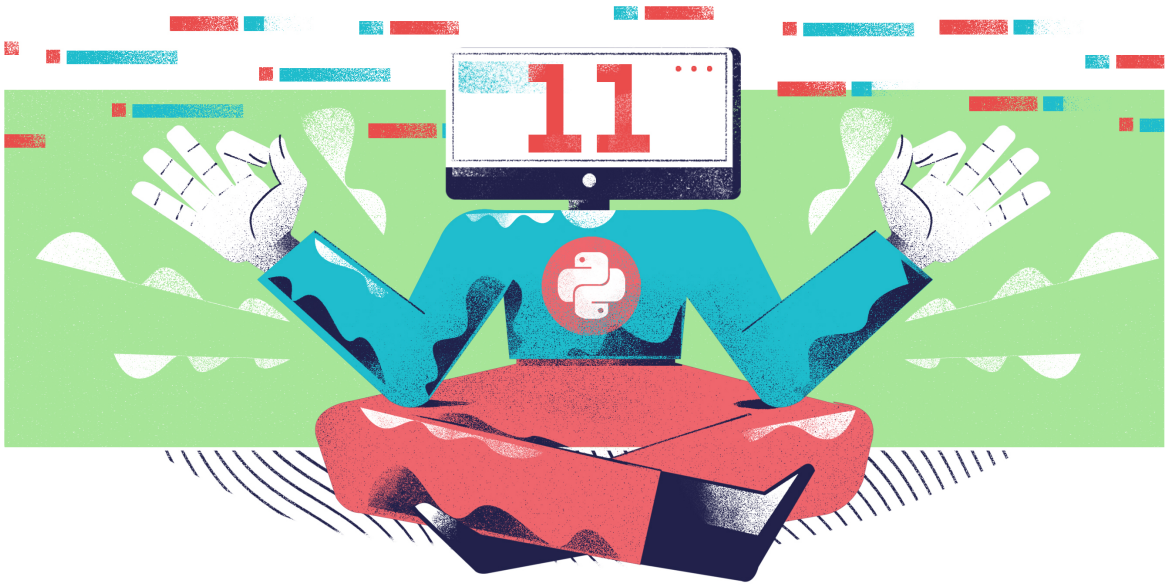
teclado

WORKING WITH DATA

Day 11: Sets



Welcome to day 11 of the 30 Days of Python series! In this post we're going to be talking about yet another collection type called sets. Sets are extremely useful for comparing collections of values, and are highly optimised for this kind of operation.

Sets are closely related to dictionaries, so if you're not familiar with dictionaries already, I'd strongly recommend you look at day 10 of this series. Much of what we're going to be talking about here is building on what we learnt in that post.

Sets

Sets are a little different to the collections we've looked at so far, in that

they're not reliably ordered. If we create a list or a tuple and print it, we know that the items will show up in the order we defined them. This isn't the case for a set.

Sets can also only contain unique elements, much like we saw with dictionary keys. In fact, we can really think of a set as a dictionary which only contains keys.

So why should we care about sets? For one, they come with a number of very useful operations for comparing the members of collections. They're therefore very useful for filtering values, and it's also extremely efficient to perform these kinds of membership tests for sets.

Defining a set

Much like dictionaries, we define sets using curly braces, but instead of key value pairs, we just add a series of comma separated values.

```
vegetables = {"carrot", "lettuce", "broccoli", "onion", "carr
ot"}
```

In this set we have the string, `"carrot"`, two times. This isn't going to produce an error, but the set will only contain one instance of `"carrot"`. The set we defined above is really identical to the one below.

```
vegetables = {"carrot", "lettuce", "broccoli", "onion"}
```

As we saw in day 10, we can define an empty dictionary using `{}`. Because of this, we can't use this option for defining an empty set. Instead, we have to write this:

```
set()
```

If we print a set which is empty, we'll also get this `set()` representation, to distinguish the output from an empty dictionary.

There are values that we can't include in a set, just like we saw with dictionary keys. The limitations are actually exactly the same, so we can't include any mutable types in a set, or any immutable types that contain mutable types.

Because of this, we can't add lists or dictionaries to a set, and we can't add tuples that contain things like lists or dictionaries.

We also can't include sets in other sets, because sets can also be modified. The following is therefore illegal in Python:

```
nested_sets = {{1, 2, 3}, {"a", "b", "c"}}
```

If we try, we'll get a `TypeError`:

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    nested_sets = {{1, 2, 3}, {"a", "b", "c"}}
TypeError: unhashable type: 'set'
```

Modifying sets

We have several ways to modify a set in Python, and most of these options are the same as those we have for dictionaries.

Adding items to a set

We have a few options for adding new items to a set.

First we have the `add` method, which allows us to add a single value to the set we call it on:

the set we call it on.

```
vegetables = {"carrot", "lettuce", "broccoli", "onion"}
vegetables.add("potato")

print(vegetables)  # {'lettuce', 'broccoli', 'onion', 'potat
o', 'carrot'}
```

Note that the print output is not in the same order as the set we defined. Your output will likely also be in a different order if you run the code yourself.

If we try to add an item which already exists in the set, the set is left unchanged.

If we want to add several items at once, we can instead use the `update` method. In this case, we can pass in any iterable, and the items it contains will be added if they are valid.

```
vegetables = {"carrot", "lettuce", "broccoli", "onion"}
vegetables.update(["potato", "pumpkin"])

print(vegetables)  # {'broccoli', 'lettuce', 'carrot', 'potat
o', 'pumpkin', 'onion'}
```

In either case, if we attempt to add a mutable item to the set, we'll get a `TypeError` similar to the one we saw before.

Deleting items from a set

We have a lot of options for removing items from a set as well.

First we have the `remove` method, which allows us to remove a single item. We have to specify what we want to remove when we call the method:

```
vegetables = {"carrot", "lettuce", "broccoli", "onion"}
vegetables.remove("lettuce")

print(vegetables)  # {'broccoli', 'carrot', 'onion'}
```

If we try to remove an item which doesn't exist, we get a `KeyError`. If we don't want this `KeyError` to be raised, we can use another method called `discard`.

`discard` works in exactly the same way, but it will only try to remove the item if it exists. If the item isn't in the set, the operation does nothing.

Both `remove` and `discard` require us to know what we want to remove, but sometimes it doesn't really matter. We just want one of the items. In these cases we should use the `pop` method.

Just like with other collections, `pop` gives us the item that was removed.

```
vegetables = {"carrot", "lettuce", "broccoli", "onion"}
random_vegetable = vegetables.pop()  # 'lettuce'

print(vegetables)  # {'broccoli', 'onion', 'carrot'}
```

Finally, we have the `clear` method, which will empty the set for us.

Set operations

Sets provide us with several new operations for efficiently comparing collections. The main operations we're going to be looking at today are union, intersection, difference, and symmetric difference.

If you've ever studied sets in mathematics, these terms will probably be

If you've ever studied sets in mathematics, these terms will probably be familiar to you.

Union

The union of two sets is essentially the combination of the two sets. The result includes all of the members of both sets, where duplicate members are included only once.

We can find the union of two sets with the `union` method.

```
letters = {"a", "b", "c"}
numbers = {1, 2, 3}

letters_and_numbers = letters.union(numbers)

print(letters_and_numbers)  # {'a', 'c', 1, 2, 3, 'b'}
```

`union` is very similar to the `update` method, but `union` creates a new set, while `update` modifies an existing set.

Intersection

When we find the intersection of two sets, we get a new set containing the elements common to both sets.

For example, if we have a set containing numbers divisible by 2, and another set with numbers divisible by 3, the intersection of these sets will contain only numbers divisible by 2 and 3.

```
mod_2 = {2, 4, 6, 8, 10, 12, 14, 16, 18}
mod_3 = {3, 6, 9, 12, 15, 18}

mod_6 = mod_2.intersection(mod_3)

print(mod_6)  # {18, 12, 6}
```

Difference

Difference

We have to be a little bit careful when using `difference`, because unlike the other set operations we've mentioned, with `difference` the order of the sets matters.

For example, let's say we have two bundles of games:

```
bundle_1 = {"Resident Evil 3", "Final Fantasy VII", "Cyberpun
k 2077"}
bundle_2 = {"Doom Eternal", "Halo Infinite", "Resident Evil
 3"}
```

If we call `difference` on `bundle_1`, passing in `bundle_2`, we will get a different result to if we call `difference` on `bundle_2`, passing in `bundle_1`:

```
print(bundle_1.difference(bundle_2))  # {'Final Fantasy VII',
'Cyberpunk 2077'}
print(bundle_2.difference(bundle_1))  # {'Halo Infinite', 'Do
om Eternal'}
```

What `difference` gives us, is every item in the set we called the method on, except those that also feature in this second set.

When we write this:

```
bundle_1.difference(bundle_2)  # {'Final Fantasy VII', 'Cyber
punk 2077'}
```

We get `"Final Fantasy VII"` and `"Cyberpunk 2077"`, because they feature in `bundle_1`, but not in `bundle_2`. `"Resident Evil 3"` isn't included, because it's in both sets.

Symmetric difference

`symmetric_difference` gives us all of the items which only feature in one of the sets. Unlike `difference` the order of the sets doesn't matter.

If we return to our game bundle example, we can see that the symmetric difference of `bundle_1` and `bundle_2` is all of the games except `"Resident Evil 3"`, because `"Resident Evil 3"` is the only game that features in both sets:

```
bundle_1 = {"Resident Evil 3", "Final Fantasy VII", "Cyberpun
k 2077"}
bundle_2 = {"Doom Eternal", "Halo Infinite", "Resident Evil
 3"}

print(bundle_1.symmetric_difference(bundle_2))

# {'Cyberpunk 2077', 'Final Fanstay VII', 'Halo Infinite', 'D
oom Eternal'}
```

Set operations with other collections

We can only call the set operator methods on sets, but we can actually pass in any collection we like into the method. This is because Python is going to convert the collection we pass in to a set before performing the operation.

For example, if we go back to our letters and numbers example, it's perfectly fine if `numbers` is a list, as long as we call the method on the `letters` set:

```
letters = {"a", "b", "c"}
numbers = [1, 2, 3]

letters_and_numbers = letters.union(numbers)

print(letters_and_numbers)  # {'a', 'c', 1, 2, 3, 'b'}
```

```
print(letters_and_numbers)  # {'a', 'c', 1, 2, 3, 'b'}
```

Checking if items are in collections

Very often we want to check if a value is in a collection, and we can perform this kind of check using the `in` keyword. `in` will yield `True` is the item is found, and `False` otherwise:

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

print(3 in numbers)  # True
print(7 in numbers)  # False
```

While this kind of test is extremely efficient for sets, we can use the `in` keyword with any collection. For example, we can check if a given letter is in a string:

```
print("j" in "Python")  # False
print("n" in "Python")  # True
```

We can also use `in` to check if a key is in a dictionary:

```
student = {
        "name": "Eric Cartman",
        "age": 10,
        "school": "South Park Elementary"
}

print("grades" in student)  # False
print("school" in student)  # True
```

Or among its values:

```
student = {
        "name": "Eric Cartman"
```

```
        "name": "Eric Cartman",
        "age": 10,
        "school": "South Park Elementary"
}

print(10 in student.values())  # True
```

Exercises

1) Create an empty set and assign it to a variable.

2) Add three items to your empty set using either several `add` calls, or a single call to `update`.

3) Create a second set which includes at least one common element with the first set.

4) Find the union, symmetric difference, and intersection of the two sets. Print the results of each operation.

5) Create a sequence of numbers using `range`, then ask the user to enter a number. Inform the user whether or not their number was within the range you specified.

If you want an extra challenge, also tell the user if their number was too high or too low.

You can find our solutions to the exercises here.

Extra Resources

In addition to the method syntax for set operations, we also have a number of operators we can use to accomplish the same task. You can read more about that on our blog.

We also have a more detailed information on `symmetric_difference` in another blog post.

If you're interested in learning more about sets, you can find a lot of additional information in the official documentation, including some additional methods.
