Metaclasses

February 19, 2026

1 Metaclasses - originally by Michael Foord

Metaclasses have a reputation for being 'deep-black-magic' in Python. The cases where you need them are genuinely rare, but the basic principles are surprisingly easy to understand.

- Everything is an object
- Everything has a type
- No real difference between 'class' and 'type'
- Classes are objects
- Their type is *type*
- Typically the term *type* is used for the built-in types and the term *class* for user-defined classes. Since Python 2.2 there has been no real difference and 'class' and 'type' are synonyms.

[1]:
```
class Something:
    pass

print(Something)
```

```
<class '__main__.Something'>
```

[2]:
```
print(type(int))  # A built in type
print(type(Something)) # A user defined type
print(type(type))
```

```
<class 'type'>
<class 'type'>
<class 'type'>
```

Here we can see that a class created at the interactive interpreter is a first class object.

The Class of a Class is… Its metaclass…

Just as an object is an instance of its class; a class is an instance of its metaclass.

The metaclass is called to create the class.

In exactly the same way as any other object in Python.

So when you create a class… The interpreter calls the metaclass to create it…

For a normal class that inherits from object this means that type is called to create the class:

[ ]:
```

```

[3]:
```
help(type)
```

```
Help on class type in module builtins:

class type(object)
 |  type(object) -> the object's type
 |  type(name, bases, dict, **kwds) -> a new type
 |
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __dir__(self, /)
 |      Specialized __dir__ implementation for types.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __instancecheck__(self, instance, /)
 |      Check if an object is an instance.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __sizeof__(self, /)
 |      Return memory consumption of the type object.
 |
 |  __subclasscheck__(self, subclass, /)
 |      Check if a class is a subclass.
 |
 |  __subclasses__(self, /)
 |      Return a list of immediate subclasses.
```

```
|
|  mro(self, /)
|      Return a type's method resolution order.
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  __prepare__(…)
|      __prepare__() -> dict
|      used to create the namespace for the class statement
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs)
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __abstractmethods__
|
|  __annotations__
|
|  __dict__
|
|  __text_signature__
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __base__ = <class 'object'>
|      The base class of the class hierarchy.
|
|      When called, it accepts no arguments and returns a new featureless
|      instance that has no instance attributes and cannot be given any.
|
|
|  __bases__ = (<class 'object'>,)
|
|  __basicsize__ = 904
|
|  __dictoffset__ = 264
|
|  __flags__ = 2148031744
|
|  __itemsize__ = 40
|
```

```
|  __mro__ = (<class 'type'>, <class 'object'>)
|
|  __weakrefoffset__ = 368
```

Help on class type in module **builtin**:

class type(object) | type(object) -> the object's type | **type(name, bases, dict) -> a new type**

It is this second usage of type that is important. When the Python interpreter executes a class statement (like in the example with the interactive interpreter from a couple of sections back), it calls type with the following arguments:

- The name of the class as a string
- A tuple of base classes - for our example this is the 'one-pl' [1] (object,)
- A dictionary containing members of the class (class attributes, methods, etc) mapped by their names

2 Easy to Demonstrate

[4]:
```
def __init__(self):
    self.message = 'Hello World'

def say_hello(self):
    print(self.message)

attrs = {'__init__': __init__, 'say_hello': say_hello}
bases = (object,)
Hello = type('Jello', bases, attrs)
```

[5]:
```
print(Hello)
```

```
<class '__main__.Jello'>
```

[6]:
```
h = Hello()
h.say_hello()
print(type(h))
```

```
Hello World
<class '__main__.Jello'>
```

This code creates a dictionary of class attributes, and then calls type to create a class called Hello.

[7]:
```
class Hello:
    def __init__(self):
        self.message = 'Hello World'

    def say_hello(self):
        print(self.message)
```

[8]:
```
h = Hello()
h.say_hello()
print(type(h))
```

```
Hello World
<class '__main__.Hello'>
```

3 The Magic of metaclass

We can provide a custom metaclass by setting **metaclass** in a class definition to any callable that takes the same arguments as type.

The normal way to do this is to inherit from type:

[9]:
```
class PointlessMetaclass(type):
    def __new__(meta, name, bases, namespace):
        print('MetaBase.__new__\nCreating class object', end='\n\n')
        # Do stuff ...
        return super().__new__(meta, name, bases, namespace)

    def __init__(cls, name, bases, nmspc):
        print('MetaBase.__init__\nInitialising class object')
        # Do stuff ...
        super().__init__(name, bases, nmspc)
```

The important thing is that inside the body of the **new** method we have access to the arguments passed to create the new class. We can introspect the dictionary of attributes and modify, add or remove members.

It is important to override **___new___** rather than **___init___. When you instantiate a class both** ___init___** and **___new___** are called. **___init___** initialises an instance - but **___new___** is responsible for creating it. So if our metaclass is going to customise class creation we need to override **___new___** on type.

The reason to use a new type rather than just a factory function is that if you use a factory function (that just calls type) then the metaclass won't be inherited.

4 In Action...

[10]:
```
class WhizzBang(metaclass=PointlessMetaclass):
    def __init__(self):
        pass

WhizzBang
```

```
MetaBase.__new__
Creating class object
```

```
MetaBase.__init__
Initialising class object
```

[10]:
```
__main__.WhizzBang
```

[11]:
```
type(WhizzBang)
```

[11]:
```
__main__.PointlessMetaclass
```

[12]:
```
print(WhizzBang.__class__)
print(WhizzBang.__class__.__class__)
```

```
<class '__main__.PointlessMetaclass'>
<class 'type'>
```

WhizzBang is a class, but instead of being an instance of type the class object is now an instance of our custom metaclass…

5 What can we do with this?

Our metaclass will be called whenever a new class is created that uses it. Here are some ideas:

- Decorate all methods in a class for logging, or profiling
- Automatically mix-in new methods
- Register classes as they are created (Auto-register plugins or create a db schema from class members for example)
- Provide interface registration, auto-discovery of features and interface adaptation
- Class verification: prevent subclassing, verify all methods have docstrings
- The important things is that the class is only actually created by the final call to type in the metaclass - so you are free to modify the dictionary of attributes as you see fit (and the name plus the tuple of base classes of course)

Several of the popular Python ORM (Object Relational Mappers for working with databases) use metaclasses in these ways.

Metaclasses are inherited so you can provide a base-class that uses your metaclass and sub-classes inherit it without explicitly having to declare it.

6 But…

I have only once used it in production. We built a business object model that applied type constraints on Python objects. I've ever needed to use one in production code… (I have used them for profiling and we make extensive use of them in Ironclad - but I didn't write these.)

With modern versions of Python you can now use class decorators to achieve a lot of the things that previously you might have used metaclasses for.

For a truly awful example (with a slightly more in depth but still easy-to-digest look at metaclasses) see The Selfless Metaclass by Michael Foord. It does bytecode and method signature rewriting to avoid the need to explicitly declare self. Smile

[1] A ‘one-pl’ is tuple with only one element…