Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array are as follows –

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Properties of Arrays:

1.Fixed Type: Like arrays in many other programming languages, Python's arrays have a fixed data type. All elements in the array must be of the same data type.

2.Efficiency: Arrays are more memory efficient than lists, especially when dealing with large datasets of a single data type. This efficiency comes from the fact that arrays store elements in a contiguous block of memory.

3.Indexing and Slicing: Elements in an array can be accessed using index values, and slices of arrays can be obtained using the colon (`:`) notation, similar to lists.

4.Mutability: Python's arrays are mutable, meaning you can change, add, or remove elements after the array is created.

5.Numeric Data Types: Python's `array` module supports a range of numeric data types, such as integers, floats, and doubles.

6.Basic Operations: Arrays support basic operations such as concatenation, repetition, and membership testing.

7.Performance: Due to their fixed type and contiguous memory allocation, arrays can offer better performance.

Advantages:

1. **Efficiency**: Arrays can be more memory-efficient than lists, especially when dealing with large datasets of a single data type. This efficiency comes from the fact that arrays store elements in a contiguous block of memory.

2. **Performance**: Due to their fixed type and contiguous memory allocation, arrays can offer better performance for certain operations compared to lists, especially for numerical computations. Operations such as element access and iteration can be faster with arrays.

3. **Typed Elements**: Arrays enforce a fixed data type for all elements, ensuring type safety and potentially preventing unintended type errors.

4. **Slicing**: Arrays support slicing operations, allowing you to extract subarrays efficiently.

5. **Interoperability**: Arrays can be easily converted to and from other data structures such as lists, making them suitable for interoperability with other Python code.

Disadvantages:

1.Fixed Type: The fixed type of arrays can be limiting in some cases, especially when dealing with heterogeneous data. Unlike lists, which can hold elements of different types, arrays require all elements to be of the same type.

2.Limited Functionality: Arrays offer fewer built-in methods and functionalities compared to lists. For example, arrays lack many of the convenient methods provided by lists, such as `append()`, `extend()`, and `remove()`.

3.Memory Management: Since arrays require contiguous memory allocation, resizing an array (e.g., to add more elements) can be inefficient and may involve copying elements to a new memory location.

4.Complexity: Working with arrays requires a good understanding of memory management and low-level data manipulation, which can be more complex compared to using higher-level data structures like lists.

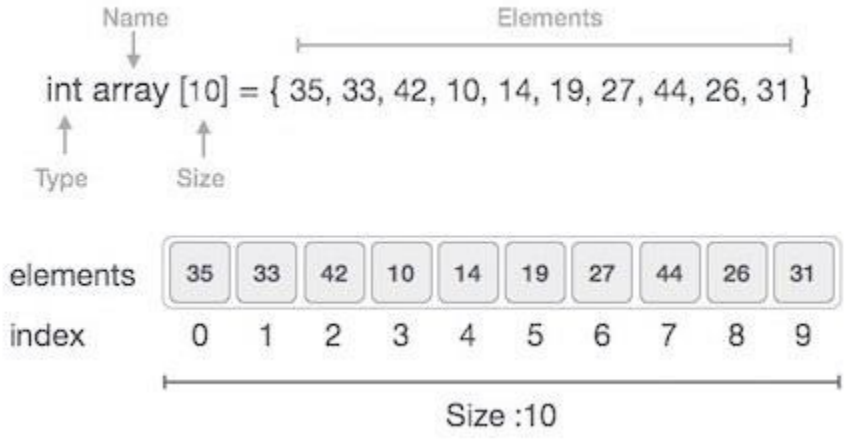
5.Type Constraints: While arrays enforce a fixed data type for elements, this constraint can also be a disadvantage if you need to work with heterogeneous data or if your data types may vary over time.

Array vs List vs Tuples vs Set vs Dictionary

Property	Arrays	Lists	Tuples	Sets	Dictionaries
Type	Fixed type	Dynamic type	Dynamic type	Dynamic type	Dynamic keys, dynamic values
Mutability	Mutable	Mutable	Immutable	Mutable	Mutable
Ordered	Ordered	Ordered	Ordered	Unordered	Unordered keys, unordered values
Duplicate Elements	Allowed	Allowed	Allowed	Not allowed	Allowed (Keys must be unique)
Indexing	Yes	Yes	Yes	No	No
Slicing	Yes	Yes	Yes	No	No
Insertion Order	Preserved	Preserved	Preserved	Not applicable	Not applicable
Performance	Efficient	Less efficient	Less efficient	Efficient for membership test	Efficient for key-based operations
Use Case	Numerical data, performance-critical operations	General-purpose lists	Immutable collections, returning multiple values from functions	Unique elements, membership test	Key-value pairs, fast lookup by key
Memory Efficiency	More efficient	Less efficient	Less efficient	More efficient	Less efficient

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered –

- Index starts with 0.
- Array length is 10, which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

The basic operations supported by an array are as stated below −

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then, the array is declared as shown below −

```
from array import *

arrayName = array(typecode, [Initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common type codes used are as follows –

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Example

The below code creates an array named **array1**.

```
from array import *

array1 = array('i', [10,20,30,40,50])

for x in array1:
   print(x)
```

Output

When we compile and execute the above program, it produces the following result –

```
10
20
30
40
50
```

Basic Operations on Arrays:

Accessing Array Element

We can access each element of an array using the index of the element. The below code shows how to access an array element.

Example

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1[0])

print (array1[2])
```

Output

When we compile and execute the above program, it produces the following result, which shows the element is inserted at index position 1.

```
10
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Example

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1.insert(1,60)

for x in array1:
   print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position 1.

Output

```
10
60
20
30
40
50
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Example

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1.remove(40)

for x in array1:
   print(x)
```

Output

When we compile and execute the above program, it produces the following result which shows the element is removed form the array.

```
10
20
30
50
```

Search Operation

You can perform a search for an array element based on its value or its index.

Example

Here, we search a data element using the python in-built index() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

Output

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

```
3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Example

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80

for x in array1:
   print(x)
```

Output

When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

```
10
20
80
40
50
```

Time Complexity:

1. **Accessing Element by Index**: O(1)
    - Accessing an element in an array by its index is a constant-time operation because arrays provide direct access to memory locations based on the index.

2. **Searching for an Element (Linear Search)**: O(n)
    - In the worst-case scenario, if you need to search for an element in an unsorted array, you may need to traverse through all the elements, resulting in a time complexity of O(n).

3. **Insertion or Deletion at the End**: O(1)
    - Inserting or deleting an element at the end of an array (assuming there's available space) is a constant-time operation because it only involves updating the length of the array.

4. **Insertion or Deletion at Arbitrary Index**: O(n)
    - Inserting or deleting an element at an arbitrary index in an array (not at the end) requires shifting elements to accommodate the change, resulting in a linear time complexity O(n) because, in the worst case, you may need to move all elements after the insertion/deletion point.

Space Complexity:

- The space complexity of an array is O(n), where n is the number of elements in the array. This is because arrays require contiguous memory allocation, and the amount of memory required is directly proportional to the number of elements stored in the array.