Experiment 1: Write a Python Program to Python program to implement List operations:

```
# Function to perform list operations
def list_operations():
# Initializing a sample list
my_list = [1, 2, 3, 4, 5]

# Displaying the original list
print("Original List:", my_list)

# Appending an element to the list
my_list.append(6)
print("List after appending 6:", my_list)

# Inserting an element at a specific position
my_list.insert(2, 10)
print("List after inserting 10 at index 2:", my_list)

# Removing an element by value
my_list.remove(4)
print("List after removing element 4:", my_list)

# Removing an element by index
popped_element = my_list.pop(1)
print("List after popping element at index 1:", my_list)
print("Popped element:", popped_element)

# Checking if an element exists in the list
if 3 in my_list:
print("3 is present in the list.")
else:
print("3 is not present in the list.")

# Sorting the list
my_list.sort()
print("Sorted List:", my_list)

# Reversing the list
my_list.reverse()
print("Reversed List:", my_list)

# Calling the list_operations function
list_operations()
```

Output:

Original List: [1, 2, 3, 4, 5]
List after appending 6: [1, 2, 3, 4, 5, 6]
List after inserting 10 at index 2: [1, 2, 10, 3, 4, 5, 6]
List after removing element 4: [1, 2, 10, 3, 5, 6]
List after popping element at index 1: [1, 10, 3, 5, 6]
Popped element: 2
3 is present in the list.
Sorted List: [1, 3, 5, 6, 10]
Reversed List: [10, 6, 5, 3, 1]

Experiment 2: Implementation of Nested List, Length, Concatenation, Membership, Iteration,Indexing and Slicing functions.

```
# Nested list

nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Length of the nested list

print("Length of the nested list:", len(nested_list))

# Concatenation of nested lists

concatenated_list = nested_list + [[10, 11, 12]]

print("Concatenated List:", concatenated_list)

# Membership check

if [4, 5, 6] in nested_list:

    print("[4, 5, 6] is present in the nested list.")

else:

    print("[4, 5, 6] is not present in the nested list.")

# Iterating through the elements of the nested list

print("Iterating through the elements:")

for sublist in nested_list:

    for element in sublist:

        print(element, end=" ")

    print()

# Indexing

print("Element at index [1][2]:", nested_list[1][2])

# Slicing

sliced_list = nested_list[0:2]

print("Sliced List:", sliced_list)
```

Output:

Length of the nested list: 3
Concatenated List: [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
[4, 5, 6] is present in the nested list.
Iterating through the elements:
1 2 3
4 5 6
7 8 9
Element at index [1][2]: 6
Sliced List: [[1, 2, 3], [4, 5, 6]]

Experiment 3: Implementation of List methods : Add, Append, Extend & Delete.

```
# Function to demonstrate list methods
def list_operations():
    # Initialize a list
    my_list = [1, 2, 3, 4, 5]
    # Display the original list
    print("Original List:", my_list)
    # Add elements using the '+' operator
    added_list = my_list + [6, 7]
    print("List after adding elements using '+':", added_list)
    # Append a single element
  my_list.append(8)
  print("List after appending 8:", my_list)
  # Append multiple elements using extend
  my_list.extend([9, 10])
  print("List after extending with [9, 10]:", my_list)
  # Delete elements using pop
  popped_element = my_list.pop(1)
  print("List after popping element at index 1:", my_list)
  print("Popped element:", popped_element)
  # Remove an element by value
  my_list.remove(4)
  print("List after removing element 4:", my_list)
```

Output:

Original List: [1, 2, 3, 4, 5]
List after adding elements using '+': [1, 2, 3, 4, 5, 6, 7]
List after appending 8: [1, 2, 3, 4, 5, 8]
List after extending with [9, 10]: [1, 2, 3, 4, 5, 8, 9, 10]
List after popping element at index 1: [1, 3, 4, 5, 8, 9, 10]
Popped element: 2
List after removing element 4: [1, 3, 5, 8, 9, 10]

Experiment 4: Program to eliminate punctuations in a given string.

```
import string

def eliminate_punctuations(input_string):
    # Define a string of punctuations
    punctuations = string.punctuation

    # Remove punctuations from the input string
    result_string = "".join(char for char in input_string if char not in punctuations)

    return result_string

# Example usage
input_string = "Hello, World! This is an example string with punctuations."
result = eliminate_punctuations(input_string)

print("Original String:", input_string)
print("String after eliminating punctuations:", result)
```

This program defines a function `eliminate_punctuations` that takes an input string and removes all punctuations using a list comprehension. The `string.punctuation` constant provides a string of all ASCII punctuation characters. The example usage demonstrates how to use the function on a given string.

Output:

Original String: Hello, World! This is an example string with punctuations.
String after eliminating punctuations: Hello World This is an example string with punctuations

Experiment 5: Sorting a sentence in an alphabetical order.

```
def sort_sentence_alphabetically(sentence):
    # Split the sentence into words
    words = sentence.split()

    # Sort the words alphabetically
    sorted_words = sorted(words)

    # Join the sorted words to form the sorted sentence
    sorted_sentence = ' '.join(sorted_words)

    return sorted_sentence

# Example usage
input_sentence = "This is a sample sentence to be sorted alphabetically."
result = sort_sentence_alphabetically(input_sentence)

print("Original Sentence:", input_sentence)
print("Sentence after sorting alphabetically:", result)
```

This program defines a function `sort_sentence_alphabetically` that takes a sentence as input, splits it into words, sorts the words alphabetically, and then joins them back into a sorted sentence. The example usage demonstrates how to use this function on a given sentence.

Output:

Original Sentence: This is a sample sentence to be sorted alphabetically.
Sentence after sorting alphabetically: This a alphabetically. be is sample sentence sorted to

Experiment 6: Implementation of Breadth First Search Traversal Technique.

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start_node):
        visited = set()
        queue = deque([start_node])

        while queue:
            current_node = queue.popleft()

            if current_node not in visited:
                print(current_node, end=" ")
                visited.add(current_node)

                for neighbor in self.graph[current_node]:
                    if neighbor not in visited:
                        queue.append(neighbor)

# Example usage
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)

start_node = 2
print("Breadth-First Search Traversal starting from node", start_node)
graph.bfs(start_node)
```

This program defines a simple graph class and demonstrates BFS traversal. You can modify the graph by adding edges as needed. The `bfs` method performs the BFS traversal starting from a specified node and prints the nodes visited in the traversal order.

Output:

```
# Example usage
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
# start_node = 2
```

Solution:

Breadth-First Search Traversal starting from node 2

2 0 3 1

Experiment 7: Depth First Search Traversal Technique Implementation.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs_util(self, node, visited):
        visited.add(node)
        print(node, end=" ")

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, start_node):
        visited = set()
        self.dfs_util(start_node, visited)

# Example usage
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
start_node = 2
print("Depth-First Search Traversal starting from node", start_node)
graph.dfs(start_node)
```

This program defines a simple graph class and demonstrates DFS traversal. You can modify the graph by adding edges as needed. The `dfs_util` method is a recursive utility function for DFS, and the `dfs` method initializes the traversal from a specified starting node. The nodes visited during the traversal are printed in the order they are visited.

Output:

```
# Example usage
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)

# start_node = 2
solution:Depth-First Search Traversal starting from node 2  2 0 1 3
```

Experiment 8: Implementation of Water Jug Problem using Heuristic Method.

The Water Jug Problem involves finding a sequence of steps to measure a specific amount of water using two jugs of known capacities. Here's a Python program implementing the Water Jug Problem using a heuristic method, specifically the A* algorithm:

```
import heapq

class State:
    def __init__(self, jug1, jug2, parent=None, action=None, cost=0, heuristic=0):
        self.jug1 = jug1
        self.jug2 = jug2
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def water_jug_heuristic(jug1_capacity, jug2_capacity, target_amount):
    start_state = State(0, 0)
    goal_state = State(target_amount, 0)

    visited_states = set()
    priority_queue = [start_state]

    while priority_queue:
        current_state = heapq.heappop(priority_queue)

        if (current_state.jug1, current_state.jug2) == (goal_state.jug1, goal_state.jug2):
            return construct_path(current_state)

        visited_states.add((current_state.jug1, current_state.jug2))

        next_states = generate_next_states(current_state, jug1_capacity, jug2_capacity)

        for state in next_states:
            if (state.jug1, state.jug2) not in visited_states:
                heapq.heappush(priority_queue, state)

    return None

def generate_next_states(current_state, jug1_capacity, jug2_capacity):
    next_states = []

    # Fill jug1
    next_states.append(State(jug1_capacity, current_state.jug2, current_state, "Fill Jug 1"))

    # Fill jug2
    next_states.append(State(current_state.jug1, jug2_capacity, current_state, "Fill Jug 2"))
```

```
    # Empty jug1
    next_states.append(State(0, current_state.jug2, current_state, "Empty Jug 1"))

    # Empty jug2
    next_states.append(State(current_state.jug1, 0, current_state, "Empty Jug 2"))

    # Pour from jug1 to jug2
    pour_amount = min(current_state.jug1, jug2_capacity - current_state.jug2)
    next_states.append(State(current_state.jug1 - pour_amount, current_state.jug2 + pour_amount, current_state, f"Pour Jug 1 to Jug 2 ({pour_amount} units)"))

    # Pour from jug2 to jug1
    pour_amount = min(current_state.jug2, jug1_capacity - current_state.jug1)
    next_states.append(State(current_state.jug1 + pour_amount, current_state.jug2 - pour_amount, current_state, f"Pour Jug 2 to Jug 1 ({pour_amount} units)"))

    return next_states

def construct_path(final_state):
    path = []
    current_state = final_state
    while current_state:
        path.append((current_state.jug1, current_state.jug2, current_state.action))
        current_state = current_state.parent
    return reversed(path)

def main():
    jug1_capacity = 4
    jug2_capacity = 3
    target_amount = 2

    result = water_jug_heuristic(jug1_capacity, jug2_capacity, target_amount)

    if result:
        print(f"Solution found to measure {target_amount} units of water:")
        for step in result:
            print(f"{step[2]} -> Jug 1: {step[0]}, Jug 2: {step[1]}")
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

This program uses the A* algorithm to find a solution to the Water Jug Problem. It defines a `State` class to represent the state of the jugs, and the heuristic function guides the search towards the goal state. The `generate_next_states` function generates possible next states, and the `construct_path` function constructs the solution path. The `main` function sets up the problem parameters and calls the heuristic algorithm to find a solution
.

Output:

Solution to Water Jug Problem:
Solution found to measure 2 units of water:
None -> Jug 1: 0, Jug 2: 0
Fill Jug 2 -> Jug 1: 0, Jug 2: 3
Pour Jug 2 to Jug 1 (3 units) -> Jug 1: 3, Jug 2: 0
Fill Jug 2 -> Jug 1: 3, Jug 2: 3
Pour Jug 2 to Jug 1 (1 units) -> Jug 1: 4, Jug 2: 2
Empty Jug 1 -> Jug 1: 0, Jug 2: 2
Pour Jug 2 to Jug 1 (2 units) -> Jug 1: 2, Jug 2: 0