IBM

Metaclass programming in Python

Pushing object-oriented programming to the next level

Level: Introductory

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

26 Feb 2003

Most readers are already familiar with the concepts of object-oriented programming: inheritance, encapsulation, polymorphism. But the *creation* of objects of a given class, with certain parents, is usually thought of as a "just so" operation. It turns out that a number of new programming constructs become either easier, or possible at all, when you can customize the process of object creation. Metaclasses enable certain types of "aspect-oriented programming," for example, you can enhance classes with features like tracing capabilities, object persistence, exception logging, and more.

Review of object-oriented programming

Let's start with a 30-second review of just what OOP is. In an object-oriented programming language, you can define *classes*, whose purpose is to bundle together related data and behaviors. These classes can inherit some or all of their qualities from their *parents*, but they can also define attributes (data) or methods (behaviors) of their own. At the end of the process, classes generally act as templates for the creation of *instances* (at times also called simply *objects*). Different instances of the same class will typically have different data, but it will come in the same shape -- for example, the `Employee` objects `bob` and `jane` both have a `.salary` and a `.room_number`, but not the same room and salary as each other.

Some OOP languages, including Python, allow for objects to be *introspective* (also called *reflective*). That is, an introspective object is able to describe itself: What class does the instance belong to? What ancestors does that class have? What methods and attributes are available to the object? Introspection lets a function or method that handles objects make decisions based on what kind of object it is passed. Even without introspection, functions frequently branch based on instance data -- for example, the route to `jane.room_number` differs from that to `bob.room_number` because they are in different rooms. With introspection, you can also safely calculate the bonus `jane` gets, while skipping the calculation for `bob`, for example, because `jane` has a `.profit_share` attribute, or because `bob` is an instance of the subclass `Hourly(Employee)`.

More in this series

- "Metaclass programming in Python, Part 2" goes into more detail on the subtleties of Python metaclasses.
- "Metaclass programming in Python, Part 3" recommends avoiding overly clever custom metaclasses.
- Read more articles by David and Michele.

A metaprogramming rejoinder

The basic OOP system sketched above is quite powerful. But there is one element brushed over in the description: in Python (and other languages), classes are themselves objects that can be passed around and introspected. Since objects, as stated, are produced using classes as templates, what acts as a template for producing classes? The answer, of course, is *metaclasses*.

Python has always had metaclasses. But the machinery involved in metaclasses became much better exposed with Python 2.2. Specifically, with version 2.2, Python stopped being a language with just one special (mostly hidden) metaclass that created every class object. Now programmers can subclass the aboriginal metaclass `type` and even dynamically generate classes with varying metaclasses. Of course, just because you *can* manipulate metaclasses in Python 2.2, that does not explain why you might want to.

Moreover, you do not need to use custom metaclasses to manipulate the production of classes. A slightly less brain-melting concept is a *class factory*: An ordinary function can return a class that was dynamically created within the

function body. In traditional Python syntax, you can write:

```
Python 1.5.2 (#0, Jun 27 1999, 11:23:01) [...]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> def class_with_method(func):
...     class klass: pass
...     setattr(klass, func.__name__, func)
...     return klass
...
>>> def say_foo(self): print 'foo'
...
>>> Foo = class_with_method(say_foo)
>>> foo = Foo()
>>> foo.say_foo()
foo
```

The factory function `class_with_method()` dynamically creates and returns a class that contains the method/function passed into the factory. The class itself is manipulated within the function body before being returned. The `new` module provides a more concise spelling, but without the same options for custom code within the body of the class factory, for example:

```
>>> from new import classobj
>>> Foo2 = classobj('Foo2',(Foo,),{'bar':lambda self:'bar'})
>>> Foo2().bar()
'bar'
>>> Foo2().say_foo()
foo
```

In all these cases, the behaviors of the class (`Foo`, `Foo2`) are not directly written as code, but are instead created by calling functions at runtime, with dynamic arguments. And it should be emphasized that it is not merely the *instances* that are so dynamically created, but the *classes* themselves.

Metaclasses: a solution looking for a problem?

> Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why). -- Python Guru Tim Peters

Methods (of classes), like plain functions, can return objects. So in that sense it is obvious that class factories can be classes just as easily as they can be functions. In particular, Python 2.2+ provides a special class called `type` that is just such a class factory. Of course, readers will recognize `type()` as a less ambitious built-in function of older Python versions -- fortunately, the behaviors of the old `type()` function are maintained by the `type` class (in other words, `type(obj)` returns the type/class of the object `obj`). The new `type` class works as a class factory in just the same way that the function `new.classobj` long has:

```
>>> X = type('X',(),{'foo':lambda self:'foo'})
>>> X, X().foo()
(<class '__main__.X'>, 'foo')
```

But since `type` is now a (meta)class, you are free to subclass it:

```
>>> class ChattyType(type):
...     def __new__(cls, name, bases, dct):
...         print "Allocating memory for class", name
...         return type.__new__(cls, name, bases, dct)
...     def __init__(cls, name, bases, dct):
...         print "Init'ing (configuring) class", name
...         super(ChattyType, cls).__init__(name, bases, dct)
...
>>> X = ChattyType('X',(),{'foo':lambda self:'foo'})
Allocating memory for class X
Init'ing (configuring) class X
>>> X, X().foo()
```

```
(<class '__main__.X'>, 'foo')
```

The magic methods `.__new__()` and `.__init__()` are special, but in conceptually the same way they are for any other class. The `.__init__()` method lets you configure the created object; the `.__new__()` method lets you customize its allocation. The latter, of course, is not widely used, but exists for every Python 2.2 new-style class (usually inherited but not overridden).

There is one feature of `type` descendents to be careful about; it catches everyone who first plays with metaclasses. The first argument to methods is conventionally called `cls` rather than `self`, because the methods operate on the *produced* class, not the metaclass. Actually, there is nothing special about this; all methods attach to their instances, and the instance of a metaclass is a class. A non-special name makes this more obvious:

```
>>> class Printable(type):
...     def whoami(cls): print "I am a", cls.__name__
...
>>> Foo = Printable('Foo',(),{})
>>> Foo.whoami()
I am a Foo
>>> Printable.whoami()
Traceback (most recent call last):
TypeError:  unbound method whoami() [...]
```

All this surprisingly non-remarkable machinery comes with some syntax sugar that both makes working with metaclasses easier, and confuses new users. There are several elements to the extra syntax. The resolution order of these new variations is tricky though. Classes can inherit metaclasses from their ancestors -- notice that this is **not** the same thing as *having* metaclasses as ancestors (another common confusion). For old-style classes, defining a global `_metaclass_` variable can force a custom metaclass to be used. But most of the time, and the safest approach, is to set a `_metaclass_` class attribute for a class that wants to be created via a custom metaclass. You must set the variable in the class definition itself since the metaclass is not used *if* the attribute is set later (after the class object has already been created). For example:

```
>>> class Bar:
...     __metaclass__ = Printable
...     def foomethod(self): print 'foo'
...
>>> Bar.whoami()
I am a Bar
>>> Bar().foomethod()
foo
```

Solving problems with magic

So far, we have seen the basics of metaclasses. But putting metaclasses to work is more subtle. The challenge with utilizing metaclasses is that in typical OOP design, classes do not really *do* much. The inheritance structure of classes is useful to encapsulate and package data and methods, but it is typically instances that one works with in the concrete.

There are two general categories of programming tasks where we think metaclasses are genuinely valuable.

The first, and probably more common category is where you do not know at design time *exactly* what a class needs to do. Obviously, you will have some idea about it, but some particular detail might depend on information that is not available until later. "Later" itself can be of two sorts: (a) When a library module is used by an application; (b) At runtime when some situation exists. This category is close to what is often called "Aspect-Oriented Programming" (AOP). We'll show what we think is an elegant example:

```
% cat dump.py
#!/usr/bin/python
import sys
if len(sys.argv) > 2:
    module, metaklass  = sys.argv[1:3]
    m = __import__(module, globals(), locals(), [metaklass])
    __metaclass__ = getattr(m, metaklass)
```

```
class Data:
    def __init__(self):
        self.num = 38
        self.lst = ['a','b','c']
        self.str = 'spam'
    dumps   = lambda self: `self`
    __str__ = lambda self: self.dumps()

data = Data()
print data

% dump.py
<__main__.Data instance at 1686a0>
```

As you would expect, this application prints out a rather generic description of the `data` object (a conventional instance object). But if *runtime* arguments are passed to the application, we can get a rather different result:

```
% dump.py gnosis.magic MetaXMLPickler
<?xml version="1.0"?>
<!DOCTYPE PyObject SYSTEM "PyObjects.dtd">
<PyObject module="__main__" class="Data" id="720748">
<attr name="lst" type="list" id="980012" >
  <item type="string" value="a" />
  <item type="string" value="b" />
  <item type="string" value="c" />
</attr>
<attr name="num" type="numeric" value="38" />
<attr name="str" type="string" value="spam" />
</PyObject>
```

The particular example uses the serialization style of `gnosis.xml.pickle`, but the most current `gnosis.magic` package also contains metaclass serializers `MetaYamlDump`, `MetaPyPickler`, and `MetaPrettyPrint`. Moreover, a user of the `dump.py` "application" can impose the use of any "MetaPickler" desired, from any Python package that defines one. Writing an appropriate metaclass for this purpose will look something like:

```
class MetaPickler(type):
    "Metaclass for gnosis.xml.pickle serialization"
    def __init__(cls, name, bases, dict):
        from gnosis.xml.pickle import dumps
        super(MetaPickler, cls).__init__(name, bases, dict)
        setattr(cls, 'dumps', dumps)
```

The remarkable achievement of this arrangement is that the application programmer need have no knowledge about what serialization will be used -- nor even whether serialization or some other cross-sectional capability will be added **at the command-line**.

Perhaps the most common use of metaclasses is similar to that of MetaPicklers: adding, deleting, renaming, or substituting methods for those defined in the produced class. In our example, a "native" `Data.dump()` method is replaced by a different one from outside the application, at the time the class `Data` is created (and therefore in every subsequent instance).

More ways to solve problems with magic

There is a programming niche where classes are often more important than instances. For example, *declarative mini-languages* are Python libraries whose program logic is expressed directly in class declarations. David examines them in his article "Create declarative mini-languages". In such cases, using metaclasses to affect the process of class creation can be quite powerful.

One class-based declarative framework is `gnosis.xml.validity`. Under this framework, you declare a number of "validity classes" that express a set of constraints about valid XML documents. These declarations are very close to those contained in DTDs. For example, a "dissertation" document can be configured with the code:

```
from gnosis.xml.validity import *
class figure(EMPTY):      pass
```

```
class _mixedpara(Or):     _disjoins = (PCDATA, figure)
class paragraph(Some):    _type = _mixedpara
class title(PCDATA):      pass
class _paras(Some):       _type = paragraph
class chapter(Seq):       _order = (title, _paras)
class dissertation(Some): _type = chapter
```

If you try to instantiate the `dissertation` class without the right component subelements, a descriptive exception is raised; likewise for each of the subelements. The proper subelements will be generated from simpler arguments when there is only one unambiguous way of "lifting" the arguments to the correct types.

Even though validity classes are often (informally) based on a pre-existing DTD, instances of these classes print themselves as unadorned XML document fragments, for example:

```
>>> from simple_diss import *
>>> ch = LiftSeq(chapter, ('It Starts','When it began'))
>>> print ch
<chapter><title>It Starts</title>
<paragraph>When it began</paragraph>
</chapter>
```

By using a metaclass to create the validity classes, we can generate a DTD out of the class declarations themselves (and add an extra method to the classes while we do it):

```
>>> from gnosis.magic import DTDGenerator, \
...                          import_with_metaclass, \
...                          from_import
>>> d = import_with_metaclass('simple_diss',DTDGenerator)
>>> from_import(d,'**')
>>> ch = LiftSeq(chapter, ('It Starts','When it began'))
>>> print ch.with_internal_subset()
<?xml version='1.0'?>
<!DOCTYPE chapter [
<!ELEMENT figure EMPTY>
<!ELEMENT dissertation (chapter)+>
<!ELEMENT chapter (title,paragraph+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT paragraph ((#PCDATA|figure))+>
]>
<chapter><title>It Starts</title>
<paragraph>When it began</paragraph>
</chapter>
```

The package `gnosis.xml.validity` knows nothing about DTDs and internal subsets. Those concepts and capabilities are introduced entirely by the metaclass `DTDGenerator`, without *any* change made to either `gnosis.xml.validity` or `simple_diss.py`. `DTDGenerator` does not substitute its own `.__str__()` method into classes it produces -- you can still print the unadorned XML fragment -- but it a metaclass could easily modify such magic methods.

Meta conveniences

The package `gnosis.magic` contains several utilities for working with metaclasses, as well as some sample metaclasses you can use in aspect-oriented programming. The most important of these utilities is `import_with_metaclass()`. This function, utilized in the above example, lets you import a third-party module, but create all the module classes using a custom metaclass rather than `type`. Whatever new capability you might want to impose on that third-party module can be defined in a metaclass that you create (or get from somewhere else altogether). `gnosis.magic` contains some pluggable serialization metaclasses; some other package might contain tracing capabilities, or object persistence, or exception logging, or something else.

The `import_with_metclass()` function illustrates several qualities of metaclass programming:

```
def import_with_metaclass(modname, metaklass):
    "Module importer substituting custom metaclass"
    class Meta(object): __metaclass__ = metaklass
```

```
    dct = {'__module__':modname}
    mod = __import__(modname)
    for key, val in mod.__dict__.items():
        if inspect.isclass(val):
            setattr(mod, key, type(key,(val,Meta),dct))
    return mod
```

One notable style in this function is that an ordinary class `Meta` is produced using the specified metaclass. But once `Meta` is added as an ancestor, its descendent is also produced using the custom metaclass. In principle, a class like `Meta` could carry with it *both* a metaclass producer *and* a set of inheritable methods -- the two aspects of its bequest are orthogonal.

Resources

- A useful book on metaclasses is *Putting Metaclasses to Work* by Ira R. Forman and Scott Danforth (Addison-Wesley; 1999).

- For metaclasses in Python specifically, Guido van Rossum's essay, "Unifying types and classes in Python 2.2" is useful as well.

- Also by David on *developerWorks*, read:
  - "Guide to Python introspection"
  - "Create declarative mini-languages"
  - "XML Matters: Enforcing validity with the gnosis.xml.validity library"

- Don't know Tim Peters? You should! Begin with Tim's wiki page and end with reading news:comp.lang.python more regularly.

- New to AOP? You may find this "Introduction to Aspect-Oriented Programming" (PDF) by Ken Wing Kuen Lee of the Hong Kong University of Science and Technology interesting.

- Gregor Kiczales and his team at Xerox PARC coined the term aspect-oriented programming in the 1990s and championed it as a way to allow software programmers to spend more time writing code and less time correcting it.

- "Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP)" by Karl J. Lieberherr also describes AOP.

- You'll also find subject-oriented programming interesting. As described by the folks at IBM Research, it's essentially the same thing as aspect-oriented programming.

- Find and download the Gnosis utils, mentioned several times in this article, at David's site.

- Find more resources for Linux developers in the *developerWorks* Linux zone.

About the authors

David Mertz thought his brain would melt when he wrote about continuations or semi-coroutines, but he put the gooey mess back in his skull cavity and moved on to metaclasses. David may be reached at mertz@gnosis.cx; his life pored over at his personal Web page. Suggestions and recommendations on this, past, or future columns are welcome. Learn about his forthcoming 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.