

NAME : VANGA MANIKANTA VARAPRASAD

ROLL NO: 2303A52130

BATCH: 44

Task-1

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "AI-ASSISTED-CODING" containing a file "lab1-4.py".
- Code Editor:** Displays Python code to check if a number is prime without using functions or loops.

```
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")
```
- Terminal:** Shows the command-line output for two test cases:

```
Enter a number: 25
25 is not a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding> & C:/Users/morav/AppData\Local/Python/pythoncore-3.14-64/python.exe c:/users/morav/OneDrive/Desktop/labs/ai-assisted-coding/lab1-4.py
Enter a number: 2
2 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding> & C:/Users/morav/OneDrive\Desktop\labs\ai-assisted-coding/lab1-4.py
Enter a number: 11
11 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding>
```
- CHAT:** A sidebar titled "FILE PATH FOR PYTHON SCRIPT LOCATION" contains the prompt "Generate a prime number logic without using function and using user-input". Below it, a message says "Used 1 reference" and provides the generated code.
- Code Snippet:** The generated code is:

```
# 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")
else:
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break

    if is_prime:
        print(f"{num} is a prime number")
    else:
        print(f"{num} is not a prime number")
```

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 VS Code interface with the following details:

- File Explorer:** Shows a folder named "AI-ASSISTED-CODING" containing a file "lab1-4.py".
- Code Editor:** Displays the original code for checking if a number is prime:

```
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
```
- Terminal:** Shows the command `powershell` and the output of running the script.
- CHAT:** A sidebar titled "FILE PATH FOR PYTHON SCRIPT LOCATION" shows the path "C:/Users/morav/Desktop/labs/ai-assisted-coding/lab1-4.py". It also displays the optimized prime number checking logic generated by AI:# Task-1: Check if a number is prime without using functions
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 3
 for i in range(3, int(num ** 0.5) + 1, 2):
 if num % i == 0:
 is_prime = False
 break

if is_prime:
 print(f"{num} is a prime number")
else:
 print(f"{num} is not a prime number")

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 a code editor interface with the following details:

- File Explorer:** Shows a folder named "AI-ASSISTED-CODING" containing a file "lab1-4.py".
- Code Editor:** Displays the following Python code:

```
# Task-1: Check if a number is prime using a function

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

# Get user input
num = int(input("Enter a number: "))

# Output result
if is_prime(num):
    print(f"{num} is a prime number")
else:
    print(f"{num} is not a prime number")
```
- Terminal:** Shows the command line interface with the following interaction:

```
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding> & C:/Users/morav/AI-ASSISTED-CODING/lab1-4.py
Enter a number: 2
2 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding> & C:/Users/morav/AI-ASSISTED-CODING/lab1-4.py
Enter a number: 11
11 is a prime number
PS C:\Users\morav\OneDrive\Desktop\labs\ai-assisted-coding>
```
- Chat:** A sidebar titled "Prime Number Check Using Functions" contains the following text:

Used 1 reference
✓ Updating code to use a prime-check function
Here's an updated version that uses a function while keeping the optimized checks:

```
# Task-1: Check if a number is prime using a function

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
```

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 a terminal window with a dark background. At the top, there's a header bar with icons for file operations and settings. Below the header, a text input field says "FILE PATH FOR PYTHON SCRIPT LOCATION". A large text area contains the following text:

Comparative Analysis –With vs Without Functions
Expected Output
Comparison table or short analytical report

Below this text area, there's a button labeled "lab1-4.py". Underneath the main text area, there's a list of items with checkmarks:

- Used 1 reference
- ✓ Preparing comparison summary

Following this, a message says "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 (<code>is_prime</code>)	Without Function (inline)
Reusability	High: callable from anywhere	Low: logic duplicated where needed

Aspect	With Function (<code>is_prime</code>)	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

```

1  # Single user input reused for both approaches
2  num = int(input("Enter a number: "))
3
4  # Function-based check
5  if is_prime(num):
6      print("[with function] (num) is a prime number")
7  else:
8      print(f"[with function] (num) is not a prime number")
9
10 # Inline (no-function) check
11 if num <= 1:
12     inline_prime = False
13 elif num == 2:
14     inline_prime = True
15 elif num % 2 == 0:
16     inline_prime = False
17 else:
18     inline_prime = True
19     for i in range(3, int(num ** 0.5) + 1, 2):
20         if num % i == 0:
21             inline_prime = False
22             break
23
24 if inline_prime:
25     print("[without function] (num) is a prime number")
26 else:
27     print(f"[without function] (num) is not a prime number")
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

```

CHAT

Aspect	Iterative	Recursive (naive)
Time complexity	$O(n)$	$O(q^n)$ 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

Prime Checking: Algorithmic Approaches

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 / odds	$\sim \frac{1}{2} - \frac{1}{3}$ of \sqrt{n} checks	Small/medium n	Prunes obvious composites
Deterministic Miller-Rabin (3/2/64-bit)	$\sim O(k \log^k n)$	Larger ints, single check	Fast and exact with fixed bases
Probabilistic Miller-Rabin	$\sim O(k \log^k n)$	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 → `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	