John K. Hancock
Udacity: Data Wrangling with MongoDB
P3: Wrangle Open Street Map Project
April 12, 2016
FINAL Project –Revison 1

OBJECTIVE:

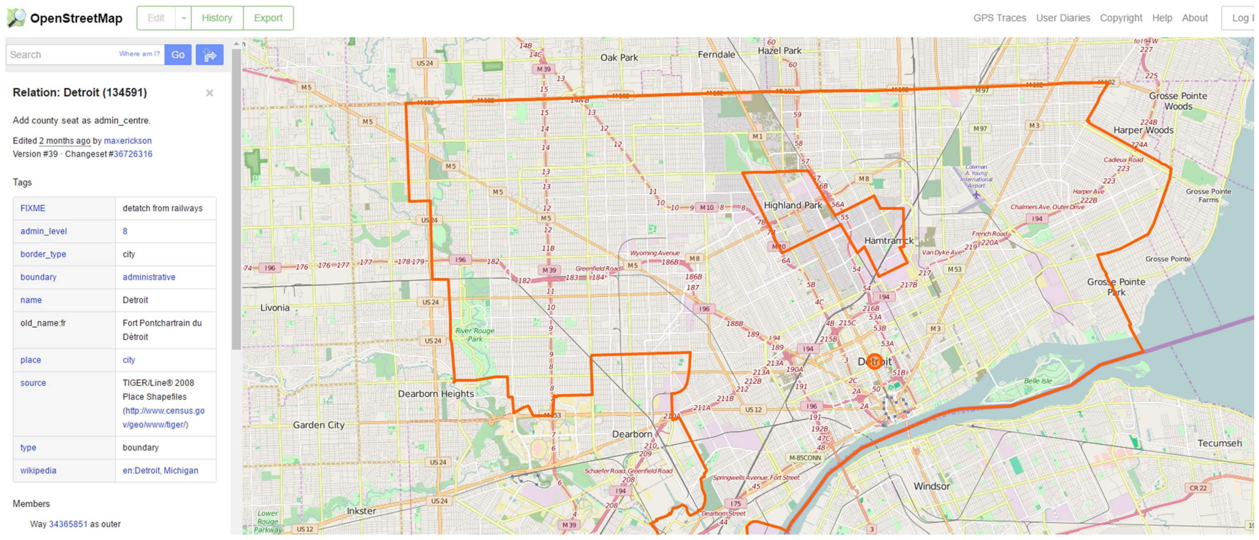
The objective of the project is to extract, audit, clean, shape, and load data from the Open Street Map project(osm) for the city of Detroit, MI into a MongoDB database instance. The project will achieve that by exporting out the osm file for Detroit from the website (https://www.openstreetmap.org). Then, audit the data, clean it, shape the data into a list of JSON objects and load it into an instance of MongoDB. Finally, perform searches and other data analysis in MongoDB.

METHODOLOGY:

Part One –The Audit

(Note: *The python file corresponding to the Audit section of this report can be found in the file, Part_One_Audit.py*)

1. Export the osm file
   A search was entered into the search box for the city of Detroit, MI



Relation: Detroit (134591)

Add county seat as admin_centre.

Edited 2 months ago by maxerickson
Version #39 · Changeset #36726316

Tags

FIXME	detatch from railways
admin_level	8
border_type	city
boundary	administrative
name	Detroit
old_name:fr	Fort Pontchartrain du Détroit
place	city
source	TIGER/Line® 2008 Place Shapefiles (http://www.census.gov/geo/www/tiger/)
type	boundary
wikipedia	en:Detroit, Michigan

Members

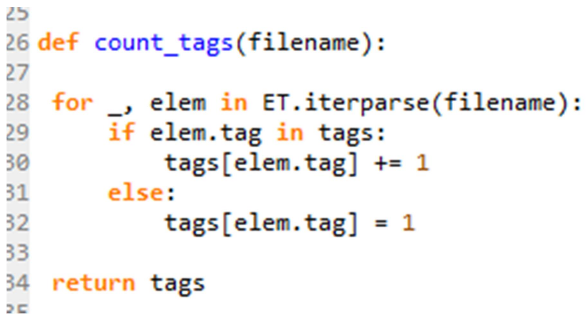
Way 34365851 as outer

The result were two copies of a 735.65 MB file titled: detroit_michigan.osm and detroit_michigan.xml. For this project, I used the file titled: detroit_michigan.osm.

2. Measuring Data Validity: Does the Data Conform to a Schema

   The first step was to see what tags are in the dataset and confirm its structure.

   The function, count_tags(filename), counts the number of tags in the file. At Line 19 of the audit file[1], a python dictionary is created. The function, count_tags(filename), takes in the osm file and using iterparse (see line 28), the function iterates through the entire osm file. It counts the name of each tag and tracks the number of time that tag appears in the file in the loop below:



```
25
26 def count_tags(filename):
27
28   for _, elem in ET.iterparse(filename):
29       if elem.tag in tags:
30           tags[elem.tag] += 1
31       else:
32           tags[elem.tag] = 1
33
34   return tags
35
```

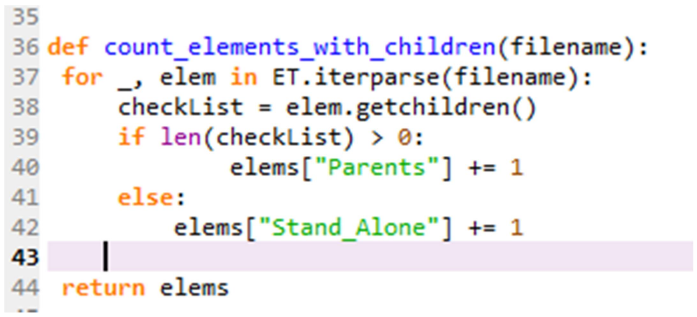
   The results are below:

```
{'bounds': 1,
 'member': 36910,
 'nd': 3981063,
 'node': 3485494,
 'osm': 1,
 'relation': 5027,
 'tag': 1920712,
 'way': 332193}
```

   We can verify that the downloaded dataset conforms to the schema as documented on the open street map website. The next validity check is to see how many elements in the file have children and how many are standalone elements. To answer this, the function, count_elements_with_children(filename)[2], performs this check.

   Again, using iterparse, the function first checks if the element tag has children by calling the function, getchildren(). This function returns a list of all of the children in the tag to the list, checklist. If the length of this list is greater than 0, then the element has children. The dictionary, elems, tracks the number of element parents and the standalones.

[1] See the file, *Part_One_Audit.py*
[2] See line line 36, in *Part_One_Audit.py*



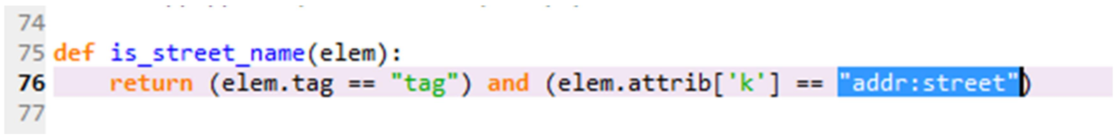
```
def count_elements_with_children(filename):
 for _, elem in ET.iterparse(filename):
     checkList = elem.getchildren()
     if len(checkList) > 0:
             elems["Parents"] += 1
     else:
         elems["Stand_Alone"] += 1
     
 return elems
```

The results show that there are 440,571 elements with child elements, and 9,320,830 elements that are stand alone.

{'Parents': 440571, 'Stand_Alone': 9320830}

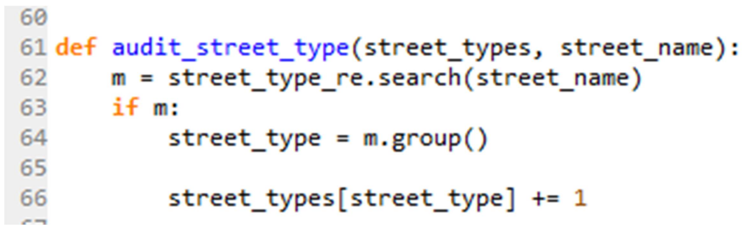
3. <u>Measuring Data Accuracy: Perform an Audit of the Data</u>

Now that the dataset conforms to a valid schema, the next step is to examine the quality of the data and identify what needs to be cleaned. Using code from lesson 3 of the Data Wrangling course, the audit() function iterates through the osm file calling is_street_name(elem). This function checks to see if the attribute, "addr:street", is in the element.



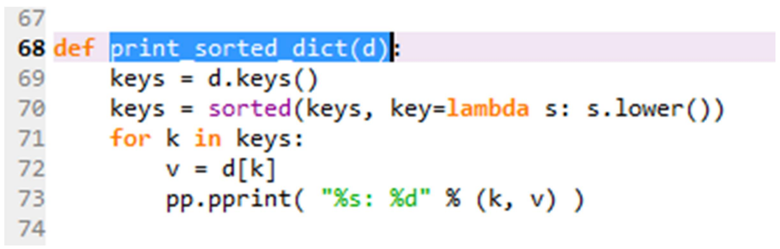
```
def is_street_name(elem):
    return (elem.tag == "tag") and (elem.attrib['k'] == "addr:street")
```

If yes, it calls audit_street_type(street_types, street_name), which uses regular expressions to parse out the street types from the file and loads them in the street_types dictionary.



```
def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()

        street_types[street_type] += 1
```

Finally, the results of this audit are printed out from the dictionary using the function, print_sorted_dict(d) which sorts the data.



```
def print_sorted_dict(d):
    keys = d.keys()
    keys = sorted(keys, key=lambda s: s.lower())
    for k in keys:
        v = d[k]
        pp.pprint( "%s: %d" % (k, v) )
```

The results showed tje abbreviations for the types of streets which will have to be cleaned before loading:

'Blvd: 7'
'Blvd.: 19'
'Boulevard: 1371'

'Rd: 102'
'Rd.: 6'
'River: 8'
'Road: 10631'
'road: 2'

(See Appendix One for a complete log of these results)

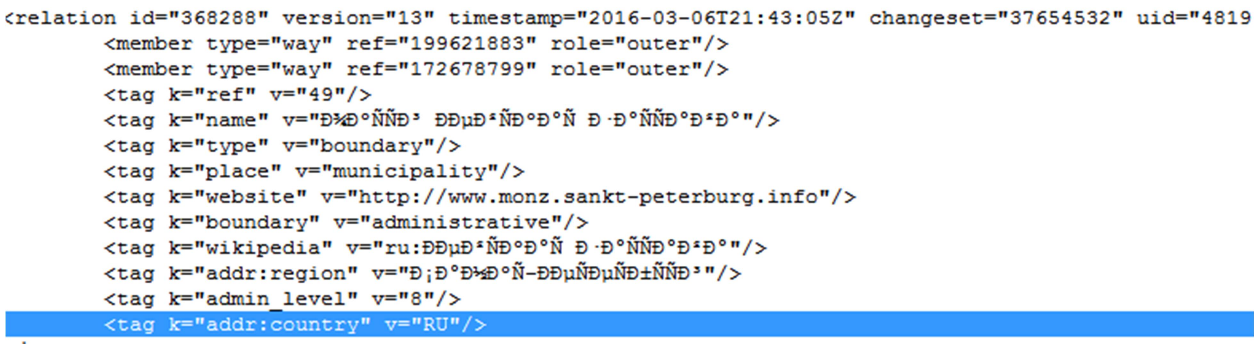
One issue that I encountered was that there are street names that were obviously not in Detroit or for that matter were not even in the U.S.

Excerpts from the audit of street names show the following peculiar entries:

'Adolf-Tibus-Straße: 1'
'Charlottenstraße: 1'
'Kwadeveldenstraat: 4'
'Paardskerkhofstraat: 1'
'Zakładowa: 2'
'вулиця: 1'
'Ивановского: 1'
'Кирова: 1'
'Корзуна: 1'
'Пирогова: 3'
'Ситничка: 1'
'улица: 2'

(See Appendix Two for a complete log of these results)

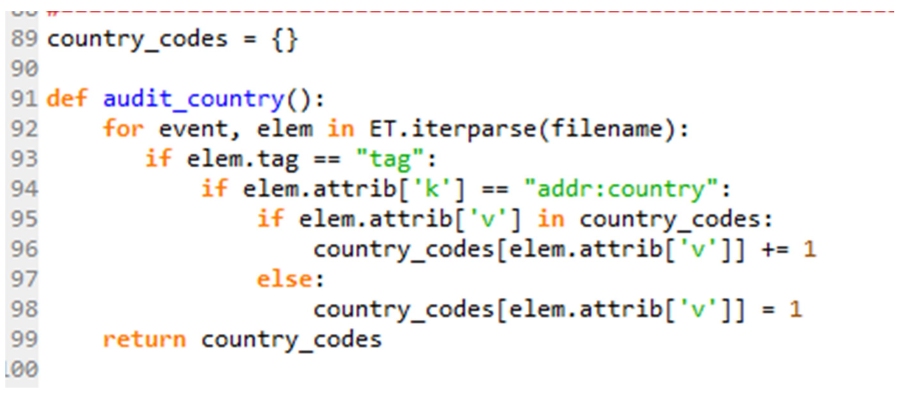
After looking at elements in the osm file that contain these street names, I found that there is data from this dataset that is outside of the U.S.



```
<relation id="368288" version="13" timestamp="2016-03-06T21:43:05Z" changeset="37654532" uid="4819
        <member type="way" ref="199621883" role="outer"/>
        <member type="way" ref="172678799" role="outer"/>
        <tag k="ref" v="49"/>
        <tag k="name" v="Ð¾Ð°ÑÑÐ³ ÐÐµÐ²ÑÐ°Ð°Ñ Ð·Ð°ÑÑÐ°Ð²Ð°"/>
        <tag k="type" v="boundary"/>
        <tag k="place" v="municipality"/>
        <tag k="website" v="http://www.monz.sankt-peterburg.info"/>
        <tag k="boundary" v="administrative"/>
        <tag k="wikipedia" v="ru:ÐÐµÐ²ÑÐ°Ð°Ñ Ð·Ð°ÑÑÐ°Ð²Ð°"/>
        <tag k="addr:region" v="Ð¡Ð°Ð½Ð°Ñ-ÐÐµÑÐµÑÐ±ÑÑÐ³"/>
        <tag k="admin_level" v="8"/>
        <tag k="addr:country" v="RU"/>
```

Measuring Data Accuracy: Perform an Audit of the Data – Non US Entries

The function, audit_country(), iterates through the file and looks at the tag attribute, “addr:country”. It then extracts the value of that tag and loads them into the dictionary, country_codes = {}:



```python
#-----------------------------------------------------------------------
country_codes = {}

def audit_country():
    for event, elem in ET.iterparse(filename):
        if elem.tag == "tag":
            if elem.attrib['k'] == "addr:country":
                if elem.attrib['v'] in country_codes:
                    country_codes[elem.attrib['v']] += 1
                else:
                    country_codes[elem.attrib['v']] = 1
    return country_codes
```

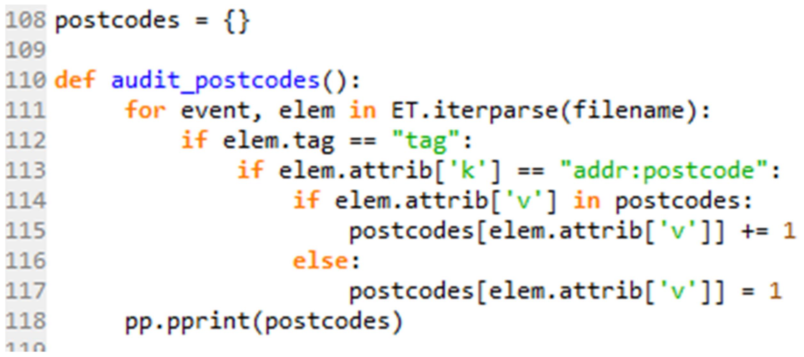
The results of this audit is below:

{'AU': 1, 'BE': 5, 'CA': 36, 'DE': 14, 'GB': 5, 'RU': 3, 'US': 243}

(See Appendix Three for the log of these results)

4. Measuring Data Accuracy: Perform an Audit of the Data – Erroneous Postal Codes

Now that there is data from outside of the U.S., is there data from outside of the city of Detroit as well? From the USPS website[3], the postal code range is: 48201 to 48288. Any entry with a post code outside of this range is outside of Detroit. Similar to audit_country(), the function, audit_postcodes, iterates through the file, and records the value for the tag attribute, "addr:postcode" into a dictionary and prints out the results:



```python
postcodes = {}

def audit_postcodes():
    for event, elem in ET.iterparse(filename):
        if elem.tag == "tag":
            if elem.attrib['k'] == "addr:postcode":
                if elem.attrib['v'] in postcodes:
                    postcodes[elem.attrib['v']] += 1
                else:
                    postcodes[elem.attrib['v']] = 1
    pp.pprint(postcodes)
```

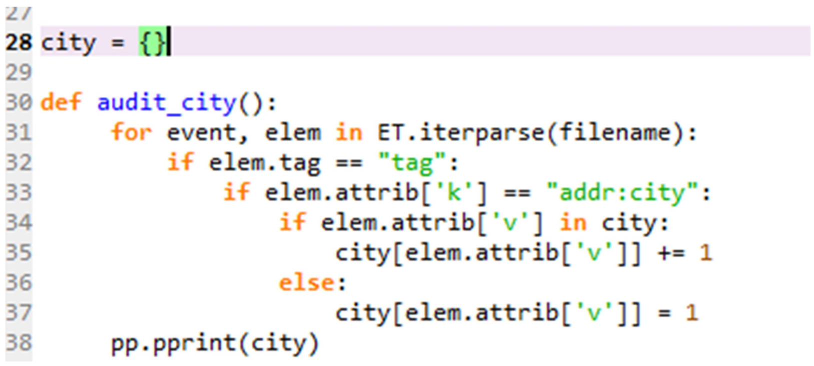
[3] (See USPS.com website: Link and list of zip codes attached.)

(See Appendix Four for a complete log of these results)

The results of this audit show that in addition to postcodes outside of Detroit, there are different versions of how this data is entered. Some are zip+4 and others start with “MI”.

5. Measuring Data Accuracy: Perform an Audit of the Data – Erroneous City Values

The function, audit_city(), again iterates through the file, looks at the tag attribute, "addr:city", and then records the value of each tag into the dictionary, city = {}:



```
city = {}

def audit_city():
    for event, elem in ET.iterparse(filename):
        if elem.tag == "tag":
            if elem.attrib['k'] == "addr:city":
                if elem.attrib['v'] in city:
                    city[elem.attrib['v']] += 1
                else:
                    city[elem.attrib['v']] = 1
    pp.pprint(city)
```

We see that there is data from outside the U.S.:

```
'Łódź': 2,
'Новокузнецк': 1,
'Слуцк': 3,
'Суми': 1,
'Чусовой': 1
```

(See Appendix Five for a complete log of these results)

In sum, the audit shows that there is a lot of abbreviations of streest that have to corrected, and that there is data in the osm file which violates the scope of this project -- load only Detroit elements into a MongoDB instance.

Part Two – Clean, Shape, and Load the Data

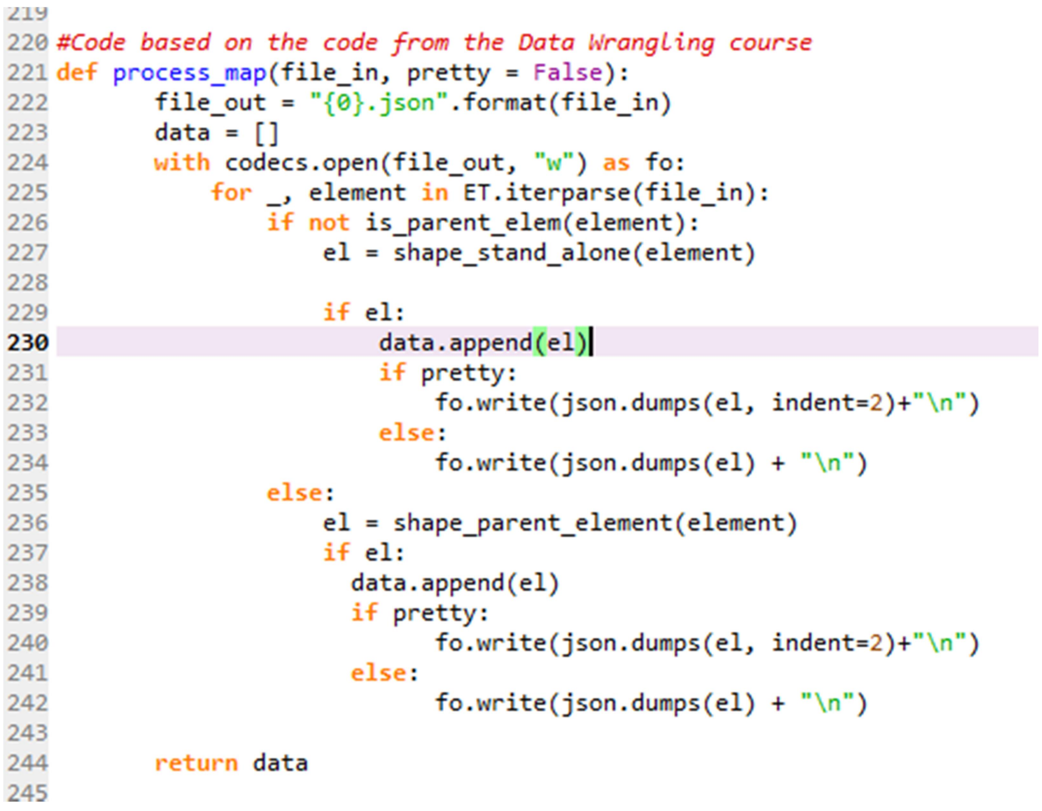
(Note: *The python file corresponding to the Audit section of this report can be found in the file, Part_Two_Clean_and_Shape)*

5. Cleaning and Shaping the Data

To re-state, the objective of the project is to create a MongoDB database instance that only contains street map data for the city of Detroit. The audit of the country data and postal codes found that there are elements in the dataset outside of Detroit. The cleaning plan is to iterate through the file. Identify those elements that are stand-alone, and those that have child elements.

If the element has children, then it will be inspected to see if it's a Detroit element and if it passes inspection, it will be cleaned and shaped into a JSON object.

Starting with the function from the Data Wrangling course, process_map(file_in, pretty = False), the function takes in the osm file and writes out a json file that will be loaded into MongoDB.



```
#Code based on the code from the Data Wrangling course
def process_map(file_in, pretty = False):
       file_out = "{0}.json".format(file_in)
       data = []
       with codecs.open(file_out, "w") as fo:
           for _, element in ET.iterparse(file_in):
               if not is_parent_elem(element):
                   el = shape_stand_alone(element)

                   if el:
                       data.append(el)
                       if pretty:
                           fo.write(json.dumps(el, indent=2)+"\n")
                       else:
                           fo.write(json.dumps(el) + "\n")
               else:
                   el = shape_parent_element(element)
                   if el:
                     data.append(el)
                     if pretty:
                           fo.write(json.dumps(el, indent=2)+"\n")
                     else:
                           fo.write(json.dumps(el) + "\n")

       return data
```

The function iterates through the osm and calls the is_parent_elem(element) function. This function checks to see if the element is a parent or a standalone. If it's not a parent element, then the attributes of the child element do not have to be cleaned, so the function, shape_stand_alone(elem) is called.

The functions, clean_city(city), clean_postcodes(postal_code), and clean_abbreviations(abbr) cleans the elements before they are loaded.

The function, check_element_id(elem), checks each element. If it the city, country, and postcodes are not valid, the element id is added to the set, element_Id_err_set.

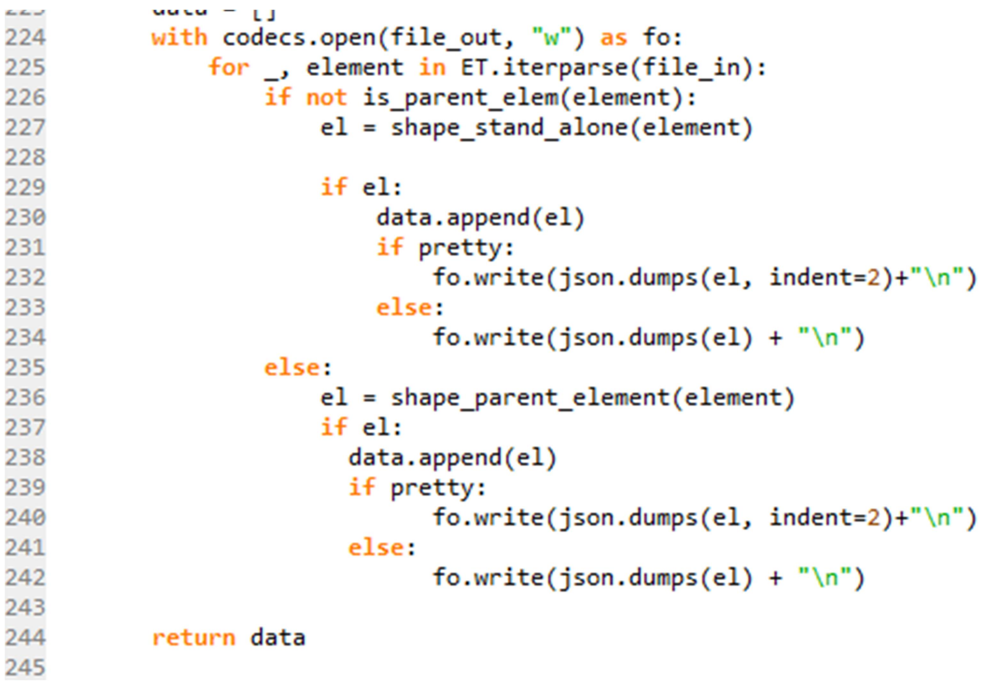
There are two functions that will shape the element. The function, shape_stand_alone(elem), shapes elements that do not have attachments. The function, shape_parent_element(elem), shapes elements that have attachments. The function, is_parent_elem(elem), checks whether the element contains child elements or not. This function iterates through the element and returns a load_node dictionary object which contains tag elements of the object. This dictionary also includes another dictionary object called, created, which contains the created information for the tag.

```python
def shape_stand_alone(elem):
    load_node = {}
    created = {}
    pos = []


    if elem.tag == "node" or elem.tag == "way" or elem.tag == "relation":
                load_node["type"] = elem.tag
                for tag in elem.iter(elem.tag):
                   keys = tag.keys()
                   for k in keys:

                       if k in CREATED:
                           created[k] = tag.attrib[k]
                       elif k == "lat":
                           pos.append(float(tag.attrib[k]))
                       elif k == "lon":
                           pos.append(float(tag.attrib[k]))
                       else:
                            load_node[k] = tag.attrib[k]

                load_node["created"] = created

                if len(pos) > 0:
                    load_node["pos"] = sorted(pos, reverse=True)
    return load_node
```

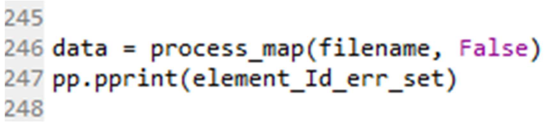
If the element contains child elements, then the attributes of the child will have to checked and cleaned. The function, shape_parent_element(elem), does this task. First, it checks to see if the element is valid by calling, check_element_id(elem). This function iterates through the child elements and calls the following check functions. If the attribute value fails these checks, it's element id is added to the error set, element_Id_err_set, on line 22:

Function Name	**What It Does**
clean_city(city)	Checks if the attribute value starts with "Detroit" and returns None if it doesn't.
is_valid_postcode(postcode)	Checks if the postcode is a legitimate Detroit, MI postal code based on the US postal code range for the city. This function calls clean_postcodes(postal_code) which strips out the "MI" before some of the post codes found in the audit. It also strips out the zip+4 postcodes for some the entries, and returns a 5 digit int number which is checked against the Detroit post code range.

Finally, check_element_id(elem) checks the country value for the element attribute, "addr:country". As we see saw from the audit, only values with "US" are legitimate elements for this project. If the element check passes, the function proceeds with the shaping and cleaning of the parent and child element and returns a load_node dictionary which is shaped into a JSON object.



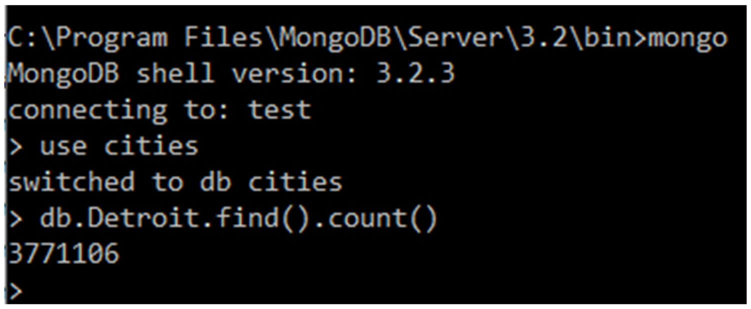
```
    data = []
    with codecs.open(file_out, "w") as fo:
        for _, element in ET.iterparse(file_in):
            if not is_parent_elem(element):
                el = shape_stand_alone(element)

                if el:
                    data.append(el)
                    if pretty:
                        fo.write(json.dumps(el, indent=2)+"\n")
                    else:
                        fo.write(json.dumps(el) + "\n")
            else:
                el = shape_parent_element(element)
                if el:
                  data.append(el)
                  if pretty:
                        fo.write(json.dumps(el, indent=2)+"\n")
                  else:
                        fo.write(json.dumps(el) + "\n")

    return data

```



```

data = process_map(filename, False)
pp.pprint(element_Id_err_set)

```

6. <u>Load the Element</u>

The JSON file is loaded into the collection, Detroit, in the cities MongoDB database using the Mongo import. A total of 3,771,106 records were loaded.



```
C:\Program Files\MongoDB\Server\3.2\bin>mongo
MongoDB shell version: 3.2.3
connecting to: test
> use cities
switched to db cities
> db.Detroit.find().count()
3771106
>
```

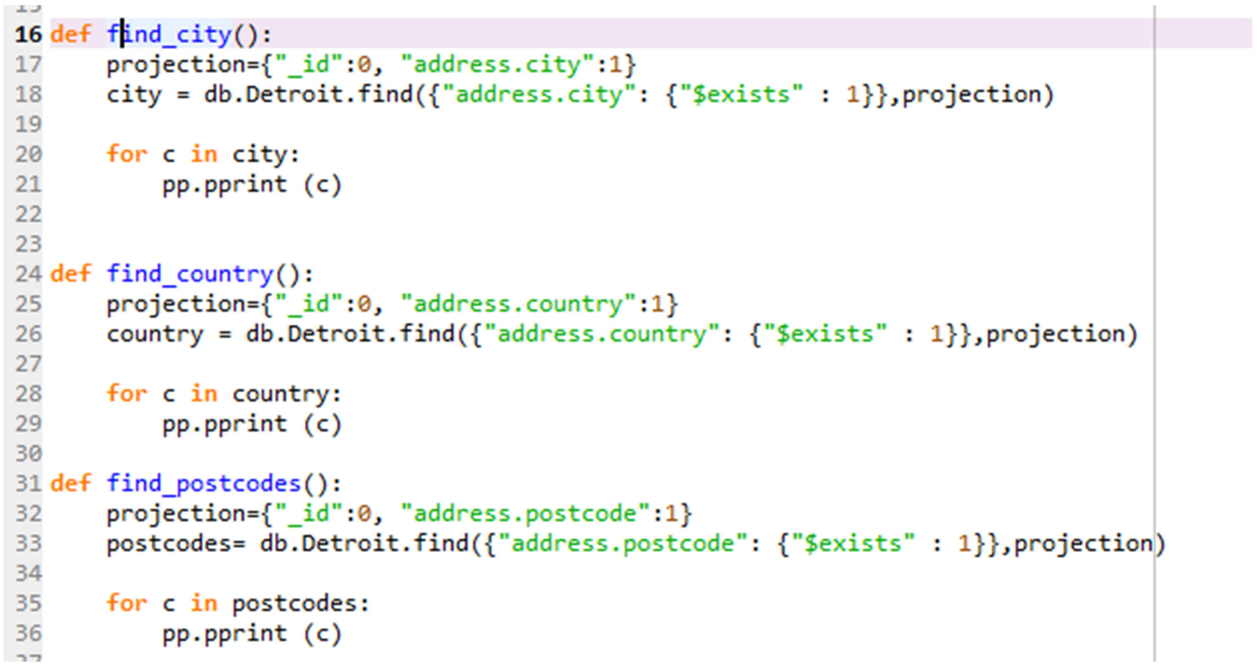
Part Three – Search Data with MongoDB

(Note: *The python file corresponding to the Audit section of this report can be found in the file, Part_Three_Search_Mongo.*)

7. Using Mongo DB Queries

   In the python file, searchMongo.py,

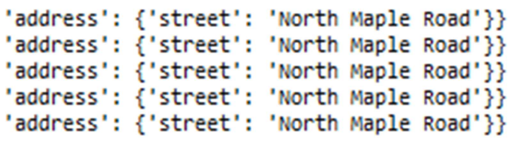
   The functions, find_city(),find_country(), and find_postcodes(), contain queries that are run against the Mongo database, cities in the Detroit collection.



```python
def find_city():
    projection={"_id":0, "address.city":1}
    city = db.Detroit.find({"address.city": {"$exists" : 1}},projection)

    for c in city:
        pp.pprint (c)


def find_country():
    projection={"_id":0, "address.country":1}
    country = db.Detroit.find({"address.country": {"$exists" : 1}},projection)

    for c in country:
        pp.pprint (c)

def find_postcodes():
    projection={"_id":0, "address.postcode":1}
    postcodes= db.Detroit.find({"address.postcode": {"$exists" : 1}},projection)

    for c in postcodes:
        pp.pprint (c)
```

   The results show that no elements for entries outside of Detroit are in the database.

   Finally, the function, find_addresses(), checks to see if the street names were correctly cleaned. Below is a snippet of the results:



```
'address': {'street': 'North Maple Road'}}
'address': {'street': 'North Maple Road'}}
'address': {'street': 'North Maple Road'}}
'address': {'street': 'North Maple Road'}}
'address': {'street': 'North Maple Road'}}
```

   See Appendicies, 6, 7, 8, and 9.

   A tally of the top users show that “Gone” and “'woodpeck_fixbot” were the top users. A full log of the users can be found in Appendix 10:

```
[('Gone', 948046),
 ('woodpeck_fixbot', 675089)
```

Finally, the function, find_notes(), searches the collection to find all of the notes from the users that entered data. A sampling of some of the notes show the difficulties that the users had to enter the data.

This may be due to the fact that Detroit, MI is struggling economically and many sites are demolished or closed.

```
{'note': 'Not clear where this is or was.'}
{'note': 'original node feature for park'}
{'note': 'Closed 2005, demolished 2013'}
{'note': 'Old Mackenzie High School which was demolished in 2013.'}
{'note': 'demolished. People seem to be interested in the location though.'}
{'note': 'There is no park listed here from Dearborn Heights. These are the '
         'fields for Annapolis High School.'}
{'note': 'Demolished 2000 or earlier'}
{'note': 'incorrectly positioned'}
{'note': 'Emma Thomas School was demolished in 2013.'}
{'note': 'Also formerly Highland Park Adult Education Center'}
{'note': 'Closed school, now demolished.'}
{'note': 'This has been torn down. Will remove when able to document '
         'boundaries on LPS versus park.'}
{'note': 'May be best to delete this. Believe closed when Sarah Fisher '
         'facility closed.'}
{'note': 'Not clear where this is. Does not seem like it was part of Webster '
         'as it was damaged by arson and closed.'}
```

(See Appendix 11 for full log.)

Recommendations:

The major finding from this project is that there was quite a lot of non-Detroit, MI data in the exported osm file. To correct the erroneous data and ensure that new data is entered correctly:

1. Develop algorithms that can iterate through the data and identify those elements that may have incorrect data based on incorrect country names, city data, postal codes, for a start.

2. For those elements with erroneous data, identify the user who entered them and isolate those records and inspect them for other errors

3. Build validation algorithms on the entering of data so that it’s entered in an uniform manner. For example, the zip codes should be a five digit integer. Anything other than that and the entry should not be accepted.