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

NOTICE

www.DataScienceInTamil.com

- 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:

Python Notes For Professionals.pdf – this is the book we follow
https://docs.python.org
https://www.programiz.com/python-programming/set

What to cover today

- Topic : SET
- Properties of Set
- Set – Built in methods
- Set methods and equivalent operators
  - add()
  - Update()
  - Union() (operator '|')
  - Pop()
  - Remove()
  - Discard()
  - Copy()
  - Clear()
  - Difference()
  - Difference_update()
  - symmetric_difference()
  - SYMMETRIC_DIFFERENCE_UPDATE()
  - Subset and superset (Operator a<=b, a>=b)
  - What happens if both sets has the same elemetns
  - Disjoint sets
  - Testing membership in set

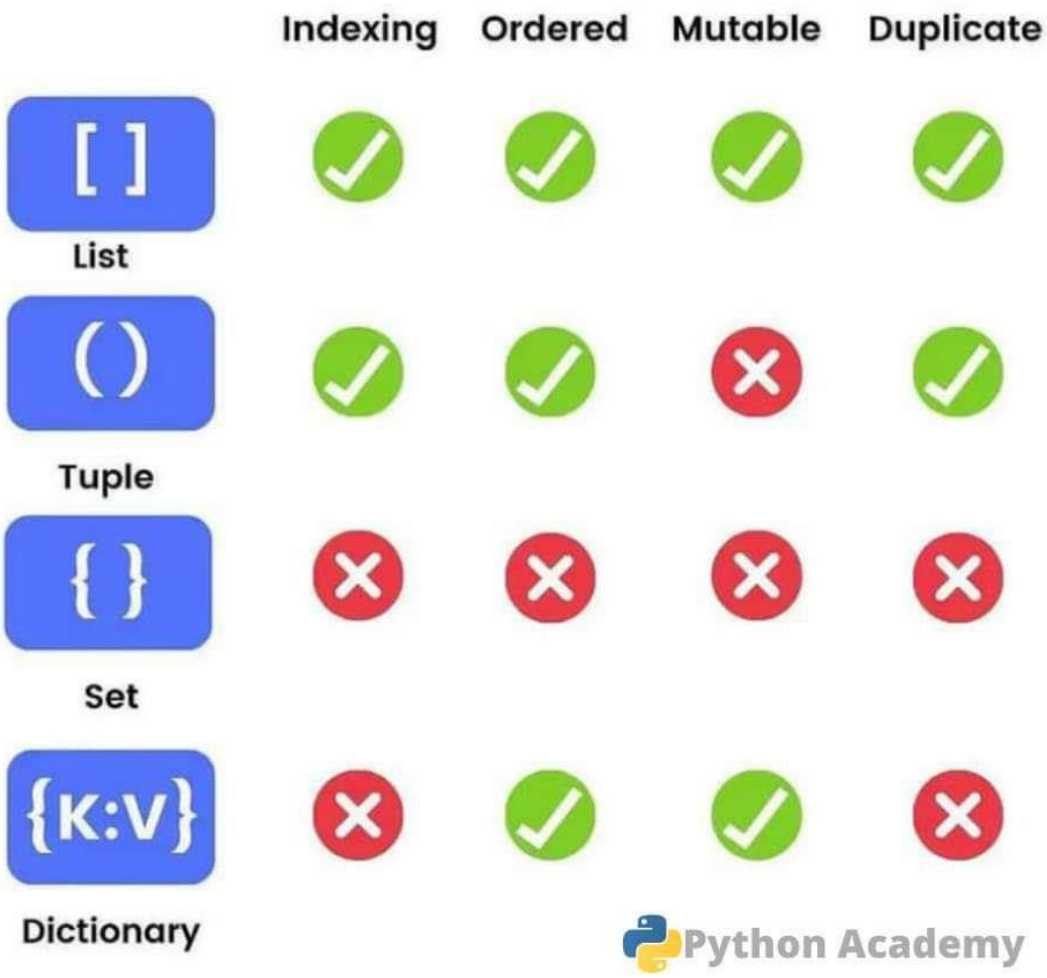
- **Set DOES NOT support indexing, why?**
- **Frozenset()**
- **Return value from frozenset()**
- **Collections module and Counter**
  - **Sets versus multiset**
  - **Set of Sets**
  - **Counter()**

Topic : SET

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)

Python Data Structures



Note:frozen set are immutable

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

Set of Sets (not possible)
TypeError: unhashable type: 'set

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

Ie set is mutable, but the elements of the set must be immuatbale / 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}
```

Instead, use frozenset:

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

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)          # duplicates will be removed
{'orange', 'banana', 'pear', 'apple'}
======
```

Set methods and equivalent operators

The latter operations have equivalent operators as shown below:

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

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))
----------------
```

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 UDATE, IT GIVE 'NONE' VALUE
print (a)
```

output
{'A', 20, 'B', 10, 30, 'C'}
None
==========

Union() / (operator '|')

a.union(b) returns a new set, 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'}
b= {10,20,30}
a=a.union(b) # b=b|a

print (a)
```

output

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

Intersection() / (operator '&') and intersection_update()

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

1. it checks only one sets

```
a = {'A', 'B', 'C', 'F', 5}
b = {'A', 'B', 'C', 'E', 'F'}
print(a.intersection(b))
print("============")
print("original set a", a)
print("LEN  of original set a", len(a))# intersection(). keeps the original set
print("+++++++++++++++")

a = {'A', 'B', 'C', 'F', 5}
b = {'A', 'B', 'C', 'E', 'F'}
a.intersection_update(b)
print("============")
print("original set a", a)
print("LEN  of original set a", len(a)) # intersection_update(). removes the unwanted items / un common values
```

output

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

==========
original set a {'F', 'A', 'B', '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 UDATE, IT GIVES 'NONE' VALUE
print (a)
```

output
None
=======
Raises key error if we want to remove a non exisiting 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 ELEMNETS OF THE SET, it gives the empty set
print (setA) # IF WE PRINT THE SET NOW IT SHOWS EMPTY SET
```

```
output
set()
========
```

Difference() / Operator ('-')

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 sete a AFTER difference update  5
{'B', 'E', 'C', 'D', 'A'}
{'C', 200, 'B', 300, 'A'}

=========
```

Difference between DIFFERENCE() and DIFFERENCE_UPDATE()

Difference_update()

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

a.difference_update(b,c) # d = a.difference_update(b,c) NOT possible
print(a)
print("After diffence_update(), the values in set a is  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
==========
{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

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

```
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()

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<=b, a>=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 has 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 occurances

```
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 idex, item in enumerate(setA):
    print(idex, item)
    print(idex, 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 add new item
lst.append("ZZZZZ")

frozenList = frozenset(lst)
print(frozenList)
output

-7287935274471443 5
-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 unavailable 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/Jessica Arul/PycharmProjects/pythonProject1/Jessica.py", line 18, in <module>
    result = fruitSetObj.sMethod(fruits1, fruits2)
  File "C:/Users/Jessica Arul/PycharmProjects/pythonProject1/Jessica.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

Set operations return new sets, but have the corresponding in-place versions:

method	in-place operation	in-place method
union	s |= t	update
intersection	s &= t	intersection_update
difference	s -= t	difference_update

symmetric_difference s ^= t symmetric_difference_update

For example:

```
s = {1, 2}
s.update({3, 4})    # s == {1, 2, 3, 4}
```

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 and 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.

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

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

output

{frozenset({1000, 2000}), (1, 2), 'jessi', frozenset({200, 100})}
==========
{'S', (1, 2), 'h', 'jessi', '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), ('jessi', ))
print(setOfFrozenSets)
```

o/p:

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

Another code

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

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

output

```
{frozenset({1000, 2000}), (1, 2), 'jessi', frozenset({200, 100})}
===========
{(1, 2), 'jessi', '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 individualy 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})}
```

=====

==========

Set Operations using Methods and Builtins

We define two sets a and b
a = {1, 2, 2, 3, 4}
b = {3, 3, 4, 4, 5}

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().

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 avild 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('SsDDDHHHAAAA')                    # 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)              # a new counter from keyword args
print(c4)
print("===========")
```

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 liks 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 MULTISET 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'>
```

=======