```
"""
A closure in Python is a nested function that remembers and has access to variables from its enclosing scope, even after the outer function has finished executing. Thi
Here's a breakdown of the key characteristics of a Python closure:
Nested Function: A closure is always an inner function defined within another function.
Access to Enclosing Scope: The inner function can access variables (also known as "free variables") from the outer (enclosing) function's scope.
Remembers Environment: Crucially, the inner function retains access to these variables even after the outer function has completed its execution and its local variable
Returned from Outer Function: The outer function typically returns the inner function as its result.

Closures in Python are like вЂњmemory-equippedвЂќ functions. They allow a function to remember values from the environment in which it was created even if that environ

How Closures are Created
A closure is formed when:

A function is defined inside another function (nested function).
The inner function references variables from the outer function.
The outer function returns the inner function.
Example:

def outer_function(message):
    # 'message' is a variable in the enclosing scope
    def inner_function():
        print(message)
    return inner_function

# Create a closure
my_closure = outer_function("Hello from the closure!")

# Call the closure - it still remembers 'message'
my_closure() # Output: Hello from the closure!
Why use closures?
Data Hiding and Encapsulation: Closures can be used to create functions with private or hidden states, similar to how objects encapsulate data.
Factory Functions: They are useful for creating functions dynamically based on some input parameters.
Decorators: Python decorators extensively use closures to modify or enhance the behavior of other functions.
Partial Application: Closures can be used to create new functions by fixing some arguments of an existing function.
Note: If you need to modify a variable from the enclosing scope within the inner function, you must use the nonlocal keyword to explicitly declare that the variable is
"""
def func():
    print("Inside Function")
    return 10


f1 = func
print(f1) # <function func at 0x00000200F8688C20>

f2 = func()
print(f2) # 10

a = f1()
print("a value is ", a)


def outer_func(name):
    def inner_func():
        print("Hello ", name)
    return inner_func


inner_func_obj = outer_func("User!!!")
print(inner_func_obj) # <function outer_func.<locals>.inner_func at 0x000001B00EA2B420>

inner_func_obj()


def outer_func():
    a = 10
    def inner_func():
        print("Inside Inner Function... a value is ", a)
        print("Inner Function Execution Completed...")
    print("Inside Outer Function... a value is ", a)
    print("Outer Function Execution Completed...")
    return inner_func


inner_func_obj = outer_func()
print(inner_func_obj)

inner_func_obj()


def outer_func():
    a = 10
    def inner_func():
        print("Inside Inner Function, a value is : ", a)
        return 100
    print("Inside Outer Function, a value is: ", a)
    return inner_func()


a = outer_func()
print(a)


# Examples of Closures
def multiplier(multiplier_value):
    def inner_func(num):
        return multiplier_value * num
    return inner_func


multiplier_of_3 = multiplier(3)
print(multiplier_of_3(3))
print(multiplier_of_3(9))

multiplier_of_5 = multiplier(5)
print(multiplier_of_5(3))
print(multiplier_of_5(5))


def adder(add_value):
    def inner_fun(value):
        return add_value + value
    return inner_fun


adder_of_10 = adder(10)
print(adder_of_10(7)) # 17
print(adder_of_10(35)) # 45

adder_of_50 = adder(50)
print(adder_of_50(50)) # 100
print(adder_of_50(100)) # 150
```