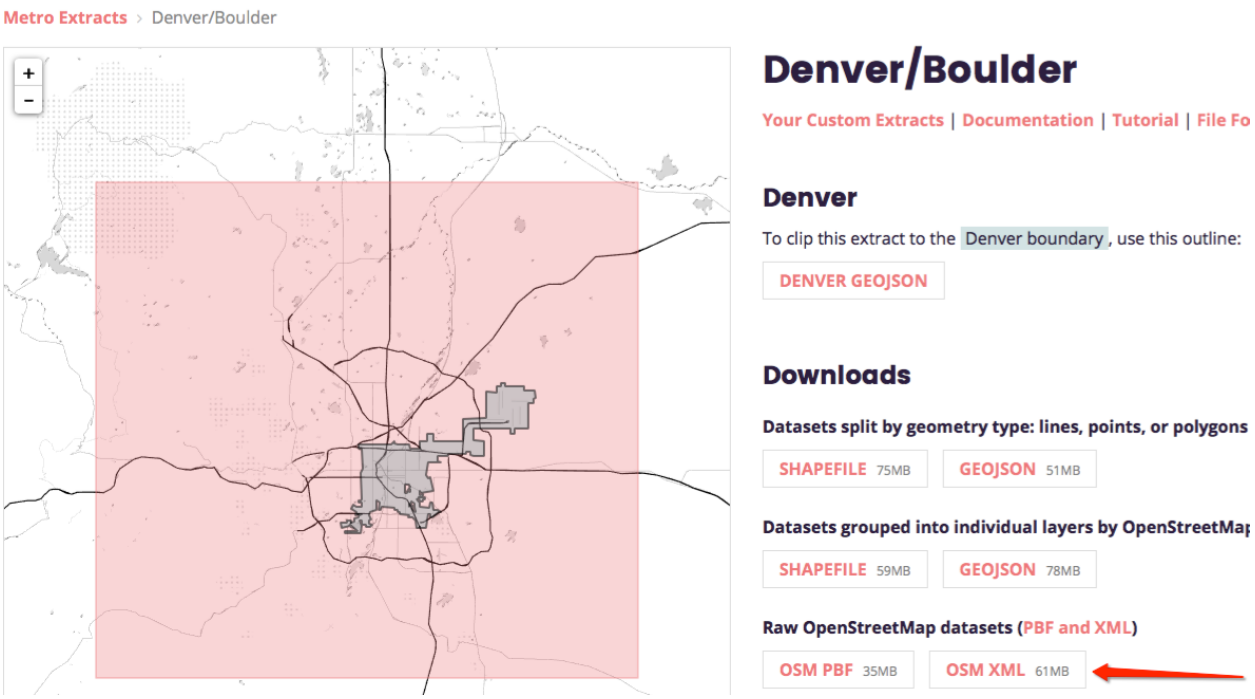
OPEN STREET MAP DATA WRANGLING WITH SQL

Awad Bin-Jawed

I will demonstrate the process of data wrangling for the city of **Denver, CO, USA**. This project will be an iterative process, so hopefully I will get a desired result as accurate as possible after an extensive cycle of repeating operations. This is why we say this whole cycle is convergent.

I extracted the OSM XML file of Denver.

I will need to audit the street types and standardize the data; this way, I will know exactly what to extract.
Mapzen URL https://mapzen.com/data/metro-extracts/metro/denver- boulder_colorado/85928879/Denver/



DATA AUDITING

The **audit.py** script generates two outputs:

1) **audit.txt**
2) **audit_xsd.txt**

The **audit.txt** contains captured output of 6 functions:

print(str(audit_street(OSM_FILE)))
fix_street(audit_street(OSM_FILE))
print(str(audit_city(OSM_FILE)))
print(str(audit_postcode(OSM_FILE)))
output.write(str(audit_phone(OSM_FILE)))
fix_phones(phone_list)

The **audit_xsd.txt** contains the result of validation of the full map set.

Misspelled/Abbreviated variables:

I added “What is Expected” and updated the inconsistencies:

expected = ["Street", "Avenue", "Boulevard", "Drive", "Court","Place", "Square", "Lane", "Road", "Trail", "Parkway", "Commons"]

In addition, I added the mapping:

```
mapping = { "Av": "Avenue",
            "Ave": "Avenue",
            "Ave.": "Avenue",
            "Avenue)": "Avenue"
            "Baselin": "Baseline",
            "Blf": "Boulevard",
            "Blvd": "Boulevard",
            "Blvd.": "Boulevard",
            "Cir.": "Circle",
            "Ct": "Court",
            "Dr": "Drive",
            "Hwy": "Highway",
            "Ln": "Lane",
            "Pkwy": "Parkway",
            "Pky": "Parkway",
            "Rd": "Road",
            "Rd.": "Road",
            "St": "Street",
            "St.": "Street"
            "Strret": "Street"
            "Thornton,": "Thornton"
            }
```

Abbreviations:

Av needs to be converted to Avenue
Ave needs to be converted to Avenue
Ave. needs to be converted to Avenue
Avenue) needs to be converted to have the right parenthesis “)” removed
Baselin needs to be converted to Baseline
Blf needs to be converted to Boulevard
Blvd needs to be converted to Boulevard
Blvd. needs to be converted to Boulevard
Cir. needs to be converted to Circle
Ct needs to be converted to Court
Dr needs to be converted to Drive
Hwy needs to be converted to Highway
Ln needs to be converted to Lane
Pkwy needs to be converted to Parkway
Pky needs to be converted to Parkway
Rd needs to be converted to Road
Rd. needs to be converted to be Road
St needs to be converted to Street
St. needs to be converted to Street
Strret needs to be converted Street

```
# update street name according to the mapping dictionary
#def fix_street(osm_file):
def fix_street(st_types):
    # st_types = audit_street(osm_file)
    for st_type, ways in st_types.iteritems():
        for name in ways:
            if st_type in mapping:
                better_name = name.replace(st_type, mapping[st_type])
                print name, "=>", better_name


def audit_city(osm_file):
    city_list = set()

    context = iter(ET.iterparse(osm_file, events=("start", "end")))
    _, root = next(context) # root.tag == "osm"
    for event, elem in context:
        if event == "start":
            if elem.tag == "node" or elem.tag == "way":
                for tag in elem.iter("tag"):
                    if tag.attrib['k'] == "addr:city" and tag.attrib['v'] !=
"Denver":
                        city_list.add(tag.attrib['v'])
        if event == "end":
            root.clear()
    return city_list


# zipcode data from http://www.unitedstateszipcodes.org/zip-code-database/
zipcode = pd.read_csv("zipcode.csv")
denver_zipcode = zipcode[(zipcode.primary_city == "Denver") & (zipcode.state ==
"CO")].zip


denver_zipcode_str = [str(x) for x in list(denver_zipcode)]


def audit_postcode(osm_file):
    code_list = set()
    long_code = 0
```

```
    context = iter(ET.iterparse(osm_file, events=("start", "end")))
    _, root = next(context) # root.tag == "osm"
    for event, elem in context:
        if event == "start":
            if elem.tag == "node" or elem.tag == "way":
                for tag in elem.iter("tag"):
                    if tag.attrib['k'] == "addr:postcode":
                        if len(tag.attrib['v']) > 5:
                            long_code += 1
                            tag.attrib['v'] = tag.attrib['v'].split('-')[0]
                        code_list.add(tag.attrib['v'])
        if event == "end":
            root.clear()
    print 'There are', long_code, 'long post codes.'
    return [code for code in code_list if code not in denver_zipcode_str]


def audit_phone(osm_file):
    phone_list = []


    context = iter(ET.iterparse(osm_file, events=("start", "end")))
    _, root = next(context) # root.tag == "osm"
    for event, elem in context:
        if event == "start":
            if elem.tag == "node" or elem.tag == "way":
                for tag in elem.iter("tag"):
                    if (tag.attrib['k'] == "phone") or (tag.attrib['k'] ==
"contact:phone"):
                        phone_list.append(tag.attrib['v'])
        if event == "end":
            root.clear()
    return phone_list


# standard format: XXX-XXX-XXXX or 1-XXX-XXX-XXXX
def fix_phone(phone):
    # first deal with strings with no separators
    if re.compile(r'^(\d{3})(\d{3})(\d{4})$').search(phone):
        # XXXXXXXXXX
        return phone[:3] + '-' + phone[3:6] + '-' + phone[6:]
    elif re.compile(r'^1(\d{10})$').search(phone):
       # 1XXXXXXXXXX
```

```
            return phone[1:4] + '-' + phone[4:7] + '-' + phone[7:]
        elif re.compile(r'^\+1(\d{10})$').search(phone):
            # +1XXXXXXXXXX
            return phone[2:5] + '-' + phone[5:8] + '-' + phone[8:]
        elif re.compile(r'^\+1\s(\d{10})$').search(phone):
            # +1 XXXXXXXXXX
            return phone[3:6] + '-' + phone[6:9] + '-' + phone[9:]
        elif re.compile(r'^\+1\s(\d{3})\s(\d{3})(\d{4})$').search(phone):
            # +1 XXX XXXXXXX
            return phone[3:6] + '-' + phone[7:10]+ '-' + phone[10:]
        elif re.compile(r'^01\s(\d{3})\s(\d{3})\s(\d{4})$').search(phone):
            # 01 XXX XXX XXXX
            # return phone[3:].replace(' ', '-')
            return phone[3:6] + '-' + phone[7:10]+ '-' + phone[11:]


        # if more than one number in the string, split to a list
        elif phone.find(';') > 0:
            for ph in phone.split(';'):
                fix_phone(ph)

        # if there are 3 or 4 digital parts, concatenate with dash
        # ????
        elif len(re.findall('\d+', phone)) > 2:
            return '-'.join(re.findall('\d+', phone))


        else:
            print phone


def fix_phones(phone_list):
    for phone in phone_list:
        fix_phone(phone)


# validate the XML OSM file via schema
def validator(osm_file, schema):
    xmlschema_doc = lxml_ET.parse(schema)
    xmlschema = lxml_ET.XMLSchema(xmlschema_doc)


    for event, element in lxml_ET.iterparse(osm_file, events=("end", )):
```

```
if not xmlschema.validate(element):
    print xmlschema.error_log, element.attrib
element.clear()
if element.tag in ["node", "way", "relation"]:
    for ancestor in element.xpath("ancestor-or-self::*"):
        while ancestor.getprevious() is not None:
            del ancestor.getparent()[0]
```

The following print statement:

```
for st_type, ways in st_types.iteritems():
    for name in ways:
        better_name = update_name(name, mapping)
        if name != better_name:
            print name, "=>", better_name
            name = better_name
```

This will print the street names after the update of the original, unclean street names.
Essentially, the goal of the scripts is to analyze the data of Denver.
Using the editor (in this case Atom), you may manually analyze the large .osm file.
There is a method run() in all files which can be called from report.py
or in an interpreter.

The function of shape_element() in **data.py** is it converts a first level **<node>** or **<way>** element from a OSM XML file into a dictionary.

It also tests second level **<tag>** elements on problematic characters and converts second level **<tag>** or **<node>** elements into subdictionaries.

If the element tag is "node", it returns a dictionary in the format {"node": {'id': 1234, ...}, "node_tags": [{'id': 4321, ...}, ...] }.

If the element tag is "way", it returns a dictionary in the format {"way": {'id': 1234, ...}, "way_nodes": [{'id': 4321, ...}, ...], "way_tags": [{'id': 1423,...}, ...] }.

The line in **data.py** that can be replaced….

```
###
def run():
SAMPLE_PATH = OSM_PATH[0:-4] + '_sample.osm'
process_map(SAMPLE_PATH, validate=True)
#run()
###
```

with a common block:

```
if __name__ == “___main___”:
SAMPLE_PATH = OSM_PATH[0:-4] + '_sample.osm'
process_map(SAMPLE_PATH, validate=True)
```

But in my experience, I found the data would not be better processed. If I wish to process larger OSM_PATH, I can write process_map(OSM_PATH, validate=False). The data is valid in this sense, but the validation in this script is very slow.

mapparser.py
This script will yield a defaultdic dictionary of tags and occurrences.

users.py
This script will yield a defaultdic dictionary of user's id numbers.

tagtype.py
This script will yield a simple dictionary of categorized <tag> tags. It also outputs problematic tags.

After auditing, the next step is to prepare the data for insertion to a SQL database. We parse the elements in the OSM XML file, which will transform them from .doc format to tabular format. The process involved using the iterparse through the elements in the XML and shaping the elements into several structures. The schema was to ensure the data is in the correct format. Next we attempt to write each data structure to the appropriate .csv files.

DATA OVERVIEW

File Sizes:

```
denver-boulder_colorado.osm ..... 68  MB
nodes.csv ....................... 4.1 MB
nodes_tags.csv: ................. .171 MB
ways.csv: ....................... .327 MB
ways_nodes.csv: ................. 1.3 MB
ways_tags.csv: .................. .659 MB
```

Upon executing **query.py**, on the respective .csv files, we are able to output statistical overview of the dataset:

```
import
sqlite3
          import csv

          # open database
          con = sqlite3.connect('db.sqlite')
          cur = con.cursor()
          # Step 1.1 - import nodes



          # Creating table for nodes
          cur.execute('DROP TABLE IF EXISTS "nodes"')  # clear previous table if it exists
          cur.execute('CREATE TABLE "nodes" ('
                      'id INTEGER,'
                      'lat REAL,'
                      'lon REAL,'
                      'user TEXT,'
                      'uid INTEGER,'
                      'version INTEGER,'
                      'changeset INTEGER,'
                      'timestamp DATE )')
```

```
# reading "nodes.csv" and put every row in database
with open('nodes.csv') as f:
    reader = csv.reader(f)
    next(reader)  # skip header
    for row in reader:
        if len(row) == 0:  # skipping empty rows
            continue
        cur.execute('INSERT INTO nodes VALUES (?, ?, ?, ?, ?, ?, ?, ?)', row) # put
in database


# saving database
con.commit()
# Step 1.2 - import nodes_tags
# Creating table for nodes_tags
cur.execute('DROP TABLE IF EXISTS "nodes_tags"')
cur.execute('CREATE TABLE "nodes_tags" ('
            'id INTEGER,'
            'key TEXT,'
            'value TEXT,'
            'type TEXT)')


# reading "nodes_tags.csv" and put every row in database
with open('nodes_tags.csv') as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        if len(row) == 0:
            continue
        cur.execute('INSERT INTO nodes_tags VALUES (?, ?, ?, ?)', row)
```

```
# saving
con.commit()

# Step 1.3 - import ways

# Creating table for ways
cur.execute('DROP TABLE IF EXISTS "ways"')
cur.execute('CREATE TABLE "ways" ('
            'id INTEGER,'
            'user TEXT,'
            'uid INTEGER,'
            'version INTEGER,'
            'changeset INTEGER,'
            'timestamp DATE )')


# reading "ways.csv" and put every row in database
with open('ways.csv') as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        if len(row) == 0:
            continue
        cur.execute('INSERT INTO ways VALUES (?, ?, ?, ?, ?, ?)', row)

# saving
con.commit()

# Step 1.4 - import ways_nodes

# Creating table for ways_nodes
cur.execute('DROP TABLE IF EXISTS "ways_nodes"')
cur.execute('CREATE TABLE "ways_nodes" ('
            'id INTEGER,'
            'node_id INTEGER,'
            'position INTEGER )')
```

```
# reading "ways_nodes.csv" and put every row in database
with open('ways_nodes.csv') as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        if len(row) == 0:
            continue
        cur.execute('INSERT INTO ways_nodes VALUES (?, ?, ?)', row)

# saving
con.commit()

# Step 1.5 - import ways_tags

# Creating table for ways_tags
cur.execute('DROP TABLE IF EXISTS "ways_tags"')
cur.execute('CREATE TABLE "ways_tags" ('
            'id INTEGER,'
            'key TEXT,'
            'value TEXT,'
            'type TEXT)')

# reading "ways_tags.csv" and put every row in database
with open('ways_tags.csv') as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        if len(row) == 0:
            continue
        cur.execute('INSERT INTO ways_tags VALUES (?, ?, ?, ?)', row)

# saving
con.commit()

# Step 2.1 computing number of nodes
cur.execute('SELECT COUNT(id) FROM nodes')
amount = cur.fetchone()[0]  # extract first element from row
print(u'Number of nodes: {}'.format(amount))

# Step 2.2 computing number of nodes_tags
cur.execute('SELECT COUNT(id) FROM nodes_tags')
amount = cur.fetchone()[0]
print(u'Number of nodes_tags: {}'.format(amount))
```

```
# Step 2.3 computing number of ways
cur.execute('SELECT COUNT(id) FROM ways')
amount = cur.fetchone()[0]
print(u'Number of ways: {}'.format(amount))

# Step 2.4 computing number of ways_nodes
cur.execute('SELECT COUNT(id) FROM ways_nodes')
amount = cur.fetchone()[0]
print(u'Number of ways_nodes: {}'.format(amount))

# Step 2.5 computing number of ways_tags
cur.execute('SELECT COUNT(id) FROM ways_tags')
amount = cur.fetchone()[0]
print(u'Number of ways_tags: {}'.format(amount))

# Step 2.6 computing number of unique users
cur.execute('SELECT COUNT(DISTINCT "user") FROM nodes')  # at this point I use
"DISTINCT" keyword, in order to compute number of unique users
amount = cur.fetchone()[0]
print(u'Number of unique users: {}'.format(amount))

# Step 2.7 computing number of cafes
cur.execute('SELECT COUNT(id) FROM nodes_tags WHERE value="cafe"')  # skips
everything not a cafe
amount = cur.fetchone()[0]
print(u'Number of cafes: {}'.format(amount))

# Step 2.8 computing number of restaurants
cur.execute('SELECT COUNT(id) FROM nodes_tags WHERE value="restaurant"')  # skips
everything not a restaurant
amount = cur.fetchone()[0]
print(u'Number of restaurants: {}'.format(amount))
```

```
Number of nodes ..... 46827
Number of nodes_tags ..... 4628
Number of ways: ..... 5189
Number of ways_nodes: ..... 52491
Number of ways_tags: ..... 18812
Number of unique users: ..... 853
Number of cafes ..... 5
Number of restaurants ..... 28
```

Problems Encountered:

sample.py

This script had a broken format:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm>
        <node changeset="7551724" ... />
                <node />
....
                <tag k="type" v="restriction" />
        </relation>
        </osm>
```

This created problems, which were due to output.write() breaking indentations.

The first one can be fixed by output.write('<osm>\n\t'), the second one can be fixed by manually editing the last line in the new file.

data.py

This program creates 4 csv files from denver-boulder_colorado_sample.osm

to be added to the database. The validator in it WILL NOT raise errors because

it doesn't check the nested tags.

The xml+cerberus+schema code is very slow. It is much slower than the lxml+.xsd code.

Additional Ideas

The state of Colorado has recently been grabbing headlines due to a controversial legislative action... the legalization of recreational marijuana in 2012. The Colorado Amendment 64 was a successful measure to overrule the longstanding drug policy of Colorado's own constitution, which has kept prohibited cannabis since 1917.
It would be interesting to map out cannabis and crime in the state of Colorado. More specifically, the Denver metro area.

The crime CSV file, extracted from the Denver Open Data Catalog comprised of data sets of criminal offenses during the past 5 years (since 2012 until now). See **mapcrime.py** in the repositories

The shops CSV file, extracted from the Open Colorado Data Sets, comprised a listing of active shops/dispensaries as of November 2017. See **mapshops.py** in repositories Google maps was utilized via pygmaps. I show the first 10 places. You can change the variable count (I use count=10) and get more places on the map.

Some Anticipated Problems:
1) A question that arises is, "How can we be sure if the crimes are drug-related charges?" If they are DUI, it would require further inquiry to distinguish between cannabis or other intoxicants (ie, alcohol).

2) The Crime and Shops datasets are downloaded from a website. This means they are merely snapshots at the time of download.

Some Possible Solutions:
1) A possible solution would be to filter out the crimes to those specifically cannabis-related. This would eliminate crimes otherwise unrelated to cannabis. This would allow us to map out the crimes in a less subjective manner.
2) The Crime and Shops would need a clean install and replacement for each analysis. This would ensure the datasets are recent and up-to-date.

For a map comparing both crime and shops, See **MapBothCrimeAndShops.py** in repositories.

I show the first 10 places. You can change the variable count (I use count=10) and get more places on the map.

For a map comparing both crime and shops, see **MapBothCrimeAndShops.py** in repositories.