INTRODUCTION TO DATA SCIENCE

JOHN P DICKERSON

Lecture #3 – 09/03/2019

CMSC320
Tuesdays & Thursdays
5:00pm – 6:15pm

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

ANNOUNCEMENTS

Register on Piazza: piazza.com/umd/fall2019/cmsc320

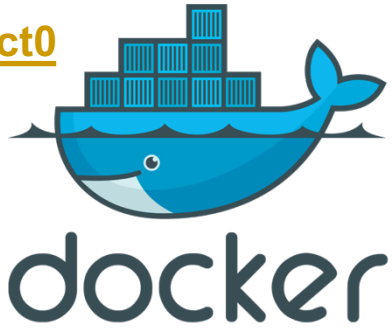
- 267 have registered already ♡
- 31 have not registered yet 💔

If you were on Piazza, you'd know …

- Project 0 is out! It is "due" this Wednesday evening.
- Link: https://github.com/cmsc320/fall2019/tree/master/project0

We've also linked some reading for the week!

- First quiz is due Thursday at noon; on ELMS now.



ANNOUNCEMENTS

Office hours are posted on the course webpage:

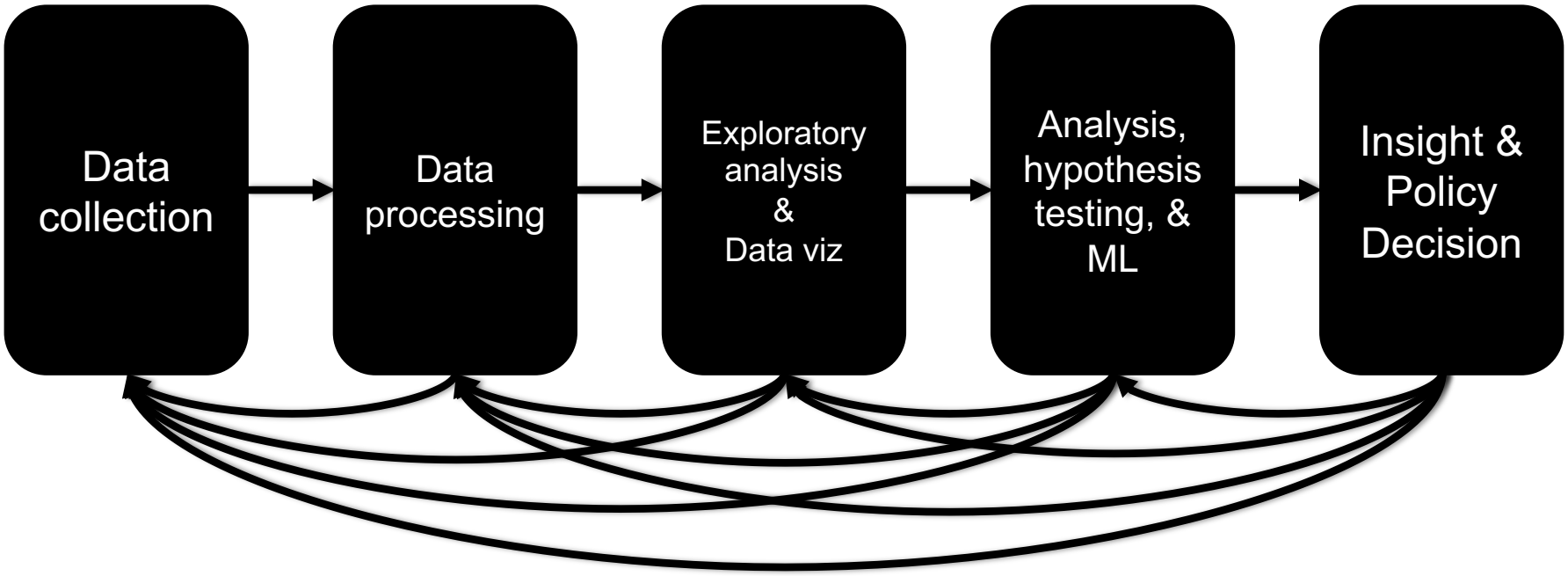
- https://cmsc320.github.io/
- Subject to change; I will update the course webpage if so!

Office hours are held in AVW 1120. (AVW?! Yuck.)

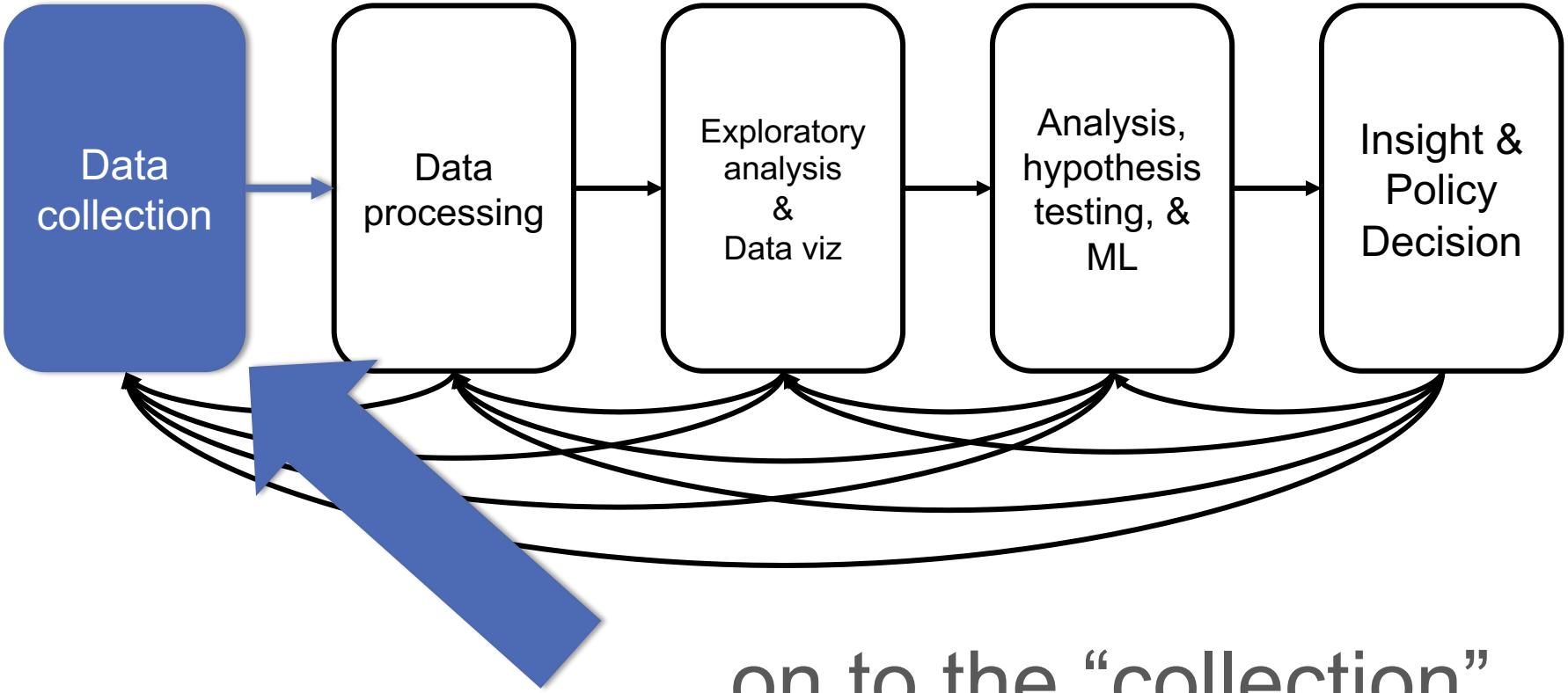
We have coverage before noon and after noon for every weekday (MTWThF).

- TAs will also "cover" Piazza for the working hours of the day on which they are holding office hours.

THE DATA LIFECYCLE



TODAY'S LECTURE



… on to the "collection" part of things …

GOTTA CATCH ‘EM ALL

Five ways to get data:

- **Direct download and load from local storage**
- **Generate locally via downloaded code (e.g., simulation)**
- **Query data from a database (covered in a few lectures)**
- **Query an API from the intra/internet**
- **Scrape data from a webpage**

} Covered today.

WHEREFORE ART THOU, API?

A web-based Application Programming Interface (API) like we'll be using in this class is a contract between a server and a user stating:

> **"If you send me a specific request, I will return some information in a structured and documented format."**

(More generally, APIs can also perform actions, may not be web-based, be a set of protocols for communicating between processes, between an application and an OS, etc.)

“SEND ME A SPECIFIC REQUEST”

Most web API queries we’ll be doing will use HTTP requests:

- `conda install –c anaconda requests=2.12.4`

```
r = requests.get(    'https://api.github.com/user',
                    auth=('user', 'pass')    )
```

```
r.status_code
```

```
200
```

```
r.headers[‘content-type’]
```

```
‘application/json; charset=utf8’
```

```
r.json()
```

```
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

http://docs.python-requests.org/en/master/

HTTP REQUESTS

https://www.google.com/**?q=cmsc320&tbs=qdr:m**

??????????

HTTP GET Request:

GET /?q=cmsc320&tbs=qdr:m HTTP/1.1
Host: www.google.com
User-Agent: **Mozilla/5.0 (X11; Linux x86_64; rv:10.0.1) Gecko/20100101 Firefox/10.0.1**

```
params = { “q”: “cmsc320”, “tbs”: “qdr:m” }
r = requests.get(    “https://www.google.com”,
                     params = params )
```

*be careful with https:// calls; `requests` will not verify SSL by default

RESTFUL APIS

This class will just query web APIs, but full web APIs typically allow more.

Representational State Transfer (RESTful) APIs:

- GET: perform query, return data
- POST: create a new entry or object
- PUT: update an existing entry or object
- DELETE: delete an existing entry or object

Can be more intricate, but verbs ("put") align with actions

QUERYING A RESTFUL API

Stateless: with every request, you send along a token/authentication of who you are

```
token = "super_secret_token"
r = requests.get("https://github.com/user",
                 params={"access_token": token})
print( r.content )
```

```
{"login":"JohnDickerson","id":472985,"avatar_url":"ht…
```

GitHub is more than a GETHub:

- PUT/POST/DELETE can edit your repositories, etc.
- Try it out: https://github.com/settings/tokens/new

AUTHENTICATION AND OAUTH

Old and busted:

```
r = requests.get("https://api.github.com/user",
                 auth=("JohnDickerson", "ILoveKittens"))
```

New hotness:

- What if I wanted to grant an app access to, e.g., my Facebook account without giving that app my password?
- OAuth: grants access tokens that give (possibly incomplete) access to a user or app without exposing a password

"... I WILL RETURN INFORMATION IN A STRUCTURED FORMAT."

So we've queried a server using a well-formed GET request via the `requests` Python module. What comes back?

General structured data:

- Comma-Separated Value (CSV) files & strings
- Javascript Object Notation (JSON) files & strings
- HTML, XHTML, XML files & strings

Domain-specific structured data:

- Shapefiles: geospatial vector data (OpenStreetMap)
- RVT files: architectural planning (Autodesk Revit)
- You can make up your own! Always document it.

GRAPHQL?

An alternative to REST and ad-hoc webservice architectures

- Developed internally by Facebook and released publicly

Unlike REST, the requester specifies the format of the response

```
GET /books/1

{
  "title": "Black Hole Blues",
  "author": {
    "firstName": "Janna",
    "lastName": "Levin"
  }
  // ... more fields here
}
```

```
GET /graphql?query={ book(id: "1") { title, author { firstName } } }

{
  "title": "Black Hole Blues",
  "author": {
    "firstName": "Janna",
  }
}
```

https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b

CSV FILES IN PYTHON

Any CSV reader worth anything can parse files with any delimiter, not just a comma (e.g., “TSV” for tab-separated)

```
1,26-Jan,Introduction,—,"pdf, pptx",Dickerson,
2,31-Jan,Scraping Data with Python,Anaconda's Test Drive.,,Dickerson,
3,2-Feb,"Vectors, Matrices, and Dataframes",Introduction to pandas.,,Dickerson,
4,7-Feb,Jupyter notebook lab,,,"Denis, Anant, & Neil",
5,9-Feb,Best Practices for Data Science Projects,,,Dickerson,
```

Don’t write your own CSV or JSON parser

```
import csv
with open(“schedule.csv”, ”rb”) as f:
    reader = csv.reader(f, delimiter=“,”, quotechar=’”’)
    for row in reader:
        print(row)
```

(We’ll use `pandas` to do this much more easily and efficiently)

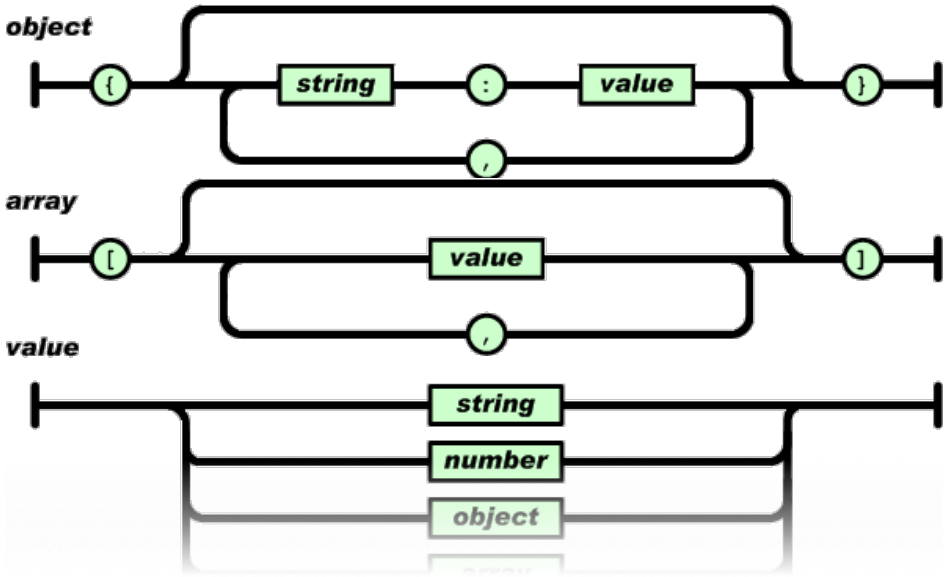
JSON FILES & STRINGS

JSON is a method for serializing objects:

- Convert an object into a string (done in Java in 131/132?)
- Deserialization converts a string back to an object

Easy for humans to read (and sanity check, edit)

Defined by three universal data structures



Python dictionary, Java Map, hash table, etc …

Python list, Java array, vector, etc …

Python string, float, int, boolean, JSON object, JSON array, …

Images from: http://www.json.org/

JSON IN PYTHON

Some built-in types: `"Strings", 1.0, True, False, None`

Lists: `["Goodbye", "Cruel", "World"]`

Dictionaries: `{"hello": "bonjour", "goodbye", "au revoir"}`

Dictionaries within lists within dictionaries within lists:

```
[1, 2, {"Help":[
                "I'm", {"trapped": "in"},
                "CMSC320"
                ]}]
```

JSON FROM TWITTER

```
GET https://api.twitter.com/1.1/friends/list.json?cursor=-
1&screen_name=twitterapi&skip_status=true&include_user_entitie
s=false
```


```
{
        "previous_cursor": 0,
        "previous_cursor_str": "0",
        "next_cursor": 1333504313713126852,
        "users": [{
                "profile_sidebar_fill_color": "252429",
                "profile_sidebar_border_color": "181A1E",
                "profile_background_tile": false,
                "name": "Sylvain Carle",
                "profile_image_url":
 "http://a0.twimg.com/profile_images/2838630046/4b82e286a659fae310012520f4f7
 56bb_normal.png",
                "created_at": "Thu Jan 18 00:10:45 +0000 2007", …
```

PARSING JSON IN PYTHON

Repeat: don't write your own CSV or JSON parser

- https://news.ycombinator.com/item?id=7796268
- rsdy.github.io/posts/dont_write_your_json_parser_plz.html

Python comes with a fine JSON parser

```
import json

r = requests.get(
"https://api.twitter.com/1.1/statuses/user_timeline.jso
n?screen_name=JohnPDickerson&count=100", auth=auth )

data = json.loads(r.content)
```

```
json.load(some_file)  # loads JSON from a file
json.dump(json_obj, some_file)  # writes JSON to file
json.dumps(json_obj)  # returns JSON string
```

XML, XHTML, HTML FILES AND STRINGS

Still hugely popular online, but JSON has essentially replaced XML for:

- Asynchronous browser ←→ server calls
- Many (most?) newer web APIs

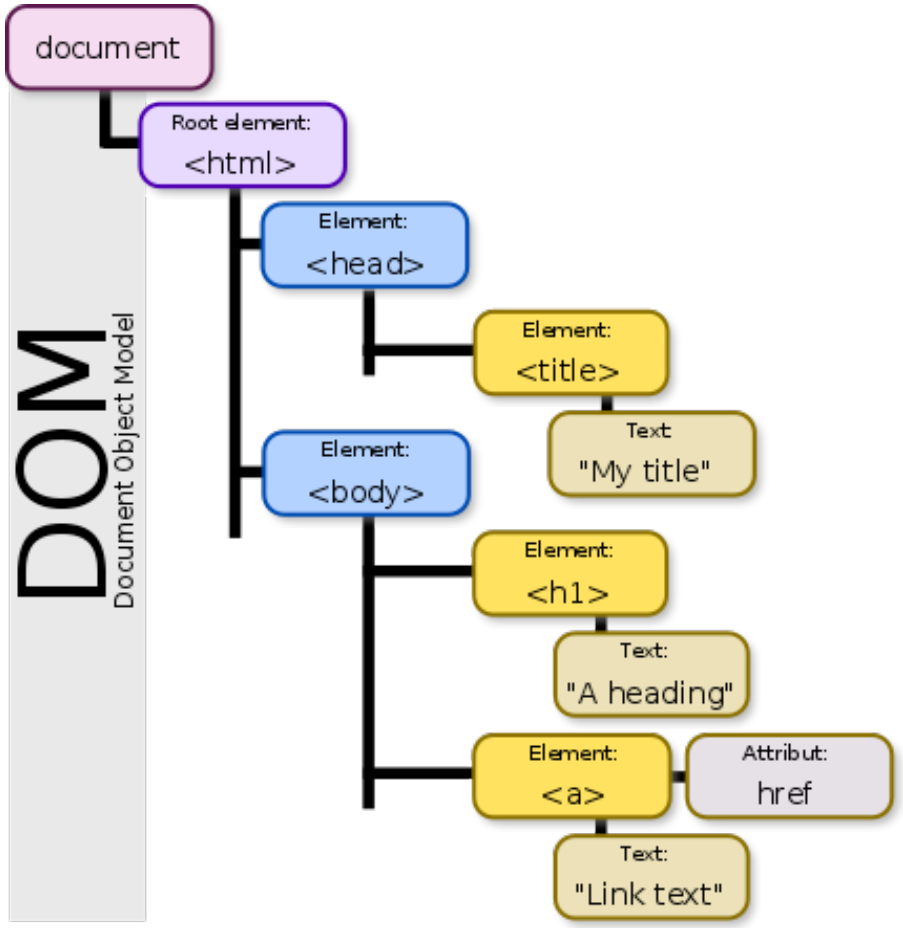
XML is a hierarchical markup language:

```
<tag attribute="value1">
        <subtag>
                Some content goes here
        </subtag>
        <openclosetag attribute="value2" />
</tag>
```

You probably won't see much XML, but you will see plenty of HTML, its substantially less well-behaved cousin …

Example XML from: Zico Kolter

DOCUMENT OBJECT MODEL (DOM)



XML encodes Document-Object Models ("the DOM")

The DOM is tree-structured.

Easy to work with! Everything is encoded via links.

Can be huge, & mostly full of stuff you don't need …

SAX

SAX (Simple API for XML) is an alternative “lightweight” way to process XML.

A SAX parser generates a stream of events as it parses the XML file. The programmer registers handlers for each one.

It allows a programmer to handle only parts of the data structure.

Example from John Canny

SCRAPING HTML IN PYTHON

HTML – the specification – is fairly pure

HTML – what you find on the web – is horrifying

We'll use BeautifulSoup:

- `conda install -c asmeurer beautiful-soup=4.3.2`

```
import requests
from bs4 import BeautifulSoup

r = requests.get(
"https://cs.umd.edu/class/fall2019/cmsc320/" )

root = BeautifulSoup( r.content )
root.find("div", id="schedule")\
    .find("table")\                  # find all schedule
    .find("tbody").findAll("a")  # links for CMSC320
```

BUILDING A WEB SCRAPER IN PYTHON

Totally not hypothetical situation:

- You really want to learn about data science, so you choose to download all of last semester's CMSC320 lecture slides to wallpaper your room …
- … but you now have carpal tunnel syndrome from clicking refresh on Piazza last night, and can no longer click on the PDF and PPTX links.

Hopeless? No! Earlier, you built a scraper to do this!

```
lnks = root.find("div", id="schedule")\
    .find("table")\                  # find all schedule
    .find("tbody").findAll("a")  # links for CMSC320
```

Sort of. You only want PDF and PPTX files, not links to other websites or files.

REGULAR EXPRESSIONS

Given a list of URLs (strings), how do I find only those strings that end in *.pdf or *.pptx?

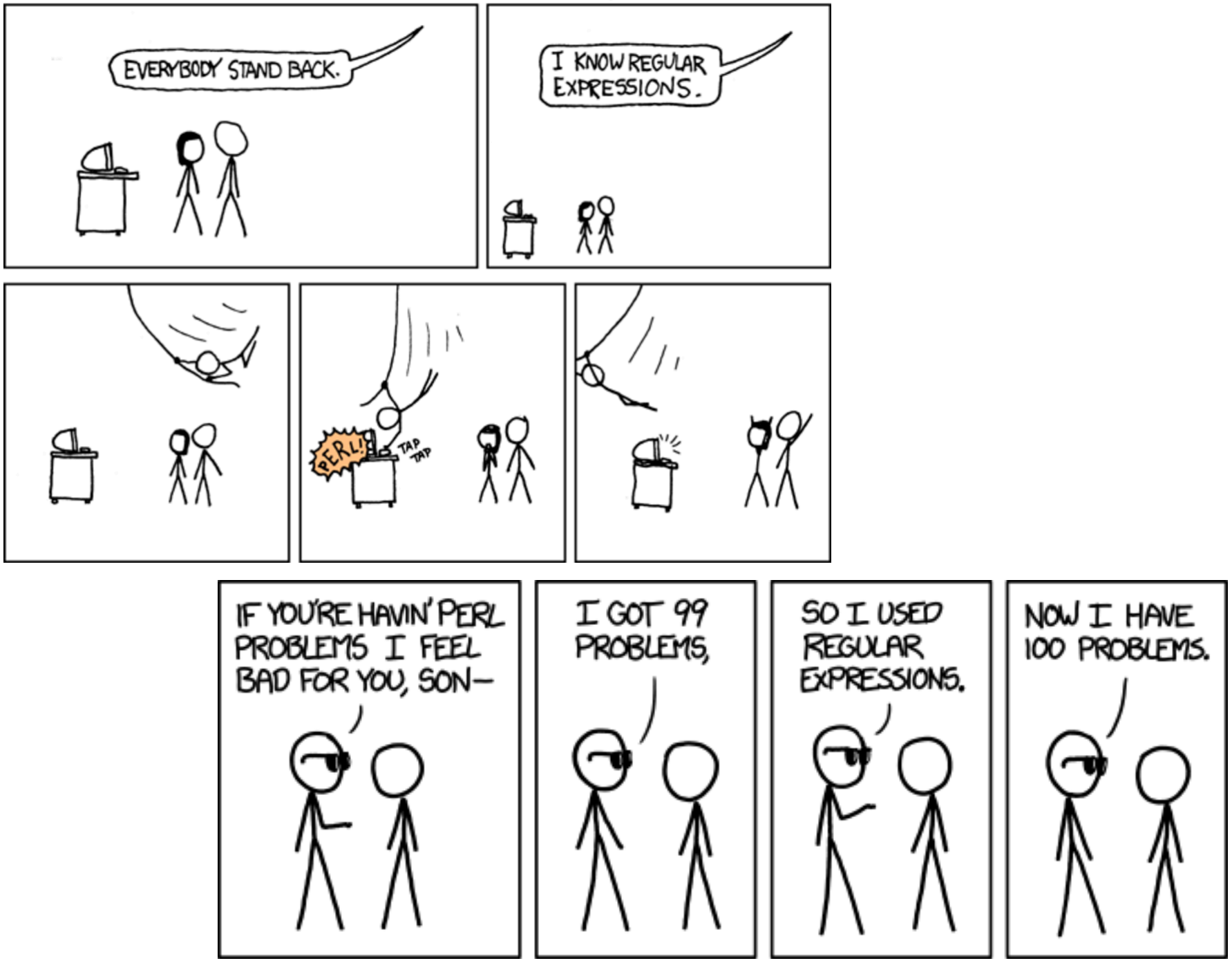
- Regular expressions!
- (Actually Python strings come with a built-in `endswith` function.)

```
"this_is_a_filename.pdf".endswith(("".pdf", ".pptx"))
```

What about .pDf or .pPTx, still legal extensions for PDF/PPTX?

- Regular expressions!
- (Or cheat the system again: built-in string `lower` function.)

```
"tHiS_IS_a_FileNAme.pDF".lower().endswith(
                                    (".pdf", ".pptx"))
```


EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP
IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON—
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.

REGULAR EXPRESSIONS

Used to search for specific elements, or groups of elements, that match a pattern

Indispensable for data munging and wrangling

Many constructs to search a variety of different patterns

Many languages/libraries (including Python) allow "compiling"

Much faster for repeated applications of the regex pattern

https://blog.codinghorror.com/to-compile-or-not-to-compile/

REGULAR EXPRESSIONS

Used to search for specific elements, or groups of elements, that match a pattern

```
import re

# Find the index of the 1st occurrence of "cmsc320"
match = re.search(r"cmsc320", text)
print( match.start() )
```

```
# Does start of text match "cmsc320"?
match = re.match(r"cmsc320", text)
```

```
# Iterate over all matches for "cmsc320" in text
for match in re.finditer(r"cmsc320", text):
    print( match.start() )
```

```
# Return all matches of "cmsc320" in the text
match = re.findall(r"cmsc320", text)
```

MATCHING MULTIPLE CHARACTERS

Can match sets of characters, or multiple and more elaborate sets and sequences of characters:

- Match the character ‘a’: `a`
- Match the character ‘a’, ‘b’, or ‘c’: `[abc]`
- Match any character except ‘a’, ‘b’, or ‘c’: `[^abc]`
- Match any digit: `\d` (= `[0123456789]` or `[0-9]`)
- Match any alphanumeric: `\w` (= `[a-zA-Z0-9_]`)
- Match any whitespace: `\s` (= `[ \t\n\r\f\v]`)
- Match any character: `.`

Special characters must be escaped: `.^$*+?{}\[]|()`

Thanks to: Zico Kolter

MATCHING SEQUENCES AND REPEATED CHARACTERS

A few common modifiers (available in Python and most other high-level languages; +, {n}, {n,} *may* not):

- Match character ‘a’ exactly once: `a`
- Match character ‘a’ zero or once: `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}`
- Match character ‘a’ at least n times: `a{n,}`

Example: match all instances of “University of <somewhere>” where <somewhere> is an alphanumeric string with at least 3 characters:

- `\s*University\sof\s\w{3,}`

GROUPS

What if we want to know more than just "did we find a match" or "where is the first match" …?

Grouping asks the regex matcher to keep track of certain portions – surrounded by (parentheses) – of the match

```
\s*([Uu]niversity)\s([Oo]f)\s(\w{3,})
```

```
regex = r"\s*([Uu]niversity)\s([Oo]f)\s(\w{3,})"
m = re.search( regex, "university Of Maryland" )
print( m.groups() )
```

```
('university', 'Of', 'Maryland')
```

SIMPLE EXAMPLE: PARSE AN EMAIL ADDRESS

Mail::RFC822::Address Perl module for RFC 822

NAMED GROUPS

Raw grouping is useful for one-off exploratory analysis, but may get confusing with longer regexes

- Much scarier regexes than that email one exist in the wild …

Named groups let you attach position-independent identifiers to groups in a regex

```
(?P<some_name> …)
```

```
regex = "\s*[Uu]niversity\s[Oo]f\s(?P<school>(\w{3,}))"
m = re.search( regex, “University of Maryland” )
print( m.group(‘school’) )
```

```
'Maryland'
```

SUBSTITUTIONS

The Python `string` module contains basic functionality for find-and-replace within strings:

```
"abcabcabc".replace("a", "X")
```

```
'XbcXbcXbc'
```

For more complicated stuff, use regexes:

```
text = "I love Introduction to Data Science"
re.sub(r"Data Science", r"Schmada Schmience", text)
```

```
'I love Introduction to Schmada Schmience'
```

Can incorporate groups into the matching

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


Thanks to: Zico Kolter

COMPILED REGEXES

If you're going to reuse the same regex many times, or if you aren't but things are going slowly for some reason, try compiling the regular expression.

- https://blog.codinghorror.com/to-compile-or-not-to-compile/

```
# Compile the regular expression "cmsc320"
regex = re.compile(r"cmsc320")

# Use it repeatedly to search for matches in text
regex.match( text )     # does start of text match?
regex.search( text )    # find the first match or None
regex.findall( text )   # find all matches
```

Interested? CMSC330, CMSC430, CMSC452, talk to me.

DOWNLOADING A BUNCH OF FILES

Import the modules

```
import re
import requests
from bs4 import BeautifulSoup
try:
    from urllib.parse import urlparse
except ImportError:
    from urlparse import urlparse
```

Get some HTML via HTTP

```
# HTTP GET request sent to the URL url
r = requests.get( url )

# Use BeautifulSoup to parse the GET response
root = BeautifulSoup( r.content )
lnks = root.find("div", id="schedule")\
           .find("table")\
           .find("tbody").findAll("a")
```

DOWNLOADING A BUNCH OF FILES

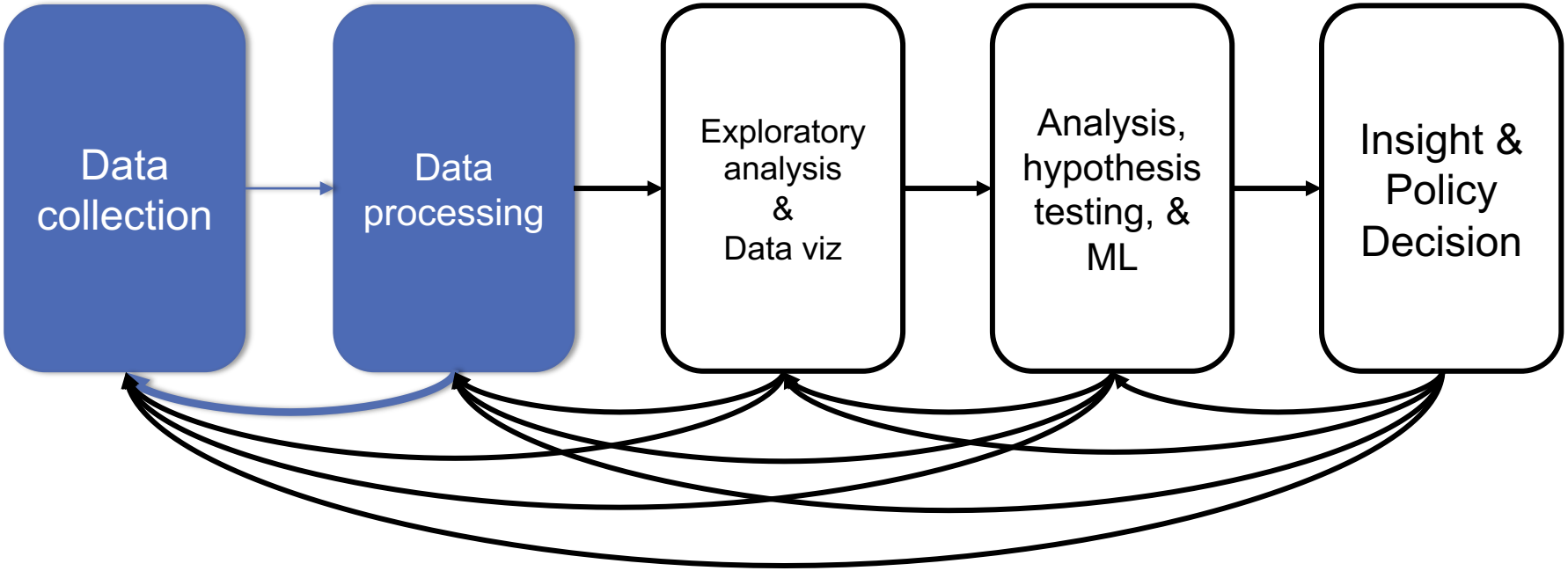
Parse exactly what you want

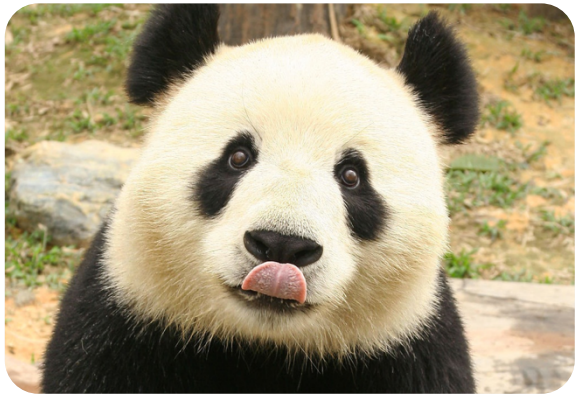
```
# Cycle through the href for each anchor, checking
# to see if it's a PDF/PPTX link or not
for lnk in lnks:
    href = lnk['href']

    # If it's a PDF/PPTX link, queue a download
    if href.lower().endswith(('.pdf', '.pptx')):
```

Get some more data?!

```
        urld = urlparse.urljoin(url, href)
        rd = requests.get(urld, stream=True)

        # Write the downloaded PDF to a file
        outfile = path.join(outbase, href)
        with open(outfile, 'wb') as f:
            f.write(rd.content)
```

NEXT LECTURE





NEXT CLASS:

NUMPY, SCIPY, AND DATAFRAMES

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$