ASSIGNMENT – 11

NAME: G. Bala Varshitha

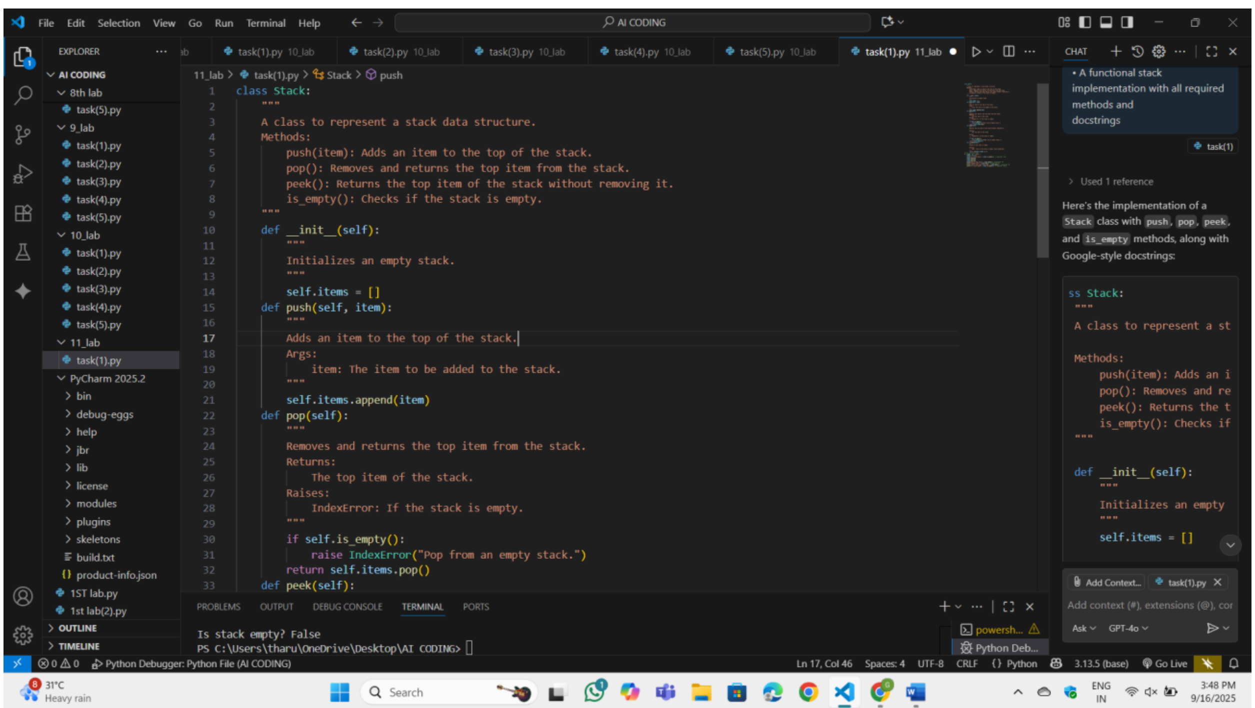
HT.NO: 2403A52050

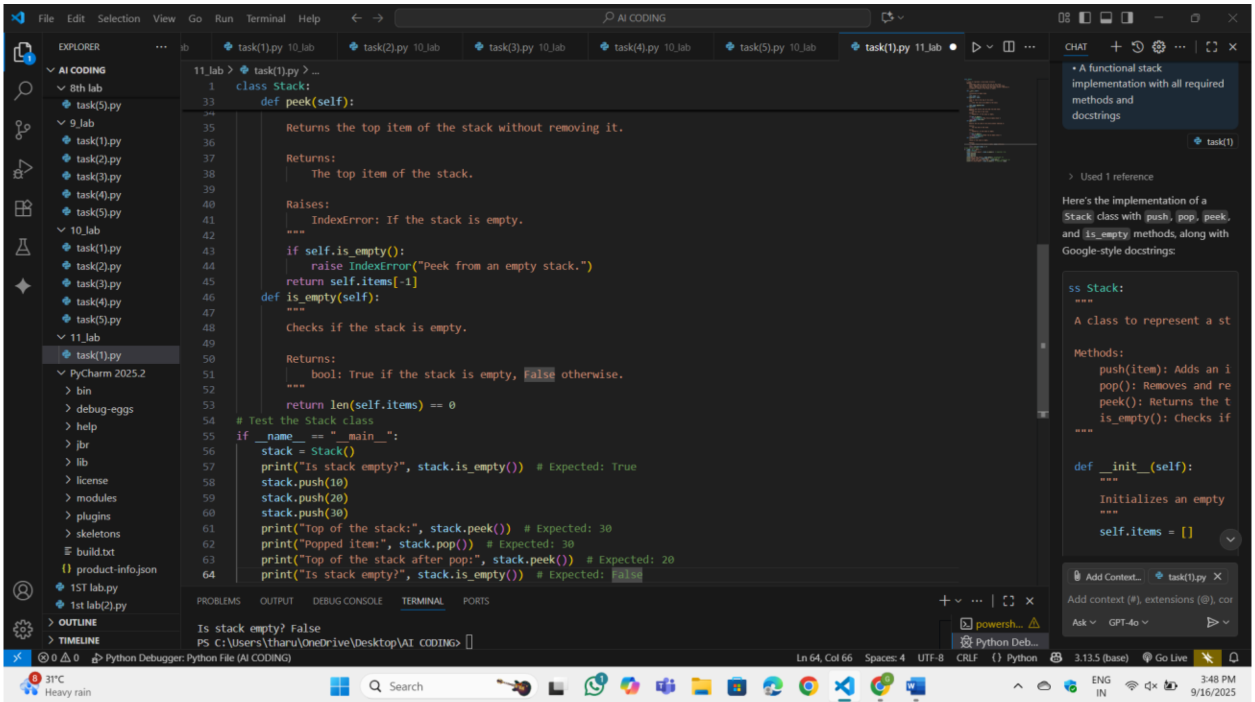
BATCH: AIB03

Task-1

Prompt: generate a Stack class with push, pop, peek, and is_empty methods.

Code:





OP:

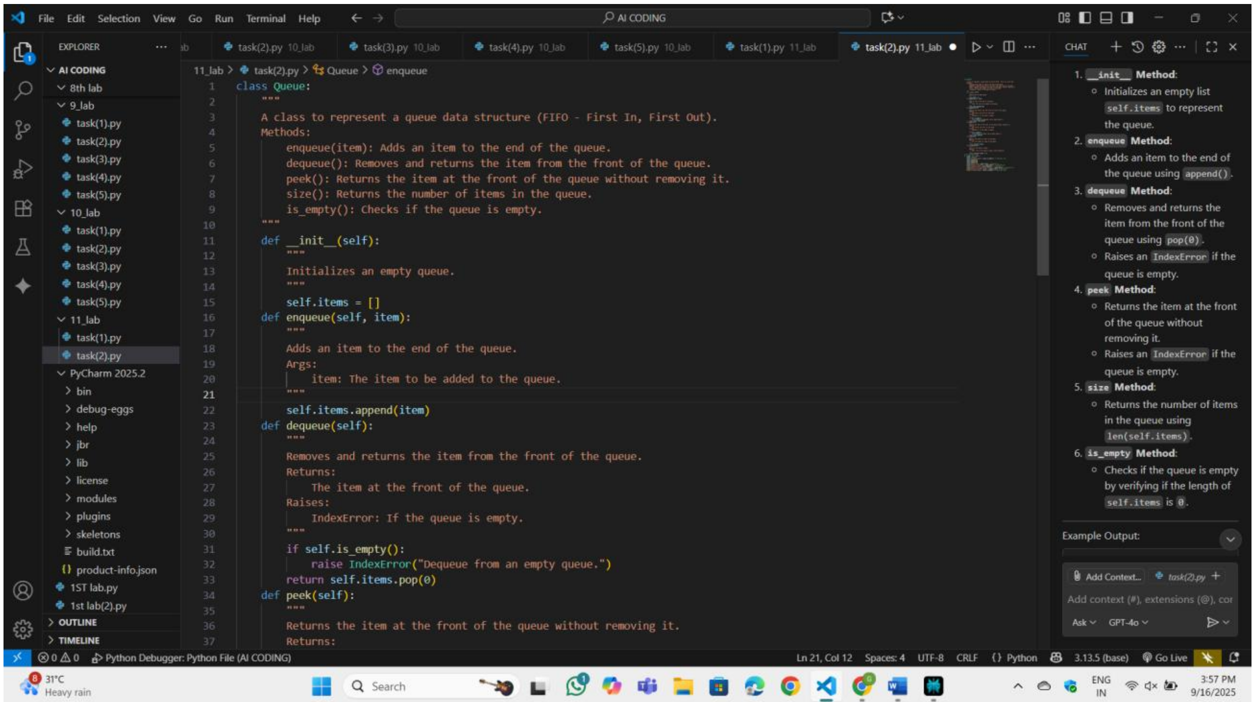
Observation:

In this program at the beginning, the collection is empty. Whenever something new is added, it goes on the top, and if something needs to be removed, the new one which was added comes out first. We can just see what is on top without removing it. We can even see if nothing is inside.

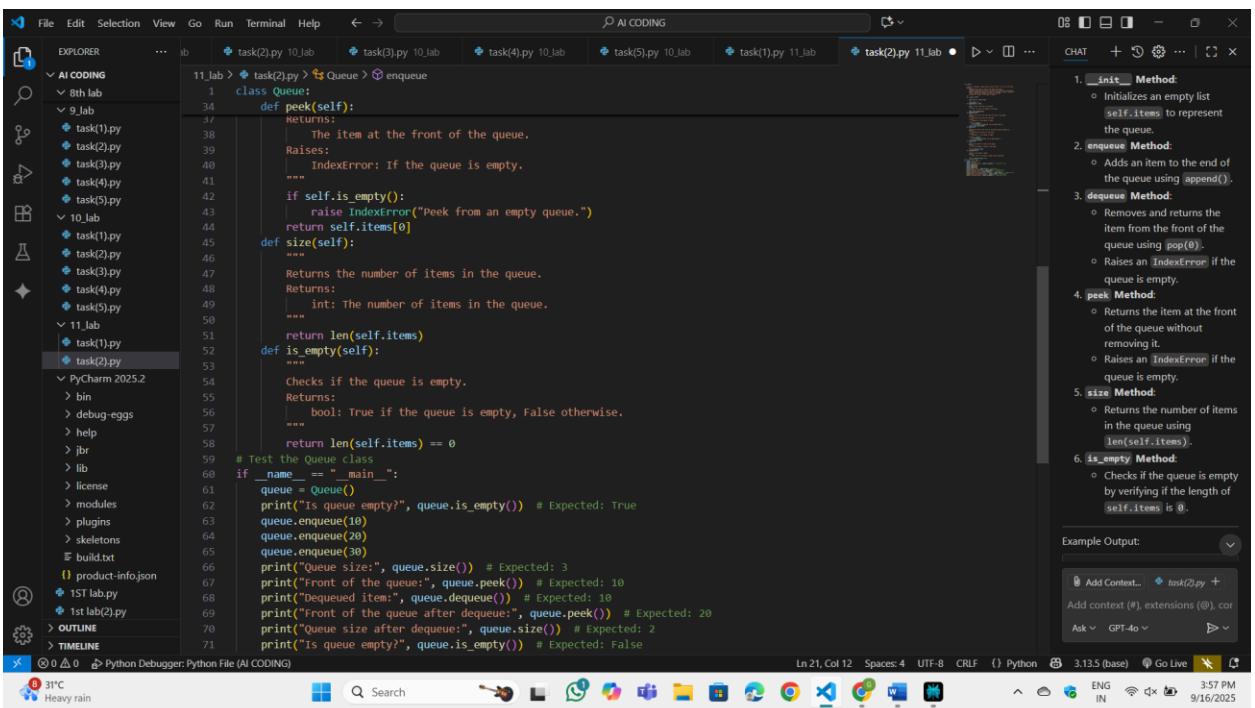
Task-2

Prompt: implement a Queue using Python lists.

Code:



class Queue:
    """
    A class to represent a queue data structure (FIFO - First In, First Out).
    Methods:
        enqueue(item): Adds an item to the end of the queue.
        dequeue(): Removes and returns the item from the front of the queue.
        peek(): Returns the item at the front of the queue without removing it.
        size(): Returns the number of items in the queue.
        is_empty(): Checks if the queue is empty.
    """
    def __init__(self):
        """
        Initializes an empty queue.
        """
        self.items = []
    def enqueue(self, item):
        """
        Adds an item to the end of the queue.
        Args:
            item: The item to be added to the queue.
        """
        self.items.append(item)
    def dequeue(self):
        """
        Removes and returns the item from the front of the queue.
        Returns:
            The item at the front of the queue.
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue.")
        return self.items.pop(0)
    def peek(self):
        """
        Returns the item at the front of the queue without removing it.
        Returns:



class Queue:
    def peek(self):
        Returns:
            The item at the front of the queue.
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty queue.")
        return self.items[0]
    def size(self):
        """
        Returns the number of items in the queue.
        Returns:
            int: The number of items in the queue.
        """
        return len(self.items)
    def is_empty(self):
        """
        Checks if the queue is empty.
        Returns:
            bool: True if the queue is empty, False otherwise.
        """
        return len(self.items) == 0
# Test the Queue class
if __name__ == "__main__":
    queue = Queue()
    print("Is queue empty?", queue.is_empty()) # Expected: True
    queue.enqueue(10)
    queue.enqueue(20)
    queue.enqueue(30)
    print("Queue size:", queue.size()) # Expected: 3
    print("Front of the queue:", queue.peek()) # Expected: 10
    print("Dequeued item:", queue.dequeue()) # Expected: 10
    print("Front of the queue after dequeue:", queue.peek()) # Expected: 20
    print("Queue size after dequeue:", queue.size()) # Expected: 2
    print("Is queue empty?", queue.is_empty()) # Expected: False

OP:

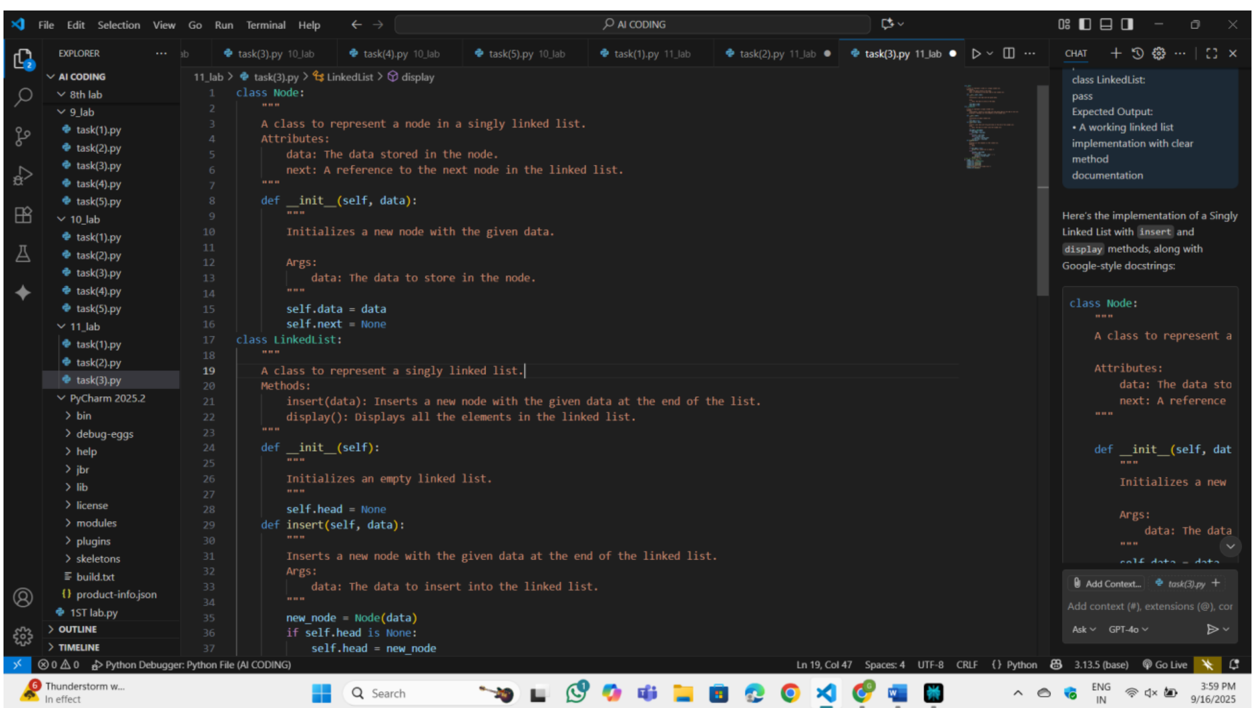
Observation:

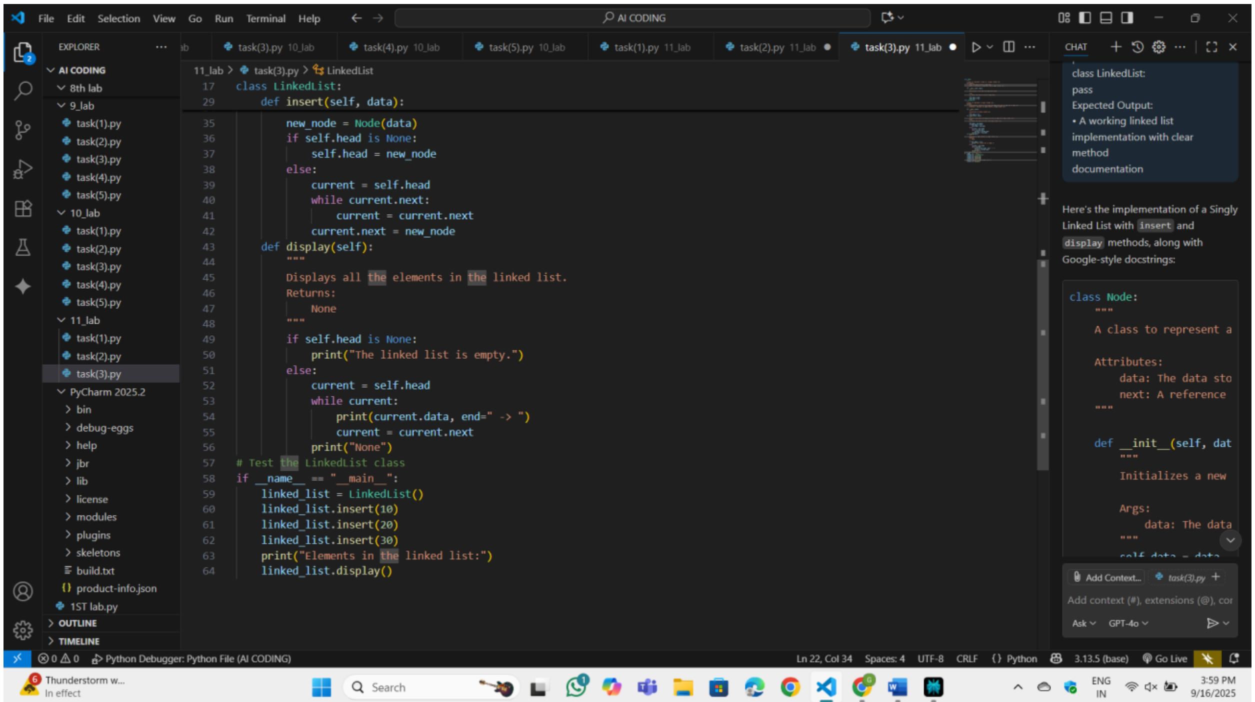
In this program at the beginning, the line is empty. Whenever something new comes, it joins at the end of the line, and when something leaves, it is always the first one that came in. You can also just look at who is at the front without removing them. There's a way to count how many are currently in the line, and also to check if the line is completely empty.

Task-3:

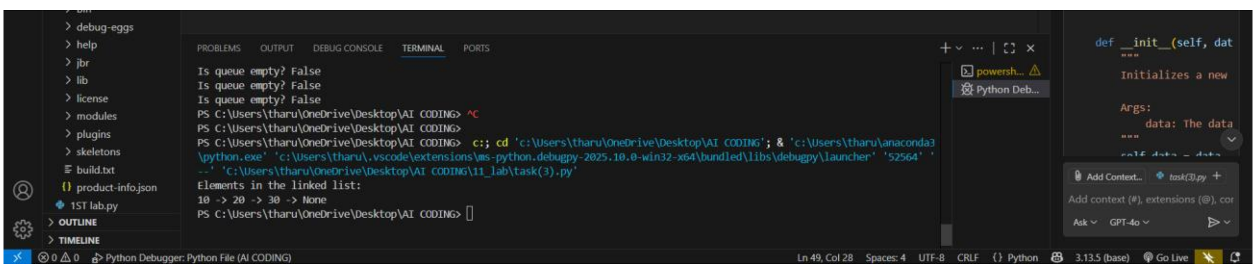
Prompt: generate a Singly Linked List with insert and display methods.

Code:





OP:



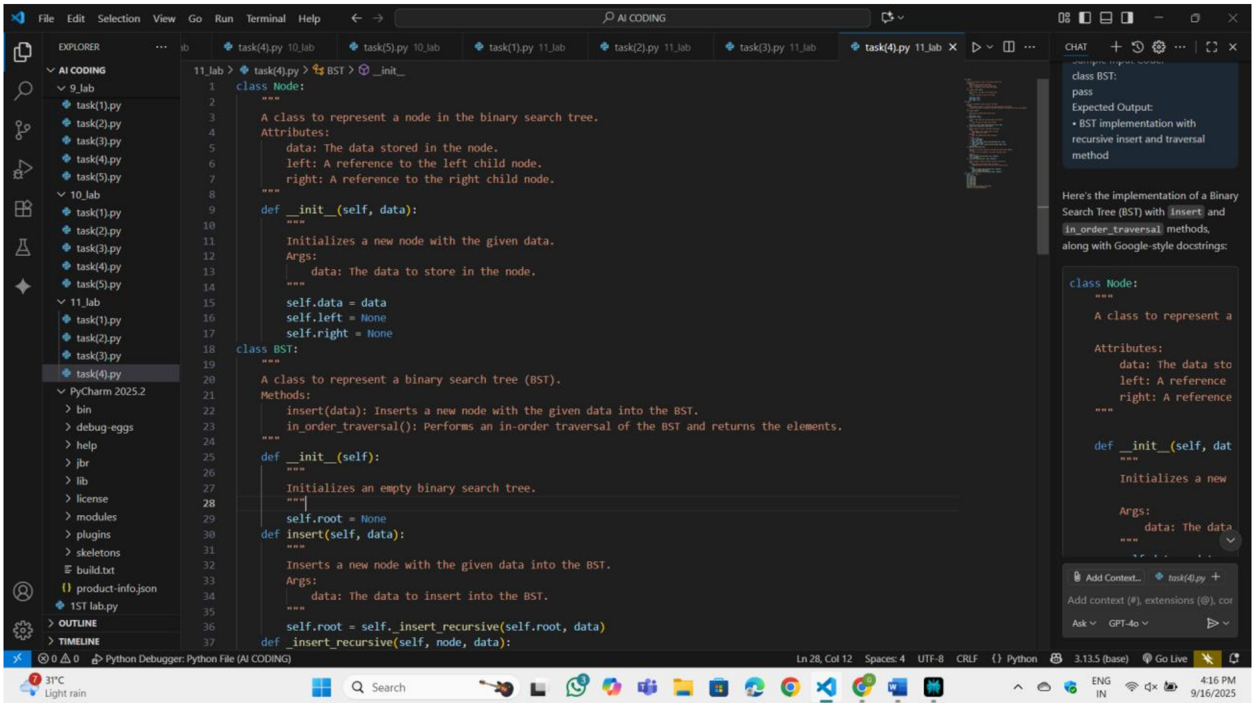
Observation:

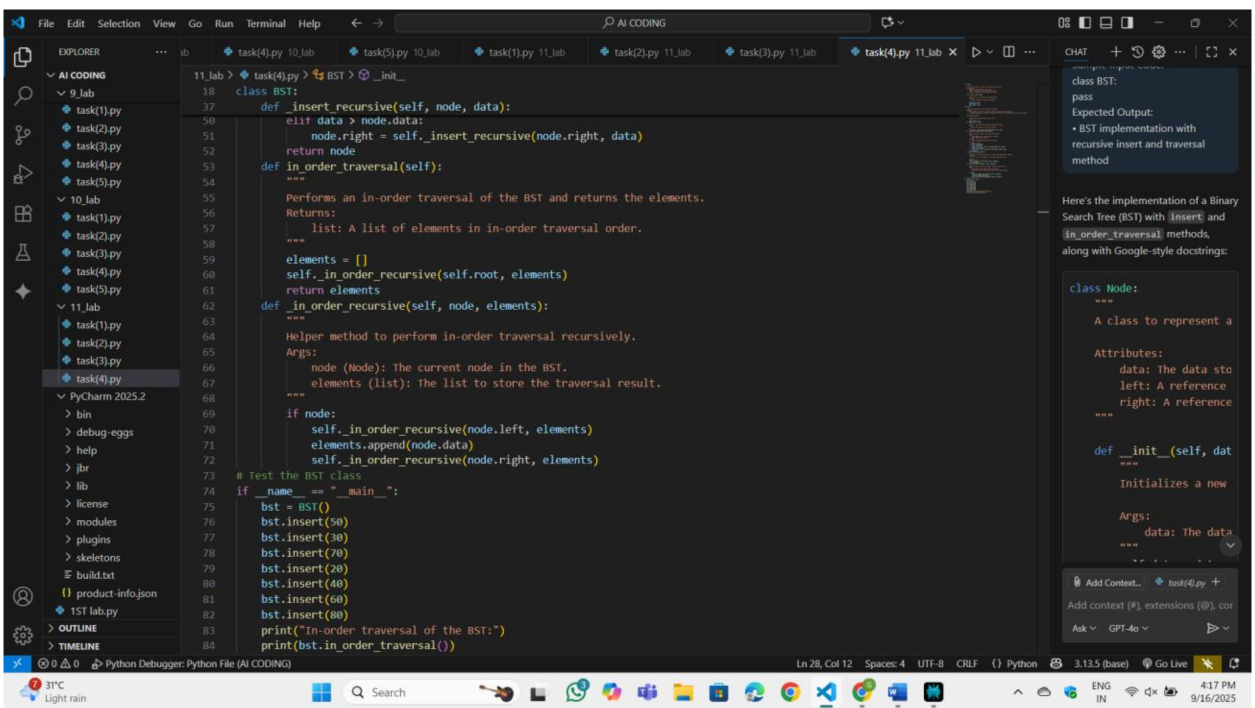
In this program at the beginning, the line is empty. Whenever something new comes, it joins at the end of the line, and when something leaves, it is always the first one that came in. You can also just look at who is at the front without removing them. There's a way to count how many are currently in the line, and also to check if the line is completely empty.

Task-4:

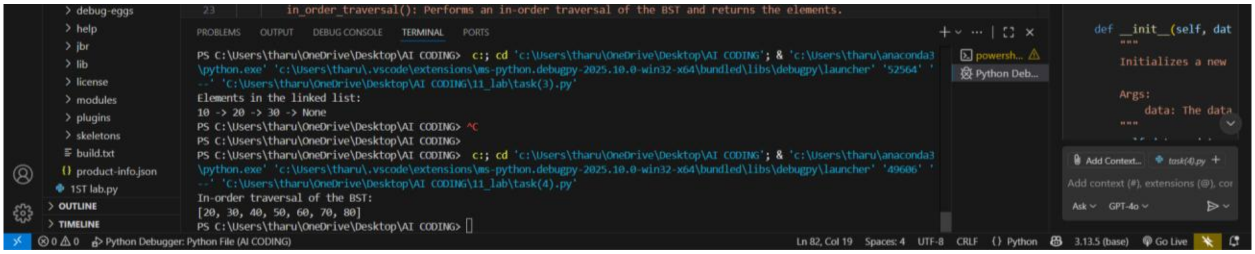
Prompt: create a BST with insert and in-order traversal methods.

Code:


class Node:
"""
A class to represent a node in the binary search tree.
Attributes:
data: The data stored in the node.
left: A reference to the left child node.
right: A reference to the right child node.
"""
def __init__(self, data):
"""
Initializes a new node with the given data.
Args:
data: The data to store in the node.
"""
self.data = data
self.left = None
self.right = None
class BST:
"""
A class to represent a binary search tree (BST).
Methods:
insert(data): Inserts a new node with the given data into the BST.
in_order_traversal(): Performs an in-order traversal of the BST and returns the elements.
"""
def __init__(self):
"""
Initializes an empty binary search tree.
"""
self.root = None
def insert(self, data):
"""
Inserts a new node with the given data into the BST.
Args:
data: The data to insert into the BST.
"""
self.root = self._insert_recursive(self.root, data)
def _insert_recursive(self, node, data):


class BST:
def _insert_recursive(self, node, data):
elif data > node.data:
node.right = self._insert_recursive(node.right, data)
return node
def in_order_traversal(self):
"""
Performs an in-order traversal of the BST and returns the elements.
Returns:
list: A list of elements in in-order traversal order.
"""
elements = []
self._in_order_recursive(self.root, elements)
return elements
def _in_order_recursive(self, node, elements):
"""
Helper method to perform in-order traversal recursively.
Args:
node (Node): The current node in the BST.
elements (list): The list to store the traversal result.
"""
if node:
self._in_order_recursive(node.left, elements)
elements.append(node.data)
self._in_order_recursive(node.right, elements)
# Test the BST class
if __name__ == "__main__":
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
print("In-order traversal of the BST:")
print(bst.in_order_traversal())

OP:



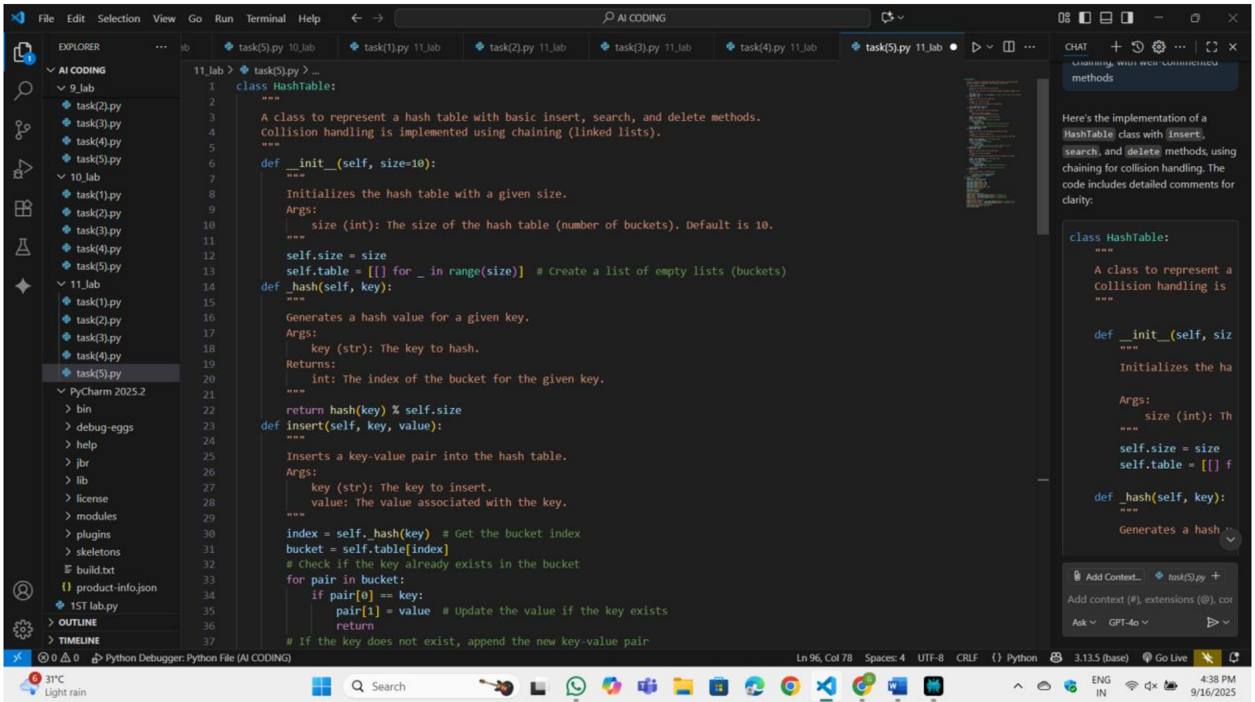
Observation:

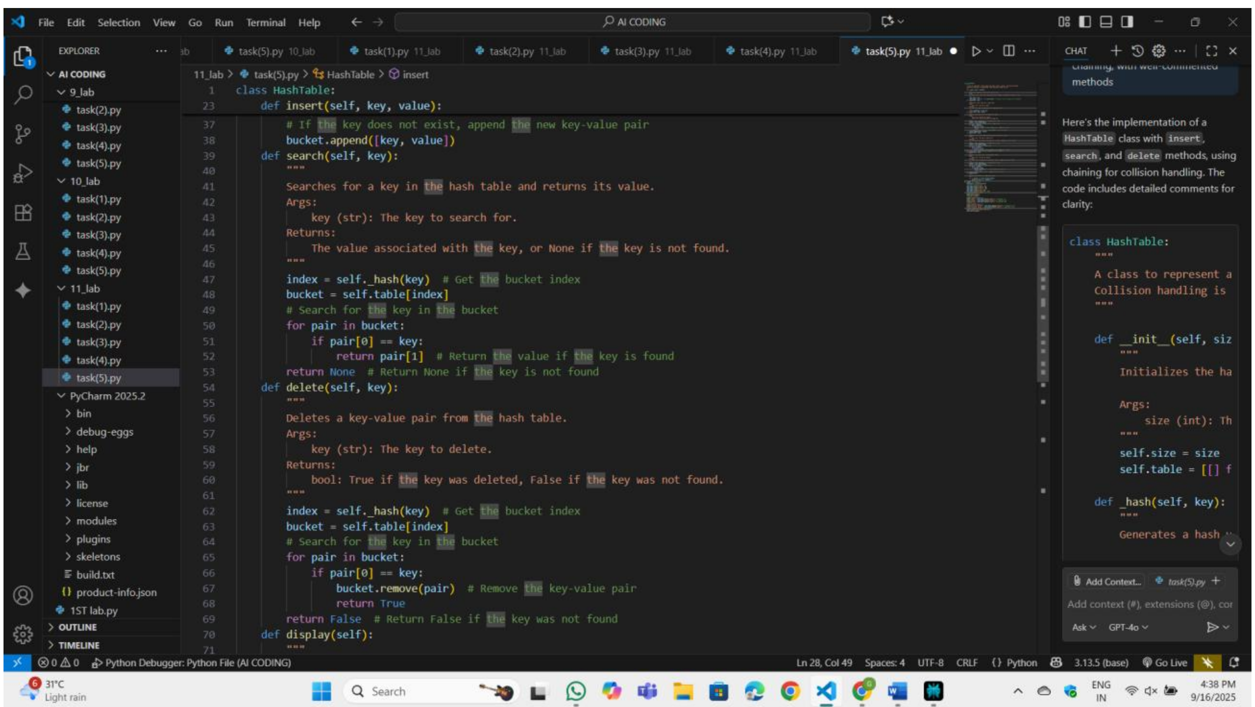
This program is about creating and organizing a tree-like structure where each piece of data is stored in special boxes called nodes. Every node has a value, along with links that can connect to smaller nodes on the left and larger nodes on the right. When a new value is added, it is placed in the proper position by comparing it with existing values until it finds its correct spot. There is also a way to go through the tree in order, which means visiting the left side first, then the main value, and finally the right side, so all the values come out sorted.

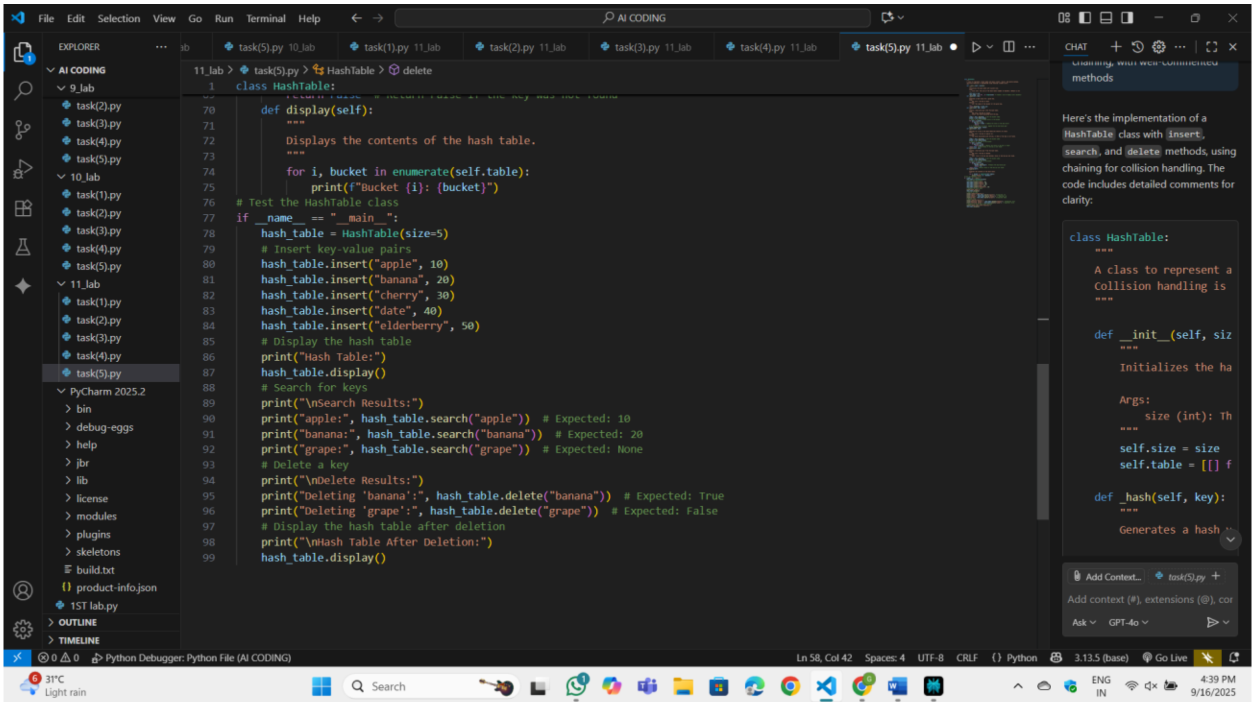
Task-5:

Prompt: implement a hash table with basic insert, search, and delete.

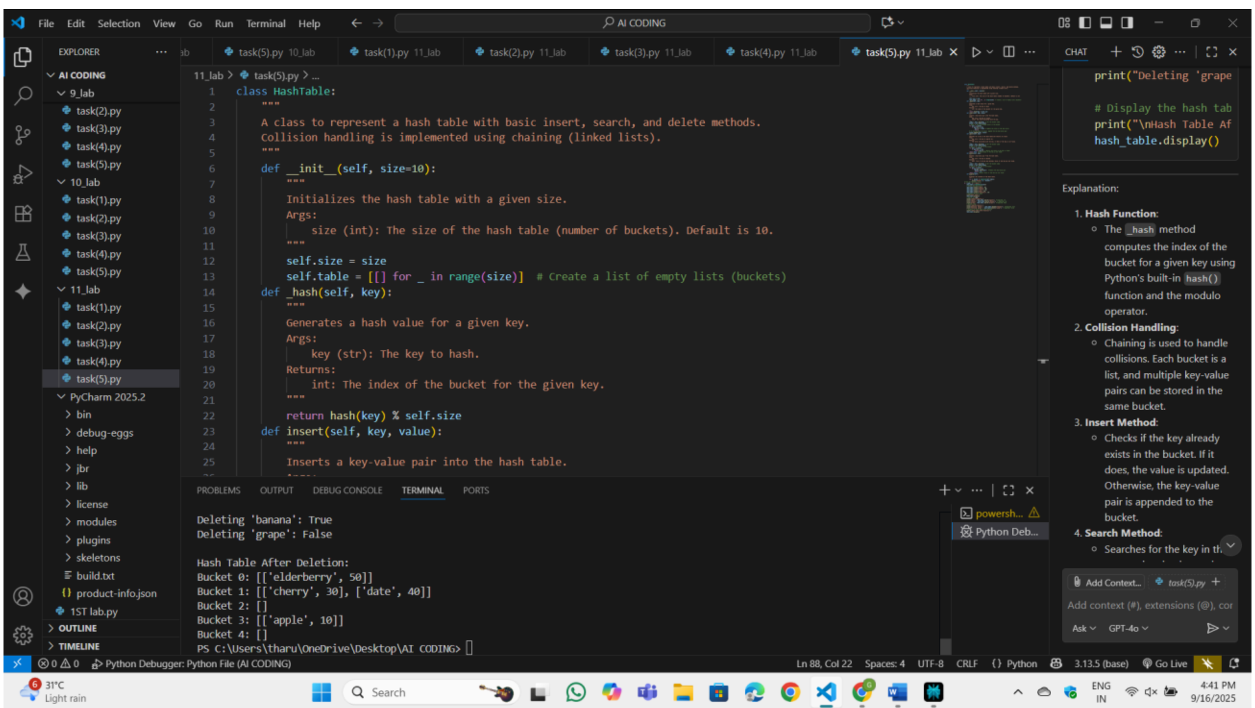
Code:


class HashTable:
    """
    A class to represent a hash table with basic insert, search, and delete methods.
    Collision handling is implemented using chaining (linked lists).
    """
    def __init__(self, size=10):
        """
        Initializes the hash table with a given size.
        Args:
            size (int): The size of the hash table (number of buckets). Default is 10.
        """
        self.size = size
        self.table = [[] for _ in range(size)] # Create a list of empty lists (buckets)
    def _hash(self, key):
        """
        Generates a hash value for a given key.
        Args:
            key (str): The key to hash.
        Returns:
            int: The index of the bucket for the given key.
        """
        return hash(key) % self.size
    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        Args:
            key (str): The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key) # Get the bucket index
        bucket = self.table[index]
        # Check if the key already exists in the bucket
        for pair in bucket:
            if pair[0] == key:
                pair[1] = value # Update the value if the key exists
                return
        # If the key does not exist, append the new key-value pair
Here's the implementation of a HashTable class with insert, search, and delete methods, using chaining for collision handling. The code includes detailed comments for clarity:


        # If the key does not exist, append the new key-value pair
        bucket.append([key, value])
    def search(self, key):
        """
        Searches for a key in the hash table and returns its value.
        Args:
            key (str): The key to search for.
        Returns:
            The value associated with the key, or None if the key is not found.
        """
        index = self._hash(key) # Get the bucket index
        bucket = self.table[index]
        # Search for the key in the bucket
        for pair in bucket:
            if pair[0] == key:
                return pair[1] # Return the value if the key is found
        return None # Return None if the key is not found
    def delete(self, key):
        """
        Deletes a key-value pair from the hash table.
        Args:
            key (str): The key to delete.
        Returns:
            bool: True if the key was deleted, False if the key was not found.
        """
        index = self._hash(key) # Get the bucket index
        bucket = self.table[index]
        # Search for the key in the bucket
        for pair in bucket:
            if pair[0] == key:
                bucket.remove(pair) # Remove the key-value pair
                return True
        return False # Return False if the key was not found
    def display(self):
Here's the implementation of a HashTable class with insert, search, and delete methods, using chaining for collision handling. The code includes detailed comments for clarity:



OP:



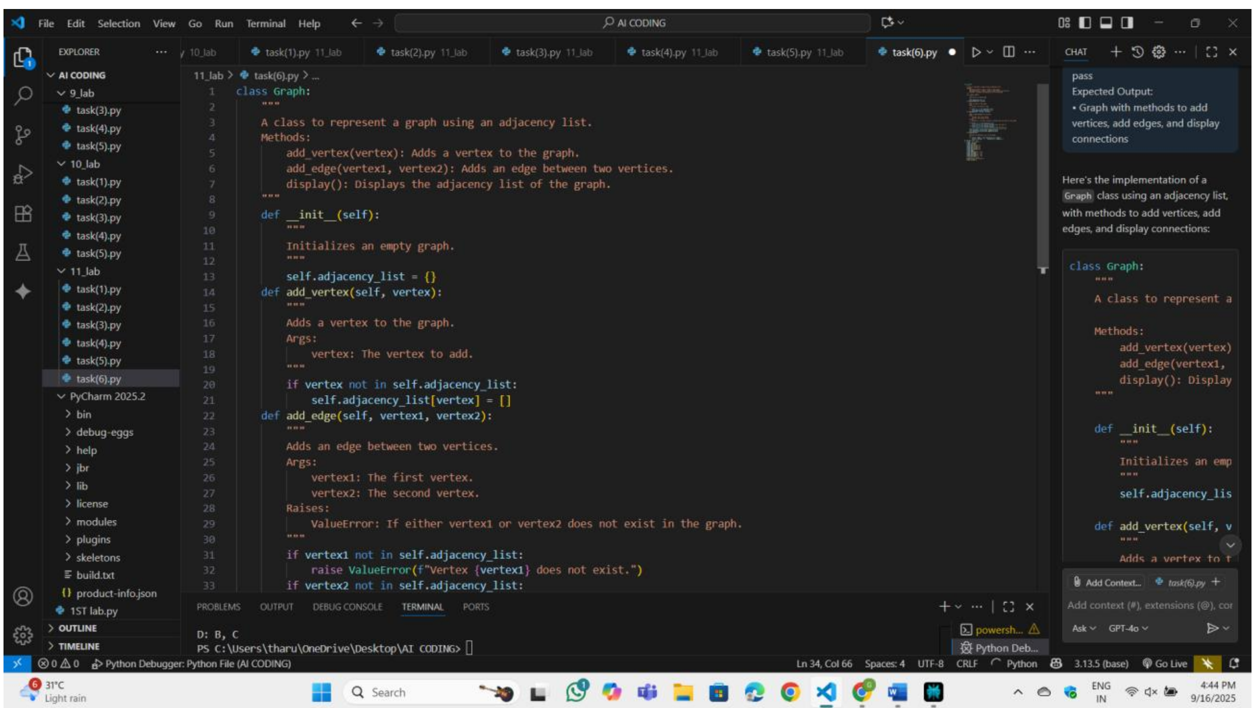
Observation:

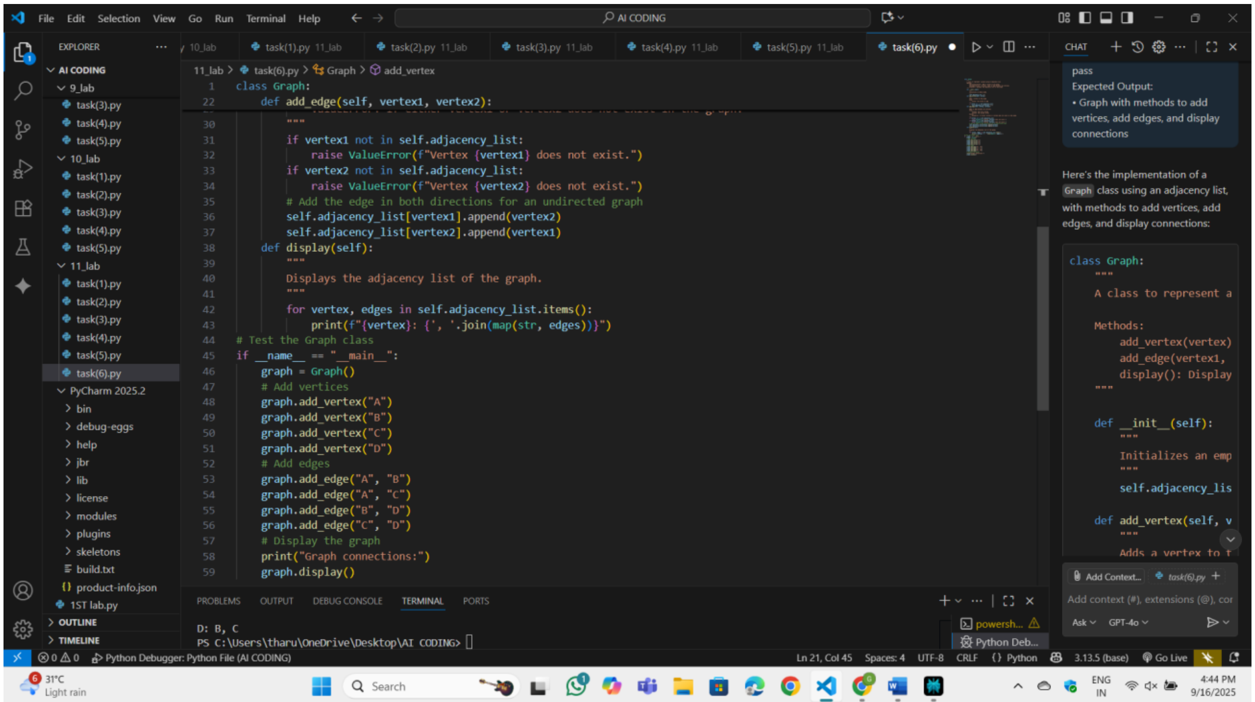
This program is about storing data in a special table where each piece of information is placed in a specific spot calculated from its key. Sometimes, more than one key can end up in the same spot, and in that case, they are simply kept together in a small list at that position. When adding something new, if the key already exists, its value gets updated if not, the new pair is added. To get information, it searches the correct spot and returns the value if the key is found, otherwise nothing. You can also remove a key from the table, and there's a way to display everything stored inside.

Task 6:

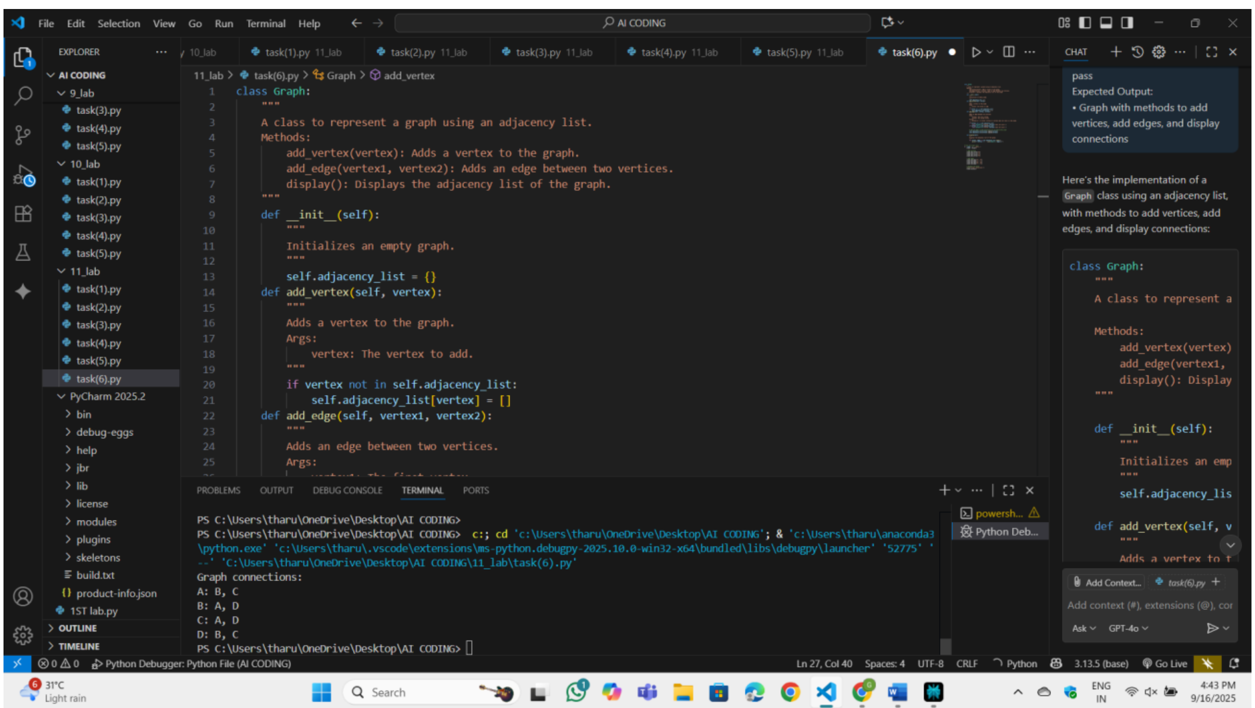
Prompt: implement a graph using an adjacency list.

Code:





OP:



Observation:

This program treats a network like a map where each point keeps a small list of its neighboring points it directly connects to. New points can be added by creating an empty spot for their connections, and links between two points are recorded on both sides so each knows about the other. If a link is requested between points that don’t exist, it’s considered a mistake and the process is stopped with an error message. There’s also a simple way to go through every point and show which other points it’s connected to, making the whole map easy to read.