Python Decorators

## python decorators

## Python Decorators: Introduction

Python decorators are one of the powerful features of Python which allows you to dynamically change the behavior or extend the functionality of a function.

Decorators are perfect for the situation where we need to extend the functionality of a function without actually modifying it.

Decorators in their simplest form look somewhat like this. @example\_decorator

def example\_func(foo):

#function body

return something

Where, example\_decorator is a decorator and example\_func is to be decorated.

Python decorators are hard to get along with. Once understood we can do so many powerful things but initially, programmers find it difficult to understand.

So before jumping into details of Python decorators, you need to learn some of the underlying mechanism of functions in Python that must be understood first.

### Functions are the first class objects

Everything in Python is an object and functions(even classes as well) are no exceptions to it. Functions are also objects with attributes.

A function can be assigned to any variable as a normal object. Functions can also be passed to functions as arguments and yes, they can also be returned from a function as values.

Apart from this, a function can also be defined inside another function called nested functions. Let’s take an example of each to demonstrate these above-mentioned features.

#### A function can be assigned to a variable as normal object

def func():

print ('Hello')

obj = func()

obj

#ouput: Hello

#### A function can be passed as argument to another function

def func1(name):

print ('Hello', name)

def func2(func)

name = 'John'

return func(name)

Let’s pass function func1 as the argument in func2 in the interpreter.

>>> func2(func1)

Hello John

#### A function can be defined under another function and function can also return another function: Nested function

def func1():

def func2(): #nested function

print ('Hello')

return func2() #returning function

Let’s try this on the interpreter.

>>> func1()

Hello

Learn about [Python closures](http://www.trytoprogram.com/python-programming/python-closures/) to know in detail about nested function and enclosing scope.

## How to create Python decorators?

So, now finally let’s dig into how to actually create Python decorators.

A decorator is just a callable that takes a function as an argument and returns a replacement function.  Anything that implements \_\_call\_\_() function behind the scene is called callable and decorator is one of them.

Previously we mentioned that with decorators we can extend the functionality of a function without actually modifying it. Let’s see and implement this in a program.

def display():

print ('Harry')

#defining decorator function

def decorator\_func(func):

def inner\_func():

print ('I am decorated Harry')

return inner\_func()

#Original function

display()

#After decorating with decorator\_func

decorator\_func(display)

**Output**

Harry

I am decorarted Harry

### Python Decorator’s Syntax

The decoration occurs in the line just before the function. The symbol ‘@’ is followed by the decorator functions name.

#defining decorator function

def decorator\_func(func):

def inner\_func():

print ('I am decorated Harry')

return inner\_func()

@decorator\_func

def display():

print ('Harry')

## Python decorators with parameters

Decorators can be both parameterized or non-parameterized. We have already discussed decorators without parameters. Now well will learn about decorators with parameters.

Let’s create a simple decorator to find the square of a number. Here both the function and the decorator will have parameters.

def perfect\_square(str\_param):

def middle\_decorator(func):

def inner(x):

print (str\_param,' of ',x)

print ('The sqaure of', x ,' is ')

return func(x)

return inner

return middle\_decorator

@perfect\_square('Find the square')

def display\_square(a):

return a\*a

Let’s see what happens when we try the function.

>>> display\_square(2)

Find the square of 2

The square of 2 is

4

So far we have used functions as decorators. But in Python, we can use classes as decorators as well. Let’s learn how we can use Python classes as decorators.

## Python Decorator Classes

We already know that decorator simply is a callable object which takes functions as arguments. In Python, we can also define other objects as a callable using \_\_call\_\_() method.

Here we will learn how to make a class callable using \_\_call\_\_() function and use that class as a decorator.

Let’s make a decorator class used to decorate a function for finding the square of a number.

class decorator\_class:

def \_\_init\_\_(self, x):

self.x = x

def \_\_call\_\_(self, a):

print("The square of", a, 'is')

print (self.x(a))

@decorator\_class

def display\_square(a):

return a\*a

Let’s see what happens when we try this on interpreter.

>>> display\_square(2)

The square of 2 is

4

That’s pretty much what we expected. What we did was defined a decorator class and used it to decorate the function display\_square.