Web Scraping and RESTful APIs

Jamie Saxon

Introduction to Programming for Public Policy

November 2, 2016

Getting Data

- Wonderful but rare, to find free, formatted data.
- How do we use the data that's available, but 'un-processed'?
- Look for 'undocumented APIs,' scrape when necessary, and hope for pretty interfaces.

Examples of Different Resources

Qualities and techniques best demonstrated by example:

1. Illinois State Board of Education **Report Card** provides a lot of data, but back-breaking methods for accessing it.
2. Census provides both a 'consumer-level' **site** and **an API** for retrieving well-formatted data at any level.
3. Bureau of Labor Statistics similarly provides a **respectable API** for time series data whose **coding** is, nevertheless, 'somewhat' abstruse.
   - The ATUS that you've used exists only as bulk text files.
4. Weather Underground provides an expensive API for predictions, but quietly **exposes** hourly data for any airport in the US.
5. Twitter famously provides one of the most-comfortable, fast, featureful, compulsively perfect **APIs** ever.

Consider the people who want to use your (their!) data!

A Spectrum

- Unmanageable, awful websites...
- 'Trivially' extractable data.
- Fabulous APIs, ready for consumption.

Three Tools

1. `curl`: request simple web resources, issue post requests, get options.
2. **requests** library: retrieve web resources in python.
   - Provides methods for authentication, `POST`ing data, etc.
   - Basically, `curl` for python.
3. **beautifulsoup** library: provides mechanisms for 'quickly' accessing and extracting elements of web pages in python.

curl: Weather Data

- Useful new expression: `for` loops and `$variables` in bash:

```
addr="https://www.wunderground.com/history/airport/KMDW"
for x in $(seq 1 31); do
  curl ${addr}/2016/10/${x}/DailyHistory.html?format=1
done
```

- Bulk download (and format!) data from available end-points.

Requests: Minimal Example

- Just request an address – here, **Census**.

```
import requests

addr =  "http://api.census.gov/data/2014/acs5/profile?"
addr += "for=tract:*&in=state:42+county:*"
addr += "&get=NAME,DP02_0058E,DP02_0058M"

resp = requests.get(addr) # this is it!!
# pt = requests.put('addr', data = {'key':'value'})
# ... and options, delete, etc.

s = resp.status_code
t = resp.text
c = resp.content
j = resp.json()
```

Beautiful Soup

- (Real scraping is basically the worst.)
- 'Art' to scanning the raw webpage and finding the tag you want.
- Take an example of the **Illinois State Report Card** (which we used a few weeks back): 560 schools in the district, not doing it by hand.
  - Check the html to see if you can find it? No!?
  - Chrome: View → Developper → Developper Tools → Elements.
  - Firefox: Tools → Developper → Toggle Tools → Inspector.
  - Find the element of interest... find its source.
- Most complex webpages load content from many different sources; they may not all be rendered as part of the base URL.
- Best case: find it in the webpage or wait for it to load.

Beautiful Soup: Minimal Example

- Please grap the url from this **link**.
- Go to View/Tools → Source, and look for the address.

```
import requests
from bs4 import BeautifulSoup

sch_ad =  "https://iircapps.niu.edu/Apps_2_0/"
sch_ad += "/en/School/150162990250849/Profile?"
sch_ad += "helperUrl=//illinoisreportcard.com/helper.html"

resp = requests.get(sch_ad)
soup = BeautifulSoup(resp.content, "html.parser")

street = soup.find("span", "street").contents[0]
```

- Full documentation **here**.

BS4: Find and Find All

- If you can identify the objects, `find()` and `find_all()` are usually the fastest accessors.
  - These yield the first, and all instances, respectively.
- Consider this example, from our website:

```
addr = "https://harris-ippp.github.io/lectures/"
resp = requests.get(addr)
html = resp.content
soup = bs(html, "html.parser")
```

- Find all the rows in the table:

```
soup.find_all("tr")
```

BS4: Children and Contents

- Consider all of the 'true' conditions of chicken health benefits.
- The problem is that the truth is in a different element from the item.
- We need to to look at rows in their entirety, printing the first column if the second is true.
- `children()` provides an iterable, and `.contents` provides a list.

BS4: Parts of Elements

Accessing parts of elements get/dictionary:

- `soup.find("img").get("src")`
- `soup.find("ul").get("class")`
- `soup.find("em").contents`
- `soup.find("ul")["class"]`
- `soup.find("a")["href"]`

Advanced Scraping (For Information)

- On **many** web pages, you have to wait for the page to 'load.'
- Could do this with browser, then save page... not scalable!
- So use a web driver – ugh!

```
■ conda install --channel \
https://conda.anaconda.org/conda-forge selenium
■ conda install phantomjs
```

```
from selenium import webdriver

driver = webdriver.PhantomJS()
addr = "http://illinoisreportcard.com/District.aspx?"
addr += "source=schoolsindistrict&Districtid=1501629902 5"
driver.get(addr)
driver.page_source
```

RESTful APIs

Representational State Transfer: Doing it the 'Right' Way.

REST in Practice*

- Data on the web could obviously be better 'exposed' and integrated.
- Application Programming Interfaces (APIs) allow/instruct users/developpers on how to use a resource.

APIs: Fancy-Coded Web Addresses

- RESTful web services use (usually) HTTP methods as meaningful verbs, and web addresses as functions.
- `GET` is a pure retrieval and `POST` corresponds to a send.
  - `DELETE`, `PUT`, etc. may also be defined.
- You can then access these resources with `requests`, `curl`, etc.

*The most-readable resource on this I have found is **here**.

RESTful Principles

Roy Fielding standardized good principles for HTTP 1.1 and RESTful services. The philosophy is that:

- Client and server are stateless and separated (server doesn't 'remember' anything about a session).
- Service is scalable and cachable; this allows for expansion.
- But REST is a uniform/consistent style of accessing resources... best illustrated by example.

Twitter API

- Fantastic, clear API: **https://api.twitter.com/1.1/**
- **Many methods**, e.g., show users or tweets by user:

```
/users/show.json?screen_name=iamjohnoliver
/statuses/user_timeline.json?screen_name=iamjohnoliver
```

- Basically: carefully follow each model. Query starts by ? and terms are separated by & (except in `search`).
- Requires access keys, readily generated with a Twitter account.†

†Follow **these instructions** and checkout **this script** if you want to try this.

Census API

- More-typical, but still excellent, API from the **Census**.
- For example, five-year **ACS estimates** as of 2014 (**variables**).

http://api.census.gov/data/2014/acs5/profile?get=DP02_0037PE,NAME&for=state:*

- Returning to week 4:

```
import requests, pandas as pd

addr =  "http://api.census.gov/data/2014/acs5/"
addr += "profile?get=DP02_0037PE,NAME&for=state:*"
j = requests.get(addr).json()

df = pd.DataFrame(j[1:], columns = j[0])
print(df)
```

Two Notes on APIs

1. A lot of APIs end up with 3rd party python libaries.
   - For two examples, these are tweepy and sunlightlabs/census.
   - Both of them are good! But they obfuscate the original intention, and often are not as well documented as the original site.
   - I usually find it easier just to understand the API.

2. Many cities/states use Socrata or CKAN (open source/data.gov). Socrata comes with the SODA API, for many (all?) datasets (e.g., **Chicago Divvy**).

A Soapbox on Standards

- Tremendous range in how hard you have to work to get data.
- Lots of jurisdictions and agencies are touting their open data efforts. But very often, the released data sets are awful.
- Even when they're very good (e.g., city crime, education) they may not be standardized across jurisdictions.
- Need standards (schemas) to minimize overlapping efforts.