**Section 12: Python Decorators**

**09.03.**

**98. Decorators with Python Overview**

Allows to decorate a function

It can be considered as an on/off switch to add/remove functionalities to a function

The additional functionality can be added to an existing function

A decorator is defined with @ and is placed at the top of an already defined function

Functions can be passed as arguments and defined inside another functions

For example:

def my\_func():

return “returned text”

func\_var = my\_func

func\_var()

Even if then we delete the my\_func() function, we still have access to func\_var()

This is possible because the copy is deep, not just a reference

To define a decorator, we must do next steps:

def new\_decorator(original\_func):

def wrap\_func():

# code

original\_func()

# code

return wrap\_func

Next we can apply the decorator in the following way:

@new\_decorator

def func\_needs\_decorator():

# code

In this way the funct\_needs\_Decorator will be passed inside new\_decorator

We can turn on/off this by putting @new\_decorator or not (above the function definition)

**99. Decorators Homework**

Decorators are heavily used in web frameworks as Flask and Django

**10.03.**

<https://www.programiz.com/python-programming/decorator>

The decorators are considered metaprogramming

That is because a part of the program modifies another part of the program

Even functions are objects in Python

A higher order function is a function that takes other functions as arguments

The decorator acts as a wrapper

We can pass arguments in decorated functions

These arguments are passed in the functions inside the decorator (inside wrap\_func())

For more flexibility we can use \*args and \*\*kwargs

Decorators can be chained by putting multiple @ on sequential rows above a function

The order matters because the decorators will be applied in a bottom-up fashion