

2303A51797

Batch : 26

Task 1: Auto-Generating Function Documentation in a Shared

Codebase

Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

You are given a Python script containing multiple functions without any docstrings.

Using an AI-assisted coding tool:

- Ask the AI to automatically generate Google-style function docstrings for each function

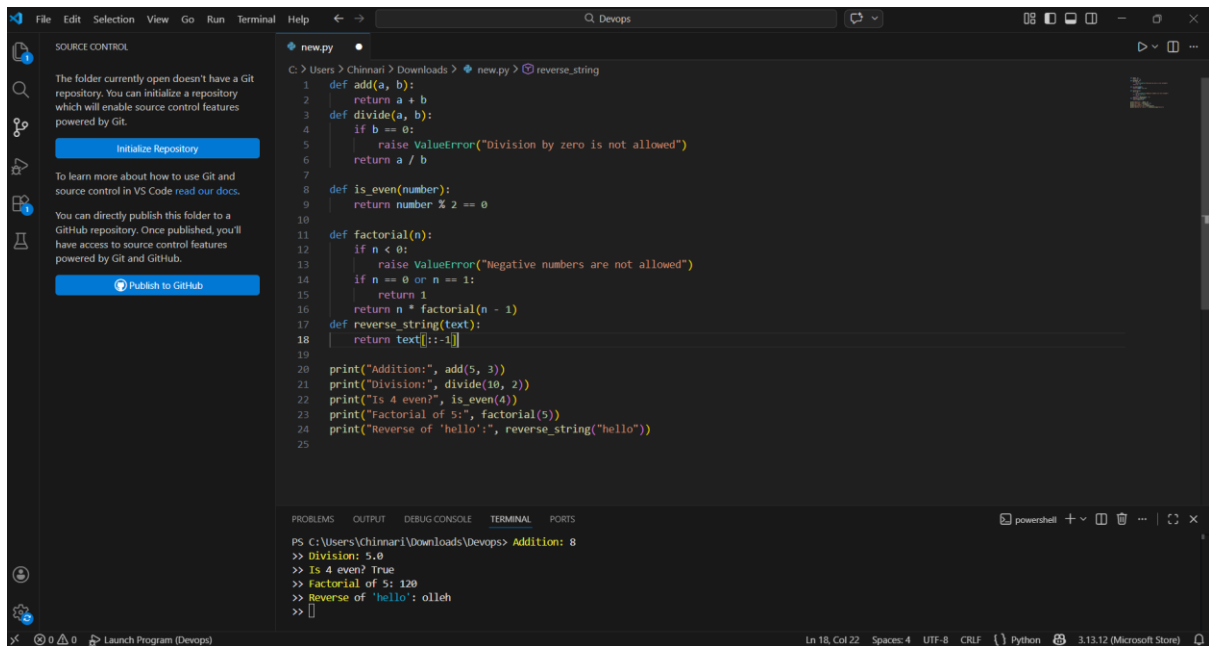
- Each docstring should include:

- o A brief description of the function
- o Parameters with data types
- o Return values
- o At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

Expected Outcome

- A Python script with well-structured Google-style docstrings
 - Docstrings that clearly explain function behavior and usage
- Improved readability and usability of the codebase give code for ths question for vs code



Task 2: Enhancing Readability Through AI-Generated Inline Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

You are provided with a Python script containing:

- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

- Automatically insert inline comments only for complex or non-obvious logic
- Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Expected Outcome

- A Python script with concise, meaningful inline comments
- Comments that explain why the logic exists, not what Python syntax does

- Noticeable improvement in code readability i want code for this with output

```

1  def fibonacci(n):
2      sequence = []
3      a, b = 0, 1
4
5      for _ in range(n):
6          sequence.append(a)
7          a, b = b, a + b
8
9      return sequence
10 def bubble_sort(arr):
11     n = len(arr)
12
13     for i in range(n):
14         for j in range(0, n - i - 1):
15             if arr[j] > arr[j + 1]:
16                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
17
18     return arr
19 def binary_search(arr, target):
20     left, right = 0, len(arr) - 1
21
22     while left <= right:
23         mid = (left + right) // 2
24         if arr[mid] == target:
25             return mid
26         elif arr[mid] < target:
27             left = mid + 1
28
29     return -1
30
31 if __name__ == "__main__":
32     print("Fibonacci (first 7 numbers):")
33     print(fibonacci(7))
34
35     print("\nBubble Sort:")
36     numbers = [64, 34, 25, 12, 22, 11, 90]
37     print("Before sorting:", numbers)
38     sorted_numbers = bubble_sort(numbers.copy())
39     print("After sorting:", sorted_numbers)
40
41     print("\nBinary Search:")
42     index = binary_search(sorted_numbers, 25)
43     print("Element 25 found at index:", index)

```

Output:

```

Fibonacci (first 7 numbers):
[0, 1, 1, 2, 3, 5, 8]

Bubble Sort:
Before sorting: [64, 34, 25, 12, 22, 11, 90]

```

```

19 def binary_search(arr, target):
20     left, right = 0, len(arr) - 1
21
22     while left <= right:
23         mid = (left + right) // 2
24         if arr[mid] == target:
25             return mid
26         elif arr[mid] < target:
27             left = mid + 1
28
29     return -1
30
31 if __name__ == "__main__":
32     print("Fibonacci (first 7 numbers):")
33     print(fibonacci(7))
34
35     print("\nBubble Sort:")
36     numbers = [64, 34, 25, 12, 22, 11, 90]
37     print("Before sorting:", numbers)
38     sorted_numbers = bubble_sort(numbers.copy())
39     print("After sorting:", sorted_numbers)
40
41     print("\nBinary Search:")
42     index = binary_search(sorted_numbers, 25)
43     print("Element 25 found at index:", index)

```

Output:

```

After sorting: [11, 12, 22, 25, 34, 64, 90]

Binary Search:
Element 25 found at index: 3

```

Task 3: Generating Module-Level Documentation for a Python

Package

Scenario

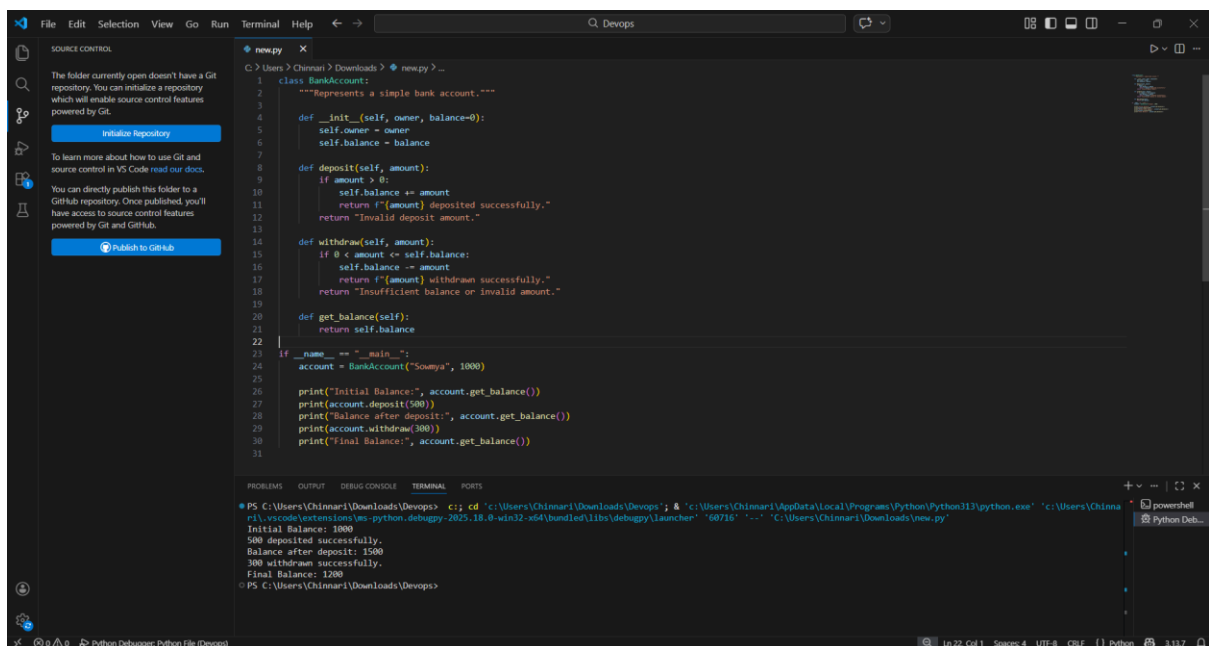
Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.

Task Description

- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

Expected Outcome

- A well-written multi-line module-level docstring
- Clear overview of what the module does and how to use it
- Documentation suitable for real-world projects or repositories



Scenario

Task Description

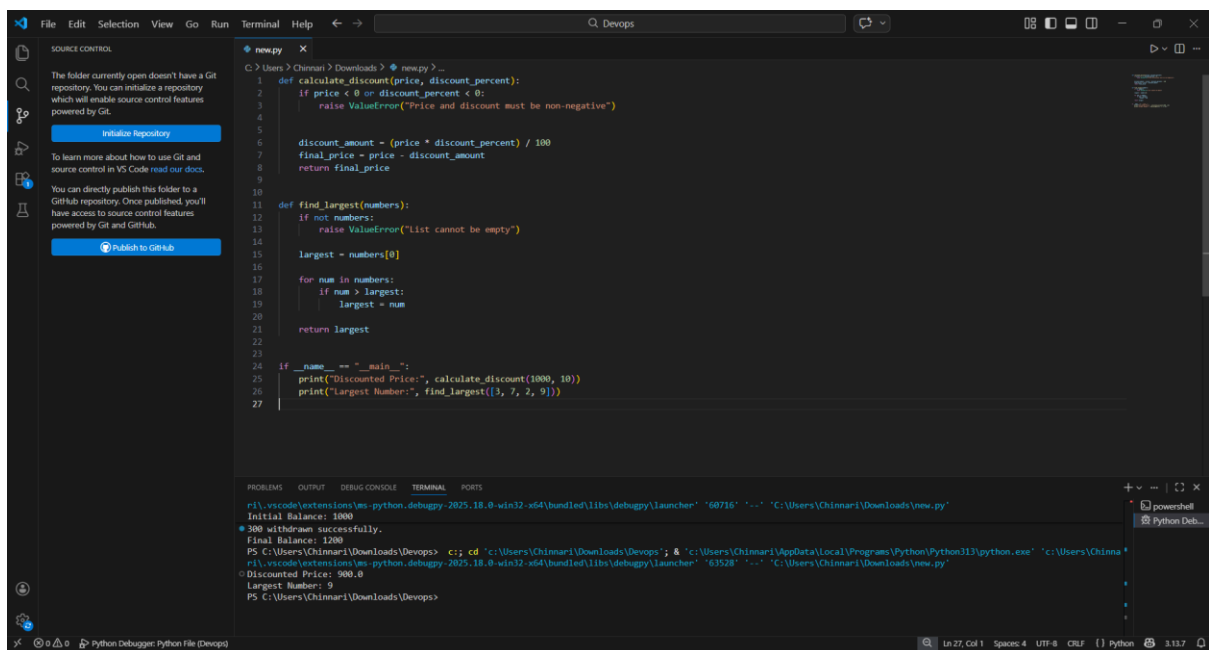
You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

- Automatically convert these comments into structured Google-style or NumPy-style docstrings
- Preserve the original meaning and intent of the comments
- Remove redundant inline comments after conversion

Expected Outcome

- Functions with clean, standardized docstrings
- Reduced clutter inside function bodies
- Improved consistency across the codebase i want code for this with output



```
1 def calculate_discount(price, discount_percent):
2     if price < 0 or discount_percent < 0:
3         raise ValueError("Price and discount must be non-negative")
4
5     discount_amount = (price * discount_percent) / 100
6     final_price = price - discount_amount
7     return final_price
8
9
10
11 def find_largest(numbers):
12     if not numbers:
13         raise ValueError("List cannot be empty")
14
15     largest = numbers[0]
16
17     for num in numbers:
18         if num > largest:
19             largest = num
20
21     return largest
22
23
24 if __name__ == "__main__":
25     print("Discounted Price:", calculate_discount(1000, 10))
26     print("Largest Number:", find_largest([3, 7, 2, 9]))
27
```

```
Initial Balance: 1000
Final Balance: 1200
PS C:\Users\Chinnar1\Downloads\Devops> c:\Users\Chinnar1\AppData\Local\Programs\Python\Python313\python.exe 'c:\Users\Chinnar1\Downloads\new.py'
Discounted Price: 900.0
Largest Number: 9
PS C:\Users\Chinnar1\Downloads\Devops>
```

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.

Task Description

Design a small Python utility that:

- Reads a given .py file
- Automatically detects:
 - o Functions
 - o Classes

- Inserts placeholder Google-style docstrings for each detected function or class

AI tools may be used to assist in generating or refining this utility.

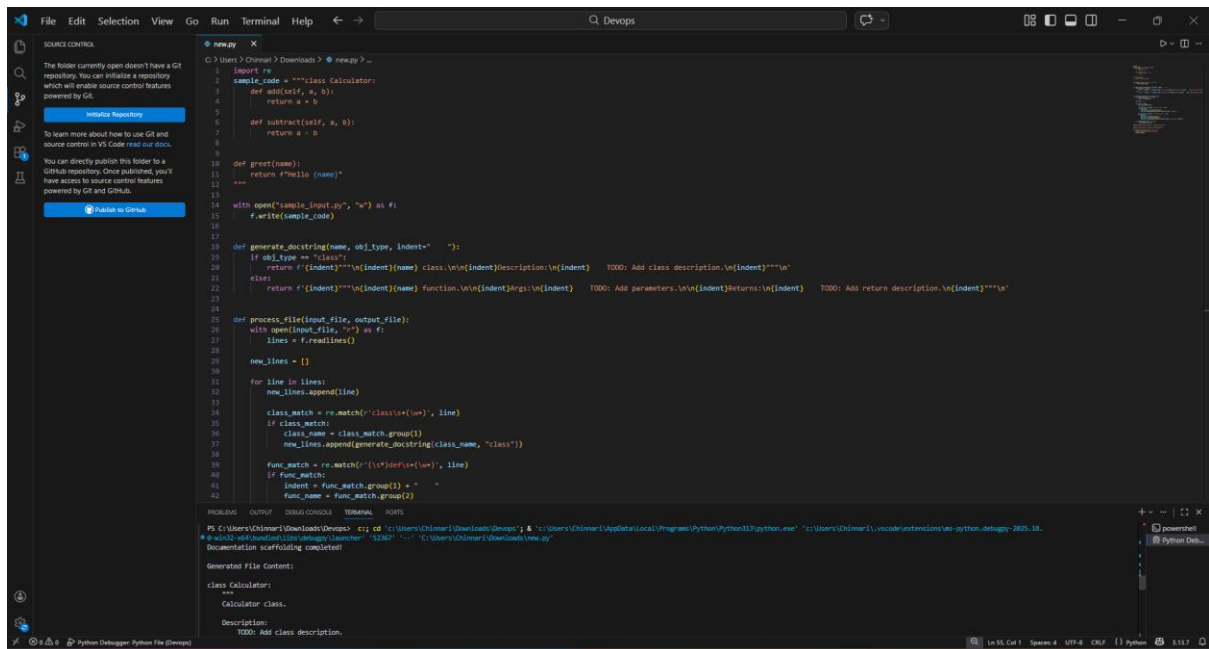
Note: The goal is documentation scaffolding, not perfect

documentation.

Expected Outcome

- A working Python script that processes another .py file
- Automatically inserted placeholder docstrings
- Clear demonstration of how AI can assist in documentation

automation i want code for this with output



```
1 import re
2 sample_code = """class Calculator:
3     def add(self, a, b):
4         return a + b
5
6     def subtract(self, a, b):
7         return a - b
8
9
10 def greet(name):
11     return f'Hello {name}'
12
13 """
14 with open("sample_input.py", "r") as f:
15     f.write(sample_code)
16
17 def generate_docstring(name, obj_type, indent="    "):
18     if obj_type == "class":
19         return f'({indent})"""({name}) class.\n\n({indent})Description:\n\n({indent})    TODO: Add class description.\n\n({indent})"""'
20     else:
21         return f'({indent})"""({name}) function.\n\n({indent})Args:\n\n({indent})    TODO: Add parameters.\n\n({indent})Returns:\n\n({indent})    TODO: Add return description.\n\n({indent})"""'
22
23
24 def process_file(input_file, output_file):
25     with open(input_file, "r") as f:
26         lines = f.readlines()
27
28     new_lines = []
29
30     for line in lines:
31         new_lines.append(line)
32
33     class_match = re.match(r'^(class\s+)(\w+)', line)
34     if class_match:
35         class_name = class_match.group(1)
36         new_lines.append(generate_docstring(class_name, "class"))
37
38     func_match = re.match(r'^(def\s+)(\w+)', line)
39     if func_match:
40         func_name = func_match.group(1)
41         new_lines.append(generate_docstring(func_name, "function"))
42
43
44 # Example usage
45 process_file("sample_input.py", "sample_output.py")
```

Generated File Content:

```
class Calculator:
    """
    Calculator class.

    Description:
    TODO: Add class description.
    """
```