4. Sets, frozen sets

Unordered collection of distinct objects, called **elements** or **members**.
Sets are distinguished from other object types by the unique operations that can be performed on them.
It is Python's built-in set type and has the following characteristics:

- Sets are **unordered**;
- Set **elements are unique. Duplicates are not allowed**;
- A set itself may be modified, but the _ (e.g. `lists` or `dicts` cannot be set members);
- The elements in a set can be **objects of different types**.

1 Defining a Set

A set can be created in **two ways**.

(1) Using function `set(<iter>)`, where `<iter>` is e.g. **list**, **tuple** or **string**:

```
set(['foo', 'bar', 'baz', 'qux', 'foo', 1])  #<iter> is list; no duplications
>> {1, 'bar', 'baz', 'foo', 'qux'}
```

```
set(('foo', 'bar', 'baz', 'qux', 'foo'))  #<iter> is tuple
>> {'bar', 'baz', 'foo', 'qux'}
```

```
set('quux')  #<iter> is string
>> {'q', 'u', 'x'}
```

(2) Using the curly braces, `x = {<obj>, <obj>, ..., <obj>}`:

```
x = {'q', 'u', 'u', 'x', 10}
>> {10, 'q', 'u', 'x'}
```

Observe the **difference with str**:

```
{'foo'}
>> {'foo'}
set('foo')
>> {'f', 'o'}
```

A set can be empty. However, **Python interprets empty curly braces as an empty dictionary**, so the only way to define an empty set is with the `set()`:

```
x = set()  # empty set
```

2 Set size and membership

`len()` function – the number of elements in the set:

```
x = {'q', 'u', 'u', 'x',10}
len(x)
>> 4
```

`in` and `not in` to test membership:

```
'x' in x
>> True
'x' not in x
>> False
```

3 Set operations

Sets **cannot be indexed or sliced**. However, Python provides operations on set objects that generally mimic the operations defined for mathematical sets.

Most, but not all set operations in Python can be performed in **2 ways: operator** or **method**. **Difference** between operators and methods: **for operator**, **both operands must be sets**, while `.method()` takes **any iterable** as an argument, convert it to a set, and then performs the operation.

(1) Union of two or more sets.

```
a | b
.union(a [,b ...])
```

(2) Intersection of two or more sets. Returns the elements common to all the sets.

```
a & b
.intersection(a [,b ...])
```

(3) Difference between two or more sets. **Not commutative operation**.

```
a - b
.difference(a [,b ...])
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}

x1.difference(x2)
>> {'bar', 'foo'}

x1 - x2
>> {'bar', 'foo'}

x2 - x1
>> {'quux', 'qux'}
```

In case of multiple sets, the operation is performed **from left to right**.

(4) Symmetric difference between **two** sets. Return the set of all elements in either **a** or **b**, but not both.

```
a ^ b
a.symmetric_difference(b)
```

```
a = {'foo', 'bar', 'baz'}
b = {'baz', 'qux', 'quux'}

a.symmetric_difference(b)
>> {'bar', 'foo', 'quux', 'qux'}
```

(5) isDisjoint – `True` if two sets `x1` and `x2` have **no elements in common**.

```
a = {'foo', 'bar', 'baz'}
b = {'baz', 'was', 'quux'}

a.isdisjoint(b)
>> False
```

(6) isSubset – `True` if `x1` is a subset of `x2`. In set theory, a set `x1` is considered a subset of another set `x2` if every element of `x1` is in `x2`. Also `True` if two sets are identical.

```
x1 = {'foo', 'bar', 'baz'}

x1.issubset({'foo', 'bar', 'baz', 'qux', 'quux'})
>> True
```

(7) isProper subset – `True` if `x1` is a proper subset of `x2`.
A proper subset is the **same as a subset, except that the sets cannot be identical**.
A set `x1` is considered a proper subset of another set `x2` if every element of `x1` is in `x2`, and `x1` and `x2` are not equal.

```
a < b
```

```
x1 = {'foo', 'bar'}
x2 = {'foo', 'bar', 'baz'}

x1 < x2
>> True
```

(8) isSuperset is the **reverse of a subset**. A set `x1` is considered a superset of another set `x2` if `x1` contains every element of `x2`.

```
a >= b
a.issuperset(b)
```

```
x1 = {'foo', 'bar', 'baz'}
x1.issuperset({'foo', 'bar'})
>> True
```

```
x2 = {'baz', 'qux', 'quux'}
x1 >= x2
>> False
```

(9) isProper superset – the **same as a superset, except that the sets cannot be identical**. A set `x1` is considered a proper superset of another set `x2` if `x1` contains every element of `x2`, and `x1` and `x2` are not equal.

```
a > b
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'bar'}

x1 > x2
>> True
```

4 Modify set

4.1 Augmented assignment update operators and methods

Each of the **union**, **intersection**, **difference**, and **symmetric difference** operators above has an **augmented assignment (AA)** form that can be used to modify a set.

For each, there is a corresponding method as well.

(1) AA: **Union** – modify a set by union.

```
a |= b
a.update(b[, ...])
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'baz', 'qux'}

x1 |= x2
>> {'bar', 'baz', 'foo', 'qux'}
```

```
x1.update(['corge', 'garply'])
>> {'bar', 'baz', 'corge', 'foo', 'garply', 'qux'}
```

(2) AA: **Intersection** – Modify a set by intersection. Retains only elements found in both `x1` and `x2`.

```
a &= b
a.intersection_update(b[, ...])
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'baz', 'qux'}

x1 &= x2
>> {'baz', 'foo'}
```

```
x1.intersection_update(x2)
>> {'baz', 'foo'}
```

(3) AA: **Difference** – Modify a set by difference. Update `x1`, removing elements found in `x2`.

```
a -= b
a.difference_update(b[, ...])
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'baz', 'qux'}

x1 -= x2
>> {'bar'}
```

```
x1.difference_update(x2)
>> set()
```

(4) AA: **Symmetric difference** – Modify a set by symmetric difference. Update `x1`, removing elements found in `x2`. Retains elements found in either `x1` or `x2`, but not both:

```
a ^= b
a.symmetric_difference_update(b)
```

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'baz', 'qux'}

x1 ^= x2
>> {'bar', 'qux'}
```

4.2 Other methods to modify sets

(1) **add** – add an element **(a sigle immutable object)** to a set:

```
a.add(b)
```

```
x = {'foo', 'bar', 'baz'}

x.add('qux')
>> {'bar', 'baz', 'foo', 'qux'}
```

(2) **remove** – remove an element from a set.
– <`elem`> must be **a single immutable object**.
– if <`elem`> is not in a set, **exception will be raised**.

```
a.remove(b)
```

```
x = {'foo', 'bar', 'baz'}

x.remove('baz')
>> {'bar', 'foo'}
```

(3) **discard** – remove an element from a set.
– <`elem`> must be **a single immutable object**.
– **but**, if <`elem`> is not in a set, **method quietly does nothing instead of raising an exception.**

```
a.discard(b)
```

```
x = {'foo', 'bar', 'baz'}

x.discard('baz')
>> {'bar', 'foo'}
```

(4) **pop** – **remove and return an arbitrary element** from a set.
– if x is empty, `x.pop()` raises an exception.

```
a.pop()
```

```
x = {'foo', 'bar', 'baz'}

x.pop()
>> 'baz'
x
>> {'bar', 'foo'}
```

(5) **clear** – **remove all** elements from a set.

```
a.clear()
```

(6) **copy** – copy a set. **The only way how can you copy a mutable object**.

```
b = a.copy()
```

5 Frozen sets

Python provides another built-in type called a frozenset, which is in all respects exactly **like a set**, except that a **frozenset is immutable**. You **can perform non-modifying operations** on a frozenset.

5.1 Define a frozenset

```
x = frozenset(['foo', 'bar', 'baz'])
```

5.2 Motivation behind frozensets

Frozensets are useful in situations where you want to use a set, but you need an immutable object.

For example, **(1) you cannot define a set whose elements are also sets**, because set elements must be immutable:

```
x1 = set(['foo'])
x2 = set(['bar'])
x3 = set(['baz'])

x = {x1, x2, x3}
```

gives:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-85-d9872a3ec1df> in <module>
3 x3 = set(['baz'])
4
----> 5 x = {x1, x2, x3}

TypeError: unhashable type: 'set'
```

But set of frozen sets are OK:

```
x1 = frozenset(['foo'])
x2 = frozenset(['bar'])
x3 = frozenset(['baz'])

x = set([x1, x2, x3])
>> {frozenset({'baz'}), frozenset({'bar'}), frozenset({'foo'})}
```

(2) you cannot use sets as keys in dictionaries but frozensent are also OK:

```
x = frozenset({1, 2, 3})
y = frozenset({'a', 'b', 'c'})

d = {x: 'foo', y: 'bar'}
>> {frozenset({1, 2, 3}): 'foo', frozenset({'a', 'b', 'c'}): 'bar'}
```

5.3 Frozensets and augmented assignment updates

Since a frozenset is immutable, you might think it can't be the target of an augmented assignment operator. However:

```
f = frozenset(['foo', 'bar', 'baz'])
s = {'baz', 'qux', 'quux'}

f &= s
>> frozenset({'baz'})
```

Python does not perform augmented assignments on frozensets in place. The statement `x &= s` is effectively equivalent to `x = x & s`. It isn't modifying the original `x`. **It is reassigning `x` to a new object, and the object `x` originally referenced is gone.**

Some objects in Python are modified in place when they are the target of an augmented assignment operator. But frozensets aren't.