Wrangling OpenStreetMap Data¶

In this project, I will apply data wrangling techniques to prepare a slice of OpenStreetMap for insertion into a SQLite database.

Data Selection¶

I decided to look at Seattle, WA, because I live there. The data was obtained from MapZen as a preselected section of the map covering the Seattle metro area. For my initial investigation and for CSV conversion validation, I studied a procedurally sampled subset of the 2GB .osm file. The full dataset was processed thereafter.

Problems With the Data¶

I used the audit functions in data.py alongside a Sqlite database containing the unmodified data to discovered a number of issues while the data:

- Inconsistent street name abbreviations ("St" versus "Street"; "Fifth Ave" versus "5th Avenue")
- Inconsistent directional suffixes ("Sunnyside Avenue **N**"; "**Northeast** 45th Street")
- Canadian address representation
- Some of the elements in the dataset are incomplete (for example, some tags missing 'uid' field, which will be required by the schema)

Street Name Abbreviations¶

Upon auditing the sample data against some expected street types, many unexpected street types and abbreviations appeared. Some were acceptable ("Alley", "Crescent"); I added these to my list of acceptable strings. Others were not acceptable ("St.", "Stree"). I corrected the latter by regex matching the last word in the street name string, and applying a dictionary mapping.
One important type of unexpected street type stands out in the audit. Directionals seem to be appearing at the ends of lots street names (rather than the street type):

```
 'NW': set(['14th Ave NW',            '22nd Ave NW',            '25th Ave NW',            '26th Ave. NW',
...
```

Let's do something about that.

Directionals¶

According to Wikipedia, "Seattle and King County make systematic use of directionals (such as N for north or NE for northeast) in street names, although residents often omit the directionals when describing addresses in their own neighborhoods."
The convention governing directionals in Seattle is somewhat complicated:

- For "streets" that run East-West, directionals come at the beginning of the name (**E** Pike Street)
- For "avenues" that run North-South, directionals come at the end of the name (Rainier Ave **S**)

This explains the intermittent street name complication. While auditing roadway and address names, I discovered that the directionals were also subject to an inconsistent directional convention ("SW" versus "Southwest").
I resolved the directional problems using regular expressions to catch both common directional reprsentations...

In [ ]:

```
directional_re = re.compile(r'((\s[NESW][EW]?)|(\Dth|\Sst))$', re.IGNORECASE)
```

Now I can iterate on my street name cleaning. When I find a directional at the end of a street name, I'll audit the word just before it. This also gives me an opportunity to make the directional representation consistent using a mapping.

In [ ]:

```
def update_street_name(name, mapping):     """Case Study Code + directionals"""     if directional_re.searc
h(name):         s = street_type_re2.search(name)  # matches 2nd to last word         d = street_type_re.se
arch(name)   # matches last word     else:         s = street_type_re.search(name)         d = None     if
s:         name.replace(s.group(), street_name_mapping[s.group()])     if d:         name.replace(d.group()
, directional_mapping[d.group()])     return name
```

Canadian Data¶

Due to Seattle's proximity to the Canada-United States border, I suspect I will see some foreign tags. I'll start by querying the node tags and way tags, grouped by city. Sure enough, Saanich, B.C. appeared in the top three, grouped by count:

```
Seattle|264507 Kirkland|62231 Saanich|23105
```

I don't have anything against Canadian data, so I'll include all those records in my database. However, I've heard that Canadian postcodes are not uniform with American ones. I don't mind having two postcode systems represented by the same "postcode" label, but I'll need to enforce consistency for both systems. I'll query my database again to see what's out there:

```
sqlite> SELECT tags.value, COUNT(*) as count    ...> FROM (SELECT * FROM nodes_tags UNION ALL SELECT * FROM ways_tags) ta
gs    ...> WHERE tags.type LIKE 'postcode' AND NOT tags.value LIKE '9%'    ...> GROUP BY tags.value    ...> ORDER BY coun
t DESC; value|count V9B1L8|90 Lacey, WA 98503|28 Olympia, 98502|22 V8R 1L6|17 V0S 1N0|13 Olympia, WA 98502|12 ...
```

Looks like I'll also need to clean some data entry hiccups stateside. I can kill two birds with one stone:

In [ ]:

```
postcode_re = re.compile(r'(9\d\d\d\d|\w\d\w ?\d\w\d)')  def update_postcode(postcode):     """Clean Canadi
an and US postcodes, or return None if postcode is invalid."""     m = postcode_re.search(postcode)     if
m:         return m.group(1).replace(' ', '').upper()     return postcode
```

Missing Fields¶

Downstream, when I try to import my converted CSV to SQL, I'll need all my XML elements to be *valid* according to the schema. In my XML sample, I am missing several values that are required by my schemas for CSV output/slqite insertion. I don't believe a few missing 'uid' or 'user' values will impact the usefulness of my data, so I'll just fill them in.

In [ ]:

```
    if element.tag == 'node':         for attribute in node_attr_fields:  # look for the schema's required
fields             if attribute not in element.attrib:                 node_attribs[attribute] = "None"
```

Overview of the Data¶

Size of the Database¶

The filesystem reveals the sizes of the database and the uncompressed, raw OSM data.

```
1.1G seattle.db 1.7G seattle_washington.osm
```

Number of Unique Users¶

I'll count them by unique uid. The schema counts uids against nodes and ways.

```
sqlite> SELECT COUNT(DISTINCT uid)     ...> FROM (SELECT uid FROM nodes UNION ALL SELECT uid FROM ways); 3818
```

Out of curiosity, I ran the following queries to determine the min and max contributions of an individual user:

```
sqlite> SELECT items.uid, items.user, COUNT(*) as count    ...> FROM (SELECT uid, user FROM nodes UNION ALL SELECT uid, u
ser FROM ways) items    ...> GROUP BY uid     ...> ORDER BY count DESC    ...> LIMIT 10; 447903|Glassman|1302039 1382179|
SeattleImport|735295 85280|tylerritchie|655468 147510|woodpeck_fixbot|560815 1408522|Omnific|391375  sqlite> SELECT items
.uid, items.user, COUNT(*) as count    ...> FROM (SELECT uid, user FROM nodes UNION ALL SELECT uid, user FROM ways) items
...> GROUP BY uid    ...> ORDER BY count ASC    ...> LIMIT 5; 346|Tom Chance|1 1852|Sven Anders|1 2231|Deanna Earley|1 43
88|hakan|1 4652|Deelkar|1
```

Clearly, some users are contributing much more than others. This may account for the observation that the most frequently occuring cities are scattered across the geographical area of the dataset--the small group of frequent contributors may not cover the entire area evenly.

Number of Nodes and Ways¶

I should get one row for each node or way in its respective table. I'll just count them all.

```
sqlite> SELECT count(*) FROM ways; 812507
```

```
sqlite> SELECT count(*) FROM nodes; 8165830
```

Those numbers are in line with my expectations. Ways are composed of multiple nodes, so we must have at least as many of the latter.

Coffee¶

Seattle is known all over the world for its coffee culture. It should come as no surprise that the top "cuisine" represented among the nodes in the dataset is "coffee_shop":

```
sqlite> SELECT value, COUNT(*) as count    ...> FROM nodes_tags    ...> WHERE key LIKE 'cuisine'    ...> GROUP BY value
...> ORDER BY count DESC    ...> LIMIT 5; coffee_shop|1318 pizza|826 sandwich|750 mexican|746 burger|598
```

However, I also know from the docs that OpenStreetMap supports a "cafe" amenity that should turn up lots more:

```
sqlite> SELECT value, COUNT(*) as count    ...> FROM nodes_tags    ...> WHERE key LIKE 'amenity'    ...> GROUP BY value
...> ORDER BY count DESC    ...> LIMIT 5; bench|7588 bicycle_parking|6860 restaurant|6086 waste_basket|3162 cafe|3016
```

Ah, there they are.

Benches¶

I was quite surprised to see so many benches, since it never would have occured to me to log a bench in OpenStreetMap. I wonder if this is the result of one person's wild crusade to document every bench in the Pacific Northwest, or if benches are really just that popular aroud here.

```
sqlite> SELECT user, COUNT(*) as count    ...> FROM nodes JOIN nodes_tags ON nodes.id = nodes_tags.id    ...> WHERE nodes
_tags.value LIKE 'bench'    ...> GROUP BY user    ...> ORDER BY count DESC LIMIT 10; alester|2636 Omnific|926 seattlefyi|
450 pluton_od|434 Ballard OpenStreetMap|288 Glassman|262 PhilNi|260 sctrojan79|162 tpotter|132 Skybunny|110
```

Looks like there are a few bench fans out there.

```
sqlite> SELECT COUNT(DISTINCT user)    ...> FROM nodes JOIN nodes_tags on nodes.id = nodes_tags.id    ...> WHERE nodes_ta
gs.value LIKE 'bench'; 135
```

135 of them, to be exact.

Additional Ideas¶

One interesting piece of data I have not yet considered is the "height" tag that appears in many ways--it is used to measure the relative altitude of various nodes and ways in the dataset. It seems like OpenStreetMap supports mapping in 3D. Indeed, the docs suggest the possibility of leveraging OpenStreetMap data to generate 3D-printed, topographic maps of sidewalks and roadways. With broader user participation, and a reliable, automated system for reporting height features, such maps might serve blind people learning the lay of the land. This would be especially useful in a city such as Seattle, where many ways are laid over steep hills and broken terrain.
I started by querying for the IDs of ways having a tag describing height.

```
sqlite> SELECT ways.id    ...> FROM ways JOIN ways_tags ON ways.id = ways_tags.id    ...> GROUP BY ways_tags.id    ...> H
AVING key LIKE 'height'    ...> LIMIT 10; 138257778 169503748 296070147 324652703 357831866 445398675 445398676 445398677
458208106 458208107
```

Unfortunately, the ways associated with these height-having IDs do not appear to be sufficiently detailed to support a project such as the above.

```
sqlite> SELECT * from ways_tags WHERE id='138257778'; 138257778|barrier|fence|regular 138257778|fence_type|split_rail|reg
ular 138257778|height|1.2|regular  sqlite> SELECT * from ways_tags WHERE id='445398675'; 445398675|building|yes|regular 4
45398675|building|brick|material 445398675|height|42|regular
```

These values are actually uniform (the default unit for height in OpenStreetMap is meters), but there just doesn't seems to be enough data available for things like roads and sidewalks:

```
sqlite> SELECT count(*) FROM ways_tags WHERE key='height'; 954
```

The docs cite GPS vertical accuracy as a key limiting factor for this dearth. It seems like robust, OpenStreetMap-driven, 3D-printed relief maps will have to wait for GPS technology to catch up. Alternatively, a software patch might enable opters-in to donate height data collected by their barometric pressure-sensing fitness trackers as they jog around town.

Conclusions¶

OpenStreetMap data is both incomplete and rife with human error. Since we have no measurement for the completeness of this dataset, we cannot easily use it to draw characterizations about life in Seattle. The data would benefit from automated improvements leveraging the power of GPS and other data-generating devices to add more mapping accuracy and granularity, and to enforce consistency and uniformity in the nominal features of the data. With some of the same techniques utilized in this exercise, many more users could begin contributing huge amounts of relatively clean data to the project.