[**http://python.jobbole.com/86821/**](http://python.jobbole.com/86821/)

[**http://blog.jobbole.com/21351/**](http://blog.jobbole.com/21351/)

[**https://www.liaoxuefeng.com/**](https://www.liaoxuefeng.com/)

**动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。**

**深刻理解Python中的元类(metaclass)**

2014/08/10 · [Python](http://blog.jobbole.com/category/python/), [开发](http://blog.jobbole.com/category/programmer/) · [34 评论](http://blog.jobbole.com/21351/#article-comment)· [metaclass](http://blog.jobbole.com/tag/metaclass/), [Python](http://blog.jobbole.com/tag/python/)

本文由 [伯乐在线](http://blog.jobbole.com/) - [bigship](http://www.jobbole.com/members/bigship) 翻译。未经许可，禁止转载！  
英文出处：[stackoverflow](http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python)。欢迎加入[翻译组](http://group.jobbole.com/category/feedback/trans-team/)。

**译注：**这是一篇在[Stack overflow](http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python)上很热的帖子。提问者自称已经掌握了有关Python OOP编程中的各种概念，但始终觉得元类(metaclass)难以理解。他知道这肯定和自省有关，但仍然觉得不太明白，希望大家可以给出一些实际的例子和代码片段以帮助理解，以及在什么情况下需要进行元编程。于是e-satis同学给出了神一般的回复，该回复获得了985点的赞同点数，更有人评论说这段回复应该加入到Python的官方文档中去。而e-satis同学本人在Stack Overflow中的声望积分也高达64271分。以下就是这篇精彩的回复（提示：非常长）

[](http://jbcdn2.b0.upaiyun.com/2012/06/python-icon.jpg)

**类也是对象**

在理解元类之前，你需要先掌握Python中的类。Python中类的概念借鉴于Smalltalk，这显得有些奇特。在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成立：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> class ObjectCreator(object):  …       pass  …  >>> my\_object = ObjectCreator()  >>> print my\_object  <\_\_main\_\_.ObjectCreator object at 0x8974f2c> |

但是，Python中的类还远不止如此。类同样也是一种对象。是的，没错，就是对象。只要你使用关键字class，Python解释器在执行的时候就会创建一个对象。下面的代码段：

Python



|  |  |
| --- | --- |
| 1  2  3 | >>> class ObjectCreator(object):  …       pass  … |

将在内存中创建一个对象，名字就是ObjectCreator。**这个对象（类）自身拥有创建对象（类实例）的能力，而这就是为什么它是一个类的原因。**但是，它的本质仍然是一个对象，于是乎你可以对它做如下的操作：

1)   你可以将它赋值给一个变量

2)   你可以拷贝它

3)   你可以为它增加属性

4)   你可以将它作为函数参数进行传递

下面是示例：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | >>> print ObjectCreator     # 你可以打印一个类，因为它其实也是一个对象  <class '\_\_main\_\_.ObjectCreator'>  >>> def echo(o):  …       print o  …  >>> echo(ObjectCreator)                 # 你可以将类做为参数传给函数  <class '\_\_main\_\_.ObjectCreator'>  >>> print hasattr(ObjectCreator, 'new\_attribute')  Fasle  >>> ObjectCreator.new\_attribute = 'foo' #  你可以为类增加属性  >>> print hasattr(ObjectCreator, 'new\_attribute')  True  >>> print ObjectCreator.new\_attribute  foo  >>> ObjectCreatorMirror = ObjectCreator # 你可以将类赋值给一个变量  >>> print ObjectCreatorMirror()  <\_\_main\_\_.ObjectCreator object at 0x8997b4c> |

**动态地创建类**

因为类也是对象，你可以在运行时动态的创建它们，就像其他任何对象一样。首先，你可以在函数中创建类，使用class关键字即可。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> def choose\_class(name):  …       if name == 'foo':  …           class Foo(object):  …               pass  …           return Foo     # 返回的是类，不是类的实例  …       else:  …           class Bar(object):  …               pass  …           return Bar  …  >>> MyClass = choose\_class('foo')  >>> print MyClass              # 函数返回的是类，不是类的实例  <class '\_\_main\_\_'.Foo>  >>> print MyClass()            # 你可以通过这个类创建类实例，也就是对象  <\_\_main\_\_.Foo object at 0x89c6d4c> |

但这还不够动态，因为你仍然需要自己编写整个类的代码。由于类也是对象，所以它们必须是通过什么东西来生成的才对。当你使用class关键字时，Python解释器自动创建这个对象。但就和Python中的大多数事情一样，Python仍然提供给你手动处理的方法。还记得内建函数type吗？这个古老但强大的函数能够让你知道一个对象的类型是什么，就像这样：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> print type(1)  <type 'int'>  >>> print type("1")  <type 'str'>  >>> print type(ObjectCreator)  <type 'type'>  >>> print type(ObjectCreator())  <class '\_\_main\_\_.ObjectCreator'> |

这里，type有一种完全不同的能力，它也能动态的创建类。type可以接受一个类的描述作为参数，然后返回一个类。（我知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在Python中是为了保持向后兼容性）

type可以像这样工作：

Python



|  |  |
| --- | --- |
| 1 | type(类名, 父类的元组（针对继承的情况，可以为空），包含属性的字典（名称和值）) |

比如下面的代码：

Python



|  |  |
| --- | --- |
| 1  2 | >>> class MyShinyClass(object):  …       pass |

可以手动像这样创建：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> MyShinyClass = type('MyShinyClass', (), {})  # 返回一个类对象  >>> print MyShinyClass  <class '\_\_main\_\_.MyShinyClass'>  >>> print MyShinyClass()  #  创建一个该类的实例  <\_\_main\_\_.MyShinyClass object at 0x8997cec> |

你会发现我们使用“MyShinyClass”作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。

type 接受一个字典来为类定义属性，因此

Python



|  |  |
| --- | --- |
| 1  2 | >>> class Foo(object):  …       bar = True |

可以翻译为：

Python



|  |  |
| --- | --- |
| 1 | >>> Foo = type('Foo', (), {'bar':True}) |

并且可以将Foo当成一个普通的类一样使用：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> print Foo  <class '\_\_main\_\_.Foo'>  >>> print Foo.bar  True  >>> f = Foo()  >>> print f  <\_\_main\_\_.Foo object at 0x8a9b84c>  >>> print f.bar  True |

当然，你可以向这个类继承，所以，如下的代码：

Python



|  |  |
| --- | --- |
| 1  2 | >>> class FooChild(Foo):  …       pass |

就可以写成：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> FooChild = type('FooChild', (Foo,),{})  >>> print FooChild  <class '\_\_main\_\_.FooChild'>  >>> print FooChild.bar   # bar属性是由Foo继承而来  True |

最终你会希望为你的类增加方法。只需要定义一个有着恰当签名的函数并将其作为属性赋值就可以了。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> 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 |

你可以看到，在Python中，类也是对象，你可以动态的创建类。这就是当你使用关键字class时Python在幕后做的事情，而这就是通过元类来实现的。

**到底什么是元类（终于到主题了）**

元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象，不是吗？但是我们已经学习到了Python中的类也是对象。好吧，元类就是用来创建这些类（对象）的，元类就是类的类，你可以这样理解 为：

Python

|  |  |
| --- | --- |
| 1  2 | MyClass = MetaClass()  MyObject = MyClass() |

你已经看到了type可以让你像这样做：

Python

|  |  |
| --- | --- |
| 1 | MyClass = type('MyClass', (), {}) |

这是因为函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。现在你想知道那为什么type会全部采用小写形式而不是Type呢？好吧，我猜这是为了和str保持一致性，str是用来创建字符串对象的类，而int是用来创建整数对象的类。type就是创建类对象的类。你可以通过检查\_\_class\_\_属性来看到这一点。Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> 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'> |

现在，对于任何一个\_\_class\_\_的\_\_class\_\_属性又是什么呢？

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> a.\_\_class\_\_.\_\_class\_\_  <type 'type'>  >>> age.\_\_class\_\_.\_\_class\_\_  <type 'type'>  >>> foo.\_\_class\_\_.\_\_class\_\_  <type 'type'>  >>> b.\_\_class\_\_.\_\_class\_\_  <type 'type'> |

因此，元类就是创建类这种对象的东西。如果你喜欢的话，可以把元类称为“类工厂”（不要和工厂类搞混了:D） type就是Python的内建元类，当然了，你也可以创建自己的元类。

**\_\_metaclass\_\_属性**

你可以在写一个类的时候为其添加\_\_metaclass\_\_属性。

Python

|  |  |
| --- | --- |
| 1  2  3 | class Foo(object):  \_\_metaclass\_\_ = something…  […] |

如果你这么做了，Python就会用元类来创建类Foo。小心点，这里面有些技巧。你首先写下class Foo(object)，但是类对象Foo还没有在内存中创建。Python会在类的定义中寻找\_\_metaclass\_\_属性，如果找到了，Python就会用它来创建类Foo，如果没有找到，就会用内建的type来创建这个类。把下面这段话反复读几次。当你写如下代码时 :

Python



|  |  |
| --- | --- |
| 1  2 | class Foo(Bar):      pass |

Python做了如下的操作：

Foo中有\_\_metaclass\_\_这个属性吗？如果是，Python会在内存中通过\_\_metaclass\_\_创建一个名字为Foo的类对象（我说的是类对象，请紧跟我的思路）。如果Python没有找到\_\_metaclass\_\_，它会继续在Bar（父类）中寻找\_\_metaclass\_\_属性，并尝试做和前面同样的操作。如果Python在任何父类中都找不到\_\_metaclass\_\_，它就会在模块层次中去寻找\_\_metaclass\_\_，并尝试做同样的操作。如果还是找不到\_\_metaclass\_\_,Python就会用内置的type来创建这个类对象。

现在的问题就是，你可以在\_\_metaclass\_\_中放置些什么代码呢？答案就是：可以创建一个类的东西。那么什么可以用来创建一个类呢？type，或者任何使用到type或者子类化type的东东都可以。

**自定义元类**

元类的主要目的就是为了当创建类时能够自动地改变类。通常，你会为API做这样的事情，你希望可以创建符合当前上下文的类。假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定\_\_metaclass\_\_。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，\_\_metaclass\_\_实际上可以被任意调用，它并不需要是一个正式的类（我知道，某些名字里带有‘class’的东西并不需要是一个class，画画图理解下，这很有帮助）。所以，我们这里就先以一个简单的函数作为例子开始。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | # 元类会自动将你通常传给‘type’的参数作为自己的参数传入  def upper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):      '''返回一个类对象，将属性都转为大写形式'''      #  选择所有不以'\_\_'开头的属性      attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_')) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # 将它们转为大写形式      uppercase\_attr = dict((name.upper(), value) for name, value in attrs)        # 通过'type'来做类对象的创建      return type(future\_class\_name, future\_class\_parents, uppercase\_attr)    \_\_metaclass\_\_ = upper\_attr  #  这会作用到这个模块中的所有类    class Foo(object):      # 我们也可以只在这里定义\_\_metaclass\_\_，这样就只会作用于这个类中      bar = 'bip' |

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | print hasattr(Foo, 'bar')  # 输出: False  print hasattr(Foo, 'BAR')  # 输出:True    f = Foo()  print f.BAR  # 输出:'bip' |

现在让我们再做一次，这一次用一个真正的class来当做元类。

Python

### type 简介

从 [官方文档](https://docs.python.org/2/library/functions.html#type) 中，我们可以知道：

* 和 dict 类似，type 也是一个工厂构造函数，调用其将返回 一个type类型的实例（即 类）。
* type 有两个重载版本：
  + type(object)，即我们最常用的版本。
  + type(name, bases, dict)，一个更强大的版本。通过指定 类名称（name）、父类列表（bases）和 属性字典（dict） 动态合成一个类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # 请记住，'type'实际上是一个类，就像'str'和'int'一样  # 所以，你可以从type继承  class UpperAttrMetaClass(type):      # \_\_new\_\_ 是在\_\_init\_\_之前被调用的特殊方法      # \_\_new\_\_是用来创建对象并返回之的方法      # 而\_\_init\_\_只是用来将传入的参数初始化给对象      # 你很少用到\_\_new\_\_，除非你希望能够控制对象的创建      # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写\_\_new\_\_      # 如果你希望的话，你也可以在\_\_init\_\_中做些事情      # 还有一些高级的用法会涉及到改写\_\_call\_\_特殊方法，但是我们这里不用      def \_\_new\_\_(upperattr\_metaclass, future\_class\_name, future\_class\_parents, future\_class\_attr):          attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_'))          uppercase\_attr = dict((name.upper(), value) for name, value in attrs)          return type(future\_class\_name, future\_class\_parents, uppercase\_attr) |

但是，这种方式其实不是OOP。我们直接调用了type，而且我们没有改写父类的\_\_new\_\_方法。现在让我们这样去处理:

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class UpperAttrMetaclass(type):      def \_\_new\_\_(upperattr\_metaclass, future\_class\_name, future\_class\_parents, future\_class\_attr):          attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_'))          uppercase\_attr = dict((name.upper(), value) for name, value in attrs)            # 复用type.\_\_new\_\_方法          # 这就是基本的OOP编程，没什么魔法          return type.\_\_new\_\_(upperattr\_metaclass, future\_class\_name, future\_class\_parents, uppercase\_attr) |

你可能已经注意到了有个额外的参数upperattr\_metaclass，这并没有什么特别的。类方法的第一个参数总是表示当前的实例，就像在普通的类方法中的self参数一样。当然了，为了清晰起见，这里的名字我起的比较长。但是就像self一样，所有的参数都有它们的传统名称。因此，在真实的产品代码中一个元类应该是像这样的：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | class UpperAttrMetaclass(type):      def \_\_new\_\_(cls, name, bases, dct):          attrs = ((name, value) for name, value in dct.items() if not name.startswith('\_\_')          uppercase\_attr  = dict((name.upper(), value) for name, value in attrs)          return type.\_\_new\_\_(cls, name, bases, uppercase\_attr) |

如果使用super方法的话，我们还可以使它变得更清晰一些，这会缓解继承（是的，你可以拥有元类，从元类继承，从type继承）

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | class UpperAttrMetaclass(type):      def \_\_new\_\_(cls, name, bases, dct):          attrs = ((name, value) for name, value in dct.items() if not name.startswith('\_\_'))          uppercase\_attr = dict((name.upper(), value) for name, value in attrs)          return super(UpperAttrMetaclass, cls).\_\_new\_\_(cls, name, bases, uppercase\_attr) |

就是这样，除此之外，关于元类真的没有别的可说的了。使用到元类的代码比较复杂，这背后的原因倒并不是因为元类本身，而是因为你通常会使用元类去做一些晦涩的事情，依赖于自省，控制继承等等。确实，用元类来搞些“黑暗魔法”是特别有用的，因而会搞出些复杂的东西来。但就元类本身而言，它们其实是很简单的：

1)   拦截类的创建

2)   修改类

3)   返回修改之后的类

**为什么要用metaclass类而不是函数?**

由于\_\_metaclass\_\_可以接受任何可调用的对象，那为何还要使用类呢，因为很显然使用类会更加复杂啊？这里有好几个原因：

1）  意图会更加清晰。当你读到UpperAttrMetaclass(type)时，你知道接下来要发生什么。

2） 你可以使用OOP编程。元类可以从元类中继承而来，改写父类的方法。元类甚至还可以使用元类。

3）  你可以把代码组织的更好。当你使用元类的时候肯定不会是像我上面举的这种简单场景，通常都是针对比较复杂的问题。将多个方法归总到一个类中会很有帮助，也会使得代码更容易阅读。

4） 你可以使用\_\_new\_\_, \_\_init\_\_以及\_\_call\_\_这样的特殊方法。它们能帮你处理不同的任务。就算通常你可以把所有的东西都在\_\_new\_\_里处理掉，有些人还是觉得用\_\_init\_\_更舒服些。

5） 哇哦，这东西的名字是metaclass，肯定非善类，我要小心！

**究竟为什么要使用元类？**

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？好吧，一般来说，你根本就用不上它：

“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。”  —— Python界的领袖 Tim Peters

元类的主要用途是创建API。一个典型的例子是Django ORM。它允许你像这样定义：

Python

|  |  |
| --- | --- |
| 1  2  3 | class Person(models.Model):      name = models.CharField(max\_length=30)      age = models.IntegerField() |

但是如果你像这样做的话：

Python



|  |  |
| --- | --- |
| 1  2 | guy  = Person(name='bob', age='35')  print guy.age |

这并不会返回一个IntegerField对象，而是会返回一个int，甚至可以直接从数据库中取出数据。这是有可能的，因为models.Model定义了\_\_metaclass\_\_， 并且使用了一些魔法能够将你刚刚定义的简单的Person类转变成对数据库的一个复杂hook。Django框架将这些看起来很复杂的东西通过暴露出一个简单的使用元类的API将其化简，通过这个API重新创建代码，在背后完成真正的工作。

**结语**

首先，你知道了类其实是能够创建出类实例的对象。好吧，事实上，类本身也是实例，当然，它们是元类的实例。

Python

|  |  |
| --- | --- |
| 1  2  3 | >>>class Foo(object): pass  >>> id(Foo)  142630324 |

Python中的一切都是对象，它们要么是类的实例，要么是元类的实例，除了type。type实际上是它自己的元类，在纯Python环境中这可不是你能够做到的，这是通过在实现层面耍一些小手段做到的。其次，元类是很复杂的。对于非常简单的类，你可能不希望通过使用元类来对类做修改。你可以通过其他两种技术来修改类：

1） [Monkey patching](http://en.wikipedia.org/wiki/Monkey_patch)

2)   class decorators

当你需要动态修改类时，99%的时间里你最好使用上面这两种技术。当然了，其实在99%的时间里你根本就不需要动态修改类 :D