Lecture 5: Lists and Lists Operations

Content

- Introduction to lists:
  - Creation, Indexing, Accessing, Updating, Deleting list's elements
  - Slicing
- Basic List Operations:
  - List length
  - Concatenation
  - Repetition
  - Membership
  - Append
  - Extend
  - Sort
  - Pop
  - Count
- Iterating over lists with for loops
- List comprehensions

Python Collections (Arrays)

- 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, unchangeable, and unindexed. No duplicate members.
  - Dictionary is a collection which is ordered and changeable. No duplicate members.

Introduction to lists

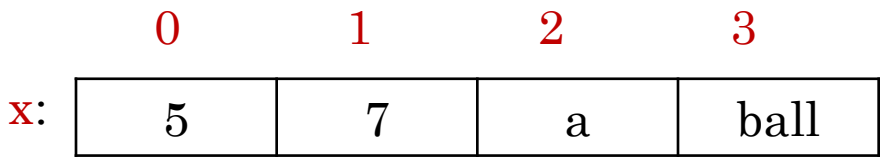
- The list is a most versatile datatype available in Python, which can be written as a list of comma-separated values (elements) between square brackets. Good thing about a list that items in a list need not all have the same type.
- A list in Python is known as a “sequence data type” like strings.
- It is an ordered collection of values enclosed within square brackets [ ].

```
list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ]

list3 = ["a", "b", "c", "d"]

 list4 = ["abc", 34, True, 40, "male"]
```

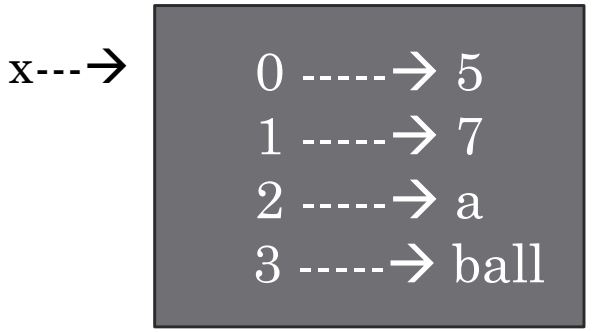
- List items are indexed, the first item has index [0], the second item has index [1] etc.

Visualizing Lists

- Informal



- Formal



A state diagram that shows the "map" from indices to elements.

Terminology

- Each value of a list is called as **element.**
- It can be of any type such as **numbers, characters, strings and even the nested lists** as well.
- The elements can be modified or mutable which means the elements can be replaced, added or removed.

Terminology

- **Ordered:**
  - When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
  - If you add new items to a list, the new items will be placed at the end of the list.
- **Changeable**
  - The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- **Allow Duplicates**
  - Since lists are indexed, lists can have items with the same value
  - Example

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
```

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.
- **Example:**

```
list1=["physics", "computer science", "art"]
list2=[1,2,3,4,5]
print("the first element in list1 is: ", list1[0])
print("the first two elements in list2 are: ", list2[0:2])
```

- This will produce following result:

```
the first element in list1 is: physics
the first three elements in list2 is: [1, 2]
```

Updating Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method:
- **Example:**

```
list1=["physics", "computer science", "art"]
print(" Value available at index 2 is: ", list1[2])
list1[2] = 2001
print("New value available at index 2 : " , list1[2])
```

- This will produce following result:

```
Value available at index 2 :
art
New value available at index 2 :
2001
```

Delete List Elements

- Items can be deleted using their specific indexes or values
- To remove a list element using its value, you can use the remove() method
- Syntax: list.remove(value)
- **Example:**

```
fruits = ['apple', 'banana', 'cherry', 'banana']
fruits.remove('banana')  # Removes the first 'banana'
print(fruits) # output ['apple', 'cherry', 'banana']
```

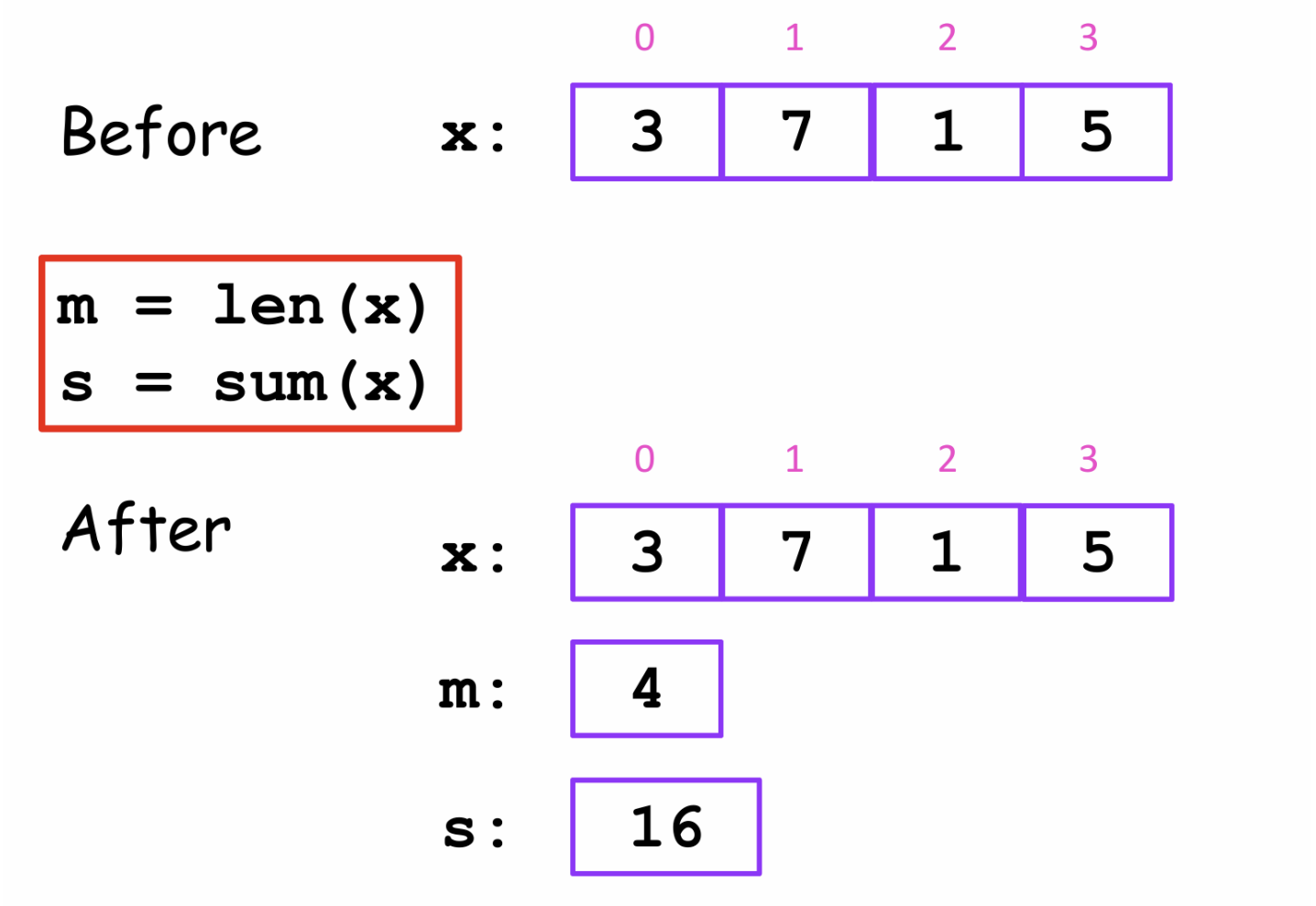
- To remove a list element using its index, you can use the del method
- Syntax: del list[index]
- **Example:**

```
my_list = [10, 20, 30, 40, 50]
del my_list[2]   # Deletes the element at index 2 (30)
print(my_list)   # Output: [10, 20, 40, 50]
```

List Built-In Functions

- There are two basic built-in functions:
  - **len** returns the length of a list
  - Syntax: len(list)
  - **sum** returns the sum of the elements in a list provided all the elements are numerical.
  - Syntax: sum(list)

len() and sum() Functions



len example

- Example 1:

```
x=[1, 3, -1, 10, 'roll', 'a']
len(x)
```

```
6
```

- Example 2:

```
x=[1, 3, -1, 10, 'roll', 'a']
length=len(x) # this will assign the output to the variable lenght
print(length)
```

```
6
```

sum example

- Example 1:

```
x=[1, 3, -1, 10]
sum(x)
```

```
13
```

- Example 2:

```
x=[1, 3, -1, 10]
result=sum(x) # this will assign the output to the variable result
print(result)
```

```
13
```

len and sum: Common errors

```
>>> x = [10,20,30]
>>> s = x.sum()
```

- AttributeError: 'list' object has no attribute 'sum‘

```
>>> x = [10,20,30, ‘a’]
>>> s = sum(x)
```

- TypeError: unsupported operand type(s) for +: 'int' and 'str’

```
>>> n = x.len()
```

- AttributeError: 'list' object has no attribute 'len'

Basic List Operations (concatenation)

- 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.
- In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter
- To concatenate two lists, we use + between list's names
- Example

```
list1 = [10, 20, 30]
list2 = ['a','b','c']
list3 = list1 + list2
print(list3)
```

```
[10, 20, 30, 'a', 'b', 'c']
```

Basic List Operations (repetition)

- The multiplication operation * will produce a repletion of the element in the list
- Example:

```
list1 = [10, 20, 30]
list1*2
```

```
[10, 20, 30, 10, 20, 30]
```

Basic List Operations (membership)

- To find if an element in a list or not, we use the membership operation to search the list. This will return true if found and false if not found
- Example:

```
list1 = [10, 20, 30]
10 in list1
```

```
True
```

```
'a' in list1
```

```
False
```

Indexing, Slicing, and Matrixes

- Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.
- Example 1:

Assuming following input:

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Example	Results	Description
L[i]	L[2]	'SPAM!'	Offsets start at zero
L[-i]	L[-2]	'Spam'	Negative: count from the right starting from -1
L[i:]	L[1:]	['Spam', 'SPAM!']	Slicing fetches sections from the provided index to the end of the list
L[:i]	L[:2]	[`spam', `Spam']	Slicing fetches sections from the first element to the element in i-1 index

- Example 2

  if x = [10,40,50,30,20]

  This

```
y = x[1:3]
z = x[:3]
w = x[3:]
```

  Is same as:

```
y = [40,50]
z = [10,40,50]
w = [30,20]
```

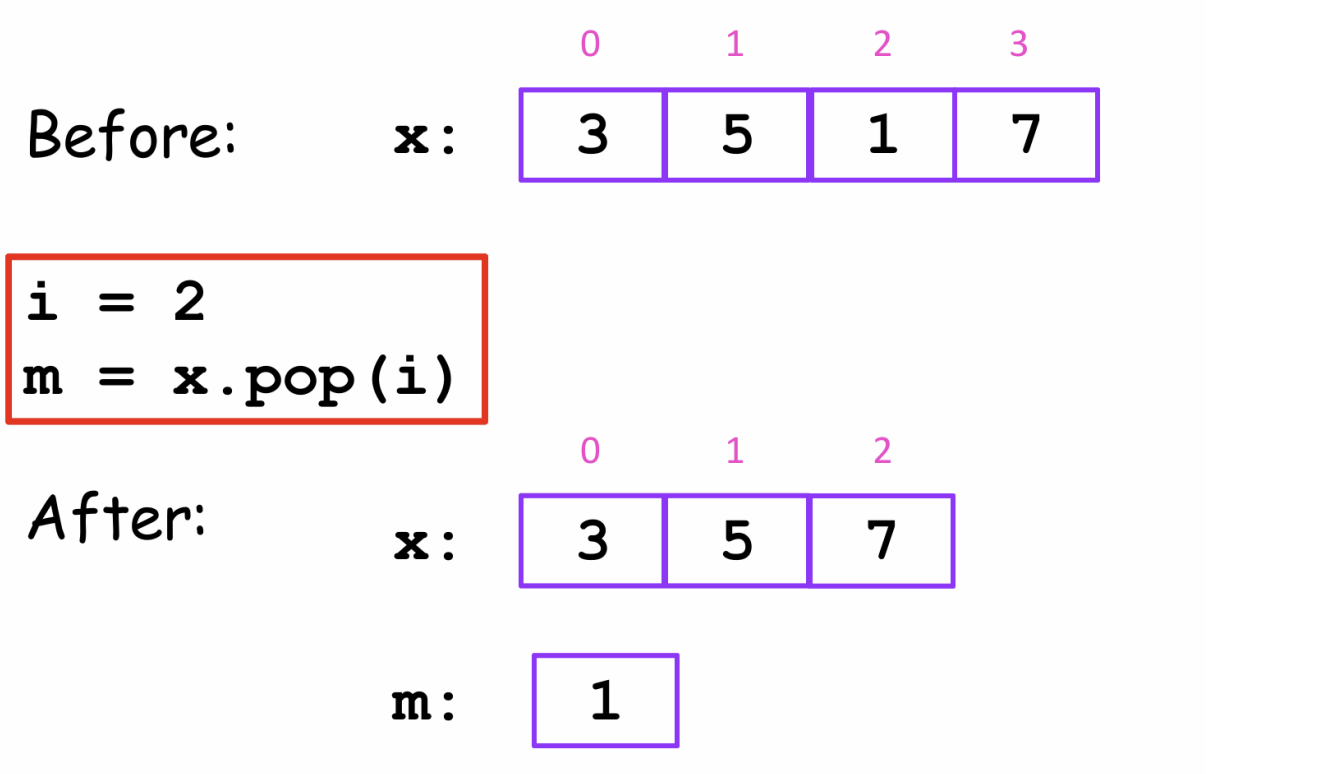
List Methods

- When these functions are applied to a list, they actually return something
  - pop
  - count
- When these methods are applied to a list, they affect the list.
  - append
  - extend
  - insert
  - sort

Let's see what they do through examples…

Pop Method

- **pop** removes an element and assign it to a variable.
- Syntax: element = list.pop(index)



Pop example

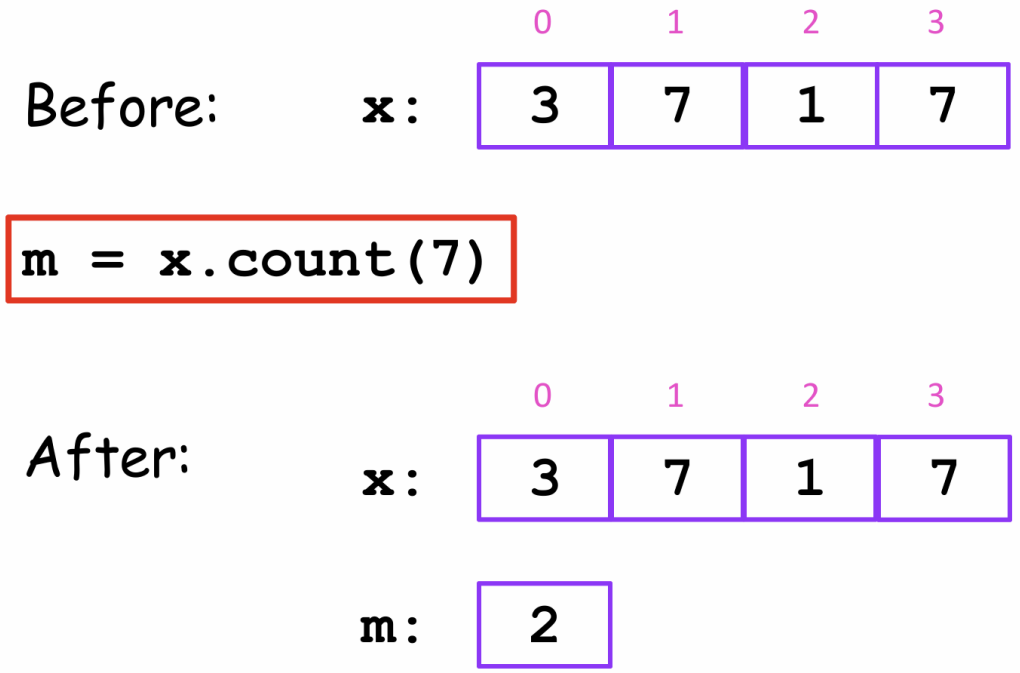
```
X= [5, 7, 'a', 'ball', 10]
print("the original list is:", X)
i=3
y=X.pop(i)
print("list after removing element at index", i, "is:", X)
print("the removed element is:",y)
```

- Output

```
the original list is: [5, 7, 'a', 'ball', 10]
list after removing element at index 3 is: [5, 7, 'a', 10]
the removed element is: ball
```

Count Method

- **count** computes the number of items in a list that have a value.
- Syntax: list.count(value)



Count example

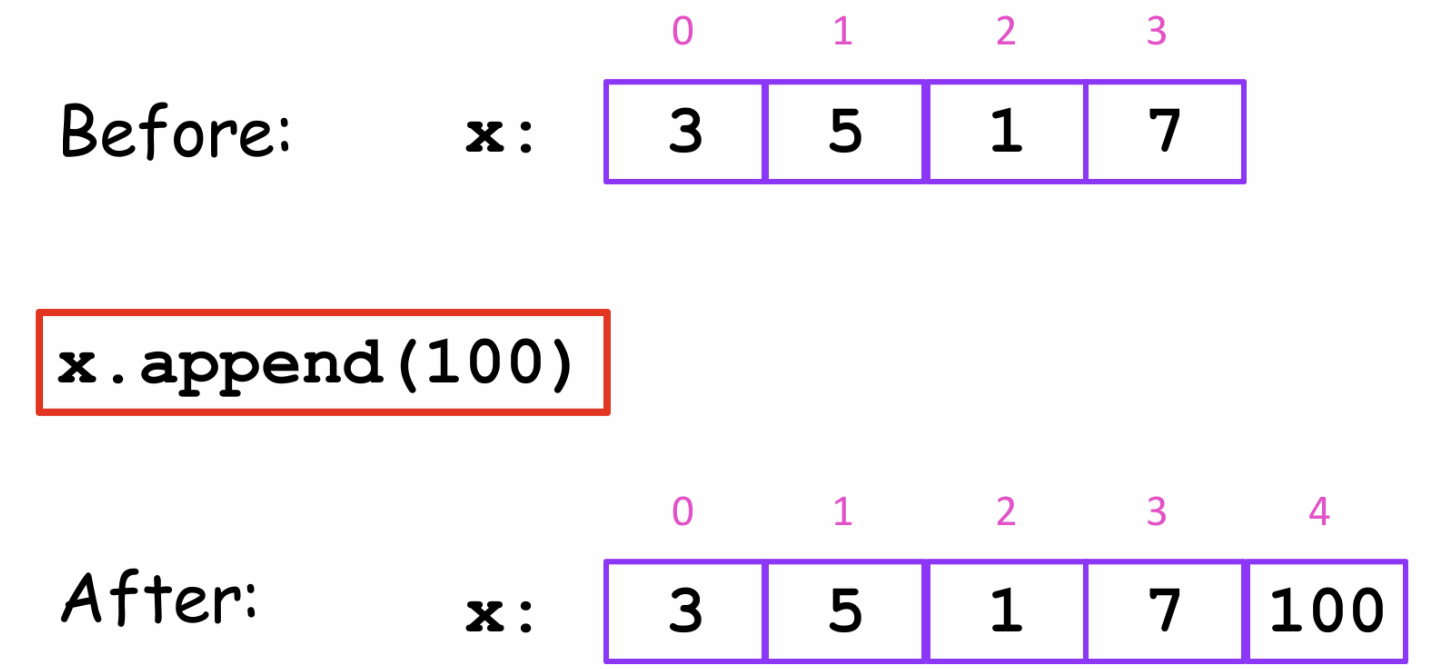
```
X=[10, 20, 10, 30, 5, 4]
search_element=10
print("there are ", X.count(search_element), "elements in the list from number", search_element)
```

- Output

```
there are  2 elements in the list from number 10
```

Append Method

- Use **append** when you want to "add an item" at the end of a given list.
- Syntax: list.append(new_value)



Append example

```
X= [5, 7, 'a', 'ball', 10]
X.append(10)
print(X)
```

- Output

```
[5, 7, 'a', 'ball', 10, 10]
```

Be Careful About Types

This is OK and synonymous with `x = [0,10]`:

```
x = [0]
x.append(10)
```

You need the square brackets. It is your way of telling Python that `x` is a list, not an int.

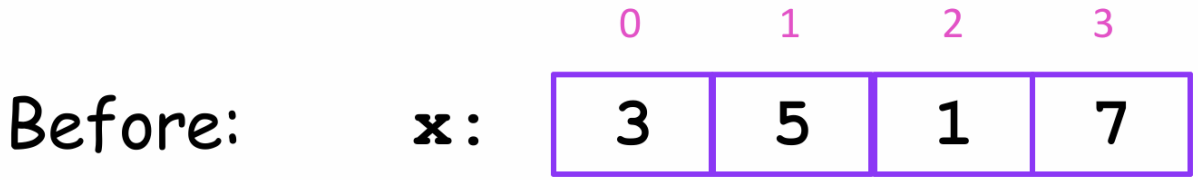
This is not OK:

```
x = 0
x.append(10)
```

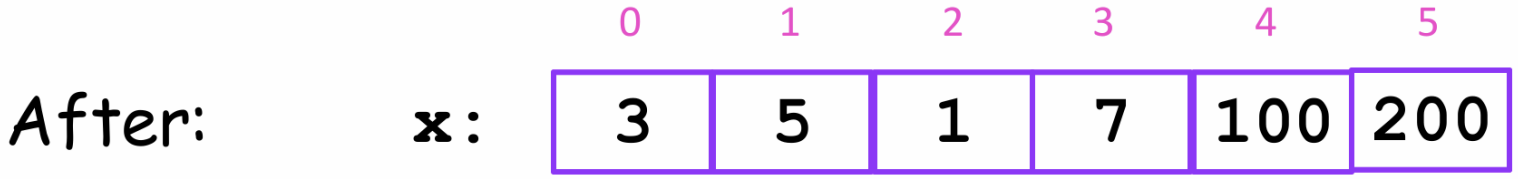
```
AttributeError: 'int' object has
      no attribute 'append'
```

Extend Method

- Use **extend** when you want to "add one list" onto the end of another list.
- Syntax: list.extend(new_list)




```
t = [100,200]
x.extend(t)
```



Extend example

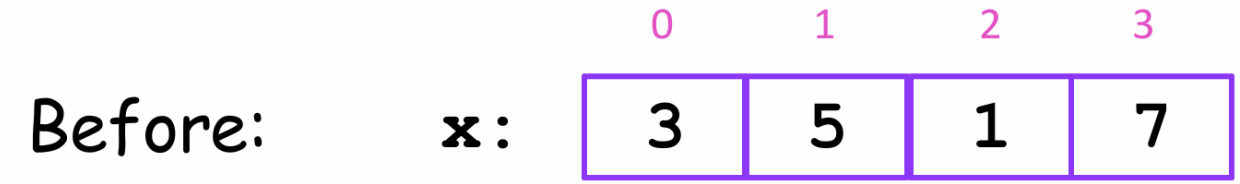
```
X= [5, 7, 'a', 'ball', 10]
Y= [4, 8, 'b']
X.extend(Y)
print(X)
```

- Output

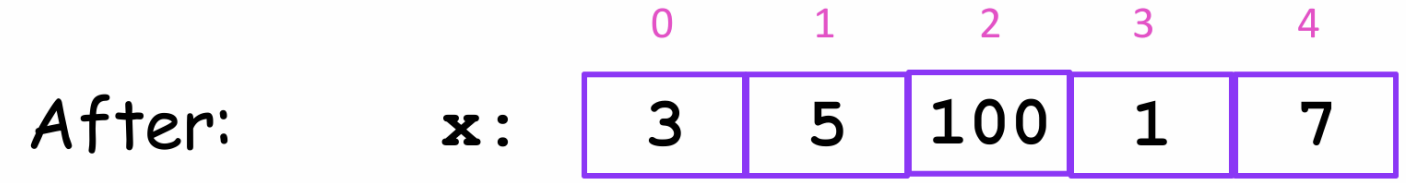
```
[5, 7, 'a', 'ball', 10, 4, 8, 'b']
```

Insert method

- Use insert when you want to insert an item into the list. Items get "bumped" to the right if they are at or to the right of the specified insertion point.
- Syntax: list.insert(index, new_value)




```
i = 2
a = 100
x.insert(i,a)
```



Insert example

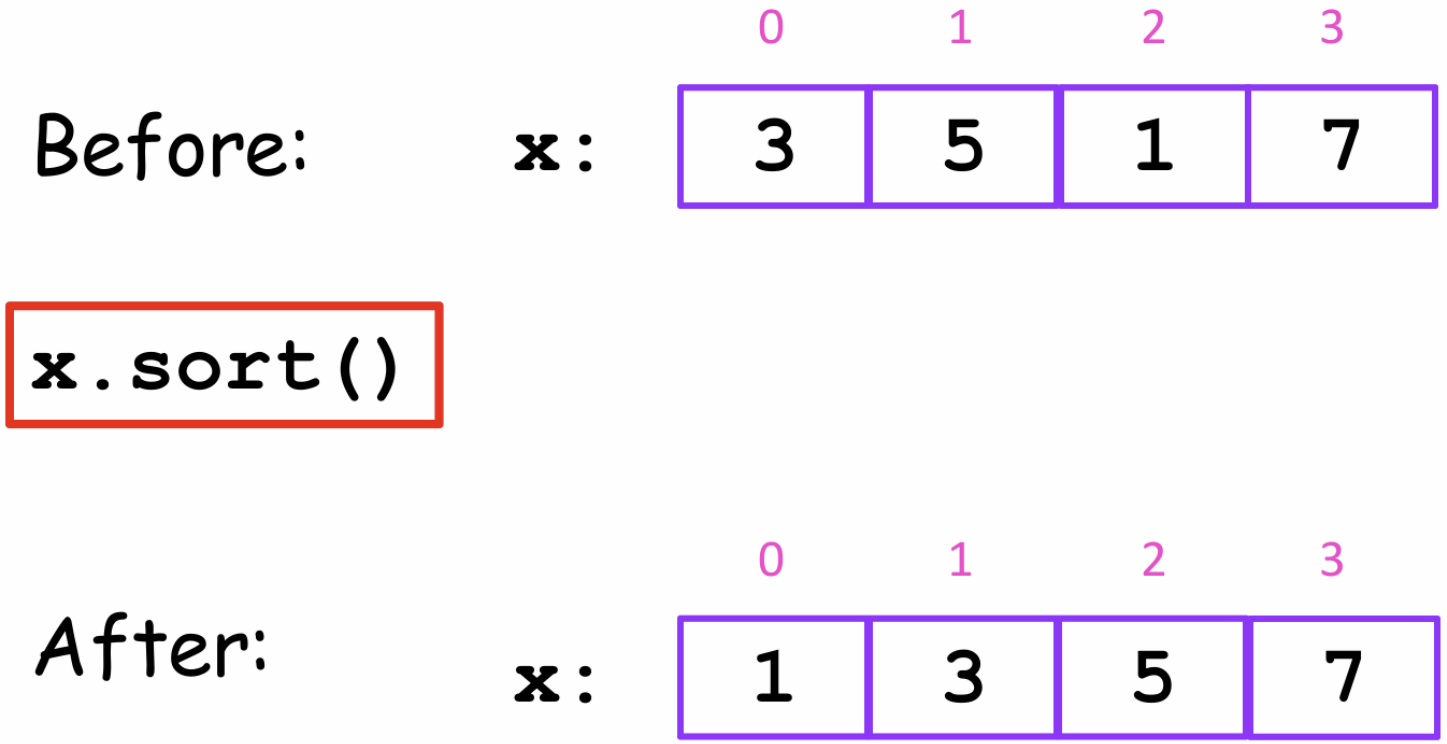
```
X= [5, 7, 'a', 'ball', 10]
i=2
val=50
X.insert(i,val)
print(X)
```

- Output

```
[5, 7, 50, 'a', 'ball', 10]
```

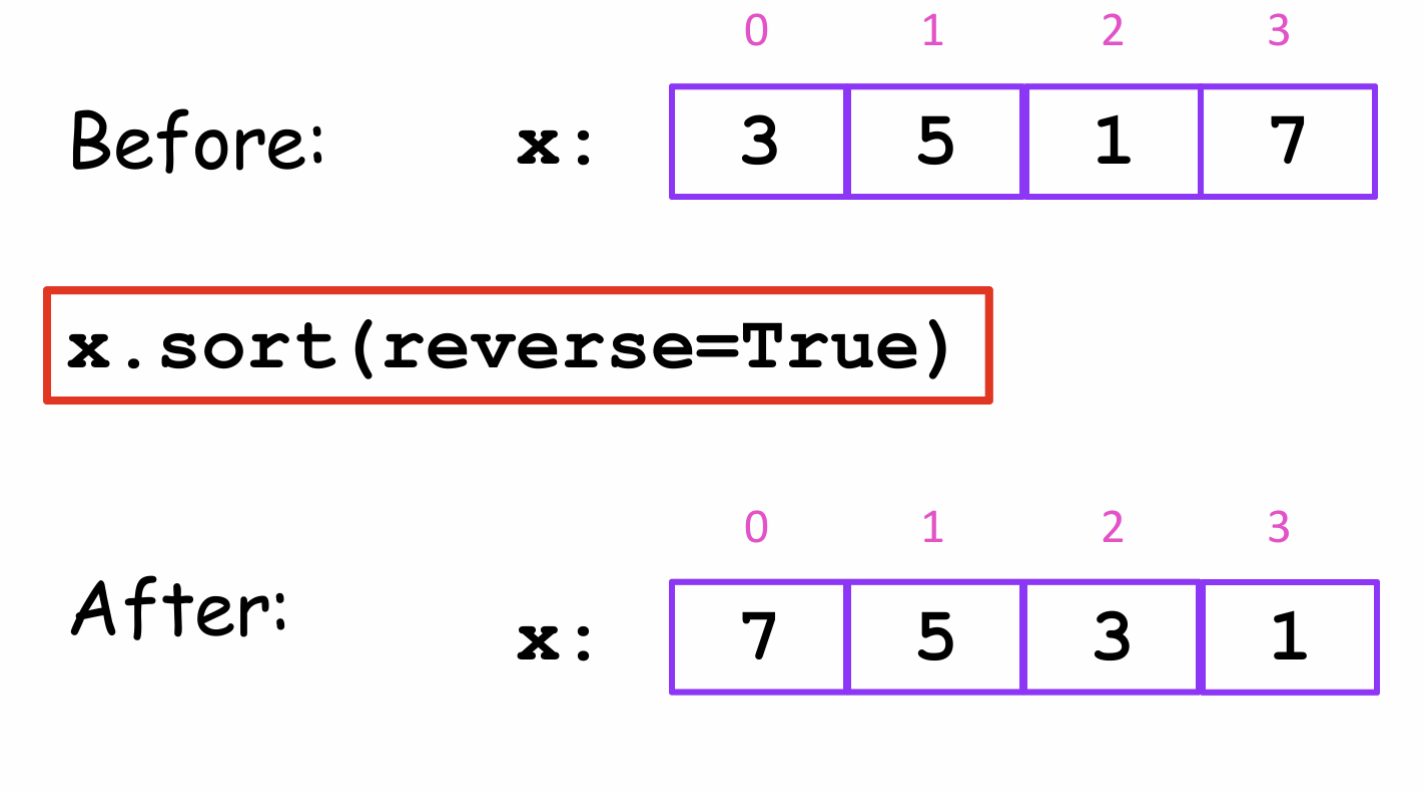
Sort Method

- Use sort when you want to order the elements in a list from little to big.
- Syntax: list.sort()



Sort Method

- An optional argument is being used to order the elements in a list from big to little.



Sort example

```
X= [5, 7, 1, -5, 10]
X.sort()
print(X)
X.sort(reverse=True)
print(X)
```

- Output

```
[-5, 1, 5, 7, 10]
[10, 7, 5, 1, -5]
```

```
a=['s','u','a','m']
a.sort()
print(a)
```

```
['a', 'm', 's', 'u']
```

```
a=['s',1, 0,'m']
a.sort()
print(a)
```

```
---------------------------
TypeError
```

Operation summary

L=[1,2,3]

Python Expression	Results	Description
len(L)	3	Length
sum(L)	6	Addition for numerical list
L.pop(2)	3	Return element in the specified index
L.count(2)	1	Return how many the element repeated
[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
L.append(10)	[1,2,3,10]	Add the new value to the end of list
L.extend([10,20])	[1,2,3,10,20]	Add the new list to the end of old list
L.insert(2,5)	[1,2,5,3]	Add the new value in the specified index
L.sort(reverse=True)	[3,2,1]	Sort the list form big to small

What is a For Loop?

- A for loop is used to iterate over a sequence.
- It allows you to execute a block of code repeatedly.
- Example:

```
for i in range(5):
  print(i)
```

- Output:

```
0
1
2
3
4
```

Iterating over lists with for loops

- A for loop is used to iterate over a sequence (like a list) and perform actions on each element.
- Syntax:

```
for i in list:
    # do anything here
```

Key Points

- **Iterates Automatically**: The loop automatically goes through each element in the list.
- **Temporary Variable**: The variable (e.g., i) holds the current element during each iteration. It represents the index value of each element in the list.
- **Flexible**: Works with any data type in the list (strings, numbers, etc.).

Example 1

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

Example 2: Calculating Squares

```
numbers=[1,2,3]
for i in numbers:
  print(i**2)
```

```
1
4
9
```

List comprehensions

- List comprehension is a simplest way of creating sequence of elements that satisfy a certain condition.
- Offers a more compact and readable alternative to for loops.
- Supports conditions and nested loops.
- Syntax : List = [expression **for** variable **in** range]

  expression → operation to perform

  variable → variable representing each element

  range → list, or sequence

Example 1

```
numbers = [1, 2, 3, 4, 5]

# Squaring each number
squared = [num ** 2 for num in numbers]
print(squared)
```

```
[1, 4, 9, 16, 25]
```

Example 2

```
fruits = ["apple", "banana", "cherry"]
new_list = [fruit.upper() for fruit in fruits]
print(new_list)
```

```
['APPLE', 'BANANA', 'CHERRY']
```

Example 3

- cross products:

```
vec1 = [2,4,6]
vec2 = [4,3,-9]

A=[x*y for x in vec1 for y in vec2]
print(A)



B=[x+y for x in vec1 for y in vec2]
print(B)

C=[vec1[i]*vec2[i] for i in range(len(vec1))]
print(C)
```

```
[8, 6, -18, 16, 12, -36, 24, 18, -54]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
[8, 12, -54]
```

Example 4

- can also use `if`:

```
vec=[1,2,3]
new1= [3*x for x in vec if x > 2]
print(new1)

new2=[3*x for x in vec if x <= 2]
print(new2)
```

```
[9]
[3, 6]
```