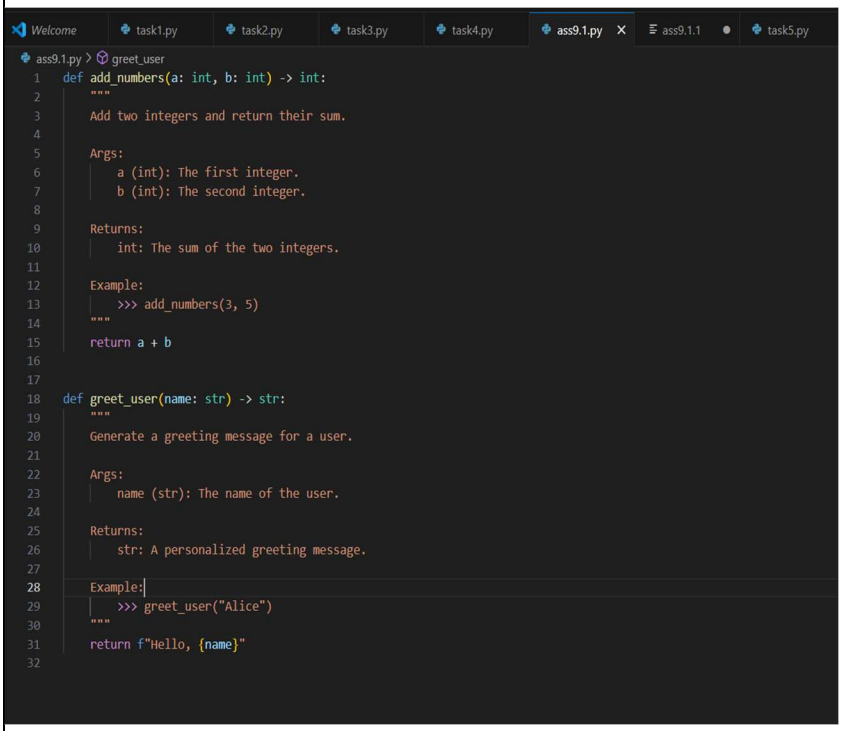
	AI ASSISTED CODING **NAME:J.KEERTHI PRIYA** **HALL TICKET NO:2403A510G4** **ASSIGNMENT:9.1**	
	Lab 9 – Documentation Generation: Automatic Documentation and Code Comments **Lab Objectives** • To use AI-assisted coding tools for generating Python documentation and code comments. • To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation. • To practice generating and refining docstrings, inline comments, and module-level documentation. • To compare outputs from different prompting styles for quality analysis.	
1	**Task Description #1** (Documentation – Google-Style Docstrings for Python Functions) • Task: Use AI to add Google-style docstrings to all functions in a given Python script. • Instructions: o Prompt AI to generate docstrings without providing any input-output examples. o Ensure each docstring includes: ▪ Function description ▪ Parameters with type hints ▪ Return values with type hints ▪ Example usage o Review the generated docstrings for accuracy and formatting. • Expected Output #1: o A Python script with all functions documented using correctly formatted Google-style docstrings. **PROMPT:** Add Google-style docstrings to all functions in the given Python script. Instructions: • Write a docstring for every function in the script using Google-style formatting. • Each docstring must include: o A clear function description o Parameters with type hints and explanations	

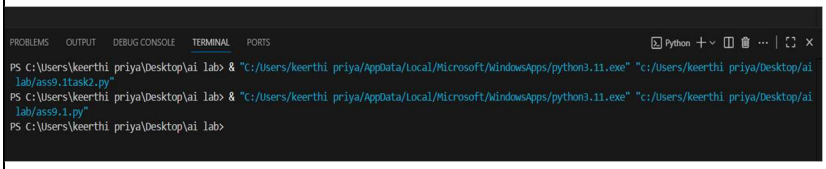
- Return values with type hints and explanations
- An example usage (only function call, no input-output results)

- Ensure the formatting and content are accurate, concise, and consistent

CODE :



OUTPUT:



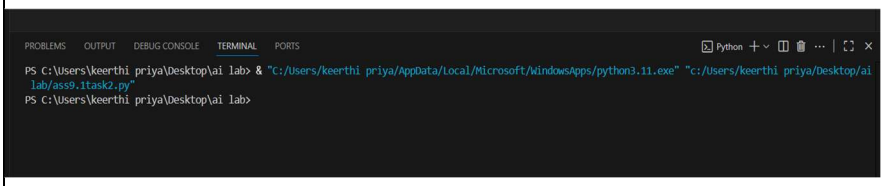
OBSERVATION:

1. Docstring Format (Google-style)
   - Both functions use Google-style docstrings correctly.
   - They include:
     - Function description ✔
     - Parameters with type hints ✔
     - Return values with type hints ✔
     - Example usage (only function call, no results) ✔
2. Clarity
   - The function descriptions are concise and clear.

	○ Parameters are well explained. ○ Return value descriptions are meaningful.	
	Task Description #2 (Documentation – Inline Comments for Complex Logic) • Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts. • Instructions: ○ Provide a Python script without comments to the AI. ○ Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections. ○ Verify that comments improve code readability and maintainability. • Expected Output #2: ○ Python code with concise, context-aware inline comments for complex logic blocks. **PROMPT:** Add meaningful inline comments to a Python program. **Instructions:** • Review the given Python script carefully. • Add inline comments **only for complex or non-intuitive logic parts** of the code. • Do **not** add comments for obvious syntax or simple operations (e.g., loops, variable assignments). • Ensure comments explain the **reasoning** behind tricky logic, not just what the code does. • Verify that your comments improve the overall **readability and maintainability** of the program. **CODE:**	

```python
ass9.1task2.py > ...
def find_second_largest(nums: list[int]) -> int:
    unique_nums = list(set(nums))
    unique_nums.sort(reverse=True)
    return unique_nums[1]

def find_second_largest(nums: list[int]) -> int:
    # Convert to set to remove duplicates, since repeated numbers
    # should not affect the "second largest" result
    unique_nums = list(set(nums))

    # Sort in descending order so the largest number comes first
    unique_nums.sort(reverse=True)

    # Return the second element, which is the second largest
    return unique_nums[1]

```

OUTPUT:



OBSERVATION:

1.Functionality

Both functions correctly return the second largest unique number from a list of integers.

Use of set(nums) removes duplicates, ensuring repeated numbers don't affect the result.

Sorting in descending order (reverse=True) ensures that index [1] corresponds to the second largest element.

2.Comparison of Versions

i)First version:

Works fine but lacks explanation.

A new reader may need time to figure out why set and sort(reverse=True) are used.

ii)Second version:

Adds inline comments explaining the reasoning behind key steps.

Improves readability and maintainability.

Makes it clear why d

Task Description #3 (Documentation – Module-Level Documentation)

- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
  - Supply the entire Python file to AI.
  - Instruct AI to write a single multi-line docstring at the top of the file.
  - Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
- Expected Output #3:
  - A complete, clear, and concise module-level docstring at the beginning of the file.

PROMPT:

Create a **module-level docstring** for the given Python file.

Instructions:

- Write a single multi-line docstring at the very top of the file.
- The docstring should summarize:
  - The **purpose** of the module (what the file is for).
  - Any important **dependencies** (standard libraries or third-party modules it relies on).
  - The **main functions and/or classes** defined in the file.
- Keep the explanation concise and clear.
- Do **not** rewrite the entire code or include unnecessary details.
- Ensure the docstring helps readers quickly understand the module's functionality and usage.

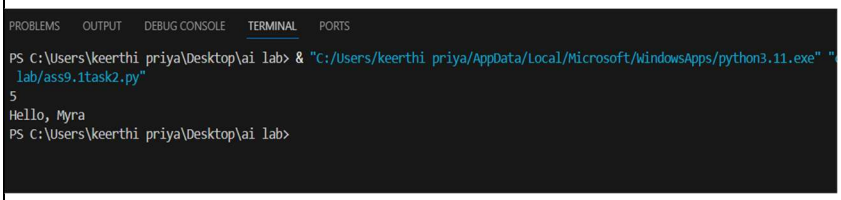
CODE:

```
"""
Module: utils

This module provides utility functions for basic mathematical operations
and user interaction. It is designed to demonstrate the use of
Google-style docstrings for functions and module-level documentation.

Dependencies:
    - No external dependencies (uses only Python standard library).

Main Functions:
    - add_numbers(a: int, b: int) -> int:
        Adds two integers and returns their sum.
    - greet_user(name: str) -> str:
        Generates a personalized greeting message.
    - find_second_largest(nums: list[int]) -> int:
        Finds and returns the second largest unique number in a list.

Usage:
    Import this module and call the functions directly.

    Example:
        from utils import add_numbers, greet_user, find_second_largest
"""

def add_numbers(a: int, b: int) -> int:
    """
    Add two integers and return their sum.

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The sum of the two integers.

    Example:
```

```
    Example:
        >>> add_numbers(3, 5)
    """
    return a + b


def greet_user(name: str) -> str:
    """
    Generate a greeting message for a user.

    Args:
        name (str): The name of the user.

    Returns:
        str: A personalized greeting message.

    Example:
        >>> greet_user("Alice")
    """
    return f"Hello, {name}"


def find_second_largest(nums: list[int]) -> int:
    """
    Find the second largest unique number in a list.

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

    Returns:
        int: The second largest unique number in the list.

    Example:
        >>> find_second_largest([1, 3, 5, 3, 4])
    """
    unique_nums = list(set(nums))
```

```python
def find_second_largest(nums: list[int]) -> int:
    """
    Find the second largest unique number in a list.

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

    Returns:
        int: The second largest unique number in the list.

    Example:
        >>> find_second_largest([1, 3, 5, 3, 4])
    """
    unique_nums = list(set(nums))
    unique_nums.sort(reverse=True)
    return unique_nums[1]
print(add_numbers(2,3))
print(greet_user('Myra'))
```

OUTPUT:



OBSERVATION:

1.Module-Level Docstring

Present at the very top of the file ✓

Clearly summarizes:

Purpose → explains the module provides utility functions for math and greetings.

Dependencies → correctly states that no external libraries are required.

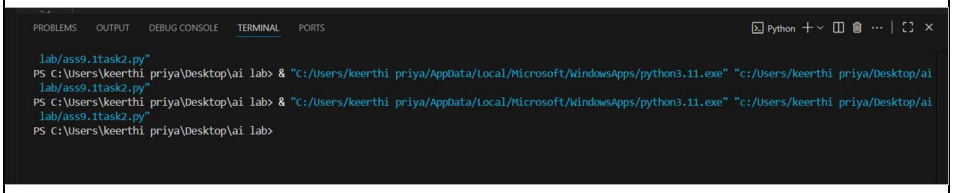
Main Functions → lists add_numbers, greet_user, and find_second_largest with short descriptions.

	Usage → shows how to import and use the functions. This makes the file self-explanatory for new readers.	

	Task Description #4 (Documentation – Convert Comments to Structured Docstrings)
	• Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
	• Instructions:
	  o Provide AI with Python code containing inline comments.
	  o Ask AI to move relevant details from comments into function docstrings.
	  o Verify that the new docstrings keep the meaning intact while improving structure.
	• Expected Output #4:
	  o Python code with comments replaced by clear, standardized docstrings.
	PROMPT:
	Transform existing inline comments into structured **Google-style function docstrings**.
	Instructions:
	• Review the Python file with inline comments.
	• Move the meaningful comments into the **function's docstring** instead of keeping them inline.
	• Ensure the new docstrings include:
	  o Function description
	  o Parameters with type hints
	  o Return values with type hints
	  o Example usage
	• Keep the intent of the original comments intact.
	• Verify that the new documentation is **clearer and more structured** than the inline comments.
	CODE:	

```python
def find_second_largest(nums: list[int]) -> int:
    # Convert to set to remove duplicates, since repeated numbers
    # should not affect the "second largest" result
    unique_nums = list(set(nums))

    # Sort in descending order so the largest number comes first
    unique_nums.sort(reverse=True)

    # Return the second element, which is the second largest
    return unique_nums[1]

def find_second_largest(nums: list[int]) -> int:
    """
    Find the second largest unique number in a list.

    This function removes duplicates, sorts the numbers
    in descending order, and returns the second element.

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

    Returns:
        int: The second largest unique number in the list.

    Example:
        >>> find_second_largest([10, 20, 20, 5, 15])
    """
    unique_nums = list(set(nums))
    unique_nums.sort(reverse=True)
    return unique_nums[1]
```

OUTPUT:



OBSERVATION:

- Inline Comments Removed
  - The inline comments in the input version explained each step separately.
  - In the output version, those details are consolidated into a structured Google-style docstring.
- Improved Readability
  - Instead of having scattered comments, the function has a single entry-point docstring.
  - Readers understand the purpose and logic without scanning multiple lines.

Task Description #5 (Documentation – Review and Correct Docstrings)

- Task: Use AI to identify and correct inaccuracies in existing docstrings.
- Instructions:
  - Provide Python code with outdated or incorrect docstrings.
  - Instruct AI to rewrite each docstring to match the current code behavior.
  - Ensure corrections follow Google-style formatting.
- Expected Output #5:
  - Python file with updated, accurate, and standardized docstrings.

PROMPT:

Identify and correct inaccuracies in existing Python function docstrings.
Instructions:

- Review the provided Python file with outdated or incorrect docstrings.
- Update each docstring so it accurately reflects the current code behavior.
- Ensure all updated docstrings follow Google-style formatting and include:
  - Function description
  - Parameters with type hints
  - Return values with type hints
  - Example usage
- Do not change the function's logic, only the docstrings.

CODE: (with inaccurate docstrings)

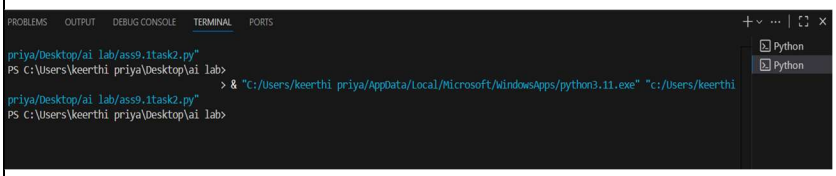
```python
def multiply_numbers(a: int, b: int) -> int:
    """
    Add two numbers and return their sum.   <-- INCORRECT
    """
    return a * b


def greet_user(name: str, age: int) -> str:
    """
    Returns the name of the user.   <-- OUTDATED, misses age
    """
    return f"Hello, {name}. You are {age} years old!"
```

CODE: (with corrected docstrings)

ass9.1task2.py > ...

```python
def multiply_numbers(a: int, b: int) -> int:
    """
    Multiply two integers and return their product.

    Args:
        a (int): The first integer.
        b (int): The second integer.

    Returns:
        int: The product of the two integers.

    Example:
        >>> multiply_numbers(4, 5)
    """
    return a * b


def greet_user(name: str, age: int) -> str:
    """
    Generate a personalized greeting message including the user's age.

    Args:
        name (str): The name of the user.
        age (int): The age of the user.

    Returns:
        str: A personalized greeting message containing the user's name and age.

    Example:
        >>> greet_user("Alice", 25)
    """
    return f"Hello, {name}. You are {age} years old!"

```

OUTPUT:



OBSERVATION:

Inline Comments Removed

The inline comments in the input version explained each step

	separately. In the output version, those details are consolidated into a structured Google-style docstring. Improved Readability Instead of having scattered comments, the function has a single entry-point docstring. Readers understand the purpose and logic without scanning multiple lines.	

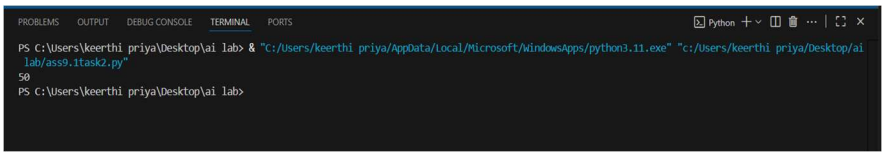
	Task Description #6 (Documentation – Prompt Comparison Experiment)
	• Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
	• Instructions:
	  o Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
	  o Use AI to process the same Python function with both prompts.
	  o Analyze and record differences in quality, accuracy, and completeness.
	• Expected Output #6:
	  o A comparison table showing the results from both prompts with observations.
	PROMPT:
	Vague Prompt Simulation:
	Pretend the user asked only:
	"Add comments to this function."
	Provide the output for this vague request.
	Detailed Prompt Simulation:
	Pretend the user asked instead:
	"Add Google-style docstrings with parameters, return types, and examples."
	Provide the output for this detailed request.
	Comparison Task:
	Create a comparison table showing the outputs from both prompts side	

by side. Include observations on **quality, accuracy, and completeness**.

CODE(VAGUE PROMPT):

```python
def calculate_area(length, width):
    return length * width
def calculate_area(length, width):
    # Multiply length and width to get area
    return length * width
def calculate_area(length, width):
    """Calculate the area of a rectangle.

    Args:
        length (float or int): The length of the rectangle.
        width (float or int): The width of the rectangle.

    Returns:
        float: The computed area of the rectangle.

    Example:
        >>> calculate_area(5, 10)
        50
    """
    return length * width
print(calculate_area(5,10))
```

OUTPUT:



OBSERVATION:

 A vague prompt gives minimal results (single comment, lacks structure).

 A detailed prompt produces high-quality, structured documentation (parameters, return types, examples).

 Detailed prompting is especially important in collaborative coding, large projects, and generating professional docs.