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

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.

```
2. L = ['red', 'green', 'blue']
3. 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

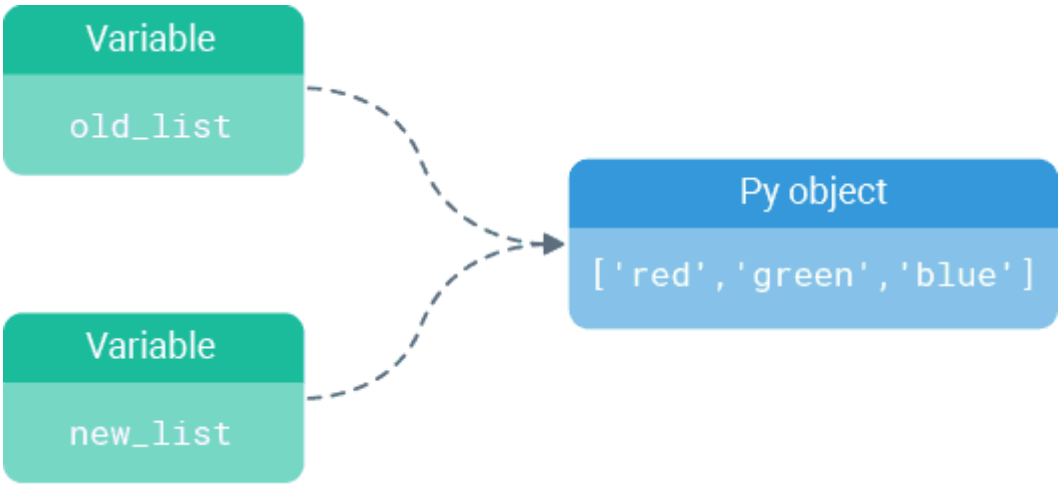
```
# Create a copy of list 'L'
L = ['red', 'green', 'blue']
X = L.copy()
print(X)
# Prints ['red', 'green', 'blue']
```

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.

Built-in Functions with List

Python also has a set of built-in functions that you can use with list objects.

Method	Description
all()	Returns True if all list items are true
any()	Returns True if any list item is true
enumerate()	Takes a list and returns an enumerate object
len()	Returns the number of items in the list
list()	Converts an iterable (tuple, string, set etc.) to a list
max()	Returns the largest item of the list
min()	Returns the smallest item of the list
sorted()	Returns a sorted list
sum()	Sums items of the list

Detail of above function

Python all() Function

Determines whether all items in an iterable are True

Usage

The `all()` function returns True if all items in an iterable are True. Otherwise, it returns False.

If the iterable is empty, the function returns True.

Syntax

```
all(iterable)
```

Parameter	Condition	Description
iterable	Required	An iterable of type (list, string, tuple, set, dictionary etc.)

Falsy Values

In Python, all the following values are considered False.

- Constants defined to be false: `None` and `False`.
- Zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- Empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Basic Examples

```
# Check if all items in a list are True

L = [1, 1, 1]
print(all(L))   # Prints True

L = [0, 1, 1]
print(all(L))   # Prints False
```

Here are some scenarios where `all()` returns False.

```
L = [True, 0, 1]
print(all(L))   # Prints False


T = ('', 'red', 'green')
print(all(T))   # Prints False


S = {0j, 3+4j}
print(all(S))   # Prints False
```

all() on a Dictionary

When you use `all()` function on a dictionary, it checks if all the keys are true, not the values.

```
D1 = {0: 'Zero', 1: 'One', 2: 'Two'}
print(all(D1))   # Prints False


D2 = {'Zero': 0, 'One': 1, 'Two': 2}
print(all(D2))   # Prints True
```

all() on Empty Iterable

If the iterable is empty, the function returns True.

```
# empty iterable
L = []
print(all(L))   # Prints True


# iterable with empty items
L = [[], []]
print(all(L))   # Prints False
```

Python any() Function

Determines whether any item in an iterable is True

Usage

The `any()` function returns True if any item in an `iterable` is True. Otherwise, it returns False.

If the iterable is empty, the function returns False.

Syntax

```
any(iterable)
```

Parameter	Condition	Description
iterable	Required	An iterable of type (list, string, tuple, set, dictionary etc.)

Falsy Values

In Python, all the following values are considered False.

- Constants defined to be false: `None` and `False`.
- Zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`

- Empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Basic Examples

```
# Check if any item in a list is True


L = [0, 0, 0]
print(any(L))   # Prints False


L = [0, 1, 0]
print(any(L))   # Prints True
```

Here are some scenarios where `any()` returns True.

```
L = [False, 0, 1]
print(any(L))   # Prints True


T = ('', [], 'green')
print(any(T))   # Prints True


S = {0j, 3+4j, 0.0}
print(any(S))   # Prints True
```

any() on a Dictionary

When you use `any()` function on a dictionary, it checks if any of the keys is true, not the values.

```
D1 = {0: 'Zero', 0: 'Nil'}
print(any(D1))   # Prints False


D2 = {'Zero': 0, 'Nil': 0}
print(any(D2))   # Prints True
```

any() on Empty Iterable

If the iterable is empty, the function returns False.

```
L = []
print(any(L))   # Prints False
```

Python enumerate() Function

Adds a counter to an iterable

Usage

The enumerate() function adds a counter to an iterable and returns it as an enumerate object.

By default, enumerate() starts counting at 0 but if you add a second argument start, it'll start from that number instead.

Syntax

enumerate(iterable,start)

Parameter	Condition	Description
iterable	Required	An iterable (e.g. list, tuple, string etc.)

start	Optional	A number to start counting from. Default is 0.

Basic Example

```
# Create a list that can be enumerated
L = ['red', 'green', 'blue']
x = list(enumerate(L))
print(x)
# Prints [(0, 'red'), (1, 'green'), (2, 'blue')]
```

Specify Different Start

By default, `enumerate()` starts counting at 0 but if you add a second argument `start`, it'll start from that number instead.

```
# Start counter from 10
L = ['red', 'green', 'blue']
x = list(enumerate(L, 10))
print(x)
# Prints [(10, 'red'), (11, 'green'), (12, 'blue')]
```

Iterate Enumerate Object

When you iterate an enumerate object, you get a tuple containing (counter, item)

```
L = ['red', 'green', 'blue']
for pair in enumerate(L):
    print(pair)
# Prints (0, 'red')
# Prints (1, 'green')
```

```
# Prints (2, 'blue')
```

You can unpack the tuple into multiple variables as well.

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

for index, item in enumerate(L):

    print(index, item)

# Prints 0 red

# Prints 1 green
# Prints 2 blue
```

Python len() Function

Returns the number of items of an object

Usage

The `len()` function returns the number of items of an object.

The object may be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

Syntax

```
len(object)
```

Parameter	Condition	Description

object	Required	A sequence or a collection.

len() on Sequences

```
# number of characters in a string
S = 'Python'
x = len(S)
print(x)
# Prints 6

# number of items in a list
L = ['red', 'green', 'blue']
x = len(L)
print(x)
# Prints 3

# number of items in a tuple
T = ('red', 'green', 'blue')
x = len(T)
print(x)
# Prints 3
```

len() on Collections

```
# number of key:value pairs in a dictionary
D = {'name': 'Bob', 'age': 25}
x = len(D)
print(x)
# Prints 2

# number of items in a set
S = {'red', 'green', 'blue'}
x = len(S)
```

```
print(x)
# Prints 3
```

Python list() Function

Creates a list from an iterable

Usage

The `list()` function creates a list from an iterable.

The iterable may be a sequence (such as a string, tuple or range) or a collection (such as a dictionary, set or frozen set)

There is another way you can create lists based on existing lists. It is called List comprehension.

Syntax

```
list(iterable)
```

Parameter	Condition	Description
iterable	Required	A sequence or a collection

Examples

`list()` with no arguments creates an empty list.

```
L = list()

print(L)
# Prints []
```

You can convert any sequence (such as a string, tuple or range) into a list using a `list()` method.

```
# string into list

T = list('abc')

print(T)

# Prints ['a', 'b', 'c']


# tuple into list

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

print(L)

# Prints [1, 2, 3]


# sequence into list

L = list(range(0, 4))

print(L)
# Prints [0, 1, 2, 3]
```

You can even convert any collection (such as a dictionary, set or frozen set) into a list.

```
# dictionary keys into list

L = list({'name': 'Bob', 'age': 25})

print(L)

# Prints ['age', 'name']


# set into list
```

```
L = list({1, 2, 3})

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

Python max() Function

Returns the largest item

Usage

The `max()` function can find

- the largest of two or more values (such as numbers, strings etc.)
- the largest item in an iterable (such as list, tuple etc.)

With optional `key` parameter, you can specify custom comparison criteria to find maximum value.

Syntax

```
max(val1,val2,val3… ,key)
```

Parameter	Condition	Description
val1,val2,val3…	Required	Two or more values to compare

key	Optional	A function to specify the comparison criteria. Default value is None.

– OR –

```
max(iterable,key,default)
```

Parameter	Condition	Description
iterable	Required	Any iterable, with one or more items to compare
key	Optional	A function to specify the comparison criteria. Default value is None.
default	Optional	A value to return if the iterable is empty. Default value is False.

Find Maximum of Two or More Values

If you specify two or more values, the largest value is returned.

```
x = max(10, 20, 30)

print(x)
# Prints 30
```

If the values are strings, the string with the highest value in alphabetical order is returned.

```
x = max('red', 'green', 'blue')

print(x)
# Prints red
```

You have to specify minimum two values to compare. Otherwise, TypeError exception is raised.

Find Maximum in an Iterable

If you specify an Iterable (such as list, tuple, set etc.), the largest item in that iterable is returned.

```
L = [300, 500, 100, 400, 200]

x = max(L)

print(x)
# Prints 500
```

If the iterable is empty, a ValueError is raised.

```
L = []

x = max(L)

print(x)
# Triggers ValueError: max() arg is an empty sequence
```

To avoid such exception, add `default` parameter. The `default` parameter specifies a value to return if the provided iterable is empty.

```
# Specify default value '0'

L = []

x = max(L, default='0')

print(x)
```

```
# Prints 0
```

Find Maximum with Built-in Function

With optional `key` parameter, you can specify custom comparison criteria to find maximum value. A `key` parameter specifies a function to be executed on each iterable's item before making comparisons.

For example, with a list of strings, specifying `key=len` (the built-in len() function) finds longest string.

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

x = max(L, key=len)

print(x)
# Prints orange
```

Find Maximum with Custom Function

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

```
# Find out who is the oldest student

def myFunc(e):
  return e[1]           # return age


L = [('Sam', 35),
   ('Tom', 25),
   ('Bob', 30)]


x = max(L, key=myFunc)

print(x)
# Prints ('Sam', 35)
```

A key function takes a single argument and returns a key to use for comparison.

Find Maximum with lambda

A key function may also be created with the lambda expression. It allows us to in-line function definition.

```
# Find out who is the oldest student
L = [('Sam', 35),
   ('Tom', 25),
   ('Bob', 30)]


x = max(L, key=lambda student: student[1])
print(x)
# Prints ('Sam', 35)
```

Find Maximum of Custom Objects

Let's create a list of students (custom object) and find out who is the oldest student.

```
# Custom class
class Student:
        def __init__(self, name, age):
                self.name = name
                self.age = age
        def __repr__(self):
                return repr((self.name, self.age))


# a list of custom objects
L = [Student('Sam', 35),
        Student('Tom', 25),
        Student('Bob', 30)]
```

```
x = max(L, key=lambda student: student.age)


print(x)
# Prints ('Sam', 35)
```

Python min() Function

Returns the smallest item

Usage

The `min()` function can find

- the smallest of two or more values (such as numbers, strings etc.)
- the smallest item in an iterable (such as list, tuple etc.)

With optional `key` parameter, you can specify custom comparison criteria to find minimum value.

Syntax

min(val1,val2,val3… ,key)

Parameter	Condition	Description

val1,val2,val3…	Required	Two or more values to compare
key	Optional	A function to specify the comparison criteria. Default value is None.

– OR –

```
min(iterable,key,default)
```

Parameter	Condition	Description
iterable	Required	Any iterable, with one or more items to compare
key	Optional	A function to specify the comparison criteria. Default value is None.
default	Optional	A value to return if the iterable is empty. Default value is False.

Find Minimum of Two or More Values

If you specify two or more values, the smallest value is returned.

```
x = min(10, 20, 30)

print(x)
# Prints 10
```

If the values are strings, the string with the lowest value in alphabetical order is returned.

```
x = min('red', 'green', 'blue')

print(x)
# Prints blue
```

You have to specify minimum two values to compare. Otherwise, TypeError exception is raised.

Find Minimum in an Iterable

If you specify an Iterable (such as list, tuple, set etc.), the smallest item in that iterable is returned.

```
L = [300, 500, 100, 400, 200]

x = min(L)

print(x)
# Prints 100
```

If the iterable is empty, a ValueError is raised.

```
L = []

x = min(L)

print(x)
# Triggers ValueError: min() arg is an empty sequence
```

To avoid such exception, add `default` parameter. The `default` parameter specifies a value to return if the provided iterable is empty.

```
# Specify default value '0'
L = []
x = min(L, default='0')
print(x)
# Prints 0
```

Find Minimum with Built-in Function

With optional `key` parameter, you can specify custom comparison criteria to find minimum value. A `key` parameter specifies a function to be executed on each iterable's item before making comparisons.

For example, with a list of strings, specifying `key=len` (the built-in len() function) finds shortest string.

```
L = ['red', 'green', 'blue']
x = min(L, key=len)
print(x)
# Prints red
```

Find Minimum with Custom Function

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

```
# Find out who is the youngest student
def myFunc(e):
  return e[1]           # return age

L = [('Sam', 35),
   ('Tom', 25),
   ('Bob', 30)]

x = min(L, key=myFunc)
print(x)
```

```
# Prints ('Tom', 25)
```

A key function takes a single argument and returns a key to use for comparison.

Find Minimum with lambda

A key function may also be created with the lambda expression. It allows us to in-line function definition.

```
# Find out who is the youngest student
L = [('Sam', 35),
   ('Tom', 25),
   ('Bob', 30)]


x = min(L, key=lambda student: student[1])
print(x)
# Prints ('Tom', 25)
```

Find Minimum of Custom Objects

Let's create a list of students (custom object) and find out who is the youngest student.

```
# Custom class
class Student:
        def __init__(self, name, age):
                self.name = name
                self.age = age
        def __repr__(self):
                return repr((self.name, self.age))


# a list of custom objects
L = [Student('Sam', 35),
```

```
            Student('Tom', 25),
            Student('Bob', 30)]


x = min(L, key=lambda student: student.age)


print(x)
# Prints ('Tom', 25)
```

Python sorted() Function

Sorts the items of an iterable

Usage

The `sorted()` method sorts the items of any iterable

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

Syntax

```
sorted(iterable,key,reverse)
```

The method has two optional arguments, which must be specified as keyword arguments.

Parameter	Condition	Description
iterable	Required	Any iterable (list, tuple, dictionary, set etc.) to sort.
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.

Return Value

The method returns a new sorted list from the items in iterable.

Sort Iterables

`sorted()` function accepts any iterable like list, tuple, dictionary, set, string etc.

```
# strings are sorted alphabetically
L = ['red', 'green', 'blue', 'orange']
x = sorted(L)
print(x)
# Prints ['blue', 'green', 'orange', 'red']
```

```
# numbers are sorted numerically
L = [42, 99, 1, 12]
x = sorted(L)
print(x)
# Prints [1, 12, 42, 99]
```

If you want to sort the list in-place, use built-in sort() method.

sort() is actually faster than sorted() as it doesn't need to create a new list.

```
# Sort a tuple
L = ('cc', 'aa', 'dd', 'bb')
x = sorted(L)
print(x)
# Prints ['aa', 'bb', 'cc', 'dd']
```

sorted() function sorts a dictionary by keys, by default.

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}
x = sorted(D)
print(x)
# Prints ['Bob', 'Max', 'Sam', 'Tom']
```

To sort a dictionary by values use the sorted() function along with the values() method.

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}
x = sorted(D.values())
print(x)
# Prints [20, 25, 30, 35]
```

Sort in Reverse Order

You can also sort an iterable in reverse order by setting reverse to true.

```
L = ['cc', 'aa', 'dd', 'bb']
x = sorted(L, reverse=True)
```

```
print(x)
# Prints ['dd', 'cc', 'bb', 'aa']
```

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

x = sorted(L, key=len)

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

Sort with Custom Function

You can also pass in your own custom function as the `key` function. A key function should take a single argument and return a key to use for sorting.

```
# Sort by the age of students

def myFunc(e):
  return e[1]           # return age


L = [('Sam', 35),
   ('Max', 25),
   ('Bob', 30)]

x = sorted(L, key=myFunc)

print(x)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```

Sort with lambda

A key function may also be created with the lambda expression. It allows us to in-line function definition.

```
# Sort by the age of students
L = [('Sam', 35),
   ('Max', 25),
   ('Bob', 30)]
x = sorted(L, key=lambda student: student[1])
print(x)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```

Sort with Operator Module Functions

To access items of an iterable, Python provides convenience functions like itemgetter() and attrgetter() from operator module.

```
# Sort by the age of students
from operator import itemgetter
L = [('Sam', 35),
   ('Max', 25),
   ('Bob', 30)]
x = sorted(L, key=itemgetter(1))
print(x)
# Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```

Multiple Level Sorting

The operator module functions allow multiple levels of sorting as well.

```
# Sort by grade then by age
from operator import itemgetter
L = [('Bob', 'B', 30),
            ('Sam', 'A', 35),
```

```
    ('Max', 'B', 25),
    ('Tom', 'A', 20),
    ('Ron', 'A', 40),
    ('Ben', 'B', 15)]
x = sorted(L, key=itemgetter(1,2))
print(x)
# Prints [('Tom', 'A', 20),
#         ('Sam', 'A', 35),
#         ('Ron', 'A', 40),
#         ('Ben', 'B', 15),
#         ('Max', 'B', 25),
#         ('Bob', 'B', 30)]
```

Sort Custom Objects

Let’s create a list of custom objects.

```
# Custom class
class Student:
        def __init__(self, name, grade, age):
                self.name = name
                self.grade = grade
                self.age = age
        def __repr__(self):
                return repr((self.name, self.grade, self.age))

# a list of custom objects
L = [Student('Bob', 'B', 30),
        Student('Sam', 'A', 35),
        Student('Max', 'B', 25)]
```

Here are some techniques to sort a list of custom objects.

```
# with lambda
x = sorted(L, key=lambda student: student.age)
print(x)
# [('Max', 'B', 25), ('Bob', 'B', 30), ('Sam', 'A', 35)]

# with attrgetter
from operator import attrgetter
x = sorted(L, key=attrgetter('age'))
print(x)
# [('Max', 'B', 25), ('Bob', 'B', 30), ('Sam', 'A', 35)]

# Custom Objects - Multiple Level Sorting
# Sort by grade then by age
x = sorted(L, key=attrgetter('grade', 'age'))
print(x)
# [('Sam', 'A', 35), ('Max', 'B', 25), ('Bob', 'B', 30)]
```

Python sum() Function

Sums items of an iterable

Usage

The `sum()` function sums the items of an iterable and returns the total.

If you specify an optional parameter start, it will be added to the final sum.

This function is created specifically for numeric values. For other values, it will raise TypeError.

Syntax

```
sum(iterable,start)
```

Parameter	Condition	Description
iterable	Required	An iterable (such as list, tuple etc.)
start	Optional	A value to be added to the final sum. Default is 0.

Examples

```
# Return the sum of all items in a list

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

x = sum(L)

print(x)
# Prints 15
```

If you specify an optional parameter start, it will be added to the final sum.

```
# Start with '10' and add all items in a list

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

x = sum(L, 10)

print(x)
# Prints 25
```