**Lab Exercise 3- Functional Closures and Decorators in Python**

In this lab, you'll explore **closures** and **decorators** in Python, two powerful functional programming concepts.

**1. Closures in Python**

A **closure** is a function object that remembers values in the enclosing scopes, even if the outer function has finished executing.

**Task 1: Creating a Closure**

**Task**: Write a function multiplier\_of(n) that takes an integer n and returns a new function. The returned function should take another integer x and return the product of x and n.

# Closure function

def multiplier\_of(n):

def multiplier(x):

return x \* n

return multiplier

# Use the closure

times3 = multiplier\_of(3)

times5 = multiplier\_of(5)

# Test the closure functions

print(times3(10)) # Should print 30

print(times5(10)) # Should print 50

**Exercise Breakdown**:

* multiplier\_of(n) creates a closure by returning the multiplier() function.
* The inner function multiplier() remembers the value of n even after multiplier\_of() has finished execution.

**2. Decorators in Python**

A **decorator** is a function that takes another function and extends its behavior without explicitly modifying it.

**Task 2: Creating a Simple Decorator**

**Task**: Create a decorator called print\_info that prints the name of the function being called, along with its arguments and return value.

# Simple decorator function

def print\_info(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling function: {func.\_\_name\_\_}")

print(f"Arguments: {args}, {kwargs}")

result = func(\*args, \*\*kwargs)

print(f"Return value: {result}")

return result

return wrapper

# Example usage of decorator

@print\_info

def add(a, b):

return a + b

@print\_info

def greet(name):

return f"Hello, {name}!"

# Test the decorated functions

print(add(3, 4))

print(greet("Alice"))

**Exercise Breakdown**:

* print\_info is a decorator that adds additional functionality to any function.
* It prints the function name, arguments, and return value.

**3. Decorator with Arguments**

Decorators can also take arguments. This allows them to be more flexible and reusable.

**Task 3: Creating a Decorator with Arguments**

**Task**: Write a decorator called repeat that takes an integer n as an argument. The decorated function should be called n times.

# Decorator with arguments

def repeat(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(n):

result = func(\*args, \*\*kwargs)

return result

return wrapper

return decorator

# Example usage of decorator

@repeat(3) # Will call this function 3 times

def say\_hello():

print("Hello!")

# Test the decorator with arguments

say\_hello()

**Exercise Breakdown**:

* The repeat decorator takes an integer n and calls the decorated function n times.
* This shows how decorators can be customized by passing arguments.

**4. Combining Multiple Decorators**

You can stack multiple decorators on a single function.

**Task 4: Using Multiple Decorators**

**Task**: Combine print\_info and repeat decorators on a single function to demonstrate the combined behavior.

# Combine two decorators

@repeat(2) # First, the function will be repeated 2 times

@print\_info # Then, each call will print information about the function

def multiply(a, b):

return a \* b

# Test the function with combined decorators

multiply(3, 5)

**Exercise Breakdown**:

* First, repeat(2) decorates multiply(), making it repeat twice.
* Then, print\_info is applied, logging the function calls and results for each repetition.