**Advanced Programming Summative Assessment Report**

**Section 1**

a) #function to remove inactive seats

def removeInactive():

global merged\_CSV

indexInactive = merged\_CSV[merged\_CSV["PROGRAM STATUS"] == "INACTIVE"].index

merged\_CSV\_temp = merged\_CSV.drop(indexInactive, inplace=False)

merged\_CSV = merged\_CSV\_temp

messagebox.showinfo("Attention", "Establishments with inactive programs removed.")

merged\_CSV.to\_csv("chosen csv name.csv", index=False, sep=',')

#function to generate seating column

def extractSeating():

global merged\_CSV

merged\_CSV["Seating Type"] = merged\_CSV["PE DESCRIPTION"].str.extract('(\(.+\))', expand=True)

merged\_CSV["PE DESCRIPTION"] = merged\_CSV["PE DESCRIPTION"].str.replace("(\s\(.+\)\s)", " ", regex=True)

messagebox.showinfo("Attention", "Seating Arrangements extracted into new column.")

#extract year of inspection for calculating mean median mode

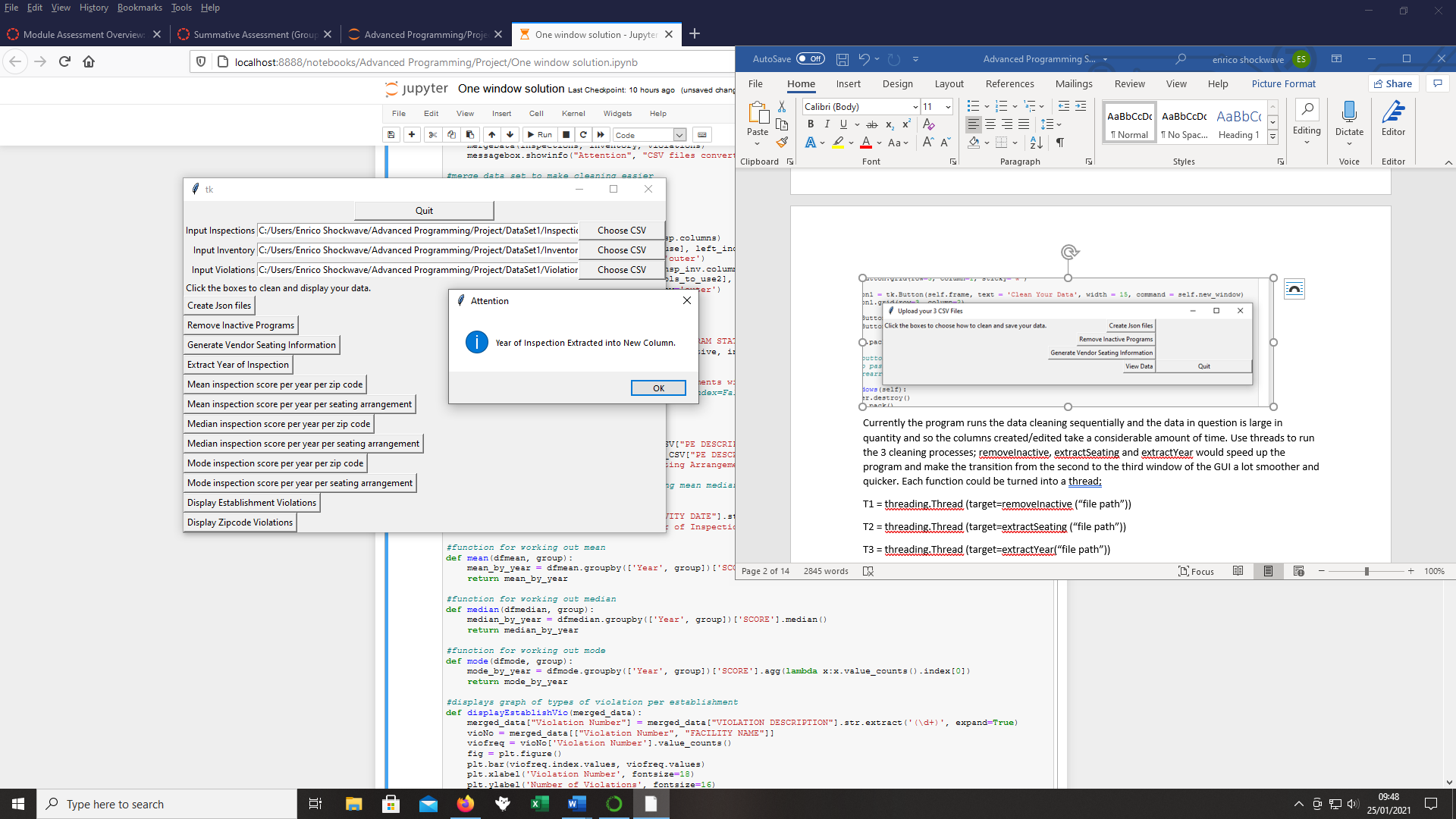
def extractYear():

global merged\_CSV

merged\_CSV["Year"] = merged\_CSV["ACTIVITY DATE"].str.split("/").str[2]

messagebox.showinfo("Attention", "Year of Inspection Extracted into New Column.")

The methods associated with the buttons for cleaning the data could be redesigned to use python threads.

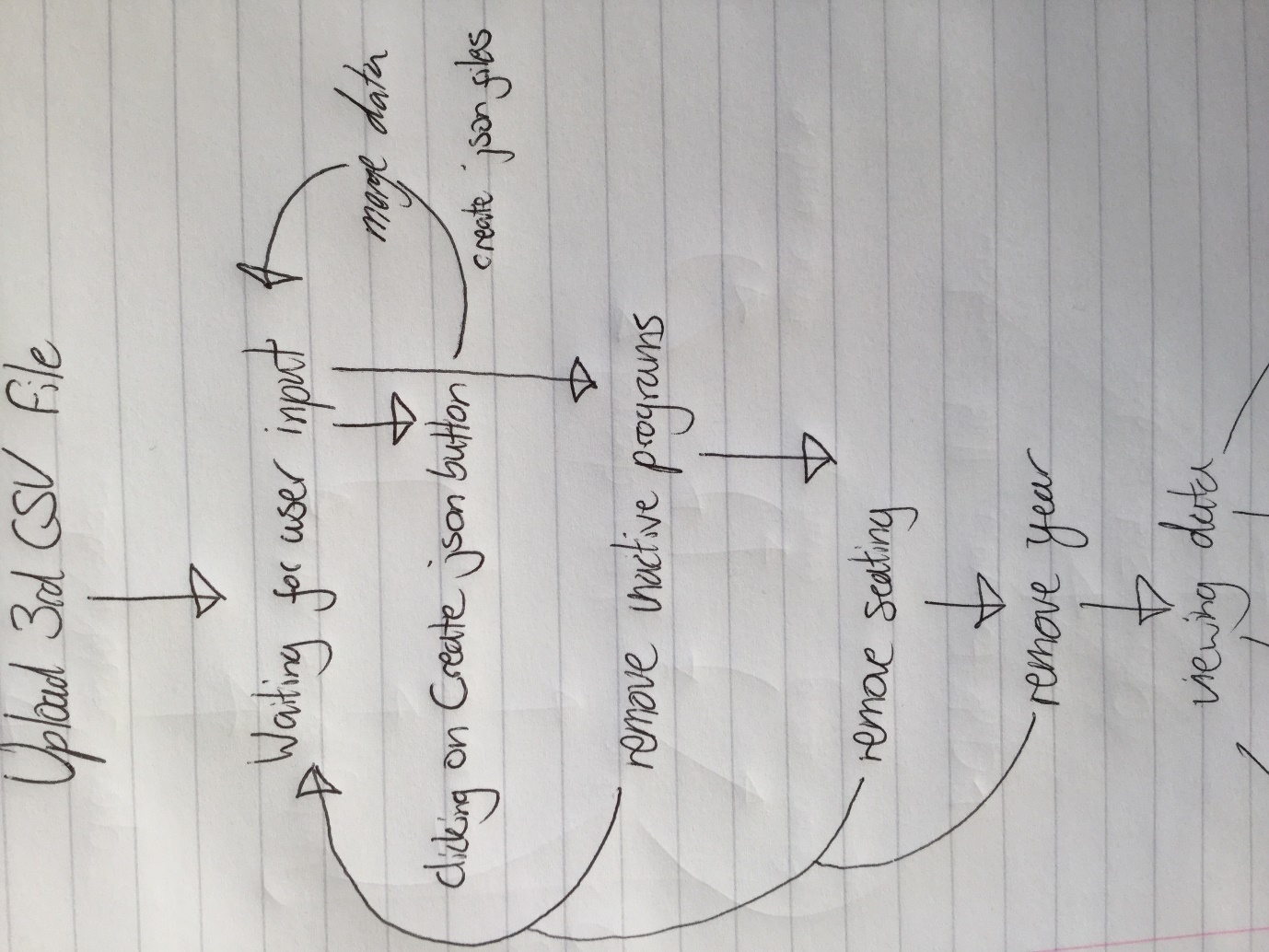


Currently the program runs the data cleaning sequentially and the data in question is large in quantity and so the columns created/edited take a considerable amount of time. Use threads to run the 3 cleaning processes; removeInactive, extractSeating and extractYear would speed up the program and make the transition from the cleaning to the data to viewing the data smoother and quicker. Additionally, only 1 pop up box would be required to inform the user the function has finished rather than 3. Each function could be turned into a thread;

T1 = threading.Thread (target=removeInactive (“file path”))

T2 = threading.Thread (target=extractSeating (“file path”))

T3 = threading.Thread (target=extractYear(“file path”))

These could all be simultaneously initiated with the click of one button instead of three and once completed automatically open up another GUI window that displays the data analysis. They could run concurrently because each function affects different columns in the data. I think one issue that could arise would be the data being overwritten if one function finishes before the others and the new columns created would not save so making sure that everything was kept and nothing lost during the cleaning process would be the main concern with threading in this way. Another possible issue would be that because they are all writing to new columns, the columns could end up being blended into 1 rather than 2 (seating arrangements and year of inspection) which would cause problems with the data analysis aspects of the program.

You can see in the centre of the flow diagram that the 3 processes run separately and all need to be applied before the user has their required data. Incorporating these to run with one user interaction instead of 3 makes for a more friendly user interaction and would probably reduce the amount of time taken compared to running the 3 separately.

b)

entry\_csv3=tk.Entry(self.frame, text="", width=70)

entry\_csv3.grid(row=2, column=1, sticky="w")

self.csvlabel1 = tk.Label(self.frame, text="Input Inspections")

self.csvlabel1.grid(row=0, column=0 ,sticky="e")

self.csvbutton1 = tk.Button(self.frame, text="Choose CSV", width=15, command=lambda:get\_csv(entry\_csv1))

self.csvbutton1.grid(row=0, column=2, sticky="w")

self.csvlabel2 = tk.Label(self.frame, text="Input Inventory")

self.csvlabel2.grid(row=1, column=0 ,sticky="e")

self.csvbutton2 = tk.Button(self.frame, text="Choose CSV", width=15, command=lambda:get\_csv(entry\_csv2))

self.csvbutton2.grid(row=1, column=2, sticky="w")

self.csvlabel3 = tk.Label(self.frame, text="Input Violations")

self.csvlabel3.grid(row=2, column=0 ,sticky="e")

self.csvbutton3 = tk.Button(self.frame, text="Choose CSV", width=15, command=lambda:get\_csv(entry\_csv3))

self.csvbutton3.grid(row=2, column=2, sticky="w")

I used a combination of labels, entry and button widgets to create the GUI where the user uploads their csv files. In the first row I have text labels that show what name CSV file I would like the user to upload (although in reality the order does not have any bearing on the program) and next to it an entry widget in column 1 that is empty to begin with and displays the file path of the selected file. To select the file there is a button in column 2 that is clearly defined to choose their CSV. The natural eye movement of left to right makes this process straightforward for the user. In column 3 there is a save button where the user can click to back up a copy of their csv file. This is repeated for 3 rows corresponding to the amount of csv files required. Below the buttons, there is a big, slightly raised button that moves on to the next screen titled clean your data. This is the only button on the row to emphasise that it should be the last button to be pressed and had slight padding to further show that this will function differently compared to the others. The layout of the buttons and text widgets in uniform columns presents a logical pattern for the user to follow and hopefully makes the first window of the GUI intuitive and easy to work out how to use. This follows on for the data cleaning and displaying aspects of the program as well.

c) The use of lambda functions in python allows for code reuse much more fluently than my experience with java. When this is combined with some of the data analysis tools that pandas offers there seems to be a huge gap in terms of speed of development compared to what would be required to produce something comparable in java.

#function for working out mean

def mean(dfmean, group):

mean\_by\_year = dfmean.groupby(['Year', group])['SCORE'].mean()

return mean\_by\_year

meanZipButton = tk.Button(self.frame, text="Mean inspection score per year per zip code",

command=lambda:clicker(mean(merged\_CSV, 'Zip Codes'),

"Mean Inspection Score Per Year by Zip Code"))

meanZipButton.grid(row=9, column=0, columnspan=2, sticky='w')

meanSeatButton = tk.Button(self.frame, text="Mean inspection score per year per seating arrangement",

command=lambda: clicker(mean(merged\_CSV, 'Seating Type'),

In the above code I have reused the function that generates the mean value for each year based on 2 distinct criteria (seating number and zip code). I am unaware of any quick and concise methods for analysis data in this way in java so comparing python and java regarding data analytics tools, python makes designing a program with that functionality in mind more straightforward and concise, and therefore speedy. This was the same case with the functions for median and mode. Additionality I was able to use a lambda function to display this data in a separate window when the relevant button was clicked. There are 6 buttons that the user has the options to press and so using only 1 function for 6 buttons naturally speeds up the development of the program compared to what would be required in java.

def extractSeating():

global merged\_CSV

merged\_CSV["Seating Type"] = merged\_CSV["PE DESCRIPTION"].str.extract('(\(.+\))', expand=True)

merged\_CSV["PE DESCRIPTION"] = merged\_CSV["PE DESCRIPTION"].str.replace("(\s\(.+\)\s)", " ", regex=True)

messagebox.showinfo("Attention", "Seating Arrangements extracted into new column.")

Lastly, the regex module made creating new columns from the existing data incredibly easy once I found a formula that matched the data needing to be extracted. Combining this with column selection from pandas seems to be far simpler and concise when compared to doing the same in java.

**Section 2**

1. I decided to converted the CSV files into json files because the sample data given was quite large and JSON is a better format for storing large data sets when compared to XML since it is more compact. The program takes some time working through that data set as it is and so using XML would have made this more of an issue. The data is readily accessible when compared to XML which was another pro of using JSON. One consideration of using XML was that JSON provides no namespace support. However, since the brief outlined that the input data would be of the same form as the sample data used then there was little need to make it so that saved files could be written and read by multiple sources. The lack of a validating schema seems to be the only disadvantage to converting the CSV files to JSON.

#convert csv file to json

def convertJson():

inspections.to\_json("Inspections.json")

inventory.to\_json("Inventory.json")

violations.to\_json("Violations.json")

mergeData(inspections, inventory, violations)

messagebox.showinfo("Attention", "CSV files converted to JSON files and data merged.")

#extract year of inspection for calculating mean median mode

def extractYear():

global merged\_CSV

merged\_CSV["Year"] = merged\_CSV["ACTIVITY DATE"].str.split("/").str[2]

messagebox.showinfo("Attention", "Year of Inspection Extracted into New Column.")

#function for working out mode

def mode(dfmode, group):

pd.set\_option('display.max\_rows', None)

mode\_by\_year = dfmode.groupby(['Year', group])['SCORE'].agg(lambda x:x.value\_counts().index[0])

return mode\_by\_year

def clicker(function, title):

global pop

pop = tk.Toplevel(root)

pop.title(title)

pop.geometry("800x400")

pop\_label = tk.Label(pop, text=function)

pop\_label.grid(row=0, column=0)

I merged the data based on overlapping columns into one dataframe to make the analysis easier.

def mergeData(csv1, csv2, csv3):

global merged\_CSV

insp = csv1

inv = csv2

vio= csv3

cols\_to\_use=inv.columns.difference(insp.columns)

insp\_inv= pd.merge(insp, inv[cols\_to\_use], left\_index=True,

right\_index=True, how='outer')

cols\_to\_use2=vio.columns.difference(insp\_inv.columns)

merged\_CSV = pd.merge(insp\_inv, vio[cols\_to\_use2], left\_index=True,

right\_index=True, how='outer')

merged\_CSV.to\_csv("merged\_data.csv", index=False, sep=',')

I had to extract 2 new columns in order to be able to produce analysis for the mean, median and mode for the inspection score per year. The first was the year of inspection, as in the initial data set the data column included the month and day. I used a regex in order to copy the year of inspection into a new column that would act as in index for the analysis. I used another regex to slice the seating information into a new column leaving the other information in the PE DESCRIPTION column. I now had the necessary columns for the analysis, seating information, year of inspection, inspection score and zip code. I created functions for mean, median and mode that grouped the year and either the seating arrangements or zip code (which is left as a variable argument) against the inspection scores for that year. I created 6 buttons that corresponded to each method of analysis and used a lambda function to generate a pop up window that displayed the analysis depending on which button the user pressed. I passed either the seating information or zip code as an argument into this lambda function.

meanZipButton = tk.Button(self.frame, text="Mean inspection score per year per zip code",

command=lambda:clicker(mean(merged\_CSV, 'Zip Codes'),

"Mean Inspection Score Per Year by Zip Code"))

meanZipButton.grid(row=9, column=0, columnspan=2, sticky='w')

meanSeatButton = tk.Button(self.frame, text="Mean inspection score per year per seating arrangement",

command=lambda: clicker(mean(merged\_CSV, 'Seating Type'),

"Mean Inspection Score Per Year by Seating Type"))

c)

I used matplotlib to generate a bar graph which displayed the violation number on the x axis and the number of violations committed by each establishment on the y axis. Merging the data sets made analysis this more straightforward that comparing 3 separate data sheets. Violation number was already available so did not require any cleaning. There were roughly 50 types of violations and so I thought it would be better to put them on the x axis as the y axis would contain higher values. I used value.counts() to calculate how many times each violation had been committed and then converted it into a variable “viofreq”, then plotted the index of that on the x axis (which corresponded to the violation number) against the value of each violation on the y axis (which corresponded to the number of times that violation had been committed). I then linked a function to display this graph in a pop up window when the relevant button was clicked by the user.

def displayEstablishVio(merged\_data):

global fig

merged\_data["Violation Number"] = merged\_data["VIOLATION DESCRIPTION"].str.extract('(\d+)', expand=True)

vioNo = merged\_data[["Violation Number", "FACILITY NAME"]]

viofreq = vioNo['Violation Number'].value\_counts()

fig = plt.figure()

plt.bar(viofreq.index.values, viofreq.values)

plt.xlabel('Violation Number', fontsize=18)

plt.ylabel('Number of Violations', fontsize=16)

return fig

def graphClicker(function, title):

global pop

pop = tk.Toplevel(root)

pop.title(title)

pop.geometry("900x500")

canvas = FigureCanvasTkAgg(fig,

master = pop)

canvas.draw()

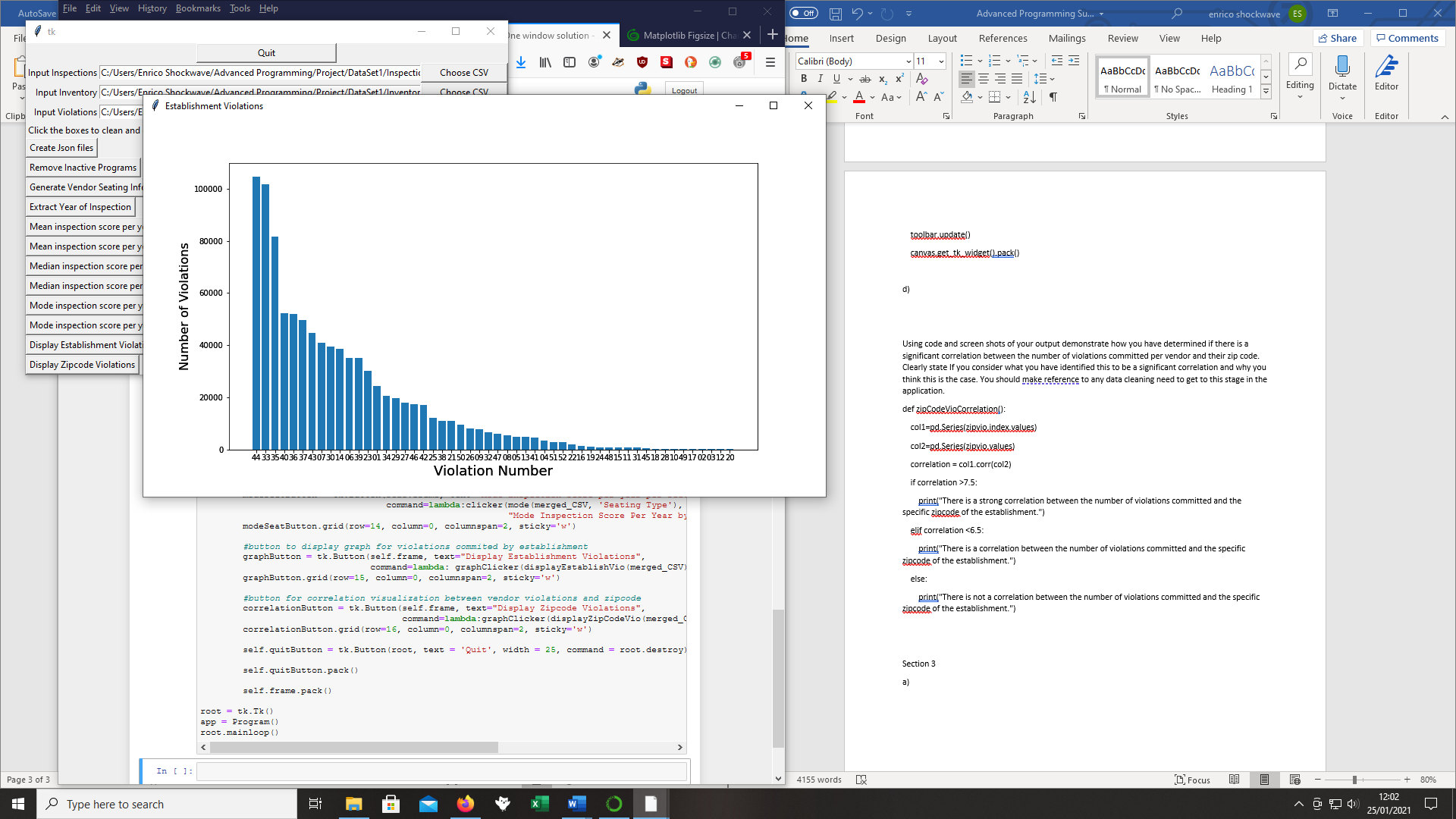
canvas.get\_tk\_widget().pack()

toolbar = NavigationToolbar2Tk(canvas,

pop)

toolbar.update()

canvas.get\_tk\_widget().pack()



d)

def displayZipCodeVio(merged\_data):

global fig

global zipvio

zipvio = merged\_data['Zip Codes'].value\_counts()

fig = plt.figure(figsize=(24,8))

xaxis = list(map(str, zipvio.index.values))

plt.bar(xaxis, zipvio.values)

plt.xlabel('Zip Codes', fontsize=18)

plt.ylabel('Number of Violations', fontsize=16)

zipCodeVioCorrelation()

return fig

def zipCodeVioCorrelation():

col1=pd.Series(zipvio.index.values)

col2=pd.Series(zipvio.values)

correlation = col1.corr(col2)

if correlation >7.5:

print("There is a strong correlation between the number of violations committed and the specific zipcode of the establishment.")

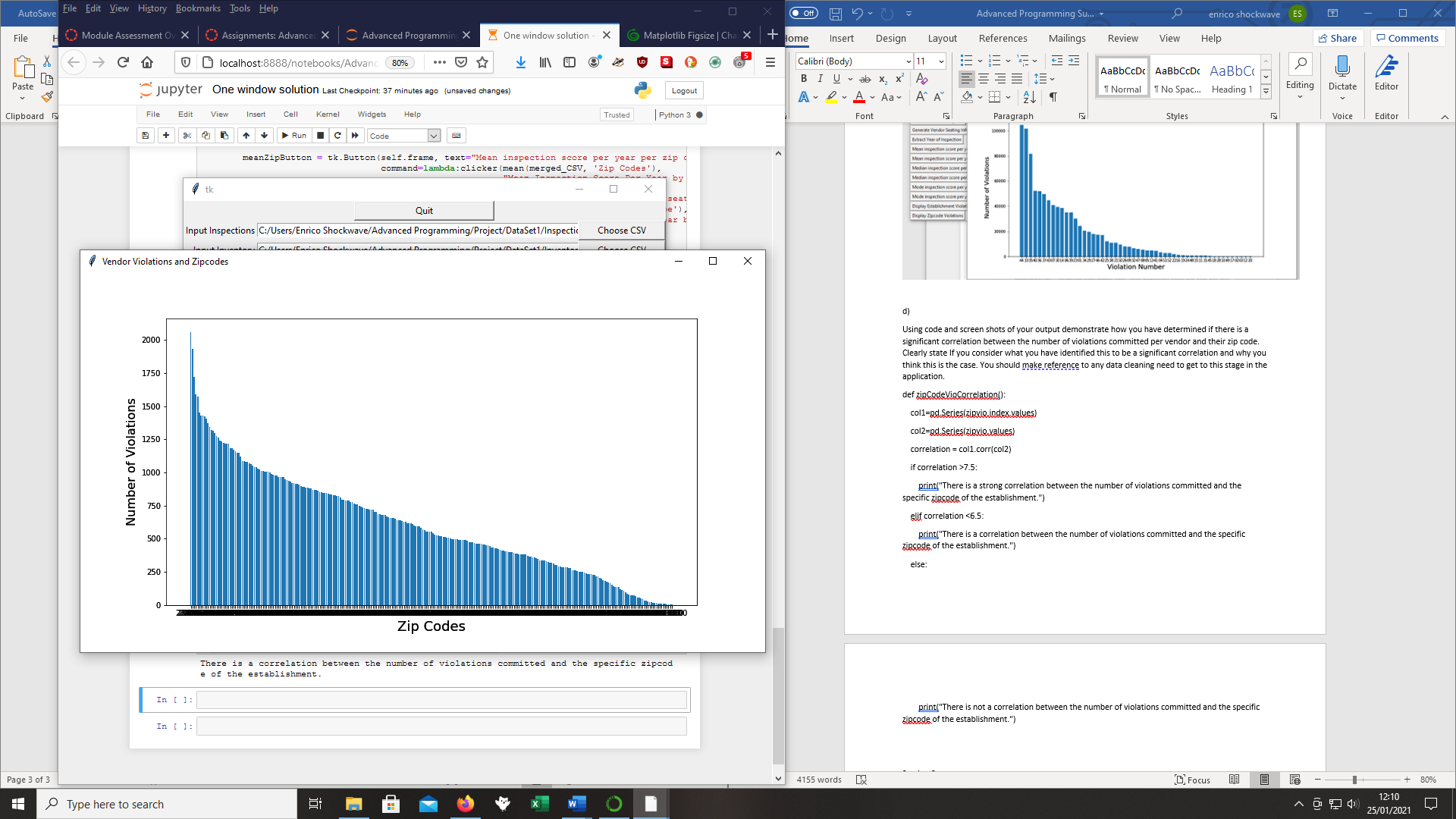
elif correlation <6.5:

print("There is a correlation between the number of violations committed and the specific zipcode of the establishment.")

else:

print("There is not a correlation between the number of violations committed and the specific zipcode of the establishment.")

I created a method that took the value.counts() of zip codes as the x axis (the zipvio variable) and then plotted them against the number of violations committed per zip code on the y axis. This resulted in the graph below. I then created another method that took the index of the zipvio variable and the values of those indexes and ran the corr function from pandas against both of them. The output to the console would depend on what number resulted from that calculation. If the coefficient was above 6.5 then the console would print out a message telling the user there is a relationship between zipcodes and the number of violations committed. If it was much higher then it would state a strong relationship. If it was under 6.5 then it would state there was no correlation between the two series.



**Section 3**

The use of data cleaning or data manipulation can have very tangible consequences in real world contexts. Programs more sophisticated than the one I attempted to build do similar functions regarding assessing the following of health and safety and safety protocols by staff in establishments serving food. As show with this line of code:

indexInactive = merged\_data[merged\_data["PROGRAM STATUS"] == "INACTIVE"].index

merged\_data.drop(indexInactive, inplace=True)

it is remarkably easy to remove certain columns of data from a file, and doing so can change the meaning of the data. It would be equally easy to remove a row of data, i.e. the performance of one or several specific establishments that could be operating well below expected hygiene standards. The omission of these low performers could lead to unsafe practices going unpunished and continuing indefinitely and putting the health of customers and patrons of the establishment in danger. From a legal perspective, it seems that if data is cleaned in such a way to make whom that data is about seem more favourable at the expense of another entity, whether that is a customer or a business partner, there should be grounds for deterring this practice. However it is difficult to state that there should be a legal penalty for doing so since data can manipulated in many different ways.

You can observe the first graph above and see the column marked 44 towers above all the rest, with the data stating “44. Floors, walls and ceilings: properly built, maintained in good repair and clean”. This could mean that all some 100000 counts of violation relate to walls and ceilings being damaged and a possible safety hazard, or it could simply mean that a tile on the floor is broken. It is difficult to say since data can be portrayed in all sorts of ways depending on who is manipulating it. Ethically, to obscure harmful information or omit it completely is obviously wrong. Conversely, if somebody is not deterred by this consideration, as I mentioned above, it is remarkable easy to change data to make it more suited to your needs if you know how to process it. When you look at the 1st graph for Violation 29, it does not look anywhere near as serious as 44, however 29 is Toxic substances properly identified, stored, used which would be a great concern for somewhere that processes food and serves it to the public. Ethically and legally, manipulating data bears a great deal of responsibility and so should be regulated accordingly.

References

[**McKinney W. (2017).** Python ***for Data Analysis. 2nd* ed. Springer. O'Reilly Media**](https://eu.alma.exlibrisgroup.com/leganto/public/44YORK_INST/citation/36793477170001381?auth=SAML)