Aissignment-13.1

2303A51679

Batch-23

Task Description #1 (Refactoring – Removing Code Duplication)

• Task: Use AI to refactor a given Python script that contains multiple repeated code blocks.

• Instructions:

o Prompt AI to identify duplicate logic and replace it with functions or classes.

o Ensure the refactored code maintains the same output.

o Add docstrings to all functions.

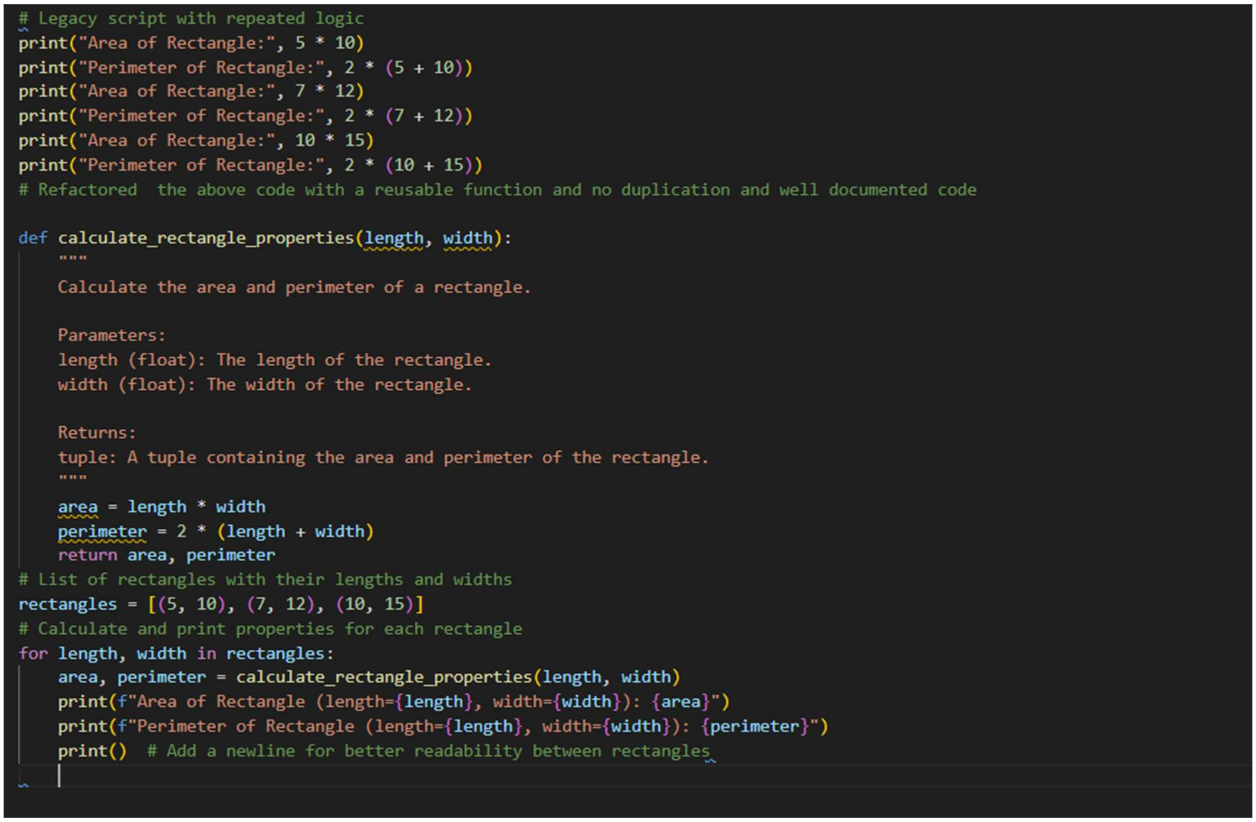
• Sample Legacy Code:

```
# Legacy script with repeated logic
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))
```

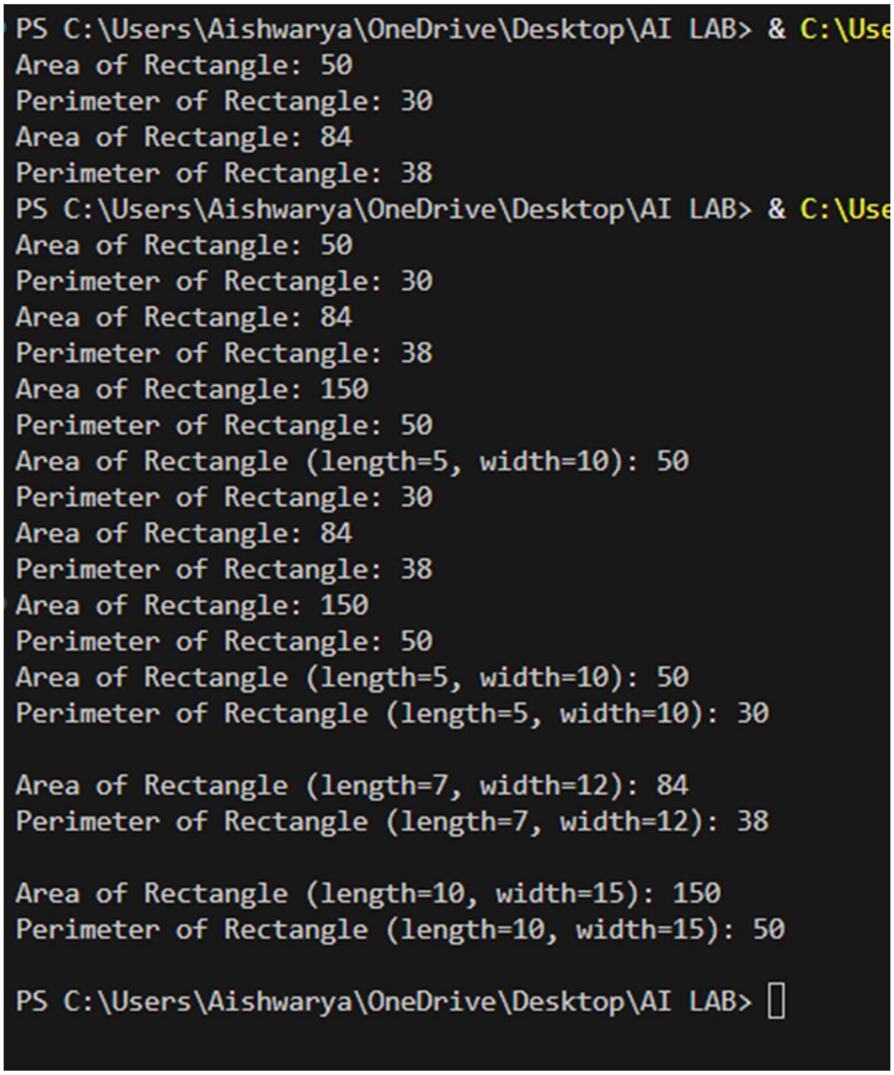
• Expected Output:

o Refactored code with a reusable function and no duplication.

o Well documented code



```python
# Legacy script with repeated logic
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))
# Refactored  the above code with a reusable function and no duplication and well documented code

def calculate_rectangle_properties(length, width):
    """
    Calculate the area and perimeter of a rectangle.

    Parameters:
    length (float): The length of the rectangle.
    width (float): The width of the rectangle.

    Returns:
    tuple: A tuple containing the area and perimeter of the rectangle.
    """
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter
# List of rectangles with their lengths and widths
rectangles = [(5, 10), (7, 12), (10, 15)]
# Calculate and print properties for each rectangle
for length, width in rectangles:
    area, perimeter = calculate_rectangle_properties(length, width)
    print(f"Area of Rectangle (length={length}, width={width}): {area}")
    print(f"Perimeter of Rectangle (length={length}, width={width}): {perimeter}")
    print()  # Add a newline for better readability between rectangles
```



```
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> & C:\Use
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> & C:\Use
Area of Rectangle: 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
Area of Rectangle (length=5, width=10): 50
Perimeter of Rectangle: 30
Area of Rectangle: 84
Perimeter of Rectangle: 38
Area of Rectangle: 150
Perimeter of Rectangle: 50
Area of Rectangle (length=5, width=10): 50
Perimeter of Rectangle (length=5, width=10): 30

Area of Rectangle (length=7, width=12): 84
Perimeter of Rectangle (length=7, width=12): 38

Area of Rectangle (length=10, width=15): 150
Perimeter of Rectangle (length=10, width=15): 50

PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> ▯
```

Task Description #2 (Refactoring – Extracting Reusable Functions)

• Task: Use AI to refactor a legacy script where multiple calculations are embedded directly inside the main code block.

• Instructions:

o Identify repeated or related logic and extract it into reusable functions.

o Ensure the refactored code is modular, easy to read, and documented with docstrings.

• Sample Legacy Code:

Week7

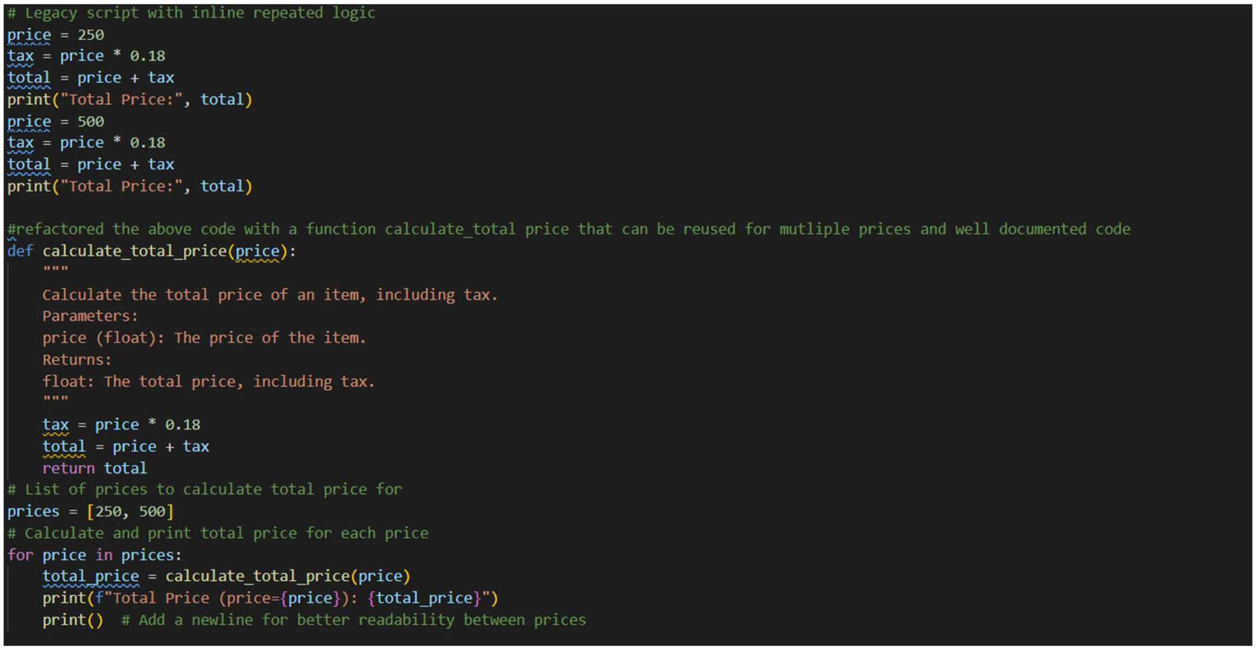
-

Monda

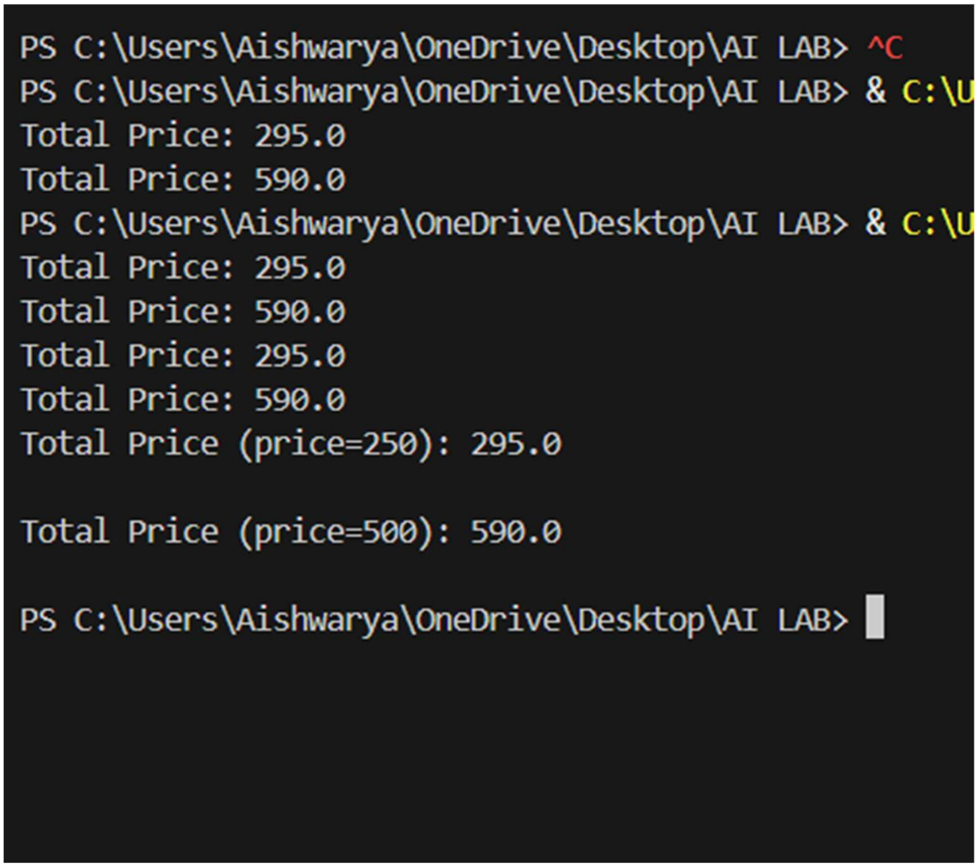
y

```
# Legacy script with inline repeated logic
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)
```

- Expected Output:
    - Code with a function calculate_total(price) that can be reused for multiple price inputs.
    - Well documented code



```python
# Legacy script with inline repeated logic
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)

#refactored the above code with a function calculate_total price that can be reused for mutliple prices and well documented code
def calculate_total_price(price):
    """
    Calculate the total price of an item, including tax.
    Parameters:
    price (float): The price of the item.
    Returns:
    float: The total price, including tax.
    """
    tax = price * 0.18
    total = price + tax
    return total
# List of prices to calculate total price for
prices = [250, 500]
# Calculate and print total price for each price
for price in prices:
    total_price = calculate_total_price(price)
    print(f"Total Price (price={price}): {total_price}")
    print()  # Add a newline for better readability between prices
```



```
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> ^C
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> & C:\U
Total Price: 295.0
Total Price: 590.0
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB> & C:\U
Total Price: 295.0
Total Price: 590.0
Total Price: 295.0
Total Price: 590.0
Total Price (price=250): 295.0

Total Price (price=500): 590.0

PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB>
```

Task Description #3: Refactoring Using Classes and Methods (Eliminating Redundant Conditional Logic)

Refactor a Python script that contains repeated if–elif–else grading logic by implementing a structured, object-oriented solution using a class and a method.

Problem Statement

The given script contains duplicated conditional statements used to assign grades based on student marks. This redundancy violates clean code principles and reduces maintainability.

You are required to refactor the script using a class-based design to improve modularity, reusability, and readability while preserving the original grading logic.

Mandatory Implementation Requirements

1. Class Name: GradeCalculator

2. Method Name: calculate_grade(self, marks)

3. The method must:

o Accept marks as a parameter.

o Return the corresponding grade as a string.

o The grading logic must strictly follow the conditions below:

▪ Marks ≥ 90 and ≤ 100 → "Grade A"

▪ Marks ≥ 80 → "Grade B"

▪ Marks ≥ 70 → "Grade C"

▪ Marks ≥ 40 → "Grade D"

▪ Marks ≥ 0 → "Fail"

Note: Assume marks are within the valid range of 0 to 100.

4. Include proper docstrings for:

o The class

o The method (with parameter and return descriptions)

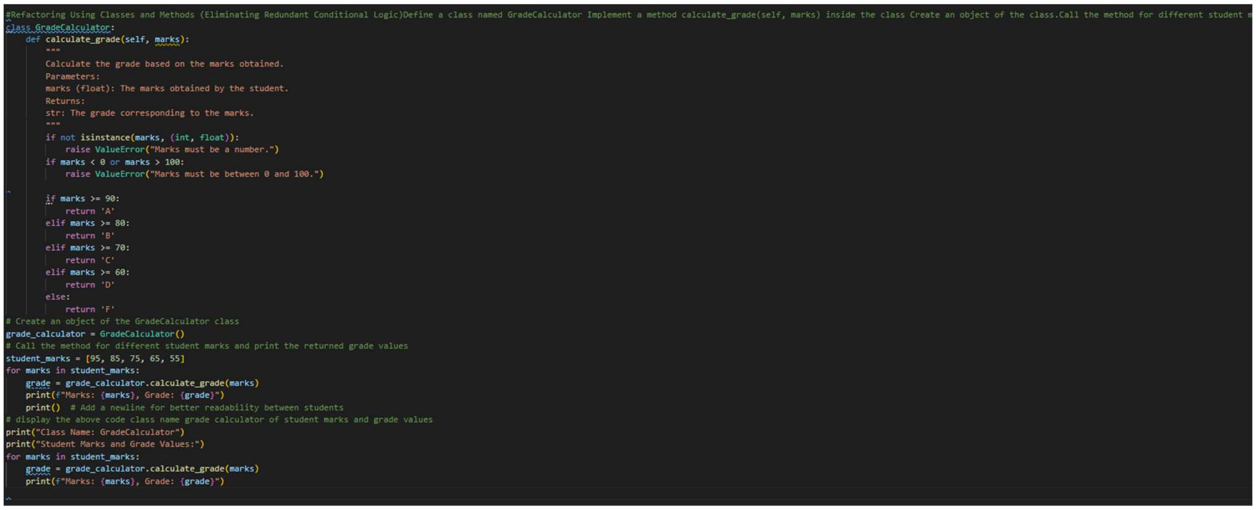
5. The method must be reusable and called multiple times without rewriting conditional logic.

- Given code:

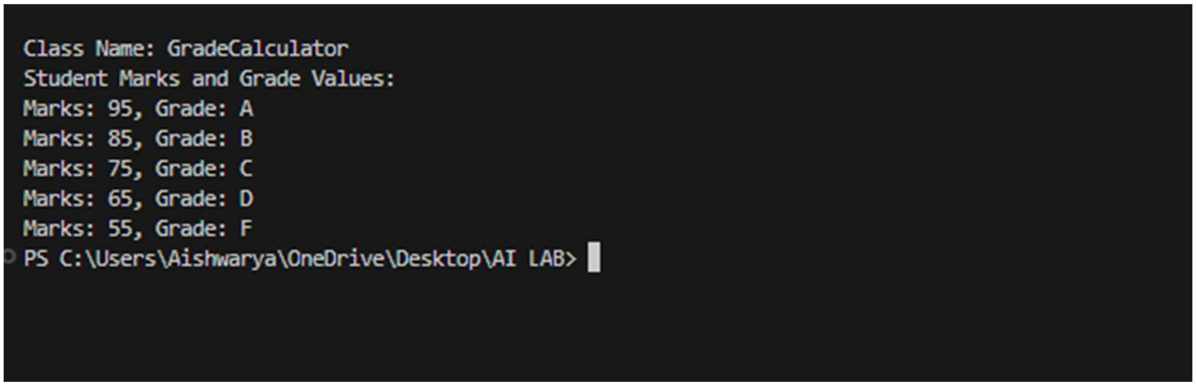
```
marks = 85
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
marks = 72
if marks >= 90:
print("Grade A")
elif marks >= 75:
print("Grade B")
else:
print("Grade C")
```

Expected Output:

- Define a class named GradeCalculator.
- Implement a method calculate_grade(self, marks) inside the class.
- Create an object of the class.
- Call the method for different student marks.
- Print the returned grade values.



```
#Refactoring Using Classes and Methods (Eliminating Redundant Conditional Logic)Define a class named GradeCalculator Implement a method calculate_grade(self, marks) inside the class Create an object of the class.Call the method for different student m
class GradeCalculator:
    def calculate_grade(self, marks):
        """
        Calculate the grade based on the marks obtained.
        Parameters:
        marks (float): The marks obtained by the student.
        Returns:
        str: The grade corresponding to the marks.
        """
        if not isinstance(marks, (int, float)):
            raise ValueError("Marks must be a number.")
        if marks < 0 or marks > 100:
            raise ValueError("Marks must be between 0 and 100.")

        if marks >= 90:
            return 'A'
        elif marks >= 80:
            return 'B'
        elif marks >= 70:
            return 'C'
        elif marks >= 60:
            return 'D'
        else:
            return 'F'
# Create an object of the GradeCalculator class
grade_calculator = GradeCalculator()
# Call the method for different student marks and print the returned grade values
student_marks = [95, 85, 75, 65, 55]
for marks in student_marks:
    grade = grade_calculator.calculate_grade(marks)
    print(f"Marks: {marks}, Grade: {grade}")
    print()  # Add a newline for better readability between students
# display the above code class name grade calculator of student marks and grade values
print("Class Name: GradeCalculator")
print("Student Marks and Grade Values:")
for marks in student_marks:
    grade = grade_calculator.calculate_grade(marks)
    print(f"Marks: {marks}, Grade: {grade}")
```



```
Class Name: GradeCalculator
Student Marks and Grade Values:
Marks: 95, Grade: A
Marks: 85, Grade: B
Marks: 75, Grade: C
Marks: 65, Grade: D
Marks: 55, Grade: F
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB>
```

Task Description #4 (Refactoring – Converting Procedural Code to Functions)

• Task: Use AI to refactor procedural input–processing logic into functions.

Instructions:

o Identify input, processing, and output sections.

o Convert each into a separate function.

o Improve code readability without changing behavior.

• Sample Legacy Code:

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

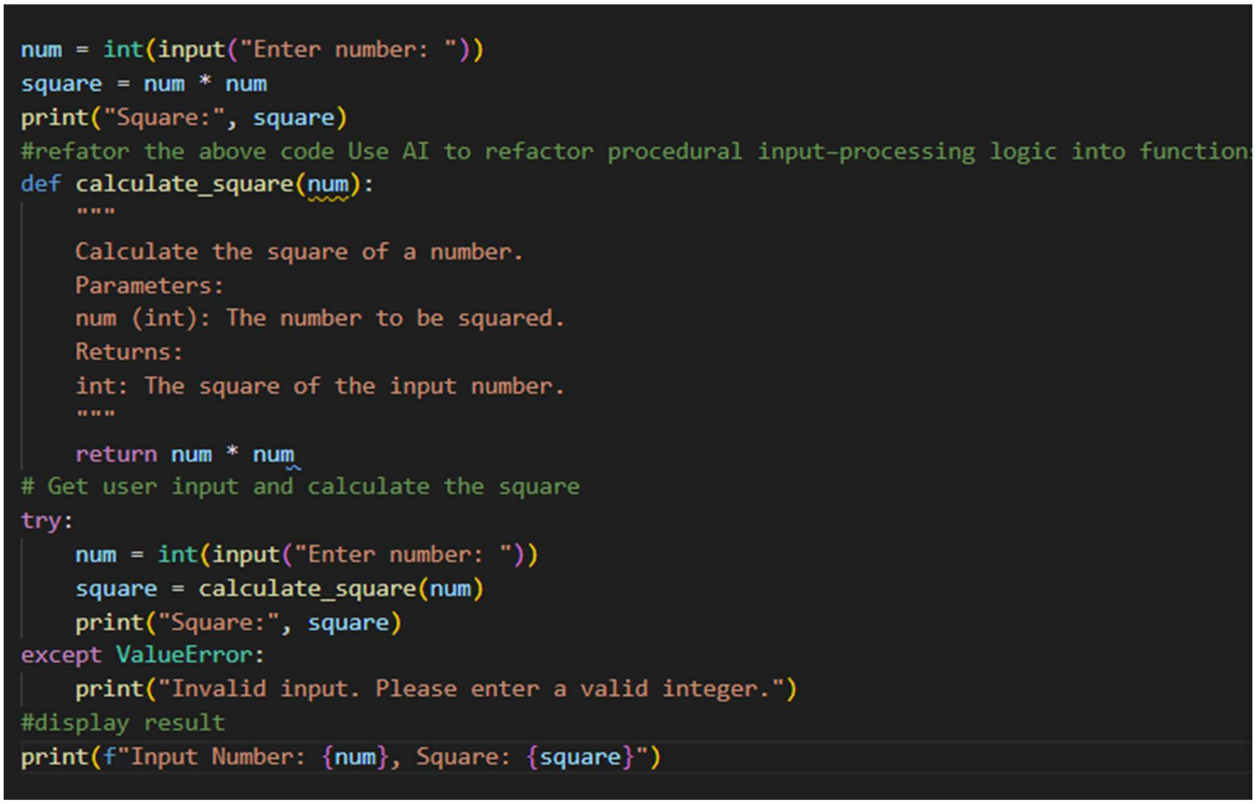
square = num * num

print("Square:", square)

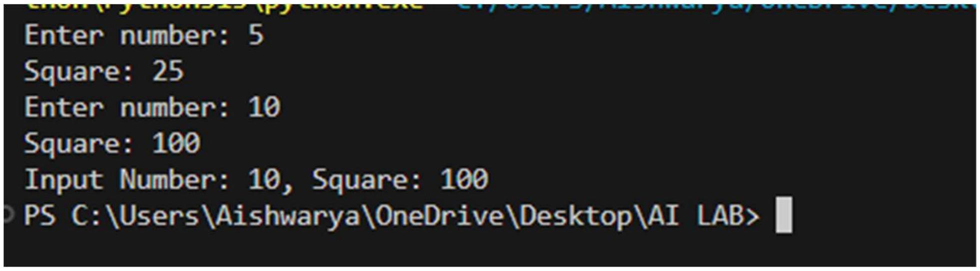
• Expected Output:

o Modular code using functions like get_input(), calculate_square(), and

display_result().



```python
num = int(input("Enter number: "))
square = num * num
print("Square:", square)
#refator the above code Use AI to refactor procedural input-processing logic into function
def calculate_square(num):
    """
    Calculate the square of a number.
    Parameters:
    num (int): The number to be squared.
    Returns:
    int: The square of the input number.
    """
    return num * num
# Get user input and calculate the square
try:
    num = int(input("Enter number: "))
    square = calculate_square(num)
    print("Square:", square)
except ValueError:
    print("Invalid input. Please enter a valid integer.")
#display result
print(f"Input Number: {num}, Square: {square}")
```



```
Enter number: 5
Square: 25
Enter number: 10
Square: 100
Input Number: 10, Square: 100
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB>
```

Task 5 (Refactoring Procedural Code into OOP Design)

• Task: Use AI to refactor procedural code into a class-based design.

Focus Areas:

o Object-Oriented principles

o Encapsulation

Legacy Code:

salary = 50000

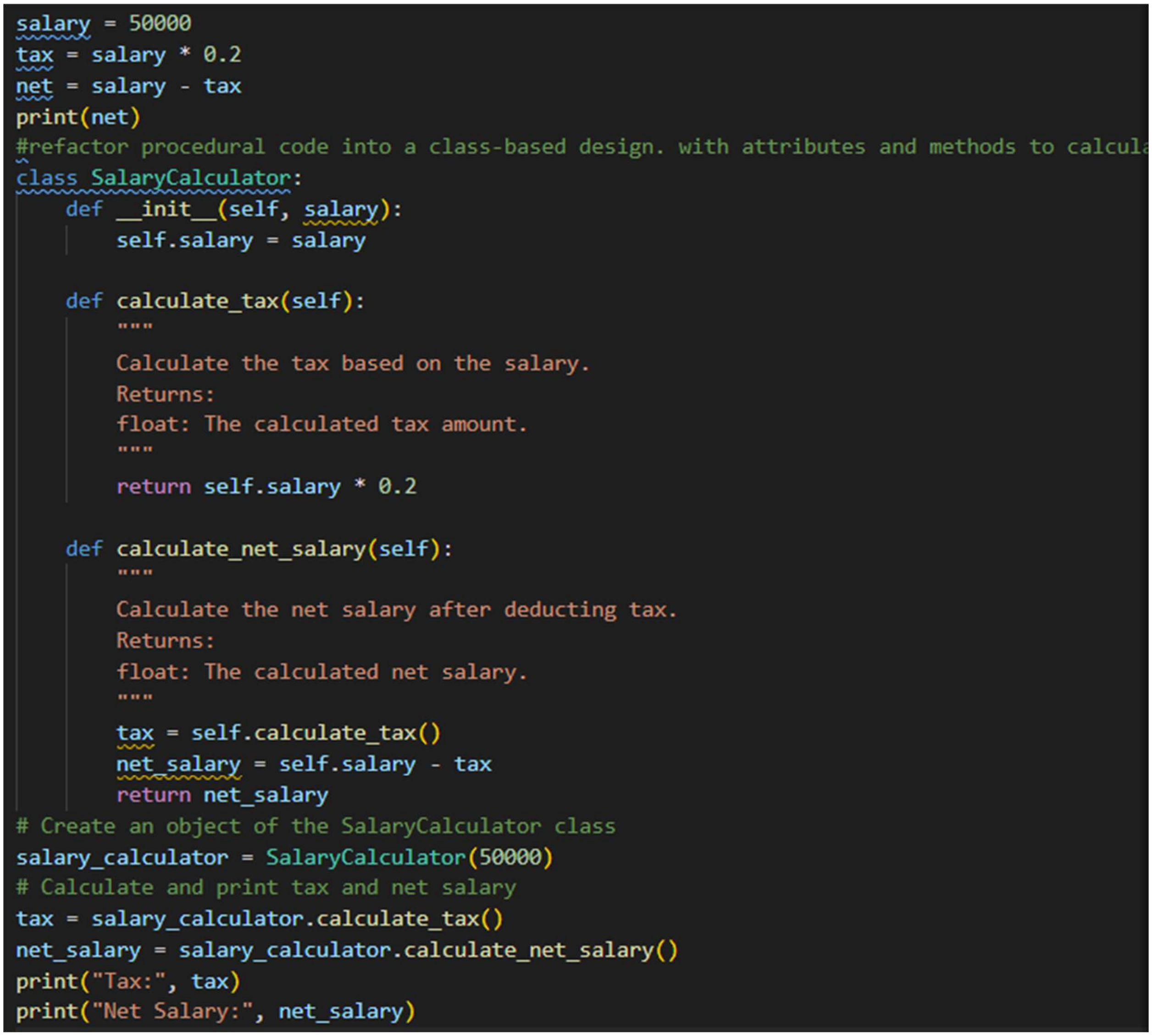
tax = salary * 0.2

net = salary - tax

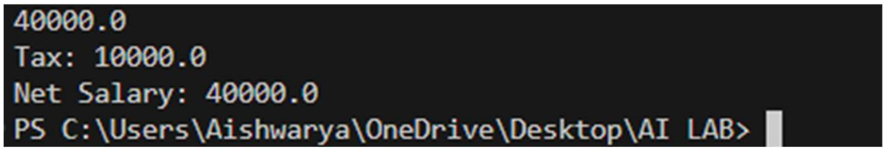
print(net)

Expected Outcome:

o A class like EmployeeSalaryCalculator with methods and attributes.



```python
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
#refactor procedural code into a class-based design. with attributes and methods to calcula
class SalaryCalculator:
    def __init__(self, salary):
        self.salary = salary

    def calculate_tax(self):
        """
        Calculate the tax based on the salary.
        Returns:
        float: The calculated tax amount.
        """
        return self.salary * 0.2

    def calculate_net_salary(self):
        """
        Calculate the net salary after deducting tax.
        Returns:
        float: The calculated net salary.
        """
        tax = self.calculate_tax()
        net_salary = self.salary - tax
        return net_salary
# Create an object of the SalaryCalculator class
salary_calculator = SalaryCalculator(50000)
# Calculate and print tax and net salary
tax = salary_calculator.calculate_tax()
net_salary = salary_calculator.calculate_net_salary()
print("Tax:", tax)
print("Net Salary:", net_salary)
```



```
40000.0
Tax: 10000.0
Net Salary: 40000.0
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB>
```

Task 6 (Optimizing Search Logic)

- Task: Refactor inefficient linear searches using appropriate data structures.

- Focus Areas:

o Time complexity

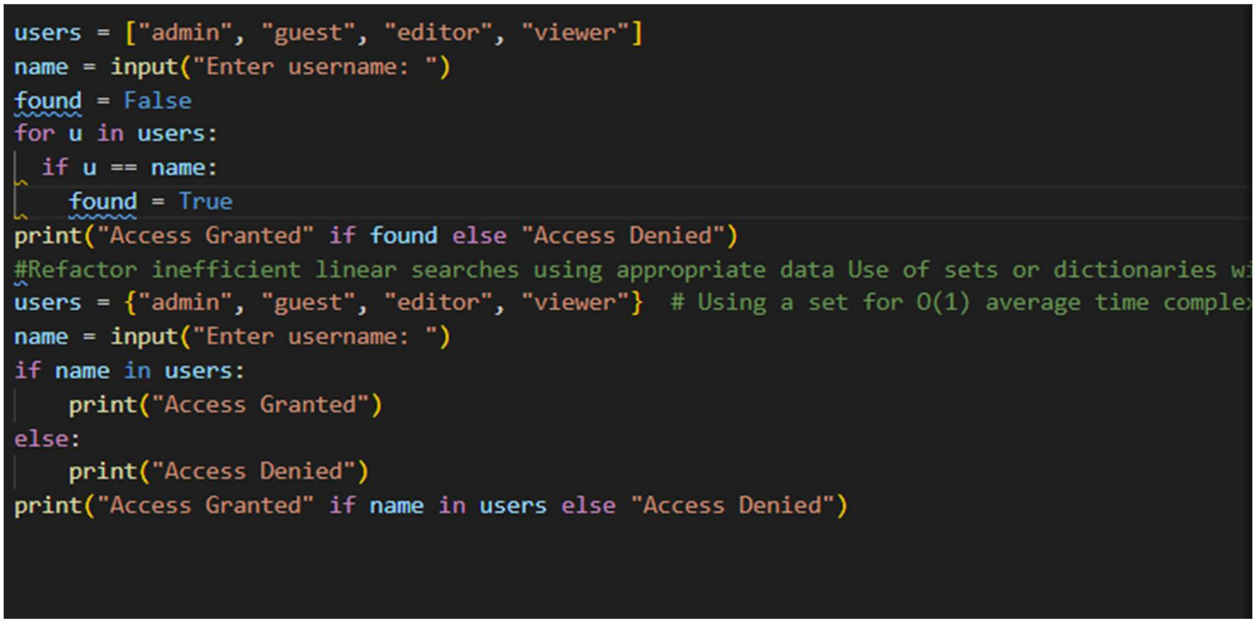
o Data structure choice

Legacy Code:

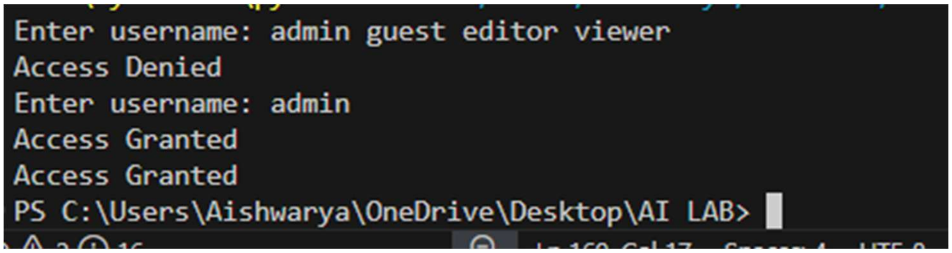
```
users = ["admin", "guest", "editor", "viewer"]
name = input("Enter username: ")
found = False
for u in users:
if u == name:
found = True
print("Access Granted" if found else "Access Denied")
```

Expected Outcome:

o Use of sets or dictionaries with complexity justification



```python
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")
#Refactor inefficient linear searches using appropriate data Use of sets or dictionaries wi
users = {"admin", "guest", "editor", "viewer"}  # Using a set for O(1) average time comple
name = input("Enter username: ")
if name in users:
    print("Access Granted")
else:
    print("Access Denied")
print("Access Granted" if name in users else "Access Denied")
```



```
Enter username: admin guest editor viewer
Access Denied
Enter username: admin
Access Granted
Access Granted
PS C:\Users\Aishwarya\OneDrive\Desktop\AI LAB>
```

Task 7 – Refactoring the Library Management System

Problem Statement

You are provided with a poorly structured Library Management script that:

- Contains repeated conditional logic

- Does not use reusable functions

- Lacks documentation

- Uses print-based procedural execution

- Does not follow modular programming principles

Your task is to refactor the code into a proper format

1. Create a module library.py with functions:

o add_book(title, author, isbn)

o remove_book(isbn)

o search_book(isbn)

2. Insert triple quotes under each function and let Copilot complete the docstrings.

3. Generate documentation in the terminal.

4. Export the documentation in HTML format.

5. Open the file in a browser.

Given Code

```
# Library Management System (Unstructured Version)
# This code needs refactoring into a proper module with documentation.
library_db = {}
# Adding first book
title = "Python Basics"
author = "John Doe"
isbn = "101"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
else:
print("Book already exists.")
# Adding second book (duplicate logic)
title = "AI Fundamentals"
author = "Jane Smith"
isbn = "102"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
```

```
else:
print("Book already exists.")
# Searching book (repeated logic structure)
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
else:
print("Book not found.")
# Removing book (again repeated pattern)
isbn = "101"
if isbn in library_db:
del library_db[isbn]
print("Book removed successfully.")
else:
print("Book not found.")
# Searching again
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
else:
print("Book not found.")
```

Task 7 – Refactoring the Library Management System

Problem Statement

You are provided with a poorly structured Library Management script that:

- Contains repeated conditional logic
- Does not use reusable functions

- Lacks documentation
- Uses print-based procedural execution
- Does not follow modular programming principles

Your task is to refactor the code into a proper format

1. Create a module library.py with functions:

o add_book(title, author, isbn)

o remove_book(isbn)

o search_book(isbn)

2. Insert triple quotes under each function and let Copilot complete the docstrings.
3. Generate documentation in the terminal.
4. Export the documentation in HTML format.
5. Open the file in a browser.

Given Code

```
# Library Management System (Unstructured Version)
# This code needs refactoring into a proper module with documentation.
library_db = {}
# Adding first book
title = "Python Basics"
author = "John Doe"
isbn = "101"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
else:
print("Book already exists.")
```

```
# Adding second book (duplicate logic)
title = "AI Fundamentals"
author = "Jane Smith"
isbn = "102"
if isbn not in library_db:
library_db[isbn] = {"title": title, "author": author}
print("Book added successfully.")
else:
print("Book already exists.")
# Searching book (repeated logic structure)
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
else:
print("Book not found.")
# Removing book (again repeated pattern)
isbn = "101"
if isbn in library_db:
del library_db[isbn]
print("Book removed successfully.")
else:
print("Book not found.")
# Searching again
isbn = "101"
if isbn in library_db:
print("Book Found:", library_db[isbn])
```

```
else:
print("Book not found.")
```

Task 8– Fibonacci Generator

Write a program to generate Fibonacci series up to n.

The initial code has:

- Global variables.

- Inefficient loop.

- No functions or modularity.

Task for Students:

- Refactor into a clean reusable function (generate_fibonacci).

- Add docstrings and test cases.

- Compare AI-refactored vs original.

Bad Code Version:

```
# fibonacci bad version
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
```

Task 9 – Twin Primes Checker

Twin primes are pairs of primes that differ by 2 (e.g., 11 and 13, 17 and 19).

The initial code has:

- Inefficient prime checking.

- No functions.

- Hardcoded inputs.

Task for Students:

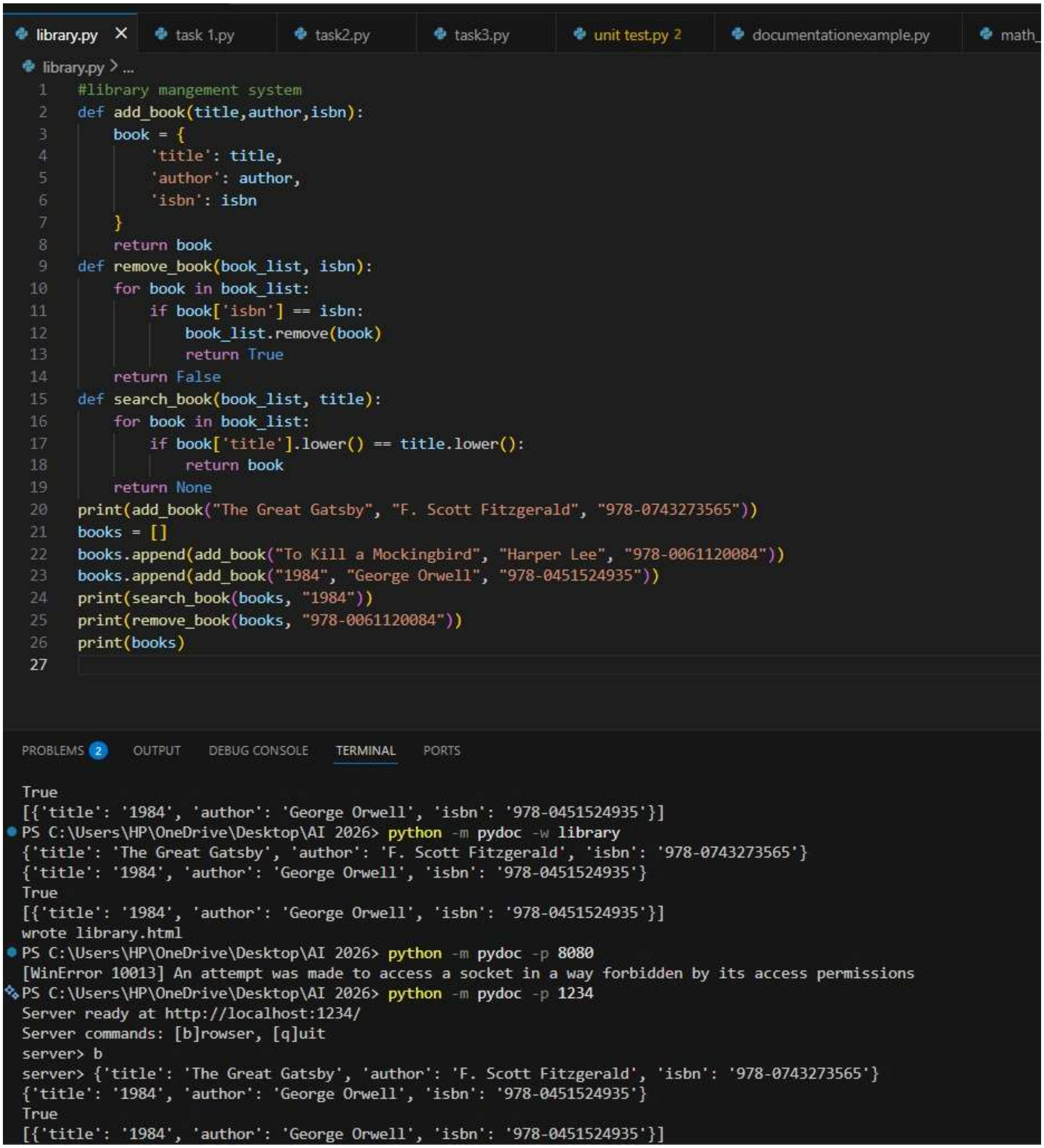
- Refactor into is_prime(n) and is_twin_prime(p1, p2).

- Add docstrings and optimize.

- Generate a list of twin primes in a given range using AI.

Bad Code Version:

```
# twin primes bad version
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")
```



```python
#library mangement system
def add_book(title,author,isbn):
    book = {
        'title': title,
        'author': author,
        'isbn': isbn
    }
    return book
def remove_book(book_list, isbn):
    for book in book_list:
        if book['isbn'] == isbn:
            book_list.remove(book)
            return True
    return False
def search_book(book_list, title):
    for book in book_list:
        if book['title'].lower() == title.lower():
            return book
    return None
print(add_book("The Great Gatsby", "F. Scott Fitzgerald", "978-0743273565"))
books = []
books.append(add_book("To Kill a Mockingbird", "Harper Lee", "978-0061120084"))
books.append(add_book("1984", "George Orwell", "978-0451524935"))
print(search_book(books, "1984"))
print(remove_book(books, "978-0061120084"))
print(books)

```

```
True
[{'title': '1984', 'author': 'George Orwell', 'isbn': '978-0451524935'}]
PS C:\Users\HP\OneDrive\Desktop\AI 2026> python -m pydoc -w library
{'title': 'The Great Gatsby', 'author': 'F. Scott Fitzgerald', 'isbn': '978-0743273565'}
{'title': '1984', 'author': 'George Orwell', 'isbn': '978-0451524935'}
True
[{'title': '1984', 'author': 'George Orwell', 'isbn': '978-0451524935'}]
wrote library.html
PS C:\Users\HP\OneDrive\Desktop\AI 2026> python -m pydoc -p 8080
[WinError 10013] An attempt was made to access a socket in a way forbidden by its access permissions
PS C:\Users\HP\OneDrive\Desktop\AI 2026> python -m pydoc -p 1234
Server ready at http://localhost:1234/
Server commands: [b]rowser, [q]uit
server> b
server> {'title': 'The Great Gatsby', 'author': 'F. Scott Fitzgerald', 'isbn': '978-0743273565'}
{'title': '1984', 'author': 'George Orwell', 'isbn': '978-0451524935'}
True
[{'title': '1984', 'author': 'George Orwell', 'isbn': '978-0451524935'}]
```

index
library c:\users\hp\onedrive\desktop\ai 2026\library.py

#library mangement system

Functions

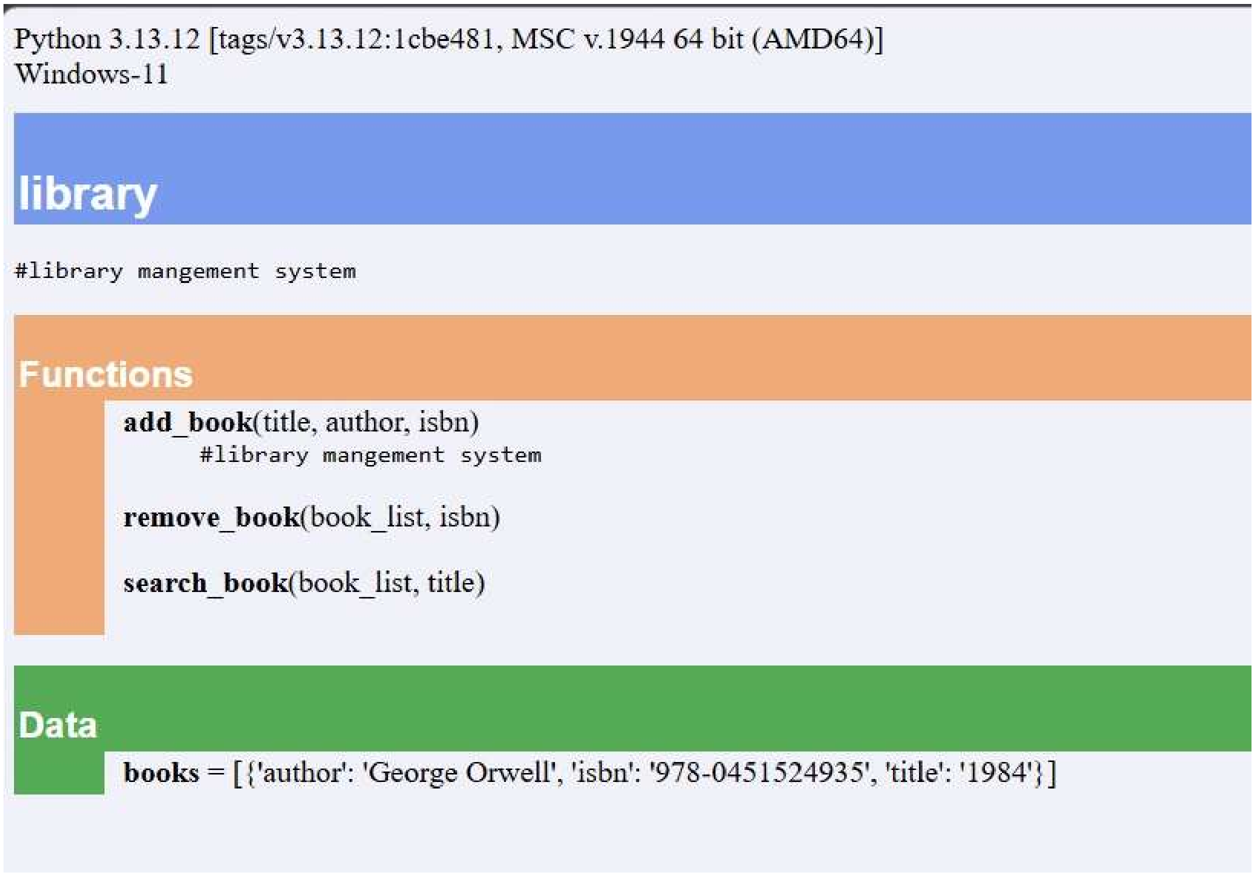
add_book(title, author, isbn)
#library mangement system

remove_book(book_list, isbn)

search_book(book_list, title)

Data

books = [{'author': 'George Orwell', 'isbn': '978-0451524935', 'title': '1984'}]



```
Python 3.13.12 [tags/v3.13.12:1cbe481, MSC v.1944 64 bit (AMD64)]
Windows-11

library

#library mangement system

Functions
    add_book(title, author, isbn)
        #library mangement system

    remove_book(book_list, isbn)

    search_book(book_list, title)

Data
    books = [{'author': 'George Orwell', 'isbn': '978-0451524935', 'title': '1984'}]
```

Task 8– Fibonacci Generator

Write a program to generate Fibonacci series up to n.

The initial code has:

- Global variables.

- Inefficient loop.

- No functions or modularity.

Task for Students:

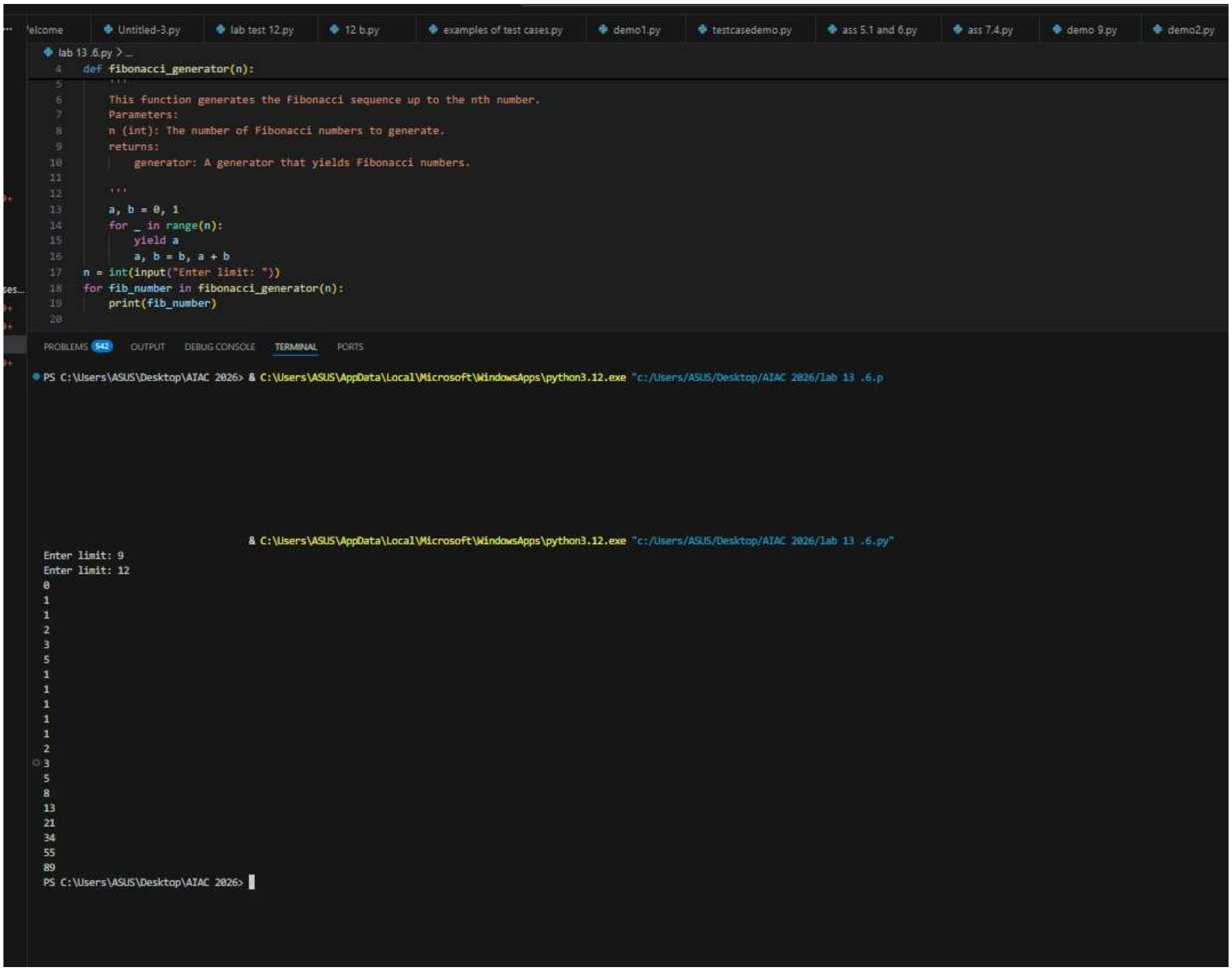
- Refactor into a clean reusable function (generate_fibonacci).

- Add docstrings and test cases.

- Compare AI-refactored vs original.

Bad Code Version:

```
# fibonacci bad version
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
```



```python
def fibonacci_generator(n):
    '''
    This function generates the Fibonacci sequence up to the nth number.
    Parameters:
    n (int): The number of Fibonacci numbers to generate.
    returns:
        generator: A generator that yields Fibonacci numbers.

    '''
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
n = int(input("Enter limit: "))
for fib_number in fibonacci_generator(n):
    print(fib_number)
```

```
PS C:\Users\ASUS\Desktop\AIAC 2026> & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.p




                               & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Enter limit: 9
Enter limit: 12
0
1
1
2
3
5
1
1
1
1
1
2
3
5
8
13
21
34
55
89
PS C:\Users\ASUS\Desktop\AIAC 2026>
```

Task 9 – Twin Primes Checker

Twin primes are pairs of primes that differ by 2 (e.g., 11 and 13, 17 and 19).

The initial code has:

- Inefficient prime checking.

- No functions.

- Hardcoded inputs.

Task for Students:

- Refactor into is_prime(n) and is_twin_prime(p1, p2).

- Add docstrings and optimize.

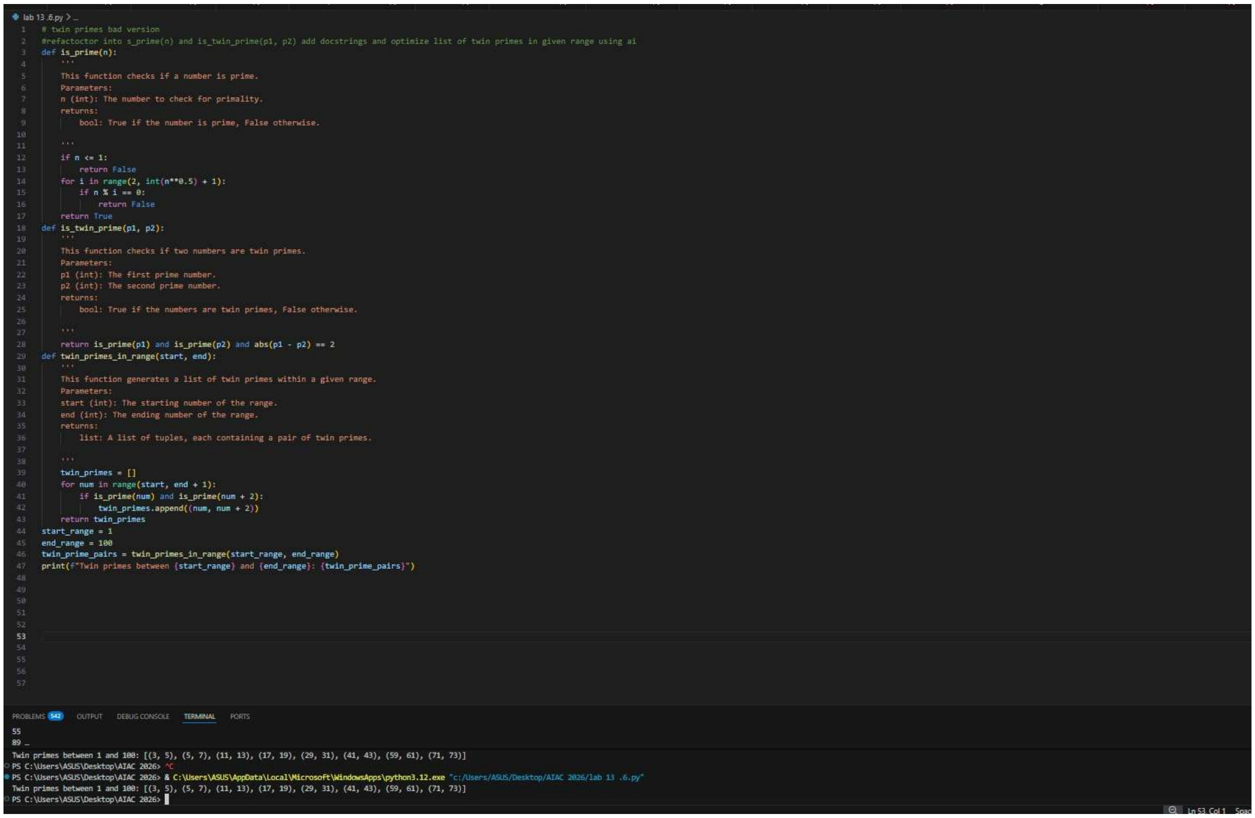
- Generate a list of twin primes in a given range using AI.

Bad Code Version:

```
# twin primes bad version

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")/



```python
# twin primes bad version
#refactoctor into s_prime(n) and is_twin_prime(p1, p2) add docstrings and optimize list of twin primes in given range using ai
def is_prime(n):
    '''
    This function checks if a number is prime.
    Parameters:
    n (int): The number to check for primality.
    returns:
        bool: True if the number is prime, False otherwise.

    '''
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
def is_twin_prime(p1, p2):
    '''
    This function checks if two numbers are twin primes.
    Parameters:
    p1 (int): The first prime number.
    p2 (int): The second prime number.
    returns:
        bool: True if the numbers are twin primes, False otherwise.

    '''
    return is_prime(p1) and is_prime(p2) and abs(p1 - p2) == 2
def twin_primes_in_range(start, end):
    '''
    This function generates a list of twin primes within a given range.
    Parameters:
    start (int): The starting number of the range.
    end (int): The ending number of the range.
    returns:
        list: A list of tuples, each containing a pair of twin primes.

    '''
    twin_primes = []
    for num in range(start, end + 1):
        if is_prime(num) and is_prime(num + 2):
            twin_primes.append((num, num + 2))
    return twin_primes
start_range = 1
end_range = 100
twin_prime_pairs = twin_primes_in_range(start_range, end_range)
print(f"Twin primes between {start_range} and {end_range}: {twin_prime_pairs}")
```

```
55
89 _
Twin primes between 1 and 100: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
PS C:\Users\ASUS\Desktop\AIAC 2026> ^C
PS C:\Users\ASUS\Desktop\AIAC 2026> & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Twin primes between 1 and 100: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
PS C:\Users\ASUS\Desktop\AIAC 2026> 
```

Task 10 – Refactoring the Chinese Zodiac Program

Objective

Refactor the given poorly structured Python script into a clean, modular, and reusable implementation.

The current program reads a year from the user and prints the corresponding Chinese Zodiac sign. However, the implementation contains repetitive conditional logic, lacks modular design, and does not follow clean coding principles.

Your task is to refactor the code to improve readability, maintainability, and structure.

Chinese Zodiac Cycle (Repeats Every 12 Years)

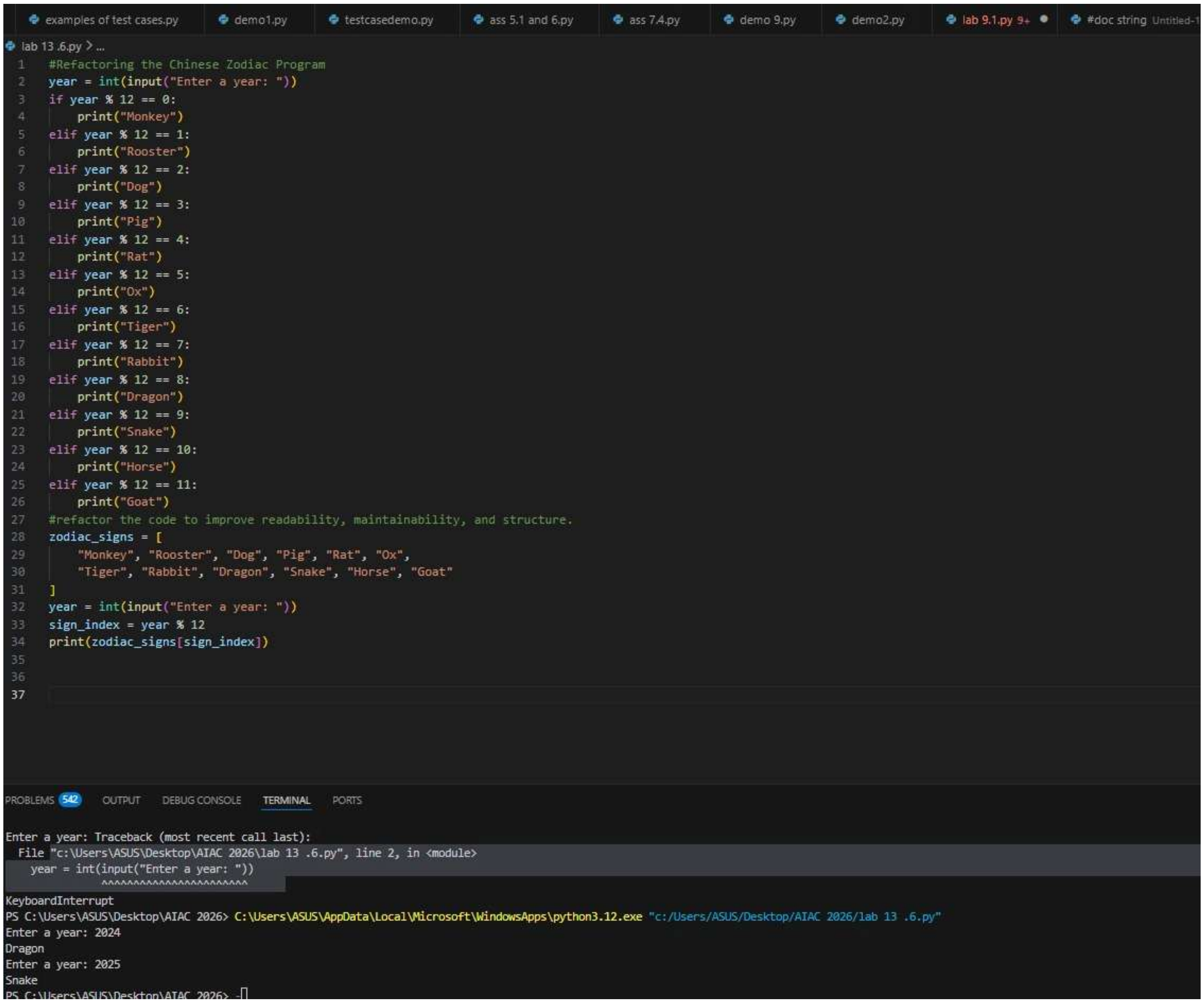
1. Rat
2. Ox
3. Tiger
4. Rabbit
5. Dragon
6. Snake
7. Horse
8. Goat (Sheep)
9. Monkey
10. Rooster
11. Dog
12. Pig

```
# Chinese Zodiac Program (Unstructured Version)
# This code needs refactoring.
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")
```

You must:

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



```python
#Refactoring the Chinese Zodiac Program
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")
#refactor the code to improve readability, maintainability, and structure.
zodiac_signs = [
    "Monkey", "Rooster", "Dog", "Pig", "Rat", "Ox",
    "Tiger", "Rabbit", "Dragon", "Snake", "Horse", "Goat"
]
year = int(input("Enter a year: "))
sign_index = year % 12
print(zodiac_signs[sign_index])
```

```
Enter a year: Traceback (most recent call last):
  File "c:\Users\ASUS\Desktop\AIAC 2026\lab 13 .6.py", line 2, in <module>
    year = int(input("Enter a year: "))
           ^^^^^^^^^^^^^^^^^^^^^^^
KeyboardInterrupt
PS C:\Users\ASUS\Desktop\AIAC 2026> C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Enter a year: 2024
Dragon
Enter a year: 2025
Snake
PS C:\Users\ASUS\Desktop\AIAC 2026>
```

Task 11 – Refactoring the Harshad (Niven) Number Checker

Refactor the given poorly structured Python script into a clean, modular, and reusable implementation.

A Harshad (Niven) number is a number that is divisible by the sum of its digits.

For example:

- 18 → 1 + 8 = 9 → 18 ÷ 9 = 2 (Harshad Number)
- 19 → 1 + 9 = 10 → 19 ÷ 10 ≠ integer (Not Harshad)

Problem Statement

The current implementation:

- Mixes logic and input handling
- Uses redundant variables
- Does not use reusable functions properly
- Returns print statements instead of boolean values
- Lacks documentation

You must refactor the code to follow clean coding principles.

```
# Harshad Number Checker (Unstructured Version)

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")
```

You must:

1. Create a reusable function: is_harshad(number)
2. The function must:
   - Accept an integer parameter.
   - Return True if the number is divisible by the sum of its digits.

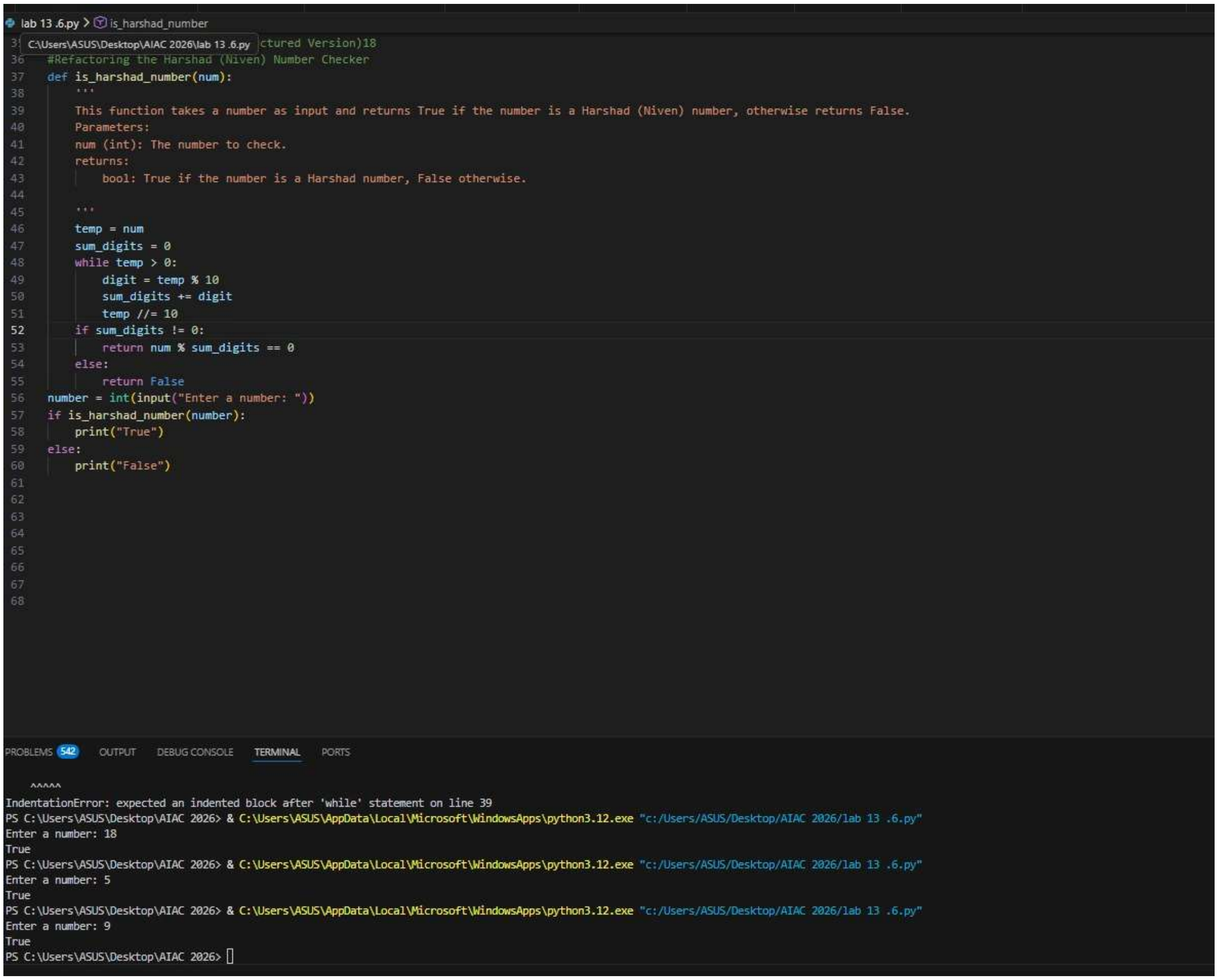
o Return False otherwise.

3. Separate user input from core logic.

4. Add proper docstrings.

5. Improve readability and maintainability.

6. Ensure the program handles edge cases (e.g., 0, negative numbers).


lab 13 .6.py > is_harshad_number
C:\Users\ASUS\Desktop\AIAC 2026\lab 13 .6.py
PROBLEMS 542
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS

```python
ctured Version)18
#Refactoring the Harshad (Niven) Number Checker
def is_harshad_number(num):
    '''
    This function takes a number as input and returns True if the number is a Harshad (Niven) number, otherwise returns False.
    Parameters:
    num (int): The number to check.
    returns:
        bool: True if the number is a Harshad number, False otherwise.

    '''
    temp = num
    sum_digits = 0
    while temp > 0:
        digit = temp % 10
        sum_digits += digit
        temp //= 10
    if sum_digits != 0:
        return num % sum_digits == 0
    else:
        return False
number = int(input("Enter a number: "))
if is_harshad_number(number):
    print("True")
else:
    print("False")
```

```
    ^^^^^
IndentationError: expected an indented block after 'while' statement on line 39
PS C:\Users\ASUS\Desktop\AIAC 2026> & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Enter a number: 18
True
PS C:\Users\ASUS\Desktop\AIAC 2026> & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Enter a number: 5
True
PS C:\Users\ASUS\Desktop\AIAC 2026> & C:\Users\ASUS\AppData\Local\Microsoft\WindowsApps\python3.12.exe "c:/Users/ASUS/Desktop/AIAC 2026/lab 13 .6.py"
Enter a number: 9
True
PS C:\Users\ASUS\Desktop\AIAC 2026> 
```

Task 12 – Refactoring the Factorial Trailing Zeros Program

Refactor the given poorly structured Python script into a clean, modular, and efficient implementation.

The program calculates the number of trailing zeros in n! (factorial of n).

Problem Statement

The current implementation:

- Calculates the full factorial (inefficient for large n)
- Mixes input handling with business logic
- Uses print statements instead of return values
- Lacks modular structure and documentation

You must refactor the code to improve efficiency, readability, and maintainability.

```
# Factorial Trailing Zeros (Unstructured Version)
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)
```

You must:

1. Create a reusable function: count_trailing_zeros(n)
2. The function must:
    - Accept a non-negative integer n.
    - Return the number of trailing zeros in n!.
3. Do NOT compute the full factorial.

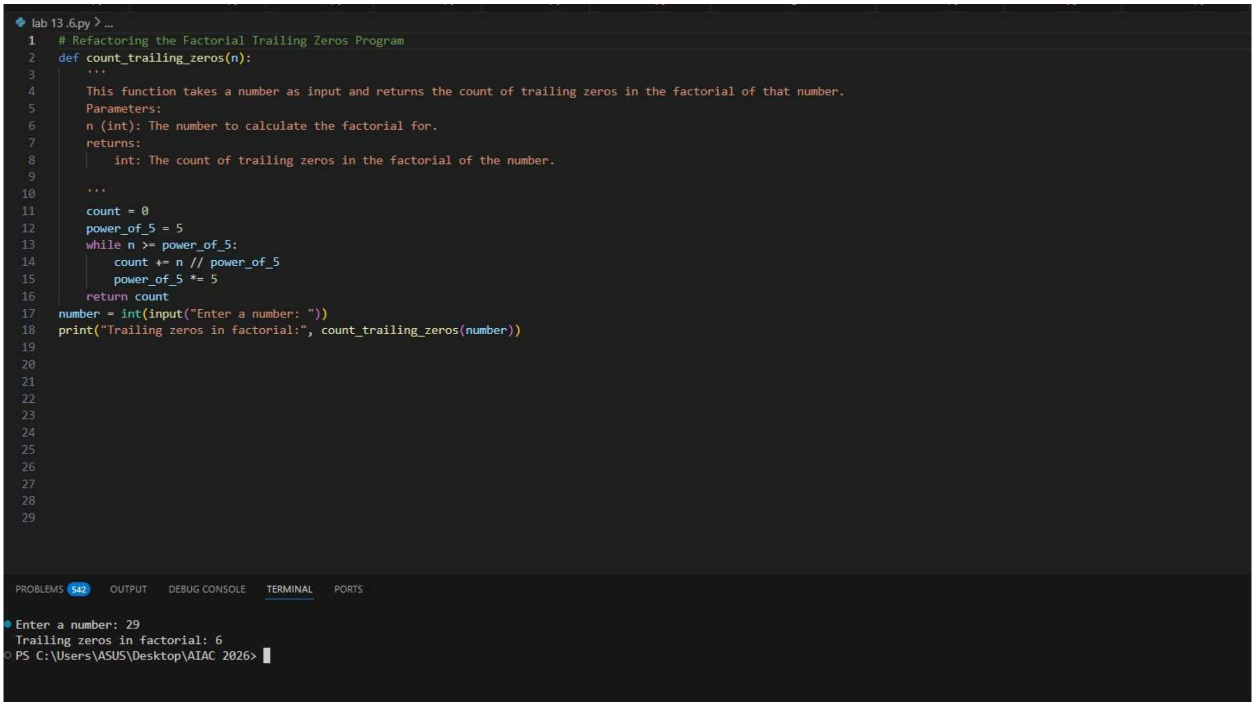
4. Use an optimized mathematical approach (count multiples of 5).

5. Add proper docstrings.

6. Separate user interaction from core logic.

7. Handle edge cases (e.g., negative numbers, zero).

Test Cases Design



```python
# Refactoring the Factorial Trailing Zeros Program
def count_trailing_zeros(n):
    '''
    This function takes a number as input and returns the count of trailing zeros in the factorial of that number.
    Parameters:
    n (int): The number to calculate the factorial for.
    returns:
        int: The count of trailing zeros in the factorial of the number.

    '''
    count = 0
    power_of_5 = 5
    while n >= power_of_5:
        count += n // power_of_5
        power_of_5 *= 5
    return count
number = int(input("Enter a number: "))
print("Trailing zeros in factorial:", count_trailing_zeros(number))
```

```
Enter a number: 29
Trailing zeros in factorial: 6
PS C:\Users\ASUS\Desktop\AIAC 2026>
```

Task 13 (Collatz Sequence Generator – Test Case Design)

• Function: Generate Collatz sequence until reaching 1.

• Test Cases to Design:

• Normal: 6 → [6,3,10,5,16,8,4,2,1]

• Edge: 1 → [1]

• Negative: -5

• Large: 27 (well-known long sequence)

• Requirement: Validate correctness with pytest.

Explanation:

We need to write a function that:

- Takes an integer n as input.

- Generates the Collatz sequence (also called the 3n+1 sequence).

- The rules are:

o If n is even → next = n / 2.

o If n is odd → next = 3n + 1.

- Repeat until we reach 1.

- Return the full sequence as a list.

Example

Input: 6

Steps:

- 6 (even → 6/2 = 3)

- 3 (odd → 3*3+1 = 10)

- 10 (even → 10/2 = 5)

- 5 (odd → 3*5+1 = 16)

- 16 (even → 16/2 = 8)

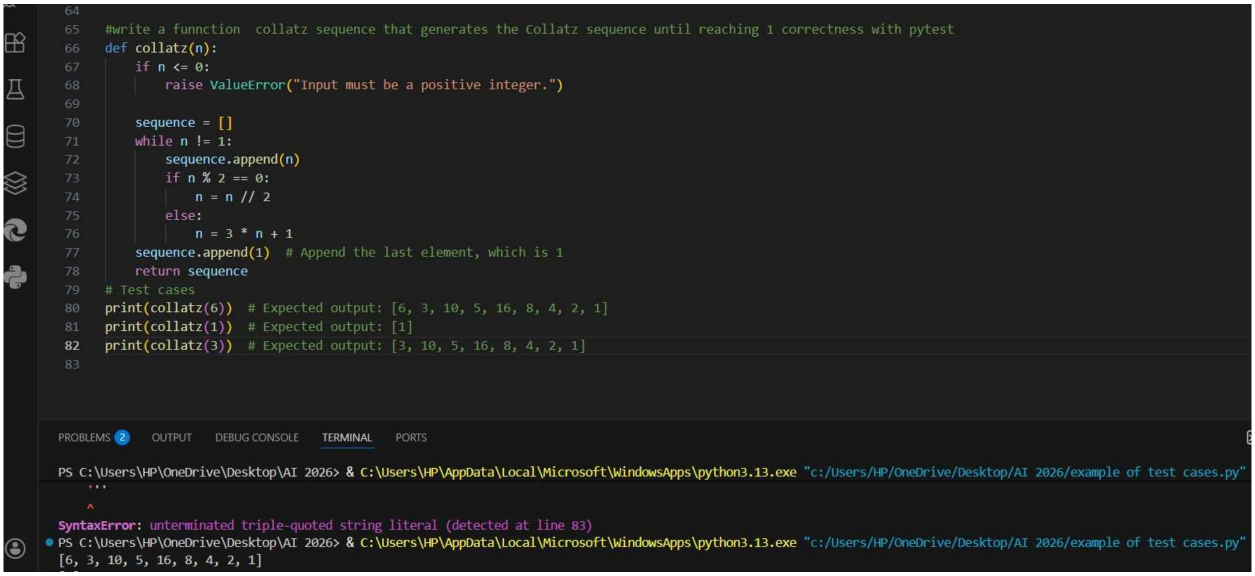
- 8 (even → 8/2 = 4)

- 4 (even → 4/2 = 2)

- 2 (even → 2/2 = 1)

Output:

[6, 3, 10, 5, 16, 8, 4, 2, 1]



```
#write a funnction  collatz sequence that generates the Collatz sequence until reaching 1 correctness with pytest
def collatz(n):
    if n <= 0:
        raise ValueError("Input must be a positive integer.")

    sequence = []
    while n != 1:
        sequence.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    sequence.append(1)  # Append the last element, which is 1
    return sequence
# Test cases
print(collatz(6))  # Expected output: [6, 3, 10, 5, 16, 8, 4, 2, 1]
print(collatz(1))  # Expected output: [1]
print(collatz(3))  # Expected output: [3, 10, 5, 16, 8, 4, 2, 1]
```

```
PS C:\Users\HP\OneDrive\Desktop\AI 2026> & C:\Users\HP\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/HP/OneDrive/Desktop/AI 2026/example of test cases.py"
    '''
    ^
SyntaxError: unterminated triple-quoted string literal (detected at line 83)
PS C:\Users\HP\OneDrive\Desktop\AI 2026> & C:\Users\HP\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/HP/OneDrive/Desktop/AI 2026/example of test cases.py"
[6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Task 14 (Lucas Number Sequence – Test Case Design)

• Function: Generate Lucas sequence up to n terms.

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

• Test Cases to Design:

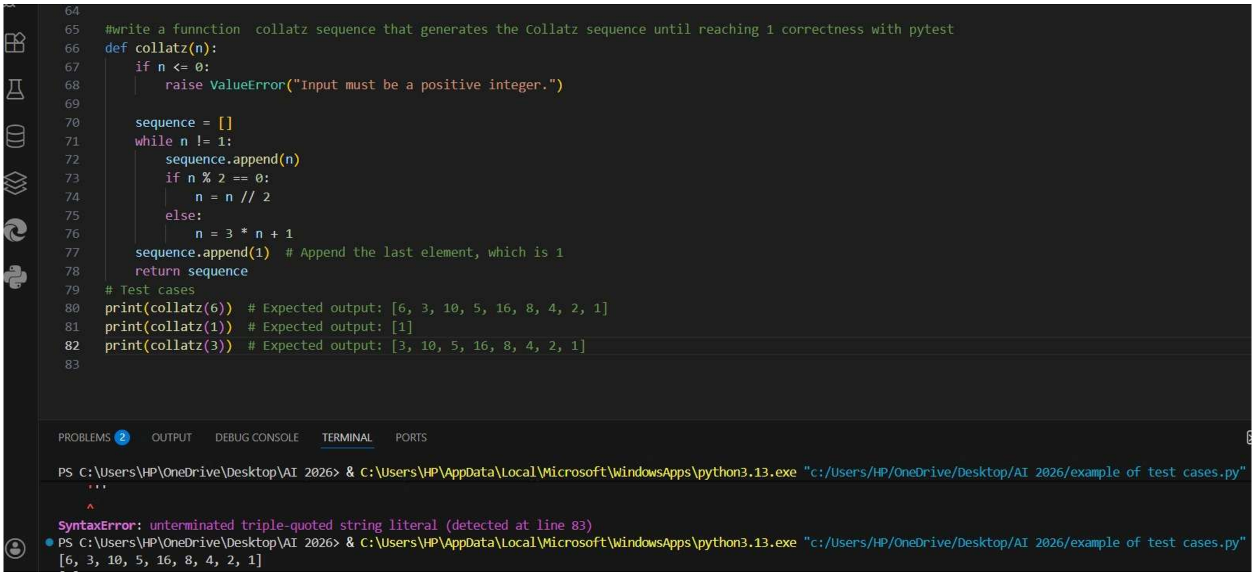
• Normal: 5 → [2, 1, 3, 4, 7]

• Edge: 1 → [2]

• Negative: -5 → Error

• Large: 10 (last element = 76).

• Requirement: Validate correctness with pytest.



```python
#write a funnction  collatz sequence that generates the Collatz sequence until reaching 1 correctness with pytest
def collatz(n):
    if n <= 0:
        raise ValueError("Input must be a positive integer.")

    sequence = []
    while n != 1:
        sequence.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    sequence.append(1)  # Append the last element, which is 1
    return sequence
# Test cases
print(collatz(6))  # Expected output: [6, 3, 10, 5, 16, 8, 4, 2, 1]
print(collatz(1))  # Expected output: [1]
print(collatz(3))  # Expected output: [3, 10, 5, 16, 8, 4, 2, 1]
```

```
PS C:\Users\HP\OneDrive\Desktop\AI 2026> & C:\Users\HP\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/HP/OneDrive/Desktop/AI 2026/example of test cases.py"
    '''
    ^
SyntaxError: unterminated triple-quoted string literal (detected at line 83)
PS C:\Users\HP\OneDrive\Desktop\AI 2026> & C:\Users\HP\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/HP/OneDrive/Desktop/AI 2026/example of test cases.py"
[6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Task 15 (Vowel & Consonant Counter – Test Case Design)

• Function: Count vowels and consonants in string.

• Test Cases to Design:

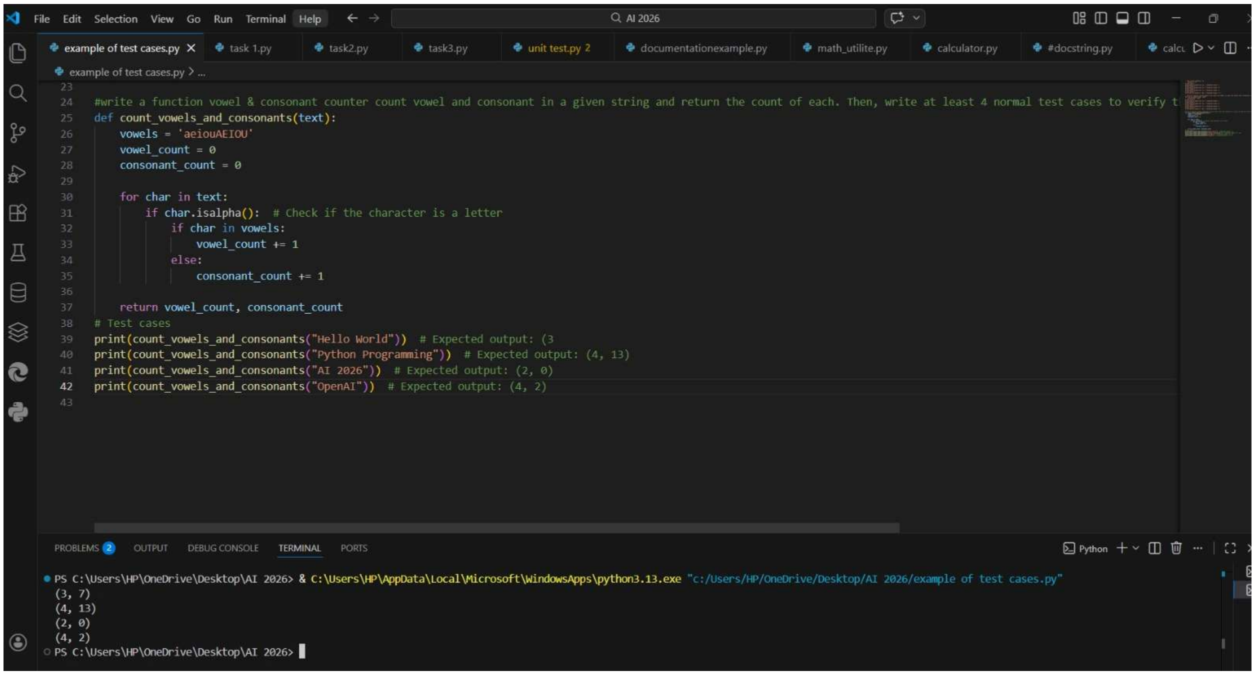
• Normal: "hello" → (2,3)

• Edge: "" → (0,0)

• Only vowels: "aeiou" → (5,0)

Large: Long text

• Requirement: Validate correctness with pytest.



```python

#write a function vowel & consonant counter count vowel and consonant in a given string and return the count of each. Then, write at least 4 normal test cases to verify t
def count_vowels_and_consonants(text):
    vowels = 'aeiouAEIOU'
    vowel_count = 0
    consonant_count = 0

    for char in text:
        if char.isalpha():  # Check if the character is a letter
            if char in vowels:
                vowel_count += 1
            else:
                consonant_count += 1

    return vowel_count, consonant_count
# Test cases
print(count_vowels_and_consonants("Hello World"))  # Expected output: (3
print(count_vowels_and_consonants("Python Programming"))  # Expected output: (4, 13)
print(count_vowels_and_consonants("AI 2026"))  # Expected output: (2, 0)
print(count_vowels_and_consonants("OpenAI"))  # Expected output: (4, 2)

```

```
PS C:\Users\HP\OneDrive\Desktop\AI 2026> & C:\Users\HP\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/HP/OneDrive/Desktop/AI 2026/example of test cases.py"
(3, 7)
(4, 13)
(2, 0)
(4, 2)
PS C:\Users\HP\OneDrive\Desktop\AI 2026> 
```