Week 2 Web Applications and Services - Reading Material

Module 2 Introduction

Congratulations on making it through the first lab in this course!

Remember that putting your Python skills to practice with exercises like this one is the way to getting the hang of it all. By practicing solving complex challenges, you'll become a lot more confident in your programming abilities, and you'll be able to achieve a lot more.

In this module, we'll look into a bunch of different tools that can be really useful in today's IT world. You'll first learn how you can use different text formats to store data in text files, retrieve it, and even transmit it over the internet.

Later on, we'll look into how we can get our code to interact with services running on different computers using a module called Python Requests.

We'll see a bunch of different examples and give you pointers to more information. Don't forget that the best way to get comfortable with all these modules and libraries is to come up with your own examples and practice writing scripts on your local computer!

Web Applications and Services

A ***web application*** is an application that you interact with over HTTP. Most of the time when you're using a website on the Internet, you're interacting with a web application. So, how does this look behind the scenes?

Your web browser sends an HTTP request to a web server. Then, the web server passes the request along to the web application in charge of deciding what information to show you. The application then generates the website content (in HTML format). The application is also in charge of serving images and any other necessary data so that your web browser can render the website on your computer.

Lots of web applications also have APIs that you can use from your scripts! Web applications that have an API are also known as ***web services***. Instead of browsing to a web page to type and click around, you can use your program to send a message known as an ***API call*** to the web service. The part of the program that listens on the network for API calls is called an ***API endpoint***.

When you interact with a web service like this, you don't even care what language the other application is using. You interact with it using a specified protocol, and the only important constraint is that both the service and your program know how to use this protocol.

So, how does that work? That's coming up!

Data Serialization

If you have two programs that need to communicate with each other, how do you get that data from one place to another? We're going to talk about two aspects of that problem: what to send, and how to send it.

First, what do you send? When you have a conversation with another person, you don't send thoughts and memories directly between your brains. At least not yet! You first have to convert your thoughts into language, and then transmit that language to another person. They take that language, and convert it back into thoughts. It's the same with programs running in different places, or at different times.

In a previous course, we took a list of lists in memory and wrote it to disk as a ***Comma-Separated Value (CSV)*** file. This is one example of a technique called ***data serialization***. Data serialization is the process of taking an in-memory data structure, like a Python object, and turning it into something that can be stored on disk or transmitted across a network. Later, the file can be read, or the network transmission can be received by another program and turned back into an object again. Turning the serialized object back into an in-memory object is called ***deserialization***.

Data serialization is extremely useful for communicating with web services. A web service's ***API endpoint*** takes messages in a specific format, containing specific data. By the end of this module, we'll be sending messages to web services, but for now let's concentrate on how to serialize Python objects into some common formats.

Let's start with the contact information from one of our CSV examples. We'll keep just two entries to keep our examples short, but there's no limit to how long these can be.

```
name,username,phone,department,role

Sabrina Green,sgreen,802-867-5309,IT Infrastructure,System Administrator

Eli Jones,ejones,684-3481127,IT Infrastructure,IT specialist
```

Instead of having a list of lists, we could turn this information into a list of dictionaries. In each of these dictionaries, the key will be the name of the column, and the value will be the corresponding information in each row. It could look something like this:

```
people = [

      {

      "name": "Sabrina Green",

      "username": "sgreen",

      "phone": "802-867-5309",

      "department": "IT Infrastructure",

      "role": "Systems Administrator"
```

```
        },
        {
        "name": "Eli Jones",
        "username": "ejones",
        "phone": "684-348-1127",
        "department": "IT Infrastructure",
        "role": "IT Specialist"
        },
]
```

Using a structure like this lets us do interesting things with our information that's much harder to do with CSV files. For example, let's say we want to record more than one phone number for each person. Instead of using a single string for "phone", we could represent that data in another dictionary, like this:

```
people = [
        {
        "name": "Sabrina Green",
        "username": "sgreen",
        "phone": {
              "office": "802-867-5309",
              "cell": "802-867-5310"
        },
        "department": "IT Infrastructure",
        "role": "Systems Administrator"
        },
        {
        "name": "Eli Jones",
        "username": "ejones",
        "phone": {
```

```
            "office": "684-348-1127"

        },

        "department": "IT Infrastructure",

        "role": "IT Specialist"

        },

]
```

Now, we can record multiple phone numbers per person, and give them descriptive names like "office" and "cell". This would be hard to store in a CSV file, because the data is not *flat*. To help us with that, there's a bunch of different formats that we can use to store our data when the structure isn't flat.

we'll look into a few of the most common ones.

There are lots and lots of ways to serialize data. In this course, we'll cover a couple of the most common ones and we'll look into how you can use them from Python. Once you get the hang of how this works, it's super easy to use a different format if needed.

JSON (JavaScript Object Notation) is the serialization format that we'll use the most in this course. We'll go into some details later but, for now, let's just use the **json** module to convert our **people** list of dictionaries into JSON format.

```
import json

with open('people.json', 'w') as people_json:

        json.dump(people, people_json, indent=2)
```

This code uses the **json.dump()** function to serialize the **people** object into a JSON file. The contents of the file will look something like this:

```
[

  {

        "name": "Sabrina Green",

        "username": "sgreen",

        "phone": {

        "office": "802-867-5309",

        "cell": "802-867-5310"

        },
```

```
        "department": "IT Infrastructure",
        "role": "Systems Administrator"
    },
    {
        "name": "Eli Jones",
        "username": "ejones",
        "phone": {
        "office": "684-348-1127"
        },
        "department": "IT Infrastructure",
        "role": "IT Specialist"
    },
]
```

YAML (Yet Another Markup Language) has a lot in common with JSON. They're both formats that can be easily understood by a human when looking at the contents. In this example, we're using the **yaml.safe_dump()** method to serialize our object into YAML:

```
import yaml
with open('people.yaml', 'w') as people_yaml:
        yaml.safe_dump(people, people_yaml)
```

That code will generate a **people.yaml** file that looks like this:

```
- department: IT Infrastructure
  name: Sabrina Green
  phone:
      cell: 802-867-5310
      office: 802-867-5309
  role: Systems Administrator
  username: sgreen
- department: IT Infrastructure
```

```
name: Eli Jones

phone:

    office: 684-348-1127

role: IT Specialist

username: ejones
```

While this doesn't look exactly like the JSON example above, both formats list the names of the fields as part of the format, so that both the programs *parsing* the data and the humans looking at it can make sense out of it. The main difference is how these formats are used. JSON is used frequently for transmitting data between web services, while YAML is used the most for storing configuration values.

These are just a couple of the most common data serialization formats. We've left out some other pretty common ones like ***Python pickle***, ***Protocol Buffers***, or the ***eXtensible Markup Language (XML)***. Each of them is useful in a specific context, although not the focus of this course. You can read more about them by following those links.

More About JSON

Alright, we've seen a couple of different serialization formats. Let's now dive into more details about ***JSON (JavaScript Object Notation)***, which you'll be using in the lab at the end of this module.

As we mentioned before, JSON is ***human-readable***, which means it's encoded using printable characters, and formatted in a way that a human can understand. This doesn't necessarily mean that you *will* understand it when you look at it, but you *can*.

Lots of web services send messages back and forth using JSON. In this module, and in future ones, you'll serialize JSON messages to send to a web service.

JSON supports a few ***elements*** of different data types. These are very basic data types; they represent the most common basic data types supported by any programming language that you might use.

JSON has ***strings,*** which are enclosed in quotes.

```
"Sabrina Green"
```

It also has ***numbers,*** which are not.

```
1002
```

JSON has ***objects***, which are key-value pair structures like Python dictionaries.

```
{
```

```
  "name": "Sabrina Green",
  "username": "sgreen",
  "uid": 1002
}
```

And a key-value pair can contain another object as a value.

```
{
  "name": "Sabrina Green",
  "username": "sgreen",
  "uid": 1002,
  "phone": {
      "office": "802-867-5309",
      "cell": "802-867-5310"
  }
}
```

JSON has ***arrays***, which are equivalent to Python lists. Arrays can contain strings, numbers, objects, or other arrays.

```
[
  "apple",
  "banana",
  12345,
  67890,
  {
      "name": "Sabrina Green",
      "username": "sgreen",
      "phone": {
      "office": "802-867-5309",
      "cell": "802-867-5310"
```

```
        },
        "department": "IT Infrastructure",
        "role": "Systems Administrator"
    }
]
```

And as you've probably noticed, JSON elements are always ***comma-delimited***. With these basics under your belt, you could create valid JSON by hand, and edit examples of JSON that you encounter. Except we don't really want to do that, since it's clunky and we're bound to make a ton of errors! Instead, let's use the **json** library that does all the heavy lifting for us.

```
Import json
```

The **json** library will help us turn Python objects into JSON, and turn JSON strings into Python objects! The **dump()** method serializes basic Python objects, writing them to a file. Like in this example:

```
import json

people = [
    {
        "name": "Sabrina Green",
        "username": "sgreen",
        "phone": {
        "office": "802-867-5309",
        "cell": "802-867-5310"
        },
        "department": "IT Infrastructure",
        "role": "Systems Administrator"
    },
    {
        "name": "Eli Jones",
        "username": "ejones",
        "phone": {
```

```
        "office": "684-348-1127"

        },

        "department": "IT Infrastructure",

        "role": "IT Specialist"

    }

]

with open('people.json', 'w') as people_json:

        json.dump(people, people_json)
```

That gives us a file with a single line that looks like this:

```
[{"name": "Sabrina Green", "username": "sgreen", "phone": {"office": "802-867-5309", "cell": "802-867-5310"}, "department": "IT Infrastructure", "role": "Systems Administrator"}, {"name": "Eli Jones", "username": "ejones", "phone": {"office": "684-348-1127"}, "department": "IT Infrastructure", "role": "IT Specialist"}]
```

JSON doesn't *need* to contain multiple lines, but it sure can be hard to read the result if it's formatted this way! Let's use the **indent** parameter for **json.dump()** to make it a bit easier to read.

```
with open('people.json', 'w') as people_json:

        json.dump(people, people_json, indent=2)
```

The resulting file should look like this:

```
[

    {

        "name": "Sabrina Green",

        "username": "sgreen",

        "phone": {

        "office": "802-867-5309",

        "cell": "802-867-5310"

        },

        "department": "IT Infrastructure",

        "role": "Systems Administrator"
```

```
  },
  {
      "name": "Eli Jones",
      "username": "ejones",
      "phone": {
      "office": "684-348-1127"
      },
      "department": "IT Infrastructure",
      "role": "IT Specialist"
  }
]
```

Now it's much easier to follow! In fact, it looks very similar to how you'd write out the object in Python. Cool!

Another option is to use the **dumps()** method, which also serializes Python objects, but returns a string instead of writing directly to a file.

```
>>> import json
>>>
>>> people = [
...   {
...   "name": "Sabrina Green",
...   "username": "sgreen",
...   "phone": {
...   "office": "802-867-5309",
...   "cell": "802-867-5310"
...   },
...   "department": "IT Infrastructure",
...   "role": "Systems Administrator"
...   },
```

```
...   {
...   "name": "Eli Jones",
...   "username": "ejones",
...   "phone": {
...   "office": "684-348-1127"
...   },
...   "department": "IT Infrastructure",
...   "role": "IT Specialist"
...   }
... ]
>>> people_json = json.dumps(people)
>>> print(people_json)
[{"name": "Sabrina Green", "username": "sgreen", "phone": {"office":
"802-867-5309", "cell": "802-867-5310"}, "department": "IT Infrastructure",
"role": "Systems Administrator"}, {"name": "Eli Jones", "username": "ejones",
"phone": {"office": "684-348-1127"}, "department": "IT Infrastructure",
"role": "IT Specialist"}]
```

The **load()** method does the inverse of the **dump()** method. It deserializes JSON from a file into basic Python objects. The **loads()** method also deserializes JSON into basic Python objects, but parses a string instead of a file.

```
>>> import json
>>> with open('people.json', 'r') as people_json:
...   people = json.load(people_json)
...
>>> print(people)
[{'name': 'Sabrina Green', 'username': 'sgreen', 'phone': {'office':
'802-867-5309', 'cell': '802-867-5310'}, 'department': 'IT Infrastructure',
'role': 'Systems Administrator'}, {'name': 'Eli Jones', 'username': 'ejones',
'phone': {'office': '684-348-1127'}, 'department': 'IT Infrastructure',
'role': 'IT Specialist'}, {'name': 'Melody Daniels', 'username': 'mdaniels',
'phone': {'cell': '846-687-7436'}, 'department': 'User Experience Research',
'role': 'Programmer'}, {'name': 'Charlie Rivera', 'username': 'riverac',
```

```
'phone': {'office': '698-746-3357'}, 'department': 'Development', 'role':
'Web Developer'}]
```

Remember that JSON elements can only represent simple data types. If you have complex Python objects, you won't be able to automatically serialize them as JSON. Take a look at this table to see in detail how Python objects are converted into JSON elements.