ASSIGNMENT-9.2

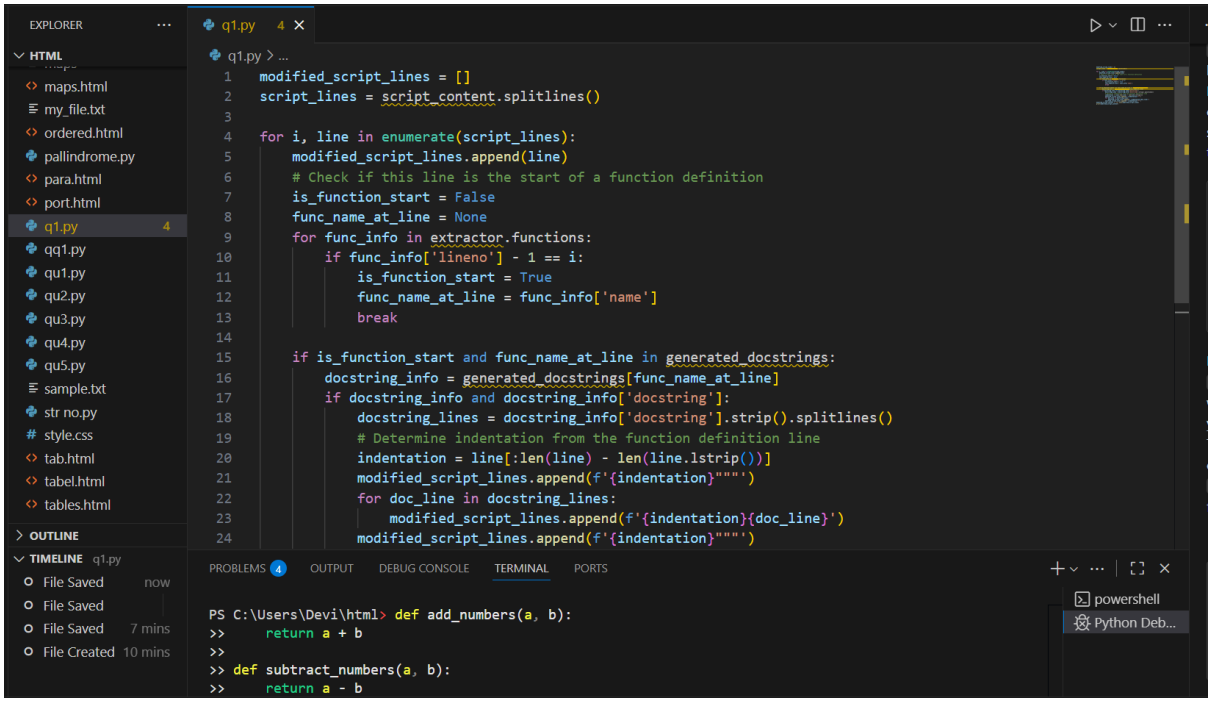
S.MOHANTH SIDDARTHA

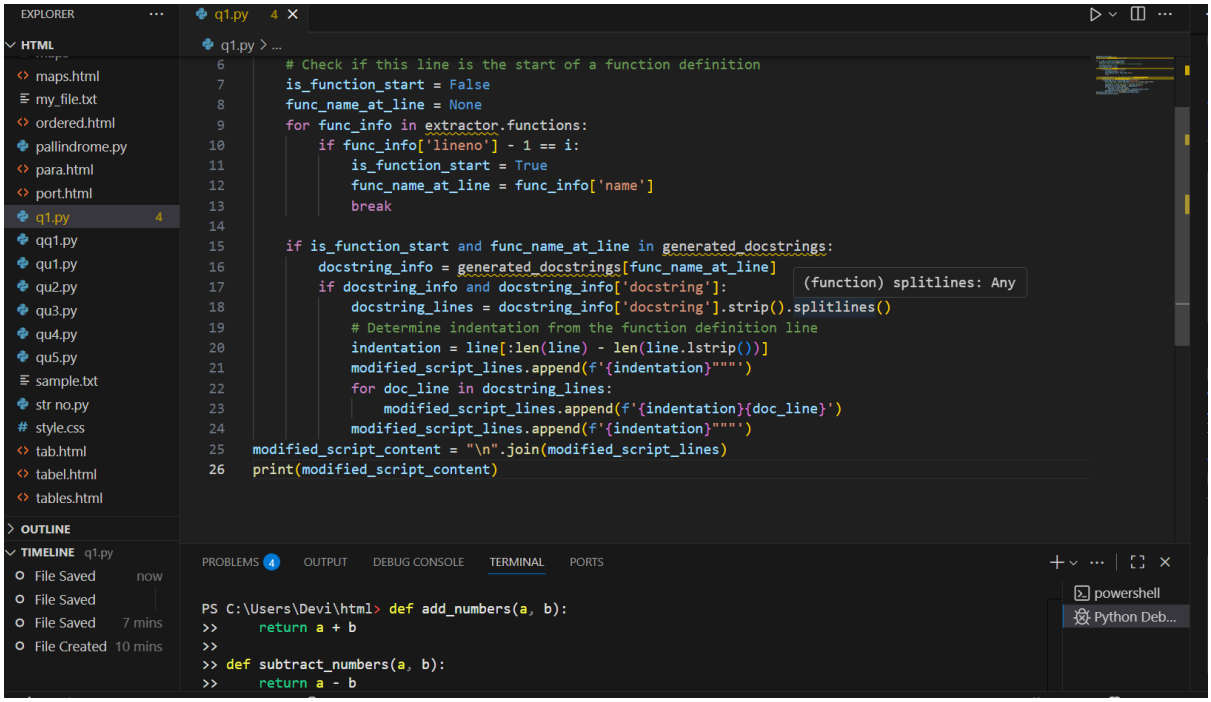
2403A52047

BATCH_03

TASK-1:

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





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.

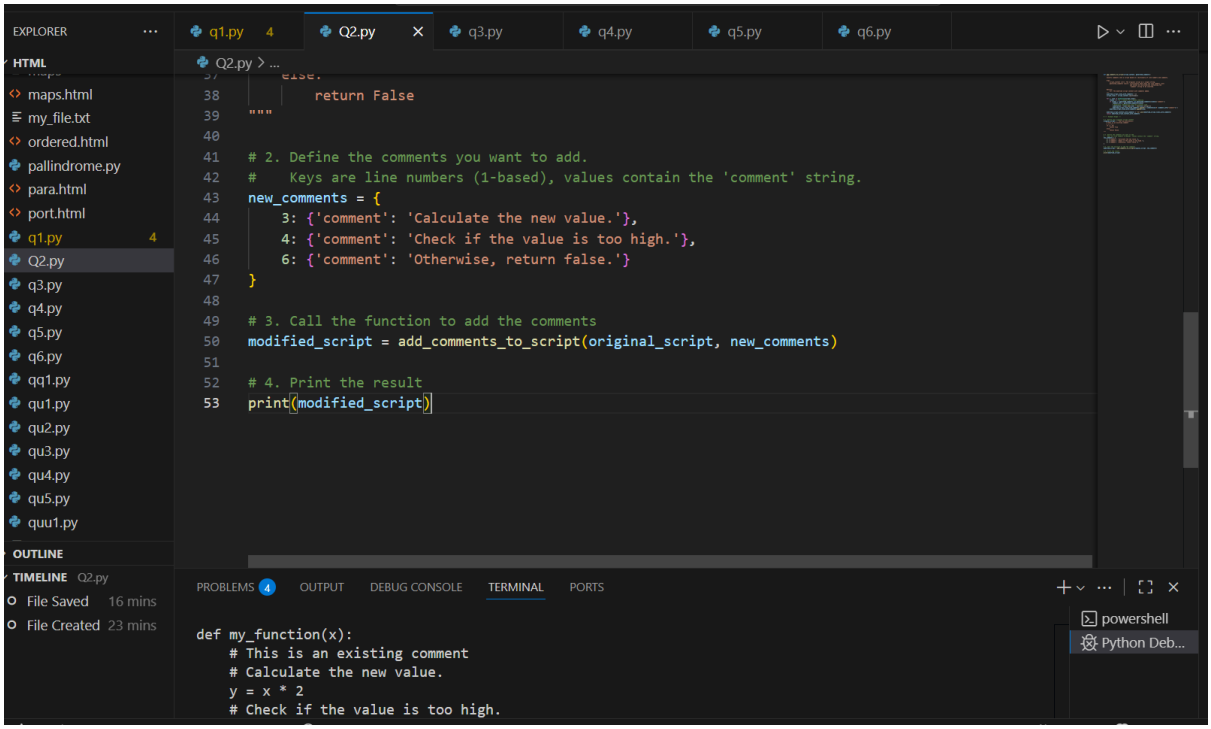
Code:

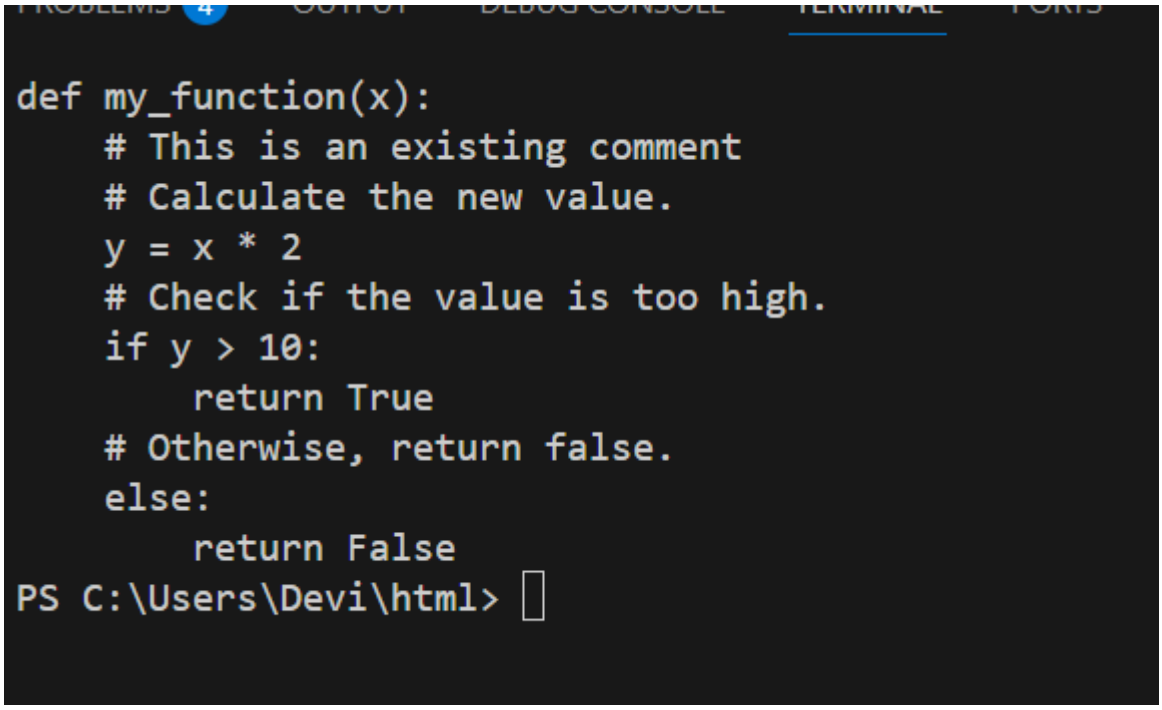
```python
def add_comments_to_script(script_content, generated_comments):
    """
    Inserts comments into a script based on a dictionary of line numbers and comments.

    Args:
        script_content (str): The original script as a single string.
        generated_comments (dict): A dictionary where keys are line numbers (int)
                                   and values are dictionaries containing the
                                   'comment' string to be inserted.

    Returns:
        str: The modified script content with comments added.
    """
    modified_script_lines_with_comments = []
    script_lines = script_content.splitlines()

    for i, line in enumerate(script_lines):
        lineno = i + 1  # Adjust for 1-based indexing
        if lineno in generated_comments and generated_comments[lineno]['comment']:
            comment_info = generated_comments[lineno]
            # Determine indentation from the original line
            indentation = line[:len(line) - len(line.lstrip())]
            modified_script_lines_with_comments.append(f'{indentation}# {comment_info["comment"]}')
        modified_script_lines_with_comments.append(line)
```

```
def my_function(x):
    # This is an existing comment
    # Calculate the new value.
    y = x * 2
    # Check if the value is too high.
```

```python
def add_comments_to_script(script_content, generated_comments):
        modified_script_lines_with_comments.append(line)

    modified_script_content_with_comments = "\n".join(modified_script_lines_with_comments)
    return modified_script_content_with_comments

# --- Example Usage ---

# 1. Define your original script content
original_script = """def my_function(x):
    # This is an existing comment
    y = x * 2
    if y > 10:
        return True
    else:
        return False
"""

# 2. Define the comments you want to add.
#    Keys are line numbers (1-based), values contain the 'comment' string.
new_comments = {
    3: {'comment': 'Calculate the new value.'},
    4: {'comment': 'Check if the value is too high.'},
    6: {'comment': 'Otherwise, return false.'}
```

```
def my_function(x):
    # This is an existing comment
    # Calculate the new value.
    y = x * 2
    # Check if the value is too high.
```





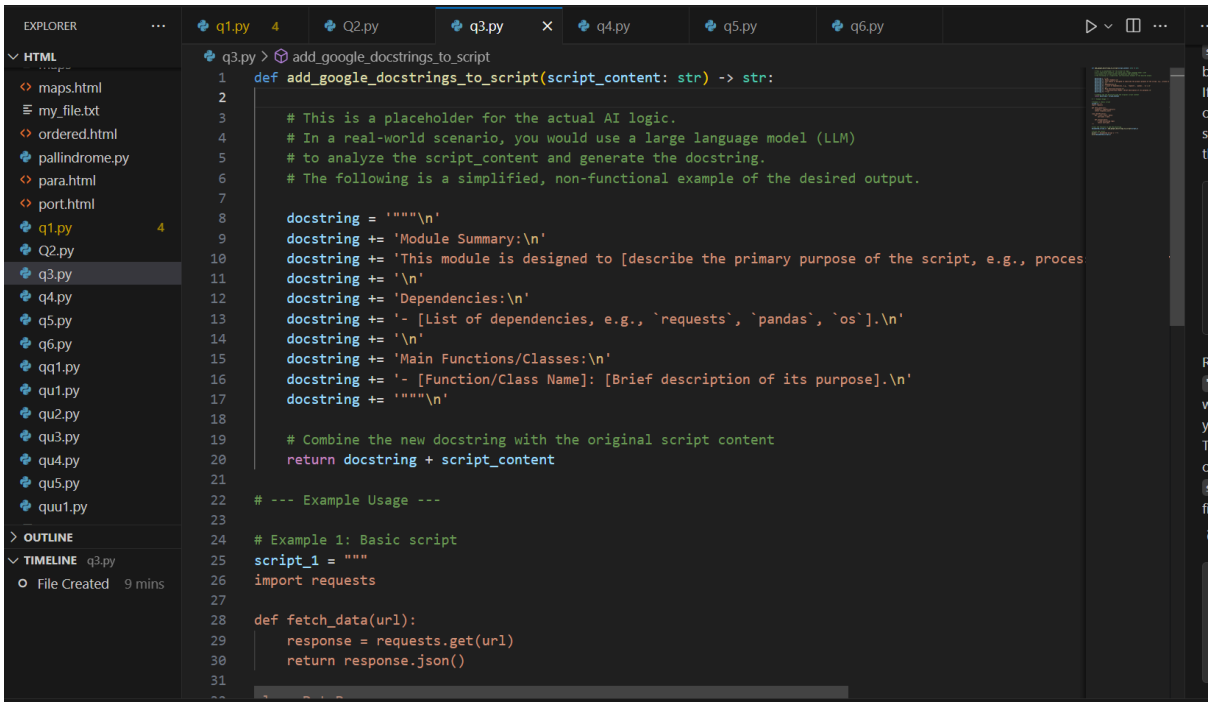
1. **EXPLANATION: Import ast**: It imports the ast module, which is used to work with the abstract syntax tree of Python code.
2. **ComplexityAnalyzer Class**: This class inherits from ast.NodeVisitor, allowing it to traverse the syntax tree of the script.
3. **__init__ Method**: The constructor initializes 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:



```python
script_1 = """
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
"""

# Call the function to add the docstring
documented_script_1 = add_google_docstrings_to_script(script_1)

# Print the result
print("--- Documented Script 1 ---")
print(documented_script_1)
```

```
--- 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:

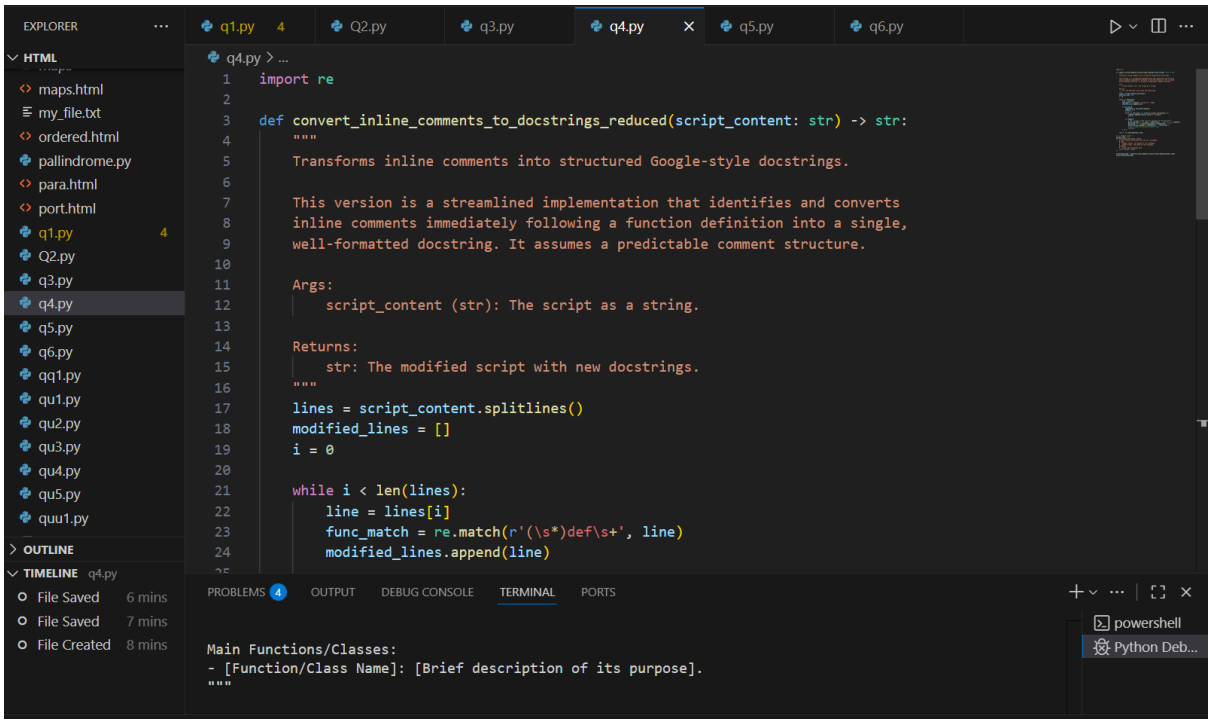
This Python script reads another `.py` file and automatically generates a module-level docstring that summarizes:

- 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:



EXPLORER

HTML

- maps.html
- my_file.txt
- ordered.html
- pallindrome.py
- para.html
- port.html
- q1.py 4
- 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 q4.py

- File Saved 6 mins
- File Saved 7 mins
- File Created 8 mins

q1.py 4 | Q2.py | q3.py | q4.py | q5.py | q6.py

q4.py > ...

```python
def convert_inline_comments_to_docstrings_reduced(script_content: str) -> str:
        line = lines[i]
        func_match = re.match(r'(\s*)def\s+', line)
        modified_lines.append(line)

        if func_match:
            indentation = func_match.group(1)
            comments = []
            j = i + 1
            while j < len(lines) and lines[j].strip().startswith('#'):
                comments.append(lines[j].strip('#').strip())
                j += 1

            if comments:
                # Use a single string join for the body of the docstring
                docstring_body = "\n".join(f"{indentation}    {c}" for c in comments)
                docstring = f'"""\n{docstring_body}\n{indentation}    """'
                modified_lines.append(f"{indentation}    {docstring}")
                i = j  # Skip the processed comment lines
                continue
        i += 1

    return "\n".join(modified_lines)

# --- Example Usage ---
```

PROBLEMS 4 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

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

powershell

Python Deb...

EXPLORER

HTML

- maps.html
- my_file.txt
- ordered.html
- pallindrome.py
- para.html
- port.html
- q1.py 4
- 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 q4.py

- File Saved 6 mins
- File Saved 7 mins
- File Created 8 mins

q1.py 4 | Q2.py | q3.py | q4.py | q5.py | q6.py

q4.py > ...

```python
def convert_inline_comments_to_docstrings_reduced(script_content: str) -> str:

    return "\n".join(modified_lines)

# --- Example Usage ---
example_code = """
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
"""

transformed_code = convert_inline_comments_to_docstrings_reduced(example_code)
print(transformed_code)
```

PROBLEMS 4 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

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

powershell

Python Deb...

```python
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:

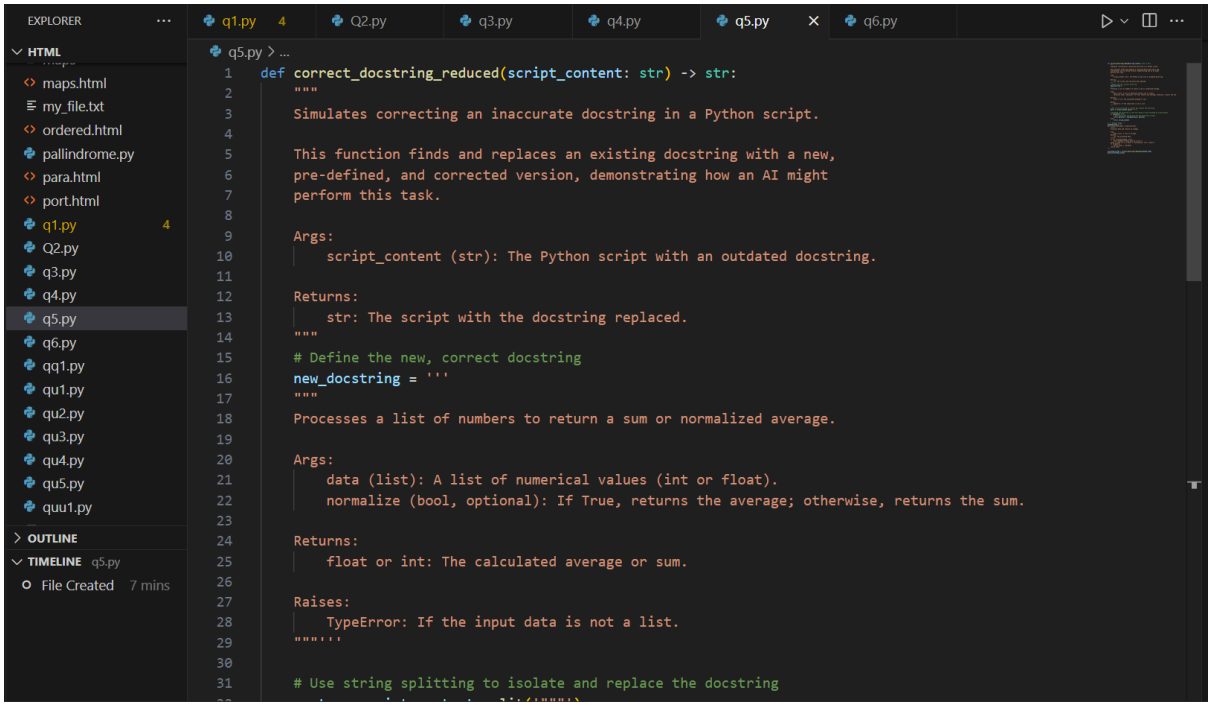
- **`tokenize`**: Finds all inline comments in the code.

  - **`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:



```python
def correct_docstring_reduced(script_content: str) -> str:
    # Use string splitting to isolate and replace the docstring
    parts = script_content.split('"""')

    # Assuming the docstring is the first block of text enclosed by triple quotes
    if len(parts) >= 3:
        # Reconstruct the string with the new docstring in place
        return parts[0] + new_docstring + parts[2]
    else:
        return script_content

# --- Example Usage ---
outdated_code = """
def process_data(data, normalize=True):
    \"\"\"
    Processes data and returns an integer.

    Args:
        data (list): A list of strings.
    Returns:
        int: The processed data.
    \"\"\"
    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
"""

corrected_script = correct_docstring_reduced(outdated_code)
```

```python
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.

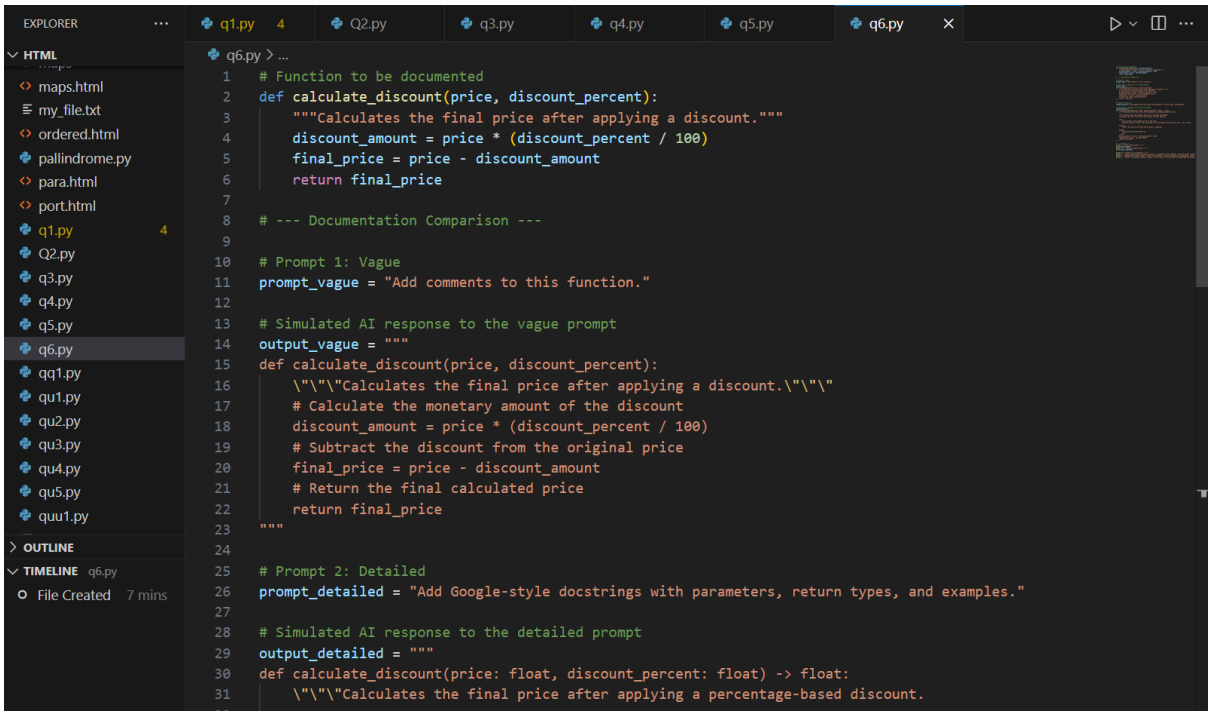
- **`parse_docstring()`**: Extracts argument names and checks if a return section exists.
- **`get_return_type()`**: Detects if the function has a return statement.

- **`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:



EXPLORER

HTML

- maps.html
- my_file.txt
- ordered.html
- pallindrome.py
- para.html
- port.html
- q1.py 4
- 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

q1.py 4 | Q2.py | q3.py | q4.py | q5.py | q6.py

q6.py > ...

```python
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
"""

# --- Analysis ---
print("--- Vague Prompt Output ---")
print(output_vague)
print("\n--- Detailed Prompt Output ---")
print(output_detailed)

print("\n--- Analysis of Differences ---")
print("1.  Quality: Vague prompt produced low-quality, redundant inline comments. Detailed prompt prod
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 4
- 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

q1.py 4 | Q2.py | q3.py | q4.py | q5.py | q6.py

q6.py > ...

```python
        80.0
    \"\"\"
    discount_amount = price * (discount_percent / 100)
    final_price = price - discount_amount
    return final_price
"""

# --- Analysis ---
print("--- Vague Prompt Output ---")
print(output_vague)
print("\n--- Detailed Prompt Output ---")
print(output_detailed)

print("\n--- Analysis of Differences ---")
print("1.  Quality: Vague prompt produced low-quality, redundant inline comments. Detailed prompt prod
print("2.  Completeness: Vague prompt's output was incomplete, lacking information on parameters, type
```


```python
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 h
igh-quality, structured docstring.
2.  Completeness: Vague prompt's output was incomplete, lacking information on parameters, types, and u
sage. 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 p
urpose 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.