**Let’s create Python’s List with functionalities**

Source Code 👻

'''

import sys

L = []

# dynamic array expanding array size by 8 >>>

for i in range(100):

    sys.getsizeof(L)

    L.append(i)

'''

'''

ctypes is a foreign function library for Python. It provides

 C compatible data types, and allows calling functions in DLLs or

 shared libraries. It can be used to wrap these libraries in pure

 Python.

'''

# using c's array we are creaging python's list

import ctypes

class MeraList:

    def \_\_init\_\_(self):

        self.size = 1

        self.n = 0

        # create a C type array with size = self.sizse

        self.A = self.\_\_make\_array(self.size)

    def \_\_len\_\_(self):

        return self.n

    def \_\_str\_\_(self):

        #[1,2,3]

        result = ''

        for i in range(self.n):

            result += str(self.A[i]) + ' , '

        return '[ ' + result[:-2] + ']'

    def \_\_getitem\_\_(self,index):

        if 0 <= index < self.n:

            return self.A[index]

        else :

            return 'IndexError - Index out of range'

    def append(self,item):

        if self.n == self.size: # need to resize

            # resize

            self.\_\_resize(self.size \*2 )

        # append

        self.A[self.n] = item

        self.n += 1

    def pop(self):

        if self.n == 0:

            return "Empty List"

        print(self.A[self.n-1])

        self.n -= 1

    def clear(self):

        self.n = 0

        self.size = 1

    def find(self,item):

        for i in range(self.n):

            if self.A[i] == item :

                return i

        return 'ValueError - not in list'

    def \_\_delitem\_\_(self,index):

        if index >= self.n or index < 0 :

            return 'IndexError - not vlaid index'

        # delte from the position

        for i in range(index,self.n-1):

            self.A[i] = self.A[i+1]

        self.n -= 1

    def insert(self,index,item):

        if self.size == self.n :

            self.\_\_resize(self.size \* 2)

        for i in range(self.n, index,-1):

            self.A[i] = self.A[i-1]

        self.A[index] = item

        self.n += 1

    def remove(self,item):

        for i in range(self.n-1):

            if self.A[i] == item:

                self.\_\_delitem\_\_(i)

                return

        return "ValueError - item not found "

    def \_\_resize(self,new\_capacity):

        # create a new array with new capacity

        B = self.\_\_make\_array(new\_capacity)

        self.size = new\_capacity

        # copy the content of A to B

        for i in range(self.n):

            B[i] = self.A[i]

        # reassign A

        self.A = B

    def \_\_make\_array(self,capacity):

        # this code creates a C types (static,referential array) with size capacity

        return (capacity\*ctypes.py\_object)()

L = MeraList()

L.append('Pranav')

L.append(3.4)

L.append(True)

L.append(100)

L.append('infinite')

L.append('infinity')

print("length of L : " , len(L))

print(L)

# print(L[0])

# L.pop()

L.insert(2,'pooja')

# print(L.find('Pranav')) # found in position index 0

print(L)

print(L.\_\_delitem\_\_(500))

print(L)

print(L.remove('infinite'))

print(L.remove('infinite'))

print(L)

Explanation 👻👻👀👀✅😍

This code provides a detailed look into implementing a custom dynamic array in Python, similar to Python’s built-in list but with a more granular, lower-level understanding of resizing and memory allocation. Let's break it down thoroughly to explain each part and the areas where someone might get confused.

**Part 1: Basic Setup and List Size Expansion**

import sys

L = []

# dynamic array expanding array size by 8 >>>

for i in range(100):

sys.getsizeof(L)

L.append(i)

Here, a regular Python list (L) is created and populated with values from 0 to 99. However, the sys.getsizeof(L) is called for each iteration but does nothing with the output. The purpose of calling sys.getsizeof(L) might be to check the memory usage of L as it expands, because Python lists automatically resize when their capacity is exceeded.

**Key Point to Note**:

* **List Resizing**: Python lists resize automatically by allocating more memory as needed, which is not shown here but can be observed by printing sys.getsizeof(L) in each iteration.
* **Confusion Point**: It may be unclear why sys.getsizeof(L) is used without any output. Printing this value would show the dynamic resizing and memory usage of the list.

**Part 2: Creating a Custom Dynamic Array with ctypes**

The main part of the code uses ctypes, a foreign function library in Python that allows for the creation of low-level C-compatible arrays. Using ctypes, we can create a more controlled list implementation (MeraList) that expands its size like a dynamic array in C++.

**Explanation of MeraList Components**:

**1. Initialization (\_\_init\_\_)**

class MeraList:

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

self.size = 1 # Initial capacity of the list

self.n = 0 # Number of elements in the list

self.A = self.\_\_make\_array(self.size)

* self.size: Initial capacity (starts at 1).
* self.n: Tracks the actual number of elements in the list.
* self.A: The underlying array is created by calling self.\_\_make\_array(self.size), which allocates a C-type array with the given capacity.

**2. Length Method (\_\_len\_\_)**

def \_\_len\_\_(self):

return self.n

This method simply returns the count of elements in the list.

**3. String Representation (\_\_str\_\_)**

def \_\_str\_\_(self):

result = ''

for i in range(self.n):

result += str(self.A[i]) + ' , '

return '[ ' + result[:-2] + ']'

This method allows for a readable string representation of the list, removing the last comma and space for neat output.

**4. Accessing an Element (\_\_getitem\_\_)**

def \_\_getitem\_\_(self, index):

if 0 <= index < self.n:

return self.A[index]

else:

return 'IndexError - Index out of range'

This function returns the item at a specified index if it's valid. Otherwise, it returns an error message.

**Potential Confusion**:

* Python usually raises an IndexError directly, but here the code returns a string message instead, which could be unexpected for Python users.

**Part 3: Adding and Removing Elements**

**Append Method**

def append(self, item):

if self.n == self.size: # If array is full, resize it

self.\_\_resize(self.size \* 2)

self.A[self.n] = item

self.n += 1

* Checks if there is space to add a new element. If not, it doubles the size of the array by calling self.\_\_resize(self.size \* 2).
* Appends the new item and increments the element count n.

**Pop Method**

def pop(self):

if self.n == 0:

return "Empty List"

print(self.A[self.n-1])

self.n -= 1

* Decrements the count, effectively removing the last item.
* **Confusion Point**: This doesn’t handle reducing the array’s capacity when elements are removed, so even if elements are popped, memory allocation remains high.

**Part 4: Additional List Functionality**

**Clear Method**

def clear(self):

self.n = 0

self.size = 1

* Resets the list by setting n to 0 and resizing size back to 1.

**Find Method**

def find(self, item):

for i in range(self.n):

if self.A[i] == item:

return i

return 'ValueError - not in list'

* Searches for the first occurrence of an item.
* Returns the index if found, otherwise an error message.

**Delete by Index (\_\_delitem\_\_)**

def \_\_delitem\_\_(self, index):

if index >= self.n or index < 0:

return 'IndexError - not valid index'

for i in range(index, self.n - 1):

self.A[i] = self.A[i + 1]

self.n -= 1

* Shifts elements left to remove an item at a specified index.
* **Confusion Point**: This operation has O(n) complexity due to the shifting, which could be slow for large lists.

**Insert Method**

def insert(self, index, item):

if self.size == self.n:

self.\_\_resize(self.size \* 2)

for i in range(self.n, index, -1):

self.A[i] = self.A[i - 1]

self.A[index] = item

self.n += 1

* Expands the array if needed, shifts elements to the right, and inserts the new item at the given index.
* **Confusion Point**: Similar to \_\_delitem\_\_, this operation has O(n) complexity due to shifting.

**Remove Method**

def remove(self, item):

for i in range(self.n - 1):

if self.A[i] == item:

self.\_\_delitem\_\_(i)

return

return "ValueError - item not found"

* Calls \_\_delitem\_\_ on the first occurrence of the specified item.
* If the item is not found, it returns a message indicating that.

**Part 5: Internal Helper Methods for Resizing**

**Resize Method**

def \_\_resize(self, new\_capacity):

B = self.\_\_make\_array(new\_capacity)

self.size = new\_capacity

for i in range(self.n):

B[i] = self.A[i]

self.A = B

* Creates a new, larger array with new\_capacity and copies elements from the old array.
* Sets the list’s array reference to the new array.

**Key Point**:

* \_\_resize handles dynamic resizing, similar to how Python’s list resizes automatically. This operation has an O(n) complexity due to the copying step.

**Creating a C Array (\_\_make\_array)**

def \_\_make\_array(self, capacity):

return (capacity \* ctypes.py\_object)()

* Creates a C-type array of pointers, with each pointer referring to a Python object.
* **Confusion Point**: Understanding ctypes and why it’s used might be challenging for beginners. Here, ctypes is used for more control over memory management.

**Example Usage**

L = MeraList()

L.append('Pranav')

L.append(3.4)

L.append(True)

L.append(100)

L.append('infinite')

L.append('infinity')

print("length of L : " , len(L)) # Shows the number of items

print(L) # Displays items in list format

L.insert(2, 'pooja') # Inserts 'pooja' at index 2

print(L)

print(L.\_\_delitem\_\_(500)) # Tries to delete an invalid index

print(L.remove('infinite')) # Removes 'infinite' from the list

Each of these operations tests different list features. This class implementation illustrates how Python’s list might work under the hood, including dynamic resizing and basic list operations, offering a more fundamental understanding of data structures.

class Node:

    def \_\_init\_\_(self,value):

        self.data = value

        self.next = None

"""

a = Node(1)

b = Node(2)

c = Node(3)

a.next = b

b.next = c

print('B node connected to the node  : ',(id(b.next)))

print(id(a))

print(id(b))

print(id(c))

"""

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None     # create empty linked list

        self.n = 0  # number of nodes In the LL

    def \_\_len\_\_(self):

        return self.n

    def insert\_head(self,value):

        # new node

        new\_node = Node(value)

        # create connection

        new\_node.next = self.head

        # reassign head

        self.head = new\_node

        # increment

        self.n += 1

    def \_\_str\_\_(self):

        curr = self.head

        result = ''

        while curr != None:

            result += str(curr.data) + ' -> '

            curr = curr.next

        return result[:-3]

    def append(self,value):

        new\_node = Node(value)

        if self.n == None:

            self.head = new\_node

            self.n += 1

            return

        curr = self.head

        while curr.next != None:

            curr = curr.next

        # you are at the last node

        curr.next = new\_node

        self.n += 1

    def insert\_after(self,after,value):

        new\_node = Node(value)

        curr = self.head

        while curr != None :

            if curr.data == after:

                break

            curr = curr.next

        # case 1 break -> item apko mil gaya -> curr -> not none

        if curr != None:

            # logic

            new\_node.next - curr.next

            curr.next = new\_node

        else :

            return 'Item not found'

    def clear(self):

        self.head = None

        self.n = 0

    def delete\_head(self):

        if self.head == None:

            #empty

            return 'Empty lInked list'

        self.head = self.head.next

        self.n = self.n -1

    def pop(self):

        if self.head == None:

            # empty

            return 'Empty LL'

        # kya linked list me 1 item hai

        if curr.next == None :

            # head hi hoga ( delete from head)

            return

        curr = self.head

        while curr.next.next != None:

            curr = curr.next

        # curr -> 2nd last node

        curr.next = None

        self.n = self.n -1

    def remove(self,value):

        if self.haed == None:

            return 'Empty linked list'

        if self.head.data == value:

            # you want to remove the head node

            return self.delete\_head()

        curr = self.head

        while curr.next != None:

            if curr.next.data == value:

                break

            curr = curr.next

        # 2 cases item mil gaya

        # item nai mila

        if curr.next == None:

            # item nahi mila

            return "Not found"

        else:

            curr.next = curr.next.next

    def search(self,item):

        curr = self.head

        pos = 0

        while curr != None:

            if curr.data == item:

                return pos

            curr = curr.next

            pos = pos + 1

        return 'Not Found'

    def \_\_getitem\_\_(self,index):

        curr = self.head

        pos = 0

        while curr != None:

            if pos == index:

                return curr.data

            curr = curr.data

            pos += 1

        return "indexError"

L = LinkedList()

L.insert\_head(4)

L.insert\_head(3)

L.insert\_head(2)

L.insert\_head(1)

L.append(5)

L.append("pranav ")

print(len(L))

print(L)

Let's break down this code thoroughly to understand each part and to highlight potential issues and challenging concepts for those who may be new to linked lists in Python. This code defines a simple singly linked list data structure, which contains nodes linked in a sequence, with several methods for manipulating this list.

**1. Node Class**

class Node:

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

self.data = value

self.next = None

* **Purpose**: The Node class represents a single element in a linked list.
* **Attributes**:
  + data: Holds the value of the node.
  + next: A pointer to the next node in the list (initialized as None, meaning it points to nothing initially).

The Node class is very simple, but it's foundational because each Node will be linked to others in the LinkedList.

**2. LinkedList Class**

The LinkedList class is where most of the linked list functionality is implemented. Each method in this class allows different interactions with the list.

**Initialization (\_\_init\_\_ Method)**

class LinkedList:

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

self.head = None # Initializes the list as empty

self.n = 0 # Keeps track of the number of nodes

* self.head points to the first node in the list. It starts as None, meaning the list is initially empty.
* self.n tracks the length of the list.

**\_\_len\_\_ Method**

def \_\_len\_\_(self):

return self.n

This returns the number of nodes in the list. Useful for easily checking the list's size.

**insert\_head Method**

def insert\_head(self, value):

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

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

self.head = new\_node # Update head to point to the new node

self.n += 1 # Increase the count of nodes

* This method inserts a new node at the beginning of the list.
* **Potential Issue**: Make sure that self.head is properly reassigned to new\_node. Beginners might accidentally set self.head to None after the assignment.

**\_\_str\_\_ Method**

def \_\_str\_\_(self):

curr = self.head

result = ''

while curr != None:

result += str(curr.data) + ' -> '

curr = curr.next

return result[:-3]

* This converts the list to a string for easy printing.
* **Key Point**: It iterates through the list and concatenates each node’s data. The final [:-3] trims the extra ' -> ' at the end.

**append Method**

python

Copy code

def append(self, value):

new\_node = Node(value)

if self.head is None: # If the list is empty, set head to new node

self.head = new\_node

self.n += 1

return

curr = self.head

while curr.next is not None:

curr = curr.next

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

self.n += 1

* Appends a new node at the end of the list.
* **Potential Issue**: The condition if self.n == None should be if self.head is None. This typo can lead to logic errors and incorrect list structure. The correct code checks whether self.head is None to see if the list is empty.

**insert\_after Method**

python

Copy code

def insert\_after(self, after, value):

new\_node = Node(value)

curr = self.head

while curr is not None:

if curr.data == after:

break

curr = curr.next

if curr is not None:

new\_node.next = curr.next # Insert new node after the found node

curr.next = new\_node

else:

return 'Item not found'

* **Purpose**: Inserts a node with value after the first node that contains after.
* **Potential Issue**: Typo in new\_node.next - curr.next should be new\_node.next = curr.next. The current line does nothing and may confuse new learners as they might think it creates a link, but it does not.

**clear Method**

python

Copy code

def clear(self):

self.head = None # Set head to None, effectively clearing the list

self.n = 0 # Reset node count to zero

* Clears the entire linked list.
* **Note**: This doesn't delete the nodes explicitly, but by removing references to them, they are garbage collected in Python.

**delete\_head Method**

python

Copy code

def delete\_head(self):

if self.head is None:

return 'Empty Linked list'

self.head = self.head.next

self.n -= 1

* Deletes the head node and reassigns self.head to the next node.
* **Potential Issue**: If self.head is None, it means the list is empty. This case is handled here by returning a message.

**pop Method**

python

Copy code

def pop(self):

if self.head is None:

return 'Empty LL'

if self.head.next is None:

self.head = None

self.n -= 1

return

curr = self.head

while curr.next.next is not None:

curr = curr.next

curr.next = None

self.n -= 1

* Removes the last node in the list.
* **Key Point**: curr.next.next ensures curr stops at the second-to-last node.

**remove Method**

python

Copy code

def remove(self, value):

if self.head is None:

return 'Empty linked list'

if self.head.data == value:

return self.delete\_head()

curr = self.head

while curr.next is not None:

if curr.next.data == value:

break

curr = curr.next

if curr.next is None:

return "Not found"

else:

curr.next = curr.next.next

* **Purpose**: Removes the node with the specified value.
* **Potential Issue**: self.haed should be self.head. This typo will cause an error since self.haed doesn’t exist.

**search Method**

python

Copy code

def search(self, item):

curr = self.head

pos = 0

while curr is not None:

if curr.data == item:

return pos

curr = curr.next

pos += 1

return 'Not Found'

* Searches for an item and returns its position in the list.
* **Potential Issue**: This method only returns the position of the first occurrence of item.

**\_\_getitem\_\_ Method**

python

Copy code

def \_\_getitem\_\_(self, index):

curr = self.head

pos = 0

while curr is not None:

if pos == index:

return curr.data

curr = curr.next

pos += 1

return "indexError"

* Allows you to access list items by index.
* **Potential Issue**: curr = curr.data should be curr = curr.next. curr = curr.data incorrectly assigns curr to a data value, which is an error since curr should hold nodes, not data.

**Example Usage**

python

Copy code

L = LinkedList()

L.insert\_head(4)

L.insert\_head(3)

L.insert\_head(2)

L.insert\_head(1)

L.append(5)

L.append("pranav")

print(len(L))

print(L)

In this example:

* **Inserting at Head**: L.insert\_head(4) inserts 4 at the head, and so on.
* **Appending**: L.append(5) adds 5 at the end.
* **Output**: The \_\_str\_\_ method formats the list when printing.

Overall, the code forms a basic linked list implementation in Python, with a few key areas to watch out for in terms of typos and logical errors. Make sure each line correctly references nodes and updates the next pointers accurately.