Python → Collections → Set

Theory: Set

20 minutes 7 / 10 problems solved

Start practicing

4919 users solved this topic. Latest completion was about 8 hours ago.

When you need to get rid of duplicates in a sequence or intend to perform some mathematical operations, you may use a **set** object. A **set** is an **unordered** container of **hashable** objects. You will learn more about hashable objects later, for now, remember that only immutable data types can be elements of a set. Due to their form, sets **do NOT** record element position or order of insertion, so you cannot retrieve an element by its index.

§1. Creating sets

First things first, we create a set by listing its elements in curly braces. The only exception would be an **empty set** that can be formed with the help of a `set()` function:

```
empty_set = set()
print(type(empty_set))   # <class 'set'>

empty_dict = {}
print(type(empty_dict))  # <class 'dict'>
```

If you pass a string or a list into `set()`, the function will return a set consisting of all the elements of this string/list:

```
flowers = {'rose', 'lilac', 'daisy'}

# the order is not preserved
print(flowers)  # {'daisy', 'lilac', 'rose'}


letters = set('philharmonic')
print(letters)  # {'h', 'r', 'i', 'c', 'o', 'l', 'a', 'p', 'm', 'n'}
```

Each element is considered a part of a set only once, so double letters are counted as one element:

```
letters = set('Hello')
print(len(letters))  # the length equals 4
print(letters)       # {'H', 'e', 'o', 'l'}
```

Moreover, using sets can help you avoid repetitions:

```
states = ['Russia', 'USA', 'USA', 'Germany', 'Italy']
print(set(states))  # {'Russia', 'USA', 'Italy', 'Germany'}
```

Have a look: as the order of naming the elements doesn't play any role, the following two sets will be equal.

```
set1 = {'A', 'B', 'C'}
set2 = {'B', 'C', 'A'}
print(set1 == set2)  # True
```

§2. Working with a set's elements

You can:

- get the number of set's elements with the help of `len()` function.
- go through all the elements using `for loop`.
- check whether an element belongs to a specific set or not (`in / not in` operators), you get the boolean value.

```
nums = {1, 2, 2, 3}
print(1 in nums, 4 not in nums)  # True True
```

Current topic:

✓ Set

Topic depends on:

✓ Comparisons Stage 1 16

✓ List Stage 1 13

Topic is required for:

✓ Set operations

✓ Hashable

Table of contents:

- add a new element to the set with `add()` method or `update()` it with another collection

```
nums = {1, 2, 2, 3}
nums.add(5)
print(nums)  # {1, 2, 3, 5}

another_nums = {6, 7}
nums.update(another_nums)
print(nums)  # {1, 2, 3, 5, 6, 7}

# we can also add a list
text = ['how', 'are', 'you']
nums.update(text)
print(nums)  # {'you', 1, 2, 3, 5, 6, 7, 'are', 'how'}

# or a string
word = 'hello'
nums.add(word)
print(nums)  # {1, 2, 3, 'how', 5, 6, 7, 'hello', 'you', 'are'}
```

Note that when we update a set with a list, those are the elements of the list that are added to the set rather than the list itself.

- delete an element from a specific set using `discard/remove` methods. The only difference between them operating is a situation when the element to be removed is absent from this set. In this case, `discard` does nothing and `remove` generates a `KeyError` exception.

```
nums.remove(2)
print(nums)  # {1, 3, 5}

empty_set = set()
empty_set.discard(2)  # nothing happened
empty_set.remove(2)   # KeyError: 2
```

- remove one random element using `pop()` method. As it's going to be random, you don't need to choose an argument.

```
nums = {1, 2, 2, 3}
nums.pop()
print(nums)  # {2, 3}
```

- delete all elements from the set with `clear()` method.

§3. When to use sets?

One important feature of sets (and all unordered collections in general) is that they allow you to run membership tests *much faster* than lists. In real life, if you have a list and you try to check by hand whether a particular item is present there, the only way to do this is to look through the entire list until you find this item. Python does the same thing: it looks for the needed item starting from the beginning of a list, because it has no idea where it may be placed. If the item is located at the end or there is no such item at all, Python will iterate over the majority of items in the list by the time it discovers this fact. So, in case your program is looking for items in a large list many times, it will be slow.

And that's where sets come to help us! In sets membership testing works almost instantly, since they use a different way of storing and arranging values. So, depending on the situation, you need to decide what is more important to you: preserving the order of items in your collection or testing for membership in a faster way. In the first case, it's reasonable to store your items in the list, in the second it's better to use set.

§4. Frozenset

The only difference between `set` and `frozenset` is that set is a mutable data type, but frozenset is not. To create a frozenset, we use the `frozenset()` function.

```
empty_frozenset = frozenset()
print(empty_frozenset)  # frozenset()
```

We can also create a frozenset from a list, string or set:

```
frozenset_from_set = frozenset({1, 2, 3})
print(frozenset_from_set)  # frozenset({1, 2, 3})

frozenset_from_list = frozenset(['how', 'are', 'you'])
print(frozenset_from_list)  # frozenset({'you', 'are', 'how'})
```

As mentioned above, a frozenset is immutable. This means that while the elements of a set can change, in a frozenset they remain unchanged after creation. You can not add or remove items.

```
empty_frozenset.add('some_text')  # AttributeError: 'frozenset' object has no attr
ibute 'add'
```

So why do we need frozenset exactly? Since a set is mutable, we can't make it an element of another set.

```
text = {'hello', 'world'}
nested_text = {'!'}
nested_text.add(text)  # TypeError: unhashable type: 'set'
```

But with a frozenset, such problems will not appear. It can be an element of another set or an element of another frozenset due to its hashability and immutability.

```
some_frozenset = frozenset(text)
nested_text.add(some_frozenset)
print(nested_text)  # {'!', frozenset({'world', 'hello'})}
```

Also, these properties of frozensets allow them to be keys in a Python dictionary, but you will learn more about this later.

§5. Conclusions

All things considered, now you know how to work with sets:

- you know how to create a new set and what can be stored in a set (immutable data types only).
- you understand the difference between the set and other Python objects.
- you can work with a set's elements: add new elements or delete them, differentiate `discard` and `remove` methods, etc.
- you know when to use sets (this really can save your time!).
- you know that `frozenset` is an immutable alternative of set.

Report a typo

392 users liked this theory. **8** didn't like it. **What about you?**

Start practicing

This content was created over 1 year ago and updated 6 days ago. Share your feedback below in comments to help us improve it!

Comments (7) Hints (2) Useful links (0) Show discussion