1.5 Representing Data III: Collections

In the past two sections, we learned about three common types of data: numeric, boolean, and textual. We've focused so far on describing *individual* pieces of data of these types: how to write literals to represent these values, and what operations we can perform on them. Now, we'll look at three different data types that are used to represent *collections* of data, grouping multiple pieces of data together into a single entity.

Set data

Our first kind of collection is the **set** data type, which is a collection of zero or more *distinct* values, where *order does not matter*. Examples include: the set of all people in Toronto; the set of words of the English language; and the set of all countries on Earth.

Each piece of data contained in a set is called an **element** of that set. In your previous studies, you may have seen sets written by using curly braces, with each element inside the braces separated by commas. For example, $\{1, 2, 3\}$ or $\{\text{'hi'}, \text{'bye'}\}$.

In Python, we represent sets with the `set` data type. Set literals are written using curly braces, matching the mathematical notation we just described. Here are some examples of set literals:

```
>>> {'David'}
{'David'}
>>> {1, 2, 3}
{1, 2, 3}
>>> {1, 2.0, 'three'}
{1, 2.0, 'three'}
```

In Python, sets can have elements of the same type, or of different types. We call a set where every element has the same type a **homogeneous set**, and a set of where there are elements of different types a **heterogeneous set**. In this course we'll typically work with homogeneous sets, but occasionally it will be useful to have hetergeneous sets.

There is another way of writing sets: *set builder notation*, which defines the form of elements of a set using variables. We saw an example of this earlier when defining the set of rational numbers, $\mathbb{Q} = \{\frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q \neq 0\}$. We'll explore this notation in more detail in Section 1.7, but for now we'll stick with plain set literals to keep things simple.

Operations on sets

As with strings, there are many different operations we can perform on sets. In this section we'll just illustrate three that reuse Python operators we've already seen, and then in the next chapter we'll see a few more.

First we have **set equality**. Two sets are equal when they contain the exact same elements (ignoring what order the elements are written in). In Python, we (unsurprisingly!) use the `==` operator to compare sets.

```
>>> {1, 2, 3} == {3, 1, 2}
True
```

The analog of "substrings" for sets is the notion of subsets. Given two sets S_1 and S_2, we say that S_1 is a **subset** of S_2 when all elements of S_1 are contained in S_2.[1] In Python, we use the `<=` operator to write subset expressions:

[1] This is analogous to the definition of substrings, except without any mention of order, because order doesn't matter in sets.

```
>>> {1} <= {1, 2, 3}
True
>>> {1, 4} <= {1, 2, 3}
False
>>> {1, 2, 3} <= {1, 2, 3}  # Every set is a subset of its
True
```

You might have noticed an inconsistency with strings here: Python uses `in` to check for substrings, but `<=` to check for subsets. This is because `in` is used to express a different fundamental set operation: **element checking**. In mathematics, we use the symbol $\in$ to mean "is an element of" or "is in", e.g. $1 \in \{1, 2, 3\}$. In Python, we use the `in` operator to express the same computation:

```
>>> 1 in {1, 2, 3}
True
>>> 10 in {1, 2, 3}
False
```

List data

The second form of collection we will study is the **list**, which is a sequence of zero or more values that may contain duplicates. Like sets, the values contained in a list are called the *elements* of the list. Unlike sets, lists can contain duplicates, and order matters in a list.[2] List data is used instead of a set when the elements of the collection should be in a specified order, or if it may contain duplicates. Examples include: the list of all people in Toronto, ordered by age; the list of words of the English language, ordered alphabetically, and the list of names of students at the University of Toronto (two students may have the same name!), ordered alphabetically.

[2] So both strings and lists are examples of sequences, where order matters. In fact, in some programming languages (but not Python) strings are represented simply as lists of characters, rather than a separate data type.

In mathematics, lists are written using square brackets with each element contained in the list separated by commas, for example, $[1, 2, 3]$. In Python, lists are represented using the `list` data type,[3] and list literals use this same syntax:

[3] There is another closely related Python data type called `tuple` that we'll see later in this course.

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['David']
['David']
>>> [1, 2.0, 'three']
[1, 2.0, 'three']
```

Like sets, Python lists can be either **homogeneous** (all elements have the same type) or **heterogenous** (elements have different types).

Operations on lists

As with sets, we'll just illustrate a few different operations on lists here, leaving others to future chapters.

First, **list equality**: two lists are equal when the contain the same elements in the same order. In Python, we use `==` to compare lists for equality:

```
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] == [3, 2, 1]  # Unlike sets, order matters for lists
False
```

Like sets, lists support **element checking** using the `in` operator:

```
>>> 3 in [1, 2, 3]
True
>>> 10 in [1, 2, 3]
False
```

And like strings, lists support **list concatenation** (using the `+` operator) and **list indexing** (using square brackets), where the indexing also starts at 0.

```
>>> ['David', 'Mario'] + ['Jacqueline', 'Diane']
['David', 'Mario', 'Jacqueline', 'Diane']
>>> (['David', 'Mario', 'Jacqueline', 'Diane'])[0]
'David'
```

Mapping data (association pairs)

While sets and lists have some important differences, they share one key feature: they are both groups of individual elements. However, another common form of grouped data is *associations* from one collection of data to another. For example: associations from the name of a country to its GDP; associations from University of Toronto student number to name; associations from food item to price on a restaurant menu.

So our final data type for this section is the **mapping**, which is a collection of *association pairs*,[4] where each pair consists of a **key** and **associated value** for that key. (Note that keys are themselves pieces of data like names or numbers.) In a mapping, each key must be unique, but values can be duplicated. A key cannot exist in the mapping without a corresponding value.

[4] "Association pairs" are also called "key-value pairs".

For example, to represent a restaurant menu with a mapping, each food item would be a key, and the corresponding value would be its price. In the mapping, each food item is unique (and must be associated with exactly one price), but it is possible for two food items to have the same price.

We use curly braces to represent a mapping.[5] Each key-value association pair in a mapping is written using a colon, with the key on the left side of the colon and its associated value on the right. For example, here is how we could write a mapping representing the menu items of a restaurant:

[5] This is similar to sets, because mappings are quite similar to sets. Both data types are unordered, and both have a uniqueness constraint (a set's elements are unique; a mapping's keys are unique).

$$\{\text{`fries'} : 5.99, \text{`steak'} : 25.99, \text{`soup'} : 8.99\}$$

In Python, we represent mappings using the `dict` (short for "dictionary") data type, and write literals using the syntax described above.

```
>>> {'fries': 5.99, 'steak': 25.99, 'soup': 8.99}
{'fries': 5.99, 'steak': 25.99, 'soup': 8.99}
>>> {1: 'one', 2.5: 110, 'three': False}
{1: 'one', 2.5: 110, 'three': False}
```

Like sets and lists, Python dictionaries can have both keys and associated values of different types. A **homogeneous dictionary** is a dictionary where every key has the same type, and every associated value has the same type, but the key type and associated value type can be different.[6] A **heterogeneous** dictionary is a dictionary where the keys have different types or the associated values have different types, or both.

[6] So the "menu" dictionary example above is a homogenous dictionary.

Operations on mappings

We'll end off by discussing three fundamental operations on mappings.

As usual, our first operation is **mapping equality**: two mappings are equal when they contain the exact same key-value pairs. In Python, we use `==` to compare two dictionaries for equality.

```
>>> {'fries': 5.99, 'steak': 25.99, 'soup': 8.99} == {'fries': 5.99, 'steak': 25.99, 'soup': 8.
True
>>> {1: 'David', 2: 'Mario'} == {2: 'Mario', 1: 'David'}  # order does not matter in mappings
True
>>> {1: 'David', 2: 'Mario'} == {1: 'Mario', 2: 'David'}
False
```

The second operation is analogous to element checking for sets and lists: **key checking**. Given a mapping M and value k, we use $k \in M$ to express that k is a *key* in M. In Python, we accomplish the same task with the `in` operator:

```
>>> 'fries' in {'fries': 5.99, 'steak': 25.99, 'soup': 8.9
True
```

One warning: in Python there is no equivalent operator to check whether there is a given "associated value" in a dictionary. So for example, we can't use `in` to check for the presence of prices in our menu example:

```
>>> 5.99 in {'fries': 5.99, 'steak': 25.99, 'soup': 8.99}
False
```

The third operation on mappings is **key lookup**. Given a mapping M and key k that appears in the mapping, this operation returns the value that is associated with k in M. We use the same square bracket syntax as string/list indexing, writing $M[k]$ to denote this operation. Here is an example of this syntax in Python:

```
>>> {'fries': 5.99, 'steak': 25.99, 'soup': 8.99}['fries']
5.99
```

That's it for now—as with all of the other data types, we'll explore Python dictionaries in more detail throughout this course. Before we wrap up, we'll leave you with one final note about empty collections.

Empty collections

All three kinds of collections we've studied allow for a collection of size zero. An empty set or list contains zero elements, and an empty mapping contains zero key-value pairs. In Python, the literal `[]` represents an empty list.

But we have a problem with sets and dictionaries: both of their literals use curly braces in Python, so does `{}` represent an empty set or an empty dictionary? The answer (for historical reasons) is that `{}` is the literal for an empty dictionary—Python has no literal to represent an empty set. Instead, we can create an empty set in Python with the expression `set()`, which is syntax we haven't seen yet but will explore in the next chapter.

Summary

The collection data types are a bit more complex that the earlier data types we looked at, and they might blur together as you are first encountering them. We've put together a summary table for the three data types we studied in this section to help you review what you've learned.

Abstract data type	set	list	mapping
Python data type	`set`	`list`	`dict`
Description	collection of elements	sequence of elements	collection of association (key-value) pairs
Example Python literal	`{1, 2, 3}`	`[1, 2, 3]`	`{1: 'one', 2: 'two'}`
"Empty" Python literal	no literal, but use `set()`	`[]`	`{}`
Order matters?	no	yes	no
May contain duplicates?	no	yes	no duplicate keys, but possibly duplicate values
Definition of homogeneous	all elements have same type	all elements have same type	all keys have same type, and all values have same type
Equality checking	`==` operator	`==` operator	`==` operator
Element checking	`in` operator	`in` operator	`in` operator (for keys only)
Other operations	subset checking (`<=`)	concatenation (`+`) indexing (`...[...]`)	key lookup (`...[...]`)

References

- Appendix A.2 Python Built-In Data Types Reference