__(self):
第11行        print("__next__",end="\t")
第12行        if self.First > 100000: # 退出循环的条件
第13行            raise StopIteration();
第14行
第15行        rtnVal=self.First
第16行        self.First, self.Second = self.Second, self.First + self.Second # 计算下一个值
第17行        return rtnVal # 返回下一个值
第18行
第19行    for n in Fibo():
第20行        print(n)
第21行
第22行    #eof
```

3、__getitem__(self,key)

对于Fibo类，虽然能用于for...in场景，但还不能用于类似Fibo()[10]即下标场景，如果有__getitem__(self,key)方法函数，则可以实现，如例程7-8所示。

例程7-8

```
第1行    #__iter__()、__next__()方法函数的示例
第2行    #本程序可以大大优化，没必要每次都从头计算
第3行
第4行    class Fibo():
第5行        def __init__(self):
第6行            print("__init__")
第7行            self.First,self.Second=0,1
第8行        def __getitem__(self,Key):
第9行            self.First,self.Second=0,1
第10行           for i in range(Key):
第11行               self.First, self.Second = self.Second, self.First + self.Second
第12行           return self.First
第13行
第14行   myFibo=Fibo()
第15行   for i in range(10):
第16行       print("第",i,"个：",myFibo[i])
第17行
第18行   #eof
```

例程7-8能实现myFibo[5]，但还不能实现myFibo[2:5]，也就是切片形式，例程7-9是其优化。虽如此，还有很多种情况没有考虑，比如：myFibo[10:2]、myFibo[10:2:-2]等等。

例程7-9

```
第1行  #__iter__()、__next__()方法函数的示例
第2行  #本程序可以大大优化，没必要每次都从头计算
第3行
第4行  class Fibo():
第5行      def __init__(self):
第6行          print("__init__")
第7行          self.First,self.Second=0,1
第8行      def __getitem__(self,Key):
第9行          First,Second=0,1
第10行         if isinstance(Key,int):#当key为int时
第11行             for i in range(Key):
第12行                 First, Second = Second, First + Second
第13行             return First
第14行
第15行         if isinstance(Key,slice):#当key为slice对象时
第16行             numBeg=Key.start
第17行             if numBeg is None:numBeg = 0
第18行
第19行             numEnd=Key.stop
第20行
第21行             numStep=Key.step#间隔
第22行             if numStep==None:numStep=1#当没有间隔
第23行
第24行
第25行             fiboList = []
第26行             for i in range(numEnd):
第27行                 if i>=numBeg:
第28行                     if len(fiboList)==0:
第29行                         appendIndex=i
第30行                         fiboList.append(First)
第31行                     else:
第32行                         if i-appendIndex==numStep:
第33行                             fiboList.append(First)
第34行                             appendIndex=i
第35行
第36行                 First,Second = Second, First + Second
第37行             return fiboList
第38行
第39行 myFibo=Fibo()
第40行 for i in range(10):
第41行     print("第",i,"个：",myFibo[i])
第42行 print(myFibo[:10])
第43行 print(myFibo[:10:1])
第44行 #eof
```

例程7-10是将__getitem__(self,Key)应用在UpDown类。

例程7-10

```
#__getitem__()方法函数

class UpDown(object):
    def __init__(self,u,d):#UpDown对象初始化
        self.Up=u
        self.Down=d
    def __getitem__(self,Index):
        if Index%2==0:
            return self.Up
        else:
            return self.Down

myUD=UpDown(2,3)
print(myUD[0],myUD[1])

#eof
```

和例程7-10相比，例程7-11更能体现__getitem__()的用法。

例程7-11

```
#__getitem__()方法函数

class Prime():
    def __init__(self,N):#得到N以内的所有质数
        self.__primeList=[]
        self.__maxIntNum=N
        for i in range(2,N+1):
            if self.__isPrime(i)==True:
                self.__primeList.append(i)

    def __isPrime(self,N):
        lastNum=N//2+1
        for i in range(2,lastNum+1):
            if N%i==0:
                return False
        else:
            return True

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

prime=Prime(1000)
for p in prime:
    print(p,end="\t")

```

```
第26行 print(prime[2:10])
第27行
第28行 #eof
```

4、__getattr__(self,attr)

当调用一个类不存在的属性或者方法时，将会报错，如例程7-12所示，如果没有定义__getattr__(self,attr)调用myUD.getUp()，其报错内容为“AttributeError: 'UpDown' object has no attribute 'getUp'”【属性错误：UpDown对象没有getUp属性】。

例程7-12

```
第1行  #__getitem__()方法函数
第2行
第3行  class UpDown(object):
第4行      def __init__(self,u,d):#UpDown对象初始化
第5行          self.Up=u
第6行          self.Down=d
第7行      def __getitem__(self,Index):
第8行          if Index%2==0:
第9行              return self.Up
第10行         else:
第11行             return self.Down
第12行     def __getattr__(self,attr):
第13行         if attr=="Up":
第14行             return self.Up
第15行         if attr=="Down":
第16行             return self.Down
第17行
第18行         #如果不是Up或Down属性
第19行         raise AttributeError("UpDown对象没有定义%s%s"%(attr,"属性或方法"))
第20行
第21行 myUD=UpDown(1,2)
第22行 print(myUD.Up,"/",myUD.Down)
第23行 myUD.getUp()#本行将触发异常
第24行
第25行 #eof
```

5、__call__()

类的__call__(self)方法函数能将类的实例函数化，如例程7-13第11行所示。另外，__call__()方法函数除self参数外，还可以增加其他参数。一旦一个类增加了__call__(self)方法函数，则成为可调用，通过callable()函数可以测试其是否可以调用，如例程第13-20行所示。从中也可以看出用callable()测试函数，其值为True。

例程7-13

```
第1行  #__call__()方法函数
第2行
第3行  class UpDown(object):
第4行      def __init__(self,u,d):#UpDown对象初始化
第5行          self.Up=u
第6行          self.Down=d
```

```
第7行     def __call__(self):
第8行         print(self.Up,"/",self.Down,sep='')
第9行
第10行    myUD=UpDown(1,3)
第11行    myUD()
第12行
第13行    print(callable(myUD))#输出：True
第14行    listA=[]
第15行    print(callable(listA))#输出：False
第16行
第17行    def Test():
第18行        pass
第19行
第20行    print(callable(Test))#输出:True
第21行
第22行    #eof
```

Python的__call__()与C++的括号运算符重载很相似，也能实现类似的功能，比如：函数对象等(参加本章第8八节)。

第五节　类属性

类可以实例化为对象，即便来自同一个类，不同对象的属性也互不影响。在Python语言中，还存在类属性，该属性为类所有，也为所有该类的对象所有。基于此，可以将属性的分为类属性和对象属性(也成实例属性)。在一般的表达中，如果没有特别申明，属性都是指对象属性。例程7-14是类属性的应用示例，其中__objectLastNum、__Index为对象属性，primeList、__classLastNum为类属性。

例程7-14的功能求出质数列表。在第38行、第43行，分别申明myPrime=Prime(100)和hisPrime=Prime(50)，很明显，Prime(50)仅仅是Prime(100)子集，既然是一个子集，就没有必要再次计算质数列表。为达到该目标，两个对象之间必须共享数据，在Python中可以通过类属性实现。在本例中，当执行hisPrime=Prime(50)时，虽然生成一个新的对象名为hisPrime，但该对象与myPrime共享PrimeList和__classLastNum。在__init__(self,N)方法函数中，如果新的对象的N值小于其他对象最大的N值，则不重新进行质数计算，这将减少计算量，直接应用以前计算的结果即可。

和对象属性一样，类属性也可以分为私有和公有。在例程中，primeList为公有，可以在类外直接访问，如第39、44行所示。__classLastNum为私有，只能在类内访问，如第9、10等行所示。在类内访问类属性时，必须加上__class__限定。

例程7-14

```
第1行     #类属性
第2行
第3行     class Prime():
第4行         primeList=[]#类共有属性
第5行         __classLastNum=2#类私有属性
第6行         def __init__(self,N):#得到N以内的所有质数
第7行             self.__objectLastNum=N
第8行             self.__Index=0
第9行             if N>self.__class__.__classLastNum:
第10行                for i in range(self.__class__.__classLastNum,N+1):
第11行                    if self.__isPrime(i)==True:
第12行                        self.__class__.primeList.append(i)#访问类属性
第13行                self.__class__.__classLastNum=N
第14行
```

```
第15行     def __iter__(self):
第16行         return self
第17行
第18行     def __next__(self):
第19行         if self.__Index>=len(self.__class__.primeList):
第20行             raise StopIteration
第21行
第22行         if self.__class__.primeList[self.__Index]>=self.__objectLastNum:
第23行             raise StopIteration()
第24行
第25行         rtnVal=self.__class__.primeList[self.__Index]
第26行         self.__Index+=1
第27行         return rtnVal
第28行
第29行
第30行     def __isPrime(self,N):
第31行         if N==2:return True
第32行         lastNum=N//2+1
第33行         for i in range(2,lastNum+1):
第34行             if N%i==0:return False
第35行         return True
第36行
第37行
第38行 myPrime=Prime(100)
第39行 for i in myPrime.primeList:
第40行     print(i,end='\t')
第41行
第42行 print()
第43行 hisPrime=Prime(50)
第44行 for i in hisPrime.primeList:
第45行     print(i,end="\t")
第46行
第47行 print()
第48行 for i in hisPrime:
第49行     print(i,end="\t")
第50行
第51行 print()
第52行 for i in myPrime:
第53行     print(i,end="\t")
第54行
第55行 print()
第56行 herPrime=Prime(200)
第57行 for i in herPrime:
第58行     print(i,end="\t")
第59行
```

```
第60行 #eof
```

第六节 @property装饰器

例程7-15是一个含有出生年份Birth属性的Student学生类。该例所示设计明显存在问题，当年份低于0或者高于150时，系统不会提示任何信息。例程7-16是Student的改进，虽然解决面临的问题，但显得有些复杂。Python提供的@property是对该问题的折衷解决，如例程7-17所示。

例程7-15

```
第1行 class Student:
第2行     def __init__(self,b):
第3行         self.__Birth=b#__Birth为出生年份
第4行 
第5行 studA=Student(-10)#明显不合理
第6行 studB=Student(1800)#明显不合理
第7行 
第8行 #eof
```

例程7-16

```
第1行 class Student:
第2行     def setBirth(self,b):
第3行         if not isinstance(b,int):
第4行             raise ValueError("出生年份必须为整数！")
第5行         if b<2016 and b>1900:
第6行             raise ValueError("出生年份必须介于1900-2016之间！")
第7行         self.__Birth=b
第8行     def getBirth(self):
第9行         return self.__Birth
第10行 
第11行 studA=Student()
第12行 studA.setBirth(25000)#明显不合理，将触发异常
第13行 
第14行 #eof
```

@property能将getter函数转变为属性，原有getter函数将变得不能直接使用，如print(studA.Birth())将不能执行。@Birth.setter则将Birth()函数转换为setter函数，与@property类似，原有setter函数将变得不能直接使用，即studA.Birth(1990)将不能正确执行。如果一个属性要求能读能写，则既要有@property，也得有该属性的setter设置。如果一个属性只需要读，则可以只有@property，如Age属性。如果一个属性只允许写，而不能读，则该属性setter不能单独存在，即第3行-5行如果删除，第7行将报错。

例程7-17

```
第1行 class Student:
第2行     __slots__=("__Birth")
第3行     @property
第4行     def Birth(self):#getter函数
第5行         return self.__Birth
第6行 
第7行     @Birth.setter
第8行     def Birth(self,b):#setter函数
第9行 
```

```
        if not isinstance(b,int):
第10行          raise ValueError("出生年份必须为整数!")
第11行      if b>2016 or b<1900:
第12行          raise ValueError('出生年份必须介于1900-2016之间!')
第13行      self.__Birth=b
第14行
第15行  @property#Age属性只读
第16行  def Age(self):#Age仅有@propery，是只读属性
第17行      return 2016-self.__Birth
第18行
第19行  studA=Student()
第20行  studA.Birth=1990
第21行  print(studA.Birth)
第22行  print(studA.Age)
第23行
第24行  #eof
```

第七节　枚举类

例程7-18

```
第1行  #枚举类
第2行  from enum import Enum
第3行
第4行  Week=Enum("Week",('Sun','Mon','Tue','Wed','Thu','Fri','Sat'))#从1开始计数
第5行
第6行  myWeek1=Week.Mon
第7行  print(myWeek1)
第8行
第9行  if myWeek1==Week.Mon:print("星期一")
第10行 if myWeek1==Week.Sun:print("星期天")
第11行
第12行 myWeek2=Week(1)#将整数转换为Week类型
第13行 if myWeek2==Week.Mon:print("星期一")
第14行 if myWeek2==Week.Sun:print("星期天")
第15行
第16行 print(Week["Tue"])#输出：Week.Tue
第17行 print(type(myWeek2))#输出<enum 'Week'>
第18行
第19行 #eof
```

例程7-19

```
第1行  #枚举类
第2行  from enum import Enum
第3行
第4行  class Week(Enum):
第5行      Sun=0
```

```
第6行     Mon=1
第7行     Tue=2
第8行     Wed=3
第9行     Thu=4
第10行    Fri=5
第11行    Sat=6
第12行
第13行  myWeekSun=Week.Sun
第14行
第15行  if myWeekSun==Week.Sun:print("星期天")
第16行  if myWeekSun==Week.Mon:print("星期一")
第17行  if myWeekSun==Week.Tue:print("星期二")
第18行  if myWeekSun==Week.Wed:print("星期三")
第19行  if myWeekSun==Week.Thu:print("星期四")
第20行  if myWeekSun==Week.Fri:print("星期五")
第21行  if myWeekSun==Week.Sat:print("星期六")
第22行
第23行  print(myWeekSun.value)
第24行  print(Week.Sun.value)
第25行  print(Week(1))#输出：Week.Mon
第26行  print(Week["Mon"])#输出：Week.Mon
第27行
第28行  for name, member in Week.__members__.items():
第29行      print(name, '=>', member)
第30行
第31行
第32行  #eof
```

第八节　函数对象

所谓函数对象，就是一个对象有类似的函数行为，在Python，如果一个类定义了__call__()函数，就有了函数能力，例程7-20的功能是利用过滤函数filter()列出小于某个数的list，此处为小于10。LT类相对于函数，具有更好灵活性。

例程7-20

```
第1行   #函数对象
第2行
第3行   class LT:
第4行       def __init__(self,N):
第5行           self.__Num=N
第6行       def __call__(self,M):
第7行           return M<self.__Num
第8行
第9行   listLT10=filter(LT(10),[3,5,10,43,56,3,21,9,8])
第10行  for i in listLT10:
第11行      print(i,end="\t")#输出：3 5 3 9 8
第12行
第13行  #eof
```

第九节　继承与多态

例程7-21

```
第1行  #类的继承
第2行
第3行  class myShape:
第4行      def getArea(self):
第5行          pass
第6行      def getPeri(self):
第7行          pass
第8行
第9行  class Rect(myShape):
第10行     def __init__(self,w,h):
第11行         self.__Width=w
第12行         self.__Height=h
第13行     def getArea(self):
第14行         return self.__Width*self.__Height
第15行     def getPeri(self):
第16行         return self.__Width+self.__Height
第17行
第18行 class Circle(myShape):
第19行     def __init__(self,r):
第20行         self.__Radius=r
第21行     def getArea(self):
第22行         return 3.14*self.__Radius*self.__Radius
第23行     def getPeri(self):
第24行         return 2*3.14*self.__Radius
第25行
第26行 def printArea(argOne):
第27行     print(argOne.getArea())
第28行
第29行 def TwoShapeAreaAdd(argFirst,argSecond):
第30行     return argFirst.getArea()+argSecond.getArea()
第31行
第32行 myShape=myShape()
第33行 printArea(myShape)#输出：None
第34行 myRect=Rect(10,20)
第35行 myCircle=Circle(10)
第36行
第37行 printArea(myRect)
第38行 printArea(myCircle)
第39行
第40行 print(TwoShapeAreaAdd(myRect,myCircle))#输出：514.0
第41行
第42行 #eof
```

例程7-22

```
#int类的继承
class myInt(int):
    def __init__(self,N):
        self.__Num=N
        self.__Digital=['零','壹','贰','叁','肆','伍','陆','柒','捌','玖']
    def __str__(self):
        tmp=''
        N=self.__Num
        if N==0:return self.__Digital[0]
        while N>0:
            tmp=str(self.__Digital[N%10])+tmp
            N//=10
        return tmp

a=myInt(1012)
print(a)

#eof
```

issubclass()函数可以判断一个类是不是另外一个类的子类或者派生类，如例程7-23所示第10行所示，由于Cat是Animal的派生类，因此其输出值为True，第11行的Dog不是Plant的派生类，所以其输出值为False。

例程7-23

```
#issubclass()函数

class Plant():
    pass
class Animal():
    pass
class Cat(Animal):
    pass


print(issubclass(Cat,Animal))#输出:True
print(issubclass(Cat,Plant))#输出:False

#eof
```

第十节　动态绑定

类的属性和方法可以动态绑定。

例程7-24

```
#! /usr/bin/python

class UpDown():
    def __init__(self,u,d):
```

```
第5行         self.Up=u
第6行         self.Down=d
第7行     def printUD(self):
第8行         print(self.Up,"/",self.Down,sep="")
第9行
第10行 if __name__=="__main__":
第11行     myUD=UpDown(2,3)
第12行
第13行     def getChnUD(self):
第14行         return "one fifth"
第15行
第16行     UpDown.getChnUD=getChnUD
第17行     print(myUD.getChnUD())
第18行
第19行 #eof
```

第十一节　装饰器

例程7-25

```
第1行 # -*- coding:utf-8 -*-
第2行 #装饰器带类参数
第3行
第4行 class locker:
第5行     def __init__(self):
第6行         print("locker.__init__() should be not called.")
第7行
第8行     @staticmethod
第9行     def acquire():
第10行         print("locker.acquire() called. ( 这是静态方法 ) ")
第11行
第12行     @staticmethod
第13行     def release():
第14行         print(" locker.release() called. ( 不需要对象实例 ) ")
第15行
第16行 def deco(cls):
第17行     '''cls 必须实现acquire和release静态方法'''
第18行     def _deco(func):
第19行         def __deco():
第20行             print("before %s called [%s]." % (func.__name__, cls))
第21行             cls.acquire()
第22行             try:
第23行                 return func()
第24行             finally:
第25行                 cls.release()
第26行         return __deco
第27行
```

```
    return _deco
第28行
第29行 @deco(locker)
第30行 def myfunc():
第31行     print(" myfunc() called.")
第32行
第33行 myfunc()
第34行 print("\nHere!!!\n")
第35行 myfunc()
第36行
第37行 #eof
```