CODE DOCUMENTATION

This document provides a general documentation for the code used for the completion of this project. A description of the code contained in each file is given below.

1) Change_street_address.py:

This file contains two major functions named as 'modify street address' and 'create full address'. A brief description of what these functions do is below:

modify_street_address: Called from main.py. Takes in the address string from the "addr:street" attribute of each node element, as well as a regular expressions to find appropriate end names for streets and length of the address string, as arguments from the main function. Returns a street address sliced from the address argument, if proper end names suggesting street types are found. Also returns a flag variable is_full_address, if it is found that the address string either contains no end-word, or contains more components than just the street address.

The regular expressions are created in the main function. The regular expression reg_ex is used to search for suitable 'end-words' in the address string. If an appropriate end-word is found, the address string is sliced from the beginning until the last occurrence of the end word. The function then checks if the original address string is longer in length than the newly sliced string (with the help of the regular expression re_len), which would be the case if the address string contained not only the street name, but rather a more detailed description of the address. (Sometimes the entire address is contained in the addr:street attribute and not just the street address).

If it is found that the address argument contained more information than just the street name, a flag variable (is_full_address) is activated. The return value for the function is the flag variable and the street address sliced from the original string. The activated flag value will in turn trigger creation of a new field "addr:fulll", which is used in openstreetmaps to contain unstructured addresses. The full address is created using a separate function, described below.

create_full_address: Called from main.py, if the return value is_full_address from the above function is 1. Takes in the xml element (node/tag) as argument, which will contain all tags such as name, houseno, addr:housename, streety, city, state and postcode. The values of all these tags are combined to generate the full address, and then returned.

2) Check_phone_numbers.py

Identifies all the unique phone numbers in the file, so as to identify potential errors in them

3) Clean_phone_numbers.py:

This file contains the function "check_and_clean_number()". This function takes in the number string from the osm file as argument. Having explored the possible formats of numbers in the file using the script in check_phone_numbers.py, the following strategy was adopted to clean the phone numbers.

1) Strip of the observed special characters from the phone number, which are {-, +, (,)}, in addition to white spaces and replace them with empty strings ""

2) Replace ',' with ';' and split the string at ';'. This is because both ';' and ',' are used to separate individual phone numbers. Splitting at these special characters enables to separate the string into individual numbers

3) For each number,

- If the number begins with '910', this indicates a wrongly specified format (the zero should not be present after the country code if the same is used. This is applicable to both landline and cellphones). The new number will have a '+' attached to it, the 0 in 910 shall be ignored The rest of the number will be attached after +91.
- If the number begins with 0, replace the zero with +91. This is applicable to both landline and cell phone numbers and and would yield the standard format for both of them
- If the number does not begin with '910', but begins with '91', attach a + to the beginning of the number.

For numbers that do not fall into the above three categories,

- If the length of number is 8, then affix '+9180' to the beginning of the number. Note that the landline numbers in Bangalore are of 8 digits in length, excluding the area and country code. Checking for 8 digit length for the phone numbers yields the landline numbers from the list
- If the length of number is 10, then affix '+91' to the beginning of the number. These are for cellphone numbers.
- Those numbers that do not fall into the above two categories are left the way they are.

Please not that this approach only corrects errors related to formatting, and wrong specification of country and area codes. If there are other errors in the actual phone number itself (such as missing digits, extra digits etc.,) they are not corrected.

An exception to the general set of rules mentioned above are 'hotline' numbers beginning with 1800. These numbers are sometimes wrongly specified starting with +91 in the osm file. Hence they are identified using regular expressions and corrected separately, to remain in the format 1800-xxxx

4) Check_pincode.py:

This script was used to identify possible errors in pincode. The approach used in this regard is rather simple, in that the only criterion employed to look for the correctness of pincodes is to check if their length is 6 digits. The code identifies all those elements with wrong pincodes, and stores their element ID as well as the wrong pincode.

5**) Main.py:**

The main function in this file is *process_map()*, which takes in the osm file name and a flag to check if schema validation is necessary or not. This function in turn calls the *get_element* function which parses the osm file, and returns the elements. These elements are then processed by the shape_element function which in turn calls the functions *modify_street_address*, *check_and_clean_numbers* and *create_full_address* mentioned above to correct the street address, modify phone numbers and create a full address tag if required. The *shape_element* function also transforms and returns the data in a proper format, which are then written as csv files by the *process_map()* function. The shape element function has been created according to the instructions provided by udacity, so as to write csv files in a format that would conform to a particular schema of the final database, into which the data will be entered. The details of the schema is mentioned below.

The tables in the database are created according to the following schema. There shall be
i) A nodes table, which contains the attributes id, lat, lon, user, uid, version, changeset and timestamp
ii) A node tags table with attributes id, key, value and type.

iii) A ways table, with attributes id, user, uid, version, changeset and timestamp
iv) A way tags table with attributes id, key value and type.
v) A way nodes table with attributes id, node_id and position.

These tables form the database, and in order to insert data as per this schema, we create csv files containing the data in this format, after transforming the data using the *shape_element* function. More details of the shape element function has been provided in the appendix.

6) Create_database.py:

Creates the relational tables and populates them with the data from csv files. The code is rather simple, and the accompanying comments should provide sufficient explanation of the same.

APPENDIX

[A brief description of the idea behind shape element function is given below. The shape element function has been designed as per the instructions from Udacity and the hence text below presents the same set of instructions as provided in the Udacity webpage related to this project. This has been included in this report for the sake of completeness and clarity, so as a person viewing this project will be able to understand the work in its entirety without referring to Udacity]

The Shape Element Function function takes as input an iterparse Element object and returns a dictionary.

I. If the element top level tag is "node":

The dictionary returned has the format {"node": .., "node_tags": ...}

The "node" field holds a dictionary of the following top level node attributes:
- id
- user
- uid
- version
- lat
- lon
- timestamp
- changeset

All other attributes are ignored

The "node_tags" field holds a list of dictionaries, one per secondary tag. Secondary tags are child tags of node which have the tag name/type: "tag". Each dictionary has the following fields from the secondary tag attributes:
- id: the top level node id attribute value
- key: the full tag "k" attribute value if no colon is present or the characters after the colon if one is.
- value: the tag "v" attribute value
- type: either the characters before the colon in the tag "k" value or "regular" if a colon is not present.

Additionally,

- if the tag "k" value contains problematic characters, the tag is ignored

- if the tag "k" value contains a ":" the characters before the ":" is set as the tag type
  and characters after the ":" is set as the tag key
- if there are additional ":" in the "k" value they are ignored and kept as part of
  the tag key. For example:

  <tag k="addr:street:name" v="Lincoln"/>
  is turned into
  {'id': 12345, 'key': 'street:name', 'value': 'Lincoln', 'type': 'addr'}

- If a node has no secondary tags then the "node_tags" field contains an empty list.

The final return value for a "node" element looks something like:

```
{'node': {'id': 757860928,
          'user': 'uboot',
          'uid': 26299,
       'version': '2',
          'lat': 41.9747374,
          'lon': -87.6920102,
          'timestamp': '2010-07-22T16:16:51Z',
      'changeset': 5288876},
 'node_tags': [{'id': 757860928,
                'key': 'amenity',
                'value': 'fast_food',
                'type': 'regular'},
               {'id': 757860928,
                'key': 'cuisine',
                'value': 'sausage',
                'type': 'regular'},
               {'id': 757860928,
                'key': 'name',
                'value': "Shelly's Tasty Freeze",
                'type': 'regular'}]}
```

II. If the element top level tag is "way":
The dictionary has the format {"way": ..., "way_tags": ..., "way_nodes": ...}

The "way" field holds a dictionary of the following top level way attributes:
- id
- user
- uid
- version
- timestamp
- changeset

All other attributes are ignored

The "way_tags" field holds a list of dictionaries, following the exact same rules as
for "node_tags".

Additionally, the dictionary has a field "way_nodes". "way_nodes" holds a list of
dictionaries, one for each nd child tag. Each dictionary has the fields:

- id: the top level element (way) id
- node_id: the ref attribute value of the nd tag
- position: the index starting at 0 of the nd tag i.e. what order the nd tag appears within the way element

The final return value for a "way" element looks something like:


```
{'way': {'id': 209809850,
        'user': 'chicago-buildings',
        'uid': 674454,
        'version': '1',
        'timestamp': '2013-03-13T15:58:04Z',
        'changeset': 15353317},
 'way_nodes': [{'id': 209809850, 'node_id': 2199822281, 'position': 0},
               {'id': 209809850, 'node_id': 2199822390, 'position': 1},
               {'id': 209809850, 'node_id': 2199822392, 'position': 2},
               {'id': 209809850, 'node_id': 2199822369, 'position': 3},
               {'id': 209809850, 'node_id': 2199822370, 'position': 4},
               {'id': 209809850, 'node_id': 2199822284, 'position': 5},
               {'id': 209809850, 'node_id': 2199822281, 'position': 6}],
 'way_tags': [{'id': 209809850,
               'key': 'housenumber',
               'type': 'addr',
               'value': '1412'},
              {'id': 209809850,
               'key': 'street',
               'type': 'addr',
               'value': 'West Lexington St.'},
              {'id': 209809850,
               'key': 'street:name',
               'type': 'addr',
               'value': 'Lexington'},
              {'id': '209809850',
               'key': 'street:prefix',
               'type': 'addr',
               'value': 'West'},
              {'id': 209809850,
               'key': 'street:type',
               'type': 'addr',
               'value': 'Street'},
              {'id': 209809850,
               'key': 'building',
               'type': 'regular',
               'value': 'yes'},
              {'id': 209809850,
               'key': 'levels',
               'type': 'building',
               'value': '1'},
              {'id': 209809850,
               'key': 'building_id',
               'type': 'chicago',
               'value': '366409'}]}
```