Timothy Davison

5/6/2018

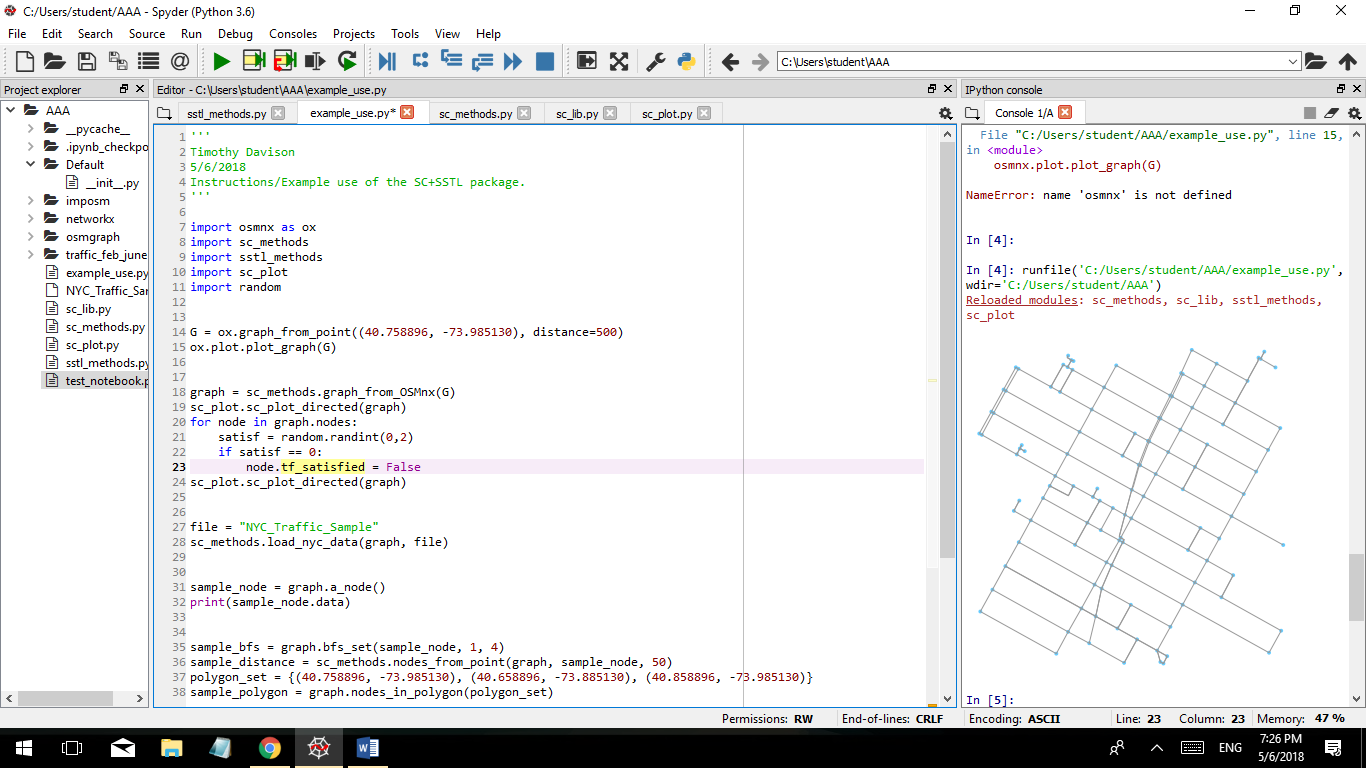
**Instructions for and Examples of Usage of SC/SSTL**

This document was produced for Meiyi Ma et al. to explain and introduce code written for the SC/SSTL project at the University of Virginia. It will seek to explain the usage of that code beginning with the downloading of OpenStreetMap data, continuing through the generation of a custom SC/SSTL graph, then through the loading of data into that graph, and finally to the usage of basic model checking algorithms to be built upon and customized in the future. The files are written in Python 3.6.0 and require the following packages:

* [Networkx](https://networkx.github.io/)
* [OSMnx](https://github.com/gboeing/osmnx)
* [Geoplotlib](https://github.com/andrea-cuttone/geoplotlib)
* [Pandas](https://pandas.pydata.org/)

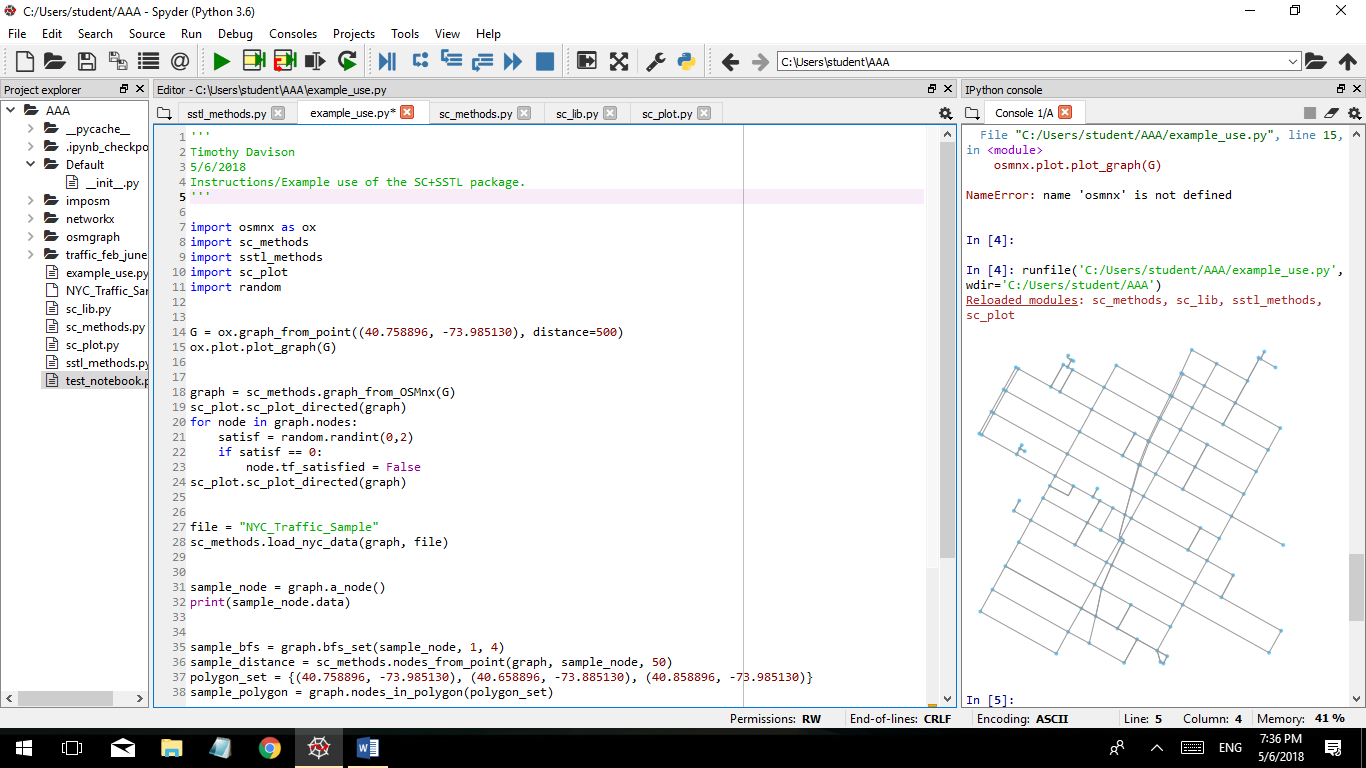
**Part I – OSM to Plotting the Custom Graph**

OSMnx is a tool which takes OpenStreetMap data (extracted based on a distance from a specified point, the name of a city/locality, or through some other possible area-definition methods) and converts those OpenStreetMap nodes, ways, and relations to a Networkx graph. It also provides convenient plotting methods for visualization. The following code snipped will generate such a graph and plot it in the user’s console:



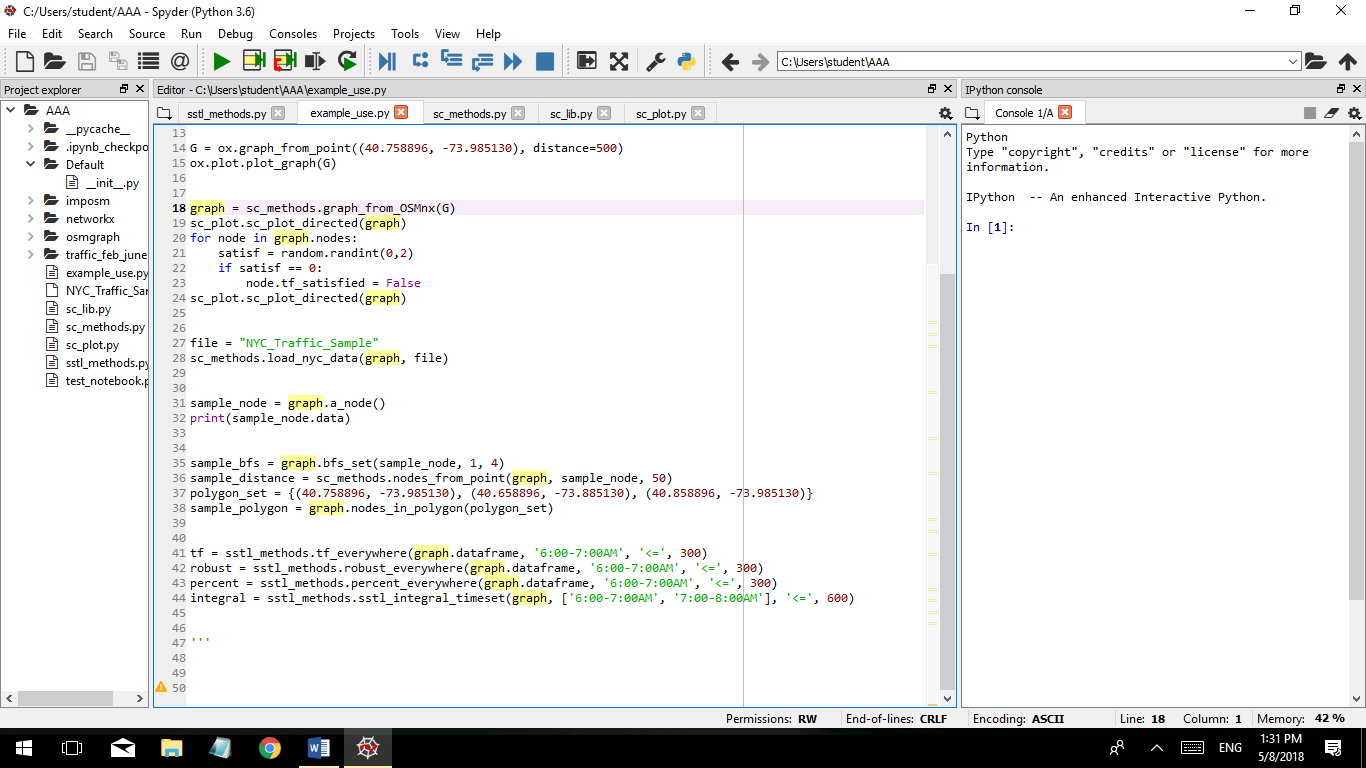
Once downloaded into this Networkx format, our tool converts that graph into a custom graph class from our library. Most notably, the new graph saves streets (which were previously represented as edges) as nodes, and intersections (which were previously represented as nodes) as edges. This mapping is one-to-one, with the result that multiple blocks along a street are saved as distinct nodes (provided they were previously saved as distinct edges).

The following snippet of code uses our own function and set of node/edge/graph classes to generate a graph from the previous Networkx graph. Also below is code used for visualization of our graph; note that in the first graph, all nodes are green. This is used to represent satisfaction at each node (something we’ll address later). In the second graph, the nodes are a mixture of green and red; these colors represent nodes with green as satisfied and red as unsatisfied. The code in between the two plot statements randomly assigns this attribute of satisfaction.



**II) Retrieving Subsets and Target Areas of the Graph**

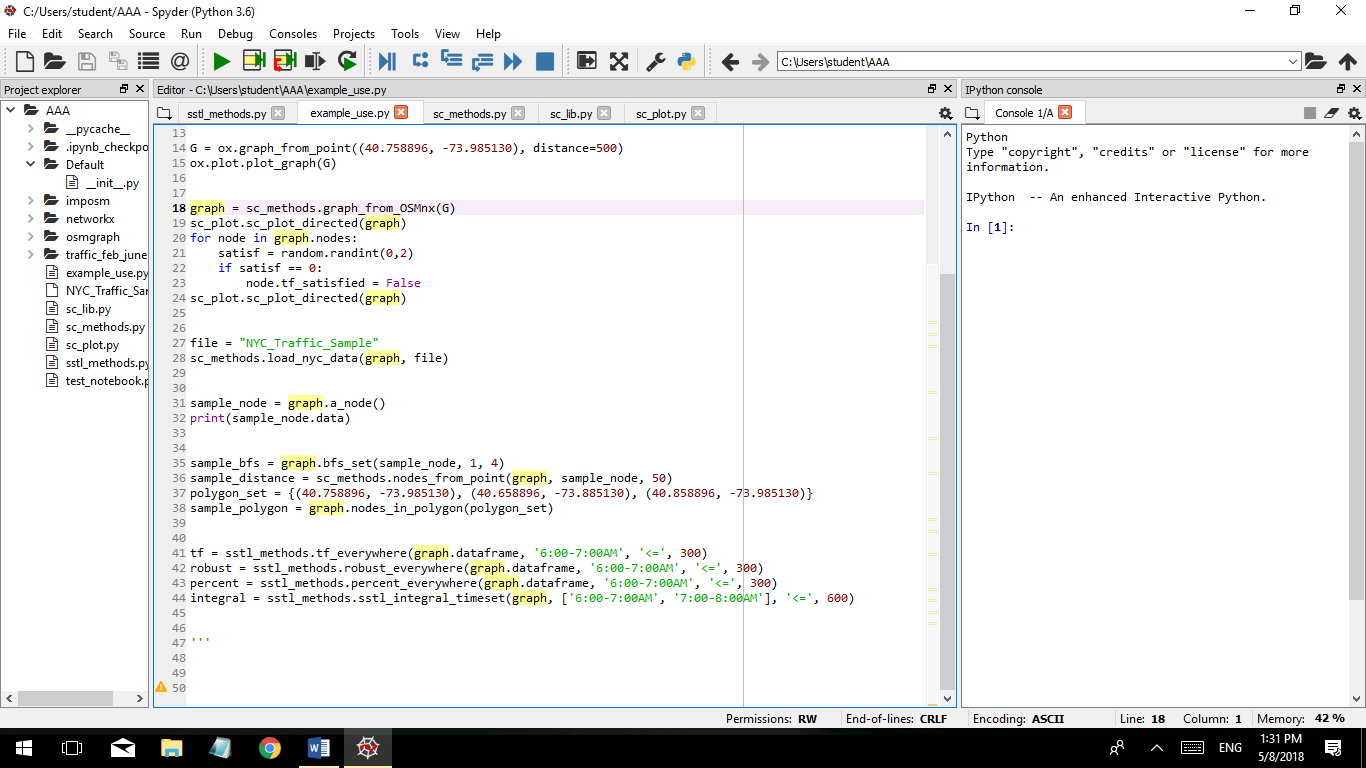
Spatial requirements necessitate the drawing up of target areas and subsets of the graph. One can retrieve subsets by calling up classes from the attribute dictionary of the graph class (ie, all nodes marked as being in the “School District” set), or by using one of the following functions:



Sample\_bfs follows the breadth first search algorithm and returns a set of all nodes within (i,j) steps of a given root node. Sample\_distance returns a set of all nodes within some Euclidian distance of a given root node. Finally, Sample\_polygon takes a set of nodes or points – three minimum – and returns a set of all nodes which lie within the polygon drawn from those points using the convex hull algorithm.

**III) Loading and Storing Data**

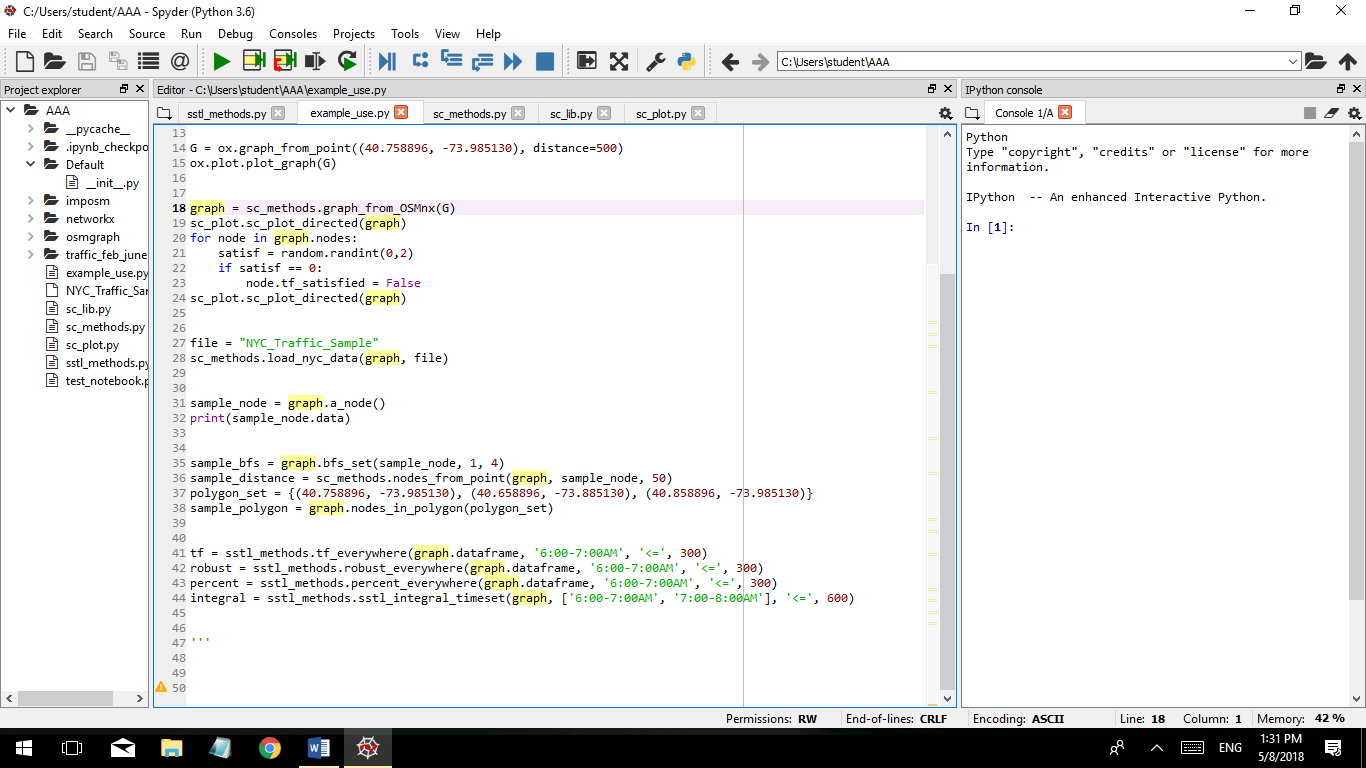
One challenge this project must face is that there exists no uniform method for storing smart city data. Therefore, one will have to customize the method for loading data into the graph based on how that data is originally stored; in the case of our NYC OpenData, we had a CSV of traffic data based on hours of the day as columns. The load\_nyc\_data method below breaks up that CSV and matches the data to a node based on the name of the given street. A more sophisticated dataset might have data specified with latitude and longitude coordinates, allowing a more exact mapping of data. The method below loads the data from a csv file called “NYC\_Traffic\_Sample” into the graph:



The graph itself stores data using pandas dataframes; each node has its own dataframe to hold all relevant information, and the graph itself has a composite dataframe of the data of all its nodes.

**IV) Model Checking**

Once data is available in the graph, one can begin checking for requirement satisfaction. There currently exist four methods of checking a requirement: true/false satisfaction, robustness (which gives either True, or the largest instance of violation), percent, and integral. The usage of these methods is demonstrated below, according to the spatial qualifier “Everywhere”:



In the first three instances, the requirement being checked is whether traffic on all streets between 6:00 and 7:00 AM is less than or equal to 300 cars. Similar methods exist for Somewhere, Eventually, and Always; note that the code for these methods are largely the same, but require different datasets (with rows as different times in a given temporal check, and rows as given locations in a spatial check). To combine them (for instance, to check an “Everywhere, Always”) one must first extract the target location (using one of the graph subset methods of part III) and check the threshold satisfaction over the target timeframe.

**V) Going Forward**

In order to use the provided code well, one must adapt the function loading data into the graph for unique data formats depending upon what is available. Other work which remains to be done includes:

* Develop an interface for storing and parsing requirements from English statements to target location/time, target data frame field, operator, and threshold;
* Develop methods which combine the base-level model checking requirements provided here;
* Develop an algorithm for efficient model checking over a system (brute force isn’t practical in the real world);

It has been a pleasure to work on this project, and I wish you the best of luck!!