http://www.programiz.com(learn python)

List

Built-in List Methods:

list.append(elem) --

adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.

list.insert(index, elem) --

inserts the element at the given index, shifting elements to the right.

list.extend(list2) --

adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend(). does not return the new list, just modifies the original.

list.index(elem) --

searches for the given element from the start of the list and returns its index.
Returns the position of element, if same element occurs multiple times in list that in such case it returns the 1st occurance of that element.
Throws a ValueError if the element does not appear (use "in" to check without a ValueError).

list.remove(elem) --

searches for the first instance of the given element and removes it (throws ValueError if not present)

list.sort() --

sorts the list in place (does not return it). (The sorted() function shown below is preferred.)

List = [‘123’, ‘abc’, ‘xyz’, ‘zara’, ‘8’]
List.sort()
Print List
>>>>>> [‘123’, ‘8’, ‘abc’, ‘xyz’, ‘zara’’]

list.reverse() -- reverses the list in place (does not return it)

list.pop(index) --

removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).
If index is not specified than its removes and return last element.

list.count(element) --

Returns count of how many times element occurs in list.

Example:- 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

Notice that these are *methods* on a list object, while len() is a function that takes the list (or string or whatever) as an argument.

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.
In fact, lists respond to all of the general sequence operations we used on strings

Python Expression	**Results**	**Description**
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition

3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

```
>>> s+[12]
[1, 2, 3, 4, 5, 6, 12]

>>> s+[13]
[1, 2, 3, 4, 5, 6, 13]
```

Tuple to be converted into list.

```
aTuple = (123, 'xyz', 'zara', 'abc');
aList = list(aTuple)
print "List elements : ", aList

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

Built-in List Functions:

Python includes the following list functions −

cmp(list1, list2)
Compares elements of both lists.

len(list)
Gives the total length of the list.

max(list)
Returns item from the list with max value.

min(list)
Returns item from the list with min value.

list(seq)

Converts a tuple into list.

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).
To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque

>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
>>>print queue
>>>(["Eric", "John", "Michael", "Terry", "Graham"])
>>> queue.popleft()
'Eric'
>>> queue.popleft()
'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])
```

Deque objects support the following methods:

It also support the append(),extend(),pop(),remove()

appendleft(*x*)
Add *x* to the left side of the deque.

clear()
Remove all elements from the deque leaving it with length 0.

extendleft(*iterable*)
Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

rotate(*n*)
Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left. Rotating one step to the right is equivalent to: d.appendleft(d.pop()).

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...    print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
```

```
['g', 'h', 'i']
>>> d[0]                        # peek at leftmost item
'g'
>>> d[-1]                       # peek at rightmost item
'i'

>>> list(reversed(d))              # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                    # search the deque
True
>>> d.extend('jkl')               # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                 # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))             # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                   # empty the deque
>>> d.pop()                     # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')            # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

List Comprehensions

List comprehensions provide a concise way to create lists.

A list comprehension consists

of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the

context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
# create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

# the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]

>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice).

For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del can also be used to delete entire variables:

```
>>>
>>> del a
```

Dictionary :-

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Updating Dictionary:

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
```

Delete Dictionary Elements:

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the del statement. Following is a simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary
```

Properties of Dictionary Keys:

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result –

```
dict['Name']:  Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example:

```
dict = {['Name']: 'Zara', 'Age': 7}
print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result –

```
Traceback (most recent call last):
   File "test.py", line 3, in <module>
      dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

Function:

SN	Function with Description

1	**cmp(dict1, dict2)**
	
	Compares elements of both dict.
	
	The method **cmp()** compares two dictionaries based on key and values.
	
	`cmp(dict1, dict2)`
	
	• **dict1** -- This is the first dictionary to be compared with dict2.
	• **dict2** -- This is the second dictionary to be compared with dict1.
	
	Return Value
	
	This method returns 0 if both dictionaries are equal,
	
	-1 if dict1 < dict2 and
	
	1 if dict1 > dic2.
	
	The following example shows the usage of cmp() method.
	
	`dict1 = {'Name': 'Zara', 'Age': 7};`
	`dict2 = {'Name': 'Mahnaz', 'Age': 27};`
	`dict3 = {'Name': 'Abid', 'Age': 27};`
	`dict4 = {'Name': 'Zara', 'Age': 7};`
	`print "Return Value : %d" % cmp (dict1, dict2)`
	`print "Return Value : %d" % cmp (dict2, dict3)`
	`print "Return Value : %d" % cmp (dict1, dict4)`
	
	When we run above program, it produces following result –
	
	`Return Value : -1`
	`Return Value : 1`
	`Return Value : 0`

2	**len(dict)** **Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.**

	`dict = {'Name': 'Zara', 'Age': 7};`
	`print "Length : %d" % len (dict)`
	
	When we run above program, it produces following result –
	
	`Length : 2(return the no of keys)`

3	**str(dict)** **Produces a printable string representation of a dictionary**
4	**type(variable)** **Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.**

SN	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict*
2	**dict.copy()** Returns a shallow copy of dictionary *dict*
3	**dict.fromkeys()** Create a new dictionary with keys from seq and values *set* to *value*.
4	**dict.get(key, default=None)** For *key* key, returns value or default if key not in dictionary
5	**dict.has_key(key)** Returns *true* if key in dictionary *dict*, *false* otherwise
6	**dict.items()**

	Returns a list of *dict*'s (key, value) tuple pairs
7	**dict.keys()** Returns list of dictionary dict's keys
8	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if *key* is not already in dict
9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict*
10	**dict.values()** Returns list of dictionary *dict*'s values

Tuple :

Tuples are sequences, just like lists.
The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

```
>>>tup1[0]:  physics
```

```
>>>tup2[1:5]:  [2, 3, 4, 5]
```

```
tup = ();    Empty tuple
```

```
tup = (50,);    Tuple with Single value
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;


# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3
```

When the above code is executed, it produces the following result –

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements :

Removing individual tuple elements is not possible
To explicitly remove an entire tuple, just use the **del** statement.
For example:

```
tup = ('physics', 'chemistry', 1997, 2000);

del tup;
print "After deleting tup : "
print tup


>>>After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.
In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

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

Built-in Tuple Functions

Python includes the following tuple functions −
SNFunction with Description 1
cmp(tuple1, tuple2)

Compares elements of both tuples. 2
len(tuple)

Gives the total length of the tuple. 3
max(tuple)

Returns item from the tuple with max value. 4
min(tuple)

Returns item from the tuple with min value. 5
tuple(seq)

Converts a list into tuple.