SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 2

Program	**: B. Tech (CSE)**
Specialization	**: -**
Course Title	**: AI Assisted Coding**
Course Code	**: 23CS002PC304**
Semester	**: II**
Academic Session	**: 2025-2026**
Name of Student	**: J Harshith yadav**
Enrollment No.	**: 2403A51L49**
Batch No.	**: 52**
Date	**: 09/01/26**

Submission Starts here

Screenshots:

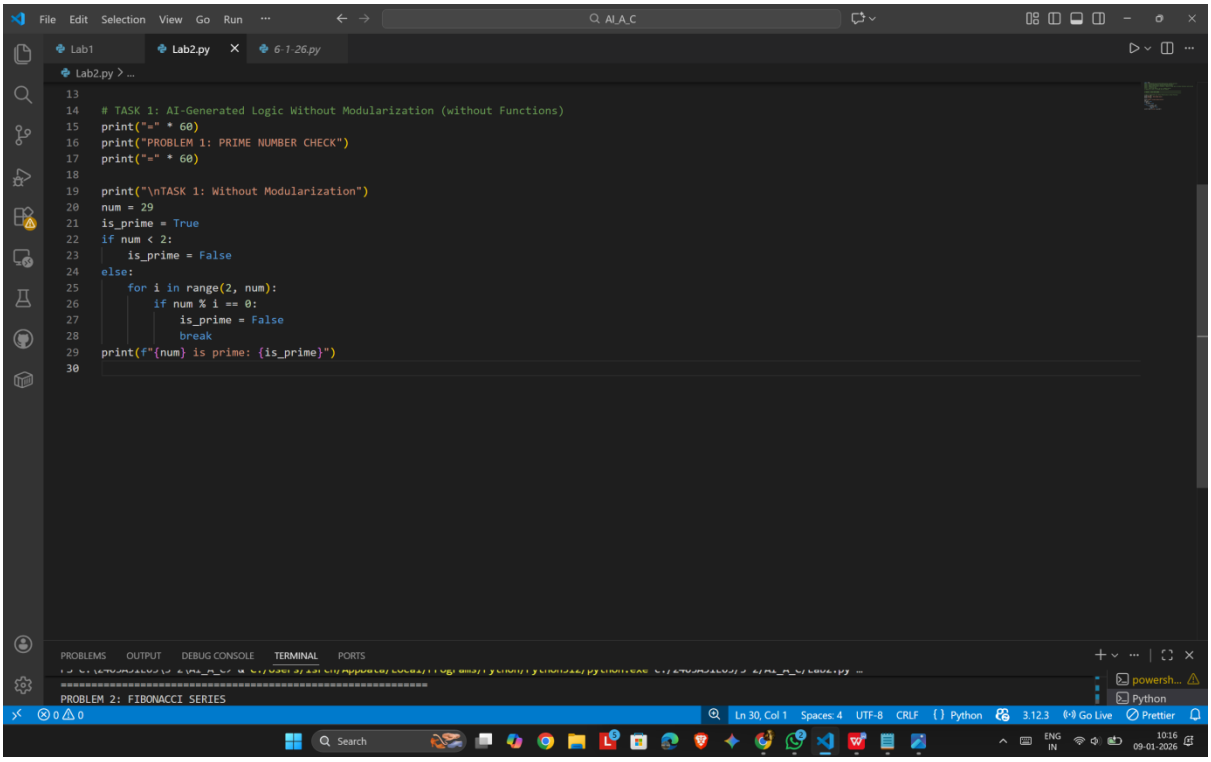
Problem 1- Check for Prime

TASK-1:

Prompt :

 # TASK 1: AI-Generated Logic Without Modularization (Check for Prime without using Functions)

Code:



Output:

```
TASK 1: Without Modularization
29 is prime: True
```

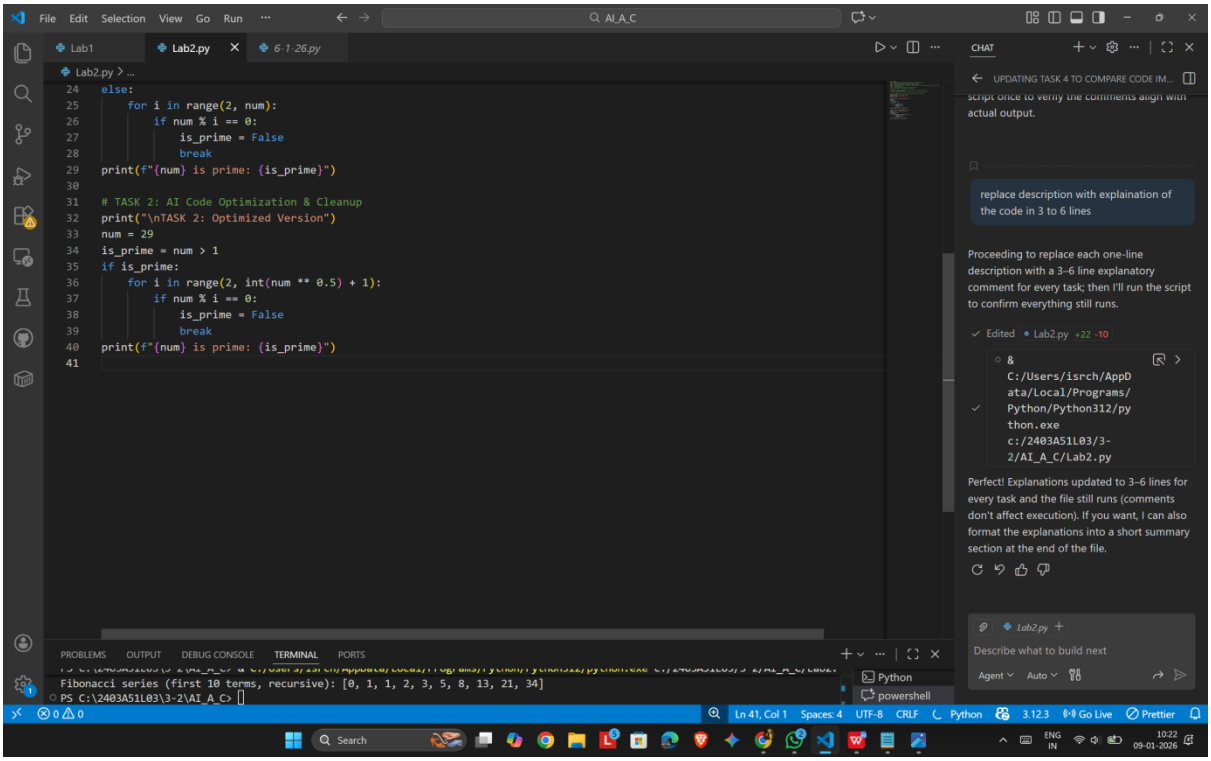
Explanation:

This procedural prime check loops from 2 to n-1 testing divisibility and breaks on the first divisor.
It uses a boolean flag to track primality and does not encapsulate logic in a function.
For 29 no divisors are found, so the printed result is True.

TASK-2:

Prompt:
 # TASK 2: AI Code Optimization & Cleanup

Code:



Output:

```
TASK 2: Optimized Version
29 is prime: True
```

Explanation:

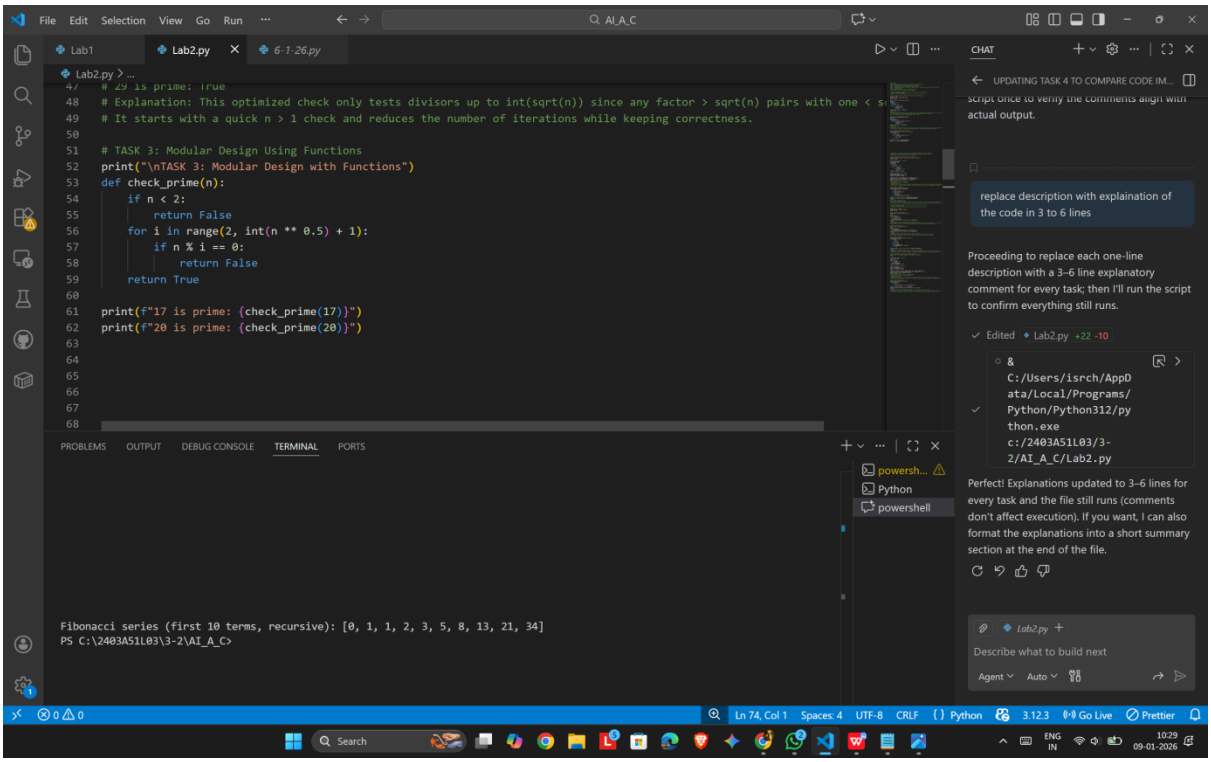
This optimized check only tests divisors up to int(sqrt(n)) since any factor > sqrt(n) pairs with one < sqrt(n).
It starts with a quick n > 1 check and reduces the number of iterations while keeping correctness.

SR UNIVERSITY

TASK-3:

Prompt:
 # TASK 3: Modular Design Using Functions

Code:



Output:

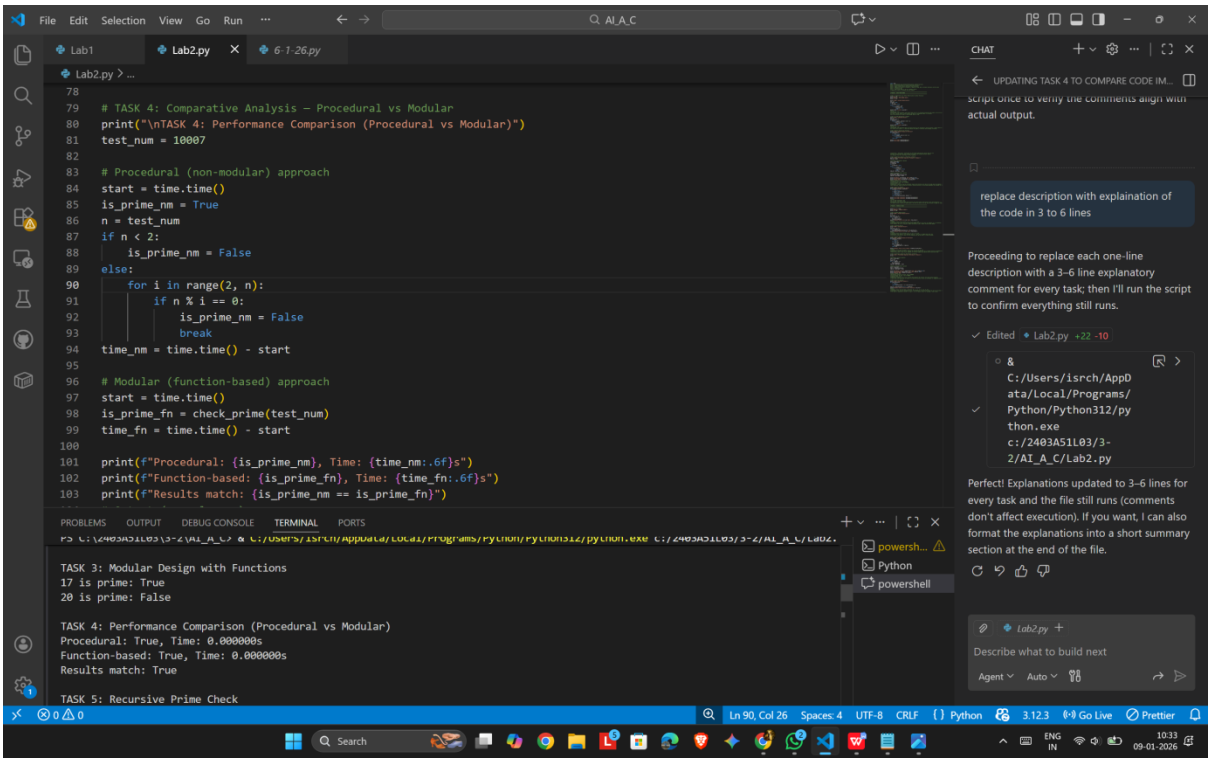
```
TASK 3: Modular Design with Functions
17 is prime: True
20 is prime: False
```

Explaination:
'check_prime' encapsulates the sqrt-based algorithm and returns False for n<2.
Using a function makes the logic reusable and clearer when checking multiple numbers.
It produces True for 17 and False for 20 as expected.

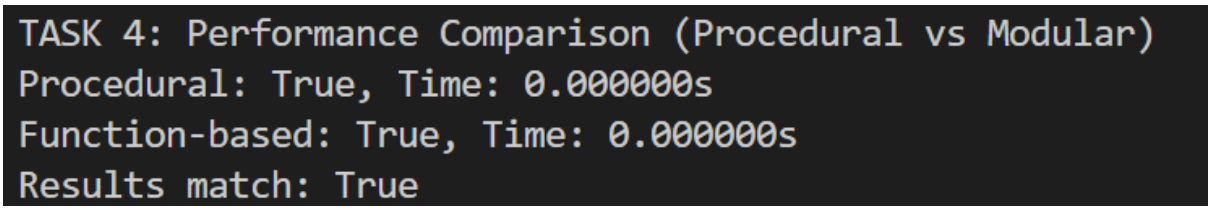
TASK -4:

Prompt:
 # TASK 4: Comparative Analysis — Procedural vs Modular

Code:



Output:



Explanation:

Times how long the non-modular loop and the function call take on the same input and compares results.
Both approaches compute primality and agree; timing differences are environment-dependent and intended for basic comparison.

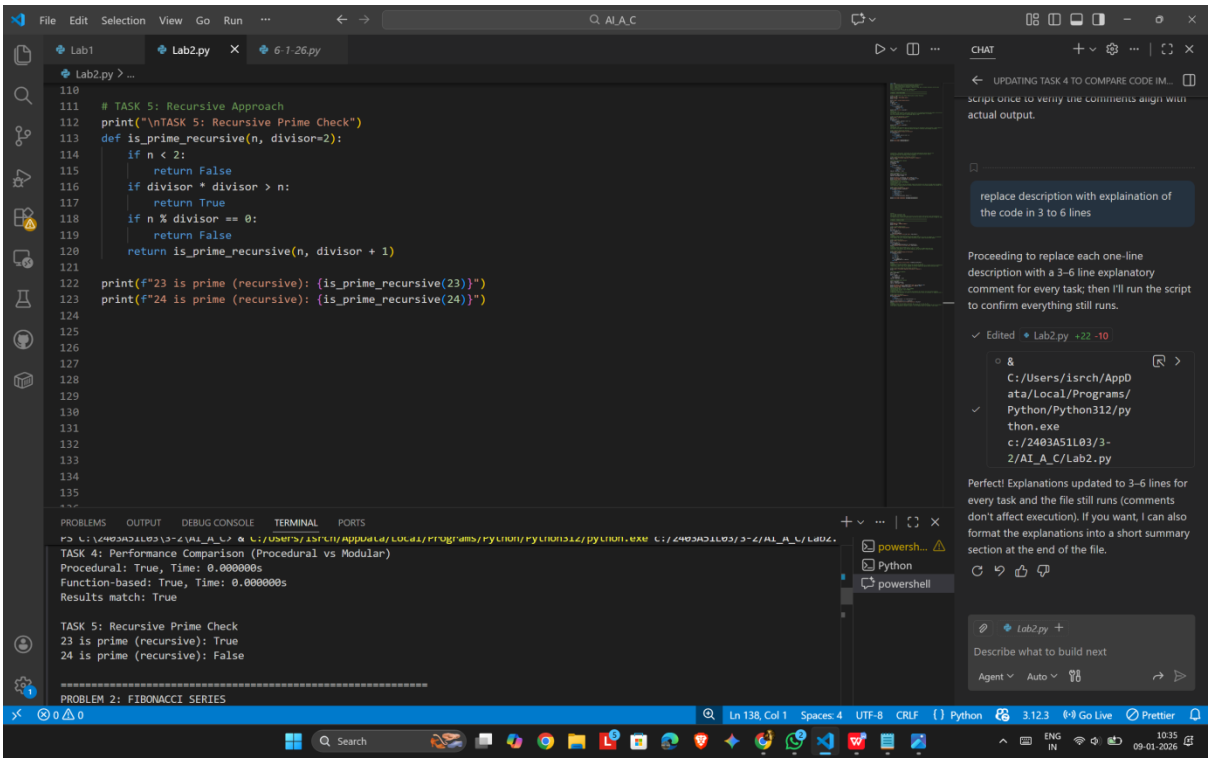
TASK - 5:

Prompt:

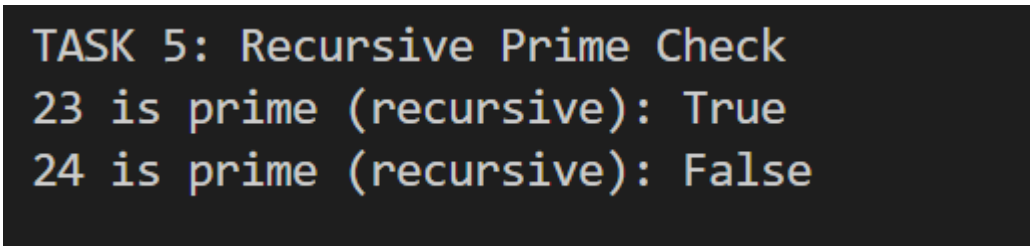
 # TASK 5: Recursive Approach

SR UNIVERSITY

Code:



Output:



Explanation:

The recursive check tests divisors by calling itself with divisor+1 until divisor*divisor > n.
It returns False on the first found divisor; this form is clear but can be less efficient or deeper on large n.

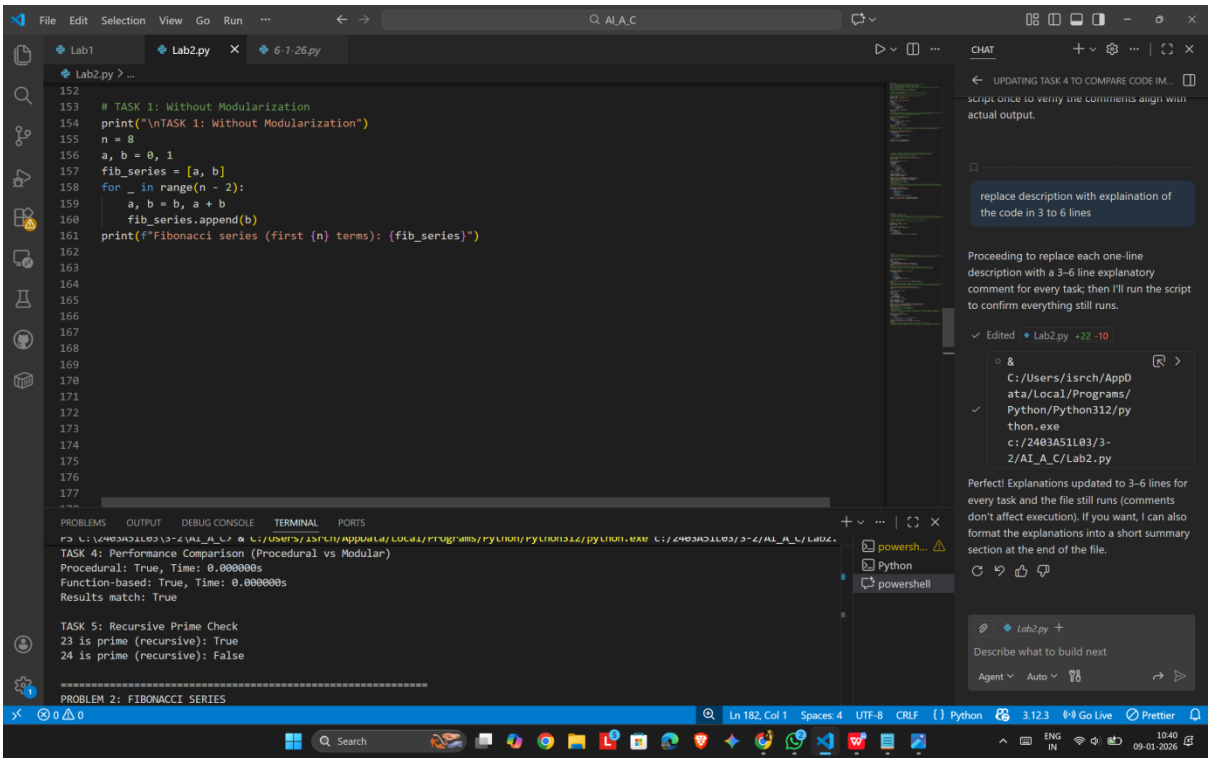
SR UNIVERSITY

Problem -2 : Fibonacci Series

TASK-1 :

Prompt:
 # TASK 1: Without Modularization

Code:



Output:

```
TASK 1: Without Modularization
Fibonacci series (first 8 terms): [0, 1, 1, 2, 3, 5, 8, 13]
```

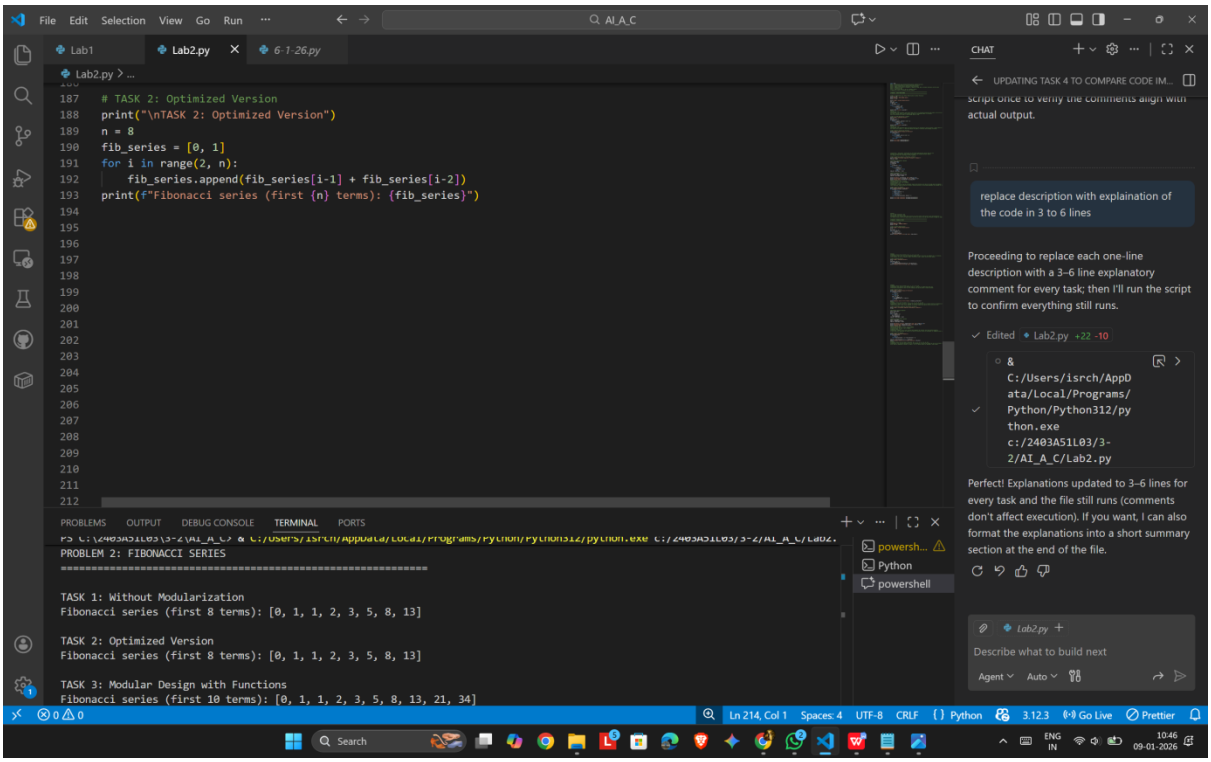
Explanation:
Starts with [0,1] and iteratively appends the sum of the last two elements for n-2 iterations. This produces the first n Fibonacci numbers efficiently using a simple loop and tuple updates.

TASK-2 :

Prompt:
 # TASK 2: Optimized Version

SR UNIVERSITY

Code:



Output:

```
TASK 2: Optimized Version
Fibonacci series (first 8 terms): [0, 1, 1, 2, 3, 5, 8, 13]
```

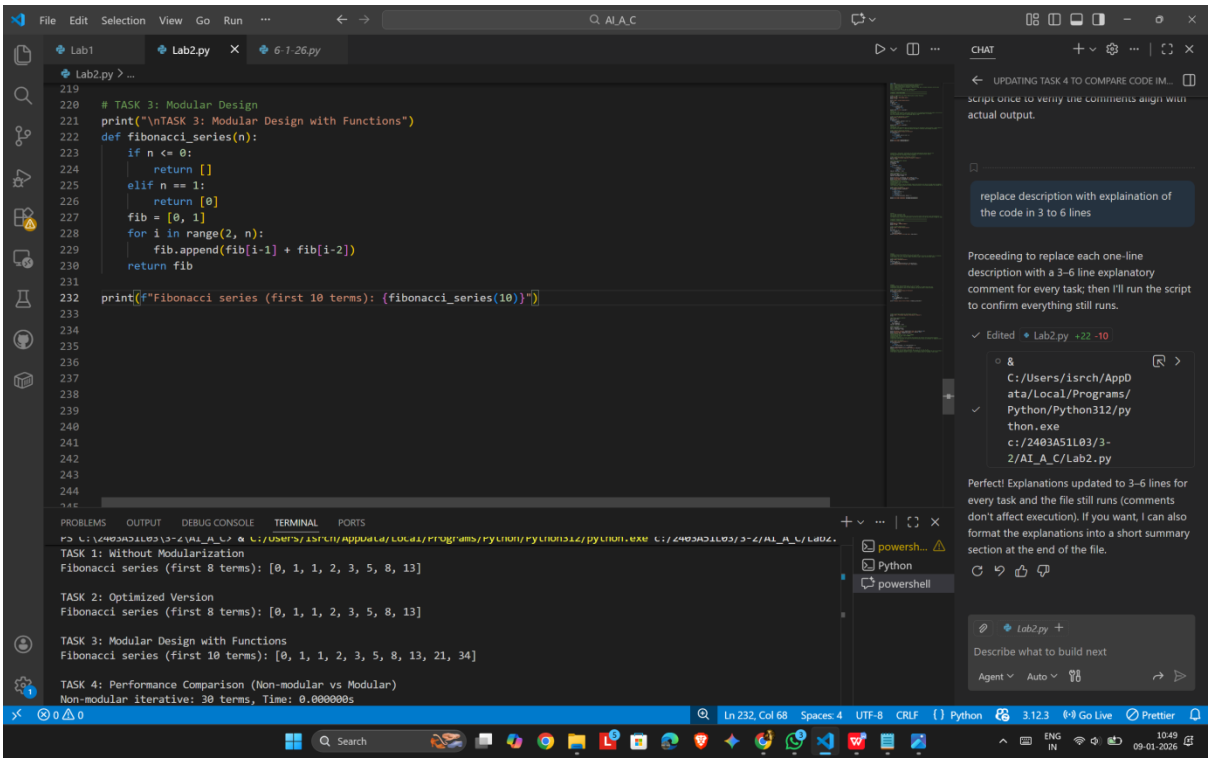
Explanation:
Uses explicit list indexing (fib[i-1] + fib[i-2]) to compute each next term.
Functionally equivalent to Task 1 but the indexing style may be easier to read and extend.

TASK - 3:

Prompt:
 # TASK 3: Modular Design

SR UNIVERSITY

Code:



Output:

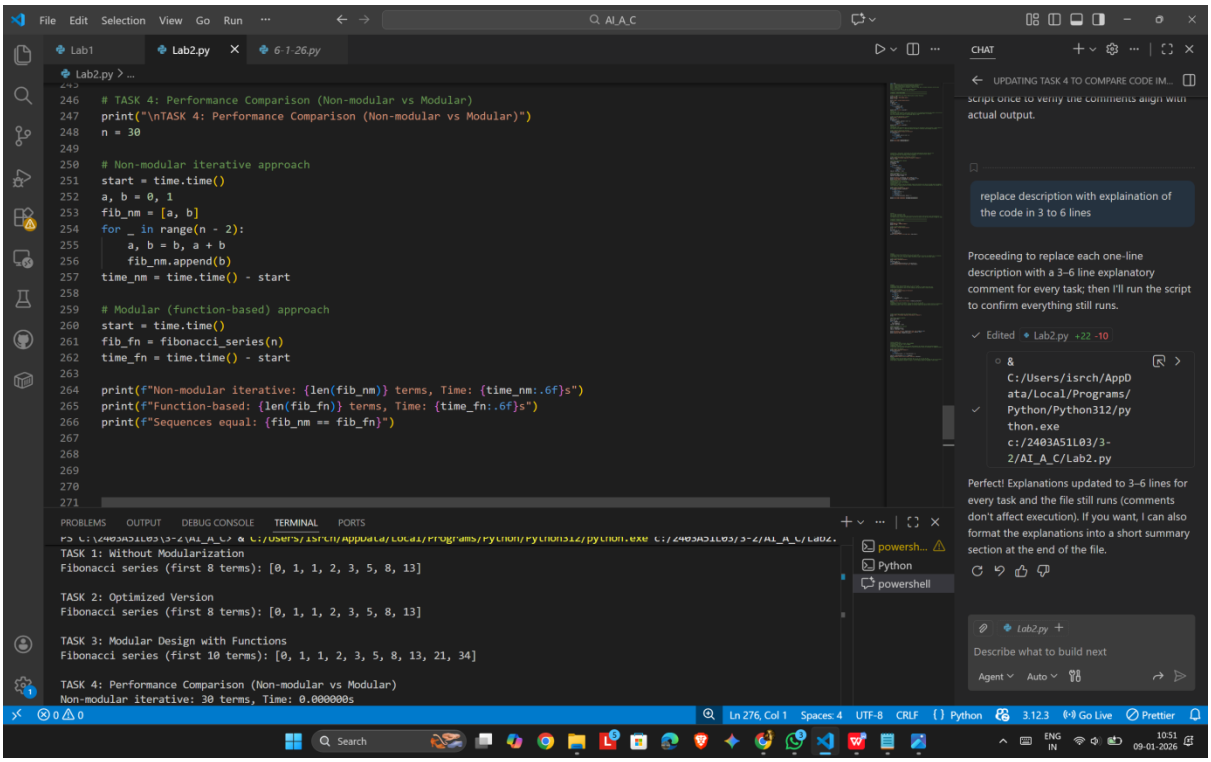
```
TASK 3: Modular Design with Functions
Fibonacci series (first 10 terms): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Explanation:
`fibonacci_series(n)` packages the iterative generation into a reusable function and handles edge cases.
Modularizing makes it easy to get sequences of different lengths and improves code clarity and reuse.

TASK-4:

Prompt:
 # TASK 4: Performance Comparison (Non-modular vs Modular)

Code:



Output:

```
TASK 4: Performance Comparison (Non-modular vs Modular)
Non-modular iterative: 30 terms, Time: 0.000000s
Function-based: 30 terms, Time: 0.000000s
Sequences equal: True
```

Explanation: Compares the in-line iterative build against the function result and confirms equality.
Timing shows both approaches are similar for n=30; the function version is preferable for readability and reuse.

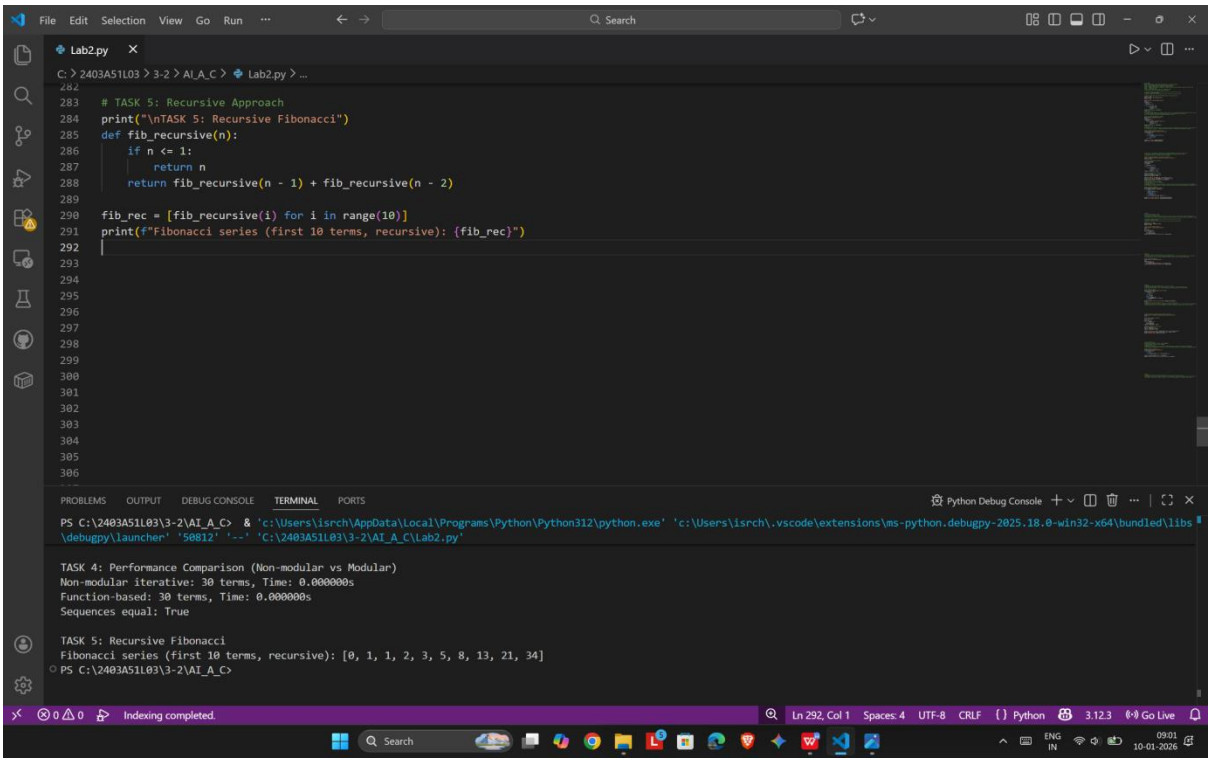
TASK -5:

Prompt:

 # TASK 5: Recursive Approach

SR UNIVERSITY

Code:



Output:

```
TASK 5: Recursive Fibonacci
Fibonacci series (first 10 terms, recursive): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\2403A51L03\3-2\AI_A_C>
```

Explanation:

The recursive definition mirrors the mathematical recurrence but makes two calls per non-base case.

This leads to exponential runtime for larger n, so it is mainly useful for teaching or small inputs.