# Lab Assisgnment

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : Thota Akhileshwari

Enrollment No : 2403a52010

Batch No. : 02

Date : 24-09-2025

**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.

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

**Prompt:** Write a Python function that takes a list of integers and returns a tuple containing the sum of even numbers and the sum of odd numbers. First, include a manually written docstring using Google Style. Then, generate a second version of the function with an AI-generated docstring. Finally, compare the two docstrings in terms of clarity, formatting, and completeness.

### AI generated docstring:

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

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing two integers: the sum of even numbers and the sum of odd numbers.
  """
  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)

# Example usage:
my_list = [1, 2, 3, 4, 5, 6]
even_sum, odd_sum = sum_even_odd(my_list)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Sum of even numbers: 12
Sum of odd numbers: 9
```

### Manullay created doctstring:

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

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing two integers: the sum of even numbers and the sum of odd numbers.
  """
  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)

# Example usage:
my_list = [1, 2, 3, 4, 5, 6]
even_sum, odd_sum = sum_even_odd(my_list)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Sum of even numbers: 12
Sum of odd numbers: 9
```

**Comparison:** Both of the code cells you provided contain the same Python function sum_even_odd (or sum_even_odd_ai_docstring). The only difference between the two cells is the docstring.

One docstring was written manually following the Google Style Guide, while the other was generated by an AI. The core logic and functionality of the Python function itself are identical in both cells. The difference lies solely in the documentation provided within the docstring.

### Explanation:

Imagine you have a list of numbers, like [1, 2, 3, 4, 5, 6].

The code's job is to go through this list and pick out the even numbers (numbers that can be divided by 2 without a remainder, like 2, 4, and 6) and add them up.

It also picks out the odd numbers (numbers that have a remainder when divided by 2, like 1, 3, and 5) and adds those up.

Finally, it gives you back two numbers: the total sum of all the even numbers and the total sum of all the odd numbers from the list.

The two versions of the code you have do the exact same thing. The only real difference is how they write down the instructions or notes about what the code does (the "docstring").

So, in short, the code takes a list of numbers and figures out the total of the even ones and the total of the odd ones.

### Observation:

The code adds up even numbers and odd numbers separately from a list.

Both versions of the code do the exact same job and give the same answers for the example list.

The main difference is how the notes (called "docstrings") inside the code are written. One set of notes was written by a person, and the other was made by a computer.

The code is easy to read and understand.

**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

**Prompt:** Create a Python class named sru_student that includes attributes such as name, roll_no, and hostel_status. Implement two methods: fee_update() to modify fee-related information, and display_details() to show student details.

First, write the code with your own manual comments explaining each line or block. Then, use an AI tool to automatically generate inline comments for the same code.

Finally, compare your manual comments with the AI-generated ones and reflect on the differences in clarity.

### *Manually created docstring:*

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

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing two integers: the sum of even numbers and the sum of odd numbers.
  """
  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)

# Example usage:
my_list = [1, 2, 3, 4, 5, 6]
even_sum, odd_sum = sum_even_odd(my_list)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
Sum of even numbers: 12
Sum of odd numbers: 9
```

### *AI-generated docstring:*

```python
class sru_student:
    # Constructor to initialize the student object
    def __init__(self, name, roll_no, hostel_status):
        # Assign the name attribute
        self.name = name
        # Assign the roll_no attribute
        self.roll_no = roll_no
        # Assign the hostel_status attribute
        self.hostel_status = hostel_status
        # Initialize the fee_update attribute to False
        self.fee_update = False

    # Method to update the fee status
    def fee_update_status(self):
        # Set fee_update to True
        self.fee_update = True
        # Print a confirmation message
        print(f"Fee status updated for {self.name}")

    # Method to display student details
    def display_details(self):
        # Print the student's name
        print(f"Name: {self.name}")
        # Print the student's roll number
        print(f"Roll No.: {self.roll_no}")
        # Print the student's hostel status
        print(f"Hostel Status: {self.hostel_status}")
```

```
        # Print the student's roll number
        print(f"Roll No.: {self.roll_no}")
        # Print the student's hostel status
        print(f"Hostel Status: {self.hostel_status}")
        # Print the student's fee update status
        print(f"Fee Updated: {self.fee_update}")

# Example usage:
# Create a new student object
student1 = sru_student("Alice", "SRU123", "In-campus")
# Display the student's details
student1.display_details()
# Update the student's fee status
student1.fee_update_status()
# Display the student's details again to see the updated fee status
student1.display_details()
```

```
Name: Alice
Roll No.: SRU123
Hostel Status: In-campus
Fee Updated: False
Fee status updated for Alice
Name: Alice
Roll No.: SRU123
Hostel Status: In-campus
Fee Updated: True
```

### Observation:

- Accuracy and Clarity: The comments accurately and clearly describe what each line or block of code does.
- Level of Detail: The comments explain the purpose and logic behind the code, not just a literal translation.
- Conciseness: The comments are brief and to the point, avoiding unnecessary verbosity.
- Usefulness: The comments provide valuable insights for someone reading the code and are not merely restating the obvious.
- Consistency: The comments maintain a consistent style and level of detail throughout the code.

### 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

**Prompt:** Develop a Python module that includes 3–4 basic arithmetic functions such as add, subtract, multiply, and divide.

Write manual docstrings for each function using NumPy-style documentation to practice structured commenting.

Then, use an AI tool to automatically generate both a module-level docstring and function-level docstrings.

Finally, compare your manually written docstrings with the AI-generated ones to evaluate differences in clarity.

### *AI generated docstring:*

```python
# calculator_ai.py

"""
A simple calculator module providing basic arithmetic operations.
"""

def add(a, b):
    """Adds two numbers.

    Args:
        a (float or int): The first number.
        b (float or int): The second number.

    Returns:
        float or int: The sum of the two numbers.
    """
    return a + b

def subtract(a, b):
    """Subtracts the second number from the first.

    Args:
        a (float or int): The first number.
        b (float or int): The number to subtract.
```

```python
def multiply(a, b):
    """Multiplies two numbers.

    Args:
        a (float or int): The first number.
        b (float or int): The second number.

    Returns:
        float or int: The product of the two numbers.
    """
    return a * b

def divide(a, b):
    """Divides the first number by the second.

    Args:
        a (float or int): The numerator.
        b (float or int): The denominator.

    Returns:
        float or int: The result of the division.

    Raises:
```

---

```
Args:
    a (float or int): The numerator.
    b (float or int): The denominator.

Returns:
    float or int: The result of the division.

Raises:
    ZeroDivisionError: If the denominator is zero.
"""
if b == 0:
    raise ZeroDivisionError("division by zero")
return a / b
```

***Manullay created docstring:***

```python
# This is a simple example of code with inline comments

# Define a variable and assign a value
x = 10

# Define another variable
y = 5

# Add the two variables
sum_result = x + y # Calculate the sum

# Print the result
print(f"The sum of x and y is: {sum_result}") # Display the sum
```

```
The sum of x and y is: 15
```

## Observation:

- Granularity: In cell there's a comment for almost every single line or a very small block of code (e.g., `# Define a variable and assign a value`, `# Calculate the sum`). In cell , the comments are generally for larger blocks, like the purpose of a method (`# Method to update the student's fee`) or the initialization block (`# Initialize the sru_student class...`).
- Purpose: Both sets of comments aim to explain what the code does. However, the comments in are very literal and describe the immediate action of the line, while the comments in focus more on the overall purpose of a function or section.
- Placement: Both use inline comments (`# comment`) placed on the same line as the code or on the line above the code block they describe.
- Detail: The comments in are quite detailed for a very simple script. The comments in are also detailed but are more focused on the functional aspect of the class and its methods.
- Readability: For very simple code like in excessive inline comments on every line can sometimes make the code look cluttered. For more complex structures like the class .