List Topic

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Chapter 20: List

The Python List is a general data structure widely used in Python programs. They are found in other languages, often referred to as dynamic arrays. They are both mutable and a sequence data type that allows them to be indexed and sliced. The list can contain different types of objects, including other list objects.

Python Collections

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. This immutable
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered, changeable and non-indexed. No duplicate key members (if so, it takes last key value).
- An ordered collection of n values (n >= 0)
- Lists are mutable, so you can change the values in a list
- A list contains items separated by commas and enclosed within square brackets [ ]. lists are almost similar to arrays in C. One difference is that all the items belonging to a list can be same or of different data type.

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

The elements of a list can be accessed via an index, or numeric representation of their position. Lists in Python are zero-indexed

In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets

The elements of a list are not restricted to a single data type, which makes sense given that Python is a dynamic data language:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

```
a = [1, 2, 3]
print (a)
b = ['a', 1, 'python', (1,2)]
b[2] ='something else'
print (b)
```

Not hashable; mutable

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

```
myList = []
str1 = "ABCD"
for val in str1:
    myList.append(val)
print (myList)
"""IF WE APPEND A STRING INTO A LIST USING FOR LOOP IT SPLITS THE STIRNG
TO AN INDIVIDUAL CHARACTERS AND THEN ADD INTO LIST"""
```

OUTPUT

```
['A', 'B', 'C', 'D']
```

```
silapthikaram = ["சிலப்பதிகாரம்", 21, "Sr.Data Analyst", "Single", "Ph.D", "Canara Bank",
"sb_54", False]
emptyListInt = []
emptyListstr = []
for item in silapthikaram:
    if  isinstance(item,int) == True:
        emptyListInt.append(item + 10)
    else:
        emptyListstr.append(item+ "AAA")
print(emptyListstr)
print(emptyListInt)
ouput
```

['சிலப்பதிகாரம்AAA', 'Sr.Data AnalystAAA', 'SingleAAA', 'Ph.DAAA', 'Canara BankAAA', 'sb_54AAA']
[31, 10]
==========

List comprehension (reduces the lines of code) will cover later along with lambda

However, Python has an easier way to solve this issue using List Comprehension. List comprehension is an elegant way to define and create lists based on existing lists.

The above prgm can be rewritten as below

```
myList = [letter for letter in 'Python']
print (myList)
```

output

['P', 'y', 't', 'h', 'o', 'n']

==========

```
silapthikaram = ["சிலப்பதிகாரம்", 21, "Sr.Data Analyst", "Single", "Ph.D", "Canara Bank", "sb_54", False]
print(silapthikaram)
print(silapthikaram[  :  ])
print(silapthikaram[  1:5  ])

for index, item in enumerate(silapthikaram):
    print(index, item)
```

output

['சிலப்பதிகாரம்', 21, 'Sr.Data Analyst', 'Single', 'Ph.D', 'Canara Bank', 'sb_54', False]

['சிலப்பதிகாரம்', 21, 'Sr.Data Analyst', 'Single', 'Ph.D', 'Canara Bank', 'sb_54', False]

சிலப்பதிகாரம்

==========

```
silapthikaram = ["சிலப்பதிகாரம்", 21, "Sr.Data Analyst", "Single", "Ph.D", "Canara Bank", "sb_54", False]
manimkalai = ["மணிமேகலை", 49, "Data Scientist", "married", "masters",]
tamil = silapthikaram + manimkalai
print(tamil)
(silapthikaram.extend(manimkalai))
print(silapthikaram)
```

output

C:\Users\Melcose\Programs\Python\Python39-32\python.exe C:/Users/Melcose/PycharmProjects/pythonProject/DataScienceMel.py

['சிலப்பதிகாரம்', 21, 'Sr.Data Analyst', 'Single', 'Ph.D', 'Canara Bank', 'sb_54', False, 'மணிமேகலை', 49, 'Data Scientist', 'married', 'masters']

['சிலப்பதிகாரம்', 21, 'Sr.Data Analyst', 'Single', 'Ph.D', 'Canara Bank', 'sb_54', False, 'மணிமேகலை', 49, 'Data Scientist', 'married', 'masters']

==========

```
list = [123,'abcd',10.2,'d']    #can be an array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will output whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

====

Pop function

```
names = ['Alice', 'Craig', 'Diana', 'Diana', 'Eric', "Bob"]
print (names.pop(4))
print (names)
```

Remove and return item at index (defaults to the last item) with names.pop([index]), returns the item

========

Python Lists

Python Collections datatype

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

- **List** is a collection which is ordered and changeable. Allows duplicate members.not hasable ..no hashid
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered, changeable, and indexd(only key can be indexed – after changing the dict to list. No duplicate key members.

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.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Example

Create a List:

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

Access Items

```
silapthikaram = ["சிலப்பதிகாரம்", 21, "Sr.Data Analyst", "Single", "Ph.D", "Canara Bank",
"sb_54", False]
print(silapthikaram[2])
print(silapthikaram[-2])
print(silapthikaram[1:-1])
print(silapthikaram[1::2])
```

Sr.Data Analyst

sb_54

[21, 'Sr.Data Analyst', 'Single', 'Ph.D', 'Canara Bank', 'sb_54']

[21, 'Single', 'Canara Bank', False]

You access the list items by referring to the index number:

Example

Print the second item of the list:

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

Negative Indexing

Negative indexing means beginning 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 "orange":

```
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" and 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 index -4 (included) to index -1 (excluded)

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

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)
```

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)
```

You will learn more about for loops in our Python For Loops Chapter.

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")
```

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))
```

Add 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)
```

To add an item at the specified index, use the insert() method:

Example

Insert an item as the second position:

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

Remove Item

There are several methods to remove items from a list:

Example

The remove() method removes the specified item:

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

Example

The pop() method removes the specified index, (or the last item if index is not specified):

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

Example

The del keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0] # observe..there is no dot/ . after the del
print(thislist)
```

Example

The del keyword can also delete the list completely:

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

Example

The clear() method empties the list:

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

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)
```

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 are 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)
```

The list() Constructor

It is also possible to use the list() constructor to make a new list.

Example

Using the list() constructor to make a List:

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

Test Yourself With Exercises

Exercise:

Print the second item in the fruits list.

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

Section 20.1: List methods and supported operators

==========

1) **append(value)** – appends a new element to the end of the list

Note that the append() method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

```
a = [1, 2, 3, 4, 5]
a.append(6)
a.append(7,)
a.append([80, 90])
print(a)
output
[1, 2, 3, 4, 5, 6, 7, [80, 90]]
======
Append an element of a different type, as list elements do not need to have the same type

a = [1, 2, 3, 4, 5]
a.append(6)
```

```
a.append("AAA")
print (a)
output
[1, 2, 3, 4, 5, 6, 'AAA']
=====

a = [1, 2, 3, 4, 5, 5]
b = [10,20]
a.append(b)
print (a)
print (a[6]) # TAKING THE 6 INDEX OF THE LIST, WHICH IS ANOTHER
LIST THAT WE ADDED JUST NOW

output
[1, 2, 3, 4, 5, 5, [10, 20]]
[10, 20]

==========
```

2.extend(enumerable) - extends the list by appending elements from another enumerable

```
a = [1, 2, 3, 4, 5, 5]
b = [10,20]
a.extend(b)
print (a)
output
[1, 2, 3, 4, 5, 5, 10, 20]

-----------
range(stop) → range object range(start, stop[, step]) → range
object
a = [1, 2, 3, 4, 5, 5]
a.extend(range(3))
print (a)
output
[1, 2, 3, 4, 5, 5, 0, 1, 2] #0,1,2 IS FROM range()
--------------
a = [1, 2, 3, 4, 5, 5]
a.extend(range(2,10,2)) # THIS range(START WITH 2, GOES UP TO 10,
STEP 2)
print (a)
OUTOUT
[1, 2, 3, 4, 5, 5, 2, 4, 6, 8]
```

```
=========
In append(), if we use range(), we will get surprise results, so
DON'T use it
a = [1, 2, 3, 4, 5, 5]
a.append(range(3))
print (a)
output
[1, 2, 3, 4, 5, 5, range(0, 3)]
=======
Extend of the list can be possible using OPERATORS
a = [1, 2, 3, 4, 5, 5]
b = [10, 20, "A", "BBB"]
print (a + b) # WE USE + OPERATOR TO CONCANTENATE 2 LISTS /
OUTPUT
[1, 2, 3, 4, 5, 5, 10, 20, 'A', 'BBB']
===========
```

3)index(value, [startIndex]) - gets the index of the first occurrence of the input value. If the input value is not in the list a ValueError exception is raised. If a second argument is provided, the search is started at that specified index.

```
a = [10, 20, 30, 40, 50, 50, "AAA"]
print(a.index(40))
print(a.index(50))
output
3
4
=====
```
silapthikaram = ["சிலப்பதிகாரம்", 21, **"Sr.Data Analyst"**, **"Single"**, **"Ph.D"**, **"Canara Bank"**, **"sb_54"**, 21, **False**]
for item **in** enumerate(silapthikaram):
    # print(item)
    pass
print(silapthikaram.index(21))
print(silapthikaram.index(21, 3,8)) # it find the index of 21, from 3rd index to 8th index
output
1
7

Notes

```
a.index(7) # Returns: 6
a.index(49) # ValueError, because 49 is not in a.
a.index(7, 7) # Returns: 7
a.index(7, 8) # ValueError, because there is no 7 starting at
index
========
```

4)insert(index, value) - inserts value just **before the specified index.** Thus after the insertion the new element occupies position index.

```
a.insert(0, 0)  # insert 0 at position 0 a.insert(2, 5)  # insert
5 at position
=======
```

5)pop([index]) - removes and returns the item at index. With no argument it removes and returns the last
element of the list.

```
a.pop(2) # Returns: 5 #
a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
 a.pop(8) # Returns: 7 # a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# With no argument:
a.pop() # Returns: 10 # a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
==========
```

6)remove(value) - removes the **first occurrence of the specified value**. If the provided value cannot be found, a6. ValueError is raised.

```
a.remove(0)
a.remove(9)
 # a: [1, 2, 3, 4, 5, 6, 7, 8]
 a.remove(10) # ValueError, because 10 is not in the list - a
```

==========

7)reverse() - reverses the list in-place and returns None

```
a = [10, 20, 30, 40, 50, 50, "AAA"]
print (a.reverse())
```

output

None

TO AVOID IT AND GET THE REVERSE VALUES USE THE BELOW CODE

```
a = [10, 20, 30, 40, 50, 50, "AAA"]
a.reverse()
print(a)
```

OUTPUT

['AAA', 50, 50, 40, 30, 20, 10]

Or use the below code to reverse the list or any iterable

```
a = [10, 20, 30, 40, 50, 50, "AAA"]
print(a[::-1])
```

```
['AAA', 50, 50, 40, 30, 20, 10]
```

==========

8)count(value)

counts the number of occurrences of some value in the list

list.count(5)

==========

9) sort() - sorts the list in numerical and lexicographical order and returns None

```
lst = [10, 50, 20, 30, 40, 50, 100, 50,]
lst.sort()
print (lst)
```

output

```
[10, 20, 30, 40, 50, 50, 50, 100] # sorts the list in numerical
```

====

```
lst = ['a', 'c', 's', 'a',] # sorts the list in lexicographical order
```

```
lst.sort()
print (lst)

output
['a', 'a', 'c', 's']
```

Note: we can **not sort** a list if it has numerical and string

```
lst = ['a', 'c', 's', 'a',20,1,5,]
lst.sort()
print (lst)
output
TypeError: '<' not supported between instances of 'int' and 'str'
======
```

Sort and reverse can be combined together

Lists can also be reversed when sorted using the reverse=True flag in the sort() method.

```
lst = [10, 100, 20, 3, 40, 50, 50,]
lst.sort(reverse=True)
print (lst)
output
[100, 50, 50, 40, 20, 10, 3]
==========
```

10)clear()

removes all items from the list and gives empty list

list.clear()

==========

: multiplying an existing list by an integer will produce a larger list consisting of that many copies11. of the original. This can be useful for example for list initialization:

```
print (lst * 3)
lst1 = [1, 2, 3, 4, 5]
print (lst1 * 5)
output
['ABCD', 'ABCD', 'ABCD']
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2,
3, 4, 5]
========
```

12) Element deletion - it is possible to delete multiple elements in the list using the del keyword and slice notation:

```
a = list(range(10))
del a[::2]
```

```
# a = [1, 3, 5, 7, 9]
del a[-1]
# a = [1, 3, 5, 7]
del a[:]
# a = []
Del (list) // deletes the complete list object
----------
::2 (every other element) ::3 (Every third element) and so on will
be removed (the items divided by 3 or 2 will be removed

a = list(range(20))
del a[::5]
print(a)

Output:
[1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19]
Note: 5,10,15,20 will be removed as is where divided by 5
==================================
a = list(range(20))
print(a)
del a[8::2] # from 8, every other element will be removed
print(a)
```

```
output
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 13, 15, 17, 19]
```

=======

13)Copying

The default assignment "=" assigns a **reference** of the **original list to the new name.** That is, the **original name and new name are both pointing to the same list object.** Changes made through any of them will be reflected in another. This is often not what you intended.

There are 5 ways of copying

1.using assignment operator // lst2 ==lst1
2. using list() // list(list1)
3.using copy()//(import copy package)
4. using deepcopy() //(import copy package)
5. using slicing //b = a[:]

copy using assignment operator

the change made in the list did effect in other lists

In Python, Assignment statements do not copy objects, they create bindings between a target and an object. When we use = operator user thinks that this creates a new object;but, it doesn't. It only creates a new variable that shares the reference of the original object.

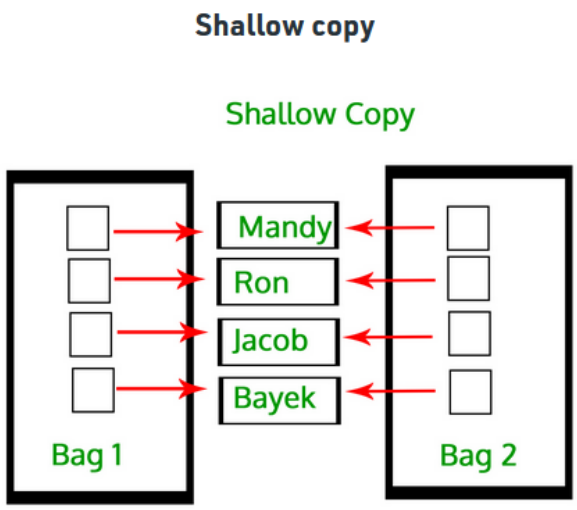
```
import copy
orignalList = [10, 20, 3, 40, 50, 50,]
print("Original list",orignalList)
print("ID of the original list : ", id (orignalList))
duplicateList = orignalList
print("duplicate list",orignalList)
print("ID of the duplicate list : ", id (duplicateList))
print("====================")

duplicateList.append(1000)

print("Original list after copying and adding new element",orignalList)
print("duplicate  list after copying and adding new element",duplicateList)
print("====================")

print("ID of the original list : ", id (orignalList))
print("ID of the original list after co[ying and adding new element: ", id (orignalList))
print("ID of the duplicate list : ", id (duplicateList))
print("ID of the duplicate  list after copying and adding new element: ", id (duplicateList))
```

output

```
Original list [10, 20, 3, 40, 50, 50]
ID of the original list :  25030568
duplicate list [10, 20, 3, 40, 50, 50]
```

```
ID of the duplicate list :  25030568
==================
Original list after copying and adding new element [10, 20, 3, 40, 50, 50, 1000]
duplicate  list after copying and adding new element [10, 20, 3, 40, 50, 50, 1000]
==================
ID of the original list :  25030568
ID of the original list after co[ying and adding new element:  25030568
ID of the duplicate list :  25030568
ID of the duplicate  list after copying and adding new element:  25030568
==================
Original list after adding new element [10, 20, 3, 40, 50, 50, 1000]
duplicate  list after adding new element [10, 20, 3, 40, 50, 50, 1000]
==================
ID of the original list :  35975080
ID of the original list after adding new element:  35975080
ID of the duplicate list :  35975080
ID of the duplicate  list after adding new element:  35975080
-------------


========== ================================================
```

Shallow copy (using copy.copy)



A shallow copy creates a new object which stores the reference of the original elements.

So, a **shallow copy doesn't create a copy of nested objects,** instead it just copies the reference of nested objects. This means, a copy process **does not recurse** or create copies of nested objects itself.

This means it will create new and independent object with same content.

the change made in the list did not effect in other lists

```
Copy using copy() method
import copy
```

```
orignalList = [10, 20, 3, 40, 50, 50,]
print("Original list",orignalList)
print("ID of the original list : ", id (orignalList))
duplicateList = copy.copy(orignalList)
print("duplicate list",orignalList)
print("ID of the duplicate list : ", id (duplicateList))
print("====================")

duplicateList.append(1000)

print("Original list after copying and adding new element",orignalList)
print("duplicate  list after copying and adding new element",duplicateList)
print("====================")

print("ID of the original list : ", id (orignalList))
print("ID of the original list after co[ying and adding new element: ", id (orignalList))
print("ID of the duplicate list : ", id (duplicateList))
print("ID of the duplicate  list after copying and adding new element: ", id (duplicateList))
```

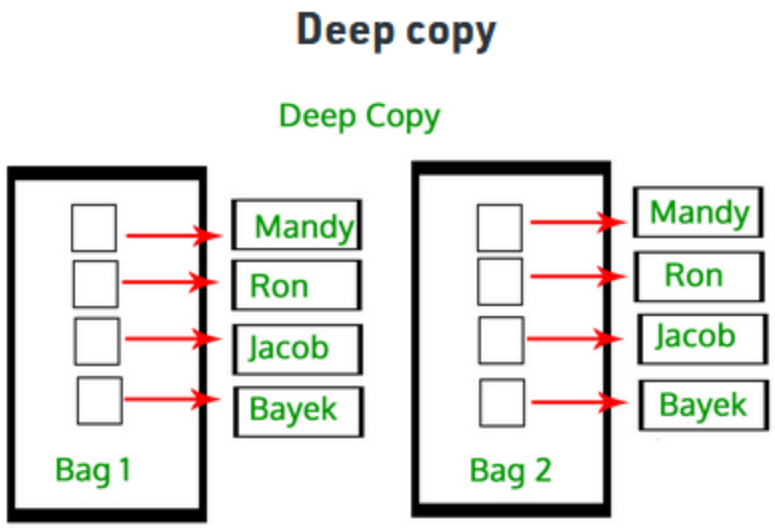
Output

```
Original list [10, 20, 3, 40, 50, 50]
ID of the original list :  33943464
duplicate list [10, 20, 3, 40, 50, 50]
ID of the duplicate list :  33942536
====================
```

```
Original list after copying and adding new element [10, 20, 3, 40,
50, 50]
duplicate  list after copying and adding new element [10, 20, 3,
40, 50, 50, 1000]
====================
ID of the original list :  33943464
ID of the original list after co[ying and adding new element:
33943464
ID of the duplicate list :  33942536
ID of the duplicate  list after copying and adding new element:
33942536
================ ================================
==================
```

Deepcopy (using copy.deepcopy)

the change made in the list did not effect in other lists



Copy using deepcopy() method

A deep copy creates a new object and **recursively adds the copies of nested objects** present in the original elements.
The deep copy creates independent copy of original object and all its nested objects.

```
import copy
orignalList = [10, 20, 3, 40, 50, 50,]
print("Original list",orignalList)
print("ID of the original list : ", id (orignalList))
duplicateList = copy.deepcopy(orignalList)
print("duplicate list",orignalList)
```

```
print("ID of the duplicate list : ", id (duplicateList))
print("====================")

duplicateList.append(1000)

print("Original list after copying and adding new element",orignalList)
print("duplicate  list after copying and adding new element",duplicateList)
print("====================")

print("ID of the original list : ", id (orignalList))
print("ID of the original list after co[ying and adding new element: ", id (orignalList))
print("ID of the duplicate list : ", id (duplicateList))
print("ID of the duplicate  list after copying and adding new element: ", id (duplicateList))
```

```
output
Original list [10, 20, 3, 40, 50, 50]
ID of the original list :  21884840
duplicate list [10, 20, 3, 40, 50, 50]
ID of the duplicate list :  21883912
====================
Original list after copying and adding new element [10, 20, 3, 40,
50, 50]
duplicate  list after copying and adding new element [10, 20, 3,
40, 50, 50, 1000]
====================
```

```
ID of the original list :  21884840
ID of the original list after co[ying and adding new element:
21884840
ID of the duplicate list :  21883912
ID of the duplicate  list after copying and adding new element:
21883912

==============
```

Copy vs Deep copy()

The default assignment "=" assigns a reference of the original list to the new name. That is, the original n and new name are both pointing to the same list object. Changes made through any of them will be refl in another. This is often not what you intended.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

If you want to create a copy of the list you have below options.

You can slice it:

```
new_list = old_list[:]
```

You can use the built in list() function:

```
new_list = list(old_list)
```

You can use generic copy.copy():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

This is a little slower than list() because it has to find out the datatype of old_list first.

If the list contains objects and you want to copy them as well, use generic copy.deepcopy():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

Section 20.2: Accessing list values

Python lists are zero-indexed, and act like arrays in other languages
Attempting to access an index outside the bounds of the list will raise an IndexError.
Negative indices are interpreted as counting from the end of the list

```
lst = [1, 2, 3, 4]
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of ran
```

This is functionally equivalent to
lst[len(lst)-1] # 4

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,90]

print(a[len(a)-1])
print(a[len(a)-2])
```

output
90
9

Lists allow to use slice notation as lst[start:end:step]. The output of the **slice notation is a new list containing elements from index start to end-1**. If options are omitted start defaults to beginning of list, end to end of list and step to 1:

```
lst = [5, 10,20,40,67]
print (lst[1])
print(lst[:3])
print(lst[::2])
print (lst[-1:0:-1]) #[4, 3, 2, 1] // -1 is reverse step
print (lst[-1:0:-2]) #[67, 20] //  -2 is reverse step
print(lst[5:8])  # [] since starting index is greater than length of lst, returns empty list
print (lst[1:10]) # [2, 3, 4] same as omitting ending index
#
```

```
10
[5, 10, 20]
[5, 20, 67]
[67, 40, 20, 10]
[67, 20]
[]
[10, 20, 40, 67]
```

=======

With this in mind, you can print a reversed version of the list by calling

```
lst[::-1]    # [4, 3, 2, 1]
```

=======

When using **step** lengths of negative amounts, the starting index has to be greater than the ending index otherwise the result will be an empty list (This is important)

lst[3:1:-1] # [4, 3]

======

Using negative step indices are equivalent to the following code:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Section 20.3: Checking if list is empty

The emptiness of a list is associated to the boolean **False**, so you don't have to check len(lst) == 0, but just lst or not lst

```
lst=[]
if not lst:
    print("List is empty")
```

OR use the below

```
lst = []
if len(lst) == 0:
    print ("List is empty")
```

Section 20.4: Iterating over a list

AN ITEM FROM LIST CAN BE ACCESSED / ITERATED USING THREE WAYS

1. By using normal for loop
2. By using enumerate (it always takes an iterator object)
3. Finding a length using range function and supply the length to list index

1.By using normal for loop

```
my_list = ["foo", 'bar', 'baz']
for item in my_list:
    print(item)
```

output

foo

```
bar
baz
====
```

2. By using enumerate (it always takes an iterator object)

You can also get the position of each item at the same time:

```
my_list = ["foo", 'bar', 'baz']
for (index, item) in enumerate(my_list): # enumerate gives both
index and value, but for loop give ony value
    print ("The item in position {} is : {}" .format(index, item))

output
The item in position 0 is : foo
The item in position 1 is : bar
The item in position 2 is : baz
====
```

3. Finding a length using range function and supply the length to list index

```
my_list = ["foo", 'bar', 'baz']
for i in range (0, len(my_list)):
```

```
    print ((my_list[i]))

""" WE FIND THE LEN OF THE LIST AND THE SAME IS USED AS INDEX
 IN ORDER TO TAKE THE VALUE"""
```

OUTPUT
foo
bar
baz

Changing items in a list while iterating on it may have unexpected results:

```
my_list = ["foo", 'bar', 'baz']
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print (item)
```

output

```
foo
baz
```

Note: we asked to delete first element 'foo', but it deletes the second element. **SO AVOID MODIFYING THE LIST IN THE FOR LOOP**

If we want to modidy, use the below program, that is by not using the for loop

```
my_list = ["foo", 'bar', 'bazooo']
if my_list[0] == 'foo':
    del my_list[0]
print (my_list)
```

output

[bar', 'bazooo']

In this last example, we deleted the first item at the first iteration, but that caused bar to be skipped

Section 20.5: Checking whether an item is in a list

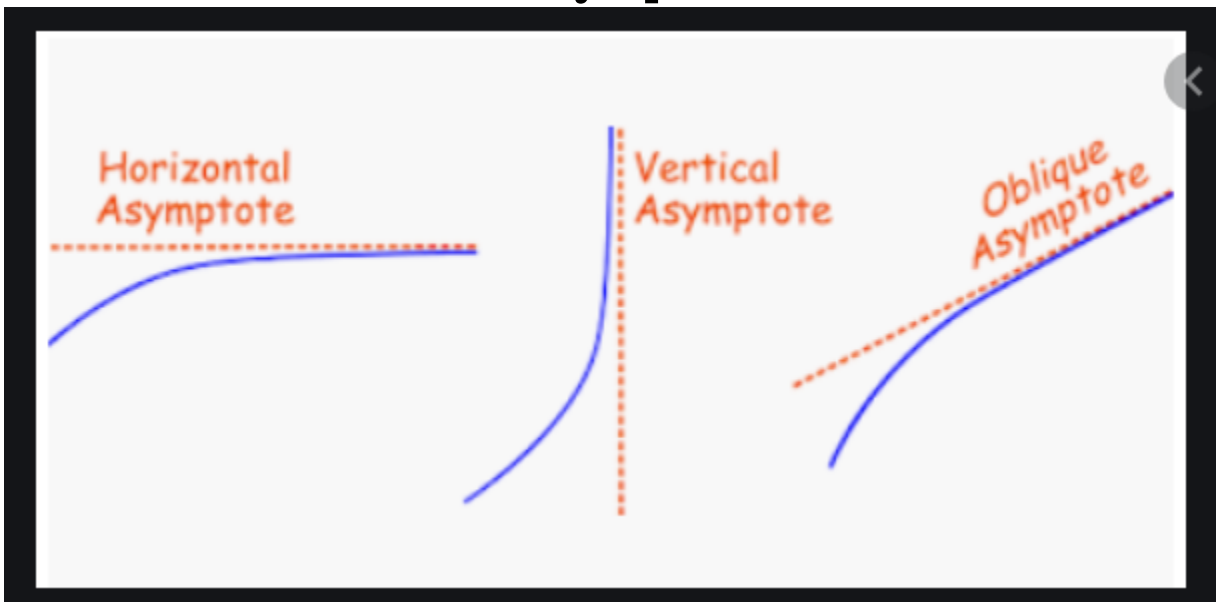
Python makes it very simple to check whether an item is in a list. Simply use the in operator.

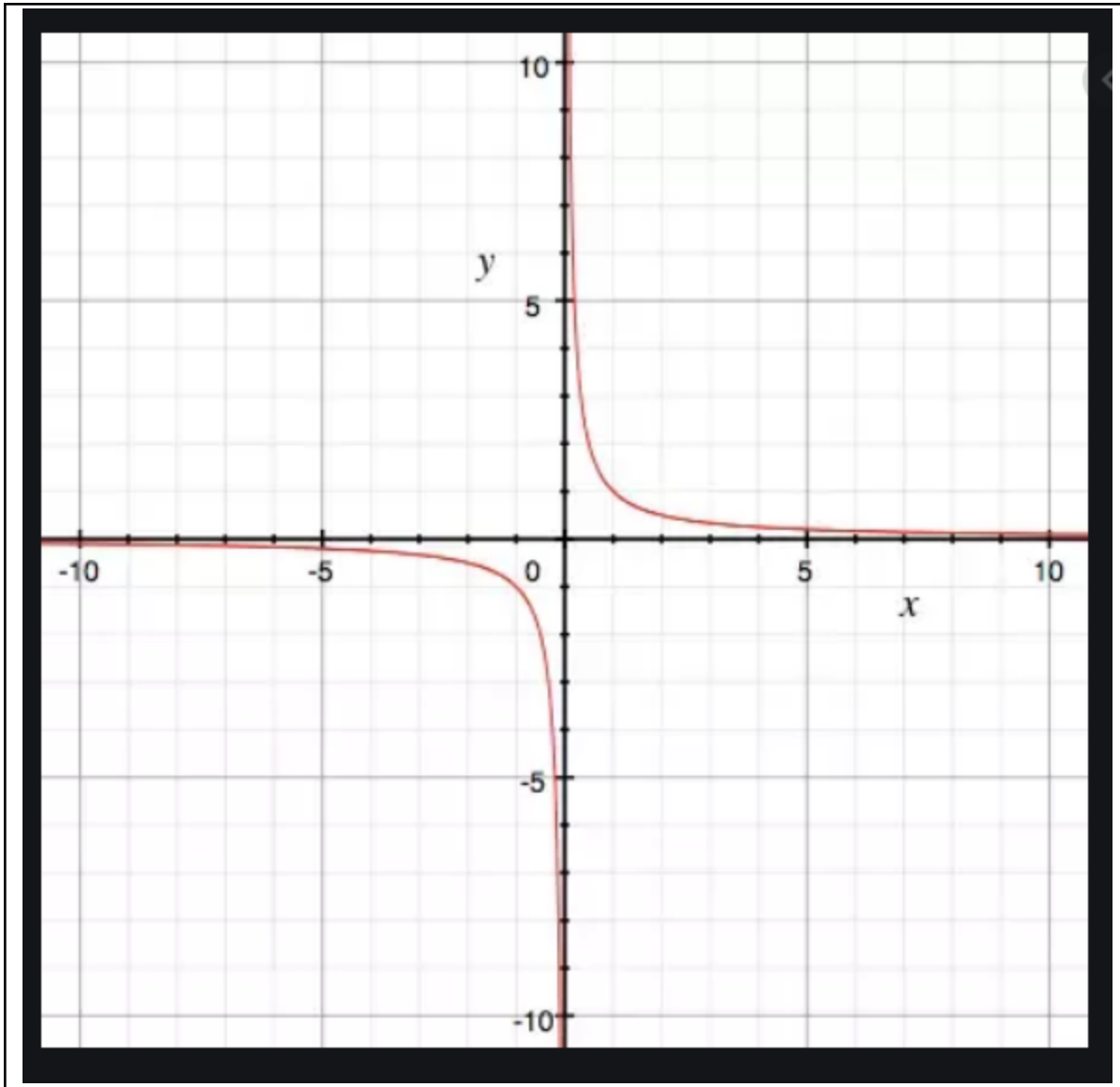
```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True
'toast' in lst
# Out: False
```

Note: the in operator on sets is asymptotically faster than on lists. If you need to use it many times on potentially large lists, you may want to convert your list to a set, and test the presence of elements on the set.

```
slst = set(lst)
'test' in slst
# Out: True
```

The **definition** of **asymptotic** is a line that approaches a curve but never touches. A curve and a line that get closer but **do** not intersect are examples of a curve and a line that are **asymptotic** to each other.




10
y
5
-10
-5
0
5
10
x
-5
-10

Section 20.6: Any and All

You can use all() to determine if all the values in an iterable evaluate to True
Return True if bool(x) is True for all values x in the iterable.
If the iterable is empty, return True.

```
numbs = [1, 1, 0, 1]
a=all (numbs)
print (a)

numbs = [1, 1, 1, 1]
print (all(numbs))

numbs = [1, 1, 0, 1]
print (any(numbs))

numbs = []
print (all(numbs))
'''all = GIVES TRUE'''

numbs = []
print (any(numbs))
```

```
'''any = GIVES FALSE" If the iterable is empty, return False."'''

nums = [None, None, None]
print (any(nums))
"""IT GIVES FALSE,IT TREATS "NONE" AS FALSE VALSE"""

nums = [None, None, None]
print (all(nums))
"""IT GIVES FALSE, ALL NONEs FALSE FALSE"""

nums = [None, None, None, True]
print (all(nums))
"""IT GIVES FALSE"""

"""
vals = [1, 2, 3, 4]
any (val > 12  for val in vals)
print (val)
"""
"""
THE ABOVE EXPRESSION DOES NOT WORK, SO USE THE BELOW.
THE ABOVE IS NOT A FOR LOOP BUT CALLED generator expression
```

```
IN COMPREHENSION and generator expression THE VARIABLE NAME(val)
MUST BE PLACED BEFORE AND AFTER THE for KEYWORD IN THE FOR LOOP.
WE USE print + EXPRESSION IN ONE SENTENCE / AS ONE EXPRESSION
"""
vals = [1, 2, 3, 4,]
print(any (val > 12  for val in vals))

print (any((val * 2) > 6 for val in vals))
"""
HERE, WE FIRST MULTIPLY EVERY LIST'S VALUE WITH 2, THEN CHECKS
IF THE VALUES ARE > 6. IF ANY ONE VALUE IS > 6, IT GIVES TRUE
"""
```

OUTPUT

```
False
True
True
True
False
False
False
False
False
True
```

=======

```
vals = [1,2,3,4]
a =  any(val > 12 for val in vals)
print(a)
```

output

False

=======

You can use `all()` to determine if all the values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Likewise, `any()` determines if one or more values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

While this example uses a list, it is important to note these built-ins work with any iterable, including generator

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Section 20.7: Reversing list elements

Using reverse() method

```
my_list = ['C', 'B', 'A']
my_list.reverse()
print (my_list)
```

output

['A', 'B', 'C']

===

Using reversed() method

```
my_list = ['C', 'B', 'A', 'Z']
reversed(my_list)
print (my_list)
```

output

['C', 'B', 'A', 'Z']

========

THE ABOVE PROGRAM CAN BE WRITTEN USING FUNCTION

```
def method1(lst):
   return lst.reverse()

my_list = ['C', 'B', 'A']
method1 (my_list)
print ("GET THIS RESULT FROM return",my_list)
a=my_list
""" HERE WE ASSIGN THE RESULT THAT WE GOT FROM return of method1,
TO A VARIABLE (a) AND THEN PRINT IT"""
print (a)
```

========

The above program can be rewritten by passing a list parameter as None to a function

```
def method1(lst = None):
    if lst == None:
        lst = []
    return lst.reverse()

my_list = ['C', 'B', 'A', 3, 2, 1]
method1 (my_list)
print ("GET THIS RESULT FROM return",my_list)
a=my_list
print (a)
output
GET THIS RESULT FROM return [1, 2, 3, 'A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
==========
```

IMPORTANT NOTE: reversed() method **DOES NOT** reverse the items of the list, it just reverse the object. See the below prgm.

reverse() actually reverses the elements **in the** container. **reversed**() doesn't actually **reverse** anything, it merely returns an object that can be used to iterate over the container's elements in **reverse** order.

```
my_list = [1, 2, 3, 4,]
reversed(my_list)
print (my_list)
output
[1, 2, 3, 4] # reversed () method did not reverse the list items.
So USE reverse (), because reverse (), is part the List class'
method, whereas reversed
IS A CLASS. Reversed() gives iterator object - see below
my_list = [1, 2, 3, 4,]
a = reversed(my_list)
print (a)
print (next(a))
print(list(a))
output
<list_reverseiterator object at 0x000002B03825D088>
4
[3, 2, 1]
```

==========

Note don’t use this below code and it doesn’t give the expected result

```
my_list = ['C', 'B', 'A']
print (my_list.reverse())
output
None
```

or the below code

```
my_list = ['C', 'B', 'A']
rev = my_list.reverse()
print (rev)
```

output none

========

The right approach is below

```
my_list = ['C', 'B', 'A']
my_list.reverse()
print(my_list)
```

output

['A', 'B', 'C']

Section 20.8: Concatenate and Merge list

The simplest way to concatenate list1 and list2:

merged = list1 + list2

```
list1 = ['C', 'B', 'A',]
list2 = [3, 2, 1]
print (list1 + list2)
```

output

['C', 'B', 'A', 3, 2, 1]

=====

zip returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument. sequences or iterables:

```
list1 = ['a1', 'b1', 'c1',]
list2 = ['a2', 'b2', 'b3']
for a, b in zip(list1, list2):
    print (a,b)
```

output

a1 a2

b1 b2

c1 b3

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and return it.

ZIP

zip(*iterables) –> zip object

Return a zip object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration

@overload def zip(__iter1: Iterable[_T1]) -> Iterator[Tuple[_T1]]
Possible types:
• (__iter1: Iterable[_T1]) -> Iterator[Tuple[_T1]]
• (__iter1: Iterable[_T1], __iter2: Iterable[_T2]) -> Iterator[Tuple[_T1, _T2]]
• (__iter1: Iterable[_T1], __iter2: Iterable[_T2], __iter3: Iterable[_T3]) -> Iterator[Tuple[_T1, _T2, _T3]]
(It goes on and on and on)
=======

```
list1 = ['a1', 'b1', 'c1',]
list2 = ['a2', 'b2', 'b3']
c=(zip(list1,list2))
print(list(c)
```

output
[('a1', 'a2'), ('b1', 'b2'), ('c1', 'b3')]
==========
If the lists have different lengths then the result will include only as many elements as the shortest one:

```
list1 = ['a1', 'b1', 'c1', 'd1', 'f1']
list2 = ['a2', 'b2', 'b3']
for a, b in zip(list1, list2):
    print (a,b)
```

output
a1 a2
b1 b2
c1 b3
NOTE: When we use for loop we get tubles in sequence (ie one by one)
======

```
blst = ['a1', 'b1', 'c1', 'd1', 'f1']
alist = []
print(list(zip(alist, blst)))
```

output
[]
Notes : If a list is empty and if we try to merge with another list that has elements in it (using zip), it results an emply list

If the lists have different lengths then the result will include only as many elements as the shortest one (here the shortest list is [], so we get [] as output

==========

Merging 2 lists using zip

```
alst = ['a1', 'b1', 'c1', 'd1', 'f1']
blst = ['a1', 'b1', 'c1', 'd1', 'f1']
print(list(zip(alst, blst)))
```

output

[('a1', 'a1'), ('b1', 'b1'), ('c1', 'c1'), ('d1', 'd1'), ('f1', 'f1')]

=======

Merging 3 lists using zip

```
alst = ['a1', 'b1', 'c1', 'd1', 'f1']
blst = ['a1', 'b1', 'c1', 'd1', 'f1']
clst = ['a1', 'b1', 'c1', 'd1', 'f1']

print(list(zip(alst, blst,clst)))
```

output

[('a1', 'a1', 'a1'), ('b1', 'b1', 'b1'), ('c1', 'c1', 'c1'), ('d1', 'd1', 'd1'), ('f1', 'f1', 'f1')]

=========

For padding lists of unequal length to the longest one with Nones use itertools.zip_longest (itertools.izip_longest in Python 2)

```
import itertools
alst = ['a1', 'b1', ]
blst = ['c1', 'd1', 'e1', ]
```

```
clst = ['f1', 'g1', 'h1', 'i1', ]

for a,b,c in itertools.zip_longest(alst, blst, clst):
    print (a, b, c)
```

output

a1 c1 f1
b1 d1 g1
None e1 h1
None None i1

The above program can be written as below, without using forloop

```
print(list(itertools.zip_longest(alst, blst, clst)))
==========
import itertools

list1 = ['a1', 'b1', 'c1', 'd1', 'f1']
list2 = ['a2', 'b2', 'b3']
c = itertools.zip_longest(list1, list2)
print(c)
print(list(c))
output
<itertools.zip_longest object at 0x0000029D3AEF3F48>
```

```
[('a1', 'a2'), ('b1', 'b2'), ('c1', 'b3'), ('d1', None),
('f1', None)]
```

=======

itertools def zip_longest(*p: Iterable,
fillvalue: Any = ...) -> Iterator

zip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> zip_longest object

Return a zip_longest object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration.

When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

```
import itertools

list1 = ['a1', 'b1', 'c1', 'd1', 'f1']
list2 = ['a2', 'b2', 'b3']
c = itertools.zip_longest(list1, list2, fillvalue="AAA")
print(c)
print(list(c))
```

output

<itertools.zip_longest object at 0x000001F80925B638>

[('a1', 'a2'), ('b1', 'b2'), ('c1', 'b3'), ('d1', 'AAA'), ('f1', 'AAA')]

==========

Insert to a specific index values

insert(self, __index, __object)

```
alist = [123, 'AAA', 'b', 345]
alist.insert(2,"BBB")
print (alist)
```

output

[123, 'AAA', 'BBB', 'b', 345]

Notes: Diff between append () and insert()

append(), adds the the given element to the **end of the list**, where as insert() insert the given element based on index given.

Section 20.9: Length of a list

Use len() to get the one-dimensional length of a list.

len() also works on strings, dictionaries, and other data structures similar to lists.

Note that len() is a built-in function, not a method of a list object.

Also note that the cost of len() is O(1), meaning it will take the same amount of time to get the length of a list regardless of its length.

Section 20.10: Remove duplicate values in list

Removing duplicate values in a list **can be done by converting the list to a set (**that is an unordered collection of distinct objects). If a list data structure is needed, then the set can be converted back to a list using the function list():

```
alist = [123, 'AAA', 'b', 345, 'AAA']
alist = list(set(alist))
print (alist)
```

output

[345, 'AAA', 123, 'b']

Note that by converting a list to a set the original ordering is lost.

==========

OrderedDict

Read this https://www.geeksforgeeks.org/ordereddict-in-python/ to understand OrderedDict

To **preserve the order** of the list one can use an OrderedDict – see the pgm below

OrderedDict removes the duplicate but maintains the order

```
import collections
alist = [123, 'AAA', 'b', 345, 'AAA']
```

```
alist = collections.OrderedDict.fromkeys(alist).keys()
print (alist)
print (type(alist))
output
odict_keys([123, 'AAA', 'b', 345])
<class 'odict_keys'>

Note: the output is in dictionary form, the list type has been
converted to <class 'odict_keys'>  type. Again we have to convert
from <class 'odict_keys'>   to list type as below. (we do perfom
this in order to remove the duplicate and keep the order of the
list intact

import collections
alist = [123, 'AAA', 'b', 345, 'AAA']
alist = collections.OrderedDict.fromkeys(alist).keys() # It
removes the the duplicates and gives the list items as dictionary
keys
print (alist)
print (type(alist))
alist = list(alist) # here we convert the <class 'odict_keys'>  to
list type
```

```
print (alist)
print (type(alist))

output
odict_keys([123, 'AAA', 'b', 345])
<class 'odict_keys'>
[123, 'AAA', 'b', 345]
<class 'list'>
=====
```

Section 20.11: Comparison of lists

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; **if they are equal**, the next two items are compared, and so on, until either sequence is exhausted.

By default, tuples (or lists) are compared similar to how strings are compared: the comparison checks **corresponding items** until it finds a pair that aren't equal (or one collection runs out of items), and then the comparison stops

Notes: that if the first items of the first and and second list value is EQUAL, it goes to next set

As soon as an unequal pair is found, the overall result is the result of comparing the unequal items.

The comparison of pairs will stop when *either* an unequal pair of items is found *or*--if the lists are different lengths--the end of the shorter list is reached.

```
print ([10, 10, 102, ] > [10, 10, 101,5 ]) // TRUE, the reason is it takes and compare the SHORTEST list


print ([1, 10, 100, ] < [2, 1, 1, ]) // TRUE
```

Note: it takes the FIRST item from both lists and compare/gives the results.If it is true,It DOES not compare the second or consecutive itmes for comparison

```
---------
print ([1, 10, 100, ] ⩽ [1, 10, 100, ]) # TRUE
print ([1, 10, 100, ] < [1, 10, 100, ]) #FALSE, because lists are equal
print ([1, 10, 100, ] ⩾ [1, 10, 100, ]) # TRUE
print ([1, 10, 100, ] > [1, 10, 100, ]) #FALSE, because lists are equal
---------
```

```
print ([10, 10, 100, ] == [10, 10, 100]) // // TRUE (Check the
COMPARISON OPERATOR TO COMPARE THE 2 LISTS

print (['a', "b" , 'c', ] == ['a', "b" , 'c', ]) // TRUE (Check
the COMPARISON OPERATOR TO COMPARE THE 2 LISTS

print (['a', "b" , 'c', ] == ['a', "b" , 'A', ]) # FALSE
---------

NOTE : ALL THE ABOVE RULES WILL BE WORKED ONLY IF WE USE OTHTER
OPERATORS BUT ==  (COMAPRISON OPERATOR) ..SEE EXAMPLE
print ([10, 10, 100, ] == [10, 10, 100, 5, ]) // FALSE, HERE THE
SHORTEST LIST WILL NOT BE TAKEN INTO ACCOUNT, THE COMPARISON
OPERATOR CHECKS THE NUMBER OF ELEMENTS TOO
-----------
```
If one of the lists is contained at the start of the other, the shortest list wins.
```
print ([1, 10, ] < [1, 10, 100 ]) TRUE
print ([1, 10, ] > [1, 10, 100 ]) FALSE
```

when we have 2 different len of list to compare, if the first list's len is lesser than second, it becomes TRUE . This rule applicable only if the len of first list is lesser than the second

---

BUT see below the == (COMAPRISON OPERATOR) …

```
print ([1, 10, ] == [1, 10, 100 ]) // FALSE…
```

========

You can use a list comprehension for comparing two lists element wise then use `all` function to check all of comparison are `True`

```
a = [8,9,9,11]
b = [8,7,20,10]
x = [a[i] >= b[i] for i in range(len(a))]
print(x)
```

output

```
[True, True, False, True]
```

```
a = [8,9,9,11]
b = [8,7,20,10]
print(all([a[i] >= b[i] for i in range(len(a))]))
print(any([a[i] >= b[i] for i in range(len(a))]))
```

output

```
False
```

```
True
------------
```

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

If one of the lists is contained at the start of the other, the shortest list wins.

```
[1, 10] < [1, 10, 100]
# True
```

Section 20.12: Accessing values in nested list

Starting with a three-dimensional list:

```
alist = [ [[1, 2, ], [3, 4, ]],    [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,] ] ]
print (alist[1][2][2])
```

output is , from first index ([[5, 6, 7,], [8, 9, 10], [11, 12, 13,]]) , then 2nd index ([11, 12, 13,]])) then 2nd index value (13)
#Accesses the third element in the second list in the second list
======
Performing support operations:
Append()

```
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,]] ]
''' TO append/add THE VALUE 10 after the 4'''
alist [0] [1].append(10)
print (alist)
output
[[[1, 2], [3, 4, 10]], [[5, 6, 7], [8, 9, 10], [11, 12, 13]]] //
BY DEFAULT, APPEND ADD AN ITEM AT THE END OF THE LIST
=====
```

Insert()

```
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,]] ]
''' TO INSERT THE VALUE 100'''
alist [0] [1].insert(1, 100)
print (alist)
```

```
output
[[[1, 2], [3, 100, 4]], [[5, 6, 7], [8, 9, 10], [11, 12, 13]]]
======
Len()
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11,
12, 13,]] ]
print(len(alist[1]))
output
3
=====
Output
Using nested for loops to print the list:

alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11,
12, 13,]] ]
for row in alist:
    print (row)
    for col in row:
        print (col)

[[1, 2], [3, 4]]
[1, 2]
```

```
[3, 4]
[[5, 6, 7], [8, 9, 10], [11, 12, 13]]
[5, 6, 7]
[8, 9, 10]
[11, 12, 13]
```

It shows 2 rows, first row has 2 columns and the second row has 3 columns

==========

Extend the above code with another **for** loop

```
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,]] ]
print(len(alist))
for row in alist:
    print(row)
    for col in row:
        print(col)
        for item in col:
            print(item)
```

output

```
2
[[1, 2], [3, 4]]
[1, 2]
1
2
[3, 4]
3
4
```

```
[[5, 6, 7], [8, 9, 10], [11, 12, 13]]
[5, 6, 7]
5
6
7
[8, 9, 10]
8
9
10
[11, 12, 13]
11
12
13
==============
```

Note that this operation can be used in a list comprehension or even as a generator to produce efficiencies, e.g.

```
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,]] ]
print ([col for row in alist for col in row])
```

output

```
[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10], [11, 12, 13]]
```

Note: since it list comprehension, the 2 rows are appeared together

```
-------------
```

The above code can be possible using the below code - using **for** loop

```
for row in alist:
    print(row)
```

output

```
[[1, 2], [3, 4]]
[[5, 6, 7], [8, 9, 10], [11, 12, 13]]
```

Note: since it a **for** loop, the 2 rows are appeared one after another

==========

```
#Inserts 15 into the third position in the second list
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11, 12, 13,]] ]
alist[1].insert(2, 15)
print(alist)
```

output

```
[[[1, 2], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [11, 12, 13]]]
```

==========

Another way to use nested for loops. The other way is better but I've needed to use this on occasion:

```
alist = [[[1, 2], [3, 4]], [[5, 6, 7], [8, 9, 10], [11,
12, 13]]]
for row in range(len(alist)):
    print(row)
    for col in range(len(alist[row])):
        print(alist[row][col])
```

output

```
0
[1, 2]
[3, 4]

1
[5, 6, 7]
[8, 9, 10]
[11, 12, 13]
========
```

Using slices in nested list

```
alist = [ [[1, 2, ], [3, 4, ]],   [[5, 6, 7, ], [8, 9, 10], [11,
12, 13,]] ]
print (alist[1] [1:])
```

```
output
[[8, 9, 10], [11, 12, 13]]
=====
```

Section 20.13: Initializing a List to a Fixed Number of Element

Doubt :When can we use/how can we use this concept

For immutable elements (e.g. None, string literals etc.)

```
my_list = [None, ] * 10
print (my_list)
my_list=['test', ] * 10
print((my_list))
```

output

[None, None, None, None, None, None, None, None, None, None]

['test', 'test', 'test', 'test', 'test', 'test', 'test', 'test', 'test', 'test']

Note: Since 'None' is a class None type, it does not have quotes (' ') surround it, where as 'test' is user defined, so it has surrounded with quotes(' ')

=======

def add(self, element: _T) -> None

Add an element to a set.

This has no effect if the element is already present.

```
my_list = [{1}] * 10
print(my_list)
my_list[0].add(2)
print (my_list)
output
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2},
{1, 2}, {1, 2}]
======
```

Instead, to initialize the list with a fixed number of different mutable objects, use:

```
Page 154..Doubt..the below prg and gives the same answer as above
..how does this say it it has DIFFERENT mutable object
my_list = [{1} for _ in range (10)]
print (my_list)
```

output

[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]

Chapter 21: List comprehensions

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

Syntax for general list comprehension (Type 1)

[<expression> for <element> in <iterable>]

<expression> for <element> in <iterable>] ## the names of **expression** and the **element** must be same

Example

```
lst = [2,3,4,5,6,7,8,9,10,11,]
lstCom = [(element + 10)for element in lst] #This is a new list, so we always have to assign to the another llist

print(lstCom)
output
[12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
```

Syntax for list comprehension with if condition (Type 2)

[<expression> for <element> in <iterable> if <condition>]

use the if condition after for loop, if there is no else part in the comprehension

Example

```
lst = [2,3,4,5,6,7,8,9,10,11,]
lstCom = [(element+100)for element in lst if element%2==0]
print(lstCom)
output
[102, 104, 106, 108, 110]
```

Syntax for list comprehension with if ..else condition (Type 3)

If we want to use if and else together in the comprehension, use it before the 'for' loop

else can be used in List comprehension constructs, but be careful regarding the syntax. **The if/else clauses should be used before for loop, not after**

Example

[<expression> if <condtion>**else <conditiion>** for <element> in <iterable>]

```
lst = [2,3,4,5,6,7,8,9,10,11,]
```

```
lstCom = [(element + 20)if element %2 ==0 else element%3
==0 for element in lst]
print(lstCom)
```

output

```
[22, True, 24, False, 26, False, 28, True, 30, False]
--------------
```

The above list comprehension is also possible in generator expression

```
lst = [2,3,12,4, 21,5,6,7,8,9,10,11,]
lstCom =((element + 20)if element %2 ==0 else element%3
==0 for element in lst)
print(lstCom)
print(list(lstCom))
```

output

```
<generator object <genexpr> at 0x0000018C642EE148>
[22, True, 32, 24, True, False, 26, False, 28, True, 30, False]
-----------
```

Each <element> in the <iterable> is plugged in to the <expression> if the (optional) <condition> evaluates

to true . **All results are returned at once in the new list.**

Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length

The for expression sets x to each value in turn from (1, 2, 3, 4). The result of the expression x * x is appended to an **internal list.** The internal list is assigned to the variable my_list when completed.

Besides a speed increase (as explained here), a list comprehension is roughly equivalent to the following for-loop:

```
squares = [ ]
for x in (1, 2, 3, 4, 5):
    squares.append(x*x)
    print (x)
print (squares)
```

output
1
2
3
4
5
[1, 4, 9, 16, 25]
============
The expression applied to each element can be as complex as needed:

```
# Get a list of uppercase characters from a string
my_list = ["Hello World"]
my_list=[((s.upper()) for s in my_list]
print (my_list)
```

output

```
['HELLO WORLD']
```

Note: But this is not that we intented. It happens because we did not give the iterable inside comprehension. See the next program

---

```
my_list=[s.upper() for s in "Hello world"]
print (my_list)
```

```
output
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D'] # the
reson, the iterable is given inside the comprehension . this is
RIGHT way of doing
=========================
We can pass parameter to a function using list
comprehension
def calculateSalary(a):
    return 1000*a

my_list =  [10,20]
my_list = [calculateSalary(i) for i in my_list] # ##
the names of expression (word) and the element must be
same (word)
print(my_list)

======
# Strip off any commas from the end of strings in a list
```

```
my_list =  ['these,', 'words,,,', 'mostly', 'have, commas,']
my_list = [word.strip(',') for word in my_list] # ## the names of
expression (word) and the element must be same (word)
print(my_list)
output
['these', 'words', 'mostly', 'have, commas']

Note: in the list an item('words',,,) has three commas. The
strip(), remove 2 commas and keep one comma, because the one comma
is mandataory requirement for the sepearation of the each item in
the list. If an item has only one comma, that CAN NOT be removed..
that comma is for item separation, so it can not be removed
If we don't give any separator value inside the strip(), it prints
the list as it is. See below

my_list =  ['these,', 'words,,,', 'mostly', 'have, commas,']
my_list = [word.strip('') for word in my_list] # strip function
does not have any strip value to stripped
print(my_list)
output
['these,', 'words,,,', 'mostly', 'have, commas,']
==========
```

```
# Organize letters in words more reasonably - in an alphabetical
orde
def split(self,
          sep: Optional[str] = ...,
          maxsplit: int = ...) → List[str]
Return a list of the words in the string, using sep as the
delimiter string.

sep
The delimiter according which to split the string. None (the
default value) means split according to any whitespace, and
discard empty strings from the result.
maxsplit
Maximum number of splits to do. -1 (the default value) means no
limit.

=============
```

Practice the below individual programs before going to next pgm

```
sentence = "Beautiful"
b = list(sentence)
print(b)
output
['B', 'e', 'a', 'u', 't', 'i', 'f', 'u', 'l']
============
sentence = "Beautiful is better than ugly"
sentence =[word for word in sentence.split()]
print(sentence)
# output is : ['Beautiful', 'is', 'better', 'than',
'ugly']

# ===============

# sentence = "Beautiful is better than ugly"
sentence = ['Beautiful', 'is', 'better', 'than', 'ugly']
sentence =" ".join(sentence)
print(sentence)
```

```
print(type(sentence))
# output: Beautiful is better than ugly
# join() joins the list items into one string

# ============
Fisrt learn lambda , Sorted function then execute the
below function
sentence = "Beautiful is better than ugly"
sentence =[(sorted(word, key=lambda x: x.lower()))  for
word in sentence.split()]
# sentence =[(sorted(word, key=lambda x: x))  for word in
sentence.split()]
print(sentence)
# output is below
# [['a', 'B', 'e', 'f', 'i', 'l', 't', 'u', 'u'], ['i',
's'], ['b', 'e', 'e', 'r', 't', 't'], ['a', 'h', 'n',
't'], ['g', 'l', 'u', 'y']]
# ============= split() gives 5 number lists of list
```

```python
a = "Hello World"
# Sorted() gives the whitespace,  numbers,  uppercase ,
lowercase in this order
b=(sorted(a))
print(b)
print(type(b)) # sorted() gives a list output / and list
does not have lower()
c = (("" .join(b)))
# =============

sentence = "Beautiful is better than ugly"
sentence =[("".join(sorted(word, key=lambda x:
x.lower())))  for word in sentence.split()]
# sentence =[(sorted(word, key=lambda x: x))  for word in
sentence.split()]
print(sentence)
# ==============
sentence = "Beautiful is better than ugly"
sentence =[word for word in sentence.split()]
```

```
# sentence =[(sorted(word, key=lambda x: x))  for word in
sentence.split()]
print(sentence)
# output: ['Beautiful', 'is', 'better', 'than', 'ugly']
print(" ".join(sentence)) # join(), again join all the
list items to a string
# outout: Beautiful is better than ugly

==============

sentence = "Beautiful is better than ugly"
# sentence=["".join(sorted(word, key = lambda x: x.lower())) for
word in sentence.split()]
sentence=["".join(sorted(word, key = lambda x:x.lower())) for word
in sentence.split()]
print(sentence)
''' IT SPLITS THE SENTENCE INTO INDIVIDUAL WORDS,
THEN THE INDIVIDUAL WORDS ARE SORTED IN ALPHAPTICAL ORDER
THEN JOIN THE INDIVIDUAL LIST ITEMS TO STRING'''
OUTPUT
['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

Doubt: check how the join() and the lambda functions are working

==========

Check if a character is available in a string

```
a= 'apple'
lst = list(a)
b = (lst)
print(b)

for char in b:
    if char in ['a','e', 'i', 'o', 'u']:
        print("Yes")
    else:
        print("No")
```

```
output
['a', 'p', 'p', 'l', 'e']
Yes
No
No
No
```

Yes

======

Find a non-vowles from a string and replace with *

```
str = "apple"
lst1 = list(str)
print(lst1)
emptyList = []
for char in lst1:
    if char in ['a','e', 'i', 'o', 'u']:
        emptyList.append(char)
        # print(emptyList)
    else:
        emptyList.append('*')
print(emptyList)
output
['a', 'p', 'p', 'l', 'e']
['a', '*', '*', '*', 'e']

# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']
```

```
[x for x in 'apple' if  x in aeiou else '*']
Output
    [x for x in 'apple' if  x in aeiou else '*']
                                            ^
SyntaxError: invalid syntax
========
replaceVowel = [X if X in 'aeiou' else '*'  for X in 'apple'] #
the names of expression and the element must be same. Here the
expression is
  # X if X in 'aeiou' else '*'  , element is X
print (replaceVowel)
```

output

['a', '*', '*', '*', 'e']

IMPORTANT:Note this uses a different language construct, a conditional expression, which itself is not part of the comprehension syntax. Whereas the **if after the for...in is a part of list comprehensions** and used to filter elements from the source iterable.

```
=========
lst = ['a', 'p', 'p', 'l', 'e']
vowels = ['a', 'e', 'i', 'o', 'u']
listCom = [(element) if element in vowels else '*' for element in
```

```
lst]
print(listCom)
```
ouput
['a', '*', '*', '*', 'e']
==========

Double Iteration

Order of double iteration [... for x in ... for y in ...] is either natural or counter-intuitive. The rule of thumb is to follow an equivalent for loop

```
def foo (i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```
output this prgm is not working page 156
SyntaxError: 'yield' outside function
Reason: for loop has yield , that is not acceptatble

==========

The above code can be re written in order to get the result – see below

```
def foo (i):
    return i, i + 10

for i in range(3):
    for x in foo(i):
        print (str(x))
output
0
10
1
11
2
12
==========
```

the above code can be written in one line as compreshension as below

```
def foo (i):
    return i, i + 10
```

```
listCom = [str (x) for i in range(3) for x in foo(i)
]
print(listCom)
```

output

['0', '10', '1', '11', '2', '12']

Note: observe the str(x) in the comprehension, if we removed the str. It gives numerical list [0, 10, 1, 11, 2, 12]

==========

In-place Mutation and Other Side Effect

Before using list comprehension, understand the difference between functions called for their side effects (mutating, or in-place functions) which usually **return None**, and functions that return an interesting value.

Many functions (especially pure functions) simply take an object and **return some object**. An in-place function modifies the existing object, which is called a side effect. Other examples include input and output operations such as printing.

Example for **in-place** function (returns None)

```
lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]
a  = lst.sort()
print(a)
```

output

None

==========

The above program can be written using comprehension

```
my_list = [lstA.sort() for lstA in [ [2, 1, ], [4, 3, ], [0, 1, ] ]]
print (my_list)
```

output

[None, None, None]

Doubt write this prgrram using for loop to get the same result

---

Instead of getting the out None, if we want this list to be sorted , use the below pgm

```
lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]
mylist = [a.sort() for a in lst]
print(mylist)
print(lst)
```

output

[None, None, None]
[[1, 2], [3, 4], [0, 1]]

========

sort() // sort () is an example for an **in-place** function, because it modifies content of the list, but gives None, where a **pure** function mofify the content but gives output

the above program can be written as below using inside a function (functions that return an interesting value.

```
def method1(lstA):
    lstA.sort()
    print ("after sort The values are :", lstA.sort()) # if we print here, it gives the EXPECTED result , is None
    return lstA

lstB = [ [2, 1, ], [4, 3, ], [0, 1, ] ]
a= method1(lstB) # calling the function method1 and  and stores returns value. It supposed to be the value after applying the sort(), that is None
print (a)  # but we get the list it self, which is unexpected
```

output

after sort The values are : None
[[0, 1], [2, 1], [4, 3]]

```
-------------
def m(lst):
    return lst.sort()

lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]
a = m(lst)
print(a)
```

output

None

========

Example for pure function (returns some object)

sorted() returns a sorted list rather than sorting in-place:

Sorted ()

```
lst = [ [2, 1, ], [4, 3, ], [0, 1, ] ]
a= sorted(lst)
print (a)
```

output

[[0, 1], [2, 1], [4, 3]]

==========

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3

```
[print (x) for x in (1, 2, 3,)]
```

Instead

```
for x in (1,2,3):
    print (x)
```

Both give same output, still use of the for loop is preferable

=======

In some situations, side effect functions (in-place) are suitable for list comprehension. random.randrange() has the side effect of changing the state of the random number generator, but it also returns an interesting value.

Additionally, next() can be called on an iterator

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

---

random. randrange (stop)

random. randrange (start, stop [, step])

Return a randomly selected element from range(start, stop, step). This is equivalent to choice(range(start, stop, step)), but doesn't actually build a range object.

The positional argument pattern matches that of range(). Keyword arguments should not be used because the function may use them in unexpected ways

---

The following random value generator is not pure fn, yet makes sense as the random generator is reset every time the expression is evaluated:

```
a= [random.randrange (1, 7) for _ in range(10)] # DOUBT : What does the _ (hypen) do, it is iteration variable
print (a)
```

output
[2, 3, 2, 5, 4, 6, 1, 5, 2, 1] # it prints 10 randam numbers between 1 and 7
NOTE: randrange(), includes start and excludes the stop

DOUBT 1: This program gives error if we write the same in for loop
---

```
a= [random.randint (1, 7) for _ in range(10)] #  : What does the _ (hypen) do
print (a)
```

output
[3, 7, 7, 3, 4, 2, 3, 7, 6, 2]

NOTE: randint(), includes start and the stop

==========
How the step in randomrange works randomrange(start, stop, step)
First it prints the start number, the start number + step, then it keeps add the step number

Here first it prints 1
Then it adds 1 + 3 (step numner) = 4
Then 4 + 3 step number = 7
And so on..

```
from random import randrange
a= [randrange (1,10,3) for _ in range(10)]
print (a)
```

output

```
[1, 4, 7, 4, 4, 7, 7, 7, 1, 1]
```

=====

Whitespace in list comprehensions

More complicated list comprehensions can reach an undesired length, or become less readable. Although less common in examples, it is possible to break a list comprehension into multiple lines like so:

Doubt: the below pgm is not understandable – do it again

```
a=[x for x in 'foo' if x not in 'bar']
print (a)
```

output

```
['f', 'o', 'o']
```

For readability / whitespace purpose, the above pgm should have been written as below

```
a=[
   x for x
```

```
    in 'foo'
    if x not in 'bar'
]
print (a)
output
['f', 'o', 'o']
--------------
lst1 = [1,2,3,4]
lst2 = [1,6,7,8]

a = [val for val in lst1 if val  in lst2]
print(a)

a = [val for val in lst1 if val  not in lst2]
print(a)
output
[1]
[2, 3, 4]
==========
```

Section 21.2: Conditional List Comprehensions

Chapter 22: List slicing (selecting parts of lists)

Slicing does not have **from** and **to** index

Section 22.1: Using the third **"step"** argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print (lst[::2])
'''IT PRINTS THE FIRST CHAR THEN 1 + 2 + 2 AND SO ON'''

lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print (lst[::3])
'''IT PRINTS THE FIRST CHAR THEN 1 + 3 + 3 AND SO ON'''
```

OUTPUT

```
['a', 'c', 'e', 'g']
['a', 'd', 'g']
```

=======

Section 22.2: Selecting a sublist from a list

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print(lst[2:3])
print(lst[2:2])
'''IT PRINTS EMPTY [] LIST, THE END INDEX N-1 IS 1, WHICH IS
```

```
LESS THAN THE STARTING INDEX, THAT IS THE END INDEX MUST BE GREATER THAN THE START INDEX'''
print(lst[2:])
'''IT PRINTS FROM INDEX 2 TO THE END'''
print (lst[:4])
'''IT PRINTS FROM 0TH INDEX TO N-1'''
print (lst[:])
'''IT PRINTS ALL'''
```

OUTPUT

```
['c']
[]
['c', 'd', 'e', 'f', 'g', 'h']
['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Section 22.3: Reversing a list with slicing

```
a = [1, 2, 3, 4, 5]
print (a[::-1])
```

output

```
[5, 4, 3, 2, 1]
=======
a = [1, 2, 3, 4, 5]
b=(a[::-1])
a.reverse()
```

```
print (a)

if a == b:
    print (True)
print (b)
output
output
[5, 4, 3, 2, 1]
True
[5, 4, 3, 2, 1]
=========
```

Section 22.4: Shifting a list using slicing

```
def shift_list(array,s):
    s%= len(array)#-7 is divided by 5, output -2
    s*= -1 #output-2 is conveted into +2

    shiftedArray = array[s:] + array[:s]
    # shiftedArray = array[2:] + array[:2]
    return shiftedArray

myArray =[1,2,3,4,5]
a = shift_list(myArray, -7)
print(a)
```

output

[3, 4, 5, 1, 2]

```
------------
def shift_list(array,s):
    s%= len(array)#-7 is divided by 5, output -2
    s*= -1 #output-2 is conveted into +2

    shiftedArray = array[s:] + array[:s]
    # shiftedArray = array[2:] + array[:2]
    return shiftedArray

myArray =[1,2,3,4,5]
a = shift_list(myArray, 5)
print(a)

a = shift_list(myArray, 3)
print(a)
```

output

[1, 2, 3, 4, 5]

[3, 4, 5, 1, 2]

Chapter 141: List destructuring (aka packing and unpacking)

Section 141.1: Destructuring assignment

In assignments, you can split an Iterable into values using the "unpacking" syntax:

```
a, b = (1,2)
print (a)
print (b)
```

output

1
2
===
If you try to unpack more than the length of the iterable, you'll get an error:

```
a,b,c = [1]
output
```

a,b,c = [1]
ValueError: not enough values to unpack (expected 3, got 1)
========

Destructuring as a list

You can unpack a list of unknown length using the following syntax:

```
head, *tail = [1, 2, 3, 4, 5]
print (head)
print (tail)
output
1
[2, 3, 4, 5]
------
The above can be rewritten (Equavelent) as below
lst = [1, 2, 3, 4, 5]
head = lst[0]
tail = lst [1:]
print (head)
print (tail)
```

```
output
1
[2, 3, 4, 5]

======
*head, tail = [1, 2, 3, 4, 5]
print (tail)
print (head)
Output
5
[1, 2, 3, 4]
==========
```

It also works with multiple elements or elements form the end of the list:

```
head, tail, *body = [1, 2, 3, 4, 5]
print (head)
print (tail)
print (body)
```

output

1

2

[3, 4, 5]

=====

The below also possible

```
head, tail, body, *others = [1, 2, 3, 4, 5]
print (head)
```

```
print (tail)
print (body)
print (others)
Output
1
2
3
[4, 5]
====
```

What happens if we give a variable with * and without * to unpack the list
Note: the variable with * will have the ouput value inside the list[]- see the below pgm

```
head, tail, body, *others, a = [1, 2, 3, 4, 5]
print (head)
print (type(head))
print (tail)
print (body)
print (others)
print (type(others))
print (a)
output
1
<class 'int'>
2
3
```

```
[4]
<class 'list'>
5
========
```

Ignoring values in destructuring assignments

If you're only interested in a given value, you can use _ to indicate you aren't interested. Note: this will still set _ , just most people don't use it as a variable.

```
a, _ = [1,2]
print (a)
```

output

1

```
a, _, c = (1, 2, 3)
print (a)
print (c)
```

output

1

3

Note: the _ (underscore means we are not interested on that particular value. But if we print the underscore, still we will get the equavelent value

Ignoring lists in destructuring assignments

Finally, you can ignore many values using the *_ syntax in the assignment:

```
a, *_ = [1, 2, 3, 4, 5]
print (a)
Output
1
--------
a, *_, c = [1, 2, 3, 4, 5]
print (a)
print (c)
Output
1
5
--------
```

What happens if we print the *_ (it gives result but not in the list form / mode)

which is not really interesting, as you could using indexing on the list instead. Where it gets nice is to keep first and last values in one assignment:

```
a, *_, c = [1, 2, 3, 4, 5]
print (a)
print (*_)
print (c)
output
1
2 3 4
5
```

```
========
```
or extract several values at once:
```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5]
print (a, b, c)
```
output
```
1 3 5
```

Section 141.2: Packing function arguments

In functions, you can define a number of mandatory arguments:

```
def fun1 (arg1, arg2, arg3):
    return (arg1, arg2, arg3)
```

which will make the function callable only when the three arguments are given:

```
fun1 (1,2,3)
--------
```
and you can define the arguments as optional, by using default values:

```
def fun2 (arg1='a', arg2 = 'b', arg3 ='c'):
    return (arg1, arg2, arg3)
```

so you can call the function in many different ways, like:

```
a=fun2 (1)
print (a)
```

```
a = fun2(1, 2)
print (a)

a=fun2 (arg2 =2, arg3 = 3)
print (a)
```
output
```
(1, 'b', 'c')
(1, 2, 'c')
('a', 2, 3)
```

But you can also use the destructuring syntax to pack arguments up, so you can assign variables using a list or a dict.
=====

Packing a list of arguments

Consider you have a list of values

```
def fun1 (arg1, arg2, arg3):
    return arg1, arg2, arg3

lst = [1, 2, 3]
```

You can call the function with the list of values as an argument using the * syntax:

```
a = fun1(*lst)
print (a)

a=fun1 (*['w', 't', 'f'])
print (a)
output
(1, 2, 3)
('w', 't', 'f')
======
```

But if you do not provide a list which length matches the number of arguments:

```
a=fun1 (*['oops'])
output
a=fun1 (*['oops'])
TypeError: fun1() missing 2 required positional arguments: 'arg2'
and 'arg3'
======================
```

```
def fun1 (*arg1):
    for i in arg1:
        print(i)

fun1 (1,2,3,4,5)
```

```
Output:
    1
2
3
4
5
```

Packing keyword arguments (**)

Now, you can also pack arguments using a dictionary. You can use the ** operator to tell Python to unpack the dict as parameter values:

```
def fun1 (arg1, arg2, arg3):
    return (arg1, arg2, arg3)

d = {'arg1' :1, 'arg2' : 2, 'arg3' : 3}

a = fun1(**d)
print (a)
```

output

(1, 2, 3)

Note: `a = fun1(**d), brings the VALUE in the dictionary (2 stars)`

If we give `a = fun1(*d), it gives to the key only- see below the output(1 star)`

```
('arg1', 'arg2', 'arg3')
==========
```

when the function only has positional arguments (the ones without default values) you need the dictionary to be contain of all the expected parameters, and have **no extra parameter**, or you'll get an error:

```
def fun1 (arg1, arg2):
    return (arg1, arg2)

d = {'arg1' :1, 'arg2' : 2, 'arg3' : 3}
# d = {'arg1' :1, 'arg2' : 2} #this is correct number of arguments suppose to send
a = fun1(**d)
print (a)
ouput
a = fun1(**d)
TypeError: fun1() got an unexpected keyword argument 'arg3'
---
If we pass extra arguments that too fail - see the pgm
def fun1 (arg1, arg2,arg3):
    return (arg1, arg2, arg3)

d = {'arg1' :1, 'arg2' : 2, 'arg3' : 3, 'arg4': 4}# we send 4 arguments instead of 3
```

```
a = fun1(**d)
print (a)
output
a = fun1(**d)
TypeError: fun1() got an unexpected keyword argument 'arg4'
==========
```

But there you can omit values, as they will be replaced with the defaults:

```
def fun2 (arg1='a', arg2 = 'b', arg3 ='c'):
    return (arg1, arg2, arg3)

a= fun2 (**{'arg1' : 9})
print (a)
output
(9, 'b', 'c')
==========
```

In real world usage, **functions can have both positional and optional arguments**, and it works the same:

```
def fun3 (arg1, arg2 = 'b', arg3 ='c'):
    return (arg1, arg2, arg3)

#you can call the function with just an iterable:
a= fun3 (*[1])
print (a)
# Returns: (1, 'b', 'c')
```

```
a=fun3 (*[1,2,3])
print (a)
# Returns: (1, 2, 3)

# or with just a dictionary:
a=fun3(**{'arg1' :1})# ** gives value, * gives keys
print (a)
# Returns: (1, b, c)

#or you can use both in the same call:
a = fun3(*[1,2], **{'arg3' : 45})
print (a)
output
(1, 'b', 'c')
(1, 2, 3)
(1, 'b', 'c')
(1, 2, 45)
```

Section 141.3: Unpacking function arguments

When you want to create a function that can accept any number of arguments, and not enforce the position or the name of the argument at "compile" time, it's possible and here's how:

```
def fun1 (*args, **kwargs):
    print (args, kwargs)
```

The *args and **kwargs parameters are special parameters that are set to a tuple and a dict, respectively:

```
a=fun1 (1,2,3)
print (a)
# Prints: (1, 2, 3) {}

a=fun1 (a=1, b=2, c=3)
print (a)
# Prints: () {'a': 1, 'b': 2, 'c': 3}

a=fun1('x', 'y', 'z', a=1, b=2, c=3)
print (a)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

Output

```
(1, 2, 3) {}
None
() {'a': 1, 'b': 2, 'c': 3}
None
('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
None
=======
```

If you look at enough Python code, you'll quickly discover that it is widely being used when passing arguments over to another function. For example if you want to extend the string class:

Note: Doubt: what does this code do? //not able to understand/ revisit

```
class MyString(str):
    def __init__(self, *args, **kwargs):
        print ('Constructing MyString')
        super(MyString, self).__init__(*args, *kwargs)

a = MyString()
```

output
Constructing MyString