

ASSIGNMENT:9.2

HTNO:2403A51284

BATCH:12

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.

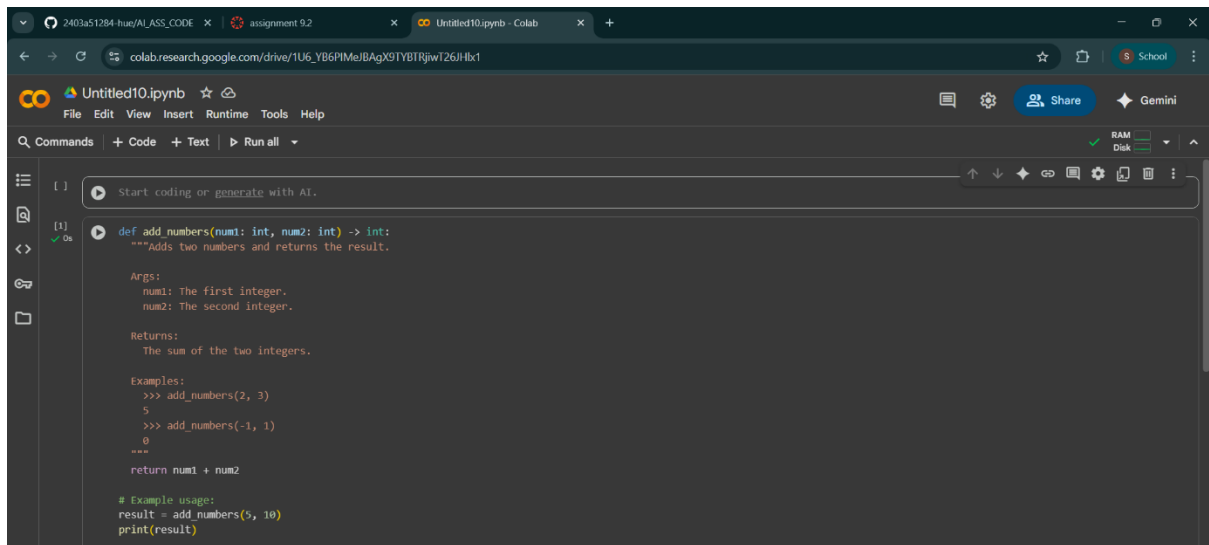
- Expected Output #1:

- o A Python script with all functions documented using correctly formatted Google-style docstrings

PROMPT:

- A brief function description
- Parameters with type hints
- Return values with type hints
- Example usage (without input-output examples)
Ensure the docstrings are accurate and correctly formatted.

CODE:



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code cell contains a Python function definition with extensive inline comments explaining its purpose, arguments, return value, and usage. The code is as follows:

```
[1] def add_numbers(num1: int, num2: int) -> int:
    """Adds two numbers and returns the result.

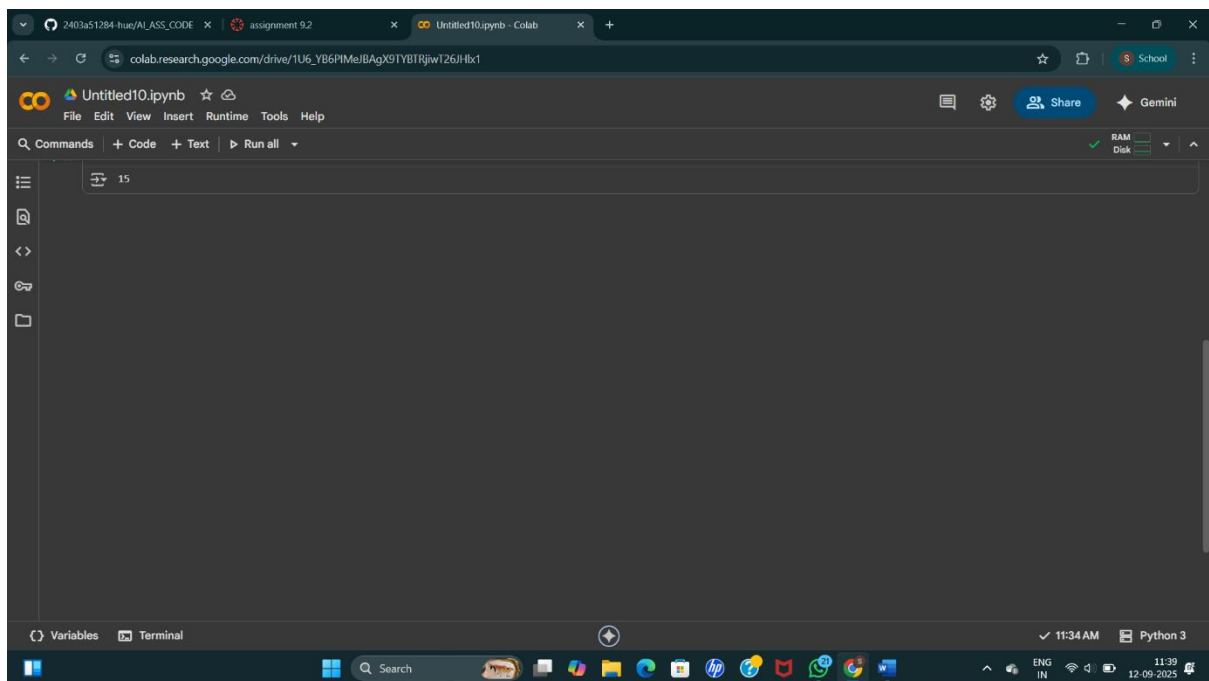
    Args:
        num1: The first integer.
        num2: The second integer.

    Returns:
        The sum of the two integers.

    Examples:
        >>> add_numbers(2, 3)
        5
        >>> add_numbers(-1, 1)
        0
    """
    return num1 + num2

# Example usage:
result = add_numbers(5, 10)
print(result)
```

OUTPUT:



The screenshot shows the same Google Colab notebook after execution. The output cell displays the result of the function call, which is 15. The interface shows the 'Run' button and the output area.

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:
 - o Provide a Python script without comments to the AI.
 - o Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
 - o Verify that comments improve code readability and maintainability.
- Expected Output #2:
 - o Python code with concise, context-aware inline comments for complex logic blocks

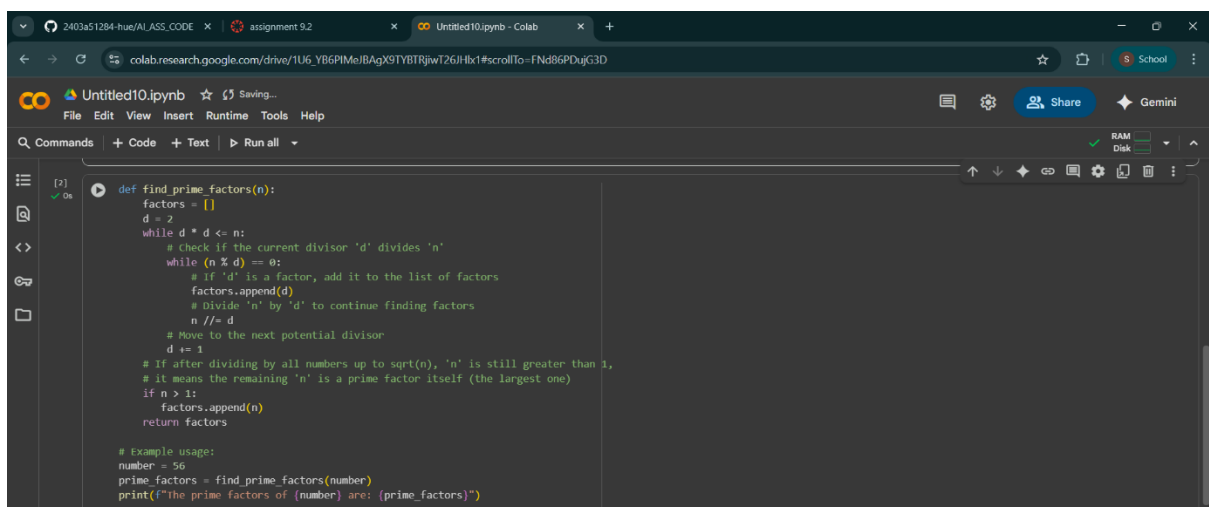
PROMPT:

Add meaningful inline comments to the following Python script, explaining only the complex or non-intuitive logic parts.

Skip comments for obvious syntax or straightforward code.

Focus on tricky sections to improve readability and maintainability

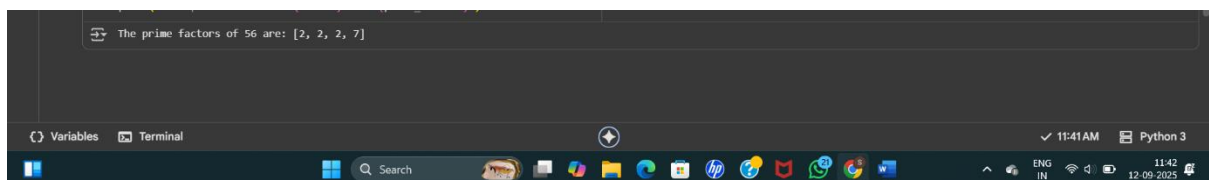
CODE:



```
def find_prime_factors(n):
    factors = []
    d = 2
    while d * d <= n:
        # check if the current divisor 'd' divides 'n'
        while (n % d) == 0:
            # if 'd' is a factor, add it to the list of factors
            factors.append(d)
            # Divide 'n' by 'd' to continue finding factors
            n //= d
        # Move to the next potential divisor
        d += 1
    # If after dividing by all numbers up to sqrt(n), 'n' is still greater than 1,
    # it means the remaining 'n' is a prime factor itself (the largest one)
    if n > 1:
        factors.append(n)
    return factors

# Example usage:
number = 56
prime_factors = find_prime_factors(number)
print(f"The prime factors of {number} are: {prime_factors}")
```

OUTPUT:



```
The prime factors of 56 are: [2, 2, 2, 7]
```

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:

- o Supply the entire Python file to AI.
- o Instruct AI to write a single multi-line docstring at the top of the file.
- o Ensure the docstring clearly describes functionality and usage without rewriting the entire code.

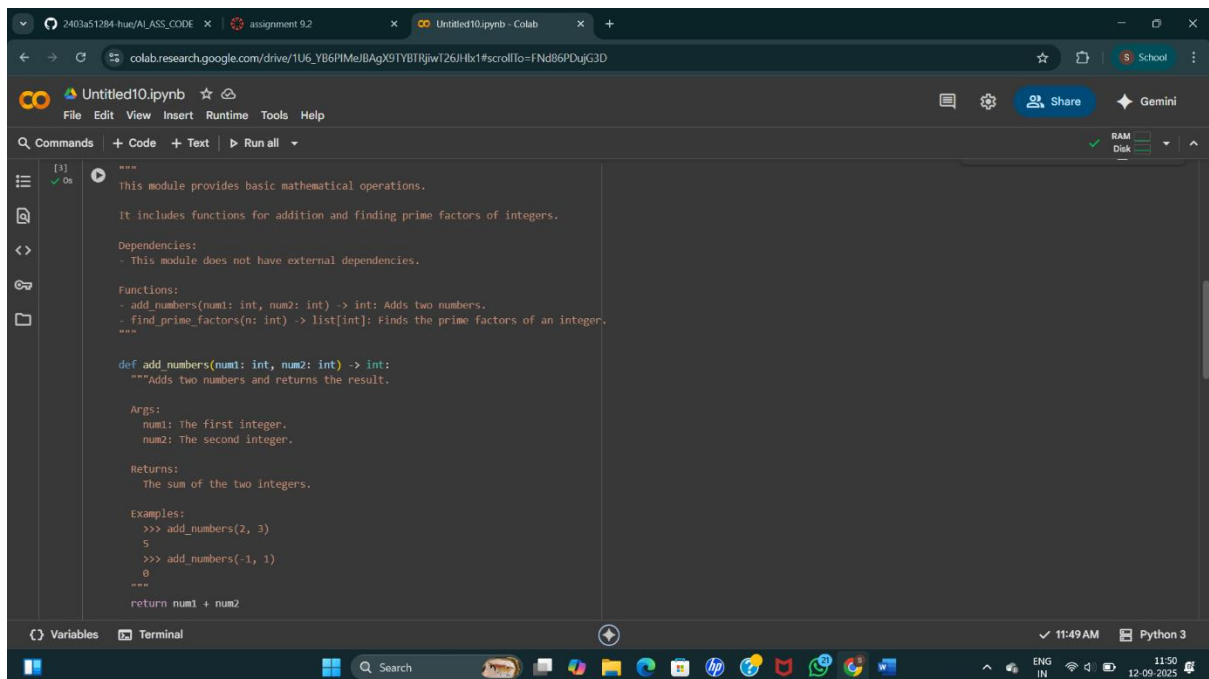
• Expected Output #3:

- o A complete, clear, and concise module-level docstring at the beginning of the file

PROMPT:

Write a single multi-line module-level docstring at the top of the following Python file. The docstring should summarize the file's purpose, dependencies, and main functions/classes. Do not rewrite the code; just provide a clear, concise description of functionality and usage.

CODE:



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code cell contains a multi-line docstring at the top, followed by a function definition. The docstring describes the module's purpose, dependencies, functions, arguments, returns, and examples. The function 'add_numbers' is defined with two parameters and returns their sum.

```
"""
This module provides basic mathematical operations.

It includes functions for addition and finding prime factors of integers.

Dependencies:
- This module does not have external dependencies.

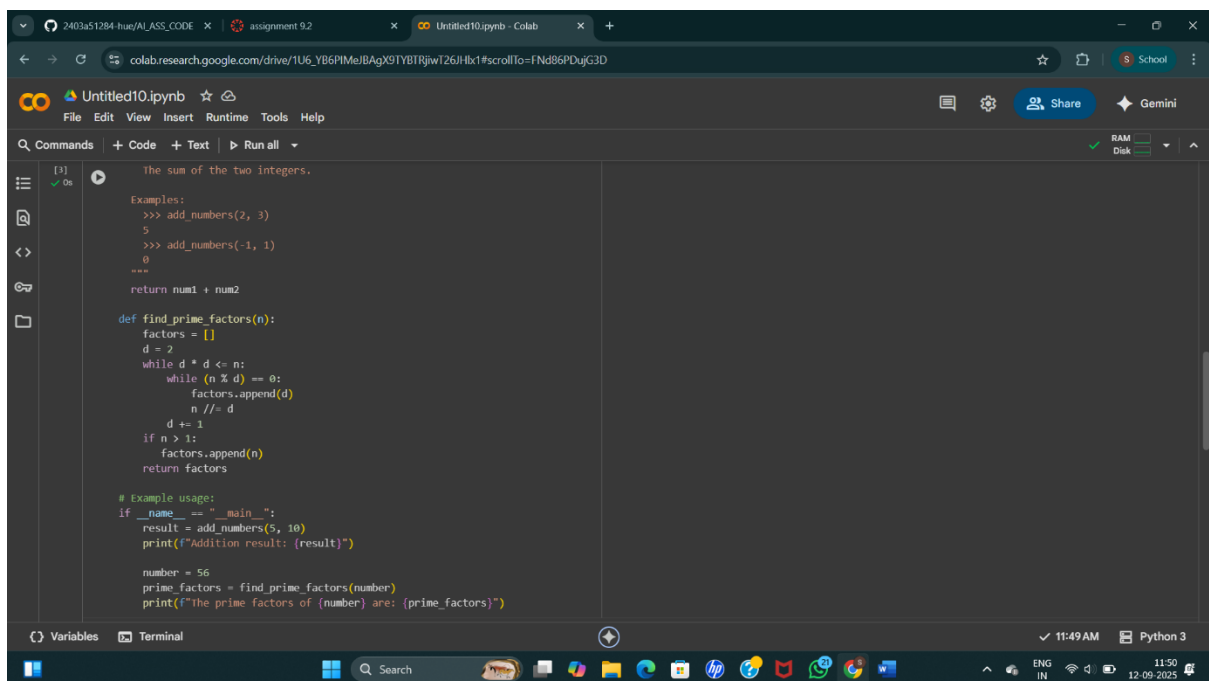
Functions:
- add_numbers(num1: int, num2: int) -> int: Adds two numbers.
- find_prime_factors(n: int) -> list[int]: Finds the prime factors of an integer.
"""

def add_numbers(num1: int, num2: int) -> int:
    """Adds two numbers and returns the result.

    Args:
        num1: The first integer.
        num2: The second integer.

    Returns:
        The sum of the two integers.

    Examples:
        >>> add_numbers(2, 3)
        5
        >>> add_numbers(-1, 1)
        0
    """
    return num1 + num2
```



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code cell contains a multi-line docstring at the top, followed by two function definitions and a main block. The docstring describes the module's purpose, dependencies, functions, arguments, returns, and examples. The function 'find_prime_factors' is defined with one parameter and returns a list of prime factors. The main block demonstrates the usage of both functions.

```
"""
The sum of the two integers.


Examples:
>>> add_numbers(2, 3)
5
>>> add_numbers(-1, 1)
0
"""
return num1 + num2

def find_prime_factors(n):
    factors = []
    d = 2
    while d * d <= n:
        while (n % d) == 0:
            factors.append(d)
            n //= d
        d += 1
    if n > 1:
        factors.append(n)
    return factors

# Example usage:
if __name__ == "__main__":
    result = add_numbers(5, 10)
    print(f"Addition result: {result}")

    number = 56
    prime_factors = find_prime_factors(number)
    print(f"The prime factors of {number} are: {prime_factors}")
```

OUTPUT:

A terminal window with a dark background. On the left, there is a small icon of a document and a prompt character '<>'. To the right of the prompt, the output text is displayed in two lines: 'Addition result: 15' and 'The prime factors of 56 are: [2, 2, 2, 7]'.

```
<> Addition result: 15
The prime factors of 56 are: [2, 2, 2, 7]
```

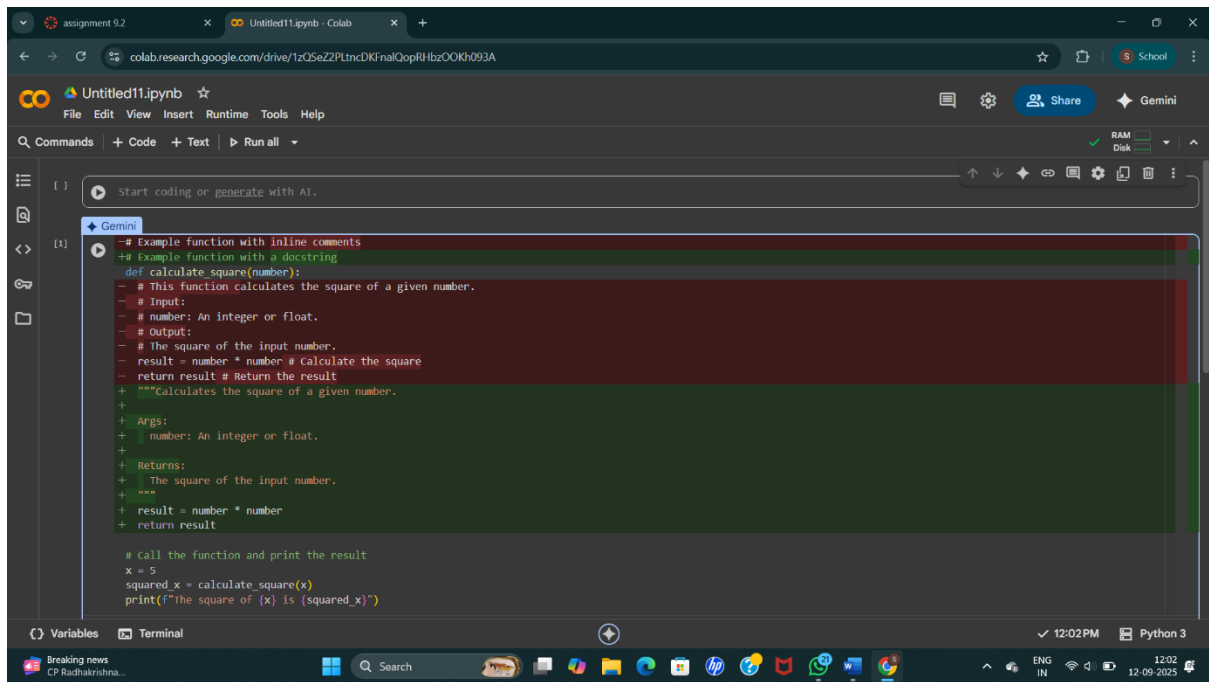
Task Description #4 (Documentation – Convert Comments to Structured Docstrings)

- 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 all existing inline comments in the following Python code into structured function docstrings using Google style.
Move relevant details from comments into the docstrings, ensuring the meaning is preserved and the documentation is clear and standardized.

CODE:



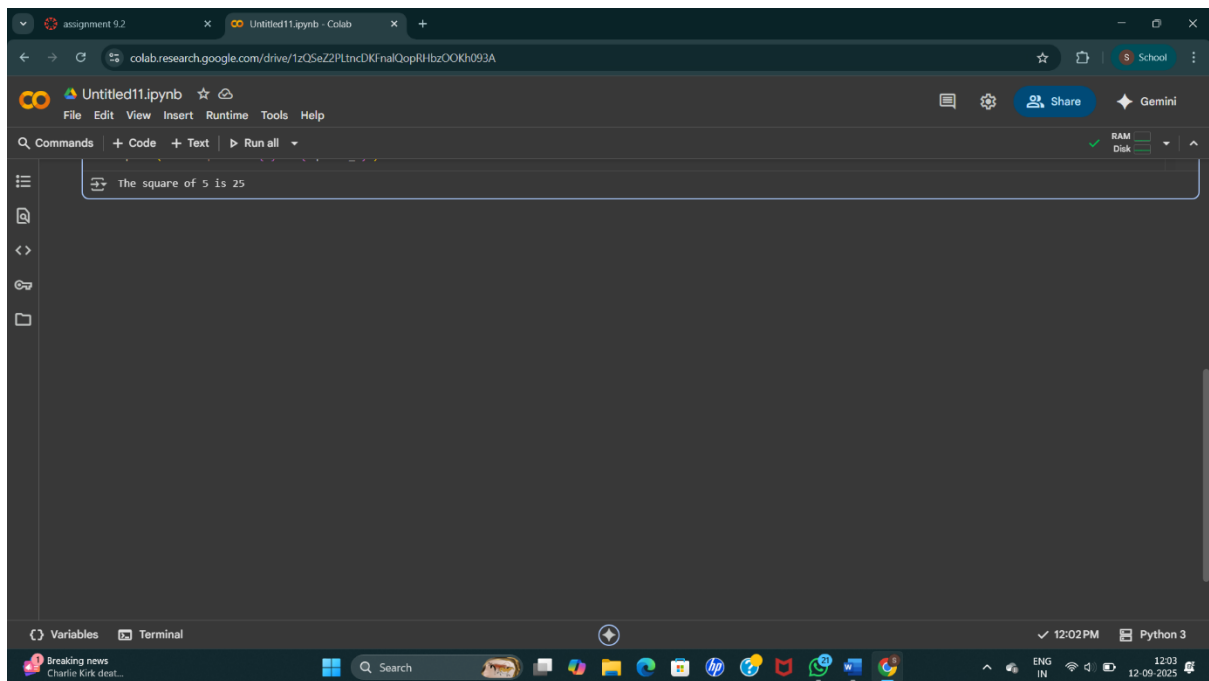
```
[1] Start coding or generate with AI.

+ Gemini

[1]
-# Example function with inline comments
-# Example function with a docstring
def calculate_square(number):
-# This function calculates the square of a given number.
-# Input:
-# number: An integer or float.
-# Output:
-# The square of the input number.
- result = number * number # Calculate the square
- return result # Return the result
- """Calculates the square of a given number.
-
- Args:
-     number: An integer or float.
-
- Returns:
-     The square of the input number.
- """
+ result = number * number
+ return result

# Call the function and print the result
x = 5
squared_x = calculate_square(x)
print(f"The square of {x} is {squared_x}")
```

OUTPUT:



```
The square of 5 is 25
```

- o Python file with updated, accurate, and standardized Task Description #5 (Documentation – Review and Correct Docstrings)
- Task:5 Use AI to identify and correct inaccuracies in existing docstrings.
- Instructions:
 - o Provide Python code with outdated or incorrect docstrings.
 - o Instruct AI to rewrite each docstring to match the current code behavior.

o Ensure corrections follow Google-style formatting.

• Expected Output #5:

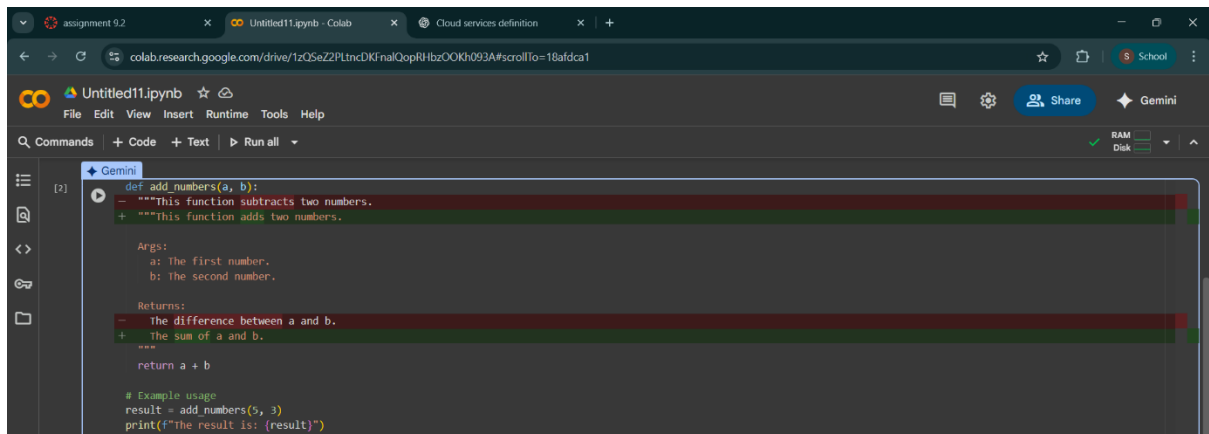
docstrings

PROMPT:

Review the following Python code and identify any outdated or incorrect docstrings.

Rewrite each docstring to accurately reflect the current code behavior, using Google-style formatting for all corrections.

CODE:



```
def add_numbers(a, b):  
    """This function subtracts two numbers.  
    + """This function adds two numbers.  
  
    Args:  
        a: The first number.  
        b: The second number.  
  
    Returns:  
    - The difference between a and b.  
    + The sum of a and b.  
    """  
    return a + b  
  
# Example usage  
result = add_numbers(5, 3)  
print(f"The result is: {result}")
```

OUTPUT:



```
The result is: 8
```

Description #6 (Documentation – Prompt Comparison

Task (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:

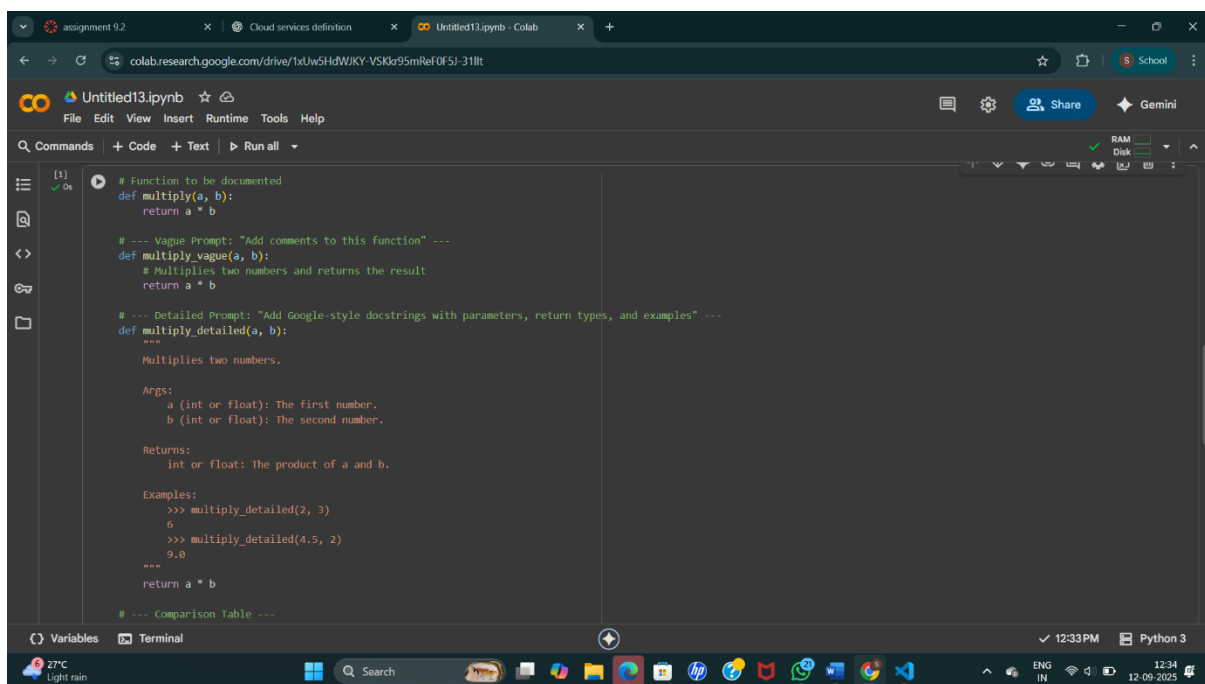
Prompt 1 (Vague):

Add comments to this function.

Prompt 2 (Detailed):

Add Google-style docstrings to this function, including a brief description, parameter types, return type, and example usage.

CODE:



The screenshot shows a Google Colab notebook titled 'Untitled13.ipynb'. The code defines three functions: `multiply(a, b)`, `multiply_vague(a, b)`, and `multiply_detailed(a, b)`. The `multiply_detailed` function includes a Google-style docstring with a description, arguments, returns, and examples. The notebook interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for running, saving, and sharing, and a status bar at the bottom showing the current time and Python version.

```
[1] ✓ 0s
def multiply(a, b):
    return a * b

# --- Vague Prompt: "Add comments to this function" ---
def multiply_vague(a, b):
    # Multiplies two numbers and returns the result
    return a * b

# --- Detailed Prompt: "Add Google-style docstrings with parameters, return types, and examples" ---
def multiply_detailed(a, b):
    """
    Multiplies two numbers.

    Args:
        a (int or float): The first number.
        b (int or float): The second number.

    Returns:
        int or float: The product of a and b.

    Examples:
        >>> multiply_detailed(2, 3)
        6
        >>> multiply_detailed(4.5, 2)
        9.0
    """
    return a * b

# --- Comparison Table ---
```



```
def multiply_detailed(a, b):
    """Returns:
    int or float: The product of a and b.

    Examples:
    >>> multiply_detailed(2, 3)
    6
    >>> multiply_detailed(4.5, 2)
    9.0
    """
    return a * b

# --- Comparison Table ---
comparison = [
    ["Prompt type", "Documentation Output", "Quality", "Accuracy", "Completeness"],
    ["Vague",
     "# Multiplies two numbers and returns the result",
     "Basic",
     "Correct",
     "Lacks parameter/return details, no examples"],
    ["Detailed",
     """"Google-style docstring with parameters, return types, and examples""",
     "High",
     "Correct",
     "Includes parameter types, return type, and usage examples"]
]

# Print comparison table
for row in comparison:
    print("{<12} {<60} {<10} {<10} {<50}".format(*row))
```

OUTPUT:

	Prompt Type	Documentation Output	Quality	Accuracy	Completeness
	Vague	# Multiplies two numbers and returns the result	Basic	Correct	Lacks parameter/return details, no examples
	Detailed	""Google-style docstring with parameters, return types, and examples""	High	Correct	Includes parameter types, return type, and usage examples