ASSIGNMENT-13.5

2303A51600

BATCH-29

TASK-1

PROMPT:

"""You are given a Python script that contains repeated logic.

Your task is to:

1. Identify duplicate code blocks.

2. Refactor them into a reusable function.

3. Ensure the output remains the same as the original code.

4. Add proper docstrings to all functions.

Legacy Code:

print("Area of Rectangle:", 5 * 10)

print("Perimeter of Rectangle:", 2 * (5 + 10))

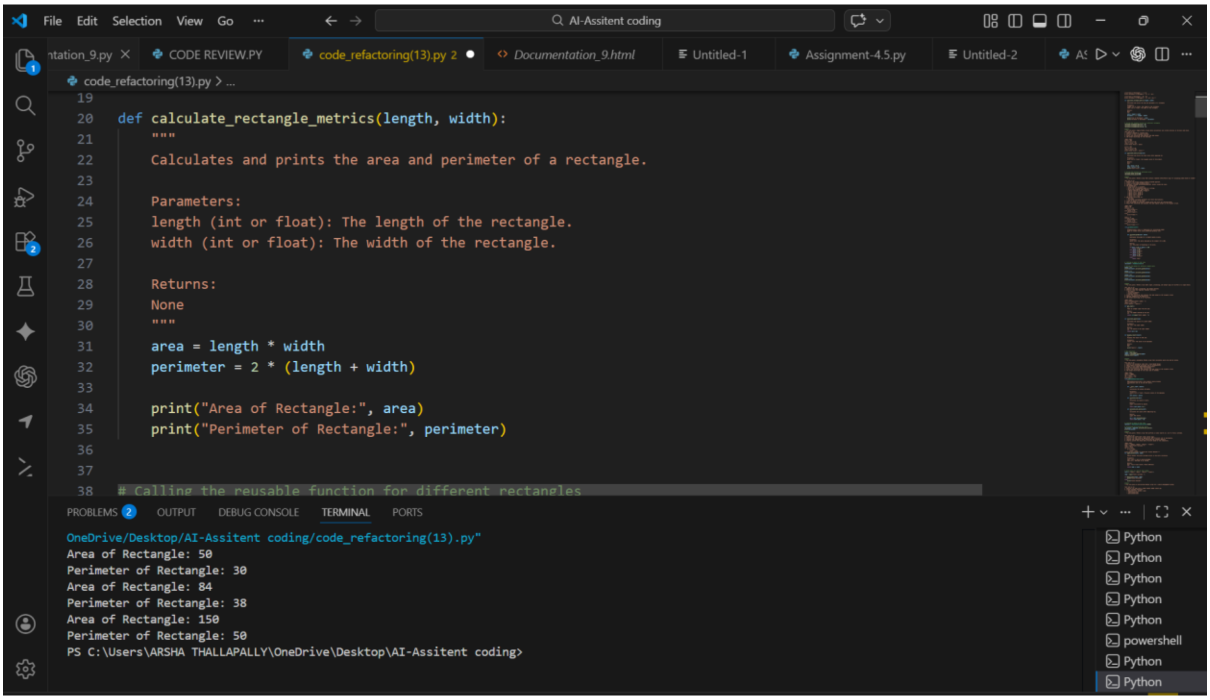
print("Area of Rectangle:", 7 * 12)

print("Perimeter of Rectangle:", 2 * (7 + 12))

print("Area of Rectangle:", 10 * 15)

print("Perimeter of Rectangle:", 2 * (10 + 15))"""

CODE:



OBSERVATION:

Duplicate logic is removed by introducing reusable functions.
Code becomes easier to maintain and modify in one place.
Output remains unchanged while structure improves.

TASK-2

PROMPT:

You are given a legacy Python script where calculations are written directly in the main code block.

Your task is to:

1. Identify repeated or related logic.

2. Extract it into a reusable function.

3. Ensure the refactored code produces the same output.

4. Add proper docstrings for the function.

Legacy Code:

price = 250

tax = price * 0.18

total = price + tax

print("Total Price:", total)

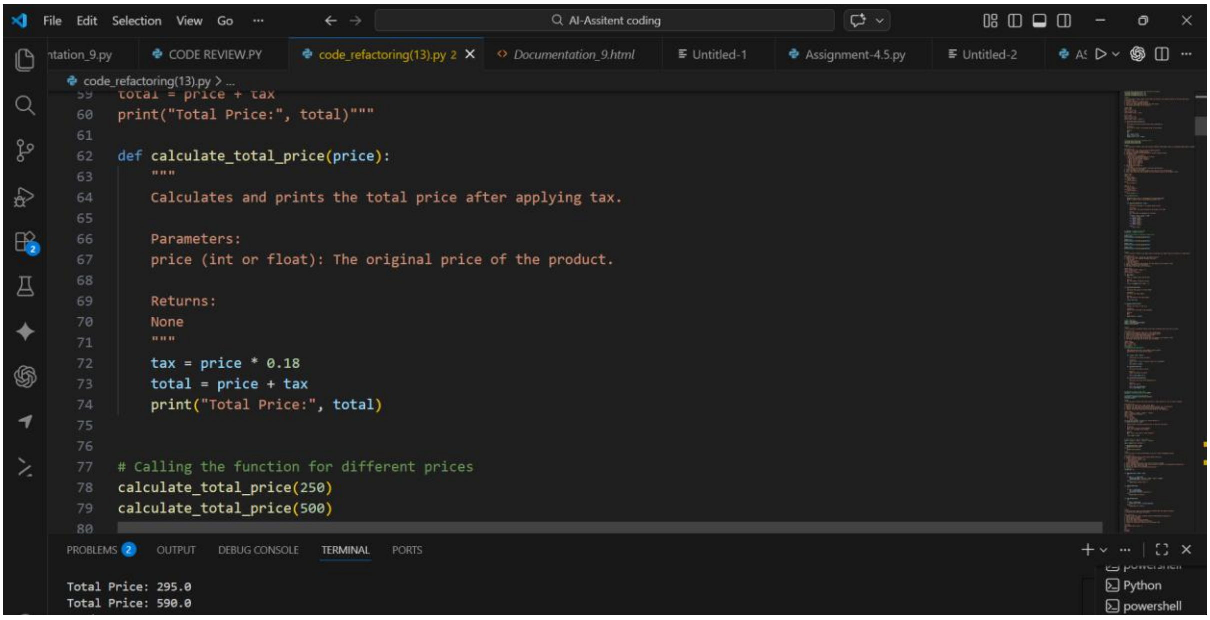
price = 500

tax = price * 0.18

total = price + tax

print("Total Price:", total)

CODE:



OBSERVATION:

Inline calculations are extracted into a reusable function.
Logic and output handling are clearly separated.
Code becomes more readable and testable.

TASK-3

PROMPT:

"""You are given a Python script that contains repeated if–elif–else logic for assigning grades based on student marks.

Your task is to:

1. Refactor the script using an object-oriented approach.

2. Create a class named GradeCalculator.

3. Implement a method calculate_grade(self, marks) inside the class.

4. The method must:

   - Accept marks as a parameter.

   - Return the corresponding grade as a string.

   - Follow this grading logic exactly:

     • Marks ≥ 90 and ≤ 100 → "Grade A"

     • Marks ≥ 80 → "Grade B"

     • Marks ≥ 70 → "Grade C"

     • Marks ≥ 40 → "Grade D"

     • Marks ≥ 0 → "Fail"

5. Add proper docstrings for:

   - The class

   - The method (including parameter and return descriptions).

6. Create an object of the class.

7. Call the method for different student marks and print the returned grade.

8. Ensure the refactored code produces the same logical output as the original script.

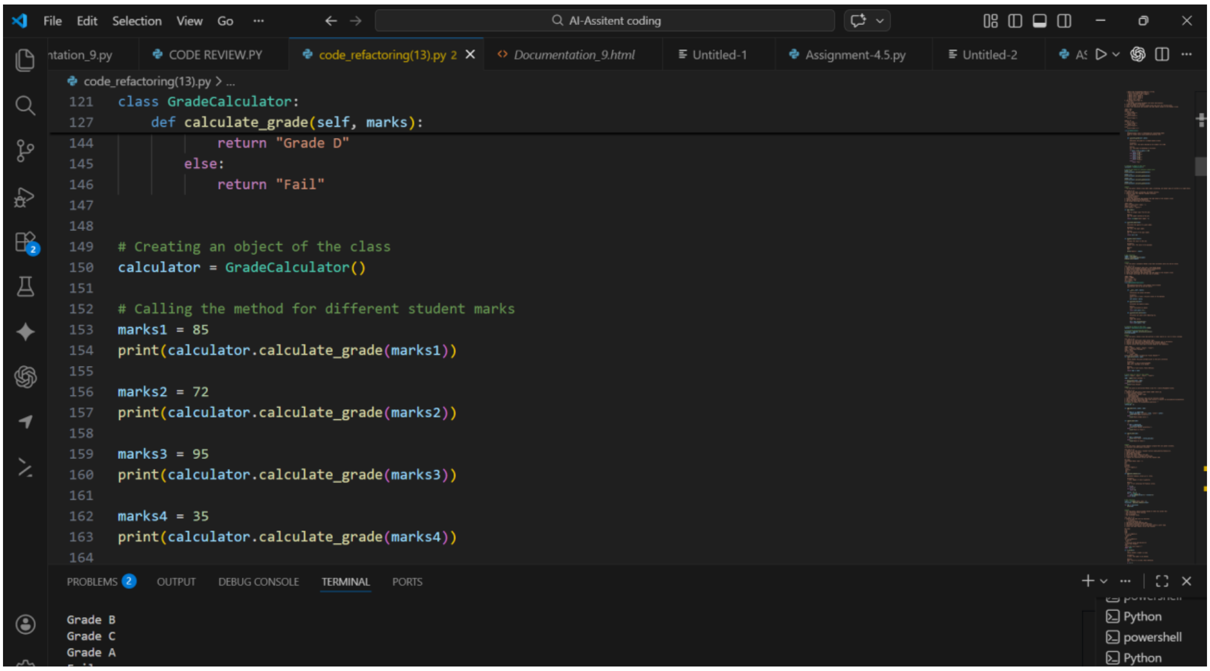
Legacy Code:

marks = 85

if marks >= 90:

```
    print("Grade A")
elif marks >= 75:
    print("Grade B")
else:
    print("Grade C")


marks = 72
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
else:
    print("Grade C")"""
```

CODE:



OBSERVATION:

Repeated conditional grading logic is centralized into a single method.
Object-oriented design improves modularity and reusability.
Future grading rule changes require modification in only one place.

TASK-4

PROMPT:

"""You are given a Python script where input, processing, and output logic are written in a single block.

Your task is to:

1. Identify the input, processing, and output sections.

2. Refactor them into separate reusable functions:

   - get_input()

   - calculate_square()

   - display_result()

3. Ensure the refactored code produces the same output as the original script.

4. Improve readability and modularity.

5. Add proper docstrings to all functions.

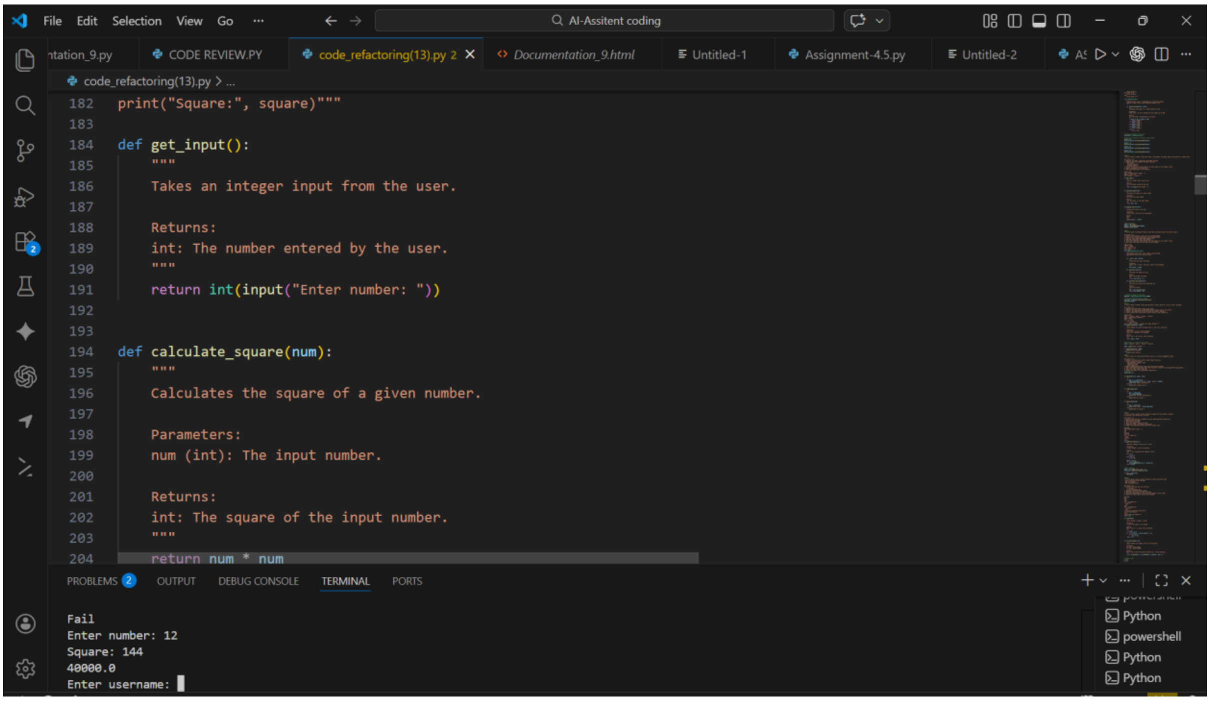
Legacy Code:

num = int(input("Enter number: "))

square = num * num

print("Square:", square)"""

CODE:



OBSERVATION:

Input, processing, and output responsibilities are separated into functions.

Readability improves by giving meaningful function names.

Behavior remains the same while structure becomes modular.

TASK-5

PROMPT:

"""You are given a procedural Python script that calculates salary tax and net salary.

Your task is to:

1. Refactor the procedural code into a class-based design.
2. Apply object-oriented principles such as encapsulation.
3. Create a class named EmployeeSalaryCalculator.
4. Move the calculation logic into methods.
5. Ensure the refactored code produces the same output as the original script.
6. Add proper docstrings for the class and its methods.

Legacy Code:

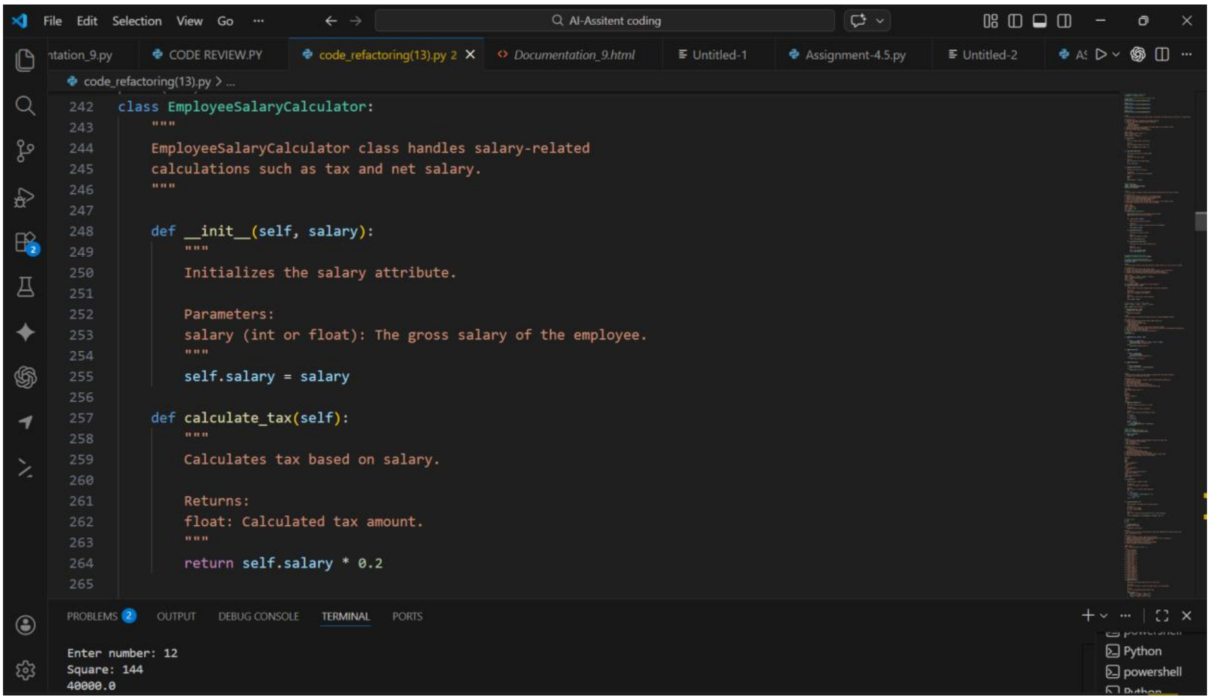
salary = 50000

tax = salary * 0.2

net = salary - tax

print(net)"""

CODE:



OBSERVATION:

Procedural salary logic is converted into a class-based design.
Encapsulation improves data handling and security.
Code becomes scalable for multiple employees.

TASK-6

PROMPT:

"""You are given a Python script that performs a linear search on a list to find a username.

Your task is to:

1. Identify the inefficient linear search logic.

2. Refactor the code using an appropriate data structure (set or dictionary).

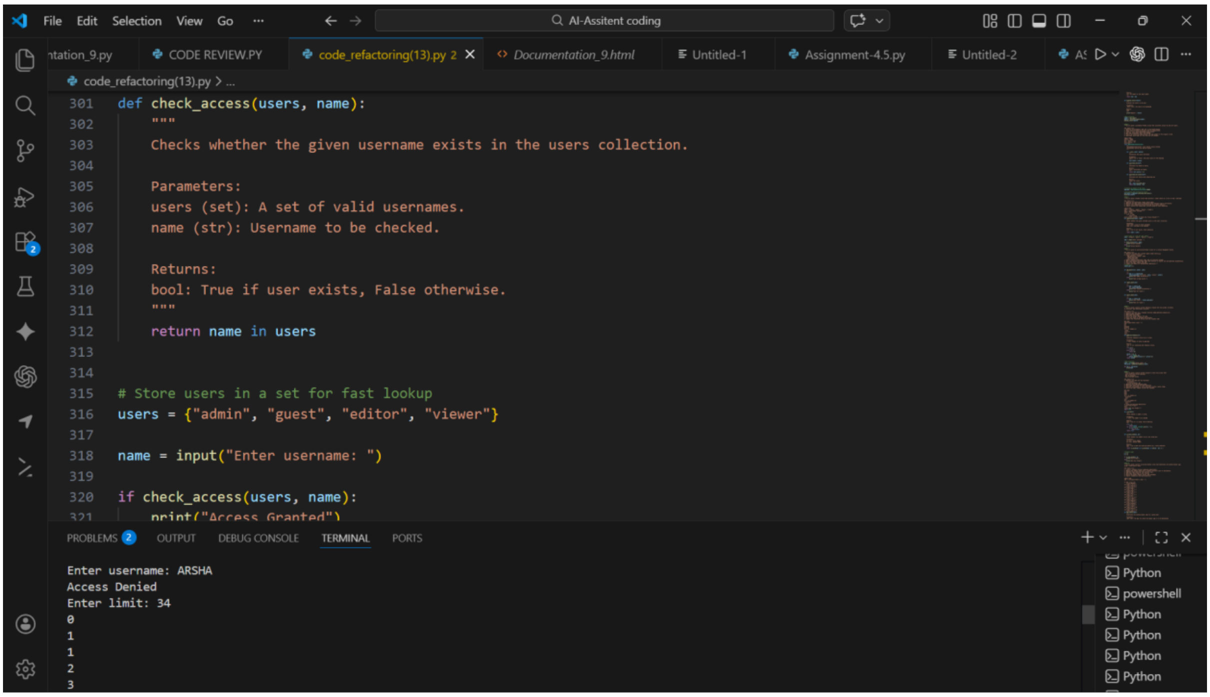
3. Improve time complexity while preserving the original behavior.

4. Clearly justify the chosen data structure based on time complexity.

Legacy Code:

```
users = ["admin", "guest", "editor", "viewer"]

name = input("Enter username: ")

found = False

for u in users:

    if u == name:

        found = True

print("Access Granted" if found else "Access Denied")"""
```

CODE:



OBSERVATION:

Linear search is replaced with faster data structure lookup.
Time complexity is reduced from O(n) to O(1) on average.
Logic becomes simpler and more efficient.

TASK-7

PROMPT:

"""You are given an unstructured Python script for a Library Management System.

Your task is to:

1. Refactor the code into a proper module named library.py.

2. Create reusable functions:

   - add_book(title, author, isbn)

   - remove_book(isbn)

   - search_book(isbn)

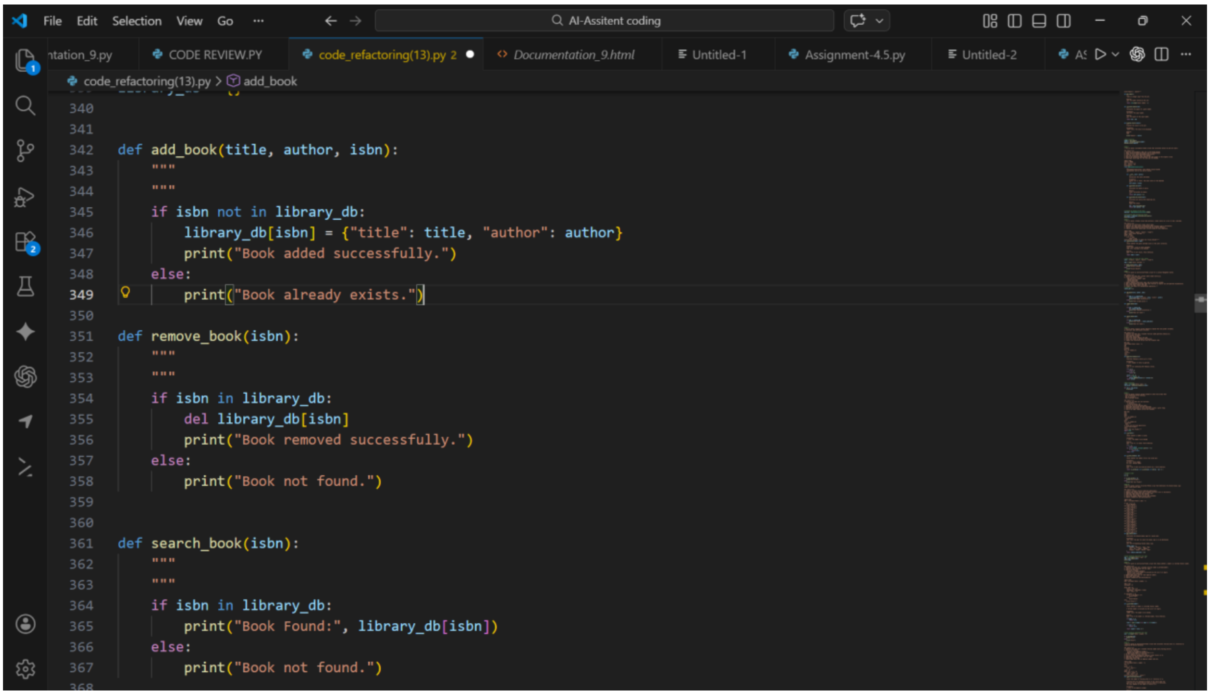
3. Remove repeated conditional logic and use functions instead.

4. Add triple-quoted docstrings under each function so Copilot can auto-generate documentation.

5. Ensure the logic remains the same.

6. Prepare the module for documentation generation."""

CODE:



```python
def add_book(title, author, isbn):
    """
    """
    if isbn not in library_db:
        library_db[isbn] = {"title": title, "author": author}
        print("Book added successfully.")
    else:
        print("Book already exists.")

def remove_book(isbn):
    """
    """
    if isbn in library_db:
        del library_db[isbn]
        print("Book removed successfully.")
    else:
        print("Book not found.")


def search_book(isbn):
    """
    """
    if isbn in library_db:
        print("Book Found:", library_db[isbn])
    else:
        print("Book not found.")
```

OBSERVATION:

Repeated library operations are converted into reusable functions.

Modular design improves maintainability and clarity.

Documentation makes functions easier to understand and use.

TASK-8

PROMPT:

"""You are given a poorly written Fibonacci program that uses global variables,

no functions, and inefficient structure.

Your task is to:

1. Refactor the code into a reusable function named generate_fibonacci(n).

2. Remove global variables.

3. Add proper docstrings.

4. Ensure the output remains the same.

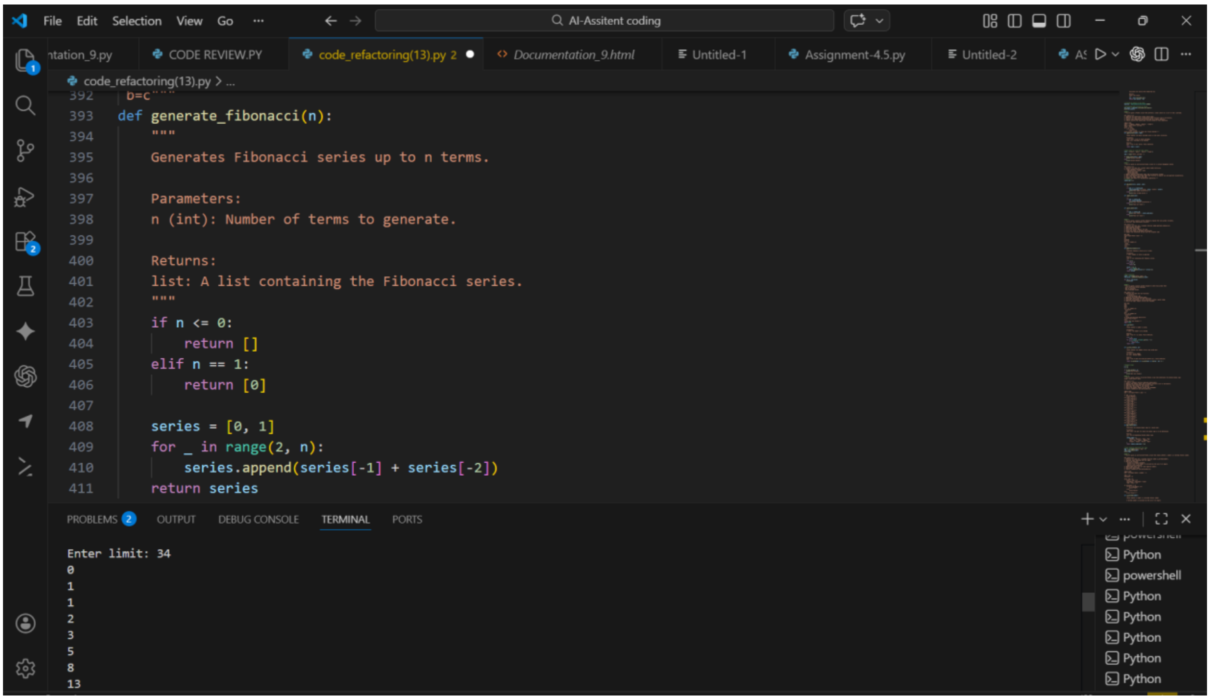
5. Write test cases to validate the function.

6. Compare the refactored version with the original code.

Bad Code:

```
n=int(input("Enter limit: "))

a=0

b=1

print(a)

print(b)

for i in range(2,n):

 c=a+b

 print(c)

 a=b

 b=c"""
```

CODE:



OBSERVATION:

Global variables are removed and the logic is encapsulated inside a function, which improves modularity and reusability.
The refactored version is easier to test because input and output are separated from the computation logic.

TASK-9

PROMPT:

"""You are given a poorly written program to check twin primes that:

- Uses inefficient prime checking.

- Has no functions.

- Uses hardcoded values.

Your task is to:

1. Refactor the code into two functions:

   - is_prime(n)

   - is_twin_prime(p1, p2)

2. Optimize the prime-checking logic.

3. Add proper docstrings for both functions.

4. Generate and display all twin prime pairs within a given range.

5. Ensure the logic remains correct and reusable.

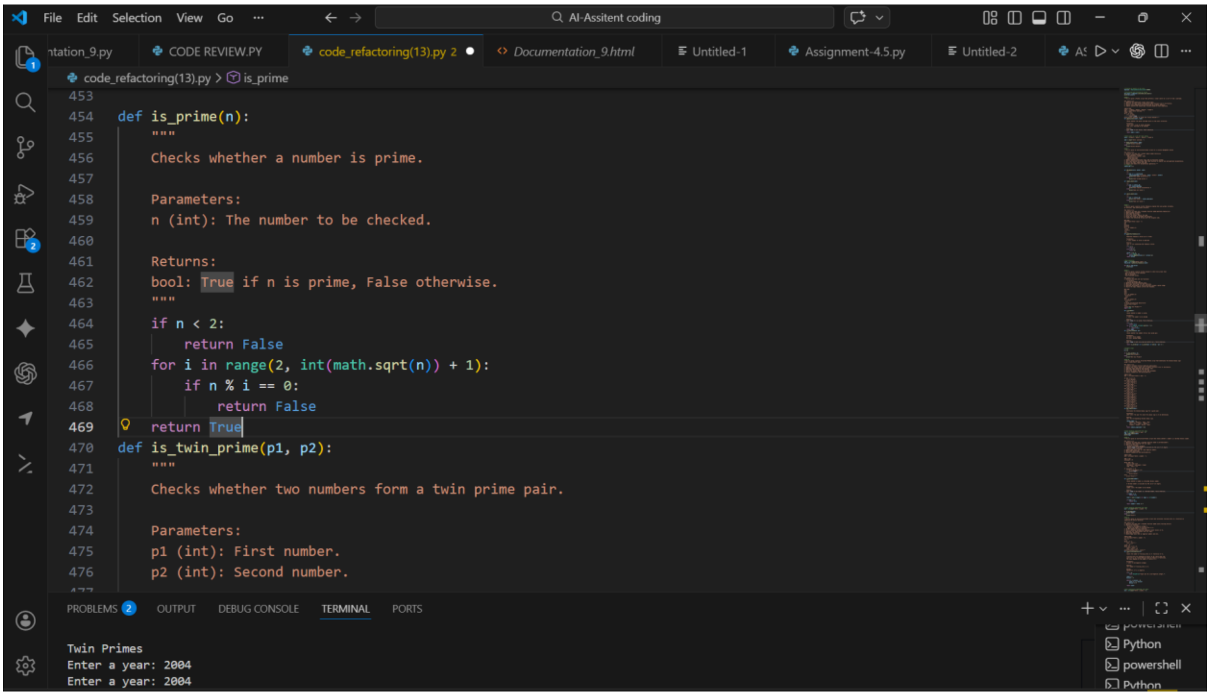
Bad Code:

```
a=11

b=13

fa=0

for i in range(2,a):

 if a%i==0:

  fa=1

fb=0

for i in range(2,b):

 if b%i==0:

  fb=1

if fa==0 and fb==0 and abs(a-b)==2:

 print("Twin Primes")
```

else:

print("Not Twin Primes")"""

CODE:



OBSERVATION:

The prime-checking logic is moved into a reusable function, which eliminates hardcoded values and repeated code.
Efficiency improves by optimizing the prime check instead of testing all numbers blindly.
Code becomes modular, readable, and easier to test for different inputs while keeping the same logical behavior.

TASK-10

PROMPT:

"""You are given a poorly structured Python script that determines the Chinese Zodiac sign

using a long if–elif chain.

Your task is to:

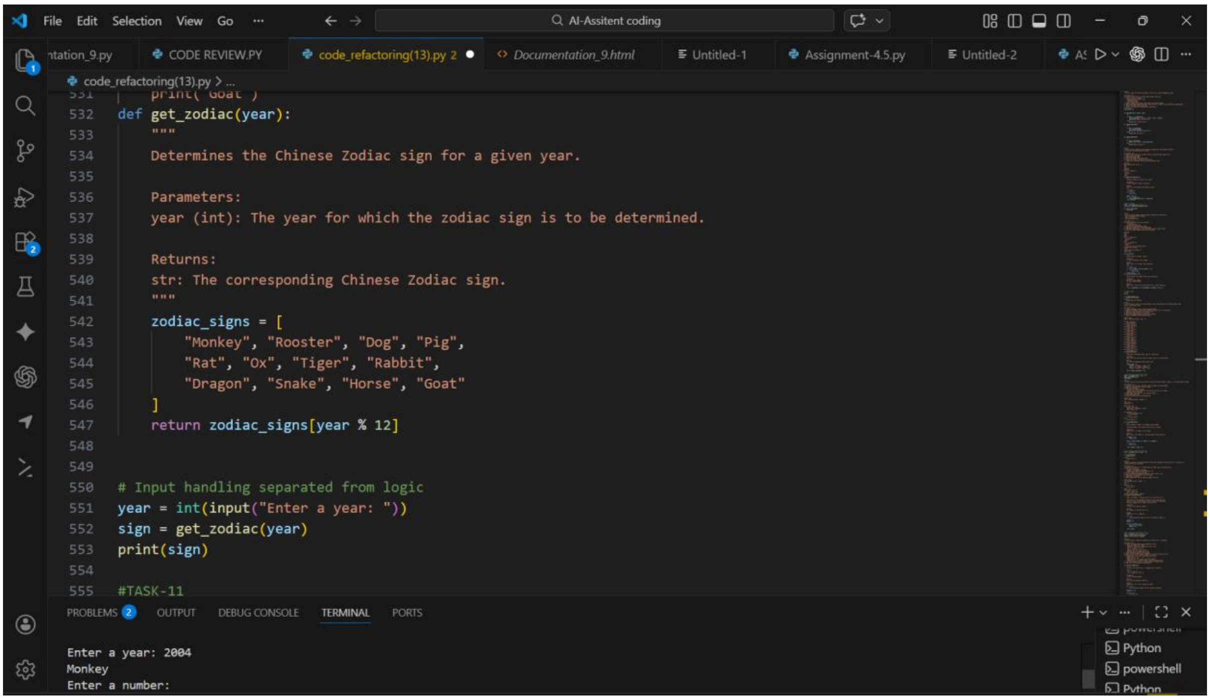
1. Create a reusable function named get_zodiac(year).
2. Replace the if–elif chain with a cleaner structure (list or dictionary).
3. Separate input handling from business logic.
4. Add proper docstrings to the function.
5. Ensure the output remains correct and unchanged.
6. Improve readability and maintainability.

Legacy Code:

```
year = int(input("Enter a year: "))


if year % 12 == 0:
   print("Monkey")
elif year % 12 == 1:
   print("Rooster")
elif year % 12 == 2:
```

```
    print("Dog")
elif year % 12 == 3:
    print("Pig")
elif year % 12 == 4:
    print("Rat")
elif year % 12 == 5:
    print("Ox")
elif year % 12 == 6:
    print("Tiger")
elif year % 12 == 7:
    print("Rabbit")
elif year % 12 == 8:
    print("Dragon")
elif year % 12 == 9:
    print("Snake")
elif year % 12 == 10:
    print("Horse")
elif year % 12 == 11:
    print("Goat")"""
```

CODE:



OBSERVATION:

The long if–elif chain is replaced with a cleaner data structure (list or dictionary), which simplifies the logic and reduces redundancy. Separating input handling from the zodiac calculation function improves modularity and makes the logic reusable and testable.

TASK-11

PROMPT:

"""You are given an unstructured Python script that checks whether a number is a Harshad (Niven) number.

Your task is to:

1. Refactor the code into a reusable function named is_harshad(number).

2. Separate input handling from core logic.

3. Ensure the function:

   - Accepts an integer parameter.

   - Returns True if the number is divisible by the sum of its digits.

   - Returns False otherwise.

4. Handle edge cases such as 0 and negative numbers.

5. Add proper docstrings.

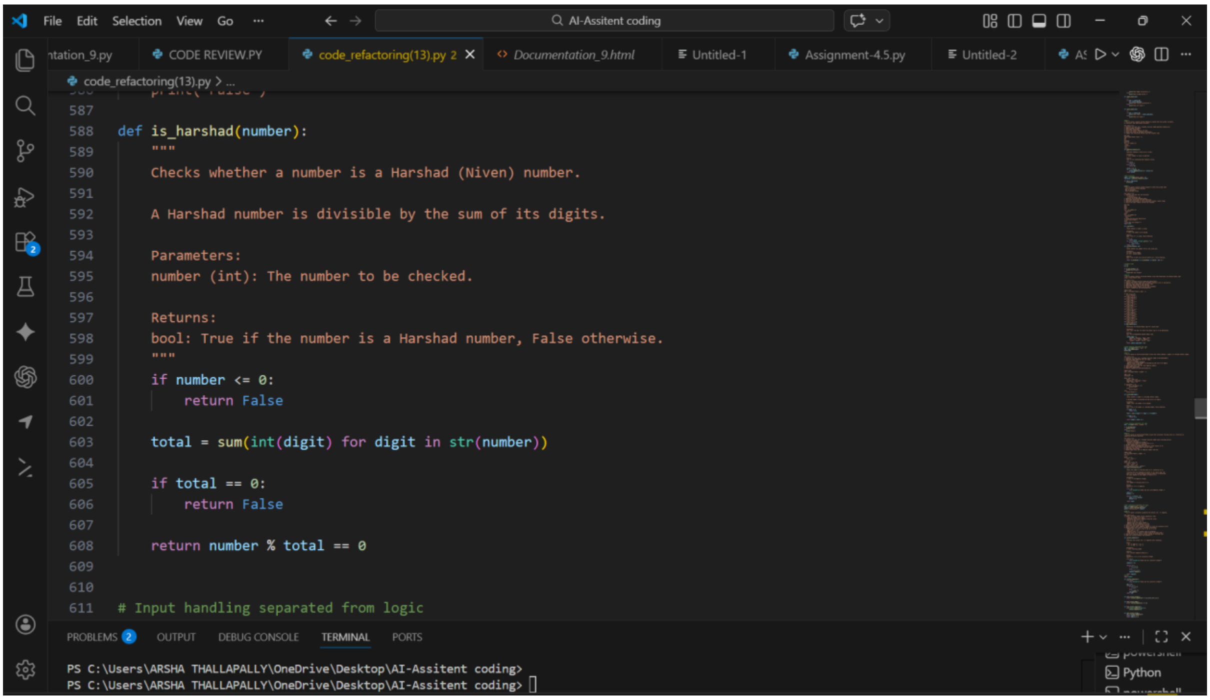
6. Improve readability and maintainability.

Legacy Code:

```
num = int(input("Enter a number: "))


temp = num

sum_digits = 0


while temp > 0:

    digit = temp % 10

    sum_digits = sum_digits + digit

    temp = temp // 10


if sum_digits != 0:
```

```
    if num % sum_digits == 0:
        print("True")
    else:
        print("False")
else:
    print("False")"""
```

CODE:



OBSERVATION:

The digit-sum and divisibility logic is moved into a reusable function, which removes redundancy and improves modularity. Separating input handling from computation makes the function easier to test and reuse in other programs.

TASK-12

PROMPT:

"""You are given an unstructured Python script that calculates trailing zeros in a factorial by

computing the entire factorial.

Your task is to:

1. Refactor the code into a reusable function named count_trailing_zeros(n).

2. Ensure the function:

 - Accepts a non-negative integer n.

 - Returns the number of trailing zeros in n!.

3. Do NOT compute the full factorial.

4. Use an optimized mathematical approach (count factors of 5).

5. Separate user input/output from core logic.

6. Add proper docstrings.

7. Handle edge cases such as negative numbers and zero.

Legacy Code:

n = int(input("Enter a number: "))

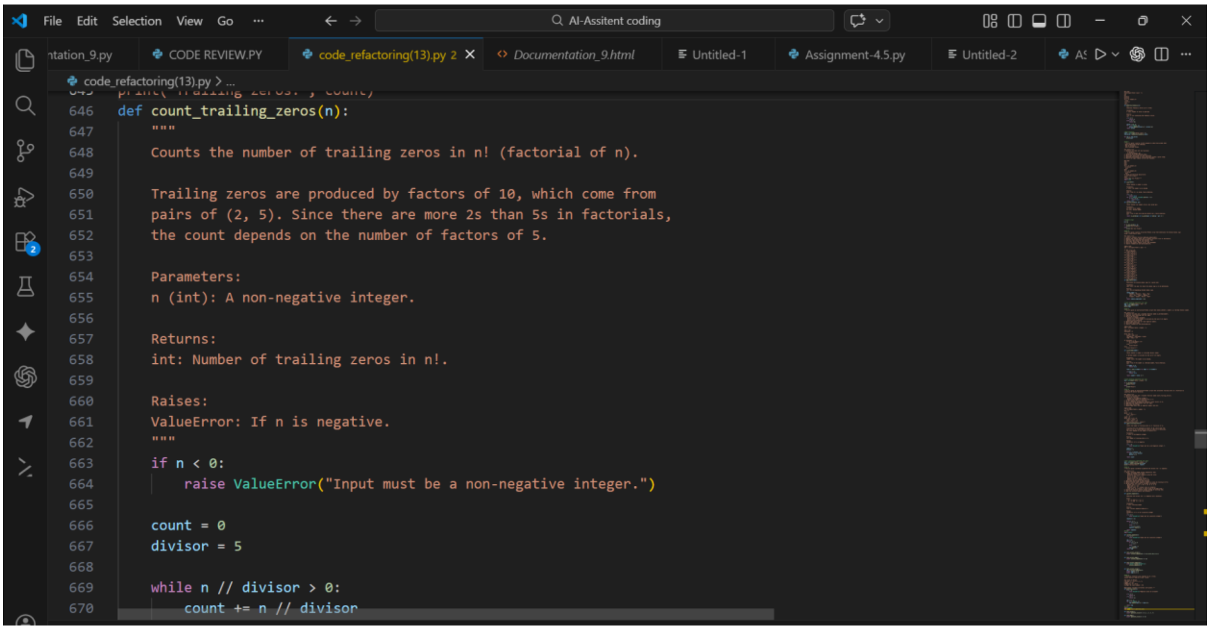
fact = 1

i = 1

while i <= n:

```
    fact = fact * i
    i = i + 1
count = 0
while fact % 10 == 0:
    count = count + 1
    fact = fact // 10
print("Trailing zeros:", count)"""
```

CODE:



```
def count_trailing_zeros(n):
    """
    Counts the number of trailing zeros in n! (factorial of n).

    Trailing zeros are produced by factors of 10, which come from
    pairs of (2, 5). Since there are more 2s than 5s in factorials,
    the count depends on the number of factors of 5.

    Parameters:
    n (int): A non-negative integer.

    Returns:
    int: Number of trailing zeros in n!.

    Raises:
    ValueError: If n is negative.
    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")

    count = 0
    divisor = 5

    while n // divisor > 0:
        count += n // divisor
```

OBSERVATION:

The inefficient full factorial computation is replaced with an optimized mathematical approach using powers of 5, which greatly improves performance.
Core logic is encapsulated inside a reusable function, making the code modular and easier to test.

Readability and maintainability improve while ensuring the output remains correct and unchanged.

TASK-13

PROMPT:

"""You are given a problem to generate the Collatz (3n + 1) sequence.

Your task is to:

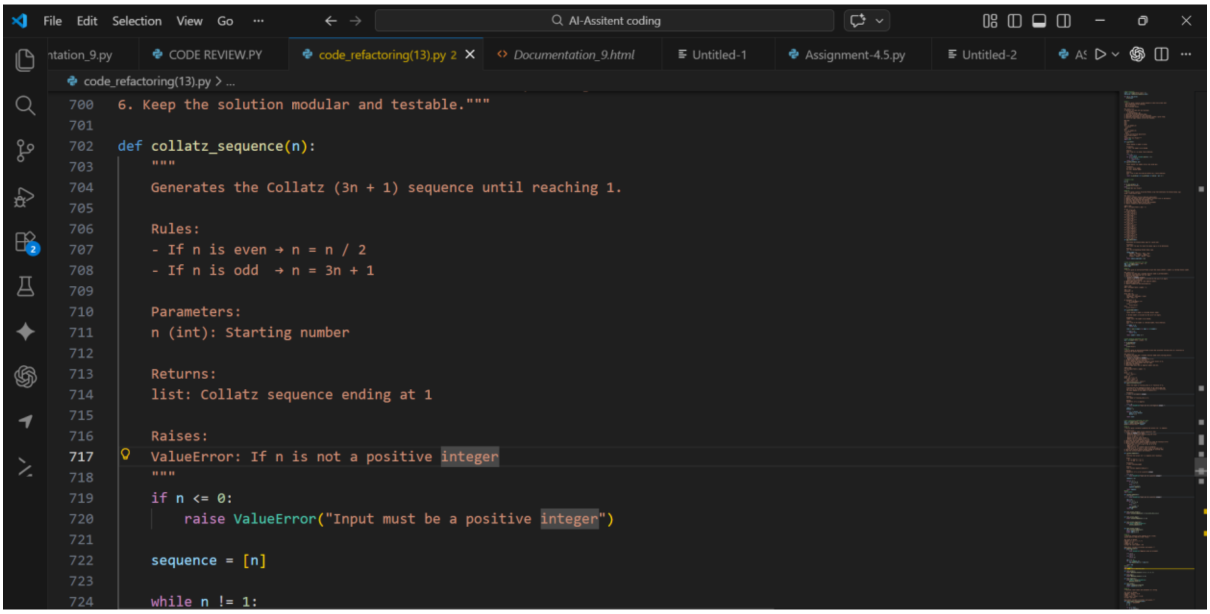
1. Write a function named collatz_sequence(n) that:
   - Takes an integer n as input.
   - Generates the Collatz sequence using the rules:
     - If n is even → n = n / 2
     - If n is odd → n = 3n + 1
   - Repeats until the value reaches 1.
   - Returns the full sequence as a list.
2. Add proper docstrings to the function.
3. Handle invalid input (negative numbers or zero) by raising an error.
4. Design pytest test cases to validate correctness:
   - Normal case: 6 → [6, 3, 10, 5, 16, 8, 4, 2, 1]
   - Edge case: 1 → [1]

- Negative case: -5 → should raise an exception
- Large case: 27 → verify it starts with 27 and ends with 1

5. Ensure the function returns values instead of printing them.

6. Keep the solution modular and testable."""

CODE:



OBSERVATION:

The sequence generation logic is encapsulated inside a reusable function, which removes dependency on inline or procedural code.
Input validation improves reliability by handling negative and invalid values explicitly.

TASK-14

PROMPT:

"""Function: Generate Lucas sequence up to n terms.

(Starts with 2,1 then Fn = Fn-1 + Fn-2)

Test Cases to Design:

• Normal: 5 → [2, 1, 3, 4, 7]

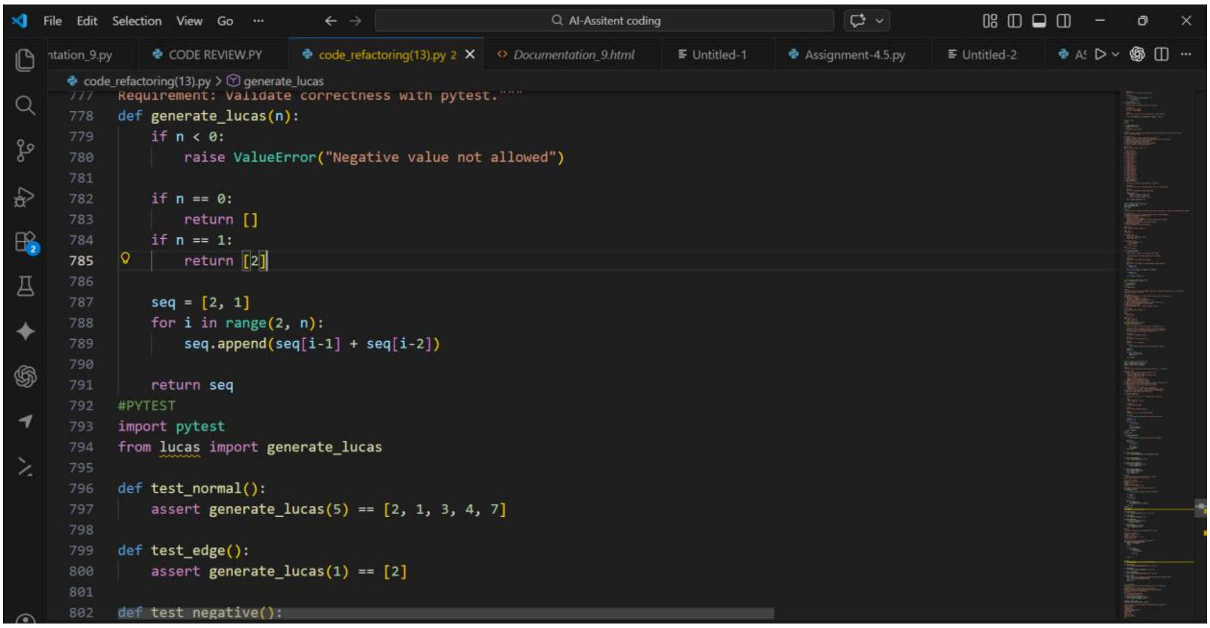
• Edge: 1 → [2]

• Negative: -5 → Error

• Large: 10 (last element = 76)

Requirement: Validate correctness with pytest."""

CODE:



OBSERVATION:

The sequence generation logic is moved into a reusable function, which improves modularity and avoids repeated code.
Edge and negative input cases are handled explicitly, increasing reliability and robustness of the program.

TASK-15

PROMPT:

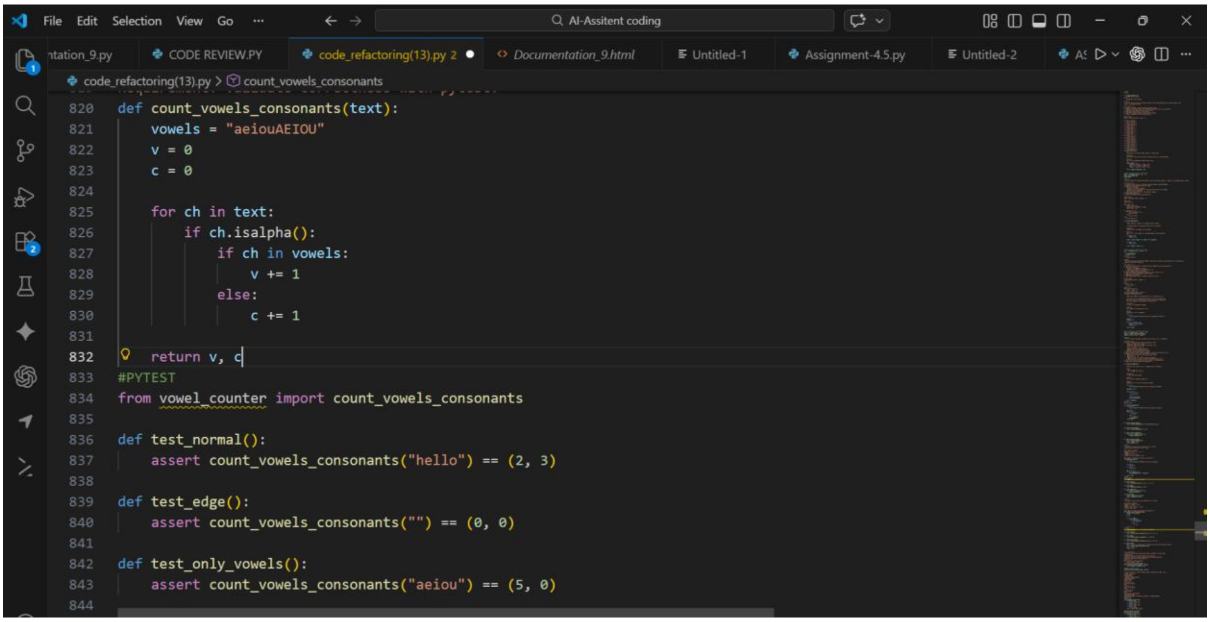
Function: Count vowels and consonants in a string.

Test Cases to Design:

- Normal: "hello" → (2,3)
- Edge: "" → (0,0)
- Only vowels: "aeiou" → (5,0)
- Large: Long text

Requirement: Validate correctness with pytest."""

CODE:



```python
def count_vowels_consonants(text):
    vowels = "aeiouAEIOU"
    v = 0
    c = 0

    for ch in text:
        if ch.isalpha():
            if ch in vowels:
                v += 1
            else:
                c += 1

    return v, c
#PYTEST
from vowel_counter import count_vowels_consonants

def test_normal():
    assert count_vowels_consonants("hello") == (2, 3)

def test_edge():
    assert count_vowels_consonants("") == (0, 0)

def test_only_vowels():
    assert count_vowels_consonants("aeiou") == (5, 0)
```

OBSERVATION:

String processing logic is encapsulated inside a reusable function, improving modularity and separation of concerns.
Edge cases such as empty strings and all-vowel inputs are handled correctly, increasing robustness.
Test cases ensure accurate counting of vowels and consonants while preserving the expected behavior.