OpenStreetMap Data for Austin, TX

Map Area

Austin, TX USA

https://mapzen.com/data/metro-extracts/metro/austin_texas/

This area is more familiar to me at the moment so I chose it. It also meets the project requirements of having at least 50 MB file size uncompressed. As delineated in class, the data was obtained by downloading the readily available extract above. Uncompressing the file gave a 1.4 GB osm file. A sample of this file was generated using the code provided in the instructions for the project. The link to this smaller osm file is:

https://www.dropbox.com/s/084lnztuwgxdgtm/sample.osm?dl=0

Problems Encountered in the Map

Exploration of the sample osm file as well as the whole osm file showed that:

1. Street names need to be unabbreviated.
2. Inconsistent abbreviation for street names poses challenge in cleaning data (ex: IH35, I H 35, I-35, I35)
3. Phone number format is not consistent (ex: (512) 782-5659, +1 512-472-1666, 51224990093, 512 466-3937, etc.)
4. More than one phone number are entered in the field (ex: "Main: (512) 899-4300 Catering: (512) 899-4343")
5. Postcodes didn't have a consistent format--some have county codes, some do not.
6. City name format is not consistent (ex: Pflugerville, TX; Pflugerville)

Notes About the Files

I ran my final code using Jupyter Notebook, using data_extraction.ipynb. Exploration of data is detailed in exploration_audit.ipynb. But the py files needed to create the csv files are cities.py, mapping_street.py, schema.py, clean.py, and data.py.

Cleaning of Street Names

Using the method described in the case study exercises for the course, street names were audited using a regex and if street names don't follow that regex they get added to a dictionary of type set which makes sure that if the street name is already present, it won't get added to the dictionary. In the auditing function, the iterparse method is used to iterate through the xml tree.

```
def is_street_name(elem):
    return (elem.attrib['k'] == 'addr:street')

def audit(osmfile):
    osm_file = open(osmfile, 'r')
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):
        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
    osm_file.close()
    return street_types
```

The result from the audit showed that aside from the fact that some street names are heavily abbreviated, some street names are abbreviated inconsistently. This was addressed in the final code for cleaning up the street names, although there were still some problems remaining after clean up, such as certain streets have different names. For example, Ranch Road 620 is also referred to as Farm-to-Market Road 620, US Highway 290 is also Country Road 290. These were not addressed in the project although it could be easily added to the "mapping_street" dictionary (discussed below).

In the process of auditing and coming up with functions to clean street names, I found it easier to create separate functions to fix different problems. The final function uses subfunctions that will be described below.

In updating Farm-to-Market and Road-to-Market Roads, the challenge was to make it possible to update the following:

- FM / RM
- FM Road / RM Road
- Farm to Market / Ranch to Market
- Farm to Market Road / Ranch to Market Road
- FM620 / RM1431 (with number of the road attached)

elif statements were used in the function "update_farm_ranch_to_market" to fix the problem:

```
def update_farm_ranch_to_market(name):
    parts = name.split()
    if "Farm-to-Market" in parts or "Ranch-to-Market" in parts:
        if "Road" in parts:
            return name
        else:
            try:
                parts.insert(parts.index("Farm-to-Market")+1, "Road")
                name = " ".join(parts)
            except ValueError:
                parts.insert(parts.index("Ranch-to-Market")+1, "Road")
                name = " ".join(parts)
    elif "Farm" in parts and "to" in parts and "Market" in parts:
        newname = []
        for i in range(parts.index("Farm")):
            newname.append(parts[parts.index(i)])
        newname.append("Farm-to-Market")
        newname += parts[parts.index("Market")+1:]

        if "Road" in parts:
            name = " ".join(newname)
        else:
            newname.insert(newname.index("Farm-to-Market")+1,"Road")
            name = " ".join(newname)

    elif "Ranch" in parts and "to" in parts and "Market" in parts:
        newname = []
        for i in range(parts.index("Ranch")):
            newname.append(parts[parts.index(i)])
```

```
            newname.append("Ranch-to-Market")
            newname += parts[parts.index("Market")+1:]

            if "Road" in parts:
                name = " ".join(newname)
            else:
                newname.insert(newname.index("Ranch-to-Market")+1,"Road")
                name = " ".join(newname)

        return name
```

In some cases, "Highway" is needed to be appended. For example:

- US 290 (United States Highway 290)
- I35 (Interstate Highway 35)

The following function, "append_highway" was created:

```
def append_highway(name):
    newparts = []
    parts = name.split()
    for item in parts:
        if (item == "Interstate" or item == "States") and "Highway" not in parts:
            newparts.append(item)
            newparts.append("Highway")
        else:
            newparts.append(item)
    name = ' '.join(newparts)
    return name
```

In fixing common problems such as spellling out the full street names in abbreviated names, a problem occurred where there is more than one meaning to the abbreviation. For example:

- "St" in "Pecan St" (Pecan Street) and Rue de St Germanaine (Rue de Saint Germaine)
- "H" and "I" in "I H 35" (Interstate Highway 35) and "Avenue H" (as is) and "Avenue I" (as is)
- "C" in "C R" (Country Road) and "Avenue C" (as is)
- "N" in "N ..." (North) and "Avenue N" (as is)

This problem was addressed by adding some lines to the "update_name" function, which was used in the case study exercises. In the case study, the "update_name" function uses a dictionary ("mapping_street") of abbreviated to unabbreviated pairs of street names to update the street names. elif statements were used. In the "update_name" function below, lines of code were added to address the problems above. "St" for "Saint" was updated first before "St" for "Street". "N", "C", "I" and "H" were also attended to first before mapping them to the "mapping" dictionary. The other functions "update_farm_ranch_to_market" and "append_highway" were added towards the last part of the function, after the "unabbreviations".

```
def update_name(name, mapping_street):
    parts = name.split()
    newparts = []
    for item in parts:
        if item == "St" and "Rue" in parts:
            newparts.append("Saint")
        elif item == "N" or item == "C" or item == "I" or item == "H":
            try:
                if newparts[0] == "Avenue":
                    newparts.append(item)
                else:
                    newparts.append(mapping[item])
            except IndexError:
                newparts.append(mapping[item])
        else:
            if item in mapping.keys():
                newparts.append(mapping[item])
```

```
            else:
                newparts.append(item)
    name = ' '.join(newparts)
    name = append_highway(name)
    name = update_farm_ranch_to_market(name)
    return name
```

A note about my use of try/except statements: they were used without regard for whether if/else statements can be used, as I thought it was fine to use, until a forum mentor informed me to only use try/except when if/else can't be used anymore. I did try to use if/else statements more if I can, it's just that sometimes, I deemed it better to use try/except statements (fewer lines of code).

Cleaning Postcodes

Another area to update is the postal codes which do not follow a uniform format. Most of the postcodes do not include the county codes. I decided to remove the county codes. But if a total cleaning is needed, a new field for county codes should be created in order to not lose the county codes data. This wasn't done here (however, it might be easily fixable if MongoDB was used--I used SQL).

To audit the postcodes, a regex of the form r'^7\d\d\d\d$' (must have five digits which must start with "7" and must end with a digit, hence the caret at the beginning and a dollar sign at the end) was used to exclude any entries following the 5-digit postcode format. Anything not following this format can be printed off and examined to see how they can be updated.

```
import re

def is_postcode(element):
    return (element.attrib['k'] == "addr:postcode" or element.attrib['k'] == "postal_code")

postcode_re = re.compile('r^7\d\d\d\d$')

counter = 0
for element in get_element(SAMPLE_FILE):
    if counter == 25:
        break
    if element.tag == "node" or element.tag == "way":
        for tag in element.iter("tag"):
            if is_postcode(tag):
                if postcode_re.search(tag.attrib['v']) == None:
                    print tag.attrib['v']
                    counter += 1
```

In the code above, the "get_element" function from the code used to create the sample file (and also used in the "shape_element" function to be discussed later) came in handy in iterating through the xml tree during auditing. Also, instead of parsing through the whole xml, whether it is a sample or the whole file, using the variable "counter" allowed for the ability to stop iterating at the desired number of elements. The use of this approach was inspired by the lines of code used in the very first problem on using the csv module where instead of parsing the whole csv file, we only parse for a certain number of lines. Results of the above code gave these outliers (among others):

```
78704-5639
14150
TX 78613
Texas
```

To update the postcodes, the "update_postcode" function was created. It uses a different but similar regex:

```
def update_postcode(postcode):
    try:
        postcode = re.compile(r'7\d\d\d\d').search(postcode).group()
    except AttributeError:
```

```
        postcode = 'None'
    return postcode
```

The use of the regex above in the function allowed for the 5-digit postcode to be extracted from the jumble of 5-digit numbers if there is a one, instead of doing any manipulations (remove county codes, remove "TX" to obtain the correct format).

Cleaning Phone Numbers

A similar approach to auditing post codes was used to audit the phone numbers. "phone_re_audit" was used for auditing osm files.

```
phone_re_audit = re.compile(r'^\d\d\d\-\d\d\d\-\d\d\d\d$')
```

I simply chose the xxx-xxx-xxxx format for the phone numbers. Some of the results of the audit using a similar code as in the auditing of postcodes, are:

```
+1 512-472-1666
(512) 494-9300
51224990093
+1-512-666-5286;+1-855-444-8301
Main: (512) 206-1000 Catering: (512) 206-1024
```

After auditing, I decided to use the same regex but cleaning the phone numbers is a little involved because of the dashes that need to be appended. Also, the occurrence of two phone numbers complicated the case. This was addressed using a limit of number of characters in the phone number (10).

```
def update_phone(number):
    phone_re = re.compile(r'^\d\d\d\-\d\d\d\-\d\d\d\d$')
    if phone_re.search(number) == None:
        phno = []
        number = list(number.lstrip("+1"))
        for char in number:
            try:
                if int(char) in [x for x in range(10)]:
                    if len(phno) == 10:
                        continue
                    phno.append(char)
            except ValueError:
                continue
        number = "".join(phno)
        number = number[:3] + "-" + number[3:6] + "-" + number[6:]

    return number
```

Another approach to the two phone numbers entered in the field would be using the .findall() method instead of .search() method in the regex statement, resulting in a list of phone numbers but I didn't do this as I realized it late.

Cleaning the City Names

Auditing and cleaning the city names follow a similar method as auditing and cleaning the street names. A list of expected cities was created and any city not found in this list was added to a set that should contain city names that need to be updated:

```
def is_city(element):
    return element.attrib['k'] == "addr:city"

cities = set()

for element in get_element(SAMPLE_FILE):
```

```
    if element.tag == "node" or element.tag == "way":
        for tag in element.iter("tag"):
            if is_city(tag):
                if tag.attrib['v'] not in expectedcities:
                    cities.add(tag.attrib['v'])
```

I found that running this code on the whole osm file returned only a few lines. A dictionary similar to that used in cleaning the street names was then created. The function "update_city" involved only a few lines:

```
def update_city(city, expectedcities, mapping_city):
    if city not in expectedcities:
        if city in mapping_city.keys():
            city = mapping_city[city]
        else:
            city = "None"
    return city
```

There were still a lot of items to update, and some of these are evident after creation of the database. But only the items discussed above were done.

Instead of using all the functions above individually in the final xml data extraction, particularly in the "shape_element" function, a "clean" function was created and eventually used.

```
def clean(value, tag, mapping_street, expectedcities, mapping_city):
    if is_street_name(tag):
        value = update_name(value, mapping_street)
    elif is_phone(tag):
        value = update_phone(value)
    elif is_postcode(tag):
        value = update_postcode(value)
    elif is_city(tag):
        value = update_city(value, expectedcities, mapping_city)
    return value
```

Extraction of Data from OSM File to CSV Files

Data were extracted from the OSM file using the functions from the case study exercises from the course. However, we were guided to write the "shape_element" function which not only parses the osm xml data but also cleans the data using the functions discussed above. The general scheme for processing osm files start from creating csv files as output files using the codecs module, then shaping the output, validating this output against a set schema and then writing the output onto the csv files.

The shape_element function

Parsing and cleaning of xml data occurs in the shape_element function. It extracts values of attributes from an xml element, instead of a whole xml tree (http://effbot.org/zone/celementtree.htm, http://effbot.org/zone/element-iterparse.htm, https://classroom.udacity.com/nanodegrees/nd002/parts/0021345404/modules/316820862075461/lessons/5436095827/concepts/54475500150923#). Along with this extraction, the data is updated accordingly and appended to receptacles (lists and dictionaries).

I encountered problems processing the whole osm file with validation set to True even if I didn't obtain any errors processing the sample file with validation set to True. To figure out what was wrong, I gathered from the course forum that I needed to look for missing fields in the csv file obtained from running the program with validation set to False. However, I found it impossible to find any in the large csv output files. Besides, it was also impossible to load the whole file using a spreadsheet program. Blindly, I resorted to including lines of code to address missing data, though doing this was futile. An example of this is:

```
 if element.attrib[field] == '':
     node_attribs[field] = '999999'
 else:
     node_attribs[field] = element.attrib[field]
```

Eventually, I thought about looking at the output csv files from the processing of the whole file with validation on which failed. I took the last line in the resulting (incomplete) csv file, which happen to be nodes_tags.csv, using the csv module and the reader method.

```
import csv
with open("nodes_tags.csv", "r") as f:
    lastrow = None
    for lastrow in csv.reader(f):
        pass
    print lastrow
```

(Reference: http://stackoverflow.com/questions/20296955/reading-last-row-from-csv-file-python-error)

The result from this code gave me the 'id' of the element I can use to search where the line that causes the error is in the whole osm file:

```
import xml.etree.cElementTree as ET

def get_element(osm_file, tags=('node', 'way', 'relation')):
    """Yield element if it is the right type of tag
    Reference:
    http://stackoverflow.com/questions/3095434/inserting-newlines-in-xml-file-generated-via-xml-etree-elementtree-in-
    """
    context = ET.iterparse(osm_file, events=('start', 'end'))
    _, root = next(context)
    for event, elem in context:
        if event == 'end' and elem.tag in tags:
            yield elem
            root.clear()

counter = 0
for element in get_element(OSM_FILE):
    counter += 1
    if element.tag == 'node':
        if element.attrib['id'] == '4133425201':
            print counter
            break

 Out: 6338418
```

This gave me the element number in the osm xml which gives rise to the error. I then generated a smaller osm file using this information, using the same lines of code used to make the sample osm file provided in the course.

```
SAMPLE_FILE = "expectederror_file.osm"

with open(SAMPLE_FILE, 'wb') as output:
    output.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    output.write('<osm>\n')

    for i, element in enumerate(get_element(OSM_FILE)):
        if i > 6338416:
            output.write(ET.tostring(element, encoding='utf-8'))
    output.write('</osm>')
```

I then processed "expectederror_file.osm" using my code with validation off and found the missing field or empty cell. I then found that the error was because of the value "service area" for attribute 'k', which should have been ignored

since it contains a problem character. This then told me that the problem in the code is the cleaning of the 'k' values. The reason was the following lines of code in the shape_element function:

```
try:
    problem_chars.search(tag.attrib['k']).group()
except AttributeError:
   try:
      lower_colon.search(tag.attrib['k']).group()
      kvalue = tag.attrib['k'].split(":")
      nodetags['type'] = kvalue[0]
      if len(kvalue) == 2:
          nodetags['key'] = kvalue[1]
      else:
          nodetags['key'] = ":".join(kvalue[1:])
   except AttributeError:
      nodetags['type'] = default_tag_type
      nodetags['key'] = tag.attrib['k']
```

This would have worked if I included "continue" in the third line of the code.

```
try:
    problem_chars.search(tag.attrib['k']).group()
    continue
except AttributeError:
    ....
```

However, I was instructed by the forum mentor to use if/else statements and use "continue". I did change my code to use the if/else statement:

```
if problem_chars.search(tag.attrib['k']) != None:
    continue
else:
    if lower_colon.search(tag.attrib['k']) != None:
         kvalue = tag.attrib['k'].split(":")
         nodetags['type'] = kvalue[0]
         if len(kvalue) == 2:
             nodetags['key'] = kvalue[1]
         else:
             nodetags['key'] = ':'.join(kvalue[1:])
    else:
         nodetags['type'] = default_tag_type
         nodetags['key'] = tag.attrib['k']
```

After this, I was able to obtain validated csv files and went on to create the SQL database.

Creation and Querrying of SQL Database

Creating the SQL database (atx_osm.db) was done using Python according to the method outlined in the course forum (https://discussions.udacity.com/t/creating-db-file-from-csv-files-with-non-ascii-unicode-characters/174958/6), using the schema specified in the following site: https://gist.github.com/swwelch/f1144229848b407e0a5d13fcb7fbbd6f. The process was straightfoward. All codes are contained in this notebook: https://github.com/mudspringhiker/wrangle_open_streetmap_data/blob/master/db_creation.ipynb

Querrying for list of cities showed that pretty much of all the cities were cleaned:

```
cities = cur.execute("""SELECT tags.value, COUNT(*) as count
                        FROM (SELECT * FROM nodes_tags
                          UNION ALL
                          SELECT * FROM ways_tags) tags
                        WHERE tags.key = 'city'
```

```
                      GROUP BY tags.value
                      ORDER By count DESC""").fetchall()
print cities

Out: [(u'Austin', 3068),
    (u'Round Rock', 113),
    (u'Kyle', 64),
    (u'Cedar Park', 43),
    (u'Pflugerville', 37),
    (u'Leander', 33),
    (u'Buda', 26),
    (u'Georgetown', 17),
    (u'Dripping Springs', 13),
    (u'West Lake Hills', 12),
    (u'Bastrop', 9),
    (u'Elgin', 9),
    (u'Lakeway', 9),
    (u'Wimberley', 8),
    (u'Taylor', 7),
    (u'Bee Cave', 6),
    (u'Del Valle', 5),
    (u'Manor', 5),
    (u'Manchaca', 4),
    (u'Cedar Creek', 3),
    (u'Hutto', 3),
    (u'Spicewood', 3),
    (u'Creedmoor', 2),
    (u'Lago Vista', 2),
    (u'San Marcos', 2),
    (u'Sunset Valley', 2),
    (u'Webberville', 2),
    (u'Driftwood', 1),
    (u'Jonestown', 1),
    (u'Lost Pines', 1),
    (u'Manchacha', 1),
    (u'Maxwell', 1),
    (u'Smithville', 1)]
```

Looking at the postcodes, there were three "None" values.

```
postcode = cur.execute("""SELECT tags.value, COUNT(*) as count
                      FROM (SELECT * FROM nodes_tags
                            UNION ALL SELECT * FROM ways_tags) tags
                      WHERE tags.key = 'postcode'
                      GROUP BY tags.value
                      ORDER By count DESC""").fetchall()
```

Result from the above query include:

```
(u'None', 3)
```

To figure out what these should be, I querried for the accompanying information with these values.

```
missing_postcodes = cur.execute("""SELECT *
                                   FROM (SELECT * FROM nodes_tags
                                     UNION ALL
                                     SELECT * FROM ways_tags) tags
                                   WHERE tags.key = 'postcode'
                                   AND tags.value = 'None'""").fetchall()

print missing_postcodes

Out:    [(2152207067, u'postcode', u'None', u'addr'),
        (247506590, u'postcode', u'None', u'addr'),
        (383791236, u'postcode', u'None', u'addr')]
```

To determine what info is accompanying id 2152207067, the following query was done:

```
cur.execute("""SELECT *
         FROM (SELECT * FROM nodes_tags
         UNION ALL
         SELECT * FROM ways_tags) tags
         WHERE tags.id = 2152207067""")
missing_postcode1_info = cur.fetchall()
print missing_postcode1_info

Out: [(2152207067, u'name', u'Nyle Maxwell - Taylor', u'regular'),
    (2152207067, u'shop', u'car', u'regular'),
    (2152207067, u'website', u'www.nylemaxwellcjd.com', u'regular'),
    (2152207067, u'street', u'United States Highway 79', u'addr'),
    (2152207067, u'postcode', u'None', u'addr')]
```

From this result and accessing the provided website, it can be found that the postcode should be 76574.

The other postcodes were determined in the same way.

Number of Nodes and Ways

Nodes:

```
In [17]: cur.execute("SELECT COUNT(*) FROM nodes")
         nodes = cur.fetchall()
         nodes
Out[17]:
         [(6356394,)]
```

This value is the same as the one obtained from the exploration of dataset using xml.etree.cElementTree module of Python (see p3_wrangle_openstreetmap_1.ipynb, High Level Tags).

```
In [18]: cur.execute("SELECT COUNT(*) FROM ways")
         ways = cur.fetchall()
         ways
Out[18]:
        [(666390,)]
```

This is also the same number obtained from the the ElementTree module in Python (p3_wrangle_openstreetmap_1.ipynb, High Level Tags).

Number of Users/Contributors

```
In [19]: cur.execute("""SELECT COUNT(DISTINCT(e.uid))
                        FROM (SELECT uid from nodes UNION ALL SELECT uid FROM ways) e""")
         users = cur.fetchall()
         users
Out[19]:
         [(1146,)]
```

This number is lower than the one obtained using the ElementTree module in the exploration of the xml osm (1155 users, p3_wrangle_openstreetmap_1.ipynb, Exploring Users). This might be because when the csv files were created, the key values with problematic characters were removed, along with the rest of the record containing that value.

Top 10 Contributing Users

```
cur.execute("""SELECT e.user, COUNT(*) as num
         FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways) e
         GROUP BY e.user
```

```
            ORDER BY num DESC
            LIMIT 10""").fetchall()
```

Output:

```
[(u'patisilva_atxbuildings', 2743705),
 (u'ccjjmartin_atxbuildings', 1300514),
 (u'ccjjmartin__atxbuildings', 940070),
 (u'wilsaj_atxbuildings', 359124),
 (u'jseppi_atxbuildings', 300983),
 (u'woodpeck_fixbot', 223425),
 (u'kkt_atxbuildings', 157847),
 (u'lyzidiamond_atxbuildings', 156383),
 (u'richlv', 50212),
 (u'johnclary_axtbuildings', 48232)]
```

I explored the use of pandas to look at the list of users:

```
contributions = cur.execute("""SELECT e.user, COUNT(*) as num
                           FROM (SELECT user FROM nodes UNION ALL SELECT user FROM ways) e
                           GROUP BY e.user
                           ORDER BY num DESC""").fetchall()
import pandas

contributions_df = pd.DataFrame(contributions)
contributions_df['users'] = contributions_df[0]
contributions_df['count'] = contributions_df[1]
del contributions_df[0]
del contributions_df[1]
contributions_df.head(10)
```

This created a better looking table than the results of the sql query:

Out[29]:

	users	count
0	patisilva_atxbuildings	2743705
1	ccjjmartin_atxbuildings	1300514
2	ccjjmartin__atxbuildings	940070
3	wilsaj_atxbuildings	359124
4	jseppi_atxbuildings	300983
5	woodpeck_fixbot	223425
6	kkt_atxbuildings	157847
7	lyzidiamond_atxbuildings	156383
8	richlv	50212
9	johnclary_axtbuildings	48232

Locations of Restaurants

The query used to obtain a list of all the restaurants in the Austin, TX area was:

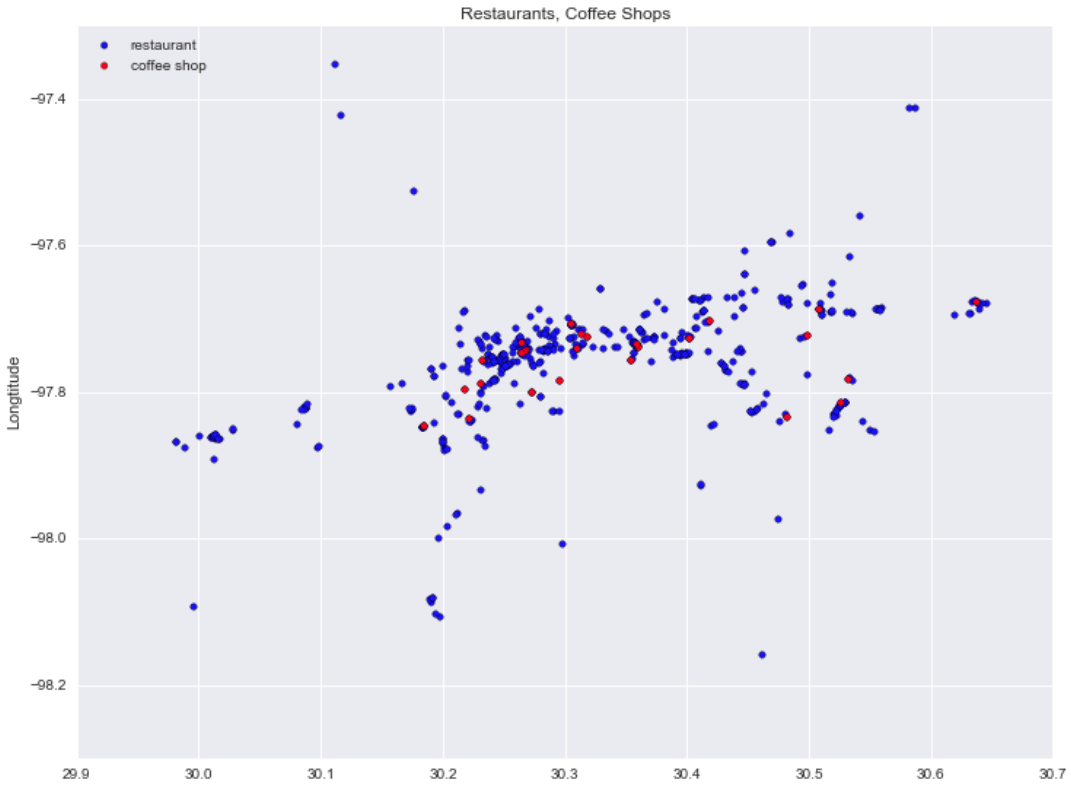
```
cuisine_loc = cur.execute("""SELECT b.id, b.value, nodes.lat, nodes.lon
                             FROM (SELECT * FROM nodes_tags UNION ALL SELECT * FROM ways_tags) b
                               JOIN nodes ON b.id = nodes.id
                             WHERE b.key = 'cuisine'""").fetchall()
```

Obtaining the locations of the coffee shops will then have a similar code:

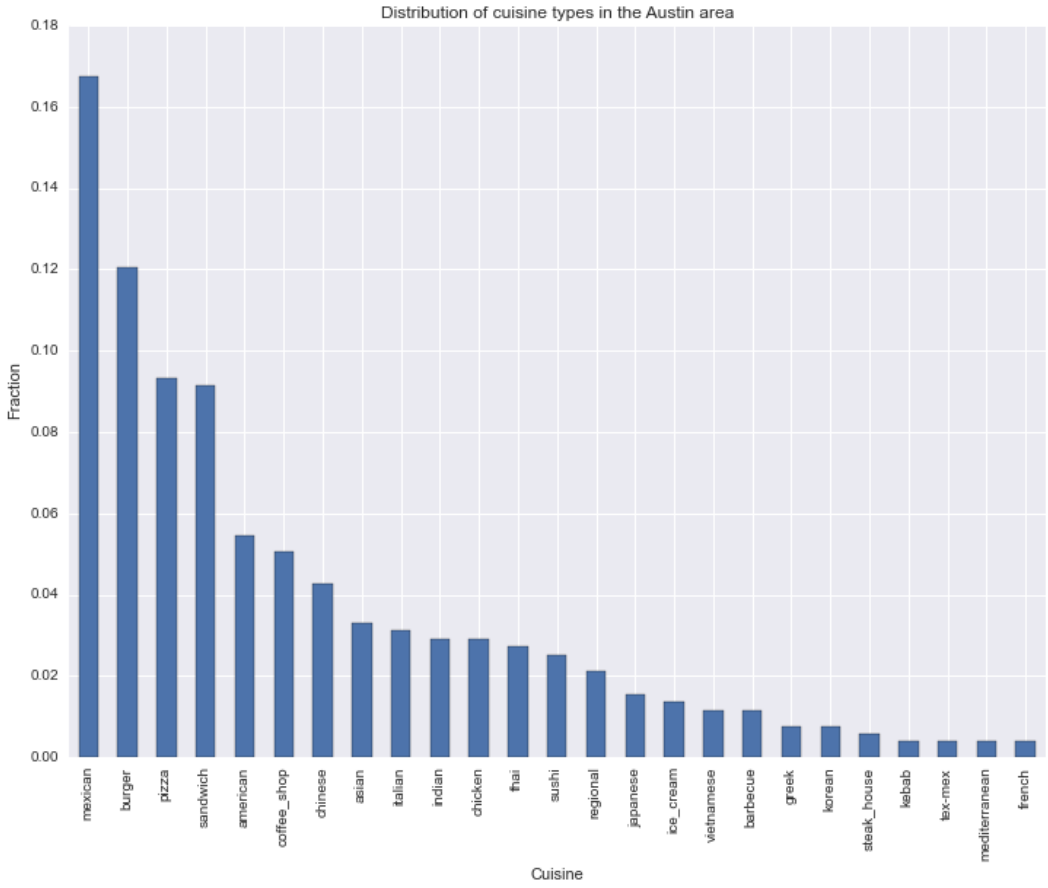
```
coffee_loc = cur.execute("""SELECT b.id, b.value, nodes.lat, nodes.lon
                            FROM (SELECT * FROM nodes_tags UNION ALL SELECT * FROM ways_tags) b
                              JOIN nodes ON b.id = nodes.id
                            WHERE b.value = 'coffee_shop'""").fetchall()
```

Plotting the locations of these restaurants vs. the locations of coffee shops can then be done:

```
import matplotlib.pyplot as plt
import seaborn
% matplotlib inline (only if using notebook)
plt.scatter([x[2] for x in cuisine_loc], [y[3] for y in cuisine_loc], c='blue', label="restaurant")
plt.scatter([x[2] for x in coffee_loc], [y[3] for y in coffee_loc], c='red', label="coffee shop")
plt.xlabel('Latitude')
plt.ylabel('Longtitude')
plt.title('Restaurants, Coffee Shops')
plt.legend(loc=2)
```



Lastly, querrying the database for the most popular cuisines was done. Pandas was used to eventually plot the distribution of the different types of restaurants in the Austin, TX area. It is no surprise that the area has a lot of Mexican restaurants.



File Sizes

```
austin_texas.osm        1.41 GB
atx_osm.db              820.4 MB
nodes.csv               604.3 MB
nodes_tags.csv          11.7 MB
ways.csv                48.6 MB
ways_tags.csv           70.6 MB
ways_nodes.csv          175.6 MB
```

Conclusion

Information from an xml file can be scraped for data by Python through the xml.eTree.ElementTree module. This can be converted to a csv file which can be converted to an sql database (or to a pandas dataframe, which is not shown here, but was explored in another unit of the course). SQL databases can be converted to a pandas dataframe.

Cleaning of data takes a while. Knowledge of the nature of data also is very important so the best decisions on what to do with it can be done.

Other References

Automate the Boring Stuff with Python: Practical Programming for Total Beginners, A. Sweighart, No Starch Press San Francisco, CA, USA ©2015 ISBN:1593275994 9781593275990

http://stackoverflow.com/questions/19877344/near-syntax-error-when-trying-to-create-a-table-with-a-foreign-key-in-sqlit

Brandon Rhodes - Pandas From The Ground Up - PyCon 2015, https://www.youtube.com/watch?v=5JnMutdy6Fw

Udacity Data Wrangling Course
(https://classroom.udacity.com/nanodegrees/nd002/parts/0021345404/modules/316820862075460/lessons/491558559/concepts/816599080