

Unit 4 – List - Tuples

Objectives:

- ❖ Students will learn the sequence data type objects in python like lists, tuple and dictionaries
- ❖ Students can understand the difference between mutable and immutable objects
- ❖ Students will learn how to access elements from the sequence type
- ❖ Students will be dealing with list operators and functions
- ❖ Students will be able to know about tuple operations like slicing and some built-in methods
- ❖ Students can learn how to create a dictionary and implementation of all the sequence types

Module 1 - List

4.1 Definition

Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data types used in Python. Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets and are separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list can also have another list as an item. This is called a nested list. It can have any number of items and they may be of different types (integer, float, string etc.).

4.2 How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

4.3 How to access elements from a list?

There are various ways in which we can access the elements of a list.

6.3.1 List Index: We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4. Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

- Nested lists are accessed using nested indexing.

Output

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

The diagram shows a string 'prebe' with a length of 5. Below the string, two rows of indices are provided. The first row, labeled 'index', shows indices 0 through 4 corresponding to characters 'p', 'r', 'e', 'b', 'e'. The second row, labeled 'negative index', shows indices -5 through -1 corresponding to the same characters.

	length = 5				
	'p'	'r'	'e'	'b'	'e'
index	0	1	2	3	4
negative index	-5	-4	-3	-2	-1

e
p

Dept. of IT – RK Valley

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index “maps to” one of the elements.

List indices work the same way as string indices:

4.5 Traversing a list

The most common way to traverse the elements of a list is with a *for* loop. The syntax is the same as for strings:

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
for cheese in cheeses:
    print (cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions *range* and *len*:

```
numbers = [17, 123]
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. *len* returns the number of elements in the list. *range* returns a list of indices from 0 to $n-1$, where n is the length of the list. Each time through the loop, *i* gets the index of the next element. The assignment statement in the body uses *i* to read the old value of the element and to assign the new value.

A *for* loop over an empty list never executes the body:

```
for x in empty:
    print ('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

4.6 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print (c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

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

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

List slices: The slice operator also works on lists

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print (t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Syntax: List[start:end:step]

start –starting point of the index value end

stop –ending point of the index value

step-increment of the index values

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3:1]
['b', 'c']
```

4.7 List methods

Adding Elements into a list

6.7.1 list.append()

Python provides methods that operate on lists. For example, *append* adds a new element to the end of a list

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print (t)
['a', 'b', 'c', 'd']
```

4.7.2 list.extend()

extend takes a list as an argument and appends all of the elements

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print (t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

4.7.3 list.insert(i, x)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

Example 1: Inserting an Element to the List

```
# vowel list
vowel = ['a', 'e', 'i', 'u']

# 'o' is inserted at index 3
# the position of 'o' will be 4th
vowel.insert(3, 'o')

print('Updated List:', vowel)
```

Deleting elements

4.7.4 list.remove(x)

Remove the first item from the list whose value is equal to x. It raises a ValueError if there is no such item.

4.7.5 list.clear()

Remove all items from the list. Equivalent to `del a[:]`.

Example 1: Working of `clear()` method

```
# Defining a list
list = [{1, 2}, ('a'), ['1.1', '2.2']]

# clearing the list
list.clear()

print('List:', list)
```

4.7.6 `list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print (t)
['a', 'c']
>>> print (x)
b
```

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print (t)
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print (t)
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print (t)
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

4.7.6 `list.index(x[, start[, end]])`

Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item.

The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.

Example:

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'u']

# index of 'e' in vowels
```

```

index = vowels.index('e')
print('The index of e:', index)

# element 'i' is searched
# index of the first 'i' is returned
index = vowels.index('i')

print('The index of i:', index)

```

Example: Working of index() With Start and End Parameters

```

# alphabets list
alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']

# index of 'i' in alphabets
index = alphabets.index('e')    # 2
print('The index of e:', index)

# 'i' after the 4th index is searched
index = alphabets.index('i', 4)    # 6
print('The index of i:', index)

# 'i' between 3rd and 5th index is searched
index = alphabets.index('i', 3, 5)    # Error!
print('The index of i:', index)

```

4.7.7 list.count(x)

Return the number of times *x* appears in the list.

Example:

```

# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# count element 'i'
count = vowels.count('i')
# print count
print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')
# print count
print('The count of p is:', count)

```

4.7.8 list.sort(key=None, reverse=False)

Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation). Most list methods are void; they modify the list and return None. If you accidentally write `list= list.sort()`, you will be disappointed with the result.

Example : Sort a given list

```

# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort()
# print vowels
print('Sorted list:', vowels)

```

Example : Sort the list in Descending order

```

# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']
# sort the vowels
vowels.sort(reverse=True)
# print vowels
print('Sorted list (in Descending):', vowels)

```

4.7.9 list.reverse()

Reverse the elements of the list in place.

```
# Operating System List
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)
# List Reverse
systems.reverse()
# updated list
print('Updated List:', systems)
```

4.7.10 list.copy()

Return a shallow copy of the list. Equivalent to a[:]

```
# mixed list
my_list = ['cat', 0, 6.7]
# copying a list
new_list = my_list.copy()
print('Copied List:', new_list)
```

4.8 List Comprehension

List comprehension is an elegant and concise way to create a new list from an existing list in Python. A list comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

List Membership Test:

We can test if an item exists in a list or not, using the keyword `in`.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)
```

Output

```
True
False
True
```

4.9 Built-in functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

The sum() function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

Example:

```
57 | #A PROGRAMME TO CREATE A LIST WITH N ELEMENTS and find its average, min & max value
    | n=int(input("Enter Length of List: "))
    | l=[]
```

Output:

```
Enter length of list: 6
Enter a value: 2
Enter a value: 5
Enter a value: 8
Enter a value: 3
Enter a value: 9
Enter a value: 7
average of given elements is: 5.666666666666667
Maximum value of given list is: 9.0
Minimum value of given list is: 2.0
```

Example:2

```
#A program to remove duplicate elements from a given list
l=input("Enter list of elements without comma between: ")
k=len(l)
l1=[]
for i in l:
    if i not in l1:
        l1.append(i)
print ("updated list: ",l1)
```

Output:1

```
Enter list of elements without comma between: 1234516276543
updated list: ['1', '2', '3', '4', '5', '6', '7']
```


Output:2

```
Enter list of elements without comma between: RGUKT RK Valley
updated list: ['R', 'G', 'U', 'K', 'T', ' ', 'V', 'a', 'l', 'e', 'y']
```

Example:3

```
#a program to find sum of elements of a given list
li=[2,3,4,5,8,1,7]
le=len(li)
i=0
sum=0
while(i<le):
    sum=sum+li[i]
    i=i+1
print("The sum of the elements:%d"%sum)
```

Output:

```
The sum of the elements:30
```

Module 2 – Tuples

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of *namedtuples*). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma. For example:

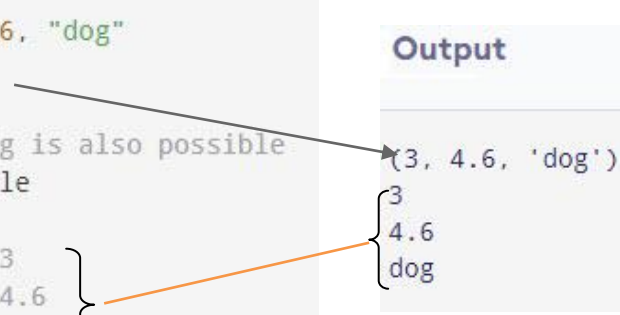
```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)    # 3
print(b)    # 4.6
print(c)    # dog
```



4.10 Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

Indexing:

We can use the index operator [] to access an item in a tuple, where the index starts from 0. So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range (6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])    # 'p'
print(my_tuple[5])    # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])    # 's'
print(n_tuple[1][1])    # 4
```

Output

p
t
s
4

Negative Indexing:

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing for accessing tuple elements
my_tuple = ('p','e','r','m','i','t')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

Output

t
p

Slicing:

We can access a range of items in a tuple by using the slicing operator *colon*:

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r','o', 'g')
print(my_tuple[:7])

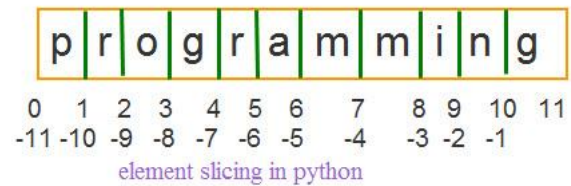
# elements 8th to end
# Output: ('m','i', 'n','g')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'n','g')
print(my_tuple[:])
```

Output

```
('r', 'o', 'g')
('p', 'r', 'o', 'g')
('m', 'i', 'n', 'g')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'n', 'g')
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



4.11 Changing a Tuple

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
>>> t=(5,46,34)
>>> t[0]=43
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    t[0]=43
TypeError: 'tuple' object does not support item assignment
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple. Deleting a tuple entirely, however, is possible using the keyword *del*.

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','m','i','n','g')

#deleting entire tuple using del keyword
del my_tuple

#trying to print the deleted tuple,
#it will cause an error because tuple has already deleted
print(my_tuple)
```

When you run the above code output will be:

```
Traceback (most recent call last):
  File "C:/Users/VJ/Desktop/jaffa.py", line 9, in <module>
    print(my_tuple)
NameError: name 'my_tuple' is not defined
```

4.12 Tuple operations

We can use + operator to combine two tuples. This is called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the * operator.

Both + and * operations result in a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Output

```
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword *in*.

```
# Membership test in tuple
my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation
print('a' in my_tuple)
print('b' in my_tuple)

# Not in operation
print('g' not in my_tuple)
```

Output

```
True
False
True
```

4.13 Iterating Through a Tuple

We can use a *for* loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
    print("Hello", name)
```

Output

```
Hello John
Hello Kate
```

4.14 Built-in functions

len()- to determine how many elements in the tuple

tuple()-a function to make a tuple

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 where it was found

min()-Returns the minimum value in a tuple

max()-Returns the maximum value in a tuple

4.15 Advantages of Tuple over List

- Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Example:1

```
#to find repeated elements in a tuple
t1=(1,3,5,2,1,6,5,4,8)
t2=()
t3=()
for i in t1:
    if i not in t2:
        t2=t2+(i,)
    else:
        t3=t3+(i,)
print("The repeated values are:",t3)
```

Output:

```
The repeated values are: (1, 5)
```

Example:2

```
#Python Program to Create a List of Tuples with the
#First Element as the Number and Second Element as the Square of the Number
l_range=int(input("Enter the Lower range:"))
u_range=int(input("Enter the upper range:"))
a=[(x,x**2) for x in range(l_range,u_range+1)]
print(a)
```

Output:

```
Enter the lower range:1
```

```
Enter the upper range:10
```

```
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100)]
```

Multiple Choice Questions

1. What will be the output of the following Python code?

```
>>>names = ['Amir', 'Bear', 'Charlton', 'Daman']  
>>>print(names[-1][-1])
```

 - a) A
 - b) Daman
 - c) Error
 - d) n**
2. Suppose list1 is [1, 3, 2], What is list1 * 2?
 - a) [2, 6, 4]
 - b) [1, 3, 2, 1, 3]
 - c) [1, 3, 2, 1, 3, 2]**
 - d) [1, 3, 2, 3, 2, 1]
3. Suppose list1 = [0.5 * x for x in range(0, 4)], list1 is:
 - a) [0, 1, 2, 3]
 - b) [0, 1, 2, 3, 4]
 - c) [0.0, 0.5, 1.0, 1.5]**
 - d) [0.0, 0.5, 1.0, 1.5, 2.0]
4. To insert 5 to the third position in list1, we use which command?
 - a) list1.insert(3, 5)
 - b) list1.insert(2, 5)**
 - c) list1.add(3, 5)
 - d) list1.append(3, 5)
5. Suppose t = (1, 2, 4, 3), which of the following is incorrect?
 - a) print(t[3])
 - b) t[3] = 45**
 - c) print(max(t))
 - d) print(len(t))
6. What will be the output of the following Python code?

```
>>>t = (1, 2)  
>>>2 * t
```

 - a) (1, 2, 1, 2)**
 - b) [1, 2, 1, 2]
 - c) (1, 1, 2, 2)
 - d) [1, 1, 2, 2]
7. What will be the output of the following Python code?

```
>>>my_tuple = (1, 2, 3, 4)  
>>>my_tuple.append( (5, 6, 7) )  
>>>print len(my_tuple)
```

 - a) 1
 - b) 2

c) 5

d) Error

Descriptive Questions

1. What is list in python, explain it's merits and demerits .
2. Can you explain tuple methods with suitable examples?
3. How can we create a dictionary and remove the item from it?
4. Write a python program to find second largest number in the given list.
5. Python Program to Generate Random Numbers from 1 to 20 and Append Them to the List
6. Python program to Sort a List of Tuples in Increasing Order by the Last Element in Each Tuple