DECORATORS

Decorating Functions : Defining functions by nesting them into another function (decorator).

By the Syntax,

@decorator\_function

def function():

    <expression>

Where,

The decorator function is defined first,

def decorator\_function(*function\_placeholder*):

    def wrapper():

        <decoration><function\_placeholder>

The actual flow of work is,

def decorator\_function(*function\_placeholder*):

    def wrapper():

        <decoration><function\_placeholder>

    return wrapper

def function():

    <expression>

function=decorator\_function(function)

function()

Note:

The functions that are being nested into another function are referred to as candy function.

Decorators - @funcs

Decorators are functions that can be used as a template using the ‘@’- Symbol. Used before defining another function.

See: Closures, Nested and Non-Local scope for more understanding.

An example Closure,

# An simple closure

def parent(*child*):

        def *child*():

                print('Closure has been called')

        return *child*

my\_closure=parent('a\_string\_passer\_that\_takes\_the\_palce\_of\_Child')

print(parent)#checking if parent function exists

del parent

# print(parent) The closure exists even after the parent has been deleted

my\_closure()

# DECORATOR

The Decorator function consists of a Wrapper function which has the decorations and calls the candy functions.

## Creating an Decorator

1. The Decorator function is defined with a parameter\_placeholder for candy function .

2. A Wrapper function is defined within the decorator.

3. The decorations are placed within the wrapper

4. The placeholder for the candy function is also called inside the wrapper

5. The wrapper is returned within the decorator function at the end without parameters.

# define decorator

def decorator\_parent(*candy\_child*):

    # define wrapper

    def wrapper():

        print("wrapper start")

        # calling The CANDY inside the wrapper

*candy\_child*()

        print("wrapper end")

    return wrapper

# Fuction to be decorated

def A\_candy\_function():

    print("Candy")

# Calling the purpose to be called inside the decorator

candy\_wrapped\_by\_decorator\_actually\_a\_closure = decorator\_parent(A\_candy\_function)

candy\_wrapped\_by\_decorator\_actually\_a\_closure()

We can also wrap multiple different candy functions with the same decorator.

We look into decorators for their applications in Flask. Here the flask applications are using the route method In flask for routing pages to the html or css pages.

# CHAINING DECORATORS:

Decorator chaining works on the same principle of calling functions inside another functions being called inside another function.

The proceedings are as follows,

@decorator3  
@decorator2  
@decorator1  
def func():  
 return object\_to\_be\_wrapped\_by\_multiple\_decorators

def decorator\_3(*candy*):

    def wrapper():

        return *candy*()+100

    return wrapper

def decorator\_2(*candy*):

    def wrapper():

        return *candy*()+10

    return wrapper

def decorator\_1(*candy*):

    def wrapper():

        return *candy*()+1

    return wrapper

@decorator\_3

@decorator\_2

@decorator\_1

def number\_candy():

    return 10000

print(number\_candy())

OUTPUT:

10111