Reference guide: Sets

Data professionals depend on sets for separating data and identifying its unique elements. As you have been discovering, set objects are similar to lists and dictionaries, yet they do not have key-value pairs or positional index[i] capability. Additionally, sets contain unique values but have no item order or index behavior. Data professionals compare sets to understand the range of data they contain, where they intersect, and what items are present in either set but not both. Sets are also helpful when cleaning data for analysis. This reading is a reference guide for sets to help you as you continue learning Python.

Sets review

A set is a collection of unique data elements, without duplicates. In Python, it is an object class—in fact, two different classes—which you'll learn about in this reading. However, sets are not unique to Python or even to computer programming; they are an important concept in general mathematics. Sets provide a simple means to identify unique data elements.

Create a set

Create a set using braces:

```
my_set = {5, 10, 10, 20}
```

Note that an empty set cannot be created with braces, as this will be interpreted as an empty dictionary.

There are two functions for creating sets in Python: `set()` and `frozenset()`. Use these on any iterable object. Or use these functions to create empty sets.

`set()`

- This is a mutable data type.
- Because it's mutable, the class comes with additional methods to add and remove data from the set.

- It can be applied to any iterable object and will remove duplicate elements from it.
- It is unordered and non-indexable.
- Elements in a set must be hashable; generally, this means they must be immutable. (Refer to the additional resources for more on hashing.)

In the examples that follow, four sets are instantiated using a variety of data types:

```
example_a = [1, 2, 2.0, '2']
set(example_a)
```

Notice that, in this example, `2` and `2.0` are evaluated as equivalent, even though one is an integer and the other is a float.

```
example_b = ('apple', (1, 2, 2, 2, 3), 2)
set(example_b)
```

In this example, `(1, 2, 2, 2, 3)` is a tuple, which is hashable (≈ immutable) and thus treated as a distinct single element in the resulting set.

```
example_c = [1.5, {'a', 'b', 'c'}, 1.5]
set(example_c)
```

This example throws an error because each element of a set must be hashable (≈ immutable), but `{'a', 'b', 'c'}` is a set, which is a mutable (unhashable) object. The following example demonstrates the `add()` method, which is one of the special methods available to sets but not to frozensets.

```
example_d = {'mother', 'hamster', 'father'}
example_d.add('elderberries')
example_d
```

An element was added to the `example_d` set, thus modifying it. This is an example of the mutability of the `set` class.

frozenset()

Frozensets are another type of set in Python. They are their own class, and they are very similar to sets, except they are immutable.

- This is an immutable data type.
- It can be applied to any iterable object and will remove duplicate elements from it.
- Because they're immutable, frozensets can be used as dictionary keys and as elements in other sets.

  In this example, a frozenset is used within a set.

```
example_e = [1.5, frozenset(['a', 'b', 'c']), 1.5]
set(example_e)
```

  Unlike `example_c` previously, this set does not throw an error. This is because it contains a frozenset, which is an immutable type and can therefore be used in sets.

Set methods

Sets are useful to determine which values are contained in a data structure and to eliminate duplicate values. There are numerous set methods—such as intersection, union, difference, and symmetric difference—that add functionality and power to working with sets.

union()

- Return a new set with elements from the set and all others.
- The operator for this function is the pipe (|).

```
set_1 = {'a', 'b', 'c'}
set_2 = {'b', 'c', 'd'}

print(set_1.union(set_2))
print(set_1 | set_2)
```

intersection()

- Return a new set with elements common to the set and all others.
- The operator for this function is the ampersand (&).

```
set_1 = {'a', 'b', 'c'}
set_2 = {'b', 'c', 'd'}

print(set_1.intersection(set_2))
print(set_1 & set_2)
```

difference()

- Return a new set with elements in the set that are not in the others.
- The operator for this function is the subtraction operator (-).

```
set_1 = {'a', 'b', 'c'}
set_2 = {'b', 'c', 'd'}

print(set_1.difference(set_2))
print(set_1 - set_2)
```

symmetric_difference()

- Return a new set with elements in either the set or other, but not both.
- The operator for this function is the caret (^).

```
set_1 = {'a', 'b', 'c'}
set_2 = {'b', 'c', 'd'}

print(set_1.symmetric_difference(set_2))
print(set_1 ^ set_2)
```

Additional resources

- Refer to the Python documentation for more information about sets and frozensets, including a complete list of available class methods.
- For methods unique to sets (and unavailable to frozensets), refer to this Python set methods documentation.

- For more examples of sets, refer to the Python tutorial on sets.
- For more information on hash tables, what makes something hashable, and hashing as a concept, refer to this resource from Runestone Academy. For an interesting story about the birth of the original hashing algorithms, check out this IEEE Spectrum article.