

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.”

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "AI" containing files: "ai", "node_modules", "13-02-2026.py", "24-02-26.py", "package-lock.json", and "package.json".
- Code Editor:** Displays the content of "24-02-26.py". The code defines a Stack class with push, pop, peek, and is_empty methods.
- Terminal:** Shows command-line output from running the script. It includes imports, variable assignments, and two error messages indicating stack underflow when popping from an empty stack.
- Status Bar:** Shows the file path "C:/Users/yasha/Desktop/AI/24-02-26.py", screen reader optimization status, line 77, column 29, spaces 4, encoding UTF-8, CRLF, Python 3.11.9, Microsoft Store, and system status (ENG IN, 71%, 14:10, 24-02-2026).

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)`

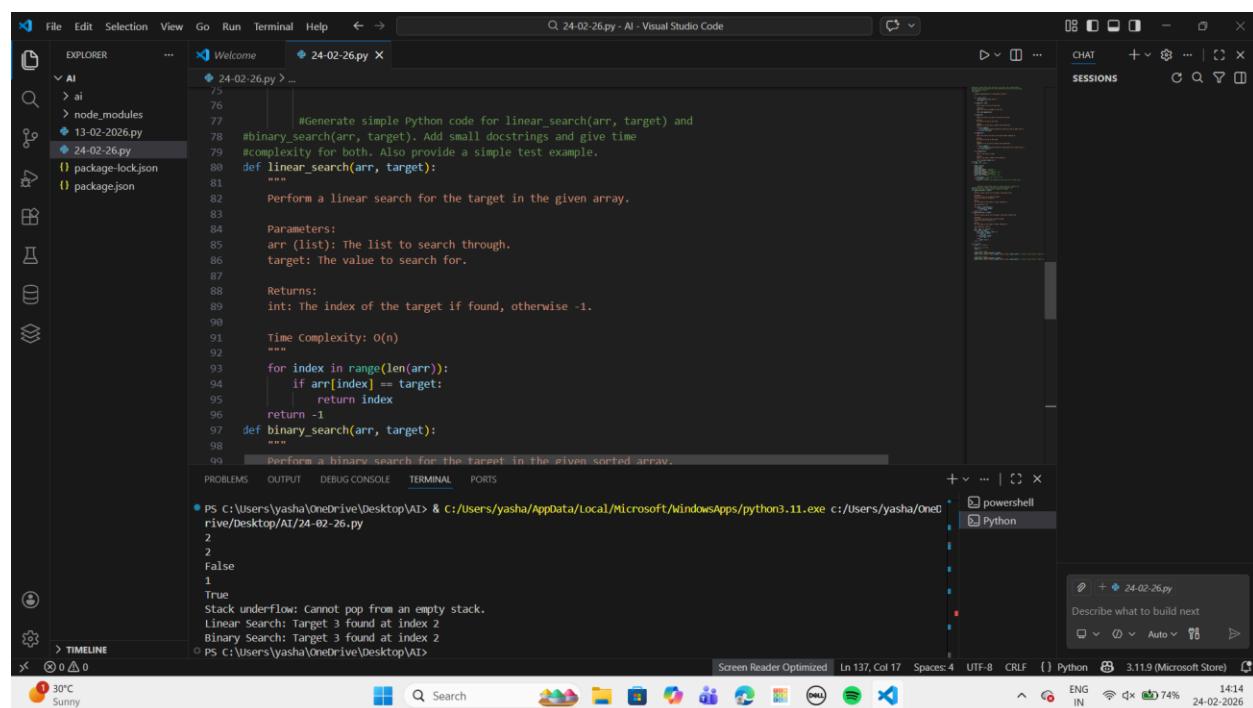
- Include docstrings explaining:
- Working principle
- Test both algorithms using different input sizes.

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 shows the Visual Studio Code interface. The left sidebar displays a file tree with files like '24-02-26.py', '13-02-2026.py', and 'package-lock.json'. The main code editor area contains the following Python code for linear and binary search:

```

#Generate simple Python code for linear_search(arr, target) and
#binary_search(arr, target). Add small docstrings and give time
#complexity for both. Also provide a simple test example.
def linear_search(arr, target):
    """
    Perform a linear search for the target in the given array.

    Parameters:
    arr (list): The list to search through.
    target: The value to search for.

    Returns:
    int: The index of the target if found, otherwise -1.

    Time Complexity: O(n)
    """
    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 in the given sorted array.
    """

```

The terminal at the bottom shows the command being run: 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. The output indicates that both linear and binary search find the target value 3 at index 2.

- **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.”

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files like `calculator.py`, `__pycache__`, `ai`, `node_modules`, and `package-lock.json`.
- Editor:** Displays the content of `calculator.py` which contains a simple calculator class with `add` and `subtract` methods, and a corresponding test class `Testcalculator` with `setUp` and `test_addition` methods.
- Terminal:** Shows command-line output from running the test script. It includes:
 - 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
 - ModuleNotFoundError: No module named 'calculator'
 - PS C:\Users\yasha\OneDrive\Desktop\AI> & C:/Users/yasha/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/yasha/OneDrive/Desktop/AI/calculator.py
 - ..
- Output:** Shows the result of the test execution: "Ran 2 tests in 0.000s".
- Status Bar:** Includes weather information (30°C, Sunny), system icons, and system status (Screen Reader Optimized, Ln 32, Col 5, Spaces: 4, UTF-8, CRLF).

- 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.

```

89     #Generate Python class for Queue with enqueue, dequeue, front, is_empty. Handle overflow/underflow
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            empty."""
104            if self.is_empty():
105                raise IndexError("Dequeue from an empty queue")
106            return self.items.pop(0)
107
108        def front(self):
109            """Return the item at the front of the queue without removing it. Raises an error if the
110            queue is empty."""

```

- 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.

The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar (EXPLORER) lists files: AI, _pycache_, ai, node_modules, 13-02-2026.py, package-lock.json, and package.json. The main editor window displays a Python file named calculator.py. The code implements bubble sort and selection sort. The terminal at the bottom shows command-line output for both sorting functions.

```
# Generate Python functions bubble_sort(arr) and selection_sort(arr) with full comments and docstrings explaining the algorithm.
def bubble_sort(arr):
    """
    Sorts an array in ascending order using the bubble sort algorithm.
    The bubble sort algorithm repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

    Time Complexity: O(n^2) - This occurs when the array is in reverse order.
    Space Complexity: O(1) - Bubble sort is an in-place sorting algorithm, meaning it requires only a constant amount of extra space.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: The sorted list in ascending order.
    """
    n = len(arr)

    # Traverse through all elements in the array
    for i in range(n):
        # 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]

# Test cases
print("Original array: [64, 34, 25, 12, 22, 11, 98]")
print("Sorted array using bubble sort: [11, 12, 22, 25, 34, 64, 98]")
print("Original array: [64, 34, 25, 12, 22, 11, 98]")
print("Sorted array using selection sort: [11, 12, 22, 25, 34, 64, 98]
```

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/calculator.py
Original array: [64, 34, 25, 12, 22, 11, 98]
Sorted array using bubble sort: [11, 12, 22, 25, 34, 64, 98]
Original array: [64, 34, 25, 12, 22, 11, 98]
Sorted array using selection sort: [11, 12, 22, 25, 34, 64, 98]
```

Bottom status bar: Ln 66, Col 13 | Spaces: 4 | UTF-8 | CRLF | Python | 3.11.9 (Microsoft Store) | 17:58 | ENG | IN | 76% | 24-02-2026

- 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.