# #Information about decorators#

## Short definition

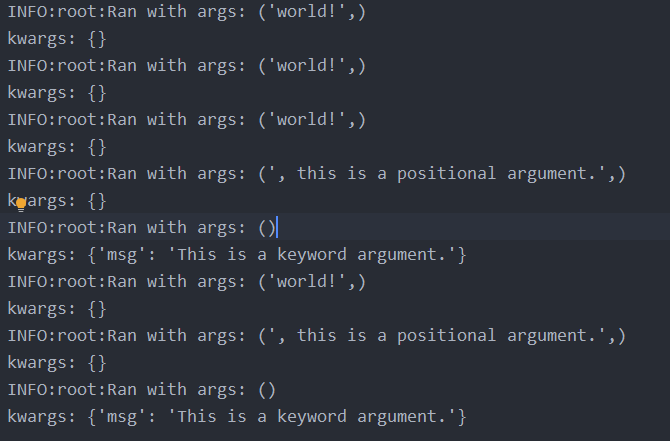
Decorators are a way to dynamically add to the functionality of you functions. So for example, if you wanted to log information when a function is run, you could use a decorator to add this functionality without modifying the source of your function

Decorators in python are a bit trippy but if you pay atention they will make sense. Here below we are describing some basic information about them and showing you some real world examples where they could be implemented. The information from this document was taken from Corey Schafer’s Youtube tutorial on decorators (<https://www.youtube.com/watch?v=FsAPt_9Bf3U&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=37>)

## Function decorators:

def decorator\_function(original\_function): # We pass in original\_function as a arg  
 def wrapper\_function(): # We define the inner function also called Wrapper function  
 print('This was printed before') # Everything here gets executed before our original\_function  
 return original\_function() # After we execute what we wanted we then execute our original\_function  
 return wrapper\_function # We return wrapper\_function in an "unsolicited way"  
  
  
@decorator\_function # This is basically saying new\_function = decorator\_function(the\_func\_we\_want\_to\_be\_executed)  
def hello\_world():  
 print("Hello World")  
  
  
hello\_world() # In the end we simply call our hello\_world function which is basically decorator\_function(hello\_world)  
  
### Some real-world examples ###  
  
def logger\_function(function\_to\_be\_logged):  
 import logging  
 logging.basicConfig(filename=f'{function\_to\_be\_logged.\_\_name\_\_}.log', level=logging.INFO) # .\_\_name\_\_ gives us the name of the function  
  
 def wrapper(\*args, \*\*kwargs): # args = positional arguments , kwargs = keyword arguments (example below)  
 logging.info(  
 f'Ran with args: {args}\nkwargs: {kwargs}'  
 )  
 return function\_to\_be\_logged(\*args, \*\*kwargs)  
  
 return wrapper  
  
@logger\_function  
def hello\_world(msg):  
 print(f'Hello {msg}')  
  
hello\_world('world!')  
  
# Example of \*arg - positional argument  
hello\_world(', this is a positional argument.')  
# Example of \*\*kwarg - keyword argument  
hello\_world(msg='This is a keyword argument.')

Example output of our .log file :



def my\_timer(orig\_func):  
 import time # Import time module  
 def wrapper(\*args, \*\*kwargs):  
 """We make a variable called to\_be\_timed otherwise we have to run the function  
 and that would print hello + {msg} again."""  
 t1 = time.time() # Get’s the current time  
 to\_be\_timed = orig\_func(\*args, \*\*kwargs)   
 t2 = time.time() - t1 # t2 - t1 gives us the actual runtime of our function  
 print(f'{orig\_func.\_\_name\_\_} ran in : {t2} sec')  
 return to\_be\_timed # We return our newly made variable  
 return wrapper  
  
@my\_timer  
def hello\_world(msg):  
 print(f'Hello {msg}!')  
  
hello\_world('world')

## Chaining Decorators

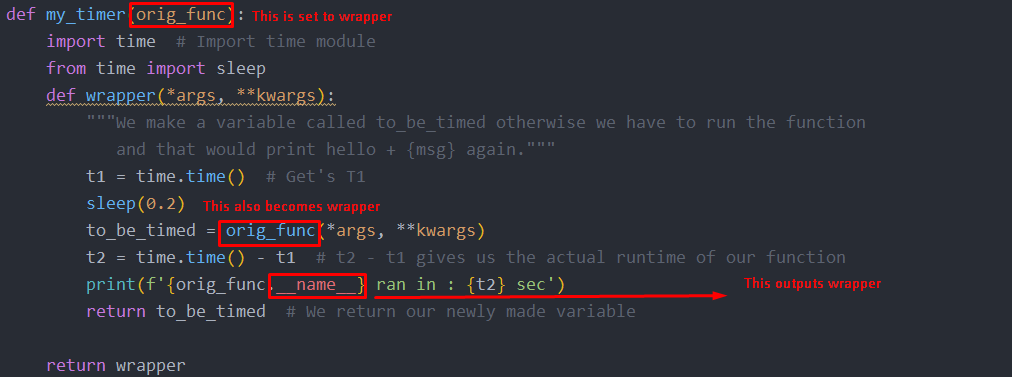
We can apply more decorators to a function, and it’s as easy as stacking them upon each other but there is a catch:

@my\_timer  
@logger\_function  
def hello\_world(msg):  
 print(f'Hello {msg}')  
  
  
hello\_world('world!')

Stacking decorators this way is basically saying:

my\_timer(logger\_function(hello\_world)) – which will give us some funny results

So in the upper example logger\_function() returns wrapper and my\_timer() gets \_\_name\_\_ from loggings inner function called wrapper thus we will get the following results.



Hello world!

wrapper ran in : 0.20046329498291016 sec

We can see that we get : wrapper ran in instead of hello\_world ran in xxxx seconds. That is because of this:

This is what happens when we do it the other way around.

@logger\_function 🡪 logger\_function(my\_timer(hello\_world))

@my\_timer  
def hello\_world(msg):  
 print(f'Hello {msg}')

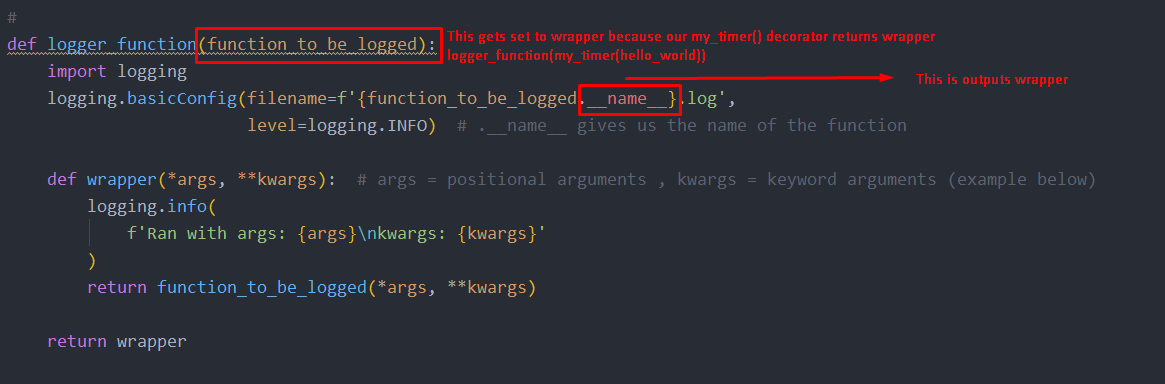
Hello\_world(‘world’)

Output:

Hello world!

hello\_world ran in : 0.20046353340148926 sec

This time output looks ok but our logging\_file name is wrapper.log instead of hello\_world.log because of the same principle explained above.Gonna put another pictures since I might forget this when I visit this document years later (hopefully not).



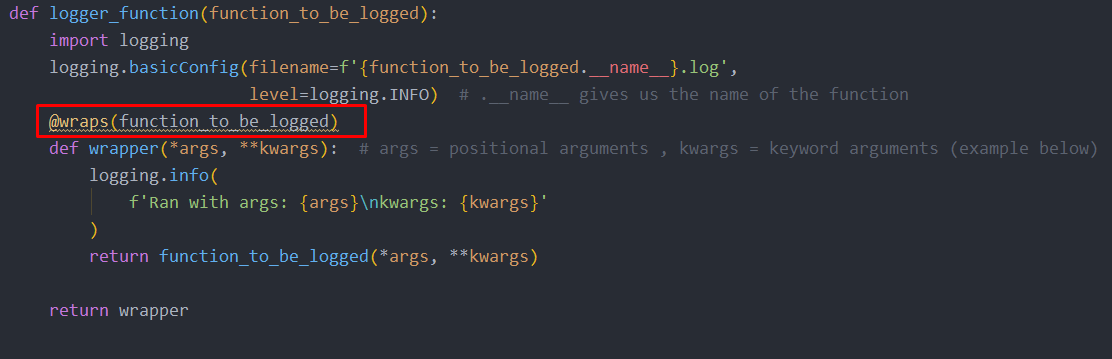
def my\_timer(orig\_func):  
 import time # Import time module  
 from time import sleep  
 def wrapper(\*args, \*\*kwargs): # We put this here because otherwise our it will raise an error since our orig\_func() has args  
 """We make a variable called to\_be\_timed otherwise we have to run the function  
 and that would print hello + {msg} again."""  
 t1 = time.time() # Get's T1  
 sleep(0.2)  
 to\_be\_timed = orig\_func(\*args, \*\*kwargs)  
 t2 = time.time() - t1 # t2 - t1 gives us the actual runtime of our function  
 print(f'{orig\_func.\_\_name\_\_} ran in : {t2} sec')  
 return to\_be\_timed # We return our newly made variable  
  
 return wrapper

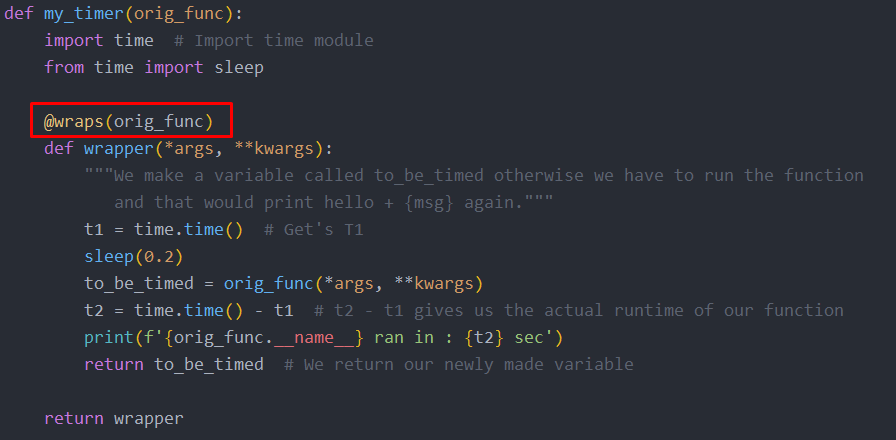
### The workaround

The workaround is very simple we just have to import wraps from the functools library and it fixes this.

from functools import wraps

After we import this we are going to wrap all our decorator arguments with this wraps() function





Don’t forget to add original\_function into wraps like this!!!!

@wraps(orig\_func)

After this the code should work like a charm.

# Doing decorators with classes

class decorator\_class():  
  
 def \_\_init\_\_(self,original\_function):  
 self.original\_function = original\_function  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print('The call method executed this before my\_timer()')  
 return self.original\_function(\*args, \*\*kwargs)  
  
@decorator\_class  
def hello\_world(msg):  
 print(f'Hello {msg}')  
  
hello\_world('world!')