Scope Basics, Function Composition & Working With Collections

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on functions introduces where a variable lives and who can see it, shows functions working together to build more complex behavior from simple parts, and applies everything to the lists and dictionaries from the beginner section — exactly the kind of data real functions are built to process.

1. Local and Global Variables — A First Look

When you create a variable inside a function, it only exists for the duration of that function call — once the function finishes, the variable is gone. This is a local variable, and the region where it exists is its local scope.

A variable created outside all functions exists for the entire lifetime of the program and is accessible from anywhere, including inside functions. This is a global variable, living in the global scope.

```
def my_function():
    message = "I am inside the function"
    print(message)

my_function()
print(message)    # NameError — message does not exist here
```

The first print, inside the function, works. The second, outside, raises a NameError — message is local to my_function, born when the function started and gone when it ended.

```
platform = "AiCE"

def show_platform():
    print(f"You are learning on the {platform} platform.")

show_platform()
```

```
print(f"Platform name: {platform}")
```

platform is global — defined outside any function — so both the function and the outer code can access it freely.

When Local and Global Names Collide

If a local variable inside a function shares a name with a global variable, the local one takes priority inside the function, without affecting the global at all:

```
score = 100    # global

def adjust_score():
    score = 75    # local — a completely separate variable
    print(f"Inside function: {score}")

adjust_score()
print(f"Outside function: {score}")
```

This produces Inside function: 75, then Outside function: 100 — two entirely separate variables that happen to share a name.

> **A Fuller Treatment Is Coming**
> *This is a first look at scope — enough to avoid the most common surprises. The next unit examines scope with full precision, including the global keyword for deliberately modifying a global variable from inside a function, and the complete LEGB rule that governs how Python resolves every name.*

2. Functions Calling Other Functions

Functions become truly powerful when they work together. One function can call another, building complex behavior from simple, focused building blocks — a technique called function composition:

```
# Each function has one clear, focused responsibility
def get_grade(score):
    if score >= 70:
        return "A"
    elif score >= 60:
        return "B"
    elif score >= 50:
        return "C"
    else:
        return "F"

def get_remark(grade):
    remarks = {
        "A": "Distinction — Outstanding performance.",
        "B": "Credit — Very good performance.",
        "C": "Pass — Satisfactory performance.",
        "F": "Fail — Please re-sit the assessment.",
    }
    return remarks.get(grade, "Unknown grade.")

def print_result(name, score):
    grade = get_grade(score)
    remark = get_remark(grade)
    print(f"{name} — Score: {score}, Grade: {grade}")
    print(f"  {remark}")

students = [("Adesina Olagunju", 85), ("Chidi Eze", 45),
("Amara Diallo", 92)]

for name, score in students:
    print_result(name, score)
```

Three functions, each with one responsibility, work together: get_grade() receives a score and returns a letter; get_remark() receives a letter and returns a remark; print_result() calls both and formats the output. If the grading scale changes, you update get_grade() alone — nothing else needs to touch.

One Job Per Function

Write functions that do one thing. Name them after what they do. Combine them to build systems. A program made of well-named, single-purpose functions is a program anyone can read, anyone can fix, and anyone can extend.

3. Functions With Lists and Dictionaries

Functions handle any Python data type, including lists and dictionaries — making them ideal for processing collections of records:

```
# AiCE cohort processing functions
def calculate_average(scores):
    """Calculate the average of a list of scores."""
    if len(scores) == 0:
        return 0
    return sum(scores) / len(scores)

def filter_passed(students, passing_mark=50):
    """Return a list of students who passed."""
    return [s for s in students if s["score"] >= passing_mark]

def print_report(students):
    """Print a formatted report for a list of students."""
    scores = [s["score"] for s in students]
    avg = calculate_average(scores)
    passed = filter_passed(students)

    for student in students:
        grade = "Pass" if student["score"] >= 50 else "Fail"
        print(f'{student["name"]:<20} | {student["score"]:>3}
| {grade}')

    print(f"Average: {avg:.2f}")
    print(f"Passed : {len(passed)} / {len(students)}")

cohort = [
    {"name": "Amara Diallo",  "score": 88},
    {"name": "Kofi Mensah",   "score": 45},
    {"name": "Ngozi Adeyemi", "score": 92},
```

```
]

print_report(cohort)
```

The program processes student records through coordinated functions, each doing one job, and produces a complete formatted report. The triple-quoted strings just beneath each def line are docstrings — they document what a function does, and writing them is standard professional practice.

> **filter_passed() Uses a List Comprehension**
> *The line [s for s in students if s["score"] >= passing_mark] is called a list comprehension — a compact way of building a filtered list in one line. It's covered fully later in this program; for now, read it conceptually: build a new list containing only the students whose score meets the passing mark.*

Key Takeaways

A local variable lives and dies inside the function that creates it; a global variable is visible everywhere, and when the two share a name, the local one wins inside its own function without touching the global. Functions calling other functions — function composition — is how complex, reliable behavior gets built from small, focused, well-named pieces, each with a single responsibility. Functions work on any data type, including the lists and dictionaries from the beginner section, and docstrings are the professional standard for documenting what each one does. With functions established as the foundation, the next unit returns to scope and return values in full depth — the precision that separates code that works from code that's genuinely reliable.