Introduction

Types of Data Structures

- Arrays
- Linked Lists
- Queues
- Stacks

Prerequisities

- Linear Arrays (`List` in Python)
- Objects and Classes (Good grip on the OOP concepts)

Arrays

Linear Arrays

- What is Linear Arrays?

> An array is a collection of a fix number of items where all the items are of the same data type.

- Arrays consist of :
    - **Element** - Each item stored in an array is called an element.
    - **Index** - Each location of an element in an array has a numerical value, which is used to access its corresponding element.

Liner Array:

Elements	50	60	70	80	90	100
Index	0	1	2	3	4	5

We can declare a liner array (in python a `list`),

```
arr = [0, 1, 2, 3, 4, 5]
```

This way an adress is created in the memory where the array is stored. We can see the memory location in Python3 with the `id()` and `hex()` methods.

> `id()` -> prints the memroy adress of an object.
> `hex()` -> converts the memory adress to hexadecimal representation.

```
>> print(hex(id(arr)))
0x7f3bec064ec0
```

So, whenever we're typing `arr`, we're not actually referencing the array itself. We're referencing a variable that has the reference of the memory location where the array is stored.

Copying an array

There are 3 ways we can copy arrays in Python:

- Simply using the assignment operator
- Shallow copy
- Deep copy

Using Assignment Operator

We can use `=` to copy arrays.

For example

```
new_arr = old_arr
```

Something to remember :

> In python, Assignment statements don't copy objects, they create bindings between a target and an object. When we use `=` operator user may think that this creates a new object which it doesn't. It only creates a new variable that shares the reference of the origianl object.

Simply put, `new_arr = old_arr` just copy the adress of the `old_arr` to `new_arr`. So, no new array has been created in the memory. These `old_arr` and `new_arr` now points to the same adress in the memory.

For example,

```
arr1 = [2, 6, 9, 4]

# Prints the address of the arr1 in the memory
print(hex(id(arr1)))

# assigning arr1 to arr2
arr2 = arr1

# Prints the address of the arr1 in the memory
print(hex(id(arr2)))

# making a change in arr1
arr1[1] = 7

# displaying the arrays
>> print(arr1)
>> print(arr2)
0x7f5dcf5d0e80
0x7f5dcf5d0e80
[2, 7, 9, 4]
[2, 7, 9, 4]
```

So you see, because the two array shares the same memory address, chaning the value in one array changes the other too.

> Sometimes we want to have the original values unchanged and only modify the new values or vice varsa. In Python we can use the other two copy methods which are,

- Shallow Copy

- Deep Copy

To make these copy work, we use the `copy` module.

Shallow Copy

> A shallow copy creates a new object which stores the refernce of the original elements. So, a shallow copy doesn't just copies the reference of nested objects. This means a copy process doesn't recurse or create copies of nested object itself.

In other words, a shallow copy can copy a normal list / array and store that in a different memory space. However, if the list is a nested list, the shallow copy method will only copy the whole list in a another memory location, however, it will not copy nested list objects. Instead it'll create the reference of origianl memory for the nested items.

For better understanding this, let's see an example,

```
import copy

old_list = [1,2,3,4]
new_list = copy.copy(old_list)

new_list.append(5)

print(f"Old list: {old_list}")
print(f"Old list memory location: {hex(id(old_list))}")
print(f"New list: {new_list}")
print(f"New list memory location: {hex(id(new_list))}")
```

After running the program, we'll get,

```
Old list: [1, 2, 3, 4]
Old list memory location: 0x7f42452b8e00
New list: [1, 2, 3, 4, 5]
New list memory location: 0x7f42452b8480
```

So, with this method we can `copy` a list in python. However, this only **really** works with normal lists. We'll get into problems if we use this method to copy nested lists.

For example,

```
import copy

old_list = [[1,2,3],[4,5,6]]
```

```
new_list = copy.copy(old_list)

new_list[0] = ['a','b','c']

print(f"Old list: {old_list}")
print(f"Old list memory location: {hex(id(old_list))}")
print(f"New list: {new_list}")
print(f"New list memory location: {hex(id(new_list))}")
```

Output:

```
Old list: [[1, 2, 3], [4, 5, 6]]
Old list memory location: 0x7f7df913c800
New list: [['a', 'b', 'c'], [4, 5, 6]]
New list memory location: 0x7f7df913c580
```

This works fine. Now, let's see what happens if we change the nested list objects inside the original list.

```
import copy

old_list = [[1, 2, 3], [4, 5, 6]]
new_list = copy.copy(old_list)

new_list[0][0] = "a"

print(f"Old list: {old_list}")
print(f"Old list memory location: {hex(id(old_list))}")
print(f"New list: {new_list}")
print(f"New list memory location: {hex(id(new_list))}")
```

Output:

```
Old list: [['a', 2, 3], [4, 5, 6]]
Old list memory location: 0x7ff49810f900
New list: [['a', 2, 3], [4, 5, 6]]
New list memory location: 0x7ff49810f680
```

`old_list` and `new_list` both changes. This happen because, the `shallow copy` method only copy the whole list in a new memory but it doesn't touch the nested list objects. Instead it creates a reference of the original memory location for them. That is why when we change only the whole list the changes happen in only the new list not both. However, when we touch the nested list objects, changes happen in both the new and the old lists. So, this is `Shallow Copy` .

> In case anyone wandering about `list.copy()` method, Yes, that is a shallow copy method too.

If you're still confused read this answer from stackoverflow.

Deep Copy

> A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

A deepcopy solves the problem we had with the nested list with the shallow copy. Let's now apply deep copy method with the nested list example we just used earlier.

```
import copy

old_list = [[1, 2, 3], [4, 5, 6]]
new_list = copy.deepcopy(old_list)

new_list[0][0] = "a"

print(f"Old list: {old_list}")
print(f"Old list memory location: {hex(id(old_list))}")
print(f"New list: {new_list}")
print(f"New list memory location: {hex(id(new_list))}")
```

Output:

```
Old list: [[1, 2, 3], [4, 5, 6]]
Old list memory location: 0x7fea699b4880
New list: [['a', 2, 3], [4, 5, 6]]
New list memory location: 0x7fea699b4600
```

So, using the `deepcopy()` we've solved the problem here.

Something to remember :

> Although in python we can change the array / list size anytime, however, in other programming languages such as C++ or Java, the size is fixed from the get go. So, in this course for learning purposes we'll always fix the size of the arrays / list from the get go too. It'll help us understand thigs more easily. For example,

```
>> arr = [0] * 10
>> print(arr)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

This will create an array of length 10. The value `0` indicate that the value of that index is empty.

Array Traversal

Array traversal is accessing the elements of an array one by one. (Basicly, `for loop`)

In python3,

```
>> arr = [0,1,2,3,4,5]
>> for i in arr:
>>     print(i, end=" ")
0 1 2 3 4 5
```

Reverse printing an Array

We have an array and we need to print it's values in reverse order.

```
# with for loop
def print_array_reverse(arr):
   for i in range(len(arr) - 1, -1, -1):
       print(arr[i], end=" ")

# with while loop
def print_array_reverse(arr):
   i = len(arr) - 1
   while i >= 0:
       print(arr[i], end=" ")
       i -= 1

>> print_array_reverse([0,1,2,3,4,5])
5 4 3 2 1 0
```

Resizing an Array

Shifting an array Left

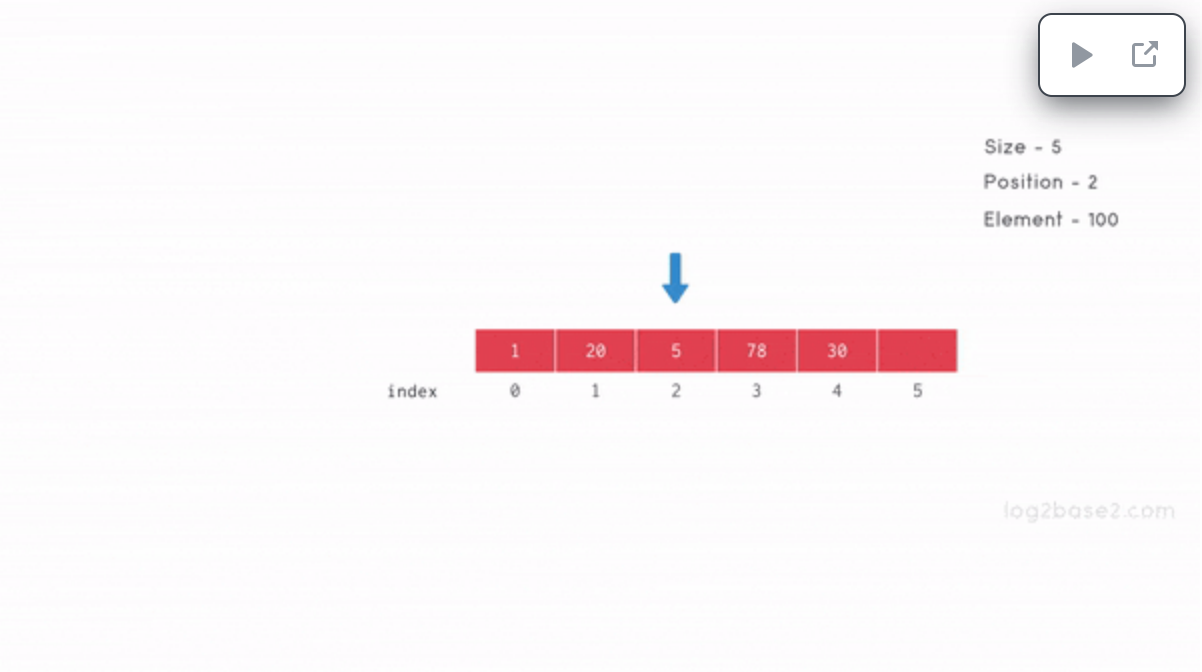
Shifting an array Right

Inserting an element into an Array

We have an array for example, `arr = [1 ,20 ,5 ,78 ,30 ,0]` (Here, `0` means empty value). We need to insert a value for example, `100` in the `2` nd index. So that the resulting array will looks like, `arr = [1 ,20 ,200, 5, 78, 30]`

So, we can achieve this by,

- Shift all the values 1 index to right (we have some empty space to the right)
- After shifting all the values to the right, index `2` will be empty. Simply, insert the value into index `2`



In python3,

```
def insert(arr, index, value, size):
    if size == len(arr):
        print("No space in array!")
        return

    if index < 0 or index > size:
        print("Wrong Index!")
```

```
            return

        else:
            for i in range(size, index - 1, -1):
                arr[i] = arr[i - 1]

            arr[index] = value
            print(arr)

>> arr = [1, 20, 5, 78, 30, 0]
>> insert(arr, 2, 100, 5))
[1, 20, 100, 5, 78, 30]
```

Removing an element from an array

Rotating an array Left

Rotating an array Right

Rotate By K cell to right

```
def rightShift(source,k):

    i=len(source)-1 #pointer at the last index of source

    while(i>=k):

        source[i]=source[i-k]  #shifting elements k places to the right

        i=i-1

    i=0

    while(i<k):

        source[i]=0 #setting first k values to 0

        i=i+1



a=[10,20,30,40,50]

rightShift(a,3)
```

```
    print(a)
```