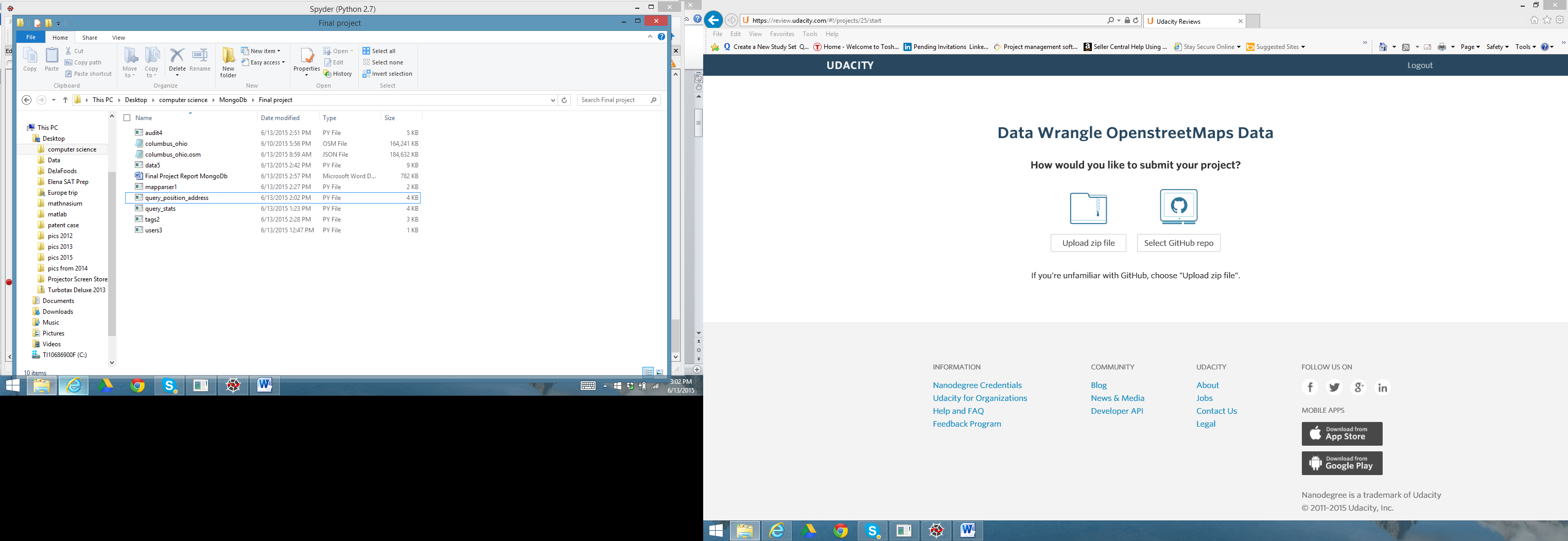
Open Street Map Final Project Submission for the Udacity Course: Data Wrangling with MongoDb

Andy DeJaco

6/13/15

1. Code Functionality & Readability

Python files modeled from lesson 4,5, and 6 were used to audit the data, clean the data, and run queries on the data. The well documented files can be found in my zip file contents.



1. Problems encountered in the Columbus Ohio Map

The street name prefixs and suffixes were abbreviated and inconsistent. For instance:

'W. 5th Ave' , ‘SR 256´, ‘State Route 55’, 'Blacklick - Eastern Rd Ste 700'. After auditing the data and printing out all unexpected abbreviations or all the expected abbreviations, I found that if I split the street name into individual words I could fix each word individually in the list and then put the street name back together successfully. I decided to consistently get rid of abbreviations and spell out the entire word. I looked at the results for this data and verified that I did not change a legitimate “W street” into a mistake by making it “West Street” as I reviewed all the examples in the database using the audit4.py program. The program audit4.py prints out all the street name changes and allows you to peruse the changes that have been made. Note that my program is not general purpose as it could create such a mistake in a region with a legitimate “W” Street and it only correct mistakes in the last 3 word positions in the street name as this was all that was required for this set. The problem examples above were cleaned into “West 5th Avenue”, State Route 256”,“State Route 55”, and'Blacklick - Eastern Road Suite 700' using the mapping below in the data5.py program after sufficient auditing of the problems with the street names in audit4.py. This is consistent with how people in my current home of Columbus Ohio would pronounce these street names. The program data5.py fixed 242 street names in the osm file before importing it to the database. Thus of the 1708 addresses contained in the file, 242 of them were fixed.

mapping = { "St": "Street",

"St.": "Street",

"Ave":"Avenue",

"Ave.":"Avenue",

"Rd.":"Road",

"Rd":"Road",

"rd":"Road",

"Blvd":"Boulevard",

"Blvd.":"Boulevard",

"Cir": "Circle",

"Ct":"Court",

"Dr":"Drive",

"dr":"Drive",

"E.":"East",

"E":"East",

"Ln":"Lane",

"N":"North",

"N.":"North",

"Parkwa":"Parkway",

"Pk":"Pike",

"Pkwy":"Parkway",

"Pky":"Parkway",

"Pl":"Place",

"S":"South",

"S.":"South",

"SE":"Southeast",

"Ste":"Suite",

"SR": "State Route",

"Rt":"Route",

"St.Rt.":"State Route",

"SW":"Southwest",

"Sw":"Southwest",

"W":"West",

"W.":"West"

}

1. Overview of the Data
   1. Name: Columbus, OH USA
   2. Size: ( Columbus\_ohio.osm: 164 Mb, Columbus\_ohio.osm.json: 184 Mb
   3. Users: 668 unique users as found with the program users3.py
   4. Nodes: 740963, Ways: 71518 (from the mapparser1.py)
   5. Number of Nodes with address fields: 1192 (Found from the query\_stats.py, query shown below)
   6. Number of Ways with address fields: 516 (Found from the query\_stats.py)
   7. Number of Nodes with position fields: 740963 (all of them Found from the query\_stats.py)
   8. Number of Ways with positions fields: 0 (Found from the query\_stats.py)

-------------------------------------excerpt from query\_stats.py --------------------------------

#find the number of nodes that contain an address field

db = get\_db('columbus\_ohio')

pipeline4 = [{"$match" : { "address" : {"$exists" :1}, "type":"node"}},

{"$group": {"\_id":"$type",

"count":{"$sum":1}}}]

result = aggregate(db, pipeline4)

address\_num = result.next()["count"]

print "number of nodes that have address fields =", address\_num

---------------------------------------

1. Other Uses of the data

I was hoping to find the address of my house and my business as I live and work in the Columbus, Ohio Metro area. Unfortunately as only 1192 of the 740963 nodes (.2%) had addresses , my two addresses were not listed. I wanted to build a query that could take in position information and produce an address and conversely take in an address and produce a position just like the web apps do. I used an address in a way document to test my approach. The address in the “way” document was 2201 Neil Avenue Columbus OH 43201.

I looked up the address on the web and found the lat,long to be [40.006700, -83.014451].

The first step was to find 1 node in a small region around that position with the following query pipeline where region was defined to be .0001 (all the queries in this section are found in query\_position\_address.py ) :

pipeline = [{"$match" : { "pos.0" : {"$gt" : lat - region, "$lt" : lat + region},

"pos.1" : {"$gt" : long-region, "$lt" : long+region }, "type":"node"}}

That query produced a node id matched to that location information that then could be looked up in a document that contained an address (after reducing the region around the position to only include one node id)

pipeline3 = [{"$match" : { "node\_refs" : {"$in" : [nid]},"address" : {"$exists" :1}}}]

print "Here's the location address ", aggregate(db, pipeline3).next()["address"

Converting from a position to an address worked pretty well so I tried to go the other way by starting with an address with the following query:

house = "2201"

street = "Neil Avenue"

zipcode = "43201"

pipeline6 = [{"$match" : { "address.housenumber" : house, "address.street":street,

"address.postcode": zipcode}}]

node\_ref = aggregate(db, pipeline6).next()["node\_refs"][0]

print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

print "node reference number for the address is =",node\_ref

pipeline7 = [{"$match" : { "id" : node\_ref}}]

position = aggregate(db, pipeline7).next()["pos"]

Which produced the position [40.0067418, -83.0145063] which is pictured on Ohio State’s campus below. Note that when the address is matched and it returns a “way” which has a few node Ids, each with a different location. I just picked the first one which could lead to some error. Ultimately, it would be very helpful in the database if every node document that is positioned on an address would contain that house address. I think this could be done programmatically by:

looking through every node in this Columbus\_ohio.osm file

looking up that node position on a website and returning an address

checking against a web query that the address produces a position very close to the original node position (what if the node was not an address location we don’t want to put in a wrong address?)

adding the checked address to the osm file

