The LEGB Rule & Return Values in Full

AiCE Software Engineering Foundations — Python Programming (Intermediate)

With local and global scope established, this lesson completes the picture with the full rule Python uses to resolve every variable name, then returns to the return statement for a complete treatment — None, dead code, early returns, and the professional patterns that make functions genuinely reliable.

1. The LEGB Rule — Python's Full Scope Priority

Python resolves variable names using a priority order called the LEGB rule. When Python encounters a name, it searches these scopes in order and uses the first match it finds:

Priority	Scope	Description
1st (highest)	Local (L)	Inside the current function
2nd	Enclosing (E)	Inside any enclosing function (for nested functions)
3rd	Global (G)	At the top level of the current file
4th (lowest)	Built-in (B)	Python's built-in names: print, len, range, etc.

For the work in this program, Local and Global are the two scopes you'll work with constantly. Enclosing scope applies to nested functions — functions defined inside other functions — which you'll encounter in more advanced Python. Built-in scope is where Python's own functions live: print, len, range, type, input, and everything else you've used throughout the beginner section.

> **Why print() Just Works, Everywhere**
> *This is why you can use print() anywhere without importing it — it lives in the built-in scope, which Python always searches last as a fallback, after checking local, enclosing, and global first.*

2. Return Values — A Complete Treatment

What Happens When There Is No Return Statement

Every Python function returns something. If you don't write a return statement, Python automatically returns a special value called None.

None is Python's representation of nothing — the absence of a value. It is not zero, not an empty string, not False. It's a distinct value of its own type (NoneType) that simply means: this function produced no result.

```
def add(num1, num2):
    result = num1 + num2
    # No return statement

answer = add(10, 5)
print(f"The answer is: {answer}")
print(type(answer))
```

This produces:

```
The answer is: None
<class 'NoneType'>
```

The function computed result but never returned it, so Python returned None automatically. If you ever see None where you expected a computed value, the most common cause is a missing return statement.

> **None Is Not an Error**
> *None is a legitimate Python value, used deliberately in many situations — as a default return value when a function has nothing meaningful to return, or as a sentinel indicating something wasn't found. The mistake is expecting a value and getting None instead, usually caused by forgetting return.*

return Terminates the Function Immediately

The moment Python hits a return statement, the function ends — no exceptions. Any code written after return inside the function body is permanently unreachable:

```
def add(num1, num2):
    result = num1 + num2
    return result
    print("This line will never run.")   # Dead code
```

> **Dead Code**
> *The unreachable print statement after return is called dead code — Python never reaches it. Most professional code editors and linters warn about dead code, since it usually indicates a logic error or a misplaced return statement.*

Early Returns — Guard Clauses

One of the most powerful uses of return is the early return — checking for invalid or edge-case conditions at the top of a function and returning immediately, before the main logic runs:

```
def divide(num1, num2):
    # Guard clause – check for invalid input first
    if num2 == 0:
        return "Error: Cannot divide by zero."
    # Main logic – only runs if the guard passes
    result = num1 / num2
    return result

print(divide(10, 2))
print(divide(10, 0))
```

For divide(10, 0), the guard clause fires immediately and the function returns an error message — the division on the next line never executes. Compare this to wrapping the main logic in an else block instead: both work, but the guard clause keeps the main logic at the outermost indentation level, making the function easier to read as it grows more complex.

3. Returning Multiple Values

A function can return more than one value separated by commas. Python packs them into a tuple automatically, and you unpack them on the receiving side:

```
def get_min_max(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    return minimum, maximum    # Returns a tuple

scores = [85, 72, 90, 68, 95, 78]
lowest, highest = get_min_max(scores)
print(f"Lowest score : {lowest}")
print(f"Highest score: {highest}")
```

lowest, highest = get_min_max(scores) unpacks the returned tuple into two separate variables at once, matched positionally.

Returning Mixed Types — The Result-or-Error Pattern

A common professional pattern returns a result alongside a status or error message, so the caller always knows whether the function succeeded:

```
def safe_divide(num1, num2):
    if num2 == 0:
        return None, "Division by zero is not allowed."
    return num1 / num2, None    # Result, no error

result, error = safe_divide(100, 4)
if error:
    print(f"Error: {error}")
else:
    print(f"Result: {result}")
```

The function always returns two values: the result and the error. If the division succeeds, error is None; if it fails, result is None. The caller checks which one is present and responds accordingly — this result-or-error pattern is used extensively in professional Python, Go, and many other languages.

Key Takeaways

Python resolves every variable name using LEGB — Local, Enclosing, Global, Built-in — checked strictly in that order. A function with no return statement silently returns None, and code placed after a return is dead code that never runs. Guard clauses handle invalid input at the top of a function and return immediately, keeping the main logic clean, while the result-or-error pattern — returning a value alongside None, or None alongside an error message — is how professional functions communicate success or failure. The next lesson introduces pure functions, the gold standard for reliable, testable code, and brings scope and return values together in a complete health analysis program.