# Announcements

New classroom

- Not yet official
- Will announce
- Still in GHC
- 6 more seats

Waitlist

388/688 Sections

- Undergrads stay in 388, grads stay in 688 (no switching)

# Announcements

HW1

- Plan out tomorrow night
- Plan due Tue 2/8

Grading infrastructure

- https://mugrade.datasciencecourse.org/
- Create new account with <andrewid>@andrew.cmu.edu
- More instructions in HW1

## Plan

- Wrap up intro slides
- Data collection and scraping

# 15-388/688 - Practical Data Science: Data collection and scraping

Pat Virtue
Carnegie Mellon University
Spring 2022

Slide credits: CMU AI, Zico Kolter

## Outline

The data collection process

Common data formats and handling

Regular expressions and parsing

## Outline

The data collection process

Common data formats and handling

Regular expressions and parsing

## The first step of data science

The first step in data science …

... is to get some data

You will typically get data in one of four ways:

1. Directly download a data file (or files) manually – not much to say
2. Query data from a database – to be covered in later lecture
3. Query an API (usually web-based, these days)
4. Scrap data from a webpage

covered today

## Issuing HTTP queries

The vast majority of automated data queries you will run will use HTTP requests (it's become the dominant protocol for much more than just querying web pages)

I know we promised to teach you know things work under the hood … but we are *not* going to make you implement an HTTP client

Do this instead (requests library, http://docs.python-requests.org/ ):

```python
import requests
response = requests.get("http://www.datasciencecourse.org")

# some relevant fields
response.status_code
response.content # or response.text
response.headers
response.headers['Content-Type']
```

# HTTP Request Basics

You've seen URLs like these:

`https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&cad=rja&uact=8…`

The weird statements after the url are *parameters,* you would provide them using the requests library like this:

```
params = {"sa":"t", "rct":"j", "q":"", "esrc":"s",
          "source":"web", "cd":"9", "cad":"rja", "uact":"8"}
response = requests.get("http://www.google.com/url", params=params)
```

HTTP GET is the most common method, but there are also PUT, POST, DELETE methods that *change* some state on the server

```
response = requests.put(...)
response = requests.post(...)
response = requests.delete(...)
```

## RESTful APIs

If you move beyond just querying web pages to web APIs, you'll most likely encounter REST APIs (Representational State Transfer)

REST is more a design architecture, but a few key points:

1. Uses standard HTTP interface and methods (GET, PUT, POST, DELETE)
2. Stateless – the server doesn't remember what you were doing

Rule of thumb: if you're sending the your account key along with each API call, you're probably using a REST API

## Querying a RESTful API

You query a REST API similar to standard HTTP requests, but will almost always need to include parameters

```
token = "" # not going to tell you mine
headers = {'Authorization': 'token '+token}
response = requests.get("https://api.github.com/user", headers=headers)
print(response.content)
#{"login":"pvirtue","id":5945661,"avatar_url":"https://avatars.githubu…
```

Get your own access token at https://github.com/settings/tokens/new

GitHub API uses GET/PUT/DELETE to let you query or update elements in your GitHub account automatically

Example of REST: server doesn’t remember your last queries, for instance you always need to include your access token if using it this way

## Authentication

Basic authentication has traditionally been the most common approach to access control for web pages

```
# this won't work anymore
response = requests.get("https://api.github.com/user",
                        auth=('pvirtue', 'passwd'))
```

Most APIs have replaced this with some form of Oauth or token-based authorization

## Outline

## Data formats

The three most common formats (judging by my completely subjective experience):

1. CSV (comma separate value) files

2. JSON (Javascript object notation) files and strings

3. HTML/XML (hypertext markup language / extensible markup language) files and strings

# CSV Files

Refers to any delimited text file (not always separated by commas)

```
"Semester","Course","Section","Lecture","Mini","Last Name","Preferred/First
Name","MI","Andrew ID","Email","College","Department","Class","Units","Grade
Option","QPA Scale","Mid-Semester Grade","Final Grade","Default Grade","Added
By","Added On","Confirmed","Waitlist Position","Waitlist Rank","Waitlisted
By","Waitlisted On","Dropped By","Dropped On","Roster As Of Date”
"F16","15688","B","Y","N",”Kolter","Zico","","zkolter","zkolter@andrew.cmu.edu","S
CS","CS","50","12.0","L","4+"," "," ","","reg","1 Jun
2016","Y","","","","","","","30 Aug 2016 4:34"
```

If values themselves contain commas, you can enclose them in quotes (our registrar apparently always does this, just to be safe)

```python
import pandas as pd
dataframe = pd.read_csv("CourseRoster_F16_15688_B_08.30.2016.csv",
                        delimiter=',', quotechar='"')
```

We’ll talk about the pandas library a lot more in later lectures

# JSON files / string

JSON originated as a way of encapsulating Javascript objects

A number of different data types can be represented

- Number: `1.0` (always assumed to be floating point)
- String: `"string"`
- Boolean: `true` or `false`
- List (Array): `[item1, item2, item3,…]`
- Dictionary (Object in Javascript): `{"key":value}`

Lists and Dictionaries can be embedded within each other:

```
[{"key":[value1, [value2, value3]]}]
```

## Example JSON data

JSON from Github API

```
{
 'login': 'pvirtue',
 'id': 5945661,
 'node_id': 'MDQ6VXNlcjU5NDU2NjE=',
 'avatar_url': 'https://avatars.githubusercontent.com/u/5945661?v=4',
 'gravatar_id': '',
 'url': 'https://api.github.com/users/pvirtue',
 'html_url': 'https://github.com/pvirtue',
 'followers_url': 'https://api.github.com/users/pvirtue/followers',
 'following_url': 'https://api.github.com/users/pvirtue/following{/other_user}',
 'gists_url': 'https://api.github.com/users/pvirtue/gists{/gist_id}',
 'starred_url': 'https://api.github.com/users/pvirtue/starred{/owner}{/repo}',
 'subscriptions_url': 'https://api.github.com/users/pvirtue/subscriptions',
 'organizations_url': 'https://api.github.com/users/pvirtue/orgs',
 'repos_url': 'https://api.github.com/users/pvirtue/repos',
 'events_url': 'https://api.github.com/users/pvirtue/events{/privacy}',
 'received_events_url': 'https://api.github.com/users/pvirtue/received_events',
 'type': 'User',
 'site_admin': False,
 'name': 'Pat Virtue', ...
```

## Parsing JSON in Python

Built-in library to read/write Python objects from/to JSON files

```
import json

# load json from a REST API call
headers = {'Authorization': 'token '+token}
response = requests.get("https://api.github.com/user", headers=headers)
data = json.loads(response.content)

json.load(file) # load json from file
json.dumps(obj) # return json string
json.dump(obj, file) # write json to file
```

# XML / HTML files

The main format for the web (though XML seems to be loosing a bit of popularity to JSON for use in APIs / file formats)

XML files contain hiearchical content delineated by tags

```
<tag attribute="value">
    <subtag>
        Some content for the subtag
    </subtag>
    <openclosetag attribute="value2”/>
</tag>
```

HTML is syntactically like XML but horrible (e.g., open tags are not always closed), more fundamentally, HTML is mean to describe appearance

# Parsing XML/HTML in Python

There are a number of XML/HTML parsers for Python, but a nice one for data science is the BeautifulSoup library (specifically focused on getting data out of XML/HTML files)

```python
# get all the links within the data science course schedule
from bs4 import BeautifulSoup
import requests
response = requests.get("http://www.datasciencecourse.org/lectures")

root = BeautifulSoup(response.content)
root.find("table").find("tbody").findAll("a")
```

You'll play some with BeautifulSoup in the first homework

# Outline

The data collection process

Common data formats and handling

Regular expressions and parsing

## Regular expressions

Once you have loaded data (or if you need to build a parser to load some other data format), you will often need to search for specific elements within the data

E.g., find the first occurrence of the string “data science”

```
import re
text = "This course will introduce the basics of data science"
match = re.search(r"data science", text)
print(match.start())
# 41
```

# Regular expressions in Python

A few common methods to call regular expressions in Python:

```
match = re.match(r"data science", text) # check if start of text matches
match = re.search(r"data science", text) # find first match or None
for match in re.finditer("data science", text):
    # iterate over all matches in the text
    ...
all_matches = re.findall(r"data science", text) # return all matches
```

You can also use “compiled” version of regular expressions

```
regex = re.compile(r"data science")
regex.match(text, [startpos, [endpos]])
regex.search(...)
regex.finditer(...)
regex.findall(...)
```

# Matching multiple potential characters

The real power of regular expressions comes in the ability to match multiple possible sequence of characters

Special characters in regular expressions: .^$*+?{}\[]|() (if you want to match these characters exactly, you need to escape them: \$)

Match sets of characters:

- Match the character ‘a’: a
- Match the character ‘a’, ‘b’, or ‘c’: [abc]
- Many any character except ‘a’, ‘b’, or ‘c’: [^abc]
- Match any digit: \d (= [0-9])
- Match any alpha-numeric: \w (= [a-zA-z0-9_])
- Match whitespace: \s (= [ \t\n\r\f\v])
- Match any character: . (including newline with re.DOTALL)

## Matching repeated characters

Can match one or more instances of a character (or set of characters)

Some common modifiers:

- Match character ‘a’ exactly once: a
- Match character ‘a’ zero or one time: a?
- Match character ‘a’ zero or more times: a*
- Match character ‘a’ one or more times: a+
- Match character ‘a’ exactly n times: a{n}

Can combine these with multiple character matching:

- Match all instances of “<something> science” where <something> is an alphanumeric string with at least one character
- \w+\s+science

# Poll 1: regular expressions

Which strings would be matched (i.e, calling re.match()) by the regular expression?

```
\w+\s+science
```

Select ALL that apply:

A. “life science”

B. “life sciences”

C. “life. Science”

D. “this data science problem”

## Grouping

We often want to obtain more information that just whether we found a match or not (for instance, we may want to know what text matched)

**Grouping:** enclose portions of the regular expression in parentheses to "remember" these portions of the match

`(\w+)\s([Ss]cience)`

```
match = re.search(r"(\w+)\s([Ss]cience)", text)
print(match.start(), match.groups())
# 41 ('data', 'science')
```

Why the 'r' before the string?  Avoids need to double escape strings

## Substitutions

Regular expressions provide a power mechanism for replacing some text with other text

```
better_text = re.sub(r"data science", r"schmada science", text)
```

To include text that was remembered in the matching using groups, use the escaped sequences \1, \2, … in the substitution text

```
better_text = re.sub(r"(\w+)\s([Ss])cience", r"\1 \2hmience", text)
```

(You can also use backreferences within a single regular expression)

# Ordering and greedy matching

There is an order of operations in regular expressions

- `abc|def` matches the strings "abc" or "def", not "ab(c or d)ef"
- You can get around this using parenthesis e.g. `a(bc|de)f`
- This also creates a group, use `a(?:bc|de)f` if you don't want to capture it

By default, regular expressions try to capture as much text as possible (greedy matching)

- `<(.*)>` applied to `<a>text</a>` will match the entire expression
- If you want to capture the *least* amount of text possible use `<(.*?)>` this will just match the `<a>` term

## Additional features

We left out a lot of elements here to keep this brief: start/end lines, lookaheads, named groups, etc

Don’t worry if you can’t remember all this notation (I had to look some things up while preparing this lecture too)

Use the docs: https://docs.python.org/3/howto/regex.html, https://docs.python.org/3/library/re.html

Try out test expressions to see what happens