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:

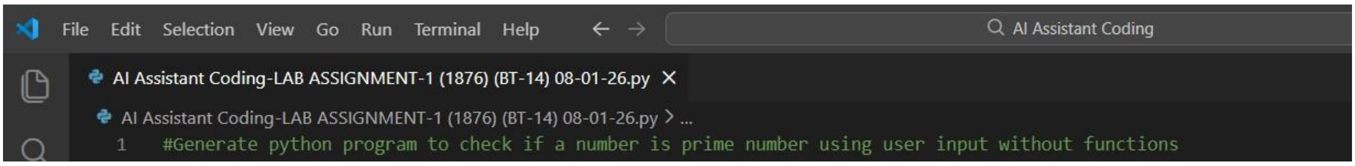
- **Checks whether a given number is prime**
- **Accepts user input**
- **Implements logic directly in the main code**
- **Does not use any user-defined functions**

- **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:

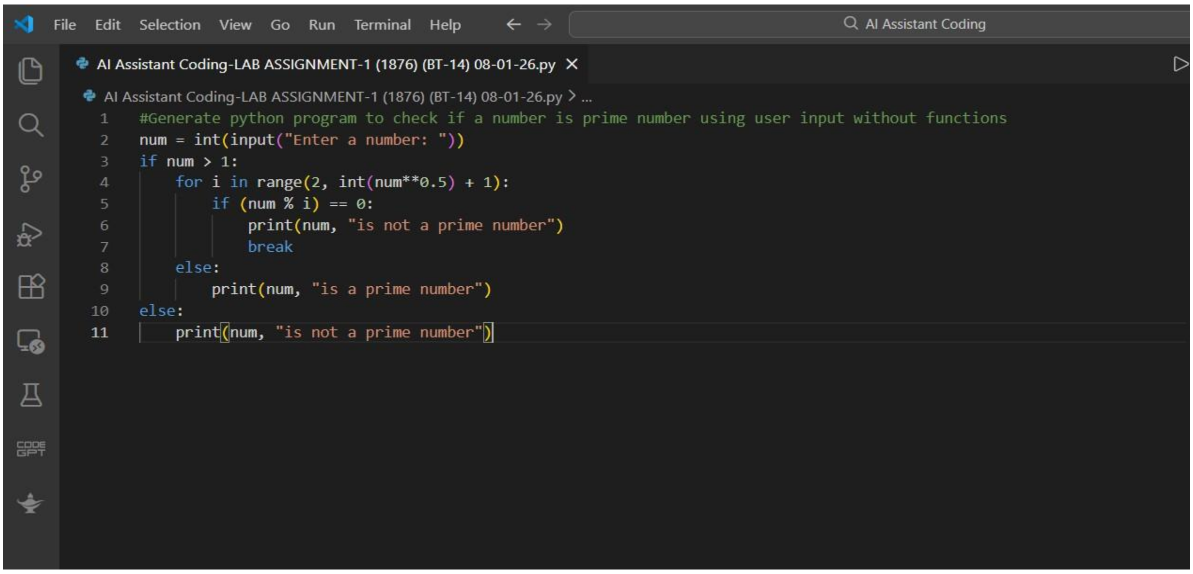


Code:

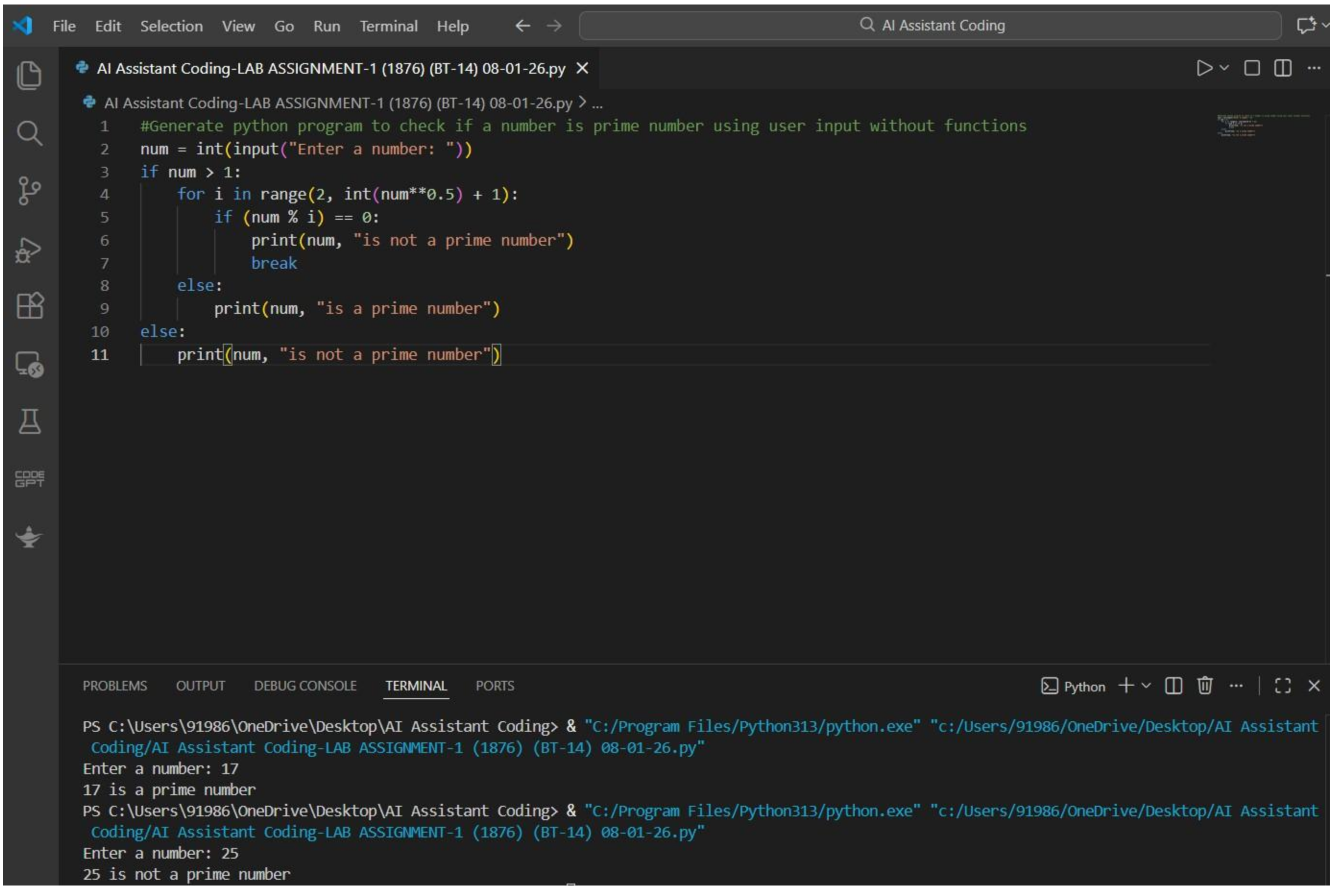
```
num = int(input("Enter a number: "))
if num > 1:
    for i in range(2, int(num**0.5) + 1):
        if (num % i) == 0:
            print(num, "is not a prime number")
            break
    else:
        print(num, "is a prime number")
else:
    print(num, "is not a prime number")
```

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

Screenshot:



Output:

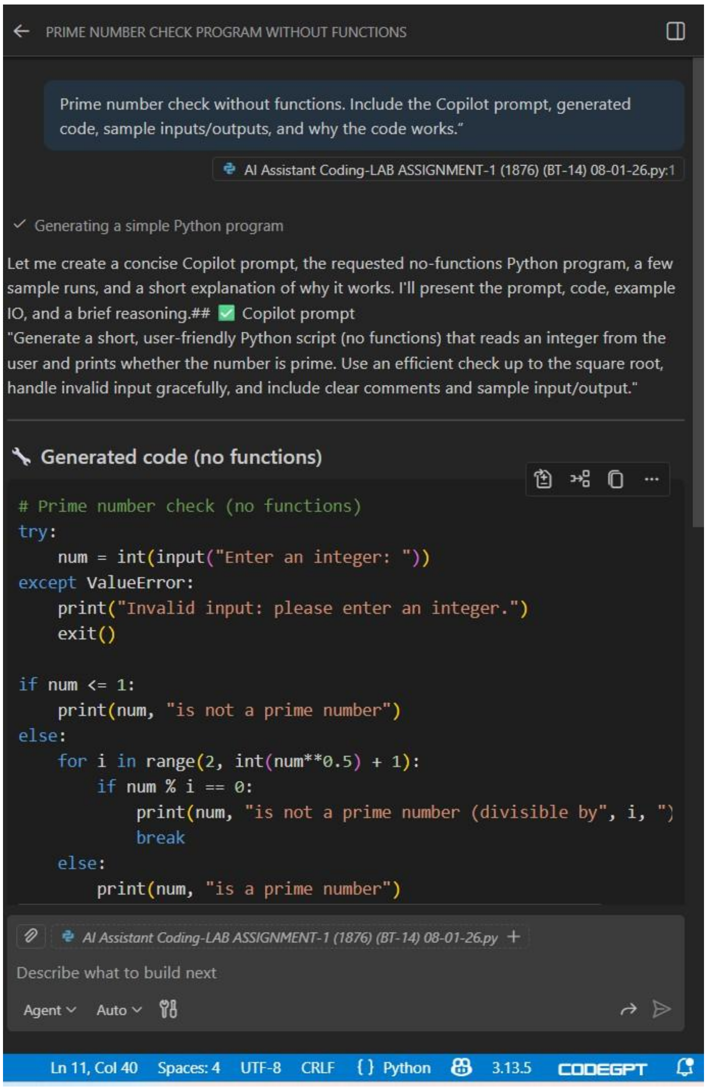


Explanation:

- 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.



CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

```python
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            print(num, "is not a prime number (divisible by", i, ")
            break
    else:
        print(num, "is a prime number")
```

Sample inputs / outputs

- Input: `17`
  Output: `17 is a prime number`
- Input: `1`
  Output: `1 is not a prime number`
- Input: `25`
  Output: `25 is not a prime number (divisible by 5)`
- Input: `abc`
  Output: `Invalid input: please enter an integer.`

Why the code works

- It 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).

> Tip: For very large inputs or repeated checks, consider using specialized algorithms or caching, but for single interactive checks this script is simple and efficient.

Raptor mini (Preview) • 1x

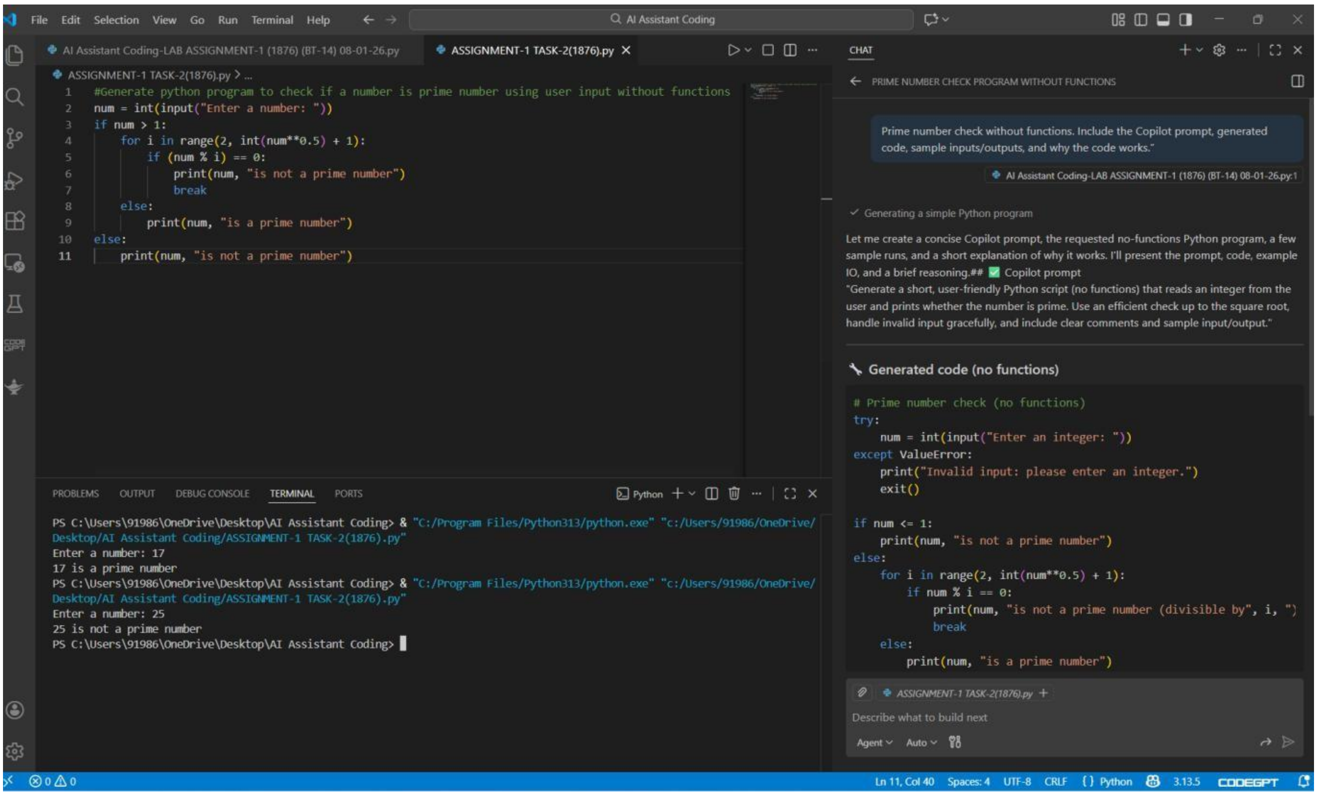
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

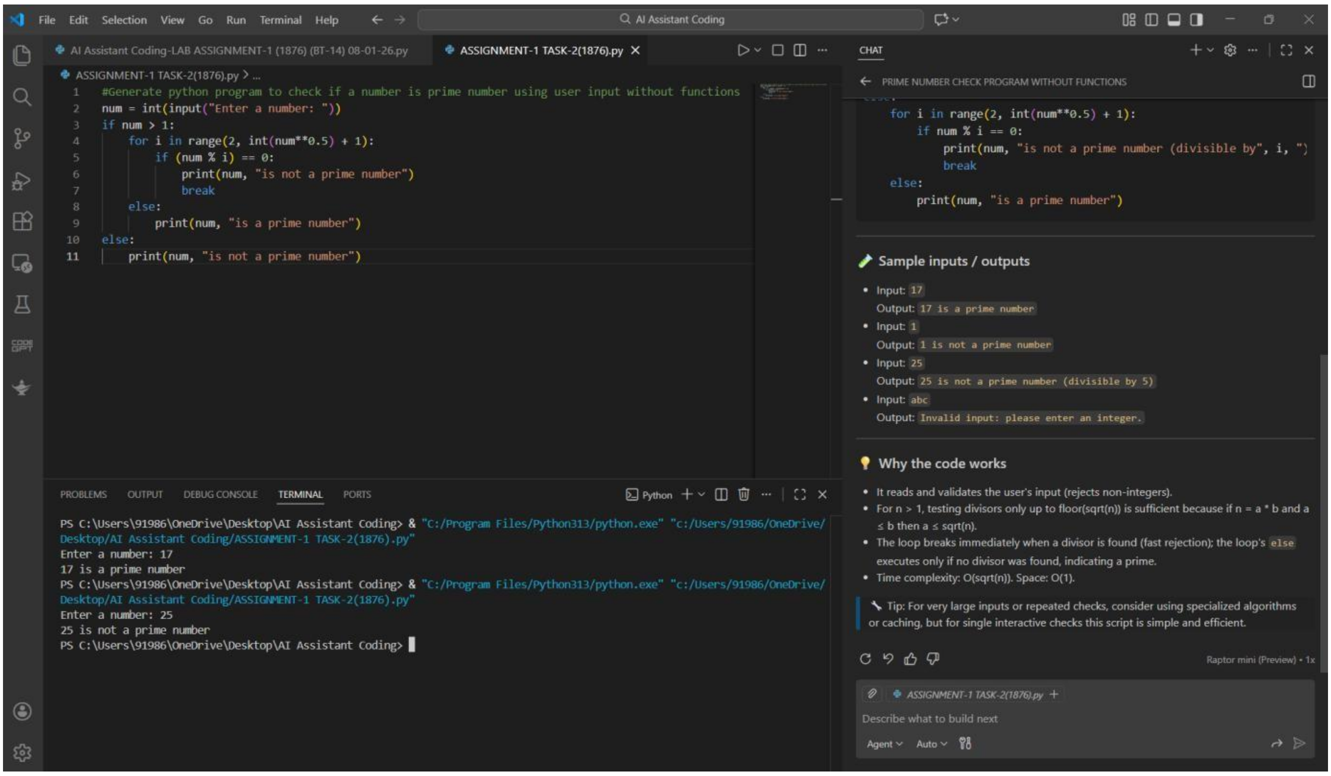
Describe what to build next

Agent Auto

Ln 11, Col 40 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

Task-1


File Edit Selection View Go Run Terminal Help
AI Assistant Coding
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py
ASSIGNMENT-1 TASK-2(1876).py
#Generate python program to check if a number is prime number using user input without functions
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")
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Enter a number: 17
17 is a prime number
Enter a number: 25
25 is not a prime number
CHAT
PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
Prime number check without functions. Include the Copilot prompt, generated code, sample inputs/outputs, and why the code works."
Generating a simple Python program
Let me create a concise Copilot prompt, the requested no-functions Python program, a few sample runs, and a short explanation of why it works. I'll present the prompt, code, example IO, and a brief reasoning.## Copilot prompt
"Generate a short, user-friendly Python script (no functions) that reads an integer from the user and prints whether the number is prime. Use an efficient check up to the square root, handle invalid input gracefully, and include clear comments and sample input/output."
Generated code (no functions)
# Prime number check (no functions)
try:
num = int(input("Enter an integer: "))
except ValueError:
print("Invalid input: please enter an integer.")
exit()
if num <= 1:
print(num, "is not a prime number")
else:
for i in range(2, int(num**0.5) + 1):
if num % i == 0:
print(num, "is not a prime number (divisible by", i, ")
break
else:
print(num, "is a prime number")
Describe what to build next


PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
Sample inputs / outputs
Input: 17
Output: 17 is a prime number
Input: 1
Output: 1 is not a prime number
Input: 25
Output: 25 is not a prime number (divisible by 5)
Input: abc
Output: Invalid input: please enter an integer.
Why the code works
It 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).
Tip: For very large inputs or repeated checks, consider using specialized algorithms or caching, but for single interactive checks this script is simple and efficient.
Enter a number: 17
17 is a prime number
Enter a number: 25
25 is not a prime number

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

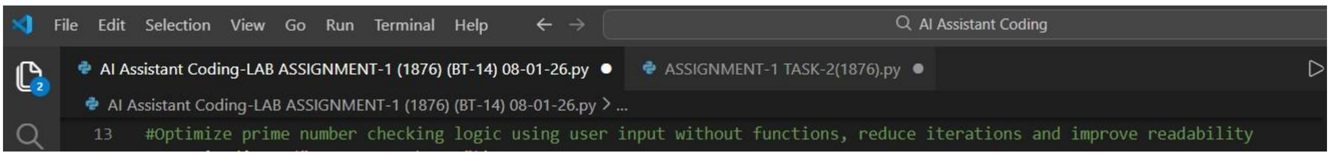
- ➢ Original and optimized code versions

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:



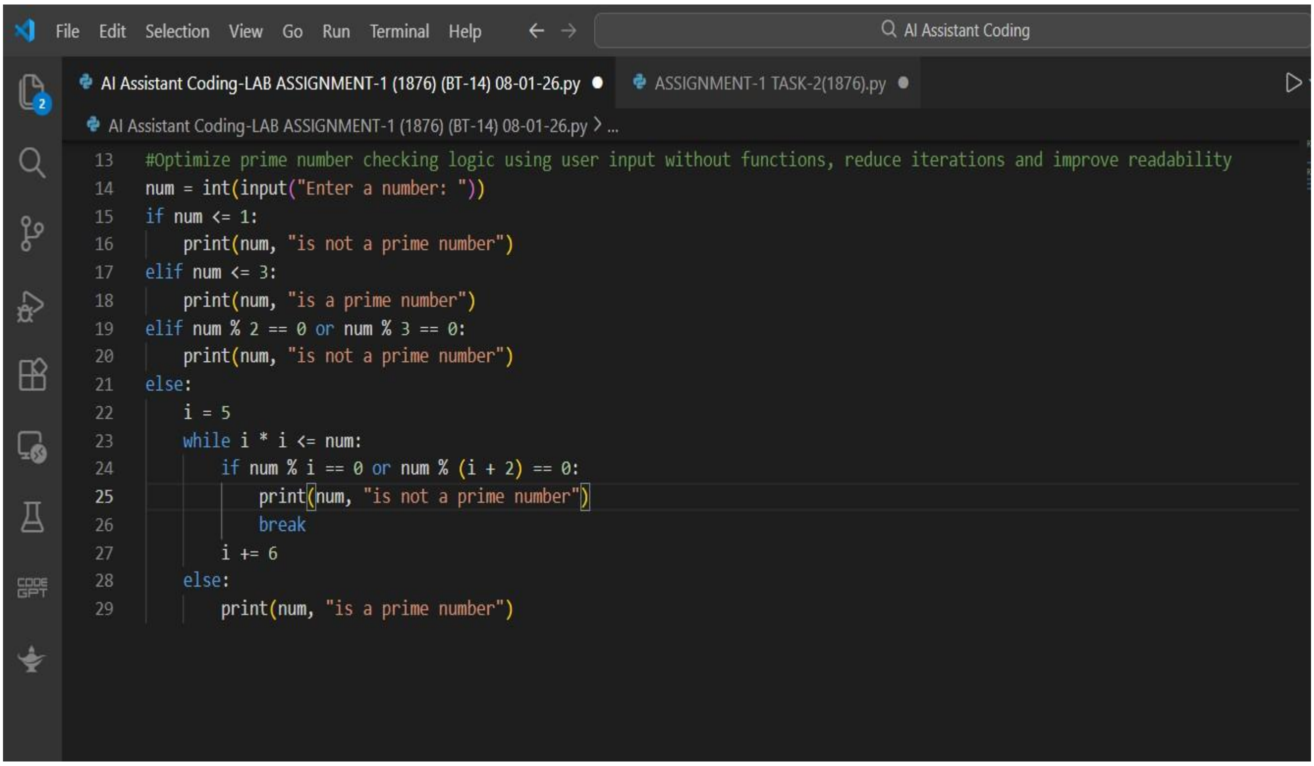
Code:

```
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")
```

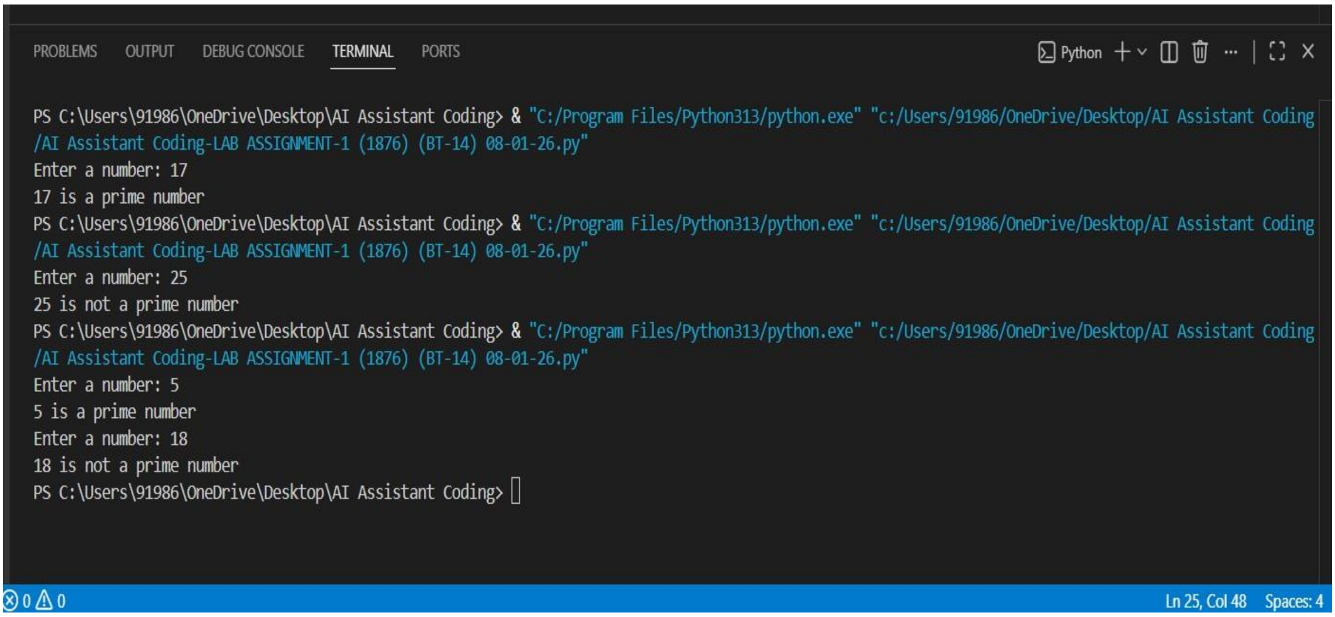
Scenario:

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:



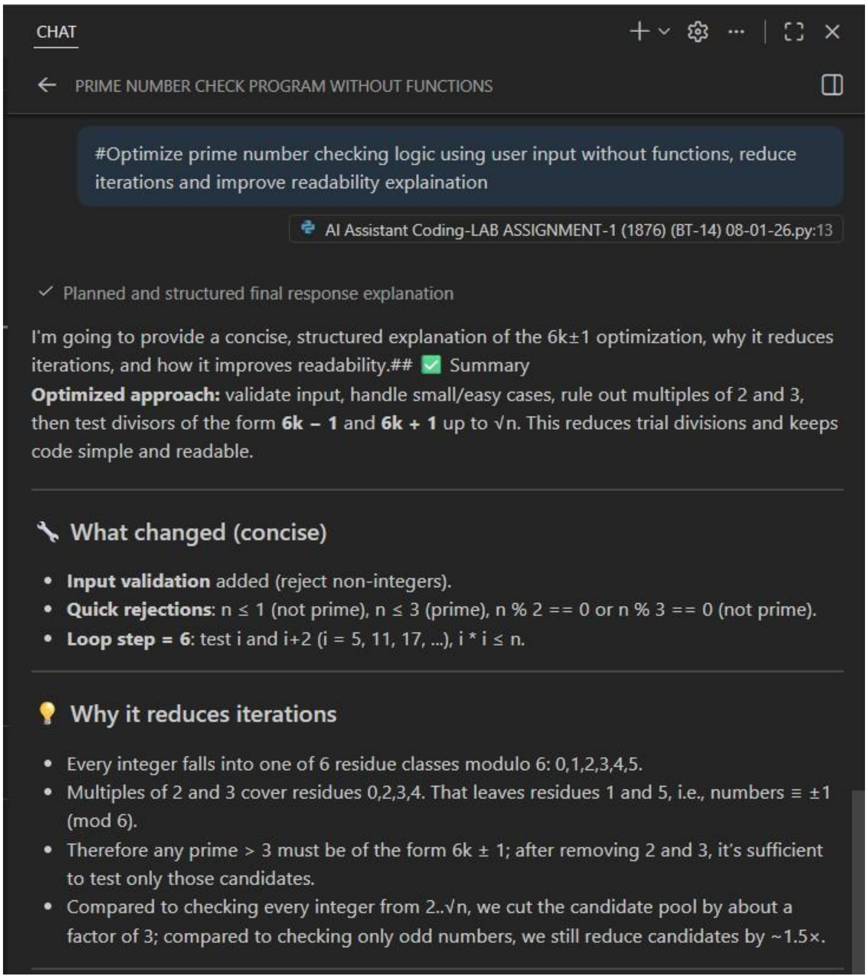
Output:



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:


CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

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

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py:13

✓ Planned and structured final response explanation

I'm going to provide a concise, structured explanation of the 6k±1 optimization, why it reduces iterations, and how it improves readability.## ✅ Summary
Optimized approach: validate input, handle small/easy cases, rule out multiples of 2 and 3, then test divisors of the form **6k − 1** and **6k + 1** up to √n. This reduces trial divisions and keeps code simple and readable.

🔧 What changed (concise)

- **Input validation** added (reject non-integers).
- **Quick rejections**: n ≤ 1 (not prime), n ≤ 3 (prime), n % 2 == 0 or n % 3 == 0 (not prime).
- **Loop step = 6**: test i and i+2 (i = 5, 11, 17, ...), i * i ≤ n.

💡 Why it reduces iterations

- Every integer falls into one of 6 residue classes modulo 6: 0,1,2,3,4,5.
- Multiples of 2 and 3 cover residues 0,2,3,4. That leaves residues 1 and 5, i.e., numbers ≡ ±1 (mod 6).
- Therefore any prime > 3 must be of the form 6k ± 1; after removing 2 and 3, it's sufficient to test only those candidates.
- Compared to checking every integer from 2..√n, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by ~1.5×.

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

💡 Why it reduces iterations

- Every integer falls into one of 6 residue classes modulo 6: 0,1,2,3,4,5.
- Multiples of 2 and 3 cover residues 0,2,3,4. That leaves residues 1 and 5, i.e., numbers ≡ ±1 (mod 6).
- Therefore any prime > 3 must be of the form 6k ± 1; after removing 2 and 3, it's sufficient to test only those candidates.
- Compared to checking every integer from 2..√n, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by ~1.5×.

⌛ Complexity

- Time: O(√n) (same asymptotic bound) but a smaller constant factor—fewer modulo operations in practice.
- Space: O(1).

🧪 Example inputs / outputs

- Input: `17` → `17 is a prime number`
- Input: `25` → `25 is not a prime number (divisible by 5)`
- Input: `1` → `1 is not a prime number`
- Input: `abc` → `Invalid input: please enter an integer.`

✨ Readability & maintainability benefits

- Clear separation: input validation → quick checks → main loop.
- The 6-step increment and checking i and i+2 makes intent explicit and concise.
- Comments (e.g., "Check 6k±1 up to sqrt(n)") make the logic easy to verify or extend.

If you want, I can add a tiny benchmark inside the file to show the actual reduction in divisor checks for a sample n. 🔧

Raptor mini (Preview) • 1x

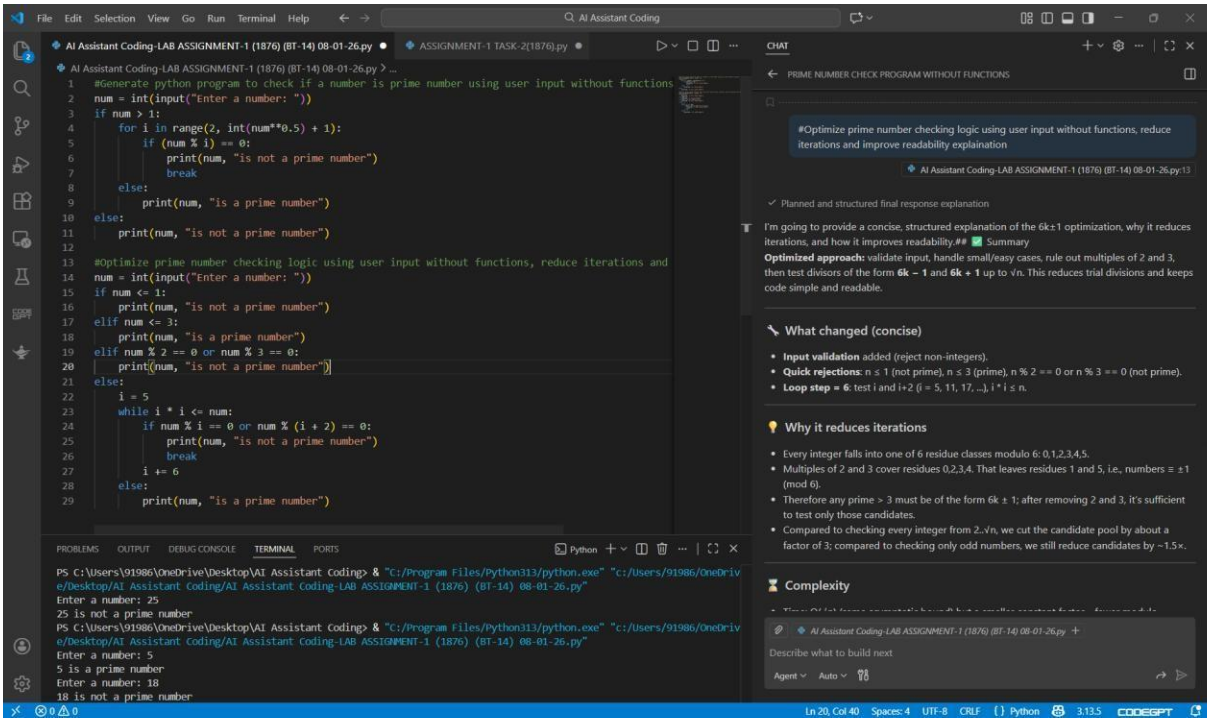
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

Describe what to build next

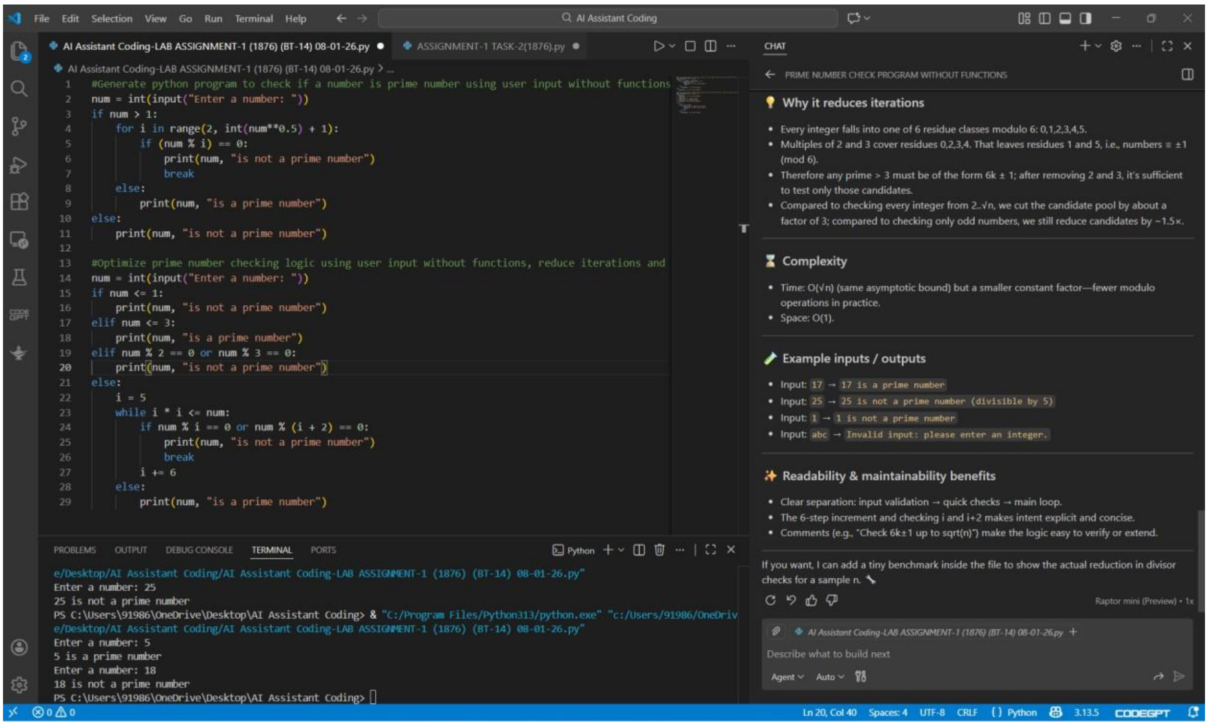
Agent Auto

Ln 20, Col 40 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

Task-2



File Edit Selection View Go Run Terminal Help
AI Assistant Coding
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py
ASSIGNMENT-1 TASK-2(1876).py
#Generate python program to check if a number is prime number using user input without functions
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 prime number")
#Optimize prime number checking logic using user input without functions, reduce iterations and
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")
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Enter a number: 25
25 is not a prime number
Enter a number: 5
5 is a prime number
Enter a number: 18
18 is not a prime number
CHAT
PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
#Optimize prime number checking logic using user input without functions, reduce iterations and improve readability explaination
Planned and structured final response explanation
I'm going to provide a concise, structured explanation of the 6k±1 optimization, why it reduces iterations, and how it improves readability.## Summary
Optimized approach: validate input, handle small/easy cases, rule out multiples of 2 and 3, then test divisors of the form 6k − 1 and 6k + 1 up to √n. This reduces trial divisions and keeps code simple and readable.
What changed (concise)
Input validation added (reject non-integers).
Quick rejections: n ≤ 1 (not prime), n ≤ 3 (prime), n % 2 == 0 or n % 3 == 0 (not prime).
Loop step = 6: test i and i+2 (i = 5, 11, 17, ...), i * i ≤ n.
Why it reduces iterations
Every integer falls into one of 6 residue classes modulo 6: 0,1,2,3,4,5.
Multiples of 2 and 3 cover residues 0,2,3,4. That leaves residues 1 and 5, i.e., numbers ≡ ±1 (mod 6).
Therefore any prime > 3 must be of the form 6k ± 1; after removing 2 and 3, it's sufficient to test only those candidates.
Compared to checking every integer from 2..√n, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by ~1.5×.
Complexity
Describe what to build next
Ln 20, Col 40 Spaces: 4 UTF-8 CRLF Python 3.13.5 CODEGPT



CHAT
PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
Why it reduces iterations
Complexity
Time: O(√n) (same asymptotic bound) but a smaller constant factor—fewer modulo operations in practice.
Space: O(1).
Example inputs / outputs
Input: 17 → 17 is a prime number
Input: 25 → 25 is not a prime number (divisible by 5)
Input: 1 → 1 is not a prime number
Input: abc → Invalid input: please enter an integer.
Readability & maintainability benefits
Clear separation: input validation → quick checks → main loop.
The 6-step increment and checking i and i+2 makes intent explicit and concise.
Comments (e.g., "Check 6k±1 up to sqrt(n)") make the logic easy to verify or extend.
If you want, I can add a tiny benchmark inside the file to show the actual reduction in divisor checks for a sample n.
Raptor mini (Preview) • 1x
Describe what to build next

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)

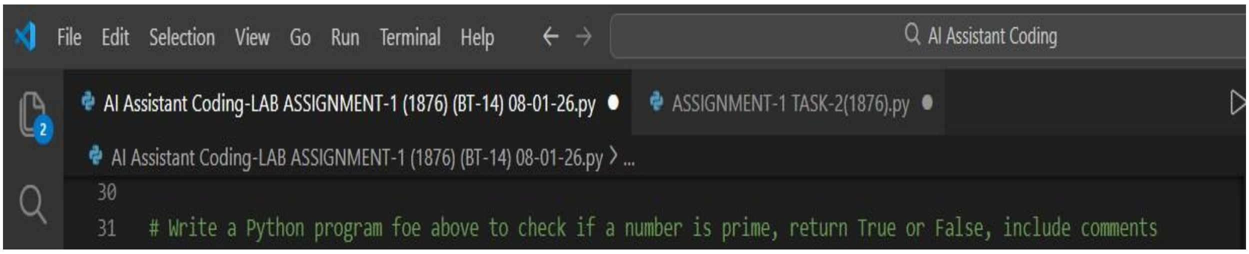
❖ Expected Output

- ➢ 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:



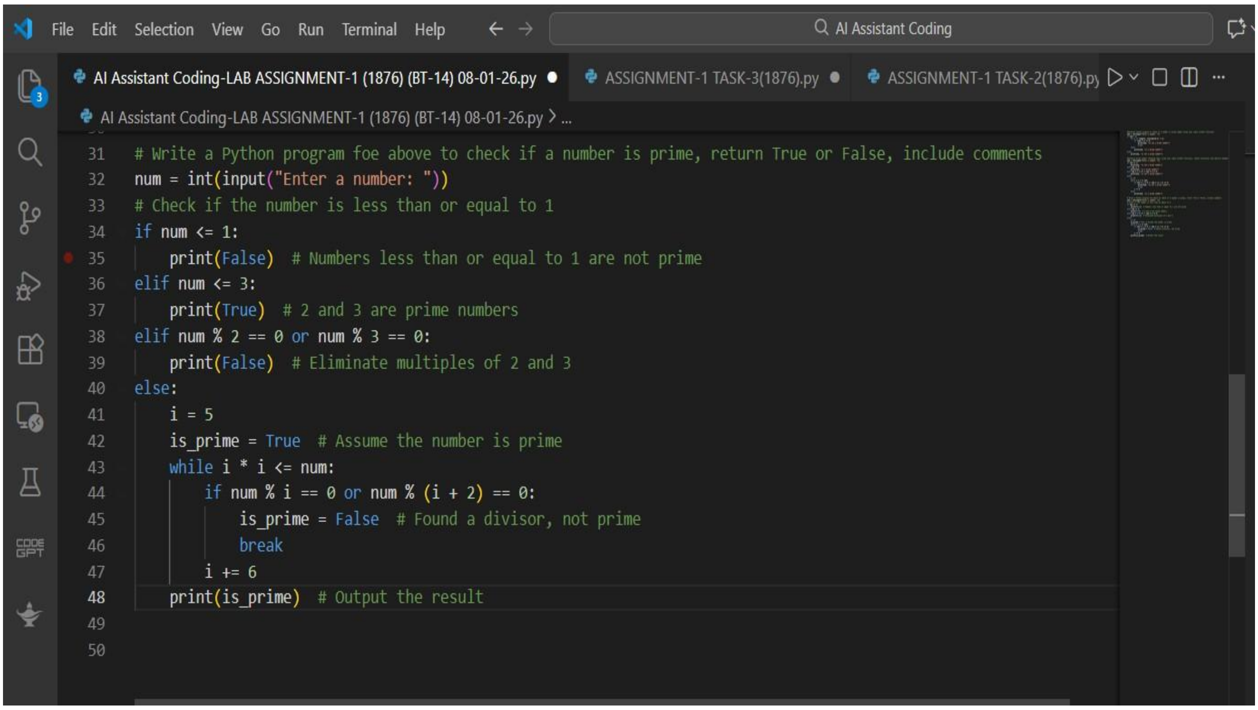
Code:

```
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
```

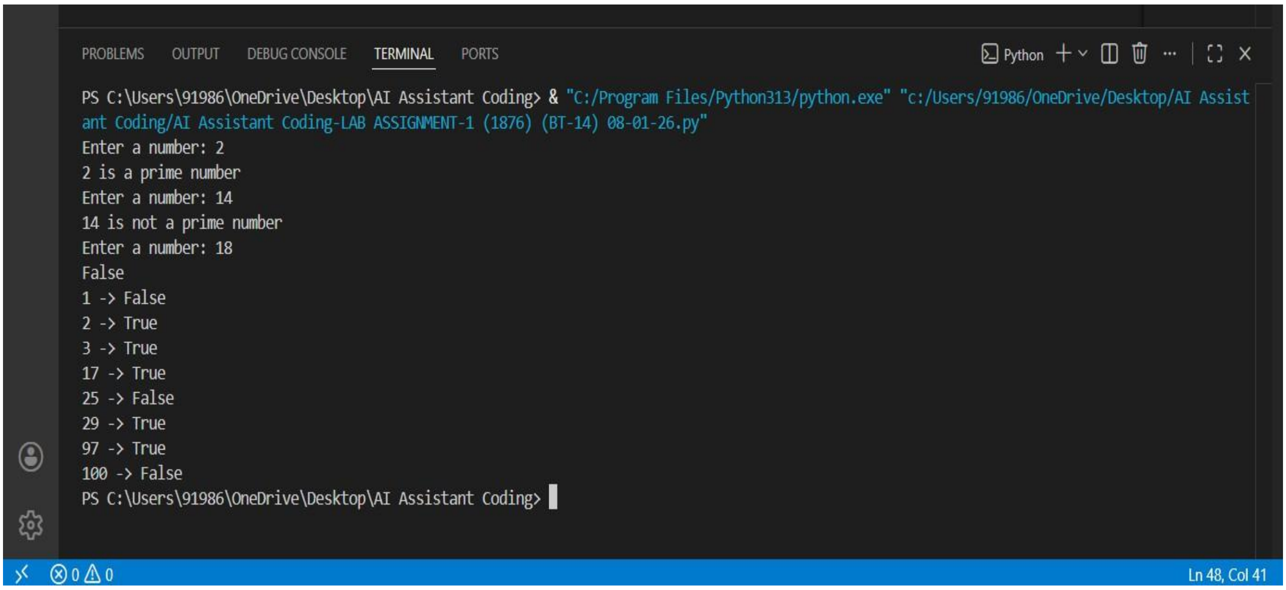
Scenario:

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:



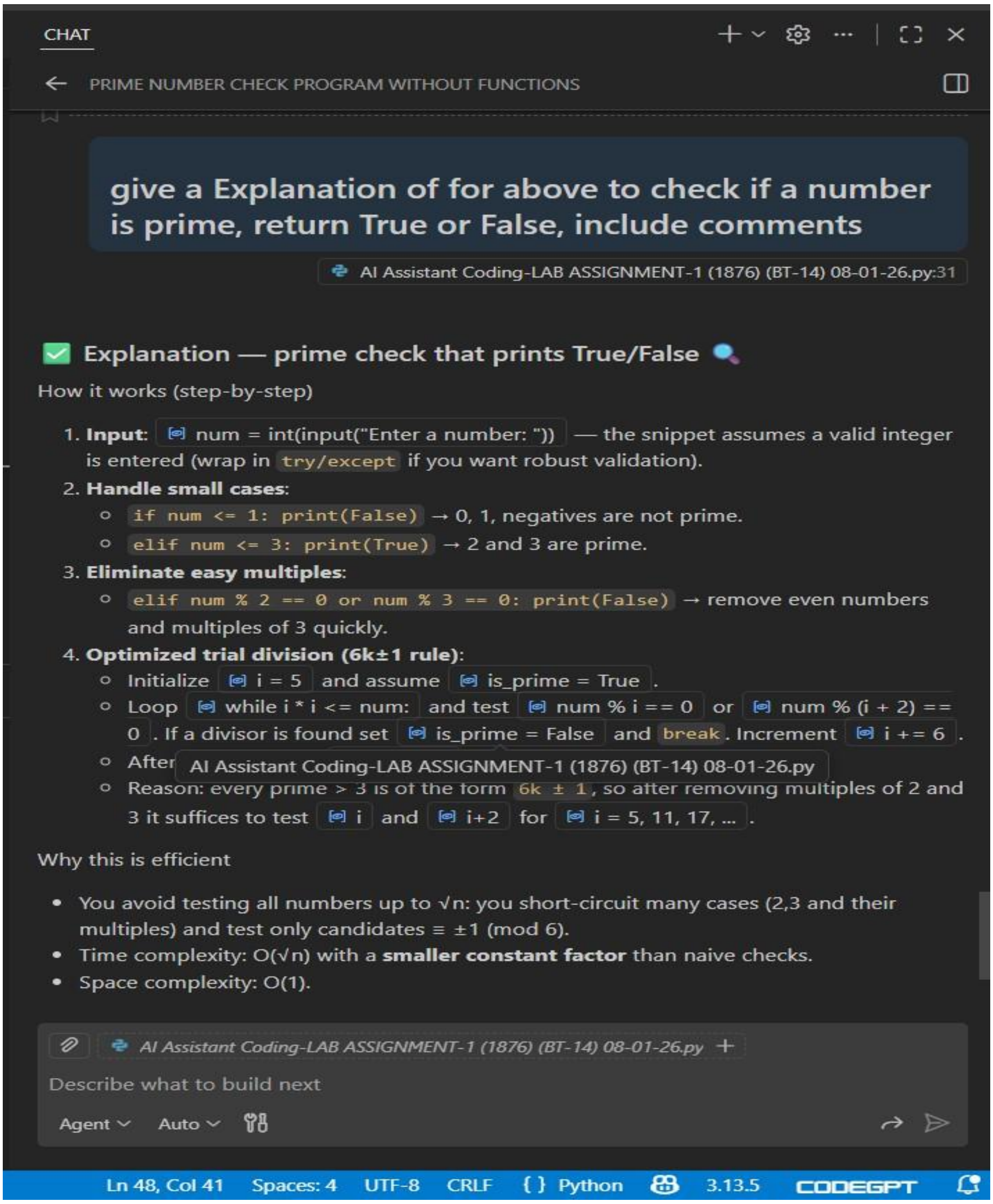
Output:



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:



CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

1. **Input:** `num = int(input("Enter a number: "))` — the snippet assumes a valid integer is entered (wrap in `try/except` if you want robust validation).
2. **Handle small cases:**
   - `if num <= 1: print(False)` → 0, 1, negatives are not prime.
   - `elif num <= 3: print(True)` → 2 and 3 are prime.
3. **Eliminate easy multiples:**
   - `elif num % 2 == 0 or num % 3 == 0: print(False)` → remove even numbers and multiples of 3 quickly.
4. **Optimized trial division (6k±1 rule):**
   - Initialize `i = 5` and assume `is_prime = True`.
   - Loop `while i * i <= num:` and test `num % i == 0` or `num % (i + 2) == 0`. If a divisor is found set `is_prime = False` and `break`. Increment `i += 6`.
   - After the loop print `is_prime`.
   - Reason: every prime > 3 is of the form `6k ± 1`, so after removing multiples of 2 and 3 it suffices to test `i` and `i+2` for `i = 5, 11, 17, ...`.

Why this is efficient

- You avoid testing all numbers up to √n: you short-circuit many cases (2,3 and their multiples) and test only candidates ≡ ±1 (mod 6).
- Time complexity: O(√n) with a **smaller constant factor** than naive checks.
- Space complexity: O(1).

Comments in the code

- Comments document intent: small-case handling, quick eliminations, and the `6k ± 1` loop — making the logic readable and maintainable.

Example

- Input `17` → prints `True`
- Input `25` → prints `False` (divisible by 5)

Would you like me to add a robust input validation wrapper or replace the print with a callable `is_prime()` function? ✨

Raptor mini (Preview) • 1x

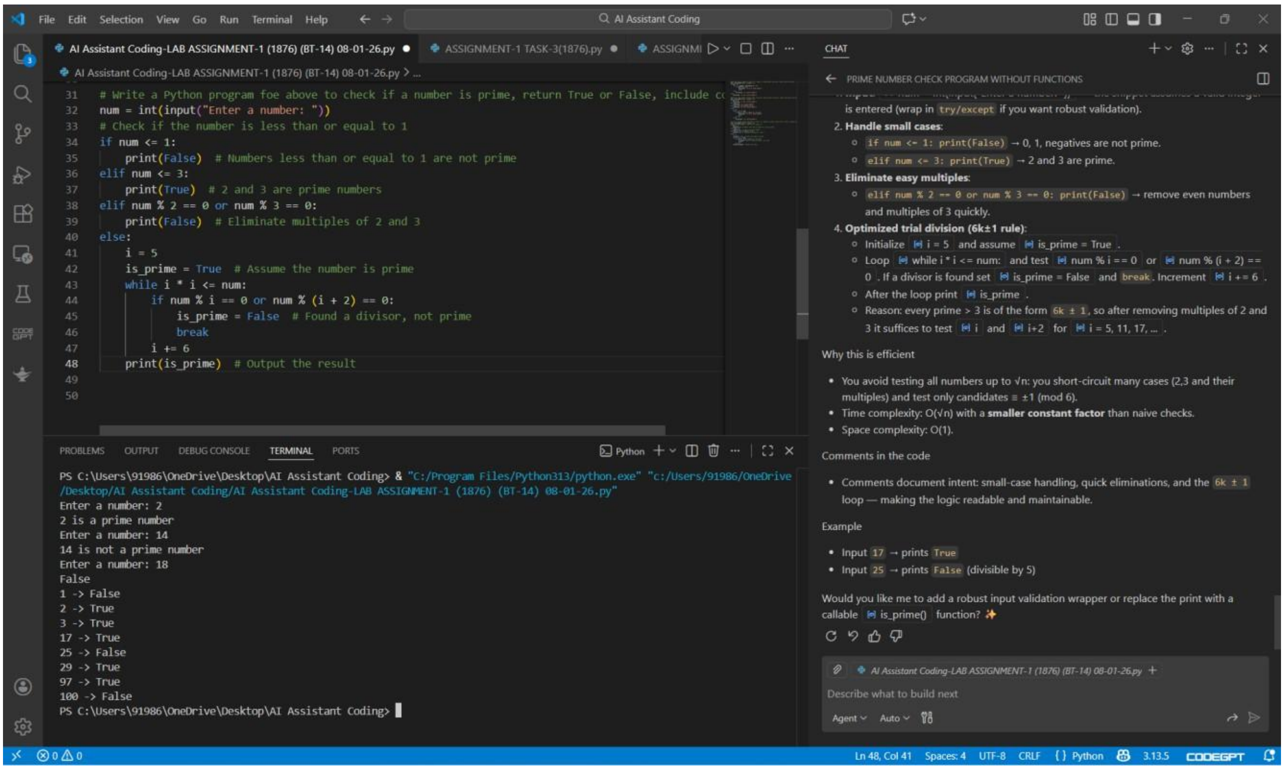
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

Describe what to build next

Agent Auto

Ln 48, Col 41 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

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

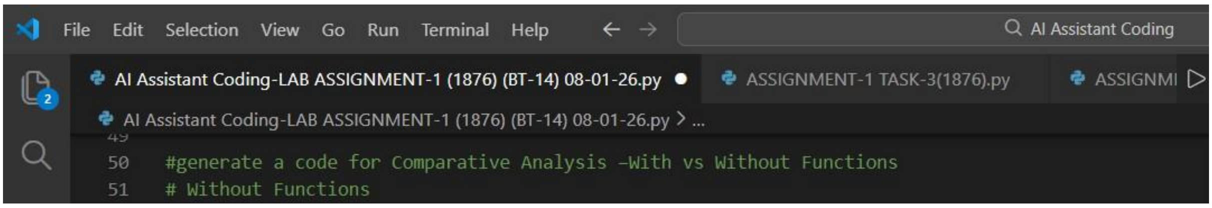
Comparison table or short analytical report

Prompt:

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

 # Without Functions

Screenshot:

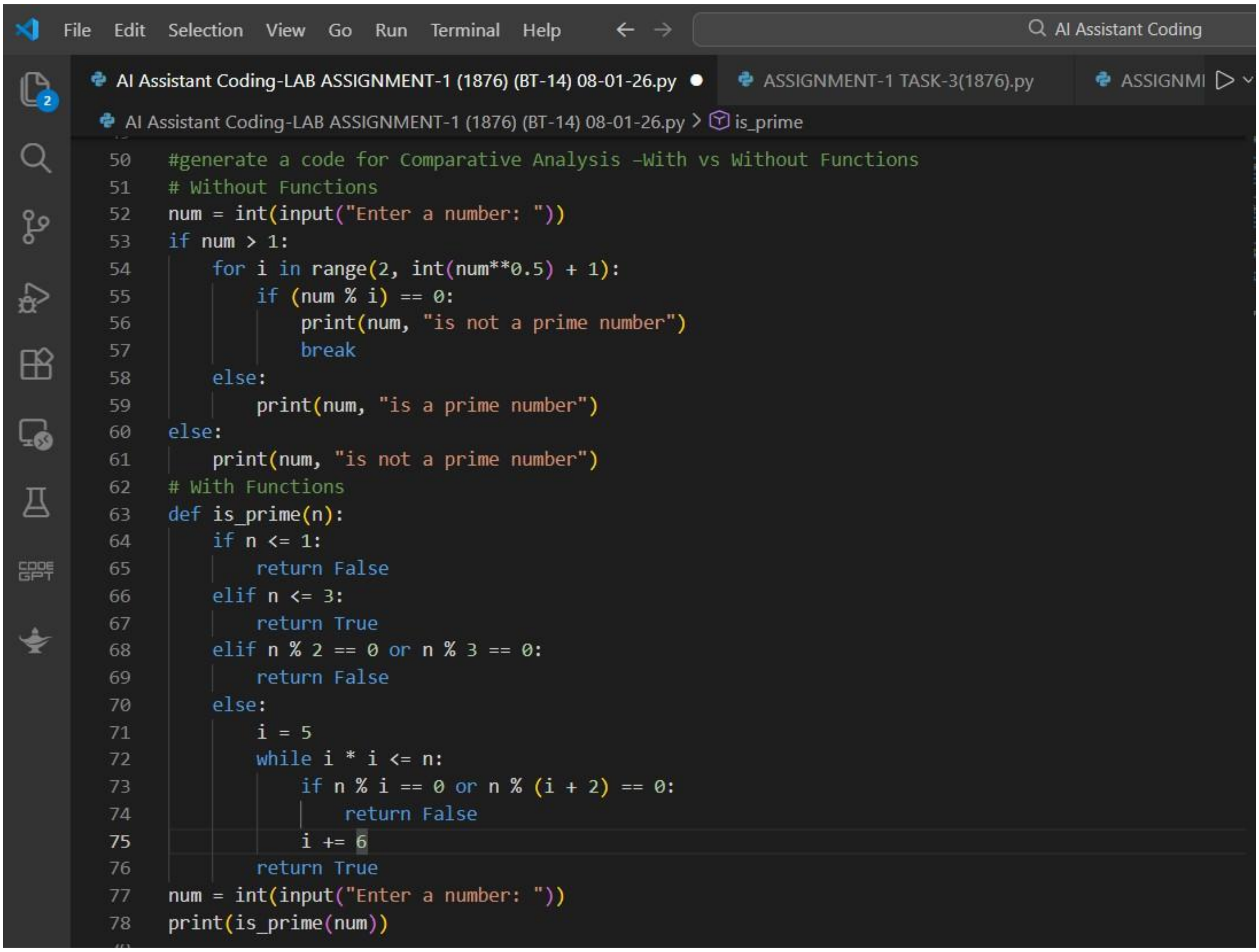


Code:

```
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:



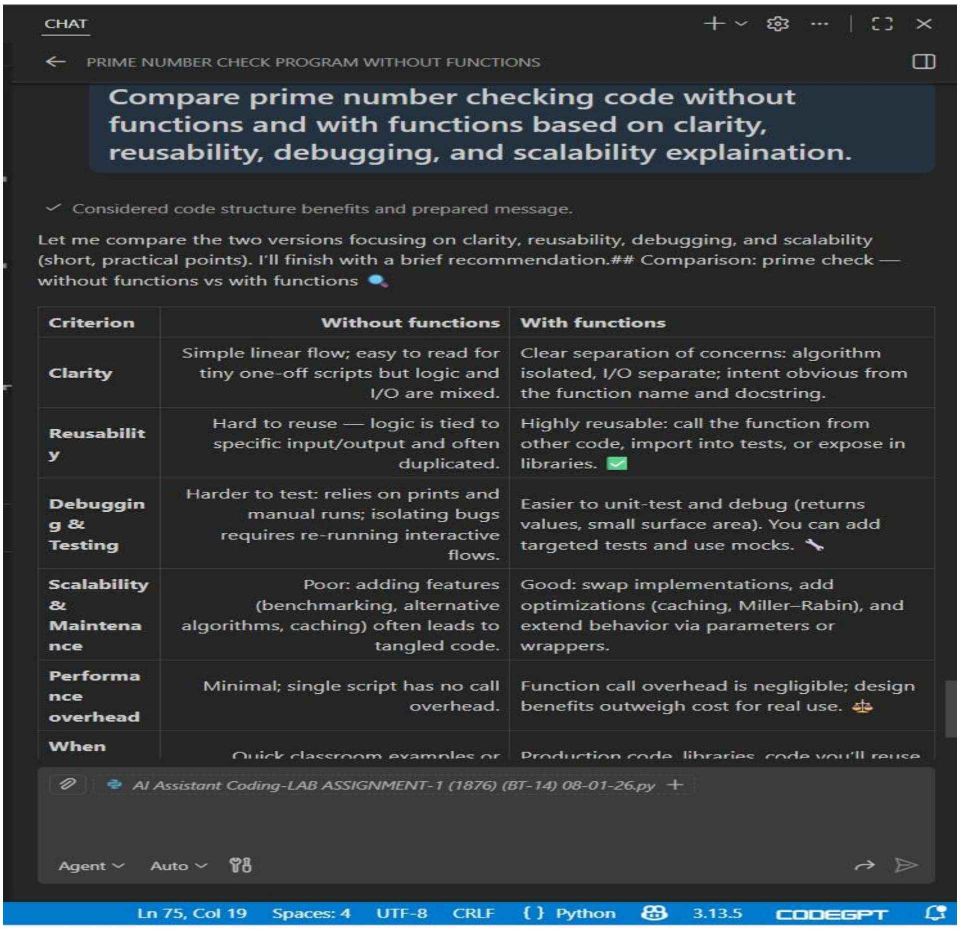
Output:

```
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding> & "C:/Program Files/Python313/python.exe" "c:/Users/91986/OneDrive/Desktop/AI Assistant Coding/AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py"
Enter a number: 2
2 is a prime number
Enter a number: 12
12 is not a prime number
Enter a number: 14
False
1 -> False
2 -> True
3 -> True
17 -> True
25 -> False
29 -> True
97 -> True
100 -> False
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding>
```

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:



CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

Compare prime number checking code without functions and with functions based on clarity, reusability, debugging, and scalability explaination.

Considered code structure benefits and prepared message.

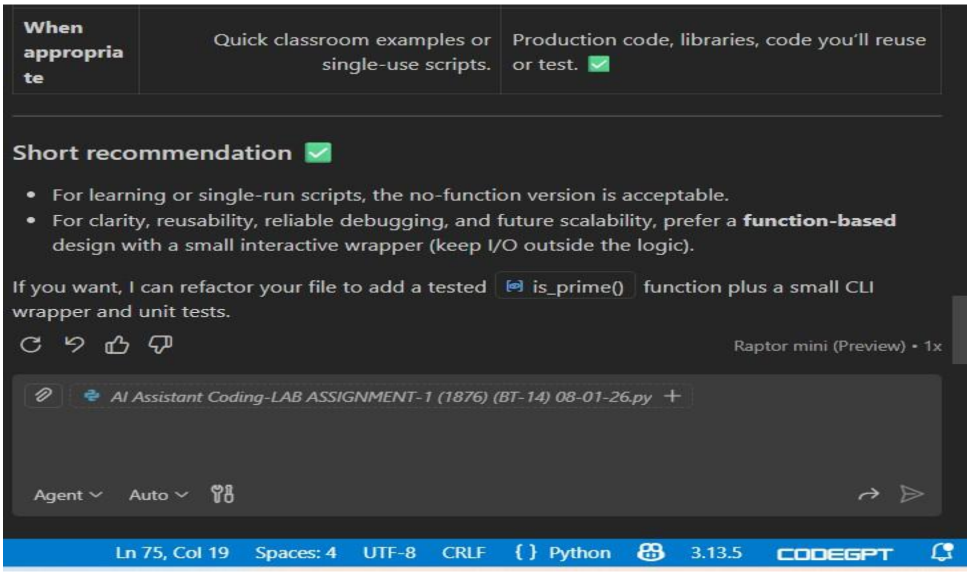
Let me compare the two versions focusing on clarity, reusability, debugging, and scalability (short, practical points). I'll finish with a brief recommendation.## Comparison: prime check — without functions vs with functions 🔍

Criterion	Without functions	With functions
Clarity	Simple linear flow; easy to read for tiny one-off scripts but logic and I/O are mixed.	Clear separation of concerns: algorithm isolated, I/O separate; intent obvious from the function name and docstring.
Reusability	Hard to reuse — logic is tied to specific input/output and often duplicated.	Highly reusable: call the function from other code, import into tests, or expose in libraries. ✅
Debugging & Testing	Harder to test: relies on prints and manual runs; isolating bugs requires re-running interactive flows.	Easier to unit-test and debug (returns values, small surface area). You can add targeted tests and use mocks. 🔧
Scalability & Maintenance	Poor: adding features (benchmarking, alternative algorithms, caching) often leads to tangled code.	Good: swap implementations, add optimizations (caching, Miller–Rabin), and extend behavior via parameters or wrappers.
Performance overhead	Minimal; single script has no call overhead.	Function call overhead is negligible; design benefits outweigh cost for real use. ⚖️
When	Quick classroom examples or	Production code, libraries, code you'll reuse

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

Agent ⌄ Auto ⌄

Ln 75, Col 19 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT



Criterion	Without functions	With functions
When appropriate	Quick classroom examples or single-use scripts.	Production code, libraries, code you'll reuse or test. ✅

Short recommendation ✅

- For learning or single-run scripts, the no-function version is acceptable.
- For clarity, reusability, reliable debugging, and future scalability, prefer a **function-based** design with a small interactive wrapper (keep I/O outside the logic).

If you want, I can refactor your file to add a tested is_prime() function plus a small CLI wrapper and unit tests.

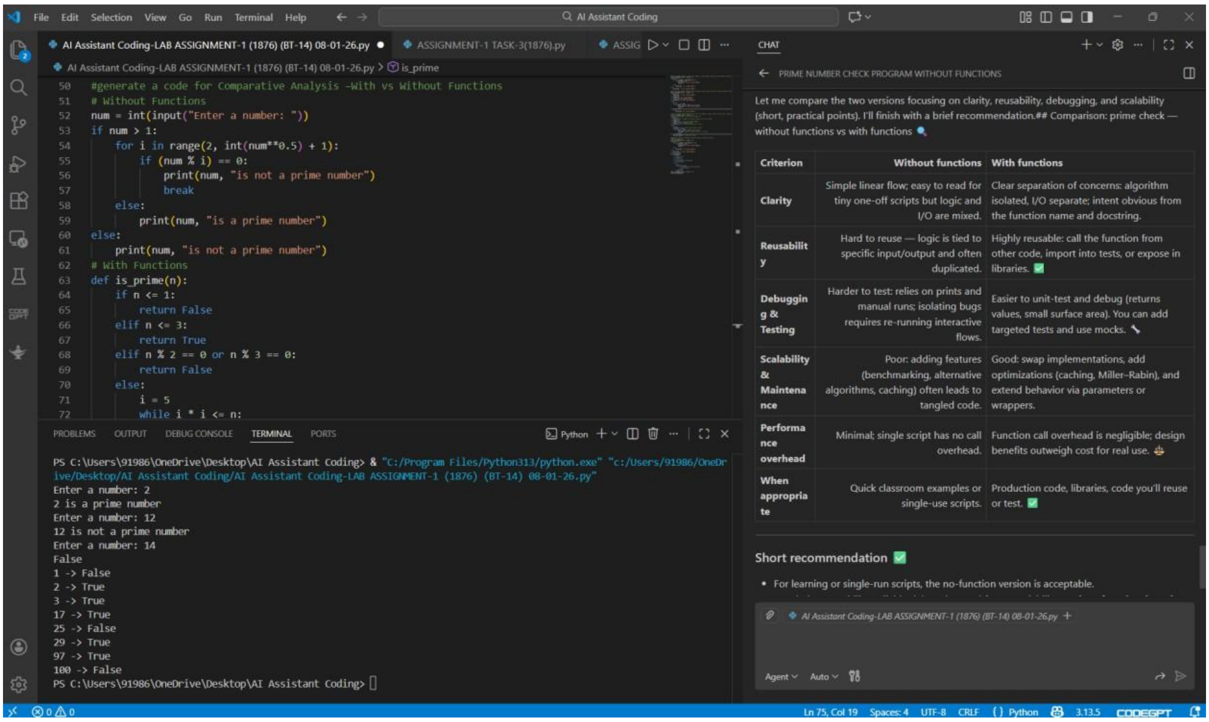
Raptor mini (Preview) • 1x

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

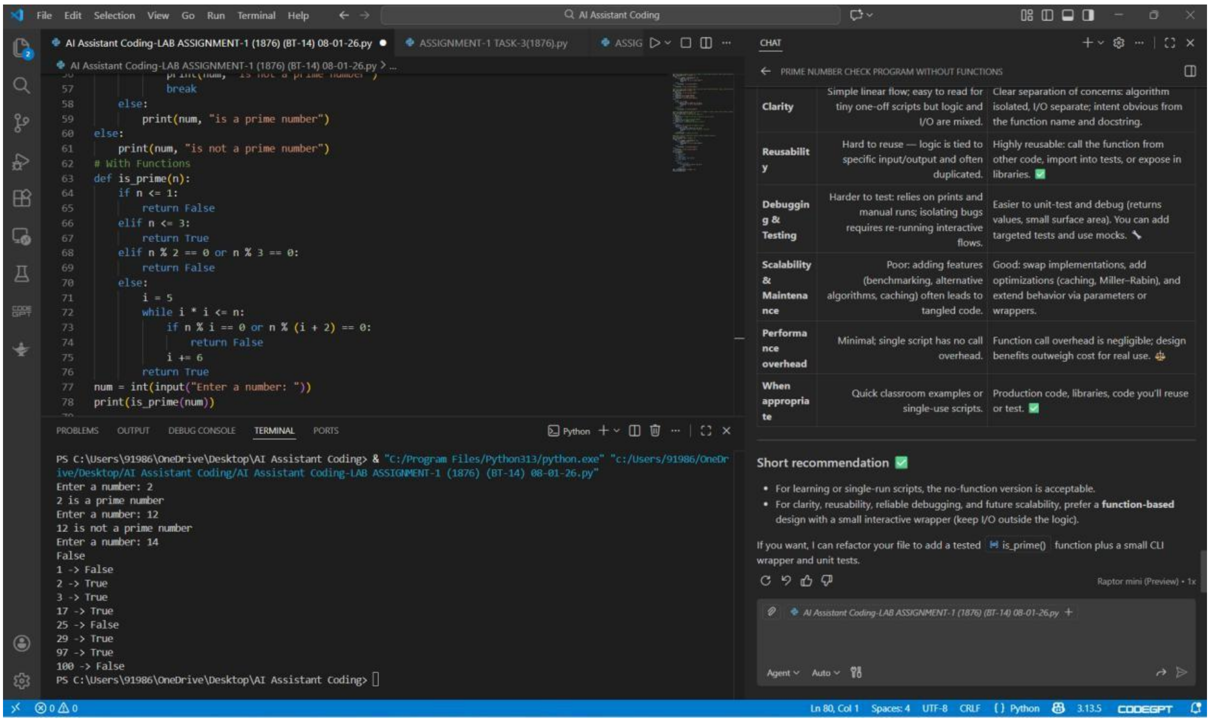
Agent ⌄ Auto ⌄

Ln 75, Col 19 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

Task-4



AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py
#generate a code for Comparative Analysis -With vs Without Functions
# Without Functions
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:
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding> & "C:/Program Files/Python313/python.exe" "c:/Users/91986/OneDrive/Desktop/AI Assistant Coding/AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py"
Enter a number: 2
2 is a prime number
Enter a number: 12
12 is not a prime number
Enter a number: 14
False
1 -> False
2 -> True
3 -> True
17 -> True
25 -> False
29 -> True
97 -> True
100 -> False
PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
Let me compare the two versions focusing on clarity, reusability, debugging, and scalability (short, practical points). I'll finish with a brief recommendation.## Comparison: prime check — without functions vs with functions
Criterion | Without functions | With functions
Clarity | Simple linear flow; easy to read for tiny one-off scripts but logic and I/O are mixed. | Clear separation of concerns: algorithm isolated, I/O separate; intent obvious from the function name and docstring.
Reusability | Hard to reuse — logic is tied to specific input/output and often duplicated. | Highly reusable: call the function from other code, import into tests, or expose in libraries.
Debugging & Testing | Harder to test: relies on prints and manual runs; isolating bugs requires re-running interactive flows. | Easier to unit-test and debug (returns values, small surface area). You can add targeted tests and use mocks.
Scalability & Maintenance | Poor: adding features (benchmarking, alternative algorithms, caching) often leads to tangled code. | Good: swap implementations, add optimizations (caching, Miller-Rabin), and extend behavior via parameters or wrappers.
Performance overhead | Minimal; single script has no call overhead. | Function call overhead is negligible; design benefits outweigh cost for real use.
When appropriate | Quick classroom examples or single-use scripts. | Production code, libraries, code you'll reuse or test.
Short recommendation
For learning or single-run scripts, the no-function version is acceptable.



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))
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding> & "C:/Program Files/Python313/python.exe" "c:/Users/91986/OneDrive/Desktop/AI Assistant Coding/AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py"
Enter a number: 2
2 is a prime number
Enter a number: 12
12 is not a prime number
Enter a number: 14
False
1 -> False
2 -> True
3 -> True
17 -> True
25 -> False
29 -> True
97 -> True
100 -> False
Short recommendation
For learning or single-run scripts, the no-function version is acceptable.
For clarity, reusability, reliable debugging, and future scalability, prefer a function-based design with a small interactive wrapper (keep I/O outside the logic).
If you want, I can refactor your file to add a tested is_prime() function plus a small CLI wrapper and unit tests.

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)

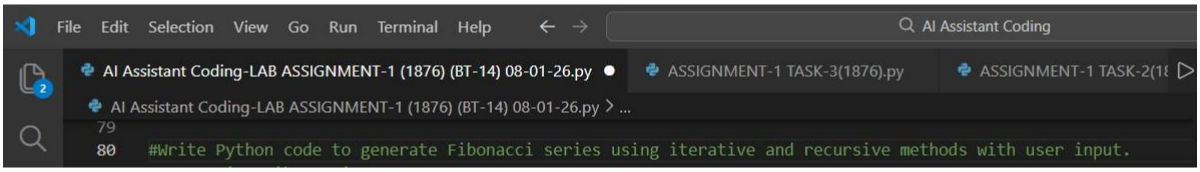
- Expected Output

  - 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.

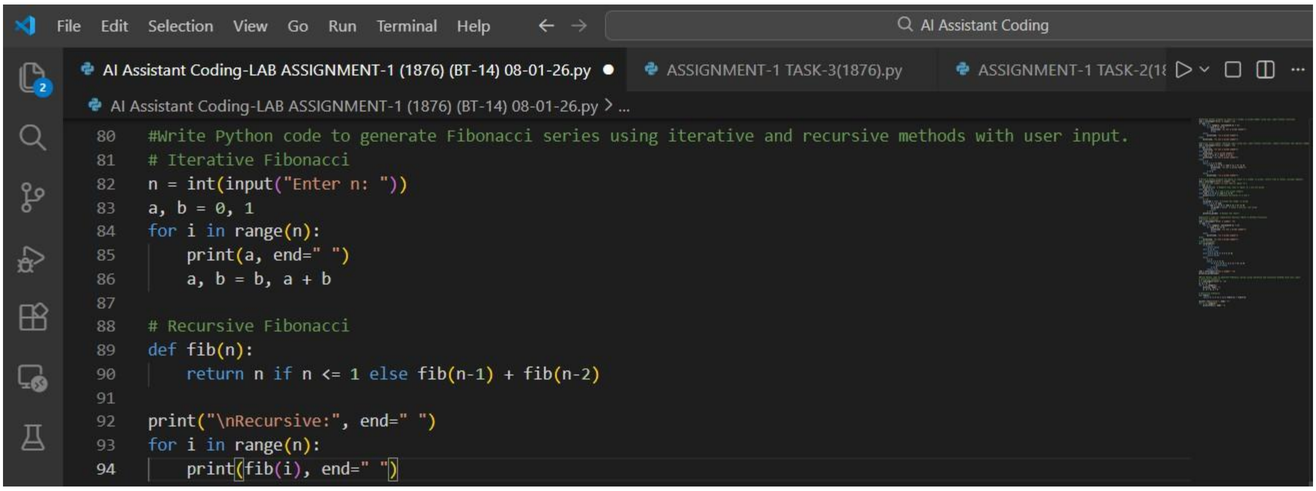
Screenshot:



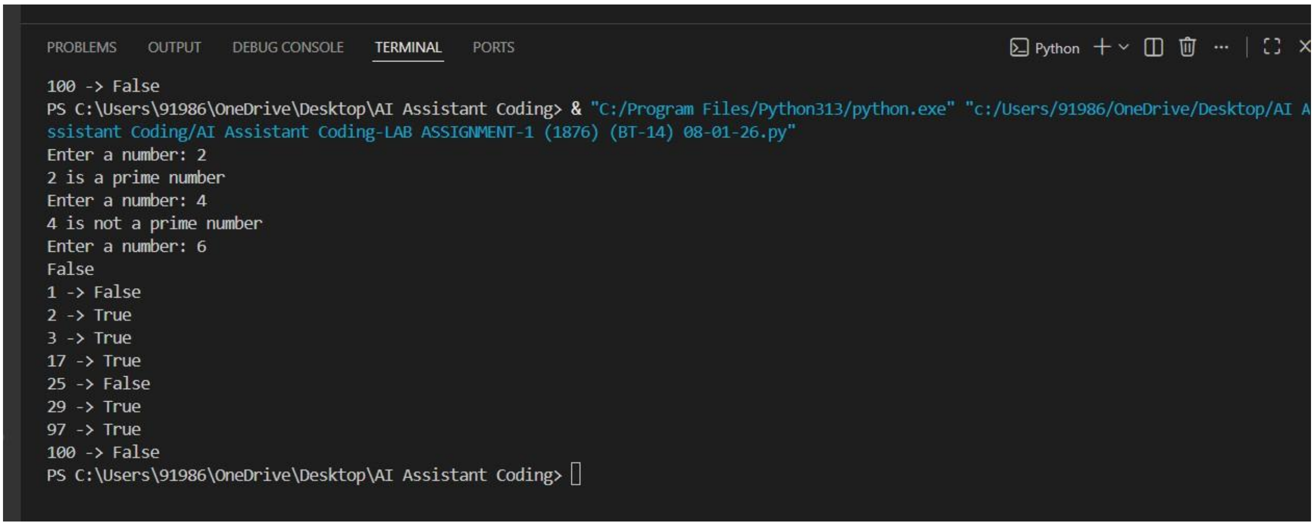
Code:

```
# 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:



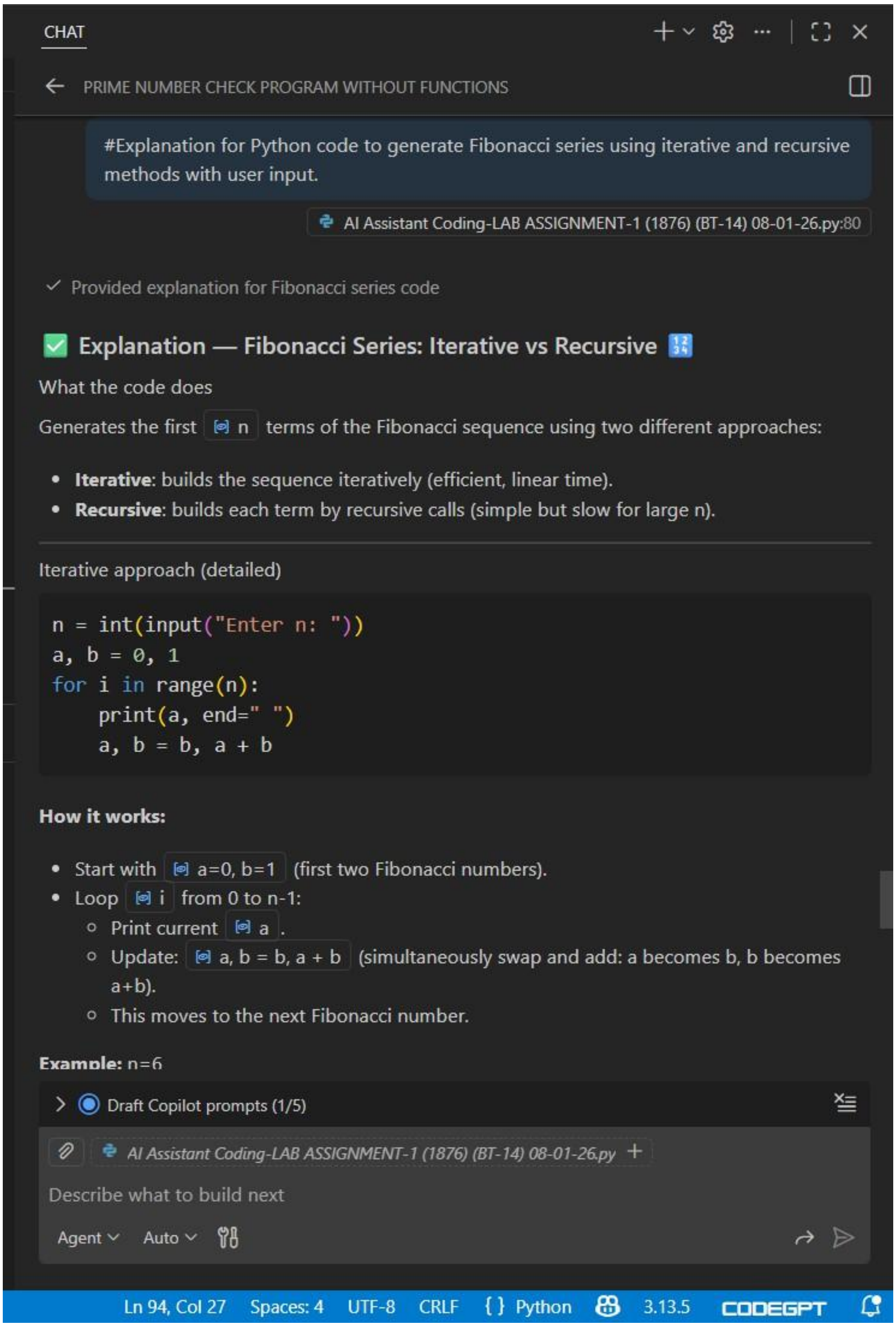
Output:



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:



CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

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

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py:80

✓ Provided explanation for Fibonacci series code

✅ Explanation — Fibonacci Series: Iterative vs Recursive 🔢

What the code does

Generates the first `n` terms of the Fibonacci sequence using two different approaches:

- **Iterative**: builds the sequence iteratively (efficient, linear time).
- **Recursive**: builds each term by recursive calls (simple but slow for large n).

Iterative approach (detailed)

```
n = int(input("Enter n: "))
a, b = 0, 1
for i in range(n):
    print(a, end=" ")
    a, b = b, a + b
```

How it works:

- Start with `a=0, b=1` (first two Fibonacci numbers).
- Loop `i` from 0 to n-1:
  - Print current `a`.
  - Update: `a, b = b, a + b` (simultaneously swap and add: a becomes b, b becomes a+b).
  - This moves to the next Fibonacci number.

Example: n=6

Draft Copilot prompts (1/5)

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py

Describe what to build next

Agent Auto

Ln 94, Col 27 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

Example: n=6

- i=0: print 0, a=1, b=1
- i=1: print 1, a=1, b=2
- i=2: print 1, a=2, b=3
- i=3: print 2, a=3, b=5
- i=4: print 3, a=5, b=8
- i=5: print 5, a=8, b=13
- Output: `0 1 1 2 3 5`

Time complexity: O(n) — one loop, each iteration is O(1).
Space complexity: O(1) — only two variables.

Recursive approach (detailed)

```python
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=" ")
```

How it works:

- Base case: if n ≤ 1, return n (so fib(0)=0, fib(1)=1).
- Recursive case: fib(n) = fib(n-1) + fib(n-2).
- The loop calls fib(0), fib(1), ..., fib(n-1) and prints each result.

Example: n=6, calling fib(0)..fib(5)

- fib(0) → 0
- fib(1) → 1
- fib(2) → fib(1) + fib(0) = 1

Draft Copilot prompts (1/5)

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py

Describe what to build next

Agent Auto

Ln 94, Col 27 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

- fib(0) → 0
- fib(1) → 1
- fib(2) → fib(1) + fib(0) = 1
- fib(3) → fib(2) + fib(1) = 2
- fib(4) → fib(3) + fib(2) = 3
- fib(5) → fib(4) + fib(3) = 5
- Output: `0 1 1 2 3 5`

Time complexity: O(2^n) — each call branches into two recursive calls; massive redundant computation (fib(3) is called multiple times).
Space complexity: O(n) — recursion depth (call stack).

Comparison table

Aspect	Iterative	Recursive
Time	O(n)	O(2^n)
Space	O(1)	O(n) — call stack
Clarity	Direct, straightforward.	Matches mathematical definition elegantly.
Scalability	Excellent; handles n ~ 10^6 easily.	Poor; n > 30–40 becomes impractical.
Redundancy	None; each value computed once.	Massive; fib(3) computed hundreds of times.
When to use	Always, unless n is tiny (< 20).	Educational demo only.

Sample runs

Input: `5`

- Iterative: `0 1 1 2 3`

Draft Copilot prompts (1/5)

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

Describe what to build next

Agent Auto

Ln 94, Col 27 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 CODEGPT

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

Clarity	Direct, straightforward.	Matches mathematical definition elegantly.
Scalability	Excellent; handles n ~ 10^6 easily.	Poor; n > 30–40 becomes impractical.
Redundancy	None; each value computed once.	Massive; fib(3) computed hundreds of times.
When to use	Always, unless n is tiny (< 20).	Educational demo only.

Sample runs

Input: `5`

- Iterative: `0 1 1 2 3`
- Recursive: `0 1 1 2 3`

Input: `8`

- Iterative: `0 1 1 2 3 5 8 13`
- Recursive: `0 1 1 2 3 5 8 13`

Key takeaways ✨

- **Iterative** is the practical choice: O(n) time, O(1) space, handles large n.
- **Recursive** is conceptually clean but computationally expensive; use for learning or tiny n only.
- For production code or competitive programming, always prefer **iterative** (or use memoization with recursion).

Would you like me to add a **memoized recursive version** (caches results to avoid redundant computation) or a **performance comparison script** that times both approaches? 🔧

Claude Haiku 4.5 • 1x

Draft Copilot prompts (1/5)

AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +

Describe what to build next

Agent Auto

Task-5