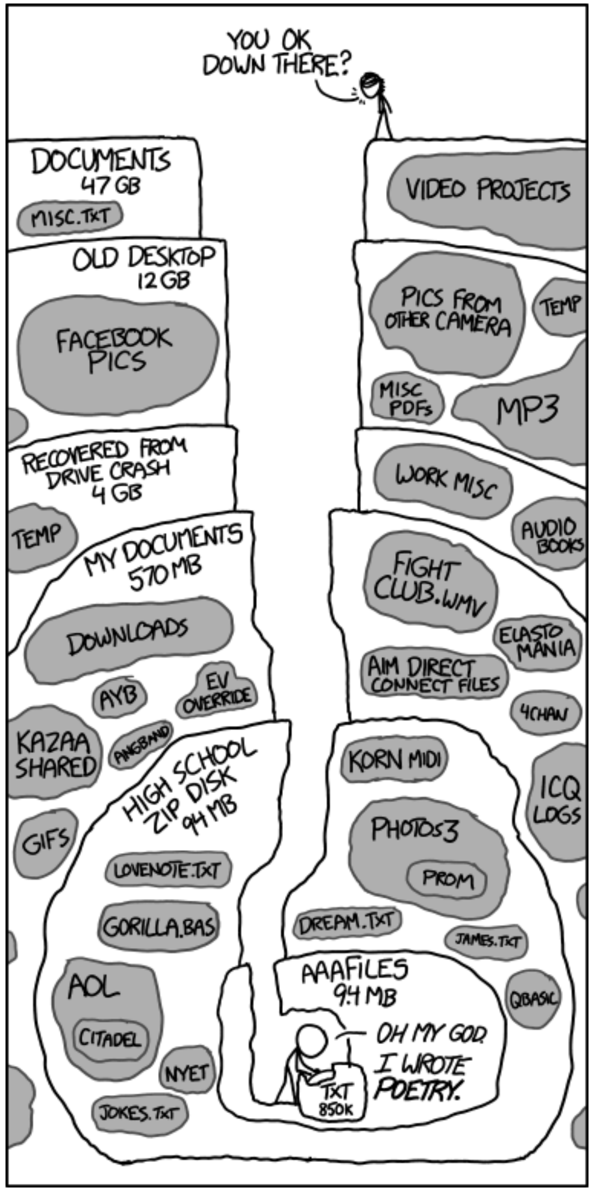
Data Scraping in Python

Nicholas Mattei, *Tulane University*
CMPS3660 – Introduction to Data Science – Fall 2019
https://rebrand.ly/TUDataScience

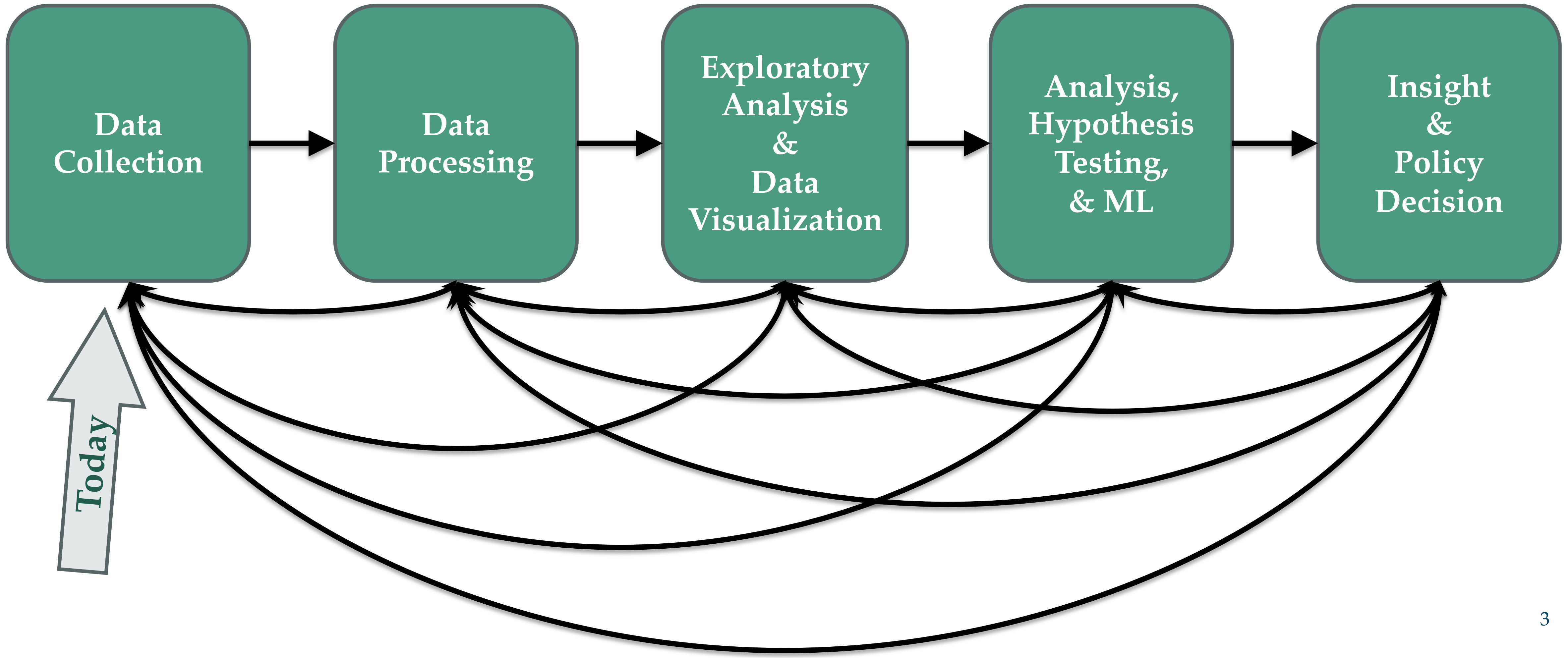
Many Thanks
Slides based off Introduction to Data Science from John P. Dickerson -
https://cmsc320.github.io/

Announcements

- Reminder: Lab 1 + 2 Due at End of Day
- Go over Questions 2
- Suggestion: How I'd setup git…
- John has an announcement!



The Data LifeCycle


Data Collection
Data Processing
Exploratory Analysis & Data Visualization
Analysis, Hypothesis Testing, & ML
Insight & Policy Decision
Today

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

$\left.\right\}$ Covered today.

Wherefore art thou, API?

- A web-based **A**pplication **P**rogramming **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.

- We're going to use the Python requests module.
  - Documentation: https://2.python-requests.org/en/master/user/quickstart/

“Send me a specific request”

- Most web API queries we’ll be doing will use HTTP requests:
- `conda install anaconda requests`

```
r = requests.get('https://api.github.com/users/nmattei')
```

```
r.status_code
```

```
200
```

```
r.headers[‘content-type’]
```

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

```
r.json()
```

```
{{'login': 'nmattei', 'id': 1206578, …}
```

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

HTTP Requests – How the Browser Does It.

- https://www.google.com/search?q=%27tulane%20university%27

??????????

- HTTP GET Request:
- GET /search? q=%27tulane%20university%27
- Host: www.google.com/search
- User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.1) Gecko/20100101 Firefox/10.0.1

```
params = { “q”: “tulane university”}
r = requests.get(“https://www.google.com/search”,
                    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.
- **Re**presentational **S**tate **T**ransfer (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

- Good example: The ITunes API
  - https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/

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/users”,
          params={”access_token”: token})
print( r.content )
```

```
{"login":”nmattei","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

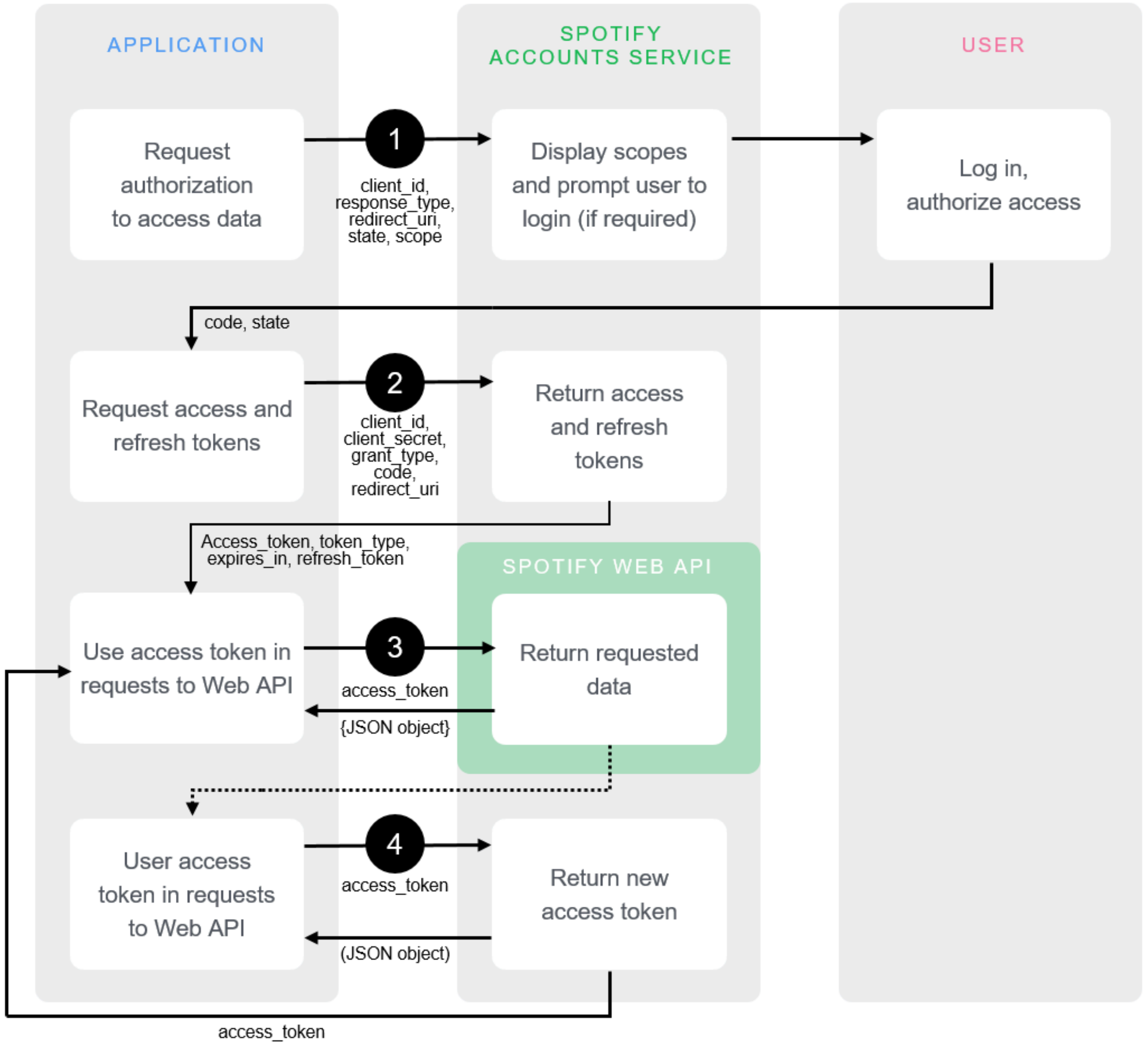
Authentication and OAuth

- Old and busted:

```
r = requests.get(“https://api.github.com/user”,
        auth=(“nmattei”, “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
- These can get complicated and are site specific.. Some examples below.
  - https://developers.facebook.com/docs/facebook-login/access-tokens/
  - https://developer.spotify.com/documentation/general/guides/authorization-guide/

Example OAuth: Spotify


APPLICATION
SPOTIFY ACCOUNTS SERVICE
USER
Request authorization to access data
1
client_id, response_type, redirect_uri, state, scope
Display scopes and prompt user to login (if required)
Log in, authorize access
code, state
Request access and refresh tokens
2
client_id, client_secret, grant_type, code, redirect_uri
Return access and refresh tokens
Access_token, token_type, expires_in, refresh_token
SPOTIFY WEB API
Use access token in requests to Web API
3
access_token
Return requested data
{JSON object}
User access token in requests to Web API
4
access_token
Return new access token
(JSON object)
access_token

"… 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",
  }
}
```

CSV Files in Python

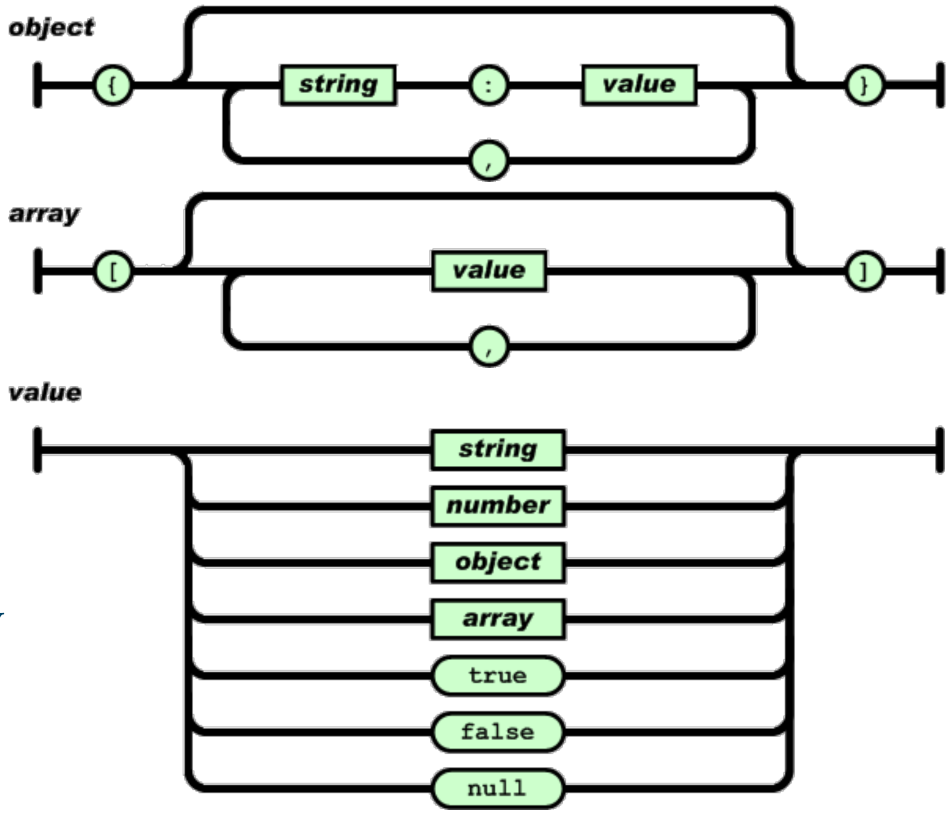
- Any CSV reader worth anything can parse files with any delimiter, not just a comma (e.g., “TSV” for tab-separated)
  - 1,26-Jan,Introduction,—,"pdf, pptx",Dickerson,
    2,31-Jan,Scraping Data with Python,Anaconda's Test Drive.,,Dickerson,
    3,2-Feb,"Vectors, Matrices, and Dataframes",Introduction 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 ITunes API

- GET https://itunes.apple.com/search?term=the%2Bmeters&entity=album

```
{ 'resultCount': 15,
  'results’: [
               {'wrapperType': 'collection’,
                'collectionType': 'Album’,
                'artistId': 7314214,
                'collectionId': 213532006,
                'amgArtistId': 4907,
                'artistName': 'The Meters’,
                'collectionName': 'The Very Best of the Meters’,
                'collectionCensoredName': 'The Very Best of the Meters’,
               …
```

Parsing JSON In Python

- Repeat: **don't** write your own CSV or JSON parser
  - https://news.ycombinator.com/item?id=7796268
  - https://rsdy.github.io/posts/dont_write_your_json_parser_plz.html
- Python comes with a fine JSON parser – USE IT!

```
import json

r = requests.get('https://itunes.apple.com/search?term=the%2Bmeters&entity=album')

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

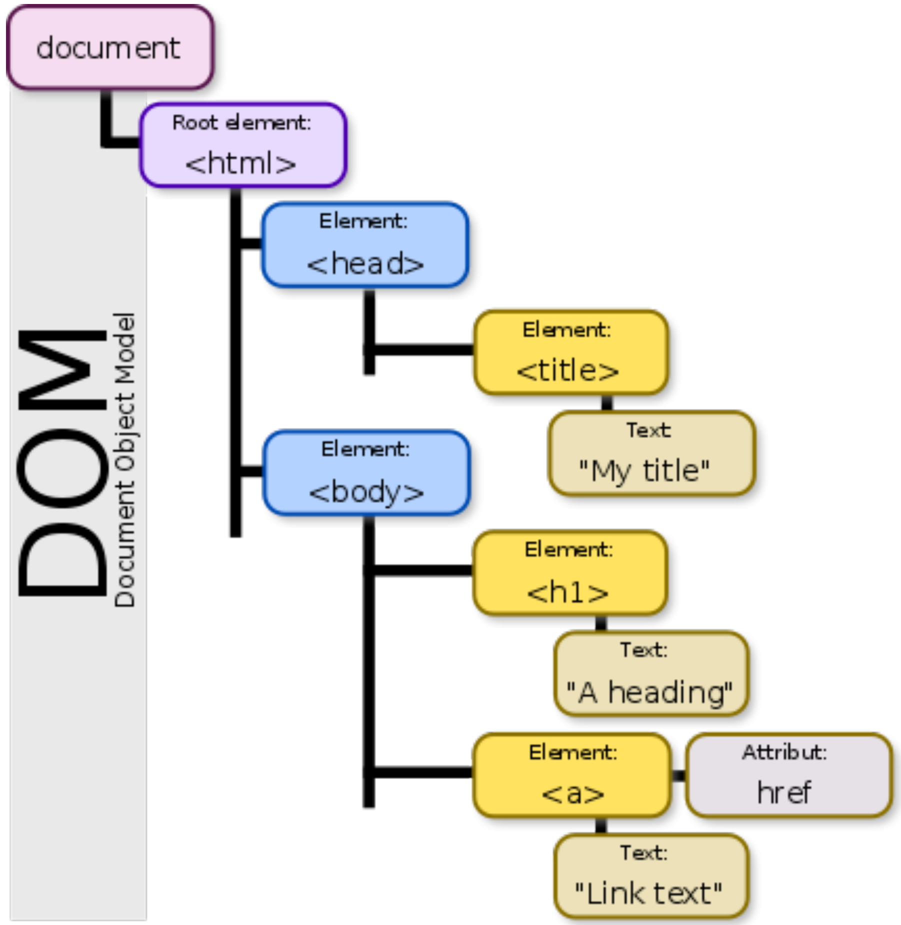
- Still hugely popular online, but JSON has essentially replaced XML for:
  - Asynchronous browser ←→ server calls
  - Many (most?) newer web APIs
- XML is a hierarchical markup language:

  -
    ```
    <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 …

Scraping HTML in Python

- HTML – the specification – is fairly pure
- HTML – what you find on the web – is horrifying
  - Getting better with automated generation -- https://tulaneintrodatascience.github.io/
- We’ll use BeautifulSoup:
  - `conda install -c beautifulsoup4`

```
import requests
from bs4 import BeautifulSoup

r = requests.get(“https://tulaneintrodatascience.github.io/”)

root = BeautifulSoup( r.content )
root.find(“table”)                      # Find a the tables.
root.find(“tbody”).findAll(“a”)  # links in that table.
```

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 the 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 links.
- Hopeless? No! Earlier, you built a scraper to do this!

```
lnks = root.find("table").findAll("a") # links within the table.
```

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

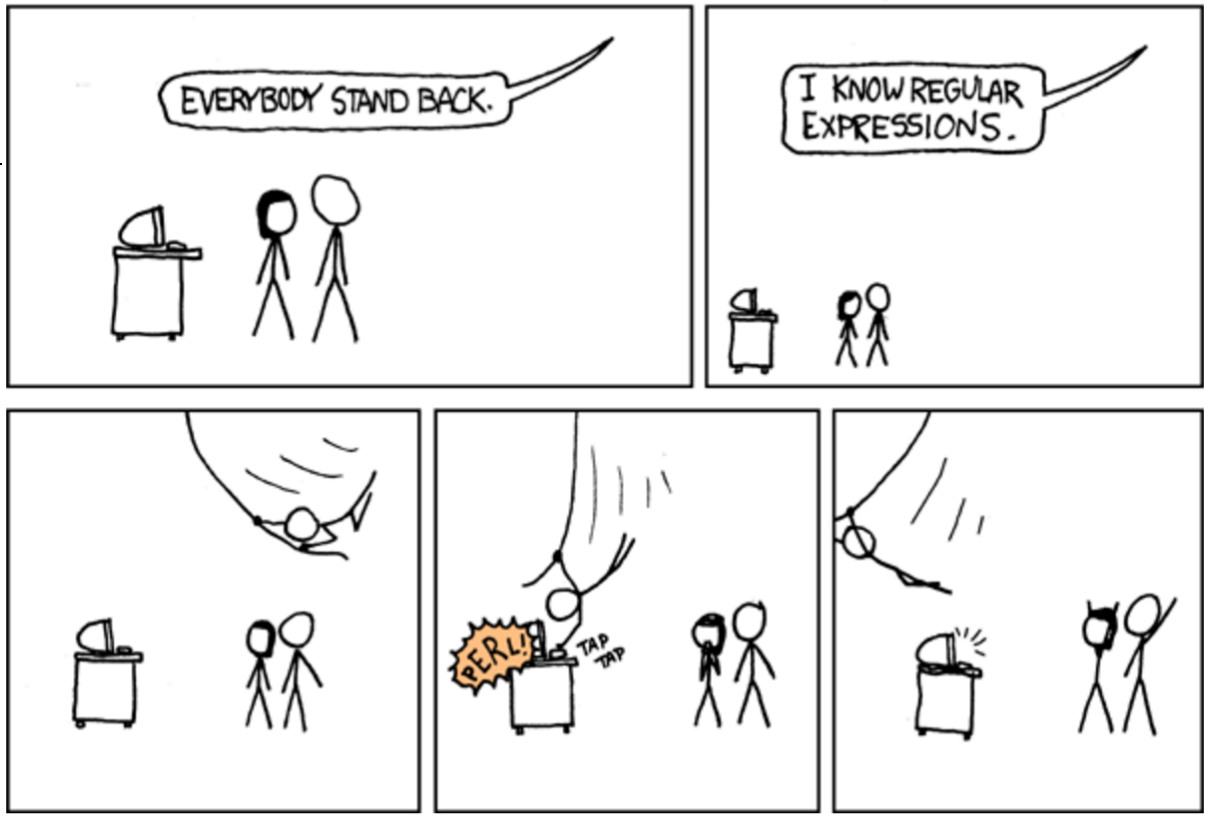
Detour: Regular Expressions

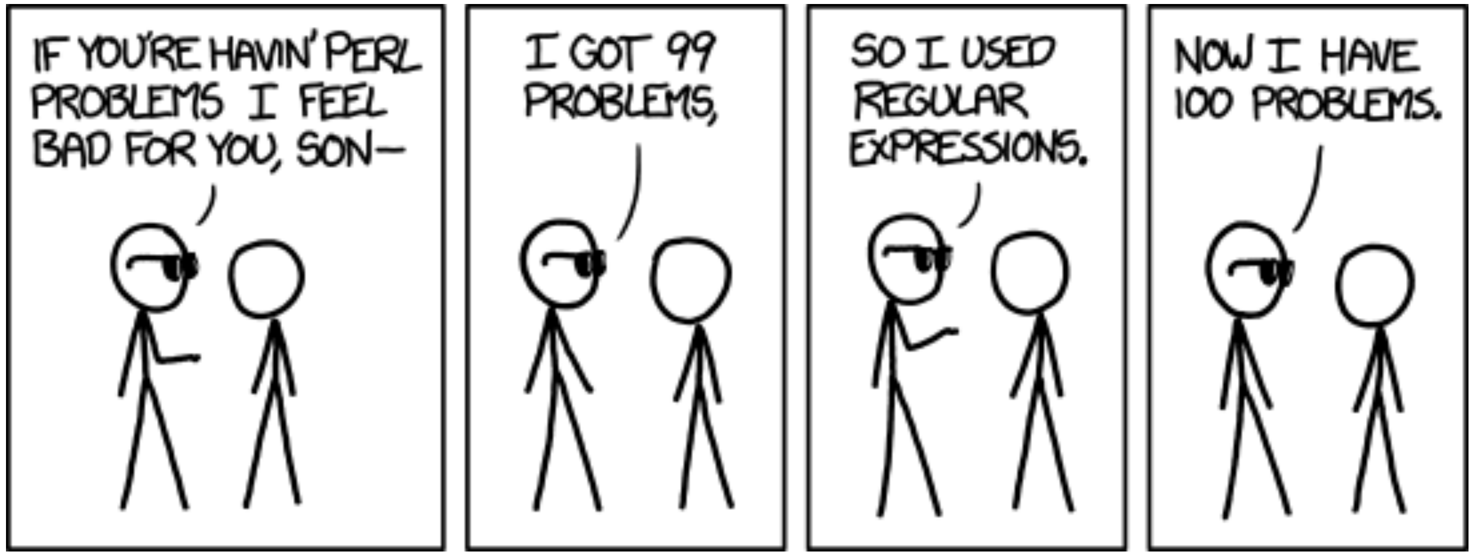
- Given a list of URLs (strings), how do I find only those strings that end in *.pdf?
  - Regular expressions!
  - (Actually Python strings come with a built-in `endswith` function.)

```
“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/

- Intro Tutorial from Python: https://docs.python.org/3/howto/regex.html

Regular Expressions

The r is for ‘raw string’ and it’s very important…
https://stackoverflow.com/questions/21104476/what-does-the-r-in-pythons-re-compiler-pattern-flags-mean

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

```
import re

# Find the index of the 1st occurrence of “CMPS 3660”
match = re.search(r”CMPS 3660”, text)
print( match.start() )
```

```
# Does start of text match “CMPS 3660”?
match = re.match(r”CMPS 3660”, text)
```

```
# Iterate over all matches for “CMPS 3660” in text
for match in re.finditer(r”cmsc320”, text):
      print( match.start() )
```

```
# Return all matches of “CMPS 3660” 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

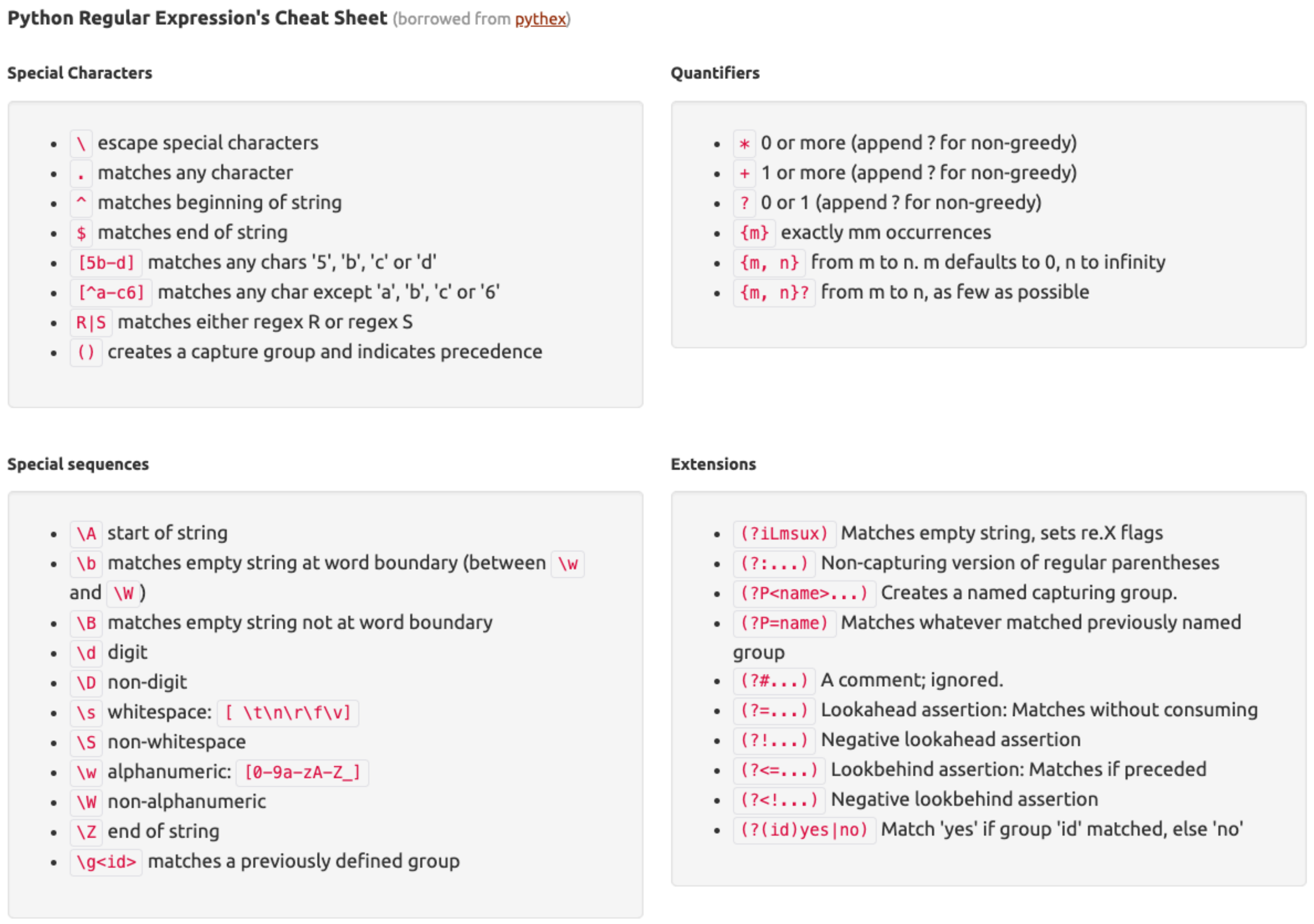
- A few common modifiers (available in Python and most high-level languages; +, {n}, {n,} *may* not):
  - Match character ‘a’ exactly once: `a`
  - Match character ‘a’ zero or once: `a?`
  - Match character ‘a’ zero or more times: `a*`
  - Match character ‘a’ one or more times: `a+`
  - Match character ‘a’ exactly *n* times: `a{n}`
  - Match character ‘a’ at least n times: `a{n,}`

- Useful to keep a tester around: http://www.pyregex.com/

Cheat Sheet

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

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



Python Regular Expression's Cheat Sheet (borrowed from pythex)

Special Characters

- `\` escape special characters
- `.` matches any character
- `^` matches beginning of string
- `$` matches end of string
- `[5b-d]` matches any chars '5', 'b', 'c' or 'd'
- `[^a-c6]` matches any char except 'a', 'b', 'c' or '6'
- `R|S` matches either regex R or regex S
- `()` creates a capture group and indicates precedence

Quantifiers

- `*` 0 or more (append ? for non-greedy)
- `+` 1 or more (append ? for non-greedy)
- `?` 0 or 1 (append ? for non-greedy)
- `{m}` exactly mm occurrences
- `{m, n}` from m to n. m defaults to 0, n to infinity
- `{m, n}?` from m to n, as few as possible

Special sequences

- `\A` start of string
- `\b` matches empty string at word boundary (between `\w` and `\W`)
- `\B` matches empty string not at word boundary
- `\d` digit
- `\D` non-digit
- `\s` whitespace: `[ \t\n\r\f\v]`
- `\S` non-whitespace
- `\w` alphanumeric: `[0-9a-zA-Z_]`
- `\W` non-alphanumeric
- `\Z` end of string
- `\g<id>` matches a previously defined group

Extensions

- `(?iLmsux)` Matches empty string, sets re.X flags
- `(?:...)` Non-capturing version of regular parentheses
- `(?P<name>...)` Creates a named capturing group.
- `(?P=name)` Matches whatever matched previously named group
- `(?#...)` A comment; ignored.
- `(?=...)` Lookahead assertion: Matches without consuming
- `(?!...)` Negative lookahead assertion
- `(?<=...)` Lookbehind assertion: Matches if preceded
- `(?<!...)` Negative lookbehind assertion
- `(?(id)yes|no)` Match 'yes' if group 'id' matched, else 'no'

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 Kentucky" )
print( m.groups() )
```

```
('university', 'Of', 'Kentucky')
```

Simple Example: Parse an Email Address

-
  ```
  (?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t] )+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?: \r\n)?[
  \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:( ?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
  \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0 31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\ ](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
  \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+ (?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?: (?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
  \031]+(?:(?:(?:\r\n)?[ \t])+|\Z |(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n) ?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-
  \031]+(?:(?:(?:\ r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n) ?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t] )*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])* )(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
  )+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*) *:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
  |\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r \n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?: \r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t ]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031 ]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]( ?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(? :(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(? :\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(? :(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)? [ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]| \\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<> @,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|" (?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t] )*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\ ".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(? :[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[ \]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000- \031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|( ?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,; :\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([ ^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\" .\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\ ]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\ [\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\ r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\] |\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \0 00-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\ .|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@, ;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(? :[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])* (?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\". \[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[ ^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\".\[\] ]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*( ?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[\["()<>@,;:\\ ".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:( ?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
  \t])+|\Z|(?=[ \["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t ])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
  ])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(? :\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
  \Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?: [^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
  ]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n) ?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
  ()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n) ?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
  @,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
  ;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t] )*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
  ".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)? (?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
  \[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?: \r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
  "()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t]) *))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
  +|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\ .(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
  |(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:( ?:\r\n)?[ \t])*))*)?;\s*)
  ```

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

Named Groups

- Raw grouping is useful for 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 Kentucky” )
print( m.group(‘school’) )
```

```
'Kentucky'
```

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 “CMPS 3660”
regex = re.compile(r”CMPS 3660”)

# 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
```

Downloading a bunch of files

Import the modules

```
import re
import requests
from bs4 import BeautifulSoup
from urllib.parse 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("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')):
```

Working code in Lecture05 Notebook!

Get some more data?!

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

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