AI ASSISTED CODING

ASSIGNMENT – 11

NAME: R.SURYANARAYANA

HT.NO: 2403A52038

BATCH: 03

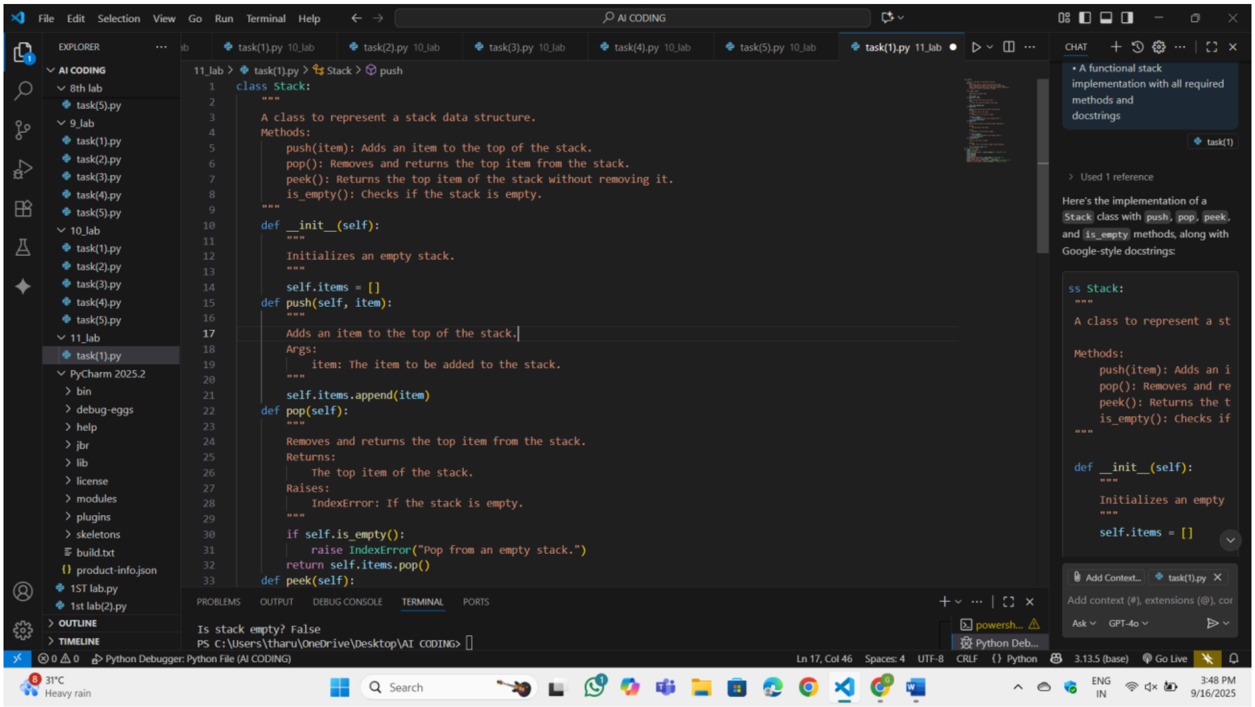
Task-1

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:

AI ASSISTED CODING



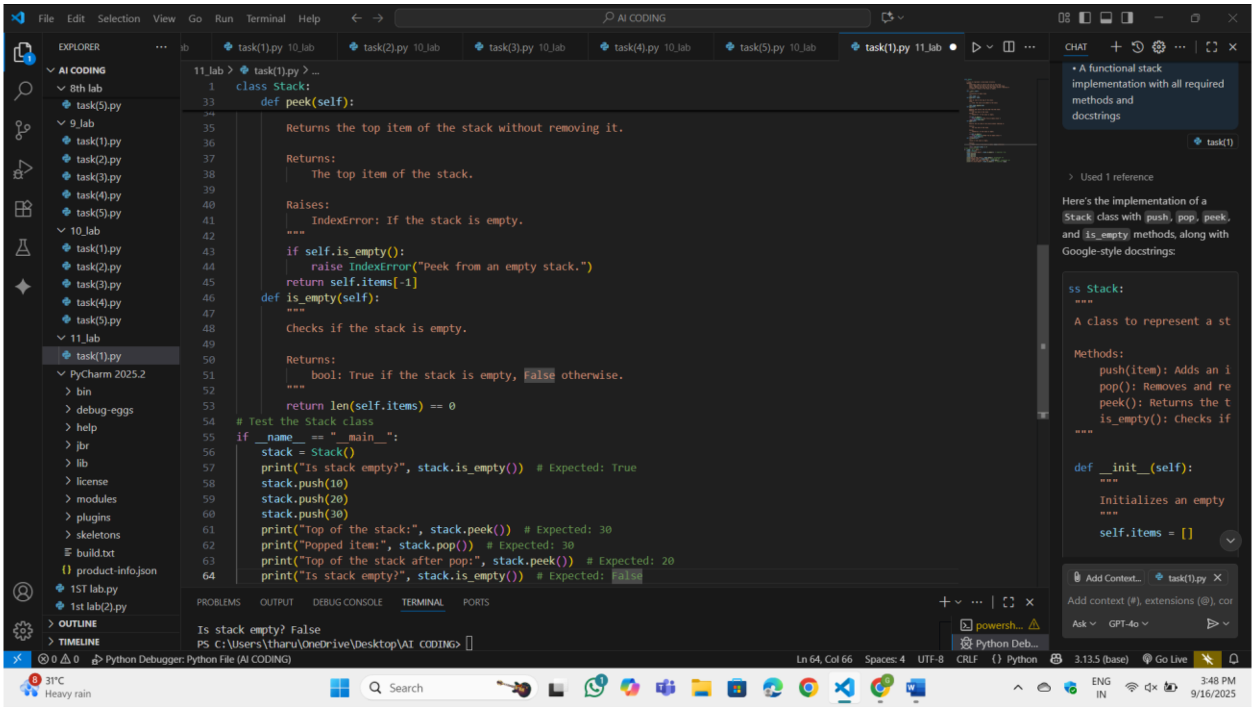
```python
class Stack:
    """
    A class to represent a stack data structure.
    Methods:
        push(item): Adds an item to the top of the stack.
        pop(): Removes and returns the top item from the stack.
        peek(): Returns the top item of the stack without removing it.
        is_empty(): 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\tharu\OneDrive\Desktop\AI CODING> 
```

- A functional stack implementation with all required methods and docstrings

Here's the implementation of a `Stack` class with `push`, `pop`, `peek`, and `is_empty` methods, along with Google-style docstrings:

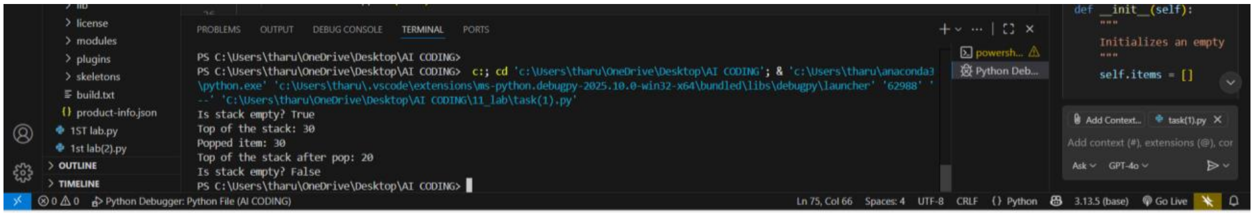
```python
ss Stack:
 """
 A class to represent a st

 Methods:
     push(item): Adds an i
     pop(): Removes and re
     peek(): Returns the t
     is_empty(): Checks if
 """

 def __init__(self):
     """
     Initializes an empty
     """
     self.items = []
```



```python
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\tharu\OneDrive\Desktop\AI CODING> 
```

OP:



```
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>  c:; cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Users\tharu\anaconda3
\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '62988' '
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(1).py'
Is stack empty? True
Top of the stack: 30
Popped item: 30
Top of the stack after pop: 20
Is stack empty? False
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
```

```
def __init__(self):
    """
    Initializes an empty
    """
    self.items = []
```

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

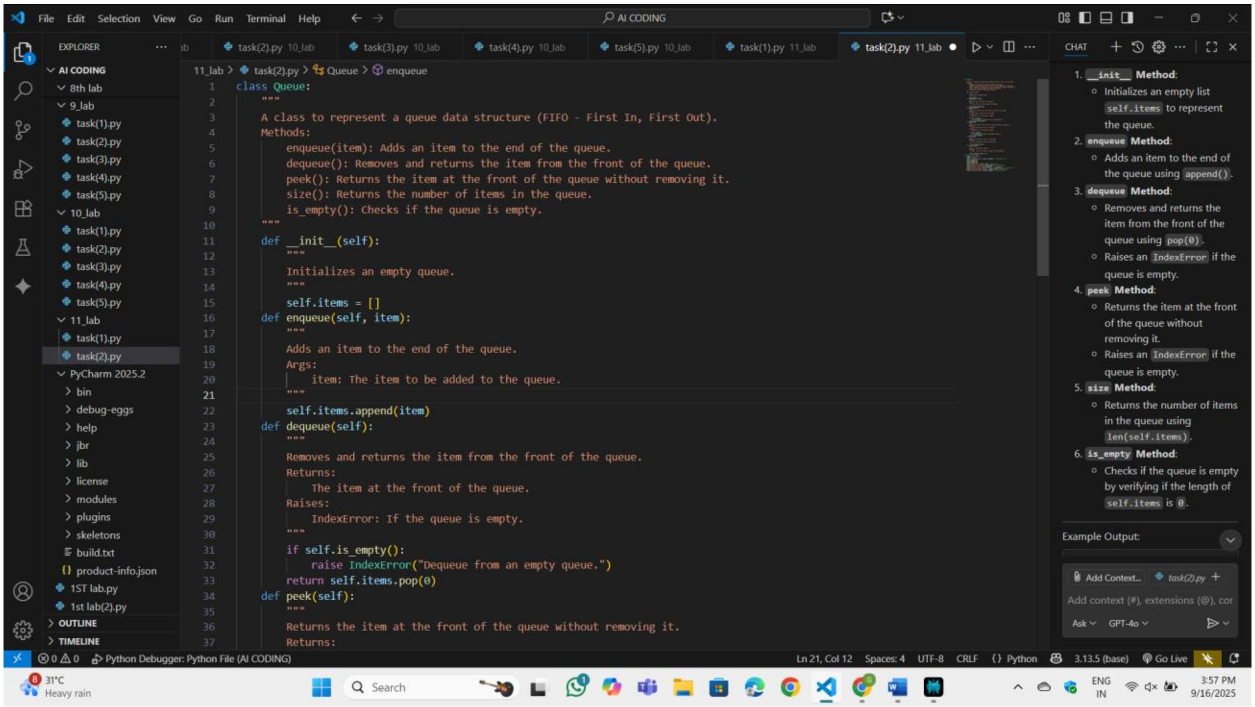
Task: Use AI to implement a Queue using Python lists.
Sample Input Code:
class Queue:
pass
Expected Output:

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

Prompt: implement a Queue using Python lists.

Code:

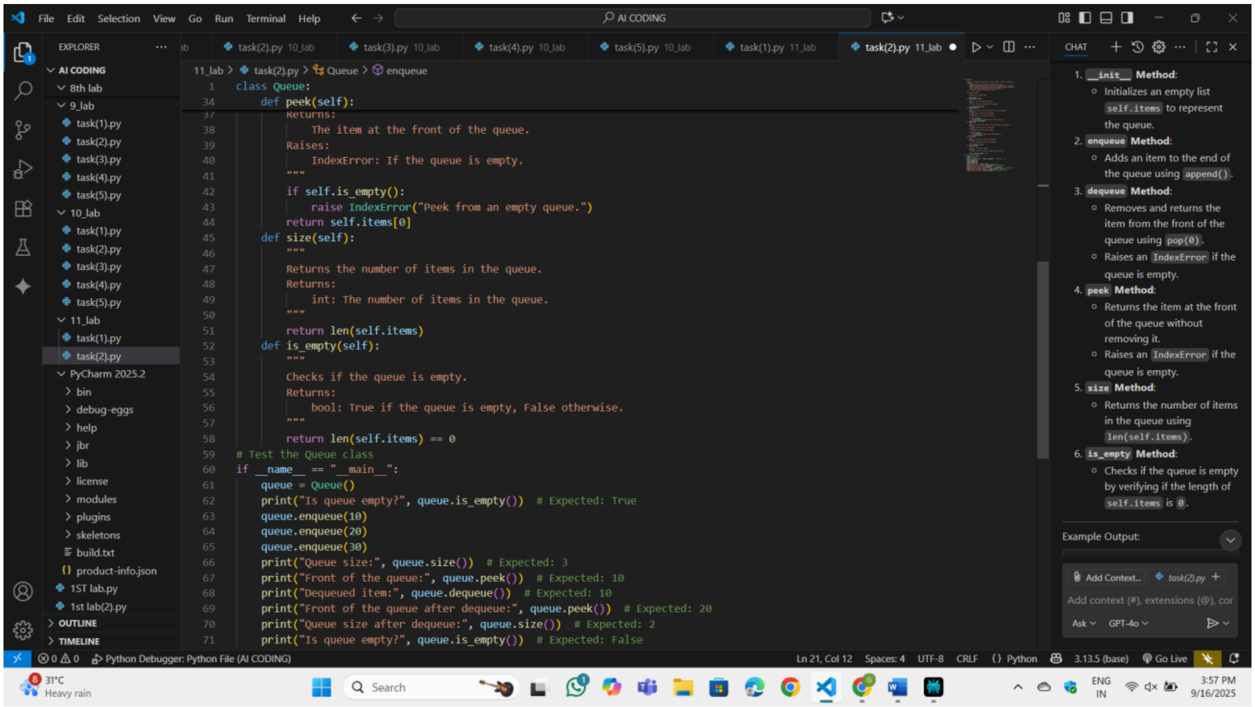
AI ASSISTED CODING



```
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:
```

1. `__init__` **Method**:
   - Initializes an empty list `self.items` to represent the queue.
2. `enqueue` **Method**:
   - Adds an item to the end of the queue using `append()`.
3. `dequeue` **Method**:
   - Removes and returns the item from the front of the queue using `pop(0)`.
   - Raises an `IndexError` if the queue is empty.
4. `peek` **Method**:
   - Returns the item at the front of the queue without removing it.
   - Raises an `IndexError` if the queue is empty.
5. `size` **Method**:
   - Returns the number of items in the queue using `len(self.items)`.
6. `is_empty` **Method**:
   - Checks if the queue is empty by verifying if the length of `self.items` is `0`.

Example Output:

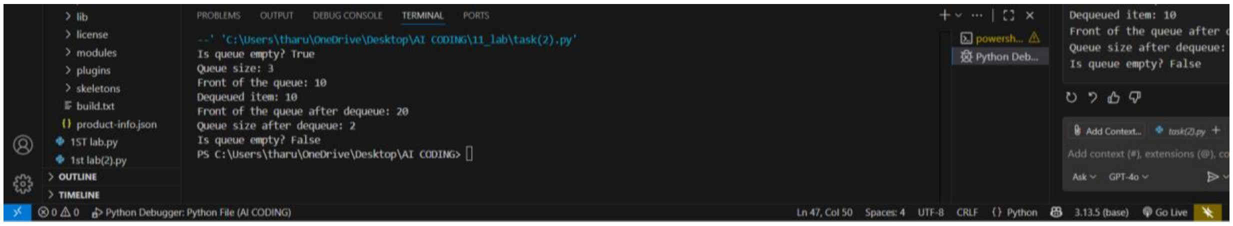


```
class Queue:
    def peek(self):
        Returns:
            The item at the front of the queue.
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty queue.")
        return self.items[0]
    def size(self):
        """
        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
```

1. `__init__` **Method**:
   - Initializes an empty list `self.items` to represent the queue.
2. `enqueue` **Method**:
   - Adds an item to the end of the queue using `append()`.
3. `dequeue` **Method**:
   - Removes and returns the item from the front of the queue using `pop(0)`.
   - Raises an `IndexError` if the queue is empty.
4. `peek` **Method**:
   - Returns the item at the front of the queue without removing it.
   - Raises an `IndexError` if the queue is empty.
5. `size` **Method**:
   - Returns the number of items in the queue using `len(self.items)`.
6. `is_empty` **Method**:
   - Checks if the queue is empty by verifying if the length of `self.items` is `0`.

Example Output:

OP:



```
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(2).py'
Is queue empty? True
Queue size: 3
Front of the queue: 10
Dequeued item: 10
Front of the queue after dequeue: 20
Queue size after dequeue: 2
Is queue empty? False
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

```
Dequeued item: 10
Front of the queue after 
Queue size after dequeue:
Is queue empty? False
```

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.

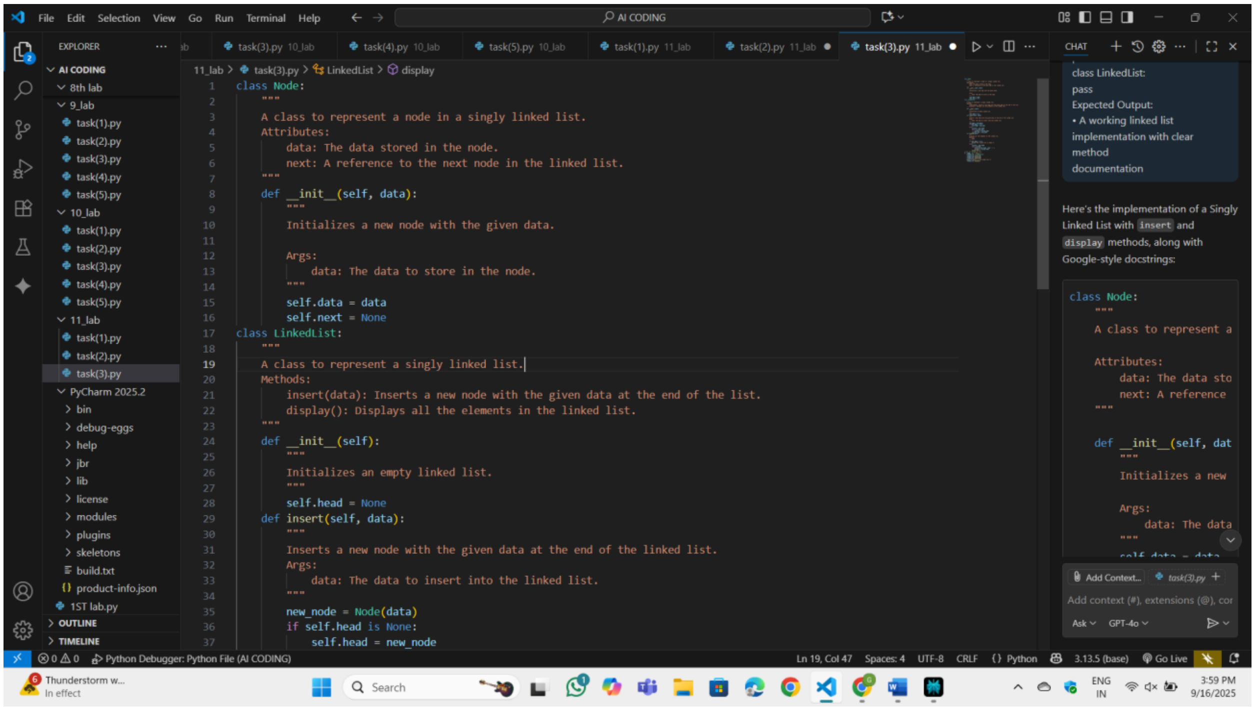
Task-3:

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

AI ASSISTED CODING

Code:

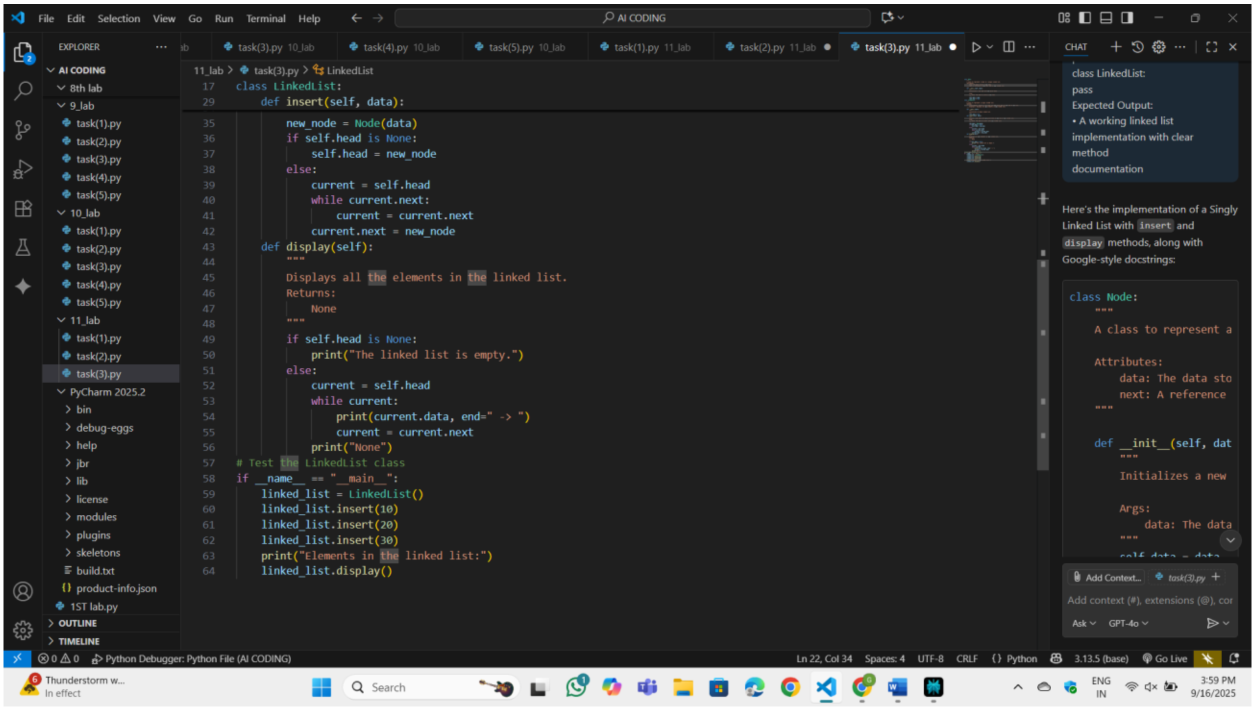


```python
class Node:
    """
    A class to represent a node in a singly linked list.
    Attributes:
        data: The data stored in the node.
        next: A reference to the next node in the linked list.
    """
    def __init__(self, data):
        """
        Initializes a new node with the given data.

        Args:
            data: The data to store in the node.
        """
        self.data = data
        self.next = None
class LinkedList:
    """
    A class to represent a singly linked list.
    Methods:
        insert(data): Inserts a new node with the given data at the end of the list.
        display(): Displays all the elements in the linked list.
    """
    def __init__(self):
        """
        Initializes an empty linked list.
        """
        self.head = None
    def insert(self, data):
        """
        Inserts a new node with the given data at the end of the linked list.
        Args:
            data: The data to insert into the linked list.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
```

class LinkedList:
pass
Expected Output:
- A working linked list implementation with clear method documentation

Here's the implementation of a Singly Linked List with `insert` and `display` methods, along with Google-style docstrings:

```python
class Node:
    """
    A class to represent a

    Attributes:
        data: The data sto
        next: A reference
    """

    def __init__(self, dat
        """
        Initializes a new

        Args:
            data: The data
        """
```

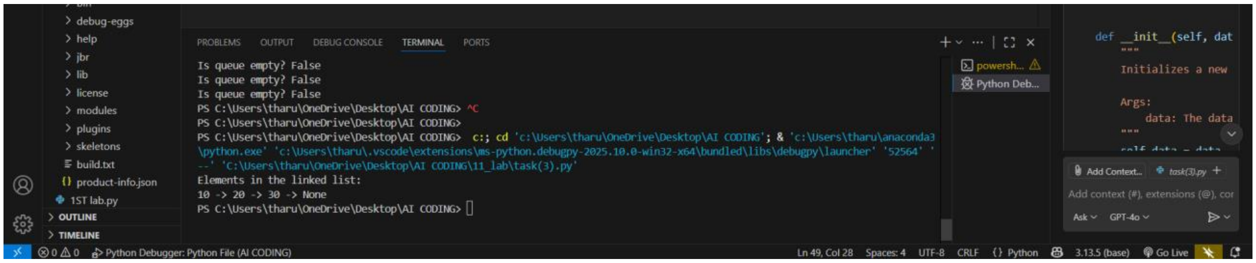


```python
class LinkedList:
    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        """
        Displays all the elements in the linked list.
        Returns:
            None
        """
        if self.head is None:
            print("The linked list is empty.")
        else:
            current = self.head
            while current:
                print(current.data, end=" -> ")
                current = current.next
            print("None")
# Test the LinkedList class
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert(10)
    linked_list.insert(20)
    linked_list.insert(30)
    print("Elements in the linked list:")
    linked_list.display()
```

class LinkedList:
pass
Expected Output:
- A working linked list implementation with clear method documentation

Here's the implementation of a Singly Linked List with `insert` and `display` methods, along with Google-style docstrings:

OP:



```
Is queue empty? False
Is queue empty? False
Is queue empty? False
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> ^C
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>  c:; cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Users\tharu\anaconda3
\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '52564' '
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(3).py'
Elements in the linked list:
10 -> 20 -> 30 -> None
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

```
def __init__(self, dat
    """
    Initializes a new

    Args:
        data: The data
    """
```

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.

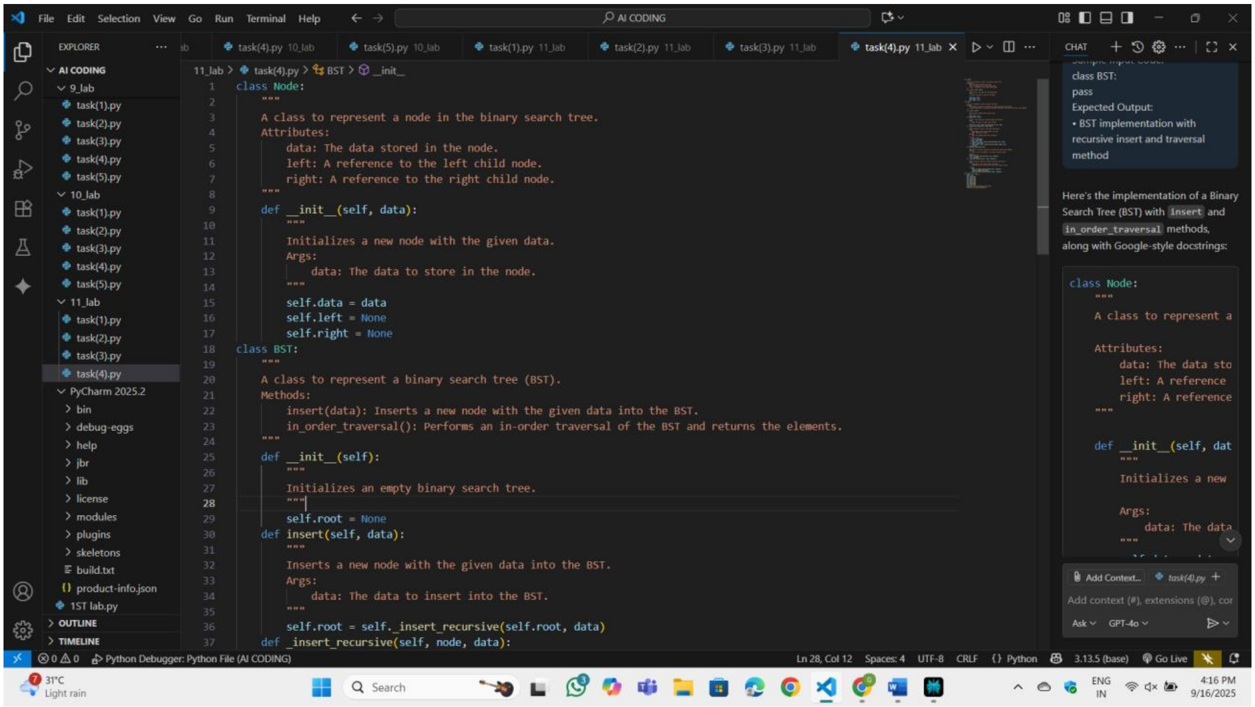
Task-4:

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:

AI ASSISTED CODING



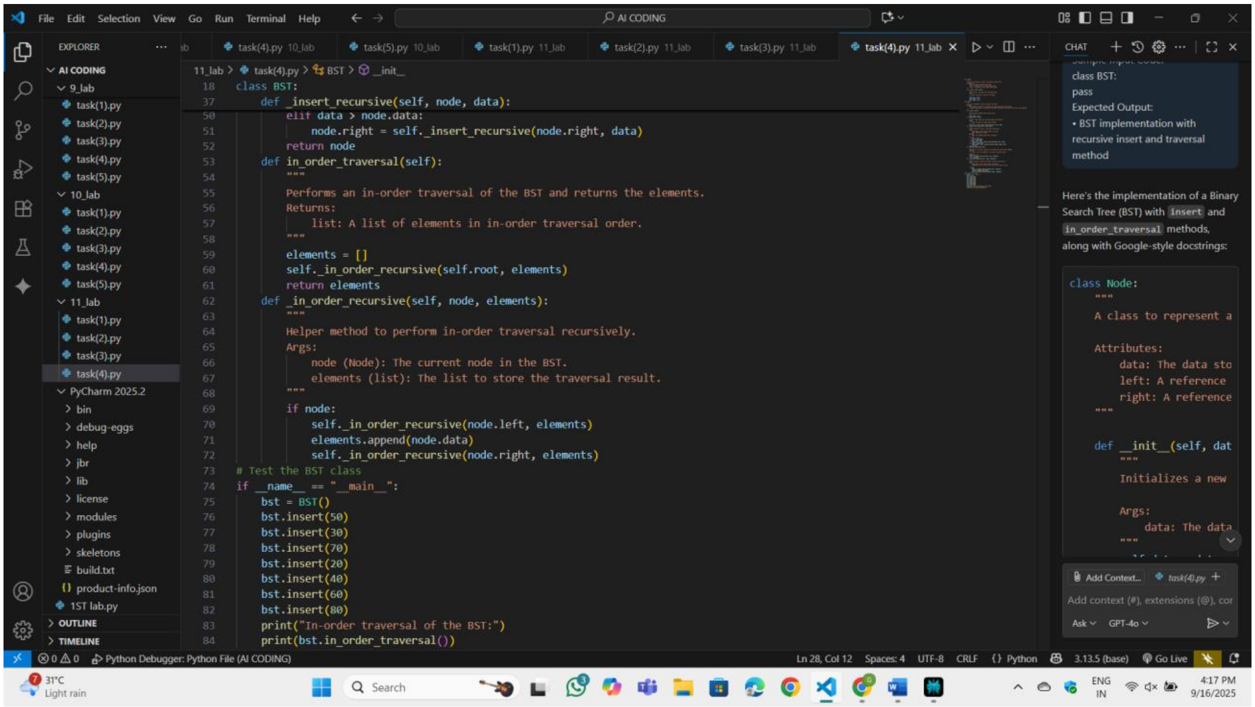
```
class Node:
    """
    A class to represent a node in the binary search tree.
    Attributes:
        data: The data stored in the node.
        left: A reference to the left child node.
        right: A reference to the right child node.
    """
    def __init__(self, data):
        """
        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:
pass
Expected Output:
- BST implementation with recursive insert and traversal method

Here's the implementation of a Binary Search Tree (BST) with `insert` and `in_order_traversal` methods, along with Google-style docstrings:

```
class Node:
    """
    A class to represent a

    Attributes:
        data: The data sto
        left: A reference
        right: A reference
    """

    def __init__(self, dat
        """
        Initializes a new

        Args:
            data: The data
        """
```

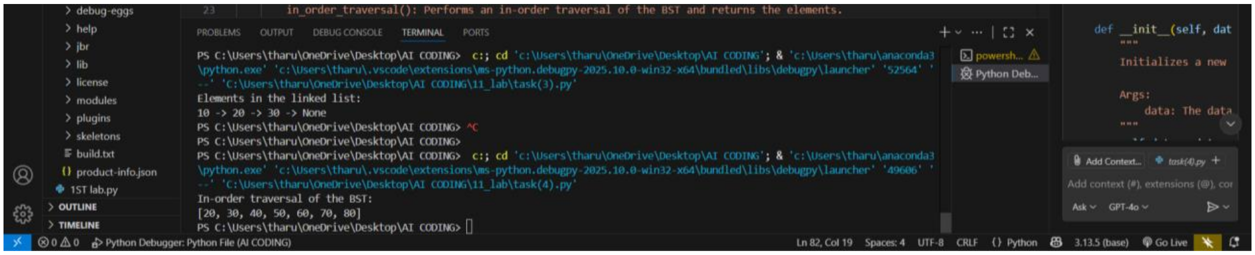
File Edit Selection View Go Run Terminal Help
AI CODING
task(4).py 10_lab
task(5).py 10_lab
task(1).py 11_lab
task(2).py 11_lab
task(3).py 11_lab
task(4).py 11_lab
11_lab > task(4).py > BST > __init__
CHAT
Add Context...
task(4).py
Add context (#), extensions (@), cor
Ask
GPT-4o
Ln 28, Col 12 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 (base) Go Live
Python Debugger: Python File (AI CODING)
31°C Light rain
Search
ENG IN
4:16 PM 9/16/2025



```
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())
```

AI CODING
task(4).py 11_lab
11_lab > task(4).py > BST > __init__
CHAT
Ln 28, Col 12 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 (base) Go Live
31°C Light rain
4:17 PM 9/16/2025

OP:



```
in_order_traversal(): Performs an in-order traversal of the BST and returns the elements.
```

```
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>  c:; cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Users\tharu\anaconda3
\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '52564' '
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(3).py'
Elements in the linked list:
10 -> 20 -> 30 -> None
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> ^C
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>  c:; cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Users\tharu\anaconda3
\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '49606' '
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(4).py'
In-order traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

```
def __init__(self, dat
    """
    Initializes a new

    Args:
        data: The data
    """
```

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:

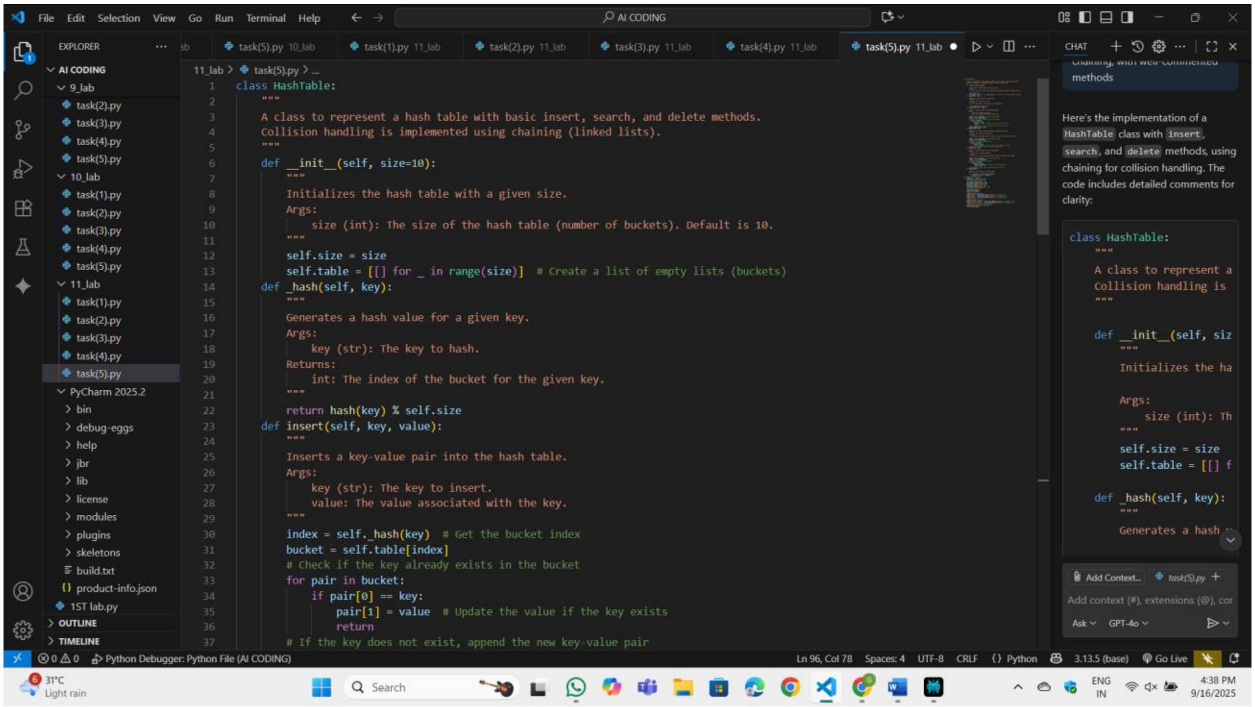
Task: Use AI to implement a hash table with basic insert, search, and delete

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

AI ASSISTED CODING

Code:

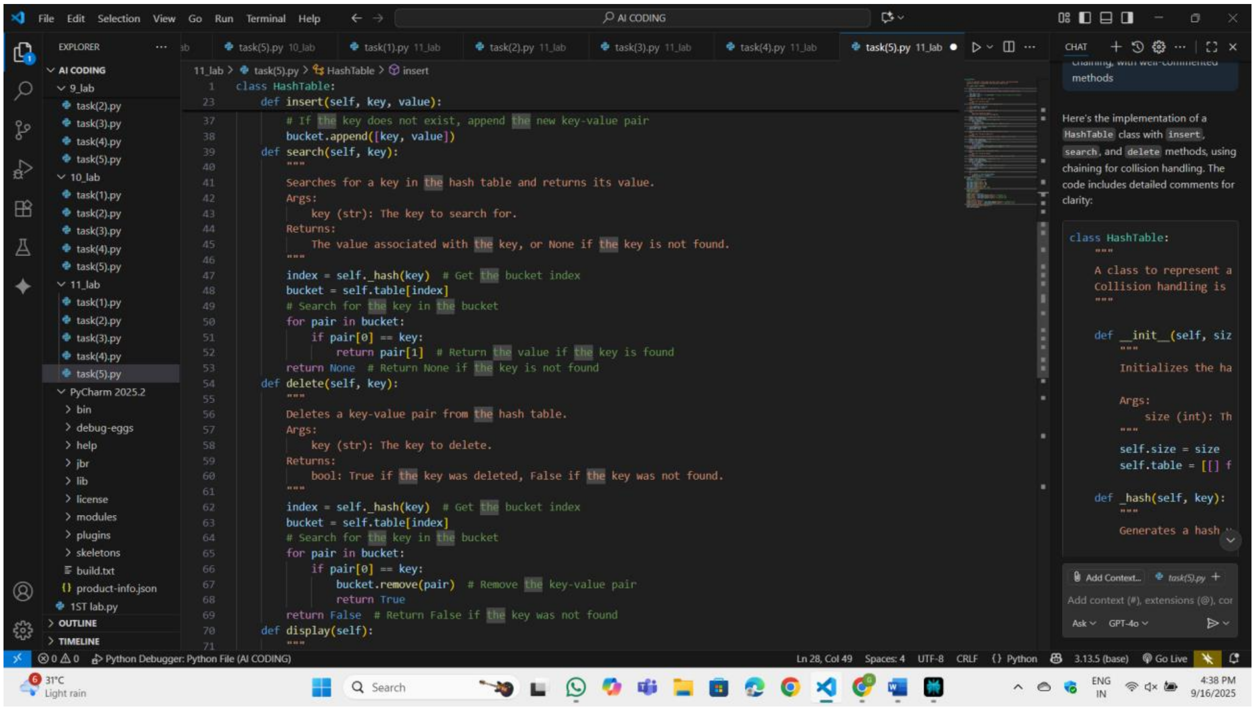


```python
class HashTable:
    """
    A class to represent a hash table with basic insert, search, and delete methods.
    Collision handling is implemented using chaining (linked lists).
    """
    def __init__(self, size=10):
        """
        Initializes the hash table with a given size.
        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
```

methods

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:

```python
class HashTable:
    """
    A class to represent a
    Collision handling is
    """

    def __init__(self, siz
        """
        Initializes the ha

        Args:
            size (int): Th
        """
        self.size = size
        self.table = [[] f

    def _hash(self, key):
        """
        Generates a hash
```

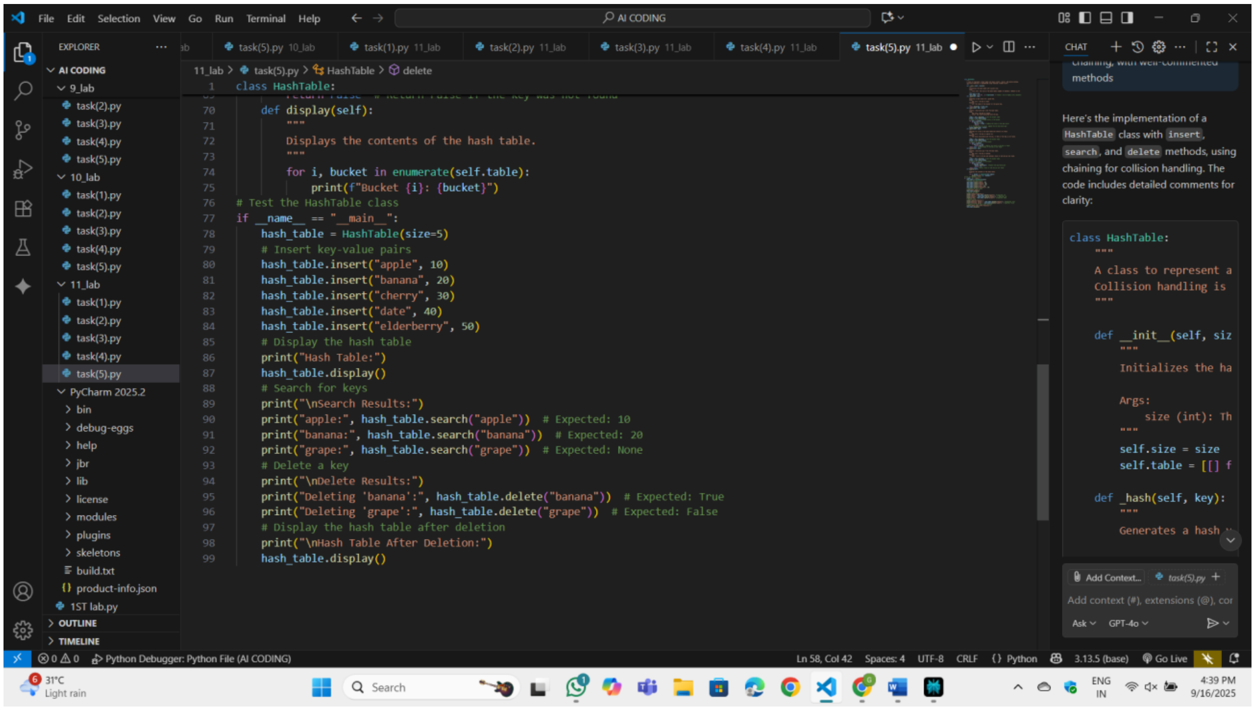


```python
class HashTable:
    def insert(self, key, value):
        # If the key does not exist, append the new key-value pair
        bucket.append([key, value])
    def search(self, key):
        """
        Searches for a key in the hash table and returns its value.
        Args:
            key (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):
        """
```

methods

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:

AI ASSISTED CODING



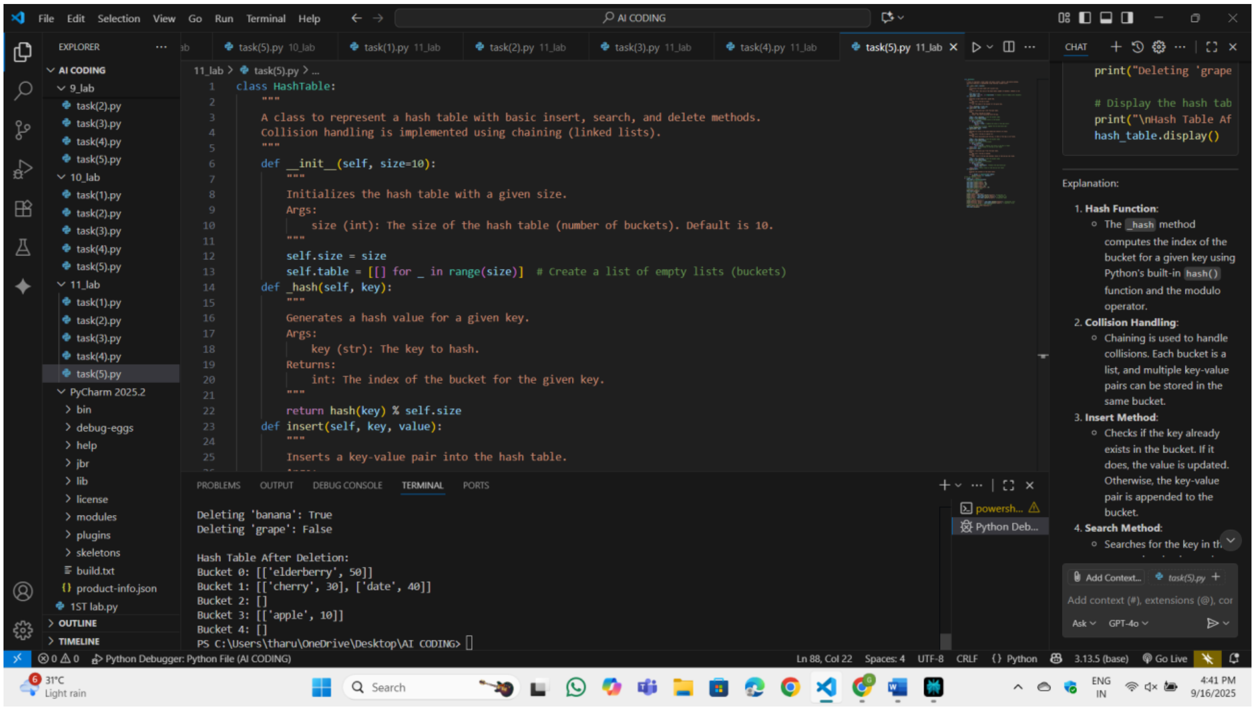
```
class HashTable:
    def display(self):
        """
        Displays the contents of the hash table.
        """
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")
# Test the HashTable class
if __name__ == "__main__":
    hash_table = HashTable(size=5)
    # Insert key-value pairs
    hash_table.insert("apple", 10)
    hash_table.insert("banana", 20)
    hash_table.insert("cherry", 30)
    hash_table.insert("date", 40)
    hash_table.insert("elderberry", 50)
    # Display the hash table
    print("Hash Table:")
    hash_table.display()
    # Search for keys
    print("\nSearch Results:")
    print("apple:", hash_table.search("apple"))  # Expected: 10
    print("banana:", hash_table.search("banana"))  # Expected: 20
    print("grape:", hash_table.search("grape"))  # Expected: None
    # Delete a key
    print("\nDelete Results:")
    print("Deleting 'banana':", hash_table.delete("banana"))  # Expected: True
    print("Deleting 'grape':", hash_table.delete("grape"))  # Expected: False
    # Display the hash table after deletion
    print("\nHash Table After Deletion:")
    hash_table.display()
```

methods

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:

```
class HashTable:
    """
    A class to represent a
    Collision handling is
    """

    def __init__(self, siz
        """
        Initializes the ha

        Args:
            size (int): Th
        """
        self.size = size
        self.table = [[] f

    def _hash(self, key):
        """
        Generates a hash
```

OP:



```
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.
```

```
Deleting 'banana': True
Deleting 'grape': False

Hash Table After Deletion:
Bucket 0: [['elderberry', 50]]
Bucket 1: [['cherry', 30], ['date', 40]]
Bucket 2: []
Bucket 3: [['apple', 10]]
Bucket 4: []
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
```

```
    print("Deleting 'grape

    # Display the hash tab
    print("\nHash Table Af
    hash_table.display()
```

Explanation:

1. **Hash Function**:
   - The `_hash` method computes the index of the bucket for a given key using Python's built-in `hash()` function and the modulo operator.
2. **Collision Handling**:
   - Chaining is used to handle collisions. Each bucket is a list, and multiple key-value pairs can be stored in the same bucket.
3. **Insert Method**:
   - Checks if the key already exists in the bucket. If it does, the value is updated. Otherwise, the key-value pair is appended to the bucket.
4. **Search Method**:
   - Searches for the key in th

Observation:

AI ASSISTED CODING

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
```

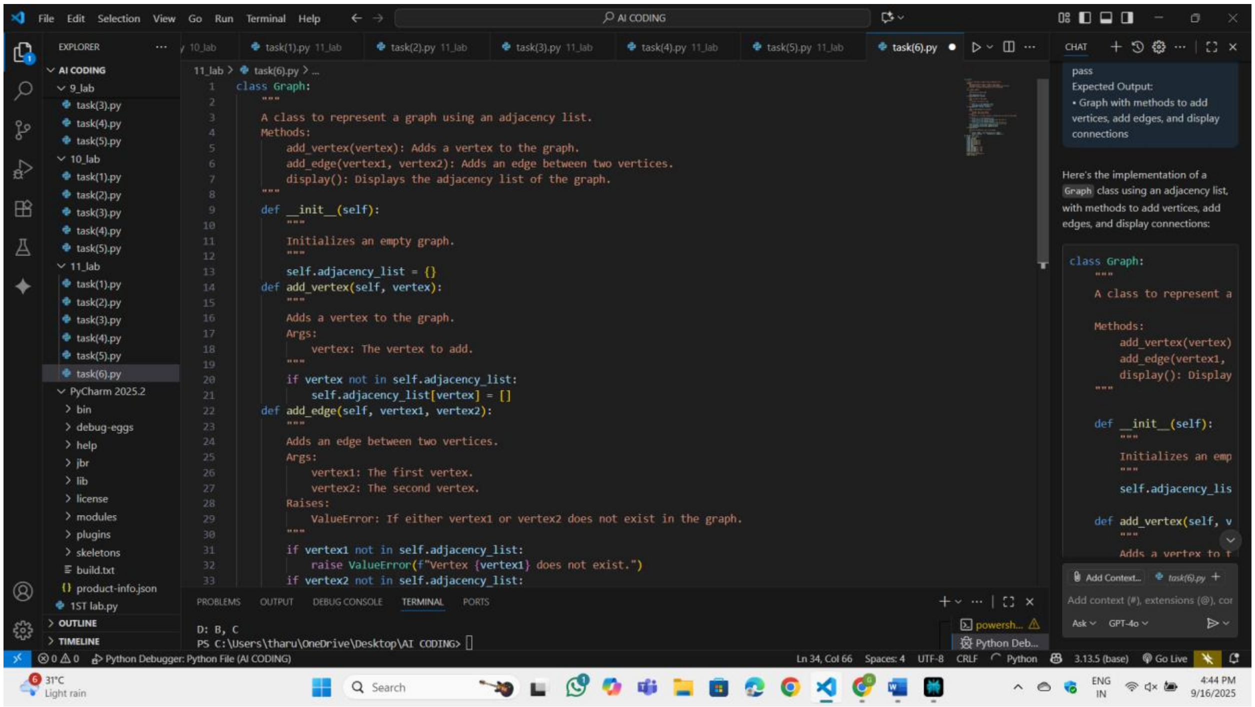
Expected Output:

- Graph with methods to add vertices, add edges, and display connections

Prompt: implement a graph using an adjacency list.

Code:

AI ASSISTED CODING



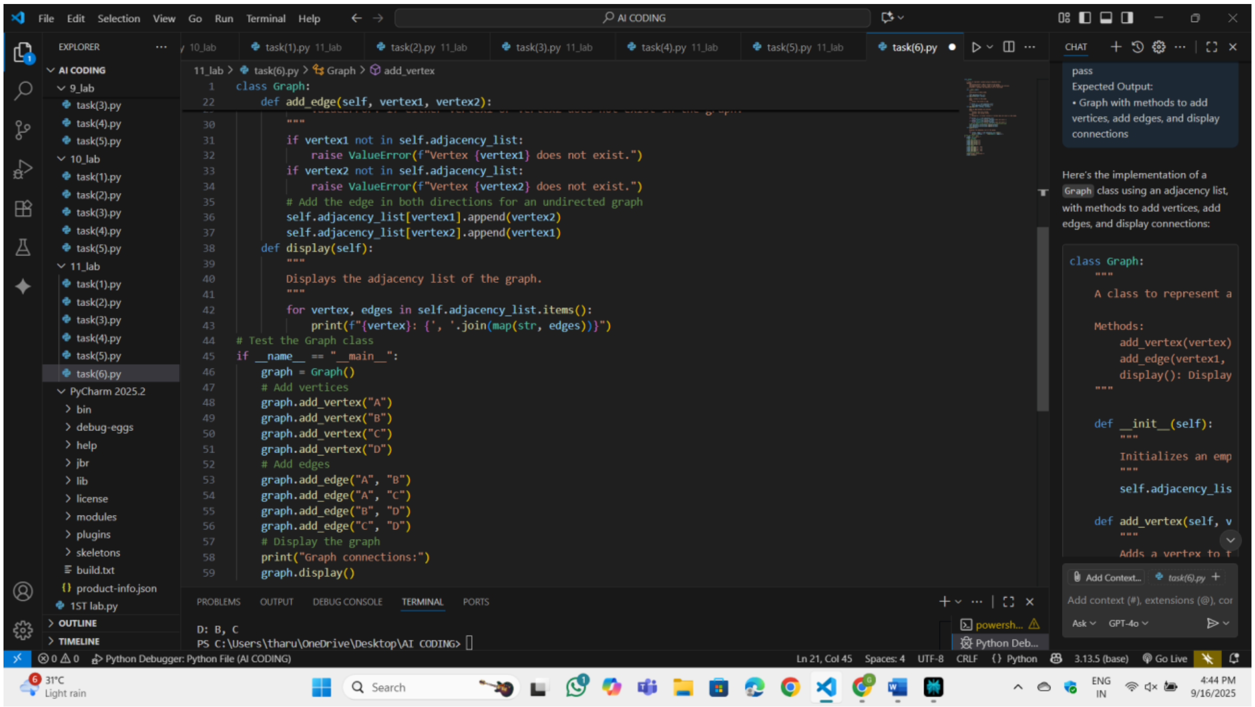
```
class Graph:
    """
    A class to represent a graph using an adjacency list.
    Methods:
        add_vertex(vertex): Adds a vertex to the graph.
        add_edge(vertex1, vertex2): Adds an edge between two vertices.
        display(): Displays the adjacency list of the graph.
    """
    def __init__(self):
        """
        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:
```

```
D: B, C
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

pass
Expected Output:
- Graph with methods to add vertices, add edges, and display connections

Here's the implementation of a `Graph` class using an adjacency list, with methods to add vertices, add edges, and display connections:

```
class Graph:
    """
    A class to represent a

    Methods:
        add_vertex(vertex)
        add_edge(vertex1,
        display(): Display
    """

    def __init__(self):
        """
        Initializes an emp
        """
        self.adjacency_lis

    def add_vertex(self, v
        """
        Adds a vertex to t
```



```
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()
```

```
D: B, C
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

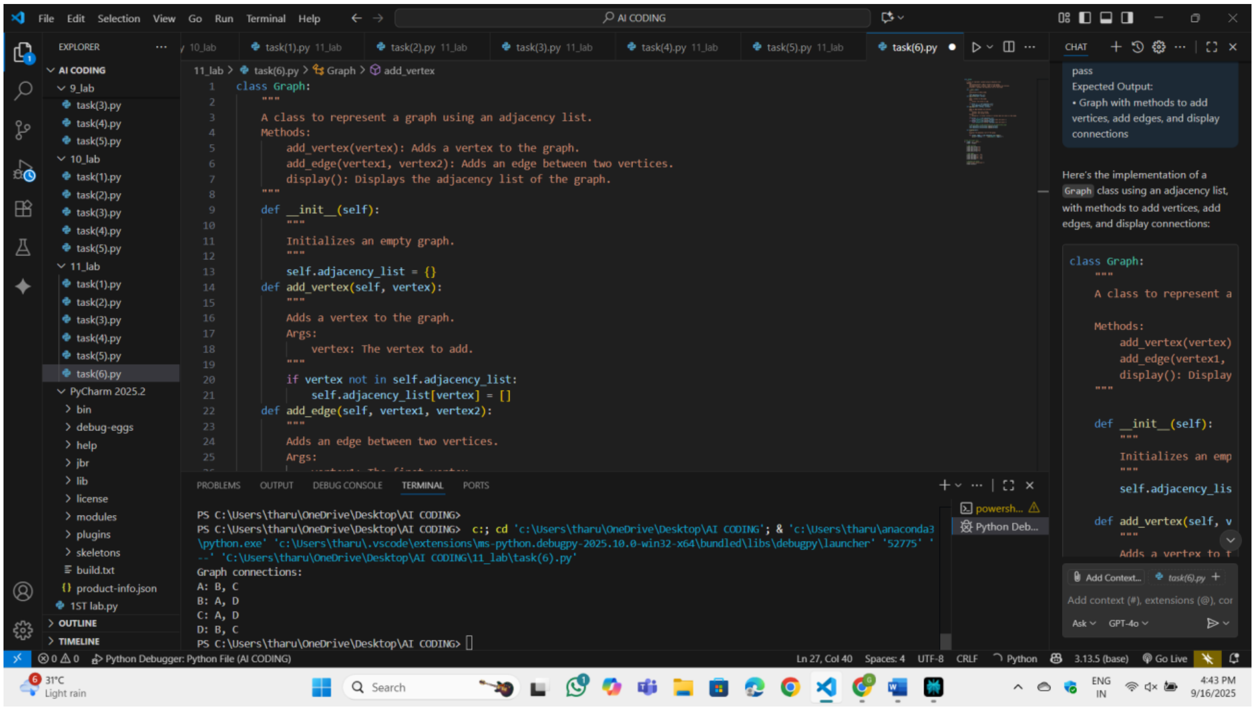
pass
Expected Output:
- Graph with methods to add vertices, add edges, and display connections

Here's the implementation of a `Graph` class using an adjacency list, with methods to add vertices, add edges, and display connections:

```
class Graph:
    """
    A class to represent a

    Methods:
        add_vertex(vertex)
        add_edge(vertex1,
        display(): Display
    """

    def __init__(self):
        """
        Initializes an emp
        """
        self.adjacency_lis

    def add_vertex(self, v
        """
        Adds a vertex to t
```

File Edit Selection View Go Run Terminal Help
AI CODING
EXPLORER
AI CODING
11_lab > task(6).py > Graph > add_vertex
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
CHAT
Add Context... task(6).py
Ask GPT-4o
Python Debugger: Python File (AI CODING)
Ln 21, Col 45 Spaces: 4 UTF-8 CRLF Python 3.13.5 (base) Go Live
31°C Light rain
4:44 PM 9/16/2025

OP:



```python
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:
```

```
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>
PS C:\Users\tharu\OneDrive\Desktop\AI CODING>  c:; cd 'c:\Users\tharu\OneDrive\Desktop\AI CODING'; & 'c:\Users\tharu\anaconda3
\python.exe' 'c:\Users\tharu\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '52775' '
--' 'C:\Users\tharu\OneDrive\Desktop\AI CODING\11_lab\task(6).py'
Graph connections:
A: B, C
B: A, D
C: A, D
D: B, C
PS C:\Users\tharu\OneDrive\Desktop\AI CODING> 
```

pass
Expected Output:
• Graph with methods to add vertices, add edges, and display connections

Here's the implementation of a `Graph` class using an adjacency list, with methods to add vertices, add edges, and display connections:

```
class Graph:
    """
    A class to represent a

    Methods:
        add_vertex(vertex)
        add_edge(vertex1,
        display(): Display
    """


    def __init__(self):
        """
        Initializes an emp
        """
        self.adjacency_lis

    def add_vertex(self, v
        """
        Adds a vertex to t
```

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
```

AI ASSISTED CODING

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

AI ASSISTED CODING

- 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: