**Let’s create Python’s List with their 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.