**All About “Decorators” in Python**

OOPs in Python — Extension

[[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----e7260adb9ce7--------------------------------)](https://medium.com/@shahooda637?source=post_page-----e7260adb9ce7--------------------------------)

[Isha Choudhary](https://medium.com/@shahooda637?source=post_page-----e7260adb9ce7--------------------------------)

·

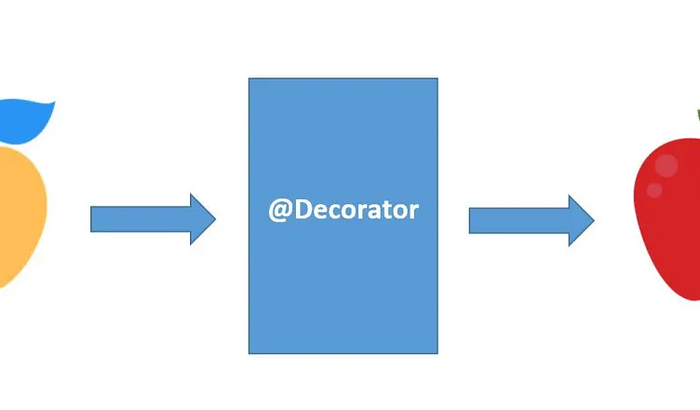
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa0449bd47c69&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40shahooda637%2Fall-about-decorators-python-e7260adb9ce7&user=Isha+Choudhary&userId=a0449bd47c69&source=post_page-a0449bd47c69----e7260adb9ce7---------------------post_header-----------)

3 min read

·

Sep 14

33



*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 the article on*[*OOPs in Python — Part 01*](https://medium.com/@shahooda637/the-python-oops-moments-part-01-9e16bfd4cc09)*.*

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.

**Example 01**

# 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

**Example 02**

#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 bahavior 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.

**Conclusion**

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.

*Have you ever tried tapping the clap twice? Its magical, try it!*