AI Assisted Coding

Assignment – 11.4

Name : S.Sathwik

# Roll Number : 2403A51273

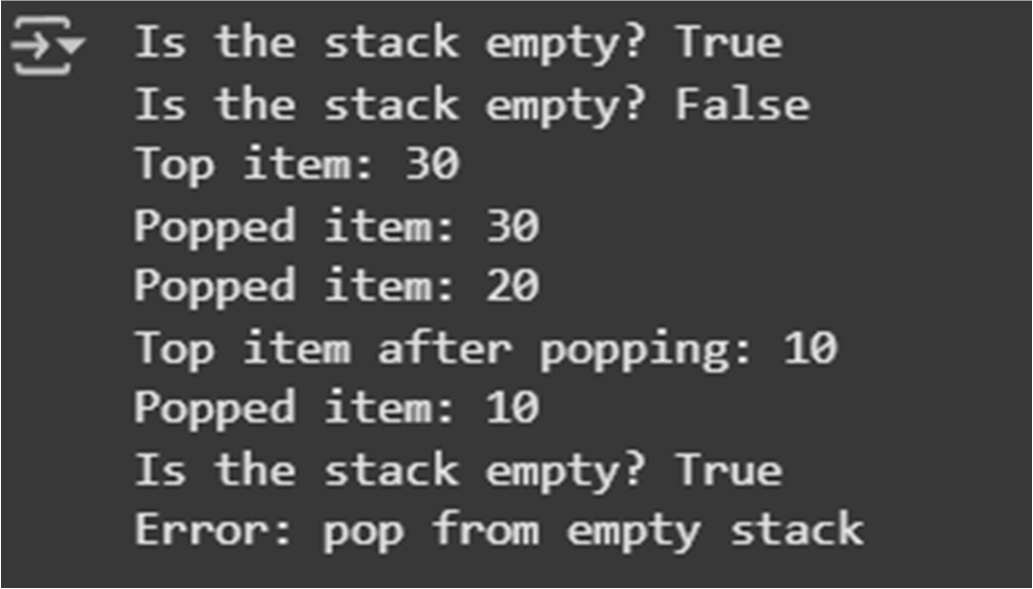
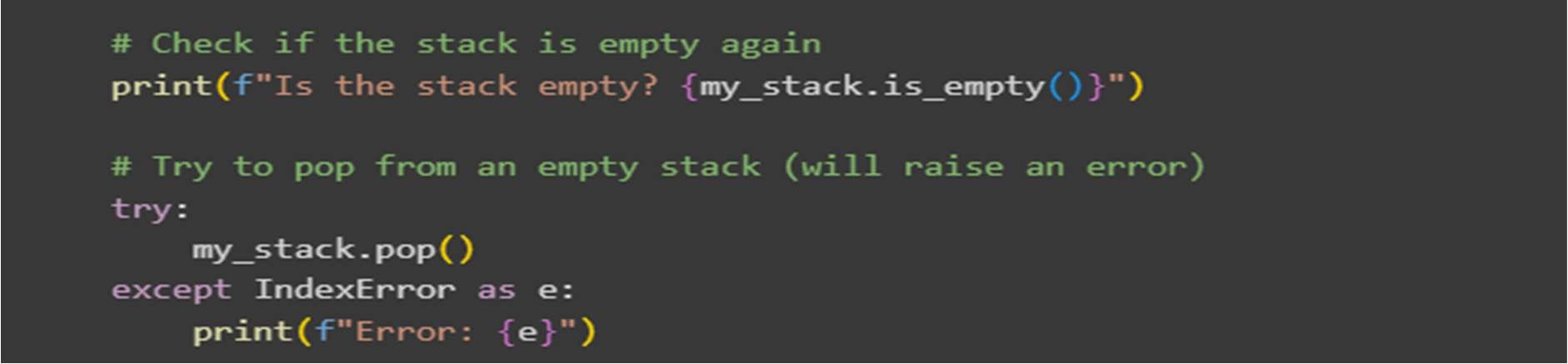
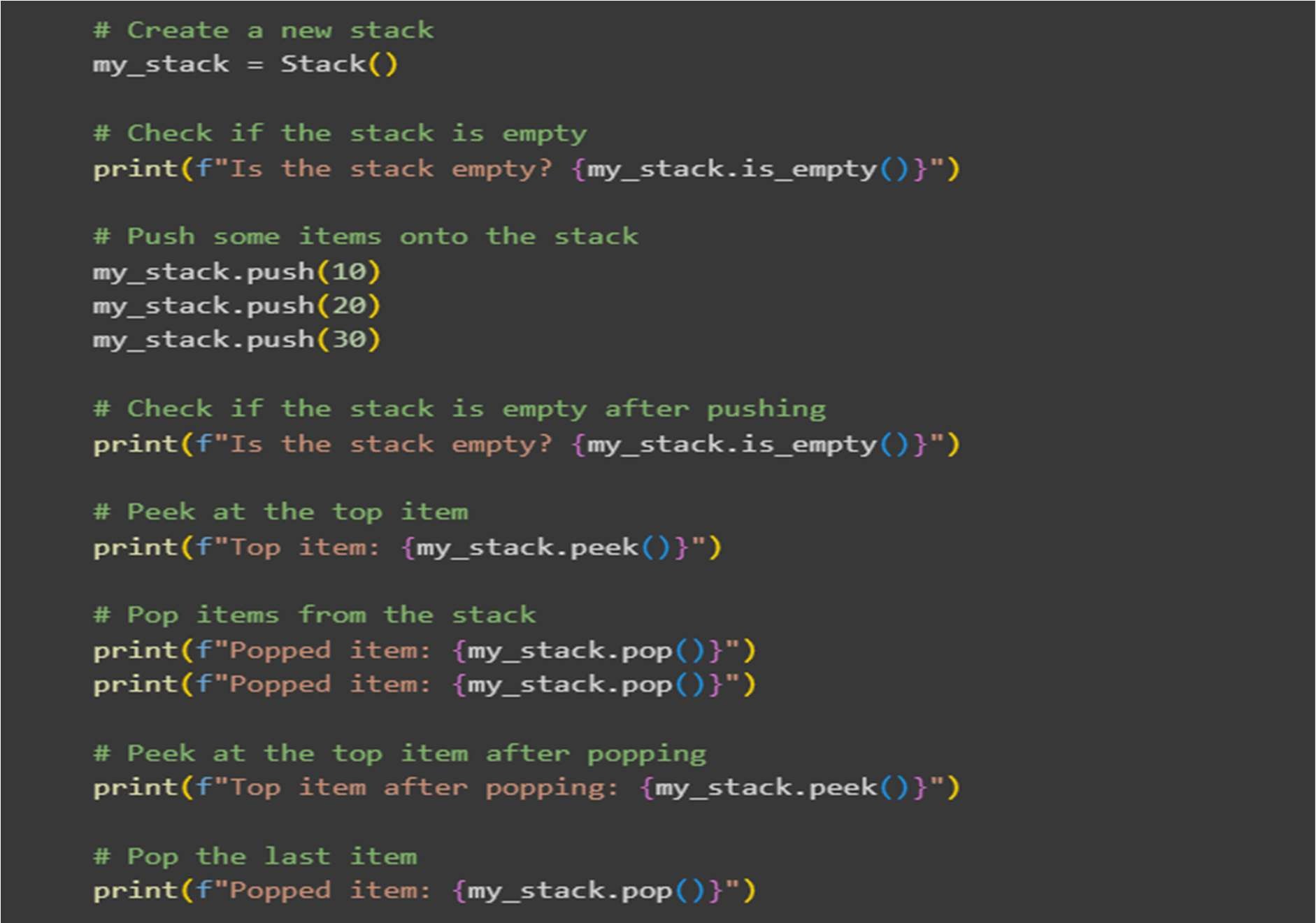
# Batch : 12

# 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. o Test stack operations using sample data.
    - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).
* Expected Output: o A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts.

Code :

Output:



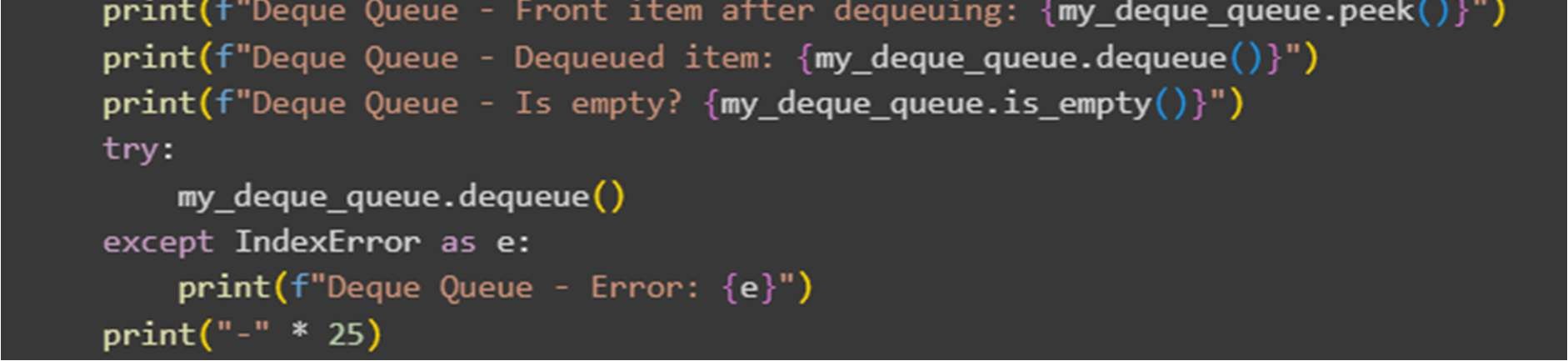
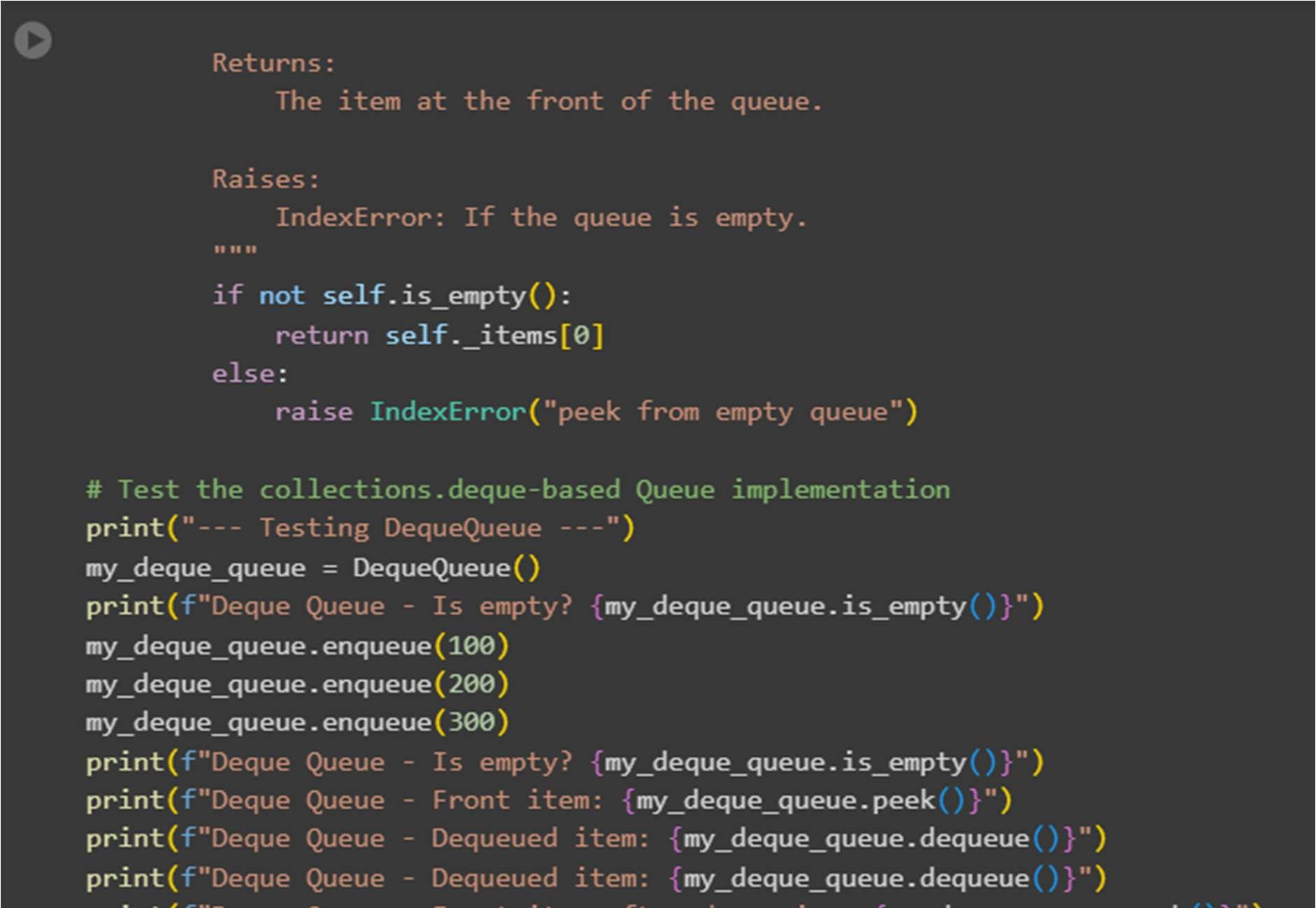
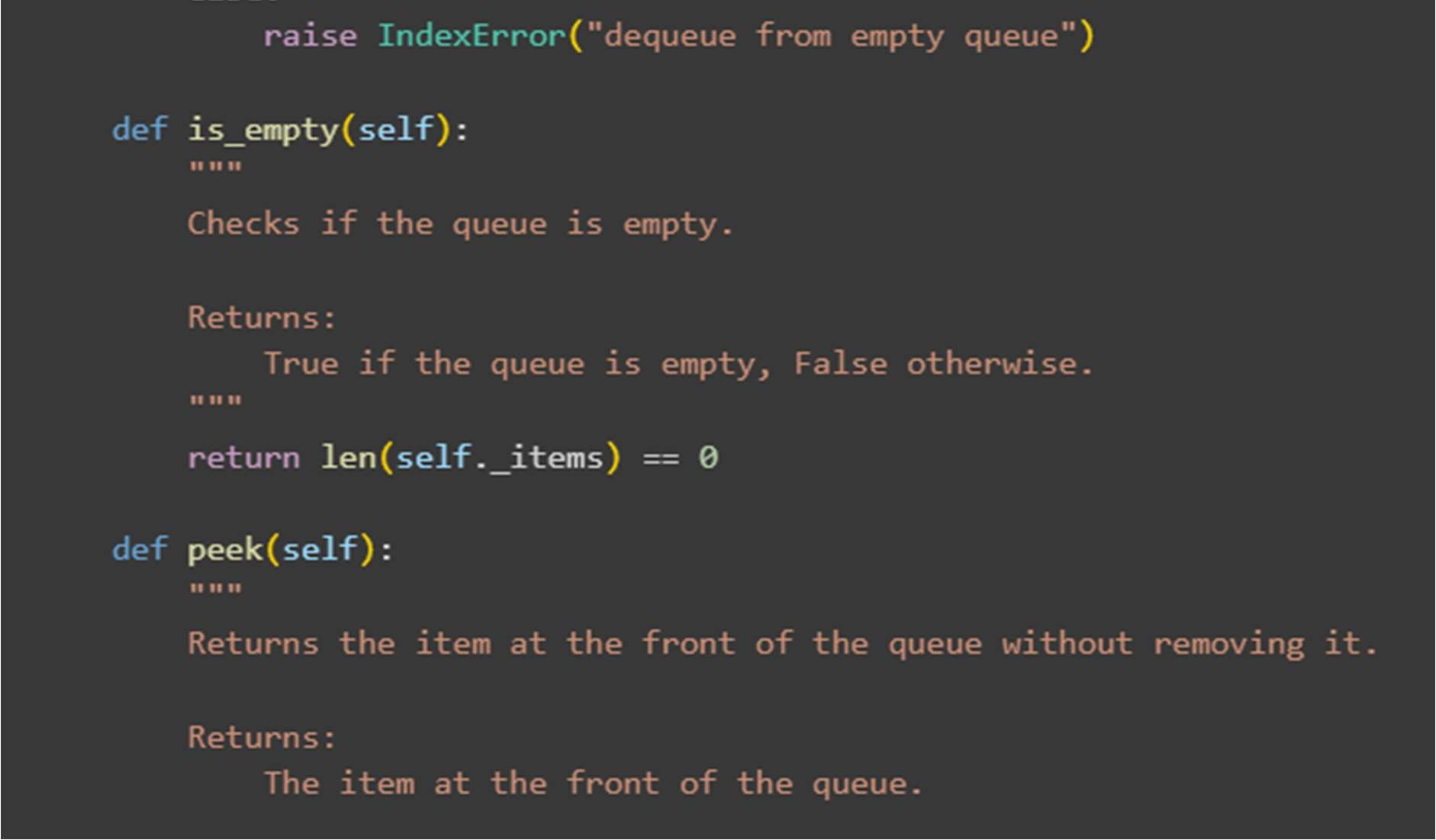
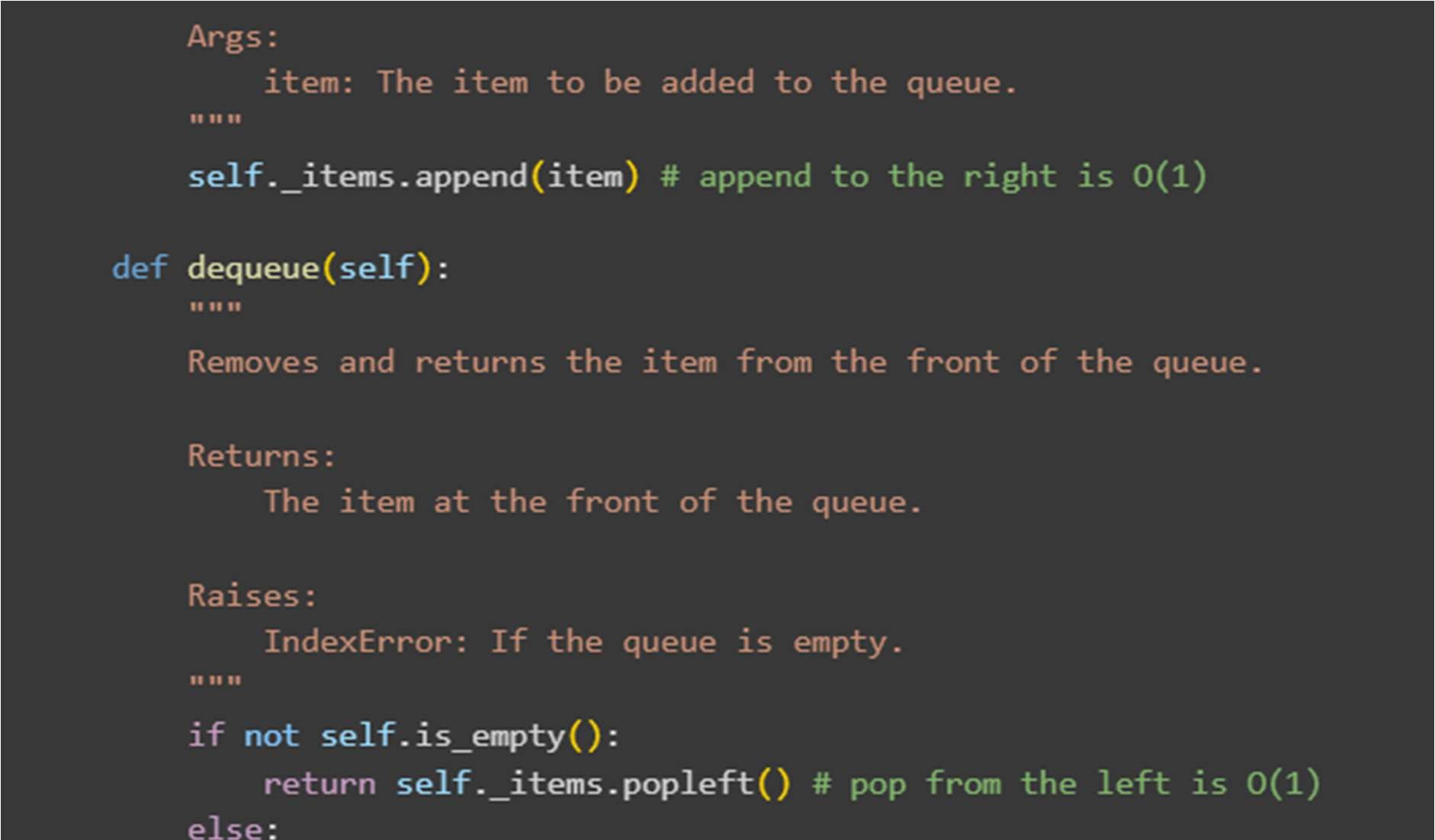
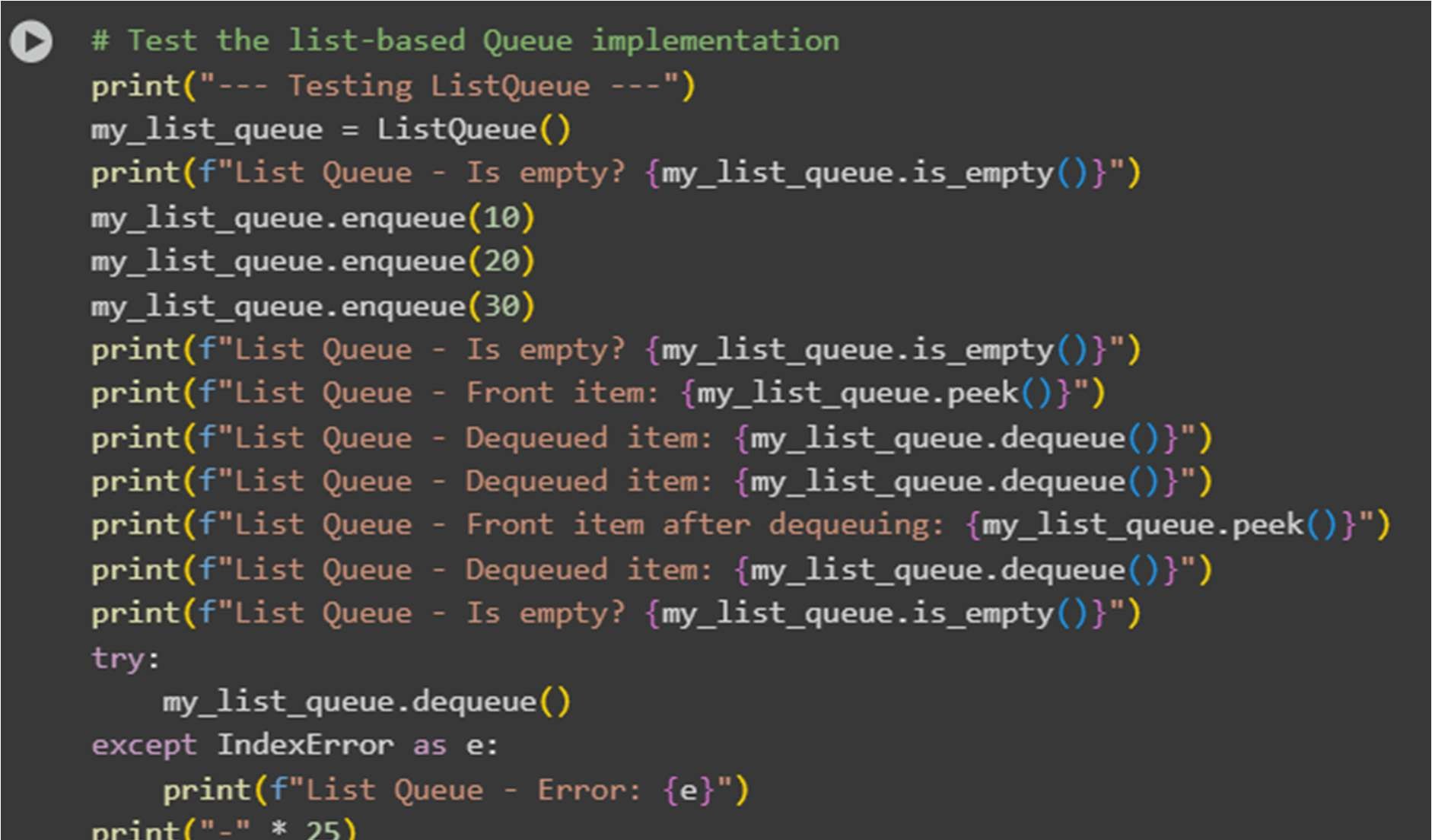
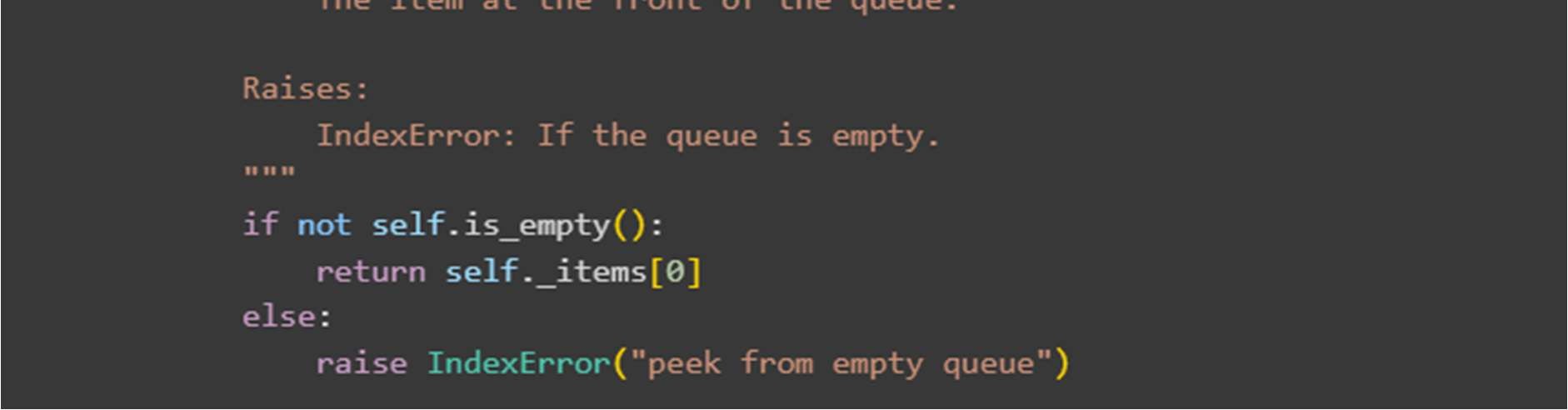
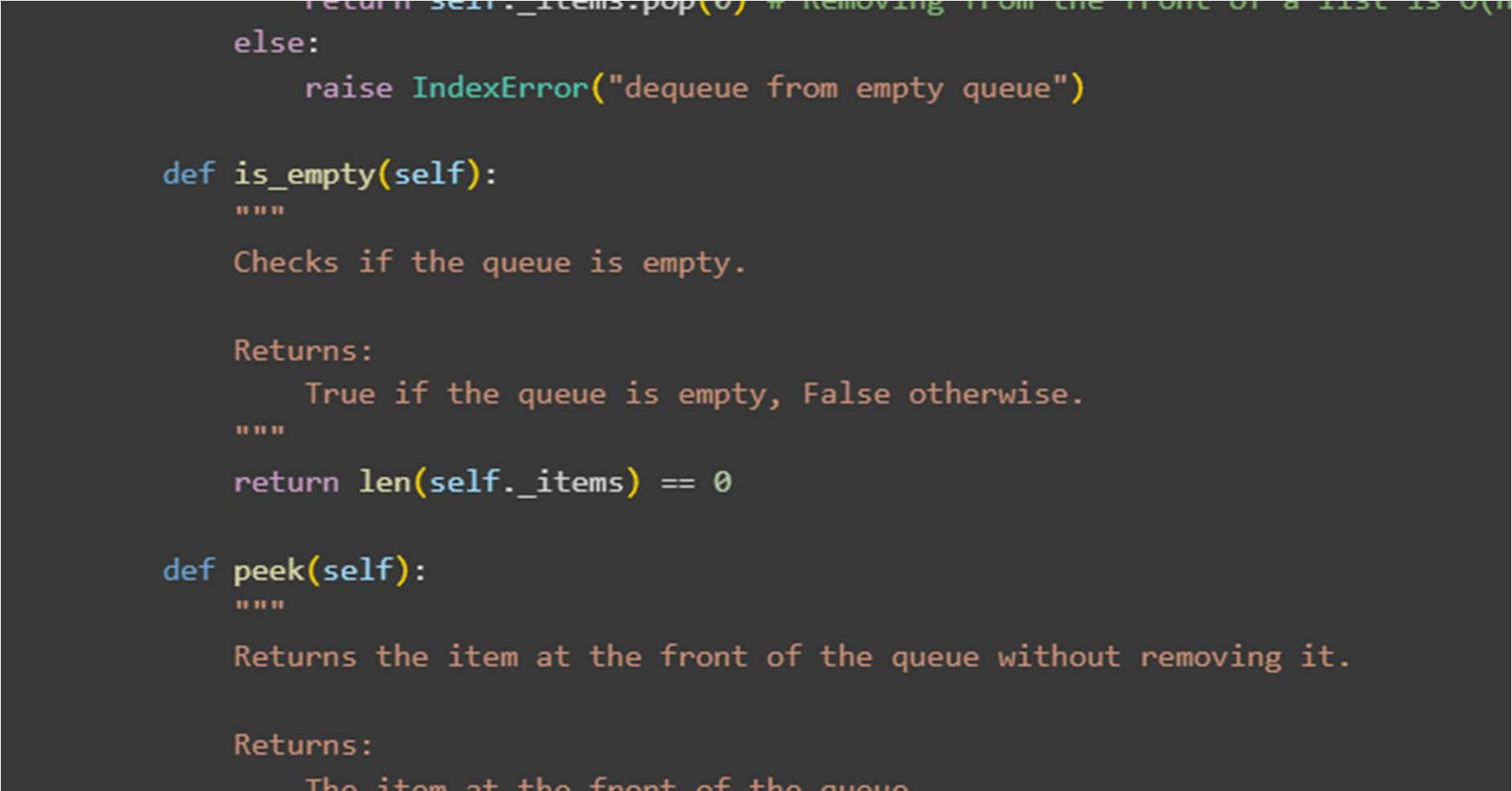
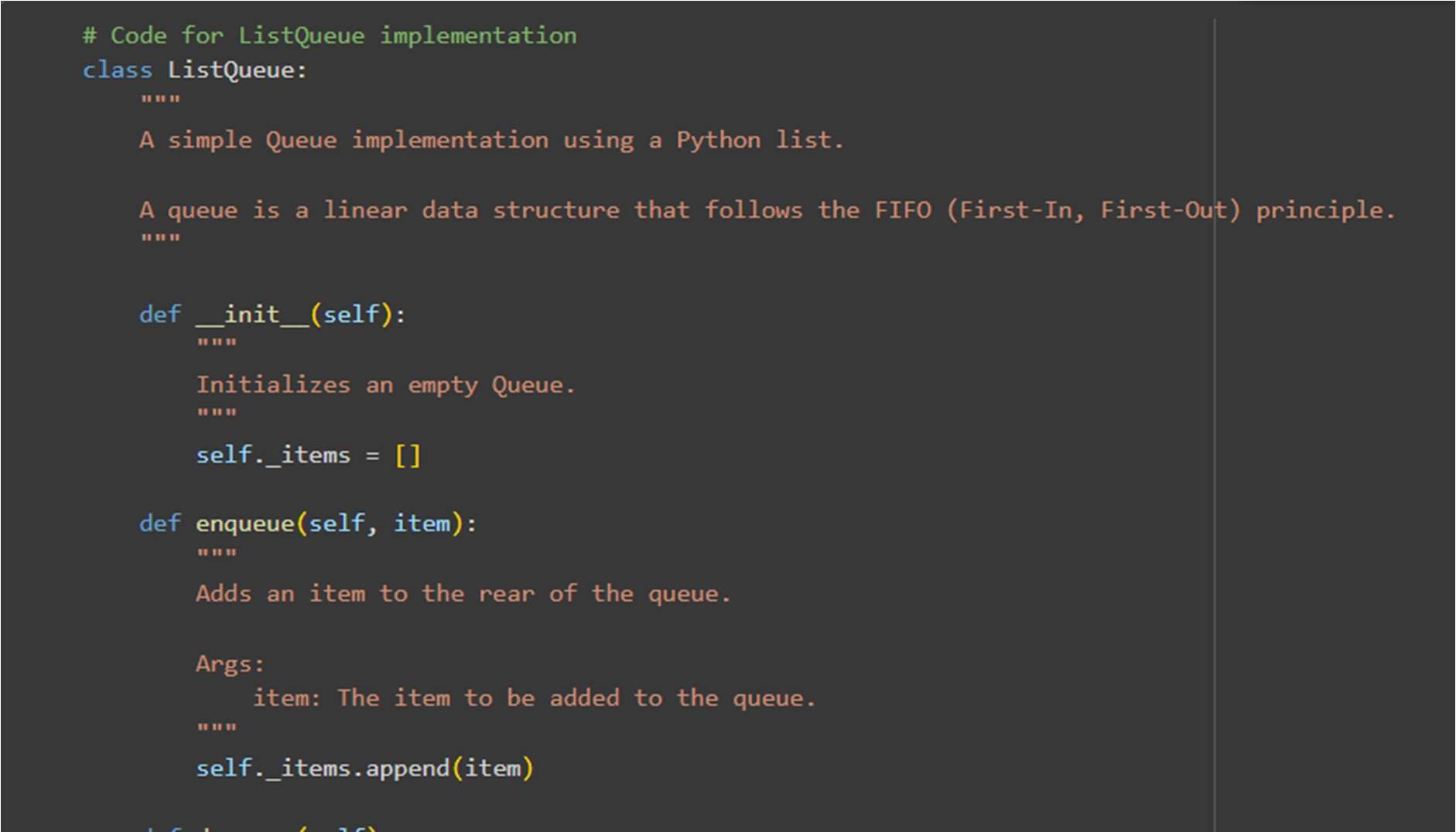
Explanation:

* 1. my\_stack = Stack(): This line creates a new, empty instance of your Stack class and assigns it to the variable my\_stack.
  2. print(f"Is the stack empty? {my\_stack.is\_empty()}"): This calls the is\_empty() method on the my\_stack object to check if the stack is currently empty. Since it was just created, it will print True.
  3. my\_stack.push(10), my\_stack.push(20), my\_stack.push(30): These lines use the push() method to add the integers 10, 20, and 30 onto the top of the stack, in that order.
  4. print(f"Is the stack empty? {my\_stack.is\_empty()}"): This checks if the stack is empty again after adding items. It will now print False.
  5. print(f"Top item: {my\_stack.peek()}"): This calls the peek() method, which returns the item at the top of the stack (which is 30) without removing it.
  6. print(f"Popped item: {my\_stack.pop()}"), print(f"Popped item: {my\_stack.pop()}"): These lines call the pop() method twice. Each call removes and returns the item from the top of the stack. First 30 is popped, then 20.
  7. print(f"Top item after popping: {my\_stack.peek()}"): After the two pops, the item now at the top of the stack is 10. This line calls peek() again to show this.
  8. print(f"Popped item: {my\_stack.pop()}"): This pops the last remaining item (10) from the stack.
  9. print(f"Is the stack empty? {my\_stack.is\_empty()}"): This checks the stack's empty status again. Since all items have been popped, it will print True.
  10. try...except IndexError as e: print(f"Error: {e}"): This block attempts to call my\_stack.pop() one more time. Since the stack is now empty, calling pop() will raise an IndexError. The try...except block catches this specific error, and the code inside the except block is executed, printing the error message "pop from empty stack". This demonstrates how the pop() method handles an empty stack.

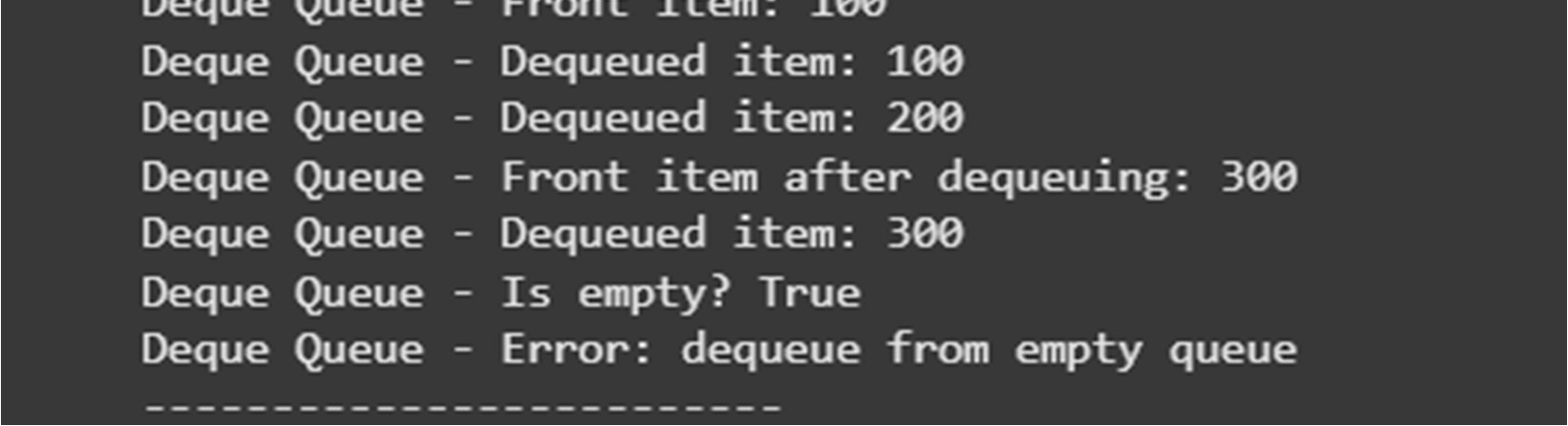
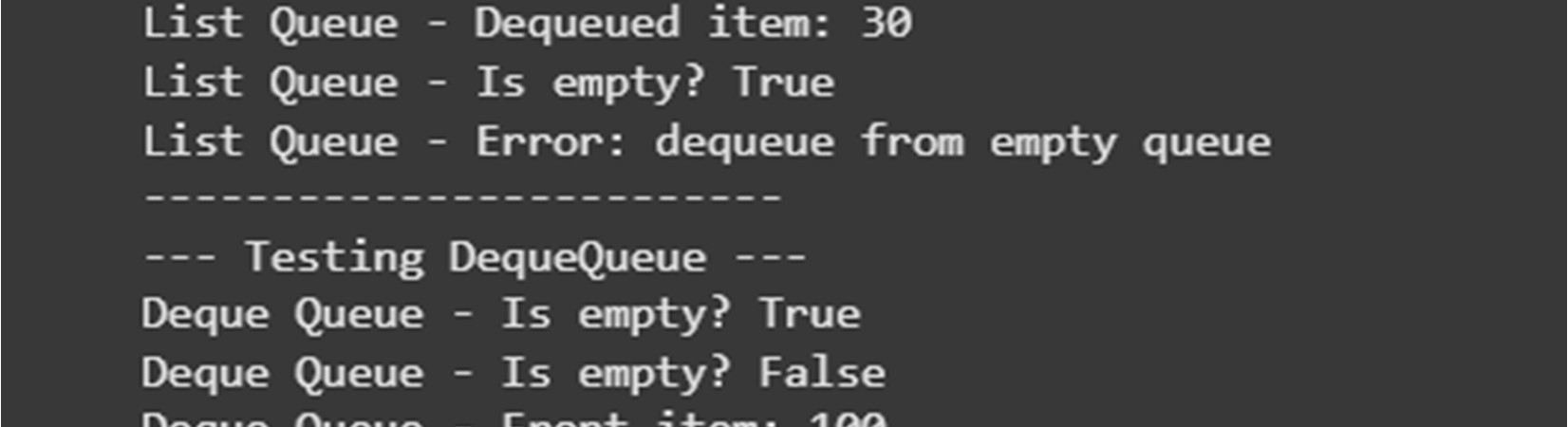
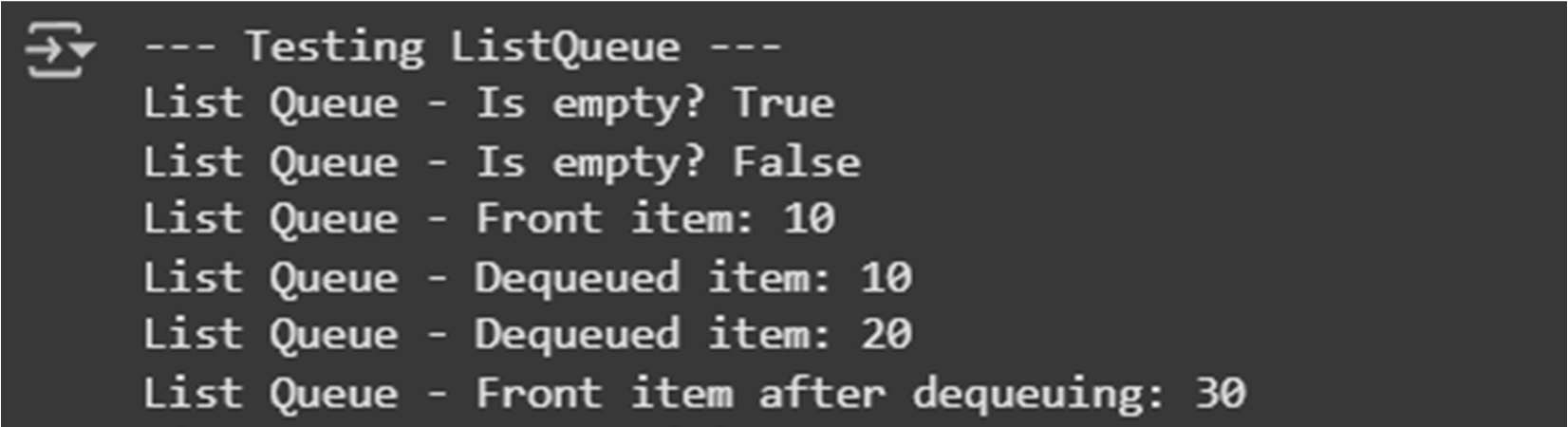
# Task 2: Queue Implementation with Performance Review

* 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 e icient implementation (using collections.deque).
* Expected Output: o Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison.

Code:

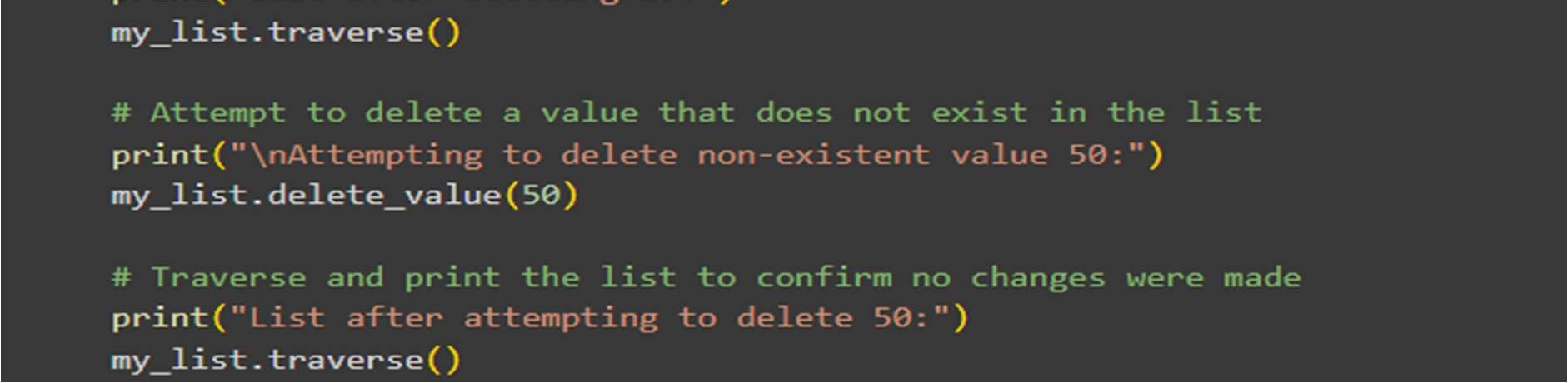
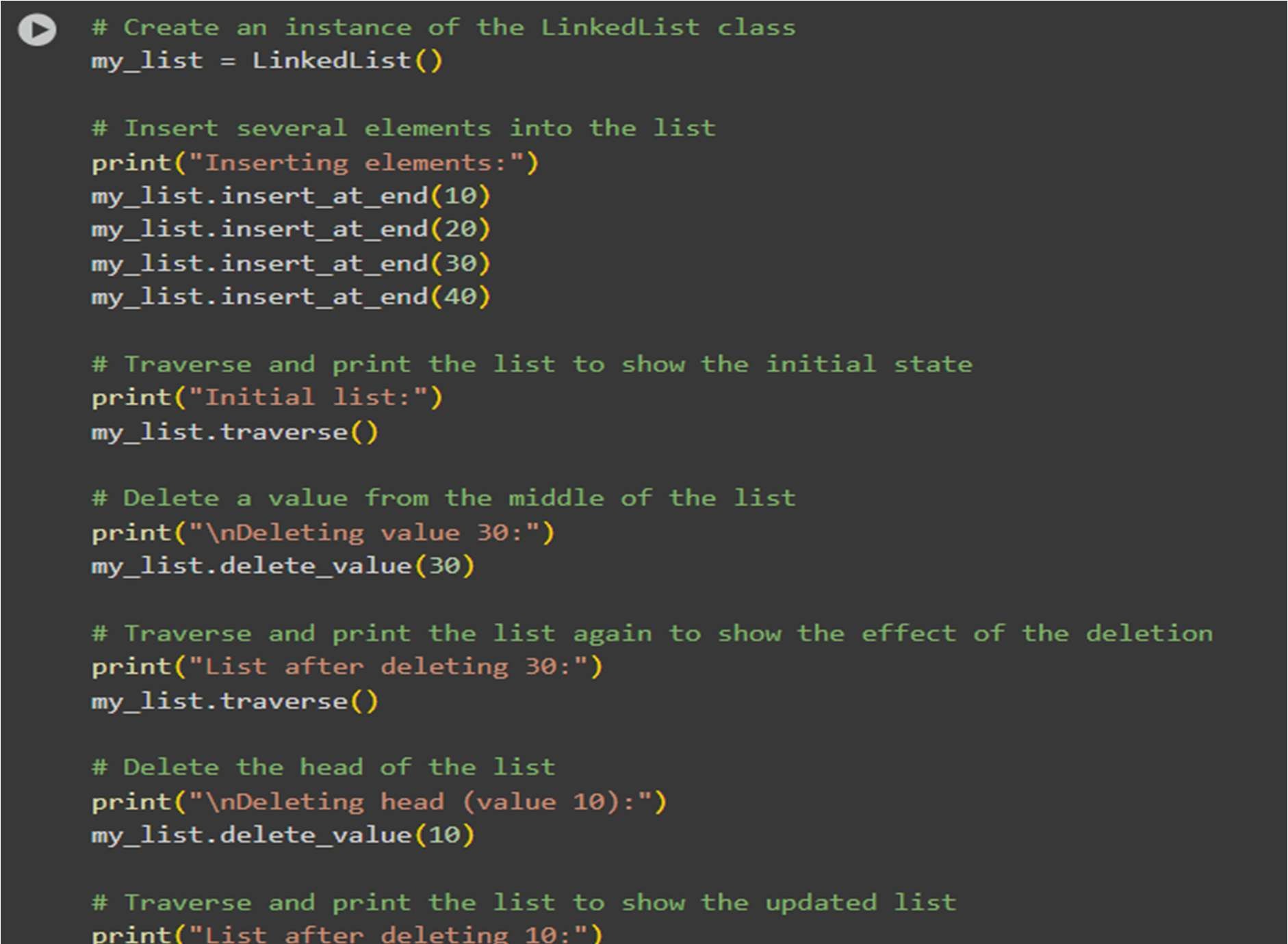


Output :

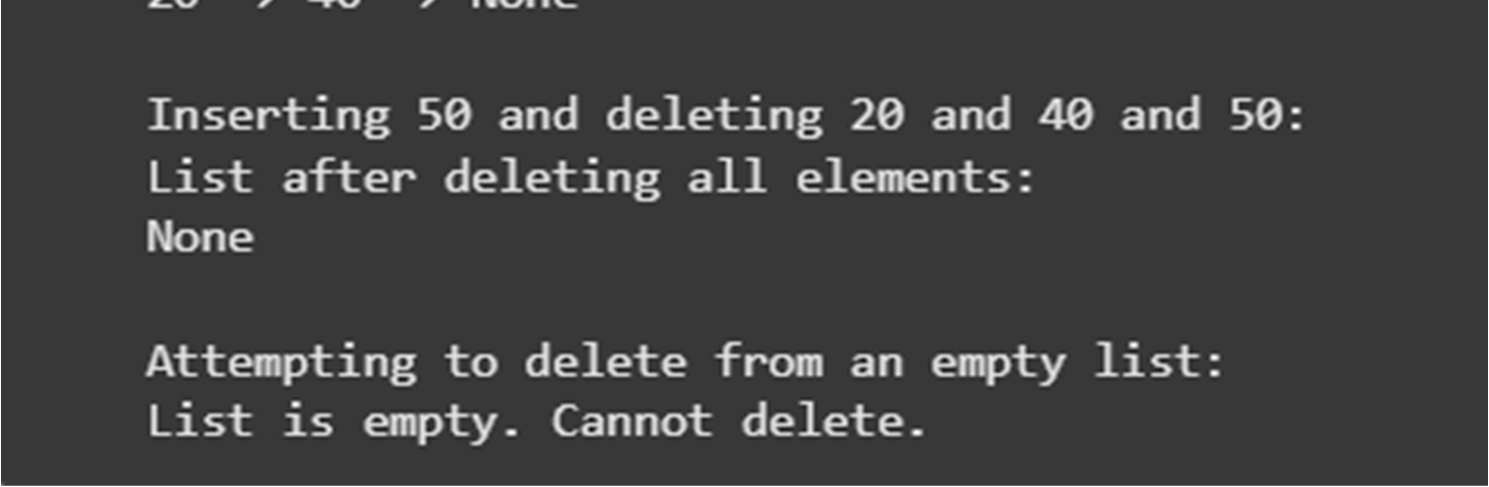
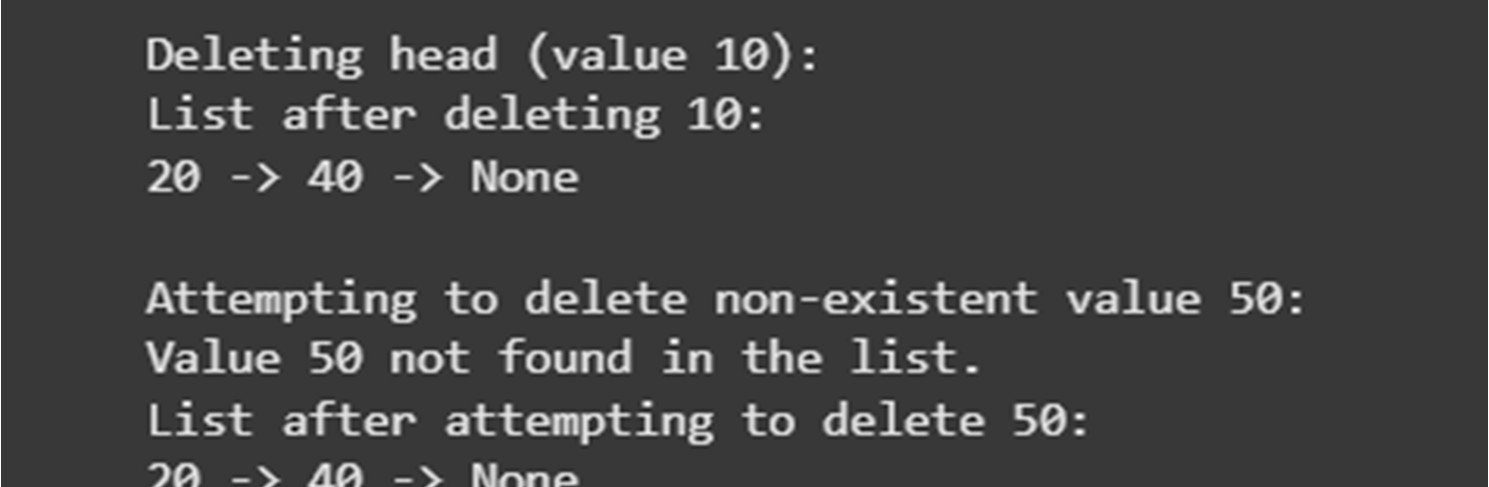
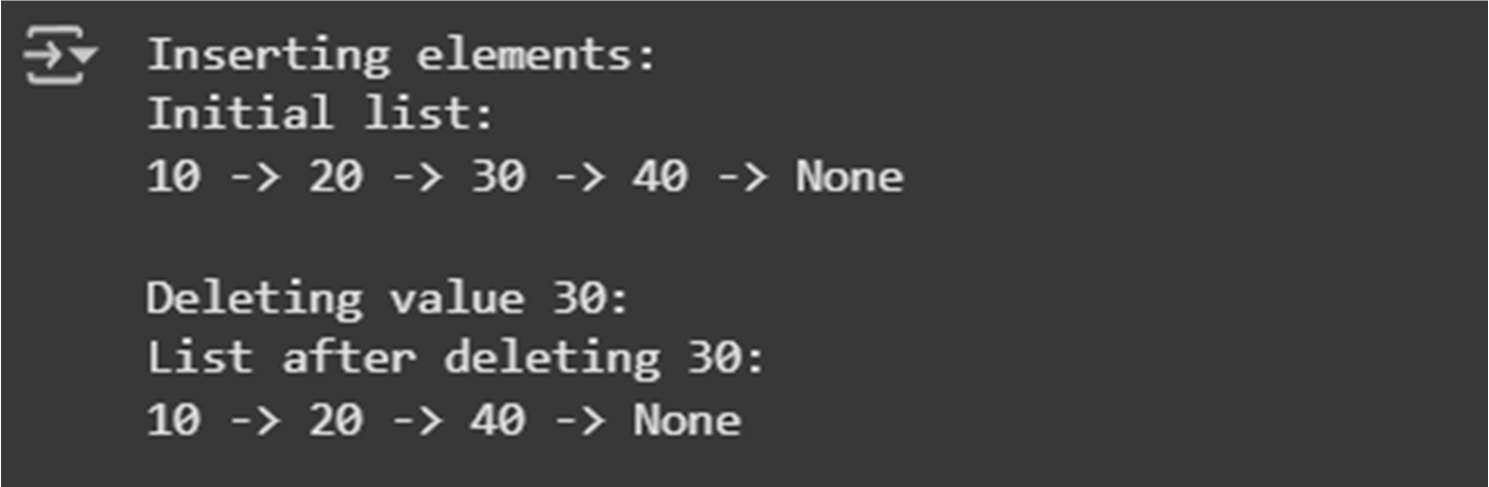
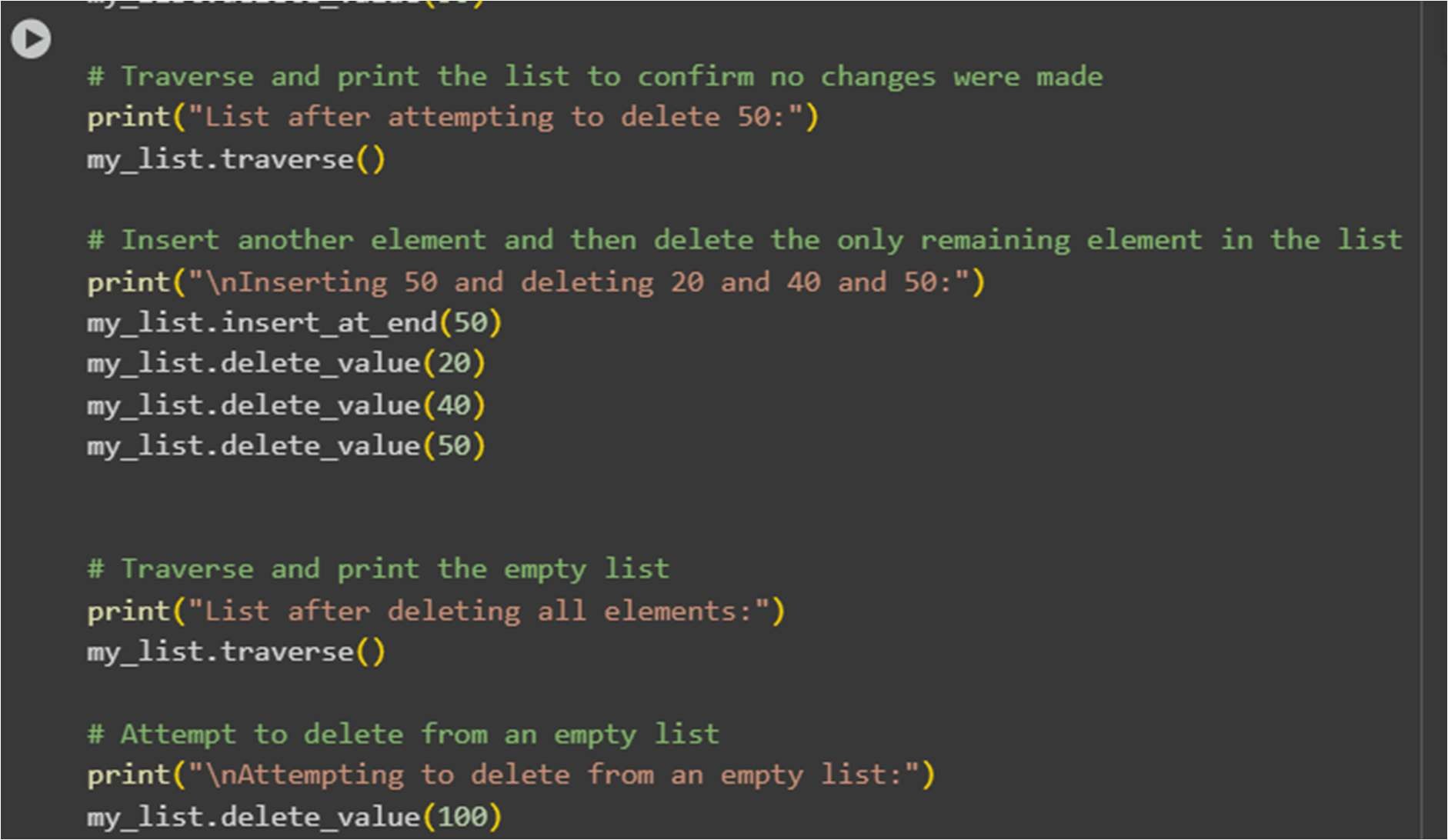


# Task 3: Singly Linked List with Traversal

* 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). o Ask AI to suggest test cases to validate all operations.
* Expected Output: o A functional linked list implementation with clear comments explaining the logic of insertions and deletions Code:



Output:



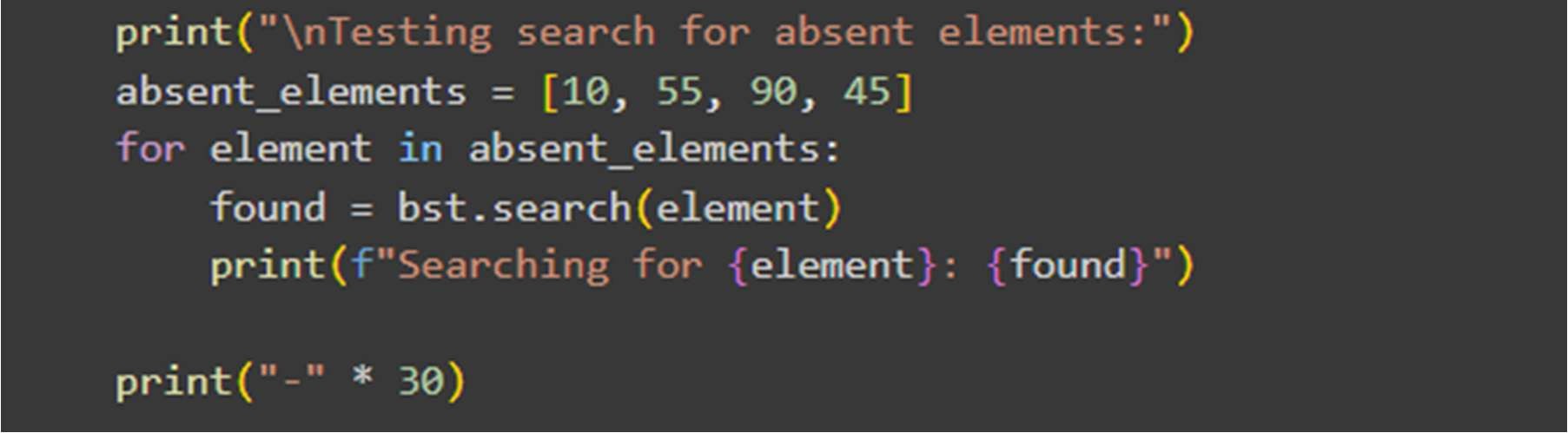
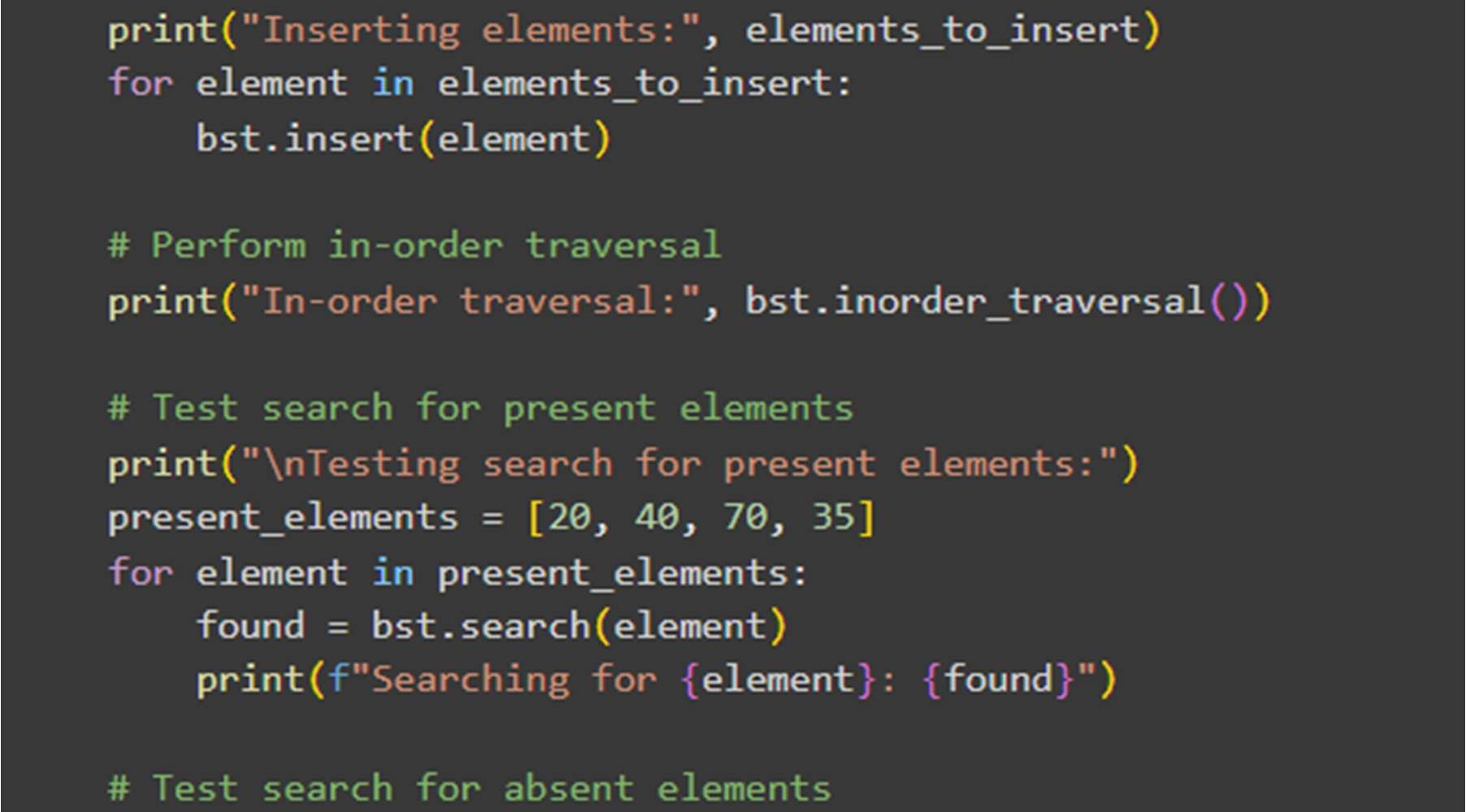
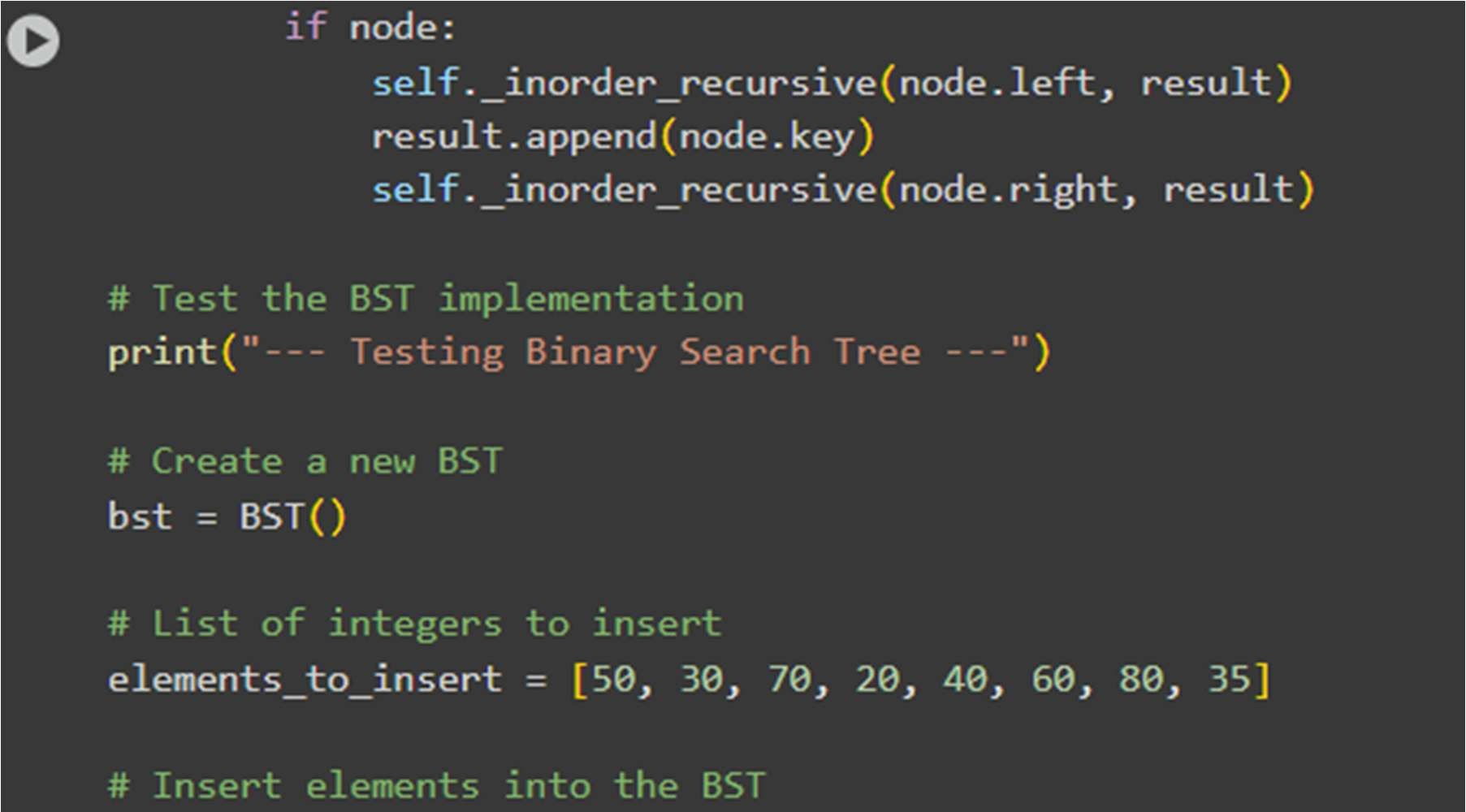
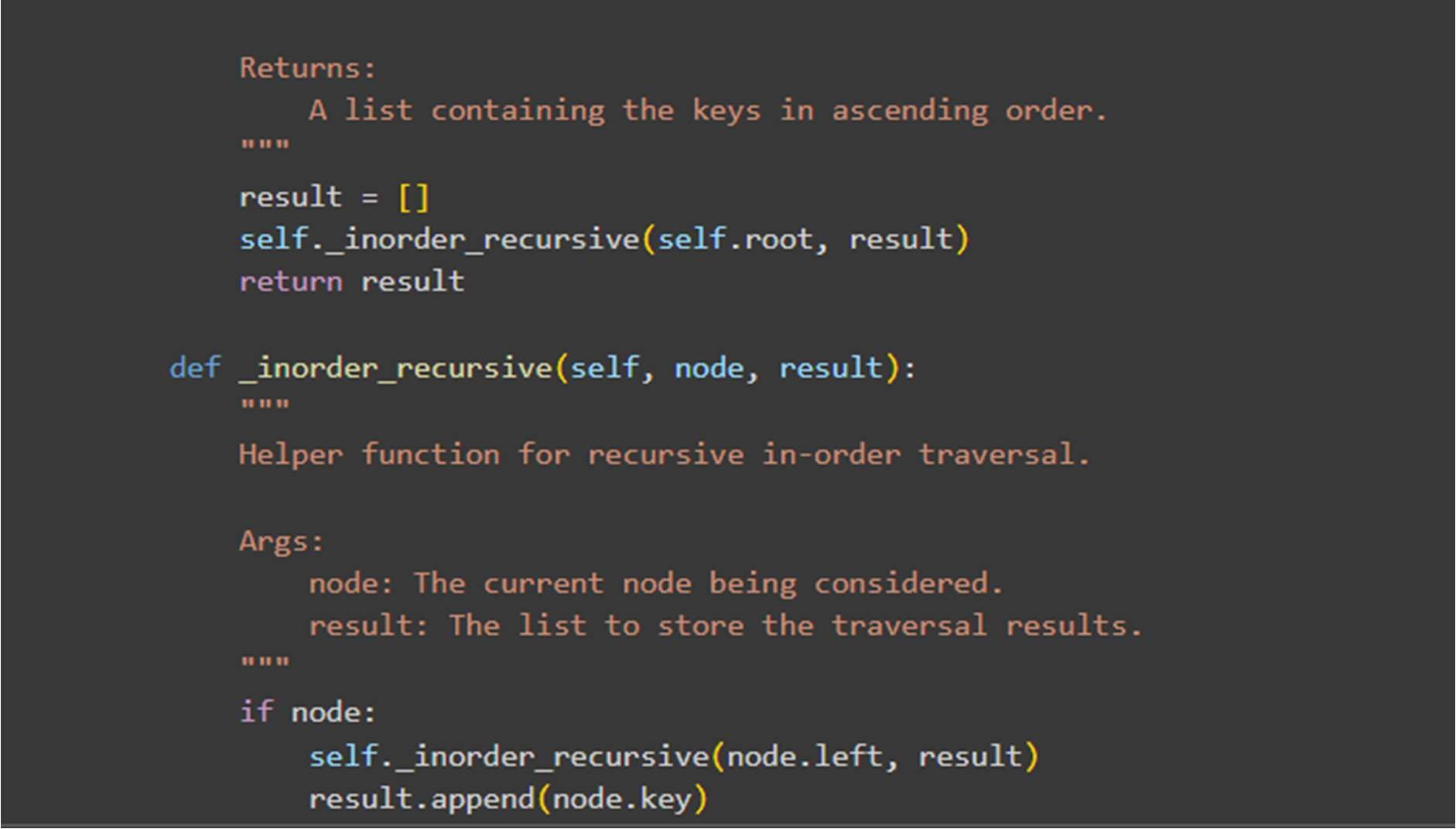
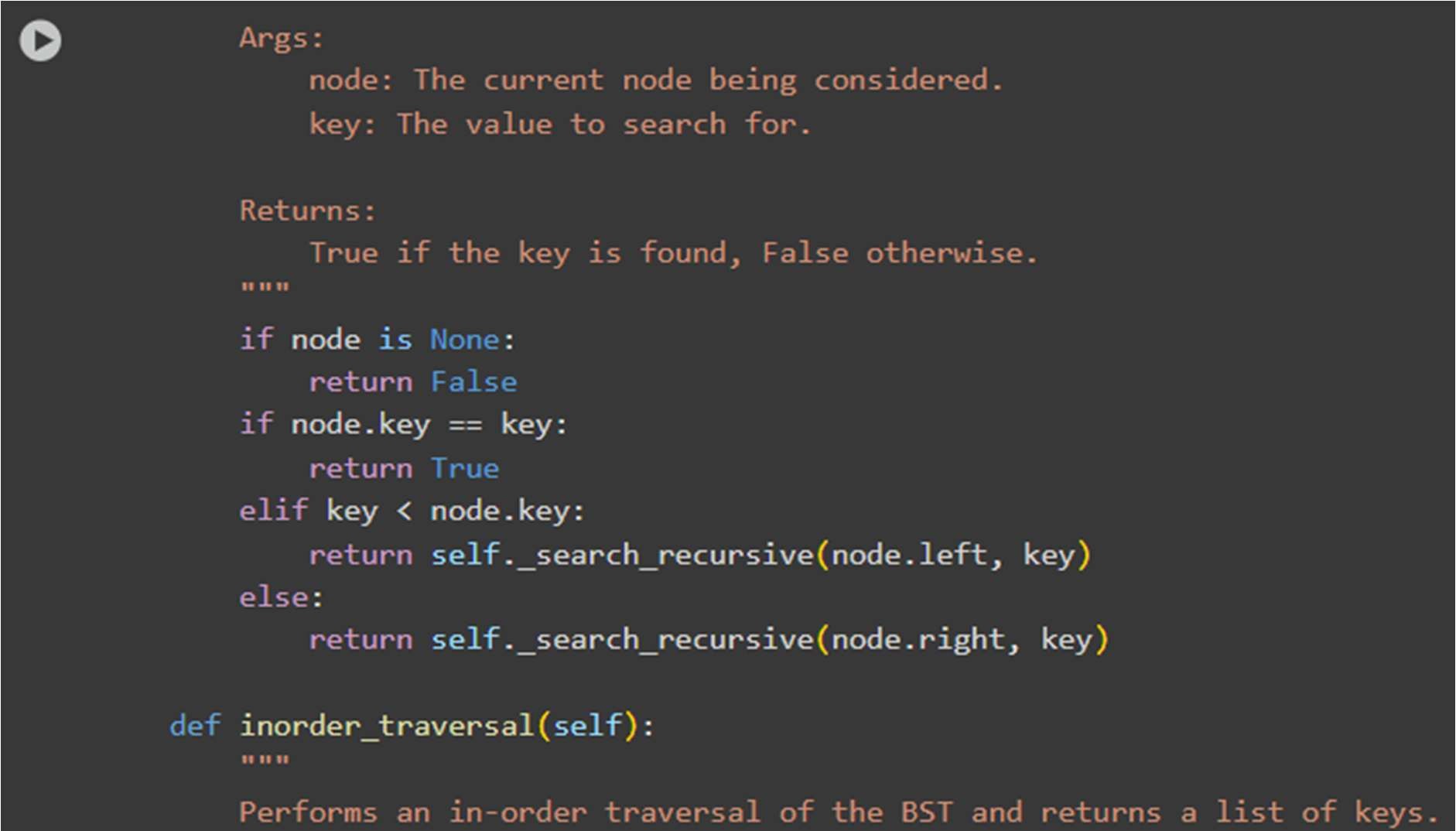
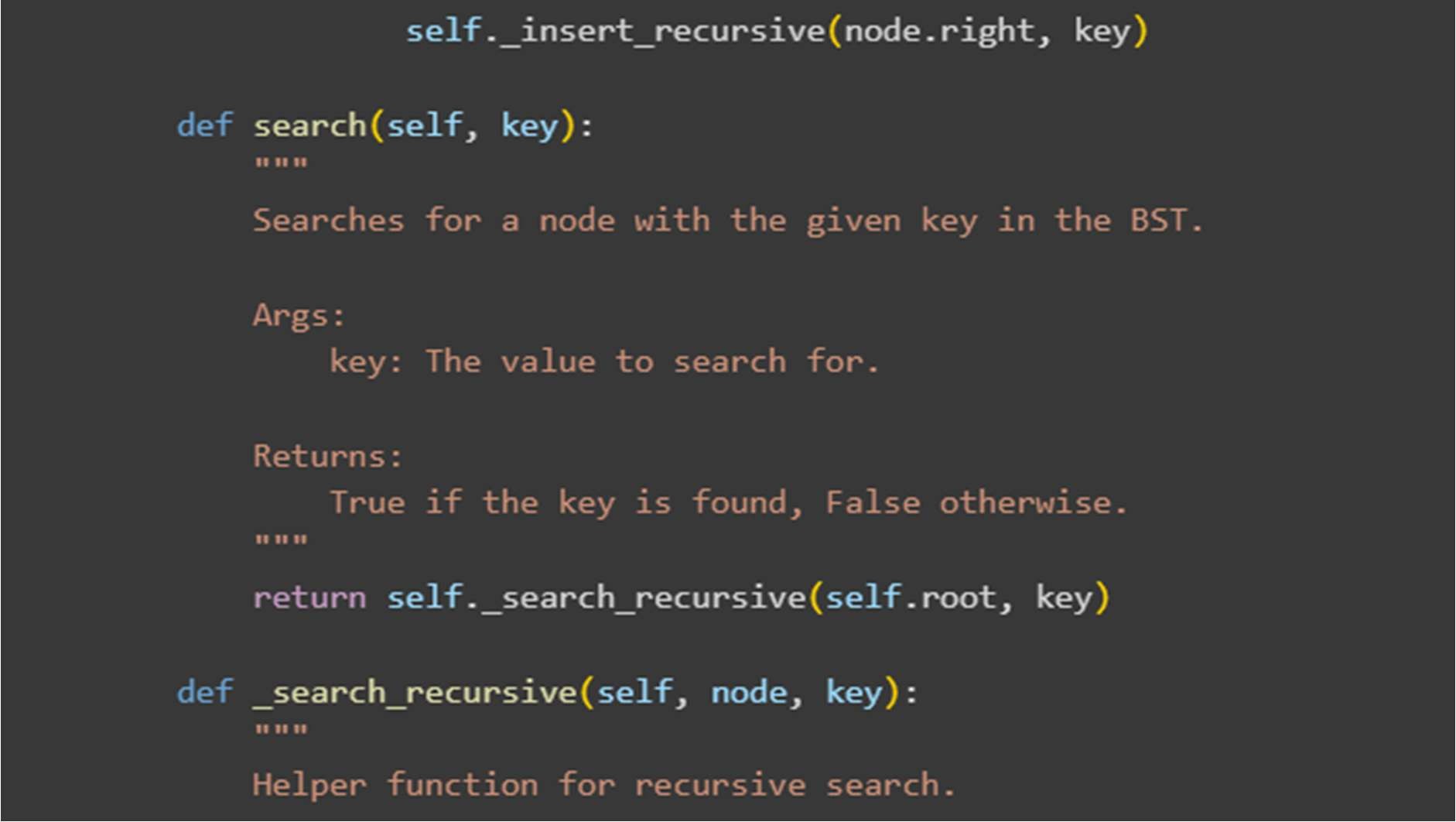
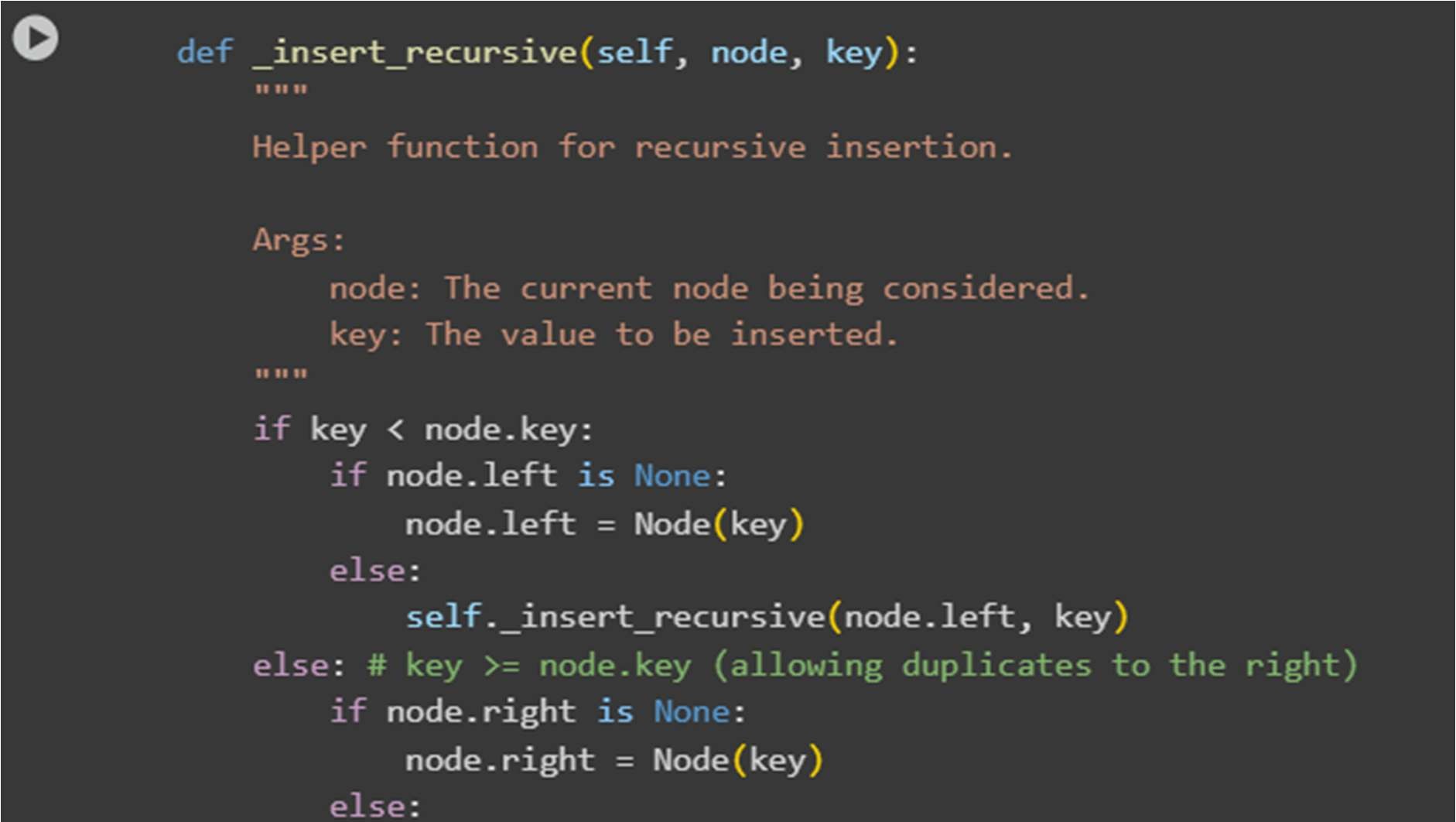
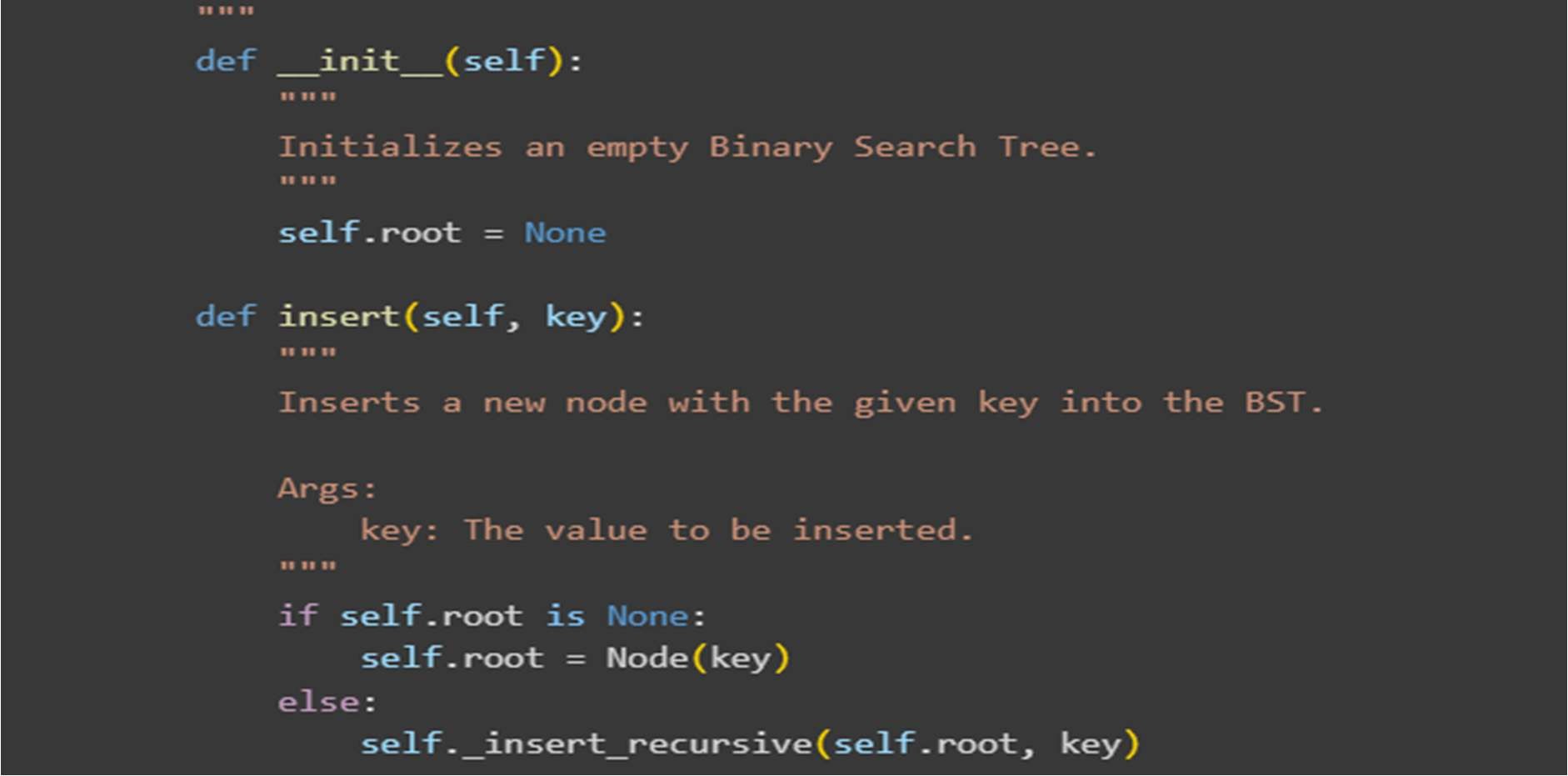
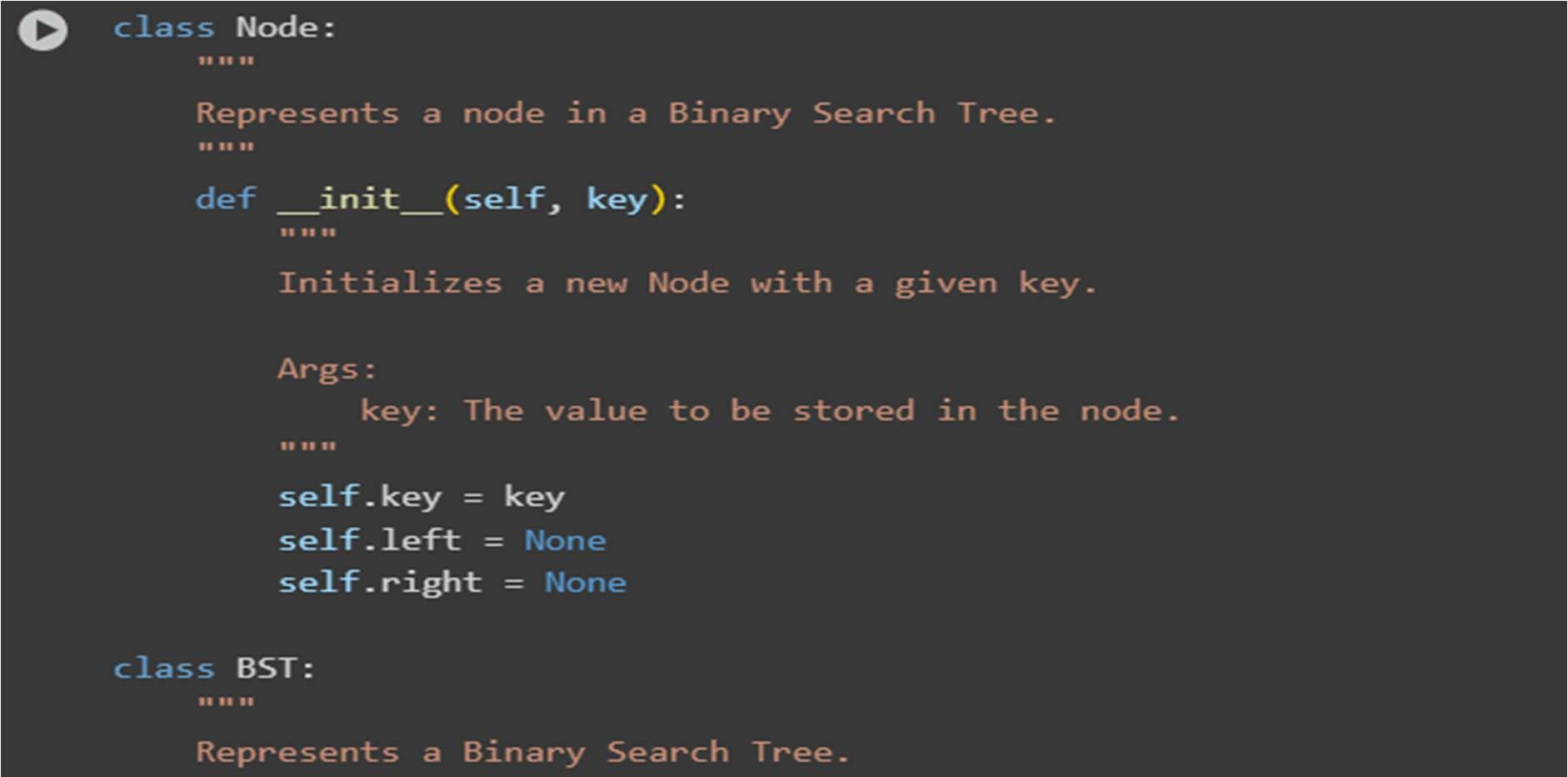
Explanation :

* 1. my\_list = LinkedList(): This line creates a new, empty instance of your LinkedList class.
  2. print("Inserting elements:"): This simply prints a header to indicate the start of insertion tests.
  3. my\_list.insert\_at\_end(10), my\_list.insert\_at\_end(20), my\_list.insert\_at\_end(30), my\_list.insert\_at\_end(40): These lines call the insert\_at\_end() method to add the values 10, 20, 30, and 40 to the end of the linked list. The list will become 10 -> 20 -> 30 -> 40 -> None.
  4. print("Initial list:"): Prints a header.
  5. my\_list.traverse(): Calls the traverse() method to print the current elements of the list. This confirms the initial insertions.
  6. print("\nDeleting value 30:"): Prints a header for the deletion test.
  7. my\_list.delete\_value(30): Calls delete\_value() to remove the node with the value 30 from the list. The list should become 10 -> 20 -> 40 -> None.
  8. print("List after deleting 30:"): Prints a header.
  9. my\_list.traverse(): Calls traverse() to show the list after deleting 30.
  10. print("\nDeleting head (value 10):"): Prints a header for deleting the head node.
  11. my\_list.delete\_value(10): Calls delete\_value() to remove the node with the value 10, which is currently the head. The list should become 20 -> 40 -> None.
  12. print("List after deleting 10:"): Prints a header.
  13. my\_list.traverse(): Calls traverse() to show the list after deleting the head.
  14. print("\nAttempting to delete non-existent value 50:"): Prints a header for testing deletion of a non-existent value.
  15. my\_list.delete\_value(50): Calls delete\_value() to try and remove 50, which is not in the list. The delete\_value method should handle this gracefully (e.g., by printing a message) and the list should remain unchanged: 20 -> 40 -> None.
  16. print("List after attempting to delete 50:"): Prints a header.
  17. my\_list.traverse(): Calls traverse() to confirm the list is unchanged.
  18. print("\nInserting 50 and deleting 20 and 40 and 50:"): Prints a header for testing deletion of multiple elements, including inserting a new one first.
  19. my\_list.insert\_at\_end(50): Inserts 50 at the end: 20 -> 40 -> 50 -> None.
  20. my\_list.delete\_value(20), my\_list.delete\_value(40), my\_list.delete\_value(50): These lines sequentially delete the remaining elements (20, then 40, then 50) until the list is empty.
  21. print("List after deleting all elements:"): Prints a header.
  22. my\_list.traverse(): Calls traverse() on the now empty list, which should just print None.
  23. print("\nAttempting to delete from an empty list:"): Prints a header for testing deletion from an empty list.
  24. my\_list.delete\_value(100): Calls delete\_value() on the empty list. The method should handle this by printing a message like "List is empty. Cannot delete."

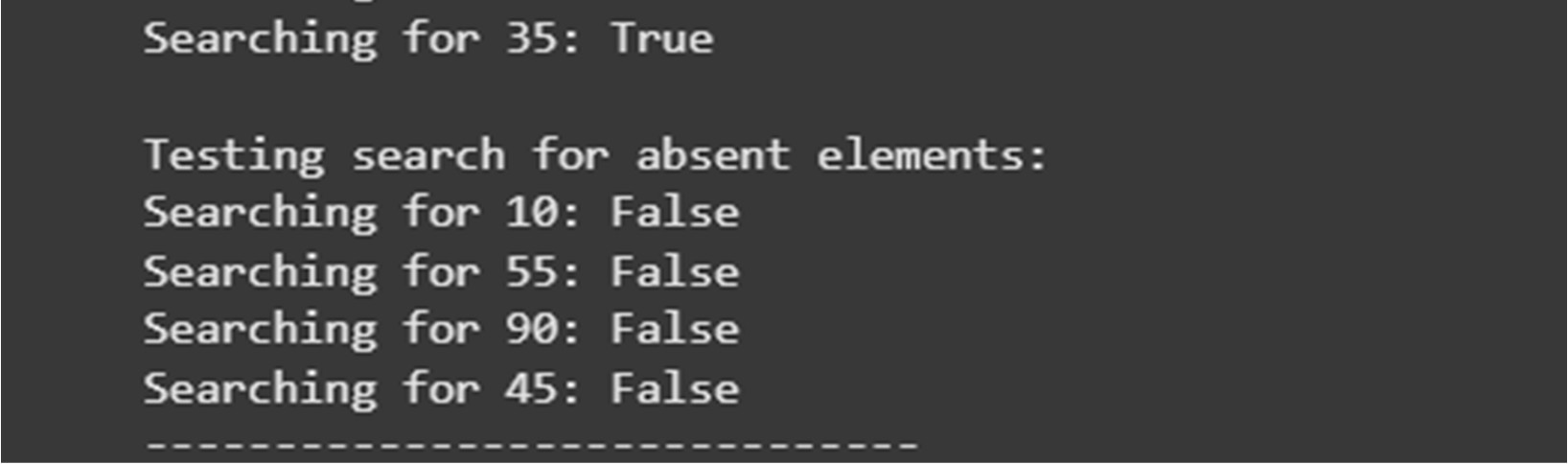
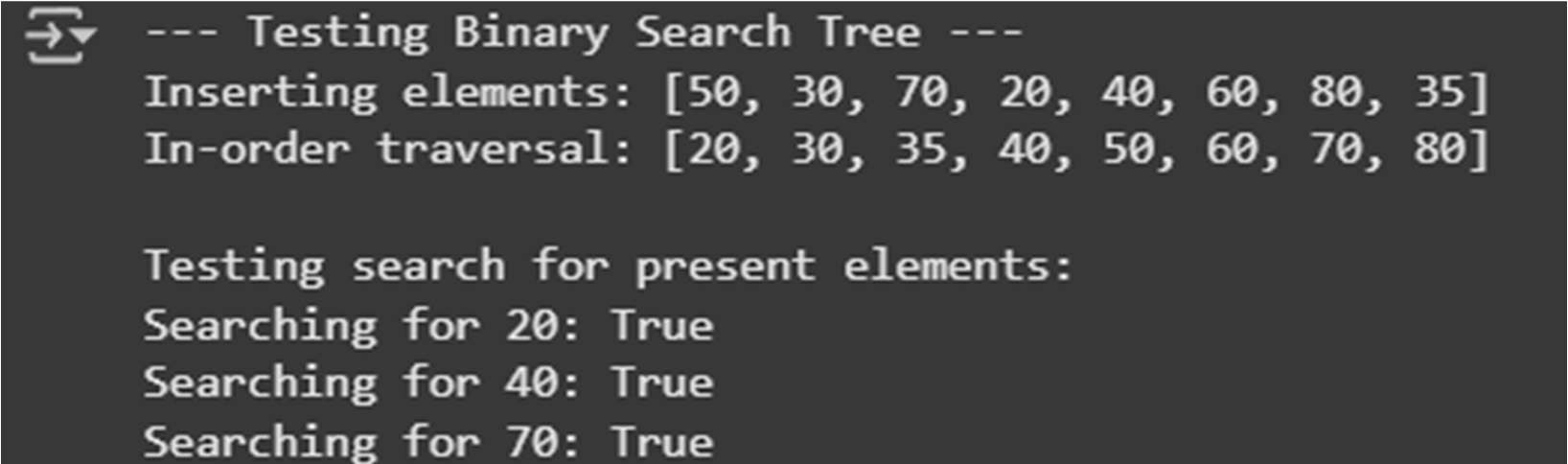
# Task 4: Binary Search Tree (BST)

* Task: Implement a Binary Search Tree with methods for insert(), search(), and inorder\_traversal().
* Instructions: o 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.

Code:



Output:



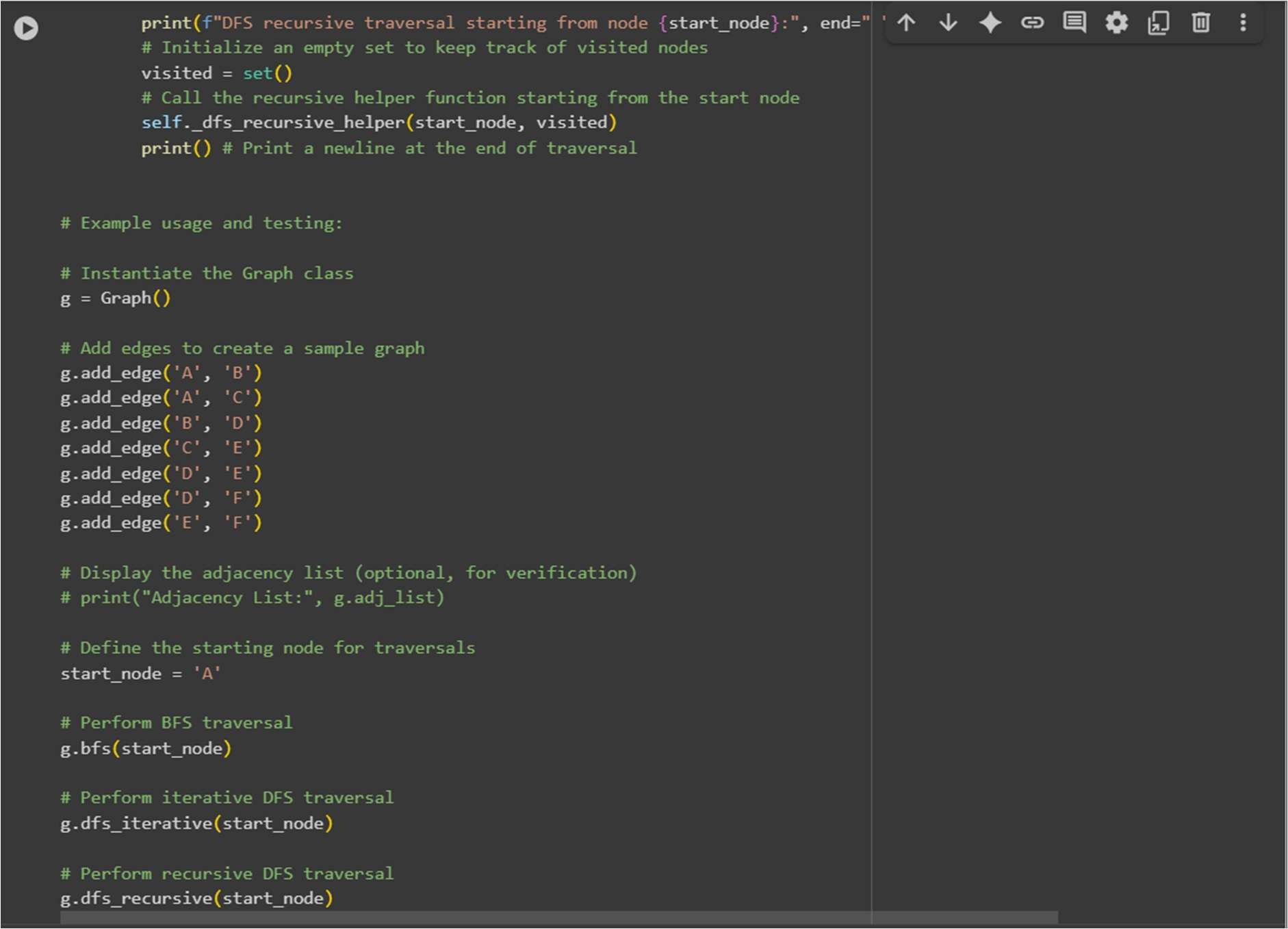
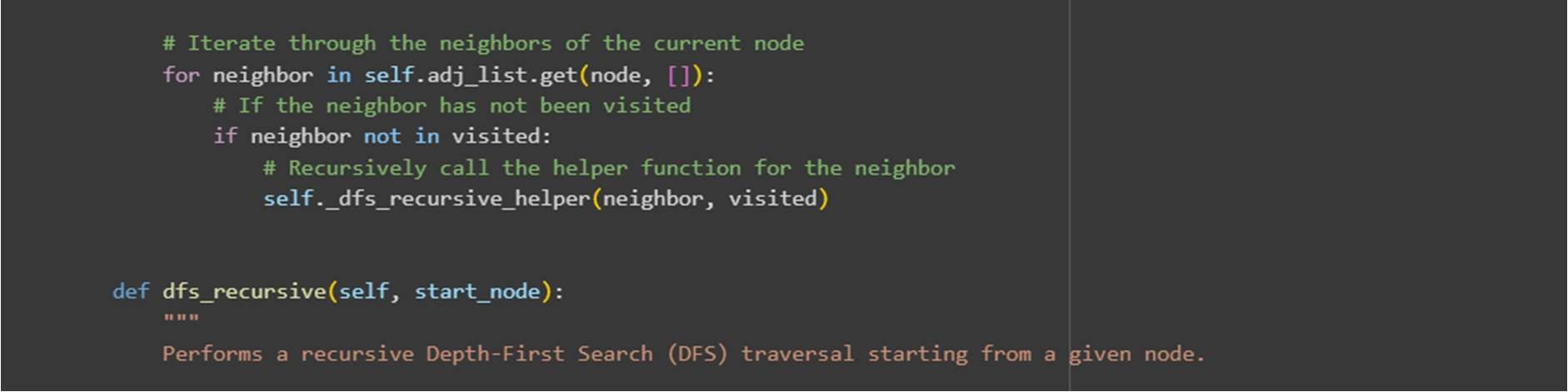
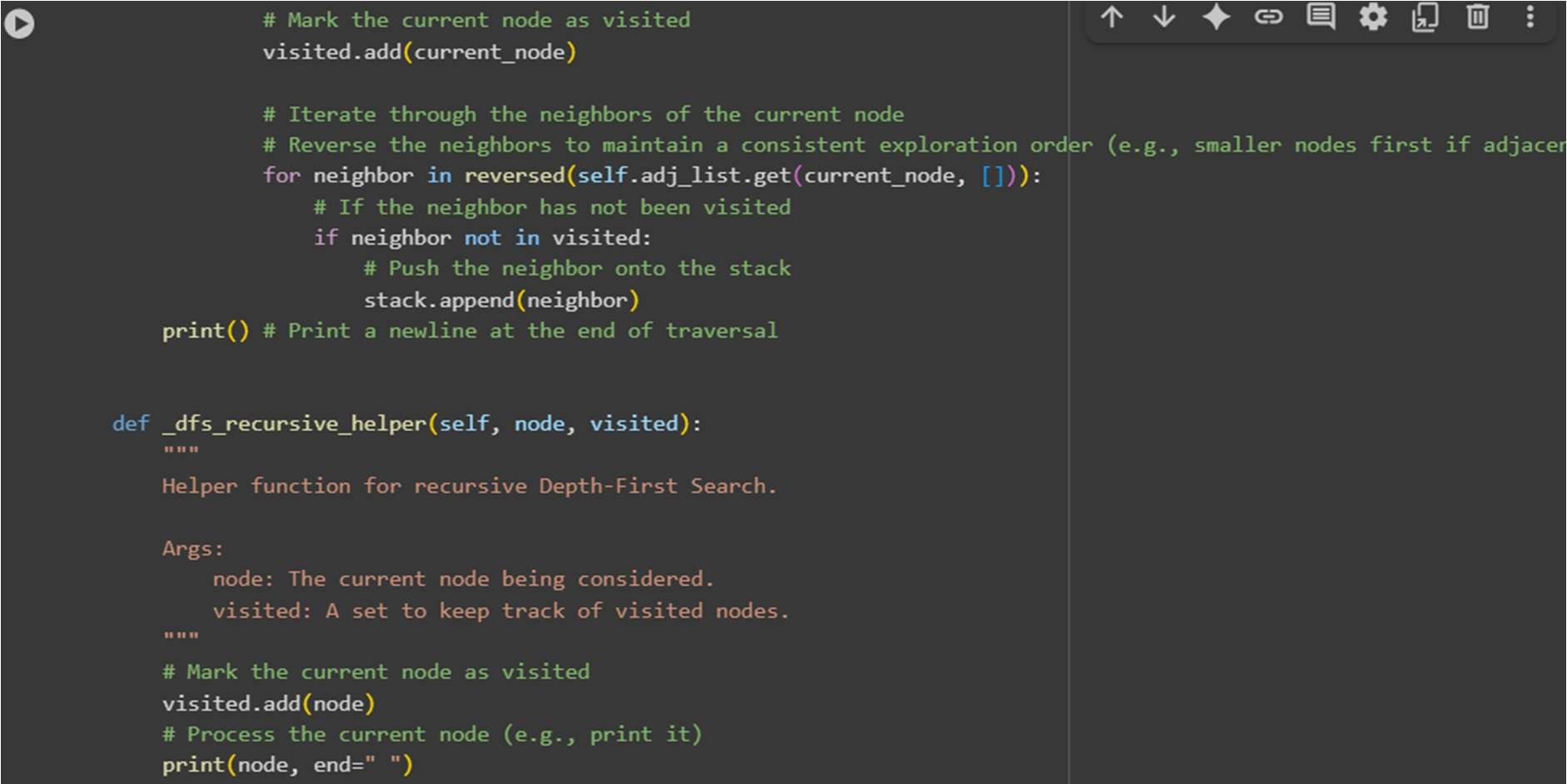
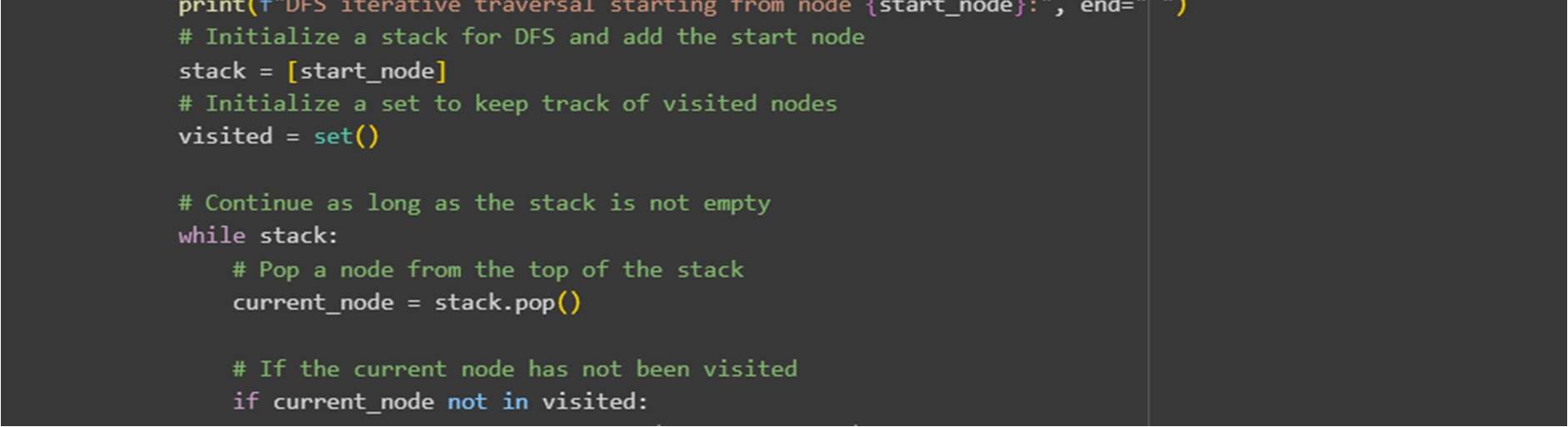
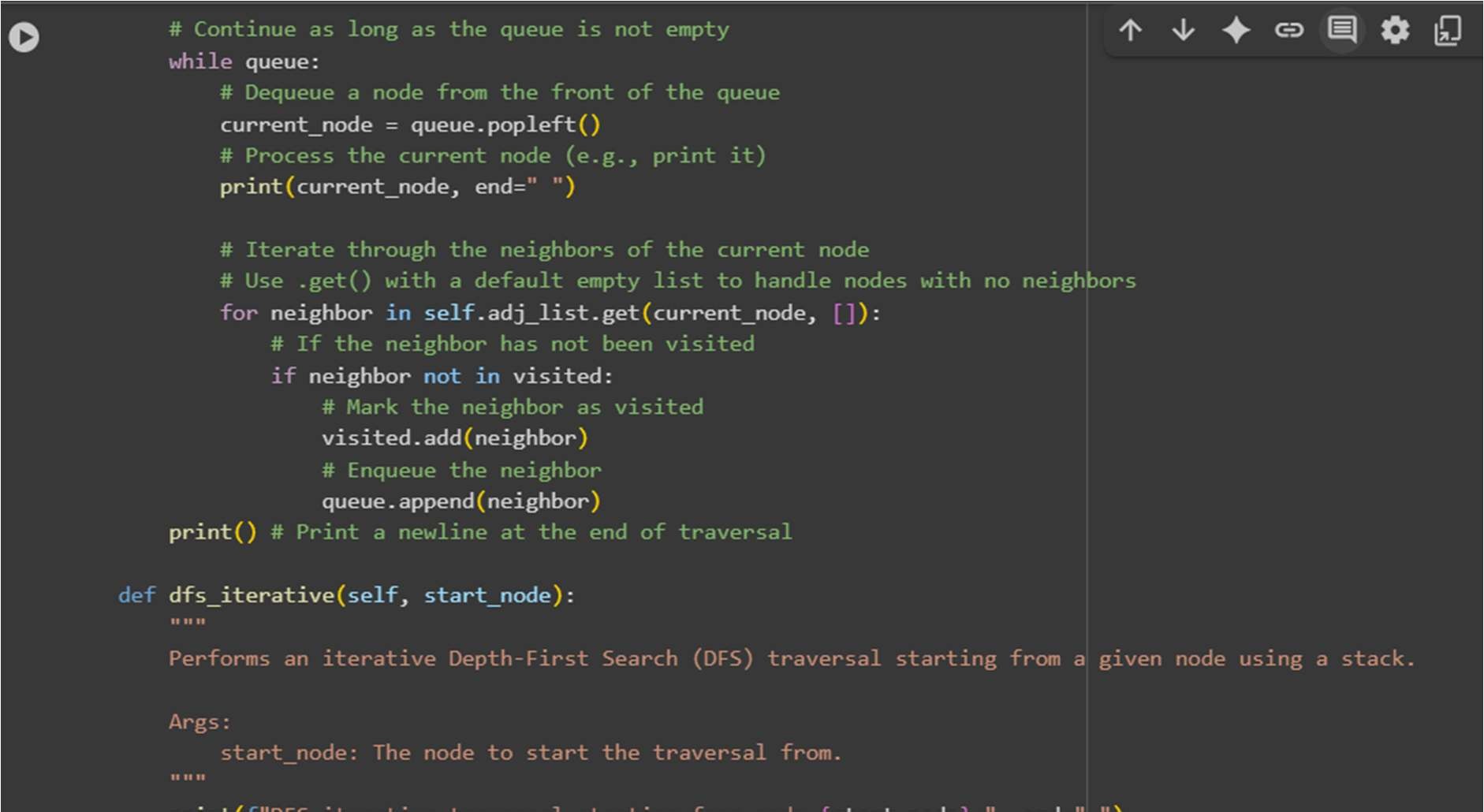
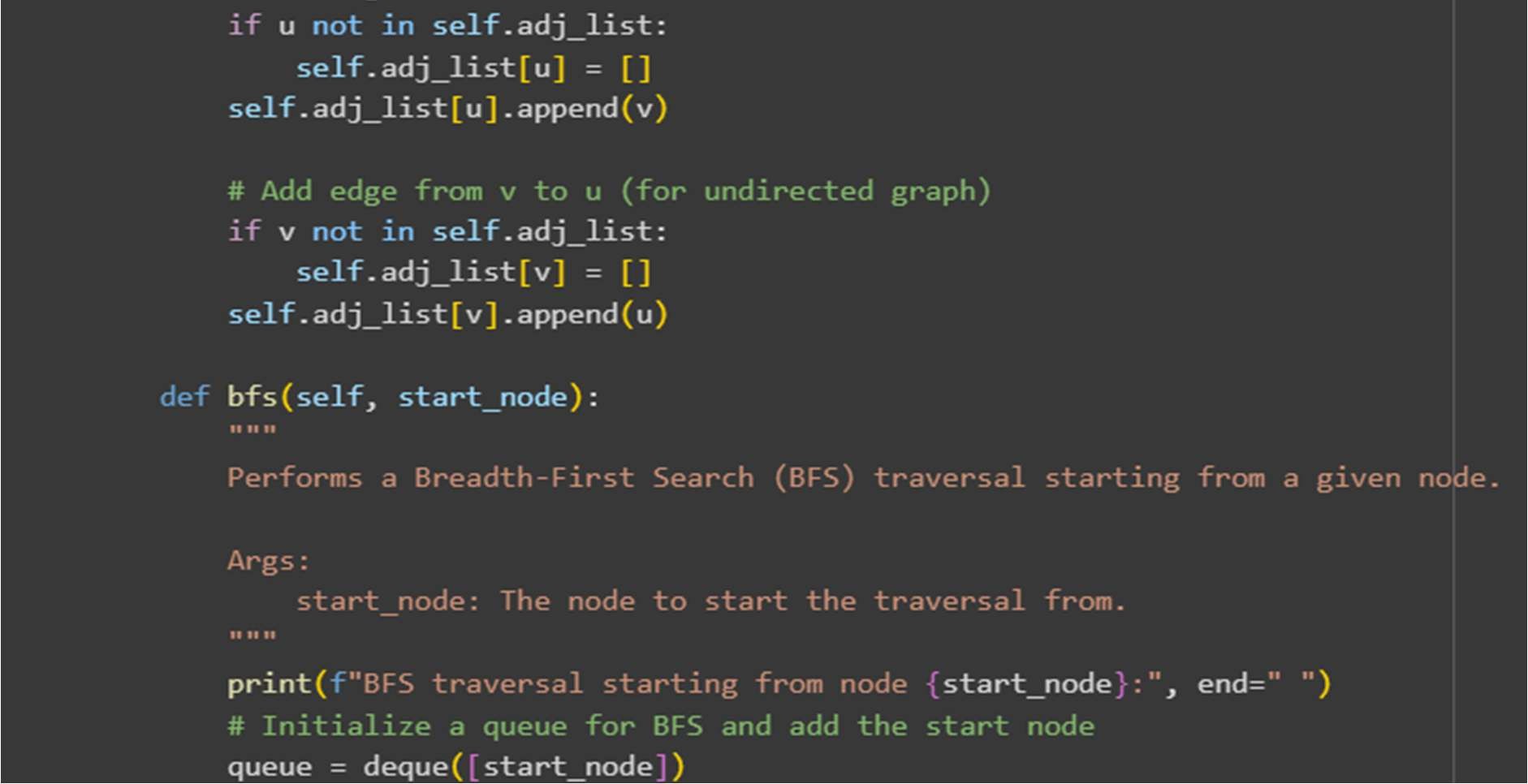
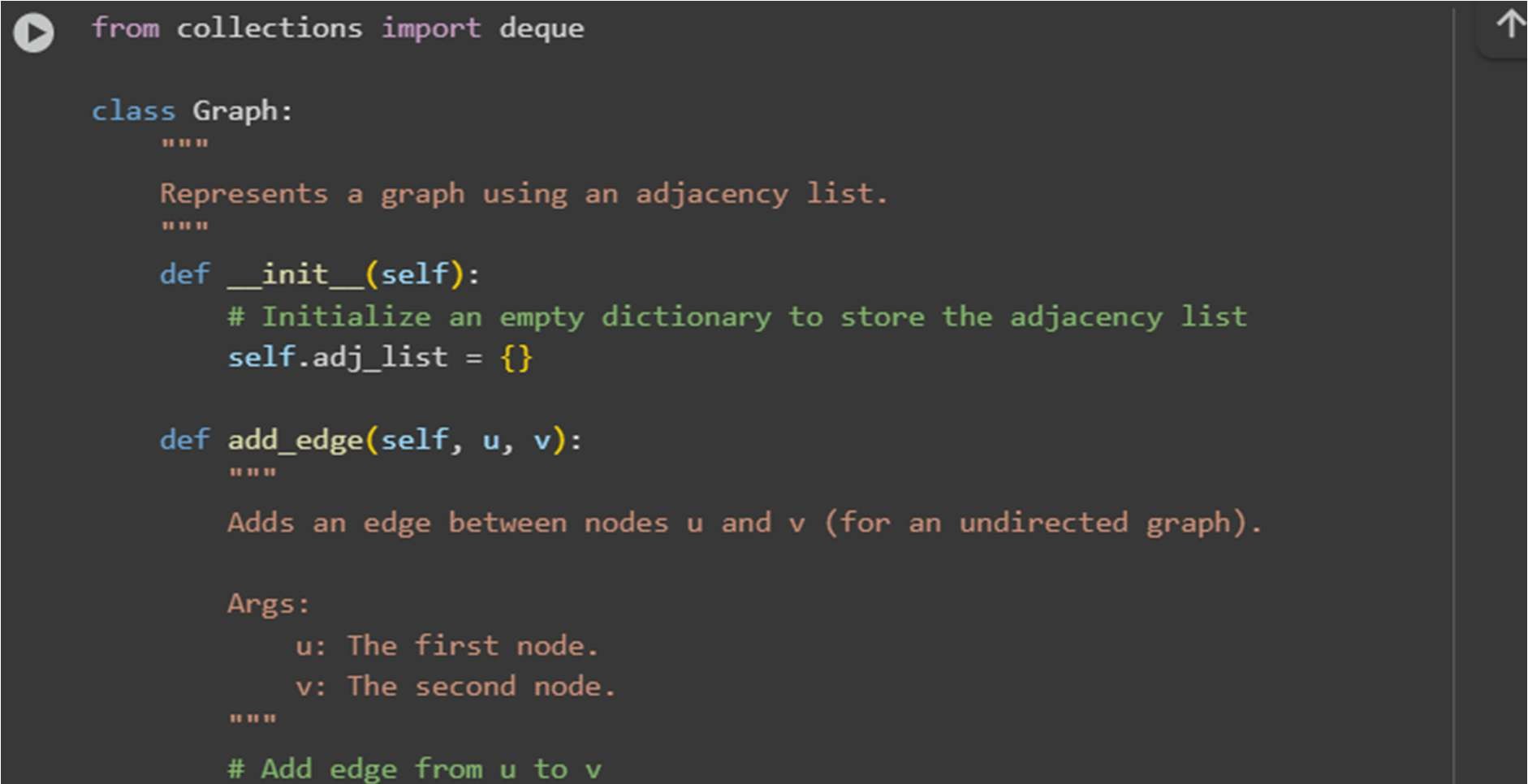
# Task 5: Graph Representation and BFS/DFS Traversal

* Task: Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().
* Instructions:

o Start with an adjacency list dictionary. o Ask AI to generate BFS and DFS implementations with inline comments. o Compare recursive vs iterative DFS if suggested by AI.

* Expected Output: o A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

Code:



Output:

