Assignment 11

TASK 1 : Use AI to help implement a Stack class in Python with the  
following operations: push(), pop(), peek(), and is\_empty()

CODE :

class Stack:

    def \_\_init\_\_(self):

        self.items = []

    def push(self, item):

        """Adds an item to the top of the stack."""

        self.items.append(item)

    def pop(self):

        """Removes and returns the item from the top of the stack."""

        if not self.is\_empty():

            return self.items.pop()

        else:

            return "Stack is empty"

    def peek(self):

        """Returns the item at the top of the stack without removing it."""

        if not self.is\_empty():

            return self.items[-1]

        else:

            return "Stack is empty"

    def is\_empty(self):

        """Checks if the stack is empty."""

        return len(self.items) == 0

# Example usage:

my\_stack = Stack()

my\_stack.push(10)

my\_stack.push(20)

my\_stack.push(30)

print("Stack after pushes:", my\_stack.items)

print("Top element (peek):", my\_stack.peek())

print("Popped element:", my\_stack.pop())

print("Stack after pop:", my\_stack.items)

print("Is stack empty?", my\_stack.is\_empty())

print("Popped element from empty stack:", my\_stack.pop())

OUTPUT:

Stack after pushes: [10, 20, 30]

Top element (peek): 30

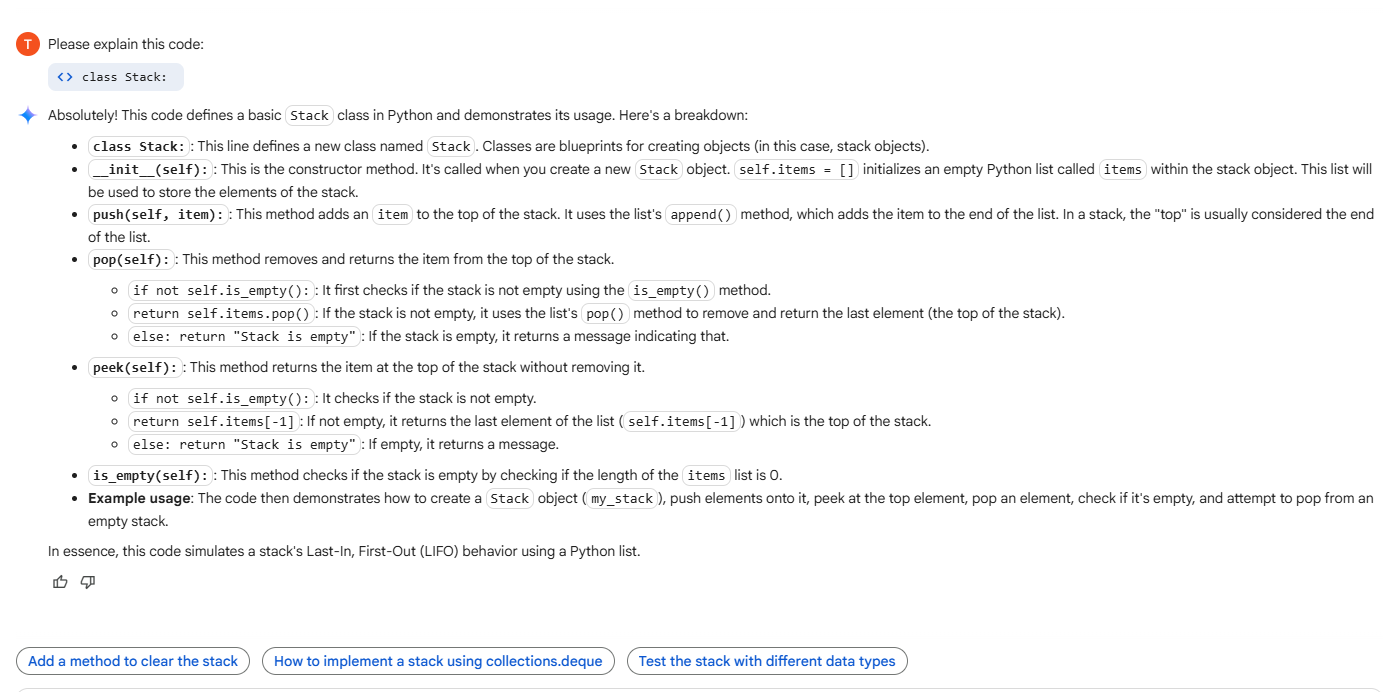
Popped element: 30

Stack after pop: [10, 20]

Is stack empty? False

Popped element from empty stack: 20’

EXPLAINATION:



Task 2: Implement a Queue with enqueue(), dequeue(), and is\_empty()  
methods

CODE:

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def enqueue(self, item):

        """Adds an item to the rear of the queue."""

        self.items.append(item)

    def dequeue(self):

        """Removes and returns the item from the front of the queue."""

        if not self.is\_empty():

            return self.items.pop(0) # Remove the first item

        else:

            return "Queue is empty"

    def is\_empty(self):

        """Checks if the queue is empty."""

        return len(self.items) == 0

# Example usage:

my\_queue = Queue()

my\_queue.enqueue(10)

my\_queue.enqueue(20)

my\_queue.enqueue(30)

print("Queue after enqueues:", my\_queue.items)

print("Dequeued element:", my\_queue.dequeue())

print("Queue after dequeue:", my\_queue.items)

print("Is queue empty?", my\_queue.is\_empty())

print("Dequeued element from empty queue:", my\_queue.dequeue())

OUTPUT:

Queue after enqueues: [10, 20, 30]

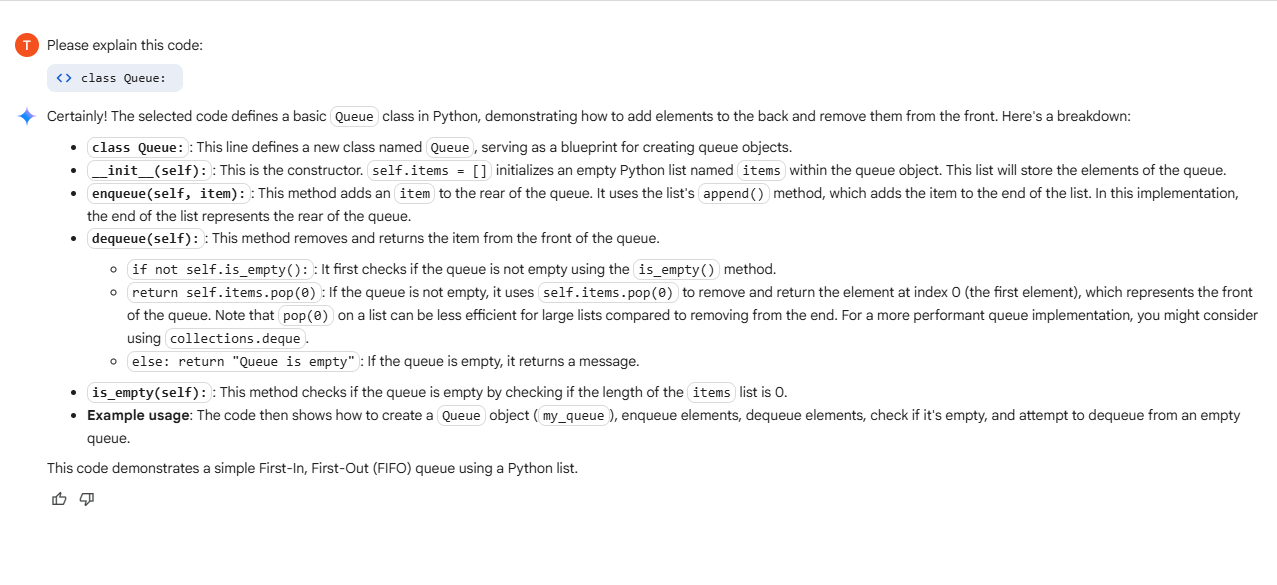
Dequeued element: 10

Queue after dequeue: [20, 30]

Is queue empty? False

Dequeued element from empty queue: 20

EXPLAINATION:



TASK 3 : Implement a Singly Linked List with operations:  
insert\_at\_end(), delete\_value(), and traverse().

CODE :

class Node:

    """Represents a node in a singly linked list."""

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None # Pointer to the next node

class SinglyLinkedList:

    """Represents a singly linked list."""

    def \_\_init\_\_(self):

        self.head = None # Head of the list

    def insert\_at\_end(self, data):

        """Inserts a new node with the given data at the end of the list."""

        new\_node = Node(data)

        if self.head is None:

            self.head = new\_node

            return

        last\_node = self.head

        while last\_node.next:

            last\_node = last\_node.next

        last\_node.next = new\_node

    def delete\_value(self, value):

        """Deletes the first node with the given value from the list."""

        current\_node = self.head

        # If the head node itself holds the value to be deleted

        if current\_node is not None and current\_node.data == value:

            self.head = current\_node.next

            current\_node = None # Dereference the old head

            return

        # Search for the value to be deleted, keeping track of the previous node

        prev\_node = None

        while current\_node is not None and current\_node.data != value:

            prev\_node = current\_node

            current\_node = current\_node.next

        # If the value was not found in the list

        if current\_node is None:

            print(f"Value {value} not found in the list.")

            return

        # Unlink the node from the linked list

        prev\_node.next = current\_node.next

        # Dereference the node to be deleted

        current\_node = None

    def traverse(self):

        """Prints the data of each node in the list."""

        current\_node = self.head

        if current\_node is None:

            print("Linked list is empty.")

            return

        while current\_node:

            print(current\_node.data, end=" -> ")

            current\_node = current\_node.next

        print("None") # Indicate the end of the list

# Example Usage:

my\_list = SinglyLinkedList()

# Insert elements

my\_list.insert\_at\_end(10)

my\_list.insert\_at\_end(20)

my\_list.insert\_at\_end(30)

my\_list.insert\_at\_end(40)

print("Original list:")

my\_list.traverse()

# Delete a value

my\_list.delete\_value(20)

print("\nList after deleting 20:")

my\_list.traverse()

my\_list.delete\_value(50) # Try deleting a value not in the list

print("\nList after trying to delete 50:")

my\_list.traverse()

my\_list.delete\_value(10) # Delete the head

print("\nList after deleting 10:")

my\_list.traverse()

my\_list.delete\_value(40) # Delete the last element

print("\nList after deleting 40:")

my\_list.traverse()

my\_list.delete\_value(30) # Delete the last remaining element

print("\nList after deleting 30:")

my\_list.traverse()

my\_list.delete\_value(100) # Delete from empty list

print("\nList after trying to delete from empty list:")

my\_list.traverse()

OUTPUT:

Original list:

10 -> 20 -> 30 -> 40 -> None

List after deleting 20:

10 -> 30 -> 40 -> None

Value 50 not found in the list.

List after trying to delete 50:

10 -> 30 -> 40 -> None

List after deleting 10:

30 -> 40 -> None

List after deleting 40:

30 -> None

List after deleting 30:

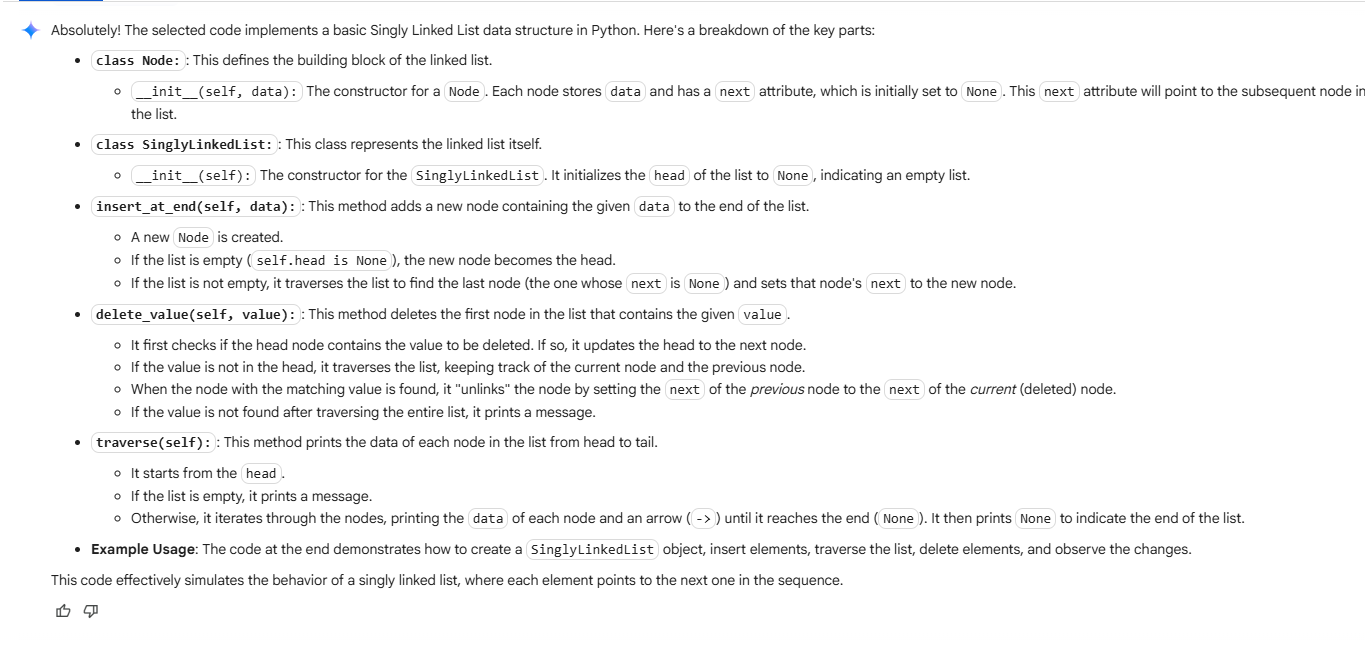
Linked list is empty.

Value 100 not found in the list.

List after trying to delete from empty list:

Linked list is empty.

EXPLAINATION:



TASK 4 : Implement a Binary Search Tree with methods for insert(),  
search(), and inorder\_traversal()

CODE :

class TreeNode:

    """Represents a node in a Binary Search Tree."""

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BinarySearchTree:

    """Represents a Binary Search Tree."""

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, key):

        """Inserts a new node with the given key into the BST."""

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

    def \_insert\_recursive(self, root, key):

        """Helper function for recursive insertion."""

        if root is None:

            return TreeNode(key)

        if key < root.key:

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

        elif key > root.key:

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

        # If key is equal, we can choose to ignore or handle duplicates

        return root

    def search(self, key):

        """Searches for a node with the given key in the BST."""

        return self.\_search\_recursive(self.root, key)

    def \_search\_recursive(self, root, key):

        """Helper function for recursive search."""

        # Base cases: root is none or key is present at root

        if root is None or root.key == key:

            return root

        # Key is greater than root's key

        if key > root.key:

            return self.\_search\_recursive(root.right, key)

        # Key is smaller than root's key

        return self.\_search\_recursive(root.left, key)

    def inorder\_traversal(self):

        """Performs an in-order traversal of the BST and prints the keys."""

        self.\_inorder\_recursive(self.root)

        print() # Add a newline for cleaner output

    def \_inorder\_recursive(self, root):

        """Helper function for recursive in-order traversal."""

        if root:

            self.\_inorder\_recursive(root.left)

            print(root.key, end=" ")

            self.\_inorder\_recursive(root.right)

# Example Usage:

my\_bst = BinarySearchTree()

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

for key in keys\_to\_insert:

    my\_bst.insert(key)

print("In-order traversal:")

my\_bst.inorder\_traversal() # Output: 20 30 40 50 60 70 80

search\_key = 40

result = my\_bst.search(search\_key)

if result:

    print(f"Node with key {search\_key} found in the BST.")

else:

    print(f"Node with key {search\_key} not found in the BST.")

search\_key = 90

result = my\_bst.search(search\_key)

if result:

    print(f"Node with key {search\_key} found in the BST.")

else:

    print(f"Node with key {search\_key} not found in the BST.")

OUTPUT:

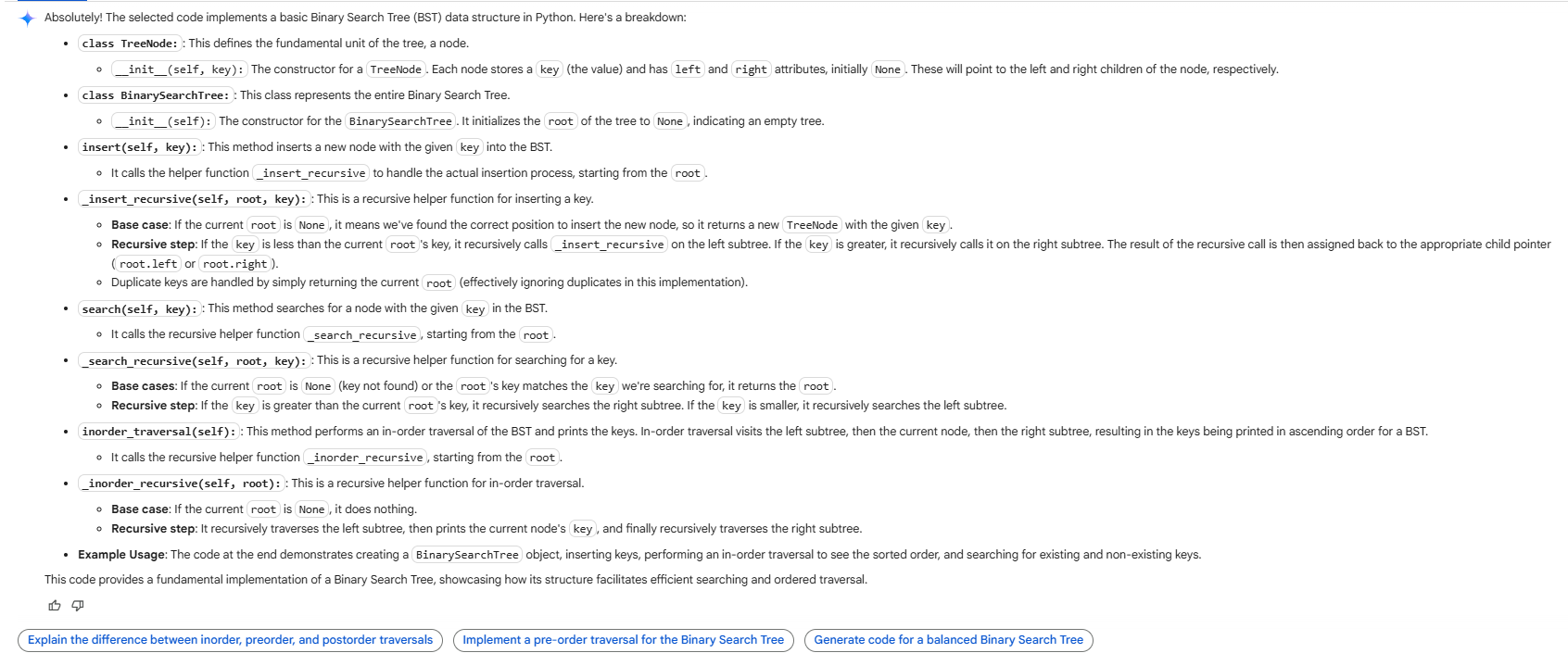
In-order traversal:

20 30 40 50 60 70 80

Node with key 40 found in the BST.

Node with key 90 not found in the BST.

EXPLAINATION:



TASK 5 : Implement a Graph using an adjacency list, with traversal  
methods BFS() and DFS().

CODE:

class Graph:

    """Represents a graph using an adjacency list."""

    def \_\_init\_\_(self):

        """Initializes an empty adjacency list."""

        self.graph = {}

    def add\_edge(self, u, v):

        """Adds an edge between vertices u and v."""

        # Add u to the graph if not already present

        if u not in self.graph:

            self.graph[u] = []

        # Add v to the graph if not already present

        if v not in self.graph:

            self.graph[v] = []

        # Add the edge (u, v) and (v, u) for an undirected graph

        self.graph[u].append(v)

        self.graph[v].append(u)

    def bfs(self, start\_node):

        """Performs a Breadth-First Search starting from a given node."""

        if start\_node not in self.graph:

            print(f"Node {start\_node} not found in the graph.")

            return

        visited = set()

        queue = []

        queue.append(start\_node)

        visited.add(start\_node)

        print(f"BFS starting from node {start\_node}:")

        while queue:

            current\_node = queue.pop(0)  # Dequeue from the front

            print(current\_node, end=" ")

            # Get neighbors and process them

            if current\_node in self.graph:

                for neighbor in self.graph[current\_node]:

                    if neighbor not in visited:

                        visited.add(neighbor)

                        queue.append(neighbor)

        print() # Newline after BFS traversal

    def dfs(self, start\_node):

        """Performs a Depth-First Search starting from a given node."""

        if start\_node not in self.graph:

            print(f"Node {start\_node} not found in the graph.")

            return

        visited = set()

        def dfs\_recursive(node):

            """Helper function for recursive DFS traversal."""

            visited.add(node)

            print(node, end=" ")

            if node in self.graph:

                for neighbor in self.graph[node]:

                    if neighbor not in visited:

                        dfs\_recursive(neighbor)

        print(f"DFS starting from node {start\_node}:")

        dfs\_recursive(start\_node)

        print() # Newline after DFS traversal

# Example code to demonstrate Graph usage with BFS and DFS

# Create a graph instance

my\_graph = Graph()

# Add edges to the graph

my\_graph.add\_edge(0, 1)

my\_graph.add\_edge(0, 2)

my\_graph.add\_edge(1, 2)

my\_graph.add\_edge(2, 0) # Adding a redundant edge to show adjacency list handling

my\_graph.add\_edge(2, 3)

my\_graph.add\_edge(3, 3) # Self-loop example

my\_graph.add\_edge(4, 5) # Disconnected component

print("Graph adjacency list:", my\_graph.graph)

# Demonstrate BFS traversal

my\_graph.bfs(0)

my\_graph.bfs(4) # BFS from a disconnected node

my\_graph.bfs(10) # BFS from a non-existent node

# Demonstrate DFS traversal

my\_graph.dfs(0)

my\_graph.dfs(4) # DFS from a disconnected node

my\_graph.dfs(10) # DFS from a non-existent node

OUTPUT:

Graph adjacency list: {0: [1, 2, 2], 1: [0, 2], 2: [0, 1, 0, 3], 3: [2, 3, 3], 4: [5], 5: [4]}

BFS starting from node 0:

0 1 2 3

BFS starting from node 4:

4 5

Node 10 not found in the graph.

DFS starting from node 0:

0 1 2 3

DFS starting from node 4:

4 5

Node 10 not found in the graph.

GRAPH:

import networkx as nx

import matplotlib.pyplot as plt

# Assuming 'my\_graph' is the Graph instance from the previous cell

# Create a NetworkX graph from the adjacency list

G = nx.Graph(my\_graph.graph)

# Draw the graph

pos = nx.spring\_layout(G) # You can choose different layouts, e.g., nx.circular\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_size=700, node\_color='skyblue', font\_size=10, font\_weight='bold')

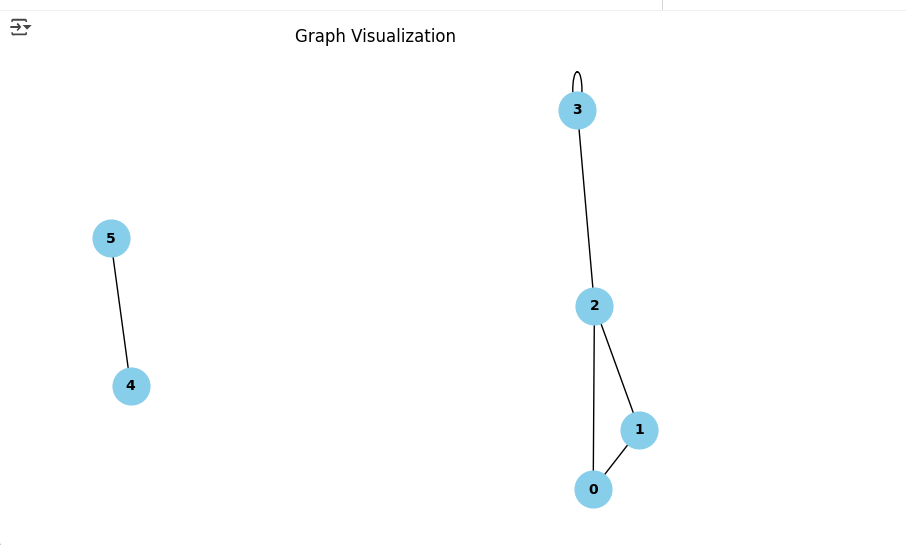
# Draw edge labels (optional)

# edge\_labels = nx.get\_edge\_attributes(G, 'weight') # if you had edge weights

# nx.draw\_edge\_labels(G, pos, edge\_labels=edge\_labels)

plt.title("Graph Visualization")

plt.show()



EXPLAINATION:

