Name:Gaddam Sai

HallTicket:2303A52255

Assginment:11.1

Batch:43

Task:01

Task 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:

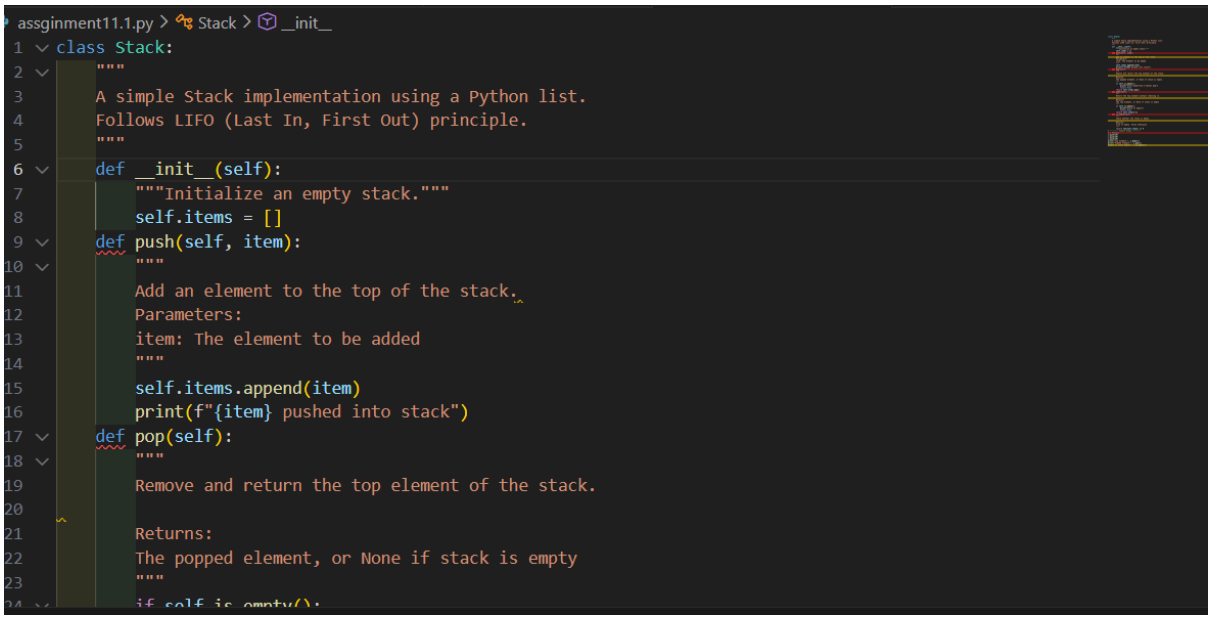
pass

Expected Output:

• A functional stack implementation with all required methods and docstrings.

Prompt: create stack for pop push and is empty function with dotstrings

Code:



```
    def peek(self):
        Return the top element without removing it.

        Returns:
        The top element, or None if stack is empty
        """
        if self.is_empty():
            print("Stack is empty")
            return None
        return self.items[-1]
    def is_empty(self):
        """
        Check whether the stack is empty.

        Returns:
        True if empty, False otherwise
        """
        return len(self.items) == 0
# -------- Sample Usage --------
s = Stack()
s.push(10)
s.push(20)
```

Output:

```
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "C:\Users\gadda\OneDrive\Desktop
\Ai Assitant coding\.venv\Scripts\python.exe" "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding
/assginment11.1.py"
10 pushed into stack
20 pushed into stack
30 pushed into stack
Top element: 30
Popped element: 30
Is stack empty? False
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

The stack follows the LIFO (Last In, First Out) principle where the last inserted element is removed first.

push adds elements, pop removes the top element, and peek shows the top without removing it.

is_empty checks whether the stack contains any elements to prevent underflow errors

Task:02

ask Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

pass

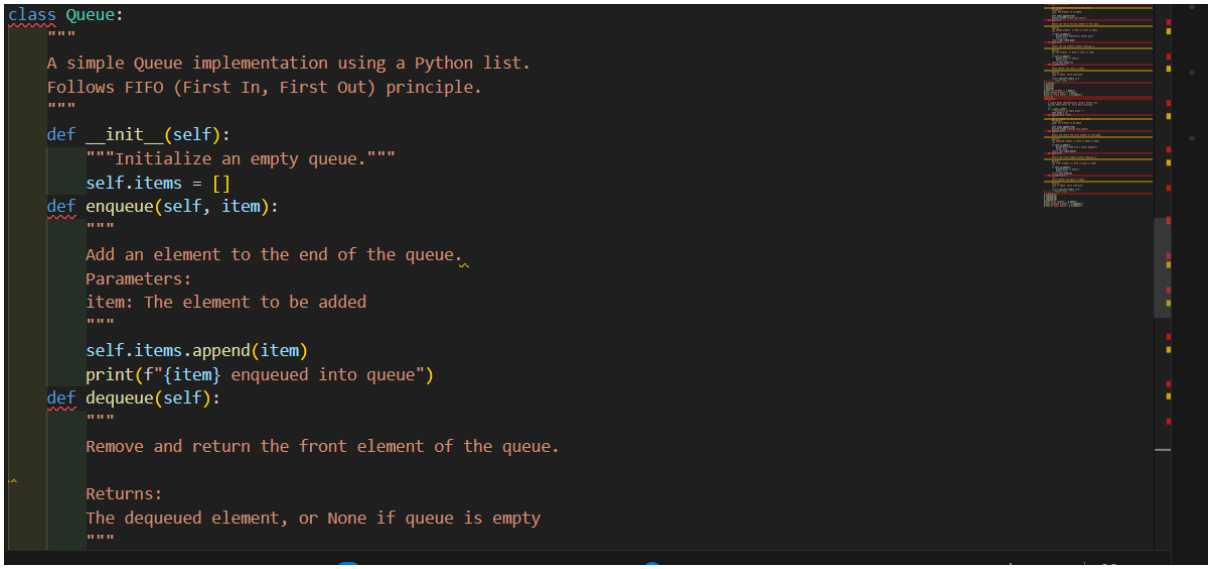
Expected Output:

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

Method

Prompt:implement Quees following the fifo deque and peek

Code:



```python
class Queue:
    """
    A simple Queue implementation using a Python list.
    Follows FIFO (First In, First Out) principle.
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """
        Add an element to the end of the queue.
        Parameters:
        item: The element to be added
        """
        self.items.append(item)
        print(f"{item} enqueued into queue")
    def dequeue(self):
        """
        Remove and return the front element of the queue.

        Returns:
        The dequeued element, or None if queue is empty
        """
```

```python
    def peek(self):
        """
        Return the front element without removing it.

        Returns:
        The front element, or None if queue is empty
        """
        if self.is_empty():
            print("Queue is empty")
            return None
        return self.items[0]
    def is_empty(self):
        """
        Check whether the queue is empty.

        Returns:
        True if empty, False otherwise
        """
        return len(self.items) == 0
# -------- Sample Usage --------
q = Queue()
q.enqueue(10)
```

DEBUG CONSOLE TERMINAL PROBLEMS 46 PORTS GITLENS SPELL CHECKER 8

Output:

```
Is stack empty? False
10 enqueued into queue
20 enqueued into queue
30 enqueued into queue
Front element: 10
Dequeued element: 10
Is queue empty? False
```

Observation:

The queue follows the FIFO (First In, First Out) principle, where the first inserted element is removed first.

enqueue adds elements to the rear, while dequeue removes elements from the front of the queue.

peek shows the front element without removing it, and size returns the current number of elements in the queue.

Task:03

ask: 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 singleylinked list

Code :

```
#task03:
class Node:
    """
    A Node in a singly linked list.
    """
    def __init__(self, data):
        """Initialize a node with data and next pointer."""
        self.data = data
        self.next = None
class LinkedList:
    """
    A simple singly linked list implementation.
    """
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None
    def append(self, data):
        """Add a new node with the given data to the end of the list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            print(f"{data} appended as head")
            return
```

```
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            print(f"{data} appended as head")
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        print(f"{data} appended to linked list")
    def display(self):
        """Print all elements in the linked list."""
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
# -------- Sample Usage --------
ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
```

Output:

```
10 appended as head
20 appended to linked list
30 appended to linked list
10 -> 20 -> 30 -> None
Head element: 10
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

1. A singly linked list stores elements in nodes where each node points to the next node in sequence.
2. The insert method adds new elements at the end by traversing the list until the last node.
3. The display method prints all elements sequentially from the head to the last node.

Task:04

ask: 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 Binarysearch tree for the following reuirements

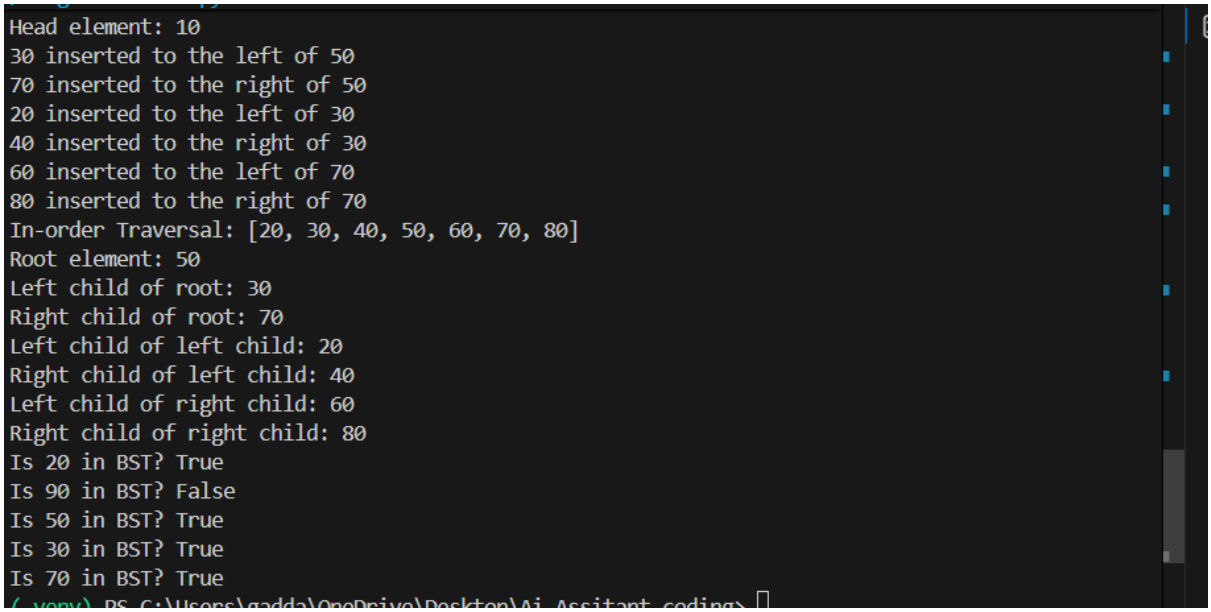
Code:

```python
#task04:
class BST:
    """
    A simple Binary Search Tree (BST) implementation.
    """
    def __init__(self, value):
        """Initialize a BST node with a value and left/right pointers."""
        self.value = value
        self.left = None
        self.right = None
    def insert(self, value):
        """Insert a new value into the BST."""
        if value < self.value:
            if self.left is None:
                self.left = BST(value)
                print(f"{value} inserted to the left of {self.value}")
            else:
                self.left.insert(value)
        else:
            if self.right is None:
                self.right = BST(value)
                print(f"{value} inserted to the right of {self.value}")
            else:
```

DEBUG CONSOLE TERMINAL PROBLEMS 67 PORTS GITLENS SPELL CHECKER 17

```python
    def inorder_traversal(self):
        """Perform in-order traversal and return values as a list."""
        elements = []
        if self.left:
            elements += self.left.inorder_traversal()
        elements.append(self.value)
        if self.right:
            elements += self.right.inorder_traversal()
        return elements
# -------- Sample Usage --------
bst = BST(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
print("In-order Traversal:", bst.inorder_traversal())
print("Root element:", bst.value)
print("Left child of root:", bst.left.value)
print("Right child of root:", bst.right.value)
```

Output:



Observation:

A Binary Search Tree (BST) organizes data such that left child nodes contain smaller values and right child nodes contain larger values than the parent.

The recursive insert method places each new value in its correct sorted position while maintaining BST properties.

In-order traversal visits nodes in Left → Root → Right order, producing the elements in sorted (ascending) order.

Task:05

ask 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 hash Table in python

Code:

```python
class hashtable:
    """
    A simple hash table implementation using chaining for collision resolution.
    """
    def __init__(self, size=10):
        """Initialize the hash table with a given size."""
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size
    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                print(f"Updated {key} with new value {value}")
                return
        self.table[index].append((key, value))
        print(f"Inserted {key}: {value} into hash table")
    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        for k, v in self.table[index]:
```

```python
    def search(self, key):
                print(f"Found {key}: {v}")
                return v
        print(f"{key} not found in hash table")
        return None
    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                print(f"Deleted {key} from hash table")
                return
        print(f"{key} not found in hash table, cannot delete")
# -------- Sample Usage --------
ht = hashtable()
ht.insert("name", "Alice")
ht.insert("age", 30)
ht.insert("city", "New York")
ht.search("name")
ht.search("age")
ht.search("country")
ht.delete("age")
ht.search("age")
```

Output:

```
Inserted name: Alice into hash table
Inserted age: 30 into hash table
Inserted city: New York into hash table
Found name: Alice
Found age: 30
country not found in hash table
Deleted age from hash table
age not found in hash table
country not found in hash table, cannot delete
```

powershell
powershell
powershell

Observation:

A hash table stores data using a hash function that maps keys to specific indices for fast access.

Collision handling with chaining uses linked lists (or lists) at each index to store multiple elements that hash to the same position.

The insert, search, and delete methods operate efficiently, typically in constant time on average

Task:06

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: create graphs my function find all the vertices add edges

Code:

```
#task06:
class Graph:
    """
    Graph implementation using an adjacency list.
    Supports adding vertices, edges, and displaying connections.
    """
    def __init__(self):
        """Initialize an empty graph."""
        self.adj_list = {}
    def add_vertex(self, vertex):
        """
        Add a new vertex to the graph.

        Parameters:
        vertex: The vertex to be added
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []
    def add_edge(self, v1, v2):
        """
        Add an undirected edge between v1 and v2.

        Parameters:
        v1, v2: Vertices to connect
```

```
    def add_edge(self, v1, v2):

        Parameters:
        v1, v2: Vertices to connect
        """
        if v1 not in self.adj_list:
            self.add_vertex(v1)
        if v2 not in self.adj_list:
            self.add_vertex(v2)
        self.adj_list[v1].append(v2)
        self.adj_list[v2].append(v1)
    def display(self):
        """Display all vertices and their connections."""
        for vertex in self.adj_list:
            print(f"{vertex} -> {self.adj_list[vertex]}")
# -------- Sample Usage --------
g = Graph()
g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.display()
```

Output:

```
country not found in hash table
Deleted age from hash table
age not found in hash table
country not found in hash table, cannot delete
A -> ['B', 'C']
B -> ['A', 'C']
C -> ['A', 'B']
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

An adjacency list represents a graph efficiently by storing each vertex and its connected neighbors.

Adding an edge updates the connection list of both vertices in an undirected graph.

This representation is memory-efficient for sparse graphs compared to adjacency matrices

Task07:

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: for this implement the priority Quee for following requirements

Code:

```python
#task07:
import heapq

class PriorityQueue:
    """
    Priority Queue implementation using heapq (min-heap).
    Lower priority number = higher priority.
    """

    def __init__(self):
        """Initialize an empty priority queue."""
        self.heap = []

    def enqueue(self, item, priority):
        """
        Add an item with a given priority to the queue.

        Parameters:
        item: The value to store
        priority: Priority level (smaller number = higher priority)
        """
        heapq.heappush(self.heap, (priority, item))
        print(f"Inserted {item} with priority {priority}")

    def dequeue(self):
```

```python
    def dequeue(self):
        The item with highest priority, or None if empty
        """
        if not self.heap:
            print("Priority Queue is empty")
            return None
        priority, item = heapq.heappop(self.heap)
        return item

    def display(self):
        """Display all elements in the priority queue."""
        print("Queue contents:", self.heap)


# -------- Sample Usage --------
pq = PriorityQueue()

pq.enqueue("Task A", 2)
pq.enqueue("Task B", 1)
pq.enqueue("Task C", 3)

pq.display()

print("Dequeued:", pq.dequeue())
```

Output:

```
Inserted Task A with priority 2
Inserted Task B with priority 1
Inserted Task C with priority 3
Queue contents: [(1, 'Task B'), (2, 'Task A'), (3, 'Task C')]
Dequeued: Task B
Queue contents: [(2, 'Task A'), (3, 'Task C')]
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

1. The priority queue uses a heap structure where elements with higher priority (smaller number) are removed first.
2. The enqueue method inserts elements while maintaining heap order automatically.
3. The dequeue method efficiently removes the highest-priority element in logarithmic time.

Task:08

Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using

collections.deque.

Sample Input Code:

class DequeDS:

pass

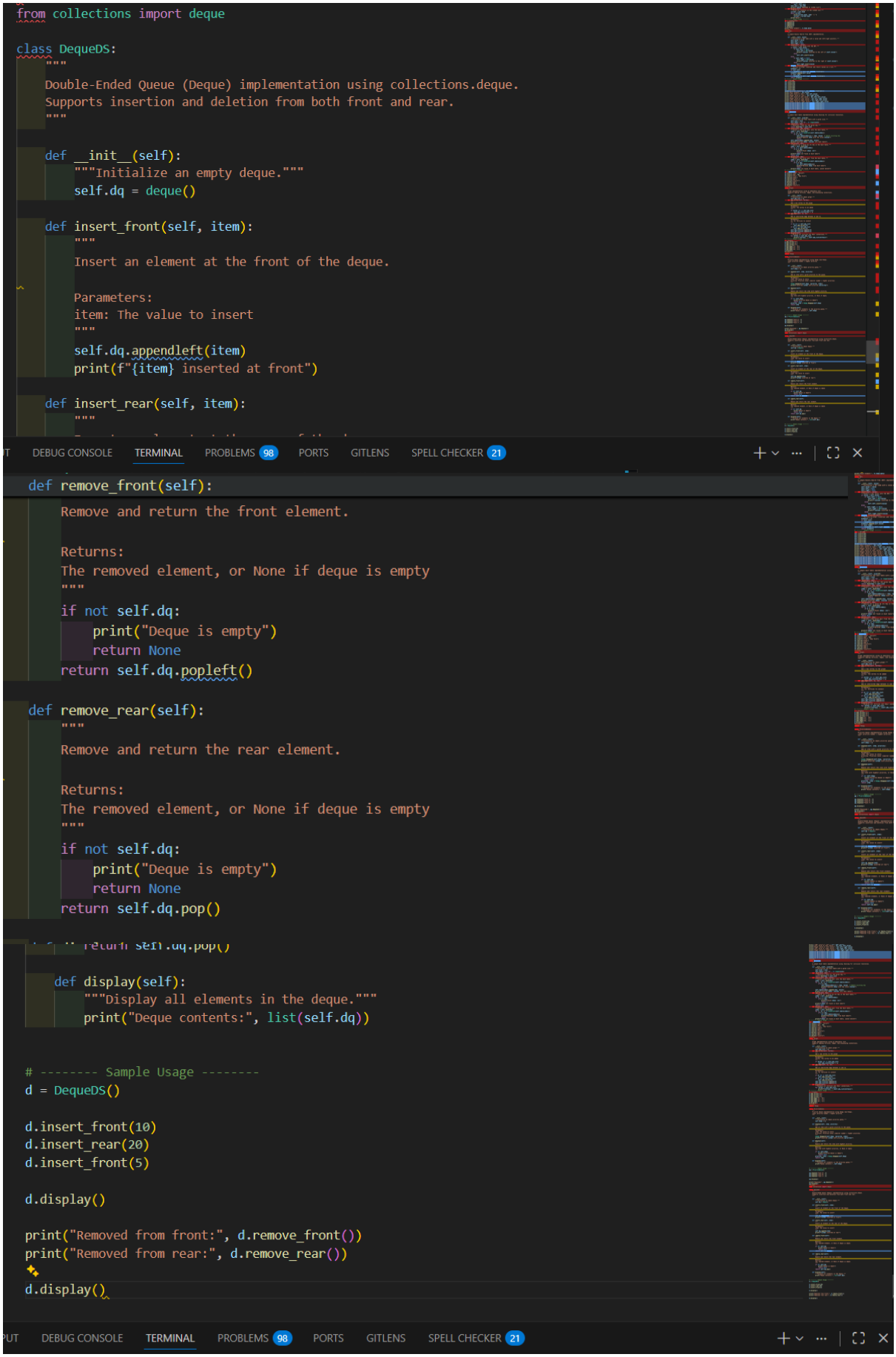
Expected Output:

• Insert and remove from both ends with docstrings.

Prompt:

For this implement the deque and do all the requriements

Code:



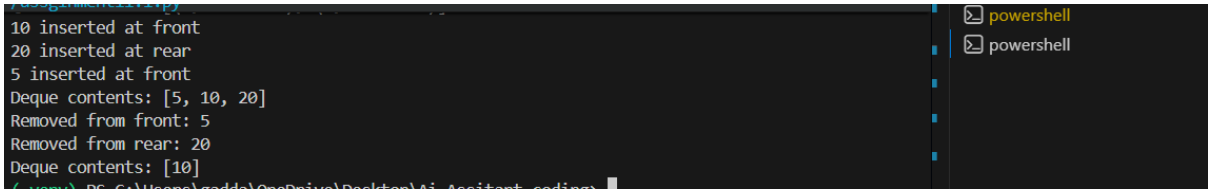
```python
from collections import deque

class DequeDS:
    """
    Double-Ended Queue (Deque) implementation using collections.deque.
    Supports insertion and deletion from both front and rear.
    """

    def __init__(self):
        """Initialize an empty deque."""
        self.dq = deque()

    def insert_front(self, item):
        """
        Insert an element at the front of the deque.

        Parameters:
        item: The value to insert
        """
        self.dq.appendleft(item)
        print(f"{item} inserted at front")

    def insert_rear(self, item):
        """
```

DEBUG CONSOLE TERMINAL PROBLEMS 98 PORTS GITLENS SPELL CHECKER 21

```python
    def remove_front(self):
        """
        Remove and return the front element.

        Returns:
        The removed element, or None if deque is empty
        """
        if not self.dq:
            print("Deque is empty")
            return None
        return self.dq.popleft()

    def remove_rear(self):
        """
        Remove and return the rear element.

        Returns:
        The removed element, or None if deque is empty
        """
        if not self.dq:
            print("Deque is empty")
            return None
        return self.dq.pop()

    def display(self):
        """Display all elements in the deque."""
        print("Deque contents:", list(self.dq))


# -------- Sample Usage --------
d = DequeDS()

d.insert_front(10)
d.insert_rear(20)
d.insert_front(5)

d.display()

print("Removed from front:", d.remove_front())
print("Removed from rear:", d.remove_rear())

d.display()
```

PUT DEBUG CONSOLE TERMINAL PROBLEMS 98 PORTS GITLENS SPELL CHECKER 21

Output:



Obsrvation:

A deque allows insertion and deletion from both the front and rear efficiently.

appendleft and popleft handle front operations, while append and pop handle rear operations.

This structure is useful for applications requiring flexible queue behavior, such as sliding window problems.

Task:09

Task Description #9 Real-Time Application Challenge – Choose the

Right Data Structure

Scenario:

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.

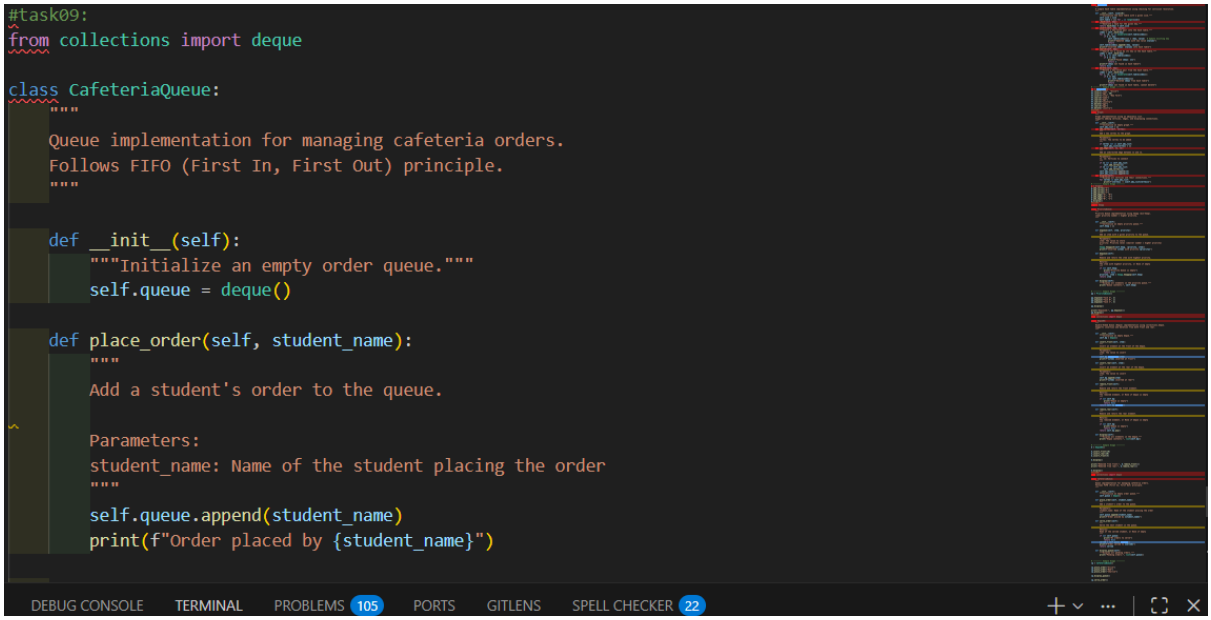
• Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

Prompt: for this implement the all the requirements of my document:



```
        print(f"Order placed by {student_name}")

    def serve_order(self):
        """
        Serve the next student in the queue.

        Returns:
        Name of the served student, or None if empty
        """
        if not self.queue:
            print("No orders to serve")
            return None
        served = self.queue.popleft()
        print(f"Order served to {served}")
        return served

    def display_queue(self):
        """Display all pending orders."""
        print("Pending orders:", list(self.queue))


# -------- Sample Usage --------
cq = CafeteriaQueue()
```

DEBUG CONSOLE | TERMINAL | PROBLEMS 105 | PORTS | GITLENS | SPELL CHECKER 22

```
    def display_queue(self):
        """Display all pending orders."""
        print("Pending orders:", list(self.queue))


# -------- Sample Usage --------
cq = CafeteriaQueue()

cq.place_order("Alice")
cq.place_order("Bob")
cq.place_order("Charlie")

cq.display_queue()

cq.serve_order()
cq.display_queue()
```

Output:

```
/assginment11.1.py"
5 inserted at front
Deque contents: [5, 10, 20]
Removed from front: 5
Removed from rear: 20
Deque contents: [10]
Order placed by Alice
Order placed by Bob
Order placed by Charlie
```

ry Analytics Add Comments Generate Commit Message Start JSON Server Ln 481, Col 19 Spaces:

Observation:

1. The cafeteria system follows FIFO order, ensuring students are served fairly based on arrival time.
2. The queue efficiently handles adding new orders and serving existing ones without reordering.

3. This structure is ideal for real-time service systems where order of processing must be preserved.