**Classes as objects**

Before understanding metaclasses, you need to master classes in Python. And Python has a very peculiar idea of what classes are, borrowed from the Smalltalk language.

In most languages, classes are just pieces of code that describe how to produce an object. That's kinda true in Python too:

>>> class ObjectCreator(object):

... pass

...

>>> my\_object = ObjectCreator()

>>> print(my\_object)

<\_\_main\_\_.ObjectCreator object at 0x8974f2c>

But classes are more than that in Python. Classes are objects too.

Yes, objects.

As soon as you use the keyword class, Python executes it and creates an OBJECT. The instruction

>>> class ObjectCreator(object):

... pass

...

creates in memory an object with the name "ObjectCreator".

**This object (the class) is itself capable of creating objects (the instances), and this is why it's a class**.

But still, it's an object, and therefore:

* you can assign it to a variable
* you can copy it
* you can add attributes to it
* you can pass it as a function parameter

e.g.:

>>> print(ObjectCreator) # you can print a class because it's an object

<class '\_\_main\_\_.ObjectCreator'>

>>> def echo(o):

... print(o)

...

>>> echo(ObjectCreator) # you can pass a class as a parameter

<class '\_\_main\_\_.ObjectCreator'>

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

False

>>> ObjectCreator.new\_attribute = 'foo' # you can add attributes to a class

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

True

>>> print(ObjectCreator.new\_attribute)

foo

>>> ObjectCreatorMirror = ObjectCreator # you can assign a class to a variable

>>> print(ObjectCreatorMirror.new\_attribute)

foo

>>> print(ObjectCreatorMirror())

<\_\_main\_\_.ObjectCreator object at 0x8997b4c>

**Creating classes dynamically**

Since classes are objects, you can create them on the fly, like any object.

First, you can create a class in a function using class:

>>> def choose\_class(name):

... if name == 'foo':

... class Foo(object):

... pass

... return Foo # return the class, not an instance

... else:

... class Bar(object):

... pass

... return Bar

...

>>> MyClass = choose\_class('foo')

>>> print(MyClass) # the function returns a class, not an instance

<class '\_\_main\_\_.Foo'>

>>> print(MyClass()) # you can create an object from this class

<\_\_main\_\_.Foo object at 0x89c6d4c>

But it's not so dynamic, since you still have to write the whole class yourself.

Since classes are objects, they must be generated by something.

When you use the class keyword, Python creates this object automatically. But as with most things in Python, it gives you a way to do it manually.

Remember the function type? The good old function that lets you know what type an object is:

>>> print(type(1))

<type 'int'>

>>> print(type("1"))

<type 'str'>

>>> print(type(ObjectCreator))

<type 'type'>

>>> print(type(ObjectCreator()))

<class '\_\_main\_\_.ObjectCreator'>

Well, [type](http://docs.python.org/2/library/functions.html#type) has a completely different ability, it can also create classes on the fly. type can take the description of a class as parameters, and return a class.

(I know, it's silly that the same function can have two completely different uses according to the parameters you pass to it. It's an issue due to backwards compatibility in Python)

type works this way:

type(name of the class,

tuple of the parent class (for inheritance, can be empty),

dictionary containing attributes names and values)

e.g.:

>>> class MyShinyClass(object):

... pass

can be created manually this way:

>>> MyShinyClass = type('MyShinyClass', (), {}) # returns a class object

>>> print(MyShinyClass)

<class '\_\_main\_\_.MyShinyClass'>

>>> print(MyShinyClass()) # create an instance with the class

<\_\_main\_\_.MyShinyClass object at 0x8997cec>

You'll notice that we use "MyShinyClass" as the name of the class and as the variable to hold the class reference. They can be different, but there is no reason to complicate things.

type accepts a dictionary to define the attributes of the class. So:

>>> class Foo(object):

... bar = True

Can be translated to:

>>> Foo = type('Foo', (), {'bar':True})

And used as a normal class:

>>> print(Foo)

<class '\_\_main\_\_.Foo'>

>>> print(Foo.bar)

True

>>> f = Foo()

>>> print(f)

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print(f.bar)

True

And of course, you can inherit from it, so:

>>> class FooChild(Foo):

... pass

would be:

>>> FooChild = type('FooChild', (Foo,), {})

>>> print(FooChild)

<class '\_\_main\_\_.FooChild'>

>>> print(FooChild.bar) # bar is inherited from Foo

True

Eventually you'll want to add methods to your class. Just define a function with the proper signature and assign it as an attribute.

>>> def echo\_bar(self):

... print(self.bar)

...

>>> FooChild = type('FooChild', (Foo,), {'echo\_bar': echo\_bar})

>>> hasattr(Foo, 'echo\_bar')

False

>>> hasattr(FooChild, 'echo\_bar')

True

>>> my\_foo = FooChild()

>>> my\_foo.echo\_bar()

True

And you can add even more methods after you dynamically create the class, just like adding methods to a normally created class object.

>>> def echo\_bar\_more(self):

... print('yet another method')

...

>>> FooChild.echo\_bar\_more = echo\_bar\_more

>>> hasattr(FooChild, 'echo\_bar\_more')

True

You see where we are going: in Python, classes are objects, and you can create a class on the fly, dynamically.

This is what Python does when you use the keyword class, and it does so by using a metaclass.

**What are metaclasses (finally)**

Metaclasses are the 'stuff' that creates classes.

You define classes in order to create objects, right?

But we learned that Python classes are objects.

Well, metaclasses are what create these objects. They are the classes' classes, you can picture them this way:

MyClass = MetaClass()

my\_object = MyClass()

You've seen that type lets you do something like this:

MyClass = type('MyClass', (), {})

It's because the function type is in fact a metaclass. type is the metaclass Python uses to create all classes behind the scenes.

Now you wonder why the heck is it written in lowercase, and not Type?

Well, I guess it's a matter of consistency with str, the class that creates strings objects, and int the class that creates integer objects. type is just the class that creates class objects.

You see that by checking the \_\_class\_\_ attribute.

Everything, and I mean everything, is an object in Python. That includes ints, strings, functions and classes. All of them are objects. And all of them have been created from a class:

>>> age = 35

>>> age.\_\_class\_\_

<type 'int'>

>>> name = 'bob'

>>> name.\_\_class\_\_

<type 'str'>

>>> def foo(): pass

>>> foo.\_\_class\_\_

<type 'function'>

>>> class Bar(object): pass

>>> b = Bar()

>>> b.\_\_class\_\_

<class '\_\_main\_\_.Bar'>

Now, what is the \_\_class\_\_ of any \_\_class\_\_ ?

>>> age.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> name.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> foo.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> b.\_\_class\_\_.\_\_class\_\_

<type 'type'>

So, a metaclass is just the stuff that creates class objects.

You can call it a 'class factory' if you wish.

type is the built-in metaclass Python uses, but of course, you can create your own metaclass.

**The**[**\_\_metaclass\_\_**](http://docs.python.org/2/reference/datamodel.html?highlight=__metaclass__#__metaclass__)**attribute**

You can add a \_\_metaclass\_\_ attribute when you write a class:

class Foo(object):

\_\_metaclass\_\_ = something...

[...]

If you do so, Python will use the metaclass to create the class Foo.

Careful, it's tricky.

You write class Foo(object) first, but the class object Foo is not created in memory yet.

Python will look for \_\_metaclass\_\_ in the class definition. If it finds it, it will use it to create the object class Foo. If it doesn't, it will use type to create the class.

Read that several times.

When you do:

class Foo(Bar):

pass

Python does the following:

Is there a \_\_metaclass\_\_ attribute in Foo?

If yes, create in memory a class object (I said a class object, stay with me here), with the name Fooby using what is in \_\_metaclass\_\_.

If Python can't find \_\_metaclass\_\_, it will look for a \_\_metaclass\_\_ at the MODULE level, and try to do the same (but only for classes that don't inherit anything, basically old-style classes).

Then if it can't find any \_\_metaclass\_\_ at all, it will use the Bar's (the first parent) own metaclass (which might be the default type) to create the class object.

Be careful here that the \_\_metaclass\_\_ attribute will not be inherited, the metaclass of the parent (Bar.\_\_class\_\_) will be. If Bar used a \_\_metaclass\_\_ attribute that created Bar with type() (and not type.\_\_new\_\_()), the subclasses will not inherit that behavior.

Now the big question is, what can you put in \_\_metaclass\_\_ ?

The answer is: something that can create a class.

And what can create a class? type, or anything that subclasses or uses it.

**Custom metaclasses**

The main purpose of a metaclass is to change the class automatically, when it's created.

You usually do this for APIs, where you want to create classes matching the current context.

Imagine a stupid example, where you decide that all classes in your module should have their attributes written in uppercase. There are several ways to do this, but one way is to set \_\_metaclass\_\_ at the module level.

This way, all classes of this module will be created using this metaclass, and we just have to tell the metaclass to turn all attributes to uppercase.

Luckily, \_\_metaclass\_\_ can actually be any callable, it doesn't need to be a formal class (I know, something with 'class' in its name doesn't need to be a class, go figure... but it's helpful).

So we will start with a simple example, by using a function.

# the metaclass will automatically get passed the same argument

# that you usually pass to `type`

def upper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):

"""

Return a class object, with the list of its attribute turned

into uppercase.

"""

# pick up any attribute that doesn't start with '\_\_' and uppercase it

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

# let `type` do the class creation

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

\_\_metaclass\_\_ = upper\_attr # this will affect all classes in the module

class Foo(): # global \_\_metaclass\_\_ won't work with "object" though

# but we can define \_\_metaclass\_\_ here instead to affect only this class

# and this will work with "object" children

bar = 'bip'

print(hasattr(Foo, 'bar'))

# Out: False

print(hasattr(Foo, 'BAR'))

# Out: True

f = Foo()

print(f.BAR)

# Out: 'bip'

Now, let's do exactly the same, but using a real class for a metaclass:

# remember that `type` is actually a class like `str` and `int`

# so you can inherit from it

class UpperAttrMetaclass(type):

# \_\_new\_\_ is the method called before \_\_init\_\_

# it's the method that creates the object and returns it

# while \_\_init\_\_ just initializes the object passed as parameter

# you rarely use \_\_new\_\_, except when you want to control how the object

# is created.

# here the created object is the class, and we want to customize it

# so we override \_\_new\_\_

# you can do some stuff in \_\_init\_\_ too if you wish

# some advanced use involves overriding \_\_call\_\_ as well, but we won't

# see this

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

But this is not really OOP. We call type directly and we don't override or call the parent \_\_new\_\_. Let's do it:

class UpperAttrMetaclass(type):

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

# reuse the type.\_\_new\_\_ method

# this is basic OOP, nothing magic in there

return type.\_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, uppercase\_attr)

You may have noticed the extra argument upperattr\_metaclass. There is nothing special about it: \_\_new\_\_ always receives the class it's defined in, as first parameter. Just like you have self for ordinary methods which receive the instance as first parameter, or the defining class for class methods.

Of course, the names I used here are long for the sake of clarity, but like for self, all the arguments have conventional names. So a real production metaclass would look like this:

class UpperAttrMetaclass(type):

def \_\_new\_\_(cls, clsname, bases, dct):

uppercase\_attr = {}

for name, val in dct.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return type.\_\_new\_\_(cls, clsname, bases, uppercase\_attr)

We can make it even cleaner by using super, which will ease inheritance (because yes, you can have metaclasses, inheriting from metaclasses, inheriting from type):

class UpperAttrMetaclass(type):

def \_\_new\_\_(cls, clsname, bases, dct):

uppercase\_attr = {}

for name, val in dct.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return super(UpperAttrMetaclass, cls).\_\_new\_\_(cls, clsname, bases, uppercase\_attr)

That's it. There is really nothing more about metaclasses.

The reason behind the complexity of the code using metaclasses is not because of metaclasses, it's because you usually use metaclasses to do twisted stuff relying on introspection, manipulating inheritance, vars such as \_\_dict\_\_, etc.

Indeed, metaclasses are especially useful to do black magic, and therefore complicated stuff. But by themselves, they are simple:

* intercept a class creation
* modify the class
* return the modified class

**Why would you use metaclasses classes instead of functions?**

Since \_\_metaclass\_\_ can accept any callable, why would you use a class since it's obviously more complicated?

There are several reasons to do so:

* The intention is clear. When you read UpperAttrMetaclass(type), you know what's going to follow
* You can use OOP. Metaclass can inherit from metaclass, override parent methods. Metaclasses can even use metaclasses.
* Subclasses of a class will be instances of its metaclass if you specified a metaclass-class, but not with a metaclass-function.
* You can structure your code better. You never use metaclasses for something as trivial as the above example. It's usually for something complicated. Having the ability to make several methods and group them in one class is very useful to make the code easier to read.
* You can hook on \_\_new\_\_, \_\_init\_\_ and \_\_call\_\_. Which will allow you to do different stuff. Even if usually you can do it all in \_\_new\_\_, some people are just more comfortable using \_\_init\_\_.
* These are called metaclasses, damn it! It must mean something!

**Why would you use metaclasses?**

Now the big question. Why would you use some obscure error prone feature?

Well, usually you don't:

Metaclasses are deeper magic that 99% of users should never 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).

*Python Guru Tim Peters*

The main use case for a metaclass is creating an API. A typical example of this is the Django ORM.

It allows you to define something like this:

class Person(models.Model):

name = models.CharField(max\_length=30)

age = models.IntegerField()

But if you do this:

guy = Person(name='bob', age='35')

print(guy.age)

It won't return an IntegerField object. It will return an int, and can even take it directly from the database.

This is possible because models.Model defines \_\_metaclass\_\_ and it uses some magic that will turn the Person you just defined with simple statements into a complex hook to a database field.

Django makes something complex look simple by exposing a simple API and using metaclasses, recreating code from this API to do the real job behind the scenes.

**The last word**

First, you know that classes are objects that can create instances.

Well in fact, classes are themselves instances. Of metaclasses.

>>> class Foo(object): pass

>>> id(Foo)

142630324

Everything is an object in Python, and they are all either instances of classes or instances of metaclasses.

Except for type.

type is actually its own metaclass. This is not something you could reproduce in pure Python, and is done by cheating a little bit at the implementation level.

Secondly, metaclasses are complicated. You may not want to use them for very simple class alterations. You can change classes by using two different techniques:

* [monkey patching](http://en.wikipedia.org/wiki/Monkey_patch)
* class decorators

99% of the time you need class alteration, you are better off using these.

But 98% of the time, you don't need class alteration at all.

**def** \_\_init\_\_(self, x):  
    self.x = x  
  
**def** printX(self):  
    **print** self.x  
  
Test = type('Test', (object,), {'\_\_init\_\_': \_\_init\_\_, 'printX': printX})

## [The Most Basic Example](http://www.voidspace.org.uk/python/articles/metaclasses.shtml#id7)

So we can create a very simple metaclass which does nothing.

**class** PointlessMetaClass(type):  
    **def** \_\_new\_\_(meta, classname, bases, classDict):  
        **return** type.\_\_new\_\_(meta, classname, bases, classDict)

## [Using Metaclasses](http://www.voidspace.org.uk/python/articles/metaclasses.shtml#id8)

You can set the metaclass for a class by setting its \_\_metaclass\_\_ attribute.

We'll adapt the example for above so that we can see what's going on.

**class** MetaClass(type):  
    **def** \_\_new\_\_(meta, classname, bases, classDict):  
        **print** 'Class Name:', classname  
        **print** 'Bases:', bases  
        **print** 'Class Attributes', classDict  
        **return** type.\_\_new\_\_(meta, classname, bases, classDict)  
  
**class** Test(object):  
  
    \_\_metaclass\_\_ = MetaClass  
  
    **def** \_\_init\_\_(self):  
        **pass**  
  
    **def** method(self):  
        **pass**  
  
    classAttribute = 'Something'

If you run this you will see something like :

Class Name: Test

Bases: (<type 'object'>,)

Class Attributes {'\_\_module\_\_': '\_\_main\_\_',

'\_\_metaclass\_\_': <class '\_\_main\_\_.MetaClass'>,

'method': <function method at 0x00B412B0>,

'\_\_init\_\_': <function \_\_init\_\_ at 0x00B41070>,

'classAttribute': 'Something'}

You can also set the metaclass at the module level. If you set the variable \_\_metaclass\_\_ in a module, it will be used for all following class definitions that don't have an explicit metaclasses. New style classes inherit their metaclasses from object, so that means all old style classes. So you can do things like make all classes into new style classes by setting \_\_metaclass\_\_ = type.

Links realted to metaclasses:

<http://blog.thedigitalcatonline.com/blog/2014/10/14/decorators-and-metaclasses/>

<http://jakevdp.github.io/blog/2012/12/01/a-primer-on-python-metaclasses/>

http://www.voidspace.org.uk/python/articles/metaclasses.shtml