A Study of Python's More Advanced Features Part III: Classes and Metaclasses

Posted on Aug 09, 2013 (2013-08-09T17:10:00-07:00) - Permanent link

Classes as Objects

Classes, like functions, are objects in Python. As soon as a class is defined, Python creates a class object and assigns it to a variable with the name of the class. Classes are objects of type `type` (though not always... more on this later.)

Class objects are callable (i.e. they implement a `__call__` method), and calling them creates objects of that class. You can treat classes like any other object. For example, you can assign attributes to them, you can assign them to variables, and you can use them where a callable is needed, for example in `map`. This is in fact exactly what happens when you do something like `map(str, [1,2,3])` to convert a list of integers to a list of strings, since `str` is a class.

See the code block below to get a sense of all this.

```
>>> class C(object):
...     def __init__(self, s):
...             print s
...
>>> myclass = C
>>> type(C)
<type 'type'>
>>> type(myclass)
<type 'type'>
>>> myclass(2)
2
<__main__.C object at 0x10e2bea50>
>>> map(myclass, [1,2,3])
1
2
3
[<__main__.C object at 0x10e2be9d0>, <__main__.C object at 0x10e2bead0>, <__main__.C object at 0
x10e2beb10>]
>>> map(C, [1,2,3])
1
2
3
[<__main__.C object at 0x10e2be950>, <__main__.C object at 0x10e2beb50>, <__main__.C object at 0
x10e2beb90>]
>>> C.test_attribute = True
>>> myclass.test_attribute
True
```

Because of this, the `class` keyword in Python does not have to appear on the main scope as it does in other programming languages like C++. In Python, it can be nested in a function, for example. We can use this to create dynamic classes on the fly in a function. Let's look at an example:

```
>>> def make_class(class_name):
...     class C(object):
...             def print_class_name(self):
...                     print class_name
...     C.__name__ = class_name
...     return C
...
>>> C1, C2 = map(make_class, ["C1", "C2"])
>>> c1, c2 = C1(), C2()
>>> c1.print_class_name()
C1
>>> c2.print_class_name()
C2
>>> type(c1)
<class '__main__.C1'>
>>> type(c2)
<class '__main__.C2'>
>>> c1.print_class_name.__closure__
(<cell at 0x10ab6dbe8: str object at 0x10ab71530>,)
```

Notice that the two classes created using `make_class` here are different objects, and therefore objects created by them are not of the same type. We set the class name manually after the class is created, similar to how we do in decorators. Also note that the `print_class_name` method of the created class captures its closure and has `class_name` in a closure cell. If you do not feel comfortable with closures, now might be a good time to look at part two of the series (http://sahandsaba.com/python-decorators.html) for an exploration of closures and decorators.

Metaclasses

If classes are objects that make objects, then what are objects that make classes called? (And trust me, this is not a "chicken or egg" type of riddle!) The answer is metaclasses. The most basic metaclass is in fact `type`. When passed one parameter, `type` simply returns the type of the object passed to it. It does not act as a metaclass in this case. However, when passed three parameters, `type` acts as a metaclass and creates a class based on the parameters passed. The parameters are simple: the class's name, its parents (classes to inherit from), and a dictionary of its attributes. The latter two can be empty. Let's try this:

```
>>> MyClass = type("MyClass", (object,), {"my_attribute": 0})
>>> type(MyClass)
<type 'type'>
>>> o = MyClass()
>>> o.my_attribute
0
```

Note that the second parameter must be a tuple, hence the odd syntax. If you want methods, then create the functions and pass them as attributes. Like this:

```
>>> def myclass_init(self, my_attr):
...     self.my_attribute = my_attr
...
>>> MyClass = type("MyClass", (object,), {"my_attribute": 0, "__init__": myclass_init})
>>> o = MyClass("Test")
>>> o.my_attribute
'Test'
>>> o.__init__
<bound method MyClass.myclass_init of <__main__.MyClass object at 0x10ab72150>>
```

We can make our own metaclasses by writing callable objects (functions or classes) that take three parameters and return an object. Such a metaclass can be applied to a class by setting the special `__metaclass__` attribute of the class. For our first example, let's do something very odd, just to see how far we can go with metaclasses:

```
>>> def mymetaclass(name, parents, attributes):
...     return "Hello"
...
>>> class C(object):
...     __metaclass__ = mymetaclass
...
>>> print C
Hello
>>> type(C)
<type 'str'>
```

Notice how in the above code, `C` ends up simply being a variable referencing the string `"Hello"`. Of course, hopefully no one in their right mind would do something like that. The point here was to see how metaclasses work. Next, let's do something possibly more useful. In part two of the series (http://sahandsaba.com/python-decorators.html) we saw how a class decorator could be used to make every method in a class logged. Let's do the same thing here, but this time using a metaclass instead. We borrow the `logged` decorator from that post.

```
def log_everything_metaclass(class_name, parents, attributes):
    print "Creating class", class_name
    myattributes = {}
    for name, attr in attributes.items():
        myattributes[name] = attr
        if hasattr(attr, '__call__'):
            myattributes[name] = logged("%b %d %Y - %H:%M:%S",
                                        class_name + ".")(attr)
    return type(class_name, parents, myattributes)



class C(object):
    __metaclass__ = log_everything_metaclass

    def __init__(self, x):
        self.x = x

    def print_x(self):
        print self.x

# Usage:
print "Starting object creation"
c = C("Test")
c.print_x()
```

```
# Output:
Creating class C
Starting object creation
- Running 'C.__init__' on Aug 05 2013 - 13:50:58
- Finished 'C.__init__', execution time = 0.000s
- Running 'C.print_x' on Aug 05 2013 - 13:50:58
Test
- Finished 'C.print_x', execution time = 0.000s
```

As you can see, class decorators and metaclasses have quite a bit in common. In fact, anything that can be done with a class decorator can done using a metaclass. Class decorators have a simpler syntax and are easier to read, and should therefore be used if possible. Metaclasses are capable of more since they are run before the class is created, rather than after, which is the case with decorators. Just to be sure about this, let’s do both and see the order of execution for ourselves:

```
def my_metaclass(class_name, parents, attributes):
    print "In metaclass, creating the class."
    return type(class_name, parents, attributes)



def my_class_decorator(class_):
    print "In decorator, chance to modify the class."
    return class_



@my_class_decorator
class C(object):
    __metaclass__ = my_metaclass

    def __init__(self):
        print "Creating object."

c = C()
```

```
# Output:
In metaclass, creating the class.
In decorator, chance to modify the class.
Creating object.
```

A Sample Use-Case For Metaclasses

Let's consider a more useful application here. Suppose that we are writing a set of classes for processing ID3v2 tags, such as ones used on MP3 music files. You can read up on them on Wikipedia (http://en.wikipedia.org/wiki/ID3). The basic summary that we need for this example is that tags consist of frames, and each frame is identified by a four letter identifier. For example, `TOPE` is the original artist frame, `TOAL` is the original album name, etc. Suppose here that we want to write a separate class for each type of frame, and allow the user of our ID3v2 tag library to add their own frame classes to support new or custom frames with ease. We can use metaclasses to implement a class factory pattern here. It would look something like this:

```python
frametype_class_dict = {}


class ID3v2FrameClassFactory(object):
    def __new__(cls, class_name, parents, attributes):
        print "Creating class", class_name
        # Here we could add some helper methods or attributes to c
        c = type(class_name, parents, attributes)
        if attributes['frame_identifier']:
            frametype_class_dict[attributes['frame_identifier']] = c
        return c

    @staticmethod
    def get_class_from_frame_identifier(frame_identifier):
        return frametype_class_dict.get(frame_identifier)


class ID3v2Frame(object):
    frame_identifier = None
    __metaclass__ = ID3v2FrameClassFactory
    pass


class ID3v2TitleFrame(ID3v2Frame):
    __metaclass__ = ID3v2FrameClassFactory
    frame_identifier = "TIT2"


class ID3v2CommentFrame(ID3v2Frame):
    __metaclass__ = ID3v2FrameClassFactory
    frame_identifier = "COMM"


title_class = ID3v2FrameClassFactory.get_class_from_frame_identifier('TIT2')
comment_class = ID3v2FrameClassFactory.get_class_from_frame_identifier('COMM')
print title_class
print comment_class
```

```
# Output:
Creating class ID3v2Frame
Creating class ID3v2TitleFrame
Creating class ID3v2CommentFrame
<class '__main__.ID3v2TitleFrame'>
<class '__main__.ID3v2CommentFrame'>
```

Of course, the above code could be accomplished using class decorators as well. For comparison purposes, let’s see this too:

```
frametype_class_dict = {}


class ID3v2FrameClass(object):
    def __init__(self, frame_id):
        self.frame_id = frame_id

    def __call__(self, cls):
        print "Decorating class", cls.__name__
        # Here we could add some helper methods or attributes to c
        if self.frame_id:
            frametype_class_dict[self.frame_id] = cls
        return cls

    @staticmethod
    def get_class_from_frame_identifier(frame_identifier):
        return frametype_class_dict.get(frame_identifier)


@ID3v2FrameClass(None)
class ID3v2Frame(object):
    pass


@ID3v2FrameClass("TIT2")
class ID3v2TitleFrame(ID3v2Frame):
    pass


@ID3v2FrameClass("COMM")
class ID3v2CommentFrame(ID3v2Frame):
    pass


title_class = ID3v2FrameClass.get_class_from_frame_identifier('TIT2')
comment_class = ID3v2FrameClass.get_class_from_frame_identifier('COMM')
print title_class
print comment_class
```

```
Decorating class ID3v2Frame
Decorating class ID3v2TitleFrame
Decorating class ID3v2CommentFrame
<class '__main__.ID3v2TitleFrame'>
<class '__main__.ID3v2CommentFrame'>
```

As you can see, we can pass parameters directly to decorators but not metaclasses. Parameters to metaclasses would have to be passed through the attributes. Because of this, the decorator solution is cleaner and easier to maintain. However, also note that by the time the decorator is called, the class is already created, which means that it is too late to change any of its more readonly properties. For example, you can not change `__doc__` once the class is created. Let's see this in practice:

```
>>> def mydecorator(cls):
...     cls.__doc__ = "Test!"
...     return cls
...
>>> @mydecorator
... class C(object):
...     """Docstring to be replaced with Test!"""
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in mydecorator
AttributeError: attribute '__doc__' of 'type' objects is not writable
>>> def mymetaclass(cls, parents, attrs):
...     attrs['__doc__'] = 'Test!'
...     return type(cls, parents, attrs)
...
>>> class D(object):
...     """Docstring to be replaced with Test!"""
...     __metaclass__ = mymetaclass
...
>>> D.__doc__
'Test!'
```

Metaclasses Derived From `type`

As we discussed, the most basic metaclass is `type` and as such classes are usually of type `type`. A natural question to ask at this point is what type does `type` itself have? The answer is `type`. Meaning `type` is itself a class, and it has itself as its metaclass. This is of course extraordinary, made possible at the interpreter level of Python. You can not write a class that has itself as its metaclass.

Knowing that `type` itself is a class, we can write classes that inherit from it. Such classes can be used as metaclasses, and classes that use them will have their type equal to the metaclass deriving from `type`. Let's look at an example:

```
>>> class meta(type):
...     def __new__(cls, class_name, parents, attributes):
...             print "meta.__new__"
...             return super(meta, cls).__new__(cls, class_name, parents, attributes)
...     def __call__(self, *args, **kwargs):
...             print "meta.__call__"
...             return super(meta, self).__call__(*args, **kwargs)
...
>>> class C(object):
...     __metaclass__ = meta
...
meta.__new__
>>> c = C()
meta.__call__
>>> type(C)
<class '__main__.meta'>
```

Notice that when the class is called to create a new object, the metaclass's `__call__` function is called, which then calls `type.__call__` to create the object. In next section, we put everything we have learned so far together.

Putting It All Together

Suppose a class `C` has its metaclass set to `my_metaclass` and is decorated with `my_class_decorator`. Further, assume that `my_metaclass` is itself a class, deriving from `type`. Let's put everything together to form a summary of how `C` is created and how objects of its type are created. First, this is how the code looks like:

```python
class my_metaclass(type):
    def __new__(cls, class_name, parents, attributes):
        print "- my_metaclass.__new__ - Creating class instance of type", cls
        return super(my_metaclass, cls).__new__(cls,
                                                class_name,
                                                parents,
                                                attributes)

    def __init__(self, class_name, parents, attributes):
        print "- my_metaclass.__init__ - Initializing the class instance", self
        super(my_metaclass, self).__init__(self)

    def __call__(self, *args, **kwargs):
        print "- my_metaclass.__call__ - Creating object of type ", self
        return super(my_metaclass, self).__call__(*args, **kwargs)


def my_class_decorator(cls):
    print "- my_class_decorator - Chance to modify the class", cls
    return cls


@my_class_decorator
class C(object):
    __metaclass__ = my_metaclass

    def __new__(cls):
        print "- C.__new__ - Creating object."
        return super(C, cls).__new__(cls)

    def __init__(self):
        print "- C.__init__ - Initializing object."

c = C()
print "Object c =", c
```

At this point, you can take a minute to test your understanding and try to guess the order in which the print statements are executed!

First let's go through the flow of how Python interprets this code. Then we will look at the output to confirm our understanding.

1. Python reads the class declaration and prepares the three parameters to pass to the metaclass. The three parameters are `class_name`, `parents`, and `attributes`.
2. Python looks for a `__metaclass__` attribute. If it is set, which it is in this case, it calls the metaclass, passing the three parameters to it. The result returned is the class.

3. In our case, the metaclass is itself a class, and hence calling it is akin to creating a new class. This means that first `my_metaclass.__new__` will be called with 4 parameters. This creates a new instance of the metaclass class. Then `my_metaclass.__init__` will be called on this instance. The result is then returned as the new class. So `C` at this point will be set to this object.
4. Python then looks for all the decorators to apply to the class. In this case, there is only one. Python calls the decorator, passing the class returned from the metaclass to it as the parameter. The class is replaced by the object returned from the decorator.
5. The class's type will be the same as the metaclass set.
6. When the class is called to create a new object, since the class's type is the metaclass, Python looks for `__call__` in the metaclass. In our case `my_metaclass.__call__` simply calls `type.__call__`, which creates an object based on the class it is passed.
7. Next `type.__call__` creates an object by looking for `C.__new__` first and running it.
8. Finally `type.__call__` runs `C.__init__` on the result returned from `C.__new__`.
9. The returned object is then ready to be used.

So based on this logic, we would expect to see `my_metaclass.__new__` called first, then `my_metaclass.__init__`, then `my_class_decorator`. At this point the class `C` is fully prepared. When we call `C` to create an object, which calls `my_metaclass.__call__` first (anytime an object is called Python looks for `__call__` in their class to invoke), then `C.__new__` is invoked by `type.__call__`, and finally `C.__init__` is called. Let's look at the output:

```
- my_metaclass.__new__ - Creating class instance of type <class '__main__.my_metaclass'>
- my_metaclass.__init__ - Initializing the class instance <class '__main__.C'>
- my_class_decorator - Chance to modify the class <class '__main__.C'>
- my_metaclass.__call__ - Creating object of type  <class '__main__.C'>
- C.__new__ - Creating object.
- C.__init__ - Initializing object.
Object c = <__main__.C object at 0x1043feb90> <class '__main__.C'>
```

Metaclasses in The Wild

Metaclasses, though powerful, are rather esoteric. A search of `__metaclass__` on GitHub narrowed down to Python only results (https://github.com/search?l=Python&q=__metaclass__&type=Code) yields lots of links to "cookbooks" and other educational material, a few test cases (such as a test case for Jython), and many places where `__metaclass__ = type` is used to ensures new-style Python classes are used. To be fair, none of these count as real uses of metaclasses. Sifting through the results, I could only find two places where metaclasses are really used: `ABCMeta` and `djangoplugins`.

`ABCMeta` is a metaclass that allows registration of abstract base classes. See the official documentation on ABCMeta (http://docs.python.org/2/library/abc.html) for more details, as an explanation of ABC's is outside the scope of this article.

For `djangoplugins`, the idea is based on the following article on a simple plugin framework for Python (http://martyalchin.com/2008/jan/10/simple-plugin-framework/), where a metaclass is used to create a plugin mount system. Though I haven't yet managed to work on it in detail, I have a feeling the same type of framework can be achieved using decorators instead of metaclasses. If you have any opinions on this feel free to leave a comment.

Final Note on Metaclasses

While understanding metaclasses helps in gaining a deeper understanding of how classes and objects behave in Python, real world use-cases for them can be hard to come by as shown in the previous section. Most of what can be accomplished using them can also be done with decorators. As such, if your first intuition for a solution is to use metaclasses, take a moment to consider if it is entirely necessary to use them or not. If it is possible to do without them, it is most likely a better idea to do without them. The result will be easier to read, debug, and maintain.

Comments

Recommended Articles

A Study of Python's More Advanced Features Part II: Closures, Decorators and functools

A Study of Python's More Advanced Features Part I: Iterators, Generators, itertools

Combinatorial Generation Using Coroutines With Examples in Python

Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js

30 Python Language Features and Tricks You May Not Know About