Lists

Lists:

- Lists are Python's most flexible ordered **collection** object type.
- lists can contain any sort of object: numbers, strings, and even other lists.
- lists may be **changed in place** by assignment to offsets and slices, list method calls, deletion statements, and more—they are ***mutable*** objects.

Python lists are:

- *Ordered collections of arbitrary objects*
- *Accessed by offset*
- *Variable-length, heterogeneous, and arbitrarily nestable*
    - lists can grow and shrink in place
    - you can create lists of lists of lists
- *Of the category "mutable sequence"*
    - lists are mutable
- *Arrays of object references*
    - Python lists contain zero or more references to other objects
    - lists really are arrays inside the standard Python interpreter, not linked structures.

Operation	Interpretation
`L = []`	An empty list
`L = [123, 'abc', 1.23, {}]`	Four items: indexes 0..3
`L = ['Bob', 40.0, ['dev', 'mgr']]`	Nested sublists
`L = list('spam')`	List of an iterable's items, list of successive integers
`L = list(range(-4, 4))`	
`L[i]`	Index, index of index, slice, length
`L[i][j]`	
`L[i:j]`	
`len(L)`	
`L1 + L2`	Concatenate, repeat
`L * 3`	
`for x in L: print(x)`	Iteration, membership
`3 in L`	
`L.append(4)`	Methods: growing
`L.extend([5,6,7])`	
`L.insert(i, X)`	
`L.index(X)`	Methods: searching
`L.count(X)`	
`L.sort()`	Methods: sorting, reversing,
`L.reverse()`	copying (3.3+), clearing (3.3+)
`L.copy()`	
`L.clear()`	
`L.pop(i)`	Methods, statements: shrinking
`L.remove(X)`	
`del L[i]`	
`del L[i:j]`	
`L[i:j] = []`	
`L[i] = 3`	Index assignment, slice assignment
`L[i:j] = [4,5,6]`	
`L = [x**2 for x in range(5)]`	List comprehensions and maps (Chapter 4, Chapter 14, Chapter 20)
`list(map(ord, 'spam'))`	

Lists in Action

Basic List Operations:

```
>>> l = [1,2,3,4]
>>> l          ➔    [1, 2, 3, 4]
>>> len(l)     ➔    4
>>> l = l + l
>>> l          ➔    [1, 2, 3, 4, 1, 2, 3, 4]
>>> l = ['Namste '] * 4
>>> l          ➔    ['Namste ', 'Namste ', 'Namste ', 'Namste ']

>>> [1, 'some string'] + "add me"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list

>>> str([1, 'some string']) + "add me"
"[1, 'some string']add me"

>>> [1, 'some string'] + list("add me")
[1, 'some string', 'a', 'd', 'd', ' ', 'm', 'e']
```

List Iteration and Comprehensions:

```
>>> 3 in [1,2,3,4]     ➔    True
>>> 7 in [1,2,3,4]     ➔    False

>>> for x in [1,2,3]:
...     print(x, end="-")
...
1-2-3-

>>> li = [c*4 for c in "test"]
>>> li          ➔    ['tttt', 'eeee', 'ssss', 'tttt']
```

- **list comprehensions** are a way to build a new list by applying an expression to each item in a sequence

```
>>> name = ['ashwani', 'praveen', 'suhash']
>>> name        ➔    ['ashwani', 'praveen', 'suhash']

>>> name1 = ['Mr '+ n for n in name]
>>> name        ➔    ['ashwani', 'praveen', 'suhash']
>>> name1       ➔    ['Mr ashwani', 'Mr praveen', 'Mr suhash']
```

Indexing, Slicing, and Matrixes

```
>>> l = [1, 'one', ['two', 2]]
>>> l[0]        ➔    1
>>> l[1]        ➔    'one'
>>> l[2]        ➔    ['two', 2]
>>> l[0] = 99
```

```
>>> l                    ➔    [99, 'one', ['two', 2]]

>>> matrix = [[1,2,3], [4,5,6], [7,8,9]]
>>> matrix               ➔    [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix[0]            ➔    [1, 2, 3]
>>> matrix[0][1]         ➔    2
>>> matrix[2][1]         ➔    8

[
 0  1  2
[1, 2, 3],          => row0
[4, 5, 6],          =>row1
[7, 8, 9]           =>row2
]
```

Changing Lists in Place

- lists are **mutable.**

Index and slice assignments:

```
>>> l                    ➔    [99, 'one', ['two', 2]]
>>> l[0] = 1
>>> l                    ➔    [1, 'one', ['two', 2]]
>>> l[0:2] = [3, 'three']
>>> l                    ➔    [3, 'three', ['two', 2]]
```

- Both index and slice assignments are **in-place changes**
- Python replaces the single object reference at the designated offset with a new one.
- *Slice assignment* - replaces an entire section of a list in a single step.
- it is perhaps best thought of as a combination of two steps:
    - *Deletion*. The slice you specify to the left of the = is deleted
    - *Insertion*. The new items contained in the iterable object to the right of the = are inserted into the list on the left, at the place where the old slice was deleted
- This isn't what really happens, but it can help clarify why the number of items inserted doesn't have to match the number of items deleted.

```
>>> l = [1,2,3]
>>> l[1:2] = [4,5]
>>> l                    ➔    [1, 4, 5, 3]
>>> l[1:1] = [6,7]
>>> l                    ➔    [1, 6, 7, 4, 5, 3]
>>> l[1:2] = []
>>> l                    ➔    [1, 7, 4, 5, 3]
```

- slice assignment replaces an entire section, or "column," all at once
- even if the column or its replacement is empty.
- slice assignment can be used to replace (by overwriting), expand (by inserting), or shrink (by deleting) the subject list.

List method calls:

```
>>> l = [1,456,67,23,75,32]
```

```
>>> l.sort()
>>> l                    ➔    [1, 23, 32, 67, 75, 456]
>>> l.append('append me')
>>> l                    ➔    [1, 23, 32, 67, 75, 456, 'append me']
>>> l.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

- Comparisons never automatically convert types, except when comparing numeric type objects.

- **sort**(), orders a list in place
- it uses Python standard comparison tests (here, string comparisons, but applicable to every object type), and **by default sorts in ascending order**.

```
>>> l.remove('append me')
>>> l                    ➔    [1, 23, 32, 67, 75, 456]
```

- You can modify sort behavior by passing in ***keyword arguments***—a special "name=value"

```
>>> l.sort(reverse=True)
>>> l                    ➔    [456, 75, 67, 32, 23, 1]
```

- **append** and **sort** change the associated list object in place.

```
>>> l = [45,84,646,4651,1,25,48]
>>> l1 = l.sort()
>>> l1
```

- sort() and append() will change list in place. They return None.

```
>>> l = [45,84,646,4651,1,25,48]
>>> l1 = sorted(l)
>>> l1                   ➔    [1, 25, 45, 48, 84, 646, 4651]
```

- If you do not want to change original list, use python function **sorted().** It returns a new list. It can be used with any collection.
- help(sorted)

- **extend**() insert multiple items at the end.

```
>>> l.extend([5,6,7])
>>> l                    ➔    [45, 84, 646, 4651, 1, 25, 48, 5, 6, 7]

>>> l.append([77,88,99])
>>> l                    ➔    [45, 84, 646, 4651, 1, 25, 48, 5, 6, 7, [77, 88, 99]]


>>> l.append(77,88,99)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (3 given)
```

- **reverse**() reverses the list in-place

```
>>> l.reverse()
>>> l                    ➔    [[77, 88, 99], 7, 6, 5, 48, 25, 1, 4651, 646, 84, 45]
```

- **pop**() - delete an item from the end of the list

```
>>> l.pop()              ➔    45
>>> l                    ➔    [[77, 88, 99], 7, 6, 5, 48, 25, 1, 4651, 646, 84]
```

- **index**()—a search for the *index of* an item,

```
>>> l.index(48)          ➔    4
```

- **insert()** an item at an offset

```
>>> l.insert(1, "One")
>>> l                    ➔    [[77, 88, 99], 'One', 7, 6, 5, 48, 25, 1, 4651, 646, 84]
```

- **count()** count the number of occurrences

```
>>> l.count(7)           ➔    1
```

- **del** statement to delete an item or section in place

```
>>> del l[1]
>>> l                    ➔    [[77, 88, 99], 7, 6, 5, 48, 25, 1, 4651, 646, 84]

>>> del l[7:]
>>> l                    ➔    [[77, 88, 99], 7, 6, 5, 48, 25, 1]
```