Lists

- Dynamic Arrays, **mutable** collection of data
- A list is the Python equivalent of an array, but is resizeable
- **Can contain** elements of **different types.**
- An **ordered** group of **sequences** enclosed inside **square brackets** and **separated** by symbol comma(,)
- list1 = [] **# Creation of empty List**
- list2 = list() **# empty List using list function**
- list3 = [Sequence1, Sequence2]

Examples

- country = [‘India’]
- List of strings
- states = [‘Tamilnadu’, ’Gujrat’, ’Mizoram’]
- array=[2,3,4,5,7,19,23,15,20] # list of integers
- L=[7575, “Shyam”, 25067.56] #list of mixed data
- # Following is a Nested 2-D list
- L=[[7575, “John”, 25067.56], [7531, “Joe”, 56023.2], [7821, “Jill”, 43565.23]]

Basic List Operations

- list=[2, 34, 52, 12, 9]
- len(list) #length of list is 5
- Print([1, 3, 9] + [8, 6, 5]) #Concatenation of two lists
- [‘Hello’] * 3 # [‘Hello’ , ‘Hello’ , ‘Hello’]
- [5]*3 # [5, 5, 5]
- print(7 in [1, 3, 7]) # Membership, True
- for n in [1, 3, 7] :

      print(n) # print 1,3,7

Accessing Values in Lists

- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index
- **list1 = ['physics', 'chemistry', 1995, 2000]**
- **list2 = [9, 3, 8, 4, 5, 2, 7, 1, 6]**
- **print (“first element: ", list1[0]) #physics**

Slicing

- In addition to accessing list elements one at a time, Python provides **concise syntax to access sublists**; this is **known** as **slicing**
- **Syntax for slicing**
- **list_name[start : end : step]**
- **list2 = [9, 3, 8, 4, 5, 2, 7, 1, 6]**
- **print (list2[3 : 6])   # 4,5,2**
- **print(list2[0 : : 2])   # 9,8,5,7,6**

- **n=[0, 1, 2, 3, 4]**
- **n[2 : 4]** ***# slice from index 2 to 4 ; prints [2, 3]***
- **n[2 :]** ***# slice from index 2 to the end; prints [2, 3, 4]***
- **n[: 2]** ***# slice from the start to index 2 ; prints [0, 1]***
- **n[:]** ***# slice of the whole list; prints[0, 1, 2, 3, 4]***
- **n[: -1]** ***# slice index can be negative; prints [0, 1, 2, 3]***
- **print(n[: : -1])** **# prints reverse list**
- **n[2 : 4] = [8, 9]** ***# Assign a new sublist to a slice***
- **print (n)** ***# Prints updated list: [0, 1, 8, 9, 4]***

List slicing

- **n=[0, 1, 2, 3, 4]**
- **n[2 : 4] = [8, 9]** ***# Assign a new sublist to a slice***
- **L1=[1,2,3,4,5,6,7,8,9,22,33]**
- **L2="*czechoslovakia*"**
- **L1[3:6]=L2[5:10]**
- **print(L1)** **# [1, 2, 3, 'o', 's', 'l', 'o', 'v', 7, 8, 9, 22, 33]**

- You can also assign to slices. For the most part, this means modifying multiple elements in a list:
- mylist = [10, 20, 30, 40, 50, 60, 70]
  mylist[3:5] = 'XY‘
  #So long as the item on the right side is iterable, you can assign it to a slice.

#You can expand and contract a list by assigning more or fewer item to a slice:

- mylist = [10, 20, 30, 40, 50, 60, 70]
  mylist[3:5] = 'WXYZ‘

- In the above example, our slice was only two items long. However, we assigned four elements to it. This meant that the list grew as a result of our assignment. We can similarly shrink it:

-
  ```
  mylist=[10, 20, 30, 'W', 'X', 'Y', 'Z', 60, 70]
  mylist[3:7] = [99, 98]
  # [10, 20, 30, 99, 98, 60, 70]
  ```

Accessing by index

- names = ['Amir', 'Barry', 'Chales', 'Dev']
- print(names[-2]) # Chales
- print(names[1]) # Barry
- names[-1][-1] # print ‘v’
- print(names[-2][-2]) # print 'e'
- print(names[1][-5]) # print 'B'
- print(names[-2][3]) # print 'l'

List methods

- L=[92,56,12,78,92,13,1,92,5]
- L.count(92) *#print number of occurecces of 92 -> 3*
- L.sort() *# will sort the existing list*
- L.sort(reverse=True) *# will reverse sort the list*
- l=[23,*"bill",67, 89, 90,"abc", "xyz"]*
- *"""remove searches for an element in list and*
- *deletes it"""*
- l.remove(*"abc")*
- l.remove(89)
- print(l)
- del(l[2]) *#deletes third element*

Deleting whole list

- del L[:] # Deletes all elements from a list, leaves empty list object
- L.clear() # Remove all items from the list. Equivalent to del L[:]
- print(L) # Prints empty list []
- del L # Important to note here that entire list object is deleted
- print(L) # It will give error because list object does not exist now

- L=[23,*"bill",67, 90,"abc"*]
- L.append('bar') *# Add a new element to the end of the list*
- L.insert(3,22) # Inserts a new element at specified index, need two argument
- x = L.pop() *# Remove and return the last element of the list* ->abc
- x=L.pop(2) #pop out by index-> 67
- L1=[78,"psit",23]
- L.extend(L1) # Appends a specified list

- **L=["abc",30,50,30,4]**
- **x=L.index(30)** **#returns index of a specified element->1**
- **x=L.index(50,0,3)** **#returns index of a specified element in a specified range of index 2**
- list.clear()

Built-in functions for list

- list1 = ['abs', 'dad','zara','zero','abc']
- list2 = [456, 700, 200, 201]
- max(list1) *#finds and returns max value->*zero
- max(list2) *#finds and returns max value->*700
- min(list1) *#finds and returns min value->*abc
- min(list2) # ->200
- len(list1) *#returns length of the list->* 5
- list("Harsh") *# takes sequence types and converts them to lists-*>['H', 'a', 'r', 's', 'h']

```
L=[2,"hello",3,4.0+5.2j, 2.53,4,'python',2.3+3.5j,'new',3.72]
Lint, Lstr, Lfloat, Lcomplex=[],[],[],[]
for i in L:
    if type(i)==int:
        Lint.append(i)
    if type(i) is str:              # type(i)==str
        Lstr.append(i)
    if isinstance(i,float):    # type(i)==float
        Lfloat.append(i)
    if type(i)==complex:
        Lcomplex.append(i)
print(Lint)
print(Lstr)
print(Lfloat)
print(Lcomplex)
```

2-D List

- X = [[12,7,3], [4 ,5,6], [7 ,8,9]] #3x3, 2-D matrix
- X = [[12,7,3],   # 0th row
-         [4 ,5,6],   # 1st row
-         [7 ,8,9]]   # 2nd row

- **Accessing elements of 2-D List**
- X[1][2]=6 # 1st row, 2nd element
- X[0][0]=12

3X4 Matrix, 2-D matrix

- X= [[12, 7, 3, 6],   # 0th row

  [4, 5, 6, 16],   # 1st row
- [7, 8, 9, 17]]   # 2nd row
  - 0th 1st 2nd 3rd elements of a row

3X4X2 matrix, 3-D Matrix

- Y= [[[1,2], [7,4], [3,6],[12,23]],   # 0th row

  [[12,22], [5,6], [7,8],[1,2]],   # 1st row
- [[11,5], [6,4], [6,9],[2,43]]]   # 2nd row
  - 0th   1st   2nd   3rd   elements of a row

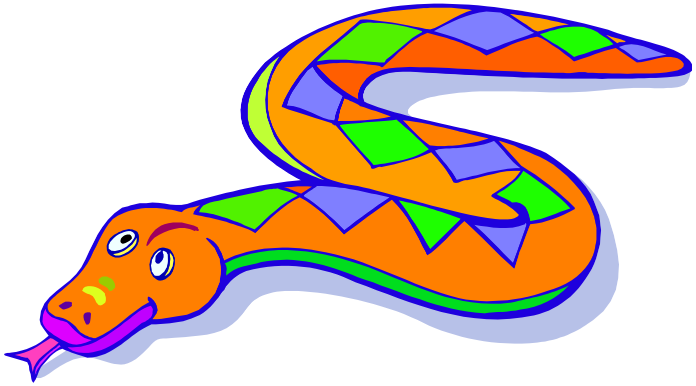
```
X = [[12, 7, 3],            # 3x3 matrix
     [4, 5, 6],
     [7, 8, 9]]
Y = [[5, 8, 1, 2],          # 3x4 matrix
     [6, 7, 3, 0],
     [4, 5, 9, 1]]
result = [[0, 0, 0, 0],   # result will be 3x4 matrix
          [0, 0, 0, 0],
          [0, 0, 0, 0]]
for i in range(len(X)):   #iterate through rows of X
    for j in range(len(Y[0])):#iterate through columns of Y
        for k in range(len(Y)): #iterate through rows of Y
            result[i][j] += X[i][k] * Y[k][j]
```

Containers

lists, sets, tuples and dictionaries

- **Python includes several built-in container types:**
- Strings, lists, tuples are sequences
- **Lists**: Dynamic Arrays, **mutable** collection of data of **different type.**
- **Tuple**: Light weight **immutable** collection of data of **different type.**
- **Dictionaries**: **Associative arrays**, **keys** must be immutable

Generating Lists using “List Comprehensions”



List Comprehensions

- A powerful feature of the Python language.
  - Generate a new list by applying a function to every member of an original list.
  - Python programmers use list comprehensions extensively. You'll see many of them in real code.
- Syntax of list comprehension is based on set builder notation in mathematics. Set builder notation is a mathematical notation for describing a set by stating the property that its members should satisfy.
- It is tricky also, If you're not careful, you might think it is a for-loop, an 'in' operation, or an 'if' statement since all three of these keywords ('for', 'in', and 'if') can also be used in the syntax of a list comprehension.
- It's something special all its own.

List Comprehensions Syntax 1

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

Note: Non-standard colors on next several slides to help clarify the list comprehension syntax.

[expression **for** name **in** list]

- Where expression is some calculation or operation acting upon the variable name.
- For each member of the list, we set name equal to that member, calculate a new value using expression, and then we collect these new values into a new list which becomes the return value of the list comprehension.

List Comprehension Syntax 2

- If the original list contains a variety of different types of values, then the calculations contained in the expression should be able to operate correctly on all of the types of list members.
- If the members of list are other containers, then the name can consist of a container of names that match the type and "shape" of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

List Comprehension Syntax 3

- The expression of a list comprehension could also contain user-defined functions.

```
>>> def subtract(a, b):
      return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]

>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Filtered List Comprehension 1

[expression **for** name **in** list **if** filter **]**

- Similar to regular list comprehensions, except now we might not perform the expression on every member of the list.
- We first check each member of the list to see if it satisfies a filter condition. Those list members that return False for the filter condition will be omitted from the list before the list comprehension is evaluated.

Filtered List Comprehension 2

[expression **for** name **in** list **if** filter]

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

Nested List Comprehensions

- Since list comprehensions take a list as input and they produce a list as output, it is only natural that they sometimes be used in a nested fashion.

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2].
- So, the outer one produces: [8, 6, 10, 4].