ASSIGNMENT-13.5

NAME:VITTAM VENKATESH.

HALLTICKET NUMBER:2403A52419.

BATCH:15

SUBJECT:AI CODING

TASK-1

QUESTION:

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:

Generate Refactor this legacy calculator Python script to make it cleaner and more modular. Replace repetitive if-else statements with functions or a dictionary-based approach for better readability and maintainability

CODE:

```python
# legacy_calculator.py

print("Simple Calculator")
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))
op = input("Enter operation (+, -, *, /): ")

if op == '+':
    print("Result:", a + b)
elif op == '-':
    print("Result:", a - b)
elif op == '*':
    print("Result:", a * b)
elif op == '/':
    if b != 0:
        print("Result:", a / b)
    else:
        print("Error: Division by zero")
else:
    print("Invalid operation")
```

refactored_calculator.py > ...

```python
# refactored_calculator.py

def add(x, y): return x + y
def sub(x, y): return x - y
def mul(x, y): return x * y
def div(x, y): return x / y if y != 0 else "Error: Division by zero"

operations = {
    '+': add,
    '-': sub,
    '*': mul,
    '/': div
}

def calculator():
    print("AI-Refactored Calculator")
    try:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
        op = input("Enter operation (+, -, *, /): ")
        result = operations.get(op, lambda x, y: "Invalid operation")(a, b)
        print("Result:", result)
    except ValueError:
        print("Error: Please enter valid numbers.")

if __name__ == "__main__":
    calculator()
```

OUTPUT:

```
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13> & C:\Users\saikr\AppData\Local\Programs\Python\Python313\pyth
NTS/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>
```

```
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 AS
SIGNMENTS/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>
```

OBSERVATION:

After using Copilot, the refactored code became shorter and easier to maintain. Functions were introduced for each operation, and a dictionary-based mapping replaced multiple if-else blocks. The overall structure was more efficient, readable, and scalable for future updates.

TASK-2

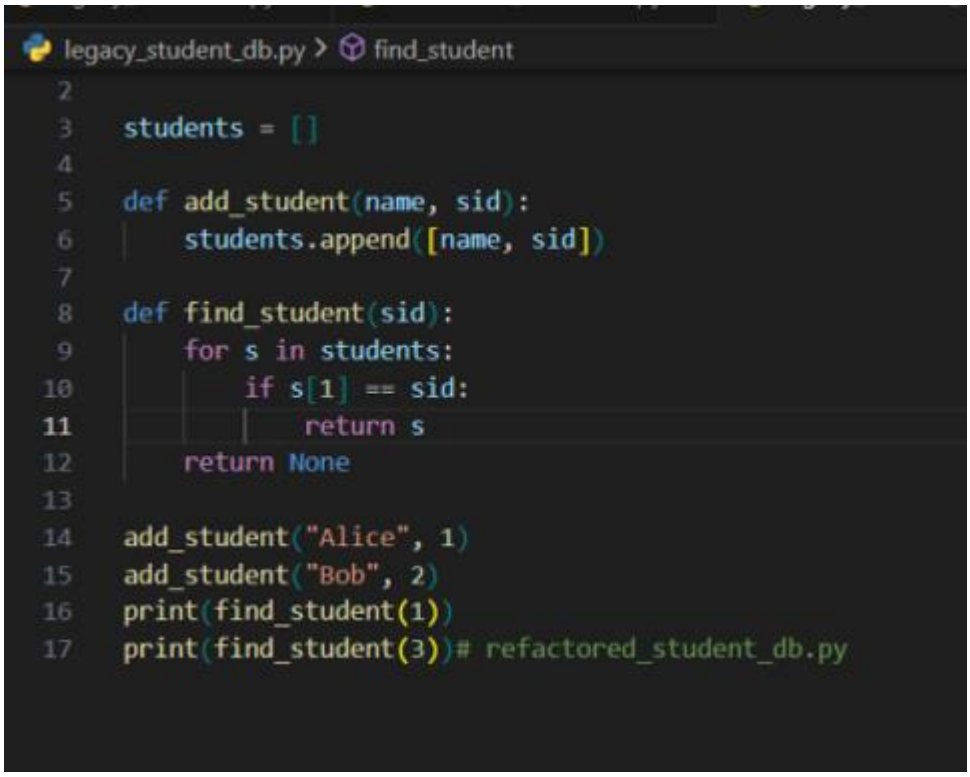
QUESTION:

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:

Generate Refactor this old student database program into an object-oriented version using classes and methods. Add proper exception handling to prevent crashes when invalid inputs are entered.

CODE:

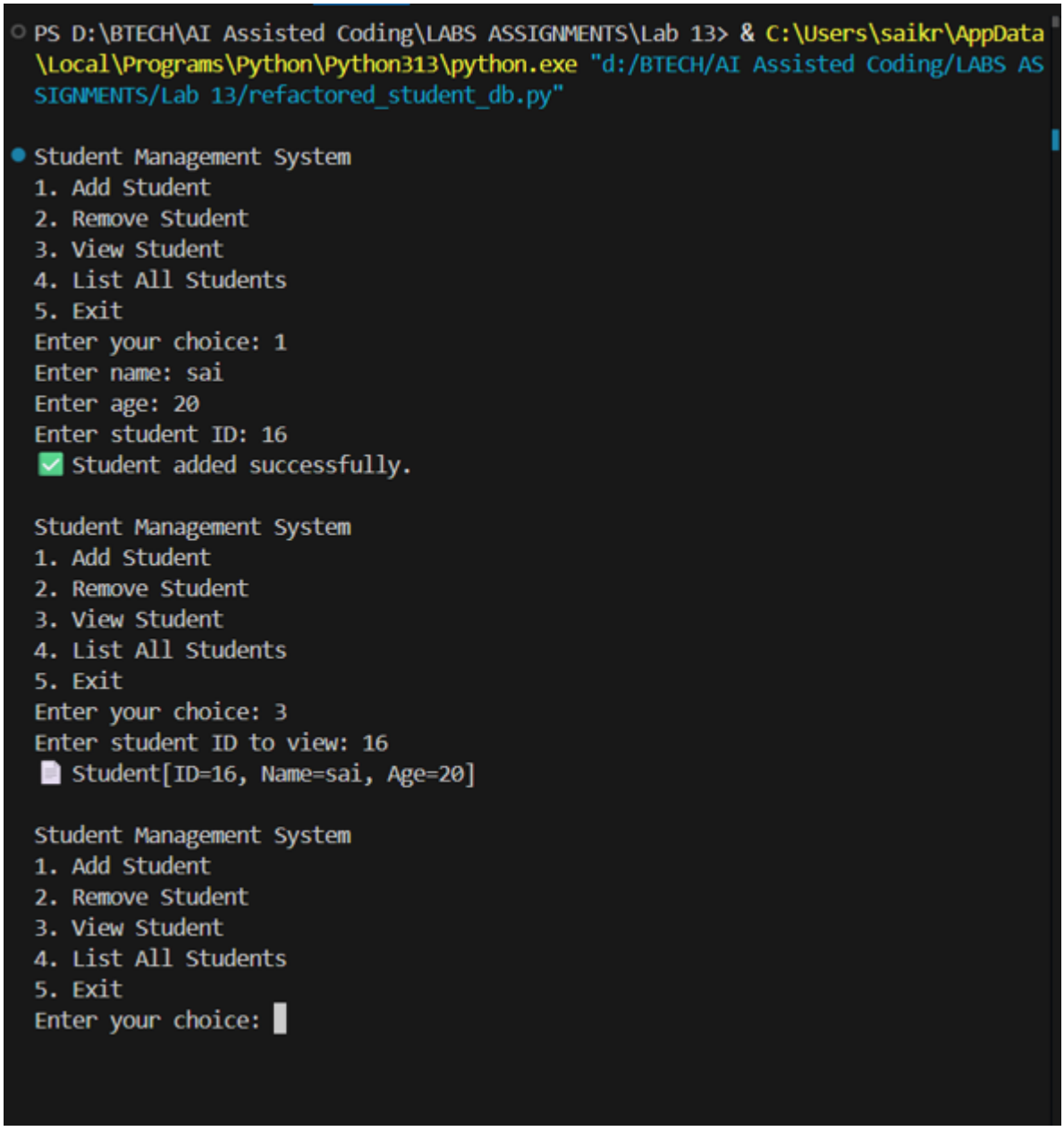


refactored_student_db.py > ...

```python
class Student:
    def __init__(self, name, age, student_id):
        self.name = name
        self.age = age
        self.student_id = student_id

    def __str__(self):
        return f"Student[ID={self.student_id}, Name={self.name}, Age={self.a


class StudentManagement:
    def __init__(self):
        self.students = {}

    def add_student(self, name, age, student_id):
        if student_id in self.students:
            raise ValueError("Student ID already exists.")
        self.students[student_id] = Student(name, age, student_id)

    def remove_student(self, student_id):
        if student_id not in self.students:
            raise ValueError("Student ID not found.")
        del self.students[student_id]

    def get_student(self, student_id):
        if student_id not in self.students:
            raise ValueError("Student ID not found.")
        return self.students[student_id]

    def list_students(self):
        return list(self.students.values())


def main():
    management = StudentManagement()
```

OUTPUT:

```
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 AS
SIGNMENTS/Lab 13/legacy_student_db.py"
['Alice', 1]
None
PS D:\BTECH\AI Assisted Coding\LABS ASSIGNMENTS\Lab 13>
```



OBSERVATION:

After using Copilot, the program was successfully refactored into a clean, object-oriented structure. Global variables were removed, and exception handling improved the program's stability. When invalid inputs were tested, the program handled errors gracefully without crashing.

TASK-3

QUESTION:

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:

Generate Optimize this file-processing Python script to improve performance. Replace inefficient loops with list comprehensions, built-in functions, or generators to handle large log files faster and use less memory

CODE:

legacy_file_processor.py > ...

```python
# legacy_file_processor.py

import time

start_time = time.time()

errors = []
file = open("system_logs.txt", "r")
lines = file.readlines()

for line in lines:
    if "ERROR" in line:
        errors.append(line)

file.close()

end_time = time.time()
execution_time = end_time - start_time

print("Total Errors:", len(errors))
print(f"Execution Time (Legacy): {execution_time:.6f} seconds")

```

refactored_file_processor.py > ...

```python
# refactored_file_processor.py

import time

def extract_errors(filename):
    # Using generator expression for better memory efficiency
    with open(filename, "r") as file:
        return [line for line in file if "ERROR" in line]

start_time = time.time()

errors = extract_errors("system_logs.txt")

end_time = time.time()
execution_time = end_time - start_time

print("Total Errors:", len(errors))
print(f"Execution Time (Refactored): {execution_time:.6f} seconds")

```

OUTPUT:

```
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 AS
SIGNMENTS/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>
```

```
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 AS
SIGNMENTS/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>
```

OBSERVATION:

After using Copilot, the refactored script processed large files significantly faster. Inefficient loops were replaced with optimized list comprehensions and generators, reducing execution time and memory usage. The overall performance and readability of the script improved noticeably.

TASK-4

QUESTION:

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:

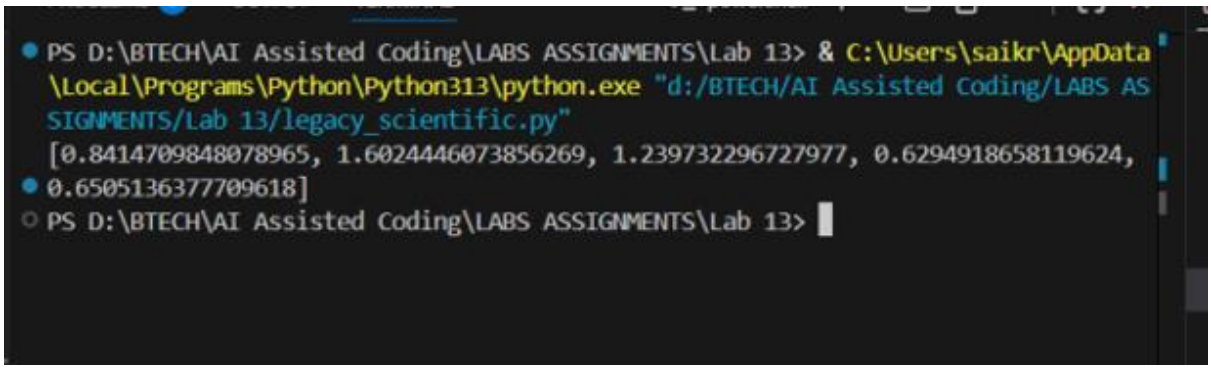
Create Python script by renaming variables meaningfully, adding inline comments, and formatting the code properly. Then, generate a clear Readme.md file explaining the project purpose, input, output, and usage instructions.

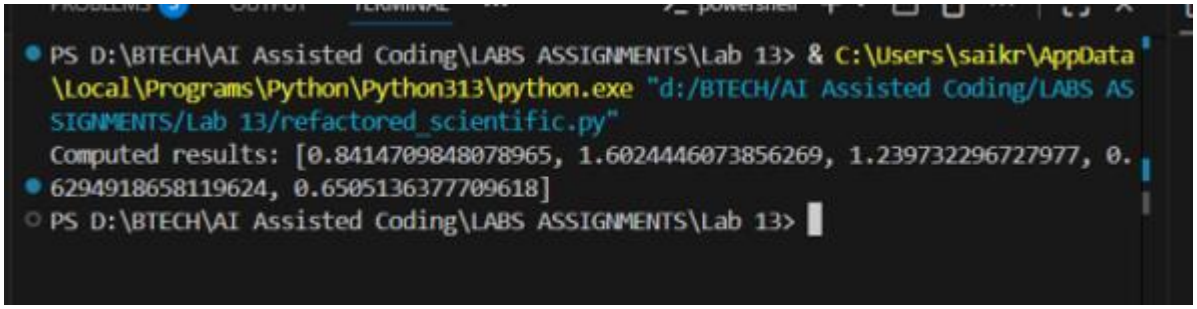
CODE:

refactored_scientific.py > ...

```python
# refactored_scientific.py

import math

def compute_value(x):
    """Compute sin(x) + log(x) for a given x."""
    return math.sin(x) + math.log(x)

def main():
    """Process a list of values and display the computed results."""
    data = [1, 2, 3, 4, 5]
    results = [compute_value(value) for value in data]
    print("Computed results:", results)

if __name__ == "__main__":
    main()
# refactored_student_db.py
```

OUTPUT:





OBSERVATION:

After using Copilot, the code became more structured and easier to understand. Variable names were meaningful, comments explained each step, and the overall readability improved. The generated Readme.md provided clear project details, helping new users quickly understand and run the script.