# Getting Data

## Lecture 9

Centre for Data Science, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

# Contents

# Introduction

- In order to be a data scientist we need data.
- In fact, data scientist spend a large fraction of time:
  1. Acquiring data
  2. Cleaning data
  3. Transforming data
- In this chapter, we'll look at different ways of **getting data** into Python and into the **right formats**.

# stdin and stdout

- **sys.stdin**: Used to accept input from command line directly.
- **sys.stdout**: Used to display output directly to the screen console.
- Python scripts uses these commands to access data at the command line.
- **Example 1**: A script that reads in lines of text and output the ones that match a regular expression:

```
1  # egrep.py
2  import sys, re
3
4  # sys.argv is the list of command-line arguments
5  # sys.argv[0] is the name of the program itself
6  # sys.argv[1] will be the regex specified at the command line
7  regex = sys.argv[1]
8
9  # for every line passed into the script
10 for line in sys.stdin:
11     # if it matches the regex, write it to stdout
12     if re.search(regex, line):
13         sys.stdout.write(line)
```

# stdin and stdout (Contd.)

- **Example 2**: Python program to counts the lines it receives and then writes out the count:

```
1 # line_count.py
2 import sys
3
4 count = 0
5 for line in sys.stdin:
6     count += 1
7
8 # print goes to sys.stdout
9 print(count)
```

# stdin and stdout (Contd.)

- **Example 3**: Read input and count how many lines of a file contain numbers.
- In Windows, we'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

- In a Unix system we'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

- The | is the pipe character, which means use the output of the left command as the input of the right command.

# stdin and stdout (Contd.)

- **Example 4**: A script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1) # nonzero exit code indicates error

# strip(): Remove spaces at the beginning and end of the string
counter = Counter(word.lower() # lowercase words
                  for line in sys.stdin
                  for word in line.strip().split()# split on spaces
                  if word) # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

# stdin and stdout (Contd.)

- After which we could do something like:

```
$ cat the_bible.txt | python most_common_words.py 10
36397    the
30031    and
20163    of
7154     to
6484     in
5856     that
5421     he
5226     his
5060     unto
4297     shall
```

- (If you are using Windows, then use *type* instead of *cat*.)

# Reading files

- **File**: It is some information or data which stays in the computer storage devices.
- **Examples**: music files, video files, text files.
- We can also explicitly read from and write to files directly in our code.
- Python makes working with files pretty simple.

# The Basics of Text Files

- The first step to working with a text file is to obtain a file object using **open**:

```
1  # 'r' means read-only, it's assumed if you leave it out
2  file_for_reading = open('reading_file.txt', 'r')
3  file_for_reading2 = open('reading_file.txt')
4
5  # 'w' is write -- will destroy the file if it already exists!
6  file_for_writing = open('writing_file.txt', 'w')
7
8  # 'a' is append -- for adding to the end of the file
9  file_for_appending = open('appending_file.txt', 'a')
10
11 # don't forget to close your files when you're done
12 file_for_writing.close()
```

# The Basics of Text Files (Contd.)

- Because it is easy to forget to close our files, we should always use them in a **with** block, at the end of which they will be closed automatically:

```
with open(filename) as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

- If we need to read a whole text file, we can just iterate over the lines of the file using **for** loop:

```
starts_with_hash = 0

with open('input.txt') as f:
    for line in f: # look at each line in the file
        if re.match("^#",line): # use a regex to see if it starts with '#'
            starts_with_hash += 1 # if it does, add 1 to the count
```

# The Basics of Text Files (Contd.)

- **Example**: Imagine we have a file full of email addresses, one per line, and we need to generate a histogram of the domains.

```
def get_domain(email_address: str) -> str:
"""Split on '@' and return the last piece"""
return email_address.lower().split("@")[-1]

# a couple of tests
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.
  datasciencester.com'

from collections import Counter

with open('email_addresses.txt', 'r') as f:
domain_counts = Counter(get_domain(line.strip())
for line in f
if "@" in line)

```

# Delimited Files

- These files are very often:
  - *comma-separated*
  - *tab-separated*

  where, a comma or a tab indicating where one field ends and the next field starts in a line.
- Python libraries to read comma-separated or tab-delimited files:
  - Python's csv module
  - Pandas
- **csv.reader**: This is used to iterate over the rows of a file that has no header.

# Delimited Files (Contd.)

- For example, if we had a tab-delimited file of stock prices:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

- We could process them with:

```
1 import csv
2
3 with open('tab_delimited_stock_prices.txt') as f:
4   tab_reader = csv.reader(f, delimiter='\t')
5   for row in tab_reader:
6     date = row[0]
7     symbol = row[1]
8     closing_price = float(row[2])
9     process(date, symbol, closing_price)
```

# Delimited Files (Contd.)

- If our file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

- We can skip the header row with an initial call to reader.next
- Or get each row as a dict (with the headers as keys) by using *csv.DictReader*:

```
1 with open('colon_delimited_stock_prices.txt') as f:
2   colon_reader = csv.DictReader(f, delimiter=':')
3   for dict_row in colon_reader:
4     date = dict_row["date"]
5     symbol = dict_row["symbol"]
6     closing_price = float(dict_row["closing_price"])
7     process(date, symbol, closing_price)
```

- Even if file doesn't have headers, we can still use *DictReader* by passing it the keys as a *fieldnames* parameter.

# Delimited Files (Contd.)

- We can similarly write out delimited data using **csv.writer**

```
1 todays_prices = {'AAPL': 90.91, 'MSFT': 41.68, 'FB': 64.5 }
2
3 with open('comma_delimited_stock_prices.txt', 'w') as f:
4     csv_writer = csv.writer(f, delimiter=',')
5     for stock, price in todays_prices.items():
6         csv_writer.writerow([stock, price])
```

- *csv.writer* will do the right thing if fields themselves have commas.

```
results = [["test1", "success", "Monday"],
["test2", "success, kind of", "Tuesday"],
["test3", "failure, kind of", "Wednesday"],
["test4", "failure, utter", "Thursday"]]
```

# Scraping the Web

- Web scraping is used to extract information from web pages.
- Fetching web pages is pretty easy; but getting meaningful structured information out of them is somewhat difficult.

# HTML and the Parsing Thereof

- HTML is popular language for writing the web pages.
- Text is marked up into elements and their attributes:

```
<html>
    <head>
        <title>A web page</title>
    </head>
    <body>
        <p id="author">Joel Grus</p>
        <p id="subject">Data Science</p>
    </body>
</html>
```

# HTML and the Parsing Thereof (Contd.)

- **Beautiful Soup library**:
  - This library is used to extract data from HTML pages.
  - This produces a parse tree out of the various elements on a web page and provides a simple interface for accessing them.
- **Requests library**: This is used for making HTTP requests.
- **html5lib library**: Python's built-in HTML parser is not that lenient and doesn't always cope well with HTML that's not perfectly formed.
  - For that reason, we'll also install the *html5lib* parser.

```
1  python -m pip install beautifulsoup4 requests html5lib
2
```

# HTML and the Parsing Thereof (Contd.)

- *requests.get*: This produces a string containing the HTML.
- *BeautifulSoup* function produces the parse tree.

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  # I put the relevant HTML file on GitHub. In order to fit
5  # the URL in the book I had to split it across two lines.
6  # Recall that whitespace-separated strings get concatenated.
7  url = ("https://raw.githubusercontent.com/"
8  "joelgrus/data/master/getting-data.html")
9  html = requests.get(url).text
10 soup = BeautifulSoup(html, 'html5lib')
11
```

- We'll typically work with Tag objects, which correspond to the tags representing the structure of an HTML page.

# HTML and the Parsing Thereof (Contd.)

- Beautiful Soup **find()** method:
  - This matches for a first matching tag and return a Tag object.
- Example: To find the first $< p >$ tag (and its contents):

```
first_paragraph = soup.find('p') # or
first_paragraph = soup.p
```

*output*: Joel Grus

- We can get the text contents of a Tag using its text property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

*output*: Joel Grus *output*: ['Joel', 'Grus']

# HTML and the Parsing Thereof (Contd.)

- And we can extract a tag's attributes by treating it like a dict:

```
first_paragraph_id = soup.p['id'] # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id') # returns None if no 'id'
```

- Beautiful Soup **find_all()** method: We can get multiple tags at once as follows:

```
all_paragraphs = soup.find_all('p') # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

- Frequently, we'll want to find tags with a specific class:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
if 'important' in p.get('class', [])]
```

# HTML and the Parsing Thereof (Contd.)

- These methods can be combined to implement more elaborate logic.
- For example, if we want to find every $<$ *span* $>$ element that is contained inside a $<$ *div* $>$ element, we could do this:

```
# Warning: will return the same <span> multiple times
# if it sits inside multiple <div>s.
# Be more clever if that's the case.
spans_inside_divs = [span
for div in soup('div') # for each <div> on the page
for span in div('span')] # find each <span> inside it
```

- Just this handful of features will allow us to do quite a lot.

# Example: Keeping Tabs on Congress

- The VP of Policy at DataSciencester is worried about potential regulation of the data science industry and asks you to quantify what Congress is saying on the topic.
- In particular, he wants you to find all the representatives who have press releases about "data."
- At the time of publication, there is a page with links to all of the representatives' websites at *https://www.house.gov/representatives*.
- And if you "view source," all of the links to the websites look like:

```
<td>
    <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

# Example: Keeping Tabs on Congress (Contd.)

- Let's start by collecting all of the URLs linked to from that page:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")

all_urls = [a['href']
    for a in soup('a')
    if a.has_attr('href')]

print(len(all_urls)) # 965 for me, way too many
```

- This returns way too many URLs. If we look at them, the ones we want start with either *http://* or *https://*, have some kind of name, and end with either *.house.gov* or *.house.gov/*.

# Example: Keeping Tabs on Congress (Contd.)

- This is a good place to use a regular expression:

```
1  import re
2
3  # Must start with http:// or https://
4  # Must end with .house.gov or .house.gov/
5  regex = r"^https?://.*\.house\.gov/?$"
6
7  # Let's write some tests!
8  assert re.match(regex, "http://joel.house.gov")
9  assert re.match(regex, "https://joel.house.gov")
10 assert re.match(regex, "http://joel.house.gov/")
11 assert re.match(regex, "https://joel.house.gov/")
12 assert not re.match(regex, "joel.house.gov")
13 assert not re.match(regex, "http://joel.house.com")
14 assert not re.match(regex, "https://joel.house.gov/biography")
15
16 # And now apply
17 good_urls = [url for url in all_urls if re.match(regex, url)]
18
19 print(len(good_urls)) # still 862 for me
```

# Example: Keeping Tabs on Congress (Contd.)

- That's still way too many, as there are only 435 representatives.
- If we look at the list, there are a lot of duplicates.
- We can get rid of them by using set.

```
1 good_urls = list(set(good_urls))
2 print(len(good_urls)) # only 431 for me
```

# Example: Keeping Tabs on Congress (Contd.)

- There are always a couple of House seats empty, or maybe there's a representative without a website.
- When we look at the sites, most of them have a link to press releases.
- For example:

```
1 html = requests.get('https://jayapal.house.gov').text
2 soup = BeautifulSoup(html, 'html5lib')
3
4 # Use a set because the links might appear multiple times.
5 links = {a['href'] for a in soup('a') if 'press releases' in a.
      text.lower()}
6 print(links) # {'/media/press-releases'}
```

# Example: Keeping Tabs on Congress (Contd.)

- This is a relative link, which means we need to remember the originating site.
- For scraping we write:

```
from typing import Dict, Set

press_releases: Dict[str, Set[str]] = {}

for house_url in good_urls:
    html = requests.get(house_url).text
    soup = BeautifulSoup(html, 'html5lib')
    pr_links = {a['href'] for a in soup('a') if 'press releases'
                    in a.text.lower()}
    print(f"{house_url}: {pr_links}")
    press_releases[house_url] = pr_links
```

# Example: Keeping Tabs on Congress (Contd.)

- If we watch these as they scroll by, we'll see a lot of */media/press-releases* and *media-center/press-releases*, as well as various other addresses.
- One of these URLs is `https://jayapal.house.gov/media/press-releases`.
- Our goal is to find out which congress people have press releases mentioning "data."
- We'll write a slightly more general function that checks whether a page of press releases mentions any given term.

# Example: Keeping Tabs on Congress (Contd.)

- If you visit the site and view the source, it seems like there's a snippet from each press release inside a $< p >$ tag, so we'll use that as our first attempt:

```
def paragraph_mentions(text: str, keyword: str) -> bool:
    """
    Returns True if a <p> inside the text mentions {keyword}
    """
    soup = BeautifulSoup(text, 'html5lib')
    paragraphs = [p.get_text() for p in soup('p')]
    return any(keyword.lower() in paragraph.lower()
        for paragraph in paragraphs)
```

- Let's write a quick test for it:

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
assert paragraph_mentions(text, "twitter") # is inside a <p>
assert not paragraph_mentions(text, "facebook") # not inside a <p>
```

# Example: Keeping Tabs on Congress (Contd.)

- At last we're ready to find the relevant congress people and give their names to the VP:

```
for house_url, pr_links in press_releases.items():
    for pr_link in pr_links:
        url = f"{house_url}/{pr_link}"
        text = requests.get(url).text
        if paragraph_mentions(text, 'data'):
            print(f"{house_url}")
                break # done with this house_url
```

- After running this we get a list of representatives.

# Using APIs

- API is the acronym for '*Application Programming Interface*", which is a intermediary software that allows two applications to talk to each other.
- Many websites and web services provide APIs, which allow to explicitly request data in a structured format.
- This saves us the trouble of having to scrape them.

# JSON and XML

- HTTP protocol is used for transferring text, therefore, requested data through a web API needs to be serialized into a string format.
- Popular serialization format:
  - *JavaScript Object Notation* (JSON)
  - *Extensible Markup Language* (XML)
- JSON objects look quite similar to Python dicts, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2019,
  "topics" : [ "data", "science", "data science"] }
```

- We can parse JSON using Python's json module.

# JSON and XML (Contd.)

- **loads** function: This deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
        "author" : "Joel Grus",
        "publicationYear" : 2019,
        "topics" : [ "data", "science", "data science"] }"""

# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
assert deserialized["publicationYear"] == 2019
assert "data science" in deserialized["topics"]
```

# JSON and XML (Contd.)

- Sometimes an API provider provides responses in XML:

```
<Book>
    <Title>Data Science Book</Title>
    <Author>Joel Grus</Author>
    <PublicationYear>2014</PublicationYear>
    <Topics>
        <Topic>data</Topic>
        <Topic>science</Topic>
        <Topic>data science</Topic>
    </Topics>
</Book>
```

- We can use Beautiful Soup to get data from XML similarly to how we used it to get data from HTML

# Using an Unauthenticated API

- Most APIs these days require first client's authentication before using them.
- Start by taking a look at **GitHub's API**, with which we can do some simple things unauthenticated:

```
1 import requests, json
2
3 github_user = "joelgrus"
4 endpoint = f"https://api.github.com/users/{github_user}/repos"
5
6 repos = json.loads(requests.get(endpoint).text)
```

- *repos* is a list of Python *dicts*, each representing a public repository in *joelgrus*'s GitHub account.

# Using an Unauthenticated API (Contd.)

- The *repos* can be used to figure out which months and days of the week we are most likely to create a repository.
- The only issue is that the dates in the response are strings:

```
"created_at": "2013-07-05T02:02:28Z"
```

- To parse date, we'll install one date parser:

```
python -m pip install python-dateutil
```

- Example:

```
from collections import Counter
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

# Using an Unauthenticated API (Contd.)

- Similarly, we can get the languages of last five repositories:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["pushed_at"],
                             reverse=True)[:5]
last_5_languages = [repo["language"]
                             for repo in last_5_repositories]
```

- The benefits of using Python is that there is already a built library for pretty much any API we're interested in accessing.

# Finding APIs

- If we need data from a specific site, look for a “developers” or “API” section of the site for details, and try searching the web for “python <sitename> api” to find a library.
- List of APIs that have Python wrappers can be obtained from **Real Python on GitHub**.
- And if we can’t find what we need, scraping is the solution.

# Example: Using the Twitter APIs

- Twitter is a popular source for collecting information:
  - It can be use to get real-time news.
  - For measuring reactions/sentiment to current events.
  - Finding links related to specific topics.
- Twitter data can be accessed through its APIs.
- **Twython** is a popular python library used to interact with the Twitter APIs.
- Installing *twython* library:

  *python -m pip install twython*

# Getting Credentials

- Interaction with Twitter's APIs require some credentials.
  - This requires a Twitter account.
- Steps for getting the credentials:
  1. Go to `https://developer.twitter.com/`.
  2. "**Sign in**" to your Twitter account.
  3. Click Apply for a developer account.
  4. Request access for your own personal use.
  5. Fill out the application. It requires 300 words on why you need access.
  6. Once you get approved, go back to **developer.twitter.com**, find the "Apps" section, and click "Create an app."
  7. Fill out all the required fields.
  8. Click CREATE.
- Goto "**Keys and tokens**" tab —> "**Consumer API keys**" section:
  - *API key*
  - *API secret key*

# Using Twython

- The first step to use the Twitter API is authenticate yourself.
  - API providers want to know who's accessing their data.
- OAuth (**O**pen **Auth**orization):
  - Authorization protocol used to grant access to the secured information without giving the credentials.
  - Twitter, Amazon, Facebook use OAuth to grant $3^{rd}$ party access to protected information.
- *OAuth 1.0*: Required to perform actions (e.g., tweeting) or connect to the Twitter stream.
- *OAuth 2.0*: Used for doing simple searches.

# Using Twython (Contd.)

- First, we need API key and API secret key (also known as the "*consumer key*" and "*consumer secret*", respectively).
- We can get it from environment variables.

```
1 import os
2
3 # Feel free to plug your key and secret in directly
4 CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY")
5 CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET")
```

# Using Twython (Contd.)

```
1  import webbrowser
2  from twython import Twython
3
4  # Get a temporary client to retrieve an authentication URL
5  temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
6  temp_creds = temp_client.get_authentication_tokens()
7  url = temp_creds['auth_url']
8
9  # Now visit that URL to authorize the application and get a PIN
10 print(f"go visit {url} and get the PIN code and paste it below")
11 webbrowser.open(url)
12 PIN_CODE = input("please enter the PIN code: ")
13
14 # Now we use that PIN_CODE to get the actual tokens
15 auth_client = Twython(CONSUMER_KEY, CONSUMER_SECRET,
16                       temp_creds['oauth_token'],
17                       temp_creds['oauth_token_secret'])
18 final_step = auth_client.get_authorized_tokens(PIN_CODE)
19 ACCESS_TOKEN = final_step['oauth_token']
20 ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']
```

# Using Twython (Contd.)

```
# And get a new Twython instance using them.
twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET,
ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

```

- Once we have an authenticated *Twython* instance, we can start performing searches using Twitter Search API:

```
# Search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

- Twitter Search API uses "search criteria" that can be keywords, user-names, locations, etc.

# Using Twython (Contd.)

- If we run this, we should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp;amp;
    analytical chops to derive meaning from big data are in demand
    . http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated
    assignment for @rdpeng in @coursera data science
    specialization. So easy and Awesome.
```

- The Twitter Search API only guarantees a handful of recent results limited in numbers.
- It delivers data in batches.
- More results require repeated requests.

# Using Twython (Contd.): **Streaming API**

- **Streaming API**: This is useful collecting tweets in the real-time scenario by opening connection between the *client app* and the *API*.
- **Twitter firehose** is a streaming API that provide results in real-time.
- Streaming API require *OAuth 1.0* authentication credentials, i.e., access tokens.
- A class definition that inherits from *TwythonStreamer* is required to access the Streaming API with Twython. This must override the *on_success* method, and possibly its *on_error* method to manage the received tweets.

# Using Twython (Contd.)

```
1  from twython import TwythonStreamer
2
3  # Appending data to a global variable is pretty poor form
4  # but it makes the example much simpler
5  tweets = []
6
7  class MyStreamer(TwythonStreamer):
8      def on_success(self, data):
9          """
10         What do we do when Twitter sends us data?
11         Here data will be a Python dict representing a tweet.
12         """
13         # We only want to collect English-language tweets
14         if data.get('lang') == 'en':
15             tweets.append(data)
16             print(f"received tweet #{len(tweets)}")
17         # Stop when we have collected enough
18         if len(tweets) >= 100:
19             self.disconnect()
20
21     def on_error(self, status_code, data):
22         print(status_code, data)
23         self.disconnect()
```

# Using Twython (Contd.)

- *MyStreamer* will connect to the Twitter stream and wait for Twitter to feed it data.
- Each time it receives some data (here, a tweet represented as a Python object), it passes it to the *on_success* method, which appends it to our tweets list if its language is English, and then disconnects the streamer after it's collected 100 tweets.

```
1  stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
2           ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
3  """
4  starts consuming public statuses that contain the keyword 'data'
5  """
6  stream.statuses.filter(track='data')
7  """
8  if instead we wanted to start consuming a sample of *all* public
       statuses
9  stream.statuses.sample()
10 """
```

# Using Twython (Contd.)

- This will run until it collects 100 tweets (or until it encounters an error) and stop, at which point we can start analyzing those tweets.
- For instance, we could find the most common hashtags with:

```
top_hashtags = Counter(hashtag['text'].lower()
        for tweet in tweets
        for hashtag in tweet["entities"]["hashtags"])
print(top_hashtags.most_common(5))
```

- Each tweet contains a lot of data. Which We can dig through the **Twitter API documentation**.

# For Further Exploration

- **pandas** is the primary library that data science types use for working with—and, in particular, importing—data.
- **Scrapy** is a full-featured library for building complicated web scrapers that do things like follow unknown links.
- **Kaggle** hosts a large collection of datasets.

Thank You
Any Questions?