```
# DECORATORS , MODIFY AND EXTEND THE BEHAVIOUS OF FUNCTIONS AND
METHODS, WITHOUT CHANGING THE ACTUAL CODE

# ,ALLOWS YOU TO WRAP ANOTHER FUNCTION

# FUNCTIONS CAN BE PASSED AS AN aGRUMENT TO ANOTHER FUNCTION

# WRAPPER FUNCTION --> a DECORATOR USES INNER FUNCTION TO MODIFY THE
BEHAVIOUR

def decorator_function(original_function):
    def wrapper_function():
        print("Function is being decorated")
        return original_function()
    return wrapper_function




@decorator_function
def display():
    print("display function is being called")


display()




# The decorator that accepts an argument
import time

# The decorator that accepts an argument

def time_decorator(log_message):
    def decorator_function(original_function):
        def wrapper_function():
            start_time = time.time()  # Record the start time
            result = original_function()  # Call the original function
            print
            end_time = time.time()  # Record the end time
            execution_time = end_time - start_time
```

```python
            print(f"{log_message} - Function 
{original_function.__name__} took {execution_time:.4f} seconds to 
execute.")
            return result
        return wrapper_function
    return decorator_function

# Using the decorator with a custom message
# TIME DEC = (MSG)(SLOW FUNC)
@time_decorator("Execution Time Log")
def slow_function():
    time.sleep(2)  # Simulate a slow function by sleeping for 2 seconds
    print("Slow function executed.")

slow_function()



def decorator_function(MSG):
    def wrapper_function(original_function):
        def inner_wrapper(*args, **kwargs):
            print(f"Function is being decorated with message: {MSG}")
            return original_function(*args, **kwargs)
        return inner_wrapper
    return wrapper_function


@decorator_function("This is a custom message")
def display(name):
    print(f"Display function is called by {name}")

display("Alice")
```