# Assignment-13

2303A51272

A.Srivani

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.

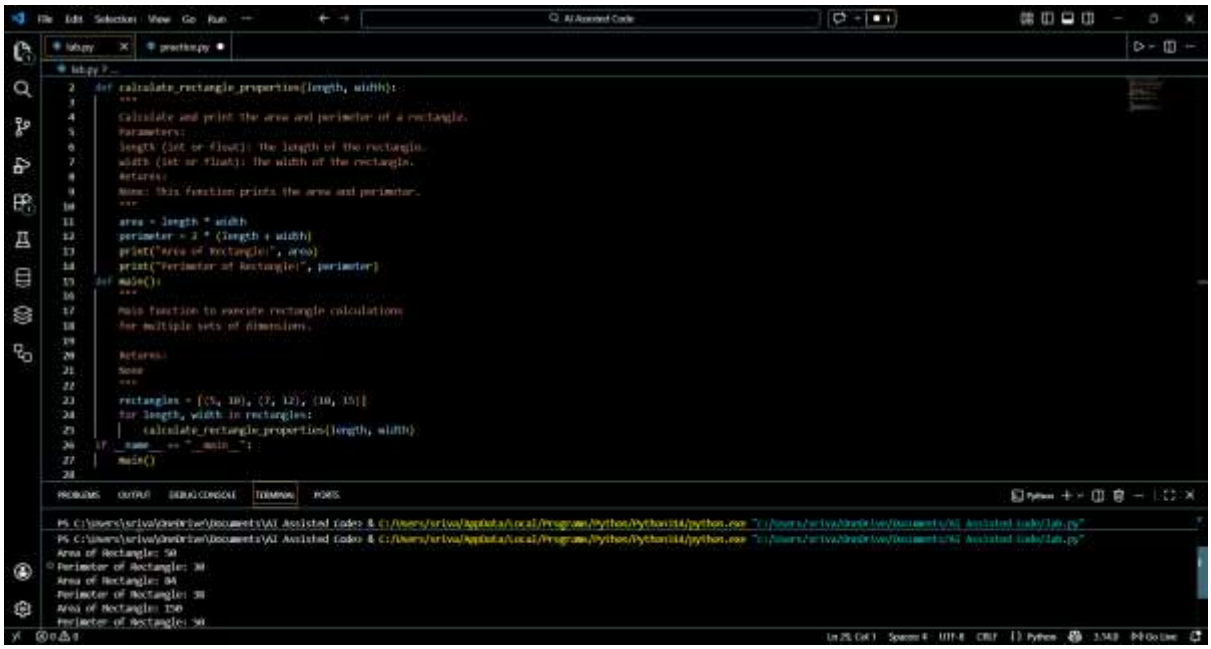
• Sample Legacy Code:

```
# 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




**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

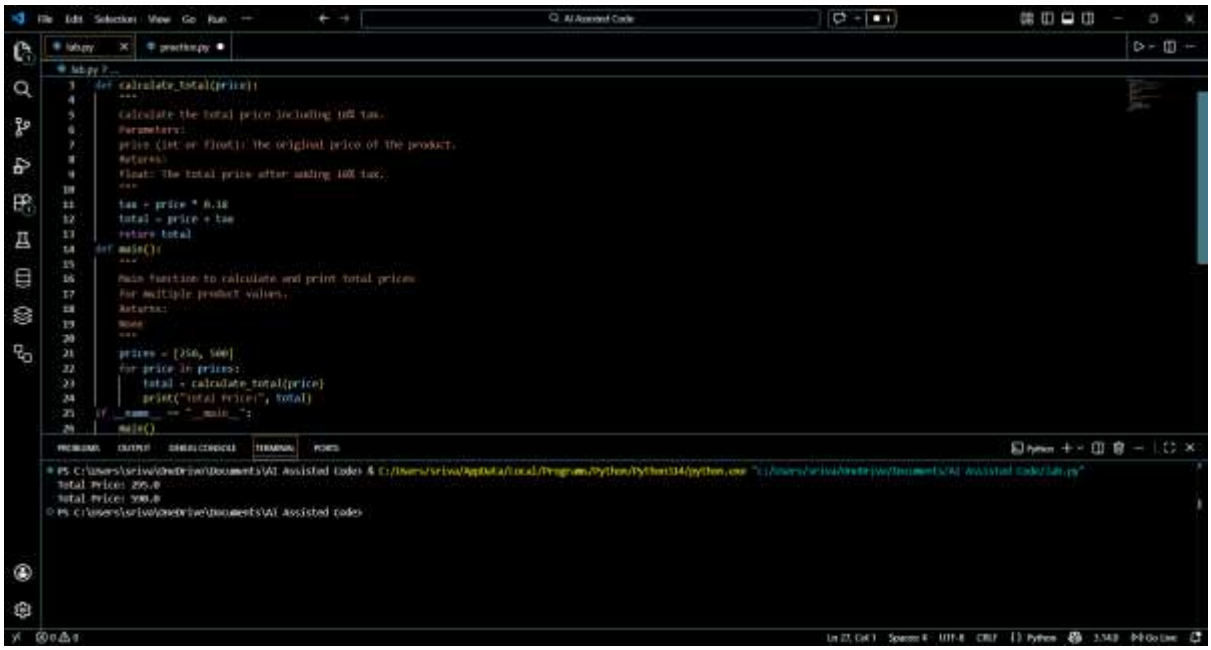
print("Total Price:", total)

• Expected Output:

o Code with a function calculate_total(price) that can be reused

for multiple price inputs.

o Well documented code



**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:

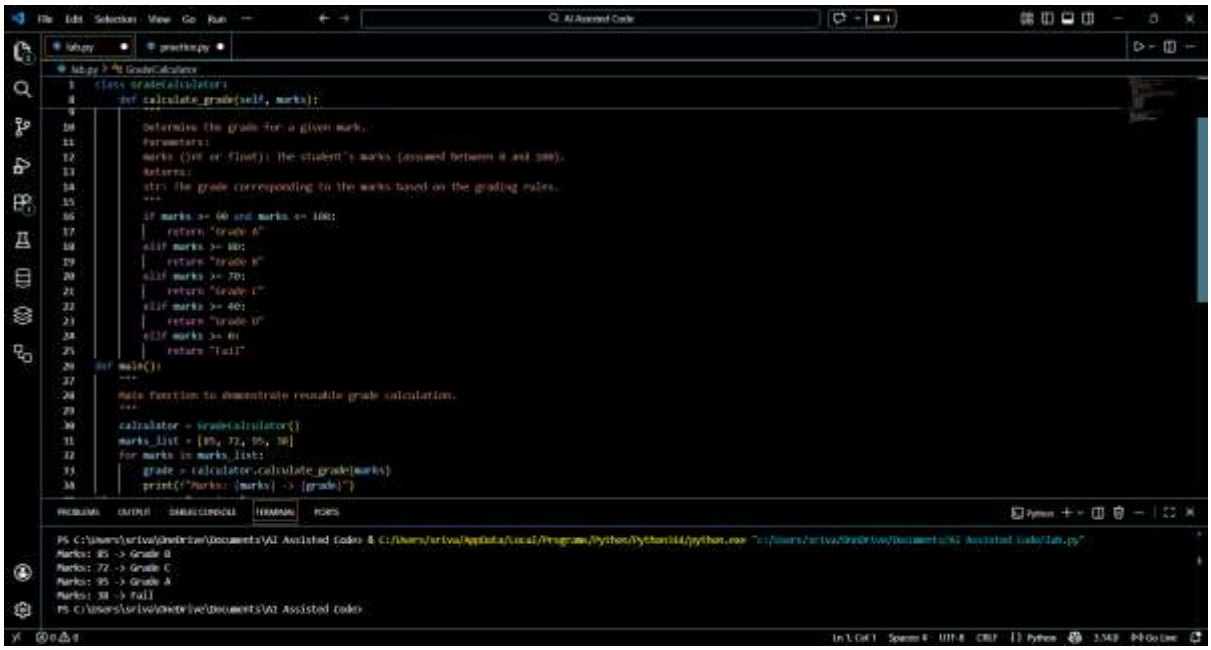
• Define a class named GradeCalculator.

• 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.




**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: "))

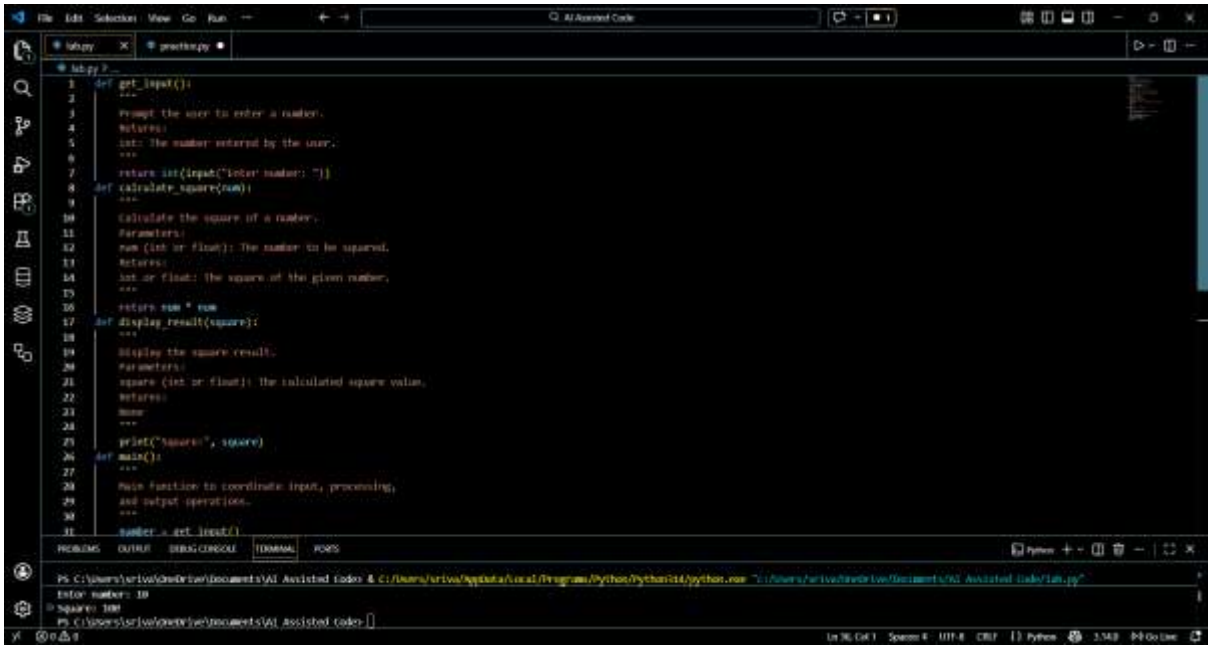
square = num * num

print("Square:", square)

• Expected Output:

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

display_result().




## 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

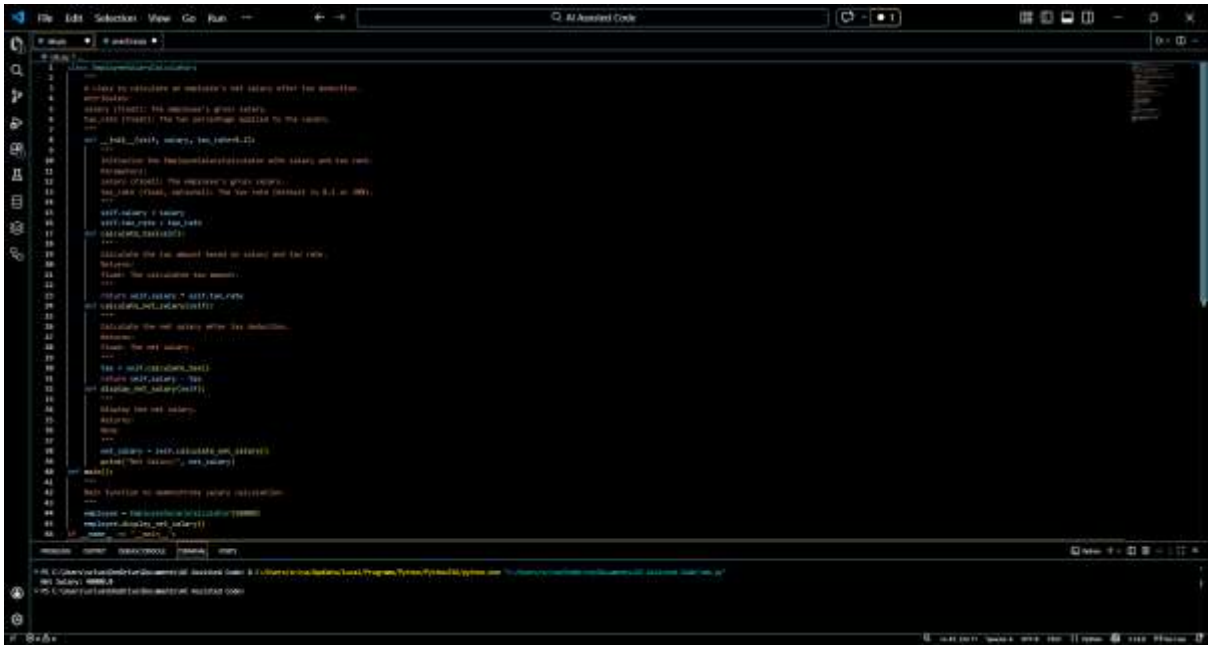
tax = salary * 0.2

net = salary - tax

print(net)

Expected Outcome:

o A class like EmployeeSalaryCalculator with methods and attributes.



## 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:

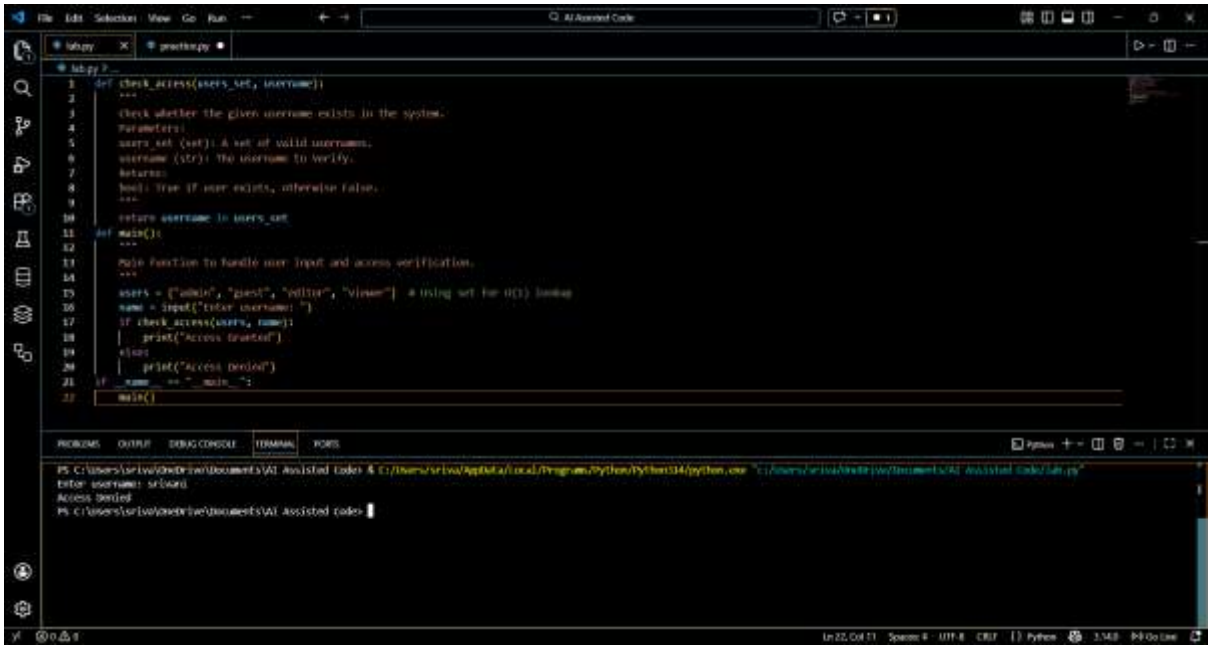
if u == name:

found = True

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

Expected Outcome:

o Use of sets or dictionaries with complexity justification




## 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.

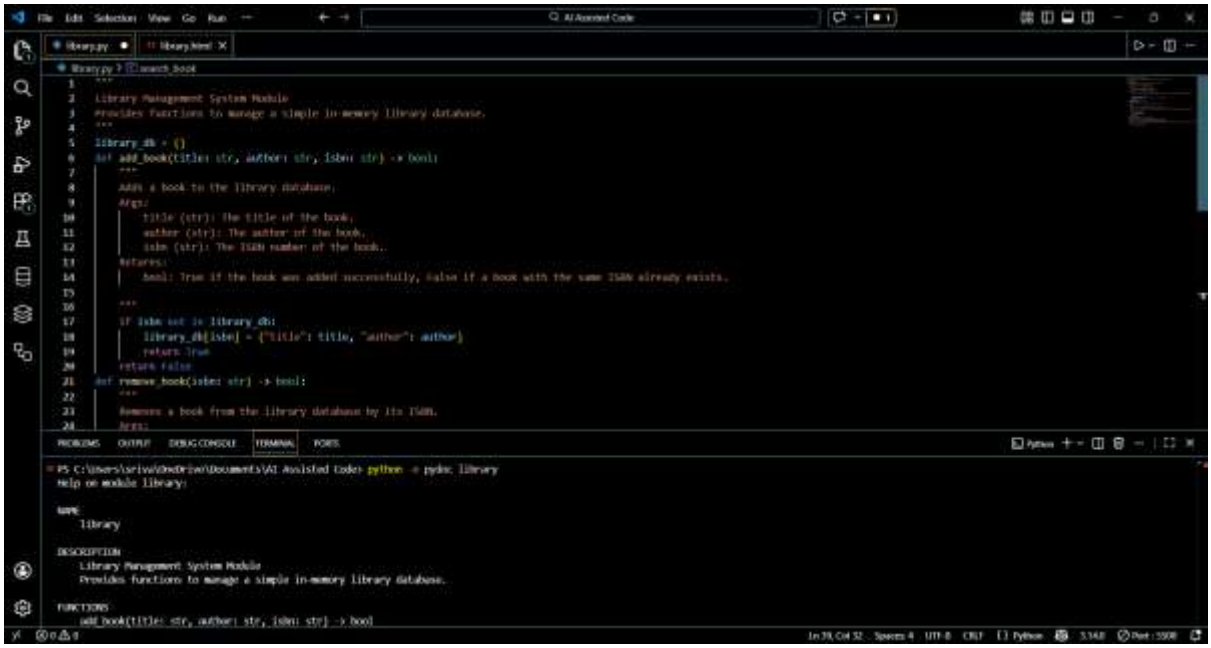
5. Open the file in a browser.

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.")
```



index

**library** c:\users\arva\onedrive\documents\ai assisted code\library.py

Library Management System Module
Provides functions to manage a simple in-memory library database.

**Functions**

**add_book**(title: str, author: str, isbn: str) -> bool
Adds a book to the library database.
Args:
title (str): The title of the book.
author (str): The author of the book.
isbn (str): The ISBN number of the book.
Returns:
bool: True if the book was added successfully, False if a book with the same ISBN already exists.

**remove_book**(isbn: str) -> bool
Removes a book from the library database by its ISBN.
Args:
isbn (str): The ISBN number of the book to be removed.
Returns:
bool: True if the book was removed successfully, False if the book was not found in the database.

**search_book**(isbn: str) -> dict | None
Searches for a book in the library database by its ISBN.
Args: isbn (str): The ISBN number of the book to search for.
Returns: dict | None: A dictionary containing the book's title and author if found, or None if the book is not found in the database.

**Data**
**library_db** = {}

## 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:

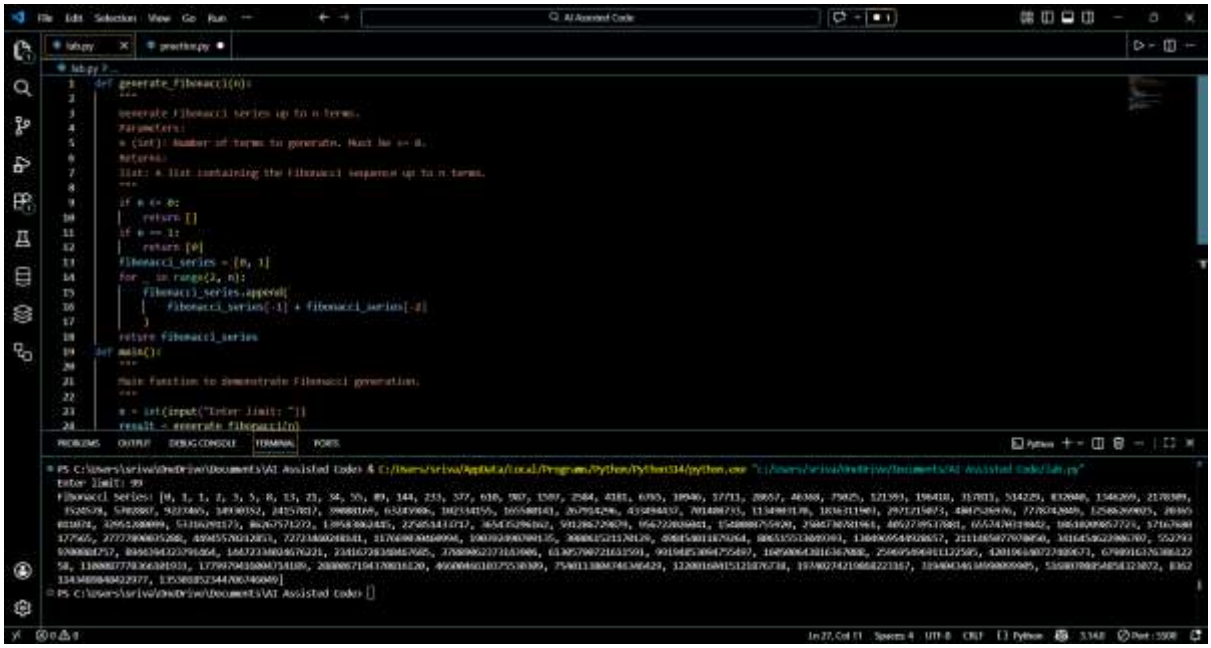
• Refactor into a clean reusable function (generate_fibonacci).

• 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.

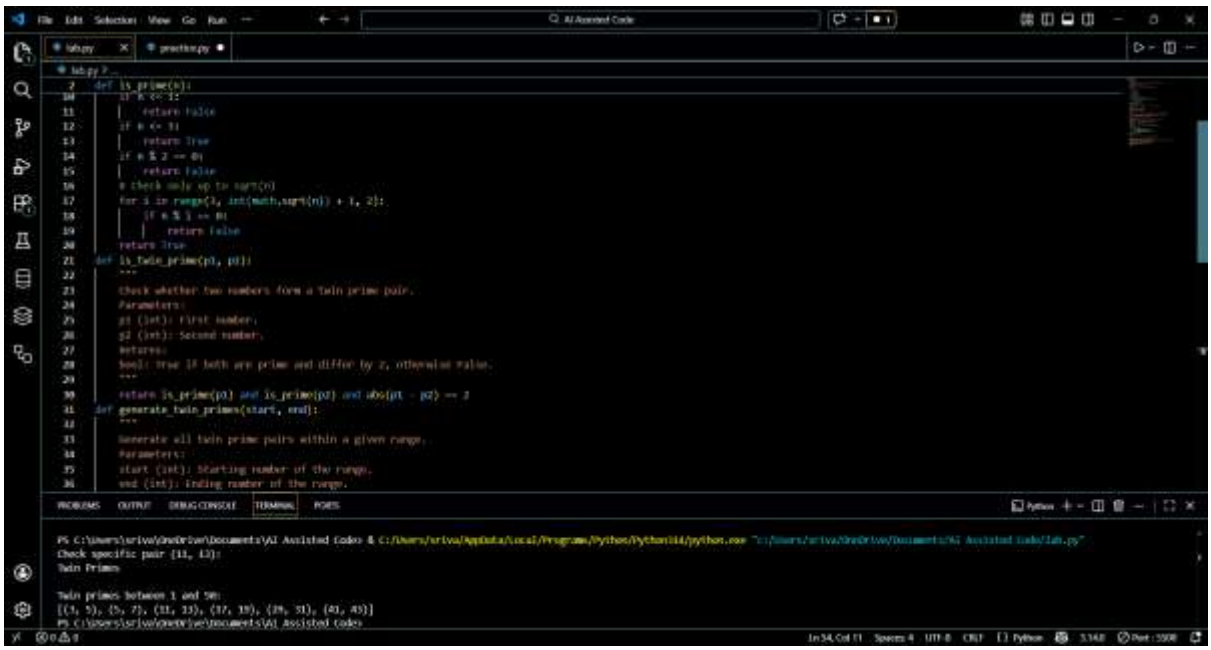
Bad Code Version:

# 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")
```



## 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

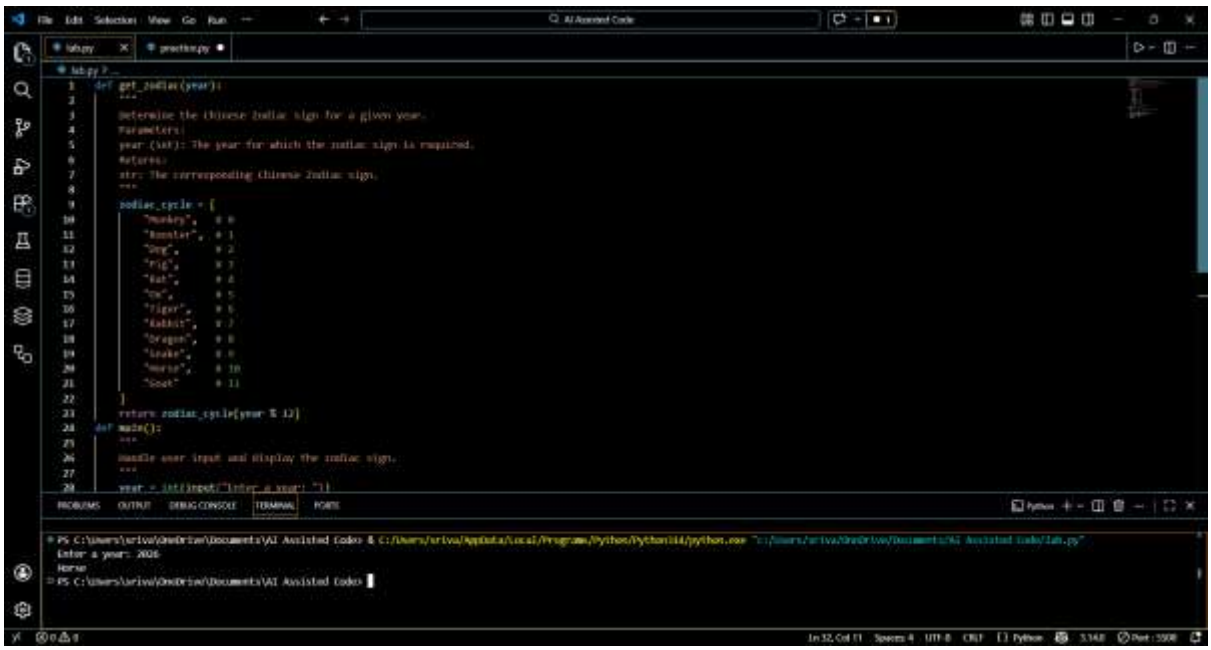
dictionary).

3. Add proper docstrings.

4. Separate input handling from logic.

5. Improve readability and maintainability.

6. Ensure output remains correct.




## 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:

o Accept an integer parameter.

o Return True if the number is divisible by the sum of its digits.

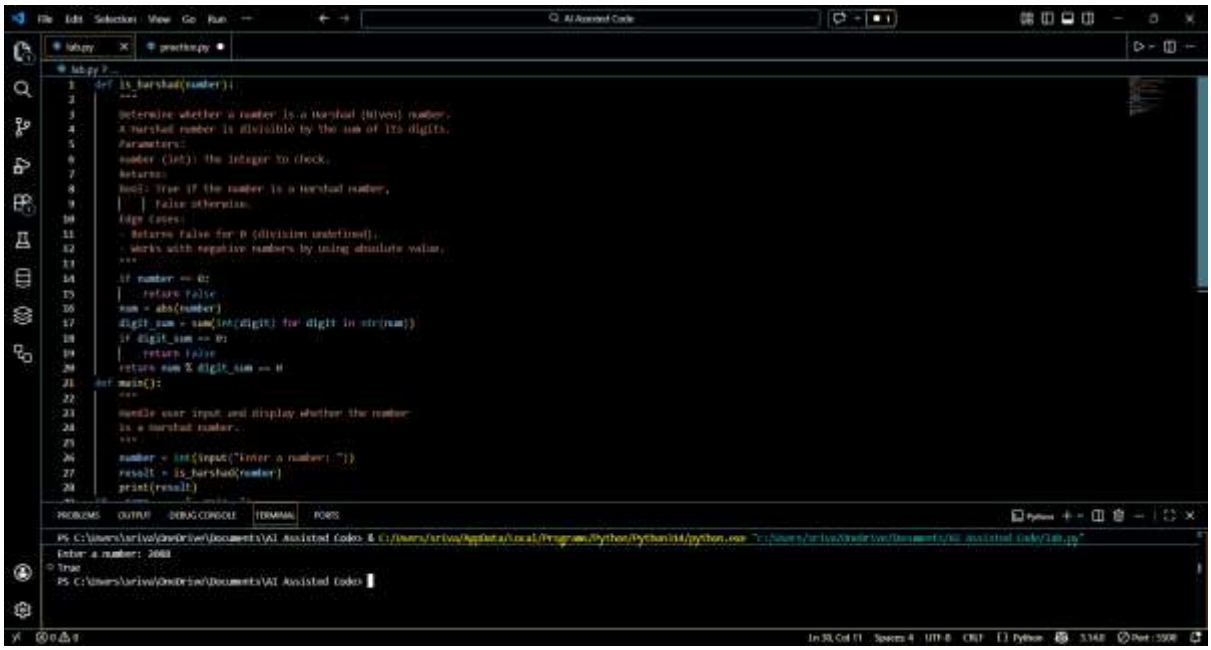
o Return False otherwise.

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).



## 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:

o Accept a non-negative integer n.

o Return the number of trailing zeros in n!.

3. Do NOT compute the full factorial.

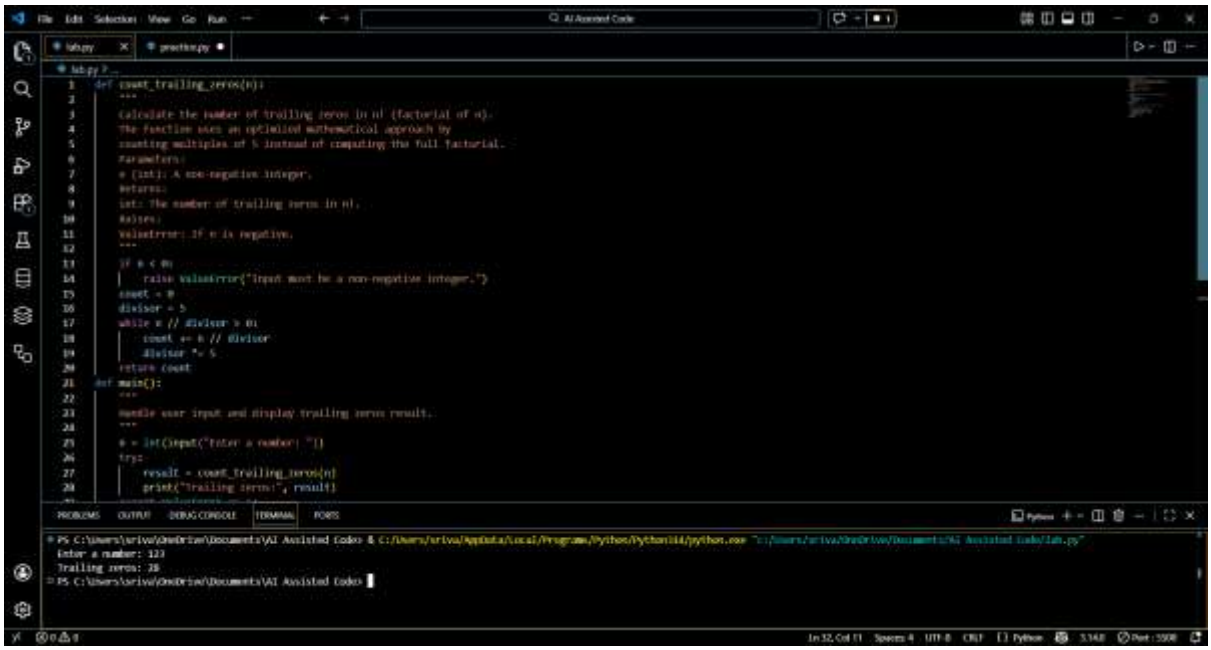
4. Use an optimized mathematical approach (count multiples of 5).

5. Add proper docstrings.

6. Separate user interaction from core logic.

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

Test Cases Design




## 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)

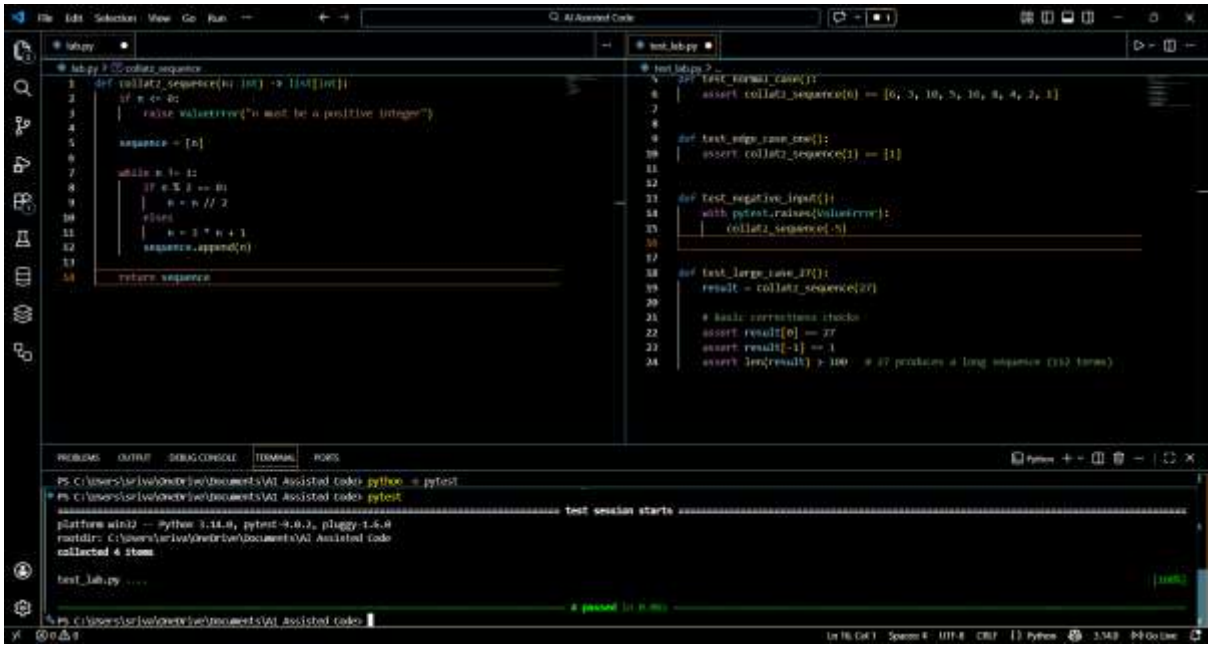
• 8 (even → 8/2 = 4)

• 4 (even → 4/2 = 2)

• 2 (even → 2/2 = 1)

Output:

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



## 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:

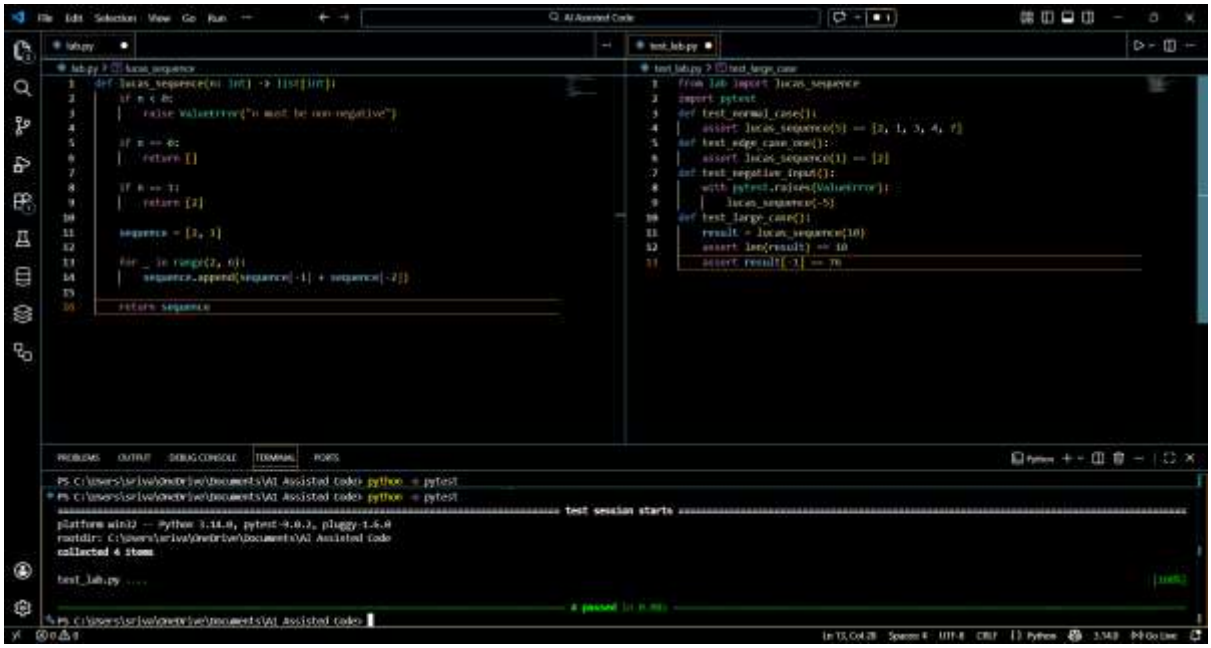
• Normal: 5 → [2, 1, 3, 4, 7]

• Edge: 1 → [2]

• Negative: -5 → Error

• Large: 10 (last element = 76).

• Requirement: Validate correctness with pytest.



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

• Function: Count vowels and consonants in string.

• Test Cases to Design:

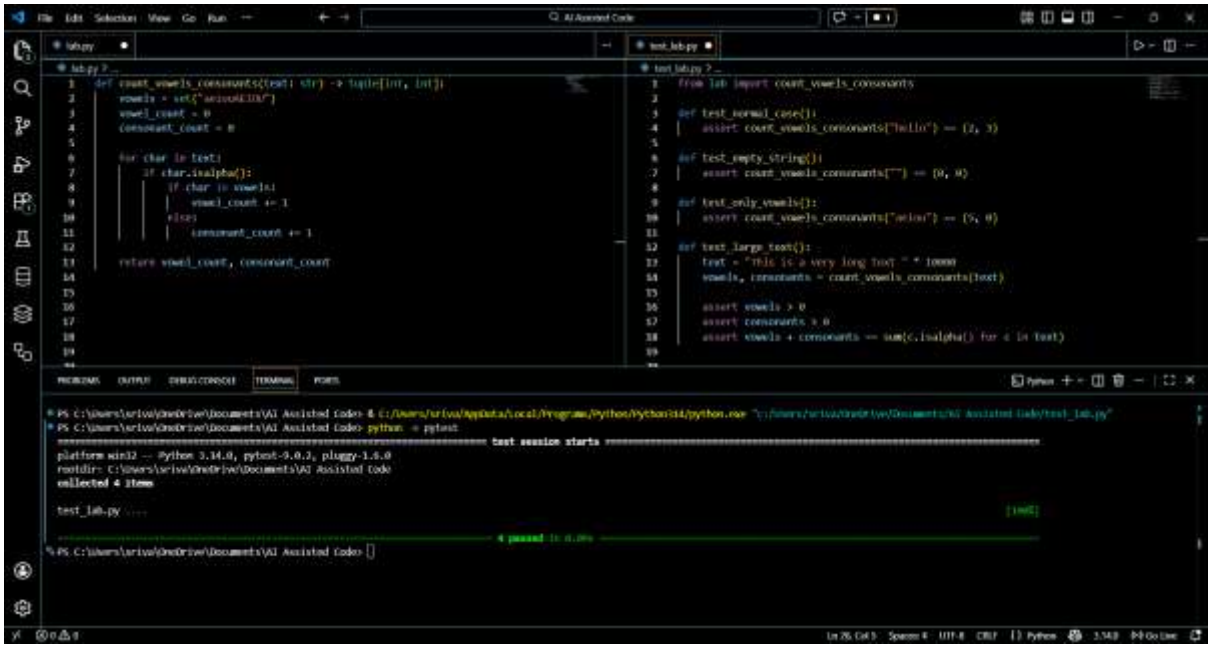
• Normal: "hello" → (2,3)

• Edge: "" → (0,0)

• Only vowels: "aeiou" → (5,0)

Large: Long text

• Requirement: Validate correctness with pytest.


lab.py
def count_vowels_consonants(text: str) -> tuple[int, int]:
vowels = set("aeiouAEIOU")
vowel_count = 0
consonant_count = 0
for char in text:
if char.isalpha():
if char in vowels:
vowel_count += 1
else:
consonant_count += 1
return vowel_count, consonant_count
test_lab.py
from lab import count_vowels_consonants
def test_normal_case():
assert count_vowels_consonants("hello") == (2, 3)
def test_empty_string():
assert count_vowels_consonants("") == (0, 0)
def test_only_vowels():
assert count_vowels_consonants("aeiou") == (5, 0)
def test_large_text():
text = "this is a very long text " * 10000
vowels, consonants = count_vowels_consonants(text)
assert vowels > 0
assert consonants > 0
assert vowels + consonants == sum(c.isalpha() for c in text)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS C:\Users\sriva\OneDrive\Documents\AI Assisted Code> & C:/Users/sriva/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/sriva/OneDrive/Documents/AI Assisted Code/test_lab.py"
PS C:\Users\sriva\OneDrive\Documents\AI Assisted Code> python -m pytest
test session starts
platform win32 -- Python 3.14.0, pytest-9.0.2, pluggy-1.6.0
rootdir: C:\Users\sriva\OneDrive\Documents\AI Assisted Code
collected 4 items
test_lab.py ....
[100%]
4 passed in 0.09s
PS C:\Users\sriva\OneDrive\Documents\AI Assisted Code>