UDACITY

‹ Back to Data Analyst Nanodegree

Wrangle OpenStreetMap Data

REVIEW

CODE REVIEW 4

HISTORY

data.py 3

```python
import csv
import codecs
import pprint
import re
import xml.etree.cElementTree as ET
import codecs
import datetime
from collections import defaultdict


# Code to more efficiently parse the OSM xml and return a list of XML elements
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()

# Dictionary of terms to replace, i.e. "bbq" --> "barbecue"
cuisine_dict = {
    "bbq": "barbecue",
```

AWESOME

Good work finding the inconsistent/incorrect issues with the cuisine names.

```
    "bar&grill": "grill",
    "donuts": "donut",
    "bagels": "bagel",
    "steak": "steak_house",
    "burgers": "burger",
    "salads": "salad",
    "sandwiches": "sandwich",
    "sandwhiches": "sandwich",
    "sandwhich": "sandwich",
    "hot_dogs": "hot_dog",
    "chicago_dogs": "hot_dog",
    "coffee": "coffee_shop",
    "texmex": "tex_mex",
    "roast_beel": "roast_beef",
    "puertorican": "puerto_rican",
    "jewish_deli": "jewish",
    "fish_tacos": "tacos",
    "juice_bar": "juice",
    "mexcian_food": "mexican",
    "mexican_food": "mexican",
    "wrap": "wraps",
    "french_fries": "fries",
    "drive_thru_coffee": "coffee_shop",
    "healthy_cuisine_-_greek": "greek",
    "tailand": "thai",
    "thailand": "thai"
}

# These are cuisine terms we will throw out and not allow, based on the large-ish sam
# NOTE the some terms - "vegan", "vegetarian", "drive_through" - should be included i
#   besides cuisine tags, and the code detects them and adds the appropriate sub-tags
disallowed = ["", "food", "vegan", "vegetarian", "vegetarian_and_vegan", "mon", "ret"
              "lunch", "diner", "dinner", "bar", "pub", "drive_through"]

# strips the string of whitespace and leading underscore, and converts to lower-case
# i.e. "_pizza" --> "pizza";  "Fries " --> "fries"
def str_space_undersc_lower(string):
    if string == "" or string == None:
        return ""
    string = string.strip()[1:] if string.strip()[0] == '_' else string.strip()
    return string.lower()

# Replaces spaces between cuisine terms with an underscore
# i.e. "coffee shop" --> "coffee_shop"
def repl_space(string):
    return "_".join(string.split())

# splits strings on comma and/or semicolon
def spl(string):
    return re.split("[,;]", string)

# Replace string with its substitute in the cuisine dictionary, if it is in the dict
# If its not in dict and its not in the disallowed list and if it is 3+ characters lo
#   then use it as is.
def dict_lookup(string):
    if string in cuisine_dict:
        return cuisine_dict[string]
    elif string in disallowed:
        return None
    elif len(string) < 3:
        return None
    else:
        return string

# check if have text indicating they are using text for a drive through for a cuisine
def has_drive_through_text(text):
    lower_text = text.lower()
    return ("drive_through" in lower_text) or ("drivethrough" in lower_text)

# check if have text for vegan in cuisine text
def has_vegan_text(text):
    lower_text = text.lower()
    return "vegan" in lower_text

# check if have text for vegetarian in cuisine text
def has_vegetarian_text(text):
    lower_text = text.lower()
    return "vegetarian" in lower_text

# call dict_lookup after cleaning with str_space_undersc_lower and then repl_space
def process(string):
    return dict_lookup(repl_space(str_space_undersc_lower(string)))

# clean up each element after splitting them, and then rejoin
def shape_cuisine_element(cuisine_str):
    return ";".join(filter(None, map(process, re.split('[,;]', cuisine_str))))


node_attribs = ['id', 'user', 'uid', 'version', 'lat', 'lon', 'timestamp', 'changeset
node_tag_keys = ['id', 'key', 'value', 'type']
way_attribs = ['changeset', 'id', 'timestamp', 'uid', 'user', 'version']

OSM_PATH = "phoenix.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 sc
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']

def get_valid_type_key(thestring):
    if PROBLEMCHARS.search(thestring):
        return False, None, None
    elif ':' in thestring:
        return True, thestring[: thestring.index(':')], thestring[thestring.index(':'
    else:
        return True, "regular", thestring

# Populate the node attribute dictionary, using the "lat", "lon", "uid", and "changes
# Note: we assume valid string attributes if not lat, lon, uid or changeset
```

```
def pop_attrib(tag):
    thedict = {}
    for attrib in node_attribs:
        if attrib == 'lat' or attrib == 'lon':
            thedict[attrib] = float(tag.get(attrib))
        elif attrib == 'id' or attrib == 'uid' or attrib == 'changeset':
            thedict[attrib] = int(tag.get(attrib))
        else:
            thedict[attrib] = tag.get(attrib)
    return thedict

# populate the way attributes dictionary
def get_way_attribs(tag):
    attrib_dict = {}
    for attrib in way_attribs:
        if attrib == 'changeset' or attrib == 'id' or attrib == 'uid':
            attrib_dict[attrib] = int(tag.get(attrib))
        else:
           attrib_dict[attrib] = tag.get(attrib)
    return attrib_dict

# populate the way nodes dictionary
def pop_way_nodes(tag):
```

SUGGESTION

Great job commenting for code that is not intuitively readable. Although it is not required for this project, particularly for arguments. Here is the resource
https://www.python.org/dev/peps/pep-0257/

```
    nodes = []
    tagid = int(tag.get('id'))
    nd_tags = tag.findall('nd')
    for index, nd_tag in enumerate(nd_tags):
        newdict = {}
        newdict['id'] = tagid
        newdict['node_id'] = int(nd_tag.get('ref'))
        newdict['position'] = index
        nodes.append(newdict)
    return nodes

# Populate the drive_through dictionary, using the passed in tag id and tag type
# Note this is used when we find that the cuisine tag is indicating this is a drive t
def pop_drive_through(tag_id, tag_type):
    return {
        "id": tag_id, "key": "drive_through", "value": "yes", "type": tag_type
    }

# Populate the diet dictionary, using the passed in information for tag id, tag type,
#  that this is indicating a vegan or a veretagrian restaurant (or both)
# Note this is used when we find the cuisine tag is indicating this is a vegan and/or
def pop_diet(tag_id, tag_type, has_vegan, has_vegetarian):

    if has_vegan and has_vegetarian:
        text = "vegan;vegetarian"
    elif has_vegan:
        text = "vegan"
    elif has_vegetarian:
        text = "vegetarian"

    if has_vegan or has_vegetarian:
        return { "id": tag_id, "key": "diet", "value": text, "type": tag_type }
    else:
        return None

# Populate the tags dictionary based on the keys and values in the tag's subtags.
# If this is a cuisine tag, we do some extra checking and cleaning up.
# In particular, if the cuisine tag has indications that this is a vegan or vegetarian
# restaurant, then we add a dictionary for a "diet" subtag.  If it indicates a drive t
# restaurant, then we add a dictionary for a "drive_through" subtag.
# We also do the cuisine tag cleanups based on formatting (stripping underscores, whit
#  and a dictionary lookup to replace terms.
def pop_tags(tag):
    thelist = []
    for subtag in tag.iter("tag"):
        newdict = {}
        valid, typ, key = get_valid_type_key(subtag.get('k'))
        val = subtag.get('v')
        if valid and val != None and val != "":

            cleaned_val = get_subtag_value(subtag)
            if cleaned_val != None and cleaned_val != '':
                newdict['id'] = int(tag.get('id'))
                newdict['key'] = key
                newdict['value'] = cleaned_val
                newdict['type'] = typ
                thelist.append(newdict)

            if key == "cuisine":
                need_drive_through_dict = has_drive_through_text(val)
                need_dict_for_vegan = has_vegan_text(val)
                need_dict_for_veggie = has_vegetarian_text(val)

                if need_drive_through_dict and len(tag.findall("./tag/[@k='drive_throu
                    thelist.append(pop_drive_through(int(tag.get('id')), typ))
                if (need_dict_for_vegan or need_dict_for_veggie) and len(tag.findall(
                    thelist.append(pop_diet(int(tag.get('id')), typ, need_dict_for_veg
    return thelist

def get_subtag_value(subtag):
    val = subtag.get('v')
    if subtag.get('k') == "cuisine":
        return shape_cuisine_element(val)
    else:
        return val

# Make the dictionary for way tags
def make_way_dict(tag):
    return { "way": get_way_attribs(tag), 'way_nodes': pop_way_nodes(tag), 'way_tags'

# Make the dictionary for node tags
def make_node_dict(tag):
    return { "node": pop_attrib(tag), 'node_tags': pop_tags(tag) }

# Clean up the element, by creating a dictionary for a way or node with cleaned up val
def shape_element(element, node_attr_fields=NODE_FIELDS, way_attr_fields=WAY_FIELDS,
                  problem_chars=PROBLEMCHARS, default_tag_type='regular'):

    if element.tag == 'node':
        return make_node_dict(element)
```

AWESOME

Great job calling all the functions within the `shape_element` function to update cleaned data back to the m

```
    elif element.tag == 'way':
        return make_way_dict(element)

# Parse the OSM xml and get a dictionary for every element, then write the dictionary
def process_map(file_in, validate):
    #Iteratively process each XML element and write to csv(s)

    with open(NODES_PATH, 'w', newline="", encoding='utf-8') as nodes_file, open(NODE
        open(WAYS_PATH, 'w', newline="", encoding='utf-8') as ways_file, open(WAY_NOD
        open(WAY_TAGS_PATH, 'w', newline="", encoding='utf-8') as way_tags_file:

        nodes_writer = csv.DictWriter(nodes_file, fieldnames=NODE_FIELDS, delimiter='
        node_tags_writer = csv.DictWriter(nodes_tags_file, fieldnames=NODE_TAGS_FIELD
        ways_writer = csv.DictWriter(ways_file, fieldnames=WAY_FIELDS, delimiter='|')
        way_nodes_writer = csv.DictWriter(way_nodes_file, fieldnames=WAY_NODES_FIELDS
        way_tags_writer = csv.DictWriter(way_tags_file, fieldnames=WAY_TAGS_FIELDS, d

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

        for element in get_element(file_in, tags=('node', 'way')):
            el = shape_element(element)
            if el:
                #if validate:
                #    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'])
```

README.txt 1

references_urls.txt

overpass_api_usage.txt

audit.py

RETURN TO PATH

Student FAQ