

Assignment

Lab 13 – Code Refactoring: Improving Legacy Code with AI

HALLTICKET NUMBER : 2503A52416

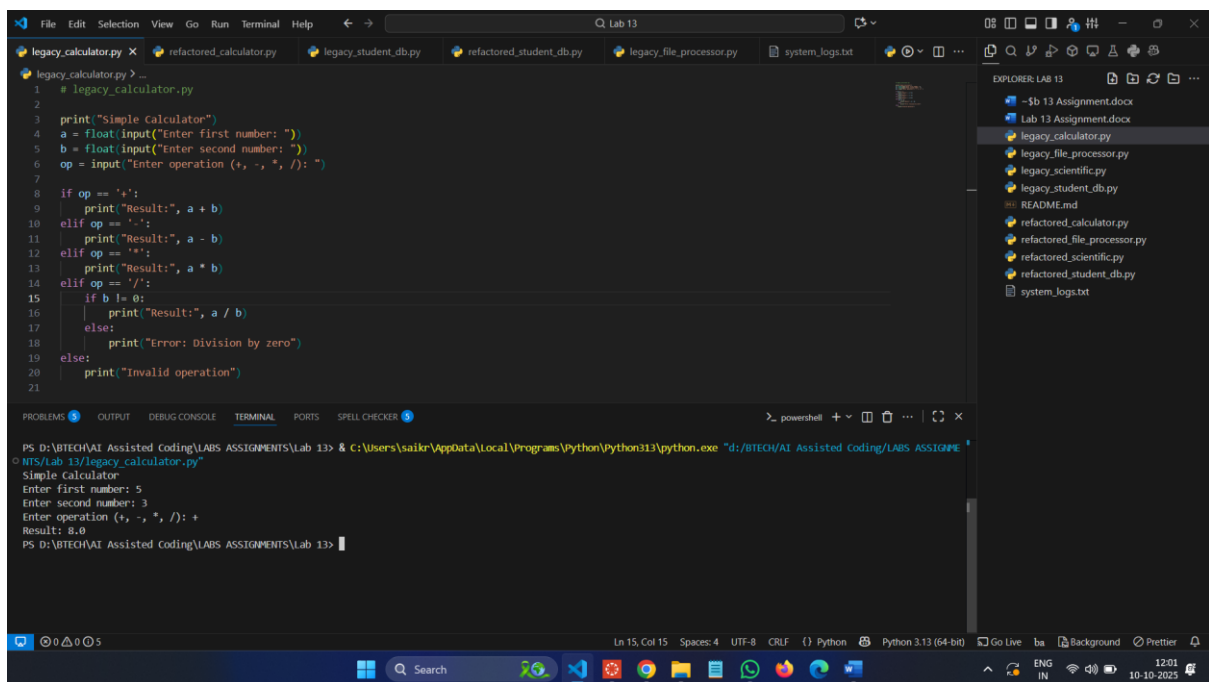
Task 1: Refactoring a Legacy Calculator Script

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 :



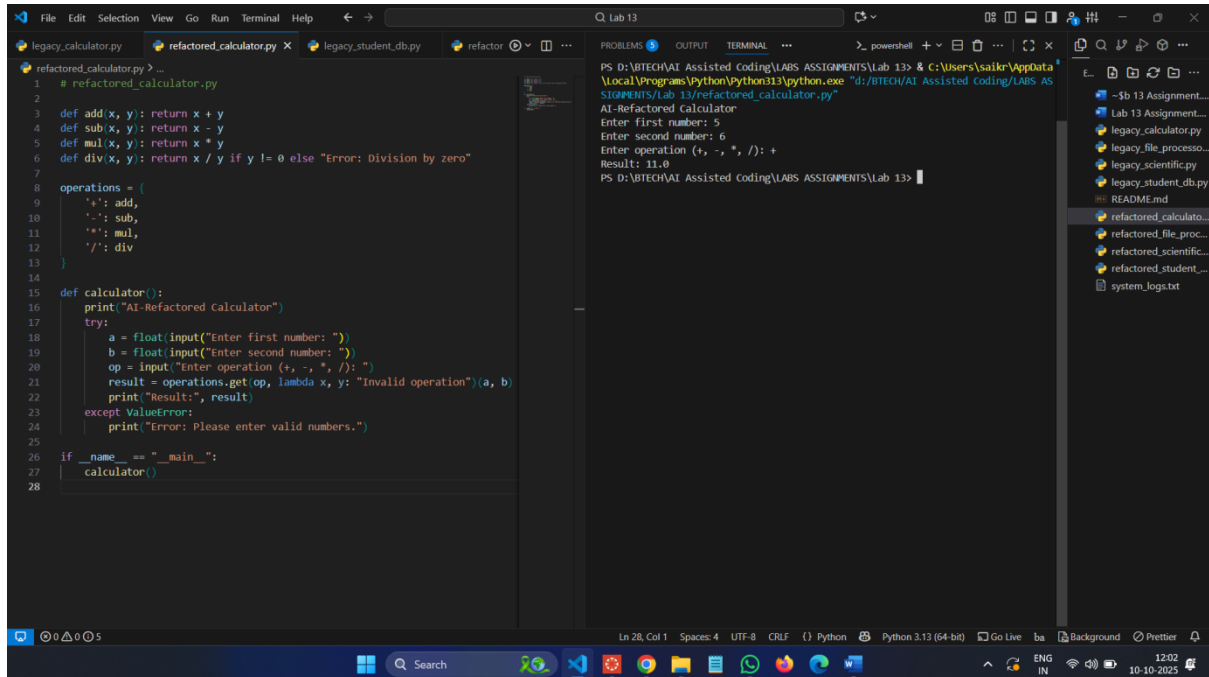
The screenshot shows a Visual Studio Code editor with a file explorer on the right and a terminal at the bottom. The file explorer shows a project named 'Lab 13' with several files: 'Lab 13 Assignment.docx', 'legacy_calculator.py', 'legacy_file_processor.py', 'legacy_scientific.py', 'legacy_student_db.py', 'README.md', 'refactored_calculator.py', 'refactored_file_processor.py', 'refactored_scientific.py', 'refactored_student_db.py', and 'system_logs.txt'. The main editor window displays the code for 'legacy_calculator.py'.

```
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
```

The terminal shows the output of the script:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saiqr\AppData\Local\Programs\Python\Python313\python.exe "d:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\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 the file `refactored_calculator.py` open. The code is as follows:

```
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 program running successfully:

```
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>
```

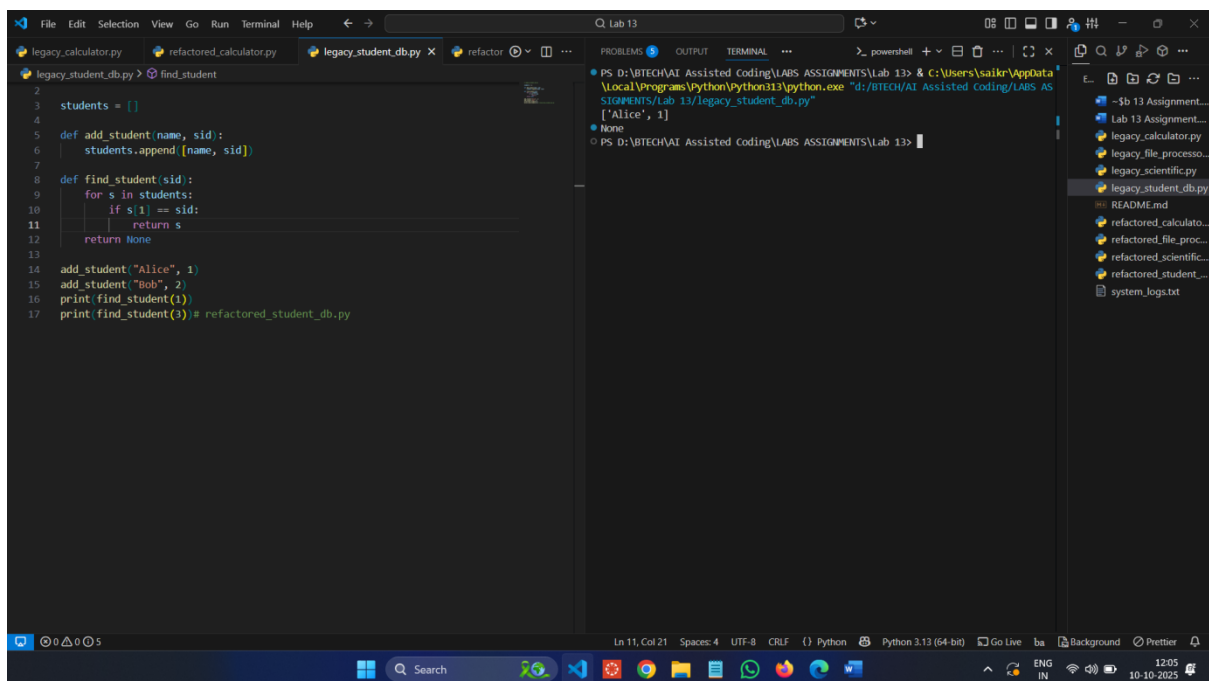
Task 2: Modernizing a Student Database Program

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 Visual Studio Code editor with a dark theme. The main editor window displays a Python file named `legacy_student_db.py`. The code defines a list `students` and two functions: `add_student` and `find_student`. The `find_student` function iterates through the `students` list to find a student by their `sid`. The code is as follows:

```
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
```

The right sidebar shows the `TERMINAL` panel with the following output:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\sai\AppData\Local\Programs\Python\Python313\python.exe "d:/BTECH/AI Assisted Coding/LAB ASSIGNMENTS/Lab 13/legacy_student_db.py"
['Alice', 1]
None
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```

The bottom status bar indicates the file is at line 11, column 21, with 4 spaces, UTF-8 encoding, CRLF line endings, and is a Python 3.13 (64-bit) file.

Refactored Code &Output :

The screenshot shows a VS Code editor with a Python 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 `students` list and methods for adding, removing, getting, and listing students. A `main` function uses a `while` loop to interact with the `StudentManagement` object.

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

The terminal output shows the execution of the script. It prompts the user to enter their choice (1-5). Choice 1 (Add Student) is selected, and the user enters name 'sai', age 20, and student ID 16. The output confirms 'Student added successfully.' Choice 3 (View Student) is then selected, and the user enters student ID 16. The output shows 'Student[ID=16, Name=sai, Age=20]'. Choice 5 (Exit) is then selected.

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 :

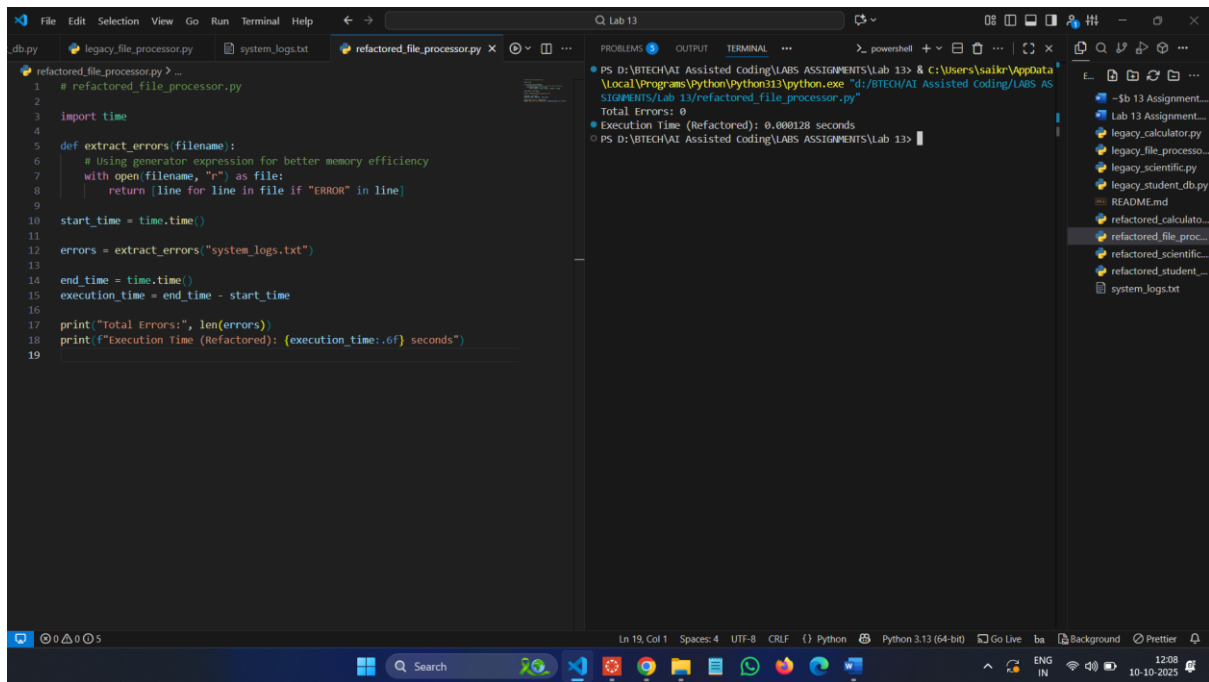
The screenshot shows a Visual Studio Code editor window with a Python file named `legacy_file_processor.py` open. The code is a legacy script that reads lines from `system_logs.txt` and counts the number of errors. The script includes timing logic to measure execution time. The output panel shows the execution results, indicating that the script ran successfully with 0 errors and a very short execution time.

```
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\satir\AppData\Local\Programs\Python\Python313\python.exe "d:\BTECH\AI Assisted Coding\LABS 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 :



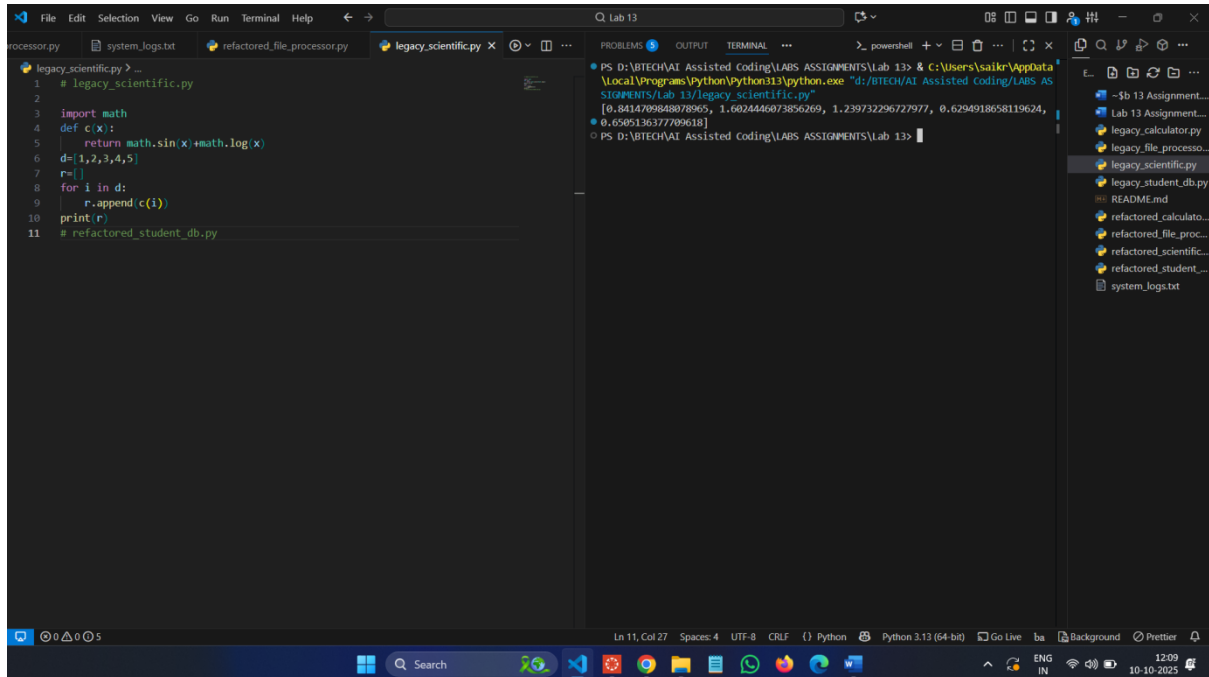
Task 4: Enhancing Readability and Documentation

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 :



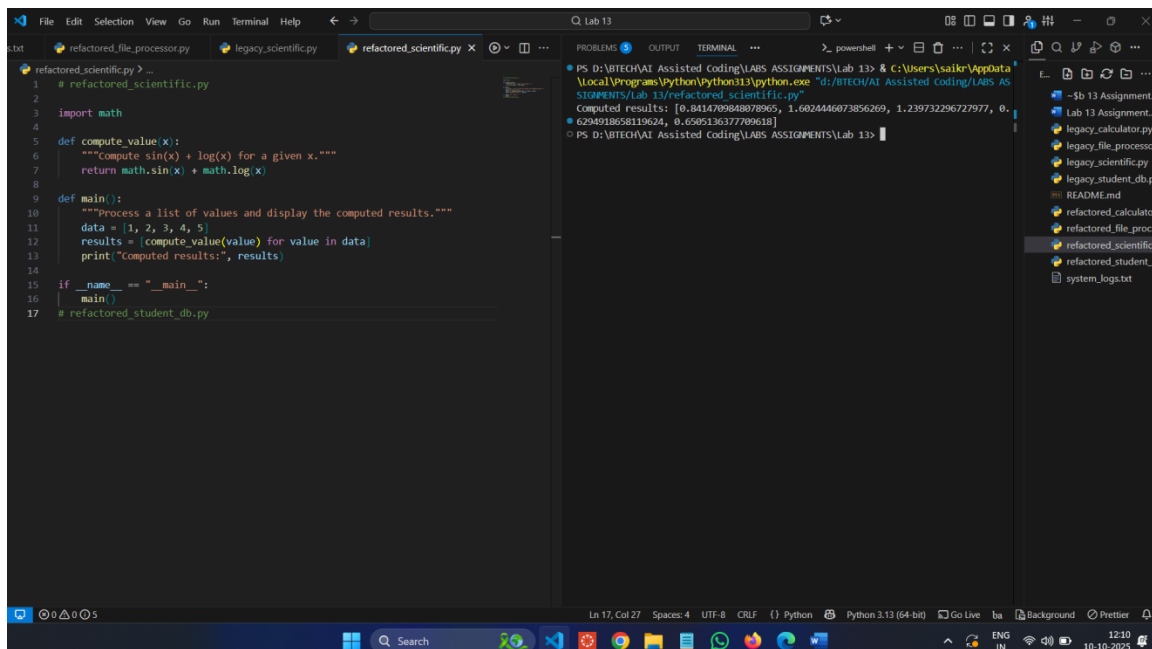
The screenshot shows a Visual Studio Code editor with a file named `legacy_scientific.py`. The code is as follows:

```
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
```

The output window on the right 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/legacy_scientific.py"
[0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.629491858119624, 0.6505136377709618]
```

Refactored Code & Output :



The screenshot shows a Visual Studio Code editor with a file named `refactored_scientific.py`. The code is as follows:

```
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
```

The output window on the right 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_scientific.py"
Computed results: [0.8414709848078965, 1.6024446073856269, 1.239732296727977, 0.629491858119624, 0.6505136377709618]
```

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.