ASSIGNMENT 9.3

AI ASSISTANT CODING

NAME: K.sneha

HT NO:2303A51332

Batch:20

Task 1: Basic Docstring Generation

Prompt: Generate a Google Style function-level docstring for a Python function that calculates and returns the sum of even numbers and the sum of odd numbers from a given list of integers.

The docstring should include:

- A clear description of the function

- Args section with parameter types

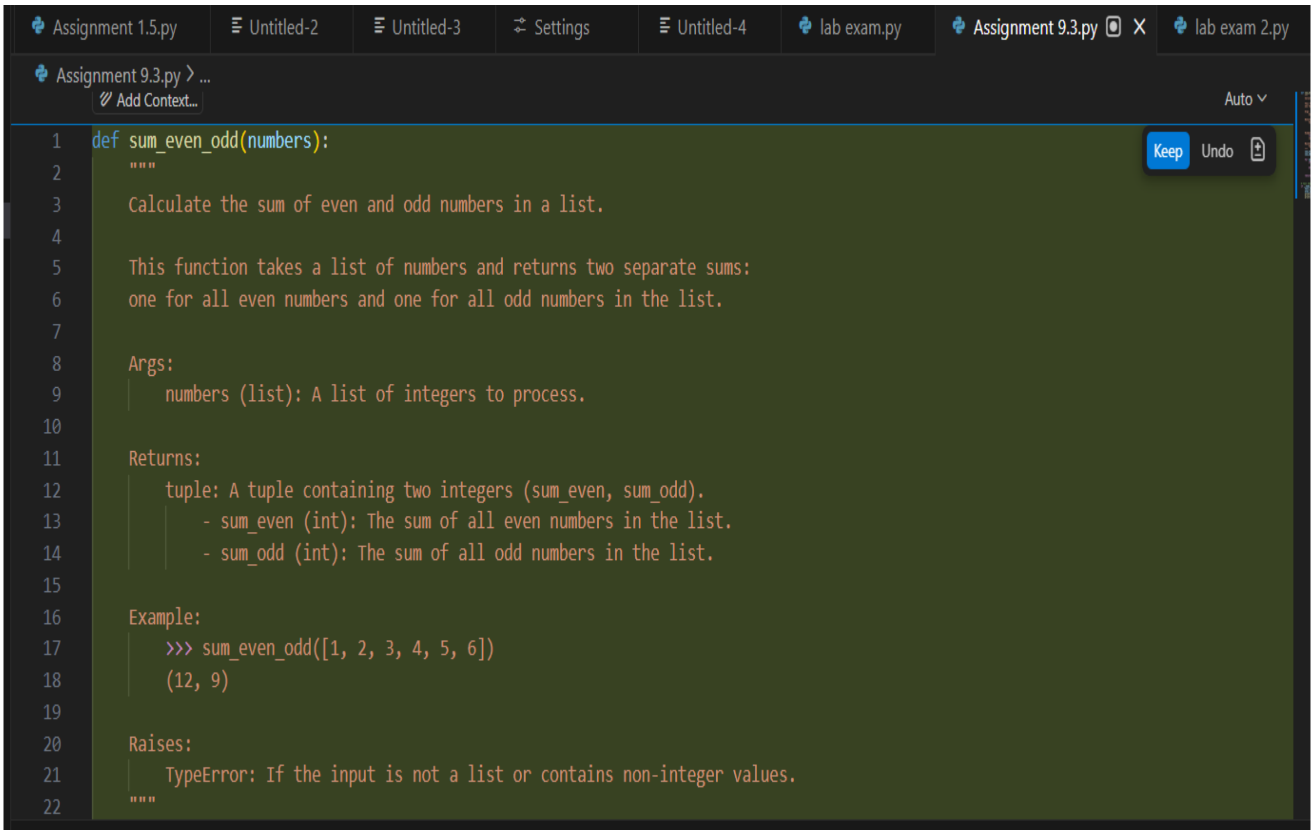
- Returns section with explanation of returned values

- Raises section for possible errors

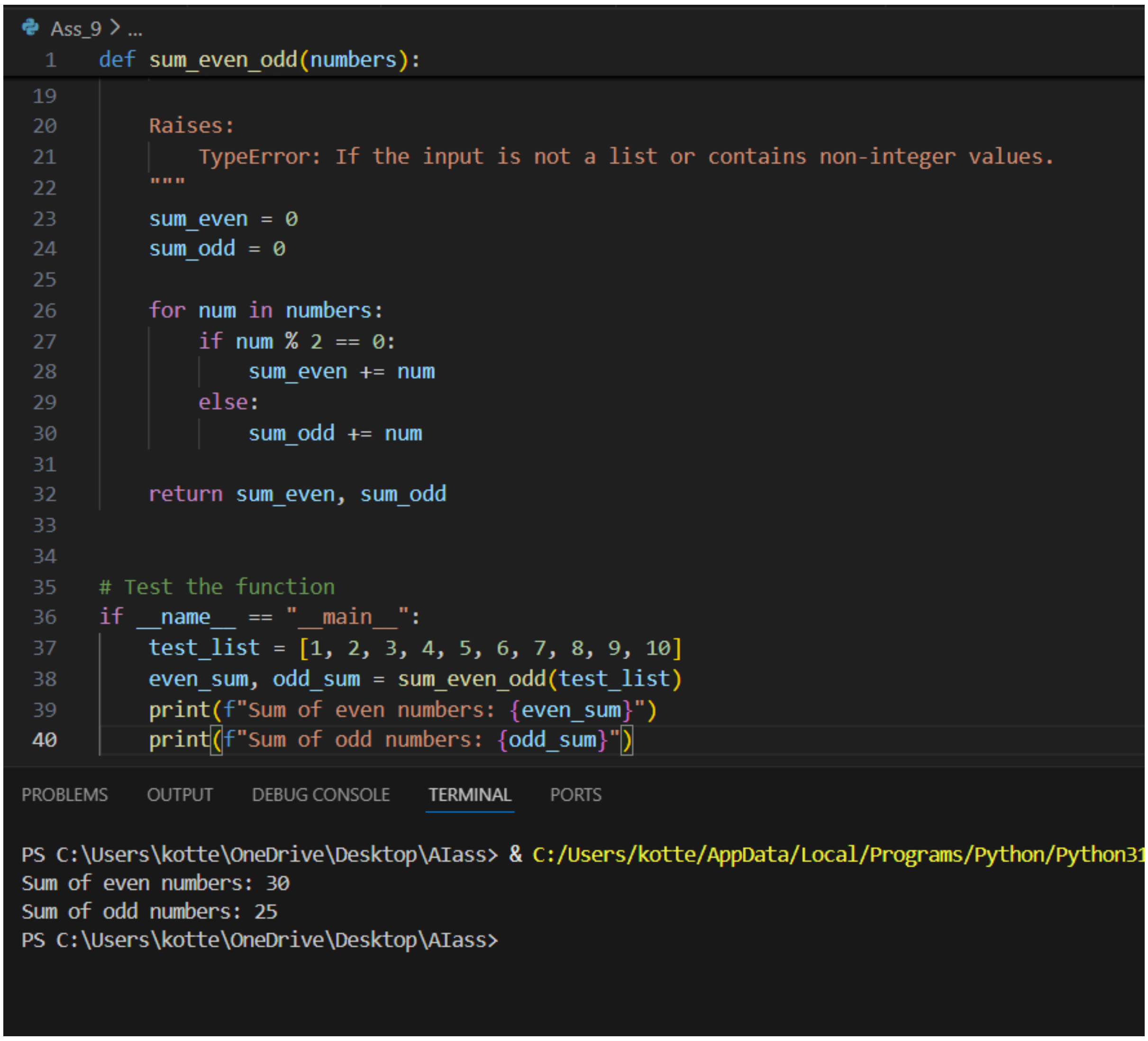
- Example usage

Follow proper Google Style documentation format.

#CODE



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

    This function takes a list of numbers and returns two separate sums:
    one for all even numbers and one for all odd numbers in the list.

    Args:
        numbers (list): A list of integers to process.

    Returns:
        tuple: A tuple containing two integers (sum_even, sum_odd).
            - sum_even (int): The sum of all even numbers in the list.
            - sum_odd (int): The sum of all odd numbers in the list.

    Example:
        >>> sum_even_odd([1, 2, 3, 4, 5, 6])
        (12, 9)

    Raises:
        TypeError: If the input is not a list or contains non-integer values.
    """
```



```
Ass_9 > ...
def sum_even_odd(numbers):

    Raises:
        TypeError: If the input is not a list or contains non-integer values.
    """
    sum_even = 0
    sum_odd = 0

    for num in numbers:
        if num % 2 == 0:
            sum_even += num
        else:
            sum_odd += num

    return sum_even, sum_odd


# Test the function
if __name__ == "__main__":
    test_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    even_sum, odd_sum = sum_even_odd(test_list)
    print(f"Sum of even numbers: {even_sum}")
    print(f"Sum of odd numbers: {odd_sum}")
```

```
PS C:\Users\kotte\OneDrive\Desktop\AIass> & C:/Users/kotte/AppData/Local/Programs/Python/Python31
Sum of even numbers: 30
Sum of odd numbers: 25
PS C:\Users\kotte\OneDrive\Desktop\AIass>
```

Comparison: Manual vs AI-Generated Docstring

Aspect	Manual Google-Style Docstring	AI-Generated Docstring
Clarity	Very clear and beginner-friendly	Clear but slightly generic
Format	Strict Google style (Args, Returns, Raises)	Mixed style (Parameters / Returns)
Type Details	Explicit type hints (list[int], tuple[int, int])	Types are less specific
Examples	Includes usage example	No example provided
Error Explanation	Clearly states error condition	Correct but brief
Completeness	High – covers all aspects	Moderate – missing example

Analysis & Understanding

Manual Docstring

- More detailed and structured
- Best for academic, professional, and long-term maintenance
- Follows Google Python Style Guide
- Helpful for beginners and reviewers

AI-Generated Docstring

- Fast and convenient
- Good for quick documentation
- May miss examples, strict formatting, or type specificity
- Useful as a starting point, but often needs refinement

Observation

- The manual Google-style docstring is more detailed, structured, and precise, making it suitable for academic use, professional projects, and long-term code maintenance.
- The AI-generated docstring is clear and concise, but it is more generic and lacks examples and strict type specificity.
- AI tools are efficient for quick documentation, but they may miss important details such as edge cases, usage examples, or exact formatting standards.
- Manual documentation ensures full control over clarity, correctness, and completeness, while AI assistance works best as a starting point rather than a final solution.
- Combining AI-generated docstrings with human refinement results in the best-quality documentation

Task 2: Automatic Inline Comments

Prompt: Automatically generate meaningful inline comments for a Python class named sru_student.

The class contains:

- Attributes: name, roll_no, hostel_status

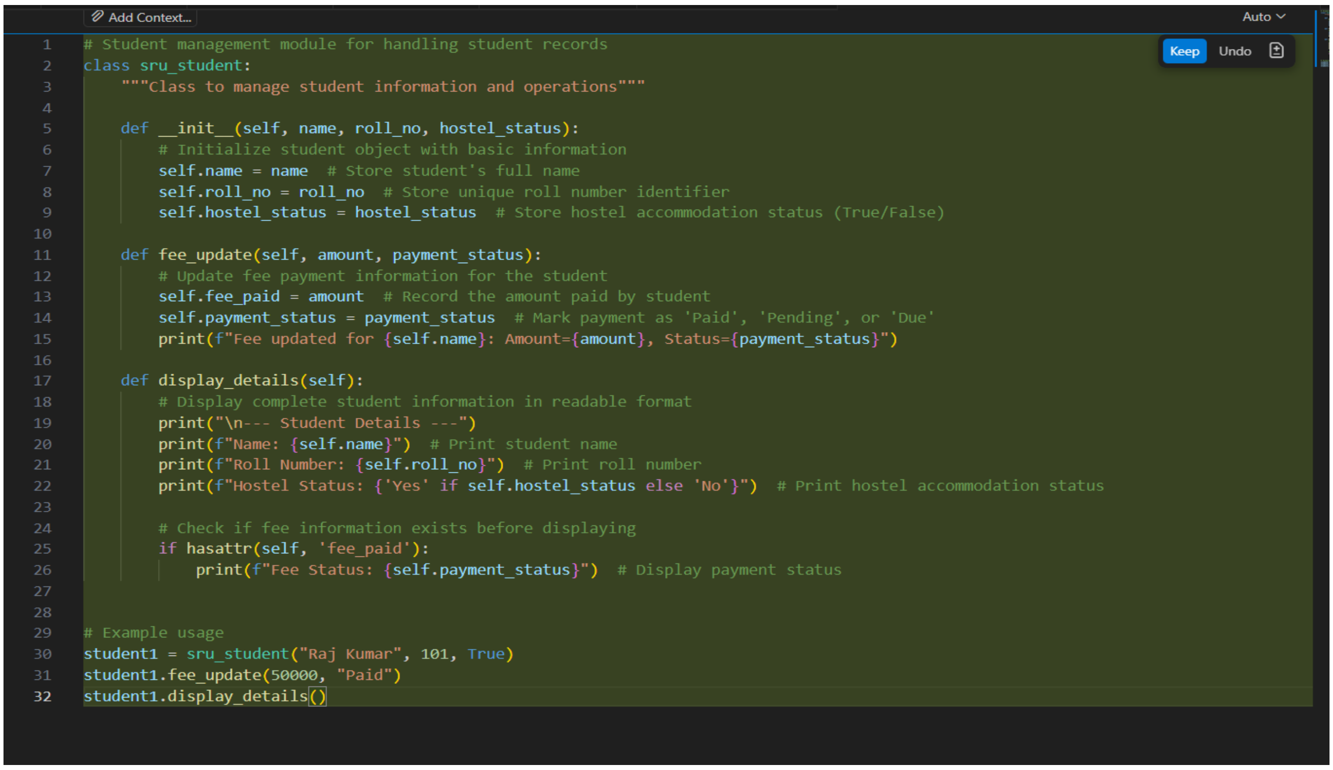
- Methods: fee_update() and display_details()

Add clear and helpful comments for each logical block and important line.

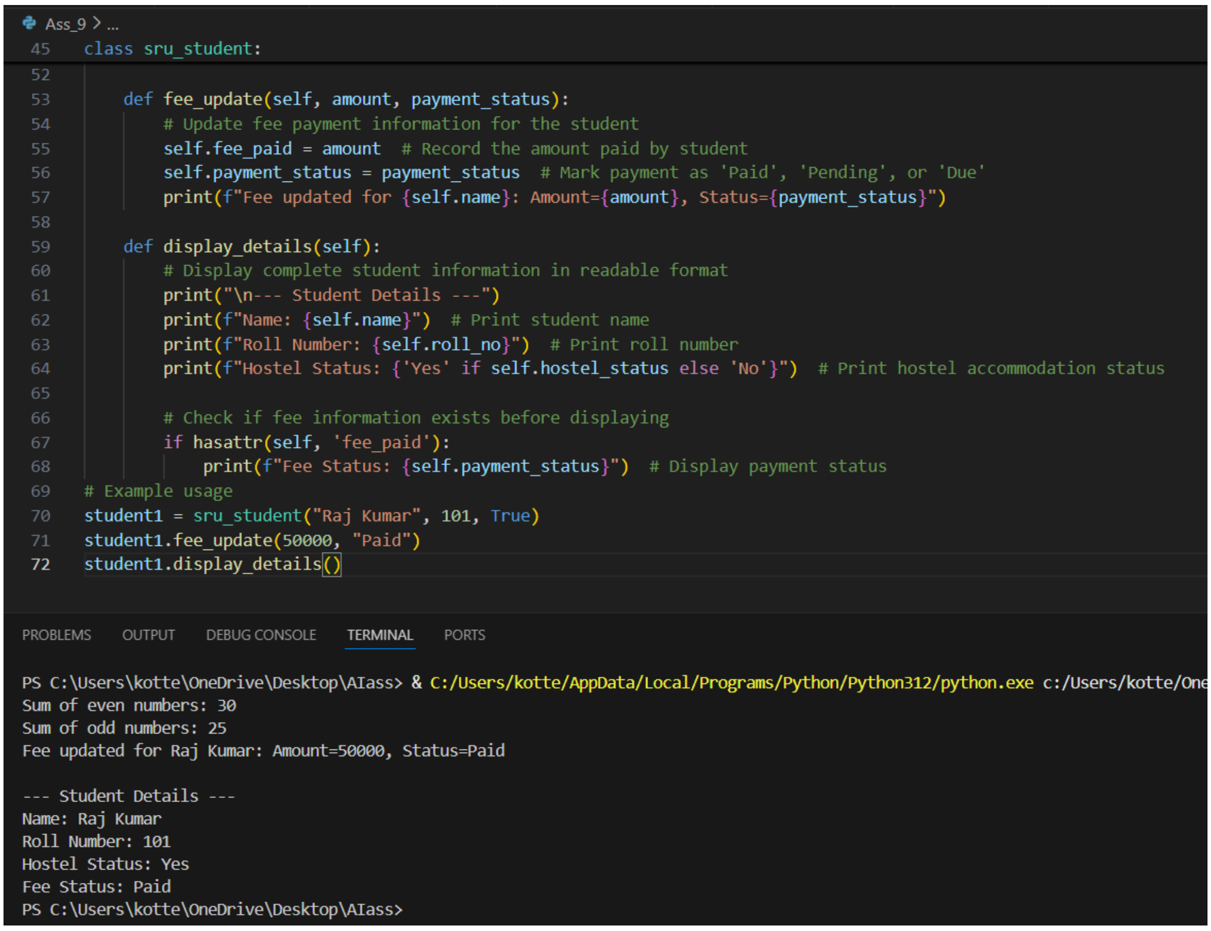
Avoid redundant or obvious comments.

Ensure the comments help new developers understand the purpose and logic of the code**.**

#CODE:



```python
# Student management module for handling student records
class sru_student:
    """Class to manage student information and operations"""

    def __init__(self, name, roll_no, hostel_status):
        # Initialize student object with basic information
        self.name = name  # Store student's full name
        self.roll_no = roll_no  # Store unique roll number identifier
        self.hostel_status = hostel_status  # Store hostel accommodation status (True/False)

    def fee_update(self, amount, payment_status):
        # Update fee payment information for the student
        self.fee_paid = amount  # Record the amount paid by student
        self.payment_status = payment_status  # Mark payment as 'Paid', 'Pending', or 'Due'
        print(f"Fee updated for {self.name}: Amount={amount}, Status={payment_status}")

    def display_details(self):
        # Display complete student information in readable format
        print("\n--- Student Details ---")
        print(f"Name: {self.name}")  # Print student name
        print(f"Roll Number: {self.roll_no}")  # Print roll number
        print(f"Hostel Status: {'Yes' if self.hostel_status else 'No'}")  # Print hostel accommodation status

        # Check if fee information exists before displaying
        if hasattr(self, 'fee_paid'):
            print(f"Fee Status: {self.payment_status}")  # Display payment status


# Example usage
student1 = sru_student("Raj Kumar", 101, True)
student1.fee_update(50000, "Paid")
student1.display_details()
```



```python
class sru_student:

    def fee_update(self, amount, payment_status):
        # Update fee payment information for the student
        self.fee_paid = amount  # Record the amount paid by student
        self.payment_status = payment_status  # Mark payment as 'Paid', 'Pending', or 'Due'
        print(f"Fee updated for {self.name}: Amount={amount}, Status={payment_status}")

    def display_details(self):
        # Display complete student information in readable format
        print("\n--- Student Details ---")
        print(f"Name: {self.name}")  # Print student name
        print(f"Roll Number: {self.roll_no}")  # Print roll number
        print(f"Hostel Status: {'Yes' if self.hostel_status else 'No'}")  # Print hostel accommodation status

        # Check if fee information exists before displaying
        if hasattr(self, 'fee_paid'):
            print(f"Fee Status: {self.payment_status}")  # Display payment status
# Example usage
student1 = sru_student("Raj Kumar", 101, True)
student1.fee_update(50000, "Paid")
student1.display_details()
```

```
PS C:\Users\kotte\OneDrive\Desktop\AIass> & C:/Users/kotte/AppData/Local/Programs/Python/Python312/python.exe c:/Users/kotte/One
Sum of even numbers: 30
Sum of odd numbers: 25
Fee updated for Raj Kumar: Amount=50000, Status=Paid

--- Student Details ---
Name: Raj Kumar
Roll Number: 101
Hostel Status: Yes
Fee Status: Paid
PS C:\Users\kotte\OneDrive\Desktop\AIass>
```

Comparison: Manual vs AI-Generated Inline Comments

Aspect	Manual Comments	AI-Generated Comments
Clarity	Very clear and explanatory	Clear but slightly generic
Detail Level	Explains *why* and *what*	Mostly explains *what*
Redundancy	Minimal redundancy	Some obvious comments
Accuracy	Fully accurate	Accurate
Beginner Friendly	Highly suitable	Moderately suitable
Context Awareness	Better understanding of logic	Less contextual depth

Missing, Redundant, or Incorrect AI Comments

Missing

- AI comments do not explain why hasattr() is used
- No clarification about hostel_status being boolean

Redundant

- Comments like *"Assigns name to the student"* repeat obvious code behavior

Incorrect

- No incorrect comments found (logic is correctly interpreted)

Critical Discussion: Strengths & Limitations of AI Comments

Strengths

- Saves time and effort
- Generates readable and consistent comments
- Useful for quick understanding
- Helpful for large codebases

Limitations

- Often too generic
- Misses deeper intent or design reasoning
- Can add unnecessary comments
- Still requires human review

Observation

- The manual inline comments are more meaningful and explanatory. They clarify both the purpose and logic of the code, making it easier for beginners and new developers to understand.
- The AI-generated comments are clear but mostly describe what the code is doing rather than why it is being done.
- AI comments sometimes include obvious or redundant explanations, such as repeating simple assignments.
- Important logic details (like the reason for using hasattr() to check if fee data exists) are better explained in the manual version.
- AI-generated comments are useful for saving time, but they still require human review and refinement.

Task 3: Module-Level and Function-Level Documentation

Prompt: Generate structured NumPy-style documentation for a Python calculator module that contains the following functions: add, subtract, multiply, and divide.

The documentation should include:

1. A detailed module-level docstring explaining:

  - Purpose of the module

  - List of available functions

  - Example usage

2. NumPy-style function-level docstrings for each function including:

  - Parameters

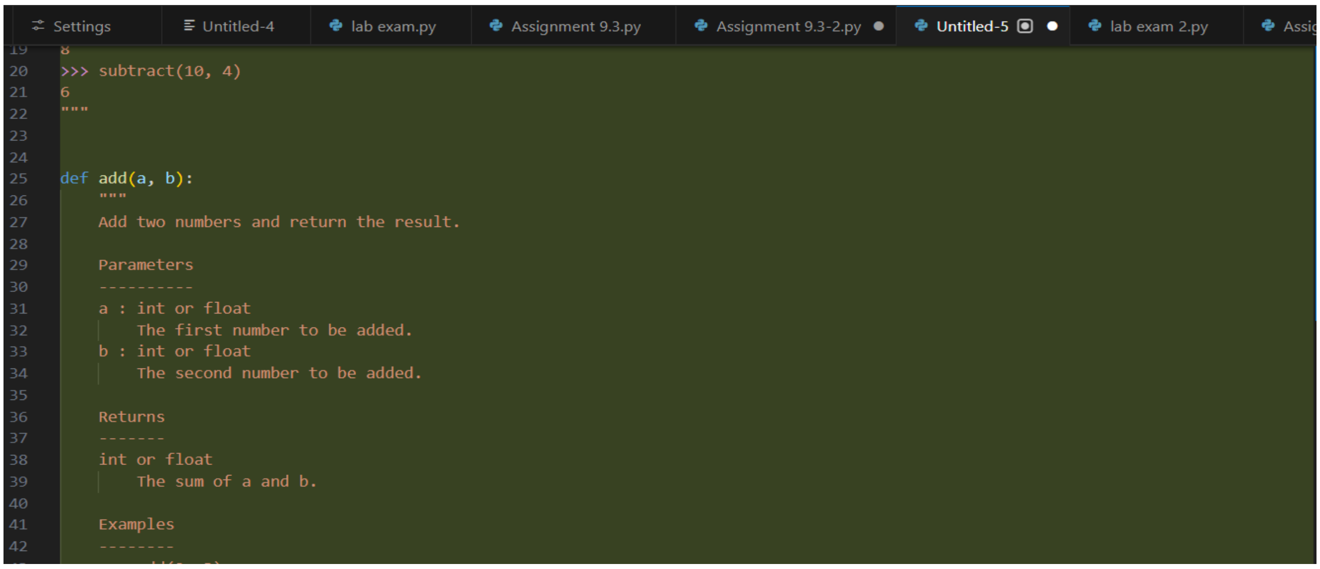
  - Returns

  - Raises (if applicable)

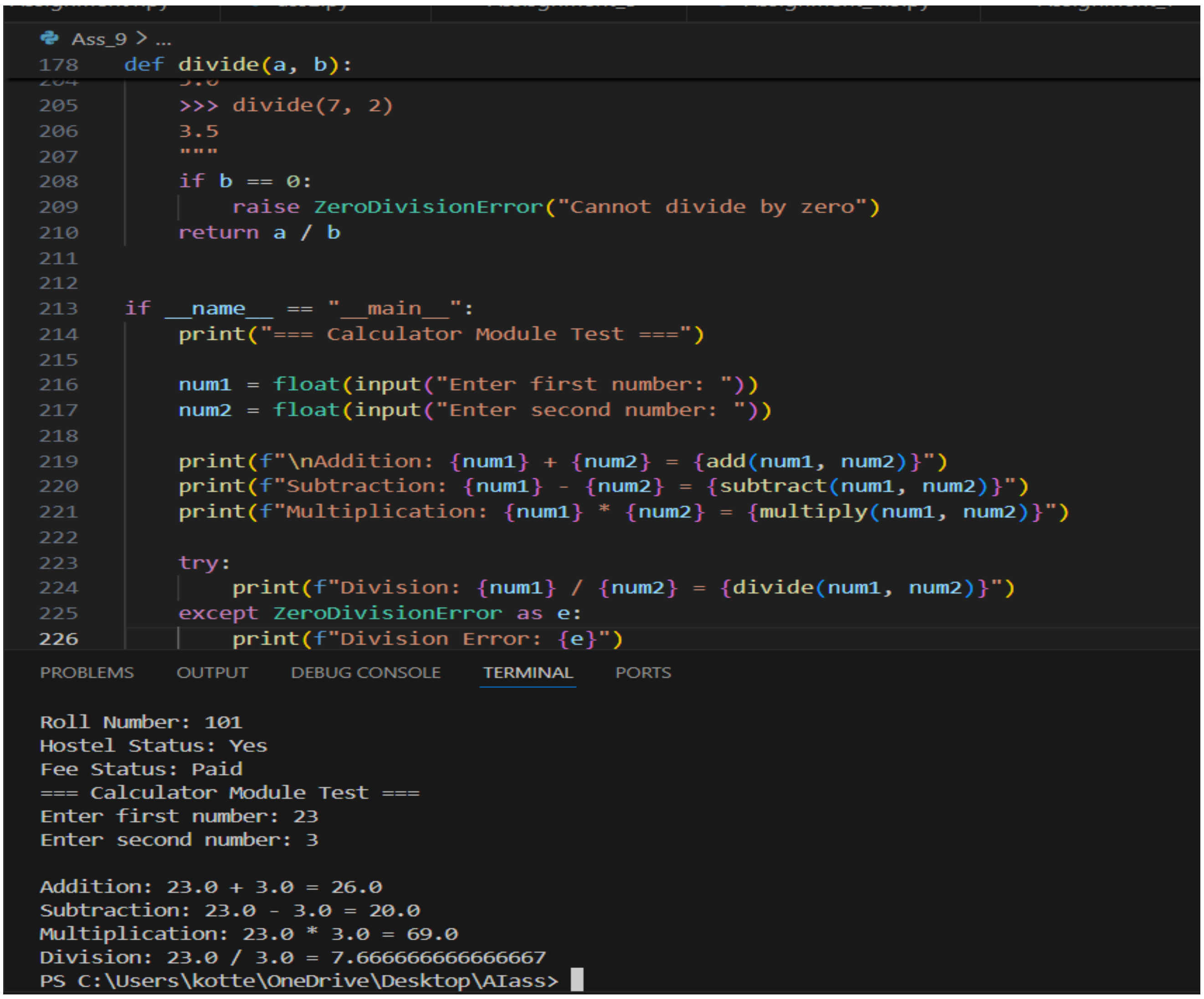
  - Examples

Ensure proper NumPy documentation structure and formatting.

#code:



```
    8
    >>> subtract(10, 4)
    6
    """


def add(a, b):
    """
    Add two numbers and return the result.

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

    Returns
    -------
    int or float
        The sum of a and b.

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



```
def divide(a, b):
    >>> divide(7, 2)
    3.5
    """
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b


if __name__ == "__main__":
    print("=== Calculator Module Test ===")

    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))

    print(f"\nAddition: {num1} + {num2} = {add(num1, num2)}")
    print(f"Subtraction: {num1} - {num2} = {subtract(num1, num2)}")
    print(f"Multiplication: {num1} * {num2} = {multiply(num1, num2)}")

    try:
        print(f"Division: {num1} / {num2} = {divide(num1, num2)}")
    except ZeroDivisionError as e:
        print(f"Division Error: {e}")
```

```
Roll Number: 101
Hostel Status: Yes
Fee Status: Paid
=== Calculator Module Test ===
Enter first number: 23
Enter second number: 3

Addition: 23.0 + 3.0 = 26.0
Subtraction: 23.0 - 3.0 = 20.0
Multiplication: 23.0 * 3.0 = 69.0
Division: 23.0 / 3.0 = 7.666666666666667
PS C:\Users\kotte\OneDrive\Desktop\AIass>
```

Comparison: Manual vs AI-Generated Documentation

Aspect	**Manual Docstrings**	**AI-Generated Docstrings**
Structure	Strict NumPy style	Mostly NumPy style
Clarity	Very clear and precise	Clear but generic
Examples	Can include detailed examples	Often missing
Error Handling	Explicitly documented	Correct but brief
Consistency	Fully consistent	Sometimes varies
Time Efficiency	Time-consuming	Very fast

Understanding Structured Documentation for Multi-Function Scripts

- Module-level docstrings explain the overall purpose and scope of the script.
- Function-level docstrings describe inputs, outputs, errors, and usage.
- NumPy-style documentation improves readability, standardization, and reusability.
- AI tools are helpful for quick documentation, but manual refinement ensures quality.
- Well-documented modules are easier to maintain and share across projects.

Observation

- The manual NumPy-style docstrings are well-structured, precise, and consistent, making the module easy to understand and reuse across projects.
- AI-generated docstrings closely follow the NumPy format and provide clear descriptions, showing strong accuracy for standard functions like add, subtract, multiply, and divide.
- Manual documentation demonstrates better control over technical wording, examples, and exception handling, especially for edge cases such as division by zero.
- AI-generated documentation improves speed and uniformity, but may sometimes be slightly verbose or generic.
- Both module-level and function-level docstrings significantly improve readability, maintainability, and usability of multi-function scripts.