AI ASSISTED CODING LAB

ASSIGNMENT-11

ENROLLMENT NO:2503A51L10

BATCH NO: 19

NAME: K.Praneeth

TASK DESCRIPTION 1:

Use AI to help implement a **Stack** class in Python with the following operations: push (), pop (), peek (), and is empty ().

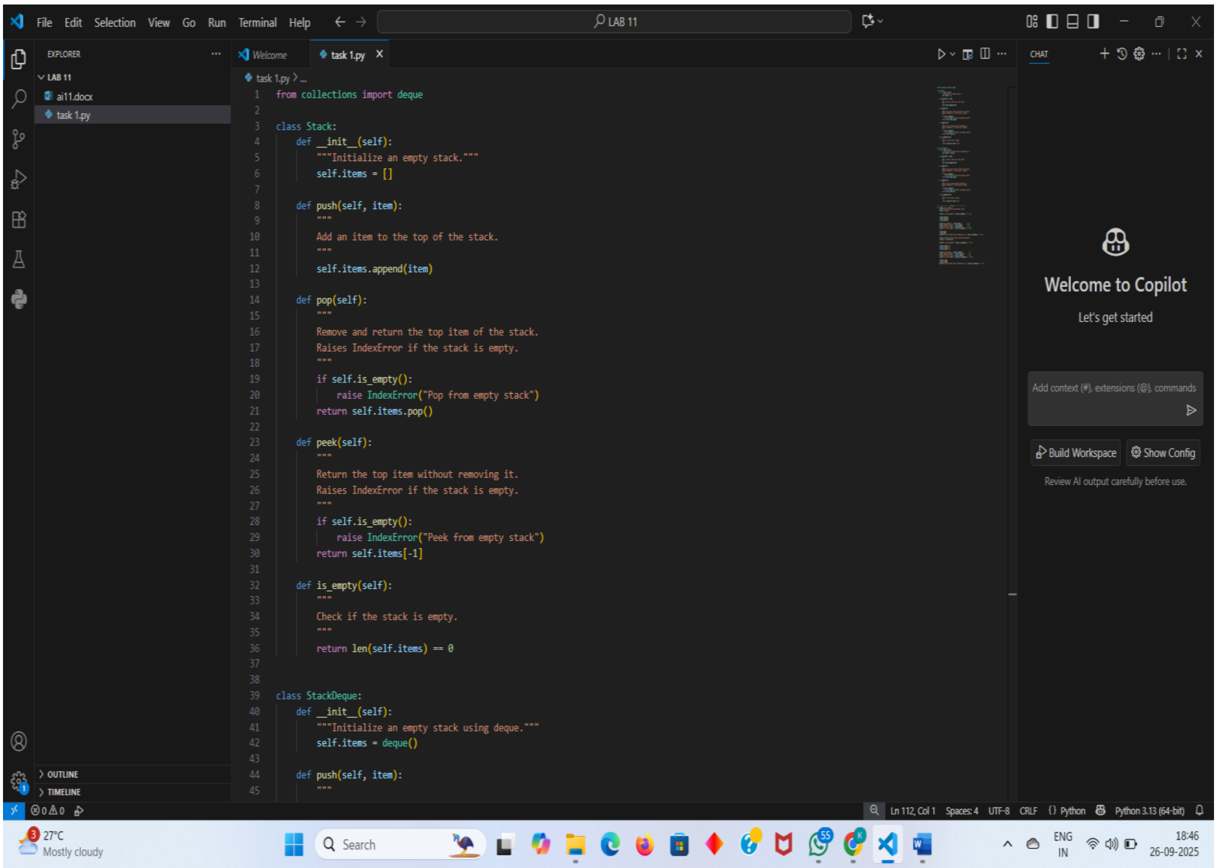
Instructions:

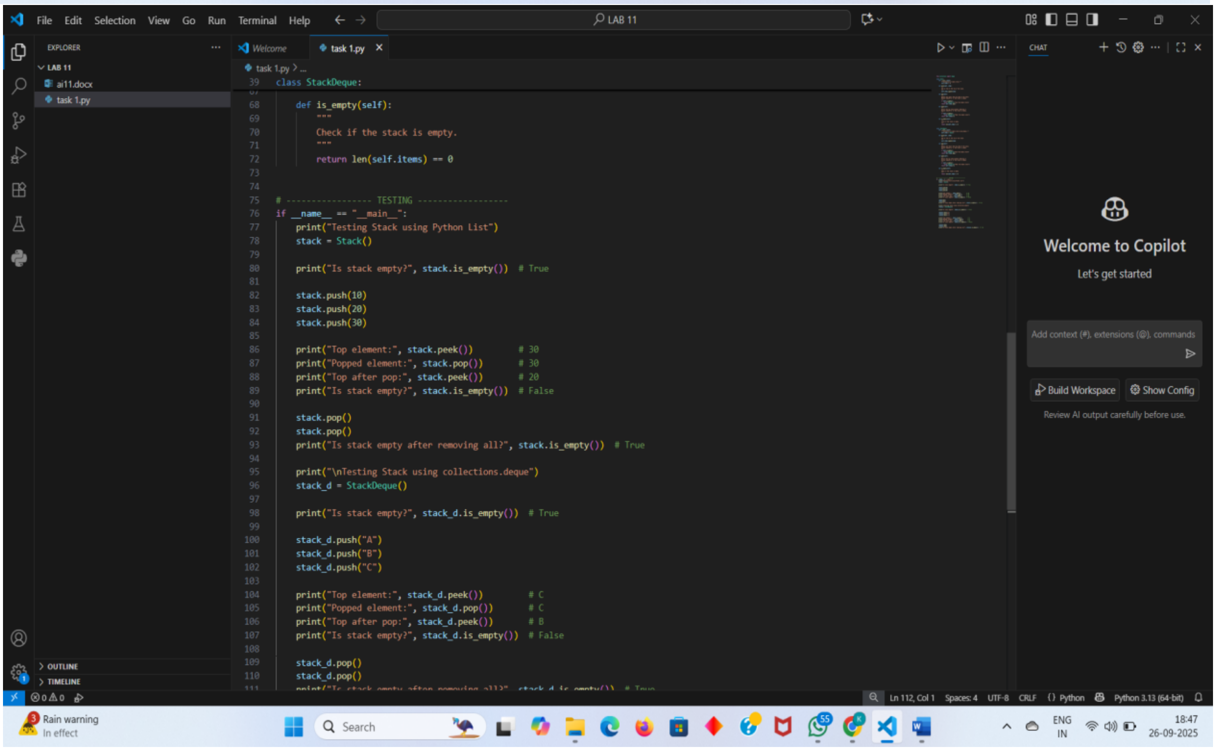
- Ask AI to generate code skeleton with docstrings.
- Test stack operations using sample data.
- Request AI to suggest optimizations or alternative implementations (e.g., using collections. Deque).

PROMPT 1:

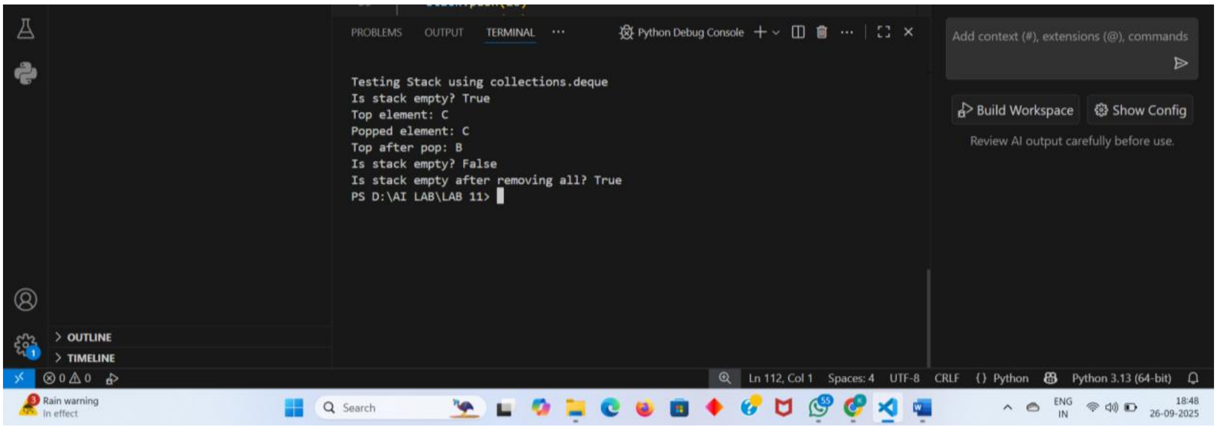
Generate a simple Stack class in Python that includes four basic operations: push () to add an item, pop () to remove the top item, peek () to view the top item without removing it, and is empty () to check if the stack has no elements. First, I'd like the AI to generate a clean code skeleton with helpful comments or docstrings explaining each method. Then, I want to test the stack using example data to make sure all the operations work correctly. Finally, I’d like the AI to suggest better ways to implement the stack, such as using Python’s collections. Deque, and explain why that might be more efficient or readable.

CODE GENERATED:





OUTPUT:



OBSERVATION:

This assignment effectively demonstrates the use of AI to support structured programming tasks. By implementing a Stack class with core operations—push (), pop (), peek (), and is empty ()—the task reinforces foundational concepts in data structures. The prompt encourages clean code practices through the use of docstrings and method documentation, while also promoting testing with sample data to validate functionality. Additionally, the request for AI-driven suggestions on alternative implementations, such as using collections. Deque, introduces learners to performance considerations and Pythonic design choices. Overall, the assignment balances technical execution with reflective learning, making it a strong exercise in both coding and code evaluation.

TASK DESCRIPTION 2:

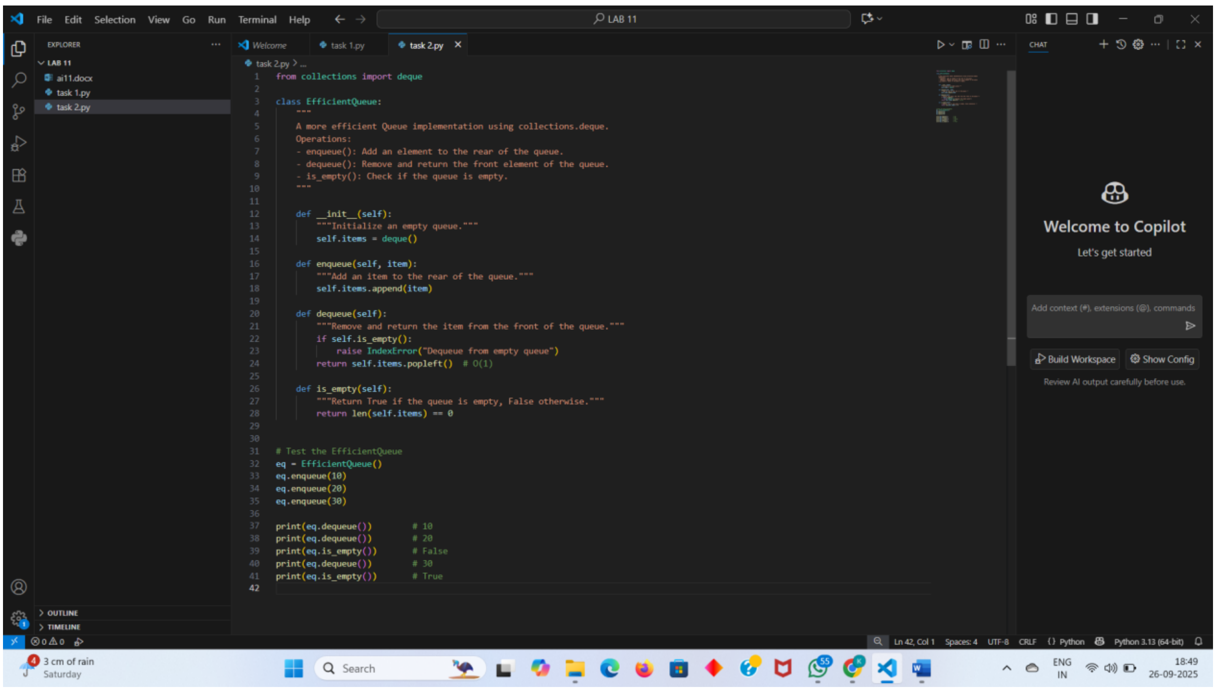
Implement a **Queue** with enqueue (), dequeue (), and is empty () methods.

- **Instructions**:
    - First, implement using Python lists.
    - Then, ask AI to review performance and suggest a more efficient implementation (using collections. Deque).

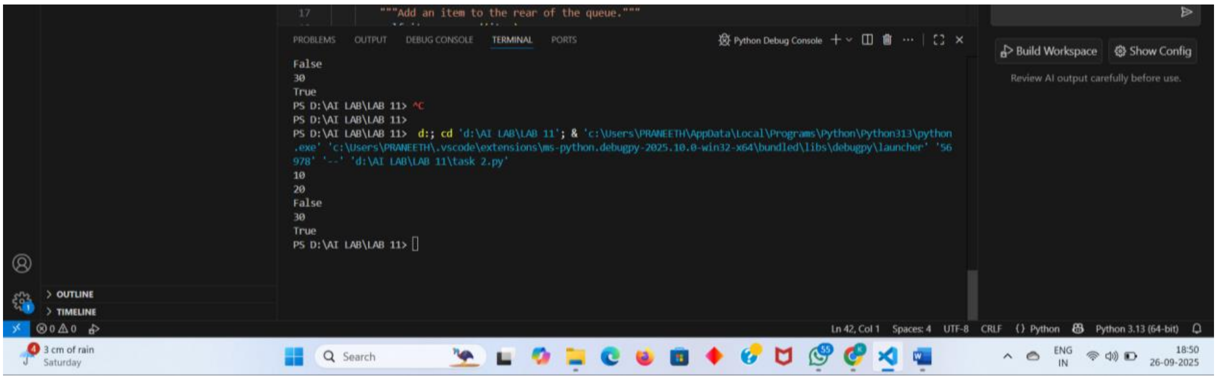
PROMPT 1:

Generate a basic Queue class in Python with three functions: enqueue () to add an item, dequeue () to remove the first item, and is empty () to check if the queue is empty. First, I'll use a regular Python list to build it. Then, I'd like the AI to look at how well this version performs and suggest a better way to write it—like using collections. Deque—and explain why that might be faster or more efficient

CODE GENERATED:



OUTPUT:



OBSERVATION:

This assignment provides a practical introduction to queue data structures and highlights the importance of performance-aware coding. By first implementing a Queue class using Python lists, it reinforces the basic logic behind enqueueing and dequeuing operations. However, it also encourages deeper thinking by asking for a performance review, which reveals the limitations of list-based queues—particularly the inefficiency of removing items from the front. The follow-up suggestion to use collections. Deque introduces learners to a more optimized and Pythonic solution, demonstrating how built-in modules can improve both speed and memory usage. Overall, the task blends hands-on coding with thoughtful evaluation, making it a strong exercise in both implementation and improvement.

TASK DESCRIPTION 3:

Implement a **Singly Linked List** with operations: instated (), delete value (), and traverse ().

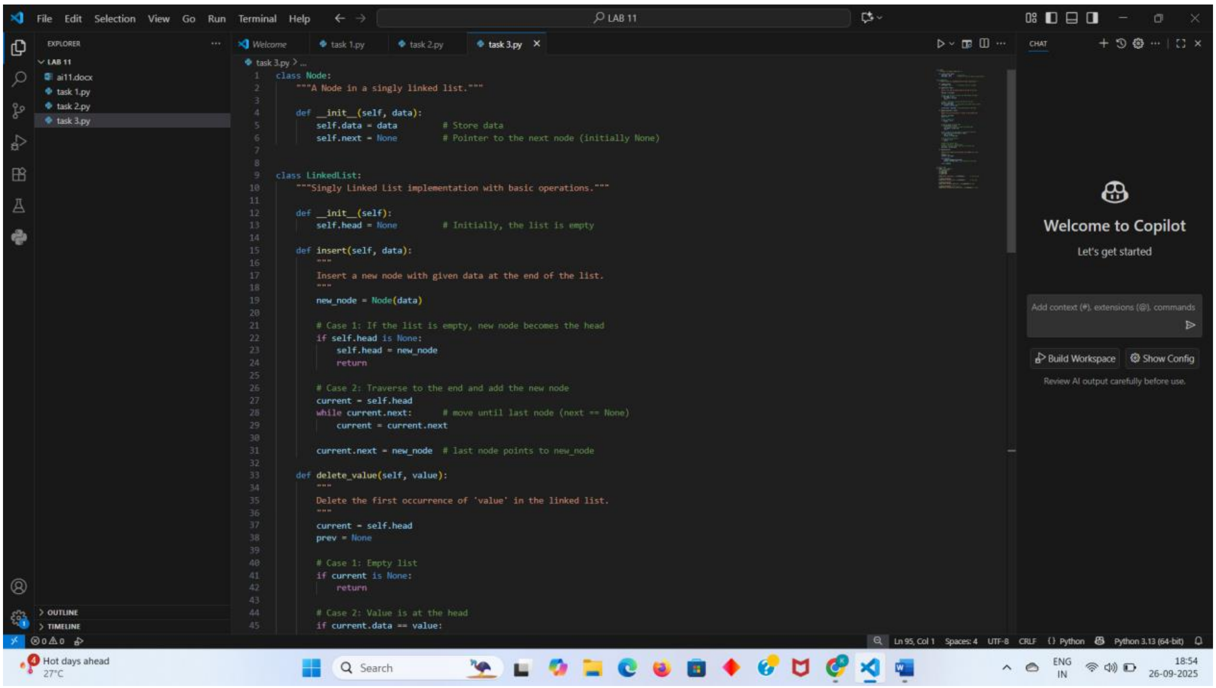
- **Instructions**:
    - Start with a simple class-based implementation (Node, LinkedList).
    - Use AI to generate inline comments explaining pointer updates (which are non-trivial).

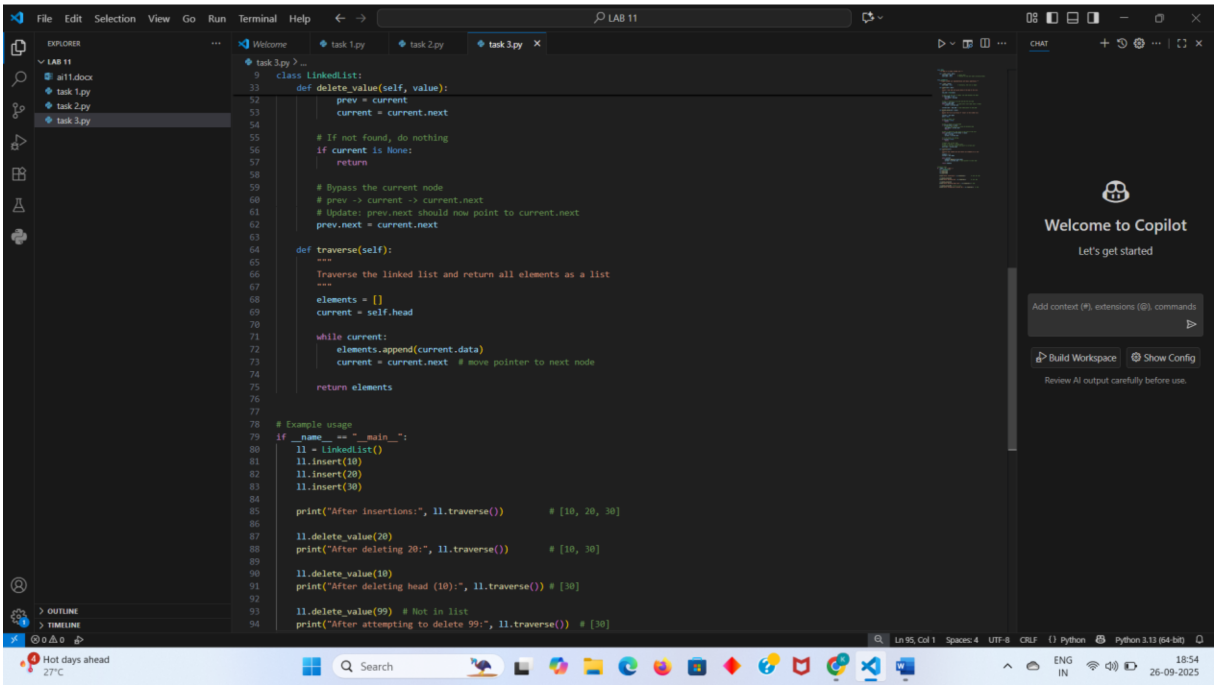
- Ask AI to suggest test cases to validate all operations.

PROMPT 1:

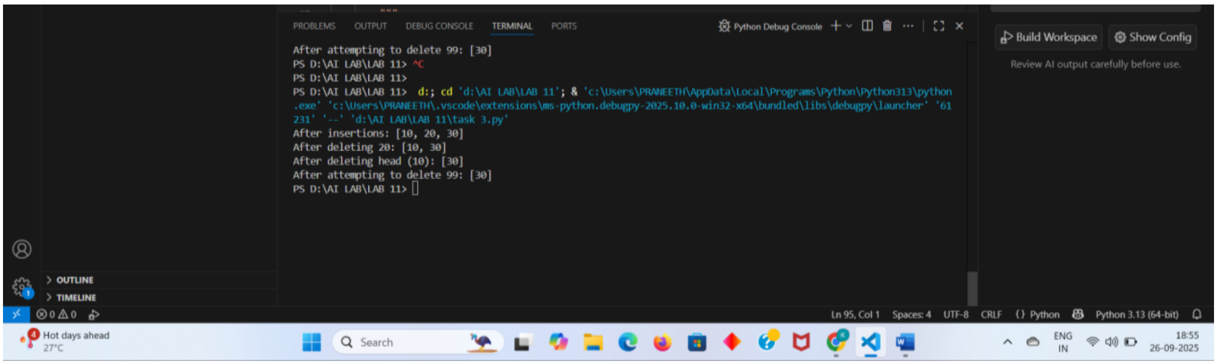
Generate a Singly Linked List in Python with three main operations: instated () to add a node at the end, delete value () to remove a node with a specific value, and traverse () to print or return all the elements in order. I'll start with a basic class-based setup using a Node class and a LinkedList class. I'd like the AI to help write inline comments that explain how pointers are updated during each operation, since that part can be tricky. Finally, I want the AI to suggest test cases to check if all the functions work correctly, including edge cases like deleting from an empty list or removing the last node.

CODE GENERATED:





OUTPUT:



OBSERVATION:

This assignment offers a hands-on approach to understanding how singly linked lists work, especially the role of pointers in dynamic data structures. By implementing operations like instated (), delete value (), and traverse (), it reinforces the concept of node manipulation and sequential access. The use of a class-based design with Node and LinkedList classes encourages clean organization and object-oriented thinking. Asking AI to generate inline comments for pointer updates adds clarity to a commonly misunderstood part of linked list logic, making the code more readable and educational. Additionally, requesting test cases—especially for edge conditions like deleting from an empty list or removing the last node—promotes thorough validation and defensive programming. Overall, the assignment blends implementation, explanation, and testing in a way that deepens both conceptual understanding and practical coding skills.

TASK DESCRIPTION 4:

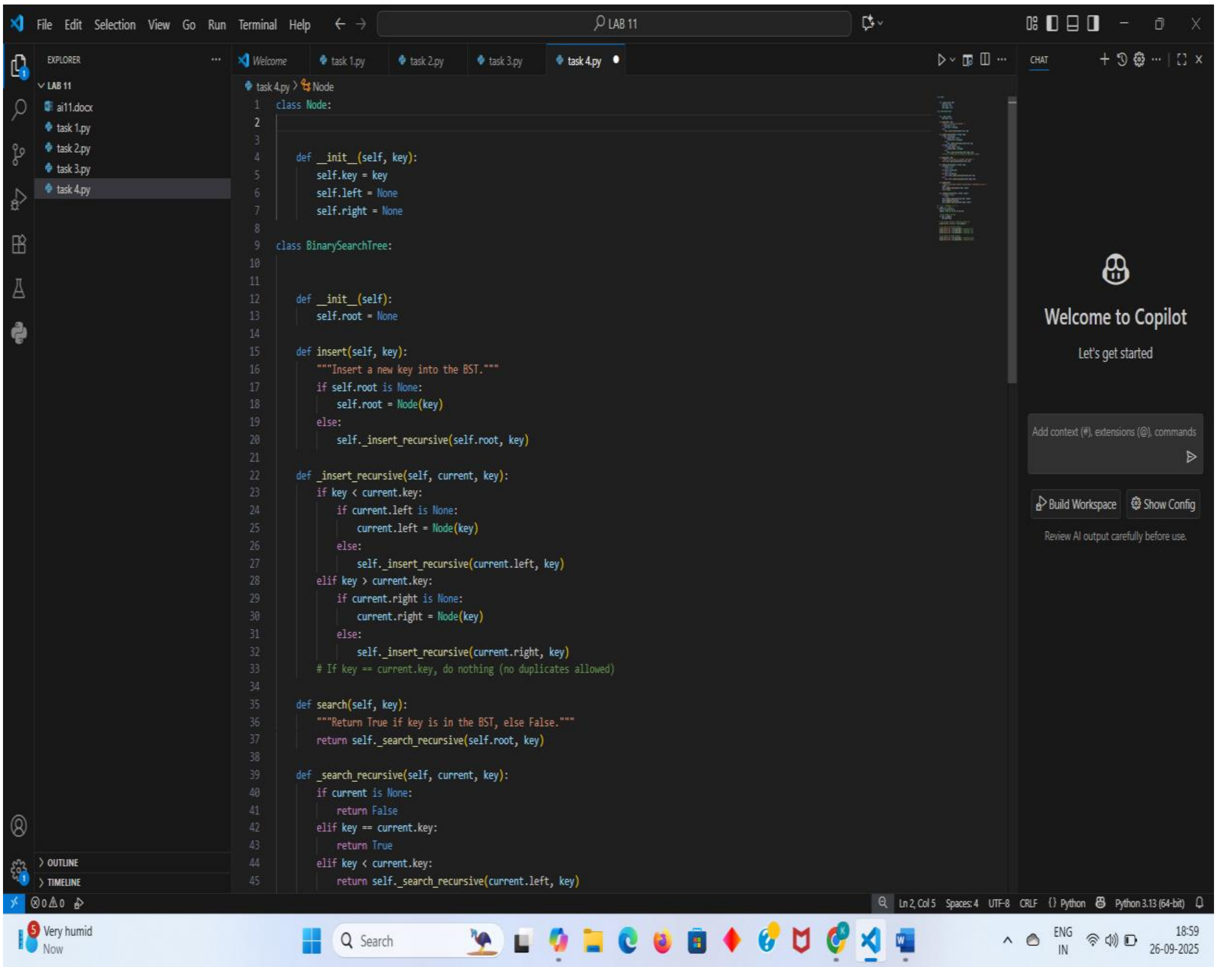
- Implement a **Binary Search Tree** with methods for insert (), search (), and inorder_traversal ().
- **Instructions**:
    - Provide AI with a partially written Node and BST class.
    - Ask AI to complete missing methods and add docstrings.

Test with a list of integers and compare outputs of search () for present vs absent elements.

PROMPT 1:

Generate binary Search Tree in Python and have already started writing the Node and BST classes. I want help from AI to complete the missing methods: insert () to add values, search () to find a value, and inorder_traversal () to return the sorted order of elements. Please add docstrings to explain what each method does. After the code is complete, I'd like to test it using a list of integers and compare the results of search () for values that are in the tree versus values that are not.

CODE GENERATED:


task 4.py
class Node:

    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:

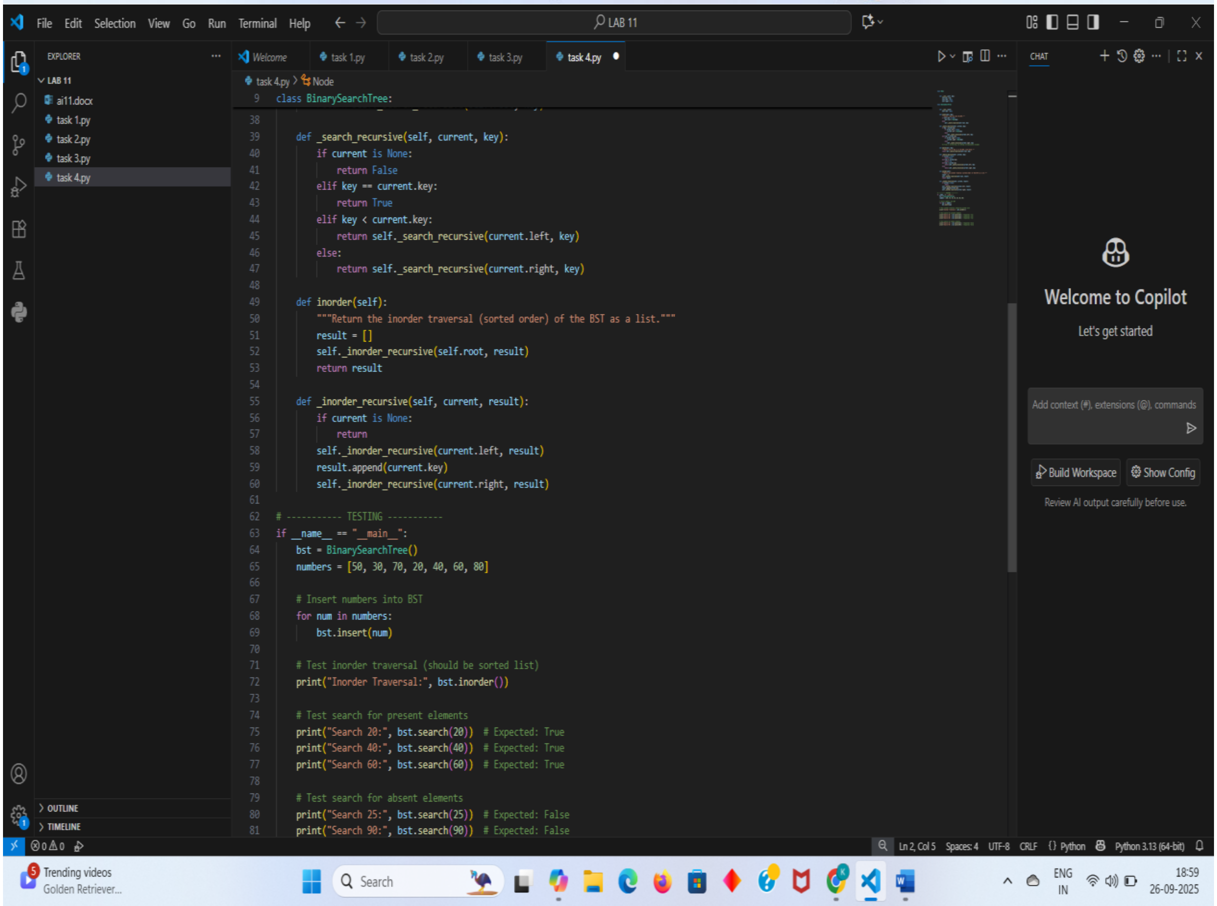
    def __init__(self):
        self.root = None

    def insert(self, key):
        """Insert a new key into the BST."""
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, current, key):
        if key < current.key:
            if current.left is None:
                current.left = Node(key)
            else:
                self._insert_recursive(current.left, key)
        elif key > current.key:
            if current.right is None:
                current.right = Node(key)
            else:
                self._insert_recursive(current.right, key)
        # If key == current.key, do nothing (no duplicates allowed)

    def search(self, key):
        """Return True if key is in the BST, else False."""
        return self._search_recursive(self.root, key)

    def _search_recursive(self, current, key):
        if current is None:
            return False
        elif key == current.key:
            return True
        elif key < current.key:
            return self._search_recursive(current.left, key)
Welcome to Copilot
Let's get started


task 4.py
class BinarySearchTree:

    def _search_recursive(self, current, key):
        if current is None:
            return False
        elif key == current.key:
            return True
        elif key < current.key:
            return self._search_recursive(current.left, key)
        else:
            return self._search_recursive(current.right, key)

    def inorder(self):
        """Return the inorder traversal (sorted order) of the BST as a list."""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, current, result):
        if current is None:
            return
        self._inorder_recursive(current.left, result)
        result.append(current.key)
        self._inorder_recursive(current.right, result)

# ---------- TESTING ----------
if __name__ == "__main__":
    bst = BinarySearchTree()
    numbers = [50, 30, 70, 20, 40, 60, 80]

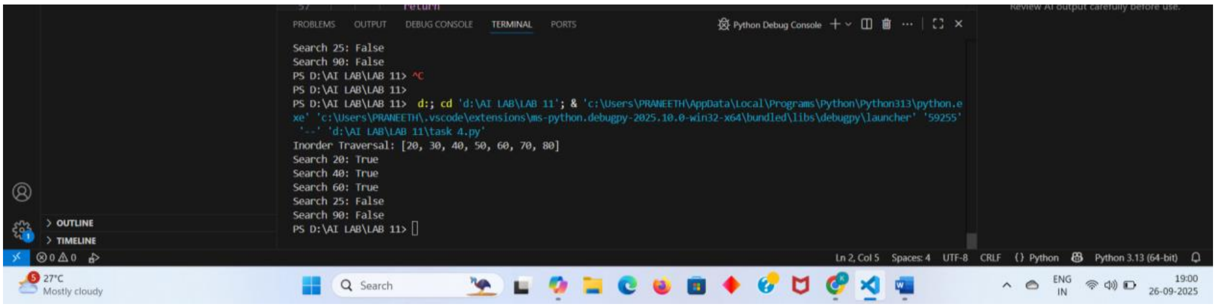
    # Insert numbers into BST
    for num in numbers:
        bst.insert(num)

    # Test inorder traversal (should be sorted list)
    print("Inorder Traversal:", bst.inorder())

    # Test search for present elements
    print("Search 20:", bst.search(20)) # Expected: True
    print("Search 40:", bst.search(40)) # Expected: True
    print("Search 60:", bst.search(60)) # Expected: True

    # Test search for absent elements
    print("Search 25:", bst.search(25)) # Expected: False
    print("Search 90:", bst.search(90)) # Expected: False
Welcome to Copilot
Let's get started

OUTPUT:



OBSERVATION:

This assignment offers a solid introduction to recursive data structures and algorithmic thinking through the implementation of a Binary Search Tree (BST). By starting with a partially written Node and BST class, it encourages learners to focus on completing core methods—insert (), search (), and inorder_traversal ()—which are fundamental to understanding tree-based logic. The use of AI to fill in missing code and generate docstrings promotes clarity and reinforces best practices in documentation. Testing the tree with a list of integers and comparing search results for present versus absent values adds a layer of validation and encourages attention to edge cases. Overall, the assignment blends hands-on coding, conceptual understanding, and reflective testing, making it a well-rounded exercise in both implementation and evaluation.

TASK DESCRIPTION 5:

Implement a **Graph** using an adjacency list, with traversal methods BFS () and DFS ().

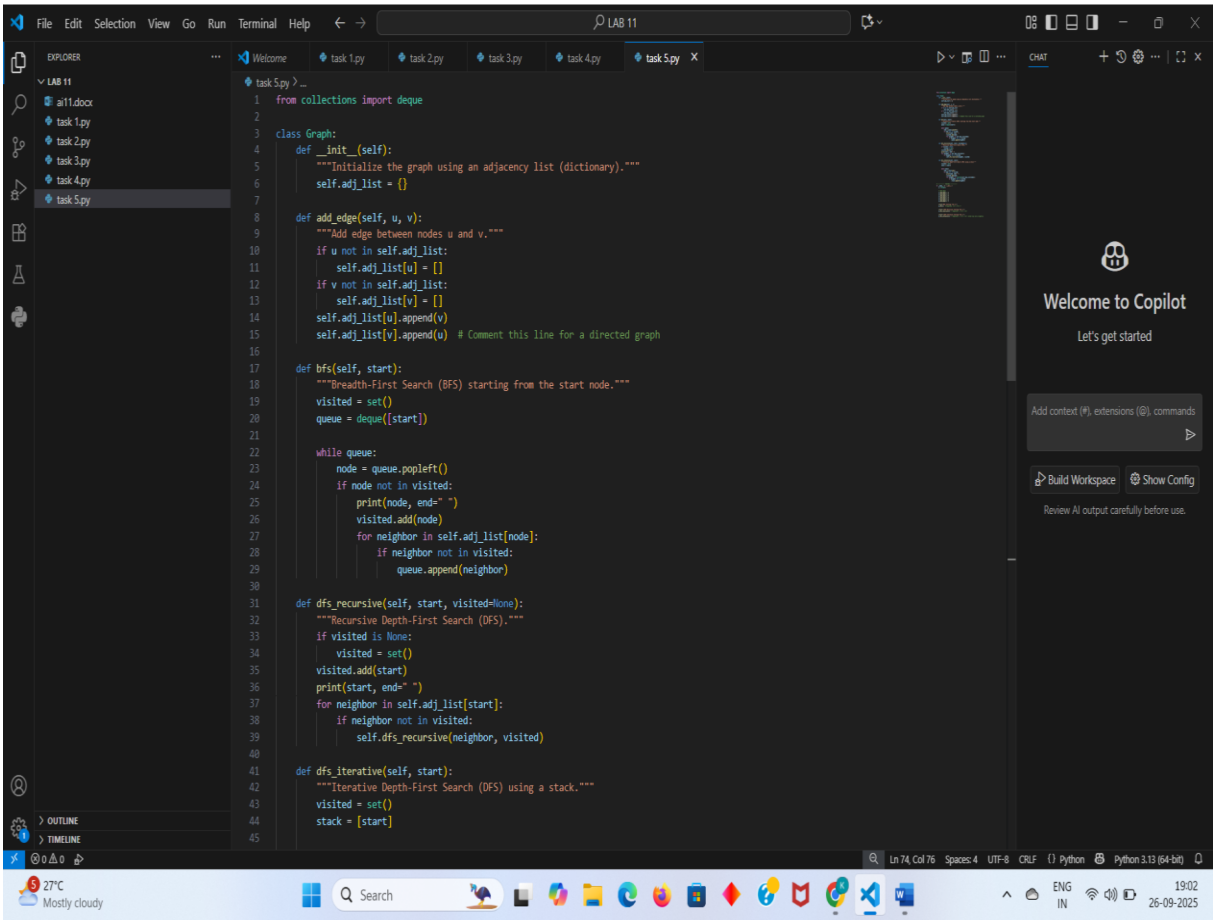
- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.

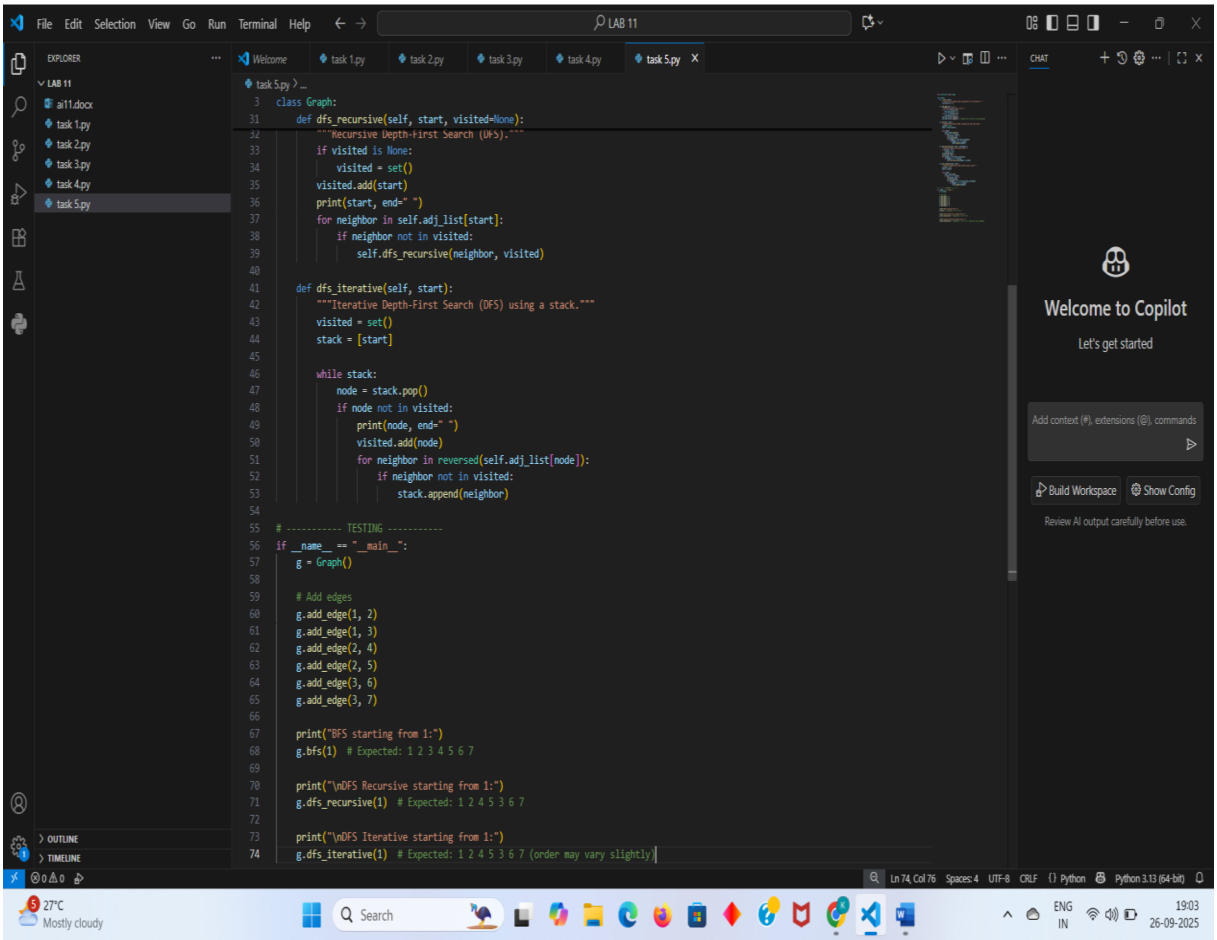
- Compare recursive vs iterative DFS if suggested by AI.

PROMPT 1:

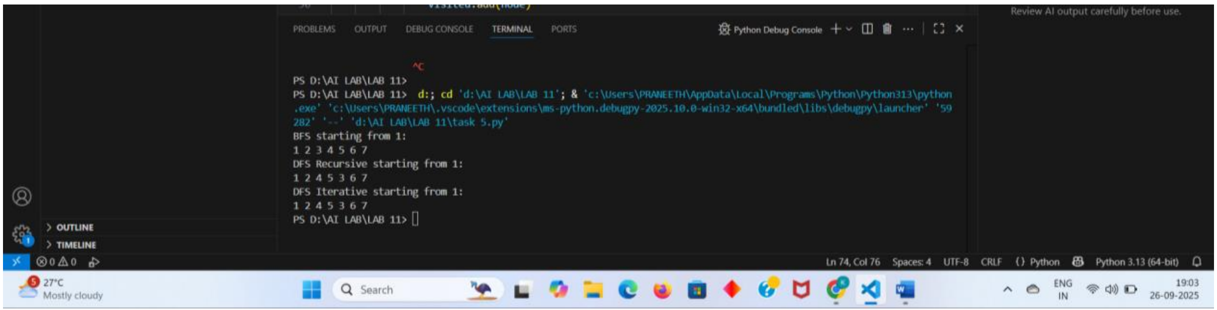
Generate a Graph in Python using an adjacency list represented as a dictionary. I need help from AI to write the traversal methods: BFS () for breadth-first search and DFS () for depth-first search. Please include inline comments that explain how each part of the traversal works, especially how nodes are visited and tracked. If possible, I'd also like to compare recursive and iterative versions of DFS, and understand which one is better in terms of readability or performance.

CODE GENERATED:





OUTPUT:



OBSERVATION:

This assignment provides a strong foundation for understanding graph traversal techniques using Python. By implementing a graph with an adjacency list, learners engage with a widely used and memory-efficient representation of graph structures. The inclusion of both BFS (Breadth-First Search) and DFS (Depth-First Search) encourages exploration of different traversal strategies, each with distinct use cases and behaviours. Requesting inline comments for each step of the traversal helps clarify how nodes are visited, queued, or stacked—making the logic more transparent and easier to follow. The comparison between recursive and iterative DFS adds a valuable layer of analysis, prompting reflection on performance trade-offs and code readability. Overall, this assignment blends implementation, explanation, and evaluation, making it a well-rounded exercise in algorithmic thinking and practical coding.