1) ★ Python Decorators

= functions which modify the functionality of another function. They help to make your code shorter and more "Pythonic"

Eg:- (i) s = 'This is a global variable'
def func():
print locals()

print globals()
o/p = [---------- ---------- ----------]

this is a dictionary of all the globals and what they stand for

(ii) print globals()['s']
o/p = This is a global variable.

- We can check the global variables and local variables of any function using locals() } built in
globals() } func.

(iii) print globals().keys()
o/p = [---------- ----------]

This will give all the global keys of the dictionary

(iv) func()
o/p = { }.

because we have not passed any variables/arguments so it will show empty { }

2) Functions within functions

eg:-(i) def hello (name = 'Jose'):
print 'The hello() function is executed'
def greet():
return 'This is inside the greet() function'
def welcome():
return '\t This is inside the welcome() function'
print greet()
print welcome()
print 'Now we are back inside the hello() function'

* hello()
o/p = The hello() function is executed
This is inside the greet() function
This is inside the welcome() function
Now we are back inside the hello() function

(ii) welcome()
o/p = not defined

this is due to scope welcome is not defined outside hello function, so the only functions were it is aware is inside hello() function

welcome() is not globally defined

(iii)
```
def hello(name == 'Jose'):
    def greet():
        return '\t This is inside the greet() function'
    def welcome():
        return '\t This is inside the welcome() function'
    if name == 'Jose':
        return greet    # here we returning the function and not greet(), welcome() the actual results.
    else:
        return welcome

x = hello()
x
o/p = <function - main - . greet>
    # here x points to greet.
    here hello has by default the name Jose and the condition is if it is Jose return greet ∴ x points to greet in hello() function.

print x()
o/p = This is inside the greet() function.
```

if we put the parantheses () after the function it gets executed and if we do not put parantheses it can be passed around and assigned to other variables

Scanned by CamScanner

3) Functions as Arguments

eg:- (i) def hello():
return 'Hi Jose!'

def other(func):
print 'Other code goes here!'
print func()

func is also a function and not an argument.

other()
o/p = error :: this expects an argument as func is a function and not an argument.

other(hello)
o/p = Other code goes here!
Hi Jose!

here other takes in argument and prints 'other code goes here'
Now hello is also an argument for function func which returns 'Hi Jose!'

- here we are passing function as objects and using it as another functions

(ii) def new_decorator(func):
def wrap_func():
print 'Code here, before executing the func'
func()
print 'Code here will execute after the func()'
return wrap_func

```
def func_needs_decorator():
    print 'This function needs a decorator!'
func_needs_decorator()
o/p = This function needs a decorator!
```

#here we are building decorator manually.

func_needs_decorator + new_decorator (func_needs_ ↓ function as a argument

#calls → func_needs_decorator()

```
o/p = Code here, before executing the func
      This function needs a decorator!
      Code here will execute after the func()
```

#here new_decorator wrapped the (func_needs_decorator) to modify func_needs_decorator

- '@' decorator symbol.

eg:- (iii) @new_decorator

```
def func_needs_decorator():
    print : This function needs a decorator!
func_needs_decorator()
o/p= Code here, before executing the func
     This function needs a decorator!
     Code here will execute after the func()
```

#here we are using @ to build a decorator using Python automatically