Course: AI Assisted Coding (23CS002PC304)

Assignment: Lab 12

Name : Mukesh Babu

HT NO : 2303A52061

Batch No : 39

Task 1: Data Structures – Stack Implementation with AI Assistance

Task Description

Use AI assistance to generate a Python program that implements a Stack data structure. The stack must include push(element), pop(), peek(), and is_empty() methods, with proper error handling for stack underflow and clear docstrings.

Prompt Used

> "Write a Python program that implements a Stack data structure using a class. Include methods for push(element), pop(), peek(), and is_empty(). Make sure to include proper error handling for stack underflow (e.g., trying to pop from an empty stack) and add clear, descriptive docstrings for each method. Include a small test at the end to demonstrate its functionality."

Code

```
class Stack:
    """A class representing a Stack data structure using a list."""

    def __init__(self):
        """Initializes an empty stack."""
        self.items = []

    def is_empty(self):
        """Checks if the stack is empty.
        Returns: bool: True if empty, False otherwise.
        """
        return len(self.items) == 0

    def push(self, element):
        """Adds an element to the top of the stack.
        Args: element: The item to be added to the stack.
```

```
        """
        self.items.append(element)
        print(f"Pushed: {element}")

    def pop(self):
        """Removes and returns the top element of the stack.
        Returns: The element removed from the top of the stack.
        Raises: IndexError: If the stack is empty (underflow).
        """
        if self.is_empty():
            raise IndexError("Stack Underflow: Cannot pop from an empty stack.")
        popped_item = self.items.pop()
        print(f"Popped: {popped_item}")
        return popped_item

    def peek(self):
        """Returns the top element without removing it.
        Returns: The top element of the stack.
        Raises: IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty: Nothing to peek.")
        return self.items[-1]

# --- Testing the Stack ---
if __name__ == "__main__":
    my_stack = Stack()
    my_stack.push(10)
    my_stack.push(20)
    my_stack.push(30)

    print(f"Top element is: {my_stack.peek()}")
    my_stack.pop()
    my_stack.pop()

    print(f"Is stack empty? {my_stack.is_empty()}")
    my_stack.pop()

    # Testing underflow error handling
    try:
        my_stack.pop()
    except IndexError as e:
        print(f"Error caught: {e}")
```

Output

Pushed: 10
Pushed: 20
Pushed: 30
Top element is: 30
Popped: 30
Popped: 20
Is stack empty? False
Popped: 10
Error caught: Stack Underflow: Cannot pop from an empty stack.

Explanation

A Stack works on the **LIFO (Last-In, First-Out)** principle—imagine a stack of plates where you can only take the top one off or put a new one on top. In Python, we can easily mimic this behavior using a standard list. The push method uses append() to add to the end of the list, and pop uses the list's built-in pop() method to remove the last item. To make the code robust, we added an if self.is_empty(): check inside the pop() and peek() methods. If someone tries to remove an item when there are no items left, the program gracefully raises an IndexError rather than crashing unexpectedly.

Task 2: Algorithms – Linear vs Binary Search Analysis

Task Description

Use AI to implement and compare Linear Search and Binary Search algorithms in Python. Provide docstrings explaining the working principles, and test both algorithms using different input sizes.

Prompt Used

> "Implement Linear Search and Binary Search algorithms in Python. Write clear docstrings for both explaining their working principles, time complexity, and space complexity. Finally, write some test cases that search for elements in an array, demonstrating how both functions return the correct index."

Code

```
def linear_search(arr, target):
    """
    Performs a linear search to find a target element in an array.
```

```
    Working Principle: Checks each element in the array sequentially from
    start to finish until the target is found or the list ends.

    Time Complexity: O(n) in the worst case, where n is the number of elements.
    Space Complexity: O(1) as it uses no extra memory.

    Args:
        arr (list): The list to search through.
        target: The element to find.
    Returns:
        int: The index of the target if found, else -1.
    """
    for index, element in enumerate(arr):
        if element == target:
            return index
    return -1

def binary_search(arr, target):
    """
    Performs a binary search to find a target element in a SORTED array.

    Working Principle: It compares the target value to the middle element of the array.
    If they are not equal, the half in which the target cannot lie is eliminated,
    and the search continues on the remaining half until it is found.

    Time Complexity: O(log n) in the worst case.
    Space Complexity: O(1) for the iterative approach.

    Args:
        arr (list): A SORTED list to search through.
        target: The element to find.
    Returns:
        int: The index of the target if found, else -1.
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
```

```
            left = mid + 1
        else:
            right = mid - 1

    return -1

# --- Testing and Comparison ---
if __name__ == "__main__":
    # Test Data
    test_array = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
    target_value = 23
    missing_value = 99

    print("--- Linear Search ---")
    print(f"Index of {target_value}: {linear_search(test_array, target_value)}")
    print(f"Index of {missing_value}: {linear_search(test_array, missing_value)}")

    print("\n--- Binary Search ---")
    print(f"Index of {target_value}: {binary_search(test_array, target_value)}")
    print(f"Index of {missing_value}: {binary_search(test_array, missing_value)}")
```

Output

```
--- Linear Search ---
Index of 23: 5
Index of 99: -1

--- Binary Search ---
Index of 23: 5
Index of 99: -1
```

Explanation

Linear Search is the most straightforward way to find something: you start at the beginning of the list and check every single item until you find what you're looking for. It's safe and works on any list, but it's slow for massive datasets ($O(N)$ time).

Binary Search, on the other hand, is much smarter but has a strict rule: *the data must be sorted first*. It works like finding a word in a dictionary; you open it to the middle, see if your word comes before or after that page, and then cut the problem in half. It keeps halving the search area, which makes it incredibly fast ($O(\log N)$ time). Both codes successfully return the index

5 for the number 23, and -1 indicating the number 99 isn't there.

Task 3: Test Driven Development – Simple Calculator Function

Task Description

Apply Test Driven Development (TDD) using AI assistance to develop a calculator function. First, generate test cases, run them to see failures, and then implement the code to pass the tests.

Prompt Used

> "Let's do Test-Driven Development (TDD) for a simple Python calculator.
>
> Step 1: Write a Python unittest file containing test cases for add(a, b) and subtract(a, b).
>
> Step 2: Write the actual calculator functions in a separate module so that the tests will pass."

Code

File 1: test_calculator.py (Created First)

```
import unittest
from calculator import add, subtract

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(10, 5), 15)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(-5, -5), -10)

    def test_subtract(self):
        self.assertEqual(subtract(10, 5), 5)
        self.assertEqual(subtract(-1, 1), -2)
        self.assertEqual(subtract(0, 0), 0)

if __name__ == '__main__':
    unittest.main()
```

(Note: If run at this stage, Python throws an ImportError because calculator.py doesn't exist yet —this is the "Red" phase of TDD).

File 2: calculator.py (Implementation created after)

```
def add(a, b):
    """Returns the sum of two numbers."""
    return a + b

def subtract(a, b):
    """Returns the difference when b is subtracted from a."""
    return a - b
```

Output (After Implementation)

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Explanation

Test-Driven Development (TDD) flips the normal coding process upside down. Instead of writing code and then testing it, we write the tests *first*.

1. **Red Phase**: We wrote test_calculator.py. When we tried to run it, it failed completely because the functions add and subtract didn't even exist yet.
2. **Green Phase**: We wrote the absolute minimum code required in calculator.py to make the tests pass.
3. **Refactor Phase**: (Not needed here since the code is very simple).
   When we ran the test file again after creating the implementation, the output printed .. and OK, meaning both our addition and subtraction logic worked perfectly against all edge cases (positive numbers, negative numbers, and zeros).

Task 4: Data Structures – Queue Implementation with AI Assistance

Task Description

Use AI assistance to generate a Python program that implements a Queue data structure with methods: enqueue(element), dequeue(), front(), and is_empty(). Handle queue overflow and underflow conditions, and include docstrings.

Prompt Used

"Write a Python program that implements a Queue data structure using a class. It should have a maximum capacity to handle overflow. Include methods for enqueue(element), dequeue(), front(), and is_empty(). Ensure there is proper exception handling for queue overflow (adding to a full queue) and underflow (removing from an empty queue). Include detailed docstrings."

Code

```
class Queue:
  """A class representing a Queue data structure with a fixed capacity."""

  def __init__(self, capacity):
    """
    Initializes an empty queue with a maximum capacity.
    Args: capacity (int): The maximum number of elements the queue can hold.
    """
    self.items = []
    self.capacity = capacity

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

  def is_full(self):
    """Checks if the queue has reached its maximum capacity."""
    return len(self.items) >= self.capacity

  def enqueue(self, element):
    """
    Adds an element to the back of the queue.
    Raises: OverflowError: If the queue is already full.
    """
    if self.is_full():
      raise OverflowError("Queue Overflow: Cannot enqueue, queue is full.")
    self.items.append(element)
    print(f"Enqueued: {element}")

  def dequeue(self):
    """
    Removes and returns the front element of the queue.
    Raises: IndexError: If the queue is empty (underflow).
    """
    if self.is_empty():
      raise IndexError("Queue Underflow: Cannot dequeue from an empty queue.")
```

```python
        # pop(0) removes the first element (front of the queue)
        dequeued_item = self.items.pop(0)
        print(f"Dequeued: {dequeued_item}")
        return dequeued_item

    def front(self):
        """
        Returns the front element without removing it.
        Raises: IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue is empty: No front element.")
        return self.items[0]

# --- Testing the Queue ---
if __name__ == "__main__":
    q = Queue(capacity=3)

    q.enqueue("Alice")
    q.enqueue("Bob")
    q.enqueue("Charlie")

    print(f"Front of queue is: {q.front()}")

    # Testing overflow
    try:
        q.enqueue("David")
    except OverflowError as e:
        print(f"Error caught: {e}")

    q.dequeue()
    q.dequeue()
    q.dequeue()

    # Testing underflow
    try:
        q.dequeue()
    except IndexError as e:
        print(f"Error caught: {e}")
```

Output

Enqueued: Alice

```
Enqueued: Bob
Enqueued: Charlie
Front of queue is: Alice
Error caught: Queue Overflow: Cannot enqueue, queue is full.
Dequeued: Alice
Dequeued: Bob
Dequeued: Charlie
Error caught: Queue Underflow: Cannot dequeue from an empty queue.
```

Explanation

A Queue operates on the **FIFO (First-In, First-Out)** principle, exactly like a line of people waiting at a ticket counter. The first person to get in line (enqueue) is the first person to get served and leave (dequeue).

To simulate this in Python, we use a list. When we enqueue, we add the item to the end of the list using append(). When we dequeue, we remove the item from index 0 using pop(0). We also added a capacity limit. If the list length reaches the capacity, trying to add another item triggers an OverflowError. Conversely, if the line is completely empty, trying to remove a person triggers an IndexError (Underflow).

Task 5: Algorithms – Bubble Sort vs Selection Sort

Task Description

Use AI to implement Bubble Sort and Selection Sort algorithms and compare their behavior. Include comments explaining each step, and docstrings mentioning time and space complexity.

Prompt Used

> "Implement Bubble Sort and Selection Sort algorithms in Python. For both functions, include detailed inline comments explaining what each line/step is doing. Add docstrings that specify their time and space complexity. Provide test cases printing the arrays before and after sorting."

Code

```python
def bubble_sort(arr):
    """
    Sorts an array using the Bubble Sort algorithm.

    Time Complexity: Best Case O(n) (if already sorted), Worst/Average Case O(n^2).
    Space Complexity: O(1) (in-place sorting).
    """
```

```
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place, so we don't check them again
        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 two elements were swapped by inner loop, then the array is sorted
        if not swapped:
            break
    return arr

def selection_sort(arr):
    """
    Sorts an array using the Selection Sort algorithm.

    Time Complexity: Best/Worst/Average Case O(n^2).
    Space Complexity: O(1) (in-place sorting).
    """
    n = len(arr)
    # Traverse through all array elements
    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 part
        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr

# --- Testing the Sorting Algorithms ---
if __name__ == "__main__":
    # Test Data
    array1 = [64, 34, 25, 12, 22, 11, 90]
    array2 = [64, 25, 12, 22, 11]
```

```
print("--- Bubble Sort ---")
print(f"Original: {array1}")
print(f"Sorted:   {bubble_sort(array1.copy())}")

print("\n--- Selection Sort ---")
print(f"Original: {array2}")
print(f"Sorted:   {selection_sort(array2.copy())}")
```

Output

```
--- Bubble Sort ---
Original: [64, 34, 25, 12, 22, 11, 90]
Sorted:   [11, 12, 22, 25, 34, 64, 90]

--- Selection Sort ---
Original: [64, 25, 12, 22, 11]
Sorted:   [11, 12, 22, 25, 64]
```

Explanation

Both of these algorithms are fundamental ways to sort data, though they take different approaches:

1. **Bubble Sort** works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. It's called "bubble sort" because the largest numbers "bubble" up to the top (the end of the list) with each pass. We optimized it with a swapped flag—if it goes through the list and doesn't make any swaps, it knows the list is already sorted and stops early.
2. **Selection Sort** divides the list into a "sorted" section and an "unsorted" section. It scans the entire unsorted section to find the absolute minimum value, and then swaps that minimum value into the first position of the unsorted section (effectively growing the sorted section by one).

Both algorithms have a Time Complexity of $O(n^2)$, meaning they are quite slow for very large lists, but they are easy to understand and use no extra memory (Space Complexity $O(1)$).