AI ASSISTANT CODING ASSIGNMENT -12.2

NAME : MOGILI KEERTHI

ROLL NO : 2303A51427

BATCH NO : 21

Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Lab Objectives:

• Apply AI-assisted programming to implement and optimize sorting and searching algorithms.

• Compare different algorithms in terms of efficiency and use cases.

• Understand how AI tools can suggest optimized code and complexity improvements.

Task Description -1 (Data Structures – Stack Implementation with AI Assistance)

➢ Task: Use AI assistance to generate a Python program that implements a Stack data structure.

Instructions:

Prompt AI to create a Stack class with the following methods:

➢ push(element)

➢ pop()

➢ peek()

➢ is_empty()

➢ Ensure proper error handling for stack underflow.

➢ Ask AI to include clear docstrings for each method.

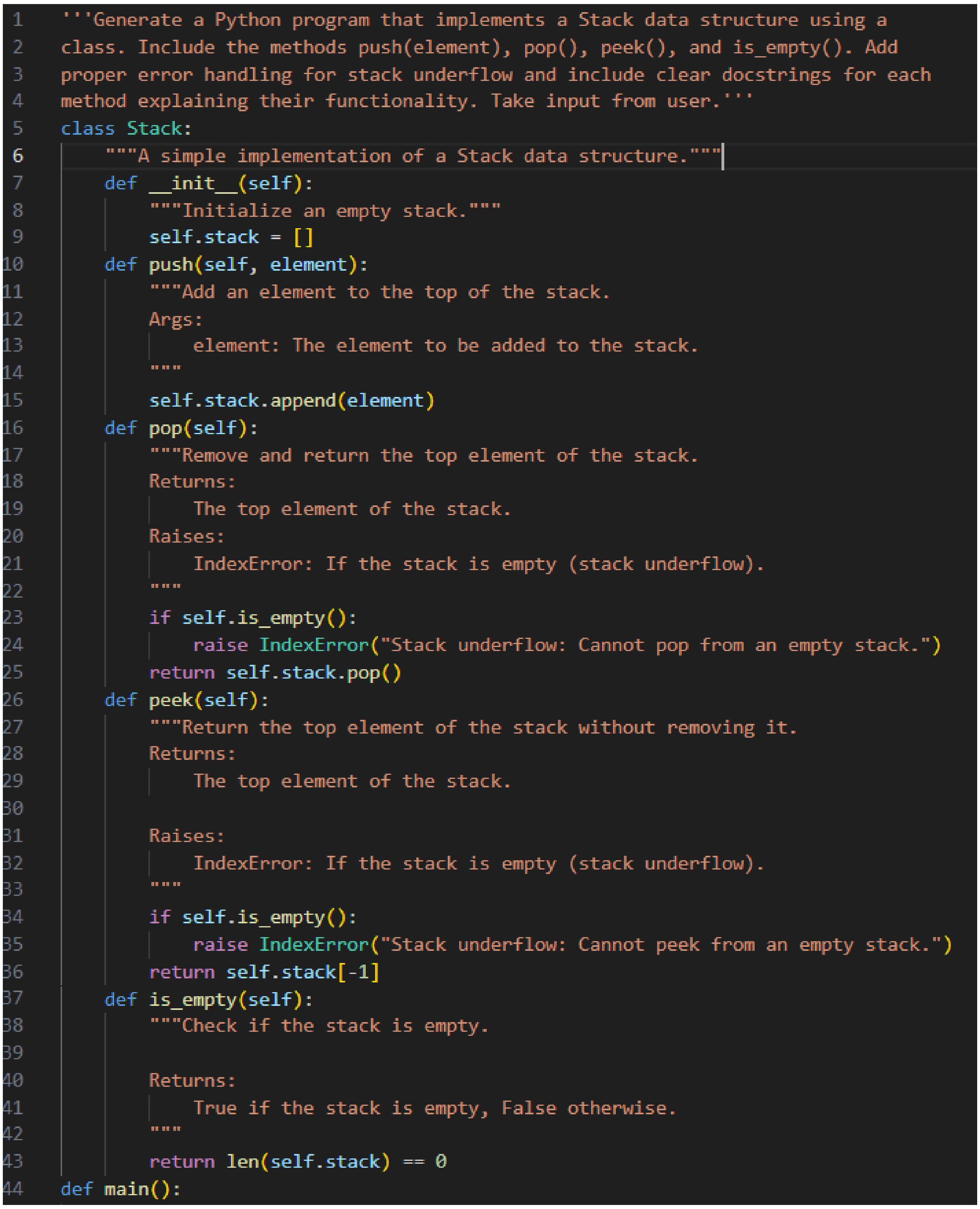
Expected Output:➢ A functional Python program implementing a Stack using a class.

➢ Properly documented methods with docstrings.

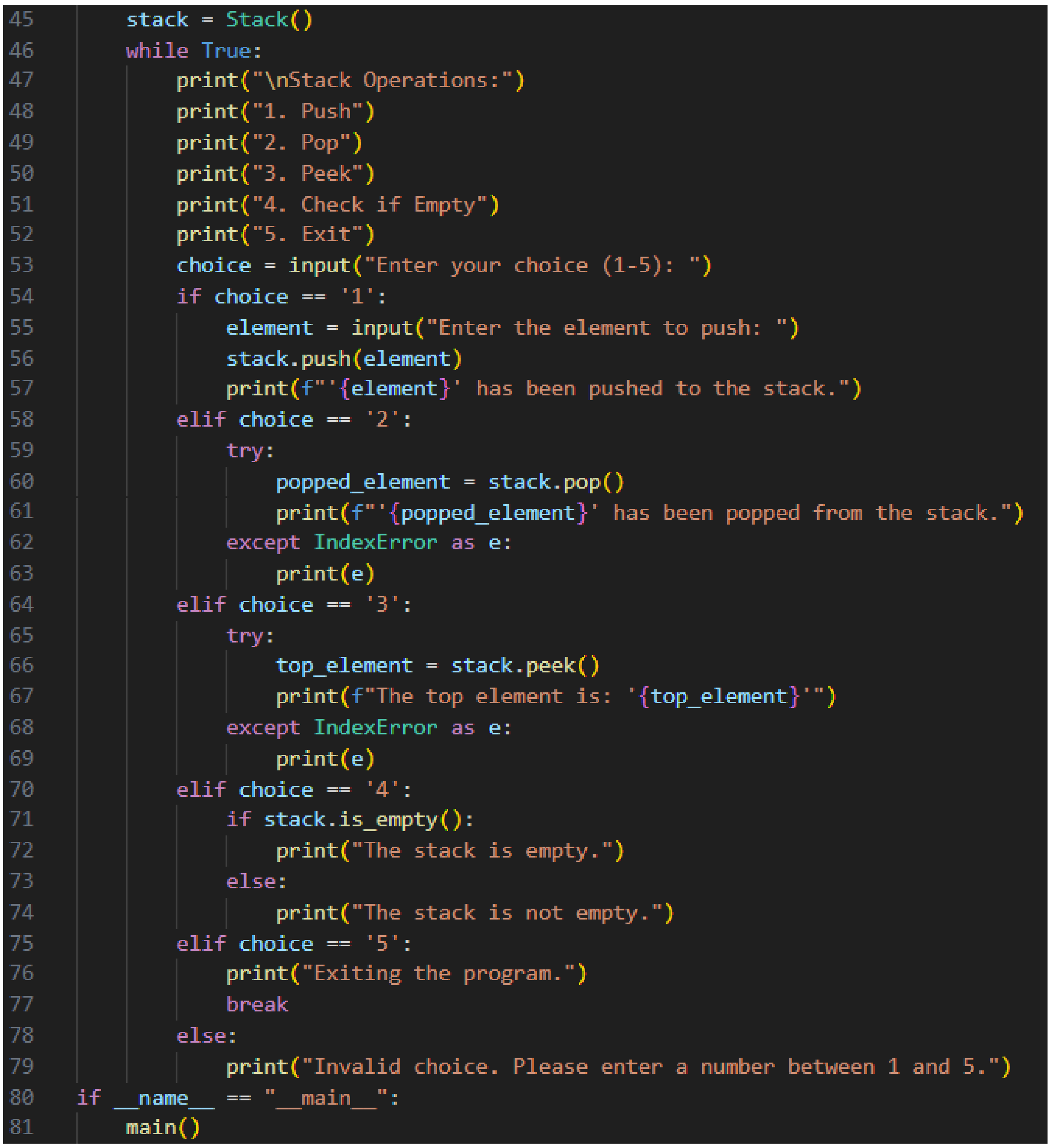
Prompt:

Generate a Python program that implements a Stack data structure using a class. Include the methods push(element), pop(), peek(), and is_empty(). Add proper error handling for stack underflow and include clear docstrings for each method explaining their functionality. Take input from user.

Code:

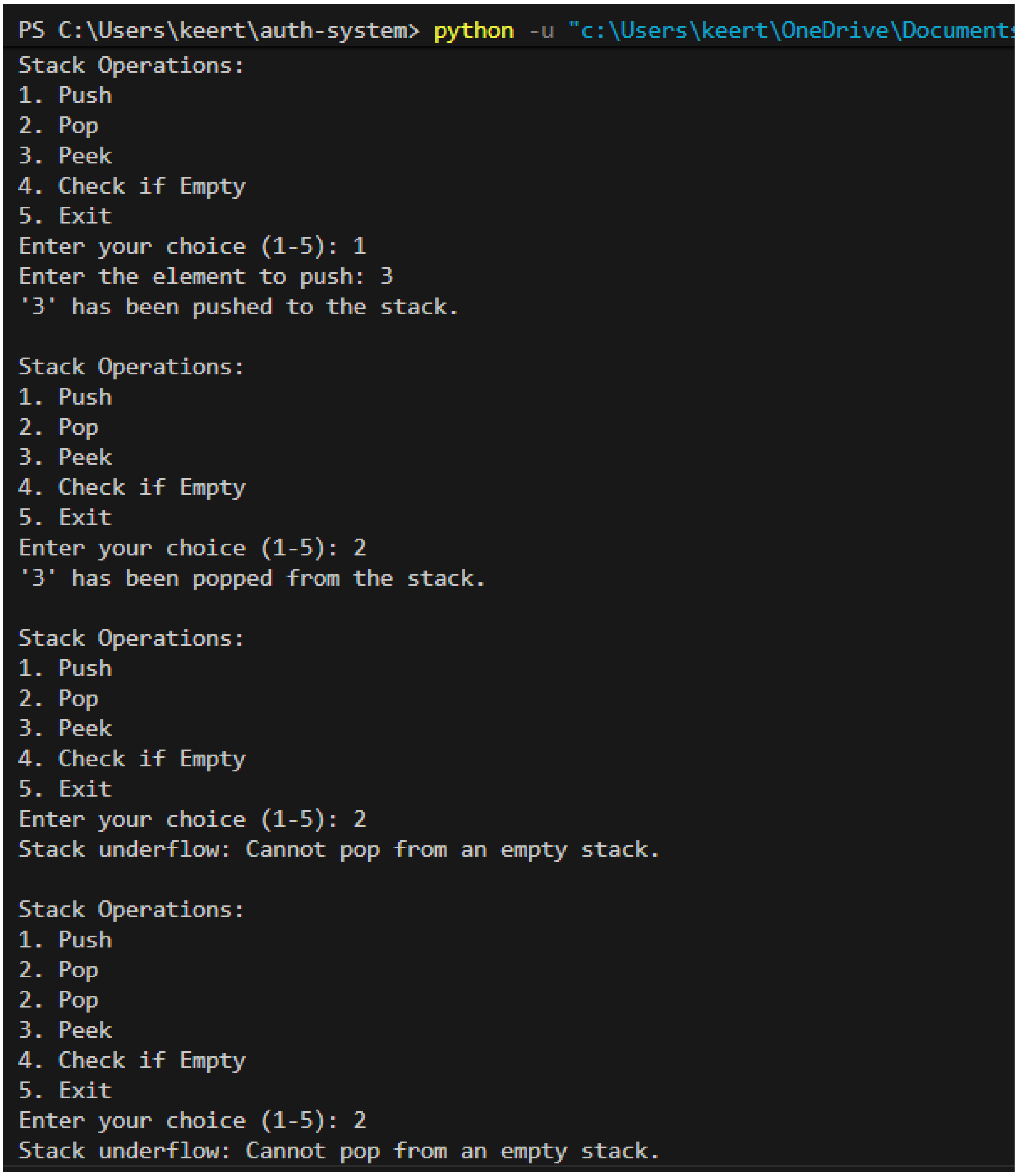


```python
'''Generate a Python program that implements a Stack data structure using a
class. Include the methods push(element), pop(), peek(), and is_empty(). Add
proper error handling for stack underflow and include clear docstrings for each
method explaining their functionality. Take input from user.'''
class Stack:
    """A simple implementation of a Stack data structure."""
    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []
    def push(self, element):
        """Add an element to the top of the stack.
        Args:
            element: The element to be added to the stack.
        """
        self.stack.append(element)
    def pop(self):
        """Remove and return the top element of the stack.
        Returns:
            The top element of the stack.
        Raises:
            IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot pop from an empty stack.")
        return self.stack.pop()
    def peek(self):
        """Return the top element of the stack without removing it.
        Returns:
            The top element of the stack.

        Raises:
            IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot peek from an empty stack.")
        return self.stack[-1]
    def is_empty(self):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
def main():
```



```python
    stack = Stack()
    while True:
        print("\nStack Operations:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if Empty")
        print("5. Exit")
        choice = input("Enter your choice (1-5): ")
        if choice == '1':
            element = input("Enter the element to push: ")
            stack.push(element)
            print(f"'{element}' has been pushed to the stack.")
        elif choice == '2':
            try:
                popped_element = stack.pop()
                print(f"'{popped_element}' has been popped from the stack.")
            except IndexError as e:
                print(e)
        elif choice == '3':
            try:
                top_element = stack.peek()
                print(f"The top element is: '{top_element}'")
            except IndexError as e:
                print(e)
        elif choice == '4':
            if stack.is_empty():
                print("The stack is empty.")
            else:
                print("The stack is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
if __name__ == "__main__":
    main()
```

Output:



```
PS C:\Users\keert\auth-system> python -u "c:\Users\keert\OneDrive\Document
Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 1
Enter the element to push: 3
'3' has been pushed to the stack.

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
'3' has been popped from the stack.

Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
Stack underflow: Cannot pop from an empty stack.

Stack Operations:
1. Push
2. Pop
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
Stack underflow: Cannot pop from an empty stack.
```

Observation

A Stack is used in this implementation because it strictly follows the Last-In-First-Out (LIFO) principle, where the most recently added element is the first one to be removed. This behavior is ideal for scenarios such as expression evaluation, undo operations, and function call management. The stack supports efficient insertion and deletion through the push and pop operations, both of which run in constant time O(1). Other data structures like queues or general lists do not maintain this LIFO order, making them unsuitable for tasks that require accessing the latest element first.

Task Description -2 (Algorithms – Linear vs Binary Search Analysis)

➢ Task: Use AI to implement and compare Linear Search and Binary Search algorithms in Python.

Instructions:

➢ Prompt AI to generate:

➢ linear_search(arr, target)

➢ binary_search(arr, target)

➢ Include docstrings explaining:

➢ Working principle

➢ Test both algorithms using different input sizes.

Expected Output:

• Python implementations of both search algorithms.

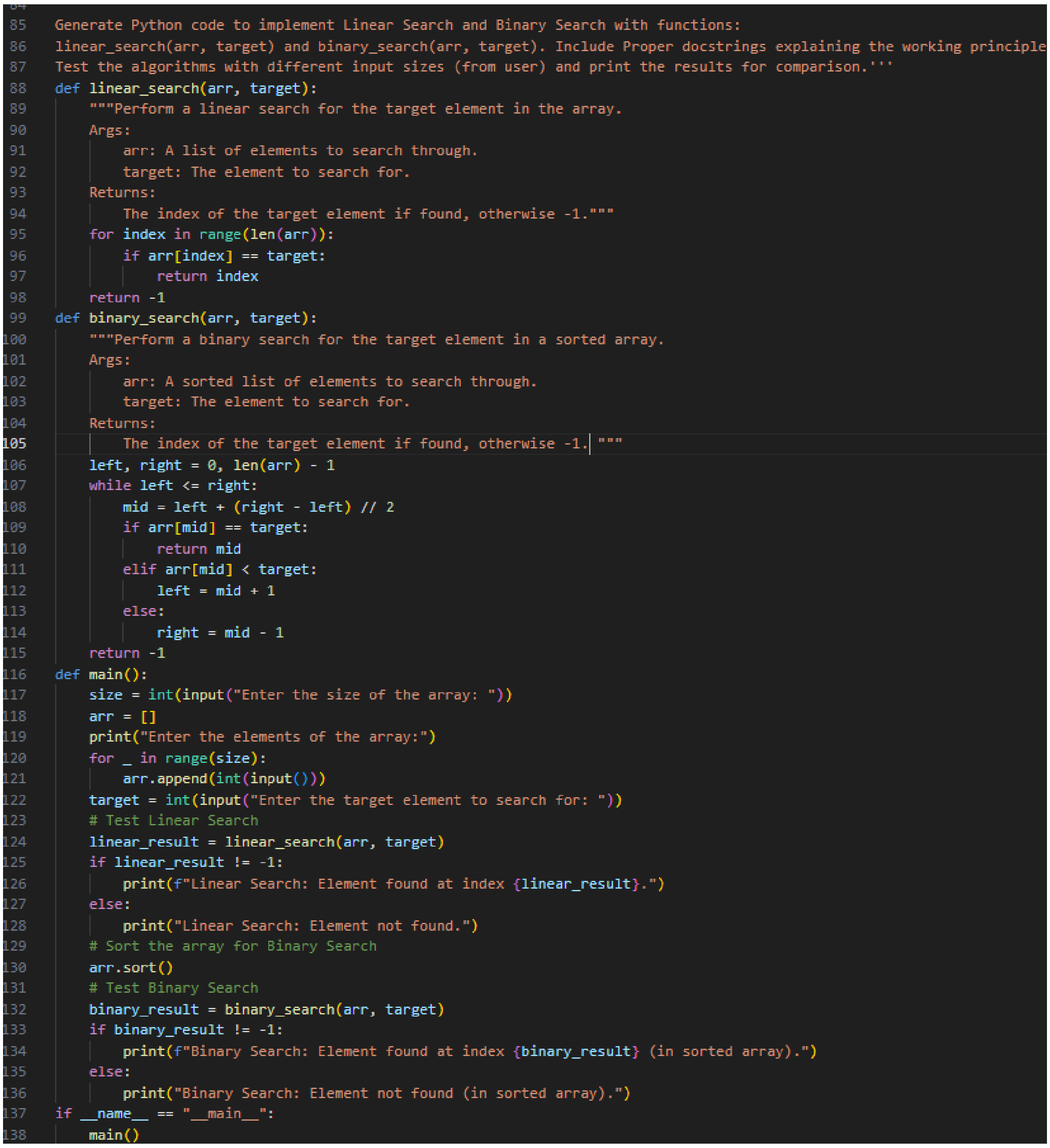
• AI-generated comments and complexity analysis.

• Test results showing correctness and comparison.

Prompt:

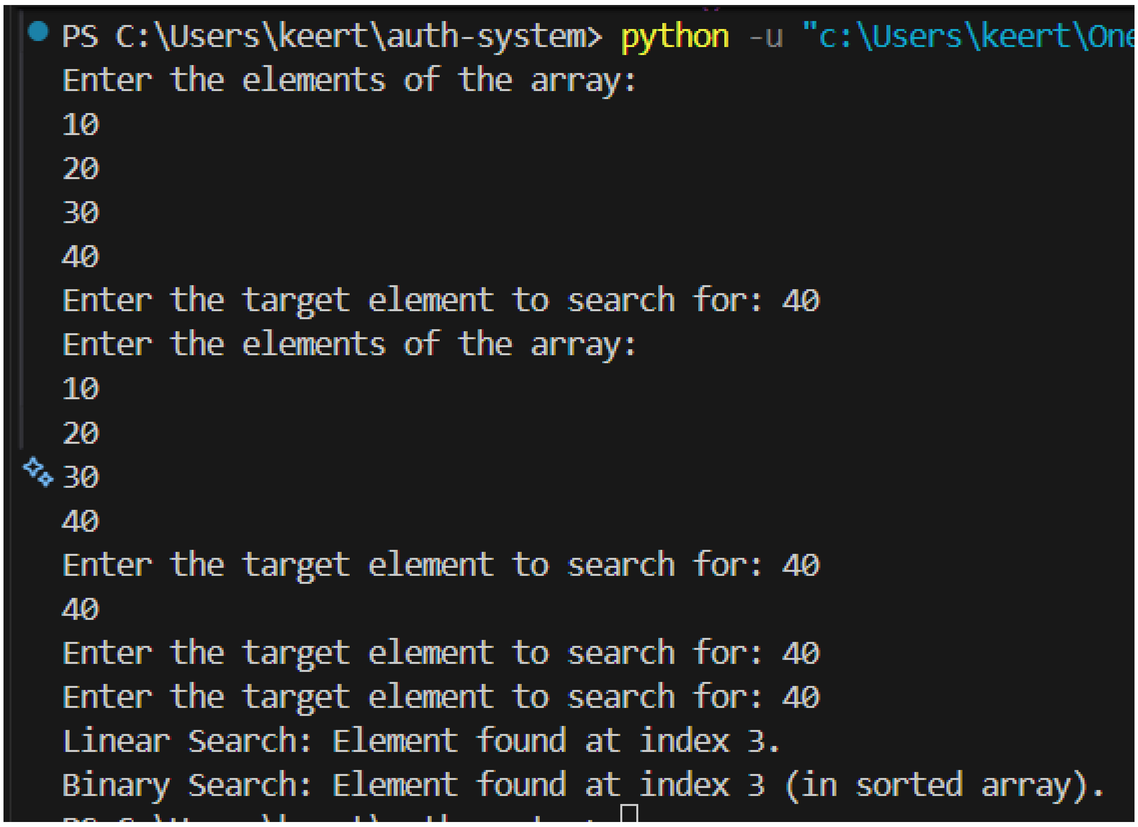
Generate Python code to implement Linear Search and Binary Search with functions: linear_search(arr, target) and binary_search(arr, target). Include Proper docstrings explaining the working principle. Test the algorithms with different input sizes (from user) and print the results for comparison.

Code:



```python
Generate Python code to implement Linear Search and Binary Search with functions:
linear_search(arr, target) and binary_search(arr, target). Include Proper docstrings explaining the working principle
Test the algorithms with different input sizes (from user) and print the results for comparison.'''
def linear_search(arr, target):
    """Perform a linear search for the target element in the array.
    Args:
        arr: A list of elements to search through.
        target: The element to search for.
    Returns:
        The index of the target element if found, otherwise -1."""
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
def binary_search(arr, target):
    """Perform a binary search for the target element in a sorted array.
    Args:
        arr: A sorted list of elements to search through.
        target: The element to search for.
    Returns:
        The index of the target element if found, otherwise -1. """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
def main():
    size = int(input("Enter the size of the array: "))
    arr = []
    print("Enter the elements of the array:")
    for _ in range(size):
        arr.append(int(input()))
    target = int(input("Enter the target element to search for: "))
    # Test Linear Search
    linear_result = linear_search(arr, target)
    if linear_result != -1:
        print(f"Linear Search: Element found at index {linear_result}.")
    else:
        print("Linear Search: Element not found.")
    # Sort the array for Binary Search
    arr.sort()
    # Test Binary Search
    binary_result = binary_search(arr, target)
    if binary_result != -1:
        print(f"Binary Search: Element found at index {binary_result} (in sorted array).")
    else:
        print("Binary Search: Element not found (in sorted array).")
if __name__ == "__main__":
    main()
```

Output:



```
PS C:\Users\keert\auth-system> python -u "c:\Users\keert\One
Enter the elements of the array:
10
20
30
40
Enter the target element to search for: 40
Enter the elements of the array:
10
20
30
40
Enter the target element to search for: 40
40
Enter the target element to search for: 40
Enter the target element to search for: 40
Linear Search: Element found at index 3.
Binary Search: Element found at index 3 (in sorted array).
```

Observation

Linear Search and Binary Search use the Array/List data structure because it provides direct and efficient access to elements through indexed positions. Linear Search benefits from the ability to scan elements sequentially, while Binary Search specifically requires a sorted array to repeatedly divide the search space and locate an element in logarithmic time. Other data structures, such as linked lists or trees, are not suitable for Binary Search because they do not support constant-time random indexing needed for efficient midpoint access.

Task Description -3 (Test Driven Development – Simple Calculator Function)

➢ Task:

Apply Test Driven Development (TDD) using AI assistance to develop a calculator function.

Instructions:

➢ Prompt AI to first generate unit test cases for addition and subtraction.

➢ Run the tests and observe failures.

➢ Ask AI to implement the calculator functions to pass all tests.

➢ Re-run the tests to confirm success.

Expected Output:

➢ Separate test file and implementation file.

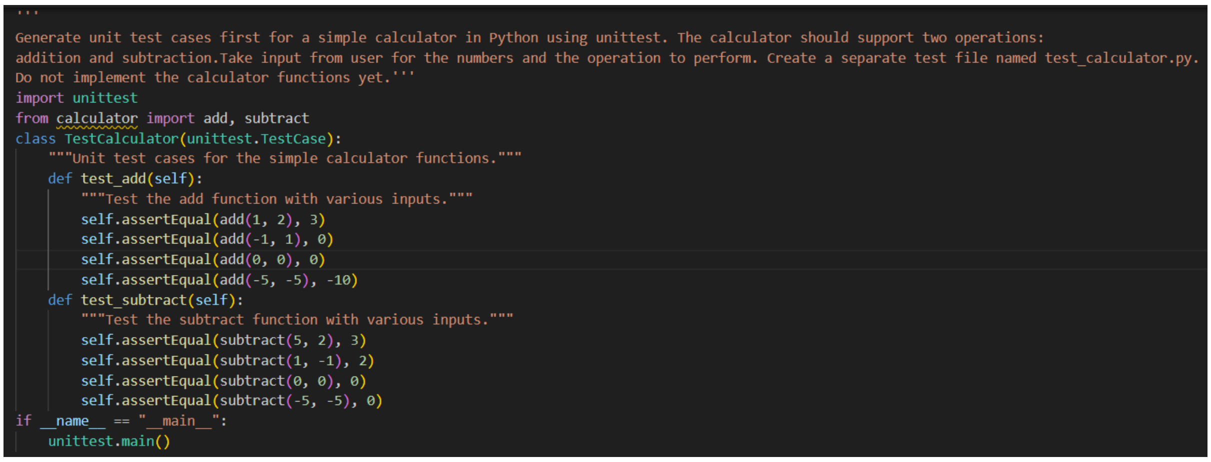
➢ Test cases executed before implementation.

➢ Final implementation passing all test cases.

Prompt:

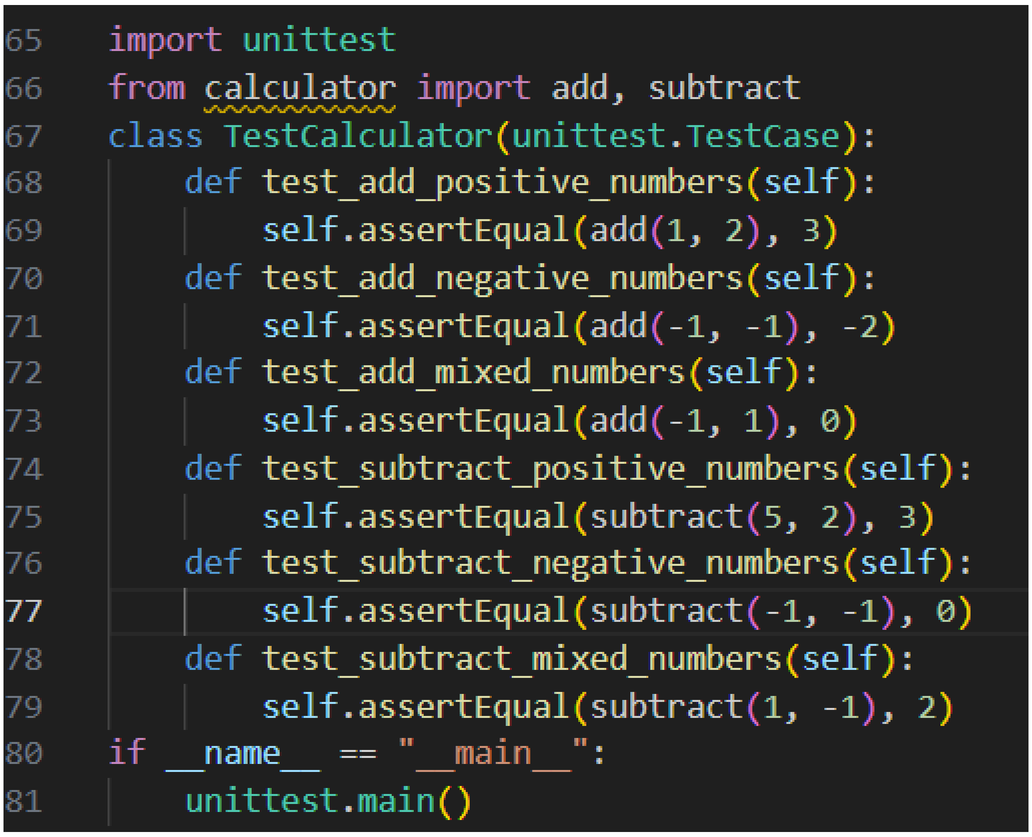
Generate unit test cases first for a simple calculator in Python using unittest. The calculator should support two operations: addition and subtraction.Take input from user for the numbers and the operation to perform. Create a separate test file named test_calculator.py. Do not implement the calculator functions yet.

Code:



```python
'''
Generate unit test cases first for a simple calculator in Python using unittest. The calculator should support two operations:
addition and subtraction.Take input from user for the numbers and the operation to perform. Create a separate test file named test_calculator.py.
Do not implement the calculator functions yet.'''
import unittest
from calculator import add, subtract
class TestCalculator(unittest.TestCase):
    """Unit test cases for the simple calculator functions."""
    def test_add(self):
        """Test the add function with various inputs."""
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)
        self.assertEqual(add(-5, -5), -10)
    def test_subtract(self):
        """Test the subtract function with various inputs."""
        self.assertEqual(subtract(5, 2), 3)
        self.assertEqual(subtract(1, -1), 2)
        self.assertEqual(subtract(0, 0), 0)
        self.assertEqual(subtract(-5, -5), 0)
if __name__ == "__main__":
    unittest.main()
```

Now implement the calculator functions add() and subtract() in calculator.py so that all the existing unit tests pass.



```python
import unittest
from calculator import add, subtract
class TestCalculator(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(add(1, 2), 3)
    def test_add_negative_numbers(self):
        self.assertEqual(add(-1, -1), -2)
    def test_add_mixed_numbers(self):
        self.assertEqual(add(-1, 1), 0)
    def test_subtract_positive_numbers(self):
        self.assertEqual(subtract(5, 2), 3)
    def test_subtract_negative_numbers(self):
        self.assertEqual(subtract(-1, -1), 0)
    def test_subtract_mixed_numbers(self):
        self.assertEqual(subtract(1, -1), 2)
if __name__ == "__main__":
    unittest.main()
```

Observation:

The Test-Driven Development (TDD) ensures that test cases are created before writing the actual program logic. This approach helps validate that the implementation meets the defined requirements and significantly reduces bugs during development. The use of AI-assisted tools such as GitHub Copilot further accelerates both test creation and code implementation while preserving code quality and correctness.

Task Description -4 (Data Structures – Queue Implementation with AI Assistance)

➢ Task:

Use AI assistance to generate a Python program that implements a Queue data structure.

Instructions:

➢ Prompt AI to create a Queue class with the following methods:

• enqueue(element)

• dequeue()

• front()

• is_empty()

➢ Handle queue overflow and underflow conditions.

➢ Include appropriate docstrings for all methods.

Expected Output:

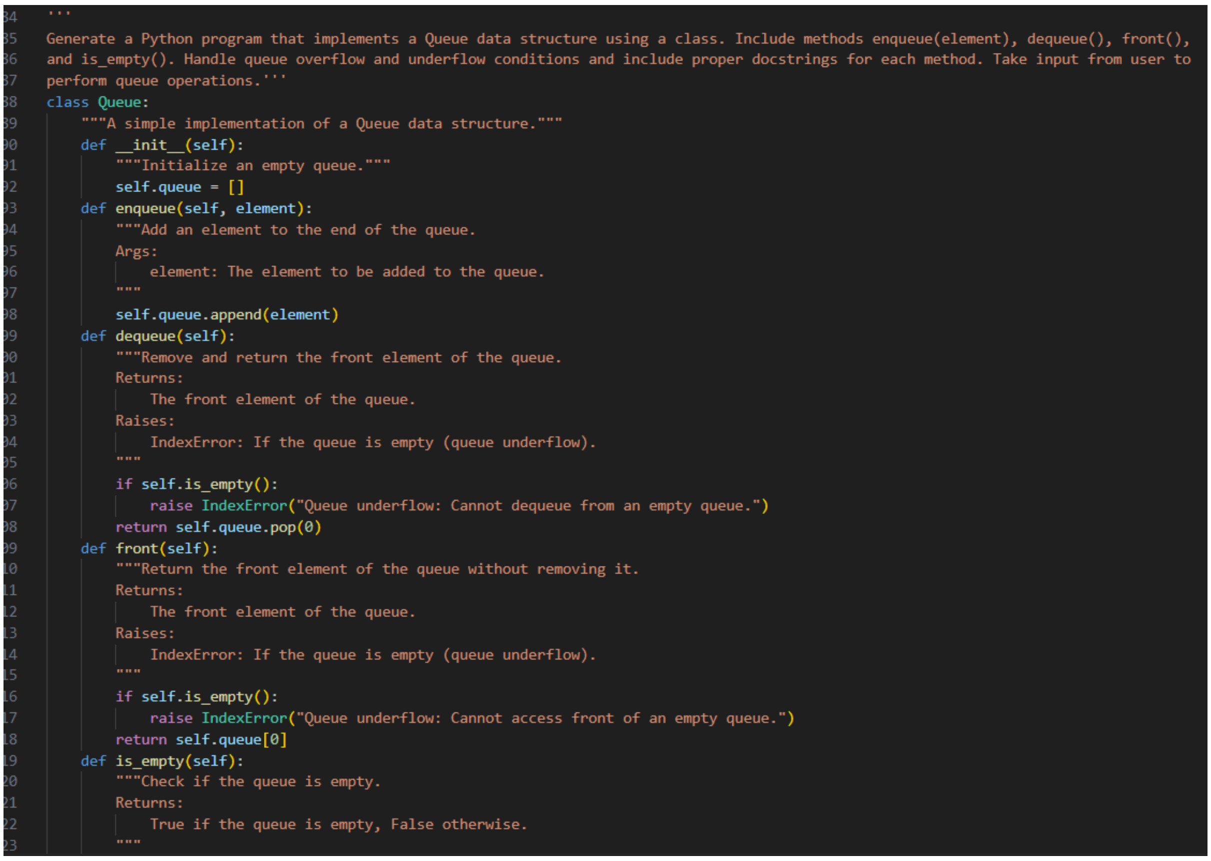
➢ A fully functional Queue implementation in Python.

➢ Proper error handling and documentation.

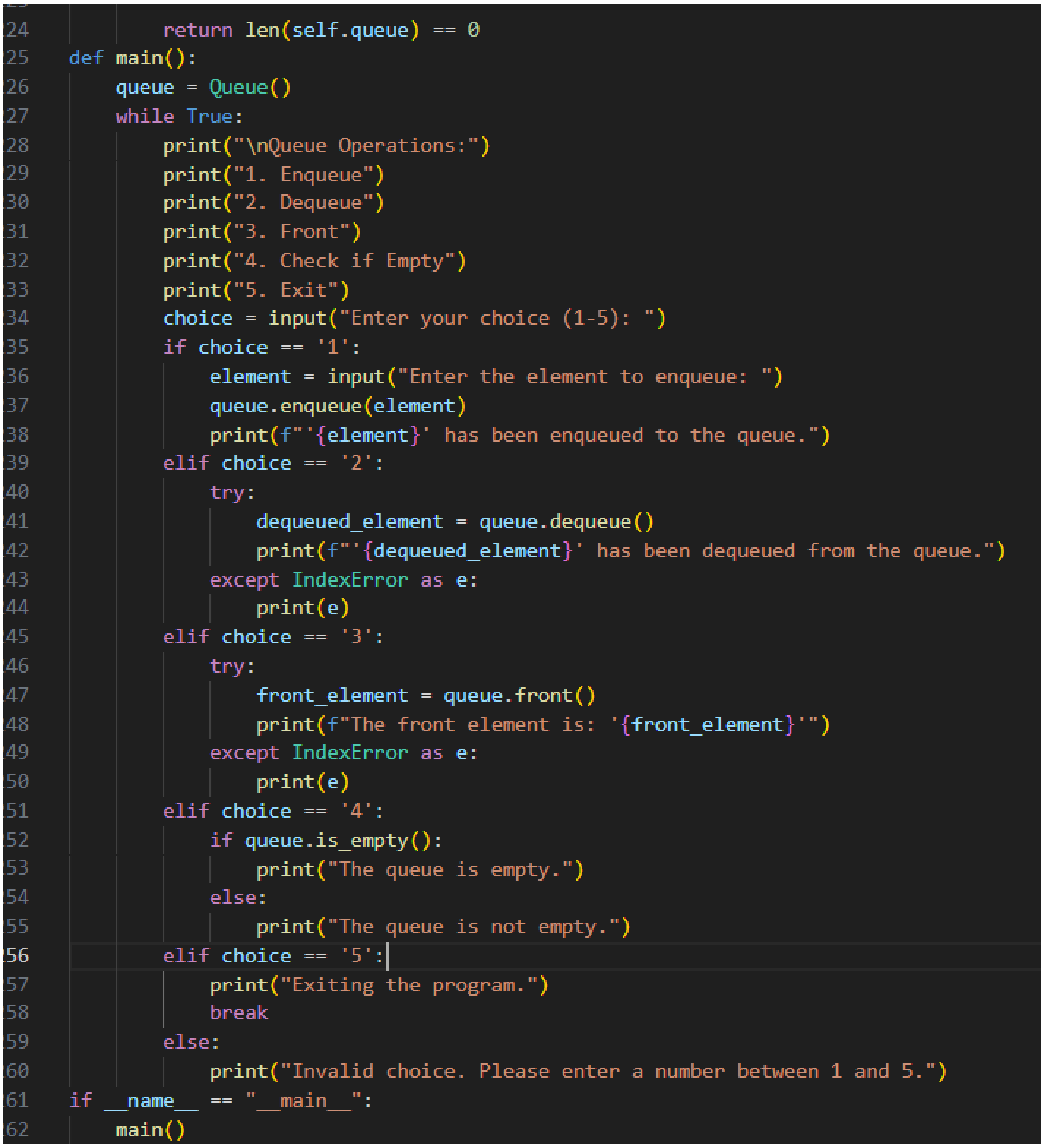
Prompt:

Generate a Python program that implements a Queue data structure using a class. Include methods enqueue(element), dequeue(), front(), and is_empty(). Handle queue overflow and underflow conditions and include proper docstrings for each method. Take input from user to perform queue operations.

Code:

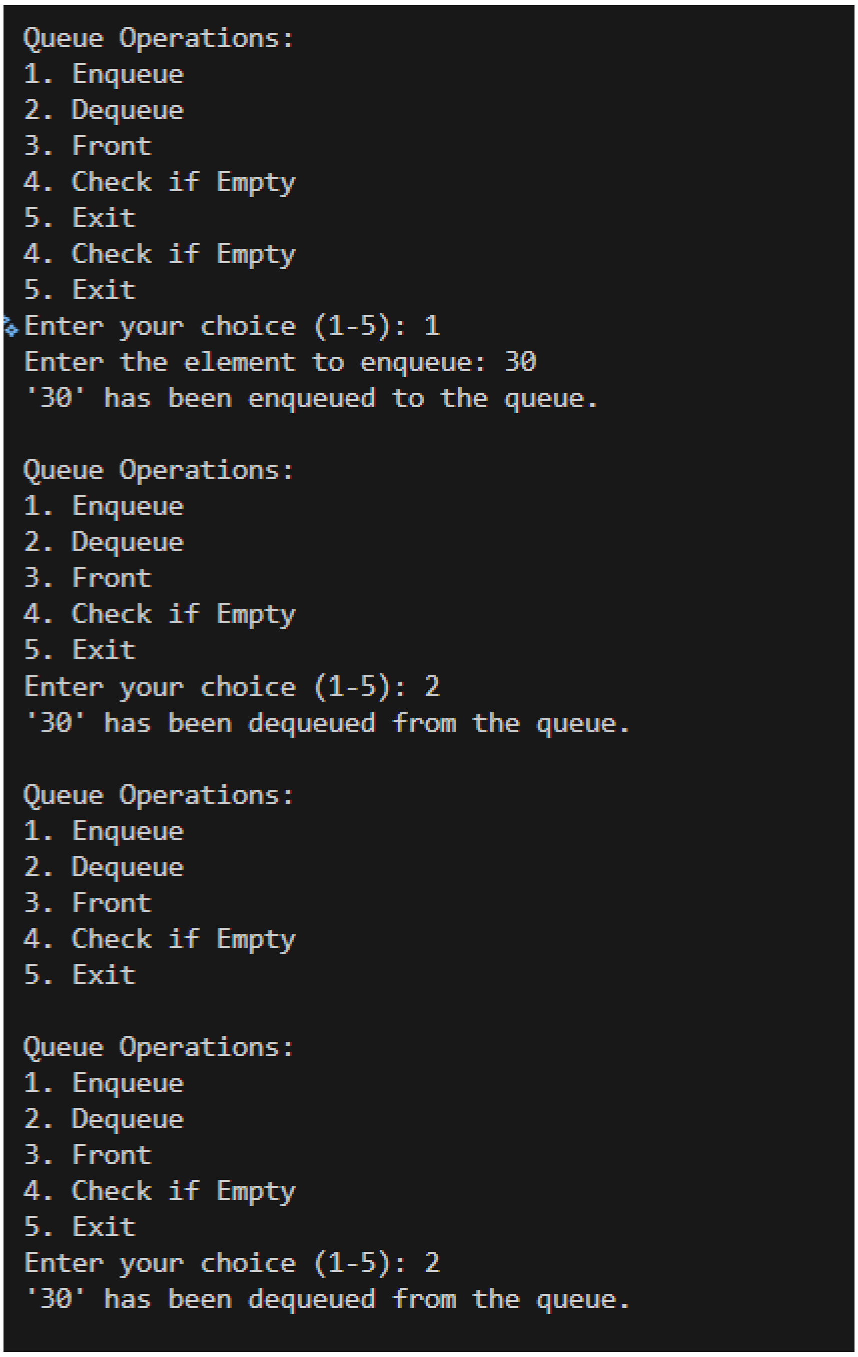


```python
'''
Generate a Python program that implements a Queue data structure using a class. Include methods enqueue(element), dequeue(), front(),
and is_empty(). Handle queue overflow and underflow conditions and include proper docstrings for each method. Take input from user to
perform queue operations.'''
class Queue:
    """A simple implementation of a Queue data structure."""
    def __init__(self):
        """Initialize an empty queue."""
        self.queue = []
    def enqueue(self, element):
        """Add an element to the end of the queue.
        Args:
            element: The element to be added to the queue.
        """
        self.queue.append(element)
    def dequeue(self):
        """Remove and return the front element of the queue.
        Returns:
            The front element of the queue.
        Raises:
            IndexError: If the queue is empty (queue underflow).
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot dequeue from an empty queue.")
        return self.queue.pop(0)
    def front(self):
        """Return the front element of the queue without removing it.
        Returns:
            The front element of the queue.
        Raises:
            IndexError: If the queue is empty (queue underflow).
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot access front of an empty queue.")
        return self.queue[0]
    def is_empty(self):
        """Check if the queue is empty.
        Returns:
            True if the queue is empty, False otherwise.
        """
```



```python
        return len(self.queue) == 0
def main():
    queue = Queue()
    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Front")
        print("4. Check if Empty")
        print("5. Exit")
        choice = input("Enter your choice (1-5): ")
        if choice == '1':
            element = input("Enter the element to enqueue: ")
            queue.enqueue(element)
            print(f"'{element}' has been enqueued to the queue.")
        elif choice == '2':
            try:
                dequeued_element = queue.dequeue()
                print(f"'{dequeued_element}' has been dequeued from the queue.")
            except IndexError as e:
                print(e)
        elif choice == '3':
            try:
                front_element = queue.front()
                print(f"The front element is: '{front_element}'")
            except IndexError as e:
                print(e)
        elif choice == '4':
            if queue.is_empty():
                print("The queue is empty.")
            else:
                print("The queue is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
if __name__ == "__main__":
    main()
```

OUTPUT:



```
Queue Operations:
1. Enqueue
2. Dequeue
3. Front
4. Check if Empty
5. Exit
4. Check if Empty
5. Exit
Enter your choice (1-5): 1
Enter the element to enqueue: 30
'30' has been enqueued to the queue.

Queue Operations:
1. Enqueue
2. Dequeue
3. Front
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
'30' has been dequeued from the queue.

Queue Operations:
1. Enqueue
2. Dequeue
3. Front
4. Check if Empty
5. Exit

Queue Operations:
1. Enqueue
2. Dequeue
3. Front
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
'30' has been dequeued from the queue.
```

Observation

A Queue is appropriate because it follows the First-In First-Out (FIFO) principle, ensuring that elements are processed in the exact order they arrive. This behavior is essential for applications such as task scheduling, buffering, and request handling, where fairness and order must be preserved. Other data structures, such as Stacks (LIFO) or general Linked Lists, do not inherently maintain FIFO ordering, making them less suitable for these types of operations.

Task Description -5 (Algorithms – Bubble Sort vs Selection Sort)

➢ Task: Use AI to implement Bubble Sort and Selection Sort algorithms and compare their behavior.

Instructions:

➢ Prompt AI to generate:

• bubble_sort(arr)

• selection_sort(arr)

➢ Include comments explaining each step.

➢ Add docstrings mentioning time and space complexity.

Expected Output:

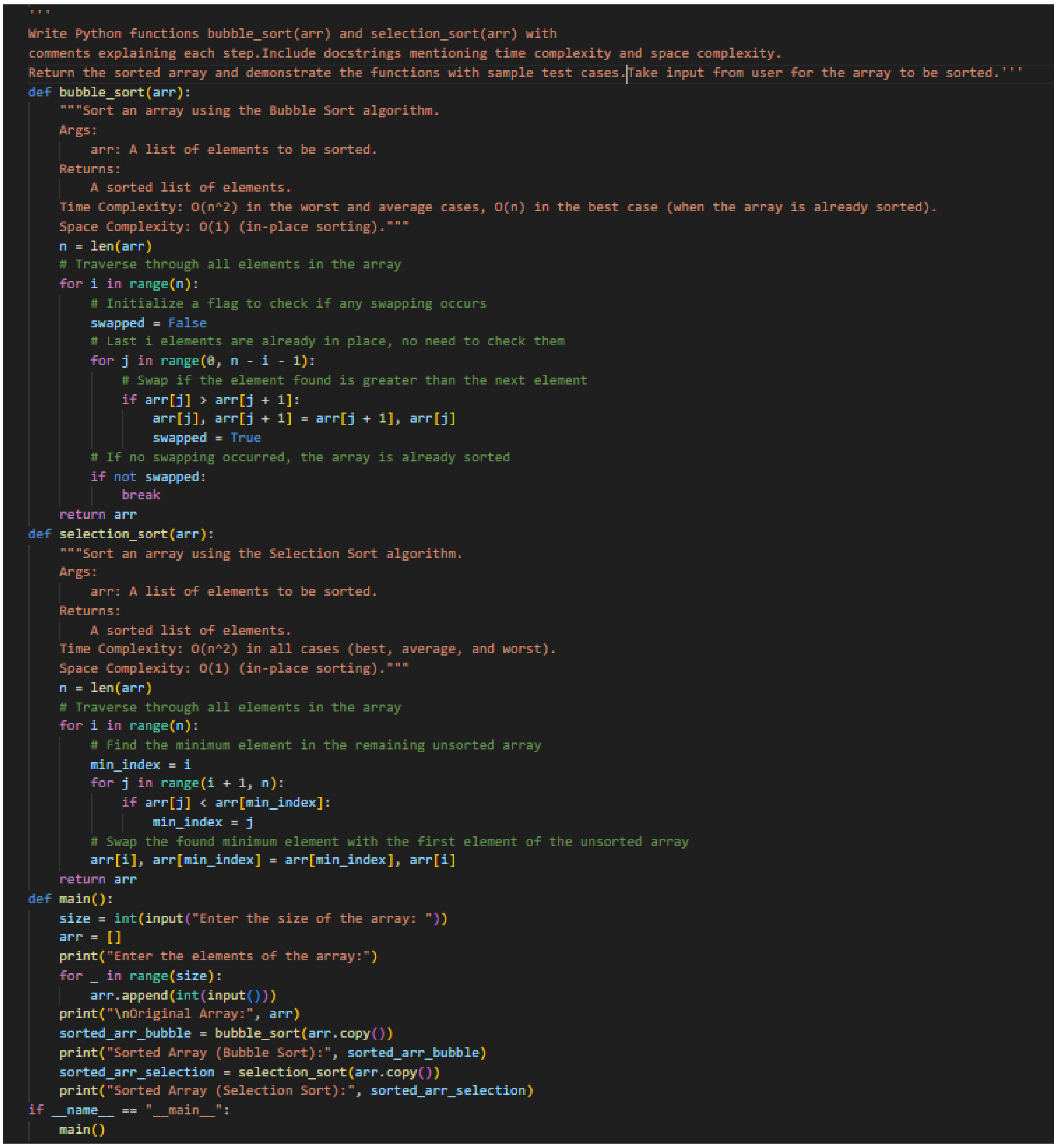
• Correct Python implementations of both sorting algorithms.

• Complexity analysis in docstrings.

Prompt:

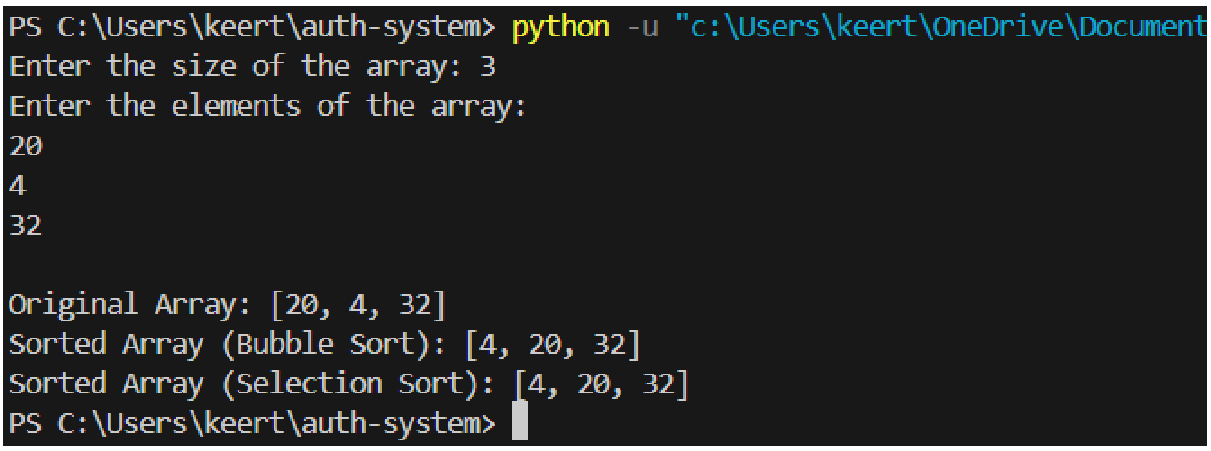
Write Python functions bubble_sort(arr) and selection_sort(arr) with comments explaining each step. Include docstrings mentioning time complexity and space complexity. Return the sorted array and demonstrate the functions with sample test cases. Take input from user for the array to be sorted.

Code:



```python
'''
Write Python functions bubble_sort(arr) and selection_sort(arr) with
comments explaining each step.Include docstrings mentioning time complexity and space complexity.
Return the sorted array and demonstrate the functions with sample test cases.Take input from user for the array to be sorted.'''
def bubble_sort(arr):
    """Sort an array using the Bubble Sort algorithm.
    Args:
        arr: A list of elements to be sorted.
    Returns:
        A sorted list of elements.
    Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the array is already sorted).
    Space Complexity: O(1) (in-place sorting)."""
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False
        # Last i elements are already in place, no need to check them
        for j in range(0, n - i - 1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swapping occurred, the array is already sorted
        if not swapped:
            break
    return arr
def selection_sort(arr):
    """Sort an array using the Selection Sort algorithm.
    Args:
        arr: A list of elements to be sorted.
    Returns:
        A sorted list of elements.
    Time Complexity: O(n^2) in all cases (best, average, and worst).
    Space Complexity: O(1) (in-place sorting)."""
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Find the minimum element in the remaining unsorted array
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Swap the found minimum element with the first element of the unsorted array
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
def main():
    size = int(input("Enter the size of the array: "))
    arr = []
    print("Enter the elements of the array:")
    for _ in range(size):
        arr.append(int(input()))
    print("\nOriginal Array:", arr)
    sorted_arr_bubble = bubble_sort(arr.copy())
    print("Sorted Array (Bubble Sort):", sorted_arr_bubble)
    sorted_arr_selection = selection_sort(arr.copy())
    print("Sorted Array (Selection Sort):", sorted_arr_selection)
if __name__ == "__main__":
    main()
```

OUTPUT:



```
PS C:\Users\keert\auth-system> python -u "c:\Users\keert\OneDrive\Document
Enter the size of the array: 3
Enter the elements of the array:
20
4
32

Original Array: [20, 4, 32]
Sorted Array (Bubble Sort): [4, 20, 32]
Sorted Array (Selection Sort): [4, 20, 32]
PS C:\Users\keert\auth-system>
```

Observation

Sorting algorithms such as Bubble Sort and Selection Sort require efficient indexed access to elements, which is naturally supported by arrays (Python lists). Arrays allow quick comparison and swapping of elements using their indices, making them ideal for these algorithms. In contrast, data structures like linked lists, stacks, or queues do not provide efficient random access, making element comparison and swapping significantly slower and less suitable for these sorting methods.