Getting Data from the Internet With Python

In addition to reading files locally, you can also read them over the internet.

When you use a web browser like Chrome to go to a URL ("uniform resource locator," or web address) like https://sociologicalgobbledygook.com, what you're actually doing is sending a request using the HTTPS protocol (which is the same at the HTTP protocol, but with some more encryption on top). HTTP stands for "hypertext transfer protocol," and it was originally designed for the web. There are other protocols—you may have seen FTP, for example, which stands for "file transfer protocol," and "SMTP" which stands for "simple mail transfer protocol," but HTTP/S is by far the most important. It turns out that people make information other than web pages available through that protocol, which you can request the same way you request a web page. And you don't need a web browser to do it.

HTTP requests are made up of "verbs," which are just capitalized (by convention) words that describe what you want to do. Servers out on the internet will ask you to use one verb or another in order to access their resources. The two most important verbs are GET and POST.

GET requests

A GET request is the basic request of the web. When you send a GET request to a URL, you're asking for the resources associated with that location. (Sometimes, internet people use the word "endpoint" to mean a specific location on a server.). Ordinary web access through a browser is made up of GET requests: when you type "https://sociologicalgobbledygook.com" into your browser, you make a GET request to the route endpoint of that website.

You can send data to a server via a GET request. When you do, the data is actually built into the URL itself. If you've ever seen a URL like https://sociologicalgobbledygook.com/catinfo?cat=Leonidas&info=cutenessquotient then you've seen data being sent to the server that way. The stuff after the question mark is known as a "query string," and it is a collection of key-value pairs much like a Python dictionary. The URL above translates to "hey server at sociologicalgobbledygook.com, I'd like to tell your 'catinfo' endpoint that the cat is 'Leonidas' and the info is 'cutenessquotient'." Typically, the server will use that data to find some information and generate a response to send back. For example, the catinfo endpoint might be programmed to connect to a database containing information about cats, so that the server will look up Leonidas in that database and find his cuteness quotient (it's *INFINITY* in case you were curious), and then send that information back to you.

That being said, in most cases you won't want to write your own query strings to send data via GET requests. There are complicated rules for how query strings are composed (because there are a bunch of things that aren't allowed in them,

like spaces). Instead, you'll use code written by someone else to translate a more useful format into a query string before sending them along.

POST requests

POST requests are designed for sending more substantial data to a server. For example, suppose you run a service that allows people to upload files over the web: you'd tell them to submit those files via a POST request. You can also send longer text data through a POST request, and lots of other things besides. Typically, when you see a form on a web page (like a registration form, or a survey), the information that you fill in there gets sent to the server in a POST request. POST requests come in a variety of formats, but the most common and important is "multipart/form-data"—when in doubt, it's safe to assume that this is what you're sending.

I'll explain how to send a POST request below.

Headers

In addition to the content, or "body," of a request, HTTP requests also have headers. You can think of this like the headers of a document, they tend to contain metadata, or information about the information inside, such as the size of the data being sent. For the most part, you don't need to worry about headers, but there are a couple of important pieces of information that they tend to contain. First, they often contain authentication information: many servers will require you to send a username and password, or a "token" (a special string issued by the server that identifies you, sort of like the server's equivalent of a social security number) in a particular format in the headers in order to be allowed to get access to the data they have. Second, they often contain a description of the type of content being sent. For example, if you send text, it might contain information about the character encoding (we'll learn about that in a subsequent lesson) that the text is in. If you send a file, it might contain information about the format the file is in. More about formats and encodings below.

Responses

Ok, so you've sent a GET or a POST request to a server somewhere. The thing you get back is called a response. It has several parts.

First is a response code: a 3-digit number saying, in broad terms, what the server thinks of your response. Have you ever navigated to a web page that doesn't exist and gotten a 404 error? 404 is a response code that means exactly what you think it means: that the server can't find the endpoint you asked for

(or the webpage you asked for). Generally, any response code in the 400s is the server's way of telling you that you screwed up—another common one is 403, which means "forbidden"—typically returned by a server when you ask for something that requires authentication, but you haven't given it the password or whatever other authentication resource it wants in a form that it recognizes. 401 means roughly the same thing.

Typically, the response code you want to get is 200, which means "ok." Anything in the 200 range is going to be a desired response code. If you get anything in the 500 range, it usually signifies a programming error on the server side (i.e., not your fault).

The response will also contain information about the format of the content inside, though its Content-Type header. A response often has other headers too, but Content-Type tends to be the most important (along with Content-Encoding), as they will tell you what kind of content you're getting back, which is information that tends to be necessary in order to do anything with it!

The Requests library

Probably the most popular way to work with HTTP requests in Python is through a library called Requests, which makes it very easy to send requests and deal with responses. It's a third party library (not distributed with Python itself), but it comes pre-installed for you on Azure Notebooks, and also will already be installed if you've followed the personal machine installation instructions for this course and installed the Anaconda distribution.

To get access to the Requests library if it's installed, you can just use the import command. Enter the following line of code:

```
import requests
```

Then you'll have a collection of useful functions available to you in the requests namespace. The most important are `requests.get` and `requests.post` which, as you might expect, allow you to send GET and POST requests.

Suppose you wanted to get the home page for this course. You could run

```
response = requests.get("https://sociologicalgobbledygook.com")
```

This will assign the HTTP response from the server for the course home page to a variable called "response." This variable is an object provided by the Requests library that contains a bunch of useful methods and attributes. These include:

`response.text` will give you the text of the webpage, in this case, a string of HTML code. `response.status_code` will give you the status code of the response. Hopefully it's 200. `response.headers` will give you a Dictionary containing all the headers of the response. So you can find out things like the Content-Type by asking for `response.headers["Content-Type"]`.

Requests also lets you send data with your get and post requests. Take a look at the Requests documentation to see how. It's pretty straightforward: here's how to send ordinary data in a GET request and here's how to send ordinary data in a POST request.

APIs

Programmers talk a lot about APIs. API stands for "application programming interface," and just means a set of commands that programmers make for other programmers to use. Software has an API—for example, the fact that Requests gives you get and post functions is part of the API of the Requests library.

Internet resources also have APIs. The API of an internet resource typically describes the endpoints you can make requests to, the format those requests have to be in, and the kind of data you'll get back.

For example, there's a wonderful service called Courtlistener that contains massive amounts of case and court data from the American legal system. The documentation for their API explains how to get that information via programming. (The Courtlistener API is very complicated, and we won't be working with it too much in this class, but you should know it's out there.)

Suppose I wanted to get some information from Courtlistener—say, I wanted to know the date that their database of Supreme Court opinions started. Here's what I'd do.

1. First, I'd sign up for an API key on the Courtlistener website. It's free for individual use, but you have to actually sign up, they don't just let you access things willy-nilly
2. The documentation for the Courtlistener API gives instructions for how to authenticate with API keys—can you see where? So I'd want to create some headers of the appropriate format, which Requests lets you do pretty easily. Let's suppose my API key were "12345." Here's what I'd do:

```
import requests
my_headers = {"Authorization": "Token 12345"}
```

Then I'd make a request to the endpoint that has the basic information about the Supreme Court data they have. As it turns out, that is https://www.courtlistener.com/api/rest/v3/courts/scotus/

So my next few lines of code would be:

```
scotusresponse = requests.get("https://www.courtlistener.com/api/rest/v3/courts/scotus/", he
print(scotusresponse.json()["start_date"])
```

And that would (assuming my untested code is correct) give me the answer I wanted. Try it yourself!

JSON format

You'll notice that `scotusresponse.json` call in the code above. What's that?

JSON is a plain-text data format that is very popular on the internet. If you look at a JSON file, it looks like a bunch of nested Python lists and dictionaries—and that's because that's basically what it is. Actually, it's almost identical to the syntax for the JavaScript equivalent of lists and dictionaries, because it's derived from JavaScript (JSON stands for JavaScript Object Notation). For the most part, if you want to write Python lists and dictionaries to disk, you'll want to use JSON format.

There's a built-in Python library called, unsurprisingly, "json" that has has functionality to read and write JSON files from disk or from strings in memory. But if you receive data from the network in JSON format using Requests, you can call the `.json()` method on the response that you receive, and it'll kindly parse the JSON for you. ("Parse" means turn it from its representation as a JSON-formatted string into ordinary Python lists and dictionaries.)

So, when you call the courts endpoint, Courtlistener sends back data in JSON format by default. So calling the `.json()` method will turn that into a Python dict, and then, by looking at the documentation, we can see that the `start_date` key is the right one to look for in that dict to get the information we need—and there it is!

That's a lot to absorb right now—don't worry, we'll get some more practice with fetching data using code in class.

Further information

- The most comprehensive and reliable source of information on all things HTTP is the Mozilla Developer's Network (MDN); further details on most of the information here can be found on their page on HTTP.
- To learn more about Python's built-in JSON library, you can use the PYMOTW page on the subject.