**Project 1: Mongo DB**

Sarah Cooney, Mingyaun Li, Jason Meng

# **1. Importing to a Database in MongoDB**

The first step was setting up the database in MongoDB. We downloaded the zipcodes.json file and used the import command to create the database and upload the data. Figure 1 shows a sample of data from the file. Figure 2 shows the import command used to create the database. The mongod command must be run to establish a connection to the server before running the import command.

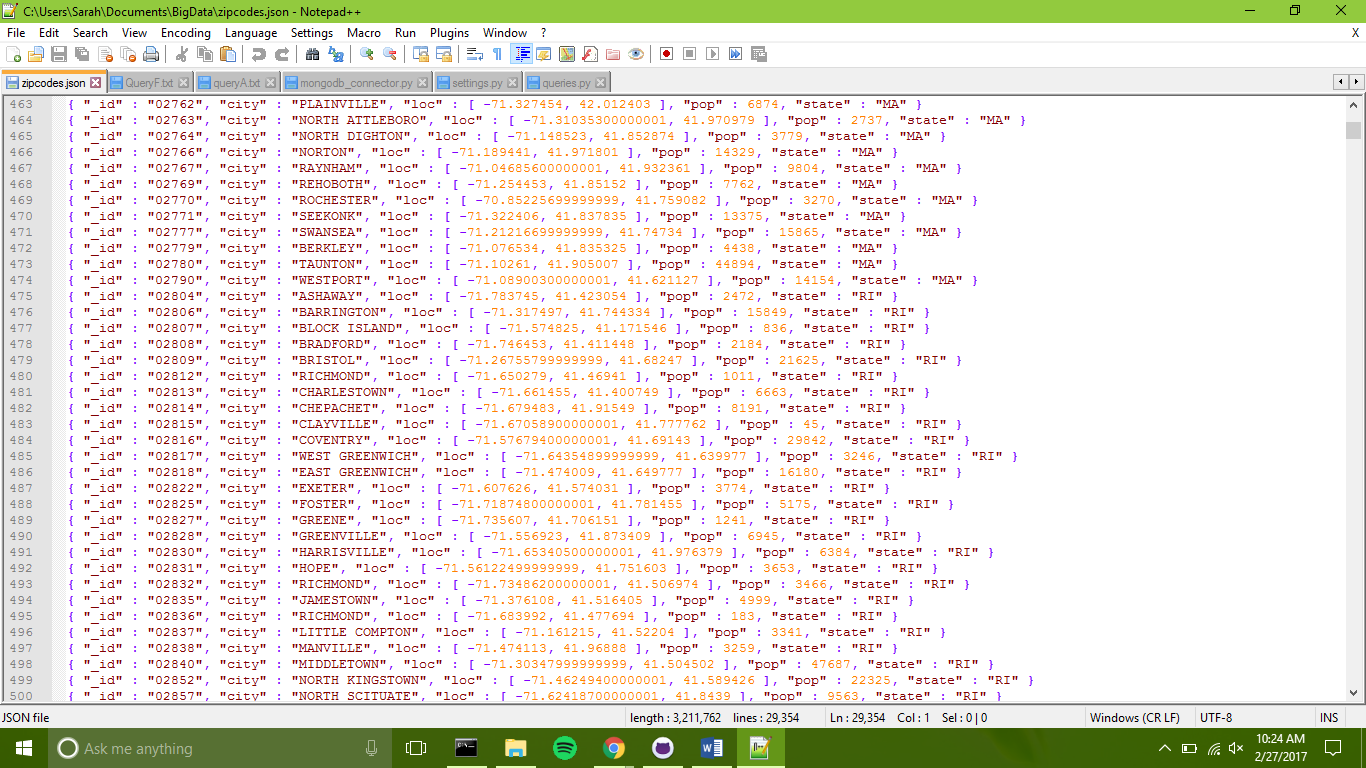
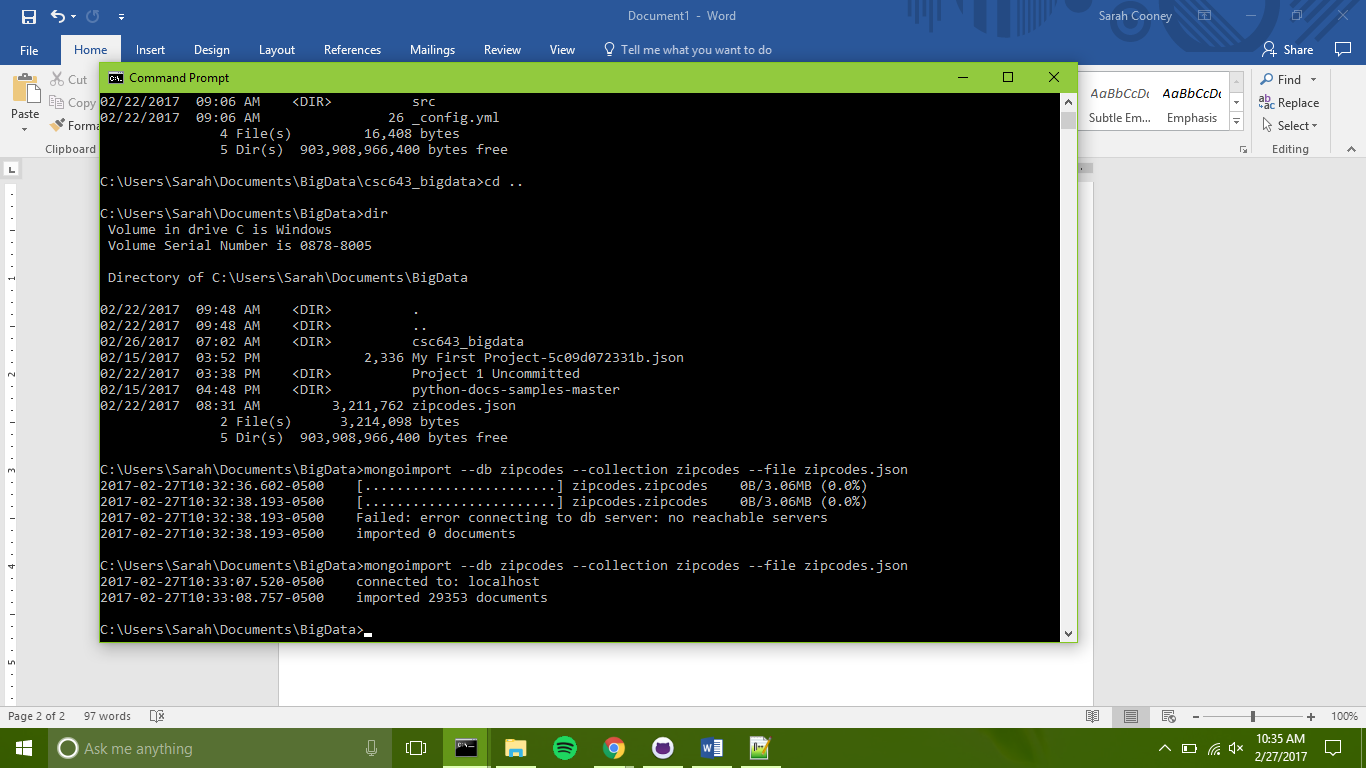


Figure 1. Sample of Zipcodes.json

Figure 2. Command to import zipcodes.json as a mongo database



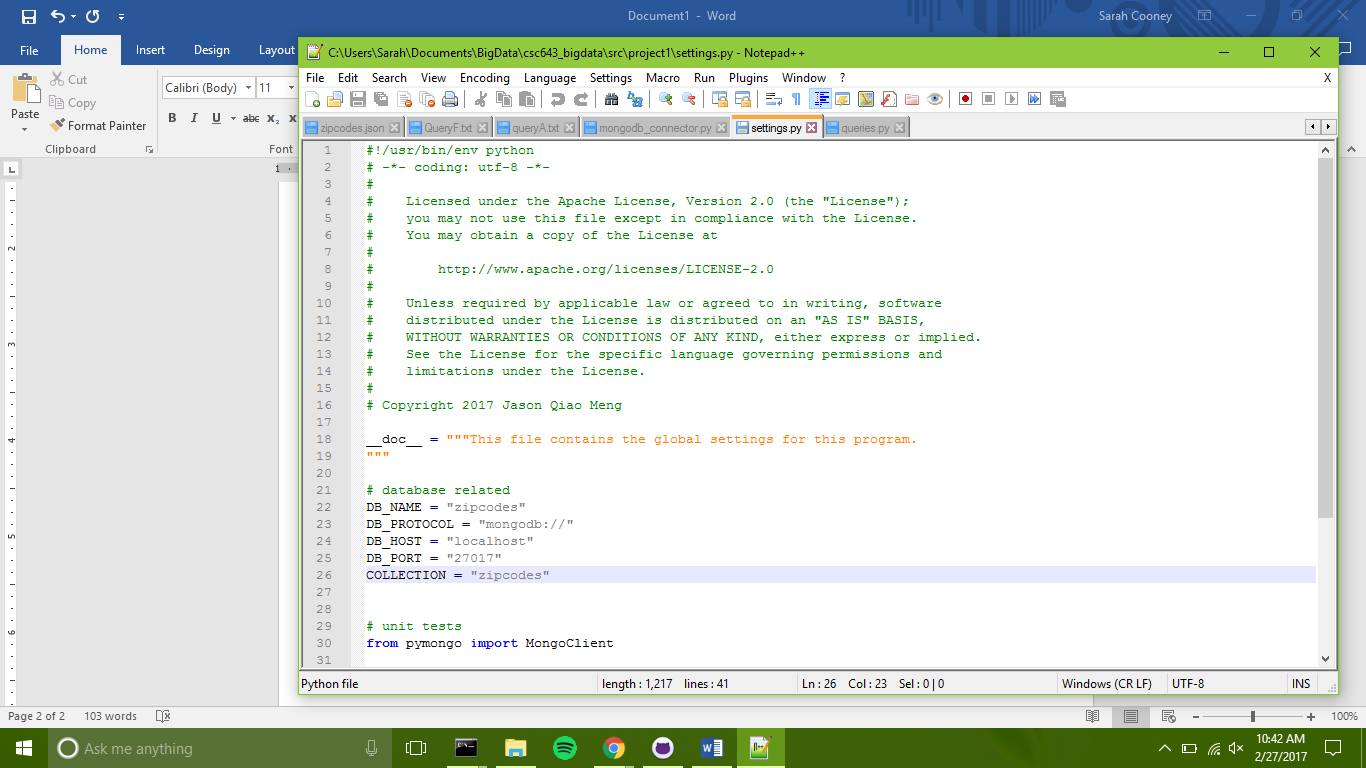
# **2. settings.py and mongodb\_connector.py**

These files use python to establish a connection to the zipcodes mongo database. The mongod command must be run in the command line first to start the server before using these files to establish a connection to the database.

## **2.1 settings.py**

The settings.py holds variables containing the global settings for connecting to a database with mongod\_connector.py. The variables are shown in Figure 3. DB\_NAME and COLLECTION are used to specify which database and collection to use after the connection has been establish. DB\_PROTOCOL specifies the type of database connection, in this case MongoDB. DB\_HOST and DB\_PORT specify the server and port on which to connect.

Figure 3. settings.py



## **2.2 mongodb\_connector.py**

This file uses the pymongo library to create a connection to a MongoDB database. The e variables defined in settings.py are utilized in this file. The file contains a class called MongoDB which is used to connect to the mongo client and get an instance of a database. Figure 4 shows the \_\_init\_\_ method which creates and initializes an empty instance of the MongoDB class.

The code for the simple\_connection\_string method is shown in Figure 5. This method sets up and returns the string used by pymongo to connect to the Mongo client, and incorporates the protocol, host, and port variables defined in settings.py. The get\_client method actually establishes and returns a connection to the Mongo client. The method starts by checking if the client exists. If it does not, the simple\_connection\_string method is used with the pymongo MongoClient method to attempt to make a connection. If the connection fails, an error message is presented and the client is set to “None” and returned. Otherwise, the successfully connected client is returned. The code for the get\_client method is shown in Figure 6. The next method in the class is get\_database. This method is used to access a specific

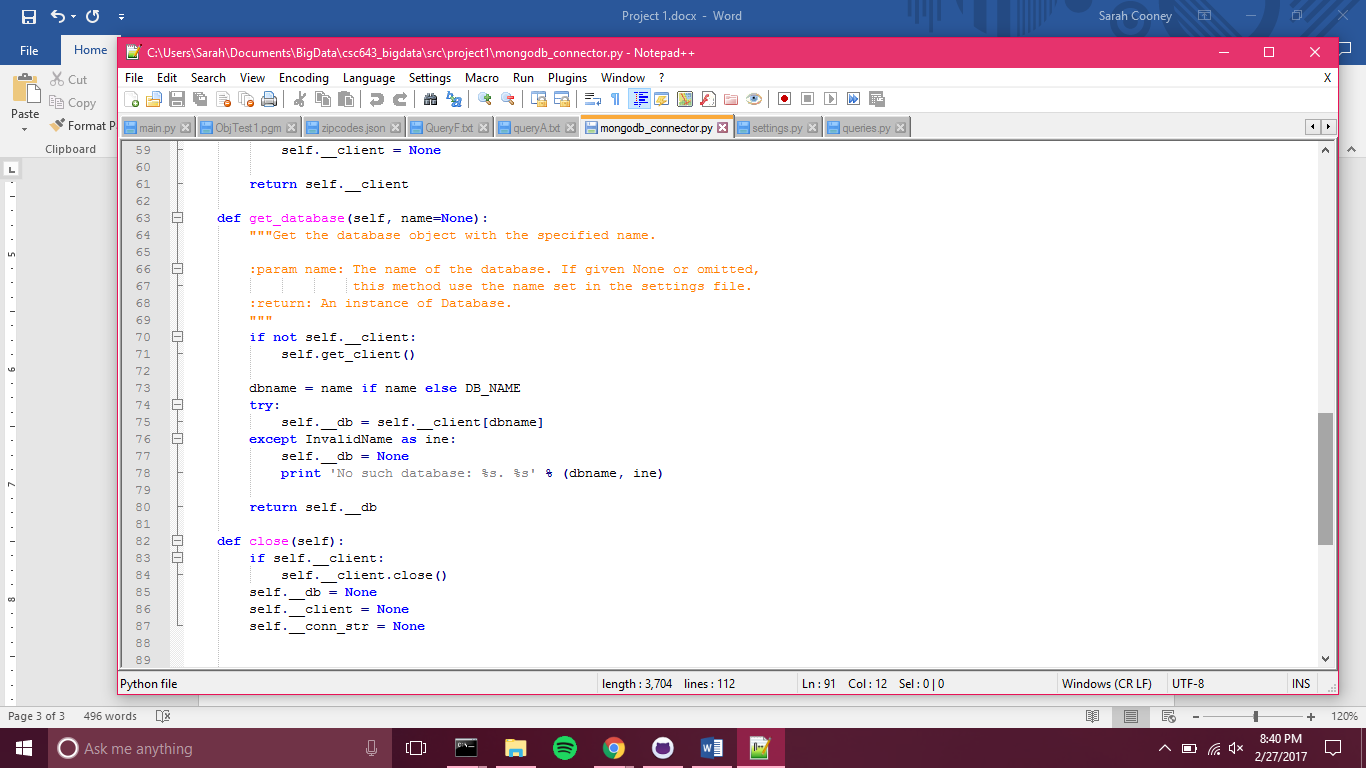


Figure 7. The method to get a specific database instance

Figure 6. The method to connect to the mongo client

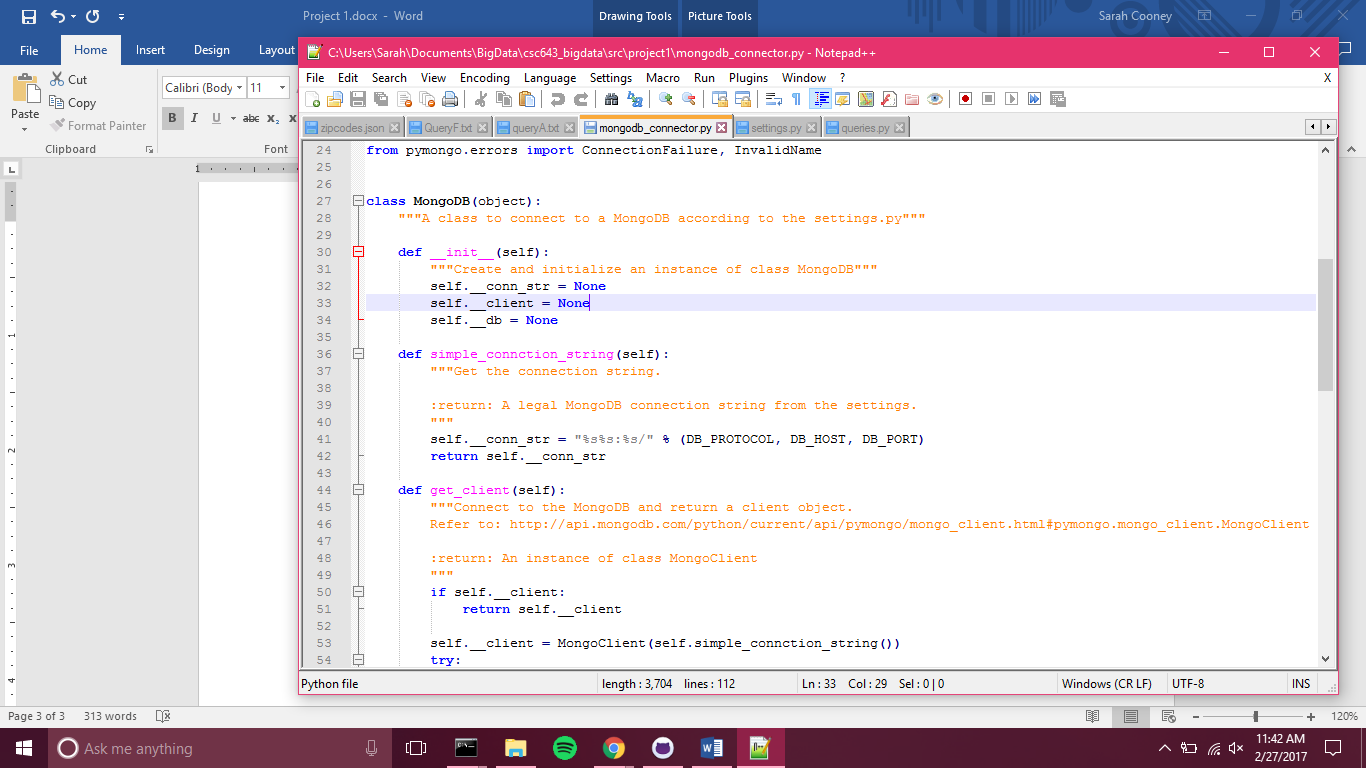
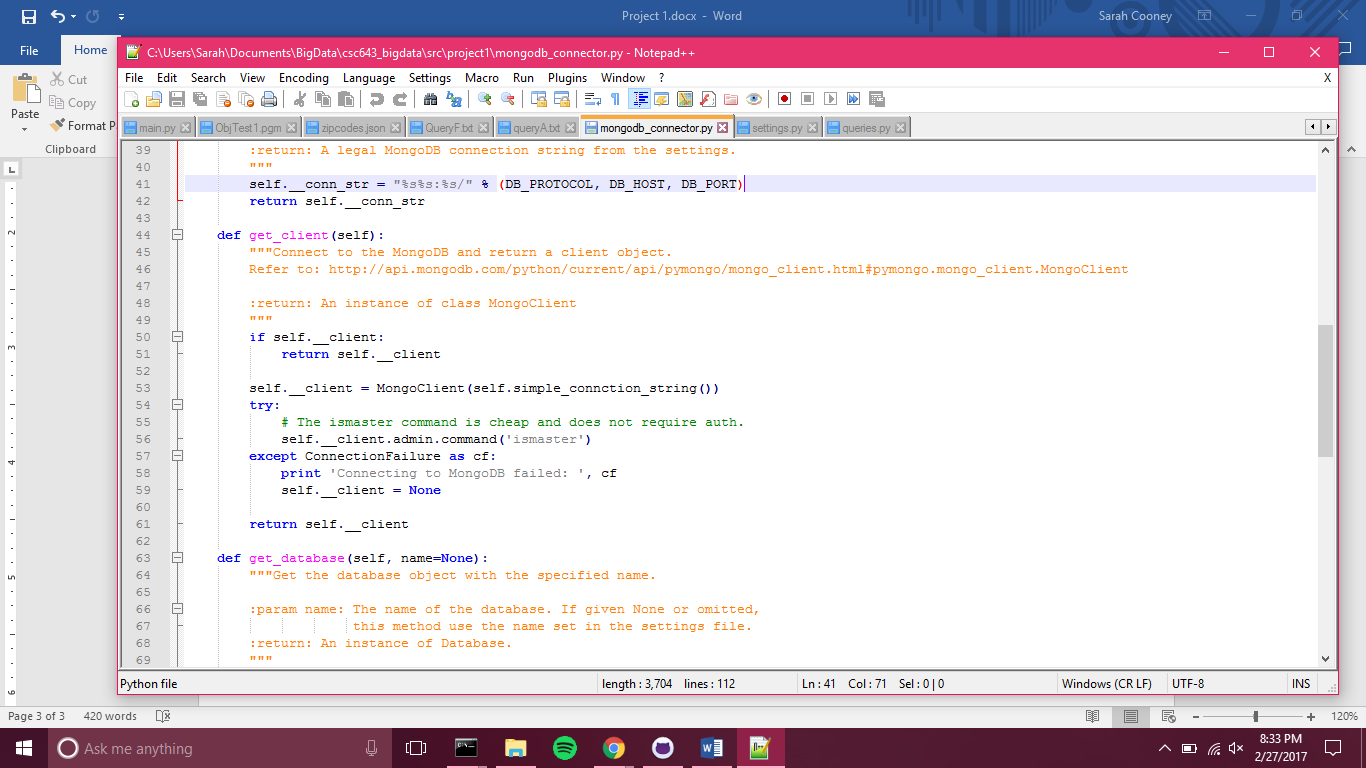


Figure 5. The method to set up the string used to connect to the Mongo client

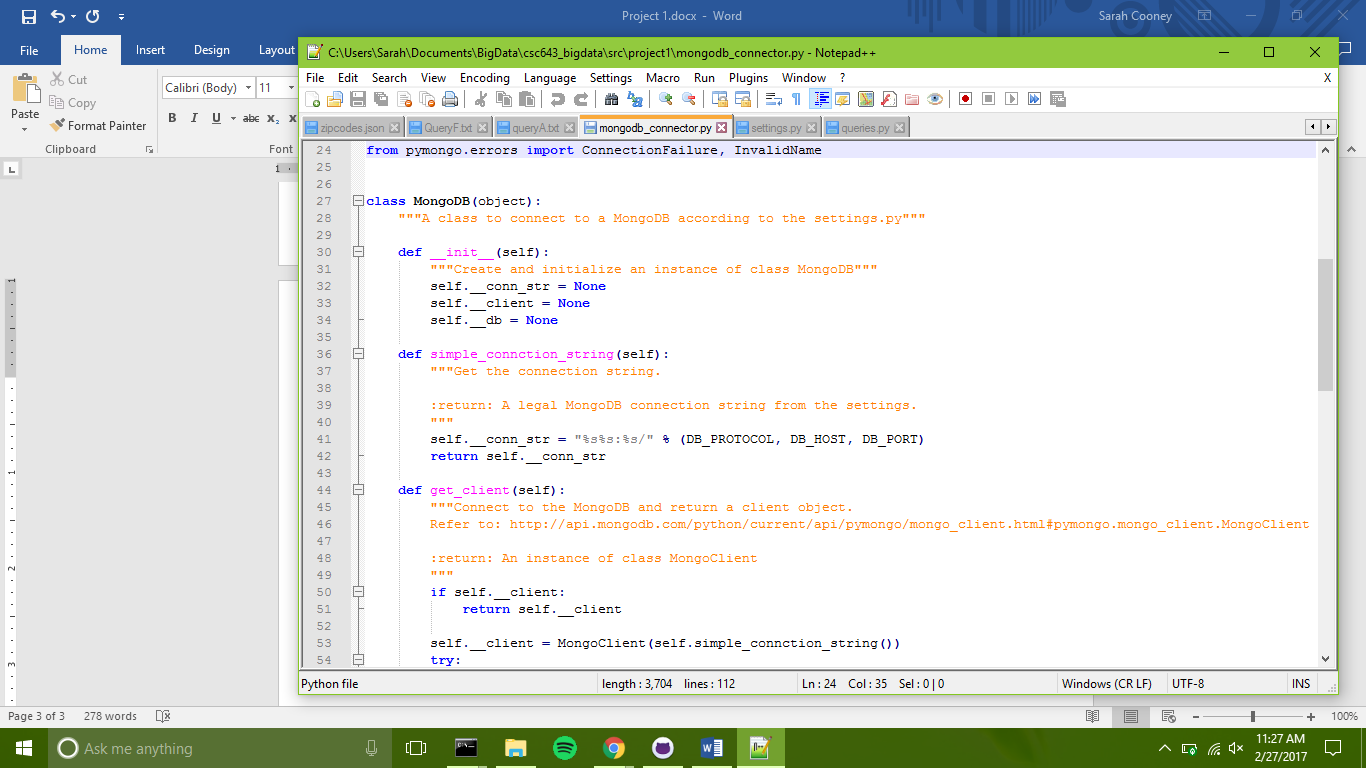


Figure 4. The initialization method for the MongoDB class

Mongo database. The method takes an optional name parameter. If no name is specified, the method defaults to the name variable defined in settings.py. An error is thrown if no database of the specified name exists, whether by parameter or from the settings.py variable. Figure 7 shows the get\_database method.

The final method in the class, shown in Figure 8, is the close method. This function closes the connection to the MongoDB client and sets all of the instance variables back to a null state.

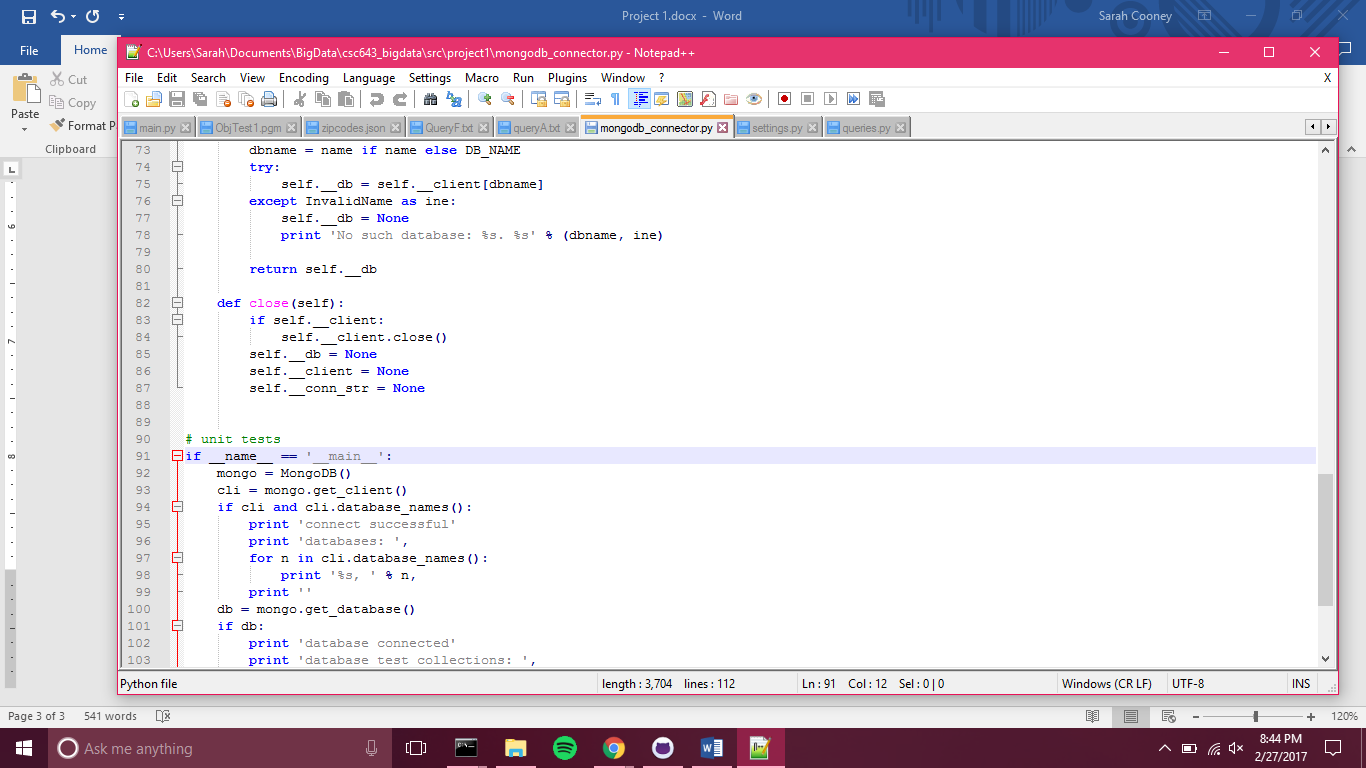


Figure 8. The method to close the Mongo client instance

# **3. Mongo Queries**

This section will describe the setup of the queries.py file and give details about each of the six queries. Screenshots of the output for each query will also be shown.

The queries.py file contains a method for each of the six queries and a main section that runs all six queries in succession.

### **3.1 Query A**

Query A: Find the total number of cities in the database.

This query, shown in Figure 9, is a simple pymongo query. First, the get\_database method from the MongoDB class is called with no name parameter, which means it will access the database named in settings.py. Then, the settings.py collection variable is used to get the collection containing all of the data from the zipcodes.json file. The distinct command is used with the ‘city’ parameter to find all of the distinct cities in the database, since some have multiple zipcodes; for instance, Boston. The python len method was called on the collection that was returned to find the number. Figure 10 shows the output from this query.

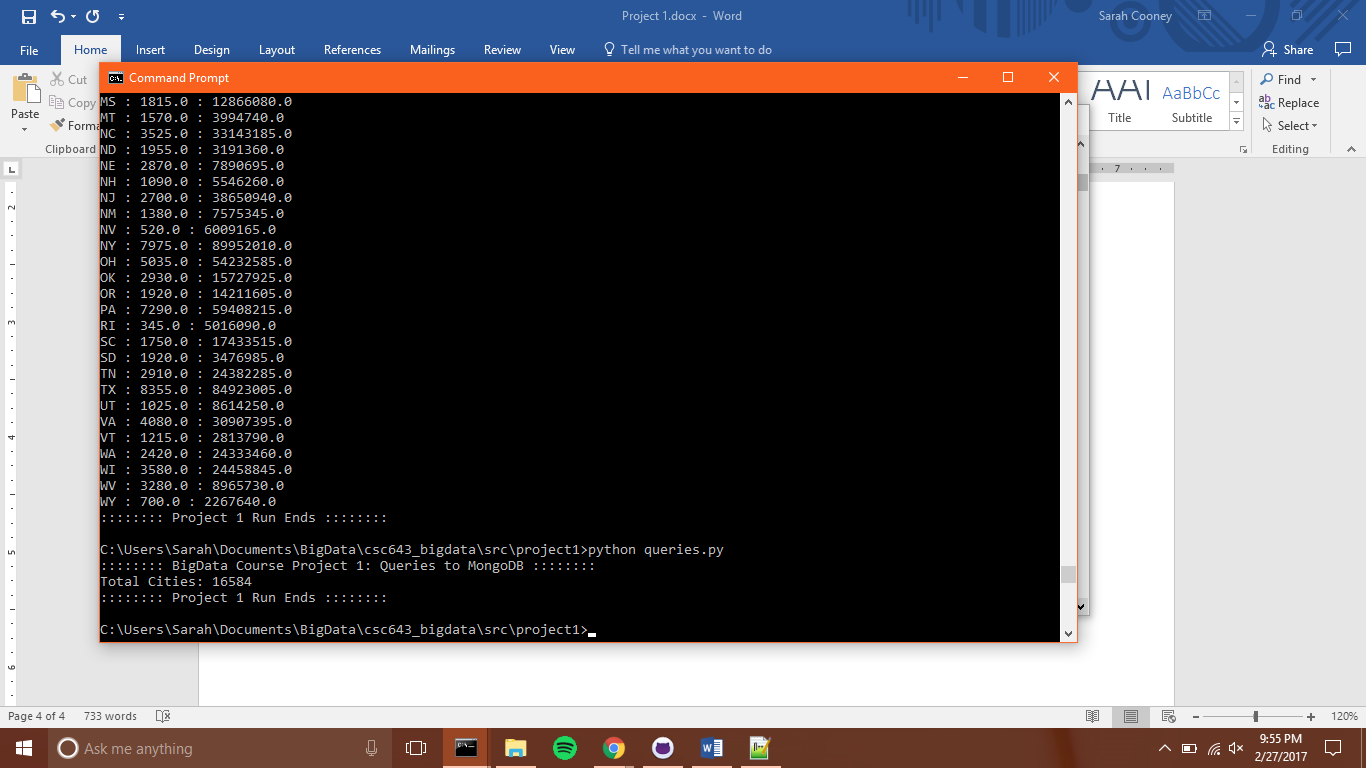
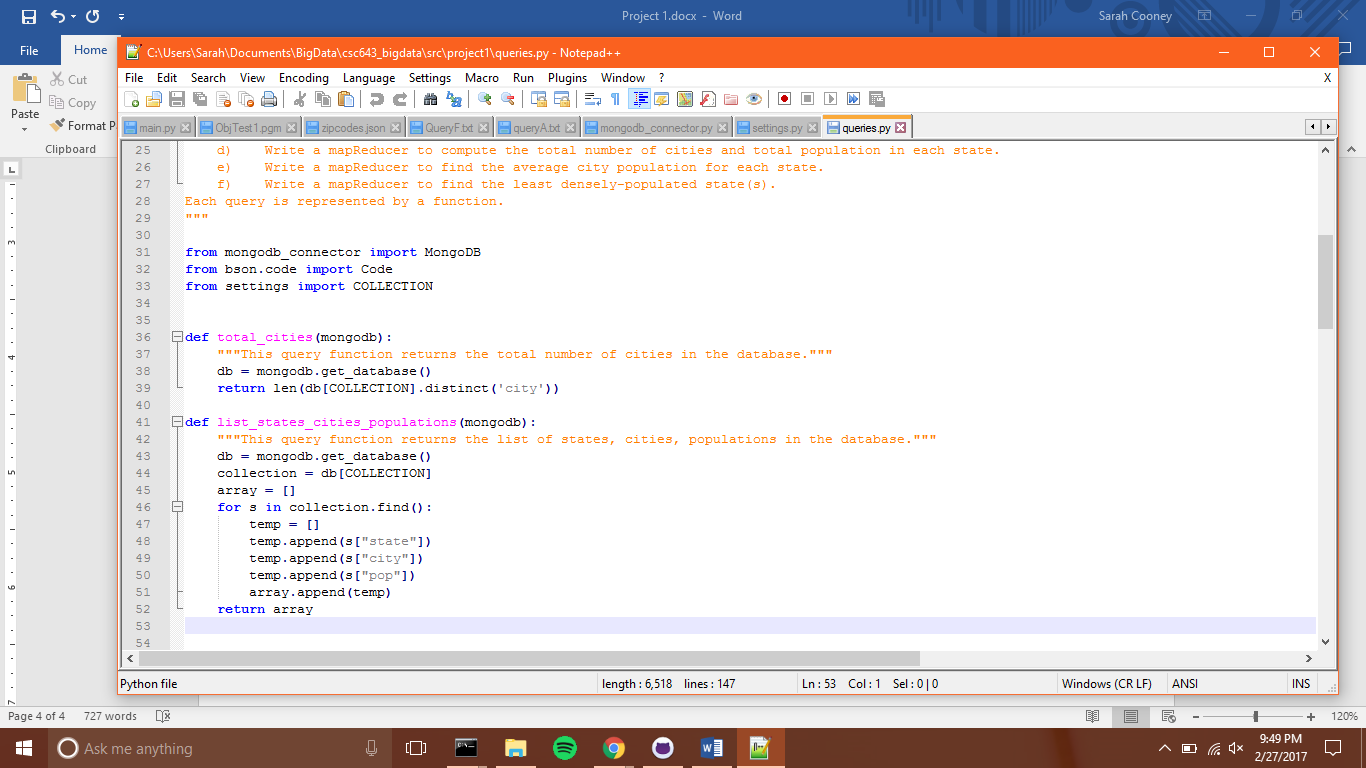


Figure 10. Output for query A

Figure 9. Code for query A



### **3.2 Query B**

Query B: Create the list of states, cities, and city populations.

Like query A, the first step taken in this query is to access the database and collection defined in settings.py. An array is created, and for each entry in the database, the state, city, and population information are appended and then added to the array. The method returns this array, which is printed in the output. Figure 11 shows the code for this query, and Figure 12 contains a sample of the output produced.

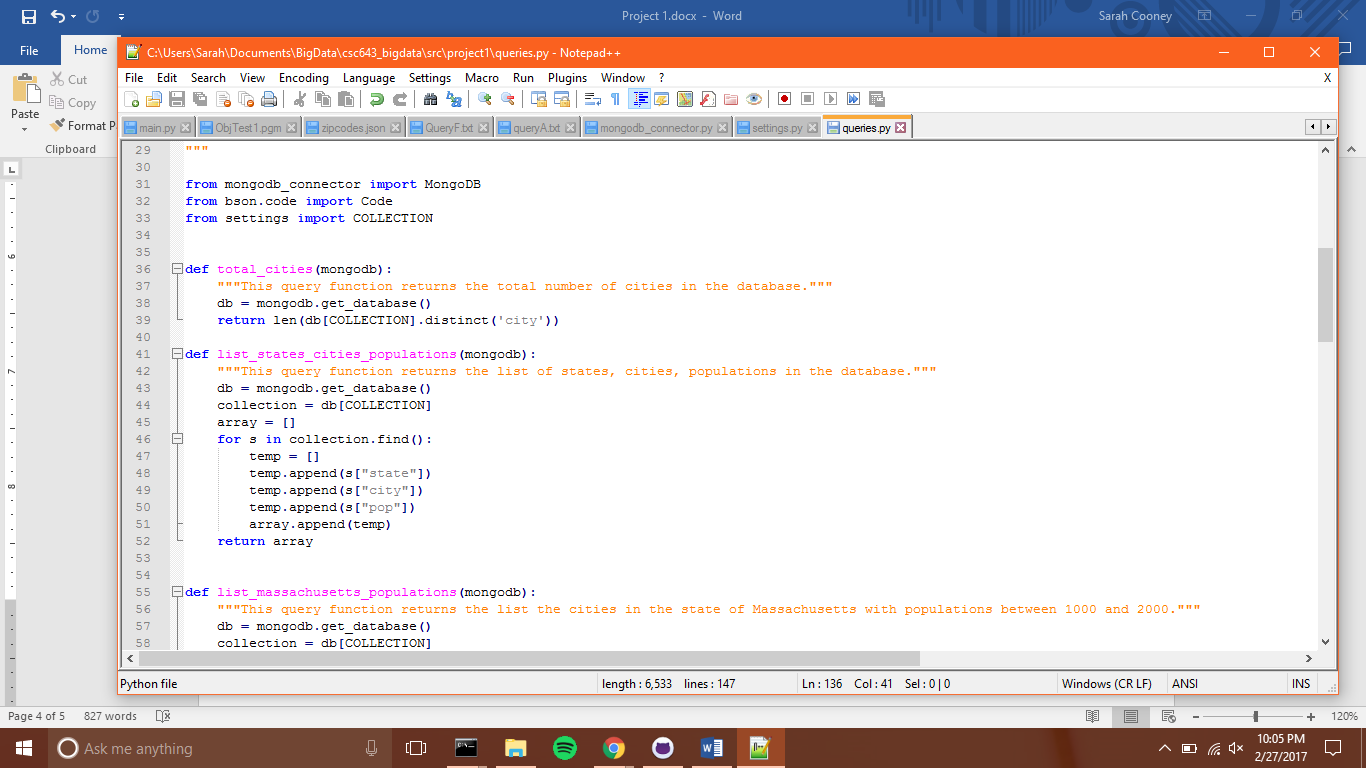


Figure 11. Code for query B

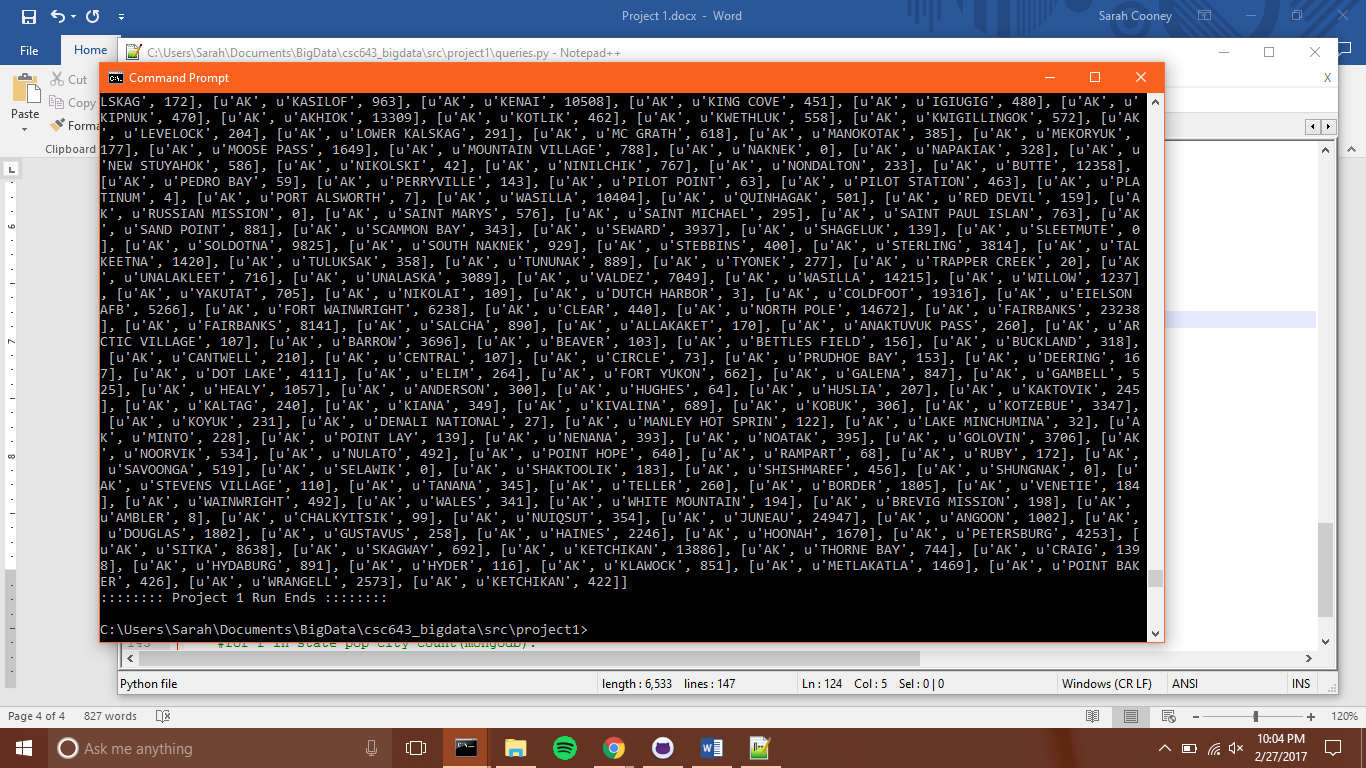


Figure 12. Some output for query B

### **3.3 Query C**

Query C: List the cities in the state of Massachusetts with populations between 1000 and 2000.

This query first establishes a connection to the database and collection specified in settings.py. Then, the Mongo find command is called on the collection. Regex-type parameters are included with the find command to filter and return just cities in the state of Massachusetts and then just those with populations between 1000 and 2000. The results of the find are returned as a list. The code for query C is shown in Figure 13, and the output is shown in Figure 14.

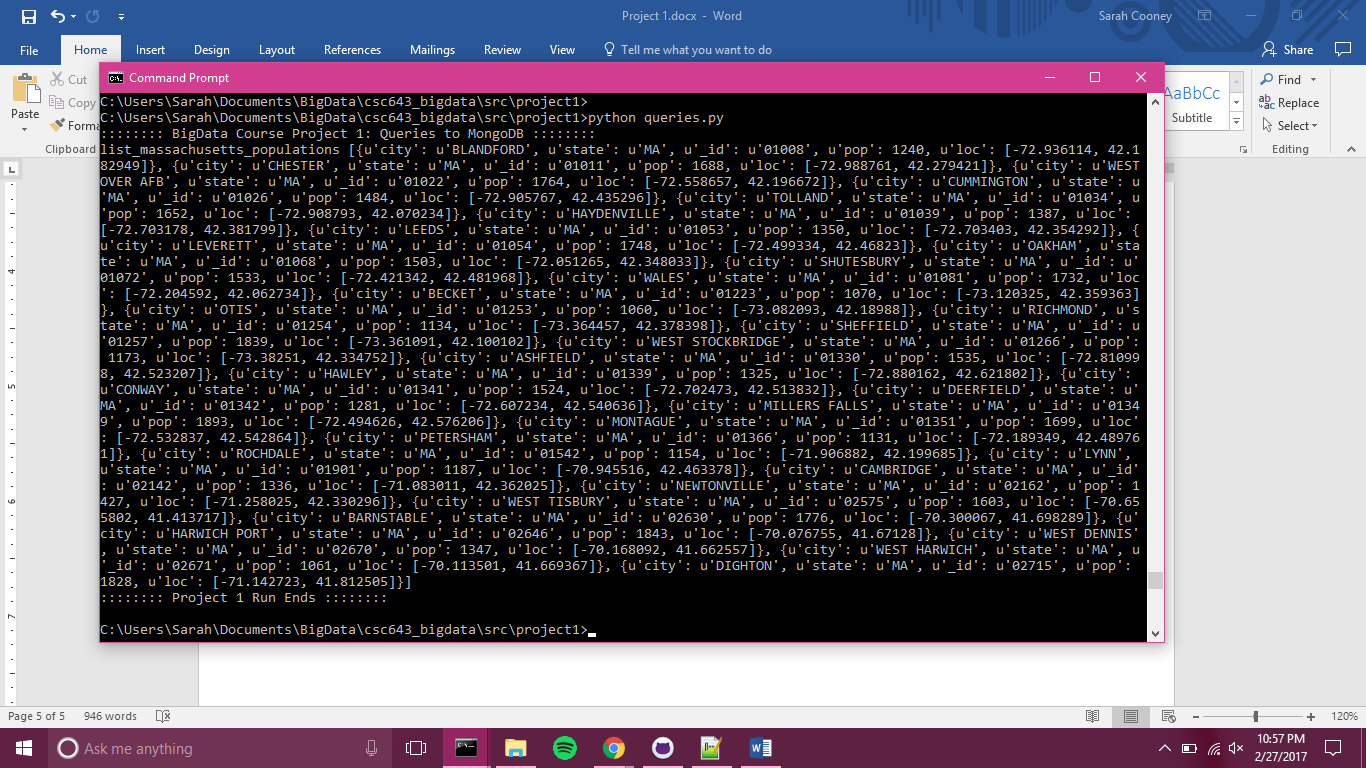
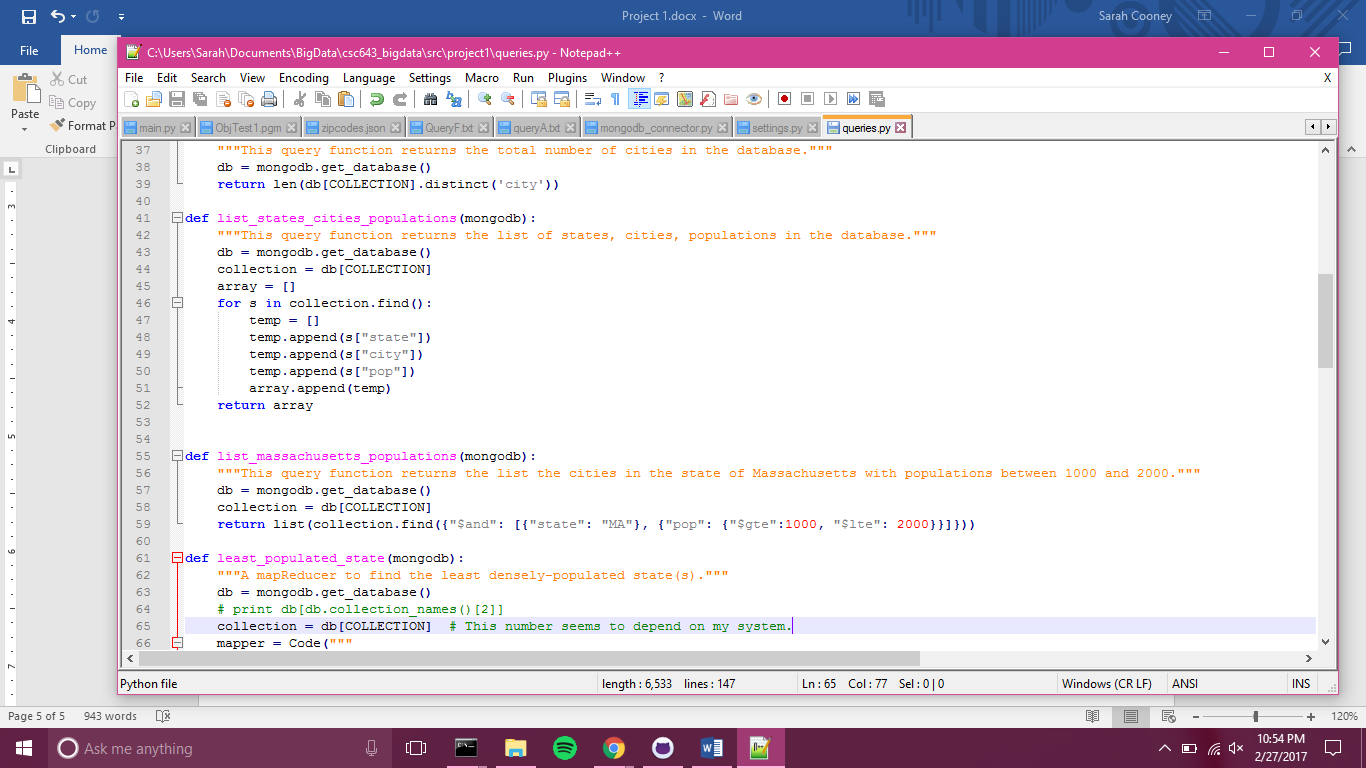


Figure 14. Output for query C

Figure 13. Code for query C



### **3.4 Query D**

Query D: Write a mapReducer to compute the total number of cities and total population in each state.

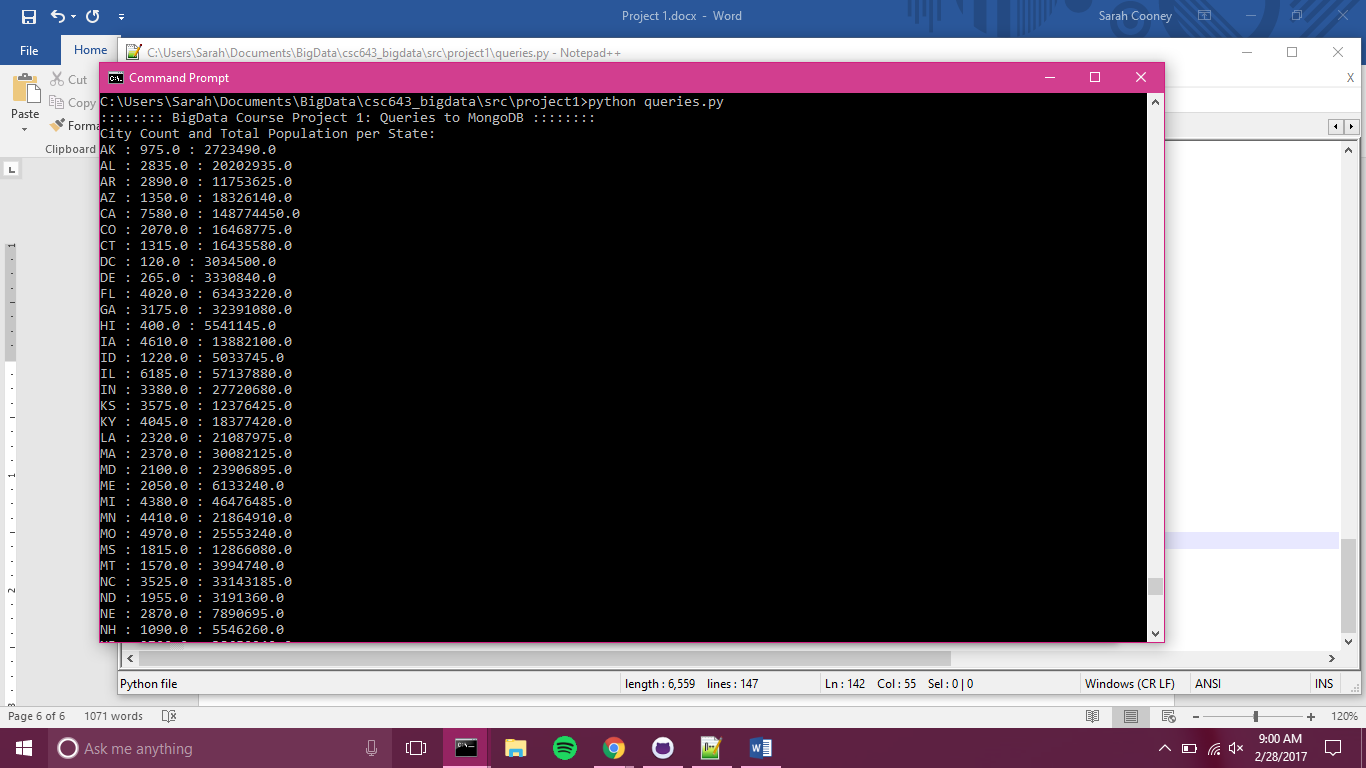
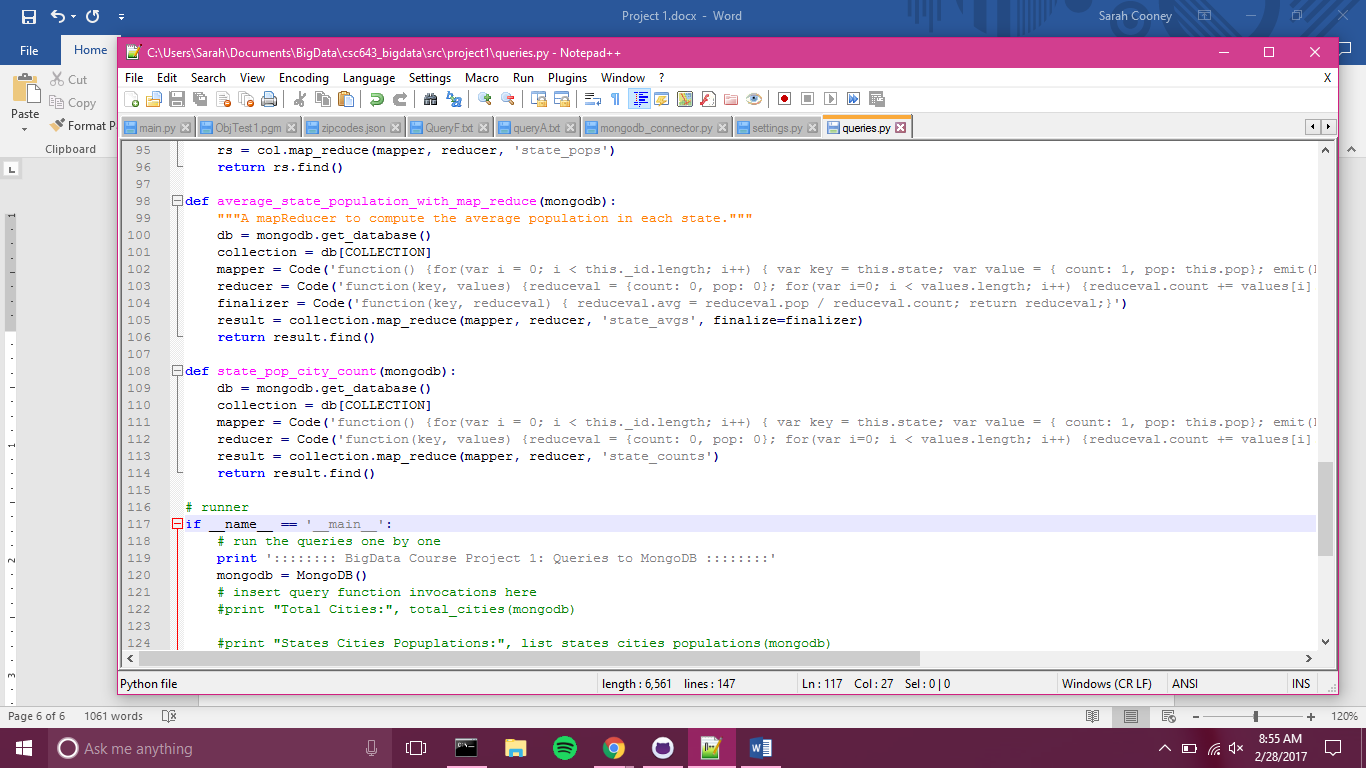


Figure 16. Output sample for query D

Figure 15. Code sample for query D



The mapper for this function looped through each entry in the zipcodes collection and created a key value pair where the key was the state and the value was a count field set to one and a population field containing the population for that zipcode. The reducer looped through these values and aggregating on the state key added the number of cities and total population for each state. This result was returned. Figure 15 contains a sample of the code for query D, and Figure 16 a sample of the output.

### **3.5 Query E**

Query E: Write a mapReducer to find the average city population for each state.

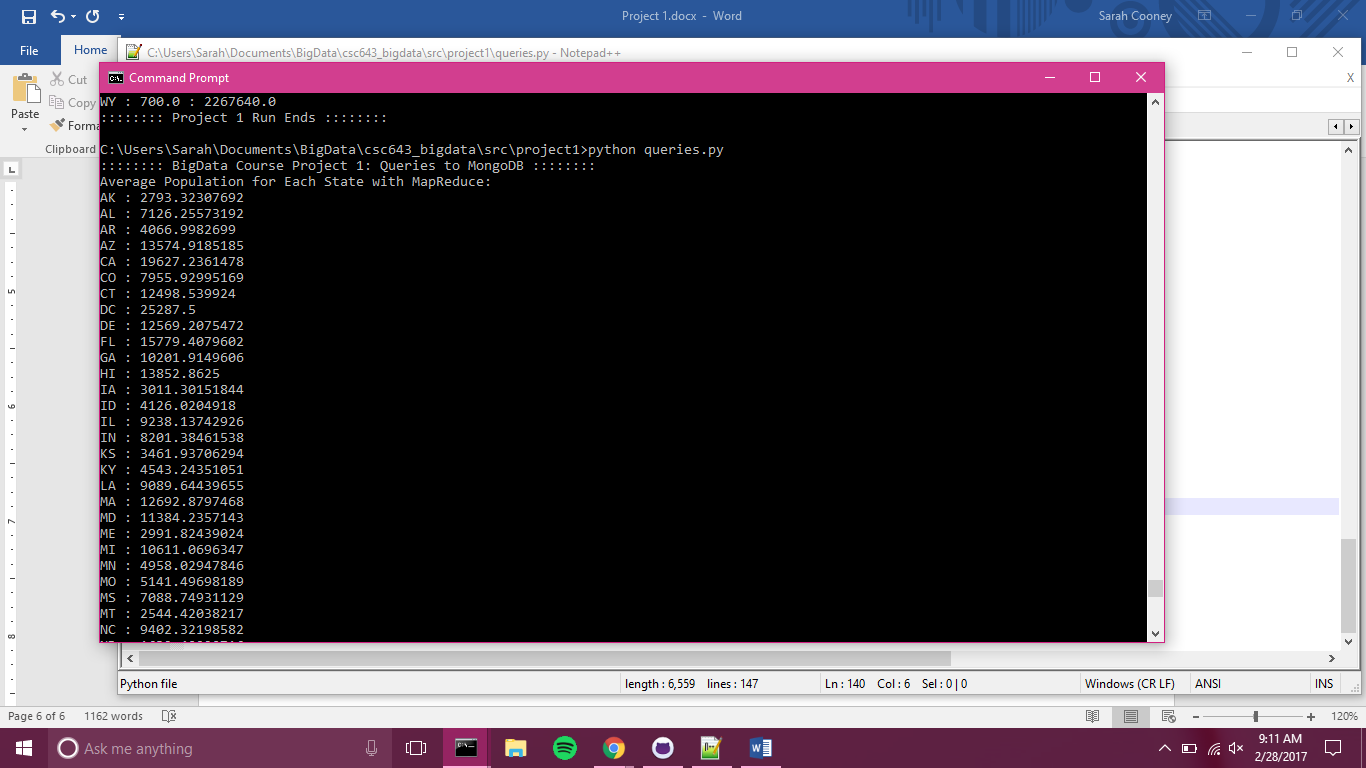
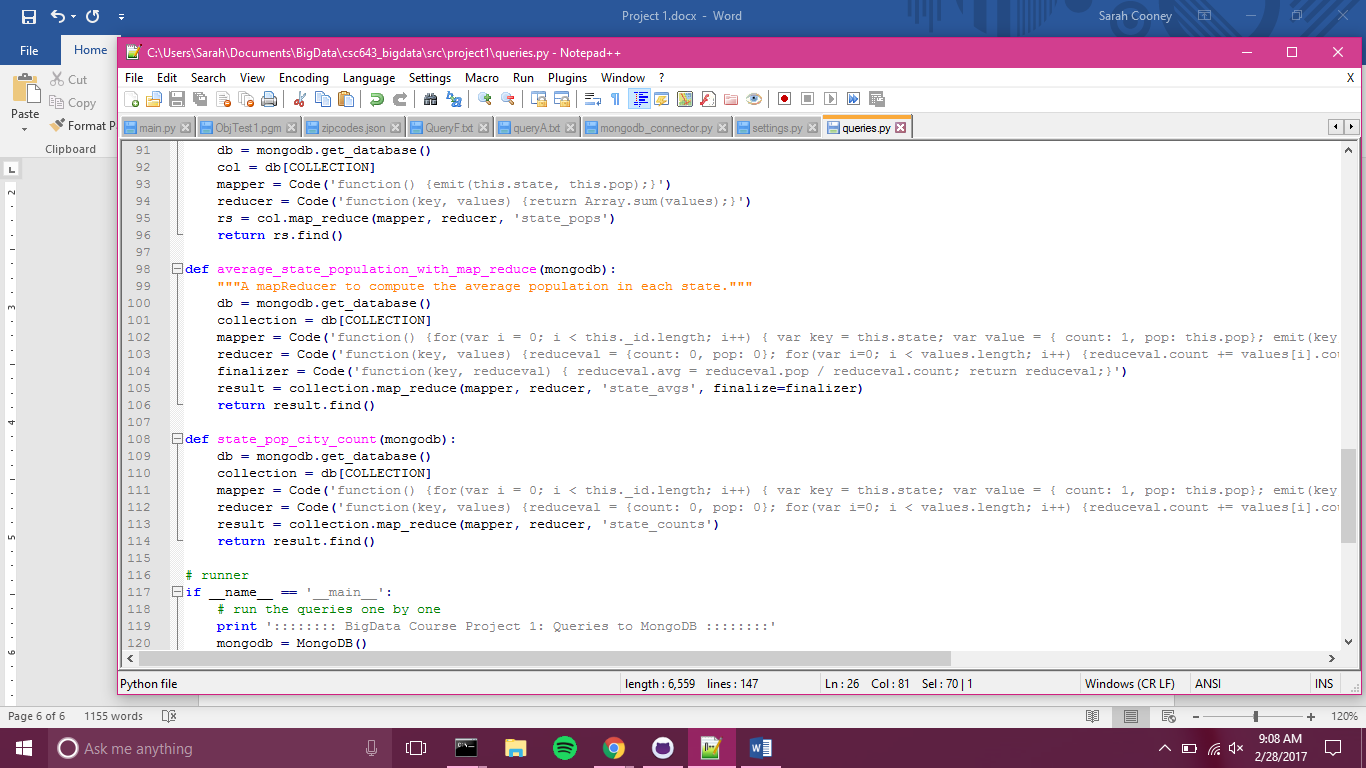


Figure 18. Output sample for query E

This query is built from the code for Query D. It uses the same mapper and reducer. However, a finalize method is added to compute the average from the number of cities and the total population for each state. A sample of the code for query E is shown in Figure 17, and sample output is shown in Figure 18.

Figure 17. Code sample for query E



### **3.6 Query F**

Query F: Write a mapReducer to find the least densely-populated state(s).

The mapper for this query emits the state as a key and population as the value for each piece of data. The reducer then sums the population for each key (state). The reducer returns a collection containing the fields state and total population as key and value, respectively. The find command is then used on this collection in conjunction with the sort command, which is run on the population totals, and returned in ascending order. The results of the find are then limited to one, which is the state with the smallest population. The code for query F is in Figure 19, and Figure 20 displays the output.

Figure 19. Code for query F

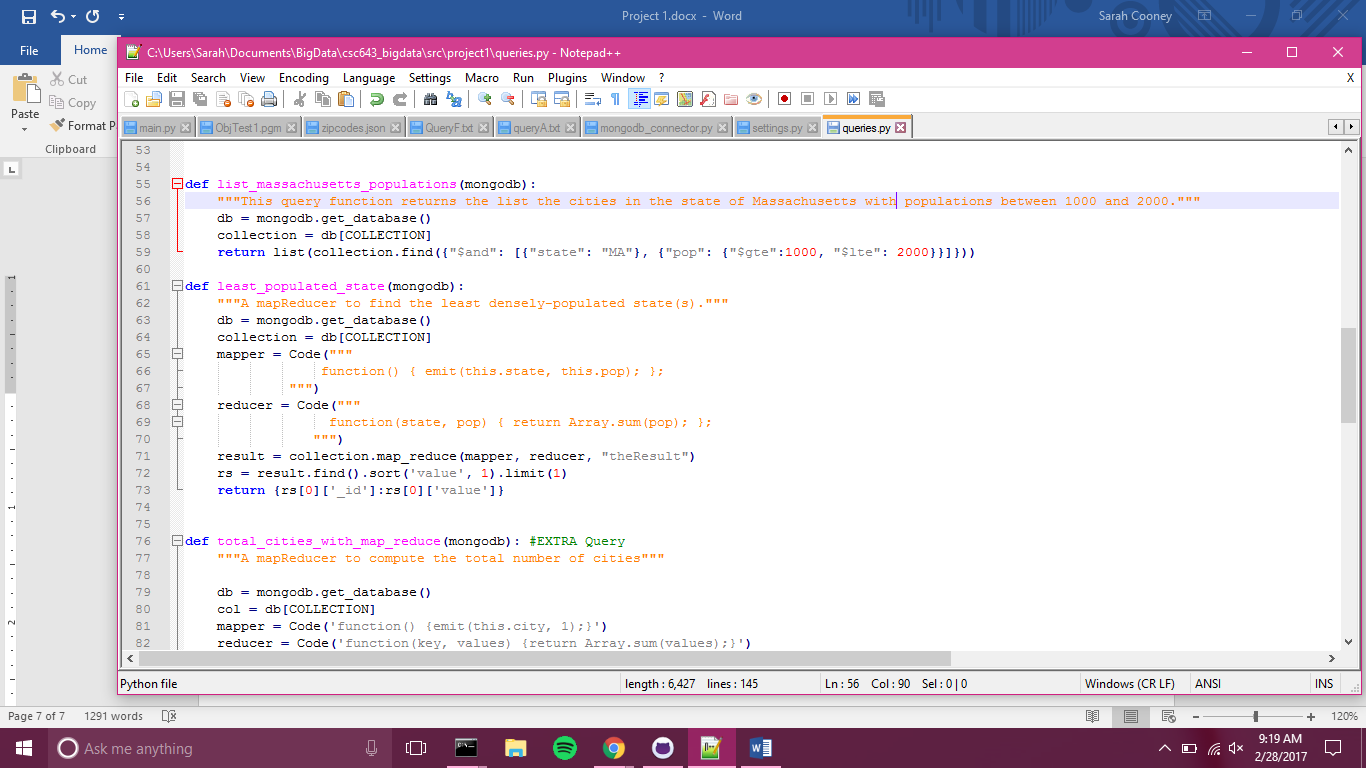


Figure 20. Output for query F

