AIDD 30-Day Challenge — Task 8

Spec-Driven Python CLI Calculator (Using SpecKitPlus)

This project follows all **five SDD phases**:

- `/constitution`
- `/specify`
- `/plan`
- `/tasks`
- `/implement`

All required screenshots and code files are included in the repository.

✨ Constitution Phase

Prompt:

Write a short Constitution for a Python CLI Calculator using SpecKitPlus. Mention project purpose, learning goals, and key principles of SDD.

Claude Output:

Project Constitution — Python CLI Calculator

Purpose

The goal of this project is to build a simple and reliable calculator using Python and the five phases of SpecKitPlus. It trains me to follow Spec-Driven Development (SDD), where planning comes before coding.

Learning Goals

- Understand all 5 SpecKitPlus phases
- Learn how to define requirements before writing code
- Increase confidence with AI-assisted coding
- Build a clean and testable calculator
- Document everything properly using markdown

Scope

- Accept **two numeric inputs**
- Perform **Addition, Subtraction, Multiplication, Division**
- Prevent **division-by-zero errors**
- Display results clearly in the terminal

Core Principles

- Clarity over complexity
- Step-by-step SDD workflow
- Reliability + error handling
- Testability of every operation

Success Criteria

- All four operations work correctly
- No unexpected crashes
- Inputs are validated
- All SDD phases documented properly

✨ Specify Phase

Prompt:

Write clear specifications for a Python CLI calculator using SpecKitPlus.

Claude Output:

📌 Specification — Calculator Requirements

1. Functional Requirements

- Accept two numeric inputs
- Allow one operation choice:
  - Addition
  - Subtraction
  - Multiplication
  - Division
- Display output clearly

2. Input Requirements

- Accept integers and decimals
- Reject non-numeric input
- Ask again if invalid

3. Operation Rules

- `a + b`
- `a - b`
- `a * b`
- `a / b` (only if $b \neq 0$)

4. Output Requirements

Print results like:

Result: 7 + 3 = 10

5. Error Handling Requirements

- Handle division by zero
- Handle invalid numeric input
- Handle invalid operation choice
- No program crash allowed

6. User Experience Requirements

- Simple
- Fast
- Clean messages
- Program exits safely

7. Success Conditions

- All operations produce correct results
- Invalid inputs handled nicely
- Output readable
- Matches specification

✨ Plan Phase

Prompt:

Create a technical plan describing program structure, flow, functions, and testing.

Claude Output:

🛠️ Technical Plan — Calculator Architecture & Flow

1. Program Structure

1. Ask for first number
2. Ask for second number
3. Show menu
4. Perform calculation
5. Display result
6. Exit

2. Core Components

- **Input Handler** (reads numeric values)
- **Operation Selector** (menu)
- **Calculator Functions**
  - add()
  - subtract()
  - multiply()
  - divide()
- **Output Formatter**

3. Calculation Flow

- Convert inputs to float
- Match operation
- Perform calculation
- Handle errors
- Display full formatted expression

4. Error Handling Plan

- Invalid number → error message
- Invalid menu choice → error message
- Division by zero → block operation

5. Testing Plan

Operation	Example	Expected
Addition	10 + 5	15
Subtraction	20 – 4	16
Multiplication	6 × 7	42
Division	12 ÷ 3	4

6. Output Format

Result: <num1> <operator> <num2> = <answer>

7. Completion Criteria

- All operations correct
- No crashes
- Safe input handling
- Code matches plan

✨ Tasks Phase

Prompt:

Break the work into a small task list based on SpecKitPlus.

Claude Output:

📋 Task List (SDD Tasks)

1. Create project folder
2. Write constitution
3. Write specifications
4. Write technical plan
5. Create calculator.py
6. Add functions (add, subtract, multiply, divide)
7. Add input validation
8. Add menu selection logic
9. Add formatted output
10. Run 4 tests
11. Capture screenshots
12. Finalize README.md

13. Push to GitHub

✨ Implement Phase

Prompt:

Using all previous phases, write final Python implementation.

Claude Output:

💻 Final Implementation — calculator.py

```
def get_number(label: str) -> float:
    """Ask the user for a number and convert it to float."""
    try:
        value = float(input(f"Enter {label} number: "))
        return value
    except ValueError:
        print("Error: Please enter a valid numeric value.")
        raise SystemExit(1)



def add(a: float, b: float) -> float:
    return a + b



def subtract(a: float, b: float) -> float:
    return a - b
```

```
def multiply(a: float, b: float) -> float:
    return a * b



def divide(a: float, b: float) -> float:
    if b == 0:
        print("Error: Cannot divide by zero.")
        raise SystemExit(1)
    return a / b



def main() -> None:
    print("=== Spec-Driven Python CLI Calculator ===")

    # Step 1: Read inputs
    num1 = get_number("the first")
    num2 = get_number("the second")

    # Step 2: Show menu
    print("\nChoose operation:")
    print("1. Addition (+)")
    print("2. Subtraction (-)")
    print("3. Multiplication (*)")
    print("4. Division (/)")
```

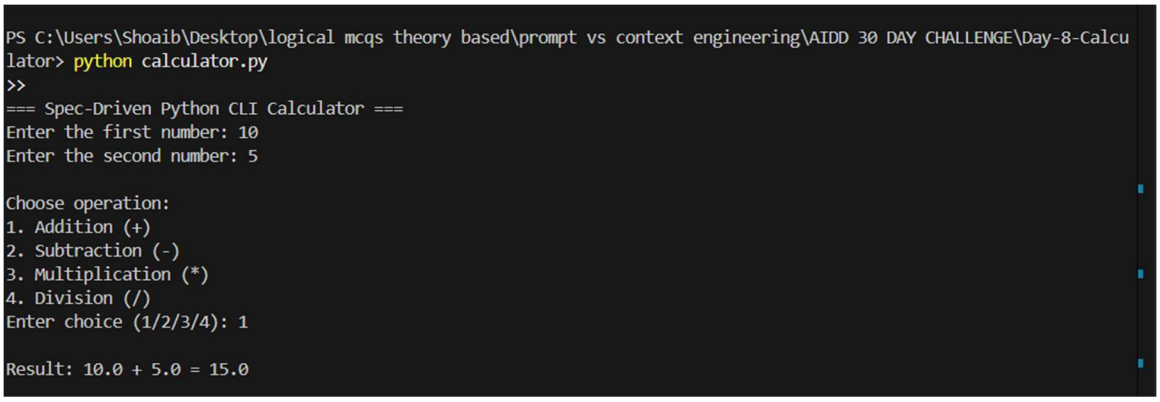
```
    choice = input("Enter choice (1/2/3/4): ").strip()

    # Step 3: Map choice to operation
    if choice == "1":
        operator = "+"
        result = add(num1, num2)
    elif choice == "2":
        operator = "-"
        result = subtract(num1, num2)
    elif choice == "3":
        operator = "*"
        result = multiply(num1, num2)
    elif choice == "4":
        operator = "/"
        result = divide(num1, num2)
    else:
        print("Error: Invalid operation choice.")
        raise SystemExit(1)

    print(f"\nResult: {num1} {operator} {num2} = {result}")
    print("\nThank you for using the Spec-Driven Calculator!")


if __name__ == "__main__":
    main()
```

✨ Final Testing Screenshots

All four calculator operations were tested successfully.

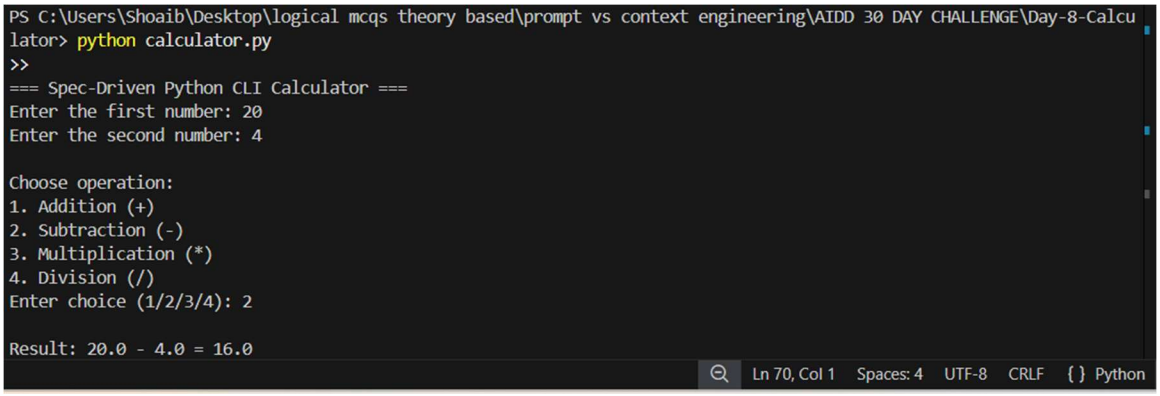
Addition Test

`screenshots/addition.png`



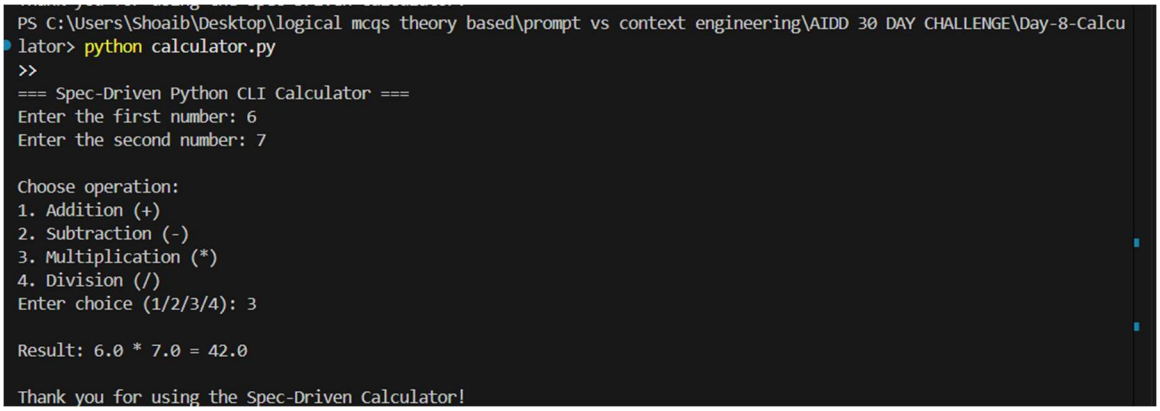
Subtraction Test

`screenshots/subtraction.png`



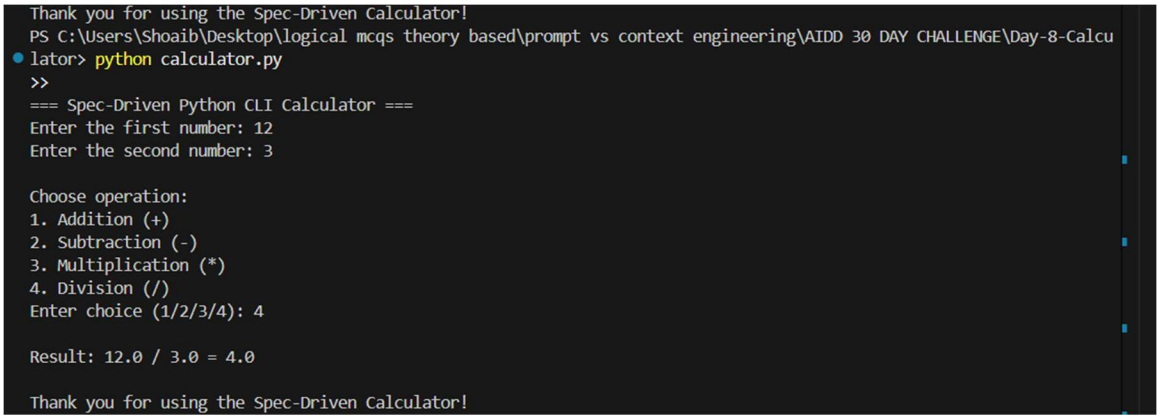
Multiplication Test

`screenshots/multiplication.png`



Division Test

`screenshots/division.png`



Submission Note

The calculator strictly follows all five SDD phases and demonstrates proper Spec-Driven Development.