Metaclass programming in Python, Part 2

Understanding the arcana of inheritance and instance creation

Level: Introductory

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

28 Aug 2003

Michele and David's initial *developerWorks* article on metaclass programming prompted quite a bit of feedback, some of it from perplexed readers trying to grasp the subtleties of Python metaclasses. This article revisits the working of metaclasses and their relation to other OOP concepts. It contrasts class instantiation with inheritance, distinguishes classmethods and metamethods, and explains and solves metaclass conflicts.

Metaclasses and their discontents

In our initial article on metaclass programming in Python, we introduced the concept of metaclasses, showed some of their power, and demonstrated their use in solving problems such as dynamic customization of classes and libraries at run-time.

That article proved quite popular, but there were elisions in our condensed summary. Certain details in the use of metaclasses merit further explanation. Based on the feedback of our readers and on discussions in comp.lang.python, we will address some of those trickier point in this second article. In particular, we think the following points are important for any programmer wanting to master metaclasses:

- Users must understand the differences and interactions between metaclass programming and traditional object-oriented programming (under both single and multiple inheritance).
- Python 2.2 added the built-in functions `staticmethod()` and `classmethod()` to create methods that do not require an instance object during invocation. To an extent, *classmethods* overlap in purpose with (meta)methods defined in metaclasses. But the precise similarities and differences have also generated confusion in the minds of many programmers.
- Users should understand the cause and the resolution of metatype conflicts. This becomes essential when you want to use more than one custom metaclass. We explain the concept of *composition* of metaclasses.

More in this series

- The first installment of "Metaclass programming in Python" introduces metaclass programming concepts as compared to object-oriented concepts.
- "Metaclass programming in Python, Part 3" recommends avoiding overly clever custom metaclasses.
- Read more articles by David and Michele.

Instantiation versus inheritance

Many programmers are confused about the difference between a metaclass and a base class. At the superficial level of "determining" a class, both look similar. But once you look deeper, the concepts drift apart.

Before presenting some examples, it is worth being precise about some nomenclature. An *instance* is a Python object that was "manufactured" by a class; the class acts as a sort of template for the instance. Every instance is an instance of *exactly one* class (but a class might have multiple instances). What we often call an instance object -- or perhaps a "simple instance" -- is "final" in the sense that it cannot act as a template for other objects (but it might still be a *factory* or a *delegate*, which serve overlapping purposes).

Some instance objects are themselves classes; and all classes are instances of a corresponding *metaclass*. Even classes only come into existence through the instantiation mechanism. Usually classes are instances of the built-in, standard metaclass `type`; it is only when we specify metaclasses other than `type` that we need to think about metaclass programming. We also call the class used to instantiate an object the *type* of that object.

Running *orthogonal* to the idea of instantiation is the notion of inheritance. Here, a class can have one or multiple

parents, not just one unique type. And parents can have parents, creating a transitive subclass relation, conveniently accessible with the built-in function `issubclass()`. For example, if we define a few classes and an instance:

Listing 1. Typical inheritance hierarchy

```
>>> class A(object): a1 = "A"
...
>>> class B(object): a2 = "B"
...
>>> class C(A,B):    a3 = "C(A,B)"
...
>>> class D(C):      a4 = "D(C)"
...
>>> d = D()
>>> d.a5 = "instance d of D"
```

Then we can test the relations:

Listing 2. Testing ancestry

```
>>> issubclass(D,C)
True
>>> issubclass(D,A)
True
>>> issubclass(A,B)
False
>>> issubclass(d,D)
[...]
TypeError: issubclass() arg 1 must be a class
```

The interesting question now -- the one necessary for understanding the contrast between superclasses and metaclasses -- is how an attribute like `d.attr` is resolved. For simplicity, we discuss only the standard look-rule, not the fallback to `.__getattr__()`. The first step in such resolution is to look in `d.__dict__` for the name `attr`. If found, that's that; but if not, something fancy needs to happen, such as:

```
>>> d.__dict__, d.a5, d.a1
({'a5': 'instance d'}, 'instance d', 'A')
```

The trick to finding an attribute that isn't attached to an instance is to look for it in the class of the instance, then after that in all the superclasses. The order in which superclasses are checked is called the *method resolution order* for the class. You can look at it with the (meta)method `.mro()` (but only from class objects):

```
>>> [k.__name__ for k in d.__class__.mro()]
['D', 'C', 'A', 'B', 'object']
```

In other words, the access to `d.attr` first looks in `d.__dict__`, then in `D.__dict__`, `C.__dict__`, `A.__dict__`, `B.__dict__`, and finally in `object.__dict__`. If the name is not found in any of those places, an `AttributeError` is raised.

Notice that metaclasses were never mentioned in the lookup procedure.

Metaclasses versus ancestors

Here is a simple example of normal inheritance. We define a `Noble` base class, with subclasses such as `Prince`, `Duke`, `Baron`, etc.

Listing 3. Attribute inheritance

```
>>> for s in "Power Wealth Beauty".split(): exec '%s="%s"'%(s,s)
...
>>> class Noble(object):      # ...in fairy tale world
...     attributes = Power, Wealth, Beauty
...
>>> class Prince(Noble):
...     pass
...
```

```
>>> Prince.attributes
('Power', 'Wealth', 'Beauty')
```

The class `Prince` inherits the attributes of the class `Noble`. An instance of `Prince` still follows the lookup chain discussed above:

Listing 4. Attributes in instances

```
>>> charles=Prince()
>>> charles.attributes        # ...remember, not the real world
('Power', 'Wealth', 'Beauty')
```

If the `Duke` class should happen to have a custom metaclasses, it can obtain some attributes that way:

```
>>> class Nobility(type): attributes = Power, Wealth, Beauty
...
>>> class Duke(object): __metaclass__ = Nobility
...
```

As well as being a class, `Duke` is an instance of the metaclass `Nobility`--attribute lookup proceeds as with any object:

```
>>> Duke.attributes
('Power', 'Wealth', 'Beauty')
```

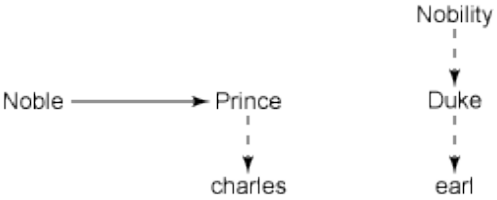
But `Nobility` is *not* a superclass of `Duke`, so there is no reason why an *instance* of `Duke` would find `Nobility.attributes`:

Listing 5. Attributes and metaclasses

```
>>> Duke.mro()
[<class '__main__.Duke'>, <type 'object'>]
>>> earl = Duke()
>>> earl.attributes
[...]
AttributeError: 'Duke' object has no attribute 'attributes'
```

The availability of metaclass attributes is not transitive; in other words, the attributes of a metaclass are available to its instances, but not to the instances of the instances. Just this is the main difference between metaclasses and superclasses. A diagram emphasizes the orthogonality of inheritance and instantiation:

Figure 1. Instantiation versus inheritance



Since `earl` still has a class, you can indirectly retrieve the attributes, however:

```
>>> earl.__class__.attributes
```

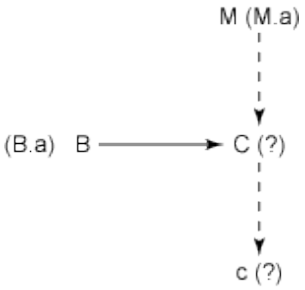
Figure 1 contrasts simple cases where *either* inheritance or metaclasses are involved, but not both. Sometimes, however, a class C has both a custom metaclass M and a base class B:

Listing 6. Combining metaclasses and superclasses

```
                >>> class M(type):
...     a = 'M.a'
...     x = 'M.x'
...
>>> class B(object): a = 'B.a'
...
>>> class C(B): __metaclass__=M
...
>>> c=C()
```

Graphically:

Figure 2. Combined superclass and metaclass



From the prior explanation, we could imagine that `C.a` would resolve to *either* `M.a` or `B.a`. As it turns out, lookup on a class follows its MRO before it looks in its instantiating metaclass:

Listing 7. Resolution with metaclasses and superclasses

```
>>> C.a, C.x
('B.a', 'M.x')
>>> c.a
'B.a'
>>> c.x
[...]
AttributeError: 'C' object has no attribute 'x'
```

You can still enforce a attribute value using a metaclass, you just need to set it on the class object being instantiated rather than as an attribute of the metaclass:

Listing 8. Setting attribute in metaclass

```
>>> class M(type):
...     def __init__(cls, *args):
...         cls.a = 'M.a'
...
>>> class C(B): __metaclass__=M
...
>>> C.a, C().a
('M.a', 'M.a')
```

More on class magic

The fact that the instantiation constraint is weaker than the inheritance constraint is essential for implementing the special methods like `.__new__()`, `.__init__()`, `.__str__()`, etc. We will discuss the `.__str__()` method; an analysis is similar for the other special methods.

Readers probably know that the printed representation of a class object can be modified by overriding its `.__str__()` method. In the same sense, the printed representation of a class can be modified by overriding the `.__str__()` methods of its metaclass. For instance:

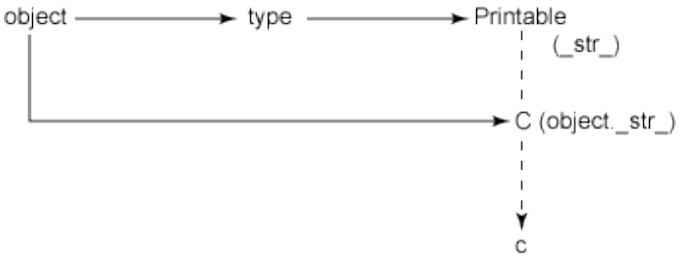
Listing 9. Customizing printout of a class

```
>>> class Printable(type):
...     def __str__(cls):
...         return "This is class %s" % cls.__name__
...
>>> class C(object): __metaclass__ = Printable
...
>>> print C       # equivalent to print Printable.__str__(C)
This is class C
>>> c = C()
>>> print c       # equivalent to print C.__str__(c)
```

```
<C object at 0x40380a6c>
```

The situation can be represented with the following diagram:

Figure 3. Metaclasses and magic methods



From the previous discussion, it is clear that the `.__str__()` method in `Printable` cannot override the `.__str__()` method in C, which is inherited from `object` and therefore has precedence; printing `c` still gives the standard result.

If C inherited its `.__str__()` method from `Printable` rather than from `object`, it would cause a problem: `C` instances do not have a `.__name__` attribute and printing `c` would generate an error. Of course, you could still define a `.__str__()` method in `C` that would change the way `c` prints.

Classmethods versus metamethods

Another common confusion arises between Python classmethods and methods defined in a metaclass, best called *metamethods*.

Consider this example:

Listing 10. Metamethods and classmethods

```
>>> class M(Printable):
...     def mm(cls):
...         return "I am a metamethod of %s" % cls.__name__
...
>>> class C(object):
...     __metaclass__=M
...     def cm(cls):
...         return "I am a classmethod of %s" % cls.__name__
...     cm=classmethod(cm)
...
>>> c=C()
```

Part of the confusion is due to the fact that in Smalltalk terminology, `C.mm` would be called a "class method of `C`." Python classmethods are a different beast, however.

The metamethod "mm" can be invoked from either the metaclass or from the class, but not from the instance. The classmethod can be called both from the class and from its instances (but does not exist in the metaclass).

Listing 11. Invoking a metamethod

```
>>> print M.mm(C)
I am a metamethod of C
>>> print C.mm()
I am a metamethod of C
>>> print c.mm()
[...]
AttributeError: 'C' object has no attribute 'mm'
>>> print C.cm()
I am a classmethod of C
>>> print c.cm()
I am a classmethod of C
```

Also, the metamethod is retrieved by `dir(M)` but not by `dir(C)` whereas the classmethod is retrieved by `dir(C)` and `dir(c)`.

You can only call the metaclass methods that are defined in the class MRO by dispatching on the metaclass (built-ins like `print` do this behind the scenes):

Listing 12. Magic metaclass method

```
>>> print C.__str__()
[...]
TypeError: descriptor '__str__' of 'object' object needs an argument
>>> print M.__str__(C)
This is class C
```

It is important to notice that this dispatch conflict is not limited to magic methods. If we change `C` by adding an attribute `C.mm`, the same issue exists (it does not matter if the name is a regular method, classmethod, staticmethod, or simple attribute):

Listing 13. Non-magic metaclass method

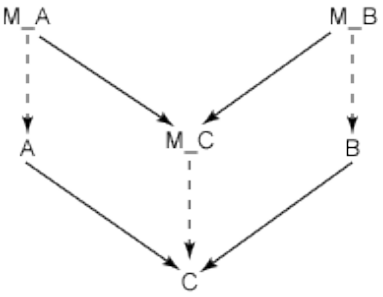
```
>>> C.mm=lambda self: "I am a regular method of %s" % self.__class__
>>> print C.mm()
[...]
TypeError: unbound method <lambda>() must be called with
    C instance as first argument (got nothing instead)
```

Conflicting metaclasses

Once you work seriously with metaclasses, you will be bitten at least once by metaclass/metatype conflicts. Consider a class `A` with metaclass `M_A` and a class `B` with metaclass `M_B`; suppose we derive `C` from `A` and `B`. The question is: what is the metaclass of `C`? Is it `M_A` or `M_B`?

The correct answer is `M_C`, where `M_C` is a metaclass that inherits from `M_A` and `M_B`, as in the following graph (in the Resources later in this article, see the link to *Putting metaclasses to work* for a discussion):

Figure 4. Avoiding the metaclass conflict



However, Python does not (yet) automatically create `M_C`. Instead, it raises a `TypeError`, warning the programmer of the conflict:

Listing 14. Metaclass conflicts

```
>>> class M_A(type): pass
...
>>> class M_B(type): pass
...
>>> class A(object): __metaclass__ = M_A
...
>>> class B(object): __metaclass__ = M_B
...
>>> class C(A,B): pass    # Error message less specific under 2.2
[...]
TypeError: metaclass conflict: the metaclass of a derived class must
```

```
    be a (non-strict) subclass of the metaclasses of all its bases
```

The metatype conflict can be avoided by manually creating the needed metaclass for `C`:

Listing 15. Manually resolving metaclass conflict

```
>>> M_AM_B = type("M_AM_B", (M_A,M_B), {})
>>> class C(A,B): __metaclass__ = M_AM_B
...
>>> type(C)
<class 'M_AM_B'>
```

The resolution of metatype conflicts becomes more complicated when you wish to "inject" additional metaclasses into a class, beyond those in its ancestors. Also, depending on the metaclasses of parent classes, redundant metaclasses can occur -- both identical metaclasses in different ancestors and superclass/subclass relationships among metaclasses. The module **`noconflict`** is available to help users resolve these issues in a robust and automatic way (see Resources).

Conclusion

A number of warnings and corner cases are discussed in this article. Working with metaclasses requires a certain degree of trial-and-error before the behavior becomes intuitive. However, the issues are by no means intractable -- this fairly short article touches on most of the pitfalls. Play with the cases yourself. You will find, at the end of the day, that whole new realms of program generalization are available with metaclasses; the gains are well worth the few dangers.

Resources

- The authors continue to recommend *Putting Metaclasses to Work* by Ira R. Forman, 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.
- Raymond Hettinger has written an excellent article on the descriptor protocol introduced in Python 2.2. Descriptors are a means of altering the behavior of attribute/method access, which is an interesting programming technique in itself. But of particular value relative to this article is Hettinger's explanation of the lookup chain that underlies Python's concept of OOP.
- Michele's `noconflict` module is discussed in the online Active State Python Cookbook. This module lets users automatically resolve metatype conflicts.
- The Gnosis Utilities library contains a number of tools for working with metaclasses, generally within the `gnosis.magic` subpackage. You may download the last stable version of the whole package from gnosis.cx.
- You can also browse the experimental branch, which includes a version of `noconflict`.
- Coauthor Michele has written an article on the new method resolution order (MRO) algorithm in Python 2.3. While most programmers can remain blissfully ignorant on the details of the changes, it is worthwhile for all Python programmers to understand the concept of MRO -- and perhaps have an inkling that better and worse approaches exist.
- The predecessor to this article is Metaclass programming in Python, Part 1 (*developerWorks*, February 2003).
- Guide to Python introspection (*developerWorks*, December 2002) showcases Python's introspection capabilities, from basic to advanced.
- Read David's Charming Python column on the *developerWorks* Linux zone.
- Find more articles about Linux and Linux programming in the *developerWorks* Linux zone.

About the authors

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. Now he has been trapped in Python's gravitational field. He will let his readers judge the final outcome. You can contact Michele at mis6+@pitt.edu, or you can read his Web site.

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-at-gnosis.cx; his life pored over at his personal Web page. Suggestions and recommendations on this, past, or future columns are welcome. David's book *Text Processing in Python* was recently published by Addison-Wesley; check it out.

Share this....