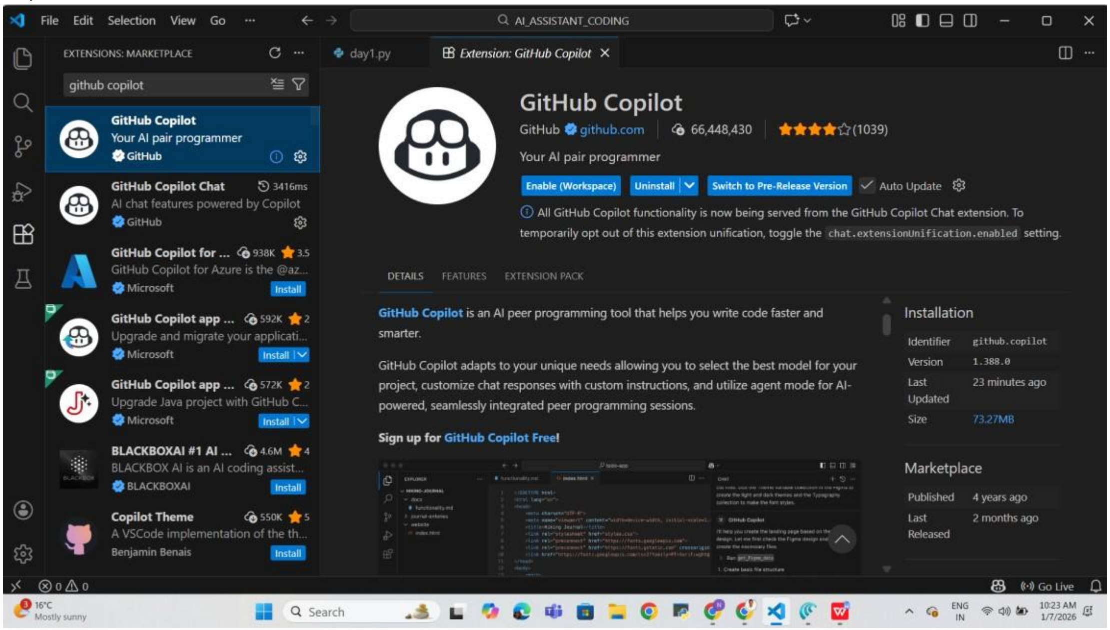
Lab assignment - 1.4 2303A51797

8-1-26

TASK - 0

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)

❖ 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

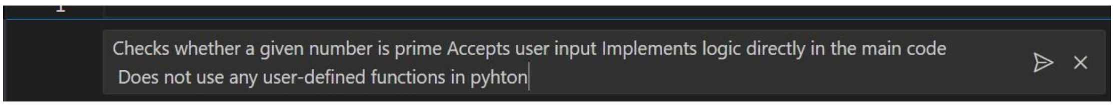
➢ Does not use any user-defined functions

❖ Expected Output

➢ Correct prime / non-prime result

➢ Screenshots showing Copilot-generated code suggestions

➢ Sample inputs and outputs



ass-1.4.py > ...

```python
# Prime number checker without user-defined functions

number = int(input("Enter a number to check if it is prime: "))

if number < 2:
    print(f"{number} is not a prime number.")
else:
    is_prime = True
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            is_prime = False
            break

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

```
:/AI_ASSISTANT_CODING/ass-1.4.py
Enter a number to check if it is prime: 5
5 is a prime number.
PS D:\AI_ASSISTANT_CODING>
```

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

```
optimize prime number checking logic and improve efficiency of this code
Add Context...                                                     Auto
```

```python

        # Prime number checker without user-defined functions

number = int(input("Enter a number to check if it is prime: "))

        # Check if prime
is_prime = number > 1 and all(number % i != 0 for i in range(2, int(number ** 0.5) + 1))

print(f"{number} is {'a prime' if is_prime else 'not a prime'} number.")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe"
:/AI_ASSISTANT_CODING/ass-1.4.py
5 is a prime number.
Enter a number to check if it is prime: 5
5 is a prime number.
PS D:\AI_ASSISTANT_CODING>
```

The optimization checks divisibility only up to √n instead of n, reducing unnecessary iterations. This works because any composite number must have a factor less than or equal to √n. Early termination using break or all() stops execution as soon as a divisor is found. Thus, time complexity improves from O(n) to O(√n), with best case O(1).

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

```
Uses a user-defined function to check primality Returns a Boolean value Includes meaningful comments (AI-ass
Accept  Accept Word  ···                                                                      Auto
```

```
ass-1.4.py > ...
# User-defined function to check if a number is prime
def is_prime_number(n):
    """
    Checks if a number is prime.

    Args:
        n: Integer to check for primality

    Returns:
        Boolean: True if prime, False otherwise
    """
    # Numbers less than 2 are not prime
    if n < 2:
        return False

    # Check divisibility from 2 to square root of n
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False

    # If no divisors found, number is prime
    return True

# Test the function
test_number = int(input("Enter a number to check if it is prime: "))
result = is_prime_number(test_number)
print(f"{test_number} is {'a prime' if result else 'not a prime'} number.")
```

```
PS D:\AI_ASSISTANT_CODING> & "C:/Users/Achanti sai teja/AppData/Local/Programs/Python/Python313/python.exe" d:/AI_ASSISTANT_CODING/ass-1.4.py
Enter a number to check if it is prime: 5
5 is a prime number.
Enter a number to check if it is prime: 5
5 is a prime number.
Enter a number to check if it is prime: 5
5 is a prime number.
PS D:\AI_ASSISTANT_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

Comparison Report of the 3 Prime Number Checker Codes Code 1 – Loop with flag and break:

Uses a loop up to √n and stops early when a divisor is found. Efficient and easy to understand. Time complexity is O(√n).

Code 2 – all() with generator expression:

Also checks up to √n but uses Python’s built-in all() with short-circuiting. More concise and Pythonic, same O(√n) time complexity, slightly better readability.

Code 3 – User-defined function (is_prime_number)

Encapsulates logic in a function, improving reusability and modularity. Performance is the same (O(√n)), best suited for larger programs or repeated checks.

Overall Conclusion:

All three have the same time complexity (O(√n)). Code 2 is the most concise, Code 1 is beginnerfriendly, and Code 3 is best for structured and reusable 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 √n)

❖ Expected Output

➢ Two correct implementations

➢ Comparison discussing:

▪ Execution flow

▪ Time complexity

▪ Performance for large inputs

▪ When each approach is appropriate

A basic divisibility check approach

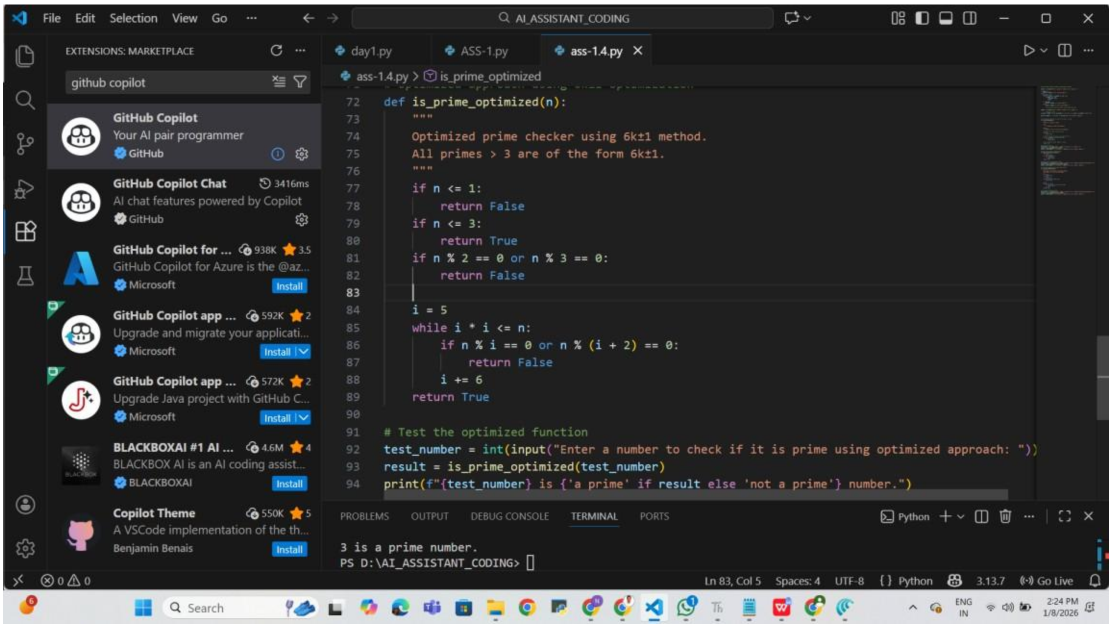
```python
# Basic divisibility check approach
def basic_divisibility_check(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# Test the basic divisibility check
test_number = int(input("Enter a number to check if it is prime using basic divisibility: "))
result = basic_divisibility_check(test_number)
print(f"{test_number} is {'a prime' if result else 'not a prime'} number.")
```

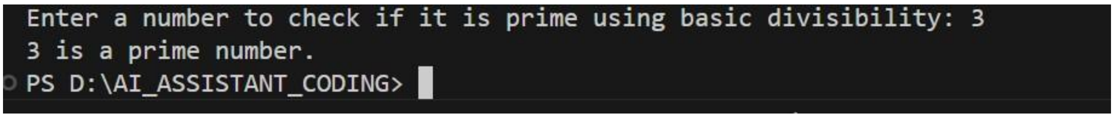
```
Enter a number to check if it is prime using basic divisibility: 3
3 is a prime number.
PS D:\AI_ASSISTANT_CODING>
```

An optimized approach

Add Context... Auto

```python
        print(f"Factorial of {number} is {result}")
```





Comparison (Short Notes) Execution
Flow:
Basic divisibility: Checks all numbers from 2 to n−1 sequentially.
Optimized (√n): Checks divisors only up to √n and stops early if a factor is found.
Function-based: Same optimized flow but wrapped in a reusable function.
Time Complexity:
Basic divisibility: O(n)
Optimized (√n): O(√n)
Function-based optimized: O(√n) Performance
for Large Inputs:
Basic approach is slow and inefficient for large numbers.
Optimized approaches are much faster and suitable for large inputs.
Function-based version performs the same but is cleaner for repeated use.
When Each Approach Is Appropriate
Basic approach: For learning and very small inputs.
Optimized approach: For efficient single checks.
Function-based approach: For structured programs and multiple prime checks.