

## Re: instance attributes not inherited?

---

*Source:* <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2006-01/msg02486.html>

---

- *From:* [bokr@xxxxxx](mailto:bokr@xxxxxx) (Bengt Richter)
  - *Date:* Mon, 16 Jan 2006 10:05:15 GMT
- 

On Sun, 15 Jan 2006 20:37:36 -0500, "John M. Gabriele" <[john\\_sips\\_tez@xxxxxxxxxxx](mailto:john_sips_tez@xxxxxxxxxxx)> wrote:

>David Hirschfield wrote:

>> Nothing's wrong with python's oop inheritance, you just need to know  
>> that the parent class' `__init__` is not automatically called from a  
>> subclass' `__init__`. Just change your code to do that step, and you'll be  
>> fine:

>>

>> class Parent( object ):

>> def `__init__`( self ):

>> self.x = 9

>>

>>

>> class Child( Parent ):

>> def `__init__`( self ):

>> super(Child,self).`__init__`()

Nit: Someone is posting with source using tabs ;-/ The above super should be indented from def.

>> print "Inside Child.`__init__`()"

>>

>> -David

>>

>

>How does it help that Parent.`__init__` gets called? That call simply

>>would create a temporary Parent object, right? I don't see how it

Wrong ;-) Calling `__init__` does not create the object, it operates on an

object that has already been created. There is nothing special about

a user-written `__init__` method other than the name and that is is automatically

invoked under usual conditions. Child.`__new__` is the method that creates the instance,

but since Child doesn't have that method, it gets it by inheritance from Parent, which

in turn doesn't have one, so it inherits it from its base class (object).

>should help (even though it *does* indeed work).

>

>Why do we need to pass self along in that call to super()? Shouldn't

>the class name be enough for super() to find the right superclass object?

self is the instance that needs to appear as the argument of the `__init__` call,

so it has to come from somewhere. Actually, it would make more sense to leave out

the class than to leave out self, since you can get the class from `type(self)` for

typical super usage (see custom `simple_super` at end below [1])

## Re: instance attributes not inherited?

```
>>> class Parent( object ):
... def __init__( self ):
... self.x = 9
... print 'Inside Parent.__init__()'
...
>>> class Child( Parent ):
... def __init__( self ):
... sup = super(Child,self)
... sup.__init__()
... self.sup = sup
... print "Inside Child.__init__()"
...
>>> c = Child()
Inside Parent.__init__()
Inside Child.__init__()
>>> c.sup
<super: <class 'Child'>, <Child object>>
```

sup is an object that will present an attribute namespace that it implements by looking for attributes in the inheritance hierarchy of the instance argument (child) but starting one step beyond the normal first place to look, which is the instance's class that was passed to super. For a normal method such as `__init__`, it will form the bound method when you evaluate the `__init__` attribute of the super object:

```
>>> c.sup.__init__
<bound method Child.__init__ of <__main__.Child object at 0x02EF3AAC>>
```

a bound method has the first argument bound in, and when you call the bound method without an argument, the underlying function gets called with the actual first argument (self).

```
>>> c.sup.__init__()
Inside Parent.__init__()
```

The method resolution order lists the base classes of a particular class in the order they will be searched for a method. super skips the current class, since it is shadowing the rest. The whole list:

```
>>> type(c).mro()
[<class '__main__.Child'>, <class '__main__.Parent'>, <type 'object'>]
```

Skipping to where super will start looking:

```
>>> type(c).mro()[1]
<class '__main__.Parent'>
```

If you get the `__init__` attribute from the class, as opposed to as an attribute of the instance, you get an UNbound method:

```
>>> type(c).mro()[1].__init__
<unbound method Parent.__init__>
```

## Re: instance attributes not inherited?

which can be called with the instance as the first argument:

```
>>> type(c).mro()[1].__init__(c)
Inside Parent.__init__()
```

If you want to, you can access the actual function behind the unbound method:

```
>>> type(c).mro()[1].__init__.im_func
<function __init__ at 0x02EEADBC>
```

And form a bound method:

```
>>> type(c).mro()[1].__init__.im_func.__get__(c, type(c))
<bound method Child.__init__ of <__main__.Child object at 0x02EF3AAC>>
```

Which you can then call, just like we did sup.\_\_init\_\_ above:

```
>>> type(c).mro()[1].__init__.im_func.__get__(c, type(c))()
Inside Parent.__init__()
```

Another way to spell this is:

```
>>> Parent.__init__.im_func
<function __init__ at 0x02EEADBC>
>>> Parent.__init__.im_func.__get__(c, type(c))
<bound method Child.__init__ of <__main__.Child object at 0x02EF3AAC>>
>>> Parent.__init__.im_func.__get__(c, type(c))()
Inside Parent.__init__()
```

Or without forming the unbound method and using im\_func to get the function (looking instead for the \_\_init\_\_ function per se in the Parent class dict):

```
>>> Parent.__dict__['__init__']
<function __init__ at 0x02EEADBC>
>>> Parent.__dict__['__init__'].__get__(c, type(c))
<bound method Child.__init__ of <__main__.Child object at 0x02EF3AAC>>
>>> Parent.__dict__['__init__'].__get__(c, type(c))()
Inside Parent.__init__()
```

An attribute with a \_\_get\_\_ method (which any normal function has) is a descriptor, and depending on various logic will have its \_\_get\_\_ method called with the object whose attribute is apparently being accessed. Much of python's magic is implemented via descriptors, so you may want to read about it ;-)

[1] If we wanted to, we could write our own simple\_super. E.g.,

```
>>> class simple_super(object):
... def __init__(self, inst): self.inst = inst
... def __getattr__(self, attr):
```

Re: instance attributes not inherited?

Re: instance attributes not inherited?

```
... inst = object.__getattr__(self, 'inst')
... try: return getattr(type(inst).mro()[1], attr).im_func.__get__(
... inst, type(inst))
... except AttributeError:
... return object.__getattr__(self, attr)
...
>>> class Parent( object ):
... def __init__( self ):
... self.x = 9
... print 'Inside Parent.__init__()'
...
>>> class Child( Parent ):
... def __init__( self ):
... sup = simple_super(self)
... sup.__init__()
... self.sup = sup
... print "Inside Child.__init__()"
...
>>> c = Child()
Inside Parent.__init__()
Inside Child.__init__()
>>> c.sup
<__main__.simple_super object at 0x02EF80EC>
>>> c.sup.__init__
<bound method Child.__init__ of <__main__.Child object at 0x02EF80CC>>
>>> c.sup.__init__()
Inside Parent.__init__()
```

I don't know if this helps ;-)

Regards,  
Bengt Richter

---

• **Follow-Ups:**

- ◆ **Posting examples with tabs**  
◇ From: Duncan Booth

• **References:**

- ◆ **instance attributes not inherited?**  
◇ From: John M. Gabriele
- ◆ **Re: instance attributes not inherited?**  
◇ From: David Hirschfield
- ◆ **Re: instance attributes not inherited?**  
◇ From: John M. Gabriele

- Prev by Date: **Re: XML Writer in wxPython**
- Next by Date: **Re: [ANN] pysqlite 2.1.0 released**
- Previous by thread: **Re: instance attributes not inherited?**

Re: instance attributes not inherited?

- Next by thread: *Posting examples with tabs*
- Index(es):
  - ◆ *Date*
  - ◆ *Thread*