Python Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

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.

```
# Different types of sets in Python
# set of integers
     >>> my_set = {1, 2, 3}
     >>> print(my_set)   # Output: {1, 2, 3}
# set of mixed data types
     >>> my_set = {1.0, "Hello", (1, 2, 3)}
     >>> print(my_set)   # Output: {1.0, (1, 2, 3), 'Hello'}
# set cannot have duplicates
     >>> my_set = {1, 2, 3, 4, 3, 2}
```

```
    >>> print(my_set)   # Output: {1, 2, 3, 4}

# we can make set from a list using set function
    >>> my_set = set([1, 2, 3, 2])
    >>> print(my_set)        # Output: {1, 2, 3}

# set cannot have mutable items. here [3, 4] is a mutable list
# this will cause an error.
    >>> my_set = {1, 2, [3, 4]}
    Output: Traceback (most recent call last):
     File "<string>", line 15, in <module>
     my_set = {1, 2, [3, 4]}
    TypeError: unhashable type: 'list'
```

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() function** without any argument.

```
# Distinguish set and dictionary while creating empty set
# initialize a with {}
    >>> a = {}
# check data type of a
    >>> print(type(a))     # Output : <class 'dict'>
# initialize a with set()
    >>> a = set()
# check data type of a
    >>> print(type(a))    # Output: <class 'set'>
```

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

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)        # Output: {1, 3}
    # add an element in the existing set
    >>> my_set.add(2)
    >>> print(my_set)        # Output: {1, 2, 3}
    # add multiple elements in an existing set
    >>> my_set.update([2, 3, 4, 5])
    >>> print(my_set)        # Output: {1, 2, 3, 4, 5}
    # add list and set in an existing set
    >>> my_set.update([4, 5], {1, 6, 8})
    >>> print(my_set)   # Output: {1, 2, 3, 4, 5, 6, 8}
    # set of vowels
    >>> vowels = {'a', 'e', 'u'}
    # a tuple ('i', 'o')
    >>> tup = ('i', 'o')
    # adding tuple in set as a single element
    >>> vowels.add(tup)
    >>>  print(vowels)        #  Output:  {'u',  'a',  'e',  ('i',
'o')}
    # adding tuple element individually in set
    >>> vowels.update(tup)
    >>>                                            print(vowels)
#Output:{'a',('i','o'),'u','i','o','e'}
```
As set object does not support indexing you will get an error if you want to access set with index.

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

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).The following example will illustrate this.

```
# Difference between discard() and remove()

    # initialize my_set
    my_set = {1, 3, 4, 5, 6}
    >>> print(my_set)       # Output: {1, 3, 4, 5, 6}
    # discard an element
    >>> my_set.discard(4)
    >>> print(my_set)       # Output: {1, 3, 5, 6}
    # remove an element
    >>> my_set.remove(6)
    >>> print(my_set)       # Output: {1, 3, 5}
    # discard an element not present in my_set
    >>> my_set.discard(2)
    >>> print(my_set)       # Output: {1, 3, 5}
    # remove an element not present in my_set
    >>> my_set.remove(2)         # Output: KeyError
```

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. But once you assign the set pop() remove the first element and return it.

We can also remove all the items from a set using the **clear()** method.

```
# initialize my_set
a=set('HelloWorld')
>>> print(a)          # Output:{'H','l','W','o','e','r','d'}
# pop an element
>>> x=a.pop()
>>> print(x)         # Output: 'H'
>>> print(a)         # Output: {'l','W','o','e','r','d'}
# pop another element
>>> x=a.pop()
>>> print(x)         #Output: 'l'
>>> print(a)         # Output; {'W', 'o', 'e', 'r', 'd'}
# clear set a
>>> a.clear()
>>> print(a)         # Output: set()
```

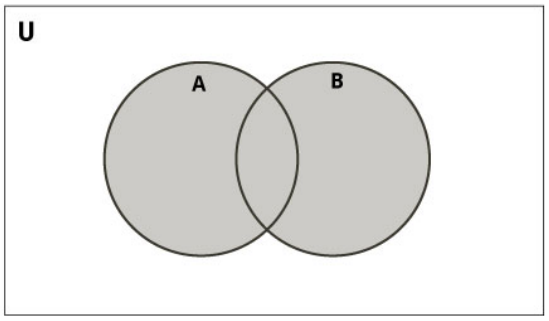
Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

Set Union

Union of A and B is a set of all elements from both sets.



Union is performed using | **operator**. Same can be accomplished using the **union() method**.

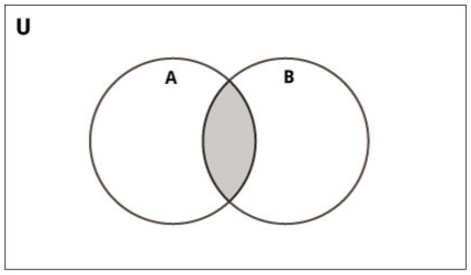
```
# union of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
# use | operator
     # Output: {1, 2, 3, 4, 5, 6, 7, 8}
     >>> print(A | B)     # Output: {1, 2, 3, 4, 5, 6, 7, 8}
# Using union() method
     # use union function
     >>> A.union(B)     # Output: {1, 2, 3, 4, 5, 6, 7, 8}
     # use union function on B
     >>> X=B.union(A)
     >>> print(X)       # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection

Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using **& operator**. Same can be accomplished using the **intersection() method**. It returns the resulting set.



```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
>>> print(A & B)         # Output: {4, 5}
# use intersection method on A
>>> A.intersection(B)    # Output: {4, 5}
# use intersection function on B
>>> X= B.intersection(A)
>>> print(X)        # Output: {4, 5}
```

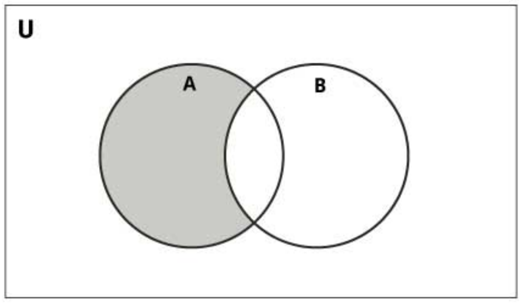
intersection_update() method does not return anything. The object set is modified and stores the result.

```
>>> A = {1, 2, 3, 4}
>>> B = {2, 3, 4, 5}
>>> result = A.intersection_update(B)
>>> print('result =', result) # Output: None
>>> print('A =', A)      # Output: {2, 3, 4}
>>> print('B =', B)      # Output: {2, 3, 4, 5}

>>> A = {1, 2, 3, 4}
>>> B = {2, 3, 4, 5, 6}
>>> C = {4, 5, 6, 9, 10}
>>> result = C.intersection_update(B, A)
>>> print('result =', result)  # Output: result = None
>>> print('C =', C)      # Output: C = {4}
>>> print('B =', B)      # Output: B = {2, 3, 4, 5, 6}
>>> print('A =', A)      # Output: A = {1, 2, 3, 4}
```

Set Difference

Difference of the set B from set A (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.
Difference is performed using **- operator.** Same can be accomplished using the **difference()** method.



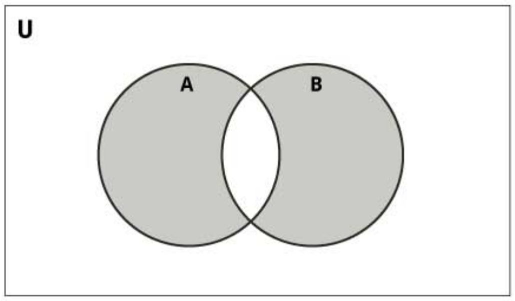
```
# Difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
>>> print(A - B)          # Output: {1, 2, 3}
# use difference() function on A
>>> A.difference(B)       # Output: {1, 2, 3}
# use - operator on B
>>> B - A                 # Output: {8, 6, 7}
# use difference function on B. It return a set
>>> C=B.difference(A)
>>> print(C)        # Output: {8, 6, 7}
# use difference_update() on A. A will store result
>>> A.difference_update(B)   # A=A-B; It return None
# Output: {1, 2, 3}
```

Set Symmetric Difference

Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).

Symmetric difference is performed using ^ **operator**. Same can be accomplished using the method **symmetric_difference()**.



```
# Symmetric difference of two sets
initialize A and B

>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}

    # use ^ operator
    >>> print(A ^ B)         # Output: {1, 2, 3, 6, 7, 8}
    # Using symmetric_difference() method
    # use symmetric_difference function on A
    >>> A.symmetric_difference(B)
    Output: {1, 2, 3, 6, 7, 8}
    # use symmetric_difference function on B
    >>> C=B.symmetric_difference(A)
    >>> print(C)      # Output: {1, 2, 3, 6, 7, 8}

    >>> A = {'a', 'c', 'd'}
    >>> B = {'c', 'd', 'e' }
    >>> result = A.symmetric_difference_update(B)
    >>> print('A =', A)        # Output: A = {'a', 'e'}
    >>> print('B =', B)      # Output: B = {'d', 'c', 'e'}
    >>> print('result =', result)      # Output: result = None
```

Methods of set object

isdisjoint()

The isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False.

Two sets are said to be disjoint sets if they have no common elements. For example:

```
A = {1, 5, 9, 0}
B = {2, 4, -5}
```

Here, sets A and B are disjoint sets.

```
>>> A = {1, 2, 3, 4}
>>> B = {5, 6, 7}
>>> C = {4, 5, 6}
>>> print('Are A and B disjoint?', A.isdisjoint(B))
>>> print('Are A and C disjoint?', A.isdisjoint(C))
```

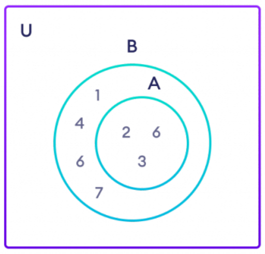
Output:

```
Are A and B disjoint? True
Are A and C disjoint? False
```

issubset()

The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

Set A is said to be the subset of set B if all elements of A are in B. Here A is a subset of B.



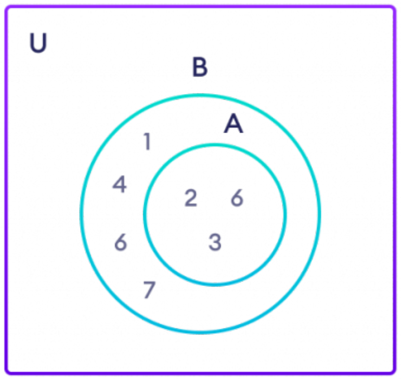
```
>>> A = {1, 2, 3}
>>> B = {1, 2, 3, 4, 5}
>>> C = {1, 2, 4, 5}
```

```
    >>> print(A.issubset(B))      #Output: True
    >>> print(B.issubset(A))      #Output: False
    >>> print(A.issubset(C))      #Output: False
    >>> print(C.issubset(B))      #Output: True
```

issuperset()

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

Set X is said to be the superset of set Y if all elements of Y are in X.



Here, set B is a superset of set A and A is a subset of set B.

```
    >>> A = {1, 2, 3, 4, 5}
    >>> B = {1, 2, 3}
    >>> C = {1, 2, 3}
    >>> print(A.issuperset(B))    # OUtput: True
    >>> print(B.issuperset(A))    # OUtput: False
    >>> print(C.issuperset(B))    # OUtput: True
```

Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

```
    # initialize my_set
    my_set = set("apple")
    # check if 'a' is present
    >>> print('a' in my_set)      # Output: True
    # check if 'p' is present
    >>> print('p' not in my_set)  # Output: False
    >>> 8 in A                    # Output: False
    >>> 8 not in A                # Output: True
```

Iterating Through a Set

We can iterate through each item in a set using a for loop.

```
    >>> for letter in set("apple"):
    ...    print(letter)
    ...
    Output: a
       P
       e
       l
```

Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with sets to perform different tasks.

all()

The all() function returns **True when all elements in the given iterable are true**. If not, it returns False. all() method returns:

- ❖ True - If all elements in an iterable are true
- ❖ False - If any element in an iterable is false
- ❖ True - For empty iterable it returns true

```
# all values true
>>> l = [1, 3, 4, 5]
print(all(l))       # Output: True
# all values false
>>> l = [0, False]
>>> print(all(l))        # Output: False
# one false value
>>> l = [1, 3, 4, 0]
>>> print(all(l))        # Output: False
# one true value
>>> l = [0, False, 5]
>>> print(all(l))        # Output: False
# empty iterable
>>> l = []
>>> print(all(l))        # Output: True
```

any()

The any() function returns **True if any element of an iterable is True**. If not, any() returns False. The any() function returns a boolean value:

- True - if at least one element of an iterable is true
- False - if all elements are false or if an iterable is empty
- False - For empty iterable it returns False

```
# True since 1,3 and 4 (at least one) is true
>>> l = [1, 3, 4, 0]
>>> print(any(l))        # Output: True
# False since both are False
>>> l = [0, False]
>>> print(any(l))        # Output: False
# True since 5 is true
>>> l = [0, False, 5]
>>> print(any(l))        # Output: True
# False since iterable is empty
>>> l = []
>>> print(any(l))        # Output: False
```

enumerate()

The enumerate() method adds counter to an iterable and returns it (the enumerate object). enumerate() method takes two parameters:

- iterable - a sequence, an iterator, or objects that supports iteration
- start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

enumerate() method adds counter to an iterable and returns it. The returned object is an enumerated object.

You can convert enumerate objects to list and tuple using list() and tuple() method respectively.

```
>>> grocery = ['bread', 'milk', 'butter']
>>> enumerateGrocery = enumerate(grocery)
>>> print(type(enumerateGrocery))  # Output:  <class
'enumerate'>
# converting to list
>>> print(list(enumerateGrocery))
# Output: [(0, 'bread'), (1, 'milk'), (2, 'butter')]
# changing the default counter
>>> enumerateGrocery = enumerate(grocery, 10)
>>> print(list(enumerateGrocery))
# Output: [(10, 'bread'), (11, 'milk'), (12, 'butter')]
-------------------
grocery = ['bread', 'milk', 'butter']
>>> for item in enumerate(grocery):
  ...   print(item)
Output: (0, 'bread')
     (1, 'milk')
     (2, 'butter')

>>> for count, item in enumerate(grocery):
  ...   print(count, item)
Output: 0 bread
        1 milk
        2 butter

# changing default start value
>>> for count, item in enumerate(grocery, 100):
   ...   print(count, item)
Output :  100 bread
          101 milk
          102 butter
```

sorted()

The sorted() function returns a sorted list from the items in an iterable. The sorted() function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list.

The syntax of the sorted() function is:

```
      sorted(iterable, key=None, reverse=False)
```

```
     # vowels list
     >>> py_list = ['e', 'a', 'u', 'o', 'i']
     >>> print(sorted(py_list))
     # Output: ['a', 'e', 'i', 'o', 'u']
     # string
     >>> py_string = 'Python'
     >>> print(sorted(py_string))
     # Output: ['P', 'h', 'n', 'o', 't', 'y']
     # vowels tuple
     >>> py_tuple = ('e', 'a', 'u', 'o', 'i')
     >>> print(sorted(py_tuple))
     # Output: ['a', 'e', 'i', 'o', 'u']
-----------------------
     >>> py_set = {'e', 'a', 'u', 'o', 'i'}
     >>> print(sorted(py_set, reverse=True))
     # Output: ['u', 'o', 'i', 'e', 'a']
     # dictionary
     >>> py_dict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5}
     >>> print(sorted(py_dict, reverse=True))
     Output: ['u', 'o', 'i', 'e', 'a']
     # frozen set
     >>> frozen_set = frozenset(('e', 'a', 'u', 'o', 'i'))
     >>> print(sorted(frozen_set, reverse=True))
     # Output: ['u', 'o', 'i', 'e', 'a']
```

```
     >>> student_tuples = [('john', 'A', 15),('jane', 'B',
     12),('dave', 'B', 10),]
     >>> sorted(student_tuples, key=lambda student: student[2])
     Output: [('dave','B',10), ('jane','B',12), ('john','A',15)]
```

```
>>> sorted(student_tuples, key=lambda student: student[2],
reverse=True)
Output: [('john', 'A', 15), ('jane', 'B', 12), ('dave',
'B', 10)]
```

sum()

The sum() function adds the items of an iterable and returns the sum.

The syntax of the sum() function is:

```
sum(iterable, start)
```

The sum() function adds start and items of the given iterable from left to right.

- iterable - iterable (list, tuple, dict, etc). The items of the iterable should be numbers.
- start (optional) - this value is added to the sum of items of the iterable. The default value of start is 0 (if omitted). sum() returns the sum of start and items of the given iterable.

```
>>> A={1,4,6,7,2,6,3}
>>> A          # Output: {1, 2, 3, 4, 6, 7}
>>> sum(A)     # Output: 23
>>> sum(A,3)   # Output: 26
>>> sum(A,1)   # Output: 24
```

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])
    >>> A.isdisjoint(B)
    False
    >>> A.difference(B)
    frozenset({1, 2})
    >>> A | B
    frozenset({1, 2, 3, 4, 5, 6})
    >>> A.add(3)
Output: AttributeError: 'frozenset' object has no attribute 'add'
```