Q1. What is the concept of a metaclass?

At first, the word Metaprogramming seems like a very funky and alien thing but if you have ever worked with decorators or metaclasses, you were doing metaprogramming there all along. In a nutshell, we can say metaprogramming is the code that manipulates code.

In Python, everything has some type associated with it. For example, if we have a variable having an integer value then its type is int. You can get the type of anything using the type() function.

Every type in Python is defined by Class. int, char, float in Python they are objects of int class or str class. So we can make a new type by creating a class of that type. For example, we can create a new type of Student by creating a Student class.

A Class is also an object, and just like any other object, it’s an instance of something called Metaclass. A special class type creates these Class objects. The type class is default metaclass which is responsible for making classes. In the above example, if we try to find out the type of Student class, it comes out to be a type.

Because Classes are also an object, they can be modified in the same way. We can add or subtract fields or methods in class in the same way we did with other objects. For example –

Code:

# Defined class without any

# class methods and variables

class test:pass

# Defining method variables

test.x = 45

# Defining class methods

test.foo = lambda self: print('Hello')

# creating object

myobj = test()

print(myobj.x)

myobj.foo()

This whole meta thing can be summarized as – Metaclass create Classes and Classes creates objects.

The metaclass is responsible for the generation of classes, so we can write our custom metaclasses to modify the way classes are generated by performing extra actions or injecting code. Usually, we do not need custom metaclasses but sometimes it’s necessary.

There are problems for which metaclass and non-metaclass-based solutions are available (which are often simpler) but in some cases, only metaclass can solve the problem. We will discuss such a problem in this article.

Creating custom Metaclass:

To create our custom metaclass, our custom metaclass has to inherit type metaclass and usually override –

* \_\_new\_\_(): It’s a method which is called before \_\_init\_\_(). It creates the object and returns it. We can override this method to control how the objects are created.
* \_\_init\_\_(): This method just initialize the created object passed as a parameter

We can create classes using the type() function directly. It can be called in following ways –

* When called with only one argument, it returns the type. We have seen it before in the above examples.
* When called with three parameters, it creates a class. Following arguments are passed to it –
  + Class name
  + Tuple having base classes inherited by class
  + Class Dictionary: It serves as a local namespace for the class, populated with class methods and variables

Code:

def test\_method(self):

print("This is Test class method!")

# creating a base class

class Base:

def myfun(self):

print("This is inherited method!")

# Creating Test class dynamically using

# type() method directly

Test = type('Test', (Base, ), dict(x="atul", my\_method=test\_method))

# Print type of Test

print("Type of Test class: ", type(Test))

# Creating instance of Test class

test\_obj = Test()

print("Type of test\_obj: ", type(test\_obj))

# calling inherited method

test\_obj.myfun()

# calling Test class method

test\_obj.my\_method()

# printing variable

print(test\_obj.x)

Q2. What is the best way to declare a class's metaclass?

In Python, we can customize the class creation process by passing the metaclass keyword in the class definition. This can also be done by inheriting a class that has already passed in this keyword. We can see below that the type of MyMeta class is type and that the type of MyClass and MySubClass is MyMeta .

Q3. How do class decorators overlap with metaclasses for handling classes?

Decorators are much, much simpler and more limited -- and therefore should be preferred whenever the desired effect can be achieved with either a metaclass or a class decorator.

Anything you can do with a class decorator, you can of course do with a custom metaclass (just apply the functionality of the "decorator function", i.e., the one that takes a class object and modifies it, in the course of the metaclass's \_\_new\_\_ or \_\_init\_\_ that make the class object!-).

There are many things you can do in a custom metaclass but not in a decorator (unless the decorator internally generates and applies a custom metaclass, of course -- but that's cheating;-)... and even then, in Python 3, there are things you can only do with a custom metaclass.

For example, suppose you want to make a class object X such that print X (or in Python 3 print(X) of course;-) displays peekaboo!. You cannot possibly do that without a custom metaclass, because the metaclass's override of \_\_str\_\_ is the crucial actor here, i.e., you need a def \_\_str\_\_(cls): return "peekaboo!" in the custom metaclass of class X.

The same applies to all magic methods, i.e., to all kinds of operations as applied to the class object itself (as opposed to, ones applied to its instances, which use magic methods as defined in the class -- operations on the class object itself use magic methods as defined in the metaclass).

One difference between decorators and inheritance is related to inheritance. Use of metaclass affects its children while the decorator affects only the current class.

The script use class-decorator to replace/overwirte the method 'func1'.

Code:

def deco4cls(cls):

cls.func1 = lambda self: 2

return cls

@deco4cls

class Cls1:

pass

class Cls1\_1(Cls1):

def func1(self):

return 3

obj1\_1 = Cls1\_1()

print(obj1\_1.func1()) # 3

The script use metaclass to replace/overwrite the method 'func1'.

Code:

class Deco4cls(type):

def \_\_init\_\_(cls, name, bases, attr\_dict):

# print(cls, name, bases, attr\_dict)

super().\_\_init\_\_(name, bases, attr\_dict)

cls.func1 = lambda self: 2

class Cls2(metaclass=Deco4cls):

pass

class Cls2\_1(Cls2):

def func1(self):

return 3

obj2\_1 = Cls2\_1()

print(obj2\_1.func1()) # 2!! the original Cls2\_1.func1 is replaced by metaclass

Q4. How do class decorators overlap with metaclasses for handling instances?

Class decorators is a tool for augmenting instance creation calls. Because they work by automatically rebinding a class name to the result of a function, though, there's no reason that we can't use them to augment the class before any instances are ever created. That is, class decorators can apply extra logic to classes, not just instances, at creation time:

Code:

def extras(Class): if required():

Class.extra = extra return Class

@extras class Client1: ... # Clientl = extras(Clientl)

@extras class Client2: ... # Rebinds class independent of instances

X = Client1() # Makes instance of augmented class

X.extra() # X is instance of original Clientl

Decorators essentially automate the prior example's manual name rebinding here. Just like with metaclasses, because the decorator returns the original class, instances are made from it, not from a wrapper object. In fact, instance creation is not intercepted at all.

In this specific case—adding methods to a class when it's created—the choice between metaclasses and decorators is somewhat arbitrary. Decorators can be used to manage both instances and classes, and they intersect with metaclasses in the second of these roles.

However, this really addresses only one operational mode of metaclasses. As we'll see, decorators correspond to metaclass\_\_init\_methods in this role, but metaclasses have additional customization hooks. As we'll also see, in addition to class initialization, metaclasses can perform arbitrary construction tasks that might be more difficult with decorators.

Moreover, although decorators can manage both instances and classes, the converse is not as direct—metaclasses are designed to manage classes, and applying them to managing instances is less straightforward.