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)

	Question	***ExpectedTime to complete***
1	Lab 8: Documentation Generation: Automatic documentation and code comments **Lab Objectives:**	Week4 - Wednesday

- To understand the importance of documentation and code comments in software development.
- To explore how AI-assisted coding tools can generate meaningful documentation and 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.

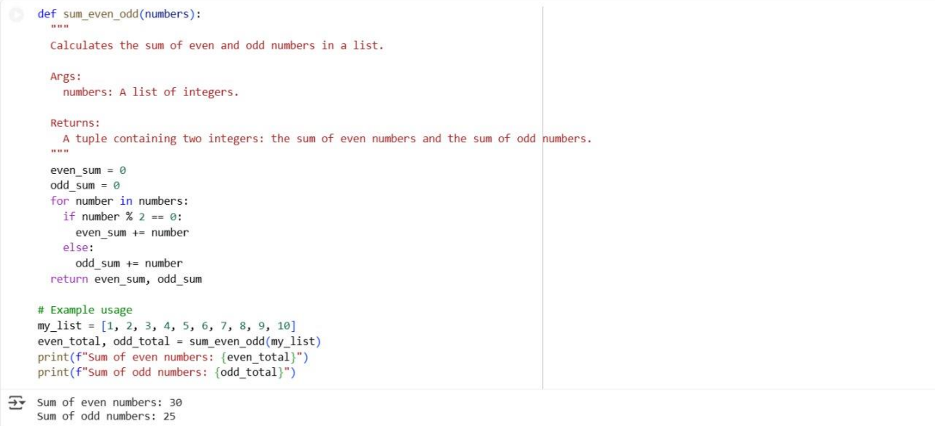
Task Description#1 Basic Docstring Generation

- 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.

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

#PROMPT:
Write python function to return sum of even and odd numbers in the given list which are entered by user.

#MANUALLY WRITTEN DOCSTRING AI CODE



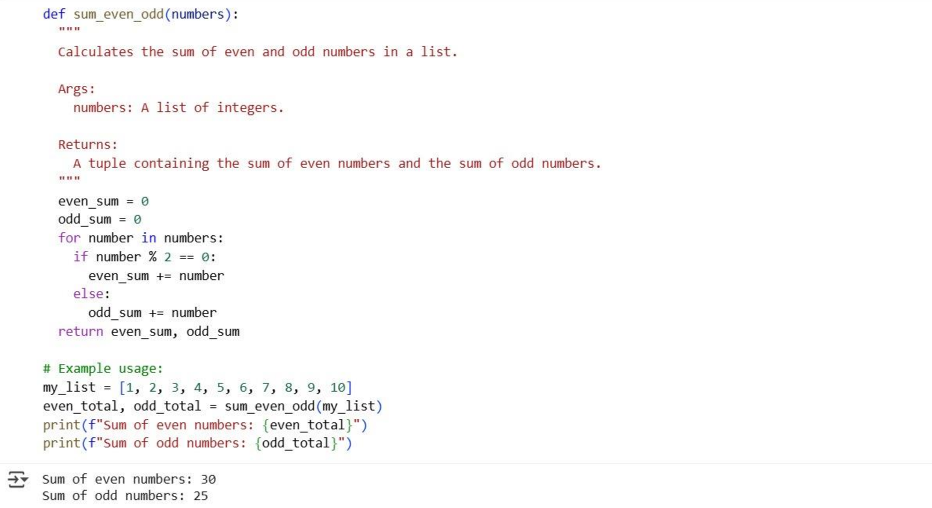
```python
def sum_even_odd(numbers):
  """
  Calculates the sum of even and odd numbers in a list.

  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, 7, 8, 9, 10]
even_total, odd_total = sum_even_odd(my_list)
print(f"Sum of even numbers: {even_total}")
print(f"Sum of odd numbers: {odd_total}")
```

```
Sum of even numbers: 30
Sum of odd numbers: 25
```

# AI CODE

#PROMPT:

Write python function to return sum of even and odd numbers in the given list



```
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 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, 7, 8, 9, 10]
even_total, odd_total = sum_even_odd(my_list)
print(f"Sum of even numbers: {even_total}")
print(f"Sum of odd numbers: {odd_total}")
```

```
Sum of even numbers: 30
Sum of odd numbers: 25
```

#COMPARISON:

- Clarity and Detail: The AI-generated docstring is more detailed and structured, clearly defining the arguments (Args) and return values (Returns) using standard docstring conventions. Your docstring is more conversational and describes the process.
- Conciseness: Your docstring is more concise.
- Standard Conventions: The AI-generated docstring follows a common standard (like Google style) which is beneficial for code readability and maintainability, especially in larger projects. Your docstring is less formal.
- Accuracy: Both accurately describe the function's purpose. There's a small typo in your docstring ("oddd" instead of "odd").

Both docstrings serve the purpose of explaining the function. The AI-generated one is more aligned with standard documentation practices, which can be helpful for collaboration and clarity. Your docstring is more informal but still conveys the function's intent.

Task Description#2 Automatic Inline Comments

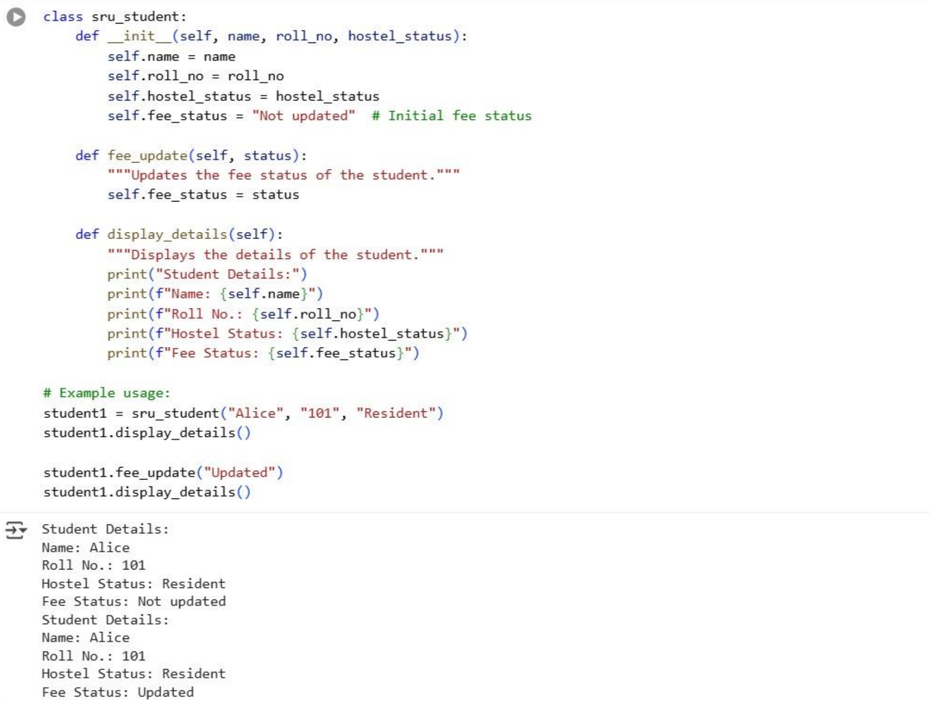
- Write python program for **sru_student** class with attributes like name, roll no., hostel_status and **fee_update** method and **display_details** method.
- Write comments manually for each line/code block
- Ask an AI tool to add inline comments explaining each line/step.
- Compare the AI-generated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

Expected Output#2: Students critically analyze AI-generated code comments.

#PROMPT:

- Write python program for **sru_student** class with attributes like name, roll no., hostel_status and **fee_update** method and **display_details** method.

# AI CODE



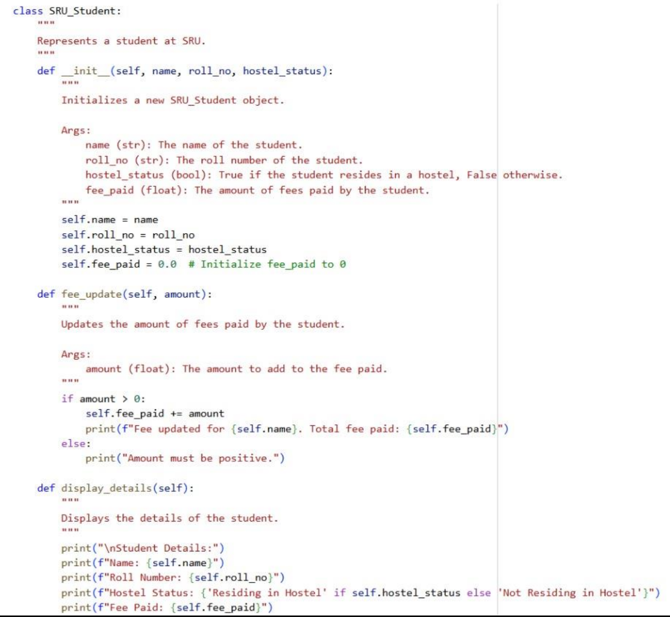
```
class sru_student:
    def __init__(self, name, roll_no, hostel_status):
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fee_status = "Not updated"  # Initial fee status

    def fee_update(self, status):
        """Updates the fee status of the student."""
        self.fee_status = status

    def display_details(self):
        """Displays the details of the student."""
        print("Student Details:")
        print(f"Name: {self.name}")
        print(f"Roll No.: {self.roll_no}")
        print(f"Hostel Status: {self.hostel_status}")
        print(f"Fee Status: {self.fee_status}")

# Example usage:
student1 = sru_student("Alice", "101", "Resident")
student1.display_details()

student1.fee_update("Updated")
student1.display_details()
```

```
Student Details:
Name: Alice
Roll No.: 101
Hostel Status: Resident
Fee Status: Not updated
Student Details:
Name: Alice
Roll No.: 101
Hostel Status: Resident
Fee Status: Updated
```

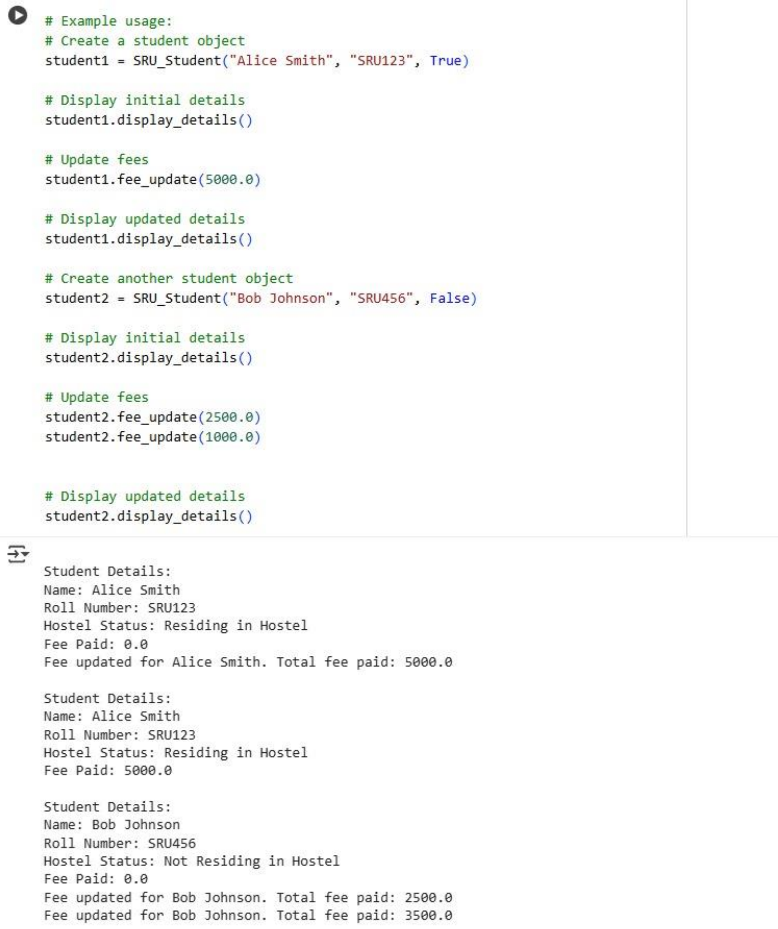
#MANUALLY AI CODE

#PROMPT:

- Write python program for **sru_student** class with attributes like name, roll no., hostel_status and **fee_update** method and **display_details** method.student details will be entered by the user.



```
class SRU_Student:
    """
    Represents a student at SRU.
    """
    def __init__(self, name, roll_no, hostel_status):
        """
        Initializes a new SRU_Student object.

        Args:
            name (str): The name of the student.
            roll_no (str): The roll number of the student.
            hostel_status (bool): True if the student resides in a hostel, False otherwise.
            fee_paid (float): The amount of fees paid by the student.
        """
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fee_paid = 0.0  # Initialize fee_paid to 0

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

        Args:
            amount (float): The amount to add to the fee paid.
        """
        if amount > 0:
            self.fee_paid += amount
            print(f"Fee updated for {self.name}. Total fee paid: {self.fee_paid}")
        else:
            print("Amount must be positive.")

    def display_details(self):
        """
        Displays the details of the student.
        """
        print("\nStudent Details:")
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.roll_no}")
        print(f"Hostel Status: {'Residing in Hostel' if self.hostel_status else 'Not Residing in Hostel'}")
        print(f"Fee Paid: {self.fee_paid}")
```



```
# Example usage:
# Create a student object
student1 = SRU_Student("Alice Smith", "SRU123", True)

# Display initial details
student1.display_details()

# Update fees
student1.fee_update(5000.0)

# Display updated details
student1.display_details()

# Create another student object
student2 = SRU_Student("Bob Johnson", "SRU456", False)

# Display initial details
student2.display_details()

# Update fees
student2.fee_update(2500.0)
student2.fee_update(1000.0)


# Display updated details
student2.display_details()
```

```
Student Details:
Name: Alice Smith
Roll Number: SRU123
Hostel Status: Residing in Hostel
Fee Paid: 0.0
Fee updated for Alice Smith. Total fee paid: 5000.0

Student Details:
Name: Alice Smith
Roll Number: SRU123
Hostel Status: Residing in Hostel
Fee Paid: 5000.0

Student Details:
Name: Bob Johnson
Roll Number: SRU456
Hostel Status: Not Residing in Hostel
Fee Paid: 0.0
Fee updated for Bob Johnson. Total fee paid: 2500.0
Fee updated for Bob Johnson. Total fee paid: 3500.0
```

#COMPARISION:

Here's a comparison: Original AI-Generated Comments: • Provided docstrings for the class and each method (*init*, update_fee, display_details) explaining their purpose, arguments, and what they do. • Included inline comments for key actions like initializing fee_paid and example usage of the class. Your Modified Comments: • Kept the docstrings for the class and methods largely the same. • Replaced the inline comments for the example usage with more descriptive comments like # student 1 details input, # Display of student1 details, # fee updation of student1, etc. Both sets of comments are helpful in understanding the code. The original comments focused more on the internal workings of the class and methods, while your modified comments provide a clear step-by-step explanation of the example usage at the end of the script. Your changes make the execution flow of the example more explicit.

Task Description#3

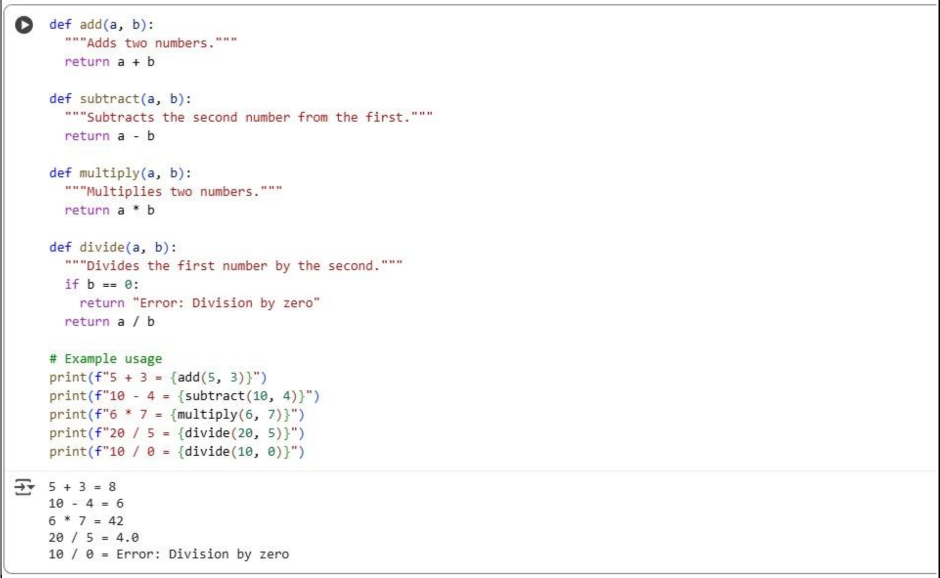
- Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply, divide).
- Incorporate manual **docstring** in code with NumPy Style
- Use AI assistance to generate a module-level docstring + individual function docstrings.

- Compare the AI-generated docstring with your manually written one.

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

#AI CODE:
#PROMPT:

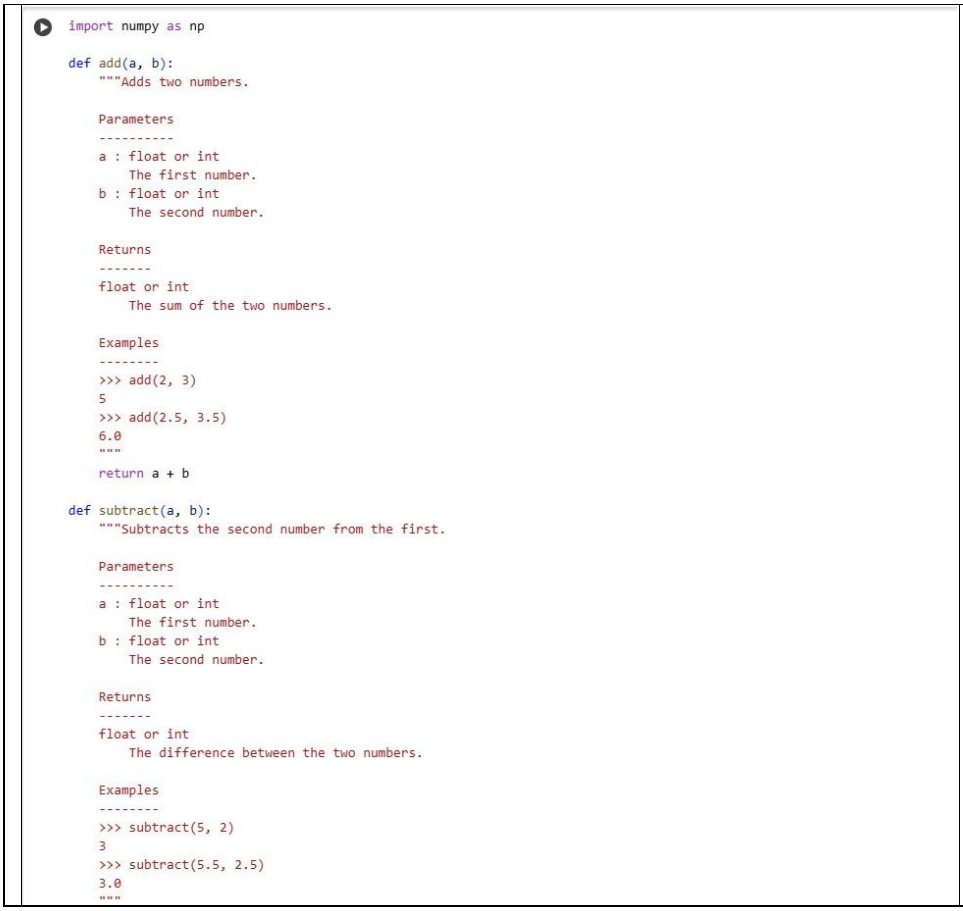
- Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply, divide).



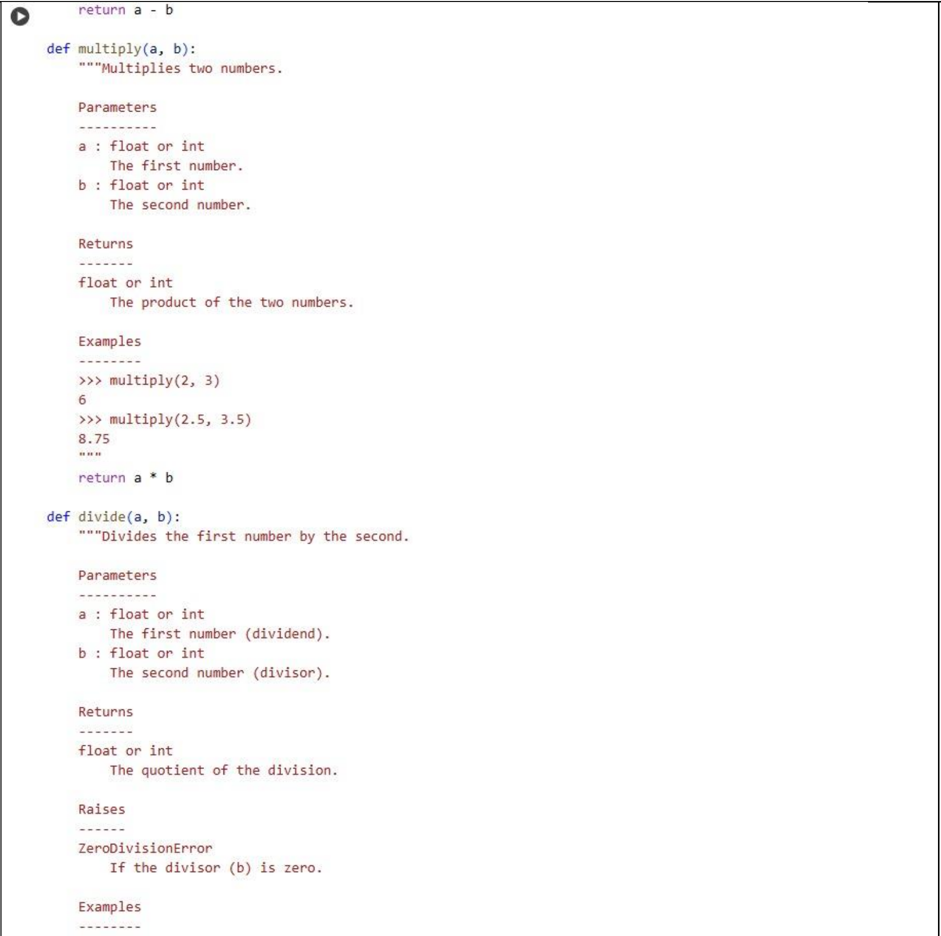
```
def add(a, b):
  """Adds two numbers."""
  return a + b

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

def multiply(a, b):
  """Multiplies two numbers."""
  return a * b

def divide(a, b):
  """Divides the first number by the second."""
  if b == 0:
    return "Error: Division by zero"
  return a / b

# Example usage
print(f"5 + 3 = {add(5, 3)}")
print(f"10 - 4 = {subtract(10, 4)}")
print(f"6 * 7 = {multiply(6, 7)}")
print(f"20 / 5 = {divide(20, 5)}")
print(f"10 / 0 = {divide(10, 0)}")
```

```
5 + 3 = 8
10 - 4 = 6
6 * 7 = 42
20 / 5 = 4.0
10 / 0 = Error: Division by zero
```

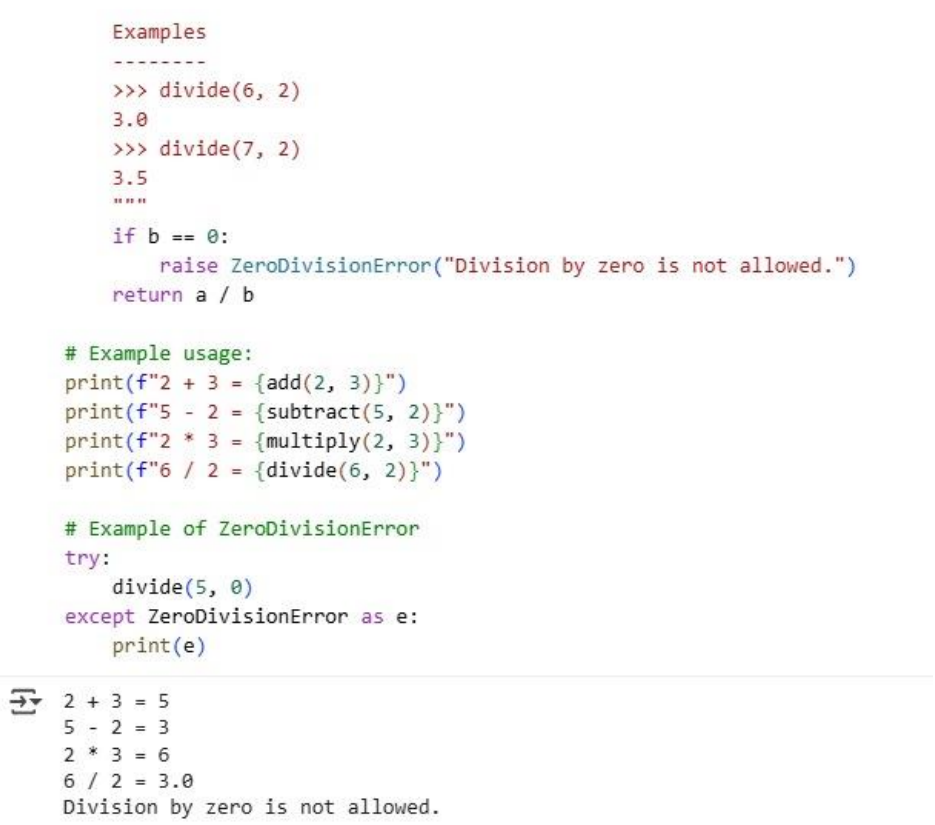
#MANUALLY AI CODE:



```
import numpy as np

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

    Parameters
    ----------
    a : float or int
        The first number.
    b : float or int
        The second number.

    Returns
    -------
    float or int
        The sum of the two numbers.

    Examples
    --------
    >>> add(2, 3)
    5
    >>> add(2.5, 3.5)
    6.0
    """
    return a + b

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

    Parameters
    ----------
    a : float or int
        The first number.
    b : float or int
        The second number.

    Returns
    -------
    float or int
        The difference between the two numbers.

    Examples
    --------
    >>> subtract(5, 2)
    3
    >>> subtract(5.5, 2.5)
    3.0
    """
```



```
    return a - b

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

    Parameters
    ----------
    a : float or int
        The first number.
    b : float or int
        The second number.

    Returns
    -------
    float or int
        The product of the two numbers.

    Examples
    --------
    >>> multiply(2, 3)
    6
    >>> multiply(2.5, 3.5)
    8.75
    """
    return a * b

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

    Parameters
    ----------
    a : float or int
        The first number (dividend).
    b : float or int
        The second number (divisor).

    Returns
    -------
    float or int
        The quotient of the division.

    Raises
    ------
    ZeroDivisionError
        If the divisor (b) is zero.

    Examples
    --------
```



```
    Examples
    --------
    >>> divide(6, 2)
    3.0
    >>> divide(7, 2)
    3.5
    """
    if b == 0:
        raise ZeroDivisionError("Division by zero is not allowed.")
    return a / b

# Example usage:
print(f"2 + 3 = {add(2, 3)}")
print(f"5 - 2 = {subtract(5, 2)}")
print(f"2 * 3 = {multiply(2, 3)}")
print(f"6 / 2 = {divide(6, 2)}")

# Example of ZeroDivisionError
try:
    divide(5, 0)
except ZeroDivisionError as e:
    print(e)
```

```
2 + 3 = 5
5 - 2 = 3
2 * 3 = 6
6 / 2 = 3.0
Division by zero is not allowed.
```

#COMPARISION:

Comparing your docstrings with the ones I generated in the previous turn (the numpy-style ones), here are the main differences:

- Style: Your docstrings are more concise and use a simpler style, similar to what you might find in the "mupy style" (though not strictly adhering to a formal style guide like NumPy or Google). They directly describe the function's purpose.
- Information: My generated docstrings (in numpy style) are more structured and provide more detailed information about the function, including:
  - A concise summary line.
  - A more detailed explanation (though in this case, it's brief for these simple functions).
  - Parameters section: Explicitly lists the input parameters (x, y) and their types (float).
  - Returns section: Explicitly states what the function returns and its type (float or str for the divide function).
- Module Docstring: Both versions have a module-level docstring explaining the overall purpose of the script. Your module docstring is shorter, while mine provides a bit more context.

Both sets of docstrings serve the purpose of explaining the code. The numpy-style docstrings I generated are more formal and provide more detailed information, which is beneficial for larger projects or when working with others who might use automated documentation tools. Your docstrings are

	clear and effective for this simple example. **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**	