Getting Data

Data Science

Getting Data

- As a data scientist you will spend a large fraction of your time acquiring, cleaning and transforming data.
- Typing data in is not a good use of your time so here we will look at different ways of getting data into Python and into the right format.

stdin and stdout

```
# egrep.py
import sys, re

if __name__ == "__main__":

    # sys.argv is the list of command-line arguments
    # sys.argv[0] is the name of the program itself
    # sys.argv[1] will be the regex specfied at the command line
    regex = sys.argv[1]

    # for every line passed into the script
    for line in sys.stdin:
        # if it matches the regex, write it to stdout
        if re.search(regex, line):
            sys.stdout.write(line)
```

stdin and stdout

```
# line_count.py
import sys

if __name__ == "__main__":

    count = 0
    for line in sys.stdin:
        count += 1

    # print goes to sys.stdout
    print(count)
```

stdin and stdout

- You can use these two programs to count how many lines of a file contain numbers.
- In Windows, you would use:
- type SomeFile.txt | python egrep.py “[0-9]” | python line_count.py
- In Linux you would use:
- cat SomeFile.txt | python egrep.py “[0-9]” | python line_count.py

stdin and stdout

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

if __name__ == "__main__":

    # 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)   # non-zero exit code indicates error
```

stdin and stdout

```
counter = Counter(word.lower()
                  for line in sys.stdin
                  for word in line.strip().split()
                  if word)

    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")
```

Reading Files

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

```
# ‘r’ means read only
file_for_reading = open(‘reading_file.txt’, ‘r’)
# ‘w’ is write-will destroy the file if it exists
file_for_writing = open(‘writing_file.txt’, ‘w’)
# ‘a’ means append – add to end of file
file_for_appending = open(‘appending_file.txt’, ‘a’)
# close the file when you are done
file_for_writing.close()
```

The Basics of Text Files

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

```
with open(filename, ‘r’) as f:
   data = function_that_gets_data_from(f)
# here f is closed
process(data)
```

The Basics of Text Files

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

```
starts_with_hash = 0
with open('input.txt', 'r') as f:
    for line in f:
        if re.match("^#", line):
            starts_with_hash += 1
print(starts_with_hash)
```

The Basics of Text Files

- Imagine you have a file full of email addresses, one per line, and you need to generate a histogram of domains.
- A good first approximation for the domains is to take the part of the email address after the @.

The Basics of Text Files

```
def get_domain(email_address):
   #split on @ and take value after @
   return email_address.lower().split("@")[-1]

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

Delimited Files

- The previous email address file had one address per line.
- More frequently we will have lots of data on each line either comma separated or tab separated.
- Each line has several fields, with a comma indicating where one field ends and the next field starts.

Delimited Files

- You should always work with csv files in binary mode by including a b after the r or w.
- If your file has no headers, you can use csv.reader to iterate over rows.
- If we had a tab delimited file of stock prices:

Delimited Files

6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.85
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34

Delimited Files

```
import csv
with open('tab_delimited_stock_prices.txt', 'rt') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing price = float(row[2])
        print(date, symbol, closing_price)
```

Delimited Files

- If your file has headers you can either 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.

Delimited Files

```
import csv
with open('colon_delimted_stock_prices.txt', 'rt')
as f:
   reader = csv.DictReader(f, delimiter=':')
   for row in reader:
      date = row["date"]
      symbol = row["symbol"]
      closing price = float(row["closing_price"])
       process(date, symbol, closing_price)
```

Delimited Files

- Even if your file does not have headers you can still use DictReader by passing in the keys as a fieldnames parameter.
- You can write out delimited data using csv.writer:

Delimited Files

```
today_prices = {‘AAPL’ : 90.91, ‘MSFT’ : 41.68, ‘FB’ : 64.5}
with open(‘comma_delimted_stock_prices.txt’, ‘wt’) as f:
    writer = csv.writer(f, delimiter=‘,’)
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

Scraping the Web

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

Scraping the Web

- To get data out of HTML, we will use the BeautifulSoup library, which builds a tree out of the various elements of a web page and provides a simple interface for accessing them.
- Use pip install html5lib to use a parser that copes well with html that is not perfectly formed.

Scraping the Web

- To use BeautifulSoup we will need to pass some HTML into the BeautifulSoup() function.
- In our examples this will be the result of a call to requests.get:

Scraping the Web

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://
www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

- We will typically work with the Tag objects which are the tags representing the structure of the HTML page.

Scraping the Web

- To find the first <p> tag and its contents you can use:

```
first_paragraph = soup.find('p')
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

Scraping the Web

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

```
first_paragraph_id = soup.p['id']
first_paragraph_id2 = soup.p.get('id')
```

- You can get multiple tags at once:

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

Scraping the Web

- Frequently you will 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', [])]

Scraping the Web

- If you want to find every element that is contained inside a you can do:

```
spans_inside_divs = [span
                     for div in soup('div')
                     for span in div('span')]
```

- You will need to carefully inspect the source HTML and reason about your selection logic to get the data you need from a web page.

Example: O'Reilly Books about Data

- O'Reilly's data science page has many data books and videos, reachable through 30-items-at-a-time directory pages with URLs like:
- http://shop.oreilly.com/category/browse-subjects/data.do?soertby=publicationDate&page=1

Example: O'Reilly Books about Data

- Whenever you want to scrape data from a website you should first check to see if its has some sort of access policy.
- Looking at http://oreilly.com/terms/ there seems to be no problem with scraping.
- You should also check for a file called robots.txt that tells webcrawlers how to behave.

Example: O'Reilly Books about Data

- The important lines from http://shop.oreilly.com/robots.txt are:
- Crawl-delay: 30
- Request-rate: 1/30
- Both say the same thing – that you should only make one page request every 30 seconds.

Example: O'Reilly Books about Data

- Let's download one of the pages and feed it to BeautifulSoup.

url = http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page=1

soup = BeautifulSoup(requests.get(url).text, 'html5lib')

Example: O'Reilly Books about Data

- If you view the page source you should see that each book (or video) is contained in a <td> table cell whose class is thumbtext.
- A good first step is to find all of the td thumbtext tag elements.

```
tds = soup('td', 'thumbtext')
print(len(tds))
```

Example: O'Reilly Books about Data

- Next we would like to filter out videos.
- If we look at the html again we see that each td contains one or more span elements whose class is pricelabel, and whose text looks like Ebook: or Video: or Print:.
- It looks like videos contains only one pricelabel, whose text starts with Video.
- We can test for videos with:

Example: O’Reilly Books about Data

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if the stripped text inside that pricelabel starts with 'Video'"""
    pricelabels = td('span', 'pricelabel')
    return (len(pricelabels) == 1 and pricelabels[0].text.strip().startswith("Video"))
print(len([td for td in tds if not is_video(td)]))
```

Example: O'Reilly Books about Data

- It looks like the book title is the text inside the <a> tag inside the :

title = td.find("div", "thumbheader").a.text

- The authors are in the text of the AuthorName . They are prefaced by a By and separated by commas.

Example: O'Reilly Books about Data

author_name = td.find('div', 'AuthorName').text

authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]

- The ISBN is contained in a link that is in the thumbheader :

isbn_link = td.find("div", "thumbheader").a.get("href")

isbn = re.match("/product/(.*)\.do", isbn_link).group(1)

Example: O'Reilly Books about Data

- The date is just the contents of the

```
date = td.find("span", "directorydate").text.strip()
```

Example: O'Reilly Books about Data

```
def book_info(td):
    """given a BeautifulSoup <td> Tag representing a book,
    extract the book's details and return a dict"""

    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
    authors = [td.strip() for td in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
        "title" : title,
        "authors" : authors,
        "isbn" : isbn,
        "date" : date
    }
```

Example: O'Reilly Books about Data

```
def scrape(num_pages=39):
    base_url = "http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page="

    books = []

    for page_num in range(1, num_pages + 1):
        print("souping page", page_num)
        url = base_url + str(page_num)
        soup = BeautifulSoup(requests.get(url).text, 'html5lib')

        for td in soup('td', 'thumbtext'):
            if not is_video(td):
                books.append(book_info(td))

        # now be a good citizen and respect the robots.txt!
        sleep(30)

    return books
```

Example: O’Reilly Books about Data

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
    return int(book["date"].split()[1])
```

Example: O'Reilly Books about Data

```
def plot_years(plt, books):
    # 2015 is the last complete year of data (when I ran this)
    year_counts = Counter(get_year(book) for book in books
                          if get_year(book) <= 2015)

    years = sorted(year_counts)
    book_counts = [year_counts[year] for year in years]
    plt.bar([x - 0.5 for x in years], book_counts)
    plt.xlabel("year")
    plt.ylabel("# of data books")
    plt.title("Data is Big!")
    plt.show()
```

Using APIs to get Data

- Because HTTP is a protocol for transferring text, the data you request through a web API needs to be serialized into a string format.
- Often this serialization uses JavaScript Object Notation (JSON).

{ “title” : “Data Science Book”,
“author”: “Joel Grus”,
“publicationYear : 2014,
“topics” : [“data”, “science”, “data science”]}

Using APIs to get Data

- We can parse JSON using Pythons json module.
- We will use its loads function, which deserializes a string representing a JSON object into a Python object:

Using APIs to get Data

```
import json
serialized = """{ "title" : "Data Science Book",
   "author": "Joel Grus",
   "publicationYear : 2014,
   "topics" : ["data", "science", "data science"]}"""
#parse the json to create a Python dict
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
   print(deserialized)
```

Using APIs to get Data

```
<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>
```

Using an Unauthenticated API

- Most APIs nowadays require you to first authenticate yourself in order to use them.
- We will first have a look at GitHubs API, with which you can do some simple things unauthenticated.

```
import requests, json
endpoint = https://api.github.com/users/joelgrus/repos
repos = json.loads(requests.get(endpoint).text)
```

Using an Unauthenticated API

- Repos is a list of Python dicts, each representing a public repository in a Github account.
- We can use this to figure out which months and days of the week a repository is most likely to be created.
- Python doesn't come with a great date parser so we will install one.
- pip install python-dateutil

Using an Unauthenticated API

```
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

- Similarly we can get the languages of the last 5 repositories:

last_5_repositories = sorted(repos, key = lambda r : r[“created_at”], reverese = True)[:5]

last_5_languages = [repo[“language”] for repo in last_5_repositories]

Finding APIs

- If you need data for a specific site, look for a developers or API section of the site for details and try searching for "python_api" to find a library.
- There is a Rotten Tomatoes API for Python and many more http://www.pythonforbeginners.com/development/list-of-python-apis/

Example: Using the Twitter APIs

- To interact with the Twitter API we will use the Twython library (pip install twython).
- In order to use Twitter's APIs, you need to get some credentials for which you need a Twitter account.

1. Go to https://apps.twitter.com/
2. If you are not signed in, click sign in and enter your Twitter username and password.

Example: Using the Twitter APIs

3. Click Create New App
4. Give it a name, a description and put any url as the website.
5. Agree to the Terms of Service and click Create.
6. Take note of the consumer key and consumer secret.
7. Go to the Keys and Access Tokens tab and click Create my access token.
8. Take note of the access token and the access token secret.

Example: Using the Twitter APIs

- The consumer key and consumer secret tell Twitter what application is accessing its APIs, while the access token and access token secret tell Twitter who is accessing its APIs.
- The consumer key/secret and the access token/secret should be treated like passwords.
- You should not share them.

Using Twython

- First we will look at the Search API, which requires only the consumer key and the secret, not the access token or secret.

```
from twython import Twython
# fill these in if you want to use the code
CONSUMER_KEY = ""
CONSUMER_SECRET = ""
ACCESS_TOKEN = ""
ACCESS_TOKEN_SECRET = ""
```

Using Twython

```
def call_twitter_search_api():
    twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)
    # 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(user, ":", text)
        print()
```

Using Twython

- The Twitter search API just shows you whatever handful of recent results it feels like.
- More often you will want a lot of tweets.
- This is where the Streaming API is useful.
- It allows you to connect to lots of data from Twitter.
- In order to use it you will have to authenticate using your access tokens.

Using Twython

- In order to access the Streaming API with Twython, we need to define a class that inherits from TwythonStreamer and overrides its on_success and on_error methods:

```
from twython import TwythonStreamer

tweets = []
```

Using Twython

```
class MyStreamer(TwythonStreamer):
    """our own subclass of TwythonStreamer that specifies
    how to interact with the stream"""

    def on_success(self, data):
        """what do we do when twitter sends us data?
        here data will be a Python object representing a tweet"""

        # only want to collect English-language tweets
        if data['lang'] == 'en':
            tweets.append(data)
            print("received tweet #", len(tweets))

        # stop when we've collected enough
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
        print(status_code, data)
        self.disconnect()
```

Using Twython

```
def call_twitter_streaming_api():
    stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                        ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
    # starts consuming public statuses that contain the keyword 'data'
    stream.statuses.filter(track='data')
```

Using Twython

- Once we have the 1000 tweets we can analyze them.

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