Metaclasses

Mikael Arakelian

November 29, 2018

What people say about Metaclasses

"Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you dont (the people who actually need them know with certainty that they need them, and dont need an explanation about why)."

Tim Peters

What people say about Metaclasses

> *"Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you dont (the people who actually need them know with certainty that they need them, and dont need an explanation about why)."*
>
> Tim Peters

Today we are going to look into the magic behind metaclasses and see when one would want to use them.

What people say about Metaclasses

> *"Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you dont (the people who actually need them know with certainty that they need them, and dont need an explanation about why)."*
>
> Tim Peters

Today we are going to look into the magic behind metaclasses and see when one would want to use them.
Before we begin talking about metaclasses, let's dive into classes.

Classes

Classes are objects

In most languages (C++, Java, C#, etc), classes are just pieces of code that describe how to produce an object. In Python classes are more than that.

Classes are objects

In most languages (C++, Java, C#, etc), classes are just pieces of code that describe how to produce an object. In Python classes are more than that.They are objects.

Classes are objects

In most languages (C++, Java, C#, etc), classes are just pieces of code that describe how to produce an object. In Python classes are more than that.They are objects.

```
>>> class Spam:
...     pass
>>> isinstance(Spam, object)
True
>>> Spam.a = 10
>>> Spam.a
10
>>> id(Spam)
140302571738696
```

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

Since classes are objects, then there must be something that creates them. How do we find out what has created the object?

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

Since classes are objects, then there must be something that creates them. How do we find out what has created the object? We use the `type` function!

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

Since classes are objects, then there must be something that creates them. How do we find out what has created the object? We use the `type` function!

```
>>> type(c)
<class 'Color'>
```

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

Since classes are objects, then there must be something that creates them. How do we find out what has created the object? We use the `type` function!

```
>>> type(c)
<class 'Color'>
```

Now, what is the type of the `Color` class?

Classes are created in runtime

Example from the standard library: `collections.namedtuple`.

```
>>> from collections import namedtuple
>>> Color = namedtuple('Color', 'red green blue')
>>> c = Color(red=1, green=31, blue=57)
>>> c
Color(red=1, green=31, blue=57)
```

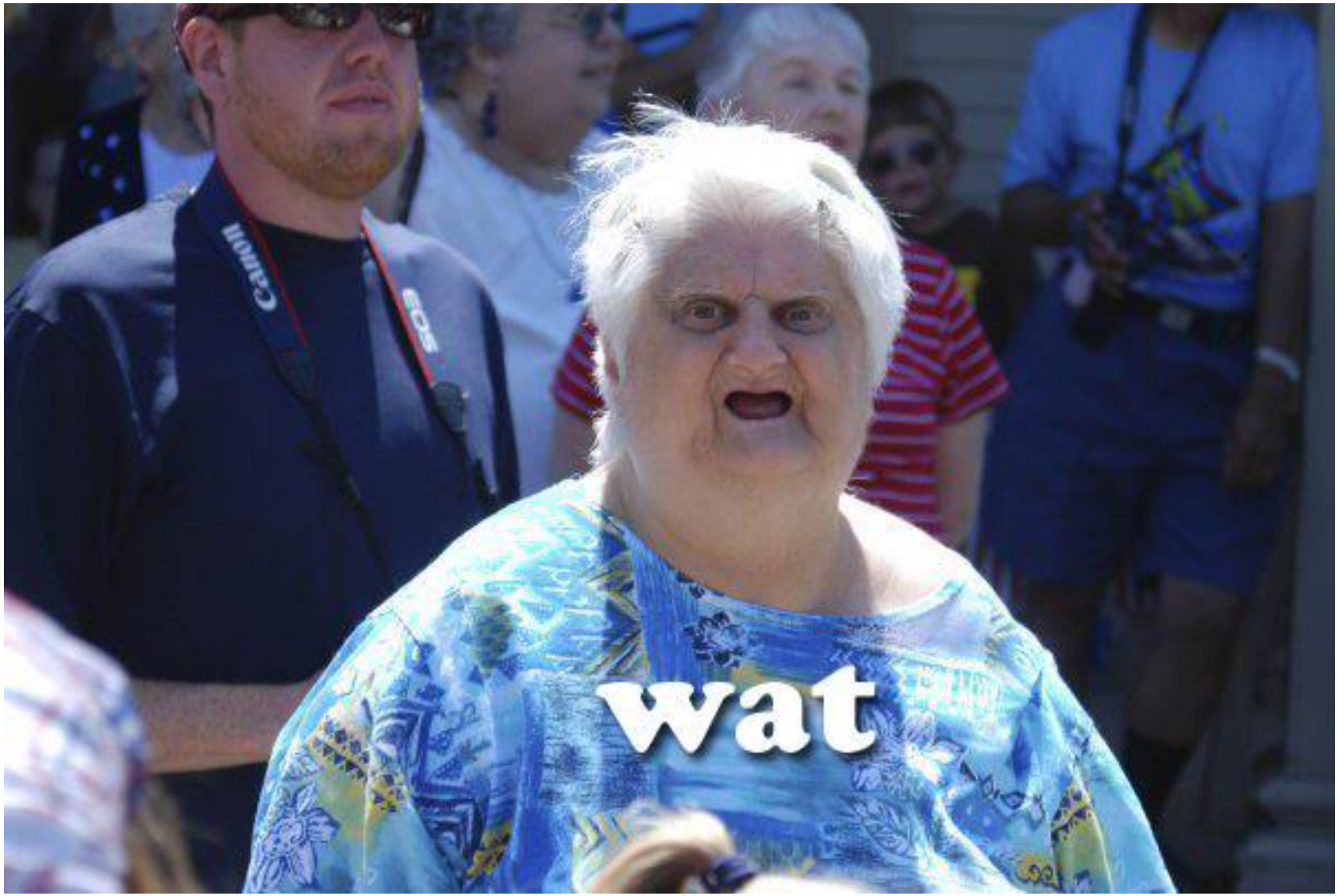
Since classes are objects, then there must be something that creates them. How do we find out what has created the object? We use the `type` function!

```
>>> type(c)
<class 'Color'>
```

Now, what is the type of the `Color` class?

```
>>> type(Color)
<class 'type'>
```

Classes are Created in Runtime



type

There are two ways to use `type` in Python

1. `type`(object) → get the object's type
2. `type`(name of the class,
   tuple of the parent classes,
   dictionary containing attributes names and values)
   → create a new type object

Thus, the class `Color` was created using the second version of `type`.

Class Creation Process

Before we dive into what Metaclasses are, let's further discuss what happens when we define a class

Class Creation Process

Before we dive into what Metaclasses are, let's further discuss what happens when we define a class

Consider a toy example

```
>>> class Spam(Base):
...     def __init__(self, ham):
...         self.ham = ham
...     def eggs(self):
...         return self.ham + ' and eggs'
```

Class Creation Process

Step 1: isolate the class body

```
>>> body = """
... def __init__(self, ham):
...     self.ham = ham
... def eggs(self):
...     return self.ham + ' and eggs'
... """
```

Class Creation Process

Step 1: isolate the class body

```
>>> body = """
... def __init__(self, ham):
...     self.ham = ham
... def eggs(self):
...     return self.ham + ' and eggs'
... """
```

Step 2: initiate the class dictionary

```
>>> clsdict = type.__prepare__('Spam', (Base,))
```

By default `clsdict` is an empty dict.

Class Creation Process

Step 3: execute the `body`

```
>>> exec(body, globals(), clsdict)
```

Class Creation Process

Step 3: execute the `body`

```
>>> exec(body, globals(), clsdict)
```

As a result, the `clsdict` is populated

```
>>> clsdict
{'__init__': <function __init__ at 0x10ebc8158>,
 'eggs': <function eggs at 0x10ebc81e0>}
```

Class Creation Process

Step 3: execute the `body`

```
>>> exec(body, globals(), clsdict)
```

As a result, the `clsdict` is populated

```
>>> clsdict
{'__init__': <function __init__ at 0x10ebc8158>,
 'eggs': <function eggs at 0x10ebc81e0>}
```

Step 4: Construct the class from its name, bases and the dictionary

```
>>> Spam = type('Spam', (Base,), clsdict)
>>> s = Spam('ham')
>>> s.eggs() == 'ham and eggs'
True
```

Metaclasses

What is a Metaclass?

What is a metaclass?

What is a Metaclass?

What is a metaclass? It is a subclass of `type` which usually overrides the __prepare__, __new__, __init__, __call__ methods

```
>>> class Meta(type):
...     pass
```

What is a Metaclass?

What is a metaclass? It is a subclass of `type` which usually overrides the __prepare__, __new__, __init__, __call__ methods

```
>>> class Meta(type):
...     pass
```

How do you use it?

What is a Metaclass?

What is a metaclass? It is a subclass of `type` which usually overrides the __prepare__, __new__, __init__, __call__ methods

```
>>> class Meta(type):
...     pass
```

How do you use it? Like this:

```
>>> class Spam(Base, metaclass=Meta):
...     def __init__(self, ham):
...         self.ham = ham
...     def eggs(self):
...         return 'bacon'
```

What is a Metaclass?

What is a metaclass? It is a subclass of `type` which usually overrides the __prepare__, __new__, __init__, __call__ methods

```
>>> class Meta(type):
...     pass
```

How do you use it? Like this:

```
>>> class Spam(Base, metaclass=Meta):
...     def __init__(self, ham):
...         self.ham = ham
...     def eggs(self):
...         return 'bacon'
```

What does it do?

What is a Metaclass?

What is a metaclass? It is a subclass of `type` which usually overrides the __prepare__, __new__, __init__, __call__ methods

```
>>> class Meta(type):
...     pass
```

How do you use it? Like this:

```
>>> class Spam(Base, metaclass=Meta):
...     def __init__(self, ham):
...         self.ham = ham
...     def eggs(self):
...         return 'bacon'
```

What does it do? It replaces every usage of type in the class creation process allowing it to modify the class at the time of definition (and more)

```
>>> clsdict = Meta.__prepare__('Spam', (Base,))
>>> Spam = Meta('Spam', (Base,), clsdict)
```

__prepare__

`__prepare__` is responsible for preparing the class namespace.

__prepare__

__prepare__ is responsible for preparing the class namespace.

```
>>> class Meta(type):
...     @classmethod
...     def __prepare__(metaclass, name, bases, **kwargs):
...         return dict()
```

__new__ and __init__

`__new__` and `__init__` are called respectively before and after the class object is created

__new__ and __init__

__new__ and __init__ are called respectively before and after the class object is created

```
class Meta(type):
    def __init__(cls, name, bases, clsdict, **kwargs):
        print('Meta.__init__', name, bases)
        return super().__init__(name, bases, clsdict)
    def __new__(cls, name, bases, clsdict, **kwargs):
        print('Meta.__new__', name, bases)
        return super().__new__(cls, name, bases, clsdict)
>>> class Spam(Base, metaclass=Meta):
...
Meta.__new__ Spam (<class '__main__.Base'>,)
Meta.__init__ Spam (<class '__main__.Base'>,)
```

__call__

`__call__` is called every time a new object is created

__call__

__call__ is called every time a new object is created

```
>>> class Meta(type):
...     def __call__(cls, *args, **kwargs):
...         print('Meta.__call__', cls.__name__, args, kwargs)
...         return super().__call__(*args, **kwargs)
>>> class Spam(Base, metaclass=Meta):
...
>>> a = Spam('ham')
Meta.__call__ Spam ('ham',) {}
Spam.__init__
```

Examples

Check Methods Are Implemented

https://github.com/michaelarakel/pyerevan-4-mettaclasses/blob/master/examples/check_methods_exist.py

Singleton

https://github.com/michaelarakel/pyerevan-4-mettaclasses/blob/master/examples/singleton.py

Object Caching

https://github.com/michaelarakel/pyerevan-4-mettaclasses/blob/master/examples/object_caching.py

Django Pseudo-Models

https://github.com/michaelarakel/pyerevan-4-mettaclasses/blob/master/examples/pseudo_django.py

Standard Library

Metaclasses are extensively used in the standard library. For inspiration see: `abc.ABCMeta`, `enum.Enum`

And Much More

Other examples include:

1. Tracking subclasses/registering handlers
2. Ensuring attributes of a class are ordered (relevant for Python ¡ 3.7)
3. Multiple-dispatch methods
4. Making all methods of a class thread-safe
5. etc

Key Takeaways

Key Takeaways

1. Classes are objects
2. Classes are created dynamically
3. Metaclasses hook into the class creation process and modify the definition and behaviour of the classes
4. Metaclasses propagate down the inheritance hierarchy

Thank You!