**LAB 10.2**

**Hall Ticket:** 2403A51313

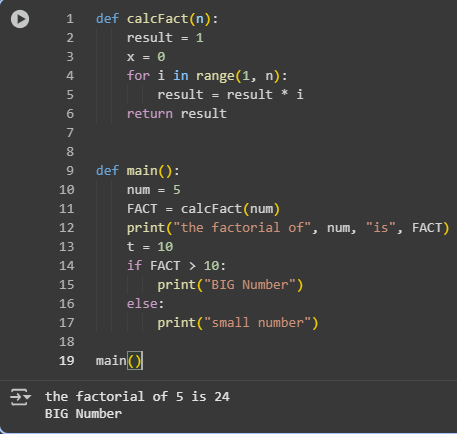
**Batch:** 13

**Task Description#1** AI-Assisted Code Review (Basic Errors)

**Prompt:**

Perform an AI-assisted code review for the given Python program. Identify and explain all logical, syntactical, and stylistic errors in paragraph form. Provide the corrected version of the code with clear inline comments describing the modifications. Highlighting the significance of error detection, corrections made, and the importance of following coding standards.

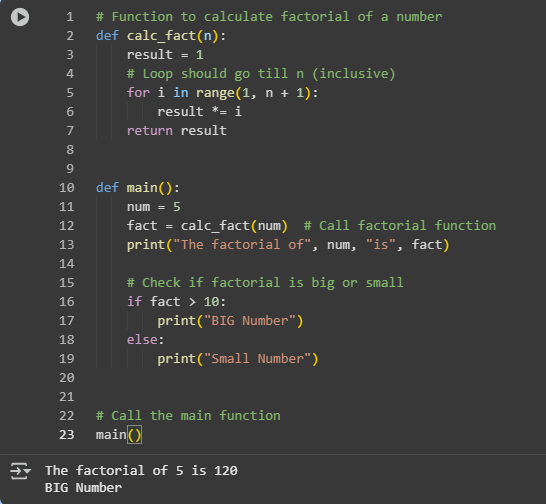
**Error Code:**

****

**Issues Found:**

The factorial calculation logic is incorrect because the loop uses range(1, n) which only goes up to n-1, so the last multiplication is skipped. This needs to be changed to range(1, n+1) to include the number n. Additionally, the variable x = 0 is declared but never used in the program, so it is unnecessary and should be removed. Similarly, the variable t = 10 is defined but never used anywhere, making it redundant. Another issue is with naming conventions; the variable FACT should ideally be written in lowercase as fact, and the function name calcFact should be updated to calc\_fact to follow Python’s PEP8 style guidelines for readability.

**Corrected Code:**

****

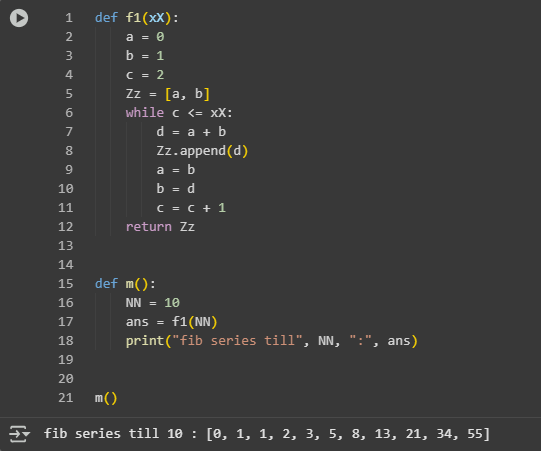
**Observation:**

The given task demonstrates the importance of AI-assisted code review in identifying and correcting basic programming errors. The original factorial program contained several issues, such as an incorrect loop range that excluded the last number, unused variables (x and t), and inconsistent naming conventions that reduced readability. Through review, these issues were identified and corrected, resulting in a cleaner and more accurate program. The corrected code now calculates the factorial properly, follows Python standards, and produces the expected output. This exercise highlights how AI support can guide students in debugging, improving code quality, and adhering to coding best practices.

**Task Description#2** Automatic Inline Comments

**Prompt:**  
Clean up the given Fibonacci Python code by applying PEP8 formatting, using meaningful variable names, and adding comments or docstrings. Point out the issues in the original code and provide the improved version.

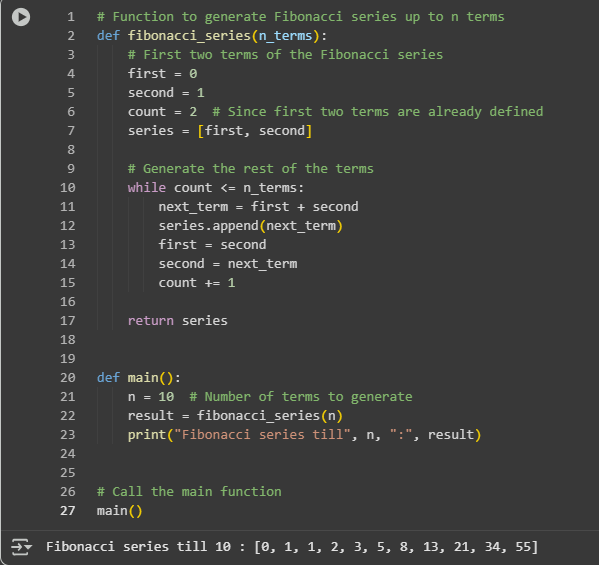
**Code:**

****

**Issues Found:**

The given Fibonacci program works but has several readability issues. The function and variable names such as f1, xX, Zz, and NN are not meaningful, making it hard to understand the purpose of each element. The code lacks inline comments, which reduces clarity for beginners or anyone reviewing the logic. Additionally, the formatting does not fully follow Python’s PEP8 guidelines, as variable names are not in snake\_case and spacing is inconsistent. The use of unexplained values such as c = 2 also makes the logic harder to follow without prior knowledge. Overall, while the logic is correct, the program needs improvements in naming, commenting, and formatting to achieve better readability and maintainability.

**Cleaned Code:**

****

**Observation:**

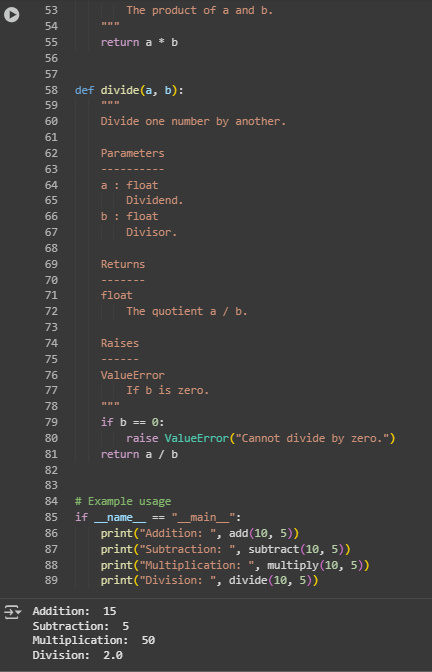
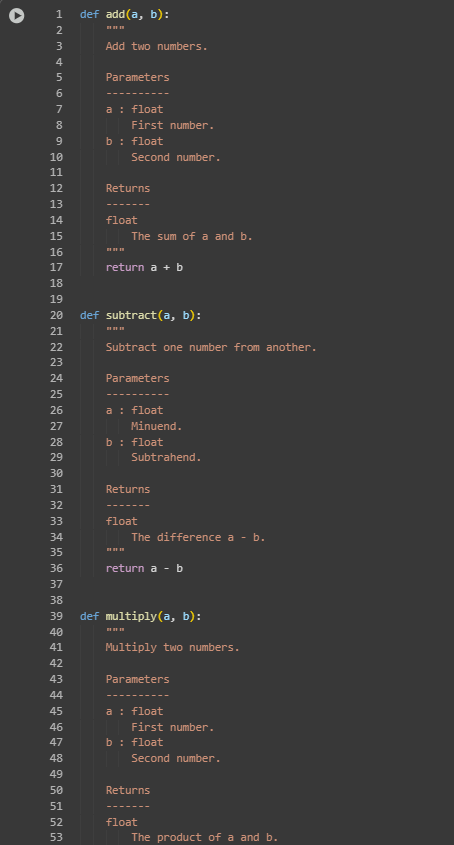
The task of reviewing and improving the Fibonacci program highlights how important code readability and proper formatting are in programming. The original program generated correct results, but its poor variable naming, lack of comments, and inconsistent formatting made it difficult to understand and maintain. After applying meaningful variable names, adding inline comments, and following PEP8 coding standards, the program became much clearer and more professional. This exercise shows that even when logic is correct, clean coding practices are essential for writing programs that are easy to read, debug, and share with others.

**Task Description#3**

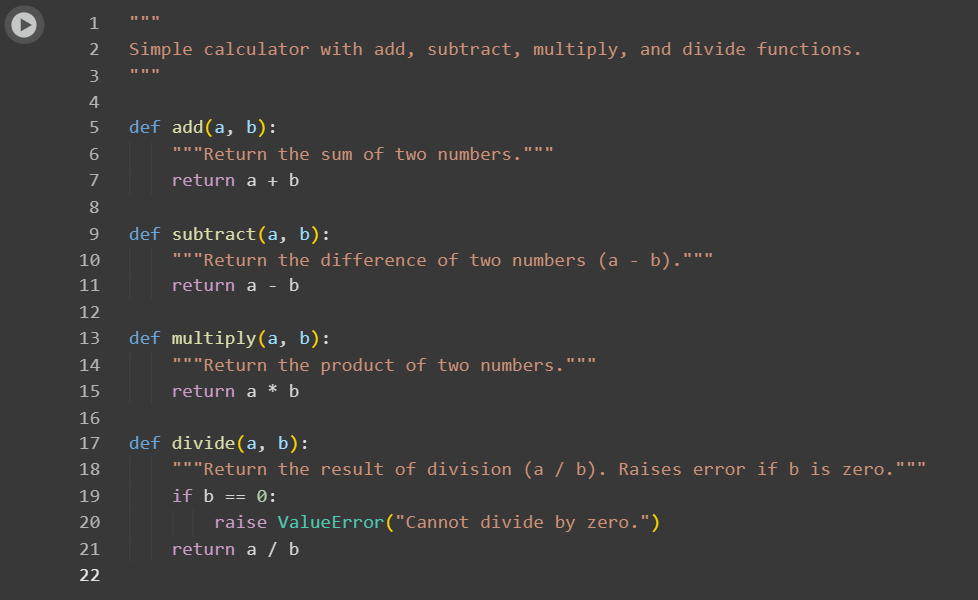
**Prompt:**

Write a Python program that contains a few calculator functions such as add, subtract, multiply, and divide. For each function, include NumPy-style manual docstrings that explain the parameters, return values, errors, and examples, and also generate AI-style docstrings that are shorter and more natural. Compare the manual docstrings with the AI-generated ones, pointing out the differences in detail and clarity. In addition, explain common examples of code smells such as long functions, duplicate code, poor naming, unused variables, magic numbers, deep nesting, and large classes, and provide simple Python examples to illustrate each of them.

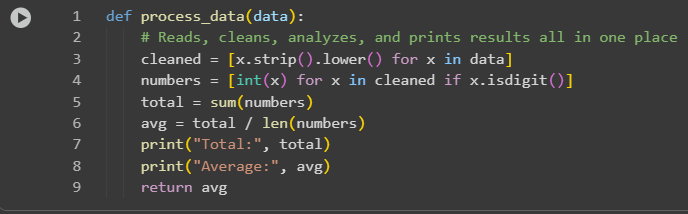
**Python Script:**

****

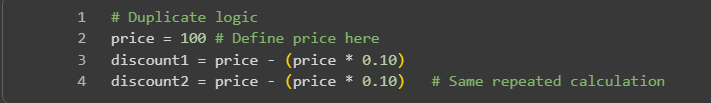
**AI-Generated Version:**

****

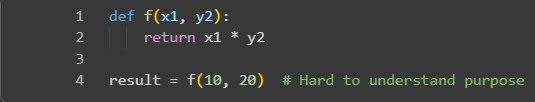
**Long Function:**

****

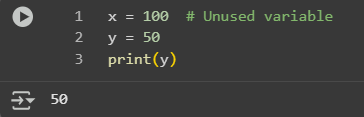
**Duplicate Code:**

****

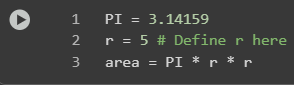
**Poor Naming:**

****

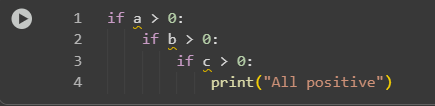
**Unused Variables:**

****

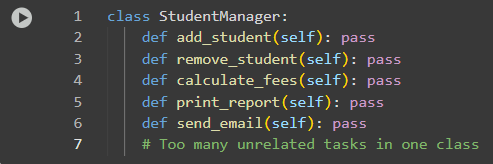
**Magic Numbers:**

****

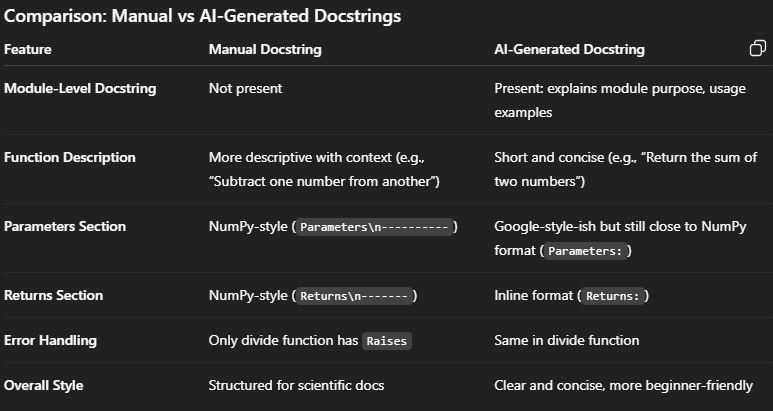
**Deep Nesting:**

****

**Large Class:**

****

**Comparision:**

****

**Observation:**

The AI-generated docstrings provide a clear and concise description of each function and include a module-level docstring that explains the overall purpose and usage of the calculator. Compared to the manually written docstrings, the AI version is shorter, more general, and beginner-friendly, making it easier to quickly understand the functionality. On the other hand, the manual docstrings follow the formal NumPy style, with well-structured sections for parameters, returns, and exceptions, which is more suitable for scientific or professional documentation. Overall, the AI-generated docstrings improve readability and provide context at the module level, while the manual docstrings offer detailed, structured information for precise technical use.