

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	: Ganesh
Enrollment No.	: 2403A51L55
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:

The screenshot shows a code editor window with a dark theme. The file 'Lab2.py' is open, containing the following Python code:

```
13
14 # TASK 1: AI-Generated Logic Without Modularization (without Functions)
15 print("=" * 60)
16 print("PROBLEM 1: PRIME NUMBER CHECK")
17 print("=" * 60)
18
19 print("\nTASK 1: Without Modularization")
20 num = 29
21 is_prime = True
22 if num < 2:
23     is_prime = False
24 else:
25     for i in range(2, num):
26         if num % i == 0:
27             is_prime = False
28             break
29 print(f"\n{num} is prime: {is_prime}")
30
```

The code defines a function to check if a number is prime by iterating from 2 to the number itself and checking for divisibility. If a divisor is found, it sets the is_prime flag to False and breaks the loop. Finally, it prints the result.

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:

```

# TASK 2: AI Code Optimization & Cleanup
print("\nTASK 2: Optimized Version")
num = 29
is_prime = num > 1
if is_prime:
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
print(f"{num} is prime: {is_prime}")

```

Output:

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

Explanation:

This optimized check only tests divisors up to $\text{int}(\sqrt{n})$ since any factor $> \sqrt{n}$ pairs with one $< \sqrt{n}$.

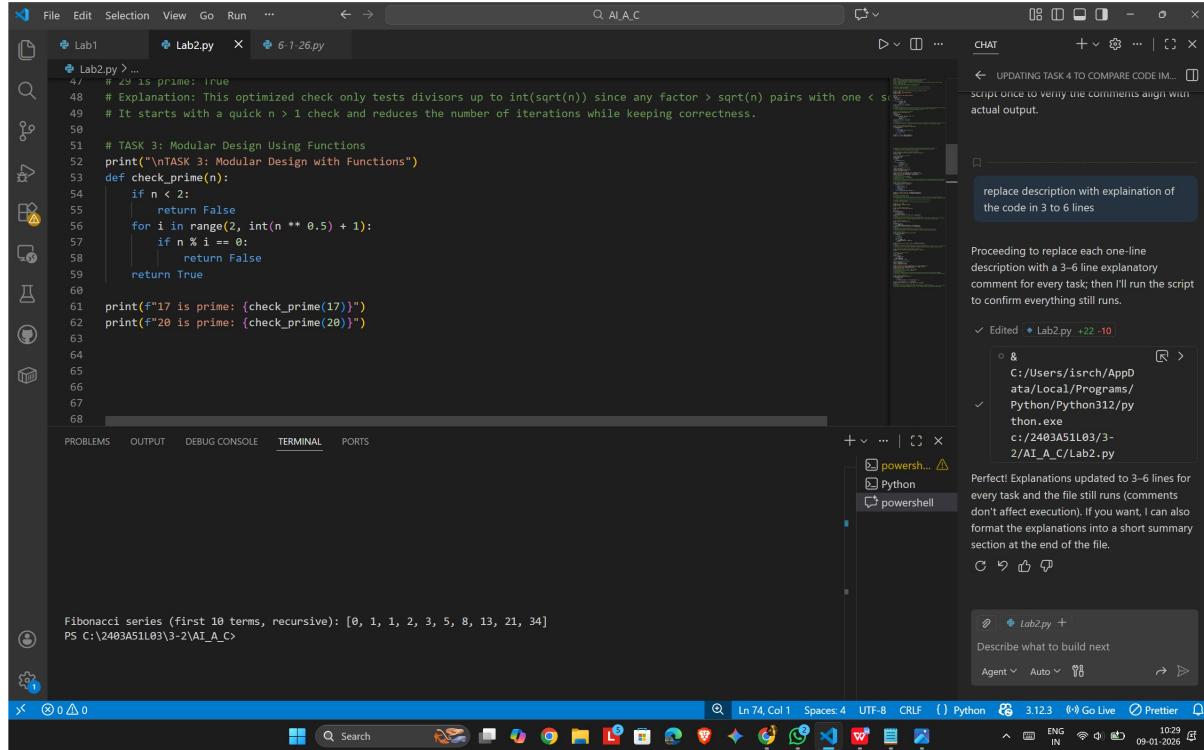
It starts with a quick $n > 1$ check and reduces the number of iterations while keeping correctness.

TASK-3:

Prompt:

TASK 3: Modular Design Using Functions

Code:



```

Lab1          Lab2.py      6-1-26.py
Lab2.py > ...
4/   # 29 is prime: true
48  # Explanation: This optimized check only tests divisors up to int(sqrt(n)) since any factor > sqrt(n) pairs with one < sqrt(n).
49  # It starts with a quick n > 1 check and reduces the number of iterations while keeping correctness.
50
51  # TASK 3: Modular Design Using Functions
52  print("\nTASK 3: Modular Design with Functions")
53  def check_prime(n):
54      if n < 2:
55          return False
56      for i in range(2, int(n ** 0.5) + 1):
57          if n % i == 0:
58              return False
59      return True
60
61  print("17 is prime: {}".format(check_prime(17)))
62  print("20 is prime: {}".format(check_prime(20)))
63
64
65
66
67
68

```

Fibonacci series (first 10 terms, recursive): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Output:

```

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

```

Explanation:

‘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:

TASK 4: Performance Comparison (Procedural vs Modular)
Procedural: True, Time: 0.00000s
Function-based: True, Time: 0.00000s
Results match: True

Explanation:

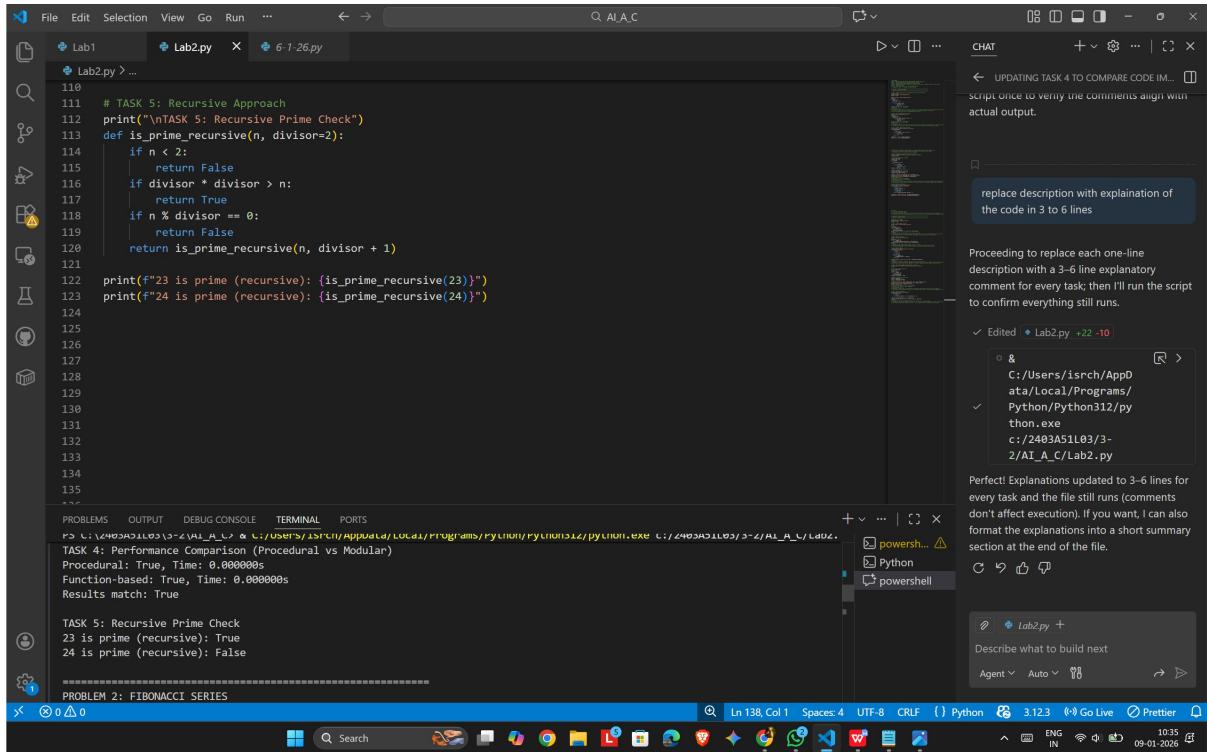
Times how long the non-modular loop and the function call take on the same input and compares results.
Both approaches compute primality and agree; timing differences are environment-dependent and intended for basic comparison.

TASK - 5:

Prompt:

TASK 5: Recursive Approach

Code:



```

110
111     # TASK 5: Recursive Approach
112     print("\nTASK 5: Recursive Prime Check")
113     def is_prime_recursive(n, divisor=2):
114         if n < 2:
115             return False
116         if divisor * divisor > n:
117             return True
118         if n % divisor == 0:
119             return False
120         return is_prime_recursive(n, divisor + 1)
121
122     print(f"23 is prime (recursive): {is_prime_recursive(23)}")
123     print(f"24 is prime (recursive): {is_prime_recursive(24)}")
124
125
126
127
128
129
130
131
132
133
134
135

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

TASK 4: Performance Comparison (Procedural vs Modular)
 Procedural: True, Time: 0.000000s
 Function-based: True, Time: 0.000000s
 Results match: True

TASK 5: Recursive Prime Check
 23 is prime (recursive): True
 24 is prime (recursive): False

PROBLEM 2: FIBONACCI SERIES

Ln 138, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.3 Go Live Prettier

Perfect! Explanations updated to 3-6 lines for every task and the file still runs (comments don't affect execution). If you want, I can also format the explanations into a short summary section at the end of the file.

Output:

```

TASK 5: Recursive Prime Check
23 is prime (recursive): True
24 is prime (recursive): False

```

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.

Problem -2 : Fibonacci Series

TASK-1 :

Prompt:

TASK 1: Without Modularization

Code:

```

152
153     # TASK 1: Without Modularization
154     print("\nTASK 1: Without Modularization")
155     n = 8
156     a, b = 0, 1
157     fib_series = [a, b]
158     for _ in range(n - 2):
159         a, b = b, a + b
160         fib_series.append(b)
161     print(f"Fibonacci series (first {n} terms): {fib_series}")
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\2403A51L03\3-AI_A_C & C:/Users/1srch/AppData/Local/Programs/Python/Python312/python.exe C:/2403A51L03/3-AI_A_C/Lab2.py

TASK 4: Performance Comparison (Procedural vs Modular)

Procedural: True, Time: 0.000000

Function-based: True, Time: 0.000000

Results match: True

TASK 5: Recursive Prime Check

23 is prime (recursive): True

24 is prime (recursive): False

=====

PROBLEM 2: FIBONACCI SERIES

0 0 0

File Edit Selection View Go Run ... ← → 🔍 AI_A_C Chat + 🌐 ... | X

← UPDATING TASK 4 TO COMPARE CODE IM... script once to verify the comments align with actual output.

replace description with explanation of the code in 3 to 6 lines

Proceeding to replace each one-line description with a 3-6 line explanatory comment for every task; then I'll run the script to confirm everything still runs.

✓ Edited Lab2.py +22 -10

o & C:/Users/1srch/AppData/Local/Programs/Python/Python312/python.exe c:/2403A51L03/3-AI_A_C/Lab2.py

Perfect! Explanations updated to 3-6 lines for every task and the file still runs (comments don't affect execution). If you want, I can also format the explanations into a short summary section at the end of the file.

G ⌂ ⌂ ⌂ ⌂

Lab2.py +

Describe what to build next

Agent Auto ⌂ 10:40 09-01-2026

Output:

TASK 1: Without Modularization

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

Explanation:

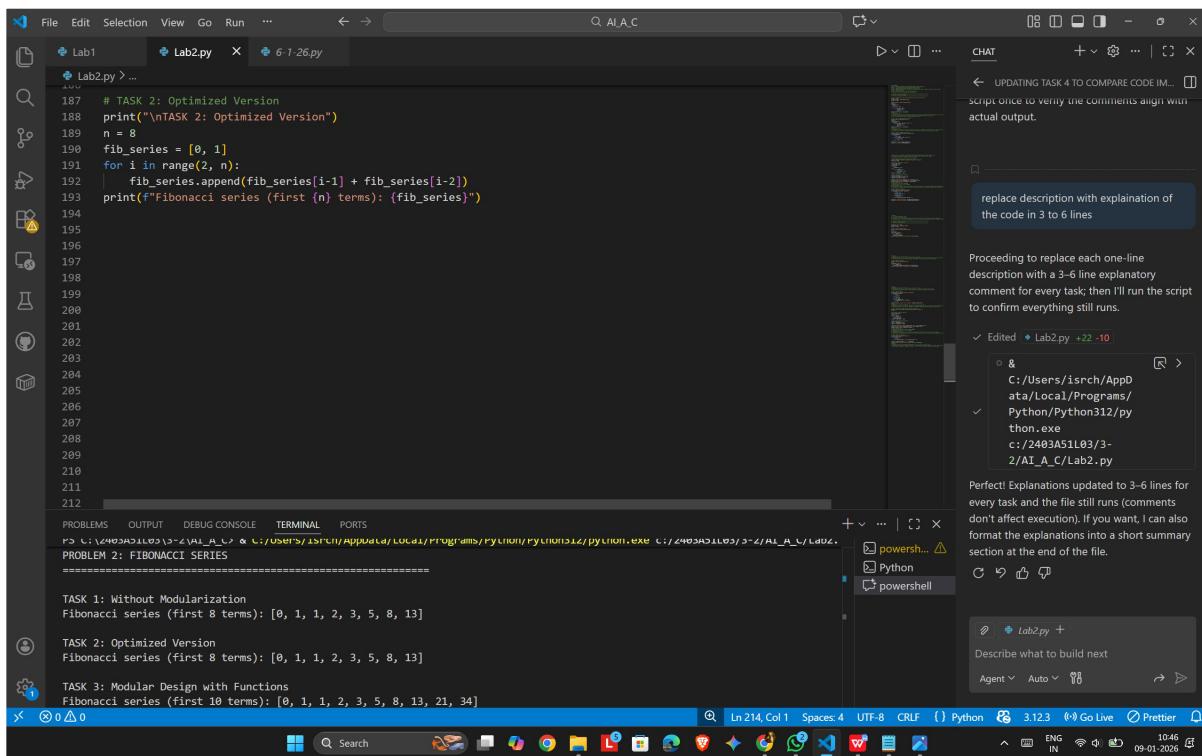
Starts with [0,1] and iteratively appends the sum of the last two elements for n-2 iterations. This produces the first n Fibonacci numbers efficiently using a simple loop and tuple updates.

TASK-2 :

Prompt:

TASK 2: Optimized Version

Code:



The screenshot shows the Visual Studio Code interface. The left pane displays the code for `Lab2.py`, which contains three tasks for generating Fibonacci series. The right pane shows the terminal output for each task. A Chat window is open at the top right, providing instructions for updating the code. The status bar at the bottom indicates the file is saved with 22 changes.

```

187 # TASK 2: Optimized Version
188 print("\nTASK 2: Optimized Version")
189 n = 8
190 fib_series = [0, 1]
191 for i in range(2, n):
192     fib_series.append(fib_series[i-1] + fib_series[i-2])
193 print("Fibonacci series (first {} terms): {}".format(n, fib_series))
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212

```

TERMINAL OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\isrch\AppData\Local\Programs\Python\Python312\python.exe C:/2403A51L03/3-2/AI_A_C/Lab2.py
=====
PROBLEM 2: FIBONACCI SERIES

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

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

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

```

Output:

```

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

```

Explanation:

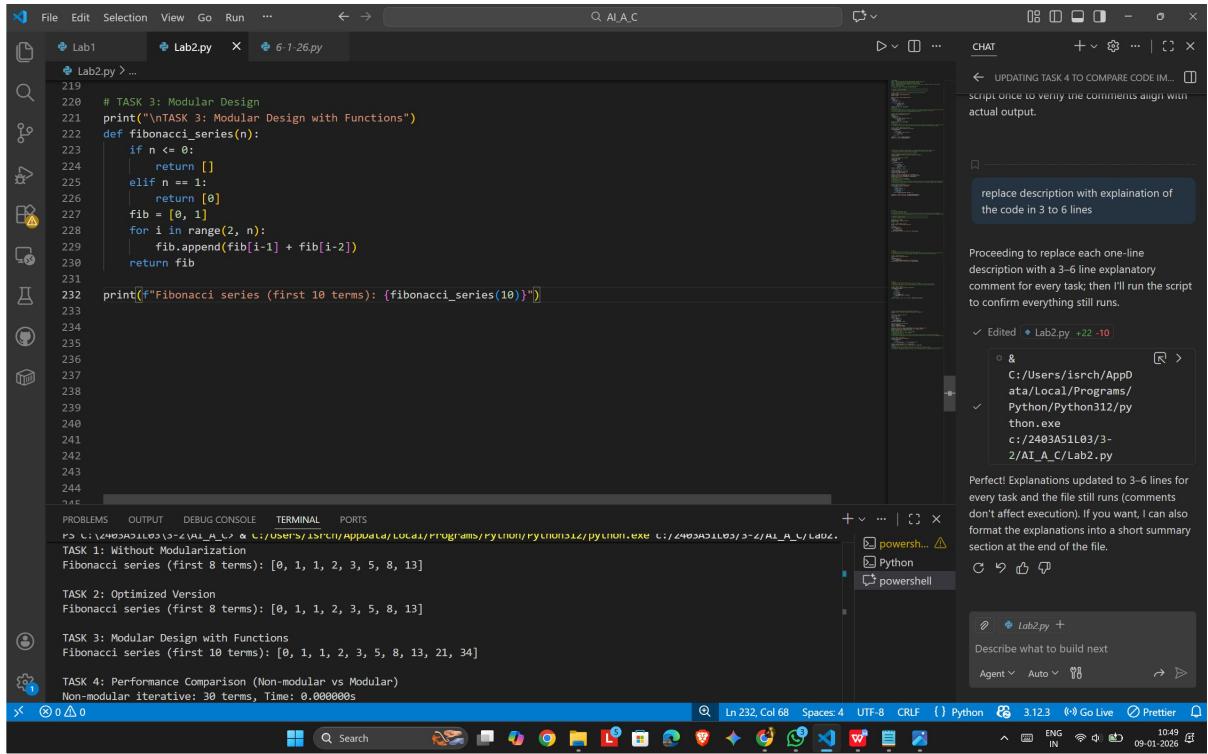
Uses explicit list indexing (`fib[i-1] + fib[i-2]`) to compute each next term.
Functionally equivalent to Task 1 but the indexing style may be easier to read and extend.

TASK - 3:

Prompt:

TASK 3: Modular Design

Code:



```

219
220     # TASK 3: Modular Design
221     print("\nTASK 3: Modular Design with Functions")
222     def fibonacci_series(n):
223         if n <= 0:
224             return []
225         elif n == 1:
226             return [0]
227         fib = [0, 1]
228         for i in range(2, n):
229             fib.append(fib[i-1] + fib[i-2])
230         return fib
231
232     print("Fibonacci series (first 10 terms):", fibonacci_series(10))
233
234
235
236
237
238
239
240
241
242
243
244

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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

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

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

Task 4: Performance Comparison (Non-modular vs Modular)
Non-modular iterative: 30 terms, Time: 0.000000s

CHAT

UPDATING TASK 4 TO COMPARE CODE IM... Script once to verify the comments align with actual output.

replace description with explanation of the code in 3 to 6 lines

Proceeding to replace each one-line description with a 3-6 line explanatory comment for every task; then I'll run the script to confirm everything still runs.

Edited * Lab2.py +22 -10

C:/Users/isch/AppData/Local/Programs/Python/Python312/python.exe c:/2403AS1103/3-2/AI_A_C/Lab2.py

Perfect! Explanations updated to 3-6 lines for every task and the file still runs (comments don't affect execution). If you want, I can also format the explanations into a short summary section at the end of the file.

Describe what to build next

Ln 232, Col 68 Spaces: 4 UTF-8 CRLF Python 3.12.3 Go Live Prettier

10:49 09-01-2026

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.

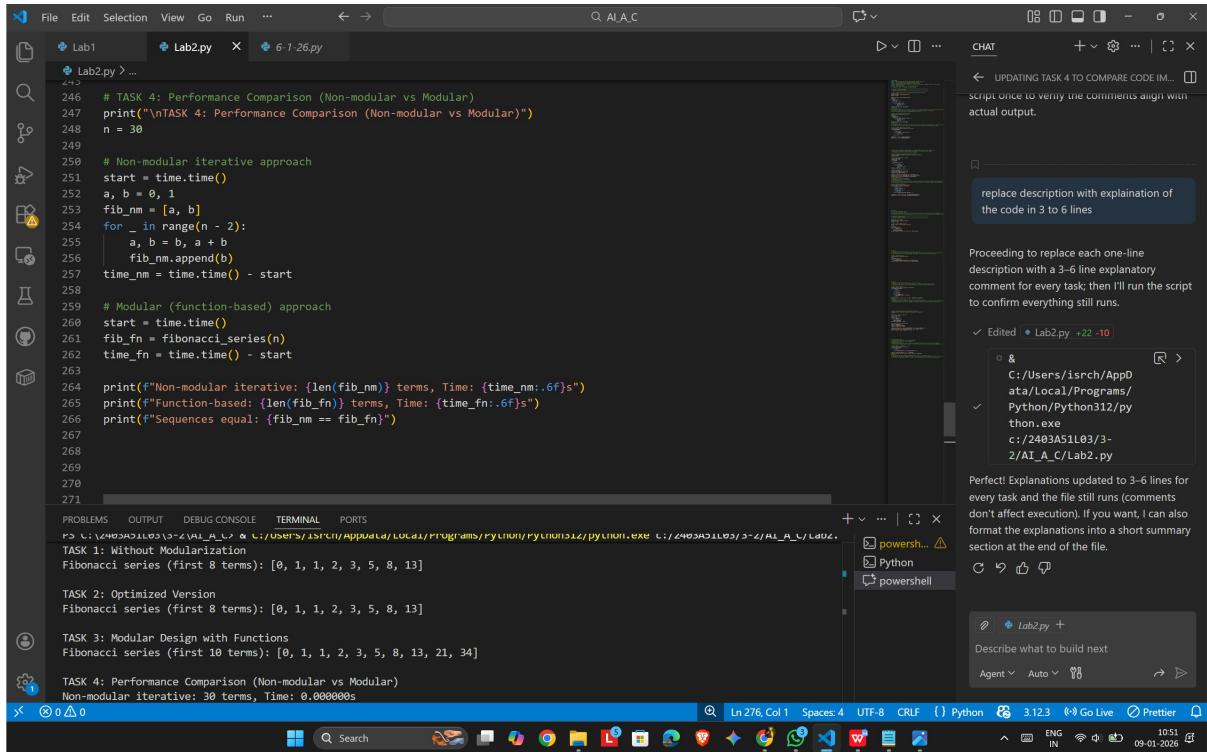
Modularizing makes it easy to get sequences of different lengths and improves code clarity and reuse.

TASK-4:

Prompt:

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

Code:



```

246 # TASK 4: Performance Comparison (Non-modular vs Modular)
247 print("\nTASK 4: Performance Comparison (Non-modular vs Modular)")
248 n = 30
249
250 # Non-modular iterative approach
251 start = time.time()
252 a, b = 0, 1
253 fib_nm = [a, b]
254 for _ in range(n - 2):
255     a, b = b, a + b
256     fib_nm.append(b)
257 time_nm = time.time() - start
258
259 # Modular (function-based) approach
260 start = time.time()
261 fib_fn = fibonacci_series(n)
262 time_fn = time.time() - start
263
264 print(f"Non-modular iterative: {len(fib_nm)} terms, Time: {time_nm:.6f}s")
265 print(f"Function-based: {len(fib_fn)} terms, Time: {time_fn:.6f}s")
266 print(f"Sequences equal: {fib_nm == fib_fn}")
267
268
269
270
271

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

C:\Users\isrch\AppData\Local\Programs\Python\Python312\python.exe C:/2403AS1103/3-2/AI_A_C/Lab2.py

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

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

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

TASK 4: Performance Comparison (Non-modular vs Modular)
Non-modular iterative: 30 terms, Time: 0.000000s

CHAT

replace description with explanation of the code in 3 to 6 lines

Proceeding to replace each one-line description with a 3-6 line explanatory comment for every task; then I'll run the script to confirm everything still runs.

Edited * Lab2.py +22 -10

C:/Users/isrch/AppData/Local/Programs/Python/Python312/python.exe c:/2403AS1103/3-2/AI_A_C/Lab2.py

Perfect! Explanations updated to 3-6 lines for every task and the file still runs (comments don't affect execution). If you want, I can also format the explanations into a short summary section at the end of the file.

Lab2.py +

Describe what to build next

Agent Auto 09-01-2026 10:51

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.

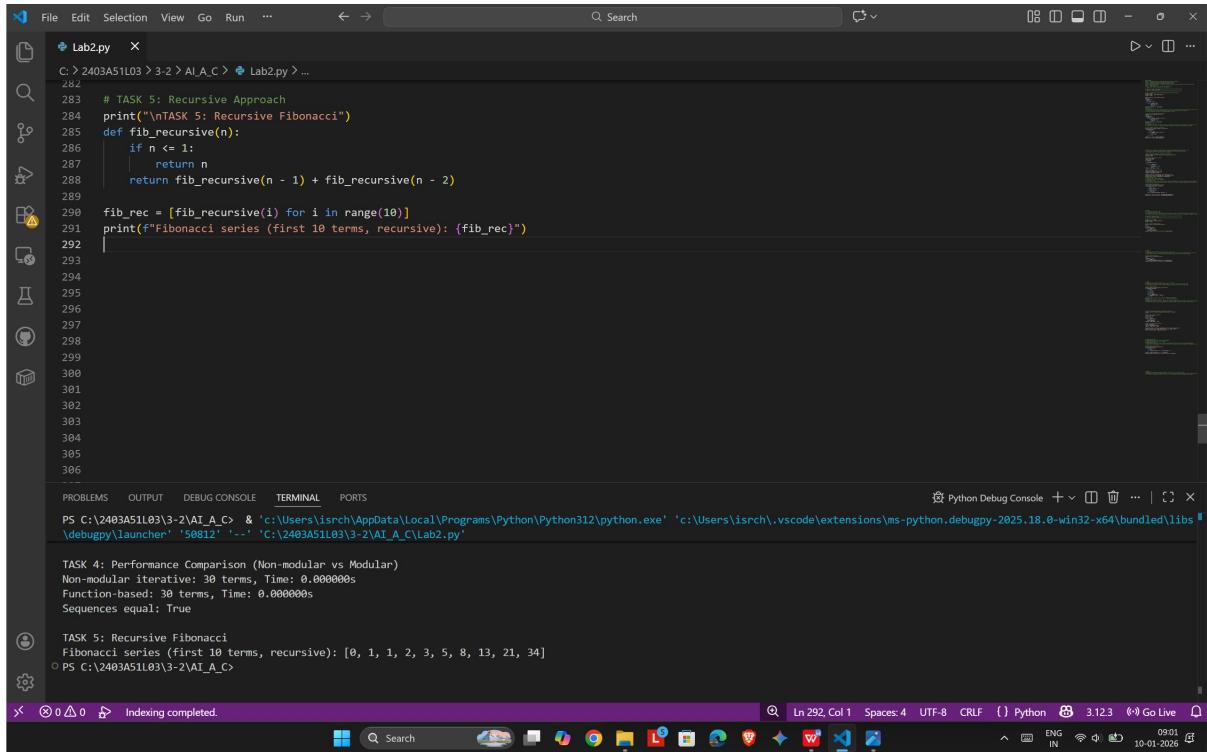
Timing shows both approaches are similar for n=30; the function version is preferable for readability and reuse.

TASK -5:

Prompt:

TASK 5: Recursive Approach

Code:



The screenshot shows the VS Code interface with the following details:

- Code Editor:** The file `Lab2.py` is open, containing Python code for generating a Fibonacci series using recursion.
- Terminal:** The terminal window shows the execution of the script and its output. It includes performance comparison results for Task 4 and Task 5, and the resulting Fibonacci series.
- Status Bar:** Shows indexing status, file path (`C:\2403A51L03\3-2\AI_A_C>`), and system information (Windows taskbar).

```

283     # TASK 5: Recursive Approach
284     print("\nTASK 5: Recursive Fibonacci")
285     def fib_recursive(n):
286         if n <= 1:
287             return n
288         return fib_recursive(n - 1) + fib_recursive(n - 2)
289
290     fib_rec = [fib_recursive(i) for i in range(10)]
291     print(f"Fibonacci series (first 10 terms, recursive): {fib_rec}")
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\2403A51L03\3-2\AI_A_C> & 'c:\Users\isrch\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\isrch\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '50812' --- 'C:\2403A51L03\3-2\AI_A_C\Lab2.py'

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

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>

Indexing completed.

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.