Metaclasses in Python 1.5

(A.k.a. The Killer Joke :-)

(*Postscript:* reading this essay is probably not the best way to understand the metaclass hook described here. See a message posted by Vladimir Marangozov which may give a gentler introduction to the matter. You may also want to search Deja News for messages with "metaclass" in the subject posted to comp.lang.python in July and August 1998.)

In previous Python releases (and still in 1.5), there is something called the ``Don Beaudry hook", after its inventor and champion. This allows C extensions to provide alternate class behavior, thereby allowing the Python class syntax to be used to define other class-like entities. Don Beaudry has used this in his infamous MESS package; Jim Fulton has used it in his Extension Classes package. (It has also been referred to as the ``Don Beaudry *hack*," but that's a misnomer. There's nothing hackish about it -- in fact, it is rather elegant and deep, even though there's something dark to it.)

(On first reading, you may want to skip directly to the examples in the section "Writing Metaclasses in Python" below, unless you want your head to explode.)

Documentation of the Don Beaudry hook has purposefully been kept minimal, since it is a feature of incredible power, and is easily abused. Basically, it checks whether the **type of the base class** is callable, and if so, it is called to create the new class.

Note the two indirection levels. Take a simple example:

```
class B:
    pass

class C(B):
    pass
```

Take a look at the second class definition, and try to fathom ``the type of the base class is callable."

(Types are not classes, by the way. See questions 4.2, 4.19 and in particular 6.22 in the Python FAQ for more on this topic.)

- The **base class** is B; this one's easy.
- Since B is a class, its type is ``class"; so the **type of the base class** is the type ``class". This is also known as types.ClassType, assuming the standard module `types` has been imported.
- Now is the type ``class" **callable**? No, because types (in core Python) are never callable. Classes are callable (calling a class creates a new instance) but types aren't.

So our conclusion is that in our example, the type of the base class (of C) is not callable. So the Don Beaudry hook does not apply, and the default class creation mechanism is used (which is also used when there is no base class). In fact, the Don Beaudry hook never applies when using only core Python, since the type of a core object is never callable.

So what do Don and Jim do in order to use Don's hook? Write an extension that defines at least two new Python object types. The first would be the type for ``class-like" objects usable as a base class, to trigger Don's hook. This type must be made callable. That's why we need a second type. Whether

an object is callable depends on its type. So whether a type object is callable depends on *its* type, which is a *meta-type*. (In core Python there is only one meta-type, the type ``type'' (types.TypeType), which is the type of all type objects, even itself.) A new meta-type must be defined that makes the type of the class-like objects callable. (Normally, a third type would also be needed, the new ``instance'' type, but this is not an absolute requirement -- the new class type could return an object of some existing type when invoked to create an instance.)

Still confused? Here's a simple device due to Don himself to explain metaclasses. Take a simple class definition; assume B is a special class that triggers Don's hook:

```
class C(B):
    a = 1
    b = 2
```

This can be though of as equivalent to:

```
C = type(B)('C', (B,), {'a': 1, 'b': 2})
```

If that's too dense for you, here's the same thing written out using temporary variables:

```
creator = type(B)               # The type of the base class
name = 'C'                      # The name of the new class
bases = (B,)                    # A tuple containing the base class(es)
namespace = {'a': 1, 'b': 2}    # The namespace of the class statement
C = creator(name, bases, namespace)
```

This is analogous to what happens without the Don Beaudry hook, except that in that case the creator function is set to the default class creator.

In either case, the creator is called with three arguments. The first one, *name*, is the name of the new class (as given at the top of the class statement). The *bases* argument is a tuple of base classes (a singleton tuple if there's only one base class, like the example). Finally, *namespace* is a dictionary containing the local variables collected during execution of the class statement.

Note that the contents of the namespace dictionary is simply whatever names were defined in the class statement. A little-known fact is that when Python executes a class statement, it enters a new local namespace, and all assignments and function definitions take place in this namespace. Thus, after executing the following class statement:

```
class C:
    a = 1
    def f(s): pass
```

the class namespace's contents would be {'a': 1, 'f': <function f ...>}.

But enough already about writing Python metaclasses in C; read the documentation of MESS or Extension Classes for more information.

Writing Metaclasses in Python

In Python 1.5, the requirement to write a C extension in order to write metaclasses has been dropped (though you can still do it, of course). In addition to the check ``is the type of the base class callable,'' there's a check ``does the base class have a __class__ attribute.'' If so, it is assumed that the __class__ attribute refers to a class.

Let's repeat our simple example from above:

```
class C(B):
```

```
    a = 1
    b = 2
```

Assuming B has a __class__ attribute, this translates into:

```
C = B.__class__('C', (B,), {'a': 1, 'b': 2})
```

This is exactly the same as before except that instead of type(B), B.__class__ is invoked. If you have read FAQ question 6.22 you will understand that while there is a big technical difference between type(B) and B.__class__, they play the same role at different abstraction levels. And perhaps at some point in the future they will really be the same thing (at which point you would be able to derive subclasses from built-in types).

At this point it may be worth mentioning that C.__class__ is the same object as B.__class__, i.e., C's metaclass is the same as B's metaclass. In other words, subclassing an existing class creates a new (meta)inststance of the base class's metaclass.

Going back to the example, the class B.__class__ is instantiated, passing its constructor the same three arguments that are passed to the default class constructor or to an extension's metaclass: *name*, *bases*, and *namespace*.

It is easy to be confused by what exactly happens when using a metaclass, because we lose the absolute distinction between classes and instances: a class is an instance of a metaclass (a ``metainstance''), but technically (i.e. in the eyes of the python runtime system), the metaclass is just a class, and the metainstance is just an instance. At the end of the class statement, the metaclass whose metainstance is used as a base class is instantiated, yielding a second metainstance (of the same metaclass). This metainstance is then used as a (normal, non-meta) class; instantiation of the class means calling the metainstance, and this will return a real instance. And what class is that an instance of? Conceptually, it is of course an instance of our metainstance; but in most cases the Python runtime system will see it as an instance of a a helper class used by the metaclass to implement its (non-meta) instances...

Hopefully an example will make things clearer. Let's presume we have a metaclass MetaClass1. It's helper class (for non-meta instances) is callled HelperClass1. We now (manually) instantiate MetaClass1 once to get an empty special base class:

```
BaseClass1 = MetaClass1("BaseClass1", (), {})
```

We can now use BaseClass1 as a base class in a class statement:

```
class MySpecialClass(BaseClass1):
    i = 1
    def f(s): pass
```

At this point, MySpecialClass is defined; it is a metainstance of MetaClass1 just like BaseClass1, and in fact the expression ``BaseClass1.__class__ == MySpecialClass.__class__ == MetaClass1'' yields true.

We are now ready to create instances of MySpecialClass. Let's assume that no constructor arguments are required:

```
x = MySpecialClass()
y = MySpecialClass()
print x.__class__, y.__class__
```

The print statement shows that x and y are instances of HelperClass1. How did this happen? MySpecialClass is an instance of MetaClass1 (``meta'' is irrelevant here); when an instance is called, its __call__ method is invoked, and presumably the __call__ method defined by MetaClass1 returns an instance of HelperClass1.

Now let's see how we could use metaclasses -- what can we do with metaclasses that we can't easily do without them? Here's one idea: a metaclass could automatically insert trace calls for all method calls. Let's first develop a simplified example, without support for inheritance or other ``advanced'' Python features (we'll add those later).

```
import types

class Tracing:
    def __init__(self, name, bases, namespace):
        """Create a new class."""
        self.__name__ = name
        self.__bases__ = bases
        self.__namespace__ = namespace
    def __call__(self):
        """Create a new instance."""
        return Instance(self)

class Instance:
    def __init__(self, klass):
        self.__klass__ = klass
    def __getattr__(self, name):
        try:
            value = self.__klass__.__namespace__[name]
        except KeyError:
            raise AttributeError, name
        if type(value) is not types.FunctionType:
            return value
        return BoundMethod(value, self)

class BoundMethod:
    def __init__(self, function, instance):
        self.function = function
        self.instance = instance
    def __call__(self, *args):
        print "calling", self.function, "for", self.instance, "with", args
        return apply(self.function, (self.instance,) + args)

Trace = Tracing('Trace', (), {})

class MyTracedClass(Trace):
    def method1(self, a):
        self.a = a
    def method2(self):
        return self.a

aninstance = MyTracedClass()

aninstance.method1(10)

print "the answer is %d" % aninstance.method2()
```

Confused already? The intention is to read this from top down. The Tracing class is the metaclass we're defining. Its structure is really simple.

- The __init__ method is invoked when a new Tracing instance is created, e.g. the definition of class MyTracedClass later in the example. It simply saves the class name, base classes and namespace as instance variables.
- The __call__ method is invoked when a Tracing instance is called, e.g. the creation of aninstance later in the example. It returns an instance of the class Instance, which is defined next.

The class Instance is the class used for all instances of classes built using the Tracing metaclass, e.g. aninstance. It has two methods:

- The __init__ method is invoked from the Tracing.__call__ method above to initialize a new instance. It saves the class reference as an instance variable. It uses a funny name because the user's instance variables (e.g. self.a later in the example) live in the same namespace.
- The __getattr__ method is invoked whenever the user code references an attribute of the instance that is not an instance variable (nor a class variable; but except for __init__ and __getattr__ there are no class variables). It will be called, for example, when aninstance.method1 is referenced in the example, with self set to aninstance and name set to the string "method1".

The __getattr__ method looks the name up in the __namespace__ dictionary. If it isn't found, it raises an AttributeError exception. (In a more realistic example, it would first have to look through the base classes as well.) If it is found, there are two possibilities: it's either a function or it isn't. If it's not a function, it is assumed to be a class variable, and its value is returned. If it's a function, we have to ``wrap'' it in instance of yet another helper class, BoundMethod.

The BoundMethod class is needed to implement a familiar feature: when a method is defined, it has an initial argument, self, which is automatically bound to the relevant instance when it is called. For example, aninstance.method1(10) is equivalent to method1(aninstance, 10). In the example if this call, first a temporary BoundMethod instance is created with the following constructor call: temp = BoundMethod(method1, aninstance); then this instance is called as temp(10). After the call, the temporary instance is discarded.

- The __init__ method is invoked for the constructor call BoundMethod(method1, aninstance). It simply saves away its arguments.
- The __call__ method is invoked when the bound method instance is called, as in temp(10). It needs to call method1(aninstance, 10). However, even though self.function is now method1 and self.instance is aninstance, it can't call self.function(self.instance, args) directly, because it should work regardless of the number of arguments passed. (For simplicity, support for keyword arguments has been omitted.)

In order to be able to support arbitrary argument lists, the __call__ method first constructs a new argument tuple. Conveniently, because of the notation *args in __call__'s own argument list, the arguments to __call__ (except for self) are placed in the tuple args. To construct the desired argument list, we concatenate a singleton tuple containing the instance with the args tuple: (self.instance,) + args. (Note the trailing comma used to construct the singleton tuple.) In our example, the resulting argument tuple is (aninstance, 10).

The intrinsic function apply() takes a function and an argument tuple and calls the function for it. In our example, we are calling apply(method1, (aninstance, 10)) which is equivalent to calling method(aninstance, 10).

From here on, things should come together quite easily. The output of the example code is something like this:

```
calling <function method1 at ae8d8> for <Instance instance at 95ab0> with (10,)
calling <function method2 at ae900> for <Instance instance at 95ab0> with ()
the answer is 10
```

That was about the shortest meaningful example that I could come up with. A real tracing metaclass (for example, Trace.py discussed below) needs to be more complicated in two dimensions.

First, it needs to support more advanced Python features such as class variables, inheritance, __init__ methods, and keyword arguments.

Second, it needs to provide a more flexible way to handle the actual tracing information; perhaps it

should be possible to write your own tracing function that gets called, perhaps it should be possible to enable and disable tracing on a per-class or per-instance basis, and perhaps a filter so that only interesting calls are traced; it should also be able to trace the return value of the call (or the exception it raised if an error occurs). Even the Trace.py example doesn't support all these features yet.

Real-life Examples

Have a look at some very preliminary examples that I coded up to teach myself how to write metaclasses:

Enum.py
: This (ab)uses the class syntax as an elegant way to define enumerated types. The resulting classes are never instantiated -- rather, their class attributes are the enumerated values. For example:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3
print Color.red
```

will print the string ``Color.red", while ``Color.red==1" is true, and ``Color.red + 1" raise a TypeError exception.

Trace.py
: The resulting classes work much like standard classes, but by setting a special class or instance attribute __trace_output__ to point to a file, all calls to the class's methods are traced. It was a bit of a struggle to get this right. This should probably redone using the generic metaclass below.

Meta.py
: A generic metaclass. This is an attempt at finding out how much standard class behavior can be mimicked by a metaclass. The preliminary answer appears to be that everything's fine as long as the class (or its clients) don't look at the instance's __class__ attribute, nor at the class's __dict__ attribute. The use of __getattr__ internally makes the classic implementation of __getattr__ hooks tough; we provide a similar hook _getattr_ instead. (__setattr__ and __delattr__ are not affected.) (XXX Hm. Could detect presence of __getattr__ and rename it.)

Eiffel.py
: Uses the above generic metaclass to implement Eiffel style pre-conditions and post-conditions.

Synch.py
: Uses the above generic metaclass to implement synchronized methods.

Simple.py
: The example module used above.

A pattern seems to be emerging: almost all these uses of metaclasses (except for Enum, which is probably more cute than useful) mostly work by placing wrappers around method calls. An obvious

problem with that is that it's not easy to combine the features of different metaclasses, while this would actually be quite useful: for example, I wouldn't mind getting a trace from the test run of the Synch module, and it would be interesting to add preconditions to it as well. This needs more research. Perhaps a metaclass could be provided that allows stackable wrappers...

Things You Could Do With Metaclasses

There are lots of things you could do with metaclasses. Most of these can also be done with creative use of __getattr__, but metaclasses make it easier to modify the attribute lookup behavior of classes. Here's a partial list.

- Enforce different inheritance semantics, e.g. automatically call base class methods when a derived class overrides
- Implement class methods (e.g. if the first argument is not named 'self')
- Implement that each instance is initialized with **copies** of all class variables
- Implement a different way to store instance variables (e.g. in a list kept outside the the instance but indexed by the instance's id())
- Automatically wrap or trap all or certain methods
    - for tracing
- for precondition and postcondition checking
- for synchronized methods
- for automatic value caching
- When an attribute is a parameterless function, call it on reference (to mimic it being an instance variable); same on assignment
- Instrumentation: see how many times various attributes are used
- Different semantics for __setattr__ and __getattr__ (e.g. disable them when they are being used recursively)
- Abuse class syntax for other things
- Experiment with automatic type checking
- Delegation (or acquisition)
- Dynamic inheritance patterns
- Automatic caching of methods

Credits

Many thanks to David Ascher and Donald Beaudry for their comments on earlier draft of this paper. Also thanks to Matt Conway and Tommy Burnette for putting a seed for the idea of metaclasses in my mind, nearly three years ago, even though at the time my response was ``you can do that with __getattr__ hooks..." :-)