Python – Data Visualisation

|  |
| --- |
| Python Packages for Data Science  *A Python library is a collection of functions and methods that allow you to perform lots of actions without writing any code. The libraries usually contain built in modules providing different functionalities which you can use directly. There are extensive libraries offering a broad range of facilities. Libraries are broadly divided in three groups.*  Scientifics computing libraries:   * + **Pandas:** offers *data structure and tools* for effective data manipulation and analysis. Offers data structure and tools for effective data manipulation and analysis. It provides fast access to structured data. The primary instrument of Pandas is a two-dimensional table consisting of columns and rows labels which are called a DataFrame. It is designed to provide an easy indexing function   + **Numpy:** Uses arrays as their inputs and outputs. It can be extended to objects for matrices, and with a little change of coding, developers perform fast array processing Arrays & Matrices   + **SciPy:** includes functions for some advanced math problems as listed here, as well as data visualization. Integrals, solving differential equations and optimisation   Visualization libraries: *These libraries enable you to create graphs, charts and maps*   * + **Matplotlib:** The Matplotlib package is the most well-known library for data visualization. The graphs are also highly customizable.   + **Seaborn:** Another high-level visualization library is Seaborn. it is based on Matplotlib. It's very easy to generate various plots such as heat maps, time series and violin plots.   Machine learning algorithms: we're able to develop a model using our data set and obtain predictions.   * + **Scikit-learn:** the Scikit-learn library contains tools statistical modelling, including regression, classification, clustering, and so on. This library is built on NumPy, SciPy and Matplotib.   + **Statsmodels:** is also a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. |
| Prepossessing of Data in Python  # Improtant working directory and function  import os  os.getcwd() # For further information check this link - <https://note.nkmk.me/en/python-os-getcwd-chdir/>  # Import pandas library  import numpy as np # useful for many scientific computing in Python  import pandas as pd # primary data structure library  # Read the online file by the URL provided, and assign it to variable "df"  path='https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/Canada.xlsx'  df\_can = pd.read\_excel(path,  sheet\_name='Canada by Citizenship',  skiprows=range(20),  skipfooter=2)  # Note: in order use read\_excel() method. Normally, we would need to download a xlrd module which pandas requires to read in excel files. You would need to run the following line of code to install the xlrd module:  !conda install -c anaconda xlrd –yes  # Let's view the rows of the dataset using the head() or tail() function.  df\_can.head() # View the Top 5 rows, You can specify the number of rows you'd like to see in brackets  df\_can.tail() # View the bottom 5 rows of the dataset  *When analyzing a dataset, it's always a good idea to start by getting basic information about your dataframe. We can do this by using the info() method along with other listed below.*  df\_can.info() # *getting basic information*  df\_can.columns.values # To get the list of column headers we can call upon the dataframe's .columns parameter.  df\_can.index.values # Similarly, to get the list of indicies we use the dataframe's.index parameter.  print(type(df\_can.columns))  print(type(df\_can.index))  *Note: The default type of index and columns is NOT list.*  # To get the index and columns as lists, we can use the tolist() method.  df\_can.columns.tolist()  df\_can.index.tolist()  ​print (type(df\_can.columns.tolist()))  print (type(df\_can.index.tolist()))  # To view the dimensions of the dataframe, we use the dataframe's.shape parameter.  df\_can.shape # size of dataframe (rows, columns)  *Note: The main types stored in pandas objects are float, int, bool, datetime64[ns] and datetime64[ns, tz], timedelta[ns], category and object (string). In addition, these dtypes have item sizes, e.g. int64 and int32.*  # Let's clean the data set to remove a few unnecessary columns. We can use pandas drop() method as follows:  # In pandas axis=0 represents rows (default) and axis=1 represents columns.  df\_can.drop(['AREA','REG','DEV','Type','Coverage'], axis=1, inplace=True)  df\_can.head(2)  # Let's rename the columns using rename() method by passing in a dictionary of old and new names as follows:  df\_can.rename(columns={'OdName':'Country', 'AreaName':'Continent', 'RegName':'Region'}, inplace=True)  df\_can.columns  # Let’s add a 'Total' column that sums up the total immigrants by country over the entire period 1980 - 2013:  df\_can['Total'] = df\_can.sum(axis=1)  # We can check to see how many null objects we have in the dataset as follows:  df\_can.isnull().sum()  # Finally, let's view a quick summary of each column in our dataframe using the describe() method.  df\_can.describe()  *Before we proceed, notice that the default index of the dataset is a numeric range from 0 to 194. This makes it difficult to do a query by a specific country. For example, to search for data on Japan, we need to know the corresponding index value.*  # This can be fixed very easily by setting the 'Country' column as the index using set\_index() method.  df\_can.set\_index('Country', inplace=True) # The opposite of set is reset. So, to reset the index use df\_can.reset\_index()  df\_can.head(3)  *Note: Column names that are integers (such as the years) might introduce some confusion. For example, when we are referencing the year 2013, one might confuse that when the 2013th positional index.*  # To avoid this ambiguity, let's convert the column names into strings: '1980' to '2013'.  df\_can.columns = list(map(str, df\_can.columns))  # [print (type(x)) for x in df\_can.columns.values] #<-- uncomment to check type of column headers  # Since we have the years column label as string, let's declare a variable to easily call upon the full range of years:  years = list(map(str, range(1980, 2014))) # We will Use it for plotting later on  years  *To filter the dataframe based on a condition, we simply pass the condition as a Boolean vector. For example, let's filter the dataframe to show the data on Asian countries (AreaName = Asia).*   * # Create the condition Boolean series   condition = df\_can['Continent'] == 'Asia'   * # Pass this condition into the dataFrame   df\_can[condition]  *We can also pass multiple criteria in the same line.*  # Let's filter for AreaNAme = Asia and RegName = Southern Asia  ​df\_can[(df\_can['Continent']=='Asia') & (df\_can['Region']=='Southern Asia')]  ​  *Note: When using 'and' and 'or' operators, pandas requires we use '&' and '|' instead of 'and' and 'or', and don't forget to enclose the two conditions in parentheses.*  *Before we proceed, Let's review the changes we have made to our dataframe.*  print('data dimensions:', df\_can.shape)  print(df\_can.columns) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visualizing Data using Matplotlib  *The primary plotting library we will explore in the course is Matplotlib.*  *Matplotlib: Standard Python Visualization Library*  *Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.*  *If you are aspiring to create impactful visualization with python, Matplotlib is an essential tool to have at your disposal.*  Matplotlib.Pyplot  *One of the core aspects of Matplotlib is matplotlib.pyplot. It is Matplotlib's scripting layer which we studied in details in the training videos. Recall that it is a collection of command style functions that make Matplotlib work like MATLAB*.  *Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In this lab, we will work with the scripting layer to learn how to generate line plots*.  *In future labs, we will get to work with the Artist layer as well to learn how it differs from the scripting layer.*  # Let's start by importing Matplotlib and Matplotlib.pyplot as follows:  %matplotlib inline # we are using the inline backend  import matplotlib as mpl  import matplotlib.pyplot as plt  print ('Matplotlib version: ', mpl.\_\_version\_\_) # Check if Matplotlib is loaded  Matplotlib version: 3.1.1  # Apply a style to Matplotlib  print(plt.style.available)  ['\_classic\_test', 'seaborn-deep', 'seaborn-poster', 'seaborn-whitegrid', 'seaborn-paper', 'seaborn-pastel', 'seaborn-white', 'bmh', 'seaborn', 'ggplot', 'grayscale', ..]  mpl.style.use(['ggplot']) # optional: for ggplot-like style  Plotting in pandas  *Fortunately, pandas has a built-in implementation of Matplotlib that we can use. Plotting in pandas is as simple as appending a .plot() method to a series or dataframe.*  Documentation:  Plotting with Series: <http://pandas.pydata.org/pandas-docs/stable/api.html#plotting>  Plotting with Dataframes: <http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-plotting>  # Line Pots (Series/Dataframe)  *What is a line plot and why use it?*  *A line chart or line plot is a type of plot which displays information as a series of data points called 'markers' connected by straight line segments. It is a basic type of chart common in many fields. Use line plot when you have a continuous data set. These are best suited for trend-based visualizations of data over a period of time.*  *Let's start with a case study*: *In 2010, Haiti suffered a catastrophic magnitude 7.0 earthquake. The quake caused widespread devastation and loss of life and aout three million people were affected by this natural disaster. As part of Canada's humanitarian effort, the Government of Canada stepped up its effort in accepting refugees from Haiti. We can quickly visualize this effort using a Line plot:*  # Plot a line graph of immigration from Haiti using df.plot().   * First, we will extract the data series for Haiti.   years = list(map(str, range(1980, 2014)))  # passing in years 1980 - 2013 to exclude the 'total' column  haiti = df\_can.loc['Haiti', years]  haiti.head()   * Next, we will plot a line plot by appending .plot() to the haiti dataframe.   haiti.plot()  *pandas automatically populated the x-axis with the index values (years), and the y-axis with the column values (population). However, notice how the years were not displayed because they are of type string. Therefore, let's change the type of the index values to integer for plotting.*  # Also, let's label the x and y axis using plt.title(), plt.ylabel(), and plt.xlabel() as follows:    haiti.index=haiti.index.map(int) #change the index values to type integer  haiti.plot(kind='line')  ​plt.title('Immigration from Haiti')  plt.ylabel('Number of immigrants')  plt.xlabel('Years')  ​plt.show() # to show the updates made to the figure  *We can clearly notice how number of immigrants from Haiti spiked up from 2010 as Canada stepped up its efforts to accept refugees from Haiti. Let's annotate this spike in the plot by using the plt.text() method.*    haiti.plot(kind='line')  ​plt.title('Immigration from Haiti')  plt.ylabel('Number of Immigrants')  plt.xlabel('Years')  ​# annotate the 2010 Earthquake.  # syntax: plt.text(x, y, label)  plt.text(2000, 6000, '2010 Earthquake') *#* *See note below*  ​plt.show() # With just a few lines of code, you were able to quickly identify and visualize the spike in immigration!  *Quick note on x and y values in plt.text(x, y, label):*   * Since the x-axis (years) is type 'integer', we specified x as a year. The y axis (number of immigrants) is type 'integer', so we can just specify the value y = 6000.   plt.text(2000, 6000, '2010 Earthquake') # if years stored as type int   * If the years were stored as type 'string', we would need to specify x as the index position of the year. Eg 20th index is year 2000 since it is the 20th year with a base year of 1980.   plt.text(20, 6000, '2010 Earthquake') # if years stored as type str  *We will cover advanced annotation methods in later modules.*  *We can easily add more countries to line plot to make meaningful comparisons immigration from different countries.*  # Let's compare the number of immigrants from India and China from 1980 to 2013.   * Step 1: Get the data set for China and India, and display dataframe.   df\_CI = df\_can.loc[['India','China'], years] # passing in years 1980 - 2013  df\_CI.head()   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | 1980 | 1981 | 1982 | 1983 | 1984 | … | | India | 8880 | 8670 | 8147 | 7338 | 5704 | … | | China | 5123 | 6682 | 3308 | 1863 | 1527 | … |   2 rows × 34 columns   * Step 2: Plot graph. We will explicitly specify line plot by passing in kind parameter to plot().   df\_CI.plot(kind='line')  *That doesn't look right...*  *Recall that pandas plots the indices on the x-axis and the columns as individual lines on the y-axis. Since df\_CI is a dataframe with the country as the index and years as the columns, we must first transpose the dataframe using transpose() method to swap the row and columns.*  df\_CI = df\_CI.transpose()  df\_CI.head()   |  |  |  | | --- | --- | --- | |  | India | China | | 1980 | 8880 | 5123 | | 1981 | 8670 | 6682 | | 1982 | 8147 | 3308 | | 1983 | 7338 | 1863 | | 1984 | 5704 | 1527 | | …a | …x | …y |   *Go ahead and plot the new transposed dataframe. pandas will auomatically graph the two countries on the same graph, Make sure to add a title to the plot and label the axes.*    df\_CI.index = df\_CI.index.map(int) # change the index to int type  df\_CI.plot(kind='line')  ​plt.title('Immigrants from China and India')  plt.ylabel('Number of Immigrants')  plt.xlabel('Years')  ​plt.show()  ​O*bserve the above plot it shows the China and India have very similar immigration trends through the years.*  Note: We didn't need to transpose Haiti's dataframe like df\_CI, That's because haiti is a series as opposed to a dataframe, and has the years as its indices as shown below.  print(type(haiti))class 'pandas.core.series.Series'  *Line plot is a handy tool to display several dependent variables against one independent variable. However, it is recommended that no more than 5-10 lines on a single graph; any more than that and it becomes difficult to interpret.*  # Let’s Compare the trend of top 5 countries that contributed the most to immigration to Canada.   * Step 1: Get the dataset.   # Recall that we created a Total column that calculates the cumulative immigration by country.  # We will sort on Total column to get our top 5 countries using pandas sort\_values() method.  # inplace = True paramemter saves the changes to the original df\_can dataframe  df\_can.sort\_values(by='Total', ascending=False, axis=0, inplace=True)   * Step 2: Get the top 5 entries   df\_top5 = df\_can.head(5)   * Step 3: transpose the dataframe   df\_top5 = df\_top5[years].transpose()  df\_top5.head()   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | India | China | United Kingdom… | Philippines | Pakistan | | 1980 | 8880 | 5123 | 22045 | 6051 | 978 | | 1981 | 8670 | 6682 | 24796 | 5921 | 972 | | 1982 | 8147 | 3308 | 20620 | 5249 | 1201 | | 1983 | 7338 | 1863 | 10015 | 4562 | 900 | | 1984 | 5704 | 1527 | 10170 | 3801 | 668 |  * # Step 2: Plot the dataframe.   # To make the plot more readeable, we will change the size using the `figsize` parameter.    df\_top5.index = df\_top5.index.map(int)  df\_top5.plot(kind='line', figsize=(14, 8))  plt.title('Immigration Trend of Top 5 Countries')  plt.ylabel('Number of Immigrants')  plt.xlabel('Years')  plt.show()  Other Plots  *Congratulations! you have learned how to wrangle data with python and create a line plot with Matplotlib. There are many other plotting styles available other than the default Line plot, all of which can be accessed by passing kind keyword to plot(). The list of plots are as follows:*   * bar for vertical bar plots df\_top5.plot(kind='bar',figsize=(12, 8)) * barh for horizontal bar plots df\_top5.plot(kind='barh',figsize=(12, 8)) * hist for histogram * box for boxplot df\_top5.plot(kind=’box’,figsize=(12, 8)) * kde or density for density plots * area for area plots * pie for pie plots df\_top5.head(5).plot(kind='pie',figsize=(14, 8),subplots=True) * scatter for scatter plots * hexbin for hexbin plot   My Exercise:  #df\_can.head()  #all(isinstance(column, str) for column in df\_can.columns) # examines the types of the column labels  #df\_top5.columns  #'United'in df\_top5.columns[2]  #df\_top5.rename(columns={'United Kingdom of Great Britain and Northern Ireland':'UnitedKingdom'}, inplace=True)  #df\_can[['Total']].head(5)  #df\_can.index  #df\_can[(df\_can['Continent']=='Asia') & (df\_can['Region']=='Southern Asia')]  #dpyears = list(map(str, range(1980, 1986)))  #df\_top5.index = df\_top5.index.map(int) # let's change the index values of df\_top5 to type integer for plotting  #df\_can.loc[['India','China'],dpyears].T.plot(kind='line',figsize=(5,3))  #df\_can.loc[['India','China'],dpyears].T.plot(kind='bar',figsize=(5,3))  #df\_can.loc[['India','China'],dpyears].T.plot(kind='barh',figsize=(5,3))  #df\_can.loc[['India','China'],dpyears].plot(kind='bar',figsize=(5,3))  #df\_can.loc[['India','China'],dpyears].T.plot(kind='box',figsize=(5,3))  #df\_can.loc[['India','China'],dpyears].T.plot(kind='box',figsize=(5,3),subplots=True)  #df\_can.loc[['India','China'],dpyears].T.plot(kind='pie',figsize=(14,6),subplots=True)  #df\_top5[['India','China']].head(5).plot(kind='pie',figsize=(14, 6),subplots=True)  #df\_can.loc[['India','China'],dpyears].plot(kind='pie',figsize=(20,15),subplots=True)  #plt.title('Immigration Trend of India & China')  #plt.ylabel('Number of Immigrants')  #plt.xlabel('Years')  plt.show()  # Area Plots  *In the last module, we created a line plot that visualized the top 5 countries that contribued the most immigrants to Canada from 1980 to 2013. With a little modification to the code, we can visualize this plot as a cumulative plot, also knows as a Stacked Line Plot or Area plot.*  *Area plots are stacked by default. And to produce a stacked area plot, each column must be either all positive or all negative values (any NaN values will defaulted to 0)*  # To produce an unstacked plot, pass stacked=False.  df\_top5.index = df\_top5.index.map(int)  df\_top5.plot(kind='area', stacked=False, figsize=(10, 5)  plt.title('Immigration Trend of Top 5 Countries')  plt.ylabel('Number of Immigrants')  plt.xlabel('Years')  plt.show()  *The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the `alpha` parameter value between 0 to 1.*  df\_top5.plot(kind='area', alpha=0.25,stacked=False, figsize=(10, 5)  Two types of plotting  *As we discussed in the video lectures, there are two styles/options of ploting with matplotlib. Plotting using the Artist layer and plotting using the scripting layer.*   * Option 1: Scripting layer (Procedural method) - using matplotlib.pyplot as 'plt'   *You can use plt i.e. matplotlib.pyplot and add more elements by calling different methods procedurally.*  *for example, plt.title(...) to add title or plt.xlabel(...) to add label to the x-axis.*  # This is what we have been using so far  df\_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))  plt.title('Immigration trend of top 5 countries')  plt.ylabel('Number of immigrants')  plt.xlabel('Years')   * Option 2: Artist layer (Object oriented method) - using an Axes instance from Matplotlib (preferred)   *You can use an Axes instance of your current plot and store it in a variable (eg. ax). You can add more elements by calling methods with a little change in syntax (by adding "set\_" to the previous methods). For example, use ax.set\_title() instead of plt.title() to add title, or ax.set\_xlabel() instead of plt.xlabel() to add label to the x-axis.*  *This option sometimes is more transparent and flexible in particular when having multiple plots, as we will see later.*    # Preferred option with more flexibility  ax = df\_top5.plot(kind='area', alpha=0.35, figsize=(10, 5))  ax.set\_title('Immigration Trend of Top 5 Countries')  ax.set\_ylabel('Number of Immigrants')  ax.set\_xlabel('Years')  *here, we will stick to the* ***scripting layer****, except for some advanced visualizations where we will need to use the* ***artist layer*** *to manipulate advanced aspects of the plots.*  *Use the scripting layer to create a stacked area plot of the 5 countries that contributed the least to immigration to Canada from 1980 to 2013. Use a transparency value of 0.45*  df\_can.sort\_values(['Total'], ascending=False, axis=0, inplace=True)  years = list(map(str, range(1980, 2014)))  df\_tail5 = df\_can.tail()  df\_tail5 = df\_tail5[years].transpose()  df\_tail5.head()  df\_tail5.index = df\_tail5.index.map(int)  df\_tail5.plot(kind='area',alpha = 0.45, figsize=(10, 5))  plt.title('Immigration Trend of 5 Countries with Least Contr to Immigration')  plt.ylabel('Number of Immigrants')  plt.xlabel('Years')  plt.show()  *Use the artist layer to create an unstacked area plot of the 5 countries that contributed the least to immigration to Canada from 1980 to 2013. Use a transparency value of 0.55.*  df\_tail5.index = df\_tail5.index.map(int)  ax = df\_tail5.plot(kind='area',alpha = 0.55,stacked = 'False', figsize=(10, 5))  ax.set\_title('Immigration Trend of 5 Countries with Least Contribution to Immigration')  ax.set\_ylabel('Number of Immigrants')  ax.set\_xlabel('Years')  Histograms  *A histogram is a way of representing the frequency distribution of numeric dataset. The way it works is it partitions the x-axis into bins, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So, the y-axis is the frequency or the number of data points in each bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.*  *Question: What is the frequency distribution of the number (population) of new immigrants from the various countries to Canada in 2013?*  *Before we proceed with creating the histogram plot, let's first examine the data split into intervals. To do this, we will us Numpy's histrogram method to get the bin ranges and frequency counts as follows:*  # let's quickly view the 2013 data  df\_can['2013'].head()  # np.histogram returns 2 values  count, bin\_edges = np.histogram(df\_can['2013'])  print(count) # frequency count  [178 11 1 2 0 0 0 0 1 2]  print(bin\_edges) # bin ranges, default = 10 bins  [0. 3412.9 6825.8 10238.7 13651.6 17064.5 20477.4 23890.3 27303.2 30716.1 34129.]  *By default, the histrogram method breaks up the dataset into 10 bins. We can see that in 2013:*   * 178 countries contributed between 0 to 3412.9 immigrants * 11 countries contributed between 3412.9 to 6825.8 immigrants * 1 country contributed between 6285.8 to 10238.7 immigrants, and so on   # Let’s plot the Histogram  df\_can['2013'].plot(kind='hist', figsize=(8, 3))  plt.title('Histogram of Immigration… Countries in 2013')  plt.ylabel('Number of Countries') # add y-label  plt.xlabel('Number of Immigrants') # add x-label  plt.show()  In the plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the aforementioned population.  *Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a xticks keyword that contains the list of the bin sizes, as follows*:  # 'bin\_e' is a list of bin intervals  count, bin\_e = np.histogram(df\_can['2013'])  df\_can['2013'].plot(kind='hist', figsize=(8, 5), xticks=bin\_e)  plt.title('Histogram of Immigration from 195 countries in 2013')  plt.ylabel('Number of Countries') # add y-label  plt.xlabel('Number of Immigrants') # add x-label  plt.show()  *Note: We could use df\_can['2013'].plot.hist(), instead.*  *In fact, using somedata.plot(kind='type\_plot', ...) is equivalent to some\_data.plot.type\_plot(...)*  *That is, passing the type of the plot as argument or method behaves the same.*  *See the pandas documentation for more info:* <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.plot.html>  *We can also plot multiple histograms on the same plot. For example, let's try to answer the following questions using a histogram.*  *Question: What is the immigration distribution for Denmark, Norway, and Sweden for years 1980 - 2013?*  # let's quickly view the dataset  df\_can.loc[['Denmark', 'Norway', 'Sweden'], years]   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | *Country* | *1980* | *1981* | *1982* | *1983* | *1984* | *1985* | *1986* | *1987* | *1988* | *1989* | *...* | *2004* | *2005* | *2006* | | *Denmark* | *272* | *293* | *299* | *106* | *93* | *73* | *93* | *109* | *129* | *129* | *...* | *89* | *62* | *101* | | *Norway* | *116* | *77* | *106* | *51* | *31* | *54* | *56* | *80* | *73* | *76* | *...* | *73* | *57* | *53* | | *Sweden* | *281* | *308* | *222* | *176* | *128* | *158* | *187* | *198* | *171* | *182* | *...* | *129* | *205* | *139* |   # Generate histogram  df\_can.loc[['Denmark', 'Norway', 'Sweden'], years].plot.hist()  *That does not look right! Don't worry, you'll often come across situations like this when creating plots. The solution often lies in how the underlying dataset is structured. This can be easily fixed by first transposing the dataset, also let's make a few modifications to improve the impact and aesthetics of the plot:*   * increase the bin size to 15 by passing in bins and set transparency to 60% by passing in alpha paramemter * label the x-axis by passing in x-label paramater * change the colors of the plots by passing in color parameter   *df\_t = df\_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose()*  *count, bin\_edges = np.histogram(df\_t, 15) # let's get the x-tick values*  *df\_t.plot(kind ='hist', figsize=(10, 6),bins=15, alpha=0.6, xticks=bin\_edges*  *,color=['coral', 'darkslateblue', 'mediumseagreen'])*  *plt.title('Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013')*  *plt.ylabel('Number of Years')*  *plt.xlabel('Number of Immigrants')*  *plt.show()*    Tip: For a full listing of colors available in Matplotlib, run the following code in your python shell:  import matplotlib  for name, hex in matplotlib.colors.cnames.items():  print(name, hex)  *If we do no want the plots to overlap each other, we can stack them using the stacked paramemter. Let's also adjust the min and max x-axis labels to remove the extra gap on the edges of the plot. We can pass a tuple (min, max) using the xlim paramater, as show below.*    df\_t = df\_can.loc[['Denmark', 'Norway', 'Sweden'],years].transpose()  count, bin\_edges = np.histogram(df\_t, 15) # Get the x-tick values  xmin = bin\_edges[0] - 10 # first bin value is 31.0, adding buffer of 10  xmax = bin\_edges[-1] + 10 # last bin value + buffer of 10  df\_t.plot(kind ='hist', figsize=(10, 6),bins=15, alpha=0.6  ,xticks=bin\_edges,color=['coral', 'darkslateblue', 'mediumseagreen']  , stacked=True, xlim=(xmin, xmax))  plt.title('Histogram… Denmark, Norway, and … from 1980 - 2013')  plt.ylabel('Number of Years')  plt.xlabel('Number of Immigrants')  plt.show()  *Use the scripting layer to display the immigration distribution for Greece, Albania, and Bulgaria for years 1980 – 2013.*  # Use an overlapping plot with 15 bins and a transparency value of 0.35.  df1=df\_can.loc[['Greece','Albania','Bulgaria'],years].T  count, i\_bin = np.histogram(df1, 15)  i\_xlim=(i\_bin[0]-100, i\_bin[-1]+100)  #i\_ylim=(count.min()-5,count.max()+5)  df1.plot.hist(figsize=(8,6)  ,bins=i\_bin  ,alpha=0.35  ,color=['coral', 'darkslateblue', 'mediumseagreen']  ,xticks=i\_bin  ,stacked=True  ,xlim=i\_xlim)#,ylim=i\_ylim)  plt.title('Histogram of Immigration’')  plt.ylabel('# of Years')  plt.xlabel('Number of Immigrants')  plt.show()  Bar Charts (Dataframe)  *A bar plot is a way of representing data where the length of the bars represents the magnitude/size of the feature/variable. Bar graphs usually represent numerical and categorical variables grouped in intervals.*  *To create a bar plot, we can pass one of two arguments via kind parameter in plot():*   1. kind=bar creates a vertical bar plot   somedata.plot(kind='bar’, ...)   1. kind=barh creates a horizontal bar plot   somedata.plot(kind='barh’, ...)  Vertical bar plot  *In vertical bar graphs, the x-axis is used for labelling, and the length of bars on the y-axis corresponds to the magnitude of the variable being measured. Vertical bar graphs are particuarly useful in analyzing time series data. One disadvantage is that they lack space for text labelling at the foot of each bar.*  *Let's start off by analyzing the effect of Iceland's Financial Crisis:*  *The 2008 - 2011 Icelandic Financial Crisis was a major economic and political event in Iceland. Relative to the size of its economy, Iceland's systemic banking collapse was the largest experienced by any country in economic history. The crisis led to a severe economic depression in 2008 - 2011 and significant political unrest.*  *Question: Let's compare the number of Icelandic immigrants (country = 'Iceland') to Canada from year 1980 to 2013.*  # Step 1: get the data  df\_iceland = df\_can.loc['Iceland', years]  df\_iceland.head()  # Step 2: plot data  df\_iceland.plot(kind='bar', figsize=(10, 6))  plt.xlabel('Year') # add to x-label to the plot  plt.ylabel('Number of immigrants') # add y-label to the plot  plt.title('Icelandic immigrants to Canada from 1980 to 2013') # add title to the plot  plt.show()  *The bar plot here shows the total number of immigrants broken down by each year. We can clearly see the impact of the financial crisis; the number of immigrants to Canada started increasing rapidly after 2008.*  *Let's annotate this on the plot using the annotate method of the scripting layer or the pyplot interface. We will pass in the following parameters:*   * s: str, the text of annotation. * xy: Tuple specifying the (x,y) point to annotate (in this case, end point of arrow). * xytext: Tuple specifying the (x,y) point to place the text (in this case, start point of arrow). * xycoords: The coordinate system that xy is given in - 'data' uses the default system of the object being annotated * arrowprops: Takes a dictionary of properties to draw the arrow: * arrowstyle: Specifies the arrow style, '->' is standard arrow. * connectionstyle: Specifies the connection type. arc3 is a straight line. * color: Specifes color of arror. * lw: Specifies the line width.   *Read the Matplotlib documentation for annotations:*  <http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.annotate>  df\_iceland.plot(kind='bar', figsize=(10, 6), rot=90) # rotate the bars by 90 degrees  plt.xlabel('Year')  plt.ylabel('Number of Immigrants')  plt.title('Icelandic Immigrants to Canada from 1980 to 2013')  # Annotate arrow  plt.annotate('', # s: str. Will leave it blank for no text  xy=(32, 70), # place head of the arrow at point (year 2012 , pop 70)  xytext=(28, 20), # place base of the arrow at point (year 2008 , pop 20)  xycoords='data', # will use the coordinate system of the object being annotated  arrowprops=dict(arrowstyle='->', connectionstyle='arc3', color='blue', lw=2)  )  plt.show()  Let's also annotate a text to go over the arrow. We will pass in the following additional parameters:   * rotation: rotation angle of text in degrees (counter clockwise) * va: vertical alignment of text [‘center’ | ‘top’ | ‘bottom’ | ‘baseline’] * ha: horizontal alignment of text [‘center’ | ‘right’ | ‘left’]   df\_iceland.plot(kind='bar', figsize=(10, 6), rot=90)    plt.xlabel('Year')  plt.ylabel('Number of Immigrants')  plt.title('Icelandic Immigrants to Canada from 1980 to 2013')  # Annotate arrow  plt.annotate('', # s: blank string  xy=(32, 70), # Head@ (year2012 , pop 70)  xytext=(28, 20), #Base@ (year2008 , pop 20)  xycoords='data', #Coordinate sys of object  arrowprops=dict(arrowstyle='->', connectionstyle='arc3',  color='blue', lw=2))  # Annotate Text  plt.annotate('2008 - 2011 Financial Crisis', # text to display  xy=(28, 30), # start the text at at point (year 2008 , pop 30)  rotation=72.5, # based on trial and error to match the arrow  va='bottom', # want the text to be vertically 'bottom' aligned  ha='left') # want the text to be horizontally 'left' aligned.  plt.show()  Horizontal Bar Plot  *Sometimes it is more practical to represent the data horizontally, especially if you need more room for labelling the bars. In horizontal bar graphs, the y-axis is used for labelling, and the length of bars on the x-axis corresponds to the magnitude of the variable being measured. As you will see, there is more room on the y-axis to label categetorical variables.*  *Using the scripting layter and the df\_can dataset, create a horizontal bar plot showing the total number of immigrants to Canada from the top 15 countries, for the period 1980 - 2013. Label each country with the total immigrant count.*   * Get the data pertaining to the top 15 countries. * Use kind='barh' to generate a bar chart with horizontal bars. * Make sure to choose a good size for the plot and to label your axes and to give the plot a title. * Loop the countries and annotate the immigrant population using the anotate function of the scripting interface.   # Get top 15 countries & Generate plot  df\_can.sort\_values(by='Total', ascending=True, inplace=True)  df\_top15 = df\_can['Total'].tail(15)  df\_top15.plot(kind='barh', figsize=(12, 12), color='steelblue')  plt.xlabel('Number of Immigrants')  plt.title('Top 15 Conuntries between 1980 - 2013')  # Annotate value labels to each country  for index, value in enumerate(df\_top15):  label = format(int(value), ',') # format int with commas  # Place text at the end of bar  # Subtract 50K from x, and 0.1 from y to fit it within the bar  plt.annotate(label, xy=(value - 50000, index - 0.10), color='white')  plt.show()  *Note: in above code label = formatted value = df\_top15[index], check below written code*   |  |  | | --- | --- | | # Annotate value labels to each country  for index,value in enumerate(df\_top15):  plt.annotate(format(int(value),','), xy=(value+4, index - 0.10), color='red')  # Creating tuple from list, converting panda’s series to list:  mylabel = tuple(df\_top15.tolist()) | # Also, check  df\_top15.head()  df\_top15.index  for index,value in enumerate(df\_top15):  print(index,value) | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pie Charts  *A pie chart is a circular graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the kind=pie keyword.*  *Before we proceed let’s prepare workspace:*   1. *Download and import our primary Canadian Immigration dataset using pandas read\_excel() method.* 2. *Clean data, we will make some modifications to the original dataset to make it easier to create our visualizations*   *Refer to Introduction to Matplotlib and Line Plots and Area Plots, Histograms, and Bar Plots for a detailed description of this processing.*  # Import primary modules  import numpy as np # useful for many scientific computing in Python  import pandas as pd # primary data structure library  %matplotlib inline  import matplotlib as mpl  import matplotlib.pyplot as plt  mpl.style.use('ggplot') # optional: for ggplot-like style  print('Matplotlib version: ', mpl.\_\_version\_\_) # Check for latest version of Matplotlib  # Download the dataset and read it into a \*pandas\* dataframe.  df\_can = pd.read\_excel('https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/Canada.xlsx',  sheet\_name='Canada by Citizenship', skiprows=range(20), skipfooter=2)  print('Data downloaded and read into a dataframe!')  # Remove unnecessary columns (eg. REG)  df\_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)  # let's rename the columns so that they make sense  df\_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)  # for sake of consistency, let's also make all column labels of type string  df\_can.columns = list(map(str, df\_can.columns))  # set the country name as index - useful for quickly looking up countries using ‘.loc’ method  df\_can.set\_index('Country', inplace=True)  # add total column  df\_can['Total'] = df\_can.sum(axis=1)  # years that we will be using in this lesson - useful for plotting later on  years = list(map(str, range(1980, 2014)))  print('data dimensions:', df\_can.shape)  *Let's use a pie chart to explore the proportion (percentage) of new immigrants grouped by continents for the entire time period from 1980 to 2013.*  Step 1: Gather data  *We will use pandas groupby method to summarize the immigration data by Continent. The general process of groupby involves the following steps:*   * **Split**: Splitting the data into groups based on some criteria * **Apply**: Applying a function to each group independently: * **Combine**: Combining the results into a data structure   *Group countries by continents and apply sum() function*  df\_continents = df\_can.groupby('Continent', axis=0).sum()  *Note, the output of the groupby method is a ‘groupby’ object. We cannot use it further until we apply a function (eg .sum(), .count(), .mean(), .std(), .aggregate(), .apply(). Etc...)*  print(type(df\_can.groupby('Continent', axis=0)))  df\_continents.head()  Step 2: Plot the data  *We will pass in kind = 'pie' keyword, along with the following additional parameters:*   * autopct - is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be fmt%pct. * startangle - Rotates the start of the pie chart by angle degrees counterclockwise from the x-axis. * shadow - Draws a shadow beneath the pie (to give a 3D feel).   # autopct create %, start angle represent starting point  df\_continents['Total'].plot(kind='pie',  figsize=(5, 6),  autopct='%1.1f%%', # add in percentages  startangle=90, # start angle 90° (Africa)  shadow=True) # add shadow  plt.title('Immigration to Canada by Continent [1980 - 2013]')  plt.axis('equal') # Sets the pie chart to look like a circle.  plt.show()  *The Graph is not very clear, the numbers and text overlap in some instances. Let's make a few modifications to improve the visuals:*   * Remove the text labels on the pie chart by passing in legend and add it as a seperate legend using plt.legend(). * Push out the percentages to sit just outside the pie chart by passing in pctdistance parameter. * Pass in a custom set of colors for continents by passing in colors parameter. * Explode the pie chart to emphasize the lowest three continents (Africa, North America, and Latin America and Carribbean) by passing in explode parameter.   # color for each continent in list  colors\_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink']  # Ratio for each continent with which to offset each wedge.  explode\_list = [0.1, 0, 0, 0, 0.1, 0.1] # Try changing the value  df\_continents['Total'].plot(kind='pie', figsize=(15, 6),  autopct='%1.1f%%',  startangle=90,  shadow=True,  labels=None, # turn off labels on pie chart  # The ratio between the center of each pie slice and autopct label  pctdistance=1.12,  colors=colors\_list, # add custom colors  explode=explode\_list) # 'explode' lowest 3 continents  # scale the title up by 12% to match pctdistance  plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.12)  plt.axis('equal')  plt.legend(labels=df\_continents.index, loc='upper left') # add legend  plt.show()  *you may try after sorting values by ascending use this df\_continents.sort\_values(['Total'],axis=0, inplace=True)*  # Using a pie chart, explore the proportion (percentage) of new immigrants grouped by continents in the year 2013.  #colors\_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink']  explode\_list = [0.1, 0, 0, 0, 0.1, 0.2] # ratio for each continent with which to offset each wedge.  df\_continents['2013'].plot(kind='pie',  figsize=(15, 6),  autopct='%1.1f%%', # percentages label  startangle=90, # start angle 90° (Africa)  shadow=True, # add shadow  labels=None, # turn off labels on pie chart  pctdistance=1.12, # label dist  #colors=colors\_list, # add custom colors  explode=explode\_list) # 'explode' lowest 3 continents  plt.title('Immigration to Canada by Continent in Year 2013',y=1.12)  plt.axis('equal') # Sets the pie chart to look like a circle.  # add legend  plt.legend(labels=df\_continents.index, loc='upper right')  plt.show()  Box Plots  *A box plot is a way of statistically representing the distribution of the data through five main dimensions:*   * ***Minimun:***Smallest number in the dataset. * ***First quartile****:* Middle number between the minimum and the median. * Second quartile (***Median***): Middle number of the (sorted) dataset. * ***Third quartile***: Middle number between median and maximum. * ***Maximum***: Highest number in the dataset.   *To make a box plot, we can use kind=box in plot method invoked on a pandas series or dataframe.*  *Let's plot the box plot for the Japanese immigrants between 1980 - 2013.*   1. *Get the dataset. Even though we are extracting the data for just one country, we will obtain it as a dataframe. This will help us with calling the dataframe.describe() method to view the percentiles.*   # To get a dataframe, place extra square brackets around 'Japan'.  df\_japan = df\_can.loc[['Japan'], years].transpose()  df\_japan.head()   1. *Plot by passing in kind='box'.*   df\_japan.plot(kind='box', figsize=(8, 6)) # df\_japan.plot.box(figsize=(8, 6))  plt.title('Box plot of Japanese Immigrants from 1980 - 2013')  plt.ylabel('Number of Immigrants')  plt.show()  *We can immediately make a few key observations from the plot above:*   * The minimum number of immigrants is around 200 (min), maximum number is around 1300 (max), and median number of immigrants is around 900 (median). * 25% of the years for period 1980 - 2013 had an annual immigrant count of ~500 or fewer (First quartile). * 75% of the years for period 1980 - 2013 had an annual immigrant count of ~1100 or fewer (Third quartile).   *We can view the actual numbers by calling the describe() method on the dataframe:*  df\_japan.describe()   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **country** | Japan | **std** | 337.219771 | **50%** | 902.000000 | | **count** | 34.000000 | **min** | 198.000000 | **75%** | 1079.000000 | | **mean** | 814.911765 | **25%** | 529.000000 | **max** | 1284.000000 |   *One of the key benefits of box plots is comparing the distribution of multiple datasets. In one of the previous labs, we observed that China and India had very similar immigration trends. Let's analyize these two countries further using box plots.*    *Compare the distribution of the number of new immigrants from India and China for the period 1980 - 2013.*  # Get the dataset for China and India and call the dataframe df\_CI  df\_CI = df\_can.loc[['China','India'], years].transpose()  df\_CI.head()  df\_CI.describe()  df\_CI.plot.box(figsize=(10, 7)) # Plot data  plt.title('Box plots of Immigrants from China and India (1980 - 2013)')  plt.xlabel('Number of Immigrants')  *We can observe that, while both countries have around the same median immigrant population (~20K), China's immigrant population range is more spread out than India's. The maximum population from India for any year (36K) is around 15% lower than the maximum population from China (42K).*  *If you prefer to create horizontal box plots, you can pass the vert parameter in the plot function and assign it to False. You can also specify a different color.*  Horizontal box plots  df\_CI.plot(kind='box', figsize=(10, 7), color='blue', vert=False)  Subplots  *Often times we might want to plot multiple plots within the same figure. For example, we might want to perform a side by side comparison of the box plot with the line plot of China and India's immigration.*  *To visualize multiple plots together, we can create a figure (overall canvas) and divide it into subplots, each containing a plot. With subplots, we usually work with the artist layer instead of the scripting layer.*  Typical syntax is :  fig = plt.figure() # create figure  ax = fig.add\_subplot(nrows, ncols, plot\_number) # create subplots  Where,  **nrows and ncols** are used to notionally split the figure into (nrows \* ncols) sub-axes,  **plot\_number** is used to identify the particular subplot that this function is to create within the notional grid. plot\_number starts at 1, increments across rows first and has a maximum of nrows \* ncols as shown below.  *We can then specify which subplot to place each plot by passing in the ax paramemter in plot() method as follows:*  fig = plt.figure() # create figure  ​  ax0 = fig.add\_subplot(1, 2, 1) # add subplot 1 (1 row, 2 columns, first plot)  ax1 = fig.add\_subplot(1, 2, 2) # add subplot 2 (1 row, 2 columns, second plot)  See tip below\*\*  ​  # Add Subplot 1: Box plot  df\_CI.plot(kind='box', color='blue', vert=False, figsize=(20, 6), ax=ax0)  ax0.set\_title('Box Plots of Immigrants from China and India (1980 - 2013)')  ax0.set\_xlabel('Number of Immigrants')  ax0.set\_ylabel('Countries')  ​  # Add Subplot 2: Line plot  df\_CI.plot(kind='line', figsize=(20, 6), ax=ax1)  ax1.set\_title ('Line Plots of Immigrants')  ax1.set\_ylabel('Number of Immigrants')  ax1.set\_xlabel('Years')  ​  plt.show()  *\*\*Tip regarding subplot convention\*\**  *In the case when nrows, ncols, and plot\_number are all less than 10, a convenience exists such that the a 3 digit number can be given instead, where the hundreds represent nrows, the tens represent ncols and the units represent plot\_number.*  *For instance,* subplot(211) == subplot(2, 1, 1) *produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 rows by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).*  *Let's try something a little more advanced. Previously we identified the top 15 countries based on total immigration from 1980 - 2013. Create a box plot for the distribution of the top 15 countries grouped by the decades 1980s, 1990s, and 2000s.*   1. Get the dataset. Get the top 15 countries based on Total immigrant population.   df\_top15 = df\_can.sort\_values(by='Total',ascending=False,axis=0).head(15)  df\_top15   1. Create a new dataframe which contains the aggregate for each decade. One way to do that:  * Create a list of all years in decades 80's, 90's, and 00's.   years\_80s = list(map(str, range(1980, 1990)))  years\_90s = list(map(str, range(1990, 2000)))  years\_00s = list(map(str, range(2000, 2010)))   * Slice the original dataframe df\_can to create a series for each decade and sum across all years for each country.   df\_80s = df\_top15.loc[:, years\_80s].sum(axis=1)  df\_90s = df\_top15.loc[:, years\_90s].sum(axis=1)  df\_00s = df\_top15.loc[:, years\_00s].sum(axis=1)   * Merge the three series into a new data frame. Call your dataframe new\_df.   new\_df = pd.DataFrame({'1980s': df\_80s, '1990s': df\_90s, '2000s':df\_00s})  new\_df.head()  Country 1980s 1990s 2000s  India 82154 180395 303591  China 32003 161528 340385  …. ….. ….. …..   1. Plot the box plots.   new\_df.plot(kind='box', figsize=(6, 6))  plt.title('Immigration from top 15 countries for decades 80s, 90s and 2000s',y=1.05)  plt.show()  *# Let's learn more about the statistics associated with the dataframe using the describe() method.*   |  |  |  |  | | --- | --- | --- | --- | |  | 1980s | 1990s | 2000s | | count | 15.000000 | 15.000000 | 15.000000 | | mean | 44418.333333 | 85594.666667 | 97471.533333 | | std | 44190.676455 | 68237.560246 | 100583.204205 | | min | 7613.000000 | 30028.000000 | 13629.000000 | | 25% | 16698.000000 | 39259.000000 | 36101.500000 | | 50% | 30638.000000 | 56915.000000 | 65794.000000 | | 75% | 59183.000000 | 104451.500000 | 105505.500000 | | max | 179171.000000 | 261966.000000 | 340385.000000 |   *Note: how the box plot differs from the summary table created. The box plot scans the data and identifies the outliers. In order to be an outlier, the data value must be: Larger than Q3 by at least 1.5 times the interquartile range (IQR), or smaller than Q1 by at least 1.5 times the IQR.*  # Let's look at decade 2000s as an example:  Q1 (25%) = 36,101.5  Q3 (75%) = 105,505.5  IQR = Q3 - Q1 = 69,404  # Using the definition of outlier, any value that is greater than Q3 by 1.5 times IQR will be flagged as outlier.  Outlier > 105,505.5 + (1.5 \* 69,404)  Outlier > 209,611.5  # let's check how many entries fall above the outlier threshold  new\_df[new\_df['2000s']> 209611.5]   |  |  |  |  | | --- | --- | --- | --- | | Country | 1980s | 1990s | 2000s | | India | 82154 | 180395 | 303591 | | China | 32003 | 161528 | 340385 |   *China and India are both considered as outliers since their population for the decade exceeds 209,611.5.*  The box plot is an advanced visualizaiton tool, and there are many options and customizations that exceed the scope of this lab. Please refer to Matplotlib documentation on [box plots](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.boxplot) for more information.  Scatter Plots  *A scatter plot (2D) is a useful method of comparing variables against each other. Scatter plots look similar to line plots in that they both map independent and dependent variables on a 2D graph. While the datapoints are connected together by a line in a line plot, they are not connected in a scatter plot.*  *The data in a scatter plot is considered to express a trend. With further analysis using tools like regression, we can mathematically calculate this relationship and use it to predict trends outside the dataset.*  *Let's creat scatter plot, to visualize the trend of total immigrantion to Canada for the years 1980 - 2013.*   1. Get the dataset. Since we are expecting to use the relationship betewen years and total population, we will convert years to int type.  |  |  |  | | --- | --- | --- | |  | year | total | | 0 | 1980 | 99137 | | 1 | 1981 | 110563 | | 2 | 1982 | 104271 | | … | … | …. |   # we can use the sum() method to get the total population per year  df\_tot = pd.DataFrame(df\_can[years].sum(axis=0))  ​# change the years to type int (useful for regression later on)  df\_tot.index = map(int, df\_tot.index)  ​# reset the index to put in back in as a column in the df\_tot dataframe  df\_tot.reset\_index(inplace = True)  ​  df\_tot.columns = ['year', 'total'] # rename columns  df\_tot.head() # view the final dataframe   1. Plot the data. In Matplotlib, we can create a scatter plot set by passing in kind='scatter' as plot argument. We will also need to pass in x and y keywords to specify the columns that go on the x- and the y-axis.     df\_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')  ​plt.title('Total Immigration to Canada from 1980 - 2013')  plt.xlabel('Year')  plt.ylabel('Number of Immigrants')  ​plt.show()  *Notice how the scatter plot does not connect the datapoints together. We can clearly observe an upward trend in the data: as the years go by, the total number of immigrants increases. We can mathematically analyze this trend using a regression line (line of best fit).*  *Now, let's try to plot a linear line of best fit, and use it to predict the number of immigrants in 2015.*   1. Get the equation of line of best fit. We will use Numpy's polyfit() method by passing in the following:   x: x-coordinates of the data.  y: y-coordinates of the data.  deg: Degree of fitting polynomial. 1 = linear, 2 = quadratic, and so on.  x = df\_tot['year'] # year on x-axis  y = df\_tot['total'] # total on y-axis  fit = np.polyfit(x, y, deg=1)  ​fit # array([ 5.56709228e+03, -1.09261952e+07])  The output is an array with the polynomial coefficients, highest powers first. Since we are plotting a linear regression y= a\*x + b, our output has 2 elements with the the slope in position 0 and intercept in position 1.   1. Plot the regression line on the scatter plot.   df\_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')  ​plt.title('Total Immigration to Canada from 1980 - 2013')  plt.xlabel('Year')  plt.ylabel('Number of Immigrants')  ​  # plot line of best fit  plt.plot(x, fit[0] \* x + fit[1], color='red') # recall that x is the Years  plt.annotate('y={0:.0f} x + {1:.0f}'.format(fit[0], fit[1]), xy=(2000, 150000))  ​plt.show()  ​# Print out the line of best fit  'No. Immigrants = {0:.0f} \* Year + {1:.0f}'.format(fit[0], fit[1])  'No. Immigrants = 5567 \* Year + -10926195'  # Using the equation of line of best fit, we can estimate the number of immigrants in 2015:  No. Immigrants = 5567 \* 2015 - 10926195  No. Immigrants = 291,310  *When compared to the actuals from Citizenship and Immigration Canada's (CIC) 2016 Annual Report, we see that Canada accepted 271,845 immigrants in 2015. Our estimated value of 291,310 is within 7% of the actual number, which is pretty good considering our original data came from United Nations (and might differ slightly from CIC data).*  *As a side note, we can observe that immigration took a dip around 1993 - 1997. Further analysis into the topic revealed that in 1993 Canada introcuded Bill C-86 which introduced revisions to the refugee determination system, mostly restrictive. Further amendments to the Immigration Regulations cancelled the sponsorship required for "assisted relatives" and reduced the points awarded to them, making it more difficult for family members (other than nuclear family) to immigrate to Canada. These restrictive measures had a direct impact on the immigration numbers for the next several years.*  #Create a scatter plot of the total immigration from Denmark, Norway, and Sweden to Canada from 1980 to 2013?   * Create a ‘df\_countries’ dataframe the consists of the numbers associated with Denmark, Norway, and Sweden only. * Sum the immigration numbers of all three countries for each year and turn the result into a ‘df\_total’ dataframe. * Reset the index in place. * Rename the columns to year and total. * Display and plot the resulting dataframe   # create df\_countries dataframe  df\_countries = df\_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose()  #df\_countries = df\_can.loc[['Denmark','Norway','Sweden'],years]  # create df\_total by summing across three countries for each year  df\_total = pd.DataFrame(df\_countries.sum(axis=1))  #df\_total = pd.DataFrame(df\_countries[years].sum(axis=0))  # reset index in place  df\_total.reset\_index(inplace=True)  # rename columns  df\_total.columns = ['year', 'total']  # change column year from string to int to create scatter plot  df\_total['year'] = df\_total['year'].astype(int)  # show resulting dataframe  df\_total.head()  # generate scatter plot  df\_total.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')  # add title and label to axes  plt.title('Immigration from Denmark, Norway, and Sweden to Canada from 1980 - 2013')  plt.xlabel('Year')  plt.ylabel('Number of Immigrants')  plt.show() # show plot  ​  ​Bubble Plots  A bubble plot is a variation of the scatter plot that displays three dimensions of data (x, y, z). The datapoints are replaced with bubbles, and the size of the bubble is determined by the third variable 'z', also known as the weight. In maplotlib, we can pass in an array or scalar to the keyword s to plot(), that contains the weight of each point.  *Let's start by analyzing the effect of Argentina's great depression.*  *Argentina suffered a great depression from 1998 - 2002, which caused widespread unemployment, riots, the fall of the government, and a default on the country's foreign debt. In terms of income, over 50% of Argentines were poor, and seven out of ten Argentine children were poor at the depth of the crisis in 2002.*  *Let's analyze the effect of this crisis, and compare Argentina's immigration to that of it's neighbour Brazil. Let's do that using a bubble plot of immigration from Brazil and Argentina for the years 1980 - 2013. We will set the weights for the bubble as the normalized value of the population for each year.*   1. Get the data for Brazil and Argentina. Like in the previous example, we will convert the Years to type int and bring it in the dataframe.   df\_can\_t = df\_can[years].transpose() # transposed dataframe  ​df\_can\_t.index = map(int, df\_can\_t.index) # cast the Years (the index) to type int  ​# let's label the index. This will automatically be the column name when we reset the index  df\_can\_t.index.name = 'Year' # reset index to bring the Year in as a column  ​df\_can\_t.reset\_index(inplace=True)  df\_can\_t.head() # view the changes   1. Create the normalized weights.   *There are several methods of normalizations in statistics, each with its own use. In this case, we will use feature scaling to bring all values into the range [0,1]. The general formula is:*  *Where, X is an original value, X' is the normalized value. The formula sets the max value in the dataset to 1, and sets the min value to 0. The rest of the datapoints are scaled to a value between 0-1 accordingly.*  # normalize Brazil data  norm\_brazil = (df\_can\_t['Brazil'] - df\_can\_t['Brazil'].min()) / (df\_can\_t['Brazil'].max() - df\_can\_t['Brazil'].min())  # normalize Argentina data  norm\_argentina = (df\_can\_t['Argentina'] - df\_can\_t['Argentina'].min()) / (df\_can\_t['Argentina'].max() - df\_can\_t['Argentina'].min())   1. Plot the data.   *To plot two different scatter plots in one plot, we can include the axes one plot into the other by passing it via the ax parameter.We will also pass in the weights using the s parameter. Given that the normalized weights are between 0-1, they won't be visible on the plot. Therefore, we will multiply weights by 2000 to scale it up on the graph, and,*  *add 10 to compensate for the min value (which has a 0 weight and therefore scale with x2000).*  # Brazil  ax0 = df\_can\_t.plot(kind='scatter',  x='Year',  y='Brazil',  figsize=(7, 4),  alpha=0.5, # transparency  color='green',  s=norm\_brazil \* 2000 + 10, # pass in weights  xlim=(1975, 2015)  )  # Argentina  ax1 = df\_can\_t.plot(kind='scatter',  *The size of the bubble corresponds to the magnitude of immigrating population for that year, compared to the 1980 - 2013 data. The larger the bubble, the more immigrants in that year.*  *From the plot above, we can see a corresponding increase in immigration from Argentina during the 1998 - 2002 great depression. We can also observe a similar spike around 1985 to 1993. In fact, Argentina had suffered a great depression from 1974 - 1990, just before the onset of 1998 - 2002 great depression.*  x='Year',  y='Argentina',  alpha=0.5,  color="blue",  s=norm\_argentina \* 2000 + 10,  ax = ax0  )  ax0.set\_ylabel('Number of Immigrants')  ax0.set\_title('Immigration from Brazil and Argentina from 1980 - 2013')  ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')  ​  *On a similar note, Brazil suffered the Samba Effect where the Brazilian real (currency) dropped nearly 35% in 1999. There was a fear of a South American financial crisis as many South American countries were heavily dependent on industrial exports from Brazil. The Brazilian government subsequently adopted an austerity program, and the economy slowly recovered over the years, culminating in a surge in 2010. The immigration data reflect these events.*  *Previously in this lab, we created box plots to compare immigration from China and India to Canada. Create bubble plots of immigration from China and India to visualize any differences with time from 1980 to 2013. You can use df\_can\_t that we defined and used in the previous example.*  Step 1: Normalize the data pertaining to China and India.  # normalize Brazil data  norm\_india = (df\_can\_t['India'] - df\_can\_t['India'].min()) / (df\_can\_t['India'].max() - df\_can\_t['India'].min())  ​# normalize Argentina data  norm\_china= (df\_can\_t['China'] - df\_can\_t['China'].min()) / (df\_can\_t['China'].max() - df\_can\_t['China'].min())  Step 2: Generate the bubble plots.  # China  ax0 = df\_can\_t.plot(kind='scatter', x='Year', y='China', figsize=(7, 4)  , alpha=.7, color='green', s=norm\_brazil \* 2000 + 10  , xlim=(1975, 2015))  ​  # India  ax1 = df\_can\_t.plot(kind='scatter', x='Year', y='India', alpha=.7  , color="orange", s=norm\_argentina \* 2000 + 10, ax = ax0)  ​  ax0.set\_ylabel('Number of Immigrants')  ax0.set\_title('Immigration from China and India from 1980 - 2013')  ax0.legend(['China', 'India'], loc='upper left', fontsize='x-large')  Waffle Charts, Word Clouds, and Regression Plots:  *Let’s setup the workspace before we go further:*  # Import Primary Modules:  #!conda install -c anaconda xlrd --yes  import numpy as np # useful for many scientific computing in Python  import pandas as pd # primary data structure library  from PIL import Image # converting images into arrays  # Download the dataset and read it into a pandas dataframe:  df\_can = pd.read\_excel('https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/Canada.xlsx',  sheet\_name='Canada by Citizenship',  skiprows=range(20),  skipfooter=2)  print('Data downloaded and read into a dataframe!')  *Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to Introduction to Matplotlib and Line Plots and Area Plots, Histograms, and Bar Plots for a detailed description of this preprocessing.*  # clean up the dataset to remove unnecessary columns (eg. REG)  df\_can.drop(['AREA','REG','DEV','Type','Coverage'], axis = 1, inplace = True)  # let's rename the columns so that they make sense  df\_can.rename (columns = {'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace = True)  # for sake of consistency, let's also make all column labels of type string  df\_can.columns = list(map(str, df\_can.columns))  # set the country name as index - useful for quickly looking up countries using .loc method  df\_can.set\_index('Country', inplace = True)  # add total column  df\_can['Total'] = df\_can.sum (axis = 1)  # years that we will be using in this lesson - useful for plotting later on  years = list(map(str, range(1980, 2014)))  print ('data dimensions:', df\_can.shape)  # Import matplotlib:  %matplotlib inline  ​import matplotlib as mpl  import matplotlib.pyplot as plt  import matplotlib.patches as mpatches # needed for waffle Charts  ​mpl.style.use('ggplot') # optional: for ggplot-like style  print ('Matplotlib version: ', mpl.\_\_version\_\_) # check for latest version of Matplotlib i.e. >= 2.0.0  Waffle Charts  *A waffle chart is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.*  *Let's revisit the previous case study about Denmark, Norway, and Sweden.*  # Create a new dataframe for these three countries  df\_dsn = df\_can.loc[['Denmark', 'Norway', 'Sweden'], :]  ​df\_dsn # let's take a look at our dataframe  *Unfortunately, unlike R, waffle charts are not built into any of the Python visualization libraries. Therefore, we will learn how to create them from scratch.*   1. The first step into creating a waffle chart is determing the proportion of each category with respect to the total.   # compute the proportion of each category with respect to the total  total\_values = sum(df\_dsn['Total'])  category\_proportions = [(float(value) / total\_values) for value in df\_dsn['Total']]  # print out proportions  for i, proportion in enumerate(category\_proportions):  print (df\_dsn.index.values[i] + ': ' + str(proportion))   1. The second step is defining the overall size of the waffle chart.   width = 40 # width of chart  height = 10 # height of chart  total\_num\_tiles = width \* height # total number of tiles  print ('Total number of tiles is ', total\_num\_tiles)   1. The third step is using the proportion of each category to determe it respective number of tiles   # compute the number of tiles for each catagory  tiles\_per\_category = [round(proportion \* total\_num\_tiles) for proportion in category\_proportions]  # print out number of tiles per category  for i, tiles in enumerate(tiles\_per\_category):  print (df\_dsn.index.values[i] + ': ' + str(tiles))   1. The fourth step is creating a matrix that resembles the waffle chart and populating it.   # initialize the waffle chart as an empty matrix  waffle\_chart = np.zeros((height, width))  ​# define indices to loop through waffle chart  category\_index = 0  tile\_index = 0  ​# populate the waffle chart  for col in range(width):  for row in range(height):  tile\_index += 1  ​ # if the number of tiles populated for the current category is equal to its corresponding allocated tiles...  if tile\_index > sum(tiles\_per\_category[0:category\_index]):  # ...proceed to the next category  category\_index += 1  # set the class value to an integer, which increases with class  waffle\_chart[row, col] = category\_index  print ('Waffle chart populated!')  waffle\_chart # Let's take a peek at how the matrix looks like.   1. Next step is to map the waffle chart matrix into a visual.   # instantiate a new figure object  fig = plt.figure()  ​# use matshow to display the waffle chart  colormap = plt.cm.coolwarm  plt.matshow(waffle\_chart, cmap=colormap)  plt.colorbar()   1. Lets prettify the chart.   # instantiate a new figure object  fig = plt.figure()  ​# use matshow to display the waffle chart  colormap = plt.cm.coolwarm  plt.matshow(waffle\_chart, cmap=colormap)  plt.colorbar()  ​# get the axis  ax = plt.gca()  ​# set minor ticks  ax.set\_xticks(np.arange(-.5, (width), 1), minor=True)  ax.set\_yticks(np.arange(-.5, (height), 1), minor=True)  # add gridlines based on minor ticks  ax.grid(which='minor', color='w', linestyle='-', linewidth=2)  ​plt.xticks([])  plt.yticks([])   1. Last step is to create a legend and add it to chart.   # compute cumulative sum of individual categories to match color schemes between chart and legend  values\_cumsum = np.cumsum(df\_dsn['Total'])  total\_values = values\_cumsum[len(values\_cumsum) - 1]  # create legend  legend\_handles = []  for i, category in enumerate(df\_dsn.index.values):  label\_str = category + ' (' + str(df\_dsn['Total'][i]) + ')'  color\_val = colormap(float(values\_cumsum[i])/total\_values)  legend\_handles.append(mpatches.Patch(color=color\_val, label=label\_str))  # add legend to chart  plt.legend(handles=legend\_handles,  loc='lower center',  ncol=len(df\_dsn.index.values),  bbox\_to\_anchor=(0., -0.2, 0.95, .1))  *Now it would very inefficient to repeat these seven steps every time we wish to create a waffle chart. So let's combine all seven steps into one function called create\_waffle\_chart. This function would take the following parameters as input:*   * categories: Unique categories or classes in dataframe. * values: Values corresponding to categories or classes. * height: Defined height of waffle chart. * width: Defined width of waffle chart. * colormap: Colormap class * value\_sign: In order to make our function more generalizable, we will add this parameter to address signs that could be associated with a value such as %, $, and so on. value\_sign has a default value of empty string.   def create\_waffle\_chart(categories, values, height, width, colormap, value\_sign=''):  total\_values = sum(values) **# compute the proportion of each category with respect to the total**  category\_proportions = [(float(value) / total\_values) for value in values]  total\_num\_tiles = width \* height # total number of tiles **# compute the total number of tiles**  print ('Total number of tiles is', total\_num\_tiles)    **# compute the number of tiles for each catagory**  tiles\_per\_category = [round(proportion \* total\_num\_tiles) for proportion in category\_proportions]  for i, tiles in enumerate(tiles\_per\_category): **# print out number of tiles per category**  print (categories [i] + ': ' + str(tiles))    waffle\_chart = np.zeros((height, width)) **# initialize the waffle chart as an empty matrix**  category\_index = 0 **# define indices to loop through waffle chart**  tile\_index = 0  **# populate the waffle chart**  for col in range(width):  for row in range(height):  tile\_index += 1  **# if the number of tiles populated for the current category**  **# is equal to its corresponding allocated tiles...**  if tile\_index > sum(tiles\_per\_category[0:category\_index]):  # ...proceed to the next category  category\_index += 1  **# set the class value to an integer, which increases with class**  waffle\_chart[row, col] = category\_index  fig = plt.figure() **# instantiate a new figure object**  **# use matshow to display the waffle chart**  colormap = plt.cm.coolwarm  plt.matshow(waffle\_chart, cmap=colormap)  plt.colorbar()  ax = plt.gca() **# get the axis**  ax.set\_xticks(np.arange(-.5, (width), 1), minor=True) **# set minor ticks**  ax.set\_yticks(np.arange(-.5, (height), 1), minor=True)  **# add gridlines based on minor ticks**  ax.grid(which='minor', color='w', linestyle='-', linewidth=2)  plt.xticks([])  plt.yticks([])  **# compute cumulative sum of individual categories to match color schemes between chart and legend**  values\_cumsum = np.cumsum(values)  total\_values = values\_cumsum[len(values\_cumsum) - 1]  **# create legend**  legend\_handles = []  for i, category in enumerate(categories):  if value\_sign == '%':  label\_str = category + ' (' + str(values[i]) + value\_sign + ')'  else:  label\_str = category + ' (' + value\_sign + str(values[i]) + ')'    color\_val = colormap(float(values\_cumsum[i])/total\_values)  legend\_handles.append(mpatches.Patch(color=color\_val, label=label\_str))  **# add legend to chart**  plt.legend(handles=legend\_handles, loc='lower center', ncol=len(categories), bbox\_to\_anchor=(0., -0.2, 0.95, .1))  *Now to create a waffle chart, all we have to do is call the function create\_waffle\_chart. Let's define the input parameters and call our function to create a waffle chart:*  # define the input parameters  width = 40 # width of chart  height = 10 # height of chart  categories = df\_dsn.index.values # categories  values = df\_dsn['Total'] # correponding values of categories  colormap = plt.cm.coolwarm # color map class  # call our function to create a waffle chart  create\_waffle\_chart(categories, values, height, width, colormap, value\_sign='')  *There seems to be a new Python package for generating waffle charts called* [*PyWaffle*](https://pypi.org/project/pywaffle/)*, but it looks like the repository is still being built. But feel free to check it out and play with it.*  *# Example from damo*  !pip install pywaffle matplotlib  from pywaffle import Waffle  import matplotlib.pyplot as plt  data = {'Democratic': 48, 'Republican': 46, 'Libertarian': 3}  fig = plt.figure(  FigureClass=Waffle,  rows=5,  values=data,  colors=("#983D3D", "#232066", "#DCB732"),  title={'label': 'Vote Percentage in 2016 US Presidential Election', 'loc': 'left'},  labels=["{0} ({1}%)".format(k, v) for k, v in data.items()],  legend={'loc': 'lower left', 'bbox\_to\_anchor': (0, -0.4), 'ncol': len(data), 'framealpha': 0},  #plot\_direction='NW',  tight=False,  figsize=(9, 6))  fig.set\_facecolor('#EEEEEE')  plt.show()  Word Clouds  *Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.*  *Luckily, a Python package already exists in Python for generating word clouds. The package, called word\_cloud was developed by Andreas Mueller. You can learn more about the package by following this link.*  *Let's use this package to learn how to generate a word cloud for a given text document. First, let's install the package.*  # install wordcloud  !conda install -c conda-forge wordcloud==1.4.1 --yes  ​# import package and its set of stopwords  from wordcloud import WordCloud, STOPWORDS  ​print ('Wordcloud is installed and imported!')  ​  *Word clouds are commonly used to perform high-level analysis and visualization of text data. Accordinly, let's digress from the immigration dataset and work with an example that involves analyzing text data. Let's try to analyze a short novel written by Lewis Carroll titled Alice's Adventures in Wonderland. Let's go ahead and download a .txt file of the novel.*  # download file and save as alice\_novel.txt  !wget --quiet https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/alice\_novel.txt  ​  # open the file and read it into a variable alice\_novel  alice\_novel = open('alice\_novel.txt', 'r').read()  print ('File downloaded and saved!')  *Next, let's use the stopwords that we imported. We use the function set to remove any redundant stopwords.*  stopwords = set(STOPWORDS)  *Create a word cloud object and generate a word cloud. For simplicity, let's generate a word cloud using only the first 2000 words in the novel.*  # instantiate a word cloud object  alice\_wc = WordCloud(background\_color='white', max\_words=2000, stopwords=stopwords)  ​  # generate the word cloud  alice\_wc.generate(alice\_novel)  # display the word cloud # Awesome! Now that the word cloud is created, let's visualize it.  plt.imshow(alice\_wc, interpolation='bilinear')  plt.axis('off')  plt.show()  *Interesting! So in the first 2000 words in the novel, the most common words are Alice, said, little, Queen, and so on. Let's resize the cloud so that we can see the less frequent words a little better.*  *However, said isn't really an informative word. So let's add it to our stopwords and re-generate the cloud.*  stopwords.add('said') # add the words said to stopwords  ​# re-generate the word cloud  alice\_wc.generate(alice\_novel)  ​# display the cloud  fig = plt.figure()  fig.set\_figwidth(14) # set width  fig.set\_figheight(18) # set height  ​plt.imshow(alice\_wc, interpolation='bilinear')  plt.axis('off')  plt.show()  *Excellent! This looks really interesting! Another cool thing you can implement with the word\_cloud package is superimposing the words onto a mask of any shape. Let's use a mask of Alice and her rabbit. We already created the mask for you, so let's go ahead and download it and call it alice\_mask.png.*  # download image  !wget --quiet https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Images/alice\_mask.png  # save mask to alice\_mask  alice\_mask = np.array(Image.open('alice\_mask.png'))  print('Image downloaded and saved!')  # Let's take a look at how the mask looks like.  fig = plt.figure()  fig.set\_figwidth(14) # set width  fig.set\_figheight(18) # set height  ​plt.imshow(alice\_mask, cmap=plt.cm.gray, interpolation='bilinear')  plt.axis('off')  plt.show()  *Shaping the word cloud according to the mask is straightforward using word\_cloud package. For simplicity, we will continue using the first 2000 words in the novel.*  # instantiate a word cloud object  alice\_wc = WordCloud(background\_color='white', max\_words=2000, mask=alice\_mask, stopwords=stopwords)  ​  # generate the word cloud  alice\_wc.generate(alice\_novel)  ​# display the word cloud  fig = plt.figure()  fig.set\_figwidth(14) # set width  fig.set\_figheight(18) # set height  ​plt.imshow(alice\_wc, interpolation='bilinear')  plt.axis('off')  plt.show()  Really impressive!  *Unfortunately, our immmigration data does not have any text data, but where there is a will there is a way. Let's generate sample text data from our immigration dataset, say text data of 90 words.*  #Let's recall how our data looks like, and what was the total immigration from 1980 to 2013?  df\_can.head()  total\_immigration = df\_can['Total'].sum()  *Using countries with single-word names, let's duplicate each country's name based on how much they contribute to the total immigration.*  max\_words = 90  word\_string = ''  for country in df\_can.index.values:  # check if country's name is a single-word name  if len(country.split(' ')) == 1:  repeat\_num\_times = int(df\_can.loc[country, 'Total']/float(total\_immigration)\*max\_words)  word\_string = word\_string + ((country + ' ') \* repeat\_num\_times)  #print(country,df\_can.loc[country, 'Total'],repeat\_num\_times)  # display the generated text  word\_string  *We are not dealing with any stopwords here, so there is no need to pass them when creating the word cloud.*  # create the word cloud  wordcloud = WordCloud(background\_color='white').generate(word\_string)  ​print('Word cloud created!')  # display the cloud  fig = plt.figure()  fig.set\_figwidth(14)  fig.set\_figheight(18)  plt.imshow(wordcloud, interpolation='bilinear')  plt.axis('off')  plt.show()  # display the cloud  fig = plt.figure()  fig.set\_figwidth(14)  fig.set\_figheight(18)  ​plt.imshow(wordcloud, interpolation='bilinear')  plt.axis('off')  plt.show()  *According to the above word cloud, it looks like the majority of the people who immigrated came from one of 15 countries that are displayed by the word cloud. One cool visual that you could build, is perhaps using the map of Canada and a mask and superimposing the word cloud on top of the map of Canada. That would be an interesting visual to build*  Regression Plots  *Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics. You can learn more about seaborn by following this link and more about seaborn regression plots by following this link.*  *In lab Pie Charts, Box Plots, Scatter Plots, and Bubble Plots, we learned how to create a scatter plot and then fit a regression line. It took ~20 lines of code to create the scatter plot along with the regression fit. In this final section, we will explore seaborn and see how efficient it is to create regression lines and fits using this library!*  *Let's first start with installing seaborn.*  # install seaborn  !conda install -c anaconda seaborn --yes  import seaborn as sns ​# import library  ​print('Seaborn installed and imported!')  *Create a new dataframe that stores that total number of landed immigrants to Canada per year from 1980 to 2013.*  # we can use the sum() method to get the total population per year  df\_tot = pd.DataFrame(df\_can[years].sum(axis=0))  ​# change the years to type float (useful for regression later on)  df\_tot.index = map(float, df\_tot.index)  ​  # reset the index to put in back in as a column in the df\_tot dataframe  df\_tot.reset\_index(inplace=True)  ​df\_tot.columns = ['year', 'total'] # rename columns  df\_tot.head()​ # view the final dataframe  # With seaborn, generating a regression plot is as simple as calling the regplot function.  import seaborn as sns  ax = sns.regplot(x='year', y='total', data=df\_tot)  *This is not magic; it is seaborn! You can also customize the color of the scatter plot and regression line. Let's change the color to green.*  import seaborn as sns  ax = sns.regplot(x='year', y='total', data=df\_tot, color='green')  *You can always customize the marker shape, so instead of circular markers, let's use '+'.*  import seaborn as sns  ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+')  *Let's blow up the plot a little bit so that it is more appealing to the sight.*  plt.figure(figsize=(15, 10))  ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+')  *And let's increase the size of markers so they match the new size of the figure, and add a title and x- and y-labels.*  plt.figure(figsize=(15, 10))  ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+', scatter\_kws={'s': 200})  ​ax.set(xlabel='Year', ylabel='Total Immigration') # add x- and y-labels  ax.set\_title('Total Immigration to Canada from 1980 - 2013') # add title  *And finally increase the font size of the tickmark labels, the title, and the x- and y-labels so they don't feel left out!*  plt.figure(figsize=(15, 10))  ​sns.set(font\_scale=1.5)  ​ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+', scatter\_kws={'s': 200})  ax.set(xlabel='Year', ylabel='Total Immigration')  ax.set\_title('Total Immigration to Canada from 1980 - 2013')  *Amazing! A complete scatter plot with a regression fit with 5 lines of code only. Isn't this really amazing?*  *If you are not a big fan of the purple background, you can easily change the style to a white plain background.*  plt.figure(figsize=(15, 10))  ​sns.set(font\_scale=1.5)  sns.set\_style('ticks') # change background to white background  ​ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+', scatter\_kws={'s': 200})  ax.set(xlabel='Year', ylabel='Total Immigration')  ax.set\_title('Total Immigration to Canada from 1980 - 2013')  *Or to a white background with gridlines.*  plt.figure(figsize=(15, 10))  ​sns.set(font\_scale=1.5)  sns.set\_style('whitegrid')  ​  ax = sns.regplot(x='year', y='total', data=df\_tot, color='green', marker='+', scatter\_kws={'s': 200})  ax.set(xlabel='Year', ylabel='Total Immigration')  ax.set\_title('Total Immigration to Canada from 1980 - 2013')  *Question: Use seaborn to create a scatter plot with a regression line to visualize the total immigration from Denmark, Sweden, and Norway to Canada from 1980 to 2013.*  df\_test = df\_can.loc[['Denmark', 'Sweden','Norway'],years].transpose()  print(df\_test.head())  df\_total = pd.DataFrame(df\_test.sum(axis=1))  df\_total.reset\_index(inplace=True)  print(df\_total.head())  df\_total.columns = ['year', 'total'] # rename columns  # change column year from str to int to create scatter plot  df\_total['year'] = df\_total['year'].astype(int)  # define figure size , background style and font size  plt.figure(figsize=(15, 10))  sns.set(font\_scale=1.5)  sns.set\_style('whitegrid')  # generate plot and add title and axes labels  ax = sns.regplot(x='year', y='total', data=df\_total, color='green', marker='+', scatter\_kws={'s': 200})  ax.set(xlabel='Year', ylabel='Total Immigration')  ax.set\_title('Total Immigrationn from Denmark, Sweden, and Norway to Canada from 1980 - 2013')  Generating Maps with Python  *In this lab, we will learn how to create maps for different objectives. To do that, we will part ways with Matplotlib and work with another Python visualization library, namely Folium. What is nice about Folium is that it was developed for the sole purpose of visualizing geospatial data. While other libraries are available to visualize geospatial data, such as plotly, they might have a cap on how many API calls you can make within a defined time frame. Folium, on the other hand, is completely free.*  ***Toolkits:*** *This lab heavily relies on pandas and Numpy for data wrangling, analysis, and visualization. The primary plotting library we will explore in this lab is Folium.*  ***Datasets:***   1. San Francisco Police Department Incidents for the year 2016 - Police Department Incidents from San Francisco public data portal. Incidents derived from San Francisco Police Department (SFPD) Crime Incident Reporting system. Updated daily, showing data for the entire year of 2016. Address and location has been anonymized by moving to mid-block or to an intersection. 2. Immigration to Canada from 1980 to 2013 - International migration flows to and from selected countries - The 2015 revision from United Nation's website. The dataset contains annual data on the flows of international migrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. For this lesson, we will focus on the Canadian Immigration data   Downloading and Prepping Data  # Import Primary Modules:  import numpy as np # useful for many scientific computing in Python  import pandas as pd # primary data structure library  Introduction to Folium  *Folium is a powerful Python library that helps you create several types of Leaflet maps. The fact that the Folium results are interactive makes this library very useful for dashboard building.*  *From the official Folium documentation page:*   * *Folium builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library. Manipulate your data in Python, then visualize it in on a Leaflet map via Folium.* * *Folium makes it easy to visualize data that's been manipulated in Python on an interactive Leaflet map. It enables both the binding of data to a map for choropleth visualizations as well as passing Vincent/Vega visualizations as markers on the map.* * *The library has a number of built-in tilesets from OpenStreetMap, Mapbox, and Stamen, and supports custom tilesets with Mapbox or Cloudmade API keys. Folium supports both GeoJSON and TopoJSON overlays, as well as the binding of data to those overlays to create choropleth maps with color-brewer color schemes.*   *Let's install Folium, It is not available by default. So, we first need to install it before we are able to import it.*  !conda install -c conda-forge folium=0.5.0 --yes  import folium # you may add - print('Folium installed and imported!')  *Generating the world map is straigtforward in Folium. You simply create a Folium Map object and then you display it. What is attactive about Folium maps is that they are interactive, so you can zoom into any region of interest despite the initial zoom level.*  # define the world map  world\_map = folium.Map()  world\_map ​# display world map  *Go ahead. Try zooming in and out of the rendered map above. You can customize this default definition of the world map by specifying the centre of your map and the intial zoom level.*  *All locations on a map are defined by their respective Latitude and Longitude values. So you can create a map and pass in a center of Latitude and Longitude values of [0, 0].*  *For a defined center, you can also define the intial zoom level into that location when the map is rendered. The higher the zoom level the more the map is zoomed into the center.*  *Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map.*    # define the world map centered around Canada with a low zoom level  world\_map = folium.Map(location=[56.130, -106.35], zoom\_start=4)  ​world\_map # display world map  *Let's create the map again with a higher zoom level*  # re-define with a higher zoom level  world\_map = folium.Map(location=[56.130, -106.35], zoom\_start=8)  ​world\_map # display world map  ​  *As you will see, the higher the zoom level the more the map is zoomed into the given center.*    *Let’s create a map of Mexico with a zoom level of 4.*  # define Mexico's geolocation coordinates  m\_lat = 23.6345  m\_long = -102.5528  # define the world map centered around Mexico  mexico\_map = folium.Map(location=[m\_lat, m\_long], zoom\_start=4)  mexico\_map # display world map  *Another cool feature of Folium is that you can generate different map styles.*  **A. Stamen Toner Maps**  *These are high-contrast B+W (black and white) maps. They are perfect for data mashups and exploring river meanders and coastal zones.*  # create a map with Stamen Toner map style  world\_map = folium.Map(location=[56.130, -106.35], zoom\_start=4, tiles='Stamen Toner')  world\_map ​# display map  **B. Stamen Terrain Maps**  *These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads.*  # create a map with Stamen Terrain map style  world\_map = folium.Map(location=[56.130, -106.35], zoom\_start=4, tiles='Stamen Terrain')  world\_map ​# display map  *Feel free to zoom in and out to see how this style compares to the default one.*  **C. Mapbox Bright Maps**  *These are maps that quite similar to the default style, except that the borders are not visible with a low zoom level. Furthermore, unlike the default style where country names are displayed in each country's native language, Mapbox Bright style displays all country names in English*.  *Let's create a world map with this style.*  # create a world map with a Mapbox Bright style.  world\_map = folium.Map(tiles='Mapbox Bright')  ​world\_map # display the map  *Zoom in and notice how the borders start showing as you zoom in, and the displayed country names are in English.*    Create a map of Mexico to visualize its hill shading and natural vegetation. Use a zoom level of 6.  # define Mexico's geolocation coordinates  mexico\_latitude = 23.6345  mexico\_longitude = -102.5528  # define the world map  mexico\_map = folium.Map(location=[mexico\_latitude, mexico\_longitude], zoom\_start=6,tiles='Stamen Terrain')  mexico\_map # display world map  Maps with Markers  Let's download and import the data on police department incidents using pandas read\_csv() method.  # Download the dataset and read it into a pandas dataframe:  df\_incidents = pd.read\_csv('https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/Police\_Department\_Incidents\_-\_Previous\_Year\_\_2016\_.csv')  ​print('Dataset downloaded and read into a pandas dataframe!')  # Let's take a look at the first five items in our dataset.  df\_incidents.head()  *So each row consists of 13 features:*   1. **IncidntNum:** Incident Number 2. **Category:** Category of crime or incident 3. **Descript:** Description of the crime or incident 4. **DayOfWeek:** The day of week on which the incident occurred 5. **Date:** The Date on which the incident occurred 6. **Time:** The time of day on which the incident occurred 7. **PdDistrict:** The police department district 8. **Resolution:** The resolution of the crime in terms whether the perpetrator was arrested or not 9. **Address:** The closest address to where the incident took place 10. **X:** The longitude value of the crime location 11. **Y:** The latitude value of the crime location 12. **Location:** A tuple of the latitude and the longitude values 13. **PdId:** The police department ID   df\_incidents.shape# Let's find out how many entries there are in our dataset.  *So the dataframe consists of 150,500 crimes, which took place in the year 2016. In order to reduce computational cost, let's just work with the first 100 incidents in this dataset.*  # get the first 100 crimes in the df\_incidents dataframe  limit = 100  df\_incidents = df\_incidents.iloc[0:limit, :]  df\_incidents.shape # Let's confirm that our dataframe now consists only of 100 crimes.  *Now that we reduced the data a little bit, let's visualize where these crimes took place in the city of San Francisco. We will use the default style and we will initialize the zoom level to 12.*  # San Francisco latitude and longitude values  latitude = 37.77  longitude = -122.42  # create map and display it  sanfran\_map = folium.Map(location=[latitude, longitude], zoom\_start=12)  ​sanfran\_map # display the map of San Francisco  Now let's superimpose the locations of the crimes onto the map. The way to do that in Folium is to create a feature group with its own features and style and then add it to the sanfran\_map.  # instantiate a feature group for the incidents in the dataframe  incidents = folium.map.FeatureGroup()  ​# loop through the 100 crimes and add each to the incidents feature group  for lat, lng, in zip(df\_incidents.Y, df\_incidents.X):  incidents.add\_child(  folium.features.CircleMarker(  [lat, lng],  radius=5, # define the circle markers size  color='yellow',  fill=True,  fill\_color='blue',  fill\_opacity=0.6  )  )  ​# add incidents to map  sanfran\_map.add\_child(incidents)  *Note here sanfran\_map = folium.Map(location=[37.77, -122.42], zoom\_start=12) which we created earlier*  *You can also add some pop-up text that would get displayed when you hover over a marker. Let's make each marker display the category of the crime when hovered over.*  # instantiate a feature group for the incidents in the dataframe  incidents = folium.map.FeatureGroup()  ​# loop through the 100 crimes and add each to the incidents feature group  for lat, lng, in zip(df\_incidents.Y, df\_incidents.X):  incidents.add\_child(  folium.features.CircleMarker(  [lat, lng],  radius=5, # define the circle markers size  color='yellow',  fill=True,  fill\_color='blue',  fill\_opacity=0.6  )  )  ​# add pop-up text to each marker on the map  latitudes = list(df\_incidents.Y)  longitudes = list(df\_incidents.X)  labels = list(df\_incidents.Category)  ​  for lat, lng, label in zip(latitudes, longitudes, labels):  folium.Marker([lat, lng], popup=label).add\_to(sanfran\_map)  # add incidents to map  sanfran\_map.add\_child(incidents)  *Isn't this really cool? Now you are able to know what crime category occurred at each marker.*  *If you find the map to be so congested will all these markers, there are two remedies to this problem. The simpler solution is to remove these location markers and just add the text to the circle markers themselves as follows:*  sanfran\_map = folium.Map(location=[latitude, longitude], zoom\_start=12)  for lat, lng, label in zip(df\_incidents.Y, df\_incidents.X, df\_incidents.Category):  folium.features.CircleMarker(  [lat, lng],  radius=5, # define the circle markers size  color='yellow',  fill=True,  popup=label,  fill\_color='blue',  fill\_opacity=0.6  ).add\_to(sanfran\_map)  ​  sanfran\_map # show map  *The other proper remedy is to group the markers into different clusters. Each cluster is then represented by the number of crimes in each neighborhood. These clusters can be thought of as pockets of San Francisco which you can then analyze separately.*  *To implement this, we start off by instantiating a MarkerCluster object and adding all the data points in the dataframe to this object.*  from folium import plugins  # let's start again with a clean copy of the map of San Francisco  sanfran\_map = folium.Map(location = [latitude, longitude], zoom\_start = 12)  # instantiate a mark cluster object for the incidents in the dataframe  incidents = plugins.MarkerCluster().add\_to(sanfran\_map)  # loop to add each data point to the mark cluster  for lat, lng, label, in zip(df\_incidents.Y, df\_incidents.X, df\_incidents.Category):  folium.Marker(  location=[lat, lng],  icon=None,  popup=label,  ).add\_to(incidents)  sanfran\_map # display map  *Notice how when you zoom out all the way, all markers are grouped into one cluster, the global cluster, of 100 markers or crimes, which is the total number of crimes in our dataframe. Once you start zooming in, the global cluster will start breaking up into smaller clusters. Zooming in all the way will result in individual markers.*  Choropleth Maps  *A Choropleth map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per-capita income. The choropleth map provides an easy way to visualize how a measurement varies across a geographic area or it shows the level of variability within a region. Below is a Choropleth map of the US depicting the population by square mile per state.*  *Now, let's create our own Choropleth map of the world depicting immigration from various countries to Canada.*  *Let's first download and import our primary Canadian immigration dataset using pandas read\_excel() method*.  #!conda install -c anaconda xlrd --yes  # Download the dataset and read it into a pandas dataframe:  df\_can = pd.read\_excel('https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/Canada.xlsx',  sheet\_name='Canada by Citizenship',  skiprows=range(20),  skipfooter=2)  ​print('Data downloaded and read into a dataframe!')  df\_can.head() # Let's take a look at the first five items in our dataset.  print(df\_can.shape) # print the dimensions of the dataframe  # clean up the dataset to remove unnecessary columns (eg. REG)  df\_can.drop(['AREA','REG','DEV','Type','Coverage'], axis=1, inplace=True)  ​# let's rename the columns so that they make sense  df\_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)  ​# for sake of consistency, let's also make all column labels of type string  df\_can.columns = list(map(str, df\_can.columns))  df\_can['Total'] = df\_can.sum(axis=1) ​# add total column  ​# years that we will be using in this lesson - useful for plotting later on  years = list(map(str, range(1980, 2014)))  print ('data dimensions:', df\_can.shape)  df\_can.head() # Let's take a look at the first five items of our cleaned dataframe.  *In order to create a Choropleth map, we need a GeoJSON file that defines the areas/boundaries of the state, county, or country that we are interested in. In our case, since we are endeavoring to create a world map, we want a GeoJSON that defines the boundaries of all world countries. Let's go ahead and download it. Let's name it world\_countries.json.*  # download countries geojson file  !wget --quiet [https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data\_Files/world\_countries.json -O world\_countries.json](https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data_Files/world_countries.json%20-O%20world_countries.json)  print('GeoJSON file downloaded!')  *Now that we have the GeoJSON file, let's create a world map, centered around [0, 0] latitude and longitude values, with an intial zoom level of 2, and using Mapbox Bright style.*  world\_geo = r'world\_countries.json' # geojson file  ​# create a plain world map  world\_map = folium.Map(location=[0, 0], zoom\_start=2, tiles='Mapbox Bright')  *And now to create a Choropleth map, we will use the choropleth method with the following main parameters:*   * ***geo\_data****, which is the GeoJSON file.* * ***data****, which is the dataframe containing the data.* * ***columns****, which represents the columns in the dataframe that will be used to create the Choropleth map.* * ***key\_on****, which is the key or variable in the GeoJSON file that contains the name of the variable of interest. To determine that, you will need to open the GeoJSON file using any text editor and note the name of the key or variable that contains the name of the countries, since the countries are our variable of interest. In this case, name is the key in the GeoJSON file that contains the name of the countries. Note that this key is case\_sensitive, so you need to pass exactly as it exists in the GeoJSON file.*   # generate choropleth map using the total immigration of each country to Canada from 1980 to 2013  world\_map.choropleth(  geo\_data=world\_geo,  data=df\_can,  columns=['Country', 'Total'],  key\_on='feature.properties.name',  fill\_color='YlOrRd',  fill\_opacity=0.7,  line\_opacity=0.2,  legend\_name='Immigration to Canada'  )  world\_map ​# display map  *As per our Choropleth map legend, the darker the color of a country and the closer the color to red, the higher the number of immigrants from that country. Accordingly, the highest immigration over the course of 33 years (from 1980 to 2013) was from China, India, and the Philippines, followed by Poland, Pakistan, and interestingly, the US.*  Notice how the legend is displaying a negative boundary or threshold. Let's fix that by defining our own thresholds and starting with 0 instead of -6,918!  world\_geo = r'world\_countries.json'  ​# create a numpy array of length 6 and has linear spacing from the minium to the maximum of total immigration  threshold\_scale = np.linspace(df\_can['Total'].min(),  df\_can['Total'].max(),  6, dtype=int)  threshold\_scale = threshold\_scale.tolist() # change the numpy array to a list  threshold\_scale[-1] = threshold\_scale[-1] + 1 # make sure that the last value of the list is greater than the maximum  ​# let Folium determine the scale.  world\_map = folium.Map(location=[0, 0], zoom\_start=2, tiles='Mapbox Bright')  world\_map.choropleth(  geo\_data=world\_geo,  data=df\_can,  columns=['Country', 'Total'],  key\_on='feature.properties.name',  threshold\_scale=threshold\_scale,  fill\_color='PuBuGn’, # changed YlOrRd  fill\_opacity=0.7,  line\_opacity=0.2,  legend\_name='Immigration to Canada',  reset=True  )  world\_map |
| Final Assignment : Generating Maps with Python  **Introduction:**  A survey was conducted to gauge an audience interest in different data science topics, namely:  Big Data (Spark / Hadoop)  Data Analysis / Statistics  Data Journalism  Data Visualization  Deep Learning  Machine Learning  The participants had three options for each topic: Very Interested, Somewhat interested, and Not interested. 2,233 respondents completed the survey.  The survey results have been saved in a csv file and can be accessed through this link: https://cocl.us/datascience\_survey\_data.  If you examine the csv file, you will find that the first column represents the data science topics and the first row the choices for each topic.  *Task1:* *Use the pandas read\_csv method to read the csv file into a pandas dataframe, that looks like the following:*  In order to read the data into a dataframe like the above, one way to do that is to use the index\_col parameter in order to load the first column as the index of the dataframe. Here is the documentation on the pandas read\_csv method: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read\_csv.html  *Once you have succeeded in creating the above dataframe, please upload a screenshot of your dataframe with the actual numbers.* ***(5 marks)***  Task1 - Downloading and Prepping Data  # Import Primary Modules:  import numpy as np # useful for many scientific computing in Python  import pandas as pd # primary data structure library  # 1. Download the dataset and read it into a pandas dataframe:!  df\_survey = pd.read\_csv('https://cocl.us/datascience\_survey\_data/Topic\_Survey\_Assignment.csv')  print('Dataset downloaded and read into a pandas dataframe!')  # Rename the column and Set - index column  df\_survey.columns = ['', 'Very interested', 'Somewhat interested','Not interested']  df\_survey.set\_index('',inplace = True)  # View the Dataframe  df\_survey[0:]  *Task2:* *Use the artist layer of Matplotlib, replicate the bar chart below to visualize the percentage of the respondents' interest in the different data science topics surveyed.*  To create this bar chart, you can follow the following steps:   * Sort the dataframe in descending order of Very interested. * Convert the numbers into percentages of the total number of respondents. Recall that 2,233 respondents completed the survey. Round percentages to 2 decimal places. * As for the chart: * use a figure size of (20, 8), * bar width of 0.8, * use color #5cb85c for the Very interested bars, color #5bc0de for the Somewhat interested bars, and color #d9534f for the Not interested bars, * use font size 14 for the bar labels, percentages, and legend, * use font size 16 for the title, and, * display the percentages above the bars as shown above, and remove the left, top, and right borders.   *Once you are satisfied with your chart, please upload a screenshot of your plot. (5 marks)*  Task2 – Creating Bar Graph  #df\_survey['Total'] = df\_survey.sum(axis=1)  #df\_survey.drop(['Total'], axis=1, inplace=True)  #new\_df = pd.DataFrame({'1980s': df\_80s, '1990s': df\_90s, '2000s':df\_00s})  #df\_survey[0:]  #R1=len(df\_survey)  #for index, value in enumerate(df\_survey):  # df\_survey[value] = round(df\_survey[value]/2233\*100,2)  df\_survey.sort\_values(by='Very interested', ascending=False, inplace=True)  df\_survey\_pct=round(df\_survey[:]/2233\*100,2)  df\_survey\_pct  # Let's start by importing Matplotlib and Matplotlib.pyplot as follows:  %matplotlib inline  import matplotlib as mpl  import matplotlib.pyplot as plt  print ('Matplotlib version: ', mpl.\_\_version\_\_)  # Creat list of label:  df\_survey\_pct1 = df\_survey\_pct.T  label\_list=[]  for index, value in enumerate(df\_survey\_pct1):  label\_list.extend(df\_survey\_pct1[value])  label\_list  # Plot Graph  df\_survey.sort\_values('Very interested',ascending=False, axis=0, inplace=True)  df\_survey\_pct=round(df\_survey[:]/2233\*100,2)  colors\_list = ['#5cb85c', '#5bc0de', '#d9534f']  ax = df\_survey\_pct.plot(kind='bar', figsize=(20, 8), color = colors\_list,width=.8, fontsize=14)  ax.set\_yticks([])  ax.set\_title('Percentage of Respondents’ Intrest in Data Science Areas',fontsize=16)  ax.spines['left'].set\_visible(False)  ax.spines['right'].set\_visible(False)  ax.spines['top'].set\_visible(False)  #ax.set\_xlabel('Data Science Topics',fontsize=14)  #ax.set\_ylabel('Topics',fontsize=14)  #ax.set\_xticks(np.arange(0, 6, 1), minor=True)  ax.legend(loc='upper right', fontsize=14)  # Annotate value labels to each country  for index, value in enumerate(label\_list):  #label = '%11.2f%% '%value # format int with commas  #label = format(str(value), '%d'%value)  label = value  if index in [0,3,6,9,12,15]:  val= 0.26  elif index in [1,4,7,10,13,16]:  val = 0.33  else:  val = 0.39  # place text at the top of bar (index is x, y is bar to make it fit within the bar)  ax.annotate(label, xy=(index/3.00-val,value+2),va='center',ha='center',xycoords='data',fontsize=14)  #print(index,value,label)  *Task3: Create a Choropleth Map*  *In the final lab, we created a map with markers to explore crime rate in San Francisco, California. In this question, you are required to create a Choropleth map to visualize crime in San Francisco.*  *Before you are ready to start building the map, let's restructure the data so that it is in the right format for the Choropleth map. Essentially, you will need to create a dataframe that lists each neighborhood in San Francisco along with the corresponding total number of crimes. Based on the San Francisco crime dataset, you will find that San Francisco consists of 10 main neighborhoods, namely:*  *Central, Southern, Bayview, Mission, Park, Richmond, Ingleside, Taraval, Northern, and Tenderloin.*  *Convert the San Francisco dataset, which you can also find here, https://cocl.us/sanfran\_crime\_dataset, into a pandas dataframe, like the one shown below, that represents the total number of crimes in each neighborhood.*  *Once you are happy with your dataframe, upload a image of your dataframe. (5 marks)*  Task3a: Creating crime dataframe for map  # 1. Download the dataset and read it into a pandas dataframe:!  df\_sfcrime = pd.read\_csv('https://cocl.us/sanfran\_crime\_dataset/name.csv')  print('Dataset downloaded and read into a pandas dataframe!')  # 2. Rename the column and Set - index column  #df.set\_index('',inplace = True) #df\_sfcrime.dtypes #df\_sfcrime['PdDistrict'].value\_counts()  #df\_sfcrime[(df\_sfcrime['Category']!= "NON-CRIMINAL")].groupby('PdDistrict',axis=0).count()  #df\_survey['Total'] = df\_survey.sum(axis=1)  #df\_survey.drop(['Total'], axis=1, inplace=True)  df\_sftot\_crime = df\_sfcrime.groupby('PdDistrict',axis=0).count()[['IncidntNum']]  df\_sftot\_crime.sort\_values(by='IncidntNum',ascending=False,inplace=True)  df\_sftot\_crime.reset\_index(inplace=True)  df\_sftot\_crime.columns = ['Neighborhood','Count']  df\_sftot\_crime  *Task3: Create a Choropleth Map continued..*  *Now you should be ready to proceed with creating the Choropleth map.*  *As you learned in the Choropleth maps lab, you will need a GeoJSON file that marks the boundaries of the different neighborhoods in San Francisco. In order to save you the hassle of looking for the right file, I already downloaded it for you and I am making it available via this link: https://cocl.us/sanfran\_geojson.*  *For the map, make sure that:*  *it is centred around San Francisco,*   * *you use a zoom level of 12* * *you use fill\_color = 'YlOrRd'* * *you define fill\_opacity = 0.7* * *you define line\_opacity=0.2* * *you define a legend and use the default threshold scale.*   *If you follow the lab on Choropleth maps and use the GeoJSON correctly, you should be able to creat the following map:*  *Once you are ready to submit your map, please upload a screenshot of your Choropleth map. (5 marks)*  Task3b: Plot Choropleth Map  # Install folium for map creation  !conda install -c conda-forge folium=0.5.0 --yes  import folium  ​print('Folium installed and imported!')  #define the world map for Test  #world\_map = folium.Map()  # define the world map centered around Canada with a low zoom level  #sanfran\_map = folium.Map(location=[37.773972, -122.431297], zoom\_start=12)  # display world map  #sanfran\_map  ​  # Download countries geojson file  #!wget --quiet [link](https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data_Files/world_countries.json) -O world\_countries.json  !wget --quiet https://cocl.us/sanfran\_geojson/san-francisco.geojson -O san-francisco.geojson  print('GeoJSON file downloaded!')  #world\_geo = r'world\_countries.json' # geojson file  sf\_geo = r'san-francisco.geojson' # geojson file  # Create a plain world map  sf\_map = folium.Map(location=[37.773972, -122.431297], zoom\_start=12)  # Generate choropleth map using the total crime incident in each neighborhood of Sanfrancisco Area  sf\_map.choropleth(  geo\_data=sf\_geo,  data=df\_sftot\_crime,  columns=['Neighborhood','Count'],  key\_on='feature.properties.DISTRICT',  fill\_color='YlOrRd',  fill\_opacity=0.7,  line\_opacity=0.2,  legend\_name='Crime Rate in San Francisco',  reset=True  )  ​  # display map  sf\_map |