Your Cover Here

atlas book skeleton

Author Name

atlas book skeleton

by Author Name

If your book has an ISBN or a book ID number, add it here as well.

Table of Contents

Preface Title **v**

CHAPTER 1: Tools For Social Justice in Python **7**

Organization of the text 7

Introduction 7

Section 1 7

Section 2 7

Section 3 7

Introduction 8

The intention of this text/thank you 8

Prerequisite Knowledge 9

A quick note on installation 10

CHAPTER 2: Section 1 - Working With Data For Non-Profits And Government **11**

Web Scraping 12

Installation 12

Mac OSX or Ubuntu 12

Windows 12

Getting started with requests 12

Working With GETs And POSTs 13

Parsing HTML 15

Crawling the web 16

Turning PDFs into CSVs 19
Installation 20
Data Cleaning techniques 23

CHAPTER 3: Section 2 - Data Visualization 27
Design Process 27
Design Examples 29
Installation 29
Building our first flask app 29
C3.js 35
C3.js and flask 35
Free GIS Systems, in Python - working with leaflet and flask 36
Our API 41
Section 3 44
Facial recognition, facial comparison, and image search 45
Understanding Face Comparison, With OpenCV 51
Multithreading applications 53
What's all this then? 53
Conclusion 57

Appendix Title 59

Index 61

Preface Title

This Is an A-Head

Congratulations on starting your new project! We've added some skeleton files for you, to help you get started, but you can delete any or all of them, as you like. In the file called chapter.html, we've added some placeholder content showing you how to markup and use some basic book elements, like notes, sidebars, and figures.

Tools For Social Justice in Python

1

By *Eric Schles*

Organization of the text

Introduction

- The intention of the text

Section 1

working with data for non-profits and government

- web scraping
- PDF to csv

Section 2

Visualizing data for analysts and to find patterns and creating interfaces

- web dev in flask, bootstrap, and angular.js - easy
- gis systems for free - easy
- c3

Section 3

Searching across massive data sets

- facial recognition and facial comparison - building a cbir - easy
- searching across text in mass - easy

Introduction

The intention of this text/thank you

The intention of this report is to explain the interested reader how to use computer science to solve social problems. To build systems and change minds and hearts. Often times those interested in social justice are not technologists and so they are unaware of what technology can do to make their work easier and more effective.

This text would not be possible without my mentors, friends, family, and collaborators. So to my parents Marc and Andrea, I say thank you for embuing me with a social compass, and for believing a life of service is the most fulfilling one. To my friends, I say thank you, you are far to numerous to name, but you know who you are. If not for you, I would have never made it through school or life, you are oxygen, and I hope you all know that. To my mentors, Ethan Mann, James Javor, Ellen Eurbanek, the late Samuel Schack, John Temple, and Andrew Raiseji you have my deepest thanks. If not for your guidance, and support I would have never found the direction that has led me to this point. It is my only hope that I can be half the mentor that each of you has been to me. And finally to my collaborators, I say thank you for your partnership and fraternity in this fight for a better tomorrow. I will name each of them throughout the text and so their names and organizations are merely listed here.

- Freddie Andrade wikitongues
- Daniel wikitongues
- Miguel Code in mexico - fix this (get spanish name)
- Marianne Belloti Exversion
- John Temple Manhattan District Attorney's Office Human Trafficking Response Unit
- Genna ECPATUSA
- Janai Smith ECPATUSA
- Geofrey Digital Ocean
- Phil Self Employed
- Lior Shahverdi Self Employed
- Alan NYU
- Tim Savage NYU
- Steve EPA
- Rob Spectre - Twilio
- Jon Gottfried - Major League Hacking
- Ben Singleton - NYPD
- Bryan Britten - Manhattan DAs office

- Tara Byrne - ???
- Noelle - heatseeknyc
- Ziba Cramner - Demand Abolition
- Dhakir - Demand Abolition
- Delaney - Demand Abolition
- Andreas Mueller - NYU
- Darius - ???

And now I need to devote a paragraph to this person. 5 years ago I was just a graduate student hoping to work on human trafficking. I got to do a small project with the department of homeland security and I thought that was going to be it. However thanks to you, Brandon Diamond, I was able to share that work on the stage of New York Tech Meetup, just a month after I finished it. Thanks to you I was able to become a part of the social justice community and face the fear of speaking to thousands of people, judging you. You've always been a good friend, but when you recommended me to speak in front of that community, you changed my ability to have an impact on the world. That's not something even most good friends can do. I owe you more than these words. I owe you my undying gratitude and hope we will be friends for a long time to come. I love you, man. Thank you for helping me do what I've always selfishly wanted to do. For bringing my work forward. For making me better, smarter, stronger and more capable. I learn something new from you every time we speak, not something you can usually say about someone you've known since before high school.

And finally to James Powell, without your help I'd never be able to speak to as many audiances as I have. You have given me limitless speaking opportunities and chances to share my story and the story of the voiceless - those who are trafficked. Thank you for everything you've done for me.

Prerequisite Knowledge

The intention of this text is to cover ground quickly. However, I want this text to be accessible to those who don't know everything. Here are some of my favorite guides to get all the prerequisites out of the way.

list of prerequisites:

For Beginners:

- **Codecademy**
- **Khan Academy Python**

Transition from Beginner to Intermediate

- **Learn to Code the Hard Way**

- **Python Open Book Project**

Intermediate

- **Data Structures and Algorithms in Python**

Intermediate and Advanced Topics:
More Python Resources:

- **General Set of Guides for Python**

Data Science Resources:

- **Set of Guides for Learning Data Science (Mostly Free)**
- **A list of Tools for Data Science**
- **A talk on named entity recognition** Web Dev:
- **A list of web dev tools and learning resources**
- **A curated list of Web Dev Guides**

References:

- **Thomas Levine's personal Blog**
- **Aaron Schumacher's personal Blog**
- **Data Science 101**
- **R-Bloggers**
- **Hardcore currated list**
- **James Powell's Blog**

A quick note on installation

A lot of the tools we'll be using throughout this text will rely on pip, so we'll download and install that now. From time to time we'll need python packages that don't have working pip repos (at the time of this writing) and so alternatives will be presented for those libraries.

To get pip head over to **get-pip.py** simply save the file as get-pip.py. The run sudo python get-pip.py (on ubuntu or mac) or run python get-pip.py with a administrator rights (on windows).

Section 1 - Working With Data For Non-Profits And Government

2

The biggest problem facing government agencies and non-profits today is data. These problems stem from cleaning, storing, maintaining and sometimes even accessing this data. Lots of entities have been slow to the technology revolution and often they don't even have basic functionality like search. Much of the data government deals with is either trapped in emails, pdfs, word documents, or pictures, and thus unsearchable. Of course, many, many solutions exist to this problem, but often times getting such a solution installed is a herculean task, taking many many years. Therefore, writing your own solution is usually going to be faster. This type of task addresses our first strategy to being effective in government: hack around obstacles.

For whatever reason, I've found that government administrators are much more comfortable with having the python language on their machines than a open source programs or tools like Gephi. My guess is this is a mix of fear and misunderstanding of open source technology in general, but I can't be sure exactly why. In any event, I've found myself waiting months or even sometimes being told a flat out 'No!' to installing specific open source projects. And therefore, I've had to come up with my own solutions to many well solved problems. This certainly isn't the case for every government institution or non-profit, but from what I understand talking to other technical colleagues across local government, it seems to be more the rule than the exception.

Some specific obstacles we'll be hacking around are: getting access to data your organization is already open sourcing and getting data out of PDFs.

- Web Scraping
- Turning PDFs into CSVs

Web Scraping

Web scraping is the ability to programmatically store and process data from API's, websites, RSS feeds, and other content that exists on the internet. The steps usually involve two things - download the relevant content locally and then parse that content, extracting the relevant information.

There are many great blog posts, tutorials, and tools for doing web scraping. Here we'll touch on my three of my favorite:

- **Python's requests Library**
- **Python's lxml library**
- **Python's Selenium library** & **Phantom.js**

Installation

Mac OSX or Ubuntu

sudo pip install -U requests lxml selenium

Windows

with administrator rights:
    pip install -U requests lxml selenium

Getting started with requests

The requests library is extremely powerful and very high level. Before we talk about how we'll use it, let's discuss some reasons why we'd use it. The common reasons I've seen for using requests is there is some data served somewhere on the internet that people at your organization care about. For my work fighting slavery, we've often wanted to store the content of a website we think is encouraging or creating human trafficking. We want to do this for a few reasons, but the most important one is to gather complete and robust evidence because often times this data can be deleted, lost or destroyed. If that happens then a vital piece of an investigation may be lost. Another common reason I've seen is for collecting statistics that another website posts, but that isn't publicly available in another format easily. The other big reason only happens in large government agencies. This is the case when a certain process publishes regular reports from a database, but you can't find anyone who has access to the database, but you need the data.

There's one common theme here: there is data that is publicly or semi-publicly available that you'd like to be able to manipulate or store locally, and this data is updated overtime.

The requests library allows you to do some very important things, we'll only talk about two of the methods in the library - GET and POST. It is assumed that you already know object oriented programming, so if you don't please go check out the resources I listed above so that you know how classes, objects, and data structures work.

Working With GETs And POSTs

Note: For this section of the text please refer to `making_requests.py` and `test_server.py` in **chapter1 of this text**.

A GET request asks the server for some data. A POST request sends some data to the server. They are easy to do in the requests library and have specific uses:

The canonical GET request is something like visiting a web page - the html that is rendered by your browser was by a GET request (more often than not).

Here's how that method might look:

```
import requests
r = requests.get("https://www.google.com")
```

This request returns an object - which we've named r, short for response. This is a classical pattern called request, response and you'll find it in other places, like javascript MEAN stack applications. There are a number of properties that the response object returns. The most important ones are:

- text - stores the html of the page as a string
- url - stores the url of the request
- content - stores the content of the request, in case we are dealing with something other than html like a pdf
- raw - the same thing as content more or less, but there is a difference in practice, although I'm not sure what exactly. Best to try content and if you don't get back everything you wanted, try raw.
- headers - the headers of the request.

You'd access this information in the following way:

```
import requests
r = requests.get("https://www.google.com")
print r.text
print r.url
print r.content
```

```
print r.raw
print r.headers
```

A POST request is one that sends data to the server. Since I don't want to have people sending the same fake data to some website, I wrote my own test server for this example and the remainder of the section.

So you'll need to get the github repo - `https://github.com/EricSchles/tools_for_social_justice` which you can get by typing:

`git clone https://github.com/EricSchles/tools_for_social_justice.git`

then navigate over to `tools_for_social_justice/code/chapter1/`. From there you'll be able to run the server with `python test_server.py` and all the examples with `make_requests.py`.

So go ahead and run the server and then the following code will work:

```
payload = {"username":"eric","pass":"1234"}
r = requests.post("http://localhost:5000/form",data=payload)
```

This request will post the following data - `"eric"` and `"1234"` to the form, which is pretty neat! Often times you'll need to log into certain government websites, before you can access their data and this will allow you to do just that. Other uses include automatically filling out forms for surveys - say your unit needs some piece of software and the agency happens to be sending around a suggestion form, you can write the data so that it's clear that you really want this one specific thing and then pass it around to all your coworkers to run this script. Just make sure you get your bosses approval first! But your coworkers will love you for this. I've spent way too much time filling out internal surveys just to drop the hint that our unit needs one specific thing.

I do want to note, that not all websites will allow you to pass credentials and then scrape their content, sometimes a web page is generated dynamically, sometimes a website will pass back a cookie, tracking your state while accessing the content of the website. We'll address each of these cases as well, but this is the simplest form of submitting data to a website.

An important thing to note is, the data here - called payload is sent after the html page is loaded, by the server. And so it makes sense that a dictionary is passed - the keys correspond to the names of the fields in the html page, the values, the actual data being passed to the form.

When making use of an api that requires credentials, we'll need to pass a different keyword - auth (instead of data). This will send the credentials along, before the html page is loaded.

```
data = ('admin','1234')
r = requests.get("http://localhost:5000/secret-page",auth=data)
```

Some important differences to note - first we pass a tuple - the credentials listed in order. Then we use the `auth` keyword to pass the data. Lastly we make use of a get request, because we want to get some data from the server, rather than sending some form data to the server.

The next example illustrates sending files via a form to the server. This can be extremely powerful, if you know you have some internal web service and 500 files that need to be processed. Perhaps classically this service only processed one file at a time. Rather than having to track down the maintainer of this service, if they are even still around, it's far easier to have python just send all the files across.

This simple example illustrates how to send a file via post request.

```
files = {"file": open("report.csv","r")}
r = requests.post("http://localhost:5000/get_csv",files=files)
```

Notice that this is a dictionary, where the key corresponds to the name of the input form and the value corresponds to the file in question. For pdfs and microsoft documents you can use `open([filename],"rb")`. This stands for read binary.

Parsing HTML

Now that we know how to get and send data programmatically, we are ready to start being selective about what we download and keep or write a web crawler. Web crawlers are extremely powerful for mapping every page of a website and infact, I've done just this to forensically download and store websites over time, to be used as evidence of human trafficking. Other uses exist in the investigative world, like navigating the deep web.

```
import requests
import lxml.html
from unidecode import unidecode
from sys import argv

website = argv[1]
r = requests.get(website)
html = lxml.html.fromstring(unidecode(r.text))
print html.xpath("//a/@href")
```

In this simple example we see the power of web parsing, in its simplest form - grabbing all the links from a web page. Once we have a web page in memory, we transform the ascii characters into something python can understand, via unidecode, and then loading the html into the lxml tree data structure. Once we've parsed all the tags via lxml's fromstring method, we are free to make use

of xpath - a micro language that will allow you to quickly and efficiently traverse a tree to query for certain peices of information, in this case all the hyper links in an html page.

While xpath as a language is far too deep in terms of length to go into in detail, we'll cover a few features here:

`//` - searches the whole tree for the following tag, from the root of the tree `/` - search from the relative node that is referenced so `//a/p` will search after all the a tags, and `/a` searches only the top level of nodes `/a/@href` - searches for a tags with the href attribute `/a/p[@id="hello"]` - searches for all a tags with `id="hello"`

If the xpath language isn't familiar to you I'd recommend the following guides:

- **w3schools**
- **mozilla guide**

Crawling the web

One of the great things of python is how readable and compact you can make the code. The following piece of code is a fully capable web crawler (albeit without any bells or whistles). A slightly more robust version of this (complete with a database) lives on the github repo for the book **here**

```
import requests #sudo pip install requests
import lxml.html #sudo pip install lxml.html
from unidecode import unidecode #sudo pip install unidecode

def links_grab(url):
    r = requests.get(url)
    html = lxml.html.fromstring(unidecode(r.text))
    return html.xpath("//a/@href") + [url] #ensures the url is stored in the final

def crawl(base_url,start_depth=6):
    return crawler([base_url],base_url,start_depth)

def crawler(urls, base_url, depth):
    urls = list(set(urls))
    domain_name = base_url.split("//")[1].split("/")[0]
    url_list = []
    for url in urls:
        if domain_name in url:
            url_list += links_grab(url)
    url_list = list(set(url_list)) #dedup list
    url_list = [uri for uri in url_list if uri.startswith("http")]
    if depth > 1:
```

```
            url_list += crawler(url_list, base_url, depth-1)
        urls += url_list
        urls = list(set(urls))
        num_urls = len(urls)
        return url_list
```

So let's break down the code. The `links_grab` function should look familiar, since we made use of it in the last example. Notice the small difference - the return statement is adding two lists together, which is why [url] is stored as a list.

Next we'll look at the crawler function.

The first line - `urls = list(set(urls))` deduplicates the urls as they are passed in, this ensures no url will be scraped twice, which would be redundant.

The next important line is - `if domain_name in url:` - this ensures that we are only grabbing content from one domain at a time. I include this because otherwise our list of urls will grow exponetially fast and by the fact that we aren't trying to necessarily re-implement the google crawler. Typically in investigative work, we only care about grabbing all the html for a single domain, rather than multiple. Of course, there will be some circumstances when you want all the domains linked to a website, if that is the case, simply remove this line. You can of course add further customization - simply by ignoring certain domain names. Below is a modified crawler method that ignores popular websites you are unlikely to care about when trying to store neifarious websites:

```
def crawler(urls, base_url, depth):
    domains = ["www.google.com","www.yahoo.com","www.linkedin.com","www.github.com"]
    urls = list(set(urls))
    url_list = []
    for url in urls:
        if not any([domain in url for domain in domains]):
            url_list += links_grab(url)
    url_list = list(set(url_list)) #dedup list
    url_list = [uri for uri in url_list if uri.startswith("http")]
    if depth > 1:
        url_list += crawler(url_list, base_url, depth-1)
    urls += url_list
    urls = list(set(urls))
    num_urls = len(urls)
    return url_list
```

Here we simply create a new list of domains and if any of them are found, we don't scrape those urls. The assumption is that you don't start from one of the urls on the no-scrape list.

Once we've scraped all our links we dedup again with `url_list = list(set(url_list)) #dedup list` and then make sure all our urls are

reachable via their uri `url_list = [uri for uri in url_list if uri.startswith("http")]`. Many href's refer to relative paths for websites. Since this is just an introductory example, I don't include the code to resolve this here, however in the github repo, I have all the necessary added steps to resolve relative urls. I have to warn you however, this makes things very slow. You'd be surprised how many urls you can get from a single scrape.

Notice the next lines `if depth > 1:` and `url_list += crawler(url_list,base_url,depth-1)`, this is the recursive step and sort of the heart of the engine. Since we are moving through the urls recursively, scraping new links from the current step, our list of urls will get big fast. We can think of the structure of a single scraping like a b-tree, where the number of edges at a given level corresponds with the number of links on a page, and each node being the actual html page, storing said edges. This structure shows just how exponentially large our b-tree grows, often very big very fast. It would be an interesting study to look at how the number of links grow from page to page on average for different types of websites, allowing us to understand an average bounding on the time complexity of a type of scrape. But enough about the how, let's talk about the why.

Often times we need to scrape a set of pages for a few reasons:

1) To get information trapped from some database that you are supposed to have access to, but don't, mostly because the person who runs said database is being difficult and doesn't want to give you a direct feed. However the information is of course, published publicly in a far less machine readable format.

This is an example of hacking around obstacles, something you'll need to do often if you decide to work for the government or really any non-profit. I'm not sure why folks are stubbornly unhelpful in these roles, but usually they are.

2) To grab information for an investigation. Often times on the sex trafficking world traffickers will post to a website like backpage.com but have their own websites that you can link to. The content on these smaller websites will often change, sometimes very regularly, and sometimes not at all. It is important to an investigation to know all the people that are featured on a given website of this kind, and so being able to readily store such information is of paramount importance.

3) To do market research. An unfortunate part of the non-profit space is grant writing. This takes up a lot of time for many NGOs and non-profits, so knowing what everyone else in the space is saying with real-time accuracy is often a big deal. Being able to scrape the other folks in the space becomes an important part of the grant writing process - knowing not only what they are saying, but what they are not saying.

4) Scraping other government entities. If you thought your IT department was bad, wait till you meet the IT folks in other government agencies, these are typically the least helpful people possible. If your IT staffer decides he/she

doesn't like you and doesn't want to help, at least you can escalate things to superiors so that they have to do it, if other people in other IT departments don't like you, there is no path forward, ever. But often when working on a problem with a social justice theme, you'll need the information that other government agencies are using. Thanks to open data standards that are being forced on many public agencies, for the first time, you'll have the ability to actually circumvent this difficult people, so that you can actually do your job! Often times, you'll be able to make the scraping happen, so you can get your work done.

Turning PDFs into CSVs

This next section really addresses this last point from the previous section - working with government data from other entities. As I've already said, many government institutions will have to release a bunch of their data to the public, in an attempt to create a more open and transparent government. Of course, this has lead to a ton of problems. Partially because many government instutions aren't really equiped to do that, from a technical stand point. And so, often what you get, is folks just publish static pdf documents to the web and think they are being open and transparent. One of the most important things open governance needs is the ability to make this data machine readable. However, this is likely a far off dream, so we are going to hack around it.

The uses of machine readable data across government are limitless, but here are a few my favorite ideas -

1. Creating citizen based analysis to show however government could be more effective:
    - Maps showing you were all the safest parks are based on crime data
    - High level financial analysis showing where and government dollars are actually spent, as well as how they could be spent better

1. Creating citizen based feedback loops showing government how they could better govern:
    - Active voting from quarter to quarter, based on the performance of a specific agency, which partially determines funding level

Both of these could be accomplished via an API, number (1) from an api-GET request and number (2) from an api-POST request. Thats it.

In lew of such a fully functional API and having to work with PDFs isn't the end of the world. In this section we'll cover how to parse PDFs and give you stragies for getting the information you want.

The first strategy involves finding static text that doesn't change from document to document, this is typically useful for quarterly reports that are gener-

ated from a database. Because parts of the text don't change, you can use these as relative offsets to only process the parts of the document you care about. I'll refer to these pieces of unchanging text as invariants or text invariants.

Once we have the invariants figured out, we'll need to understand the structure of the data or text we want. This is typically a table, or a chart. Ofcourse, from time to time we also want some text, but this method generalizes nicely to that case as well.

Installation

Pandas - `sudo pip install pandas` or `sudo apt-get install python-pandas` #ubuntu only Poppler-utils:

- On ubuntu - **http://poppler.freedesktop.org/**
- On Windows - **http://blog.alivate.com.au/poppler-windows/**

Note for windows you'll need to set the environment variables or pdftotext. For information on how to do this please see **this guide**

The following piece of code can be found in the chapter1 section of the github:

```
from subprocess import call
from sys import argv
import pandas as pd
def transform(filename):
    call(["pdftotext","-layout",filename])
    return filename.split(".")[0] + ".txt"

def segment(contents):
    relevant = []
    start = False
    for line in contents:
        if "PROSECUTIONS CONVICTIONS" in line:
            start = True
        if "The above statistics are estimates only, given the lack" in line:
            start = False
        if start:
            relevant.append(line)
    return relevant

def parse(relevant):
    tmp = {}
    df = pd.DataFrame()
    for line in relevant:
        split_up = line.split(" ")
        # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
        split_up = [elem for elem in split_up if elem != '']
```

```
            if len(split_up) == 7:
                tmp["year"] = split_up[0]
                tmp["prosecutions"] =split_up[1]
                tmp["convictions"] = split_up[3]
                tmp["victims identified"] = split_up[5]
                tmp["new or ammended legistaltion"] = split_up[6]
                #print tmp
                df = df.append(tmp,ignore_index=True)
        return df

    if __name__ == '__main__':
        txt_file = transform(argv[1])
        text = open(txt_file,"r").read().decode("ascii","ignore")
        contents = text.split("\n")
        relevant = segment(contents)
        df = parse(relevant)
        df.to_csv("results.csv")
```

This code parses the pdf - trafficking_report.pdf which can also be found in the github. This is a very long report, but what we care about is one table in particular (by way of example). This table can be found on page 45 of the report and it details arrest details about human traffickers internationally.

Let's walk through each of the methods, which are defined in the order they are called.

```
    def transform(filename):
        call(["pdftotext","-layout",filename])
        return filename.split(".")[0] + ".txt"
```

The transform method calls pdftotext, a command line utility that transforms a pdf into a .txt document. Notice the use of the layout flag which preserves the formatting as much as possible from our pdf file, this is crucial to ensuring our .txt document will be easily parsable.

```
    def segment(contents):
        relevant = []
        start = False
        for line in contents:
            if "PROSECUTIONS CONVICTIONS" in line:
                start = True
            if "The above statistics are estimates only, given the lack" in line:
                start = False
            if start:
                relevant.append(line)
        return relevant
```

In the segmentation step we find the text invariants that start and end the section of the document we wish to parse. Here the start invariant is "PROSECUTIONS CONVICTIONS" and the end invariant is "The above statistics are estimates only, given the lack". What's returned is a simple dictionary of the relevant lines of text from the transformed .txt document.

```
def parse(relevant):
    tmp = {}
    df = pd.DataFrame()
    for line in relevant:
        split_up = line.split(" ")
        # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
        split_up = [elem for elem in split_up if elem != '']

        if len(split_up) == 7:
            tmp["year"] = split_up[0]
            tmp["prosecutions"] =split_up[1]
            tmp["convictions"] = split_up[3]
            tmp["victims identified"] = split_up[5]
            tmp["new or ammended legistaltion"] = split_up[6]
            #print tmp
            df = df.append(tmp,ignore_index=True)
    return df
```

The final section is the parse method. Here we create a dictionary which will be appended to the pandas DataFrame that we'll make use of. The reason for storing things in a pandas dataframe is because of the ease of transformation to other persistent file stores such as CSV, EXCEL, a database connection, and others. Notice that inside the loop we split up the line by whitespace, since this is a table, we should expect tabular data to appear in the same position on different lines. Also notice that we expect the size of each split up line to be of length 7. Not that this does not capture a few of the lines in the original pdf, it is left as an exercise to handle these minor cases. Length is not the best metric for ensuring you are processing the correct information, but the intention of this code is to be as readible as possible, not necessarily as sophisticated as possible. Other ways you can check to ensure your scraping the correct text is with regex, checking for certain character invariants that should appear on every line, or by using advanced machine learning techniques which we will talk about in the next section. Finally, the tmp dictionary is assigned each of the values from the table and appended to the dataframe. Notice that we only need to create tmp dictionary once and then simply overwrite it's contents on each loop through the relevant content. Then we simply return the dataframe to main. Here a single line is used to send the dataframe to a csv:

```
df.to_csv("results.csv")
```

And we are done!

There are other, more elegant ways of parsing pdfs, like those found in **this chapter** of automate the boring stuff. Unfortunately, using a library like PyPDF2 won't work in all cases. So while my method is certainly not elegant, it is robust and will work 99% of the time.

Data Cleaning techniques

Now that we know how to handle PDFs under the best possibel scenario, when there are no Optical Character Recognition errors, its time to deal with slightly less than ideal situations, IE when there are mistakes in your data. In the typical large company setting data isn't entered the same way consistently, which is an annoying problem, however for that we have things like named entity recognition, which we'll get into. What I'm talking about is when your data is plain and wrong. Here's an example of what I mean:

A row is supposed to say:

For check number ending in 7519:

But it actually says:

F0e c$#ck namb ending in 7%@!:

How the heck are you supposed to handle this situation?! This is what actually happened to me while working for the manhattan DAs human trafficking response unit. We'd get a ton of documents from subpoena compliance, but we could only afford an absolutely terrible OCR solution and so the .txt documents hand BOAT loads of errors just like the above. And here's the truly frustrating thing, they were never consistent. The OCR solution would mess things up, often differently, for the same words. And so you'd never get consistent results. This is really bad for two important reasons:

1. Because it means my invariant scheme falls apart :((((<- the many chins of sadness rained down upon me the day I realized this.
2. Because it means your data will be inconsistent and incorrect in your CSV :(((((<- even more chins of sadness, I had to stress eat to cope with the level of depression and thus became even fatter.

So, how do we get around this? Unfortunately there is no one size fits all answer, but there are lots of things you can do:

Let's say you know that there are certain words that will never appear in a line with something you do care about, but this data is being collected, I.E. you have extra data. Then you could do something like this:

```
def parse(relevant):
    tmp = {}
    df = pd.DataFrame()
    to_avoid = [
```

```
        "Hard to see",
        "weird",
        "Something else you want to avoid"
    ]

    for line in relevant:
        if any([elem in line for elem in to_avoid]):
            continue

        split_up = line.split(" ")
        # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
        split_up = [elem for elem in split_up if elem != '']

        if len(split_up) == 7:
            tmp["year"] = split_up[0]
            tmp["prosecutions"] =split_up[1]
            tmp["convictions"] = split_up[3]
            tmp["victims identified"] = split_up[5]
            tmp["new or ammended legistaltion"] = split_up[6]
            #print tmp
            df = df.append(tmp,ignore_index=True)
    return df
```

With this you can keep a list of all the terms you want to avoid and simply not process any line that has this term you know will cause problems. So now we can handle the case when we have too much data, what happens when we miss some data?

For that we'll need to account for all the variations of a given starting or closing phrase:

```
def segment(contents):
    relevant = []
    start = False
    starting = [
        "PROSECUTIONS CONVICTIONS",
        "PROSECUTIONS",
        "CONVICTIONS",
        "CONVICTION$",
        "C0NV1CT!ONS"
        ]
    ending = [
        "The above statistics are estimates only, given the lack",
        ", given the lack",
        "The above statistics are estimates",
        "statistics are estimates"
        ]

    for line in contents:
        if any([elem in line for elem in starting]):
```

```
            start = True
        if any([elem in line for elem in ending]):
            start = False
        if start:
            relevant.append(line)
    return relevant
```

Here we can see employing a similar trick. Every time we encounter a new type of OCR error, we handle it in specific.

Section 2 - Data Visualization 3

So far we've seen how to process data and make use of statistics/machine learning to automate analysis and get stuff done. Unfortunately for most non-technical folks working in government and non-profit land, this is really not useful. There are a few reasons behind this:

1. You're going to leave - The government and/or non-profit don't pay very well and eventually you'll need to actually make a living, paying down student debt, raising a family, covering medical expenses, going to the dentist and a whole other laundry list of things you won't be able to afford. Everyone knows you are leaving, because you won't be the first technologist who's ever tried to make a difference and you won't be the last. But sadly, they really can't afford you and never will be able to.
2. They don't actually understand what you've built - few non-profits or government folks understand or embrace technology. Which is why you can have such a large impact. But it also means they don't really understand what you are doing, why you are doing it or how you are doing it. So you need to do a lot of hand holding throughout the process

For these reasons, building clean, pretty interfaces is almost twice as important as actually making sure the technology works. If they don't understand what you've made intuitively (aka without really understanding) they won't use it. No matter how much money the organization as a whole paid for the software, service or technology.

So we are going to talk about web development and data visualization.

Design Process

Websites built for the social justice space need to be as simple as possible. You can do a max of three things per page, but really you should only do one thing. And each tool should never really do more than one to two types of tasks. It's also important to understand, that no matter how useful a tool or set of tools

might be, there is a finite amount of brain space for actually using tools, no matter how much easier the next thing might make someones life. So, you need to make sure that whatever you build is as useful as possible, even if it's technically very boring.

In fact, the tool that has had the highest adoption rate since I started working in government is the PDF to CSV tool I touched on in section one. All the sophisticated stuff I've put into production, took 2 years to find the right client - a national organization of folks who came from fortune 500 companies, to understand and make use of those analyses. So, the point, keep it simple, otherwise no one will use it.

So how should design of websites like this look? I don't know. And neither does anyone else. There are certain rules I've picked up, but in all honesty, you'll need to do an agile iterative design, before you have any idea about what you are going to build or how it's going to look. If you can add bootstrap, great! But don't do it until after you've iterated on your design with all the members of your team.

So how do you do agile in the government or non-profit space?

1. Find a person who is excited about technology - this is going to be your alpha tester. He or she will be your companion on the journey to creating something useful. For me, this was a young woman named Rachel. Over the past year we've worked together extremely closely. It hasn't always been fun, but its almost always been productive. The way our interactions typically work is, I get a requirement from my boss or I pitch an idea. I explain the idea to Rachel, she gets way too excited and thinks I'll finish it tomorrow, and then I get to work. Typically later that day I'll have a prototype with a basic flask app and a button. The button will give her the output from the task and she'll give feedback about the look of the output, the interface and the workflow. I'll take her notes and iterate on the design. Usually we'll do this three to five times.
2. Show someone else - Usually by this point, a week or two later, I'll have something that other people will understand and be able to use. Typically all the design work has already happened with Rachel, since she knows the rest of the team very well. However, making sure the workflow is intuitive to other folks is important, because it gets their buy in. Its important to get as much of the teams buy-in as possible. This ensures that more people will adopt the tool. Because remember, just because you build, it doesn't mean they'll use it. Even if its what they asked for.
3. Get by in from management - Its a sad truth, but no matter how good a tool is or how much time its going to save, if your boss doesn't care, it won't get put into production. So make sure you have his or her approval. It's also important that they know about the project from the beginning.

So although you have someone else who is working on it, make sure they are onboard with what you are doing.

4. Get ITs buy in - Unfortunately my personal situation didn't really allow for this. So I recommend making friends with IT early on in your time. Your going to need them on your side if you want to get stuff into production, at least in many government institutions. Unfortunately, most of the time ITs (preference) is to say 'no' to new projects. So if you can get ally's here, it will make a world of difference. Assuming you can convince them to be on your side and actually do stuff that's meaningful, you should run it by them after you tell your boss, but before making the request to put it into production. If possible show them a prototype.

So really steps (3) and (4) happen after you've already started the iteration process with your alpha tester, but before you've completed it. So maybe after the second iteration.

Design Examples

Now that we've touched on the design process, let's talk about the design itself.

Installation

flask - sudo pip install flask

Building our first flask app

Flask is one of the simplest and most useful web frameworks in the world. This is not only because you can create servers with the absolute minimum number of explicit lines of code, but because it has a ton of extensions that make it robust when necessary, but flexible enough to just get stuff done.

The following example is done entirely from the python shell:

```
>>> from flask import Flask
>>> app = Flask(__name__)
>>> @app.route("/",methods=["GET","POST"])
... def index():
...     return "Hello there"
...
>>> app.run()
```

The same app can be found **here**

As you can see, writing a very minimal flask app is extremely easy. And the good news is it continues to be easy for larger apps. So how does a flask app work?

First there is the flask object - `Flask` - this object creates the context for the running website. You can add routes to the app context and then run those routes. A route represents a url path and the action(s) the server will take when a browser visits the page, or more technically when a request is made to the endpoint. Notice that route takes one parameter by default and lots of optional parameters. The most common one is methods, which tells the server what kinds of requests the end point should handle. Typically get requests are for getting html pages or other kinds of data from the server, whereas post requests are for sending data to the server, as is the case with forms. (You may remember this type of discussion from chapter 1).

Let's do something slightly more complex now. Doing so will require a few files and some folders. Here is our file structure:

web*app/ run.py app/ _init*.py server.py templates/ index.html

It may seem artificial to seperate our application into all these different pieces, but this structure will be very useful once we start using blueprints - a pattern that lets us split up pieces of the web application into different application contexts. This allows for faster development and better organizational structure at scale. It also leads us well to understanding the structure of the django framework, which we'll get into soon!

run.py:

```
from app import app

app.run(debug=True)
```

init.py

```
from flask import Flask

app = Flask(__name__)

from app import server
```

Basically, **init**.py handles the instantiation of our application context and run.py handles actually running the server.

The server.py file is where all our routes are going to live for now. This application will still be very simple - its simply a form that takes in a name and email address and then displays the pair in a list below the form. While this may be a trival example it illustrates two major tasks in web development - getting data from the server and sending data to the server. Another important aspect of

this application, is the data is persistent, because we make use of a simple database.

So let's take a look at server.py:

```
from app import app
from flask import render_template,request,jsonify,redirect,url_for
from flask.ext.sqlalchemy import SQLAlchemy

###Models

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///database.db"
db = SQLAlchemy(app)

class Store(db.Model):
    __tablename__ = 'store'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(400),nullable=False)
    email = db.Column(db.String(400),nullable=False)

    def __init__(self,name,email):
        self.name = name
        self.email = email

    def __repr__(self):
        return  "<store %r>" % self.name

###Controller

@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())

@app.route("/add_data",methods=["POST"])
def add():
    print "posted data to the server"
    name = request.form.get("name")
    email = request.form.get("email")
    store = Store(name,email)
    db.session.add(store)
    db.session.commit()
    print store
    return redirect(url_for("index"))
```

Here we've split up our concerns into model and controller - this idea of splitting up concerns in this way comes from the model,view,controller pattern. The view logic lives in the html file, which we will see in a few moments. The model concerns model our data, telling our database what kind of data should

be stored, as well as how that data should be represented. We express storage via:

```
__tablename__ = 'store'

id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String(400),nullable=False)
email = db.Column(db.String(400),nullable=False)

def __init__(self,name,email):
    self.name = name
    self.email = email
```

We model our data as sqlalchemy's general purpose Object Relational Model. The individual fields are modeled via db.Column. We then inform the database what to do on instantiate of our record object - with the **init** method. Notice that we don't need to pass an id, it is generated automatically. The id is good practice and is useful for indexing data, allowing for faster queries, and database tunning generally.

Now we can explore the other part of our database class:

```
def __repr__(self):
    return  "<store %r>" % self.name
```

Here we make use of the **repr** method which will determine the representation of individual objects of our database, assuming we want high level information about the objects. We can also access individual fields from our objects in the following way:

testing_database.py:

```
from app.server import Store

store = Store.query.all()[0] #get any old object.
print store.name
print store.email
```

Notice that we can access the individual fields, and so, **repr** is more of a notational convenience more than anything else. This way we can know what object we are dealing with, without having to deal with accessors.

Now let's look at the controller:

```
@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())

@app.route("/add_data",methods=["POST"])
```

```
def add():
    print "posted data to the server"
    name = request.form.get("name")
    email = request.form.get("email")
    store = Store(name,email)
    db.session.add(store)
    db.session.commit()
    print store
    return redirect(url_for("index"))
```

Here we don't have a ton of new stuff. Notice that we only expose a POST method for our add_route, which will be used for submitting our form data. Notice that I name the route the same as the method, this is good practice, as we will see, for `url_for`, which resolves method names to their url.

At this point, it makes sense to bring in the html to understand the interaction between the view and the controller:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>

<form method="post" action="">
<label>Name:</label><input type=text name=name>
<label>Email:</label><input type=text name=email>
<input type=submit>
</form>

<ul>

</ul>

</body>
</html>
```

So as you can see, the form action calls the add_data route, via url_for. Notice the use of openning - . This is what tells the jinga template engine to execute different python and flask methods. Typically it's for calling routes, rendering data, or resolving urls. Notice also that the name attribute is present in both of the input fields and these correspond to the `request.form.get`, which grabs the necessary input from the form. Finally, we can look back at our controller:

```
store = Store(name,email)
db.session.add(store)
db.session.commit()
```

These three lines are what create our Store object, with our name and email and then save these values to our database. Notice there was no specific connection made here, nor any database specific code. Meaning, we can safely swap databases by changing a single line of code:

`app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///database.db"`

The two recommended databases are postgres and sqlite - postgres for production and sqlite for local testing, however SQLAlchemy supports a plethora of datastores.

Next, let's look at this part of the html:

```
<ul>

</ul>
```

Here we see a dynamically generated list, that grows with the size of our data list. Notice that we can access specific data from our datum, since data is a list of type Store. The is used to evaluate individual datum. Notice also that we need an `endfor` statement, ending our for-loop.

This html corresponds to the index method:

```
@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())
```

Notice that data is passed as an optional parameter to the `render_template` function, which is why the list is called data in the html. Whatever we name the optional parameter, will be the name of that parameter on the front end. The render_template function accepts all native python data structures - variables, lists, and dictionaries. In this case, we pass a list, which is all the objects in the Store table. Notice that we chose to pass all, but we could also have filtered on certain parameters. To filter a query with sqlalchemy objects we simply need to do something like the following:

`print [obj for obj in Store.query.filter(Store.name == 'Eric')]`

As you can see, the filter expects anything that would satisfy a WHERE clause in SQL. Most of the time I just work with basic booleans like equality, less than or greater than, but other more complex query terms are possible.

C3.js

We now have all the necessary rigging to work with C3.js or D3.js, vincent, or any other data visualization package we could want. Let's start with C3.js since it is very easy.

C3.js and flask

Now that we understand how flask works, we are ready to jump into C3.js! With C3 we can make beautiful dynamic charts and graphs with just a few lines of code. There isn't too much new in the server below, since that we are generating random lists of integers to be visualized.

server.py:

```
from flask import Flask,render_template
import json
from random import randint
app = Flask(__name__)

@app.route("/",methods=["GET","POST"])
def index():
    data1 = ['data1']
    data2 = ['data2']
    for i in xrange(0,randint(0,15)):
        data1.append(randint(0,150))
    for i in xrange(0,randint(2,25)):
        data2.append(randint(7,450))
    return render_template("line_chart.html",data1=json.dumps(data1),data2=json.dumps(data2),


app.run(debug=True)
```

Here is where all the magic happens, specifically within the script tags:

line_chart.html:

```
<html>
    <head>
        <title>Data Viz example</title>

        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/c3/0.4.10/c3.min.
        <script src="https://cdnjs.cloudflare.com/ajax/libs/c3/0.4.10/c3.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.6/d3.min.js"></script>
    </head>
    <body>

    <div id="chart"></div>
```

```
    <script>
        var data1 = JSON.parse();
        var data2 = JSON.parse();
        var data3 = JSON.parse();
        var chart = c3.generate({
            bindto: '#chart',
            data: {
                columns: [
                    data1,
                    data2
                ]
            }
        });
    </script>

</body>
</html>
```

Notice that we need to create a div tag with an id that is the same as the bindto parameter in our `c3.generate` function. In this case we chose to name our id chart. Notice the use of the # infront of chart. Notice also that we create a variable called chart for this chart that will be generated. Notice also that adding or data to our graph is extremely easy, we just need to include our columns. Since the data is being loaded from the server, it might not be clear what these columns of data look like. So let's be explicit:

`data1` will be a list, where the first element (at index 0) is the name of the data set, in this case, data1. This is stored as a string. The rest of the elements are technically optional and can be an integer or a float. So data1 might look like this:

`['data1', 4,17,25,47]`

The same pattern will hold for data2 and any other columns you may want to add.

With c3 we can make lots of types of charts very, very easily!

You can find more examples and the complete server for splines, bar charts, and pie charts in the github **here**.

Free GIS Systems, in Python – working with leaflet and flask

Leaflet is a wonderful library. It allows you to visualize a whole bunch of information, geographically! One of the things I've gotten into recently is making use of GIS maps to show my boss and other folks how their data looks. Making use of leaflet allows us to make this super pretty.

With leaflet we can do things like, create polygons, add visual representations of data, add paths, create heat maps, and add various layers of control and interactivity.

All and all, leaflet lets us visualy interogate a ton of data and do so in an easy and fun way! With something very visually pleasing.

So enough talk, let's look at how to make use of leaflet, with this basic example, **lifted directly from their docs**:

http://leafletjs.com/examples/quick-start-example.html

Let's start by looking at the head of the html page:

```
<head>
    <title>Leaflet Quick Start Guide Example</title>
    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.c
</head>
```

Here we include the necessary CSS via a CDN (content delivery network), note that the official documentation says to use 0.7.4 but when I tried it 0.7.3 is the one that worked. I'm not sure how long this will be true for, but for now, please use 0.7.3.

Now let's look at the body of the html, up to the script tag:

```
<body>
<div id="map" style="width: 600px; height: 400px"></div>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></script>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet-src.js"></script>
```

Here we see that we add our map in a div tag, with id "map". Notice also that we set a width and height - this is super important otherwise our map won't appear. Notice also that we load the leaflet javascript AFTER the map. This may not matter on your system, but it did matter on mine.

Now let's look at the map script, which comes after loading leaflet.js:

```
<script>
    var map = L.map('map').setView([51.505, -0.09], 13);

    L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token=pk.eyJ1Ijo
        maxZoom: 18,
        attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap</a> co
            '<a href="http://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, ' +
            'Imagery © <a href="http://mapbox.com">Mapbox</a>',
        id: 'mapbox.streets'
    }).addTo(map);
```

```
    L.marker([51.5, -0.09]).addTo(map)
        .bindPopup("<b>Hello world!</b><br />I am a popup.").openPopup();

    L.circle([51.508, -0.11], 500, {
        color: 'red',
        fillColor: '#f03',
        fillOpacity: 0.5
    }).addTo(map).bindPopup("I am a circle.");

    L.polygon([
        [51.509, -0.08],
        [51.503, -0.06],
        [51.51, -0.047]
    ]).addTo(map).bindPopup("I am a polygon.");


    var popup = L.popup();

    function onMapClick(e) {
        popup
            .setLatLng(e.latlng)
            .setContent("You clicked the map at " + e.latlng.toString())
            .openOn(map);
    }

    map.on('click', onMapClick);

    </script>
```

There is a lot to unpack here, so we'll go through it line by line:

`var map = L.map('map').setView([51.505, -0.09], 13);`

On this line we instantiate our map object and link it to the map id in our div tag. To understand this constructor in more detail, check out **the reference**.

Notice that we chain this object with a setView method - which sets the geographic coordinates that our map will start at and the zoom level. **Here is the referenece for setView**

Next we pull in a tiling. This is how the map will look. Leaflet's documentation recommends Mapbox's tiling system. So we'll make use of it. Head over to **mapbox** to get api access.

Here's how you add tiling:

```
L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token=pk.ey
    maxZoom: 18,
    attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap<,
        '<a href="http://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, ' +
        'Imagery © <a href="http://mapbox.com">Mapbox</a>',
```

```
    id: 'mapbox.streets'
}).addTo(map);
```

Basically we are mapping a request to the mapbox api with a maximum zoom, attribution, and id preset. Notice that we also pass in an access token in this case, the string: pk.eyJ1IjoibWFwYm94IiwiYSI6IjZjNmRjNzk3ZmE2MTcwOTEwMGY0MzU3YjUzOWFmNWZhIn0.Y8bhBaUMqFiPrDRW9hieoQ which clearly is randomly generated.

Notice that we are creating a tileLayer object and then adding it to our map layer. Essentially, we can think of these data structures as components of a map object is sort of like the base visual representation but it is ultimately an abstraction. The tile layer is what gives it a visual form. In this case we pull in the look of the tiling from the tile layer. And then we can super impose other structures ontop of the tile layer.

We see this pattern in the remaining objects we add to our map:

```
L.marker([51.5, -0.09]).addTo(map)
    .bindPopup("<b>Hello world!</b><br />I am a popup.").openPopup();

L.circle([51.508, -0.11], 500, {
    color: 'red',
    fillColor: '#f03',
    fillOpacity: 0.5
}).addTo(map).bindPopup("I am a circle.");

L.polygon([
    [51.509, -0.08],
    [51.503, -0.06],
    [51.51, -0.047]
]).addTo(map).bindPopup("I am a polygon.");
```

The marker, circle and polygon are all examples of objects that take up a discrete piece of the map:

- A marker takes up a single point in this case - (51.5, -0.09).
- The circle has both a center and radius, in this case - [(51.508,-0.11), 500].
- The polygon has just a set of coordinates, connected by points: [51.509,-0.08], [51.503,-0.06], [51.51, -0.047]

Notice that we simply chain the various shapes to our map data structure, and then add bindPopups to each, identifying what the object's data is. Notice of course that we could have had anything in the data since we can embed html.

Now that we understand the objects that must be loaded from the server, let's understand the elements of the code that are dynamic from the front end:

```
var popup = L.popup();

function onMapClick(e) {
    popup
        .setLatLng(e.latlng)
        .setContent("You clicked the map at " + e.latlng.toString())
        .openOn(map);
}

map.on('click', onMapClick);
```

Here we create a popup object. Notice that we make use of this popup object by wrapping it in a function that calls three of it's methods:

`setLatLng` - This method captures the latitude/longitude that was clicked and saves it to the popup object, binding the marker to the location you clicked. `setContent` - This method displays the latitude/longitude `openOn` - opens the popup object and displays the associated content

Finnally we have - `map.on('click',onMapClick)`:

This method turns on the click action for the map, and sets it to the function we created. Notice that we have one global popup object that gets changed as we click on different places in the map.

So now we understand a lot of what Leaflet.js can do, which is great. Of course, all of this was static, as far as the data goes. By making use of server, which can request different data points you can:

- filter by different data points,
- bring in different data sets,
- extrapolate data points,
- regroup the data elements dynamically, according to some clustering algorithm

Clearly this adds a whole layer of flexibility and dynamics that would otherwise be lost. So how do we do that?

Let's start with the most basic server and templated html possible:

server.py:

```
import json
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/",methods=["GET","POST"])
def index():
    states = [{
        "type":"Feature",
        "properties": {"party": "Republican"},
        "geometry": {
```

```
            "type": "Polygon",
            "coordinates": [[
                [-104.05, 48.99],
                [-97.22,  48.98],
                [-96.58,  45.94],
                [-104.03, 45.94],
                [-104.05, 48.99]
            ]]
        }
    }, {
        "type": "Feature",
        "properties": {"party": "Democrat"},
        "geometry": {
            "type": "Polygon",
            "coordinates": [[
                [-109.05, 41.00],
                [-102.06, 40.99],
                [-102.03, 36.99],
                [-109.04, 36.99],
            [-109.05, 41.00]
            ]]
        }
    }]

    return render_template("index.html",states=json.dumps(states))
```

Here we have a simple flask app, the only difference is the static geojson - stored as a python dictionary. Notice the necessary pieces - type, properties, geometry. These are all the necessary keys. The geometry key is where the lat/long actually lives. Notice that we make use of json.dumps to pass our states to the front end. After that our application is complete. As a next example, let's look at a real-time map - that is continually populated from an api.

Our API

Since we already know what geojson should look like, building an api shoudl be easy:

```
from flask import Flask, render_template, redirect, url_for
import random
import json
app = Flask(__name__)

@app.after_request
def after_request(response):
  response.headers.add('Access-Control-Allow-Origin', '*')
  response.headers.add('Access-Control-Allow-Headers', 'Content-Type,Authorization')
```

```
    response.headers.add('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE')
    return response

@app.route("/send_areas",methods=["GET","POST"])
def send_areas():
    areas = [{
        "type":"Feature",
        "properties": {"glob": "1st"},
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [[random.randint(100,106) + random.random(), random.randint(44,49) +
            ]
        }
    }, {
        "type": "Feature",
        "properties": {"glob": "2nd"},
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [[random.randint(100,106) + random.random(), random.randint(44,49) +
            ]
        }
    }]
    return json.dumps(areas)

app.run(debug=True,port=5001,threaded=True)
```

Notice that, except for the randomly generated data, everything here is exactly the same as our last example. However there is something new here:

```
@app.after_request
def after_request(response):
  response.headers.add('Access-Control-Allow-Origin', '*')
  response.headers.add('Access-Control-Allow-Headers', 'Content-Type,Authorization')
  response.headers.add('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE')
  return response
```

Since we'll be running both a server and api, we'll need to allow for cross origin from both the server and client side. We'll handle the front end piece in a second. But here is the server side - essentially this is just appending authorization to the header. Notice that we allow GET,PUT,POST, and DELETE methods. We are also giving full access via Access-Control-Allow-Origin, *. The* means all domains are allowed to access our api. Of course, we could restrict this to just a single domain or a list of domains. If this were a closed API or an api that required some kind of authentication that would be best. However, this is ran-

domly generated data so we are not in any sort of danger by making it completely open.

Now onto the front end real-time piece:

For this we'll make use of an extention of leaflet.js which can be found here - **realtime-leaflet.js**

Here is our server:

```
from flask import Flask

app = Flask(__name__)

@app.route("/realtime",methods=["GET","POST"])
def realtime():
    return render_template("realtime.html")

app.run()
```

Notice it is as simple as possible. Next let's look at realtime.html:

```
<html>
<head>
    <title>Leaflet Realtime</title>
    <link rel="stylesheet" href="http://cdn.leafletjs.com/leaflet-0.7.3/leaflet.css" />

</head>
<body>
    <div id="map" style="width: 600px; height: 400px"></div>

    <!-- source: https://github.com/perliedman/leaflet-realtime -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></script>
    <script src="http://cdn.leafletjs.com/leaflet-0.7.3/leaflet-src.js"></script>
    <script src=""></script>
    <script src=" "></script>
</body>
</html>
```

Notice that it too is very simple. Most of this is just including the necessary javascript libraries. The only file we really care about is index.js, which we'll look at next. Notice that we also include a file called leaflet-realtime.js which we got from the github repo for realtime-leaflet.js

index.js:

```
var map = L.map('map'),
    realtime = L.realtime({
        url: 'http://localhost:5001/send_areas', /*'https://wanderdrone.appspot.com/' - works
        crossOrigin: true,
        type: 'json'
```

```
    }, {
        interval: 3 * 1000
    }).addTo(map);

L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token=pk.ey
    maxZoom: 18,
    attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap<,
        '<a href="http://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, ' +
        'Imagery © <a href="http://mapbox.com">Mapbox</a>',
    id: 'mapbox.streets'
    }).addTo(map);

realtime.on('update', function() {
    map.fitBounds(realtime.getBounds(), {maxZoom: 3});
});
```

As you can see there isn't a lot new here either. The major difference is the map object which has crossOrigin set to true (allowing us to access our API like we did on the api side) setting our interval (how often we make calls out to our api) and adding our realtime map to the map object. Note that the tileLayer is exactly the same as before. Finally we turn realtime 'on' by calling realtime's on method. We set the method to update (meaning it will make calls to the api) and we pass in a function which sets some configuration. In this case the anonymous function simply sets the bounds, which are preset and then sets the maxZoom (how zoomed into the map we are).
to see our code in action you can check out [this repo] and then run the following code:

first we start up the api:

`python geo_json_api.py`

Then start the server:

python server.py

Then open a web browser and head to - **http://localhost:5000/realtime**

How might we use this? An interesting use might be setting up our api to pull from a series of open data sources, like New York City's open real time apis and marrying that with say turnstyle data. Watching how city complaints marry with city subway usage might be an interesting experiment. Of course, the content our api serves over time would need to grow, each request serving more and more content or we'd only see one data point at a time.

Section 3

Searching across massive data sets

- facial recognition, facial comparison, and image search - building a cbir
- multithreading applications for faster processing

Facial recognition, facial comparison, and image search

An important innovation in the world of computers has been search. We've more or less figured search out, and for the rest of the world, with databases, this is very useful, but for most folks working in government, things are still stored in a file system. If you want to do file search efficiently, just use **Solr**.

However, after text based documents, the second biggest need for search is images. There are some standard tools out there, but it's actually worth it to play around with your own solution. Sometimes you'll need some custom stuff, sometimes you'll need to something special, like my boss asked me to do - image recognition and search off of one picture. I had to get creative since usually results are not that good for a single picture against other pictures, which are not the same shot. To be clear my task was this:

If someone was in a picture, find them in all other pictures they appear in.

No easy feat. But I got pretty close. I ended up settling on two techniques/tools - OpenCVs face comparison algorithms and a wonderful post by the writer of **pyimagesearch**. If you haven't seen it, I recommend checking out that blog, it's full of great posts about computer vision.

So face comparison searches and compares faces against each other, producing a distance metric, the smaller the distance between two faces, IE the smaller the output, the more likely they are the same person's face. For pyimagesearch, I made use of their **backgroud comparison post**. In this post Adrian lays out how to compare the backgrounds of two images - by transforming the pictures into histograms, using a visual bag of words, and then making use of a chi^2 distance, to build an index. Finally, we search across the index to look for a picture or a similar picture in our set of pictures. From this we can see if the face and background match for a particular picture - if they do, it's probably the same person. But each of these metrics own their own is powerful for investigations. Then we can see who else has been in a the same room, and thus who else knows each other. We can also see this if we can find multiple people in the same picture.

One note - we could have used Caffe for the background image search. The trouble with Caffe is, I find it extremely difficult to install. Also, I'm not terribly good at C++, which is a major drawback. The following code is written entirely in python (or wrappers in python). Caffe does have python wrappers for some of its stuff, and it is a wonderful library if you have the computational power to truly leverage it, but most of the time in government you don't have the ability to install whatever you want and you definitely don't have the computational power to leverage such a tool.

Background compare:

The complete code with data can be found **here** `index_search.py`:

```
import numpy as np
import cv2
import argparse
import csv
from glob import glob


args = argparse.ArgumentParser()
args.add_argument("-d","--dataset",help="Path to the directory that contains the ima
args.add_argument("-i","--index",help="Path to where the computed index will be stor
args.add_argument("-q","--query",help="Path to the query image")
args.add_argument("-r","--result-path",help="Path to the result path")

args = vars(args.parse_args())

class ColorDescriptor:
    def __init__(self,bins):
        self.bins = bins

    #off_center means that we expect no people in the center of the picture
    #this is the feature construction and processing function
    #cX,cY stand for center X and center Y, aka the middle of the picture
    #off_center creates four quadrants for the picture
    #full_picture checks the full picture and only creates one histogram per picture
    #face_compare creates a segmentation around possible faces, and uses only the fa
    def describe(self,image,off_center=False,full_picture=False,face_compare=False)
        image = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
        features = []
        (h,w) = image.shape[:2]
        (cX,cY) = (int(w*0.5), int(h*0.5))
        segments = [(0,cX,0,cY),(cX,w,0,cY),(cX,w,cY,h),(0,cX,cY,h)]
        if off_center:
            for (startX,endX,startY,endY) in segments:
                cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
                hist = self.histogram(image,cornerMask)
                features.extend(hist)
        elif full_picture:
            hist = self.histogram(image,None)
            features.extend(hist)
        elif face_compare:
            cascPath = "haarcascade_frontalface_default.xml"
            faceCascade = cv2.CascadeClassifier(cascPath)
            #gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            # Detect faces in the image
            faces = faceCascade.detectMultiScale(
                image,
```

```
                    scaleFactor=1.1,
                    minNeighbors=2,
                    minSize=(25, 25),
                    flags = cv2.cv.CV_HAAR_SCALE_IMAGE
                )

                for (x,y,w,h) in faces:
                    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                    cv2.rectangle(cornerMask, (x,y), (x+w, y+h), 255, -1)
                    hist = self.histogram(image,cornerMask)
                    features.extend(hist)
            else:
                ellipMask = np.zeros(image.shape[:2],dtype="uint8")
                (axesX, axesY) = (int(w * 0.75) / 2, int(h * 0.75) / 2)
                cv2.ellipse(ellipMask, (cX, cY), (axesX,axesY), 0,0,360,255,-1)
                for (startX,endX,startY,endY) in segments:
                    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                    cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
                    cornerMask = cv2.subtract(cornerMask,ellipMask)
                    hist = self.histogram(image,cornerMask)
                    features.extend(hist)
            return features

        def histogram(self,image,mask):
            hist = cv2.calcHist([image],[0,1,2],mask,self.bins,
            [0, 180, 0, 256, 0, 256])
            hist2 = cv2.normalize(hist,np.zeros(image.shape[:2],dtype="uint8")).flatten()
            return hist2

    class Searcher:
        def __init__ (self,indexPath):
            self.indexPath = indexPath
        def search(self, queryFeatures, limit=10):
            results = {}
            with open(self.indexPath) as f:
                reader = csv.reader(f)
                try:
                    for row in reader:
                        features = [float(x) for x in row[1:] if x!= '']
                        d = self.chi2_distance(features, queryFeatures)
                        results[row[0]] = d
                except IndexError:
                    pass
            results = sorted([(v,k) for (k,v) in results.items()])
            return results[:limit]
        def chi2_distance(self,histA,histB, eps = 1e-10):
            d = 0.5 * np.sum([((a-b)**2) / (a+b+eps)
                              for (a,b) in zip(histA,histB)])
            return d
```

```
if __name__ == "__main__":
    #i have no idea where these numbers came from.. check - http://www.pyimagesearch
    cd = ColorDescriptor((8,12,3))

    if args["index"] and args["dataset"]:
        with open(args["index"],"w") as output:
            for img_type in [".png",".jpg",".PNG",".JPG",".img",".jpeg",".jif",".jf
                for imagePath in glob(args["dataset"] + "/*" + img_type):
                    imageID = imagePath[imagePath.rfind("/") + 1:]
                    image = cv2.imread(imagePath)
                    features = cd.describe(image)
                    features = [str(f) for f in features]
                    output.write("%s,%s\n" % (imageID, ",".join(features)))
    if args["index"] and args["query"]:
        query = cv2.imread(args["query"])
        features = cd.describe(query)
        searcher = Searcher(args["index"])
        results = searcher.search(features,limit=4)

        cv2.imshow("query",query)
        for (score,resultID) in results:
            result = cv2.imread(args["result_path"] + "/" + resultID)
            cv2.imshow("Result",result)
            cv2.waitKey(0)
```

Rather than explaining all this code, because Adrian explains all of it and extremely well, I'll simply explain how to use it:

```
python index_search.py -d [directory of pictures] -i [name of file] #create an index
```

```
python index_search.py -i [name of index file] -q [path to query picture] -r [path to picture directory] #search for a picture
```

A specific example:

```
python index_search.py -d pic_db/ -i index.csv
```

```
python index_search.py -i index.csv -q person.jpg -r pic_db/
```

So here's how the code works:

First let's look at the main method:

```
cd = ColorDescriptor((8,12,3))

if args["index"] and args["dataset"]:
    with open(args["index"],"w") as output:
        for img_type in [".png",".jpg",".PNG",".JPG",".img",".jpeg",".jif",".jfif",
            for imagePath in glob(args["dataset"] + "/*" + img_type):
                imageID = imagePath[imagePath.rfind("/") + 1:]
                image = cv2.imread(imagePath)
```

```
                features = cd.describe(image)
                features = [str(f) for f in features]
                output.write("%s,%s\n" % (imageID, ",".join(features)))
if args["index"] and args["query"]:
    query = cv2.imread(args["query"])
    features = cd.describe(query)
    searcher = Searcher(args["index"])
    results = searcher.search(features,limit=4)

    cv2.imshow("query",query)
    for (score,resultID) in results:
        result = cv2.imread(args["result_path"] + "/" + resultID)
        cv2.imshow("Result",result)
        cv2.waitKey(0)
```

As you can see there isn't a ton going on here, we loop through all the images in the pictures folder, doing the feature transformations on each picture and then generating an index with all the feature-histogram-vectors. Now let's dig a little deeper into the describe method. The describe method acts on the matrix representation of an image. The image is a matrix at this point, because it was read in with open cv's imread method. So really the describe method is a matrix transformation, which produces features according to some rules about the image. The way in which you decide to describe images, turns out to be very very important. Essentially the describe is doing a segmentation and then a transformation.

One fun thing I did was, I allowed you to segment pictures by face, into four corners, or the way Adrian does it; with four corners and an elliptical in the center. My ways don't always yield good results, and in fact more than half the time Adrians method works the best, but sometimes, my methods have been effective so I will mention them here.

To understand that we'll look at the segmentation methods one at a time:

```
for (startX,endX,startY,endY) in segments:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Here we setup a cornerMask which is a matrix of zeroes. Notice that we apply the rectangle method, which takes in starting and finishing coordinates and then 'draws' a rectangle on the cornerMask. Then we create a histogram with the segmentation of the image defined by the rectangle we 'drew' which acts as sort of a boundary. Depending on how you draw your segmentation, will effect your histogram and thus your features.

The full picture is uninteresting, because it is just the histogram transformation with no segmentation.

The next interesting segmentation is one done with semantic meaning built in - finding the startX,startY and endX,endY from the faces in the picture:

```
cascPath = "haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascPath)
#gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Detect faces in the image
faces = faceCascade.detectMultiScale(
    image,
    scaleFactor=1.1,
    minNeighbors=2,
    minSize=(25, 25),
    flags = cv2.cv.CV_HAAR_SCALE_IMAGE
)

for (x,y,w,h) in faces:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask, (x,y), (x+w, y+h), 255, -1)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Here we use opencv's face detection classifier to find all the 'boxes' around the potential faces in the image. This is then used to create our rectangles, similarly to the previous example. However, now rather than segmenting on corners, we segment on faces. We then use this segmentation to create histograms and finally feature vectors of just the faces. Unfortunately, in practice this technique tends to perform the worst, for now. I need to figure out how to better tune my parameters so that this is reliable, or I may abandon it as an idea. But still, it was fun to try!

The final method is what Adrian did:

```
ellipMask = np.zeros(image.shape[:2],dtype="uint8")
(axesX, axesY) = (int(w * 0.75) / 2, int(h * 0.75) / 2)
cv2.ellipse(ellipMask, (cX, cY), (axesX,axesY), 0,0,360,255,-1)
for (startX,endX,startY,endY) in segments:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
    cornerMask = cv2.subtract(cornerMask,ellipMask)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Creating an elliptical mask for the center of the picture and generating segments for the remaining four corners and then subtracting the center from the four corners. Nothing else is new here and thus it can largely be ignored.

The next important piece of the main function is performing a query which is captured here:

```
query = cv2.imread(args["query"])
features = cd.describe(query)
searcher = Searcher(args["index"])
results = searcher.search(features,limit=4)
```

Notice that we describe the query image the same way we described the index. This is imperative - if we describe our feature vectors differently in indexing and querying, our queries won't work.

The search function is fairly simple:

```
results = {}
with open(self.indexPath) as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            features = [float(x) for x in row[1:] if x!= '']
            d = self.chi2_distance(features, queryFeatures)
            results[row[0]] = d
    except IndexError:
        pass
results = sorted([(v,k) for (k,v) in results.items()])
return results[:limit]
```

Essentially, it applies the chi^2 distance function to each of the features in the index against the query image's features. The results are returned in sorted order, lowest to highest. This is because if the distance is smallest, they are the images can be expected to be the least different.

Understanding Face Comparison, With OpenCV

The `face_compare2.py` contains how I do face comparison with open CV. It works quiet well, but I do confess it is not my most well organized piece of code. I have not yet learned how to be elegant with image manipulation, alas, the struggles of perfection in a sea of desires and things one aspires to be perfect at. But such is the calamity of curious minds, and it is something I am proud of, to be pulled in so many possible directions.

Note: For this code to work from this repo, you'll need to install OpenCV from the follow githubs: **opencv**, **opencv-contrib**, and it's not a bad idea to install **opencv-extra**, but only if you have space for the extra data, it's not necessary.

And the code that I'll reference throughout this section can be found **here**

In any event, there is a lot to unpack in that file and so I'm simply going to explain some high level steps:

First we instantiate our recognizers:

```
recog = {}
recog["eigen"] = cv2.face.createEigenFaceRecognizer()
recog["fisher"] = cv2.face.createFisherFaceRecognizer()
recog["lbph"] = cv2.face.createLBPHFaceRecognizer()
```

Notice that recently OpenCV has changed and now the face recognizers live in a seperate supplemental library. The core of OpenCV can be found here: **https://github.com/Itseez/opencv** and the face recognizers can be found here: **https://github.com/Itseez/opencv_contrib**. I'm not sure why they decided to move such a handy set of tools out into some contrib repo, but that is how it is.

The next thing to do is normalize the pictures:

normalize_cv(filename,compare,directory_of_pictures)

In this case, this means making them all the same height and width, because the face comparison algorithms require this. I'm not sure why this is the case.

Next we read in a picture and hand label it 1:

```
face = cv2.imread(new_filename,0)
face,label = face[:, :], 1
```

And we read in a different picture with a different face and hand label it 2:

```
compare_face = cv2.imread(new_compare, 0)
compare_face, compare_label = compare_face[:,:], 2
```

This establishes the base comparison for what will be considered a face we are searching for and a face that is different from the one we want. Perhaps using 2 isn't the best, because the distance isn't great enough, but this is intended to be a toy example, you should feel free to tune these "magic" numbers to improve the precision of your own code.

Next we train the each recognizer on the training data:

```
for recognizer in recog.keys():
    recog[recognizer].train(image_array,label_array)
```

And then we compare all the trained data against the directory of pictures:

```
test_images = [(np.asarray(cv2.imread(img,0)[:,:]),img) for img in test_images]
possible_matches = []
for t_face,name in test_images:
    t_labels = []
    for recognizer in recog.keys():
```

```
        try:
            [label, confidence] = recog[recognizer].predict(t_face)
            possible_matches.append({"name":name,"confidence":confidence,"recognizer":recogni
```

Notice that we simply need to call the predict function on each image in our directory.

Finally, we simply can print out all the possible matches:

```
for i in possible_matches:
    print i
```

Running this code is fairly simple and can be accomplished with the following:

`python face_compare2.py [first picture] [baseline comparison picture] [directory of pictures to search against]`

Multithreading applications

While Python has a Global Interpretter Lock (GIL), meaning it cannot be truly multithreaded, there are a lot of things you can still do that look and feel like multithreading. And you can absolutely parallelize your code. For more on the GIL check out these sources:

- **PythonWiki**
- **These awesome slides from David Beazley**
- **This sweet general purpose wikipedia article on GILs**

If you are new to doing multiple things in one program, I'll just give you the two second crash course. Feel free to skip ahead if this is review.

What's all this then?

Since we can't truly multithread things in python we'll need to think of ways around the problem. One of my favorite ways to get around the GIL is with mutliprocessing's Process object and the Queue. More or less what happens is we set up things like we would in a normal for-loop except we don't wait for things to finish before moving onto the next iteration of the loop. The way this works is our program spins up different workers that execute in subprocesses. Of course, the program still finishes, but it may not finish in the sequential order we coded it, so we need to be careful that order doesn't matter. In order to make use of this type of program, we'll need a problem where individual pieces are computationally hard and where order doesn't matter. Fortunately I have just the

problem - figuring out what neighborhood you are in, based on your latitude and longitude. For this we'll make use of Zillow's categorization of all the different neighborhoods of New York City. They have lists for many such cities and states across America. But figuring out where you are can be an extremely difficult task (sometimes intellectually as much as physically).

Here is the code in all it's glory:

```
import pandas as pd
import shapefile
# http://www.digital-geography.com/importing-shapefiles-in-python/#.VfBKBXUViko
from glob import glob
from shapely.geometry import Point, MultiPoint
#check out http://streamhacker.com/2010/03/23/python-point-in-polygon-shapely/, http
from unidecode import unidecode
import time
import random
from multiprocessing import Process,Queue
#where I got the data: http://www.zillow.com/howto/api/neighborhood-boundaries.htm

#This method transforms the shape file from zillow and finds all the points for the
#different neighborhoods around the city
#This can easily be extended to find any neighborhood in the zillow dataset
#This method returns a list of geo objects which come from the shapefile object, che
def grab_nyc_neighborhoods():
    filename = glob("*.shp")
    ctr = shapefile.Reader(filename[0])
    geomet = ctr.shapeRecords()
    nyc = []
    for geo in geomet:
        if any([place for place in geo.record if type(place) == type(str()) and "New
           nyc.append(geo)
    return nyc

#This method determines whether or not a give point is inside a neighborhood by loop
#and then just stating whether or not it's in the neighborhood or not
#neighborhoods is a list of geo objects which come from the shapefile object - check
def determine_neighborhood(nyc,location):
    point = Point((location["lat"],location["long"]))
    neighborhoods = []
    for geo in nyc:
        points = [tuple([ind_point[1],ind_point[0]]) for ind_point in geo.shape.poin
        poly = MultiPoint(points).convex_hull
        if point.within(poly):
            neighborhoods.append(geo)
    return neighborhoods

#This method is the main workhorse of the program
#It figures out what neighborhood each point is in
#It then passes back a dataframe which will be saved when all pieces of information
```

```
def create_neighborhood_grouping(nyc,location,q):
    tmp_dict = {}
    neighborhood = determine_neighborhood(nyc,location)
    if len(neighborhood) >= 1:
        neighborhood = neighborhood[0]
        tmp_dict["neighborhood"] = neighborhood.record
    else:
        tmp_dict["neighborhood"] = ""
    tmp_dict.update(location)
    q.put(tmp_dict)

#This method processes the csv
def process_csv(doc):
    df = pd.DataFrame().from_csv(doc,index_col=False)
    locations = []
    for i in df.index:
        locations.append({"lat":df.ix[i]["Latitude"], "long":df.ix[i]["Longitude"],})
    return locations

#As you can see here we have a main ->
#The main gets the locations, saves them to a dataframe and then saves that dataframe to a csv
if __name__ == '__main__':
    nyc = grab_nyc_neighborhoods()
    print "getting lat/long"
    locations = process_csv("random_lat_long.csv")
    df = pd.DataFrame()
    print "creating dataframe"
    results = []
    q = Queue()
    for ind,loc in enumerate(locations):
        print loc, ind
        p = Process(target=create_neighborhood_grouping,args=(nyc,loc,q))
        p.start()
        results.append(q.get())
    for result in results:
        df = df.append(result,ignore_index=True)
    df.to_csv("grouping.csv")
```

There is a lot going on in 79 long file. So we'll start with main:

We get the Process and Queue objects from mutliprocessing. By wrapping each function call in a process object - p we are able to execute each call to the function without having to wait for the previous call to finish. This allows us to spin up as many workers as needed - the average on my laptop is 7 with a high of 15 and a low of 5. These workers allow individual tasks, in this case, finding whcih neighborhood a randomly selected point in New York City, belongs to. Once we have our function call and arguments sent to our object at instantiation, the next thing to do is start the process via `p.start()`. Notice that we also pass in a q object. Our queue object will act as a store for our results. The won-

derful thing about the queue is it can serve as a mechanism for passing messages, in this case the results of our function call, create_neighborhood_grouping. More or less what happens inside a process is the result is pickled. However this makes it hard to retrieve from by parent process, thus we store the results in the queue which is passed up to the parent process upon completion of the function call. Then the queue unpickles the result, effectively messaging the result as a message to the results list, which then is passed into a dataframe.

For more information on this process I highly encourage checking out:

- **Python's Introduction to multiprocessing**
- **Wonderful how to for multiprocessing**

Both have simple examples of multiprocessing and explain much of the high level idea.

Next we'll look at how to figure out if a point is inside an area:

```
def determine_neighborhood(nyc,location):
    point = Point((location["lat"],location["long"]))
    neighborhoods = []
    for geo in nyc:
        points = [tuple([ind_point[1],ind_point[0]]) for ind_point in geo.shape.poi
        poly = MultiPoint(points).convex_hull
        if point.within(poly):
            neighborhoods.append(geo)
    return neighborhoods
```

The MultiPoint and Point objects both come from `shapely.geometry` a python module for doing GIS computations. Once we have a point object (which we pass in the latitude and longitude as above) and we have an area of interest, we can simply check the within method of the point to see if it lies inside the polygon in question. Notice that we make use of the convex_hull representation of the polygon. This will fill in the entire polygon, rather than just looking at the boundary of the polygon. Therefore if we are inside the polygon as well as on the edge of the polygon, we'll know it. Notice also this line:

`points = [tuple([ind_point[1],ind_point[0]]) for ind_point in geo.shape.points]`

The reason we display the points as such is, for some reason shape file is organized in longitude, latitude so we reorder it for latitude, longitude.

The only thing left to understand is reading in the shapefile, to get the neighborhoods in the first place:

```
def grab_nyc_neighborhoods():
    filename = glob("*.shp")
    ctr = shapefile.Reader(filename[0])
    geomet = ctr.shapeRecords()
```

```
nyc = []
for geo in geomet:
    if any([place for place in geo.record if type(place) == type(str()) and "New York City
       nyc.append(geo)
return nyc
```

Here we make use of the shapefile library (there is a reference in the above code on how to install it). This is used to read in any shapefiles. Then we simply loop through the records looking for new york city specific records.

Running this process sequentially takes far longer than running it in parallel. One of the major failings of government work is a lack of distributed computing. The hope of this last section and really this text as a whole is that it will make your work faster and easier, assuming you want or have a government job. Government work is far too slow. We need justice and often times we need it yesterday.

Conclusion

If you found this text interesting or useful, I encourage you to check out my long form of this text - **https://github.com/EricSchles/booktools**

It may end up being a full book one day. For now it is the accumulation of some of my learnings and collected talks from the past few years. At present it is unfinished by likely will be at some point.

And finally - if you want to get involved further I recommend contacting me. With the support of my friends, I will be holding a series of hackathons to get tools into the hands of those looking to fight slavery. If my response to your individual query or ask for help is slow, it is because I am one person and starting out on what I hope will be a life long journey towards making a difference in the world.

Contact info: ericschles@gmail.com or twitter - @EricSchles

Appendix Title A

This Is an A-Head

An appendix is generally used for extra material that supplements your main book content.

Index