Nested Functions, Closures, and Decorators in Python

1. Nested Functions

A **nested function** is a function defined inside another function. It is useful for encapsulation, improving code structure, and restricting access to inner functions.

Example:

```
def outer_function():
    print("Inside Outer Function")

    def inner_function():
        print("Inside Inner Function")

    inner_function()  # Calling inner function inside the outer function

outer_function()
```

Output:

```
Inside Outer Function

Inside Inner Function
```

Key Points:

- Inner functions are only accessible within the outer function.
- Useful for hiding implementation details from the outside.

2. Closures

A **closure** is a nested function that retains access to the variables from its enclosing function even after the enclosing function has finished execution.

Example:

```
def outer_function(message):
    print("Inside Outer Function")

    def inner_function():
        print(f"Message from Closure: {message}")  # Retains 'message' from outer scope

    return inner_function  # Returning inner function

closure_ref = outer_function("Hello, Closures!")
closure_ref()
```

Output:

```
Inside Outer Function

Message from Closure: Hello, Closures!
```

Key Points:

- Closures allow the inner function to remember variables from the outer function.
- Useful in data hiding and maintaining state without using global variables.

3. Decorators

A **decorator** is a function that takes another function as input, modifies or enhances its behavior, and returns a new function without modifying the original function's code.

Example:

```
def outer_decorator(function):
    print("Inside Outer Decorator")  # Runs when the decorator is applied
    def inner_wrapper():
        print("Before calling the function")
        function()  # Calling the actual function
        print("After calling the function")
    return inner_wrapper  # Returning the modified function

@outer_decorator  # Applying the decorator
def print_message():
    print("Hello, Decorators!")

print_message()  # Calling the decorated function
```

Output:

```
Inside Outer Decorator
Before calling the function
Hello, Decorators!
After calling the function
```

Key Points:

- Decorators modify function behavior without changing their definition.
- The `@decorator_function` syntax is used to apply decorators.
- Commonly used for logging, authentication, and measuring execution time.

Conclusion

- **Nested Functions** help in better code organization and encapsulation.
- **Closures** allow inner functions to retain access to outer variables even after execution.
- **Decorators** dynamically modify function behavior, improving code reusability and readability.

Summary Table

Concept	Definition	Use Cases
Nested Functions	A function inside another function.	Code organization, modularity.
Closures	Inner function retains access to outer function’s variables even after execution.	Function factories, maintaining state.
Decorators	A function that modifies another function’s behaviour.	Logging, authentication, performance measurement.