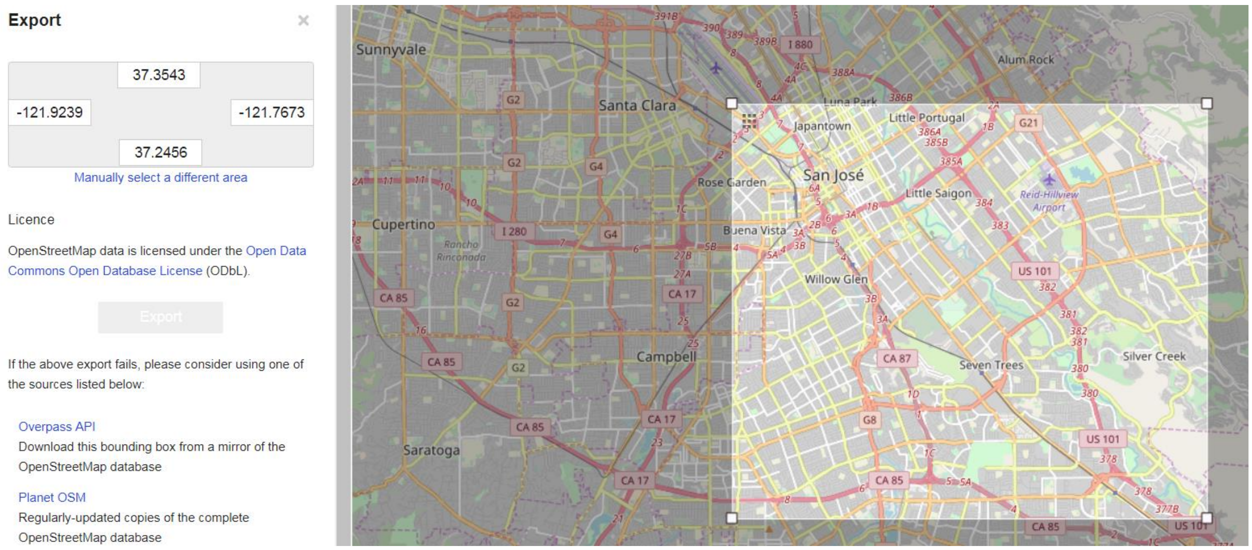
OpenStreetMap Data Wrangling Project

Map Area:
San Jose, US
Link: http://www.openstreetmap.org/export#map=12/37.2999/-121.8456
minlat="37.2456000" minlon="-121.9239000" maxlat="37.3543000" maxlon="-121.7673000"



This is South East of San Jose in California, also South East to the center of Silicone Valley. If keep driving East, there are mountains. If keep driving south, will reach Los Angeles in 6 hours. If keep driving north, will reach San Francisco around 1 hour. This is the neighborhood of my living area, so I am interested to know more about my neighborhood.

Problems Encountered:

1. Inconsistent street type spelling.
Using the test function Python code that I've completed from case study project, I identified various street address is using abbreviation for street type. For example, using Ave rather than Avenue, using Ln rather than Lane.

Note: please see attached py file for the street name audit: "street_name_audit.py"

For example:
```
'Ave': {'Hillsdale Ave', 'Foxworthy Ave', 'Meridian Ave', 'Cherry Ave'},
'Ct': {'Perivale Ct'},
'Dr': {'Linwood Dr'},
'Hwy': {'Monterey Hwy'},
'Julian': {'West Julian'},
'Ln': {'Gaundabert Ln', 'Branham Ln'},
'Rd': {'Silver Creek Valley Rd', 'Quimby Rd'},
```

Therefore, I modify my final shape_element function by applying the update_name function before writing to csv file.

```python
def update_name(name, mapping): # name is a string object, mapping is a Dictionary object

    # Steven: mapping is the St. to Street etc. a dictionary object, so you can call with dictionary key

    better_name = ''
    for item in name.split():
        if item.capitalize() in mapping:  # if the name component is contain in mapping's dictionary "key"
            better_name = ' '.join([better_name, mapping[item.capitalize()]]) # Steven: I add capitalize so in the mapping I
        else:
            better_name = ' '.join([better_name, item.capitalize()]) # if name not in mapping, then just keep the original sp
    return better_name.strip()
```

Apply this update_name() function within shape_element() function prior to writing to csv file, so that all abbreviation are transformed to full spelling in street key.

```python
if dict_tag['key'] == 'street': # Steven: This is to modify the address value (v) to make sure it is in "expectec
    dict_tag['value'] = update_name(dict_tag['value'],mapping)
```

2. Inconsistent postal zip code format:

I explore the data (after writing to 5 csv files extracting from the original osm file) using SQLite3, and check on the postal code using below SQL code:

```sql
select value, count(*) as cnt
from (select * from nodes_tags union all select * from ways_tags)
where key = 'postcode'
group by value
order by cnt desc;
```

```
value       cnt
----------  ----------
95125       79
95113       66
95126       66
95112       55
95110       48
95123       46
95050       40
95118       36
95135       29
95128       25
95136       11
95116       8
95121       7
95122       7
95138       7
95111       6
95124       6
95148       3
94024       1
95110-2007  1
95112-5005  1
95120       1
95127       1
95191       1
CA 95116    1
sqlite>
```

And I notice few zip-code is not formatted same with other using 5-digit format, this could be tricky sometimes if I want to do analysis based on the zip code.
Therefore, I created a Python function update_zipcode(), which utilizing RE library to make sure returning the 5-digit format of zip-code, per below:

```
def update_zipcode(zipcode):
    """
        Update the dict_tag['value'] (string) to the 5-digit format
        Args:
                zipcode (string): to pass in the dict_tag['value'], giving a string type of zip code

        Returns:
                string: return a string of 5-digit zip code
    """
    zip = re.compile(r'9\d{4}')  # to search for 5 digit start with number 9
    match = zip.search(zipcode)
    if match: # if find the correct pattern then use the correct 5-digit pattern
        return match.group(0)
    else: # if not finding 5-digit pattern, then keep as is, so I can capture it in my next round of checking using SQL
        return zipcode
```

And I apply this update_zipcode() function within the shape_element() function before writing to csv files, the code are in the clean_write_to_csv.py file, as follows:

```
if dict_tag['key'] == 'postcode': # Steven: this is to make sure all zip codde are in 5 digit format in 'value'
    dict_tag['value'] = update_zipcode(dict_tag['value']) # Utilize update_zipcode function created above
```

As a result, after this function added and reproduced the csv files, I have the zip codes all in consistent 5-digit format.

```
value       cnt
----------  ----------
95125       79
95113       66
95126       66
95112       56
95110       49
95123       46
95050       40
95118       36
95135       29
95128       25
95136       11
95116       9
95121       7
95122       7
95138       7
95111       6
95124       6
95148       3
94024       1
95120       1
95127       1
95191       1
sqlite>
```

3. Complicated variety of “key” existed in both ways and nodes’ <tag>.

When I start trying to run some statistics to understand more of the dataset, I decide to see how many keys existed in ways and nodes with below SQL code.

```
-- how many unique key type exist in nodes_tags? => 259
select count(distinct(key)) from nodes_tags;

-- how many unique key type exist in ways_tags? => 341
select count(distinct(key)) from ways_tags;

-- How many unique key type in both nodes_tags and ways_tags ?  => 456
select count(distinct(key))
from (select * from nodes_tags union select * from ways_tags);

-- What is the key types existed ? Just to show the top 25 for initial exploration.
select key, count(*)
from (select * from nodes_tags union select * from ways_tags)
group by key
order by count(*) desc
limit 25;
```

```
key         count(*)
----------  ----------
highway     29267
name        14843
building    14025
county      9960
name_base   8897
name_type   8516
cfcc        6987
footway     5039
oneway      4981
surface     4895
reviewed    4739
lanes       4149
service     3849
crossing    2972
amenity     2466
road_marki  1899
source      1614
cycleway    1563
sidewalk    1506
maxspeed    1393
name_direc  1098
bicycle     1044
shop        970
ref         968
website     954
sqlite>
```

This makes me quite confused at first because there are way too many keys (over 400), and the name of the key are too diverse to consider them to be the same nature. For example, I know what ‘highway’ and ‘amenity’ means, but there are also ‘name’, ‘cfcc’, ‘source’, and ‘reviewed’ etc which doesn’t sound like a physical object. Therefore, I keep searching on osm website for different naming convention, and this web page “Map Features” has share the standard
=> http://wiki.openstreetmap.org/wiki/Map_Features

It turns out that “key” can store not only the physical object like Building, Amenity or Shop, but can also contain additional properties like “disused”, “address” or annotation like “source” etc. Also, it can add “namespace” to the key, which is a prefix, suffix or inflix to add to a key (text prior to a colon) intended

to group closely related keys. For example, I see quite a few TIGER:xxx as the key, upon later search it is the data produced by the US Census Bureau.
=> http://wiki.openstreetmap.org/wiki/TIGER

The map features standard help me to understand more how keys and values are structured, but it is still quite large mapping to read through, so I am focusing my following data overview on few keys that is easily understood.

Data Overview and Additional Exploration:

1. File size:
map (San Jose).xml ... 62 MB
project.db .... 39 MB
nodes.csv .... 22 MB
nodes_tags.csv .... 0.8 MB
ways.csv .... 2.6 MB
ways_tags.csv .... 5 MB
ways_nodes.cv .... 7.5 MB

2. Number of unique users => 562
--SQL query:
```
select count(distinct(u.uid))
from(select uid from nodes UNION SELECT uid FROM ways) u;
```

3. Number of nodes => 264090
--SQL query:
```
select count(*) from nodes;
```

4. Number of ways => 43362
--SQL query:
```
select count(*) from ways;
```

5. Explore “value” with “amenity” key in both ways and nodes:
-- Count of total number of unique amenity's value: => 104
```
select count(distinct(value))
from (select * from nodes_tags union select * from ways_tags)
where key = 'amenity';
```

-- Count of total number of amenity's value: => 2466
```
select count(value)
from (select * from nodes_tags union all select * from ways_tags)
where key = 'amenity';
```

-- What is the top 20 count of amenity’s each value ? => screenshot below, parking and restaurant have most count.
```
select value, count(*)
```

```
from (select * from nodes_tags union all select * from ways_tags)
where key = 'amenity'
group by value
order by count(*) desc
limit 20;
```

value	count(*)
parking	506
restaurant	380
fast_food	217
school	177
place_of_w	117
cafe	85
fuel	85
bench	79
bank	75
bicycle_pa	53
toilets	47
bar	42
dentist	39
post_box	38
pharmacy	27
doctors	21
bicycle_re	20
vending_ma	18
atm	17
fire_stati	17

6. There is <nd> tag under <Way> tab which link to specific <node> id. I am curious to know how many nodes are referenced in each way, show the top 10.

```
select id, count(node_id) as cnt
from ways_nodes
group by id
order by cnt desc
limit 10;
```

id	cnt
50372807	610
38550004	501
38549940	498
38550020	495
38549977	494
49639162	493
528935863	415
33174055	249
547196323	237
50372806	231

7. Following the above, I am curious to see what are the key of the ways that are reference to the highest number of nodes. The tricky thing is that each way can have multiple tags/keys which can mean variety of object or property, so the list drag pretty long. But through my own judgment, the top 1 is a lake, the second one is San Jose City boundary.

```
select *
from (select id, count(node_id) as cnt
from ways_nodes
group by id) t1
inner join (select * from ways_tags) as t2
```

```
on t1.id = t2.id
order by cnt desc
limit 10;
```

id	cnt	id	key	value	type
50372807	610	50372807	name	Lake Cunningham	regular
50372807	610	50372807	natural	water	regular
50372807	610	50372807	scvwd:SHAP	2164013.1542000	regular
50372807	610	50372807	scvwd:WB_T	Other	regular
38550004	501	38550004	admin_leve	8	regular
38550004	501	38550004	border_typ	city	regular
38550004	501	38550004	boundary	administrative	regular
38550004	501	38550004	name	San Jose	regular
38550004	501	38550004	place	city	regular
38550004	501	38550004	source	TIGER/Liner 200	regular

Improvement possibility:

As mentioned in the “problem encountered” section, the large variety of the key gives me difficulty if I want to do more large scale analysis or draw statistic insight on specific key types (such as only the primary features, or only on address, or only on special properties). For example, my SQL query in #7 above have printed all tags and keys associated with each way id, and I have to do another judgment to determine which key is the primary feature to tell me what it is.
I have to refer back to the map feature webpage (http://wiki.openstreetmap.org/wiki/Map_Features) to search before I make the judgment that is quite time consuming and cannot scale for larger analysis.

Therefore, I would like to introduce another grouping within tag, to give me another layer of grouping, and group the tag by “primary feature”, “address”, “annotation”, “name”, “properties”, “References”, “Restriction” and “Namespace”. To achieve it, I need to design the Python code to create a new field following the mapping. In fact, in the case study, the shape_element code in Python has extract the text prior to colon within the key (eg. k="addr:street:name", to extract the “addr” out), and enter it into the “type” column in csv. It makes it easier to identify the address type of tag, and query only on address key for related analysis.

Another possibility, since there are already standard naming convention for “key” and “value” in tag available on OSM web, I would appreciate if OSM can create a new attribute to store the “type” of the key, by mapping the user input of the key with its standard mapping on the web, and it can be done at the time of the user input, so it can request user to confirm the correctness of the match of the key to the type.

The down side of such additional “type” layer is it requires additional maintenance of the mapping from either or both OSM system and from the user. This can be time-consuming.