**ASSIGNMENT – 11.4**

**NAME :** SAI CHARAN .P

**ROLL:NO :** 2403A52343

**BATCH :** AI 13

**TASK-1:**

**Prompt:-**

I’m working on a Python project and need to implement a basic Stack (LIFO) data structure.

Can you help me do the following:

* Build a Stack class with methods: push, pop, peek, and is\_empty.
* Use Google-style docstrings for documentation.
* Add inline comments explaining any non-obvious parts of the code.
* Write a small test/demo at the end using sample data to show that all methods work.

After that, could you suggest a more efficient or Pythonic way to implement a stack (maybe with collections.deque) and explain why it might be better?

**Code:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

A computer screen shot of a black screen

AI-generated content may be incorrect.

**Explanation:**

The code defines a Stack class and then demonstrates its usage with various operations. Here's a breakdown:

**Stack Class Definition:**

* **\_\_init\_\_(self)**: This is the constructor of the Stack class. It initializes an empty list called \_items which will be used to store the elements of the stack.
* **push(self, item)**: This method adds an item to the top of the stack. It uses the append() method of the \_items list, which adds the item to the end of the list. In a stack implementation using a list, the end of the list is considered the "top" of the stack.
* **pop(self)**: This method removes and returns the item from the top of the stack. It first checks if the stack is empty using is\_empty(). If not empty, it uses the pop() method of the \_items list, which removes and returns the last element (the top of the stack). If the stack is empty, it raises an IndexError with a descriptive message.
* **peek(self)**: This method returns the item from the top of the stack without removing it. Similar to pop(), it checks if the stack is empty. If not empty, it accesses the last element of the \_items list using indexing [-1]. If the stack is empty, it raises an IndexError.
* **is\_empty(self)**: This method checks if the stack is empty by checking the length of the \_items list. It returns True if the length is 0 (empty) and False otherwise.

**Test Stack Operations:**

The rest of the code demonstrates how to use the Stack class:

1. An instance of the Stack class is created.
2. is\_empty() is called to show that the stack is initially empty.
3. Several elements (an integer, a string, and a list) are added to the stack using push().
4. peek() is used to view the top element without removing it.
5. pop() is called to remove and print the top element.
6. is\_empty() is checked again to show the stack is no longer empty.
7. A while loop is used to continuously pop() elements until the stack is empty, printing each removed element.
8. Finally, try...except blocks are used to demonstrate that calling pop() or peek() on an empty stack correctly raises an IndexError.

In essence, this code provides a basic but functional implementation of a Last-In, First-Out (LIFO) stack data structure using a Python list.

**TASK-2:**

**Prompt:**

I need to implement a Queue in Python with the following methods:

Can you:

1. Help me write a simple **Queue class** using a basic Python **list**, with methods:
   * enqueue()
   * dequeue()
   * is\_empty()
2. Then, take a look at the performance of that list-based version. Let me know if there are any **efficiency issues**, especially with dequeue().
3. After that, show me how to rewrite the Queue using **collections.deque**, and explain why it’s more efficient.

Also, include:

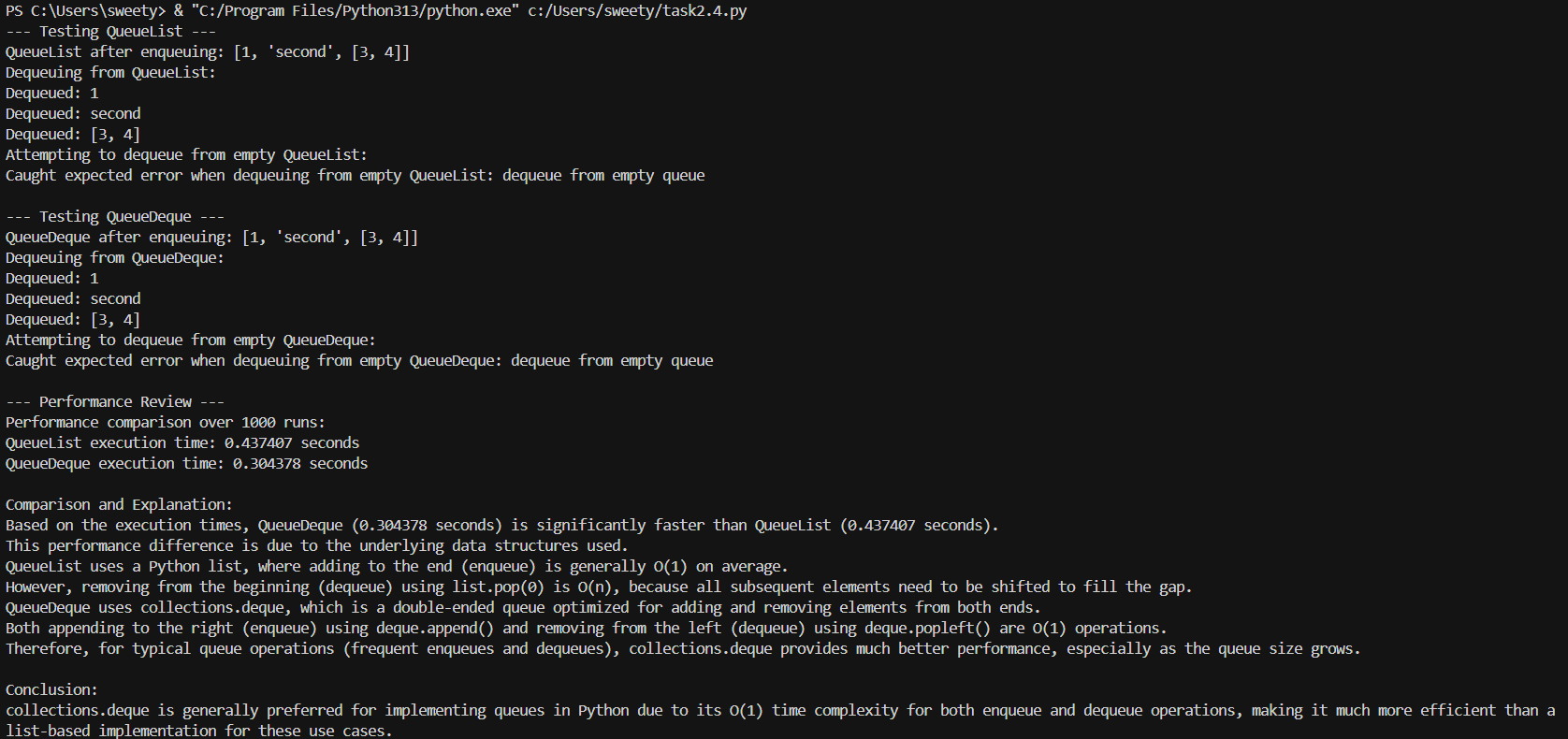
* Docstrings for each method
* Inline comments where needed
* A quick example that tests both versions

**Code:**

****



**Output:**



**Explanation:**

The code does two main things: it defines and tests two different Queue implementations (one using a Python list and one using collections.deque), and then it compares their performance using the timeit module.

Here's a breakdown:

**Queue Implementations:**

* **QueueList Class**:
  + This class implements a queue using a standard Python list (self.\_items).
  + enqueue(self, item): Adds an item to the end of the list using append(). This is generally an O(1) operation on average.
  + dequeue(self): Removes and returns the item from the beginning of the list using pop(0). **This is the key performance difference.** Removing from the start of a Python list is an O(n) operation because all subsequent elements need to be shifted one position to the left to fill the gap.
  + is\_empty(self): Checks if the list is empty by checking its length, which is an O(1) operation.
* **QueueDeque Class**:
  + This class implements a queue using collections.deque (self.\_items). A deque (double-ended queue) is optimized for adding and removing elements from both ends.
  + enqueue(self, item): Adds an item to the right side (rear) of the deque using append(). This is an O(1) operation.
  + dequeue(self): Removes and returns the item from the left side (front) of the deque using popleft(). This is also an O(1) operation.
  + is\_empty(self): Checks if the deque is empty by checking its length, which is an O(1) operation.

**Testing Queue Operations:**

The code then proceeds to test both QueueList and QueueDeque:

1. Instances of both classes are created.
2. Elements are enqueued into each queue.
3. Elements are dequeued from each queue, and the dequeued elements are printed.
4. try...except blocks are used to demonstrate that attempting to dequeue from an empty queue raises an IndexError in both implementations.

**Performance Review using timeit:**

This section measures and compares the execution time of enqueueing and dequeuing a large number of elements (1000 in this case) for both QueueList and QueueDeque over a specified number of runs (1000 runs).

1. **operations\_list() and operations\_deque() functions**: These functions define the sequence of operations (enqueueing 1000 items and then dequeuing all of them) for each queue implementation.
2. **timeit.timeit()**: This function is used to measure the execution time of the provided functions (operations\_list and operations\_deque) over num\_runs.
3. **Printing Results**: The execution times for both implementations are printed.
4. **Comparison and Explanation**: The code then provides a summary comparing the execution times and explains the reason for the performance difference. It highlights that QueueDeque is significantly faster because its dequeue operation (popleft()) is O(1), while QueueList's dequeue (pop(0)) is O(n), making the list-based implementation much slower for large queues.

In summary, this code effectively demonstrates the functionality of both list-based and deque-based queues and provides a clear performance comparison, illustrating why collections.deque is the preferred choice for efficient queue implementations in Python.

**TASK-3:**

**Prompt:**

I’m building a Singly Linked List in Python and need help with the following:

Here’s what I need:

1. Start with the basics — create a Node class and a LinkedList class.
2. Add the following methods to the linked list:

* insert\_at\_end(value)
* delete\_value(value)
* traverse()

Please add **detailed inline comments** to explain what’s happening during pointer changes — especially when inserting or deleting nodes. That part confuses me the most.

Also, could you include:

* Docstrings for each method (Google-style)
* Some suggested **test cases** to check if the code works properly — including edge cases like deleting from an empty list or deleting the first node.
* A small example at the end that runs each method and prints the list

The goal is to understand both how it works and how to test it properly.

**Code:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Explanation:**

The code provides a complete implementation of a Singly Linked List in Python, along with test cases to demonstrate its functionality.

Here's a breakdown:

1. Node Class:

* This is a simple class that represents a single node within the linked list.
* \_\_init\_\_(self, data, next\_node=None): The constructor takes data (the value to be stored in the node) and an optional next\_node argument, which is a reference to the next node in the sequence. By default, next\_node is set to None, indicating the end of a list or sublist.
* self.data = data: Stores the data passed to the constructor.
* self.next = next\_node: Stores the reference to the next node. This is the "pointer" that links nodes together.

2. LinkedList Class:

* This class represents the entire singly linked list.
* \_\_init\_\_(self): The constructor initializes the linked list by setting the head attribute to None. The head is a reference to the first node in the list. An empty list has no head.
* insert\_at\_end(self, data):
  + Creates a new\_node with the given data.
  + If the list is empty (self.head is None), the new\_node becomes the head.
  + If the list is not empty, it traverses the list starting from the head until it reaches the last\_node (the node whose next is None).
  + It then updates the next pointer of the last\_node to point to the new\_node, effectively adding the new node to the end. Inline comments explain these pointer updates.
* delete\_value(self, value):
  + This method deletes the *first* occurrence of a node with the specified value.
  + It first checks if the head node contains the value and, if so, updates the head to the next node, effectively removing the original head.
  + If the value is not in the head, it traverses the list, keeping track of the current\_node and the prev\_node.
  + When the current\_node's data matches the value, it updates the next pointer of the prev\_node to point to the current\_node's next, thereby unlinking the current\_node from the list. Inline comments explain these pointer updates.
  + If the loop finishes without finding the value, it means the value was not in the list, and the method simply returns.
* traverse(self):
  + This method iterates through the linked list starting from the head.
  + For each current\_node, it prints the data followed by " -> ".
  + It moves to the next node by updating current\_node to current\_node.next.
  + It prints "None" at the end to indicate the end of the list. Inline comments explain the pointer updates during traversal.

3. Test Cases:

The code includes a comprehensive set of test cases to validate the functionality of the insert\_at\_end, delete\_value, and traverse methods. These tests cover various scenarios, including:

* Inserting into an empty list and a non-empty list.
* Consecutive insertions.
* Deleting from an empty list.
* Deleting the head, a middle node, and the tail node.
* Deleting a value that is not in the list.
* Deleting the first occurrence of a duplicate value.
* Traversing an empty list, a single-element list, and a multi-element list.

Each test case is clearly labeled and includes print statements to show the state of the list before and after the operation, making it easy to verify the correctness of the implementation.

In summary, this code provides a well-structured and tested implementation of a singly linked list, with clear explanations of the pointer manipulations that are fundamental to this data structure.

**TASK-4:**

**Prompt:**

I’m working on implementing a Binary Search Tree (BST) in Python. I’ve started writing the Node and BST classes but haven’t finished all the methods yet.

Here's what I need help with:

1. Complete the following missing methods in the BST class:

* insert(value) – insert a value into the correct position in the tree
* search(value) – return True if the value is found, otherwise False
* inorder\_traversal() – return a list of values in in-order sequence

1. Add Google-style docstrings to all methods and include inline comments where the logic might be tricky (like recursive insert/search).
2. Provide a test example at the end where:

* A list of integers is inserted
* The search() method is used to find both present and absent values
* The inorder\_traversal() output is printed to confirm sorting

I can provide a partial class if you want — otherwise, feel free to start from scratch and show the full implementation.

**Code:**

**A screen shot of a computer screen

AI-generated content may be incorrect.**

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Explanation:**

The code provides a Python implementation of a Binary Search Tree (BST) and demonstrates its core operations: insertion, searching, and in-order traversal.

Here's a breakdown:

1. Node Class:

* This class represents a single node within the BST.
* \_\_init\_\_(self, key): The constructor takes a key as input, which is the value stored in the node.
* self.key = key: Stores the value.
* self.left = None: Initializes the left child reference to None. In a BST, the left child's key is always less than the parent's key.
* self.right = None: Initializes the right child reference to None. In a BST, the right child's key is always greater than or equal to the parent's key (this implementation places equal keys in the right subtree).

2. BST Class:

* This class represents the entire Binary Search Tree.
* \_\_init\_\_(self): The constructor initializes the BST by setting the root attribute to None. The root is the topmost node of the tree. An empty tree has no root.
* insert(self, key):
  + This method inserts a new node with the given key into the BST while maintaining the BST properties.
  + If the tree is empty, the new node becomes the root.
  + If the tree is not empty, it uses a recursive helper function \_insert\_recursive.
  + \_insert\_recursive(current\_node, key): This function traverses the tree starting from current\_node.
    - If the key is less than the current\_node's key, it goes to the left subtree. If the left child is None, the new node is inserted there. Otherwise, it recursively calls itself on the left child.
    - If the key is greater than or equal to the current\_node's key, it goes to the right subtree. If the right child is None, the new node is inserted there. Otherwise, it recursively calls itself on the right child.
* search(self, key):
  + This method searches for a node with the given key in the BST.
  + It uses a recursive helper function \_search\_recursive.
  + \_search\_recursive(current\_node, key): This function traverses the tree.
    - If the current\_node is None (meaning the key was not found) or the current\_node's key matches the key, it returns the current\_node (or None).
    - If the key is less than the current\_node's key, it recursively searches in the left subtree.
    - If the key is greater than the current\_node's key, it recursively searches in the right subtree.
  + The method returns the found Node object or None if the key is not present.
* inorder\_traversal(self):
  + This method performs an in-order traversal of the BST, which visits the left subtree, then the current node, and then the right subtree. This traversal order results in visiting the nodes in ascending order of their keys.
  + It uses a recursive helper function \_inorder\_recursive.
  + \_inorder\_recursive(current\_node):
    - If the current\_node is not None, it recursively calls itself on the left child, then appends the current\_node's key to a result list, and finally recursively calls itself on the right child.
  + The method returns the result list containing the keys in sorted order.

Test BST Operations:

The latter part of the code demonstrates the usage of the BST class:

1. An instance of the BST class is created.
2. A list of integers is defined to be inserted into the tree.
3. Each integer from the list is inserted into the BST using the insert method.
4. inorder\_traversal() is called, and the resulting sorted list of keys is printed to verify the structure.
5. The search() method is called with a value known to be present in the tree, and the result is printed.
6. The search() method is called with a value known to be absent from the tree, and the result is printed.

This code provides a solid foundation for working with Binary Search Trees, illustrating how to build, search, and traverse this important data structure.

**TASK-5:**

**Prompt:**

I’m implementing a Graph in Python using an adjacency list (dictionary-based) representation.

Can you help me do the following:

1. Create a graph structure using a dictionary to store adjacency lists.
2. Implement two traversal methods:

* bfs(start\_node) – Breadth-First Search
* dfs(start\_node) – Depth-First Search

1. Include inline comments that explain the key steps during traversal (e.g., how nodes are visited, queued/stacked, and marked as visited).
2. If there are different ways to implement DFS (e.g. recursive vs iterative), briefly compare them and suggest when to use which.
3. Provide a small example graph and show the output of both BFS and DFS starting from a sample node.

Also, please include:

* Google-style docstrings for the methods
* Clear, readable code
* Any helpful notes about time/space complexity or limitations of each method

**Code:**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

**Output:**

A computer screen with white text

AI-generated content may be incorrect.

**Explanation:**

The code provides a Python implementation of a Graph data structure using an adjacency list and includes methods for Breadth-First Search (BFS) and Depth-First Search (DFS) traversals.

Here's a breakdown:

**1. Graph Class:**

* This class represents the graph.
* \_\_init\_\_(self): The constructor initializes an empty dictionary self.\_graph. This dictionary serves as the adjacency list, where keys represent vertices and their corresponding values are lists of neighboring vertices.
* **add\_edge(self, u, v)**:
  + This method adds an undirected edge between vertices u and v.
  + It checks if u and v are already keys in self.\_graph. If not, they are added with empty lists as their initial neighbors.
  + It then appends v to the list of neighbors for u and u to the list of neighbors for v, representing the connection in both directions for an undirected graph.
* **bfs(self, start\_node)**:
  + This method performs a Breadth-First Search starting from the start\_node.
  + It first checks if the start\_node exists in the graph.
  + It initializes a deque (double-ended queue) called queue and adds the start\_node to it. BFS uses a queue to explore nodes level by level.
  + It initializes a set called visited to keep track of visited nodes and adds the start\_node to it.
  + The while queue: loop continues as long as there are nodes to visit.
  + Inside the loop, it dequeues a current\_node from the front of the queue.
  + It prints the current\_node.
  + It then iterates through the neighbors of the current\_node (found in self.\_graph[current\_node]).
  + For each neighbor, if it hasn't been visited, it's added to the visited set and enqueued. Inline comments explain these steps.
* **dfs(self, start\_node)**:
  + This method performs a Depth-First Search starting from the start\_node.
  + It first checks if the start\_node exists in the graph.
  + It initializes an empty set called visited.
  + It uses a recursive helper function \_dfs\_recursive(current\_node, visited). DFS naturally lends itself to a recursive implementation, using the call stack to manage the traversal depth.
  + \_dfs\_recursive(current\_node, visited):
    - It marks the current\_node as visited.
    - It prints the current\_node.
    - It then iterates through the neighbors of the current\_node.
    - For each neighbor that hasn't been visited, it recursively calls \_dfs\_recursive on that neighbor. Inline comments explain these steps.
  + The initial call to \_dfs\_recursive starts the traversal from the start\_node.

**Test Graph Traversals:**

The latter part of the code demonstrates the usage of the Graph class and its traversal methods:

1. An instance of the Graph class is created.
2. Edges are added to the graph using the add\_edge method to create a sample graph structure, including a disconnected component (G and H). The resulting adjacency list is printed.
3. BFS traversal is performed starting from node 'A', and the output is printed.
4. DFS traversal is performed starting from node 'A', and the output is printed.
5. BFS and DFS traversals are also performed starting from the disconnected node 'G' to show how the algorithms handle disconnected components.
6. Finally, attempts are made to perform BFS and DFS from a node ('Z') that does not exist in the graph to show the error handling.

This code provides a clear and functional implementation of a graph with BFS and DFS traversals, illustrating how these algorithms explore the connections within a graph.