Q1. What is the concept of a metaclass ?

A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.

Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way they implement them. Python is supporting them.

Some programmers see metaclasses in Python as "solutions waiting or looking for a problem".

There are numerous use cases for metaclasses. Just to name a few:

* logging and profiling
* interface checking
* registering classes at creation time
* automatically adding new methods
* automatic property creation
* proxies
* automatic resource locking/synchronization.

Principially, metaclasses are defined like any other Python class, but they are classes that inherit from "type". Another difference is, that a metaclass is called automatically, when the class statement using a metaclass ends. In other words: If no "metaclass" keyword is passed after the base classes (there may be no base classes either) of the class header, type() (i.e. \_\_call\_\_ of type) will be called. If a metaclass keyword is used, on the other hand, the class assigned to it will be called instead of type.

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

In Python 1.5, the requirement to write a C extension in order to write metaclasses has been dropped (though you can still do it, of course). In addition to the check ``is the type of the base class callable,'' there's a check ``does the base class have a \_\_class\_\_ attribute.'' If so, it is assumed that the \_\_class\_\_ attribute refers to a class.

It is easy to be confused by what exactly happens when using a metaclass, because we lose the absolute distinction between classes and instances: a class is an instance of a metaclass (a ``metainstance''), but technically (i.e. in the eyes of the python runtime system), the metaclass is just a class, and the metainstance is just an instance. At the end of the class statement, the metaclass whose metainstance is used as a base class is instantiated, yielding a second metainstance (of the same metaclass). This metainstance is then used as a (normal, non-meta) class; instantiation of the class means calling the metainstance, and this will return a real instance. And what class is that an instance of? Conceptually, it is of course an instance of our metainstance; but in most cases the Python runtime system will see it as an instance of a a helper class used by the metaclass to implement its (non-meta) instances...

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

While introducing people to Python metaclasses I realized that sometimes the big problem of the most powerful Python features is that programmers do not perceive how they may simplify their usual tasks. Therefore, features like metaclasses are considered a fancy but rather unuseful addition to a standard OOP language, instead of a real game changer.

This post wants to show how to use metaclasses and decorators to create a powerful class that can be inherited and customized by easily adding decorated methods.

## Metaclasses and decorators: a match made in space[¶](https://www.thedigitalcatonline.com/blog/2014/10/14/decorators-and-metaclasses/#metaclasses-and-decorators-a-match-made-in-space)

Metaclasses are a complex topic, and most of the times even advanced programmers do not see a wide range of practical uses for them. Chances are that this is the part of Python (or other languages that support metaclasses, like Smalltalk and Ruby) that fits the least the "standard" object-oriented patterns or solutions found in C++ and Java, just to mention two big players.

Indeed metaclasess usually come in play when programming advanced libraries or frameworks, where a lot of automation must be provided. For example, Django Forms system heavily relies on metaclasses to provide all its magic.

We also have to note, however, that we usually call "magic" or "tricks" all those techniques we are not familiar with, and as a result in Python many things are called this way, being its implementation often peculiar compared to other languages.

Time to bring some spice into your programming: let's practice some Python wizardry and exploit the power of the language!

In this post I want to show you an interesting joint use of decorators and metaclasses. I will show you how to use decorators to mark methods so that they can be automatically used by the class when performing a given operation.

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

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, not after the fact... but that's a pretty advanced sub-niche of your question, so let me give simpler examples).

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).