INTRODUCTION TO DATA SCIENCE

JOHN P DICKERSON

Lecture #4 – 09/10/2020

CMSC320
Tuesdays & Thursdays
5:00pm – 6:15pm
(… or anytime on the Internet)

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

ANNOUNCEMENTS

Register on Piazza: piazza.com/umd/fall2020/cmsc320

- 249 have registered already
- ~1 has not registered yet

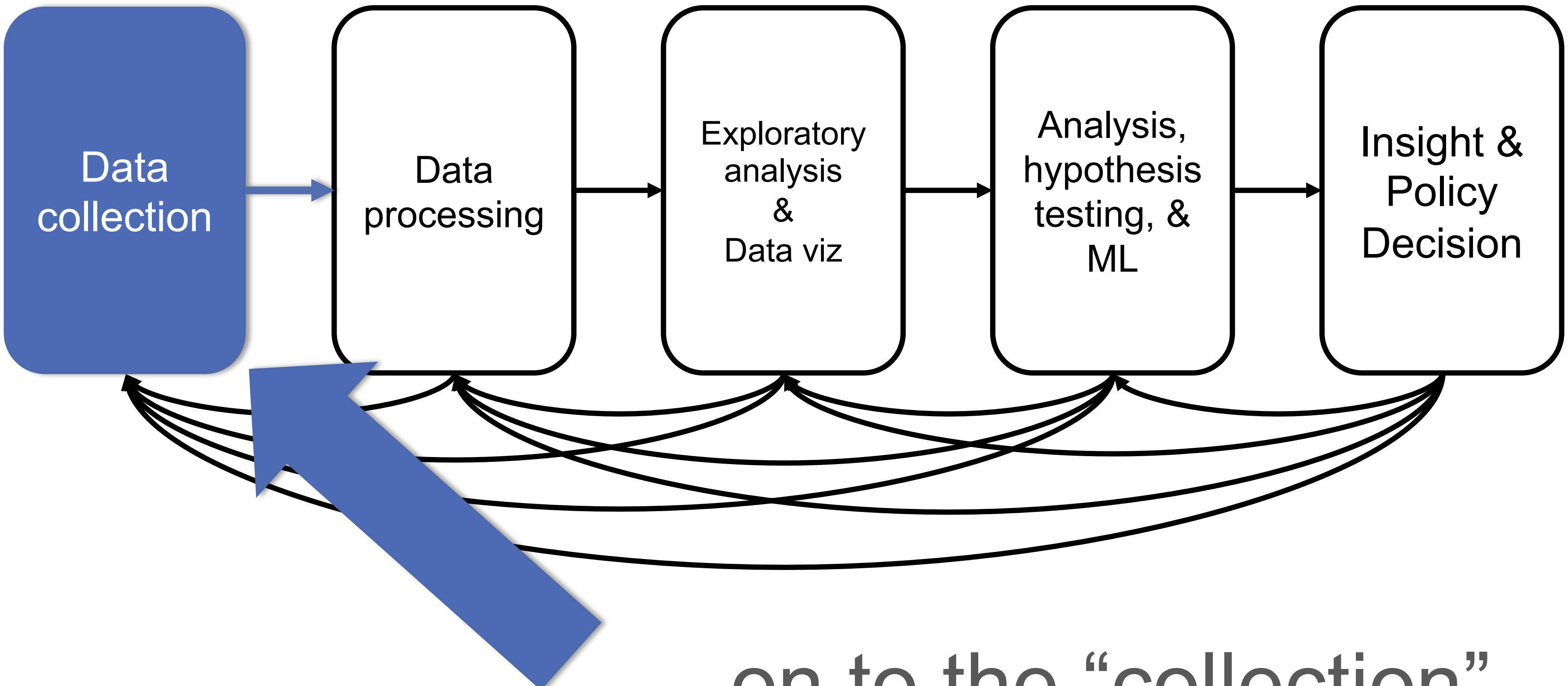
If you were on Piazza, you'd know …

- Project 1 will be out shortly. **(Worth 10% of grade, as are each of the four projects.)**
- Link will be on course website @ cmsc320.github.io

We've also linked some reading for the week!

- Second quiz is due Tuesday at noon; on ELMS now.

TODAY'S LECTURE (CONTINUATION OF LEC #3)



… 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)

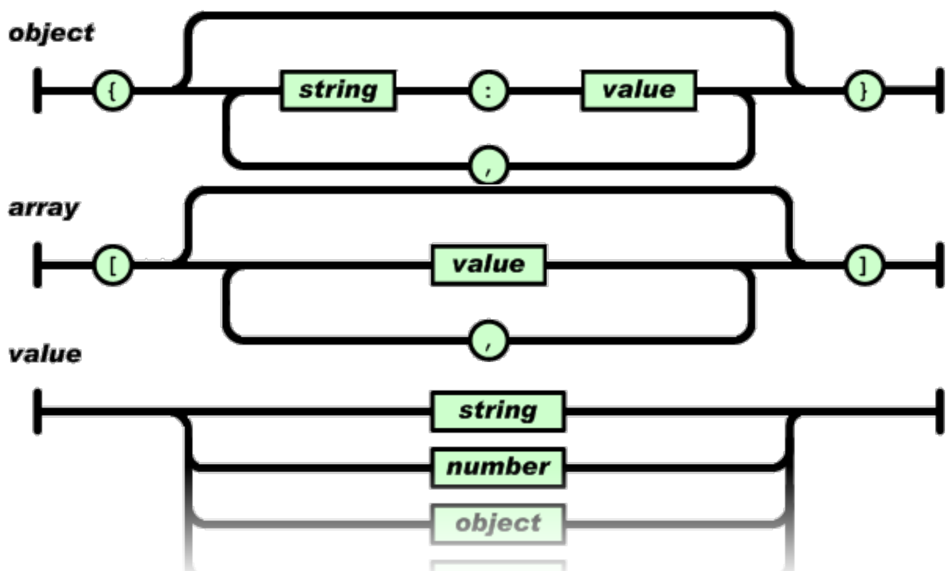
JSON FILES & STRINGS

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_entities=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/2838630046/4b82e286a659fae310012520f4f756bb_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.json?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

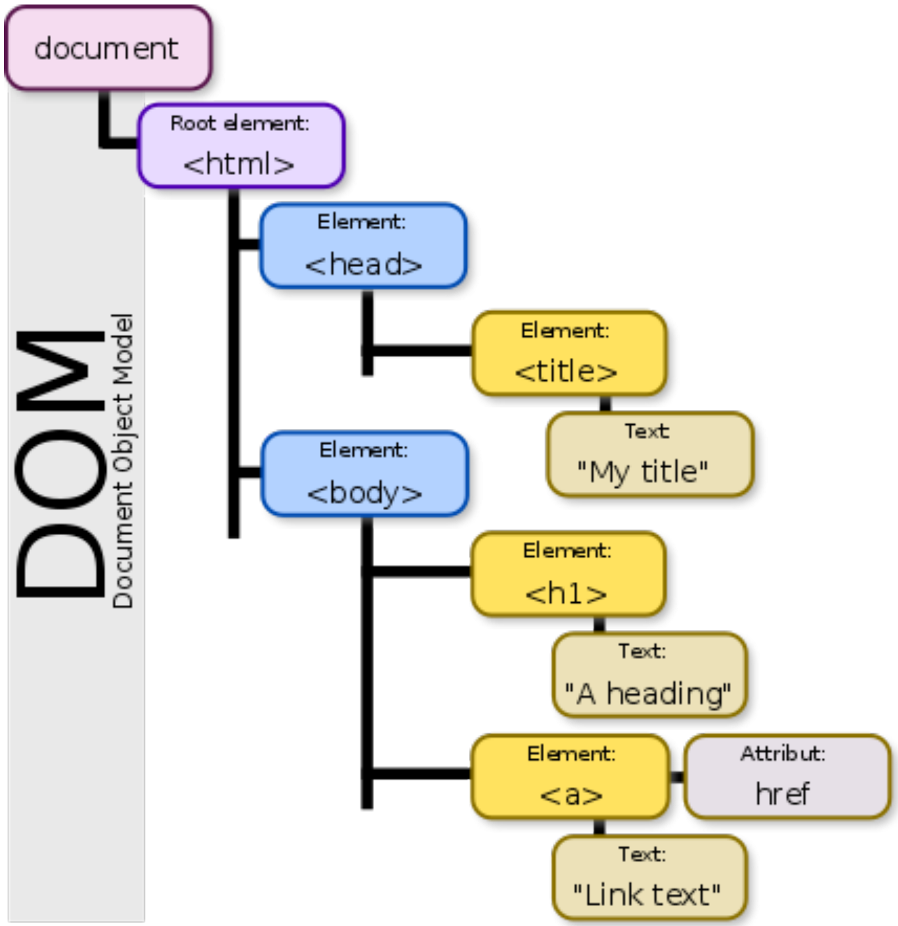
XML is a hierarchical markup language:

```
<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://cmsc320.github.io" )

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?

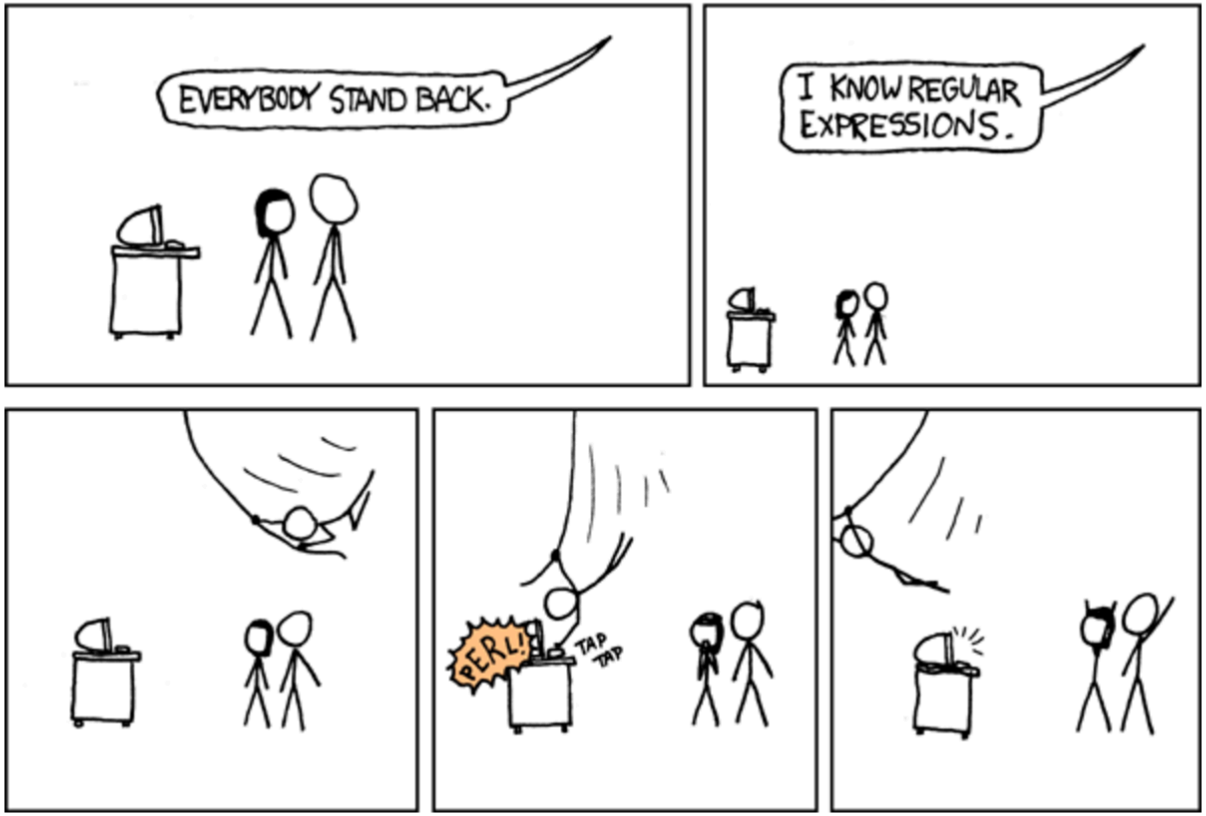
- Regular expressions!
- (Actually Python strings come with a built-in `endswith` function.)

```
“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

```
(?:(?:\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 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

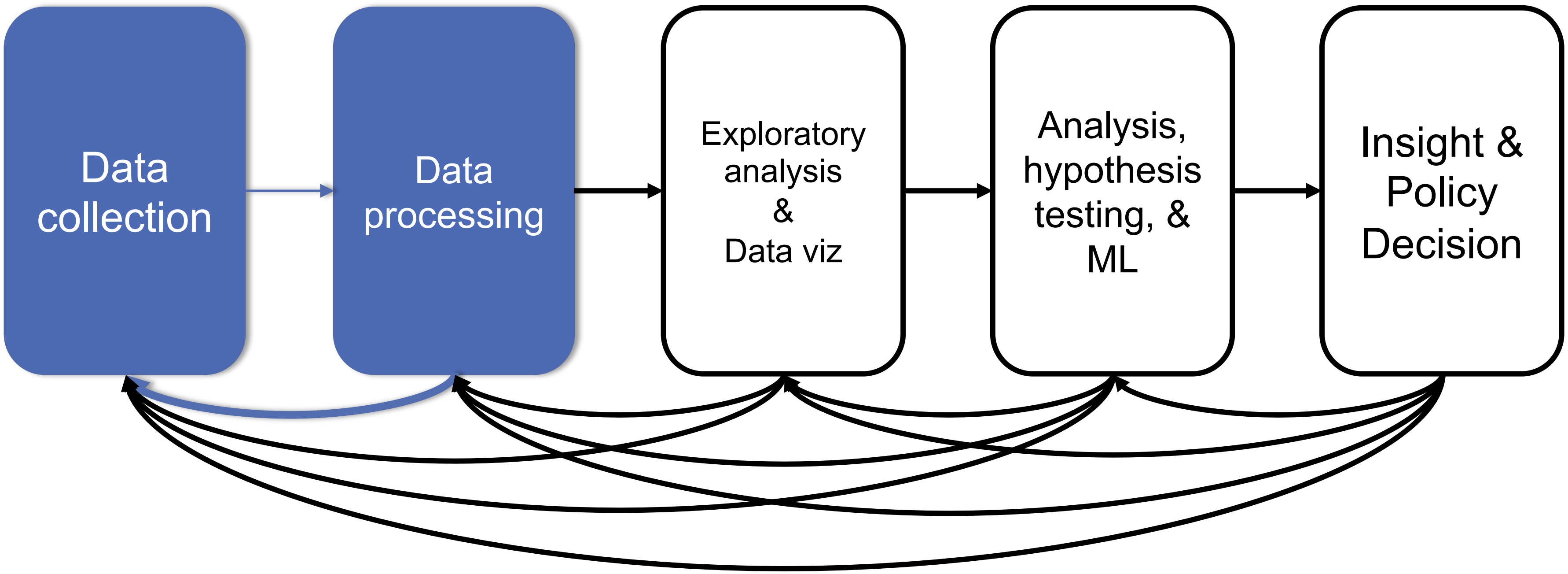
Parse exactly what you want

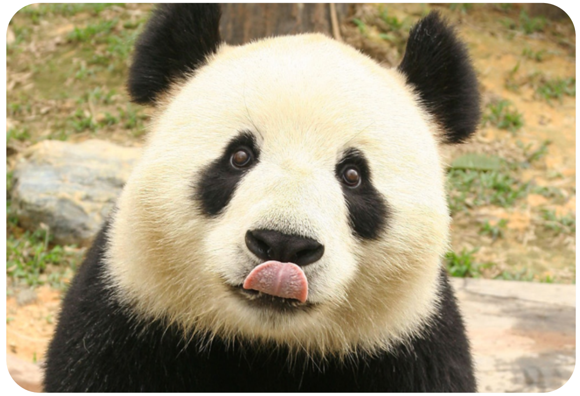
```
# Cycle through the href for each anchor, checking
# to see if it's a PDF/PPTX link or not
for lnk in lnks:
    href = lnk['href']

    # 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 …


Data collection
Data processing
Exploratory analysis & Data viz
Analysis, hypothesis testing, & ML
Insight & Policy Decision



NEXT UP:

NUMPY, SCIPY, AND DATAFRAMES

pandas

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

NEXT FEW CLASSES

1. **NumPy: Python Library for Manipulating nD Arrays**

   Multidimensional Arrays, and a variety of operations including Linear Algebra

2. **Pandas: Python Library for Manipulating Tabular Data**

   Series, Tables (also called **DataFrames**)
   Many operations to manipulate and combine tables/series

3. **Relational Databases**

   Tables/Relations, and SQL (similar to Pandas operations)

4. **Apache Spark**

   Sets of objects or key-value pairs
   MapReduce and SQL-like operations

NEXT FEW CLASSES

1. **NumPy: Python Library for Manipulating nD Arrays**

   Multidimensional Arrays, and a variety of operations including Linear Algebra

2. **Pandas: Python Library for Manipulating Tabular Data**

   Series, Tables (also called **DataFrames**)
   Many operations to manipulate and combine tables/series

3. **Relational Databases**

   Tables/Relations, and SQL (similar to Pandas operations)

4. **Apache Spark**

   Sets of objects or key-value pairs
   MapReduce and SQL-like operations

NUMERIC & SCIENTIFIC APPLICATIONS

Number of third-party packages available for numerical and scientific computing

These include:

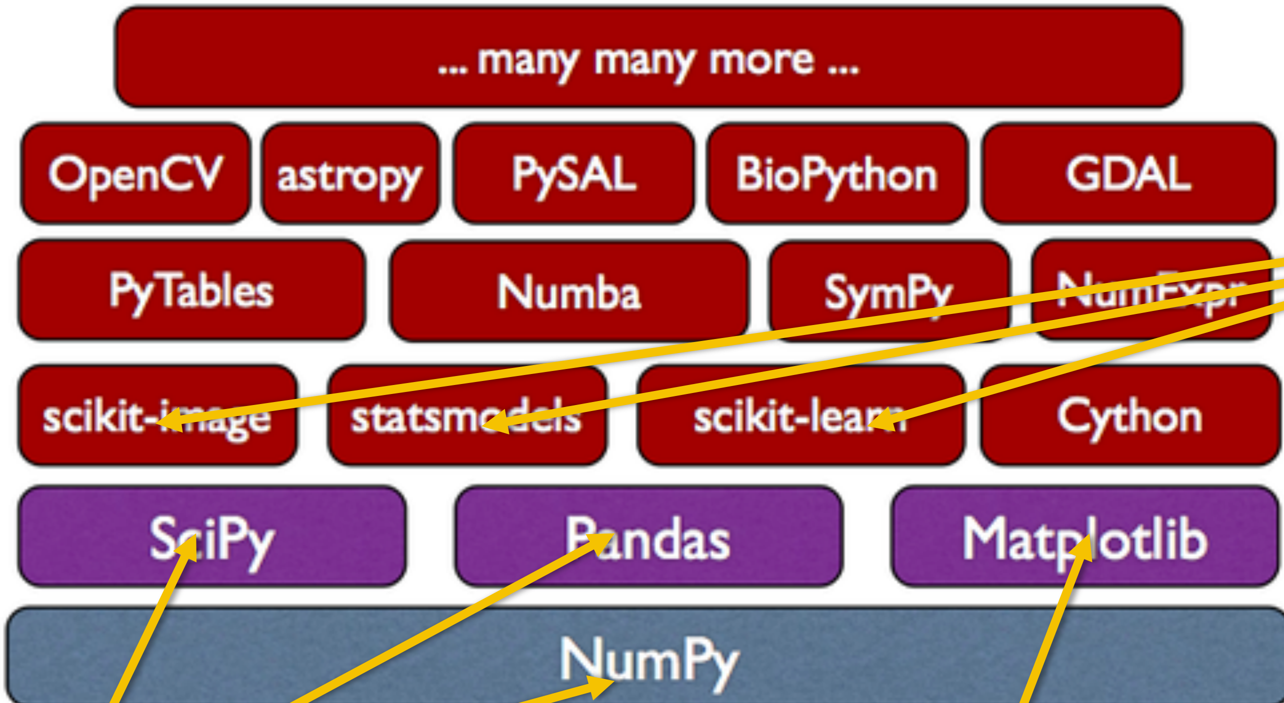
- NumPy/SciPy – numerical and scientific function libraries.
- numba – Python compiler that support JIT compilation.
- ALGLIB – numerical analysis library.
- pandas – high-performance data structures and data analysis tools.
- pyGSL – Python interface for GNU Scientific Library.
- ScientificPython – collection of scientific computing modules.

NUMPY AND FRIENDS

By far, the most commonly used packages are those in the NumPy stack. These packages include:

- NumPy: similar functionality as Matlab
- SciPy: integrates many other packages like NumPy
- Matplotlib & Seaborn – plotting libraries
- iPython via Jupyter – interactive computing
- Pandas – data analysis library
- SymPy – symbolic computation library

THE NUMPY STACK



Mid- & Late-semester

Today/next class

Later

Image from Continuum Analytics

NUMPY

Among other things, NumPy contains:

- A powerful *n*-dimensional array object.
- Sophisticated (broadcasting/universal) functions.
- Tools for integrating C/C++ and Fortran code.
- Useful linear algebra, Fourier transform, and random number capabilities, etc.

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

[FSU]

NUMPY

`ndarray` object: an *n*-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance

Several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size. Modifying the size means creating a new array.
- NumPy arrays must be of the same data type, but this can include Python objects – may not get performance benefits
- More efficient mathematical operations than built-in sequence types.

[FSU]

NUMPY DATATYPES

Wider variety of data types than are built-in to the Python language by default.

Defined by the `numpy.dtype` class and include:

- intc (same as a C integer) and intp (used for indexing)
- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- float16, float32, float64
- complex64, complex128
- bool_, int_, float_, complex_ are shorthand for defaults.

These can be used as functions to cast literals or sequence types, as well as arguments to NumPy functions that accept the `dtype` keyword argument.

NUMPY DATATYPES

```
>>> import numpy as np
>>> x = np.float32(1.0)
>>> x
1.0
>>> y = np.int_([1,2,4])
>>> y
array([1, 2, 4])
>>> z = np.arange(3, dtype=np.uint8)
>>> z
array([0, 1, 2], dtype=uint8)
>>> z.dtype
dtype('uint8')
```


[FSU]

NUMPY ARRAYS

There are a couple of mechanisms for creating arrays in NumPy:

- Conversion from other Python structures (e.g., lists, tuples)
  - Any sequence-like data can be mapped to a ndarray
- Built-in NumPy array creation (e.g., `arange`, `ones`, `zeros`, etc.)
  - Create arrays with all zeros, all ones, increasing numbers from 0 to 1 etc.
- Reading arrays from disk, either from standard or custom formats (e.g., reading in from a CSV file)

NUMPY ARRAYS

In general, any numerical data that is stored in an array-like container can be converted to an `ndarray` through use of the `array()` function. The most obvious examples are sequence types like lists and tuples.

```
>>> x = np.array([2,3,1,0])

>>> x = np.array([2, 3, 1, 0])

>>> x = np.array([[1,2.0],[0,0],(1+1j,3.)])

>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j],
[ 1.+1.j, 3.+0.j]])
```

[FSU]

NUMPY ARRAYS

Creating arrays from scratch in NumPy:

- `zeros(shape)` – creates an array filled with 0 values with the specified shape. The default `dtype` is `float64`.

```
>>> np.zeros((2, 3))
array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

- `ones(shape)` – creates an array filled with 1 values.
- `arange()` – like Python's built-in `range`

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([ 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.2)
array([ 2. , 2.2, 2.4, 2.6, 2.8])
```

NUMPY ARRAYS

`linspace()`– creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

```
>>> np.linspace(1., 4., 6)
array([ 1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

`random.random(shape)` – creates arrays with random floats over the interval [0,1).

```
>>> np.random.random((2,3))
array([[ 0.75688597, 0.41759916, 0.35007419],
       [ 0.77164187, 0.05869089, 0.98792864]])
```

[FSU]

NUMPY ARRAYS

Printing an array can be done with the print

- statement (Python 2)
- function (Python 3)

```
>>> import numpy as np
>>> a = np.arange(3)
>>> print(a)
[0 1 2]
>>> a
array([0, 1, 2])
>>> b = np.arange(9).reshape(3,3)
>>> print(b)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> c =
np.arange(8).reshape(2,2,2)
>>> print(c)
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

[FSU]

INDEXING

Single-dimension indexing is accomplished as usual.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

Multi-dimensional arrays support multi-dimensional indexing.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

INDEXING

Using fewer dimensions to index will result in a subarray:

```
>>> x = np.arange(10)
>>> x.shape = (2,5)
>>> x[0]
array([0, 1, 2, 3, 4])
```

This means that `x[i, j] == x[i][j]` but the second method is less efficient.

INDEXING

Slicing is possible just as it is for typical Python sequences:

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13], [21, 24, 27]])
```

ARRAY OPERATIONS

Basic operations apply element-wise. The result is a new array with the resultant elements.

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> a+b
array([0, 2, 4, 6, 8])
>>> a-b
array([0, 0, 0, 0, 0])
>>> a**2
array([ 0,  1,  4,  9, 16])
>>> a>3
array([False, False, False, False,  True], dtype=bool)
>>> 10*np.sin(a)
array([ 0., 8.41470985, 9.09297427, 1.41120008, -
7.56802495])
>>> a*b
array([ 0,  1,  4,  9, 16])
```

ARRAY OPERATIONS

Since multiplication is done element-wise, you need to specifically perform a dot product to perform matrix multiplication.

```
>>> a = np.zeros(4).reshape(2,2)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[0,0] = 1
>>> a[1,1] = 1
>>> b = np.arange(4).reshape(2,2)
>>> b
array([[0, 1],
       [2, 3]])
>>> a*b
array([[ 0.,  0.],
       [ 0.,  3.]])
>>> np.dot(a,b)
array([[ 0.,  1.],
       [ 2.,  3.]])
```

ARRAY OPERATIONS

There are also some built-in methods of `ndarray` objects.

Universal functions which may also be applied include `exp`, `sqrt`, `add`, `sin`, `cos`, etc.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.68166391, 0.98943098,
0.69361582],
       [ 0.78888081, 0.62197125,
0.40517936]])
>>> a.sum()
4.1807421388722164
>>> a.min()
0.4051793610379143
>>> a.max(axis=0)
array([ 0.78888081, 0.98943098,
0.69361582])
>>> a.min(axis=1)
array([ 0.68166391, 0.40517936])
```

ARRAY OPERATIONS

An array shape can be manipulated by a number of methods.

`resize(size)` will modify an array in place.

`reshape(size)` will return a copy of the array with a new shape.

```
>>> a =
np.floor(10*np.random.random((3,4)))
>>> print(a)
[[ 9.  8.  7.  9.]
 [ 7.  5.  9.  7.]
 [ 8.  2.  7.  5.]]
>>> a.shape
(3, 4)
>>> a.ravel()
array([ 9., 8., 7., 9., 7., 5., 9.,
7., 8., 2., 7., 5.])
>>> a.shape = (6,2)
>>> print(a)
[[ 9.  8.]
 [ 7.  9.]
 [ 7.  5.]
 [ 9.  7.]
 [ 8.  2.]
 [ 7.  5.]]
>>> a.transpose()
array([[ 9., 7., 7., 9., 8., 7.],
       [ 8., 9., 5., 7., 2., 5.]])
```

LINEAR ALGEBRA

One of the most common reasons for using the NumPy package is its linear algebra module.

It's like Matlab, but free!

```
>>> from numpy import *
>>> from numpy.linalg import *
>>> a = array([[1.0, 2.0],
               [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]
>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> inv(a) # inverse
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

LINEAR ALGEBRA

```
>>> u = eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1., 0.],
       [ 0., 1.]])
>>> j = array([[0.0, -1.0], [1.0, 0.0]])
>>> dot(j, j) # matrix product
array([[-1., 0.],
       [ 0., -1.]])
>>> trace(u) # trace (sum of elements on diagonal)
2.0
>>> y = array([[5.], [7.]])
>>> solve(a, y) # solve linear matrix equation
array([[-3.],
       [ 4.]])
>>> eig(j) # get eigenvalues/eigenvectors of matrix
(array([ 0.+1.j, 0.-1.j]),
 array([[ 0.70710678+0.j, 0.70710678+0.j],
        [ 0.00000000-0.70710678j,
0.00000000+0.70710678j]]))
```

SCIPY?

In its own words:

> SciPy is a collection of mathematical algorithms and convenience functions built on the NumPy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data.

Basically, SciPy contains various tools and functions for solving common problems in scientific computing.

SCIPY

SciPy gives you access to a ton of specialized mathematical functionality.

- **Just know it exists.** **We won't use it much in this class.**

Some functionality:

- Special mathematical functions (scipy.special) -- elliptic, bessel, etc.
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Compressed Sparse Graph Routines (scipy.sparse.csgraph)
- Spatial data structures and algorithms (scipy.spatial)
- Statistics (scipy.stats)
- Multidimensional image processing (scipy.ndimage)
- Data IO (scipy.io) – overlaps with pandas, covers some other formats

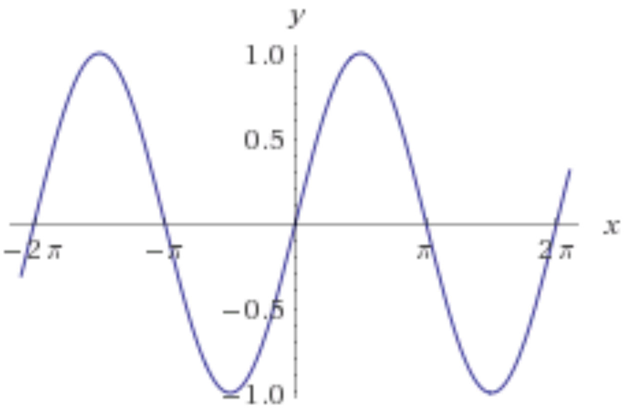
ONE SCIPY EXAMPLE

We can't possibly tour all of the SciPy library and, even if we did, it might be a little boring.

- Often, you'll be able to find higher-level modules that will work around your need to directly call low-level SciPy functions

Say you want to compute an integral:

$$\int_a^b \sin x \, dx$$



SCIPY.INTEGRATE

We have a function object – `np.sin` defines the sin function for us.

We can compute the definite integral from $x = 0$ to $x = \pi$ using the quad function.

```
>>> res = scipy.integrate.quad(np.sin, 0, np.pi)
>>> print(res)
(2.0, 2.220446049250313e-14) # 2 with a very small error
margin!
>>> res = scipy.integrate.quad(np.sin, -np.inf, +np.inf)
>>> print(res)
(0.0, 0.0) # Integral does not converge
```

SCIPY.INTEGRATE

Let's say that we don't have a function object, we only have some (*x*,*y*) samples that "define" our function.

We can estimate the integral using the trapezoidal rule.

```
>>> sample_x = np.linspace(0, np.pi, 1000)
>>> sample_y = np.sin(sample_x) # Creating 1,000 samples
>>> result = scipy.integrate.trapz(sample_y, sample_x)
>>> print(result)
1.99999835177

>>> sample_x = np.linspace(0, np.pi, 1000000)
>>> sample_y = np.sin(sample_x) # Creating 1,000,000 samples
>>> result = scipy.integrate.trapz(sample_y, sample_x)
>>> print(result)
2.0
```

WRAP UP: FIRST PART

Shift thinking from imperative coding to operations on datasets

Numpy: A low-level abstraction that gives us really fast multi-dimensional arrays

Next class:

Pandas: Higher-level tabular abstraction and operations to manipulate and combine tables

Reading Homework focuses on Pandas and SQL