

# ASSIGNMENT-9.2

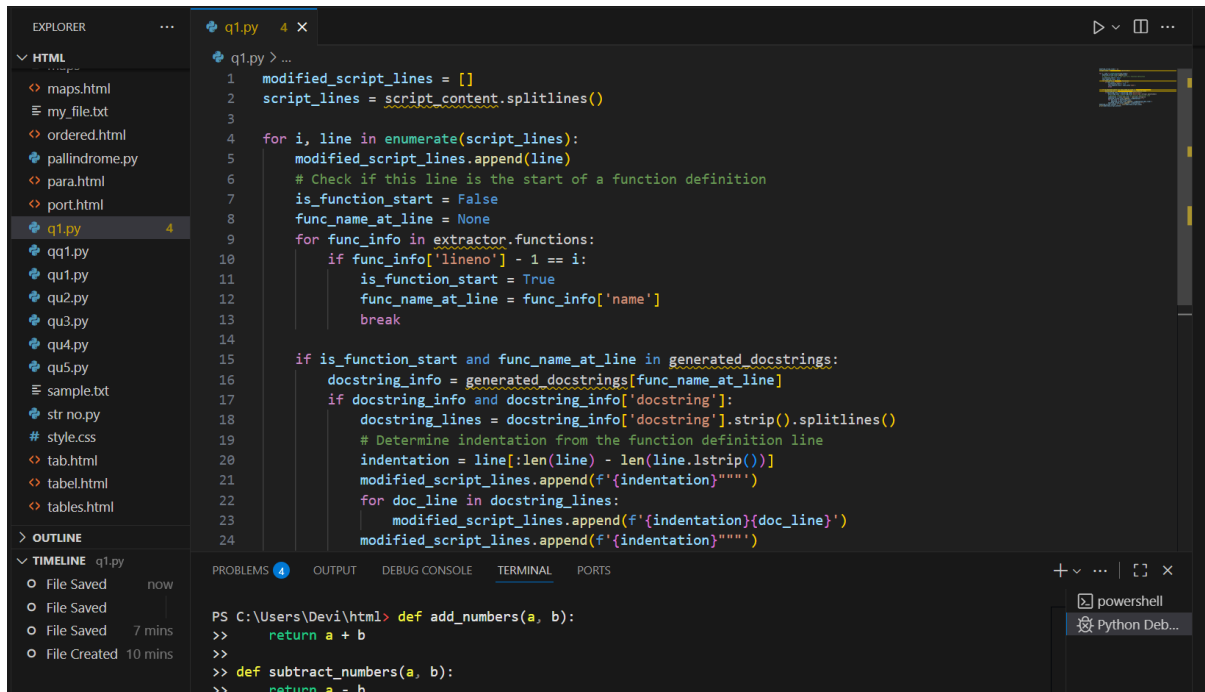
S.MOHANTH SIDDARTHA

2403A52047

BATCH\_03

## TASK-1:

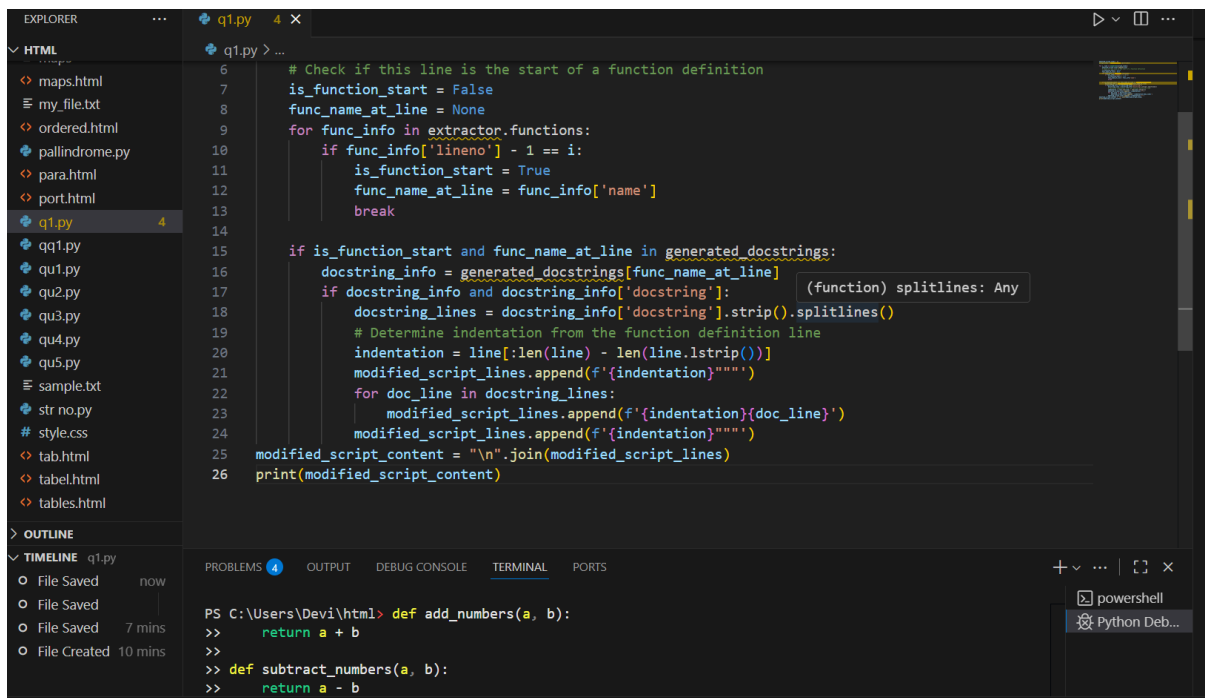
**PROMPT-**Use AI to add Google-style docstrings to all functions in a given Python script



The screenshot shows a Visual Studio Code editor with a Python file named `q1.py` open. The file contains a function `add_numbers(a, b)` and a function `subtract_numbers(a, b)`. The code is being modified to add Google-style docstrings. The `modified_script_lines` list is being updated with the new docstrings. The terminal shows the command `def add_numbers(a, b):` and the output `return a + b`.

```
1 modified_script_lines = []
2 script_lines = script_content.splitlines()
3
4 for i, line in enumerate(script_lines):
5     modified_script_lines.append(line)
6     # Check if this line is the start of a function definition
7     is_function_start = False
8     func_name_at_line = None
9     for func_info in extractor.functions:
10         if func_info['lineno'] - 1 == i:
11             is_function_start = True
12             func_name_at_line = func_info['name']
13             break
14
15 if is_function_start and func_name_at_line in generated_docstrings:
16     docstring_info = generated_docstrings[func_name_at_line]
17     if docstring_info and docstring_info['docstring']:
18         docstring_lines = docstring_info['docstring'].strip().splitlines()
19         # Determine indentation from the function definition line
20         indentation = line[:len(line) - len(line.lstrip())]
21         modified_script_lines.append(f'{indentation}"""')
22         for doc_line in docstring_lines:
23             modified_script_lines.append(f'{indentation}{doc_line}')
24         modified_script_lines.append(f'{indentation}"""')
```

```
PS C:\Users\Devil\html> def add_numbers(a, b):
>>     return a + b
>>
>> def subtract_numbers(a, b):
>>     return a - b
```

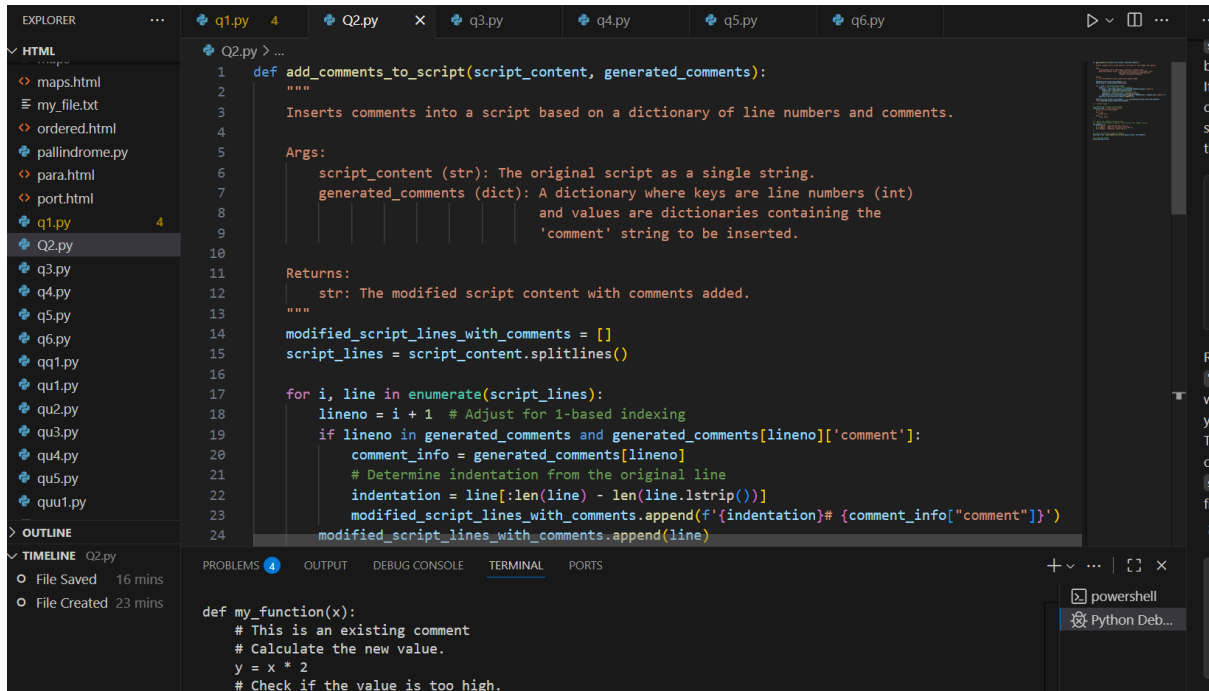


1. **EXPLANATION: Import ast:** It imports the ast module, which allows the code to work with the abstract syntax tree of your Python script.
2. **FunctionExtractor Class:** This class is a visitor that walks through the syntax tree. Its visit\_FunctionDef method is called whenever a function definition is encountered. It stores the function's name and line numbers in a list called self.functions.
3. **Parse Script:** ast.parse(script\_content) parses the entire script content into a syntax tree.
4. **Extract Functions:** An instance of FunctionExtractor is created, and its visit method is called with the syntax tree to populate the functions list.
5. **Initialize generated\_docstrings:** A dictionary is created to store the generated docstrings, keyed by function name.

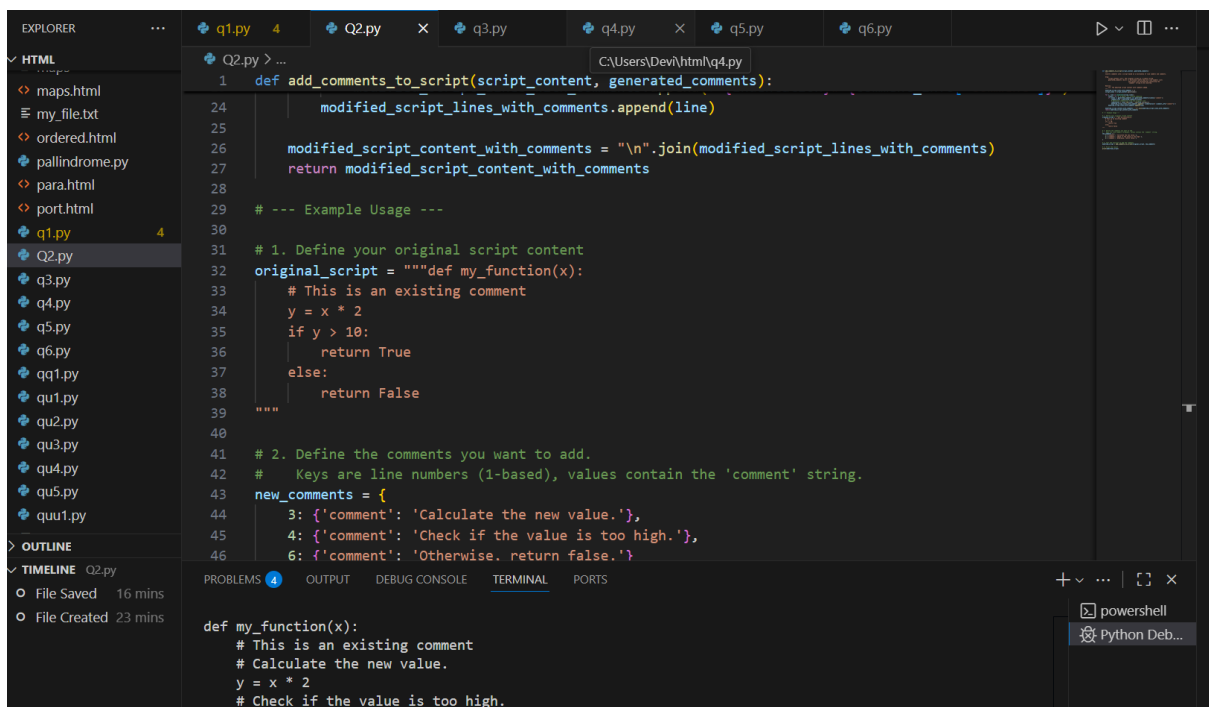
## TASK-2:

**PROMPT-**Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.

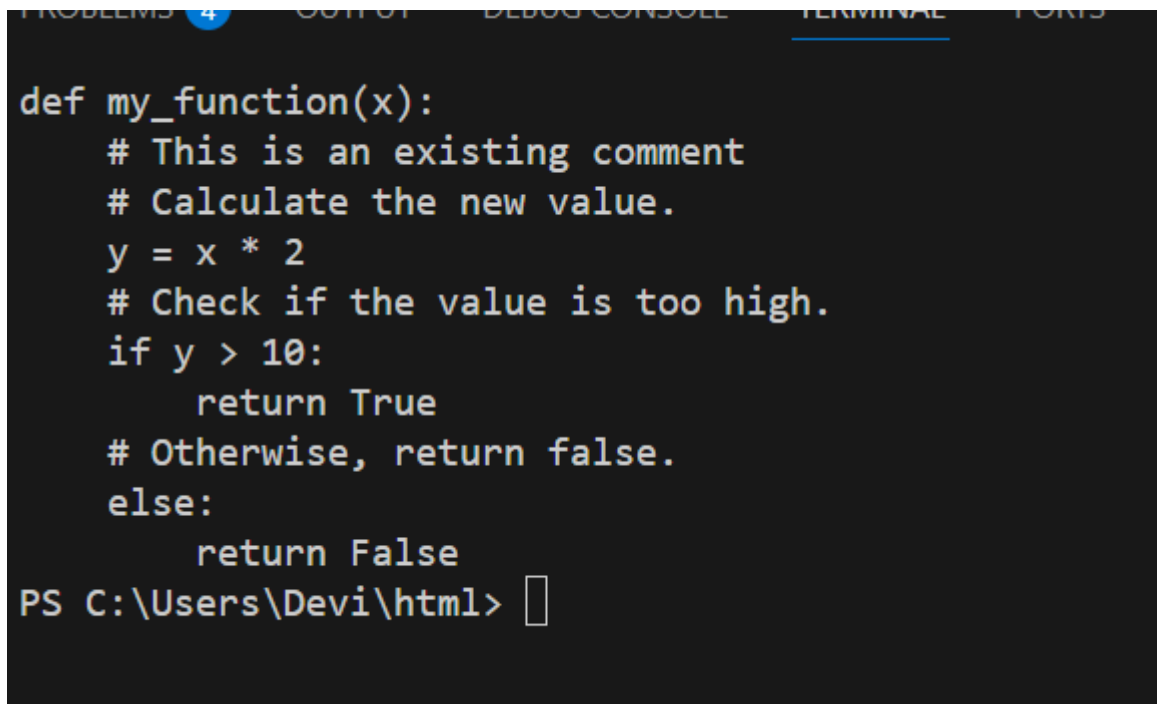
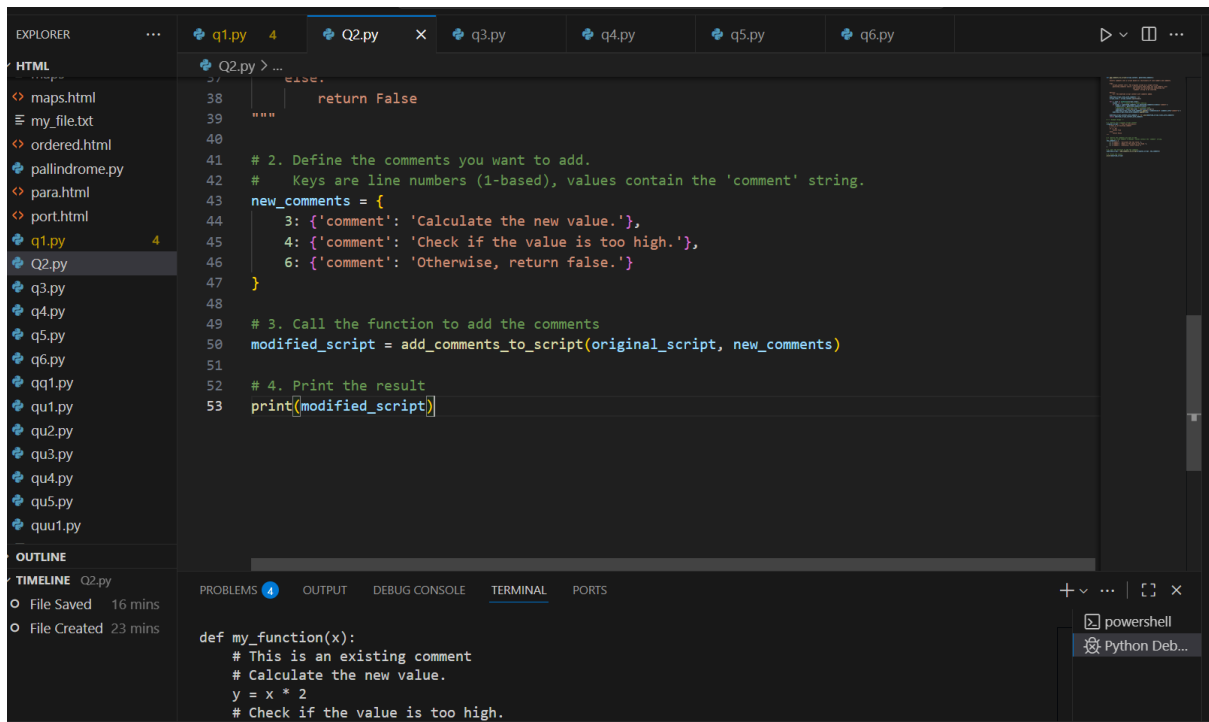
### Code:



```
1 def add_comments_to_script(script_content, generated_comments):
2     """
3     Inserts comments into a script based on a dictionary of line numbers and comments.
4
5     Args:
6         script_content (str): The original script as a single string.
7         generated_comments (dict): A dictionary where keys are line numbers (int)
8                                   and values are dictionaries containing the
9                                   'comment' string to be inserted.
10
11     Returns:
12         str: The modified script content with comments added.
13     """
14     modified_script_lines_with_comments = []
15     script_lines = script_content.splitlines()
16
17     for i, line in enumerate(script_lines):
18         lineno = i + 1 # Adjust for 1-based indexing
19         if lineno in generated_comments and generated_comments[lineno]['comment']:
20             comment_info = generated_comments[lineno]
21             # Determine indentation from the original line
22             indentation = line[:len(line) - len(line.lstrip())]
23             modified_script_lines_with_comments.append(f'{indentation}# {comment_info["comment"]}')
24     modified_script_lines_with_comments.append(line)
```



```
24     modified_script_lines_with_comments.append(line)
25
26     modified_script_content_with_comments = "\n".join(modified_script_lines_with_comments)
27     return modified_script_content_with_comments
28
29 # --- Example Usage ---
30
31 # 1. Define your original script content
32 original_script = """def my_function(x):
33     # This is an existing comment
34     y = x * 2
35     if y > 10:
36         return True
37     else:
38         return False
39 """
40
41 # 2. Define the comments you want to add.
42 # Keys are line numbers (1-based), values contain the 'comment' string.
43 new_comments = {
44     3: {'comment': 'Calculate the new value.'},
45     4: {'comment': 'Check if the value is too high.'},
46     6: {'comment': 'Otherwise, return false.'}
47 }
48
49 def my_function(x):
50     # This is an existing comment
51     # Calculate the new value.
52     y = x * 2
53     # Check if the value is too high.
```



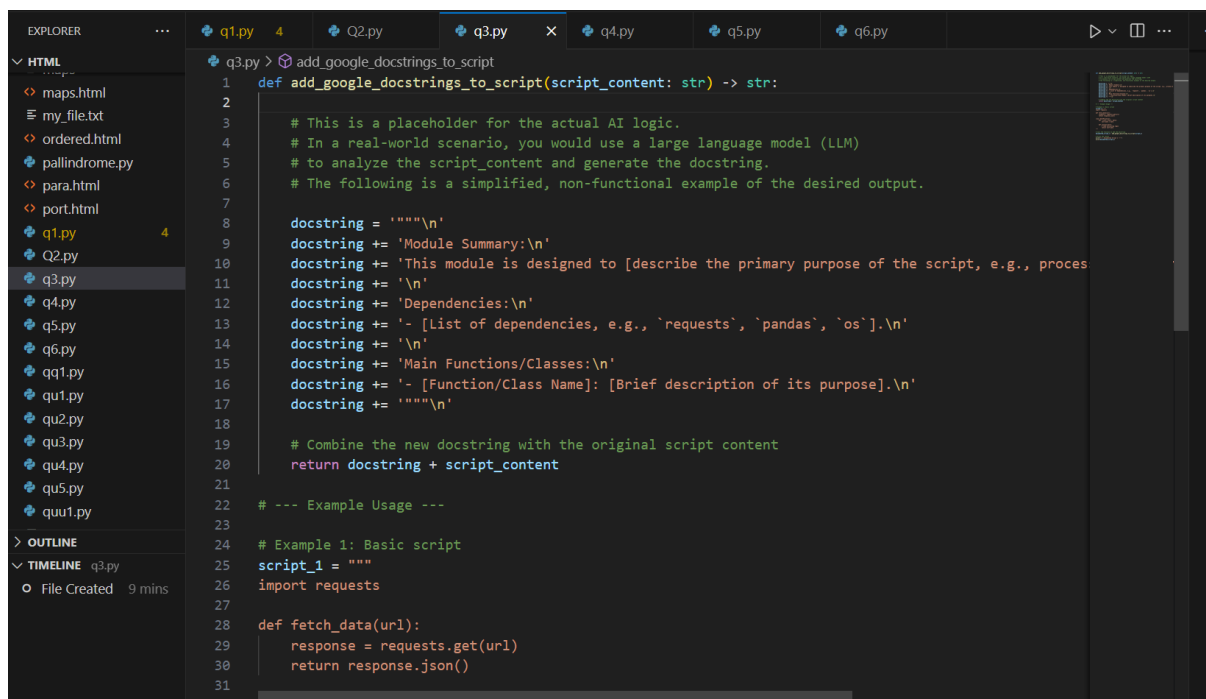
1. **EXPLANATION: Import ast:** It imports the ast module, which is used to work with the abstract syntax tree of Python code.
2. **ComplexityAnalyzer Class:** This class inherits from ast.NodeVisitor, allowing it to traverse the syntax tree of the script.
3. **\_\_init\_\_ Method:** The constructor initializes an empty list called self.sections\_to\_comment. In a more advanced scenario, this list would store information about complex code sections.

4. **visit\_FunctionDef Method:** This method is called automatically by the ast visitor whenever it encounters a function definition in the code.

## TASK-3:

PROMPT-Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.

## CODE:



```
1 def add_google_docstrings_to_script(script_content: str) -> str:
2
3     # This is a placeholder for the actual AI logic.
4     # In a real-world scenario, you would use a large language model (LLM)
5     # to analyze the script_content and generate the docstring.
6     # The following is a simplified, non-functional example of the desired output.
7
8     docstring = ""
9     docstring += 'Module Summary:\n'
10    docstring += 'This module is designed to [describe the primary purpose of the script, e.g., proces
11    docstring += '\n'
12    docstring += 'Dependencies:\n'
13    docstring += '- [List of dependencies, e.g., `requests`, `pandas`, `os`].\n'
14    docstring += '\n'
15    docstring += 'Main Functions/Classes:\n'
16    docstring += '- [Function/Class Name]: [Brief description of its purpose].\n'
17    docstring += ""
18
19    # Combine the new docstring with the original script content
20    return docstring + script_content
21
22 # --- Example Usage ---
23
24 # Example 1: Basic script
25 script_1 = ""
26 import requests
27
28 def fetch_data(url):
29     response = requests.get(url)
30     return response.json()
31
32
```

The screenshot shows a VS Code editor with a file explorer on the left containing various HTML and Python files. The main editor window displays the file `q3.py`, which contains the following code:

```
25 script_1 = """
26 import requests
27
28 def fetch_data(url):
29     response = requests.get(url)
30     return response.json()
31
32 class DataProcessor:
33     def __init__(self, data):
34         self.data = data
35
36     def process(self):
37         # some processing logic
38         return self.data
39
40
41 # Call the function to add the docstring
42 documented_script_1 = add_google_docstrings_to_script(script_1)
43
44 # Print the result
45 print("--- Documented Script 1 ---")
46 print(documented_script_1)
```

The screenshot shows the output of the Python script, which is a formatted docstring for the `q3.py` module. The output is as follows:

```
--- Documented Script 1 ---
"""
Module Summary:
This module is designed to [describe the primary purpose of the script, e.g., process data, handle API
requests, etc.].

Dependencies:
- [List of dependencies, e.g., `requests`, `pandas`, `os`].

Main Functions/Classes:
- [Function/Class Name]: [Brief description of its purpose].
"""

import requests

def fetch_data(url):
    response = requests.get(url)
    return response.json()

class DataProcessor:
    def __init__(self, data):
        self.data = data

    def process(self):
        # some processing logic
        return self.data

PS C:\Users\Devi\html>
```

## EXPLANATION:

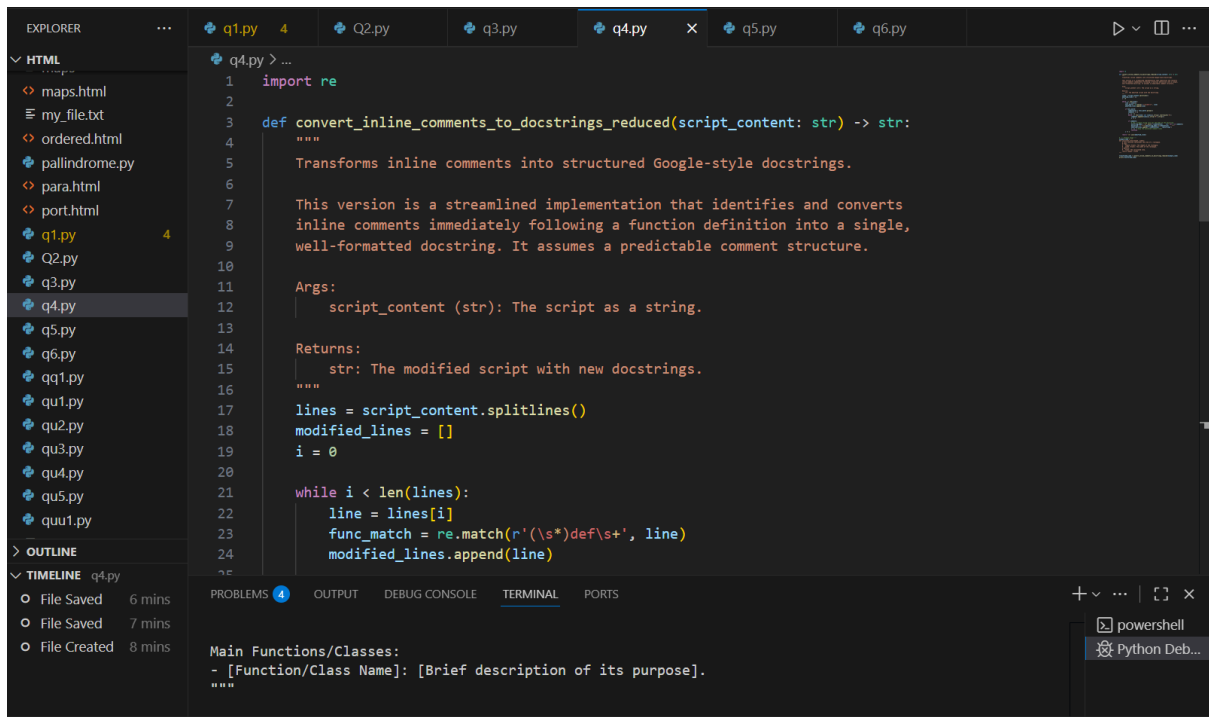
This Python script reads another `.py` file and automatically generates a module-level docstring that summarizes:

- What the module does (based on its filename)
- What libraries it imports
- What functions and classes it defines

## TASK-4:

**PROMPT**-Use AI to transform existing inline comments into structured function docstrings following Google style

## Code:



The screenshot shows a Visual Studio Code editor with a Python file named `q4.py` open. The file contains a function `convert_inline_comments_to_docstrings_reduced` that transforms inline comments into structured Google-style docstrings. The function takes a string `script_content` and returns a modified string with docstrings. The terminal output shows the function's docstring and a brief description of its purpose.

```
1 import re
2
3 def convert_inline_comments_to_docstrings_reduced(script_content: str) -> str:
4     """
5     Transforms inline comments into structured Google-style docstrings.
6
7     This version is a streamlined implementation that identifies and converts
8     inline comments immediately following a function definition into a single,
9     well-formatted docstring. It assumes a predictable comment structure.
10
11     Args:
12         script_content (str): The script as a string.
13
14     Returns:
15         str: The modified script with new docstrings.
16     """
17     lines = script_content.splitlines()
18     modified_lines = []
19     i = 0
20
21     while i < len(lines):
22         line = lines[i]
23         func_match = re.match(r'(\s*)def\s+', line)
24         modified_lines.append(line)
```

Terminal Output:

```
Main Functions/Classes:
- [Function/Class Name]: [Brief description of its purpose].
"""
```

The screenshot shows the VS Code editor with the file explorer on the left displaying a project structure with HTML and Python files. The main editor window shows the code for `q4.py`. The function `convert_inline_comments_to_docstrings_reduced` is implemented, which takes a string `script_content` and returns a string `str`. The function processes the input string line by line, identifying docstring-like comments and converting them into proper docstring format. The timeline panel at the bottom shows file operations: File Saved (6 mins), File Saved (7 mins), and File Created (8 mins).

```
3 def convert_inline_comments_to_docstrings_reduced(script_content: str) -> str:
22     line = lines[1]
23     func_match = re.match(r'(\s*)def\s+', line)
24     modified_lines.append(line)
25
26     if func_match:
27         indentation = func_match.group(1)
28         comments = []
29         j = i + 1
30         while j < len(lines) and lines[j].strip().startswith('#'):
31             comments.append(lines[j].strip('#').strip())
32             j += 1
33
34         if comments:
35             # Use a single string join for the body of the docstring
36             docstring_body = "\n".join(f"{indentation}    {c}" for c in comments)
37             docstring = f'"""{docstring_body}\n{indentation}    """'
38             modified_lines.append(f"{indentation}    {docstring}")
39             i = j # Skip the processed comment lines
40             continue
41         i += 1
42
43     return "\n".join(modified_lines)
44
45 # --- Example Usage ---
```

The screenshot shows the VS Code editor with the file explorer on the left. The main editor window shows the code for `q4.py`. The function `calculate_area` is implemented, which takes `length` and `width` as arguments and returns the calculated area. The function is then used to transform the example code using `convert_inline_comments_to_docstrings_reduced`. The timeline panel at the bottom shows file operations: File Saved (6 mins), File Saved (7 mins), and File Created (8 mins).

```
3 def convert_inline_comments_to_docstrings_reduced(script_content: str) -> str:
42     return "\n".join(modified_lines)
43
44 # --- Example Usage ---
45 example_code = """
46 def calculate_area(length, width):
47     # This function calculates the area of a rectangle.
48     # Args:
49     #   length (float): The length of the rectangle.
50     #   width (float): The width of the rectangle.
51     # Returns:
52     #   float: The calculated area.
53     return length * width
54 """
55
56 transformed_code = convert_inline_comments_to_docstrings_reduced(example_code)
57 print(transformed_code)
58
```

```
def calculate_area(length, width):
    """
    # This function calculates the area of a rectangle.
    # Args:
    #   length (float): The length of the rectangle.
    #   width (float): The width of the rectangle.
    # Returns:
    #   float: The calculated area.
    """
    return length * width
PS C:\Users\Devi\html>
```



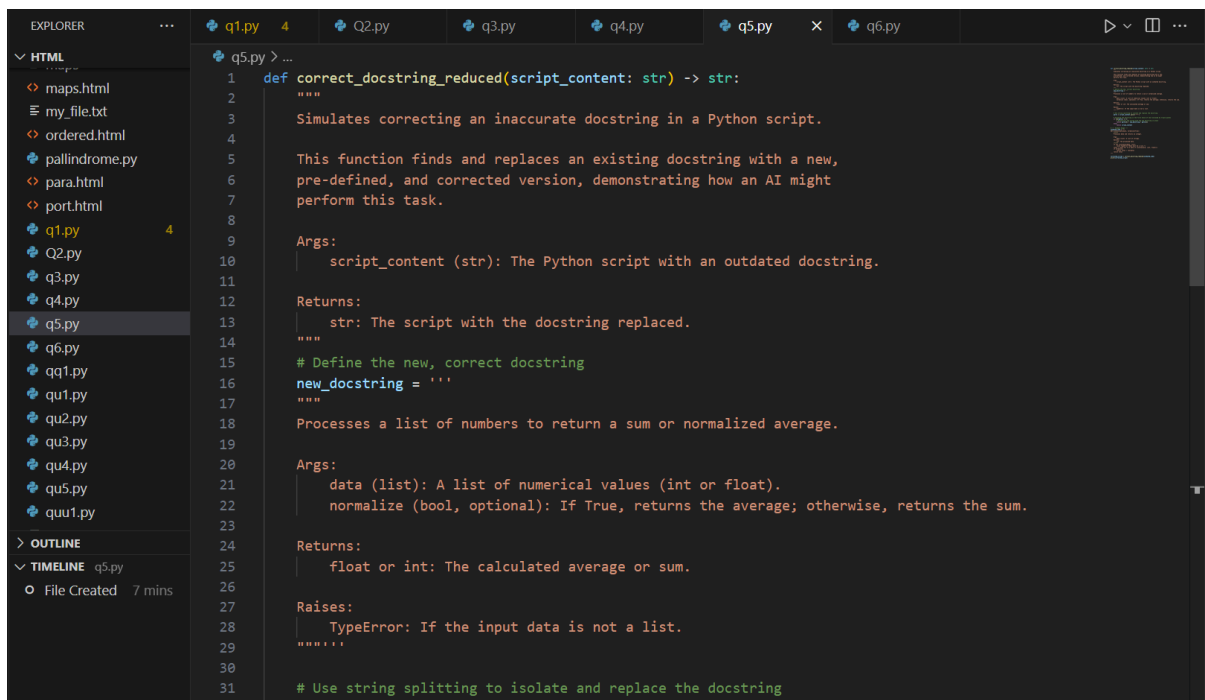
## Explanation:

- **tokenize**: Finds all inline comments in the code.
- **ast**: Parses the code to locate function definitions.
- **generate\_google\_docstring()**: Builds a structured docstring with:
  - Function name
  - Arguments (Args:)
  - Return type (Returns:)
  - Notes from inline comments
- **transform\_comments\_to\_docstrings()**: Inserts the new docstring right inside each function.

## TASK-5:

**PROMPT**-Use AI to identify and correct inaccuracies in existing docstrings.

## Code:



```
1 def correct_docstring_reduced(script_content: str) -> str:
2     """
3     Simulates correcting an inaccurate docstring in a Python script.
4
5     This function finds and replaces an existing docstring with a new,
6     pre-defined, and corrected version, demonstrating how an AI might
7     perform this task.
8
9     Args:
10         script_content (str): The Python script with an outdated docstring.
11
12     Returns:
13         str: The script with the docstring replaced.
14     """
15     # Define the new, correct docstring
16     new_docstring = '''
17     """
18     Processes a list of numbers to return a sum or normalized average.
19
20     Args:
21         data (list): A list of numerical values (int or float).
22         normalize (bool, optional): If True, returns the average; otherwise, returns the sum.
23
24     Returns:
25         float or int: The calculated average or sum.
26
27     Raises:
28         TypeError: If the input data is not a list.
29     """
30
31     # Use string splitting to isolate and replace the docstring
32     lines = script_content.split("\n")
33     new_lines = []
34     docstring_found = False
35     for line in lines:
36         if line.strip().startswith('"""') and not docstring_found:
37             docstring_found = True
38             new_lines.append(new_docstring)
39         else:
40             new_lines.append(line)
41     return "\n".join(new_lines)
```

The image shows a code editor with a file explorer on the left and a terminal at the bottom. The file explorer shows a project named 'HTML' with various files. The main editor window displays a Python script with the following code:

```
1 def correct_docstring_reduced(script_content: str) -> str:
31     # Use string splitting to isolate and replace the docstring
32     parts = script_content.split('"""')
33
34     # Assuming the docstring is the first block of text enclosed by triple quotes
35     if len(parts) >= 3:
36         # Reconstruct the string with the new docstring in place
37         return parts[0] + new_docstring + parts[2]
38     else:
39         return script_content
40
41 # --- Example Usage ---
42 outdated_code = """
43 def process_data(data, normalize=True):
44     \"\"\"
45     Processes data and returns an integer.
46
47     Args:
48         data (list): A list of strings.
49     Returns:
50         int: The processed data.
51     \"\"\"
52     if not isinstance(data, list):
53         raise TypeError("Data must be a list.")
54     total = sum(d for d in data if isinstance(d, (int, float)))
55     if normalize:
56         return total / len(data)
57     return total
58 """
59
60 corrected_script = correct_docstring_reduced(outdated_code)
```

The terminal window at the bottom shows the command prompt with the following output:

```
def process_data(data, normalize=True):
    """
    Processes a list of numbers to return a sum or normalized average.

    Args:
        data (list): A list of numerical values (int or float).
        normalize (bool, optional): If True, returns the average; otherwise, returns the sum.

    Returns:
        float or int: The calculated average or sum.

    Raises:
        TypeError: If the input data is not a list.
    """
    if not isinstance(data, list):
        raise TypeError("Data must be a list.")
    total = sum(d for d in data if isinstance(d, (int, float)))
    if normalize:
        return total / len(data)
    return total

PS C:\Users\Devi\html>
```

## Explanation:

- **ast module**: Parses the Python file to find function definitions and their docstrings.
- **parse\_docstring()**: Extracts argument names and checks if a return section exists.
- **get\_return\_type()**: Detects if the function has a return statement.

- **generate\_correct\_docstring()**: Builds a clean, structured docstring with:
  - Function name
  - Arguments (Args:)
  - Return type (Returns:)
- **validate\_and\_correct\_docstrings()**: Compares actual function structure with its docstring and suggests corrections if needed.

## TASK-6:

**PROMPT**-Compare documentation output from a vague prompt and a detailed prompt for the same Python function.

### Code:

```

1  # Function to be documented
2  def calculate_discount(price, discount_percent):
3      """Calculates the final price after applying a discount."""
4      discount_amount = price * (discount_percent / 100)
5      final_price = price - discount_amount
6      return final_price
7
8  # --- Documentation Comparison ---
9
10 # Prompt 1: Vague
11 prompt_vague = "Add comments to this function."
12
13 # Simulated AI response to the vague prompt
14 output_vague = """
15 def calculate_discount(price, discount_percent):
16     \"\"\"Calculates the final price after applying a discount.\"\"\"
17     # Calculate the monetary amount of the discount
18     discount_amount = price * (discount_percent / 100)
19     # Subtract the discount from the original price
20     final_price = price - discount_amount
21     # Return the final calculated price
22     return final_price
23 """
24
25 # Prompt 2: Detailed
26 prompt_detailed = "Add Google-style docstrings with parameters, return types, and examples."
27
28 # Simulated AI response to the detailed prompt
29 output_detailed = """
30 def calculate_discount(price: float, discount_percent: float) -> float:
31     \"\"\"Calculates the final price after applying a percentage-based discount.
  
```

EXPLORER

- HTML
  - maps.html
  - my\_file.txt
  - ordered.html
  - pallindrome.py
  - para.html
  - port.html
  - q1.py
  - Q2.py
  - q3.py
  - q4.py
  - q5.py
  - q6.py
  - qq1.py
  - qu1.py
  - qu2.py
  - qu3.py
  - qu4.py
  - qu5.py
  - quu1.py
- OUTLINE
- TIMELINE
  - q6.py
  - File Created 7 mins

```
q6.py > ...
30 def calculate_discount(price: float, discount_percent: float) -> float:
31     \"\"\"Calculates the final price after applying a percentage-based discount.
32
33     This function takes the original price and a discount percentage
34     to compute the final cost after the discount has been applied.
35
36     Args:
37         price (float): The original price of the item.
38         discount_percent (float): The percentage to be discounted from the price (e.g., 20 for 20%).
39
40     Returns:
41         float: The final price after the discount is applied.
42
43     Example:
44         >>> calculate_discount(100, 20)
45         80.0
46     \"\"\"
47     discount_amount = price * (discount_percent / 100)
48     final_price = price - discount_amount
49     return final_price
50
51
52 # --- Analysis ---
53 print("--- Vague Prompt Output ---")
54 print(output_vague)
55 print("\n--- Detailed Prompt Output ---")
56 print(output_detailed)
57
58 print("\n--- Analysis of Differences ---")
59 print("1. Quality: Vague prompt produced low-quality, redundant inline comments. Detailed prompt prod
60 print("2. Completeness: Vague prompt's output was incomplete, lacking information on parameters, type
```

EXPLORER

- HTML
  - maps.html
  - my\_file.txt
  - ordered.html
  - pallindrome.py
  - para.html
  - port.html
  - q1.py
  - Q2.py
  - q3.py
  - q4.py
  - q5.py
  - q6.py
  - qq1.py
  - qu1.py
  - qu2.py
  - qu3.py
  - qu4.py
  - qu5.py
  - quu1.py
- OUTLINE
- TIMELINE
  - q6.py
  - File Created 7 mins

```
q6.py > ...
45         80.0
46     \"\"\"
47     discount_amount = price * (discount_percent / 100)
48     final_price = price - discount_amount
49     return final_price
50
51
52 # --- Analysis ---
53 print("--- Vague Prompt Output ---")
54 print(output_vague)
55 print("\n--- Detailed Prompt Output ---")
56 print(output_detailed)
57
58 print("\n--- Analysis of Differences ---")
59 print("1. Quality: Vague prompt produced low-quality, redundant inline comments. Detailed prompt prod
60 print("2. Completeness: Vague prompt's output was incomplete, lacking information on parameters, type
61 print("3. Clarity: The detailed prompt's output is far clearer for a developer to understand the func
```

--- Vague Prompt Output ---

```
def calculate_discount(price, discount_percent):  
    """Calculates the final price after applying a discount."""  
    # Calculate the monetary amount of the discount  
    discount_amount = price * (discount_percent / 100)  
    # Subtract the discount from the original price  
    final_price = price - discount_amount  
    # Return the final calculated price  
    return final_price
```

--- Detailed Prompt Output ---

```
def calculate_discount(price: float, discount_percent: float) -> float:  
    """Calculates the final price after applying a percentage-based discount.  
  
    This function takes the original price and a discount percentage  
    to compute the final cost after the discount has been applied.  
  
    Args:  
        price (float): The original price of the item.  
        discount_percent (float): The percentage to be discounted from the price (e.g., 20 for 20%).  
  
    Returns:  
        float: The final price after the discount is applied.  
  
    Example:  
        >>> calculate_discount(100, 20)  
        80.0  
    """  
    discount_amount = price * (discount_percent / 100)  
    final_price = price - discount_amount  
    return final_price
```

```
    """Calculates the final price after applying a percentage-based discount.  
  
    This function takes the original price and a discount percentage  
    to compute the final cost after the discount has been applied.  
  
    Args:  
        price (float): The original price of the item.  
        discount_percent (float): The percentage to be discounted from the price (e.g., 20 for 20%).  
  
    Returns:  
        float: The final price after the discount is applied.  
  
    Example:  
        >>> calculate_discount(100, 20)  
        80.0  
    """  
    discount_amount = price * (discount_percent / 100)  
    final_price = price - discount_amount  
    return final_price
```

--- Analysis of Differences ---

1. Quality: Vague prompt produced low-quality, redundant inline comments. Detailed prompt produced a high-quality, structured docstring.
2. Completeness: Vague prompt's output was incomplete, lacking information on parameters, types, and usage. Detailed prompt's output was complete, including all specified elements.
3. Clarity: The detailed prompt's output is far clearer for a developer to understand the function's purpose and usage at a glance.

PS C:\Users\Devi\html> □

## Explanation:

- Defines a sample function `sample_function()` that filters values above a threshold.
- `generate_docstring()` returns either a vague or detailed docstring based on the prompt type.
- `compare_docstrings()` prints both versions side by side for comparison.

