Python Decorators

**A decorator takes in a function, adds some functionality and returns it. In this article, you will learn how you can create a decorator and why you should use it.**

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

This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

## What you need to know before learning decorators?

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 (Yes! Even classes), are [objects](https://www.programiz.com/python-programming/class). Names that we define are simply identifiers bound to these objects. [Functions](https://www.programiz.com/python-programming/function) are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object.

Here is an example.

def first(msg):

print(msg)

first("Hello")

second = first

second("Hello")

When you run the code, both functions first and second gives same output. Here, the names first and second refer to the same function object.

Now things start getting weirder.

Functions can be passed as arguments to another function.

If you have used functions like map, filter and reduce in Python, then you already know about this.

Such function that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.

def inc(x):

return x + 1

def dec(x):

return x - 1

def operate(func, x):

result = func(x)

return result

We invoke the function as follows.

>>> operate(inc,3)

4

>>> operate(dec,3)

2

Furthermore, a function can return another function.

def is\_called():

def is\_returned():

print("Hello")

return is\_returned

new = is\_called()

#Outputs "Hello"

new()

Here, is\_returned() is a nested function which is defined and returned, each time we call is\_called().

## Getting back To decorators

Functions and methods are called **callable** as they can be called.

In fact, any object which implements the special method \_\_call\_\_() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

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

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

def ordinary():

print("I am ordinary")

>>> ordinary()

I am ordinary

>>> # let's decorate this ordinary function

>>> pretty = make\_pretty(ordinary)

>>> pretty()

I got decorated

I am ordinary

In the example shown above, make\_pretty() is a decorator. In the assignment step.

pretty = make\_pretty(ordinary)

The function ordinary() got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

ordinary = make\_pretty(ordinary).

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

@make\_pretty

def ordinary():

print("I am ordinary")

is equivalent to

def ordinary():

print("I am ordinary")

ordinary = make\_pretty(ordinary)

This is just a syntactic sugar to implement decorators.

## Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

def divide(a, b):

return a/b

This function has two parameters, a and b. We know, it will give error if we pass in b as 0.

>>> divide(2,5)

0.4

>>> divide(2,0)

Traceback (most recent call last):

...

ZeroDivisionError: division by zero

Now let's make a decorator to check for this case that will cause the error.

def smart\_divide(func):

def inner(a,b):

print("I am going to divide",a,"and",b)

if b == 0:

print("Whoops! cannot divide")

return

return func(a,b)

return inner

@smart\_divide

def divide(a,b):

return a/b

This new implementation will return None if the error condition arises.

>>> divide(2,5)

I am going to divide 2 and 5

0.4

>>> divide(2,0)

I am going to divide 2 and 0

Whoops! cannot divide

In this manner we can decorate functions that take parameters.