DATA STRUCTURES

Data structure defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

SEQUENCE:

It is the most basic data structures in Python. In the sequence data structure, each element has a specific index. This index value starts from zero and is automatically incremented for the next element in the sequence. In Python sequence is a generic term for an ordered set. For instance, string is a sequence of characters.

random.choice(seq)

This is a widely used function in practice, wherein you would want to randomly pick up an item from a List/sequence.

LIST

It is a sequence in which elements are written as a list of comma-separated values (items) between square brackets. The key feature of a list is that it can have elements that belong to **different data types.** A list can have duplicate elements

List is mutable i.e., the value of it's elements can be changed.

SYNTAX: **list_variable=[val1, val2, . . .]**

ACCESSING VALUES IN A LIST

```
L[start:stop:step]
```

Start position *End position* *The increment*

UPDATING VALUES IN A LIST

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator.

```
z = [3, 7, 4, 2]
```

```
z[1] = "fish"
```

```
print(z)
```

```
[3, 'fish', 4, 2]
```

One can add elements in a list with the append( ) / insert( )method. These methods cannot be called on other values such as integers or strings as they are list methods only.

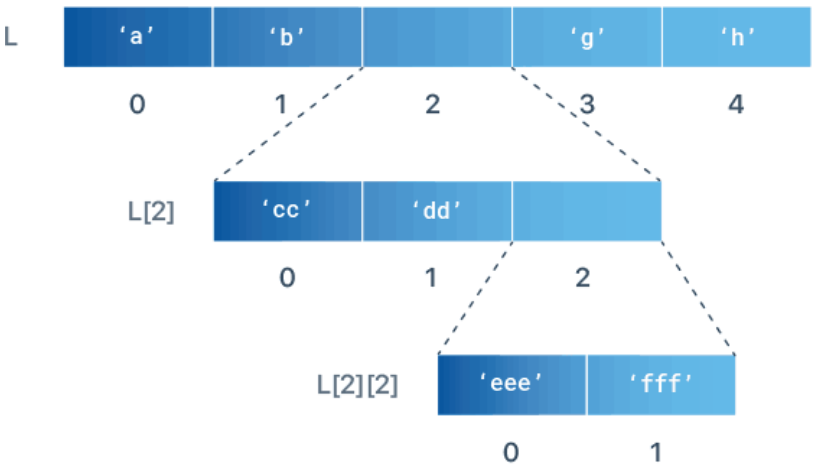
DELETING FROM A LIST

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting or the **remove()** method if you do not know. The del statement and the pop( ) method both do the same thing. The only difference between them is that pop( ) returns the removed item.

```
In [39]: a=[4,5,6,7,8,9,8,8,7]
         a.pop(3)
Out[39]: 7

In [40]: a
Out[40]: [4, 5, 6, 8, 9, 8, 8, 7]

In [41]: del a[1]
         a
Out[41]: [4, 6, 8, 9, 8, 8, 7]

In [42]: a.remove(8)
         a
Out[42]: [4, 6, 9, 8, 8, 7]
```

NESTED LIST

Nested List means a list within another list.



You can specify an element in the nested list by using a set of indices. For instance to get 'cc' we would write L[2][0]. To access 'eee' we would write L[2][2][0].

```
fruits = ["Strawberries", "Nectarines", "Apples", "Grapes", "Peaches", "Cherries", "Pears"]
vegetables = ["Spinach", "Kale", "Tomatoes", "Celery", "Potatoes"]

dirty_dozen = [fruits, vegetables]
```

INDEX OUT OF ERROR BOUND:

When using slice operation, an IndexError is generated if the index is outside the list.

ALIASING AND CLONING OF LISTS

When one list is assigned to another list using the assignment operator (=), then a new copy of the list is not made. Instead, assignment makes the two variables point to one list in memory. This is known as aliasing (second name for a piece of data).

If you want to modify a list and keep a copy of the original list then you should create a separate copy of the list and not just the reference. This process is called cloning. The slice operation is used to clone the list.

```
list1=[1,2,3,4,5,6,7,8]
list2=list1                 #Creates an Alias of list 1
list3=list1[2:6]            #Cloning the list
```

BASIC LIST OPERATIONS

Lists behave the similar way as strings when operators like + (concatenation) and * (repetition) are used.

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

LIST METHODS

Find it here!!

https://docs.python.org/3/tutorial/datastructures.html.

Sr.No.	Methods with Description
1	list.append(obj) Appends object obj to list
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index

6	list.pop(obj=list[-1]) Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort([func]) Sorts objects of list, use compare func if given

LIST COMPREHENSIONS

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
In order to create an empty list we can write L=[] or by using the list constructor L=list( ). List comprehension is used to create new lists where each element is obtained by applying some operations to each member of another sequence or iterable. It is also used to create a subsequence o those elements that satisfy a certain condition.

SYNTAX: **List=[new_item for item in sequence [if])**

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

```
['apple', 'banana', 'mango']
```

You can also use list comprehension to combine the elements of two lists. For example:

```
print([(x,y) for x in [10,20,30] for y in [40,10,60] if x!=y])
```

```
[(10, 40), (10, 60), (20, 40), (20, 10), (20, 60), (30, 40), (30, 10), (30, 60)]
```

LOOPING IN LISTS

Python's **for** and **in** constructs are extremely useful when working with lists.

USING FOR LOOP

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
 print(x)
```

```
apple
banana
cherry
```

LOOP THROUGH THE INDEX NUMBERS

You can also loop through the list items by referring to their index number.
Use the **range()** and **len()** functions to create a suitable iterable.

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

```
apple
banana
cherry
```

LOOP USING AN ITERATOR

An iterator is often used to wrap an iterable and return each item of interest. All iterators are iterable, but all iterables are not iterators. An iterator can only be used in a single for loop whereas an iterable can be used repeatedly in subsequent for loops.

The iterator is used to loop over the elements of the list. For this, the iterator fetches the value and then automatically points to the next element in the list when it is used with the next( ) method.

```
thislist = ["apple", "banana", "cherry"]
it=iter(thislist)
for i in range(len(thislist)):
  print(next(it))
```

```
apple
banana
cherry
```

LOOPING USING LIST COMPREHENSION

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

```
apple
banana
cherry
```

LOOPING USING ENUMERATE( ) FUNCTION:

This is used when you want to print both the index as well as an item in the list. The enumerate( ) function returns an enumerate object which contains the index and value of all the items of the list as a tuple.

```
thislist = ["apple", "banana", "cherry"]
for index,i in enumerate(thislist):
  print(f"{i} is at index {index}")
```

```
apple is at index 0
banana is at index 1
cherry is at index 2
```

FUNCTIONAL PROGRAMMING ASSOCAITED WITH LISTS

Functional programming decomposes a problem into a set of functions.

filter() FUNCTION

The filter() function constructs a list from those elements pf the list for which a function returns True.

SYNTAX: filter(function,sequence)

If the sequence is a string, unicode or tuple then the result will be of the same type, otherwise it is always a list.

```
def check(x):
    if(x%2==0 or x%4==0):
        return True

evens=list(filter(check,range(2,22)))
print(evens)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Using Lambda functions.

```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))
```

```
[1, 3, 5, 13]
```

map() FUNCTION:

The map() function applies a particular function on the sequence, and returns the modified list. The map function calls function(item) for each item in the sequence and returns a list of the return values.

SYNTAX: map(function,sequence)

```
# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

```
[2, 4, 6, 8]
```

Using Lambda functions.

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

```
[2, 4, 6, 8]
```

You can pass more than one sequence in the map function, but in this case we need to take care of two things:

- The function must have as many arguments as there are sequences
- Each argument is called with the corresponding item from each sequence (or None if one sequence is shorter than the other)

```
def addition(n1,n2):
    return n1 + n2

numbers1 = (1, 2, 3, 4)
numbers2 = (5,6,7,8,9)
result = map(addition, numbers1,numbers2)
print(list(result))
```

```
[6, 8, 10, 12]
```

reduce() FUNCTION:

The reduce function returns a single value generated by calling the function on the **first two items** of the sequence, then **on the result and the next item,** and so on.

SYNTAX: reduce(function,sequence)

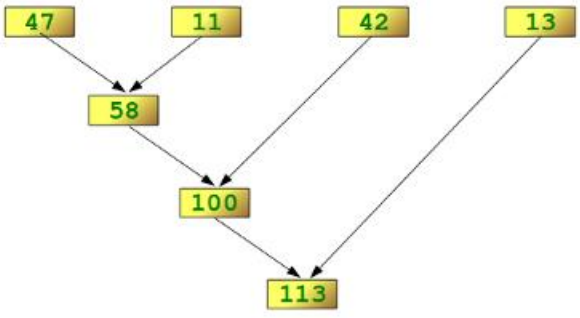
Key points to remember:

- Import functools library as it is the module that contains the function reduce
- If there is only one item in the sequence, then its value is returned
- If the sequence is empty an exception is raised
- Creating a list in a very extensive range will generate a MemoryError or Overflow Error.

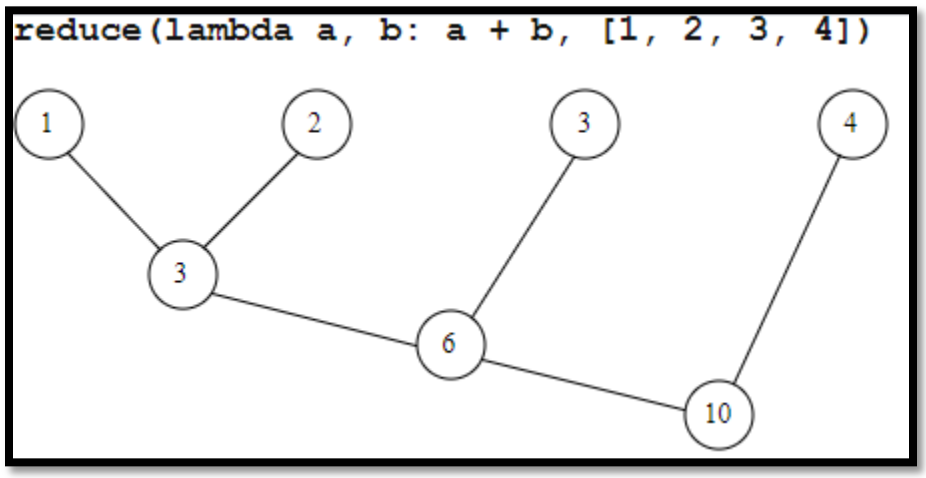
EXAMPLE:

```
import functools
def add(x,y):
    return x+y
num_list=[47,11,42,13]
print("Sum of values in list = ")
print(functools.reduce(add,num_list))
```

```
Sum of values in list =
113
```



Using Lambda functions.



TUPLE

A tuple is a sequence of **ordered, immutable objects** (like strings). This means that while you can change the value of one or more items in a list, you cannot change the values in a tuple. Tuple allows duplicate values.

Tuple uses parenthesis to define its elements whereas lists use square brackets.

We can create a tuple by putting the elements between comma-separated values withing parentheses OR simply just put different comma-separated values without parenthesis (**tuple packing**) . For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value otherwise in the absence of the comma Python treats the element as an ordinary data type (for instance integer in the example below) –

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

ACCESS TUPLE ELEMENTS

1. INDEXING

We can use the index operator `[]` to access an item in a tuple, where the index starts from 0. So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an `IndexError`.The index must be an integer, so we cannot use float or other types. This will result in `TypeError`.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])  # 'p'
print(my_tuple[5])  # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])     # 's'
print(n_tuple[1][1])     # 4
```

Output

```
p
t
s
4
```

CHECK IF ITEM EXISTS

To determine if a specified item is present in a tuple use the in keyword:

Example
Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

2. NEGATIVE INDEXING

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

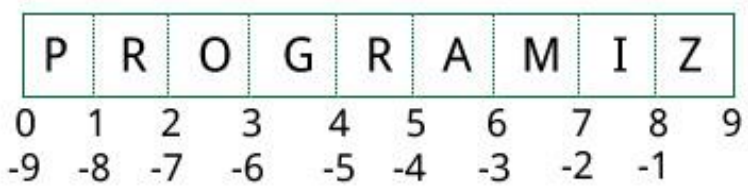
3. SLICING

We can access a range of items in a tuple by using the slicing operator colon :.

```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','i','z')
```

```
# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So, if we want to access a range, we need the index that will slice the portion from the tuple.



UPDATING A TUPLE

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

When the above code is executed, it produces the following result –

```
(12, 34.56, 'abc', 'xyz')
```

Once a tuple is created you cannot change the tuple or more precisely you cannot add or remove items from the tuple. But the possible bypass could be to convert it into a list then perform the changes then convert it back into a tuple.

DELETING ELEMENTS IN A TUPLE

Tuples are **immutable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items. To explicitly remove an entire tuple, just use the **del** statement.

BASIC TUPLE OPERATIONS

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

In fact, tuples respond to all of the general sequence operations we used on strings.

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
Tup1=(1,2,3,4,5) **Tup2=(1,2,3,4,5)** **Print(Tup1>Tup2)**	False	Comparison (use >, < ,=)

BUILT IN TUPLE FUNCTIONS

Unlike lists, tuples do not support **remove(), pop(), append(), sort(), reverse()** and **insert()** methods.

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a sequence into tuple.
6	sorted(seq) Returns sorted list

TUPLE METHODS

Method	Description
count()	Returns the number of times a specified value occurs in a tuple (t.count('x'))
index()	Searches the tuple for a specified value and returns the position of where it was found (t.index('x'))

TUPLES UTILITY IN FUNCTIONS

FOR RETURNING MULTIPLE VALUES

You can use **return (arg list)** to return more than one values from a function.

FOR PASSING VARIABLE-LENGTH ARGUMENTS TO A FUNCTION

In such cases we use a variable-length argument that begins with a '*' symbol. Any argument that starts with a '*' symbol is known as **gather** and specifies a variable length argument.

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

The opposite of gather is **scatter**. So, in case you have a function that accepts multiple arguments but not a tuple, and you wish to pass a tuple containing that many arguments, then the tuple is scattered to pass individual elements.

Observe the example below:

```
Tup=(56,3)
quo,rem=divmod(Tup)
print(quo,rem)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-d84a286c3186> in <module>
      1 Tup=(56,3)
----> 2 quo,rem=divmod(Tup)
      3 print(quo,rem)

TypeError: divmod expected 2 arguments, got 1
```

Here Tup is passed as a single argument, but the divmod( ) expects two arguments, hence the error occurs

```
Tup=(56,3)
quo,rem=divmod(*Tup)
print(quo,rem)
```

```
18 2
```

Here the symbol * denotes that there may be more than one argument. SO Python, extracts the values (scatters them) to obtain two values on which the operation is to be performed

THE zip() FUNCTION

zip( ) is a built in function that **takes two or more sequences** and **zips them into a list of tuples.** The tuple thus formed has o**ne element from each sequence.**

```
Tup=(1,2,3,4,5)
List1=['a','b','c','d','e']
print(list(zip(Tup,List1)))
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

If the two sequences have different length, then the result has the length of the shorter one

```
Tup=(1,2,3,4,5)
List1=['a','b','c','d']
print(list(zip(Tup,List1)))
```

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

UNPACKING A TUPLE

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Packing a tuple:

fruits = ("apple", "banana", "cherry")

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
cherry
```

USING ASTERIX*

If the number of variables is less than the number of values in the tuple, you can add an * to the variable name and the values will be assigned to the variable as a list:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
['cherry', 'strawberry', 'raspberry']
```

If the asterix is added to another variable name than the last, Python will assign values to the variable until the number of values left, matches the number of variables left.

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

```
apple
['mango', 'papaya', 'pineapple']
cherry
```

ADVANTAGES OF TUPLE OVER LIST

- Tuple is faster than list but they cannot be changed.
- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.

- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

SETS

Sets are same as lists but with a difference that sets are lists with no duplicate entries. A set is an unordered collection of items and is mutable. This means that we can easily add or remove items from it. Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc. Moreover, since a set is unordered, so you cannto be sure in which order the items will appear hence a set is also *unindexed*.

CREATING PYTHON SETS

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.
It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set constructor (set() function) without any argument.

ACCESSING ELEMENTS OF A SET

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

```
banana
apple
cherry
```

MODIFYING A SET IN PYTHON

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it. We can add a single element using the `add()` method, and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# TypeError: 'set' object does not support indexing
my_set[0]

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
```

```
print(my_set)
```

The copy( ) method makes a shallow copy of the set. This means that all the objects in the new set are references to the same objects as the original set.

REMOVING ELEMENTS FROM A SET

A particular item can be removed from a set using the methods discard() and remove(). The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element not present in my_set you will get an error.
# Output: KeyError
```

```
my_set.remove(2)
```

Similarly, we can remove and return an item using the pop() method.
Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary. We can also remove all the items from a set using the clear() method.

JOIN TWO SETS

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:

The union() method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

```
{'a', 1, 2, 3, 'b', 'c'}
```

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

```
{'a', 1, 2, 3, 'b', 'c'}
```

PYTHON SET METHODS

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty

remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

BUILT-IN FUNCTIONS WITH SET

Function	Description
all()	Returns `True` if all elements of the set are true (or if the set is empty).
any()	Returns `True` if any element of the set is true. If the set is empty, returns `False`.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

FROZENSETS

PYTHON FROZENSET

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the frozenset() function.
This data type supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable, it does not have methods that add or remove elements.

```
# Frozensets  initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

DICTIONARY

Dictionaries are used to store data values in key: value pairs. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}. Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples. However, the keys are case-sensitive. Two keys with the same name but in different case are not the same in python. The dictionaries are not sequencing, rather than that they are mappings. Mappings are collections of objects that store objects by key instead of by relative positions. A dictionary is a collection which is unordered (you cannot refer to an item by using an index), changeable(we can change, add or remove items after the dictionary has been created) and does not allow duplicates (cannot have two items with the same key- If you try to add a duplicate key, then the last assignment is retained). Dictionaries are written with curly brackets.

CREATING A DICTIONARY

We can also create a dictionary using the built-in `dict()` function. The dict( ) creates a directory directly from a sequence of key value pairs

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Dictionary comprehensions is another way of creating a dictionary.

SYNTAX: **{new_key : expression_for_new_value for new_key in sequence [if condition]}**

{new_key : expression_for_new_value for (new_key,new_value) in dict.items() [if condition]}

The syntax has three parts

1. For loop is used to go through the sequence
2. If condition is optional and if specified only those values in the sequence are evaluated using the expression which satisfy the condition
3. Expression

```
Dict={x: 2*x for x in range(1,10)}
print(Dict)
```

```
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}
```

ACCESSING ELEMENTS IN A DICTIONARY

TRADITIONAL METHOD OF SQUARE BRACKETS

To access values in a dictionary square brackets are used along with the key to obtain its value.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

```
Mustang
```

If you try to access an item with a key, which is not specified in the dictionary, a KeyError is generated

get() METHOD

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict.get("model")
print(x)
```

```
Mustang
```

TO GET ALL THE KEYS IN THE DICTIONARY- keys() METHOD

The key() method returns a list of all the keys used in the dictionary in an arbitrary order. The sorted() function is used to sort the keys.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict.keys()
print(x)
print(sorted(thisdict))
```

```
dict_keys(['brand', 'model', 'year'])
['brand', 'model', 'year']
```

UPDATE DIRECTORY ELEMENTS

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator. You can change the value of a specific item by referring to its key name. If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
thisdict["price"]="$15,0000"
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018, 'price': '$15,0000'}
```

USING UPDATE METHOD

The **update()** method will update the dictionary with the items from the given argument.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

REMOVING ELEMETNS FROM DICTIONARY

We can remove a particular item in a dictionary by using the **pop()** method. This method removes an item with the provided **key** and returns the **value**.
The **popitem()** method can be used to remove and return an **arbitrary (key, value)** item pair from the dictionary. All the items can be removed at once, using the **clear()** method.
We can also use the **del** keyword to remove individual items or the entire dictionary itself.

LOOPING THROUGH A DICTIONARY

You can loop over a dictionary to access only values, only keys and both using the for loop.

When looping through a dictionary, the return value are the *keys* of the dictionary.

TO GET THE KEYS

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
```

```
brand
model
year
```

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.keys():
  print(x)
```

```
brand
model
year
```

TO PRINT THE VALUES

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
```

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
```

TO PRINT BOTH KEY AND VALUES

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
```

NESTED DICTIONARY

A dictionary can contain dictionaries, this is called nested dictionaries.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
print(myfamily)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 201
1}}
```

Or, if you want to add three dictionaries into a new dictionary, create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
print(myfamily)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 201
1}}
```

COPY A DICTIONARY

You cannot copy a dictionary simply by typing **dict2 = dict1,** because: **dict2** will only be a *reference* to dict1, and changes made in dict1 will automatically also be made in **dict2**.

UING THE copy() METHOD

Use the built-in Dictionary method **copy()**

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
print(id(thisdict)==id(mydict))
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
False
```

USING THE dict() FUNCTION

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
print(id(thisdict)==id(mydict))
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
False
```

DICTIONARY FUNCTIONS

Sr.No.	Function with Description
1	cmp(dict1, dict2)- Not available in Python 3 Compares elements of both dict.
2	len(dict) Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict) Produces a printable string representation of a dictionary
4	type(variable) Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.

sorted()	Return a new sorted list of keys in the dictionary.

DICTIONARY METHODS

Sr.No.	Methods with Description
1	dict.clear() Removes all elements of dictionary *dict*
2	dict.copy() Returns a shallow copy of dictionary *dict*
3	dict.fromkeys() Create a new dictionary with keys from seq and values *set* to *value*.
4	dict.get(key, default=None) For *key* key, returns value or default if key not in dictionary
5	dict.has_key(key) Returns *true* if key in dictionary *dict*, *false* otherwise
6	dict.items() Returns a list of *dict*'s (key, value) tuple pairs
7	dict.keys() Returns list of dictionary dict's keys
8	dict.setdefault(key, default=None) Similar to get(), but will set dict[key]=default if *key* is not already in dict

9	dict.update(dict2) Adds dictionary *dict2*'s key-values pairs to *dict*
10	dict.values() Returns list of dictionary *dict*'s values

DIFFERENCE BETWEEN LISTS AND DICTIONARY

- Lists are ordered sequences while dictionary is unordered.
- The index in list needs to be a number whereas the index in dictionary can be any immutable type.
- Lists are used to lookup a value whereas a dictionary is used to take one value and look up another value.
- The key-value pair may not be displayed in the order I which it was specified while defining the dictionary. This is because Python uses hashing algorithms to provide fast access to the items stored in the dictionary. This also makes dictionary preferrable to use over a list of tuples.

WHEN TO USE WHICH DATA STRUCTURE

- Use list to store a collection of data that does not need random access
- Use lists if the data has to be frequently modified
- Use a set if you want to ensure that every element in the data structure must be unique
- Use tuples when you want that your data should not be altered

LISTS VS TUPLES VS DICTIONARY VS SET

- Tuples are lists which cannot be edited. While tuples are immutable, lists on the other hand are mutable. Tuples have fixed size so you cannot add or delete items from it, but you can easily add or delete items in a list.

- Due to the mutability differences, tuples are easier in memory and processor in comparison to lists. Tuples are best used as **heterogeneous** collections while list are being used as **homogeneous** collection
- Sets are used to store unordered values and do not have index. Unlike tuples and lists, sets can have not duplicate data.
- Dictionary is used to store key-value pairs. It is the best data structure for frequent look up operations.