

# Assignment 13

## Lab 13 – Code Refactoring: Improving Legacy Code with AI

Htno : 2503A52L16

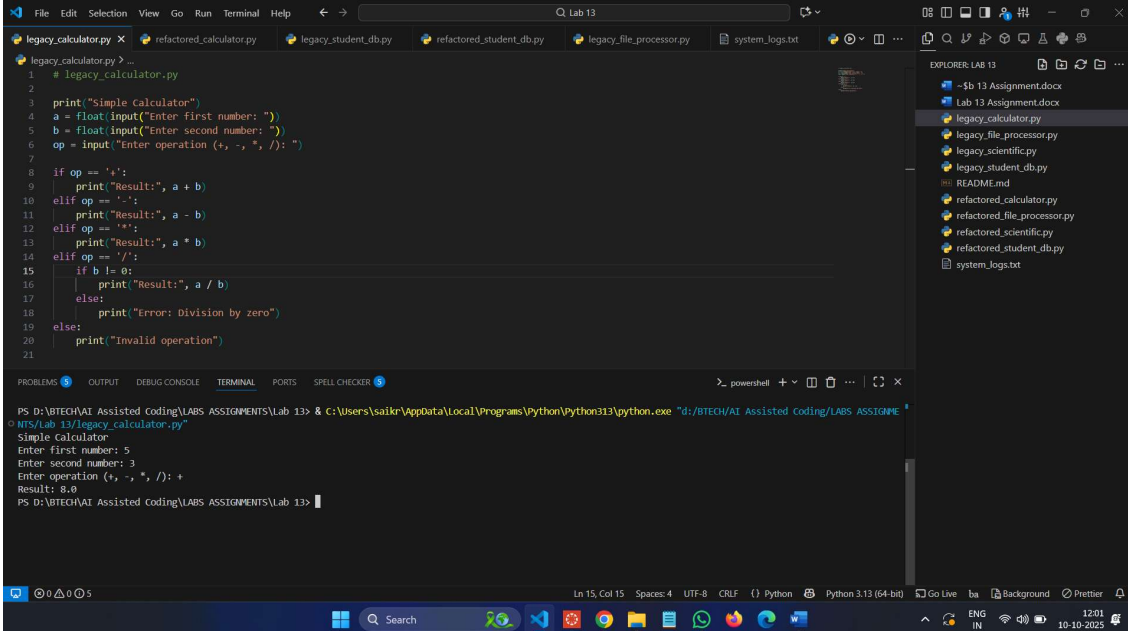
### Task 1: Refactoring a Legacy Calculator Script

#### 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

#### Legacy Code & Output :



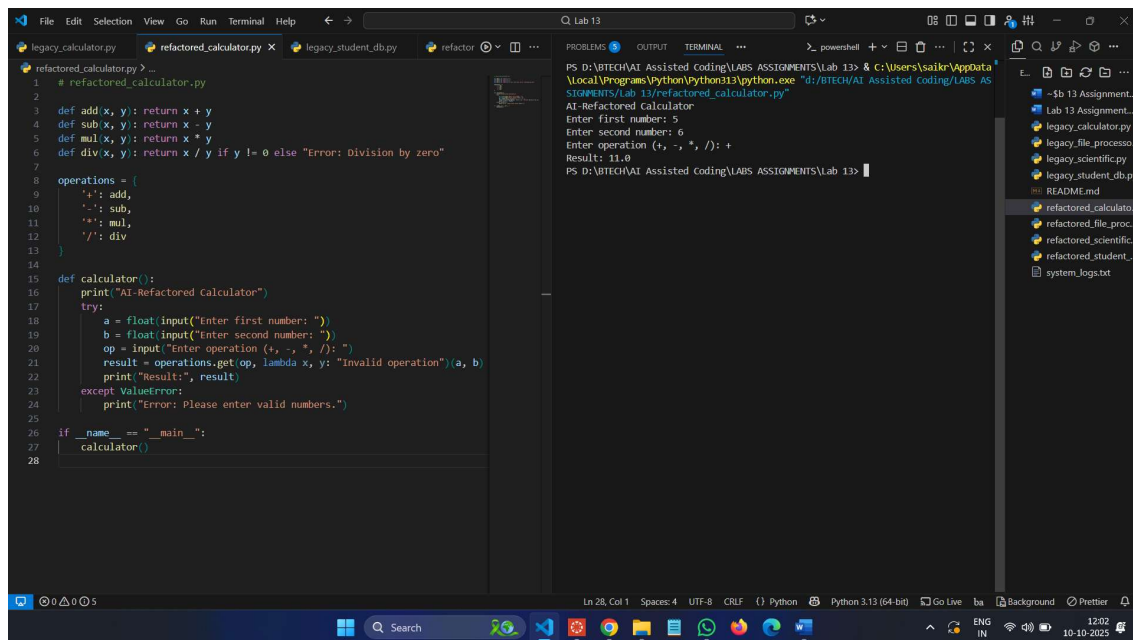
The screenshot displays a Visual Studio Code editor window with a file explorer on the right and a terminal at the bottom. The file explorer shows a project named 'LAB 13' containing several files: 'lab\_13\_assignment.docx', 'legacy\_calculator.py', 'legacy\_file\_processor.py', 'legacy\_scientific.py', 'legacy\_student\_db.py', 'README.md', 'refactored\_calculator.py', 'refactored\_file\_processor.py', 'refactored\_scientific.py', 'refactored\_student\_db.py', and 'system\_logs.txt'. The main editor window shows the 'legacy\_calculator.py' file, which contains a legacy Python script for a basic calculator. The script uses long, repetitive if-else statements for each operation. The terminal at the bottom shows the output of running the script, which prompts the user to enter a first number, a second number, and an operation, and then displays the result.

```
1 # legacy_calculator.py
2
3 print("Simple Calculator")
4 a = float(input("Enter first number: "))
5 b = float(input("Enter second number: "))
6 op = input("Enter operation (+, -, *, /): ")
7
8 if op == '+':
9     print("Result:", a + b)
10 elif op == '-':
11     print("Result:", a - b)
12 elif op == '*':
13     print("Result:", a * b)
14 elif op == '/':
15     if b != 0:
16         print("Result:", a / b)
17     else:
18         print("Error: Division by zero")
19 else:
20     print("Invalid operation")
21
```

Terminal Output:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\sakir\AppData\Local\Programs\Python\Python313\python.exe "d:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13\legacy_calculator.py"
Simple Calculator
Enter first number: 5
Enter second number: 3
Enter operation (+, -, *, /): +
Result: 8.0
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

## Refactored Code & Output :



The screenshot shows a VS Code editor with a file named `refactored_calculator.py` open. The code is a Python script that implements a calculator with a menu-driven interface. It uses a dictionary for operations and includes error handling for invalid inputs. The terminal output shows the program running and processing user input.

```
1 # refactored_calculator.py
2
3 def add(x, y): return x + y
4 def sub(x, y): return x - y
5 def mul(x, y): return x * y
6 def div(x, y): return x / y if y != 0 else "Error: Division by zero"
7
8 operations = {
9     '+': add,
10    '-': sub,
11    '*': mul,
12    '/': div
13 }
14
15 def calculator():
16     print("AI-Refactored Calculator")
17     try:
18         a = float(input("Enter first number: "))
19         b = float(input("Enter second number: "))
20         op = input("Enter operation (+, -, *, /): ")
21         result = operations.get(op, lambda x, y: "Invalid operation")(a, b)
22         print("Result:", result)
23     except ValueError:
24         print("Error: Please enter valid numbers.")
25
26 if __name__ == "__main__":
27     calculator()
28
```

The terminal output shows the following sequence of events:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiir\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/AI Assisted Coding/LAB ASSIGNMENTS/Lab 13/refactored_calculator.py"
AI-Refactored Calculator
Enter first number: 5
Enter second number: 6
Enter operation (+, -, *, /): +
Result: 11.0
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

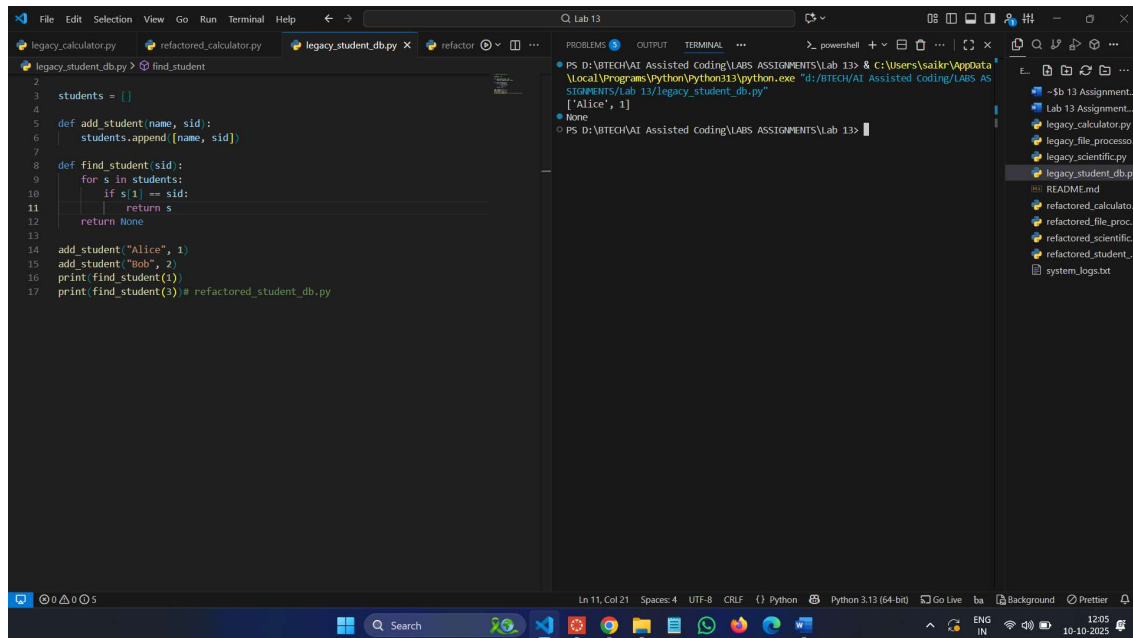
## Task 2: Modernizing a Student Database Program

### 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.

## Legacy Code & Output :



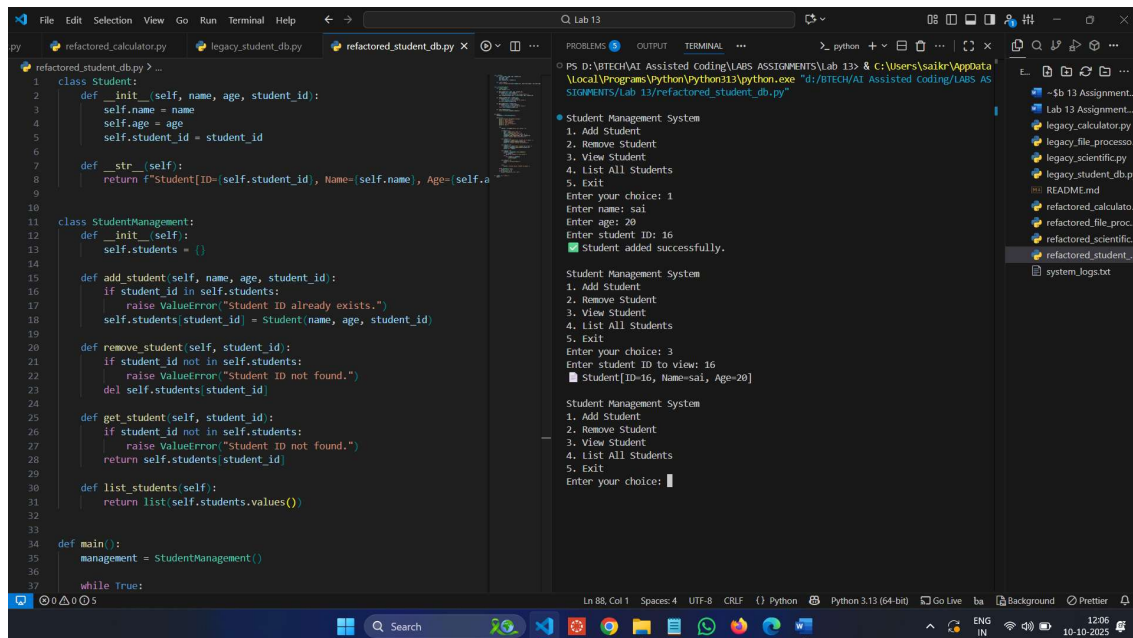
The screenshot shows a VS Code editor with a file named `legacy_student_db.py`. The code defines a list `students` and functions `add_student`, `find_student`, and `main`. The `main` function calls `add_student` twice and `find_student` three times. The terminal output shows the execution of the script, which prints the names of the added students and the result of the find operations.

```
1 # legacy_student_db.py
2
3 students = []
4
5 def add_student(name, sid):
6     students.append([name, sid])
7
8 def find_student(sid):
9     for s in students:
10         if s[1] == sid:
11             return s
12     return None
13
14 add_student("Alice", 1)
15 add_student("Bob", 2)
16 print(find_student(1))
17 print(find_student(3)) # refactored_student_db.py
```

Terminal Output:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> python.exe "d:/BTECH/AI Assisted Coding/LAB ASSIGNMENTS/Lab 13/legacy_student_db.py"
None
None
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

## Refactored Code & Output :



The screenshot shows a VS Code editor with a file named `refactored_student_db.py`. The code defines two classes: `Student` and `StudentManagement`. The `Student` class has attributes `name`, `age`, and `student_id`. The `StudentManagement` class has a list `students` and methods `add_student`, `remove_student`, `get_student`, and `list_students`. The `main` function creates a `StudentManagement` object and enters a loop where it prompts the user to perform various operations. The terminal output shows the execution of the script, which prompts the user for choices and displays the results of the operations.

```
1 # refactored_student_db.py
2
3 class Student:
4     def __init__(self, name, age, student_id):
5         self.name = name
6         self.age = age
7         self.student_id = student_id
8
9     def __str__(self):
10         return f"Student ID={self.student_id}, Name={self.name}, Age={self.age}"
11
12 class StudentManagement:
13     def __init__(self):
14         self.students = {}
15
16     def add_student(self, name, age, student_id):
17         if student_id in self.students:
18             raise ValueError("Student ID already exists.")
19         self.students[student_id] = Student(name, age, student_id)
20
21     def remove_student(self, student_id):
22         if student_id not in self.students:
23             raise ValueError("Student ID not found.")
24         del self.students[student_id]
25
26     def get_student(self, student_id):
27         if student_id not in self.students:
28             raise ValueError("Student ID not found.")
29         return self.students[student_id]
30
31     def list_students(self):
32         return list(self.students.values())
33
34 def main():
35     management = StudentManagement()
36
37     while True:
```

Terminal Output:

```
Student Management System
1. Add Student
2. Remove Student
3. View Student
4. List All Students
5. Exit
Enter your choice: 1
Enter name: sai
Enter age: 20
Enter student ID: 16
Student added successfully.

Student Management System
1. Add Student
2. Remove Student
3. View Student
4. List All Students
5. Exit
Enter your choice: 3
Enter student ID to view: 16
Student ID=16, Name=sai, Age=20

Student Management System
1. Add Student
2. Remove Student
3. View Student
4. List All Students
5. Exit
Enter your choice: 1
```

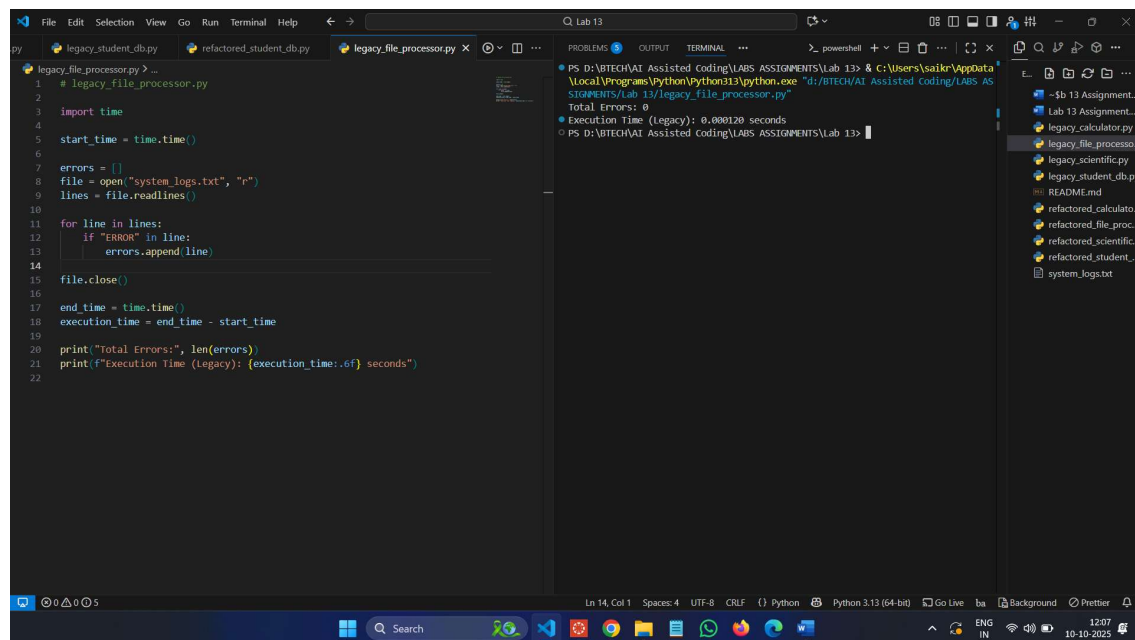
### Task 3: Optimizing Performance in File Processing

#### 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.

#### Legacy Code & Output :



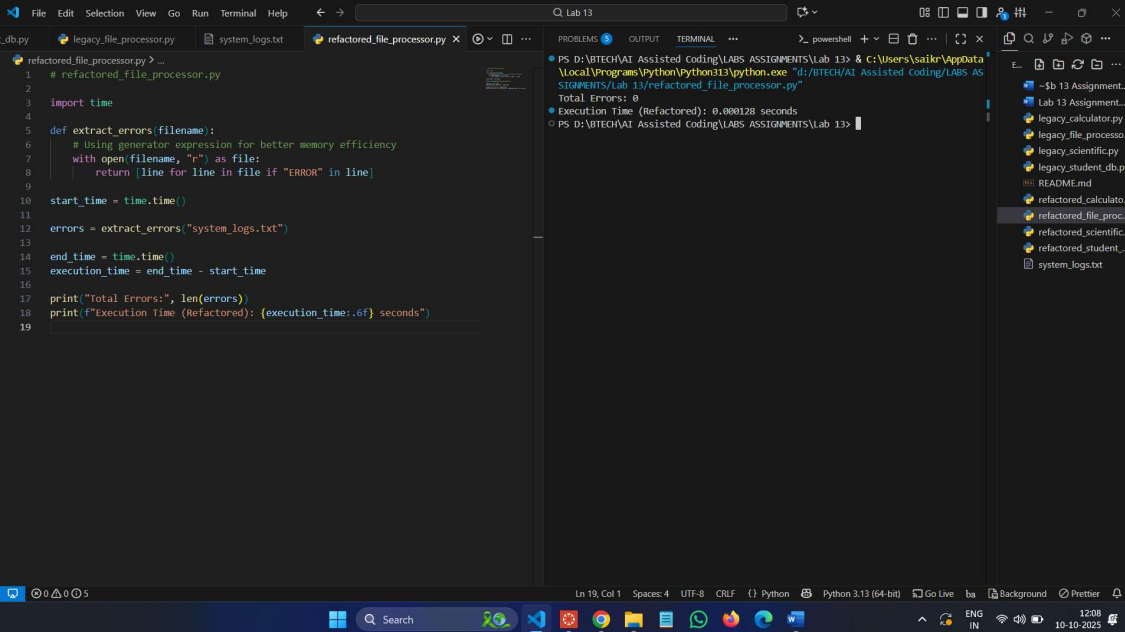
The screenshot shows a Visual Studio Code editor with a Python file named `legacy_file_processor.py` open. The code is a legacy script that reads a file line by line and checks for errors. The output window shows the execution time of the legacy script as 0.000120 seconds.

```
1 # legacy_file_processor.py
2
3 import time
4
5 start_time = time.time()
6
7 errors = []
8 file = open("system_logs.txt", "r")
9 lines = file.readlines()
10
11 for line in lines:
12     if "ERROR" in line:
13         errors.append(line)
14
15 file.close()
16
17 end_time = time.time()
18 execution_time = end_time - start_time
19
20 print("Total Errors:", len(errors))
21 print(f"Execution Time (Legacy): {execution_time:.6f} seconds")
22
```

Output:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saike\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/ai Assisted coding/LAB ASSIGNMENTS/Lab 13/legacy_file_processor.py"
Total Errors: 0
Execution Time (Legacy): 0.000120 seconds
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

## Refactored Code & Output :



The screenshot shows a Visual Studio Code editor with a Python file named `refactored_file_processor.py` open. The code is a refactored version of a file processor, featuring a function `extract_errors` that uses a generator expression for better memory efficiency. The script processes `system_logs.txt` and prints the total errors and execution time.

```
1 # refactored_file_processor.py
2
3 import time
4
5 def extract_errors(filename):
6     # Using generator expression for better memory efficiency
7     with open(filename, "r") as file:
8         return (line for line in file if "ERROR" in line)
9
10 start_time = time.time()
11
12 errors = extract_errors("system_logs.txt")
13
14 end_time = time.time()
15 execution_time = end_time - start_time
16
17 print("Total Errors:", len(errors))
18 print(f"Execution Time (Refactored): {execution_time:.6f} seconds")
19
```

The terminal output shows the execution results:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiir\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/AI Assisted Coding/LAB ASSIGNMENTS/Lab 13/refactored_file_processor.py"
Total Errors: 0
Execution Time (Refactored): 0.000128 seconds
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

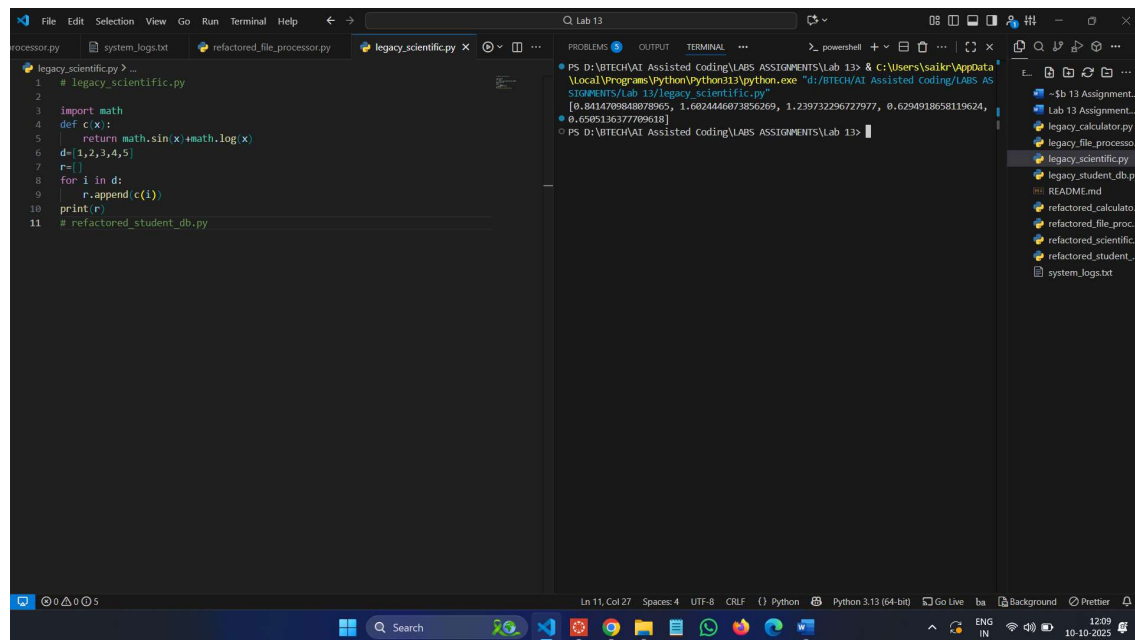
## Task 4: Enhancing Readability and Documentation

### 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.

## Legacy Code & Output :

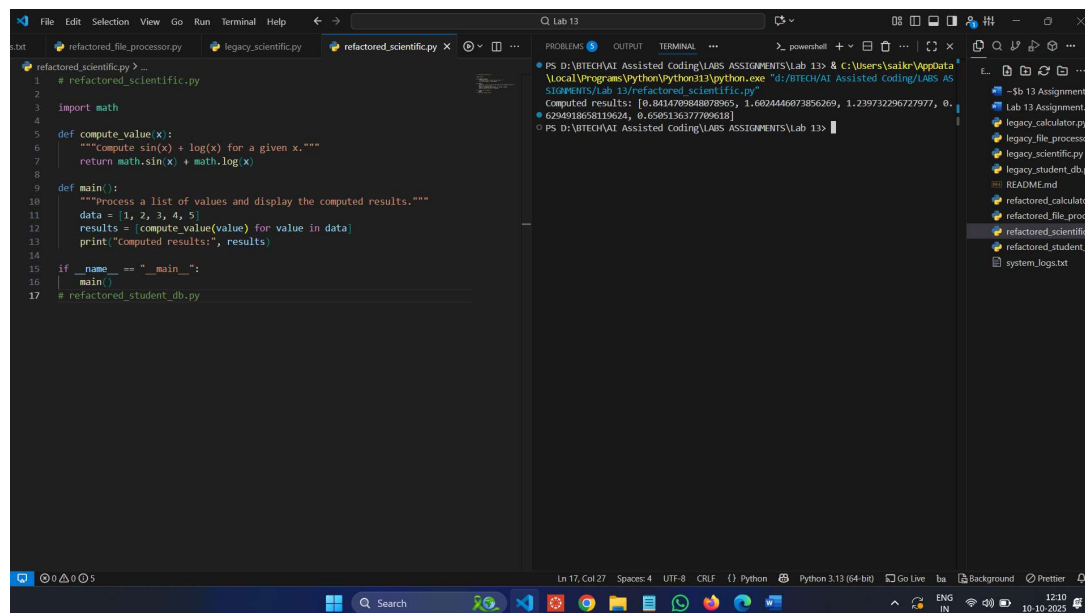


The screenshot shows a Visual Studio Code editor with a file named `legacy_scientific.py` open. The code is a simple script that defines a function `c(x)` which returns  $\sin(x) + \log(x)$ . It then iterates over a list `d = [1, 2, 3, 4, 5]`, appends the result of `c(i)` to a list `r`, and finally prints `r`. The output window on the right shows the execution results: `[0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.6294918658119624, 0.6505136377789618]`. The terminal window shows the command used to run the script: `PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiKr\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/ai assisted coding/LABs ASSIGNMENTS/Lab 13/legacy_scientific.py"`.

```
1 # legacy_scientific.py
2
3 import math
4 def c(x):
5     return math.sin(x)+math.log(x)
6 d=[1,2,3,4,5]
7 r=[]
8 for i in d:
9     r.append(c(i))
10 print(r)
11 # refactored_student_db.py
```

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiKr\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/ai assisted coding/LABs ASSIGNMENTS/Lab 13/legacy_scientific.py"
[0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.6294918658119624, 0.6505136377789618]
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

## Refactored Code & Output :



The screenshot shows a Visual Studio Code editor with a file named `refactored_scientific.py` open. The code is a refactored version of the legacy code, using a function `compute_value(x)` to calculate  $\sin(x) + \log(x)$ . It then uses a list comprehension to process a list of values `data = [1, 2, 3, 4, 5]` and displays the results. The output window on the right shows the execution results: `Computed results: [0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.6294918658119624, 0.6505136377789618]`. The terminal window shows the command used to run the script: `PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiKr\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/ai assisted coding/LABs ASSIGNMENTS/Lab 13/refactored_scientific.py"`.

```
1 # refactored_scientific.py
2
3 import math
4
5 def compute_value(x):
6     """Compute sin(x) + log(x) for a given x."""
7     return math.sin(x) + math.log(x)
8
9 def main():
10     """Process a list of values and display the computed results."""
11     data = [1, 2, 3, 4, 5]
12     results = [compute_value(value) for value in data]
13     print("Computed results:", results)
14
15 if __name__ == "__main__":
16     main()
17 # refactored_student_db.py
```

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiKr\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/ai assisted coding/LABs ASSIGNMENTS/Lab 13/refactored_scientific.py"
Computed results: [0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.6294918658119624, 0.6505136377789618]
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

## Observation

1. The legacy programs were unorganized, repetitive, and lacked error handling.
2. After refactoring with AI tools, the codes became cleaner and easier to understand.
3. Using functions, classes, and comprehensions improved performance and readability.
4. Error handling and documentation made the programs more reliable.
5. Overall, the refactored versions are modern, efficient, and easier to maintain.