Lab 07: Dealing with Data

Contents

Objectives

By the end of this lab you will understand:

- How data is stored on the Internet
- How to parse data stored in various formats

By the end of this lab you will be able to:

- Use Python modules to interact with web APIs
- Use Python modules to interact with different file formats

1 Web Scraping

1.1 urllib

In Python, one way to get the data stored at a website is using the built-in `urllib` module (in Python 3 this has been split into three separate parts). In particular, you can get and store the contents of a website at a URL `url` (where this is stored as a string) by running:

```
from urllib import urlopen

contents = urlopen(url).read()
```

Task 1. Test that this works on your computer by running the above code, either in a file or in the interactive interpreter, using `https://raw.githubusercontent.com/smeiklej/secu2002_2017/master/data/hello_world.txt`. Print the value of `contents` to make sure it's what you expect.

1.2 requests

While `urllib` and its successor `urllib2` provide the functionality we need, they are notoriously difficult to work with, and easily broken. A more popular way[1] to get the data stored at websites is instead by using the `requests` module. This module is not built-in (meaning it does not come with the basic Python distribution), so you'll likely need to install it yourselves. To do so, open a command-line interface and type in:

```
> pip install requests
```

This should now allow you to import the module just like you have installed others. (If you're curious, `pip` is what is known as a *package manager* for keeping track of and installing additional modules and packages.) If it doesn't work, you can also install this module using PyCharm by going to Preferences > Project: lab07 > Project Interpreter (or whatever your project name is if not lab07), pressing the + icon, and then searching for and installing `requests` from there. If neither of these options work, please call one of us over to help.

To get the contents of a website at a URL `url` using the `requests` module, you can run:

```
import requests

r = requests.get(url)
```

This variable `r` is keeping track of a number of different values. For example, `r.status_code` gives the status of the web request, where 200 means the website was successfully accessed and its contents extracted. If this has gone well, the contents can be found in `r.text`.

Task 2. Again, test that this works on your computer by running the above code, either in a file or in the interactive interpreter, using `https://raw.githubusercontent.com/smeiklej/secu2002_2017/master/data/hello_world.txt`. Print the value of `r.text` to make sure it's what you expect.

1.3 APIs

For a general website, the text returned when we get the data stored at the URL will be HTML code (to see this, feel free to try the above tasks with the URL of a website you visit regularly). Extracting information from this in a process known as *scraping* can range from fairly easy (if you're familiar with HTML and the website is static) to extremely difficult (especially for modern websites with many dynamic features). Either way, we won't be covering it here.

Luckily, we often don't need to do this kind of scraping, as many big websites provide a helpful *API*, which is short for Application Programming Interface.[2] Using these interfaces, we can engage in a wide variety of behaviors. For example, you could send a message or create a post on social media (using the APIs available for Instagram, Facebook, Twitter, etc.), get up-to-date weather for your local area, perform live tube tracking by using the API available from TfL, or draw on the artificial intelligence (AI) of IBM Watson. Some of these APIs are publicly available, some require you to register, and some even require you to pay. Many also implement what is called *rate limiting*, meaning you can access them only a certain number of times within some defined time period. A fairly extensive list of publicly available APIs (where registration may be required) can be found at `https://github.com/toddmotto/public-apis`.

For now, we'll focus on extracting relatively simple data from publicly available APIs. For big services, these APIs are often well documented; e.g., the Github API documentation can be found at `https://developer.github.com/v3/`, and you can find all of the commits made to the repository for this module at `https://api.github.com/repos/smeiklej/secu2002_2017/commits`. As another example, you can find the weather forecast for the upcoming week in London at `https://www.metaweather.com/api/location/44418/`, or for a specific date YYYY/MM/DD in the past using `https://www.metaweather.com/api/location/44418/YYYY/MM/DD`.

[1] `https://stackoverflow.com/questions/2018026/what-are-the-differences-between-the-urllib-urllib2-and-requests-module`
[2] `https://en.wikipedia.org/wiki/Application_programming_interface`

2 File Formats

When we interact with a web API, we normally get data back in one of a small number of common formats. Before we can start using the data, we need to *parse* it into a format we can use, which means we need to know something about its structure. For example, if you clicked on the links given above, you would have seen values returned in what is known as *JSON* format. Before discussing this, we first discuss another common file format, which is CSV.

2.1 csv

We have already seen CSV files several times in previous labs, but so far the way we have interacted with them is fairly *manual*; i.e., we have parsed them ourselves by either manually identifying exactly where in a row a value we're interested in sits (like `line[48:]`), or by manually splitting the row into its columns using the `split` function.

The `csv` module makes the process of interacting with CSV files easier and more flexible. There is an extensive list of features for this module, but here we focus on only two: `csv.reader` and `csv.writer`, and `csv.DictReader` and `csv.DictWriter`. (Below we describe the ways to read in files, but the process can always be reversed in order to write, or you can look up the appropriate functions in the Python documentation.)

The first one, `csv.reader`, essentially runs the code for us to split the file according to its delimiter. Again, this is important to keep the code flexible in the face of changes to the underlying data. For example, if you update the master repository you'll notice that the file `church_metal_theft.csv` now uses `|` as the delimiter between columns rather than `:::`. This means any code using "hard-coded" constants to index into the data (like `line[48:]`) won't work anymore.

To read every row in this file, we can run:

```
import csv

f = open('church_metal_theft.csv', 'r')
r = csv.reader(f, delimiter='|')
for row in r:
    print row
```

If you run this yourself, you'll see that `row` is a list where each item is a column in the CSV file.

As the name suggests, the pair `csv.DictReader` and `csv.DictWriter` has the same functionality as if we used dictionaries to map the column's meaning to its value, as is done in the solutions provided for Labs 04 and 05. This works only if the CSV file has a *header* signalling what the columns are, which the file `church_metal_theft.csv` now does. To print out the start date for every crime, we could now run:

```
import csv

f = open('church_metal_theft.csv', 'r')
r = csv.DictReader(f, delimiter='|')
for row in r:
    print row['start date']
```

Task 3. Copy the solution provided for `church_theft.py` from Lab 05 into your current directory. That solution defines a function `create_row` that takes in a row and outputs a dictionary mapping the column's meaning to its value in that row. Now that the CSV file has a header, use `csv.DictReader` to replace the use of this function in creating the variable `spreadsheet`. (You can then delete all the code that comes afterwards.)

2.2 json

Beyond CSV, another commonly used file format for storing associated data is JSON (especially on the Internet, as we saw with the APIs discussed above). This is often used to store lists of data entries, in which each entry might itself have a number of entries.

To parse a JSON file stored in a value `f`, you can run:

```
import json
text = json.loads(f)
```

For example, to get the commits to the repository for this module you could run:

```
import json
import requests
r = requests.get('https://api.github.com/repos/smeiklej/secu2002_2017/commits')
text = json.loads(r.text)
```

To make things even easier, the `requests` module has support for JSON built in, so you could in fact achieve the same functionality by running

```
import requests
r = requests.get('https://api.github.com/repos/smeiklej/secu2002_2017/commits')
text = r.json()
```

Task 4. Using the URL for the Github API provided above, write code in a file `github.py` to print out the commit messages for every commit to the master repository. Here you might find `map` useful, and the fact that the commit message is stored at `x['commit']['message']` for every entry `x` in the JSON file. Feel free to run this for your own repository as well, by replacing my username and repository name with your own.

3 Pickling

When we are working with data internally, it can be nice to keep it intact, as we have already gone to the trouble to parse it into a particular data structure or format. For example, if we have a lot of data stored in a Python dictionary and need to close a file, or update the values in the dictionary many times, we would probably rather keep the data in the dictionary, as opposed to writing it to an external data file and then re-parsing it every time we use it again.

Luckily, Python has a built-in `pickle` module that allows us to keep its data structures intact when saving to a file. For example, if we have values stored in a dictionary `mydict`, we could run

```
import pickle

fname = 'data/pickles/mydict.p'
pickle.dump(mydict, open(fname, 'wb'))
```

This stores the dictionary, intact, in the specified file. To load the dictionary when we need it again, we could run

```
fname = 'data/pickles/mydict.p'
mydict = pickle.load(open(fname,'rb'))
```

If you're curious, the `'rb'` and `'wb'` flags signal to the `open` function that this is a *binary* file, meaning it has some encoding beyond plain text.

4 Putting It All Together

A growing concern about cryptocurrencies like Bitcoin is the ability to use them to launder money, and to provide financing for illegal operations such as human trafficking, terrorism, and the sale of drugs. This is due largely to the fact that such cryptocurrencies operate online without going through any intermediary, and also because Bitcoin users identify themselves solely using pseudonyms (i.e., identifiers that have no association with their real-world identity). Even though this doesn't provide the level of anonymity users might expect [1], it still presents an obstacle for law enforcement agencies looking to investigate criminal activity in Bitcoin.

To make it even easier to hide flows of money, there are now many thousands of other cryptocurrencies,[3] and several services for moving money back and forth between them. To interact

[3] `https://coinmarketcap.com/`

with these services, a user with coins held in one cryptocurrency (e.g., Bitcoin) sends those coins to the service and also indicates which cryptocurrency they want to change them into (e.g., Ethereum, the cryptocurrency with the second highest market capitalization). The service then performs the exchange and sends the user the right number of coins (according to the exchange rate) in the currency they've requested.

In this lab, we will focus on one of these services, ShapeShift (`https://shapeshift.io`), as it has a simple and publicly available API. In particular, the API allows people to automatically perform exchanges without needing to create an account or register, which would seem to make it attractive to criminals wishing to hide their identity. On the other hand, it also has features that allow us as researchers to access data about recent transactions. The documentation for their entire API is at `https://info.shapeshift.io/api`. For this lab we'll focus only on the list of recent transactions, which is described at `https://info.shapeshift.io/api#api-4`.

As described in the documentation, you can access a JSON list of recent ShapeShift transactions at `https://shapeshift.io/recenttx/[max]`, where `[max]` is a number between 1 and 50. Today we will scrape this data and compute basic statistics about these transactions, but this is currently an active area of my research so I'd be happy to discuss further opportunities if you're interested.

The data is stored in JSON format, and each entry itself contains five entries: (1) `curIn`, which is the input cryptocurrency (i.e., the currency sent to ShapeShift by the user); (2) `curOut`, which is the output cryptocurrency (i.e., the currency the user wants to change into); (3) `timestamp`, which is the time of the transaction specified in Unix time;[4] (4) `amount`, which is the amount of the input currency sent to the service; and (5) `txid`, which is a unique identifier for this transaction.

Task 5. In a file `scrape_shapeshift.py`, access the 50 most recent ShapeShift transactions at the URL specified above, and parse them into a variable. Save this static snapshot in a file `shapeshift.p`.

Now, create a separate file `analyze_shapeshift.py` to analyze the data you've collected. In particular, retrieve the entries from `shapeshift.p` and print out the following three statistics:

1. How many transactions have Ethereum (code `ETH`) as the input currency.
2. How many have Bitcoin (code `BTC`) as the output currency.
3. How many transactions are trading Zcash (code `ZEC`) to Bitcoin.

For an extra point (but really more just for you to practice), define a function `add_to_dict` that takes in three inputs: a dictionary `d`, a key `k`, and a numeric value `v`, and returns an updated dictionary. The function should either add the value to the existing one if there is already a mapping for that key, or add the mapping if it doesn't. Using this function, populate two dictionaries, `dict_in` and `dict_out`. The first dictionary should map input currencies to the total amount paid in that currency, and the second should map output currencies to the number of times they were used. Using these dictionaries, print out the currency that had the most number of coins paid in (and the amount), and the currency that was used as output the most number of times (and the number of times).

5 Handing In

As always, feel free to continue trying out the various concepts we've covered in this lab. Once you are done, add the files into a `lab07` directory; this should mean adding:

- `github.py`
- `church_theft.py`
- `scrape_shapeshift.py`
- `analyze_shapeshift.py`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.

[4] `https://en.wikipedia.org/wiki/Unix_time`

References

[1] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage, A fistful of bitcoins: characterizing payments among men with no names. *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2013.