

Chapter 2

Data structures and script flow

Objectives

- Describe the basics of lists and dictionaries.
- Parse an unknown GeoJSON structure.
- Explore if/else logic.
- Create a function.
- Use a `for` loop.
- Use a `while` loop.

Introduction

As you're working through your Python projects, you'll inevitably encounter some complex data structures. These can include files written in markup languages, such as JSON, YAML, or XML, that you're given as input data to serve as parameters or as an output from your script. Regardless of whether you're ingesting or creating these data structures, it's handy to have a firm grasp on how to treat this data Pythonically ("Pythonic" refers to the idioms of Python usage, readability, conciseness, and clarity).

In this chapter, you'll work through some basics of lists and dictionaries as Python data types. We'll assume you already have a good understanding of basic Python data types, such as strings, integers, and booleans. You'll review some properties of lists and dictionaries and learn how you can use those properties to interrogate and modify more complex data structures. You'll also look at conditional logic and looping. Ultimately, you'll build up to how you can use these fundamentals to explore and summarize the data in a GeoJSON file.

Tip: Exercise data accompanies the tutorials. Review the section at the beginning of the book for details and instructions on how to access the data.

JSON as Python

A lot of the complex data structures you encounter working with data are based on JSON. JSON stands for JavaScript Object Notation and is extremely common. As you're working with online data, a lot of APIs traffic in JSON data (or at least give you that as an option).

At its most basic level, JSON consists of objects that are key or value pairs enclosed by curly brackets. This is analogous to the dictionary data type in Python. JSON objects also often come in arrays, which are similar to lists in Python. Things often come in arrays, which hold nested data and are similar to an object that contains an array of objects, each of which possibly contains additional objects or arrays. If you see a GeoJSON file, you can be relatively certain that the file will contain some spatial data and that the file roughly conforms to the GeoJSON specification. It may contain combinations of points, lines, or polygons, and each feature may have different attributes. Because of this, using Python to explore the structure of a JSON file is a handy skill to have.

Tutorial 2-1: Data structures—lists and dictionaries

Sometimes you may be unsure about the structure or quality of the data you're given. Maybe a colleague gave it to you, or you got the data from an API. In many cases (especially when you're working with web services and APIs), the data you're given is JSON.

In this tutorial, you'll review the basics of working with lists and dictionaries in Python and learn how this enables you to explore an unknown JSON file Pythonically.

List basics

In this section, you'll walk through some basics of lists and how you can work with them. It's worth having these concepts at hand when you're trying to figure out the structure of some new or unknown data.

First, you'll create a list. The easiest way to create your own list in Python is to write it yourself. Doing so gives you a great idea of how lists are structured. Start by assigning a variable name to your new list, and then all you need to do is put some values between square brackets with commas to separate them.

```
# create a list
list_1 = [1, 2, 3, 5, 7, 8]
```

Now that you've made a list, you can print that list or use functions on that list.

```
print(list_1)

len(list_1)
```

Tip: The function `len()` is a built-in Python function that tells you the number of items in a container. It works with lists, tuples, strings, and many other data types.

Use an index to access values in the list

Now that you've created a list, you can access the items in the list using a special property of lists. The `index` values of the list items represent each item in order. The index starts at 0, which represents the first item in the list. The list index increases by 1 for each subsequent value, so the second item in the list has an index of 1.

You can access an item in a certain position in the list by using square brackets with your list variable, as shown in this example.

```
| list_1[0]
```

This returns a value of 1.

```
| list_1[1]
```

This returns a value of 2, which is the second item in the list.

Try this with additional numbers. It will work until you reach the end of the list.

Use an index to access the last value in the list

The index starts at the beginning of a list, so counting higher from 0 is always an option. Sometimes you would be more interested in the last item in the list. In that case, you can use `-1` to start at the end.

```
| list_1[-1]
```

This returns a value of 8, which is the last item in the list.

You can also count backward by increasing the negative index. Using `-2` gives you the second-to-last item in the list.

```
| list_1[-2]
```

This returns a value of 7, which is the second-to-last item in the list.

Use indexes to slice a list

In the same way that you access a single value in a list, you can also pull out multiple values. This is often called slicing the list. Using square brackets, you can provide two index numbers with a colon. This will retrieve all the list items, including the first index and up to (but not including) the second index.

```
| list_1[1:-2]
```

This returns a new list with the values 2, 3, and 5.

Save the slice as a new list

You can always save the values returned by indexing or slicing into the list as new variables.

```
| list_2 = list_1[1:-2]
```

Add a new item to a list

One of the great things about lists is that they are mutable—they can be changed. You can use the `.append()` method to add items to the end of a list.

```
| list_2.append("a new value")
```

Note that you previously had a list of all numbers, but now you have added a string. Python is permissive in allowing you to put whatever you want in a list. This capability can be handy or problematic. Imagine trying to do the sum or max of a list of numbers but finding out that there's a string included.

```
list_2
[2, 3, 5, 'a new value']
```

Sum the values in a list

Because you have a list of numbers, you can use the `sum()` function to add them all. It's worth noting that this function works only on lists of numbers. If you try to use it on a list with strings or other data types, you'll get an error.

```
sum(list_1)
26

sum(list_2)

-----
TypeError                                Traceback (most recent call last)
~\AppData\Local\Temp\1\ipykernel_31192\3152765143.py in <cell line: 0>()
----> 1 sum(list_2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Remove an item from a list

Similarly to how you can add items to a list, you can also take them out. You use a function called `pop()` to remove an item. You'll provide the index of the item you want to remove, and `pop` will return that item.

```
| list_2.pop(-1)
```

This returns 'a new value', which was the last item in that list (represented by index `-1`). Now when you look at the `list_2` values, the popped item is no longer in the list.

Tip: Now that you have removed the string 'a new value', `list_2` contains only numeric values. You can use the `sum()` function on that list without encountering an error.

Dictionary basics

Lists are great, but sometimes you need a little more structure in your data. Dictionaries can be helpful in these situations.

Create a dictionary

You can create your own dictionary by using curly brackets. If you provide an open and closed curly bracket (`{}`), you'll end up with an empty dictionary. You can also create a dictionary populated with values by providing comma-separated key-value pairs.

```
| dict_1 = {  
    'key1': 'value1',  
    'key2': 'value2',  
    'key3': 'value3'  
}
```

Each key and its corresponding value are separated by a colon. Each pair is separated by a comma.

Print the keys and values

One way you can get a good idea of what's in a dictionary is by looking at the keys. You can do this by calling the `.keys()` method on the dictionary.

```
dict_1.keys()  
✓ 0.0s  
dict_keys(['key1', 'key2', 'key3'])
```

You can also do this with the `.values()` method.

```
dict_1.values()
✓ 0.0s
dict_values(['value1', 'value2', 'value3'])
```

Access a specific key-value pair

Once you understand what the keys in a dictionary are, you can use a key to retrieve its corresponding value from the dictionary. You can do this by providing the key to the dictionary with square brackets.

```
| dict_1['key1']
```

This returns `'value1'`, which is the corresponding value to this key.

Update the value for a key

In addition to using a key to access a value, you can also use that key to replace the corresponding value.

```
| dict_1['key1'] = 'updates!'
```

Now the value that corresponds with `'key1'` has been changed. When you retrieve the value at `'key1'`, you'll get the value `'updates!'`.

```
| dict_1['key1']
```

Add a new key-value pair

You can use the same syntax to add new key-value pairs to a dictionary.

```
| dict_1['a new key'] = 'a new value'
```

At this point, you have worked with all string values for keys and values, but you can use whatever data types you want. In this example, you'll add a list as a value to the dictionary.

```
| dict_1['a list'] = list_2
```

Combine two dictionaries

Occasionally, you need to combine two dictionaries. You can iterate through each key from one dictionary and add it to another, but there's a handy built-in method on the dictionary class called `.update()` that does that for you.

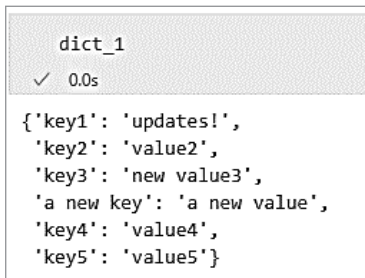
```
| # create a second dictionary
| dict_2 = {
```

```

    'key3': 'new value3',
    'key4': 'value4',
    'key5': 'value5'
}

# combine two dictionaries
dict_1.update(dict_2)

```



```

dict_1
✓ 0.0s
{'key1': 'updates!',
 'key2': 'value2',
 'key3': 'new value3',
 'a new key': 'a new value',
 'key4': 'value4',
 'key5': 'value5'}

```

Note that the keys from `dict_2` that didn't exist in `dict_1` were added. The one key that existed in both dictionaries (`'key3'`) was updated with the value from `dict_2`.

Parsing an unknown GeoJSON structure

It's important to understand the fundamentals of lists and dictionaries. They're handy for gathering and organizing your own data as you write a script.

These same tools are highly useful for exploring JSON datasets. The arrays and objects that make up JSON are represented in Python as lists and dictionaries. It's common in the workplace to be given JSON data without context. Being able to quickly explore the data can give you a good idea of what the data looks like without having to open the whole file (which might be quite large).

Load JSON data

First, you'll load some data. You'll need to import the `json` package to handle the conversion between JSON and Python data types.

```
import json
```

You'll use a context manager to open the file. There's a longer discussion about context managers later in the book. In short, though, the context manager is represented by the `with` statement in the following code. It saves you from having to open and close the file reader.

Once you have a file reader created using the context manager, you can use the `json.load()` function to convert the file into Python data types.

```

with open('data.geojson') as f:
    data = json.load(f)

```