反射

概述

运行时，区别于编译时，指的是程序被加载到内存中执行的时候。

反射，reﬂection，指的是运行时获取类型定义信息。

一个对象能够在运行时，像照镜子一样，反射出其类型信息。

简单说，在Python中，能够通过一个对象，找出其type、class、attribute或method的能力，称为反射或者自

省。

具有反射能力的函数有 type()、isinstance()、callable()、dir()、getattr()等

反射相关的函数和方法

需求

有一个Point类，查看它实例的属性，并修改它。动态为实例增加属性

class  Point:

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_str\_\_(self):

return  "Point({},  {})".format(self.x,  self.y)

def  show(self):

print(self.x,  self.y)

p  =  Point(4,  5)

print(p)

print(p.\_\_dict\_\_)

p.\_\_dict\_\_['y']  =  16

print(p.\_\_dict\_\_)

p.z  =  10

print(p.\_\_dict\_\_)

print(dir(p))  #  ordered  list

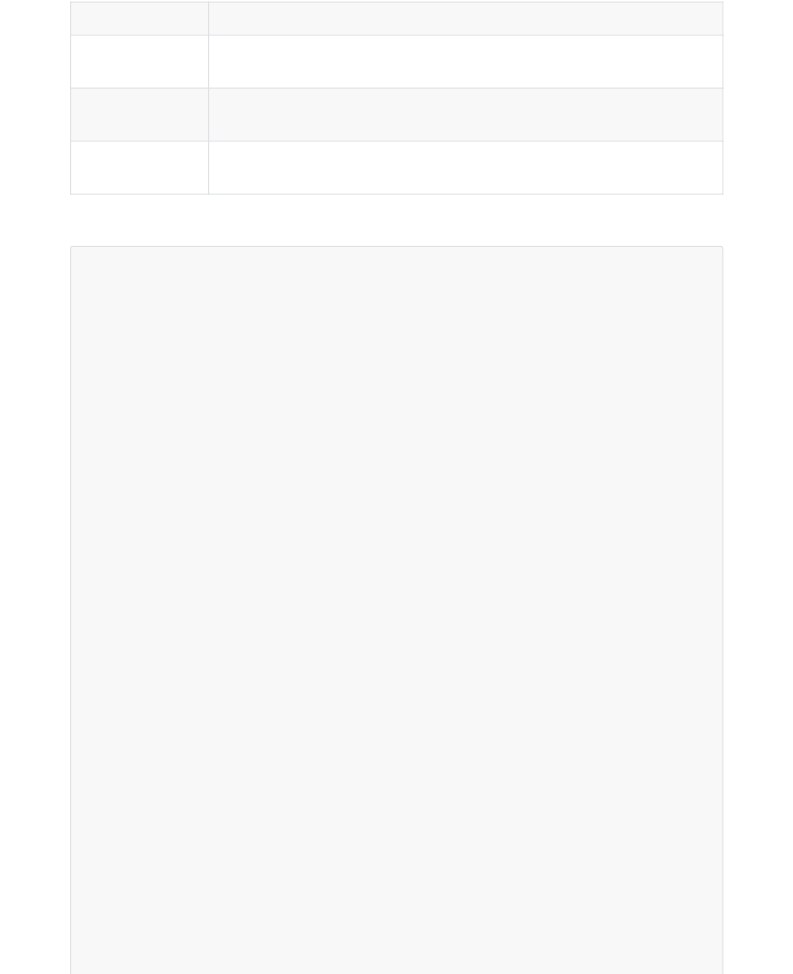
print(p.\_\_dir\_\_())  #  list

上例通过属性字典\_\_dict\_\_来访问对象的属性，本质上也是利用的反射的能力。

但是，上面的例子中，访问的方式不优雅，Python提供了内置的函数。

用上面的方法来修改上例的代码

|  |  |
| --- | --- |
| 内建函数 | 意义 |
| getattr(object,  name[, default]) | 通过name返回object的属性值。当属性不存在，将使用default返回，如果没有  default，则抛出AttributeError。name必须为字符串 |
| setattr(object,  name, value) | object的属性存在，则覆盖，不存在，新增 |
| hasattr(object,  name) | 判断对象是否有这个名字的属性，name必须为字符串 |



class  Point:

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_str\_\_(self):

return  "Point({},  {})".format(self.x,  self.y)

def  show(self):

print(self)

p1  =  Point(4,  5)

p2  =  Point(10,  10)

print(repr(p1),  repr(p2),  sep='\n')

print(p1.\_\_dict\_\_)

setattr(p1,  'y',  16)

setattr(p1,  'z',  10)

print(getattr(p1,  '\_\_dict\_\_'))

#  动态调用方法

if  hasattr(p1,  'show'):

getattr(p1,  'show')()

#  动态增加方法

#  为类增加方法

if  not  hasattr(Point,  'add'):

setattr(Point,  'add',  lambda  self,other:  Point(self.x  +  other.x,  self.y  +  other.y))

print(Point.add)

print(p1.add)

print(p1.add(p2))  #  绑定

#  为实例增加方法，未绑定

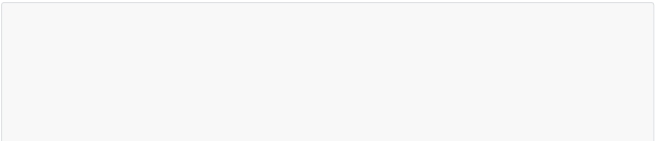
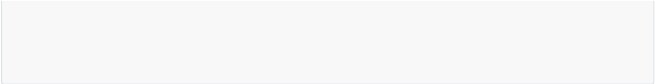
if  not  hasattr(p1,  'sub'):

setattr(p1,  'sub',  lambda  self,  other:  Point(self.x  -  other.x,  self.y  -  other.y))

print(p1.sub(p1,  p1))

print(p1.sub)

#  add在谁里面，sub在谁里面



print(p1.\_\_dict\_\_)

print(Point.\_\_dict\_\_)

思考

这种动态增加属性的方式和装饰器修饰一个类、Mixin方式的差异？

这种动态增删属性的方式是运行时改变类或者实例的方式，但是装饰器或Mixin都是定义时就决定了，因此反射能

力具有更大的灵活性。

练习

命令分发器，通过名称找对应的函数执行。

思路：名称找对象的方法

class  Dispatcher:

def  \_\_init\_\_(self):

pass

def  reg(self,  name,  fn):

setattr(self,  name,  fn)

def  run(self):

while  True:

cmd  =  input('>>>').strip()

if  cmd  ==  'quit':

break

getattr(self,  cmd,  lambda  :print('Unknown  Cmd  {}'.format(cmd)))()

dis  =  Dispatcher()

dis.reg('ls',  lambda  :print('ls'))

dis.run()

上例中使用getattr方法找到对象的属性的方式，比自己维护一个字典来建立名称和函数之间的关系的方式好多了。

反射相关的魔术方法

\_\_getattr\_\_()  、  \_\_setattr\_\_()  、  \_\_delattr\_\_()  这三个魔术方法，分别测试这三个方法

\_\_getattr\_\_()

class  Base:

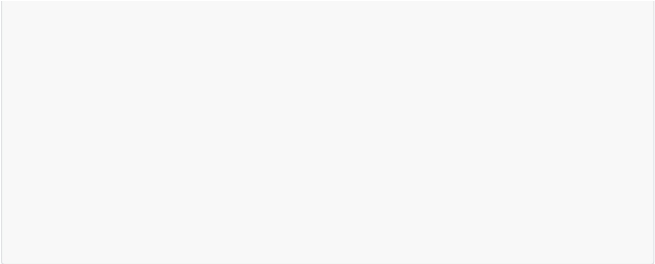
n  =  0

class  Point(Base):

z  =  6

def  \_\_init\_\_(self,  x,  y):

self.x  =  x



self.y  =  y

def  show(self):

print(self.x,  self.y)

def  \_\_getattr\_\_(self,  item):

return  "missing  {}".format(item)

p1  =  Point(4,5)

print(p1.x)

print(p1.z)

print(p1.n)

print(p1.t)  #  missing

一个类的属性会按照继承关系找，如果找不到，就会执行   \_\_getattr\_\_()   方法，如果没有这个方法，就会抛出

AttributeError异常表示找不到属性。

查找属性顺序为：

instance.\_\_dict\_\_  -->  instance.\_\_class\_\_.\_\_dict\_\_  -->  继承的祖先类（直到object）的\_\_dict\_\_  ---找不到-->  调

用\_\_getattr\_\_()

\_\_setattr\_\_()

class  Base:

n  =  0

class  Point(Base):

z  =  6

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  show(self):

print(self.x,  self.y)

def  \_\_getattr\_\_(self,  item):

return  "missing  {}".format(item)

def  \_\_setattr\_\_(self,  key,  value):

print("setattr  {}={}".format(key,value))

p1  =  Point(4,5)

print(p1.x)  #  missing，why

print(p1.z)

print(p1.n)

print(p1.t)  #  missing

p1.x  =  50

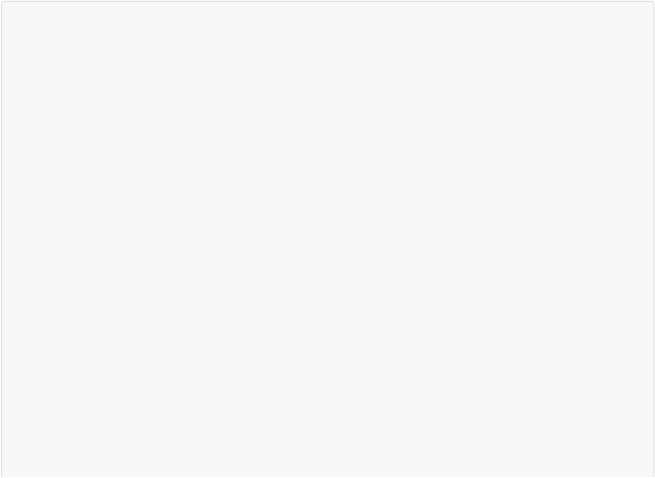
print(p1.\_\_dict\_\_)

p1.\_\_dict\_\_['x']  =  60

print(p1.\_\_dict\_\_)

print(p1.x)

实例通过.点号设置属性，例如self.x = x，就会调用  \_\_setattr\_\_()  ，属性要加到实例的  \_\_dict\_\_  中，就需要自己



完成。

class  Point(Base):

z  =  6

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  show(self):

print(self.x,  self.y)

def  \_\_getattr\_\_(self,  item):

return  "missing  {}".format(item)

def  \_\_setattr\_\_(self,  key,  value):

print("setattr  {}={}".format(key,value))

self.\_\_dict\_\_[key]  =  value  #  操作字典

\_\_setattr\_\_()  方法，可以拦截对实例属性的增加、修改操作，如果要设置生效，需要自己操作实例的

\_\_dict\_\_  。

例子：  \_\_getattr\_\_  和  \_\_setattr\_\_  综合使用

class  B:

b  =  200

class  A(B):

z  =  100

d  =  {}

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

setattr(self,  'y',  y)

self.\_\_dict\_\_['a']  =  5

def  \_\_getattr\_\_(self,  item):

print('~~~~~~~~~~~',  item)

return  self.d[item]

def  \_\_setattr\_\_(self,  key,  value):

print(key)

print(value)

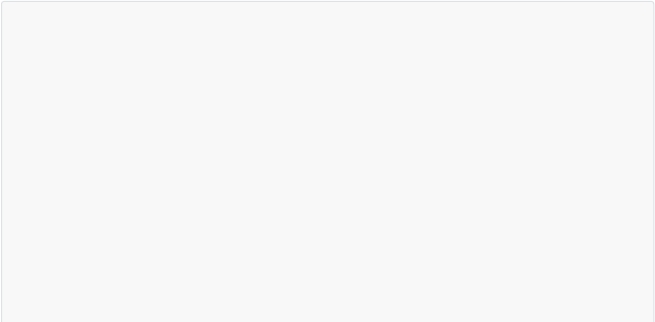
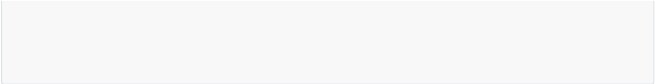
self.d[key]  =  value

def  \_\_delattr\_\_(self,  item):

print('can  not  del  {}'.format(item))

a  =  A(4,  5)

print(a.\_\_dict\_\_)



print(A.\_\_dict\_\_)

print(a.x,  a.y)

print(a.a)

\_\_delattr\_\_()

class  Point:

Z  =  5

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_delattr\_\_(self,  item):

print('Can  not  del  {}'.format(item))

p  =  Point(14,  5)

del  p.x

p.z  =  15

del  p.z

del  p.Z

print(Point.\_\_dict\_\_)

print(p.\_\_dict\_\_)

del  Point.Z

print(Point.\_\_dict\_\_)

可以阻止通过实例来删除属性的操作。但是通过类依然可以删除属性。

\_\_getattribute\_\_

class  Base:

n  =  0

class  Point(Base):

z  =  6

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_getattr\_\_(self,  item):

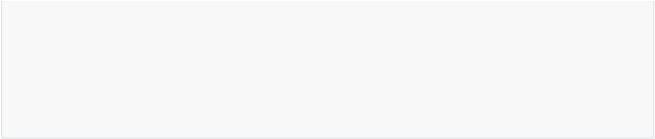
return  "missing  {}".format(item)

def  \_\_getattribute\_\_(self,  item):

return  item

p1  =  Point(4,5)

print(p1.\_\_dict\_\_)



print(p1.x)

print(p1.z)

print(p1.n)

print(p1.t)

print(Point.\_\_dict\_\_)

print(Point.z)

实例的所有的属性访问，第一个都会调用  \_\_getattribute\_\_  方法，它阻止了属性的查找，该方法应该返回（计算

后的）值或者抛出一个AttributeError异常。

它的return值将作为属性查找的结果。

如果抛出AttributeError异常，则会直接调用  \_\_getattr\_\_  方法，因为表示属性没有找到。

class  Base:

n  =  0

class  Point(Base):

z  =  6

def  \_\_init\_\_(self,  x,  y):

self.x  =  x

self.y  =  y

def  \_\_getattr\_\_(self,  item):

return  "missing  {}".format(item)

def  \_\_getattribute\_\_(self,  item):

#raise  AttributeError("Not  Found")

#pass

#return  self.\_\_dict\_\_[item]

return  object.\_\_getattribute\_\_(self,  item)

p1  =  Point(4,5)

print(p1.\_\_dict\_\_)

print(p1.x)

print(p1.z)

print(p1.n)

print(p1.t)

print(Point.\_\_dict\_\_)

print(Point.z)

\_\_getattribute\_\_  方法中为了避免在该方法中无限的递归，它的实现应该永远调用基类的同名方法以访问需要的

任何属性，例如  object.\_\_getattribute\_\_(self,  name)  。

注意，除非你明确地知道  \_\_getattribute\_\_  方法用来做什么，否则不要使用它。

总结

属性查找顺序：

|  |  |
| --- | --- |
| 魔术方法 | 意义 |
| \_\_getattr\_\_() | 当通过搜索实例、实例的类及祖先类查不到属性，就会调用此方法 |
| \_\_setattr\_\_() | 通过 . 访问实例属性，进行增加、修改都要调用它 |
| \_\_delattr\_\_() | 当通过实例来删除属性时调用此方法 |
| \_\_getattribute\_\_ | 实例所有的属性调用都从这个方法开始 |



实例调用\_\_getattribute\_\_()  -->  instance.\_\_dict\_\_  -->  instance.\_\_class\_\_.\_\_dict\_\_  -->  继承的祖先类（直到

object）的\_\_dict\_\_  -->  调用\_\_getattr\_\_()