Chapter 4

Meta Classes

Python classes are also objects, with the particularity that these can create other objects (their instances). Since classes are objects, we can assign them to variables, copy them, add attributes, pass them as parameters to a function, etc.

```
class ObjectCreator:
    pass

print(ObjectCreator)


def visualize(o):
    print(o)

visualize(ObjectCreator)
```

```
<class 'ObjectCreator'>
<class 'ObjectCreator'>
```

```
# Here we check if ObjectCreator has the attribute weight
print(hasattr(ObjectCreator, 'weight'))
```

```
False
```

```
# Here we are directly adding the weight attribute
ObjectCreator.weight = 80
print(hasattr(ObjectCreator, 'weight'))
print(ObjectCreator.weight)
```

```
True
80
```

```
# Assigning the class to a new variable
# Note that both variables reference the same object
ObjectCreatorMirror = ObjectCreator
print(id(ObjectCreatorMirror))
print(id(ObjectCreator))
print(ObjectCreatorMirror.weight)
```

```
140595089871608
140595089871608
80
```

Note that any changes we make to a class affect all of the class objects, including those that were already instantiated:

```
class Example:
    pass

x = Example()
print(hasattr(x, 'attr'))
Example.attr = 33
y = Example()
print(y.attr)
Example.attr2 = 54
print(y.attr2)
Example.method = lambda self: "Calling Method..."
print(y.method())
print(hasattr(x, 'attr'))
```

```
False
33
54
Calling Method...
True
```

4.1 Creating classes dynamically

Since classes are objects, we can create them at runtime just like any other object. For example, you can create a class within a function using the `class` statement:

```
def create_class(name):
    if name == 'MyClass':
        class MyClass:  # Usual way of creating a class
            pass
        return MyClass
    else:
        class OtherClass:
            pass
        return OtherClass

c1 = create_class('MyClass')
print(c1())
```

```
<MyClass object at 0x1078ff710>
```

We could also create a class in runtime using Python's `exec` command, which runs the code written in the input string. (You should be extremely careful with this function, and never execute a user given code, as it may contain malicious instructions).

```
name = "MyClass"
my_class = """
class %s():
    def __init__(self, a):
        self.at = a
""" % (name)
exec(my_class)
e = MyClass(8)
print(e.at)
```

```
8
```

That's pretty much, more of the same we have done so far. Now let's do it dynamically. First, let's remember that the `type` function returns an object's type:

```
print(type(1))
print(type("1"))
print(type(c1))
print(type(c1()))
# type is also an object of type 'type', it is an instance of itself
print(type(type))
```

```
<class 'int'>
<class 'str'>
<class 'type'>
<class 'MyClass'>
<class 'type'>
```

`type` can also create objects in runtime by taking a class descriptors as parameters. In other words, if we call `type` with only one argument, we are asking the type of the argument, but if you call it with three arguments, we are asking for the creation of a class. The first argument is the class name; the second argument is a tuple that contains all the parent classes. Finally, the third argument is a dictionary that contains all the class's attributes and methods: `{attr_name:attr_value}` or `{method_name:function}`. Below we show an example:

```
name = "MyClass"
c2 = type(name, (), {})
```

```
# We can do the same with a function
def create_class(name):
    c = type(name, (), {})
    return c

# Here we create the class MyClass2
create_class("MyClass2")()
```

Obviously we can also add attributes:

```
def create_class(name, attr_name, attr_value):
    return type(name, (), {attr_name: attr_value})

Body = create_class("Body", "weight", 100)
bd = Body()  # using it as a normal class to create instances.
print(bd.weight)
```

```
100
```

We can also add functions to the class dictionary, to create the methods of the class:

```
# a function that will be used as a method in the class we shall create
def lose_weight(self, x):
    self.weight -= x

Body = type("Body", (), {"weight": 100, "lose_weight": lose_weight})
bd = Body()

print(bd.weight)
bd.lose_weight(10)
print(bd.weight)
```

```
100
90
```

To inherit from the `Body` class:

```
class MyBody(Body):
    pass
```

we should write:

```
MyBody = type("MyBody", (Body,), {})
print(MyBody)
print(MyBody.weight)
```

```
<class 'MyBody'>
100
```

If we want to add methods to `MyBody`:

```
def see_weight(self):
    print(self.weight)

MyBody = type("MyBody", (Body,), {"see_weight": see_weight})
print(hasattr(Body, "see_weight"))
print(hasattr(MyBody, "see_weight"))
```

```
7  print(getattr(MyBody, "see_weight"))
8  print(getattr(MyBody(), "see_weight"))
9
10 m1 = MyBody()
11 m1.see_weight()
```

```
False
True
<function see_weight at 0x1078e02f0>
<bound method MyBody.see_weight of <MyBody object at 0x1078ffc50>>
100
```

4.2 Metaclasses

Metaclasses are Python's class creators; they are the classes of classes. `type` is Python's metaclass by default. It is written in lower_case to maintain consistency with `str`, the class that creates string objects, and with `int`, the class that creates objects of integer type. `type` is simply the class that creates objects of type `class`.

In Python all objects are created from a class:

```
1  height = 180
2  print(height.__class__)
3  name = "Carl"
4  print(name.__class__)
5
6
7  def func(): pass
8  print(func.__class__)
9
10
11 class MyClass():
12     pass
13 print(MyClass.__class__)
```

```
<class 'int'>
<class 'str'>
<class 'function'>
```

```
<class 'type'>
```

We can also check what is the creator class of all the previous classes:

```
print(height.__class__.__class__)
print(name.__class__.__class__)
print(func.__class__.__class__)
print(MyClass.__class__.__class__)
```

```
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
```

"metaclass" keyword argument in base classes

We can add the `metaclass` keyword in the list of keyword arguments of a class. If we do it, Python uses that metaclass to create the class; otherwise, Python will use `type` to create it:

```
class MyBody(Body):
    pass


class MyOtherBody(Body, metaclass=type):
    pass
```

Python asks if the `metaclass` keyword is defined within `MyBody` class arguments. If the answer is "yes", like in `MyOtherBody`, a class with that name is created in memory using the value of `metaclass` as a creator. If the answer is "no", Python will use the same metaclass of the parent class to create the new class. In the case of `MyBody`, the metaclass used is Body's metaclass i.e: `type`. What can we put in `metaclass`?: Anything that can create a class. In Python, `type` or any object that inherits from it can create a class.

Personalized Metaclasses

Before we start explaining of to personalize a metaclass, we will take a look at the structure of regular Python classes we have been using so far:

```
class System:
```

```
    # users_dict = {}  we will do this automatically inside __new__

    # cls is the object that represents the class
    def __new__(cls, *args, **kwargs):
        cls.users_dict = {}
        cls.id_ = cls.generate_user_id()
        # object has to create the class (everything inherits from object)
        return super().__new__(cls)

    # recall that self is the object that represents the instance of the class
    def __init__(self, name):
        self.name = name

    def __call__(self, *args, **kwargs):
        return [System.users_dict[ar] for ar in args]

    @staticmethod
    def generate_user_id():
        count = 0
        while True:
            yield count
            count += 1

    def add_user(self, name):
        System.users_dict[name] = next(System.id_)


if __name__ == "__main__":
    e = System("Zoni")
    e.add_user("KP")
    e.add_user("CP")
    e.add_user("BS")
    print(e.users_dict)
    print(e("KP", "CP", "BS"))
    print(System.mro())  # prints the class and superclasses
```

```
{'KP': 0, 'CP': 1, 'BS': 2}
[0, 1, 2]
[<class '__main__.System'>, <class 'object'>]
```

The __new__ method is in charge of the construction of the class. `cls` corresponds to the object that represents the created class. Any modification we want to do in the class before its creation can be done inside the __new__ method. In the example above, we are creating a dictionary (`users_dict`) and an id (`id_`). Both of them will belong to the class (static), not to the instances of the class. Note that __new__ **has to return the created class**, in this case returning the result of the __new__ method of the superclass.

Inside __init__, **the class is already created**. Now the main goal is to initialize the instances of it, by modifying `self`, the object that represents the instance of the class. In the example above, the instance initialization just registers the variable `name` inside the instance (`self.name = name`).

Finally, the __call__ method is in charge of the action that will be performed every time an instance of the class is called with parenthesis (treated as a callable). In the example, when we execute `e("KP", "CP", "BS")`, we are executing `e.__call__` with the passed arguments.

Now we are ready to understand how to personalize a metaclass. Following the same structure of regular Python classes mentioned above, imagine that the class now is a metaclass, and the instance is a class. In other words, instead of `cls` we use `mcs` in the __new__ method and instead of `self` we use `cls` in the __init__ method. The __call__ method will be in charge of the action performed when an instance of the metaclass (i.e. the class) is called with parenthesis.

The primary purpose of metaclasses is to change a class automatically during its creation. To control the creation and initialization of a class, we can implement the __new__ and __init__ methods in the metaclass (overriding). We must implement __new__: when we want to control the creation of a new object (class); and __init__: when we want to control the object initialization (in this context a class) after its creation.

```
1 class MyMetaClass(type):
2
3     def __new__(meta, clsname, bases, clsdict):
4         print('-----------------------------------')
5         print("Creating Class: {} ".format(clsname))
6         print(meta)
7         print(bases)
8         # Suppose we want to have a mandatory attribute
```

```
9           clsdict.update(dict({'mandatory_attribute': 10}))
10          print(clsdict)
11          return super().__new__(meta, clsname, bases, clsdict)
12          # we are calling 'type' __new__ method after doing the desired
13          # modifications. Note hat this method is the one that would have
14          # been called had we not used this personalized metaclass
15
16
17  class MyClass(metaclass=MyMetaClass):
18
19      def func(self, params):
20          pass
21
22      my_param = 4
23
24  m1 = MyClass()
25  print(m1.mandatory_attribute)
```

```
-----------------------------------
Creating Class: MyClass
<class 'MyMetaClass'>
()
{'mandatory_attribute': 10, 'my_param': 4, '__qualname__': 'MyClass',
'func': <function MyClass.func at 0x1078e0620>, '__module__': 'builtins'}
10
```

Overwriting the __call__ method

The `__call__` method is executed each time the **already created** class is **called** to instantiate a new object. Here is an example of how the `__call__` method can be intercepted whenever an object is instantiated:

```
1  class MyMetaClass(type):
2
3      def __call__(cls, *args, **kwargs):
4          print("__call__ of  {}".format(str(cls)))
5          print("__call__ *args= {}".format(str(args)))
6          return super().__call__(*args, **kwargs)
```

```
7
8
9  class MyClass(metaclass=MyMetaClass):
10
11     def __init__(self, a, b):
12         print("MyClass object with a=%s, b=%s" % (a, b))
13
14 print('creating a new object...')
15 obj1 = MyClass(1, 2)
```

```
creating a new object...
__call__ of  <class 'MyClass'>
__call__ *args= (1, 2)
MyClass object with a=1, b=2
```

Overwriting the __init__ method

We can also override the `__init__` method to mimic the behavior of the previous example. The main difference is that `__init__` (just like `__new__`) is called upon when creating the class, however `__call__` is called when creating a new instance:

```
1  class MyMetaClass(type):
2
3      def __init__(cls, name, bases, dic):
4          print("__init__ of  {}".format(str(cls)))
5          super().__init__(name, bases, dic)
6
7
8  class MyClass(metaclass=MyMetaClass):
9
10     def __init__(self, a, b):
11         print("MyClass object with a=%s, b=%s" % (a, b))
12
13 print('creating a new object...')
14 obj1 = MyClass(1, 2)
```

```
__init__ of  <class 'MyClass'>
```

```
creating a new object...
MyClass object with a=1, b=2
```

4.3 Hands-On Activities

Activity 4

In the file called *AC04_0_provided_code.py* we have two implemented classes and a *main*. You have to create a metaclass called `MetaRobot` that must add the following data and methods to the `Robot` class:

- The `creator` (static) variable: It must be your user id
- The `start_ip` (static) variable: It is the IP address from where the robot is initialized. The address is "190.102.62.283"
- The `check_creator` method: This method verifies that the robot exists inside the list of programmers. I must print out a message indicating if the creator is inside the programmer's list or not.
- The `disconnect` method: By using this method, the robot can disconnect any hacker that is on the same port as the robot. In case the robot finds a hacker in the same port, it must print out a message telling the situation. Assume that the port's `hacker` attribute has to be changed to 0 to disconnect it.
- The `change_node` method: With this method, the robot can modify the node (port) to anyone that gets inside the network. It must print out a message indicating from what node it is coming from and what is its destination.

Consider that only the `Robot` class can be builded from the `MetaRobot` metaclass. In case any other class is attempted to be created from `MetaRobot` you should raise an error. It is forbiden to modify the *AC04_0_provided_code.py* file, everything has to be done through the `MetaRobot` metaclass.