AI ASSISTED CODING

HALL TICKET :2403A51266

B:12

ASSIGNMENT :11.4

QUESTION:

Task 1: Implementing a Stack (LIFO)
• Task: Use AI to help implement a Stack class in Python with the following operations: push(), pop(), peek(), and is_empty().
• Instructions:
○ Ask AI to generate code skeleton with docstrings.
○ Test stack operations using sample data.
○ Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).
• Expected Output:
○ A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts

Prompt:

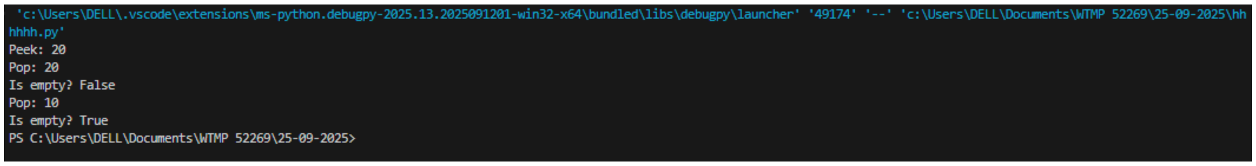
"Implement a Stack class in Python with methods push(), pop(), peek(), and is_empty(). Write Google-style docstrings and inline comments for tricky parts. Provide a simple test case that demonstrates pushing, popping, and peeking. Then, suggest an optimization or alternative implementation using collections.deque."

CODE:

```
class Stack:


    def __init__(self):
        self._items = []

    def push(self, item):
```

```python
        self._items.append(item)

    def pop(self):

        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self._items.pop()

    def peek(self):

        if self.is_empty():
            return None
        return self._items[-1]

    def is_empty(self):
        """Check if the stack is empty."""
        return len(self._items) == 0


# Sample usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(10)
    stack.push(20)
    print("Peek:", stack.peek())
    print("Pop:", stack.pop())
    print("Is empty?", stack.is_empty())
    print("Pop:", stack.pop())
    print("Is empty?", stack.is_empty())
```

OUTPUT:



```
 'c:\Users\DELL\.vscode\extensions\ms-python.debugpy-2025.13.2025091201-win32-x64\bundled\libs\debugpy\launcher' '49174' '--' 'c:\Users\DELL\Documents\WTMP 52269\25-09-2025\hh
hhhh.py'
Peek: 20
Pop: 20
Is empty? False
Pop: 10
Is empty? True
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>
```

Task 2: Queue Implementation with Performance Review

• Task: Implement a Queue with enqueue(), dequeue(), and is_empty() methods.

• Instructions:

○ First, implement using Python lists.

○ Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).

• Expected Output:

○ Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison

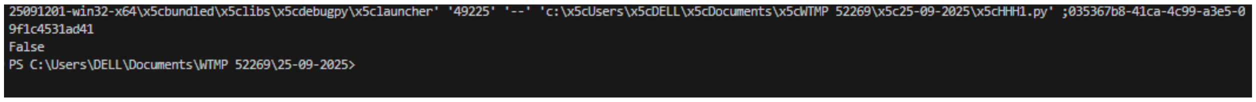
Prompt:

"Implement a Queue class in Python with methods enqueue(), dequeue(), and is_empty(). First, implement it using Python lists. Then explain the performance drawbacks. Next, implement an optimized version using collections.deque and provide an AI-generated performance comparison. Include test cases for both versions."

CODE:2.1

```python
class QueueList:


    def __init__(self):
        self._items = []

    def enqueue(self, item):

        self._items.append(item)

    def dequeue(self):

        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self._items.pop(0)

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self._items) == 0


if __name__ == "__main__":
    q = QueueList()
    q.enqueue(1)
    q.enqueue(2)
    print(q.dequeue())
    print(q.is_empty())
```

OUTPUT:

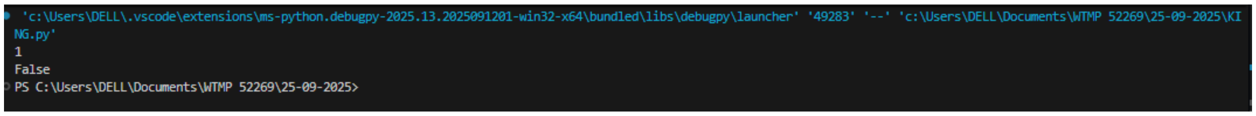


```
25091201-win32-x64\x5cbundled\x5clibs\x5cdebugpy\x5claucher' '49225' '--' 'c:\x5cUsers\x5cDELL\x5cDocuments\x5cWTMP 52269\x5c25-09-2025\x5cHHH1.py' ;035367b8-41ca-4c99-a3e5-0
9f1c4531ad41
False
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>
```

CODE 2.2:

```python
from collections import deque

class QueueDeque:


    def __init__(self):
        self._items = deque()

    def enqueue(self, item):

        self._items.append(item)

    def dequeue(self):

        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self._items.popleft()

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self._items) == 0


if __name__ == "__main__":
    q = QueueDeque()
    q.enqueue(1)
    q.enqueue(2)
    print(q.dequeue())  # Output: 1
    print(q.is_empty())  # Output: False
```

OUTPUT:



```
 'c:\Users\DELL\.vscode\extensions\ms-python.debugpy-2025.13.2025091201-win32-x64\bundled\libs\debugpy\launcher' '49283' '--' 'c:\Users\DELL\Documents\WTMP 52269\25-09-2025\KI
NG.py'
1
False
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>
```

Task 3: Singly Linked List with Traversal
• Task: Implement a Singly Linked List with operations:
insert_at_end(), delete_value(), and traverse().
• Instructions:
○ Start with a simple class-based implementation (Node,
LinkedList).
○ Use AI to generate inline comments explaining pointer updates
(which are non-trivial).
○ Ask AI to suggest test cases to validate all operations.
• Expected Output:
○ A functional linked list implementation with clear comments
explaining the logic of insertions and deletions

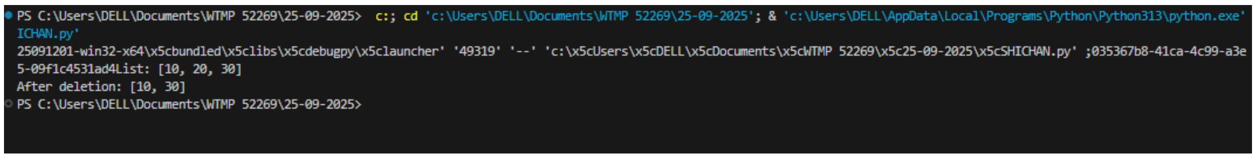
Prompt:

"Implement a Singly Linked List in Python with a Node and LinkedList class. Add methods insert_at_end(), delete_value(), and traverse(). Use inline comments to clearly explain pointer updates (since they are tricky). Also, suggest test cases to validate insertions, deletions (head, middle, last node), and traversal."

CODE:

```python
class Node:


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


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

    def insert_at_end(self, data):

        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 delete_value(self, value):
        current = self.head
        prev = None


        while current:
            if current.data == value:
                if prev is None:

                    self.head = current.next
                else:

                    prev.next = current.next
                return True
            prev = current
            current = current.next
        return False

    def traverse(self):

        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        return elements


if __name__ == "__main__":
    ll = LinkedList()
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("List:", ll.traverse())
    ll.delete_value(20)
    print("After deletion:", ll.traverse())
```

OUTPUT:



```
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>  c:; cd 'c:\Users\DELL\Documents\WTMP 52269\25-09-2025'; & 'c:\Users\DELL\AppData\Local\Programs\Python\Python313\python.exe'
ICHAN.py'
25091201-win32-x64\x5cbundled\x5clibs\x5cdebugpy\x5claucher' '49319' '--' 'c:\x5cUsers\x5cDELL\x5cDocuments\x5cWTMP 52269\x5c25-09-2025\x5cSHICHAN.py' ;035367b8-41ca-4c99-a3e
5-09f1c4531ad4List: [10, 20, 30]
After deletion: [10, 30]
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>
```

Task 4: Binary Search Tree (BST)
• Task: Implement a Binary Search Tree with methods for insert(), search(), and inorder_traversal().
• Instructions:
○ Provide AI with a partially written Node and BST class.

○ Ask AI to complete missing methods and add docstrings.
○ Test with a list of integers and compare outputs of search() for present vs absent elements.
• Expected Output:
○ A BST class with clean implementation, meaningful docstrings, and correct traversal output

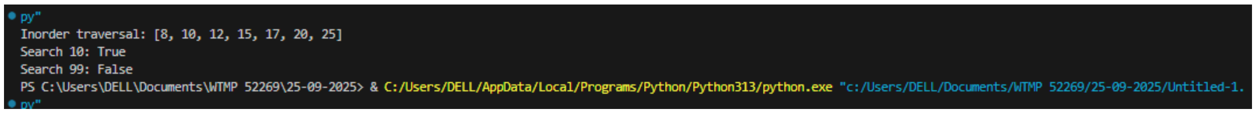
Prompt:

"Complete a Binary Search Tree implementation in Python. Provide insert(), search(), and inorder_traversal() methods inside a BST class, with a supporting BSTNode class. Add Google-style docstrings. Show tests using a list of integers, and demonstrate searching for both present and absent elements."

CODE:

```python
class BSTNode:



    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinarySearchTree:

    def __init__(self):
        self.root = None

    def insert(self, value):

        if self.root is None:
            self.root = BSTNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = BSTNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = BSTNode(value)
            else:
                self._insert_recursive(node.right, value)

    def search(self, value):

        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        if node is None:
            return False
        if node.value == value:
            return True
        elif value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder_traversal(self):
        elements = []
        self._inorder_recursive(self.root, elements)
        return elements

    def _inorder_recursive(self, node, elements):
        if node:
            self._inorder_recursive(node.left, elements)
            elements.append(node.value)
            self._inorder_recursive(node.right, elements)


if __name__ == "__main__":
    bst = BinarySearchTree()
    for val in [15, 10, 20, 8, 12, 17, 25]:
        bst.insert(val)
    print("Inorder traversal:", bst.inorder_traversal())
    print("Search 10:", bst.search(10))
    print("Search 99:", bst.search(99))
```

OUTPUT:



```
● py"
  Inorder traversal: [8, 10, 12, 15, 17, 20, 25]
  Search 10: True
  Search 99: False
  PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025> & C:/Users/DELL/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/DELL/Documents/WTMP 52269/25-09-2025/Untitled-1.
● py"
```

Task 5: Graph Representation and BFS/DFS Traversal
• Task: Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().
• Instructions:
○ Start with an adjacency list dictionary.
○ Ask AI to generate BFS and DFS implementations with inline comments.
○ Compare recursive vs iterative DFS if suggested by AI.
• Expected Output:
○ A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

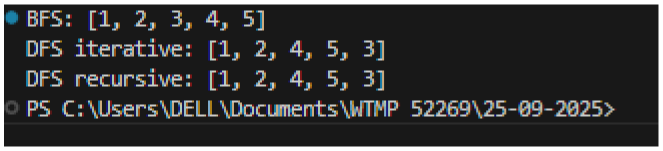
Prompt:

"Implement a Graph in Python using an adjacency list (dictionary). Add methods bfs(start), dfs(start) (iterative), and dfs_recursive(start). Include inline comments explaining the traversal logic step by step. Also, compare recursive vs iterative DFS. Provide a test graph with at least 5–6 nodes and show traversal outputs."

CODE:

```python
from collections import deque

class Graph:


    def __init__(self):
        self.adj_list = {}

    def add_edge(self, u, v):

        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def bfs(self, start):

        visited = set()
        queue = deque([start])
        visited.add(start)
        result = []

        while queue:
            vertex = queue.popleft()
            result.append(vertex)
            # Visit all adjacent nodes
            for neighbor in self.adj_list.get(vertex, []):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        return result

    def dfs(self, start):
        visited = set()
        stack = [start]
        result = []

        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)

                neighbors = self.adj_list.get(vertex, [])
                for neighbor in reversed(neighbors):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return result

    def dfs_recursive(self, start):

        visited = set()
        result = []

        def dfs_helper(v):
            visited.add(v)
            result.append(v)
            for neighbor in self.adj_list.get(v, []):
                if neighbor not in visited:
                    dfs_helper(neighbor)

        dfs_helper(start)
        return result


if __name__ == "__main__":
    graph = Graph()
    edges = [(1, 2), (1, 3), (2, 4), (3, 5), (4, 5)]
    for u, v in edges:
        graph.add_edge(u, v)

    print("BFS:", graph.bfs(1))
    print("DFS iterative:", graph.dfs(1))
    print("DFS recursive:", graph.dfs_recursive(1))
```

OUTPUT:



```
BFS: [1, 2, 3, 4, 5]
DFS iterative: [1, 2, 4, 5, 3]
DFS recursive: [1, 2, 4, 5, 3]
PS C:\Users\DELL\Documents\WTMP 52269\25-09-2025>
```