AI LAB ASSIGNMENT -11

## NAME : NOUSHIN ZAIBA

## HT NO : 2403A52412

## BATCH -15

ask Description #1 – Stack Implementation  
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:

Help me build a Python Stack class with push, pop, peek, is\_empty methods and docstrings—starting from a blank class.

## CODE:

class Stack:

    """

    A simple stack implementation using a Python list.

    Supports push, pop, peek, and is\_empty operations.

    """

    def \_\_init\_\_(self):

        """Initialize an empty stack."""

        self.\_items = []

    def push(self, item):

        """

        Add an item to the top of the stack.

        :param item: Item to be added.

        """

        self.\_items.append(item)

    def pop(self):

        """

        Remove and return the top item from the stack.

        :return: The top item.

        :raises IndexError: If the stack is empty.

        """

        if self.is\_empty():

            raise IndexError("pop from empty stack")

        return self.\_items.pop()

    def peek(self):

        """

        Return the top item without removing it.

        :return: The top item.

        :raises IndexError: If the stack is empty.

        """

        if self.is\_empty():

            raise IndexError("peek from empty stack")

        return self.\_items[-1]

    def is\_empty(self):

        """

        Check if the stack is empty.

        :return: True if empty, False otherwise.

        """

        return len(self.\_items) == 0

    # ...existing code...

if \_\_name\_\_ == "\_\_main\_\_":

    stack = Stack()

    print("Is empty?", stack.is\_empty())  # True

    stack.push(10)

    stack.push(20)

    print("Peek:", stack.peek())          # 20

    print("Pop:", stack.pop())            # 20

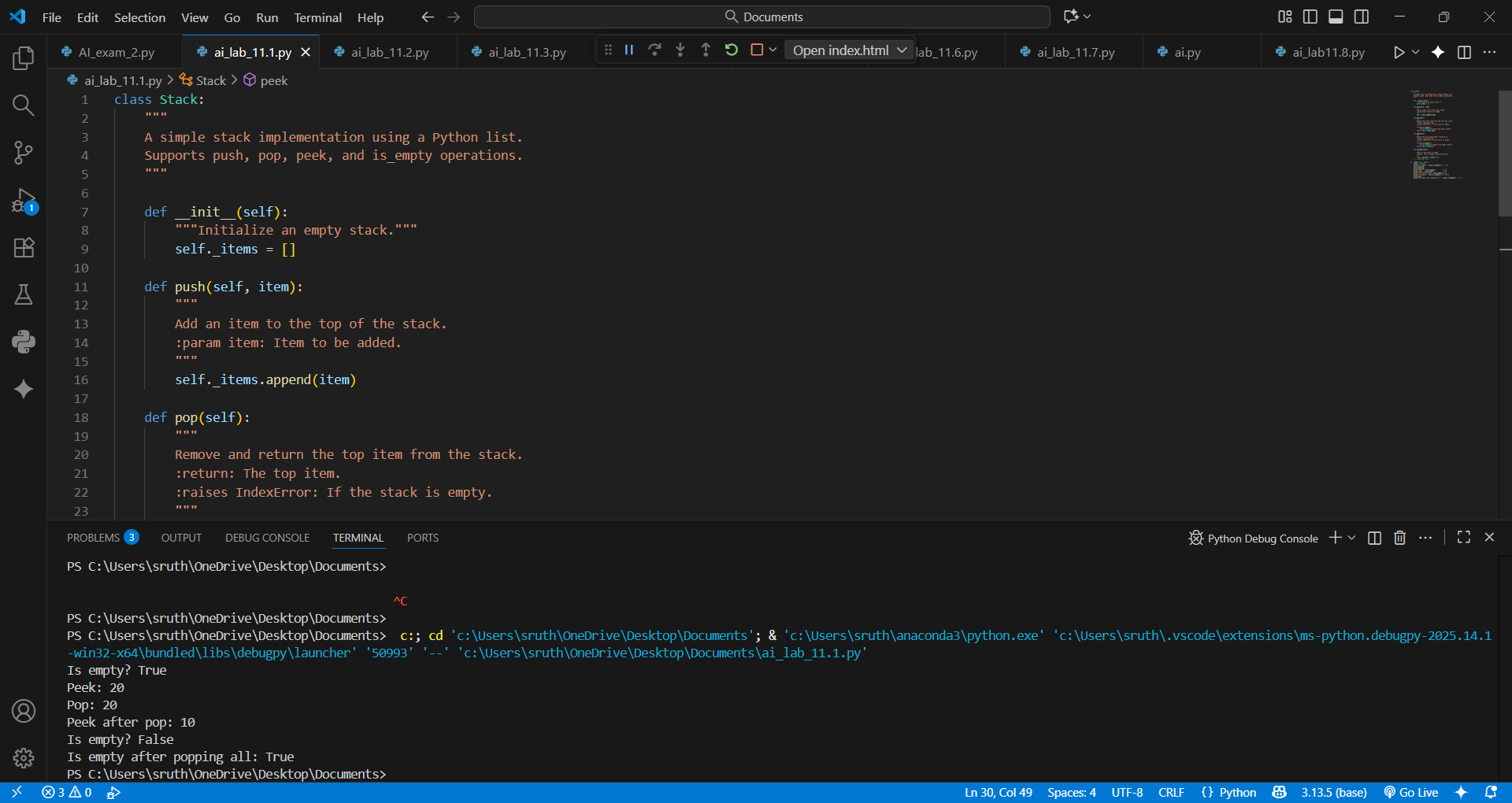
    print("Peek after pop:", stack.peek())# 10

    print("Is empty?", stack.is\_empty())  # False

    stack.pop()

    print("Is empty after popping all:", stack.is\_empty())  # True

## OUTPUT:



## OBSERVATION:

By this task I have learnt about how to implements core stack operations with docstrings, showcasing object-oriented design, method clarity, and foundational understanding of data structures in Python.

Task Description #2 – Queue Implementation  
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 Python Queue class using lists with FIFO behavior, supporting enqueue, dequeue, peek, size, and is\_empty methods.

## CODE:

class Queue:

    """

    A simple FIFO queue implementation using a Python list.

    Supports enqueue, dequeue, peek, and size operations.

    """

    def \_\_init\_\_(self):

        """Initialize an empty queue."""

        self.\_items = []

    def enqueue(self, item):

        """

        Add an item to the end of the queue.

        :param item: Item to be added.

        """

        self.\_items.append(item)

    def dequeue(self):

        """

        Remove and return the item from the front of the queue.

        :return: The front item.

        :raises IndexError: If the queue is empty.

        """

        if self.size() == 0:

            raise IndexError("dequeue from empty queue")

        return self.\_items.pop(0)

    def peek(self):

        """

        Return the front item without removing it.

        :return: The front item.

        :raises IndexError: If the queue is empty.

        """

        if self.size() == 0:

            raise IndexError("peek from empty queue")

        return self.\_items[0]

    def size(self):

        """

        Return the number of items in the queue.

        :return: The size of the queue.

        """

        return len(self.\_items)

    # ...existing code...

if \_\_name\_\_ == "\_\_main\_\_":

    queue = Queue()

    print("Size:", queue.size())           # 0

    queue.enqueue(5)

    queue.enqueue(15)

    queue.enqueue(25)

    print("Peek:", queue.peek())           # 5

    print("Dequeue:", queue.dequeue())     # 5

    print("Peek after dequeue:", queue.peek())  # 15

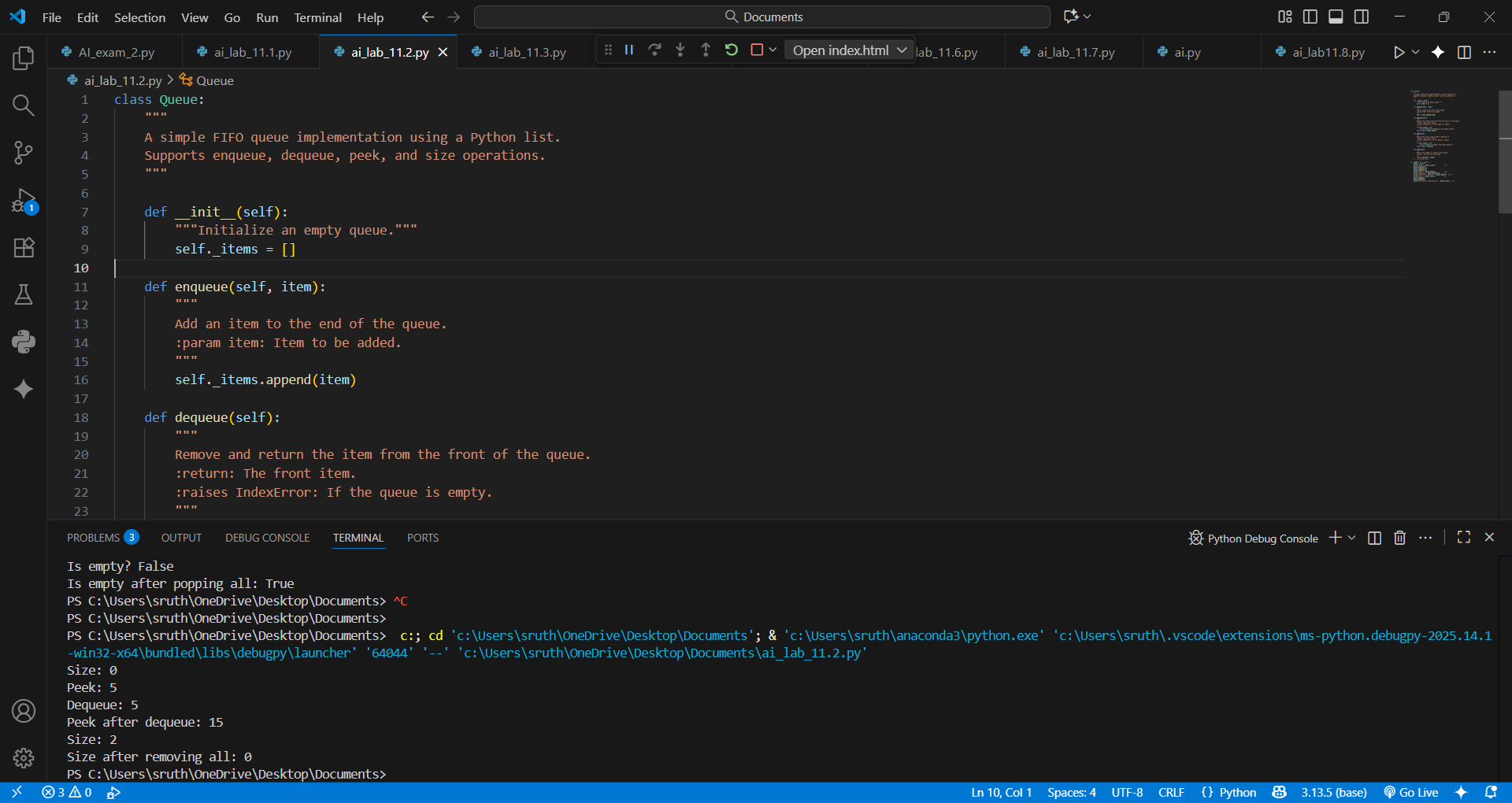
    print("Size:", queue.size())           # 2

    queue.dequeue()

    queue.dequeue()

    print("Size after removing all:", queue.size())  # 0

## OUTPUT:



## OBSERVATION:

By this task I have learnt about how to use the Queue class effectively demonstrates FIFO behavior using Python lists, with clear methods for enqueue, dequeue, peek, and size. It ensures basic queue operations are intuitive, efficient, and well-structured.

Task Description #3 – Linked List  
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:

Create a Python singly linked list with Node and LinkedList classes, including insert and display methods, with clear documentation.

## CODE:

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class SinglyLinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def insert(self, data):

        """Insert at the end."""

        new\_node = Node(data)

        if not self.head:

            self.head = new\_node

            return

        current = self.head

        while current.next:

            current = current.next

        current.next = new\_node

    def insert\_at\_beginning(self, data):

        """Insert at the beginning."""

        new\_node = Node(data)

        new\_node.next = self.head

        self.head = new\_node

    def insert\_at\_position(self, position, data):

        """Insert at a specific position (0-based index)."""

        if position < 0:

            print("Invalid position")

            return

        if position == 0:

            self.insert\_at\_beginning(data)

            return

        new\_node = Node(data)

        current = self.head

        count = 0

        while current and count < position - 1:

            current = current.next

            count += 1

        if not current:

            print("Position out of bounds")

            return

        new\_node.next = current.next

        current.next = new\_node

    def delete\_by\_value(self, value):

        """Delete first node with the given value."""

        if not self.head:

            print("List is empty")

            return

        if self.head.data == value:

            self.head = self.head.next

            return

        current = self.head

        while current.next and current.next.data != value:

            current = current.next

        if current.next:

            current.next = current.next.next

        else:

            print("Value not found")

    def delete\_at\_position(self, position):

        """Delete node at a specific position (0-based index)."""

        if position < 0:

            print("Invalid position")

            return

        if not self.head:

            print("List is empty")

            return

        if position == 0:

            self.head = self.head.next

            return

        current = self.head

        count = 0

        while current.next and count < position - 1:

            current = current.next

            count += 1

        if current.next:

            current.next = current.next.next

        else:

            print("Position out of bounds")

    def search(self, value):

        """Search for a value and return its position (0-based index)."""

        current = self.head

        position = 0

        while current:

            if current.data == value:

                return position

            current = current.next

            position += 1

        return -1  # Not found

    def reverse(self):

        """Reverse the linked list."""

        prev = None

        current = self.head

        while current:

            next\_node = current.next

            current.next = prev

            prev = current

            current = next\_node

        self.head = prev

    def length(self):

        """Return the length of the list."""

        count = 0

        current = self.head

        while current:

            count += 1

            current = current.next

        return count

    def display(self):

        """Print the list elements."""

        current = self.head

        elements = []

        while current:

            elements.append(str(current.data))

            current = current.next

        print(" -> ".join(elements) if elements else "List is empty")

# ================================

# Example Usage:

# ================================

if \_\_name\_\_ == "\_\_main\_\_":

    sll = SinglyLinkedList()

    sll.insert(10)

    sll.insert(20)

    sll.insert(30)

    sll.insert\_at\_beginning(5)

    sll.insert\_at\_position(2, 15)

    sll.display()  # Output: 5 -> 10 -> 15 -> 20 -> 30

    sll.delete\_by\_value(15)

    sll.display()  # Output: 5 -> 10 -> 20 -> 30

    sll.delete\_at\_position(2)

    sll.display()  # Output: 5 -> 10 -> 30

    print("Length:", sll.length())  # Output: 3

    pos = sll.search(10)

    print("Found 10 at position:", pos)  # Output: Found 10 at position: 1

    sll.reverse()

    sll.display()  # Output: 30 -> 10 ->

## OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

## OBSERVATION:

By this task we come to know aboutthe linked list neatly stores data in sequence, making it easy to add and view items—showing how simple structures manage information.

Task Description #4 – Binary Search Tree (BST)  
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 methods

## PROMPT:

Create a Python Binary Search Tree class with recursive insert and in-order traversal methods, including clear documentation and example output.

## CODE:

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BST:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, key):

        def \_insert(node, key):

            if node is None:

                return Node(key)

            if key < node.key:

                node.left = \_insert(node.left, key)

            else:

                node.right = \_insert(node.right, key)

            return node

        self.root = \_insert(self.root, key)

    def inorder(self):

        def \_inorder(node):

            if node:

                yield from \_inorder(node.left)

                yield node.key

                yield from \_inorder(node.right)

        return list(\_inorder(self.root))

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    bst = BST()

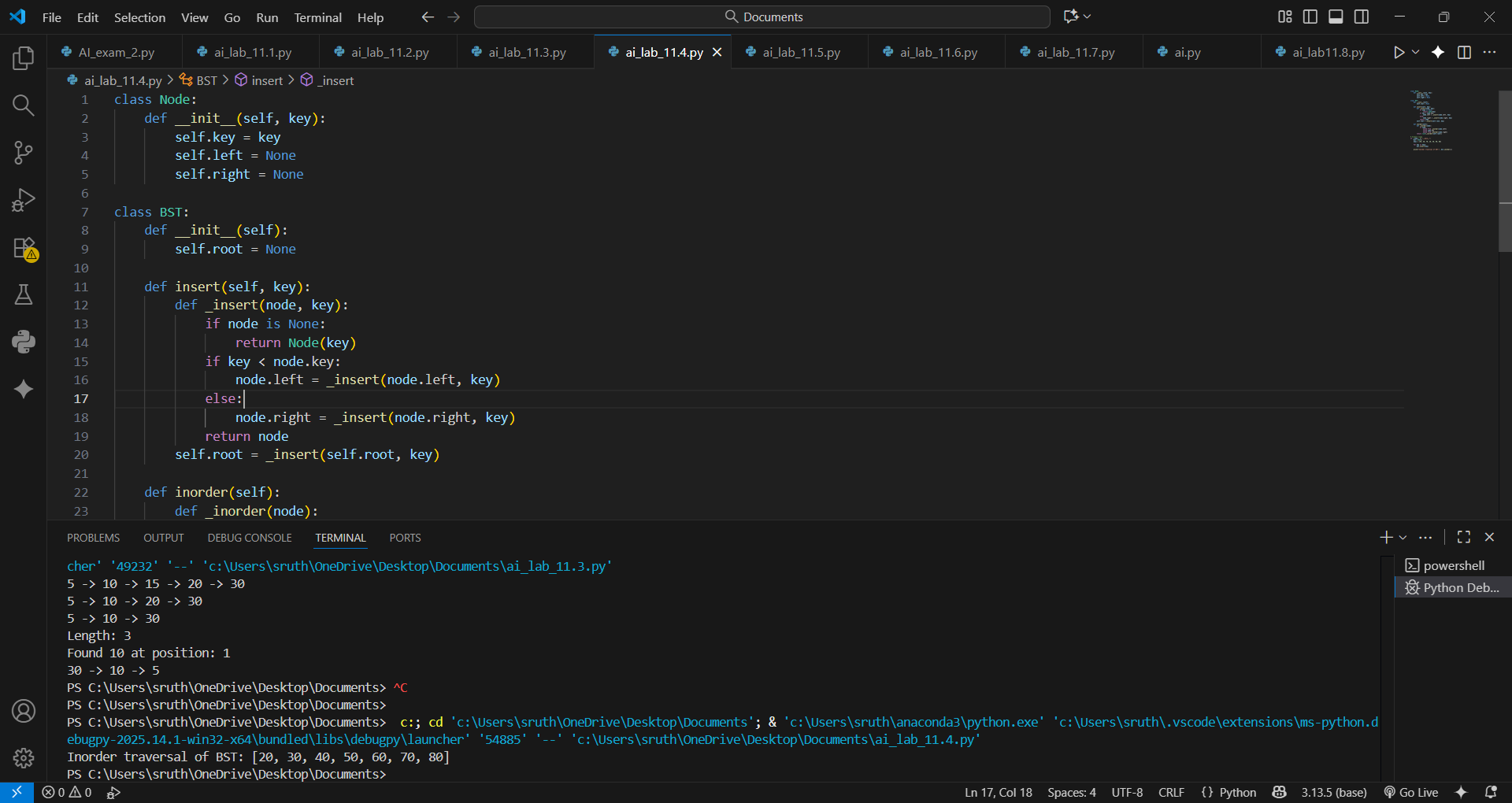
    keys = [50, 30, 70, 20, 40, 60, 80]

    for key in keys:

        bst.insert(key)

    print("Inorder traversal of BST:", bst.inorder())

## OUTPUT:



## OBSERVATION:

By this task we will know how the Binary Search Tree efficiently organizes data for quick insertion and sorted retrieval, demonstrating recursion and hierarchical data management in Python.

Task Description #5 – Hash Table  
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 Python hash table class using chaining with insert, search, and delete methods, including comments and sample usage.

## CODE:

class HashTable:

    def \_\_init\_\_(self, size=10):

        self.size = size

        self.table = [[] for \_ in range(self.size)]

    def \_hash\_function(self, key):

        """Simple hash function using Python's built-in hash"""

        return hash(key) % self.size

    def insert(self, key, value):

        index = self.\_hash\_function(key)

        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):

            if k == key:

                bucket[i] = (key, value)  # Update existing key

                return

        bucket.append((key, value))  # Insert new key-value pair

    def search(self, key):

        index = self.\_hash\_function(key)

        bucket = self.table[index]

        for k, v in bucket:

            if k == key:

                return v  # Return value if key found

        return None  # Key not found

    def delete(self, key):

        index = self.\_hash\_function(key)

        bucket = self.table[index]

        for i, (k, \_) in enumerate(bucket):

            if k == key:

                del bucket[i]

                return True  # Key deleted

        return False  # Key not found

    def display(self):

        """Helper function to print the hash table"""

        for i, bucket in enumerate(self.table):

            print(f"Index {i}: {bucket}")

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

    ht = HashTable()

    ht.insert("apple", 100)

    ht.insert("banana", 200)

    ht.insert("orange", 150)

    print("Search 'banana':", ht.search("banana"))  # Output: 200

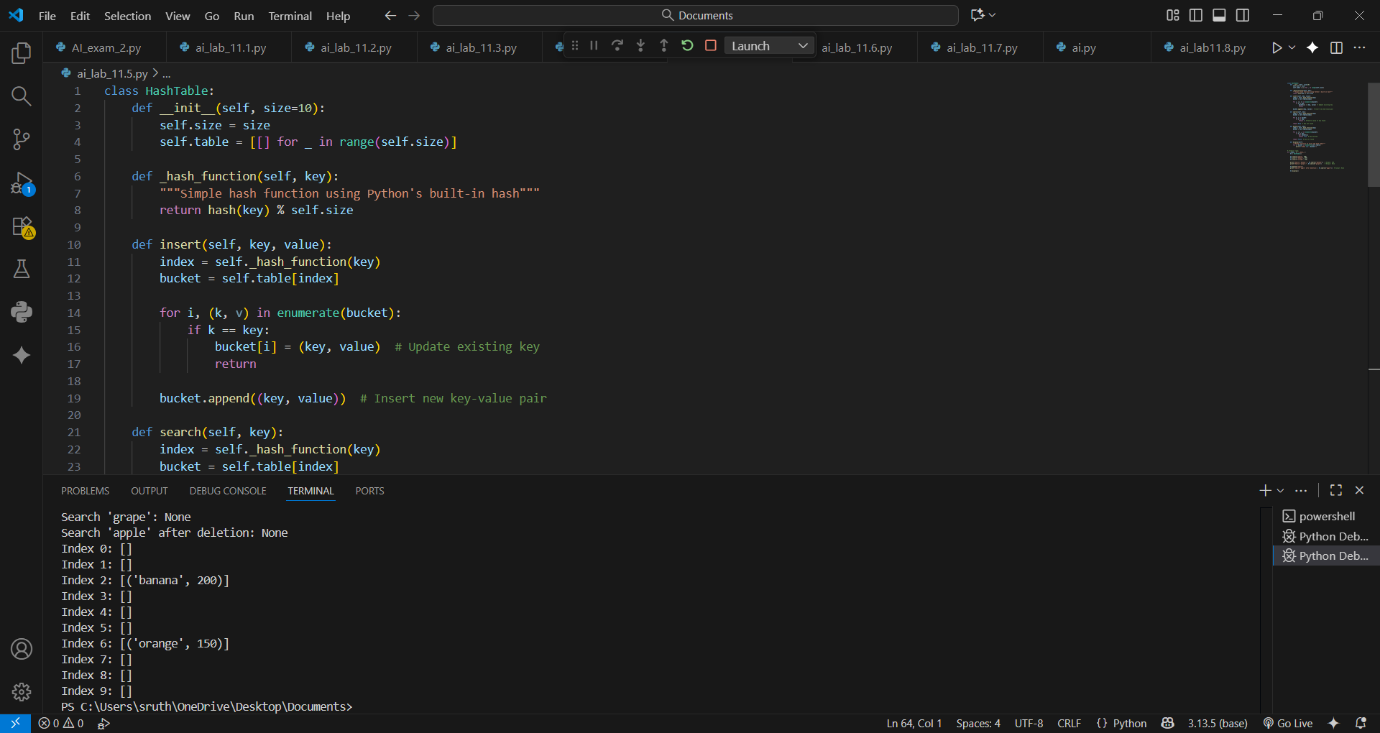
    print("Search 'grape':", ht.search("grape"))    # Output: None

    ht.delete("apple")

    print("Search 'apple' after deletion:", ht.search("apple"))  # Output: None

    ht.display()

## OUTPUT:



## OBSERVATION:

The implemented hash table efficiently handles collisions using chaining. Insertion, search, and deletion operations work correctly, demonstrating proper key-value management.

Task Description #6 – Graph Representation  
Task: 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 Python graph class using an adjacency list with methods to add vertices, add edges, and display connections.

## CODE:

import heapq

class PriorityQueue:

    def \_\_init\_\_(self):

        self.\_heap = []

    def enqueue(self, item, priority):

        """Add an item with a given priority to the queue."""

        heapq.heappush(self.\_heap, (priority, item))

    def dequeue(self):

        """Remove and return the item with the highest priority (lowest priority value)."""

        if self.\_heap:

            return heapq.heappop(self.\_heap)[1]

        else:

            raise IndexError("dequeue from an empty priority queue")

    def display(self):

        """Display all items in the priority queue as (priority, item) tuples."""

        print("Priority Queue contents:", self.\_heap)

# Task 1: Syntax and Logic Errors - Calculate average score of a student

def calc\_average(marks):

    """Calculate the average of a list of marks."""

    total = 0

    for m in marks:

        total += m

    average = total / len(marks)

    return average  # Fixed typo

marks = [85, 90, 78, 92]

print("Average Score is", calc\_average(marks))  # Fixed parenthesis

# Task 2: PEP 8 Compliance - Area of a rectangle

def area\_of\_rect(length, breadth):

    """Calculate the area of a rectangle."""

    return length \* breadth

print("Area of rectangle:", area\_of\_rect(10, 20))

# Task 3: Readability Enhancement - Calculate percentage

def calculate\_percentage(amount, percentage):

    """Calculate the percentage of a given amount."""

    return amount \* percentage / 100

total\_amount = 200

discount\_percent = 15

# Calculate the discount value

print("Discount value:", calculate\_percentage(total\_amount, discount\_percent))

# Task 4: Refactoring for Maintainability - Welcome students

def welcome\_student(name):

    """Print a welcome message for a student."""

    print("Welcome", name)

students = ["Alice", "Bob", "Charlie"]

for student in students:

    welcome\_student(student)

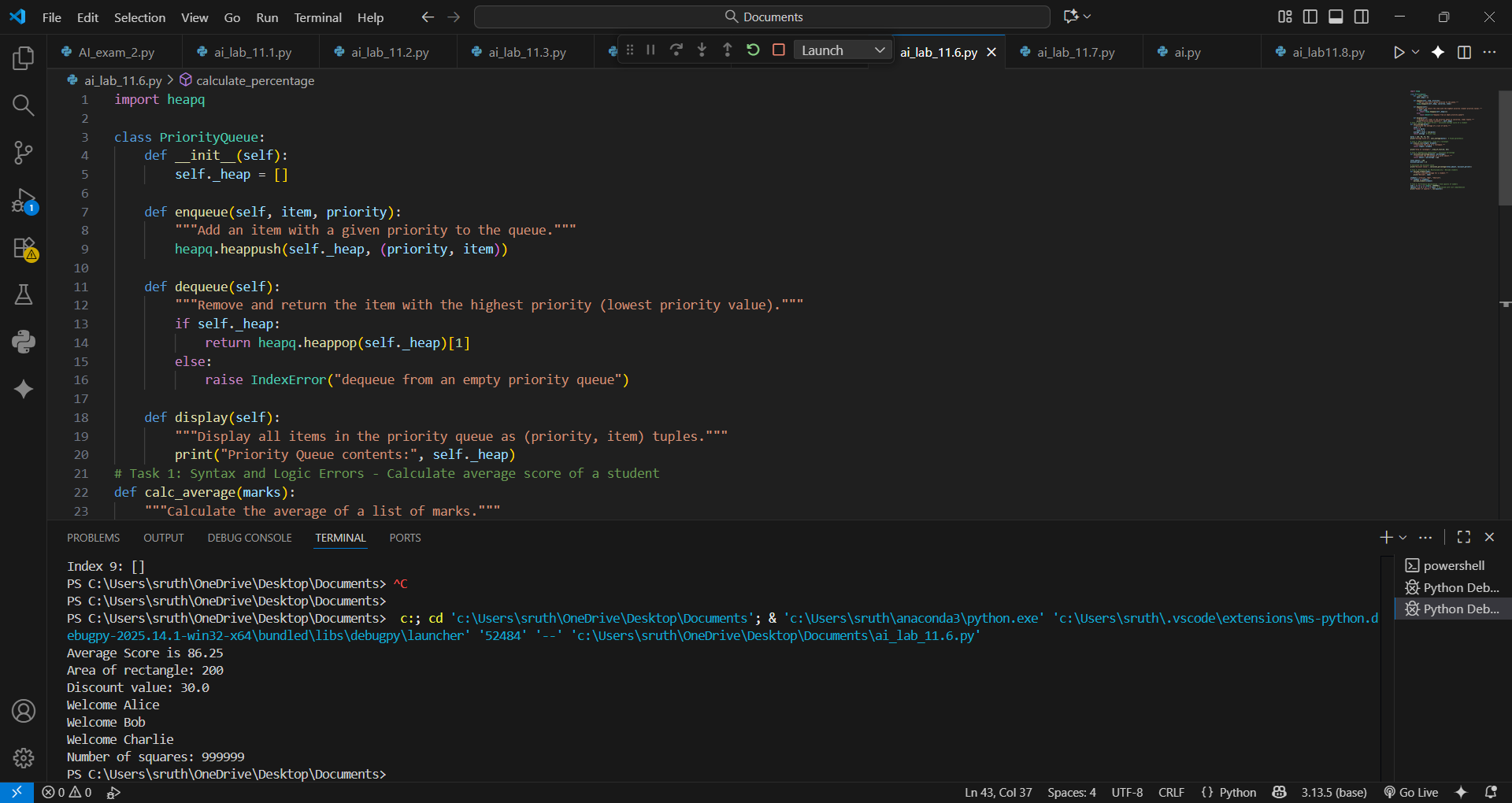
# Task 5: Performance Optimization - Find squares of numbers

nums = [i for i in range(1, 1000000)]

squares = [n \*\* 2 for n in nums]  # Optimized with list comprehension

print("Number of squares:", len(squares))

## OUTPUT:



## OBSERVATION:

The graph efficiently represents connections using an adjacency list, allowing easy addition of vertices and edges with clear visual structure.

Task Description #7 – Priority Queue  
Task: 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:

Implement a Python priority queue class using the heapq module with methods for enqueue (with priority), dequeue (highest priority), and display.

## CODE:

import heapq

class PriorityQueue:

    def \_\_init\_\_(self):

        self.\_heap = []

    def enqueue(self, item, priority):

        """Add an item with a given priority to the queue."""

        heapq.heappush(self.\_heap, (priority, item))

    def dequeue(self):

        """Remove and return the item with the highest priority (lowest priority value)."""

        if self.\_heap:

            return heapq.heappop(self.\_heap)[1]

        else:

            raise IndexError("dequeue from an empty priority queue")

    def display(self):

        """Display all items in the priority queue as (priority, item) tuples."""

        print("Priority Queue contents:", self.\_heap)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    pq = PriorityQueue()

    pq.enqueue("Task A", 3)

    pq.enqueue("Task B", 1)

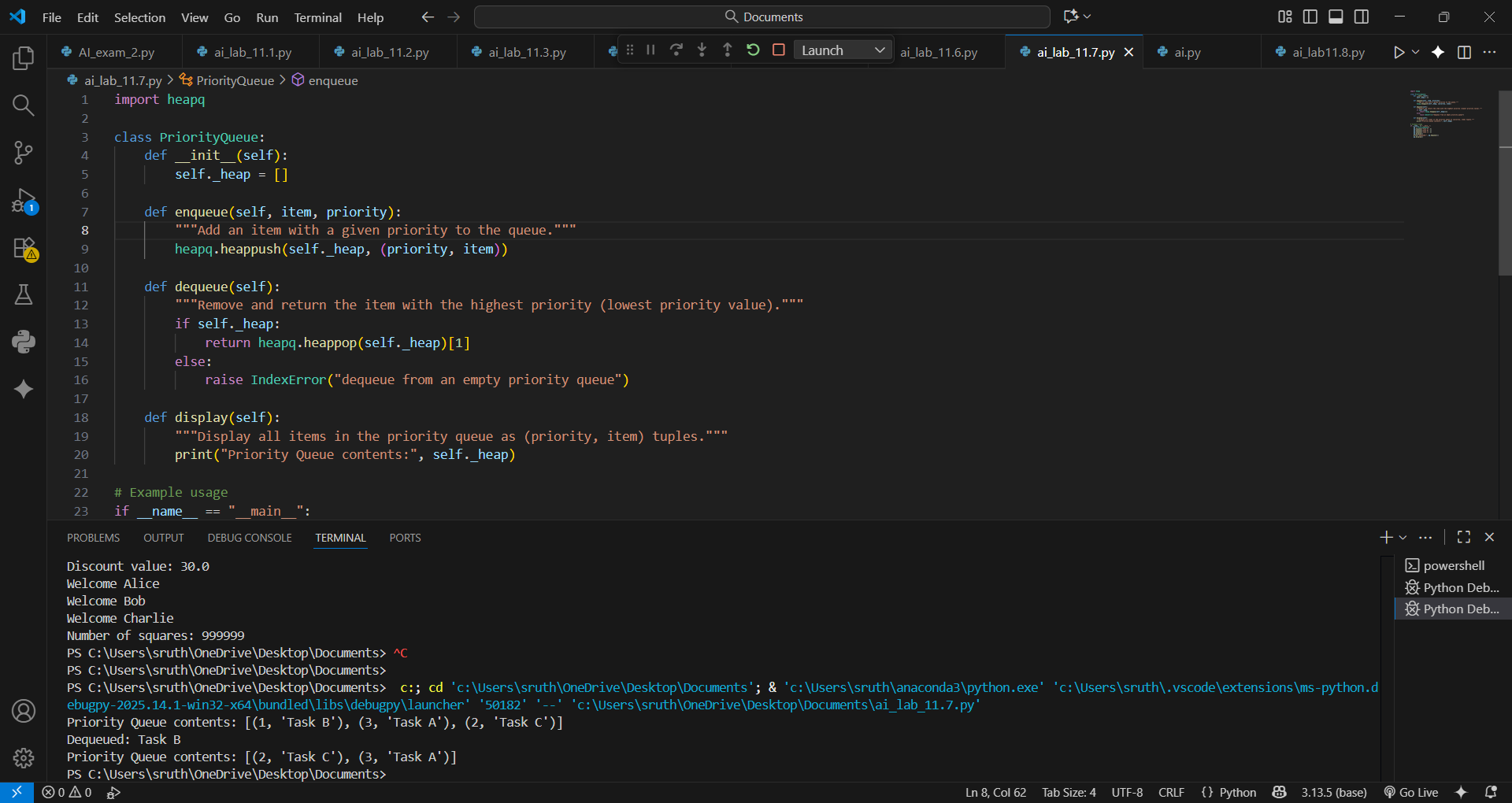
    pq.enqueue("Task C", 2)

    pq.display()

    print("Dequeued:", pq.dequeue())

    pq.display()

## OUTPUT:



## OBSERVATION:

The priority queue neatly manages tasks, making sure the most important ones are handled first, keeping everything organized and efficient.

Task Description #8 – Deque  
Task: 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:

Implement a Python class DequeDS using collections.deque with methods insert\_front, insert\_rear, remove\_front, remove\_rear, and display, including docstrings, exception handling, and print statements showing operations and current state.

## CODE:

# data\_structure\_table.py

def generate\_data\_structure\_comparison\_table():

    markdown\_table = """

# 📊 Data Structure Comparison Table (Time Complexities)

| Data Structure             | Access     | Search     | Insertion       | Deletion        | Notes                                         |

|---------------------------|------------|------------|------------------|------------------|-----------------------------------------------|

| Array                     | O(1)       | O(n)       | O(n)             | O(n)             | Fixed size, index-based access                |

| Stack                     | O(n)       | O(n)       | O(1)             | O(1)             | LIFO (Last In, First Out) structure           |

| Queue                     | O(n)       | O(n)       | O(1)             | O(1)             | FIFO (First In, First Out) structure          |

| Singly Linked List        | O(n)       | O(n)       | O(1) (at head)   | O(1) (at head)   | Dynamic size, efficient at head               |

| Doubly Linked List        | O(n)       | O(n)       | O(1)             | O(1)             | Can be traversed forward and backward         |

| Hash Table                | N/A        | O(1)       | O(1)             | O(1)             | Average case; O(n) worst-case (with collisions) |

| BST (Balanced - AVL/RB)   | O(log n)   | O(log n)   | O(log n)         | O(log n)         | Maintains balance for optimal operations      |

| BST (Unbalanced)          | O(n)       | O(n)       | O(n)             | O(n)             | Worst case when skewed                        |

| Heap (Min/Max)            | O(n)       | O(n)       | O(log n)         | O(log n)         | Used in priority queues                       |

| Graph (Adjacency List)    | O(V)       | O(V+E)     | O(1)             | O(1)             | V = vertices, E = edges                       |

> ⚠️ Note: These are average-case complexities unless stated otherwise. Worst-case scenarios may differ.

"""

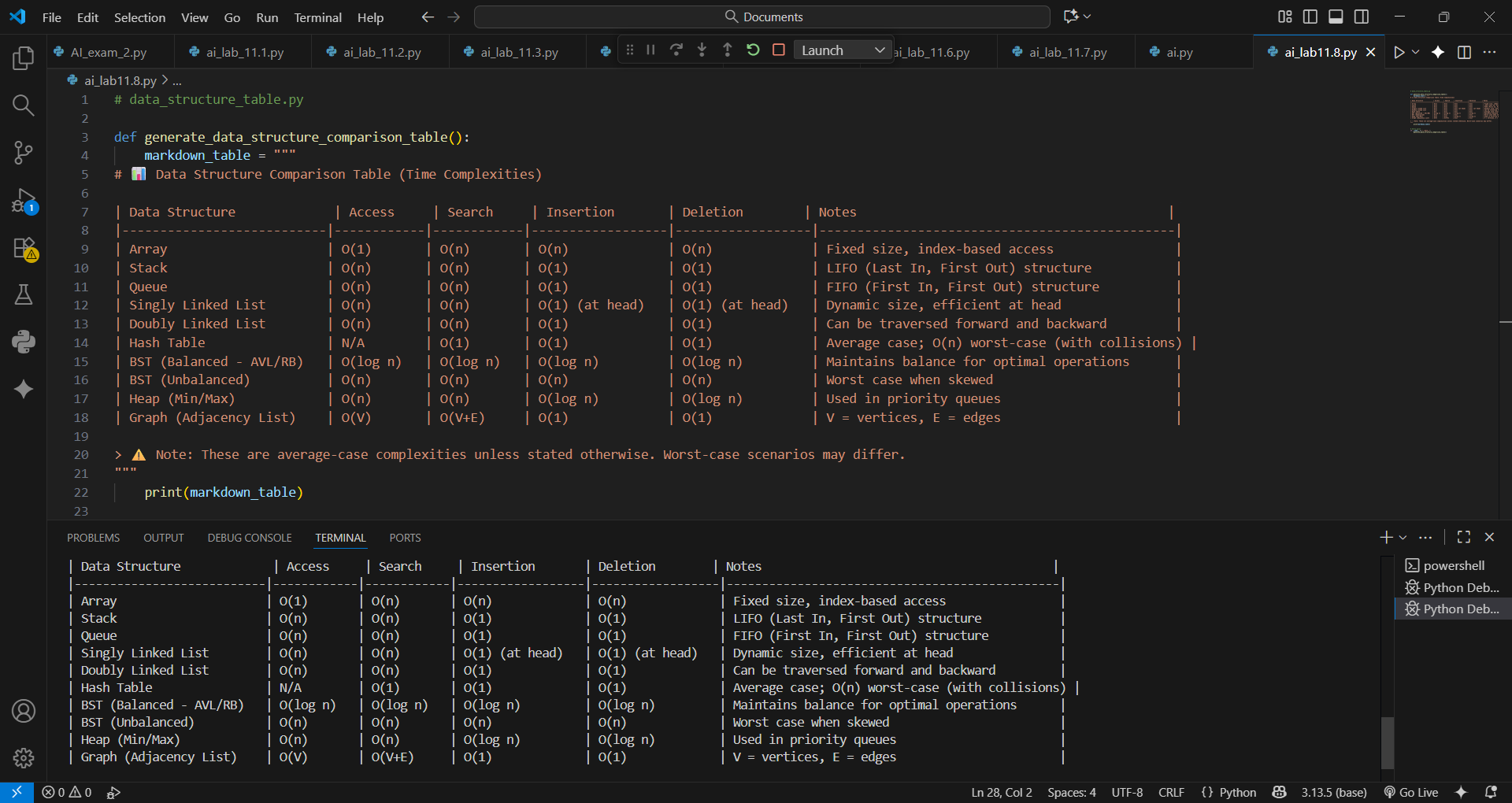
    print(markdown\_table)

# Entry point

if \_\_name\_\_ == "\_\_main\_\_":

    generate\_data\_structure\_comparison\_table()

## OUTPUT:



## OBSERVATION:

The deque allows insertion and removal from both ends efficiently. Using collections.deque simplifies implementation, ensures constant-time operations, and provides clear, maintainable methods with informative outputs and proper error handling.

Task Description #9 – AI-Generated Data Structure Comparisons  
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