Lab 4 – v1.1

Introduction

This lab requires you to develop an object-oriented script that reads some data on US States from two text files and then prints some reports based on that data. It is similar to Lab 3 up to a point.

The first data file is the ***us_states.txt*** file that we used in Lab 3. Reading the data from this file should follow the same steps as in Lab 3: read the text lines, remove the newlines, split by comma, strip the spaces around the data values, and convert some data values to integers. Please review the Lab 3 document for details. However, after we read and prepare the data in this way, we don't store it in a list of lists, as we did in Lab 3. This time, we create an object (an instance of the class State) for each state, and we store all these State objects in another object (an instance of the class StateList).

The second data file needs to be created by you, and named ***us_states_adjacency.txt***. This file should have one line for each US State, just like us_states.txt. The line should list all neighboring states of that state, separated by comma. These are two sample lines from that file:

```
Delaware, Maryland, Pennsylvania, New Jersey
Florida, Alabama, Georgia
```

That is, Delaware borders Maryland, Pennsylvania, and New Jersey, while Florida borders Alabama and Georgia. You need to type similar lines for all states. You will take the adjacency information from this **network graph** ("**sociogram**") figure. Please read the entire page, as we will refer to it later. You may also want to consult this image. Be careful when typing the data information into your *us_states_adjacency.txt* file. Typos could affect the results of your program.

The instructor provided a ***lab4.py*** file in Blackboard. That file shows what you need to do in this lab, and also defines the structure of your solution. You are required to download this file on your computer, rename it (by appending your last name(s) as usual), and modify it to develop your solution. This is the only file that you need to submit for grading.

You are not allowed to change the names of the classes or methods already defined in *lab4.py*, nor their parameters. You are also not allowed to change the statements that follow the `if __name__ == "__main__"` statement. You are allowed to remove the "**pass**" statements (which do nothing) from the class methods and replace them with your own statements. You are also allowed to define new functions, methods, and classes – although you shouldn't need to do so. The remaining sections of this document analyze the contents of the *lab4.py* file, and outline the changes that you need to make to that file to develop your solution.

The State Class

This class defines the data attributes that we want to use for each US state: **name** (string), **capital** (string), **population** (int), **area** (int), **seats** (int), and **neighbors** (list of strings). You should use these exact names and types for the attributes that you define in the constructor of this class. Note that the constructor needs to extract (parse) and aggregate the values for these attributes from a text **line** that was read from the *us_states.txt* file, and from the **neighbors** parameter, which is a dictionary that will be described in the section on the **Neighbors** class below. The job of this constructor is to define the six attributes and assign them the appropriate values for the current state (self).

The **__str__()** method should return a string that concatenates the six attributes of each state in a format that will be used when we generate the reports. For example, these are two lines from the report that lists the states sorted by population:

```
             Florida          Tallahassee   19317568      65758 27   Alabama, Georgia
            Illinois          Springfield   12875255      57914 18   Iowa, Wisconsin, Indiana, Kentucky, Missouri
```

That is, the state name is printed on 20 characters, aligned to the right, the capital name is printed in the same way, the population is printed on 10 characters, the area is printed on 10 characters, and the house seats are printed on 2 characters. These five attributes are separated by one space. After them, we print three spaces and then we print the neighbors attribute as a list of state names separated by a comma and a space.

The StateList Class

First, notice that this is a subclass of the built-in list class. That is, this class essentially defines a list of State objects: self[0] is a State object, self[1] is another State object, and so on, because **self** is a list that has the additional methods that we defined in StateList.

The **__init__** constructor reads the **us_state.txt** file line by line and then calls the State constructor for each text line, so that a State object is created for each US State. After creating all State objects and appending them to the **self** list, the constructor calls the **checkIntegrity**() method. This method will verify that all state names that are used in the neighbor lists of all states are legitimate. In other words, if A is a state in StateList, and S is the name of a state in A's neighbors list, then there exists another object B in the StateList such that S == B.name. We do this integrity check because our script merges state data from two different text files (**us_states.txt** and **us_states_adjacency.txt**) which could contain mismatches or typos. If the checkIntegrity() method finds a neighbor that breaks the constrain, the method should print the names of states S and A, and then call **sys.exit()** to quit the script. This gives the programmer (you) the opportunity to fix the text files and run the script again.

Next, we define the **printSortedPopReverse**() method, which prints all states sorted by populations, from the largest to the smallest. We already showed how the output of this method should be formatted. And we did this in Lab 3 too.

The **printSortedDegreeReverse**() method prints all states sorted by their degree in the network graph. The degree of a state is the number of neighbors of that state. These are two sample lines from the output of this method:

```
    Rhode Island           Providence    1050292       1545  2   Massachusetts, Connecticut [2]
  South Carolina             Columbia    4723723      32020  7   Georgia, North Carolina [2]
```

All the six attributes should be printed in the same way that we've seen before. After them, we print the degree of that state in square brackets.

The **computeNetworkDensity**() method should compute the density of the entire sociogram (network graph). The density is the number of actual neighboring relationships between states divided by the maximum possible number of neighboring relationships – that is, assuming that all states bordered each other. The density for this sociogram is 0.91, as indicated in the webpage that we read earlier. You are required to find the formula for computing this value and implement it here.

The **printTriStates**() method should print all tristates. A tristate is a list of three states that border each other. That is, every state in this list borders the two other states. There are more such triplets in our data than we might think. These are two sample lines from the output of this method:

```
Tristate: Alabama, Florida, Georgia
Tristate: Alabama, Georgia, Tennessee
```

The **findTwoNeighborsLargestPop**() method should find the two states that are neighbors and have a total population largest than any other pair of neighboring states. This method should return a **tuple** of two items. The first item is a concatenation of the two state names, separated by comma, and the second item is their total population.

Defining the **__str__()** method for this class is optional. You may omit it. However, this method could help you visualize your data when you develop your script

The Neighbors Class

Notice that this is a subclass of **dict**. Therefore, this class is essentially a dictionary. The key is an individual state name, and the value is the list of neighbors of that state. This dictionary is populated by the constructor of the Neighbors class, which reads the text lines from the us_states_adjacency.py file, splits them by comma, strips the surrounding spaces, and in the end stores the data in the dictionary.

Defining the **__str__()** method for this class is optional. It could help you if you define it, as noted above.

The `if __name__ == "__main__"` block

This is the entry point of our script, so it's very important to understand exactly what these statements do. The first statement creates an instance of the Neighbors class. This instance will be essentially a dictionary populated with data from the **us_states_adjacency.txt** file, as described above. The next statement creates an instance of the **StateList** class. This will be essentially a list of instances of the **State** class, each instance corresponding to a US State, as described above. We need to pass the neighbors object to the **StateList** constructor because the neighboring information for each state is needed when we create **State** objects in that constructor.

The **sList** object that we create here contains all data for all states that we need in this script. The subsequent statements in our script just call some methods of the **sList** object to print some reports based on the data. We already presented all these methods above, so we need not go into details here.

Remember that you are not allowed to add/delete/change any statements in the main if block. Your job in this lab is to provide appropriate implementations (statements) for the class methods that are called from this if block, such that, when we run the script, it will generate all the expected reports formatted appropriately.

If all this seems a bit overwhelming at first, don't panic. Take a little break and come back to it later. It would be great for you to work on this lab every day. This is your first object-oriented application. Developing such applications requires you to think in a specific way, which has a learning curve. Remember that all class methods are just tools at your disposal. They do what you tell them to do, and only when you call them. Each method is designed to perform a well-defined task, and should be as independent as possible from the other methods. Do not try to make the methods do more things that they are required to do. Keep them short and easy to understand.

The structure of object-oriented applications seems large because it does much more than just help you solve the problem at hand. It actually models your view on a little part of the world. The model is relevant to your current problem (Lab 4) and to many other similar, and often more difficult, problems that you might want to solve in the future.