Online Data

Lesson 8 with *Ian Carroll*

Lesson Objectives

- Distinguish three sources for online data
- Understand how HTTP and web-services work
- Learn Python idioms for requesting data

Specific Achievements

- Programatically aquire data embedded in a web page
- Request data through a REST API
- Use the census package to aquire data

Why script data aquistion?

- Too time intensive to aquire manually
- Integrate updated or new data
- Reproducibility
- There’s an API between you and the data

Top of Section

Aquiring Online Data

Data can be available on the web in many different forms. The difficulty you will have aquiring that data for running analyses depends on which of three approaches the data source requires.

Scraping 😕

If a web browser can read HTML and JavaScript and display a human readable page, why can’t you right a program (a “bot”) to read HTML and JavaScript and store the data?

Web Service or API 😉

An Application Programming Interface (API, as opposed to GUI) that is compatible with passing data around the internet using HTTP (Hyper-text Transfer Protocol). This is not the fastest protocol for moving large datasets, but it is universal (it underpins web browsers, after all).

API Wrapper 😂

Major data providers can justify writing a package, specific to your language of choice (e.g. Python or R), that facilitates accessing the data they provide through a web service. Sadly … not all do so.

Top of Section

Requests

That “http” at the beginning of the URL for a possible data source is a protocol—an understanding between a client and a server about how to communicate. The client does not have to be a web browser, so long as it knows the protocol. After all, servers exist to serve.

The requests package provides a simple interface to issueing HTTP requests and handling the response.

worksheet-8.ipynb

```
import requests
```

```
response = requests.get('https://xkcd.com/869')
response
```

```
<Response [200]>
```

The response is still binary, it takes a browser-like parser to translate the raw content into an HTML document. BeautifulSoup does a fair job, while making no attempt to “render” a human readable page.

worksheet-8.ipynb

```
from bs4 import BeautifulSoup

doc = BeautifulSoup(response.text, 'lxml')
'\n'.join(doc.prettify().splitlines()[0:10])
```

```
'<!DOCTYPE html>\n<html>\n <head>\n  <link href="/s/b0dcca.css" rel="stylesheet" title="Default" type=
```

Searching the document for desired content is the hard part. This search uses a CSS query, to find the image below a section of the document with attribute `id = comic`.

worksheet-8.ipynb

```
img = doc.select('#comic > img')
img
```

```
[<img alt="Server Attention Span" src="//imgs.xkcd.com/comics/server_attention_span.png" title="They ha
```

It makes sense to query by CSS if the content being scraped always appears the same in a browser; stylesheets are separate from delivered content.

worksheet-8.ipynb

```
img = doc.select_one('#comic > img')
img['title']
```

```
"They have to keep the adjacent rack units empty. Otherwise, half the entries in their /var/log/syslog
```

Was that so bad?

Pages designed for humans are increasingly harder to parse programmatically.

- Servers provide different responses based on client “metadata”
- JavaScript often needs to be executed by the client
- The HTML `<table>` is drifting into obscurity (mostly for the better)

HTML Tables

Sites with easilly accessible html tables nowadays may be specifically geared toward non-human agents. The US Census provides some documentation for their data services in a massive such table:

https://api.census.gov/data/2017/acs/acs5/variables.html

worksheet-8.ipynb

```
import pandas as pd

vars = (
  pd
  .read_html('https://api.census.gov/data/2017/acs/acs5/variables.html')
  .pop()
)
vars.head()
```

```
         Name            Label  ...   Group Unnamed: 8
0      AIANHH        Geography  ...     NaN        NaN
1      AIHHTL        Geography  ...     NaN        NaN
2       AIRES        Geography  ...     NaN        NaN
3        ANRC        Geography  ...     NaN        NaN
4  B00001_001E  Estimate!!Total  ...  B00001        NaN

[5 rows x 9 columns]
```

We can use our data manipulation tools to search this unwieldy documentation for variables of interest

worksheet-8.ipynb

```
idx = (
  vars['Label']
  .str
  .contains('Median household income')
)
vars.loc[idx, ['Name', 'Label']]
```

```
              Name                                              Label
11214   B19013_001E  Estimate!!Median household income in the past ...
11215  B19013A_001E  Estimate!!Median household income in the past ...
11216  B19013B_001E  Estimate!!Median household income in the past ...
11217  B19013C_001E  Estimate!!Median household income in the past ...
11218  B19013D_001E  Estimate!!Median household income in the past ...
11219  B19013E_001E  Estimate!!Median household income in the past ...
11220  B19013F_001E  Estimate!!Median household income in the past ...
11221  B19013G_001E  Estimate!!Median household income in the past ...
11222  B19013H_001E  Estimate!!Median household income in the past ...
11223  B19013I_001E  Estimate!!Median household income in the past ...
11932   B19049_001E  Estimate!!Median household income in the past ...
11933   B19049_002E  Estimate!!Median household income in the past ...
```

Top of Section

Web Services

The US Census Burea provides access to its vast stores of demographic data over the Web via their API at https://api.census.gov.

The **I** in **GUI** is for interface—its the same in **API**, where buttons and drop-down menus are replaced by functions and object attributes.

Instead of interfacing with a user, this kind of **i**nterface is suitable for use in **p**rogramming another software **a**pplication. In the case of the Census, the main component of the application is some relational database management system. There probably are several GUIs designed for humans to query the Census database; the Census API is meant for communication between your program (i.e. script) and their application.

Inspect this URL in your browser.

In a web service, the already universal system for transferring data over the internet, known as HTTP is half of the interface. All you really need is documentation for how to construct the URL in a standards compliant way that the service will recognize.

Section	Description
`https://`	**scheme**
`api.census.gov`	**authority**, or simply domain if there's no user authentication
`/data/2015/acs5`	**path** to a resource within a hierarchy
`?`	beginning of the **query** component of a URL
`get=NAME`	first query parameter
`&`	query parameter separator

Section	Description
`for=county`	second query parameter
`&`	query parameter separator
`in=state:*`	third query parameter
`#`	beginning of the **fragment** component of a URL
`irrelevant`	a document section, it isn't even sent to the server

worksheet-8.ipynb

```python
path = 'https://api.census.gov/data/2017/acs/acs5'
query = {
  'get': 'NAME,B19013_001E',
  'for': 'tract:*',
  'in': 'state:24',
}
response = requests.get(path, params=query)
response
```

```
<Response [200]>
```

Response Header

The response from the API is a bunch of 0s and 1s, but part of the HTTP protocol is to include a “header” with information about how to decode the body of the response.

Most REST APIs return as the “content” either:

1. Javascript Object Notation (JSON)
    - a UTF-8 encoded string of key-value pairs, where values may be lists
    - e.g. `{'a':24, 'b': ['x', 'y', 'z']}`
2. eXtensible Markup Language (XML)
    - a nested `<tag></tag>` hierarchy serving the same purpose

The header from Census says the content type is JSON.

worksheet-8.ipynb

```python
response.headers['Content-Type']
```

```
'application/json;charset=utf-8'
```

Response Content

Use a JSON reader to extract a Python object. To read it into a Panda’s `DataFrame`, use Panda’s `read_json`.

worksheet-8.ipynb

```python
data = pd.read_json(response.content)
data.head()
```

```
                                                0            1  ...       3       4
0                                            NAME  B19013_001E  ...  county   tract
1  Census Tract 105.01, Wicomico County, Maryland        68652  ...     045  010501
2  Census Tract 5010.02, Carroll County, Maryland        75069  ...     013  501002
3  Census Tract 5077.04, Carroll County, Maryland        88306  ...     013  507704
4  Census Tract 5061.02, Carroll County, Maryland        84810  ...     013  506102

[5 rows x 5 columns]
```

API Keys & Limits

Most servers request good behavior, others enforce it.

- Size of single query
- Rate of queries (calls per second, or per day)
- User credentials specified by an API key

From the Census FAQ What Are the Query Limits?:

> *You can include up to 50 variables in a single API query and can make up to 500 queries per IP address per day… Please keep in mind that all queries from a business or organization having multiple employees might employ a proxy service or firewall. This will make all of the users of that business or organization appear to have the same IP address.*

Top of Section

Specialized Packages

The third tier of access to online data is much preferred, if it exists: a dedicated package in your programming language's repository (PyPI or CRAN).

- Additional guidance on query parameters
- Returns data in native formats
- Handles all "encoding" problems

The census package is a user contributed suite of tools that streamline access to the API.

worksheet-8.ipynb

```
from census import Census

key = None
c = Census(key, year=2017)
c.acs5
```

```
<census.core.ACS5Client object at 0x128c91ad0>
```

Compared to using the API directly via the requests package:

Pros

- More concise code, quicker development
- Package documentation (if present) is usually more user friendly than API documentaion.
- May allow seemless update if API changes

Cons

- No guarantee of updates
- Possibly limited in scope

Query the Census ACS5 survey for the variable `B19001_001E` and each entity's `NAME`.

worksheet-8.ipynb

```
variables = ('NAME', 'B19013_001E')
```

The census package converts the JSON string into a Python dictionary. (No need to check headers.)

worksheet-8.ipynb

```
response = c.acs5.state_county_tract(
    variables,
    state_fips='24',
    county_fips=Census.ALL,
    tract=Census.ALL,
)
response[0]
```

```
{'NAME': 'Census Tract 105.01, Wicomico County, Maryland', 'B19013_001E': 68652.0, 'state': '24', 'cour
```

The Pandas `DataFrame()` constructor will accept the list of dictionaries as the sole argument, taking column names from “keys”.

worksheet-8.ipynb

```
df = (
  pd
  .DataFrame(response)
  .query("B19013_001E >= 0")
)
```

The seaborn package provides some nice, quick visualizations.

worksheet-8.ipynb

```
import seaborn as sns

sns.boxplot(
  data = df,
  x = 'county',
  y = 'B19013_001E',
)
```

Top of Section

Paging & Stashing

A common strategy that web service providers take to balance their load, is to limit the number of records a single API request can return. The user ends up having to flip through “pages” with the API, handling the response content at each iteration. Options for stashing data are:

1. Store it all in memory, write to file at the end.
2. Append each response to a file, writing frequently.
3. Offload these decisions to database management software.

To repeat the exercise below at home, request an API key at https://api.data.gov/signup/, and store it in an adjacent `api_key.py` file with the single variable `API_KEY = your many digit key`.

The “data.gov” API provides a case in point. Take a look at the request for comments posted by the US Department of Interior about Bears Ear National Monument. The document received over two million comments, all accessible through Regulations.gov.

Load the `API_KEY` variable by running a file you have saved it in.

worksheet-8.ipynb

```
%run path/to/api/key.py
```

worksheet-8.ipynb

```
import requests

api = 'https://api.data.gov/regulations/v3/'
path = 'document.json'
query = {
    'documentId':'DOI-2017-0002-0001',
    'api_key':API_KEY,
}
doc = (
    requests
    .get(api + path, params=query)
    .json()
)
```

Extract data from the returned JSON object, which gets mapped to a Python dictionary called `doc`.

Console

```
> doc['numItemsRecieved']
```

```
{'label': 'Number of Comments Received', 'value': '2839046'}
```

Initiate a new API query for public submission (PS) comments and print the dictionary keys in the response.

worksheet-8.ipynb

```
query = {
    'dktid':doc['docketId']['value'],
    'dct':'PS',
    'api_key':API_KEY,
}
path = 'documents.json'
dkt = (
     requests
    .get(api + path, params=query)
    .json()
)
```

To inspect the return, we can list the keys in the parsed `dkt`.

Console

```
> list(dkt.keys())
```

```
['documents', 'totalNumRecords']
```

The purported claimed number of results is much larger than the length of the documents array contained in this response.

Console

```
> len(dkt['documents'])
```

```
25
```

The following commands prepare Python to connect to a database-in-a-file, and creates empty tables in the database if they do not already exist (i.e. it is safe to re-run after you have populated the database).

Step 1: Boilerplate

The SQLAlchemy package has a lot of features, and requires you to be very precise about how to get started.

schema.py

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

Step 2: Table Definition

Define the tables that are going to live in the database using Python classes. For each class, its attributes will map to columns in a table.

schema.py

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, Text

Base = declarative_base()

class Comment(Base):
    __tablename__ = 'comment'

    id = Column(Integer, primary_key=True)
    comment = Column(Text)

engine = create_engine('sqlite:///BENM.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

For each document, we’ll just store the “commentText” found in the API response.

Console

```
> doc = dkt['documents'].pop()
+ doc['commentText']
```

```
"Dear Ryan Zinke,\n\nOur national monuments and public lands and waters help define who we are as a nat
```

Step 3: Connect (and Initialize)

worksheet-8.ipynb

```
from schema import Session, Comment

session = Session()
engine = session.bind
```

You could inspect the BENM database now using any sqlite3 client: you would find one empty “comment” table with fields “id” and “comment”.

Add a new `rpp` parameter to request `100` documents per page.

worksheet-8.ipynb

```
query['rpp'] = 10
```

In each request, advance the query parameter `po` to the number of the record you want the response to begin with. Insert the documents (the key:value pairs stored in `values`) in bulk to the database with `engine.execute()`.

worksheet-8.ipynb

```
table = Comment.metadata.tables['comment']
for i in range(0, 15):

    # advance page and query
    query['po'] = i * query['rpp']
    response = requests.get(api + path, params=query)
    page = response.json()
    docs = page['documents']

    # save page with session engine
    values = [{'comment': doc['commentText']} for doc in docs]
    insert = table.insert().values(values)
    engine.execute(insert)
```

```
<sqlalchemy.engine.result.ResultProxy object at 0x128383250>
<sqlalchemy.engine.result.ResultProxy object at 0x12837a790>
<sqlalchemy.engine.result.ResultProxy object at 0x12837ced0>
```

```
<sqlalchemy.engine.result.ResultProxy object at 0x12838b110>
<sqlalchemy.engine.result.ResultProxy object at 0x128386c50>
<sqlalchemy.engine.result.ResultProxy object at 0x128386b10>
<sqlalchemy.engine.result.ResultProxy object at 0x12838b290>
<sqlalchemy.engine.result.ResultProxy object at 0x12837a850>
<sqlalchemy.engine.result.ResultProxy object at 0x128386190>
<sqlalchemy.engine.result.ResultProxy object at 0x128390750>
<sqlalchemy.engine.result.ResultProxy object at 0x128396610>
<sqlalchemy.engine.result.ResultProxy object at 0x128394bd0>
```

View the records in the database by reading everyting we have so far back into a `DataFrame`.

worksheet-8.ipynb

```
df = pd.read_sql_table('comment', engine)
```

Don't forget to disconnect from your database!

worksheet-8.ipynb

```
engine.dispose()
```

Top of Section

Takeaway

- Web scraping is hard and unreliable, but sometimes there is no other option.
- Web services are the most common resource.
- Search PyPI for an API you plan to use.

Web services do not always have great documentation—what parameters are acceptable or necessary may not be clear. Some may even be poorly documented on purpose, if the API wasn't designed for public use. Even if you plan to aquire data using the "raw" web service, try a search for a relevant package on Python. The package documention could help.

Top of Section

Exercises

Exercise 1

Identify the name of the census variable in the table of ACS variables whose label includes "COUNT OF THE POPULATION". Next use the Census API to collect the data for this variable, for every county in the U.S. state with FIPS code '24', into a pandas DataFrame.

Exercise 2

Request an API key for Regulations.gov or find one you have permission to access. Use the API to collect 3 "pages" of comments posted on the "Revised Definition of 'Waters of the United States'". Over half a million were received before the comment period closed on April 15^th^, 2019. Modify `schema.py` to save the comments into a new SQLite file.

Top of Section

If you need to catch-up before a section of code will work, just squish it's 🍅 to copy code above it into your clipboard. Then paste into your interpreter's console, run, and you'll be ready to start in on that section. Code copied by both 🍅 and 📋 will also appear below, where you can edit first, and then copy, paste, and run again.

```
# Nothing here yet!
```