

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

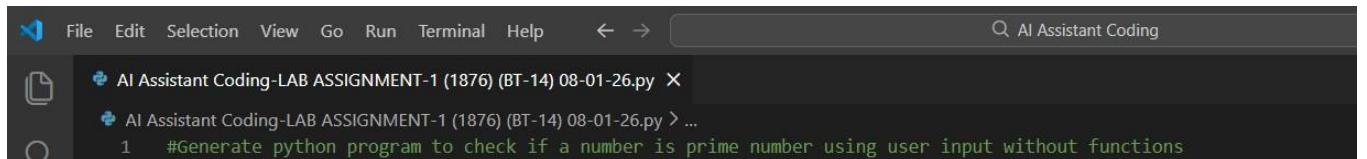
❖ Expected Output

- 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:



The screenshot shows a code editor window titled "AI Assistant Coding". The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. There are two tabs open: "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py" and another tab that is partially visible. The code in the main editor window is:

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

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:

A screenshot of a code editor window titled "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py". The code is as follows:

```
1 #Generate python program to check if a number is prime number using user input without functions
2 num = int(input("Enter a number: "))
3 if num > 1:
4     for i in range(2, int(num**0.5) + 1):
5         if (num % i) == 0:
6             print(num, "is not a prime number")
7             break
8         else:
9             print(num, "is a prime number")
10    else:
11        print(num, "is not a prime number")
```

Output:

A screenshot of a terminal window titled "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py". The terminal shows the following interaction:

```
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: 17
17 is a prime number
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: 25
25 is not a prime number
```

Explanation:

- t reads and validates the user's input (rejects non-integers).
- For $n > 1$, testing divisors only up to $\text{floor}(\sqrt{n})$ is sufficient because if $n = a * b$ and $a \leq b$ then $a \leq \sqrt{n}$.
- 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.

The screenshot shows a user interface for generating Python code. At the top, there is a header bar with the title "← PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS". Below this is a dark-themed code editor window. In the top right corner of the editor, there is a small AI icon and the text "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py:1". The main area of the editor contains the generated Python code:

```
# 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")
```

At the bottom of the editor, there is a status bar with the text "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py +". Below the editor, there is a text input field with the placeholder "Describe what to build next". At the very bottom, there are several small icons and status indicators, including "Agent", "Auto", "Spaces: 4", "UTF-8", "CRLF", "Python 3.13.5", and "CODEGPT".

CHAT + ⌂ ⚙️ ⋮ | ☰ X

← PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS ⏷

```
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 $\text{floor}(\sqrt{n})$ is sufficient because if $n = a * b$ and $a \leq b$ then $a \leq \sqrt{n}$.
- 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.

C ⌂ ⌛ ⌚ 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

The screenshot shows the AI Assistant Coding interface with the following details:

- Left Panel:** Shows the code editor with a file named "ASSIGNMENT-1 TASK-2(1876).py". The code checks if a number is prime without using functions.
- Terminal:** Displays the command line output of running the script with inputs 17 and 25, showing that 17 is prime and 25 is not.
- Right Panel:**
 - CHAT:** A conversation with AI about generating a simple Python program for prime number check.
 - Generated code (no functions):** A copy of the original code.
 - Code Completion:** An AI-generated completion of the code, adding comments and a try-except block.
 - Sample Inputs / Outputs:** A table showing sample inputs and their outputs.

The screenshot shows the AI Assistant Coding interface with the following details:

- Left Panel:** Shows the code editor with the same prime number check script.
- Terminal:** Displays the command line output of running the script with inputs 17 and 25.
- Right Panel:**
 - CHAT:** A conversation with AI about generating a simple Python program for prime number check.
 - Generated code (no functions):** A copy of the original code.
 - Code Completion:** An AI-generated completion of the code, adding comments and a try-except block.
 - Sample Inputs / Outputs:** A table showing sample inputs and their outputs.
 - Why the code works:** A section explaining the logic of the prime check algorithm.
 - Tip:** A note suggesting to use specialized algorithms or caching for large inputs.

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:



The screenshot shows a code editor interface with the following details:

- Top Bar:** Contains "File", "Edit", "Selection", "View", "Go", "Run", "Terminal", and "Help" buttons.
- Search Bar:** A magnifying glass icon followed by "AI Assistant Coding".
- Tab Bar:** Shows three tabs:
 - "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py" (active tab)
 - "ASSIGNMENT-1 TASK-2(1876).py"
 - "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py > ..."
- Bottom Status Bar:** Displays the number "13" and the text "#Optimize prime number checking logic using user input without functions, reduce iterations and improve readability".

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:

The screenshot shows a code editor interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Toolbar:** Includes icons for Undo, Redo, Selection, View, Run, Terminal, Help, and a CodeGPT icon.
- Search Bar:** Q AI Assistant Coding.
- Tab Bar:** AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py (active), ASSIGNMENT-1 TASK-2(1876).py, and AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py > ...
- Code Area:** Displays Python code for prime number checking. The code uses a combination of direct print statements and a while loop with break to determine if a number is prime or not.

```
13 #Optimize prime number checking logic using user input without functions, reduce iterations and improve readability
14 num = int(input("Enter a number: "))
15 if num <= 1:
16     print(num, "is not a prime number")
17 elif num <= 3:
18     print(num, "is a prime number")
19 elif num % 2 == 0 or num % 3 == 0:
20     print(num, "is not a prime number")
21 else:
22     i = 5
23     while i * i <= num:
24         if num % i == 0 or num % (i + 2) == 0:
25             print(num, "is not a prime number")
26             break
27         i += 6
28     else:
29         print(num, "is a prime number")
```

Output:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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: 17

17 is a prime number

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: 25

25 is not a prime number

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: 5

5 is a prime number

Enter a number: 18

18 is not a prime number

PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding>

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:

The screenshot shows a code editor interface with a dark theme. At the top, there's a header bar with 'CHAT' and various icons. Below it, a title bar says 'PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS'. The main area contains a code snippet and an AI-generated explanatory text.

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

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 $6k \pm 1$ 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 \sqrt{n} . This reduces trial divisions and keeps code simple and readable.

💡 **What changed (concise)**

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

💡 **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 $\equiv \pm 1 \pmod{6}$.
- Therefore any prime > 3 must be of the form $6k \pm 1$; after removing 2 and 3, it's sufficient to test only those candidates.
- Compared to checking every integer from $2.. \sqrt{n}$, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by $\sim 1.5 \times$.

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 $\equiv \pm 1 \pmod{6}$.
- Therefore any prime > 3 must be of the form $6k \pm 1$; after removing 2 and 3, it's sufficient to test only those candidates.
- Compared to checking every integer from $2..\sqrt{n}$, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by $\sim 1.5\times$.

⌚ Complexity

- Time: $O(\sqrt{n})$ (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 $6k\pm 1$ 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. ↴

C ⌂ ⌄ ⌁ ⌂ 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

The screenshot shows a Jupyter Notebook environment with two code cells. The first cell contains Python code for a simple prime check using nested loops and conditionals. The second cell contains optimized code using modular arithmetic to reduce iterations. A sidebar panel titled 'PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS' provides explanatory text and highlights the optimized code.

```
File Edit Selection View Go Run Terminal Help → Q AI Assistant Coding
```

```
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py • ASSIGNMENT-1 TASK-2(1876).py ... CHAT
```

```
#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")
```

```
#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
```

```
Python + × ⚡ ... | ☰ ×
```

```
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding> & "C:/Program Files/Python313/python.exe" "c:/Users/91986/OneDrive/Desktop/AT Assistant Coding/AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py"
Enter a number: 25
25 is not a prime number
PS C:\Users\91986\OneDrive\Desktop\AI Assistant Coding> & "C:/Program Files/Python313/python.exe" "c:/Users/91986/OneDrive/Desktop/AT Assistant Coding/AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py"
Enter a number: 5
5 is a prime number
Enter a number: 18
18 is not a prime number
```

```
← PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS
```

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

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

Planned and structured final response explanation

I'm going to provide a concise, structured explanation of the $6k \pm 1$ 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 \sqrt{n} . This reduces trial divisions and keeps code simple and readable.

What changed (concise)

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

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 $\equiv \pm 1 \pmod{6}$.
- Therefore any prime > 3 must be of the form $6k \pm 1$; after removing 2 and 3, it's sufficient to test only those candidates.
- Compared to checking every integer from $2, \sqrt{n}$, we cut the candidate pool by about a factor of 3; compared to checking only odd numbers, we still reduce candidates by $\sim 1.5x$.

Complexity

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

Describe what to build next

Agent Auto 98

In 20, Col 40 Spaces: 4 UTF-8 CRLF I Python 3.13.5 CODEIGHT

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)

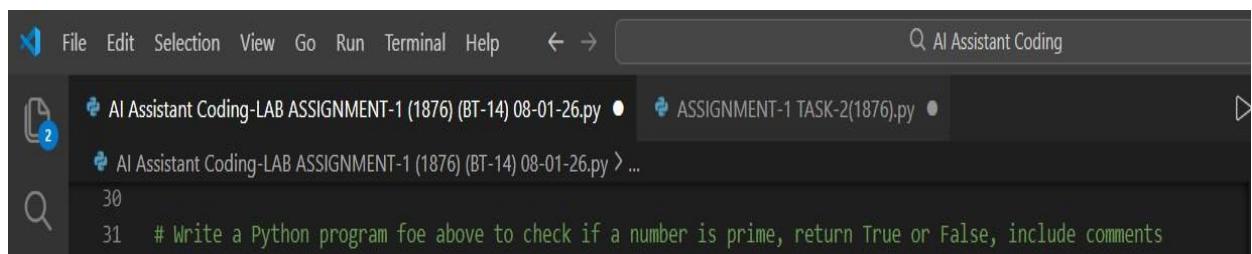
❖ Expected Output

- 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:



```
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py
ASSIGNMENT-1 TASK-2(1876).py

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

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:

The screenshot shows a code editor interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar labeled "AI Assistant Coding". Below the menu, there are three tabs: "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py", "ASSIGNMENT-1 TASK-3(1876).py", and "ASSIGNMENT-1 TASK-2(1876).py". The main code area contains the following Python script:

```
31 # Write a Python program for above to check if a number is prime, return True or False, include comments
32 num = int(input("Enter a number: "))
33 # Check if the number is less than or equal to 1
34 if num <= 1:
35     print(False) # Numbers less than or equal to 1 are not prime
36 elif num <= 3:
37     print(True) # 2 and 3 are prime numbers
38 elif num % 2 == 0 or num % 3 == 0:
39     print(False) # Eliminate multiples of 2 and 3
40 else:
41     i = 5
42     is_prime = True # Assume the number is prime
43     while i * i <= num:
44         if num % i == 0 or num % (i + 2) == 0:
45             is_prime = False # Found a divisor, not prime
46             break
47         i += 6
48     print(is_prime) # Output the result
49
50
```

Output:

The screenshot shows a terminal window with a dark theme. The top navigation bar includes PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. There is also a Python interpreter icon and some other icons. The terminal window displays the following text:

```
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: 14
14 is not a prime number
Enter a number: 18
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>
```

The status bar at the bottom right shows "Ln 48, Col 41".

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:

The screenshot shows a user interface for an AI Assistant Coding-LAB. At the top, there's a header bar with 'CHAT' and various icons. Below it, a title bar says 'PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS'. The main area contains a question: 'give a Explanation of for above to check if a number is prime, return True or False, include comments'. Below the question, a file path 'AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py:31' is shown. A green checkmark indicates a successful explanation: 'Explanation — prime check that prints True/False'. The explanation text is as follows:

How it works (step-by-step)

- Input:** `num = int(input("Enter a number: "))` — the snippet assumes a valid integer is entered (wrap in `try/except` if you want robust validation).
- Handle small cases:**
 - `if num <= 1: print(False)` → 0, 1, negatives are not prime.
 - `elif num <= 3: print(True)` → 2 and 3 are prime.
- Eliminate easy multiples:**
 - `elif num % 2 == 0 or num % 3 == 0: print(False)` → remove even numbers and multiples of 3 quickly.
- 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`.
 - Reason: every prime > 3 is of the form $6k \pm 1$, 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 \sqrt{n} : you short-circuit many cases (2,3 and their multiples) and test only candidates $\equiv \pm 1 \pmod{6}$.
- Time complexity: $O(\sqrt{n})$ with a **smaller constant factor** than naive checks.
- Space complexity: $O(1)$.

At the bottom, there are buttons for 'Agent', 'Auto', and 'Run', along with a status bar showing 'Ln 48, Col 41' and 'CODEGPT'.

CHAT

← PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

```
.. 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 ($6k \pm 1$ 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 $6k \pm 1$, 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 \sqrt{n} : you short-circuit many cases (2,3 and their multiples) and test only candidates $\equiv \pm 1 \pmod{6}$.
- Time complexity: $O(\sqrt{n})$ 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 $6k \pm 1$ 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? ⚡

C ↴ ⌟ ⌛ 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

The screenshot shows the AI Assistant Coding interface with two tabs open: "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py" and "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py". The code is a Python script for checking if a number is prime. It includes comments explaining the logic, such as handling small cases, eliminating easy multiples, and applying the 6k±1 rule. The terminal below shows the execution of the script, testing various numbers from 2 to 100. The interface also displays AI-generated notes and examples.

```
31 # Write a Python program for above to check if a number is prime, return True or False, include comments
32 num = int(input("Enter a number: "))
33 # check if the number is less than or equal to 1
34 if num <= 1:
35     print(False) # Numbers less than or equal to 1 are not prime
36 elif num <= 3:
37     print(True) # 2 and 3 are prime numbers
38 elif num % 2 == 0 or num % 3 == 0:
39     print(False) # Eliminate multiples of 2 and 3
40 else:
41     i = 5
42     is_prime = True # Assume the number is prime
43     while i * i <= num:
44         if num % i == 0 or num % (i + 2) == 0:
45             is_prime = False # Found a divisor, not prime
46             break
47         i += 6
48     print(is_prime) # Output the result
49
50
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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: 14
14 is not a prime number
Enter a number: 18
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>

CHAT

← PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

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 $6k \pm 1$, 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 \sqrt{n} ; you short-circuit many cases (2,3 and their multiples) and test only candidates $\equiv 1 \pmod{6}$.
- Time complexity: $O(\sqrt{n})$ 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? ✎

Describe what to build next

Agent Auto ↻ 3/6

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

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:

```

File Edit Selection View Go Run Terminal Help ← → Q AI Assistant Coding
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py ● ASSIGNMENT-1 TASK-3(1876).py ASSIGNMI ▶
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py > ...
49
50 #generate a code for Comparative Analysis –With vs Without Functions
51 # Without Functions

```

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

```

```

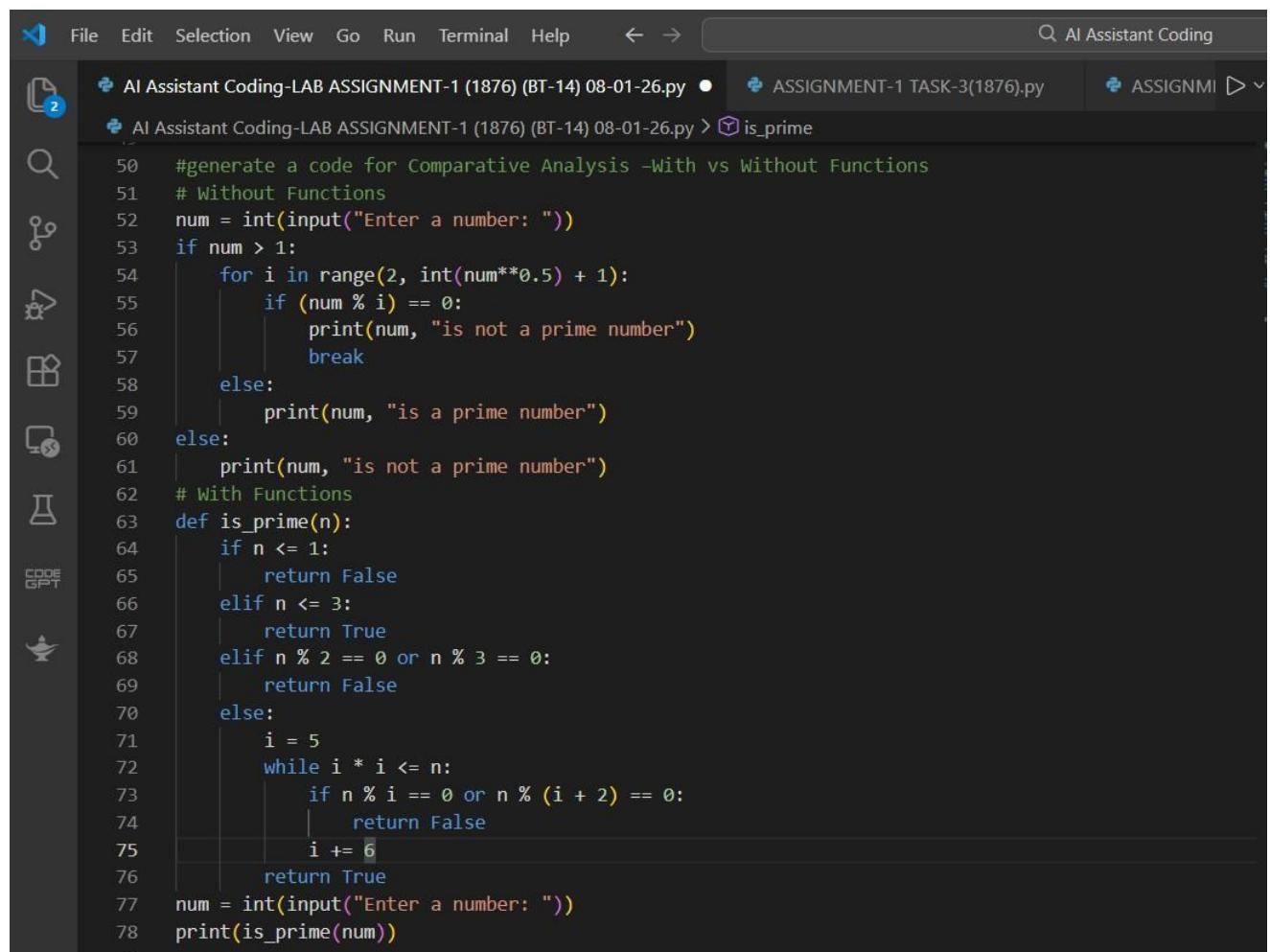
else:
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

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

print(is_prime(num))

```

Screenshot:



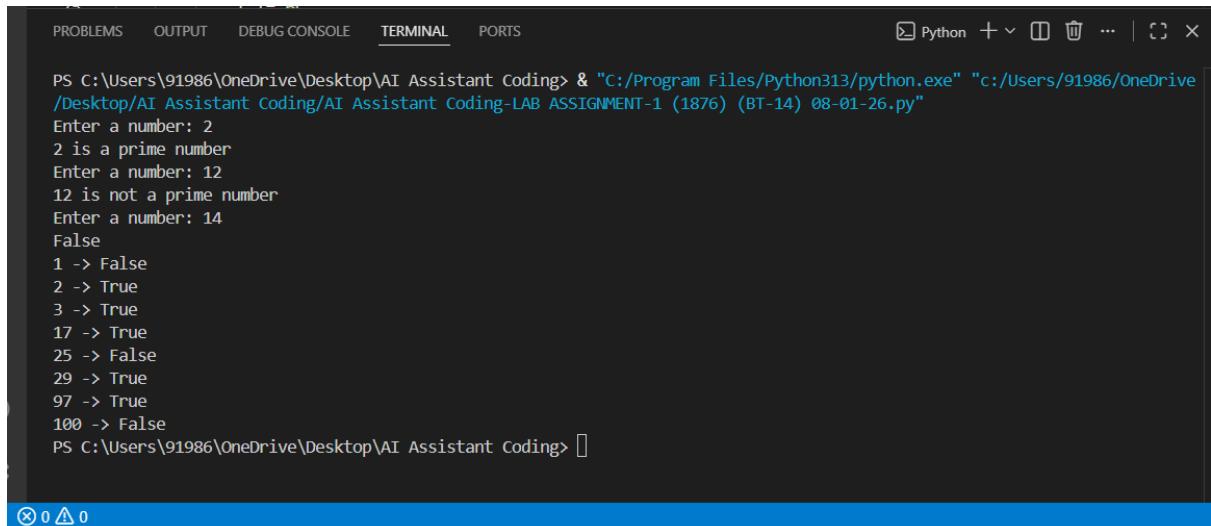
The screenshot shows a code editor interface with a dark theme. The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar labeled "AI Assistant Coding". The main workspace displays two tabs: "AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py" and "ASSIGNMENT-1 TASK-3(1876).py". The code in the active tab is as follows:

```

#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:
            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))

```

Output:



A screenshot of a terminal window from a code editor. The window has tabs at the top: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS. On the right side of the window are icons for Python, a plus sign, a dropdown arrow, a trash can, three dots, a copy/paste icon, and a close button. The main area of the terminal shows the following text:

```
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>
```

The status bar at the bottom of the terminal window shows the count of changes (0) and additions (0).

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 explanation.

✓ 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. <input checked="" type="checkbox"/>
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. <input checked="" type="checkbox"/>
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. <input checked="" type="checkbox"/>
When	Quick classroom examples or single-use scripts.	Production code, libraries, code you'll reuse or test. <input checked="" type="checkbox"/>

Agent ▾ Auto ▾ 🔍

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

When appropriate	Quick classroom examples or single-use scripts.	Production code, libraries, code you'll reuse or test. <input checked="" type="checkbox"/>
------------------	---	--

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

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-3(1876).py ASSIG D x □ ...

generate a code For Comparative Analysis -With vs Without Functions

```

50 num = int(input("Enter a number: "))
51 if num > 1:
52     for i in range(2, int(num**0.5) + 1):
53         if (num % i) == 0:
54             print(num, "is not a prime number")
55             break
56         else:
57             print(num, "is a prime number")
58     else:
59         print(num, "is not a prime number")
60 # with Functions
61 def is_prime(n):
62     if n <= 1:
63         return False
64     elif n <= 3:
65         return True
66     elif n % 2 == 0 or n % 3 == 0:
67         return False
68     else:
69         i = 5
70         while i * i <= n:
71             if n % i == 0 or n % (i + 2) == 0:
72                 return False
73             i += 6
74         return True
75 num = int(input("Enter a number: "))
76 print(is_prime(num))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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>

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

Let me compare the two versions focusing on clarity, reusability, debugging, and scalability (short, practical point). 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.

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 CODESPT

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-3(1876).py ASSIG D x □ ...

generate a code For Comparative Analysis -With vs Without Functions

```

50 num = int(input("Enter a number: "))
51 if num > 1:
52     for i in range(2, int(num**0.5) + 1):
53         if (num % i) == 0:
54             print(num, "is not a prime number")
55             break
56         else:
57             print(num, "is a prime number")
58     else:
59         print(num, "is not a prime number")
60 # with Functions
61 def is_prime(n):
62     if n <= 1:
63         return False
64     elif n <= 3:
65         return True
66     elif n % 2 == 0 or n % 3 == 0:
67         return False
68     else:
69         i = 5
70         while i * i <= n:
71             if n % i == 0 or n % (i + 2) == 0:
72                 return False
73             i += 6
74         return True
75 num = int(input("Enter a number: "))
76 print(is_prime(num))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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>

CHAT

PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS

Let me compare the two versions focusing on 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 80, Col 1 Spaces: 4 UTF-8 CRLF { } Python 🐍 3.13.5 CODESPT

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 \sqrt{n})

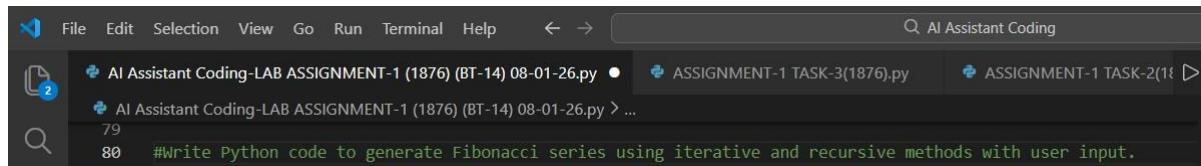
❖ Expected Output

- 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:



A screenshot of a code editor window titled "AI Assistant Coding". The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar. The status bar shows "Q AI Assistant Coding". The code editor displays a Python script with the following content:

```
File Edit Selection View Go Run Terminal Help ← → Q AI Assistant Coding
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py ● ASSIGNMENT-1 TASK-3(1876).py ▶
AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py > ...
79
80 #Write Python code to generate Fibonacci series using iterative and recursive methods with user input.
```

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:

A screenshot of a code editor interface. At the top, there's a menu bar with File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar labeled 'AI Assistant Coding'. Below the menu, there are several tabs: 'AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py' (selected), 'ASSIGNMENT-1 TASK-3(1876).py', and 'ASSIGNMENT-1 TASK-2(1876).py'. On the left side, there are icons for search, file operations, and other tools. The main area contains Python code for generating Fibonacci series using iterative and recursive methods.

```
80 #Write Python code to generate Fibonacci series using iterative and recursive methods with user input.
81 # Iterative Fibonacci
82 n = int(input("Enter n: "))
83 a, b = 0, 1
84 for i in range(n):
85     print(a, end=" ")
86     a, b = b, a + b
87
88 # Recursive Fibonacci
89 def fib(n):
90     return n if n <= 1 else fib(n-1) + fib(n-2)
91
92 print("\nRecursive:", end=" ")
93 for i in range(n):
94     print(fib(i), end=" ")
```

Output:

A screenshot of a terminal window. The tab bar at the top includes PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. The terminal shows the command 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". The output of the script is displayed below, showing prime number checks for various numbers.

```
100 -> False
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: 4
4 is not a prime number
Enter a number: 6
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:

The basic approach checks divisibility up to $N-1$, resulting in unnecessary iterations and higher time complexity. The optimized approach checks only up to \sqrt{N} because any factor larger than \sqrt{N} must have a corresponding smaller factor.

Explanation:

The screenshot shows a code editor interface with the following details:

- CHAT** tab is selected at the top.
- The title bar says "PRIME NUMBER CHECK PROGRAM WITHOUT FUNCTIONS".
- The main area contains the following text:

```
#Explanation for Python code to generate Fibonacci series using iterative and recursive methods with user input.
```
- A status bar at the bottom indicates "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 + ⌂ ⌂ ⌂ | ⌂ X

← 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)

```
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 $n \leq 1$, return n (so $\text{fib}(0)=0$, $\text{fib}(1)=1$).
- Recursive case: $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$.
- The loop calls $\text{fib}(0)$, $\text{fib}(1)$, ..., $\text{fib}(n-1)$ and prints each result.

Example: n=6, calling $\text{fib}(0)..\text{fib}(5)$

- $\text{fib}(0) \rightarrow 0$
- $\text{fib}(1) \rightarrow 1$
- $\text{fib}(2) \rightarrow \text{fib}(1) + \text{fib}(0) = 1$

> 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 $n \sim 10^6$ 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) X≡

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 $n \sim 10^6$ easily.	Poor; $n > 30-40$ becomes impractical.
Redundancy	None; each value computed once.	Massive; $\text{fib}(3)$ 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? ✍

C ↪ ⌘ ⌘

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

The screenshot shows the AI Assistant Coding interface. The main area displays two Python files: `AI Assistant Coding-LAB ASSIGNMENT-1 (1876) (BT-14) 08-01-26.py` and `ASSIGNMENT-1 TASK-3(1876).py`. The left panel contains the code for generating Fibonacci series using iterative and recursive methods. The right panel provides an analysis of the code's clarity, scalability, redundancy, and when to use it, comparing iterative and recursive approaches. The terminal below shows the execution of the code, demonstrating its functionality. A sidebar on the right includes a 'Key takeaways' section with tips about iterative vs recursive programming.

```
79
80 #Write Python code to generate Fibonacci series using iterative and recursive methods with user
81 # Iterative Fibonacci
82 n = int(input("Enter n: "))
83 a, b = 0, 1
84 for i in range(n):
85     print(a, end=" ")
86     a, b = b, a + b
87
88 # Recursive Fibonacci
89 def fib(n):
90     return n if n <= 1 else fib(n-1) + fib(n-2)
91
92 print("\nRecursive:", end=" ")
93 for i in range(n):
94     print(fib(i), end=" ")
```

Clarity	Direct, straightforward.	Matches mathematical definition elegantly.
Scalability	Excellent; handles $n \sim 10^6$ easily.	Poor; $n > 30-40$ becomes impractical.
Redundancy	None; each value computed once.	Massive; <code>fib(3)</code> 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? ↗

Draft Copilot prompts (1/5)

Describe what to build next

Agent Agent Auto Auto