SCHOOLOFCOMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE			DEPARTMENTOFCOMPUTER SCIENCE ENGINEERING
ProgramName:B. Tech		**AssignmentType: Lab**	**AcademicYear:**2025-2026
CourseCoordinatorName		Venkataramana Veeramsetty	
Instructor(s)Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
CourseCode	24CS002PC215	**CourseTitle**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
DateandDay of Assignment	Week7 - WednesDay	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	
AssignmentNumber:13.3(Presentassignmentnumber)/**24**(Totalnumberofassignments)			

Q.No.	Question	*Expected Time to complete*
1	**Lab 13 – Code Refactoring: Improving Legacy Code with AI Suggestions** **Lab Objectives** • To introduce the concept of code refactoring and why it matters (readability, maintainability, performance). • To practice using AI tools for identifying and suggesting improvements in	Week5 - Monday

legacy code.

- To evaluate the before vs. after versions for clarity, performance, and correctness.
- To reinforce responsible AI-assisted coding practices (avoiding over-reliance, validating outputs).

Learning Outcomes

After completing this lab, students will be able to:

1. Use AI to analyze and refactor poorly written Python code.
2. Improve code **readability, efficiency, and error handling**.
3. Document AI-suggested improvements through comments and explanations.
4. Apply refactoring strategies without changing functionality.
5. Critically reflect on AI's refactoring suggestions.

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
```

Expected Output

- Refactored version with dictionary-based dispatch or separate functions.
- Cleaner and modular design.

PROMPT: the redundant 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 refactor the above code with output

CODE:

```
print(f"Area of rectangle (2, 3): {calculate_area('rectangle', 2, 3)}")
print(f"Area of square (4): {calculate_area('square', 4)}")
print(f"Area of circle (5): {calculate_area('circle', 5)}")

try:
    calculate_area("rectangle", 2)
except ValueError as e:
    print(f"Error for rectangle with only one dimension: {e}")

try:
    calculate_area("triangle", 2)
except ValueError as e:
    print(f"Error for invalid shape: {e}")
```

```
Area of rectangle (2, 3): 6
Area of square (4): 16
Area of circle (5): 78.53981633974483
Error for rectangle with only one dimension: For a rectangle, both x and y must be provided.
Error for invalid shape: Invalid shape provided. Choose from 'rectangle', 'square', or 'circle'.
```

OBSERVATION: The calculate_area function accurately computes areas for rectangles, squares, and circles. It effectively handles invalid shapes and missing dimensions for rectangles by raising ValueError. The output confirms correct area calculations for valid inputs and proper error message display for invalid ones. The refactoring improved code structure and error handling.

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: def read_file(filename): f = open(filename, "r") data = f.read() f.close() return data refactor this python code with open() and try-except. Give with output.

CODE:

```
# Create a dummy file for testing
with open("my_test_file.txt", "w") as f:
    f.write("This is a test file.")

# Test with an existing file
file_content = read_file("my_test_file.txt")
if file_content:
    print("\nContent of my_test_file.txt:")
    print(file_content)

# Test with a non-existing file
non_existent_content = read_file("non_existent_file.txt")
if non_existent_content is None:
    print("\nAs expected, read_file returned None for a non-existent file.")
```

```
Content of my_test_file.txt:
This is a test file.
Error: File 'non_existent_file.txt' not found.

As expected, read_file returned None for a non-existent file.
```

OBSERVATION: The read_file function successfully reads content from an existing file. It includes robust error handling, printing a user-friendly message and returning None if the file is not found. The output clearly demonstrates both successful file reading and proper error handling for missing files.

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: Generate this legacy class 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

CODE:

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

    def __init__(self, name, age, mark1, mark2, mark3):
        """
        Initializes a Student object.

        Args:
            name (str): The name of the student.
            age (int): The age of the student.
            mark1 (int): The mark in the first subject.
            mark2 (int): The mark in the second subject.
            mark3 (int): The mark in the third subject.
        """
        self.name = name
        self.age = age
        self.marks = [mark1, mark2, mark3]

    def details(self):
        """Prints the student's name and age."""
        print(f"Student Details:\n  Name: {self.name}\n  Age: {self.age}")

    def total(self):
        """Calculates and returns the total marks of the student."""
        return sum(self.marks)

# Example usage (will be added in a new cell):
# student1 = Student("Alice", 16, 85, 90, 78)
# student1.details()
# print(f"  Total marks: {student1.total()}")

# student2 = Student("Bob", 17, 92, 88, 95)
# student2.details()
# print(f"  Total marks: {student2.total()}")
```

OBSERVATION: This code defines a Student class to manage student details and marks, including methods to display details and calculate the total marks.

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: nums = [1,2,3,4,5,6,7,8,9,10] squares = [] for i in nums: squares.append(i * i) refactor this inefficient loop with list comprehension and give output. give total code.

Gemini

```
# Original loop
nums = [1,2,3,4,5,6,7,8,9,10]
squares_loop = []
for i in nums:
    squares_loop.append(i * i)
print("Output from original loop:", squares_loop)

# Refactored with list comprehension
squares_comprehension = [i * i for i in nums]
print("Output from list comprehension:", squares_comprehension)
```

```
Output from original loop: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Output from list comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

OBSERVATION: The list comprehension [i * i for i in nums] effectively replaces the explicit loop for creating the squares list. Both the original loop and the list comprehension produce the same output, demonstrating that list comprehension is a concise and efficient way to achieve the same result as the loop in this case.