

Python Programming - II

-By Nimesh Kumar Dagur, CDAC,
India

Python Strings

- Strings in Python are identified as a contiguous set of characters in between quotation marks.
- Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator

Working With Strings

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways.
- They can be enclosed in single quotes or double quotes:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Working With Strings

- String literals can span multiple lines in several ways.
- Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:
- Note that newlines still need to be embedded in the string using `\n`; the newline following the trailing backslash is discarded.
- strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`.
- End of lines do not need to be escaped when using triple-quotes, but they will be included in the string.

Working With Strings

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
significant."
```



```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Working With Strings

- Strings can be concatenated with the + operator, and repeated with *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Working With Strings

- Two string literals next to each other are automatically concatenated;
- the first line above could also have been written
word = 'Help' 'A';
- this only works with two literals, not with arbitrary string expressions:

```
>>> 'str' 'ing'  
'string'
```

```
# <- This is ok
```

Working With Strings

- Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0.
- substrings can be specified with the ***slice notation***: *two indices separated by a colon*.

word="HelpA"

```
>>> word[4]
```

```
'A'
```

```
>>> word[0:2]
```

```
'He'
```

```
>>> word[2:4]
```

```
'lp'
```


Working With Strings

- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]      # The first two characters
'He'
>>> word[2:]      # Everything except the first two characters
'lpA'
```

Working With Strings

- creating a new string with the combined content is easy and efficient:

```
>>> 'x' + word[1:]  
'xelpA'  
>>> 'Splat' + word[4]  
'SplatA'
```

- Here's a useful invariant of slice operations:
 $s[:i] + s[i:]$ equals s .

```
>>> word[:2] + word[2:]  
'HelpA'  
>>> word[:3] + word[3:]  
'HelpA'
```

Working With Strings

- Indices may be negative numbers, to start counting from the right.

```
>>> word[-1]      # The last character
'A'
>>> word[-2]      # The last-but-one character
'p'
>>> word[-2:]      # The last two characters
'pA'
>>> word[:-2]      # Everything except the last two characters
'Hel'
```

But note that -0 is really the same as 0, so it does not count from the right!

```
>>> word[-0]      # (since -0 equals 0)
'H'
```

Working With Strings

- The built-in function *len()* returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'  
>>> len(s)  
34
```

String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length

String Formatting Operator

- One of Python's coolest features is the string format operator %.
- This operator is unique to strings and makes up for the pack of having functions from C's printf() family

```
print("My name is %s and weight is %d kg!" % ('Zara', 21))
```



```
My name is Zara and weight is 21 kg!
```

String Formatting Operator

List of set of symbols which can be used along with %

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number

String Formatting : Example(s)

```
>>> a=12
>>> b=20
>>> s=a+b
>>> s
32
>>> print("the result of addition of ",a," and ",b," is ",s)
the result of addition of 12 and 20 is 32
>>> print("the result of addition of %d and %d is %d"%(a,b,s))
the result of addition of 12 and 20 is 32
>>> print("the result of addition of {0} and {1} is {2} ".format(a,b,s))
the result of addition of 12 and 20 is 32
>>> print("the result of addition of "+a+" and "+b+" is "+s)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print("the result of addition of "+a+" and "+b+" is "+s)
TypeError: Can't convert 'int' object to str implicitly
>>> print("the result of addition of "+str(a)+" and "+str(b)+" is "+str(s))
the result of addition of 12 and 20 is 32
```


Built-in String Methods

- **capitalize():** It returns a copy of the string with only its first character capitalized.

str.capitalize ()

```
str = "this is string example....wow!!!";  
print( "str.capitalize() : ", str.capitalize())
```



```
str.capitalize() : This is string example....wow!!!
```

Built-in String Methods

- **count()**: The method **count()** returns the number of occurrences of substring sub in the range [start, end].

Syntax

```
str.count(sub, start= 0,end=len(string))
```

```
str = "this is string example....wow!!!";  
  
sub = "i";  
print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40)  
sub = "wow";  
print "str.count(sub) : ", str.count(sub)
```



```
str.count(sub, 4, 40) : 2  
str.count(sub, 4, 40) : 1
```

Built-in String Methods

- **find():** It determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

Syntax

```
str.find(str, beg=0, end=len(string))
```

Index if found and -1 otherwise.

```
>>> s = 'hello world'
>>> s.find('l', 0, len(s))
2
>>> s.find('l', 1, len(s))
2
>>> s.find('l', 4, len(s))
9
>>> s.find('l', 2, len(s))
2
>>> s.find('l', 3, len(s))
3
```

Built-in String Methods

- **lower()**: The method **lower()** returns a copy of the string in which all case-based characters have been lowercased.

```
str.lower()
```

```
str = "THIS IS STRING EXAMPLE....WOW!!!";  
print str.lower()
```



```
this is string example....wow!!!
```

Built-in String Methods

- **upper()** :The method **upper()** returns a copy of the string in which all case-based characters have been uppercased.
- **swapcase()**: The method **swapcase()** returns a copy of the string in which all the case-based characters have had their case swapped.
- **title()**: The method **title()** returns a copy of the string in which first characters of all the words are capitalized.
- **replace()** : The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

```
str.replace(old, new[, max])
```

```
>>> s = 'hello world'
>>> s.swapcase()
'HELLO WORLD'
>>> s = 'HELLO WORLD'
>>> s.swapcase()
'hello world'
>>> s = 'Hello World'
>>> s.swapcase()
'hELLO wORLD'
>>> s = 'Hello world'
>>> s.swapcase()
'hELLO WORLD'
>>> s = 'hello world'
>>> s.title()
'Hello World'
>>> s = 'hello world'
>>> s.upper()
'HELLO WORLD'
```

```
>>> s = 'Hello World'
>>> s.replace('l', 'x')
'Hexxo Worxd'
>>> s.replace('l', 'x', 2)
'Hexxo World'
>>> s.replace('l', 'x', 1)
'Hexlo World'
```

Data Type Conversion

- Sometimes, you may need to perform conversions between the built-in types.
- To convert between types, you simply use the type name as a function.
- There are several built-in functions to perform conversion from one data type to another.
- These functions return a new object representing the converted value.

Data Type Conversion

Function	Description
<code>int(x [,base])</code>	Converts x to an integer. base specifies the base if x is a string.
<code>long(x [,base])</code>	Converts x to a long integer. base specifies the base if x is a string.
<code>float(x)</code>	Converts x to a floating-point number.
<code>complex(real [,imag])</code>	Creates a complex number.
<code>str(x)</code>	Converts object x to a string representation.

Data Type Conversion

<code>chr(x)</code>	Converts an integer to a character.
<code>ord(x)</code>	Converts a single character to its integer value.
<code>hex(x)</code>	Converts an integer to a hexadecimal string.
<code>oct(x)</code>	Converts an integer to an octal string.

Python Operators

- Operators are special symbols in Python that carry out arithmetic or logical computation.
- The value that the operator operates on is called the operand.

```
>>> 2+3  
5
```

Type of operators in Python

Python has a number of operators which are classified below:

- **Arithmetic operators**
- **Comparison (Relational) operators**
- **Logical (Boolean) operators**
- **Assignment operators**
- **Special operators:** Identity operators & membership operator

Arithmetic operators

Operator	Meaning	Example
+	Add two operands or unary plus	$x + y$ $+2$
-	Subtract right operand from the left or unary minus	$x - y$ -2
*	Multiply two operands	$x * y$
/	Divide left operand by the right one	x / y
%	Modulus - remainder of the division of left operand by the right	$x \% y$ (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	$x // y$
**	Exponent - left operand raised to the power of right	$x**y$ (x to the power y)

Arithmetic operators

```
x = 15
y = 4
print('x + y = ',x+y)
print('x - y = ',x-y)
print('x * y = ',x*y)
print('x / y = ',x/y)
print('x // y = ',x//y)
print('x ** y = ',x**y)
```

Output

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

In Python 3.x

3.75

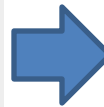
Comparison operators

- Comparison operators are used to compare values.
- It either returns True or False according to the condition

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	<code>x > y</code>
<	Less than - True if left operand is less than the right	<code>x < y</code>
==	Equal to - True if both operands are equal	<code>x == y</code>
!=	Not equal to - True if operands are not equal	<code>x != y</code>
>=	Greater than or equal to - True if left operand is greater than or equal to the right	<code>x >= y</code>
<=	Less than or equal to - True if left operand is less than or equal to the right	<code>x <= y</code>

Comparison operators

```
x = 10
y = 12
print('x > y is',x>y)
print('x < y is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```



Output

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

Logical operators

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

- **True, False** : True and False are truth values in Python. They are the results of comparison operations or logical (Boolean) operations in Python.
- True and False in python is same as 1 and 0

Logical operators

```
>>> 1 == 1
True
>>> 5 > 3
True
>>> True or False
True
>>> 10 <= 1
False
>>> 3 > 7
False
>>> True and False
False
```

```
>>> True == 1
True
>>> False == 0
True
>>> True + True
2
```

Logical operators

Truth table for and

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Truth table for or

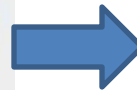
A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Truth tabel for not

A	not A
True	False
False	True

Logical operators

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```



```
x and y is False
x or y is True
not x is False
```

Assignment operators

- Assignment operators are used in Python to assign values to variables.
- `a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.
- There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Assignment operators

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
=	x = 5	x = x 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Special operators

- Python language offers some special type of operators like the **identity operator** or the **membership operator**.

Identity operators

- **is** and **is not** are the identity operators in Python.
- They are used to check if two values (or variables) are located on the same part of the memory.
- Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Identity operators

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]
print(x1 is not y1)
print(x2 is y2)
print(x3 is y3)
```

Output

```
False
True
False
```


Membership operators

- **in** and **not in** are the membership operators in Python.
- They are used to test whether a value or variable is found in a sequence (string, list, tuple and dictionary).
- In a dictionary we can only test for presence of key, not the value.

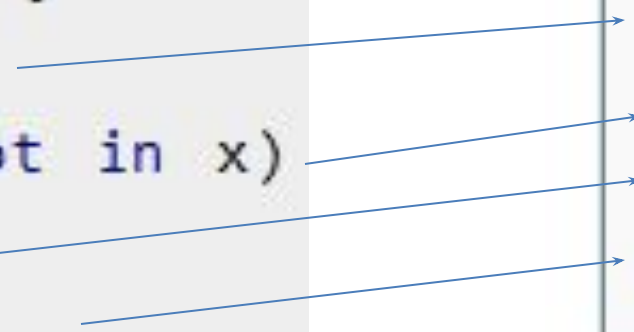
Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Membership operators

```
x = 'Hello world'
y = {1:'a',2:'b'}
print('H' in x)
print('hello' not in x)
print(1 in y)
print('a' in y)
```

Output

True
True
True
False

A diagram consisting of four blue arrows pointing from the code lines in the left box to the output values in the right box. The first arrow points from 'print('H' in x)' to 'True'. The second arrow points from 'print('hello' not in x)' to 'True'. The third arrow points from 'print(1 in y)' to 'True'. The fourth arrow points from 'print('a' in y)' to 'False'.

Order Of Operation

- First level of precedence: top to bottom
- Second level of precedence
 - If there are multiple operations that are on the same level then precedence goes from left to right.

()	Brackets (inner before outer)
**	Exponent
*, /, %	Multiplication, division, modulo
+, -	Addition, subtraction

Almost all operators except the exponent() support the Left-to-right Associativity.**

Python Operators Precedence

Operators from highest precedence to lowest

Operator	Description
<code>**</code>	Exponentiation (raise to the power)
<code>* / % //</code>	Multiply, divide, modulo and floor division
<code>+ -</code>	Addition and subtraction
<code>>> <<</code>	Right and left bitwise shift
<code>&</code>	Bitwise 'AND'
<code>^ </code>	Bitwise exclusive 'OR' and regular 'OR'
<code><= < > >=</code>	Comparison operators
<code><> == !=</code>	Equality operators
<code>= %= /= //= -= += *= **=</code>	Assignment operators
<code>is is not</code>	Identity operators
<code>in not in</code>	Membership operators
<code>not or and</code>	Logical operators

Highest



lowest

Python Operators Precedence

```
a = 20  
b = 10  
c = 15  
d = 5  
e = 0
```

```
e = (a + b) * c / d      #( 30 * 15 ) / 5  
print "Value of (a + b) * c / d is ", e
```

```
e = ((a + b) * c) / d    # (30 * 15) / 5  
print "Value of ((a + b) * c) / d is ", e
```

```
e = (a + b) * (c / d);    # (30) * (15/5)  
print "Value of (a + b) * (c / d) is ", e
```

```
e = a + (b * c) / d;      # 20 + (150/5)  
print "Value of a + (b * c) / d is ", e
```

Output

```
Value of (a + b) * c / d is 90  
Value of ((a + b) * c) / d is 90  
Value of (a + b) * (c / d) is 90  
Value of a + (b * c) / d is 50
```

Bit-wise operators

- **AND(&):** bits(0 and 1) are set to 1 if both the inputs are 1 otherwise 0.

And (&)		
Input1	input2	Output
0	0	0
0	1	0
1	0	0
1	1	1

- Or(|): bits are set to 1 if both the inputs or any one of the input is 1.

Input1	input2	Output
0	0	0
0	1	1
1	0	1
1	1	1

- XOR(^):bits are set to 1 if any one of the input is 1 otherwise 0.

Input1	input2	Output
0	0	0
0	1	1
1	0	1
1	1	0

shift operators: >>(right shift) and <<(left shift)

left shift(<<):

a=20 => 10100

00010100

a<<2 01010000

right shift(>>)

a>>2 000101

~(Python 1's complement):

-(number+1)

10 => 1010

~10 = -(1010+1)
= -(1011) binary
= -11 decimal

Python Lists:

- The most basic data structure in Python is the sequence.
- Each element of a sequence is assigned a number - its position or index.
- The first index is zero, the second index is one, and so forth.
- Most common sequences are lists and tuples.
- There are certain things you can do with all sequence types.
- These operations include indexing, slicing, adding, multiplying, and checking for membership.
- In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists:

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets ([]).
- To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.
- A list can even have another list as an item. These are called nested list.



```
# empty list
```

```
my_list = []
```

```
# list of integers
```

```
my_list = [1, 2, 3]
```

```
# list with mixed datatypes
```

```
my_list = [1, "Hello", 3.4]
```

```
# nested list
```

```
my_list = ["mouse", [8, 4, 6]]
```

Example

```
>>> list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
>>> tinylist = [123, 'john']
>>> print list
['abcd', 786, 2.23, 'john', 70.2]
>>> print list[0]
abcd
>>> print list[1:3]
[786, 2.23]
>>> print tinylist * 2
[123, 'john', 123, 'john']
>>> print list + tinylist
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Accessing Values in Lists

- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index

```
list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5, 6, 7 ];  
  
print "list1[0]: ", list1[0]  
print "list2[1:5]: ", list2[1:5]
```



```
list1[0]:  physics  
list2[1:5]:  [2, 3, 4, 5]
```


Accessing Values in Lists

```
>>> my_list = ['p','r','o','b','e']
>>> my_list[0]
'p'
>>> my_list[2]
'o'
>>> my_list[4]
'e'
>>> my_list[4.0]
...
TypeError: list indices must be integers, not float
>>> my_list[5]
...
IndexError: list index out of range

>>> n_list = ["Happy", [2,0,1,5]]
>>> n_list[0][1]      # nested indexing
'a'
>>> n_list[1][3]      # nested indexing
5
```

```
n_list = ["Happy", [2,0,1,5]]
```


Updating Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method.

```
list = ['physics', 'chemistry', 1997, 2000];  
  
print "Value available at index 2 : "  
print list[2]           1997  
list[2] = 2001;  
print "New value available at index 2 : "  
print list[2]           2001
```



```
Value available at index 2 :  
1997  
  
New value available at index 2 :  
2001
```

Python List `append()` Method

- The method **`append()`** appends a passed *obj* into the existing list.

```
list.append(obj)
```

```
aList = [123, 'xyz', 'zara', 'abc'];  
aList.append( 2009 );  
print "Updated List : ", aList
```



```
Updated List : [123, 'xyz', 'zara', 'abc', 2009]
```

Delete List Elements

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know

```
list1 = ['physics', 'chemistry', 1997, 2000];  
  
print list1  
del list1[2];  
print "After deleting value at index 2 : "  
print list1
```



```
['physics', 'chemistry', 1997, 2000]  
After deleting value at index 2 :  
['physics', 'chemistry', 2000]
```

Python List remove() Method

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];  
  
aList.remove('xyz');  
print "List :", aList  
aList.remove('abc');  
print "List :", aList
```



```
List : [123, 'zara', 'abc', 'xyz']  
List : [123, 'zara', 'xyz']
```

Basic List Operations

- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	1 2 3	Iteration

Indexing & Slicing

- Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods:

- **len():** The method **len()** returns the number of elements in the *list*.

```
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']  
  
print "First list length : ", len(list1)  
print "Second list length : ", len(list2)
```



```
First list length : 3  
Second list length : 2
```

Built-in List Functions & Methods:

- **max():** The method **max** returns the elements from the *list* with maximum value.
- **Note: in Python3.x version max is used only for same kind of values at a time**

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]  
  
print "Max value element : ", max(list1)  
print "Max value element : ", max(list2)
```



```
Max value element :  zara  
Max value element :  700
```


Built-in List Functions & Methods:

- **min()** : The method **min()** returns the elements from the *list* with minimum value.
- **Note: in Python3.x version min is used only for same kind of values at a time**

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]

print "min value element : ", min(list1)
print "min value element : ", min(list2)
```



```
min value element : 123
min value element : 200
```

Built-in List Functions & Methods:

- **count()** : The method **count()** returns count of how many times *obj* occurs in list.
- **Syntax:**

```
list.count(obj)
```

```
aList = [123, 'xyz', 'zara', 'abc', 123];  
  
print "Count for 123 : ", aList.count(123)  
print "Count for zara : ", aList.count('zara')
```



```
Count for 123 : 2  
Count for zara : 1
```

Built-in List Functions & Methods:

- **extend()** : The method **extend()** appends the contents of *seq* to list.
- *seq* is the list of elements.

```
aList = [123, 'xyz', 'zara', 'abc', 123];  
bList = [2009, 'manni'];  
aList.extend(bList)  
  
print "Extended List : ", aList
```



```
Extended List : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

```
>>> list = [1,2,3,4,'a']
```

```
>>> list1 = [10,20]
```

```
>>> list.append(list1)
```

```
>>> list
```

```
[1, 2, 3, 4, 'a', [10, 20]]
```

```
>>> list = [1,2,3,4,'a']
```

```
>>> list1 = [10,20]
```

```
>>> list.extend(list1)
```

```
>>> list
```

```
[1, 2, 3, 4, 'a', 10, 20]
```

Built-in List Functions & Methods:

- **insert()** : The method **insert()** inserts object *obj* into list at offset *index*.
- **Syntax:**

```
list.insert(index, obj)
```

```
aList = [123, 'xyz', 'zara', 'abc']  
aList.insert( 3, 2009)  
print "Final List : ", aList
```



```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```


Built-in List Functions & Methods:

- **index()** : The method **index()** returns the lowest index in list that *obj* appears

```
list.index(obj)
```

```
aList = [123, 'xyz', 'zara', 'abc'];  
  
print "Index for xyz : ", aList.index( 'xyz' )  
print "Index for zara : ", aList.index( 'zara' )
```



```
Index for xyz : 1  
Index for zara : 2
```

Built-in List Functions & Methods:

- **pop():** pop([*i*]) return and remove the item at position *i* (last item if *i* is not provided)

```
>>> alist = ['a', 1, 'b', 30]
>>> alist.pop()
30
>>> alist
['a', 1, 'b']
>>> alist.pop(1)
1
>>> alist
['a', 'b']
```

Built-in List Functions & Methods:

- **sort():** Sort items in a list in ascending order
- **reverse():** Reverse the order of items in a list
- **clear():** Remove all items and empty the list

```
>>> my_list = [3, 8, 1, 6, 0, 8, 4]
>>> my_list.sort()
>>> my_list
[0, 1, 3, 4, 6, 8, 8]
>>> my_list.reverse()
>>> my_list
[8, 8, 6, 4, 3, 1, 0]
>>> my_list.clear()
>>> my_list
[]
```