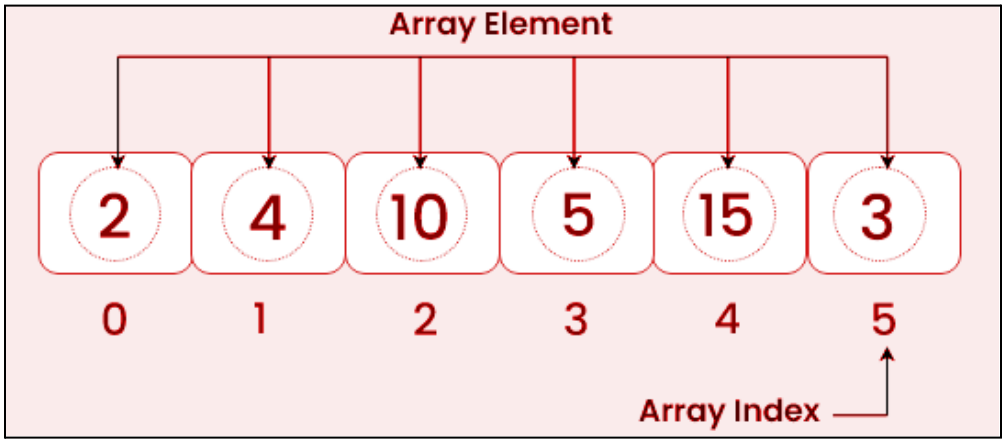
Linear Array

In applications where we have a small number of items to handle, we tend to specify separate variable names for each item. When we have to keep track of more related data, we need to organize data so that we can use one name to refer to several items. That's when we use Arrays.

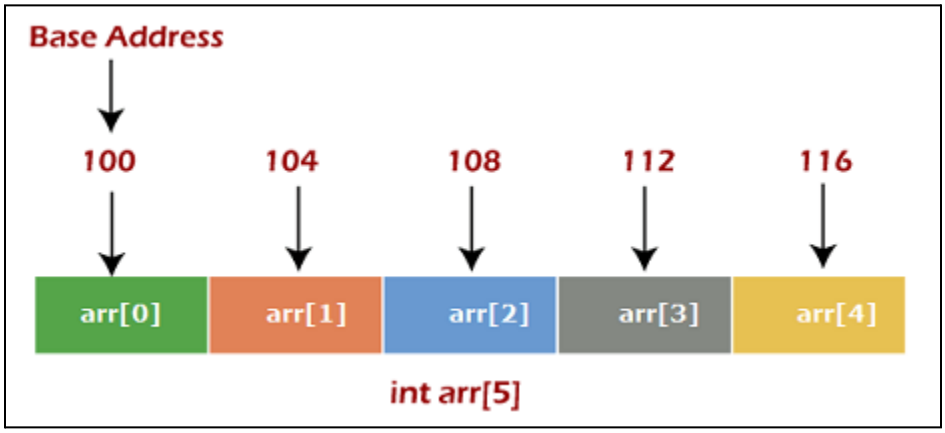


Definition: An array is a collection of items of the same variable type that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0. Each element in an array is accessed through its index.

Array Properties:

- **Homogeneous**: the elements of an array are of the same type, cannot mix int and float
- **Fixed Size**: it has a specified number of elements and cannot be changed
- **Index**: it starts with an index of 0 and ends with index = the size of the array minus 1.
- **Ordered**: elements are ordered starting from the 0^{th} index and all the non-dummy values will be at the left followed by the dummy values if any (zero/None/null)
- **Random Access**: elements can be accessed directly using the index
- **Contiguous Memory**: elements are simply placed one after the other in memory
- **Mutable**: elements of an array can be changed after initialization

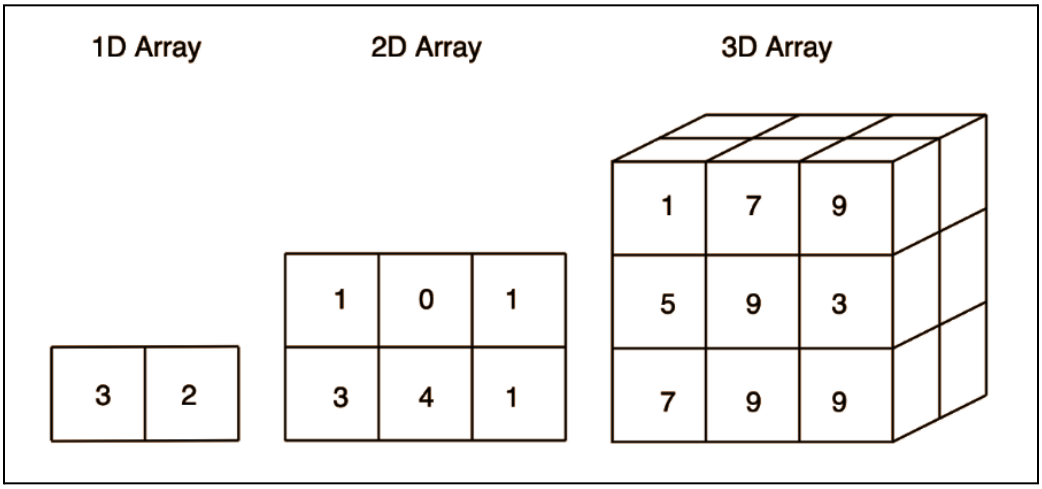
Memory allocation of an Array:



All the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory.

Types of Array:

- One-dimensional arrays: These arrays store a single row of elements.
- Multidimensional arrays: These arrays store multiple rows of elements.



Two-dimensional arrays, sometimes called matrices, are quite common. The best way to think about a two-dimensional array is to visualize a table of columns and rows: the first dimension in the array refers to the rows, and the second dimension refers to the columns

Linear Array Declaration/Initialization:

- Array name
- Type of data to be stored in Array
- Size of Array

An array declaration tells the computer two major pieces of information about an array. First, the size allows the computer to determine how many memory locations must be allocated. Second, the array type tells the computer how much space is required to hold each value.

```
import numpy as np

# Creating a 1D array with zeros of length 5
arr = np.zeros(5, dtype=int)

# Creating a 1D array with integers
arr_int = np.array([1, 2, 3, 4, 5], dtype=int)

# Creating an array with floats
arr_float = np.array([1.0, 2.0, 3.0, 4.0, 5.0], dtype=float)

# Creating an array with strings
arr_str = np.array(['a', 'b', 'c', 'd'], dtype=str)
```

Linear Array Operations:

- Read/Write:

```
# Reading the second element of a 1D array
arr_1D[1]

# Writing the second element of a 1D array
arr_1D[1] = 5
```

- Iteration:

```
arr = np.array([1, 2, 3, 4], dtype=int)

for i in arr:
        print(i)

for i in range(0, len(arr),1):
        print(arr[i])
```

- Resize:

We can not resize an array because it has a fixed memory location. However, if we ever need to resize an array, we need to create a new array with a new length and then copy the values from the original array. For example, if we have an array [10, 20, 30, 40, 50] whose length is 5 and want to resize the array with length 8. The new array will be [10, 20, 30, 40, 50, None, None, None].

```
def resize_array(arr, new_size):
        arr2= np.zeros(new_size, dtype=int)
        for i in range(len(arr)):
                arr2[i] = arr[i]
        return arr2
```

- Copy:

```
def copy_array(arr):
        arr2 = np.zeros(len(arr), dtype=int)
        for i in range(len(new_arr)):
                arr2[i] = arr[i]
        return arr2
```

- Shift:

Shift Left: Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. The first element in the array will fall off at the beginning and be lost forever. The last slot of the array before the shift (ie. the slot where the last element was until the shift) is now unused (we can put a 0 there to signify that). For example, shifting the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, 0].

```
def shift_left(arr):
        for i in range(1, len(arr), 1):
                arr[i-1] = arr[i]
        arr[len(arr) -1] = 0
        return arr
```

Shift Right: Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. The last element in the array will fall off at the end and be lost forever. The first slot of the array before the shift (ie., the slot where the first element was until the shift) is now unused (we can put a 0 there to signify that). For example, shifting the array [5, 3, 9, 13, 2] right by one position will result in the array [0, 5, 3, 9, 13].

```
def shift_right(arr):
        for i in range(len(arr) -1, 0, -1):
                arr[i] = arr[i-1]
        arr[0] = 0
        return arr
```

- Rotate:

Rotate Left: Rotating an array left is equivalent to shifting a circular array left, where the first element is not lost but moved to the last slot. For example, rotating the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, 5].

```
def rotate_left(arr):
        temp = arr[0]
        for i in range(1, len(arr), 1):
                arr[i-1] = arr[i]
        arr[len(arr)-1] = temp
        return arr
```

Rotate Right: Rotating an array right is equivalent to shifting a circular array right where the last element will not be lost, but rather move to the 1st slot. For example, rotating the array [5, 3, 9, 13, 2] right by one position will result in the array [2, 5, 3, 9, 13].

```
def rotate_right(arr):
        temp = arr[len(arr)-1]
        for i in range(len(arr) -1, 0, -1):
                arr[i] = arr[i-1]
        arr[0] = temp
        return arr
```

- Reverse:

Out-of-place Reverse: Creating a new array with the same size as the original array and then copying the values in reverse order.

```
def reverse_out_of_place(arr):
        arr2 = np.zeros(len(arr), dtype=int)
        i = 0
        while(i <= len(arr) -1):
                arr2[i] = arr[len(arr) -1- i]
                i += 1
        return arr2
```

In-place Reverse: Swapping values from the beginning position to the end position without creating any extra array

```
def reverse_in_place(arr):
        j = len(arr) - 1
        for i in range(len(arr) // 2):
                arr[i], arr[j] = arr[j], arr[i]
                j -= 1
        return arr
```

New Linear Array Property: Size:

- ❖ Size indicates the number of non-dummy values in an array.
- ❖ Size of an array can never exceed the length of that array. Which means size > length is not possible.
- ❖ Size cannot be negative.
- ❖ Size is used in array insertion or deletion, or any task that involves working with the non-dummy values only.
- ❖ If size < length, insertion in an array is possible, otherwise must resize the array.
- ❖ If size > 0, at least one element from the array can be deleted, otherwise, there is nothing to delete.

- Insert:

First, check whether insertion is possible. If not, resize the array. Valid places for inserting are index 0 to size.

Insert End:

```
def insert_end(arr, size, elem):
        if size == len(arr):
                arr = resize_array(arr, size+1)
        arr[size] = elem
        return arr
```

Insert Anywhere Else: Inserting anywhere except the end means creating an empty space for the element by right-shifting all the elements that are right of the desired index for insertion.

```
def insert_anywhere(arr, size, index, elem):
        if index < 0 or index >= size:
                return "Insertion Not Possible"
        if size == len(arr):
                arr = resize_array(arr, size+1)
        for i in range(size, index, -1): #Right Shift
                arr[i] = arr[i-1]
        arr[index] = elem
        return arr
```

- Delete:

At first, check whether deletion is possible or not. Valid places of deletion: index 0 to size-1.

Delete End:

```
def delete_end(arr, size):
        if size == 0:
                return "Deletion Not Possible"
        arr[size-1] = 0
        return arr
```

Delete Anywhere Else: Deleting anywhere except the end means removing the element which will create an empty space. This space must be filled by left-shifting all the elements that are right of the index where the deletion occurred.

```
def delete_anywhere(arr, size, index):
        if size == 0 or (index < 0 and index >= size):
                return "Deletion Not Possible"
        for i in range(index, size-1, 1): #Left Shift
                arr[i] = arr[i+1]
        arr[size-1] = 0
        return arr
```