Contents

[Decorators 2](#_Toc15479329)

[Python Decorators [C.M.S] 3](#_Toc15479330)

[# Logger Example 4](#_Toc15479331)

[# Timer Example 4](#_Toc15479332)

[What does functools.wraps do? 5](#_Toc15479333)

[Example : using functools.wraps to preserve information of our function 7](#_Toc15479334)

[# Chaining Decorators in Python 7](#_Toc15479335)

[Decorators with arguments [C.M.S] 8](#_Toc15479336)

[programiz.com 9](#_Toc15479337)

[# A FUNCTION THAT DOESN'T RETURN ANYTHING 9](#_Toc15479338)

[# 1st EXAMPLE: A FUNCTION THAT DOESN’T RETURN A VALUE 9](#_Toc15479339)

[# 2nd : A PRACTICAL EXAMPLE OF DECORATOR, DIVISION BY ZERO NOT ALLOWED 10](#_Toc15479340)

[# 3rd : CHAINING DECORATORS IN PYTHON 10](#_Toc15479341)

[Amuls Academy 11](#_Toc15479342)

[# 1st EXAMPLE : A FUNCTION THAT RETURNS A VALUE 11](#_Toc15479343)

[# 2nd EXAMPLE : CHAINING DECORATORS 11](#_Toc15479344)

[# 3rd EXAMPLE : DECORATORS WITH ARGUMENTS 12](#_Toc15479345)

[# 4th EXAMPLE 12](#_Toc15479346)

# [Decorators](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/)

Decorator is a **callable object** (callable means can be called like function):

* Decorators avoid duplication of code
* Decorators allow programmers to extend the behavior of function or class without having to modify the function itself.

**In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.**

in a nutshell, a decorator is formed of three parts:

1. Takes a function as a parameter.
2. Add functionality to that function.
3. Returns another function that has extended functionality of that function within it.

# Python Decorators [C.M.S]

def decorator\_fun(original\_func):

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

print("wrapper executed this before{}" .format(original\_func.\_\_name\_\_))

return original\_func(\*args, \*\*kwargs)

return wrapper\_func

class decorator\_class:

def \_\_init\_\_(self, original\_func):

self.original\_func = original\_func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print("Call method executed this before {}"

.format(self. original\_func.\_\_name\_\_))

return self.original\_func(\*args, \*\*kwargs)

@decorator\_class

def display\_info(name, age):

print("display\_info ran with arguments ({}, {})".format(name, age))

display\_info("basir", "payenda")

# Call method executed this before display\_info

# display\_info ran with arguments (basir, payenda)

@decorator\_fun

def display(name, age):

print("display\_info ran with arguments ({}, {})".format(name, age))

display("lorem", "Ipsum")

# wrapper executed this beforedisplay

# display\_info ran with arguments (lorem, Ipsum)

## # Logger Example

def logger\_func(original\_func):

import logging

logging.basicConfig(filename="{}.log".format(original\_func.\_\_name\_\_),

level= logging.INFO)

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

logging.info("logger\_func args : {}, {}".format(\*args))

return original\_func(\*args, \*\*kwargs)

return wrapper

@logger\_func

def display\_info(name, age):

print("display\_info ran with arguments {} & {}".format(name, age))

display\_info("Basir Payenda", 24)

## # Timer Example

def timer(original\_func):

import time

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

t1 = time.time()

time.sleep(1)

result = original\_func(\*args, \*\*kwargs)

t2 = time.time() - t1

print("{} function executed in {} seconds".format(original\_func.\_\_name\_\_, t2))

return result

return wrapper

@timer

def display\_timer\_info(name, age):

print("display\_info ran with arguments {} & {}".format(name, age))

display\_timer\_info("John Doe", 26)

# display\_info ran with arguments John Doe & 26

# display\_timer\_info function executed in 1.000624179840088 seconds

# [What does functools.wraps do?](https://stackoverflow.com/questions/308999/what-does-functools-wraps-do)

When you use a decorator, you're replacing one function with another. In other words, if you have a decorator

def logged(func):

def with\_logging(\*args, \*\*kwargs):

print(func.\_\_name\_\_ + " was called")

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

return with\_logging

then when you say

@logged

def f(x):

"""does some math"""

return x + x \* x

it's exactly the same as saying

def f(x):

"""does some math"""

return x + x \* x

f = logged(f)

and your function f is replaced with the function with\_logging. Unfortunately, this means that if you then say

print(f.\_\_name\_\_)

it will print with\_logging because that's the name of your new function. In fact, if you look at the docstring for f, it will be blank because with\_logging has no docstring, and so the docstring you wrote won't be there anymore. Also, if you look at the pydoc result for that function, it won't be listed as taking one argument x; instead it'll be listed as taking \*args and \*\*kwargs because that's what with\_logging takes.

If using a decorator always meant losing this information about a function, it would be a serious problem. That's why we have functools.wraps. This takes a function used in a decorator and adds the functionality of copying over the function name, docstring, arguments list, etc. And since wraps is itself a decorator, the following code does the correct thing:

from functools import wraps

def logged(func):

@wraps(func)

def with\_logging(\*args, \*\*kwargs):

print(func.\_\_name\_\_ + " was called")

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

return with\_logging

@logged

def f(x):

"""does some math"""

return x + x \* x

print(f.\_\_name\_\_) # prints 'f'

print(f.\_\_doc\_\_) # prints 'does some math'

## Example : using functools.wraps to preserve information of our function

## # Chaining Decorators in Python

from functools import wraps

def my\_logger(orig\_func):

import logging

logging.basicConfig(filename='{}.log'.format(orig\_func.\_\_name\_\_),

level=logging.INFO)

@wraps(orig\_func)

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

logging.info(

'Ran with args: {}, and kwargs: {}'.format(args, kwargs))

return orig\_func(\*args, \*\*kwargs)

return wrapper

def my\_timer(orig\_func):

import time

@wraps(orig\_func)

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

t1 = time.time()

result = orig\_func(\*args, \*\*kwargs)

t2 = time.time() - t1

print('{} ran in: {} sec'.format(orig\_func.\_\_name\_\_, t2))

return result

return wrapper

@my\_logger

@my\_timer

def display(name, age):

"""Showing information about logged values """

print("Values logged are : {} & {}".format(name, age))

# display = my\_logger(my\_timer(display))

display("Nooruddin", 19)

print(display.\_\_name\_\_) # display

print(display.\_\_doc\_\_) # Showing information about logged values

# Decorators with arguments [C.M.S]

def prefix\_decorator(prefix):

def decorator\_func(original\_func):

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

print(prefix + "Executed before {}"

.format(original\_func.\_\_name\_\_))

result = original\_func(\*args, \*\*kwargs)

print(prefix + "Executed after {}"

.format(original\_func.\_\_name\_\_))

return result

return wrapper

return decorator\_func

@prefix\_decorator("Info: ")

def display(name, age):

print("New Personnel: {0}, {1} years old".format(name, age))

display("John Doe", 33)

# programiz.com

# <https://stackoverflow.com/questions/57122612/does-return-result-in-this-code-affect-on-the-result-of-this-decorator/57122741#57122741>

### # A FUNCTION THAT DOESN'T RETURN ANYTHING

def decorate\_me(func):

def wrapper():

print('I decorate ordinary!')

func()

return wrapper

@decorate\_me

def ordinary():

print('I am ordinary.')

ordinary()

# I decorate ordinary!

# I am ordinary.

### # 1st EXAMPLE: A FUNCTION THAT DOESN’T RETURN A VALUE

def decorate\_me(func):

def wrapper():

print('I decorate ordinary!')

return func() # returning

return wrapper

@decorate\_me

def ordinary():

print('I am ordinary.')

return 'Now I must be decorated' # returning

result = ordinary()

print(result) # printing returned result

# I decorate ordinary!

# I am ordinary.

# Now I must be decorated

### # 2nd : A PRACTICAL EXAMPLE OF DECORATOR, DIVISION BY ZERO NOT ALLOWED

def smart\_divide(func):

def inner(a,b):

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

if b == 0:

print("Whoops! cannot divide")

elif a < b:

a, b = b, a

return func(a,b)

return inner

@smart\_divide

def divide(a,b):

return a/b

result = divide(5, 15)

print('The result is', round(result))

# I am going to divide 5 and 15

# The result is 3

### # 3rd : CHAINING DECORATORS IN PYTHON

def star(func):

def inner(\*args, \*\*kwargs):

print("\*" \* 30)

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

print("\*" \* 30)

return inner

def percent(func):

Output:

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

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

Hello

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

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

def inner(\*args, \*\*kwargs):

print("%" \* 30)

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

print("%" \* 30)

return inner

@star

@percent

def printer(msg):

print(msg)

printer("Hello")

# Amuls Academy

### # 1st EXAMPLE : A FUNCTION THAT RETURNS A VALUE

def decorate\_msg\_upper(original\_func):

def inner(msg):

""" Decorating inner function! do something with returned value """

result = original\_func(msg)

return result.upper()

return inner

@decorate\_msg\_upper

def printer(msg):

return msg

res = printer('Hello World')

print(res)

# HELLO WORLD

### # 2nd EXAMPLE : CHAINING DECORATORS

def first\_deco(func):

def inner():

res = func()

return ' First ' + func() + ' First '

return inner

def second\_deco(func):

def wrapper():

res = func()

return 'Second' + func() + ' Second '

return wrapper

@second\_deco

@first\_deco

def ordinary():

return 'Hello World!'

# ordinary = second\_deco(first\_deco(ordinary))

print(ordinary())

# Second First Hello World! First Second

### # 3rd EXAMPLE : DECORATORS WITH ARGUMENTS

def outer(msg):

def lang\_name(func):

def inner():

res = func() + msg

return res

return inner

return lang\_name

@outer("Python")

def ordinary():

return 'Hello '

print(ordinary())

### # 4th EXAMPLE

def div\_decorator(func):

def inner(\*args):

arg\_list = list(args[1:])

for arg in arg\_list:

arg = int(arg)

if arg == 0:

return 'Cannot divide by zero!'

res = func(\*args)

return res

return inner

@div\_decorator

def div(a, b):

return a/b

@div\_decorator

def div2(a, b, c):

return a/b/c

print(div(10, 5))

print(div(10, 0))

print(div2(10, 5, 0))

### # 4th EXAMPLE : DECORATORS ON CLASS METHODS

def check\_name(func):

def wrapper(name\_ref): # self here is 'name\_ref'

if name\_ref.name == 'basir payenda': # 'name\_ref.name' == self.name

print('Hey my name is basir payenda either.')

else:

print('Ohhh I see, different names.')

return wrapper

class printing:

def \_\_init\_\_(self, name):

self.name = name

@check\_name

def intro(self):

print('My name is', self.name)

p1 = printing('basir payenda')

p1.intro()

## CLASS DECORATOR

### # 1st Example

class UpperDecorator:

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self):

res = self.func()

return res.upper()

@UpperDecorator

def message\_app():

return 'Hello World'

res = message\_app()

print(res)

## # 2nd Example:

# The parameters of the decorated function are passed to the \_\_call\_\_ method, not to the constructor \_\_init\_\_:

class UpperDecorator:

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

res = self.func(\*args, \*\*kwargs)

return res.upper()

@UpperDecorator

def message\_app(msg):

return msg

res = message\_app('Hi')

print(res)