Local & Global Scope in Depth

AiCE Software Engineering Foundations — Python Programming (Intermediate)

The previous unit introduced scope and return values briefly while teaching functions. This lesson examines both in depth — because a complete understanding of how data flows into and out of functions is what separates programs that work from programs that are reliable, predictable, and safe to build on.

1. Revisiting Scope — A Precise Definition

Scope is the set of rules that determines where in your program a variable can be seen and used. Python enforces these rules strictly — use a variable outside its scope, and you get a NameError. If two scopes contain variables with the same name, Python follows a specific priority order to decide which one to use.

There are two fundamental scopes to master at this level:

Local scope — variables created inside a function. They exist only for the duration of that function call.

Global scope — variables created outside all functions. They exist for the entire lifetime of the program.

> **The Classroom and the Billboard**
> *Anything written on a whiteboard inside a classroom is only visible to people inside that classroom — the moment they leave, they can no longer see it. Anything written on a public billboard outside is visible to everyone. In Python, a function is the classroom. Local variables are on the whiteboard. Global variables are on the billboard.*

2. Local Scope — Variables That Live and Die Inside Functions

```
def my_function():
    x = 10
```

```
    print(f"Inside function : x = {x}")

my_function()
print(f"Outside function: x = {x}")    # NameError
```

This produces:

```
Inside function : x = 10
NameError: name 'x' is not defined
```

x was created inside my_function — it's local to that function. When my_function finishes, x is destroyed; it no longer exists anywhere in memory. The print outside the function finds nothing and raises a NameError.

> **Isolation Is Safety, Not a Bug**
> *Local variables are intentionally isolated — they cannot leak out of their function. This is what makes functions safe to use: you can write a function knowing its internal variables will never accidentally interfere with the rest of your program.*

3. Global Scope — Variables Visible Everywhere

```
x = 100    # Global variable

def my_function():
    print(f"Inside function : x = {x}")

my_function()
print(f"Outside function: x = {x}")
```

This produces x = 100 both times — both the function and the outer code read the global variable freely, with no special declaration needed to read it.

4. When Local and Global Names Are the Same

This is where most scope confusion originates. What happens when a local variable inside a function shares a name with a global one?

```
x = 100    # Global

def my_function():
    x = 10    # Local — completely separate from the global x
    print(f"Inside function : x = {x}")

my_function()
print(f"Outside function: x = {x}")
```

This produces Inside function: x = 10, then Outside function: x = 100. Inside the function, the local x takes priority. Outside, the global x is completely unchanged — two entirely separate variables that happen to share a name.

> **Variable Shadowing**
> *This is called variable shadowing — the local variable shadows the global one inside the function. Many beginner bugs come from accidentally shadowing a global variable. The fix is simple: use distinct, descriptive variable names so this situation never arises.*
>
> **Same Name, Different Variable**
> *When you assign to a variable inside a function, Python always creates a local variable — even if a global with the same name exists. It does not modify the global. To modify the global, you must explicitly declare it with the global keyword.*

5. The global Keyword

The global keyword tells Python that a variable name inside a function refers to the global variable, not a new local one. Without it, any assignment inside a function creates a local variable:

```
x = 100

def my_function():
    global x    # Tell Python: use the global x
    x = 10      # This now modifies the global x
    print(f"Inside function : x = {x}")
```

```
my_function()
print(f"Outside function: x = {x}")
```

This produces x = 10 both times — with global declared, the assignment inside the function modifies the actual global variable, changing it from 100 to 10 for everyone.

A Practical global Example — Enrollment Counter

```
# Tracking total enrollments across function calls
total_enrolled = 0

def enroll(student_name):
    global total_enrolled
    total_enrolled += 1
    print(f"Enrolled: {student_name} (Total:
{total_enrolled})")

enroll("Amara Diallo")
enroll("Kofi Mensah")
enroll("Ngozi Adeyemi")

print(f"Final enrollment count: {total_enrolled}")
```

The global keyword lets total_enrolled accumulate across multiple calls. Without it, each call would increment a local variable starting at zero, and the global counter would never actually change.

> **Ask First: Could a Return Value Work Instead?**
> *Use global sparingly and deliberately. When you reach for the global keyword, ask yourself first: could I solve this with a return value instead? Usually, the answer is yes — and that's the better solution.*

Key Takeaways

A local variable is born and destroyed within its own function call and can never be seen outside it; a global variable, declared outside every function, is visible everywhere. When a local variable shares a name with a global one, the local always wins inside its function

— without ever touching the global — unless the global keyword explicitly says otherwise. Reach for global sparingly; a return value is almost always the better-designed solution. The next lesson completes the picture with the LEGB rule that governs every name lookup in Python, and a full, precise treatment of return values.