ASSIGNMENT-13

NAME: K. NAVYA SRI

ROLL NO: 2503A52L12

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.

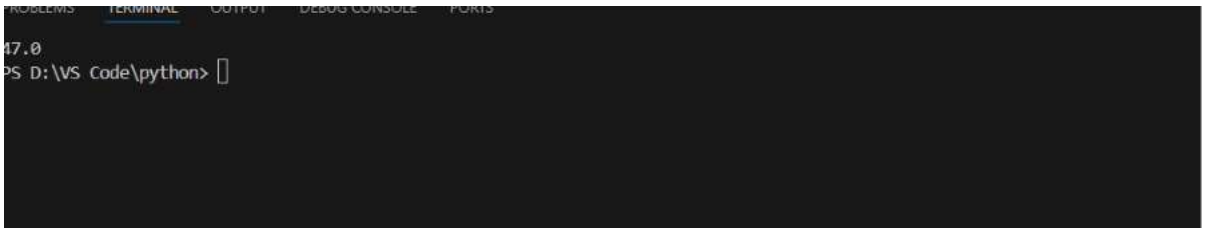
- Upload the calculator script to a GitHub repository.
- Use GitHub Copilot to suggest a more modular and cleaner version (e.g., functions, dictionary-based mapping).
- Compare the AI-suggested refactoring with the original code and document improvements.

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

CODE:

```python
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y

operations = {
    '+': add,
    '-': subtract,
    '*': multiply,
    '/': divide
}

num1 = float(input("Enter first number: "))
op = input("Enter operation (+, -, *, /): ")
num2 = float(input("Enter second number: "))

if op in operations:
    print(operations[op](num1, num2))
else:
    print("Invalid operator")

```

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:

AIAC_LAB_13.PY > ...

```python
class Student:
    def __init__(self, name, roll, marks):
        self.name = name
        self.roll = roll
        self.marks = marks

class StudentManager:
    def __init__(self):
        self.students = []

    def add_student(self, name, roll, marks):
        # Validate inputs
        try:
            if not isinstance(roll, int):
                raise ValueError("Roll number must be an integer.")
            if self.find_student(roll):
                raise Exception("Duplicate roll number.")
            self.students.append(Student(name, roll, marks))
        except Exception as e:
            print("Error adding student:", e)

    def display_students(self):
        for student in self.students:
            print(f"Name: {student.name}, Roll: {student.roll}, Marks: {student.marks}")

    def find_student(self, roll):
        for student in self.students:
            if student.roll == roll:
                return student
        return None

    def delete_student(self, roll):
        try:
            student = self.find_student(roll)
            if not student:
                raise Exception("Student not found.")
            self.students.remove(student)
        except Exception as e:
            print("Error deleting student:", e)

# Example usage:
manager = StudentManager()
manager.add_student("A", 1, [100, 100])
manager.add_student("B", "2", [90, 90]) # Invalid roll input
manager.add_student("C", 1, [80, 80])   # Duplicate roll
manager.display_students()
manager.delete_student(99)              # Non-existing student

```

OUTPUT:

```
Error deleting student: Student not found.
PS D:\VS Code\python> python -u "d:\VS Code\python\AIAC_LAB_13.PY"
Error adding student: Roll number must be an integer.
Error adding student: Duplicate roll number.
Name: A, Roll: 1, Marks: [100, 100]
Error deleting student: Student not found.
PS D:\VS Code\python>
```

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:

```python
import os

def error_lines(filename):
    try:
        with open(filename, 'r') as file:
            for line in file:
                if "ERROR" in line:
                    yield line.upper()
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return
    except IOError as e:
        print(f"Error opening file '{filename}': {e}")
        return

def count_error_lines(filename):
    count = 0
    for _ in error_lines(filename):
        count += 1
    return count

if __name__ == "__main__":
    filename = "logfile.txt"  # You can set full path if needed, e.g. r"D:\VS Code\python\logfile.txt"

    if not os.path.isfile(filename):
        print(f"File '{filename}' does not exist. Please check the path.")
    else:
        count = count_error_lines(filename)
        print(f"Number of error lines: {count}")
```

OUTPUT:

```
File 'logfile.txt' does not exist. Please check the path.
PS D:\VS Code\python> python -u "d:\VS Code\python\AIAC_LAB_13.PY"
File 'logfile.txt' does not exist. Please check the path.
PS D:\VS Code\python>
```

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:

```python
"""
Example Scientific Computation Module

This module provides functions for basic scientific computations such as
calculating the dot product of two vectors.
"""

def dot_product(vector1, vector2):
    """
    Calculate the dot product of two equal-length vectors.

    Parameters:
        vector1 (list of float): First input vector.
        vector2 (list of float): Second input vector.

    Returns:
        float: Dot product of the two vectors.

    Raises:
        ValueError: If the input vectors have different lengths.
    """
    if len(vector1) != len(vector2):
        raise ValueError("Vectors must be of the same length.")

    result = 0.0  # Initialize accumulator for sum of products
    for i in range(len(vector1)):
        result += vector1[i] * vector2[i]  # Multiply element-wise and accumulate
    return result

def main():
    """
    Main function demonstrating usage of the dot_product function.
    """
    # Example input vectors
    vector_a = [1.0, 2.0, 3.0]
    vector_b = [4.0, 5.0, 6.0]

    try:
        product = dot_product(vector_a, vector_b)
        print(f"The dot product of {vector_a} and {vector_b} is: {product}")
    except ValueError as e:
        print(f"Error: {e}")

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

OUTPUT:

```
PROBLEMS  TERMINAL  OUTPUT  DEBUG CONSOLE  PORTS

The dot product of [1.0, 2.0, 3.0] and [4.0, 5.0, 6.0] is: 32.0
PS D:\VS Code\python> python -u "d:\VS Code\python\AIAC_LAB_13.PY"
The dot product of [1.0, 2.0, 3.0] and [4.0, 5.0, 6.0] is: 32.0
PS D:\VS Code\python>
```

- **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