AI Assisted Coding

ASSIGNMENT-1

THURSDAY LAB-1

J.Shashank Reddy
2303A52213
Batch - 43

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 code to check the given number is prime or not without using functions .

CODE:

```python
# Task 1 Check if a specific number is prime
num = int(input("\nEnter a number to check if it's prime: "))
is_prime = True
if num < 2:
    is_prime = False
else:
    for i in range(2, num):
        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.")
```

Output for Task:1

```
PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & C:\Users\gadda\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding/Assginment1.py"

Enter a number to check if it's prime: 16
16 is not a prime number.

--- Optimized Prime Checking ---
```

Observations:

The program effectively checks whether a given number is prime by implementing the entire logic directly in the main code without using any user defined functions. It accepts user input and applies conditional statements and loops to test divisibility. This approach makes the code simple and easy to understand, especially for beginners learning basic programming concepts. While the absence of modularization limits reusability and scalability, it helps in clearly demonstrating the core logic. Overall, the task fulfils its objective of basic validation in a learning-focused application

TASK-2 : Efficiency & Logic Optimization (Cleanup)

PROMPT:

optimize the code and simplify the logic to check to get the optimal solution for prime number checking.

Code:

```
# Task 2: Optimized Prime Number Check
print("\n--- Optimized Prime Checking ---")
num = int(input("Enter a number to check if it's prime: "))
is_prime = True

if num < 2:
    is_prime = False
elif num == 2:
    is_prime = True
elif num % 2 == 0:
    is_prime = False
else:
    # Only check odd divisors up to sqrt(num)
    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.")
```

Task 2 OUTPUT

```
Enter a number to check if it's prime: 150000
150000 is not a prime number.

--- Function-Based Prime Checking ---
2 is a prime number
15 is not a prime number
17 is a prime number
20 is not a prime number
29 is a prime number
1 is not a prime number
```

Observations:

The original code checks divisibility from 2 to num-1, resulting in O(n) time complexity, which is inefficient for large numbers. The optimized version reduces unnecessary iterations by checking divisors only up to the square root of the number, since a larger factor would have a corresponding smaller factor already checked. This reduces the time complexity to

O(√n). Early termination using break further improves performance. Additionally, clearer variable names and simplified logic enhance readability and maintainability

TASK-3: Modular Design Using AI Assistance (Prime Number Check Using Functions

PROMPT :

Give me a code for to check the prime number using the functions

Code:

```python
# Task 3: Modular Design Using Functions

def is_prime(num):
    """
    Check if a number is prime using optimized algorithm.

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

    Returns:
        bool: True if the number is prime, False otherwise
    """
    if num < 2:
        return False
    elif num == 2:
        return True
    elif num % 2 == 0:
        return False
    else:
        # Only check odd divisors up to sqrt(num)
        for i in range(3, int(num**0.5) + 1, 2):
            if num % i == 0:
                return False
```

Output :

```
--- Interactive Prime Check ---
Enter a number to check if it's prime: 16
16 is not a prime number.
```

Observations:

The modular, function-based implementation successfully encapsulates the prime-checking logic into a reusable user-defined function that returns a Boolean value. This design improves code reusability across multiple modules and enhances maintainability. The use of meaningful, AI-assisted comments increases code clarity and helps in understanding the logic flow. Returning a Boolean value allows the function to be easily integrated into different applications or validation pipelines. Overall, the modular approach results in cleaner, more scalable, and professional-quality code.

TASK-4 : Comparative Analysis –With vs Without Functions

PROMPT:

Give me the comparative analysis for the task1 task2 task 3 .

Code:

```
# Task 4: Comparative Analysis - With vs Without Functions

print("\n--- COMPARATIVE ANALYSIS REPORT ---\n")

comparison_data = {
    "Aspect": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability", "Maintainabil
    "Without Functions (Tasks 1-2)": [
        "Low - Logic mixed with I/O",
        "Low - Code duplication required",
        "Hard - Multiple locations to fix",
        "Poor - Not scalable",
        "Difficult - Changes needed everywhere",
        "Poor - No isolation of logic"
    ],
    "With Functions (Task 3)": [
        "High - Clear separation of concerns",
        "High - Single function call",
        "Easy - Fix in one location",
        "Excellent - Modular and scalable",
        "Easy - Centralized logic",
        "Excellent - Function can be tested independently"
    ]
}
```

Output :

```
Aspect                   Without Functions                       With Functions
---------------------------------------------------------------------------------------------------------
Code Clarity             Low - Logic mixed with I/O              High - Clear separation of concerns
Reusability              Low - Code duplication required         High - Single function call
Debugging Ease           Hard - Multiple locations to fix        Easy - Fix in one location
Large-Scale Suitability  Poor - Not scalable                     Excellent - Modular and scalable
Maintainability          Difficult - Changes needed everywhere   Easy - Centralized logic
Testing                  Poor - No isolation of logic            Excellent - Function can be tested independentl
y
```

Observation:

The version without functions is straightforward and easy to follow for beginners, but the logic is tightly coupled to the main code, reducing overall clarity as the program grows. In contrast, the function-based implementation presents clearer structure by separating logic from execution, making the code easier to read and understand. Reusability is minimal in the non-modular version, whereas the function-based approach allows the prime-checking logic to be reused across multiple modules. Debugging is simpler with functions since errors can be isolated within a single unit of code, unlike the monolithic structure where issues are harder to trace. Overall, the function-based design is far more suitable for large-scale and maintainable applications, while the non-function approach is best limited to small or educational scripts.

TASK- 5:

AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking

Prompt: Iterative vs recursive Fibonacci give me the code

Code:

```python
# Task 5: Alternative Approaches to Prime Checking

print("\n--- TASK 5: ALTERNATIVE ALGORITHMIC APPROACHES ---\n")

# Approach 1: Basic Divisibility Check2
def is_prime_basic(num):
    """
    Basic approach: Check divisibility by all numbers up to num-1.

    Args:
        num (int): The number to check

    Returns:
        bool: True if prime, False otherwise
    """
    if num < 2:
        return False
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

# Approach 2: Optimized Square Root Approach
```

Output:

```
--- TASK 5: ALTERNATIVE ALGORITHMIC APPROACHES ---

--- PERFORMANCE COMPARISON ---

Number: 97
  Basic Approach:     0.0143ms
  Optimized Approach: 0.0064ms
  Speedup: 2.22x faster
```

Observations:

The basic divisibility approach is simple and easy to understand, but it performs many unnecessary checks, making it inefficient for larger input values. The optimized approach improves performance by limiting the loop to the square root of the number, reducing the number of iterations significantly. Execution flow in the optimized version is shorter and more efficient due to early termination. For large inputs, the optimized method provides much better performance compared to the naive approach. Overall, the basic method is suitable for learning purposes, while the optimized approach is more appropriate for realworld and large-scale applications.