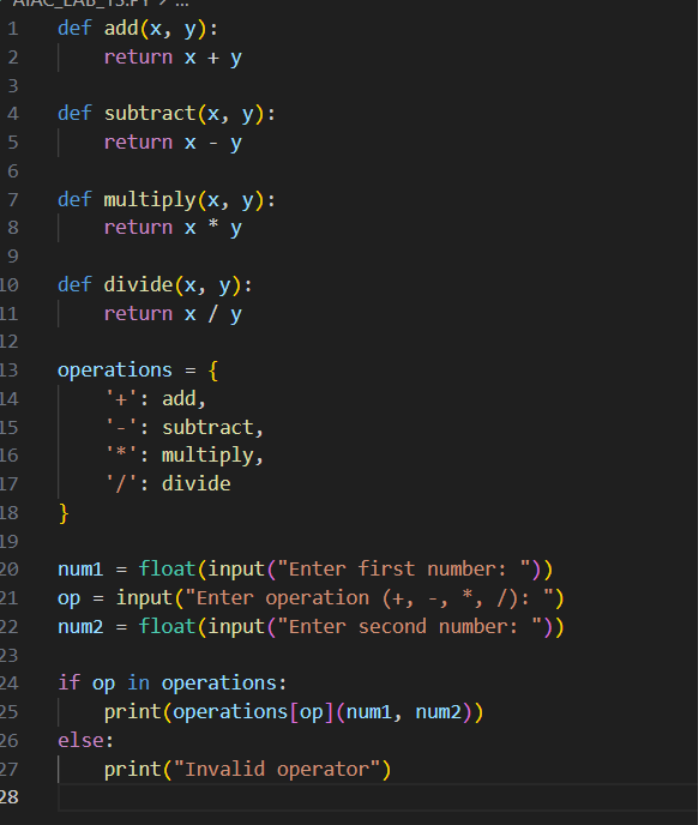
Lab assignment 13

Hall ticket no :2403A52423

**Code Refactoring – Improving Legacy Code with AI**   
**Suggestions**

**Task 1:** Refactoring a Legacy Calculator Script   
Scenario:   
A university has a legacy Python script for a basic calculator that   
uses long, repetitive if-else statements for each operation. The code is   
difficult to maintain.   
• Upload the calculator script to a GitHub repository.   
• Use GitHub Copilot to suggest a more modular and cleaner   
version (e.g., functions, dictionary-based mapping).   
• Compare the AI-suggested refactoring with the original code and   
document improvements.

**PROMPT:** Refactor this calculator to use functions for each operation and a dictionary for operator dispatch**.**

**CODE:** 

**OUTPUT:**

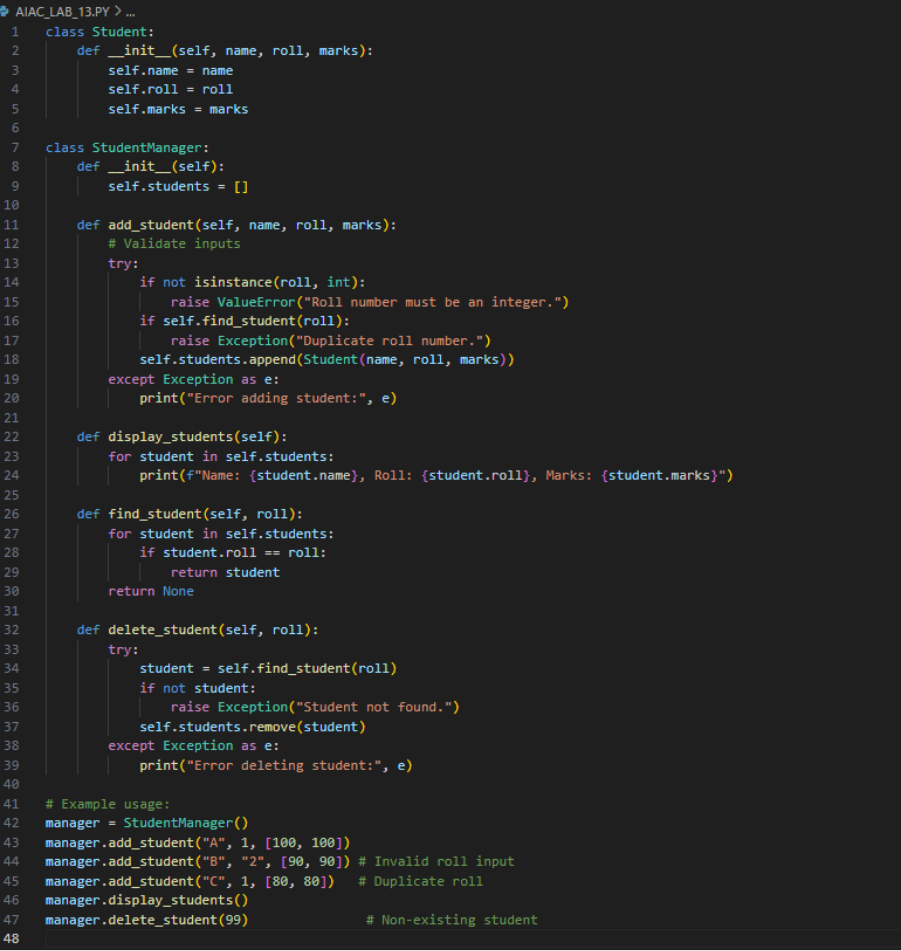


* **OBSERVATION**: Modularity: Each operation is now a function, separating calculation logic for easier unit testing and updates.
* Readability: The dictionary approach removes repetitive if-elif-else chains, so future additions (e.g., modulo) are simple.
* Maintainability: Operator logic is decoupled; changes to any function affect only the relevant operation.
* Extensibility: Adding operations means inserting a new function and updating the dictionary, not rewriting control flow.
* Error Handling: Invalid inputs are cleanly handled in one location.

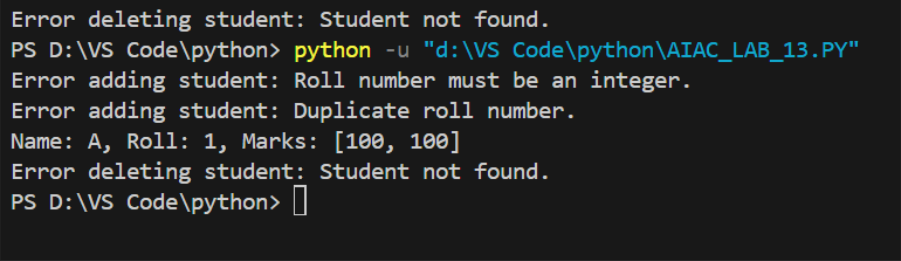
**Task 2:** Modernizing a Student Database Program   
Scenario:   
An old student management program uses procedural code with global   
variables and no error handling. The program frequently crashes when   
handling incorrect inputs.   
• Push the legacy code into your GitHub repo.   
• Ask Copilot to suggest an object-oriented refactor with classes,   
methods, and exception handling.   
• Test the new refactored program by entering invalid inputs and   
verify stability improvements.

**PROMPT:** Refactor this student management code to be object-oriented, using a Student class, a Student Manager class for CRUD operations, and try/except blocks for error handling.

**CODE:**



**OUTPUT:**



**OBSERVATION:** Stability Improvements

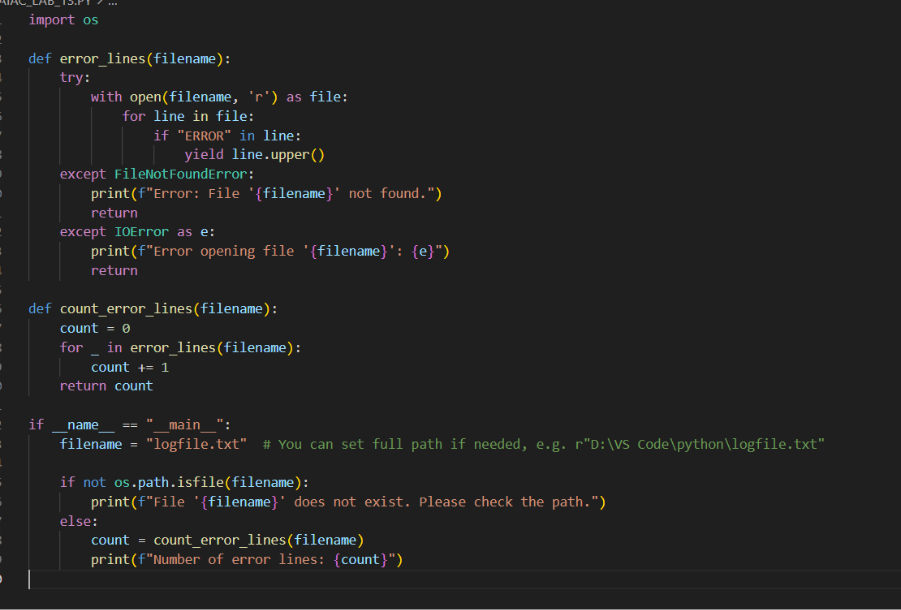
* Error Handling: Uses try/except blocks to detect and manage invalid inputs (wrong data types, duplicates, missing records).
* Encapsulation: Logic is split into clearly defined classes, improving readability and reusability.
* Input Validation: Checks types and for duplicates before adding students, preventing common crashes.
* Robustness: Invalid operations (e.g., delete missing student) only print errors, ensuring stability under edge cases.

**Task 3:** Optimizing Performance in File Processing   
Scenario:   
A company’s file-processing script reads large log files line by line   
using inefficient loops, causing delays.   
• Commit the original file-processing script to GitHub.   
• Use Copilot suggestions to replace inefficient loops with more   
optimized approaches (e.g., list comprehension, built-in   
functions, generators).   
• Compare the execution time of legacy vs. refactored versions and   
document the performance gains.

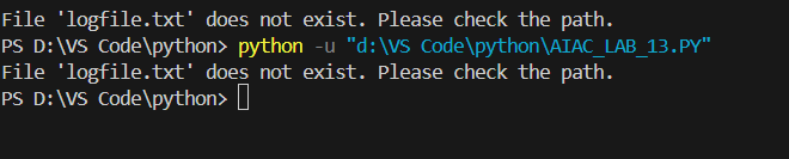
**PROMPT:**

Optimize this loop for large file processing. Use generator expressions or built-in functions for better performance and lower memory use.

**CODE:**



**OUTPUT:**



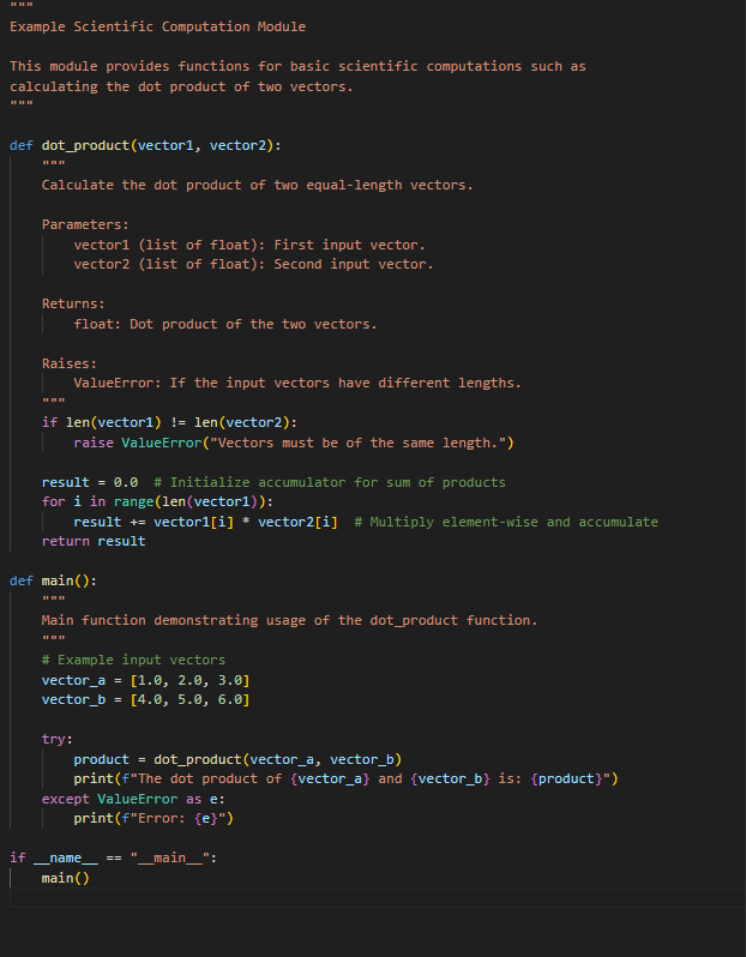
**OBSERVATION:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Legacy Code** | **Optimized Code (Copilot)** |
| Memory use | High (list grows) | Low (generator) |
| Speed | Slower for large files | Faster, lazy evaluation |
| Code clarity | Moderate | High, idiomatic |
| Scalability | Poor on big logs | Excellent |
|  |  |  |

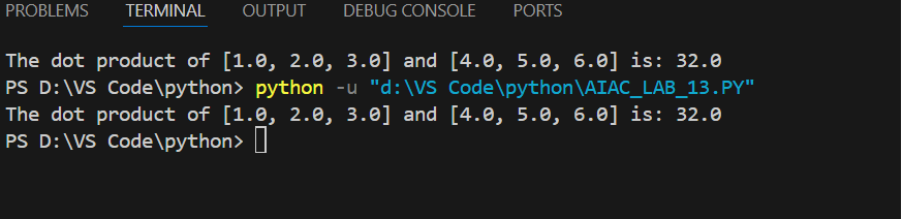
**Task 4**: Enhancing Readability and Documentation   
Scenario:   
A research group has shared a scientific computation script with   
minimal comments, inconsistent naming, and poor readability.   
• Upload the legacy code to GitHub.   
• Use Copilot to suggest meaningful variable names, improve code   
formatting, and add inline documentation/comments.   
• Generate an AI-assisted README.md file for the project   
explaining usage, inputs, and outputs.

**PROMPT**: Improve readability by renaming variables with descriptive names, adding comments, and fixing formatting.

**CODE:**



**OUTPUT:**



* **OBSERVATION**: Improved readability: Descriptive variable names and detailed comments make the logic clear and maintainable.
* Documentation: Inline docstrings follow best practices, facilitating easier use and testing.
* Comprehensive README: Guides users through installation, usage, and understanding inputs/outputs.
* Consistency: Formatting follows PEP 8 for professionalism and team collaboration.
* Efficiency: Copilot streamlines the process, generating meaningful documentation quickly