Decorator Cheatsheet

by Shirin Yamani

Intuition

What are Decorators?

By definition, a decorator is a function that takes another **function as its input argument, and returns yet another function,** without explicitly modifying it (i.e. add some more functionalities to the nature of the first input function). A funny way to better understand what are decorators is to consider it as a bride!👰 Imagine a bride without any makeup comes in, and then the function put some makeup and accessories on her: 💄–>👰

What is the intuition beyond Decorators?

Before using a decorator, you need to get familiar with some concepts! The first one is a *wrapper* function! Let's first see the code look!

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        output = func(*args, **kwargs)
        return output
    return wrapper
```

Remember, earlier I mentioned decorator job is to add some more functionality to the input function(bride without makeup–>bride with makeup) this is all done by a *wrapper* function! What is actually happening inside the *wrapper* is adding some more job to the original input function(func) then calling it(func(*args, **kwargs)) and saving its output in(output) lastly, returning the output! Thus far, all the magic of adding some more use of our original function is done inside the *wrapper* function! If you think of a wrapper definition itself, you come up with the same concept! Like you wrap a gift! So you are decorating the gift, i.e. you are adding some more stuff to the original 🎁, right?!

How to use Decorators?

We have a really nice syntactic sugar as a way of using decorators! See code below!

```
@my_decorator
def say_hi(name):
    print("Good Morning Shirin!")
```

```
say_hi('shirin')
```

```
Good Morning Shirin!
```

Simple Decorators

The "@" syntactic sugar means exactly same as when you write the definition of the decorator function by yourself! Look at the code below:

```
def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        output = func(*args, **kwargs)
        return output
    return wrapper
```

```
#Approach 1: Using Syntactic Suger @
@my_decorator
def greet(name):
    print(f'hi {name}!')
```

```
greet('Shirin')
```

```
hi Shirin!
```

```
#Approach 2: Using definition of Decorator!
greet = my_decorator(greet)
```

```
greet('Alex')
```

```
hi Alex!
```

So we can either use *(Approach1 –> @)* or traditionally use the definition of it*(Approach2)*.

Who are you really?

Pay attention to "greet" **type** at code below;

```
greet
```

```
<function __main__.my_decorator.<locals>.wrapper(*args, **kwargs)>
```

As you see, once we apply decorator on "greet", all the nature and the name of the function is gone! Even if the function has documentation, once you apply the decorator, all of it will be lost! So how can we save the name of the greet? This has solved by a decorator from "functools" library; 😇

```
def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        output = func(*args, **kwargs)
        return output
    return wrapper
```

Now see the name has given back after applying;

```
greet
```

```
<function __main__.greet(name)>
```

Fancy Decorators

Why do we need Fancy Decorators?

Imagine we wanna build a decorator that repeats the output of its input function! If so, then we need to write the func(*args,**kwargs) twice, right?

```
def repeat(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        output = func(*args, **kwargs)
        return output
    return wrapper
```

```
@repeat
def greet(name):
    print(f'hi {name}!')
```

```
greet('shirin')
```

```
hi shirin!
hi shirin!
```

But what if we want to repeat it 4 or 10 time? Obviously, re-writing the "func(*args,**kwargs)" isn't a good way, right? So that's where Fancy Decorators come in!😇 So what we really need is a Decorator which can take input,like "number of repeat". By defining the number we basically tell the decorator how many times it needs to repeat the function so far! To implement such structure, we need to modify the current block of our function in a way that it has that "number of repeat" signal! Ya ask how? By having all the block of our current decorator as another function which has the elements that we want so far! This is called Decorator with Arguments!

```
def repeat(num_repeat): #enclosing for decorator
    def decorator(func): #Main decorator
        @functools.wraps(func)
        def wrapper(*args,**kwargs):#wraps our function
            for _ in range(num_repeat):
                output = func(*args,**kwargs)
            return output
        return wrapper
    return decorator
```

Decorator w/ Argument
Apply with argument

```
@repeat(num_repeat=3)
def greet(name):
    print(f'hi {name}!')
```

```
greet("shirin")
```

```
hi shirin!
hi shirin!
hi shirin!
```

Fancy Decorators

Fancy Decorators optionally take arguments?

Imagine we want the original "repeat" function in column 3 repeat the output twice without stating anything in front of it! In other words, if we apply @repeat on a function it automatically repeat the output twice*(Situation1)*, whereas if we define the num-repeat then it do it n times*(Situation2)*!

```
#Situation 1: without argument in front of @repeat but still want the output repeated twice!
@repeat
def greet(name):
    print(f'hi {name}!')   #output: hi shirin , hi shirin
#-------------------------------------------------------------------------------------------
#Situation 2: optional argument (num_repeated=3); output gotta repeated 3 times!
@repeat(num_repeat=3)
def greet(name):
    print(f'hi {name}!')  #output: hi shirin, hi shirin, hi shirin
```

If you pay attention to the code above, the difference between the *Situation1* vs. the original @repeat in column 3 is that in *Situation1* the output function **still** will be repeated twice without stating anything in front of it! So it means that we want our decorator to behave one way when we do not give it any input, whereas behave some other way when we input sth to it! It sounds like we want the decorator to **optionally** get input argument! Code below describes how we implement such magic!

```
def repeat(_func=None, *, num_repeat=2):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(num_repeat):
                output = func(*args,** kwargs)
            return output
        return wrapper
    if _func is None:
        return decorator #means there are more inputs than _func!
    else:
        return decorator(_func) #means we wanna get the wrapper!
```

Situation 2
Situation 1

What each situation and the code mean?

As the code suggests, whenever the func is not None *i.e. (Situation1 –> func=greet)* the black box in code will be executed which basically means:

```
greet = repeat(greet) #_func = greet
```

Whereas whenever the func is None *(Situation2 func=None)* the red box in code will be executed which means:

```
greet = repeat (_func= None, num_repeat=3)(greet) #_func = None
```

where the decorator applies on greet then it output a new greet! See the comments in code!