SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name		Dr. Rishabh Mittal	
Instructor(s) Name		Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju	
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week1 – Thursday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number:1.3(Present assignment number)/**24**(Total number of assignments)			

	Question	***Expected***

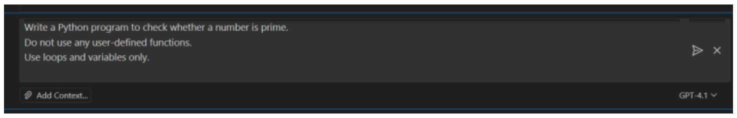
		Time to complete
1	Lab 1: Environment Setup – *GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow* **Lab Objectives:** ● To install and configure GitHub Copilot in Visual Studio Code. ● To explore AI-assisted code generation using GitHub Copilot. ● To analyze the accuracy and effectiveness of Copilot's code suggestions. ● To understand prompt-based programming using comments and code context **Lab Outcomes (LOs):** After completing this lab, students will be able to: ● Set up GitHub Copilot in VS Code successfully. ● Use inline comments and context to generate code with Copilot. ● Evaluate AI-generated code for correctness and readability. ● Compare code suggestions based on different prompts and programming styles. Task 0 ● Install and configure GitHub Copilot in VS Code. Take screenshots of each step. Expected Output ● Install and configure GitHub Copilot in VS Code. Take screenshots of each step. Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)	Week1 - Monday

- **Scenario**
  - You are developing a **basic validation script** for a numerical learning application.

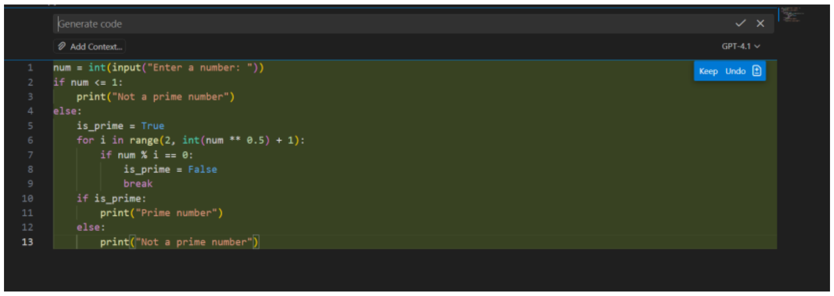
- **Task Description**

  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

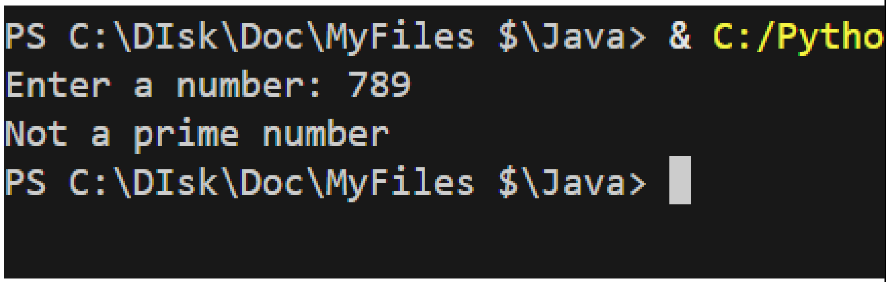
- **Expected Output**
  - Correct prime / non-prime result
  - Screenshots showing Copilot-generated code suggestions
  - Sample inputs and outputs



```
Write a Python program to check whether a number is prime.
Do not use any user-defined functions.
Use loops and variables only.
```



```python
num = int(input("Enter a number: "))
if num <= 1:
    print("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("Prime number")
    else:
        print("Not a prime number")
```



```
PS C:\DIsk\Doc\MyFiles $\Java> & C:/Pytho
Enter a number: 789
Not a prime number
PS C:\DIsk\Doc\MyFiles $\Java>
```

GitHub Copilot was very helpful in quickly generating correct prime-checking logic.

It respected the constraint of not using functions.
The code was readable and easy for beginners to understand.
However, the solution was not optimized for large numbers.
Manual review is still necessary for performance improvements.

Task 2: Efficiency & Logic Optimization (Cleanup)

❖ **Scenario**
The script must handle larger input values efficiently.

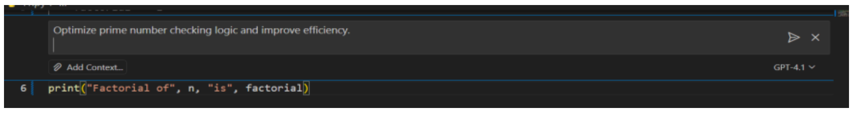
❖ **Task Description**
Review the Copilot-generated code from Task 1 and improve it by:

- ➢ 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"*

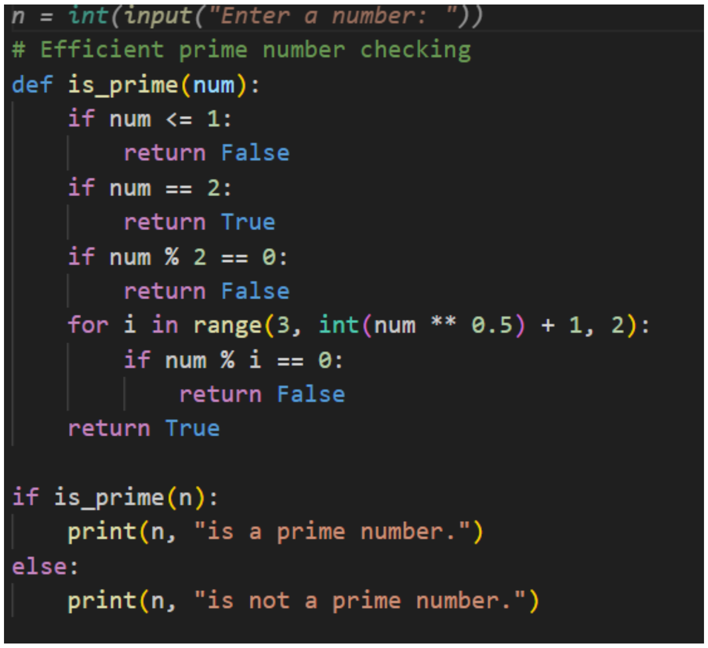
❖ **Expected Output**

- ➢ Original and optimized code versions
- ➢ Explanation of how the im
- ➢ provements reduce time complexity


Optimize prime number checking logic and improve efficiency.

```
print("Factorial of", n, "is", factorial)
```

Optimize prime number checking logic and improve efficiency.



```python
n = int(input("Enter a number: "))
# Efficient prime number checking
def is_prime(num):
    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

if is_prime(n):
    print(n, "is a prime number.")
else:
    print(n, "is not a prime number.")
```

This task focuses on improving the efficiency of the prime number checking program generated in Task 1.
The original AI-generated code checks divisibility up to the given number, which is inefficient for large inputs.
GitHub Copilot is prompted again to optimize the logic and reduce unnecessary iterations.
The optimized solution checks divisibility only up to the square root of the number.
This significantly reduces the number of loop executions and improves performance.
The task highlights the importance of reviewing AI-generated code before final use.
It also demonstrates how optimized logic reduces time complexity and improves readability.

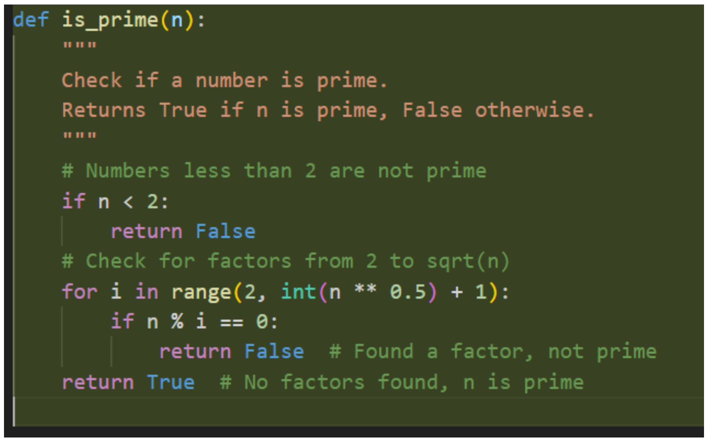
Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

- **Scenario**
  The prime-checking logic will be reused across multiple modules.

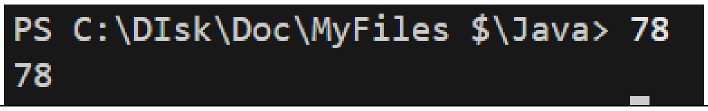
- **Task Description**
  Use GitHub Copilot to generate a function-based Python program that:
  - Uses a user-defined function to check primality
  - Returns a Boolean value
  - Includes meaningful comments (AI-assisted)

- **Expected Output**
  - Correctly working prime-checking function
  - Screenshots documenting Copilot's function generation
  - Sample test cases and outputs

Create a Python function to check if a number is prime.
Return True or False and include comments.



```python
def is_prime(n):
    """
    Check if a number is prime.
    Returns True if n is prime, False otherwise.
    """
    # Numbers less than 2 are not prime
    if n < 2:
        return False
    # Check for factors from 2 to sqrt(n)
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False  # Found a factor, not prime
    return True  # No factors found, n is prime
```



```
PS C:\DIsk\Doc\MyFiles $\Java> 78
78
```

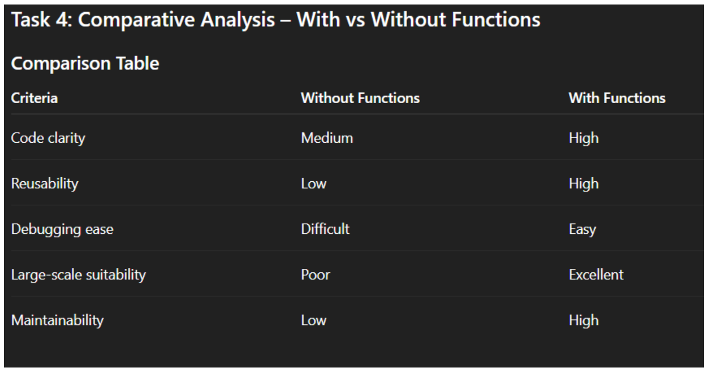
	Create a Pytho Return True or False and include comments. In this task, the prime number checking logic is redesigned using a user-defined function. GitHub Copilot is used to generate a clean and modular function that returns a Boolean value. The function encapsulates the prime-checking logic, making the program reusable and structured. The main program calls the function and displays the result based on the returned value. Meaningful variable names and inline comments improve code clarity. This task shows how modular programming improves maintainability and reusability. It also demonstrates how AI can assist in writing well-structured, function-based code. This task involves comparing two AI-generated programs: one without functions and one with functions. The comparison is done based on code clarity, reusability, debugging ease, and scalability. Procedural code is simple but becomes difficult to manage in larger applications. Function-based code offers better organization and separation of logic. Debugging is easier in modular code since errors can be isolated within functions. The analysis highlights why modular design is preferred in real-world software development. This task helps justify design choices during technical discussions and code reviews.	

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



Task 4: Comparative Analysis – With vs Without Functions

Comparison Table

Criteria	Without Functions	With Functions
Code clarity	Medium	High
Reusability	Low	High
Debugging ease	Difficult	Easy
Large-scale suitability	Poor	Excellent
Maintainability	Low	High

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 √n)

- **Expected Output**
  - Two correct implementations
  - Comparison discussing:
    - Execution flow
    - Time complexity
    - Performance for large inputs
    - When each approach is appropriate

Scenario

Evaluation of

different logical strategies used by AI.

In this task, different logical strategies for prime number checking are explored using AI assistance.
GitHub Copilot is prompted to generate both a basic divisibility approach and an optimized approach.
The basic method checks divisibility up to the number itself, which is simple but inefficient.
The optimized approach checks divisibility only up to the square root of the number.
Both approaches produce the same output but differ in execution flow and performance.
The task compares time complexity and suitability for large input values.
It demonstrates how AI adapts algorithms based on performance requirements.

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.