NAME : KOUDAGANI SAHITHYA

ENROLLMENT NO : 2403A52063

BATCH NO: 02

SUBJECT: AI ASSISTANT CODING

ASSIGNMENT: 11.3

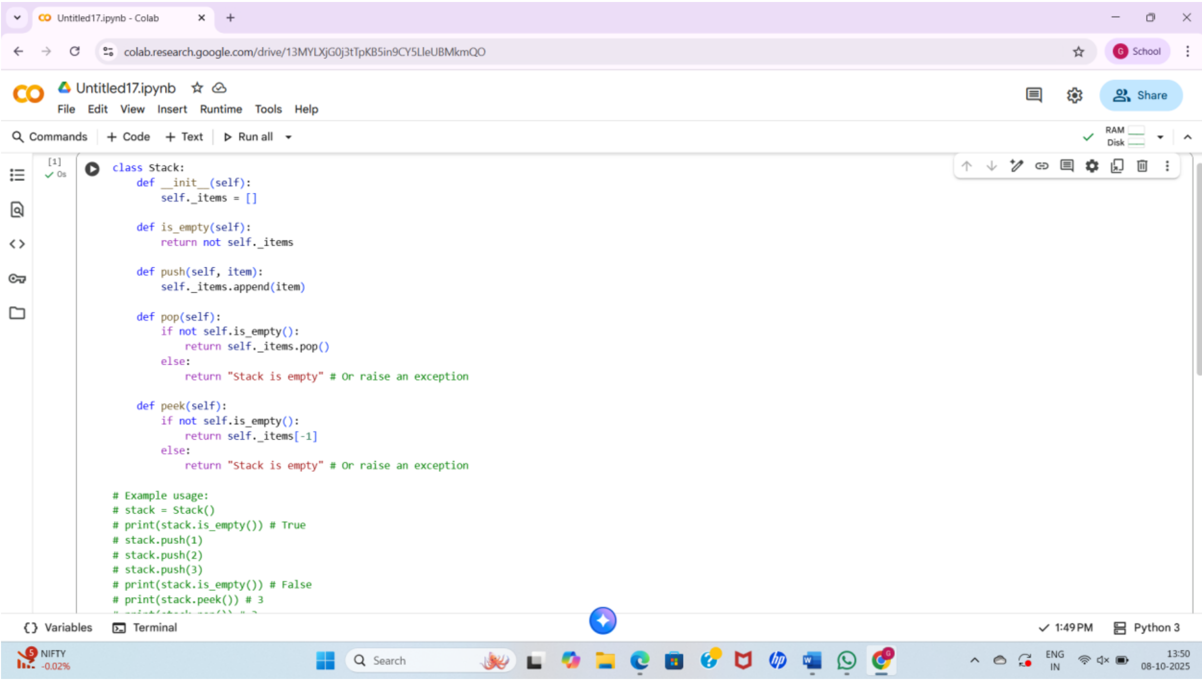
Task Description #1 – Stack class implementation

Task: Ask AI to implement a stack class with push(), pop(), peek() and is_empty() methods

PROMPT

implement a stack class with push(), pop(), peek() and is_empty() methods

code and output



Observations

- **Implementation:** It uses a Python list (_items) to store the stack elements. Lists are a good choice for this because they provide efficient append() and pop() operations for the end of the list, which correspond to the push and pop operations of a stack.
- **Encapsulation:** The use of a leading underscore (_items) is a Python convention to indicate that _items is intended for internal use within the class and shouldn't be directly accessed from outside. This promotes encapsulation.
- **is_empty():** This method efficiently checks if the stack is empty by simply checking if the _items list is empty.
- **push():** This method adds an item to the top of the stack using the list's append() method.
- **pop():** This method removes and returns the top item from the stack using the list's pop() method. It includes a basic check for an empty stack to prevent errors.
- **peek():** This method returns the top item from the stack without removing it. It also includes a check for an empty stack.
- **Error Handling:** For empty stack cases in pop() and peek(), the current implementation returns the string "Stack is empty". In a more robust implementation, you might want to raise an exception (like IndexError or a custom EmptyStackError) to indicate that an invalid operation was attempted.

- **Example Usage:** The commented-out example usage demonstrates how to create a Stack object and use its methods.

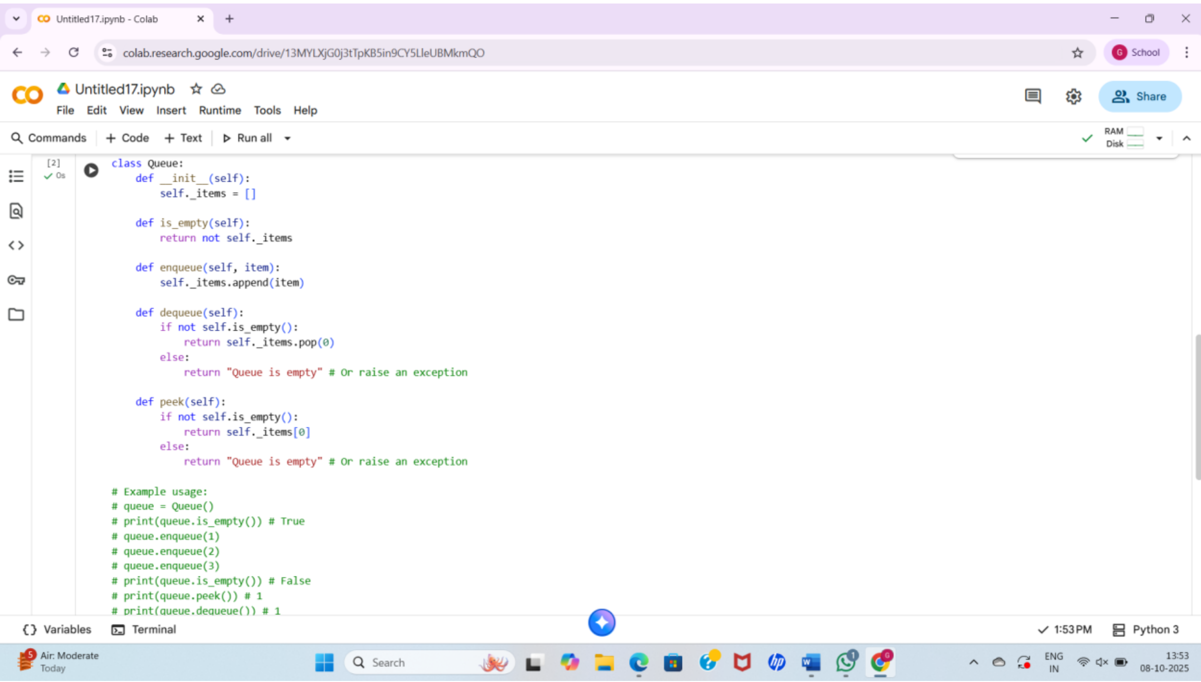
Task Description #2 – Queue Implementation

Task: Use AI to generate a Queue class with enqueue(), dequeue(), and is_empty()

Prompt

generate a Queue class with enqueue(), dequeue(), and is_empty()

code and output



Observations

- **Implementation:** Like the Stack class, it uses a Python list (_items) to store the queue elements. However, for a queue, dequeue (removing from the front) using pop(0) can be less efficient for very large queues compared to appending (enqueueing) at the end. For performance-critical applications with large queues, using collections.deque would be more suitable as it provides O(1) complexity for appending and popping from both ends.

- **Encapsulation:** Again, the use of _items with a leading underscore follows the convention for internal attributes.
- **is_empty():** This method efficiently checks if the queue is empty by checking if the _items list is empty.
- **enqueue():** This method adds an item to the rear of the queue using the list's append() method. This is an efficient operation.
- **dequeue():** This method removes and returns the item from the front of the queue using pop(0). As mentioned, this can be less efficient for large lists as it requires shifting all subsequent elements. It includes a basic check for an empty queue.
- **peek():** This method returns the item at the front of the queue without removing it. It also includes a check for an empty queue.
- **Error Handling:** Similar to the Stack class, it returns "Queue is empty" for empty queue cases in dequeue() and peek(). Raising an exception would be a more standard Pythonic approach for handling such errors.
- **Example Usage:** The commented-out example demonstrates how to use the Queue class.

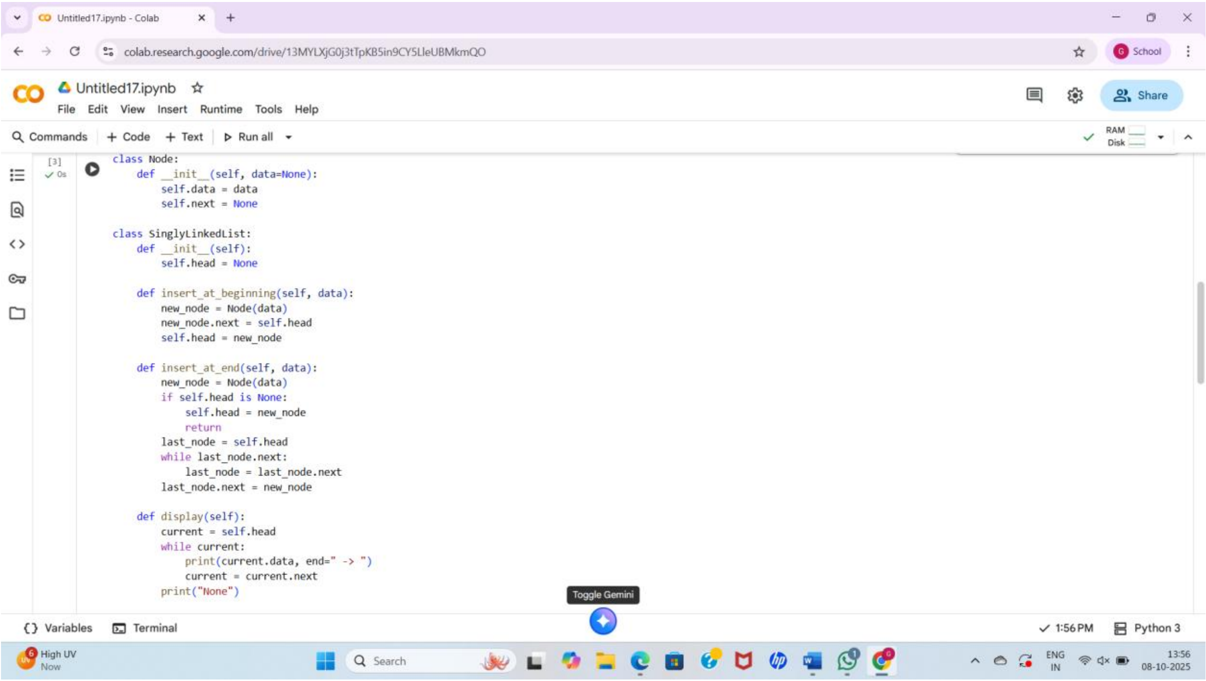
Task Description #3 – Linked List Implementation

Task: Ask AI to create a singly linked list with insert_at_end(), insert_at_beginning(), and display()

Prompt

create a singly linked list with insert_at_end(), insert_at_beginning(), and display()

code and output



Observations

- **Node Class:** The code starts by defining a Node class. This is fundamental to linked lists, where each node holds a piece of data and a reference (next) to the subsequent node in the sequence.
- **SinglyLinkedList Class:** The SinglyLinkedList class itself maintains a reference to the head of the list, which is the starting point for traversing the list.
- **__init__():** The constructor initializes the linked list with a head set to None, indicating an empty list.
- **insert_at_beginning():** This method adds a new node to the front of the list. It's efficient because it only requires updating the head reference and setting the new node's next to the original head. This operation has a time complexity of O(1).
- **insert_at_end():** This method adds a new node to the end of the list. It requires traversing the list from the head to find the last node, which can be less efficient for long lists. This operation has a time complexity of O(n), where n is the number of nodes in the list.
- **display():** This method iterates through the list from the head and prints the data of each node, followed by " -> ", until it reaches the end (where the next reference is None), finally printing "None".

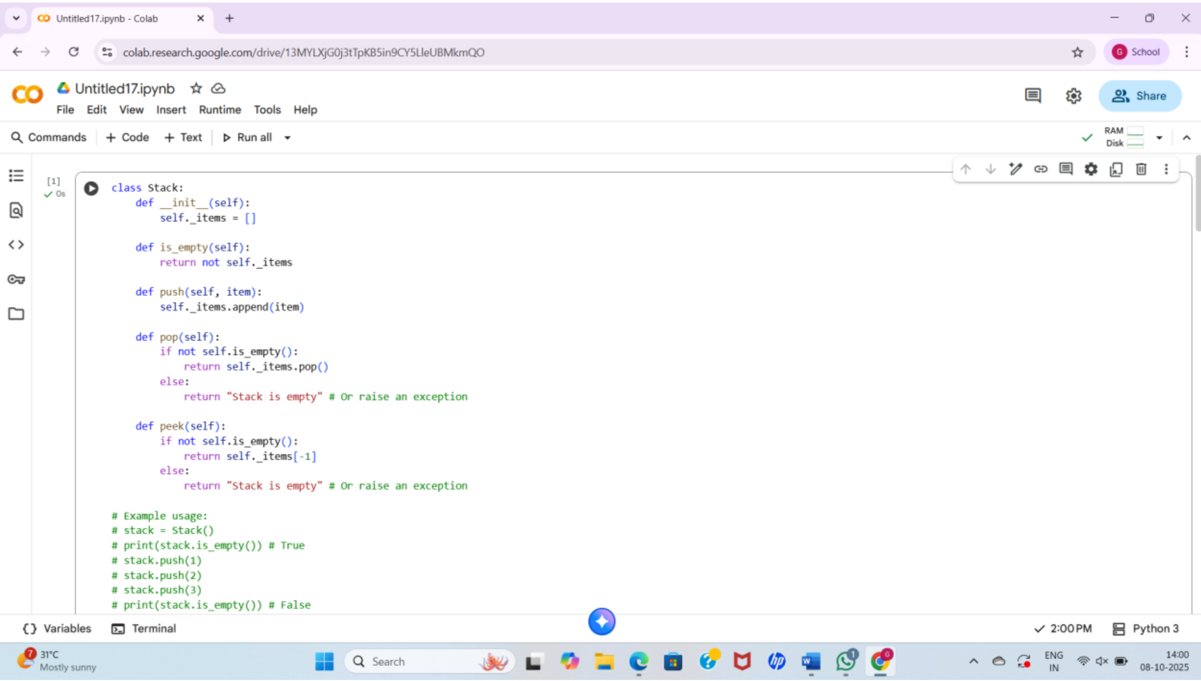
- **Dynamic Size:** Linked lists are dynamic data structures, meaning they can grow or shrink in size as needed during program execution, unlike arrays with fixed sizes.
- **Memory Allocation:** Nodes are typically allocated dynamically in memory, and they don't need to be stored in contiguous memory locations like array elements.

- ## Task Description #4 – Binary Search Tree (BST)
  Task: Ask AI to generate a simple BST with insert() and inorder_traversal().

Prompt

- generate a simple BST with insert() and inorder_traversal().

Code and output



```
class Stack:
    def __init__(self):
        self._items = []

    def is_empty(self):
        return not self._items

    def push(self, item):
        self._items.append(item)

    def pop(self):
        if not self.is_empty():
            return self._items.pop()
        else:
            return "Stack is empty" # Or raise an exception

    def peek(self):
        if not self.is_empty():
            return self._items[-1]
        else:
            return "Stack is empty" # Or raise an exception

# Example usage:
# stack = Stack()
# print(stack.is_empty()) # True
# stack.push(1)
# stack.push(2)
# stack.push(3)
# print(stack.is_empty()) # False
```

Observations

- **Node Class:** Similar to the linked list, a Node class is defined. Each node in a BST holds a value (val) and references to its left (left) and right (right) children.
- **BST Class:** The BST class maintains a reference to the root of the tree, which is the starting point for all tree operations.
- **__init__():** The constructor initializes the BST with a root set to None for an empty tree.

- **insert():** This is a public method that calls the recursive helper method _insert_rec() to insert a new key into the BST.
- **_insert_rec():** This is a private recursive helper method that performs the actual insertion. It follows the BST property: if the key is less than the current node's value, it's inserted in the left subtree; otherwise, it's inserted in the right subtree. It handles the base case where the current node is None by creating a new node.
- **inorder_traversal():** This is a public method that initiates the in-order traversal by calling the recursive helper method _inorder_traversal_rec(). It returns a list containing the elements in sorted order.
- **_inorder_traversal_rec():** This is a private recursive helper method for in-order traversal. The in-order traversal visits the left subtree, then the current node, and then the right subtree. For a BST, an in-order traversal visits the nodes in ascending order of their values.
- **Recursive Approach:** Both insertion and in-order traversal are implemented using recursion, which is a common and often elegant approach for tree operations.
- **BST Property:** The code maintains the BST property during insertion, ensuring that for any node, all values in its left subtree are less than its value, and all values in its right subtree are greater than or equal to its value.