**Decorators**

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions.

**Functions Review**

​

**def** func():

**return** 1

func()

Out[2]:

1

**Scope Review**

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to.

s **=** 'Global Variable'

​

**def** check\_for\_locals():

locali**=**10

secloc1 **=** 20*# to check if any local values are there , you need to check inside the function where local functionality*

*#need to be known*

print(locali)

print(locals()['locali'])

print(locals())

check\_for\_locals()

*# print(globals())*

10

10

{'locali': 10, 'secloc1': 20}

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the locals() and globals() functions.

print(globals()['s'])

Global Variable

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

print(globals().keys())

dict\_keys(['\_\_name\_\_', '\_\_doc\_\_', '\_\_package\_\_', '\_\_loader\_\_', '\_\_spec\_\_', '\_\_builtin\_\_', '\_\_builtins\_\_', '\_ih', '\_oh', '\_dh', 'In', 'Out', 'get\_ipython', 'exit', 'quit', '\_', '\_\_', '\_\_\_', '\_i', '\_ii', '\_iii', '\_i1', 's', 'check\_for\_locals', '\_i2', '\_i3', '\_i4'])

Note how **s** is there, the Global Variable we defined as a string:

globals()['s']

Out[5]:

'Global Variable'

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

check\_for\_locals()

locals()['locali'] *#if we are trying to check outside the function the local variable will not be working hence error*

10

{'locali': 10, 'secloc1': 20}

**---------------------------------------------------------------------------**

**KeyError** Traceback (most recent call last)

**<ipython-input-3-1acf2fa5d691>** in <module>

1 check\_for\_locals**()**

**----> 2** locals**()['locali']**

**KeyError**: 'locali'

Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions.

**def** hello(name**=**'Jose'):

**return** 'Hello '**+**name

hello()

Out[3]:

'Hello Jose'

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

greet **=** hello

greet

Out[8]:

<function \_\_main\_\_.hello(name='Jose')>

greet()

Out[9]:

'Hello Jose'

So what happens when we delete the name **hello**?

**del** hello

hello()

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-9-a75d7781aaeb>** in <module>**()**

**----> 1** hello**()**

**NameError**: name 'hello' is not defined

greet()

Out[10]:

'Hello Jose'

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

**Functions within functions**

So we've seen how we can treat functions as objects, now let's see how we can define functions inside of other functions:

**def** hello(name**=**'Jose'):

print('The hello() function has been executed')

**def** greet():

**return** '\t 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()

The hello() function has been executed

This is inside the greet() function

This is inside the welcome() function

Now we are back inside the hello() function

welcome()

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-12-a401d7101853>** in <module>

**----> 1** welcome**()**

**NameError**: name 'welcome' is not defined

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions:

**Returning Functions**

**def** hello(name):

**def** greet():

**return** '\t This is inside the greet() function'

**def** welcome():

**return** "\t This is inside the welcome() function"

**if** name **==** 'Jose':

print(greet)

**return** greet

**else**:

**return** welcome

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

x **=** hello('Jose')

print(x)

<function hello.<locals>.greet at 0x060B19C0>

<function hello.<locals>.greet at 0x060B19C0>

print(x())

​

This is inside the greet() function

Great! Now we can see how x is pointing to the greet function inside of the hello function.

print(x())

This is inside the greet() function

Let's take a quick look at the code again.

In the if/else clause we are returning greet and welcome, not greet() and welcome().

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don’t put parentheses after it, then it can be passed around and can be assigned to other variables without executing it.

When we write x = hello(), hello() gets executed and because the name is Jose by default, the function greet is returned. If we change the statement to x = hello(name = "Sam") then the welcome function will be returned. We can also do print(hello()()) which outputs *This is inside the greet() function*.

**Functions as Arguments**

Now let's see how we can pass functions as arguments into other functions:

**def** hello():

**return** 'Hi Jose!'

​

​

**def** other(x):

print('Other code would go here')

print(x())

print(hello)

other(hello)

<function hello at 0x060B1B28>

Other code would go here

Hi Jose!

we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

**Decorators**

The idea behind Decorators comes from the Gang of Four Design Patterns book (so-called as there were four people involved in defining these design patterns). In this book numerous commonly occurring object oriented design patterns are presented. One of these design patterns is the Decorator design pattern. The Decorator pattern addresses the situation where it is necessary to add additional behaviour to specific objects. One way to add such additional behaviour is to decorate the objects created with types that provide the extra functionality. These decorators wrap the original element but present exactly the same interface to the user of that element. Thus the Decorator Design pattern extends the behaviour of an object without using sub classing. This decoration of an object is transparent to the decorators’ clients. In Python Decorators are functions that take another function (or other callable object such as a method) and return a third function representing the decorated behaviour.

**What Are Decorators?**

A Decorator is a piece of code, that is used to mark a callable object (such as a function, method, class or object) typically to enhance or modify its behaviour (potentially replacing it). It thus decorates the original behaviour. Decorators are in fact callable objects themselves and as such behave more like macros in other languages that can be applied to callable objects that then return a new callable object (typically a new function).

**Creating a Decorator**

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

**def** new\_decorator(func):

​

**def** wrap\_func():

print("Code would be here, before executing the func")

​

func()

​

print("Code here will execute after the func()")

​

**return** wrap\_func

​

**def** func\_needs\_decorator(): *#original function*

print("This function is in need of a Decorator")

func\_needs\_decorator()

This function is in need of a Decorator

*# Reassign func\_needs\_decorator*

x **=** new\_decorator(func\_needs\_decorator)

x()

Code would be here, before executing the func

This function is in need of a Decorator

Code here will execute after the func()

So what just happened here? A decorator simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

function decorators decorator is a function that accepts a function as parameter and return a function .it takes the result of a function modifies the result so decorators are useful to perform some additional processing required by a function

@new\_decorator

**def** func\_needs\_decorator():

print("This function is in need of a Decorator")

func\_needs\_decorator()

​

​

Code would be here, before executing the func

This function is in need of a Decorator

Code here will execute after the func()

You've now built a Decorator manually and then saw how we can use the @ symbol in Python to automate this and clean our code. You'll run into Decorators a lot if you begin using Python for Web Development, such as Flask or Django!\*\*

**Functions with Parameters**

Decorators can be applied to functions that take parameters; however the decorator function must also take these parameters as well.

**def** logger(func):

**def** inner(x, y):

print('calling ', func.\_\_name\_\_, 'with', x, 'and', y)

func(x, y)

print('returned from ', func.\_\_name\_\_)

**return** inner

@logger

**def** my\_func(x, y):

print(x, y)

my\_func(4, 5)

calling my\_func with 4 and 5

4 5

returned from my\_func

**Stacked Decorators**

Decorators can be stacked; that is more than one decorator can be applied to the same callable object. When this occurs, each function is wrapped inside another function

**def** decor(num):

print("First")

**def** inner():

print("Second")

value **=** num()

**return** value**+**2

**return** inner *# returning inner function with decorated or modified value*

​

​

​

*# how to use decorated*

*# take a function to which decorator is applied*

​

​

@decor

**def** num():

print("Third")

**return** 10

​

*# call num() function*

print(num())

First

Second

Third

12

**def** decor(fun):

**def** inner():

value **=** fun()

print(value)

**return** value**+**2

**return** inner *# returning inner function with decorated or modified value*

​

**def** decor1(fun):

**def** inner1():

value **=** fun()

print(value)

**return** value**+**10

**return** inner1 *# returning inner function with decorated or modified value*

​

​

*# how to use decorated*

*# take a function to which decorator is applied*

@decor

@decor1

**def** num():

**return** 10

​

*# call num() function*

print(num())

10

20

22

This means that the function num() is first passed into the decor1() function and wrapped by the inner1 function. This function is then returned from the decor1 decorator. The decor1\_wrapped is then passed into the decor() function which then wraps it inside the inner function; which is returned by the decor decorator.

**Parameterised Decorators**

Decorators can also take parameters however the syntax for such decorators is a little different; there is essentially an extra layer of indirection. The decorator function takes one or more parameters and returns a function that can use the parameter and takes the callable object that is being wrapped

**def** register(active): *#default parameter value*

**def** wrap(func):

**def** wrapper():

print('Calling ', func.\_\_name\_\_, ' decorator param', active)

**if** active:

func() *#calling of func2*

print('Called ', func.\_\_name\_\_)

**else**:

func() *#calling of func1*

print('Skipped ', func.\_\_name\_\_)

**return** wrapper

**return** wrap

​

​

@register(active **=** **False**) *#default value true is considered*

**def** func1():

print('func1')

​

@register(active**=True**) *#check for false value is explictly passes*

**def** func2():

print('func2')

func1()

print('-' **\*** 10)

func2()

Calling func2 decorator param True

func2

Calling func1 decorator param False

func1

Skipped func1

----------

Called func2

Class decorators, method decorators can be learnt as part of OOPS section

*#decorator function needs to return a function which can be called , here the invoking funtion func4 will be replaced by*

*#the returning value of the decorator here func3(func) will be replacing func4() so at the placeof func4 a callable function*

*#only should be there*

*# If we are trying to write a decorator with out any inner function, and we need the return value to be only the callable function*

*#in this example we have func4(i.e func) to be returned or func3 itself, if we use func4 as return , then we will not have*

*#the option of modifying the value it self so the purpose of decorator is lost*

*# if we want to return func3 then it should be exact as that of the func4 , but func4 does not take any input parameter*

*# that is the reason why in the avove example when the invoking function had paramters we made inner function to have*

*#parameters*

​

**def** func3(func):

value **=**func()

*# return func3 # here trying to return func3 , but it will be error as it is mismatching between func4 and func3*

*# return func # here we are returning func itself , but then there is no modifyied value*

**return** value *#here value is not callable hence will be error*

​

*#So that is the reason we writt inner function , which can be used for decorating purpose and for returning*

*#purpose and it maintains the exact signature of the func4() method*

​

*# Because our decorator is returning an int, and not a callable, it can not be called as a function.*

*# Remember, the decorator's return value replaces func4()*

​

@func3

**def** func4():

**return** 10

func4()

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-66-0e6ad11a93c1>** in <module>

**----> 1** func4**()**

**TypeError**: 'int' object is not callable

func3(func1)

Calling func1 decorator param True

func1

Called func1

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-53-d871dd1d95d6>** in <module>

**----> 1** func3**(**func1**)**

**<ipython-input-52-d0134ab5369a>** in func3**(func)**

1 **def** func3**(**func**):**

2 value **=**func1**()**

**----> 3 return** value**+10**

4

5 **@**func3

**TypeError**: unsupported operand type(s) for +: 'NoneType' and 'int'

​