**Decorators**

martes, 5 de diciembre de 2023

7:19 p. m.

Datacamp ref.: <https://www.datacamp.com/tutorial/decorators-python>

GeeksforGeeks ref.: <https://www.geeksforgeeks.org/decorators-in-python/>

Patrick Loeber ref: [Decorators in Python - Advanced Python 13 - Programming Tutorial](https://www.youtube.com/watch?v=FXUUSfJO_J4)

Corey Schafer ref: [Python Tutorial: Decorators - Dynamically Alter The Functionality Of Your Functions](https://www.youtube.com/watch?v=FsAPt_9Bf3U&list=WL&index=12&t=10s)

Introduction

Decorators in Python are a high-level concept that allows you to modify the behavior of functions or methods. They're functions themselves that take another function as an argument and usually return a new function that extends or modifies the behavior of the original function.

The key idea is that they provide a clean and elegant way to modify the functions' or methods' behaviors without directly altering their code. This is crucial for keeping code modular, readable, and reusable.

Here’s a breakdown of how they work:

* Functions as First-Class Objects: In Python, functions are treated as first-class objects, meaning they can be passed around as arguments to other functions, assigned to variables, and returned from other functions.

* Decorator Syntax: Decorators use the @decorator\_name syntax above a function declaration to apply the decorator to that function.

* Wrapper Functions: Decorators typically define a wrapper function inside them. This wrapper function often adds some extra functionality before or after the original function is called.

* Multiple Decorators: You can stack multiple decorators on top of a single function, which allows for a modular and composable approach to modifying functions.

* Common Use Cases: They're used for tasks like logging, timing functions, caching, authorization, and more. They keep these functionalities separate from the core logic of the functions.

For instance, if you have a function that calculates something, you can use a decorator to log when that function is called or to time how long it takes to execute. This way, the core functionality remains untouched while these additional features are added seamlessly.

Now, Understanding a few core concepts will set you up nicely to work with decorators:

1. First-Class Functions: In Python, functions are treated as first-class citizens. This means you can pass them as arguments, return them from other functions, and assign them to variables.

1. Closures: A closure is an inner function that captures and remembers the enclosing function's variables. Decorators often use closures to define wrapper functions that modify the behavior of other functions.

1. Function Signatures and Args/kwargs: Knowing how functions can take different arguments and how decorators handle these variations is crucial. This includes understanding \*args and \*\*kwargs to handle arbitrary numbers of positional and keyword arguments.

1. @decorator Syntax: The @decorator syntax is the clean way to apply a decorator to a function. This syntactic sugar makes it easy to enhance functions without cluttering their code.

1. Decorator Chaining: Being able to stack multiple decorators on a single function is a powerful feature. Understanding the order of execution when multiple decorators are used is important.

1. Use Cases: Knowing common use cases like logging, caching, and authorization helps in recognizing when and where to apply decorators effectively.

Before moving over to actually review Decorators, a concept I haven't formally reviewed must be clarified, and that is 'Closures'.

Imagine you have a function within another function. When the inner function accesses variables from the outer function, even after the outer function has finished executing, that inner function "remembers" the environment in which it was created.

def outer\_function(x):

    def inner\_function(y):

        return x + y

    return inner\_function

closure\_example = outer\_function(5) # Here, the 5 is passed as 'x' but 'y' hasn't been defined yet

result = closure\_example(3) # Here, 'y' is defined as 3.

print(result)

Custom Decorators

Next is the functioning of an actual decorator.

def log\_function(func):

    def wrapper(\*args, \*\*kwargs):

        print(f'Calling {func.\_\_name\_\_} with args: {args} and kwargs: {kwargs}')

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

    return wrapper

@log\_function

def greet(name):

    return f'Hello, {name}!'

result = greet('David')

print(result)

printout:

Calling greet with args: ('David',) and kwargs: {}

Hello, David!

The basic structure of the decorator, to simplify is like this:

def outer\_function(argument\_function):

    def inner\_function(\*args):

        print('The inner function is doing something...')

        return(argument\_function(\*args))

    return inner\_function

@outer\_function

def argument\_function(argument):

    return f'The arugment function is doing something with the {argument}!'

result = argument\_function('\*Argument\*')

print(result)

printout:

The inner function is doing something...

The argugment function is doing something with the \*Argument\*!

JHere is a basic application of a timer decorator:

def timer\_func(func):

    def wrapper(\*args):

        s = timer()

        func()

        e = timer()-s

        print(f"It took {e:.2f}s to execute '{func.\_\_name\_\_}()' function")

    return wrapper

@timer\_func

def do\_this():

    time.sleep(2.54)

do\_this()

printout:

It took 2.54s to execute 'do\_this()' function

Decorators could be stacked to modify behaviors more than one at a time:

def uppercase\_decorator(func):

    def wrapper(\*args):

        result = func(\*args)

        return result.upper()

    return wrapper

def greeting\_decorator(func):

    def wrapper(\*args):

        result = func(\*args)

        return f"Greeting, {result}"

    return wrapper

@uppercase\_decorator

@greeting\_decorator

def greet(name):

    return f"Hello, {name}!"

result = greet('David')

print(result)

printout:

GREETING, HELLO, DAVID!

And finally, a more sophisticated application of decorators:

def repeat(n\_times):

    def decorator(func):

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

            for \_ in range(n\_times):

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

        return wrapper

    return decorator

@repeat(n\_times=3)

def greet(name):

    return print(f"Hello, {name}!")

greet("Alice")

printout:

Hello Alice!

Hello Alice!

Hello Alice!

Patrick Loeber Decorators Tutorial

Continue here: [Decorators in Python - Advanced Python 13 - Programming Tutorial](https://www.youtube.com/watch?v=FXUUSfJO_J4)

The use of one decorator from the functools module is quite useful when capturing info about custom decorators, since if not used, python would misinterpret the functions naming.

import functools

def start\_end\_decorator(func):

    @functools.wraps(func)

    def wrapper(\*args, \*\*kwargs):

        print('Start')

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

        print('End')

        return result

    return wrapper

@start\_end\_decorator

def add5(x):

    return x + 5

print(help(add5))

print(add5.\_\_name\_\_)    # add5

Printout:

Help on function add5 in module \_\_main\_\_:

add5(x)

None

add5

If not used the @functools.wraps It'll print that the name of the 'add5' function is actually 'wrapper', this might come handy when debugging with decorators. In more formal terms, this decorator 'maintains the integrity of the metadata of the function decorated'

Decorators can also be implemented as Classes and also adds functionalities and behaviors of that class to a function or to another class.

class CountCalls:

    def \_\_init\_\_(self, func):

        self.func = func

        self.num\_calls = 0

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

        self.num\_calls += 1

        print(f'This have been executed {self.num\_calls} times')

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

@CountCalls

def say\_hello():

    print('Hello!')

say\_hello()

say\_hello()

Printout:

This have been executed 1 times

Hello!

This have been executed 2 times

Hello!

Corey Schafer Decorators Tutorial

Continue here: [Python Tutorial: Decorators - Dynamically Alter The Functionality Of Your Functions](https://www.youtube.com/watch?v=FsAPt_9Bf3U&list=WL&index=9&t=11s)

During the review of this tutorial I understood why in decorators the return statement left the 'wrapper' function without execution (i.e.: without the parenthesis at the end of the function name), and according to Corey this is to left the function wrapped awaiting to be called for being executed. This means that by wrapping a function with a decorated, in the moment the function is decorated and called, it returned he wrapper awaiting to be called and when called, it performs something but at some point within the wrapper, the decorated function is called.

That was why the Decorators returns its wrappers without actually calling them.

This concept is more related to closures than to decorator per se.