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.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate 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:

The way that data is organized has a significant impact on how effectively it can be used. One of the most obvious and useful ways to organize data is as a list.

Let us list out the characteristics of a list

1. **It is a data structure**
2. **It has 0 or more elements**
3. **There is no name for each element**
4. **The elements are accessed by using index or subscript**
5. **The index starts from 0**
6. **The size of the list is not fixed. The list can grow or shrink. We can find the number of elements in a list at any point in time.**
7. **The elements of the list need not be of the same type. The list can be heterogeneous.**
8. **The type list supports a number of builtin functions for playing with the list**
9. **This is for those who know 'C'. It is not a linked list**

We shall discuss each of them with some simple examples.

1. It is a data structure

A **list** is a *linear data structure*, meaning that its elements have a linear ordering.

A data structure has information arranged in some fashion. In case of a list, the elements are put into different slots and these slots are numbered from 0 onwards. It is somewhat similar to row houses with house numbers.

2. It has 0 or more elements.

One of the ways of constructing a list is by enumerating the elements within square brackets and separating the elements by comma.

```
a = [1, 3, 5, 7]
b = []
```

The list a has 4 elements and the list b is empty.

3. There is no name for each element

4. The elements are accessed by using index or subscript

5. The index starts from 0

```
c = [ "india", "srilanka", "bangladesh", "nepal" ]
```

In analytical geometry, we refer to all the x co-ordinates as x and y co-ordinates as y.

We talk about points `(x1, y1), (x2, y2)` and so on.

Lists are similar.

`c[1] is Srilanka and c[3] is nepal.`

1 and 3 are called indices or subscripts. The index starts from 0.

`So, india is c[0] and not c[1].`

6. The size of the list is not fixed. The list can grow or shrink. We can find the number of elements in a list at any point in time.

```
d = [10, 20, 30, 40]
print(len(d)) # 4
```

We can use the function `len` to find the number of elements in the list.

Modify the list:

There are number of ways to change the number of elements in the list.

example:

```
d.pop() # would remove the last elemet
print(d)  # [10, 20, 30]
d.append(50) # would add an element at the end
print(d) # [10, 20, 30, 50]
```

7. The elements of the list need not be of the same type. The list can be heterogeneous.

```
e = ["ramnujan", 1729, "pi", 3.14, True]
f = [[1, 1], [2, "four"]]
```

In the list e, we have a string, then an integer and then a string, then a float and then a boolean value.

We can also have a list of lists as the next list f shows.

8. There are number of functions to play with the list

Please check `help(list)` in the interactive session.

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

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 item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Create a list

```
>>> a = [11, 22, 33, 44, 55]
```

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

Example

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

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

Display the list

```
>>> a
[11, 22, 33, 44, 55]
```

Find the length

```
>>> len(a)
5
```

Find the 0th element

```
>>> a[0]
11
```

Find the last element

This is wrong. The index of the last element is length - 1.

This also shows that if we index beyond the size of the list, Python gives a runtime error.

```
>>> a[len(a)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> a[len(a) - 1]
55
```

Add an element at the end

```
>>> a.append(66)
>>> a
[11, 22, 33, 44, 55, 66]
```

Add at a particular position

```
>>> a.insert(2, 77)
>>> a
[11, 22, 77, 33, 44, 55, 66]
```

Remove at the end

pop returns the element being removed

what happens if we pop an empty list?

```
>>> a.pop()
66
>>> a
[11, 22, 77, 33, 44, 55]
```

Remove at a particular index

```
>>> a.pop(2)
77
>>> a
[11, 22, 33, 44, 55]
```

Remove based on the value

remove does not return any value.

What if that value does not exist in the list?

```
>>> a.remove(33)
>>> a
[11, 22, 44, 55]
```

Check whether an element exists

```
>>> 22 in a
True
>>> 33 in a
False
```

Find the position of the element.

What if the value does not exist?

```
>>> a.index(55)
3
```

Count the number of occurrences of an element

```
>>> b = [11, 22, 11, 33, 11]
>>> b.count(11)
3
>>> b.count(22)
1
>>> b.count(44)
0
```

Arrange the elements in order

```
>>> c = [33, 11, 55, 44, 22]
>>> c
[33, 11, 55, 44, 22]
>>> c.sort()
```

```
>>> c
[11, 22, 33, 44, 55]
```

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()`.

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

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

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

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

```
fruitlist = ["apple", "banana", "cherry"]
mylist = list(fruitlist)
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.

```
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:

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:

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)
```

Why a list?

Let us say we have to find the sum of 3 numbers. We can read into three variables say a, b and c. Then add them.

What if we have to find the sum of 1000 numbers? We can also do this by reading 1000 times within a loop to the same variable and update the sum.

What if we have to arrange given 1000 numbers in order?. Then all the numbers are required. It is humanly impossible to write a program with 1000 variables and come with an algorithm to arrange them in order.

We can use a list in these cases. A single variable of list type can hold all these 1000 values. We can access each value by indexing.

How to access all the elements of a list?

We have to walk through or traverse. There are three ways depending on our intent.

`case 1:` **We want to visit every element of the list without modifying the elements of the list.**

case a: use a **for loop** on the list.

The data structure list is an **iterable**. It knows how to start the traversal process, it knows how to give us a copy of each element and it also knows to signal the end of the list.

To find the sum of all the elements of a list, we have to traverse the whole list and access each element without modifying them.

```
# file: 1_list_traversal.py
# find the sum of all the elements of the list
a = [11, 33, 22, 55, 44]
total = 0
for elem in a : # note: list is an iterable
      total = total + elem
```

```
print("total : ", total)
```

```
$ python 1_list_traversal.py
total :  165
```

case 2: We may want to visit a few elements of the list until some condition is satisfied.

- **In such cases, we use a while loop based on some condition (predicate).**

In this example, we are trying to **find the leftmost even number if any**.

```
# file: 2_list_traversal.py
# find the leftmost even number if any
# select one of these lists; comment the other
#a = [11, 33, 22, 55, 44]
a = [11, 33, 55, 77]
found = False
i = 0
while not found and (i < len(a)) :
      if a[i] % 2 == 0:
            found = True
      else:
            i += 1
if found:
      print("found an even number : ", a[i])
else:
      print("no even number found")
```

This is the case when

```
a = [11, 33, 22, 55, 44]
$ python 2_list_traversal.py
found an even number :  22
```

This is the case when

```
a = [11, 33, 55, 77]
$ python 2_list_traversal.py
no even number found
```

Think:

```
while not found and (i < len(a)) :
```

Can we interchange the operands of not? Give reasons.

Here, i is incremented in else part. What if we increment i any way?

```
if a[i] % 2 == 0:
    found = True
else:
    i += 1
```

Why should we not use for loop to iterate over all the elements of the list?

case 3: Let us revisit the **case a:** with a small change. We want to add a number to each element of the list. It is like adding grace marks to marks of every student.

```
# file: 3_list_traversal.py
# WRONG; does not work
# add 100 to each element of the list
a = [11, 33, 22, 55, 44]
print("before loop : ", a)
for elem in a :
    elem += 100
print("after loop : ", a)

$ python 3_list_traversal.py
before loop :  [11, 33, 22, 55, 44]
after loop :  [11, 33, 22, 55, 44]
```

This does not work as the loop variable ***e*** gets a copy of the element of the list.
Python works based on a concept called "always by value". We end up changing the copy of an element of the list and not the list itself.

If our requirement is to change the element of the list, we should use indexing.

If our requirement is to modify all the elements of the list, then we can iterate through all the possible indices - we can generate all the possible indices by using range function. So, this is the solution.

```
# file: 4_list_traversal.py
# add 100 to each element of the list
a = [11, 33, 22, 55, 44]
print("before loop : ", a)
for i in range(len(a)) :
    a[i] += 100
print("after loop : ", a)

$ python 4_list_traversal.py
```

```
before loop :  [11, 33, 22, 55, 44]
after loop :  [111, 133, 122, 155, 144]
```

We conclude there are three ways of walking through a list.

1. **Access all the elements.**

   Use a for loop on the list as the iterable.

2. **Modify all the elements.**

   Use a for loop on possible indices using range on number of elements of the list

3. **Stop traversal in the middle.**

   Use a while loop.

We have seen examples of traversing a list. Let us see examples of how to build the list.

Example 1:

Create a list of squares of numbers from 1 to n.

Make an empty list. Traverse through numbers from 1 to n by using a for loop.

Each time find the square of the number and append to the list.

```
# file: 5_list_build.py
# create a list of squares of numbers from 1 to n
sq_list = []
n = int(input("enter a number : "))
for i in range(1, n + 1):
      sq_list.append(i * i)
for sq in sq_list:
      print(sq, end = " ")
print()
```

```
$ python 5_list_build.py
enter a number : 5
1 4 9 16 25
```

Example 2:

Find partnership for each wicket given the fall of wickets.

Basically we have to find the differences between successive elements.

We require to access two elements of the list simultaneously.

We may use a loop based on range of indices or list slice (next section).

```
# file : 6_list_build.py
# find the difference between successive elements of a list
fall_of_wickets = [10, 50, 100, 145, 150, 175]
n = len(fall_of_wickets)
partnership = []
# do n - 1 times
for i in range(n - 1) :
	partnership.append(fall_of_wickets[i + 1] - fall_of_wickets[i])
print("fall of wickets ")
for fall in fall_of_wickets:
	print(fall, end = " ")
print()
print("partnerships ")
for score in partnership:
	print(score, end = " ")
print()

$ python 6_list_build.py
fall of wickets
10 50 100 145 150 175
partnerships
40 50 45 5 25
```

Example 3:

Create a list of common elements - like an intersection of sets.

There are two ways of doing this.

1. Sort the elements and then find the common elements.
2. Directly compare the elements of the first list and check whether it occurs in the second.

We will try the second one in this example. We leave it to you to try the first method.

```
# file : 7_list_common_elements.py
# find common elements
a = [11, 33, 22, 44, 55]
b = [33, 60, 11, 70, 80, 44]
res = []
# walk through the first list
for i in a :
	# check whether that element also occurs in the second list
	if i in b :
		res.append(i)
for elem in res :
	print(elem, end = " ")
print()
```

```
$ python 7_list_common_elements.py
11 33 44
```