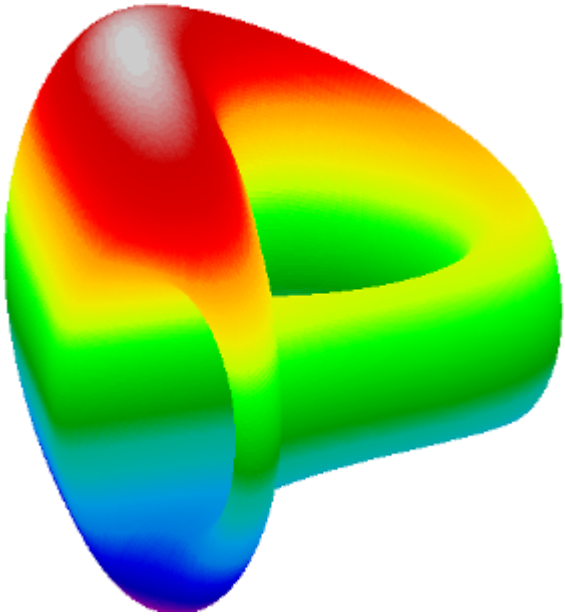
METACLASSES



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 the 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.

DEFINING METACLASSES

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.

Now we create a very simple metaclass. It's good for nothing, except that it will print the content of its arguments in the __new__ method and returns the results of the *type.__new__* call:

```
class LittleMeta(type):
    def __new__(cls, clsname, superclasses, attributedict):
        print("clsname: ", clsname)
        print("superclasses: ", superclasses)
        print("attributedict: ", attributedict)
        return type.__new__(cls, clsname, superclasses,
attributedict)
```

We will use the metaclass "LittleMeta" in the following example:

```
class S:
    pass
class A(S, metaclass=LittleMeta):
    pass
a = A()

clsname:  A
superclasses:  (<class '__main__.S'>,)
attributedict:  {'__module__': '__main__', '__qualname__': 'A'}
```

We can see LittleMeta.__new__ has been called and not type.__new__.

Resuming our thread from the last chapter: We define a metaclass "EssentialAnswers" which is capable of automatically including our augment_answer method:

```
x = input("Do you need the answer? (y/n): ")
if x.lower() == "y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

class EssentialAnswers(type):

    def __init__(cls, clsname, superclasses, attributedict):
        if required:
            cls.the_answer = the_answer


class Philosopher1(metaclass=EssentialAnswers):
    pass
class Philosopher2(metaclass=EssentialAnswers):
    pass
class Philosopher3(metaclass=EssentialAnswers):
    pass


plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())

Do you need the answer? (y/n): y
42
```

```
42
```

We have learned in our chapter "Type and Class Relationship" that after the class definition has been processed, Python calls

```
type(classname, superclasses, attributes_dict)
```

This is not the case, if a metaclass has been declared in the header. That is what we have done in our previous example. Our classes Philosopher1, Philosopher2 and Philosopher3 have been hooked to the metaclass EssentialAnswers. That's why EssentialAnswer will be called instead of type:

```
EssentialAnswer(classname, superclasses, attributes_dict)
```

To be precise, the arguments of the calls will be set the the following values:

```
EssentialAnswer('Philopsopher1',
                (),
                {'__module__': '__main__', '__qualname__':
'Philosopher1'})
```

The other philosopher classes are treated in an analogue way.

CREATING SINGLETONS USING METACLASSES

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. It is used in cases where exactly one object is needed. The conceptcan be generalized to restrict the instantiation to a certain or fixed number of objects. The term stems from mathematics, where a singleton, - also called a unit set -, is used for sets with exactly one element.

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton,
cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class SingletonClass(metaclass=Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
x = RegularClass()
```

```
y = RegularClass()
print(x == y)

True
False
```

CREATING SINGLETONS USING METACLASSES

Alternatively, we can create Singleton classes by inheriting from a Singleton class, which can be defined like this:

```
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class SingletonClass(Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
x = RegularClass()
y = RegularClass()
print(x == y)

True
False
```