**Closures in Python**

A **closure** in Python is a function object that remembers values in enclosing scopes even if those scopes are no longer present. Closures are a key feature of **functional programming** and allow for more concise and modular code.

**How Closures Work**

A closure occurs when:

1. A **nested function** is defined inside an enclosing function.
2. The **inner function references** variables from the enclosing scope.
3. The **enclosing function returns the inner function**, allowing it to be called later.

**Example of a Closure**

def outer\_function(message):

def inner\_function():

print(f"Message: {message}") # Captures 'message' from enclosing scope

return inner\_function

closure\_example = outer\_function("Hello, Python!")

closure\_example() # Output: Message: Hello, Python!

* Here, inner\_function() remembers the value of message even though outer\_function() has finished executing.

**When to Use Closures**

Closures are useful for:

1. **Data Encapsulation**: They help keep variables hidden from external access.
2. **Reducing Global Variables**: Since closures maintain state, you don’t need to use global variables.
3. **Function Factories**: Generating multiple variations of a function.

**Practical Use Case: Function Factory**

def multiplier(factor):

def multiply\_by\_factor(number):

return number \* factor # Captures 'factor' from outer function

return multiply\_by\_factor

double = multiplier(2)

triple = multiplier(3)

print(double(5)) # Output: 10

print(triple(5)) # Output: 15

* double and triple are closures that "remember" the values of factor.

**Closures vs Global Variables**

A closure helps avoid using global variables by keeping state inside the function.

**Using Global Variable (Less Recommended)**

factor = 2

def multiply(number):

return number \* factor

print(multiply(5)) # Output: 10

* The problem here is that factor is a global variable and can be changed anywhere in the code, leading to potential bugs.

**Using Closure (Better Approach)**

def multiplier(factor):

def multiply\_by\_factor(number):

return number \* factor

return multiply\_by\_factor

double = multiplier(2)

print(double(5)) # Output: 10

* Here, factor is encapsulated inside multiplier, preventing accidental modifications.

**Closures with Non-Local Variables**

If you want to modify a variable from the outer function, use the nonlocal keyword.

def counter():

count = 0 # Encapsulated state

def increment():

nonlocal count # Allows modifying 'count' from the outer scope

count += 1

return count

return increment

counter\_instance = counter()

print(counter\_instance()) # Output: 1

print(counter\_instance()) # Output: 2

**Summary**

✔ Closures allow inner functions to remember variables from enclosing scopes.  
✔ Useful for **function factories, stateful functions, and avoiding global variables**.  
✔ Use nonlocal to modify outer function variables within the inner function.