Metaclasses All Versions

Examples

Basic Metaclasses

When `type` is called with three arguments it behaves as the (meta)class it is, and creates a new instance, ie. it produces a new class/type.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__              # <type 'type'>
Dummy().__class__.__class__  # <type 'type'>
```

It is possible to subclass `type` to create an custom metaclass.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Now, we have a new custom `mytype` metaclass which can be used to create classes in the same manner as `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

When we create a new class using the `class` keyword the metaclass is by default chosen based on upon the baseclasses.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

In the above example the only baseclass is `object` so our metaclass will be the type of `object` , which is `type` . It is possible override the default, however it depends on whether we use Python 2 or Python 3:

Python 2.x ≤2.7

A special class-level attribute `__metaclass__` can be used to specify the metaclass.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x ≥3.0

A special `metaclass` keyword argument specify the metaclass.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Any keyword arguments (except `metaclass`) in the class declaration will be passed to the metaclass. Thus `class MyDummy(metaclass=mytype, x=2)` will pass `x=2` as a keyword argument to the `mytype` constructor.

Read this in-depth description of python meta-classes for more details.

Singletons using metaclasses

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here .

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
```

```
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x ≤2.7

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x ≥3.0

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Introduction to Metaclasses

What is a metaclass?

In Python, everything is an object: integers, strings, lists, even functions and classes themselves are objects. And every object is an instance of a class.

To check the class of an object x, one can call `type(x)` , so:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Most classes in python are instances of `type` . `type` itself is also a class. Such classes whose instances are also classes are called metaclasses.

The Simplest Metaclass

OK, so there is already one metaclass in Python: `type` . Can we create another one?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

That does not add any functionality, but it is a new metaclass, see that MyClass is now an instance of SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

A Metaclass which does Something

A metaclass which does something usually overrides `type` 's `__new__` , to modify some properties of the class to be created, before calling the original `__new__` which creates the class:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

Improvements requested:

Using a metaclass

Metaclass syntax

Python 2.x ≤2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x ≥3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Python 2 and 3 compatibility with `six`

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Custom functionality with metaclasses

Functionality in metaclasses can be changed so that whenever a class is built, a string is printed to standard output, or an exception is thrown. This metaclass will print the name of the class being built.

```
class VerboseMetaclass(type):

    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

You can use the metaclass like so:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

The standard output will be:

```
Creating class Spam
[insert example string here]
```

The default metaclass

You may have heard that everything in Python is an object. It is true, and all objects have a class:

```
>>> type(1)
int
```

The literal 1 is an instance of `int`. Lets declare a class:

```
>>> class Foo(object):
...    pass
...
```

Now lets instantiate it:

```
>>> bar = Foo()
```

What is the class of `bar`?

```
>>> type(bar)
Foo
```

Nice, `bar` is an instance of `Foo` . But what is the class of `Foo` itself?

```
>>> type(Foo)
type
```

Ok, `Foo` itself is an instance of `type` . How about `type` itself?

```
>>> type(type)
type
```

So what is a metaclass? For now lets pretend it is just a fancy name for the class of a class. Takeaways:

- Everything is an object in Python, so everything has a class
- The class of a class is called a metaclass
- The default metaclass is `type` , and by far it is the most common metaclass

But why should you know about metaclasses? Well, Python itself is quite "hackable", and the concept of metaclass is important if you are doing advanced stuff like meta-programming or if you want to control how your classes are initialized.

Syntax

Parameters

Remarks

When designing your architecture, consider that many things which can be accomplished with metaclasses can also be accomplished using more simple semantics:

- Traditional inheritance is often more than enough.
- Class decorators can mix-in functionality into a classes on a ad-hoc approach.
- Python 3.6 introduces `__init_subclass__()` which allows a class to partake in the creation of its subclass.