Introduction to Lists

- At the end of this class, students will be able to-
  - Use the variable type – List
  - Create and modify Lists using List built – in functions

Lists

we look at a means of structuring and accessing a collection of data. In particular, we look at a way of organizing data in a linear sequence, generally referred to as a list.

Motivation

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. We use lists in our everyday lives—we make shopping lists, to-do lists, and mental checklists. Various forms of lists are provided by programming languages, differing in the elements they can store (mixed type?), their size (variable size?), whether they can be altered (mutable?), and the operations that can be performed on them.

Lists also occur in nature. Our DNA is essentially a long list of molecules in the form of a double helix, found in the nucleus of all human cells and all living organisms. Its purpose is also to store information—specifically, the instructions that are used to construct all other cells in the body—that we call genes . Given the 2.85 billion nucleotides that make up the human genome, determining their sequencing (and thus understanding our genetic makeup) is fundamentally a computational problem. In this chapter, we look at the use of lists and other sequences in Python.

List Characteristics	Elements
Element Type	All elements of the same type
	Elements of different types
Length	Fixed length
	Varying length
Modifiability	Mutable (alterable)
	Immutable (unalterable)
Common Operations	Determine if a list is empty
	Determine the length of a list
	Access (retrieve) elements of a list
	Insert elements into a list
	Replace elements of a list
	Delete elements of a list
	Append elements to (the end of) a list

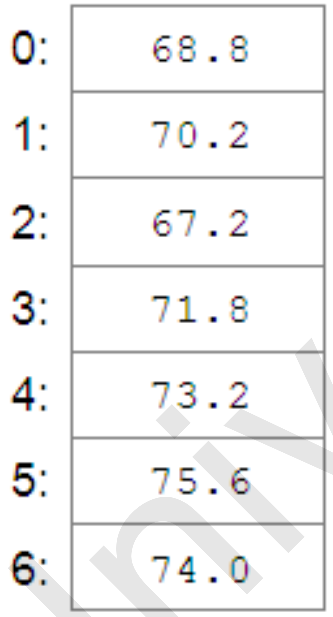
List Structures

The concept of a list is similar to our everyday notion of a list. We read off (access) items on our to-do list, add items, cross off (delete) items, and so forth. We look at the use of lists next.

What Is a List?

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

An example of a list is sequence of daily temperatures for a given week:



The location at index 0 stores the temperature for Sunday, the location at index 1 stored the temperature for Monday, and so on. It is customary in programming languages to begin numbering sequences of items with an index value of 0 rather than 1. This is referred to as *zero-based indexing*.

Lists (Sequences) in Python

A list in Python is a mutable, linear data structure of variable length, allowing mixed-type elements.

By *mutable* it is meant that the contents of the list may be altered. Lists in Python use zero-based indexing. Thus, all lists have index values 0..n-1, where n is the number of elements in the list.

Lists are denoted by a comma-separated list of elements within square brackets,

[1, 2, 3]   ['one', 'two', 'three']   ['apples', 50, True]

An **empty list** is denoted by an empty pair of square brackets, []. Elements of a list are accessed by use of an index value within square brackets,

For lst = [1, 2, 3],

lst[0] → 1   access of first element

lst[1] → 2   access of second element

lst[2] → 3   access of third element

For example, the following prints the first element of list lst,

```
print (lst[0])
```

The elements of lst can be summed as follows,

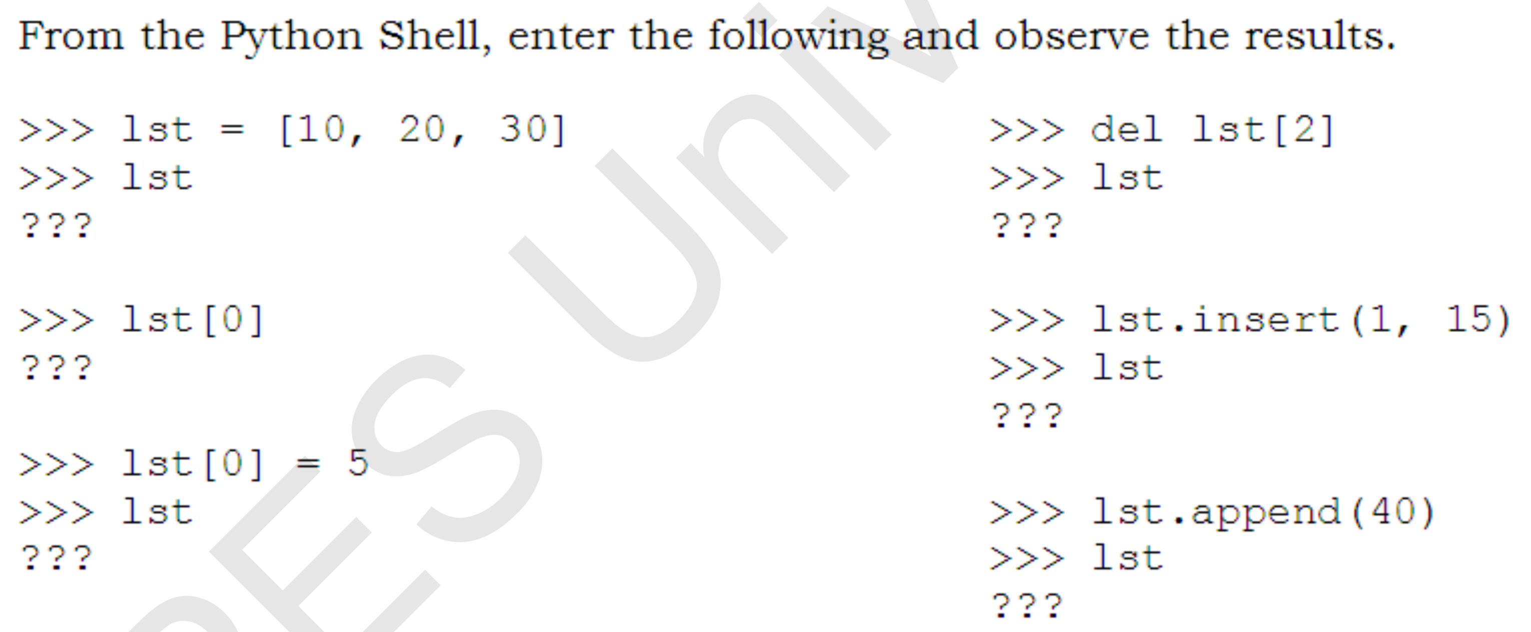
```
sum = lst[0] + lst[2] + lst[2]
```

To update,	lst[2] = 4	*replacement of 3 with 4 at index 2*
To delete,	del lst[2]	*removal of 4 at index 2*
To insert,	lst.insert(1,3)	*insertion of 3 at index 1*
To append,	lst.append(4)	appends 4 to end of list

Operation	fruit = ['banana', 'apple, 'cherry']	
Replace	fruit[2] = 'coconut'	['banana', 'apple', 'coconut']
Delete	del fruit[1]	['apple', 'cherry']
Insert	fruit.insert(2, 'pear')	['banana', 'apple', 'pear', 'cherry']
Append	fruit.append('peach')	['banana', 'apple', 'cherry', 'peach']
Sort	fruit.sort()	['apple', 'banana', 'cherry']
Reverse	fruit.reverse()	['cherry', 'apple', 'banana']

Let’s Try It

From the Python Shell, enter the following and observe the results.



From the Python Shell, enter the following and observe the results.

```
>>> lst = [10, 20, 30]
>>> lst
???

>>> lst[0]
???

>>> lst[0] = 5
>>> lst
???
```

```
>>> del lst[2]
>>> lst
???

>>> lst.insert(1, 15)
>>> lst
???

>>> lst.append(40)
>>> lst
???
```

Common List Operations

Operations commonly performed on lists include:

append	clear
Copy	Count
Extend	Index
Insert	pop
Remove	reverse
Sort	del

append()

- The append() method appends an element to the end of the list.

Syntax:

- *list*.append(*elmnt*)

- A=[1,2,3,4,5]
- >>>A.append(10)
- >>> print(A)
- [1,2,3,4,5,10]

- A=[1,2,3,4,5]
- B=[10,20,30]
- >>>A.append(B)
- >>> print(B)
- [1,2,3,4,5,[10,20,30]]

clear()

- The clear() method removes all the elements from a list.

Syntax:

- *list*.clear()
- >>>a=[1,2,3,4,5]
- >>>a.clear()
- >>>print(a)
- []

copy()

- The copy() method returns a copy of the specified list.

Syntax:

- *list*.copy()
- `>>> a=[1,2,3,4,5]`
- `>>> b=a.copy()`
- `>>> print(b)`
- `[1, 2, 3, 4, 5]`

count()

- The count() method returns the number of elements with the specified value.

Syntax:

- *list*.count(*value*)
- >>> a=["hello","hi","hello","bye","hello"]
- >>> a.count("hello")
- 3

PES UNIVERSITY

extend()

- The extend() method adds the specified list elements (or any iterable) to the end of the current list.

Syntax:

- *list*.extend(*iterable*)
- fruits = ['apple', 'banana', 'cherry']
- >>> points = (1, 4, 5, 9)
- >>> fruits.extend(points)
- >>> print(fruits)
- ['apple', 'banana', 'cherry', 1, 4, 5, 9]
- >>> b=[2,3]
- >>> fruits.extend(b)
- >>> print(fruits)
- ['apple', 'banana', 'cherry', 1, 4, 5, 9, 2, 3]

index()

- The index() method returns the position at the first occurrence of the specified value.

Syntax:

- *list*.index(*elmnt*)
- >>> a = [4, 55, 64, 32, 16, 32]
- >>> a.index(16)
- 4
- **Note:** The index() method only returns the *first* occurrence of the value.
- a = [4, 55, 64, 16,32, 16, 32]
- >>> a.index(16)
- 3

insert()

- The insert() method inserts the specified value at the specified position.

Syntax:

- *list*.insert(*pos*, *elmnt*)
- >>> a=["hello","bye","hi"]
- >>> a.insert(1,"world")
- >>> print(a)
- ['hello', 'world', 'bye', 'hi']

pop()

- The pop() method removes the element at the specified position.

Syntax:

- *list*.pop(*pos*)
- >>> a.pop(1)
- 'bye'
- >>> print(a)
- ['hello', 'hi']
- **Note:** The pop() method returns removed value.

remove()

- The remove() method removes the first occurrence of the element with the specified value.

Syntax:

- *list*.remove(*elmnt*)
- >>> a=['hello', 'world', 'bye', 'hi','world']
- >>> a.remove('world')
- >>> print(a)
- ['hello', 'bye', 'hi', 'world']

reverse()

- The reverse() method reverses the sorting order of the elements.

Syntax:

- *list*.reverse()
- >>> a=['hello', 'world', 'bye', 'hi','world']
- >>> a.reverse()
- >>> print(a)
- ['world', 'hi', 'bye', 'world', 'hello']

sort()

- The sort() method sorts the list ascending by default.
- You can also make a function to decide the sorting criteria(s).

Syntax:

- *list*.sort(reverse=True|False, key=myFunc)
- a=[5,4,3,2,1]
- >>> a.sort()
- >>> print(a)
- [1, 2, 3, 4, 5]
- >>> a.sort(reverse=True)
- >>> print(a)
- [5, 4, 3, 2, 1]

Delete

- you can also remove elements from your list. You can do this with the del statement:
- x = ["a", "b", "c", "d"]
- del(x[1])

- >>> a=['hello',‘maurice','bye','a']
- >>> a.sort(key=len)
- >>> print(a)
- ['a', 'bye', 'hello', ‘maurice']

Built-in Functions

- **Append():** Add an element to the end of the 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
- **Sort():** Sort items in a list in ascending order
- **Reverse():** Reverse the order of items in the list
- **Copy():** Returns a shallow copy of the list
- Del:

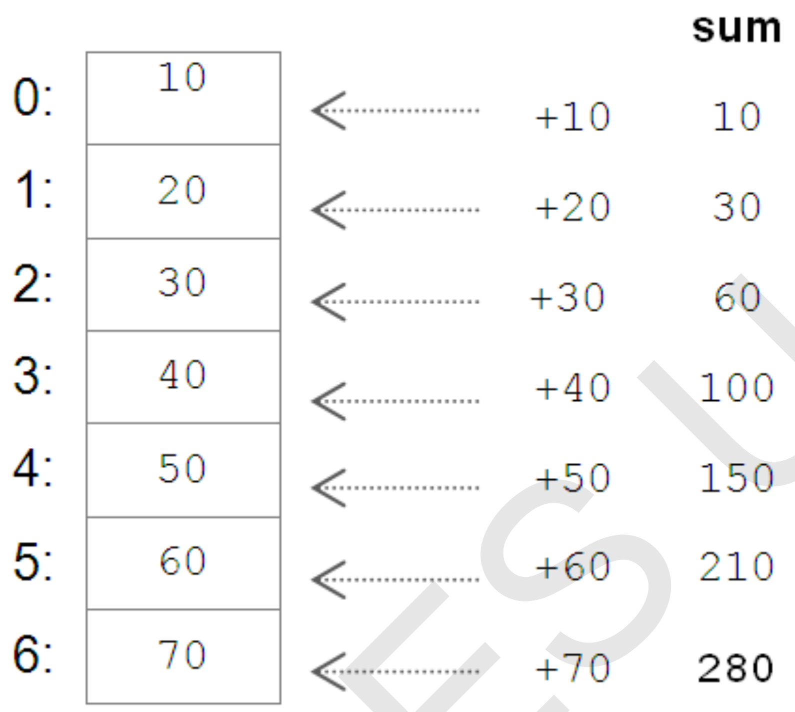
List Traversal

A **list traversal** is a means of accessing, one-by-one, each element of a list.

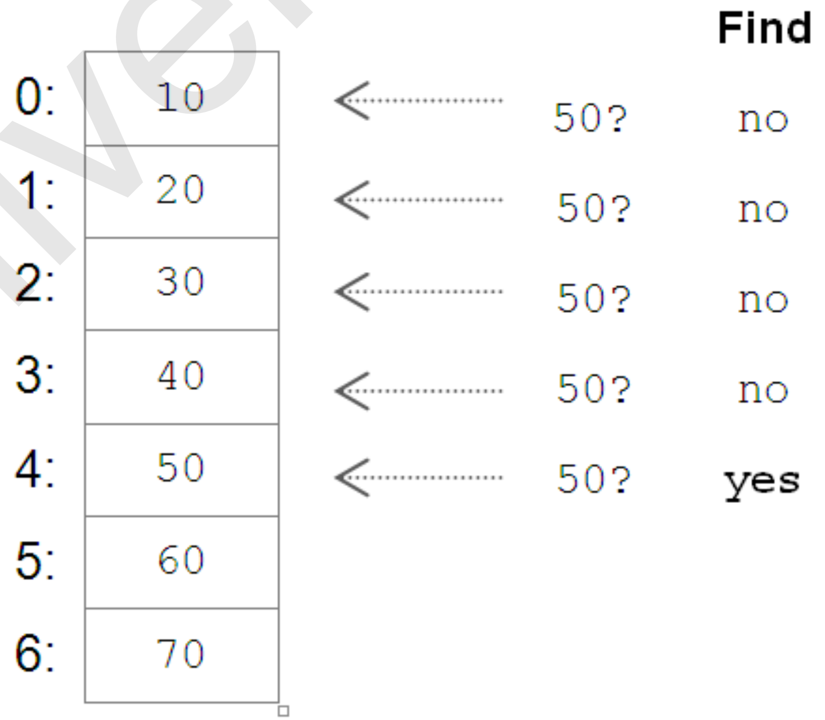
List traversal may be used, for example, to:

- search for a particular item in a list
- add up all the elements of a list

Adding up all values in the list



Searching for the value 50 in the list



PES UNIVERSITY

Write a program to add all the elements in the list.

```
a=[11,21,3,4,15,6,17,8,9,10]
s=0
for i in range(len(a)):
        s=s+a[i]
print(s)
```

Or

```
a=[11,21,3,4,15,6,17,8,9,10]
s=0
for i in a :
        s=s+i
print(s)
```

Write a program to search for the elements in the list.

```
a=[11,21,3,4,15,6,17,8,9,10]
n=int(input("enter"))
s=0
for i in range(len(a)):
        if n==a[i]:
                print(n,"present at",i+1,"pos")
                break
else:
        print("element not present")
```

WAP to find the leftmost even number if any

```
a = [11, 33, 22,55, 77]
i = 0
found=False
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")
```

Nested Lists

Lists can contain elements of any type, including other sequences.

Thus, lists can be nested to create arbitrarily complex data structures

```
class_grades = [ [85, 91, 89], [78, 81, 86], [62, 75, 77], …]
```

This list stores three exam grades for each student.

`class_grades[0]` equals `[85, 91, 89]`
`class_grades[1]` equals `[78, 81, 86]`

To access the first exam grade of the first student in the list,

```
    student1_grades = class_grades[0]  → [85, 91, 89]
    student1_exam1 = student1_grades[0] → 85
```

OR

```
    class_grades[0][0] →   85
```

To calculate the average on the first exam, a while loop can be constructed that iterates over the first grade of each student's list of grades,

```
sum = 0
k = 0

while k < len(class_grades):
    sum = sum + student_grades[k][0]
    k = k + 1

average_exam1 = sum / float(len(grades))
```

To produce a new list names `exam_avgs` containing the exam average for each student in the class,

```
exam_avgs = []
k = 0

while k < len(class_grades):
    avg = (student_grades[k][0] + \
           student_grades[k][1] + \
           student_grades[k][2]) / 3.0

    exam_avgs.append(avg)
          k = k + 1
```

Let’s Try It

From the Python Shell, enter the following and observe the results.

```
>>> lst = [[1, 2 ,3], [4, 5, 6], [7, 8, 9]]
>>> lst[0]
???
>>> lst[0][1]
???
```

```
>>> for k in lst:
        print k
???
>>> for k in lst[0]:
        print k
```

Iterating over Lists (Sequences) in Python

Python’s for statement provides a convenient means of iterating over lists (and other sequences).

Iterating over List Elements vs. List Index Values

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the *index values* of a list instead.

Loop variable iterating over the elements of a sequence	Loop variable iterating over the index values of a sequence
`nums = [10, 20, 30, 40, 50, 60]`	`nums = [10, 20, 30, 40, 50, 60]`
`for k in nums:`	`for k in range(len(nums)):`
`    sum = sum + k`	`    sum = sum + nums[k]`

Suppose the average of a list of class grades named grades needs to be computed. In this case, a for loop can be constructed to iterate over the grades,

```
for k in grades:
    sum = sum + k
print('Class average is',
      sum/float(len(grades)))
```

However, suppose that the instructor made a mistake in grading, and a point needed to be added to each student's grade? In order to accomplish this, the index value (the location) of each element must be used to update each grade value.

```
for k in range(len(grades)):
    grades[k] = grades[k] + 1
```

In this case, loop variable k is also functioning as an index variable. An **index variable** is a variable whose *changing value is used to access elements of a sequence*.

The range function can be given only one argument. In that case, the starting value of the range defaults to 0, thus,

```
range(len(grades))
```

is equivalent to

```
range(0,len(grades))
```

Let's Try It

From the Python Shell, enter the following and observe the results.

```
>>> nums = [40, 10, 60]

>>> for k in
range(len(nums)):
        print(nums[k])
???
```

```
>>> for k in range(len(nums)-1, -1, -1):
        print(nums[k])
???
```

While Loops and Lists (Sequences)

There are situations in which a sequence is to be traversed until a given condition is true. In such cases, a while loop is the appropriate control structure.

```
k = 0
item_to_find = 40
found_item = False

while k < len(nums) and not found_item:
    if nums[k] == item_to_find:
        found_item = True
    else:
        k = k + 1

if found_item:
    print('item found')
else:
    print('item not found')
```

The loop continues until either the item is found, or the complete list has been traversed. The final if statement determines which of the two possibilities of ending the loop occurred, displaying either 'item found' or 'item not found'.

Let’s Try It

From the Python Shell, enter the following and observe the results.

```
k = 0
sum = 0
nums = range(100)

while k < len(nums) and sum < 100:
    sum = sum + nums[k]
    k = k + 1

print('The first', k, 'integers sum to 100 or greater')
```

More on Python Lists

- The Assignment and Copying of Lists
- List Comprehensions

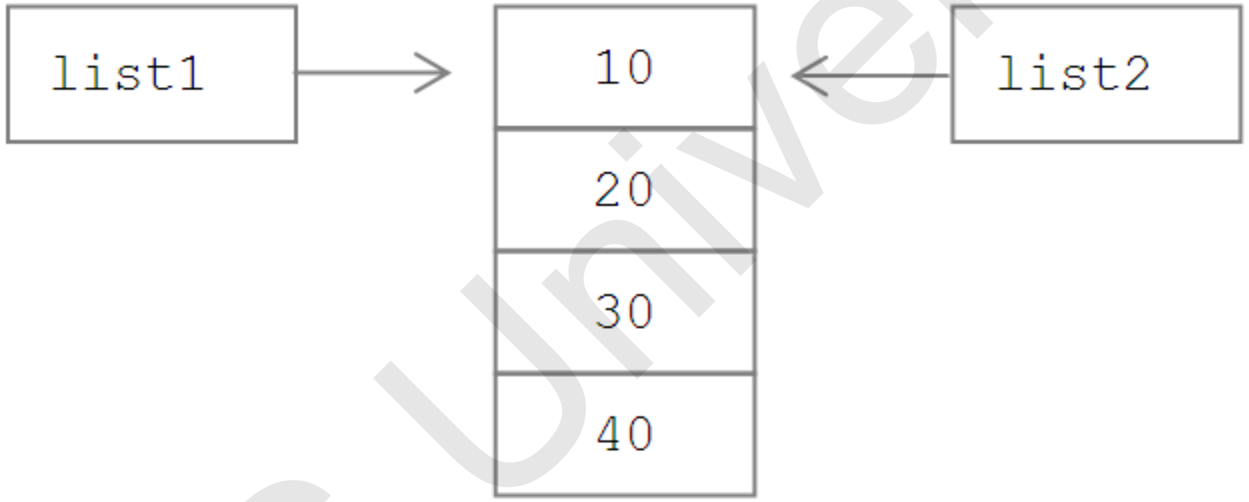
Assignment of Lists in Python

Because of the way that lists are represented in Python, when a variable is assigned to another variable holding a list,

e.g.,

list2 = list1,

each variable ends up referring to the *same instance* of the list in memory.


list1
10
20
30
40
list2

This has important implications. For example, if an element of list1 is changed, then the corresponding element of list2 will change as well,

```
>>> list1 = [10, 20, 30, 40]
>>> list2 = list1
>>> list1[0] = 5

>>> list1
[5, 20, 30, 40]   change made in list1

>>> list2
[5, 20, 30, 40]   change in list1 caused
                  by change in list2
```

This issue does not apply to strings and tuples, since they are immutable, and therefore cannot be modified.

Copying of Lists in Python

When needed, a copy of a list can be made as given below,

```
list3 = list(list1)
```

In this case, we get the following results,

```
>>> list1 = [10, 20, 30, 40]
>>> list2 = list(list1)
>>> list1[0] = 5

>>> list1
[5, 20, 30, 40]    change made in list1

>>> list2
[10, 20, 30, 40]  change in list1 does NOT cause
                   change in list2
```

When copying lists that have sublists, another means of copying called ***deep copy*** may be needed.

We discuss this further in Chapter 6 when discussing objects in Python.

Let’s Try It

From the Python Shell, enter the following and observe the results.

```
>>> list1 = ['red', 'blue', 'green']

>>> list2 = list1
>>> list1[2] = 'yellow'

>>> list1
???

>>> list2
???
```

```
>>> list1 = ['red', 'blue', 'green']

>>> list2 = list(list1)
>>> list1[2] = 'yellow'

>>> list1
???

>>> list2
???
```

PES UNIVERSITY

List Comprehensions

List comprehension is an elegant way to define and create list in python. We can create lists just like mathematical statements and in one line only. The syntax of list comprehension is easier to grasp.

- A list comprehension generally consist of these parts : Output expression, input sequence, a variable representing member of input sequence and an optional predicate part.
- For example :
- lst = [x ** 2 for x in range (1, 11) if x % 2 == 1] here, x ** 2 is output expression, range (1, 11) is input sequence, x is variable and if x % 2 == 1 is predicate part.

Example List Comprehensions	Resulting List
(a) `[x**2 for x in [1, 2, 3]]`	`[1, 4, 9]`
(b) `[x**2 for x in range(5)]`	`[0, 1, 4, 9, 16]`
(c) `nums = [-1, 1, -2, 2, -3, 3, -4, 4]` `[x for x in nums if x >= 0]`	`[1, 2, 3, 4]`
(d) `[ord(ch) for ch in 'Hello']`	`[72, 101, 108, 108, 111]`

(e) `vowels = ('a', 'e', 'i', 'o', 'u')`
`w = 'Hello'`
`[ch for ch in w if ch in vowels]`
	`['e', 'o']`

PES UNIVERSITY

Let’s Try It

From the Python Shell, enter the following and observe the results.

```
>>> temperatures = [88, 94, 97, 89, 101, 98, 102, 95, 100]
>>> [t for t in temperatures if t >= 100]
???

>>> [(t - 32) * 5/9.0 for t in temps]
???
???
```

Basic List Operations

- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

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

Slicing

- **[start : stop : steps]**
- which means that slicing will start from index start will go up to **stop** in **step** of steps. Default value of start is 0, stop is last index of list and for step it is 1
- So **[: stop]** will slice list from starting till stop index and **[start :]** will slice list from start index till end Negative value of steps shows right to left traversal instead of left to right traversal that is why **[: : -1]** prints list in reverse order.

```
# Let us first create a list to demonstrate slicing
# lst contains all number from 1 to 10
lst = range(1, 11)
print lst
#  below list has numbers from 2 to 5
lst1_5 = lst[1 : 5]
print lst1_5
#  below list has numbers from 6 to 8
lst5_8 = lst[5 : 8]
print lst5_8
#  below list has numbers from 2 to 10
lst1_ = lst[1 : ]
print lst1_
```

```
#  below list has numbers from 1 to 5
lst_5 = lst[: 5]
print lst_5
#  below list has numbers from 2 to 8 in step 2
lst1_8_2 = lst[1 : 8 : 2]
print lst1_8_2
#  below list has numbers from 10 to 1
lst_rev = lst[ : : -1]
print lst_rev
#  below list has numbers from 10 to 6 in step 2
lst_rev_9_5_2 = lst[9 : 4 : -2]
print lst_rev_9_5_2
```

PES UNIVERSITY

Summary

- List is used as a means of structuring and accessing a collection of data. In particular, we look at a way of
- Lists organize data in a linear sequence