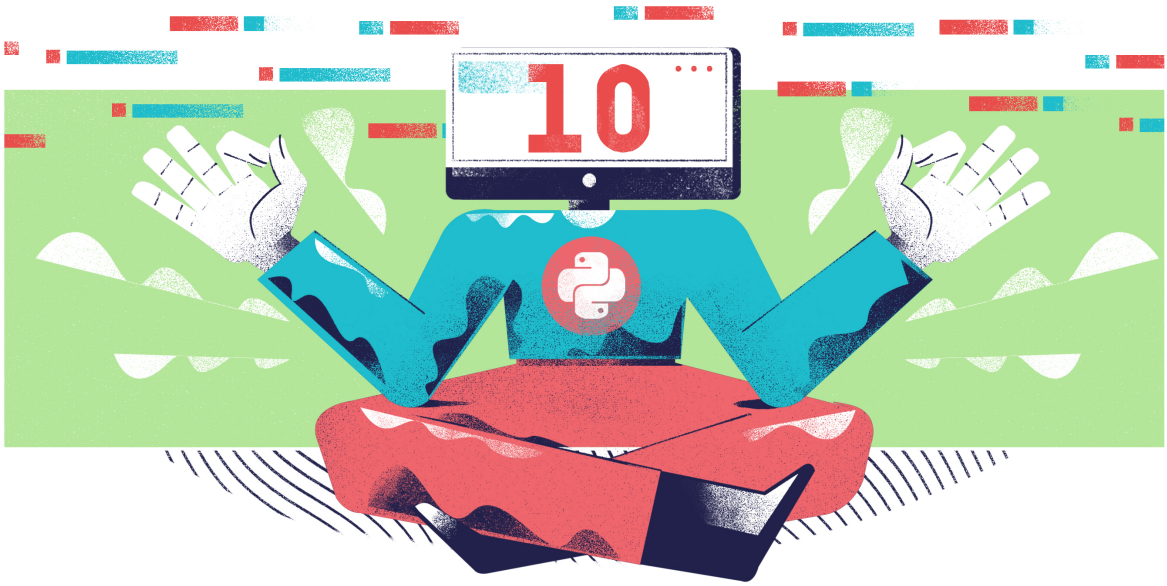
teclado

WORKING WITH DATA

# Day 10: Dictionaries




Welcome to day 10 of the 30 Days of Python series! In this post we're going to be talking about a new type of collection called dictionaries.

Last time we covered unpacking iterables, the `enumerate` function, and the ever so useful `zip` function. If you're not familiar with any of these tools and techniques, I'd recommend checking out day 9, as we're going to be using some of these tools again here.

Also we've created a short recap of Days 8 and 9 that you can view here to refresh your memory.

## What is a dictionary?

Dictionaries are Python's version of something called an *associative array*, and it works a little differently to the things like lists, strings, and tuples.

The collections we've looked at so far have all been *sequence types*, and we access their values by index. If we change the order of a sequence, we might refer to a given value by a different index after the modification, because the indices really only refer to a position in the collection.

**Sequence** is a technical term in Python, which refers to a collection which satisfies the *sequence protocol*. For something to be a sequence, it must have ordered indices which start from `0`, and which increment in steps of `1`. It must also be possible to find the length of the collection with `len`.

For example, if we have the following list:

```
simpsons = ["Homer", "Marge", "Bart", "Lisa", "Maggie"]
```

We can refer to `"Homer"` by referring to index `0` of the `simpsons` list. However, if we sort the list alphabetically, which we can do with the `sort` method, we now have to refer to `"Homer"` using index `1`.

```
simpsons = ["Homer", "Marge", "Bart", "Lisa", "Maggie"]
simpsons.sort()

print(simpsons)  # ["Bart", "Homer", "Lisa", "Maggie", "Marg
e"]
```

In an associative array, and therefore a dictionary, values are instead associated with some other term. If we change the order of the values, it doesn't matter, because each value is still going to be associated with

the same term. These associated terms are called keys.

A **key** is a little bit like a variable is that it's a thing which we can use to refer to some value. Unlike variables, keys are themselves values, as they're written as strings, integers, etc.

```
In a single dictionary, each key must be unique, but differen
t dictionaries can have the same keys as each other.
```

A dictionary can have many keys, but each key must be unique inside that dictionary, and each key must have a single value. That value can, however, be a collection, including another dictionary.

One common use case for dictionaries is very similar to how we've been using tuples to represents lines in a table. For example, instead of having a tuple like this to represent a movie:

```
("Up", "Pete Doctor", 2009)
```

We could use a dictionary with three keys: `"title"`, `"director"`, and `"release_year"`. We could then retrieve a value we want by asking the dictionary for the value associated with a given key. For example, we might want the value associated with the key `"title"`. In this case, we'd be handed the string, `"Up"`.

## Defining a dictionary

We create a dictionary using a pair of curly braces:

```
{}
```

This is actually an empty dictionary, much like `[]` is an empty list.

When we want to create a dictionary with content, we have to work in

pairs of keys and values. A dictionary can't have a key without a value,

and we can't have a value which isn't associated with a key.

Let's say we're creating a dictionary for a student, and we want to start out with a key and value representing their name:

```
student = {"name": "John Smith"}
```

First we have the key—which in this case is the string, `"name"` — followed by a colon ( `:` ), and then the value associated with that key.

If we want to add multiple keys and values, we have to separate each key value pair using commas.

For example, let's add a list of grades for this student alongside their name:

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
}
```

## Style note

In the example above, the dictionary has been broken up over several lines, much like we did with our tuples and lists. This is common when the dictionary is getting quite long, as it helps with readability.

Don't feel like you need to cram everything on to a single line like this:

```
student = {"name": "John Smith", "grades": [88, 76, 92, 85, 6
9]}
```

If the dictionary can easily fit on one line, make sure to always put a

space after each comma, just like we do for other collections.

Another important factor in creating readable dictionaries is adding a space after the colon. This helps to separate keys and values, making it quicker to spot them at a glance.

## What can we use as dictionary key names?

There are also some technical limitations as to what we can use as a key in Python dictionaries.

As we've already seen, strings are perfectly fine, but so are numbers, and even tuples. However we can't ever use a list as a key. Why not? Because a list can change.

A key feature of dictionaries is that their keys are unique. This is very important for retrieving values, as we're about to see. The problem with lists is that they provide an opportunity to violate this rule, because we can change a list after setting it as a key. By adding or removing items, we could modify a pair of lists so that they end up being identical.

The fact that this can occur means that Python simply won't allow us to use lists as keys, and the same applies to any other type we can modify.

Some of you may be wondering why we can use strings as keys, since it seems like we can modify stings as well. However, we never actually modify a string: we always create a new one.

Just now I mentioned we can use a tuple as a key, but there are some limitations here as well. This is because tuples can contain things like lists, and we can modify those inner lists.

For example, we can do this:

```
student = (
        "John Smith",
        [88, 76, 92, 85, 69]
```

```
)

# Append 77 to the list at index 1
student[1].append(77)

print(student)  # ('John Smith', [88, 76, 92, 85, 69, 77])
```

We can therefore only use a tuple which doesn't contain any mutable values as keys in a dictionary.

## Accessing values in a dictionary

We can access values in a dictionary using a subscription expression, just like we do for lists and tuples. However, instead of using an index, we retrieve values using their keys.

Let's look at our student example and try to get the grades list out of this dictionary:

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
}

print(student["grades"])  # [88, 76, 92, 85, 69]
```

As you can see, this comes with some significant readability benefits over using something like a tuple. While using indices can be pretty opaque, the keys describe the data they refer to. At least, they do if we used good key names!

So, what happens if we refer to a key that doesn't exist?

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
```

```
}


print(student["grade"])  # KeyError
```

Here I've tried to refer to a key called `"grade"`, but the real key is called `"grades"`. What we get in this case is a `KeyError`, which is very similar to the `NameError` we get for trying to access an undefined variable.

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(student["grade"])
KeyError: 'grade'
```

If we're unsure if a key exists, and we don't want an exception being raised, we can instead use a method called `get`. If the key exists, `get` works exactly the same way as using a subscription expression, but if the key doesn't exist, it will return `None` instead of crashing the program.

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
}

print(student.get("grade"))  # None
```

If we like, we can specify a different default value by passing in a second value to `get`. For example, we can tell it we want an empty list if there isn't a `"grade"` key:

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
```

```
}


print(student.get("grade", []))  # []
```

# Modifying a dictionary

There are three ways we can modify an existing dictionary. We can add new key value pairs; we can modify the values of existing keys; or we can remove a key value pair. Let's start with adding new key value pairs.

## Adding items to a dictionary

All we need to do to add a new key value pair to a dictionary is assign a value to a key which doesn't yet exist. Let's see an example:

```
student = {
        "name": "John Smith",
        "grades": [88, 76, 92, 85, 69]
}

student["age"] = 17
```

He're we've assigned a value of `17` to the key `"age"` in our dictionary. `"age"` doesn't exist, so Python creates it for us. If we print this dictionary, we'll see something like this:

```
{
        'name': 'John Smith',
        'grades': [88, 76, 92, 85, 69],
        'age': 17
}
```

We can also extend a dictionary using an existing dictionary and the `update` method. Let's look at a new example:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}
```

Here we have a dictionary containing some fairly basic information about Avengers: Endgame. We just have the movie title, the directors, and the year of release.

Let's imagine we have some secondary information about the movie in another dictionary like this:

```
meta_info = {
        "runtime": 181,
        "budget": "$356 million",
        "earnings": "$2.798 billion",
        "producer": "Kevin Feige"
}
```

What I want is to take our original `movie` dictionary and supplement it with the information from `meta_info` so that we end up with `movie` containing all of these keys and values. We accomplish this like so:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

meta_info = {
        "runtime": 181,
        "budget": "$356 million",
        "earnings": "$2.798 billion",
```

```
        "producer": "Kevin Feige"
}

movie.update(meta_info)
```

If we print `movie`, it'll now look something like this:

```
{
        'title': 'Avengers: Endgame',
        'directors': ['Anthony Russo', 'Joe Russo'],
        'year': 2019,
        'runtime': 181,
        'budget': '$356 million',
        'earnings': '$2.798 billion',
        'producer': 'Kevin Feige'
}
```

If we wanted, we could also just pass in a dictionary to `update` directly, instead of defining `meta_info` beforehand:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

movie.update({
        "runtime": 181,
        "budget": "$356 million",
        "earnings": "$2.798 billion",
        "producer": "Kevin Feige"
})
```

This can be very useful if we're working with shorter dictionaries, or just if we're not using this other dictionary for any other purpose.

## Modifying existing items in a dictionary

## Modifying existing items in a dictionary

The approach for modifying items is actually exactly the same as when we want to add new ones. Instead of using a key which doesn't exist, we can assign a value to an existing key, which will replace the current value:

```
student = {
        'name': 'John Smith',
        'grades': [88, 76, 92, 85, 69],
        'age': 17
}

student["age"] = 18
```

We can also use the `update` method to either replace many values at once, or perform some combination of addition and replacement. Any keys which already exist will have their values replaced, while new keys will be created like we saw before.

## Deleting items from a dictionary

The approach for deleting items from a dictionary is very similar to what do for lists.

Our first option is using `del` to delete a specific key from the dictionary, which will remove the key value pair:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019,
        "runtime": 181
}

del movie["runtime"]
```

This will leave us with a dictionary like this:

```
{
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}
```

We can also use `pop`, passing in a key. Much like with lists, this will also return the value that was removed.

If we refer to a key which doesn't exist using either of these methods, we get a `KeyError`.

We also have the `clear` method, which will completely empty a dictionary, just like when we used it with lists.

## Iterating over dictionaries

One thing you need to be aware of when iterating over dictionaries is that by default, we only get the keys. For example, let's try to print each item in our `movie` dictionary:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

for attribute in movie:
        print(attribute)
```

What we get in this case is this:

```
title
directors
```

```
directors
year
```

One common thing I see people trying to do when they want the values is this:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

for key in movie:
        print(movie[key])
```

This works, but we really don't need to resort to something like this. We can just call the `values` method on the dictionary to get an iterable containing the values for each key:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

for value in movie.values():
        print(value)
```

But what about if we want the key and value? Do we have to go back to an approach like this?

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

for key in movie:
```

```
for key in movie:
        print(f"{key}: {movie[key]}")
```

Thankfully not! We can use the `items` method instead. This is going to give us a series of tuples, where the first item in each tuple is the key, and the second item is the value.

We can destructure these tuples to get what we want:

```
movie = {
        "title": "Avengers: Endgame",
        "directors": ["Anthony Russo", "Joe Russo"],
        "year": 2019
}

for key, value in movie.items():
        print(f"{key}: {value}")
```

## The order of items in a dictionary

In modern Python (since version 3.7), dictionaries have been ordered as part of the language specification. This order is based on the order we add items into the dictionary in question. If we put the `"title"` key before the `"artist"` key, this will be reflected in the dictionary we create. However, this wasn't always the case.

If, for some reason, you find yourself forced to use an older Python version, such as Python 3.5, you won't be able to rely on the order of the values in your dictionaries. This is going to become relevant when we try to iterate over dictionaries, or when we try to unpack their contents into variables.

## Exercises

1) Below is a tuple describing an album:

```
(
        "The Dark Side of the Moon",
        "Pink Floyd",
        1973,
        (
                "Speak to Me",
                "Breathe",
                "On the Run",
                "Time",
                "The Great Gig in the Sky",
                "Money",
                "Us and Them",
                "Any Colour You Like",
                "Brain Damage",
                "Eclipse"
        )
)
```

Inside the tuple we have the album name, the artist (in this case, the band), the year of release, and then another tuple containing the track list.

Convert this outer tuple to a dictionary with four keys.

2) Iterate over the keys and values of the dictionary you create in exercise 1. For each key and value, you should print the name of the key, and then the value alongside it.

3) Delete the track list and year of release from the dictionary you created. Once you've done this, add a new key to the dictionary to store the date of release. The date of release for The Dark Side of the Moon was March 1st, 1973.

If you use a different album for the exercises, update the date accordingly.

4) Try to retrieve one of the values you deleted from the dictionary. This should give you a `KeyError`. Once you've tried this, repeat the step

using the `get` method to prevent the exception being raised.

## Additional resources

The `update` method can accept values in a few different formats, not just the one we saw in this post. If you're interested, we have a blog post dedicated to `update` that goes into more detail.

In addition, we have a post talking about dictionaries and loops.

If you want to learn more about dictionaries, the Python documentation is also a great place to look.

---