Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

Decorators in Python:

A decorator in Python is a function that takes another function as its argument, and returns yet another function.

Decorators can be extremely useful as they allow the extension of an existing function, without any modification to the original function source code.

First, let's discuss how to write our own decorator by taking it's one step at a time so that we can fully understand it.

Everything in Python is an object:

First of all let's understand functions in Python:

```
def hi(name="yasoob"):
    return "hi " + name

print(hi())
# output: 'hi yasoob'

# We can even assign a function to a variable like
greet = hi
# We are not using parentheses here because we are not calling the function hi
# instead we are just putting it into the greet variable. Let's try to run this

print(greet())
# output: 'hi yasoob'

# Let's see what happens if we delete the old hi function!
del hi
print(hi())
#outputs: NameError

print(greet())
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
#outputs: 'hi yasoob'
```

Defining functions within functions:

So those are the basics when it comes to functions. Let's take our knowledge one step further. In Python we can define functions inside other functions:

```
def hi(name="yasoob"):
    print("now you are inside the hi() function")

    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    print(greet())
    print(welcome())
    print("now you are back in the hi() function")

hi()
#output:now you are inside the hi() function
#       now you are in the greet() function
#       now you are in the welcome() function
#       now you are back in the hi() function

# This shows that whenever you call hi(), greet() and welcome()
# are also called. However the greet() and welcome() functions
# are not available outside the hi() function e.g:

greet()
#outputs: NameError: name 'greet' is not defined
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

Returning functions from within functions:

It is not necessary to execute a function within another function, we can return it as an output as well:

```
def hi(name="yasoob"):
    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    if name == "yasoob":
        return greet
    else:
        return welcome

a = hi()
print(a)
#outputs: <function greet at 0x7f2143c01500>

#This clearly shows that `a` now points to the greet() function in hi()
#Now try this

print(a())
#outputs: now you are in the greet() function
```

Just take a look at the code again. In the `if/else` clause we are returning `greet` and `welcome`, not `greet()` and `welcome()`. Why is that? It's because when we put a pair of parentheses after it, the function gets executed; whereas if we don't put parenthesis after it, then it can be passed around and can be assigned to other

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

variables without executing it. Did we get it? Let us explain it in a little bit more detail. When we write a = hi(), hi() gets executed and because the name is yasoob by default, the function greet is returned. If we change the statement to a = hi(name = "ali") then the welcome function will be returned. We can also do print hi()() which outputs now we are in the greet() function.

Giving a function as an argument to another function:

```
def hi():
    return "hi yasoob!"

def doSomethingBeforeHi(func):
    print("I am doing some boring work before executing hi()")
    print(func())

doSomethingBeforeHi(hi)
#outputs:I am doing some boring work before executing hi()
#        hi yasoob!
```

Now we have all the required knowledge to learn what decorators really are. Decorators let us execute code before and after a function.

Writing our first decorator:

In the last example we actually made a decorator! Let's modify the previous decorator and make a little bit more usable program:

```
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
        a_func()

        print("I am doing some boring work after executing a_func()")

    return wrapTheFunction

def a_function_requiring_decoration():
    print("I am the function which needs some decoration to remove my foul smell")

a_function_requiring_decoration()
#outputs: "I am the function which needs some decoration to remove my foul smell"

a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
#now a_function_requiring_decoration is wrapped by wrapTheFunction()

a_function_requiring_decoration()
#outputs:I am doing some boring work before executing a_func()
#        I am the function which needs some decoration to remove my foul smell
#        I am doing some boring work after executing a_func()
```

We just applied the previously learned principles. This is exactly what the decorators do in Python! They wrap a function and modify its behaviour in one way or another. Now it might be wondering why we did not use the @ anywhere in our code? That is just a short way of making up a decorated function. Here is how we could have run the previous code sample using @.

```
@a_new_decorator
def a_function_requiring_decoration():
    """Hey you! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

a_function_requiring_decoration()
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
#outputs: I am doing some boring work before executing a_func()
#         I am the function which needs some decoration to remove my foul smell
#         I am doing some boring work after executing a_func()

#the @a_new_decorator is just a short way of saying:
a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

I hope we now have a basic understanding of how decorators work in Python. Now there is one problem with our code. If we run:

```
print(a_function_requiring_decoration.__name__)
# Output: wrapTheFunction
```

That's not what we expected! Its name is "a_function_requiring_decoration". Well, our function was replaced by wrapTheFunction. It overrode the name and docstring of our function. Luckily, Python provides us a simple function to solve this problem and that is functools.wraps. Let's modify our previous example to use functools.wraps:

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")
        a_func()
        print("I am doing some boring work after executing a_func()")
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Hey yo! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
print(a_function_requiring_decoration.__name__)
# Output: a_function_requiring_decoration
```

Now that is much better. Let's move on and learn some use-cases of decorators.

Blueprint:

```
from functools import wraps
def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Function will not run"
        return f(*args, **kwargs)
    return decorated
@decorator_name
def func():
    return("Function is running")
can_run = True
print(func())
# Output: Function is running

can_run = False
print(func())
# Output: Function will not run
```

Note: `@wraps` takes a function to be decorated and adds the functionality of copying over the function name, docstring, arguments list, etc. This allows us to access the pre-decorated function's properties in the decorator.

Use-cases:

Now let's take a look at the areas where decorators really shine and their usage

makes something really easy to manage.

Authorization

Decorators can help to check whether someone is authorized to use an endpoint in a web application. They are extensively used in Flask web framework and Django. Here is an example to employ decorator based authentication:

Example :

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

Logging

Logging is another area where the decorators shine. Here is an example:

```
from functools import wraps
def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
    """Do some math."""
    return x + x

result = addition_func(4)
# Output: addition_func was called
```

Some smart uses of decorators:

Decorators with Arguments

Come to think of it, isn't @wraps also a decorator? But, it takes an argument like any normal function can do. So, why can't we do that too?

This is because when we use the @my_decorator syntax, we are applying a wrapper function with a single function as a parameter. Remember, everything in Python is an object, and this includes functions! With that in mind, we can write a function that returns a wrapper function.

Nesting a Decorator Within a Function

Let's go back to our logging example, and create a wrapper which lets us specify a logfile to output to.

```
from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " was called"
            print(log_string)
            # Open the logfile and append
            with open(logfile, 'a') as opened_file:
                # Now we log to the specified logfile
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
                opened_file.write(log_string + '\n')
            return func(*args, **kwargs)
        return wrapped_function
    return logging_decorator

@logit()
def myfunc1():
    pass

myfunc1()
# Output: myfunc1 was called
# A file called out.log now exists, with the above string

@logit(logfile='func2.log')
def myfunc2():
    pass

myfunc2()
# Output: myfunc2 was called
# A file called func2.log now exists, with the above string
```

Decorator Classes

Now we have our logit decorator in production, but when some parts of our application are considered critical, failure might be something that needs more immediate attention. Let's say sometimes we want to just log to a file. Other times we want an email sent, so the problem is brought to our attention, and still keep a log for our own records. This is a case for using inheritence, but so far we've only seen functions being used to build decorators.

Luckily, classes can also be used to build decorators. So, let's rebuild logit as a class instead of a function.

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
class logit(object):

    _logfile = 'out.log'

    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        log_string = self.func.__name__ + " was called"
        print(log_string)
        # Open the logfile and append
        with open(self._logfile, 'a') as opened_file:
            # Now we log to the specified logfile
            opened_file.write(log_string + '\n')
        # Now, send a notification
        self.notify()

        # return base func
        return self.func(*args)



    def notify(self):
        # logit only logs, no more
        pass
```

This implementation has an additional advantage of being much cleaner than the nested function approach, and wrapping a function still will use the same syntax as before:

```
logit._logfile = 'out2.log' # if change log file
@logit
def myfunc1():
    pass

myfunc1()
```

Decorators in Python

Assignment Class-28

4105436-Mohammad Nora Alam Bhuiyan

```
# Output: myfunc1 was called
```

Now, let's subclass logit to add email functionality (though this topic will not be covered here).

```
class email_logit(logit):
    '''
    A logit implementation for sending emails to admins
    when the function is called.
    '''
    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # Send an email to self.email
        # Will not be implemented here
        Pass
```

From here, @email_logit works just like @logit but sends an email to the admin in addition to logging.

References:

1. https://book.pythontips.com/en/latest/decorators.html