LAB ASSIGNMENT-13.2

< AI Assisted Coding >

NAME : V.Vamshi

HALLTICKETNUMBER : 2403A52016

BATCH NUMBER : 02

TASK_1

Task Description #1 – Remove Repetition

Task: Provide AI with the following redundant code and ask it to refactor

PYTHON CODE:

```
def calculate_area(shape, x, y=0):
  if shape == "rectangle":
    return x * y
  elif shape == "square":
    return x * x
  elif shape == "circle":
    return 3.14 * x * x
```

PROMPT:

Refactor the following Python code to remove repetition and make it modular and clean. Use either a **dictionary-based dispatch system** or **separate functions** for each shape. Add **dynamic user input handling**, so the program asks the user which shape they want, and then requests the required parameters (like radius, sides, etc.) dynamically based on that shape.Finally, display the calculated area in a user-friendly format.

CODE:

```
def calculate_area(shape, x, y=0):
    """
    Calculates the area of different shapes.

    Args:
        shape (str): The shape to calculate the area for ("rectangle", "square", or "circle").
        x (float): The first dimension (side length for square, radius for circle, length for rectangle).
        y (float, optional): The second dimension for a rectangle (width). Defaults to 0.

    Returns:
        float: The calculated area of the shape.
    """
    if shape == "rectangle":
        return x * y
    elif shape == "square":
        return x * x
    elif shape == "circle":
        return 3.14 * x * x
 # Example usage with dynamic input:
shape = input("Enter the shape (rectangle, square, or circle): ")
x = float(input("Enter the first dimension (side length, radius, or length): "))
y = 0
if shape == "rectangle":
    y = float(input("Enter the second dimension (width): "))

print(f"Area of the {shape}: {calculate_area(shape, x, y)}")
```

```
Enter the shape (rectangle, square, or circle): square
Enter the first dimension (side length, radius, or length): 4
Area of the square: 16.0
```

OBSERVATION:

This code defines a Python function calculate_area to compute the area of rectangles, squares, and circles. It takes the shape as a string and the necessary dimensions as numbers. The function then uses conditional statements to determine the shape and applies the appropriate area formula. Finally, the script prompts the user for input and displays the calculated area.

TASK_2

Task Description #2 – Error Handling in Legacy Code

Task: Legacy function without proper error handling

Python Code

def read_file(filename):
f = open(filename, "r")`
data = f.read()

f.close()
return data

Expected Output:

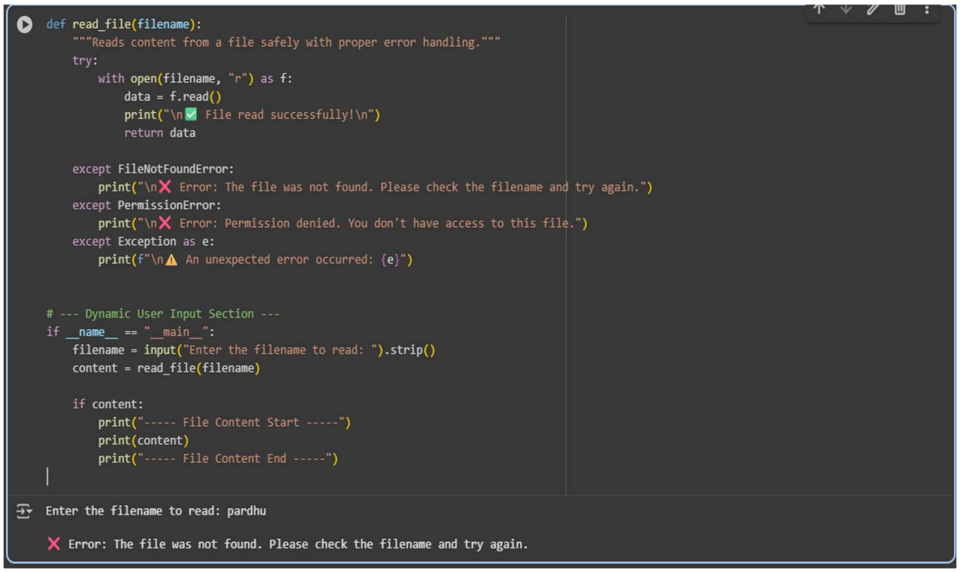
AI refactors with with open() and try-except:

PROMPT:

Refactor the following legacy Python code to include **modern file handling** using with open() and **robust error handling** with try-except blocks.

Additionally, modify the code to **accept the filename dynamically from the user** using input(), and display clear, user-friendly messages for success or failure.

CODE:



```
def read_file(filename):
    """Reads content from a file safely with proper error handling."""
    try:
        with open(filename, "r") as f:
            data = f.read()
            print("\n✅ File read successfully!\n")
            return data

    except FileNotFoundError:
        print("\n❌ Error: The file was not found. Please check the filename and try again.")
    except PermissionError:
        print("\n❌ Error: Permission denied. You don't have access to this file.")
    except Exception as e:
        print(f"\n⚠️ An unexpected error occurred: {e}")


# --- Dynamic User Input Section ---
if __name__ == "__main__":
    filename = input("Enter the filename to read: ").strip()
    content = read_file(filename)

    if content:
        print("----- File Content Start -----")
        print(content)
        print("----- File Content End -----")
```

```
Enter the filename to read: pardhu

❌ Error: The file was not found. Please check the filename and try again.
```

OBSERVATION:

The legacy code lacked proper error handling and manual file closing could lead to resource leaks.
By using the with open() statement, file handling becomes safer and more efficient.
The addition of try-except blocks ensures smooth execution even when errors occur.
Dynamic user input makes the program more flexible and interactive for real-world use.

TASK_3

Task Description #3 – Complex Refactoring

Task: Provide this legacy class to AI for readability and modularity

improvements:

Python Code

```
class Student:
def __init__(self, n, a, m1, m2, m3):
self.n = n
self.a = a
self.m1 = m1
self.m2 = m2
self.m3 = m3
def details(self):
```

print("Name:", self.n, "Age:", self.a)
def total(self):
return self.m1+self.m2+self.m3

Expected Output:

- AI improves naming (name, age, marks).
- Adds docstrings.
- Improves print readability.
- Possibly uses sum(self.marks) if marks stored in a list

PROMPT:

Refactor the following legacy Python class to improve **readability, naming conventions, and modularity**.Enhance code structure by using **clear variable names**, **docstrings**, and **more Pythonic constructs** (like storing marks in a list and using sum() for total calculation).Ensure the class remains functional and well-organized for future scalability.

CODE:

```python
class Student:
    """
    Represents a student with basic details and academic marks.

    Attributes:
        name (str): The name of the student.
        age (int): The age of the student.
        marks (list): A list containing marks of three subjects.
    """

    def __init__(self, name, age, marks):
        """
        Initializes a new Student object with name, age, and marks.
        """
        self.name = name
        self.age = age
        self.marks = marks  # list of marks instead of separate variables

    def display_details(self):
        """
        Displays the student's name and age in a readable format.
        """
        print(f"\n Student Details:\nName: {self.name}\nAge: {self.age}")

    def total_marks(self):
        """
        Returns the total of all marks.
        """
        return sum(self.marks)


# --- Example Usage ---
if __name__ == "__main__":
    # Dynamic user input
    name = input("Enter student name: ")
    age = int(input("Enter student age: "))

    marks = []
    for i in range(3):
        mark = float(input(f"Enter mark {i + 1}: "))
```

```
        marks = []
        for i in range(3):
            mark = float(input(f"Enter mark {i + 1}: "))
            marks.append(mark)

        student = Student(name, age, marks)
        student.display_details()

        print(f"Total Marks: {student.total_marks()}")
```

```
Enter student name: PARDHU
Enter student age: 19
Enter mark 1: 17
Enter mark 2: 18
Enter mark 3: 19

 Student Details:
Name: PARDHU
Age: 19
Total Marks: 54.0
```

OBSERVATION:

The legacy class used unclear variable names and lacked structure, making it difficult to extend or maintain.
The refactored version improves readability through meaningful naming, added docstrings, and a modular design.Storing marks in a list allows efficient computation using sum().
The inclusion of dynamic user input makes the program more flexible and interactive.
Overall, the code now follows better object-oriented and Pythonic practices.

TASK_4

Task Description #4 – Inefficient Loop Refactoring

Task: Refactor this inefficient loop with AI help

Python Code

```
nums = [1,2,3,4,5,6,7,8,9,10]
squares = []
for i in nums:
squares.append(i * i)
```

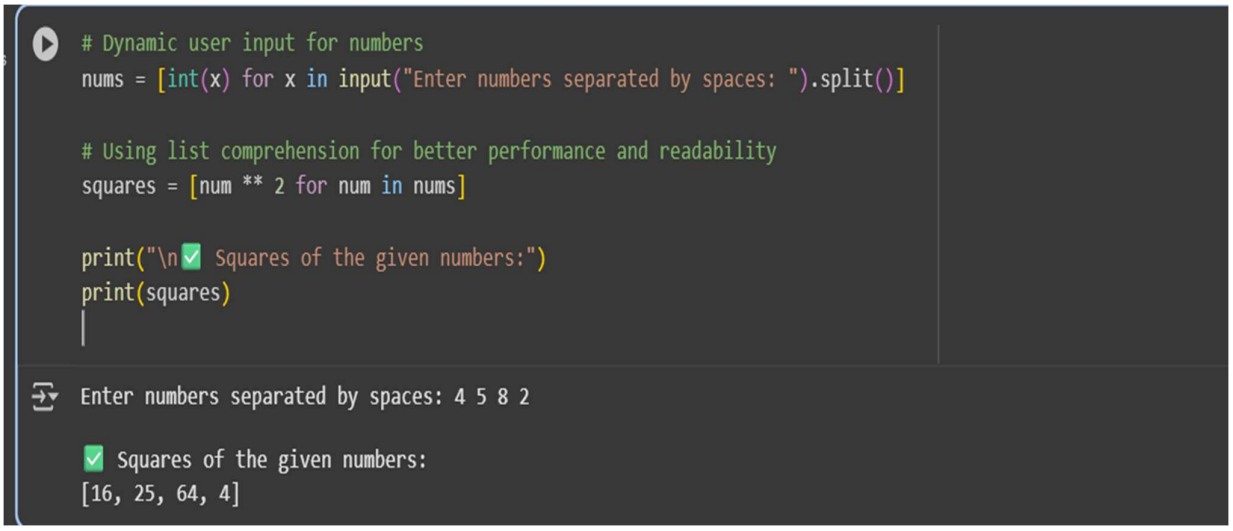
Expected Output: AI suggested a list comprehension

PROMPT:

Refactor the following Python code to make it **more efficient and concise**.
Replace the traditional for loop with a **list comprehension** for better readability and performance.

Ensure the functionality remains the same — generating a list of squared numbers from the given list.

CODE:



```
# Dynamic user input for numbers
nums = [int(x) for x in input("Enter numbers separated by spaces: ").split()]

# Using list comprehension for better performance and readability
squares = [num ** 2 for num in nums]

print("\n✅ Squares of the given numbers:")
print(squares)
```

```
Enter numbers separated by spaces: 4 5 8 2

✅ Squares of the given numbers:
[16, 25, 64, 4]
```

OBSERVATION:

The original code used a traditional for loop with append(), making it longer and slightly less efficient.The refactored version uses a **list comprehension**, which improves readability and execution speed.It provides a more **Pythonic and concise** way to generate the list of squares.Adding dynamic user input makes the program interactive and reusable for different inputs.Overall, the refactoring enhances both performance and clarity.