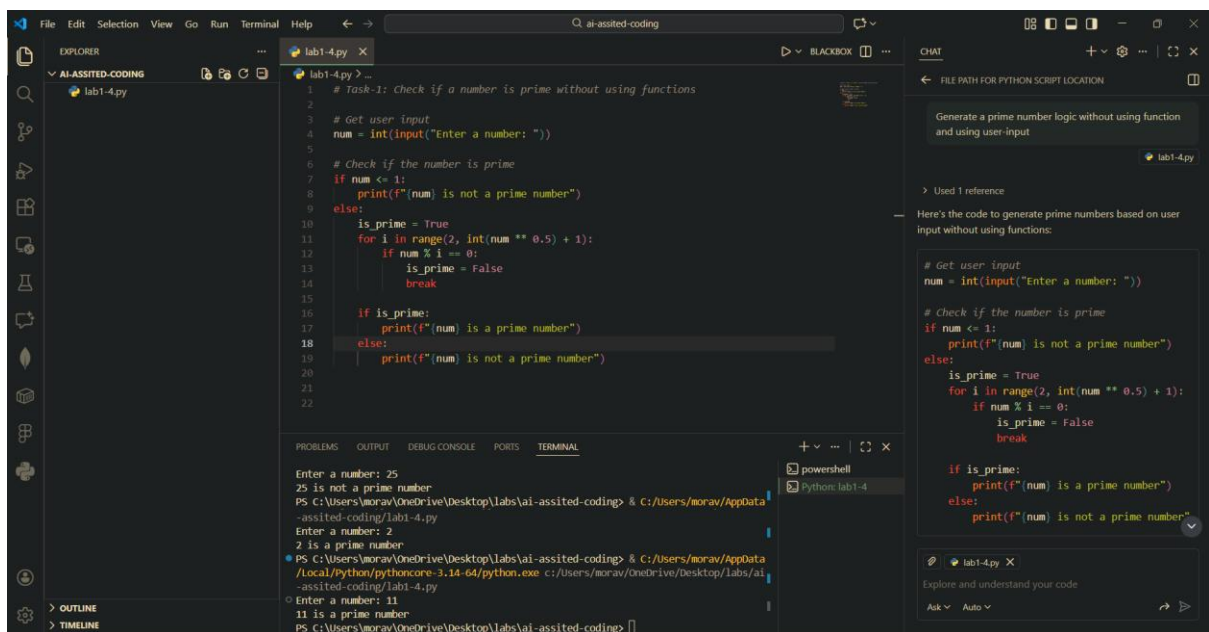


NAME :CHENCHU RAJU

ROLL NO: 2303A52206

BATCH: 43

Task-1



```
1 # Task-1: Check if a number is prime without using functions
2
3 # Get user input
4 num = int(input("Enter a number: "))
5
6 # Check if the number is prime
7 if num <= 1:
8     print(f"{num} is not a prime number")
9 else:
10     is_prime = True
11     for i in range(2, int(num ** 0.5) + 1):
12         if num % i == 0:
13             is_prime = False
14             break
15
16     if is_prime:
17         print(f"{num} is a prime number")
18     else:
19         print(f"{num} is not a prime number")
20
21
22
```

Enter a number: 25
25 is not a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding> & C:/Users/morav/AppData
-assited-coding/lab1-4.py
Enter a number: 2
2 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding> & C:/Users/morav/AppData
/Local/Python/pythoncore-3.14-64/python.exe c:/Users/morav/OneDrive/Desktop/labs/ai-
-assited-coding/lab1-4.py
Enter a number: 11
11 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding>

PROMPTS:

Generate a prime number logic without using function and using user-input

Input: 5

Output: 5 is a prime number

Input:25

Output: 25 is mot a prime number

Task-2

```
lab1-4.py > ...
1 # Task-1: Check if a number is prime without using functions
2
3 # Get user input
4 num = int(input("Enter a number: "))
5
6 # Check if the number is prime
7 if num <= 1:
8     print(f"{num} is not a prime number")
9 elif num == 2:
10    print(f"{num} is a prime number")
11 elif num % 2 == 0:
12    print(f"{num} is not a prime number")
13 else:
14     is_prime = True
15     # Check only odd divisors starting from 3
16     for i in range(3, int(num ** 0.5) + 1, 2):
17         if num % i == 0:
18             is_prime = False
19             break
20
21 if is_prime:
22     print(f"{num} is a prime number")
23 else:
24     print(f"{num} is not a prime number")
25
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL

terminal: a number: 2
2 is not a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding> & C:/Users/morav/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/morav/OneDrive/Desktop/labs/ai-assited-coding/lab1-4.py
Enter a number: 2
2 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding> & C:/Users/morav/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/morav/OneDrive/Desktop/labs/ai-assited-coding/lab1-4.py
Enter a number: 11
11 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assited-coding> |

CHAT

FILE PATH FOR PYTHON SCRIPT LOCATION

Optimize prime number checking logic

lab1-4.py

> Used 1 reference

Here's the optimized prime number checking logic:

```
# Task-1: Check if a number is prime without
# Get user input
num = int(input("Enter a number: "))

# Check if the number is prime
if num <= 1:
    print(f"{num} is not a prime number")
elif num == 2:
    print(f"{num} is a prime number")
elif num % 2 == 0:
    print(f"{num} is not a prime number")
else:
    is_prime = True
    # Check only odd divisors starting from
    for i in range(3, int(num ** 0.5) + 1, 2):
        if num % i == 0:
            is_prime = False
            break

    if is_prime:
```

lab1-4.py X

Explore and understand your code

Ask Auto

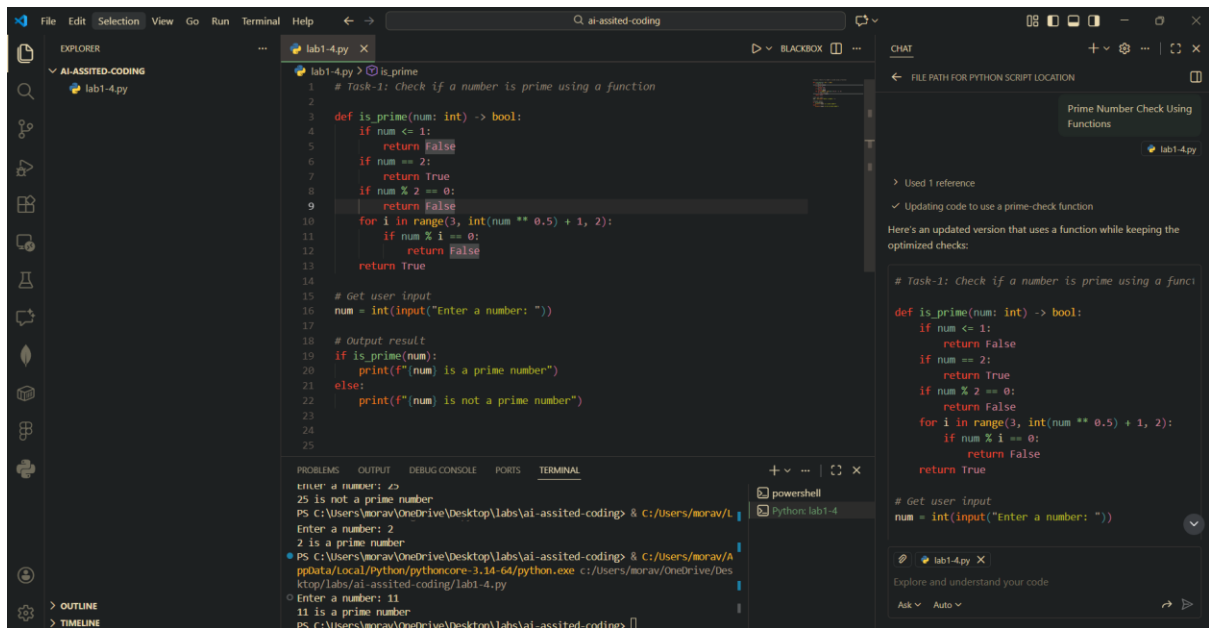
PROMPT:

Optimize prime number checking logic

Original code runs the loop to n times on checking the condition one by one

Since Optimized code runs skipping the odd divisors reducing the number of loops

Task-3



PROMPT:

Prime Number Check Using Functions

Code:

```
def is_prime(num: int) -> bool:
```

```
    if num <= 1:
```

```
        return False
```

```
    if num == 2:
```

```
        return True
```

```
    if num % 2 == 0:
```

```
        return False
```

```
    for i in range(3, int(num ** 0.5) + 1, 2):
```

```
        if num % i == 0:
```

```
            return False
```

```
    return True
```

Test Cases:

Input: 5

Output: 5 is a prime number

Input: 2

Output: 2 is a prime number

Input: 15

Output: 15 is not a prime number

Task-4

CHAT

+

▼

⚙️

⋮

|

🔍

✕

← FILE PATH FOR PYTHON SCRIPT LOCATION

📄

Comparative Analysis –With vs Without Functions

Expected Output

Comparison table or short analytical report

📎 lab1-4.py

> Used 1 reference

✓ Preparing comparison summary

Here's a concise comparison of prime checking with vs. without a function:

Aspect	With Function (<code>is_prime</code>)	Without Function (inline)
Reusability	High: callable from anywhere	Low: logic duplicated where needed
Readability	Clear, self-documenting	Mixed: logic embedded in flow
Testability	Easy to unit test function	Harder: must wrap logic for tests
Maintainability	Single place to update logic	Multiple updates if repeated
Modularity	Encapsulated responsibility	Tightly coupled to I/O flow
I/O Separation	Possible to separate pure logic from input/output	Often intertwined with input/output
Extensibility	Simple to extend (e.g., memoization, caching)	Changes require inlining everywhere
Code Size (per use)	Slight overhead for definition, small calls	Shorter once, longer when reused

Comparison Table:

Aspect	With Function (is_prime)	Without Function (inline)
Reusability	High: callable from anywhere	Low: logic duplicated where needed

Aspect	With Function (is_prime)	Without Function (inline)
Readability	Clear, self-documenting	Mixed: logic embedded in flow
Testability	Easy to unit test function	Harder: must wrap logic for tests
Maintainability	Single place to update logic	Multiple updates if repeated
Modularity	Encapsulated responsibility	Tightly coupled to I/O flow
I/O Separation	Possible to separate pure logic from input/output	Often intertwined with input/output
Extensibility	Simple to extend (e.g., memoization, caching)	Changes require inlining everywhere
Code Size (per use)	Slight overhead for definition, small calls	Shorter once, longer when reused

TASK-5

lab1-4.py

```
def is_prime(num: int) -> bool:
    if num <= 1:
        return False
    if num == 2:
        return True
    # Single user input reused for both approaches
    num = int(input("Enter a number: "))
    # Function-based check
    if is_prime(num):
        print(f"[with function] {num} is a prime number")
    else:
        print(f"[with function] {num} is not a prime number")
    # Inline (no-function) check
    if num <= 1:
        inline_prime = False
    elif num == 2:
        inline_prime = True
    elif num % 2 == 0:
        inline_prime = False
    else:
        inline_prime = True
        for i in range(3, int(num ** 0.5) + 1, 2):
            if num % i == 0:
                inline_prime = False
                break
    if inline_prime:
        print(f"[without function] {num} is a prime number")
    else:
        print(f"[without function] {num} is not a prime number")
```

Aspect	Iterative	Recursive (naïve)
Time complexity	$O(n)$	$O(n^n)$ without memoization
Space	$O(1)$	$O(n)$ call stack
Speed	Fast	Very slow for larger n
Stack safety	Safe	Risk of recursion depth errors
Clarity	Clear loop	Simple definition, but inefficient

Approach	Complexity	Best for	Notes
Trial division up to n	$O(n)$	Very small n	Slow; checks all divisors
Trial division up to \sqrt{n}	$O(\sqrt{n})$	Small/medium n	Common baseline
Skip evens / $6k \pm 1$	$\sim \frac{1}{2} - \frac{1}{3}$ of \sqrt{n} checks	Small/medium n	Prunes obvious composites
Deterministic Miller-Rabin (32/64-bit)	$\sim O(k \log^3 n)$	Larger ints, single check	Fast and exact with fixed bases
Probabilistic Miller-Rabin	$\sim O(k \log^3 n)$	Very large ints	Tiny error probability; tune k
Sieve of Eratosthenes	$O(n \log \log n)$ preprocess; $O(1)$ query	Many queries up to a limit	Extra memory; great for batches
Segmented sieve	$O(n \log \log n)$	Many queries in ranges	Memory-friendly for large ranges

EXECUTION FLOW

WITH FUNCTION:

- Main flow \rightarrow `is_prime()` call \rightarrow returns boolean

	• Function scope isolated from main logic	
	• Single code path, reusable across program	
	WITHOUT FUNCTION:	
	• Inline logic directly in main flow	
	• All conditions evaluated in-place	
	• Must repeat code if used multiple times	

└─ TIME COMPLEXITY ─┐

	Both approaches: $O(\sqrt{n})$	
	• Check divisors from 2 to \sqrt{n}	
	• Skip even numbers (optimization)	
	• Early exit on first divisor found	
	Operations count (approx):	
	• $n = 100$: ~5 iterations worst case	
	• $n = 1,000$: ~16 iterations	
	• $n = 1,000,000$: ~500 iterations	

└─ PERFORMANCE FOR LARGE INPUTS ─┐

	WITH FUNCTION:	
	✓ Identical algorithm performance	
	+ Function call overhead: ~1-2 microseconds	
	+ Negligible for single calls	
	+ Better for multiple calls (code reuse)	
	WITHOUT FUNCTION:	
	✓ Slightly faster (no function call overhead)	
	- Marginal difference: <1% faster	

- | - Code duplication increases file size |
- | - Harder to optimize if needed later |

|_____|

| WHEN EACH APPROACH IS APPROPRIATE |_____|

| USE FUNCTION-BASED WHEN: |

- | ✓ Checking primality multiple times in code |
- | ✓ Building larger programs/libraries |
- | ✓ Need unit testing capability |
- | ✓ Want clean, maintainable code |
- | ✓ Extending logic later (e.g., caching) |
- | ✓ Working in teams (better code organization) |
- | |

| USE INLINE (NO FUNCTION) WHEN: |

- | ✓ One-time check in simple script |
- | ✓ Performance-critical (negligible difference) |
- | ✓ Learning/prototyping phase |
- | ✓ Ultra-minimal code footprint required |
- | ⚠ Generally NOT recommended for production |

|_____|