Magic Methods

In Python

Art Credit: artstation.com/abelion

Introduction:

The magic methods in Python programming language is specifically for Object Oriented Design.

When we create an object, the Python interpreter calls special functions in the backend of code execution.
They are known as Magic Methods or **Dunder Methods.**

They provide some extra functionality in our class.

Why do we say Dunder?

Because their names lie between **double underscores.**
They perform some set of calculations that are just like magic while we create an object of a class.

Common Dunders Example:

```
__class__           __hash__        __init__
__init_subclass__   __le__          __lt__
__module__          __ne__          __new__
__reduce__          __reduce_ex__   __delattr__
__dict__            __dir__         __doc__
__eq__              __format__      __ge__
__getattribute__    __gt__          __add__
```

How do we check what are the dunder and how many of them are there in the standard class?

1. Create a sample class.
2. Create its object.
3. Use the **dir() function** and insert the object inside it. "print(dir(obj)"
4. This function prints a list of all the Magic Methods along with data members and member functions that are assigned to the class.

04

Implementation of some magic methods:

In this post, we are going to override/implement some of the common python magic methods.

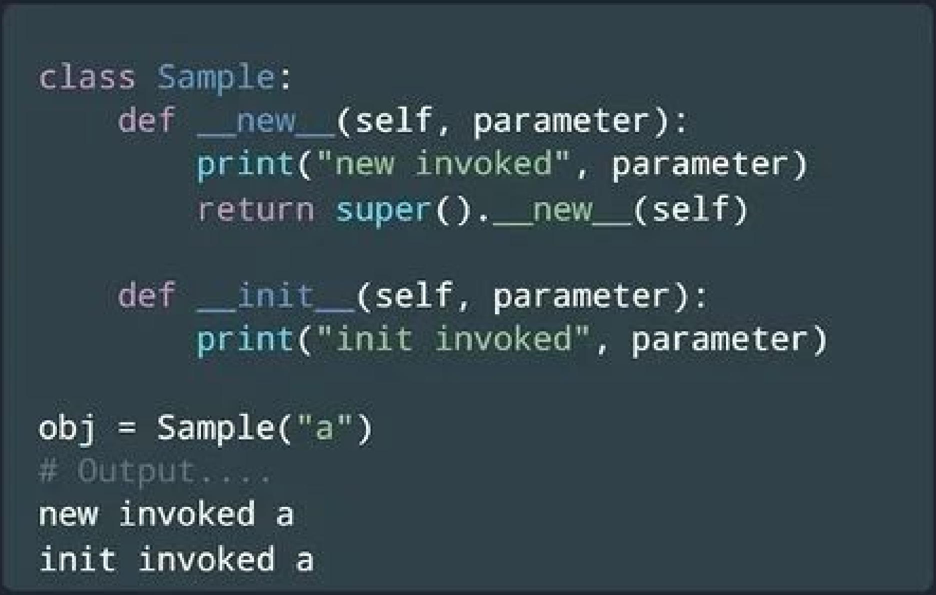
< Swipe

«««

01

__new__()

This method helps the constructor __init__() method to create objects for a class. So, when we create an instance of a class, the Python interpreter first calls the __new__() method and after that__init__() method.

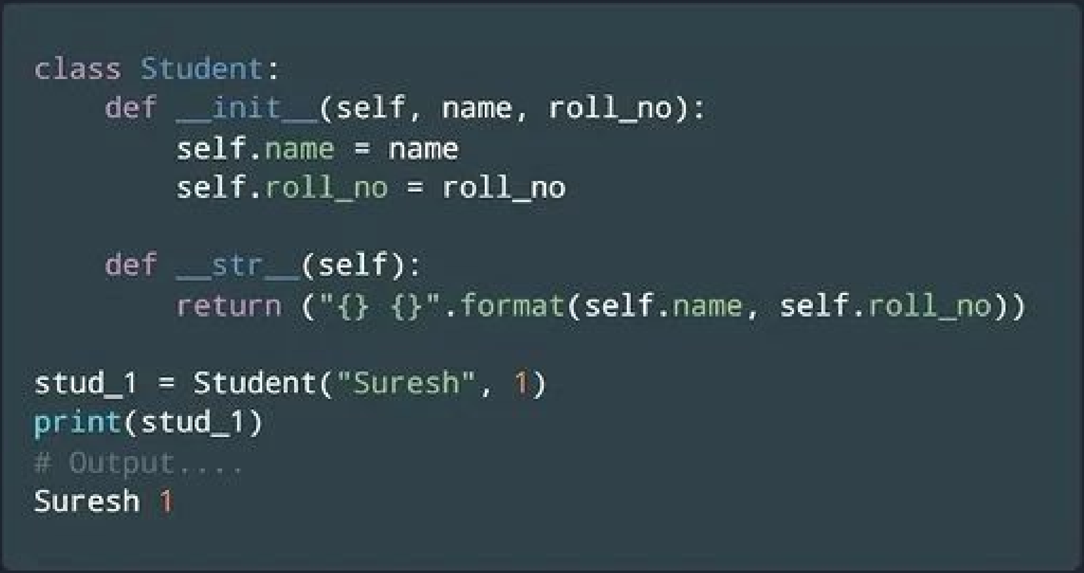


```
class Sample:
    def __new__(self, parameter):
        print("new invoked", parameter)
        return super().__new__(self)

    def __init__(self, parameter):
        print("init invoked", parameter)

obj = Sample("a")
# Output....
new invoked a
init invoked a
```

02

__str__()

This method helps us to display the object according to our requirements.
The print() function displays the memory location of the object but If we want to modify we can do this by using __str__() function.

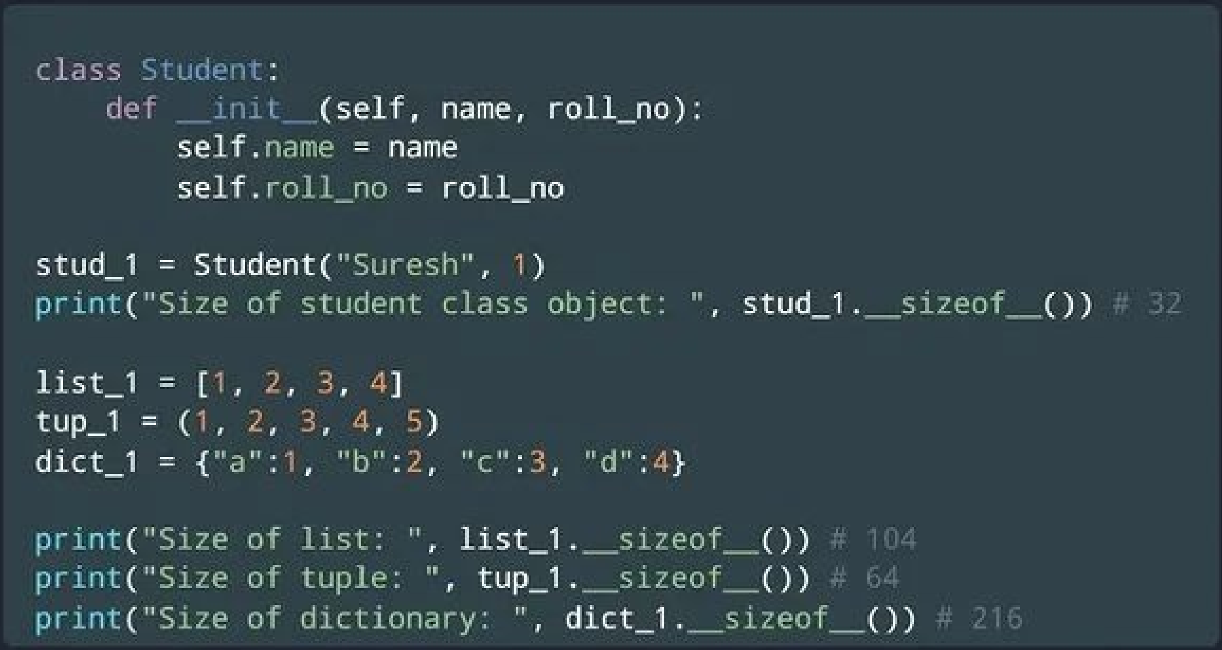


```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

    def __str__(self):
        return ("{} {}".format(self.name, self.roll_no))

stud_1 = Student("Suresh", 1)
print(stud_1)
# Output....
Suresh 1
```

03

__sizeof__()

If we want to know the memory allocated to that object, then we can call or override the __sizeof__() function and pass our object.



```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

stud_1 = Student("Suresh", 1)
print("Size of student class object: ", stud_1.__sizeof__()) # 32

list_1 = [1, 2, 3, 4]
tup_1 = (1, 2, 3, 4, 5)
dict_1 = {"a":1, "b":2, "c":3, "d":4}

print("Size of list: ", list_1.__sizeof__()) # 104
print("Size of tuple: ", tup_1.__sizeof__()) # 64
print("Size of dictionary: ", dict_1.__sizeof__()) # 216
```