A Primer on Python Metaclasses

Sat 01 December 2012

Python, Classes, and Objects

Most readers are aware that Python is an object-oriented language. By object-oriented, we mean that Python can define *classes*, which bundle **data** and **functionality** into one entity. For example, we may create a class `IntContainer` which stores an integer and allows certain operations to be performed:

In [1]:
```
class IntContainer(object):
    def __init__(self, i):
        self.i = int(i)

    def add_one(self):
        self.i += 1
```

In [2]:
```
ic = IntContainer(2)
ic.add_one()
print(ic.i)
```

```
3
```

This is a bit of a silly example, but shows the fundamental nature of classes: their ability to bundle data and operations into a single *object*, which leads to cleaner, more manageable, and more adaptable code. Additionally, classes can inherit properties from parents and add or specialize attributes and methods. This *object-oriented* approach to programming can be very intuitive and powerful.

What many do not realize, though, is that quite literally *everything* (http://www.diveintopython.net/getting_to_know_python/everything_is_an_object.html) in the Python language is an object.

For example, integers are simply instances of the built-in `int` type:

In [3]:
```
print type(1)
```

```
<type 'int'>
```

To emphasize that the `int` type really is an object, let's derive from it and specialize the `__add__` method (which is the machinery underneath the `+` operator):

(Note: We'll used the `super` *syntax to call methods from the parent class: if you're unfamiliar with this, take a look at this StackOverflow question (http://stackoverflow.com/questions/576169/understanding-python-super-and-init-methods).*

In [4]:
```
class MyInt(int):
    def __add__(self, other):
        print "specializing addition"
        return super(MyInt, self).__add__(other)

i = MyInt(2)
print(i + 2)
```

```
specializing addition
4
```

Using the `+` operator on our derived type goes through our `__add__` method, as expected. We see that `int` really is an object that can be subclassed and extended just like user-defined classes. The same is true of `float` s, `list` s, `tuple` s, and everything else in the Python language. They're all objects.

Down the Rabbit Hole: Classes as Objects

We said above that *everything* in python is an object: it turns out that this is true of *classes themselves*. Let's look at an example.

We'll start by defining a class that does nothing

In [5]:
```
class DoNothing(object):
    pass
```

If we instantiate this, we can use the `type` operator to see the type of object that it is:

In [6]:
```
d = DoNothing()
type(d)
```

Out[6]: `__main__.DoNothing`

We see that our variable `d` is an instance of the class `__main__.DoNothing` .

We can do this similarly for built-in types:

In [7]:
```
L = [1, 2, 3]
type(L)
```

Out[7]: `list`

A list is, as you may expect, an object of type `list`.

But let's take this a step further: what is the type of `DoNothing` itself?

```
In [8]: type(DoNothing)
```

```
Out[8]: type
```

The type of `DoNothing` is `type`. This tells us that the *class* `DoNothing` is itself an object, and that object is of type `type`.

It turns out that this is the same for built-in datatypes:

```
In [9]: type(tuple), type(list), type(int), type(float)
```

```
Out[9]: (type, type, type, type)
```

What this shows is that in Python, *classes are objects*, and they are objects of type `type`. `type` is a *metaclass*: a class which instantiates classes. All new-style classes (http://www.python.org/doc/newstyle/) in Python are instances of the `type` metaclass, including `type` itself:

```
In [10]: type(type)
```

```
Out[10]: type
```

Yes, you read that correctly: the type of `type` is `type`. In other words, `type` is *an instance of itself*. This sort of circularity cannot (to my knowledge) be duplicated in pure Python, and the behavior is created through a bit of a hack at the implementation level of Python.

Metaprogramming: Creating Classes on the Fly

Now that we've stepped back and considered the fact that classes in Python are simply objects like everything else, we can think about what is known as *metaprogramming*. You're probably used to creating functions which return objects. We can think of these functions as an object factory: they take some arguments, create an object, and return it. Here is a simple example of a function which creates an `int` object:

In [11]:
```
def int_factory(s):
    i = int(s)
    return i

i = int_factory('100')
print(i)
```

```
100
```

This is overly-simplistic, but any function you write in the course of a normal program can be boiled down to this: take some arguments, do some operations, and create & return an object. With the above discussion in mind, though, there's nothing to stop us from creating an object of type `type` (that is, a class), and returning that instead -- this is a *metafunction:*

In [12]:
```
def class_factory():
    class Foo(object):
        pass
    return Foo

F = class_factory()
f = F()
print(type(f))
```

```
<class '__main__.Foo'>
```

Just as the function `int_factory` constructs an returns an instance of `int` , the function `class_factory` constructs and returns an instance of `type` : that is, a class.

But the above construction is a bit awkward: especially if we were going to do some more complicated logic when constructing `Foo` , it would be nice to avoid all the nested indentations and define the class in a more dynamic way. We can accomplish this by instantiating `Foo` from `type` directly:

In [13]:
```
def class_factory():
    return type('Foo', (), {})

F = class_factory()
f = F()
print(type(f))
```

```
<class '__main__.Foo'>
```

In fact, the construct

In [14]:
```
class MyClass(object):
    pass
```

is identical to the construct

In [15]:
```
MyClass = type('MyClass', (), {})
```

`MyClass` is an instance of type `type`, and that can be seen explicitly in the second version of the definition. A potential confusion arises from the more common use of `type` as a function to determine the type of an object, but you should strive to separate these two uses of the keyword in your mind: here `type` is a class (more precisely, a *metaclass*), and `MyClass` is an instance of `type`.

The arguments to the `type` constructor are:

type(name, bases, dct)

- `name` is a string giving the name of the class to be constructed
- `bases` is a tuple giving the parent classes of the class to be constructed
- `dct` is a dictionary of the attributes and methods of the class to be constructed

So, for example, the following two pieces of code have identical results:

In [16]:
```
class Foo(object):
    i = 4

class Bar(Foo):
    def get_i(self):
        return self.i

b = Bar()
print(b.get_i())
```

```
4
```

In [17]:
```
Foo = type('Foo', (), dict(i=4))

Bar = type('Bar', (Foo,), dict(get_i = lambda self: self.i))

b = Bar()
print(b.get_i())
```

```
4
```

This perhaps seems a bit over-complicated in the case of this contrived example, but it can be very powerful as a means of dynamically creating new classes on-the-fly.

Making Things Interesting: Custom Metaclasses

Now things get really fun. Just as we can inherit from and extend a class we've created, we can also inherit from and extend the `type` metaclass, and create custom behavior in our metaclass.

Example 1: Modifying Attributes

Let's use a simple example where we want to create an API in which the user can create a set of interfaces which contain a file object. Each interface should have a unique string ID, and contain an open file object. The user could then write specialized methods to accomplish certain tasks. There are certainly good ways to do this without delving into metaclasses, but such a simple example will (hopefully) elucidate what's going on.

First we'll create our interface meta class, deriving from `type`:

In [18]:
```
class InterfaceMeta(type):
    def __new__(cls, name, parents, dct):
        # create a class_id if it's not specified
        if 'class_id' not in dct:
            dct['class_id'] = name.lower()

        # open the specified file for writing
        if 'file' in dct:
            filename = dct['file']
            dct['file'] = open(filename, 'w')

        # we need to call type.__new__ to complete the initialization
        return super(InterfaceMeta, cls).__new__(cls, name, parents, dct)
```

Notice that we've modified the input dictionary (the attributes and methods of the class) to add a class id if it's not present, and to replace the filename with a file object pointing to that file name.

Now we'll use our `InterfaceMeta` class to construct and instantiate an Interface object:

In [19]:
```
Interface = InterfaceMeta('Interface', (), dict(file='tmp.txt'))

print(Interface.class_id)
print(Interface.file)
```

```
interface
<open file 'tmp.txt', mode 'w' at 0x21b8810>
```

This behaves as we'd expect: the `class_id` class variable is created, and the `file` class variable is replaced with an open file object. Still, the creation of the `Interface` class using `InterfaceMeta` directly is a bit clunky and difficult to read. This is where `__metaclass__` comes in and steals the show. We can accomplish the same thing by defining `Interface` this way:

In [20]:
```
class Interface(object):
    __metaclass__ = InterfaceMeta
    file = 'tmp.txt'

print(Interface.class_id)
print(Interface.file)
```

```
interface
<open file 'tmp.txt', mode 'w' at 0x21b8ae0>
```

by defining the `__metaclass__` attribute of the class, we've told the class that it should be constructed using `InterfaceMeta` rather than using `type`. To make this more definite, observe that the type of `Interface` is now `InterfaceMeta`:

In [21]:
```
type(Interface)
```

Out[21]:
```
__main__.InterfaceMeta
```

Furthermore, any class derived from `Interface` will now be constructed using the same metaclass:

In [22]:
```
class UserInterface(Interface):
    file = 'foo.txt'

print(UserInterface.file)
print(UserInterface.class_id)
```

```
<open file 'foo.txt', mode 'w' at 0x21b8c00>
userinterface
```

This simple example shows how metaclasses can be used to create powerful and flexible APIs for projects. For example, the Django project (https://www.djangoproject.com/) makes use of these sorts of constructions to allow concise declarations of very powerful extensions to their basic classes.

Example 2: Registering Subclasses

Another possible use of a metaclass is to automatically register all subclasses derived from a given base class. For example, you may have a basic interface to a database and wish for the user to be able to define their own interfaces, which are automatically stored in a master registry.

You might proceed this way:

In [23]:
```
class DBInterfaceMeta(type):
    # we use __init__ rather than __new__ here because we want
    # to modify attributes of the class *after* they have been
    # created
    def __init__(cls, name, bases, dct):
        if not hasattr(cls, 'registry'):
            # this is the base class.  Create an empty registry
            cls.registry = {}
        else:
            # this is a derived class.  Add cls to the registry
            interface_id = name.lower()
            cls.registry[interface_id] = cls

        super(DBInterfaceMeta, cls).__init__(name, bases, dct)
```

Our metaclass simply adds a `registry` dictionary if it's not already present, and adds the new class to the registry if the registry is already there. Let's see how this works:

In [24]:
```
class DBInterface(object):
    __metaclass__ = DBInterfaceMeta

print(DBInterface.registry)
```

```
{}
```

Now let's create some subclasses, and double-check that they're added to the registry:

In [25]:
```
class FirstInterface(DBInterface):
    pass

class SecondInterface(DBInterface):
    pass

class SecondInterfaceModified(SecondInterface):
    pass

print(DBInterface.registry)
```

```
{'firstinterface': <class '__main__.FirstInterface'>, 'secondinterface': <clas
```

It works as expected! This could be used in conjunction with a function that chooses implementations from the registry, and any user-defined `Interface`-derived objects would be automatically accounted for, without the user having to remember to manually register the new types.

Conclusion: When Should You Use Metaclasses?

I've gone through some examples of what metaclasses are, and some ideas about how they might be used to create very powerful and flexible APIs. Although metaclasses are in the background of everything you do in Python, the average coder rarely has to think about them.

But the question remains: when should you think about using custom metaclasses in your project? It's a complicated question, but there's a quotation floating around the web that addresses it quite succinctly:

> 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).
>
> – Tim Peters

In a way, this is a very unsatisfying answer: it's a bit reminiscent of the wistful and cliched explanation of the border between attraction and love: "well, you just... know!"

But I think Tim is right: in general, I've found that most tasks in Python that can be accomplished through use of custom metaclasses can also be accomplished more cleanly and with more clarity by other means. As programmers, we should always be careful to avoid being clever for the sake of cleverness alone, though it is admittedly an ever-present temptation.

I personally spent six years doing science with Python, writing code nearly on a daily basis, before I found a problem for which metaclasses were the natural solution. And it turns out Tim was right:

I just knew.

This post was written entirely in an IPython Notebook: the notebook file is available for download here (http://jakevdp.github.com/downloads/notebooks/MetaClasses.ipynb). For more information on blogging with notebooks in octopress, see my previous post (http://jakevdp.github.com/blog/2012/10/04/blogging-with-ipython/) on the subject.

metaclasses (http://jakevdp.github.io/tag/metaclasses.html)

tutorial (http://jakevdp.github.io/tag/tutorial.html)

Comments

Bob Stein • 3 years ago

Fantastically good stuff. Your writing is a handrail in a hailstorm; I feel safely and competently led to understanding. Mostly. I was left craving a sentence or two about your six-year-ripe application of metaclass.

Seems as if your sequel to the **Tim Peters** principle could be to say: just dive in and use classes for all their worth and one day, after you've worked them under your skin deeply, you'll find yourself struggling with a problem so complex and surreal you'll just realize, oh, I need a metaclass.

The only metaclass I found in **django** was SubfieldBase, apparently unused and deprecated.

1 • Reply • Share ›

Matthew Schinckel → Bob Stein • 3 years ago

`django.db.models.base.ModelBase` is also a metaclass: it's what makes the fields work the way they do (i.e., that you get an attribute with the field name that can do nice

things).

A search of `(type)` also shows metaclasses related to queries, widgets and forms.

1 • Reply • Share ›

Bob Stein ➚ Matthew Schinckel • 3 years ago

Aha, they use `six.with_metaclass` for Py2 and Py3 compatibility. Thanks!

Reply • Share ›

Dave • 4 years ago

Thank you for this amazing article. With this knowledge, I was able to create an abstract plugin library for use in my applications.

1 • Reply • Share ›

MySchizoBuddy • 5 years ago

you haven't made a case for why should i be using metaclasses. Can you provide an example where using metaclasses is beneficial and why?

1 • Reply • Share ›

Matthew Schinckel • 5 years ago

I've been using the registry pattern, using a metaclass, but hadn't thought to have the registry as an attribute on the base class. Thanks!

1 • Reply • Share ›

papaKenya • a year ago

From the article, this is my understanding of when to use and not to use metaclasses: The metaclass only comes into the picture when you want to create a class that inherits from several base classes. If all the bases have the same type, then that will be used and there is no need to specify a metaclass - this is the case most of the time. However, if the bases have different types that are not related, then Python cannot figure out which type object to use. In this case specifying a metaclass is required, and this metaclass must be a subclass of the type of each base.
Thank you for the article,looking forward for somebody to help me understand if my conclusion is right.

Reply • Share ›

jakevdp Mod ➚ papaKenya • a year ago

I'm not sure that's the right take: "a class that inherits from several base classes" is an example of multiple inheritance, and that can be done without specifying a metaclass. The purpose of a custom metaclass is to override the mechanism by which class instances are created.

^ | v • Reply • Share ›

KIDJourney • a year ago

Good Post :)

^ | v • Reply • Share ›

bjd2385 • a year ago

What was that single problem about? I'm actually curious.

^ | v • Reply • Share ›

Jam • 3 years ago

cool

^ | v • Reply • Share ›

Dennis • 3 years ago

Brilliant. NOW I get it!

^ | v • Reply • Share ›

James • 5 years ago

You actually can create classes (besides type) that are instances of themselves in pure python:

```
class MC(type): pass
class Circular(MC, metaclass=MC): pass # or in python 2, class Circular(MC):
__metaclass__=MC
Circular.__class__ = Circular
assert type(Circular) is Circular
```

I'm not sure it's possible with a class that isn't a subclass of type, though, and I can't imagine it ever being useful

imagine it ever being useful.

Reply • Share ›

harold • 5 years ago

Great writeup!
One small erratum:
You probably shouldn't use dct={} as the default value in your __new__ definition.

As you probably know, this actually just creates a single dict instance when the program starts,
and each dct-less call to __new__ will share the same dict.
I think maybe this ends up working out in this case because the dict gets copied before making the class, but it's very lucky/fragile.

Reply • Share ›

jakevdp Mod ➚ harold • 5 years ago

Good point! I'll change that. I think that in this case, __new__ is never called without arguments, so the default values aren't needed anyway.

Reply • Share ›

nearhan • 5 years ago

Excellent article now I finally get meta-classes and the whole type - object relation. Thanks!

Question!

when using the type function to create a class object, does it automatically inherit from the object object like a new style class?

your example

class Foo(object):
i = 4

&

Foo = type('Foo', (), dict=(1=4))

Does that mean the type defined foo automatically inherits from object like a new style class?

Reply • Share ›

jakevdp Mod ➚ nearhan • 5 years ago

Good question - yes, deriving from `object` is simply a backward-compatible way of specifying a "new style class" (introduced in Python 2.2), and any class created from `type` is new-style by definition. In Python 3, this is no longer necessary as all classes are new-style classes.

Reply • Share ›