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.

```
Eli Jones,ejones,684-3481127,IT Infrastructure,IT specialist
name,username,phone,department,role
Sabrina Green,sgreen,802-867-5309,IT Infrastructure,System Administrator
```

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.

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

Data Serialization Formats

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:

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

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 Dani
els', '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.

The Python Requests Library

Up to now, we've seen how we can serialize the data that we have in our programs and turn it into a format that we can store on disk. Once the data is stored, another process can open up those files, de-serialize them, and go from there.

This works, but only if the other process has access to the same filesystem you used to store your data. What if you wanted to send a message to another computer on another network? HTTP to the rescue!

Remember that ***HTTP (HyperText Transfer Protocol)*** is the protocol of the world-wide web. When you visit a webpage with your web browser, the browser is making a series of ***HTTP requests*** to web servers somewhere out on the Internet. Those servers will answer with ***HTTP responses***. This is also how we're going to send and receive messages with web applications from our code.

The Python Requests library makes it super easy to write programs that send and receive HTTP. Instead of having to understand the HTTP protocol in great detail, you can just make very simple HTTP connections using Python objects, and then send and receive messages using the methods of those objects. Let's look at an example:

```
>>> import requests
>>> response = requests.get('https://www.google.com')
```

That's it! That was a basic request for a web page! We used the Requests library to make a ***HTTP GET*** request for a specific ***URL, or Uniform Resource Locator***. The URL tells the Requests library the name of the resource (**www.google.com**) and what protocol to use to get the resource (**https://**). The result we get is an object of type requests.Response.

Alright, now what did the web server respond with? Let's take a look at the first 300 characters of the Response.text:

```
>>> print(response.text[:300])
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="de"><head><me
ta content="text/html; charset=UTF-8" http-equiv="Content-
Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itempro
p="image"><title>Google</title><script nonce="dZfbIAn803LDGXS9
```

Now, it might be hard for you to read the HTML (HyperText Markup Language) that was returned in this response, but your web browser knows just how to turn that into a familiar-looking web page.

Even with this simple example, the Requests module has done a whole lot of work for us! We didn't have to write any code to find the web server, make a network connection, construct an HTTP message, wait for a response, or decode the response. Not that HTML can't be messy enough on its own, but let's look at the first bytes of the ***raw*** message that we received from the server:

```
>>> response = requests.get('https://www.google.com', stream=True)
>>> print(response.raw.read()[:100])
b'\x1f\x8b\x08\x00\x00\x00\x00\x00\x02\xff\xc5Z\xdbz\x9b\xc8\x96\xbe\xcfS`\xf2\xb5-
\xc6X\x02$t\xc28\xe3v\xdc\xdd\xee\xce\xa9\xb7\xdd;\xe9\x9d\xce\xf6W@\t\x88\x11`@>D\xd6\x9b
\xce\xe5<\xc3\\\xcd\xc5\xfc\xab8\x08\xc9Nz\x1f.&\x8e1U\xb5j\xd5:\xfc\xb5jU\x15\x87;^\xe2\x
16\xf7)\x97\x82b\x1e\x1d\x1d\xd2S'
```

What's all that? The response was ***compressed*** with ***gzip***, so it had to be ***decompressed*** before we could even read the text of the HTML. One more thing that the Requests library handled for us!

The requests.Response object also contains the exact request that was created for us. We can check out the headers stored in our object to see that the Requests module told the web server that it was okay to compress the content:

```
>>> response.request.headers['Accept-Encoding']
'gzip, deflate'
```

And then the server told us that the content had actually been compressed.

```
>>> response.headers['Content-Encoding']
'gzip'
```

And all this happened by default, without us having to do anything special to make it work. Amazing, right?

Useful Operations for Python Requests

There's a ton of things that we can do with Python Requests. We'll cover some of the most important features here and give you pointers for more information at the end.

First, how do we know if a request we made got a successful response? You can check out the value of ***Response.ok***, which will be **True** if the response was good, and **False** if it wasn't.

```
>>> response.ok
True
```

Now, keep in mind that this will only tell you if the web server says that the response successfully fulfilled the request. The response module can't determine if that data that you got back is the kind of data that you were expecting. You'll need to do your own checking for that!

If the boolean isn't specific enough for your needs, you can get the HTTP response code that was returned by looking at Response.status_code:

```
>>> response.status_code
200
```

Excellent! To write maintainable, stable code, you'll always want to check your responses to make sure they succeeded before trying to process them further. For example, you could do something like this:

```
response = requests.get(url)
if not response.ok:
    raise Exception("GET failed with status code {}".format(response.status_code))
```

But you don't really need to do that. Requests has us covered here, too! We can use the Response.raise_for_status() method, which will raise an **HTTPError** exception *only if* the response wasn't successful.

```
response = requests.get(url)
response.raise_for_status()
```

Up next, we'll look into the different types of HTTP request methods that we can make using this handy requests module.

HTTP GET and POST Methods

HTTP supports several ***HTTP methods***, like GET, POST, PUT, and DELETE. We're going to spend time on the two most common HTTP requests: GET and POST.

The ***HTTP GET method***, of course, retrieves or ***gets*** the resource specified in the URL. By sending a GET request to the web server, you're asking for the server to GET the resource for you. When you're browsing the web, most of what you're doing is using your web browser to issue a whole bunch of GET requests for the text, images, videos, and so forth that your browser will display to you.

A GET request can have ***parameters***. Have you ever seen a URL that looked like this?

```
https://example.com/path/to/api/cat_pictures?search=grey+kitten&max_results=15
```

The question mark separates the URL resource from the resource's parameters. These parameters are one or more key-value pairs, formatted as a ***query string***. In the example above, the **search** parameter is set to "grey+kitten", and the **max_results** parameter is set to 15.

But you don't have to write your own code to create an URL like that one. With requests.get(), you can provide a dictionary of parameters, and the Requests module will construct the correct URL for you!

```
>>> p = {"search": "grey kitten",
...      "max_results": 15}
>>> response = requests.get("https://example.com/path/to/api", params=p)
>>> response.request.url
'https://example.com/path/to/api?search=grey+kitten&max_results=15'
```

You might notice that using parameters in Requests is yet another form of data serialization. Query strings are handy when we want to send small bits of information, but as our data becomes more complex, it can get hard to represent it using query strings.

An alternative in that case is using the ***HTTP POST method***. This method sends, or ***posts***, data to a web service. Whenever you fill a web form and press a button to submit, you're using the POST method to send that data back to the web server. This method tends to be used when there's a bunch of data to transmit.

In our scripts, a POST request looks very similar to a GET request. Instead of setting the **params** attribute, which gets turned into a query string and appended to the URL, we use the **data** attribute, which contains the data that will be sent as part of the POST request.

```
>>> p = {"description": "white kitten",
...      "name": "Snowball",
...      "age_months": 6}
>>> response = requests.post("https://example.com/path/to/api", data=p)
```

Let's check out the generated URL for this request:

```
>>> response.request.url
'https://example.com/path/to/api'
```

See how much simpler the URL is on this POST now? Where did all of the parameters go? They're part of the ***body*** of the HTTP message. We can see them by checking out the **body** attribute.

```
>>> response.request.body
'description=white+kitten&name=Snowball&age_months=6'
```

Ah, ha! There they are!

So, if we need to send and receive data from a web service, we can turn our data into dictionaries and then pass that as the **data** attribute of a POST request.

Today, it's super common to send and receive data specifically in JSON format, so the Requests module can do the conversion directly for us, using the **json** parameter.

```
b'{"description": "white kitten", "name": "Snowball", "age_months": 6}'
>>> response = requests.post("https://example.com/path/to/api", json=p)
>>> response.request.url
'https://example.com/path/to/api'
>>> response.request.body
```

And that's it for our brief introduction to the Requests module. If you want to learn more, feel free to work through the Requests Quickstart.

In the project at the end of this module, you'll use the Requests module to interact with a web application. This simple application was created using the Django web framework. So, what's that, exactly? Read on to learn more!

What is Django?

The lab project at the end of this module will feature a very simple web application created using ***Django***. Django is a ***full-stack web framework*** written in Python. For this project, you'll only need to interact with it through HTTP requests, but it's still a good idea to understand what it is, and when it would be a good tool for you to use.

A full-stack web framework handles a bunch of different components that are typical when creating a web application. It contains libraries that help you handle each of the pieces: writing your application's code, storing and retrieving data, receiving web requests, and responding to them. If you need to build an application that has a web frontend, using a web framework like Django can save you a lot of time and effort, because a lot of challenges are already solved for you.

Web frameworks are commonly split into three basic components: (1) the application code, where you'll add all of your application's logic; (2) the data storage, where you'll configure what data you want to store and how you're storing it; and (3) the web server, where you'll state which pages are served by which logic.

Splitting your code like that helps you write more modular code, promotes code reuse, and allows for flexibility when viewing and accessing data. For example, you could have a simple web page where users of the system can access the information already stored in it, and a separate programmatic interface that can be used by other scripts or applications to transmit data to the system.

When you're writing a web application, there's a ton of little decisions to make. Relying on a framework like Django is similar to using external libraries for your code. There are a lot of features, which you can use very easily, instead of writing everything from scratch and re-making all of the same mistakes that we all make when writing a web application for the first time.

Django has a ton of useful components for building websites. In the lab project, Django will be used for serving the company website, including customer reviews. It does this by taking the request for a URL and parsing it using the ***urlresolver*** module. This is a core module in Django that interprets URL requests and matches them against a list of defined patterns. If a URL matches a pattern, the request is passed to the associated function, called a ***view***. This allows you to serve different pages depending on what URL is being requested. You can even build complex logic into the function handling the request to make more dynamic, interactive, and exciting pages.

Django can also handle reading and writing data from a database, letting you store and retrieve data used by your application. In the lab, the database holds the customer reviews for the company. When a user loads the website, the logic will ask the database for all available customer reviews. These are retrieved and formatted into a web page, which is served as a response to the URL request. Django makes it easy to interact with data stored in a database by using an ***object-relational mapper***, or ***ORM***. This tool provides an easy mapping between data models defined as Python classes and an underlying database that stores the data in question.

On top of this, the Django application running in the lab includes an ***endpoint*** that can be used to add new customer reviews to the database. This endpoint is configured to receive data in JSON format, sent through an HTTP POST request. The data transmitted will then be stored in the database and added to the list of all reviews. The framework even generates an interactive web form, that lets us directly interact with the endpoint using our browser, which can be really handy for testing and debugging.

Django is one of many popular web frameworks. Alternative Python-based web frameworks similar to Django include Flask, Bottle, CherryPy, and CubicWeb. There are a host of other frameworks written in other languages too, not just Python.