# 7-2 Activity: Building a Simple Calculator

## Process Summary

Marie-Chantal Foster

Cybersecurity, Southern New Hampshire University

CS-500-10018-M01 Introduction to Programming

Dr. Shadha Tabatabai

January 4, 2026

# Process Summary- Basic Calculator

When tackling a project with many moving parts, I have two approaches: my checklist method and colored-coded blocking method. The checklist method allows me to turn every requirement into a checkbox, provides a visual map to stay organized, and ensures nothing is overlooked. I combine that with a color-coded blocking method to help reduce overwhelm, then break the assignment into manageable sections. Each block is assigned a different color, and once a section is completed, I change it back to the standard black text to visually signal progress. I began by color-coding the blocks I needed to complete, then created a checklist with a timeline for each task. For example, in Part One, I generated a flowchart, marked it in orange, and set a goal to complete it by Tuesday. This included creating the flowchart in Visio, building it there, and, if needed, scheduling assistance to verify my work. I followed this same method for all other sections of the assignment. Throughout the process, I researched the required materials and sought guidance from SNHU academic support. I worked through each block one at a time, checking items off as I completed them. Occasionally, I revisit earlier blocks to refresh my memory, which helps me stay on track and better understand the next steps.

When I began the flowchart, I relied on reading resources to sketch out the first steps on paper. The basic flowchart steps consisted of a start, input your numbers, which operation to use, such as add, subtract, multiply, or divide, but if you divide by zero, an error will occur; then, whether you want to do another operation, if not, then end and exit the program. Although the flow was simple enough, it was placing the symbols in the right places to make the flow chart make sense. I used the true/false method to determine the direction of flow in my flowchart. Each decision diamond evaluated a condition and sent the user down one of two paths. For example, if the user selects the “+” operation, the decision checks whether the input matches

“+”. If the condition is true, the flow continues to the addition process and moves to the next step, such as asking whether the user wants another calculation. If the condition is false, the flow moves to the next decision to check for “-”, “*”, or “?”, and will continue this pattern until the correct operation is found. This ensures that every path eventually leads either to performing the chosen operation or exiting the program. I transferred the sketch into Visio to create the final flowchart. However, I hit a wall on the placement of three decision items: “Do another calculation?” symbol, the “divided by zero” error, and the “invalid operation” error, all of which would present an error or prompt to do another calculation before leaving the program. After several attempts, I reached out to the tutor for guidance, which helped me determine where to place the decision symbols in the flowchart. Check! I could move on to the pseudocode step.

Pseudocode is new to me. I have heard of the concept but have never explored it to understand how it works. I had to refer to external resources to understand the meaning and how to use it. The pseudocode was written as a loop-based calculator that would set up the logic for repeated calculations. For example, the pseudocode shows where I set up variables to store numbers and the result, then initialize the loop with a `While` loop that keeps running as long as the user says yes, and then asks the user what math operation to use. Also, I understood that it was an intermediate state between an idea and the implementation of writing code. After several iterations of creating pseudocode, I sought out guidance to see if I was on track. The original version showed that I repeated myself throughout the pseudocode block; therefore, the tutor pointed out that this was not an optimal way to execute code and didn’t follow the DRY principle. I rewrote the code to read more clearly and simply. Final version, check!

```
Start

Declare n1, n2, result as a float
SET Choice to "yes" # initialize choice in loop
```

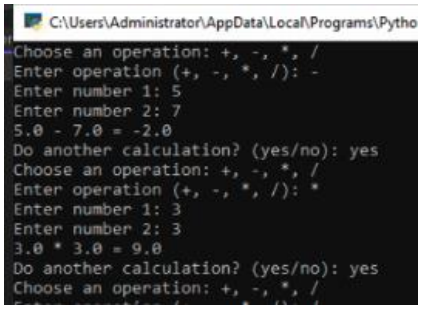
```
While choice =="yes"
      output "choose an operation: (+, -. *, /)
      input operation

      output "input first and second number: n1, n2"
      input n1, n2

      If operation == '+' then
            Set result to n1+n2
            Output n1, '+', n2 '=', result
      Else if operation =='-' then
            Set result to n1-n2
            Output n1, '-', n2 '=', result
      Else if operation =='*' then
            Set result to n1* n2
            Output n1, '*', n2, '=', result
      Else if operation =='/' then
            If n2==0 then
                  Output "Error! Cannot divide by zero.”
            Else
                  Set result  to n1 / n2 then
                  Output n1, '/', n2, '=' , result
            End If
            Output " Do another calculation? (yes/ no)."
            Input choice
End While
Output "exit program."
```

The pseudocode was a great stepping stone for writing my initial calculator code. Throughout the code-writing process, I encountered some hiccups that led to errors after I tried to run it. For example, I capitalized the “I” in the “if” statement, then forgot to define `n1` and `n2`, added “float” in the input statement, and added the “else” block for the invalid operation section. After I resolved those initial issues, I ran the code and encountered a few more, but eventually

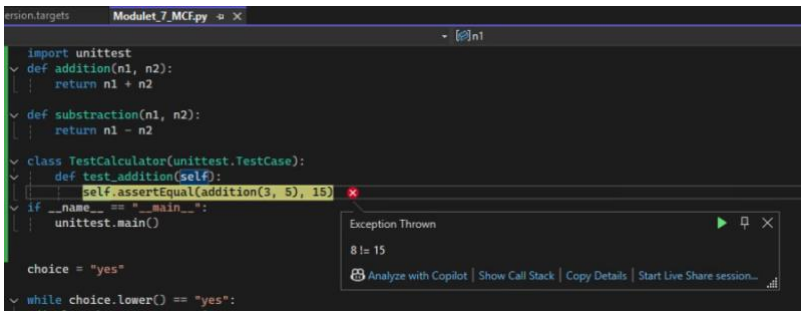
got a workable code. To my delight, success is shown here in this snippet example. I ran through all the operations to see if they all worked, and all showed positive results. Script done, check!



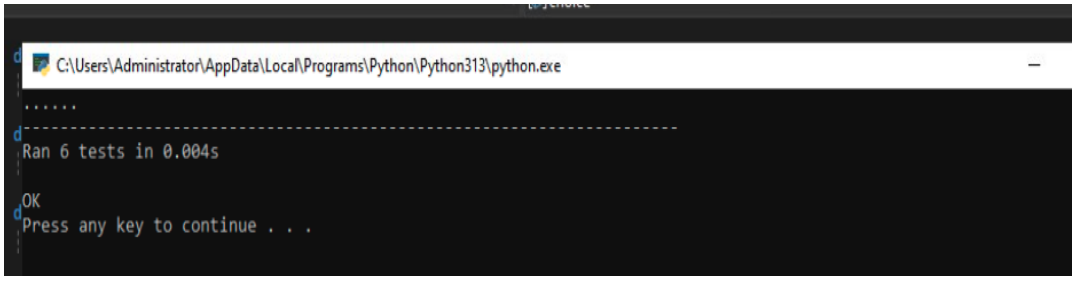

Once I felt confident in the scripted section, I researched the material to better understand the rationale for unit testing and the basic steps for conducting unit tests. I learned that unit tests are small, automated tests that verify that functions behave as expected, help catch bugs early, and ensure code reliability.

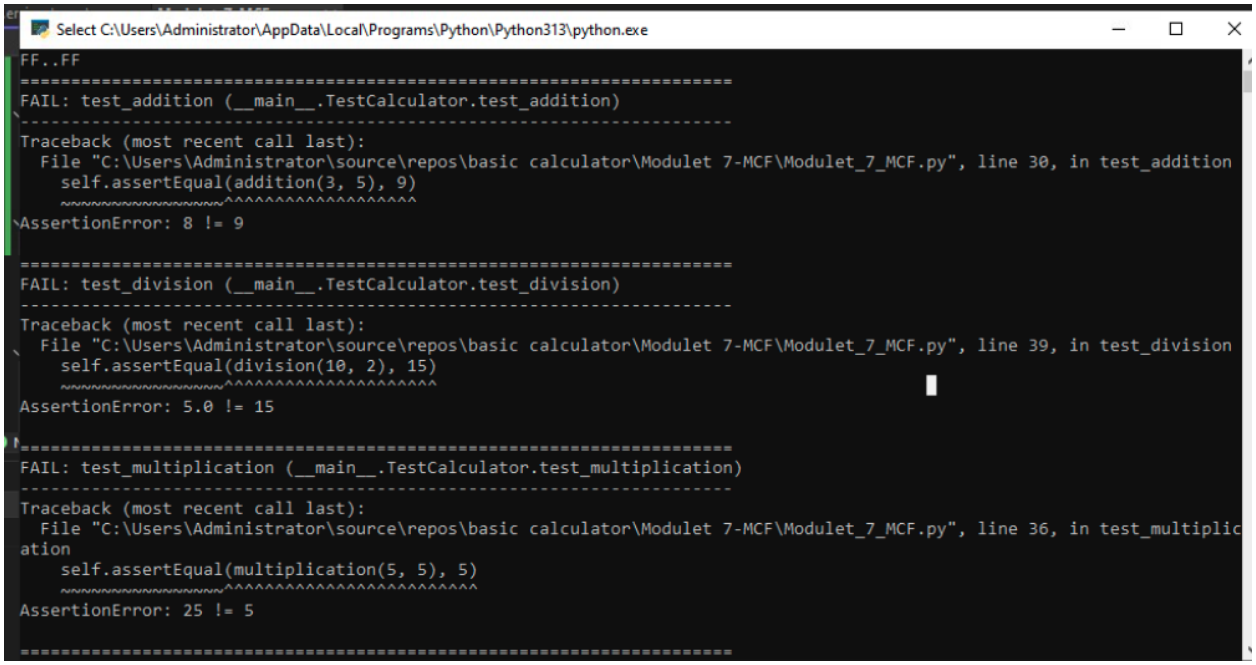
During my research on unit testing, I discovered that refactoring my code would improve its efficiency by increasing readability, reducing duplication, and making debugging during unit tests easier. I put this theory to the test and refactored sections of my code. To make sure I did it correctly, I consulted a tutor to see if I was on track. Initially, each operation was written directly in the main logic, such as `if operation == “+”: result = n1 + n2.` By reorganizing the code to create separate functions, such as `addition(n1, n2)`, the logic is cleaner and easier to test. This refactoring process helped me optimize the code. This logic would allow each arithmetic function to be tested individually.

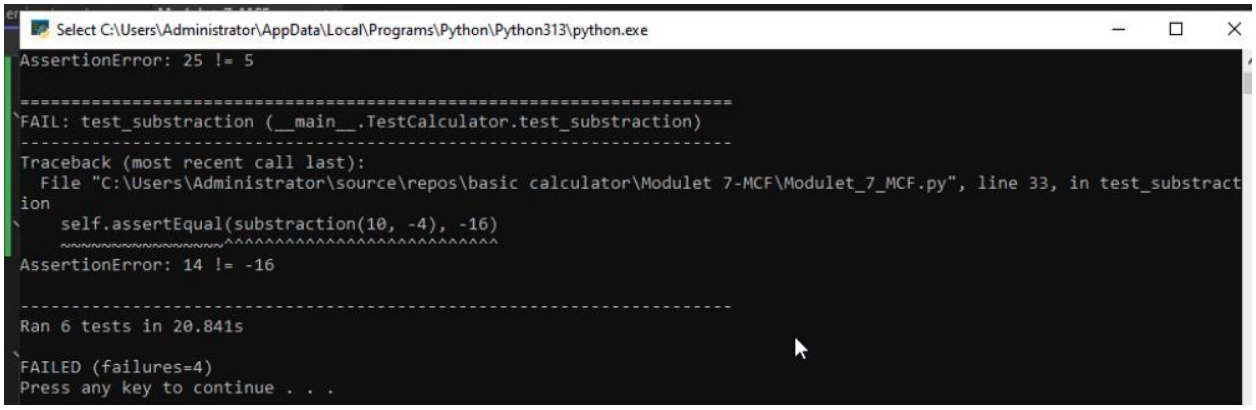
Let’s go deeper into unit testing and what I encountered, and how I resolved the errors. When I initially refactored the addition block to make it testable, I had to create a test class and a test method to verify that my `addition()` function worked correctly using the assertEqual check: `assertEqual(addition(3, 5), 8)`. After running the initial unit test, which passed, the next



step was to change the expected output from “8” to “15” to see if the test would fail, which it did. (See image). This confirms that the test was valid and sensitive to both correct and incorrect outputs. I also found out that the most useful way is to run it from the command line. The benefit of running the test using the command-line interface is improved productivity, as it allows running multiple files at once, which is what I did to see how the other operators handled correct and incorrect outputs. From this point on, I ran several tests with different scenarios to see what I could break and got almost positive results, but the division block gave me an issue. The test for division failed due to a subtle punctuation mismatch. I forgot to add the period at the end of the Error message. Even though I got this fixed, I still encounter other issues. I would fix one, then another would appear after it was resolved. It was constant back and forth. The errors I encountered were syntax errors (e.g., I forgot to add a colon after the “if” and “else” statements), indentation errors that I resolved by counting spaces across all lines, NameErrors, which I misspelled addition, multiplication, assertEqual, and return a few times, and resolved them by checking and correcting each line. Finally, after resolving the errors, I confirmed that all tests, both passing and failing, behaved as expected. I made another call for help to see if I was on track. I finally successfully completed the unit tests. Images of the unit test results are displayed.






In conclusion, this assignment gave me a deeper understanding of how programming works behind the scenes. I learned that planning tools like flowcharts and pseudocode guide the optimization of program structure, and that refactoring improves code clarity, which in turn supports unit testing. It tested my ability to identify and resolve issues. This experience gave me practical strategies I can apply to future projects.

## References

Helmers, S. (2020, November 13). *Creating flowcharts for beginners*. LinkedIn Learning. https://www.linkedin.com/learning/creating-flowcharts-for-beginners/flowcharts-swimlanes-and-more?u=2106393

Melwalli, S. (2024, September 3).Pseudocode: What it is and how to write it. *BuiltIn*. https://builtin.com/data-science/pseudocode

Silveira, O. S. (2025, April 8). *Unit tests in Python: A beginner's guide*. Dataquest. https://www.dataquest.io/blog/unit-tests-python/

Zybooks. (n.d.). *Module 7 Participation Activity*. https://learn.snhu.edu/d2l/le/content/2108542/viewContent/45814707/View