Python Lists

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:

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

- 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.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- Since lists are indexed, lists can have items with the same value
- List items can be of any data type
- A list can contain different data types
- To determine how many items a list has, use the **len()** function
- It is also possible to use the **list()** constructor when creating a new list.

Access List Items

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

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

Note: The first item has index 0.

```
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).

- if Item Exists

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

- Change Item Value

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

- Change a Range of Item Values

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)
```

Add List Items

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

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

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

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

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

Remove List Items

Remove Specified Item

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

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

Remove Specified Index

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

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

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

The **del** keyword also removes the specified index:

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

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

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

Clear the List

The **clear()** method empties the list.

The list still remains, but it has no content.

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

List count() Method

```
points = [1, 4, 2, 9, 7, 8, 9, 3, 1]
x = points.count(9)
```

index() Method

```
fruits = [4, 55, 64, 32, 16, 32]
x = fruits.index(32)
```

Note: The `index()` method only returns the *first* occurrence of the value.

reverse() Method

```
fruits = ['apple', 'banana', 'cherry']
fruits.reverse()
```

sort() Method

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort()
```

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort(reverse=True)
```

Tuples

```
mytuple = ("apple", "banana", "cherry")
```

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

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

Tuple items are ordered, unchangeable, and allow duplicate values.

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

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

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

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

A tuple can contain different data types:

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Remove Items

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Join Two Tuples

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

count()

Returns the number of times a specified value occurs in a tuple

index()

Searches the tuple for a specified value and returns the position of where it was found

Check if Item Exists

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

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple”)
```

Change Tuple Values

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Add Items

1. Convert into a list:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

2. Add tuple to a tuple.

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
```

Unpack Tuples

Packing a tuple:

```
fruits = ("apple", "banana", “cherry")
```

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits
```

```
print(green)
print(yellow)
print(red)
```

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Asterisk*

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

Loop Through a Tuple

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Loop Through the Index Numbers

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

Using a While Loop

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

Python Sets

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Duplicates Not Allowed

Add Set Items

Once a set is created, you cannot change its items, but you can add new items.

To add one item to a set use the **add()** method.

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

To add items from another set into the current set, use the **update()** method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

Add Any Iterable

Remove Set Items

To remove an item in a set, use the **remove()**, or the **discard()** method.

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered, changeable and do not allow duplicates.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Access Dictionary Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict[“model"]
```

There is also a method called **get()** that will give you the same result:

```
x = thisdict.get("model")
```

Get Keys

The **keys()** method will return a list of all the keys in the dictionary.

```
x = thisdict.keys()
```

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Get Values

The **values()** method will return a list of all the values in the dictionary.

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Get Items

The **items()** method will return each item in a dictionary, as tuples in a list.

```
x = thisdict.items()
```

if Key Exists

To determine if a specified key is present in a dictionary use the **in** keyword:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Dictionary Items

You can change the value of a specific item by referring to its key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The **update()** method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Add Dictionary Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Update Dictionary

The **update()** method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Remove Dictionary Items

The **pop()** method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

The **clear()** method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Loop Dictionaries

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

values() method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

keys() method to return the keys of a dictionary:

```
for x in thisdict.keys():
  print(x)
```

Loop through both keys and values, by using the **items()** method:

```
for x, y in thisdict.items():
  print(x, y)
```

Copy Dictionaries

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

List Comprehension

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

```
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:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

Syntax

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

Python Dictionary Comprehension

```
myDict = {x: x**2 for x in [1,2,3,4,5]}
print (myDict)
```

zip() Function

In Python, the zip() function is used to combine two or more lists (or any other iterables) into a single iterable, where elements from corresponding positions are paired together.

```
name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]
roll_no = [ 4, 1, 3, 2 ]

# using zip() to map values
mapped = zip(name, roll_no)

print(set(mapped))
```

Dictionary Comprehension with Zip:

```
# Lists to represent keys and values
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]

# but this line shows dict comprehension here
myDict = { k:v for (k,v) in zip(keys, values)}


print (myDict)
```

Enumerate() in Python

Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object. This enumerated object can then be used directly for loops or converted into a list of tuples using the list() function.

Syntax:

```
enumerate(iterable, start=0)
```

```
# Python program to illustrate
# enumerate function
l1 = ["eat", "sleep", "repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:", type(obj1))
print (list(enumerate(l1)))

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))
```