DECORATORS

An introduction to Decorators:

In Python, functions are the first class objects, which means that –

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

In simple terms:

- In Python, we can define a function inside another function.
- In Python, a function can be passed as parameter to another function (a function can also return another function).

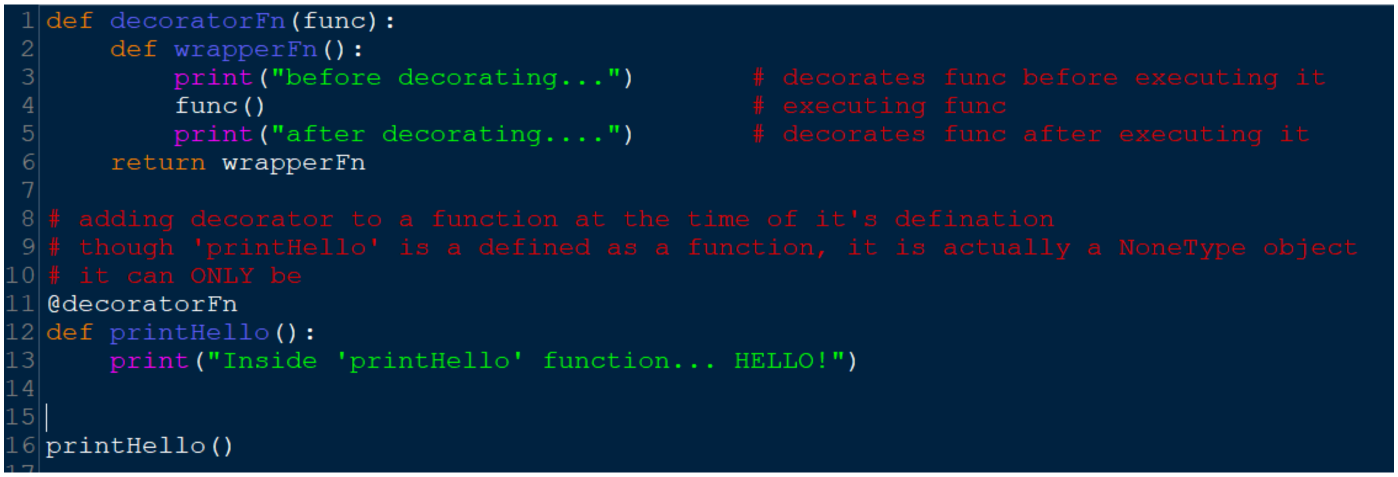
Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

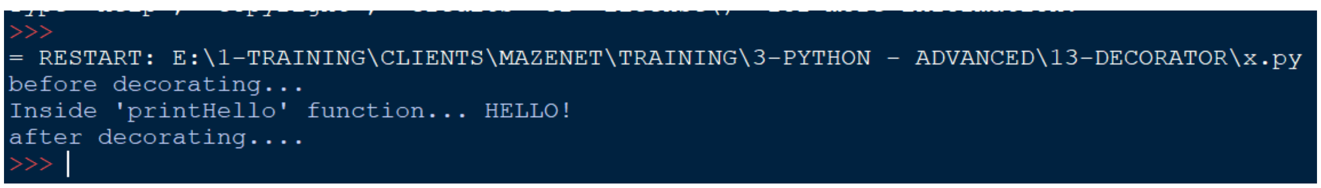
- A decorator is a tool for wrapping code around functions or classes. Decorators then explicitly apply that wrapper to functions or classes to cause them to "opt in" to the decorator's functionality.
- Decorators are extremely useful for *addressing common prerequisite cases before a function runs* (for example, ensuring authentication), or ensuring cleanup after a function runs (for example, output sanitization or exception handling).
- They are also useful for taking action on the decorated function or class itself. For example, a decorator might register a function with a signaling system or a URI registry in web applications.

At its core, a decorator is a callable that accepts a callable and returns a callable.

- A decorator is simply a function (or other callable, such as an object with a __call__ method) that accepts the decorated function as its positional argument.
- The decorator takes some action using that argument, and then either returns the original argument or some other callable (presumably that interacts with it in some way).
- Because functions are first-class objects in Python, they can be passed to another function just as any other object can be.



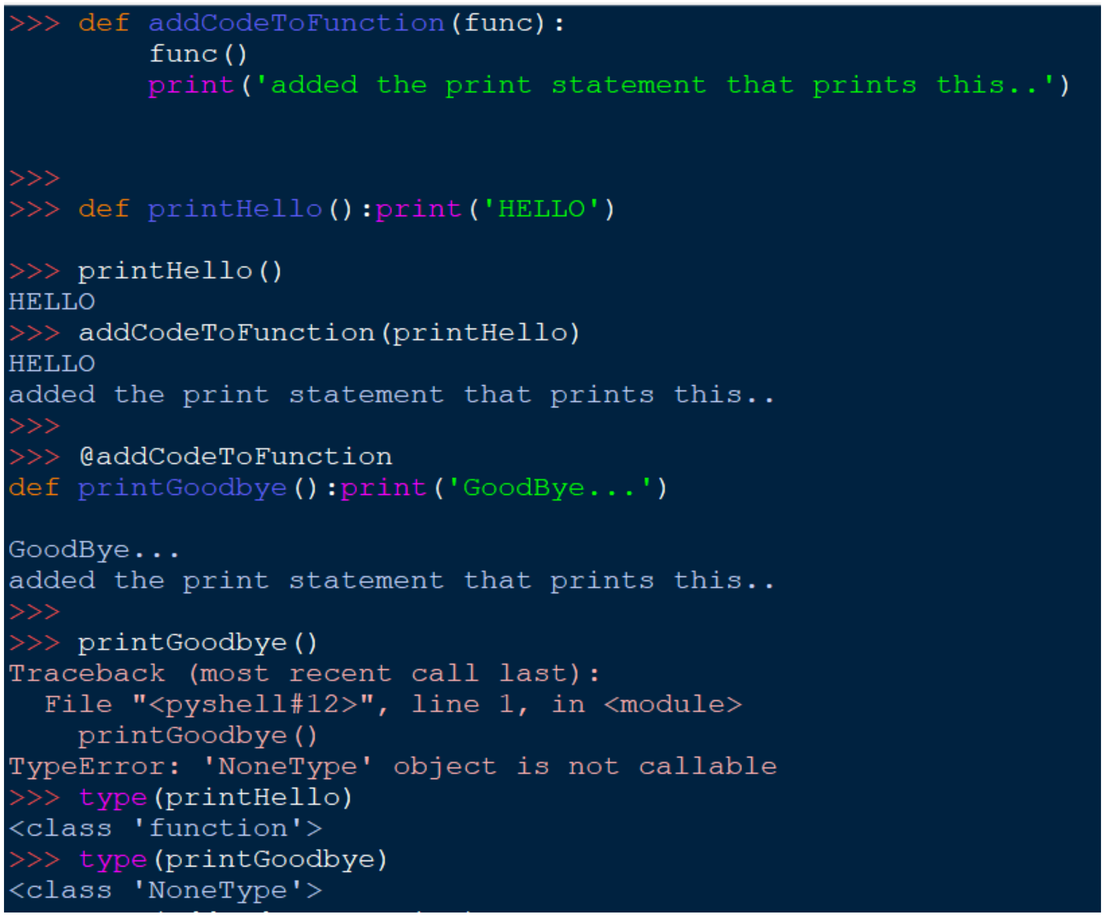
```python
def decoratorFn(func):
    def wrapperFn():
        print("before decorating...")        # decorates func before executing it
        func()                               # executing func
        print("after decorating....")        # decorates func after executing it
    return wrapperFn

# adding decorator to a function at the time of it's defination
# though 'printHello' is a defined as a function, it is actually a NoneType object
# it can ONLY be
@decoratorFn
def printHello():
    print("Inside 'printHello' function... HELLO!")


printHello()
```



```
>>>
= RESTART: E:\1-TRAINING\CLIENTS\MAZENET\TRAINING\3-PYTHON - ADVANCED\13-DECORATOR\x.py
before decorating...
Inside 'printHello' function... HELLO!
after decorating....
>>>
```

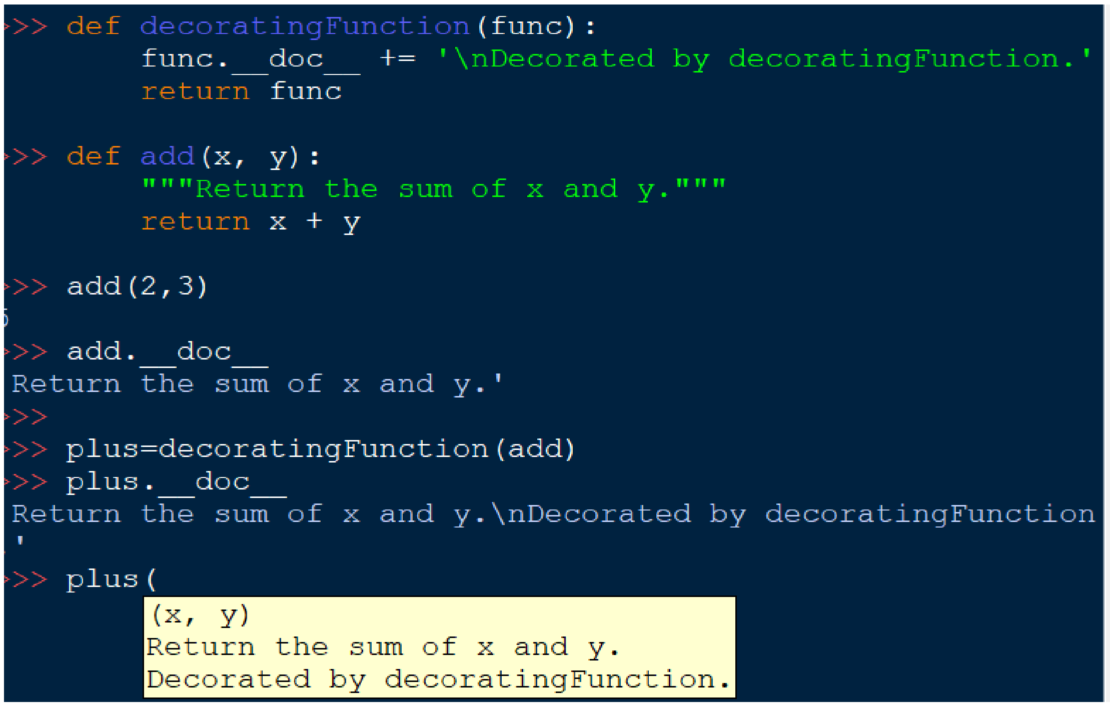
DECORATORS

A decorator is just a function that takes another function as input, and does something with it.



```
>>> def addCodeToFunction(func):
        func()
        print('added the print statement that prints this..')


>>>
>>> def printHello():print('HELLO')

>>> printHello()
HELLO
>>> addCodeToFunction(printHello)
HELLO
added the print statement that prints this..
>>>
>>> @addCodeToFunction
def printGoodbye():print('GoodBye...')

GoodBye...
added the print statement that prints this..
>>>
>>> printGoodbye()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    printGoodbye()
TypeError: 'NoneType' object is not callable
>>> type(printHello)
<class 'function'>
>>> type(printGoodbye)
<class 'NoneType'>
```

Consider the following very simple decorator. It does nothing except append a line to the decorated callable's **docstring**.



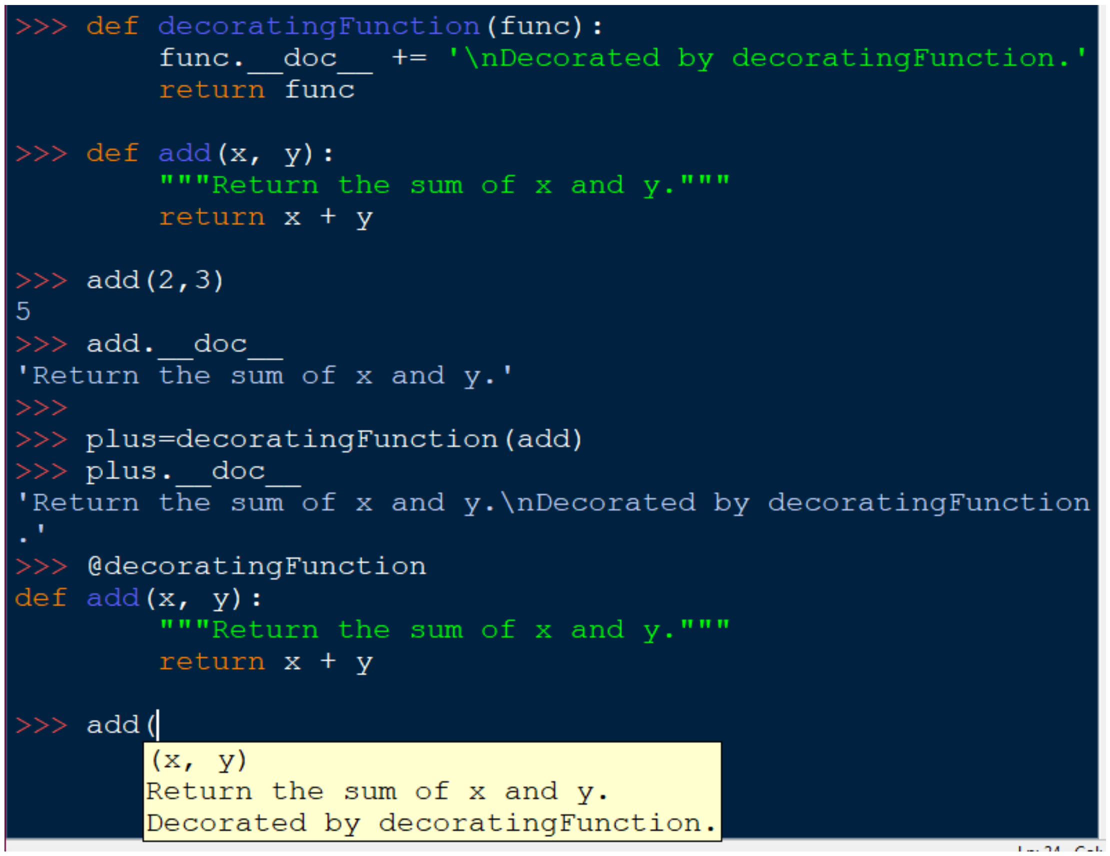
```
>> def decoratingFunction(func):
        func.__doc__ += '\nDecorated by decoratingFunction.'
        return func

>> def add(x, y):
        """Return the sum of x and y."""
        return x + y

>> add(2,3)
5
>> add.__doc__
'Return the sum of x and y.'
>>
>> plus=decoratingFunction(add)
>> plus.__doc__
'Return the sum of x and y.\nDecorated by decoratingFunction
.'
>> plus(
        (x, y)
        Return the sum of x and y.
        Decorated by decoratingFunction.
```

What has happened here is that the decorator made the modification to the function's __doc__ attribute, and then returned the original function object.

Decorator Syntax

- Most times that developers use decorators to decorate a function, they are only interested in the final, decorated function. Keeping a reference to the undecorated function is ultimately superfluous.
- Because of this (and also for purposes of clarity), it is undesirable to define a function, assign it to a particular name, and then immediately reassign the decorated function to the same name.
- Therefore, Python 2.5 introduced a special syntax for decorators. Decorators are applied by prepending an @ character to the name of the decorator and adding the line (without the implied decorator's method signature) immediately above the decorated function's declaration.

Following is the preferred way to apply a decoratingFunction **decorator to the add method:**

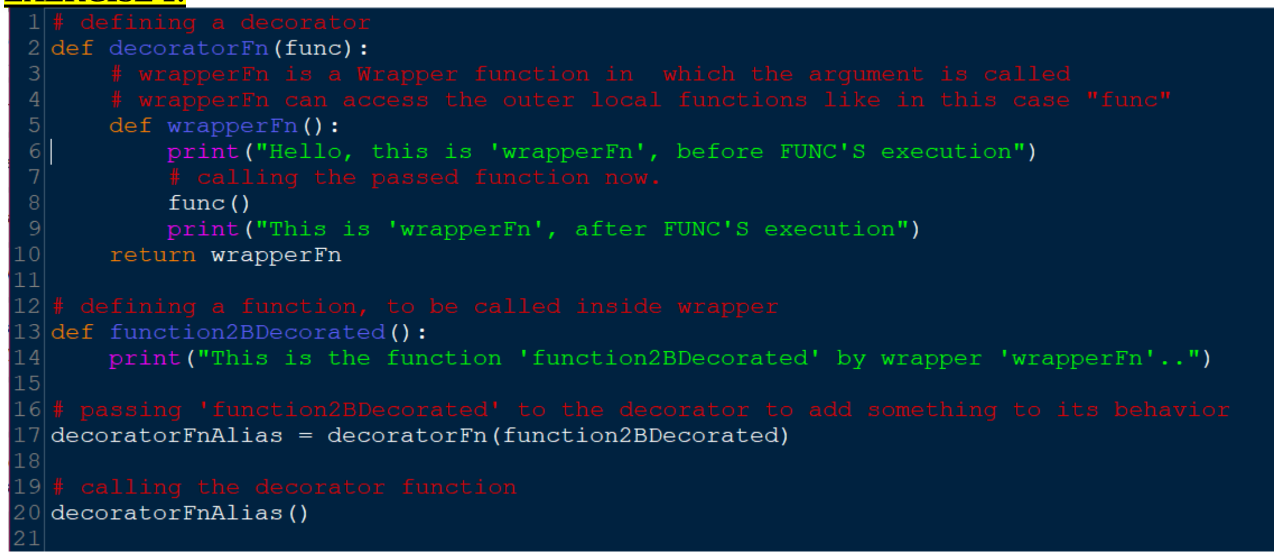


```
>>> def decoratingFunction(func):
        func.__doc__ += '\nDecorated by decoratingFunction.'
        return func

>>> def add(x, y):
        """Return the sum of x and y."""
        return x + y

>>> add(2,3)
5
>>> add.__doc__
'Return the sum of x and y.'
>>>
>>> plus=decoratingFunction(add)
>>> plus.__doc__
'Return the sum of x and y.\nDecorated by decoratingFunction
.'
>>> @decoratingFunction
def add(x, y):
        """Return the sum of x and y."""
        return x + y

>>> add(
       (x, y)
       Return the sum of x and y.
       Decorated by decoratingFunction.
```

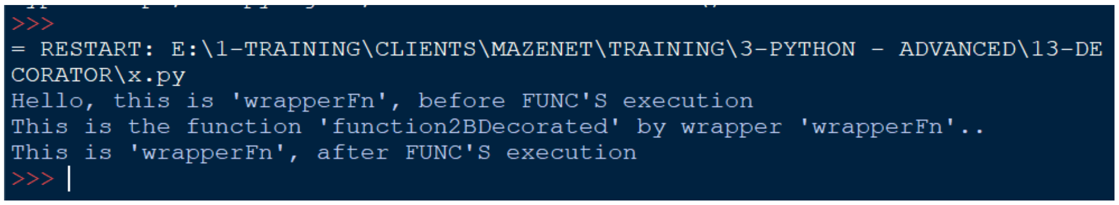
Note again that no method signature is being provided to **@decoratingFunction**. The decorator is assumed to take a single, positional argument, which is the method being decorated.

This syntax allows the decorator to be applied where the function is declared, which makes it easier to read the code and immediately realize that the decorator is in play. Readability counts.

EXERCISE 1:

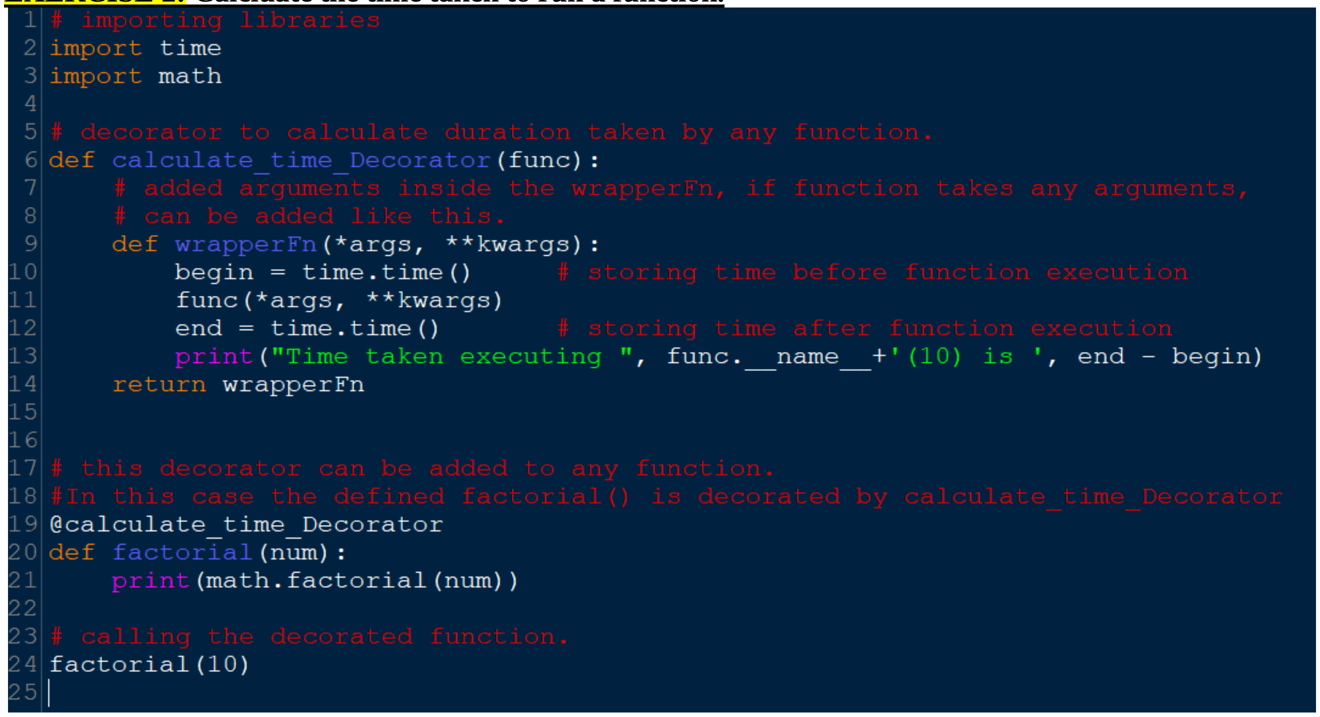


```python
# defining a decorator
def decoratorFn(func):
    # wrapperFn is a Wrapper function in  which the argument is called
    # wrapperFn can access the outer local functions like in this case "func"
    def wrapperFn():
        print("Hello, this is 'wrapperFn', before FUNC'S execution")
        # calling the passed function now.
        func()
        print("This is 'wrapperFn', after FUNC'S execution")
    return wrapperFn

# defining a function, to be called inside wrapper
def function2BDecorated():
    print("This is the function 'function2BDecorated' by wrapper 'wrapperFn'..")

# passing 'function2BDecorated' to the decorator to add something to its behavior
decoratorFnAlias = decoratorFn(function2BDecorated)

# calling the decorator function
decoratorFnAlias()
```

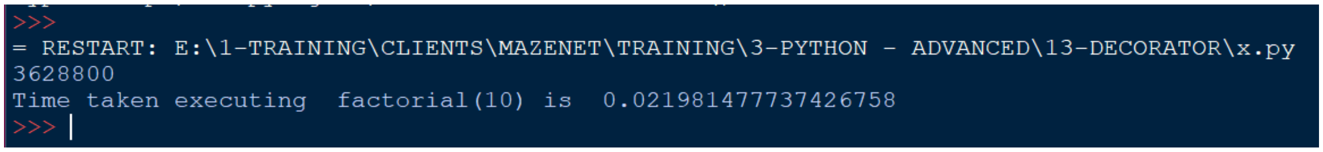


```
>>>
= RESTART: E:\1-TRAINING\CLIENTS\MAZENET\TRAINING\3-PYTHON - ADVANCED\13-DE
CORATOR\x.py
Hello, this is 'wrapperFn', before FUNC'S execution
This is the function 'function2BDecorated' by wrapper 'wrapperFn'..
This is 'wrapperFn', after FUNC'S execution
>>> 
```

EXERCISE 2: Calcluate the time taken to run a function.



```python
# importing libraries
import time
import math

# decorator to calculate duration taken by any function.
def calculate_time_Decorator(func):
    # added arguments inside the wrapperFn, if function takes any arguments,
    # can be added like this.
    def wrapperFn(*args, **kwargs):
        begin = time.time()       # storing time before function execution
        func(*args, **kwargs)
        end = time.time()         # storing time after function execution
        print("Time taken executing ", func.__name__+'(10) is ', end - begin)
    return wrapperFn


# this decorator can be added to any function.
#In this case the defined factorial() is decorated by calculate_time_Decorator
@calculate_time_Decorator
def factorial(num):
    print(math.factorial(num))

# calling the decorated function.
factorial(10)
```



```
>>>
= RESTART: E:\1-TRAINING\CLIENTS\MAZENET\TRAINING\3-PYTHON - ADVANCED\13-DECORATOR\x.py
3628800
Time taken executing  factorial(10) is  0.021981477737426758
>>> 
```