SET- DATA Type:

- A set is unordered collection of unique elements.
- Set is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
- Set will not allow duplicate values.
- Insertion order is not preserved but elements can be sorted
- The major advantage of using a set is as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.
- Sets do not support indexing, slicing,
- Sets do not support concatenation and multiplication.
- There are currently two built-in set types,
  a. set,
  b. frozenset.

Set:

- The set type is mutable means the contents of set can be changed using methods like add(), update() and remove(), discard(), pop() , clear().
- Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.

Frozenset:

- The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set cannot be changed and therefore it can be used as a key in the dictionary.
- The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().
- The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

Consider the following example to create the frozen set.

```
Frozenset = frozenset([1,2,3,4,5])
print(type(Frozenset))
```

```
print("\nprinting the content of frozen set...")
for  I  in  Frozenset:
     print(i);
Frozenset.add(6)
#gives an error since we cannot change the content of Frozenset after creation
```

We can create a set in different ways,

1. Creating an empty set using set() and add elements to that empty set.

Example:

```
>>> set1 = set()                    #creating an empty set with set()
>>> set1.add(10)                    #adding elements to empty set
>>> set1.add(20)                    #adding elements to empty set
>>> set1.add(30)                    #adding elements to empty set
>>> set1.add(10)                    #adding duplicate value to set
>>> print(set1)                     {10, 20, 30}
```

2. Creating a set with elements using   set().

Example:

```
>>> set2=set([1,2,4,'a',2+4j,True])          #creating set with set()
>>> print(set2)                              {1, 2, 4, (2+4j), 'a'}
>>> type(set2)                               <class 'set'>
```

3.Creating a set with curly braces ----->> { }

Example:

```
>>> set3={1,2,3,4,"Srinivas",True}      #creating a set with curly braces
>>> print(set3)                         {1, 2, 3, 4, 'Srinivas'}
>>> type(set3)                          <class 'set'>
```

identify differences

```
s3 = {1,20,40,True,0,30,False}
>>> s3
{0, 1, 40, 20, 30}
>>>
>>> s3 = {20,False, 40,True,0,1,30}
>>> s3
{False, True, 40, 20, 30}
```

Set Functions:

Adding items to the set:

- Python provides the add() method and update() method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set.

add():

- This method is used to add new elements in to existing set.

Example:

```
>>> set1 = {1,2,3,4,5}
>>> print(set1)                    {1, 2, 3, 4, 5}
>>> set1.add(6)                    # adding element 6
>>> set1.add(7)                    # adding element 7
>> print(set1)                     {1, 2, 3, 4, 5, 6, 7}
```

Note: **we can not add new elements to the frozenset.**

```
>>>fs = frozenset([10,20,30,40])
>>>print(fs)                              {10,20,30,40}
>>> fs.add(50)                           #trying to add new element to frozenset.
```

Error: **AttributeError**: 'frozenset' object has no attribute 'add'

update():

- To add more than one item in the set, Python provides the update() method. It accepts iterable object as an argument.

Example:

```
>>> s1 = set()
>>> s1.update([10,20,30,40])
>>> s1
{40, 10, 20, 30}
>>> s1.update((50,60))
>>> s1                              #  {10,20,60,50,40,30}
```

Removing items from the set:

remove(element):

- It will remove elements from the set, if that element is not found then it will throw error like **KeyError**

Example:

```
>>> se1={1,2,3,4,5}
>>> print(se1)                          # {1, 2, 3, 4, 5}
>>> type(se1)                           # <class 'set'>
>>> se1.remove(5)                       # removing element from set
>>> se1.remove(4)                       # removing element from set
>>> se1.remove(15)        # trying to remove element which is not there in set
Error: KeyError: 15
```

discard():

- It will remove elements from the set, if that element is not found in the set then it will do nothing. means it will not return any exception here.

Example:

```
>>> se1={1,2,3,4,5}
>>> print(se1)                  # {1, 2, 3, 4, 5}
>>> se1.discard(7)              # trying to remove element which not there in the set.
>>> se1.discard(20)             # trying to remove element which not there in the set.
>>> se1.discard(5)              # removing element which is there in the  set
>>> print(se1)                  # {1, 2, 3, 4}
```

Q. What is the difference between remove() and discard() ?

remove():

- It will remove element from the given set if it is a member of given set object. If we take the element which is not present in the set then it will throws error like **KeyError**.

discard():

- It will remove element from the given set if it is a member of given set object. If we take the element which is not present in the set then it will do nothing, means it will not throw error.

pop():

- We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped out from set.

Example:

```
>>> s1 = {90,40, 10, 20, 30}
>>> s1.pop()
```

40
>>> s1.pop()
10

Note:

- If given set object not contains any elements then pop() method returns "**keyError**" when trying to delete element from set object

Example:

```
>>> s2 = set()
>>> s2.pop()
KeyError: 'pop from an empty set'
```

Assignment operator

- By using assignment operator, if we assigning given set object in to another object then both can share the same memory address.

Example:

```
>>> set1 = {1, 2, 3, 4, 5}
>>> set2 = set1
>>> set1
{1, 2, 3, 4, 5}
>>> set2
{1, 2, 3, 4, 5}
>>> id(set1)
2693878076136
>>> id(set3)
2693878076136
```

copy():

- This function copies the elements of one set to another new set and also it creates new memory value.
- copy() method always creates new memory for new set object. so both memories are different but values are same.

  Example:

```
>>> set1={1,2,3,4,5}
>>> set2=set1.copy()        #copying se1 elements to se2
>>> set1                    {1, 2, 3, 4, 5}
>>> set2                    {1, 2, 3, 4, 5}   # id values different.
```

```
>>> id(set1)
2693878076136
>>> id(set2)
2693878076360
```

clear():

- By using clear() function we can clear or remove all elements from the given set object.

Example:

```
>>> se1={1,2,3,4,5}
>>> print(se1)                {1, 2, 3, 4, 5}
>>> type(se1)                 <class 'set'>
>>> se1.clear()               #clearing the se1, so se1 will become empty
set.
>>> print(se1)                set()
```

Python Set Operations:

- Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

isdisjoint():

- This function returns True if both are "empty sets" or if both sets "contains non-matching" elements.
- if atleast one elemet matching also returns False value.

Example:

```
>>> se1 = set()
>>> se2 = set()
>>> se1.isdisjoint(se2)       True
>>> se1=set(5)
>>> se2={1,2,3}
>>> se1.isdisjoint(se2)       True
>>> se1={1,2,3}
>>> se2={1,2,3,4}
```

```
>>> se1.isdisjoint(se2)          False
```

issubset():

- x.issubset(y) returns True, if x is a subset of y.
- " <= " is an abbreviation for "Subset of".

For example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3}
>>> se2.issubset(se1)            True
>>> se1.issubset(se2)            False
Or
>>> se2 <= se1                   True
>>> se1 <= se2                   False
```

issuperset()

- x.issuperset(y) returns True, if x is a superset of y.
- " >= " is an abbreviation for "issuperset of"

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3}
>>> se2.issuperset(se1)      False
>>> se1.issuperset(se2)      True
>>> se2 >= se1                    False
>>> se1 >= se2                    True
```

Membership:

- We can also check the elements whether they belong to set or not,

Example:

```
>> se1={1,2,3,"Python",3+5j,8}
>>> 4 in se1                 False
>>> 1 in se1                 True
>>> "Python" in se1                True
>>> 10 not in se1            True
>>> "Srinivas" not in se1    True
```

union():

- It returns the union of two sets, that means it returns all the values from both sets except duplicate values.

- The same result we can get by using '|' between two sets

Syntax: <First_Set>.union(<Second_Set>) or

<First_Set> | <Second_Set>

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se1.union(se2)                {1, 2, 3, 4, 5, 6, 7}        or
>>> se1|se2                       {1, 2, 3, 4, 5, 6, 7}
Or
>>> se2.union(se1)                {1, 2, 3, 4, 5, 6, 7}
>>> se2|se1                       {1, 2, 3, 4, 5, 6, 7}
```

intersection():

- It returns an intersection elements of two sets, that means it returns only common elements from both sets.
- That same operation we can get by sing '&' operator.

Syntax: <First_Set>.intersection(<Second_Set>) or

<First_Set> & <Second_Set>

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se1.intersection(se2)              {1, 2, 3}     or
>>> se1&se2                       {1, 2, 3}
Or
>>> se2.intersection(se1)              {1, 2, 3}
>>> se2&se1                       {1, 2, 3}
```

diffferenece():

- It returns all elements from first set which are not there in the second set.

Syntax: <First_set>.difference(<Secnd_Set>) or

<First_Set> - <Second_Set>

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se1.difference(se2)                {4, 5}        or
>>> se1-se2                       {4, 5}
```

Or

>>> se2.difference(se1) {6, 7} or

>>> se2-se1 {6, 7}

intersection_update():

- The intersection_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
- The intersection_update() method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

Syntax: <First_Set>.intersection_update(<Second_Set>)

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se1.intersection_update(se2)
>>> print(se1)                    {1, 2, 3}
>>> print(se2)                    {1, 2, 3, 6, 7}
```

Or

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se2.intersection_update(se1)
>>> print(se1)                    {1, 2, 3, 4, 5}
>>> print(se2)                    {1, 2, 3}
```

differenece_update():

- The result of difference between two sets will in First_Set.

Syntax: <First_Set>.difference_update(<Second_Set>)

Example:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}
>>> se1.difference_update(se2)
>>> print(se1)                    {4, 5}
>>> print(se2)                    {1, 2, 3, 6, 7}
```

Or

```
>>> se1={1,2,3,4,5}
```

```
>>> se2={1,2,3,6,7}
>>> se2.difference_update(se1)
>>> print(se1)                    {1, 2, 3, 4, 5}
>>> print(se2)                    {6, 7}
```

symmetric_difference():

- The symmetric difference of two sets is calculated by ^ operator or symmetric_difference() method. Symmetric difference of sets, it removes that element which is present in both sets.
- It returns unmatching elements from both sets.

Syntax: <First_Set>.symmetric_difference(<Second_Set>)

Example:

```
>>> set1={1,2,3,4,5}
>>> set2={1,2,3,6,7}
>>> set1.symmetric_difference(set2)          {4, 5, 6, 7}
```

or

```
>>> set1 ^ set2                   {4, 5, 6, 7}
```

symmetric_difference_update():

- it will store the unmatching elements from both sets into First_Set.

Syntax: <First_Set>.symmetric_difference_update(<Secon_Set>)

Example:

```
>>> set1 = {1,2,3,4,5}
>>> set2 = {1,2,3,6,7}
>>> set1.symmetric_difference_update(set2)
>>> print(set1)                        {4, 5, 6, 7}
>>> print(set2)                        {1, 2, 3, 6, 7}
```

Or

```
>>> set1 ={1,2,3,4,5}
>>> set2 = {1,2,3,6,7}
>>> set2.symmetric_difference_update(set1)
>>> print(set1)                    {1, 2, 3, 4, 5}
>>> print(set2)                    {4, 5, 6, 7}
```

set ():

Example:

```
>>> s = {10,20,20,30}
>>> s                                                    {10, 20, 30}
>>> s.add(40)                          # add new element into set
>>> s                                  {40, 10, 20, 30}
>>> s.add(40,50)          # TypeError: add() takes exactly one argument (2 given)
>>> s.add([80,40,50])              # TypeError: unhashable type: 'list'
>>> s.add((80,40,50))
>>> s                                  # {40, 10, (80, 40, 50), 20, 30}
```

Q. Is set object allowed Mutable type data ?

Set object not allowed mutable type objects data in side set object.

It means, list , set and dictionary type data not allowed in set object either directly or indirectly.

For example:

```
>>> s = {1 , 2.0 , [1 , 2]}
TypeError: unhashable type: 'list'
>>> s = {1,2.0, {1,2}}
TypeError: unhashable type: 'set'
>>> s = {1,2.0, {1:2}}
TypeError: unhashable type: 'dict'
>>> s = { 1 , 2.0 , 'python' , (1 , 2 , [ 1 , 2 ] ) }
TypeError: unhashable type: 'list'
```

Q. Is set object allowed Immutable type data ?

Yes, set allowed only immutable type objects data inside set object.

It means, numbers , string and tuple type data allowed in set object either directly or indirectly.

For example:

```
>>> s1 = { 1 , 2 , ( 1 , 2 ) , 'python' }
>>> s1
{ 1 , 2 , ( 1 , 2 ) , 'python' }
```