**OOPS in Python — Part 2**

[[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----5af6a52e2a1a--------------------------------)](https://medium.com/@shahooda637?source=post_page-----5af6a52e2a1a--------------------------------)

[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----5af6a52e2a1a--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa0449bd47c69&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40shahooda637%2Foops-in-python-part-2-5af6a52e2a1a&user=Isha+Choudhary&userId=a0449bd47c69&source=post_page-a0449bd47c69----5af6a52e2a1a---------------------post_header-----------)

14 min read

·

Sep 21

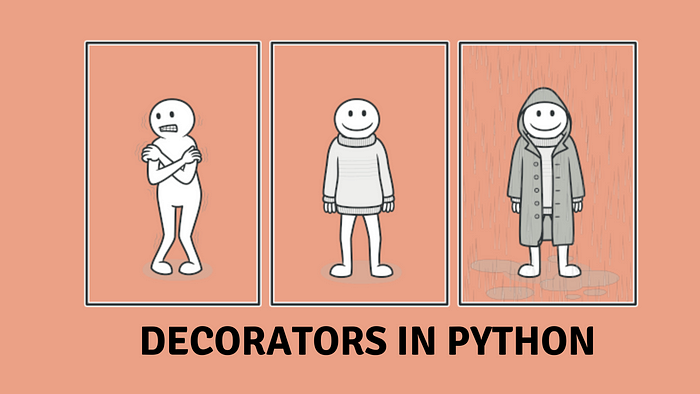
1

**(Object Oriented Programming)**



In this article, we will know more about OOPs in Python, a sequel to part 01 published earlier at “[*The Python OOPs Moments (Part — 01*](https://medium.com/@shahooda637/the-python-oops-moments-part-01-9e16bfd4cc09)*)”.*The four pillars of the object oriented programming was already discussed. This one talks about other parts of the object oriented programming such as different types of methods, decorators and more. Let’s dive in and learn about the OOPs in Python.

**Decorators**



When we are studying the object oriented programming in python, we have a lot more than just the four pillars of the oops i.e., abstraction, encapsulation, inheritance, and polymorphism as discussed in previous article.

One of the part of this article discusses one of those extra methods and function we learn under oops in python. This is Decorators.

In python, a decorator is a special type of function that is used to modify or extend the behavior of other functions or methods. Decorators are often applied or used in functions or methods to add additional functionality to them without modifying the source code directly.

**What is a Decorator in Python?**

A brief definition for decorators can be given as follows: A decorator function is a higher order function which takes another function as an argument and returns a new function. This new function usually extends or modifies the behavior of the original function. Decorator functions are typically prefixed with the special character “@” and always placed above the function being decorated.

Following sections of code shows some examples of the decorators and how they can be used to extend a function which changing the original source code.

# defining normal function  
  
def test():  
 print("The addition of two odd numbers 3 and 7")  
 print(3 + 7)  
 print("is always an even number")  
  
test()  
  
Output:  
The addition of two odd numbers 3 and 7  
10  
is always an even number

# defining a decorator function  
  
def sum\_decorator(func):  
 def inner\_deco():  
 print("The addition of two odd numbers 3 and 7")  
 func()  
 print("is always an even number")  
 return inner\_deco

# funtion without using decorator  
def odd\_add():  
 print(3+7)  
  
odd\_add()  
  
Output:  
10

# using a decorator to extend the functionality of a normal odd addition function  
  
@sum\_decorator  
def odd\_add():  
 print(3+7)  
  
odd\_add()  
  
Output:  
The addition of two odd numbers 3 and 7  
10  
is always an even number

#defining new decorator  
def deco(func):  
 def inner\_deco():  
 print("The addition odd numbers upto 10 is")  
 func()  
 return inner\_deco  
  
#using decorator on the function extending its functionality  
@deco  
def odd\_add():  
 sum = 0  
 for i in range(10):  
 if i%2 != 0:  
 sum += i  
 print(sum)  
  
#calling the decorated function  
odd\_add()  
  
Output:  
The addition odd numbers upto 10 is  
25

**Why do we need decorators in OOP? What are its benefits in coding?**

**01. Code Reusability:**

Decorators promote code reusability by allowing the coders to apply common functionality or behavior to multiple functions or methods. This practice is useful in writing code in OOP when we have multiple classes with similar behavior.

**02. Separation of Concerns:**

Decorators help us coders to separate different concerns or aspects of code. For example, we can have decorators for logging in, authentication, and more. This separation of having different decorator for each different task make the codebase cleaner and maintainable especially when working in a collaborative environment.

**03. Open or Closed Principle:**

Decorators follow the “Open/Close Principle” of object oriented programming, which states that classes or functions should be open for extension but closed for modification. In such situation, we can use decorators to extend the behavior of a function or method without changing its source code.

**04. Easy Maintenance:**

Decorators make it easier to manage and maintain code since through decorators we can always update the behavior of a function or method in one place rather than make changes throughout the codebase.

**05. Promoting Single Responsibility Principle:**

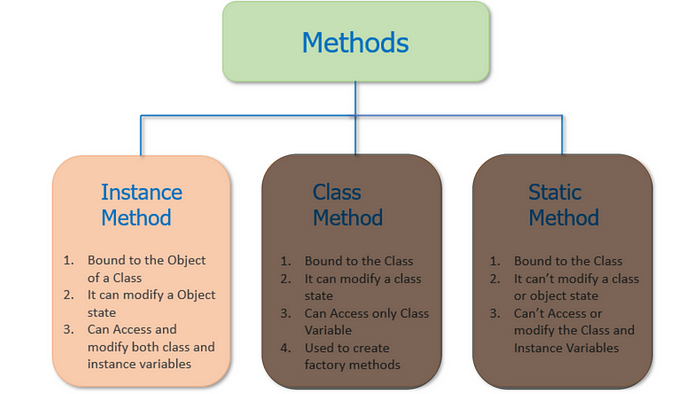
Decorators can help ensure that each function or method has a single responsibility by allowing us to modularize additional functionality to our functions or methods.

**06. Readability:**

Decorators can enhance code readability by keeping the core functionality of a function or method separate from its auxiliary features.

All in all, Decorators in python are a powerful tool for extending and modifying the behavior of functions and methods, which is particularly useful in object oriented programming while dealing with classes and objects.

**Instance Method**



Instance method in python is a type of method which as name suggests, operates on an instance of a class. These methods are defined inside a class and are intended to perform actions or operations related to the specific instance of the class.

Instance methods take the instance itself, through ‘self’ as their first parameter, which allows it to access and modify the instance’s attributes and behavior.

The syntax of the instance method is simple, and given in the cell below.

class student:  
  
 #instance method  
 def details(self, name, age, standard):  
 self.name = name  
 self.age = age  
 self.standard = standard

The instance method access the attributes of the class using the ‘self’ keyword, as done in the code above.

The code ‘self.name’ means that the instance method is accessing the ‘name’ attribute of the class.

**Modification**

Instance methods can modify the state of the instance easily by changing the values of its attributes. The instance method can also call other instance methods or perform any other actions relevant to the instance.

**Behavior**

Instance methods encapsulate the behavior of the class and define how instances of the class should interact with their data and perform operations. They often implement the core functionality of the class.

**Bound to Instances**

When you call an instance method, it is automatically bound to the instance that you call it on. Python passes the instance as the self argument implicitly, and you don’t need to specify it when calling the method.

**Common Use Cases**

Instance methods are commonly used for tasks such as initializing instance attributes in the \_\_**init\_\_** method, performing instance-specific calculations, and modifying instance attributes in a controlled manner

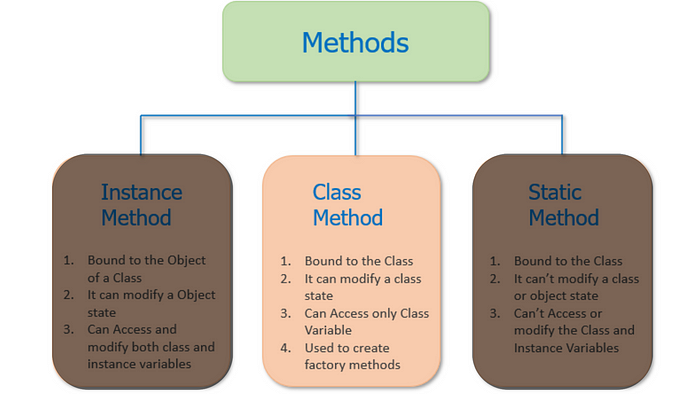
Following cells shows some examples of a class with instance method

class Dog:  
 # defining the constructor method which is an instance method also  
 def \_\_init\_\_(self, name, breed):  
 self.name = name  
 self.breed = breed  
 self.energy = 100  
  
 # the other instance method using the attributes from \_\_init\_\_ method  
 def bark(self):  
 print(f"{self.name} the {self.breed} barks loudly!")  
  
 # the new instance method performing operations on its attributes  
 def play(self, minutes):  
 self.energy -= minutes # self.energy is defined in \_\_init\_\_ as 100  
 if self.energy < 0:  
 self.energy = 0  
 print(f"{self.name} has {self.energy}% energy left.")

#defining the object  
dog1 = Dog('Bruno', 'German Shephard')  
  
#calling the bark method for dog1  
dog1.bark()  
  
# calling the play method for dog1  
dog1.play(15)  
  
Output:  
Bruno the German Shephard barks loudly!  
Bruno has 85% energy left.

Instance methods are fundamental method in OOP, as they allow us to define the behavior and actions associated with objects of a class. Moving forward, we will discuss more types of methods present in the OOP in python.

**Class Method**



We have already discussed about 4 pillars of python OOPs in this article, and about the Decorators in Python in this one.

Moving forward in the series, we will now discuss another concept in OOPs in Python which is the Class Method.

In Python, a class method is a method that is bound to the class and not the instance of the class (object). Class methods are defined using the “@classmethod” decorator and take the class itself as their first argument, typically named ‘cls’.

Class methods are often used for operations that are related to the class as a whole rather than to a specific instance of the class.

However, this is not the official definition of class methods. The detailed explanination is provided further in the notebook.

**Definition of Class Method**

To define a class method, we use the ‘@classmethod’ decorator, as discussed earlier, within a class definition. The method must take the ‘cls’ argument as its first parameter in order to create the method a class method in addition to using the ‘@classmethod’ decorator.

We can access the class methods directly through class name and can assign them to variables whenever required and that variable is able to access other methods defined in the class.

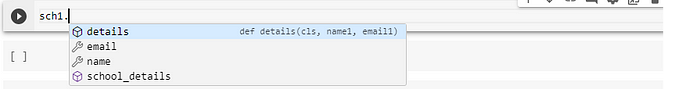
While defining a class method, we always use ‘cls’ instead of ‘self’ as it would bring the method directly to the class unlike the \_\_**init\_\_** method.

The following cells of code explains the concept through code.

# defining a class  
  
class school:  
  
 # defining the constructor \_\_init\_\_() method of the class to enable the class to take the data  
 def \_\_init\_\_(self, name, email):  
 self.name = name  
 self.email = email  
  
 # defining a class method which overloads the \_\_init\_\_() method  
 @classmethod  
 def details(cls, name1, email1):  
 return cls(name1, email1)  
  
 # defining an instance method inside the class  
 def school\_details(self):  
 print("This School was established in 1950s")



# using the class method to add data into the class through constructor   
  
sch1 = school.details('Central School', 'centralschool@gmail.com')



sch1.email  
  
Output:  
centralschool@gmail.com

sch1.name  
  
Output:  
Central School

sch1.school\_details()  
  
Output:  
This School was established in 1950s

As observed in the code above, we can perform function overloading using the class method in python. An example can be seen in the code above where we overloaded the ***\_\_init\_\_*** function. But, what do we exactly mean by function overloading?

**Function Overloading**

Function overloading is a feature that allows us to define multiple functions with the same name in the same scope, but with different parameter lists. The choice of which function to call is determined by the number or types of arguments passed to the function.

Function overloading is a form of polymorphism, where the different functions with the same name can behave differently based on their parameters.

**Static variable and Class Method**

We can also access a static variable defined inside the class through the class method variable. The example is as below.

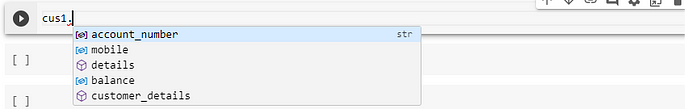
class bank:  
  
 def \_\_init\_\_(self, account\_number, balance):  
 self.account\_number = account\_number  
 self.balance = balance  
   
 mobile = 7089251232  
  
 @classmethod  
 def change\_mobile(cls, mobile\_num):  
 bank.mobile = mobile\_num  
  
 @classmethod  
 def details(cls, account\_number, balance):  
 return cls(account\_number, balance)  
   
 def customer\_details(self):  
 print('account number: ' , self.account\_number)  
 print('balance: ', self.balance)  
 print('Mobile: ', bank.mobile)

Calling the static variable inside the instance method is one way to use the static variable inside a class and through methods. We can also access them through objects and class method variables.

Also, we can use static method to change the value of static method defined inside the class by defining the class method as follows.

@classmethod  
def change\_mobile(cls, mobile\_num):  
 bank.mobile = mobile\_num

cus1 = bank.details('1287655825665', '1153225')



cus1.customer\_details()  
  
Output:  
account number: 1287655825665  
balance: 1153225  
Mobile: 7089251232

cus1.mobile  
  
Output:  
7089251232

cus1.change\_mobile(4535786624)

cus1.mobile  
  
Output:  
4535786624

**Uses and Benefits of Class Methods in OOP in Python**

**01. Accessing and Modifying Class-Level Data:**

Class methods are particularly useful for accessing and modifying class-level data or attributes. They can manipulate or manage data shared among all instances of a class.

**02. Factory Methods:**

Class methods can be used to create and return instances of a class, serving as factory methods. This is helpful when we want to customize the creation of objects, enforce certain constraints, or return cached instances.

**03. Alternate Constructor:**

Class methods can be used to provide an alternative constructors for a class. For example, we can create a class method to initialize instances from a different data format or source.

**04. Singleton Pattern:**

Class methods can be used to implement the Singleton pattern, ensuring that only one instance of a class exists throughout the program.

**05. Namespace Organization:**

Class methods can help organize the namespace by grouping related methods together within a class.

**06. Enhancing Code Readability:**

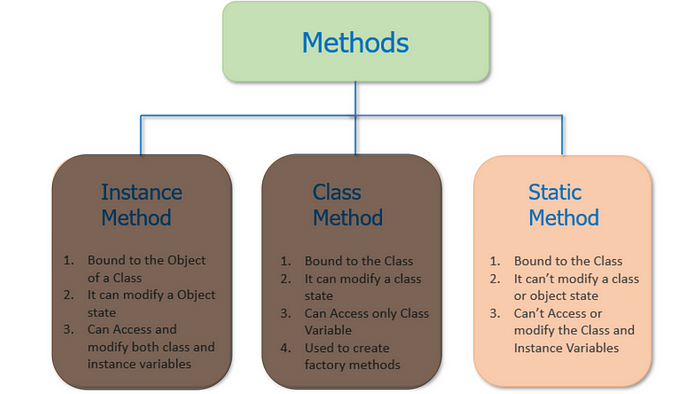
Class methods can improve code readability by explicitly indicating that a method operates at the class level, making it clear that the method does not depend on instance specific state.

**07. Testing and Mocking:**

In unit testing, class methods can be easier to mock and test because they do not rely on instance- specific state.

In Summary, Class methods in python are valuable for operations related to the class itself, class level data, or providing alternative constructors. The class methods help organize code, improve readability, and offer flexibility in managing class specific behavior and attributes within an object-oriented programming paradigm.

**Static Method**



In python object oriented programming, a static method is a method which belongs to a class rather than an instance or object of the class.

It is defines using the ‘@staticmethod’ decorator and does not have access to instance-specific data or methods. Static methods are typically used for operations that are related to the class as a whole rather than to specific instances. The static methods are often used for utility functions, helper methods, or operations that don’t depend on the state of instances.

This article discusses about the static methods in python, their uses and advantages in object oriented programming (OOP) in python.

**Defining a Static Method**

To define a static method, we use the decorator ‘@staticmethod’ within a class definition as already discusses above.

Unlike regular instance methods or class methods, static method does not take ‘self’, or ‘cls’ parameter as their first argument, respectively. In fact, the static method does not take any special first argument by default.

Following lines of code explains how we can define the static methods and use them in class and through objects in python OOP.

# defining a class  
  
class travel:  
  
 def traveller(self, name, place):  
 self.name = name  
 self.place = place  
  
 flight\_num = '23552'  
  
 @staticmethod  
 def travel\_date(travel\_date):  
 print(travel\_date)  
  
 @classmethod  
 def flight\_details(cls, flight\_number):  
 travel.flight\_num = flight\_number  
 return flight\_number

t1 = travel()  
t1.traveller('Isha Choudhary', 'Bombay')  
t1.name  
  
Output  
'Isha Choudhary'

t1.place  
  
Output:  
'Bombay'

#accessing static method  
  
t1.travel\_date('27/10/23')  
  
Output:  
27/10/23

t1.flight\_details(12535)  
  
Output:  
12535

***Note :****Static methods can also be used inside a class method. Also, one static method can be used in other static method.*

Following lines of code shares an example of it.

class datacience\_class:  
 def student\_details(self, name, mail\_id, number):  
 print(name, mail\_id, number)  
  
 @staticmethod  
 def mentor\_mail\_id(mail\_id):  
 print(mail\_id)  
  
 # static method using another static method inside it  
 @staticmethod  
 def mentor\_class(list\_mentor):  
 print(list\_mentor)  
 datacience\_class.mentor\_mail\_id(['isha@gmail.com', 'sha@gmail.com'])  
  
 @classmethod  
 def class\_name(cls, class\_name):  
 cls.mentor\_class(['isha', 'krish'])  
  
 # instance method using static method with one other static method inside it  
 def mentor(self, mentor\_list):  
 print(mentor\_list)  
 self.mentor\_class(['intro class', 'machine learning'])

s1 = datacience\_class()

s1.mentor\_class(['isha', 'sha'])  
  
Output:  
['isha', 'sha']  
['isha@gmail.com', 'sha@gmail.com']

s1.mentor(['isha', 'sha'])  
  
Output:  
['isha', 'sha']  
['intro class', 'machine learning']  
['isha@gmail.com', 'sha@gmail.com']

**Uses and Advantages of Static Methods in OOP**

**01 Organization**

Static methods are used to group related utility functions or helper methods within a class, making the codebase more organized and modular.

**02 No Instance Dependency**

Static methods do not depend on the state of instances. They can be called on the class itself without creating instances, which can be useful when you don’t need to work with instance-specific data.

**03 Code Reusability:**

Static methods promote code reusability since they can be called from multiple places without duplicating code.

**04 Namespace:**

They help organize the namespace by keeping related methods within the class scope, preventing naming conflicts.

**05 Enhanced Readability:**

Static methods improve code readability by indicating that a method does not modify instance state and has a class-level scope.

**06 Performance:**

In some cases, static methods may offer slightly better performance than instance methods since they don’t have the overhead of instance data access.

**07 Unit Testing:**

Static methods can be easily tested in isolation without the need to create instances. They are useful for unit testing and mocking in test-driven development.

In summary, static methods in Python are used to define methods that are associated with a class but do not depend on instance-specific data. They promote code organization, reusability, and readability, making it easier to work with utility functions and class-level operations within the context of object-oriented programming.

**Static (Magic/Dunder) Method**



Dunder Method are Magic Methods or special methods having double underscores before and after the name. These are called dunder methods as they have double underscore.

These methods have a specific meaning and are used to define how objects of a class behave in various different situations.

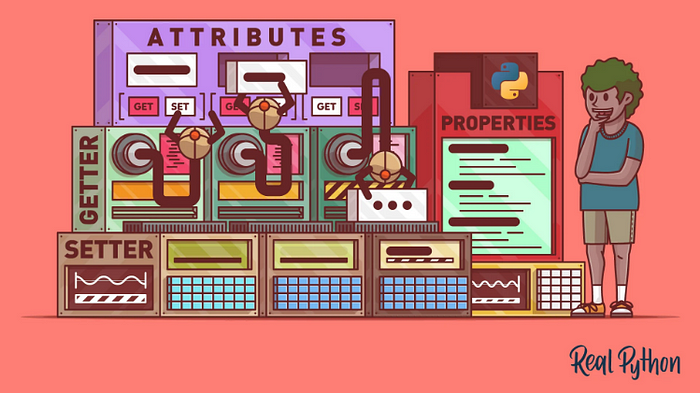
Some examples of dunder method are

1. **‘\_\_init\_\_**’ : we use it as a class instructor
2. **‘\_\_str\_\_**’ — we use it as str()
3. **‘\_\_add\_\_**’ — we use it as ‘+’
4. **‘\_\_repr\_\_**’ — we use it as repr() to return unambiguous string representation of the object. It is generally used in debugging and development.
5. **‘\_\_len\_\_**’ — we use it as len() function to check the length of the object

There are many more like these in python.

Dunder methods are fundamental part of Python’s Object oriented programming and allow you to customize the bahaviour of the objects to suit our needs. We can make our objects more powerful and intuitive by using these methods.

**Property Decorator (Getters, Setters, and Deletes)**



The “Property Decorator” in python is a built in mechanism for defining special methods which control the access, modification, and deletion of attributes or properties of an object.

The property decorator allows us to customize the behavior of “getting”, “setting”, and “deleting” the values of the attributes, and making the implementation of computed properties, and data validation possible.

This decorator is typically used in conjunction with three methods, i.e, getter, setter, and deleter as defined below.

**01 Getter Method**

The getter method is responsible for getting or retrieving the value of the attribute. We decorate this one with ‘@property’. The getter method is called automatically whenever an instance attribute is accessed in order to retrieve the value.

**02. Setter Method**

This method is responsible for setting or can say, assigning the value of the attribute. It is decorated using ‘@attribute\_name.setter’. When an instance’s attribute is assigned a new value, the setter method is automatically called to handle the assignment.

**03. Deleter Method**

The deleter method, as name suggests, is responsible for deleting the attribute. It is decorated using the decorator, ‘@attribute\_name.deleter’. The deleter method is automatically called whenever an instance’s attribute is deleted using the ‘del’ statement.

The following cells of code shows some of the examples, for these property decorators.

#defining the class  
class Circle:  
 def \_\_init\_\_(self, radius):  
 self.\_radius = radius # Private attribute with a leading underscore  
  
 #defining first property decorator  
 @property  
 def radius(self):  
 return self.\_radius  
  
 #defining setter property for recently created property **@radius**  
 @radius.setter  
 def radius(self, value):  
 if value < 0:  
 #defining negative value error  
 raise ValueError("Radius cannot be negative")  
 self.\_radius = value  
  
 @radius.deleter  
 def radius(self):  
 print("Deleting the radius attribute")  
 del self.\_radius  
  
 @property  
 def area(self):  
 return 3.14159 \* self.\_radius \*\* 2

# Creating an instance of Circle  
circle = Circle(5)

# Accessing the radius and area properties  
print(circle.radius) # Calls the getter method  
print(circle.area) # Calls the area property  
  
Output:  
5  
78.53975

# Setting the radius property  
circle.radius = 7 # Calls the setter method

# Deleting the radius property  
del circle.radius # Calls the deleter method  
  
Output:  
Deleting the radius attribute

Using the ‘property’ decorator allows us to create classes with attributes which have custom behavior. This makes our code more robust and maintainable.

The ‘property’ decorator is particularly useful when we need to add data validation, computed properties, or attribute access control to our classes.

This was all about the methods and decorators we have in python. This is part 2 of the Object Oriented Programming in python with all the explaination and coded examples regarding all the topics.