CHAPTER

7

ARRAYS IN PYTHON

Suppose there is a group of students whose marks are to be listed. The first student's marks are stored in a variable m1, the second student's marks are stored in m2, the third student's marks in m3, and so on and the 100th student's marks are stored in m100. It means, we are supposed to create 100 variables to store 100 students' marks. Thus, we are supposed to write 100 statements like this:

```
1=65
m2=61
m3=70
:
m100=67
```

Now, if we want to display these marks, we need to write another 100 statements. It means, a simple program will contain hundreds of statements and this becomes difficult for the programmer. On the other hand, just imagine there is only one variable that stores 100 students' marks. That variable will be very much useful to us since we have to write only 1 statement instead of writing 100 statements. For example, we can declare 'm' variable as an array and store all the marks there. This reduces the program size considerably and the programmer's task will become easy. Let's now learn more about array.

Array

So, what is an array? An array is an object that stores a group of elements (or values) of same datatype. The main advantage of any array is to store and process a group of elements easily. There are two points we should remember in case of arrays in Python.

- Arrays can store only one type of data. It means, we can store only integer type elements or only float type elements into an array. But we cannot store one integer, one float and one character type element into the same array.

- Arrays can increase or decrease their size dynamically. It means, we need not declare the size of the array. When the elements are added, it will increase its size and when the elements are removed, it will automatically decrease its size in memory.

In Python, there is a standard module by the name 'array' that helps us to create and use arrays. To get help on 'array' module, we can type help() at Python prompt to get the help prompt as:

```
>>>help()
help>
```

Then we should type simply 'array' at the help prompt to see all information about the array module as shown in Figure 7.1:

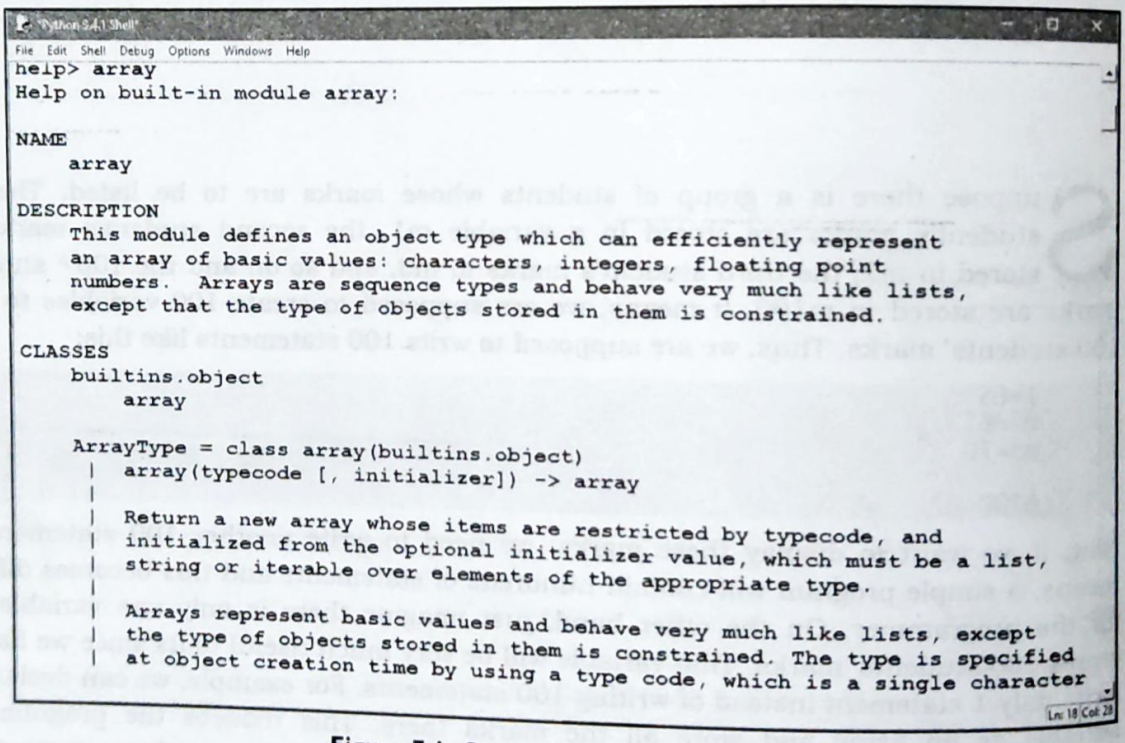

Figure 7.1: Getting Help on Array Module

Advantages of Arrays

The following are some advantages of arrays:

- Arrays are similar to lists. The main difference is that arrays can store only one type of elements; whereas, lists can store different types of elements. When dealing with a huge number of elements, arrays use less memory than lists and they offer faster execution than lists.
- The size of the array is not fixed in Python. Hence, we need not specify how many elements we are going to store into an array in the beginning.

- Arrays can grow or shrink in memory dynamically (during runtime).
- Arrays are useful to handle a collection of elements like a group of numbers or characters.
- Methods that are useful to process the elements of any array are available in 'array' module.

Creating an Array

We have already discussed that arrays can hold data of same type. The type should be specified by using a type code at the time of creating the array object as:

```
arrayname = array(type code, [elements])
```

The type code 'i', represents integer type array where we can store integer numbers. If the type code is 'f' then it represents float type array where we can store numbers with decimal point. The important type codes are given in Table 7.1:

Table 7.1: The Type Codes to Create Arrays

Typecode	C Type	Minimum size in bytes
'b'	signed integer	1
'B'	unsigned integer	1
'h'	signed integer	2
'H'	unsigned integer	2
'i'	signed integer	4
'I'	unsigned integer	4
'f'	floating point	4
'd'	double precision floating point	8
'u'	unicode character	2

We will take an example to understand how to create an integer type array. We should first write the module name 'array' and then the type code we can use is 'i' for integer type array. After that the elements should be written inside the square braces [ ] as,

```
a = array('i', [4, 6, 2, 9])
```

This is creating an array whose name is 'a' with integer type elements 4, 6, 2 and 9.

Similarly, to create a float type array, we can write:

```
arr = array('d', [1.5, -2.2, 3, 5.75])
```

This will create an array by the name 'arr' with float type elements 1.5, -2.2, 3.0 and 5.75. The type code is 'd' which represents double type elements each taking 8 bytes memory.

Of course, in the previous statements we are using 'array' module to create arrays. To use this module, we have to first import 'array' module into our program.

Importing the Array Module

There are three ways to import the array module into our program. The first way is to import the entire array module using import statement as,

```
import array
```

When we import the array module, we are able to get the 'array' class of that module that helps us to create an array. See the following example:

```
a=array.array('i',[4,6,2,9])
```

Here, the first 'array' represents the module name and the next 'array' represents the class name for which the object is created. We should understand that we are creating our array as an object of array class.

The second way of importing the array module is to give it an alias name, as:

```
import array as ar
```

Here, the array is imported with an alternate name 'ar'. Hence we can refer to the array class of 'ar' module as:

```
a=ar.array('i', [4,6,2,9])
```

The third way of importing the array module is to write:

```
from array import *
```

Observe the '*' symbol that represents 'all'. The meaning of this statement is this: import all (classes, objects, variables etc) from the array module into our program. That means we are specifically importing the 'array' class (because of * symbol) of 'array' module. So, there is no need to mention the module name before our array name while creating it. We can create the array as:

```
a=array('i',[4,6,2,9])
```

Here, the name 'array' represents the class name that is available from the 'array' module. Let's write a Python program to create an integer type array with some integer elements. Program 1 shows how to create an integer type array.

Program

Program 1: A Python program to create an integer type array.

```
# creating an array
import array
a = array.array('i', [5, 6, -7, 8])

print('The array elements are: ')
for element in a:
    print(element)
```

Output:

```
C:\>python arr.py
5
6
-7
8
```

Observe the for loop in the previous program. The variable 'element' assumes each element value from the array 'a' and hence print(element) will display all the elements of the array. The same program can be rewritten using a different type of import statement as shown in Program 2.

Program

Program 2: A Python program to create an integer type array.

```
# creating an array - v 2.0
from array import *
a = array('i', [5, 6, -7, 8])

print('The array elements are: ')
for element in a:
    print(element)
```

Output:

```
C:\>python arr.py
5
6
-7
8
```

Let's write another program to see how to create an array with a group of characters and display the elements of the array. This is shown in Program 3.

Program

Program 3: A Python program to create an array with a group of characters.

```
# creating an array with characters
from array import *
arr = array('u', ['a','b','c','d','e'])

print('The array elements are: ')
for ch in arr:
    print(ch)
```

Output:

```
C:\>python arr.py
a
b
c
d
e
```

It is possible to create an array from another array. For example, there is an array 'arr1' as:

```
arr1 = array('d', [1.5, 2.5, -3.5, 4])
```

This is a float type array with 4 elements. We want to create another array with the name 'arr2' from arr1. It means, 'arr2' should have same type code and same elements as that of 'arr1'. For this purpose, we can write:

```
arr2 = array(arr1.typecode, (a for a in arr1))
```

Here, 'arr1.typecode' gives the type code character of the array 'arr1'. This type code is used for 'arr2' array also. Observe the expression:

```
a for a in arr1
```

The first 'a' represents that its value is stored as elements in the array 'arr2'. This 'a' value is got from 'a' (each element) in arr1. So, all the elements in arr1 will be stored into arr2. Suppose, we write the expression,

```
arr2 = array(arr1.typecode, (a*3 for a in arr1))
```

In this case, the value of 'a' obtained from arr1, is multiplied by 3 and then stored as element into arr2. So, arr2 will get the elements: 4.5, 7.5, -10.5, and 12.0. This can be verified from Program 4.

Program

Program 4: A Python program to create one array from another array.

```
# creating one array from another array
from array import *
arr1 = array('d', [1.5, 2.5, -3.5, 4])

# use same type code and multiply each element of arr1 with 3
arr2 = array(arr1.typecode, (a*3 for a in arr1))
print('The arr2 elements are: ')
for i in arr2:
    print(i)
```

Output:

```
C:\>python arr.py
The arr2 elements are:
4.5
7.5
-10.5
12.0
```

Indexing and Slicing on Arrays

An *index* represents the position number of an element in an array. For example, when we create the following integer type array:

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

Python interpreter allocates 5 blocks of memory, each of 2 bytes size and stores the elements 10, 20, 30, 40 and 50 in these blocks. Figure 7.2 shows the arrangement of elements the array 'x':

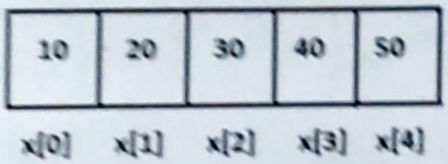

Figure 7.2: Arrangement of Elements in an Array

When we observe Figure 7.2, we can understand that the 0th element of the array is represented by x[0], the 1st element is represented by x[1] and so on. Here, 0, 1, 2, etc, are representing the position numbers of the elements. So, in general we can use i to represent the position of any element. This 'i' is called 'index' of the array. Using 'index, we can refer to any element of the array as x[i] where 'i' values will change from 0 to n-1. Here n represents the total number of elements in the array.

To find out the number of elements in an array we can use the len() function as:

```
n = len(x)
```

The len(x) function returns the number of elements in the array 'x' into 'n'. In Program 5, we are showing how to access the elements of an array using index.

Program

Program 5: A Python program to retrieve the elements of an array using array index.

```
# accessing elements of an array using index
from array import *
x = array('i', [10, 20, 30, 40, 50])

# find number of elements in the array
n = len(x)

# display array elements using indexing
for i in range(n):  # repeat from 0 to n-1
    print(x[i], end=' ')
```

Output:

```
C:\>python arr.py
10 20 30 40 50
```

It is also possible to access the elements of an array using while loop. The same program can be rewritten using while loop as Program 6.

Program

Program 6: A Python program to retrieve elements of an array using while loop.

```
# accessing elements of an array using index - v 2.0
from array import *
x = array('i', [10, 20, 30, 40, 50])
```

```
# find number of elements in the array
n = len(x)

# display array elements using indexing
i=0
while i<n:
    print(x[i], end=' ')
    i+=1
```

Output:

```
C:\>python arr.py
10 20 30 40 50
```

A *slice* represents a piece of the array. When we perform 'slicing' operations on any array, we can retrieve a piece of the array that contains a group of elements. Whereas indexing is useful to retrieve element by element from the array, slicing is useful to retrieve a range of elements. The general format of a slice is:

```
arrayname[start:stop:stride]
```

We can eliminate any one or any two in the items: 'start', 'stop' or 'stride' from the above syntax. For example,

```
arr[1:4]
```

The above slice gives elements starting from 1st to 3rd from the array 'arr'. Counting of the elements starts from 0. All the items 'start', 'stop' and 'stride' represent integer numbers either positive or negative. The item 'stride' represents step size excluding the starting element.

To understand how one can perform slicing operations on an array, we will write a Python program where we take an array 'x' and do slicing on the elements of the array 'x'. The result of slicing operations will be a new array 'y'. Please go through Program 7.

Program

Program7: A Python program that helps to know the effects of slicing operations on an array.

```
# create array y with elements from 1st to 3rd from x
y = x[1:4]
print(y)

# create array y with elements from 0th till the last element in x
y = x[0:]
print(y)

# create array y with elements from 0th to 3rd from x
y = x[:4]
print(y)

# create array y with last 4 elements from x
y = x[-4:]
print(y)

# create y with last 4th element and with 3 [-4-(-1)= -3]elements
# towards right.
```

```
y = x[-4: -1]
print(y)

# create y with 0th to 7th elements from x.
# stride 2 means, after 0th element, retrieve every 2nd element from x
y = x[0:7:2]
print(y)
```

Output:

```
C:\>python arr.py
array('i', [20, 30, 40])
array('i', [10, 20, 30, 40, 50, 60, 70])
array('i', [10, 20, 30, 40])
array('i', [40, 50, 60, 70])
array('i', [40, 50, 60])
array('i', [10, 30, 50, 70])
```

From Program 7, we can understand that slicing can be used to create new arrays from existing arrays. For example, when we write:

```
y = x[-4:]
```

we are creating a new array 'y' with the last 4 elements of 'x' that we obtained through slicing. Slicing can also be used to update an array. For example, to replace the 1st and 2nd elements of 'x' array with the elements [5, 7] of another array, we can write:

```
x[1:3] = array('i',[5, 7])
```

The elements 5, 7 are stored in the positions 1 and 2 in the array 'x'. The remaining element of the array will not be changed.

Just like indexing, slicing can also be used to retrieve elements from an array. In Program 8, we are displaying only a range of elements from the array 'x'.

Program

Program 8: A Python program to retrieve and display only a range of elements from an array using slicing.

```
# using slicing to display elements of an array.
from array import *
x = array('i', [10, 20, 30, 40, 50, 60, 70])

# display elements from 2nd to 4th only
for i in x[2:5]:
    print(i)
```

Output:

```
C:\>python arr.py
30
40
50
```

Processing the Arrays

The arrays class of arrays module in Python offers methods to process the arrays easily. The programmers can easily perform certain operations by using these methods. We should understand that a method is similar to a function that performs a specific task. But methods are written inside a class whereas a function can be written inside or outside the class. Methods are generally called as: objectname.method(). Please see Table 7.2 for methods that can be used on arrays:

Table 7.2: Array Methods

Method	Description
a.append(x)	Adds an element x at the end of the existing array a.
a.count(x)	Returns the numbers of occurrences of x in the array a.
a.extend(x)	Appends x at the end of the array a. 'x' can be another array or an iterable object.
a.fromfile(f, n)	Reads n items (in binary format) from the file object f and appends to the end of the array a. 'f' must be a file object. Raises 'EOFError' if fewer than n items are read.
a.fromlist(lst)	Appends items from the lst to the end of the array. 'lst' can be any list or iterable object.
a.fromstring(s)	Appends items from string s to the end of the array a.
a.index(x)	Returns the position number of the first occurrence of x in the array. Raises 'ValueError' if not found.
a.insert(i, x)	Inserts x in the position i in the array.
a.pop(x)	Removes the item x from the array a and returns it.
a.pop()	Remove last item from the array a.
a.remove(x)	Removes the first occurrence of x in the array a. Raises 'ValueError' if not found.
a.reverse()	Reverses the order of elements in the array a.
a.tofile(f)	Writes all elements to the file f.
a.tolist()	Converts the array 'a' into a list.
a.tostring()	Converts the array into a string.

Apart from the methods shown in Table 7.2, we can also use the following variables of Array class as shown in Table 7.3:

Table 7.3: Variables of Array Class

Variable	Description
a.typecode	Represents the type code character used to create the array a
a.itemsize	Represents the size of items stored in the array (in bytes)

Most of these methods are used in Program 9 so that there will be better clarity regarding their results.

Program

Program 9: A Python program to understand various methods of arrays class.

```
# operations on arrays
from array import*

# create an array with int values
arr = array('i', [10,20,30,40,50])
print('Original array: ', arr)

# append 30 to the array arr
arr.append(30)
arr.append(60)
print('After appending 30 and 60: ', arr)

# insert 99 at position number 1 in arr
arr.insert(1, 99)
print('After inserting 99 in 1st position: ', arr)

# remove an element from arr
arr.remove(20)
print('After removing 20: ', arr)

# remove last element using pop() method
n = arr.pop()
print('Array after using pop(): ', arr)
print('Popped element: ', n)

# finding position of element using index() method
n = arr.index(30)
print('First occurrence of element 30 is at: ', n)

# convert an array into a list using tolist() method
lst = arr.tolist()
print('List: ', lst)
print('Array: ', arr)
```

Types of Arrays

Till now, we got good idea on arrays in Python. When talking about arrays, any programming language like C or Java offers two types of arrays. They are:

- **Single dimensional arrays:** These arrays represent only one row or one column of elements. For example, marks obtained by a student in 5 subjects can be written as 'marks' array, as:

```
marks = array('i', [50, 60, 70, 66, 72])
```

The above array contains only one row of elements. Hence it is called single dimensional array or one dimensional array.

- **Multi-dimensional arrays:** These arrays represent more than one row and more than one column of elements. For example, marks obtained by 3 students each one in 5 subjects can be written as 'marks' array as:

```
marks = array([[50, 60, 70, 66, 72],
               [60, 62, 71, 56, 70],
               [55, 59, 80, 68, 65]])
```

The first student's marks are written in first row. The second student's marks are in second row and the third student's marks are in third row. In each row, the marks in 5 subjects are mentioned. Thus this array contains 3 rows and 5 columns and hence it is called multi-dimensional array.

Comparing Arrays

We can use the relational operators >, >=, <, <=, == and != to compare the arrays of same size. These operators compare the corresponding elements of the arrays and return another array with Boolean type values. It means the resultant array contains elements which are True or False. Observe the Program 25.

Program

Program 25: A Python program to compare two arrays and display the resultant Boolean type array.

```
# to know the result of comparing two arrays
from numpy import *
a = array([1,2,3,0])
b = array([0,2,3,1])

c = a == b
print('Result of a==b: ', c)
c = a > b
print('Result of a>b: ', c)
c = a <= b
print('Result of a<=b: ', c)
```

Output:

```
C:\>python arr.py
Result of a==b: [False True True False]
Result of a>b: [True False False False]
Result of a<=b: [False True TrueTrue]
```

The any() function can be used to determine if any one element of the array is True. The all() function can be used to determine whether all the elements in the array are True. The any() and all() functions return either True or False. This is shown in Program 26.

Program

Program 26: A Python program to know the effects of any() and all() functions.

```
# using any() and all() functions
from numpy import *
a = array([1,2,3,0])
b = array([0,2,3,1])

c = a > b
print('Result of a>b: ', c)

print('Check if any one element is true: ', any(c))
print('Check if all elements are true: ', all(c))

if(any(a>b)):
    print('a contains atleast one element greater than those of b')
```

Output:

```
C:\>python arr.py
Result of a>b:  [ True False False False]
Check if any one element is true:  True
Check if all elements are true:  False
a contains atleast one element greater than those of b
```

The functions logical_and(), logical_or() and logical_not() are useful to get the Boolean array as a result of comparing the compound condition. We already knew that any compound condition is a combination of more than one simple condition. For example,

```
c = logical_and(a>0, a<4)
```

In the above statement, the logical_and() function applies the compound condition a>0 and a<4 on every element of the array 'a' and returns another array 'c' that contains True or False values. We should understand that Python represents 0 as False and any other number as True. See Program 27 for further understanding of these functions.

Program

Program 27: A Python program to understand the use of logical functions on arrays.

```
# using logical_and(), logical_or(), logical_not()
from numpy import *
a = array([1,2,3], int)
b = array([0,2,3], int)

c = logical_and(a>0, a<4)
print(c)

c = logical_or(b>=0, b==1)
print(c)

c = logical_not(b)
print(c)
```

Output:

```
C:\>python arr.py
[ True True  True]
[ True True  True]
[ True False False]
```

The where() function can be used to create a new array based on whether a given condition is True or False. The syntax of the where() function is:

```
array = where(condition, expression1, expression2)
```

If the 'condition' is true, 'expression1' is executed and the result is stored into the array, else 'expression2' is executed and its result is stored into the array. For example,

```
a = array([10, 21, 30, 41, 50], int)
c = where(a%2==0, a, 0)
```

The new array 'c' contains elements from 'a' based on the condition a%2==0. This condition is applied to each element of the array 'a' and if it is True, the element of 'a' is stored into 'c' else '0' is stored into c. Hence the array 'c' looks like this: [10 0 30 0 50]. We can understand the where() function from the Program 28 in a better manner.

Program

Program 28: A Python program to compare the corresponding elements of two arrays and retrieve the biggest elements.

```
# using where() function.
from numpy import *
a = array([10, 20, 30, 40, 50], int)
b = array([1, 21, 3, 40, 51], int)
# if a>b then take element from a else from b
c = where(a>b, a, b)
print(c)
```

Output:

```
C:\>python arr.py
[10  21  30  40  51]
```

The nonzero() function is useful to know the positions of elements which are non zero. This function returns an array that contains the indexes of the elements of the array which are not equal to zero. For example,

```
a = array([1, 2, 0, -1, 0, 6], int)
```

In the preceding array, the elements which are non zero are: 1, 2, -1 and 6. Their positions or indexes are: 0, 1, 3, 5. These indexes can be retrieved into another array 'c' as:

```
c = nonzero(a)
```

Program

Program 29: A Python program to retrieve non zero elements from an array.

```
# using nonzero() function.
from numpy import *
a = array([1, 2, 0, -1, 0, 6], int)
```

```
# retrieve indexes of non zero elements from a
c = nonzero(a)

# display indexes
for i in c:
    print(i)

# display the elements
print(a[c])
```

Output:

```
C:\>python arr.py
[0  1  3  5]
[  1   2   -1   6]
```

Aliasing the Arrays

If 'a' is an array, we can assign it to 'b', as:

```
b = a
```

This is a simple assignment that does not make any new copy of the array 'a'. It means, 'b' is not a new array and memory is not allocated to 'b'. Also, elements from 'a' are not copied into 'b' since there is no memory for 'b'. Then how to understand this assignment statement? We should understand that we are giving a new name 'b' to the same array referred by 'a'. It means the names 'a' and 'b' are referencing same array. This is called 'aliasing'. Figure 7.5 shows the aliasing of array a as b:

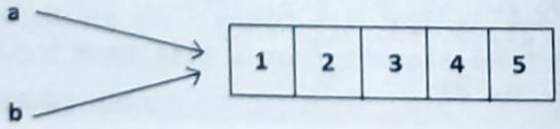

Figure 7.5: Aliasing the Array a as b

'Aliasing' is not 'copying'. Aliasing means giving another name to the existing object. Hence, any modifications to the alias object will reflect in the existing object and vice versa. To know about this, please refer to Program 30.

Program

Program 30: A Python program to alias an array and understand the affect of aliasing.

```
# aliasing an array.
from numpy import *

a = arange(1, 6)  # create a with elements 1 to 5.
b = a  # give another name b to a

print('Original array: ', a)
print('Alias array: ', b)

b[0]=99  # modify 0th element of b
```

```
print('After modification: ')
print('Original array: ', a)
print('Alias array: ', b)
```

Output:

```
C:\>python arr.py
Original array:  [1 2 3 4 5]
Alias array:  [1 2 3 4 5]
After modification:
Original array:  [99  2  3  4  5]
Alias array:  [99  2  3  4  5]
```

Viewing and Copying Arrays

We can create another array that is same as an existing array. This is done by the view() method. This method creates a copy of an existing array such that the new array will also contain the same elements found in the existing array. The original array and the newly created arrays will share different memory locations. If the newly created array is modified, the original array will also be modified since the elements in both the arrays will be like mirror images. Figure 7.6 explains how to create a view of an array:

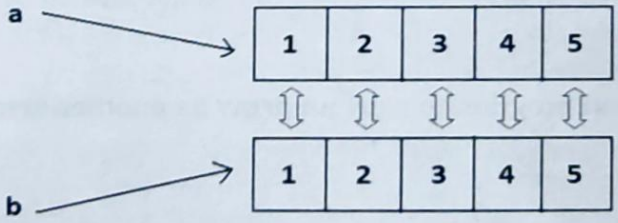

Figure 7.6: Creating a View of an Array

Program

Program 31: A Python program to create a view of an existing array.

```
# creating view for an array
from numpy import *

a = arange(1, 6)  # create a with elements 1 to 5.
b = a.view()  # create a view of a and call it b
print('Original array: ', a)
print('New array: ', b)

b[0]=99  # modify 0th element of b

print('After modification: ')
print('Original array: ', a)
print('New array: ', b)
```

Output:

```
C:\>python arr.py
Original array:  [1 2 3 4 5]
New array:  [1 2 3 4 5]
```

```
After modification:
Original array:  [99  2  3  4  5]
New array:  [99  2  3  4  5]
```

Viewing is nothing but copying only. It is called 'shallow copying' as the elements in the view when modified will also modify the elements in the original array. So, both the arrays will act as one and the same. Suppose we want both the arrays to be independent and modifying one array should not affect another array, we should go for 'deep copying'. This is done with the help of copy() method. This method makes a complete copy of an existing array and its elements. When the newly created array is modified, it will not affect the existing array or vice versa. There will not be any connection between the elements of the two arrays. Figure 7.7 shows copying an array as another array:

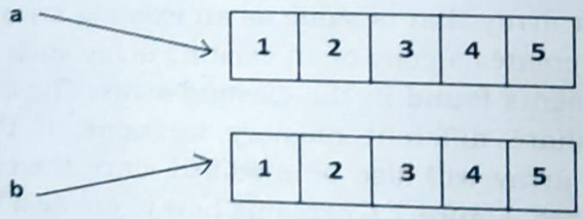

Figure 7.7: Copying an Array as another Array

Program

Program 32: A Python program to copy an array as another array.

```
# copying an array.
from numpy import *

a = arange(1, 6)  # create a with elements 1 to 5.
b = a.copy()  # create a copy of a and call it b

print('Original array: ', a)
print('New array: ', b)

b[0]=99  # modify 0th element of b

print('After modification: ')
print('Original array: ', a)
print('New array: ', b)
```

Output:

```
C:\>python arr.py
Original array:  [1 2 3 4 5]
New array:  [1 2 3 4 5]
After modification:
Original array:  [1 2 3 4 5]
New array:  [99  2  3  4  5]
```

If we display the elements of this 3D array using print(arr3), it shows the following output:

```
[[[1 2 3]
  [4 5 6]]

 [[1 1 1]
  [1 0 1]]]
```

The 2D arrays and 3D arrays are called multi-dimensional arrays in general.

Attributes of an Array

Numpy's array class is called *ndarray*. It is also known by alias name *array*. Let's remember that there is another class 'array' in Python that is different from numpy's 'array' class. This class contains the following important attributes (or variables):

The ndim Attribute

The 'ndim' attribute represents the number of dimensions or axes of the array. The number of dimensions is also referred to as 'rank'. For a single dimensional array, it is 1 and for a two dimensional array, it is 2. Consider the following code snippet:

```
arr1 = array([1,2,3,4,5])  # 1D array
print(arr1.ndim)
```

The preceding lines of code will display the following output:

```
1
```

Now, consider the following lines of code:

```
arr2 = array([[1,2,3], [4,5,6]])  # 2D array with 2 rows and 3 elements
print(arr2.ndim)
```

The preceding lines of code will display the following output:

```
2
```

The shape Attribute

The 'shape' attribute gives the shape of an array. The shape is a tuple listing the number of elements along each dimension. A dimension is called an axis. For a 1D array, shape gives the number of elements in the row. For a 2D array, it specifies the number of rows and columns in each row. We can also change the shape using 'shape' attribute. Consider the following code snippet:

```
arr1 = array([1,2,3,4,5])
print(arr1.shape)
```

The preceding lines of code will display the following output:

```
(5,)   # no. of elements
```

Now, consider the following lines of code:

```
arr2 = array([[1,2,3], [4,5,6]])
print(arr2.shape)
```

It will display the following output:

```
(2, 3)  # 2 rows and 3 cols
```

Also, consider the following lines of code:

```
arr2.shape = (3, 2)  # change shape of arr2 to 3 rows and 2 cols
print(arr2)
```

The preceding lines of code will display the following output:

```
[[1  2]
 [3  4]
 [5  5]]
```

The size Attribute

The 'size' attribute gives the total number of elements in the array. For example, consider the following code snippet:

```
arr1 = array([1,2,3,4,5])
print(arr1.size)
```

The preceding lines of code will display the following output:

```
5
```

Now, consider the following lines of code:

```
arr2 = array([[1,2,3], [4,5,6]])
print(arr2.size)
```

It will display the following output:

```
6
```

The itemsize Attribute

The 'itemsize' attribute gives the memory size of the array element in bytes. As we know, 1 byte is equal to 8 bits. For example consider the following code snippet:

```
arr1 = array([1,2,3,4,5])
print(arr1.itemsize)
```

The preceding lines of code will display the following output:

```
4
```

Now, consider the following lines of code:

```
arr2 = array([1.1,2.1,3.5,4,5.0])
print(arr2.itemsize)
```

The preceding lines of code will display the following output:

```
8
```

The dtype Attribute

The 'dtype'attribute gives the datatype of the elements in the array. For example, consider the following code snippet:

```
arr1 = array([1,2,3,4,5])  # integer type array
print(arr1.dtype)
```

The preceding lines of code will display the following output:

```
int32
```

Now, consider the following lines of code:

```
arr2 = array([1.1,2.1,3.5,4,5.0])  # float type array
print(arr2.dtype)
```

It will display the following output:

```
float64
```

The nbytes Attribute

The 'nbytes' attributegives the total number of bytes occupied by an array. The total number of bytes = size of the array * item size of each element in the array. For example,

```
arr2 = array([[1,2,3], [4,5,6]])
print(arr2.nbytes)
```

The preceding lines of code will display the following output:

```
24
```

Apart from the attributes discussed in the preceding sections, we can use reshape() and flatten() methods which are useful to convert the 1D array into a 2D array and vice versa.