AI ASSISTED CODING

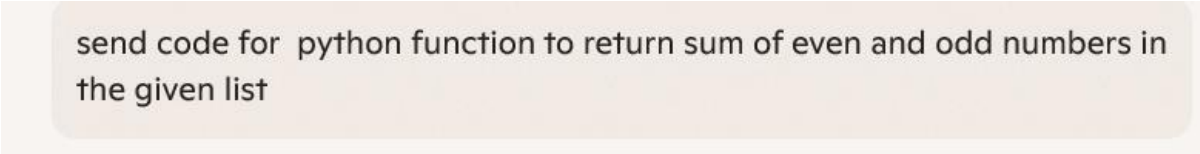
Assignment:9.3

Name:MD ZIAUDDIN

Rollno:2403A51271

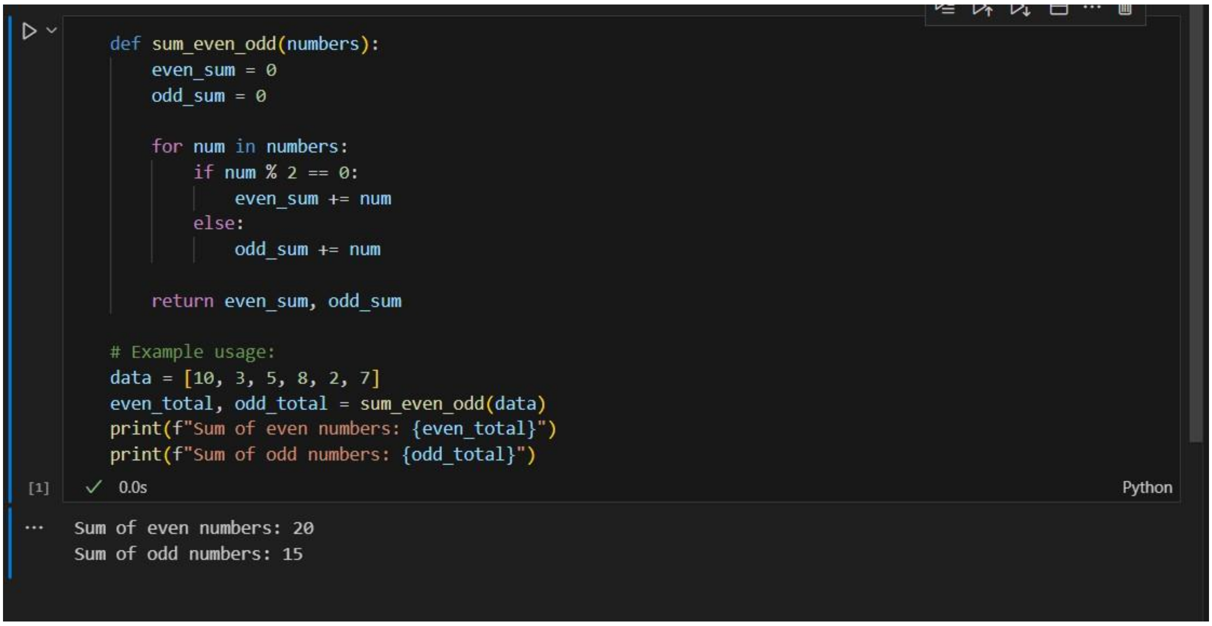
Task-1:

Prompt:



send code for python function to return sum of even and odd numbers in the given list

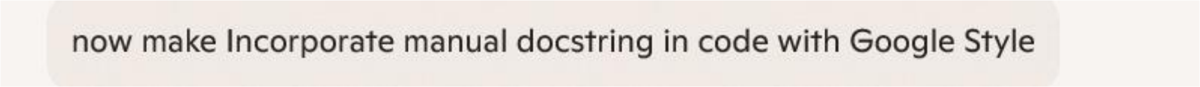
Code and output:



```
def sum_even_odd(numbers):
    even_sum = 0
    odd_sum = 0

    for num in numbers:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num

    return even_sum, odd_sum

# Example usage:
data = [10, 3, 5, 8, 2, 7]
even_total, odd_total = sum_even_odd(data)
print(f"Sum of even numbers: {even_total}")
print(f"Sum of odd numbers: {odd_total}")
```

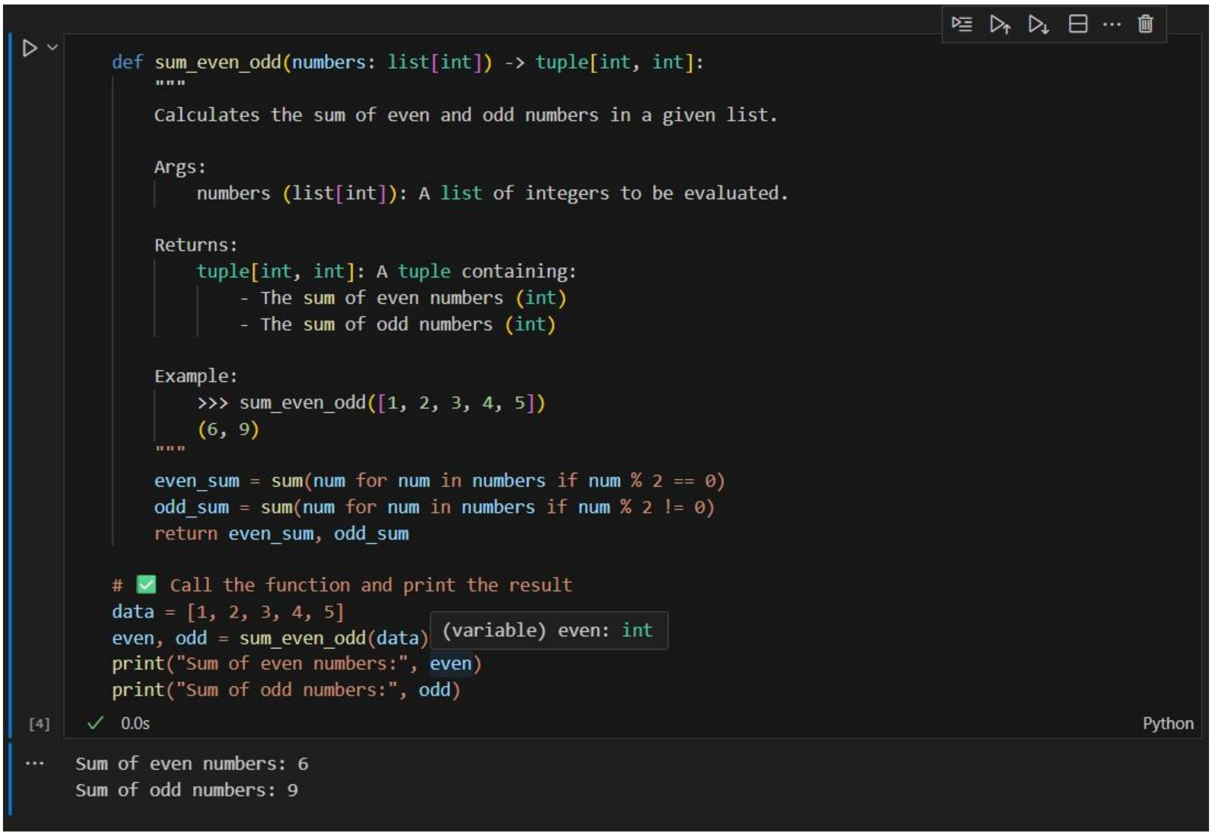
```
Sum of even numbers: 20
Sum of odd numbers: 15
```

Prompt2:



now make Incorporate manual docstring in code with Google Style

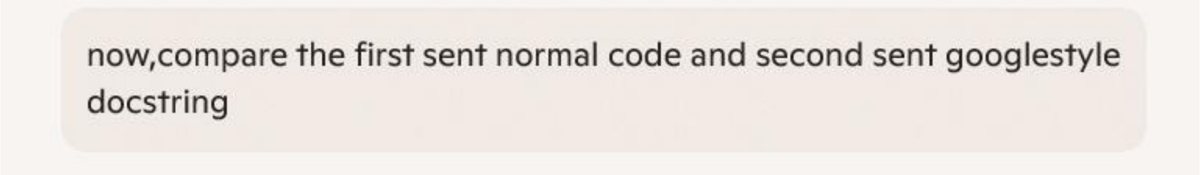
Code with output:



```python
def sum_even_odd(numbers: list[int]) -> tuple[int, int]:
    """
    Calculates the sum of even and odd numbers in a given list.

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

    Returns:
        tuple[int, int]: A tuple containing:
            - The sum of even numbers (int)
            - The sum of odd numbers (int)

    Example:
        >>> sum_even_odd([1, 2, 3, 4, 5])
        (6, 9)
    """
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum

# ✅ Call the function and print the result
data = [1, 2, 3, 4, 5]
even, odd = sum_even_odd(data)
print("Sum of even numbers:", even)
print("Sum of odd numbers:", odd)
```

```
Sum of even numbers: 6
Sum of odd numbers: 9
```

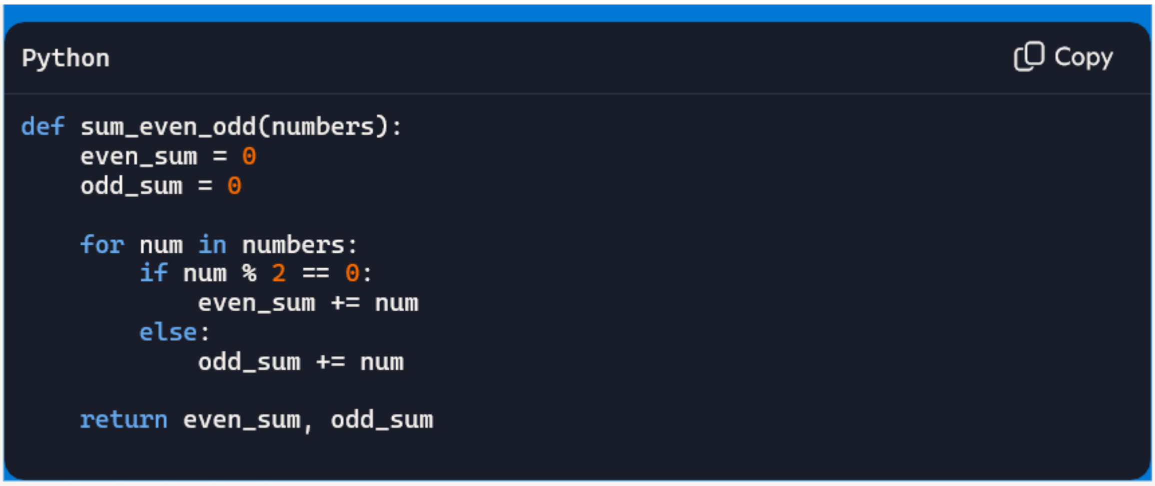
Prompt3:



now,compare the first sent normal code and second sent googlestyle docstring

Response:

1. Original Code (No Docstring)



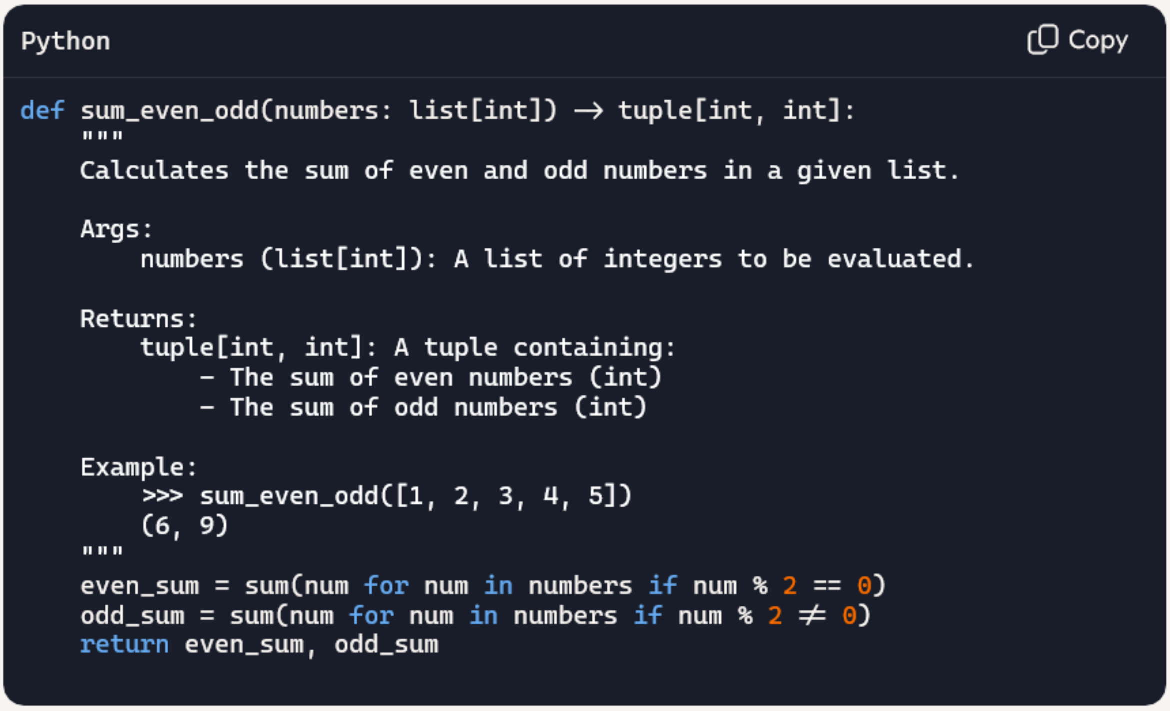
```python
def sum_even_odd(numbers):
    even_sum = 0
    odd_sum = 0

    for num in numbers:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num

    return even_sum, odd_sum
```

Characteristics:

- 🟩 Functional and correct

- + No explanation of purpose
- + No type hints
- + No guidance for usage
- + Less readable for collaborators or future you

2. Google-Style Docstring Version



```python
def sum_even_odd(numbers: list[int]) -> tuple[int, int]:
    """
    Calculates the sum of even and odd numbers in a given list.

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

    Returns:
        tuple[int, int]: A tuple containing:
            - The sum of even numbers (int)
            - The sum of odd numbers (int)

    Example:
        >>> sum_even_odd([1, 2, 3, 4, 5])
        (6, 9)
    """
    even_sum = sum(num for num in numbers if num % 2 == 0)
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    return even_sum, odd_sum
```

Characteristics:

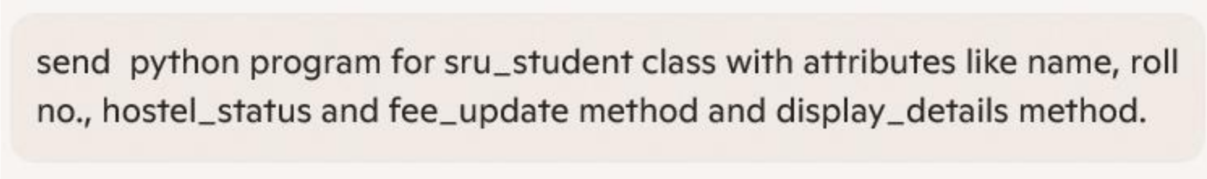
- 🟩 Clear function purpose
- 🟩 Type hints for parameters and return values
- 🟩 Example usage for quick understanding
- 🟩 Structured and IDE-friendly
- 🟩 Easier to maintain and extend

Comparison Summary

Adding a Google-style docstring doesn't change the logic—it **elevates the clarity** and makes your code **self-explanatory**, especially in collaborative or production environments.

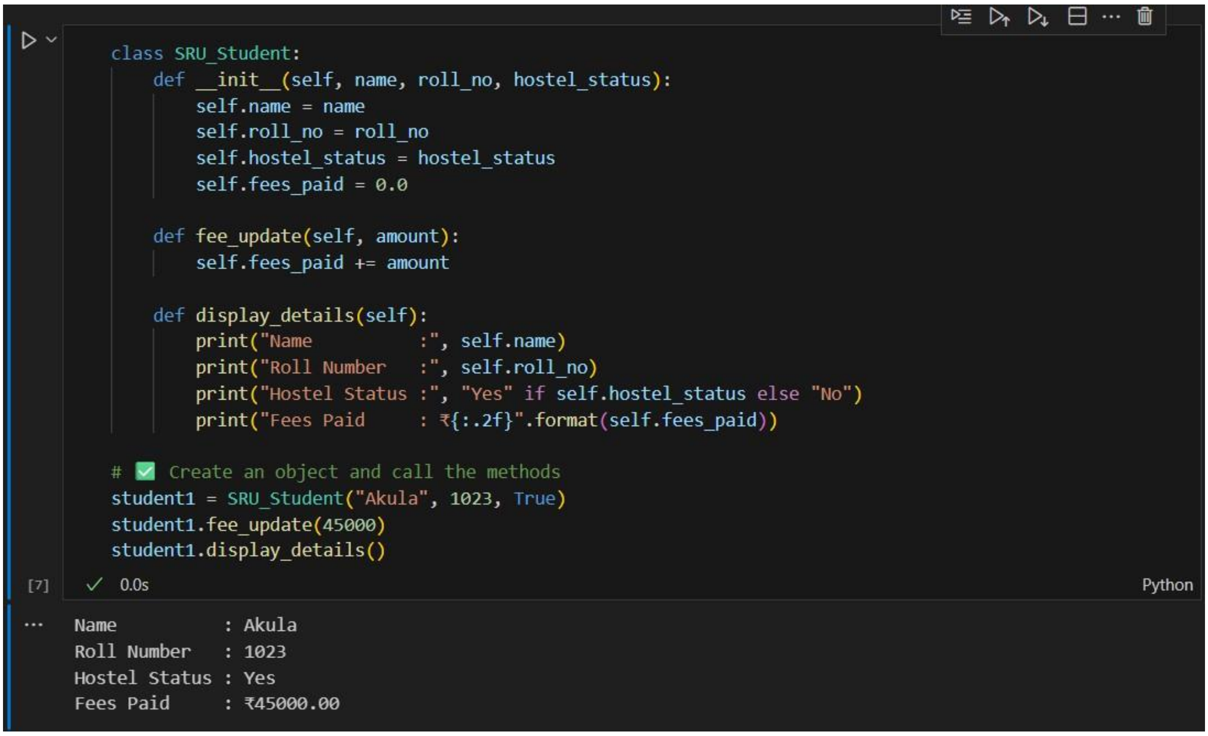
Task-2:

Prompt:



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

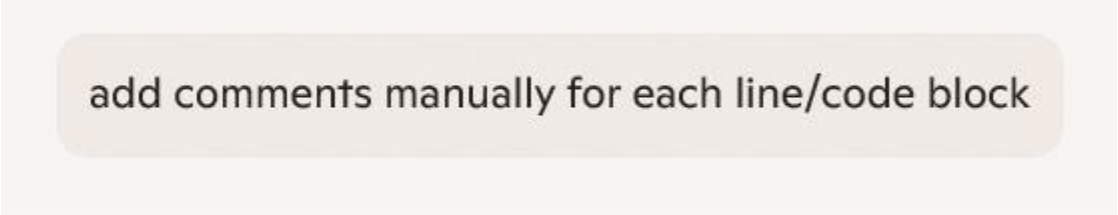
Code with output:



```python
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.fees_paid = 0.0

    def fee_update(self, amount):
        self.fees_paid += amount

    def display_details(self):
        print("Name          :", self.name)
        print("Roll Number   :", self.roll_no)
        print("Hostel Status :", "Yes" if self.hostel_status else "No")
        print("Fees Paid     : ₹{:.2f}".format(self.fees_paid))

# ✅ Create an object and call the methods
student1 = SRU_Student("Akula", 1023, True)
student1.fee_update(45000)
student1.display_details()
```

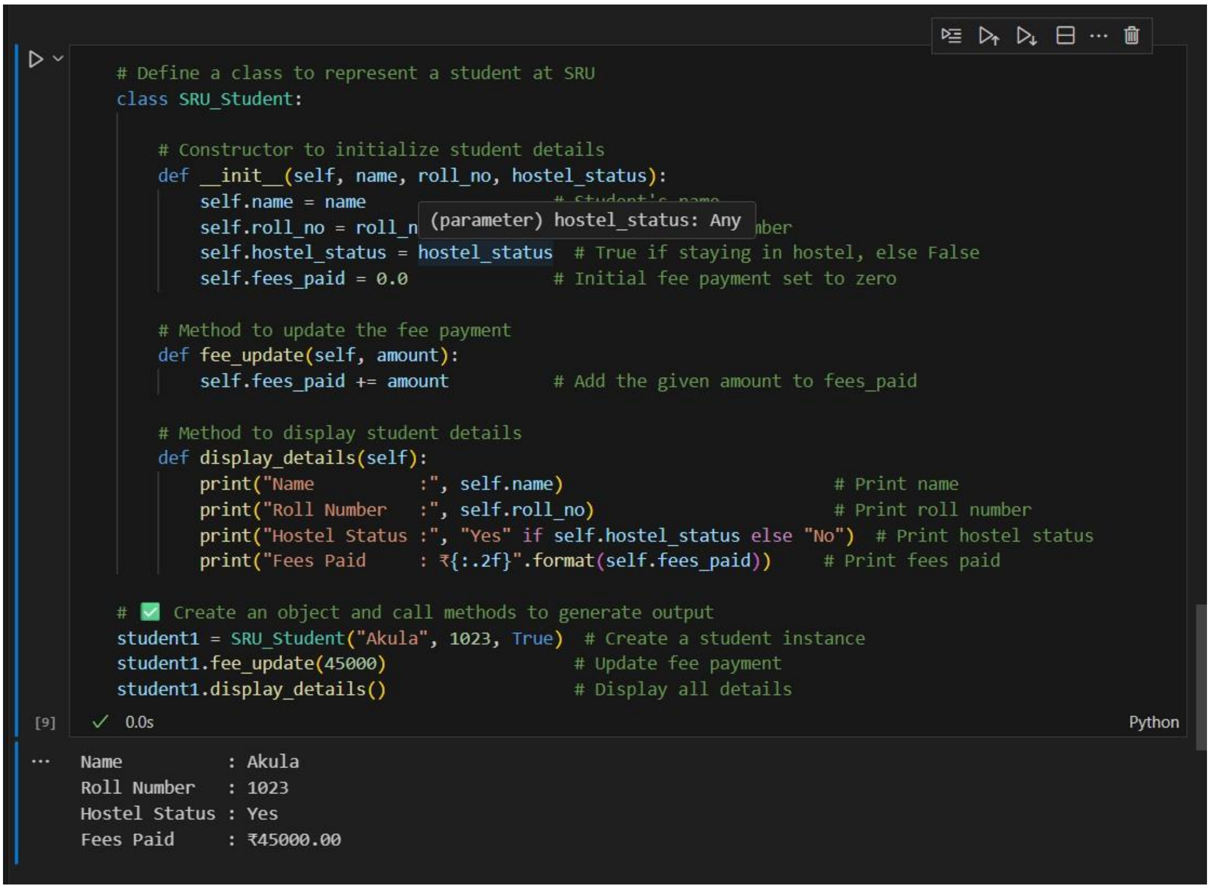
```
Name          : Akula
Roll Number   : 1023
Hostel Status : Yes
Fees Paid     : ₹45000.00
```

Prompt3:



add comments manually for each line/code block

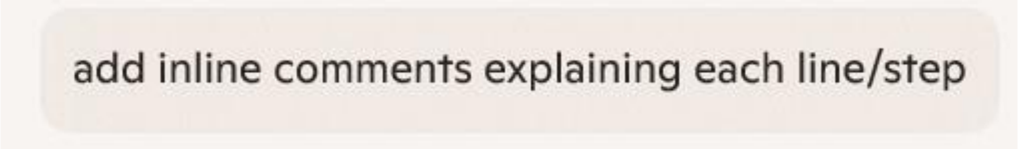
Code with output:



```python
# Define a class to represent a student at SRU
class SRU_Student:

    # Constructor to initialize student details
    def __init__(self, name, roll_no, hostel_status):
        self.name = name                  # Student's name
        self.roll_no = roll_n (parameter) hostel_status: Any mber
        self.hostel_status = hostel_status  # True if staying in hostel, else False
        self.fees_paid = 0.0              # Initial fee payment set to zero

    # Method to update the fee payment
    def fee_update(self, amount):
        self.fees_paid += amount          # Add the given amount to fees_paid

    # Method to display student details
    def display_details(self):
        print("Name          :", self.name)                           # Print name
        print("Roll Number   :", self.roll_no)                        # Print roll number
        print("Hostel Status :", "Yes" if self.hostel_status else "No")  # Print hostel status
        print("Fees Paid     : ₹{:.2f}".format(self.fees_paid))      # Print fees paid

# ✅ Create an object and call methods to generate output
student1 = SRU_Student("Akula", 1023, True)  # Create a student instance
student1.fee_update(45000)                  # Update fee payment
student1.display_details()                  # Display all details
```

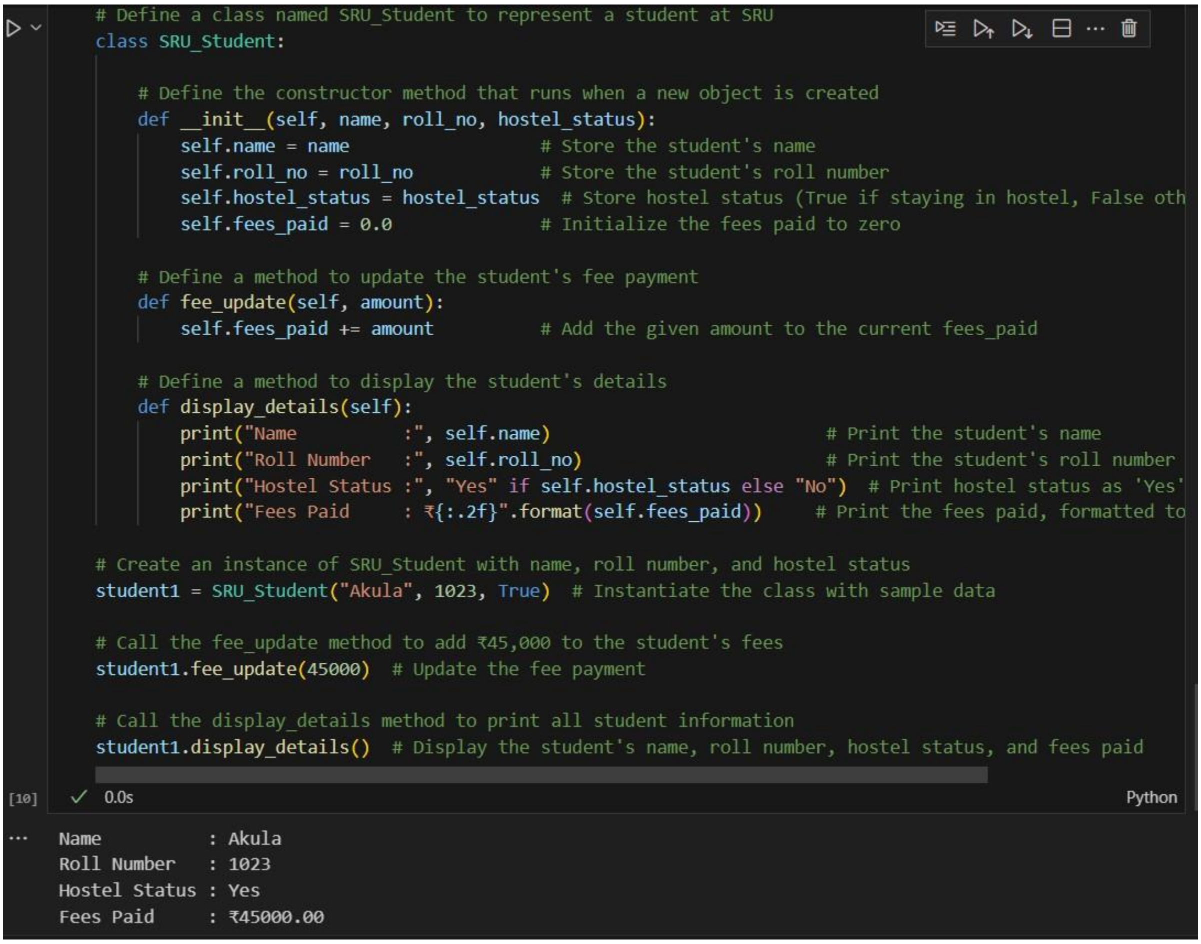
```
Name          : Akula
Roll Number   : 1023
Hostel Status : Yes
Fees Paid     : ₹45000.00
```

Prompt:



add inline comments explaining each line/step

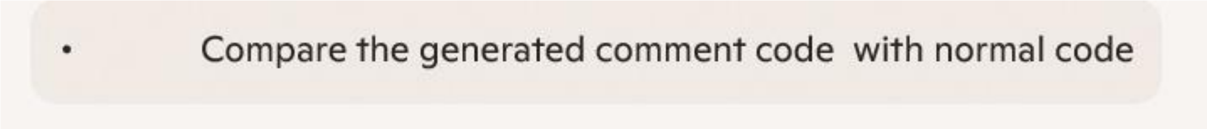
Code with output:



```python
# Define a class named SRU_Student to represent a student at SRU
class SRU_Student:

    # Define the constructor method that runs when a new object is created
    def __init__(self, name, roll_no, hostel_status):
        self.name = name                  # Store the student's name
        self.roll_no = roll_no            # Store the student's roll number
        self.hostel_status = hostel_status  # Store hostel status (True if staying in hostel, False oth
        self.fees_paid = 0.0              # Initialize the fees paid to zero

    # Define a method to update the student's fee payment
    def fee_update(self, amount):
        self.fees_paid += amount          # Add the given amount to the current fees_paid

    # Define a method to display the student's details
    def display_details(self):
        print("Name          :", self.name)                          # Print the student's name
        print("Roll Number   :", self.roll_no)                       # Print the student's roll number
        print("Hostel Status :", "Yes" if self.hostel_status else "No")  # Print hostel status as 'Yes'
        print("Fees Paid     : ₹{:.2f}".format(self.fees_paid))     # Print the fees paid, formatted to

# Create an instance of SRU_Student with name, roll number, and hostel status
student1 = SRU_Student("Akula", 1023, True)  # Instantiate the class with sample data

# Call the fee_update method to add ₹45,000 to the student's fees
student1.fee_update(45000)  # Update the fee payment

# Call the display_details method to print all student information
student1.display_details()  # Display the student's name, roll number, hostel status, and fees paid
```

```
Name          : Akula
Roll Number   : 1023
Hostel Status : Yes
Fees Paid     : ₹45000.00
```

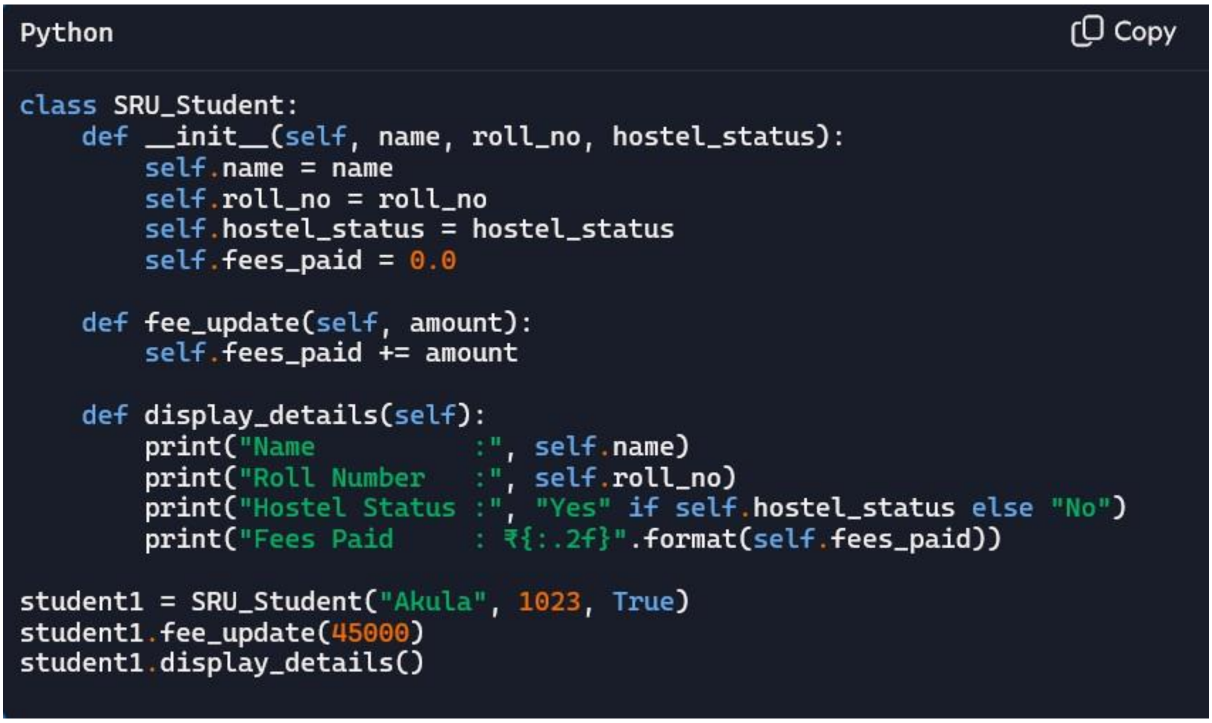
Prompt:



- Compare the generated comment code with normal code

Response:

1. Normal Code (Minimalist)

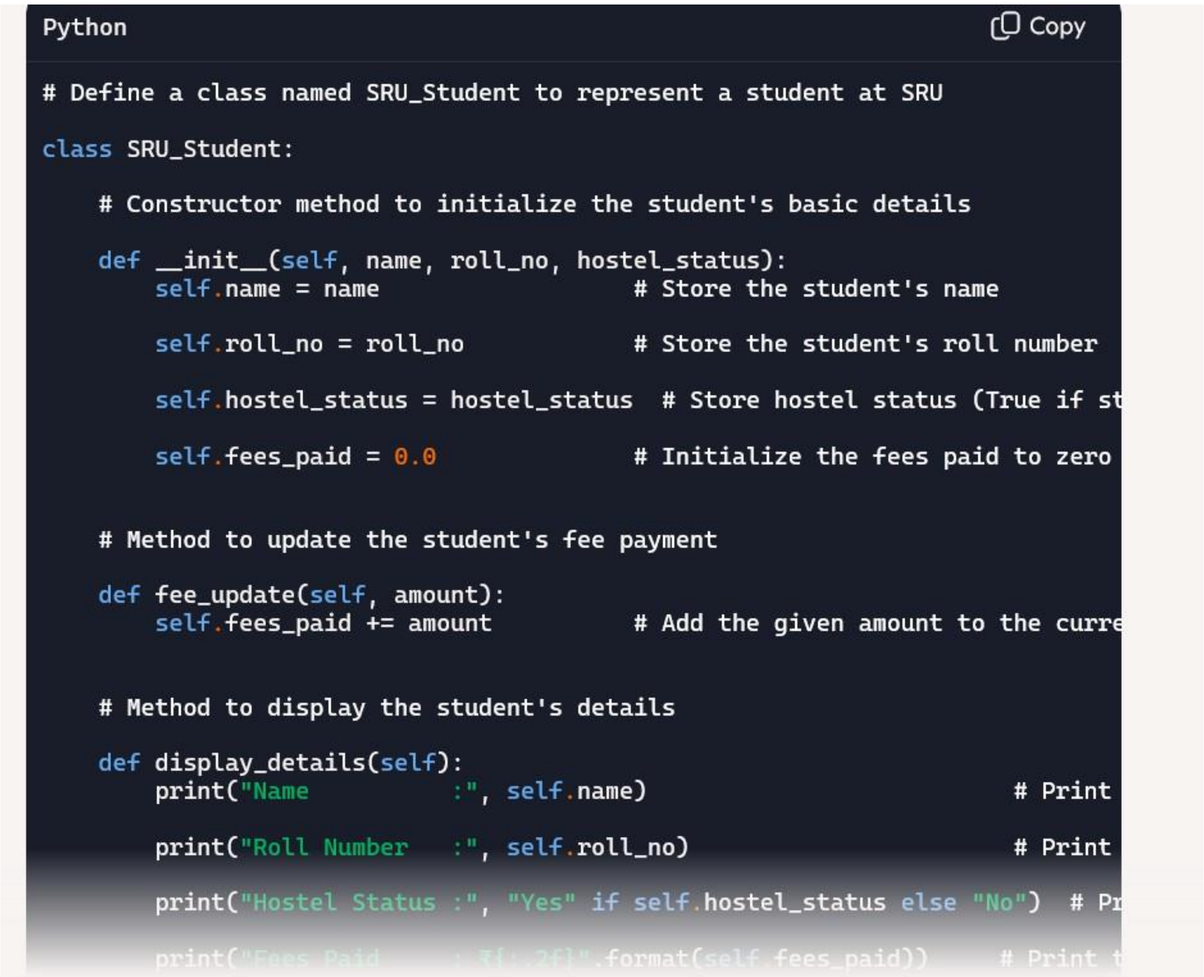


```python
class SRU_Student:
    def __init__(self, name, roll_no, hostel_status):
        self.name = name
        self.roll_no = roll_no
        self.hostel_status = hostel_status
        self.fees_paid = 0.0

    def fee_update(self, amount):
        self.fees_paid += amount

    def display_details(self):
        print("Name          :", self.name)
        print("Roll Number   :", self.roll_no)
        print("Hostel Status :", "Yes" if self.hostel_status else "No")
        print("Fees Paid     : ₹{:.2f}".format(self.fees_paid))

student1 = SRU_Student("Akula", 1023, True)
student1.fee_update(45000)
student1.display_details()
```

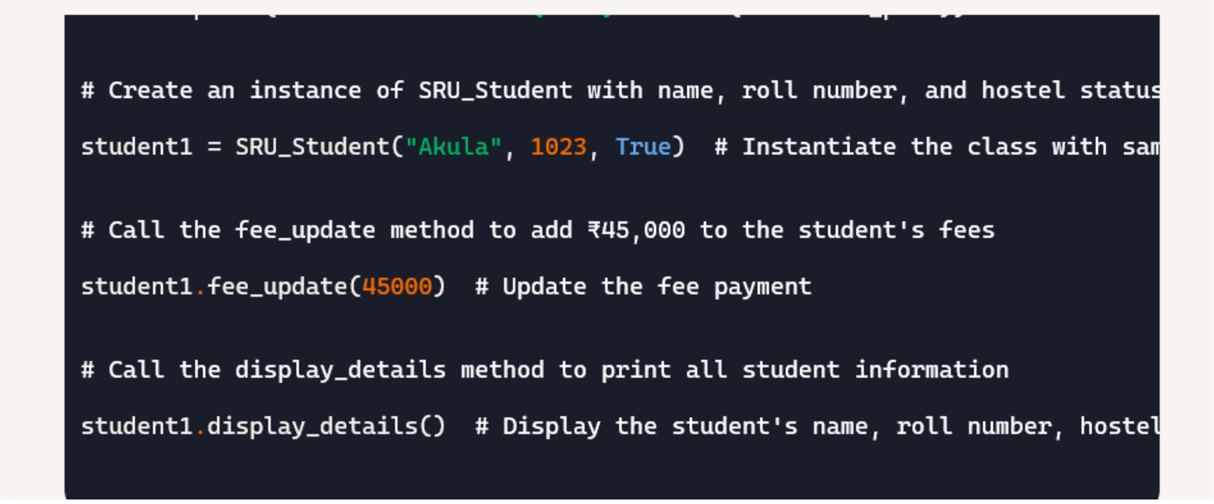
Characteristics:

- 🟩 Clean and concise
- + No explanation of logic or purpose
- + Not beginner-friendly
- 🟩 Ideal for experienced developers who already understand the structure

2. Fully Commented Code (With Inline Comments)



```python
# Define a class named SRU_Student to represent a student at SRU

class SRU_Student:

    # Constructor method to initialize the student's basic details

    def __init__(self, name, roll_no, hostel_status):
        self.name = name                  # Store the student's name

        self.roll_no = roll_no            # Store the student's roll number

        self.hostel_status = hostel_status  # Store hostel status (True if st

        self.fees_paid = 0.0              # Initialize the fees paid to zero

    # Method to update the student's fee payment

    def fee_update(self, amount):
        self.fees_paid += amount          # Add the given amount to the curre

    # Method to display the student's details

    def display_details(self):
        print("Name          :", self.name)                           # Print

        print("Roll Number   :", self.roll_no)                        # Print

        print("Hostel Status :", "Yes" if self.hostel_status else "No")  # Pr

        print("Fees Paid     : ₹{:.2f}".format(self.fees_paid))      # Print t
```



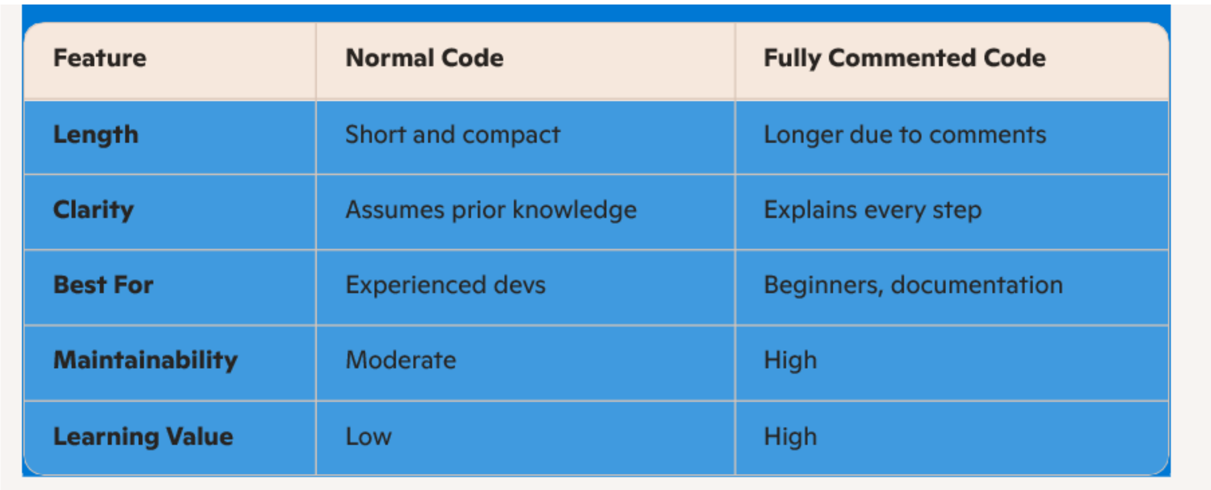
```python
# Create an instance of SRU_Student with name, roll number, and hostel status

student1 = SRU_Student("Akula", 1023, True)  # Instantiate the class with sam

# Call the fee_update method to add ₹45,000 to the student's fees

student1.fee_update(45000)  # Update the fee payment

# Call the display_details method to print all student information

student1.display_details()  # Display the student's name, roll number, hostel
```

Characteristics:

- 🟩 Every line is explained
- 🟩 Great for beginners or documentation
- 🟩 Improves readability and maintainability
- + Slightly longer and more verbose

- 🟩 Ideal for teaching, onboarding, or collaborative coding

Comparison Summary



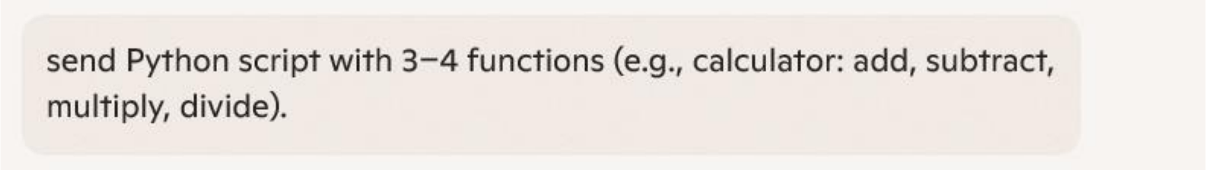
Feature	Normal Code	Fully Commented Code
Length	Short and compact	Longer due to comments
Clarity	Assumes prior knowledge	Explains every step
Best For	Experienced devs	Beginners, documentation
Maintainability	Moderate	High
Learning Value	Low	High

Final Takeaway:

- Use normal code for production or when working with experienced teams.
- Use fully commented code for teaching, debugging, or onboarding new developers.

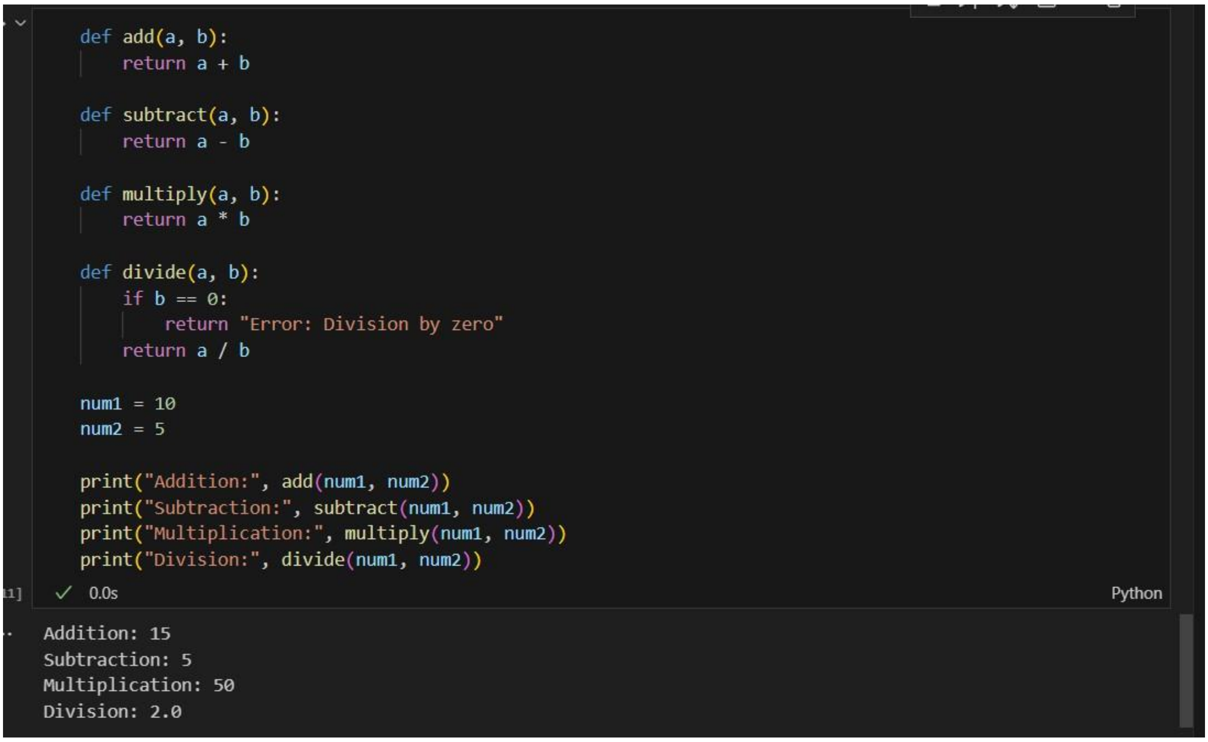
TASK-3:

Prompt:



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

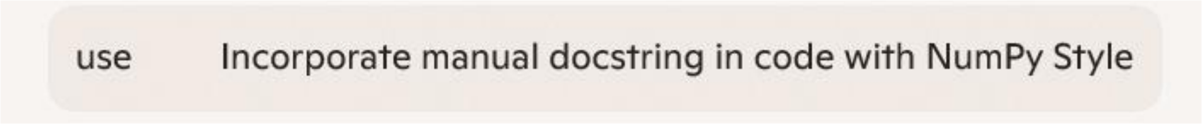
Code with output:



```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Error: Division by zero"
    return a / b

num1 = 10
num2 = 5

print("Addition:", add(num1, num2))
print("Subtraction:", subtract(num1, num2))
print("Multiplication:", multiply(num1, num2))
print("Division:", divide(num1, num2))
```

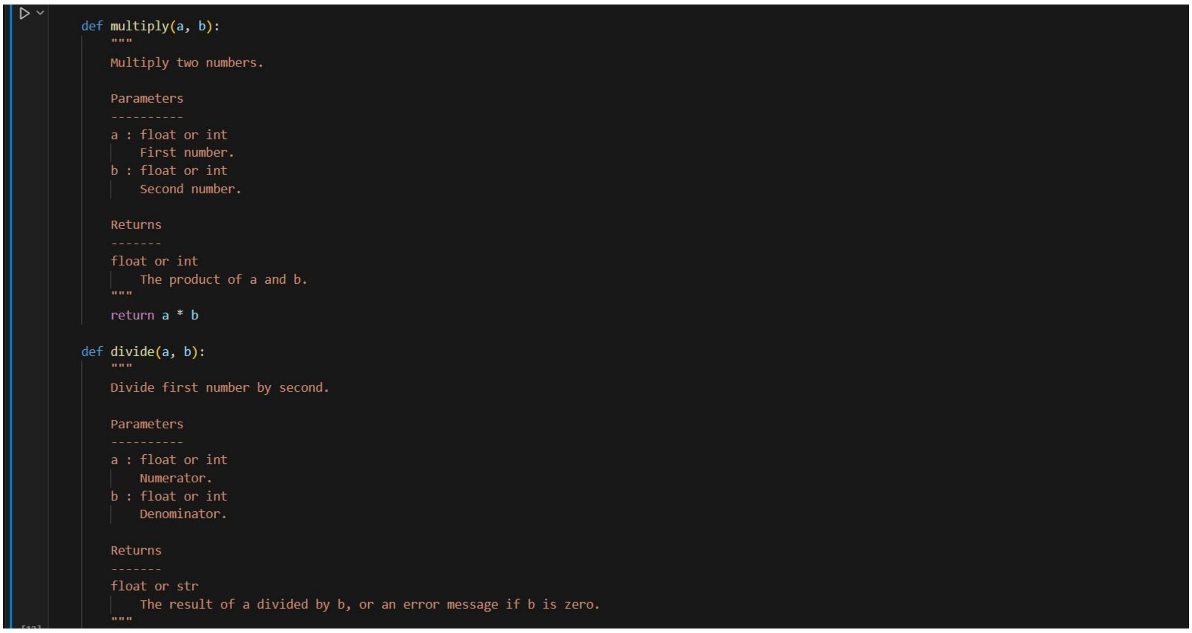
```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
```

Prompt:

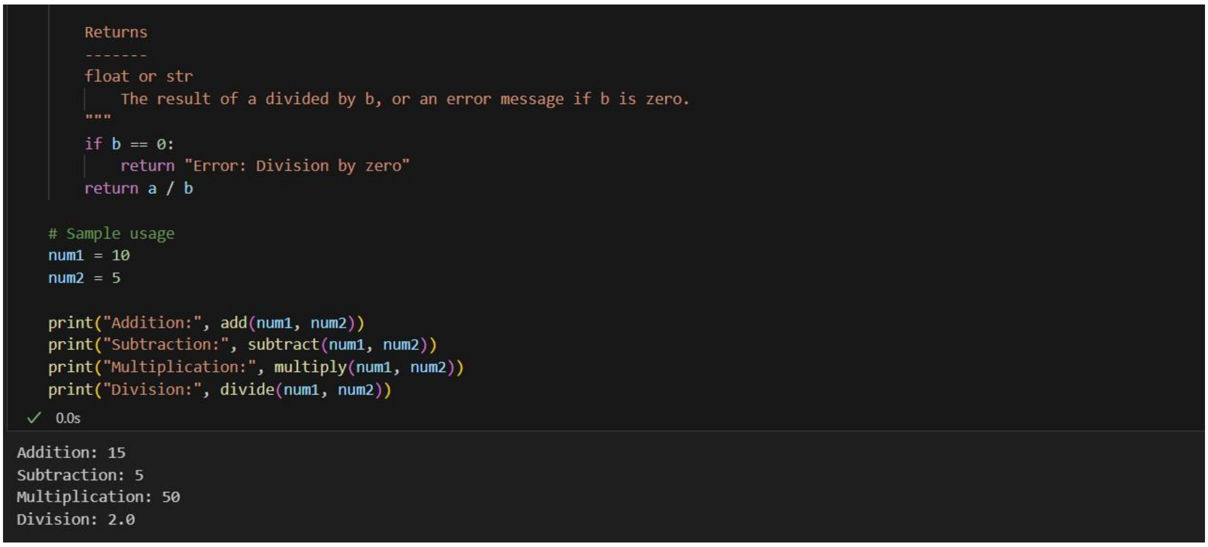


use Incorporate manual docstring in code with NumPy Style

Code with output:



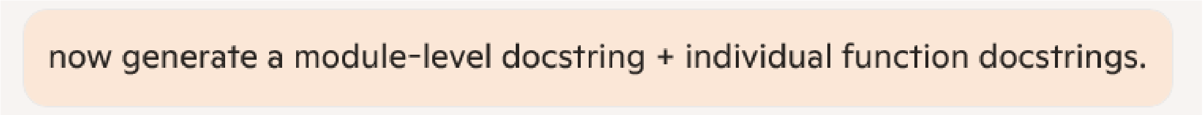
```
def multiply(a, b):
    """
    Multiply two numbers.

    Parameters
    ----------
    a : float or int
        First number.
    b : float or int
        Second number.

    Returns
    -------
    float or int
        The product of a and b.
    """
    return a * b

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

    Parameters
    ----------
    a : float or int
        Numerator.
    b : float or int
        Denominator.

    Returns
    -------
    float or str
        The result of a divided by b, or an error message if b is zero.
    """
```



```
    Returns
    -------
    float or str
        The result of a divided by b, or an error message if b is zero.
    """
    if b == 0:
        return "Error: Division by zero"
    return a / b

# Sample usage
num1 = 10
num2 = 5

print("Addition:", add(num1, num2))
print("Subtraction:", subtract(num1, num2))
print("Multiplication:", multiply(num1, num2))
print("Division:", divide(num1, num2))
```

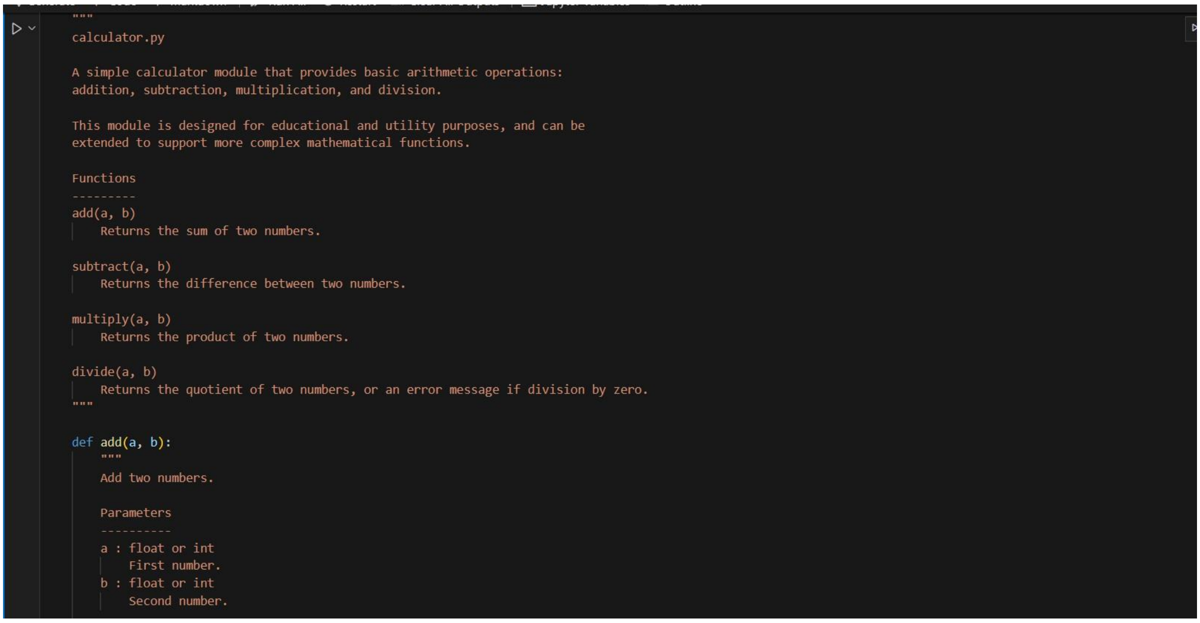
```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
```

Prompt:

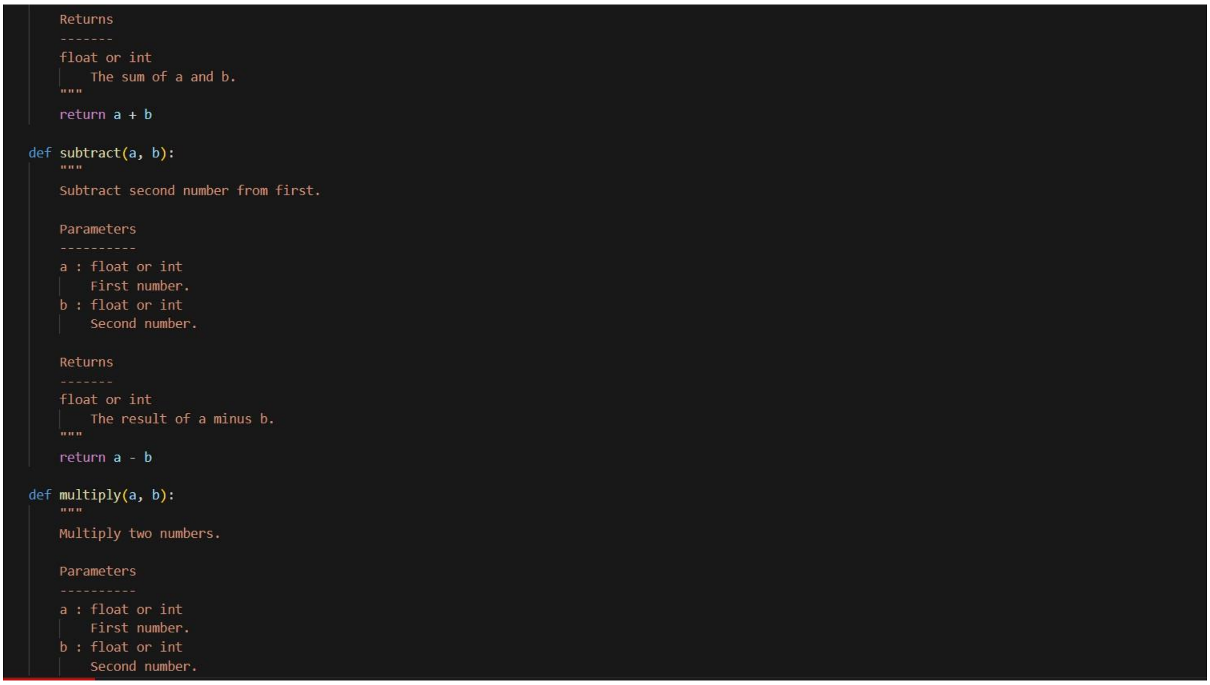


now generate a module-level docstring + individual function docstrings.

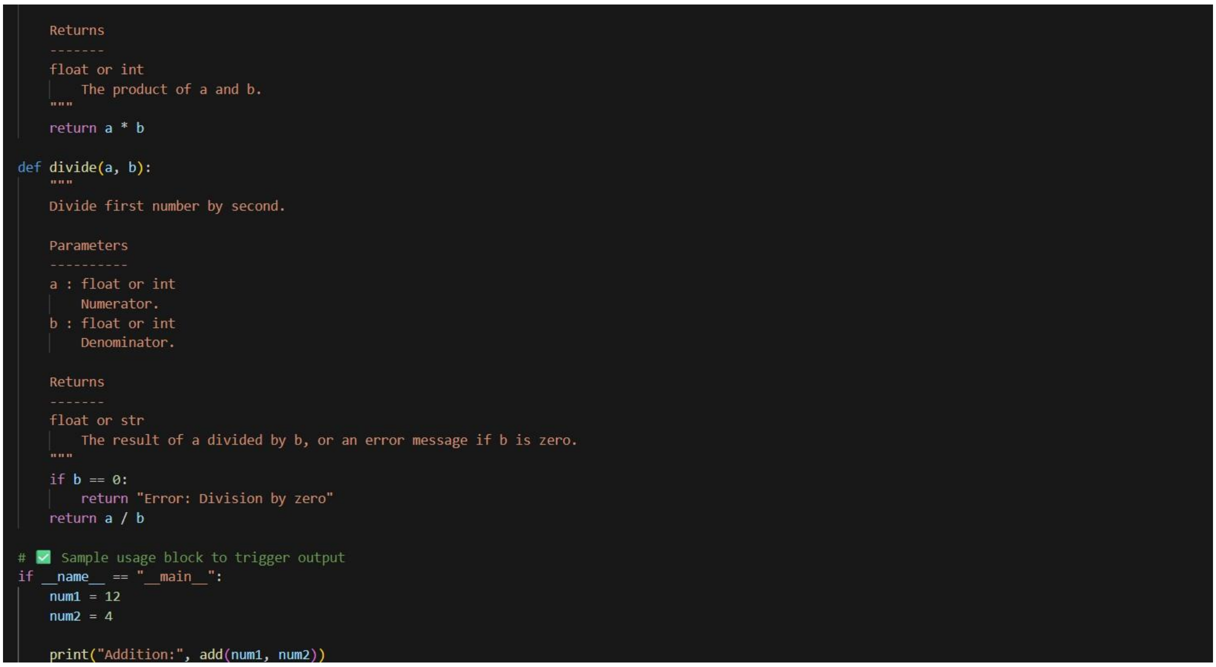
Code with output:



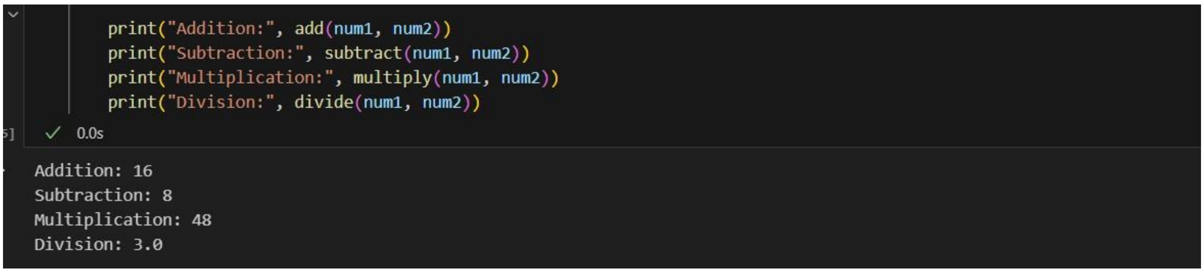
```
"""
calculator.py

A simple calculator module that provides basic arithmetic operations:
addition, subtraction, multiplication, and division.

This module is designed for educational and utility purposes, and can be
extended to support more complex mathematical functions.

Functions
---------
add(a, b)
    Returns the sum of two numbers.

subtract(a, b)
    Returns the difference between two numbers.

multiply(a, b)
    Returns the product of two numbers.

divide(a, b)
    Returns the quotient of two numbers, or an error message if division by zero.
"""

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

    Parameters
    ----------
    a : float or int
        First number.
    b : float or int
        Second number.
```



```
    Returns
    -------
    float or int
        The sum of a and b.
    """
    return a + b

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

    Parameters
    ----------
    a : float or int
        First number.
    b : float or int
        Second number.

    Returns
    -------
    float or int
        The result of a minus b.
    """
    return a - b

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

    Parameters
    ----------
    a : float or int
        First number.
    b : float or int
        Second number.
```



```
    Returns
    -------
    float or int
        The product of a and b.
    """
    return a * b

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

    Parameters
    ----------
    a : float or int
        Numerator.
    b : float or int
        Denominator.

    Returns
    -------
    float or str
        The result of a divided by b, or an error message if b is zero.
    """
    if b == 0:
        return "Error: Division by zero"
    return a / b

# ✅ Sample usage block to trigger output
if __name__ == "__main__":
    num1 = 12
    num2 = 4

    print("Addition:", add(num1, num2))
```

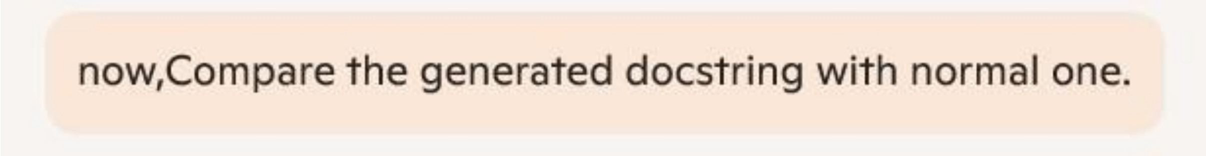


```
    print("Addition:", add(num1, num2))
    print("Subtraction:", subtract(num1, num2))
    print("Multiplication:", multiply(num1, num2))
    print("Division:", divide(num1, num2))
```

5] ✓ 0.0s

```
Addition: 16
Subtraction: 8
Multiplication: 48
Division: 3.0
```

Prompt:



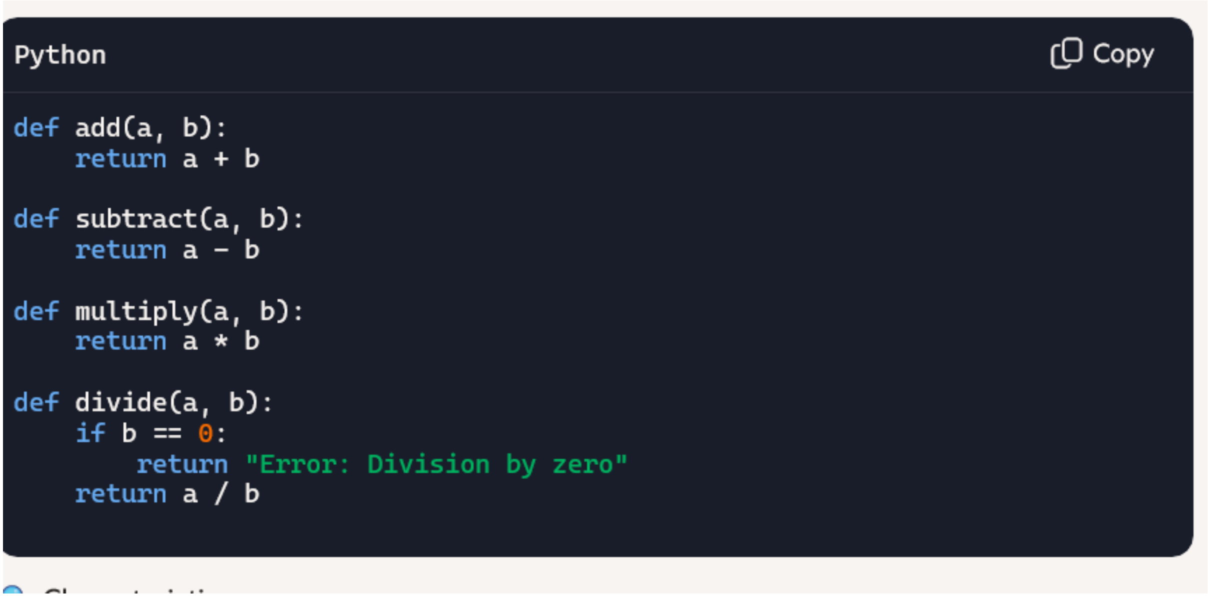
now,Compare the generated docstring with normal one.

Response:

Absolutely, Akula! Let's compare the **generated NumPy-style docstrings** with a **normal (non-docstring) version** of the same calculator code. This will help you see how documentation elevates clarity, usability, and professionalism—especially in collaborative or production environments.

Comparison: Docstring vs. Normal Code

🔷 1. Normal Code (No Docstrings)

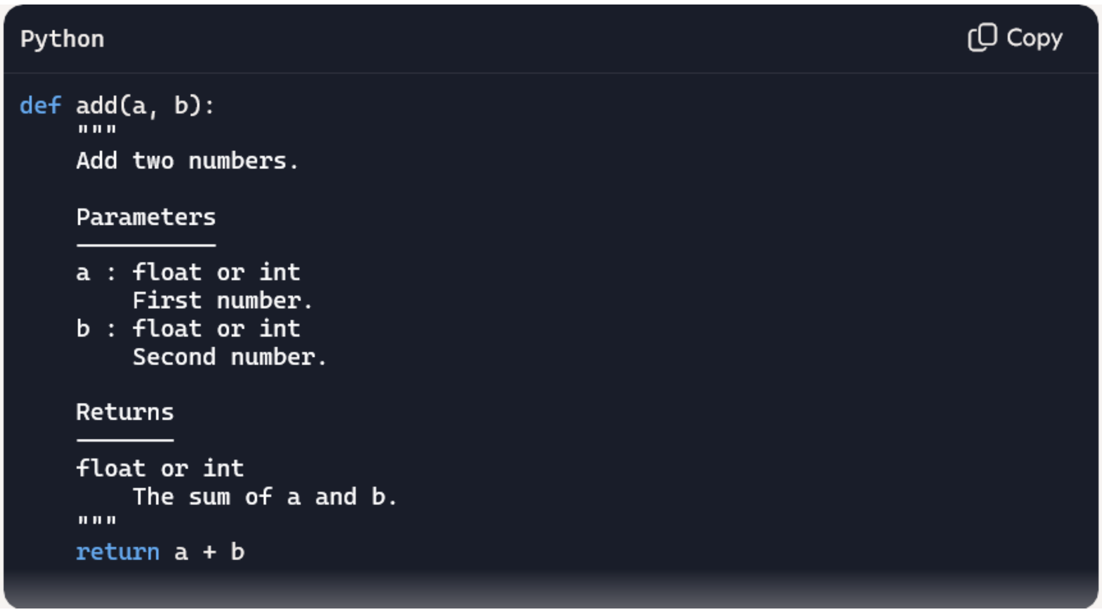


```python
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Error: Division by zero"
    return a / b
```

🔍 Characteristics:

- 🟩 Functional and concise
- + No explanation of what each function does
- + No parameter or return type guidance
- + No context for usage or edge cases
- 🟩 Good for quick scripts or experienced developers

🔷 2. Docstring Version (NumPy Style)



```python
def add(a, b):
    """
    Add two numbers.

    Parameters
    ----------
    a : float or int
        First number.
    b : float or int
        Second number.

    Returns
    -------
    float or int
        The sum of a and b.
    """
    return a + b
```

Characteristics:

- 🟩 Describes function purpose clearly
- 🟩 Lists parameters with types and descriptions
- 🟩 Specifies return type and behavior
- 🟩 Easier to integrate with IDEs, documentation tools, and linters
- 🟩 Ideal for teaching, collaboration, and scaling projects

Summary Table

Feature	Normal Code	NumPy-Style Docstring Version
Functionality	🟩 Works correctly	🟩 Works correctly
Clarity	+ Minimal	🟩 Detailed and structured
Parameter Guidance	+ None	🟩 Explicit types and roles
Return Info	+ Implicit	🟩 Clearly stated
Best For	Quick scripts	Production, teaching, teams
Tool Compatibility	+ Limited	🟩 IDEs, Sphinx, linters

Final Takeaway:

- Use **normal code** for speed and simplicity.
- Use **docstring-enhanced code** when clarity, scale, or collaboration matter.