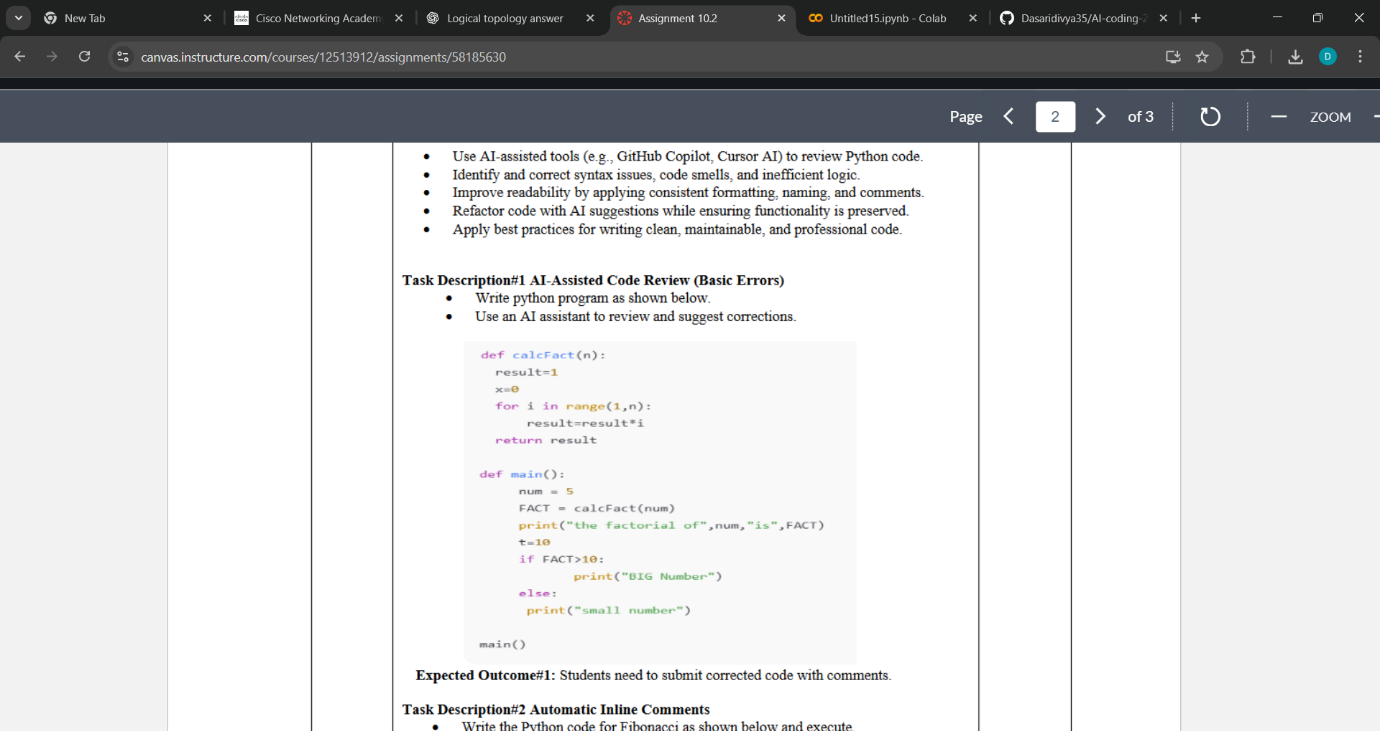
Assignment-10.2

Name:D Divya

Batch:14

Rollno:2403a51356

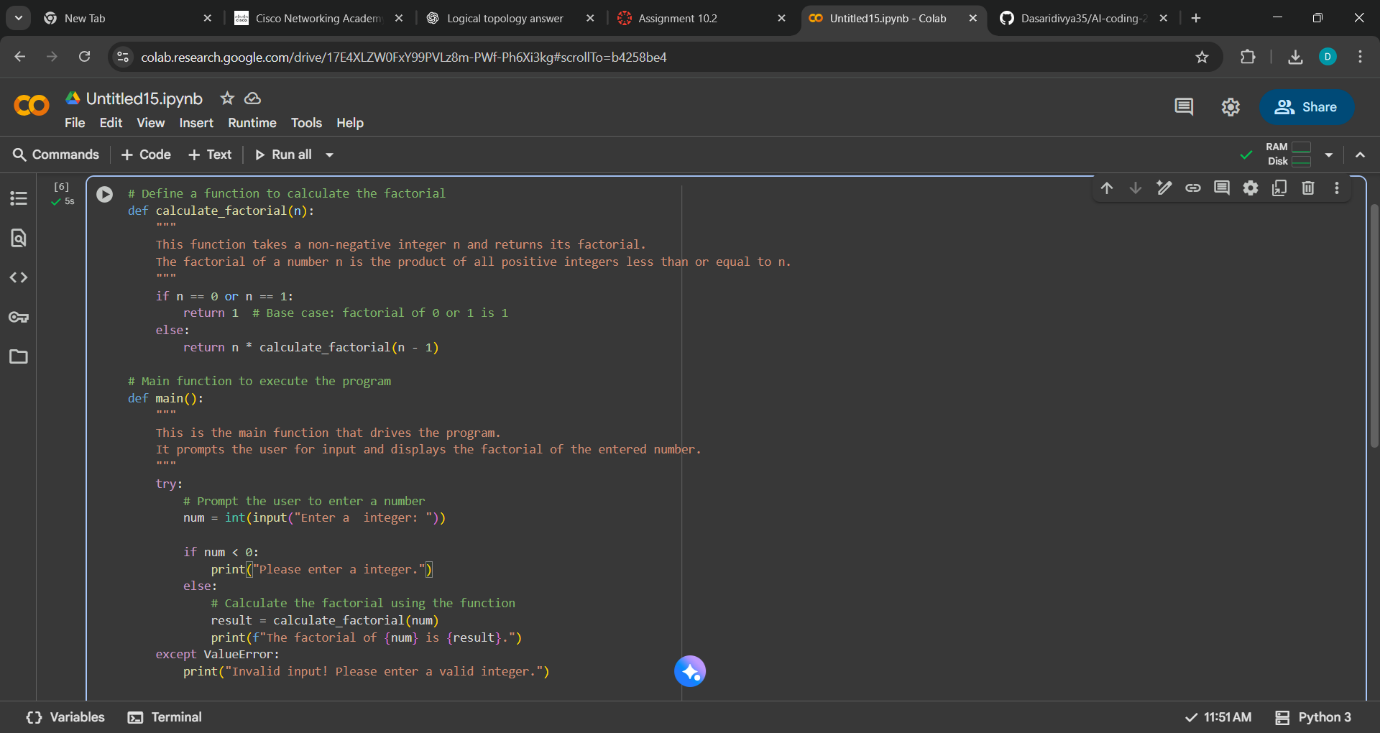
Task Description#1 AI-Assisted Code Review (Basic Errors)  
• Write python program as shown below.  
• Use an AI assistant to review and suggest corrections

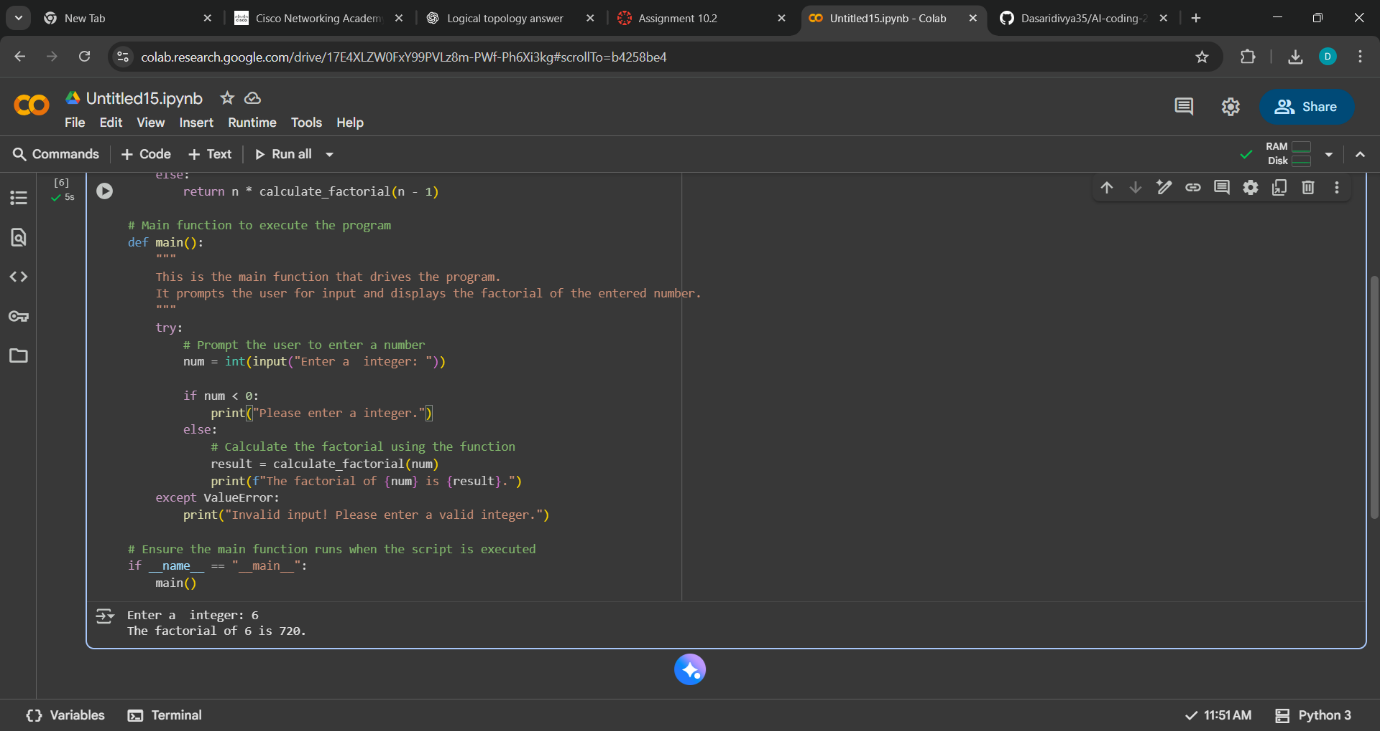


Expected Outcome#1: Students need to submit corrected code with comments

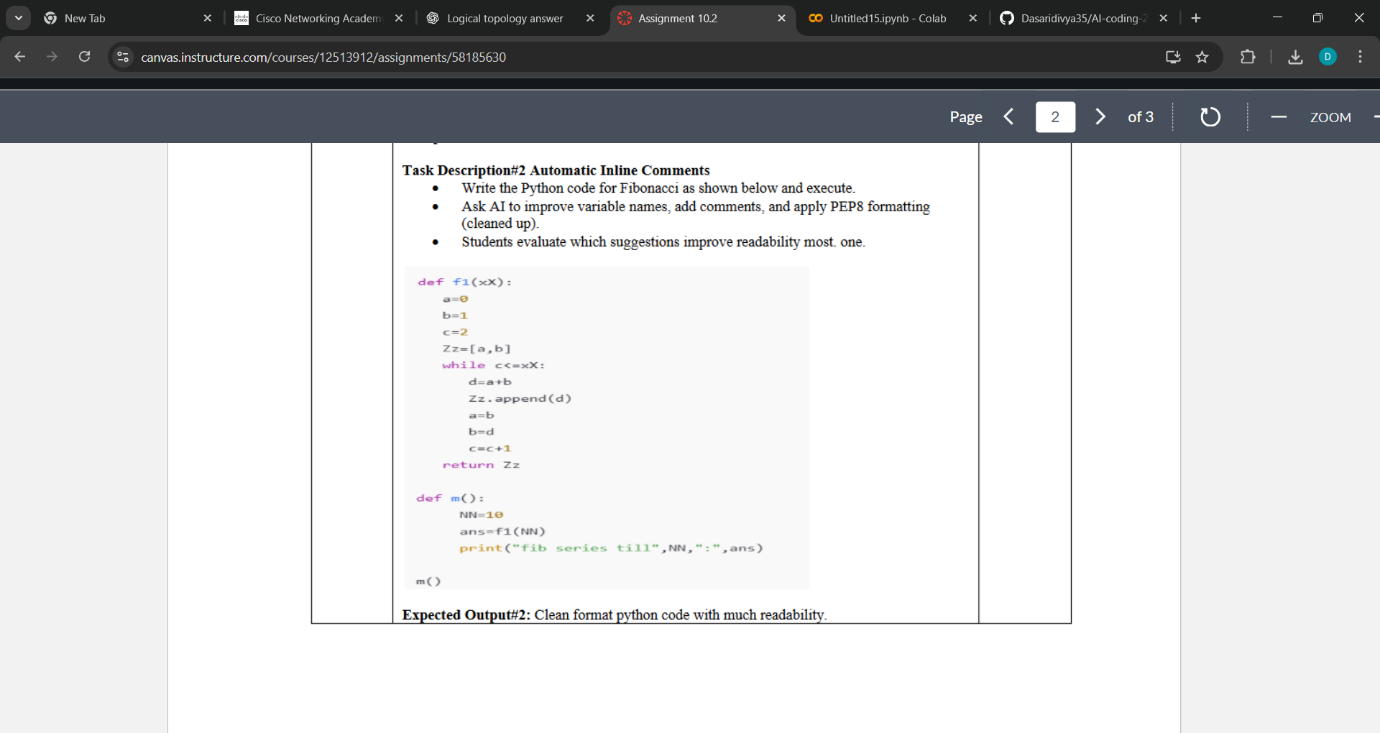
Prompt: Python program that calculates the factorial of a number using functions, with detailed comments.

Code with output:



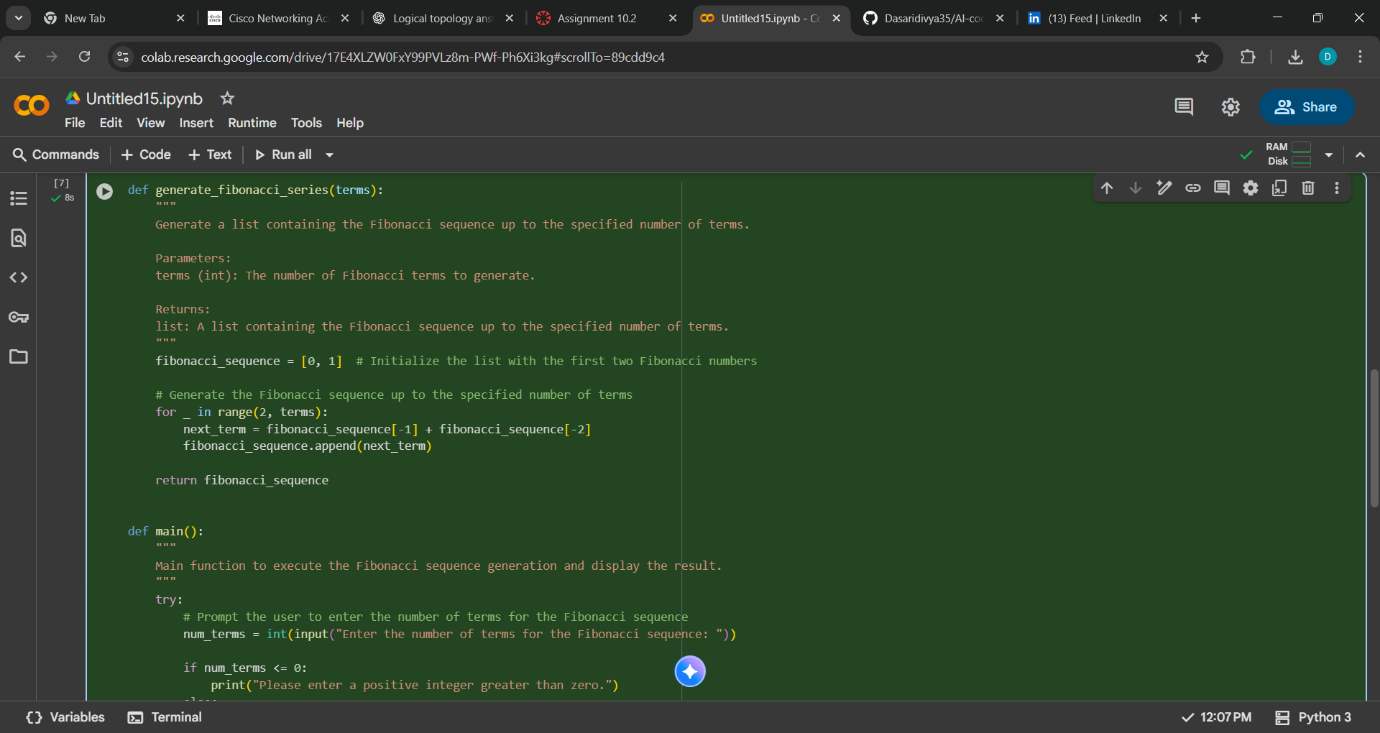


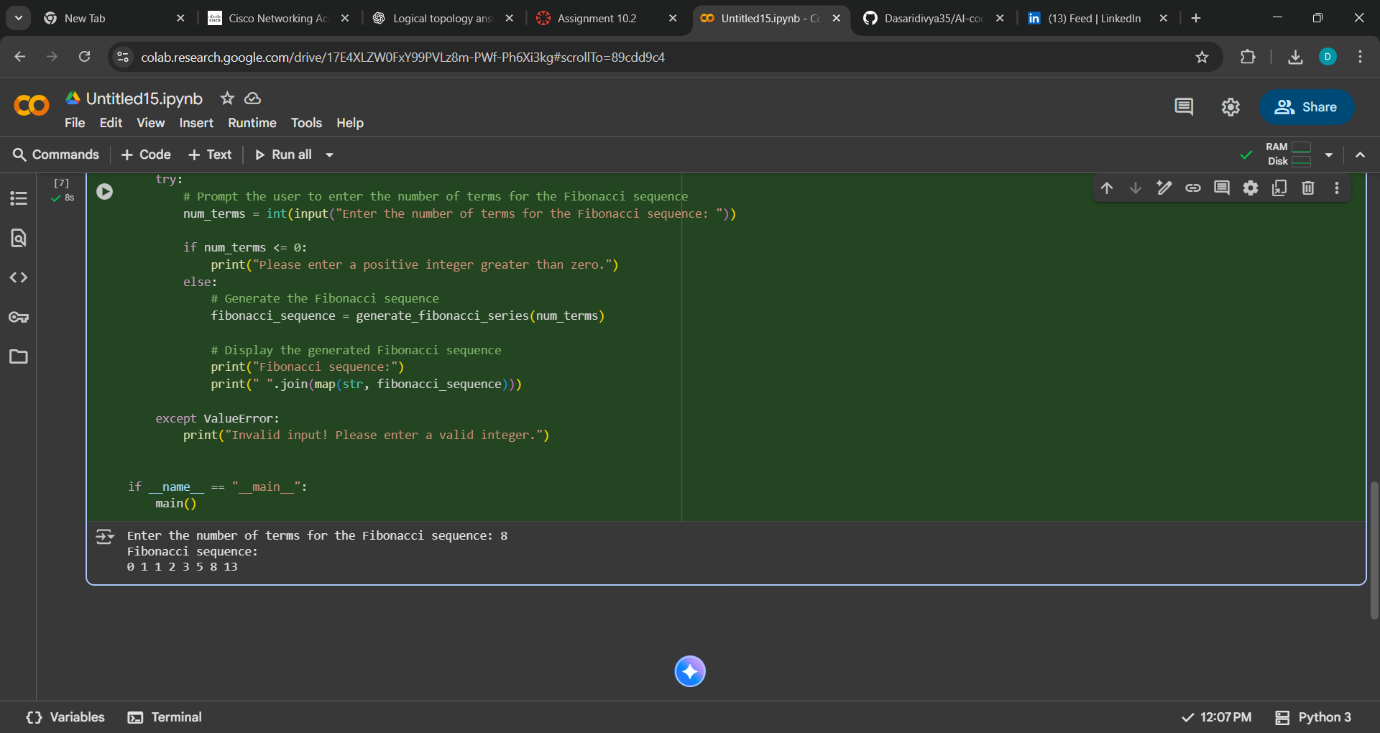
Task Description#2 Automatic Inline Comments  
• Write the Python code for Fibonacci as shown below and execute.  
• Ask AI to improve variable names, add comments, and apply PEP8 formatting  
(cleaned up).  
• Students evaluate which suggestions improve readability most. one.

  
Expected Output#2: Clean format python code with much readability.

Prompt:Python program that calculates the Fibonacci sequence up to a specified number of terms, with improved variable names, added comments, and formatted according to PEP8 standards for better readability.

Code with output:





Observation:

The use of docstrings and inline comments gives context and explanation, making it easy for another developer (or yourself later) to grasp the logic.

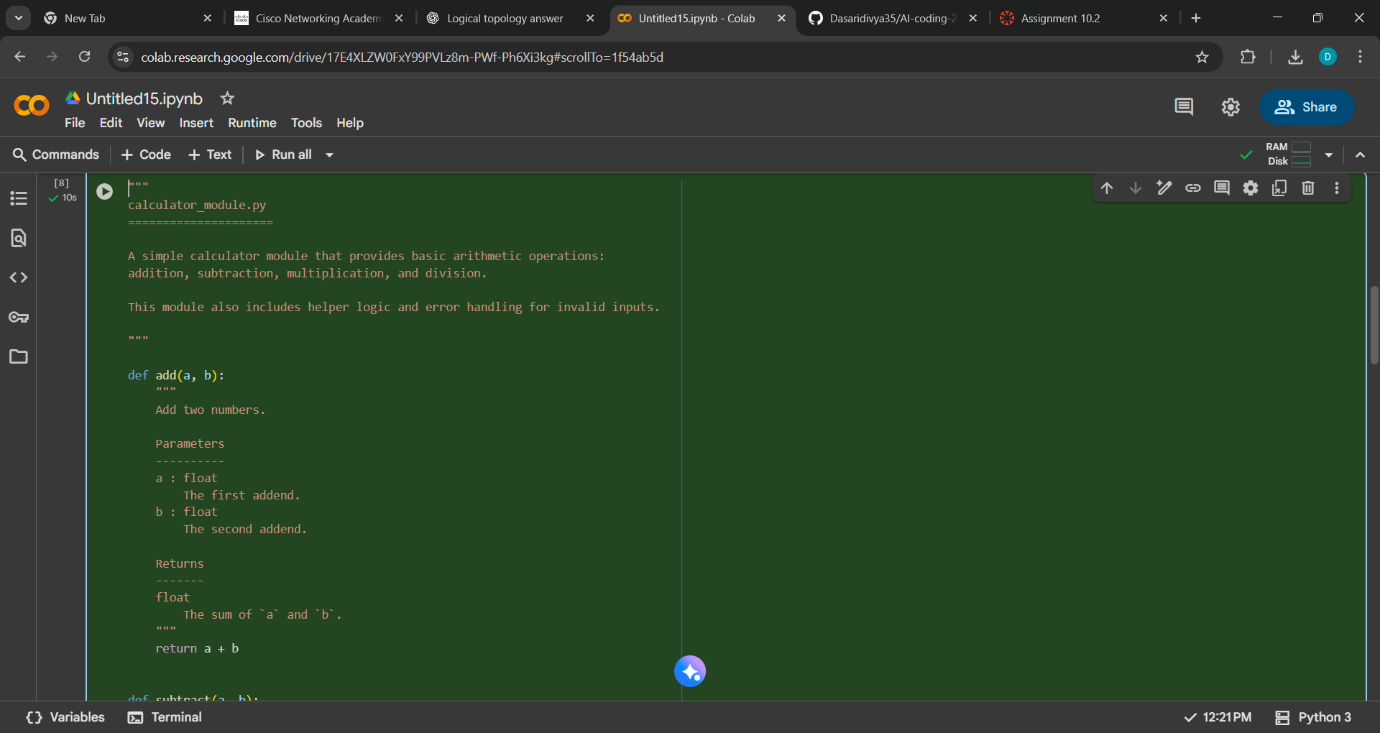
The code is modular: one function for generating the sequence, another (main) for user interaction and output.

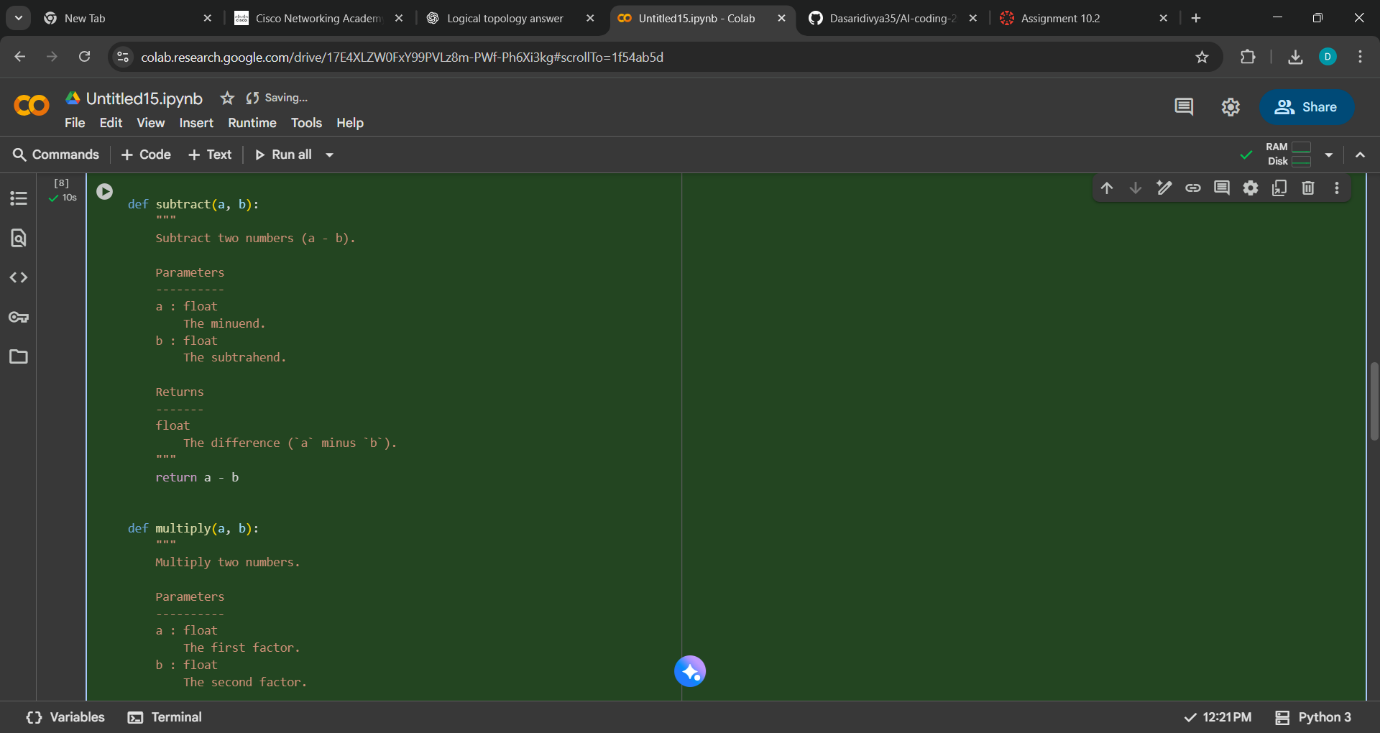
This separation helps with testing and maintenance (you can test generate\_fibonacci\_series independently).

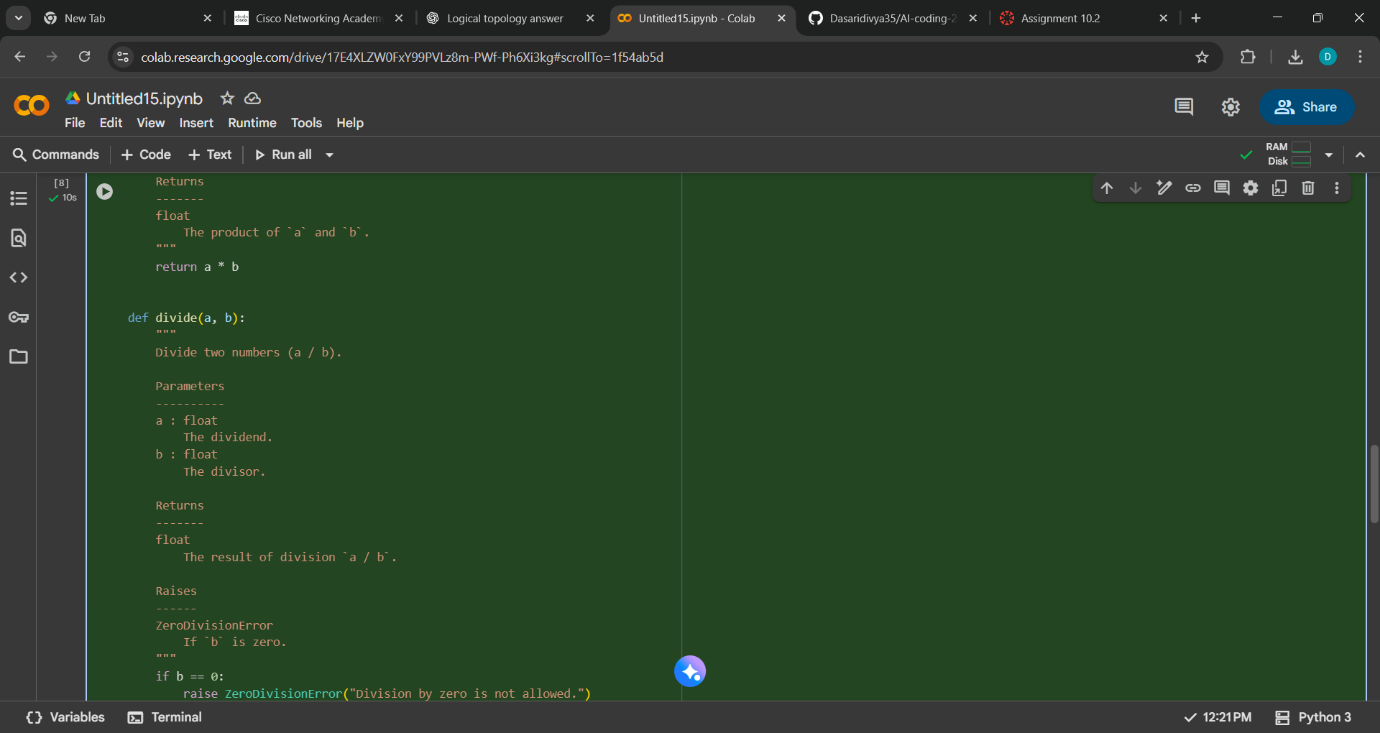
Task Description#3  
• Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply,  
divide).  
• Incorporate manual docstring in code with NumPy Style  
• Use AI assistance to generate a module-level docstring + individual function  
docstrings.  
• Compare the AI-generated docstring with your manually written one.  
Common Examples of Code Smells  
• Long Function – A single function tries to do too many things.  
• Duplicate Code – Copy-pasted logic in multiple places.  
• Poor Naming – Variables or functions with confusing names (x1, foo, data123).  
• Unused Variables – Declaring variables but never using them.  
• Magic Numbers – Using unexplained constants (3.14159 instead of PI).  
• Deep Nesting – Too many if/else levels, making code hard to read.  
• Large Class – A single class handling too many responsibilities.  
Why Detecting Code Smells is Important  
• Makes code easier to read and maintain.  
• Reduces chance of bugs in future updates.  
• Helps in refactoring (improving structure without changing behavior).  
• Encourages clean coding practices  
Dead Code – Code that is never executed.  
Expected Output#3: Students learn structured documentation for multi-function scripts  
Push documentation whole workspace as .md file in GitHub Repository  
Note: Report should be submitted a word document for all tasks in a single document with  
prompts, comments & code explanation, and output and if required, screenshots.

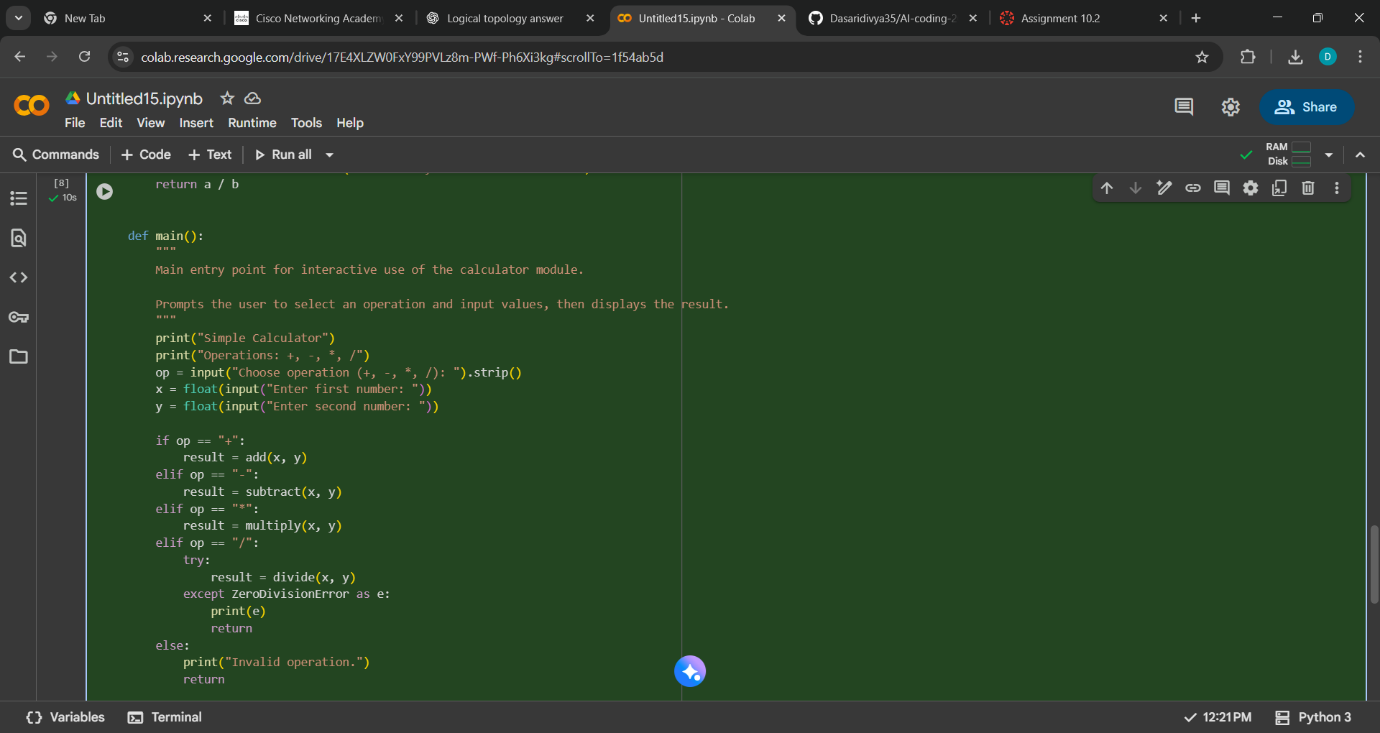
Prompt: Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply,divide).Generate NumPy‑style docstrings for this module and its functions (add, subtract, multiply, divide, main), including summary, Parameters, Returns, and Raises sections.

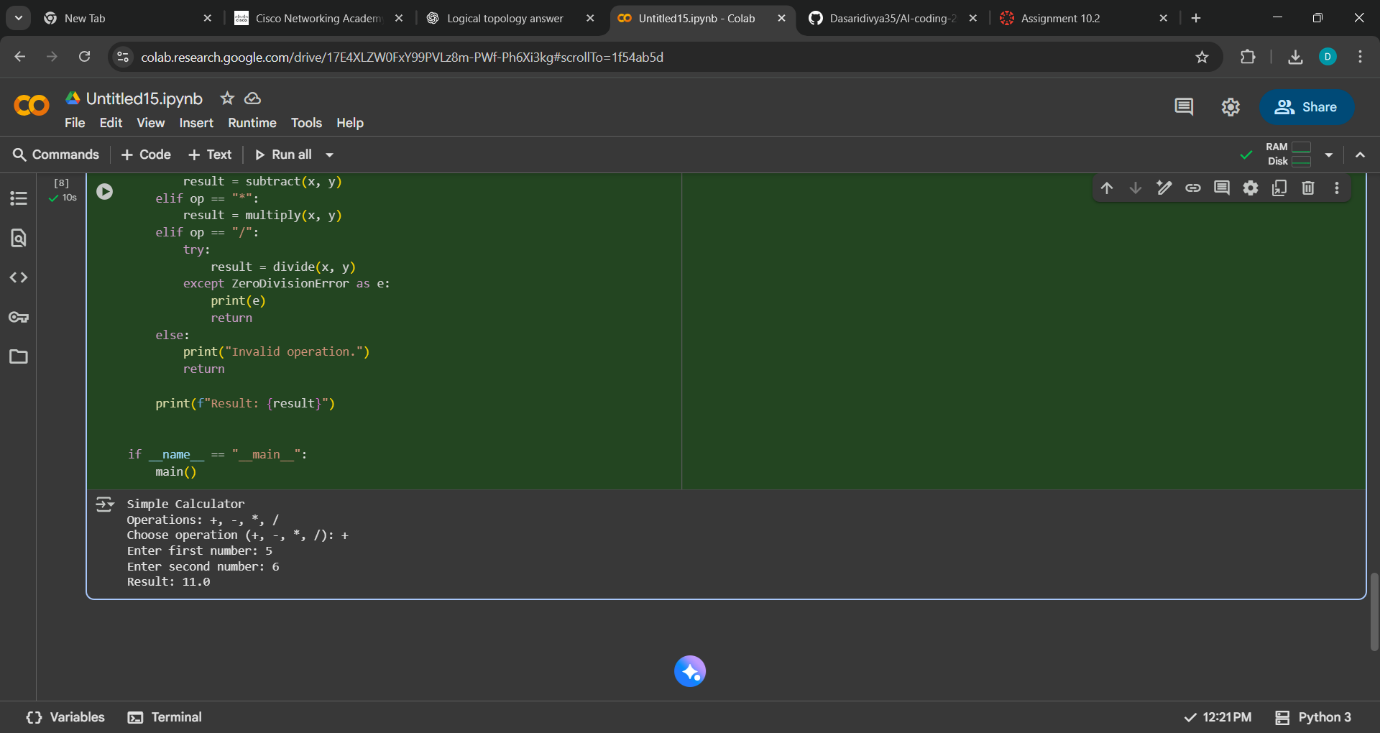
Code with output:











| **Aspect** | **Manual Docstrings** | **AI‑Generated Docstrings** | **Notes / What seems better** |
| --- | --- | --- | --- |
| **Descriptive summary** | You may write a more contextual, explanatory summary | AI tends to give succinct descriptions | If your manual summary mentions usage or caveats, it may be more helpful |
| **Parameter naming** | You can choose clear, consistent names | AI uses same parameter names but sometimes less descriptive terms (“operand” vs “addend”) | Clear parameter names are crucial for readability |
| **Raises / error conditions** | You can highlight specific domain issues or constraints | AI includes standard exception sections if it detects them | The “Raises” sections align well and are helpful for understanding failure modes |
| **Module‑level docstring** | You might include more context, examples, limitations | AI version is more skeletal but consistent | A richer manual module docstring often gives better orientation |
| **Consistency & formatting** | You may unintentionally deviate from a consistent style | AI docstrings tend to follow one style pattern uniformly | Consistency is attractive and promotes readability |
| **Length / verbosity** | Too long docstrings can be verbose and distract | AI versions are often terser | The “just right” length—concise yet informative—is ideal |

Observation:

The AI‑generated docstrings provide a strong baseline: they are concise, consistently formatted, and cover standard sections (Parameters, Returns, Raises). But your manual ones can add more context, caveats, examples, or domain-specific nuances. The best readability often comes from combining the AI’s consistent structure with your domain knowledge and context in manual extensions.