Pure Functions & Building a Complete Health Analyser

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on scope and return values introduces pure functions — the gold standard for reliable, testable code — then brings every idea from this unit together into a complete, interactive health analysis program.

1. Pure Functions — The Gold Standard

A pure function is a function that:

Always returns the same output for the same input.

Has no side effects — it doesn't modify global variables, print, write to files, or change anything outside itself.

Pure functions are the most predictable, testable, and reliable type of function. They're completely self-contained — you can reason about them in isolation without needing to know anything about the rest of the program.

```
# Pure function – same input always gives same output, no side
effects
def calculate_discount(price, discount_percent):
    discount_amount = price * (discount_percent / 100)
    final_price = price - discount_amount
    return final_price

# Impure function – modifies a global variable (side effect)
total_discounts_given = 0

def apply_discount(price, discount_percent):
    global total_discounts_given
    discount_amount = price * (discount_percent / 100)
    total_discounts_given += discount_amount    # Side effect!
    return price - discount_amount
```

Both functions produce the same result for the customer, but calculate_discount is pure — no side effects, testable in complete isolation. apply_discount modifies a global variable, a side effect that makes it harder to test, harder to reuse, and harder to reason about.

> **Isolate Side Effects Deliberately**
> *Aim to write pure functions wherever possible. When side effects are genuinely necessary — writing to a file, updating a database, printing output — isolate them in dedicated functions and keep your computation functions pure.*

2. Bringing It All Together — The AiCE Health & BMI Analyser

A capstone integrating scope discipline, guard clauses, multiple return values, and the result-or-error pattern into a real interactive health tool:

```
# AiCE Health Analyser
def calculate_bmi(weight_kg, height_m):
    """Calculate BMI. Returns (bmi, error_message)."""
    if weight_kg <= 0 or height_m <= 0:
        return None, "Weight and height must be greater than
zero."
    if height_m > 3.0:
        return None, "Height seems unrealistic. Please enter
height in metres."
    bmi = weight_kg / (height_m ** 2)
    return round(bmi, 2), None

def get_bmi_category(bmi):
    """Return the WHO BMI category for a given BMI value."""
    if bmi < 18.5:
        return "Underweight", "Consider consulting a
nutritionist."
    elif bmi < 25.0:
        return "Normal weight", "Your BMI is in the healthy
range."
    elif bmi < 30.0:
```

```
        return "Overweight", "Consider increasing physical
activity."
    else:
        return "Obese", "Please consult a healthcare
provider."

def run_health_check():
    """Main function — collects input and orchestrates the
analysis."""
    name = input("Name: ").strip().title()
    weight = float(input("Weight (kg): "))
    height = float(input("Height (m): "))

    # Calculate BMI with guard clause protection
    bmi, error = calculate_bmi(weight, height)
    if error:
        print(f"Error: {error}")
        return    # Early return from main function too

    category, advice = get_bmi_category(bmi)

    print(f"Name    : {name}")
    print(f"BMI     : {bmi}")
    print(f"Category: {category}")
    print(f"Advice  : {advice}")

run_health_check()
```

Test this twice — once with valid inputs, and once with invalid ones (a weight of 0, or a height of 5 metres). The guard clauses catch invalid input gracefully. The program uses several functions that each do one thing, communicate through return values, and never touch global state.

> **Isolation Is Safety**
>
> *The discipline of scope is the discipline of thinking clearly about what belongs where. A variable that only exists for the duration of a function is a variable that cannot cause unexpected damage anywhere else. That isolation is not a limitation — it is safety.*

3. Scope and Return — Quick Reference

Concept	Rule	Common Mistake
Local variable	Created inside a function — only visible there	Trying to access it outside the function
Global variable	Created outside functions — visible everywhere	Accidentally shadowing it with a local of the same name
global keyword	Declares intent to modify a global from inside a function	Forgetting it, causing an UnboundLocalError
return	Sends a value back and immediately ends the function	Forgetting return — function returns None silently
Multiple returns	return a, b returns a tuple — unpack with a, b = func()	Not unpacking — receiving a tuple instead of two values
Dead code	Code after a return statement — never executed	Placing logic below a return, expecting it to run

Key Takeaways

A pure function always returns the same output for the same input and has no side effects — no modified globals, no printing, no file writes — which makes it the most testable, predictable kind of function you can write. The Health Analyser demonstrates every idea from this unit working together: guard clauses catching bad input early, multiple return values carrying both a result and an error, and functions that stay pure by returning data instead of reaching for global state. With scope and return values fully understood, the next unit introduces modules and import — the mechanism for bringing pre-built Python tools into your programs and organizing your own code across multiple files.