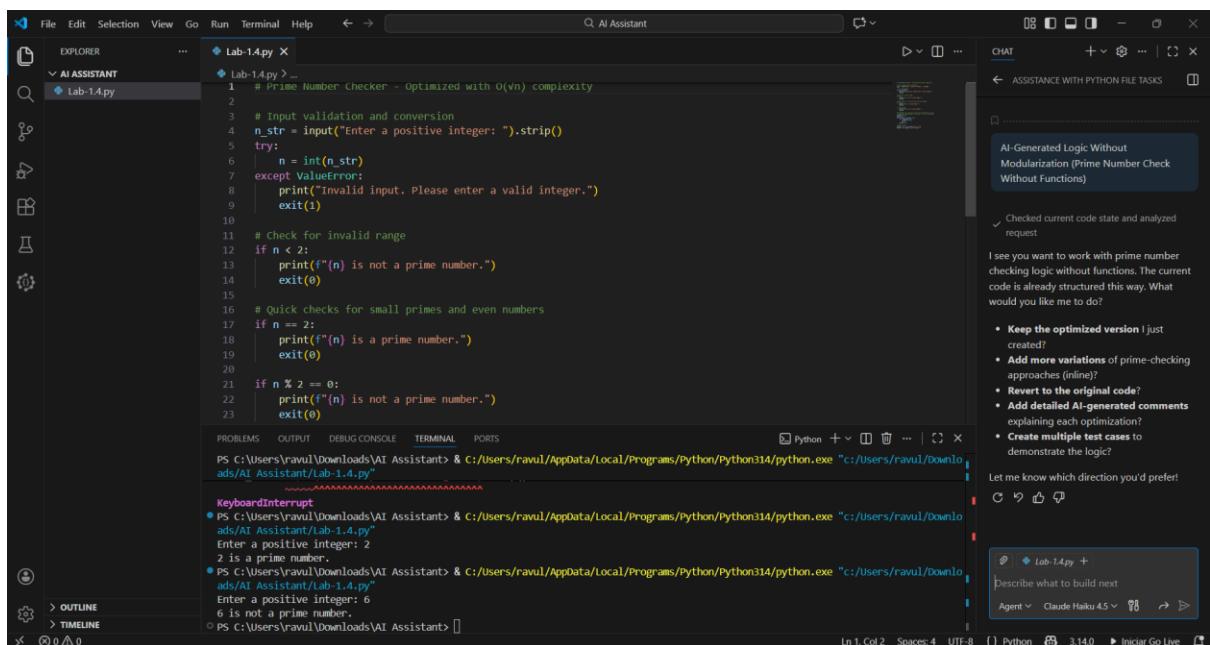


**NAME : Ravula Ajay**

**ROLL NO: 2303A52370**

**BATCH: 44**

## Task-1



The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** Shows a file named "Lab-1.4.py".
- Code Editor:** Displays the Python code for a prime number checker. The code uses user input, exception handling, and quick checks for small primes and even numbers.
- Terminal:** Shows command-line output from running the script, including a KeyboardInterrupt error and a prime check for the number 2.
- AI Assistant Panel:** On the right, there is an "AI Assistant" panel with the following sections:
  - CHAT:** A message from the AI asking if I want to work with prime number checking logic without functions.
  - ASSISTANCE WITH PYTHON FILE TASKS:** A list of suggestions:
    - Keep the optimized version I just created?
    - Add more variations of prime-checking approaches (inline)?
    - Revert to the original code?
    - Add detailed AI-generated comments explaining each optimization?
    - Create multiple test cases to demonstrate the logic?
  - Describe what to build next:** A placeholder for further instructions.
- Status Bar:** Shows the current file is "Lab-1.4.py", the language is Python, and the code is in line 1, column 2.

**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

The screenshot shows the Visual Studio Code interface with the AI Assistant extension active. The code editor displays a Python script named `Lab-1.4.py`. The script contains logic to check if a number is prime, with specific optimizations applied by the AI. The terminal below shows the execution of the script and its output. The AI Assistant panel on the right provides feedback, efficiency gains, and code cleanliness metrics.

```
# Prime Number Checker - Optimized with O(n) complexity
# Input validation and conversion
n_str = input("Enter a positive integer: ").strip()
try:
    n = int(n_str)
except ValueError:
    print("Invalid input. Please enter a valid integer.")
    exit(1)

# Check for invalid range
if n < 2:
    print(f"{n} is not a prime number.")
    exit(0)

# Quick checks for small primes and even numbers
if n == 2:
    print(f"{n} is a prime number.")
    exit(0)

if n % 2 == 0:
    print(f"{n} is not a prime number.")
    exit(0)
```

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

The screenshot shows the AI Assistant interface in a code editor. The code in `Lab-1.4.py` is a prime number checker with modular design:

```
1 # Prime Number Checker - Modular Design with Functions
2 # Demonstrates clean code architecture and reusability
3 """
4 """
5
6
7 def get_user_input() -> int:
8     """
9         Prompts user for input and validates it.
10    """
11    Returns:
12        int: A valid positive integer from user input
13
14    Raises:
15        SystemExit: If input is invalid
16
17 while True:
18     user_input = input("Enter a positive integer: ").strip()
19     try:
20         number = int(user_input)
21         if number < 0:
22             print("Please enter a non-negative integer.")
23             continue
24         return number
25     except ValueError:
26         print("Please enter a valid integer number")
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
```

The terminal window shows the execution of the script and user interactions:

```
PS C:\Users\ravul\Downloads\AI Assistant> & c:/users/ravul/appdata/local/programs/python/python314/python.exe "c:/users/ravul/downloads/ai assistant/lab-1.4.py"
Enter a positive integer: 8
8 is not a prime number.
PS C:\Users\ravul\Downloads\AI Assistant> & c:/users/ravul/appdata/local/programs/python/python314/python.exe "c:/users/ravul/downloads/ai assistant/lab-1.4.py"
Enter a positive integer: 1
1 is not a prime number.
PS C:\Users\ravul\Downloads\AI Assistant> & c:/users/ravul/appdata/local/programs/python/python314/python.exe "c:/users/ravul/downloads/ai assistant/lab-1.4.py"
Enter a positive integer: 2
2 is a prime number.
```

The AI Assistant sidebar provides feedback and key improvements:

- Refactored code for modular prime number check
- Edited `Lab-1.4.py` +87 -37
- Perfect! I've refactored your code into a modular design with proper functions:
- Key Improvements:**
  - `get_user_input()` - Handles input validation with loop
  - `is_prime(n)` - Core primality logic with O( $n$ ) complexity
  - `display_result()` - Separation of concerns for output
  - `main()` - Orchestrates the program flow

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

The screenshot shows the AI Assistant interface. In the center is a code editor with a Python file named 'Lab-1.4.py'. The code defines a function `def is_prime(n: int) -> bool:` that checks if a number is prime. The AI provides a detailed explanation of the algorithm's time complexity and edge cases. Below the code editor is a terminal window showing three command-line executions of the script with inputs 1, 2, and 3, and outputs indicating they are not prime. To the right of the code editor is a 'CHAT' panel with a sidebar titled 'ASSISTANCE WITH PYTHON FILE TASKS' containing a 'Key Sections' list. The list includes sections like 'Code Structure', 'Comparison Table', 'Pros & Cons', etc. At the bottom right of the interface is a 'Claude Haiku' section.

```
def is_prime(n: int) -> bool:
    """Determines if a number is prime using optimized trial division.

    Algorithm: O(√n) time complexity
    - Handles edge cases (n < 2)
    - Eliminates even numbers
    - Checks only odd divisors up to √n

    Args:
        n (int): The number to check for primality

    Returns:
        bool: True if n is prime, False otherwise
    """
    # Edge case: numbers less than 2 are not prime
    if n < 2:
        return False
    # 2 is the only even prime
    if n == 2:
        return True
    # Eliminate even numbers
```

PS C:\Users\ravul\Downloads\AI Assistant> & C:/Users/ravul/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/ravul/Downloads\AI Assistant/lab-1.4.py"
Enter a positive integer: 1
1 is not a prime number.
PS C:\Users\ravul\Downloads\AI Assistant> & C:/Users/ravul/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/ravul/Downloads\AI Assistant/lab-1.4.py"
Enter a positive integer: 2
2 is not a prime number.
PS C:\Users\ravul\Downloads\AI Assistant> & C:/Users/ravul/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/ravul/Downloads\AI Assistant/lab-1.4.py"
Enter a positive integer: 3
3 is not a prime number.

## Comparison Table:

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

Aspect	With Function ( <code>is_prime</code> )	Without Function (inline)
Code Size (per use)	Slight overhead for definition, small calls	Shorter once, longer when reused

## TASK-5

The screenshot shows a Python code editor interface with several tabs open. The main tab contains a recursive function definition for determining if a number is prime. The code includes comments explaining the algorithm's time complexity and edge cases. Below the code, a terminal window shows the function being run and tested with integer inputs. The interface includes a sidebar with AI assistance, a chat window, and various toolbars.

```

def is_prime(n: int) -> bool:
    """
    Determines if a number is prime using optimized trial division.

    Algorithm: O(√n) time complexity
    - Handles edge cases (n < 2)
    - Eliminates even numbers
    - Checks only odd divisors up to √n

    Args:
        n (int): The number to check for primality

    Returns:
        bool: True if n is prime, False otherwise
    """
    # Edge case: numbers less than 2 are not prime
    if n < 2:
        return False
    # 2 is the only even prime
    if n == 2:
        return True
    # Eliminate even numbers
    if n % 2 == 0:
        return False
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True

```

### EXECUTION FLOW

#### WITH FUNCTION:

- Main flow → `is_prime()` call → 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 |