Python Metaclasses & Class Decorators — Questions & Answers (Set 11)

Q1. What is the concept of a metaclass?

A metaclass is the class of a class—it controls how classes are created, configured, and finalized. When Python executes a class block, it builds a namespace dict, then calls the metaclass (default is type) to produce the class object. By customizing a metaclass, you can enforce conventions, auto-register classes, inject methods/attributes, validate definitions, or modify inheritance behavior at class creation time.

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

Use the metaclass= keyword in the class header (Python 3 style):

```
class MyMeta(type):
def __new__(mcls, name, bases, ns):
# customize class creation
return super().__new__(mcls, name, bases, ns)

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

This is explicit, modern, and avoids Python-2 compatibility hacks.

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

Both can transform or validate a class at creation time:

- Metaclass: runs before the class object exists—controls the construction pipeline (__prepare__, __new__, __init__ of the metaclass). It can influence MRO, descriptors, annotations processing, etc.
- Class decorator: runs after the class object is created—receives the class and returns either the same class (mutated) or a replacement.

Example:

```
def register(cls):
REGISTRY[cls.__name__] = cls
return cls

@register
class Service: ...
```

Rule of thumb:
- Need to control class creation deeply → use a metaclass.
- Need to wrap/augment an already-built class → use a class decorator.

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

- Metaclasses can affect instance behavior indirectly by:
* Injecting or rewriting methods/attributes on the class that instances use.

* Overriding __call__ on the metaclass to control instance construction (MyClass(...)), enabling factories, singletons, caching, validation.
- Class decorators can also modify instance behavior indirectly, but only by post-processing the class object (e.g., wrapping methods, injecting __init__ wrappers, adding properties). They cannot intercept the low-level instantiation call like a metaclass can.

Practical guidance:
- Need to intercept instantiation or enforce global invariants → metaclass.
- Need to tag/extend a class or wrap methods without changing the creation pipeline → class decorator.