ASSIGNMENT – 11

NAME: G.OMKAR

HT.NO: 2403A52039

BATCH: 03

Task-1

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

class Stack:

pass

Expected Output:

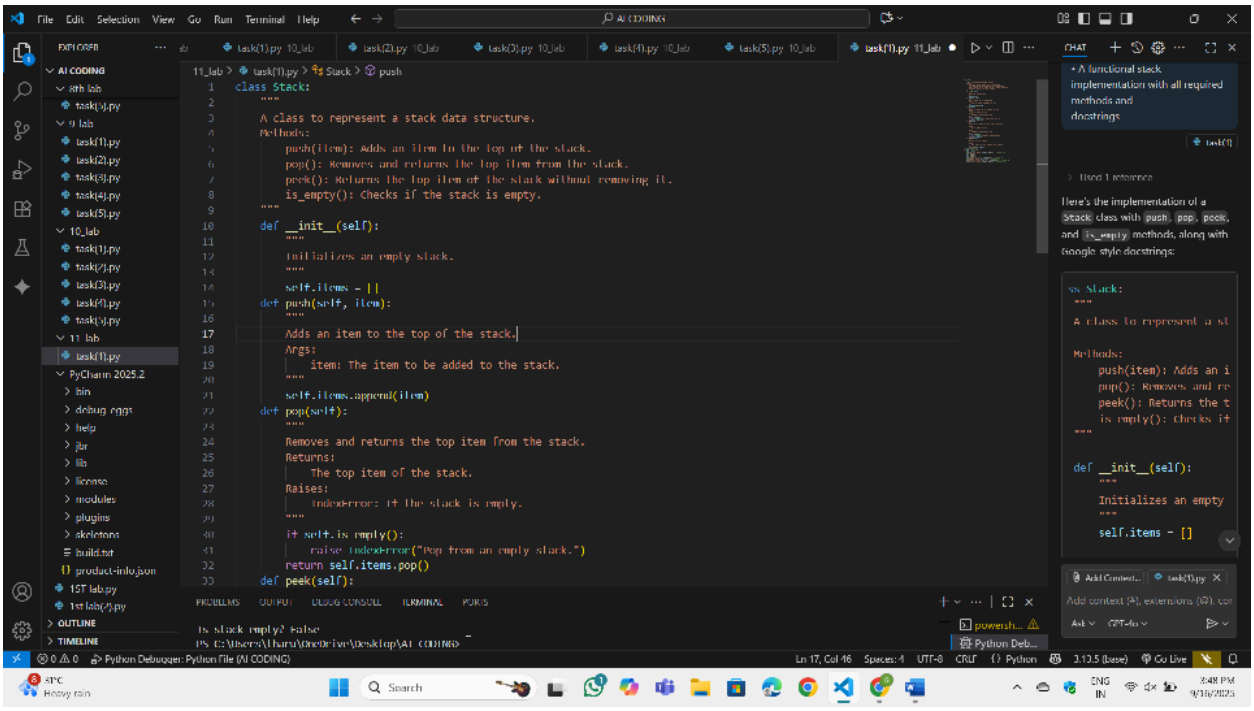
• A functional stack implementation with all required methods and

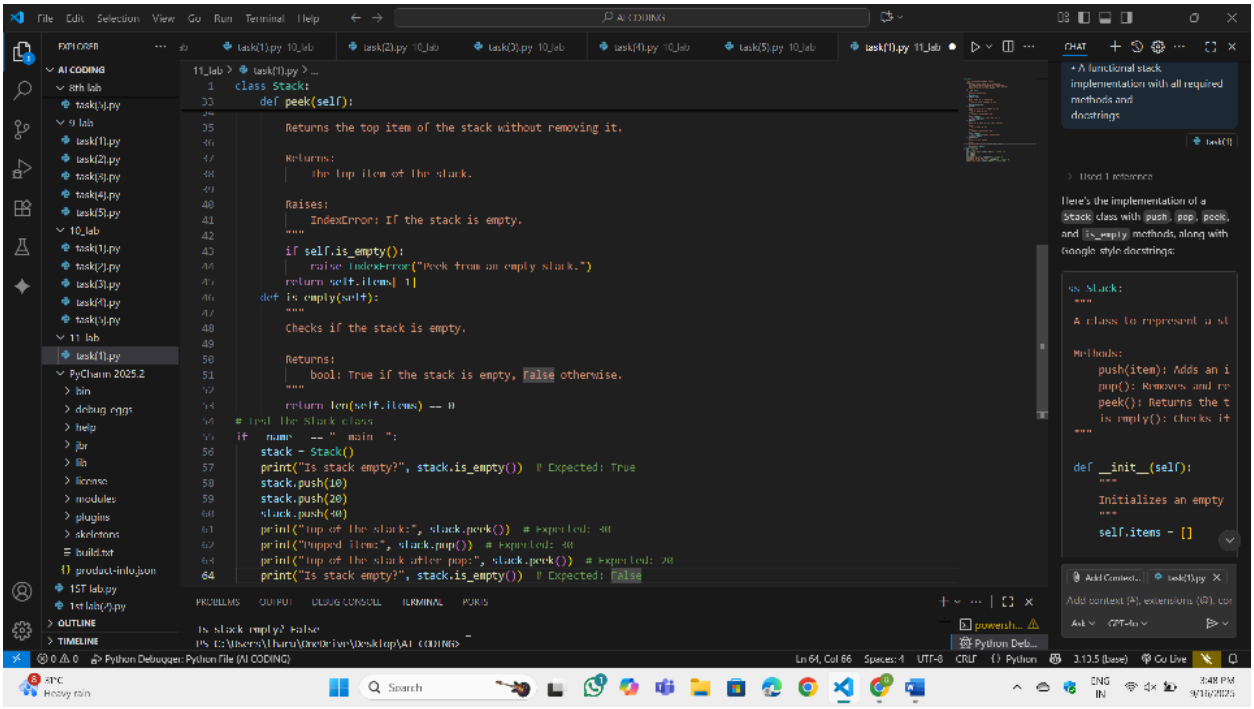
docstrings

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

methods.

Code:


class Stack:
"""
A class to represent a stack data structure.
Methods:
push(item): Adds an item to the top of the stack.
pop(): Removes and returns the top item from the stack.
peek(): Returns the top item of the stack without removing it.
is_empty(): Checks if the stack is empty.
"""
def __init__(self):
"""
Initializes an empty stack.
"""
self.items = []
def push(self, item):
"""
Adds an item to the top of the stack.
Args:
item: The item to be added to the stack.
"""
self.items.append(item)
def pop(self):
"""
Removes and returns the top item from the stack.
Returns:
The top item of the stack.
Raises:
IndexError: If the stack is empty.
"""
if self.is_empty():
raise IndexError("Pop from an empty stack.")
return self.items.pop()
def peek(self):
Is stack empty? False
PS C:\Users\Iharu\OneDrive\Desktop\AI CODING>


class Stack:
def peek(self):
Returns the top item of the stack without removing it.
Returns:
The top item of the stack.
Raises:
IndexError: If the stack is empty.
"""
if self.is_empty():
raise IndexError("Peek from an empty stack.")
return self.items[-1]
def is_empty(self):
"""
Checks if the stack is empty.
Returns:
bool: True if the stack is empty, False otherwise.
"""
return len(self.items) == 0
# Test the Stack class
if __name__ == "__main__":
stack = Stack()
print("Is stack empty?", stack.is_empty()) # Expected: True
stack.push(10)
stack.push(20)
stack.push(30)
print("Top of the stack:", stack.peek()) # Expected: 30
print("Popped item:", stack.pop()) # Expected: 30
print("Top of the stack after pop:", stack.peek()) # Expected: 20
print("Is stack empty?", stack.is_empty()) # Expected: False
Is stack empty? False
PS C:\Users\Iharu\OneDrive\Desktop\AI CODING>

OP:

Observation:

This program works like a collection where things are arranged one on top of another. 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 latest one added comes out first. It also allows me to just see what is on top without removing it. There is even a way to check if the collection has nothing inside.

Task-2

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

pass

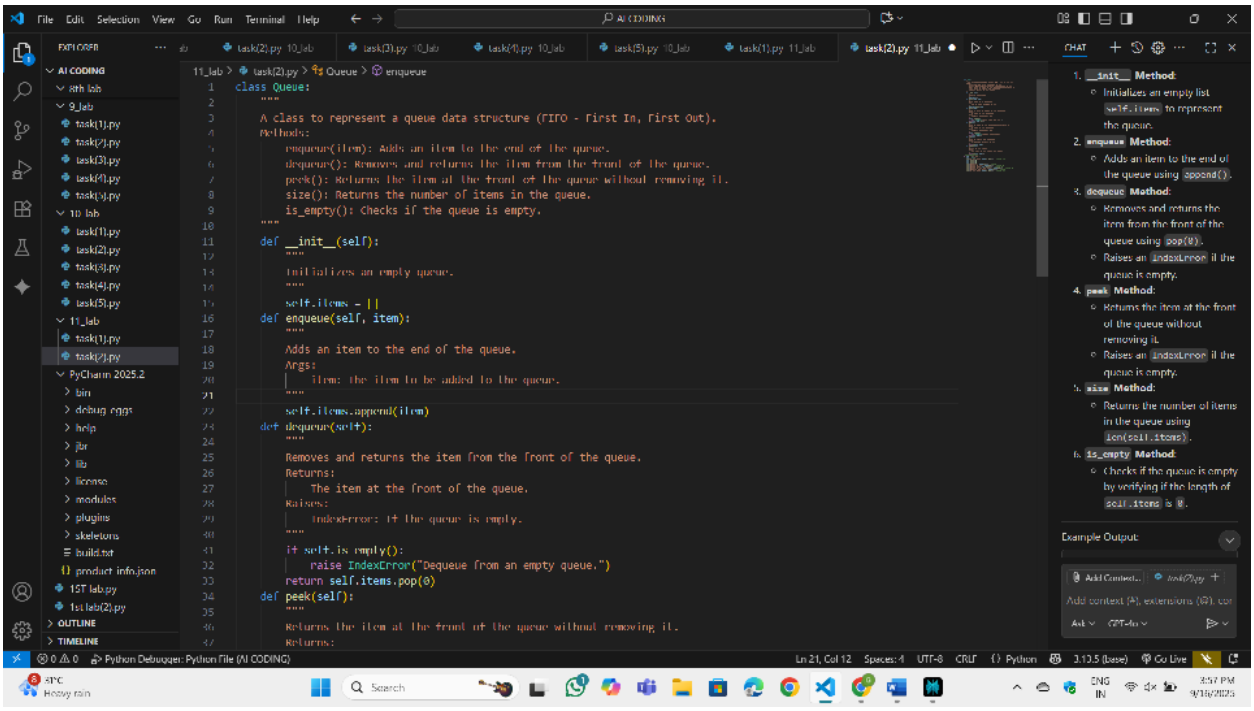
Expected Output:

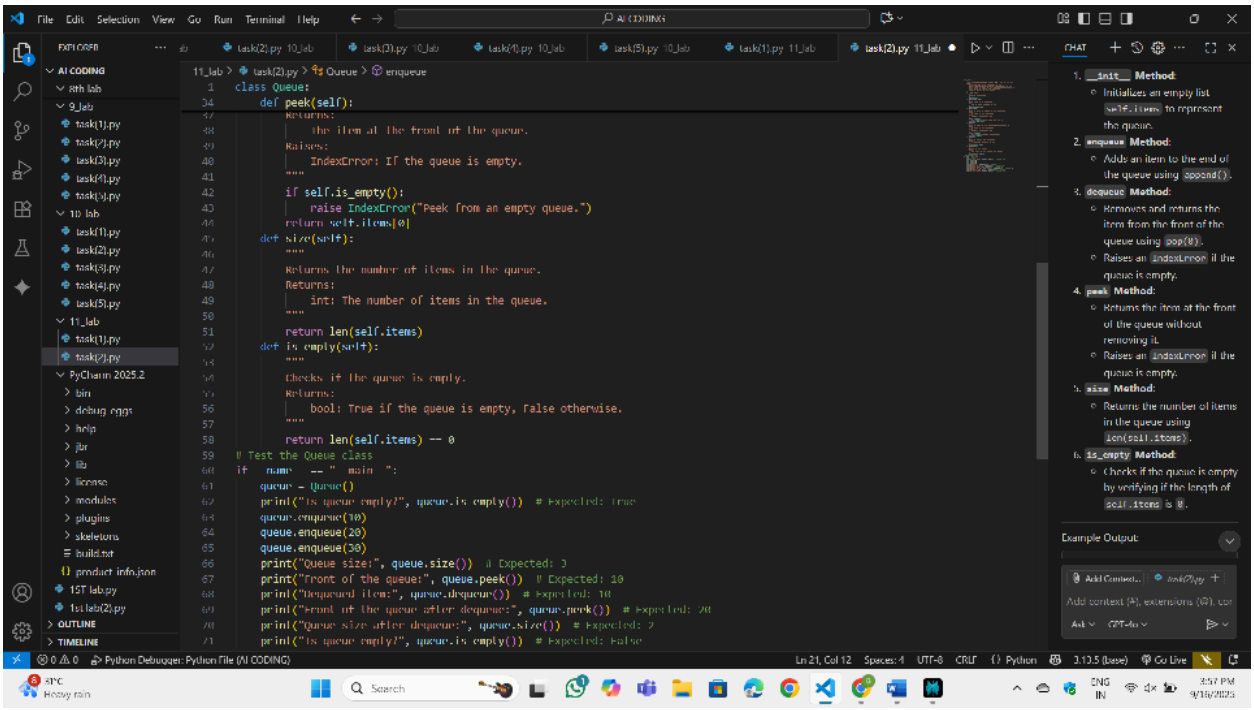
• FIFO-based queue class with enqueue, dequeue, peek, and size

methods

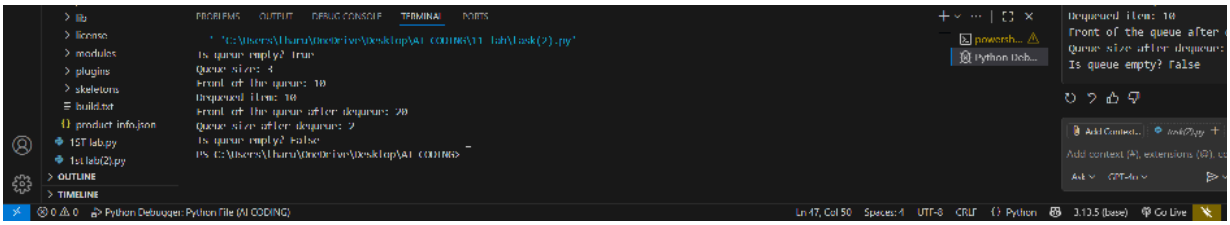
Prompt: implement a Queue using Python lists.

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:


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:

This program works like a line where people stand one after another. In 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:

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

class Node:

pass

class LinkedList:

pass

Expected Output:

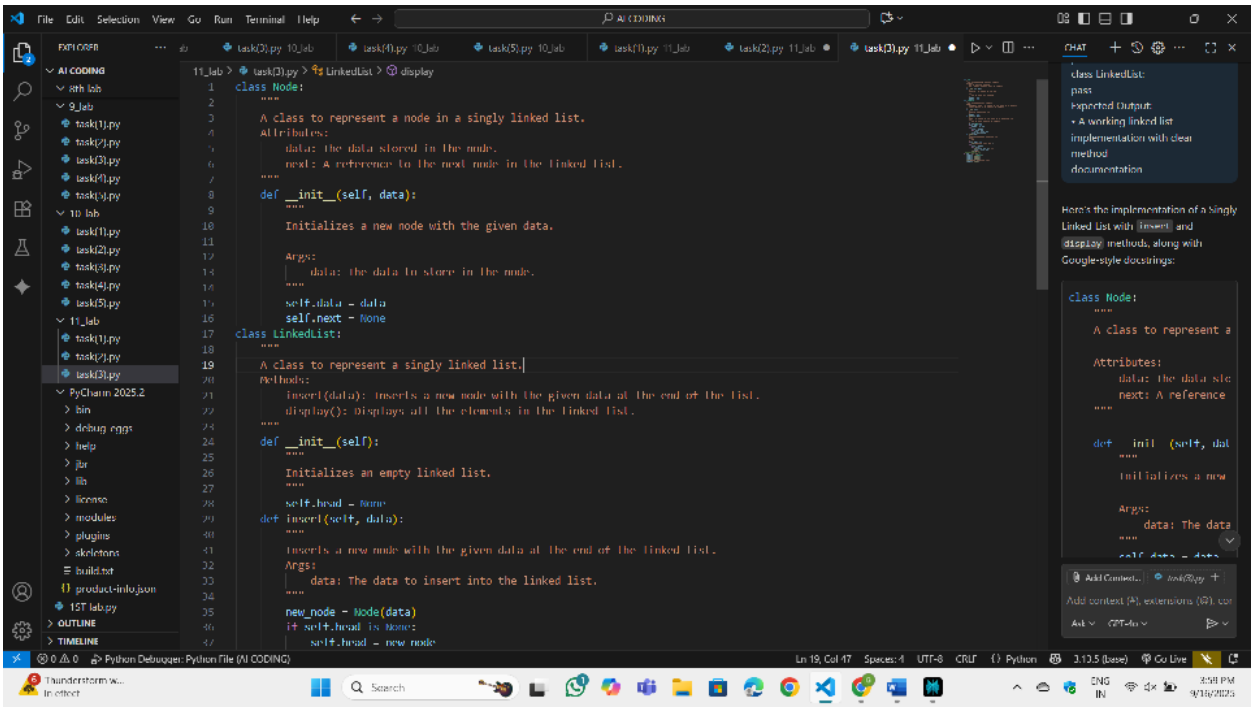
• A working linked list implementation with clear method

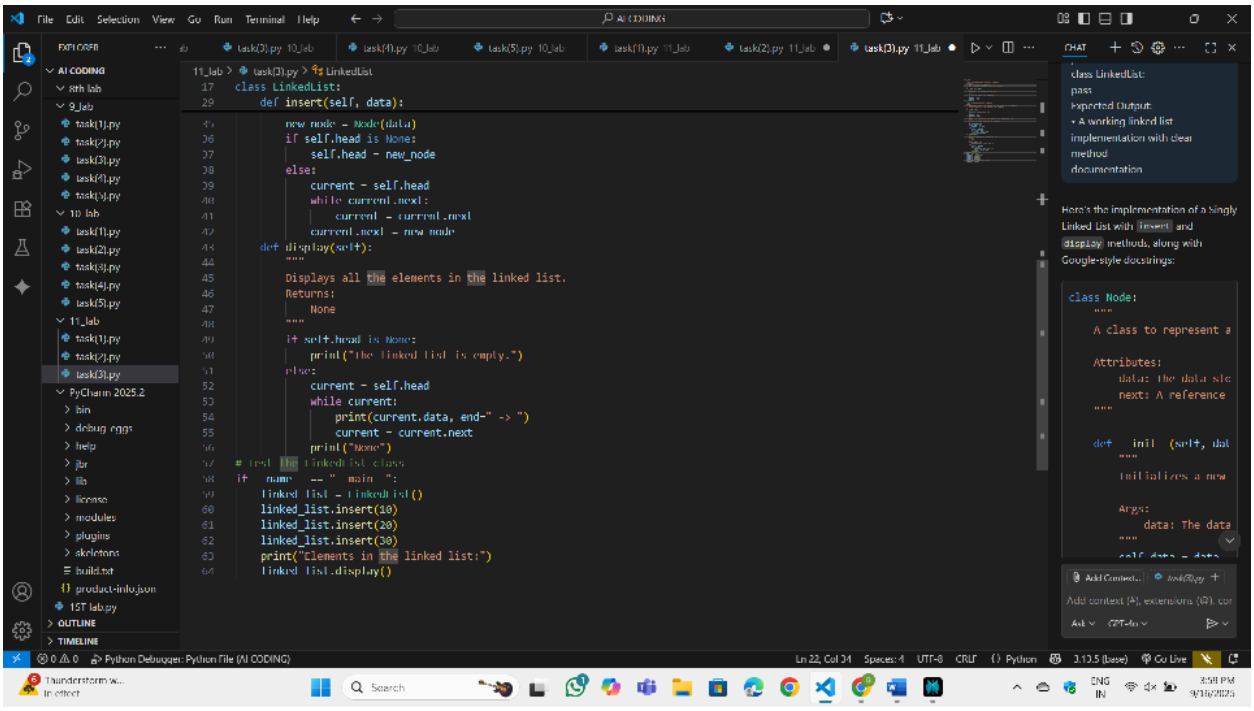
documentation

Prompt: generate a Singly Linked List with insert and display methods. A working linked list implementation with clear method

documentation

Code:





OP:

Observation:

This program works like a line where people stand one after another. In 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:

Task: Use AI to create a BST with insert and in-order traversal

methods.

Sample Input Code:

class BST:

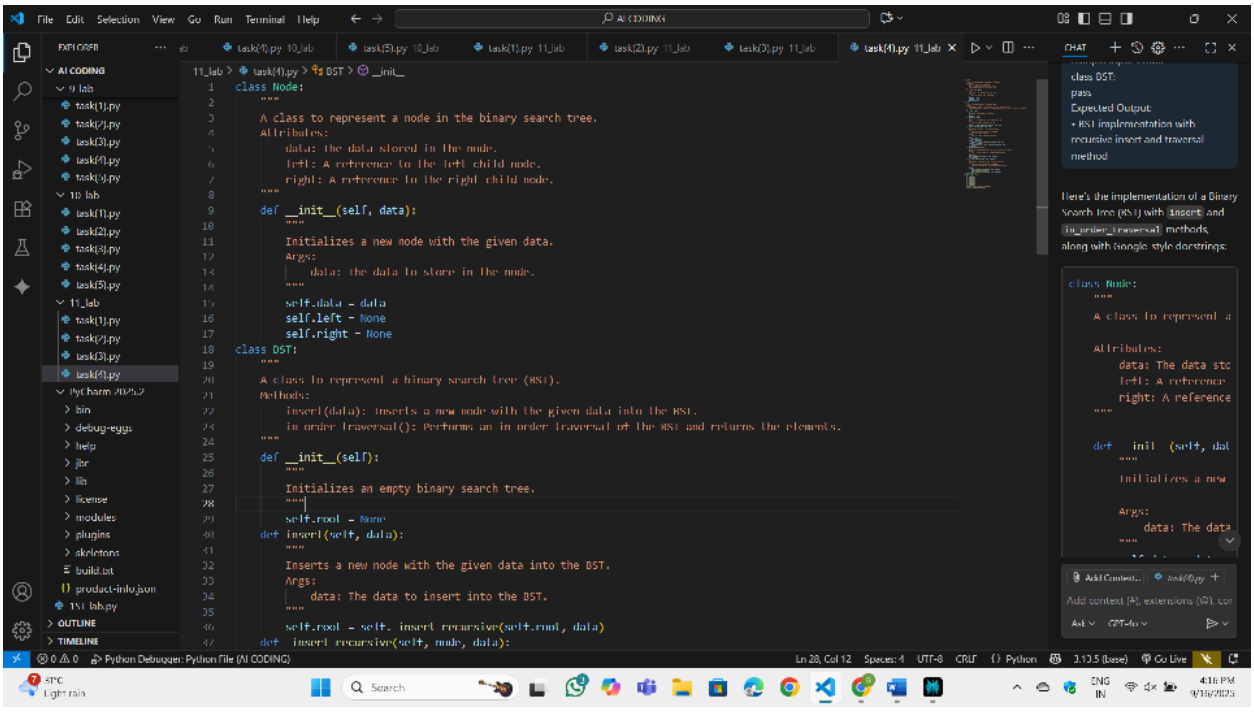
pass

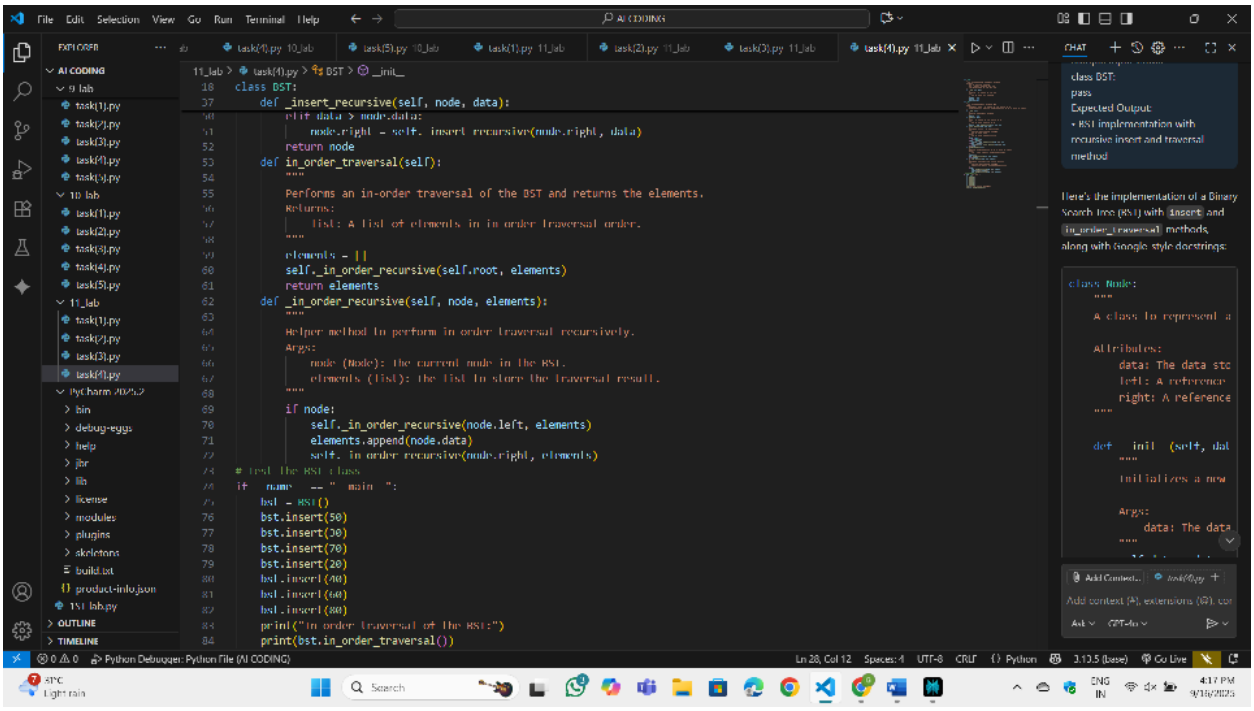
Expected Output:

• BST implementation with recursive insert and traversal method

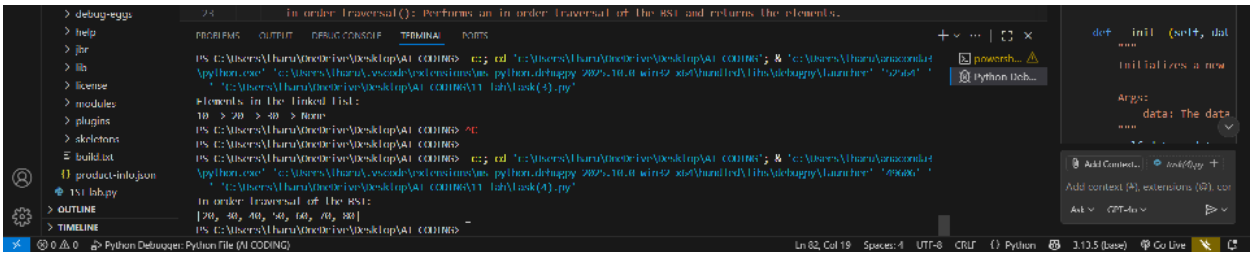
Prompt: create a BST with insert and in-order traversal methods.

Code:





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:

Task: Use AI to implement a hash table with basic insert, search, and delete

methods.

Sample Input Code:

class HashTable:

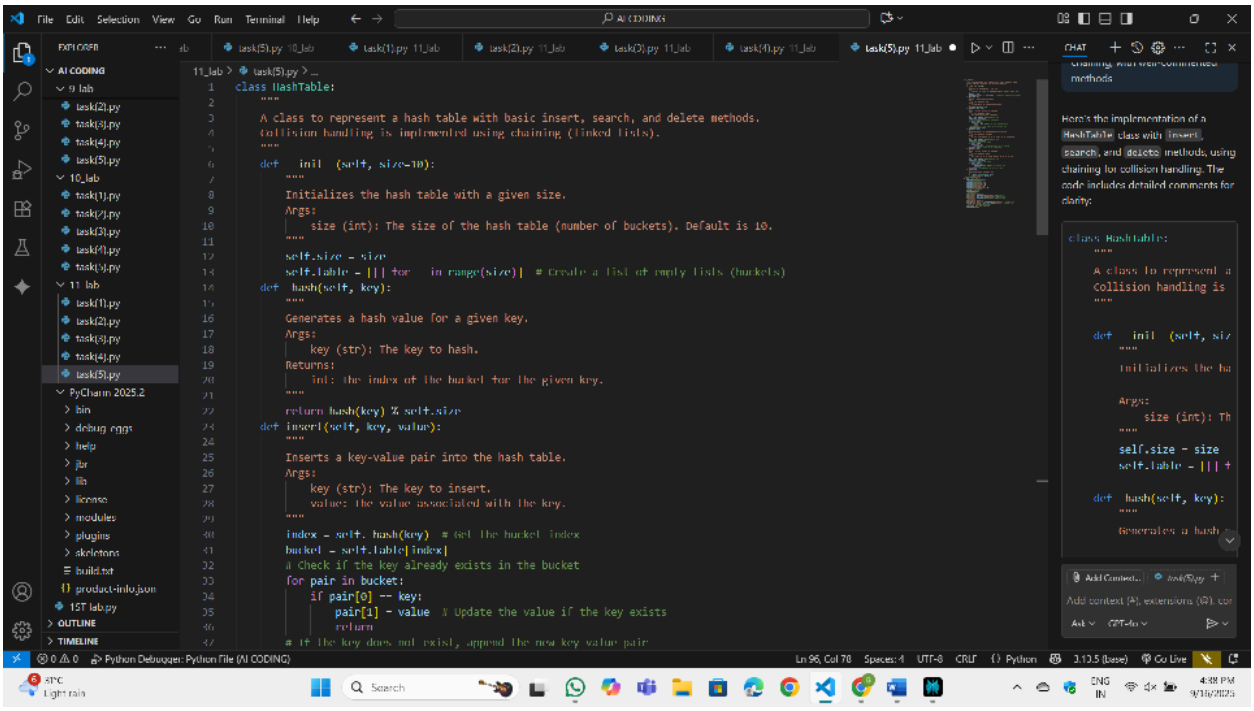
pass

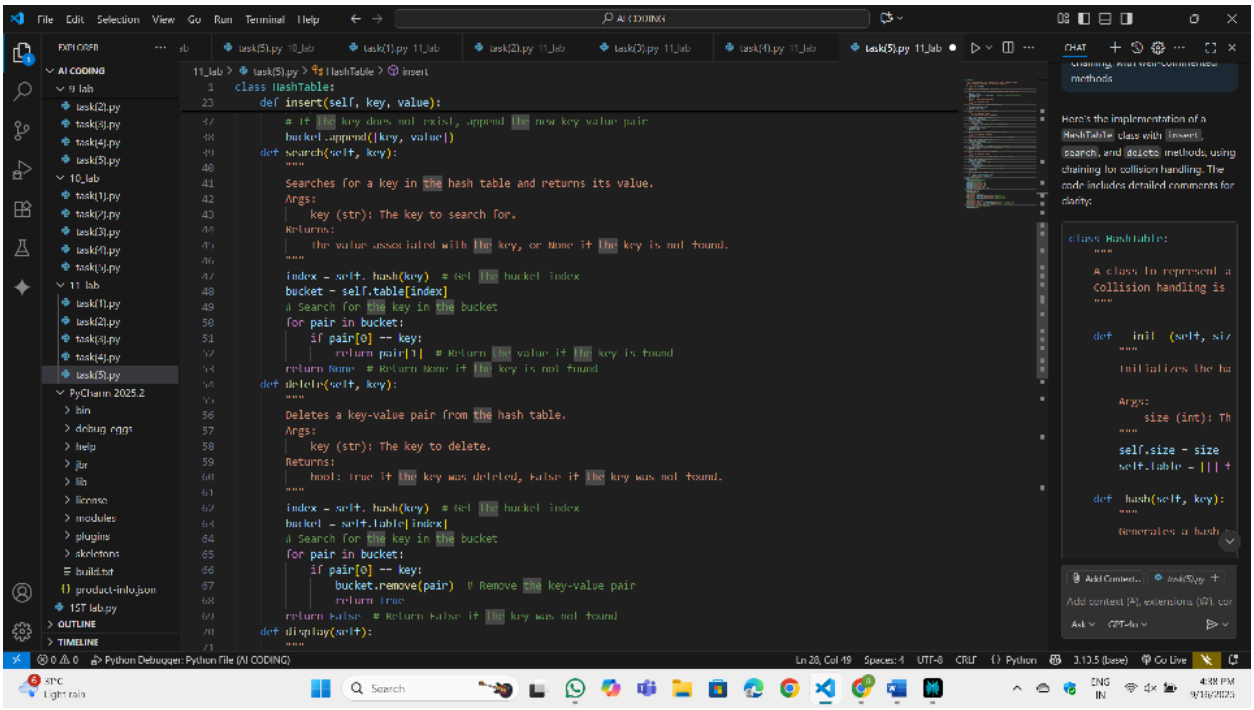
Expected Output:

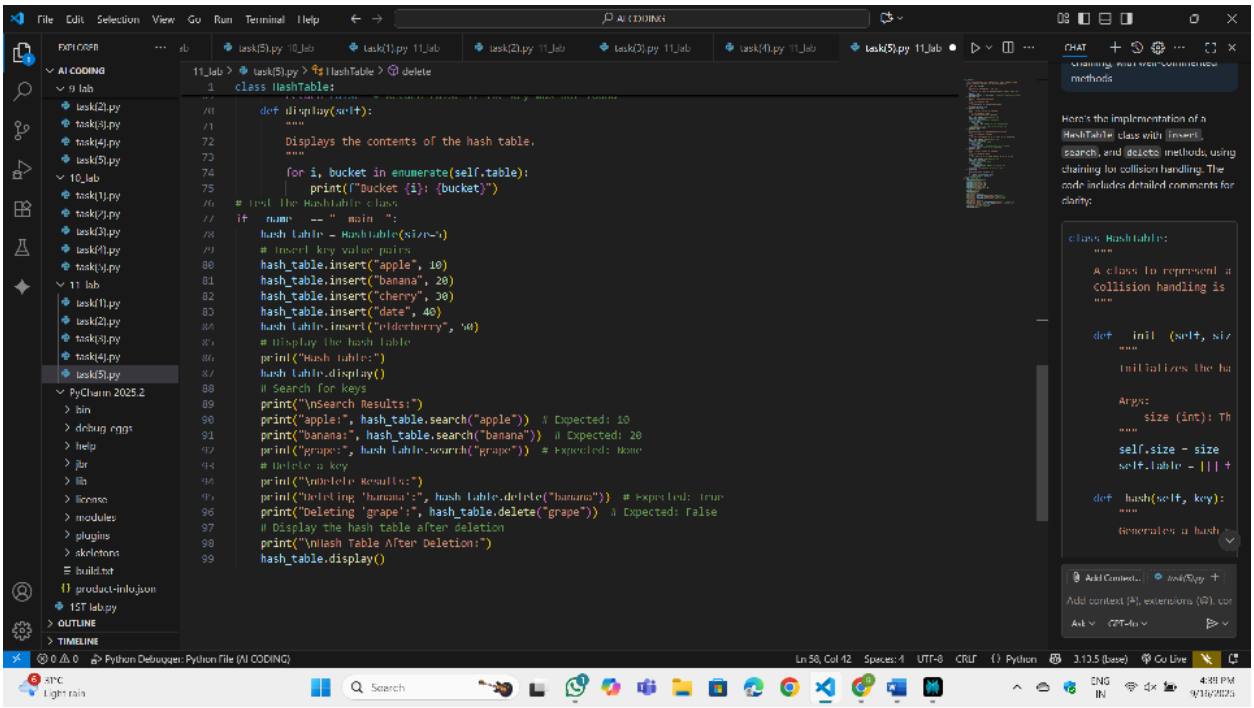
• Collision handling using chaining, with well-commented methods

Prompt: implement a hash table with basic insert, search, and delete. Collision handling using chaining, with well-commented methods

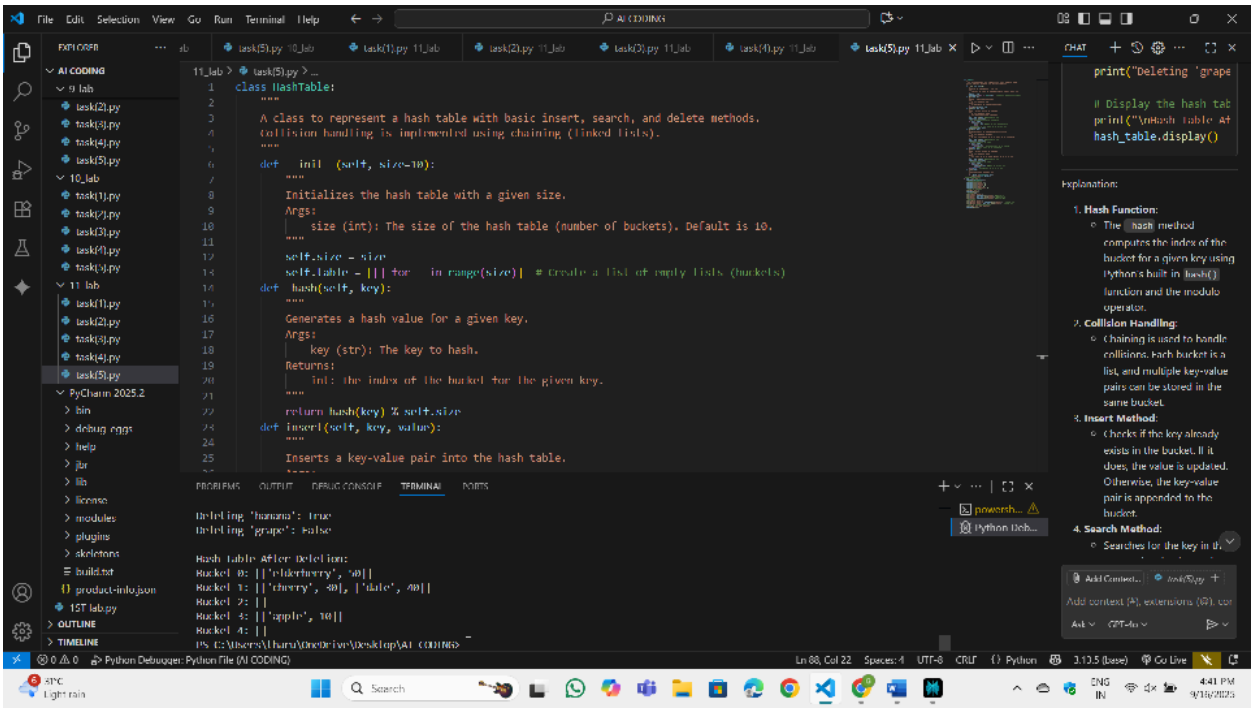
Code:


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


class HashTable:
def insert(self, key, value):
# If the key does not exist, append the new key-value pair
self.table[index].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):
"""



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 look up information, it searches the correct spot and returns the value if the key is found, or nothing if it isn't. You can also remove a key from the table, and there's a way to display everything stored inside.

Task 6:

Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

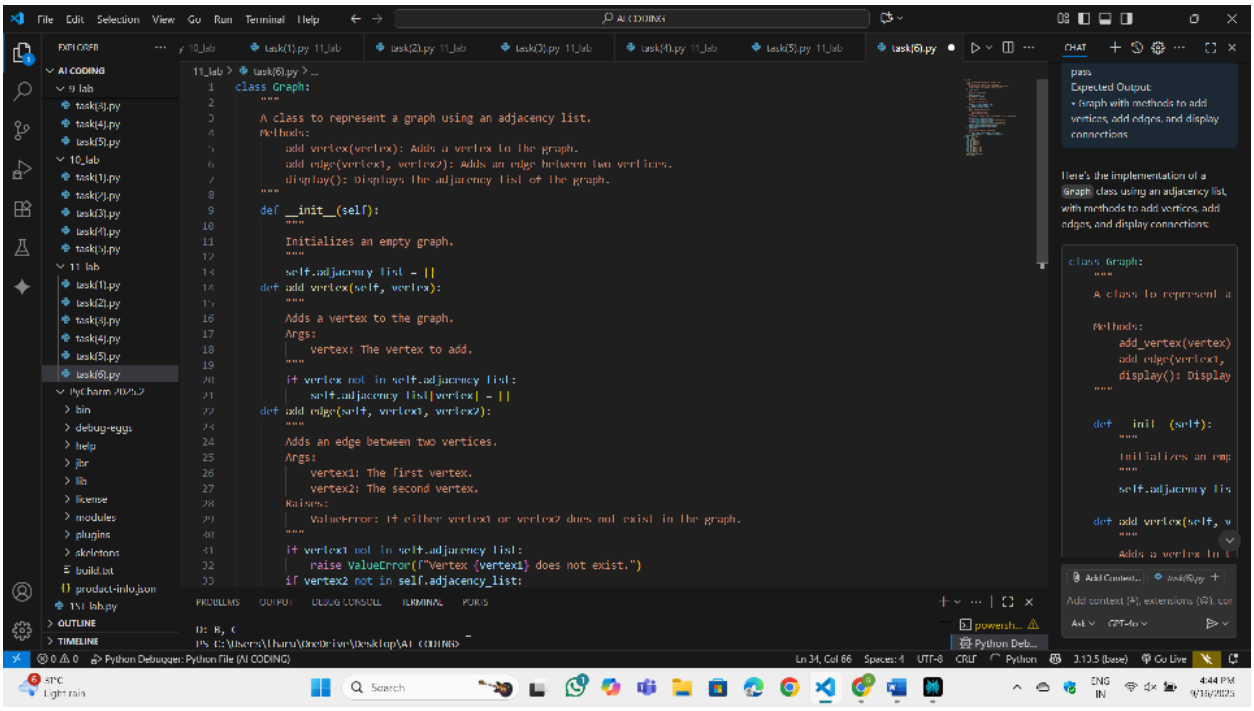
Expected Output:

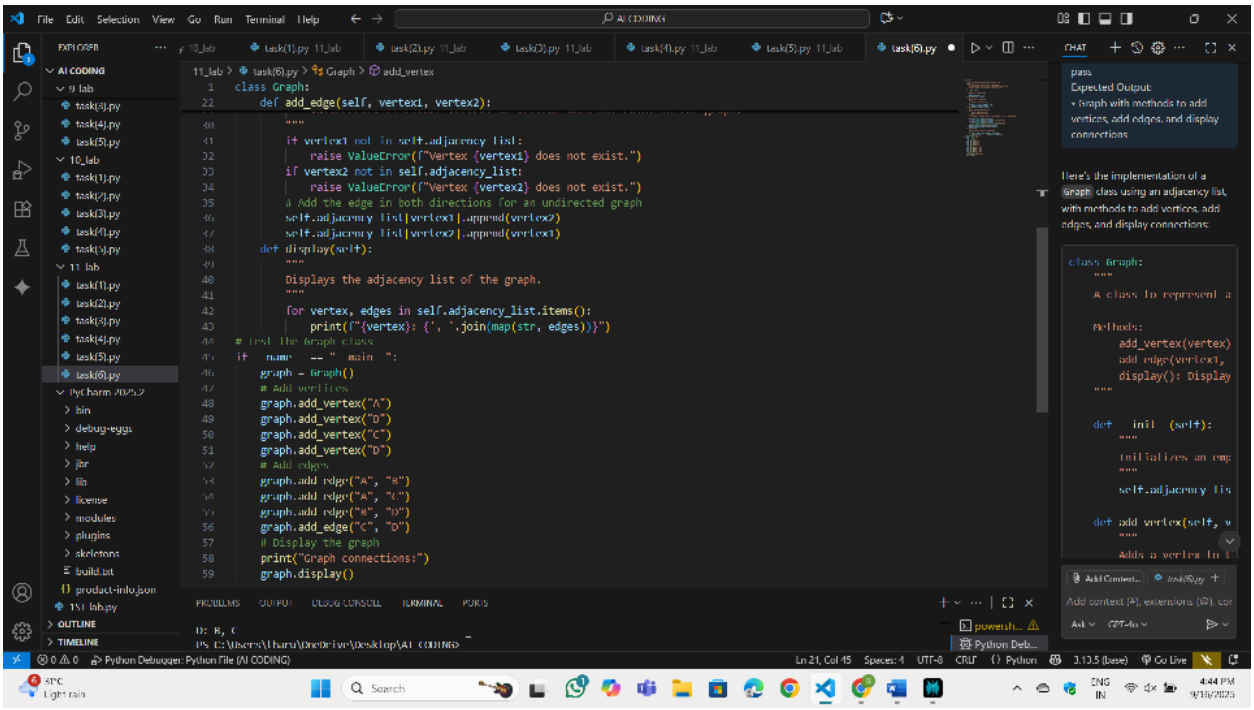
• Graph with methods to add vertices, add edges, and display

connections

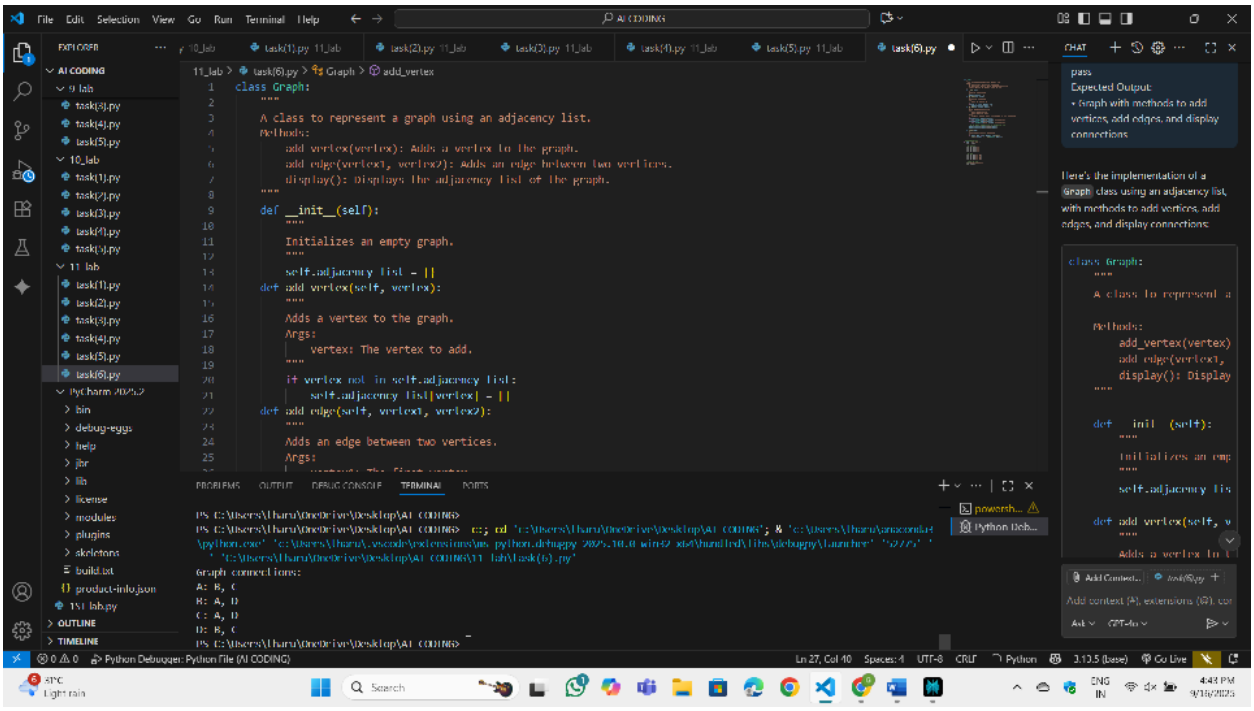
Prompt: implement a graph using an adjacency list.

Code:


class Graph:
"""
A class to represent a graph using an adjacency list.
Methods:
add_vertex(vertex): Adds a vertex to the graph.
add_edge(vertex1, vertex2): Adds an edge between two vertices.
display(): Displays the adjacency list of the graph.
"""
def __init__(self):
"""
Initializes an empty graph.
"""
self.adjacency_list = {}
def add_vertex(self, vertex):
"""
Adds a vertex to the graph.
Args:
vertex: The vertex to add.
"""
if vertex not in self.adjacency_list:
self.adjacency_list[vertex] = []
def add_edge(self, vertex1, vertex2):
"""
Adds an edge between two vertices.
Args:
vertex1: The first vertex.
vertex2: The second vertex.
Raises:
ValueError: If either vertex1 or vertex2 does not exist in the graph.
"""
if vertex1 not in self.adjacency_list:
raise ValueError(f"Vertex {vertex1} does not exist.")
if vertex2 not in self.adjacency_list:


class Graph:
def add_edge(self, vertex1, vertex2):
"""
if vertex1 not in self.adjacency_list:
raise ValueError(f"Vertex {vertex1} does not exist.")
if vertex2 not in self.adjacency_list:
raise ValueError(f"Vertex {vertex2} does not exist.")
Add the edge in both directions for an undirected graph
self.adjacency_list[vertex1].append(vertex2)
self.adjacency_list[vertex2].append(vertex1)
def display(self):
"""
Displays the adjacency list of the graph.
"""
for vertex, edges in self.adjacency_list.items():
print(f"{vertex}: {', '.join(map(str, edges))}")
# Test the Graph class
if __name__ == "__main__":
graph = Graph()
# Add vertices
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_vertex("D")
# Add edges
graph.add_edge("A", "B")
graph.add_edge("A", "C")
graph.add_edge("B", "D")
graph.add_edge("C", "D")
# Display the graph
print("Graph connections:")
graph.display()

OP:



Observation:

This setup 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.

Task 7:

ask: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

class PriorityQueue:

pass

Expected Output:

• Implementation with enqueue (priority), dequeue (highest priority), and display methods

Prompt:

Code:

OP:

Observation:

Task 8:

Use AI to implement a double-ended queue using collections.deque.

Sample Input Code:

class DequeDS:

pass

Expected Output:

• Insert and remove from both ends with docstrings.

Prompt:

Code:

OP:

Observation:

Task 9:

Task: Use AI to generate a comparison table of different data structures (stack,

queue, linked list, etc.) including time complexities.

Sample Input Code:

No code, prompt AI for a data structure comparison table

Expected Output:

• A markdown table with structure names, operations, and complexities

Prompt:

Code:

OP:

Observation:

Task 10

Your college wants to develop a Campus Resource Management System that

handles:

1. Student Attendance Tracking – Daily log of students entering/

exiting

the campus.

2. Event Registration System – Manage participants in events with quick

search and removal.

3. Library Book Borrowing – Keep track of available books and their due

dates.

4. Bus Scheduling System – Maintain bus routes and stop connections.

5. Cafeteria Order Queue – Serve students in the order they arrive.

Student Task:

• For each feature, select the most appropriate data structure from the list

below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

• Justify your choice in 2–3 sentences per feature.

• Implement one selected feature as a working Python program with AI-

assisted code generation.

Expected Output:

• A table mapping feature chosen data structure justification.

• A functional Python program implementing the chosen feature with

comments and docstrings.

Prompt:

Code:

OP:

Observation: