Metaclass programming in Python, Part 3

Metaprogramming without metaclasses

Level: Intermediate

David Mertz, Ph.D. (mertz@gnosis.cx), Developer, Gnosis Software, Inc.
Michele Simionato (mis6+@pitt.edu), Physicist, University of Pittsburgh

25 Sep 2007

Too much cleverness in programming makes designs more complicated, code more fragile, learning curves steeper, and worst of all, it makes debugging harder. Michele and David feel, in part, responsible for some excesses of cleverness that followed the enthusiastic reception of their earlier articles on Python metaclasses. In this article, they attempt to make amends, by helping programmers eschew cleverness.

Introduction

Last year, I attended the EuroPython 2006 conference. The conference was good, the organization perfect, the talks of very high level, the people extremely nice. Nonetheless, I noticed something of a disturbing trend in the Python community that prompted this article. Almost simultaneously, my co-author David Mertz was reflecting on a similar issue with some submitted patches to Gnosis Utilities. The trend at issue is the trend towards *cleverness*. Unfortunately, whereas cleverness in the Python community was once largely confined to Zope and Twisted, it now is appearing everywhere.

We have nothing against cleverness in experimental projects and learning exercises. Our gripe is with cleverness in production frameworks that we are forced to cope with as users. In this article, we hope to make a small contribution away from cleverness, at least in an area where we have some expertise, that being metaclass abuses.

For this article, we take a ruthless stance: we consider *metaclass abuse* any usage of a metaclass where you could have solved the same problem equally well without a custom metaclass. Of course, the guilt of the authors is obvious here: our earlier installments on metaclasses in Python helped popularize their usage. *Nostra culpa*.

One of the most common metaprogramming scenarios is the creation of classes with attributes and methods that are dynamically generated. Contrary to popular belief, this is a job where most of the time you *do not need* and you *do not want* a custom metaclass.

This article is intended for two sets of readers: average programmers who would benefit from knowing a few meta-programming tricks but are scared off by brain-melting concepts; and clever programmers who are too clever and should know better. The problem for the latter is that it is easy to be clever, whereas it takes a lot of time to become unclever. For instance, it took us a few months to understand how to use metaclasses, but a few years to understand how *not* to use them.

About class initialization

During class creation, attributes and methods of classes are set once and for all. Or rather, in Python, methods and attributes can be changed at nearly any point, but only if naughty programmers sacrifice transparency.

In various common situations, you may want to create classes in more dynamic ways than simply running static code for their creation. For instance, you may want to set some default class attributes according to parameters read from a configuration file; or you may want to set class properties according to the fields in a database table. The easiest way to dynamically customize class behavior uses an imperative style: first create the class, then add methods and attributes.

For example, an excellent programmer of our acquaintance, Anand Pillai, has proposed a path to Gnosis Utilities' subpackage `gnosis.xml.objectify` that does exactly this. A base class called `gnosis.xml.objectify._XO_` that is specialized (at runtime) to hold "xml node objects" is "decorated" with a number of enhanced behaviors, like so:

Listing 1. Dynamic enhancement of a base class

```
setattr(_XO_, 'orig_tagname', orig_tagname)
setattr(_XO_, 'findelem', findelem)
setattr(_XO_, 'XPath', XPath)
setattr(_XO_, 'change_pcdata', change_pcdata)
setattr(_XO_,'addChild',addChild)
```

You might think, reasonably enough, that the same enhancement can be accomplished simply by subclassing the XO base class. True, in one sense, but Anand has provided about two dozen possible enhancements, and particular users might want some of them, but *not* others. There are too many permutations to easily create subclasses for every enhancement scenario. Still, the above code is not exactly *pretty*. You could accomplish the above sort of job with a custom metaclass, attached to XO, but with behavior determined dynamically. But that brings us back to the excessive cleverness (and opacity) that we hope to avoid.

A clean, and non-ugly, solution to the above need might be to add class decorators to Python. If we had those, we might write code similar to this:

Listing 2. Adding class decorators to Python

```
features = [('XPath',XPath), ('addChild',addChild), ('is_root',is_root)]
@enhance(features)
class _XO_plus(gnosis.xml.objectify._XO_): pass
gnosis.xml.objectify._XO_ = _XO_plus
```

That syntax, however, is a thing of the future, if it becomes available at all.

When metaclasses become complicated

It might seem like all the fuss in this paper so far is about nothing. Why not, for example, just define the metaclass of XO as `Enhance`, and be done with it. `Enhance.__init__()` can happily add whatever capabilities are needed for the particular use in question. This might look like so:

Listing 3. Defining XO as Enhance

```
class _XO_plus(gnosis.xml.objectify._XO_):
      __metaclass__ = Enhance
      features = [('XPath',XPath), ('addChild',addChild)]
gnosis.xml.objectify._XO_ = _XO_plus
```

Unfortunately, things are not so simple once you start to worry about inheritance. Once you have defined a custom metaclass for your base class, all the derived classes will inherit the metaclass, so the initialization code will be run on all derived classes, magically and implicitly. This may be fine in specific circumstances (for instance, suppose you have to register in your framework all the classes you define: using a metaclass ensures that you cannot forget to register a derived class), however, in many cases you may not like this behavior because:

- You believe that *explicit is better than implicit*.
- The derived classes have the same dynamic class attributes of the base class. Setting them again for each derived class is a waste, since they would be available anyway by inheritance. This may be an especially significant issue if the initialization code is slow or computationally expensive. You might add a check in the metaclass code to see if the attributes were already set in a parent class, but this adds plumbing and it does not give real control on a per-class basis.
- A custom metaclass will make your classes somewhat magic and nonstandard: you may not want to increase your chances to incur in metaclass conflicts, issues with "__slots__", fights with (Zope) extension classes, and other guru-level intricacies. Metaclasses are more fragile than many people realize. We have rarely used them for production code, even after four years of usage in experimental code.
- You feel that a custom metaclasses is overkill for the simple job of class initialization, and you would rather use a simpler solution.

In other words, you should use a custom metaclass only when your real intention is to have code running on derived classes without users of those classes noticing it. If this is not your case, skip the metaclass and make your life (and that of your users) happier.

The classinitializer decorator

What we present in the rest of this paper might be accused of cleverness. But the cleverness need not burden users, just us authors. Readers can do something much akin to the hypothetical (non-ugly) class decorator we propose, but without encountering the inheritance and metaclass conflict issues the metaclass approach raises. The "deep magic" decorator we give in full later generally just enhances the straightforward (but slightly ugly) imperative approach, and is "morally equivalent" to this:

Listing 4. Imperative approach

```
def Enhance(cls, **kw):
    for k, v in kw.iteritems():
        setattr(cls, k, v)
class ClassToBeInitialized(object):
    pass
Enhance(ClassToBeInitialized, a=1, b=2)
```

The above imperative enhancer is not so bad. But it has a few drawbacks: it make you repeat the class name; readability is suboptimal since class definition and class initialization are separated -- for long class definitions you can miss the last line; and it feels wrong to first define something and then immediately mutate it.

The `classinitializer` decorator provides a declarative solution. The decorator converts `Enhance(cls,**kw)` into a method that can be used in a class definition:

Listing 5. The magic decorator in basic operation

```
>>> @classinitializer # add magic to Enhance
... def Enhance(cls, **kw):
...     for k, v in kw.iteritems():
...         setattr(cls, k, v)
>>> class ClassToBeInitialized(object):
...     Enhance(a=1, b=2)
>>> ClassToBeInitialized.a
1
>>> ClassToBeInitialized.b
2
```

If you have used Zope interfaces, you may have seen examples of class initializers (`zope.interface.implements`). In fact, `classinitializer` is implemented by using a trick copied from `zope.interface.advice`, which credits Phillip J. Eby. The trick uses the "__metaclass__" hook, but it *does not use* a custom metaclass. `ClassToBeInitialized` keeps its original metaclass, i.e. the plain built-in metaclass `type` of new style classes:

```
>>> type(ClassToBeInitialized)
<type 'type'>
```

In principle, the trick also works for old style classes, and it would be easy to write an implementation keeping old style classes old style. However, since according to Guido "old style classes are morally deprecated", the current implementation converts old style classes into new style classes:

Listing 6. Promotion to newstyle

```
>>> class WasOldStyle:
...     Enhance(a=1, b=2)
>>> WasOldStyle.a, WasOldStyle.b
(1, 2)
>>> type(WasOldStyle)
<type 'type'>
```

One of the motivations for the `classinitializer` decorator is to hide the plumbing and to make mere mortals able to implement their own class initializers in an easy way, without knowing the details of how class creation works and the secrets of the _metaclass_ hook. The other motivation is that even for Python wizards it is very inconvenient to

rewrite the code managing the _metaclass_ hook every time one writes a new class initializer.

As a final note, let us point out that the decorated version of `Enhance` is smart enough to continue to work as a non-decorated version outside a class scope, provided that you pass to it an explicit class argument:

```
>>> Enhance(WasOldStyle, a=2)
>>> WasOldStyle.a
2
```

The (overly) deep magic

Here is the code for `classinitializer`. You do not need to understand it to use the decorator:

Listing 7. The classinitializer decorator

```
import sys
def classinitializer(proc):
   # basic idea stolen from zope.interface.advice, P.J. Eby
   def newproc(*args, **kw):
       frame = sys._getframe(1)
       if '__module__' in frame.f_locals and not \
           '__module__' in frame.f_code.co_varnames: # we are in a class
           if '__metaclass__' in frame.f_locals:
               raise SyntaxError("Don't use two class initializers or\n"
                 "a class initializer together with a __metaclass__ hook")
           def makecls(name, bases, dic):
               try:
                   cls = type(name, bases, dic)
               except TypeError, e:
                   if "can't have only classic bases" in str(e):
                       cls = type(name, bases + (object,), dic)
                   else:  # other strange errs, e.g. __slots__ conflicts
                       raise
               proc(cls, *args, **kw)
               return cls
           frame.f_locals["__metaclass__"] = makecls
       else:
           proc(*args, **kw)
 newproc.__name__ = proc.__name__
 newproc.__module__ = proc.__module__
 newproc.__doc__ = proc.__doc__
 newproc.__dict__ = proc.__dict__
 return newproc
```

From the implementation it is clear how class initializers work: when you call a class initializer inside a class, your are actually defining a _metaclass_ hook that will be called by the class' metaclass (typically `type`). The metaclass will create the class (as a new style one) and will pass it to the class initializer procedure.

Tricky points and caveats

Since class initializers (re)define the _metaclass_ hook, they don't play well with classes that define a _metaclass_ hook explicitly (as opposed to implicitly inheriting one). If a _metaclass_ hook is defined *after* the class initializer, it *silently* overrides it.

Listing 8. table project index.html home

```
>>> class C:
...     Enhance(a=1)
...     def __metaclass__(name, bases, dic):
...         cls = type(name, bases, dic)
...         print 'Enhance is silently ignored'
...         return cls
...
Enhance is silently ignored
>>> C.a
Traceback (most recent call last):
  ...
AttributeError: type object 'C' has no attribute 'a'
```

While unfortunate, there is no general solution to this issue; we simply document it. On the other hand, if you call a

class initializer *after* the _metaclass_ hook, you will get an exception:

Listing 9. Local metaclass raises an error

```
>>> class C:
...     def __metaclass__(name, bases, dic):
...         cls = type(name, bases, dic)
...         print 'calling explicit __metaclass__'
...         return cls
...     Enhance(a=1)
...
Traceback (most recent call last):
   ...
SyntaxError: Don't use two class initializers or
a class initializer together with a __metaclass__ hook
```

Raising an error is preferable to silently overriding your explicit _metaclass_ hook. As a consequence, you will get an error if you try to use two class initializers at the same time, or if you call the same one twice:

Listing 10. Doubled enhancement creates a problem

```
>>> class C:
...     Enhance(a=1)
...     Enhance(b=2)
Traceback (most recent call last):
  ...
SyntaxError: Don't use two class initializers or
a class initializer together with a__metaclass__ hook
```

On the plus side, all issues for inherited _metaclass_ hooks and for custom metaclasses are handled:

Listing 11. Happy to enhance inherited metaclass

```
>>> class B: # a base class with a custom metaclass
...     class __metaclass__(type):
...         pass
>>> class C(B): # class with both custom metaclass AND class initializer
...     Enhance(a=1)
>>> C.a
1
>>> type(C)
<class '_main.__metaclass__'>
```

The class initializer does not disturb the metaclass of `C`, which is the one inherited by base `B`, and the inherited metaclass does not disturb the class initializer, which does its job just fine. You would have run into trouble, instead, if you tried to call `Enhance` directly in the base class.

Putting it together

With all this machinery defined, customizing class initialization becomes rather straightforward, and elegant looking. It might be something as simple as:

Listing 12. Simplest form enhancement

```
class _XO_plus(gnosis.xml.objectify._XO_):
    Enhance(XPath=XPath, addChild=addChild, is_root=is_root)
gnosis.xml.objectify._XO_ = _XO_plus
```

This example still uses the "injection" which is somewhat superfluous to the general case; i.e. we put the enhanced class back into a specific name in the module namespace. It is necessary for the particular module, but will not be needed most of the time. In any case, the argument to to `Enhance()` need not be fixed in code as above, you can equally use `Enhance(**feature_set)` for something completely dynamic.

The other point to keep in mind is that your `Enhance()` function can do rather more than the simple version

suggested above. The decorator is more than happy to tweak more sophisticated enhancement functions. For example, here is one that adds "records" to a class:

Listing 13. Variations on class enhancement

```
@classinitializer
def def_properties(cls, schema):
    """
    Add properties to cls, according to the schema, which is a list
    of pairs (fieldname, typecast). A typecast is a
    callable converting the field value into a Python type.
    The initializer saves the attribute names in a list cls.fields
    and the typecasts in a list cls.types. Instances of cls are expected
    to have private attributes with names determined by the field names.
    """
    cls.fields = []
    cls.types = []
    for name, typecast in schema:
        if hasattr(cls, name): # avoid accidental overriding
            raise AttributeError('You are overriding %s!' % name)
        def getter(self, name=name):
            return getattr(self, '_' + name)
        def setter(self, value, name=name, typecast=typecast):
            setattr(self, '_' + name, typecast(value))
        setattr(cls, name, property(getter, setter))
        cls.fields.append(name)
        cls.types.append(typecast)
```

The differing concerns of (a) what is enhanced; (b) how the magic works; and (c) what the basic class itself does are kept orthogonal:

Listing 14. Customizing a record class

```
>>> class Article(object):
...    # fields and types are dynamically set by the initializer
...    def_properties([('title', str), ('author', str), ('date', date)])
...    def __init__(self, values): # add error checking if you like
...        for field, cast, value in zip(self.fields, self.types, values):
...            setattr(self, '_' + field, cast(value))

>>> a=Article(['How to use class initializers', 'M. Simionato', '2006-07-10'])
>>> a.title
'How to use class initializers'
>>> a.author
'M. Simionato'
>>> a.date
datetime.date(2006, 7, 10)
```

Resources

Learn

- The first installment of "Metaclass programming in Python" (developerWorks, February 2003) introduces metaclass programming concepts as compared to object-oriented concepts.
- "Metaclass programming in Python, Part 2" (developerWorks, August 2003) goes into more detail on the subtleties of Python metaclasses.
- "Charming Python: Decorators make magic easy" (developerWorks, December 2006) takes a look at the newest Python facility for meta-programming.
- Check out the code from which everything was born.
- Read Michele's "A simple and useful doctester for your documentation".
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.

- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the authors

David Mertz is owner and chief consultant for Gnosis Software, Inc. whose corporate slogan is "We Know Stuff!" (and we do). You can reach David at mertz@gnosis.cx; you can investigate all aspects of his life at his personal Web page. Check out his book, *Text Processing in Python* .

Michele Simionato is a plain, ordinary, theoretical physicist who was driven to Python by a quantum fluctuation that could well have passed without consequences had he not met David Mertz. He will let his readers judge the final outcome.

Share this....

Digg this story del.icio.us Slashdot it!