**Python Decorators**

**Prerequisites for Decorators**

Before we learn about decorators, we must first understand a few basic things in Python.

**We must be comfortable with the fact that everything in Python is an object, including classes as well as functions**. Variables are simply identifiers bound to these objects.

Since functions are also objects, we can also pass functions as arguments to other functions:

def inc(x):

return x + 1

def operate(func, x):

result = func(x)

return result

print(operate(inc, 3))

**Output**

4

**In Python, we can also define a function inside a function**:

def print\_msg(message):

greeting = "Hello"

def printer():

print(greeting, message)

printer()

print\_msg("Python is awesome")

**Output**

Hello Python is awesome

 Let's modify our previous code:

**Output**

Hello Python is awesome

* Even though print\_msg() function is done executing, the returned inner printer() function can still access the message and greeting variables. Such a function is called a closure function.
* A closure is simply an inner function that remembers the values and variables in its enclosing scope even if the outer function is done executing.
* Python Decorators make an extensive use of closures.

We canreturn function as a value

def hello\_function():

def say\_hi():

return "Hi"

return say\_hi

hello = hello\_function()

hello()

Outut:

'Hi'

**Python Decorators**

A Python decorator is a function that takes in a function, adds some functionality to it and returns the original function.

Example 1:

def hello\_decorator(func):

# inner1 is a Wrapper function in

# which the argument is called

# inner function can access the outer local

# functions like in this case "func"

def inner1():

print("Hello, this is before function execution")

# calling the actual function now

# inside the wrapper function.

func()

print("This is after function execution")

return inner1

# defining a function, to be called inside wrapper

def function\_to\_be\_used():

print("This is inside the function !!")

# passing 'function\_to\_be\_used' inside the

# decorator to control its behavior

function\_to\_be\_used = hello\_decorator(function\_to\_be\_used)

# calling the function

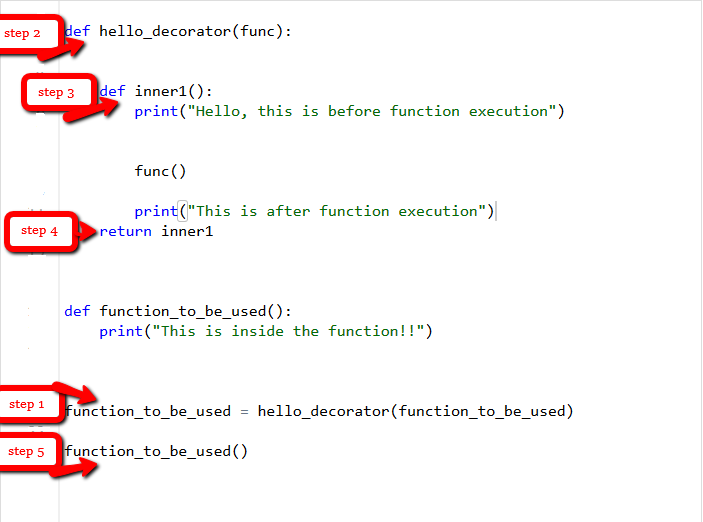
function\_to\_be\_used()

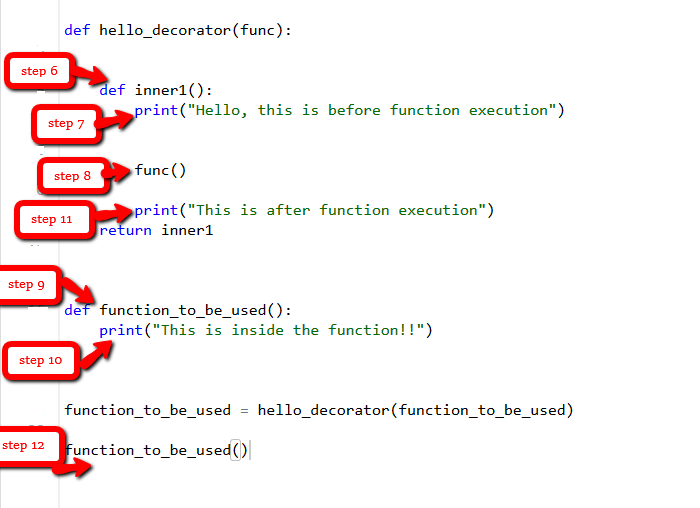
Output:

Hello, this is before function execution

This is inside the function !!

This is after function execution





Example 2:

def printer():

print("Hello, World!")

def display\_info(func):

def inner():

print("Executing",func.\_\_name\_\_,"function")

func()

print("Finished execution")

return inner

Let's call printer() normally first.

def printer():

print("Hello, World!")

def display\_info(func):

def inner():

print("Executing",func.\_\_name\_\_,"function")

func()

print("Finished execution")

return inner

printer()

**Output**

Hello, World!

Now, let's use the decorator function to run the same printer function.

def printer():

print("Hello, World!")

def display\_info(func):

def inner():

print("Executing",func.\_\_name\_\_,"function")

func()

print("Finished execution")

return inner

decorated\_func = display\_info(printer)

decorated\_func()

**Output**

Executing printer function

Hello, World!

Finished execution

In python, we have a much more elegant way of to achieve this functionality using the @ symbol.

def display\_info(func):

def inner():

print("Executing",func.\_\_name\_\_,"function")

func()

print("Finished execution")

return inner

@display\_info

def printer():

print("Hello, World!")

printer()

**Output**

Executing printer function

Hello, World!

Finished execution

@display\_info

def printer():

print("Hello, World!")

this code is same as: following

def printer():

print("Hello, World!")

data = display\_info(printer)

**Decorating Functions with Parameters**

Suppose we have a simple divide function.

def divide(a, b):

return a/b

We know this code will throw an exception if we pass the value for b as 0.

Let's make a decorator function called smart\_divide to prevent this.

def smart\_divide(func):

def inner(a, b):

print("Dividing", a, "by", b)

if b == 0:

print("Cannot divide by 0!")

return

return func(a, b)

return inner

@smart\_divide

def divide(a, b):

return a/b

value1 = divide(15, 3)

print(value1)

value2 = divide(5, 0)

print(value2)

**Output**

Dividing 15 by 3

5.0

Dividing 5 by 0

Cannot divide by 0!

None

**Chaining Decorators in Python**

In Python, a function can be decorated multiple times with different or the same decorator.

Here, are two decorator functions called star and percent. These functions print a series of star and percentage symbols before and after executing the function

def star(func):

def inner(arg):

print("\*" \* 30)

func(arg)

print("\*" \* 30)

return inner

def percent(func):

def inner(arg):

print("%" \* 30)

func(arg)

print("%" \* 30)

return inner

@star

@percent

def printer(msg):

print(msg)

printer("Decorators are wonderful")

**Output**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Decorators are wonderful

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

As you can see, these decorators are chained and they wrap the original function.

Here, we have first called the star function, and then the percent function. So the star function wraps the percent function which in turn wraps the printer function.

Example 2:

def decor1(func):

def inner():

x = func()

return x \* x # 20\* 20 = 400

return inner

def decor(func):

def inner():

x = func()

return 2 \* x # 20

return inner

@decor1

@decor

def num():

return 10

print(num()) # decor1(decor(num))

**with keyword arguments:**

|  |
| --- |
| def hello\_decorator(func):      def inner1(\*args, \*\*kwargs):            print("before Execution")            # getting the returned value          returned\_value = func(\*args, \*\*kwargs)          print("after Execution")            # returning the value to the original frame          return returned\_value        return inner1      # adding decorator to the function  @hello\_decorator  def sum\_two\_numbers(a, b):      print("Inside the function")      return a + b    a, b = 1, 2    # getting the value through return of the function  print("Sum =", sum\_two\_numbers(a, b)) |

**Output:**

before Execution

Inside the function

after Execution

Sum = 3