In this project, we created a Truss Class. A truss is defined as a rigid figure made up of beams that are joined by frictionless pins. We are initially given a set of known variables - beam orientation, joints orientation, application of external forces, and whether or not a joint is connected to a fixed support. We desire to attain information regarding the compression force for each beam as well the reaction force on the beams due to fixed supports. In order to solve for these variables, we create a matrix in the form Ax = b, where A is our data matrix of known beam coefficient variables separated into components, x is our unknowns, and b is our vector of external forces. Using this system of linear equations, we are able to solve for compression and reaction forces, gaining valuable knowledge regarding the orientation of the truss. This project would be useful to someone analyzing the forces necessary for a truss to stay stable.

In our init class, we initialize all class variables necessary in the program. The init method takes as inputs a joints file, a beams file, and an optional output file. The optional output file is the location where the program saves the image of the truss generated by our (later discussed) method, plotgeometry. The init class then stores each file, and initializes variables and containers used throughout the program. Of these variables and containers, it initializes a beams dictionary (which matches beam number to the joint indicated for the beam) and a joints dictionary (which stores the following attributes: x,y coordinates of the joints, Fx and Fy components of the external force (if any) acting on the joint, and an indicator stating whether the joint is attached to a fixed support). Additionally, it stores a counter which maintains the column index needed in the matrix to append our fixed support column, two lists (bxcoeff, and bycoeff) which stores our decomposed x and y coordinates for our beam force coefficients, and a forces list which stores the result vector. The bx/by coeffs list will be utilized as known values in our data matrix. Lastly, the init method invokes the runfunctions method, which runs all the necessary functions to find the compression forces for each beam and potentially plot the truss if desired.

Next, the getdata method reads beams and joints files for a particular truss. It utilizes as an input the beams and joints file initialized in the init method. It then parses the files to get data for each beam and joint. The method adds tuples of joint indicies as values to a corresponding beam number key. Additionally, the method adds tuples of x and y joint coordinates, x and y external force components per joint, and an indicator for whether or not the joint is attached to a fixed support, as values to the cooresponding joint number in the joints dictionary. The method also keeps track of the column indicies for the reactionary forces in the data matrix.

Following, we have the getbeamvalues method, which calculates the coefficients for the components of force for each beam in the linear system. This method takes in the beam dictionary, and appends x and y coefficients for each beam to their corresponding container. X components are appended to bxcoeffs and y components are appended to bycoeffs.

Next, and our meatiest method, is our create matrix. This method creates a CSR data matrix which contains the linear equations necessary to solve our system of equations. The motivation for using a CSR matrix is to save on

storage and to compress the data matrix in a way that unnecessary 0 values are not stored. This is a great storage technique for sparse matricies, as the computer avoids uneccesary storage allocation to the 0s of a larger matrix. We will define our matrix as a matrix with beam numbers and reactionary component indicators as the columns, and joints are the rows (split up into a row for each x and y component). As this method iterates through the joints, it locates the beam coefficient for that particular component of the joint, and stores it as a value in the sparse matrix. Additionally, the method adds a 1 for each x and y component of a joint with a fixed point. This adds the necessary parameters adequately define our linear system. While iterating through joints, I create a corresponding force vector, which stores the components of the external forces acting on the joints. After the iteration, I am left with the proper Ax = b format to solve the system of linear equations. A is our (sparse) data matrix, x is our vector ofunknowns, and b is our force vector. After the iteration, I am able to apply the sparse linear package solver from scipy to solve the system of linear equations. If the system is over/under determined, meaning there are a different number of equations and unknown, I throw a runtime error. Additionally, if the matrix is singular, I throw a runtime error. The final output of this method is a result force vector, which is the solution to the linear system and which gives the compression forces for each beam.

Following, we have the optionally invoked plot geometry method. This method is only invoked if the user provides an output file for the image. The input is the filename for the saved image. The method then utilizes the matplotlib module to plot the geometry of the plot based off of the orientation for the joints for each beam. The image is then stored and saved in the output file.

Additionally, we have the repr method, which returns the string to be printed as invoked by main.py. This method uses the class variables as an input, and returns a string with the desired output. The properly formatted beam number and force values appear in a tabular format.

Lastly, we have the runfunctions method, which is invoked by the init method. This method runs the necessary functions to obtain beam forces and plot truss geometry (if applicable). The plot geometry function only runs when the user provides the output file for the image.

And those are all my methods! Thanks for reading :).