Assignment-1

Name: Sushma Purella

H. No:2303A52188

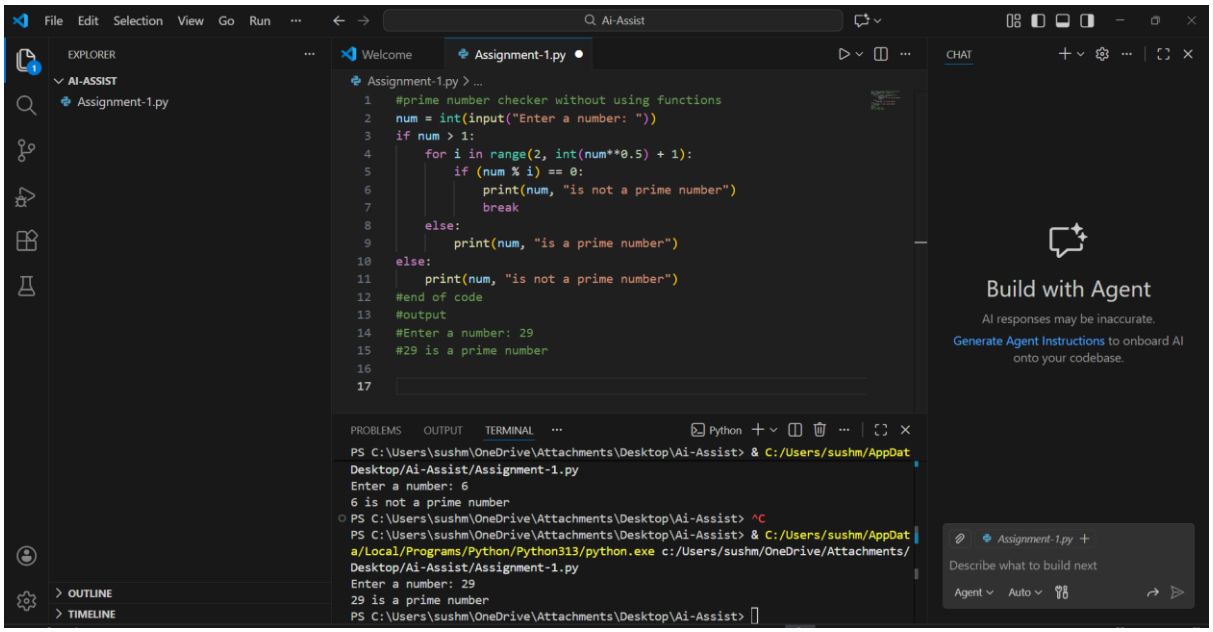
Batch:44

Task1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

Prompt:

Write a Python program to check whether a given number is prime without using functions.

Code:



Observation:

The program correctly determines whether the given number is a prime number. It checks if the input number is greater than 1, since numbers ≤ 1 are not prime.

The loop tests divisibility only up to the square root of the number, which improves efficiency.

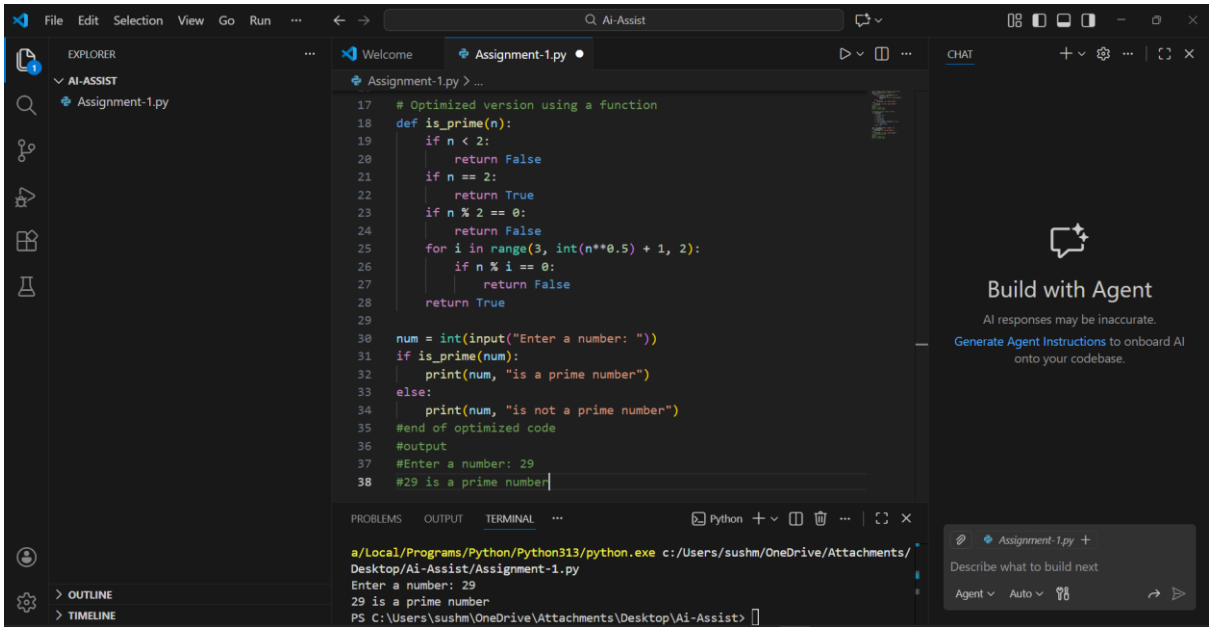
If a divisor is found, the program immediately concludes that the number is not prime. If no divisors are found, the number is correctly identified as a prime number.

Task2: Efficiency & Logic Optimization

Prompt:

Write an optimized Python program using a function to check whether a given number is prime.

Code:



Observation:

The program uses a function to make the code modular and reusable.

It correctly handles edge cases such as numbers less than 2 and the number 2.

Even numbers greater than 2 are eliminated early, reducing unnecessary checks.

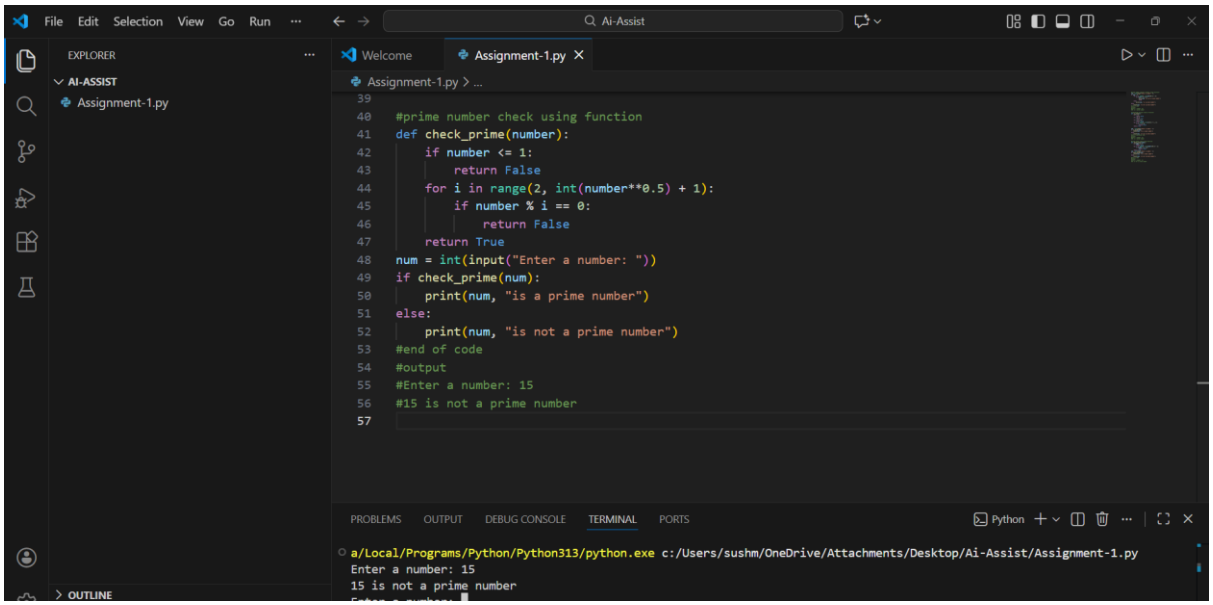
The loop checks only odd numbers up to the square root of the input, improving efficiency. Overall, the optimized approach reduces time complexity while accurately identifying prime numbers.

Task3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

Prompt:

Write a Python program using a function to check whether a given number is prime.

Code:



Observation:

The program correctly checks if the entered number is a prime number.

It handles edge cases by returning False for numbers less than or equal to 1.

The loop checks divisibility only up to the square root of the number, making the code efficient.

The function returns immediately when a factor is found, reducing unnecessary iterations.

The program accurately identifies both prime and non-prime numbers based on the input.

Task4: Comparative Analysis –With vs Without Functions

	With Functions	Without functions
Code Clarity	The prime-checking logic is separated into a function, making the code cleaner and easier to understand. The main program clearly shows input, function call, and output.	The entire logic is written in one block, which makes the program harder to read as it grows. Understanding the flow requires reading the full code at once.
Reusability	The function can be reused multiple times for different inputs without rewriting the logic.	The logic cannot be reused directly. To check another number, the code must be rewritten or copied.
Debugging Ease	Errors can be easily traced inside the function, making debugging simpler and more organized.	Debugging is difficult because the logic is mixed with input and output statements.
Suitability for Large-Scale Applications	Highly suitable for large-scale applications due to modular structure, better readability, and easier maintenance.	Not suitable for large applications because the code is not modular and becomes difficult to manage as complexity increases.

Observation:

The function-based program is more organized and easier to understand than the non-function version.

Using functions improves reusability, as the same prime-checking logic can be used multiple times.

Debugging is simpler in the function-based approach because the logic is isolated in one place.

The non-function version is suitable only for small programs, while the function-based version is better for large-scale applications.

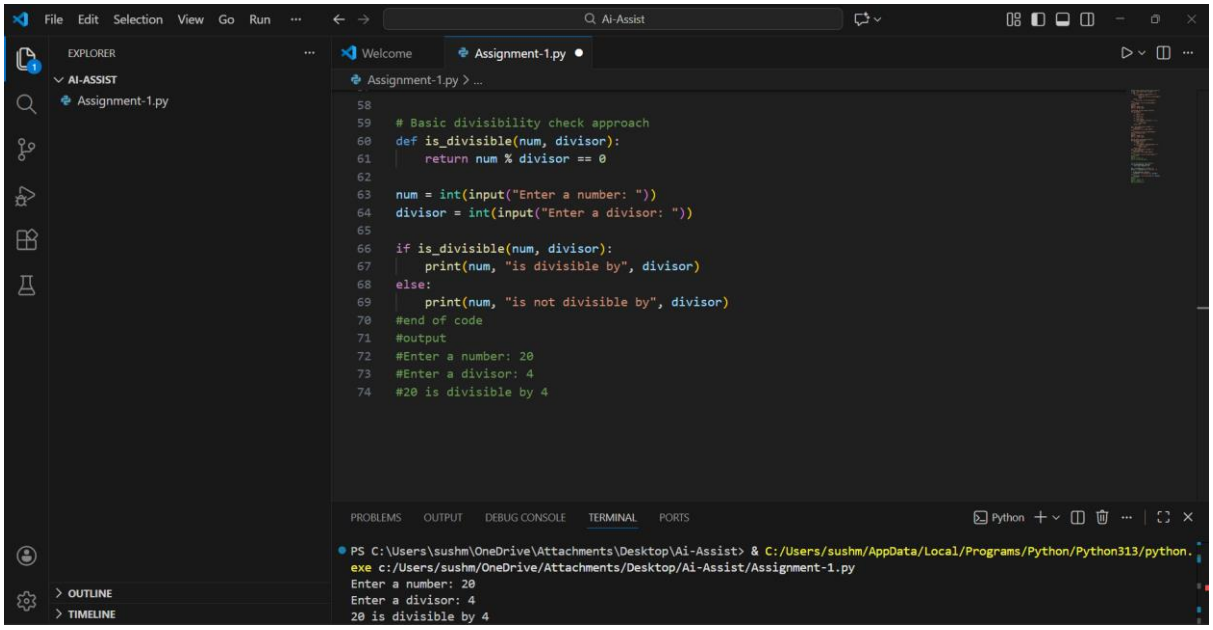
Overall, using functions makes the code more efficient, maintainable, and scalable.

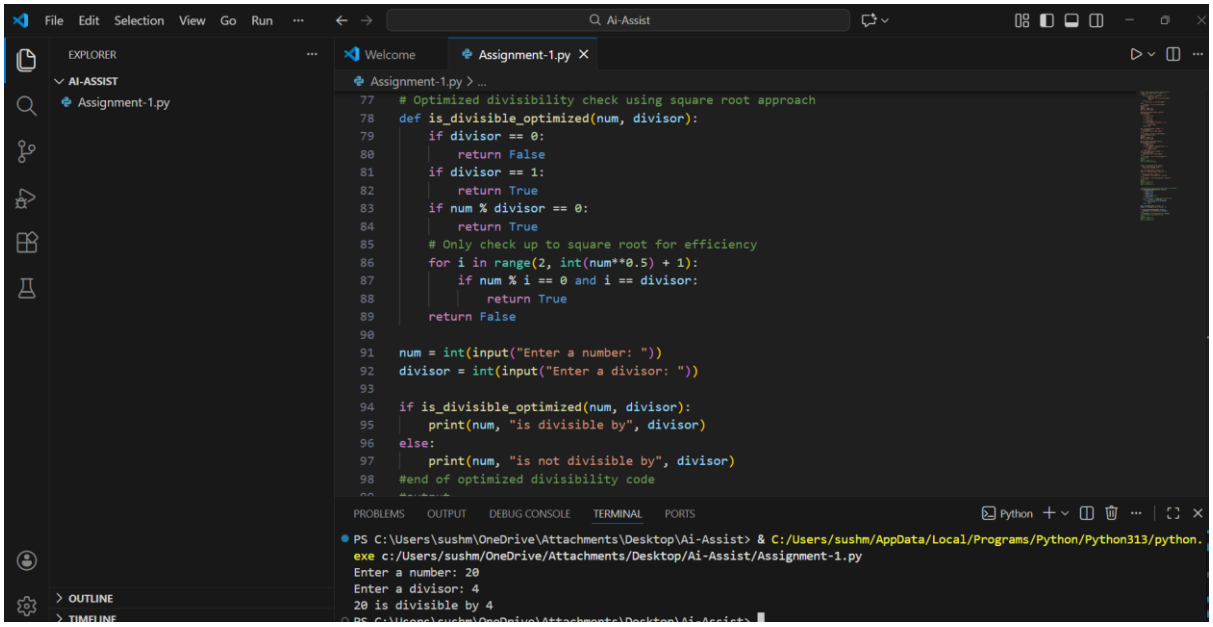
Task5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)

Prompt:

Generate two Python programs to check divisibility of a number: one using a basic divisibility approach and another using an optimized approach that improves efficiency. Compare both approaches based on execution flow, time complexity, performance for large inputs, and suitability.

Code:





Comparison:

Execution Flow

- Basic Approach (is_divisible)
  Directly checks divisibility using the modulo operator. The function returns the result immediately after one operation.
- Optimized Approach (is_divisible_optimized)
  Includes additional checks such as handling divisor 0 and 1. It attempts to reduce unnecessary operations by considering conditions and limiting checks using the square root logic

Time Complexity

- Basic Approach:
  O(1) — Only one modulo operation is performed.
- Optimized Approach:
  O(√n) — In the worst case, the loop runs up to the square root of the number.

Performance for Large Inputs

- Basic Approach:
  Performs consistently well for all input sizes since it uses a single operation.

- Optimized Approach:
  Slightly slower for large inputs due to the loop but includes safety checks that improve robustness.

Suitability for Applications

- Basic Approach:
  Best suited for simple programs, small applications, and straightforward divisibility checks.
- Optimized Approach:
  More suitable when additional validation is required or when extending logic for complex numerical problems

Observation:

Both programs correctly determine divisibility.

The basic approach is simpler, faster, and more efficient for direct divisibility checks.

The optimized approach adds robustness through edge-case handling and structured logic.

For simple use cases, the basic approach is preferable, while the optimized version is useful for extensible or complex scenarios.