

# Python From Scratch

## Python Lists

### **Lesson 6 Content**

- **Python Lists**

- List
- List Items
- Ordered
- Changeable
- Allow Duplicates
- List Length
- List Items - Data Types
- type()
- The list() Constructor
- Python Collections (Arrays)

- **Python - Access List Items**

- Access Items
- Negative Indexing
- Range of Indexes
- Range of Negative Indexes
- Check if Item Exists

- **Python - Change List Items**

- Change Item Value
- Change a Range of Item Values
- Insert Items

- **Python - Add List Items**

- Append Items
- insert Items
- Extend List
- Add Any Iterable

- **Python - Remove List Items**

- Remove Specified Item
- Remove Specified Index
- Clear the List

- **Python - Loop Lists**

- Loop Through a List
- Loop Through the Index Numbers
- Using a While Loop
- Looping Using List Comprehension

- **Python - List Comprehension**

- The Syntax
- Condition
- Iterable
- Expression

- **Python - Sort Lists**

- Sort List Alphanumerically
- Sort Descending
- Customize Sort Function
- Case Insensitive Sort
- Reverse Order

- **Python - Copy Lists**

- **Python - Join Lists**

- **Python - List Methods**

- **Python - List Exercises**

## Python Lists

```
mylist = ["apple", "banana", "cherry"]
```

### List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

Lists are created using square brackets:

#### Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

---

### List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

### Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

**Note:** There are some [list methods](#) that will change the order, but in general: the order of the items will not change.

### Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

### Allow Duplicates

Since lists are indexed, lists can have items with the same value:

#### Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]  
print(thislist)
```

### List Length

To determine how many items a list has, use the `len()` function:

#### Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

## List Items - Data Types

List items can be of any data type:

### Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

### Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

---

## type()

From Python's perspective, lists are defined as objects with the data type 'list': <class 'list'>

### Example

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

---

## The list() Constructor

It is also possible to use the `list()` constructor when creating a new list.

### Example

Using the `list()` constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

---

## Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered\*\* and changeable. No duplicate members.

\*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

\*\*As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

## Python - Access List Items

### Access Items

List items are indexed and you can access them by referring to the index number:

#### Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

**Note:** The first item has index 0.

### Negative Indexing

Negative indexing means start from the end -1 refers to the last item, -2 refers to the second last item etc.

#### Example

Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

### Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

#### Example

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

#### Example

This example returns the items from the beginning to, but NOT including, "kiwi":

```
thislist = ["apple", "banana", "cherry", "orange",
"kiwi", "melon", "mango"]
print(thislist[:4])
```

By leaving out the end value, the range will go on to the end of the list:

#### Example

This example returns the items from "cherry" to the end:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango"]
print(thislist[2:])
```

### Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list:

#### Example

This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

```
thislist = ["apple", "banana", "cherry", "orange",
"kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

### Check if Item Exists

To determine if a specified item is present in a list use the in keyword:

#### Example

Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```



## Python - Change List Items

### Change Item Value

To change the value of a specific item, refer to the index number:

#### Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

### Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

#### Example

Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]  
thislist[1:3] = ["blackcurrant", "watermelon"]  
print(thislist)
```

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

#### Example

Change the second value by replacing it with *two* new values:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:2] = ["blackcurrant", "watermelon"]  
print(thislist)
```

**Note:** The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

#### Example

Change the second and third value by replacing it with *one* value:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1:3] = ["watermelon"]  
print(thislist)
```

### Insert Items

To insert a new list item, without replacing any of the existing values, we can use the `insert()` method. The `insert()` method inserts an item at the specified index:

#### Example

Insert "watermelon" as the third item:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(2, "watermelon")  
print(thislist)
```

**Note:** As a result of the example above, the list will now contain 4 items.

## Python - Add List Items

### Append Items

To add an item to the end of the list, use the `append()` method:

#### Example

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

### Insert Items

To insert a list item at a specified index, use the `insert()` method.

The `insert()` method inserts an item at the specified index:

#### Example

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

**Note:** As a result of the examples above, the lists will now contain 4 items.

### Extend List

To append elements from *another list* to the current list, use the `extend()` method.

#### Example

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

The elements will be added to the *end* of the list.

### Add Any Iterable

The `extend()` method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

#### Example

Add elements of a tuple to a list:

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

## Python - Remove List Items

### Remove Specified Item

The `remove()` method removes the specified item.

#### Example

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

### Remove Specified Index

The `pop()` method removes the specified index.

#### Example

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

If you do not specify the index, the `pop()` method removes the last item.

#### Example

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

The `del` keyword also removes the specified index:

#### Example

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

The `del` keyword can also delete the list completely.

#### Example

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

### Clear the List

The `clear()` method empties the list.

The list still remains, but it has no content.

#### Example

Clear the list content:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

## Python - Loop Lists

### Loop Through a List

You can loop through the list items by using a **for** loop:

#### Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
    print(x)
```

### Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the **range()** and **len()** functions to create a suitable iterable.

#### Example

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

The iterable created in the example above is **[0, 1, 2]**.

### Using a While Loop

You can loop through the list items by using a **while** loop.

Use the **len()** function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

#### Example

Print all items, using a **while** loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
    print(thislist[i])
    i = i + 1
```

### Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

#### Example

A short hand **for** loop that will print all items in a list:

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```



## Python - List Comprehension

### List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

#### Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name. Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:

#### Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

### The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

### Condition

The *condition* is like a filter that only accepts the items that valuate to **True**.

#### Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

The condition `if x != "apple"` will return **True** for all elements other than "apple", making the new list contain all fruits except "apple".

The *condition* is optional and can be omitted:

#### Example

With no **if** statement:

```
newlist = [x for x in fruits]
```

### Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

#### Example

You can use the `range()` function to create an iterable:

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

#### Example

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

### Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

#### Example

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like:

#### Example

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

#### Example

Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

The *expression* in the example above says:

"Return the item if it is not banana, if it is banana return orange".

## Python - Sort Lists

### Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

#### Example

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

### Sort Descending

To sort descending, use the keyword argument `reverse = True`:

#### Example

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

### Customize Sort Function

You can also customize your own function by using the keyword argument `key = function`.

The function will return a number that will be used to sort the list (the lowest number first):

#### Example

Sort the list based on how close the number is to 50:

```
def myfunc(n):
    return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

### Case Insensitive Sort

By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:

#### Example

Case sensitive sorting can give an unexpected result:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use `str.lower` as a key function:

#### Example

Perform a case-insensitive sort of the list:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

### Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The `reverse()` method reverses the current sorting order of the elements.

#### Example

Reverse the order of the list items:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

## Python - Copy Lists

### Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

There are ways to make a copy, one way is to use the built-in List method `copy()`.

#### Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Another way to make a copy is to use the built-in method `list()`.

#### Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

## Python - Copy Lists

### Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

There are ways to make a copy, one way is to use the built-in List method `copy()`.

#### Example

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Another way to make a copy is to use the built-in method `list()`.

#### Example

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

## Python - Join Lists

### Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python. One of the easiest ways are by using the `+` operator.

#### Example

Join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

#### Example

Append list2 into list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
for x in list2:
    list1.append(x)
print(list1)
```

Or you can use the `extend()` method, which purpose is to add elements from one list to another list:

#### Example

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

---

## Python List Exercises

### Test Yourself With Exercises

Now you have learned a lot about lists, and how to use them in Python.

Are you ready for a test?

Try to insert the missing part to make the code work as expected:

#### Exercise:

Print the second item in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
print()
```