Python metaclasses by example (http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example)

August 14, 2011 at 06:05 Tags Articles (http://eli.thegreenplace.net/tag/articles) , Python (http://eli.thegreenplace.net/tag/python)

Python rightfully prides itself as a relatively straightforward language without a lot of "magic" hiding in its workings and features. Sometimes, however, to make interesting abstractions possible, one can dig deep in Python's more dusty and obscure corners to find language constructs that are a bit more magical than usual. Metaclasses are one such feature.

Unfortunately, metaclasses have a reputation for "a solution seeking a problem". The aim of this article is to demonstrate a few actual uses of metaclasses in widely used Python code.

There is a lot of material on Python metaclasses online, so this isn't just another tutorial (look in the References section below for some links I found useful). I will spend some time explaining what metaclasses are, but my main aim is the examples. That said, this article still aspires to be self-contained - you can start reading it even if you don't know what metaclasses are.

Another quick note before we begin - this article focuses on Python 2.6 & 2.7, because most of the code you find online is still for these versions [1]. In Python 3.x metaclasses work similarly, although the syntax of specifying them is a bit different. So the vast majority of this article applies to 3.x as well.

Classes are objects too

To understand metaclasses, first we should make some things clear about classes. In Python, everything is an object. And that includes classes. In fact, classes in Python are first-class objects (http://en.wikipedia.org/wiki/First-class_object) [2] - they can be created at runtime, passed as parameters and returned from functions, and assigned to variables. Here's a short interactive session that demonstrates these qualities of classes:

```
>>> def make_myklass(**kwattrs):
...   return type('MyKlass', (object,), dict(**kwattrs))
...
>>> myklass_foo_bar = make_myklass(foo=2, bar=4)
>>> myklass_foo_bar
<class __main__.MyKlass>
>>> x = myklass_foo_bar()
>>> x
<__main__.MyKlass object at 0x01F6B050>
>>> x.foo, x.bar
(2, 4)
```

Here we use the 3-argument form of the `type` built-in function to dynamically create a class named `MyKlass`, inheriting from `object` with some attributes provided as arguments. Then we create one such class. As you can see, `myklass_foo_bar` is equivalent to:

```
class MyKlass(object):
  foo = 2
  bar = 4
```

But it was created at runtime, returned from a function and assigned to a variable.

The class of a class

Every object (including built-ins) in Python has a class. We've just seen that classes are objects too, so classes must also have a class, right? Exactly. Python lets us examine the class of an object with the `__class__` attribute. Let's see this in action:

```
>>> class SomeKlass(object): pass
...
>>> someobject = SomeKlass()
>>> someobject.__class__
<class __main__.SomeKlass>
>>> SomeKlass.__class__
<type 'type'>
```

We've created a class and an object of that class. Examining the `__class__` of `someobject` we saw that it's `SomeKlass`. Next comes the interesting part. What is the class of `SomeKlass`? We can again examine it with `__class__` and we see it's `type`.

So `type` is the class of Python classes [3]. In other words, while in the example above `someobject` is a SomeKlass object, `SomeKlass` itself is a type object.

I don't know about you, but I find this reassuring. Since we learned that classes are objects in Python, it makes sense that they also have a class, and it's nice to know there's a built-in class (`type`) serving the role of being the class of classes.

Metaclass

A metaclass is defined as "the class of a class". Any class whose instances are themselves classes, is a metaclass. So, according to what we've seen above, this makes `type` a metaclass - in fact, the most commonly used metaclass in Python, since it's the default metaclass of all classes.

Since a metaclass is the class of a class, it is used to construct classes (just as a class is used to construct objects). But wait a second, don't we create classes with a standard `class` definition? Definitely, but what Python does under the hood is the following:

- When it sees a `class` definition, Python executes it to collect the attributes (including methods) into a dictionary.
- When the `class` definition is over, Python determines the metaclass of the class. Let's call it `Meta`
- Eventually, Python executes `Meta(name, bases, dct)`, where:
  - `Meta` is the metaclass, so this invocation is instantiating it.
  - `name` is the name of the newly created class
  - `bases` is a tuple of the class's base classes
  - `dct` maps attribute names to objects, listing all of the class's attributes

How do we determine the metaclass of a class? Simply stated [4], if either a class or one of its bases has a `__metaclass__` attribute [5], it's taken as the metaclass. Otherwise, `type` is the metaclass.

So what happens when we define:

```
class MyKlass(object):
  foo = 2
```

Is this: `MyKlass` has no `__metaclass__` attribute, so `type` is used instead, and the class creation is done as:

```
MyKlass = type(name, bases, dct)
```

Which is consistent to what we've seen in the beginning of the article. If, on the other hand, `MyKlass` does have a metaclass defined:

```
class MyKlass(object):
  __metaclass__ = MyMeta
  foo = 2
```

Then the class creation is done as:

```
MyKlass = MyMeta(name, bases, dct)
```

So `MyMeta` should be implemented appropriately to support such calling form and return the new class. It's actually similar to writing a normal class with a pre-defined constructor signature.

Metaclass's __new__ and __init__

To control the creation and initialization of the class in the metaclass, you can implement the metaclass's **__new__** method and/or **__init__** constructor [6]. Most real-life metaclasses will probably override just one of them. **__new__** should be implemented when you want to control the creation of a new object (class in our case), and **__init__** should be implemented when you want to control the initialization of the new object after it has been created.

So when the call to **MyMeta** is done above, what happens under the hood is this:

```
MyKlass = MyMeta.__new__(MyMeta, name, bases, dct)
MyMeta.__init__(MyKlass, name, bases, dct)
```

Here's a more concrete example that should demonstrate what's going on. Let's write down this definition for a metaclass:

```
class MyMeta(type):
    def __new__(meta, name, bases, dct):
        print '-----------------------------------'
        print "Allocating memory for class", name
        print meta
        print bases
        print dct
        return super(MyMeta, meta).__new__(meta, name, bases, dct)
    def __init__(cls, name, bases, dct):
        print '-----------------------------------'
        print "Initializing class", name
        print cls
        print bases
        print dct
        super(MyMeta, cls).__init__(name, bases, dct)
```

When Python executes the following class definition:

```
class MyKlass(object):
    __metaclass__ = MyMeta

    def foo(self, param):
        pass

    barattr = 2
```

What gets printed is this (reformatted for clarity):

```
----------------------------------
Allocating memory for class MyKlass
<class '__main__.MyMeta'>
(<type 'object'>,)
{'barattr': 2, '__module__': '__main__',
 'foo': <function foo at 0x00B502F0>,
 '__metaclass__': <class '__main__.MyMeta'>}
----------------------------------
Initializing class MyKlass
<class '__main__.MyKlass'>
(<type 'object'>,)
{'barattr': 2, '__module__': '__main__',
 'foo': <function foo at 0x00B502F0>,
 '__metaclass__': <class '__main__.MyMeta'>}
```

Study and understand this example and you'll grasp most of what one needs to know about writing metaclasses.

It's important to note here that these print-outs are actually done at class creation time, i.e. when the module containing the class is being imported for the first time. Keep this detail in mind for later.

Metaclass's `__call__`

Another metaclass method that's occasionally useful to override is **`__call__`**. The reason I'm discussing it separately from **`__new__`** and **`__init__`** is that unlike those two that get called at class creation time, **`__call__`** is called when the already-created class is "called" to instantiate a new object. Here's some code to clarify this:

```
class MyMeta(type):
    def __call__(cls, *args, **kwds):
        print '__call__ of ', str(cls)
        print '__call__ *args=', str(args)
        return type.__call__(cls, *args, **kwds)

class MyKlass(object):
    __metaclass__ = MyMeta

    def __init__(self, a, b):
        print 'MyKlass object with a=%s, b=%s' % (a, b)

print 'gonna create foo now...'
foo = MyKlass(1, 2)
```

This prints:

```
gonna create foo now...
__call__ of  <class '__main__.MyKlass'>
__call__ *args= (1, 2)
MyKlass object with a=1, b=2
```

Here **`MyMeta.__call__`** just notifies us of the arguments and delegates to **`type.__call__`**. But it

can also interfere in the process, affecting the way objects of the class are created. In a way, this is not unlike overriding the `__new__` method of the class itself, although there are some differences [7].

Examples

We've now covered enough theory to understand what metaclasses are and how to write them. At this point, it's time for the examples that should make things clearer. As I mentioned above, instead of writing synthetic examples I prefer to examine the usage of metaclasses in real Python code.

string.Template

The first example of a metaclass is taken from the Python standard library. It is one of the very few examples of metaclasses that ships with Python itself.

`string.Template` provides convenient, named string substitutions, and can serve as a very simple templating system. If you're not familiar with this class, this would be a good time to read the docs. I will just explain how it uses metaclasses.

Here are the first few lines from `class Template`:

```
class Template:
    """A string class for supporting $-substitutions."""
    __metaclass__ = _TemplateMetaclass

    delimiter = '$'
    idpattern = r'[_a-z][_a-z0-9]*'

    def __init__(self, template):
        self.template = template
```

And this is `_TemplateMetaclass`:

```
class _TemplateMetaclass(type):
    pattern = r"""
    %(delim)s(?:
      (?P<escaped>%(delim)s) |   # Escape sequence of two delimiters
      (?P<named>%(id)s)      |   # delimiter and a Python identifier
      {(?P<braced>%(id)s)}   |   # delimiter and a braced identifier
      (?P<invalid>)              # Other ill-formed delimiter exprs
    )
    """

    def __init__(cls, name, bases, dct):
        super(_TemplateMetaclass, cls).__init__(name, bases, dct)
        if 'pattern' in dct:
            pattern = cls.pattern
        else:
            pattern = _TemplateMetaclass.pattern % {
                'delim' : _re.escape(cls.delimiter),
                'id'    : cls.idpattern,
                }
        cls.pattern = _re.compile(pattern, _re.IGNORECASE | _re.VERBOSE)
```

The explanation provided in the first part of this article should be sufficient for understanding how `_TemplateMetaclass` works. Its `__init__` method looks at some class attributes (specifically, `pattern`, `delimiter` and `idpattern`) and uses them (or its own-supplied defaults) to build a compiled regex, which is then stored back into the class's `pattern` attribute.

According to its documentation, `Template` can be inherited to provide a custom delimiter and ID pattern, or the whole regex. The metaclass makes sure that these get converted into a compiled regex pattern at class creation time, so this is an optimization of a sort.

What I mean is that the same customization could be achieved without using a metaclass, by simply building the compiled regex in the constructor. However, this means that the compilation step is done each time a `Template` object is instantiated.

Consider the following usage, which IMHO is common with `string.Template`:

```
>>> from string import Template
>>> Template("$name is $value").substitute(name='me', value='2')
'me is 2'
```

Leaving regex compilation to `Template` instantiation time means it is being created and compiled each time such piece of code runs. And this is a shame - because the regex really isn't dependent on the template string, but only on the properties of the class.

With a metaclass, the `pattern` class attribute is getting created just once when the module is being loaded and the `class Template` (or its subclass) definition is being executed. This saves time when `Template` objects are created, and makes sense because at class creation time we have all the information we need to compile the regex - so why delay this operation?

One may claim that this is a premature optimization, and this could be true. I don't plan to defend this (or any) usage of a metaclass. My intention here is simply to demonstrate how metaclasses are being used in real code for various tasks. So, for this educational purpose it's a good example, since it shows an interesting use case. Whether premature optimization or not, the metaclass does make code more efficient by moving a computation one step earlier in the process of code execution.

twisted.python.reflect.AccessorType

The following example is a frequently-used demonstration of metaclasses. An excerpt from its documentation:

> Metaclass that generates properties automatically. Using this metaclass for your class will give you explicit accessor methods; a method called set_foo, will automatically create a property 'foo' that uses set_foo as a setter method. Same for get_foo and del_foo.

Here's the metaclass, shortened a bit to emphasize the important parts:

```
class AccessorType(type):
    def __init__(self, name, bases, d):
        type.__init__(self, name, bases, d)
        accessors = {}
        prefixs = ["get_", "set_", "del_"]
        for k in d.keys():
            v = getattr(self, k)
            for i in range(3):
                if k.startswith(prefixs[i]):
                    accessors.setdefault(k[4:], [None, None, None])[i] = v
        for name, (getter, setter, deler) in accessors.items():
            # create default behaviours for the property - if we leave
            # the getter as None we won't be able to getattr, etc..

            # [...] some code that implements the above comment

            setattr(self, name, property(getter, setter, deler, ""))
```

What this does is straightforward:

1. Find all attributes of the class that start with `get_`, `set_` or `del_`
2. Organize them by the property they aim to control (the part of their name that comes after the underscore)
3. For each getter, setter, deleter triple thus found:

    1. Make sure all three exist, or create suitable defaults
    2. Set them as a `property` on the class

How useful is such a metaclass? It's hard to say, really. Twisted itself doesn't use it, but does provide it as a public API. If you have several classes to write with a lot of properties, this metaclass may save quite a bit of coding.

pygments Lexer and RegexLexer

The pygments (http://pygments.org/) library presents an interesting idiom of metaclass usage. A base class is created with a custom metaclass. User classes can then inherit from this base class, and get the metaclass as a bonus [8]. First, let's look at the `LexerMeta` metaclass, which is used as the metaclass for `Lexer` - the base class of lexers in pygments:

```
class LexerMeta(type):
    """
    This metaclass automagically converts ``analyse_text`` methods into
    static methods which always return float values.
    """

    def __new__(cls, name, bases, d):
        if 'analyse_text' in d:
            d['analyse_text'] = make_analysator(d['analyse_text'])
        return type.__new__(cls, name, bases, d)
```

This metaclass overrides the `__new__` method to intercept the definition of the `analyse_text` message and turn it into a static method that always returns a floating point value (this is what the `make_analysator` function does).

Note the usage of `__new__` instead of `__init__` here. Why isn't `__init__` used? In my opinion, this is simply a matter of preference - the same effect could also be achieved with overriding `__init__`.

The second example from pygments is more complicated, but worth the effort to explain since it contains a couple of features we haven't seen in previous examples. The code for `RegexLexerMeta` is quite long, so I will snip it to leave the relevant part:

```
class RegexLexerMeta(LexerMeta):
    """
    Metaclass for RegexLexer, creates the self._tokens attribute from
    self.tokens on the first instantiation.
    """

    # [...] snip

    def __call__(cls, *args, **kwds):
        """Instantiate cls after preprocessing its token definitions."""
        if not hasattr(cls, '_tokens'):
            cls._all_tokens = {}
            cls._tmpname = 0
            if hasattr(cls, 'token_variants') and cls.token_variants:
                # don't process yet
                pass
            else:
                cls._tokens = cls.process_tokendef('', cls.tokens)

        return type.__call__(cls, *args, **kwds)
```

Generally, the code is quite clear - the metaclass examines the `tokens` class attribute, and creates `_tokens` from it. This is only done on the first instantiation of the class. There are two things of special interest here:

1. `RegexLexerMeta` inherits from `LexerMeta`, so its users also get the service `LexerMeta` provides. Inheritance of metaclasses is one of the reasons they are one of the most powerful language constructs in Python. Contrast this to class decorators, for example. For some simple tasks, class decorators could replace metaclasses, but the ability of metaclasses to form inheritance relationships is something that decorators can't do.
2. The `process_tokendef` computation is performed in `__call__` - and a special check makes sure it actually runs only in the first instantiation of the class (although `__call__` itself is called for all instantiations). Why do it like this, instead of at class creation time (say in the metaclass's `__init__`)? It appears to me this could be an optimization of a sort. pygments comes with many lexers [9], but you may want to use just one or two in any given code. Why spend the loading time on lexers you don't need, as opposed to just the lexers you use? Whether this is the real reason or not, I think it's still an interesting aspect of metaclasses to ponder - the great flexibility they provide you to choose where and how to perform their meta-work.

Conclusion

My intention in this article was to explain how metaclasses in Python work and provide a few concrete examples of metaclass usage in real Python code. I know that metaclasses have some notoriety, as many people consider them more magic than usually necessary. My opinion on the subject is that, like other language constructs, metaclasses are a tool, and the programmer is eventually responsible for using it correctly. Always write the simplest code that does the job, but if you feel that a metaclass is what you need, you're free to use a metaclass.

I hope this article demonstrated the great flexibility metaclasses provide for customizing the way classes are created and used. The examples presented several aspects of metaclass implementation and usage - overriding `__init__`, `__new__` and `__call__` methods, using metaclass inheritance, adding properties to classes, converting object methods to static methods and performing optimizations at either class definition or instantiation time.

The most notable example of metaclasses in Python is probably their usage in ORM (Object Relational Mapping) frameworks, such as Django's models. Indeed, these are forceful demonstrations of what metaclasses are capable of, but I decided not to present them here because their code is complex and the many domain-specific details would obscure the main goal of presenting metaclasses. Having read this article, however, you have everything necessary to understand the more complex examples.

P.S. If you find other interesting examples of metaclasses, please let me know. I'm very interested in seeing additional real-life uses.

References

- Data model page (http://docs.python.org/reference/datamodel.html) in the official docs.
- Metaclass programming in Python (http://www.ibm.com/developerworks/linux/library/l-pymeta/index.html) - a series of articles explaining metaclasses.
- What is a metaclass in Python? (http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python) - a great StackOverflow discussion.
- Unifying types and classes in Python 2.2 (http://www.python.org/download/releases/2.2.3/descrintro) - a very informative article by Guido von Rossum that touches on metaclasses as well.
- A StackOverflow discussion (http://stackoverflow.com/questions/4859129/python-and-python-c-api-new-versus-init) on the differences between `__new__` and `__init__`.
- Another StackOverflow discussion about the concrete uses of metaclasses (http://stackoverflow.com/questions/392160/what-are-your-concrete-use-cases-for-metaclasses-in-python).

[1] Moreover, I'm only touching upon new-style classes (i.e. those deriving from `object`). I hope that all new code you write and the vast majority of code you see these days uses new-style classes. With old-style classes, some of the specifics of metaclasses presented here are a bit different, though in general the same principles apply.

[2] The word-play here is unintended. In "first-class objects" the word class has the same meaning as "rank" or "caliber", to imply that these objects aren't worse than other objects in the language.

[3] You may notice a duplicity here - `type` is both a class and a built-in used for creating new classes and checking the types of objects. This isn't different from other built-ins in Python. For example, `dict` is both the class of dictionaries (`{}.__class__` is `<type 'dict'>`) and a constructor of dictionaries. In fact, this is true for user-defined classes as well, since the name of the class also serves as a constructor for new objects.

So you can think of `type` as just another built-in class in Python.

[4] You can look up the exact rules in the data model reference (http://docs.python.org/reference/datamodel.html).

[5] Again, a reminder that I describe Python 2 in this article. In Python 3 the syntax of defining metaclasses has been changed, and made a bit more intuitive.

[6] Explaining the exact differences between `__new__` and `__init__` will take an article of its own and I won't cover it beyond the simple examples here. You may want to google this topic for better understanding.

[7] For a good discussion of this, see this StackOverflow question (http://stackoverflow.com/questions/6966772/using-the-call-method-of-a-metaclass-instead-of-new/6966942#6966942).

[8] Since Python's metaclass lookup rules dictate that if a class itself has no metaclass, its base classes are looked at to find metaclasses.

[9] And, more importantly, aggregates several lexers in each module.

Comments

↑ Back to top