Dictionaries

Lesson Duration: 60 minutes

Learning Objectives

- Understand what a dictionary is
- Understand the advantages of a dictionary
- Be able to create a dictionary
- Be able to retrieve items from a dictionary
- Be able to add items to a dictionary
- Be able to modify an item in a dictionary

What Are Dictionaries

We have seen that we can store a collection of objects in a list. But our data is not always going to be simple integers, strings or booleans. What happens if the thing we are trying to describe with data is more complex.

Let's create a file called `dictionaries.py` in our `session_2` folder.

Imagine we are trying to describe the result of a test - going for student first name, last name, the subject and the result.

```
student = ["Anna", "Henderson", "chemistry", 92]
```

This could techbnically work - however we need to know specifically which data can be found under which index number. If you remember, we mentioned in the last lesson that we should not keep track of the order of items in a list - especially bevause we can easily shuffle it, completely rendering the list representing an exam value useless.

It would be great to give each piece of data an annotiation to show what it represents - and without the chance of shuffling it.

Enter dictionaries - or `dicts` as Python calls them (you might have heard about them as hashes, maps, associative arrays or objects in other languages)

```
student = {
  "first_name": "Anna",
  "last_name": "Henderson",
  "subject": "chemistry",
  "result": 92
}
```

In a `dict` every item is given a unique key of our choosing (which should be a string) and it is this key that is used to retrieve the vale associated with it rather than an index. Due to this, the order is irrelevant.

Even though they share some similarities with `lists`, they have fundamentally different uses - lists are collections of similar data, while a `dict` is used to describe more complex entities with more details. They can even be used together, like a list of dicts, which we will see later!

Using Dictionaries

Creating Dictionaries

Create a new file called `dictionaries.py`.

We have a couple of options for initialising an empty dictionary. We can create a new empty dictionary using braces, AKA curly brackets (`{}`), or by using the `dict` (short for dictionary) function.

```
# dictionaries.py
my_first_empty_dictionary = {}
my_second_empty_dictionary = dict()
```

We can also create dicts with key-value pairs already in them, as seen above. A key-value pair is written by linking the `str` key to the value with a colon (`:`). Each key value-pair is separated from the next pair with a comma (`,`).

> Be careful with the syntax! A misplaced comma, or a missing `:` can cause syntax errors. If you encounter them, make sure you read through your code carefully and compare even the tiny differences in your code!

```
student = {
  "first_name": "Anna",
  "last_name": "Henderson",
  "subject": "chemistry",
  "result": 92
}
```

In this dictionary, we have keys which are `str` and values which are `str`s or `int`s. Keys in general should be `str`s, or `int`s, although the latter doesn't make much sense - you might as well use a `list`, because it, by default, has `int`s for indexes, acting in a similar fashion!

Values, on the other hand, can be pretty much anything - other `str`s or `int`s, `list`s, `boolean`s, or even other `dict`s!

```
student = {
  "first_name": "Anna",
  "last_name": "Henderson",
  "subject": "chemistry",
  "result": 92,
  "contact_details": {
    "phone": "+441234567",
    "email": "anna@example.com"
  }
}
```

Accessing Elements

We can access elements in a similar manner to lists, using the square brackets - the main difference is, we have to use the key, instead of an index.

```
print(student["result"])
```

If we try to access an element for which there is no key we get a `KeyError`.

```
print(student["random_stuff"])
```

If we want to check if a key exists in a dictionary we can use the `in` keyword

```
# dictionaries.py
print ("first_name" in student)
# => True

print ("random" in student)
# => False
```

It's important to understand that if we access something in a `dict`, the datatype changes from a `dict` to the type of the value accessed!

```
print(type(student))
# => <dict>

print(type(student["result"]))
# => <int>
```

Basically, `dict`s could contain different types of data

This will be majorly important later! Refer to this part later if you're struggling with this concept!

Modifying Elements

We can add or modify key-value pairs by using either non-existing or existing keys and assigning new values.

```
# This replaces a value
student["result"] = 99
print(student)

# This adds a new value
student["date_of_birth"] = "1985/01/01"
print(student)
```

We can remove items using the `del()` method:

```
del(student["subject"])
print(student)
```

Helpful Methods

A dictionary has lots of helpful methods, including ways to list all the keys:

```
# dictionaries.py
print(list(student))
# => ['first_name', 'last_name', 'result']

print(list(student))
# => dict_keys['first_name', 'last_name', 'result']
```

`dict_keys` are practically the same as a `list`.

```
# dictionaries.py
print(student.values())

# => dict_values(['Anna', 'Henderson', 'chemistry', 92])
```

Like the `keys()` method the `values()` method also returns a list-like object.

Nested Dictionaries

Let's make a quick dictionary of the UK.

```
united_kingdom = {
    "population": 67000000,
    "capital": "London"
}

print(united_kingdom)
```

This is fine, but what if we also want to store the details of London? Would we make a separate dictionary?

We can actually store a dictionary inside of a dictionary! Sounds scary, but it can actually be very useful. Let's add the population of London to our UK dict!

```
# dictionaries.py
united_kingdom = {
    "population": 67000000,
    "capital": {
      "name": "London",
      "population": 9000000
    }
}

print(united_kingdom)
```

Task

Try to print out the population of London!

```
# dictionaries.py
london_population = united_kingdom["capital"]["population"]
print(london_population)
```

Conclusion

We have seen how we can use dictionaries to display more complex data, representing details about a single entity.