Dictionaries

Dictionaries in Python – you will use this a lot!

In Python, dictionaries are **ordered** collection of **mutable objects with immutable keys.**
Elements within a dictionary have a key & value.

Ordered

This means that there is an order associated with the elements of dictionaries.

You can perform indexing or slicing since there are index numbers present.

Mutable

Mutability implies that you can change/modify elements within a dictionary.

Note: The key is immutable, but the value can be mutable.

$$Dict = \{Key_1:Value_{1,\ldots,}\ Key_n:Value_n\}$$

```
>>> dictA = {'India':'New Delhi', 'USA':'Washington DC','Germany':'Berlin','Sri
Lanka':'Colombo’}

>>> dictB = {'Apples':1, 'Pineapple':4, 'Grapes’:3}

>>> print(dictA)
{'India': 'New Delhi', 'USA': 'Washington DC', 'Germany': 'Berlin', 'Sri Lanka'
'Colombo’}

>>> print(dictB)
{'Apples': 1, 'Pineapple': 4, 'Grapes': 3}
```

DictA

Key - Type	Value - Type
India<str>	New Delhi<str>
USA<str>	Washington DC<str>
Germany<str>	Berlin<str>

DictB

Key - Type	Value - Type
Apples<str>	1 <int>
Pineapple<str>	4 <int>
Grapes <str>	3 <int>

Dictionaries in Python

Important Dict Methods to Remember

Python Code	Function
dictA = {'India':'Delhi','USA':'Washington}, dictA = {name='Sam','age=23}	Creates a dictionary with the key value pair provided
dictA.get('India', 'defaultname'), dictA['India']	Accesses value at key name provided. If key name not present, gets default
dictA.items()	Gets a list of the key-value pairs in the dictionary
dictA.values()	Gets a list of the values in the dictionary
dictA.keys()	Gets a list of the keys in the dictionary
dictA['place of birth'] = 'Singapore'	Set the value of the key associated with it
del dictA['age']	Delete the key-value pair from the dictionary
dictA.clear()	Clear dictionary of all the key-value pairs
dictA.update(dictB)	Merges a dictionary with another dictionary or with an iterable of key-value pairs
dictA.pop(), dictA.popitem()	Removes a key from it's dictionary and returns the value, popitem() removes last item

Set & Dict Comprehension

Set Comprehension works the same way list comprehension does. Only difference is using { }.

```
# Storing a set of letters.
>>> setA = {letter for letter in ‘aeieouuuooo’}
{‘a’,’e’,’i’,’o’,’u’}
```

Set = {expression for item in iterable}

Set = {expression for item in iterable if conditional}

Set = {expression1 (if conditional) else expression2 for item in iterable}

Dict Comprehension works the same way list comprehension does. Only difference is using { } and includes a key:value.

```
# Storing a dictionary of numbers and squares.
>> dictA = {i: i**2 for I in [1,2,3,4,5]}
{1:1, 2:4, 3:9, 4:!6, 5:25}
```

Dict = {key : expression for item in iterable}

Dict = {key : expression for item in iterable if conditional}

Dict = {key : expression1 (if conditional) else expression2 for item in iterable}

When not to use Comprehension

```
>>> sum([i * i for i in range(100000000000)])
Memory error

>>> sum(i * i for i in range(100000000000))
333333333283333333333350000
```

Use generator comprehension when dealing with big *huge* amounts of data.

A list comprehension in Python works by loading the entire output list into memory.

When you use list comprehension for summing 1 billion integers, your computer becomes unresponsive. This is because Python consumes more memory than the computer would like.

When the size of a list becomes problematic, it's often helpful to use a generator instead of a list comprehension in Python.

A **generator** doesn't create a single, large data structure in memory, but instead returns an iterable. Your code can ask for the next value from the iterable as many times as necessary or until you've reached the end of your sequence, while only storing a single value at a time.