**Understanding Python Decorators With Examples**

The purpose of this article is to help you understand the concept of decorators in Python programming, and how best to use it. It is one of those advanced concepts you will encounter as you continue to learn Python.

Decorators in Python are just functions that take in another function as an argument extending it's functionality without changing it's structure.A decorator wraps another function, amplifies its behaviour and returns it.

In python functions are first class objects which means they can receive arguments or can be passed as arguments.To fully grasps the concept of decorators there are a few things you need to understand.

1. A function is an object, which means it can be assigned to another variable.

def greet():

print("Hello John")

greet\_john = greet

greet\_john()

>>>

Hello John

2. A function can be returned from another function.

def greet():

def greeting\_at\_dawn():

print("Good morning")

return greeting\_at\_dawn

salute = greet()

salute()

>>>

Good morning

3. A function can be passed as an argument of another function. A function that receives another function as its argument is known as a **higher order function**.

def greet\_some(func):

print("Good morning", end=' ')

func()

def say\_name():

print("John")

greet(say\_name)

>>>

Good morning John

The above listed points are necessary and worth knowing, if you must understand Python decorators and how to use them effectively when programming.

**Creating a Python Decorator**

There are two main things to bear in mind when creating a decorator:

• Define an outter function that take a function argument.

• Nest a wrapper function within the outter function, this would also wrap the decorated function.

The most basic decorator looks like this:

def increase\_number(func):

def increase\_by\_one():

print("incrementing number by 1 ...")

number\_plus\_one = func() + 1

return number\_plus\_one

return increase\_by\_one

def get\_number():

return 5

get\_new\_number = increase\_number(get\_number)

print(get\_new\_number())

>>>

incrementing number by 1 ...

6

Looking at the code above, the outter function increase\_number also known as the decorator receives a function argument func. increase\_by\_one is the wrapper function where the decorated get\_number function is found. As you already know by now, the decorator is assigned to another variable. This is a generally syntax for Python decorators.

However, there is a much easier way to represent decorators.A decorator is easily indentified when it begins with the @ prefix coupled with the decorated function underneath. The previous code can be refactored to look like this:

def increase\_number(func):

def increase\_by\_one():

print("incrementing number by 1 ...")

number\_plus\_one = func() + 1

return number\_plus\_one

return increase\_by\_one

@increase\_number

def get\_number():

return 5

print(get\_number())

>>>

incrementing number by 1 ...

6

The examples shows that a decorator extends the functionality of its function argument. There are cases where you may need to pass arguments to a decorator. The way around this is to pass arguments to the wrapper function which would be passed down to the wrapped/decorated function. Another example:

def multiply\_numbers(func):

def multiply\_two\_numbers(num1, num2):

print("we're multiplying two number {} and {}".format(num1, num2))

return func(num1, num2)

return multiply\_two\_numbers

@multiply\_numbers

def multiply\_two\_given\_numbers(num1, num2):

return f'{num1} \* {num2} = {num1 \* num2}'

print(multiply\_two\_given\_numbers(3, 4))

>>>

we're multiplying two number 3 and 4

3 \* 4 = 12