SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
ProgramName: B. Tech		**Assignment Type: 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
Date and Day of Assignment	Week4 - Wednesday	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	
AssignmentNumber: **9.3**(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*ExpectedTime to complete*
1	Lab 8: Documentation Generation: Automatic documentation and code comments **Lab Objectives:** • To understand the importance of documentation and code comments in software development. • To explore how AI-assisted coding tools can generate meaningful documentation and	Week4 - Wednesday

inline comments.

- To practice generating function-level and module-level docstrings automatically.
- To evaluate the quality, accuracy, and limitations of AI-generated documentation.
- To develop a small automated tool for documentation generation in Python..

Lab Outcomes (LOs):
After completing this lab, students will be able to:

- Apply AI-assisted coding tools to generate docstrings and inline comments for Python code.
- Critically analyze AI-generated documentation for correctness, completeness, and readability.
- Create structured documentation (function-level, module-level) following standard formats.

- Design and implement a mini documentation generator tool to automate code commenting and docstring creation.

Task Description#1 Basic Docstring Generation

- Write python function to return sum of even and odd numbers in the given list.
- Incorporate manual **docstring** in code with Google Style
- Use an AI-assisted tool (e.g., Copilot, Cursor AI) to generate a docstring describing the function.
- Compare the AI-generated docstring with your manually written one.
- **Prompt1:** Write a Python function to return the sum of even and odd numbers in a given list.

```
def sum_even_odd(numbers):
  """Calculates the sum of even and odd numbers in a list.

  Args:
    numbers: A list of integers or floats.

  Returns:
    A tuple containing the sum of even numbers and the sum of odd numbers
    (even_sum, odd_sum).
  """
  even_sum = 0
  odd_sum = 0
  for number in numbers:
    if number % 2 == 0:
      even_sum += number
    else:
      odd_sum += number
  return even_sum, odd_sum

even_sum, odd_sum = sum_even_odd_from_input()
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Enter comma-separated numbers: 1,2,4,5,69,7,8,6
Sum of even numbers: 20.0
Sum of odd numbers: 82.0
```

Expected Outcome#1: Students understand how AI can produce function-level documentation.

Prompt2: Incorporate a manual docstring in code .

```python
def sum_even_odd(numbers):
  """Calculates the sum of even and odd numbers in a list.

  Args:
    numbers: A list of integers or floats.

  Returns:
    A tuple containing the sum of even numbers and the sum of odd numbers
    (even_sum, odd_sum).
  """
  even_sum = 0
  odd_sum = 0
  for number in numbers:
    if number % 2 == 0:
      even_sum += number
    else:
      odd_sum += number
  return even_sum, odd_sum
```

```python
even_sum, odd_sum = sum_even_odd_from_input()
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Enter comma-separated numbers: 76,85,3,53,4,8,9,3,1
Sum of even numbers: 88.0
Sum of odd numbers: 154.0
```

Prompt3: generate and implement a docstring describing the function.

```python
def sum_even_odd_from_input():
  """Takes comma-separated numbers as input from the user, calculates the sum
  of even and odd numbers in the list, and returns the sums.

  Returns:
    A tuple containing the sum of even numbers and the sum of odd numbers
    (even_sum, odd_sum).
  """
  # Function implementation goes here
  pass
```

```python
def sum_even_odd_from_input():
  """Takes comma-separated numbers as input from the user, calculates the sum
  of even and odd numbers in the list, and returns the sums.

  Returns:
    A tuple containing the sum of even numbers and the sum of odd numbers
    (even_sum, odd_sum).
  """
  input_str = input("Enter comma-separated numbers: ")
  numbers_str_list = input_str.split(',')
  numbers = []
  for num_str in numbers_str_list:
    try:
      numbers.append(float(num_str.strip()))
    except ValueError:
      print(f"Warning: Could not convert '{num_str.strip()}' to a number. Skipping.")

  even_sum = 0
  odd_sum = 0
  for number in numbers:
    if number % 2 == 0:
      even_sum += number
    else:
      odd_sum += number
  return even_sum, odd_sum
```

Comparison:Both docstrings provide a concise summary of the function's purpose, describe the arguments, and explain the return value.
The AI-generated docstring is a good starting point but could be more specific about the type of numbers expected in the list (e.g., integers or floats). It also doesn't specify the names of the elements in the returned tuple.
The manually written docstring in Google Style is more detailed and adheres to the specified format. It explicitly mentions the expected data types for the input list and clarifies the contents of the returned tuple. This makes it easier for other developers to understand how to use the function correctly.
In summary, while AI can generate useful docstrings, manually reviewing and refining them to follow a specific style guide like Google Style often results in more comprehensive and user-friendly documentation.

Task Description#2 Automatic Inline Comments

- Write python program for **sru_student** class with attributes like name, roll no., hostel_status and **fee_update** method and **display_details** method.
- Write comments manually for each line/code block
- Ask an AI tool to add inline comments explaining each line/step.
- Compare the AI-generated comments with your manually written one.

Expected Output#2: Students critically analyze AI-generated code comments.
Prompt1: Write a Python program for an sru_student class with attributes like name, roll number, hostel_status, and a fee_update method, as well as a display_details method.

```python
class sru_student:
    """Represents a student at SRU with basic details and fee information."""

    def __init__(self, name, roll_number, hostel_status):
        """Initializes a new sru_student instance.

        Args:
            name: The name of the student.
            roll_number: The roll number of the student.
            hostel_status: The hostel status of the student (e.g., 'resident', 'day scholar').
        """
        self.name = name
        self.roll_number = roll_number
        self.hostel_status = hostel_status
        self.fees_paid = 0  # Initialize fees paid to 0

    def fee_update(self, amount):
        """Updates the fees paid by the student.

        Args:
            amount: The amount of fees paid.
        """
        if amount > 0:
            self.fees_paid += amount
            print(f"Fees updated for {self.name}. Total fees paid: {self.fees_paid}")
        else:
            print("Invalid amount. Fees update failed.")

    def display_details(self):
        """Displays the details of the student."""
        print("--- Student Details ---")
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_number}")
        print(f"Hostel Status: {self.hostel_status}")
        print(f"Fees Paid: {self.fees_paid}")
        print("-----------------------")
```

```
--- Student Details ---
Name: Alice
Roll Number: SRU123
Hostel Status: resident
Fees Paid: 0
-----------------------
Fees updated for Alice. Total fees paid: 5000
Fees updated for Alice. Total fees paid: 7000
--- Student Details ---
Name: Alice
Roll Number: SRU123
Hostel Status: resident
Fees Paid: 7000
-----------------------
--- Student Details ---
Name: Bob
Roll Number: SRU456
Hostel Status: day scholar
Fees Paid: 0
-----------------------
```

Prompt2: Write comments manually for each line or code block.

```python
class sru_student:
    """Represents a student at SRU with basic details and fee information.""" # Docstring for the class

    def __init__(self, name, roll_number, hostel_status):
        """Initializes a new sru_student instance.

        Args:
            name: The name of the student.
            roll_number: The roll number of the student.
            hostel_status: The hostel status of the student (e.g., 'resident', 'day scholar').
        """ # Docstring for the constructor
        self.name = name  # Store the student's name
        self.roll_number = roll_number  # Store the student's roll number
        self.hostel_status = hostel_status  # Store the student's hostel status
        self.fees_paid = 0  # Initialize fees paid to 0 for a new student

    def fee_update(self, amount):
        """Updates the fees paid by the student.

        Args:
            amount: The amount of fees paid.
        """ # Docstring for the fee_update method
        if amount > 0:  # Check if the amount is positive
            self.fees_paid += amount  # Add the positive amount to fees paid
            print(f"Fees updated for {self.name}. Total fees paid: {self.fees_paid}") # Print update message
        else:
            print("Invalid amount. Fees update failed.") # Print error message for invalid amount

    def display_details(self):
        """Displays the details of the student.""" # Docstring for the display_details method
        print("--- Student Details ---") # Print header
        print(f"Name: {self.name}") # Print student's name
        print(f"Roll Number: {self.roll_number}") # Print student's roll number
        print(f"Hostel Status: {self.hostel_status}") # Print student's hostel status
        print(f"Fees Paid: {self.fees_paid}") # Print total fees paid
        print("-----------------------") # Print footer
```

```python
def calculate_total_fees(students_list):
  """Calculates the total fees paid by a list of sru_student objects.

  Args:
    students_list: A list of sru_student objects.

  Returns:
    The total fees paid by all students in the list.
  """
  total_fees = 0
  for student in students_list:
    total_fees += student.fees_paid
  return total_fees
```

```python
# Example usage of the calculate_total_fees function
# Assuming student1 and student2 are already created from previous cells
all_students = [student1, student2]
total_fees_paid = calculate_total_fees(all_students)
print(f"\nTotal fees paid by all students: {total_fees_paid}")
```

```
Total fees paid by all students: 7000
```

Prompt3: add inline comments explaining each line/step.

```python
def calculate_total_fees(students_list):
  """Calculates the total fees paid by a list of sru_student objects.

  Args:
    students_list: A list of sru_student objects.

  Returns:
    The total fees paid by all students in the list.
  """
  total_fees = 0 # Initialize total_fees to 0
  for student in students_list: # Iterate through each student in the list
    total_fees += student.fees_paid # Add the current student's fees_paid to total_fees
  return total_fees # Return the final total fees
```

```
# Get student details from user input
name_input = input("Enter student's name: ")
roll_number_input = input("Enter student's roll number: ")
hostel_status_input = input("Enter student's hostel status (e.g., resident, day scholar): ")

# Create a new sru_student instance using user input
new_student = sru_student(name_input, roll_number_input, hostel_status_input)

# Display the details of the new student
new_student.display_details()
```

```
Enter student's name: priya
Enter student's roll number: 234
Enter student's hostel status (e.g., resident, day scholar): day scholar
--- Student Details ---
Name: priya
Roll Number: 234
Hostel Status: day scholar
Fees Paid: 0
-----------------------
```

Comparison:

AI-Generated Comments:

- Often focus on *what* the code does at a line-by-line level.
- Can be generated quickly for basic code explanations.
- May not always provide the broader context or purpose of a code block or function.
- Might be repetitive for straightforward lines of code.

Manually Written Comments:

- Can explain the *why* behind the code, including design choices and logic.
- Allow for more detailed explanations of complex sections.
- Can be structured to align with specific documentation standards (like Google Style docstrings).
- Provide flexibility to explain code at different levels of abstraction (line, block, function).
- Can include important notes or assumptions relevant to the code.

In summary, while AI can quickly generate basic comments, manually written comments offer more depth, context, and adherence to documentation best practices.

Task Description#3

- Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply, divide).
- Incorporate manual **docstring** in code with NumPy Style
- Use AI assistance to generate a module-level docstring + individual function docstrings.
- Compare the AI-generated docstring with your manually written one.

Expected Output#3: Students learn structured documentation for multi-function scripts

Prompt1: Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply, divide).

```python
def add(x, y):
  """Adds two numbers."""
  return x + y

def subtract(x, y):
  """Subtracts the second number from the first."""
  return x - y

def multiply(x, y):
  """Multiplies two numbers."""
  return x * y

def divide(x, y):
  """Divides the first number by the second. Handles division by zero."""
  if y == 0:
    return "Error: Division by zero!"
  return x / y
```

```python
# Get input from the user
try:
    num1_input = float(input("Enter the first number: "))
    num2_input = float(input("Enter the second number: "))
    operation = input("Enter the operation (+, -, *, /): ")

    # Perform the calculation based on the operation
    if operation == '+':
        result = add(num1_input, num2_input)
    elif operation == '-':
        result = subtract(num1_input, num2_input)
    elif operation == '*':
        result = multiply(num1_input, num2_input)
    elif operation == '/':
        result = divide(num1_input, num2_input)
    else:
        result = "Error: Invalid operation"

    # Display the result
    print(f"Result: {result}")

except ValueError:
    print("Invalid input. Please enter valid numbers.")
```

```
Enter the first number: 4
Enter the second number: 7
Enter the operation (+, -, *, /): *
Result: 28.0
```

Prompt2: Incorporate a manual docstring in the code using NumPy Style.

```python
def add(x, y):
    """
    Adds two numbers.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
    float or int
        The sum of the two numbers.
    """
    return x + y

def subtract(x, y):
    """
    Subtracts the second number from the first.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
    float or int
        The difference between the two numbers.
    """
    return x - y
```

```
def multiply(x, y):
    """
    Multiplies two numbers.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
    float or int
        The product of the two numbers.
    """
    return x * y

def divide(x, y):
    """
    Divides the first number by the second.

    Parameters
    ----------
    x : float or int
        The first number (dividend).
    y : float or int
        The second number (divisor).

    Returns
    -------
    float or str
        The result of the division, or an error message if division by zero occurs.


        Raises
        ------
        ValueError
            If the second number (y) is zero. This is handled internally and returns a string.
        """
        if y == 0:
            return "Error: Division by zero!"
        return x / y
```

Prompt3: generate a module-level docstring and individual function docstrings.

```
"""
This module provides basic arithmetic operations.

Functions
---------
add(x, y)
    Adds two numbers.
subtract(x, y)
    Subtracts the second number from the first.
multiply(x, y)
    Multiplies two numbers.
divide(x, y)
    Divides the first number by the second, handling division by zero.
power(x, y)
    Raises the first number to the power of the second.
modulo(x, y)
    Returns the remainder of the division of the first number by the second.
integer_divide(x, y)
    Performs integer division of the first number by the second.
"""

def add(x, y):
    """
    Adds two numbers.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
```

```
    float or int
        The sum of the two numbers.
    """
    return x + y

def subtract(x, y):
    """
    Subtracts the second number from the first.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
    float or int
        The difference between the two numbers.
    """
    return x - y

def multiply(x, y):
    """
    Multiplies two numbers.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
```

```
    float or int
        The difference between the two numbers.
    """
    return x - y

def multiply(x, y):
    """
    Multiplies two numbers.

    Parameters
    ----------
    x : float or int
        The first number.
    y : float or int
        The second number.

    Returns
    -------
    float or int
        The product of the two numbers.
    """
    return x * y

def divide(x, y):
    """
    Divides the first number by the second.

    Parameters
    ----------
    x : float or int
        The first number (dividend).
    y : float or int
        The second number (divisor).

    Returns
    -------
```

```python
# Get input from the user
try:
    num1_input = float(input("Enter the first number: "))
    num2_input = float(input("Enter the second number: "))
    operation = input("Enter the operation (+, -, *, /, **, %, //): ")

    # Perform the calculation based on the operation
    if operation == '+':
        result = add(num1_input, num2_input)
    elif operation == '-':
        result = subtract(num1_input, num2_input)
    elif operation == '*':
        result = multiply(num1_input, num2_input)
    elif operation == '/':
        result = divide(num1_input, num2_input)
    elif operation == '**':
        result = power(num1_input, num2_input)
    elif operation == '%':
        result = modulo(num1_input, num2_input)
    elif operation == '//':
        result = integer_divide(num1_input, num2_input)
    else:
        result = "Error: Invalid operation"

    # Display the result
    print(f"Result: {result}")

except ValueError:
    print("Invalid input. Please enter valid numbers.")
```

```
Enter the first number: 2
Enter the second number: 7
Enter the operation (+, -, *, /, **, %, //): +
Result: 9.0
```

AI-Generated Docstrings/Comments:

- Often focus on *what* the code does at a line-by-line level or a basic function summary.
- Can be generated quickly for initial explanations.
- May not always provide the broader context, design choices, or purpose of a code block/function.
- Might be less structured or not adhere to specific style guides (like NumPy or Google style) as closely without explicit instruction.

Manually Written Docstrings/Comments:

- Can explain the *why* behind the code, including design decisions and complex logic.
- Allow for more detailed and nuanced explanations.
- Can be structured to align with specific documentation standards (like NumPy or Google Style docstrings).
- Provide flexibility to explain code at different levels of abstraction (line, block, function, module).
- Can include important notes, assumptions, or examples relevant to the code's usage.

Push documentation whole workspace as .md file in GitHub Repository

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots