The Magic Methods in Python

The magic methods in Python programming language are specifically for Object Oriented Design. Every class that we create has its own magic methods. Python’s standard interpreter assigns these to every class we create inside it. So, in this article, we shall see in detail how to call and use magic methods for a better programming approach. Let the coding fun begin!

Brushing up OOP knowledge

Before we get to the main topic, let us understand and polish the knowledge of OOP concepts. We shall see only the basics. So, Object Oriented Programming is a way of enclosing the data members and member functions into a user-defined entity called a **Class**.

Class is something that holds particular data items that relate to each other and communicate in a specific way. We access the properties and the member functions using **Object**. The object is an instance of a class. In any programming language, memory is never allocated when we create a class, but it is actually created when we create its instance I.e object.

**Example:**

**The animal** is a type of class. In that, we include all the living beings that reside on Earth. So, everyone has their own way of living, food, and shelter. The animal just defines the blueprint of all these. For example, **the cat is the object of the Animal class. It has four legs, eats mice, and lives in houses or bushes.** In the same way, the tiger has four legs but it kills and eats many animals so we say that the tiger eats meat, it lives in the forest.

**Code Example with Python:**

|  |
| --- |
| class Animal:      def \_\_init\_\_(self, legs, food, shelter):          self.legs = legs          self.food = food          self.shelter = shelter        def showAnimal(self):          print("The animal has {} legs: ".format(self.legs))          print("The animal eats: {}".format(self.food))          print("The animal lives in: {}".format(self.shelter))    cat = Animal(4, "Mouse", "House")  tiger = Animal(4, "Meat", "Forest")  cat.showAnimal()  tiger.showAnimal() |

**Output:**

|  |
| --- |
| The animal has 4 legs:  The animal eats: Mouse  The animal lives in: House  The animal has 4 legs:  The animal eats: Meat  The animal lives in: Forest |

**Explanation:**

1. The animal class contains the **number of legs, food, and shelter as properties.**
2. **When we create an instance and insert the values inside the constructor, the difference in their behaviour** is clear.
3. **So, objects of the same class can be different according to the behaviour of values.**

The magic methods in OOP

So, in the above example, we have a class as Animal. Python has a set of methods namely **Dunder methods** that are responsible for holding the properties, data members, and member functions of a class.

***Definition: 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.***

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. So, how do we check what are they and how many of them are there in the standard class? Use the steps below to find them:

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

**Code:**

|  |
| --- |
| print(dir(cat)) |

**Output:**

|  |
| --- |
| \_\_class\_\_  \_\_delattr\_\_  \_\_dict\_\_  \_\_dir\_\_  \_\_doc\_\_  \_\_eq\_\_  \_\_format\_\_  \_\_ge\_\_  \_\_getattribute\_\_  \_\_gt\_\_  \_\_hash\_\_  \_\_init\_\_  \_\_init\_subclass\_\_  \_\_le\_\_  \_\_lt\_\_  \_\_module\_\_  \_\_ne\_\_  \_\_new\_\_  \_\_reduce\_\_  \_\_reduce\_ex\_\_  \_\_repr\_\_  \_\_setattr\_\_  \_\_sizeof\_\_  \_\_str\_\_  \_\_subclasshook\_\_  \_\_weakref\_\_  food  legs  shelter  showAnimal |

**The names that you see in double underscores are all magic methods.**The rest of the properties are defined by the user. As we can see \_\_init\_\_() which is the constructor of any class in Python, is also a magic method. Let us see their use one by one. **To understand their functionality always try to override the functions.**

**One thing to note is that, for any class the user defines, there is some set of default magic methods with respect to each of them.**

Use and implementation of some magic methods

In this section, we shall see the use and implementation and use of some magic methods for writing a better OOP design.

1. \_\_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. They work hand in hand with each other.

1. When a programmer opts to create an object, \_\_new\_\_() gets invoked that accepts the name of the object.
2. Then \_\_init\_\_() is invoked where the parameters including **self** are inserted into the object which in turn helps us to modify the class properties.

**Code:**

|  |
| --- |
| 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 |

**Explanation:**

1. First, we create a class as Sample.
2. Then override the \_\_new\_\_() method by creating it. Then, as usual, the self parameter comes, and after that gives a simple parameter.
3. Return a super() function with the \_\_new\_\_() function with the self parameter to get access to the customizations we make to the method.
4. Then, with the same practice call the \_\_init\_\_() function with some parameter.
5. After that create an object of a sample class.
6. **Now, when we run the code, the interpreter first calls the \_\_new\_\_(), and then it calls \_\_init\_\_() method.**

2. \_\_init\_\_():

Python is an Object Oriented Programming language. So, the class must have a constructor. This requirement is fulfilled using the \_\_init\_\_() method. When we create a class and want to give some initial parameters to it. The initializer method performs this task for us.

**Code:**

|  |
| --- |
| class Sample:      def \_\_init\_\_(self, parameter):          print("init invoked", parameter)    obj = Sample("a") |

**Output:**

|  |
| --- |
| init invoked a |

**Explanation:**

1. Create/override the \_\_init\_\_() method. Insert self parameter to signal the interpreter that this is a class method.
2. Insert required parameter.
3. Then print that parameter using the print() function.
4. After that create an object.
5. When we run the code we get the output as “init invoked a”, this states that **the interpreter calls init() and prints that parameter.**

3. \_\_str\_\_():

This method helps us to display the object according to our requirements. So, let us say that when we create an object and try to print it. The print() function displays the memory location of the object. If we want to modify we can do this. **The \_\_str\_\_() function gives a nice representation of the object.**

**Code (before using \_\_str\_\_()):**

|  |
| --- |
| class Student:      def \_\_init\_\_(self, name, roll\_no):          self.name = name          self.roll\_no = roll\_no    stud\_1 = Student("Suresh", 1)  print(stud\_1) |

**Output:**

|  |
| --- |
| <\_\_main\_\_.Student object at 0x0000023E2CF37CA0> |

**Code (after using \_\_str\_\_()):**

|  |
| --- |
| 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 |

Cool right! now we can also use similar methods. We can format the object as according to our needs.

4. \_\_repr\_\_():

Similar to the \_\_str\_\_(), we can use \_\_repr\_\_ function for the decoration of objects. The code is similar to \_\_str\_\_() implementation.

|  |
| --- |
| class Student:      def \_\_init\_\_(self, name, roll\_no):          self.name = name          self.roll\_no = roll\_no        def \_\_repr\_\_(self):          print("repr invoked")          return ("{} {}".format(self.name, self.roll\_no))    stud\_1 = Student("Suresh", 1)  print(stud\_1) |

**Output:**

|  |
| --- |
| repr invoked  Suresh 1 |

5. \_\_sizeof\_\_():

When we create a class the interpreter never assigns memory to it. It assigns memory to the object. If we want to know the memory allocated to that object, then we can call or override the \_\_sizeof\_\_() function and pass our object. This also returns size of a list =, tuple, dictionary object.

**Code:**

|  |
| --- |
| 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\_\_())    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\_\_())  print("Size of tuple: ", tup\_1.\_\_sizeof\_\_())  print("Size of dictionary: ", dict\_1.\_\_sizeof\_\_()) |

**Output:**

|  |
| --- |
| Size of student class object:  32  Size of list object:  104  Size of tuple object:  64  Size of dictionary object:  216 |

6. \_\_add\_\_():

This magic method is specifically similar to its name. It adds two variables. For an integer it returns the sum, for a string, it returns their concatenation result.

**Code:**

|  |
| --- |
| class Numbers:      def \_\_init\_\_(self, a, b):          self.a = a          self.b = b        def \_\_add\_\_(self):          print("\_\_add\_\_ invoked")          return self.a + self.b    num = Numbers(3, 4)  num\_2 = Numbers("a", "b")  print(num.\_\_add\_\_())  print(num\_2.\_\_add\_\_()) |

**Output:**

|  |
| --- |
| \_\_add\_\_ invoked  7  \_\_add\_\_ invoked  ab |

7. \_\_reduce\_\_():

This magic method returns a set or a dictionary of all the parameters of a class and their values in **key: value** format. This can be directly called using the object name with the dot operator. So, when we create a class and instantiate it with some values. The function shall return it with the name of the parameters which were given during the declaration of a class.

**Code:**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary    emp = Employee("Shrinivas", 150000)  print(emp.\_\_reduce\_\_()) |

**Output:**

|  |
| --- |
| (<function \_reconstructor at 0x0000023E22892EE0>, (<class '\_\_main\_\_.Employee'>,  <class 'object'>, None), {'name': 'Shrinivas', 'sal': 150000}) |

**Code (after overriding \_\_reduce\_\_()):**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary        def \_\_reduce\_\_(self):          return self.name, self.sal    emp = Employee("Shrinivas", 150000)  print(emp.\_\_reduce\_\_()) |

**Output:**

|  |
| --- |
| {"Shrinivas", 150000} |

**Explanation:**

When we override and try to return the parameters, we only get their values in a set.

8. \_\_hash\_\_():

The \_\_hash\_\_() function returns a specific hash value of the object stored in the heap memory. We can either override it or call it using the object name. Hashing is very useful to fetch the memory address of any random element in a computer. All programming languages use hash for the sake of simplicity and for memory allocation.

**Code:**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary        def \_\_hash\_\_(self):          return super().\_\_hash\_\_()    emp = Employee("Shrinivas", 150000)  print(emp.\_\_hash\_\_()) |

**Output:**

|  |
| --- |
| 154129100057 |

**Code:**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary    emp = Employee("Shrinivas", 150000)  print(emp.\_\_hash\_\_()) |

**Output:**

|  |
| --- |
| 154129054082 |

9. \_\_getattribute\_\_(name):

This function returns the value of the attribute of a class if it exists. We need to call the function and pass the attribute which we assigned to the class parameter using the **self**keyword. Like if we assign the value of **salary**to **self.sal**we need to call **sal** inside the \_\_getattribute\_\_() function.

**Code:**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary        def \_\_getattribute\_\_(self, name):          return super().\_\_getattribute\_\_(name)    emp = Employee("Ravi", 500000)  print(emp.\_\_getattribute\_\_("sal")) |

**Output:**

|  |
| --- |
| 50000 |

**Explanation:**

In this function, the **“self.sal”**is assigned to the **salary**parameter of the Employee class. The function returns its value as the attribute that exists inside the class. If it does not exist the function returns an error message.

10. \_\_setattr\_\_(name, value):

As the name suggests, this magic method helps us change the value of an attribute when we define the object. No need to override the **\_\_getattribute\_\_() and \_\_setattr\_\_()** functions. Just call them using the objects created.

**Code:**

|  |
| --- |
| class Employee:      def \_\_init\_\_(self, name, salary):          self.name = name          self.sal = salary      emp = Employee("Ravi", 500000)  emp.\_\_setattr\_\_("name", "Suresh")  emp.\_\_setattr\_\_("sal":600000)  print("The changed name of the employee is: ", emp.\_\_getattribute\_\_("name"))  print("The changed salary of the employee is: ", emp.\_\_getattribute\_\_("sal")) |

**Output:**

|  |
| --- |
| The changed name of the employee is: Suresh  The changed salary of the employee is: 600000 |

**Explanation:**

1. **the \_\_setattr\_\_() take two parameters.**
   1. **name of attribute**
   2. **its new value**
2. **Then it assigns that particular value to that attribute.**
3. **After that, to check the value assigned to it call the \_\_getattrbute\_\_() function using the employee object and dot operator. emp.\_\_getattribute(“name”).**

**Point to note: These two functions replace getter and setter methods for a class in Python.**