

Name : K.Bhavya Sri

H NO:2303A51863

Batch-13

Assignment-1

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:

write a Python program to accept a number from the user and check whether it is a prime number or not, using logic only in the main code (no user-defined functions), which accepts user input and display the result.

#Code

```

1 #use GitHub Copilot to generate a Python program that:
2 #> Does not use any user-defined functions
3 #> Checks whether a given number is prime
4 number = int(input("Enter a number: "))#> Accepts user input
5 is_prime = True
6 if number <= 1:
7     is_prime = False
8 for i in range(2, int(number**0.5) + 1):
9     if number % i == 0:#> Implements logic directly in the main code
10         is_prime = False
11         break
12 if is_prime:
13     print(f"{number} is a prime number.")
14 else:
15     print(f"{number} is not a prime number.")
16

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

havya/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/Bhavya/OneDrive/ai coding/LAB1.py"
 Enter a number: 5
 5 is a prime number.
 PS C:\Users\Bhavya\OneDrive\ai coding> & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/Bhavya/OneDrive/ai coding/LAB1.py"
 Enter a number: 6
 6 is not a prime number.
 PS C:\Users\Bhavya\OneDrive\ai coding>

Ln 1, Col 1 Spaces: 4 UFT-8 CRLF { } Python 3.14.2

#Explanation:

The program takes a number as input from the user.

It checks whether the number can be divided by any number other than 1 and itself.

If it is divisible, the number is not a prime number .If it is not divisible by any number, it is a prime number.

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 the given Python program for checking a prime number by reducing unnecessary iterations, limiting the loop range for early termination, and improving code readability. Show both the original code and the optimized code, and explain how the changes improve efficiency and reduce time complexity.

#Code and OUTPUT:

The screenshot shows the Visual Studio Code interface. The left sidebar has 'EXPLORER', 'AI CODING', 'OUTLINE', and 'TIMELINE' sections. The main area shows a code editor with 'LAB1.py' open. The code implements a prime number check with optimizations. The right sidebar shows a 'TERMINAL' tab with the output of running the script. The terminal output shows the user entering '5' and seeing the original and optimized results compared, along with a note about the time complexity improvement.

```
35 # Task 2: Optimized implementation (no user-defined functions)
36 import time
37 start = time.perf_counter()
38
39 optimized_is_prime = True
40 if number <= 1:
41     optimized_is_prime = False
42 elif number <= 3: # 2 and 3 are prime
43     optimized_is_prime = True
44 elif number % 2 == 0: # eliminate even numbers early
45     optimized_is_prime = False
46 else:
47     limit = int(number**0.5) + 1
48     i = 3
49     while i <= limit: # check only odd divisors
50         if number % i == 0:
51             optimized_is_prime = False
52             break
53         i += 2
54
55 end = time.perf_counter()
56
57 # Outputs comparing both versions
58 print("Original check: <number> is ('prime' if original_is_prime else 'not prime').")
59 print("Optimized check: <number> is ('prime' if optimized_is_prime else 'not prime').")
60 print("Optimized check time: <end - start: 6f> seconds")
61 # Explanation of how the improvements reduce time complexity
62 # - Both algorithms are O(sqrt(n)) in time complexity.
63 # - Optimizations:
64 #   * Early return for n <= 1, and direct handling for 2 and 3.
65 #   * Quick elimination of even numbers (reduces half of candidates).
66 #   * After ruling out 2, iterate only odd divisors (step 2), halving loop iterations.
67 # - Effect: same asymptotic complexity but ~2x fewer divisions and fewer loop iterations,
68 # which matters for large inputs.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\Bhavya\OneDrive\ai coding & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/Bhavya/OneDrive/ai coding/LAB1.py"
Enter a number: 5
5 is a prime number.
Enter a number: 5
Original check: 5 is prime.
Optimized check: 5 is prime.
Optimized check time: 0.000007 seconds
P5 C:\Users\Bhavya\OneDrive\ai coding> 
```

#EXPLANATION:

The optimized code reduces unnecessary iterations by checking divisibility only up to the square root of the number instead of all numbers up to n . Early termination using `break` stops the loop as soon as a factor is found, saving time. Using a boolean variable improves code readability and clarity.

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 using a user-defined function to check whether a given number is a prime number. The function should return a Boolean value (True or False). Accept input from the user, call the function, and display the result. Include clear and meaningful comments to explain the logic

#Code and output

```
# [Task 3: User design using AI assistance (function-based prime check)]
# Task 3: Reusable prime-checking function
# Requirement mapping:
# - Use a user-defined function (Task 3)
# - Return a Boolean value (Task 3)
# - Include meaningful comments (Task 3)

def is_prime(n):
    """
    [Task 3: Docstring]
    Return True if n is a prime number, otherwise False.
    Approach: Check divisibility for i in [2, floor(sqrt(n))]. Early return on first divisor.
    Time Complexity: O(√n).
    """
    if n <= 1:
        return False
    limit = int(n ** 0.5)
    for i in range(2, limit + 1):
        if n % i == 0:
            return False
    return True
    # [Task 3: Example usage and test cases]
tests = [2, 3, 4, 5, 12, 17, 21, 993]
for t in tests:
    print(f"{t}: {is_prime(t)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
9973: True
PS C:\Users\Bhavya\OneDrive\ai coding & C:/Users/Bhavya/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/Bhavya/OneDrive/ai coding/LAB1.py"
2: True
3: True
4: False
5: True
12: False
17: True
21: False
9973: True
PS C:\Users\Bhavya\OneDrive\ai coding> ]
```

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

Compare two Python programs for checking prime numbers: one written without user-defined functions and one written using a function. Analyze them based on code clarity, reusability, ease of debugging, and suitability for large-scale applications.

#Code

```
Compare two Python programs for checking prime numbers: one written without user-defined functions and one written using a function. Analyze them based on code clarity, reusability, ease of debugging, and suitability for large-scale applications.

# Program without user-defined functions
num = int(input("Enter a number: "))
is_prime = True
if num > 1:
    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")
else:
    print(f"{num} is not a prime number")

# Program with user-defined function
def is_prime_number(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if (n % i) == 0:
            return False
    return True
num = int(input("Enter a number: "))
if is_prime_number(num):
    print(f"{num} is a prime number")
else:
    print(f"{num} is not a prime number")

# Comparison Table
comparison_table = """
| Aspect | Without Functions | With Functions |
|-----|-----|-----|
| Code Clarity | Less clear due to inline logic | More clear with separated logic |
| Reusability | Low, logic cannot be reused easily | High, Function can be reused |
| Ease of Debugging | Harder to debug, all logic in one place | Easier to debug, isolated function logic |
| Suitability for Large-Scale Applications | Less suitable, harder to maintain | More suitable, easier to manage and scale |
"""
print(comparison_table)
```

```

# Task 4: Comparative Analysis - With vs Without Functions
#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
#Comparison table or short analytical report
# Aspect | Without Functions (Task 1) | With Functions (Task 3) |
#-----|-----|-----|
# Code Clarity | Lower clarity due to inline logic | Higher clarity with encapsulated logic |
# Reusability | Low - code duplication likely | High - function can be reused |
# Debugging Ease | Harder to isolate issues | Easier to test and |
# Suitability for Large-Scale Applications | Poor - hard to maintain and scale | Good - modular and maintainable |
# Analysis:
# 1. Code Clarity: The function-based approach (Task 3) is clearer due to encapsulation and comments.
# 2. Reusability: The function can be reused in different contexts without code duplication.
# 3. Debugging Ease: Isolated functions allow for targeted testing and debugging.
# 4. Suitability for Large-Scale Applications: Modular design supports maintainability and scalability.
# Conclusion: While the non-function approach may be simpler for quick scripts, the function-based design is superior for larger, more complex applications due to its advantages in clarity, reusability, and maintainability.
# Using functions is a best practice in software development, especially for larger projects.
# The function-based approach is recommended for production code.
# However, For quick scripts or one-off tasks, the non-function approach may suffice.
# Ultimately, the choice depends on the specific use case and project requirements.
# Encouraging best practices like modular design and code reuse is beneficial for long-term code quality.

```

#Output:

```

PS C:\Users\neera\OneDrive\Desktop\AI AC> & "C:/Users/neera/OneDrive/Desktop/AI AC/.venv/Scripts/python.exe" "c:/Users/neera/OneDrive/Desktop/AI AC/#Compare two Python programs for checkin.py"
Enter a number: 17
17 is a prime number
Enter a number: 27
27 is not a prime number

| Aspect | Without Functions | With Functions |
|-----|-----|-----|
| Code Clarity | Less clear due to inline logic | More clear with separated logic |
| Reusability | Low, logic cannot be reused easily | High, function can be reused |
| Ease of Debugging | Harder to debug, all logic in one place | Easier to debug, isolated function logic |
| Suitability for Large-Scale Applications | Less suitable, harder to maintain | More suitable, easier to manage and scale |

```

#JUSTIFICATION

The first program without functions works correctly but has all the logic inline, making it less clear and harder to reuse or debug. The second program uses a user-defined function, which separates the prime-checking logic from the main program, improving readability and maintainability. Using a function allows reusability, so the same logic can be called multiple times without rewriting code.

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 two Python programs to check if a number is prime: one using a basic check from 2 to n-1, and one optimized by checking only up to \sqrt{n} .

#Code and Output:

```

File Edit Selection View Go Run Terminal Help ← → 🔍 ai coding
LAb1.py > LAb1.pyx
EXPLORER OUTLINE TIMELINE
Welcome > LAb1.pyx
1 #Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)
2 # A basic divisibility check approach # An optimized approach (e.g., checking up to  $\sqrt{n}$ )# Two correct Implementations
3 # Basic approach for discussion: # Execution Flow# Time complexity# Performance for Large Inputs# When each approach is appropriate # Basic divisibility check approach
4 def is_prime(n):
5     if n <= 1:
6         return False
7     for i in range(2, n): # check all possible divisors
8         if n % i == 0: # divisor found - not prime
9             return False
10    return True # no divisors = prime
11
12 # Optimized approach (checking up to  $\sqrt{n}$ )
13 def is_prime_optimized(n):
14     if n <= 1:
15         return False
16     for i in range(2, int(n**0.5) + 1):
17         if n % i == 0:
18             return False
19     return True
20
21 # Comparison
22
23 test_number = 99983 # A large prime number for testing
24 # Basic approach timing
25 start_basic = time.perf_counter()
26 is_basic_prime = is_prime_basic(test_number)
27 end_basic = time.perf_counter()
28
29 start_optimized = time.perf_counter()
30 is_optimized_prime = is_prime_optimized(test_number)
31 end_optimized = time.perf_counter()
32
33 # Results
34 print(f"Basic approach: {test_number} is {'prime' if is_basic_prime else 'not prime'}, Time: {(end_basic - start_basic)*1000} seconds")
35 print(f"Optimized approach: {test_number} is {'prime' if is_optimized_prime else 'not prime'}, Time: {(end_optimized - start_optimized)*1000} seconds")
36
37 # Evaluation:
38 # Execution Flow:
39 # - Basic approach checks all numbers from 2 to n-1.
40 # - Optimized approach checks up to  $\sqrt{n}$ , reducing iterations significantly.
41 # Time Complexity:
42 # - Basic: O(n)
43 # - Optimized: O( $\sqrt{n}$ )
44 # Performance for Large Inputs:
45 # - The optimized approach is significantly faster for large n due to fewer iterations.
46 # Applications:
47 # - Basic: Suitable for small numbers or educational purposes.
48 # - Optimized: Preferred for larger numbers or performance-critical applications.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Bhavya\OneDrive\AI coding & C:\Users\Bhavya\AppData\Local\Programs\Python\Python311\python.exe "C:/Users/Bhavya/OneDrive/AI coding/LAb1.py"
Basic approach: 99983 is prime. Time: 0.002630 seconds
Optimized approach: 99983 is prime. Time: 0.000065 seconds
PS C:\Users\Bhavya\OneDrive\AI coding>

#Justification

The first program uses a basic divisibility check from 2 to n-1. While it is simple and easy to understand, it performs many unnecessary iterations, making it inefficient for large numbers.

The second program uses an optimized approach, checking divisibility only up to \sqrt{n} . This reduces the number of iterations significantly, improving efficiency while giving the same correct result.

The optimized version is faster, easier to scale, and more suitable for larger input values compared to the basic approach.