data_wrangling_xml

March 22, 2018

0.1 Wrangle-OpenStreetMap-Data

In this project I am using data mungling techniques to assess the quality of OpenStreetMap's (OSM) data and analyze using SQL for the state of New jersey. The data wrangling takes place programmatically, using Python for most of the process and exploring the data using SQL with SQLLITE

The dataset contains the data for New York City and is downloaded from overpass.api.de which is mirror image from https://www.openstreetmap.org . The data size is around 116 MB

0.1.1 Scope

OpenStreetMap (OSM) is a collaborative project to create a free editable map of the world. The creation and growth of OSM have been motivated by restrictions on use or availability of map information across much of the world, and the advent of inexpensive portable satellite navigation devices.

On the specific project, I am using data from https://www.openstreetmap.org and data mungling techniques, to assess the quality of their validity, accuracy, completeness, consistency and uniformity. The biggest part of the wrangling takes place programmatically using Python and then the dataset is entered into a SQLLITE database for further examination of any remaining elements that need attention. Finally, I perform some basic exploration and express some ideas for additional improvements.

0.1.2 Skills demonstrated

Assessment of the quality of data for validity, accuracy, completeness, consistency and uniformity. Parsing and gathering data from popular file formats such as .xml and .csv. Processing data from very large files that cannot be cleaned with spreadsheet programs. Storing, querying, and aggregating data using SQL.

0.1.3 The Dataset

OpenStreetMap's data are structured in well-formed XML documents (.osm files) that consist of the following elements: Nodes: "Nodes" are individual dots used to mark specific locations (such as a postal box). Two or more nodes are used to draw line segments or "ways". Ways: A "way" is a line of nodes, displayed as connected line segments. "Ways" are used to create roads, paths, rivers, etc. Relations: When "ways" or areas are linked in some way but do not represent the same physical thing, a "relation" is used to describe the larger entity they are part of. "Relations" are used to create map features, such as cycling routes, turn restrictions, and areas that are not contiguous.

The multiple segments of a long way, such as an interstate or a state highway are grouped into a "relation" for that highway. Another example is a national park with several locations that are separated from each other. Those are also grouped into a "relation".

All these elements can carry tags describing the name, type of road, and other attributes.

For this particular project, I am using a .osm file for part of NYC(most of Manhattan) which I downloaded from overpass.api.de. The dataset has a volume of 116 MB and can be downloaded from https://www.openstreetmap.org

0.1.4 Imports and definitions

```
In [235]: %matplotlib inline

          import xml.etree.cElementTree as ET
          from collections import defaultdict
          import re
          import pprint
          from operator import itemgetter
          from difflib import get_close_matches

          #For export to csv and data validation
          import csv
          import codecs
          import cerberus
          import geocoder
          import schema
```

```
In [237]: #OSM downloaded from openstreetmap
          NY_OSM = 'NYC.osm'
          SAMPLE_FILE = 'sample.osm'
          #The following .csv files will be used for data extraction from the XML.
          NODES_PATH = "nodes.csv"
          NODE_TAGS_PATH = "nodes_tags.csv"
          WAYS_PATH = "ways.csv"
          WAY_NODES_PATH = "ways_nodes.csv"
          WAY_TAGS_PATH = "ways_tags.csv"
```

```
In [259]: #Regular expressions
          PROBLEMCHARS = re.compile(r'[=\+/&<>;\'"\?%#$@\, \t\r\n]')
```

```
In [242]: def count_tags(filename):
          #         counts = dict()
          #         for line in ET.iterparse(filename):
          #             current = line[1].tag
          #             counts[current] = counts.get(current, 0) + 1
              counts = defaultdict(int)
              for line in ET.iterparse(filename):
                  current = line[1].tag
                  counts[current] += 1
              return counts
```

```
def test():
    tags = count_tags('NYC.osm')
    pprint.pprint(tags)
```

```
test()
```

```
defaultdict(<class 'int'>,
            {'bounds': 1,
             'member': 1913,
             'nd': 20172,
             'node': 16082,
             'osm': 1,
             'relation': 62,
             'tag': 12824,
             'way': 2110})
```

After reading through the OpenStreetMap Wiki I learned that data primitives for this data are nodes, ways, and relations.

For the purposes of this project, I will be looking at the node and way tags of this data set. Nodes are defined as a single point in space and is defined by longitude, latitude, and node id. Ways are an ordered list of nodes that either define a region, closed node, or some linear feature, open node.

0.1.5 Auditing the k Tags

Another area of interest was looking at the different 'k' tags in the data. I used three regular expressions to filter these tags and look for any problems that might have to be remedied before importing the data into a database. The first looks for tags with only lowercase letters, the second for lowercase letters separated by a colon, and the last flags any unwanted characters.

As we can see, the major elements are member, nd, node, relation, tag and way. We will audit these elements, clean them and store them in csv in order to be stored in SQLLITE

Now, we want to check whether "k" value for each "< tag >" has any issue or not. To see this, we divided the key type in four categories: "lower", for tags that contain only lowercase letters and are valid "lower_colon", for otherwise valid tags with a colon in their names "problemchars", for tags with problematic characters, and "other", for other tags that do not fall into the other three categories. I then used the iterparse method of ElementTree to compile a list of tags that fell into one of the three regular expression matches above.

We check this using the regex expressions and write in a separate file. Then we write the unique key tags belonging to each category in the respective file for later observation.

```
In [316]: #!/usr/bin/env python
          # -*- coding: utf-8 -*-

          import xml.etree.ElementTree as ET  # Use cElementTree or lxml if too slow

          OSM_FILE = "NYC.osm"  # Replace this with your osm file
          SAMPLE_FILE = "sample.osm"

          k = 10 # Parameter: take every k-th top level element

          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-genera
              """
              context = iter(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()


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

          with open(SAMPLE_FILE, 'ab') as output:

              # Write every kth top level element
              for i, element in enumerate(get_element(OSM_FILE)):
                  if i % k == 0:
                      output.write(ET.tostring(element, encoding='utf-8'))

          with open(SAMPLE_FILE, 'a') as output:
              output.write('</osm>')

In [261]: lo = set()
          lo_co = set()
          pro_co = set()
          oth = set()

          lower = re.compile(r'^([a-z]|_)*$')
          lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
          problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\, \t\r\n]')
```

```
def key_type(element, keys):
    if element.tag == "tag":
        k_value = element.attrib['k']
        if lower.search(k_value) is not None:
            keys['lower'] += 1
            lo.add(element.attrib['k'])
        elif lower_colon.search(k_value) is not None:
            keys['lower_colon'] += 1
            lo_co.add(element.attrib['k'])
        elif problemchars.search(k_value) is not None:
            keys["problemchars"] += 1
            pro_co.add(element.attrib['k'])
        else:
            keys['other'] += 1
            oth.add(element.attrib['k'])
        pass

    return keys

def write_data(data, filename):
    with open(filename, 'w') as f:
        for x in data:
            f.write(x + "\n")

def process_map1(filename):
    keys = {"lower": 0, "lower_colon": 0, "problemchars": 0, "other": 0}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)

    return keys



def test():
    # You can use another testfile 'map.osm' to look at your solution
    # Note that the assertion below will be incorrect then.
    # Note as well that the test function here is only used in the Test Run;
    # when you submit, your code will be checked against a different dataset.
    keys = process_map1('NYC.osm')
    write_data(lo, 'lower.txt')
    write_data(lo_co, 'lower_colon.txt')
    write_data(pro_co, 'problem_chars.txt')
    write_data(oth, 'other.txt')
    pprint.pprint(keys)
    #assert keys == {'lower': 5, 'lower_colon': 0, 'other': 1, 'problemchars': 1}


test()
```

```
{'lower': 220977, 'lower_colon': 322906, 'other': 18814, 'problemchars': 0}
```

0.1.6 Problems Encountered

Some of the problems that I noticed: The format for street names was not uniform with some street names being abbreviated and others not having the first letter capitalized. Inconsistent postal codes. Some of the codes were formatted with like 10020-2402 , 10020 , NY 11201

There was a postal code like just 83 The 'k' tags did not follow a specific format. Many similar tags were referenced by different names. Also, many tags had only been used once. Duplicate entries existed for some of the tags. This is due to two different data sources, Topologically Integrated Geographic Encoding and Referencing system (TIGER) and USGS Geographic Names Information System (GNIS)

0.1.7 Cleaning Street Names

The first part of the data that I cleaned was the abbreviated street names. To start I parsed through the way tags and return street names that were uncommon, according to a predefined list that I created.

```
In [270]: """
          Your task in this exercise has two steps:

          - audit the OSMFILE and change the variable 'mapping' to reflect the changes needed
              the unexpected street types to the appropriate ones in the expected list.
              You have to add mappings only for the actual problems you find in this OSMFILE,
              not a generalized solution, since that may and will depend on the particular are
          - write the update_name function, to actually fix the street name.
              The function takes a string with street name as an argument and should return th
              We have provided a simple test so that you see what exactly is expected
          """
          import xml.etree.cElementTree as ET
          from collections import defaultdict
          import re
          import pprint

          OSMFILE = "NYC.osm"
          street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)
          st_types = defaultdict(set)
          expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "La
                      "Trail", "Parkway", "Commons", "Bend", "Chase", "Circle", "Cove", "Cross
                      "Hollow", "Loop", "Park", "Pass", "Overlook", "Path", "Plaza", "Point", "
                      "Run", "Terrace", "Walk", "Way", "Trace", "View", "Vista","Concourse","S
                       "West","Mews","Broadway","Alley","street","avenue","Americas","Village"


          # UPDATE THIS VARIABLE
          mapping = { "St": "Street",
            "St.": "Street",
            "st": "Street",
            "st.": "Street",
            "ST": "Street",
            "Ave": "Avenue",
            "ave": "Avenue",
            "Avene": "Avenue",
            "avene": "Avenue",
            "Aveneu": "Avenue",
             "steet": "Street",
            "Steet": "Street",
            "Rd.": "Road",
            "W": "West",
            "N": "North",
            "S": "South",
            "E": "East"}


def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)


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


def audit(osmfile):
    osm_file = open(osmfile, "rb")
    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'])

    return street_types


def update_name(name, mapping):
    after = []
    # Split name string to test each part of the name;
    # Replacements may come anywhere in the name.
    for part in name.split(" "):
        # Check each part of the name against the keys in the correction dict
```

```
            if part in mapping.keys():
                # If exists in dict, overwrite that part of the name with the dict value
                part = mapping[part]
            # Assemble each corrected piece of the name back together.
            after.append(part)
        # Return all pieces of the name as a string joined by a space.
        return " ".join(after)


#     for w in mapping.keys():
#         if w in name:
#             if flag:
#                 continue
#             # Replace abbrev. name in string with full name value from the mapping
#             name = name.replace(w, mapping[w], 1)
#             # If St., flag to not check again in this string looking for St since
#             # re.compile() might be better
#             if w == "St.":
#                 flag = True

    return name


def test():
    st_types = audit(OSMFILE)
    #assert len(st_types) == 3
    pprint.pprint(dict(st_types))

    for st_type, ways in st_types.items():
        for name in ways:
            better_name = update_name(name, mapping)
            print (name, "=>", better_name)




    test()
```

```
{'1': {'36th St Front 1'},
 '10003': {'Irvine Place, #1, New York, NY, 10003'},
 '109': {'Central Park South Suite 109'},
 '1801': {'505th 8th Avenue Suite 1801'},
 '1807': {'5th AVE 1807'},
 '21G': {'East 80th Street, 21G'},
 '27th': {'W 27th'},
 '29th': {'29th'},
 '2N': {'400th West 20th St., Suite 2N'},
 '3': {'Irving Place #3'},
'301': {'E 55th St Ste. 301'},
'306': {'West 30th Street Suite 306'},
'42nd': {'West 42nd'},
'4B': {'Union Avenue 4B'},
'500': {'Main St., Suite 500'},
'633': {'633'},
'861': {'861'},
'A': {'Avenue A'},
'Atrium': {'Broadway Atrium'},
'Ave': {'Norman Ave', 'Union Ave', '10th Ave', 'Third Ave'},
'Avene': {'Madison Avene', '8th Avene'},
'B': {'Avenue B'},
'Blvd': {'Vernon Blvd'},
'Broadway.': {'Broadway.'},
'Brooklyn': {'334 Furman St, Brooklyn'},
'Bushwick': {'Bushwick'},
'C': {'Avenue C'},
'Center': {'World Financial Center', 'Gotham Center', 'World Trade Center'},
'D': {'Avenue D'},
'Finest': {'Avenue Of The Finest'},
'Floor': {'Madison Avenue, 15th Floor', 'Wall Street 12th Floor'},
'Floor)': {'Manhattan Avenue (2nd Floor)'},
'Fulton': {'Old Fulton'},
'Heights': {'Columbia Heights'},
'Highline': {'Highline'},
'Lafayette': {'Lafayette'},
'Level': {'Madison Ave Arcage Level'},
'Macdougal': {'Macdougal'},
'NY': {'405 West 23rd Street, New York, NY',
       '54th W 39th St New York, NY',
       'West 49th Street New York NY'},
'Oval': {'Stuyvesant Oval'},
'Piers': {'Northside Piers', 'Chelsea Piers'},
'Rd': {'43rd Rd'},
'Rico': {'Avenue Of Puerto Rico'},
'Roadbed': {'Delancey Street Eb Roadbed'},
'S': {'Central Park S', 'Park Ave S', 'Park Avenue S'},
'ST': {'110 SIXTH AVE. AT WATTS ST', 'N 9th ST'},
'Slip': {'Catherine Slip',
         'Coenties Slip',
         'Old Slip',
         'Peck Slip',
         'Pike Slip',
         'Rutgers Slip'},
'St': {'205 W 58th St',
       '330 E 84th St',
       '362nd Grand St',
       '56th St',
        'Broad St',
        'E 43rd St',
        'E Houston St',
        'East Houston St',
        'Hewes St',
        'Hudson St',
        'Jackson St',
        'Mott St',
        'N 7th St',
        'State St & Water St',
        'W 26th St',
        'W 35th St',
        'W 36th St',
        'W 57th St',
        'West 32nd St',
        'West 37 St',
        'Wooster St'},
 'St.': {'13th St.',
         'Devoe St.',
         'E. 54th St.',
         'East 73rd St.',
         'East 86th St.',
         'South 4th St.',
         'West 44th St.'},
 'Steet': {'West 25th Steet', 'West 8th Steet'},
 'Terminal': {'Grand Central Terminal'},
 'Track': {'Track'},
 'Unidos': {'519 9th Ave, New York, NY 10018, Estados Unidos'},
 'Uniti': {'3rd Ave, New York, NY 10028, Stati Uniti'},
 'Warren': {'Warren'},
 'Yards': {'Hudson Yards'},
 'ave': {'10th ave', '11th ave', '110 West 51st at 6 ave', '5th ave'},
 'st': {'grand st', 'W 35th st', 'South 4th st'}}
grand st => grand Street
W 35th st => West 35th Street
South 4th st => South 4th Street
Avenue C => Avenue C
E. 54th St. => E. 54th Street
13th St. => 13th Street
East 86th St. => East 86th Street
East 73rd St. => East 73rd Street
South 4th St. => South 4th Street
West 44th St. => West 44th Street
Devoe St. => Devoe Street
10th ave => 10th Avenue
11th ave => 11th Avenue
110 West 51st at 6 ave => 110 West 51st at 6 Avenue
5th ave => 5th Avenue
```

```
West 42nd => West 42nd
Central Park S => Central Park South
Park Ave S => Park Avenue South
Park Avenue S => Park Avenue South
West 25th Steet => West 25th Street
West 8th Steet => West 8th Street
400th West 20th St., Suite 2N => 400th West 20th St., Suite 2N
Avenue D => Avenue D
West 49th Street New York NY => West 49th Street New York NY
54th W 39th St New York, NY => 54th West 39th Street New York, NY
405 West 23rd Street, New York, NY => 405 West 23rd Street, New York, NY
W 27th => West 27th
Madison Avene => Madison Avenue
8th Avene => 8th Avenue
Avenue B => Avenue B
Avenue A => Avenue A
Main St., Suite 500 => Main St., Suite 500
Columbia Heights => Columbia Heights
Delancey Street Eb Roadbed => Delancey Street Eb Roadbed
Irving Place #3 => Irving Place #3
505th 8th Avenue Suite 1801 => 505th 8th Avenue Suite 1801
State St & Water St => State Street & Water Street
Mott St => Mott Street
W 57th St => West 57th Street
Broad St => Broad Street
Jackson St => Jackson Street
56th St => 56th Street
205 W 58th St => 205 West 58th Street
Hewes St => Hewes Street
330 E 84th St => 330 East 84th Street
362nd Grand St => 362nd Grand Street
West 37 St => West 37 Street
W 36th St => West 36th Street
W 26th St => West 26th Street
E Houston St => East Houston Street
E 43rd St => East 43rd Street
West 32nd St => West 32nd Street
East Houston St => East Houston Street
Wooster St => Wooster Street
Hudson St => Hudson Street
W 35th St => West 35th Street
N 7th St => North 7th Street
29th => 29th
Macdougal => Macdougal
World Financial Center => World Financial Center
Gotham Center => Gotham Center
World Trade Center => World Trade Center
Warren => Warren
```

```
Pike Slip => Pike Slip
Coenties Slip => Coenties Slip
Old Slip => Old Slip
Rutgers Slip => Rutgers Slip
Catherine Slip => Catherine Slip
Peck Slip => Peck Slip
Broadway Atrium => Broadway Atrium
West 30th Street Suite 306 => West 30th Street Suite 306
Avenue Of The Finest => Avenue Of The Finest
Avenue Of Puerto Rico => Avenue Of Puerto Rico
Union Avenue 4B => Union Avenue 4B
Bushwick => Bushwick
861 => 861
Norman Ave => Norman Avenue
Union Ave => Union Avenue
10th Ave => 10th Avenue
Third Ave => Third Avenue
Manhattan Avenue (2nd Floor) => Manhattan Avenue (2nd Floor)
Old Fulton => Old Fulton
Track => Track
Stuyvesant Oval => Stuyvesant Oval
Highline => Highline
Madison Avenue, 15th Floor => Madison Avenue, 15th Floor
Wall Street 12th Floor => Wall Street 12th Floor
519 9th Ave, New York, NY 10018, Estados Unidos => 519 9th Ave, New York, NY 10018, Estados Un
3rd Ave, New York, NY 10028, Stati Uniti => 3rd Ave, New York, NY 10028, Stati Uniti
Broadway. => Broadway.
5th AVE 1807 => 5th AVE 1807
East 80th Street, 21G => East 80th Street, 21G
Madison Ave Arcage Level => Madison Avenue Arcage Level
Lafayette => Lafayette
Central Park South Suite 109 => Central Park South Suite 109
E 55th St Ste. 301 => East 55th Street Ste. 301
Hudson Yards => Hudson Yards
36th St Front 1 => 36th Street Front 1
334 Furman St, Brooklyn => 334 Furman St, Brooklyn
110 SIXTH AVE. AT WATTS ST => 110 SIXTH AVE. AT WATTS Street
N 9th ST => North 9th Street
Vernon Blvd => Vernon Blvd
Irvine Place, #1, New York, NY, 10003 => Irvine Place, #1, New York, NY, 10003
Northside Piers => Northside Piers
Chelsea Piers => Chelsea Piers
633 => 633
43rd Rd => 43rd Rd
Grand Central Terminal => Grand Central Terminal
```

0.1.8 Cleaning Postal Codes

Another problem that I noticed while parsing through the data was that the postal codes were presented in different ways. Below is an excerpt of some of the different formats for postal codes.

```
In [232]: def is_zip_code(elem):
              return (elem.attrib['k'] == "addr:postcode")

          def audit_zip(osmfile):
              osm_file = open(osmfile, "rb")
              prob_zip = 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_zip_code(tag):

                              if len(tag.attrib['v']) != 5:
                                  if (tag.attrib['v'][0])=='N':
                                      print ("Fixed Zip:   ", tag.attrib['v'], "=>", tag.attril
                                  else:
                                      print ("Fixed Zip:   ", tag.attrib['v'], "=>", tag.attril

                              elif tag.attrib['v'][0:2] != '96':
                                  #print ("Fixed Zip:   ", tag.attrib['v'], "=>", tag.attrib['
                                  prob_zip.add(tag.attrib['v'])
                              elif len(tag.attrib['v']) == 5:
                                  prob_zip.add(tag.attrib['v'])
              osm_file.close()
              return prob_zip

          print ("Possible problematic zip codes:")
          audit_zip('NYC.osm')
```

```
Possible problematic zip codes:



Fixed Zip:    10020-2402 => 10020
Fixed Zip:    NY 11201 => 11201
Fixed Zip:    10011-6832 => 10011
Fixed Zip:    NY 10012 => 10012
Fixed Zip:    NY 10002 => 10002
Fixed Zip:    NY 10111 => 10111
Fixed Zip:    10012-3332 => 10012
Fixed Zip:    NY 10075 => 10075
Fixed Zip:    10018-4527 => 10018
Fixed Zip:    NY 10075 => 10075
Fixed Zip:    10016-0122 => 10016
Fixed Zip:    NY 10036 => 10036
Fixed Zip:    NY 10011 => 10011
Fixed Zip:    100014 => 10001
Fixed Zip:    NY 10003 => 10003
Fixed Zip:    NY 10003 => 10003
Fixed Zip:    10017-6927 => 10017
Fixed Zip:    NY 11201 => 11201
Fixed Zip:    10075-0381 => 10075
Fixed Zip:    NY 10003 => 10003
Fixed Zip:    NY 10036 => 10036
Fixed Zip:    NY 10036 => 10036
Fixed Zip:    NY 10036 => 10036
Fixed Zip:    NY 10007 => 10007
Fixed Zip:    NY  10011 => 10011
Fixed Zip:    NY 10011 => 10011
Fixed Zip:    NY 10001 => 10001
Fixed Zip:    10001-2062 => 10001
Fixed Zip:    10019-9998 => 10019
Fixed Zip:    NY 10016 => 10016
Fixed Zip:    New York, NY 10065 => 10065
Fixed Zip:    10002-1013 => 10002
```

```
Out[232]: {'07086',
           '10001',
           '10002',
           '10003',
           '10004',
           '10005',
           '10006',
           '10007',
           '10009',
           '10010',
           '10011',
           '10012',
           '10013',
           '10014',
           '10016',
           '10017',
           '10018',
           '10019',
           '10020',
           '10021',
           '10022',
           '10023',
           '10028',
           '10036',
           '10038',
           '10044',
           '10045',
           '10048',
           '10055',
           '10065',
           '10069',
           '10075',
           '10103',
           '10107',
           '10110',
           '10111',
           '10112',
           '10118',
           '10121',
           '10123',
           '10128',
           '10152',
           '10153',
           '10154',
           '10155',
           '10168',
           '10169',
           '10173',
           '10174',
           '10271',
           '10275',
           '10280',
           '10281',
           '10282',
           '11101',
           '11106',
           '11109',
           '11201',
           '11206',
           '11211',
           '11222',
           '11226',
           '11249',
           '11251'}
```

Functions to update and correct the abbreviated street names and postal codes before uploading into csv files to load into the sqllite database

```
In [150]: def fix_element(elem):

              # Fix Street Names:

              # mapping provides a dictionary for updating potentially problematic street type
              # Dictionary contents were updated iteratively, based on the audit results
```

```
    # UPDATE THIS VARIABLE

    mapping = { "St": "Street",
            "St.": "Street",
            "st": "Street",
            "st.": "Street",
            "ST": "Street",
            "Ave": "Avenue",
            "ave": "Avenue",
            "Avene": "Avenue",
            "avene": "Avenue",
            "Aveneu": "Avenue",
             "steet": "Street",
            "Steet": "Street",
            "Rd.": "Road",
            "W": "West",
            "N": "North",
            "S": "South",
            "E": "East"}

def fix_street(elem):
    street_types = defaultdict(set)
    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'])
                for st_type, ways in street_types.iteritems():
                    for name in ways:
                        for key,value in mapping.items():
                            n = street_type_re.search(name)
                            if n:
                                street_type = n.group()
                                if street_type not in expected:
                                    if street_type in mapping:
                                        better_name = name.replace(key,value)
                                        if better_name != name:
                                            print ("Fixed Street:", tag.attrib['v'],
                                            tag.attrib['v'] = better_name
                                            return
    # Fix Zip Codes:

def fix_zip(elem):
    if elem.tag == "node" or elem.tag == "way":
        for tag in elem.iter("tag"):
            if is_zip_code(tag):
                if len(tag.attrib['v']) != 5:
                    if (tag.attrib['v'][0])=='N':
                        #print ("Fixed Zip:   ", tag.attrib['v'], "=>", tag.attrib['
                        tag.attrib['v'] = tag.attrib['v'][-5:]
                    else:
                        #print ("Fixed Zip:    ", tag.attrib['v'], "=>", tag.attrib['
                        tag.attrib['v'] = tag.attrib['v'][0:5]



    fix_street(elem)
    fix_zip(elem)
```

0.1.9 Importing Dataset to Database

After performing the most of the cleaning through Python, I can store the dataset in the database to examine the PROBLEMATIC elements and explore it further. I am using sqllite to present a generic solution . Initially, I am exporting the data in .csv files using the schema below, creating the tables in sqllite database and importing the .csvs.

0.1.10 Exporting dataset to .CSVs

```
In [186]: SCHEMA = {
              'node': {
                  'type': 'dict',
                  'schema': {
                      'id': {'required': True, 'type': 'integer', 'coerce': int},
                      'lat': {'required': True, 'type': 'float', 'coerce': float},
                      'lon': {'required': True, 'type': 'float', 'coerce': float},
                      'user': {'required': True, 'type': 'string'},
                      'uid': {'required': True, 'type': 'integer', 'coerce': int},
                      'version': {'required': True, 'type': 'string'},
                      'changeset': {'required': True, 'type': 'integer', 'coerce': int},
                      'timestamp': {'required': True, 'type': 'string'}
                  }
              },
              'node_tags': {
                  'type': 'list',
                  'schema': {
                      'type': 'dict',
                      'schema': {
                          'id': {'required': True, 'type': 'integer', 'coerce': int},
                          'key': {'required': True, 'type': 'string'},
                          'value': {'required': True, 'type': 'string'},
                          'type': {'required': True, 'type': 'string'}
                      }
                  }
              },
              'way': {
                  'type': 'dict',
                  'schema': {
                      'id': {'required': True, 'type': 'integer', 'coerce': int},
                      'user': {'required': True, 'type': 'string'},
                      'uid': {'required': True, 'type': 'integer', 'coerce': int},
                      'version': {'required': True, 'type': 'string'},
                      'changeset': {'required': True, 'type': 'integer', 'coerce': int},
                      'timestamp': {'required': True, 'type': 'string'}
                  }
              },
              'way_nodes': {
                  'type': 'list',
                  'schema': {
                      'type': 'dict',
                      'schema': {
                          'id': {'required': True, 'type': 'integer', 'coerce': int},
                          'node_id': {'required': True, 'type': 'integer', 'coerce': int},
                          'position': {'required': True, 'type': 'integer', 'coerce': int}
                      }
                  }
              },
              'way_tags': {
                  'type': 'list',
                  'schema': {
                      'type': 'dict',
                      'schema': {
                          'id': {'required': True, 'type': 'integer', 'coerce': int},
                          'key': {'required': True, 'type': 'string'},
                          'value': {'required': True, 'type': 'string'},
                          'type': {'required': True, 'type': 'string'}
                      }
                  }
              }
          }
```

In [256]:
```
# Define the files

OSM_PATH = "example.osm"
NODES_PATH = "nodes.csv"
NODE_TAGS_PATH = "nodes_tags.csv"
WAYS_PATH = "ways.csv"
WAY_NODES_PATH = "ways_nodes.csv"
WAY_TAGS_PATH = "ways_tags.csv"

LOWER_COLON = re.compile(r'^([a-z]|_)+:([a-z]|_)+')
PROBLEMCHARS = re.compile(r'[=\+/&<>;\'"\?%#$@\, \t\r\n]')
# Make sure the fields order in the csvs matches the column order
#in the sql table schema
NODE_FIELDS = ['id', 'lat', 'lon','user','uid','version','changeset','timestamp']
```

```
         NODE_TAGS_FIELDS = ['id', 'key', 'value', 'type']
         WAY_FIELDS = ['id', 'user', 'uid', 'version', 'changeset', 'timestamp']
         WAY_TAGS_FIELDS = ['id', 'key', 'value', 'type']
         WAY_NODES_FIELDS = ['id', 'node_id', 'position']

In [257]: def shape_element(element):
             """Clean and shape node or way XML element to Python dict

             Arrgs:
                 element (element): An element of the XML tree

             Returns:
                 dict: if element is a node, the node's attributes and tags.
                       if element is a way, the ways attributes and tags along with the
                       nodes that form the way.
             """
             node_attribs = {}
             way_attribs = {}
             way_nodes = []
             tags = []  # Handle secondary tags the same way for both node and way elements
             fix_element(element)
             if element.tag == 'node':
                 node_attribs['id'] = element.get('id')
                 node_attribs['lat'] = element.get('lat')
                 node_attribs['lon'] = element.get('lon')
                 node_attribs['user'] = element.get('user')
                 node_attribs['uid'] = element.get('uid')
                 node_attribs['version'] = element.get('version')
                 node_attribs['changeset'] = element.get('changeset')
                 node_attribs['timestamp'] = element.get('timestamp')
                 for child in element:
                     if child.tag == 'tag':
                         tag = {'id': node_attribs['id']}
                         k = child.get('k')
                         if not PROBLEMCHARS.search(k):
                             k = k.split(':', 1)
                             tag['key'] = k[-1]
                             tag['value'] = child.get('v')
                             if len(k) == 1:
                                 tag['type'] = 'regular'
                             elif len(k) == 2:
                                 tag['type'] = k[0]
                         tags.append(tag)
                 return {'node': node_attribs, 'node_tags': tags}
             elif element.tag == 'way':
                 counter = 0
                 way_attribs['id'] = element.get('id')
                 way_attribs['user'] = element.get('user')
```

```
            way_attribs['uid'] = element.get('uid')
            way_attribs['version'] = element.get('version')
            way_attribs['changeset'] = element.get('changeset')
            way_attribs['timestamp'] = element.get('timestamp')
            for child in element:
                if child.tag == 'tag':
                    tag = {'id': way_attribs['id']}
                    k = child.get('k')
                    if not PROBLEMCHARS.search(k):
                        k = k.split(':', 1)
                        tag['key'] = k[-1]
                        tag['value'] = child.get('v')
                        if len(k) == 1:
                            tag['type'] = 'regular'
                        elif len(k) == 2:
                            tag['type'] = k[0]
                    tags.append(tag)
                if child.tag == 'nd':
                    nd = {'id': way_attribs['id']}
                    nd['node_id'] = child.get('ref')
                    nd['position'] = counter
                    way_nodes.append(nd)
                counter += 1
            return {'way': way_attribs, 'way_nodes': way_nodes, 'way_tags': tags}
```

```
In [212]: def get_element(osm_file, tags=('node', 'way', 'relation')):
              """Yield element if it is the right type of tag"""

              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()
```

```
In [250]: def validate_element(element, validator, schema=SCHEMA):
              """Raise ValidationError if element does not match schema"""
              if validator.validate(element, schema) is not True:
                  field, errors = next(iter(validator.errors.items()))
                  pprint.pprint(element)
                  message_string = "\nElement of type '{0}' has the following errors:\n{1}"
                  error_string = pprint.pformat(errors)

                  raise Exception(message_string.format(field, error_string))
```

```
In [207]: class UnicodeDictWriter(csv.DictWriter, object):
              """Extend csv.DictWriter to handle Unicode input"""
```

```
            def writerow(self, row):
                super(UnicodeDictWriter, self).writerow({
                    k: (v.encode('utf-8') if isinstance(v, unicode) else v)
                    for k, v in row.items()
                })

            def writerows(self, rows):
                for row in rows:
                    self.writerow(row)

In [248]: def process_map(file_in, validate):
            """Iteratively process each XML element and write to csv(s)"""

            with codecs.open(NODES_PATH, 'w') as nodes_file, \
                 codecs.open(NODE_TAGS_PATH, 'w') as nodes_tags_file, \
                 codecs.open(WAYS_PATH, 'w') as ways_file, \
                 codecs.open(WAY_NODES_PATH, 'w') as way_nodes_file, \
                 codecs.open(WAY_TAGS_PATH, 'w') as way_tags_file:

                nodes_writer = UnicodeDictWriter(nodes_file, NODE_FIELDS)
                node_tags_writer = UnicodeDictWriter(nodes_tags_file, NODE_TAGS_FIELDS)
                ways_writer = UnicodeDictWriter(ways_file, WAY_FIELDS)
                way_nodes_writer = UnicodeDictWriter(way_nodes_file, WAY_NODES_FIELDS)
                way_tags_writer = UnicodeDictWriter(way_tags_file, WAY_TAGS_FIELDS)

                nodes_writer.writeheader()
                node_tags_writer.writeheader()
                ways_writer.writeheader()
                way_nodes_writer.writeheader()
                way_tags_writer.writeheader()

                validator = cerberus.Validator()

                for element in get_element(file_in, tags=('node', 'way')):
                    #pprint.pprint(element)
                    el = shape_element(element)
                    #pprint.pprint(el)
                    if el:
                        if validate is True:
                            validate_element(el, validator)

                        if element.tag == 'node':
                            nodes_writer.writerow(el['node'])
                            node_tags_writer.writerows(el['node_tags'])
                        elif element.tag == 'way':
                            ways_writer.writerow(el['way'])
                            way_nodes_writer.writerows(el['way_nodes'])
                            way_tags_writer.writerows(el['way_tags'])
```

```
In [262]: process_map('NYC.osm', validate=True)
```

The csv files generated are imported into the sqllite database names nycosmdata.db Now let us explore our data more using SQL.

```
In [277]: # import SQLLITE to explore the data loaded from csv files into nycosmdata.db
          import pandas as pd
          import sqlite3
          conn = sqlite3.connect("nycosmdata.db")
          df = pd.read_sql_query("select * from nodes limit 5;", conn)
          df
```

```
Out[277]:          id        lat        lon      user      uid  version  changeset  \
          0  30978735  40.773892 -73.971605  mikercpc  4124310        4   40180450
          1  30978738  40.773908 -73.972081  mikercpc  4124310        4   40180450
          2  30978740  40.773899 -73.972518  mikercpc  4124310        5   40180450
          3  30978741  40.773899 -73.972834  mikercpc  4124310        4   40180450
          4  30978743  40.773990 -73.973139  mikercpc  4124310        4   40180450

                        timestamp
          0  2016-06-21T13:06:21Z
          1  2016-06-21T13:06:21Z
          2  2016-06-21T13:06:21Z
          3  2016-06-21T13:06:21Z
          4  2016-06-21T13:06:21Z
```

```
In [298]: #Letus check the number of rows in each table

          conn = sqlite3.connect("nycosmdata.db")
          cur = conn.cursor()
          counts = cur.execute("""select count(*)||'  nodes' from nodes union
                                  Select count(*)||'   nodes_tags' from nodes_tags union
                                  Select count(*)||'   ways' from ways union
                                  Select count(*)||'   ways_nodes' from ways_nodes union
                                  Select count(*)||'   ways_tags' from ways_tags
              ;""").fetchall()
          #results = cur.fetchall()
          pprint.pprint(counts)

[('146130   nodes_tags',),
 ('408564   ways_tags',),
 ('412798  nodes',),
 ('618229   ways_nodes',),
 ('71682   ways',)]
```

```
In [297]: # Number of unique users:

          no_of_users = cur.execute("""SELECT COUNT(DISTINCT(users.uid))
          FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways) users;""").fetchall()
          print(no_of_users)
```

```
[(1756,)]
```

```
In [303]: # Top 10 users

          Top_10_users = cur.execute("""SELECT nodes_ways."user" AS "User", COUNT(*) AS "Users"
          FROM (SELECT "user" FROM nodes
                UNION ALL
                SELECT "user" FROM ways) AS nodes_ways
          GROUP BY nodes_ways."user"
          ORDER BY "Users" DESC
          LIMIT 10;""").fetchall()
          pprint.pprint(Top_10_users)
```

```
[('Rub21_nycbuildings', 217134),
 ('lxbarth_nycbuildings', 71329),
 ('robge', 63708),
 ('ALE!', 13715),
 ('mikercpc', 10699),
 ('minewman', 8758),
 ('Korzun', 6626),
 ('tre1994', 4015),
 ('celosia_nycbuildings', 3570),
 ('LizBarry_nycbuildings', 2869)]
```

```
In [296]: # Top 10 zip codes

          top_zip_codes = 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
          LIMIT 10;""").fetchall()
          pprint.pprint(top_zip_codes)
```

```
[('11211', 6148),
 ('11222', 5883),
 ('10011', 2826),
 ('10003', 2565),
 ('10014', 2551),
 ('10002', 2487),
 ('11206', 2300),
 ('11101', 2256),
 ('10013', 2189),
 ('11249', 1944)]
```

```
In [302]: # Top amenities in NYC

          top_amenities = cur.execute("""SELECT value AS "Amenity",
          COUNT(value) AS "Occurrences"
          FROM(SELECT *
          FROM nodes_tags
          UNION ALL
          SELECT *
          FROM nodes_tags) as tags
          WHERE key = 'amenity'
          GROUP BY value
          ORDER BY "Occurrences" DESC
          LIMIT 10""").fetchall()
          pprint.pprint(top_amenities)
```

```
[('bicycle_parking', 5396),
 ('restaurant', 3256),
 ('cafe', 1166),
 ('fast_food', 788),
 ('bicycle_rental', 668),
 ('bar', 636),
 ('bank', 490),
 ('bench', 380),
 ('school', 380),
 ('embassy', 326)]
```

It is intresting to note that bicycle parking is the top amenity in Newyork city. I was expecting restaurants to comeup first but this makes sense when you think about the really high amount of bicycle usage.

```
In [309]: top_banks = cur.execute(""" SELECT value AS "Bank", COUNT(value) AS "ATMs"
          FROM (Select * from nodes_tags union select * from ways_tags)
                 WHERE id in(SELECT id from (Select * from nodes_tags
                             union select * from ways_tags)
                              WHERE upper(value) = 'ATM' or upper(value) = 'BANK')
                 AND  upper(key) in ('NAME','OPERATOR')
                 GROUP BY value ORDER BY "ATMs" DESC LIMIT 10;""").fetchall()
          pprint.pprint(top_banks)
```

```
[('Chase', 77),
 ('Citibank', 44),
 ('Bank of America', 36),
 ('TD Bank', 26),
 ('HSBC', 23),
 ('Capital One', 20),
 ('Wells Fargo', 10),
 ('Santander', 9),
 ('Valley National Bank', 8),
 ('UNFCU', 6)]
```

```
In [306]: #Number of cafes, hotels, pubs, and restaurants:

          top_restaurants = cur.execute("""SELECT value, COUNT(*)
          FROM (SELECT * from nodes_tags as T UNION ALL
                SELECT * from ways_tags as Z) as Q
          WHERE (value = 'restaurant' OR value = 'hotel' OR
                 value = 'pub' OR value = 'cafe')
          GROUP BY value """).fetchall()
          pprint.pprint(top_restaurants)
```

```
[('cafe', 606), ('hotel', 379), ('restaurant', 1728)]
```

```
In [301]: top_cusines = cur.execute("""SELECT value, COUNT(*)
          FROM (SELECT * from nodes_tags as T UNION ALL
                SELECT * from ways_tags as Z) as Q
          WHERE (key = 'cuisine') group by value ORDER BY 2 DESC LIMIT 10;""").fetchall()
          pprint.pprint(top_cusines)
```

```
[('coffee_shop', 162),
 ('italian', 117),
 ('pizza', 109),
 ('mexican', 96),
 ('burger', 87),
 ('american', 83),
 ('chinese', 56),
 ('japanese', 55),
 ('sandwich', 46),
 ('indian', 44)]
```

Looking at the number of restaurants by cusine tells me they are way under reported and on further analysis the values reported were vastly different .Hence they weren't grouped together. This also gave me ideas for how the data might be improved in the future, which I discuss in the next section.

0.1.11 Additional Ideas

There are several areas of improvement of the project in the future. The first one is on the completeness of the data. All the above analysis is based on a dataset that reflects a big part of Newyork city manhattab but not the whole island. The reason for this is the lack of a way to download a dataset for the entire place without including parts of the neighboring areas. The analyst has to either select a part of the city or select a wider area that includes parts of other NYC boroughs and parts of New jersey.

The data could be improved by standardizing the information that is included with the node tags for place like cafe, hotel, pub, and restaurant upon data entry. Some tags give a great deal of information about the establishment while others provide very little detail. This would help to increase the quality of the openstreetmap data and enhance the consumer experience. The financial institutions are not listed with uniformity either.

0.1.12 References

```
In [ ]: https://download.geofabrik.de/north-america.html
        https://wiki.openstreetmap.org/wiki/OSM_XML
        https://gist.github.com/carlward/54ec1c91b62a5f911c42#file-sample_project-md
        https://github.com/YannisPap/Wrangle-OpenStreetMap-Data/blob/master/Notebook/Wrangle-O
        http://overpass-api.de/
        https://www.python.org/
```