

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:

The screenshot shows a code editor interface with multiple tabs. The active tab is 'code_refactoring(13).py'. The code defines a function 'calculate_rectangle_metrics' that calculates the area and perimeter of a rectangle. It includes detailed docstrings with parameters, returns, and examples. The output panel shows the execution of the script and its results.

```
19
20 def calculate_rectangle_metrics(length, width):
21     """
22         Calculates and prints the area and perimeter of a rectangle.
23
24     Parameters:
25     length (int or float): The length of the rectangle.
26     width (int or float): The width of the rectangle.
27
28     Returns:
29     None
30     """
31     area = length * width
32     perimeter = 2 * (length + width)
33
34     print("Area of Rectangle:", area)
35     print("Perimeter of Rectangle:", perimeter)
36
37
38 # Calling the reusable function for different rectangles
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
OneDrive\Desktop\AI-Assitant coding\code_refactoring(13).py"
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
PS C:\Users\ARSHA THALLAPALLY\OneDrive\Desktop\AI-Assitant coding>
```

Python Python Python Python Python powershell Python Python

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:

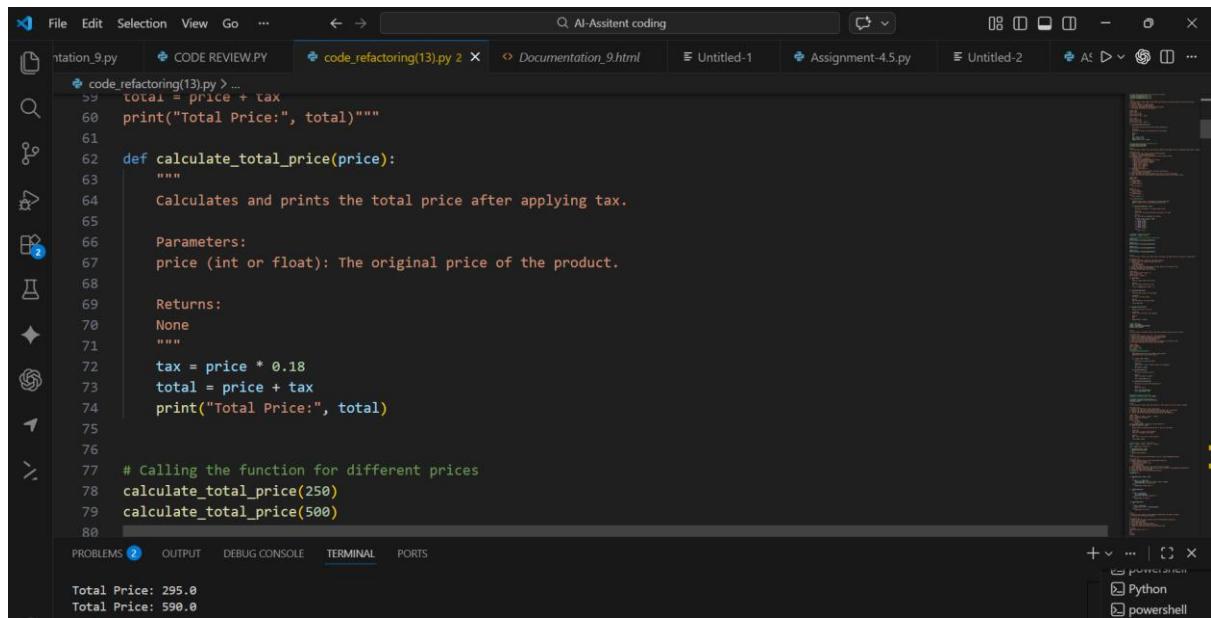
1. Identify repeated or related logic.
2. Extract it into a reusable function.
3. Ensure the refactored code produces the same output.
4. Add proper docstrings for the function.

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:



```
ntation_9.py |  CODE REVIEW.PY |  code_refactoring(13).py 2 |  Documentation_9.html |  Untitled-1 |  Assignment-4.5.py |  Untitled-2 |  AI Assistant |  ...  
code_refactoring(13).py > ...  
59     total = price + tax  
60     print("Total Price:", total)"""  
61  
62     def calculate_total_price(price):  
63         """  
64             Calculates and prints the total price after applying tax.  
65         """  
66         Parameters:  
67         price (int or float): The original price of the product.  
68  
69         Returns:  
70         None  
71         """  
72         tax = price * 0.18  
73         total = price + tax  
74         print("Total Price:", total)  
75  
76  
77     # Calling the function for different prices  
78     calculate_total_price(250)  
79     calculate_total_price(500)  
80  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  
Total Price: 295.0  
Total Price: 590.0
```

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 $\leq 100 \rightarrow$ "Grade A"
 - Marks $\geq 80 \rightarrow$ "Grade B"
 - Marks $\geq 70 \rightarrow$ "Grade C"
 - Marks $\geq 40 \rightarrow$ "Grade D"
 - Marks $\geq 0 \rightarrow$ "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:

The screenshot shows a code editor interface with multiple tabs open. The active tab contains Python code for a `GradeCalculator` class. The code defines a method `calculate_grade` that returns "Grade D" if marks are less than or equal to 35, "Grade C" if marks are between 36 and 74, "Grade B" if marks are between 75 and 90, and "Fail" if marks are greater than 90. It then creates an object of the class and calls the method for four different student marks (85, 72, 95, and 35), printing the results. The bottom right corner shows a terminal window with the output: "Grade B", "Grade C", "Grade A", and "Fail".

```
ntation_9.py | CODE REVIEW.PY | code_refactoring(13).py 2 | Documentation_9.html | Untitled-1 | Assignment-4.5.py | Untitled-2 | AS D ...  
121     class GradeCalculator:  
122         def calculate_grade(self, marks):  
144             return "Grade D"  
145         else:  
146             return "Fail"  
147  
148  
149     # Creating an object of the class  
150 calculator = GradeCalculator()  
151  
152     # Calling the method for different student marks  
153 marks1 = 85  
154 print(calculator.calculate_grade(marks1))  
155  
156 marks2 = 72  
157 print(calculator.calculate_grade(marks2))  
158  
159 marks3 = 95  
160 print(calculator.calculate_grade(marks3))  
161  
162 marks4 = 35  
163 print(calculator.calculate_grade(marks4))  
164  
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS  
Grade B  
Grade C  
Grade A  
Fail
```

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:

The screenshot shows a code editor interface with several tabs at the top: 'ntation_9.py', 'CODE REVIEW.PY', 'code_refactoring(13).py 2', 'Documentation_9.html', 'Untitled-1', 'Assignment-4.5.py', 'Untitled-2', and 'AI-Assistant coding'. The main pane displays the following Python code:

```
182     print("Square:", square)"""
183
184     def get_input():
185         """
186             Takes an integer input from the user.
187
188             Returns:
189             int: The number entered by the user.
190
191         return int(input("Enter number: "))
192
193
194     def calculate_square(num):
195         """
196             Calculates the square of a given number.
197
198             Parameters:
199             num (int): The input number.
200
201             Returns:
202             int: The square of the input number.
203
204         return num * num
```

Below the code, the terminal window shows the following output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Fail
Enter number: 12
Square: 144
40000.0
Enter username: [REDACTED]
```

The bottom right corner shows a sidebar with several Python environments listed.

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:

```
ntation_9.py CODE REVIEW.PY code_refactoring(13).py 2 Documentation_9.html Untitled-1 Assignment-4.5.py Untitled-2 AS D ...  
242     class EmployeeSalaryCalculator:  
243         """  
244             EmployeeSalaryCalculator class handles salary-related  
245             calculations such as tax and net salary.  
246         """  
247  
248     def __init__(self, salary):  
249         """  
250             Initializes the salary attribute.  
251  
252             Parameters:  
253             salary (int or float): The gross salary of the employee.  
254         """  
255         self.salary = salary  
256  
257     def calculate_tax(self):  
258         """  
259             Calculates tax based on salary.  
260  
261             Returns:  
262             float: Calculated tax amount.  
263         """  
264         return self.salary * 0.2  
265  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS  
Enter number: 12  
Square: 144  
40000.0
```

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

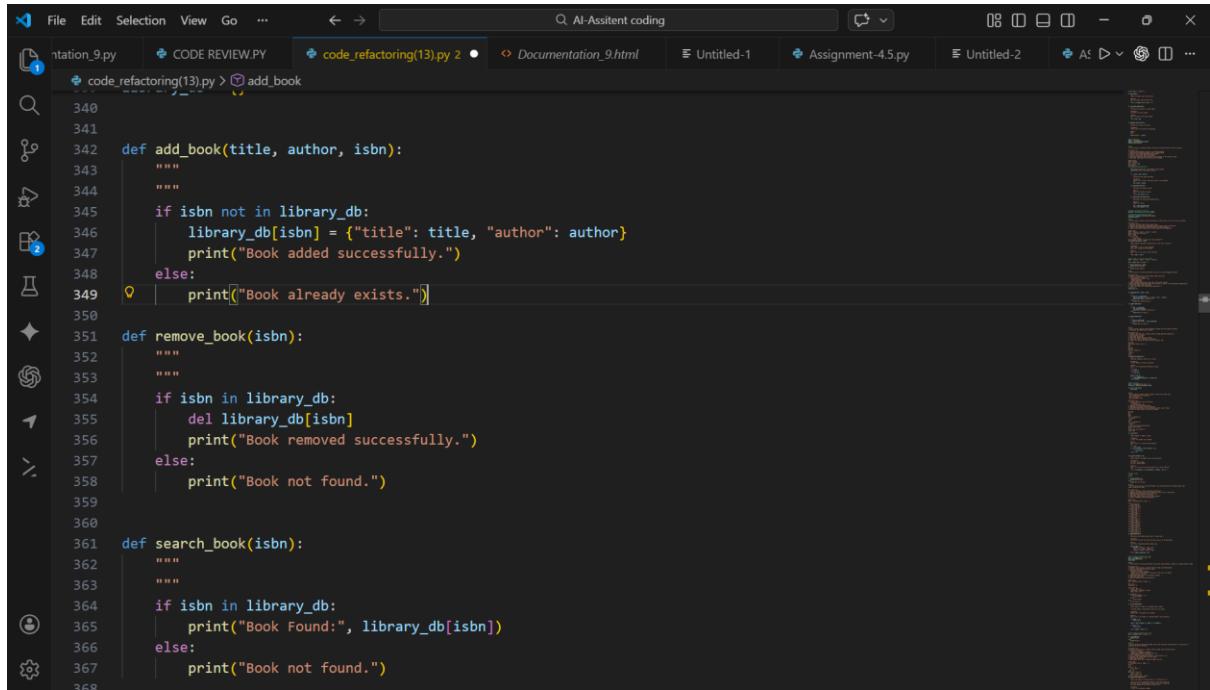
PROMPT:

"""\nYou 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:



The screenshot shows a code editor window titled "AI-Assistant coding". The main pane displays Python code for library operations. The code includes functions for adding, removing, and searching books based on ISBN. The code is well-documented with triple quotes at the start of each function definition.

```
340
341
342 def add_book(title, author, isbn):
343     """
344     """
345     if isbn not in library_db:
346         library_db[isbn] = {"title": title, "author": author}
347         print("Book added successfully.")
348     else:
349         print("Book already exists.")
350
351 def remove_book(isbn):
352     """
353     """
354     if isbn in library_db:
355         del library_db[isbn]
356         print("Book removed successfully.")
357     else:
358         print("Book not found.")
359
360
361 def search_book(isbn):
362     """
363     """
364     if isbn in library_db:
365         print("Book Found:", library_db[isbn])
366     else:
367         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):

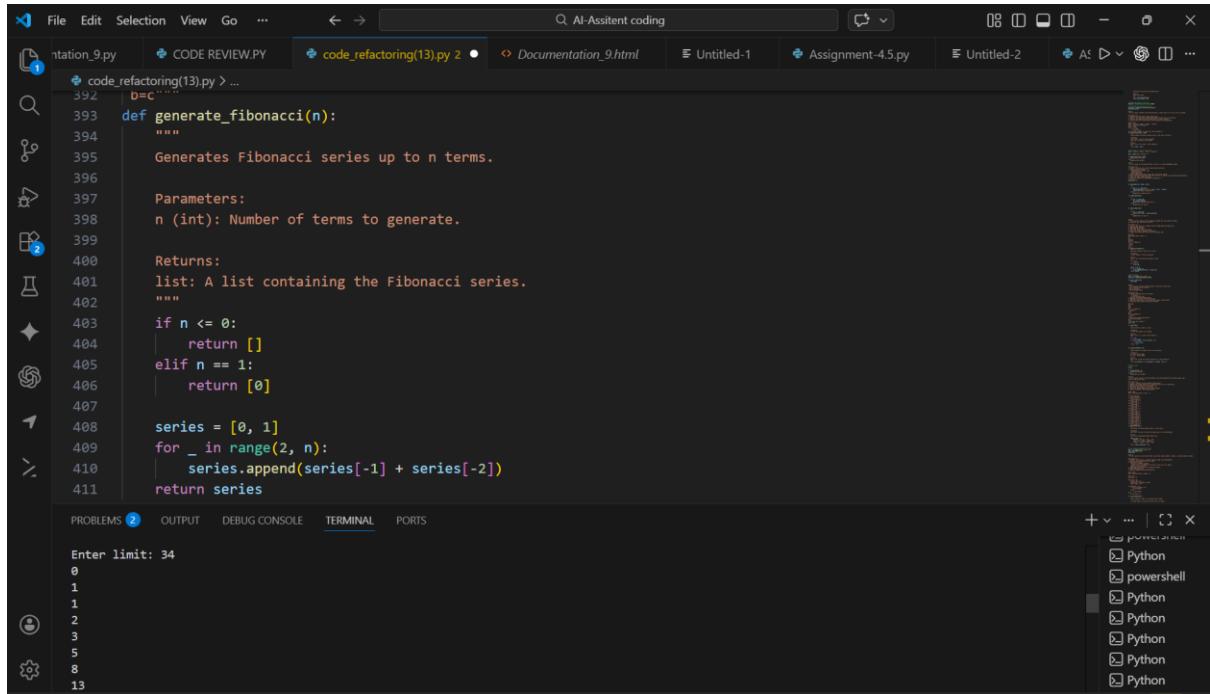
    c=a+b

    print(c)

    a=b

    b=c""""
```

CODE:



The screenshot shows a code editor interface with several tabs at the top: 'Presentation_9.py', 'CODE REVIEW.PY', 'code_refactoring(13).py 2' (which is the active tab), 'Documentation_9.html', 'Untitled-1', 'Assignment-4.5.py', 'Untitled-2', and 'Assignment-4.5.py'. The code in the 'code_refactoring(13).py 2' tab is as follows:

```
392     b=c
393 def generate_fibonacci(n):
394     """
395         Generates Fibonacci series up to n terms.
396
397     Parameters:
398         n (int): Number of terms to generate.
399
400     Returns:
401         list: A list containing the Fibonacci series.
402     """
403     if n <= 0:
404         return []
405     elif n == 1:
406         return [0]
407
408     series = [0, 1]
409     for _ in range(2, n):
410         series.append(series[-1] + series[-2])
411
412     return series
```

The 'TERMINAL' tab is selected, showing the output of the code execution:

```
Enter limit: 34
0
1
1
2
3
5
8
13
```

The right sidebar shows a file tree with several Python files and a powershell file.

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:

The screenshot shows a code editor interface with multiple tabs open. The active tab is 'code_refactoring(13).py' which contains the following Python code:

```
def is_prime(n):  
    """  
    Checks whether a number is prime.  
    Parameters:  
    n (int): The number to be checked.  
    Returns:  
    bool: True if n is prime, False otherwise.  
    """  
    if n < 2:  
        return False  
    for i in range(2, int(math.sqrt(n)) + 1):  
        if n % i == 0:  
            return False  
    return True  
def is_twin_prime(p1, p2):  
    """  
    Checks whether two numbers form a twin prime pair.  
    Parameters:  
    p1 (int): First number.  
    p2 (int): Second number.  
    """
```

The code editor has a dark theme and includes a sidebar with various icons. At the bottom, there is a terminal window showing the output of running the script with the command 'Twin Primes'. The terminal output is:

```
Twin Primes  
Enter a year: 2004  
Enter a year: 2004
```

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.

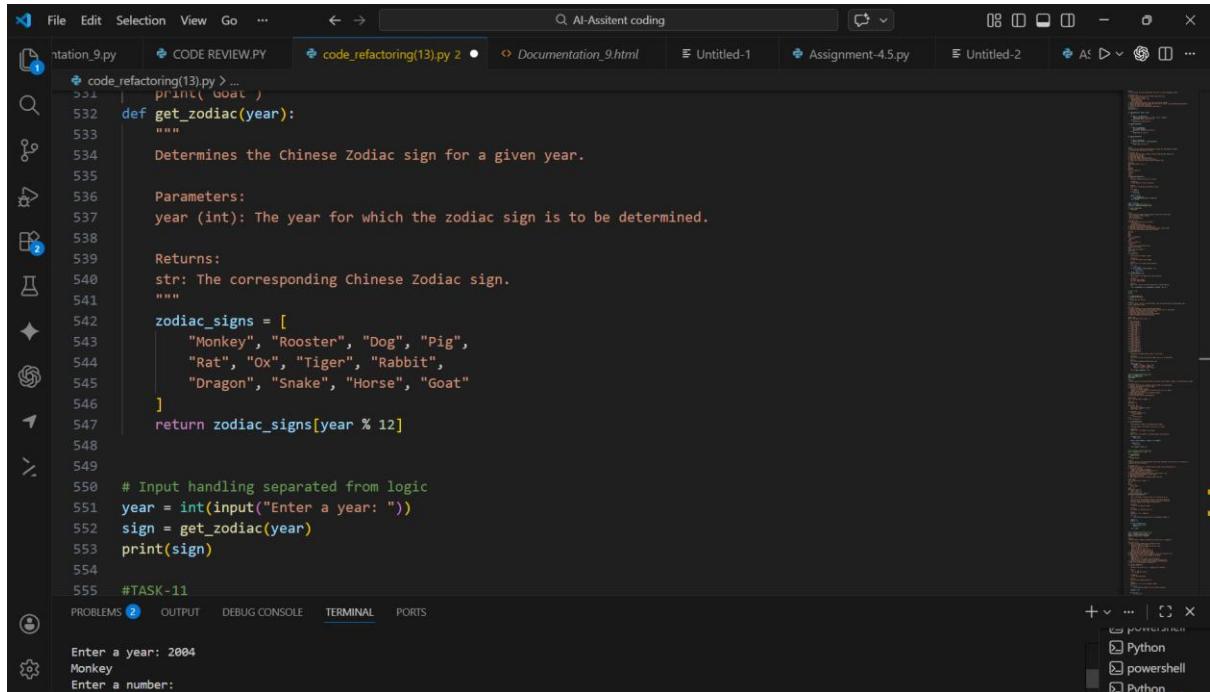
Legacy Code:

```
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:



The screenshot shows a code editor interface with multiple tabs open. The active tab is 'code_refactoring(13).py' containing the following Python code:

```
def get_zodiac(year):
    """
    Determines the Chinese Zodiac sign for a given year.

    Parameters:
    year (int): The year for which the zodiac sign is to be determined.

    Returns:
    str: The corresponding Chinese Zodiac sign.
    """

    zodiac_signs = [
        "Monkey", "Rooster", "Dog", "Pig",
        "Rat", "Ox", "Tiger", "Rabbit",
        "Dragon", "Snake", "Horse", "Goat"
    ]
    return zodiac_signs[year % 12]

# Input handling separated from logic
year = int(input("Enter a year: "))
sign = get_zodiac(year)
print(sign)

#TASK-11
```

The code includes a function `get_zodiac` that takes a year as input and returns the corresponding Chinese Zodiac sign. The logic for determining the sign is moved into the function, and input handling is separated into a call to `int(input())`. The code is annotated with docstrings and type hints.

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:

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

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:

The screenshot shows a Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** On the left, it shows files like `station_9.py`, `CODE REVIEW.PY`, `code_refactoring(13).py` (which is currently open), `Documentation_9.html`, `Untitled-1`, `Assignment-4.5.py`, and `Untitled-2`.
- Code Editor:** The main area displays Python code for checking if a number is a Harshad (Niven) number. The code includes docstrings, parameters, returns, and logic for summing digits and checking divisibility.
- Output Panel:** At the bottom, the output panel shows two lines of terminal output:

```
PS C:\Users\ARSHA THALAPALLY\OneDrive\Desktop\AI-Assistent coding>
PS C:\Users\ARSHA THALAPALLY\OneDrive\Desktop\AI-Assistent coding>
```
- Bottom Bar:** The bottom bar includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS, along with other standard VS Code icons.

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:

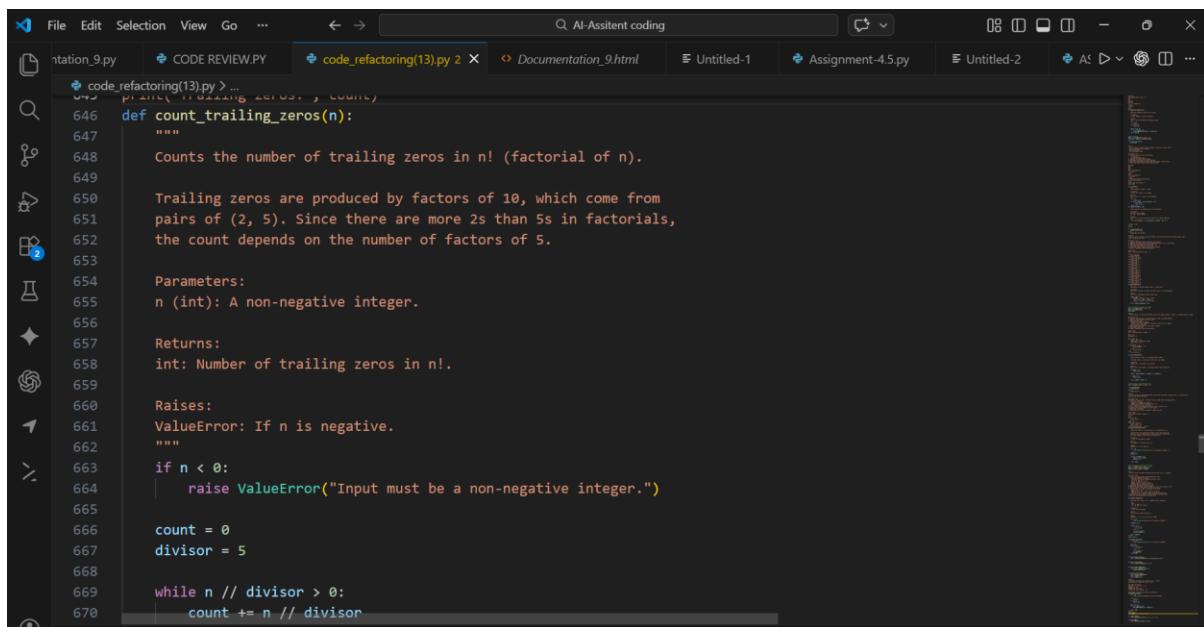
    count = count + 1

    fact = fact // 10

print("Trailing zeros:", count)"""

```

CODE:



The screenshot shows a code editor window with several tabs at the top: 'ntation_9.py', 'CODE REVIEW.PY', 'code_refactoring(13).py 2', 'Documentation_9.html', 'Untitled-1', 'Assignment-4.5.py', 'Untitled-2', and 'Assignment-4.5.py'. The main pane displays the following Python 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:

"""\nYou 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 $\rightarrow n = n / 2$
 - If n is odd $\rightarrow n = 3n + 1$
- 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:

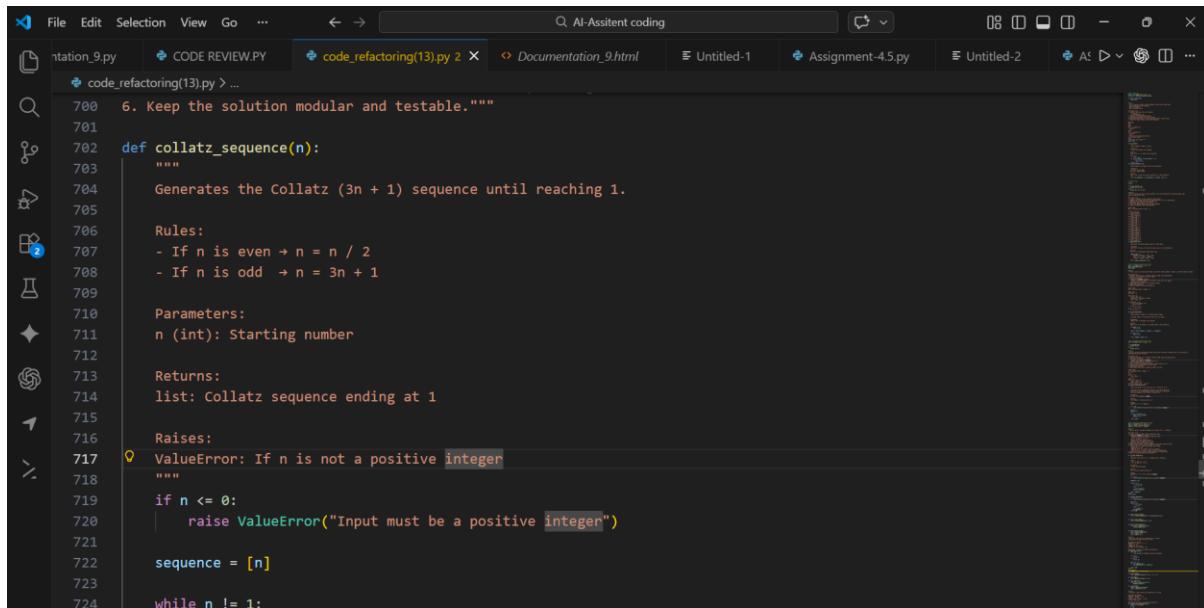
- Normal case: $6 \rightarrow [6, 3, 10, 5, 16, 8, 4, 2, 1]$
- Edge case: $1 \rightarrow [1]$

- 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:



```

File Edit Selection View Go ...
File CODE REVIEW.PY code_refactoring(13).py 2 Documentation_9.html Untitled-1 Assignment-4.py Untitled-2 AI-Assistent coding
ntation_9.py
 6. Keep the solution modular and testable.""""

def collatz_sequence(n):
    """
    Generates the Collatz (3n + 1) sequence until reaching 1.

    Rules:
    - If n is even → n = n / 2
    - If n is odd → n = 3n + 1

    Parameters:
    n (int): Starting number

    Returns:
    list: Collatz sequence ending at 1

    Raises:
    ValueError: If n is not a positive integer
    """
    if n <= 0:
        raise ValueError("Input must be a positive integer")
    sequence = [n]
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        sequence.append(n)
    return sequence

```

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 $F_n = F_{n-1} + F_{n-2}$)

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:

```
ntation_9.py CODE REVIEW.PY code_refactoring(13).py 2 Documentation_9.html Untitled-1 Assignment-4.5.py Untitled-2 AS D ...  
ntation_9.py  
code_refactoring(13).py > generate_lucas  
    """ Requirement: Validate correctness with pytest.  
    778 def generate_lucas(n):  
    779     if n < 0:  
    780         raise ValueError("Negative value not allowed")  
    781  
    782     if n == 0:  
    783         return []  
    784     if n == 1:  
    785         return [2]  
    786  
    787     seq = [2, 1]  
    788     for i in range(2, n):  
    789         seq.append(seq[i-1] + seq[i-2])  
    790  
    791     return seq  
    792 #PYTEST  
    793 import pytest  
    794 from lucas import generate_lucas  
    795  
    796 def test_normal():  
    797     assert generate_lucas(5) == [2, 1, 3, 4, 7]  
    798  
    799 def test_edge():  
    800     assert generate_lucas(1) == [2]  
    801  
    802 def test_negative():
```

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:

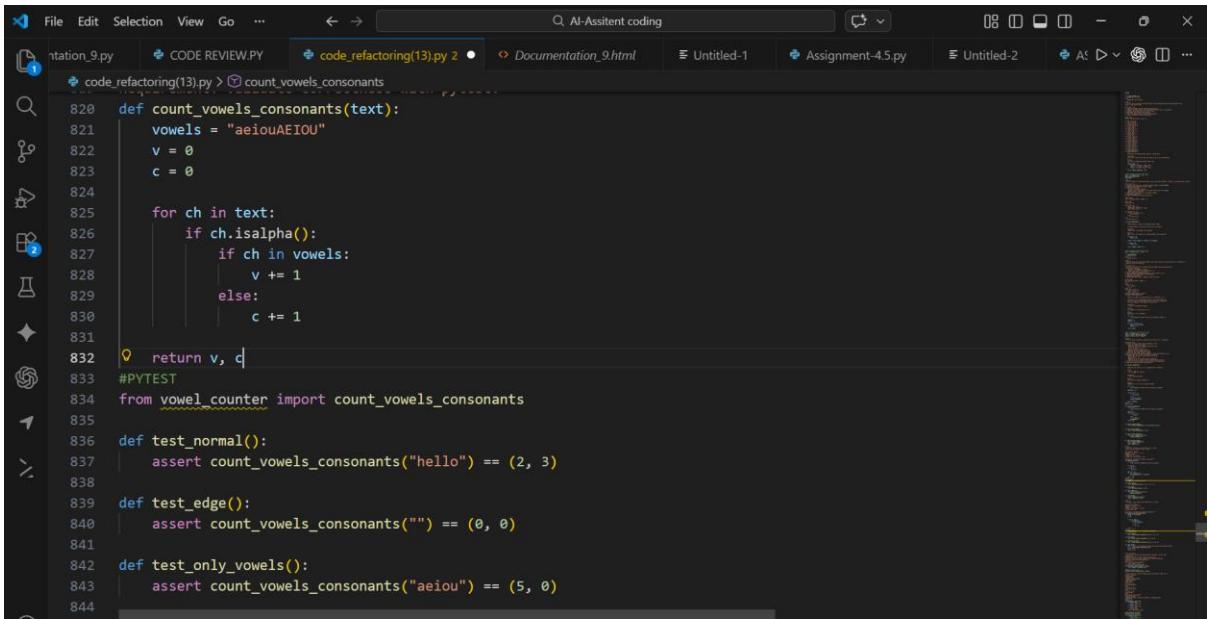
Function: Count vowels and consonants in a string.

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:



A screenshot of a code editor window titled "AI-Assistant coding". The main pane displays Python code for a function named `count_vowels_consonants`. The code uses a for loop to iterate through each character in the input text. It checks if the character is alphabetic using `isalpha()`. If it is, it checks if it is a vowel using the string `"aeiouAEIOU"`. If it is a vowel, it increments the vowel count `v` by 1. Otherwise, it increments the consonant count `c` by 1. After the loop, the function returns a tuple containing the vowel count and the consonant count. The code is annotated with several test cases using the `assert` statement to verify the function's behavior for normal text, empty strings, and strings containing only vowels.

```
820 def count_vowels_consonants(text):
821     vowels = "aeiouAEIOU"
822     v = 0
823     c = 0
824
825     for ch in text:
826         if ch.isalpha():
827             if ch in vowels:
828                 v += 1
829             else:
830                 c += 1
831
832     return v, c
833 #PYTEST
834 from vowel_counter import count_vowels_consonants
835
836 def test_normal():
837     assert count_vowels_consonants("hello") == (2, 3)
838
839 def test_edge():
840     assert count_vowels_consonants("") == (0, 0)
841
842 def test_only_vowels():
843     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.