Python Scope & Closures – Zero to Hero Guide

1. What is Scope in Python?

Definition:

Scope refers to **which part of the program can access a variable**.

Types of Scope:

Python uses the **LEGB Rule** for scope resolution:

Scope	Description
L (Local)	Inside the current function
E (Enclosing)	Inside any enclosing function (for nested functions)
G (Global)	At the top-level of the module
B (Built-in)	Python's built-in names (like `len`, `range`, etc)

2. Local vs Global Variables

```
x = 10  # Global variable

def example():
    x = 5  # Local variable
    print(x)

example()   # Output: 5
print(x)    # Output: 10
```

Python uses **the nearest defined variable** by default.

3. The `global` Keyword

To modify a global variable inside a function:

```
count = 0

def increment():
    global count
    count += 1
```

```
increment()
print(count)  # Output: 1
```

⚠ Avoid overusing `global`. It makes code harder to debug 🔧

🎯 4. Nested Functions & Enclosing Scope

```
def outer():
    message = "Hello"

    def inner():
        print(message)

    inner()

outer()
```

✔ `inner()` can access `message` from `outer()` even though it's not defined inside it.

This is **Enclosing Scope**!

🎯 5. The `nonlocal` Keyword

`nonlocal` lets us **modify a variable from the enclosing function**.

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        print("Inner count:", count)

    inner()
    print("Outer count:", count)

outer()
```

📌 Output:

```
Inner count: 1
Outer count: 1
```

---

6. What is a Closure?

A **closure** is a function that **remembers values** from its **enclosing lexical scope**, even if that scope is no longer active.

Closure = Function + Enclosing State

Example of a Closure:

```
def make_multiplier(x):
    def multiplier(n):
        return x * n
    return multiplier

times3 = make_multiplier(3)
print(times3(5))  # Output: 15
```

Even though `x` is gone (function `make_multiplier` has finished), `times3` still **remembers** that `x = 3`. That's closure.

---

7. How to Identify a Closure?

A function must:

1. Be **nested inside another function**
2. **Use a variable** from the enclosing function
3. Be **returned from the outer function**

---

8. Use Cases of Closures

Use Case	Description
Encapsulation	Hide variables from global scope
Callback functions	Useful in decorators, UI events
Factory functions	Functions that return customized functions

Real-life Analogy:

Imagine a chef (inner function) trained in a specific kitchen (outer function). Even if the kitchen closes, the chef retains the recipe. That recipe is the closure.

---

9. Advanced Closure Behavior

Closures maintain **state** without using classes:

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter()
print(c())  # 1
print(c())  # 2
```

Each call remembers the updated `count`. That's closure magic ✨

🚫 Common Pitfall

❌ Problem:

```
def bad_closures():
    funcs = []
    for i in range(3):
        def inner():
            return i
        funcs.append(inner)
    return funcs

closures = bad_closures()
print([f() for f in closures])  # ❌ Output: [2, 2, 2]
```

☑️ Fix using default arguments:

```
def good_closures():
    funcs = []
    for i in range(3):
        def inner(i=i):  # 🛠️ capture i at that time
            return i
        funcs.append(inner)
    return funcs

closures = good_closures()
print([f() for f in closures])  # ☑️ Output: [0, 1, 2]
```

🧠 10. Summary (Zero ➡️ Hero)

Concept	Learnings
🔍 Scope	LEGB Rule: Local → Enclosing → Global → Built-in
🧹 Local	Variable inside function
🌍 Global	Variable outside all functions
🧩 Enclosing	For nested functions
🔓 `global`	Modify global variable from inside
🔄 `nonlocal`	Modify enclosing variable in nested function
🧠 Closure	Function + enclosed state
🛠️ Use	Factory functions, decorators, encapsulation

🧬 Bonus: Python `inspect` closure

To inspect closures:

```
def outer():
    x = 10
    def inner():
        return x
    return inner

func = outer()
print(func.__closure__[0].cell_contents)  # Output: 10
```

☑️ Final Thought:

Closures are **powerful**, **clean**, and **memory-efficient** tools in Python 🔥 They allow you to write flexible code without needing to create classes every time 🧰
