DECORATORS IN PYTHON

Python has an interesting feature called decorators to add functionality to an existing code.

- Decorators allow us to wrap another function in order to extend the behaviour of wrapped function, without permanently modifying it.
- '@' symbol is used to denote a decorator.

In order to understand about decorators, we must first know a few basic things in Python. We must be comfortable with the fact that everything in Python are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too.

BASIC THINGS:

1. **Various different names can be bound to the same function object.**

```
def first(msg):
        print(msg)
first("Hello")  #Hello
second = first
second("Hello")  #Hello
del first
#first("Hello")    #Error first is not defined
second("Hello")  #Hello
```

Here the names first and second refer to same function object. Even if we delete first() function, we can call second() function.

2. **One function can call another function.**

```
def first():
   print("First")
   def second():
      print("Second")
   second()
first()
```

OUTPUT:

```
First
Second
```

3. **One function can return another function.**

```
def first():
   print("First")
   def second():
      print("Second")
   return second
```

```
x = first()    #x = second
x()
```

OUTPUT:
First
Second

4. **One function can be passed as a parameter to another function.**

```
def first():
   print("First")
def second(func):
   func()
   print("Second")
second(first)
```

OUTPUT:
First
Second

There are two types of decorators in python:

1. Function decorator
2. Class decorator

FUNCTION DECORATOR

A decorator takes in a function, adds some functionality and returns it.

- **Decorating functions without parameters and without '@' symbol.**

```
def outer_func(func):
        def inner_func():
                print("Before calling actual function")
                func()
                print("After calling actual function")
        return inner_func
def m1():
        print("m1 function")
m1 = outer_func(m1)
m1()
```

OUTPUT:
Before calling actual function
m1 function
After calling actual function

Here our actual function i.e m1 got decorated with lines "Before calling actual function" and "After calling actual function".

- **Decorating functions without parameters and with '@' symbol.**

```
def outer_func(func):
        def inner_func():
                print("Before calling actual function")
                func()
                print("After calling actual function")
        return inner_func
@outer_func
def m1():
        print("m1 function")
m1()
```

OUTPUT:
Before calling actual function
m1 function
After calling actual function

- **Decorating functions with parameters.**

```
def outer_func(func):
        def inner_func(a,b):
                print("Before calling actual function")
                func(a,b)
                print("After calling actual function")
        return inner_func
@outer_func
def m1(a,b):
        print(a+b)
m1(10,20)
```

OUTPUT:
Before calling actual function
30

- **Decorating functions with parameters and return statement.**

```
def outer_func(func):
        def inner_func(a,b):
                print("Before calling actual function")
                result = func(a,b)
                print("After calling actual function")
                return result
        return inner_func
@outer_func
def m1(a,b):
        print("Inside m1 function")
        return a+b
```

```
ans = m1(10,20)
print(ans)
```

OUTPUT:

```
Before calling actual function
Inside m1 function
After calling actual function
30
```

CLASS DECORATOR

We have seen that decorators are callable that accepts and returns a callable. Since classes are callable, decorators are also used to decorate classes. Before we see class decorators, we should know one fact that objects are callable using __call__ method.

Python has a set of built-in methods and __call__ is one of them. The __call__ method enables a programmer to write classes where the instances behave like functions and can be called like a function. E.g.

```
class Example:
   def __init__(self):
      print("Instance Created")
   def __call__(self):
      print("Instance is called via special method")

e = Example()

e() # __call__ method will be called automatically
```

OUTPUT :

```
Instance Created
Instance is called via special method
```

Now we will see how to decorate with class

```
class MyDecorator:
   def __init__(self, func):
      self.func = func  #self.func = m1

   def __call__(self):
      print("Before calling actual function")
      self.func()  #m1()
      print("After calling actual function")

@MyDecorator
def m1():
   print("m1 function")

m1()
```

OUTPUT:

Before calling actual function
m1 function
After calling actual function

Here when we call m1(), internally constructor of class MyDecorator will get called and immediately __call__() method will get called. In constructor, we should collect the function name which we want to decorate.