😊 Python Decorators

What are Python Decorators?

Python decorators are a powerful feature that allows modifying or extending the behavior of functions or methods without permanently altering their structure. They are a type of higher-order function, meaning they take another function as input, add functionality to it, and return a modified version of the function.

How Do Decorators Work?

A decorator wraps a function, allowing code to be executed before or after the function runs. This is done using the @decorator_name syntax, which simplifies applying decorators to functions.

Use Cases of Python Decorators

1. **Logging:** Automatically log function calls, execution time, or return values.
2. **Authentication and Authorization:** Restrict access to certain functions based on user roles.
3. **Memoization (Caching):** Store function results to optimize performance for expensive computations.
4. **Timing Functions:** Measure the execution time of a function to analyze performance.
5. **Validation and Input Processing:** Check and modify input parameters before passing them to a function.
6. **Enforcing Coding Standards:** Ensure consistent behavior in a codebase by applying rules at the function level.

Benefits of Using Decorators

1. **Code Reusability:** Decorators allow the same functionality to be reused across multiple functions without code duplication.
2. **Separation of Concerns:** They enable separation of logic, making functions cleaner by moving additional behavior elsewhere.
3. **Enhances Readability:** Since decorators encapsulate additional behavior, the core function remains easy to read and understand.
4. **Flexible and Scalable:** New features can be added to functions dynamically without modifying their actual implementation.
5. **Encapsulation of Logic:** They keep repeated patterns in one place, making debugging and maintenance easier.

Python decorators are widely used in frameworks such as Flask and Django for authentication, middleware, and request handling, making them essential for efficient programming.

😄 To begin with the Lab

1. We start by defining a function and getting output from the function.

```
[1]: def func():
         return 1
```

```
[2]: func()
```

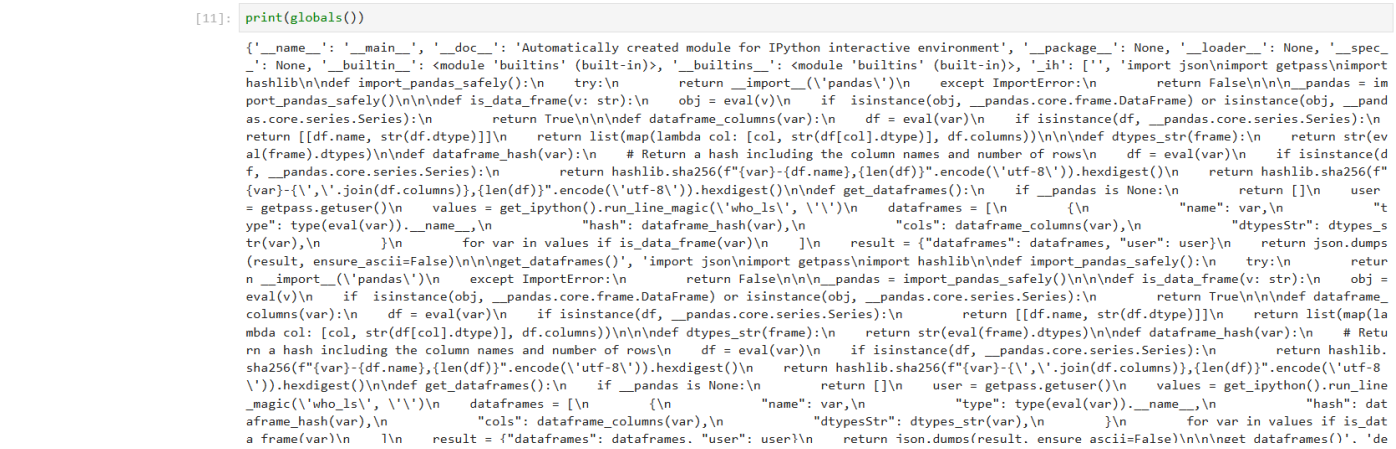
```
[2]: 1
```

2. This code demonstrates the use of the built-in functions locals () and globals (). It first creates a global variable s. The function check_for_locals is defined to print its local variables when called, but it is not executed here. Finally, globals() is called, which prints a dictionary of all global variables and their values in the current module, including s and other built-in objects.

```
[8]: s = 'Global Variable'

     def check_for_locals():
         print(locals())
```

```
[11]: print(globals())
```



3. The first statement prints all the names (keys) defined in the global scope. The second line retrieves the value of the global variable 's' from the globals dictionary. Finally, calling check_for_locals() prints the local variables within that function (which, in this case, is an empty dictionary since no local variables were defined).

```
[13]: print(globals().keys())

dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__builtin__', '__builtins__', '_ih', '_oh', '_dh', 'In', 'Out', 'get_ipytho
n', 'exit', 'quit', 'open', '_', '__', '___', '__session__', '_i', '_ii', '_iii', '_i1', 'json', 'getpass', 'hashlib', 'import_pandas_safely', '__panda
s', 'is_data_frame', 'dataframe_columns', 'dtypes_str', 'dataframe_hash', 'get_dataframes', '_1', '_i2', '_2', '_i3', 'func', '_i4', '_4', '_i5', '_5',
'_i6', '_6', '_i7', '_7', '_i8', 's', 'check_for_locals', '_i9', '_9', '_i10', '_10', '_i11', '_i12', '_12', '_i13'])
```

Note how **s** is there, the Global Variable we defined as a string:

```
[17]: globals()['s']

[17]: 'Global Variable'
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

```
[20]: check_for_locals()

{}
```

4. Below, you can see that we defined hello as a function and then returned greet with the person's name.

```
[22]: def hello(name='Jose'):
          return 'Hello '+name

[24]: hello()

[24]: 'Hello Jose'
```

5. In this snippet, the function object referenced by the name hello is assigned to the variable greet. When you evaluate greet, it shows that greet now points to the same function as hello. Calling greet () executes the function hello.

```
[27]: greet = hello

[29]: greet

[29]: <function __main__.hello(name='Jose')>

[32]: greet()

[32]: 'Hello Jose'
```

6. So, what will happen if we delete the name hello (). Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

[34]:
```
del hello
```

[36]:
```
hello()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 1 hello()

NameError: name 'hello' is not defined
```

[77]:
```
greet()
```

[77]:
```
'Hello Jose'
```

7. Inside hello(), the functions greet() and welcome() are defined **only in the local scope** of hello(). That means they exist and can be called **only** while hello() is running. Once hello() finishes, those inner functions are no longer accessible. Attempting to call welcome() outside hello() causes a NameError because welcome() does not exist in the global scope.

[79]:
```
def hello(name='Jose'):
    print('The hello() function has been executed')

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    print(greet())
    print(welcome())
    print("Now we are back inside the hello() function")
```

[81]:
```
hello()
```

```
The hello() function has been executed
         This is inside the greet() function
         This is inside the welcome() function
Now we are back inside the hello() function
```

[83]:
```
welcome()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[83], line 1
----> 1 welcome()

NameError: name 'welcome' is not defined
```

8. The function hello defines two inner functions, greet and welcome. When hello is called without arguments (defaulting to "Jose"), it returns the greet function. This means that x is now the greet function. When you call x() (or greet()), it executes the greet function and prints its returned string.

[86]:
```
def hello(name='Jose'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Jose':
        return greet
    else:
        return welcome
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

[89]:
```
x = hello()
```

[91]:
```
x
```

[91]:
```
<function __main__.hello.<locals>.greet()>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

[94]:
```
print(x())
```

```
        This is inside the greet() function
```

9. Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator.

[98]:
```
def hello():
    return 'Hi Jose!'

def other(func):
    print('Other code would go here')
    print(func())
```

[100]:
```
other(hello)
```

```
Other code would go here
Hi Jose!
```

10. This code demonstrates how decorators work in Python. It defines a decorator called new_decorator that takes a function as input, wraps it in another function (wrap_func) which prints messages before and after calling the original function, and then returns this new function.
11. Initially, func_needs_decorator is defined to print a message. When called normally, it simply prints its message. Next, the function is manually wrapped by new_decorator, so calling it prints the messages before and after its original output. Finally, the decorator syntax (@new_decorator) is used to automatically apply the decorator to func_needs_decorator, achieving the same behavior.

[104]:
```
def new_decorator(func):

    def wrap_func():
        print("Code would be here, before executing the func")

        func()

        print("Code here will execute after the func()")

    return wrap_func

def func_needs_decorator():
    print("This function is in need of a Decorator")
```

[106]:
```
func_needs_decorator()
```

```
This function is in need of a Decorator
```

[108]:
```
# Reassign func_needs_decorator
func_needs_decorator = new_decorator(func_needs_decorator)
```

[110]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

So what just happened here? A decorator simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

[113]:
```
@new_decorator
def func_needs_decorator():
    print("This function is in need of a Decorator")
```

[115]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```