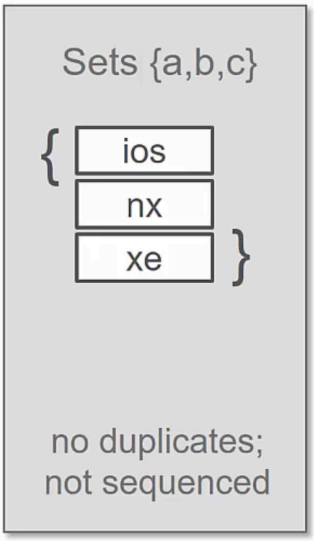
Overview

- **Unordered:** Unordered collection of items
- **Unique items:** No duplicate members allowed in set
- **Hashable:** Members must be hashable (like dictionary keys), meaning numbers, strings, tuples (no dictionaries or lists)
- **{ }:** Uses curly braces like dictionary but with only a single value for each member (no k:v)
- **Mutable:** Sets can be modified



Creating a Set

Create empty set:

```
dev_os_types = set()
```

Create populated set:

```
dev_os_types = {'ios','xr','nx'}
```

Create set from list:

```
devices_list = [dev30.type, dev31.type, dev32.type,
                dev33.type, dev34.type, dev35.type]
dev_os_types = set(devices_list) # create set of types
                                 # from list
```

Add, Remove, Update

Add an item to a set

```
dev_types = set()
dev_types.add('crs') # add device type 'crs' to set
```

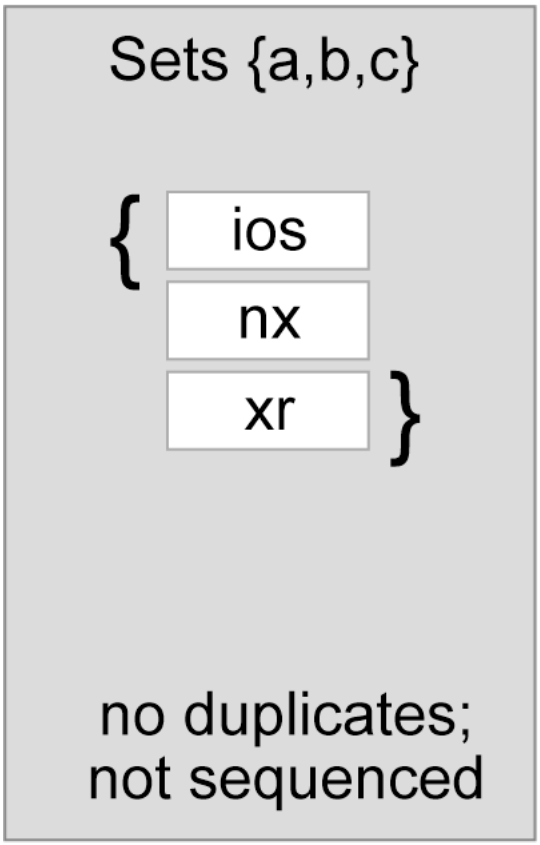
Update a list of items to a set

```
dev_types.update(['isr','asr']) # add more device types
```

Remove an item from a set by value

```
dev_types.remove('crs')  # remove device type 'crs'
```

- **Unordered:** Unordered collection of items
- **Unique Members:** No duplicate members allowed in a set
- **Hashable:** Members must be hashable—numbers, strings, or tuples (no dictionaries or lists)
- **{ }:** Uses curly braces like dictionary but with a single value for each member (no key)
- **Mutable:** Sets can be modified



Sets Overview

Sets are data structures that are best used in testing for the presence or absence of an item, and for performing operations such as unions and intersections. Some characteristics of sets:

- **Unordered:** Sets contain items in no particular order or sequence – the only consideration for a set is whether an item is 'in' the set or not.
- **Unique items**: There are no duplicate items in a set, every item is present only once. If you take a list with duplicate items, and coerce it into a set, all duplicates will be removed.
- **Hashable:** Members of a set must be hashable, meaning that they must be either simple, like numbers or strings, or tuples, which themselves are immutable and hence can be hashed. This requirement makes sense when you consider the operations that are performed on sets, such as testing for inclusion, are easily done with hashable items, but would be quite difficult and inefficient otherwise.
- **Curly braces {}:** You may be confused by the fact that sets are enclosed in curly braces, as with dictionaries. However, consider that dictionaries are populated with key-value pairs, while sets are populated with single items, which eliminates confusion and makes the two curly braces data types easy to distinguish.
- **Mutable**: You can add and remove items from a set.

Creating a Set

Unlike lists and dictionaries, there is only one way to create an empty set:

```
dev_os_types = set()
```

There is no 'empty braces' notation as is possible with lists and dictionaries. The reason is because the type of braces that are used – curly braces {} – are already used by dictionaries, and so using them to create an empty set would be ambiguous.

Creating a populated set is similar to creating populated lists, dictionaries, and tuples: include the items inside the braces. Notice that these items can be distinguished from items in a dictionary, since they are individual, not key-value pairs. Hence Python is able to know whether you were creating a set or a dictionary. The example below shows creating a populated set:

```
dev_os_types = {'ios','xr','nx'}
```

It is also possible to create a set from a list. The general format is to use the 'set' function, as in the following example. Consider a list that holds the device types of all devices that have been discovered in the network. This list will have many items, with many duplicates. Creating a set of device types from that list, is straightforward:

```
devices_list = ['xr','nx','nx','xe','xr','nx','xr','ios','nx','xe','xr','xr','nx'
dev__os_types = set(device_types_list)
if 'ios' in dev_os_types:
 # do something with ios devices
```

A more complicated example is shown in the figure. Imagine several device tuples, these being dev30, dev31, dev32, and so on, which are actually named tuples, allowing easy reference to the OS type of each device by using the '.type' suffix. In the example, the `devices_list` list is created from the 'type' field of these device tuple objects. Then this list is used to create the list of all 'dev_os_types' present in the network.

```
devices_list = [dev30.type, dev31.type, dev32.type,  dev33.type, dev34.type, dev35
dev_os_types = set(devices_list) # create set of types from list of all device's (
if 'ios' in dev_os_types:
# do something with ios devices
```

Sets can easily be created from data structures such as lists, allowing for easy testing for inclusion, and other set-type operations if so desired.

Modifying Sets

You can add items to a set using the 'add' function:

```
dev_types = set()
dev_types.add('crs') # add device type 'crs' to set
```

You can update items to a set as well - update is really just adding more than one item at a time.

```
dev_types.update(['isr','asr']) # add more device types
```

Since items are hashable, it is possible to remove items by specifying the value:

```
dev_types.remove('crs')  # remove device type 'crs'
```

Testing Sets

One of the main reasons for having sets is to test for inclusion, given some other value. For example, if there is a list of used IP addresses for the network, it may be necessary to test to see if a given IP address has already been used. The following example shows the testing of inclusion for this situation:

```
ip = '10.3.21.17'
used_ip_addresses = {'10.3.21.1','10.3.21.2','10.3.21.3', '10.2.21.4','10.3.21.5'}
if ip in used_ip_addresses # test if 'ip' is used yet
```

In addition to these types of tests, operations such as finding the intersection or union of two sets is also possible.