Python Bootcamp & Masterclass

sets

gknxt

A set is a mutable object of unordered collection of distinct hashable objects.

- A set is an unordered collection of unique and hashable objects. (So, lists. sets and dictionaries cannot be members of a set. A tuple can be a member if and only if all its elements are immutable). Though all its elements need to be hashable, the set itself is not hashable.

- A non-empty set can be created by placing either a comma-separated list of elements or an iterator within a pair of **curly braces** (`{ }`) A set can also be created using the set constructor (`set()`)

```python
s1 = {'Hi', 3, 1, 3, 2}                                 # a set is enclosed in a pair of curly braces
print("object s1:", s1, "is of type:", type(s1))
```

```
object s1: {1, 2, 3, 'Hi'} is of type: <class 'set'>
```

```python
s2 = set ()                                             # empty set
print("object s2:", s2, "is of type:", type(s2))
```

```
object s2: set() is of type: <class 'set'>
```

```python
d3 = {}                                                 # not a set
print("object d3:", d3, "is of type:", type(d3))
```

```
object d3: {} is of type: <class 'dict'>
```

```python
s3 = {5}                                                # singleton set
print("object s3:", s3, "is of type:", type(s3), "and of length:", len(s3))
```

```
object s3: {5} is of type: <class 'set'> and of length: 1
```

gknxt

set elements are unique

The elements of a set must be unique, so if a set is created with duplicate elements, all the duplicates will be deleted. Any attempt to add a duplicate element will not be honored after a set was created.

```
email_list = ['hr@gknxt.com', 'ai@gknxt.com', 'hr@gknxt.com', 'gk@gknxt.com']    # 'hr@gknxt.com' is a duplicate
email_set = set(email_list)        # converts list into a set - removes duplicates, if present
email_set
```

```
{'ai@gknxt.com', 'gk@gknxt.com', 'hr@gknxt.com'}
```

```
email_list = ['hr@gknxt.com', 'ai@gknxt.com', 'hr@gknxt.com', 'gk@gknxt.com']
email_set = set(email_list)
print("Set before adding duplicate element", email_set)
email_set.add('gk@gknxt.com')      # duplicate element - will not be added
print("Set after adding duplicate element ", email_set)
email_set.add('py@gknxt.com')      # new element - will be added
print("Set after adding new element ", email_set)
```

```
Set before adding duplicate element {'hr@gknxt.com', 'ai@gknxt.com', 'gk@gknxt.com'}
Set after adding duplicate element  {'hr@gknxt.com', 'ai@gknxt.com', 'gk@gknxt.com'}
Set after adding new element  {'hr@gknxt.com', 'ai@gknxt.com', 'gk@gknxt.com', 'py@gknxt.com'}
```

modifying an element

- Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or any other sequence-dependent behavior.

- Since the elements of a set are unordered, modifying an element is not possible. However, the element that need to be modified can be deleted and the modified version of that element can be added.

deleting an element

There are three ways to delete an element from a set:

1. `remove(x)` removes element `x` from the set. Raises `KeyError` if `x` is not contained in the set.

2. `discard(x)` removes element `x` from the set if it is present. It does nothing if `x` is not contained in the set. Returns `None`

3. `pop()` removes and returns a random element from the set. If the set is empty, it'll raise a `KeyError`.

```
s1 = {'NY', 'AZ', 'CA'}
s1.remove('AZ')
s1
```

```
{'CA', 'NY'}
```

```
s2 = {'NY', 'AZ', 'CA'}
#s2.remove('TX')           # KeyError as 'TX' is not present in the set
s2
```

```
{'AZ', 'CA', 'NY'}
```

```
s2 = {'NY', 'AZ', 'CA'}
s2.discard('AZ')
s2.discard('TX')          # No error though 'TX' is not present in the set
s2
```

```
{'CA', 'NY'}
```

```
s3 = {'NY', 'AZ', 'CA'}
s3.pop()
s3.pop()
s3.pop()
#s3.pop()                 # KeyError as pop from an empty set is not possible
s3
```

```
'NY'

'CA'

'AZ'

set()
```

gknxt

deleting all elements

`clear()` removes all elements from the set. Another way to remove all the elements of a set is to make it equal to an empty set.

`del` keyword deletes the entire set, so the set will no longer be accessible.

```
a = {0, 1, 2, 3}
a.clear()
a
```

```
set()
```

```
b = {0, 1, 2, 3}
b = set()
b
```

```
set()
```

```
c = set()
c.clear()
c
```

```
set()
```

```
d = {0, 1, 2, 3}
del d
```

string to a set

A string is a sequence type, whose elements are simply its individual characters. Since the set() constructor takes a sequence and converts its elements to its set items, passing a string to the set() constructor creates a set of the string's individual characters after removing any duplicates.

```
set('Hello!')
```

```
{'!', 'H', 'e', 'l', 'o'}
```

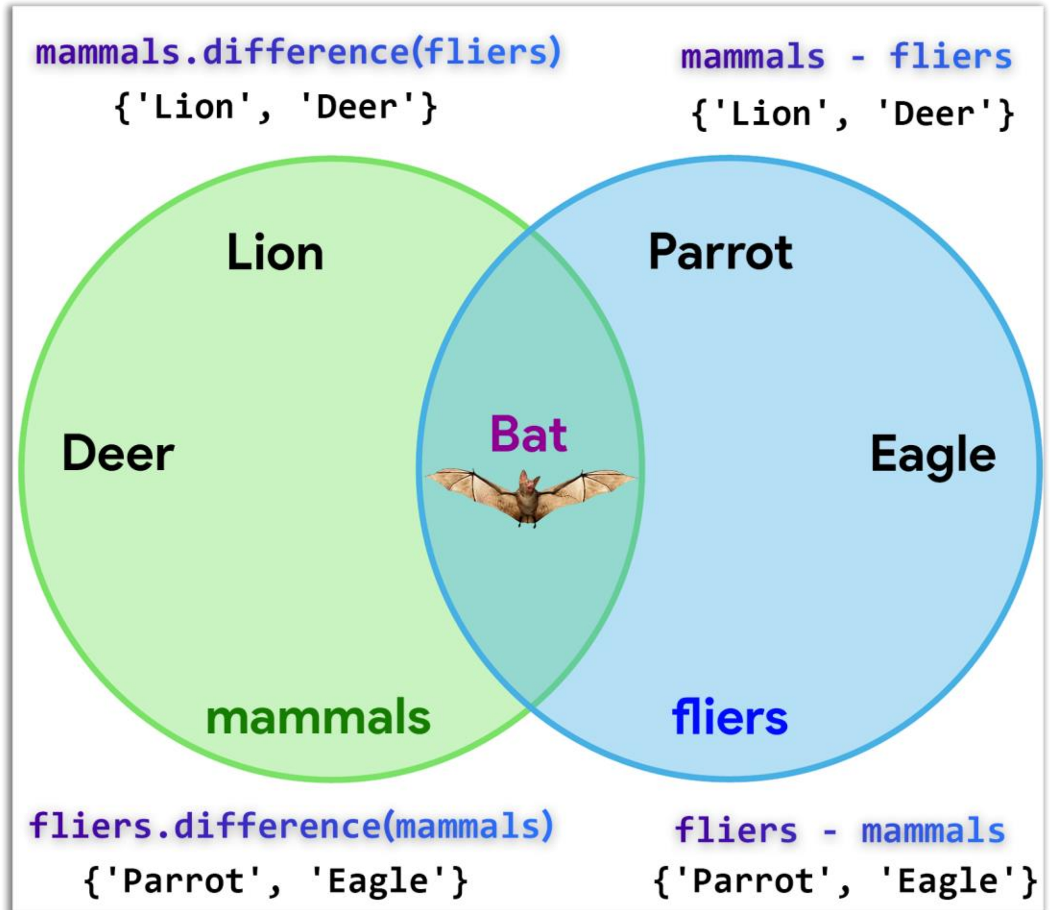
```
len(list(set('A thing of beauty is a joy forever')))  # No. of unique chars in the quote
```

```
18
```

difference

- `difference(*others)` returns a new set with elements in the set that are not in the `others`

- The overloaded minus operator (`-`) can be used in place of `difference(*others)`



```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
print("mammals.difference(fliers) =", mammals.difference(fliers))
print("mammals - fliers =", mammals - fliers)
print("fliers.difference(mammals) =", fliers.difference(mammals))
print("fliers - mammals =", fliers - mammals)
```

```
mammals.difference(fliers) = {'Lion', 'Deer'}
mammals - fliers = {'Lion', 'Deer'}
fliers.difference(mammals) = {'Parrot', 'Eagle'}
fliers - mammals = {'Parrot', 'Eagle'}
```

```
s = {1, 2, 3}
t = {1, 2}
s.difference(t)
s - t
t.difference(s)
t - s
```

```
{3}

{3}

set()

set()
```

gknxt

difference update

`difference(*others)` returns a new set with elements in the set that are not in the `others`, without modifying the existing set.

`difference_update(*others)` returns `None` and updates the set in place, removing elements found in `others`.

```
s = {1, 2, 3}
t = {1, 2}
u = s.difference_update(t)
s
t
type(u)
```

```
{3}

{1, 2}

NoneType
```

```
c = {1, 2, 3, 4, 5, 6}
d = {1, 2}
e = {3, 4}
f = {1, 3, 5}
c.difference_update(d, e, f)
c
```

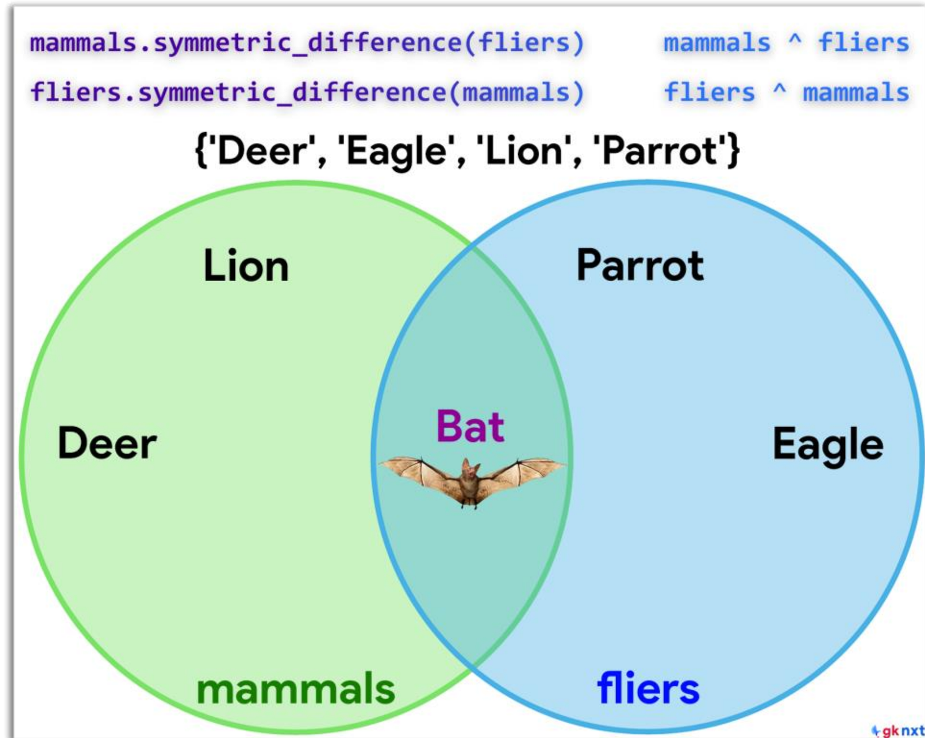
```
{6}
```

gknxt

symmetric difference

- `symmetric_difference(other)` returns a new set with elements in either the set or `other` but not both. (`other` can only be a set, not multiple sets)

- Unlike `symmetric_difference`, overloaded **bitwise XOR operator** (^) can be used on multiple sets



gknxt

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
mammals.symmetric_difference(fliers)
fliers.symmetric_difference(mammals)
mammals ^ fliers
fliers ^ mammals
```

```
{'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Deer', 'Eagle', 'Lion', 'Parrot'}
```

```
s = {1, 2, 3}
t = {2, 3, 4}
s.symmetric_difference(t)
t.symmetric_difference(s)
s ^ t
t ^ s
```

```
{1, 4}

{1, 4}

{1, 4}

{1, 4}
```

gk nxt

symmetric difference update

`symmetric_difference(other)` returns a new set with elements in either the set or `other` but not both, without modifying the existing set.

`symmetric_difference_update(other)` returns `None` and updates the set, keeping only elements found in either the set or `other` , but not in both.

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
s = mammals.symmetric_difference_update(fliers)
mammals
type(s)
```

```
{'Deer', 'Eagle', 'Lion', 'Parrot'}

NoneType
```

```
s = {1, 2, 3}
t = {1, 2}
s.symmetric_difference_update(t)
s
```

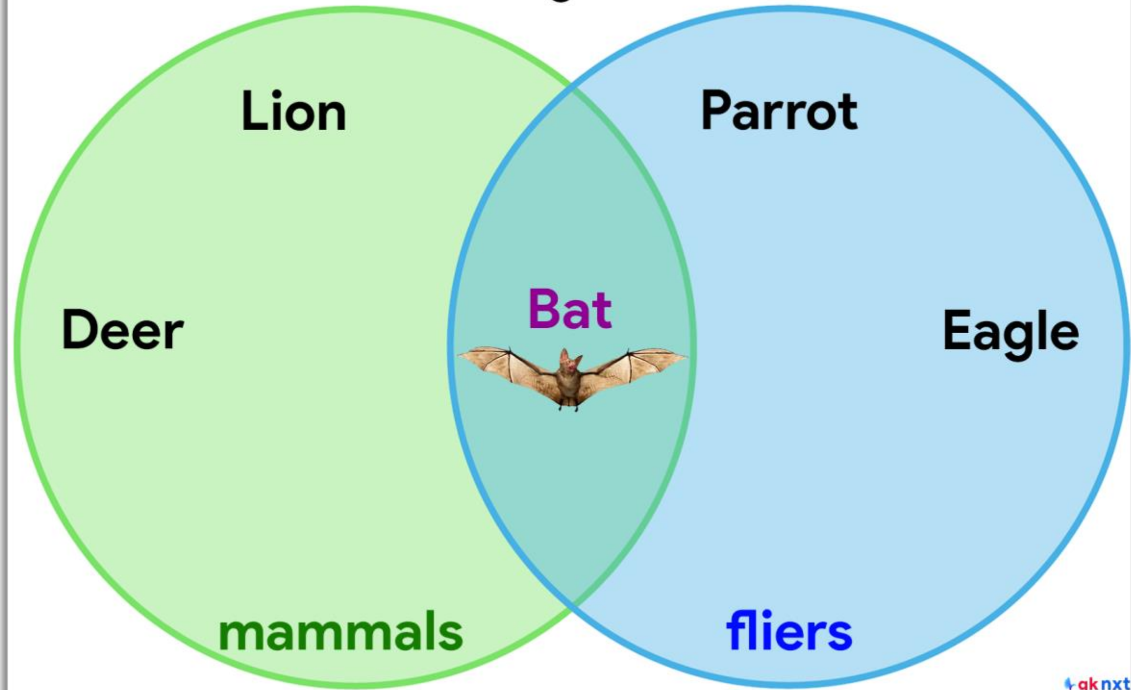
```
{3}
```

gknxt

union

- `union(*others)` returns a new set with elements from the set and all `others`

- The overloaded **bitwise OR operator** (|) can be used in place of `union(*others)`



gknxt

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
mammals.union(fliers)
fliers.union(mammals)
mammals | fliers
fliers | mammals
```

```
{'Bat', 'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Bat', 'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Bat', 'Deer', 'Eagle', 'Lion', 'Parrot'}

{'Bat', 'Deer', 'Eagle', 'Lion', 'Parrot'}
```

```
s = {1, 2, 3, 4}
t = {3, 4, 5}
s.union(t)
t.union(s)
s | t
t | s
```

```
{1, 2, 3, 4, 5}

{1, 2, 3, 4, 5}

{1, 2, 3, 4, 5}

{1, 2, 3, 4, 5}
```

gknxt

update

`union(*others)` returns a new set with elements from the set and all `others` without modifying the existing set.

`update(*others)` returns `None` and updates the set in place adding elements from all `others`.

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
mammals.update(fliers)
mammals
```

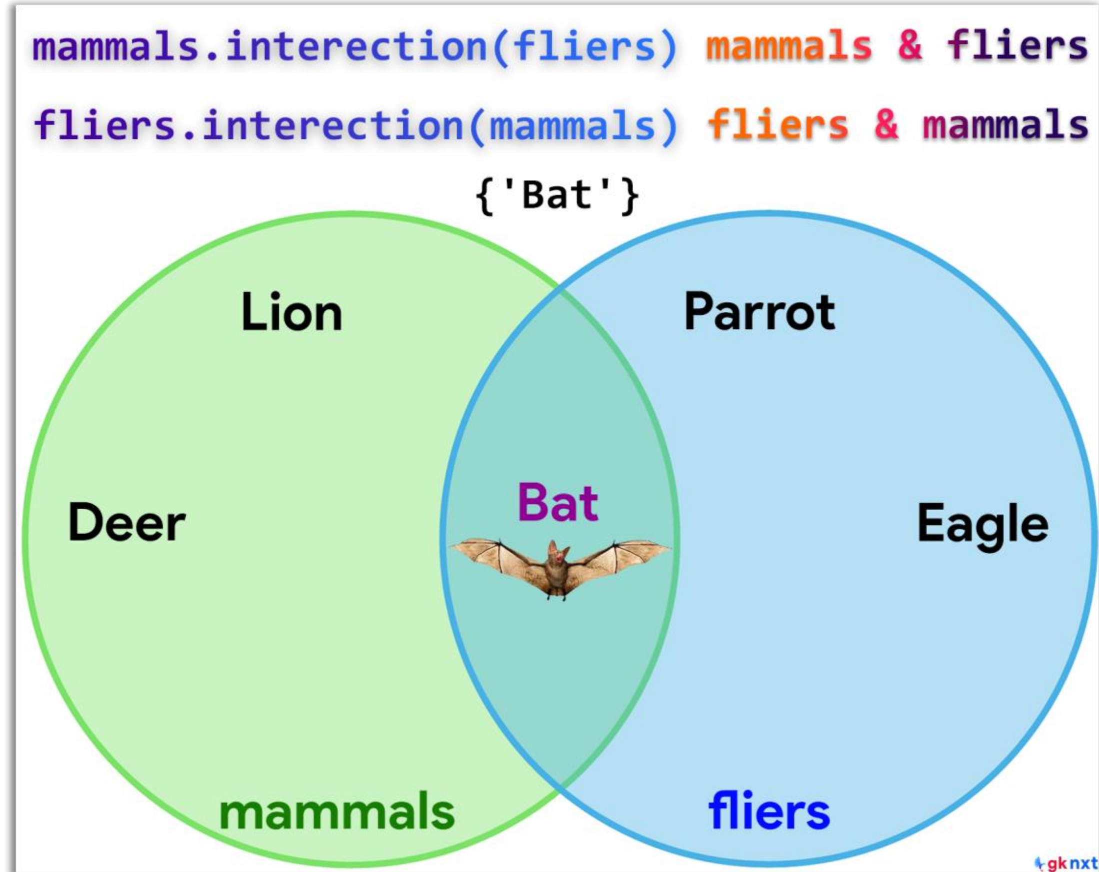
```
{'Bat', 'Deer', 'Eagle', 'Lion', 'Parrot'}
```

```
s = {1, 2, 3, 4}
t = {3, 4, 5}
s.update(t)
s
```

```
{1, 2, 3, 4, 5}
```

intersection

- `intersection(*others)` returns a new set with elements common to the set and all `others`
- The overloaded **bitwise AND operator** (`&`) can be used in place of the `intersection(*others)`



gknxt

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
mammals.intersection(fliers)
fliers.intersection(mammals)
mammals & fliers
fliers & mammals
```

```
{'Bat'}

{'Bat'}

{'Bat'}

{'Bat'}
```

```
s = {1, 2, 3, 4}
t = {3, 4, 5}
s.intersection(t)
t.intersection(s)
s & t
t & s
```

```
{3, 4}

{3, 4}

{3, 4}

{3, 4}
```

gknxt

intersection update

`intersection(*others)` returns a new set with elements common to the set and all `others` without modifying the existing set.

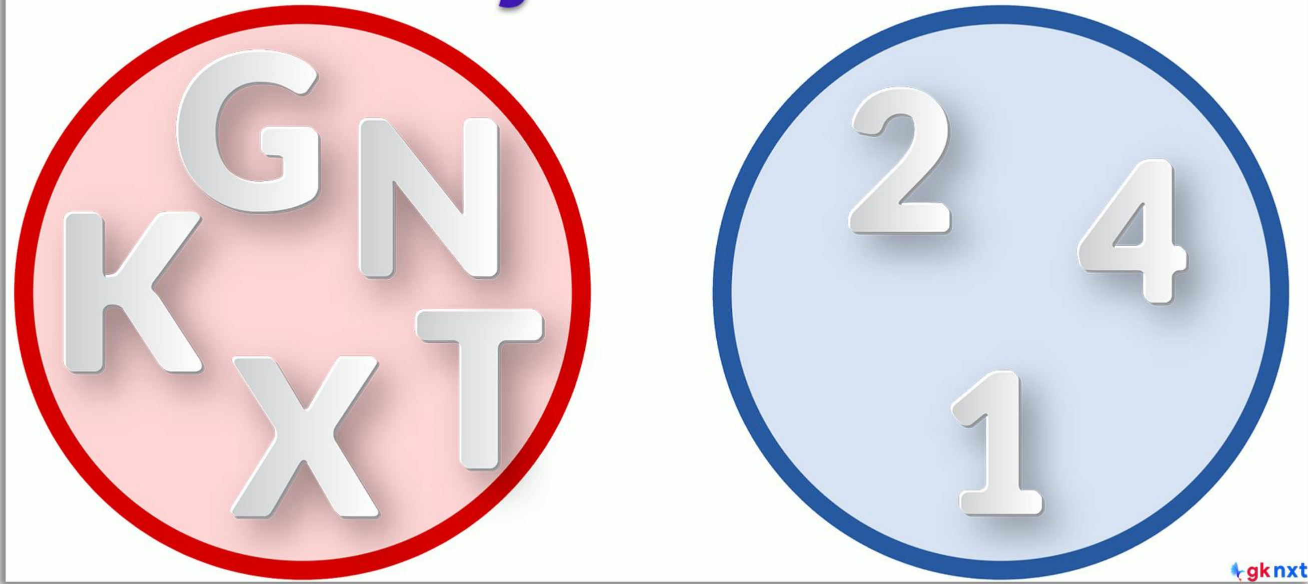
`intersection_update(*others)` returns `None` and updates the set in place with elements common to the set and all `others`.

```
mammals = set(["Lion", "Deer", "Bat"])
fliers = set (["Parrot", "Eagle", "Bat"])
mammals.intersection_update(fliers)
mammals
```

```
{'Bat'}
```

```
s = {1, 2, 3, 4}
t = {3, 4, 5}
s.intersection_update(t)
s
```

```
{3, 4}
```



`isdisjoint(other)` returns `True` if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set. There is no operator that corresponds to the .isdisjoint() method.

```
s = {4, 2, 1}
t = {'G', 'K', 'N', 'X', 'T'}
s.isdisjoint(t)
s.isdisjoint(t)
```

```
True

True
```

```
a = {9, 19, 29, 39, 49, 59}
b = {2, 5, 7, 11, 13, 17, 19}
a.isdisjoint(b)
b.isdisjoint(a)
```

```
False

False
```

```
c = set()
d = set()
c.isdisjoint(d)   # intersection between two empty sets is empty
d.isdisjoint(c)   # two empty sets are disjoint to each other
```

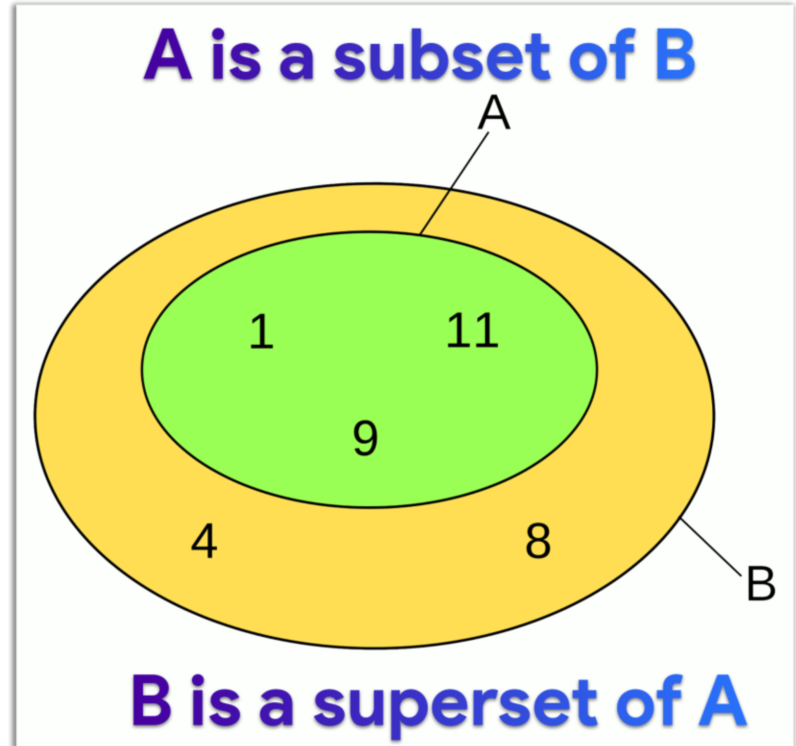
```
True

True
```

gk nxt

subset & superset

- `issubset(other)` returns `True` if every element in the set is in `other`. The symbol <= corresponds to subset and < corresponds to proper subset.

- `issuperset(other)` returns `True` if every element in the `other` is in set. The symbol >= corresponds to superset and > corresponds to proper superset.



operator Vs non-operator

- The non-operator versions of `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, and `issuperset()` methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets.

- The > operator is the only way to test whether a set is a proper superset or not. There is no corresponding method.

- The < operator is the only way to test whether a set is a proper subset or not. There is no corresponding method.

```
s = {'Bush', 'Biden'}
t = {'Reagan', 'Bush', 'Obama', 'Trump', 'Biden'}
s.issubset(t)
t.issuperset(s)
```

```
True

True
```

```
s = ['Bush', 'Biden']    # s is a list
t = {'Reagan', 'Bush', 'Obama', 'Trump', 'Biden'}
t.issubset(s)             # issubset method can take any itrable as argument
t.issuperset(s)           # issuperset method can take any itrable as argument
```

```
False

True
```

```
s = {'Bush', 'Biden'}
t = {'Reagan', 'Bush', 'Obama', 'Trump', 'Biden'}
s.issubset(s)             # any set is a subset of itself
t.issuperset(t)           # any set is a superset of itself
```

```
True

True
```

gknxt

copy()

The copy() method returns a shallow copy of the set. Deep copy doesn’t make any sense for sets because sets are only allowed to contain immutable objects.

```
s = {11, 2, 5, 4, 5, 8, (3, 7), 5, 2}
t = s.copy()  # deep copy doesn’t make sense for sets because they hold immutable objects only
print("set s:", s, "has id:", hex(id(s)))
print("set t:", t, "has id:", hex(id(t)))
```

```
set s: {2, 4, 5, 8, (3, 7), 11} has id: 0x19d3c0ba900
set t: {2, 4, 5, 8, (3, 7), 11} has id: 0x19d3c0ba200
```

```
s1 = {(1, 2, 3)}
s2 = s1.copy()
hex(id(s1.pop()))   # the tuple points to the same memory location in both sets implying shallow copy
hex(id(s2.pop()))
```

```
'0x19d391c5e80'

'0x19d391c5e80'
```

in & not in

The `in` and `not in` operators can be used to test whether a value is in a set or not.

```
s = {'Reagan', 'Bush', 'Obama', 'Trump', 'Biden'}
'Obama' in s
'Hillary' not in s
```

```
True

True
```

```
s = {'Reagan', 'Bush', ('Obama', 'Trump'), 'Biden'}
t = 'Obama'
u = ('Obama', 'Trump')
t in s
u in s
```

```
False

True
```

Online Resources

For best python resources, please visit:

gknxt.com/python/

gknxt

Python Bootcamp & Masterclass

Thank You

for your Rating & Review