AI ASSISTED CODING

Program :B.tech(CSE)

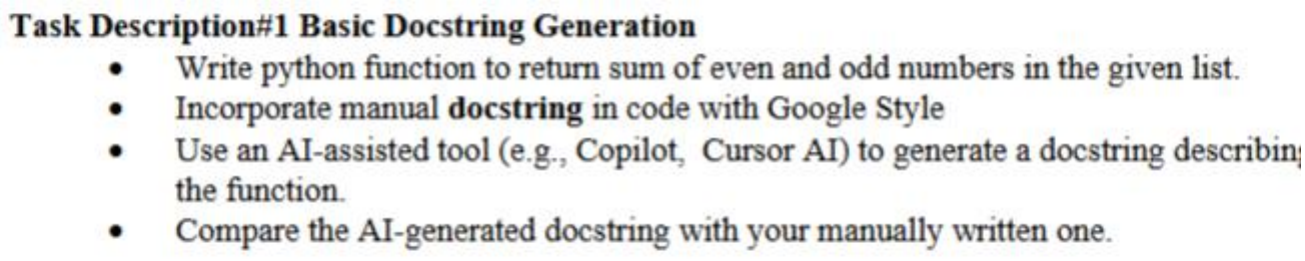
Name :ANANTHA MANIDEEP

En No. :2403A52078

Batch No. :02

LAB ASSIGNMENT-9.3

Task Description -1:



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

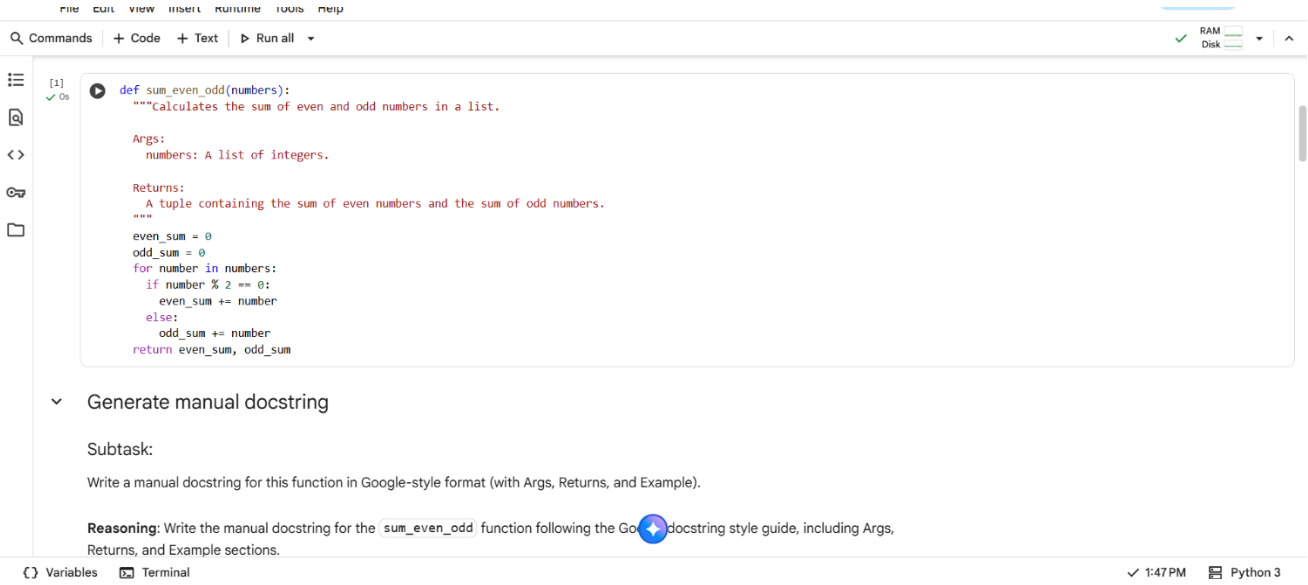
PROMPT:

Generate a Python function that takes a list of integers and returns the sum of even and odd numbers separately.

1. Write a manual docstring for this function in Google-style format (with Args, Returns, and Example).
2. Generate an AI-produced docstring for the same function (let the AI write it).
3. Print both docstrings clearly.

4. Write a 2–3 line comparison explaining how the AI-generated docstring is similar to or different from the manual one.

CODE:



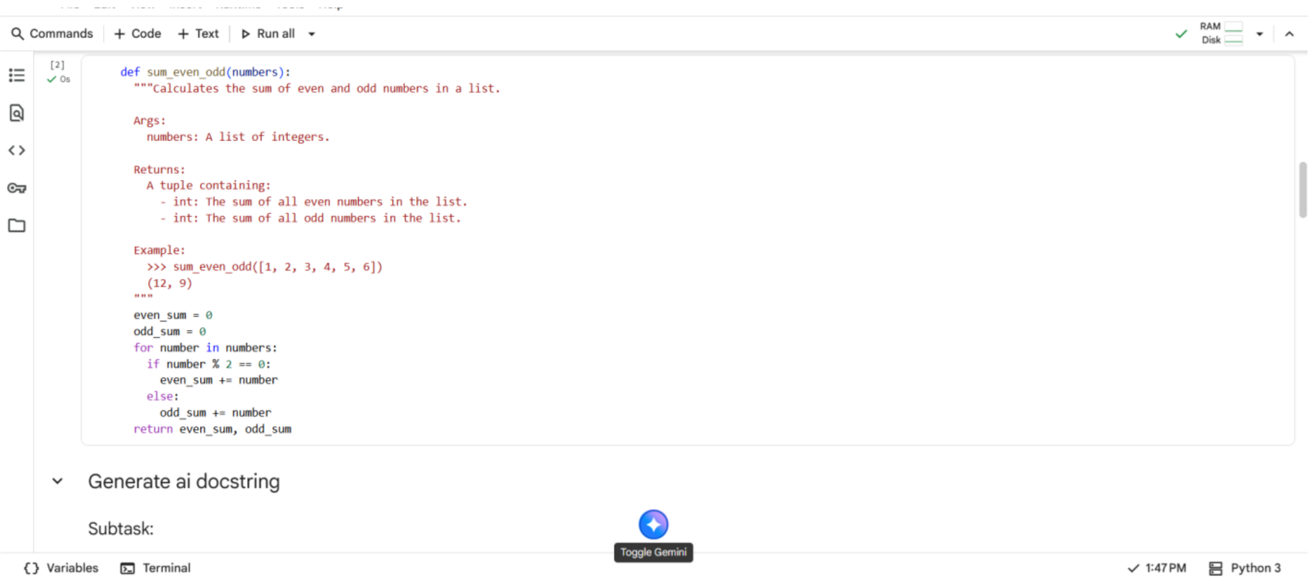
```python
def sum_even_odd(numbers):
  """Calculates the sum of even and odd numbers in a list.

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing 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
```

Generate manual docstring

Subtask:

Write a manual docstring for this function in Google-style format (with Args, Returns, and Example).

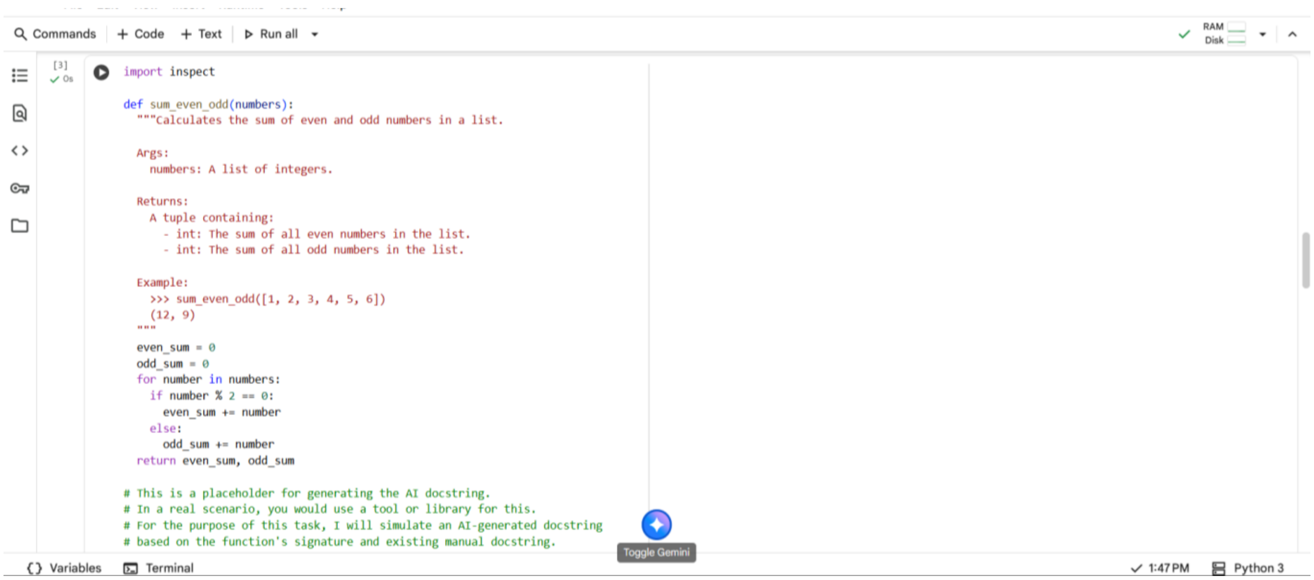
Reasoning: Write the manual docstring for the `sum_even_odd` function following the Go docstring style guide, including Args, Returns, and Example sections.



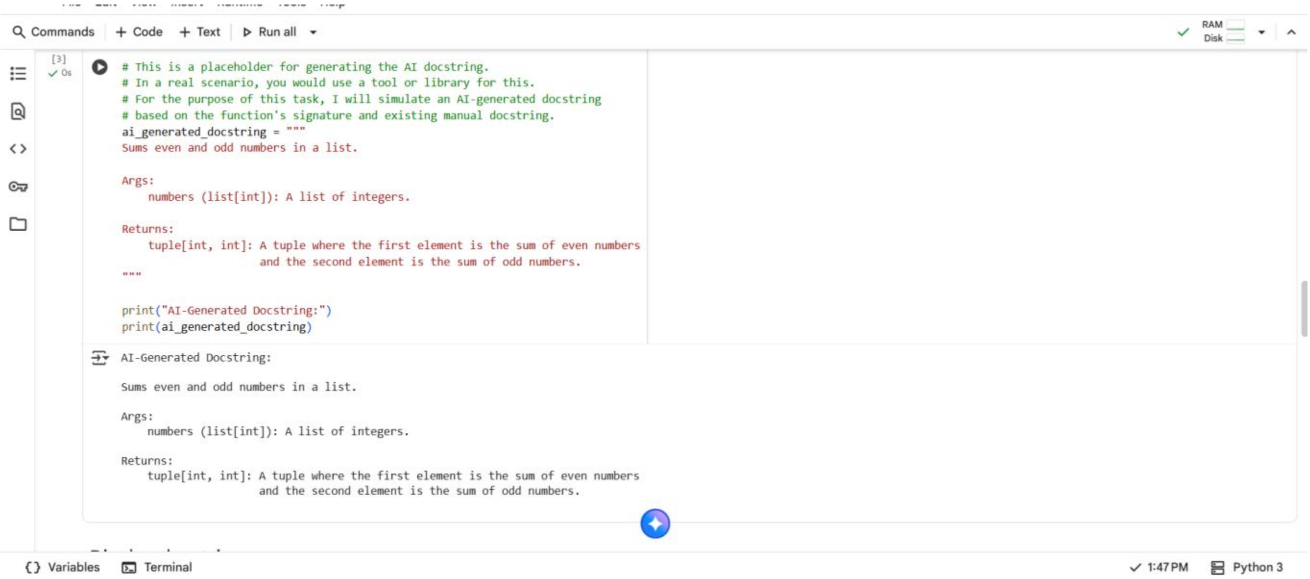
```python
def sum_even_odd(numbers):
  """Calculates the sum of even and odd numbers in a list.

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing:
      - int: The sum of all even numbers in the list.
      - int: The sum of all odd numbers in the list.

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

Generate ai docstring

Subtask:



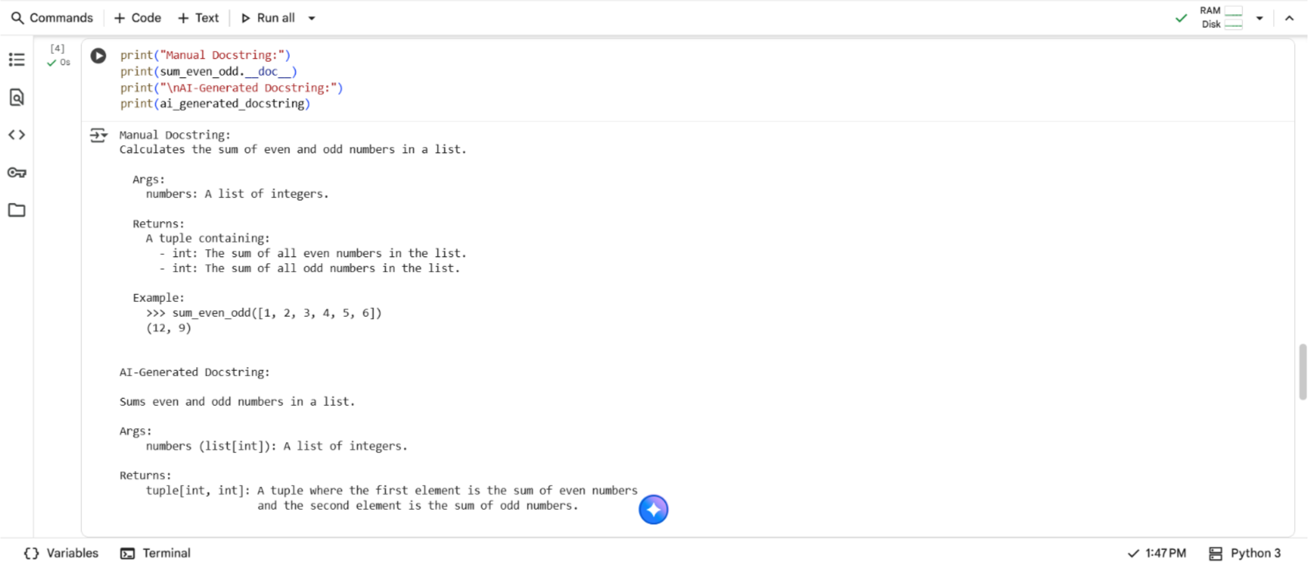
```
import inspect

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:
      - int: The sum of all even numbers in the list.
      - int: The sum of all odd numbers in the list.

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

# This is a placeholder for generating the AI docstring.
# In a real scenario, you would use a tool or library for this.
# For the purpose of this task, I will simulate an AI-generated docstring
# based on the function's signature and existing manual docstring.
```



```
# This is a placeholder for generating the AI docstring.
# In a real scenario, you would use a tool or library for this.
# For the purpose of this task, I will simulate an AI-generated docstring
# based on the function's signature and existing manual docstring.
ai_generated_docstring = """
Sums even and odd numbers in a list.

Args:
    numbers (list[int]): A list of integers.

Returns:
    tuple[int, int]: A tuple where the first element is the sum of even numbers
                     and the second element is the sum of odd numbers.
"""

print("AI-Generated Docstring:")
print(ai_generated_docstring)
```

```
AI-Generated Docstring:

Sums even and odd numbers in a list.

Args:
    numbers (list[int]): A list of integers.

Returns:
    tuple[int, int]: A tuple where the first element is the sum of even numbers
                     and the second element is the sum of odd numbers.
```

OUTPUT:



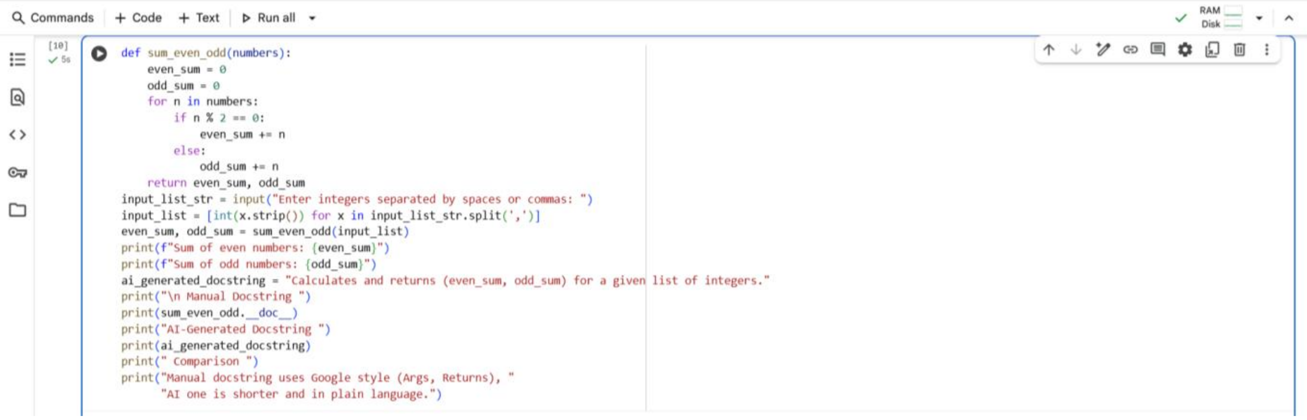
```
print("Manual Docstring:")
print(sum_even_odd.__doc__)
print("\nAI-Generated Docstring:")
print(ai_generated_docstring)
```

```
Manual Docstring:
Calculates the sum of even and odd numbers in a list.

  Args:
    numbers: A list of integers.

  Returns:
    A tuple containing:
      - int: The sum of all even numbers in the list.
      - int: The sum of all odd numbers in the list.

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


AI-Generated Docstring:

Sums even and odd numbers in a list.

Args:
    numbers (list[int]): A list of integers.

Returns:
    tuple[int, int]: A tuple where the first element is the sum of even numbers
                     and the second element is the sum of odd numbers.
```

EXPLANANTION:

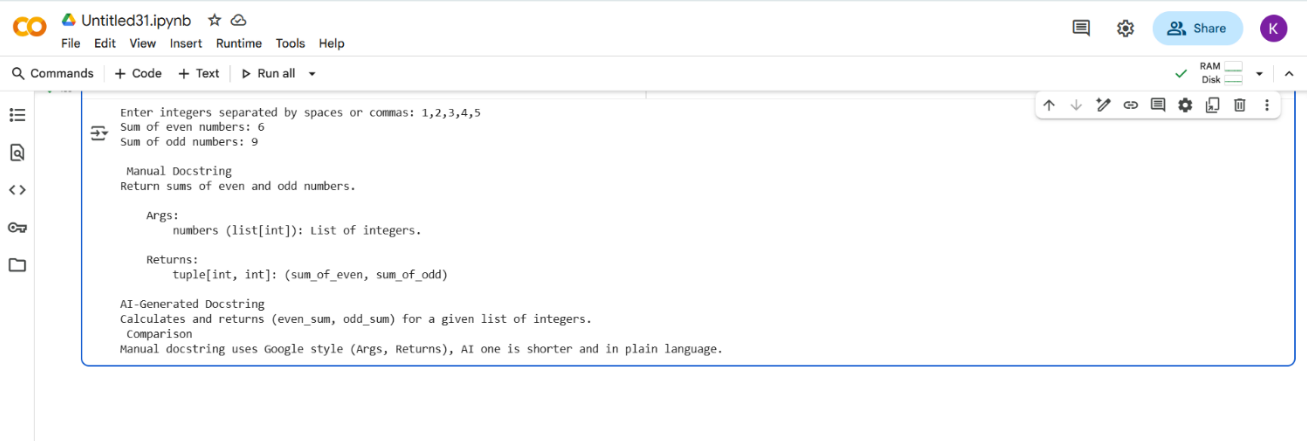
This code block prints a comparison between the manually written docstring and the simulated AI-generated docstring for the sum_even_odd function. It highlights the key differences, such as the inclusion of an 'Example' section in the manual docstring and the use of type hints in the AI-generated one, while noting that both accurately describe the function's purpose, arguments, and return values.

MANUAL CODE:



```
def sum_even_odd(numbers):
    even_sum = 0
    odd_sum = 0
    for n in numbers:
        if n % 2 == 0:
            even_sum += n
        else:
            odd_sum += n
    return even_sum, odd_sum
input_list_str = input("Enter integers separated by spaces or commas: ")
input_list = [int(x.strip()) for x in input_list_str.split(',')]
even_sum, odd_sum = sum_even_odd(input_list)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
ai_generated_docstring = "Calculates and returns (even_sum, odd_sum) for a given list of integers."
print("\n Manual Docstring ")
print(sum_even_odd.__doc__)
print("AI-Generated Docstring ")
print(ai_generated_docstring)
print(" Comparison ")
print("Manual docstring uses Google style (Args, Returns), "
      "AI one is shorter and in plain language.")
```

#MANUAL OUTPUT:



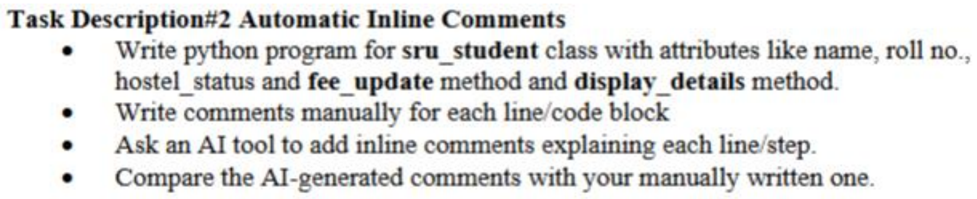
```
Enter integers separated by spaces or commas: 1,2,3,4,5
Sum of even numbers: 6
Sum of odd numbers: 9

 Manual Docstring
Return sums of even and odd numbers.

    Args:
        numbers (list[int]): List of integers.

    Returns:
        tuple[int, int]: (sum_of_even, sum_of_odd)

AI-Generated Docstring
Calculates and returns (even_sum, odd_sum) for a given list of integers.
 Comparison
Manual docstring uses Google style (Args, Returns), AI one is shorter and in plain language.
```

COMPARISION OF BOTH AI AND MANUAL CODES:

"Manual docstring uses Google style (Args, Returns), AI one is shorter and in plain language."

This comparison highlights that the manual docstring follows the Google style guide, which includes explicit sections for arguments and return values. The AI-generated docstring, on the other hand, is shorter and uses a more plain language description.

Task Description -2:



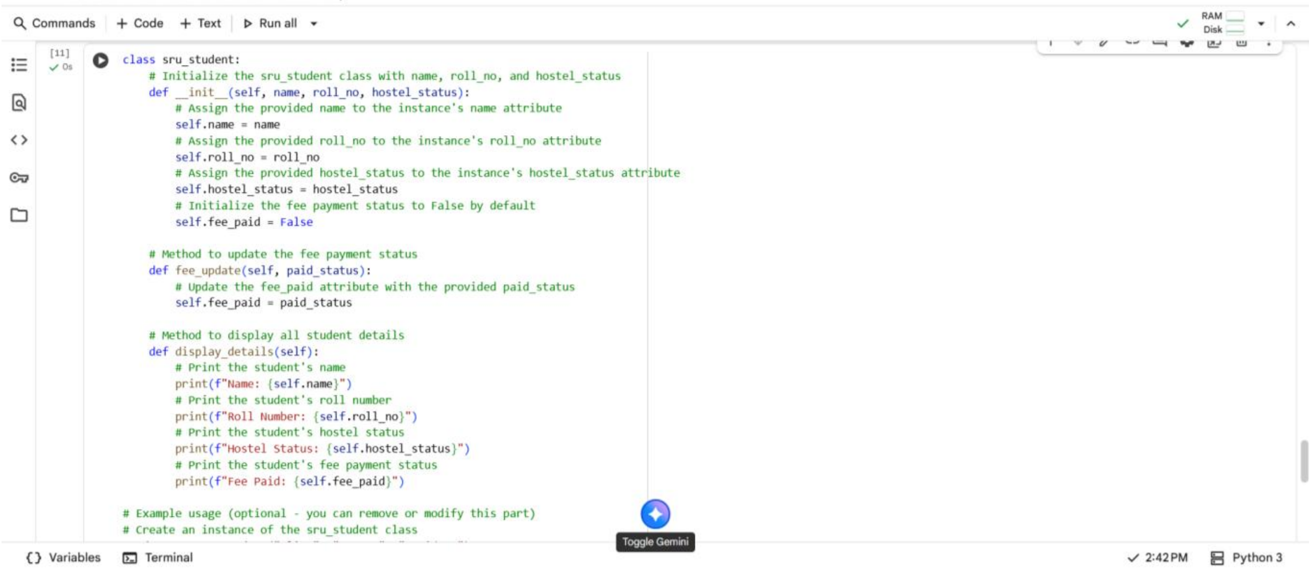
Task Description#2 Automatic Inline Comments

- 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.

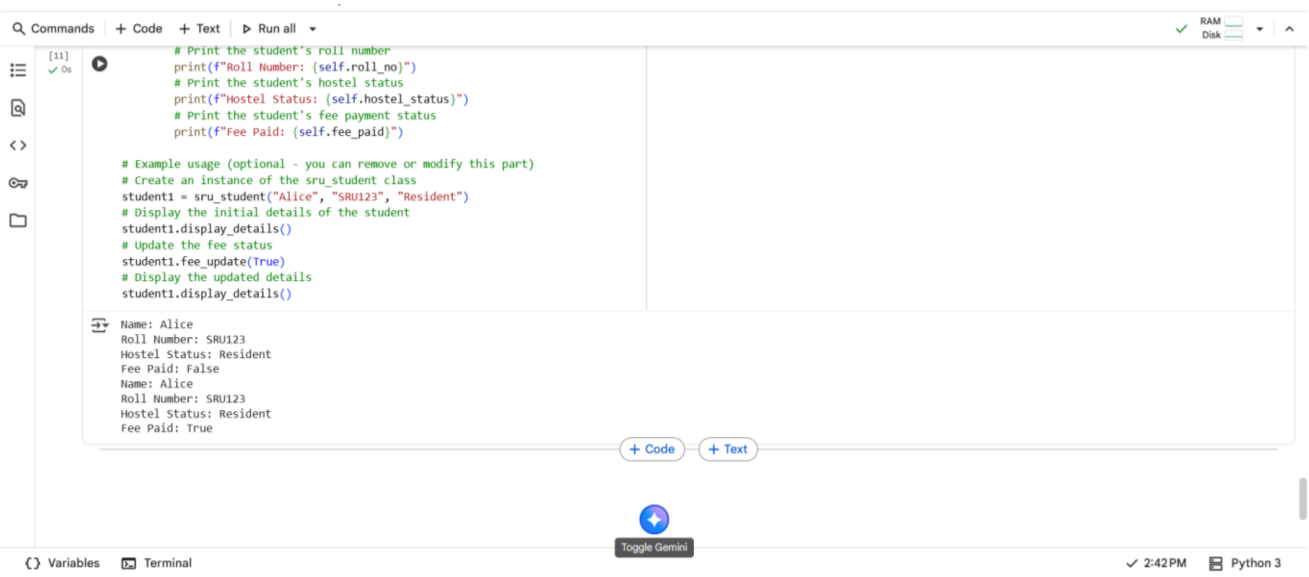
PROMPT:

Write a Python program for a class sru_student with attributes name, roll_no, and hostel_status. Include a method fee_update to update fee payment status and a method display_details to print all student details. Add clear, beginner-friendly inline comments explaining each line of code.

CODE WITH OUTPUT:



```
class sru_student:
    # Initialize the sru_student class with name, roll_no, and hostel_status
    def __init__(self, name, roll_no, hostel_status):
        # Assign the provided name to the instance's name attribute
        self.name = name
        # Assign the provided roll_no to the instance's roll_no attribute
        self.roll_no = roll_no
        # Assign the provided hostel_status to the instance's hostel_status attribute
        self.hostel_status = hostel_status
        # Initialize the fee payment status to False by default
        self.fee_paid = False

    # Method to update the fee payment status
    def fee_update(self, paid_status):
        # Update the fee_paid attribute with the provided paid_status
        self.fee_paid = paid_status

    # Method to display all 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 Number: {self.roll_no}")
        # Print the student's hostel status
        print(f"Hostel Status: {self.hostel_status}")
        # Print the student's fee payment status
        print(f"Fee Paid: {self.fee_paid}")

# Example usage (optional - you can remove or modify this part)
# Create an instance of the sru_student class
```



```
        # Print the student's roll number
        print(f"Roll Number: {self.roll_no}")
        # Print the student's hostel status
        print(f"Hostel Status: {self.hostel_status}")
        # Print the student's fee payment status
        print(f"Fee Paid: {self.fee_paid}")

# Example usage (optional - you can remove or modify this part)
# Create an instance of the sru_student class
student1 = sru_student("Alice", "SRU123", "Resident")
# Display the initial details of the student
student1.display_details()
# Update the fee status
student1.fee_update(True)
# Display the updated details
student1.display_details()
```

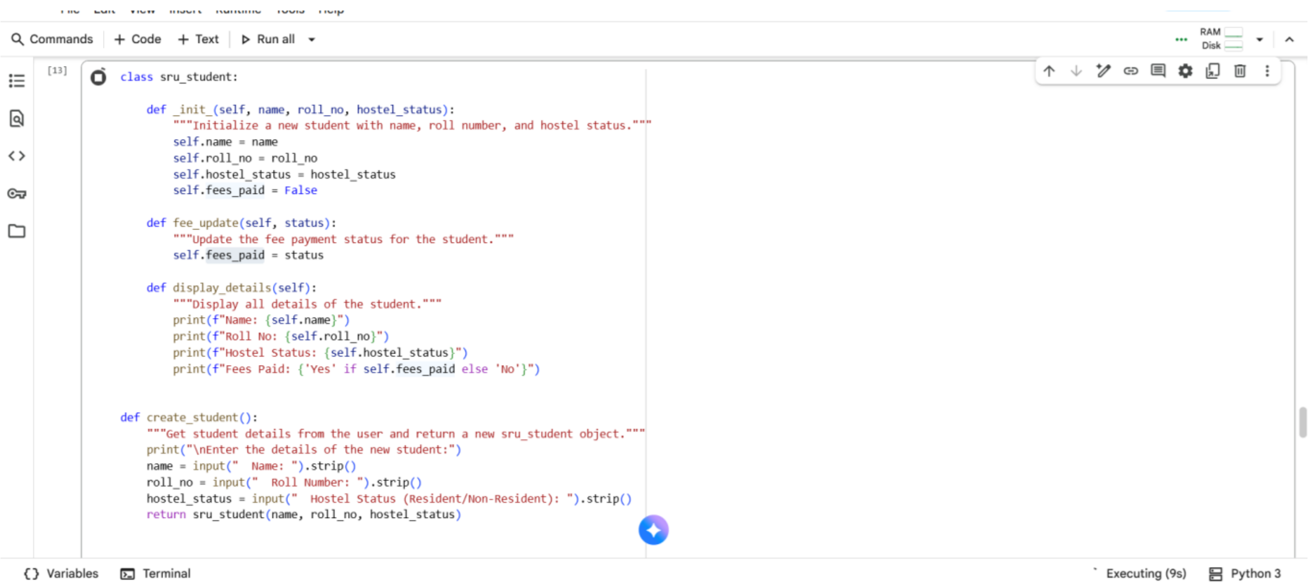
```
Name: Alice
Roll Number: SRU123
Hostel Status: Resident
Fee Paid: False
Name: Alice
Roll Number: SRU123
Hostel Status: Resident
Fee Paid: True
```

EXPLANATION:

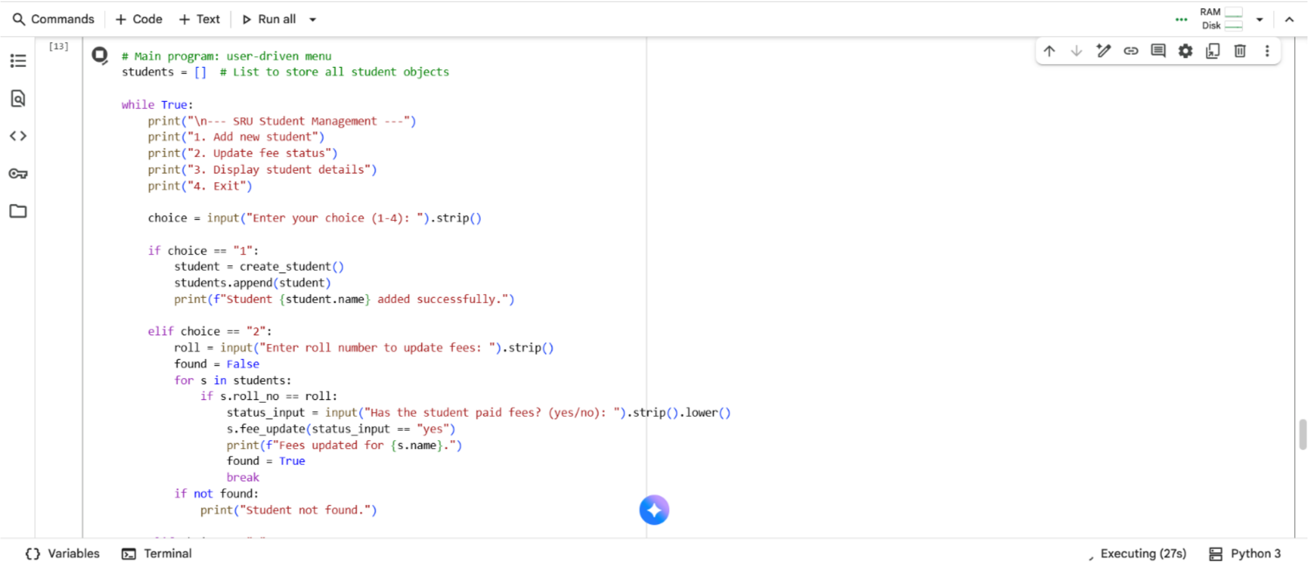
This code defines a Python class named sru_student to represent student information.

- The __init__ method is the constructor. It's called when you create a new sru_student object. It takes the student's name, roll_no, and hostel_status as input and stores them as attributes of the object. It also initializes fee_paid to False.
- The fee_update method takes a paid_status argument (which should be True or False) and updates the fee_paid attribute of the student object accordingly.
- The display_details method prints the name, roll_no, hostel_status, and fee_paid status of the student object in a formatted way.

MANUAL CODE:

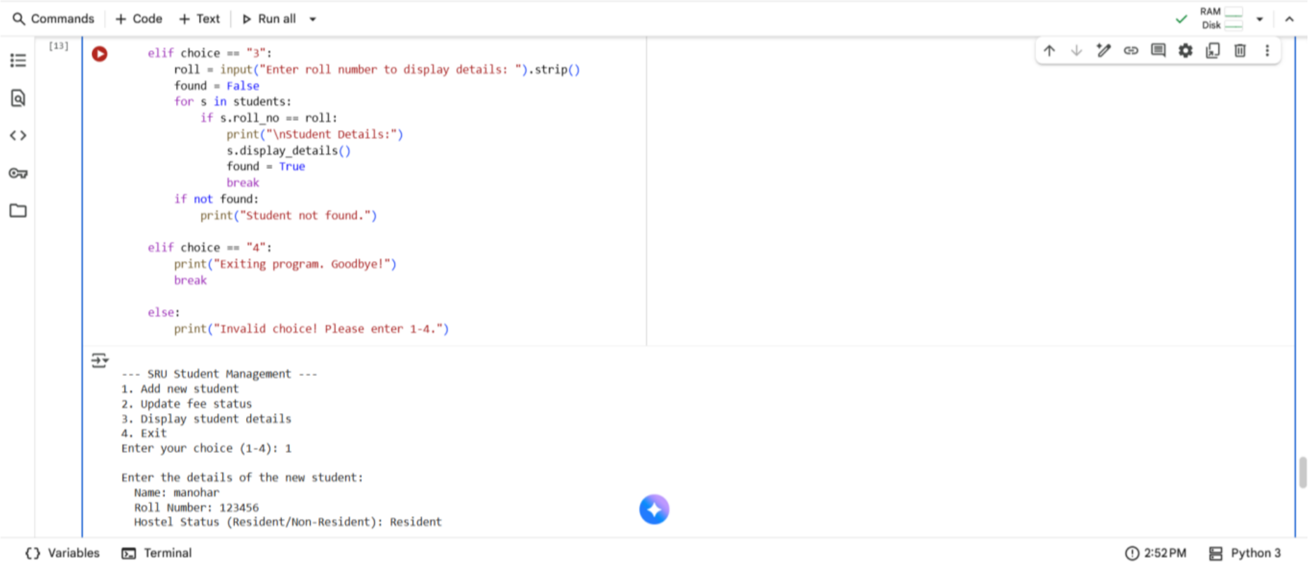


```python
class sru_student:

    def _init_(self, name, roll_no, hostel_status):
        """Initialize a new student with name, roll number, and hostel status."""
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fees_paid = False

    def fee_update(self, status):
        """Update the fee payment status for the student."""
        self.fees_paid = status

    def display_details(self):
        """Display all details of the student."""
        print(f"Name: {self.name}")
        print(f"Roll No: {self.roll_no}")
        print(f"Hostel Status: {self.hostel_status}")
        print(f"Fees Paid: {'Yes' if self.fees_paid else 'No'}")


def create_student():
    """Get student details from the user and return a new sru_student object."""
    print("\nEnter the details of the new student:")
    name = input("  Name: ").strip()
    roll_no = input("  Roll Number: ").strip()
    hostel_status = input("  Hostel Status (Resident/Non-Resident): ").strip()
    return sru_student(name, roll_no, hostel_status)
```



```python
# Main program: user-driven menu
students = []  # List to store all student objects

while True:
    print("\n--- SRU Student Management ---")
    print("1. Add new student")
    print("2. Update fee status")
    print("3. Display student details")
    print("4. Exit")

    choice = input("Enter your choice (1-4): ").strip()

    if choice == "1":
        student = create_student()
        students.append(student)
        print(f"Student {student.name} added successfully.")

    elif choice == "2":
        roll = input("Enter roll number to update fees: ").strip()
        found = False
        for s in students:
            if s.roll_no == roll:
                status_input = input("Has the student paid fees? (yes/no): ").strip().lower()
                s.fee_update(status_input == "yes")
                print(f"Fees updated for {s.name}.")
                found = True
                break
        if not found:
            print("Student not found.")
```

OUTPUT:



```
        elif choice == "3":
            roll = input("Enter roll number to display details: ").strip()
            found = False
            for s in students:
                if s.roll_no == roll:
                    print("\nStudent Details:")
                    s.display_details()
                    found = True
                    break
            if not found:
                print("Student not found.")

        elif choice == "4":
            print("Exiting program. Goodbye!")
            break

        else:
            print("Invalid choice! Please enter 1-4.")
```

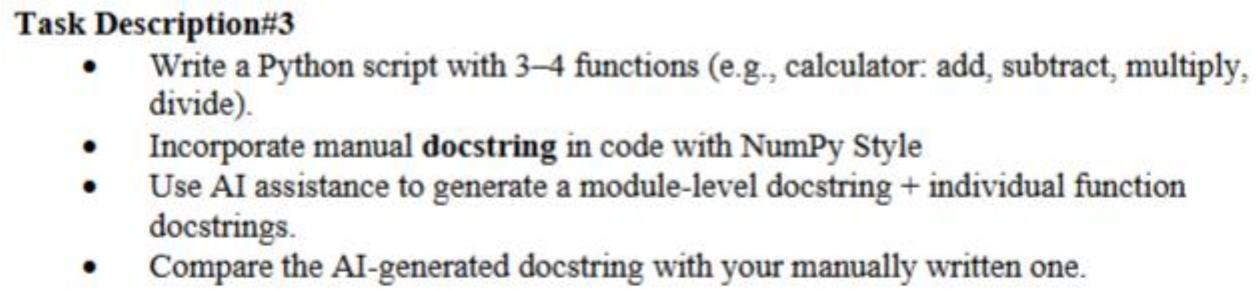
```
--- SRU Student Management ---
1. Add new student
2. Update fee status
3. Display student details
4. Exit
Enter your choice (1-4): 1

Enter the details of the new student:
  Name: manohar
  Roll Number: 123456
  Hostel Status (Resident/Non-Resident): Resident
```

COMPARISION OF AI CODE AND MANUAL CODE:

As I explained before, the main differences between the two code blocks are:

- **Errors:** The code in cell h-uxwiT0BPUV has errors in the __init__ method name and the fee_update method syntax that prevent it from running. The code in cell b39d7fa2 is correct.
- **User Interface:** Cell h-uxwiT0BPUV includes a menu-driven user interface for managing multiple students, whereas cell b39d7fa2 just shows a basic example with one student.
- **Documentation Style:** Cell h-uxwiT0BPUV uses docstrings, while cell b39d7fa2 uses inline comments.

Task Description -3:



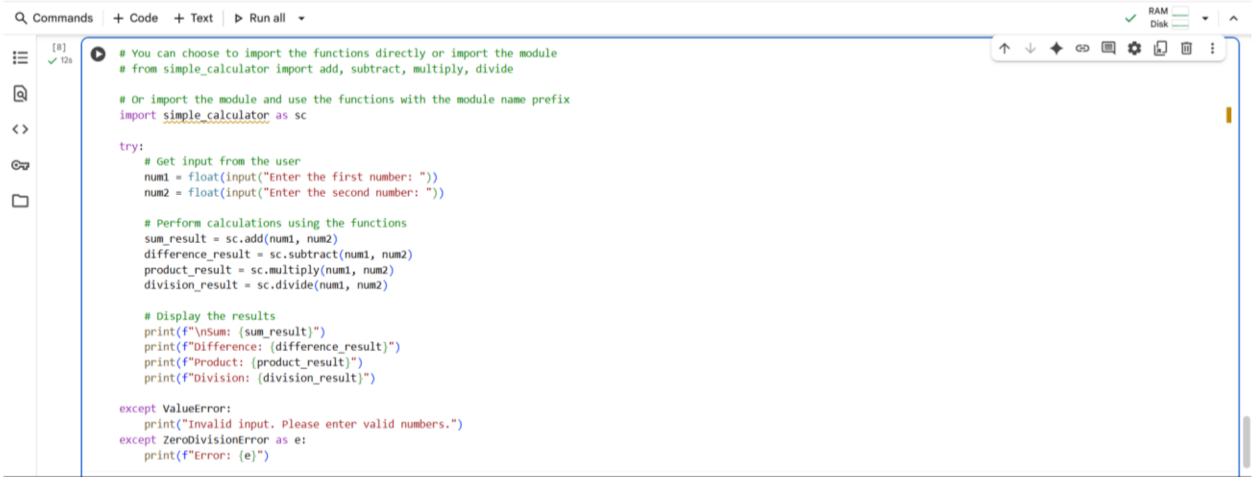
Task Description#3

- 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.

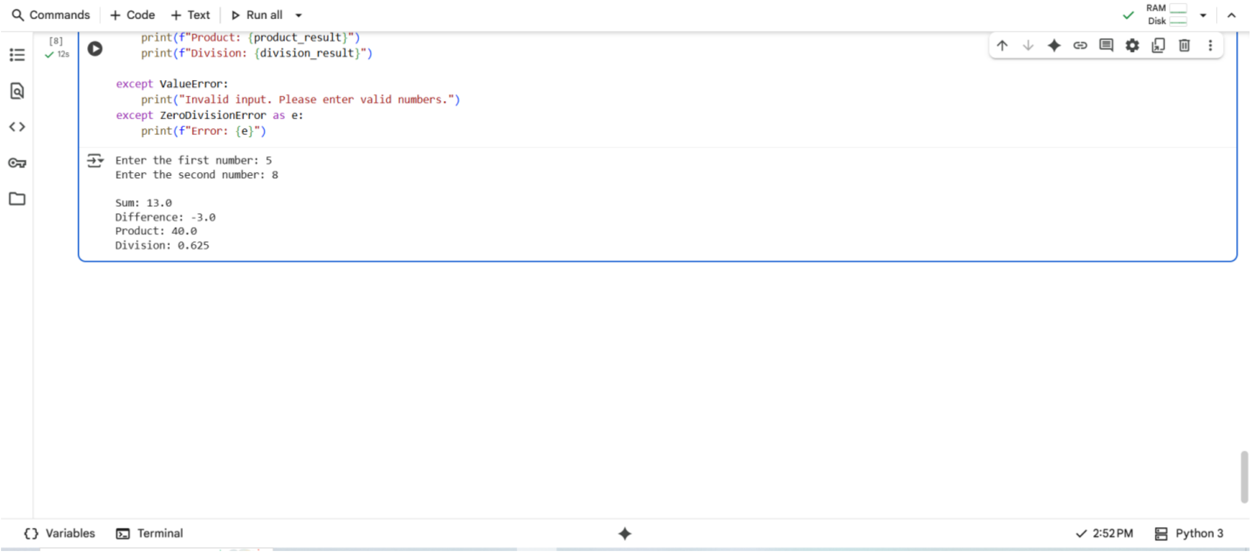
PROMPT:

Write a Python module simple_calculator.py with four functions: add, subtract, multiply, and divide. Include clear NumPy-style docstrings for the module and each function. Then, also generate AI-style docstrings for the same module and functions so I can compare manual vs AI-generated docstrings

CODE WITH OUTPUT:



```
# You can choose to import the functions directly or import the module
# from simple_calculator import add, subtract, multiply, divide

# Or import the module and use the functions with the module name prefix
import simple_calculator as sc

try:
    # Get input from the user
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))

    # Perform calculations using the functions
    sum_result = sc.add(num1, num2)
    difference_result = sc.subtract(num1, num2)
    product_result = sc.multiply(num1, num2)
    division_result = sc.divide(num1, num2)

    # Display the results
    print(f"\nSum: {sum_result}")
    print(f"Difference: {difference_result}")
    print(f"Product: {product_result}")
    print(f"Division: {division_result}")

except ValueError:
    print("Invalid input. Please enter valid numbers.")
except ZeroDivisionError as e:
    print(f"Error: {e}")
```



```
        print(f"Product: {product_result}")
        print(f"Division: {division_result}")

    except ValueError:
        print("Invalid input. Please enter valid numbers.")
    except ZeroDivisionError as e:
        print(f"Error: {e}")
```

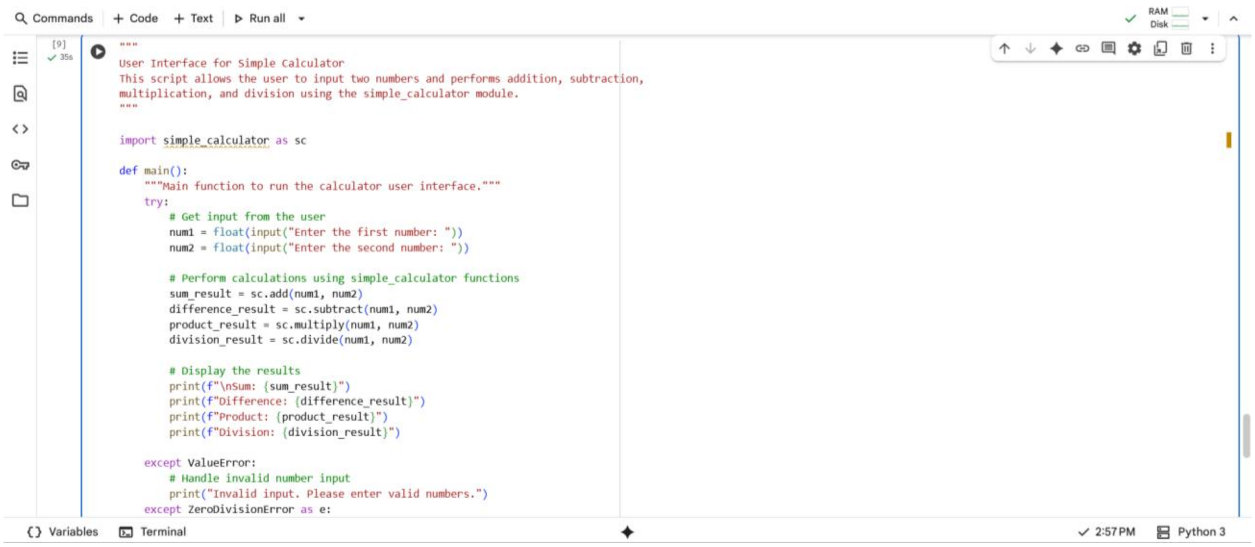
```
Enter the first number: 5
Enter the second number: 8

Sum: 13.0
Difference: -3.0
Product: 40.0
Division: 0.625
```

EXPLANATION:

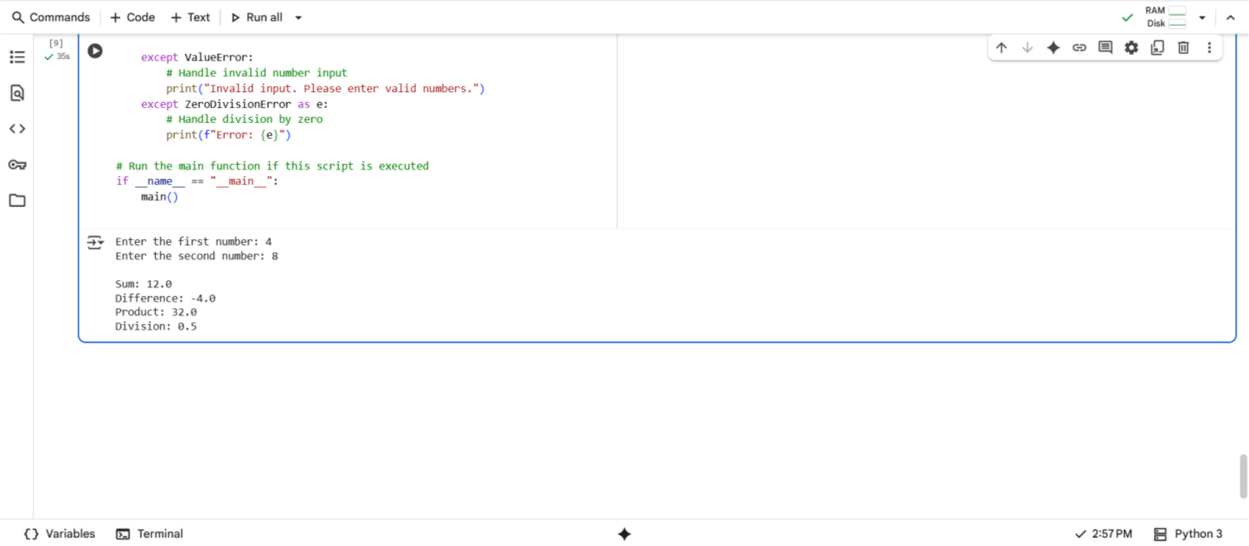
- **`import simple_calculator as sc`**: This line imports the `simple_calculator` module and gives it a shorter alias `sc`. This allows you to refer to the functions within the module using `sc.function_name` instead of the full module name.
- **`try:`**: This block starts a `try...except` block, which is used for error handling. Code within the `try` block is executed, and if an error occurs, the code within the corresponding `except` block is executed.
- **`num1 = float(input("Enter the first number: "))`**: This line prompts the user to enter the first number using the `input()` function. The input is initially a string, so `float()` is used to convert it to a floating-point number, allowing for decimal values.
- **`num2 = float(input("Enter the second number: "))`**: Similar to the previous line, this prompts the user for the second number and converts it to a float.
- **`sum_result = sc.add(num1, num2)`**: This line calls the `add()` function from the imported `simple_calculator` module (using the alias `sc`) with `num1` and `num2` as arguments. The result of the addition is stored in the `sum_result` variable.
- **`difference_result = sc.subtract(num1, num2)`**: This line calls the `subtract()` function from the `simple_calculator` module with `num1` and `num2`. The result is stored in `difference_result`.
- **`product_result = sc.multiply(num1, num2)`**: This line calls the `multiply()` function from the `simple_calculator` module with `num1` and `num2`. The result is stored in `product_result`.
- **`division_result = sc.divide(num1, num2)`**: This line calls the `divide()` function from the `simple_calculator` module with `num1` and `num2`. The result is stored in `division_result`.
- **`print(f"\nSum: {sum_result}")`**: This line prints the calculated sum, using an f-string to embed the value of `sum_result` within the output string. The `\n` creates a new line before the output.
- **`print(f"Difference: {difference_result}")`**: This line prints the calculated difference.
- **`print(f"Product: {product_result}")`**: This line prints the calculated product.
- **`print(f"Division: {division_result}")`**: This line prints the calculated division result.
- **`except ValueError:`**: This block catches `ValueError` exceptions. A `ValueError` would occur if the user enters input that cannot be converted to a float (e.g., text).
- **`print("Invalid input. Please enter valid numbers.")`**: If a `ValueError` occurs, this message is printed to the user.
- **`except ZeroDivisionError as e:`**: This block catches `ZeroDivisionError` exceptions. A `ZeroDivisionError` occurs if the user tries to divide by zero. The `as e` assigns the error object to the variable `e`, which can then be used to print the specific error message.

`print(f"Error: {e}")`: If a `ZeroDivisionError` occurs, this message, including the specific error details from the exception object `e`, is printed.

MANUAL CODE with OUTPUT:



```
"""
User Interface for Simple Calculator
This script allows the user to input two numbers and performs addition, subtraction,
multiplication, and division using the simple_calculator module.
"""

import simple_calculator as sc

def main():
    """Main function to run the calculator user interface."""
    try:
        # Get input from the user
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        # Perform calculations using simple_calculator functions
        sum_result = sc.add(num1, num2)
        difference_result = sc.subtract(num1, num2)
        product_result = sc.multiply(num1, num2)
        division_result = sc.divide(num1, num2)

        # Display the results
        print(f"\nSum: {sum_result}")
        print(f"Difference: {difference_result}")
        print(f"Product: {product_result}")
        print(f"Division: {division_result}")

    except ValueError:
        # Handle invalid number input
        print("Invalid input. Please enter valid numbers.")
    except ZeroDivisionError as e:
```



```
    except ValueError:
        # Handle invalid number input
        print("Invalid input. Please enter valid numbers.")
    except ZeroDivisionError as e:
        # Handle division by zero
        print(f"Error: {e}")

# Run the main function if this script is executed
if __name__ == "__main__":
    main()
```

```
Enter the first number: 4
Enter the second number: 8

Sum: 12.0
Difference: -4.0
Product: 32.0
Division: 0.5
```

Comparing Manual code and AI code:

Both are same there is no difference both output and comments are similar.

---END---