Simple Calculator – Process Summary

Justice Williamson

Southern New Hampshire University

CS-500: Introduction to Programming

Instructor: Ronak Nouri

October 12, 2025

Introduction

This process summary explains how I planned, built, and verified a menu-based calculator for the Module Seven Activity of building a simple calculator. The calculator performs addition, subtraction, multiplication, and division, rejects invalid menu choices, validates numeric input, and prevents division by zero. I met the assignment's planning and quality requirements by creating a flowchart, writing pseudocode, and using both automated tests and manual runs. Per the instructions, I am not including source code—only pseudocode and screenshots of console output.

Approach and Strategy

Decomposition. I split the work into two clear layers:

• **Computation layer:** small, single-purpose arithmetic operations that take two numbers and return a result.

• **User-interface layer:** a loop that displays the menu, reads and validates input, handles errors, and prints results.

Plan first. I created a flowchart to map the loop and decision points (including the divide-by-zero path), then wrote pseudocode that follows the diagram. These artifacts helped me avoid branching mistakes and made edge cases easy to reason about.

Validation & resilience. The program keeps prompting until the user enters valid numbers and shows a clear message for invalid menu choices. If the user selects Divide and the second number is 0, the program displays an error and returns to the main menu so the session can continue.

Evidence of quality. I used automated tests to check the math, floating-point behavior, and error responses. I also ran the program manually to confirm the interactive experience.

Design Artifacts

Flowchart (summary).

Start → Display menu → *Valid choice?* → (No → Show "Invalid choice" → back to menu) → (Yes → Input number A → Input number B → *Operation = Divide and B = 0?* → Yes → Show error → back to menu; No → Perform operation → Display result → *Another calculation?* → Yes → back to menu; No → End)

Pseudocode:

```
START
LOOP
  DISPLAY menu: 1) Add  2) Subtract  3) Multiply  4) Divide  5) Exit
  GET choice
  IF choice == 5 THEN
    DISPLAY "Goodbye."
    BREAK
  ENDIF

  IF choice NOT IN {1, 2, 3, 4} THEN
    DISPLAY "Invalid selection."
    CONTINUE
  ENDIF

  PROMPT and READ first number until valid → a
  PROMPT and READ second number until valid → b

  IF choice == 4 AND b == 0 THEN
    DISPLAY "Error: Cannot divide by zero."
    CONTINUE
  ENDIF

  result ← APPLY selected operation to (a, b)
  DISPLAY "Result: ", result
ENDLOOP
END
```

Issues Encountered and Resolutions

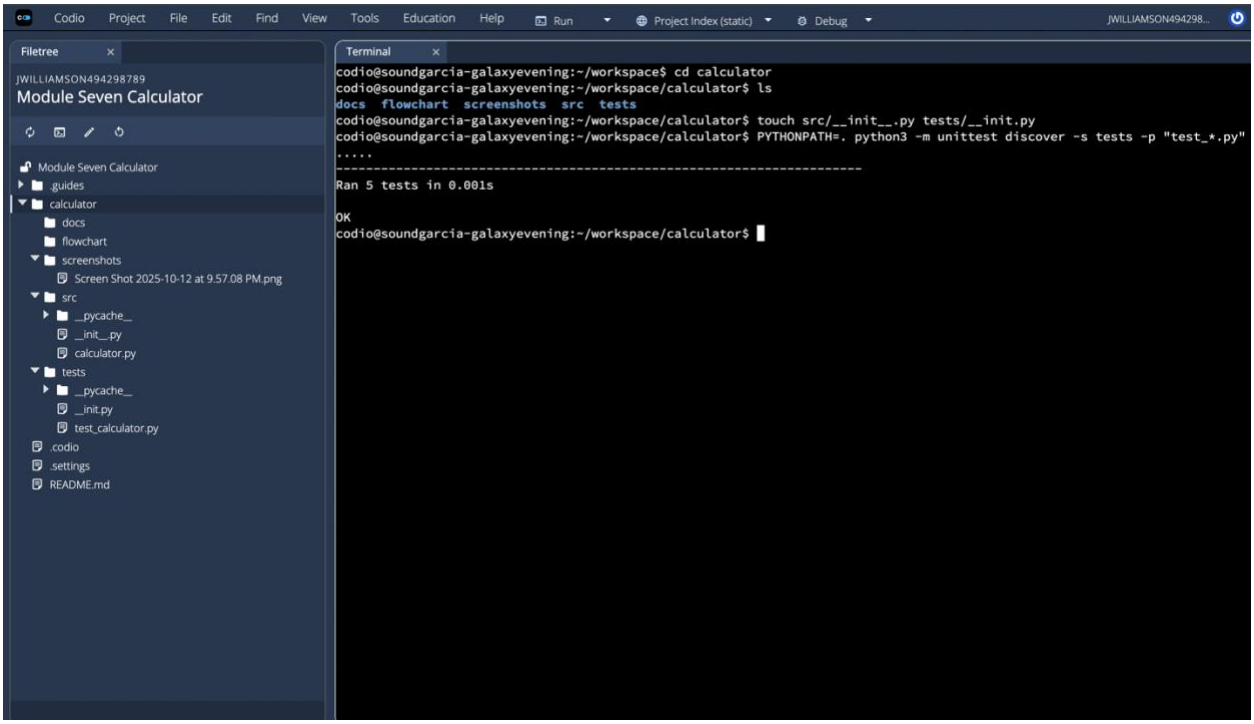
- **Invalid numeric input.** Non-numeric entries caused conversion errors. I added a validation loop that keeps prompting until a valid number is provided, which prevents crashes.
- **Division by zero.** I added a decision before dividing: if the operation is Divide and b == 0, show an error and return to the menu.
- **Test discovery/import paths (Codio).** I added __init__.py files to src/ and tests/ and ran tests with PYTHONPATH=. -so, the test runner consistently finds modules.
- **Flowchart clarity.** I added **Input number B** between **Input number A** and the divide check to reflect that operations use two operands.

Testing Strategy and Results

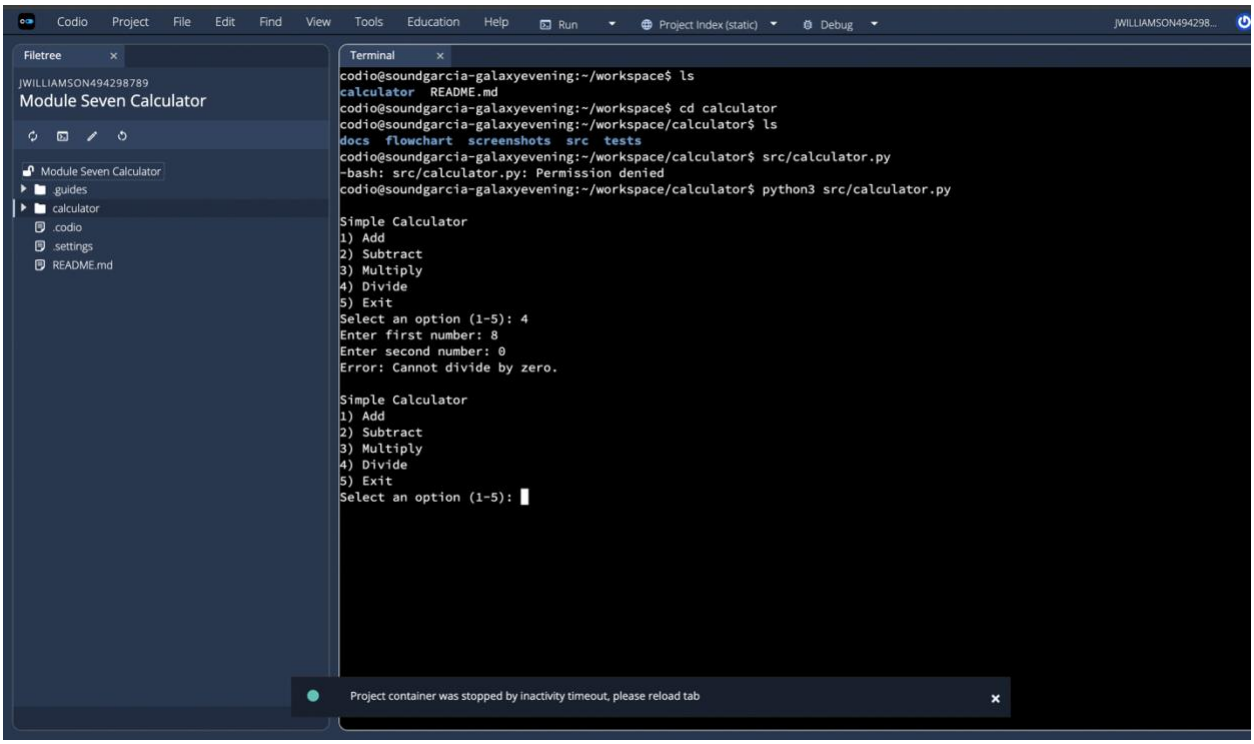
Automated tests. I verified that addition, subtraction, multiplication, and division return correct results, used approximate checks for floating-point division, and confirmed that division by zero triggers the error path.

Manual runs. I confirmed that invalid choices are rejected, valid operations print the correct results, and division by zero shows the error message and returns to the menu.

Screenshot 1 — Unit tests “OK”.



Screenshot 2 — Division-by-zero error message.



Optimization Opportunities (Performance & Maintainability)

- Use a **dispatch map** to route menu choices to functions instead of long if/elif chains.
- **Centralize prompts and messages** so updates are consistent and easy.
- Add **retry limits** and optional **bounds checks** to guard against endless invalid input.
- Keep computation **fully separate from I/O** so the same logic can be reused by a GUI or web interface without changes.

Reflection and Conclusion

Planning with the flowchart and pseudocode reduced bugs and rework. Separating the math from the user interface made the program easier to test and understand. Automated tests gave me confidence in correctness, and manual runs confirmed the user experience.