

Week: 11.1 AI ASSIST

2303A52281

T. Vikas

TASK-1

Scenario:

In this task, we used AI to design and implement a Stack data structure in Python. The objective was to understand how a stack follows the LIFO (Last In, First Out) principle and how AI can assist in writing structured, optimized, and well-documented code. Instead of manually implementing the class, AI was prompted to generate a complete and functional Stack with essential operations.

Prompt:

“Generate a beginner-friendly Python Stack class using a list. Include push, pop, peek, and is_empty methods. Add proper docstrings and handle edge cases like popping from an empty stack.”

Code:

The image shows two side-by-side code editors, both titled "AI Assisted Code".

Code Editor 1 (Top):

```
class Stack:
    """Stack data structure implementation using a list.

    A Stack follows the LIFO (Last In, First Out) principle where
    the last element added to the stack will be removed.

    Attributes:
        items (list): List of elements stored in the stack.

    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 without removing it.
        is_empty(): Checks if the stack is empty.

    Examples:
        stack = Stack()
        stack.push(1)
        stack.push(2)
        print(stack.pop())
        # Output: 2
        print(stack.pop())
        # Output: 1
        print(stack.is_empty())
        # Output: True
    """
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        return len(self.items) == 0
```

Code Editor 2 (Bottom):

```
class Stack:
    """Stack data structure implementation using a list.

    A Stack follows the LIFO (Last In, First Out) principle where
    the last element added to the stack will be removed.

    Attributes:
        items (list): List of elements stored in the stack.

    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 without removing it.
        is_empty(): Checks if the stack is empty.

    Examples:
        stack = Stack()
        stack.push(1)
        stack.push(2)
        print(stack.pop())
        # Output: 2
        print(stack.pop())
        # Output: 1
        print(stack.is_empty())
        # Output: True
    """
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        return len(self.items) == 0
```

Both code editors show identical AI-generated code for a Stack class. The interface includes a sidebar with "Implementation Summary" and "Code Methods" sections, and a status bar at the bottom.

The image displays three vertically stacked screenshots of the Visual Studio Code (VS Code) interface, illustrating the AI's progress in generating a functional Stack implementation.

Top Window: Shows the initial AI-generated code for a Stack implementation. The code includes basic methods like `push` and `pop`, along with test cases for `empty()`, `size()`, and `top()`. A detailed description of the Stack class is present in the docstring.

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            return None

    def peek(self):
        if len(self.stack) > 0:
            return self.stack[-1]
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def __str__(self):
        return str(self.stack)

    def __repr__(self):
        return f"Stack({self.stack})"

# Test cases
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
stack.push(5)
print(stack) # Stack([1, 2, 3, 4, 5])
print(stack.pop()) # 5
print(stack) # Stack([1, 2, 3, 4])
print(stack.pop()) # 4
print(stack) # Stack([1, 2, 3])
print(stack.pop()) # 3
print(stack) # Stack([1, 2])
print(stack.pop()) # 2
print(stack) # Stack([1])
print(stack.pop()) # 1
print(stack) # Stack()
print(stack.is_empty()) # True
print(stack.size()) # 0
```

Middle Window: Shows the AI's addition of a `clear()` method and the removal of the `__str__` and `__repr__` methods. The test cases remain the same.

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            return None

    def peek(self):
        if len(self.stack) > 0:
            return self.stack[-1]
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def clear(self):
        self.stack = []

    def __str__(self):
        return str(self.stack)

    def __repr__(self):
        return f"Stack({self.stack})"

# Test cases
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
stack.push(5)
print(stack) # Stack([1, 2, 3, 4, 5])
print(stack.pop()) # 5
print(stack) # Stack([1, 2, 3, 4])
print(stack.pop()) # 4
print(stack) # Stack([1, 2, 3])
print(stack.pop()) # 3
print(stack) # Stack([1, 2])
print(stack.pop()) # 2
print(stack) # Stack([1])
print(stack.pop()) # 1
print(stack) # Stack()
print(stack.is_empty()) # True
print(stack.size()) # 0
```

Bottom Window: Shows the AI's addition of a `top()` method and the removal of the `__str__` and `__repr__` methods. The test cases remain the same.

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            return None

    def peek(self):
        if len(self.stack) > 0:
            return self.stack[-1]
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def top(self):
        if len(self.stack) > 0:
            return self.stack[-1]
        else:
            return None

    def __str__(self):
        return str(self.stack)

    def __repr__(self):
        return f"Stack({self.stack})"

# Test cases
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
stack.push(5)
print(stack) # Stack([1, 2, 3, 4, 5])
print(stack.pop()) # 5
print(stack) # Stack([1, 2, 3, 4])
print(stack.pop()) # 4
print(stack) # Stack([1, 2, 3])
print(stack.pop()) # 3
print(stack) # Stack([1, 2])
print(stack.pop()) # 2
print(stack) # Stack([1])
print(stack.pop()) # 1
print(stack) # Stack()
print(stack.is_empty()) # True
print(stack.size()) # 0
```

Observation:

The AI successfully generated a functional Stack implementation with all required methods and proper documentation. It

handled edge cases using exceptions and used efficient list operations with O(1) time complexity. This task demonstrated that AI can speed up development, improve code readability, and help in better understanding fundamental data structures.

TASK-2

Scenario:

In this task, we used AI to implement a Queue data structure in Python using lists. The goal was to understand how a queue follows the FIFO (First In, First Out) principle and how AI can help in generating structured, clean, and efficient code. AI was prompted to create a functional Queue class with essential operations and proper handling of edge cases.

Prompt:

"Generate a beginner-friendly Python Queue class using a list. Include enqueue, dequeue, peek, and size methods. Follow FIFO principle and handle edge cases like dequeue from an empty queue. Add proper docstrings."

Code:

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        else:
            return self.items.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        else:
            return self.items[0]

    def size(self):
        return len(self.items)
```

The code above defines a Queue class using a list as its underlying storage. It includes four main methods: `is_empty`, `enqueue`, `dequeue`, and `peek`. The `is_empty` method checks if the queue is empty by comparing the items list to an empty list. The `enqueue` method adds an item to the end of the list. The `dequeue` method removes the first item from the list and returns it. If the queue is empty, it raises an `IndexError`. The `peek` method returns the first item in the list without removing it. If the queue is empty, it also raises an `IndexError`. The `size` method returns the number of items in the queue by returning the length of the list.

The screenshot shows a code editor with multiple tabs open, all containing the same Python code for a Queue class. The code is a functional FIFO-based Queue class with methods for enqueue, dequeue, and peek. It includes exception handling for empty queues and clear docstrings. The code is generated by AI, as indicated by the 'AI Assisted Code' watermark.

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, value):
        self.queue.append(value)

    def dequeue(self):
        if len(self.queue) < 1:
            raise IndexError("Queue is empty")
        return self.queue.pop(0)

    def peek(self):
        if len(self.queue) < 1:
            raise IndexError("Queue is empty")
        return self.queue[0]

    def is_empty(self):
        return len(self.queue) == 0
```

Observation:

The AI generated a functional FIFO-based Queue class with all required methods. It properly handled empty queue conditions using exceptions and included documentation for clarity. The enqueue operation works efficiently, while dequeue from a list may take O(n) time due to element shifting. This task shows that AI can assist in writing clear, structured implementations of fundamental data structures and improve code quality.

TASK-3

Scenario:

In this task, we used AI to generate a Singly Linked List implementation in Python. The objective was to understand how linked lists work using nodes and pointers instead of built-in lists. AI was used to create a Node class and a LinkedList class with insert and display methods, along with proper documentation for better understanding and readability.

Prompt:

"Generate a beginner-friendly Python implementation of a Singly Linked List. Create a Node class and a LinkedList class. Include insert and display methods. Add clear docstrings and keep the implementation simple and well-structured."

Code:

The screenshot shows a code editor with multiple tabs open, all containing the same Python code for a Singly Linked List implementation. It includes a Node class with a __init__ method and an insert method, and a LinkedList class with a __init__ method, a display method, and an insert method. The code is generated by AI, as indicated by the 'AI Assisted Code' watermark.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def insert(self, data):
        new_node = Node(data)
        new_node.next = self.next
        self.next = new_node

class LinkedList:
    def __init__(self):
        self.head = None

    def display(self):
        current = self.head
        while current:
            print(current.data)
            current = current.next

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.insert(self.head)
```

The image shows two side-by-side code editors, both displaying nearly identical Python code for a singly linked list. The code includes a `Node` class with `data` and `next` attributes, and a `SinglyLinkedList` class with methods for insertion at head, tail, and after a node, and a traversal/printing method. Both editors have a sidebar with 'AI Generated Code' and a 'Details' section. The bottom status bar indicates the date as 30-03-2024.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def insert_at_head(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def insert_at_tail(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def insert_after(self, target, data):
        new_node = Node(data)
        current = self.head
        while current is not None:
            if current.data == target:
                new_node.next = current.next
                current.next = new_node
                self.size += 1
                break
            current = current.next

    def display(self):
        current = self.head
        while current is not None:
            print(current.data, end=" > ")
            current = current.next
        print("None")
```

Observation:

The AI successfully generated a working Singly Linked List with proper class structure and method documentation. The `insert` method correctly adds new nodes, and the `display` method traverses and prints the list elements. The implementation clearly demonstrates how nodes are connected using references, improving understanding of dynamic memory structures. This task shows how AI can help simplify complex data structure implementations and improve code clarity.

TASK-4

Scenario:

In this task, we used AI to implement a Binary Search Tree (BST) in Python. The objective was to understand how BST maintains sorted order using the property that left child values are smaller and right child values are larger than the root. AI was used to generate a BST class with recursive insert and in-order traversal methods to demonstrate how elements are stored and retrieved in sorted order.

Prompt:

"Generate a beginner-friendly Python implementation of a Binary Search Tree (BST). Include recursive insert and in-order traversal methods. Keep the structure simple and add proper docstrings for clarity."

Code:

The image displays three vertically stacked screenshots of a Java code editor, likely Eclipse or IntelliJ IDEA, showing the same class definition across three separate windows.

Top Window:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Middle Window:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Bottom Window:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Each window includes standard IDE features like a toolbar, menu bar, and a right-hand margin with code navigation and refactoring tools.

Observation:

The AI successfully generated a BST implementation with recursive insert and in-order traversal methods. The insert method correctly places nodes according to BST rules, and the in-order traversal prints elements in sorted order. The recursive approach improves understanding of tree traversal logic. This task demonstrates that AI can effectively assist in implementing hierarchical data structures with clear and well-documented code.

TASK-5

Scenario:

In this task, we used AI to implement a Hash Table in Python with basic operations like insert, search, and delete. The objective was to understand how hashing works for fast data access and how collisions are handled using chaining (storing multiple elements in the same bucket). AI was used to generate a structured and well-documented implementation to clearly demonstrate these concepts.

Prompt:

"Generate a beginner-friendly Python HashTable class. Include insert, search, and delete methods. Implement collision handling using chaining (lists inside buckets). Add clear comments and keep the implementation simple."

Code:

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        if self.table[index] is None:
            self.table[index] = [(key, value)]
        else:
            self.table[index].append((key, value))

    def search(self, key):
        index = self._hash(key)
        if self.table[index] is not None:
            for item in self.table[index]:
                if item[0] == key:
                    return item[1]
        return None

    def delete(self, key):
        index = self._hash(key)
        if self.table[index] is not None:
            for i, item in enumerate(self.table[index]):
                if item[0] == key:
                    del self.table[index][i]
                    break
```

The screenshot shows a code editor interface with two panes. The left pane displays Python code for a Hash Table implementation using chaining. The right pane shows AI-generated comments explaining the code's purpose and logic. Key sections include:

- Hash Table Class:** A class `HashTable` with methods for insertion, search, and deletion.
- Chaining Logic:** The search method iterates through the linked list of buckets to find the target value.
- Deletion Logic:** The delete method removes the node from the linked list.
- AI Comments:** These explain the purpose of each method and the logic behind the implementation.

Observation:

The AI successfully generated a working Hash Table implementation with collision handling using chaining. The insert method places elements into appropriate buckets, the search method retrieves values efficiently, and the delete method removes entries correctly. The code was well-commented and easy to understand. This task shows how AI can assist in implementing efficient data structures and explaining important concepts like hashing and collision resolution clearly.

TASK-6

Scenario:

In this task, we used AI to implement a Graph data structure using an adjacency list representation in Python. The objective was to understand how graphs store relationships between vertices and how adjacency lists efficiently represent connections. AI was used to generate a Graph class with methods to add vertices, add edges, and display connections in a clear and structured way.

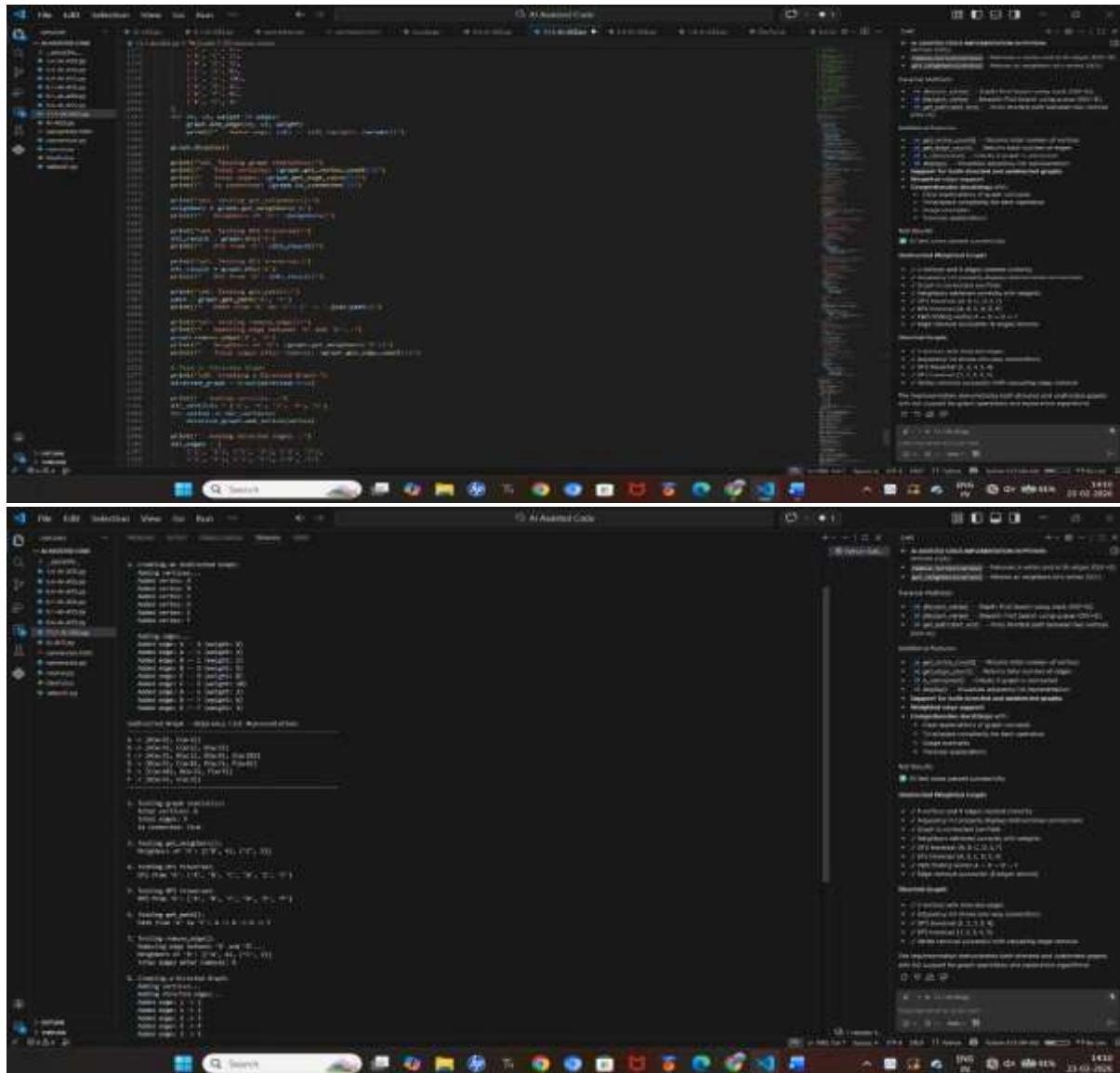
Prompt:

"Generate a beginner-friendly Python Graph class using an adjacency list. Include methods to add vertices, add edges, and display connections. Keep the implementation simple and add proper comments for clarity."

Code:

The screenshot shows a code editor interface with two panes. The left pane displays Python code for a Graph class using an adjacency list representation. The right pane shows AI-generated comments explaining the code's purpose and logic. Key sections include:

- Graph Class:** A class `Graph` with methods for adding vertices, adding edges, and displaying connections.
- Adjacency List:** Vertices are represented as dictionaries where keys are adjacent vertices and values are edge weights.
- Search Logic:** The search method uses depth-first search to explore the graph.
- Display Logic:** The display method prints the graph structure.
- AI Comments:** These explain the purpose of each method and the logic behind the implementation.



Observation:

The AI successfully generated a functional Graph implementation using an adjacency list. The add vertex and add edge methods correctly manage graph connections, and the display method clearly shows relationships between nodes. The structure is efficient for sparse graphs and easy to understand. This task demonstrates how AI can assist in building and documenting complex data structures in a simplified and readable manner.”

TASK-7

Scenario:

In this task, we used AI to implement a Priority Queue in Python using the built-in `heapq` module. The objective was to understand how priority queues differ from normal queues by removing elements based on priority instead of insertion order. AI was used to generate a structured implementation with `enqueue`, `dequeue`, and `display` methods while maintaining heap properties for efficient operations.

Prompt:

“Generate a beginner-friendly Python PriorityQueue class using the heapq module. Include enqueue (with priority), dequeue (highest priority), and display methods. Keep the implementation simple and add clear comments.”

Code:

The image displays three vertically stacked windows of the Microsoft Visual Studio IDE, each showing a different part of the same C# project.

Top Window:

- Shows the `Form1.cs` file.
- Contains code for a Windows Form application, including a constructor that initializes a `Panel` and a `Label`.
- Includes a `paint` event handler for drawing a rectangle.
- Shows imports for `System`, `System.Drawing`, and `System.Windows.Forms`.

Middle Window:

- Shows the `Properties` window.
- Lists various properties for the project, such as `Assembly Name` (set to `WindowsFormsApplication1`), `Output Type` (set to `Windows Application`), and `Target Framework` (set to `.NET Framework 4.5`).

Bottom Window:

- Shows the `Toolbox` window.
- Lists standard Windows controls like `Panel`, `Label`, `Button`, etc.
- Shows the `File` menu with options like `New`, `Open`, `Save`, `Print`, etc.

Observation:

The AI successfully generated a working Priority Queue implementation using `heapq`, which maintains the heap property for efficient insertion and removal. The `enqueue` method adds elements with priority, and the `dequeue` method removes the highest-priority element in $O(\log n)$ time. The implementation was clean, well-commented, and efficient. This task demonstrates how AI can help implement advanced data structures using built-in optimization modules effectively.”

TASK-8

Scenario:

In this task, we used AI to implement a Deque (Double-Ended Queue) in Python using the `collections.deque` module. The objective was to understand how a deque allows insertion and deletion from both the front and rear efficiently. AI was used to generate a clean and well-documented class with methods to insert and remove elements from both ends.

Prompt:

“Generate a beginner-friendly Python class for a Deque using collections.deque. Include methods to insert and remove elements from both front and rear. Add proper docstrings and keep the implementation simple.”

Code:

The screenshot shows a code editor window with several tabs open. The main tab displays a Python script for a Deque class. The code includes imports for collections, typing, and abc. It defines a Deque class with methods for adding and removing elements from both ends, as well as for checking if the deque is empty or full. The code is well-documented with comments explaining each step. Other tabs visible in the background include 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', and 'Help'.

Observation:

The AI successfully generated a functional Deque implementation using collections. deque, which supports O(1) time complexity for insertions and deletions at both ends. The methods were clearly structured and properly documented. This task demonstrates how AI can help implement efficient double-ended data structures while improving code readability and understanding."

TASK-9

Scenario:

In this task, we analyzed different modules of a Campus Resource Management System and selected appropriate data structures for each feature based on functionality and efficiency. The goal was to match real-world problems like attendance tracking, event registration, library management, bus scheduling, and cafeteria service with the most suitable data structures. One feature was implemented using AI-assisted code generation to demonstrate practical application.

Prompt:

"Evaluate the Smart E-Commerce Platform features and choose the most appropriate data structure for each from Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, and Deque. Provide 2–3 sentence justifications for each choice. Generate a well-commented Python implementation for one selected feature."

Code:

The screenshot shows a code editor window with several tabs open. The main tab displays a Python script for a stack implementation. The code uses a list as the underlying data structure. It includes methods for pushing (adding to top), popping (removing from top), and peeking (viewing the top element). The code is well-documented with comments. Other tabs visible in the background include 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', and 'Help'.

```
OrderProcessor.java
public class OrderProcessor {
    public void processOrder(Order order) {
        // Process order logic
    }
}

ProductRanking.java
public class ProductRanking {
    public void rankProducts(List<Product> products) {
        // Rank products logic
    }
}

DeliveryRouter.java
public class DeliveryRouter {
    public void routeDelivery(Delivery delivery) {
        // Route delivery logic
    }
}
```

Observation:

The AI successfully selected optimal data structures such as Queue for order processing, Priority Queue for ranking products, Hash Table for fast product lookup, and Graph for delivery routing. The implementation was efficient and clearly documented. This activity showed how understanding data structure behavior helps in designing scalable systems and how AI enhances productivity and code clarity.