
Home Lesson-7.1

# Lesson-6.5

**Lesson-6.5**
- Sets
  - Introduction
  - Iterating through Sets
  - Growing Sets
  - Set Operations
  - Other Set Methods
  - Mutability

## Sets

### Introduction

A set is an unordered collection with no duplicate elements [refer]. Unlike lists and tuples, there is no notion of order in a set. This is why it is called an unordered collection as opposed to a sequence. A set can be defined as follows:

```
even_nums = {2, 4, 6, 8, 10}
print(type(even_nums))
print(isinstance(even_nums, set))
```

Notice the similarity in syntax between sets and dictionaries. Both are enclosed within curly braces. While a dictionary has key-value pairs in it, a set just has a collection of values. A set in Python is a remarkably accurate representation of a mathematical set. Therefore, most of the properties that you are used to seeing in mathematical sets nicely carry over to Python sets. This connection is so strong that you can often forget that you are dealing with Python sets.

```
nums_1 = {2, 4, 6, 8, 10}
nums_2 = {2, 2, 4, 4, 6, 6, 8, 8, 10, 10}
print(nums_1, nums_2)
print(nums_1 == nums_2)
print(nums_1 is not nums_2)
```

As stated before, sets do not support duplicate elements. We see that `nums_1` and `nums_2` are equal sets. However, they don't point to the same object. Sets support membership just like lists, tuples and dictionaries.

```
nums = {1, 2, 3, 4, 5}
print(1 in nums)
print(6 not in nums)
```

The number of elements in a set, which is the same as its cardinality, is given by the `len` function:

```
nums = {1, 2, 3, 4, 5}
print(f'Cardinality of nums is {len(nums)}')
```

Sets cannot be indexed. This is quite reasonable as they are not ordered collections. The following code will throw an error:

```
##### Alarm! Wrong code snippet! #####
some_set = {'this', 'is', 'a', 'set'}
print(some_set[0])
##### Alarm! Wrong code snippet! #####
```

Any hashable object can be added to sets. This means most of the immutable types such as `int`, `float`, `str` and `tuple` can be added to sets. A small caveat as far as tuples are concerned: a tuple of lists is unhashable and therefore cannot be added to sets.

```
a_set = {1.0, 'one', 1, True, (1, )}    # valid set
not_a_set = {([1, 2], [3, 4])}          # not a valid set
```

`not_a_set` returns a `TypeError` as expected.

## Iterating through Sets

Though a set is not a sequence, iterating through the elements of a set is supported.

```
nums = {1, 2, 3, 4, 5}
for num in nums:
    print(num)
```

## Growing Sets

How do we define an empty set?

```
##### Alarm! Be careful about the variable name! #####
empty_set = { }
print(isinstance(empty_set, set))
print(isinstance(empty_set, dict))
##### Alarm! Be careful about the variable name! #####
```

We see that `empty_set` is in fact an empty dictionary. Computers are precise machines, which makes them very faithful. Few lessons back we used `{ }` to initialize an empty dictionary. It hasn't changed. `{ }` is still an empty dictionary. So, how do we define an empty set then?

```
empty_set = set()
print(isinstance(empty_set, set))
```

Simple enough! With the empty set and set-iteration defined, we can now grow sets from scratch.

> Consider the first 100 powers of 7:
>
> $$7^1, 7^2, \cdots, 7^{100}$$
>
> Note down the last digit of each of these powers. How many of them are unique? What are these numbers?

This problem has a simple mathematical solution. But humor me and assume that you don't know how to solve this problem. Let us go for a computational solution.

```
num = 1
digits = set()
for i in range(100):
    num *= 7
    last = num % 10
    digits.add(last)
print(digits)
```

`add` is a method used to add elements to a set. The solution to this problem is a typical use case of sets. When you expect duplicate elements to come up often and if you are not concerned with duplicates, then sets are ideal objects for storage. The same problem can be solved using lists:

```
num = 1
digits = [ ]
for i in range(100):
    num *= 7
    last = num % 10
    if last not in digits:
        digits.append(last)
print(digits)
```

## Set Operations

Mathematical sets are friendly objects. They routinely interact with each other through one of the following operations:

- Subset
- Superset
- Union
- Intersection
- Difference

Python sets strive to be as friendly as their mathematical counterparts. We will see how each of these operations are represented:

- Subset: $A$ is a subset of $B$ if every element of $A$ is present in $B$. It is denoted by $A \subseteq B$. This is a binary relationship and its outcome can be determined in one of the two ways:

```
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(A.issubset(B))    # method-1
print(A <= B)           # method-2
```

Both lines return the value `True`. A set $A$ is a proper subset of $B$ if every element in $A$ is present in $B$ and $A \neq B$. It is denoted by $A \subset B$. That is, there is at least one element in $B$ which is not in $A$:

```
A = {1, 2, 3}
B = {1, 2, 3}
print(A <= B)   # method-1
print(A < B)    # method-2
```

The `A < B` operator checks if `A` is a proper subset of `B`. In this case `A` is not a proper subset of `B`, so the second print statement returns `False`.

- Superset: $A$ is a superset of $B$ if every element of $B$ is present in $A$. It is denoted by $A \supseteq B$:

```
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
B.issuperset(A)     # method-1
print(B >= A)       # method-2
```

- Union: The union of two sets $A$ and $B$ is the set of elements that are present in either $A$ or $B$ or both. It is denoted by $A \cup B$.

```
A = {1, 3, 5}
B = {2, 4, 6}
C1 = A.union(B)     # method-1
C2 = A | B          # method-2
print(C1, C2)
print(C1 == C2)
```

When there are multiple sets, we could do the following:

```
A1, A2, A3, A4 = {1}, {2, 3}, {4, 5, 6}, {7, 8, 9, 10}
B1 = A1.union(A2, A3, A4)   # method-1
B2 = A1 | A2 | A3 | A4      # method-2
print(B1, B2)
print(B1 == B2)
```

- Intersection: The intersection of two sets $A$ and $B$ is the set of elements common to both. It is denoted by $A \cap B$.

```
A = {2, 4, 6}
B = {2, 4}
C1 = A.intersection(B)  # method-1
C2 = A & B              # method-2
print(C1, C2)
print(C1 == C2)
```

What happens if there are no elements in common? We should get the empty set:

```
even, odd = {2, 4, 6}, {1, 3, 5}
common = even & odd
assert common == set()
```

We have used an assert statement just to introduce some variation. As it doesn't raise an `AssertionError`, we are right on target.

- Difference: The difference between two sets $A$ and $B$ is the set of elements present in one set but not in the other. It is denoted by $A - B$ or $B - A$, and the two are not the same!
  - $A - B$ is the set of elements in $A$ which are not in $B$.
  - $B - A$ is the set of elements in $B$ which are not in $A$.

```
A = {1, 2, 3, 4}
B = {2, 4, 5}
C1 = A.difference(B)    # method-1
C2 = A - B              # method-2
print(C1, C2)
print(C1 == C2)
D1 = B.difference(A)    # method-1
D2 = B - A              # method-2
print(D1, D2)
print(D1 == D2)
```

## Other Set Methods

The methods that we saw in the previous section had a mathematical flavor. Now, we shall look at those methods that have a computational flavor!

To remove an element from the set, we can use the `remove` method:

```
A = {'this', 'is', 'a', 'set'}
print('Before', A)
A.remove('this')
print('After', A)
```

If we try to remove an element that is not present in the set, the interpreter will throw a `KeyError`:

```
A = {'this', 'is', 'a', 'set'}
A.remove('cool')    # error!
```

Consider the following problem:

> Given a list `L`, extract all unique elements from it and store the result in another list, `L_uniq`. The order of elements does not matter.

Let us first look at a solution that doesn't use sets:

```
L = [1, 2, 3, 3, 4, 5, 6, 1, 2, 2]
L_uniq = [ ]
for elem in L:
    if elem not in L_uniq:
        L_uniq.append(elem)
print(L_uniq)
```

Now, for some set magic:

```
L = [1, 2, 3, 3, 4, 5, 6, 1, 2, 2]
S = set(L)
L_uniq = list(S)
print(L_uniq)
```

Passing a list to the `set` function removes all duplicates and returns the unique elements.

## Mutability

Sets are mutable entities.

```
A = {1, 2, 3}
B = A
B.add(4)
print(A, B)
print(A is B)
```

`A` and `B` are the same objects. As before, there are two ways to do a shallow copy:

```
A = {1, 2, 3}
B1 = A.copy()
B2 = set(A)
B1.add(4)
B2.add(0)
print(A, B1, B2)
print(A is not B1)
print(A is not B2)
```