**### Stack Overview**

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added is the first one removed. It is commonly used in scenarios like reversing a sequence, managing recursive function calls, and evaluating expressions.

**#### Common Stack Operations:**

1. Push: Add an element to the top of the stack.

2. Pop: Remove the element from the top of the stack.

3. Peek/Top: View the top element without removing it.

4. Is\_empty: Check if the stack is empty.

5. Size: Return the number of elements in the stack.

**#### Why the Iterator Concept is Not Applied in Stack**

Stacks follow LIFO, meaning we only access the top element, not iterate through all elements in sequence. While stacks can technically use iterators, it’s uncommon because stacks are designed for restricted access, focusing on adding, removing, and viewing the top item rather than sequential traversal.

**### Assignment Notes and Implementations**

**#### Assignment 8: Stack Implementation Using List (Without Inheritance)**

**1. Define `Stack` Class Using List**

Initialize an empty list in the constructor (`\_\_init\_\_`) to serve as the stack’s container.

```python

class Stack:

def \_\_init\_\_(self):

self.items = [] # List to store stack elements

**2. Define `is\_empty()` Method**

Check if the list is empty to determine if the stack has no elements.

```python

def is\_empty(self):

return len(self.items) == 0

**3. Define `push(data)` Method**

Append new elements to the end of the list to add them to the stack.

```python

def push(self, data):

self.items.append(data)

**4. Define `pop()` Method**

Remove and return the last element if the stack is not empty.

```python

def pop(self):

if not self.is\_empty():

return self.items.pop()

raise IndexError("Stack is empty")

**5. Define `peek()` Method**

Return the last element without removing it.

```python

def peek(self):

if not self.is\_empty():

return self.items[-1]

raise IndexError("Stack is empty")

**6. Define `size()` Method**

Return the number of elements in the stack.

```python

def size(self):

return len(self.items)

**#### Assignment 8 (Extended): Stack Extending List**

**1. Define `Stack` Class by Extending List**

Inherit from the `list` class to extend its methods and restrict access to certain methods like `insert()`.

```python

class Stack(list):

def \_\_init\_\_(self):

super().\_\_init\_\_()

**2. Restrict `insert()` Method**

Override the `insert()` method to prevent use, ensuring LIFO access.

```python

def insert(self, args, kwargs):

raise NotImplementedError("Use push() instead of insert() in stack")

**#### Assignment 9 & 10: Stack Using Linked List**

**1. Define `Stack` Class Using Linked List**

Initialize a singly linked list structure with `start` as the top node and `item\_count` to track size.

```python

class Stack:

def \_\_init\_\_(self):

self.start = None # Reference to the top of the stack

self.item\_count = 0

**2. Define `push(data)` Method**

Insert a new node at the beginning.

```python

def push(self, data):

new\_node = Node(data)

new\_node.next = self.start

self.start = new\_node

self.item\_count += 1

**3. Define `pop()` Method**

Remove the top node and adjust the `start` reference.

```python

def pop(self):

if not self.is\_empty():

data = self.start.data

self.start = self.start.next

self.item\_count -= 1

return data

raise IndexError("Stack is empty")

**#### Assignment 11: Stack by Extending Singly Linked List**

**1. Define `Stack` Class by Extending SLL**

Inherit from the `SLL` class to use its linked list functionalities.

```python

from linkedlist import SLL

class Stack(SLL):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.item\_count = 0

**2. Define `push(data)` Method**

Use `insert\_at\_start` from the `SLL` class.

```python

def push(self, data):

self.insert\_at\_start(data)

self.item\_count += 1

**3. Restrict Access to Other `SLL` Methods**

Override any additional linked list methods that shouldn’t be used in the stack context.

**### Key Takeaways:**

1. Stacks operate on a LIFO principle, emphasizing top-focused operations like `push` and `pop`.

2. While stacks can technically support iteration, the concept doesn’t align with their LIFO design.

3. Using lists or linked lists for implementing stacks depends on use-case constraints, with lists often preferred for simplicity and linked lists for dynamic sizing.

Here’s the code rewritten with proper comments and corrections for clarity. Each section is commented to explain its purpose for beginners.

```python

# Define a class for a single node in the stack

class Node:

def \_\_init\_\_(self, item=None, next=None):

# Initialize a node with a value (item) and a reference to the next node

self.item = item

self.next = next

# Define a stack class using linked nodes

class Stack:

def \_\_init\_\_(self):

# Initialize an empty stack; start is the top of the stack

self.start = None

# Keep track of the number of items in the stack

self.item\_count = 0

# Check if the stack is empty

def is\_empty(self):

# Return True if start is None, meaning the stack has no items

return self.start is None

# Add an item to the stack (push operation)

def push(self, data):

# Create a new node with the data, pointing to the current start node

new\_node = Node(data, self.start)

# Make the new node the top of the stack

self.start = new\_node

# Increment the item count

self.item\_count += 1

# Remove and return the top item from the stack (pop operation)

def pop(self):

# Check if the stack is not empty before popping

if not self.is\_empty():

# Get the item at the top of the stack

data = self.start.item

# Move start to the next node, effectively removing the top item

self.start = self.start.next

# Decrease the item count

self.item\_count -= 1

# Return the popped item

return data

else:

# Raise an error if trying to pop from an empty stack

raise IndexError("Stack is empty")

# Return the top item of the stack without removing it (peek operation)

def peek(self):

# Check if the stack is not empty before peeking

if not self.is\_empty():

# Return the item at the top of the stack

return self.start.item

else:

# Raise an error if trying to peek into an empty stack

raise IndexError("Stack is empty")

# Return the number of items in the stack

def size(self):

return self.item\_count

# Create a new stack instance

s1 = Stack()

# Push items onto the stack

s1.push(10)

s1.push(20)

s1.push(30)

# Display the total number of items in the stack

print("Total elements in the stack =", s1.size())

# Display the top item in the stack without removing it

print("Top element on the stack is", s1.peek())

# Pop the top item from the stack and display it

print("Popped element is", s1.pop())

# Display the total number of items in the stack after popping

print("Total elements in the stack =", s1.size())

# Display the new top item in the stack

print("Top element on the stack is", s1.peek())

```

### Explanation of the Main Sections:

- \*\*Node Class\*\*: Represents each element in the stack with `item` as data and `next` as the link to the next node.

- \*\*Stack Class\*\*: Implements the stack using linked nodes, with methods to check if the stack is empty, push items, pop items, peek at the top item, and get the size.

- \*\*Main Code\*\*: Demonstrates pushing, peeking, and popping items from the stack, along with getting the current size.

class Node:

"""Represents a node in a singly linked list."""

def \_\_init\_\_(self, data):

# Initialize a node with data and set the next node to None initially

self.data = data

self.next = None

class SLL:

"""Represents a singly linked list."""

def \_\_init\_\_(self):

# Initialize the start (head) of the list to None, indicating an empty list

self.start = None

def is\_empty(self):

"""Check if the linked list is empty."""

return self.start is None

def insert\_at\_start(self, data):

"""Insert an element at the beginning of the list."""

new\_node = Node(data) # Create a new node with the given data

new\_node.next = self.start # Link new node to the current start

self.start = new\_node # Update start to the new node

def insert\_at\_last(self, data):

"""Insert an element at the end of the list."""

new\_node = Node(data) # Create a new node with the given data

if self.is\_empty(): # If the list is empty, make it the start node

self.start = new\_node

else:

current = self.start

# Traverse to the last node

while current.next:

current = current.next

current.next = new\_node # Link last node to the new node

def search(self, data):

"""Search for a node with the specified data."""

current = self.start # Start from the first node

while current:

if current.data == data: # If node with the data is found, return it

return current

current = current.next # Move to the next node

return None # Return None if data is not found

def insert\_after(self, prev\_data, data):

"""Insert a new node after the node with the given previous data."""

current = self.search(prev\_data) # Find the node with prev\_data

if current is None: # If not found, print a message

print(f"Node with data {prev\_data} not found.")

return

new\_node = Node(data) # Create a new node

new\_node.next = current.next # Link the new node to the next node

current.next = new\_node # Link the previous node to the new node

def print\_list(self):

"""Print all the elements of the list."""

current = self.start # Start from the first node

while current:

print(current.data, end=" -> ")

current = current.next # Move to the next node

print("None") # End of list

def \_\_iter\_\_(self):

"""Iterator to access elements of the list in a sequence."""

current = self.start

while current:

yield current.data # Yield the data of each node

current = current.next

def delete\_first(self):

"""Delete the first element from the list."""

if self.is\_empty(): # Check if the list is empty

print("List is empty. No element to delete.")

return

self.start = self.start.next # Move start to the next node

def delete\_last(self):

"""Delete the last element from the list."""

if self.is\_empty(): # Check if the list is empty

print("List is empty. No element to delete.")

return

if self.start.next is None: # If only one element is in the list

self.start = None

else:

current = self.start

# Traverse to the second-last node

while current.next.next:

current = current.next

current.next = None # Remove the last node

# Example usage

sll = SLL()

sll.insert\_at\_start(10)

sll.insert\_at\_start(5)

sll.insert\_at\_last(20)

sll.insert\_at\_last(25)

sll.insert\_after(10, 15)

print("Linked List Elements:")

sll.print\_list()

print("After deleting the first element:")

sll.delete\_first()

sll.print\_list()

print("After deleting the last element:")

sll.delete\_last()

sll.print\_list()

print("Elements in the list using iterator:")

for data in sll:

print(data, end=" ")

Let's correct and add comments to each part of the two code files you provided.

### Corrected Code for `SLL` (Singly Linked List) and Comments:

```python

class Node:

"""Represents a node in a singly linked list."""

def \_\_init\_\_(self, data):

# Initialize a node with data and set the next node to None initially

self.data = data

self.next = None

class SLL:

"""Represents a singly linked list."""

def \_\_init\_\_(self):

# Initialize the start (head) of the list to None, indicating an empty list

self.start = None

def is\_empty(self):

"""Check if the linked list is empty."""

return self.start is None

def insert\_at\_start(self, data):

"""Insert an element at the beginning of the list."""

new\_node = Node(data) # Create a new node with the given data

new\_node.next = self.start # Link new node to the current start

self.start = new\_node # Update start to the new node

def insert\_at\_last(self, data):

"""Insert an element at the end of the list."""

new\_node = Node(data) # Create a new node with the given data

if self.is\_empty(): # If the list is empty, make it the start node

self.start = new\_node

else:

current = self.start

# Traverse to the last node

while current.next:

current = current.next

current.next = new\_node # Link last node to the new node

def search(self, data):

"""Search for a node with the specified data."""

current = self.start # Start from the first node

while current:

if current.data == data: # If node with the data is found, return it

return current

current = current.next # Move to the next node

return None # Return None if data is not found

def insert\_after(self, prev\_data, data):

"""Insert a new node after the node with the given previous data."""

current = self.search(prev\_data) # Find the node with prev\_data

if current is None: # If not found, print a message

print(f"Node with data {prev\_data} not found.")

return

new\_node = Node(data) # Create a new node

new\_node.next = current.next # Link the new node to the next node

current.next = new\_node # Link the previous node to the new node

def print\_list(self):

"""Print all the elements of the list."""

current = self.start # Start from the first node

while current:

print(current.data, end=" -> ")

current = current.next # Move to the next node

print("None") # End of list

def \_\_iter\_\_(self):

"""Iterator to access elements of the list in a sequence."""

current = self.start

while current:

yield current.data # Yield the data of each node

current = current.next

def delete\_first(self):

"""Delete the first element from the list."""

if self.is\_empty(): # Check if the list is empty

print("List is empty. No element to delete.")

return

self.start = self.start.next # Move start to the next node

def delete\_last(self):

"""Delete the last element from the list."""

if self.is\_empty(): # Check if the list is empty

print("List is empty. No element to delete.")

return

if self.start.next is None: # If only one element is in the list

self.start = None

else:

current = self.start

# Traverse to the second-last node

while current.next.next:

current = current.next

current.next = None # Remove the last node

# Example usage

sll = SLL()

sll.insert\_at\_start(10)

sll.insert\_at\_start(5)

sll.insert\_at\_last(20)

sll.insert\_at\_last(25)

sll.insert\_after(10, 15)

print("Linked List Elements:")

sll.print\_list()

print("After deleting the first element:")

sll.delete\_first()

sll.print\_list()

print("After deleting the last element:")

sll.delete\_last()

sll.print\_list()

print("Elements in the list using iterator:")

for data in sll:

print(data, end=" ")

```

---

### Corrected Code for `stack` Class Using `SLL` and Comments:

```python

from linkedlist import SLL # Assuming SLL is in a separate file named linkedlist.py

class Stack:

"""Represents a stack using a singly linked list (SLL)."""

def \_\_init\_\_(self):

# Initialize an empty singly linked list to act as the stack

self.myList = SLL()

# Keep track of the number of items in the stack

self.item\_count = 0

def is\_empty(self):

"""Check if the stack is empty."""

return self.myList.is\_empty()

def push(self, data):

"""Push (add) an item to the top of the stack."""

# Insert at the start of the linked list for stack's push operation

self.myList.insert\_at\_start(data)

self.item\_count += 1 # Increment item count

def pop(self):

"""Remove and return the top item from the stack."""

if not self.is\_empty(): # Check if stack is not empty

top\_data = self.myList.start.data # Get data from top node

self.myList.delete\_first() # Remove the top node

self.item\_count -= 1 # Decrement item count

return top\_data

else:

# Raise an error if trying to pop from an empty stack

raise IndexError("Stack is empty")

def peek(self):

"""Return the top item without removing it from the stack."""

if not self.is\_empty(): # Check if stack is not empty

return self.myList.start.data # Return the data of the top node

else:

# Raise an error if stack is empty

raise IndexError("Stack is empty")

def size(self):

"""Return the number of items in the stack."""

return self.item\_count

# Example usage

s = Stack()

s.push(10)

s.push(20)

s.push(30)

print("Top element is", s.peek()) # Should display the top element, 30

print("Popped element is", s.pop()) # Should remove and return 30

print("Top element after pop is", s.peek()) # Should display the new top element, 20

```

### Explanation of `Stack` Code:

- \*\*Constructor\*\*: Initializes the stack with an empty singly linked list.

- \*\*Push\*\*: Adds an item to the top of the stack by inserting at the start of the linked list.

- \*\*Pop\*\*: Removes and returns the top item by deleting the first node in the linked list.

- \*\*Peek\*\*: Returns the top item without removing it.

- \*\*Size\*\*: Returns the number of items in the stack.