https://www.datacamp.com/community/tutorials/decorators-python

https://towardsdatascience.com/how-to-use-decorators-in-python-by-example-b398328163b

Python Decorators:

====== =========

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.

Decorators are usually called before the definition of a function you want to decorate.

here we'll show to the user how they can use decorators in their Python functions.

Functions in Python are first class citizens. This means that they support operations

such as being passed as an argument, returned from a function, modified, and assigned to a variable. This is a fundamental concept to understand before we delve into creating Python decorators.

Assigning Functions to Variables:

To kick us off we create a function that will add one to a number whenever it is called. We'll then assign the function to a variable and use this variable to call the function.

Example:

```
def plus_one(number):
    return number + 1

add_one = plus_one

print(add_one(5))
```

Output:

6

Defining Functions Inside other Functions:

Next, we'll illustrate how you can define a function inside another function in Python. Stay with me, we'll soon find out how all this is relevant in creating and understanding decorators in Python.

Example:

```
def plus_one(number):
    def add_one(number):
        return number + 1
    result = add_one(number)
    return result
print(plus_one(4))
```

Output:

5

Passing Functions as Arguments to other Functions:

Functions can also be passed as parameters to other functions. Let's illustrate that below.

Example:

```
def plus_one(number):
    return number + 1

def function_call(function):
    number_to_add = 5
    return function(number_to_add)

x = function_call(plus_one)
print(x)
```

Output:

6

Functions Returning other Functions:

A function can also generate another function. We'll show that below using an example.

Example:

```
def hello_function():
```

```
    def say_hi():
        return "Hi"
    return say_hi
hello = hello_function()
print(hello())
```

Output:

'Hi'

Nested Functions have access to the Enclosing Function's Variable Scope:

Python allows a nested function to access the outer scope of the enclosing function.

This is a critical concept in decorators -- this pattern is known as a Closure.

Example:

```
def print_message(message):
    "Enclosong Function"
    def message_sender():
        "Nested Function"
        print(message)
    message_sender()
print_message("Some random message")
```

Output:

Some random message

Creating Decorators:

With these prerequisites out of the way, let's go ahead and create a simple decorator that will convert a sentence to uppercase.

We do this by defining a wrapper inside an enclosed function.

As you can see it very similar to the function inside another function that we created earlier.

Example:

```
def uppercase_decorator(function):
    def  wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper
```

--->> Our decorator function takes a function as an argument, So we need to define a function and pass it to our decorator.

--->> We learned earlier that we could assign a function to a variable. We'll use that trick to call our decorator function.

Example1:

```
def say_hi():
    return 'hello there'
decorate = uppercase_decorator(say_hi)
print(decorate())
```

output:

'HELLO THERE'

---->> However, Python provides a much easier way for us to apply decorators.

---->> We simply use the @ symbol before the function we'd like to decorate.

Let's show that in practice below.

Example2:

```
@uppercase_decorator
def say_hi():
    return 'hello there'
say_hi()
```

Output:

'HELLO THERE'

Consider the following example2:

1. add_together function

The simple add_together function takes 2 integers as arguments and returns the summation of the 2 integer values passed.

code:

```
def add_together(a,b):
    return a + b
print(add_together(4,6))
```

Output:

10

1.1. Extending the functionality of add_together with a decorator

Lets propose the following scenario. We would like add_together to be able to take a list of 2 element tuples as its argument and return a list of integers which represents the summation of their values. We can achieve this through the use of a decorator.

Firstly, we give the decorator a sensible name which intimates what its intended purpose is. Here the decorator, called decorator_list is simply a function that takes another function as its argument. This function has been named 'fnc' in the arguments list to decorator_list.

Inside the decoratored function, we define a local function called inner. The inner function takes a list of 2-element tuples as its argument. The inner function loops through this list of 2 element tuples, and for each tuple applies the original function, add_together, with index position 0 of the tuple being the argument to a in add_together and index position 1 being the argument to b in add_together. The inner function returns a list of these values summed up. The decorator_list function finally returns the inner function.

Now, let's apply this logic and see how the decorator function works.

To apply the decorator, we use the syntax @, followed by the function name of the decorator above the function that is being decorated. This is syntactically the same as:

Example:

```
add_together = decorator_list(add_together)
```

Here, we pass the function add_together to the decorator_list function, and this function returns the inner function which we assign to the variable name add_together. As

add_together now points to the inner function, which expects a list of 2 element tuples, we can call add_together with a list of tuples as the argument.

The standard output to the console shows that we have now extended the functionality of the original add_together function, so that it can now take a list of tuples.

Code:

```
def decorator_list(fnc):
    def inner(list_of_tuples):
        return [fnc(val[0], val[1]) for val in list_of_tuples]
    return inner


@decorator_list
def add_together(a, b):
    return a + b
print(add_together([(1, 3), (3, 17), (5, 5), (6, 7)]))
# add_together = decorator_list(add_together)
```

Output:

```
[4, 20, 10, 13]
```

Example3:

```
def entryExit(f):
    def new_f():
        print("Entering into ",f.__name__)
        f()
        print("Exited from", f.__name__)
    return new_f

@entryExit
def func1():
```

```
    print('inside func1')


@entryExit
def func2():
    print('inside func2')


func1()
print("============")
func2()
print(func2.__name__)
```

Output:

```
Entering into  func1
inside func1
Exited from func1
============
Entering into  func2
inside func2
Exited from func2
```

Applying Multiple Decorators to a Single Function:

We can use multiple decorators to a single function. However, the decorators will be applied in the order that we've called them. Below we'll define another decorator that splits the sentence into a list. We'll then apply the uppercase_decorator and split_string decorator to a single function.

Example:

```
def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
```

```
        return splitted_string
    return wrapper
```

or

```
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'
say_hi()
```

Output:

```
['HELLO', 'THERE']
```

From the above output, we notice that the application of decorators is from the bottom up. Had we interchanged the order, we'd have seen an error since lists don't have an upper attribute. The sentence has first been converted to uppercase and then split into a list.

In order to solve this challenge Python provides a functools.wraps decorator.

This decorator copies the lost metadata from the undecorated function to the decorated closure.

Let's show how we'd do that.

code:

```
import functools
def uppercase_decorator(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
    return wrapper

@uppercase_decorator
def say_hi():
```

```
    "This will say hi"
    return 'hello there'
say_hi()
```

Output:

'HELLO THERE'

When we check the say_hi metadata, we notice that it is now referring to the function's metadata and not the wrapper's metadata.

```
say_hi.__name__
```

Output:

'say_hi'

```
say_hi.__doc__
```

Output:

'This will say hi'

Note: It is advisable and good practice to always use functools.wraps when defining decorators. It will save you a lot of headache in debugging.

Python Decorators Summary:

Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated. Using decorators in Python also ensures that your code is DRY(Don't Repeat Yourself). Decorators have several use cases such as:

--->> Authorization in Python frameworks such as Flask and Django

--->> Logging

--->> Measuring execution time

--->> Synchronization