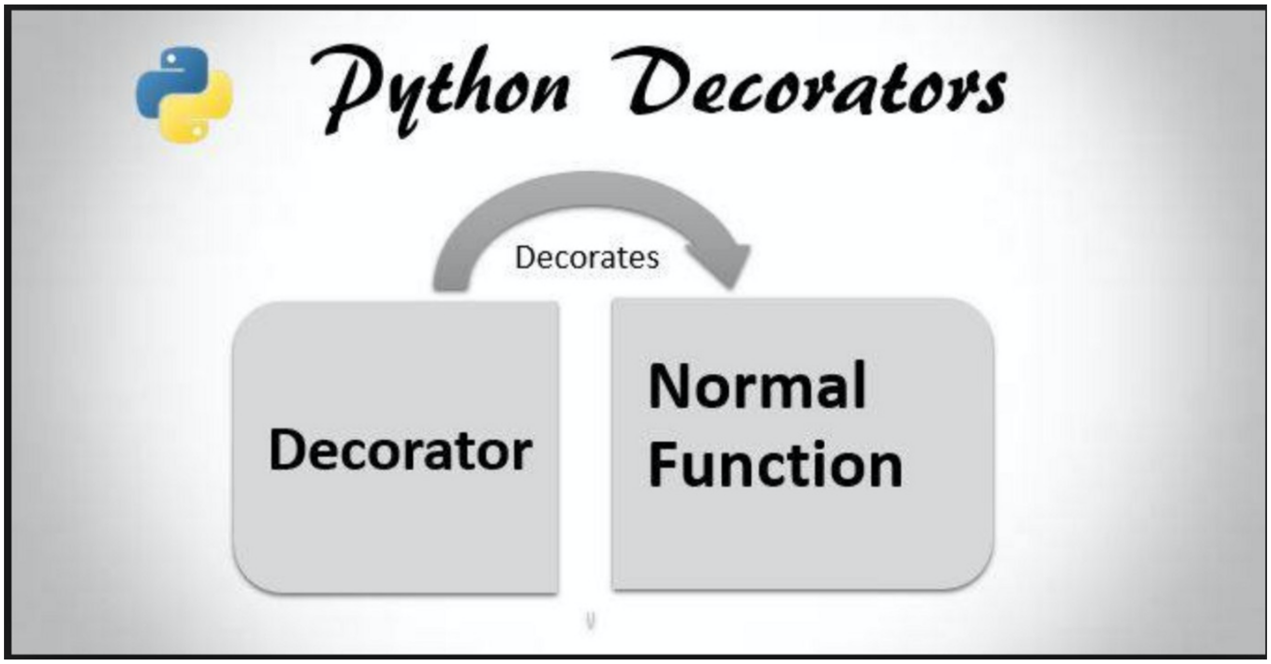
Decoding the mysterious Python Decorator

Preyansh shah Follow
May 1, 2019 · 5 min read ★



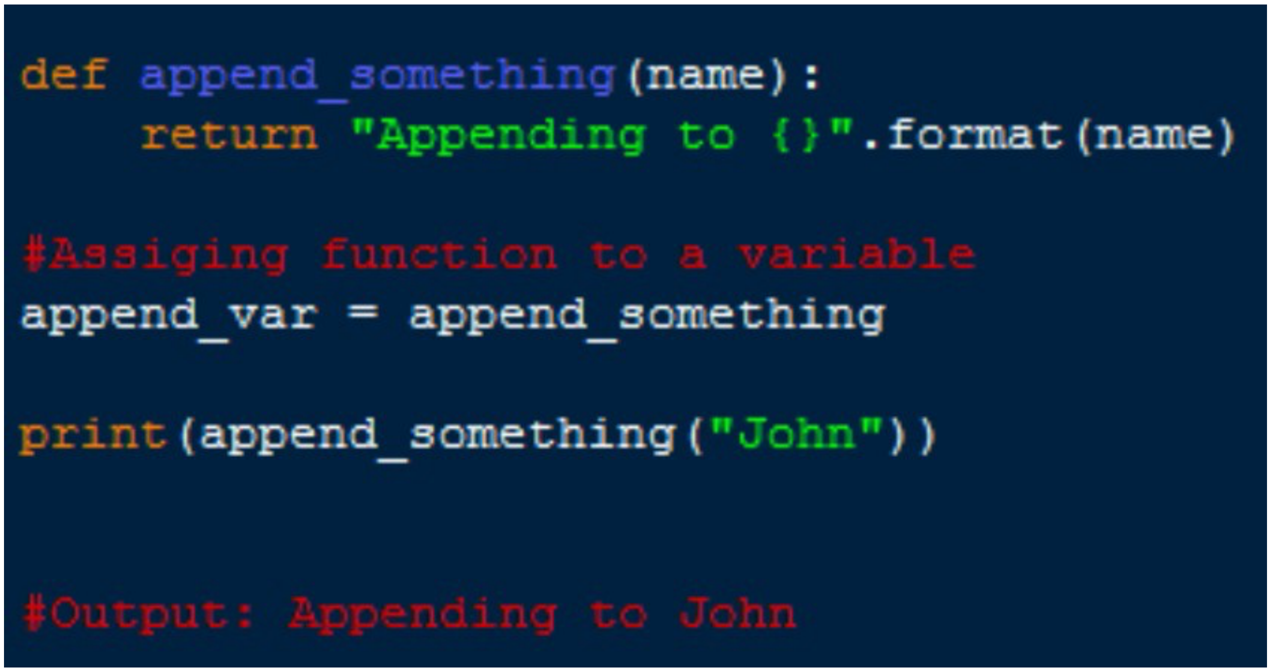
> *There is a lot of mystery surrounding the extended functionality that python decorators provide, especially for the beginners in python. There is no doubt to the fact that python decorators are hard to get along with. However, its only once we start understanding the basics of a decorator that we get to know the powerful stuff we can do with*

> *it.*

Lets get going. So what exactly are decorators? Well decorators work as wrappers which modify the behavior of a code before and after the execution of a function or a method, without modifying the function itself, thus extending the original functionality, hence decorating it. Confused with the definition? Lets start with some basics in python.

. . .

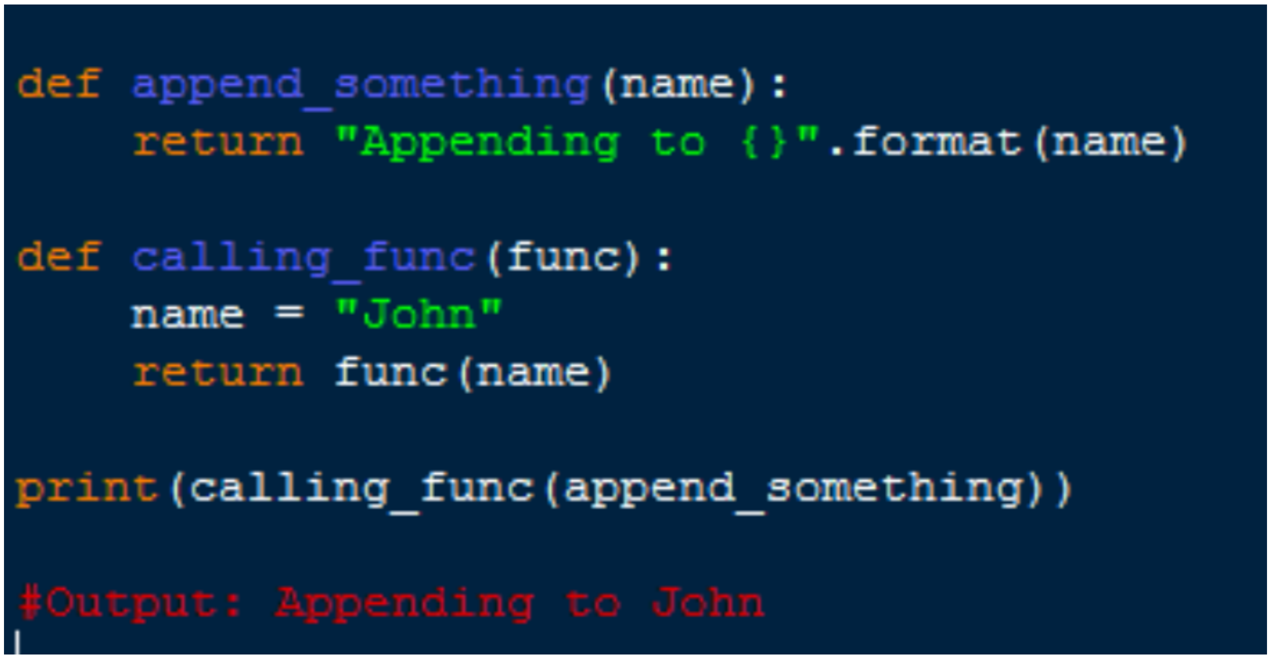
In python everything is treated as an object, and just like we can assign one variable to another, we can also assign function to a variable:



```
def append_something(name):
    return "Appending to {}".format(name)

#Assiging function to a variable
append_var = append_something

print(append_something("John"))


#Output: Appending to John
```

Fig1: Assigning function to a variable

Turns out that a function can also be passed as a parameter to some another function:

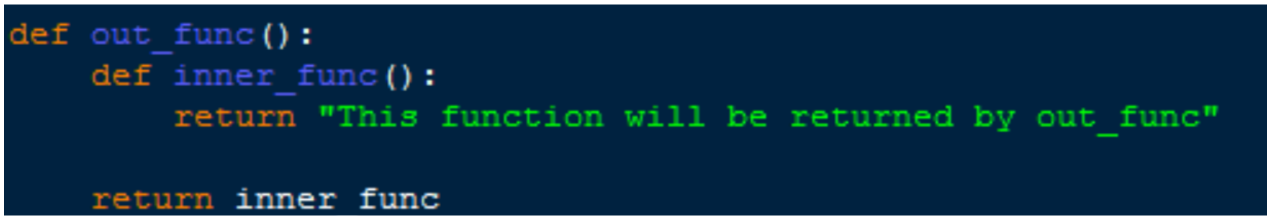


```
def append_something(name):
    return "Appending to {}".format(name)

def calling_func(func):
    name = "John"
    return func(name)

print(calling_func(append_something))

#Output: Appending to John
```

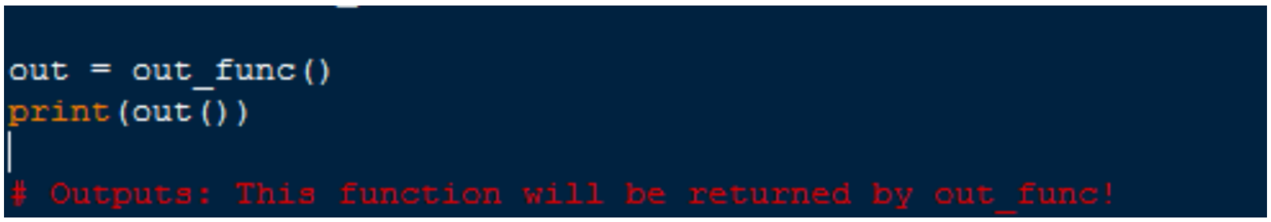
Fig2: Calling one function as a parameter of another function

As we can see in the above image, function ***append_something*** is being used as a parameter of the function ***calling_func***. We are then making an actual call to the ***append_something*** function inside the ***calling_func*** by passing the ***name*** variable as its parameter.

Interestingly functions in python can also return other functions:



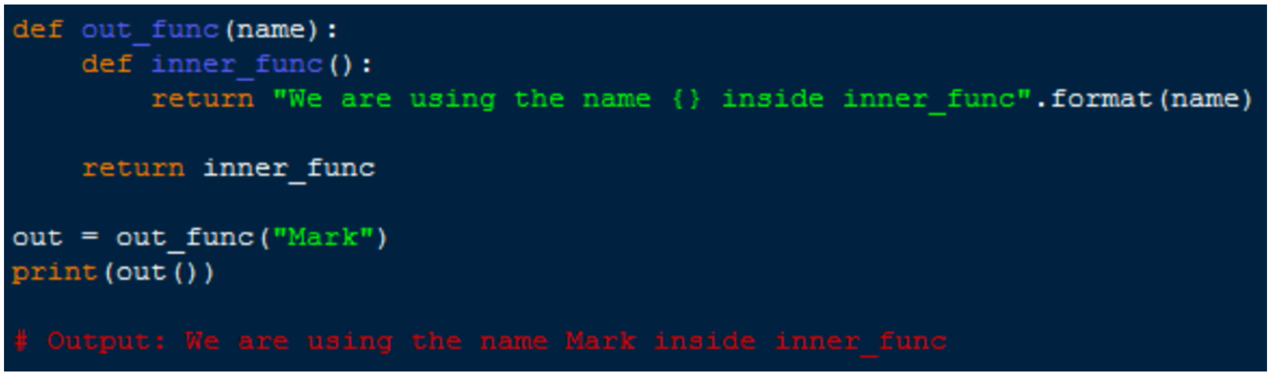
```
def out_func():
    def inner_func():
        return "This function will be returned by out_func"

    return inner_func
```



```
out = out_func()
print(out())

# Outputs: This function will be returned by out_func!
```

Fig3: Function returning another function

We can see in the above example that the function ***out_func*** is returning ***inner_func*** in its return statement. Now when we make a call to the ***out_func*** and assign it to the variable *out*, we are indirectly assigning ***inner_func*** as an object to the out variable. When we call ***out***, we are actually calling the ***inner_func*** function. This is a very important concept for taking a deep dive into decorators. Inner functions have access to the enclosing scope of the outer function and moreover, it is only allowed to have read access to the scope and not a write access. Lets modify the example above in **Fig3** by adding a 'name' parameter to the ***out_func*** which will be accessed by the ***inner_func*** which will use it:



```
def out_func(name):
    def inner_func():
        return "We are using the name {} inside inner_func".format(name)

    return inner_func

out = out_func("Mark")
print(out())

# Output: We are using the name Mark inside inner_func
```

Fig4: An example of how inner function can use the enclosing scope of outer function

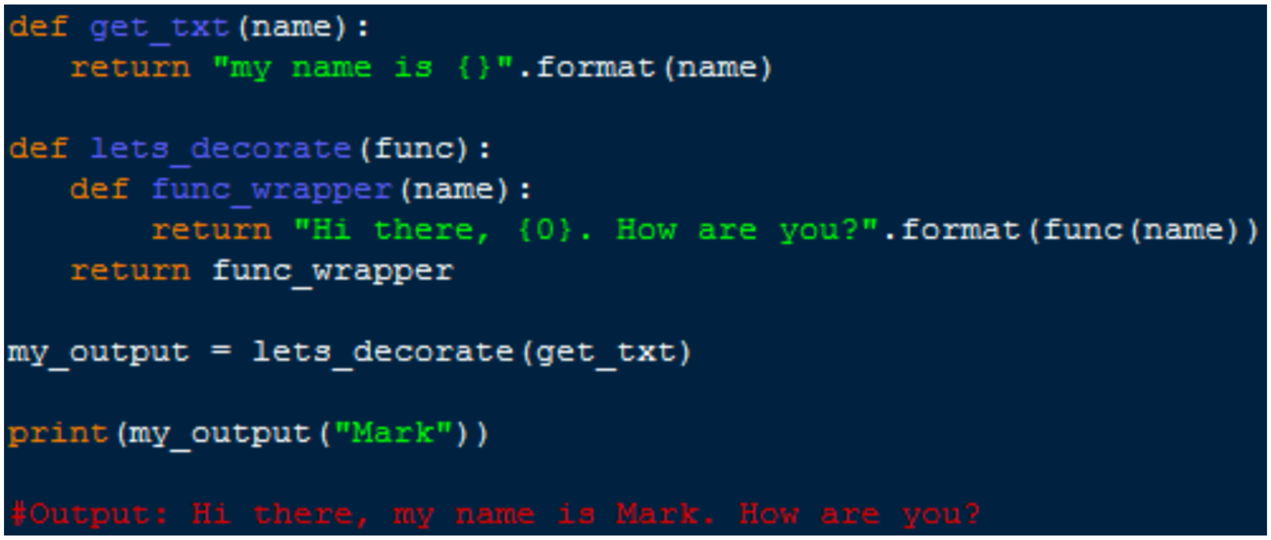
> *The above mentioned concept in Fig4 is also commonly known as* **closure** *in the language of python.*

Enough with the theory! Lets move onto decorators with the above concepts in mind.

. . .

Decorators

Using the above concepts, we can create a decorator which is nothing but a function wrapped around another function. Lets consider a function that wraps the output of another function:

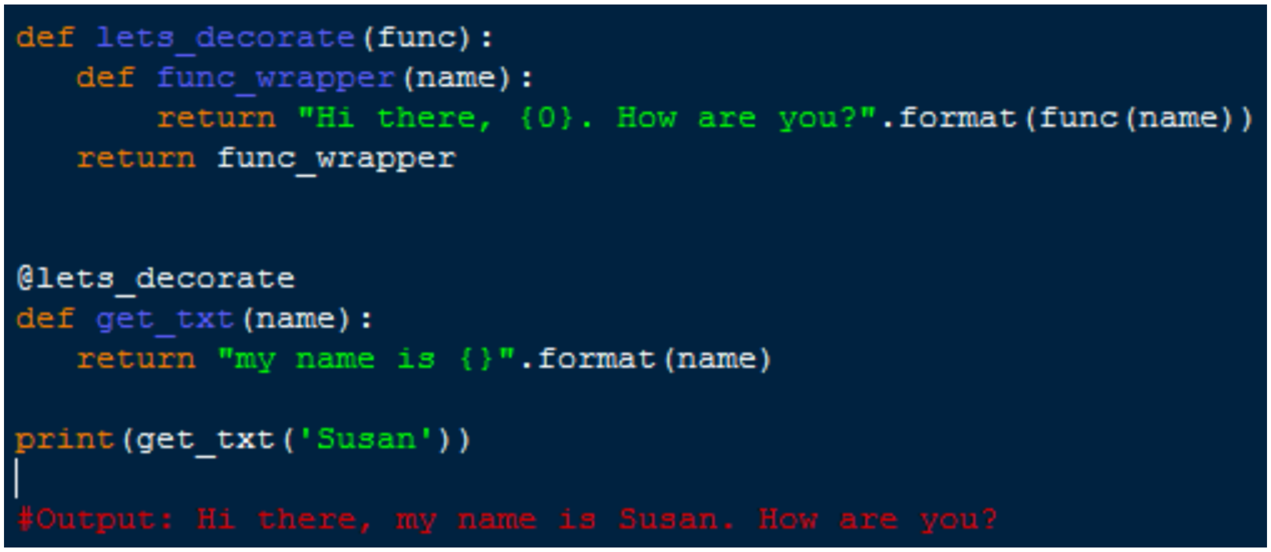


```
def get_txt(name):
   return "my name is {}".format(name)

def lets_decorate(func):
   def func_wrapper(name):
       return "Hi there, {0}. How are you?".format(func(name))
   return func_wrapper

my_output = lets_decorate(get_txt)

print(my_output("Mark"))

#Output: Hi there, my name is Mark. How are you?
```

Fig5: Decorating a function using another function

Notice how in the above example we are using the concept of inner function using the enclosed scope of outer function. Here we are using the ***lets_decorate*** function and passing ***get_txt*** function as a parameter to it. ***lets_decorate*** function returns ***func_wrapper*** which gets assigned to ***my_output*** variable. When we make a call to ***my_output*** variable by passing in a parameter we get the output. The output which we get augments the output of ***get_txt*** function (which is actually called inside the ***func_wrapper*** function), hence ***decorating*** it. I know its a little confusing at first, but if you give it a little thought, its actually a mixture of all the concepts we discussed in the above basics.

> *Fortunately, python provides a better way of creating & using decorators through some syntactic sugar:*



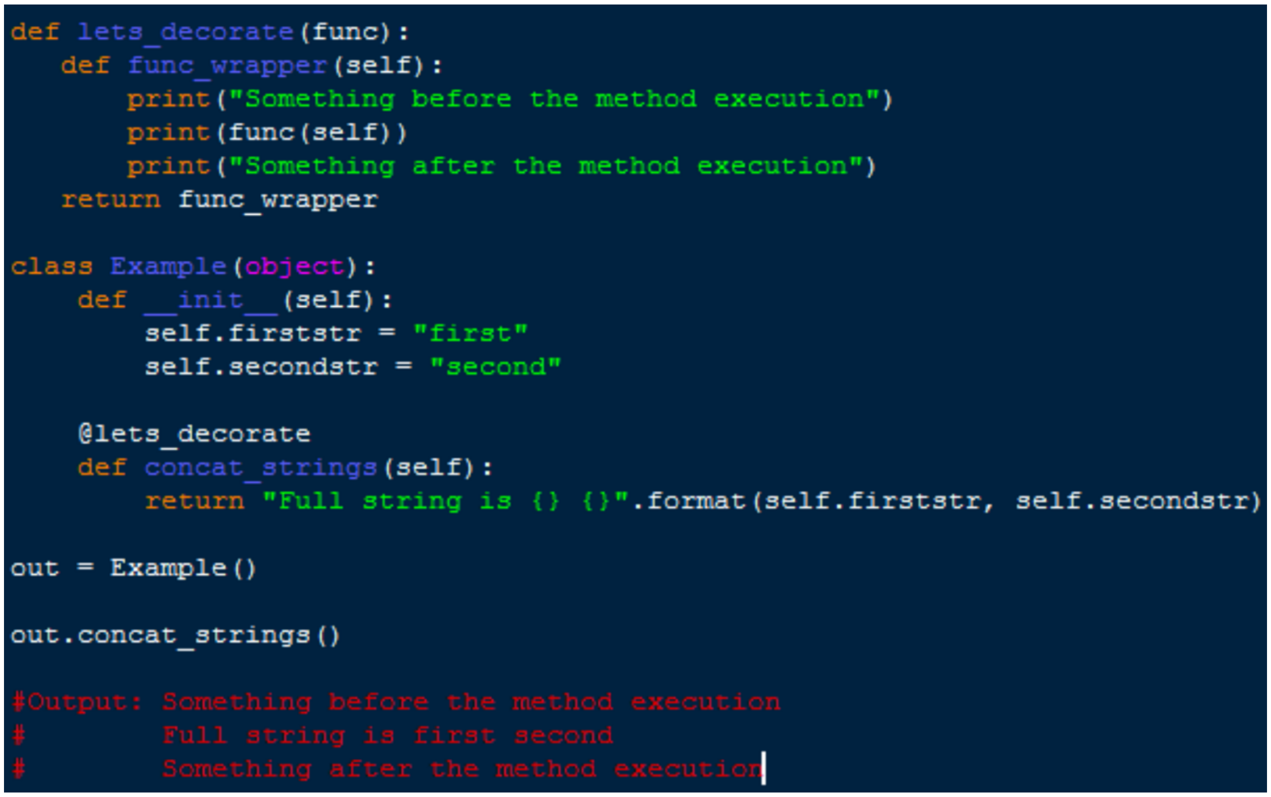
```
def lets_decorate(func):
   def func_wrapper(name):
       return "Hi there, {0}. How are you?".format(func(name))
   return func_wrapper


@lets_decorate
def get_txt(name):
   return "my name is {}".format(name)

print(get_txt('Susan'))
|
#Output: Hi there, my name is Susan. How are you?
```

Fig6: Creating and using decorators in python using syntactic sugar

We can see how the decorator *lets_decorate* is used on the function *get_txt* by prepending '@' symbol to the decorator, kind of a shortcut method of using decorators on a function.

Decorating Methods

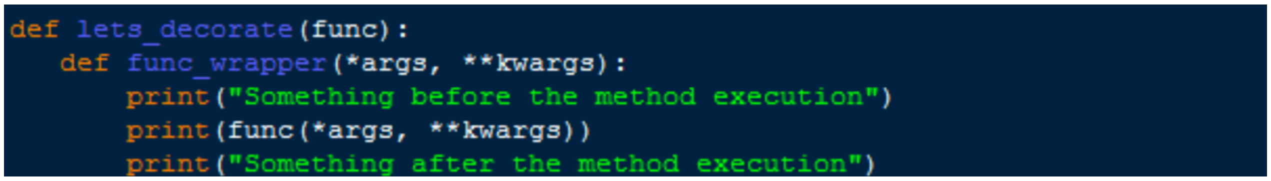
Methods expect their first parameter to be ***self***, a reference to the current object. We can build decorators for methods in the same way as above by taking self into consideration. Lets see an example:



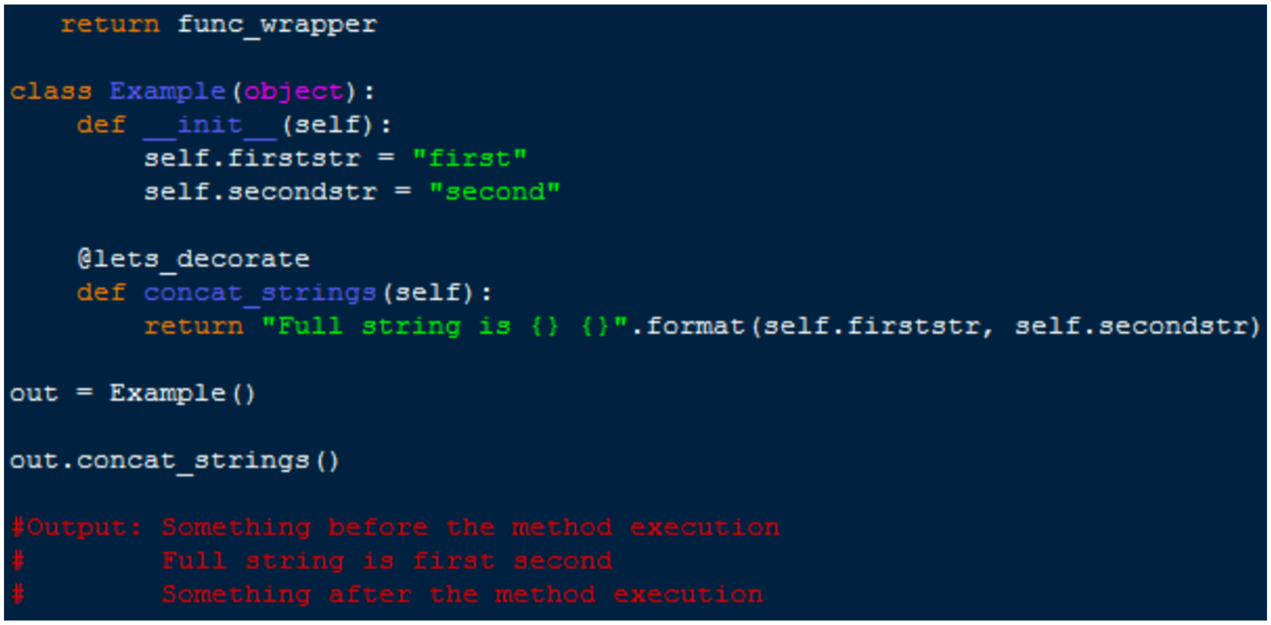
```
def lets_decorate(func):
   def func_wrapper(self):
       print("Something before the method execution")
       print(func(self))
       print("Something after the method execution")
   return func_wrapper

class Example(object):
    def __init__(self):
        self.firststr = "first"
        self.secondstr = "second"

    @lets_decorate
    def concat_strings(self):
        return "Full string is {} {}".format(self.firststr, self.secondstr)

out = Example()

out.concat_strings()

#Output: Something before the method execution
#         Full string is first second
#         Something after the method execution
```

Fig7: Using decorators on methods

Another and better approach would be to make our decorator useful for functions and methods alike.



```
def lets_decorate(func):
   def func_wrapper(*args, **kwargs):
       print("Something before the method execution")
       print(func(*args, **kwargs))
       print("Something after the method execution")
```



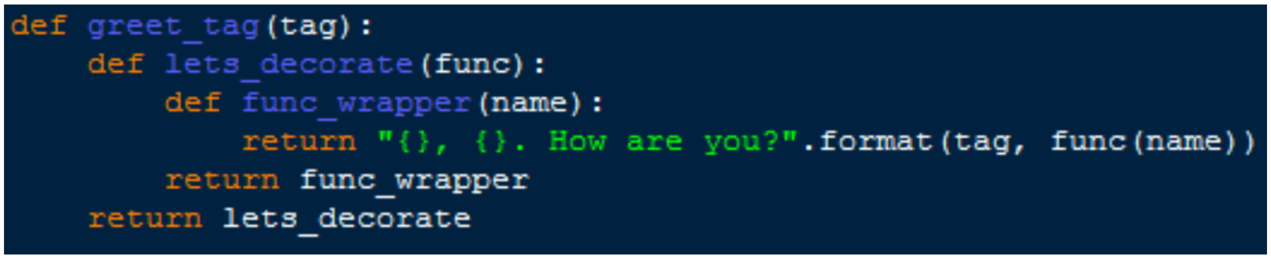
```
   return func_wrapper

class Example(object):
    def __init__(self):
        self.firststr = "first"
        self.secondstr = "second"

    @lets_decorate
    def concat_strings(self):
        return "Full string is {} {}".format(self.firststr, self.secondstr)

out = Example()

out.concat_strings()

#Output: Something before the method execution
#        Full string is first second
#        Something after the method execution
```

Fig8: Using decorators on methods with args and kwargs as parameters

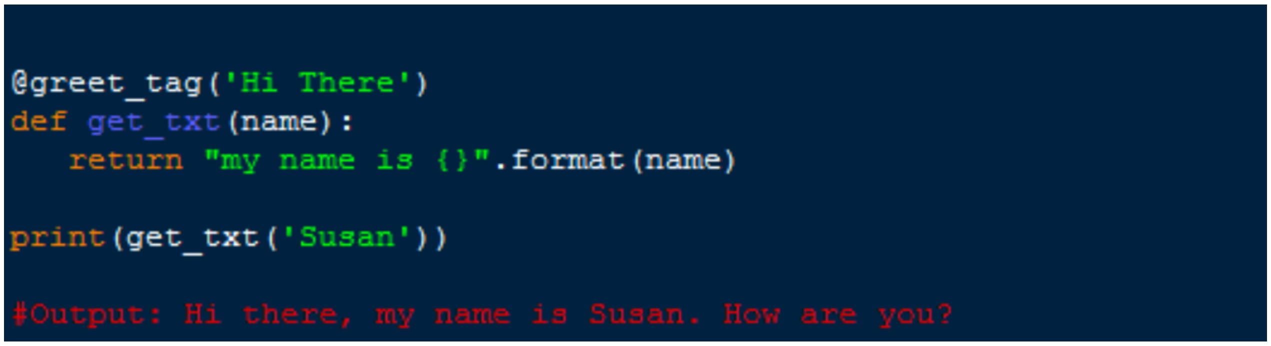
> args *and* kwargs *can be used to as parameters for the wrapper so that it can accept any arbitrary number of arguments and keyword arguments. You can learn more about args and kwargs here.*

Passing arguments to decorators

We can also pass arguments to a decorator. Decorators expect to receive a function as an argument and that is why to pass an argument, we will have to build a function that takes those extra arguments and generate our decorator.



```
def greet_tag(tag):
    def lets_decorate(func):
        def func_wrapper(name):
            return "{}, {}. How are you?".format(tag, func(name))
        return func_wrapper
    return lets_decorate
```

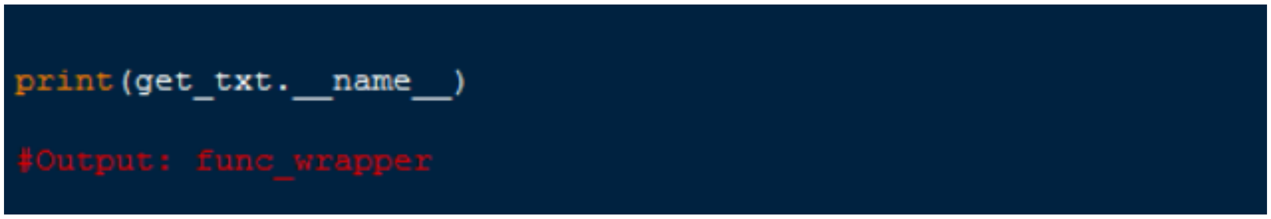


```
@greet_tag('Hi There')
def get_txt(name):
    return "my name is {}".format(name)

print(get_txt('Susan'))

#Output: Hi there, my name is Susan. How are you?
```

Fig9: Passing argument to a decorator

The use of functools

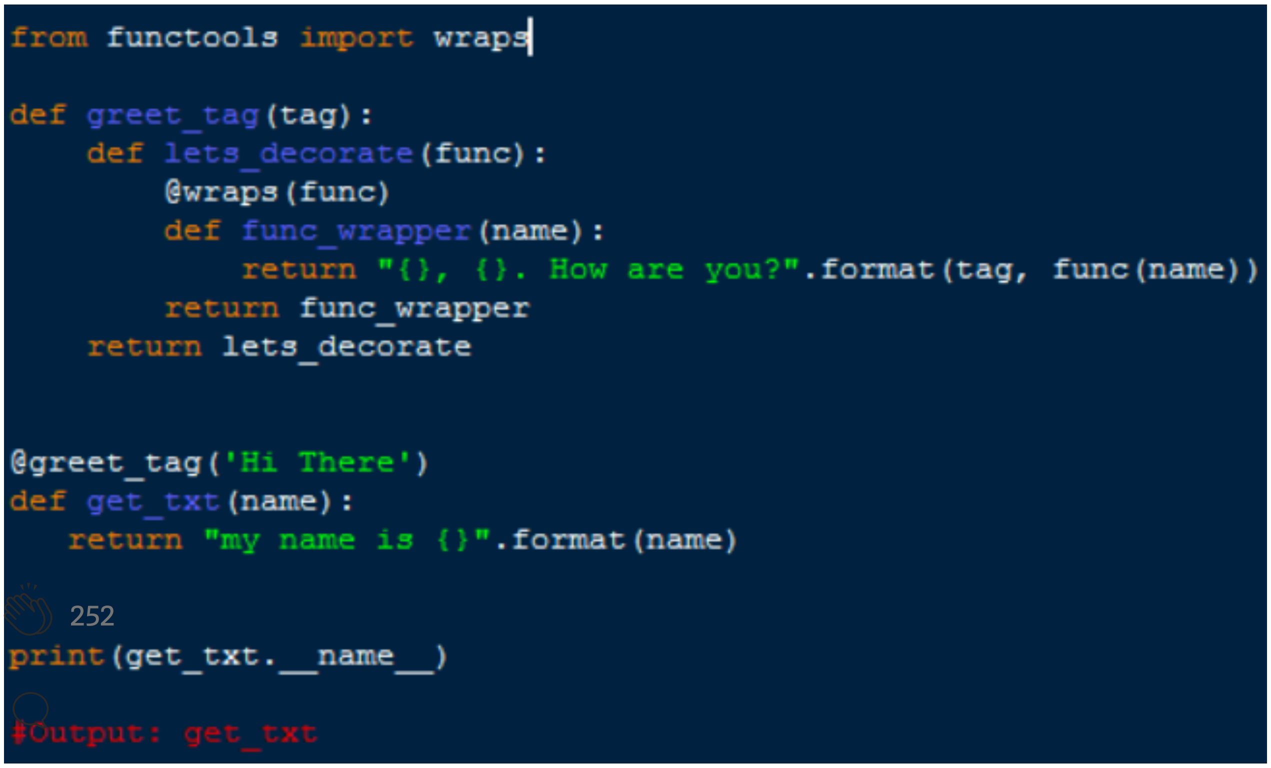
Lets try to debug the *get_txt* function above which has been decorated:



```
print(get_txt.__name__)

#Output: func_wrapper
```

Fig10: Decorator overrides the attributes of a function

As you can see instead of printing ***get_txt***, we get the output as ***func_wrapper***. So when a function is decorated, its attributes get overridden by those of the wrapper. We use **functools** module which contains **functools.wraps** to avoid this overriding of the attributes of a function getting decorated. Wraps is a decorator for updating the attributes of the wrapping function ***func_wrapper*** to those of the original function ***get_txt***. This is as simple as decorating ***func_wrapper*** by ***@wraps(func)***.



```python
from functools import wraps

def greet_tag(tag):
    def lets_decorate(func):
        @wraps(func)
        def func_wrapper(name):
            return "{}, {}. How are you?".format(tag, func(name))
        return func_wrapper
    return lets_decorate


@greet_tag('Hi There')
def get_txt(name):
   return "my name is {}".format(name)

252
print(get_txt.__name__)

#Output: get_txt
```

Fig11: Using wraps to maintain the attributes of the function being decorated

Use of Decorators

What we have covered till now is just the basics of how we can use decorators. There are a ton of things which can be done using decorators as they provide the power and elegance to your program by helping us to extend the behavior of functions and methods which we dont want to modify.

More reading resources

Here is a list of other resources worth checking out:

- **What is a decorator?**
- **Decorators I: Introduction to Python Decorators**
- **Python Decorators II: Decorator Arguments**
- **Python Decorators III: A Decorator-Based Build System**

Hope you find this post helpful for writing your libraries in python. You can share your views and concerns in the comments section. Happy coding!

Python3 Python Python Programming Programming Coding