OOP in Python after 2.2

(or: Fun with Descriptors)

Author: Michael Hudson
Email:mwh@python.net

PythonUK 2004, part of the ACCU Conference 2004

Questions

What versions of Python are people using?

Who's heard of "new-style classes"?

Who's read Guido's "descrintro.html":

`http://www.python.org/2.2.2/descrintro.html`

What about the article about the new MRO algorithm in 2.3:

`http://www.python.org/2.3/mro.html`

Who would say they understand "new-style classes"?

Last Minute Slide

Who went to Alex's talk yesterday?

Differences you can expect:

- A more lesiurely pace.
- Perhaps a little more opinionated advice.
- Considerably more pedantry :-)

Making a new-style class (1/4)

Three ways (ish):

- Derive from a new-style class (object, if no other)
- Use a __metaclass__ class variable.
- Use a __metaclass__ global.

Making a new-style class (2/4)

Derive from a new-style class:

```
class MyClass(object):
    ...
```

This is the usual way of creating a new-style class.

Can also do:

```
class MyDict(dict):
    ...
```

(subclasses of builtin types are new-style classes too).

Making a new-style class (3/4)

You can use a __metaclass__ class variable:

```
class MyClass:
    __metaclass__ = type
    ...
```

Usually only resort to this when you want a more exotic metaclass than 'type' (more later).

Making a new-style class (4/4)

The final way of causing a class statement to create a new-style class is to set __metaclass__ as a module-level variable:

```
__metaclass__ = type
...
class MyClass:
    ...
```

(note this only has an effect if there are no base classes).

I don't like this: it's a non-local effect.

Extra Opinion Slide #1

When you get this far:

```
class MyClass
             ^
```

Don't type that colon yet; instead carry on to:

```
class MyClass(object):
                      ^
```

The way new-style classes are different (1/2)

New-style classes are instances of the type 'type':

```
>>> isinstance(MyNewStyleClass, type)
True
```

Old-style classes are not:

```
>>> isinstance(MyOldStyleClass, type)
False
```

That's not it, of course: new style classes have vastly better support for multiple inheritance and take part in the descriptor dance...

The way new-style classes are different (2/2)

A key way in which new-style classes are different is their support for **descriptors** (hence the subtitle of the talk).

Descriptors live in type objects and describe and control access to the attributes of the type's instances.

In true Pythonic style, a descriptor is just a (new-style) object that implements any part of the "descriptor protocol", which is a collection of three method signatures.

Recognising a descriptor

```
class MyDescriptor(object):
  def __get__(self, ob, cls=None):
    """controls reading from an attribute"""
  def __set__(self, ob, value):
    """controls writing to an attribute"""
  def __delete__(self, ob):
    """controls deleting an attribute"""
```

In practice implementing __delete__ seems to be fairly rare.

It is relatively common for a descriptor to only implement __get__ (these are sometimes call *non-data descriptors* or, less correctly, *method* descriptors).

An example (1/3)

```
class MyClass(object):
    def method(self):
        print self

instance = MyClass()
instance.method()
```

Huh? Looks normal, right? Well, yes, but it's still an example of using descriptors: in Python after 2.2 *functions* are descriptors.

An example (2/3)

Evaluating:

```
instance.method
```

is done by object.__getattribute__, which behaves somewhat (but not entirely) like this:

```
class object:
  def __getattribute__(self, attr):
    try: return self.__dict__[attr]
    except KeyError:
      val = type(self).__dict__[attr]
      if val is a descriptor:
        return val.__get__(self, type(self))
      else:
        return val
```

An Example (3/3)

The final piece of the puzzle is functions' __get__ method:

```
class function(object):
    ...
    def __get__(self, ob, cls=None):
        return <bound method self on ob>
```

(In previous versions of Python the tp_getattr method for class objects checked explicitly for functions and made them into bound methods, so descriptors are an enormous generalization of this behaviour).

A Wrinkle

I said the object.__getattribute__ presented above wasn't entirely equivalent to the one Python actually uses. What did I mean by that?

A shallow answer is that the version above ignores inheritance. A deeper difference is that Python allows a *data descriptor* to override the contents of self.__dict__.

A data descriptor is one that defines __set__ (or __delete__, but as I said, this doesn't seem to arise).

Motivation

At first, this seems a slightly surprising design decision.

The reason is symmetry: a data descriptor must necessarily override the contents of __dict__ for attribute *setting*, so it makes sense that they override the contents of __dict__ for *getting* attributes as well.

Almost The Full PyObject_GenericGetAttr

This still ignores inheritance:

```
class object:
  def __getattribute__(self, attr):
    t = type(self)
    descr = t.__dict__.get(attr, NULL)
    if descr is a data descriptor:
      return descr.__get__(self, t)
    try:
      return self.__dict__[attr]
    except KeyError:
      if descr != NULL:
        if descr is a descriptor:
          return descr.__get__(self, t)
        else:
          return descr
      raise AttributeError, attr
```

Built-in Descriptors

Python comes with a variety of descriptor types, including:

- Functions (as mentioned above)
- classmethod (__get__ binds a method to the class, not the instance)
- staticmethod (__get__ ignores both the class and the instance)
- property (each of __get__, __set__ and __delete__ calls a user supplied function)
- super (more later)

and a few others that are only used by types implemented in C.

Another Example (1/3)

Suppose you have a class with an attribute called 'oldname' and you've decided that 'newname' is a much better name.

But the 'oldname' attribute was part of the public API of your class, so you can't just make the change without winding up your users.

What you can do is use a descriptor -- a property in fact -- to redirect reads and writes of 'oldname' to 'newname'.

Another Example (2/3)

Here's one way:

```
class MyClass(object):
  def __init__(self, val):
    self.newname = val
  def _get_oldname(ob):
    # could put warning here
    return ob.newname
  def _set_oldname(ob, val):
    # could put warning here
    ob.newname = val
  oldname = property(_get_oldname, _set_oldname)
  del _get_oldname, _set_oldname
```

Another Example (3/3)

And in usage:

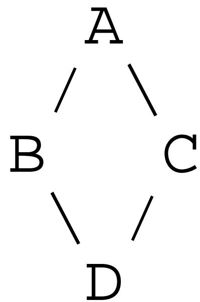
```
->> m = MyClass(1)
->> m.newname
1
->> m.oldname = 2
->> m.newname
2
->> m.oldname
2
```

Extra Opinion Slide #2

I didn't really realise this until I came to write these slides, but I don't actually use the 'property' builtin very often, I tend to implement custom descriptors myself... but maybe I'm just strange.

Method Resolution

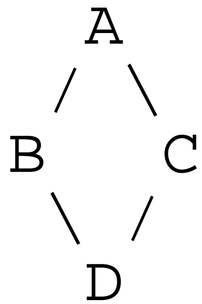
Another way new-style classes are different is the way method resolution is performed in the case of multiple inheritance. Given an inheritance hierarchy, say the infamous "inheritance diamond":



the *method resolution order* (MRO) of the class D is the order the bases are searched in in response to a request for an attribute of D (so perhaps it should be ARO, or Class Precedence List, but ...).

Method Resolution for Classic Classes

Classic classes search bases in "depth-first left-to-right order", so for the diamond:

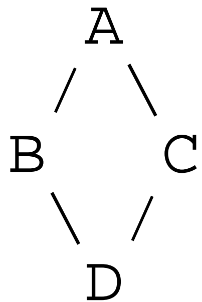


the MRO of D -- the order attributes are searched for -- goes [D, B, A, C, A].

Notice that the class A appears twice in here...

Method Resolution for New-Style Classes (1/2)

New-style classes *linearize* the MRO at class construction time and store this in the __mro__ attribute; in the diamond situation:



it is [D, B, C, A, object].

Two (fairly obvious) invariants of __mro__ are:

```
D.__mro__[0] is D
D.__mro__[-1] is object
```

(all new-style classes ultimately inherit from object).

Method Resolution for New-Style Classes (2/2)

The algorithm used to compute the MRO changed in version 2.3 -- but it's quite hard to tell the difference. The new algorithm -- which has the inscrutable name of "C3" -- enforces (and is defined by) three (hence C"3") invariants:

- local precedence order
- monotonicity
- something about the "extended precedence graph"...

I'll only explain what these mean if the audience is looking for more technical jargon :-) but these are all things you want.

Python 2.3's rules are the first for multiple inheritance I've come across that don't scare me.

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html

The Full PyObject_GenericGetAttr

So, to take inheritance into account, we just replace the line:

```
descr = t.__dict__.get(attr, NULL)
```

with:

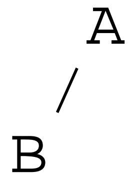
```
descr = type_lookup(t, attr)
```

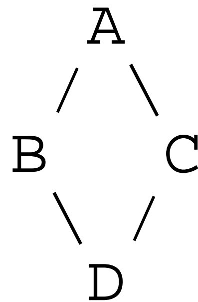
where type_lookup is defined as:

```
def type_lookup(t, attr):
    for t2 in t.__mro__:
        if attr in t2.__dict__:
            return t2.__dict__[attr]
    return NULL
```

super and Cooperative Super Calls (1/8)

The essential point of all this is that class in an inheritance hierarchy like B in the one on the left here:

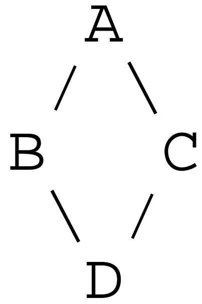




can make provisions for being turned into a diamond like that on the right. And this situation can't be avoided in the world of new-style classes: all new-style classes inherit from object, so as soon as you inherit from two or more new-style classes, the inheritance DAG is not a tree.

super and Cooperative Super Calls (2/8)

Recall the diamond:

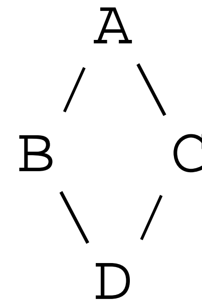


And assume that each class implements a method 'meth', and each class wants to call its superclass' implementation.

super and Cooperative Super Calls (3/8)

So, in code:

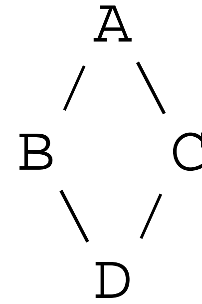
```
class A(object):
  def meth(self):
    return "A"
class B(A):
  def meth(self):
    return <superclass's result> + "B"
class C(A):
  def meth(self):
    return <superclass's result> + "C"
class D(B, C):
  def meth(self):
    return <superclass's result> + "D"
```



super and Cooperative Super Calls (4/8)

A way that doesn't work:

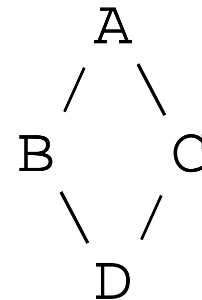
```
class A(object):
  def meth(self):
    return "A"
class B(A):
  def meth(self):
    return A.meth(self) + "B"
class C(A):
  def meth(self):
    return A.meth(self) + "C"
class D(B, C):
  def meth(self):
    return B.meth(self) + "D"
```



super and Cooperative Super Calls (5/8)

The way that works:

```
class A(object):
  def meth(self):
    return "A"
class B(A):
  def meth(self):
    return super(B, self).meth() + "B"
class C(A):
  def meth(self):
    return super(C, self).meth() + "C"
class D(B, C):
  def meth(self):
    return super(D, self).meth() + "D"
```



super and Cooperative Super Calls (6/8)

Why does this work? A super object is like a proxy for the supplied instance: looking up attributes on the super object is similar to looking up attributes on the instance. Not exactly the same -- that wouldn't be very interesting, after all.

For a start, super ignores instance attributes -- you only use a super object when you're looking for a class attribute.

The major difference -- and, indeed, the point of super -- is that it examines the __mro__ of the type of the supplied instance, and only looks in the __dict__s of classes that occur *after* the supplied class in this MRO.

super and Cooperative Super Calls (7/8)

Another way of thinking about super() is that in:

```
class Derived(Base):
    def meth(self, arg):
        ...
        super(Derived, self).meth(arg)
        ...
```

the "super(Derived, self).meth(arg)" calls the method *that would have been called if meth() hadn't been overriden in Derived*.

super and Cooperative Super Calls (8/8)

What's the take-home message here? Basically, when you're implementing a method and want to call the superclass' version, just use super(). That way your class will be more useful if incorporated into a multiple inheritance scenario, even if you have no plans on doing such a thing yourself.

This is why it's a *cooperative* super call -- it depends on the cooperation of all classes involved.

Much of this is inspired by the book 'Putting Metaclasses to Work' by Forman and Danforth published by Addsion-Welsey. This information would be more useful if the book wasn't out of print.

Extra Opinion Slide #3

Just use super(), alright?

Implementing super

```
class super(object):
  def __init__(self, cls, ob):
    self.cls = cls
    self.ob = ob
  def __getattribute__(self, attr):
    ob = self.__dict__['ob']
    obcls = type(ob)
    mro = obcls.__mro__
    i = mro.index(self.__dict__['cls'])
    for c in mro[i+1:]:
      if attr in c.__dict__:
        r = c.__dict__['attr']
        if hasattr(r, '__get__'):
          r = r.__get__(ob, obcls)
        return r
    raise AttributeError, attr
```

Can You Spot A Wart

There are various ways in which this implementation differs from the one built in to Python 2.2 and later (like error checking...) but can you see the problem in the line:

```
mro = obcls.__mro__
```

?

I Can Spot A Wart

It's insane to do so, but it's possible to write stuff like this:

```
class OddBall(object):
    __mro__ = 1
```

Then the line

```
mro = obcls.__mro__
```

will retrieve the value of this class variable __mro__, whereas the builtin version just accesses the tp_mro field of type objects directly.

We can't behave like this in pure Python can we? CAN WE?

I Can Also Remove This Wart

Well, how can we ever access the tp_mro field of type objects, which is what accessing the __mro__ attribute usually does? Well, access to tp_mro is controlled by a descriptor -- which shouldn't be a surprise -- and that descriptor is still there, in type.__dict__.

So replacing:

```
mro = obcls.__mro__
```

with:

```
mro_descr = type.__dict__['__mro__']
mro = mro_descr.__get__(obcls)
```

fixes this (rather silly) buglet.

The new __new__ method

Another change for new-style classes is the ability to override the class constructor (if you were calling __init__ the constructor, better stop: it's more properly called an initializer).

An __init__ method is only able to modify the object that has just been created, whereas if you override __new__ you can return a different object than that which would be returned by default.

__new__ is automatically converted to a staticmethod at class creation time, but it is called with the type of object being created as the first parameter (so it looks a bit like a classmethod).

An Example of __new__

One occasion where you have to override __new__ is when you are subclassing an immutable type, such as int:

```
>>> class EvenInt(int):
...     def __new__(cls, i):
...         return super(EvenInt, cls). \
...                 __new__(cls, i//2*2)
...
>>> EvenInt(1)
0
>>> EvenInt(25)
24
```

It is also often useful when implementing metaclasses...

Metaclasses (1/2)

Recall, if you've read it, Guido's famously mind-exploding essay about metaclasses in Python 1.5.

The key to the way metaclasses worked in older Python's was a check during the execution of a class statement that the "type of the base class is callable". This check is no longer present in Python 2.2 -- because almost all types are callable!

In particular the type of old-style class objects and the builtin 'type' -- the type of new-style classes and builtin types -- are callable.

Metaclasses (2/2)

In a class statement, a certain object is called to create the class object with three arguments:

- the name of the new class
- a tuple containing the supplied bases (may be empty)
- the namespace that resulted from executing the class body.

How is this "certain object" determined?

Finding the Class Constructor

A __metaclass__ class variable always wins.

If there is none such, the type of the first supplied base class is used.

If there are no supplied bases, a module level variable __metaclass__ is used, if present.

If there is no other class constructor to choose, the type of old-style classes is used.

So the 'Making a New-Style Class' slides earlier were describing the ways to arrange for the class constructor to be the builtin 'type' object.

A Detour

So let's run that one again: a class statement such as:

```
class C(B1, B2):
  ...stuff...
```

ends up having the same effect as:

```
C = <something>('C', (B1, B2), {stuff})
```

To me this is very "Pythonic":

- Looks declarative
- Isn't.

Scary Stuff

Almost every class statement ends up calling either the type of old-style classes or the builtin 'type' to build the new class, but as intimated above, much more is possible.

Using a __metaclass__ class variable, one can arrange for any callable at all to be called, and even if (as usually happens) it's the type of the first base class that gets called, one can deviously arrange for a type to return things other than instances of itself when called!

Possible doesn't mean wise, of course: it's generally a bad idea to have things that look like class statements doing something utterly unrelated.

Just Plain Odd Stuff

That said, I just can't resist this:

```
>>> class F(slice(1, 2, 3)): pass
...
>>> F
slice('F', (slice(1, 2, 3),),
      {'__module__': '__main__'})
```

The slice builtin type takes one to three arbitrary arguments; they're not usually much like these, in all honesty.

Back to matters more sensible...

Extra Opinion Slide #4

Don't reach for metaclasses too soon (a particular tendency of my own).

Keep your metaclasses sane. Something that looks like a class definition should behave like a class definition.

Uses for Metaclasses

An inconvience you sometimes have to deal with in writing descriptors is that a descriptor can't inherently know the name it has been attached to a class by, so you occasionally end up with code somewhat like:

```
class MyClass(object):
    attr = MyFunkyDescriptor('attr')
```

(for example a descriptor that forwards writes on to other objects might want to know this).

This duplication of information is a little fragile, and just plain offends me :-)

A Helpful Metaclass (1/2)

```
class MyFunkyDescriptor(object):
  def set_name(self, name):
    self.name = name
   ...

class MyMeta(type):
  def __new__(cls, name, bases, ns):
    for k, v in ns.iteritems():
      if isinstance(v, MyFunkyDescriptor):
        v.set_name(k)
    return super(MyMeta, cls).__new__(
      cls, name, bases, ns)

class MyClass:
  __metaclass__ = MyMeta
  attr = MyFunkyDescriptor()
```

A Helpful Metaclass (2/2)

This is a fairly common pattern for metaclasses: footle around with the contents of the supplied namespace in __new__, but return basically a vanilla new-style class.

Finding magic methods (1/4)

One difference between new-style and classic classes that occasionally confuses is where magic methods -- __repr__, say -- are looked for. Watch:

```
>>> class Classic:
...  def __init__(self):
...    self.__repr__ = lambda :'foo'
...
>>> class New(object):
...  def __init__(self):
...    self.__repr__ = lambda :'foo'
...
>>> print Classic()
foo
>>> print New()
<__main__.New object at 0x45e450>
```

Finding magic methods (2/4)

This means that repr is no longer defined like this:

```
def repr(ob): return ob.__repr__()
```

Instead it's more like this:

```
def repr(ob): return type(ob).__repr__(ob)
```

This seems like a complication, and is occasionally critized as such. But in fact, it's an inescapable consequence of unification of types and classes.

Finding magic methods (3/4)

The difficulty is that, *now that classes are instances of type*,

```
X.__repr__
```

is ambiguous if X is a new-style class: is it the bound method that computes the string representation of the object X or is it the unbound method that computes the string representation of *instances* of X? It gets even more mind bending if X is actually the object 'type'...

Finding magic methods (4/4)

As alluded to above, Python's solution is to only consider methods on the *type* of new-style instances when searching for 'magic' methods.

(If you now think a bit, you will see why it is essential that 'non-data descriptors' are overriden by more 'local' descriptors).

Extra Opinion Slide #5

If this change affected you, I don't want your code.

Credits

Thanks should go to:

- Armin Rigo, Thomas Wouters and Guido van Rossum for proofreading and advice,
- David Goodger for the docutils document processing system and Richard Jones for its PythonPoint writer,
- Reportlab for PythonPoint itself,
- the ACCU and Andy Robinson for organizing the conference and inviting me to speak.