**DECORATORS**

Python decorators allow you to change the behavior of a function without modifying the function itself.

**Here are the building blocks used to create Python decorators**

To get a better understanding of how decorators work, you should understand a few concepts first.

1. A function is an object. Because of that, a function can be assigned to a variable. The function can be accessed from that variable.

def my\_function():

print('I am a function.')

# Assign the function to a variable without parenthesis. We don't want to execute the function.

description = my\_function

# Accessing the function from the variable I assigned it to.

print(description())

# Output

'I am a function.'

1. A function can be nested within another function.

def outer\_function():

def inner\_function():

print('I came from the inner function.')

# Executing the inner function inside the outer function.

inner\_function()

outer\_function()

# Output

I came from the inner function.

Note that the inner\_function is not available outside the outer\_function. If I try to execute the inner\_function outside of the outer\_function I receive a NameError exception.

inner\_function()

Traceback (most recent call last):

File "/tmp/my\_script.py", line 9, in <module>

inner\_function()

NameError: name 'inner\_function' is not defined

3.  Since a function can be nested inside another function it can also be returned.

def outer\_function():

'''Assign task to student'''

task = 'Read Python book chapter 3.'

def inner\_function():

print(task)

return inner\_function

homework = outer\_function()

homework()

# Output

'Read Python book chapter 5.'

4.  A function can be passed to another function as an argument.

def friendly\_reminder(func):

'''Reminder for husband'''

func()

print('Don\'t forget to bring your wallet!')

def action():

print('I am going to the store buy you something nice.')

# Calling the friendly\_reminder function with the action function used as an argument.

friendly\_reminder(action)

# Output

I am going to the store buy you something nice.

Don't forget to bring your wallet!

**How to Create a Python Decorator**

To create a decorator function in Python, I create an outer function that takes a function as an argument. There is also an inner function that wraps around the decorated function.

Here is the syntax for a basic Python decorator:

def my\_decorator\_func(func):

def wrapper\_func():

# Do something before the function.

func()

# Do something after the function.

return wrapper\_func

def my\_fun():

pass

A Python decorator is a function that takes in a function and returns it by adding some functionality.

Basically, a decorator takes in a function, adds some functionality and returns it.

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

def ordinary():

print("I am ordinary")

Here, we have created two functions:

* ordinary() that prints "I am ordinary"
* make\_pretty() that takes a function as its argument and has a nested function named inner(), and returns the inner function.

We are calling the ordinary() function normally, so we get the output "I am ordinary". Now, let's call it using the decorator function.

def make\_pretty(func):

# define the inner function

def inner():

# add some additional behavior to decorated function

print("I got decorated")

# call original function

func()

# return the inner function

return inner

# define ordinary function

def ordinary():

print("I am ordinary")

# decorate the ordinary function

decorated\_func = make\_pretty(ordinary)

# call the decorated function

decorated\_func()

**Output**

I got decorated

I am ordinary

In the example shown above, make\_pretty() is a decorator. Notice the code,

decorated\_func = make\_pretty(ordinary)

* We are now passing the ordinary() function as the argument to the make\_pretty().
* The make\_pretty() function returns the inner function, and it is now assigned to the decorated\_func variable.

decorated\_func()

Here, we are actually calling the inner() function, where we are printing

### @ Symbol With Decorator

Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol. For example,

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

@make\_pretty

def ordinary():

print("I am ordinary")

ordinary()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I got decorated

I am ordinary

Here, the ordinary() function is decorated with the make\_pretty() decorator using the @make\_pretty syntax, which is equivalent to calling ordinary = make\_pretty(ordinary).

## Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters.

Decorators can have arguments passed to them. To add arguments to decorators I add \*argsand*\**\*kwargs to the inner functions.

* \***args** will take an unlimited number of arguments of any type, such as 10, True, or 'Brandon'.
* \*\***kwargs** will take an unlimited number of keyword arguments, such as count=99, is\_authenticated=True, or name='Brandon'.

Here is a decorator with arguments:

def my\_decorator\_func(func):

def wrapper\_func(\*args, \*\*kwargs):

# Do something before the function.

func(\*args, \*\*kwargs)

# Do something after the function.

return wrapper\_func

@my\_decorator\_func

def my\_func(my\_arg):

'''Example docstring for function'''

pass

def divide(a, b):

return a/b

This function has two parameters, a and b. We know it will give an error if we pass in b as **0**.

Now let's make a decorator to check for this case that will cause the error.

def smart\_divide(func):

def inner(a, b):

print("I am going to divide", a, "and", b)

if b == 0:

print("Whoops! cannot divide")

return

return func(a, b)

return inner

@smart\_divide

def divide(a, b):

print(a/b)

divide(2,5)

divide(2,0)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I am going to divide 2 and 5

0.4

I am going to divide 2 and 0

Whoops! cannot divide

Here, when we call the divide() function with the arguments **(2,5)**, the inner() function defined in the smart\_divide() decorator is called instead.

This inner() function calls the original divide() function with the arguments **2** and **5** and returns the result, which is **0.4**.

Similarly, When we call the divide() function with the arguments (**2,0)**, the inner() function checks that b is equal to **0** and prints an error message before returning None.