

எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... புரட்சிக்கவி

NOTICE

www.DataScienceInTamil.com

Day 11 - Batch 3 - Python Language

Chapter 08 – Set (Collection Data Type)

Watch the video in below You Tube link

<https://youtu.be/PFWtB3UaBZM>

**To watch the recorded Python and
Data Science videos in YouTube:**

<https://www.youtube.com/channel/UCTCMjShTpZg96cXloCO9q1w>

Official Website:

<https://DataScienceInTamil.com/>

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

<https://www.DatascienceInTamil.com/#faq>

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

<https://t.me/joinchat/IUZESr-zidpjZjEx>

To Join the class, please fill the form :

<https://forms.gle/QFpLHwAoinFaX2cE6>

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

<https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVPtZ2ZS1Vsdz09>

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.
- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example. definitions, please use these websites for further reading:

- Book : Python Notes For Professionals
- <https://docs.python.org>
- <https://www.w3schools.com>
- <https://www.geeksforgeeks.org>
- <https://docs.python.org>
- <https://www.askpython.com>
- <https://docs.python.org>
- <https://www.programiz.com>
- <https://docs.python.org>
- <https://www.programiz.com/python-programming/set>
- <https://www.openriskmanagement.com/visual-overview-of-built-in-python-data-types/>
- <https://pynative.com/python-sets/>
- <https://www.alphacodingskills.com/python/notes/python-set-intersection-update.php>
- <https://codedestine.com/python-set-symmetric-difference-example/>
- <https://appdividend.com/2022/01/10/python-set-isdisjoint-method-example/>

Topic : SET

What to cover today





















1. Properties of Set
2. Set – Built in methods
3. Mutable vs immutable
4. Hashable / unhashable
5. Set methods and equivalent operators
6. add()
7. Update()
8. Union() (operator '|')
9. Intersection() / (operator '&')
10. Intersection()

11. Pop()
12. Remove()
13. Discard()
14. Copy()
15. Clear()
16. Difference()/ Operator ('-')
17. Difference_update()
18. symmetric_difference() / Operator('^')
19. SYMMETRIC_DIFFERENCE_UPDATE() / operator(^=)
20. Subset and superset (Operator $a \leq b$, $a \geq b$)
21. What happens if both sets has the same elements
22. Disjoint sets
23. Testing membership in set
24. Set DOES NOT support indexing, why?
25. Frozenset()
26. Return value from frozenset()
27. Discard()
28. Remove()

- 29. Get the unique elements of a list using set
- 30. Collections module and Counter
- 31. Sets versus multiset
- 32. Set of Sets
- 33. Counter()
- 34. Find the type() of Counter

There are four collection data types in the Python programming language:

Python Data Structures

	Indexing	Ordered	Mutable	Duplicate
 List				
 Tuple				
 Set				
 Dictionary				

There are four collection data types in the Python programming language:

Collection is heterogeneous (means it accepts any data type, where as array is homogeneous)

- **List** is a collection which is ordered and changeable. Index is possible, Allows duplicate members. not hashable ..no hash-id/ unhashable type
- **Tuple** is a collection which is ordered and unchangeable. Index is possible, . Allows duplicate members. hash-id/hashable type
- **Set** is a collection which is unordered and unindexed. No duplicate members, Index is NOT possible/ no hash-id/ unhashable type
- **Dictionary** is a collection which is ordered, changeable, and indexed (only key can be indexed – after changing the dict to list. No duplicate key members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Set – Built in methods

Python has a set of built-in methods that you can use on sets.

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all the elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	Removes the items in this set that are also included in another, specified set
<code>discard()</code>	Remove the specified item
<code>intersection()</code>	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>	Returns whether two sets have a intersection or not
<code>issubset()</code>	Returns whether another set contains this set or not
<code>issuperset()</code>	Returns whether this set contains another set or not
<code>pop()</code>	Removes an element from the set
<code>remove()</code>	Removes the specified element
<code>symmetric_difference()</code>	Returns a set with the symmetric differences of two sets
<code>symmetric_difference_update()</code>	inserts the symmetric differences from this set and another
<code>union()</code>	Return a set containing the union of sets
<code>update()</code>	Update the set with the union of this set and others

Properties of Set

Sets are unordered collections of unique objects, there are two types of set:
Sets - They are mutable and new elements can be added once sets are define
`Frozenset()`. Immutable

MELCOSE'S NOTE: `{1}` creates a set of one element, but `{}` creates an empty dict.
The correct way to create an empty set is `set()`.

Few set methods **return new sets**, but have the corresponding in-place versions:

Mutable vs immutable



Python Built-in Data Type Properties

	Immutable	Mutable
Non Iterable	<div>int</div> <div>bool</div> <div>float</div> <div>complex</div>	
Iterable	<div>string</div> <div>bytes</div> <div>frozenset</div> <div>tuple</div>	<div>bytearray</div> <div>list</div> <div>set</div> <div>dict</div>
	Hashable	Non Hashable

A hashable Python object is **any object that has a hash value** — an integer identifier of **that object which never changes during its lifetime**. To check if an object is hashable or not and find out its hash value (if it's hashable), we use the `hash()` function on this object:

TypeError: unhashable type: 'set'

reason that the elements of a set must be hashable (all of Python's *immutable* built-in objects are *hashable*).

Detailed examples below

Hashable (IMMUTABLE / CAN NOT MODIFY THE VALUE /
ITEM/ ELEMENT)

```
print(hash(3))  
print(hash(30052))  
print(hash(300528))  
print(hash(3.14))  
print(hash("String"))  
print(hash(True))  
print(hash(False))  
print(hash(3+4j))
```

```
print(hash(bytes(4)))
print("=====")
print(hash( (2,8) ))
print(hash(frozenset()))
```

Unhashable (MUTABLE / CAN BE MODIFIED)

```
print(hash([2,8]))
print(hash( {2,8} ))
print(hash( {2:8} ))
print(hash(bytearray(4)))
```

Le set is mutable, but the elements of the set must be immutable / hashable

`{{1,2}, {3,4}}`

But the below set of tuple is possible

```
a = {(1,2), (3,4)}
print(a)
print("=====")
```

```
a = {(1,2), (3,4), (1,2)}
print(a)
```

output

`{(1, 2), (3, 4)}`

```
=====
{(1, 2), (3, 4)}
```

```
-----
print(set((5))) # TypeError: 'int' object is not iterable
print(set((5,))) # {5}
```

Set methods and equivalent operators

The latter operations have equivalent operators as shown below:

Method	Operator
<code>a.intersection(b)</code>	<code>a & b</code>
<code>a.union(b)</code>	<code>a b</code>
<code>a.difference(b)</code>	<code>a - b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>
<code>a.issubset(b)</code>	<code>a <= b</code>
<code>a.issuperset(b)</code>	<code>a >= b</code>

```
setOfBankInfo = {"SBI", "ERTT234_IFSC", "Loganathan_BranchHead", "Nithya_Manager",  
"Valli_IT_Head", 600034, "SB", "FD"}  
print(setOfBankInfo)  
print(type(setOfBankInfo))  
-----  
bank = "STATE BANK OF INDIA"  
print((bank))  
print(len(bank))
```



```
setResult = set("STATE BANK OF INDIA")  
print(setResult)  
print(len(setResult))
```

```
set1 = set( {'Vanitha', 20, True} )  
print(set1)
```

output
{True, 'Vanitha', 20}

add()

```
a = "hello"  
a= (set(a))  
print (a)
```

```
a.add(6) # USING ADD METHOD WE ADD ONLY ONE ELEMENT AT A  
TIME  
print (a)
```

output

```
{'e', 'l', 'h', 'o'}  
{6, 'l', 'e', 'o', 'h'}  
=====
```

Update()

""" Update a set with the union of itself and others. """

```
a = {'A', 'B', 'C'}
```

```
b = {10, 20, 30}
```

```
a.update(b) #ALWASY USE THIS TYPE OF UPDATE, IT DOES UP MIX  
AND MATCH
```

```
print (a)
```

```
a=a.update(b) # DONT ASSIGN THE RESULT TO A VARIBALE AND  
UPDATE, IT GIVE 'NONE' VALUE
```

```
print (a)
```

output

```
{'A', 20, 'B', 10, 30, 'C'}
```

```
None
```

```
=====
```

Updating a set with another set

```
a = {'A', 'B', 'C'}
b = {10, 20, 30, 'Vikki'}
a.update(b) #ALWASY USE THIS TYPE OF UPDATE, IT DOES UP MIX AND MATCH
print (a)
output
{'A', 'Vikki', 'B', 20, 'C', 10, 30}
```

Updating a set with list

```
a = {'A', 'B', 'C'}
b = [10, 20, 30, 'Vikki']
a.update(b) #ALWASY USE THIS TYPE OF UPDATE, IT DOES UP MIX AND MATCH
print (a)
```

```
output
{'C', 10, 'A', 'B', 20, 'Vikki', 30}
```

We can update multiple sets using update()

```
a = {'A', 'B', 'C'}
b = {10, 20, 30, 'Vikki'}
```

```
c = {"Mahe", "Kamal"}  
d = {10, 1000}
```

```
a.update(b,c,d) #ALWAYS USE THIS TYPE OF UPDATE, IT DOES UP MIX AND MATCH  
print (a)
```

output

```
{1000, 'Vikki', 10, 'Kamal', 'Mahe', 'C', 'B', 20, 'A', 30}
```

```
-----  
di= {1:10, 2:20, 3:30, "Name":"Magesh"}  
b= { 10,20,30,'Vikki', ("List", "of", "List"), True, frozenset(di) }  
print(b)
```

output

```
{True, 'Vikki', 10, 20, ('List', 'of', 'List'), frozenset({1, 2, 3, 'Name'}), 30}
```

Note: if we update a dict to set, only the KEYS will be updated to the set

After the update(), the id will not be changed – see below the ex code

```
a = {'A', 'B', 'C'}  
print("ID of a ", id(a))
```

```
b= {10,20,30,'Vikki'}  
print("ID of b ", id(b))
```

```
c = {"Mahe", "Kamal"}  
print("ID of c ", id(c))  
d = {10, 1000}
```

```
a.update(b,c,d) #ALWAYS USE THIS TYPE OF UPDATE, IT DOES UP MIX AND MATCH  
print (a)  
print (id(a)) # NOTE after the update, the id of a WILL NOT be changed
```

output

ID of a 3000404976896

ID of b 3000408968096

ID of c 3000408968768

{'Kamal', 'C', 10, 'Vikki', 'Mahe', 1000, 'B', 20, 'A', 30}

3000404976896

Union() / (operator '|')

a.union(b) returns a new set (new id), from both sets , but duplicate will be removed)

Return the union of sets as a new set.i.e. all elements that are in either set.)

```
a = {'A', 'B', 'C'}  
print(id(a))
```

```
b = {10, 20, 30}  
print(id(b))
```

```
a = a.union(b) # b = b | a
```

```
print(a)  
print(id(a))
```

output

```
1623236650240
```

```
1623240641440
```

```
{'A', 'B', 20, 'C', 10, 30}
```

```
1623240642112
```

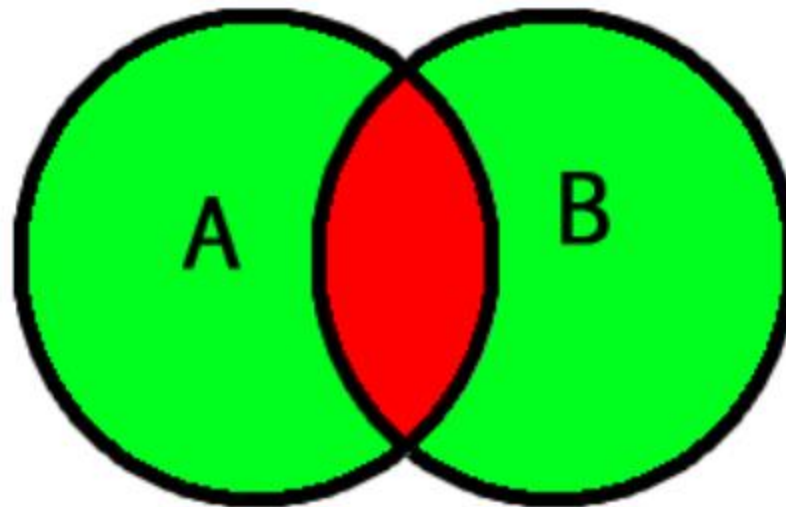
```
=====
```

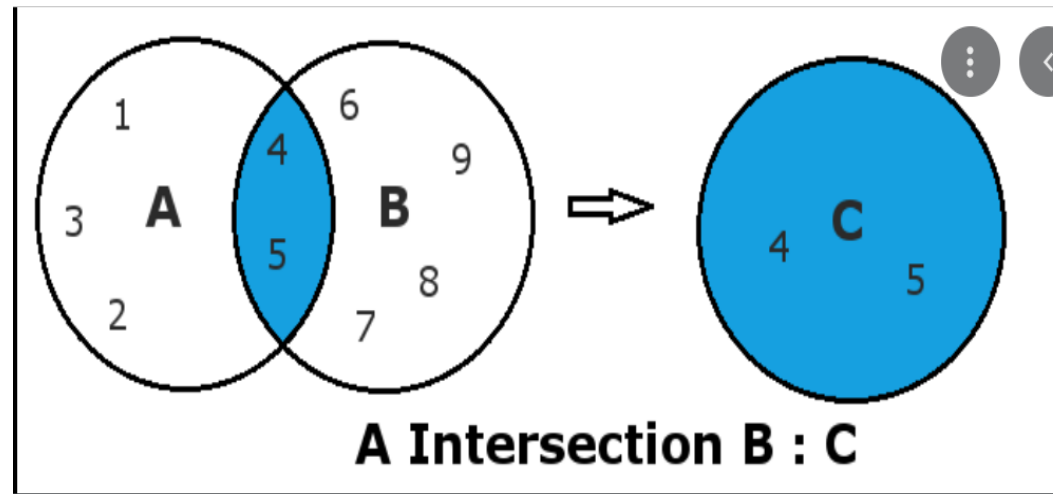
Intersection() / (operator '&')

$$A = \{1,2,3,4\}$$

$$B = \{3,4,5,6\}$$

$$A \cap B = \{3,4\}$$





`a.intersection(b)` returns a new set / (common values from both set)

1. it checks only one sets

`update()`

1. it returns a new set
2. the original will be intact / the items of original set will not be changed

```
a = {'A', 'B', 'C', 'F', 5}
print(id(a))
```

```
b = {'A', 'B', 'C', 'E', 'F'}
print(id(b))
# a.intersection(b)
print(a.intersection(b))
```



```
print(id(a))
print("=====")
print("original set a", a)
print("LEN of original set a", len(a)) # intersection(). keeps the original set
print("+++++")
```

output

```
1551038129408
1551042120608
{'B', 'A', 'F', 'C'}
1551038129408
=====
original set a {'B', 5, 'A', 'F', 'C'}
LEN of original set a 5
+++++
-----
```

intersection_update()

The `intersection_update()` method removes the items that is not present in both sets (or in all sets if the comparison is done between more than two sets)

The Python `intersection_update()` method is used to update the set which contains all common elements of two or more sets. This method can take any number of sets or iterables to compute the intersection set and update the original set with intersection set.

1. It REMOVES all the original set's value
2. Find the common values (as `intersection()`) ie removes the items that is not present in both sets
3. Then ONLY the common values are updated to the original set

```
a = {'A', 'B', 'C', 'F', 5}
print(id(a))
```

```
b = {'A', 'B', 'C', 'E', 'F', 10}
print(id(b))
a.intersection_update(b)
print(id(a))
print("=====")
print("original set a", a)
print("LEN of original set a", len(a)) # intersection_update(). removes the unwanted items /
un common values
```

output

2533253489920

2533257481120

2533253489920

=====

original set a {'A', 'B', 'F', 'C'}

LEN of original set a 4

Pop()

```
a = {'A', 'B', 'C'}
```

```
b= a.pop() #IT REMOVES ARBITRARY ELEMENT (ANY ELEMENT,  
NOT IN ORDER)
```

```
print (b)
```

```
print (a)
```

output

C

{'B', 'A'}

=====

Remove()

Removes the specified element

```
a = {'A', 'B', 'C'}  
a.remove('B')  
print (a)
```

output
{'C', 'A'}

=====

```
a = {'A', 'B', 'C'}  
a= a.remove('B') # DONT ASSIGN THE RESULT TO A VARIBALE AND  
UPDATE, IT GIVES 'NONE' VALUE  
print (a)
```

output
None

=====

Raises key error if we want to remove a non existing item from the set

```
a = {'A', 'B', 'C'}  
a.remove('E')  
print(a)
```

output

KeyError: 'E'

=====

Copy()

```
a = {'A', 'B', 'C'}  
b = {10, 20, 30}  
b= a.copy()  
print(b)
```

output

{'A', 'B', 'C'}

use copy(), before want to remove an element

```
a = {'A', 'B', 'C'}  
a.remove('C')
```

```
b= a.copy() # we use copy(), before we want to remove a element
print(b)
print(a)
print("=====")
```

```
a = {'A', 'B', 'C'}
b= a.copy() # we use copy(), before we want to remove a element
a.remove("C")
print(b)
print(a)
```

```
{'B', 'A'}
{'B', 'A'}
=====
{'B', 'C', 'A'}
{'B', 'A'}
=====
```

Clear()

```
setA = {'A', 'B', 'C'}
setB = {10, 20, 30}
setA.clear() # AFTER CLEAR THE ELEMENTS OF THE SET, it gives the empty set
```

```
print (setA) # IF WE PRINT THE SET NOW IT SHOWS EMPTY SET
```

output

set()

=====

Difference() / Operator ('-')

The **difference()** method returns a set that contains the difference between two sets. **Meaning: The returned set contains items that exist only in the first set, and not in both sets.**

Return the difference of two or more sets as a new set.

i.e. all elements that are in this set but not the others.)

2. **a.difference(b)** returns a new set with elements present in 'a' but not in other sets

3. but the len of the set is same. That other items in the set will be there . other items will not be removed.
4. the difference() **method returns a NEW set**
5. **it checks only one sets**

```
a = {'A', 'B', 'C', 10, 20}
print("LENGTH of set a BEFORE difference update ", len(a))
b = {'A', 'B', 'C', 'D', 'E'}
c = {'A', 'B', 'C', 200, 300}

print("=====")

# print(a.difference(b,c)) # possible
d = a.difference(b,c)
print(d)
print("After diffence(), the values in set a is NOT REMOVED", a)
print("LENGTH of sete a AFTER difference update ", len(a))
print(b)
print(c)
```

output

```
LENGTH of set a BEFORE difference update 5
=====
{10, 20}
After diffence(), the values in set a is NOT REMOVED {20, 'C', 10, 'B', 'A'}
```


LENGTH of set a AFTER difference update 5

{'B', 'E', 'C', 'D', 'A'}

{'C', 200, 'B', 300, 'A'}

=====

Difference between DIFFERENCE() and DIFFERENCE_UPDATE()

Difference_update()

The difference_update() method **removes the items that exist in both sets**. The difference_update() method is different from the difference() method, because the difference() method returns a new set, without the unwanted items, and the difference_update() method removes the unwanted items from the original set.

```
a = {'A', 'B', 'C', 10, 20}
print("id of a ", id(a))
print("LENGTH of set a BEFORE difference update ", len(a))
b = {'A', 'B', 'C', 'D', 'E'}
```

```
c = {'A', 'B', 'C', 200, 300, 10}
```

```
print("=====")
```

```
a.difference_update(b,c) # d = a.difference_update(b,c) NOT possible
```

```
print(a)
```

```
print("id of a ", id(a))
```

```
print("=====")
```

```
print("After diffence_update(), the unique values in set a is retained", a)
```

```
print("LENGTH of sete a AFTER difference update ", len(a))
```

```
print(b)
```

```
print(c)
```

output

LENGTH of set a BEFORE difference update 5

=====

{20, 10}

After diffence_update(), the values in set a is REMOVED {20, 10}

LENGTH of sete a AFTER difference update 2

{'E', 'A', 'C', 'B', 'D'}

{'A', 'C', 200, 300, 'B'}

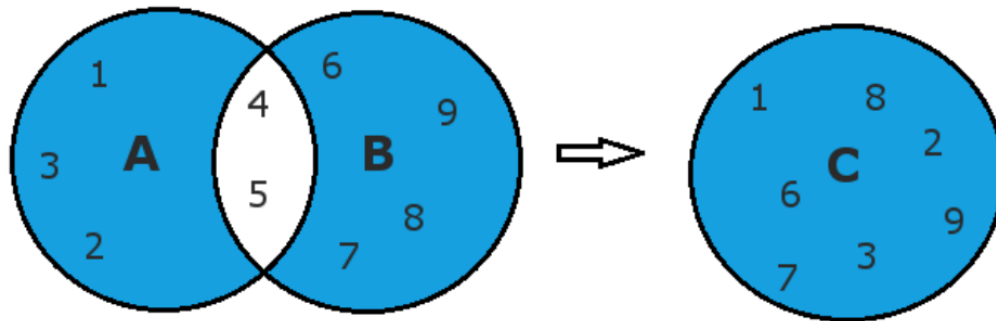
=====

symmetric_difference() / Operator('^')

a.symmetric_difference(b) **removes common items from both sets** and returns other elements in a **NEW set**

The symmetric_difference() method returns a set that contains all items from both set, but not the items that are present in both sets. Meaning: The returned set contains a mix of items that are not present in both sets.

it checks BOTH sets (which is not possible in intersection() and difference())



A Symmetric Difference B : C

```
a = {'A', 'B', 'C', 10, 20} # checks both sets
b = {'A', 'B', 'C', 'D', 'E',} # checks both sets
```

```
print("ID of set A before SYMMETRIC_DIFFERENCE() is", id(a))
print("ID of set b before SYMMETRIC_DIFFERENCE() is", id(b))
c = b.symmetric_difference(a)
print(c)
```

```
print("set a values", a)
print("set b values", b)
print("ID of set c after SYMMETRIC_DIFFERENCE() is", id(c))

print('=' * 50)
```

output

```
ID of set A before SYMMETRIC_DIFFERENCE() is 1914950752968
ID of set b before SYMMETRIC_DIFFERENCE() is 1914953512552
{20, 'D', 'E', 10} # show the odd / different values from BOTH sets, old values
from both sets are as it is ...
set a values {'B', 10, 'C', 20, 'A'}
set b values {'D', 'E', 'B', 'C', 'A'}
```

ID of set c after SYMMETRIC_DIFFERENCE() is 1914953513896 # this is new set

MELCOSE'S NOTE: a.symmetric_difference(b) == b.symmetric_difference(a)
=====

SYMMETRIC_DIFFERENCE_UPDATE() operator(^=)

This method is used to return the symmetric difference of a set and the set of elements from the iterable like string, list, set passed as an argument. It is very similar to symmetric_difference() method, with difference is that where symmetric_difference() method create and return a new set.

symmetric_difference_update() method updates the set on which this method is invoked, the id of the original set will not be changed

```
a = {'A', 'B', 'C', 10, 20} #checks both sets  
b = {'A', 'B', 'C', 'D', 'E',} #checks both sets
```

```

b.symmetric_difference_update(a)
print("=====")
print("SET a is", a)
print("SET b is", b) # common value are A,B,C are gone
print("LEN AFTER difference", len(b) )

```

output

SET a is {20, 'C', 'B', 10, 'A'}

SET b is {'D', 20, 'E', 10}

LEN AFTER difference 4

1. Melcose's Note: it returns uncommon / odd values from both sets (common values are avoided from both sets)

Also, it REMOVES the common elements from set a or set b (based on what set we want to apply symmetric_difference_update). # show the odd / different values from BOTH sets, old values are REMOVED based on, the set that we want to perform...

2. ie, if we do, a.symmetric_difference_update(b) it removes common elements in set a
3. ie, if we do, b.symmetric_difference_update(a) it removes common elements in set b

=====

Subset and superset (Operator $a \leq b, a \geq b$)

The `issuperset()` method returns `True` if a set has every elements of another set (passed as an argument). If not, it returns `False`.

In mathematics, a set A is a subset of a set B if all elements of A are also elements of B;

B is then a superset of A. It is possible for A and B to be equal; if they are unequal, then A is a proper subset of B. ... The subset relation defines a partial order on sets.

```
a = {'A', 'B', 'C', 10, 20}
b = {'A', 'B', 'C', 'D', 'E', 10, 20}
```

```
print(a.issubset(b))
print(b.issuperset(a))
```

output

True

True

a.issubset(b) tests whether each element of a is in b.

b.issuperset(a) tests whether each element of a is in b.

```
a = {'A', 'B', 'C', 11,20}
```

```
b = {'A', 'B', 'C', 'D', 'E',10,20}
```

```
print(a.issubset(b))
```

```
print(b.issuperset(a))
```

output

False

False

What happens if both sets have the same elements

```
B = {1, 2, 3}
```

```
C = {1, 2, 3}
```



```
print(B.issubset(C))
print(C.issubset(B))
print("=====")
```

```
print(B.issuperset(C))
print(C.issuperset(B))
```

Output

True

True

=====

True

True

What happens when set1 and set2 have same elements?

What happens if both set have same elements?

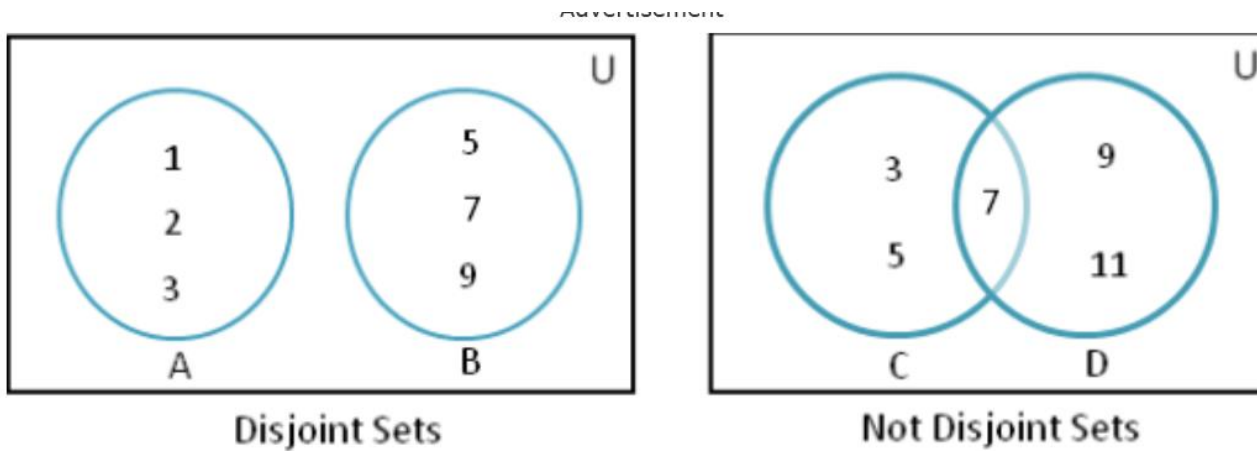
Ans: True for both issuperset() and issubset()

=====

Disjoint sets

Python Set `isdisjoint()` Method

The `isdisjoint()` method returns True if none of the items are present in both sets, otherwise it returns False.



```
B = {10, 20, 30}
C = {1, 2, 3}
print(B.isdisjoint(C))
print(C.isdisjoint(B))
```

Output

True
True

=====

Testing membership in set

The builtin **in** keyword searches for occurrences

```
B = {10, 20, 30}
```

```
print(20 in B)
```

```
print(2 in B)
```

Output

True
False

=====

Set An **unordered** collection of **unique** values. **Items must be hashable/immutable.**

Set DOES NOT support indexing

A **set does** not hold **duplicate** items. The objects(values/items) of the **set** are **immutable** ,

A set is a collection of elements with **no repeats** and without **insertion order** but **sorted order**. They are used in situations where it is only important that some things are grouped together, and **not what order they were included**. For large groups of data, **it is much faster to check whether or not an element is in a set, than it is to do the same for a list**.

Defining a set is very similar to defining a dictionary:

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a set using an existing list:

```
my_list = [1,2,3]
```

```
my_set = set(my_list)
```

Check membership of the set using in:

```
if name in first  
    print(name)
```

You can iterate over a set exactly like a list, but remember: **the values will be in an arbitrary (random), implementation defined order.**

```
=====
setA = {"CCC", "BBBB", "AAAAA"}
```

```
for index, item in enumerate(setA):
    print(index, item)
    print(index, len(item))
```

output

1 AAAAA

1 5

2 BBBB

2 4

=====

Frozenset()

The **frozenset()** function returns an **immutable frozenset** object initialized with elements from the given iterable. **Frozen set** is just an immutable version of a **Python set** object. While elements of a **set** can be modified at any time, elements of the **frozen set** **remain the same after creation**

frozenset() Parameters

The frozenset() function takes a single parameter:

- **iterable (Optional)** - the iterable which contains elements to initialize the frozenset with.

Iterable can be set, dictionary, tuple, etc.

ie, any mutable object can be converted into immutable using frozenset

(To add an element to a frozenset , first convert the frozenset to a list, then append) //

```
set = {10, 20, 20, "AA", "BB", "CC"}
```

```
dic = {10:"Ten", 20:"Twenty", 30:"Thirty", "AA":"aa"}
```

```
lst = [10, 20, 20, "AA", "BB"]
```

```
# print(hash(set))
```

```
fset = frozenset(set)
```

```
print(hash(fset))
```

converting a dict to frozenset, gives only the keys of the dict, NOT the values

```
dic = {10:"Ten", 20:"Twenty", 30:"Thirty", "AA":"aa"}
```

```
print(frozenset(dic))
```

output

```
frozenset({'AA', 10, 20, 30})
```

```
frozenDic = frozenset(dic)
```

```
print(hash(frozenDic))
```

```
frozenList = frozenset(lst)
```

```
print(hash(frozenList))
```

```
lst = list(frozenList) # convert the frozen set into list, then append new item
```

```
lst.append("ZZZZZ")
```

```
frozenList = frozenset(lst)
```

```
print(frozenList)
```

output

```
-72879352744714435
```

```
-6742259836630986438
4962448526265101559
frozenset({'AA', 10, 'BB', 20, 'ZZZZZ'})
=====
```

Return value from frozenset()

The frozenset() function returns an immutable frozenset initialized with elements from the given iterable.

If no parameters are passed, it returns an empty frozenset.

```
-----
print(frozenset(set()))
```

o/p:

```
frozenset()
```

```
=====
```

```
print(frozenset("LINDA"))
```

output

```
frozenset({'L', 'N', 'I', 'D', 'A'})
```

```
-----
```



```
vowels = ('a', 'e', 'i', 'o', 'u')
fSet = frozenset(vowels) # BULIT IN CONSTRUCTOR 'frozenset'
CONVERTS THE TUPLES 'vowels' INTO FROZEN SET
print (fSet)
fSet.add ('v') # NOT possible

print (type(frozenset))
```

```
output
frozenset({'a', 'e', 'u', 'o', 'i'})
<class 'frozenset'>
=====
```

Discard()

no error is given if try to remove non existing item

```
set = {10, 20, 20, "AA", "BB", "CC"}
print(set)

set.discard("AA")
```

```
print(set)
print("NO ERROR SHOWN even if we try to discard an item which is
not available in the set")
set.discard("ZZ"># trying to remove an item which is not available in the set
print(set)
```

output

```
{10, 20, 'BB', 'AA', 'CC'}
```

```
{10, 20, 'BB', 'CC'}
```

NO ERROR SHOWN even if we try to discard an item which is not available in the set

```
{10, 20, 'BB', 'CC'}
```

Another example:

Notes : Discard()

Discard **will not raise KeyError**, even if we try to remove a non member of an iterable(set).

Even after remove an element, the IDs will not change, remain same.

```
class SetClass:
```

```
    def sMethod(self, fruitsSet1, fruitsSet2):  
        print('Original set', fruitsSet1)  
        print('Before discard ', id(fruitsSet1))  
        fruitsSet1.discard('banana')  
        print('After discard ', id(fruitsSet1))  
        # print(fruitsSet1)  
        return fruitsSet1  
  
    # return fruitsSet2.difference(fruitsSet1)
```

```
fruits1 = {10, 20, 'banana', 'apple', 'tange1'}  
fruits2 = {10, 20, 'banana', 'apple', 'tangerine', }
```

```
fruitSetObj = SetClass()  
result = fruitSetObj.sMethod(fruits1, fruits2)  
print(result)
```

o/p:

```
Original set {'banana', 10, 'apple', 'tange1', 20}  
Before discard 1903185957600  
After discard 1903185957600  
{10, 'apple', 'tange1', 20}
```

=====

Remove()

Remove() SHOWS ERROR if we try to remove an item/element which is not available in the set

```
set = {10, 20, 20, "AA", "BB", "CC"}  
print(set)
```

```
set.remove("BB")  
print(set)
```

output

```
{'AA', 10, 20, 'BB', 'CC'}  
{'AA', 10, 20, 'CC'}
```

```
set = {10, 20, 20, "AA", "BB", "CC"}  
print(set)
```

```
print("ERROR SHOWN even if we try to remove an item which is not  
available in the set")
```

```
set.remove("ZZ")  
print(set)
```

output is error

```
set.remove("ZZ")
```

KeyError: 'ZZ'

Notes: only difference between discard() and remove() is, KeyError is shown when we try to remove non member from the set(when we use remove()).

Another example:

```
class SetClass:
```

```
    def sMethod(self, fruitsSet1, fruitsSet2):
        print('Original set', fruitsSet1)
        print('Before remove ', id(fruitsSet1))
        fruitsSet1.remove('banana1')
        print('After remove ', id(fruitsSet1))
        # print(fruitsSet1)
        return fruitsSet1

    # return fruitsSet2.difference(fruitsSet1)
```

```
fruits1 = {10,20,'banana', 'apple', 'tange1'}
fruits2 = {10,20,'banana', 'apple', 'tangerine', }
```

```
fruitSetObj = SetClass()
result = fruitSetObj.sMethod(fruits1, fruits2)
print(result)
```

o/p:

Original set {10, 'banana', 'tange1', 20, 'apple'}

Before remove 1922303333088

Traceback (most recent call last):

File "C:/Users/ Arul/PycharmProjects/pythonProject1/Jessca.py", line 18, in
<module>

result = fruitSetObj.sMethod(fruits1, fruits2)

File "C:/Users/Jessca Arul/PycharmProjects/pythonProject1/Jessca.py", line 6,
in sMethod

fruitsSet1.remove('banana1')

KeyError: 'banana1'

=====

In mathematics, a set A is a **subset** of a set B if all elements of A are also elements of B; B is then a **superset** of A. It is possible for A and B to be equal; if they are unequal, then A is a proper **subset** of B. ... The **subset** relation defines a partial order on sets.

```
set1 = set()
set1.add(5)
print(set1)

print("=====")
set1 = set()
set1.add( () )
print(set1)

print("=====")
set1 = set()
set1.add( [] ) #TypeError: unhashable type: 'list'
print(set1)
```

Notes: only hashable elements can be added to a set. [] is unhashable

Update()

```

set = {10, 20, 20, "AA", "BB", "CC"}
print(set)

# set.update("H")
# set.update("Sudha")#{'AA', 'a', 10, 20, 'CC', 'BB', 'H', 'u', 'S', 'd', 'h'}
print(set)
# set.update(["Sudha"])#{'AA', 'a', 10, 20, 'CC', 'BB', 'H', 'u', 'S', 'd', 'h'}
set.update("MMMM")

print(set)

```

output

```

{10, 'CC', 'AA', 20, 'BB'}
{'M', 10, 'CC', 'H', 'AA', 20, 'BB'}

```

Melcose's Note: observe, we give an item "MMMM" to update, since set does not accept duplicates, it accepts just only one 'M', out of "MMMM" sequence

If we want to add "MMMM" as one whole string to a set, first declare a string as a collection type of Tuple or frozenset. Then, use add or update method.

What happens if we pass a string to set or frozenset. It splits the string to characters and remove the duplicates. // Selva


```
setOfFrozenSets = { frozenset({100,200}), frozenset({1000,2000}, ) } # possible
name = ('Jess',)
setOfFrozenSets.update((1,2), 'Jess')
print(setOfFrozenSets)
```

o/p:

```
{1, 2, 'i', 'j', 'e', frozenset({1000, 2000}), 's', frozenset({200, 100})}
```

Notes: below we pass a string to set/ frozenset, so same result as above.

```
setOfFrozenSets = { frozenset({100,200}), frozenset({1000,2000}) } # possible
setOfFrozenSets.update((1,2), ('Jess'))
print(setOfFrozenSets)
```

o/p:

```
{1, 2, 'i', 'j', 'e', frozenset({1000, 2000}), 's', frozenset({200, 100})}
```

```
setOfFrozenSets = { frozenset({100,200}), frozenset({1000,2000}), } # possible
setOfFrozenSets.update(( (1,2), 'Jess'), ) # if we give an item INSIDE the tuple it takes as an
whole item
print(setOfFrozenSets)
print("=====")
```

```
setOfFrozenSets.update(( (1,2), 'Jess'), "Sudha" ) # if we give an item INSIDE the tuple it
```

takes as an whole item

```
print(setOfFrozenSets)
```

output

```
{frozenset({1000, 2000}), (1, 2), 'Jess', frozenset({200, 100})}
```

```
=====
```

```
{'S', (1, 2), 'h', 'Jess', 'u', 'a', 'd', frozenset({1000, 2000}), frozenset({200, 100})}--
```

```
-----
```

Notes: see below, we pass a string as a tuple to a set/frozenset(watch the comma(,) the comma makes a one element of string to a tuple)

```
setOfFrozenSets = { frozenset({100,200}), frozenset({1000,2000}) } # possible
```

```
setOfFrozenSets.update((1,2), ('Jess',))
```

```
print(setOfFrozenSets)
```

o/p:

```
{1, 2, frozenset({1000, 2000}), 'Jess', frozenset({200, 100})}
```

```
-----
```

Another code

```
setOfFrozenSets = { frozenset({100,200}), frozenset({1000,2000}), } # possible
setOfFrozenSets.update(( (1,2), 'Jess'), ) # if we give an item INSIDE the tuple it takes as an
whole item
print(setOfFrozenSets)
print("=====")
```

```
setOfFrozenSets.update(( (1,2), 'Jess'), ("Sudha",) ) # if we give an item INSIDE the tuple it
takes as an whole item
print(setOfFrozenSets)
```

output

```
{frozenset({1000, 2000}), (1, 2), 'Jess', frozenset({200, 100})}
=====
{(1, 2), 'Jess', 'Sudha', frozenset({1000, 2000}), frozenset({200, 100})}
```

Then how to update a string into set, create a new set, then update.(see below the example), don't use add () it gives error

```
=====
```

A set can be updated with another set, provided the second set elements will individually added in the first set

```
set1 = {10, 20, 20, "AA", "BB", "CC"}
print(set)
set2 = {"Lion", "Tiger"}
```

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

```
output  
<class 'set'>  
{'Lion', 10, 'AA', 'Tiger', 20, 'BB', 'CC'}  
=====
```

Get the unique elements of a list using set

After doing ANY operation in the list, again convert into set.

Set of Sets

Set of frozenset()

set of sets is NOT POSSIBLE as set itself is UNHASHABLE (is mutable)
set is unhashable (mutable), (i.e) elements can be modified. But, elements of a set is hashable (immutable (i.e)). That's why we are not able to add set and list to a set.

```
setOfSets = { {100, 200}, {1000, 2000} } #impossible  
print("set of sets, but set itself is unhashable, so we can not keep a set  
into another set")
```

output

```
setOfSets = { {100, 200}, {1000, 2000} } #impossible
```

TypeError: unhashable type: 'set'

=====

set of frozensets is POSSIBLE as frozenset itself is HASHABLE (is immutable)
see below the example

```
set1 = {1, 2, "Lara", frozenset({200})} #possible  
print("Set can have only HASHABLE OBJECT ", set1)
```

```
setOfFrozenSets = { frozenset({100, 200}), frozenset({1000, 2000}) } #
```

possible

```
print("inside the set, we gave 2 frozen sets, since frozenset is  
HASHABLE AND THIS RIGHT", setOfFrozenSets)
```

output

Set can have only HASHABLE OBJECT {1, 2, frozenset({200}), 'Lara'}
inside the set, we gave 2 frozen sets, since frozenset is HASHABLE AND THIS
RIGHT {frozenset({1000, 2000}), frozenset({200, 100})}

=====

=====

Collections module and Counter

Sets versus multiset

Sets are unordered collections of **distinct elements**. But sometimes we want to work with unordered collections of elements that are not necessarily distinct and keep track of the elements' multiplicities.

Consider this example:

```
bankCustomers = {'Gomathi', 'Anto', 'Emalta', 'Kamal', 'Gomathi', 'Mahesh', 'Usha',  
                'Kamal', 'Viji', 'Usha', 'Nithya', 'Nithya' 'Nithya'}  
print (bankCustomers)
```

output

```
{'Mahesh', 'Kamal', 'Usha', 'Gomathi', 'NithyaNithya', 'Viji', 'Anto', 'Emalta',  
'Nithya'}
```

```
setA = {'a', 'b', 'b', 'c'}  
print (setA)
```

out put

```
{'c', 'b', 'a'} // we lost one element of b (since it is duplicate, set removes the  
duplicate element
```

By saving the strings 'a', 'b', 'b', 'c' into a set data structure we've lost the information on the fact that 'b' occurs twice. Of course saving the elements to a list would retain this information

To avoid this we use list (list accepts duplicates)

```
lst=['a', 'b', 'b', 'c']  
print (lst)
```

output
['a', 'b', 'b', 'c']

but a list data structure introduces an **extra unneeded ordering that will slow down our computations.**

For implementing multisets Python provides the **Counter class from the collections module** (starting from version 2.7):

```
# import collections as co  
from collections import Counter  
bankCustomers = ["Gomathi", "Anto", "Emalta", "Kamal", "Gomathi",  
"Mahesh", "Usha", "Kamal", "Viji", "Usha", "Nithya",  
"Nithya", "Nithya"]  
b = Counter(bankCustomers)  
print(b)
```

Output

```
Counter({'Nithya': 3, 'Gomathi': 2, 'Kamal': 2, 'Usha': 2, 'Anto': 1, 'Emalta': 1,  
'Mahesh': 1, 'Viji': 1})
```


Note :

THE Counter TOKEN COUNTS THE DUPLICATE ELEMENTS AND SHOW AS SET

Counter is a dictionary where elements are stored as dictionary keys and their counts are stored as dictionary values.

Convert the frozenset to set

```
a={'A','B','C',10,20}
```

```
aF=frozenset(a)
```

```
aS=set(aF)
```

```
print(type(aF))
```

```
print(aS)
```

```
print(type(aS))
```

```
=====
```

Counter()

```
def __init__(self, __iterable: None = ..., **kwargs: int) -> None
```

Possible types:

- (self: Counter, **__iterable: None**, **kwargs: Dict[str, int]**) -> None
- (self: Counter, **__mapping: Mapping[_T, int]**) -> None
- (self: Counter, **__iterable: Iterable[_T]**) -> None

Create a new, empty Counter object. And if given, count elements from an input iterable. Or, initialize the count from another mapping of elements to their counts.

```
c = Counter()           # a new, empty counter
c = Counter('gallahad') # a new counter from an iterable
c = Counter({'a': 4, 'b': 2}) # a new counter from a mapping / dict
c = Counter(a=4, b=2)     # a new counter from keyword args
```

from collections **import** Counter

```
c1 = Counter()           # empty counter from Counter ()
print(c1)
print("=====")
```

```
c2 = Counter('SsDDDDHHHAAAA')           # a new counter from an iterable
print(c2)
print(type(c2))
print("=====")
#
```

```
c3 = Counter({'a': 4, 'b': 2})           # a new counter from a mapping / dict
print(c3)
```

```
print("=====")
```

```
# a new counter from keyword args  
c4 = Counter(age = 25, marks = 95)  
print(c4)  
print("=====")
```

a new counter from keyword args

output

```
Counter()  
=====  
Counter({'A': 4, 'D': 3, 'H': 3, 'S': 1, 's': 1})  
<class 'collections.Counter'>  
=====  
Counter({'a': 4, 'b': 2})  
=====  
Counter({'marks': 95, 'age': 25})  
=====
```

```
from collections import Counter  
a = Counter(['H', 'E', 'L', 'L', 'O', "Lara", 10, 20, 10, "Lara"])  
print(a)
```

output

```
Counter({'L': 2, 'Lara': 2, 10: 2, 'H': 1, 'E': 1, 'O': 1, 20: 1})
```

Melcose's Note: Though the output look likes dict, **it is muliset in the form dict**

=====

Counter is a dictionary where elements are stored as dictionary keys and their counts are stored as dictionary values.

Melcose's Note:The reason we use Counter for creating multiset is, set removes the duplicates but **WE WANT** the duplicates

See below, even we use Counter(), still, set removes all the duplicates. So we want create MULTISSET using Counter

```
from collections import Counter
a = Counter(['H', 'E', 'L', 'L', 'O', "Lara", 10, 20, 10, "Lara"])
print(set(a))
```

output

```
{'L', 'O', 'H', 10, 'Lara', 'E', 20}
```

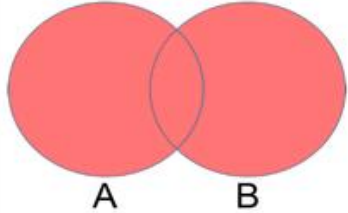
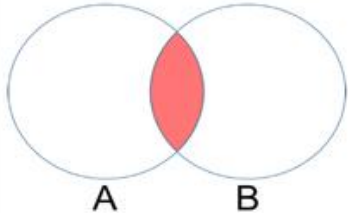
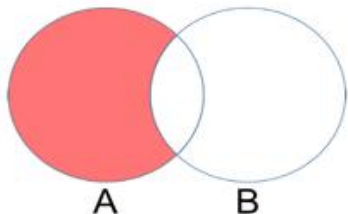
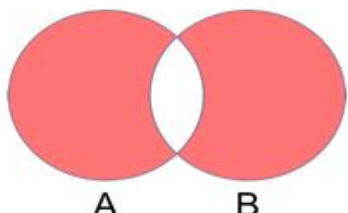
=====

Find the type() of Counter

```
from collections import Counter
a = Counter(['H', 'E', 'L', 'L', 'O', "Lara", 10, 20, 10, "Lara"])
print(a)
print(type(a))
output
Counter({'L': 2, 'Lara': 2, 10: 2, 'H': 1, 'E': 1, 'O': 1, 20: 1})
<class 'collections.Counter'>
```

=====

Recalling

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$, is the set of all values that are a member of A , or B , or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B .
Difference		$A \setminus B$, is the set of all values of A that are not members of B
Symmetric Difference		$A \triangle B$, is the set of all values which are in one of the sets, but not both.

