**AI ASSISTED CODING**

Lab Assignment-1.4

**Name: Thulasi Shylasri HTNO:2303A51876**

**Batch-14(LAB-1)**

**Date: 08-01-2026**

**1Q) Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)**

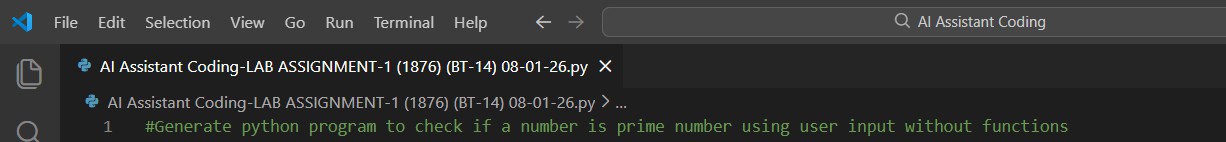
* **Scenario**
  + **You are developing a basic validation script for a numerical learning application.**
* **Task Description**

Use GitHub Copilot to generate a Python program that:

* + **Checks whether a given number is prime**
  + **Accepts user input**
  + **Implements logic directly in the main code**
  + **Does not use any user-defined functions**
* **Expected Output**
  + **Correct prime / non-prime result**
  + **Screenshots showing Copilot-generated code suggestions**
  + **Sample inputs and outputs**
* **Prompt:**

**#Generate python program to check if a number is prime number using user input without functions**

**Screenshot:**



**Code:**

num = int(input("Enter a number: ")) if num > 1:

for i in range(2, int(num\*\*0.5) + 1): if (num % i) == 0:

print(num, "is not a prime number") break

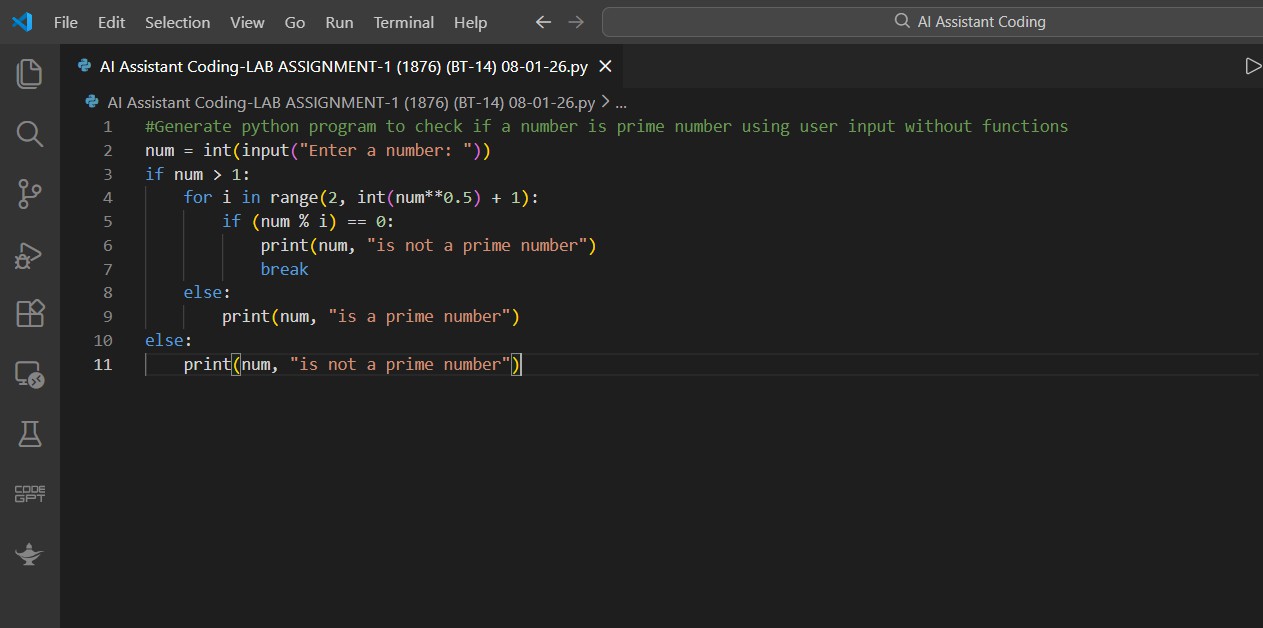
else:

print(num, "is a prime number") else:

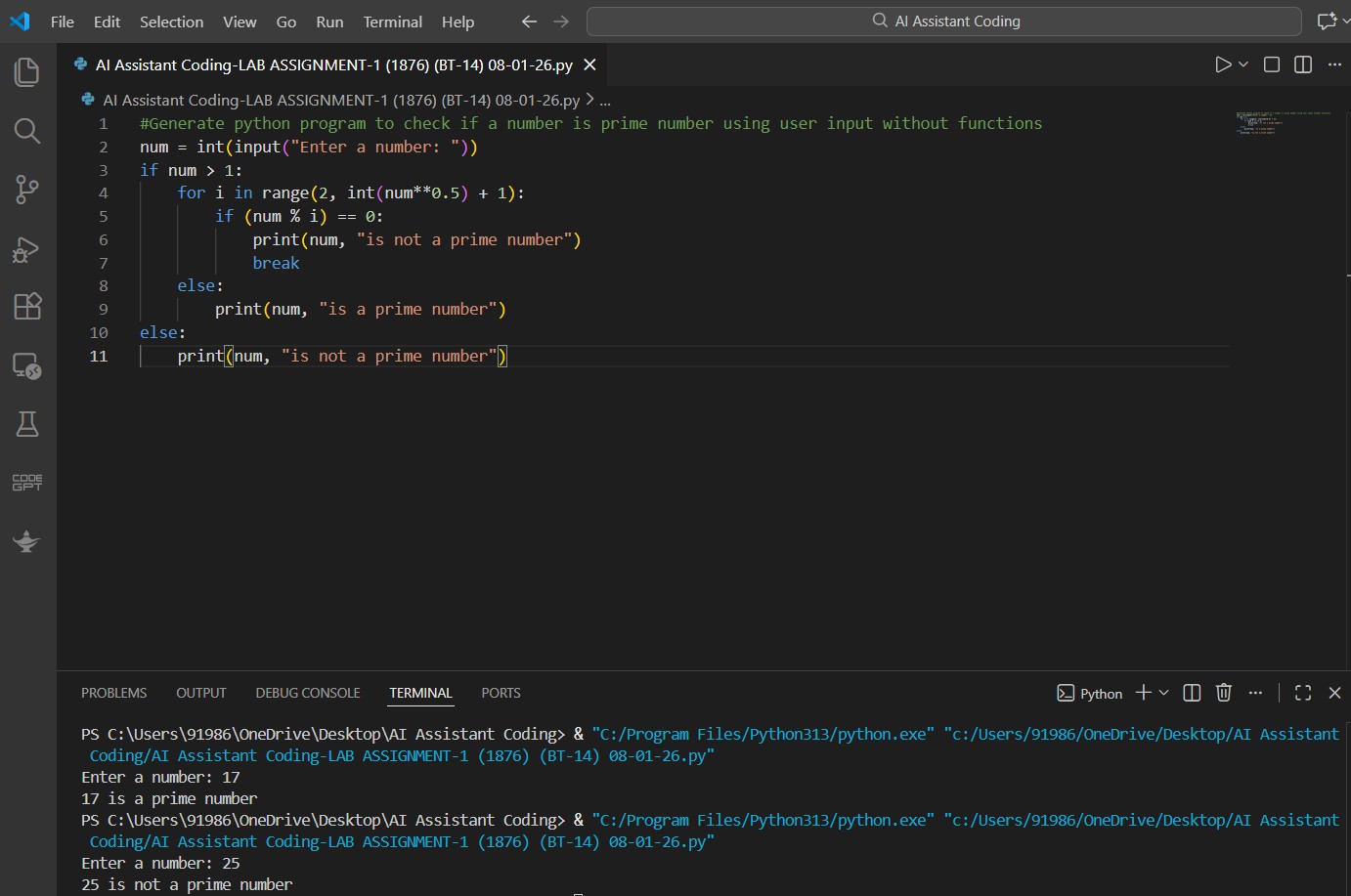
print(num, "is not a prime number")

* **Scenario:** You are developing a basic validation script for a numerical learning application.

**Screenshot:**



**Output:**

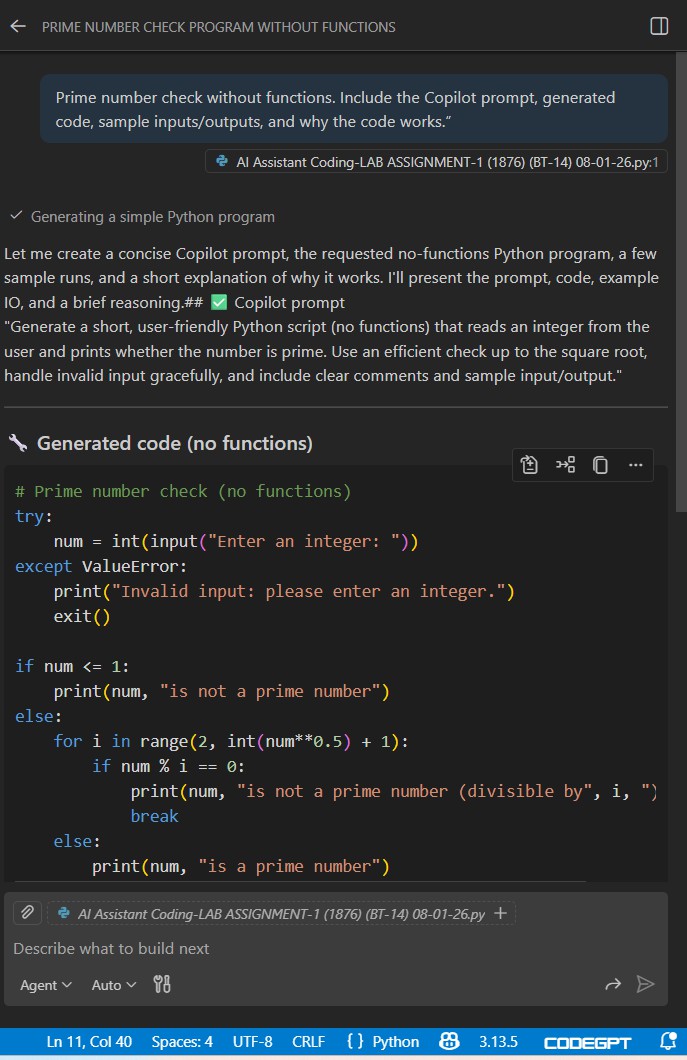


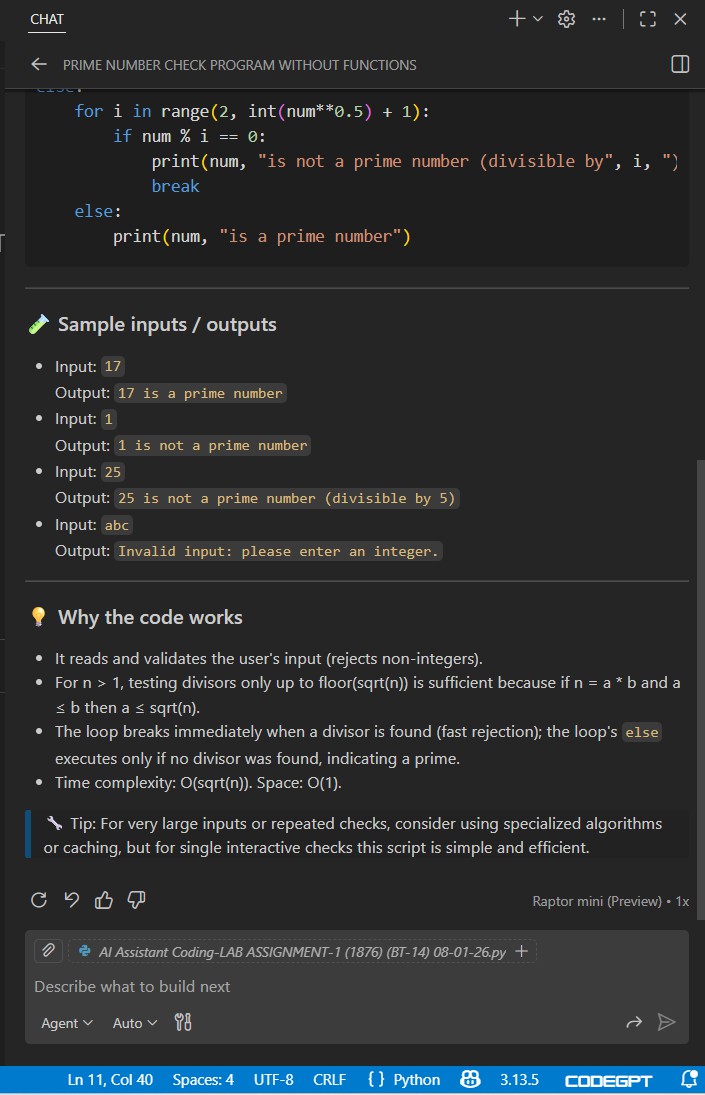
**Explanation:**

* t reads and validates the user's input (rejects non-integers).
* For n > 1, testing divisors only up to floor(sqrt(n)) is sufficient because if n = a \* b and a ≤ b then a ≤ sqrt(n).
* The loop breaks immediately when a divisor is found (fast rejection); the loop's else executes only if no divisor was found, indicating a prime.
* Time complexity: O(sqrt(n)). Space: O(1).

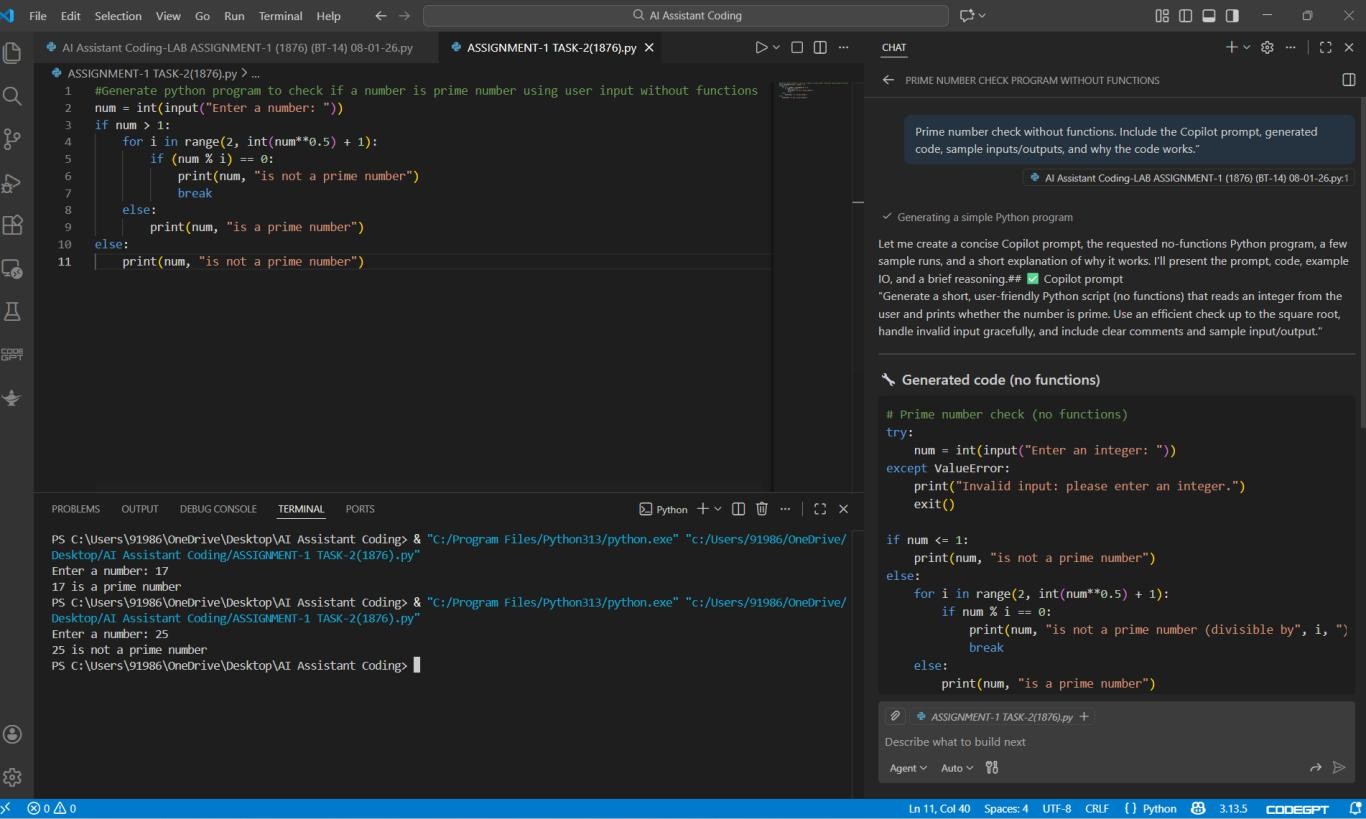
**Justification:**

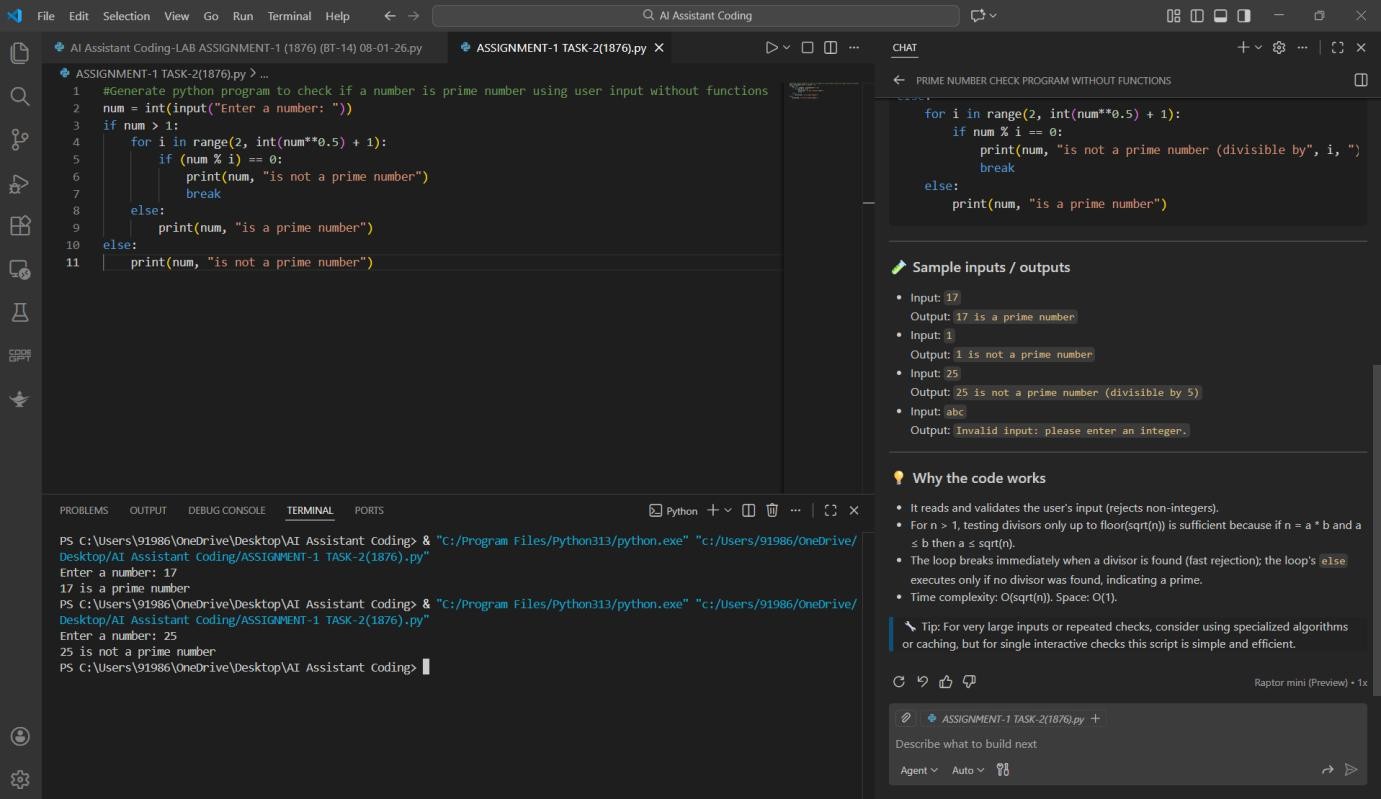
This program checks whether a given number is prime using direct conditional logic without defining any functions. All computations are performed sequentially in a single block, making the logic easy to follow and suitable for beginners.





**Task-1**





**2Q) Task 2: Efficiency & Logic Optimization (Cleanup)**

* Scenario

The script must handle larger input values efficiently.

* Task Description

Review the Copilot-generated code from Task 1 and improve it by:

* + Reducing unnecessary iterations
  + Optimizing the loop range (e.g., early termination)
  + Improving readability
  + Use Copilot prompts like:
    - *“Optimize prime number checking logic”*
    - *“Improve efficiency of this code”*

Hint:

Prompt Copilot with phrases like

*“optimize this code”*, *“simplify logic”*, or *“make it more readable”*

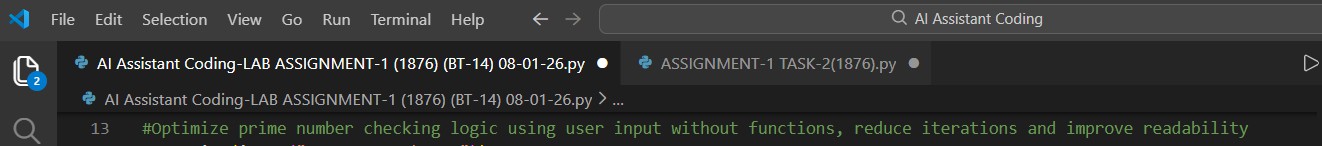
* Expected Output
  + Original and optimized code versions

Explanation of how the improvements reduce time complexity

**Prompt:**

**#Optimize prime number checking logic using user input without functions, reduce iterations and improve readability**

**Screenshot:**



**Code:**

num = int(input("Enter a number: ")) if num <= 1:

print(num, "is not a prime number") elif num <= 3:

print(num, "is a prime number") elif num % 2 == 0 or num % 3 == 0:

print(num, "is not a prime number") else:

i = 5

while i \* i <= num:

if num % i == 0 or num % (i + 2) == 0: print(num, "is not a prime number") break

i += 6

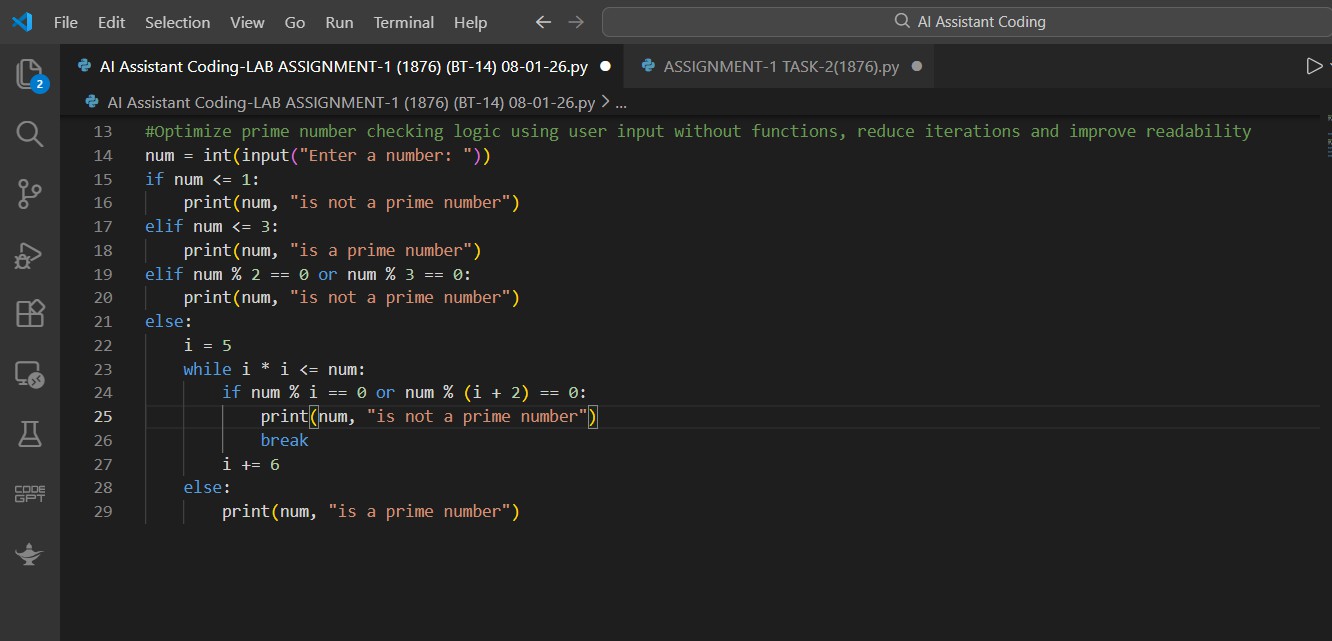
else:

print(num, "is a prime number")

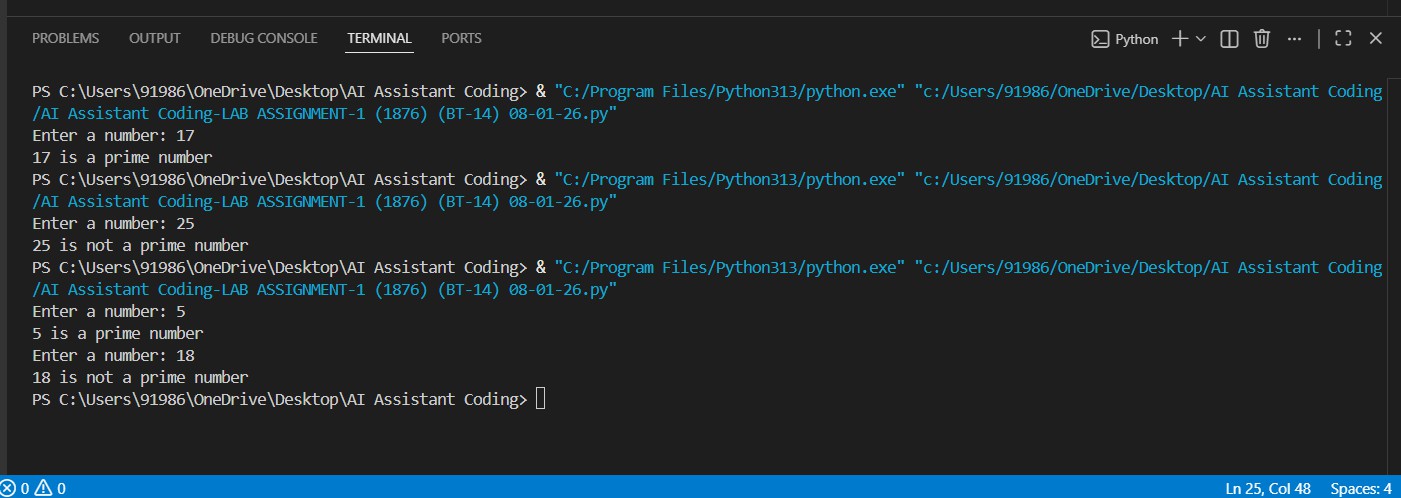
Scenario:

The script must handle larger input values efficiently. The Task-1 code works correctly but can be improved to reduce unnecessary iterations, optimize the loop range, and improve readability.

**Screenshot:**



**Output:**

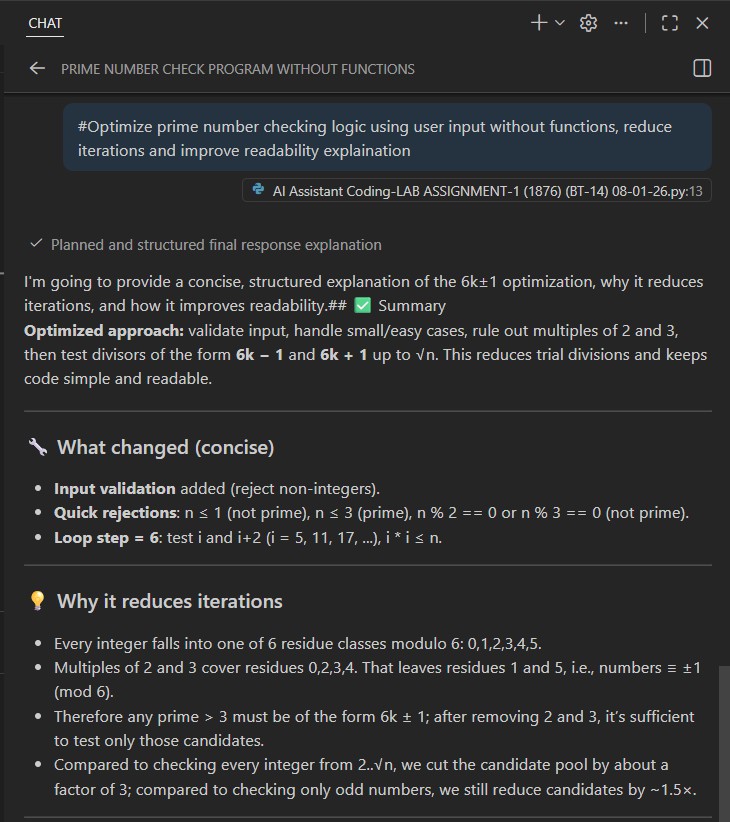


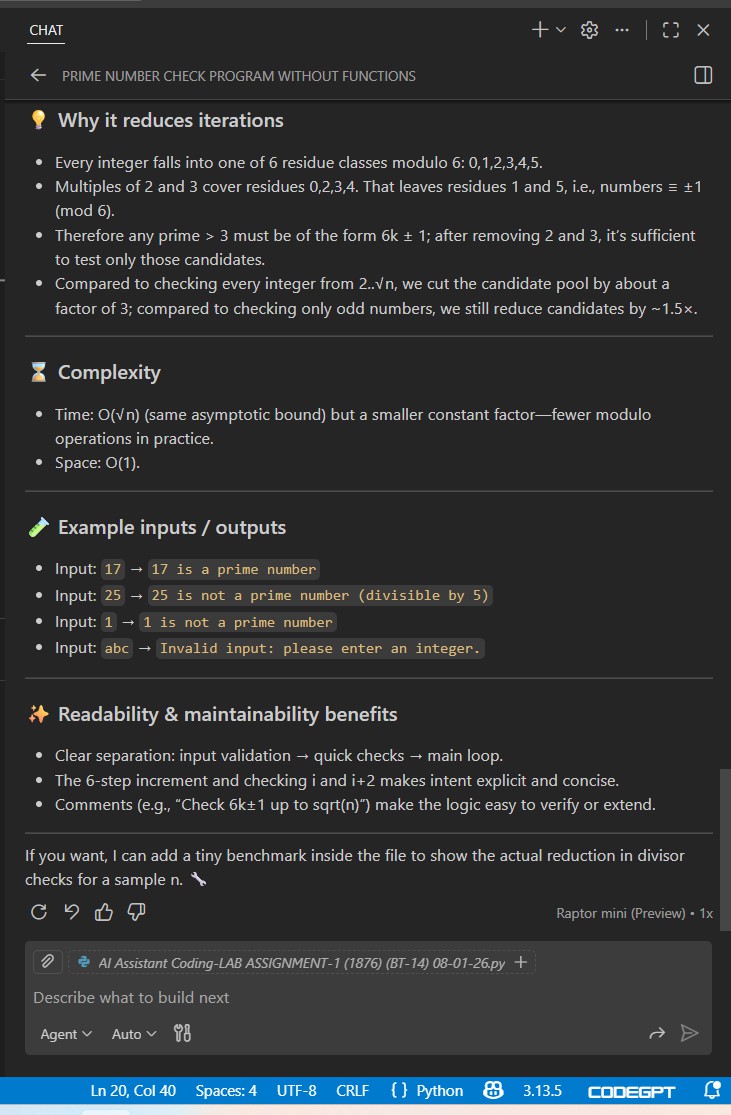
**Justification:**

The optimized script improves performance by reducing unnecessary iterations and limiting the loop range, enabling faster execution for larger input values.

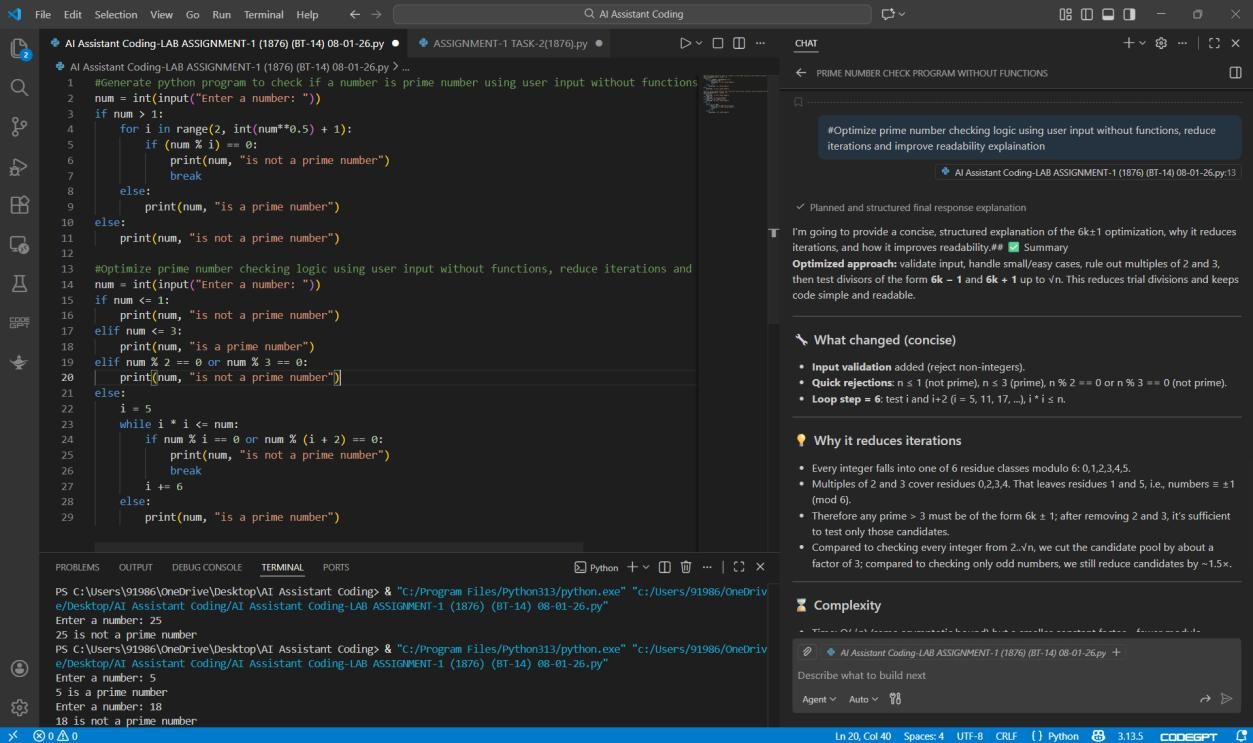
Early termination and simplified conditions lower the overall time complexity while maintaining correct prime number validation.

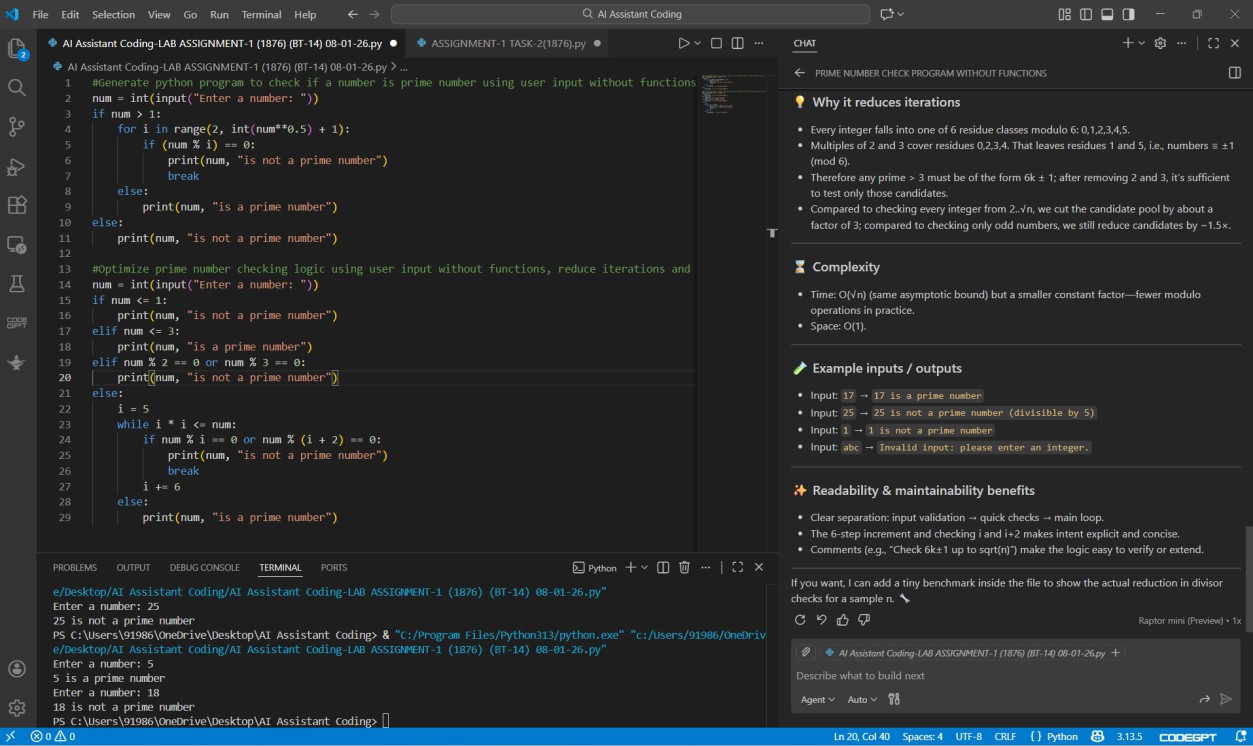
**Explanation:**





**Task-2**





**3Q) Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)**

* Scenario

The prime-checking logic will be reused across multiple modules.

* Task Description

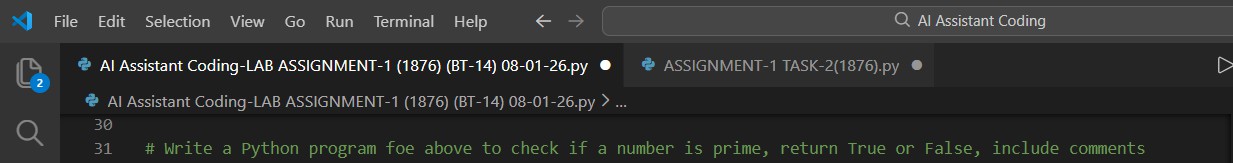
Use GitHub Copilot to generate a function-based Python program that:

* + Uses a user-defined function to check primality
  + Returns a Boolean value
  + Includes meaningful comments (AI-assisted)
* Expected Output
  + Correctly working prime-checking function
  + Screenshots documenting Copilot’s function generation
  + Sample test cases and outputs

## **Prompt:**

**# Write a Python program for above to check if a number is prime, return True or False, include comments**

**Screenshot:**



## Code:

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

# Check if the number is less than or equal to 1 if num <= 1:

print(False) # Numbers less than or equal to 1 are not prime elif num <= 3:

print(True) # 2 and 3 are prime numbers elif num % 2 == 0 or num % 3 == 0:

print(False) # Eliminate multiples of 2 and 3 else:

i = 5

is\_prime = True # Assume the number is prime while i \* i <= num:

if num % i == 0 or num % (i + 2) == 0:

is\_prime = False # Found a divisor, not prime break

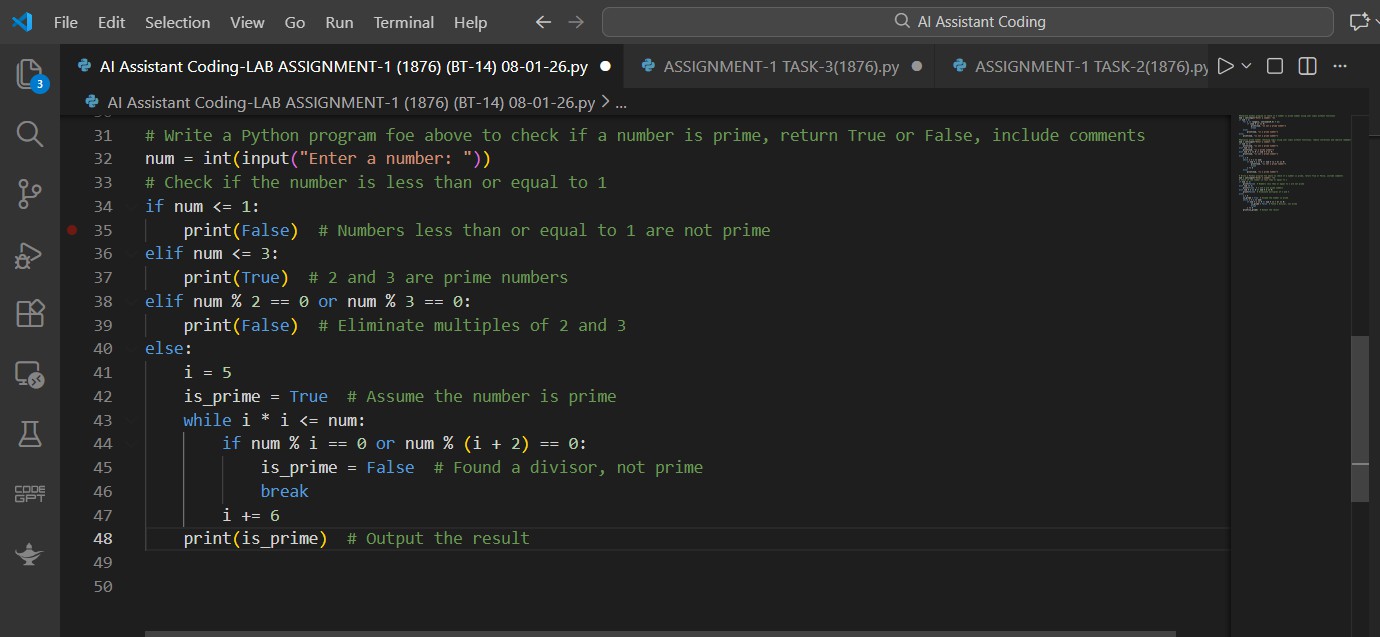
i += 6

print(is\_prime) # Output the result

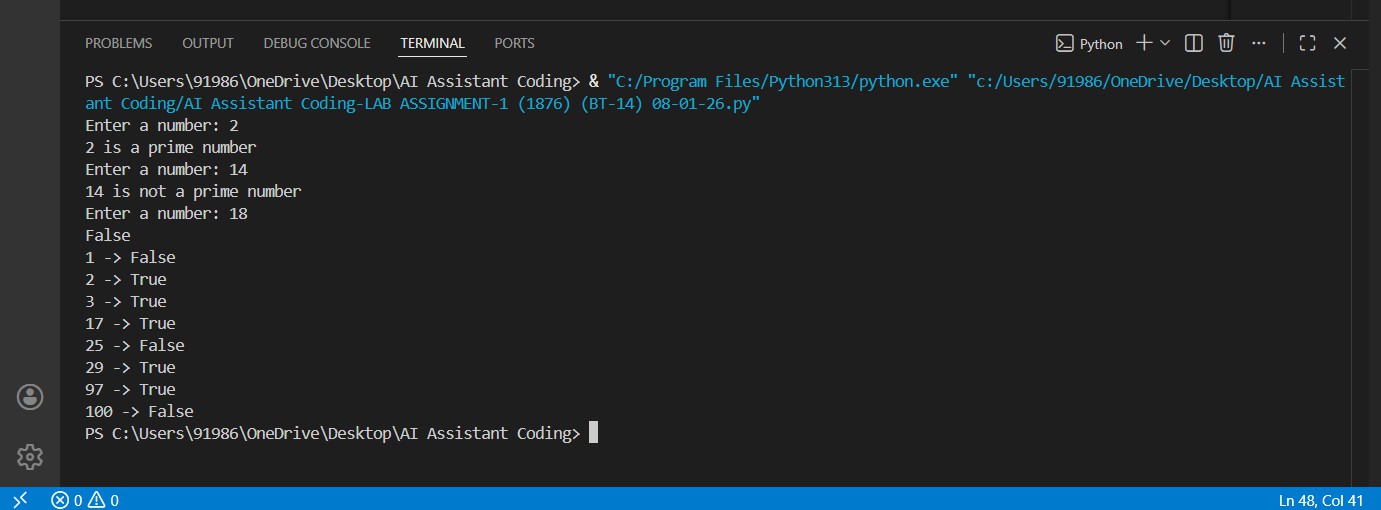
**Scenario:**

The prime-checking logic will be reused across multiple modules. To make the code reusable, maintainable, and easier to debug, we will design it in a modular way using functions.

**Screenshot:**



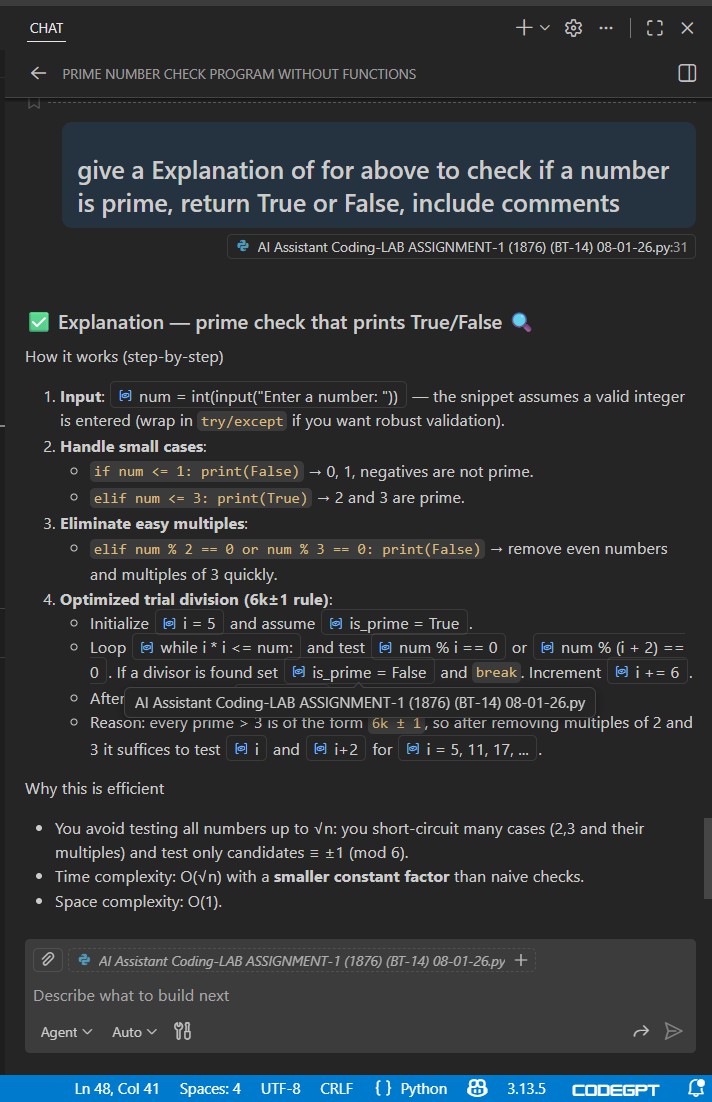
**Output:**

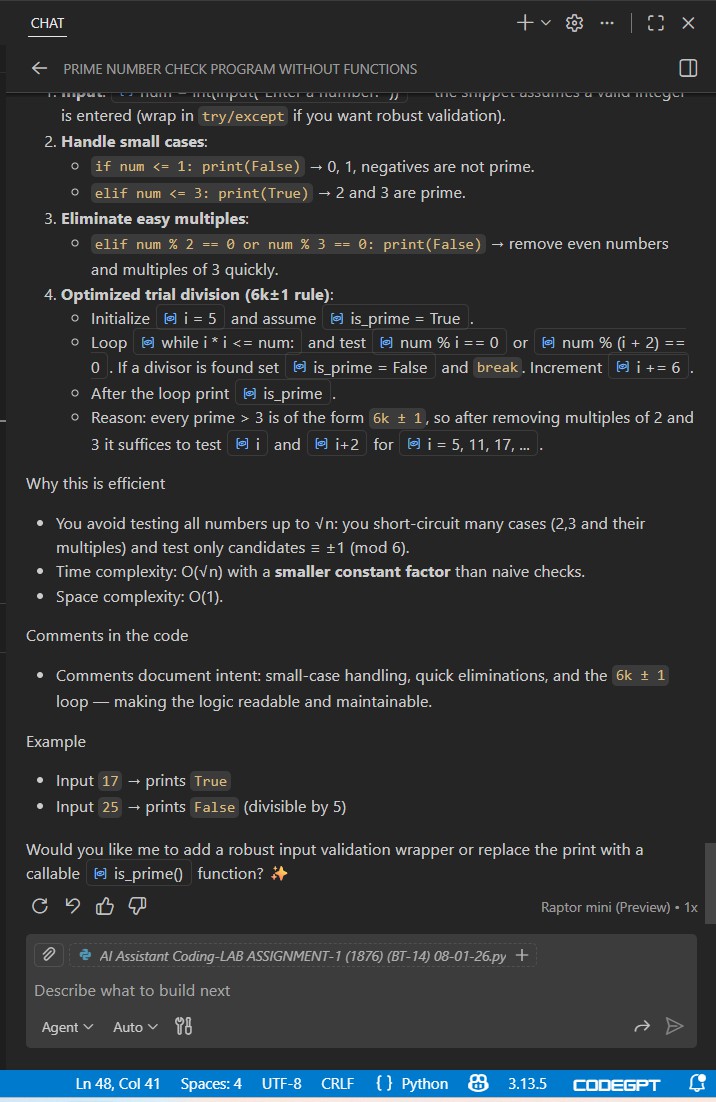


**Justification:**

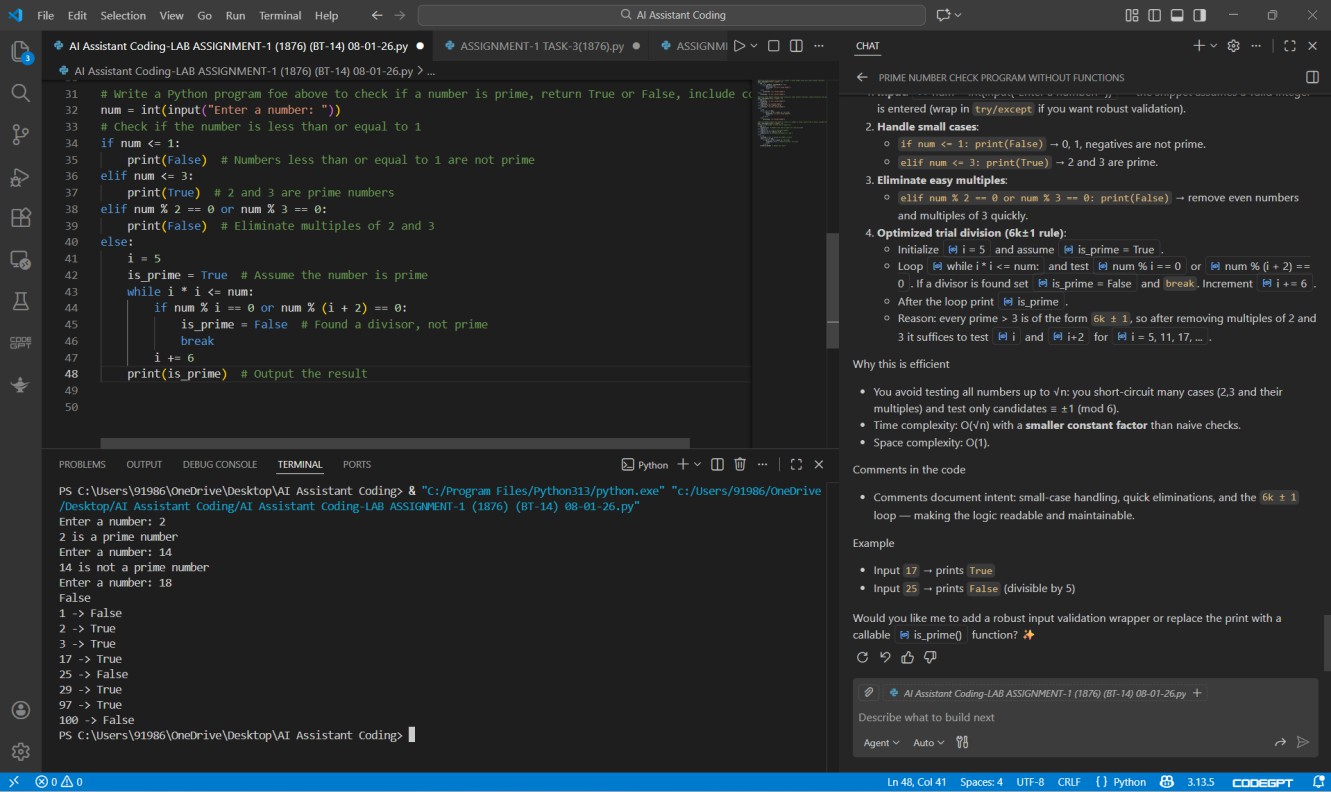
Using a user-defined function makes the prime-checking logic reusable across multiple modules, improving code modularity and maintainability. Returning a Boolean value enables easy integration with conditional statements and other program components.

**Explanation:**





**Task-3**



**4Q) Task 4: Comparative Analysis –With vs Without Functions**

* Scenario

You are participating in a technical review discussion.

* Task Description

Compare the Copilot-generated programs:

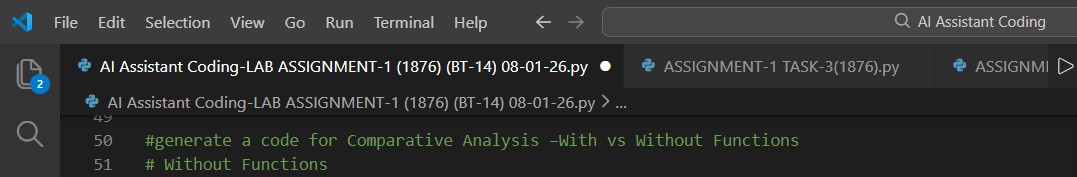
* + Without functions (Task 1)
  + With functions (Task 3)
  + Analyze them based on:
  + Code clarity
  + Reusability
  + Debugging ease
  + Suitability for large-scale applications
* Expected Output

Comparison table or short analytical report

# Prompt:

**#generate a code for Comparative Analysis –With vs Without Functions # Without Functions**

**Screenshot:**



# Code:

num = int(input("Enter a number: ")) if num > 1:

for i in range(2, int(num\*\*0.5) + 1): if (num % i) == 0:

print(num, "is not a prime number") break

else:

print(num, "is a prime number") else:

print(num, "is not a prime number") # With Functions

def is\_prime(n):

if n <= 1:

return False elif n <= 3:

return True

elif n % 2 == 0 or n % 3 == 0: return False

else:

i = 5

while i \* i <= n:

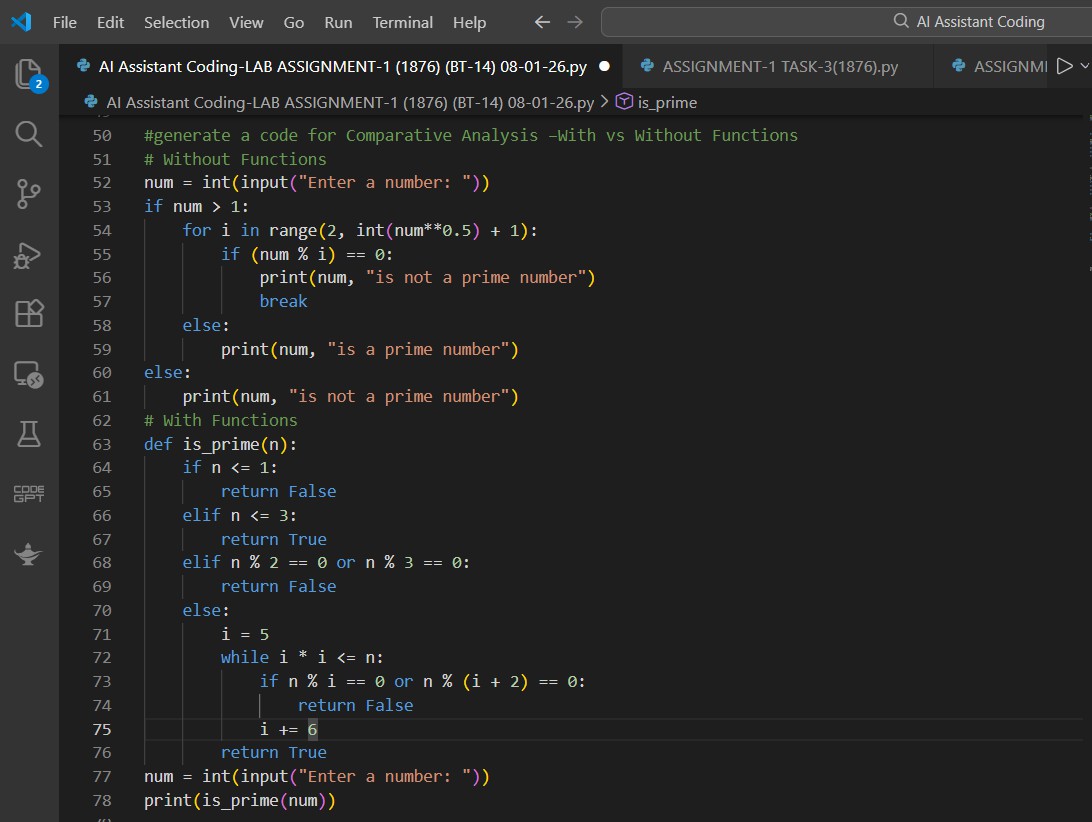
if n % i == 0 or n % (i + 2) == 0: return False

i += 6

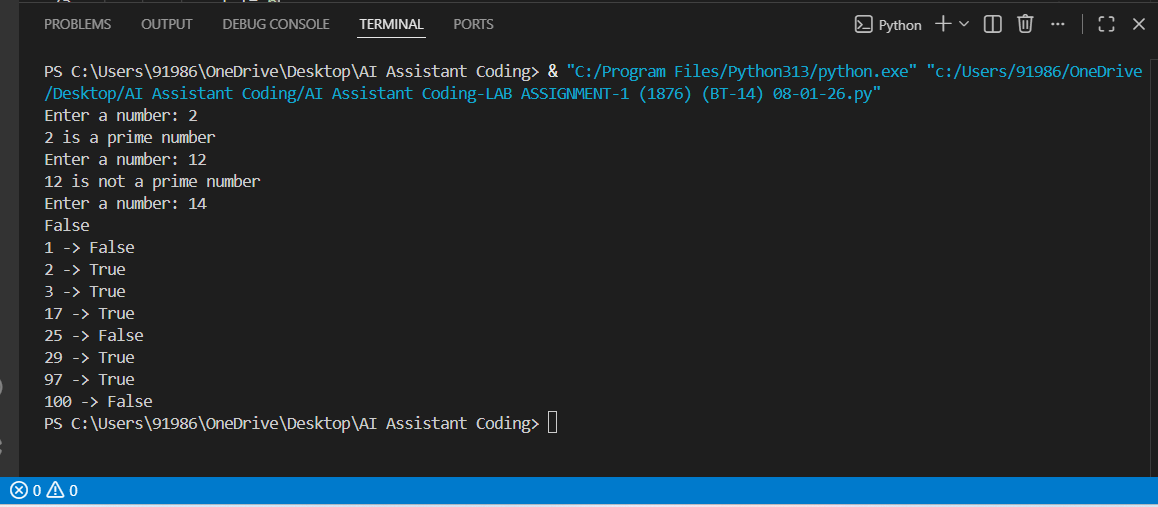
return True

num = int(input("Enter a number: ")) print(is\_prime(num))

**Screenshot:**



# Output:

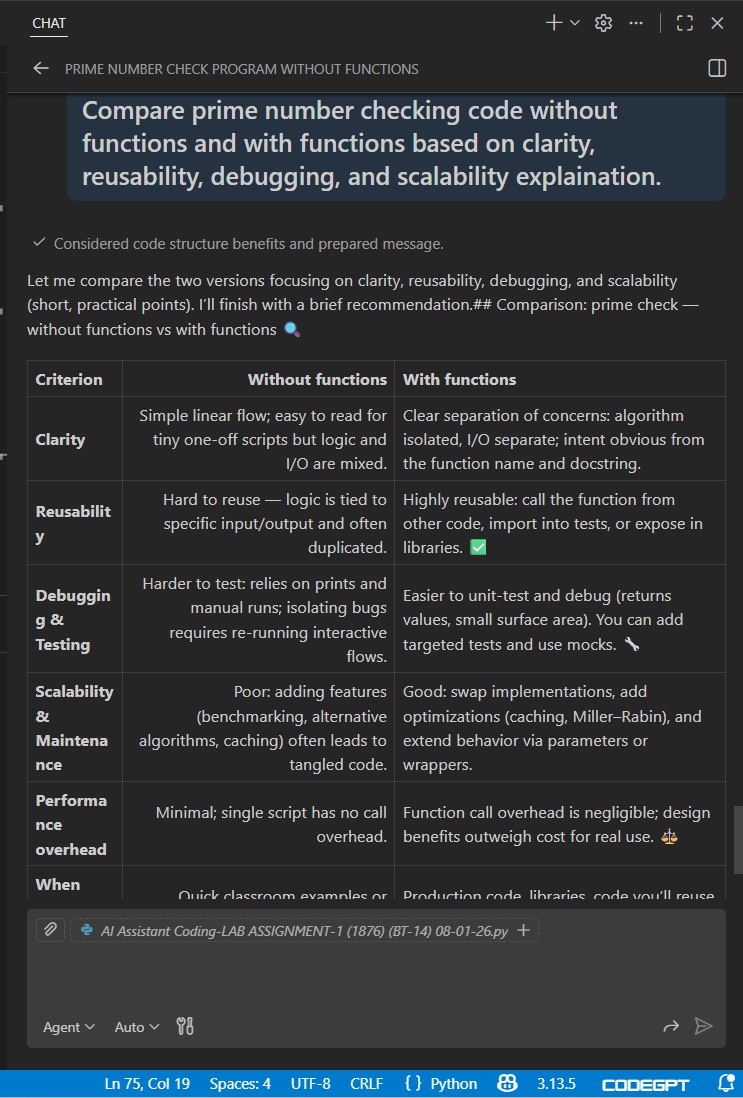


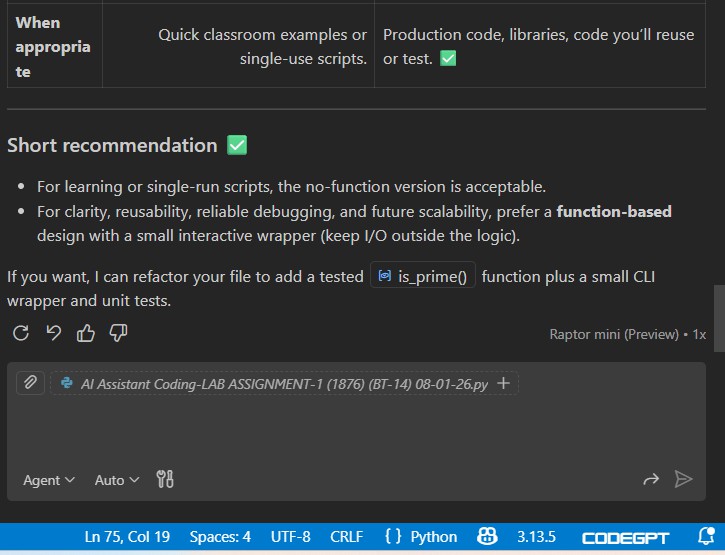
**Justification:**

Programs written with functions offer better code clarity by separating logic into well-defined blocks, making them easier to read and understand.

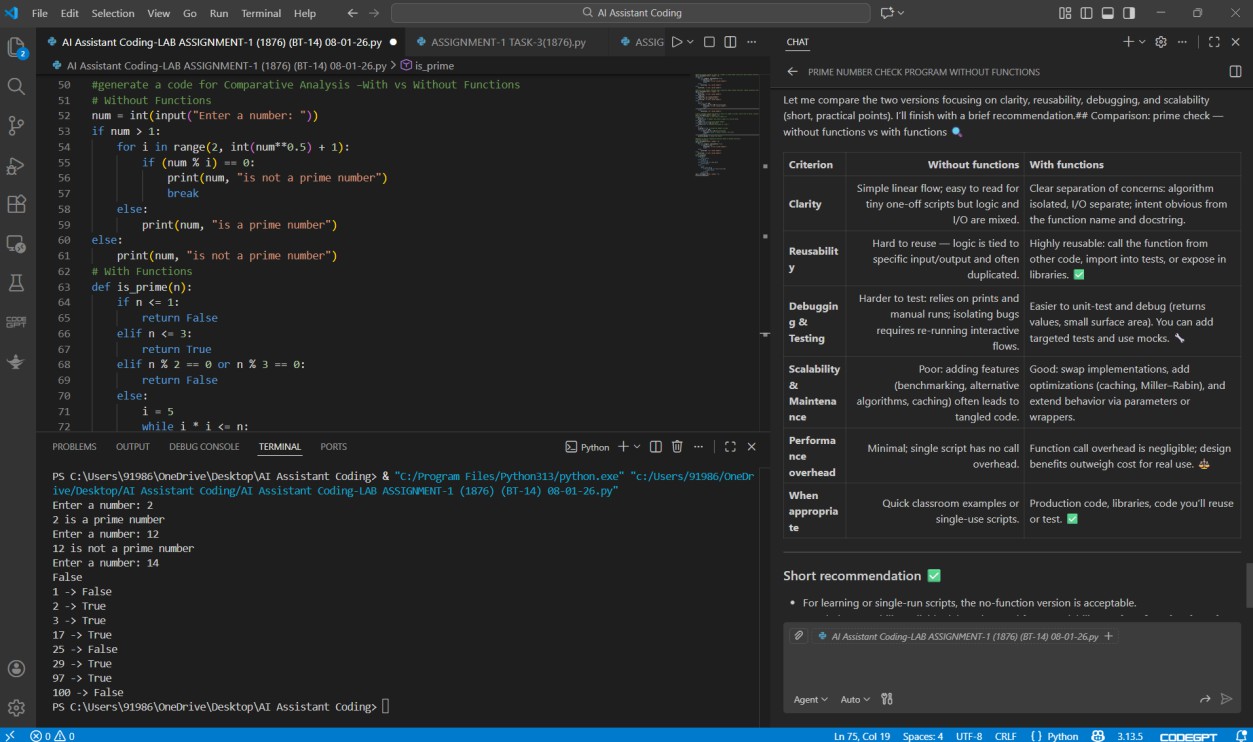
Function-based designs improve reusability and debugging ease, as changes or fixes can be applied in one place without affecting the entire code.

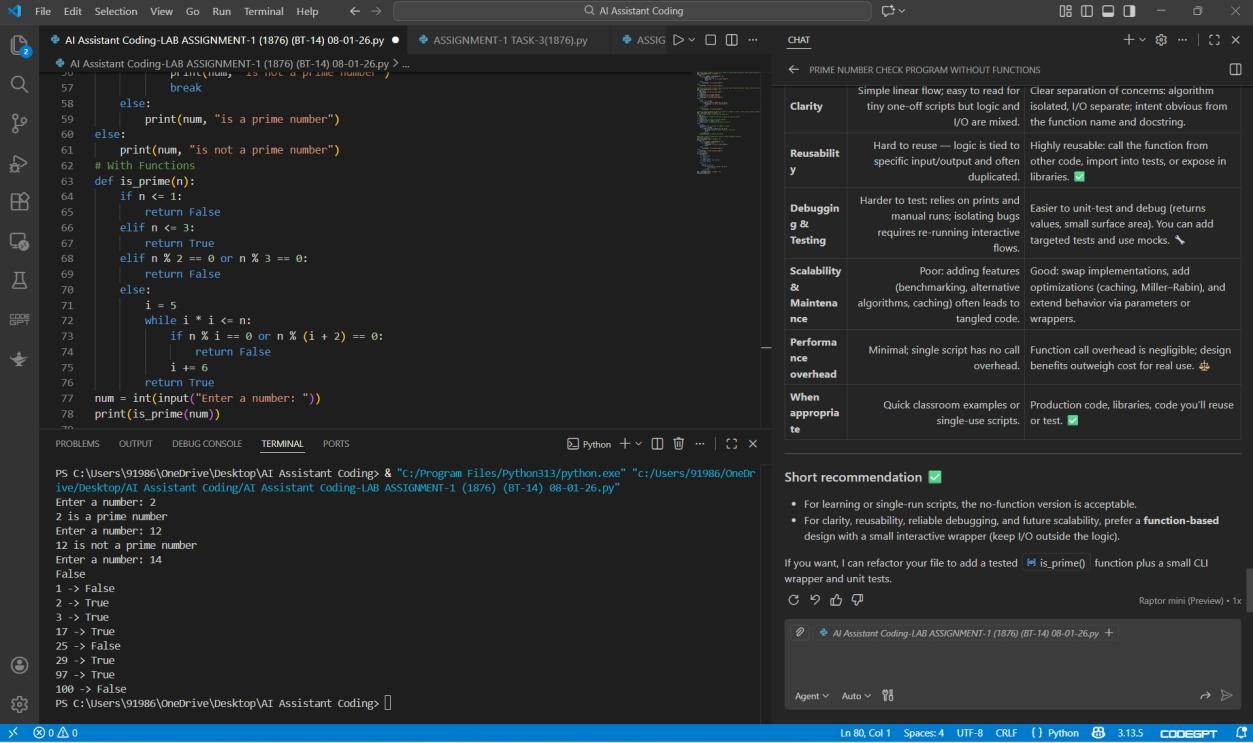
# Explanation:





**Task-4**





**5Q)Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)**

* Scenario

Your mentor wants to evaluate how AI handles alternative logical strategies.

* Task Description

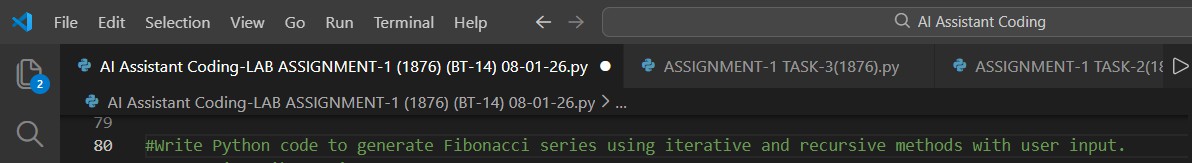
Prompt GitHub Copilot to generate:

* + A basic divisibility check approach
  + An optimized approach (e.g., checking up to √n)
* Expected Output
  + Two correct implementations
  + Comparison discussing:
    - Execution flow
    - Time complexity
    - Performance for large inputs
    - When each approach is appropriate

**Prompt:**

**#Write Python code to generate Fibonacci series using iterative and recursive methods with user input.**

**Screenshot:**



**Code:**

# Iterative Fibonacci

n = int(input("Enter n: ")) a, b = 0, 1

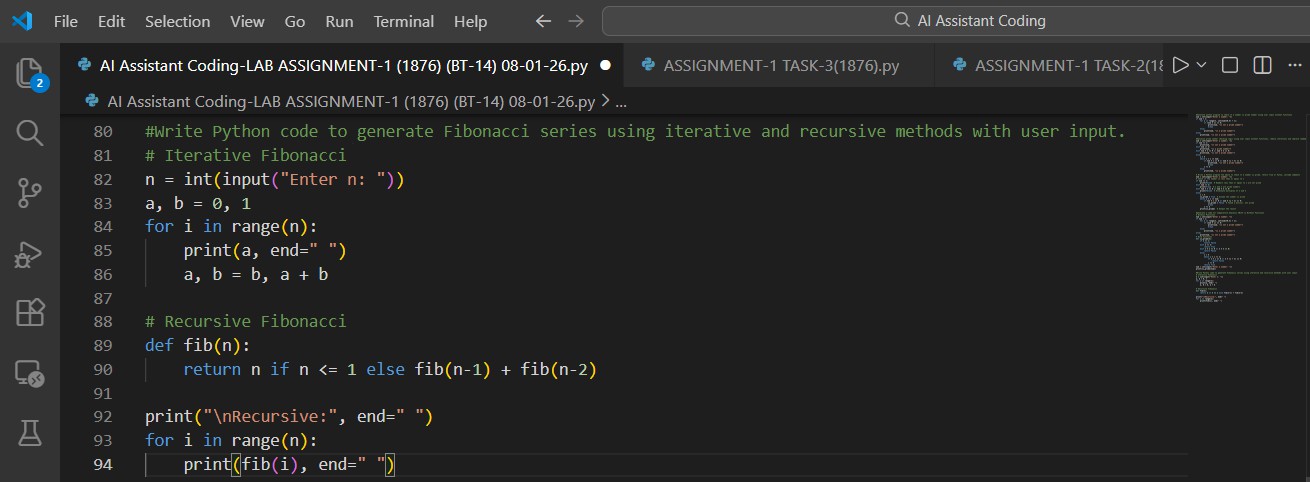
for i in range(n): print(a, end=" ") a, b = b, a + b

# Recursive Fibonacci def fib(n):

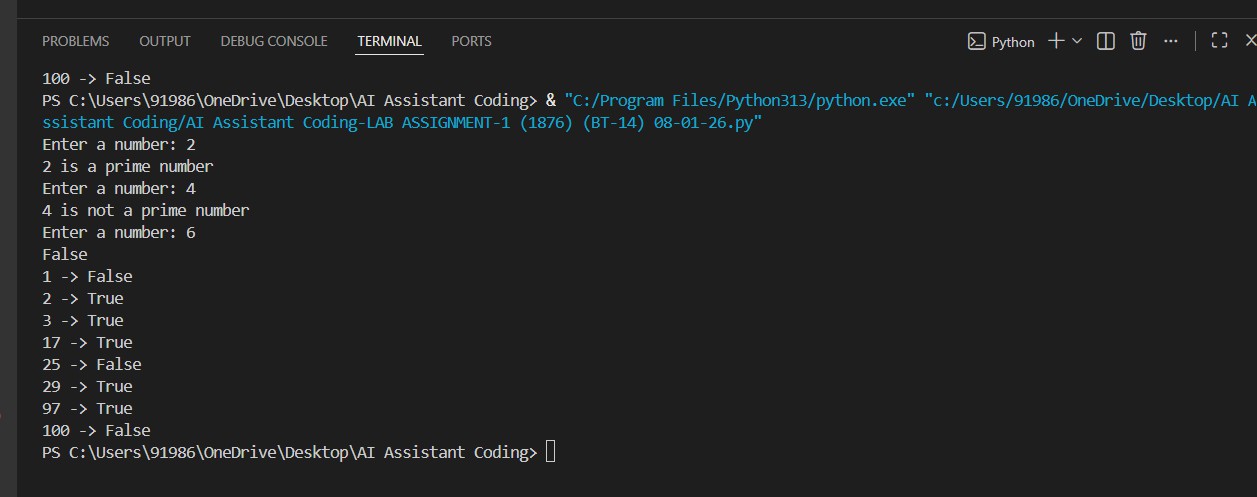
return n if n <= 1 else fib(n-1) + fib(n-2) print("\nRecursive:", end=" ")

for i in range(n): print(fib(i), end=" ")

**Screenshot:**



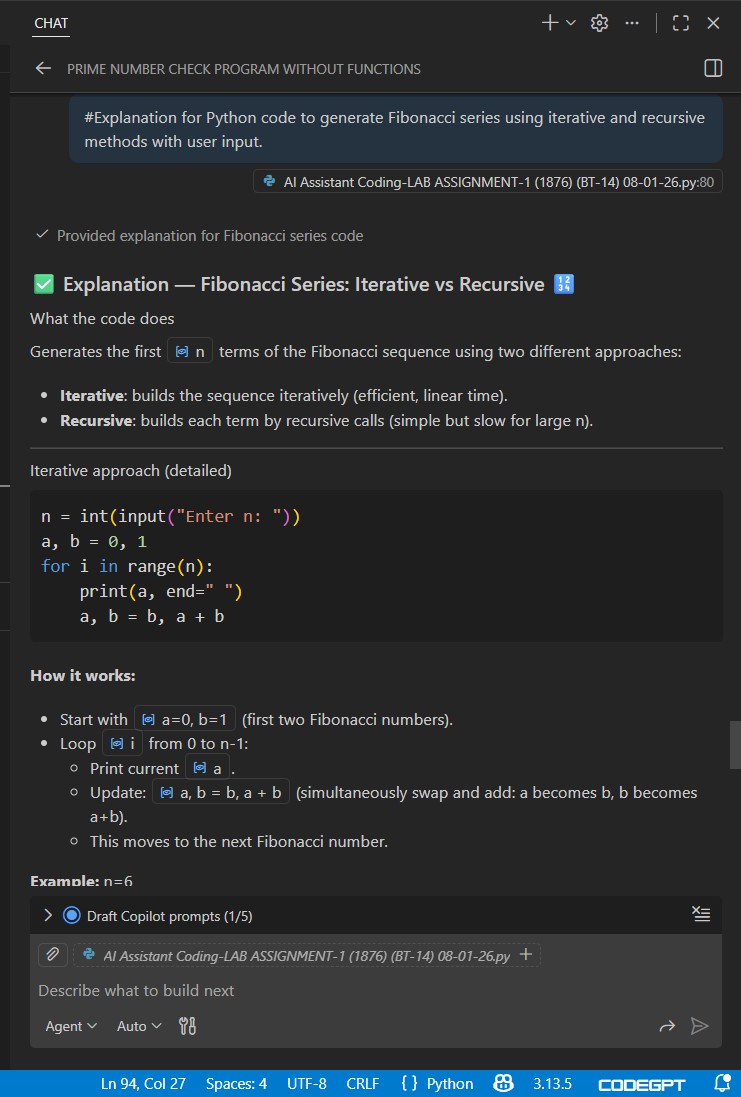
**Output:**

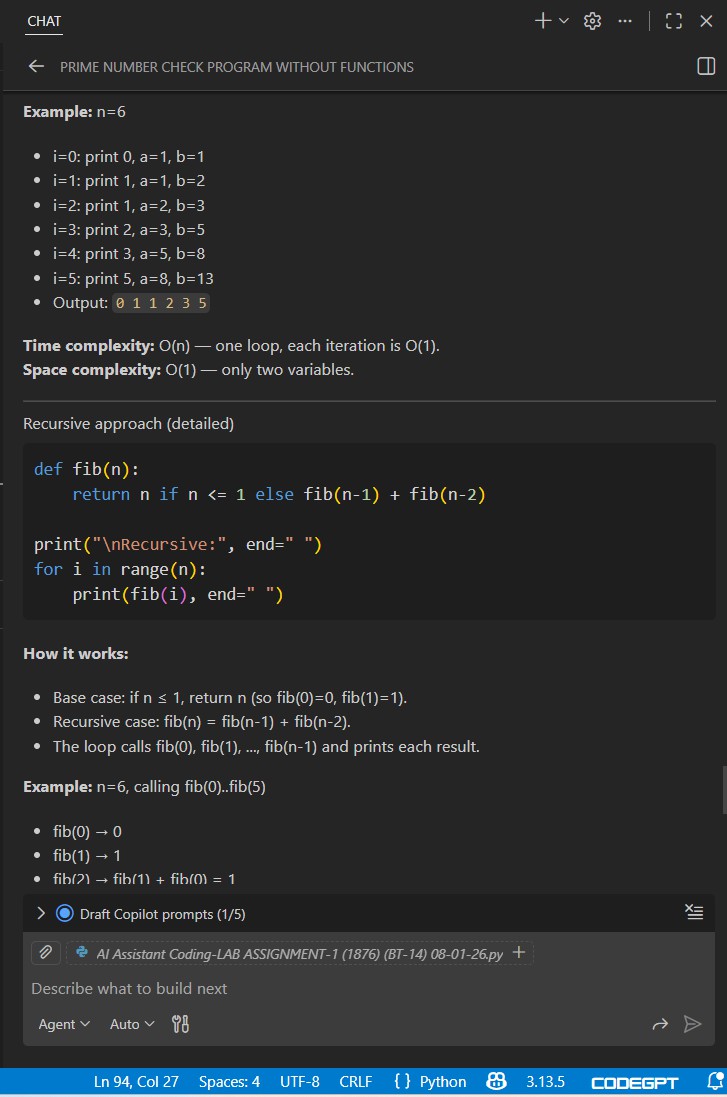


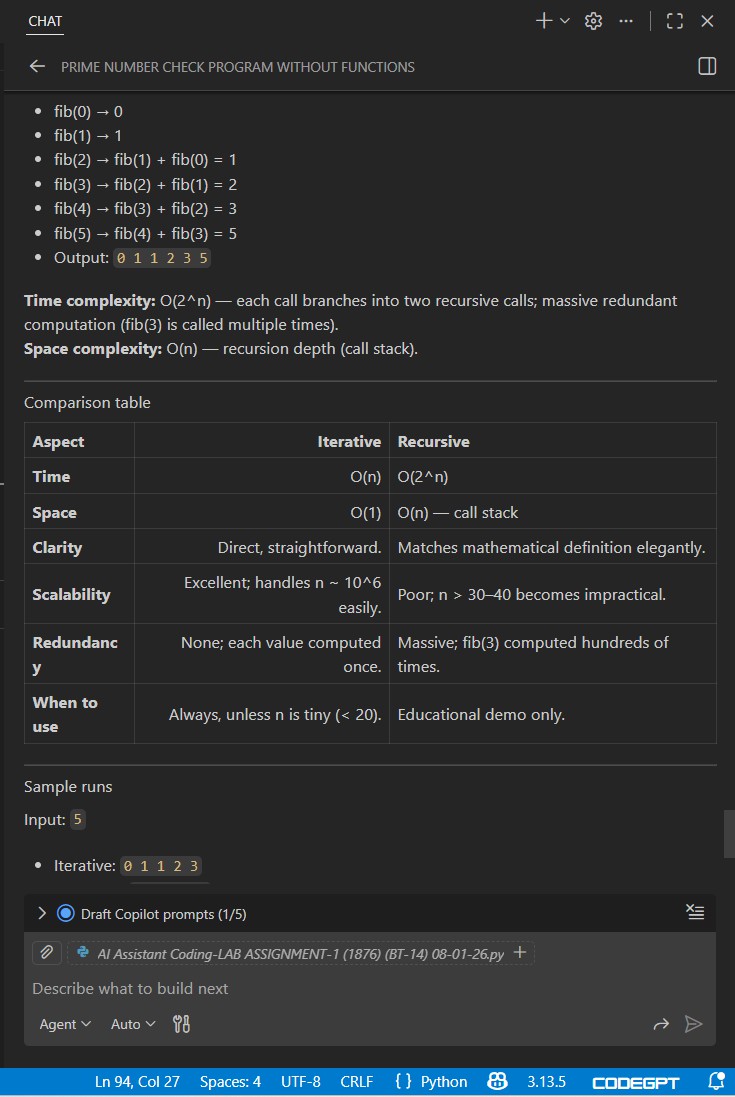
**Justification:**

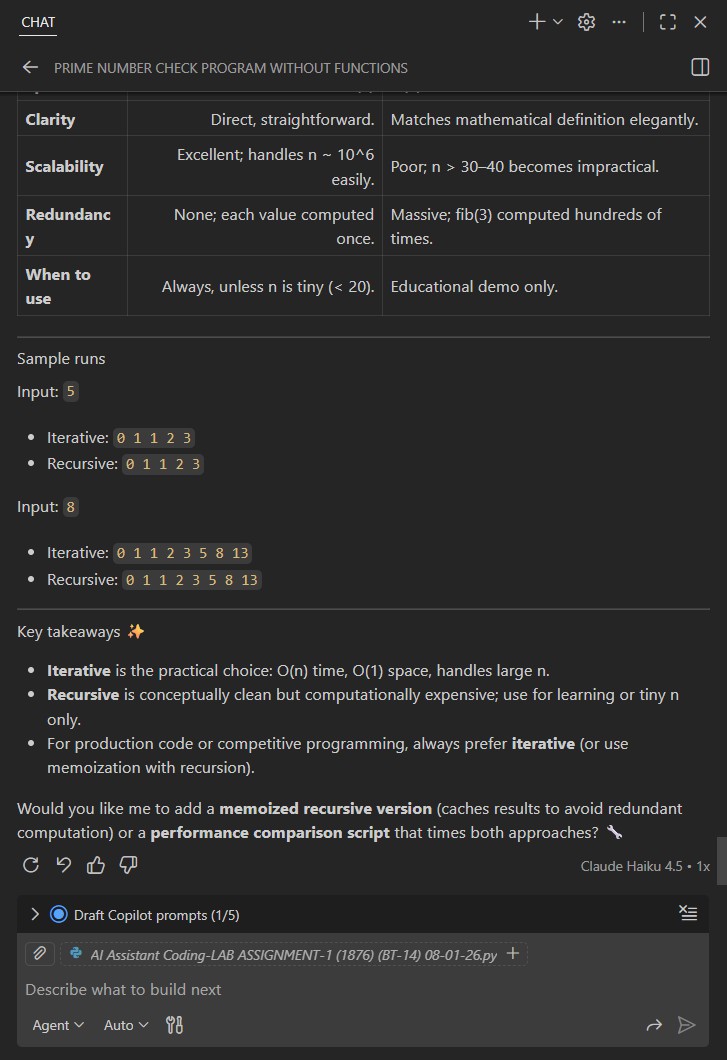
The basic approach checks divisibility up to N−1, resulting in unnecessary iterations and higher time complexity. The optimized approach checks only up to √N because any factor larger than √N must have a corresponding smaller factor.

**Explanation:**









**Task-5**

