

Online Workshop on ‘Applications of Python programming in Data analytics and Machine Learning – Research Perspective’

16.6.20 Day2 session 1

Dr. S.Mohideen Badhusha
Sr.Professor/ CSE department
Alva’s Institute Engineering and
Technology
Mijar, Moodbidri, Mangalore

Lists and Tuples



Objectives of the Day 2 session 1

To acquire a basic knowledge in List and Tuple

To comprehend the in-built functions and operations in List and Tuple

To practice the simple programs in List and Tuple

- Python offers a range of compound data types often referred to as sequences.
- List is one of the most frequently used and very versatile datatypes used in Python.

Mutable object is the one which can be modified

Mutable ordered sequence of items of mixed types

How to create a list?

- Ex: `L=[1,2,3,4,5]`
- It can have any number of items and they may be of different types (integer, float, string etc.).

empty list

my_list = []

list of integers

my_list = [1, 2, 3]

list with mixed datatypes

my_list = [1, "Hello", 3.4]

Also, a list can even have another list as an item. This is called nested list.

nested list

my_list = ["mouse", [8, 4, 6], ['a']]

List Index

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

```
print(my_list[0])
```

```
# Output: p
```

```
print(my_list[2])
```

```
# Output: o
```

```
print(my_list[4])
```

```
# Output: e
```

```
# my_list[4.0]
```

```
# Error! Only integer can be used for indexing
```

```
# Nested List
```

```
n_list = ["Happy", [2,0,1,5]]
```

```
# Nested indexing
```

```
print(n_list[0][1])
```

```
# Output: a
```

```
print(n_list[1][3])
```

```
# Output: 5
```

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.

P	R	O	G	R	A	M	I	Z	
0	1	2	3	4	5	6	7	8	9
-9	-8	-7	-6	-5	-4	-3	-2	-1	

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

```
print(my_list[-1])
```

```
# Output: e
```

```
print(my_list[-5])
```

```
# Output: p
```

How to slice lists in Python?

```
my_list = ['p','r','o','g','r','a','m','i','z']
```

```
print(my_list[2:5])  
# o/p: ['o', 'g', 'r']
```

```
print(my_list[:-5])  
# o/p: ['p', 'r', 'o', 'g']
```

```
print(my_list[5:])  
# o/p : ['a','m','i','z']
```

```
print(my_list[:])  
# o/p : ['p','r','o','g','r','a','m','i','z']
```

```
odd = [2, 4, 6, 8]
```

```
# change the 1st item  
odd[0] = 1  
# o/p: [1, 4, 6, 8]
```

```
# change 2nd to 4th items  
odd[1:4] = [3, 5, 7]  
print(odd)  
# Output: [1, 3, 5, 7]
```



```
odd = [1, 3, 5]
odd.append(7)
print(odd)
# Output: [1, 3, 5, 7]
```

```
odd.extend([9, 11, 13])
print(odd)
# Output: [1, 3, 5, 7, 9, 11, 13]
odd = [1, 3, 5]
print(odd + [9, 7, 5])
# Output: [1, 3, 5, 9, 7, 5]
```

```
print(["re"] * 3)
#Output: ["re", "re", "re"]
```

```
odd = [1, 9]
#odd.insert(index,element added)
odd.insert(1,3)
print(odd)
# Output: [1, 3, 9]
```

```
odd[2:2] = [5, 7]
print(odd)
# Output: [1, 3, 5, 7, 9]
```

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

```
# delete one item
del my_list[2]
print(my_list)
# Output: ['p', 'r', 'b', 'l', 'e', 'm']
```

```
# delete multiple items
del my_list[1:5]
print(my_list)
# Output: ['p', 'm']
```

```
# delete entire list
del my_list
```

```
print(my_list)
# Error: List not defined
```

```
my_list = ['p','r','o','b','l','e','m']  
my_list.remove('p')  
print(my_list)  
# Output: ['r', 'o', 'b', 'l', 'e', 'm']
```

```
print(my_list.pop(1))  
# Output: 'o'
```

```
print(my_list)  
# Output: ['r', 'b', 'l', 'e', 'm']
```

```
print(my_list.pop())  
# Output: 'm'
```

```
print(my_list)  
# Output: ['r', 'b', 'l', 'e']
```

```
my_list.clear()  
print(my_list)  
# Output: []
```

```
my_list = [3, 8, 1, 6, 0, 8, 4]  
print(my_list.index(8))  
# Output: 1
```

```
print(my_list.count(8))  
# Output: 2
```

```
my_list.sort()  
print(my_list)  
# Output: [0, 1, 3, 4, 6, 8, 8]
```

```
my_list.reverse()  
print(my_list)  
# Output: [8, 8, 6, 4, 3, 1, 0]
```

Summary of List Functions

Append()- Add an element into list

extend()- Add all elements of a list to the another list

Insert() - Insert an item at the defined index

Remove()- removes an item from the list

Pop()- removes and returns an element at the given index

Clear()- removes all items from the list

Index()- returns the index of the first matched item

Count()- returns the count of number of items passed as an argument

Built-in List Functions

SN	Function with Description
1	len(list)-Gives the total length of the list.
2	max(list)-Returns item from the list with max value.
3	min(list)-Returns item from the list with min value.
4	list(seq)-Converts a tuple into list.
5	sum(list)-Adds the all the elements in the list (elements should be numerical)

Lists and functions

```
print(len(nums))  
print(max(nums))  
print(min(nums))  
print(sum(nums))  
print(sum(nums)/len(nums))#average
```

Note: The sum() function only works when the list elements are numbers

Basic List Operations:

Lists respond to the `+` and `*` operators much like strings; .

In fact, lists respond to all of the general sequence operations we used on strings

Python Expression	Results	Description
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	1 2 3	Iteration

List Comprehension

List comprehension consists of an expression followed by for statement inside square brackets.

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

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

```
print(pow2)
```

This code is equivalent to the following codes

```
pow2 = []  
for x in range(10):  
    pow2.append(2 ** x)
```



```
pow2 = [2 ** x for x in range(10) if x > 5]  
print(pow2)  
#output : [64, 128, 256, 512]
```

```
odd = [x for x in range(20) if x % 2 == 1]  
print(odd)  
#output : [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

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

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

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

```
print('c' not in my_list)  
# Output: True
```

What is tuple?

In Python programming, a tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

A simple **immutable** ordered sequence of items
Items can be of mixed types, including collection types

immutable object is the one which can not be modified

Sequence Types

1. Tuple

- A simple *immutable* ordered sequence of items
- Items can be of mixed types, including collection types

1. Strings

- *Immutable object is the one which can not be modified*
- **Conceptually very much like a tuple**

2. List

Mutable object is the one which can be modified

Mutable ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionalities.
- Key difference:
 - Tuples and strings are *immutable*
 - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
 - most examples will just show the operation performed on one

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place*.
- Name **li** still points to the same memory reference when we're done.

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')  
>>> t[2] = 3.14
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#75>", line 1, in -toplevel-  
    tu[2] = 3.14
```

```
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Sequence structures

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (" , ' , or """").

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line  
string that uses triple quotes."""
```

Sequence access

- We can access individual members of a tuple, list, or string using square bracket “array” notation.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
'abc'
```

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
34
```

```
>>> st = "Hello World"
>>> st[1]      # Second character in string.
'e'
```


Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]  
'abc'
```

Negative lookup: count from right, starting with -1.

```
>>> t[-3]  
4.56
```

Slicing Operations

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]  
( 'abc', 4.56, (2,3) )
```

You can also use negative indices when slicing.

```
>>> t[1:-1]  
( 'abc', 4.56, (2,3) )
```

Slicing Operations

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]  
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]  
(4.56, (2,3), 'def')
```

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

The * Operator

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

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

```
>>> "Hello" * 3  
'HelloHelloHello'
```

Concluding Tips

List- mutable (changeable) Tuple – immutable (unchangeable)

list with mixed datatypes `my_list = [1, "Hello", 3.4]`

List and Tuple starts with 0 index

Extend() - a list is to be included as elements

Append() - to be included as single element

+ is concatenation operator for list,tuple and string.