Lists:

* It is an ordered seq. that can hold a variety of object types.

→ Uses [] brackets & commas "," to separate objects in list
→ Supports indexing & slicing.
→ It can be nested & have variety of useful methods.

e.g: (i) Integers list
list = [1, 2, 3]

(ii) Mixed obj. list
list = ['STRING', 100, 23.2]

(iii) checking length of list
len(list) [i.e. list = [1,2,3]]
o/p: 3

* Indexing & slicing works as string only.

(iv) Indexing
list = ['one', 'two', 'three']
o/p: 'one'

(v) Slicing
list[1:]
o/p: ['two', 'three']

(vi) Concatenation
→ list = ['1', '2', '3']
another list = ['4', '5', '6']
→ list + another_list
o/p: ['1', '2', '3', '4', '5', '6']

(vii) Changing the ~~de~~ object in list & changing the elements which already exists in the list
list = ['one', 'two', 'three']
list[0] = 'ONE ALL CAPS'
o/p: list = ['ONE ALL CAPS', 'two', 'three']

Append: Allows you to place any item at the end of a list.

Pop: It going to pop off an item from the end of a list

(viii) Append use: Append means add (something) to the end of a written document.

list = ['One', 'Two', 'Three']
list.append('Four') (i.e. we can use tab & select append method)
Op: list = ['One', 'Two', 'Three', 'Four']

* Append actually affect the list & we can call this affecting it in a place because it permanently changes that new list to have an element at the end of this list.

(ix) Pop method: Simply removing item from the end of a list

→ list = ['One', 'two', 'three']
→ list.pop()
Op: three.
checking → list = ['one', 'two']

(x) Using pop method with index for removing item present at any index position.

list = ['1', '2', '3']
list.pop(0)
Op: '1'
checking list = ['2', '3']

Note: Reverse indexing also works in pop. method.

(xi) Sort & Reverse:

Sort: It doesn't return anything i.e. it do sorting (arrange in proper way).

Dictionaries.

list = ['a', 'e', 'x', 'b', 'c']
num_list = [4, 1, 2, 3]

list.sort()
O/p: list = ['a', 'b', 'c', 'e', 'x']
~~num.sort()~~ num_list.sort()
O/p: num_list = [1, 3, 4, 8]

→ Reverse. num_list.reverse()
O/p: num_list = [8, 4, 3, 1].

Dictionaries:

* Dictionaries are unordered mappings for storing objects. Previously we saw how lists store obj. in an ordered seq., dictionaries use a key-value pairing instead.

→ This key-value pair allows users to quickly grab objects without needing to know an index location.
i.e.
Just call the key & it returns the value associated with that key.

→ Dictionaries use curly brace "{}" & colons ":" to signify the keys & their associated values.
eg: {'key1': 'value1', 'key2': 'value2'}

May are super flexible

Dictionaries can hold int, floatpoint no. & also string.
Also can hold lists or even other dictionaries.

Page No

✱ When to choose list & when to choose a dictionary (dictionary)?

→ Dictionaries: objects retrieved by key name.

It unordered & can not be sorted.

Pros: Without knowing index we can quickly retrieved the value.

Cons: We can't sort a dictionary because a dictionary has a key value mapping. (i.e: it can insert new key value pairs wherever it deems most efficiently) "Consider".

→ List: objects retrieved by location (indexing).
Ordered seq. can be indexed or sliced.

e.g:

(i) dict = {'key1':'value1', 'key2':'value2'}
dict['key1']
O/p: 'value1'.

(ii) prices_lookup = {'apple': 2.99, 'orange': 1.99, 'milk': 5.8}
prices_lookup['apple']
O/p: 2.99.

(iii) dictionaries holding datatypes like lists & dictionary in it.

d = {'k1': 123, 'k2': [0,1,2], 'k3': {'insidekey': 100}}
d['k2']
O/p: [0,1,2]
→ d['k3']['insidekey']
O/p: 100.

(iv) Stacking either index calls or key calls to get back the value wanted.

using eg: (iii).

→ d = {'k1': 123, 'k2': [0, 1, 2], 'k3': {'insidekey': 100}}

c ['k2']

o/p [0, 1, 2]

If we want this 2 so we can use indexing.

d ['k2'][2]

o/p 2

→ d = {'key1': ['a', 'b', 'c', 'd']}

grabbing letter 'c' & making uppercase.

(i) → mylist = d['key1'] → Grab the key

(ii) → mylist = ['a', 'b', 'c'] → made that into list

(iii) → letter = mylist[2] → Indexing of list

o/p letter = 'c' → grap the letter by index

letter.upper() → upper case of the letter.

o/p 'C'

These whole in one step.

d ['key1'][2].upper()

→ 'C'

(v) Adding new key value pair.

```
d = {'k1':100,'k2':200}
d['k3'] = 300          # Assig k3.
```

Op: d = {'k1':100, 'k2':200, 'k3':300}

(vi) Overwrite existing key pair:

```
d['k1'] = 'NEW VALUE'
```

Op: d = {'k1':'NEWVALUE', 'k2':200, 'k3':300}

(vii) Gives all the keys, values for items of a dictionary

→ d = {'k1':100, 'k2':200, 'k3':300}

→ d.keys()

Op: dict_keys(['k1','k2','k3'])

→ d.values()

Op: dict_values([100, 200, 300])

→ If we want pairings together then,

d.items()

dict_items([('k1',100),('k2',200),('k3',300)])

It comes from ~~form~~ mutation that cannot be mutated or changed

Tuples:

Immutability means it can't be changed

Page No.:

* It ~~final~~ similar to list.
Key difference is Immutability.

→ Once an element is inside a tuple, it can't be reassigned

→ It uses parenthesis : (1, 2, 3)

e.g.

(i) Count repeated letters

t = ('a', 'a', 'b')
t.count('a') t.index('a') (The first ~~letter~~ appearance is only shown.)
O/p: 2 O/p: 0.

(ii) Immutable:

t = ('a', 'a', 'b')
t[0] = 'NEW'
~~Error: op~~
~~op~~
O/p: Error

t = [1, 2, 3]
t[0] = 'New'
O/p: t = ['NEW', 2, 3]

But in list we can change