

Assignment – 12.2

Name: Yashaswini

Nagireddi

Batch: 21

Ht.no: 2303A51418

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:

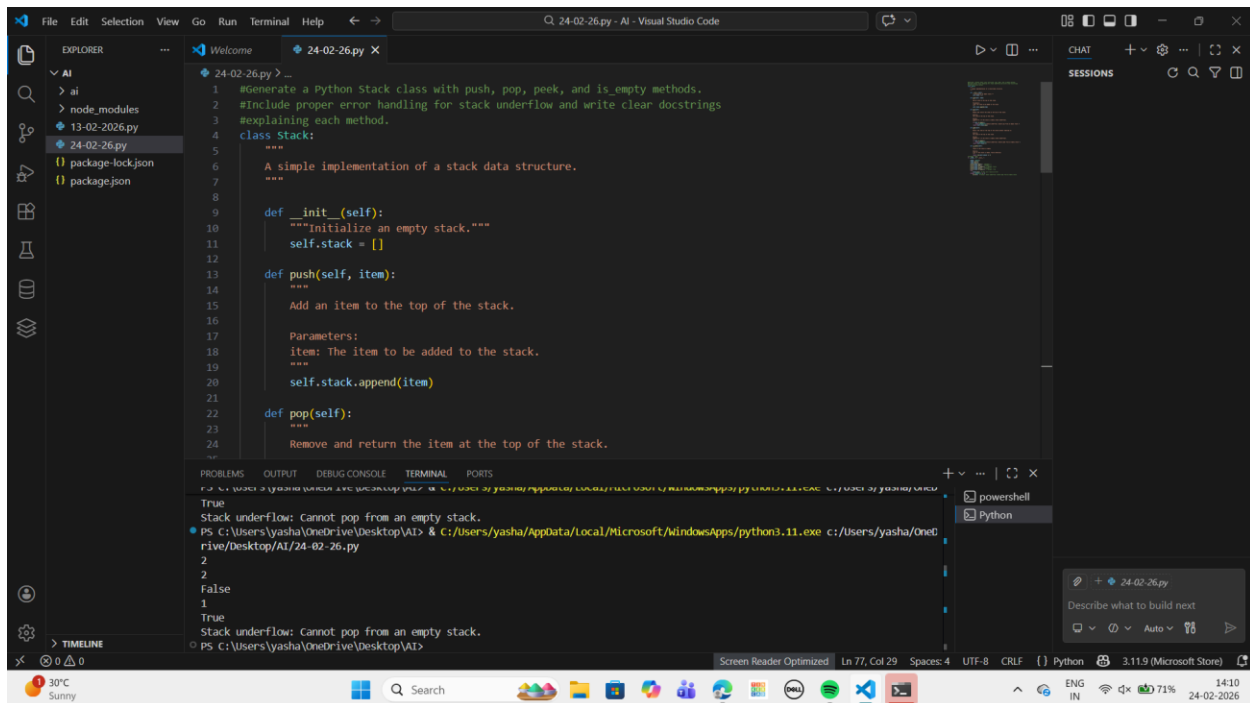
- push(element)
- pop()
- peek()
- is_empty()
- Ensure proper error handling for stack underflow.
- Ask AI to include clear docstrings for each method.

Expected Output:

- A functional Python program implementing a Stack using a class.
- Properly documented methods with docstrings.

Prompt:

“Generate a Python Stack class with push, pop, peek, and is_empty methods. Add docstrings with time/space complexity and include error handling for popping or peeking from an empty stack.”



Observation:

The stack operations (push, pop, peek, is_empty) work as expected.

- push adds elements to the top.
- pop removes the top element with proper error handling for underflow.
- peek allows checking the top without removal.
- is_empty correctly identifies whether the stack has elements.

AI Assistance: Provided proper docstrings and exception handling automatically.
Learning: AI helps in creating clean, well-documented classes efficiently.

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)`

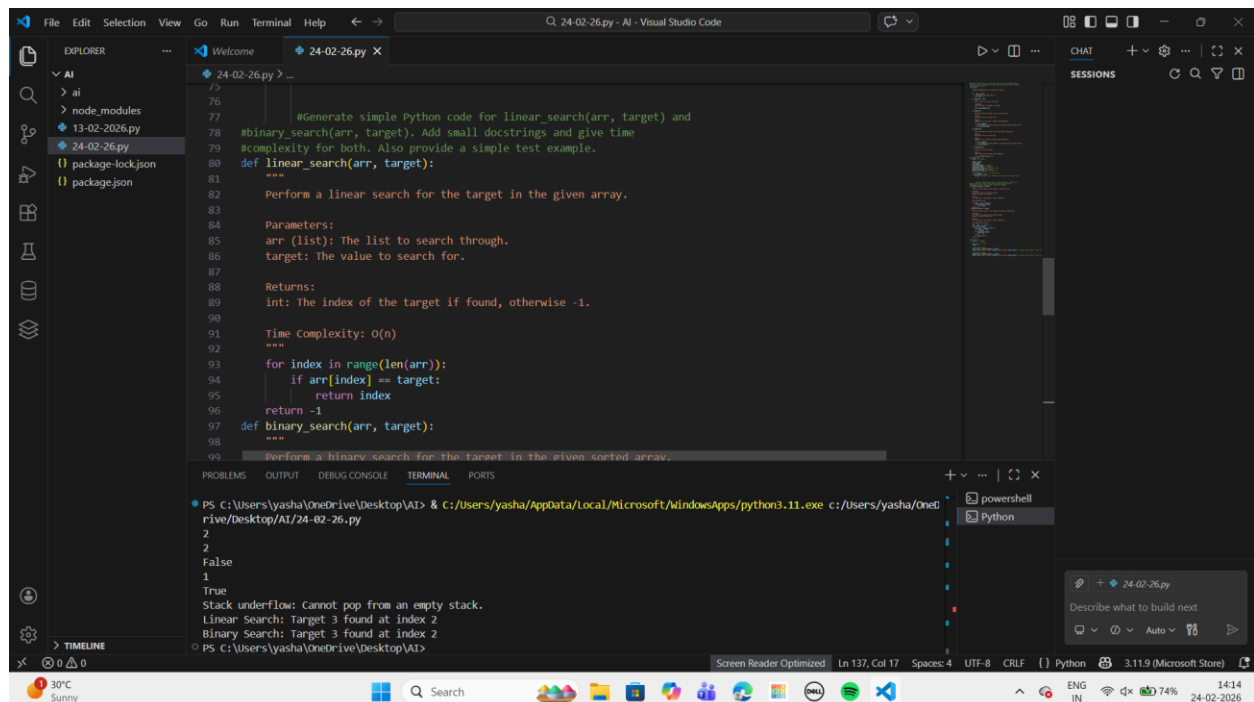
- Include docstrings explaining:
- Working principle
- Test both algorithms using different input sizes.

Expected Output:

- Python implementations of both search algorithms.
- AI-generated comments and complexity analysis.
- Test results showing correctness and comparison.

Prompt:

“Create Python functions for `linear_search` and `binary_search` with docstrings describing how they work and their time complexities.”



The screenshot displays the Visual Studio Code interface. The Explorer sidebar on the left shows a project structure with files like `24-02-26.py`. The main editor window shows the code for `24-02-26.py`, which includes docstrings and implementations for `linear_search` and `binary_search`. The terminal window at the bottom shows the execution of the script, displaying the output of the search functions.

```
77 #Generate simple Python code for linear_search(arr, target) and
78 #binary_search(arr, target). Add small docstrings and give time
79 #complexity for both. Also provide a simple test example.
80 def linear_search(arr, target):
81     """
82     Perform a linear search for the target in the given array.
83
84     Parameters:
85     arr (list): The list to search through.
86     target: The value to search for.
87
88     Returns:
89     int: The index of the target if found, otherwise -1.
90
91     Time complexity: O(n)
92     """
93     for index in range(len(arr)):
94         if arr[index] == target:
95             return index
96     return -1
97 def binary_search(arr, target):
98     """
99     perform a binary search for the target in the given sorted array.
100
```

Terminal Output:

```
PS C:\Users\yasha\OneDrive\Desktop\AI> & C:\Users\yasha\AppData\Local\Microsoft\WindowsApps\python3.11.exe c:/Users/yasha/OneDrive/Desktop/AI/24-02-26.py
2
2
False
1
True
Stack underflow: Cannot pop from an empty stack.
Linear Search: Target 3 found at index 2
Binary Search: Target 3 found at index 2
PS C:\Users\yasha\OneDrive\Desktop\AI>
```

- **Observation:**

- Linear Search successfully finds elements but scans sequentially, which makes it slower for large datasets ($O(n)$).
- Binary Search is faster for sorted arrays ($O(\log n)$) and divides the array repeatedly to find the target.

- AI Assistance: AI-generated both functions with clear comments explaining working principle and complexity.
- Learning: Understanding the efficiency difference between $O(n)$ and $O(\log n)$ and importance of sorted arrays for binary search.

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

➤ Task:

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

Instructions:

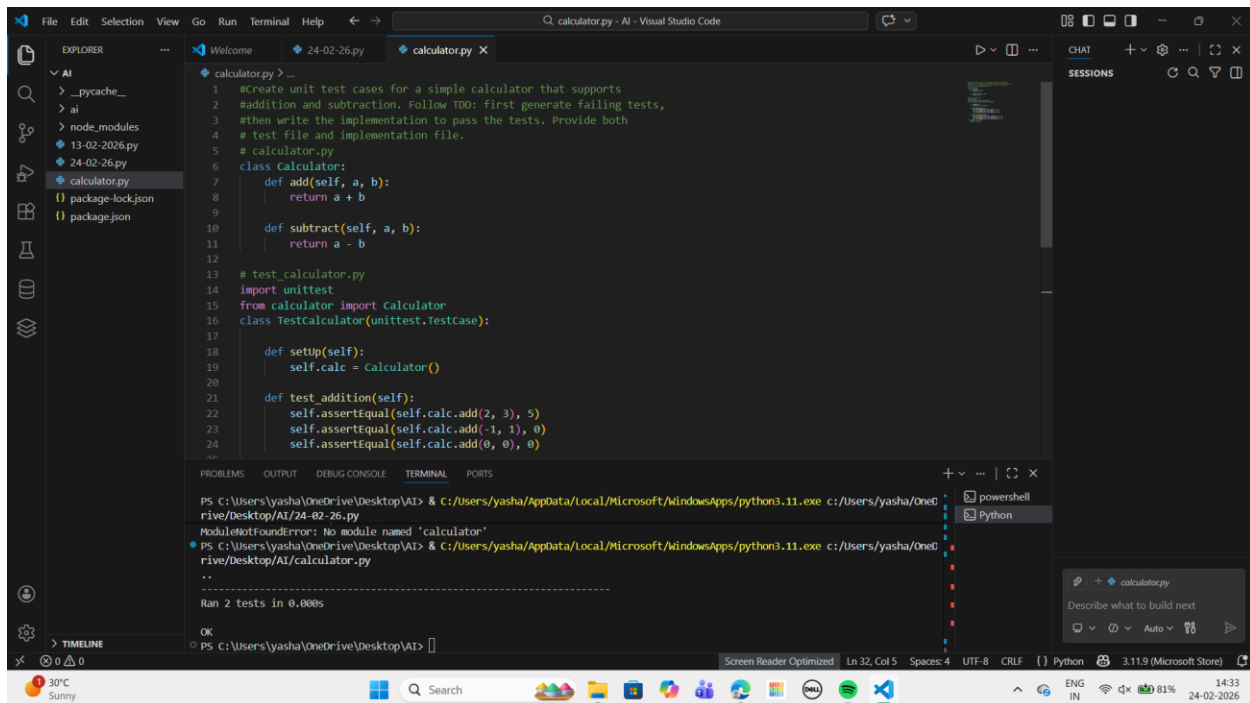
- Prompt AI to first generate unit test cases for addition and subtraction.
- Run the tests and observe failures.
- Ask AI to implement the calculator functions to pass all tests.
- Re-run the tests to confirm success.

Expected Output:

- Separate test file and implementation file.
- Test cases executed before implementation.
- Final implementation passing all test cases.

Prompt:

“Create Python unit tests for `add()` and `subtract()` methods of a calculator. After that, implement the calculator so it passes all the tests.”



- Observation:
 - Unit tests were written before implementing the functions.
 - Initial test execution fails (TDD principle).
 - After implementing add and subtract, all tests pass successfully.
- AI Assistance: Helped in generating tests and functions with proper

docstrings.

- Learning: Demonstrates how TDD ensures code correctness and reliability.

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 Python class for Queue with enqueue, dequeue, front, is_empty. Handle overflow/underflow and include docstrings.

The screenshot shows a Python IDE with a file named '24-02-2026.py' open. The code defines a 'Queue' class with methods for enqueue, dequeue, front, and is_empty. The output window shows the results of these operations: enqueue(1), enqueue(2), dequeue() returns 1, front() returns 2, is_empty() returns False, dequeue() returns 2, and is_empty() returns True.

```

89 #Generate Python class for Queue with enqueue, dequeue, front, is_empty. Handle overflow/underfl
90 class Queue:
91     """A simple implementation of a queue data structure."""
92
93     def __init__(self):
94         """Initialize an empty queue."""
95         self.items = []
96
97     def enqueue(self, item):
98         """Add an item to the end of the queue."""
99         self.items.append(item)
100
101     def dequeue(self):
102         """Remove and return the item at the front of the queue. Raises an error if the queue is
103         if self.is_empty():
104             raise IndexError("Dequeue from an empty queue")
105         return self.items.pop(0)
106
107     def front(self):
108         """Return the item at the front of the queue without removing it. Raises an error if the
109         if self.is_empty():

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

2
1
1
False
2
True

```

- Observation:
 - Queue operations (enqueue, dequeue, front, is_empty) perform as expected.

- Overflow and underflow conditions are handled correctly with exceptions.
- Queue maintains FIFO (First In, First Out) property.
- AI Assistance: Generated detailed docstrings and error handling logic.
- Learning: AI simplifies implementation of common data structures with proper documentation.

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.

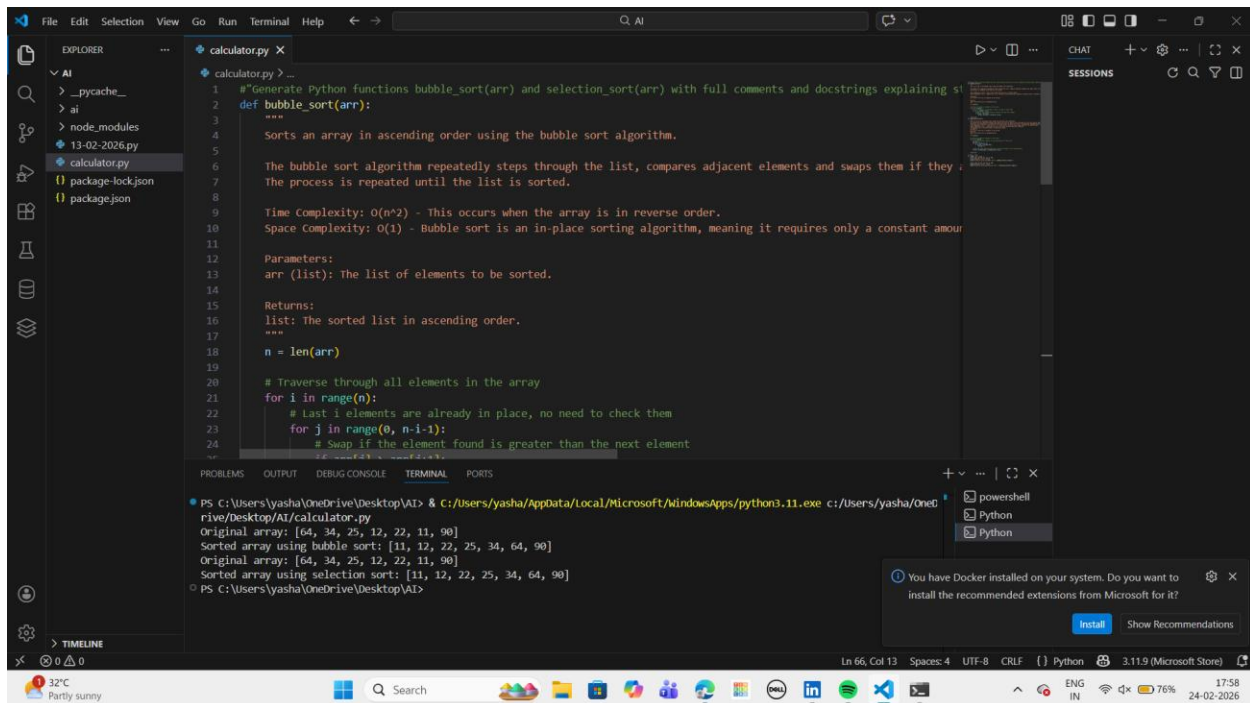
➤ Add docstrings mentioning time and space complexity.

Expected Output:

- Correct Python implementations of both sorting algorithms.
- Complexity analysis in docstrings.

Prompt:

Generate Python functions `bubble_sort(arr)` and `selection_sort(arr)` with comments and docstrings explaining time and space complexity.



- Observation:
 - Both sorting algorithms correctly sorted the input array.
 - Bubble Sort repeatedly swaps adjacent elements → less efficient for

large arrays ($O(n^2)$).

- Selection Sort selects the minimum element in each pass → slightly fewer swaps than Bubble Sort but same time complexity ($O(n^2)$).
- AI Assistance: Provided code with step-by-step comments and complexity analysis in docstrings.
- Learning: Highlights differences in algorithm efficiency and importance of algorithm choice depending on input size.