Data Science Nano Degree –

Project 1 –

Data Wrangling with Mongo DB

Code Listings

# Table of Contents

Table of Contents

[Table of Contents 1](#_Toc417684448)

[Addendum – Code Listing 2](#_Toc417684449)

[1. Data Audit 2](#_Toc417684450)

[2. Data Cleaning 7](#_Toc417684451)

[3. MongoDB Queries 13](#_Toc417684452)

# Addendum – Code Listing

## Data Audit

|  |
| --- |
| # -\*- coding: utf-8 -\*-  """  Created on Sun Mar 08 11:59:57 2015  @author: joe  """  #!/usr/bin/env python  # -\*- coding: utf-8 -\*-  """  Your task is to use the iterative parsing to process the map file and  find out not only what tags are there, but also how many, to get the  feeling on how much of which data you can expect to have in the map.  The output should be a dictionary with the tag name as the key  and number of times this tag can be encountered in the map as value.  Note that your code will be tested with a different data file than the 'example.osm'  """  import xml.etree.ElementTree as ET  import pprint  import re  #import os  MapDirPath = "C:/Users/joe/My Documents/Classes/DataScience/Project 2/" # Location for local files  #MapDirPath = "./" # Use current directory for online env  #  # Available map files:  #  #MapFile = "example.osm" # Test file = from online env  #MapFile = "Map-Bar-Harbor.osm" # Small map - downtown Bar Harbor  #MapFile = "Map-Mount-Desert-Island.osm" # Medium size map - MDI only  #MapFile = "Map-Bangor-to-Acadia.osm" # Full map for analysis  MapFile = "Map-Bangor-to-Acadia.osm" # Map file in use  #  # Zipcodesis manually generated from a map.  # contains a possibly incomplete list is zip codes  # for the bangor to Mount Desert region  # Accurate ZIP code DBs are available, but expensive.  # USPS provides an address-to-ZIP translation that could be used, but it  # would be very slow to look up each location individually  Zipcodes = ['04401', '04411', '04412', '04416', '04420',  '04421', '04428', '04429', '04431', '04438',  '04435', '04438', '04444', '04472', '04473',  '04474', '04476', '04496', '04605', '04607',  '04609', '04612', '04614', '04616', '04617',  '04640', '04679', '04660', '04662', '04664',  '04672', '04674', '04675', '04676', '04677',  '04679', '04684', '04693', '04848' ]  street\_type\_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)  def street\_type(street\_name) :  '''  Function: street\_type(street\_name)  Inputs: street\_name - name of stree, minus house number, e.g. 'Cottage Avenue'  Returns: Street type (e.g. Avenue, Ave, St, ...    Assumes that the street type is the string from the character  after the last white space to the end of the string)  '''  m = street\_type\_re.search(street\_name)  # DBG print '>>>>>StreetName: ', street\_name, " >>> Type: ", m  if m :  st = m.group()  else:  st = None  return st    def print\_in\_columns (values, ncol) :  '''  Function: print\_in\_columns  Inputs: values = list to be printed  ncol = number of columns printed in each row  Provides a way to compact the number of print lines to display lists  by printing several consequitive items on a single line.  Example: print\_in\_columns(arange(0, 15, 4))  ==>[  0, 1, 2, 3,  4, 5, 6, 7,  8, 9, 10, 11,  12, 13, 14 ]  '''  print '[',  nval = len(values)  for i in range(0, nval) :  if i == nval-1 :  str = "\t{} ]".format(values[i])  else :  str = "\t{},".format(values[i])  if (i+1) % ncol == 0 :  print str  else :  print str,    def print\_dict\_in\_columns (values, ncol) :  '''  Function: print\_in\_columns  Inputs: values = dictionary to be printed  ncol = number of columns printed in each row  Provides a way to compact the number of print lines to display lists  by printing several consequitive items on a single line.  output is sorted by key  Really only useful for single level dictionary, and where  key&value are short  Example:  dt = {'a': 1, 'c': 3, 'b': 2, 'd': 4, 'e': 5}  print\_dict\_in\_columns(dt, 3)  ==>{  'a': 1, 'b': 2, 'c': 3,  'd': 4, 'e':5 }  '''  print '{',  nval = len(values)  keylist = (values.keys())  keylist.sort()  i = -1  for k in keylist :  i += 1  if i == nval-1 :  str = "\t'{}': {} {}".format(k, values[k], '}')  else :  str = "\t'{}': {},".format(k, values[k])  if (i+1) % ncol == 0 :  print str  else :  print str,  def main():  '''    Function: main()  Inputs: None  returns: None    Iterate through the OSM file and perform a series of checks  and summaries on each line.  This is the main loop, and checks/summaries are  performed in parallel to avoid having to read the file  multiple times.  Run Series of analysis and error checking tasks  1) Determine coverage area, total number of XML elements,  number of elements by type,  2) Determine list of unique keyword attributes ('k') in <tag ...\>  3) Determine unique list of values for specified keywords.  4) Determine list of unique street types (last token  of addr:street)      '''  # Initialize counters  tagCount = {} # Counts of XML element types in file  keyCount = {} # Count of XML tag types  valueCount = {"addr:postcode" : {},  "tiger:zip\_left\_1" : {},  "tiger:zip\_right\_1" : {},  "tiger:county" : {},  "addr:country": {},  "addr:state" : {},  "addr:city" : {},  "amenity" : {},  "tourism" : {},  "cuisine" : {},  "religion" : {},  "is\_in" : {},  "is\_in:country" : {},  "gnis:County" : {},  "gnis:ST\_alpha" : {}  }  streetTypes = {}  # Full file name is dirpath+file  filename = MapDirPath + MapFile  #  # Loop through elements in the OSM file  #  total = 0  for \_, element in ET.iterparse(filename):  #DBG if (total > 200) :  #DBG break  total = total + 1  #DBG print event, elem.tag  #  # Print bounding box (lat & lon)  #  if element.tag == 'bounds' :  att = element.attrib  print "Boundary of selected map:\n \  Latitude from {} to {} \n \  Longitude from {} to {}".format(att['minlat'],  att['maxlat'],  att['minlon'],  att['maxlon'])  # Gather counts for XML tags  if tagCount.get(element.tag) :  tagCount[element.tag] += 1  else :  tagCount[element.tag] = 1  #DBG print "dt:\n", dt    # Gather counts for unique keywords in <TAG> elements  kword = element.attrib.get('k')  if kword :  if keyCount.get(kword) :  keyCount[kword] += 1  else :  keyCount[kword] = 1  # DBG print element.tag, element.attrib  # Gather counts of values for interesting <TAG> keywords (as listed in valueCounts)  #  kword = element.attrib.get('k')  val = element.attrib.get('v')  if kword and kword in valueCount.keys():  dct = valueCount[kword]  if dct.get(val) :  dct[val] += 1  else:  dct[val] = 1  valueCount[kword] = dct    # Gather list of unique Street types (Rd, St, Drive, ...)      kword = element.attrib.get('k')  if kword and kword == 'addr:street':  val = street\_type(element.attrib.get('v'))  if val :  if streetTypes.get(val) :  streetTypes[val] += 1  else:  streetTypes[val] = 1  pass  print "\nTotal number of tags: \n", total  print('\nTraverse file: Scanned {} elements'.format(total))  print('\nCount of unique XML tags:')  pprint.pprint(tagCount)  print('\nCount of unique keywords:\n\t{')  print\_dict\_in\_columns(keyCount, 2)  print('\nCount of unique values for selected keywords:')  pprint.pprint(valueCount)  print('\nCount of unique values for street types:')  pprint.pprint(streetTypes)  ##  # Additional analysys:  #  # Look for postal codes that are correct for this locality  # Known good ZIP codes are in 'Zipcodes'  # Codes to be checked are in valueCount['addr:postcode'],  # valueCount['tiger:zip\_left\_1'], and  # valueCount[tiger:zip\_right\_1]  print '\nList of bad/suspicious ZIP codes:'  badzips = []  for z in valueCount['addr:postcode'].keys()+\  valueCount['tiger:zip\_left\_1'].keys()+\  valueCount['tiger:zip\_right\_1'].keys() :  if z not in Zipcodes :  badzips.append(z)  print badzips  #  # Find list of all street types (last whitespace separated token in addr:street)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

## Data Cleaning

|  |
| --- |
| #!/usr/bin/env python  # -\*- coding: utf-8 -\*-  """  Created on Sun Mar 15 13:46:18 2015  @author: joe  """  import xml.etree.ElementTree as ET  import pprint  import re  import codecs  import json  """  Your task is to wrangle the data and transform the shape of the data  into the model we mentioned earlier. The output should be a list of dictionaries  that look like this:  {  "id": "2406124091",  "type: "node",  "visible":"true",  "created": {  "version":"2",  "changeset":"17206049",  "timestamp":"2013-08-03T16:43:42Z",  "user":"linuxUser16",  "uid":"1219059"  },  "pos": [41.9757030, -87.6921867],  "address": {  "housenumber": "5157",  "postcode": "60625",  "street": "North Lincoln Ave"  },  "amenity": "restaurant",  "cuisine": "mexican",  "name": "La Cabana De Don Luis",  "phone": "1 (773)-271-5176"  }  You have to complete the function 'shape\_element'.  We have provided a function that will parse the map file, and call the function with the element  as an argument. You should return a dictionary, containing the shaped data for that element.  We have also provided a way to save the data in a file, so that you could use  mongoimport later on to import the shaped data into MongoDB.  Note that in this exercise we do not use the 'update street name' procedures  you worked on in the previous exercise. If you are using this code in your final  project, you are strongly encouraged to use the code from previous exercise to  update the street names before you save them to JSON.  In particular the following things should be done:  - you should process only 2 types of top level tags: "node" and "way"  - all attributes of "node" and "way" should be turned into regular key/value pairs, except:  - attributes in the CREATED array should be added under a key "created"  - attributes for latitude and longitude should be added to a "pos" array,  for use in geospacial indexing. Make sure the values inside "pos" array are floats  and not strings.  - if second level tag "k" value contains problematic characters, it should be ignored  - if second level tag "k" value starts with "addr:", it should be added to a dictionary "address"  - if second level tag "k" value does not start with "addr:", but contains ":", you can process it  same as any other tag.  - if there is a second ":" that separates the type/direction of a street,  the tag should be ignored, for example:  <tag k="addr:housenumber" v="5158"/>  <tag k="addr:street" v="North Lincoln Avenue"/>  <tag k="addr:street:name" v="Lincoln"/>  <tag k="addr:street:prefix" v="North"/>  <tag k="addr:street:type" v="Avenue"/>  <tag k="amenity" v="pharmacy"/>  should be turned into:  {...  "address": {  "housenumber": 5158,  "street": "North Lincoln Avenue"  }  "amenity": "pharmacy",  ...  }  - for "way" specifically:  <nd ref="305896090"/>  <nd ref="1719825889"/>  should be turned into  "node\_refs": ["305896090", "1719825889"]  """  #  # file to be read  # Variants for course IDE, local (spyder), and local (Py Notebook)  MapDirPath = "C:/Users/joe/My Documents/Classes/DataScience/Project 2/" # Location for local files  #MapDirPath = "./" # Use current directory for online env  #  # Available map files:  #  #MapFile = "example.osm" # Test file = from online env  #MapFile = "Map-Bar-Harbor.osm" # Small map - downtown Bar Harbor  #MapFile = "Map-Mount-Desert-Island.osm" # Medium size map - MDI only  #MapFile = "Map-Bangor-to-Acadia.osm" # Full map for analysis  MapFile = "Map-Bangor-to-Acadia.osm"  street\_type\_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)  zip\_code\_re = re.compile("\d{5}(?:)?")  expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road",  "Trail", "Parkway", "Commons"]  # UPDATE THIS VARIABLE  mapping = { "St": "Street",  "St.": "Street",  "Ave": "Avenue",  "Rd.": "Road",  "N.": "North"  }  lower = re.compile(r'^([a-z]|\_)\*$')  lower\_colon = re.compile(r'^([a-z]|\_)\*:([a-z]|\_)\*$')  problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')  CREATED = [ "version", "changeset", "timestamp", "user", "uid"]  import pprint as pp  def update\_street\_name(name, mapping):  '''  Function: update\_name(name, mapping)  Inputs: name: Street name from OSM  mapping: dictionary containing standard name for each possible non-standard version  Return: edited street name  '''  # YOUR CODE HERE  #DBG street\_type\_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)  newname = name  m = street\_type\_re.search(name)  if m:  updt = mapping.get(m.group())  if updt :  newname = street\_type\_re.sub(updt, name)  #DBG print ">>> ", name, m, m.group, updt, newname  return newname.title()  ZipCodes = ['04401', '04411', '04412', '04416', '04420',  '04421', '04428', '04429', '04431', '04438',  '04435', '04438', '04444', '04472', '04473',  '04474', '04476', '04496', '04605', '04607',  '04609', '04612', '04614', '04616', '04617',  '04640', '04679', '04660', '04662', '04664',  '04672', '04674', '04675', '04676', '04677',  '04679', '04684', '04693', '04848' ]  def fix\_postal\_code(str) :  m = zip\_code\_re.search(str)  if m :  zp = m.group()  if zp not in ZipCodes :  zp = None  return zp  def fix\_address\_fields(address) :  '''  Function: fix\_address\_fields(address)  Inpiut: Reshaped address object, containing subobject  (see reshape\_subelements)  Returns: new copy of address object with appropriate edits to  specific subfields (e.g. street, postal code)  Applies fix\_\* routines to appropriate subfields, and builds a copy  of address with the possibly updated fields  '''  newaddr = {}  for kw, val in address.iteritems() :  if kw == 'street' :  newaddr[kw] = update\_street\_name(val, mapping)  elif kw == 'postcode' :  zp = fix\_postal\_code(val)  if zp :  newaddr[kw] = zp  else :  newaddr[kw] = val  return newaddr  Split\_words\_re = re.compile("[\s,;]\*([a-z0-9\_]+)")  def fix\_cuisine\_value (val) :  '''  Function: fix\_cuisine\_value (val) :  Inputs: cuisine name from OSM  Output: Cleaned and restructured cuisine name(s)  Clean cuisine values to create a more standard form that will better  support queries. Specifically, two problems in the OSM data are addressed:  1) cusine names do not consistant case, e.g. mexican/Mexican/MEXICAN. This  script will map them to all lower case.  2) Sometimes multiple cuisines are reported as a single value with  inconsistant separators. This function splits names separated by  white-space, commas, or semicolons, and returns a list as the new value.  This is not consistant with the current usage in OSM, but provides a  standardized structure for processing Mongo queries.  '''  value = val.lower()  value2 = Split\_words\_re.findall(value)  # if not val.islower() :  # DBG print ">>> Before: {}, After: {}".format(val, value2)  return value2  def shape\_subelements(element, node) :  '''  Function: shape\_subele=ments(element, node)  Inputs: element: Parent (top level) element  node: Partially built dictionary with reshaped keyowd/value structure  process attributes of second level elements  For each subelement of 'element':  add value of 'ref' attribute to reflist  for 'k', 'v' attrinute pairs:  if k value contains any characters in problemchars, discard  for keywords starting with 'addr:' and containing only one colon,  strip off 'addr:' and add to address dictionary, with valure of 'v' attribute  discard keywords with more than one colon  add other keyword/value pairs to node  when all children have been processed,  add reflist to node as 'node\_refs',  add address to node as 'address'  return updated node  '''  #  # <nd> tag ref attributes are collected as an array and added  # to parent dictionary as 'node\_refs'  # addr: addtributes are collected and added to parent as 'address'  #  address = {}  reflist = list()  tigerdata = {}  gnisdata = {}  for subel in element.iter() :  ref = subel.attrib.get('ref')  if ref :  reflist.append(ref)  k = subel.attrib.get('k')  v = subel.attrib.get('v')  if ( v != None) and (k != None) :  #DBG print "SUBEL: ", k, v  if problemchars.findall(k) :  continue  tmp = k.split(':')  if (len(tmp) > 2) :  continue  if tmp[0] == 'addr' :  address[tmp[1]] = v  elif tmp[0] == 'cuisine' :  node[k] = fix\_cuisine\_value(v)  elif tmp[0] == 'tiger' :  tigerdata[tmp[1]] = v  elif tmp[0] == 'gnis' :  gnisdata[tmp[1]] = v  else :  node[k] = v  if address :  node["address"] = fix\_address\_fields(address)  if reflist :  node["node\_refs"] = reflist  if tigerdata :  node['tiger\_data'] = tigerdata  if gnisdata :  node['gnis\_data'] = gnisdata  return node  def shape\_element(element):  '''  Function: shape\_element(element)  Inputs: element: ElementTree XML element to be processed  Returns reshaped version of the element  Reshapes element and its first level children into a dictionary suitable for loading into MongoDB  Looks at all subelements under this element (one level), and changes keyword/value pairs into  appropriate structures.  - changes tag with 'lat' and 'lon' attributes into list: [lat, lon]  - All attributes in the supplied "CREATED" list are added to a dictionary under 'created'  - Remaining keyword/value pairs are added into top level 'node' dictionary  - Process children, passing the element and the 'node' under construction to shape\_subelements()  - returns the 'node' dictionary built here.  '''  node = {}  pos = []  lon = None  lat = None  if element.tag == "node" or element.tag == "way" :  # YOUR CODE HERE  node['type'] = element.tag  #  # Find next level elements  #  #DBG print "Next level: ", element.text  #  # map lat & lon to position array  #  lat = element.attrib.get('lat')  lon = element.attrib.get('lon')  if lat != None and lon != None :  node['pos'] = [float(lat), float(lon)]  #DBG print "\n\nhave lat and lon: ", lat and lon, "pos: ", node['pos']  #  # Process attributes in "CREATED" list  #  dict\_c = {}  for i in CREATED :  val = element.attrib.get(i)  #DBG print ">>> (CREATED): ", i, val  if val :  dict\_c[i] = val  node['created'] = dict\_c  #  # Process remaining attributes  #  for i in element.attrib :  if i in ['lat', 'lon'] :  continue  if i in CREATED :  continue  val = element.attrib.get(i)  #DBG print ">>> (Next attribute) >>> ", i, " : ", val  #DBG print element.tag, i, val  node[i] = val  node = shape\_subelements(element, node)  #DBG print ">>>"  #DBG pp.pprint(node)  #DBG print "-------------------------------\n"  return node  else:  return None  def process\_map(file\_in, pretty = False):  # You do not need to change this file  file\_out = "{}{}.json".format(MapDirPath, file\_in)  file\_in = MapDirPath+file\_in  print file\_out  data = []  with codecs.open(file\_out, "w") as fo:  for \_, element in ET.iterparse(file\_in):  el = shape\_element(element)  if el:  data.append(el)  if pretty:  fo.write(json.dumps(el, indent=2)+"\n")  else:  fo.write(json.dumps(el) + "\n")  return data  def test():  # NOTE: if you are running this code on your computer, with a larger dataset,  # call the process\_map procedure with pretty=False. The pretty=True option adds  # additional spaces to the output, making it significantly larger.  data = process\_map(MapFile, False)  #pprint.pprint(data)  if MapFile == 'example.osm' :  assert data[0] == {  "id": "261114295",  "visible": "true",  "type": "node",  "pos": [  41.9730791,  -87.6866303  ],  "created": {  "changeset": "11129782",  "user": "bbmiller",  "version": "7",  "uid": "451048",  "timestamp": "2012-03-28T18:31:23Z"  }  }  assert data[-1]["address"] == {  "street": "West Lexington St.",  "housenumber": "1412"  }  assert data[-1]["node\_refs"] == [ "2199822281", "2199822390", "2199822392", "2199822369",  "2199822370", "2199822284", "2199822281"]  if \_\_name\_\_ == "\_\_main\_\_":  test() |

## MongoDB Queries

|  |
| --- |
| Mongo DB queries are listed in the table in section 1.1 of the submission document. |