MetaClasses 101: Customizing Class Creation in 🐍

Ali Nawaz

```python
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

```python
try:
    Person.objects.get(first_name="Ali", last_name="Nawaz")
except Person.DoesNotExist:
    ...
```

About Me

- Studied Mathematics and Computer Science at LUMS
- Software Engineer at Arbisoft
- OpenEdx Project - Django + React
- Interested in Nerdy things
- Compilers, Networking
- Recent interest in Cryptographic algorithms

Classes

```
class Superhero:
    msg = "I am cool"
    def whoami(self):
        return "I am superman"
```

Classes

```python
class Superhero:
    msg = "I am cool"
    def whoami(self):
        return "I am superman"
```

```
>>> s = Superhero()
>>> type(s)
<class '__main__.Superhero'>
>>> isinstance(s, Superhero)
True
>>> type(Superhero)
<class 'type'>
>>> isinstance(Superhero, type)
True
```

Dynamically Creating Classes

```
class Language:
    def whoami(self):
        return "I am Python"
```

Dynamically Creating Classes

```python
class Language:
    def whoami(self):
        return "I am Python"
```

```python
def whoami(self):
    return "I am Python"
Language = type('Language', (), {'whoami': whoami})
```

Dynamically Creating Classes

```python
class Language:
    def whoami(self):
        return "I am Python"
```

```python
def whoami(self):
    return "I am Python"
Language = type('Language', (), {'whoami': whoami})
```

```
>>> l = Language()
>>> l
<__main__.Language object at 0x10debf9d0>
>>> l.whoami()
'I am Python'
```

The Type Callable

- **type (obj)**

  return the type of obj

- **type (name, bases, attrs)**

  create a type object
  name is the class name
  bases is a tuple of base classes
  attrs is a dict of attributes and methods defined in the class body

MetaClass

- Since a class is an instance, it has its own klass
- The klass of a class is a metaclass
- type is the most common metaclass

```
class A:
  pass
>>> A.__class__
<class 'type'>
```

Custom Metaclasses

```
class Meta(type):
    pass
class B(metaclass=Meta):
    pass
>>> B.__class__
<class '__main__.Meta'>
```

Meta(name, bases, dict)

B := Meta('B', (), {})

So far so good, welcome __new__

```
class Meta(type):
    def __new__(mcls, name, bases, attrs):
        print(mcls, name, bases, attrs)
        return super().__new__(mcls, name, bases, attrs)
class B(metaclass=Meta):
    pass
---------------------------
python3 script.py
<class '__main__.Meta'> B () {'__module__': '__main__', '__qualname__': 'B'}
```

More __new__ trickery

```
class AddMagic(type):
    def __new__(mcls, name, bases, attrs):
        attrs['magic'] = 'How did it get here?'
        return super().__new__(mcls, name, bases, attrs)
class B(metaclass=AddMagic):
    pass
---------------------------
python3 -i script.py
>>> B.magic
'How did it get here?'
```

Welcome __init__

```
class Meta(type):
    def __init__(cls, name, bases, attrs):
        print(cls, name, bases, attrs)
        return super().__init__(name, bases, attrs)
class B(metaclass=Meta):
    pass
---------------------------
python3 -i script.py
<class '__main__.B'> B () {'__module__': '__main__', '__qualname__': 'B'}
```

Welcome __prepare__

```
class Meta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        print(cls, name, bases)
        return {}

class B(metaclass=Meta):
    pass
---------------------------
python3 -i script.py
<class '__main__.Meta'> B ()
```

More __prepare__ magic

```
class NoDockerDict(dict):
    def __setitem__(self, key, value):
        if 'docker' not in key.lower():
            super().__setitem__(key, value)

class NoDockerMetaClass(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return NoDockerDict()
```

More __prepare__ magic

```python
class NoDockerDict(dict):
    def __setitem__(self, key, value):
        if 'docker' not in key.lower():
            super().__setitem__(key, value)

class NoDockerMetaClass(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return NoDockerDict()
```

```python
class Container(metaclass=NoDockerMetaClass):
    def is_docker(self):
        return False
    def is_lxc(self):
        return True
```

More __prepare__ magic

```
class NoDockerDict(dict):
    def __setitem__(self, key, value):
        if 'docker' not in key.lower():
            super().__setitem__(key, value)

class NoDockerMetaClass(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return NoDockerDict()
```

```
class Container(metaclass=NoDockerMetaClass):
    def is_docker(self):
        return False
    def is_lxc(self):
        return True
```

```
>>> c = Container()
>>> c.is_docker()
Traceback ...
```

Callables

- When you do x(...)
- Python essentially does type(x).__call__(x,...)

```
def pow2(x):
    return 2**x
----------------
>>> type(pow2).__call__(pow2, 3)
8
```

Welcome __call__

- Called when the class is called to create an object

```
class Meta(type):
    def __call__(cls, *args, **kwargs):
        print(cls, args, kwargs)
        return super().__call__(*args, **kwargs)

class B(metaclass=Meta):
    def __init__(self, val) -> None:
        self.val = val
```

Welcome __call__

- Called when the class is called to create an object

```
class Meta(type):
    def __call__(cls, *args, **kwargs):
        print(cls, args, kwargs)
        return super().__call__(*args, **kwargs)

class B(metaclass=Meta):
    def __init__(self, val) -> None:
        self.val = val
```

```
>>> b = B(23)
<class '__main__.B'> (23,) {}
```

Metaclass inheritance

- Given a class C, its metaclass is the most derived class among the metaclasses of its bases.

```
class Meta(type):
  pass

class Parent(metaclass=Meta):
  pass

class Child(Parent):
  pass
```

Metaclass inheritance

- Given a class C, its metaclass is the most derived class among the metaclasses of its bases.

```
class Meta(type):
  pass

class Parent(metaclass=Meta):
  pass

class Child(Parent):
  pass
```

```
>>> type(Child)
<class '__main__.Meta'>
```

Fitting the dots

- While processing a class definition, the interpreter essentially does the following
  1. Determine the metacls(mcls)
  2. Call mcls.__prepare__ to prepare the local namespace of the class
  3. Process the body of the class
  4. mcls(name, bases, attrs)
  5. type(mcls).__call__(mcls, name, bases, attrs)
  6. type.__call__(mcls, name, bases, attrs)

     > mcls.__new__(mcls, name, bases, attrs)

     > mcls.__init__(cls, name, bases, attrs)

Examples: Django

```
class ModelBase(type):
    def __new__(cls, name, bases, attrs, **kwargs):
        ...
        new_class = super_new(cls, name, bases, new_attrs, **kwargs)
        ...
        new_class.add_to_class(
                "DoesNotExist",
                subclass_exception(
                    ...
                ),
            )
        ...
        new_class.add_to_class("objects", manager)

class Model(ModelBase):
    ...
```

Examples: Enum (from Fluent Python)

```
class Flavor(AutoConst):
    banana
    coconut
    vanilla
--------------
>>> Flavor.coconut
1
```

Examples: Enum (from Fluent Python)

```
class Flavor(AutoConst):
    banana
    coconut
    vanilla
--------------
>>> Flavor.coconut
1
```

```
class AutoConstMeta(type):
    def __prepare__(name, bases, **kwargs):
        return WilyDict()
class AutoConst(metaclass=AutoConstMeta):
    pass
```

Examples: Dataclasses

```python
@dataclass
class Name:
    first: str
    last: str
```

Examples: Dataclasses

```
@dataclass
class Name:
    first: str
    last: str
```

```
>>> me = Name('ali', 'nawaz')
>>> me.first
'ali'
```

Examples: Dataclasses

```python
@dataclass
class Name:
    first: str
    last: str
```

```python
class dataclass(type):
    def __new__(cls, name, bases, dict):
        annots = cls.parse_annotations(bases, dict)
        init = cls.create_init(annots)
        repr = cls.create_repr(annots)
        return super().__new__(cls, name, bases, {
            **dict, '__init__': init, '__repr__': repr,
        })
```

```python
>>> me = Name('ali', 'nawaz')
>>> me.first
'ali'
```

Examples: Dataclasses

```python
@dataclass
class Name:
    first: str
    last: str
```

```python
class dataclass(type):
    def __new__(cls, name, bases, dict):
        annots = cls.parse_annotations(bases, dict)
        init = cls.create_init(annots)
        repr = cls.create_repr(annots)
        return super().__new__(cls, name, bases, {
            **dict, '__init__': init, '__repr__': repr,
        })
```

```python
>>> me = Name('ali', 'nawaz')
>>> me.first
'ali'
```

```python
class meta_dataclass(metaclass=dataclass):
    ...

class Name(meta_dataclass):
    first: str
    last: str
```

Wrapping up

- Useful for class customization
- Somewhat hard
- Many other ways to customize classes
- Decorators
- __init_subclass__
- __set_name__
- __class_getitem__
- Metaclasses still most powerful

References

- https://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example/
- https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/
- https://docs.python.org/3/reference/datamodel.html#customizing-class-creation
- Fluent Python (Luciano Ramalho)