INTERMEDIATE PYTHON

LESSON 8 |Decorator| 28-01-19

***Video tutorial*:**

Python decorator:

Python decorator

By definition, a decorator is a function that takes **another function** and extends the behavior of the latter function without explicitly modifying it.

*Python () function (**another function**)*

```
def python():
    print("Hello Mostapha")

python()
```

Output of python() function

```
Hello Mostapha
```

The output below is the **python ()** function output **decorated** with asterisks

```
***************
Hello Mostapha
***************
```

How can one add the arteristeric without adjusting the **python()** function **?**

How can one extend the behavior of the **python ()** function without explicitly modifying it?

You can do that with a decorator

To properly understand the decorator function, you must understand the python closure properly

The code in the **first rectangle** is the code that has decorated the python function with asterisks

Decorators are usually called **before** the definition of a function you want to decorate

How to call a decorator?

We simply use th @ symbol before the function we'd like to decorate. (see the second small rectangle => @mydecorator)

```
def mydecorator(func):
    def wrapper():
        print(15 * "*")
        func()
        print(15 * "*")
    return wrapper

@mydecorator
def python():
    print("Hello Mostapha")

python()
```

Decorating Functions With Arguments

Python(name) function with name as argument

```
def python(name):
    print( "Hello {}".format(name))

python("Mostapha")

python("Chings")

python("Vickey")
```

Output of python(name) function

```
Hello Mostapha
Hello Chings
Hello Vickey
```

What to do to decorate the output of python(name)function with asterisks without explicitly modifying the it?

We are going to use a decorator of course

```
***************
Hello Mostapha
***************
***************
Hello Chings
***************
***************
Hello Vickey
***************
```

The decorator for functions with arguments should look like this

```
def mydecorator(func):
    def wrapper(*args, **kwargs):
        print(15 * "*")
        func(*args,**kwargs)
        print(15 * "*")
    return wrapper

@mydecorator
def python(name):
    print( "Hello {}".format(name))

python("Mostapha")

python("Chings")

python("Vickey")
```

Note:

To define a general purpose decorator that can be applied to any function we use **`args`** and **`**kwargs`**. **`args`** and **`**kwargs`** collect all positional and keyword arguments and stores them in the args and kwargs variables. **`args`** and **`kwargs`** allow us to pass as many arguments as we would like during function calls.

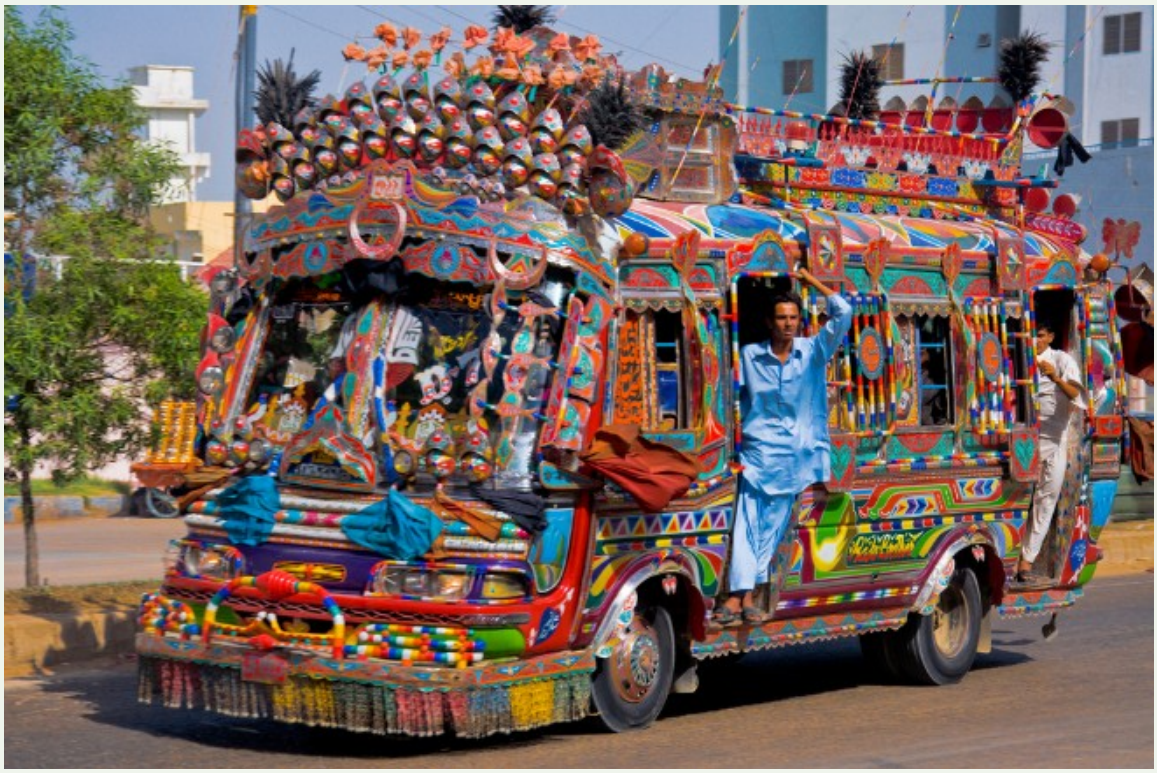
Nesting decorators

You can apply several decorators to a function by stacking them on the top of each other.

Is it possible to decorate the python function with two decorators *** decorator and ^^^^ decorator?

```
^^^^^^^^^^^^^^^^
****************
Hello Mostapha
****************
^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^
****************
Hello Chings
****************
^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^
****************
Hello Vickey
****************
^^^^^^^^^^^^^^^^
```

Yes it is possible



The city bus is decorated with several decorators

The python() function is decorated by two decorators

```
def my_1_decorator(func):
    def wrapper(*args,**kwargs):
        print(15 * "^")
        func(*args,**kwargs)
        print(15 * "^")
    return wrapper

def my_2_decorator(func):
    def wrapper(*args,**kwargs):
        print(15 * "*")
        func(*args,**kwargs)
        print(15 * "*")
    return wrapper


@my_1_decorator
@my_2_decorator
def python(name):
    print( "Hello {}".format(name))

python("Mostapha")
python("Chings")
python("Vickey")
```

Decorator of decorators

```
***************
Hello Chings
***************
Do you understand python decorator?
***************
Hello Vickey
***************
Do you understand python decorator?
```

```
def mymessagedecorator(msg):
    def mydecorator(func):
        def wrapper(*args,**kwargs):
            print(15 * "*")
            func(*args,**kwargs)
            print(15 * "*")
            print(msg)
        return wrapper
    return mydecorator


@mymessagedecorator("Do you understand python decorator?  ")
def python(name):
    print( "Hello {}".format(name))

python("Chings")

python("Vickey")
```

<u>Exercises</u>

Exercise 1:

Write a decorator that prints the time a function takes to execute,

Exercise 2:

Write a decorator that prints the activity of the script.

Exercise 3:

Write a decorator that counts and prints the number of times a function has been executed

Exercise 4:

Write a python decorator which decorates functions with one argument. The decorator should take one argument, a type, and then returns a decorator that makes function should check if the input is the correct type. If it is wrong, it should print("Bad Type")

Answers:

Answer 1:

```
def time_to_excute(func):
    import time
    def wrapper(*args, **kwargs):
        t = time.time()
        func(*args, **kwargs)
        print (func.__name__,":", round(time.time()-t,3))
    return wrapper

@time_to_excute
def test():
    for i in range(10):
        print(i)
test()
```

Answer 2:

```
def script_activuty(func):
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        print (func.__name__, args, kwargs)
    return wrapper



@script_activuty
def test():
    for i in range(3):
        return pow(i,2)
for i in range(3):
    test()
```

Answer 3:

```
def counter(func):
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        func(*args, **kwargs)
        print ('{0} has been used: {1}x'.format(func.__name__, wrapper.count))
    wrapper.count = 0
    return wrapper

@counter
def test():
    for i in range(3):
        return pow(i,2)
for i in range(5):
    test()
```

Answer 4:

```
def type_check(correct_type):
    def check(old_function):
        def new_function(arg):
            if (isinstance(arg, correct_type)):
                return old_function(arg)
            else:
                print("Bad Type")
        return new_function
    return check

@type_check(int)
def times2(num):
    return num*2

print(times2(2))
times2('Not A Number')

@type_check(str)
def first_letter(word):
    return word[0]

print(first_letter('Hello World'))
print(first_letter(('Not', 'A', 'String')))
```