Python List

A list is a sequence of values (similar to an array in other programming languages but more versatile)

The values in a list are called items or sometimes elements.

The important properties of Python lists are as follows:

- Lists are ordered – Lists remember the order of items inserted.
- Accessed by index – Items in a list can be accessed using an index.
- Lists can contain any sort of object – It can be numbers, strings, tuples and even other lists.
- Lists are changeable (mutable) – You can change a list in-place, add new items, and delete or update existing items.

Create a List

There are several ways to create a new list; the simplest is to enclose the values in square brackets `[]`

```
# A list of integers
L = [1, 2, 3]


# A list of strings
L = ['red', 'green', 'blue']
```

The items of a list don’t have to be the same type. The following list contains an integer, a string, a float, a complex number, and a boolean.

```
# A list of mixed datatypes
L = [ 1, 'abc', 1.23, (3+4j), True]
```

A list containing zero items is called an empty list and you can create one with empty
brackets `[]`

```
# An empty list
L = []
```

There is one more way to create a list based on existing list, called List comprehension.

The list() Constructor

You can convert other data types to lists using Python's list() constructor.

```
# Convert a string to a list

L = list('abc')

print(L)
# Prints ['a', 'b', 'c']

# Convert a tuple to a list

L = list((1, 2, 3))

print(L)
# Prints [1, 2, 3]
```

Nested List

A list can contain sublists, which in turn can contain sublists themselves, and so on. This is known as nested list.

You can use them to arrange data into hierarchical structures.

```
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']
```

Read more about it in the nested list tutorial.

Access List Items by Index

You can think of a list as a relationship between indexes and values. This relationship is called a mapping; each index maps to one of the values. The indexes for the values in a list are illustrated as below:

Note that the first element of a list is always at index zero.

You can access individual items in a list using an index in square brackets.

```
L = ['red', 'green', 'blue', 'yellow', 'black']


print(L[0])

# Prints red


print(L[2])
# Prints blue
```

Python will raise an `IndexError` error, if you use an index that exceeds the number of items in your list.

```
L = ['red', 'green', 'blue', 'yellow', 'black']

print(L[10])
# Triggers IndexError: list index out of range
```

Negative List Indexing

You can access a list by negative indexing as well. Negative indexes count backward from the end of the list. So, `L[-1]` refers to the last item, `L[-2]` is the second-last, and so on.

The negative indexes for the items in a list are illustrated as below:

```
L = ['red', 'green', 'blue', 'yellow', 'black']


print(L[-1])

# Prints black


print(L[-2])
# Prints yellow
```

Access Nested List Items

Similarly, you can access individual items in a nested list using multiple indexes. The first index determines which list to use, and the second indicates the value within that list.

The indexes for the items in a nested list are illustrated as below:

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']


print(L[2][2])

# Prints ['eee', 'fff']


print(L[2][2][0])
# Prints eee
```

Slicing a List

A segment of a list is called a slice and you can extract one by using a slice operator. A slice of a list is also a list.

The slice operator [n:m] returns the part of the list from the “n-th” item to the “m-th” item, including the first but excluding the last.

```
L = ['a', 'b', 'c', 'd', 'e', 'f']


print(L[2:5])

# Prints ['c', 'd', 'e']


print(L[0:2])

# Prints ['a', 'b']


print(L[3:-1])
```

```
# Prints ['d', 'e']
```

Read more about it in the list slicing tutorial.

Change Item Value

You can replace an existing element with a new value by assigning the new value to the index.

```
L = ['red', 'green', 'blue']


L[0] = 'orange'
print(L)
# Prints ['orange', 'green', 'blue']


L[-1] = 'violet'
print(L)
# Prints ['orange', 'green', 'violet']
```

Add items to a list

To add new values to a list, use append() method. This method adds items only to the end of the list.

```
L = ['red', 'green', 'yellow']
L.append('blue')
print(L)
# Prints ['red', 'green', 'yellow', 'blue']
```

If you want to insert an item at a specific position in a list, use insert() method. Note that all of the values in the list after the inserted value will be moved down one index.

```
L = ['red', 'green', 'yellow']
L.insert(1,'blue')
print(L)
```

```
# Prints ['red', 'blue', 'green', 'yellow']
```

Combine Lists

You can merge one list into another by using extend() method. It takes a list as an argument and appends all of the elements.

```
L = ['red', 'green', 'yellow']

L.extend([1,2,3])

print(L)
# Prints ['red', 'green', 'yellow', 1, 2, 3]
```

Alternatively, you can use the concatenation operator `+` or the augmented assignment operator `+=`

```
# concatenation operator

L = ['red', 'green', 'blue']

L = L + [1,2,3]

print(L)

# Prints ['red', 'green', 'blue', 1, 2, 3]


# augmented assignment operator

L = ['red', 'green', 'blue']

L += [1,2,3]

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]
```

Remove items from a list

There are several ways to remove items from a list.

Remove an Item by Index

If you know the index of the item you want, you can use pop() method. It modifies the list and returns the removed item.

If no index is specified, `pop()` removes and returns the last item in the list.

```
L = ['red', 'green', 'blue']

x = L.pop(1)

print(L)

# Prints ['red', 'blue']


# removed item

print(x)
# Prints green
```

If you don’t need the removed value, use the del statement.

```
L = ['red', 'green', 'blue']

del L[1]

print(L)
# Prints ['red', 'blue']
```

Remove an Item by Value

If you’re not sure where the item is in the list, use remove() method to delete it by value.

```
L = ['red', 'green', 'blue']

L.remove('red')

print(L)
# Prints ['green', 'blue']
```

But keep in mind that if more than one instance of the given item is present in the list, then this method removes only the first instance.

```
L = ['red', 'green', 'blue', 'red']

L.remove('red')

print(L)
# Prints ['green', 'blue', 'red']
```

Remove Multiple Items

To remove more than one items, use the del keyword with a slice index.

```
L = ['red', 'green', 'blue', 'yellow', 'black']

del L[1:4]

print(L)
# Prints ['red', 'black']
```

Remove all Items

Use clear() method to remove all items from the list.

```
L = ['red', 'green', 'blue']

L.clear()

print(L)
# Prints []
```

List Replication

The replication operator `*` repeats a list a given number of times.

```
L = ['red']

L = L * 3

print(L)
# Prints ['red', 'red', 'red']
```

Find List Length

To find the number of items in a list, use len() method.

```
L = ['red', 'green', 'blue']

print(len(L))
# Prints 3
```

Check if item exists in a list

To determine whether a value is or isn’t in a list, you can use in and not in operators with if statement.

```
# Check for presence
L = ['red', 'green', 'blue']
if 'red' in L:
    print('yes')


# Check for absence
L = ['red', 'green', 'blue']
if 'yellow' not in L:
    print('yes')
```

Iterate through a List

The most common way to iterate through a list is with a for loop.

```
L = ['red', 'green', 'blue']
for item in L:
    print(item)
# Prints red
# Prints green
# Prints blue
```

This works well if you only need to read the items of the list. But if you want to update them, you need the indexes. A common way to do that is to combine the `range()` and `len()` functions.

```
# Loop through the list and double each item
L = [1, 2, 3, 4]
for i in range(len(L)):
    L[i] = L[i] * 2


print(L)
```

```
# Prints [2, 4, 6, 8]
```

Python List Methods

Python has a set of built-in methods that you can call on list objects.

Method	Description
append()	Adds an item to the end of the list
insert()	Inserts an item at a given position
extend()	Extends the list by appending all the items from the iterable
remove()	Removes first instance of the specified item
pop()	Removes the item at the given position in the list
clear()	Removes all items from the list
copy()	Returns a shallow copy of the list
count()	Returns the count of specified item in the list
index()	Returns the index of first instance of the specified item
reverse()	Reverses the items of the list in place
sort()	Sorts the items of the list in place

Above function in Detail:-

Python List append() Method

Appends an item to a list

Usage

The append() method adds a single item to the end of the list. This method does not return anything; it modifies the list in place.

Syntax

list.append(item)

Parameter	Condition	Description
Item	Required	An item you want to append to the list

Examples

```
# Append 'yellow'
L = ['red', 'green', 'blue']
L.append('yellow')
print(L)
# Prints ['red', 'green', 'blue', 'yellow']
# Append list to a list
L = ['red', 'green', 'blue']
L.append([1,2,3])
print(L)
# Prints ['red', 'green', 'blue', [1, 2, 3]]
# Append tuple to a list
L = ['red', 'green', 'blue']
L.append((1,2,3))
```

```
print(L)
# Prints ['red', 'green', 'blue', (1, 2, 3)]
```

append() vs extend()

append() method treats its argument as a single object.

```
L = ['red', 'green']

L.append('blue')

print(L)
# Prints ['red', 'green', 'blue']
```

Use extend() method, if you want to add every item of an iterable to a list.

```
L = ['red', 'green']

L.extend('blue')

print(L)
# Prints ['red', 'green', 'b', 'l', 'u', 'e']
```

Python List insert() Method

Inserts an item into a list at specified position

Usage

Use `insert()` method to insert a single item at a specified index in a list. Note that other items are shifted to the right.

This method does not return anything; it modifies the list in place.

Syntax

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

Parameter	Condition	Description
Index	Required	Index of an item before which to insert
Item	Required	An item you want to insert

Examples

```
# Insert 'yellow' at 2nd position
L = ['red', 'green', 'blue']
L.insert(1,'yellow')
print(L)
# Prints ['red', 'yellow', 'green', 'blue']
```

You can also use negative indexing with `insert()` method.

```
# Insert 'yellow' at 2nd position
L = ['red', 'green', 'blue']
L.insert(-2,'yellow')
print(L)
# Prints ['red', 'yellow', 'green', 'blue']
```

Index greater than list length

When you specify an index greater than list length, you do not get any exception. Instead, the item is inserted at the end of the list.

```
L = ['red', 'green', 'blue']

L.insert(10,'yellow')

print(L)
# Prints ['red', 'green', 'blue', 'yellow']
```

insert() vs append()

Inserting item at the end of the list with insert() method is equivalent to append() method.

```
L = ['red', 'green', 'blue']

L.insert(len(L),'yellow')

print(L)

# Prints ['red', 'green', 'blue', 'yellow']


# is equivalent to

L = ['red', 'green', 'blue']

L.append('yellow')

print(L)
# Prints ['red', 'green', 'blue', 'yellow']
```

insert() vs extend()

insert() method treats its argument as a single object.

```
L = ['red', 'green']

L.insert(2,'blue')

print(L)
# Prints ['red', 'green', 'blue']
```

Use extend() method, if you want to add every item of an iterable to a list.

```
L = ['red', 'green']

L.extend('blue')

print(L)
# Prints ['red', 'green', 'b', 'l', 'u', 'e']
```

Python List extend() Method

Extends a list with the items from an iterable

Usage

The `extend()` method extends the list by appending all the items from the iterable to the end of the list. This method does not return anything; it modifies the list in place.

Syntax

```
list.extend(iterable)
```

Parameter	Condition	Description
iterable	Required	Any iterable (string, list, set, tuple, etc.)

Examples

```
# Add multiple items to a list
```

```
L = ['red', 'green', 'blue']

L.extend([1,2,3])

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]

# Add tuple items to a list

L = ['red', 'green', 'blue']

L.extend((1,2,3))

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]

# Add set items to a list

L = ['red', 'green', 'blue']

L.extend({1,2,3})

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]
```

extend() vs append()

`extend()` method treats its argument as an iterable object.

For example, when you pass a string (iterable object) as an argument, the method adds every character to a list instead of the string.

```
L = ['red', 'green', 'blue']

L.extend('red')

print(L)
# Prints ['red', 'green', 'blue', 'r', 'e', 'd']
```

Use append() method instead:

```
L = ['red', 'green', 'blue']

L.append('red')

print(L)
# Prints ['red', 'green', 'blue', 'red']
```

Equivalent Methods

Specifying a zero-length slice at the end is also equivalent to `extend()` method.

```
L = ['red', 'green', 'blue']

L[len(L):] = [1,2,3]

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]
```

Using concatenation operator `+` or the augmented assignment operator `+=` on a list is equivalent to using `extend()`.

```
# Concatenation operator

L = ['red', 'green', 'blue']

L = L + [1,2,3]

print(L)

# Prints ['red', 'green', 'blue', 1, 2, 3]


# Augmented assignment operator

L = ['red', 'green', 'blue']

L += [1,2,3]

print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]
```

Python List remove() Method

Removes an item from a list

Usage

Use `remove()` method to remove a single item from a list.

The method searches for the first instance of the given item and removes it. If specified item is not found, it raises ‘ValueError’ exception.

Syntax

```
list.remove(item)
```

Parameter	Condition	Description
item	Required	Any item you want to remove

Remove Single Item

```
# Remove 'green'
L = ['red', 'green', 'blue']
L.remove('green')
print(L)
# Prints ['red', 'blue']

# Remove item from the nested list
L = ['red', 'green', [1, 2, 3]]
L.remove([1, 2, 3])
print(L)
# Prints ['red', 'green']
```

The `remove()` method removes item based on specified value and not by index. If you want to delete list items based on the index, use pop() method or del keyword.

Remove Duplicate Items

`remove()` method searches for the first instance of the given item and removes it.

```
L = ['red', 'green', 'blue', 'red', 'red']

L.remove('red')

print(L)
# Prints ['green', 'blue', 'red', 'red']
```

If you want to remove multiple instances of an item in a list, use list comprehension or lambda expression.

```
# list comprehension

L = ['red', 'green', 'blue', 'red', 'red']

L = [x for x in L if x is not 'red']

print(L)

# Prints ['green', 'blue']


# lambda expression

L = ['red', 'green', 'blue', 'red', 'red']

L = list(filter(lambda x: x is not 'red', L))

print(L)
# Prints ['green', 'blue']
```

Removing Item that Doesn’t Exist

`remove()` method raises an ValueError exception, if specified item doesn’t exist in a list.

```
L = ['red', 'green', 'blue']

L.remove('yellow')
# Triggers ValueError: list.remove(x): x not in list
```

To avoid such exception, you can check if item exists in a list, using in operator inside if statement.

```
L = ['red', 'green', 'blue']

if 'yellow' in L:
    L.remove('yellow')
```

Python List pop() Method

Removes an item at specified index

Usage

The `pop()` method removes a single list item at specified index and returns it. If no index is specified, `pop()` method removes and returns the last item in the list.

Syntax

```
list.pop(index)
```

Parameter	Condition	Description
index	Optional	An index of item you want to remove. Default value is -1

Return Value

The `pop()` method returns the value of removed item.

Examples

```
# Remove 2nd list item
L = ['red', 'green', 'blue']
L.pop(1)
print(L)
# Prints ['red', 'blue']
```

You can also use negative indexing with `pop()` method.

```
# Remove 2nd list item
L = ['red', 'green', 'blue']
L.pop(-2)
print(L)
# Prints ['red', 'blue']
```

When you remove an item from the list using `pop()`, it removes it and returns its value.

```
L = ['red', 'green', 'blue']
x = L.pop(1)


# removed item
print(x)
# Prints green
```

When you don’t specify the index on `pop()`, it assumes the parameter to be -1 and removes the last item.

```
L = ['red', 'green', 'blue']
L.pop()
print(L)
# Prints ['red', 'green']
```

Python List clear() Method

Removes all items from the list

Usage

Use `clear()` method to remove all items from the list. This method does not return anything; it modifies the list in place.

Syntax

```
list.clear()
```

Basic Example

```
L = ['red', 'green', 'blue']
L.clear()
print(L)
# Prints []
```

Please note that `clear()` is not same as assigning an empty list `L = []`.

`L = []` does not empty the list in place, it overwrites the variable L with a different list which happens to be empty.

If anyone else had a reference to the original list, that remains as is; which may create a problem.

Equivalent Methods

For Python 2.x or below Python 3.2 users, `clear()` method is not available. You can use below equivalent methods.

1. You can remove all items from the list by using del keyword on a start-to-end slice.

```
L = ['red', 'green', 'blue']
del L[:]
```

```
print(L)
# Prints []
```

5. Assigning empty list to a start-to-end slice will have same effect as `clear()`.

```
L = ['red', 'green', 'blue']
L[:] = []
print(L)
# Prints []
```

9. Multiplying 0 to a list using multiplication assignment operator will remove all items from the list in place.

```
L = ['red', 'green', 'blue']
L *= 0
print(L)
# Prints []
```

Any given integer that is less than or equal to 0 would have the same effect.

Python List copy() Method

Copies the list shallowly

Usage

The `copy()` method returns the Shallow copy of the specified list.

Syntax

```
list.copy()
```

Basic Example

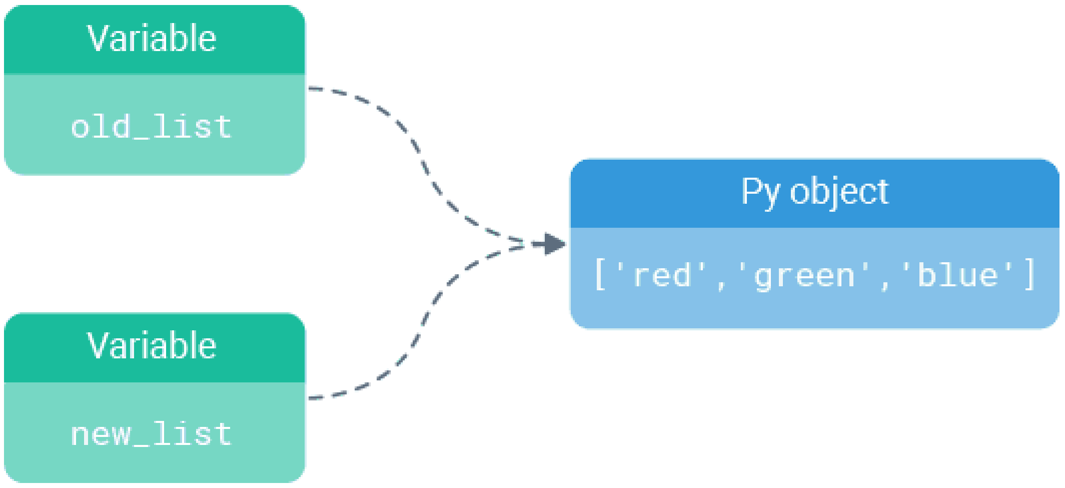
```
# Create a copy of list 'L'
L = ['red', 'green', 'blue']
X = L.copy()
print(X)
# Prints ['red', 'green', 'blue']
```

copy() vs Assignment statement

Assignment statement does not copy objects. For example,

```
old_List = ['red', 'green', 'blue']
new_List = old_List
new_List[0] = 'xx'
print(old_List)
# Prints ['xx', 'green', 'blue']
print(new_List)
# Prints ['xx', 'green', 'blue']
```

When you execute `new_List = old_List`, you don’t actually have two lists. The assignment just makes the two variables point to the one list in memory.



So, when you change new_List, old_List is also modified. If you want to change one copy without changing the other, use `copy()`method.

```
old_List = ['red', 'green', 'blue']
new_List = old_List.copy()
new_List[0] = 'xx'
print(old_List)
# Prints ['red', 'green', 'blue']
print(new_List)
# Prints ['xx', 'green', 'blue']
```

Equivalent Method

Assigning a slice of the entire list to a variable is equivalent to `copy()` method.

```
L = ['red', 'green', 'blue']
X = L[:]
print(X)
# Prints ['red', 'green', 'blue']
```

Python List count() Method

Counts the number of occurrences of an item

Usage

Use `count()` method to find the number of times the specified item appears in the list.

Syntax

```
list.count(item)
```

Parameter	Condition	Description
item	Required	Any item (of type string, list, set, etc.) you want to search for.

Examples

```
# Count number of occurrences of 'red'
L = ['red', 'green', 'blue']
print(L.count('red'))
# Prints 1
# Count number of occurrences of number '9'
L = [1, 9, 7, 3, 9, 1, 9, 2]
print(L.count(9))
# Prints 3
```

Count Multiple Items

If you want to count multiple items in a list, you can call `count()` in a loop.

This approach, however, requires a separate pass over the list for every `count()` call; which can be catastrophic for performance. Use couter() method from class `collections`, instead.

```
# Count occurrences of all the unique items
L = ['a', 'b', 'c', 'b', 'a', 'a', 'a']
from collections import Counter
print(Counter(L))
# Prints Counter({'a': 4, 'b': 2, 'c': 1})
```

Python List index() Method

Searches the list for a given item

Usage

The `index()` method searches for the first occurrence of the given item and returns its index. If specified item is not found, it raises ‘ValueError’ exception.

The optional arguments start and end limit the search to a particular subsequence of the list.

Syntax

```
list.index(item,start,end)
```

Parameter	Condition	Description
item	Required	Any item (of type string, list, set, etc.) you want to search for
start	Optional	An index specifying where to start the search. Default is 0.

end	Optional	An index specifying where to stop the search. Default is the end of the list.

Basic Example

```
# Find the index of 'green' in a list
L = ['red', 'green', 'blue', 'yellow']
print(L.index('green'))
# Prints 1
```

index() on Duplicate Items

If the list has many instances of the specified item, the index() method returns the index of first instance only.

```
# Find first occurrence of character ‘c’
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
print(L.index('c'))
# Prints 2
```

Limit index() Search to Subsequence

If you want to search the list from the middle, specify the start parameter.

```
# Find 'c' starting a position 5
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
print(L.index('c',5))
# Prints 8
```

The returned index is computed relative to the beginning of the full sequence rather than the start argument.

You can also specify where to stop the search with end parameter.

```
# Find 'c' in between 5 & 10
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
print(L.index('c',5,10))
# Prints 8
```

index() on Item that Doesn't Exist

index() method raises a 'ValueError' if specified item is not found in the list.

```
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
print(L.index('x'))
# Triggers ValueError: 'x' is not in list


# also within search bound
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
print(L.index('c',4,7))
# Triggers ValueError: 'c' is not in list
```

To avoid such exception, you can check if item exists in a list, using in operator inside if statement.

```
L = ['a','b','c','d','e','f','a','b','c','d','e','f']
if 'x' in L:
    print(L.index('x'))
```

Python List reverse() Method

Reverses the order of the list

Usage

The reverse() method reverses the order of list. This method does not return anything; it modifies the list in place.

Syntax

```
list.reverse()
```

Examples

```
L = ['red', 'green', 'blue']

L.reverse()

print(L)
# Prints ['blue', 'green', 'red']

L = [1, 2, 3, 4, 5]

L.reverse()

print(L)
# Prints [5, 4, 3, 2, 1]
```

Access List Items in Reversed Order

If you don’t want to modify the list but access items in reverse order, you can use reversed() built-in function. It returns the reversed iterator object, with which you can loop through the list in reverse order.

```
L = ['red', 'green', 'blue']

for x in reversed(L):

  print(x)

# blue

# green
# red
```

Python List sort() Method

Sorts the items of the list

Usage

Use `sort()` method to sort the items of the list.

You can optionally specify parameters for sort customization like sorting order and sorting criteria.

Syntax

```
list.sort(key,reverse)
```

Parameter	Condition	Description
key	Optional	A function to specify the sorting criteria. Default value is None.
reverse	Optional	Settting it to True sorts the list in reverse order. Default value is False.

Please note that both the arguments must be specified as keyword arguments.

Sort List

`sort()` method sorts the list of strings alphabetically and the list of numbers numerically.

```
# Sort the list of strings

L = ['red', 'green', 'blue', 'orange']

L.sort()

print(L)
# Prints ['blue', 'green', 'orange', 'red']

# Sort the list of numbers

L = [42, 99, 1, 12]

L.sort()

print(L)
# Prints [1, 12, 42, 99]
```

However, you cannot sort lists that have both numbers and strings in them, since Python doesn’t know how to compare these values.

```
L = ['red', 'blue', 1, 12, 'orange',42, 'green', 99]

L.sort()
# Triggers TypeError: '<' not supported between instances of 'int' and 'str'
```

Sort List in Reverse Order

You can also sort the list in reverse order by setting `reverse` to TRUE.

```
L = ['red', 'green', 'blue', 'orange']

L.sort(reverse=True)

print(L)
# Prints ['red', 'orange', 'green', 'blue']
```

Sort with Key

Use `key` parameter for more complex custom sorting. A `key` parameter specifies a function to be executed on each list item before making comparisons.

For example, with a list of strings, specifying `key=len` (the built-in len() function) sorts the strings by length, from shortest to longest.

```
L = ['red', 'green', 'blue', 'orange']
L.sort(key=len)
print(L)
# Prints ['red', 'blue', 'green', 'orange']
```

A function to be used as key must take a single value and return single value.

Sort with Custom Function

You can also pass in your own custom function `myFunc` as the `key` function.

```
# Sort a list of tuples based on the age of students
def myFunc(e):
  return e[1]           # return age

L = [('Bob', 30),
     ('Sam', 35),
     ('Max', 25)]
L.sort(key=myFunc)
print(L)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
# Sort a list of dictionaries based on the age of students
def myFunc(e):
    return e['age']                # return age

L = [{'name': 'Bob', 'age': 30},
     {'name': 'Sam', 'age': 35},
     {'name': 'Max', 'age': 25}]
L.sort(key=myFunc)
print(L)
# [{'age': 25, 'name': 'Max'}, {'age': 30, 'name': 'Bob'}, {'age': 35, 'name': 'Sam'}]
```

Case-insensitive Sorting

By default, the `sort()` method sorts the list in ASCIIbetical order rather than actual alphabetical order. This means uppercase letters come before lowercase letters.

```
# Case-sensitive sorting

L = ['Red', 'blue', 'Green', 'orange']

L.sort()

print(L)
# Prints ['Green', 'Red', 'blue', 'orange']
```

If you want to sort the values in regular alphabetical order, set `key` to str.lower

```
# Case-insensitive sorting

L = ['Red', 'blue', 'Green', 'orange']

L.sort(key=str.lower)

print(L)
# Prints ['blue', 'Green', 'orange', 'Red']
```

This causes the `sort()` function to treat all the list items as if they were lowercase without actually changing the values in the list.

sort() vs sorted()

The `sort()` method doesn't return anything, it modifies the original list (i.e. sorts in-place). If you don't want to modify the original list, use sorted() function. It returns a sorted copy of the list.

```
# Get a sorted copy of the list with sorted()

L = ['red', 'green', 'blue', 'orange']

x = sorted(L)

print(x)
# Prints ['blue', 'green', 'orange', 'red']

# Iterate through a sorted list without changing the original

L = ['red', 'green', 'blue', 'orange']

for x in sorted(L):
    print(x)
# Prints blue green orange red
```

Another difference is that the `sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable like tuple, dictionary etc. Also, the `sort()` method doesn't need to create a new list, so it's faster between the two.