理解Python对象的属性和描述器

Posted on 2017年9月5日 by laixintao 1 Comment

对于Python对象的属性是如何读取的，我一直存在一些疑问。对对象的属性赋值，什么时候直接指向一个新的对象，什么时候会直接抛出AttributeError错误，什么时候会通过Descriptor？Python的descriptor是怎么工作的？如果对a.x进行赋值，那么a.x不是应该直接指向一个新的对象吗？但是如果x是一个descriptor实例，为什么不会直接指向新对象而是执行__get__方法？经过一番研究和实验尝试，我大体明白了这个过程。

__getattr__ __getattribute__和__setattr__

对于对象的属性，默认的行为是对对象的字典（即`__dict__`）进行get set delete操作。比如说，对a.x查找x属性，默认的搜索顺序是a.__dict__['x']，然后是type(a).__dict__['x']，然后怼type(a)的父类(metaclass除外）继续查找。如果查找不到，就会执行特殊的方法。

```
# -*- coding: utf-8 -*-

class Foo(object):
    class_foo = 'world'
    def __init__(self):
        self.foo = 'hello'
    def __getattr__(self, name):
        print("Foo get attr run..." + self + name)


class Bar(Foo):
    def __getattr__(self, name):
        print("Bar get attr run..." + name)


bar = Bar()
print bar.foo  # hello
print bar.class_foo  # world
```

`__getattr__`只有在当对象的属性找不到的时候被调用。

```
  class LazyDB(object):
      def __init__(self):
          self.exists = 5
      def __getattr__(self, name):
          value = 'Value for %s' % name
          setattr(self, name, value)
          return value
```

```
  data = LazyDB()
  print('Before:', data.__dict__)
  print('foo:   ', data.foo)
```

```
  print('After: ', data.__dict__)
  >>>
  Before: {'exists': 5}
  foo:    Value for foo
  After:  {'exists': 5, 'foo': 'Value for foo'}
```

`__getattribute__` 每次都会调用这个方法拿到对象的属性，即使对象存在。

```
class ValidatingDB(object):
      def __init__(self):
          self.exists = 5
      def __getattribute__(self, name):
          print('Called __getattribute__(%s)' % name)
          try:
              return super().__getattribute__(name)
          except AttributeError:
              value = 'Value for %s' % name
              setattr(self, name, value)
              return value
  data = ValidatingDB()
  print('exists:', data.exists)
  print('foo:   ', data.foo)
  print('foo:   ', data.foo)
  >>>
  Called __getattribute__(exists)
  exists: 5
  Called __getattribute__(foo)
  foo:    Value for foo
  Called __getattribute__(foo)
  foo:    Value for foo
```

`__setattr__`每次在对象设置属性的时候都会调用。

判断对象的属性是否存在用的是内置函数`hasattr`。`hasattr`是C语言实现的，看了一下源代码，发现自己看不懂。不过搜索顺序和本节开头我说的一样。以后再去研究下源代码吧。

总结一下，取得一个对象的属性，默认的行为是：

1. 查找对象的__dict__
2. 如果没有，就查找对象的class的__dict__,即`type(a).__dict__['x']`
3. 如果没有，就查找父类class的__dict__
4. 如果没有，就执行`__getattr__`（如果定义了的话）
5. 否则就抛出`AttributeError`

对一个对象赋值，默认的行为是：

1. 如果定义了`__set__`方法，会通过`__setattr__`赋值
2. 否则会更新对象的`__dict__`

但是，如果对象的属性是一个Descriptor的话，会改变这种默认行为。

Python的Descriptor

对象的属性可以通过方法来定义特殊的行为。下面的代码，`Homework.grade`可以像普通属性一样使用。

```
 class Homework(object):
       def __init__(self):
           self._grade = 0
       @property
       def grade(self):
           return self._grade
       @grade.setter
       def grade(self, value):
           if not (0 <= value <= 100):
               raise ValueError('Grade must be between 0 and 100')
               self._grade = value
```

但是，如果有很多这样的属性，就要定义很多setter和getter方法。于是，就有了可以通用的Descriptor。

```
 class Grade(object):
       def __get__(*args, **kwargs):
           #...
       def __set__(*args, **kwargs):
           #...


class Exam(object):
       # Class attributes
       math_grade = Grade()
       writing_grade = Grade()
       science_grade = Grade()
```

Descriptor是Python的内置实现，一旦对象的某个属性是一个Descriptor实例，那么这个对象的读取和赋值将会使用Descriptor定义的相关方法。如果对象的`__dict__`和Descriptor同时有相同名字的，那么Descriptor的行为会优先。

```
# -*- coding: utf-8 -*-

class Descriptor(object):
    def __init__(self, name='x'):
        self.value = 0
        self.name = name
    def __get__(self, obj, type=None):
        print "get call"
        return self.value
```

```
    def __set__(self, obj, value):
        print "set call"
        self.value = value


class Foo(object):
    x = Descriptor()

foo = Foo()
print foo.x
foo.x = 200
print foo.x
print foo.__dict__
foo.__dict__['x'] = 500
print foo.__dict__
print foo.x

# --------------
# output
# get call
# 0
# set call
# get call
# 200
# {}
# {'x': 500}
# get call
# 200
```

实现了`__get__()`和`__set__()`方法的叫做data descriptor,只定义了`__get__()`的叫做non-data descriptor(通常用于method，本文后面有相应的解释）。上文提到，data descriptor优先级高于对象的`__dict__`但是non-data descriptor的优先级低于data descriptor。上面的代码删掉`__set__()`将会是另一番表现。

```
# -*- coding: utf-8 -*-

class Descriptor(object):
    def __init__(self, name='x'):
        self.value = 0
        self.name = name
    def __get__(self, obj, type=None):
        print "get call"
        return self.value


class Foo(object):
    x = Descriptor()

```

```
foo = Foo()
print foo.x
foo.x = 200
print foo.x
print foo.__dict__
foo.__dict__['x'] = 500
print foo.__dict__
print foo.x

# --------------
# output
# get call
# 0
# 200
# {'x': 200}
# {'x': 500}
# 500
```

如果需要一个“只读”的属性，只需要将`__set__()`抛出一个`AttributeError`即可。只定义`__set__()`也可以称作一个data descriptor。

调用关系

对象和类的调用有所不同。

对象的调用在`object.__getattribute__()`，将`b.x`转换成`type(b).__dict__['x'].__get__(b, type(b))`，然后引用的顺序和上文提到的那样，首先是data descriptor，然后是对象的属性，然后是non-data descriptor。

对于类的调用，由`type.__getattribute__()`将`B.x`转换成`B.__dict__['x'].__get__(None, B)`。Python实现如下：

```
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

需要注意的一点是，**Descriptor默认是由`__getattribute__()`调用的，如果覆盖`__getattribute__()`将会使Descriptor失效。**

Function，ClassMethod和StaticMethod

看起来这和本文内容没有什么关系，但其实Python中对象和函数的绑定，其原理就是Descriptor。

在Python中，方法（method）和函数（function）并没有实质的区别，只不过method的第一个参数是对象（或者类）。Class的`__dict__`中把method当做function一样存储，第一个参数预留出来作为`self`。为了支持方法调用，function默认有一个`__get__()`实现。也就是说，**所有的function都是**

non-data descriptor，返回bound method(对象调用）或unbound method(类调用)。用纯Python实现，如下。

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f               # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f               # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

bound和unbound method虽然表现为两种不同的类型，但是在C源代码里，是同一种实现。如果第一个参数`im_self`是NULL，就是unbound method，如果`im_self`有值，那么就是bound method。

总结：Non-data descriptor提供了将函数绑定成方法的作用。Non-data descriptor将`obj.f(*args)`转化成`f(obj, *args)`，将`klass.f(*args)`转化成`f(*args)`。如下表。

Transformation	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

可以看到，staticmethod并没有什么转化，和function几乎没有什么差别。因为staticmethod的推荐用法就是**将逻辑相关，但是数据不相关的functions打包组织起来。**通过函数调用、对象调用、方法调用都没有什么区别。staticmethod的纯python实现如下。

```
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

classmethod用于那些适合通过类调用的函数，例如工厂函数等。与类自身的数据有关系，但是和实际的对象没有关系。例如，Dict类将可迭代的对象生成字典，默认值为None。

```
class Dict(object):
    . . .
```

```
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)

>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

classmethod的纯Python实现如下。

```
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```

最后的话

一开始只是对对象的属性有些疑问，查来查去发现还是官方文档最靠谱。然后认识了Descriptor，最后发现这并不是少见的trick，而是Python中的最常见对象——function时时刻刻都在用它。从官方文档中能学到不少东西呢。另外看似平常、普通的东西背后，可能蕴含了非常智慧和简洁的设计。

相关阅读

1. hasattr的陷阱
2. Effective Python: Item 31
3. Descriptor HowTo Guide

Categories: Python

Tags: classmethod, Descriptor, function, Python, staticmethod, 设计模式, 面向对象