Control Statements

Control statements are statements which control or change the flow of execution. The following are the control statements available in python:

- If statement
- If…else statement
- If…elif…else statement

1) The if Statement

This statement is used to execute one or more statement depending on whether a condition is True or not.

Syntax :

```
if condition:
        statements
```

First, the condition is tested. If the condition is True, then the statements given after colon(:) are executed. We can write one or more statements after colon(:). If the condition is false, then the statements mentioned after colon are not executed.

Example :

```
num=1
if num==1:
        print(“one”)
```

Output :

```
one
```

We can also write a group of statements after colon. The group of statements in python is called **suite**. While writing a group of statements, we should write them all with proper indentation. Indentation represents the spaces left before the statements. The default indentation used in python is 4 spaces.

Example :

```
str=’yes’
if str==’yes’:
```

```
        print("Yes")
        print("This is what you said")
        print("Your response is good")
```

Output :

```
Yes
This is what you said
Your response is good
```

2) The if…else Statement

The if…else statement executes a group of statements when a condition is True;

Otherwise, it will execute another group of statements.

Syntax :

```
if condition:
        statements1
else:
        statements2
```

If the condition is True, then it will execute statements1 and if the condition is False, then it will execute statements2. It is advised to use 4 spaces as indentation before statements1 and statements2. In example we trying to display whether a given number is even or odd. The logic is simple. If the number is divisible by 2 , then it is an even number; otherwise, it is an odd number. To know whether a number is divisible by 2 or not, we can use modulus operator gives remainder of division. If the remainder is 0, then the number is divisible, otherwise not.

Example :

```
x=10
if x%2==0:
        print(x,"is even number")
else:
```

```
        print(x,"is odd number")
```

Output :

```
10 is even number
```

Example :

```
x=int(input("ENTER A NUMBER : "))
if x%2==0:
        print(x,"is even number")
else:
        print(x,"is odd number")
```

Output :

```
ENTER A NUMBER : 11
11 is odd number
```

3) The if…elif..else Statement

Sometimes, the programmer has to test multiple conditions and execute statements depending on those conditions. if…elif…else statement is useful in such situations.

Syntax :

```
if condition1:
        statements1
elif condition2:
        statements2
elif condition3:
        statements3
else:
```

```
        statements4
```

When condition1 is True, the statements1 will be executed. If condition1 is False, then condition2 is evaluated. When condition2 is True, the statements2 will be executed. When condition2 is False, the condition3 is tested. If condition3 is True, then statements3 will be executed. When condition3 is False, the statements4 will be executed. It means statements4 will be executed only if none of the conditions are True.

Example :

```
num=5
if num==0:
        print(num,"is Zero")
else:
        print(x,"is odd number")
```

Output :

```
ENTER A NUMBER : 11

11 is odd number
```

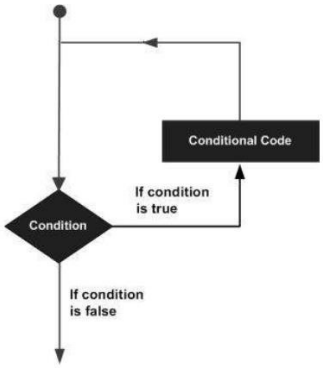
Looping

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

Python programming language provides following types of loops to handle looping requirements.



Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

While Loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax of a while loop :

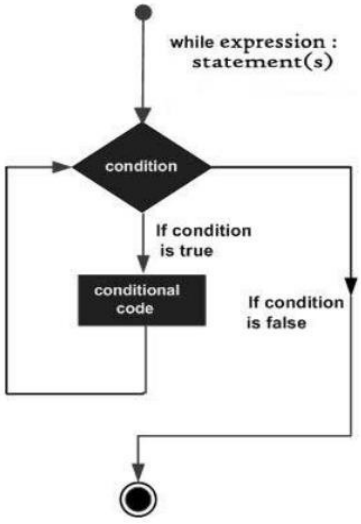
```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
count = 0

while (count < 9):
```

```
    print('The count is:', count)

    count = count + 1



    print ("Good bye!")
```

Output :

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is nolonger less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

For Loop

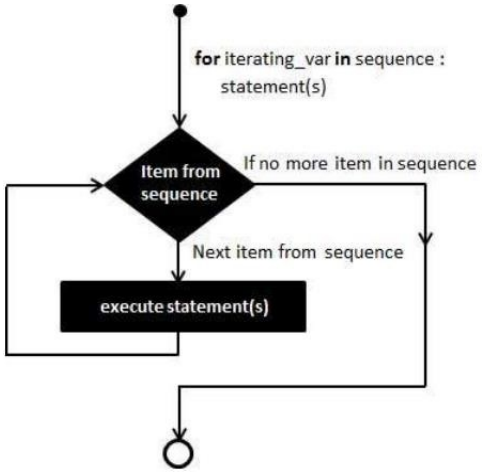
It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax of a for loop :

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



Example

```
for letter in 'Python':     # First Example
   print ('Current Letter :', letter)



fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print ('Current fruit :', fruit)



print ("Good bye!")
```

Output :

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Nested Loop

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax for a nested for loop :

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

Syntax for a nested while loop

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print i, " is prime"
   i = i + 1

print "Good bye!"
```

Output :

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
```

```
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
Good bye!
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break statement

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

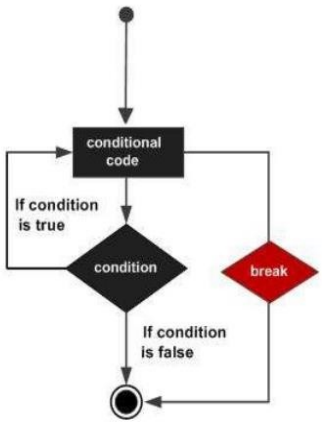
If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in Python is as follows −

```
break
```

Flow Diagram



Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

Output :

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

Continue statement

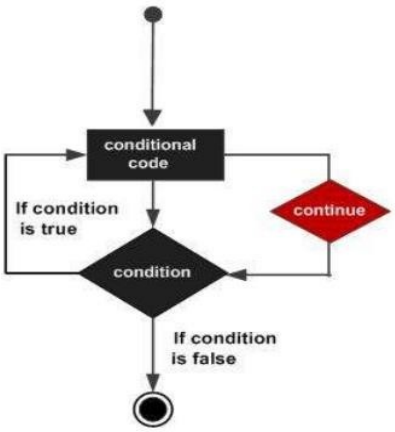
It returns the control to the beginning of the while loop.. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

Syntax

```
continue
```

Flow Diagram



Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
    var = var -1
    if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

Output :

```
Current Letter : P
Current Letter : y
Current  Letter  :  t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
```

```
Good bye!
```

Pass statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) −

Syntax

```
pass
```

Example

```
for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter

print "Good bye!"
```

Output :

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

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.

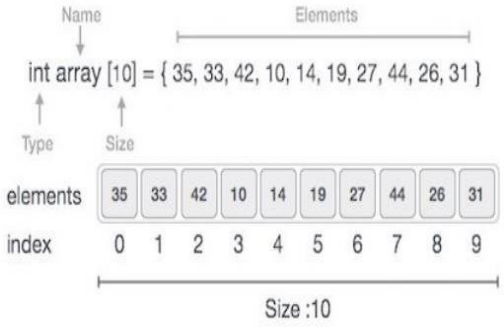
- **Element**− Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

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.

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.

Creating an Array

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

```
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 typecodes used are:

Typecode	Value	Minimum size in bytes
b	Represents signed integer	1
B	Represents unsigned integer	1
i	Represents signed integer	2
I	Represents unsigned integer	2
l	Represents signed integer	4
L	Represents unsigned integer	4
f	Represents floating point	4
d	Represents double precision floating point	8
u	Represents 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’ fro integer type array. After that the elements should be written inside the square braces [ ] as,

```
a = array('i', [10,20,30])
```

This is creating an array whose name is ‘a’ with integer type elements 10,20 and 30.

To create a float type array,

```
array2 = array(‘d’,[1.5,-2.2,3.0])
```

This is creating an array the name is ‘arr’ with float type elements 1.5, -2.2 and 3.0 . The type code is ‘d’ which represents double type elements each taking 8 bytes memory.

Basic Operations

Following are the basic operations supported by an array.

- **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.

Accessing Array Element

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

```
from array import *

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

print (array1[0])

print (array1[2])
```

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
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.

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.

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)
```

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

Output :

```
10
20
30
50
```

Search Operation

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

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))
```

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 th eprogram returns an error.

Output :

```
3
```

Update Operation

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

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)
```

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

Output :

```
10
20
80
40
50
```

Looping Array

Using For loop

Example:

```
import array
a=array.array('i',[5,6,-7,8])
for x in a:
    print(x)
```

Output:

```
5
6
-7
8
```

Using While loop

Example:

```
from array import *
x=array('i',[10,20,30,40,50])
n=len(x)            #find number of elements in the array
#display array elements using indexing
i=0
while i<n:
    print(x[i])
    i+=1
```

Output:

```
10 20 30 40 50
```

How to Importing the Array Module?

There are three ways to importing the array module into our program.

The **first** way is to import the entire array module using import statements 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.
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 alternative 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 written:

```
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.

Example:

```
import array
a=array.array(‘i’,[5,6,-7,8])
for x in a:
    print(x)
```

Output:

```
5
6
-7
8
```

Example:

```
from array import *
a=array.array(‘i’,[5,6,-7,8])
for x in a:
    print(x)
```

Output:

```
5
6
-7
8
```

Slicing the Array

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 elements by elements from the array, slicing is useful to retrieve a range of elements. The general format of 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.

```
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 elements.

Example :

```
# 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 :

```
30

40

50
```

Example :

```
# using slicing to display elements of an array

from array import  *

x=array('i',[10,20,30,40,50,60,70])


# display elements from 1st  to 3rd from x

y=x[1:4]

print(y)


# display elements from 0th till the last element in x

y=x[0:]

print(y)


# display elements from 0th  to 3rd from x

y=x[:4]

print(y)


# display elements last four from x

y=x[-4:]

print(y)


y=x[-4:-1]

print(y)


#display elements into 0th  to  7th elements from x
```

```
y=x[0:7:2]

print(y)
```

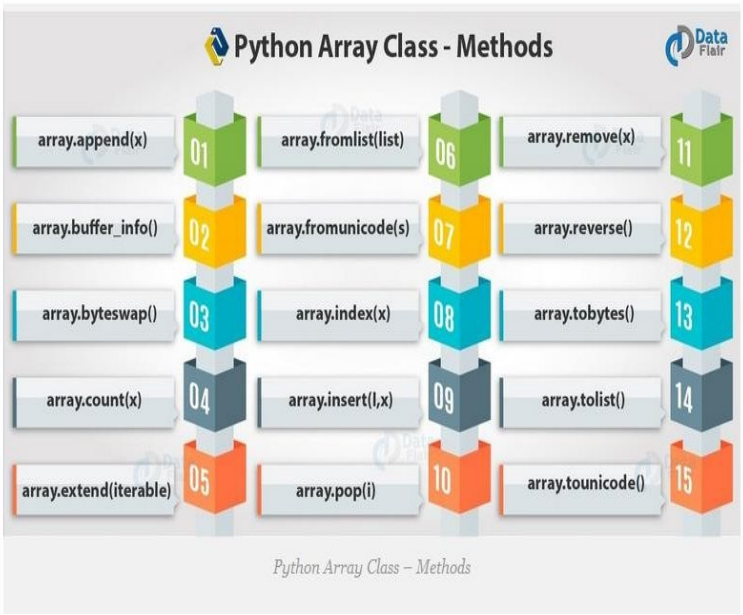
Output :

```
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])
```

Processing the Array

➢ Array Methods

Now, which methods does Array Class support? Here you go:



Python Array Class – Methods

1. array.append(x)

This appends the item *x* to the array.

```
>>> arr.append(2)

>>> arr
```

Output :

```
array('i', [1, 3, 4, 2])
```

2. array.buffer_info()

This returns a tuple that holds the address in memory and the length of elements in the buffer that holds the contents of the array.

```
>>> arr.buffer_info()
```

Output :

(43560864, 4)

3. array.byteswap()

This performs an operation of bytes wap on an array.

```
>>> arr.byteswap()

>>> arr
```

Output :

```
array('i', [16777216, 50331648, 67108864, 33554432])
```

4. array.count(x)

Let's find out how many 3s there are in our Python array.

```
>>> arr=array.array('i',[1,3,2,4,3,5])

>>> arr.count(3)
```

Output :

```
2
```

5. array.extend(iterable)

This attaches the iterable to the end of the array in Python.

```
>>> arr.extend([7,9,8])

>>> arr
```

Output :

array(‘i’, [1, 3, 2, 4, 3, 5, 7, 9, 8])

But if you add another array, make sure it is the same type. The following code throws an error.

```
>>> arr.extend(array.array('u',['H','e','l','l','o']))
```

Output :

```
Traceback (most recent call last):
File “<pyshell#19>”, line 1, in <module>
arr.extend(array.array(‘u’,[‘H’,’e’,’l’,’l’,’o’]))
TypeError: can only extend with array of same kind
```

6. array.fromlist(list)

This appends item from a list to the Python arrays.

```
>>> arr.fromlist([9,0])

>>> arr
```

Output :

```
array(‘i’, [1, 3, 2, 4, 3, 5, 7, 9, 8, 9, 0])
```

7. array.fromunicode(s)

This appends the Unicode string to the one we call it on- this should be Unicode too.

```
>>> unicodearr=array.array('u','Hello')

>>> unicodearr
```

Output :

```
array(‘u’, ‘Hello’)

>>> unicodearr.fromunicode(' world')

>>> unicodearr
```

Output :

```
array(‘u’, ‘Hello world’)
```

8. array.index(x)

This returns the index for the first occurrence of *x* in the Python array.

```
>>> arr=array.array('i',[1,3,2,4,3,5])

>>> arr.index(3)
```

Output :

```
1
```

9. array.insert(I,x)

```
>>> arr.insert(2,7)

>>> arr
```

Output :

```
array(‘i’, [1, 3, 7, 2, 4, 3, 5])
```

This inserts the element 7 at index 2.

10. array.pop(i)

This lets us drop the element at the position i.

```
>>> arr.pop(2)
```

Output :

```
7
```

11. array.remove(x)

This will let you remove the first occurrence of an element from the Python array.

```
>>> arr.remove(3)

>>> arr
```

Output :

```
array('i', [1, 2, 4, 3, 5])
```

12. array.reverse()

This reverses the Python array.

```
>>> arr.reverse()

>>> arr
```

Output :

```
array('i', [5, 3, 4, 2, 1])
```

13. array.tobytes()

This returns a representation in bytes of the values of the array in Python.

This is the same as array.tostring(), which is deprecated.

```
>>> arr.tobytes()
```

Output :

```
b'\x05\x00\x00\x00\x03\x00\x00\x00\x04\x00\x00\x00\x02\x00\x00\x00\x01\x00\x00\x00'
```

14. array.tolist()

This converts the array into a list.

```
>>> arr.tolist()
```

Output :

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

15. array.tounicode()

This converts an array to a Unicode string. You need a Unicode array for this.

```
>>> unicodearr.tounicode()
```

Output :

```
'Hello world'
```

➢ Variables of Array Class

The class *array* has the following data items-

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

1. array.typecode

This gives us the type code character we used when creating the array in Python.

```
>>> arr.typecode
```

Output :

```
'i'
```

2. array.itemsize

This returns the number of bytes one item from the Python array takes internally.

```
>>> arr.itemsize
```

Output :

```
4
```

Example :

```
#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 and 60 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
```

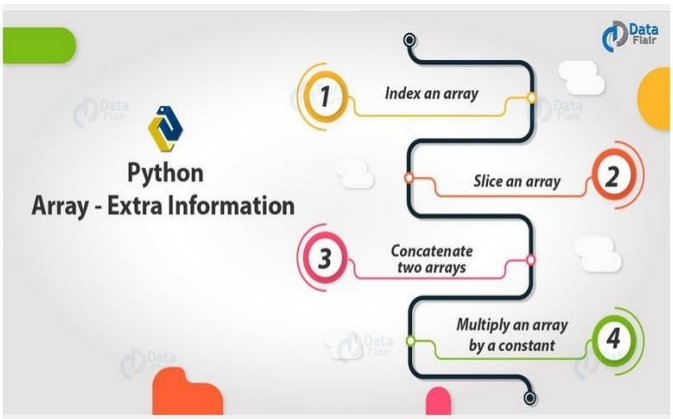
```
lst=arr.tolist()

print('List:',lst)

print('Array:',arr)
```

Output :

```
Original array: array('i', [10, 20, 30, 40, 50])
After appending 30 and 60: array('i', [10, 20, 30, 40, 50, 30, 60])
After inserting 99 in 1st position: array('i', [10, 99, 20, 30, 40, 50, 30,
60])
After removing 20: array('i', [10, 99, 30, 40, 50, 30, 60])
Array after using pop(): array('i', [10, 99, 30, 40, 50, 30])
Popped element: 60
First occurrence of element 30 is at: 2
List: [10, 99, 30, 40, 50, 30]
Array: array('i', [10, 99, 30, 40, 50, 30])
```

Python Array – More Information

Python Arrays are space-efficient collections of numeric values that are uniformly-typed. You can:



Python Arrays Module – More Information

1. How to Index an Array in Python?

>>> arr

Output

array('i', [5, 3, 4, 2, 1])

>>> arr[1]

Output

3

2. Slice an array

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

Output

array(‘i’, [3, 4, 2])

3. Concatenate two arrays in Python

```
>>> arr+arr
```

Output

array(‘i’, [5, 3, 4, 2, 1, 5, 3, 4, 2, 1])

4. Multiply an array by a constant

```
>>> arr*2
```

Output

array(‘i’, [5, 3, 4, 2, 1, 5, 3, 4, 2, 1])

So, this was all about Python Arrays Tutorial. Hope you like our explanation