IBM Data Science Professional Certificate

Course 8: Data Visualization with Python

Week 1: Introduction to Data Visualization Tools

Intro to Data Visualization

Why use visuals?

- for exploratory data analysis
- communicate data clearly
- share unbiased representation of data
- support recommendations to different stakeholders

Best Practices:

- Less is more effective, attractive, and impactive.

Intro to Matplotlib

Matplotlib:

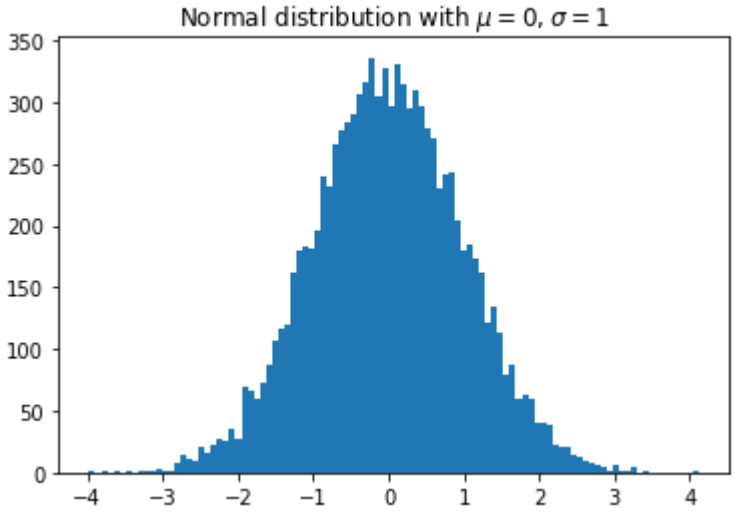
- one of the most widely used and most popular data visualization libraries in Python
- Composed of 3 main layers:
    1. Backend layer (figurecanvas, renderer, event)
        - Figurecanvas encompasses the area on which the figure is drawn
        - Renderer knows how to draw on the figurecanvas
        - Event handles user inputs such as keyboard strokes and mouse clicks
    2. Artist layer (artist)
        - artist knows how to use the renderer to draw on the canvas
        - two types of artist objects:
            - primitive (line2d, rectangle, circle, text)
            - composite (axis, tick, axes, figure)
        - each composite artist may contain other composite or primitive artists.
    3. Scripting layer (pyplot)
        - comprised mainly of pyplot, a scripting interface that is lighter than the artist layer

In [1]:
```
# generate a histogram using the artist layer

# import figurecanvas and figure artist
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
%matplotlib inline

# create figure object
fig = Figure()
```

```
canvas = FigureCanvas(fig)

# create 10000 random numbers with numpy
import numpy as np
x = np.random.randn(10000)

# create axes artist
ax = fig.add_subplot(111)

# create histogram
ax.hist(x, 100)

# add title to figure and save it
ax.set_title('Normal distribution with $\mu = 0, \sigma = 1$')
fig.savefig('matplotlib_histogram.png')
```

In [2]:

```
# generate the same histogram using the scripting layer
import matplotlib.pyplot as plt

x = np.random.randn(10000)
plt.hist(x, 100)
plt.title('Normal distribution with $\mu = 0, \sigma = 1$')
plt.savefig('matplotlib_histogram.png')
plt.show()
```



Basic Plotting with Matplotlib

Matplotlib:

- most graphs can be created with the .plot() function
- Use magic function %matplotlib inline to generate the graph within the browser and not a separate window
  - this has a the limitation that you cannot alter a figure once it's been rendered. Instead you will have to generate a fresh one.
- Magic function %matplotlib notebook overcomes the above limitation.
  - with this function in place, if a plt function is called, it checks if an active figure exists. Any functions you called will be applied to thsi active figure. If no figure exists, a new one is

made.

- Pandas has a built-in implementation of matplotlib.
  - plotting in pandas is as easy as calling the plot function on a pandas series or dataframe.
    - df.plot(kind = 'line')
    - df['col'].plot(kind = 'hist')

Line Plots

Line Plots:

- a type of plot which displays information as a series of data points called 'markers' connected by straight line segments
- useful to visualize a dataset over a period of time

Syntax:

- df.plot(kind='line')

Lab: Intro to Matplotlib and Line Plots

Dataset.

In [3]:
```
import numpy as np  # useful for many scientific computing in Python
import pandas as pd # primary data structure library
```

In [4]:
```
# install openpyxl, which pandas requires to read excel files.

#! pip3 install openpyxl
```

In [5]:
```
# read the data and store into a dataframe

# skip the 20 rows of non-data
df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSki
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2)

print('Data read into a pandas dataframe!')
```

```
Data read into a pandas dataframe!
```

In [6]:
```
df_can.head()
# tip: You can specify the number of rows you'd like to see as follows: df_can.head(10)
```

Out[6]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...

5 rows × 51 columns

In [7]:
```
# get summary info of the dataframe
df_can.info(verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Columns: 51 entries, Type to Unnamed: 50
dtypes: float64(8), int64(37), object(6)
memory usage: 77.8+ KB
```

In [8]:
```
# more summary info
df_can.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 51 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   Type         195 non-null    object
 1   Coverage     195 non-null    object
 2   OdName       195 non-null    object
 3   AREA         195 non-null    int64 
 4   AreaName     195 non-null    object
 5   REG          195 non-null    int64 
 6   RegName      195 non-null    object
 7   DEV          195 non-null    int64 
 8   DevName      195 non-null    object
 9   1980         195 non-null    int64 
 10  1981         195 non-null    int64 
 11  1982         195 non-null    int64 
 12  1983         195 non-null    int64 
 13  1984         195 non-null    int64 
 14  1985         195 non-null    int64 
 15  1986         195 non-null    int64 
 16  1987         195 non-null    int64 
 17  1988         195 non-null    int64 
 18  1989         195 non-null    int64 
 19  1990         195 non-null    int64 
 20  1991         195 non-null    int64 
 21  1992         195 non-null    int64 
 22  1993         195 non-null    int64 
 23  1994         195 non-null    int64 
 24  1995         195 non-null    int64 
 25  1996         195 non-null    int64 
```

```
 26  1997         195 non-null    int64
 27  1998         195 non-null    int64
 28  1999         195 non-null    int64
 29  2000         195 non-null    int64
 30  2001         195 non-null    int64
 31  2002         195 non-null    int64
 32  2003         195 non-null    int64
 33  2004         195 non-null    int64
 34  2005         195 non-null    int64
 35  2006         195 non-null    int64
 36  2007         195 non-null    int64
 37  2008         195 non-null    int64
 38  2009         195 non-null    int64
 39  2010         195 non-null    int64
 40  2011         195 non-null    int64
 41  2012         195 non-null    int64
 42  2013         195 non-null    int64
 43  Unnamed: 43  0 non-null      float64
 44  Unnamed: 44  0 non-null      float64
 45  Unnamed: 45  0 non-null      float64
 46  Unnamed: 46  0 non-null      float64
 47  Unnamed: 47  0 non-null      float64
 48  Unnamed: 48  0 non-null      float64
 49  Unnamed: 49  0 non-null      float64
 50  Unnamed: 50  0 non-null      float64
dtypes: float64(8), int64(37), object(6)
memory usage: 77.8+ KB
```

In [9]:

```
# get columns as a list
df_can.columns.tolist()
```

Out[9]:

```
['Type',
 'Coverage',
 'OdName',
 'AREA',
 'AreaName',
 'REG',
 'RegName',
 'DEV',
 'DevName',
 1980,
 1981,
 1982,
 1983,
 1984,
 1985,
 1986,
 1987,
 1988,
 1989,
 1990,
 1991,
 1992,
 1993,
 1994,
 1995,
 1996,
 1997,
 1998,
 1999,
 2000,
 2001,
 2002,
 2003,
```

```
 2004,
 2005,
 2006,
 2007,
 2008,
 2009,
 2010,
 2011,
 2012,
 2013,
 'Unnamed: 43',
 'Unnamed: 44',
 'Unnamed: 45',
 'Unnamed: 46',
 'Unnamed: 47',
 'Unnamed: 48',
 'Unnamed: 49',
 'Unnamed: 50']
```

In [10]:
```
df_can.index.tolist()
```

Out[10]:
```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 26,
 27,
 28,
 29,
 30,
 31,
 32,
 33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
```

```
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
```

```
108,
109,
110,
111,
112,
113,
114,
115,
116,
117,
118,
119,
120,
121,
122,
123,
124,
125,
126,
127,
128,
129,
130,
131,
132,
133,
134,
135,
136,
137,
138,
139,
140,
141,
142,
143,
144,
145,
146,
147,
148,
149,
150,
151,
152,
153,
154,
155,
156,
157,
158,
159,
160,
161,
162,
163,
164,
165,
166,
167,
168,
169,
170,
171,
172,
```

```
173,
174,
175,
176,
177,
178,
179,
180,
181,
182,
183,
184,
185,
186,
187,
188,
189,
190,
191,
192,
193,
194]
```

In [11]:
```
print(type(df_can.columns.tolist()))
print(type(df_can.index.tolist()))
```

```
<class 'list'>
<class 'list'>
```

In [12]:
```
# view dimensions of the dataframe
df_can.shape
```

Out[12]: (195, 51)

In [13]:
```
# clean the data set to remove unneeded columns
# in pandas, axis = 0 represents rows and axis = 1 represents columns
df_can.drop(['AREA','REG','DEV','Type','Coverage'], axis=1, inplace=True)
df_can.head(2)
```

Out[13]:

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2012	201
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2635	200
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	620	60

2 rows × 46 columns

In [14]:
```
# drop unnamed columns
df_can.drop(['Unnamed: 43', "Unnamed: 44", "Unnamed: 45", "Unnamed: 46", "Unnamed: 47",
df_can.head(3)
```

Out[14]:

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	200

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	200
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	343
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	122
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3616	362

3 rows × 38 columns

In [15]:
```
df_can.columns.tolist()
```

Out[15]:
```
['OdName',
 'AreaName',
 'RegName',
 'DevName',
 1980,
 1981,
 1982,
 1983,
 1984,
 1985,
 1986,
 1987,
 1988,
 1989,
 1990,
 1991,
 1992,
 1993,
 1994,
 1995,
 1996,
 1997,
 1998,
 1999,
 2000,
 2001,
 2002,
 2003,
 2004,
 2005,
 2006,
 2007,
 2008,
 2009,
 2010,
 2011,
 2012,
 2013]
```

In [16]:
```
# rename columns to make them make more sense
df_can.rename(columns={'OdName':'Country', 'AreaName':'Continent', 'RegName':'Region'},
df_can.columns
```

Out[16]:
```
Index([  'Country', 'Continent',    'Region',   'DevName',        1980,
```

```
            1981,       1982,       1983,       1984,       1985,
            1986,       1987,       1988,       1989,       1990,
            1991,       1992,       1993,       1994,       1995,
            1996,       1997,       1998,       1999,       2000,
            2001,       2002,       2003,       2004,       2005,
            2006,       2007,       2008,       2009,       2010,
            2011,       2012,       2013],
      dtype='object')
```

In [17]:
```
# add a totals column which sums the total immigrants by country over the entire period
df_can['Total'] = df_can.sum(axis=1)
```

In [18]:
```
# check to see how many null objects are in the dataset
df_can.isnull().sum()
```

Out[18]:
```
Country      0
Continent    0
Region       0
DevName      0
1980         0
1981         0
1982         0
1983         0
1984         0
1985         0
1986         0
1987         0
1988         0
1989         0
1990         0
1991         0
1992         0
1993         0
1994         0
1995         0
1996         0
1997         0
1998         0
1999         0
2000         0
2001         0
2002         0
2003         0
2004         0
2005         0
2006         0
2007         0
2008         0
2009         0
2010         0
2011         0
2012         0
2013         0
Total        0
dtype: int64
```

In [19]:
```
# get statistical summary data
df_can.describe()
```

Out[19]:

	1980	1981	1982	1983	1984	1985	1986

	1980	1981	1982	1983	1984	1985	1986
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
mean	508.394872	566.989744	534.723077	387.435897	376.497436	358.861538	441.271795
std	1949.588546	2152.643752	1866.997511	1204.333597	1198.246371	1079.309600	1225.576630
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000
50%	13.000000	10.000000	11.000000	12.000000	13.000000	17.000000	18.000000
75%	251.500000	295.500000	275.000000	173.000000	181.000000	197.000000	254.000000
max	22045.000000	24796.000000	20620.000000	10015.000000	10170.000000	9564.000000	9470.000000

8 rows × 35 columns

In [20]:
```
# filter on the list of countries
df_can.Country  # returns a series
```

Out[20]:
```
0         Afghanistan
1             Albania
2             Algeria
3      American Samoa
4             Andorra
            ...
190          Viet Nam
191    Western Sahara
192             Yemen
193            Zambia
194          Zimbabwe
Name: Country, Length: 195, dtype: object
```

In [21]:
```
# filtering on the list of countries for certain years
df_can[['Country', 1980, 1981, 1982, 1983, 1984, 1985]] # returns a dataframe
# notice that 'Country' is string, and the years are integers.
# for the sake of consistency, we will convert all column names to string later on.
```

Out[21]:

	Country	1980	1981	1982	1983	1984	1985
0	Afghanistan	16	39	39	47	71	340
1	Albania	1	0	0	0	0	0
2	Algeria	80	67	71	69	63	44
3	American Samoa	0	1	0	0	0	0
4	Andorra	0	0	0	0	0	0
...	...	...	...	...	...	...	...
190	Viet Nam	1191	1829	2162	3404	7583	5907
191	Western Sahara	0	0	0	0	0	0
192	Yemen	1	2	1	6	0	18

	Country	1980	1981	1982	1983	1984	1985
193	Zambia	11	17	11	7	16	9
194	Zimbabwe	72	114	102	44	32	29

195 rows × 7 columns

In [22]:
```
# set the country as the index
df_can.set_index('Country', inplace=True)
# tip: The opposite of set is reset. So to reset the index, we can use df_can.reset_ind
```

In [23]:
```
df_can.head(3)
```

Out[23]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2
Country													
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	

3 rows × 38 columns

In [24]:
```
# select rows with .loc and .iloc
# 1. the full row data (all columns)
df_can.loc['Japan']
```

Out[24]:
```
Continent               Asia
Region           Eastern Asia
DevName     Developed regions
1980                      701
1981                      756
1982                      598
1983                      309
1984                      246
1985                      198
1986                      248
1987                      422
1988                      324
1989                      494
1990                      379
1991                      506
1992                      605
1993                      907
1994                      956
1995                      826
1996                      994
1997                      924
1998                      897
1999                     1083
```

```
2000                     1010
2001                     1092
2002                      806
2003                      817
2004                      973
2005                     1067
2006                     1212
2007                     1250
2008                     1284
2009                     1194
2010                     1168
2011                     1265
2012                     1214
2013                      982
Total                   27707
Name: Japan, dtype: object
```

In [25]:
```
# alternate methods
df_can.iloc[87]
```

Out[25]:
```
Continent                 Asia
Region            Eastern Asia
DevName      Developed regions
1980                       701
1981                       756
1982                       598
1983                       309
1984                       246
1985                       198
1986                       248
1987                       422
1988                       324
1989                       494
1990                       379
1991                       506
1992                       605
1993                       907
1994                       956
1995                       826
1996                       994
1997                       924
1998                       897
1999                      1083
2000                      1010
2001                      1092
2002                       806
2003                       817
2004                       973
2005                      1067
2006                      1212
2007                      1250
2008                      1284
2009                      1194
2010                      1168
2011                      1265
2012                      1214
2013                       982
Total                    27707
Name: Japan, dtype: object
```

In [26]:
```
# alternate method
df_can[df_can.index == 'Japan']
```

Out[26]:

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006
Japan	Asia	Eastern Asia	Developed regions	701	756	598	309	246	198	248	...	1067	1212

1 rows × 38 columns

In [27]:
```
# 2. for year 2013
df_can.loc['Japan', 2013]
```

Out[27]:
```
982
```

In [28]:
```
# alternate method
# year 2013 is the last column, with a positional index of 36
df_can.iloc[87, 36]
```

Out[28]:
```
982
```

In [29]:
```
# 3. for years 1980 to 1985
df_can.loc['Japan', [1980, 1981, 1982, 1983, 1984, 1984]]
```

Out[29]:
```
1980    701
1981    756
1982    598
1983    309
1984    246
1984    246
Name: Japan, dtype: object
```

In [30]:
```
# Alternative Method
df_can.iloc[87, [3, 4, 5, 6, 7, 8]]
```

Out[30]:
```
1980    701
1981    756
1982    598
1983    309
1984    246
1985    198
Name: Japan, dtype: object
```

In [31]:
```
# convert all column names to strings to avoid confusion
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
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Out[31]:
```
[None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None,
```

```
 None,
 None,
 None]
```

In [32]:
```
# declare a varaible that allows us to call upon the full range
# useful for plotting later on
years = list(map(str, range(1980, 2014)))
years
```

Out[32]:
```
['1980',
 '1981',
 '1982',
 '1983',
 '1984',
 '1985',
 '1986',
 '1987',
 '1988',
 '1989',
 '1990',
 '1991',
 '1992',
 '1993',
 '1994',
 '1995',
 '1996',
 '1997',
 '1998',
 '1999',
 '2000',
 '2001',
 '2002',
 '2003',
 '2004',
 '2005',
 '2006',
 '2007',
 '2008',
 '2009',
 '2010',
 '2011',
 '2012',
 '2013']
```

In [33]:
```
# filter to show asian countries
# 1. create the condition boolean series
condition = df_can['Continent'] == 'Asia'
print(condition)
```

```
Country
Afghanistan        True
Albania           False
Algeria           False
American Samoa    False
Andorra           False
                  ...
Viet Nam           True
Western Sahara    False
Yemen              True
Zambia            False
Zimbabwe          False
Name: Continent, Length: 195, dtype: bool
```

In [34]:
```
# 2. pass this condition into the dataFrame
df_can[condition]
```

Out[34]:

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	200
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	343
Armenia	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	22
Azerbaijan	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	35
Bahrain	Asia	Western Asia	Developing regions	0	2	1	1	1	3	0	...	1
Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	417
Bhutan	Asia	Southern Asia	Developing regions	0	0	0	0	1	0	0	...	
Brunei Darussalam	Asia	South-Eastern Asia	Developing regions	79	6	8	2	2	4	12	...	
Cambodia	Asia	South-Eastern Asia	Developing regions	12	19	26	33	10	7	8	...	37
China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1960	...	4258
China, Hong Kong Special Administrative Region	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	72
China, Macao Special Administrative Region	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	2
Cyprus	Asia	Western Asia	Developing regions	132	128	84	46	46	43	48	...	
Democratic People's Republic of Korea	Asia	Eastern Asia	Developing regions	1	1	3	1	4	3	0	...	1
Georgia	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	11
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	3621
Indonesia	Asia	South-Eastern Asia	Developing regions	186	178	252	115	123	100	127	...	63

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	200
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	583
Iraq	Asia	Western Asia	Developing regions	262	245	260	380	428	231	265	...	222
Israel	Asia	Western Asia	Developing regions	1403	1711	1334	541	446	680	1212	...	244
Japan	Asia	Eastern Asia	Developed regions	701	756	598	309	246	198	248	...	106
Jordan	Asia	Western Asia	Developing regions	177	160	155	113	102	179	181	...	194
Kazakhstan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	50
Kuwait	Asia	Western Asia	Developing regions	1	0	8	2	1	4	4	...	6
Kyrgyzstan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	17
Lao People's Democratic Republic	Asia	South-Eastern Asia	Developing regions	11	6	16	16	7	17	21	...	4
Lebanon	Asia	Western Asia	Developing regions	1409	1119	1159	789	1253	1683	2576	...	370
Malaysia	Asia	South-Eastern Asia	Developing regions	786	816	813	448	384	374	425	...	59
Maldives	Asia	Southern Asia	Developing regions	0	0	0	1	0	0	0	...	
Mongolia	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	5
Myanmar	Asia	South-Eastern Asia	Developing regions	80	62	46	31	41	23	18	...	21
Nepal	Asia	Southern Asia	Developing regions	1	1	6	1	2	4	13	...	60
Oman	Asia	Western Asia	Developing regions	0	0	0	8	0	0	0	...	1
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	1431
Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	1813
Qatar	Asia	Western Asia	Developing regions	0	0	0	0	0	0	1	...	1

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	200
Republic of Korea	Asia	Eastern Asia	Developing regions	1011	1456	1572	1081	847	962	1208	...	583
Saudi Arabia	Asia	Western Asia	Developing regions	0	0	1	4	1	2	5	...	19
Singapore	Asia	South-Eastern Asia	Developing regions	241	301	337	169	128	139	205	...	39
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	493
State of Palestine	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	45
Syrian Arab Republic	Asia	Western Asia	Developing regions	315	419	409	269	264	385	493	...	145
Tajikistan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	8
Thailand	Asia	South-Eastern Asia	Developing regions	56	53	113	65	82	66	78	...	57
Turkey	Asia	Western Asia	Developing regions	481	874	706	280	338	202	257	...	206
Turkmenistan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	4
United Arab Emirates	Asia	Western Asia	Developing regions	0	2	2	1	2	0	5	...	3
Uzbekistan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	33
Viet Nam	Asia	South-Eastern Asia	Developing regions	1191	1829	2162	3404	7583	5907	2741	...	185
Yemen	Asia	Western Asia	Developing regions	1	2	1	6	0	18	7	...	16

49 rows × 38 columns

In [35]:
```
# we can 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
# don't forget to enclose the two conditions in parentheses
```

Out[35]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005
Country												
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436
Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	4171
Bhutan	Asia	Southern Asia	Developing regions	0	0	0	0	1	0	0	...	5
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	5837
Maldives	Asia	Southern Asia	Developing regions	0	0	0	1	0	0	0	...	0
Nepal	Asia	Southern Asia	Developing regions	1	1	6	1	2	4	13	...	607
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	4930

9 rows × 38 columns

In [36]:
```
# review changes made to the dataframe
print('data dimensions:', df_can.shape)
print(df_can.columns)
df_can.head(2)
```

```
data dimensions: (195, 38)
Index(['Continent', 'Region', 'DevName', '1980', '1981', '1982', '1983',
       '1984', '1985', '1986', '1987', '1988', '1989', '1990', '1991', '1992',
       '1993', '1994', '1995', '1996', '1997', '1998', '1999', '2000', '2001',
       '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010',
       '2011', '2012', '2013', 'Total'],
      dtype='object')
```

Out[36]:

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2
Country													
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	

2 rows × 38 columns

In [37]:
```
# we are using the inline backend
%matplotlib inline

# import libraries
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [38]:
```
# check if matplotlib is loaded
print('Matplotlib version: ', mpl.__version__)  # >= 3.4.2
```

```
Matplotlib version:  3.4.2
```

In [39]:
```
# apply a style
print(plt.style.available)
mpl.style.use(['ggplot']) # optional: for ggplot-like style
```

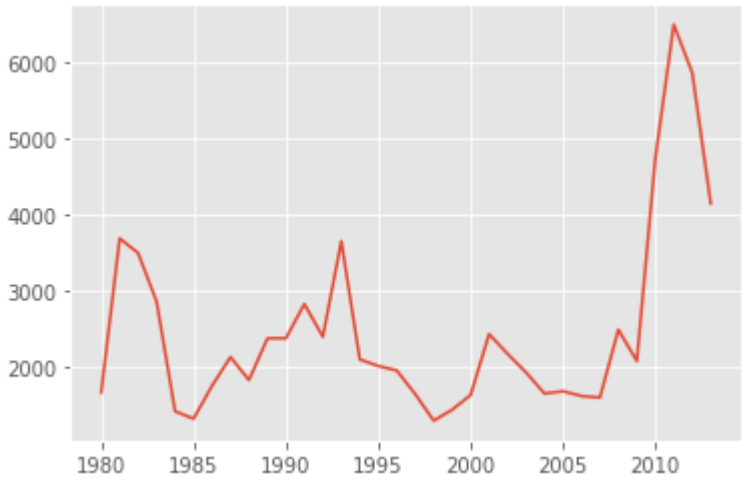
```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast',
'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-colorbli
nd', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seabor
n-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seab
orn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid', 'tableau-colorblind1
0']
```

In [40]:
```
# plot a line graph of immigration from haiti
haiti = df_can.loc['Haiti', years] # passing in years 1980 - 2013 to exclude the 'total
haiti.head()
```

Out[40]:
```
1980    1666
1981    3692
1982    3498
1983    2860
1984    1418
Name: Haiti, dtype: object
```

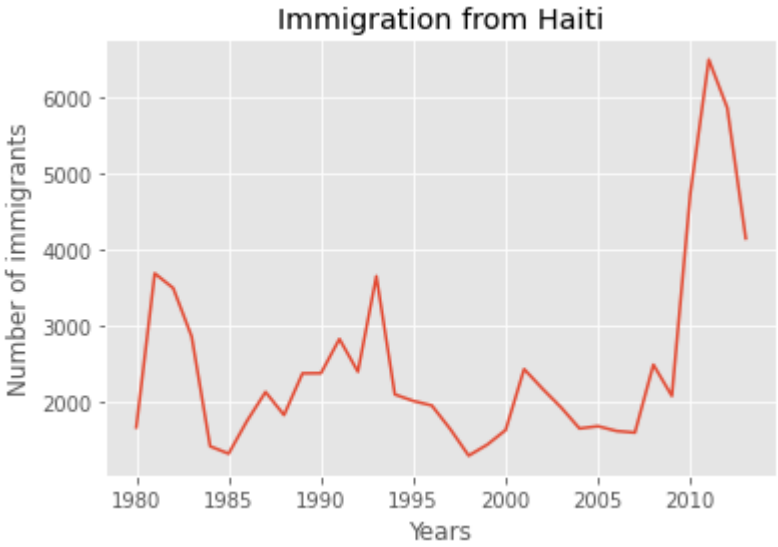
In [41]:
```
# plot the line
haiti.plot()
```

Out[41]:
```
<AxesSubplot:>
```



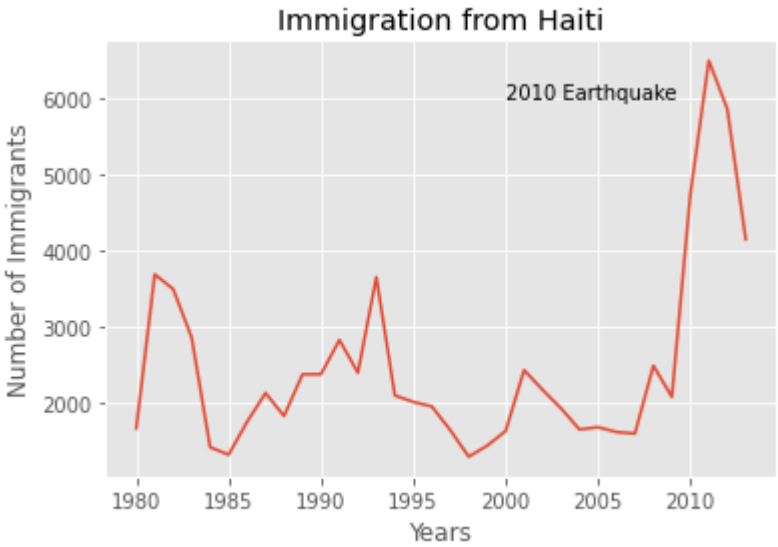
In [42]:
```
# change the type of index values to integer for plotting
haiti.index = haiti.index.map(int) # let's change the index values of Haiti to type int
```

```
haiti.plot(kind='line')

# add labels
plt.title('Immigration from Haiti')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')

plt.show() # need this line to show the updates made to the figure
```



In [43]:

```
# annotate the spike on the graph with .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') # 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') # years stored as type int
```

```
   We will cover advanced annotation methods in later modules.
```

In [44]:
```
# get the data set for china and india
df_CI = df_can.loc[['India', 'China'], years]
df_CI
```

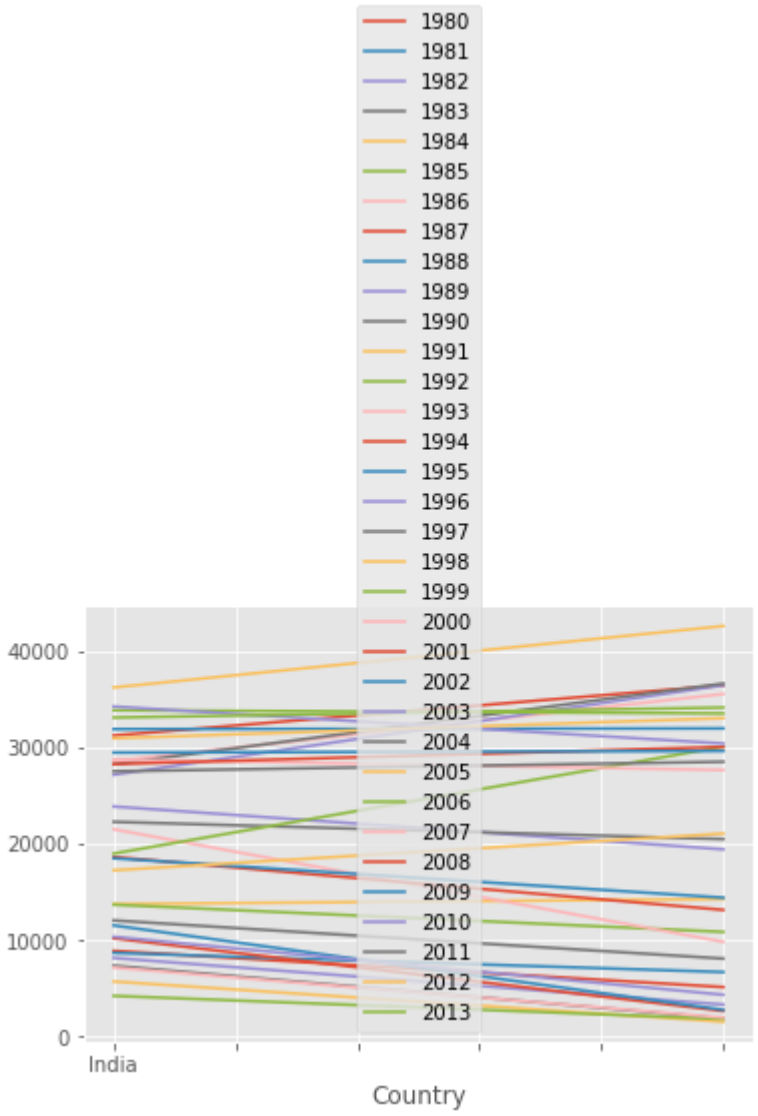
Out[44]:

Country	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2004	2005	2006
India	8880	8670	8147	7338	5704	4211	7150	10189	11522	10343	...	28235	36210	33848
China	5123	6682	3308	1863	1527	1816	1960	2643	2758	4323	...	36619	42584	33518

2 rows × 34 columns

In [45]:
```
# plot the graph
df_CI.plot(kind='line')
# recall that pandas plots the indices on x-axis and columns as individual lines on y-a
```

Out[45]:
```
<AxesSubplot:xlabel='Country'>
```



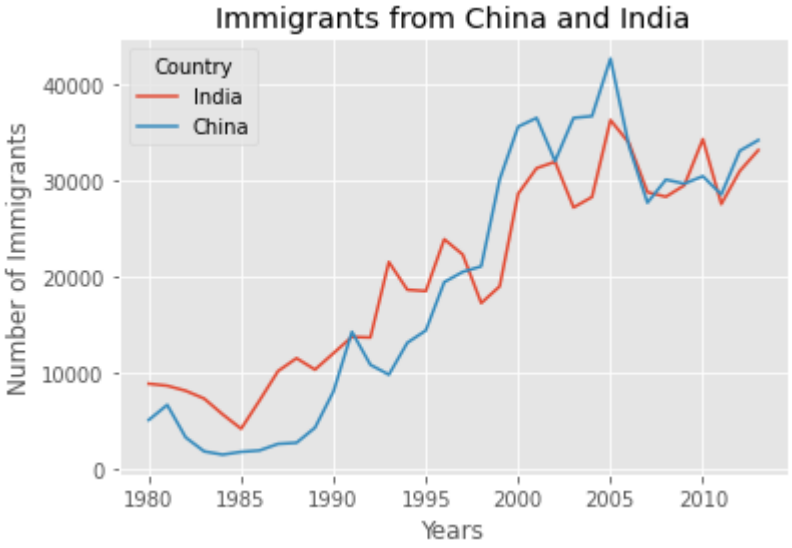
In [46]:
```
# fix the dataframe by transposing to swap the rows and columns
df_CI = df_CI.transpose()
df_CI.head()
```

Out[46]:

Country	India	China
1980	8880	5123
1981	8670	6682
1982	8147	3308
1983	7338	1863
1984	5704	1527

In [47]:
```
# graph the plot
df_CI.index = df_CI.index.map(int) # let's change the index values of df_CI to type int
df_CI.plot(kind='line')

# add labels
plt.title('Immigrants from China and India')
plt.ylabel('Number of Immigrants')
```

```
plt.xlabel('Years')

plt.show()
```



Note: How come we didn't need to transpose Haiti's dataframe before plotting (like we did for 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))
print(haiti.head(5))
```

> class 'pandas.core.series.Series'
> 1980 1666
> 1981 3692
> 1982 3498
> 1983 2860
> 1984 1418
> Name: Haiti, dtype: int64

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.

In [48]:

```
# compare the 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 cumulat
#We will sort on this column to get our top 5 countries using pandas sort_values() meth

inplace = True # paramemter saves the changes to the original df_can dataframe
df_can.sort_values(by='Total', ascending=False, axis=0, inplace=True)

# get the top 5 entries
df_top5 = df_can.head(5)

# transpose the dataframe
```

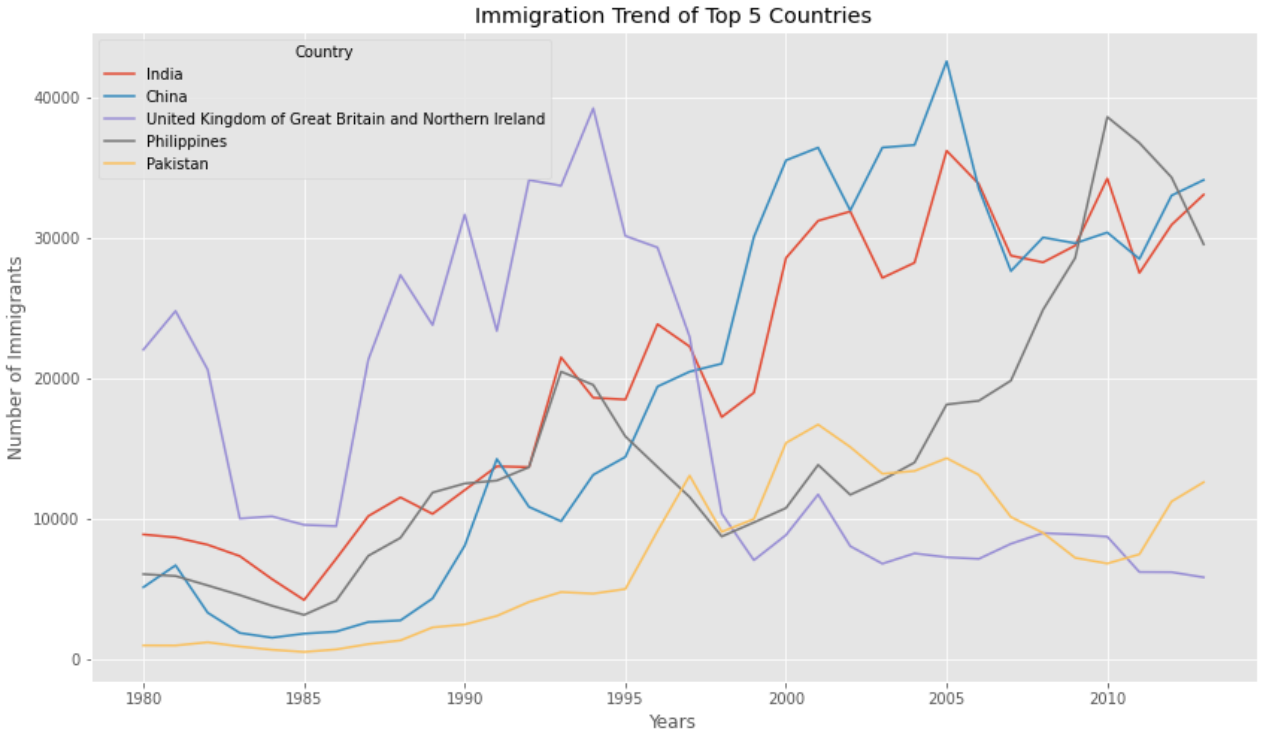
```
df_top5 = df_top5[years].transpose()

print(df_top5)
```

```
Country  India  China  United Kingdom of Great Britain and Northern Ireland  \
1980      8880   5123                                                22045
1981      8670   6682                                                24796
1982      8147   3308                                                20620
1983      7338   1863                                                10015
1984      5704   1527                                                10170
1985      4211   1816                                                 9564
1986      7150   1960                                                 9470
1987     10189   2643                                                21337
1988     11522   2758                                                27359
1989     10343   4323                                                23795
1990     12041   8076                                                31668
1991     13734  14255                                                23380
1992     13673  10846                                                34123
1993     21496   9817                                                33720
1994     18620  13128                                                39231
1995     18489  14398                                                30145
1996     23859  19415                                                29322
1997     22268  20475                                                22965
1998     17241  21049                                                10367
1999     18974  30069                                                 7045
2000     28572  35529                                                 8840
2001     31223  36434                                                11728
2002     31889  31961                                                 8046
2003     27155  36439                                                 6797
2004     28235  36619                                                 7533
2005     36210  42584                                                 7258
2006     33848  33518                                                 7140
2007     28742  27642                                                 8216
2008     28261  30037                                                 8979
2009     29456  29622                                                 8876
2010     34235  30391                                                 8724
2011     27509  28502                                                 6204
2012     30933  33024                                                 6195
2013     33087  34129                                                 5827

Country  Philippines  Pakistan
1980            6051       978
1981            5921       972
1982            5249      1201
1983            4562       900
1984            3801       668
1985            3150       514
1986            4166       691
1987            7360      1072
1988            8639      1334
1989           11865      2261
1990           12509      2470
1991           12718      3079
1992           13670      4071
1993           20479      4777
1994           19532      4666
1995           15864      4994
1996           13692      9125
1997           11549     13073
1998            8735      9068
1999            9734      9979
2000           10763     15400
2001           13836     16708
2002           11707     15110
```

```
2003          12758     13205
2004          14004     13399
2005          18139     14314
2006          18400     13127
2007          19837     10124
2008          24887      8994
2009          28573      7217
2010          38617      6811
2011          36765      7468
2012          34315     11227
2013          29544     12603
```

In [49]:

```
#Step 2: Plot the dataframe. To make the plot more readeable, we will change the size u
df_top5.index = df_top5.index.map(int) # let's change the index values of df_top5 to ty
df_top5.plot(kind='line', figsize=(14, 8)) # pass a tuple (x, y) size

# add labels
plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



Week 2: Basic and Specialized Visualization Tools

Area Plots

Area Plot:

- an extension of a line plot
- depicts cumulative totals using numbers or percentages over time
- first sort your data in descending order, then grab just the rows you want to plot, then plot
  - if needed, transpose the dataframe

- NOTE: area plots are stacked by default

Syntax:

- df.sort_values(['col'], ascending = False, axis = 0, inplace = True)
- new_df = df.head(5)
- new_df = new_df['cols'].transpose()
- new_df.plot(kind = 'area')

Histograms

Histogram:

- a way of representing the frequency distribution of a variable

Syntax (using just pyplot):

- df['col'].plot(kind = 'hist')

Syntax (using numpy and pyplot):

- count, bin_edges = np.histogram(df['col'])
- df['col'].plot(kind = 'hist', xticks = bin_edges) Numpy will partition the spread of the data into bins of equal width.

Bar Charts

Bar chart:

- commonly used to compare the values of a variable at a given point of time
- you can select a subgroup of your dataset to turn into a chart.

Syntax:

- x = list(map(str, range(col1, col2)))
- new_df = df.loc('col', x)
- new_df.plot(kind = 'bar')

Lab: Basic Visualization Tools

In [50]:
```
# let's examine the types of the column labels
all(isinstance(column, str) for column in df_can.columns)
# for consistency, we want all column labels to be string
```

Out[50]: True

In [51]:
```
# use the inline backend to generate the plots within the browser
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
mpl.style.use('ggplot')  # optional: for ggplot-like style

# check for latest version of Matplotlib
print('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

```
Matplotlib version:  3.4.2
```

Area Plots

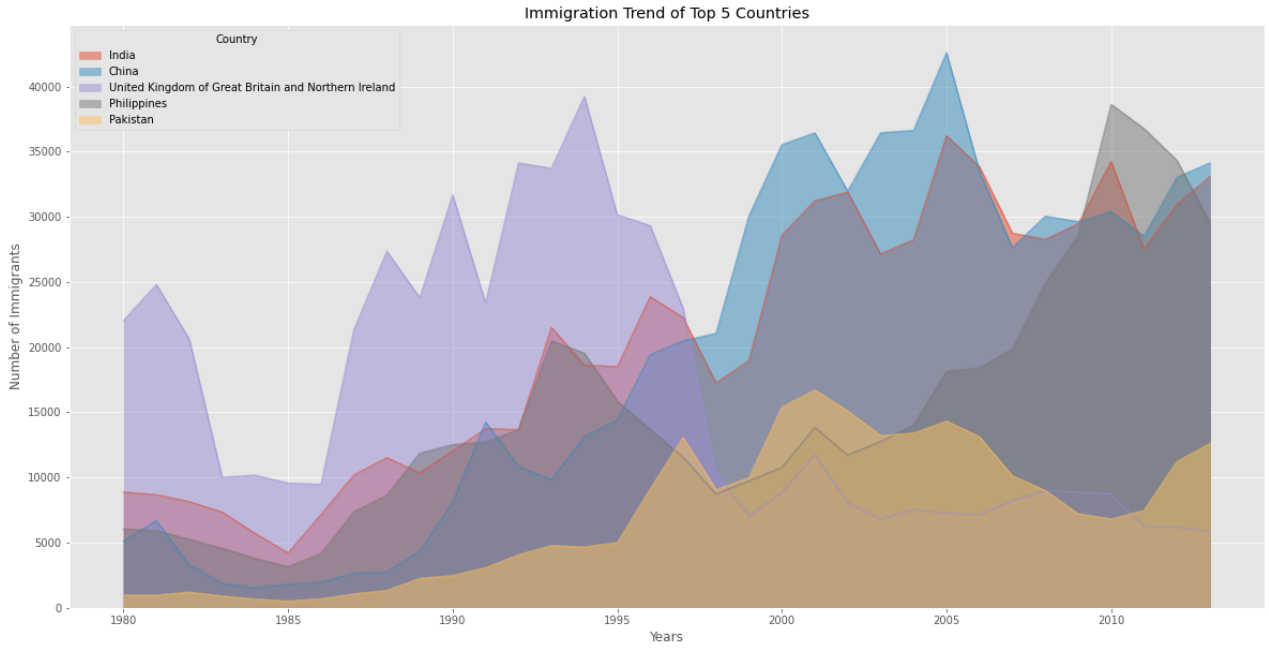
Area plots are stacked by default. And to produce a stacked area plot, each column must be either all positive or all negative values (any `NaN`, i.e. not a number, values will default to 0). To produce an unstacked plot, set parameter `stacked` to value `False`.

In [52]:
```
# sort the values in descending order
df_can.sort_values(['Total'], ascending=False, axis=0, inplace=True)

# get the top 5 entries
df_top5 = df_can.head()

# transpose the dataframe
df_top5 = df_top5[years].transpose()

df_top5.head()
```

Out[52]:

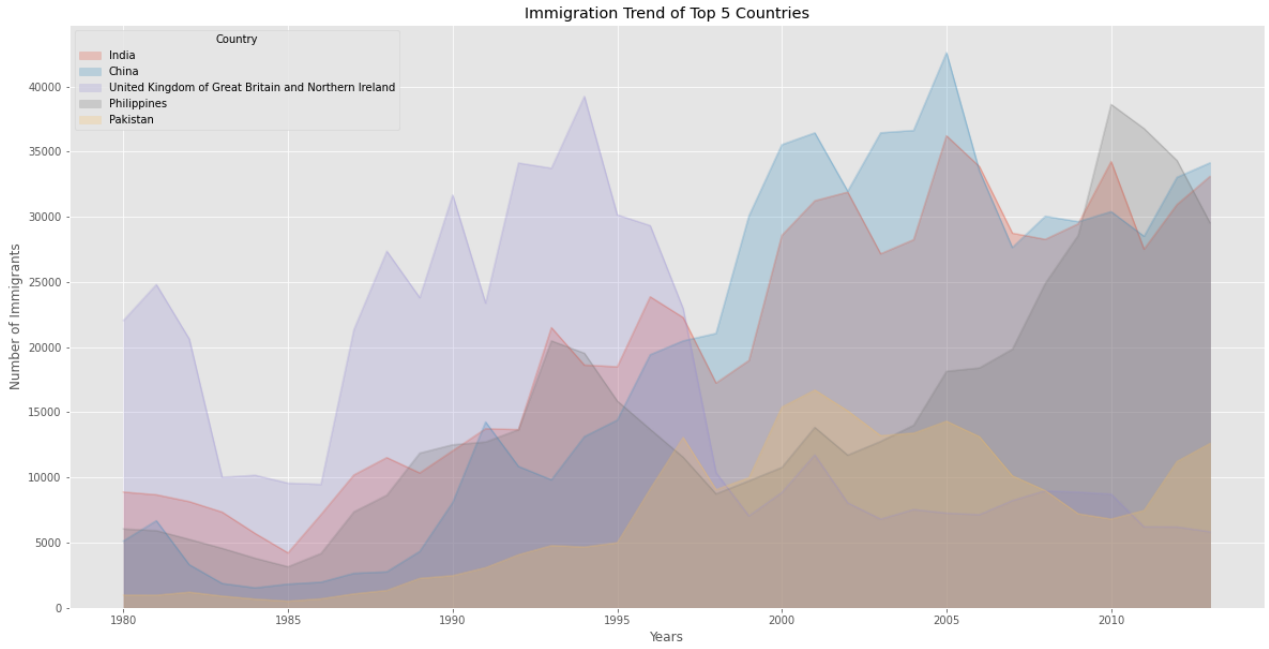
Country	India	China	United Kingdom of Great Britain and Northern Ireland	Philippines	Pakistan
1980	8880	5123	22045	6051	978
1981	8670	6682	24796	5921	972
1982	8147	3308	20620	5249	1201
1983	7338	1863	10015	4562	900
1984	5704	1527	10170	3801	668

In [53]:
```
# let's change the index values of df_top5 to type integer for plotting
df_top5.index = df_top5.index.map(int)
df_top5.plot(kind='area',
             stacked=False,
             figsize=(20, 10))  # pass a tuple (x, y) size

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
# the unstacked plot has default transparency (alpha) = .5
```



In [54]:

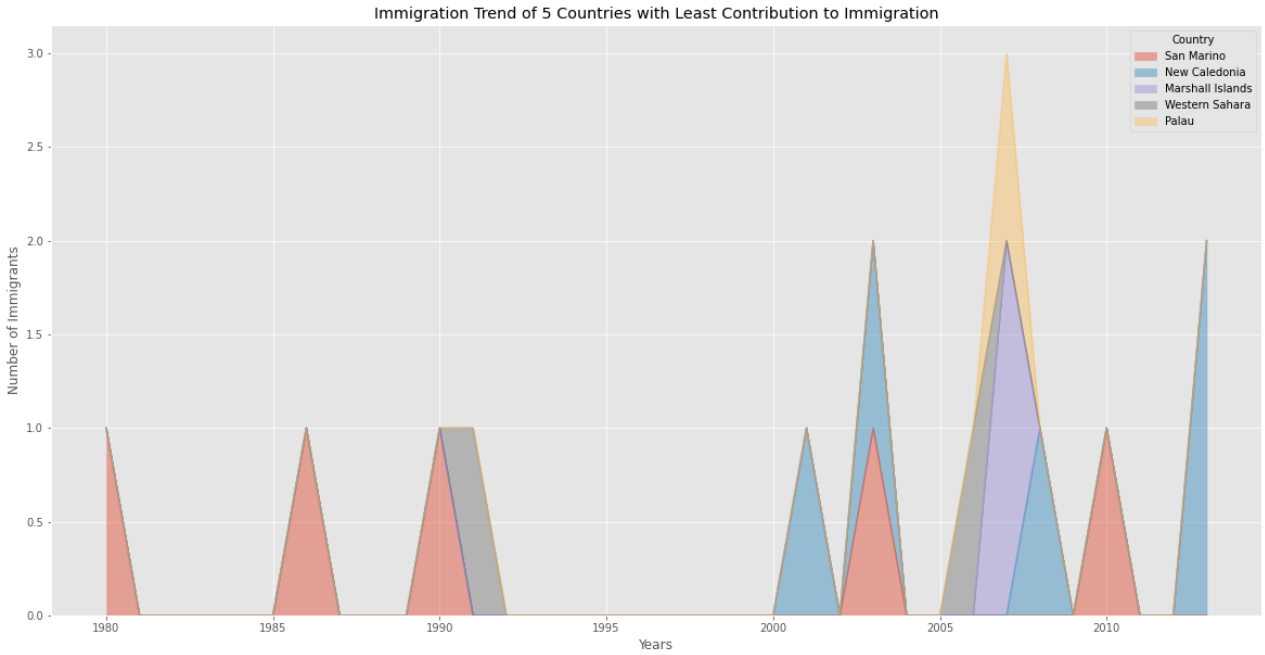
```
# change alpha value
df_top5.plot(kind='area',
             alpha=0.25,  # 0 - 1, default value alpha = 0.5
             stacked=False,
             figsize=(20, 10))

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



In [55]:

```
# use scripting layer to create a stacked plot

# get the 5 countries with the least contribution
df_least5 = df_can.tail(5)
```

```
# transpose the dataframe
df_least5 = df_least5[years].transpose()
df_least5.head()

df_least5.index = df_least5.index.map(int) # let's change the index values of df_least5
df_least5.plot(kind='area', alpha=0.45, figsize=(20, 10))

plt.title('Immigration Trend of 5 Countries with Least Contribution to Immigration')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```

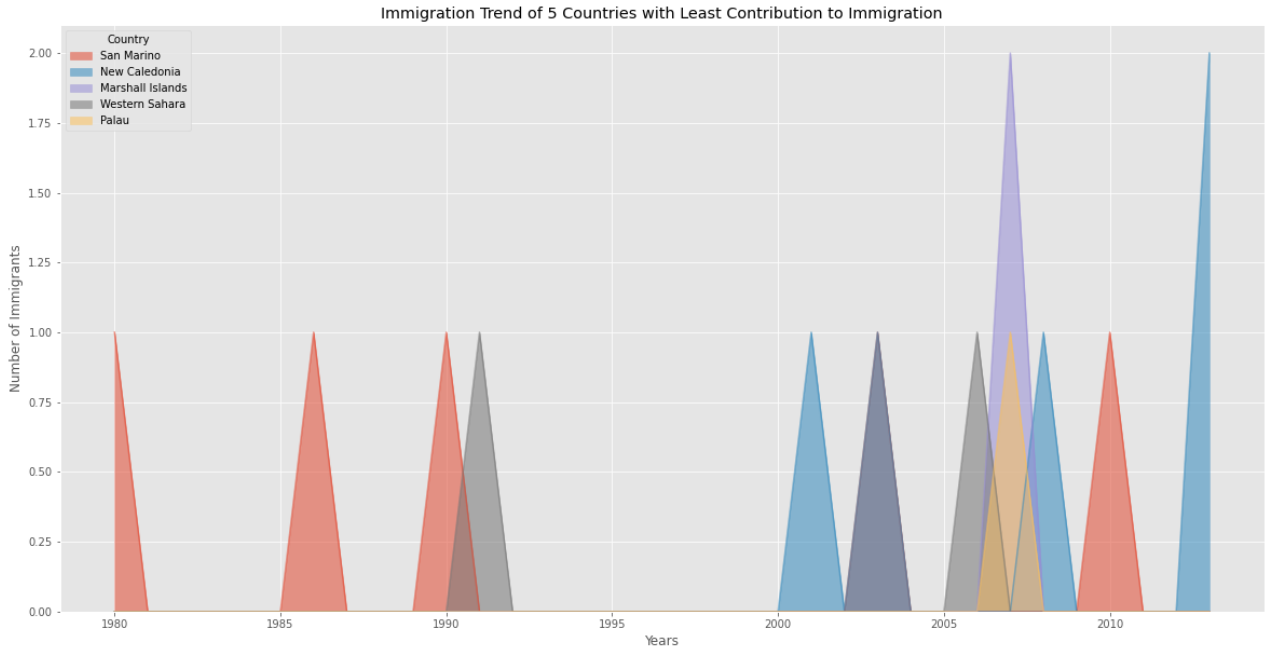


In [56]:

```
# use artist layer to create an unstacked plot

# get the 5 countries with the least contribution
df_least5 = df_can.tail(5)

# transpose the dataframe
df_least5 = df_least5[years].transpose()

df_least5.head()

df_least5.index = df_least5.index.map(int) # let's change the index values of df_least5

ax = df_least5.plot(kind='area', alpha=0.55, stacked=False, figsize=(20, 10))

ax.set_title('Immigration Trend of 5 Countries with Least Contribution to Immigration')
ax.set_ylabel('Number of Immigrants')
ax.set_xlabel('Years')
```

Out[56]:

```
Text(0.5, 0, 'Years')
```



Histograms

Side Note: We could use `df_can['2013'].plot.hist()`, instead. In fact, throughout this lesson, using `some_data.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.

In [57]:
```
# let's quickly view the 2013 data
df_can['2013'].head()
```

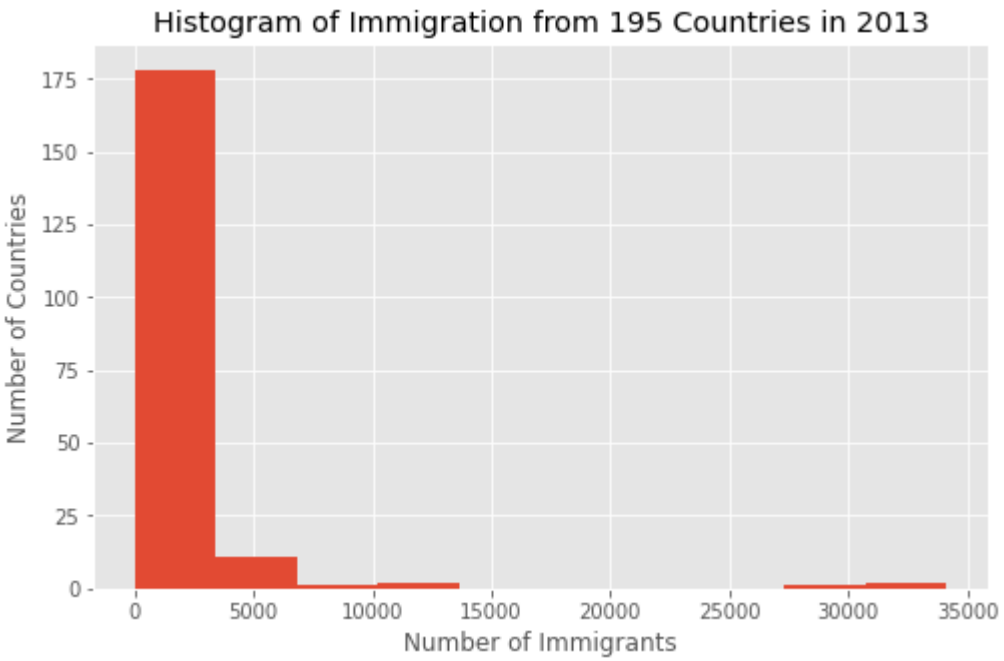
Out[57]:
```
Country
India                                              33087
China                                              34129
United Kingdom of Great Britain and Northern Ireland    5827
Philippines                                        29544
Pakistan                                           12603
Name: 2013, dtype: int64
```

In [58]:
```
# np.histogram returns 2 values, count and bin_edges
count, bin_edges = np.histogram(df_can['2013'])

print("count: ", count) # frequency count
print("bin edges: ", bin_edges) # bin ranges, default = 10 bins
```

```
count:  [178  11   1   2   0   0   0   0   1   2]
bin edges:  [    0.   3412.9  6825.8 10238.7 13651.6 17064.5 20477.4 23890.3 27303.2
 30716.1 34129. ]
```

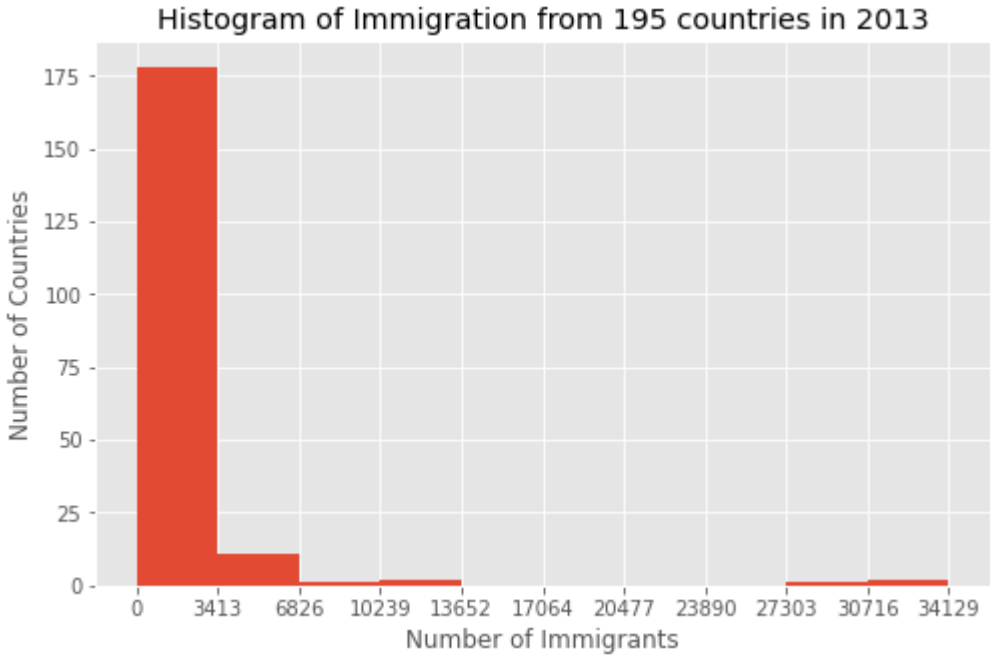
In [59]:
```
# graph the distribution using kind = hist
df_can['2013'].plot(kind='hist', figsize=(8, 5))

# add a title to the histogram
plt.title('Histogram of Immigration from 195 Countries in 2013')
```

```
# add y-label
plt.ylabel('Number of Countries')
# add x-label
plt.xlabel('Number of Immigrants')

plt.show()
# note that x-axis labels don't match bin size.
# this can be fixed by passing in an xticks argument
```



In [60]:

```
# 'bin_edges' is a list of bin intervals
count, bin_edges = np.histogram(df_can['2013'])

df_can['2013'].plot(kind='hist', figsize=(8, 5), xticks=bin_edges)

plt.title('Histogram of Immigration from 195 countries in 2013') # add a title to the h
plt.ylabel('Number of Countries') # add y-label
plt.xlabel('Number of Immigrants') # add x-label

plt.show()
```



In [61]:
```
# what is the distribution for denmark, norway, and sweden?

# let's quickly view the dataset
df_can.loc[['Denmark', 'Norway', 'Sweden'], years]
```

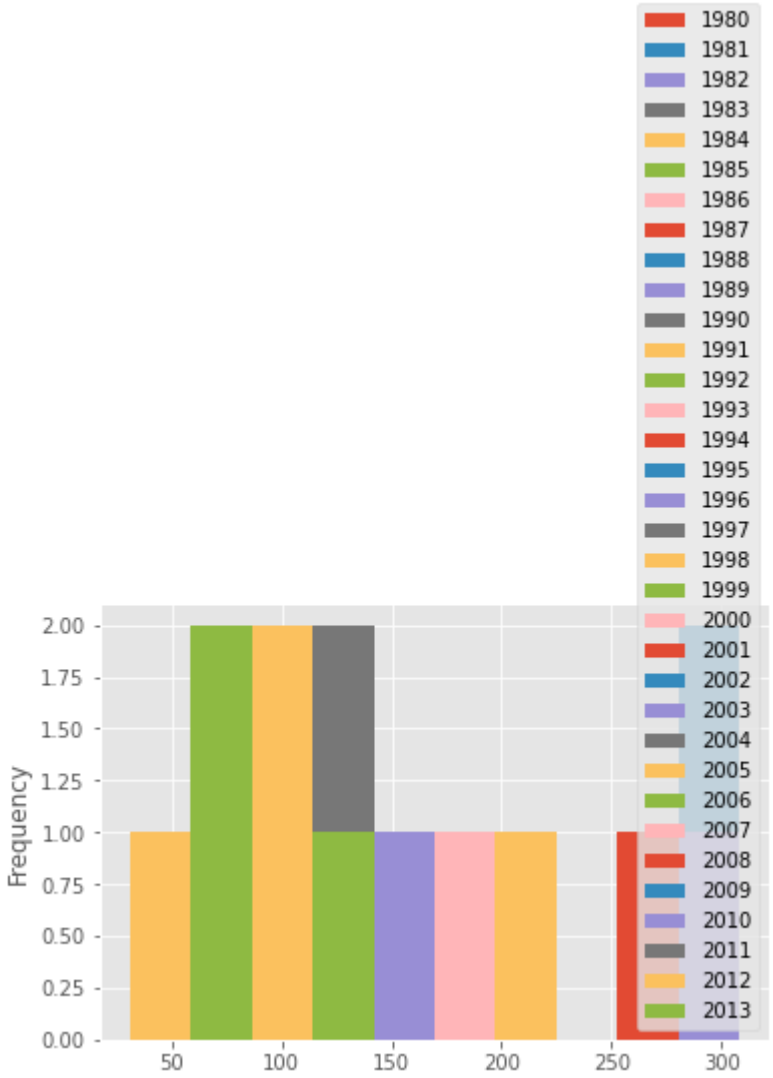
Out[61]:

Country	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2004	2005	2006	200
Denmark	272	293	299	106	93	73	93	109	129	129	...	89	62	101	9
Norway	116	77	106	51	31	54	56	80	73	76	...	73	57	53	7
Sweden	281	308	222	176	128	158	187	198	171	182	...	129	205	139	19

3 rows × 34 columns

In [62]:
```
# generate histogram
df_can.loc[['Denmark', 'Norway', 'Sweden'], years].plot.hist()
# needs to be transposed!
```

Out[62]:
```
<AxesSubplot:ylabel='Frequency'>
```

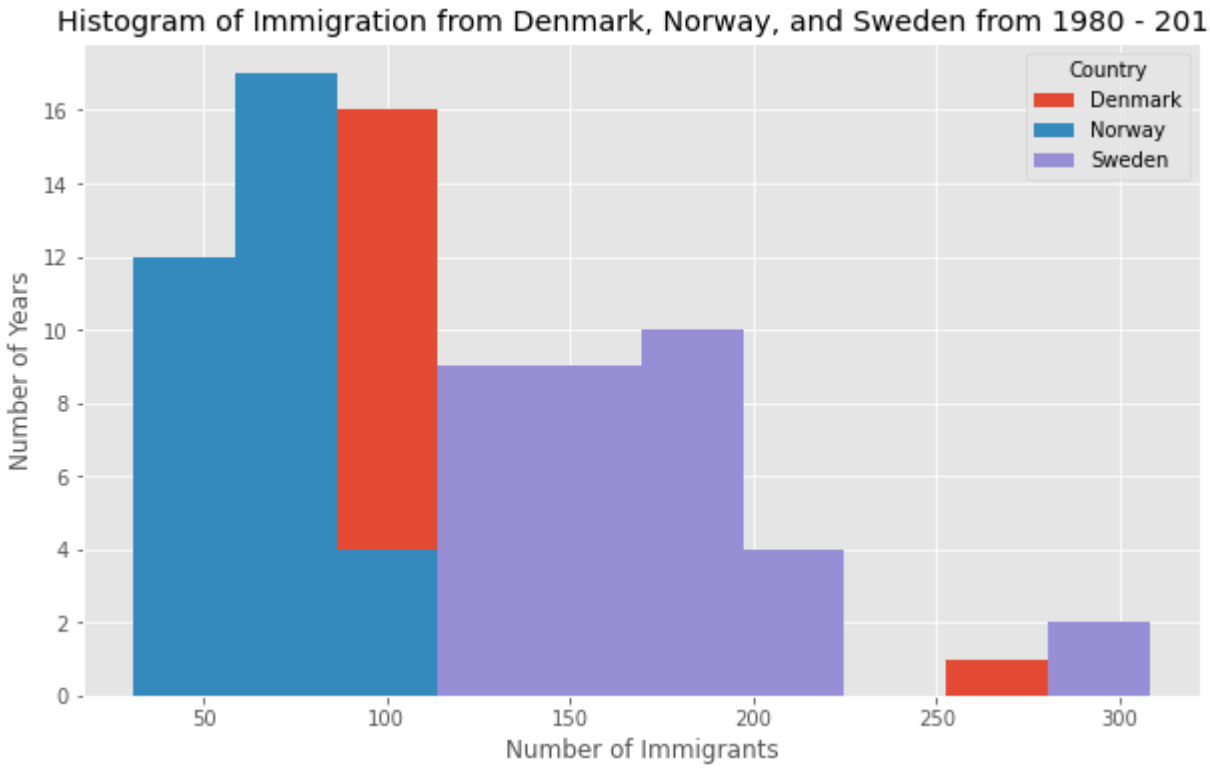


In [63]:
```
# transpose dataframe
df_t = df_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose()
df_t.head()
```

Out[63]:

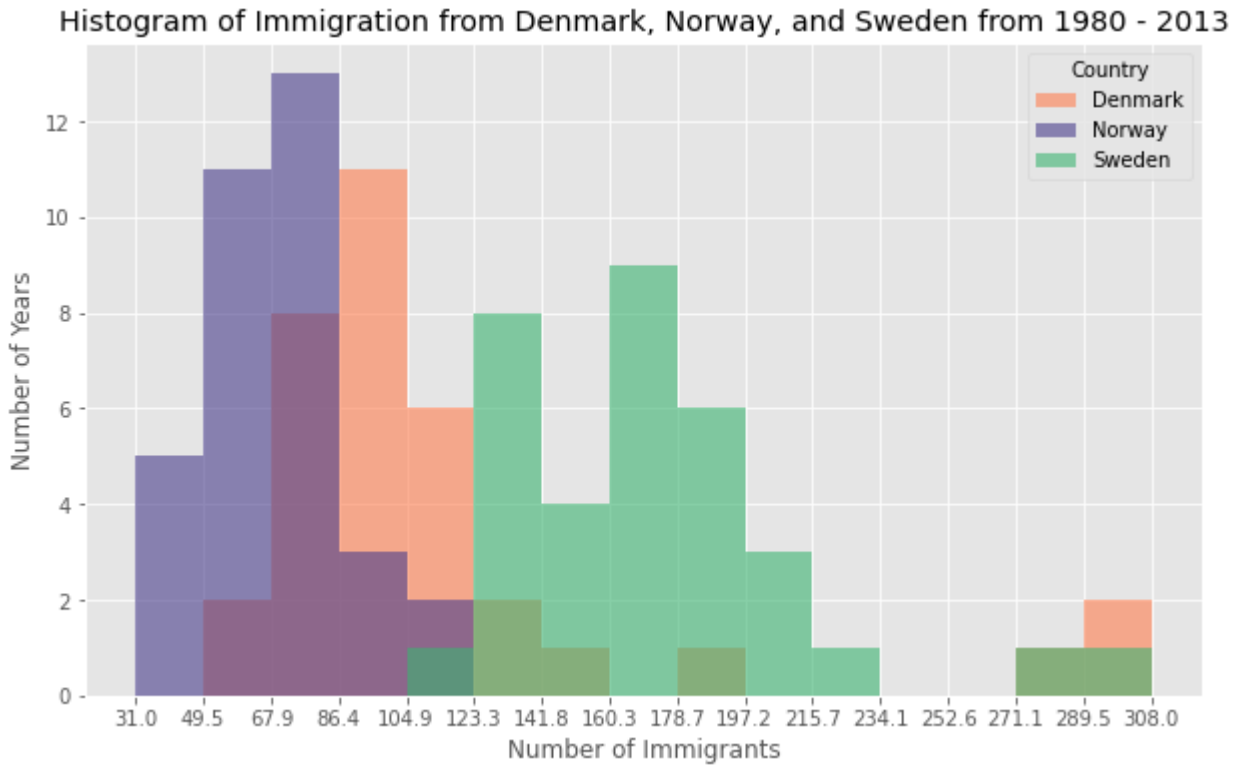
Country	Denmark	Norway	Sweden
1980	272	116	281
1981	293	77	308
1982	299	106	222
1983	106	51	176
1984	93	31	128

In [64]:
```
# generate histogram
df_t.plot(kind='hist', figsize=(10, 6))

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()
```



In [65]:

```
# make some modifications to improve the plot
# let's get the x-tick values
count, bin_edges = np.histogram(df_t, 15)

# un-stacked histogram
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()
```

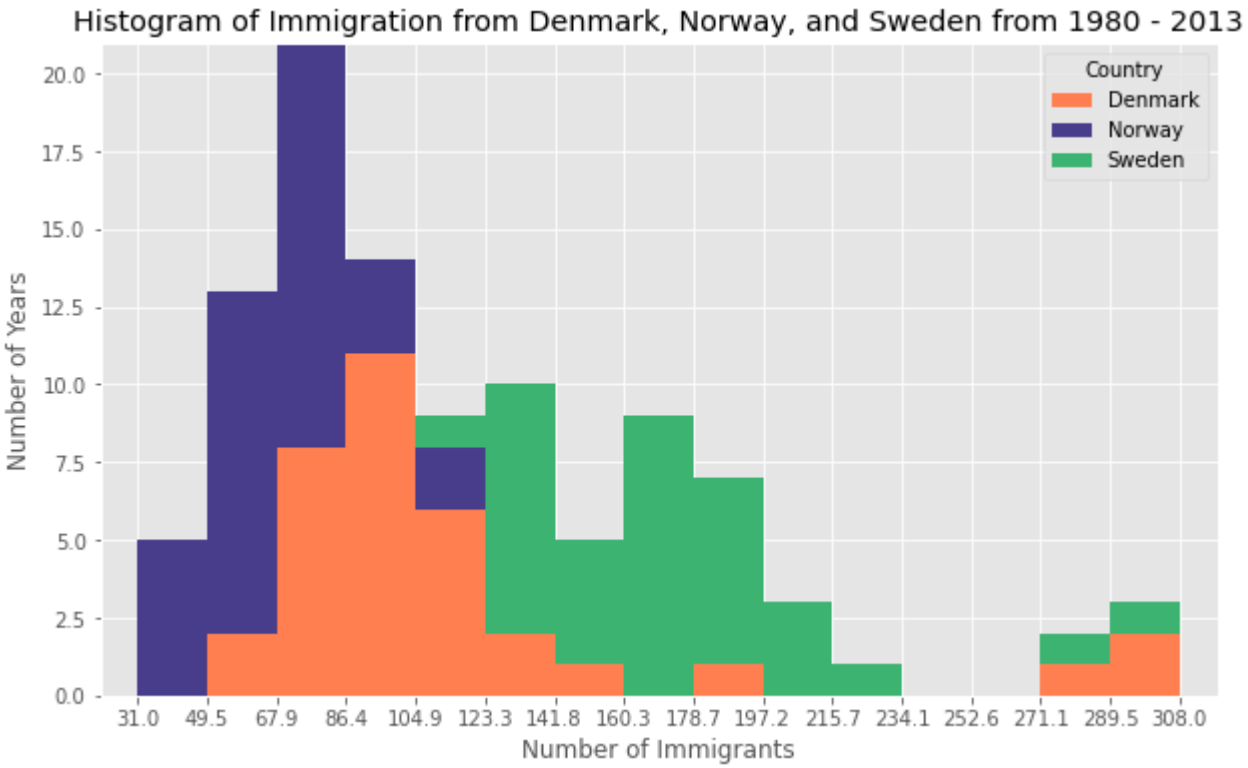


Tip: For a full listing of colors available in Matplotlib, run the following code in your python shell:

```
import matplotlib
for name, hex in matplotlib.colors.cnames.items():
    print(name, hex)
```

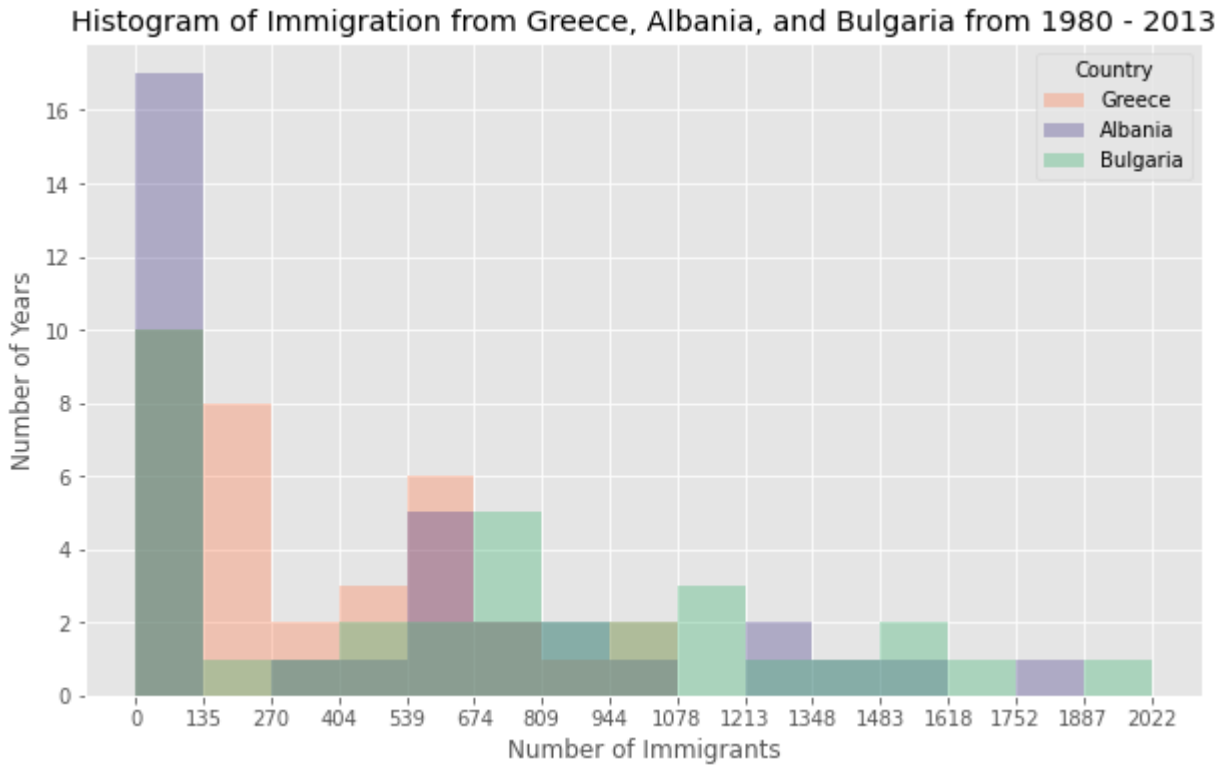
In [66]:

```
# if we don't want the plots to overlap, use the stacked parameter

count, bin_edges = np.histogram(df_t, 15)
xmin = bin_edges[0] - 10   #  first bin value is 31.0, adding buffer of 10 for aestheti
xmax = bin_edges[-1] + 10  #  last bin value is 308.0, adding buffer of 10 for aestheti

# stacked Histogram
df_t.plot(kind='hist',
          figsize=(10, 6),
          bins=15,
          xticks=bin_edges,
          color=['coral', 'darkslateblue', 'mediumseagreen'],
          stacked=True,
          xlim=(xmin, xmax)
         )

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()
```



In [67]:

```
# use the scripting layer to display the immigration for greece, albania, and bulgaria

# create a dataframe of the countries of interest (cof)
df_cof = df_can.loc[['Greece', 'Albania', 'Bulgaria'], years]

# transpose the dataframe
df_cof = df_cof.transpose()

# let's get the x-tick values
count, bin_edges = np.histogram(df_cof, 15)

# Un-stacked Histogram
df_cof.plot(kind ='hist',
                figsize=(10, 6),
                bins=15,
                alpha=0.35,
                xticks=bin_edges,
                color=['coral', 'darkslateblue', 'mediumseagreen']
                )

plt.title('Histogram of Immigration from Greece, Albania, and Bulgaria from 1980 - 2013
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()
```



Bar Chart

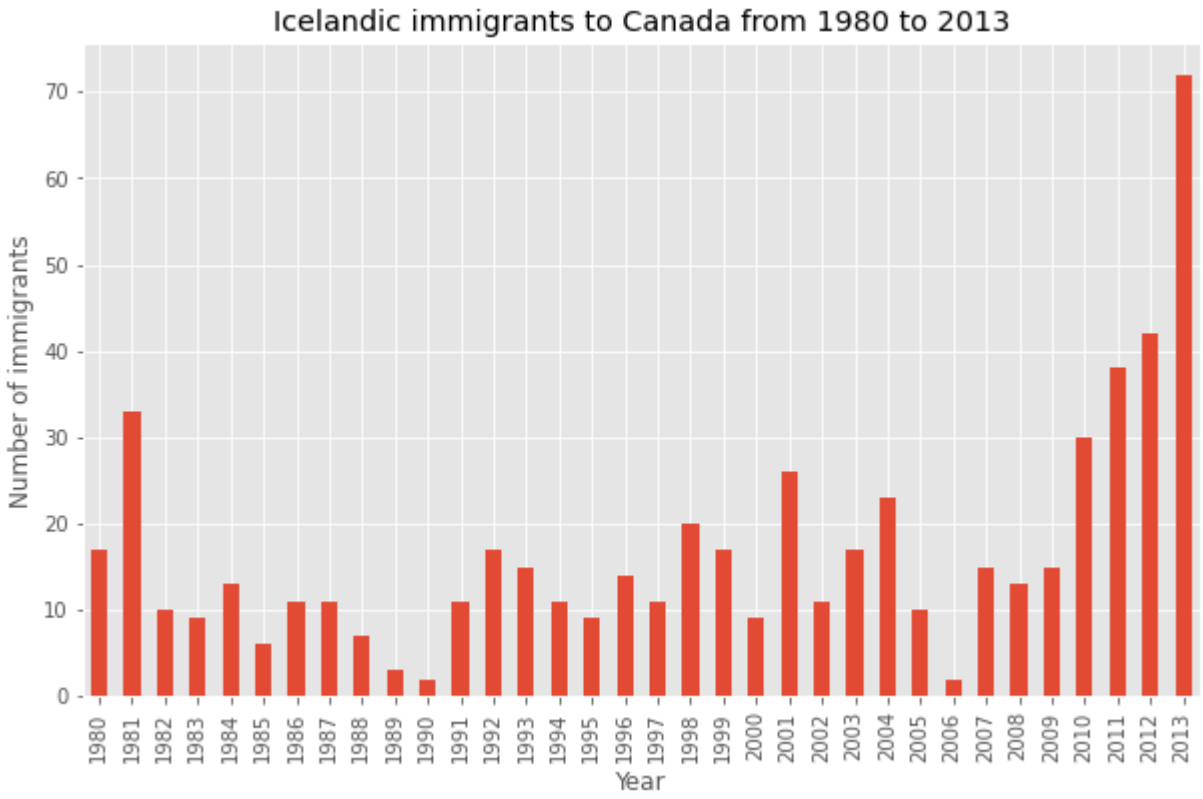
To create a bar plot, we can pass one of two arguments via `kind` parameter in `plot()`:

- `kind=bar` creates a *vertical* bar plot
- `kind=barh` creates a *horizontal* bar plot

In [68]:
```
# compare icelandic immigrants to canada
# step 1: get the data
df_iceland = df_can.loc['Iceland', years]
df_iceland.head()
```

Out[68]:
```
1980    17
1981    33
1982    10
1983     9
1984    13
Name: Iceland, dtype: object
```

In [69]:
```
# 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()
```



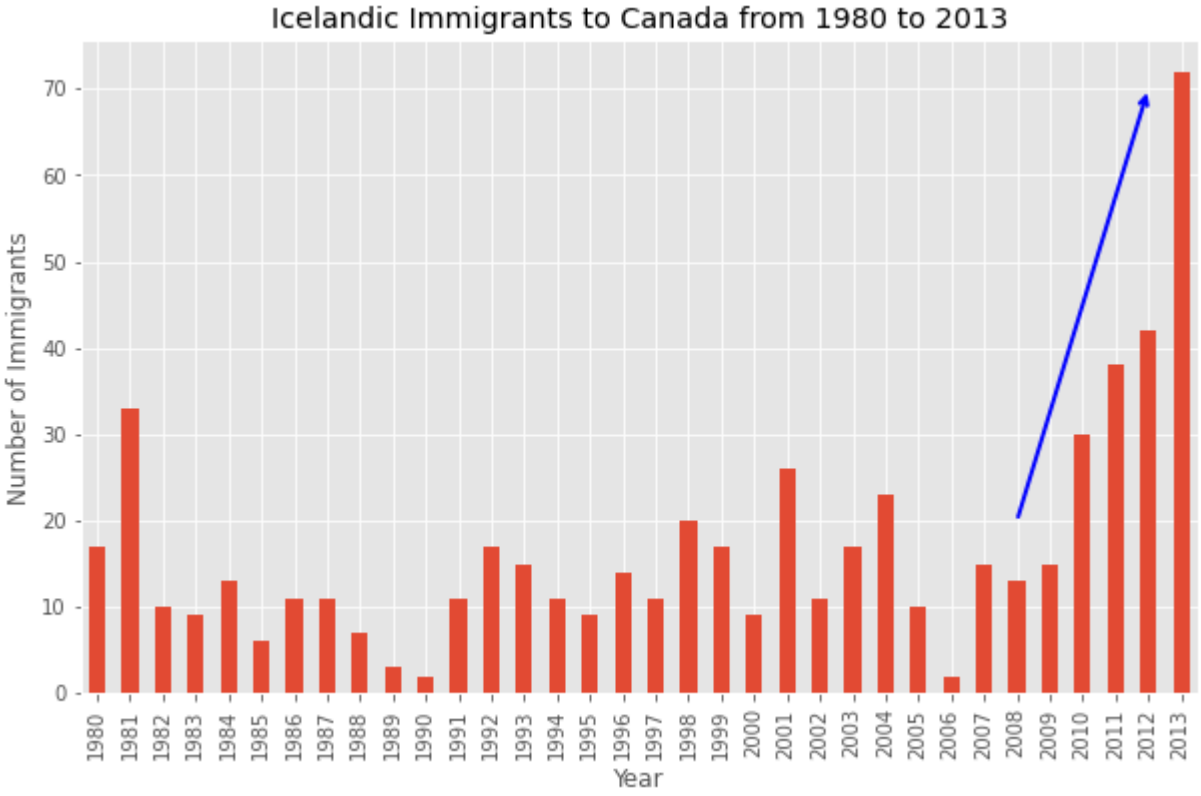
The bar plot above shows the total number of immigrants broken down by each year. We can clearly see the impact of the financial crisis; the number of immigrants to Canada started increasing rapidly after 2008.

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 coordinate system of the object being annotated (default).
- `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` : Specifies color of arrow.
  - `lw` : Specifies the line width.

In [70]:
```
# update the plot
df_iceland.plot(kind='bar', figsize=(10, 6), rot=90)  # rotate the xticks(labelled poin

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 ann
                arrowprops=dict(arrowstyle='->', connectionstyle='arc3', color='blue', lw=
                )

plt.show()
```



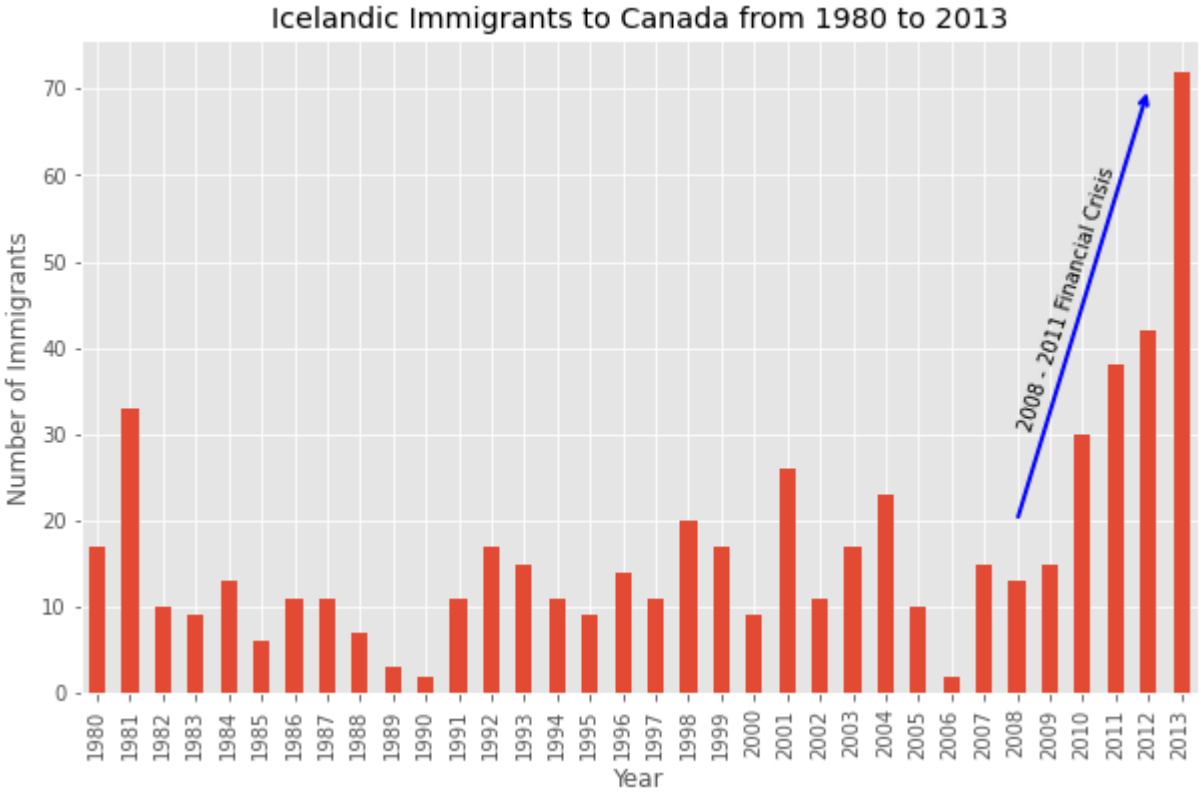
Let's also annotate a text to go over the arrow. We will pass in the following additional parameters:

- `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']

In [71]:

```
# update plot
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: 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 ann
             arrowprops=dict(arrowstyle='->', connectionstyle='arc3', color='blue', lw=
             )

# 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' algned.
```

```
            )

plt.show()
```

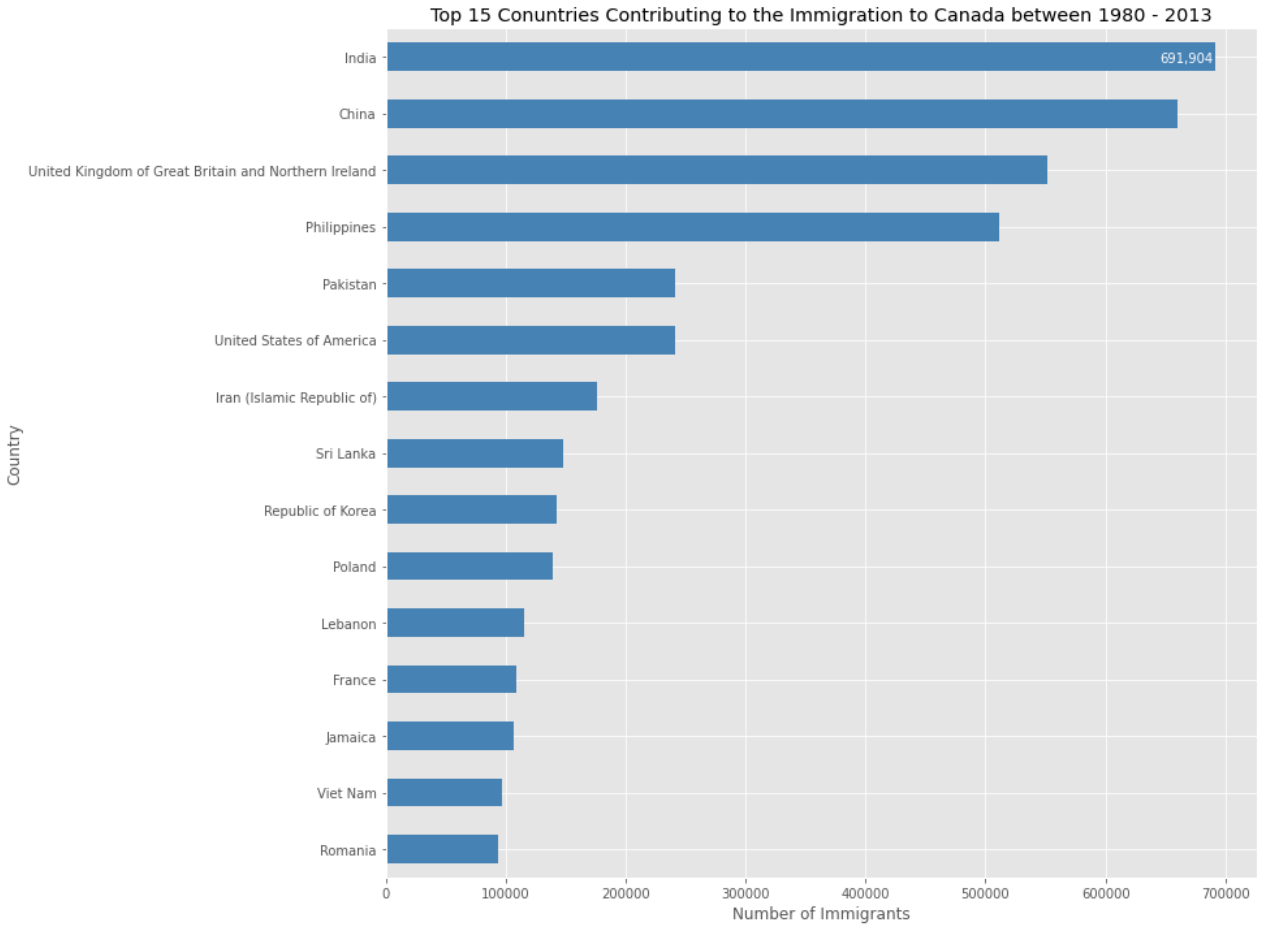


In [72]:
```
# create a horizontal bar plot for total immigrants from the top 15 countries
# sort dataframe on 'Total' column (descending)
df_can.sort_values(by='Total', ascending=True, inplace=True)

# get top 15 countries
df_top15 = df_can['Total'].tail(15)
df_top15
```

Out[72]:
```
Country
Romania                                                 93585
Viet Nam                                                97146
Jamaica                                                106431
France                                                 109091
Lebanon                                                115359
Poland                                                 139241
Republic of Korea                                      142581
Sri Lanka                                              148358
Iran (Islamic Republic of)                             175923
United States of America                               241122
Pakistan                                               241600
Philippines                                            511391
United Kingdom of Great Britain and Northern Ireland   551500
China                                                  659962
India                                                  691904
Name: Total, dtype: int64
```

In [73]:
```
# generate plot
df_top15.plot(kind='barh', figsize=(12, 12), color='steelblue')
plt.xlabel('Number of Immigrants')
```

```
plt.title('Top 15 Conuntries Contributing to the Immigration to Canada between 1980 - 2

# 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 (subtracting 47000 from x, and 0.1 from y to make it fit
plt.annotate(label, xy=(value - 47000, index - 0.10), color='white')

plt.show()
```



Pie Charts

Pie Chart:

- pie charts is a circular graph that is divided into slices to illustrate numerical proportion.
- first group the data on the column or row you want, then graph.

Syntax:

- df['col'].plot(kind = 'pie')

Box Plots

Box Plot:

- a way of statistically representing the distribution through the 5 main dimensions:

- min, max, IQR

Syntax:

- new_df = df.loc[['col'], x].transpose()
- new_df.plot(kind = 'box')

Scatter Plots

Scatter Plot:

- displays values pertaining to (typically) two variables against one another to determine if any correlation exists between them.
- a dependent variable plotted against an independent variable.

Syntax (using pyplot):

- df.plot(kind = 'scatter', x = 'col', y = 'col2')

Syntax (using seaborn):

- sns.scatterplot(x, y, data, style)

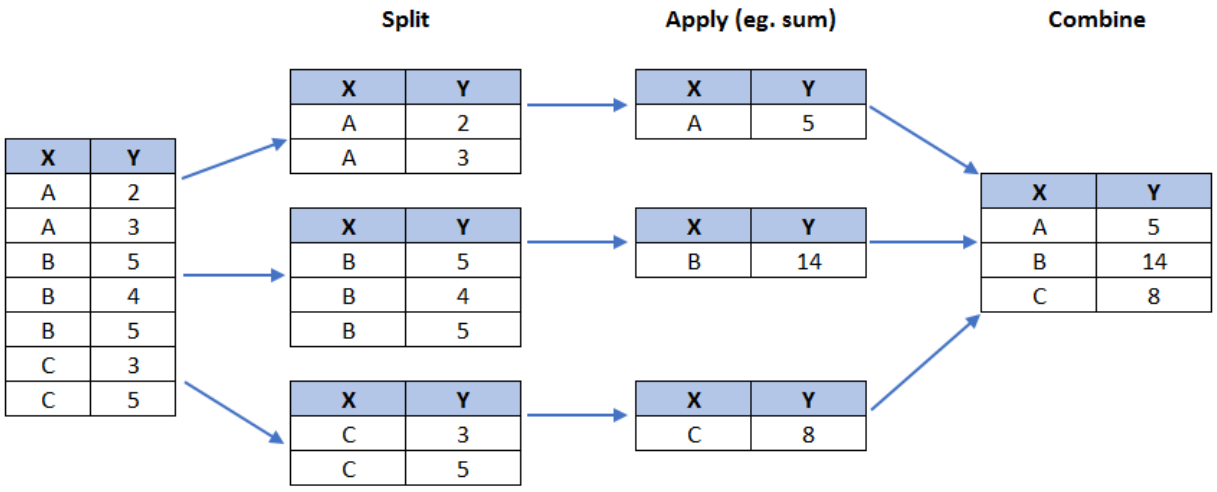
Lab: Specialized Visualization Tools

Pie Charts

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:

1. **Split:** Splitting the data into groups based on some criteria.
2. **Apply:** Applying a function to each group independently: .sum() .count() .mean() .std() .aggregate() .apply() .etc..
3. **Combine:** Combining the results into a data structure.



In [74]:
```
# 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 can not use it further until we apply a function (eg .sum())
print(type(df_can.groupby('Continent', axis=0)))

df_continents.head()
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Out[74]:

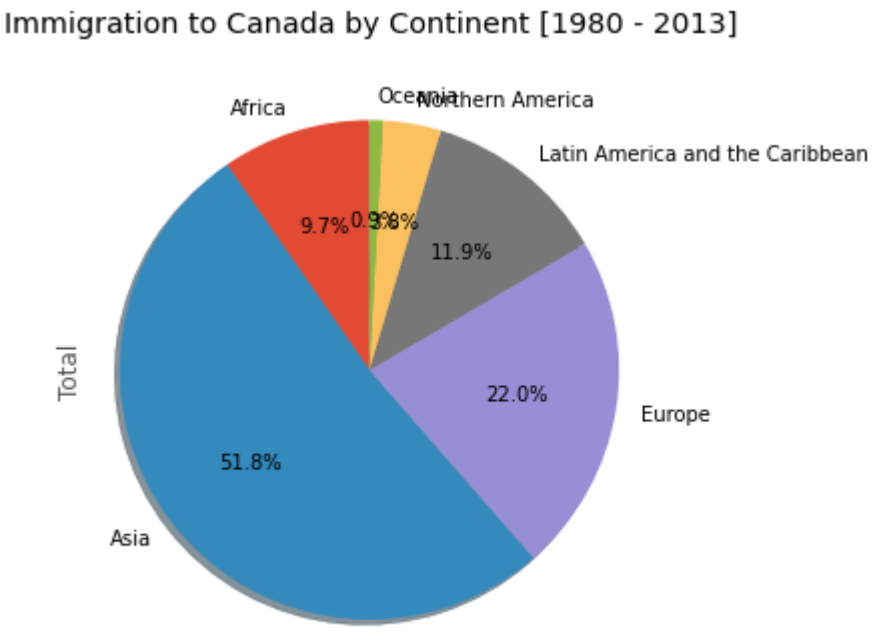
Continent	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2005	2006
Africa	3951	4363	3819	2671	2639	2650	3782	7494	7552	9894	...	27523	29188
Asia	31025	34314	30214	24696	27274	23850	28739	43203	47454	60256	...	159253	149054
Europe	39760	44802	42720	24638	22287	20844	24370	46698	54726	60893	...	35955	33053
Latin America and the Caribbean	13081	15215	16769	15427	13678	15171	21179	28471	21924	25060	...	24747	24676
Northern America	9378	10030	9074	7100	6661	6543	7074	7705	6469	6790	...	8394	9613

5 rows × 35 columns

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).

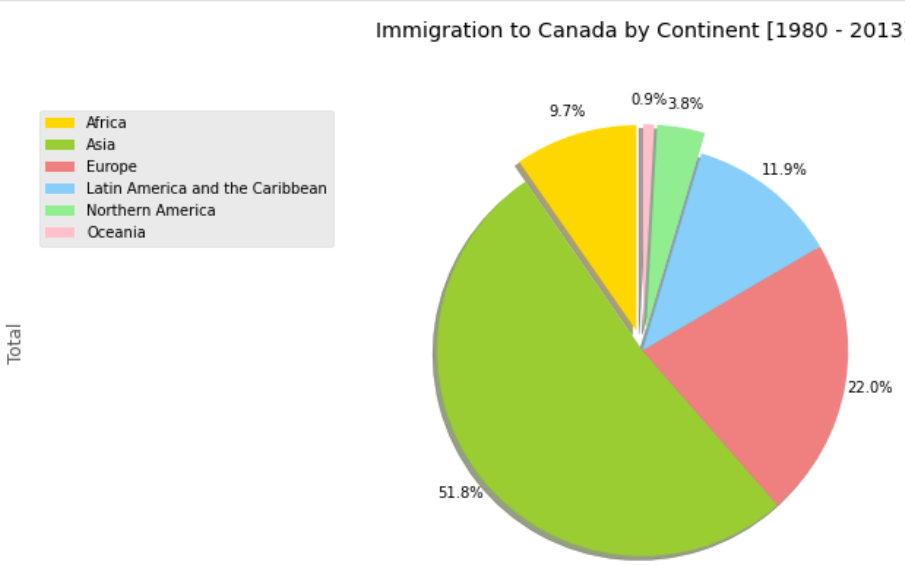
In [75]:
```
# autopct create %, start angle represent starting point
df_continents['Total'].plot(kind='pie',
                            figsize=(5, 6),
                            autopct='%1.1f%%', # add in percentages
                            startangle=90,     # start angle 90° (Africa)
                            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 above visual is not very clear, the numbers and text overlap in some instances. Let's make a few modifications to improve the visuals:

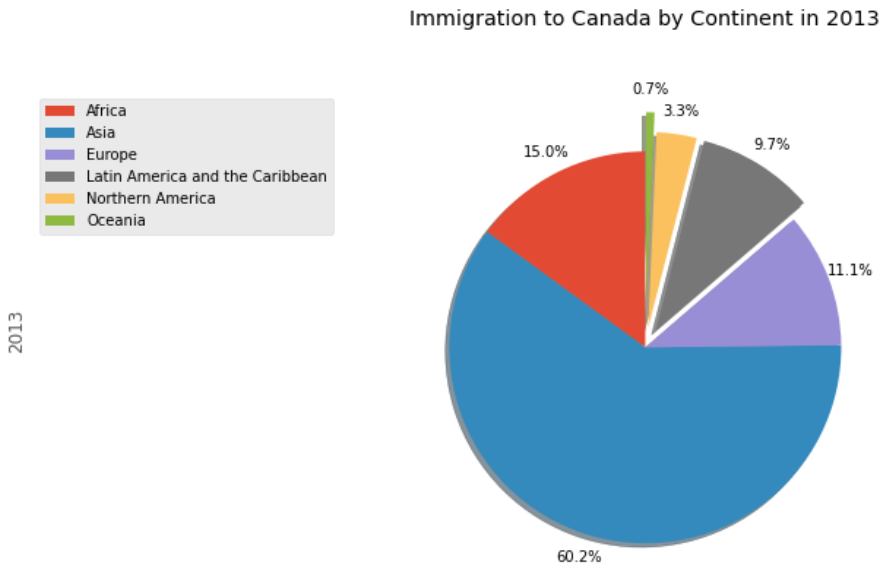
- Remove the text labels on the pie chart by passing in `legend` and add it as a seperate legend using `plt.legend()`.
- Push out the percentages to sit just outside the pie chart by passing in `pctdistance` parameter.
- Pass in a custom set of colors for continents by passing in `colors` parameter.
- **Explode** the pie chart to emphasize the lowest three continents (Africa, North America, and Latin America and Caribbean) by passing in `explode` parameter.

In [76]:
```
colors_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink
explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each continent with which to offset

df_continents['Total'].plot(kind='pie',
                            figsize=(15, 6),
                            autopct='%1.1f%%',
                            startangle=90,
                            shadow=True,
                            labels=None,         # turn off labels on pie chart
                            pctdistance=1.12,    # the ratio between the center of each
                            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')

# add legend
plt.legend(labels=df_continents.index, loc='upper left')

plt.show()
```



In [77]:

```
# use a pie chart to explore the percentage of new immigrants grouped by continents in
explode_list = [0.0, 0, 0, 0.1, 0.1, 0.2] # ratio for each continent with which to offs

df_continents['2013'].plot(kind='pie',
                           figsize=(15, 6),
                           autopct='%1.1f%%',
                           startangle=90,
                           shadow=True,
                           labels=None,                 # turn off labels on pie c
                           pctdistance=1.12,            # the ratio between the pi
                           explode=explode_list         # 'explode' lowest 3 conti
                           )

# scale the title up by 12% to match pctdistance
plt.title('Immigration to Canada by Continent in 2013', y=1.12)
plt.axis('equal')

# add legend
plt.legend(labels=df_continents.index, loc='upper left')

# show plot
plt.show()
```



Box Plots

A `box plot` is a way of statistically representing the *distribution* of the data through five main dimensions:

- **Minimum:** The smallest number in the dataset excluding the outliers.
- **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:** The largest number in the dataset excluding the outliers.

Step 1: Get the subset of 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.

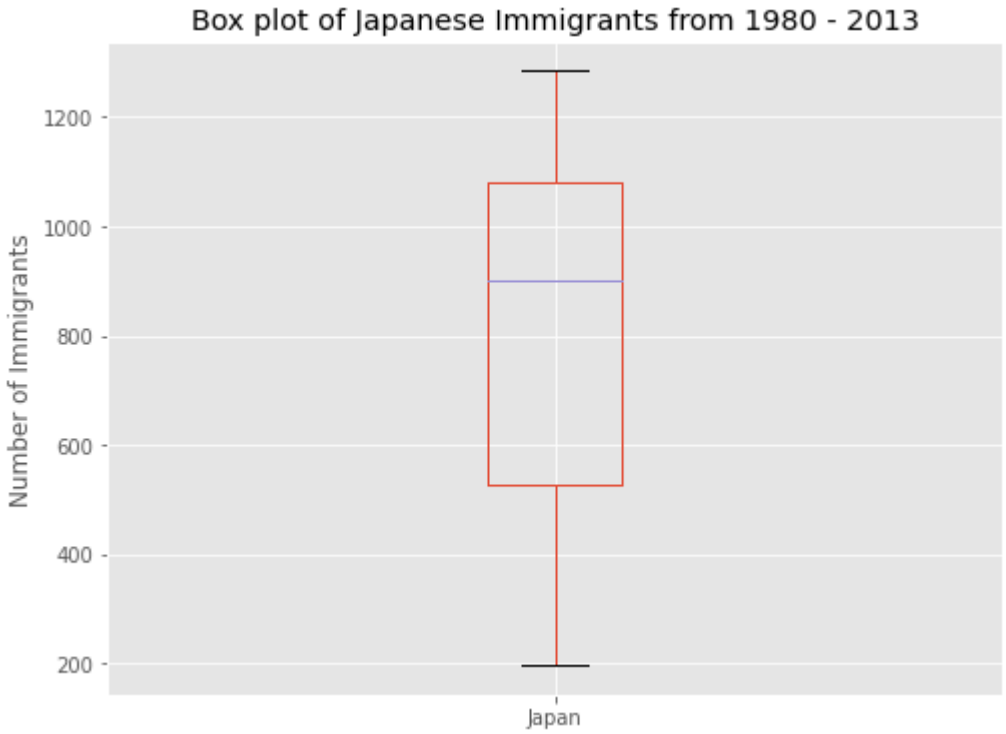
In [78]:
```
# get data for japanese immigration
# to get a dataframe, place extra square brackets around 'Japan'.
df_japan = df_can.loc[['Japan'], years].transpose()
df_japan.head()
```

Out[78]:

Country	Japan
1980	701
1981	756
1982	598
1983	309
1984	246

In [79]:
```
# step 2: plot
df_japan.plot(kind='box', figsize=(8, 6))

plt.title('Box plot of Japanese Immigrants from 1980 - 2013')
plt.ylabel('Number of Immigrants')
```

```
plt.show()
```



In [80]:
```
# view the values
df_japan.describe()
```

Out[80]:

Country	Japan
count	34.000000
mean	814.911765
std	337.219771
min	198.000000
25%	529.000000
50%	902.000000
75%	1079.000000
max	1284.000000

In [81]:
```
# compare the distribution for india and china
df_CI= df_can.loc[['China', 'India'], years].transpose()
df_CI.head()
```

Out[81]:

Country	China	India
1980	5123	8880
1981	6682	8670
1982	3308	8147

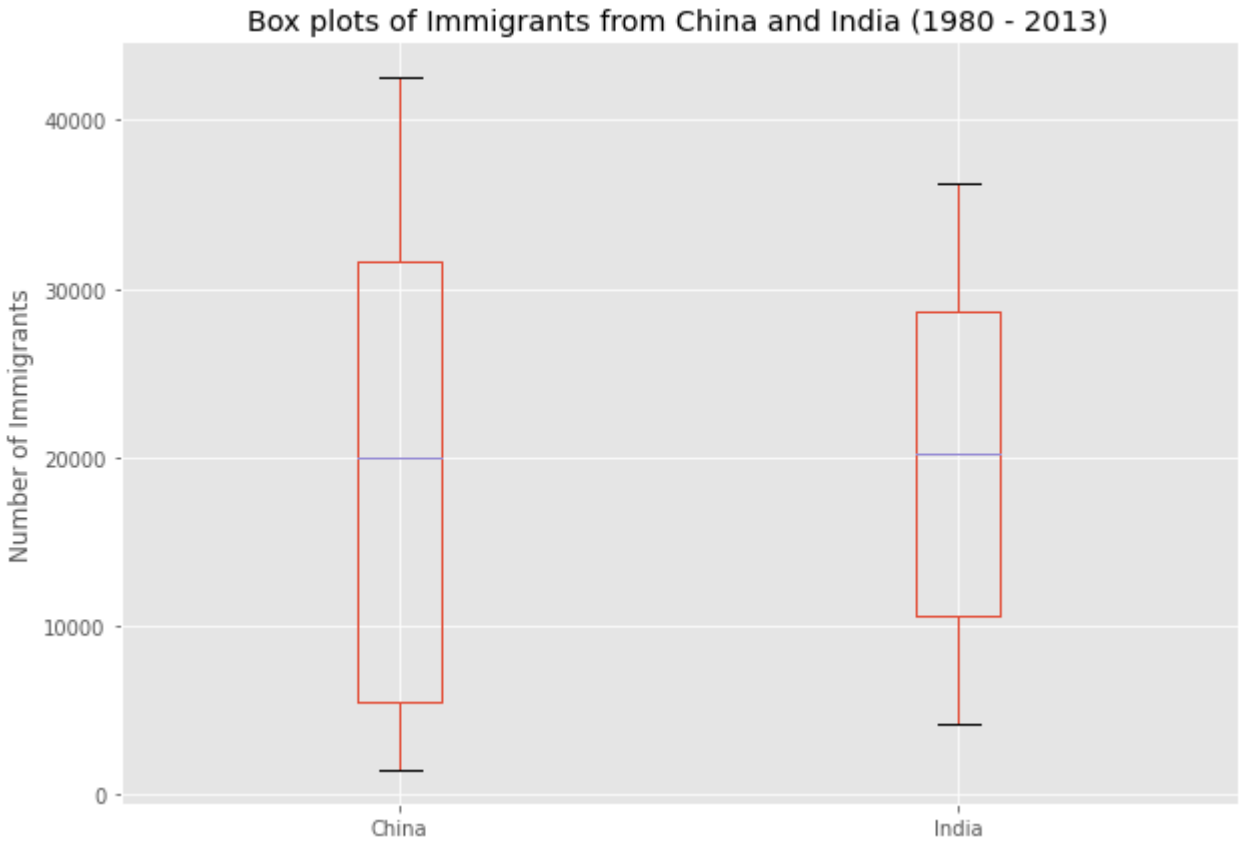
Country	China	India
1983	1863	7338
1984	1527	5704

In [82]:
```
# view percentiles
df_CI.describe()
```

Out[82]:

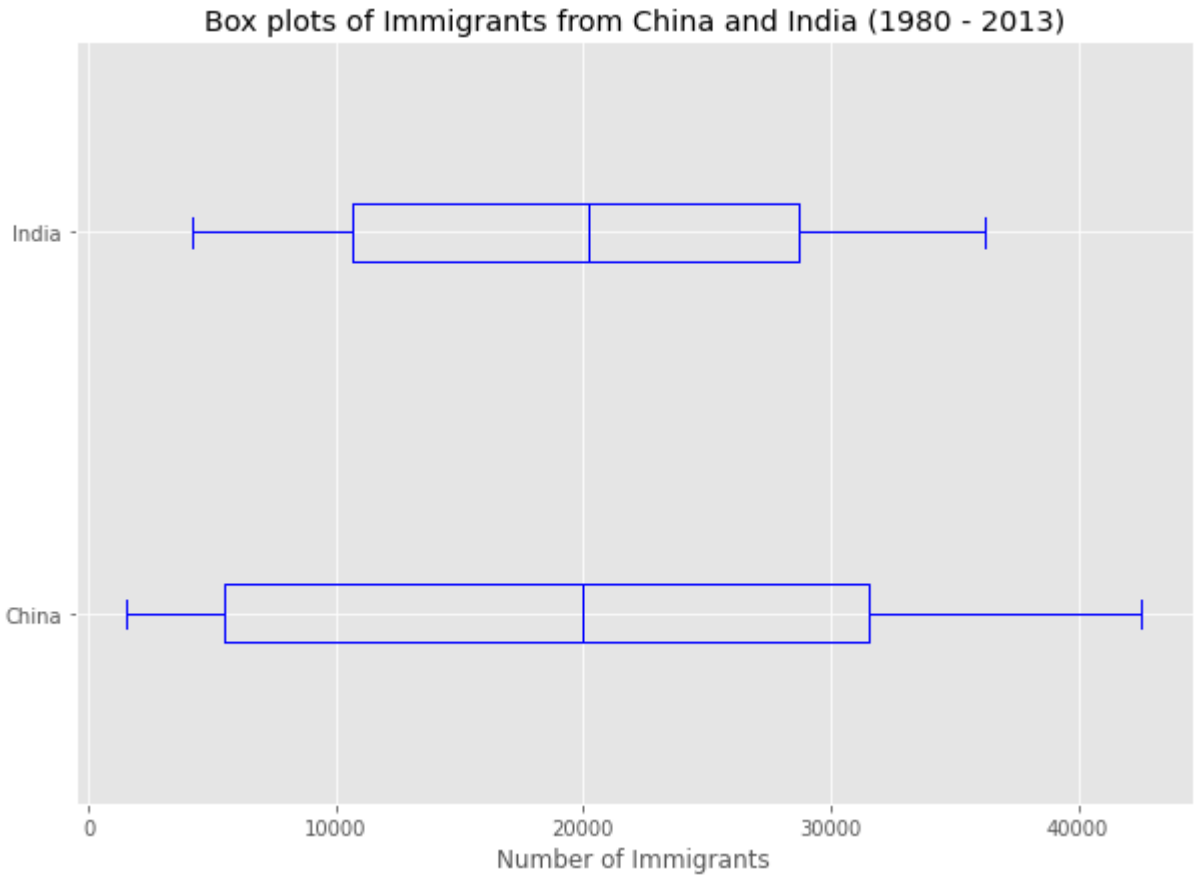
Country	China	India
count	34.000000	34.000000
mean	19410.647059	20350.117647
std	13568.230790	10007.342579
min	1527.000000	4211.000000
25%	5512.750000	10637.750000
50%	19945.000000	20235.000000
75%	31568.500000	28699.500000
max	42584.000000	36210.000000

In [83]:
```
# plot the data
df_CI.plot(kind='box', figsize=(10, 7))

plt.title('Box plots of Immigrants from China and India (1980 - 2013)')
plt.ylabel('Number of Immigrants')

plt.show()
```



In [84]:

```
# horizontal box plots
df_CI.plot(kind='box', figsize=(10, 7), color='blue', vert=False)

plt.title('Box plots of Immigrants from China and India (1980 - 2013)')
plt.xlabel('Number of Immigrants')

plt.show()
```



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.

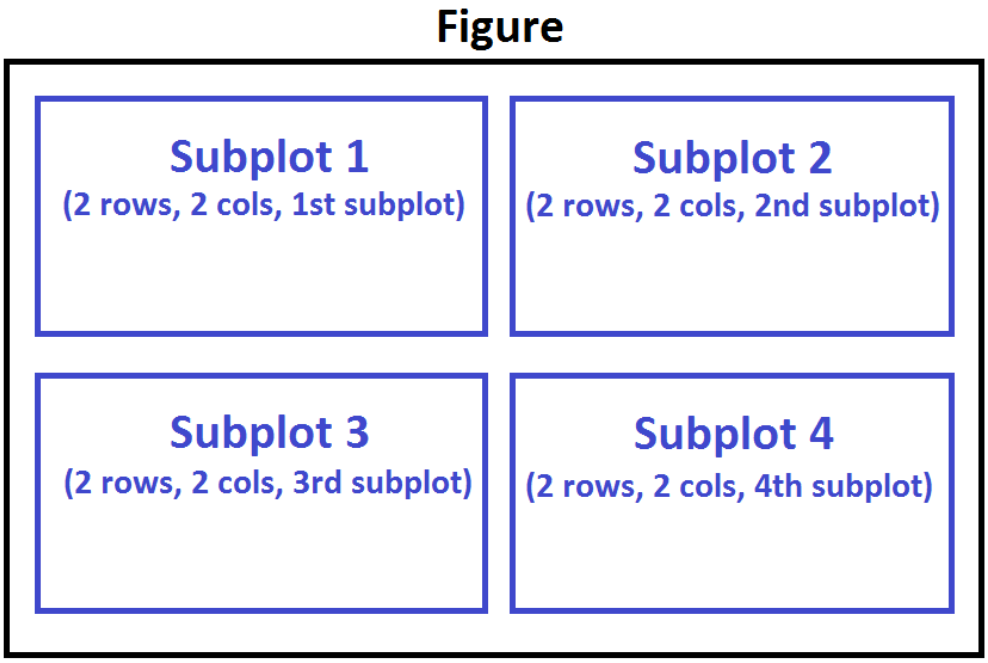
To visualize multiple plots together, we can create a **`figure`** (overall canvas) and divide it into **`subplots`**, each containing a plot. With **subplots**, we usually work with the **artist layer** instead of the **scripting layer**.

Typical syntax is :

```
fig = plt.figure() # create figure
    ax = fig.add_subplot(nrows, ncols, plot_number) # create subplots
```

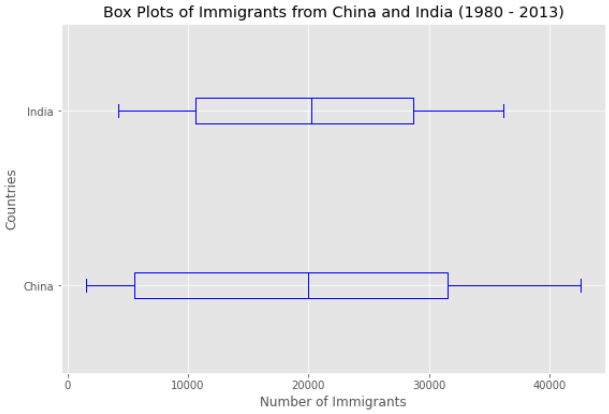
Where

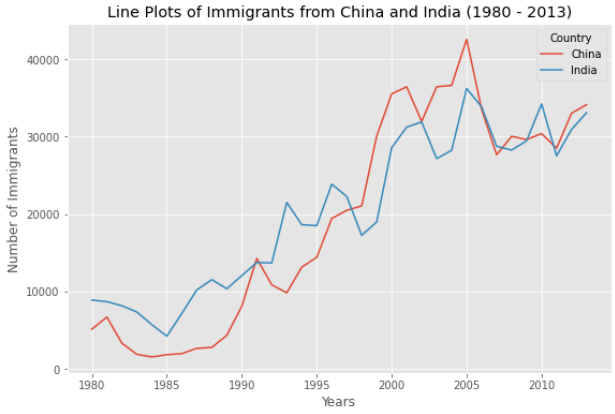
- `nrows` and `ncols` are used to notionally split the figure into (`nrows * ncols`) sub-axes,
- `plot_number` is used to identify the particular subplot that this function is to create within the notional grid. `plot_number` starts at 1, increments across rows first and has a maximum of `nrows * ncols` as shown below.



In [85]:

```
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

# Subplot 1: Box plot
df_CI.plot(kind='box', color='blue', vert=False, figsize=(20, 6), ax=ax0) # add to subp
ax0.set_title('Box Plots of Immigrants from China and India (1980 - 2013)')
ax0.set_xlabel('Number of Immigrants')
ax0.set_ylabel('Countries')

# Subplot 2: Line plot
df_CI.plot(kind='line', figsize=(20, 6), ax=ax1) # add to subplot 2
ax1.set_title ('Line Plots of Immigrants from China and India (1980 - 2013)')
ax1.set_ylabel('Number of Immigrants')
ax1.set_xlabel('Years')

plt.show()
```





In [86]:

```
# get dataset of top 15 countries based on total immigration
df_top15 = df_can.sort_values(['Total'], ascending=False, axis=0).head(15)
df_top15
```

Out[86]:

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	20
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	362
China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1960	...	425
United Kingdom of Great Britain and Northern Ireland	Europe	Northern Europe	Developed regions	22045	24796	20620	10015	10170	9564	9470	...	72
Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	181
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	143
United States of America	Northern America	Northern America	Developed regions	9378	10030	9074	7100	6661	6543	7074	...	83
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	58
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	49
Republic of Korea	Asia	Eastern Asia	Developing regions	1011	1456	1572	1081	847	962	1208	...	58
Poland	Europe	Eastern Europe	Developed regions	863	2930	5881	4546	3588	2819	4808	...	14
Lebanon	Asia	Western Asia	Developing regions	1409	1119	1159	789	1253	1683	2576	...	37
France	Europe	Western Europe	Developed regions	1729	2027	2219	1490	1169	1177	1298	...	44
Jamaica	Latin America and the Caribbean	Caribbean	Developing regions	3198	2634	2661	2455	2508	2938	4649	...	19
Viet Nam	Asia	South-Eastern Asia	Developing regions	1191	1829	2162	3404	7583	5907	2741	...	18
Romania	Europe	Eastern Europe	Developed regions	375	438	583	543	524	604	656	...	50

15 rows × 38 columns

In [87]:
```
# create new dataframe which aggregates for each decade

### type your answer here


#The correct answer is:

# 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
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
new_df = pd.DataFrame({'1980s': df_80s, '1990s': df_90s, '2000s':df_00s})

# display dataframe
new_df.head()
```

Out[87]:

Country	1980s	1990s	2000s
India	82154	180395	303591
China	32003	161528	340385
United Kingdom of Great Britain and Northern Ireland	179171	261966	83413
Philippines	60764	138482	172904
Pakistan	10591	65302	127598

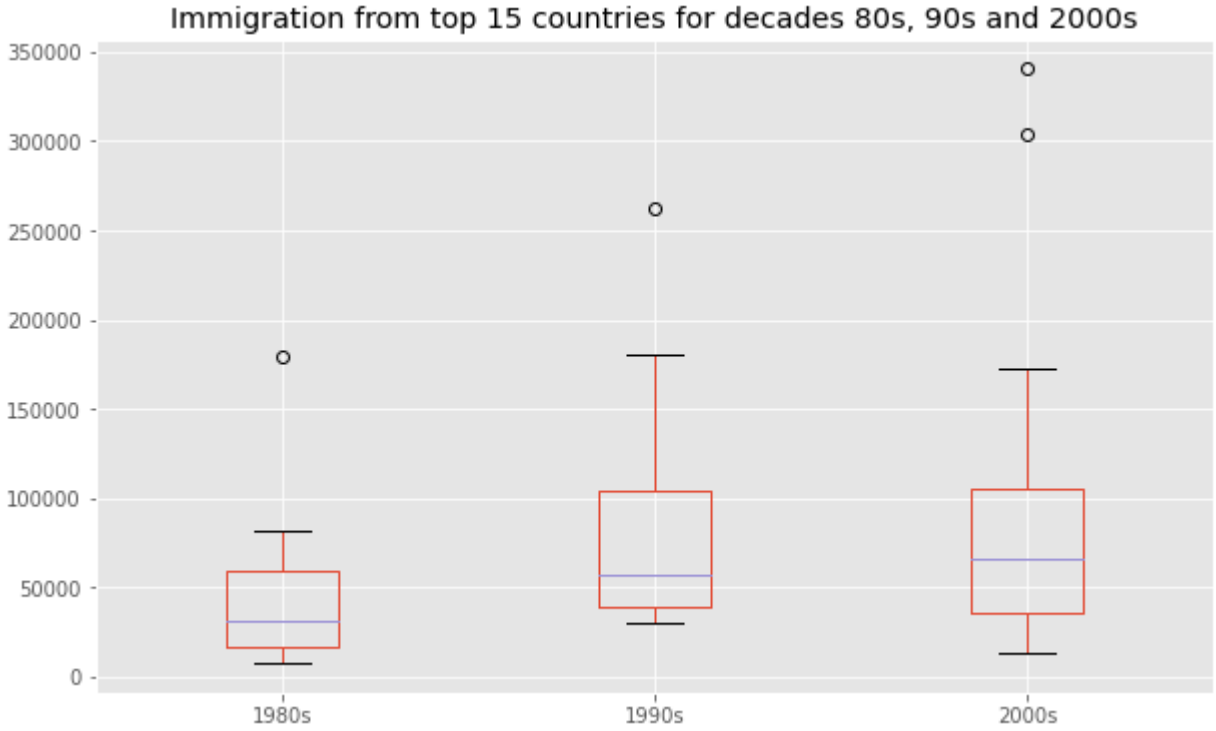
In [88]:
```
new_df.describe()
```

Out[88]:

	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

In [89]:
```
# create plots
new_df.plot(kind='box', figsize=(10, 6))
```

```
plt.title('Immigration from top 15 countries for decades 80s, 90s and 2000s')

plt.show()
```



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

In [90]:
```
# check how many entries are outliers
new_df=new_df.reset_index()
new_df[new_df['2000s']> 209611.5]
```

Out[90]:

	Country	1980s	1990s	2000s
0	India	82154	180395	303591
1	China	32003	161528	340385

Scatter Plots

Step 1: Get the dataset. Since we are expecting to use the relationship betewen `years` and `total population`, we will convert `years` to `int` type.

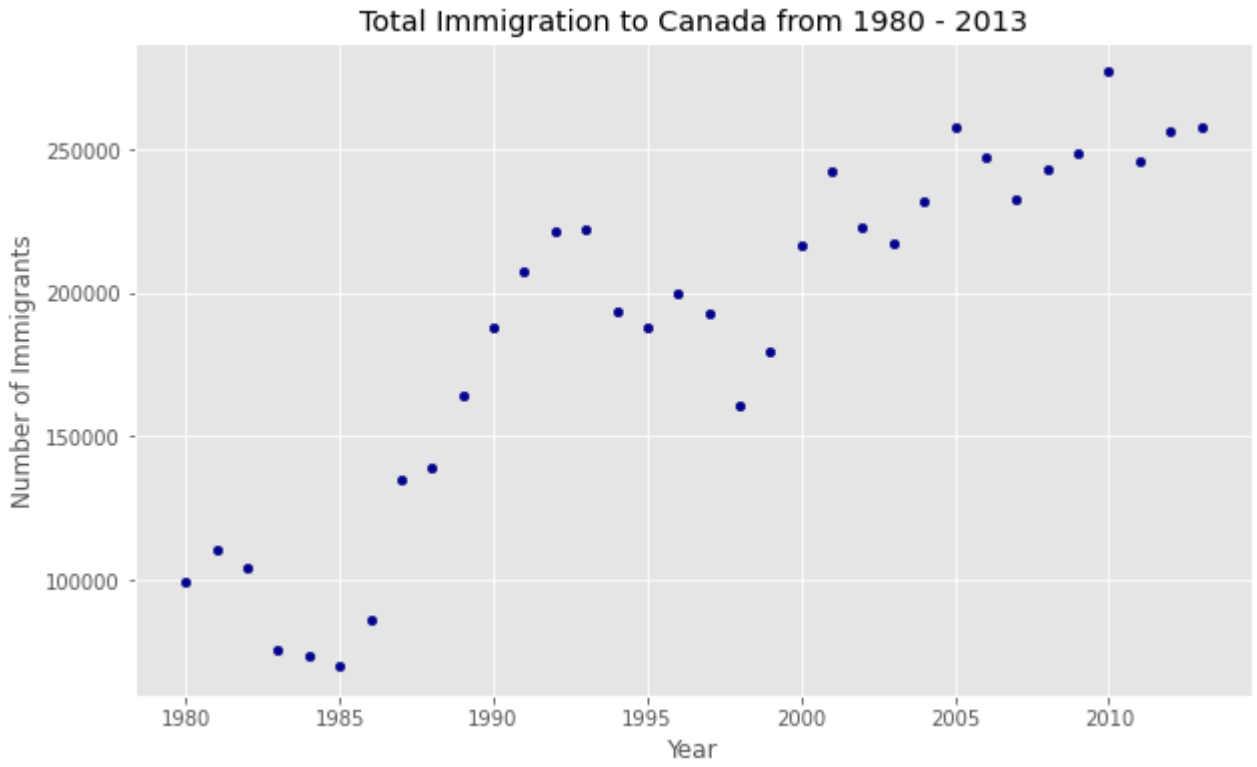
In [91]:
```
# we can use the sum() method to get the total population per year
df_tot = pd.DataFrame(df_can[years].sum(axis=0))

# 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)

# rename columns
df_tot.columns = ['year', 'total']

# view the final dataframe
df_tot.head()
```

Out[91]:

	year	total
0	1980	99137
1	1981	110563
2	1982	104271
3	1983	75550
4	1984	73417

Step 2: 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.

In [92]:
```
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 data points 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 upward trend using a regression line (line of best fit)

So let's try to plot a linear line of best fit, and use it to predict the number of immigrants in 2015.

Step 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.

In [93]:
```
x = df_tot['year']      # year on x-axis
y = df_tot['total']     # total on y-axis
fit = np.polyfit(x, y, deg=1)

fit
```

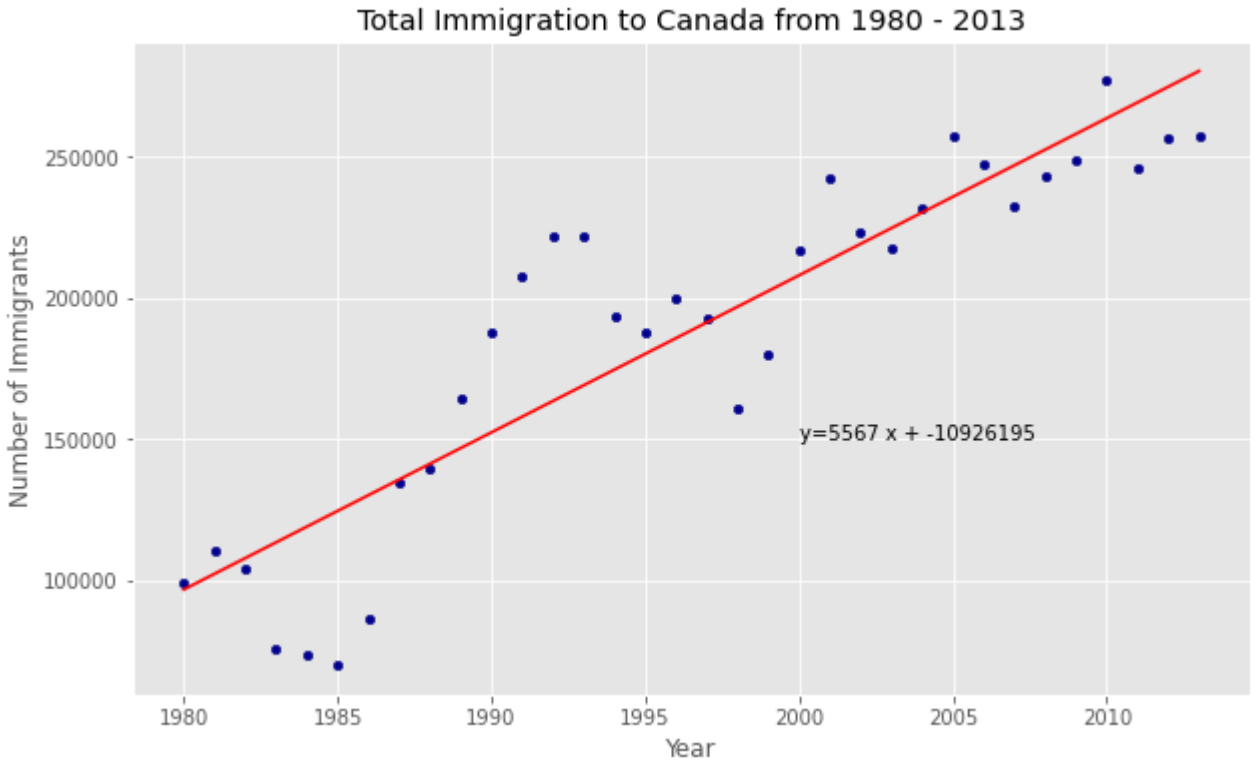
Out[93]:
```
array([ 5.56709228e+03, -1.09261952e+07])
```

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 `[5.56709228e+03, -1.09261952e+07]` with the the slope in position 0 and intercept in position 1.

Step 2: Plot the regression line on the `scatter plot`.

In [94]:
```
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])
```



Out[94]: '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 * Year - 10926195
No. Immigrants = 5567 * 2015 - 10926195
No. Immigrants = 291,310
```

When compared to the actual 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).

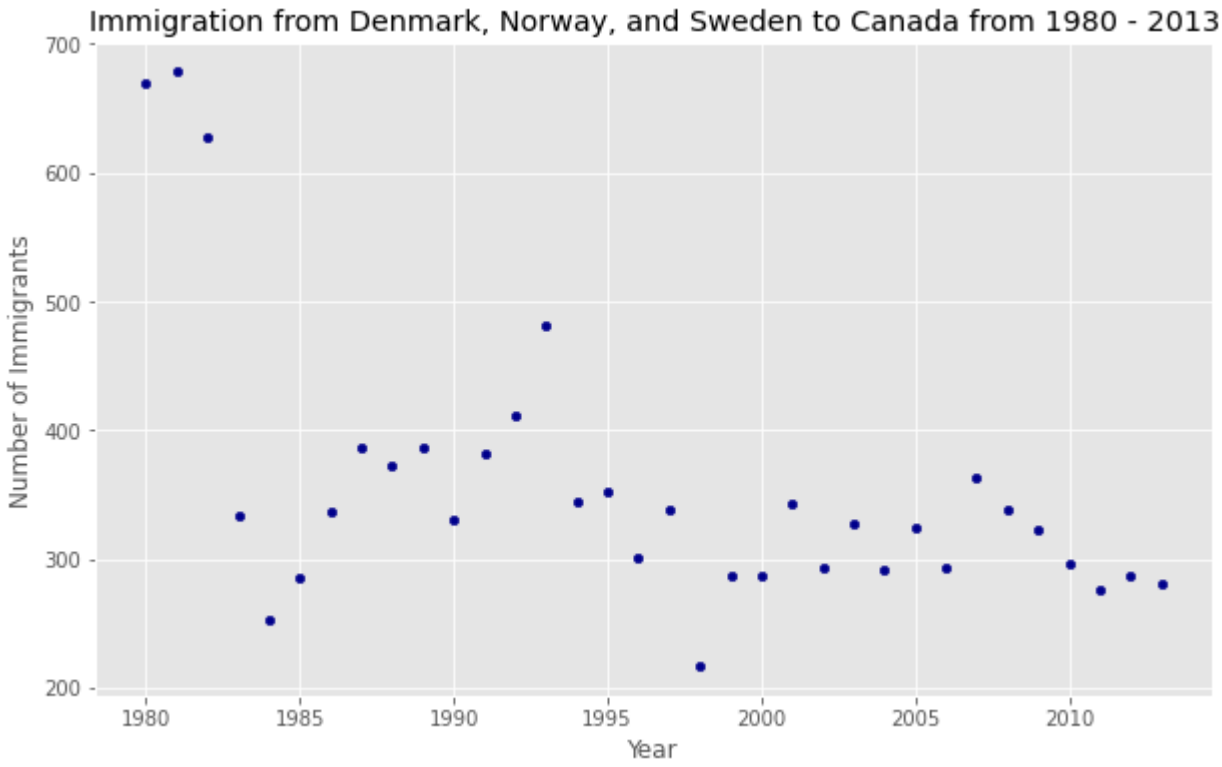
As a side note, we can observe that immigration took a dip around 1993 - 1997. Further analysis into the topic revealed that in 1993 Canada introcuded Bill C-86 which introduced revisions to the refugee determination system, mostly restrictive. Further amendments to the Immigration Regulations cancelled 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.

In [95]:
```
# create a scatter plot of the total immigration from denmark, norway, and sweden

# get data
# create df_countries dataframe
df_countries = df_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose()

# create df_total by summing across three countries for each year
df_total = pd.DataFrame(df_countries.sum(axis=1))

# 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()
```

Out[95]:

	year	total
0	1980	669
1	1981	678
2	1982	627
3	1983	333
4	1984	252

In [96]:
```
# 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')

# show plot
plt.show()
```



Bubble Plot

A `bubble plot` is a variation of the `scatter plot` that displays three dimensions of data (x, y, z). The data points 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 parameter `s` to `plot()`, that contains the weight of each point.

Step 1: Get the data for Brazil and Argentina. Like in the previous example, we will convert the `Years` to type int and include it in the dataframe.

In [97]:
```
# transposed dataframe
df_can_t = df_can[years].transpose()

# cast the Years (the index) to type int
df_can_t.index = map(int, df_can_t.index)

# let's label the index. This will automatically be the column name when we reset the i
df_can_t.index.name = 'Year'

# reset index to bring the Year in as a column
df_can_t.reset_index(inplace=True)

# view the changes
df_can_t.head()
```

Out[97]:

Country	Year	Palau	Western Sahara	Marshall Islands	New Caledonia	San Marino	American Samoa	Tuvalu	Sao Tome and Principe	Vanuatu	...

	Country	Year	Palau	Western Sahara	Marshall Islands	New Caledonia	San Marino	American Samoa	Tuvalu	Sao Tome and Principe	Vanuatu	...
0		1980	0	0	0	0	1	0	0	0	0	...
1		1981	0	0	0	0	0	1	1	0	0	...
2		1982	0	0	0	0	0	0	0	0	0	...
3		1983	0	0	0	0	0	0	0	0	0	...
4		1984	0	0	0	0	0	0	1	0	0	...

5 rows × 196 columns

Step 2: 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:

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

where X is the original value, X' is the corresponding normalized value. The formula sets the max value in the dataset to 1, and sets the min value to 0. The rest of the data points are scaled to a value between 0-1 accordingly.

In [98]:
```
# normalize Brazil data
norm_brazil = (df_can_t['Brazil'] - df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max

# normalize Argentina data
norm_argentina = (df_can_t['Argentina'] - df_can_t['Argentina'].min()) / (df_can_t['Arg
```

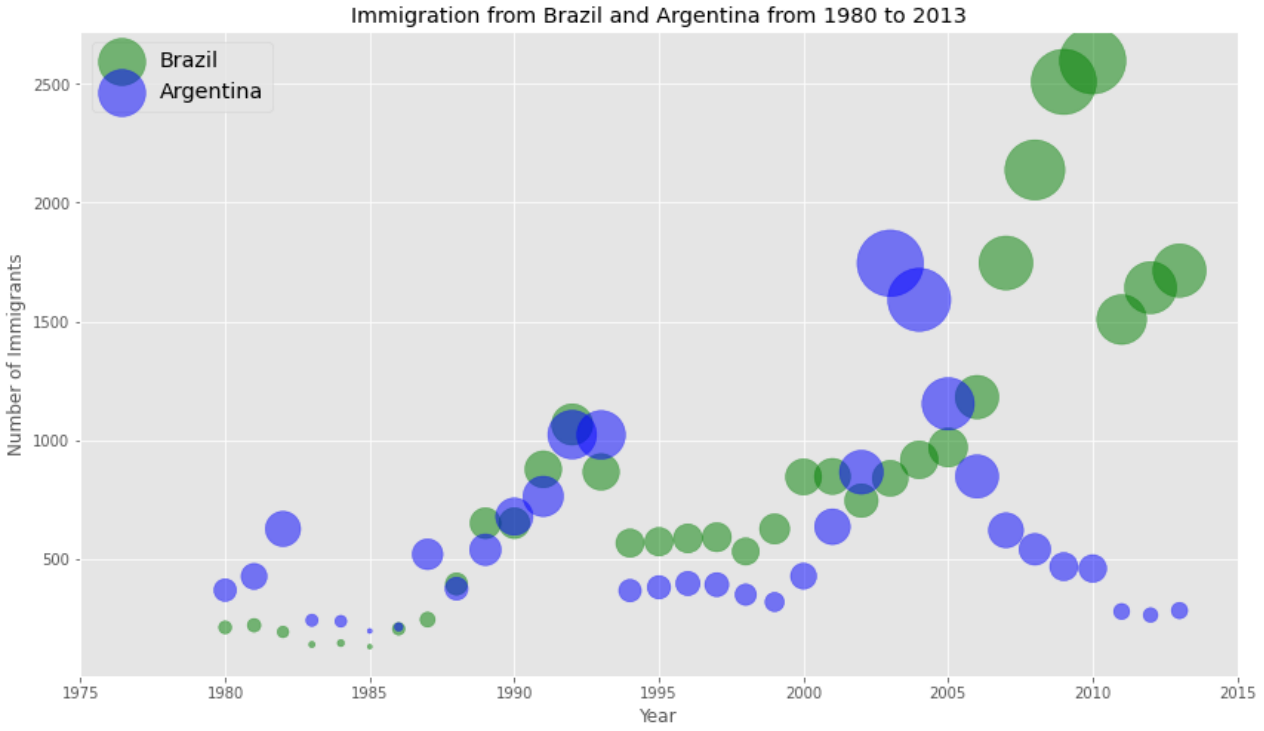
Step 3: Plot the data.

- 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 $\times 2000$).

In [99]:
```
# Brazil
ax0 = df_can_t.plot(kind='scatter',
                    x='Year',
```

```
                      y='Brazil',
                      figsize=(14, 8),
                      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',
                      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 to 2013')
ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')
```

Out[99]: <matplotlib.legend.Legend at 0x174643c1580>



In [100…

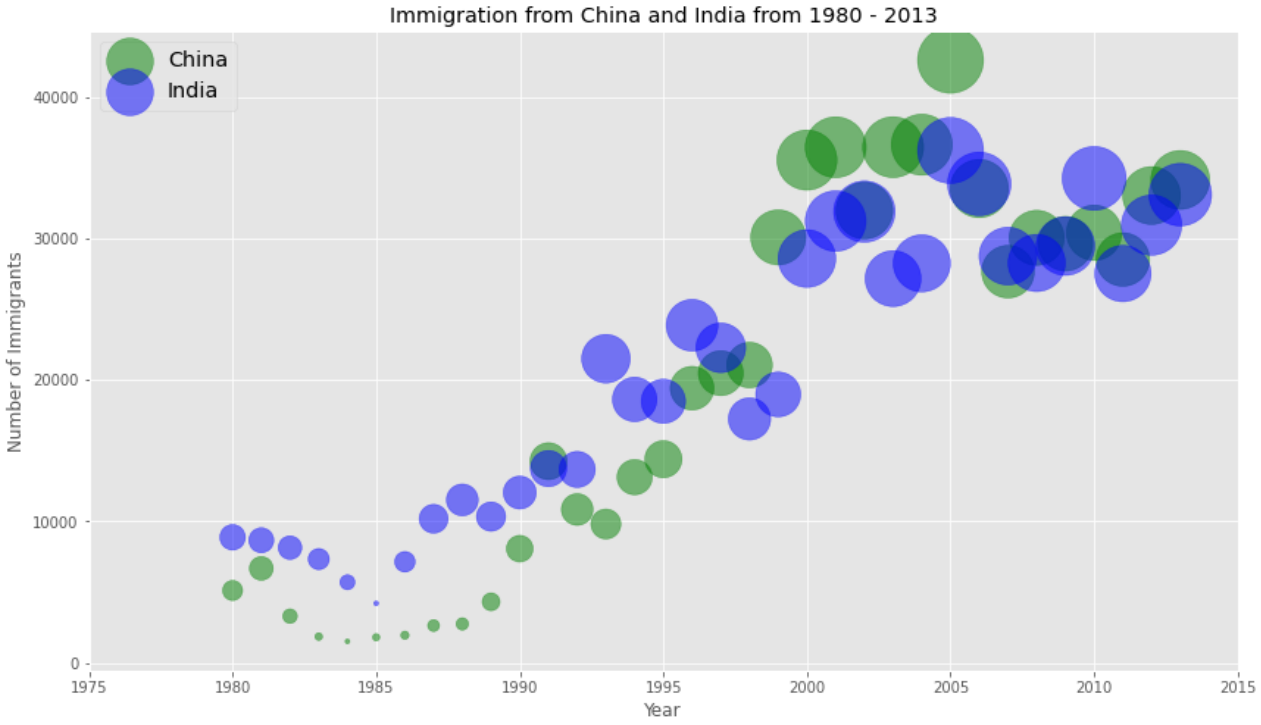
```
# normalize data for china and india
# normalized Chinese data
norm_china = (df_can_t['China'] - df_can_t['China'].min()) / (df_can_t['China'].max() -
# normalized Indian data
norm_india = (df_can_t['India'] - df_can_t['India'].min()) / (df_can_t['India'].max() -
```

In [101…

```
# China
ax0 = df_can_t.plot(kind='scatter',
                        x='Year',
                        y='China',
                        figsize=(14, 8),
```

```
                        alpha=0.5,                  # transparency
                        color='green',
                        s=norm_china * 2000 + 10,  # pass in weights
                        xlim=(1975, 2015)
                       )

# India
ax1 = df_can_t.plot(kind='scatter',
                        x='Year',
                        y='India',
                        alpha=0.5,
                        color="blue",
                        s=norm_india * 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')
```

Out[101… `<matplotlib.legend.Legend at 0x174644e32e0>`



Week 3: Advanced Visualizations and Geospatial Data

Waffle Charts

Waffle Chart:

- an interesting visualization that is normally created to display progress towards goals
- The more the tiles, the more the distribution
- matplotlib does not have a built-in function to create waffle charts, so we'll walk through creating a python function to make one.
- the easiest way to create waffle charts is using python package PyWaffle

Word Clouds

Word Cloud:

- a depiction of the importance or use of different words in a body of text.
- the more a specific word appears in a source of textual data, the bigger it appears in the cloud
- matplotlib does not have a built-in function for word clouds, but a python library for word cloud generation was created by Andreas Mueller.

Seaborn and Regression Plots

Seaborn:

- another data visualization library that is based on matplotlib
- built primarily to provide high-level interface for drawing stats graphics like regression plots, box plots, etc
- with seaborn, you can generate plots with code that is 5x less than with matplotlib

syntax:

- ax = sns.regplot(x = 'col', y = 'col2', data = df, color = 'blue', marker = '+')

Lab: Advanced Visualization Tools

In [102…

```
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

%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

# check for latest version of Matplotlib
print ('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

```
Matplotlib version:  3.4.2
```

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.

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.

In [103…

```
# let's create a new dataframe for these three countries
df_dsn = df_can.loc[['Denmark', 'Norway', 'Sweden'], :]
```

```
# let's take a look at our dataframe
df_dsn
```

Out[103…

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	200
Denmark	Europe	Northern Europe	Developed regions	272	293	299	106	93	73	93	...	62	10
Norway	Europe	Northern Europe	Developed regions	116	77	106	51	31	54	56	...	57	5
Sweden	Europe	Northern Europe	Developed regions	281	308	222	176	128	158	187	...	205	13

3 rows × 38 columns

Step 1. The first step into creating a waffle chart is determing the proportion of each category with respect to the total.

In [104…

```
# compute the proportion of each category with respect to the total
total_values = df_dsn['Total'].sum()
category_proportions = df_dsn['Total'] / total_values

# print out proportions
pd.DataFrame({"Category Proportion": category_proportions})
```

Out[104…

Country	Category Proportion
Denmark	0.322557
Norway	0.192409
Sweden	0.485034

Step 2. The second step is defining the overall size of the `waffle` chart.

In [105…

```
width = 40 # width of chart
height = 10 # height of chart

total_num_tiles = width * height # total number of tiles

print(f'Total number of tiles is {total_num_tiles}.')
```

```
Total number of tiles is 400.
```

Step 3. The third step is using the proportion of each category to determine its respective number of tiles

In [106…

```
# compute the number of tiles for each category
tiles_per_category = (category_proportions * total_num_tiles).round().astype(int)
```

```
# print out number of tiles per category
pd.DataFrame({"Number of tiles": tiles_per_category})
```

Out[106…

	Number of tiles
Country	
Denmark	129
Norway	77
Sweden	194

Step 4. The fourth step is creating a matrix that resembles the `waffle` chart and populating it.

In [107…

```
# initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width), dtype = np.uint)

# 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 cor
        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 populated!
```

In [108…

```
# view the matrix
waffle_chart
```

Out[108…

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
```

```
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]],
      dtype=uint32)
```

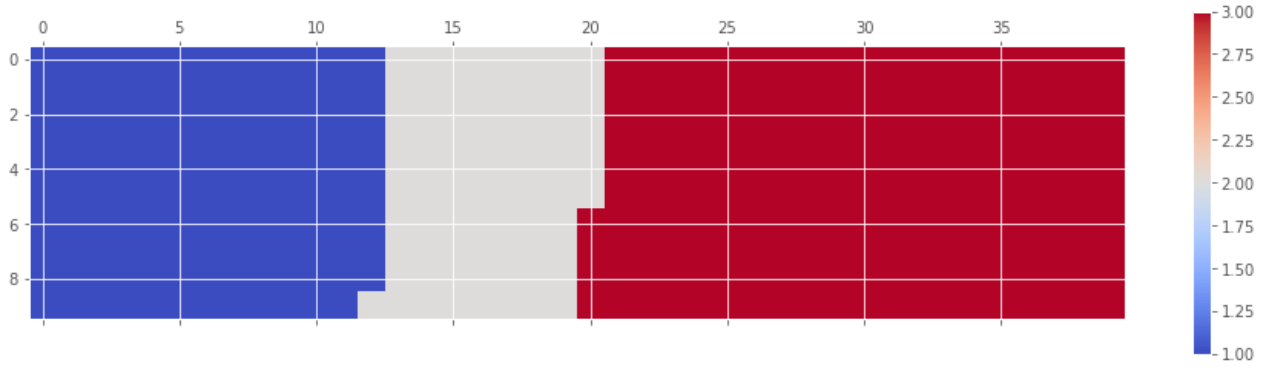
As expected, the matrix consists of three categories and the total number of each category's instances matches the total number of tiles allocated to each category.

Step 5. Map the `waffle` chart matrix into a visual.

In [109…

```
# 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()
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```

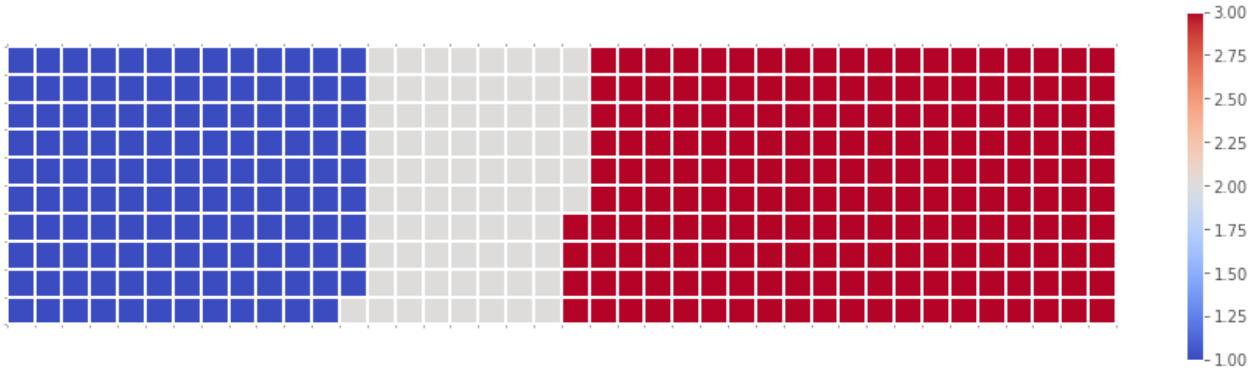


Step 6. Prettify the chart.

In [110…

```
# 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([])
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```

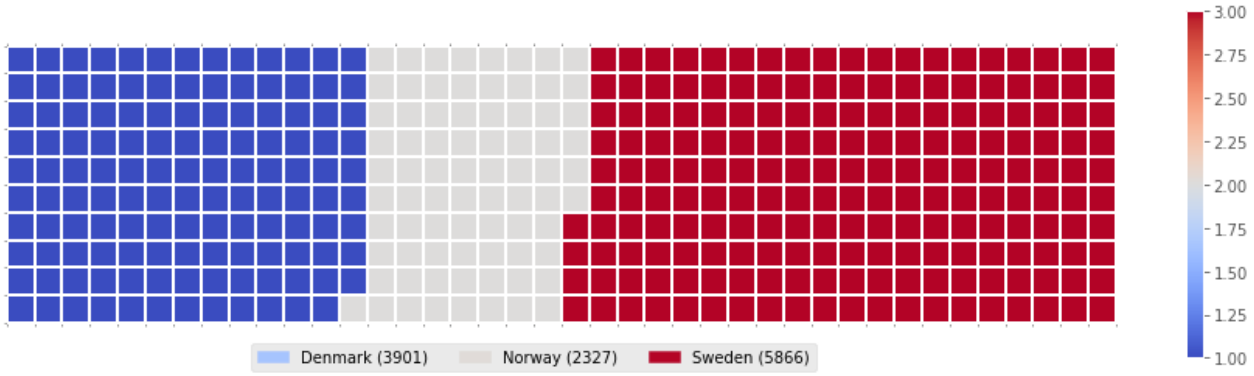


Step 7. Create a legend and add it to chart.

In [111…

```
# 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([])

# compute cumulative sum of individual categories to match color schemes between chart
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)
          )
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



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:

> 1. **categories**: Unique categories or classes in dataframe.
> 2. **values**: Values corresponding to categories or classes.
> 3. **height**: Defined height of waffle chart.
> 4. **width**: Defined width of waffle chart.
> 5. **colormap**: Colormap class
> 6. **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.

In [112…

```
def create_waffle_chart(categories, values, height, width, colormap, value_sign=''):

    # compute the proportion of each category with respect to the total
    total_values = sum(values)
    category_proportions = [(float(value) / total_values) for value in values]

    # compute the total number of tiles
    total_num_tiles = width * height # 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 categor

    # print out number of tiles per category
    for i, tiles in enumerate(tiles_per_category):
        print (df_dsn.index.values[i] + ': ' + str(tiles))

    # 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

    # 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 dridlines 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 ch
    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)
    )
    plt.show()
```

In [113…

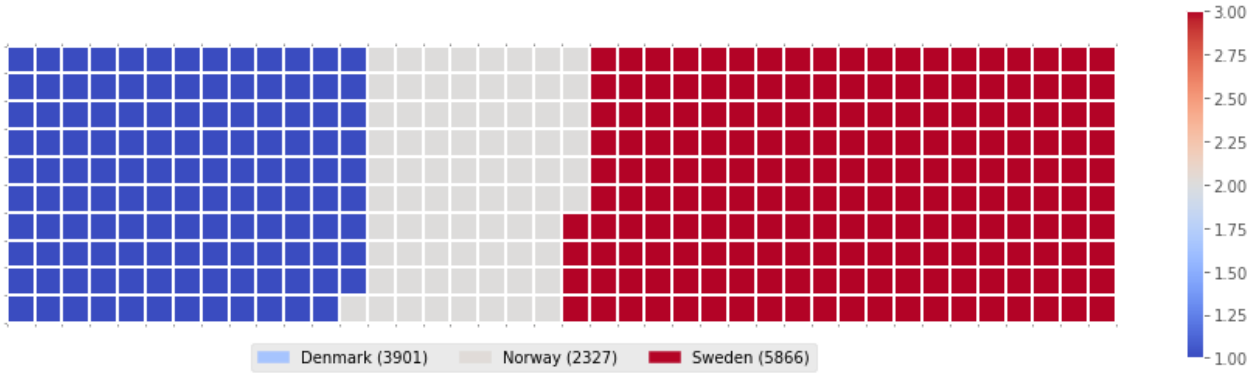
```
# now create a waffle chart by calling the function

# define 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
```

In [114…

```
# call the function
create_waffle_chart(categories, values, height, width, colormap)
```

```
Total number of tiles is 400
Denmark: 129
Norway: 77
Sweden: 194
<Figure size 432x288 with 0 Axes>
```



A new library, PyWaffle, has been created to handle waffle charts.

link to pywaffle github.

In [115…

```
# !pip3 install PyWaffle
```

In [116…

```
# example usage
# python program to generate Waffle Chart

# importing all necessary requirements
import pandas as pd
import matplotlib.pyplot as plt
from pywaffle import Waffle

# creation of a dataframe
data ={'phone': ['Xiaomi', 'Samsung',
                 'Apple', 'Nokia', 'Realme'],
       'stock': [44, 12, 8, 5, 3]
     }

df = pd.DataFrame(data)

# To plot the waffle Chart
fig = plt.figure(
    FigureClass = Waffle,
    rows = 5,
    values = df.stock,
    labels = list(df.phone)
)
```

Word Clouds

A python package already exists to create word clouds: word_cloud

link to github

In [117…
```
# !pip3 install wordcloud
```

In [118…
```
# import package and its set of stopwords
from wordcloud import WordCloud, STOPWORDS

print ('Wordcloud is installed and imported!')
```

```
Wordcloud is installed and imported!
```

In [119…
```
import urllib

# open the file and read it into a variable alice_novel
alice_novel = urllib.request.urlopen('https://cf-courses-data.s3.us.cloud-object-storag
```

In [120…
```
# use stopwords function set to remove redundant stopwords
stopwords = set(STOPWORDS)
```

In [121…
```
# create word cloud object to generate a word cloud
# let's use only first 2000 words

# 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)
```

Out[121…
```
<wordcloud.wordcloud.WordCloud at 0x17463b82c10>
```

In [122…
```
# display the word cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

In [123…

```
# resize cloud to view less frequent words better
fig = plt.figure(figsize=(14, 18))

# display the cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

In [124…

```
# said isn't informative, so let's remove it
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(figsize=(14, 18))

plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

In [125…

```
# superimpose the words onto a mask of any shape
# save mask to alice_mask
alice_mask = np.array(Image.open(urllib.request.urlopen('https://cf-courses-data.s3.us.
```

In [126…

```
# view the mask shape
fig = plt.figure(figsize=(14, 18))

plt.imshow(alice_mask, cmap=plt.cm.gray, interpolation='bilinear')
plt.axis('off')
plt.show()
```

In [127…

```
# instantiate a word cloud object
alice_wc = WordCloud(background_color='white', max_words=2000, mask=alice_mask, stopwor

# generate the word cloud
alice_wc.generate(alice_novel)

# display the word cloud
fig = plt.figure(figsize=(14, 18))

plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

In [128…

```
# try out word clouds with immigration dataset by generating sample text data

# get total immigration
total_immigration = df_can['Total'].sum()
total_immigration
```

Out[128… 6409153

In [129…

```
# using countries with single-word names, duplicate each country's name based on how mu
max_words = 90
word_string = ''
for country in df_can.index.values:
    # check if country's name is a single-word name
    if country.count(" ") == 0:
        repeat_num_times = int(df_can.loc[country, 'Total'] / total_immigration * max_w
        word_string = word_string + ((country + ' ') * repeat_num_times)
```

```
# display the generated text
word_string
```

Out[129…
```
'Colombia Morocco Egypt Portugal Guyana Haiti Romania Jamaica France Lebanon Poland Paki
stan Pakistan Pakistan Philippines Philippines Philippines Philippines Philippines Phili
ppines Philippines China China China China China China China China China India India Ind
ia India India India India India India '
```

In [130…
```
# no stopwords here, so no need to pass them through when making the cloud
# create the word cloud
wordcloud = WordCloud(background_color='white').generate(word_string)

print('Word cloud created!')
```

```
Word cloud created!
```

In [131…
```
# display the cloud
plt.figure(figsize=(14, 18))

plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Regression Plots

In [132…
```
# import library
import seaborn as sns

print('Seaborn installed and imported!')
```

```
Seaborn installed and imported!
```

In [133…
```
# create new dataframet hat stores total number of landed immigrants
# 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)

# rename columns
df_tot.columns = ['year', 'total']

# view the final dataframe
df_tot.head()
```

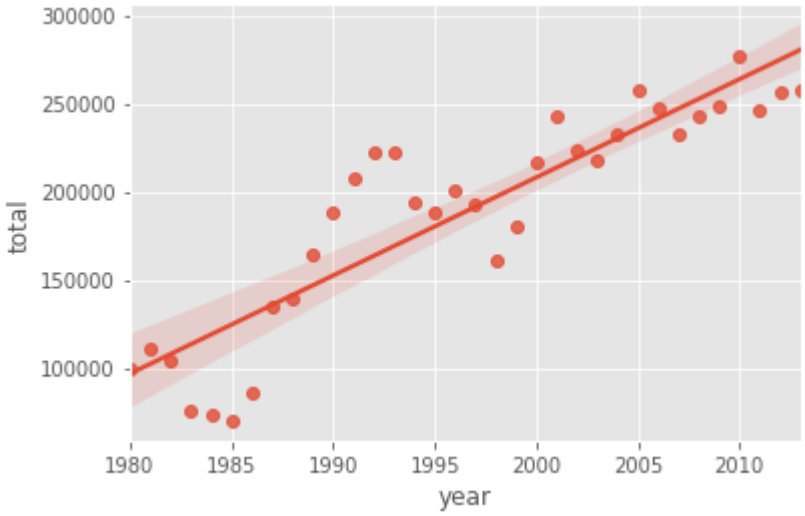
Out[133…

	year	total
0	1980.0	99137
1	1981.0	110563
2	1982.0	104271
3	1983.0	75550
4	1984.0	73417

In [134…

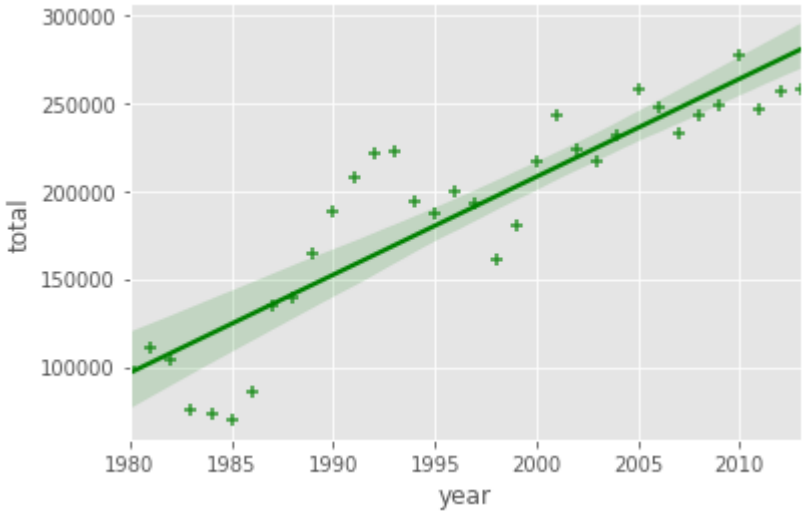
```
# generate regression plot
sns.regplot(x='year', y='total', data=df_tot)
```

Out[134… <AxesSubplot:xlabel='year', ylabel='total'>



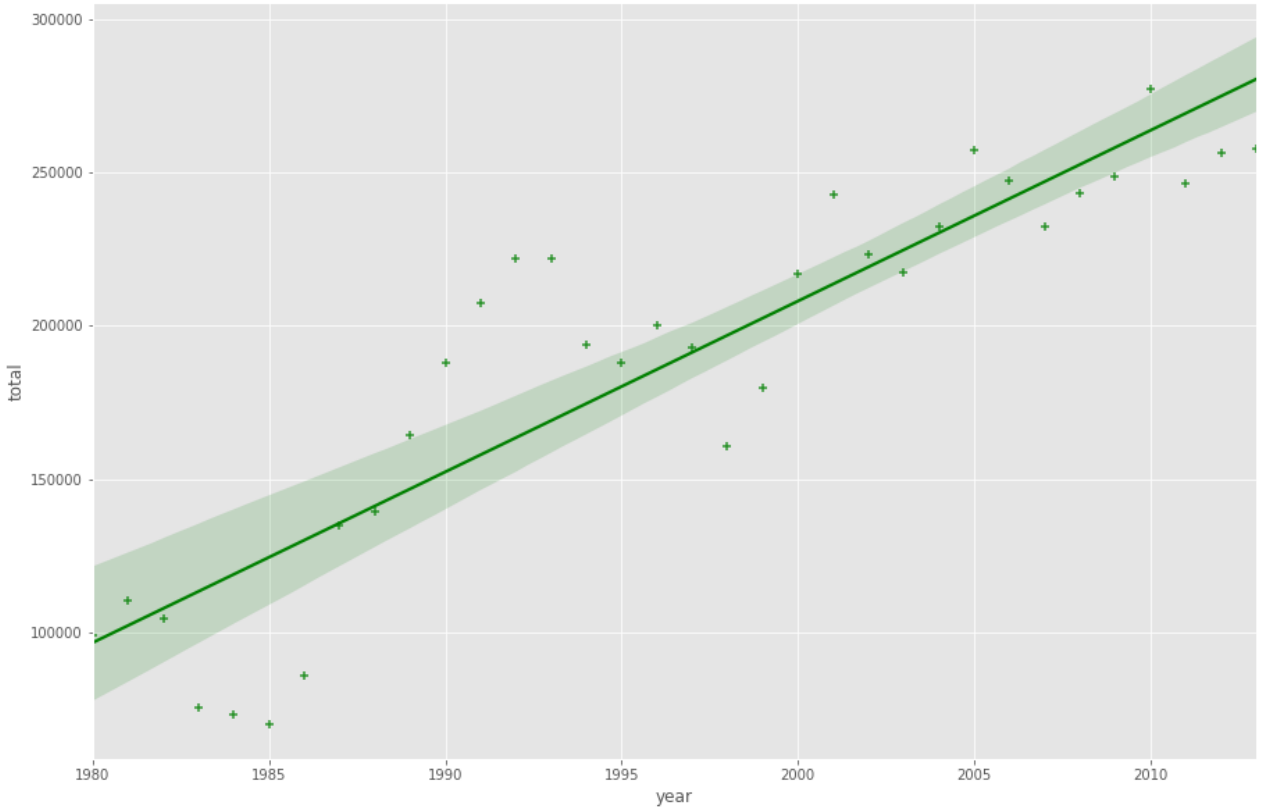
In [135…

```
# customize the color and marker
ax = sns.regplot(x='year', y='total', data=df_tot, color='green', marker='+')
plt.show()
```



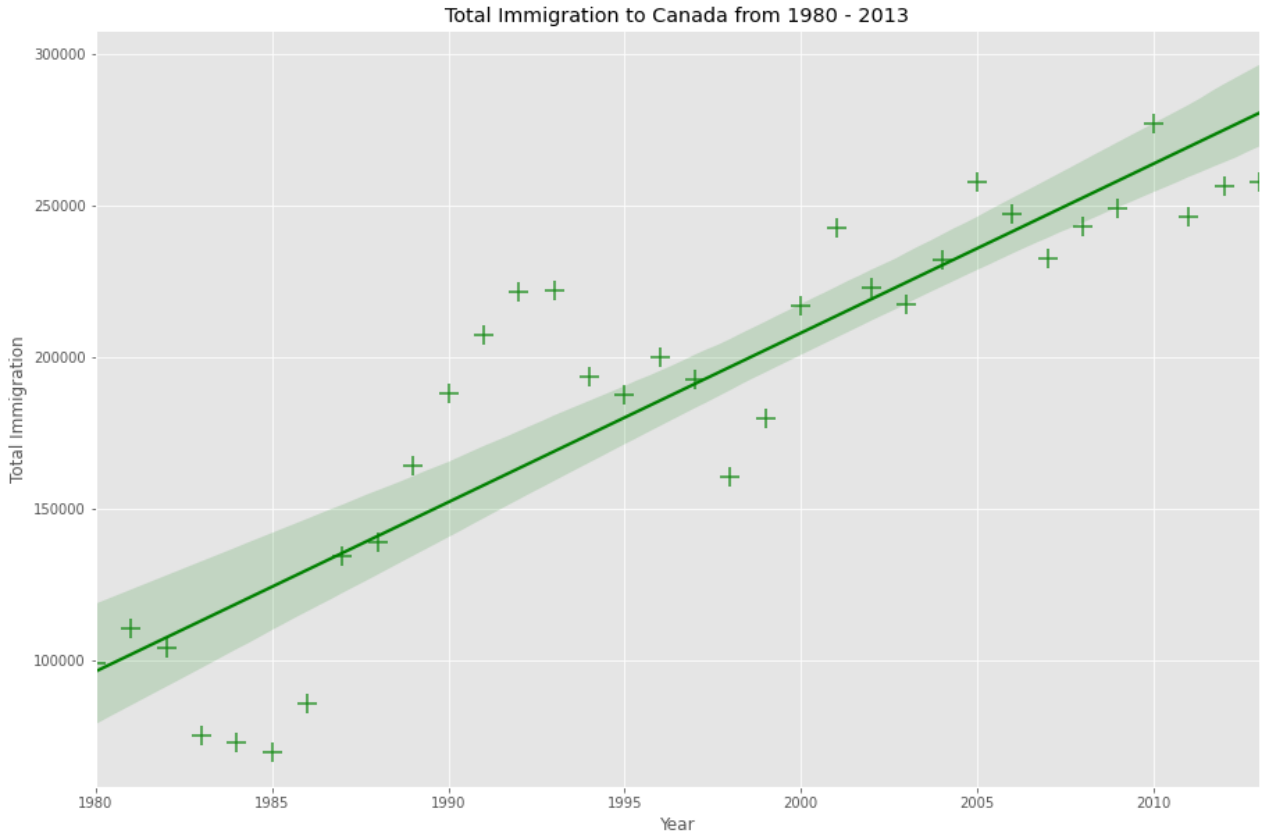
In [136…

```
# increase figure size
plt.figure(figsize=(15, 10))
sns.regplot(x='year', y='total', data=df_tot, color='green', marker='+')
plt.show()
```



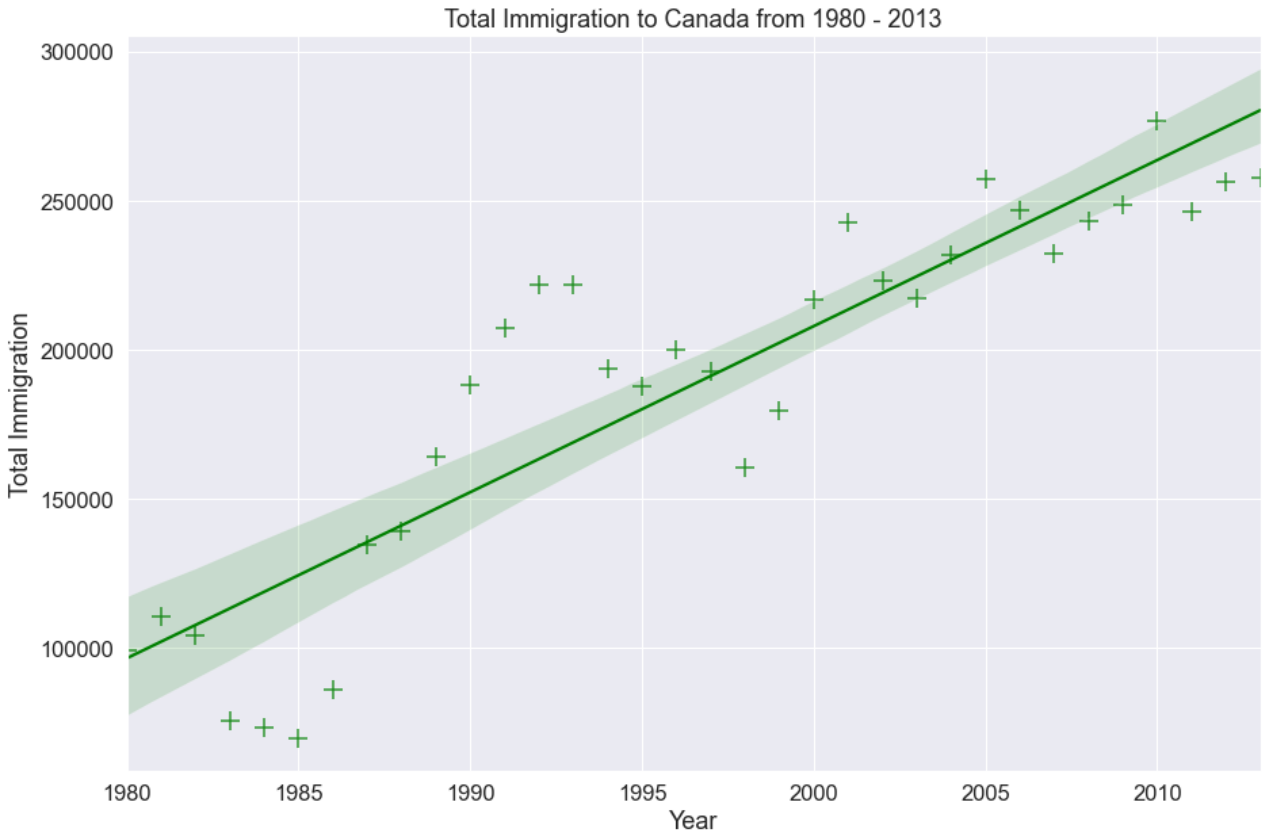
In [137…

```
# increase marker size to match new figure size
plt.figure(figsize=(15, 10))
ax = sns.regplot(x='year', y='total', data=df_tot, color='green', marker='+', scatter_k

ax.set(xlabel='Year', ylabel='Total Immigration') # add x- and y-labels
ax.set_title('Total Immigration to Canada from 1980 - 2013') # add title
plt.show()
```



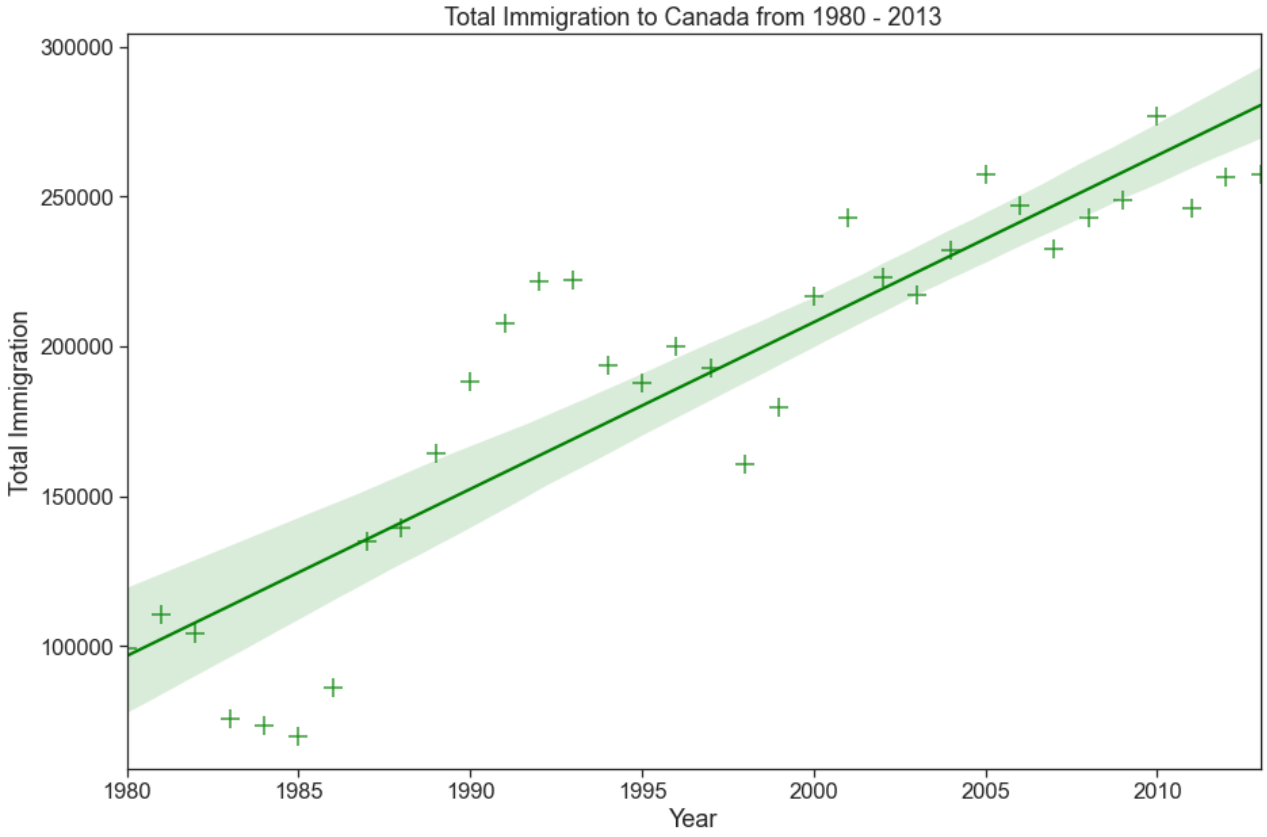
In [138…

```
# increase font size of tickmark labels, title, and labels
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_k
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



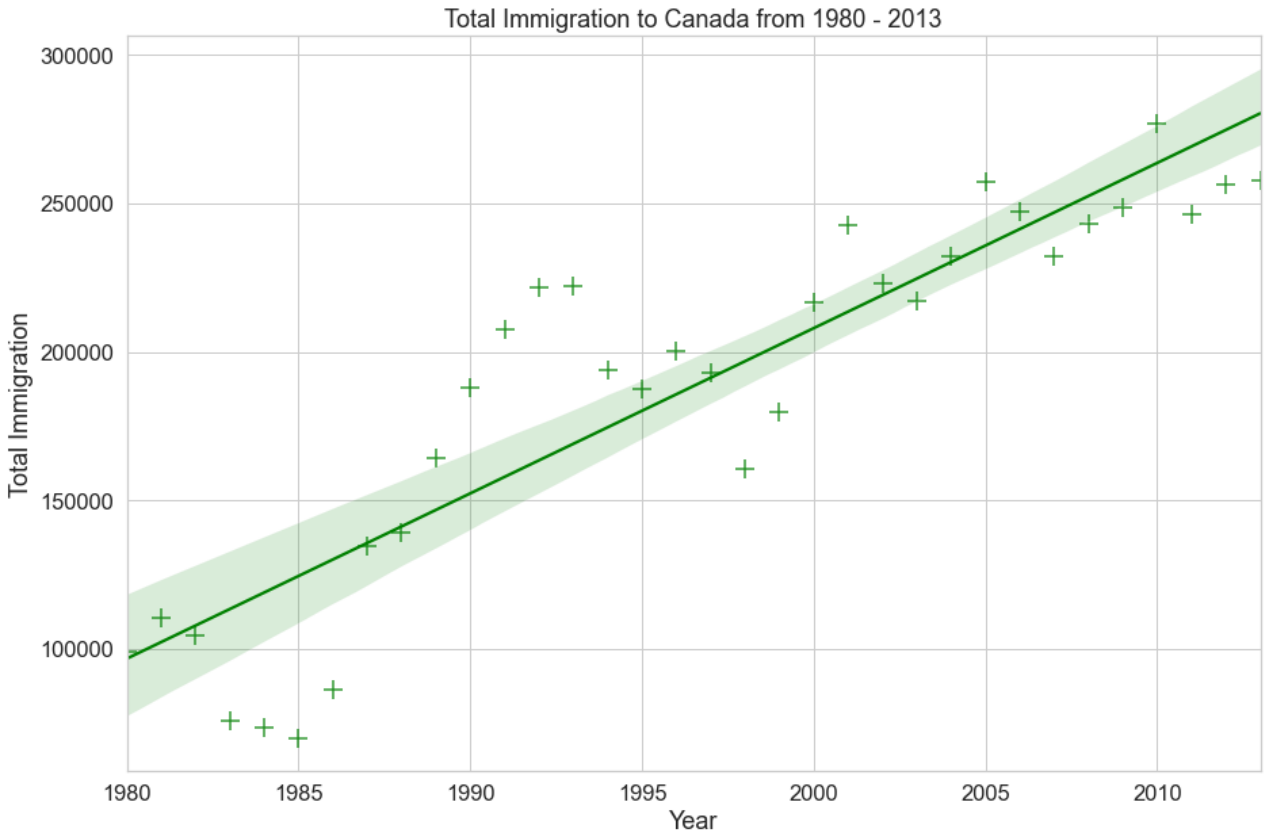
In [139…

```
# change background color
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_k
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



In [140…

```
# add grid lines
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_k
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```

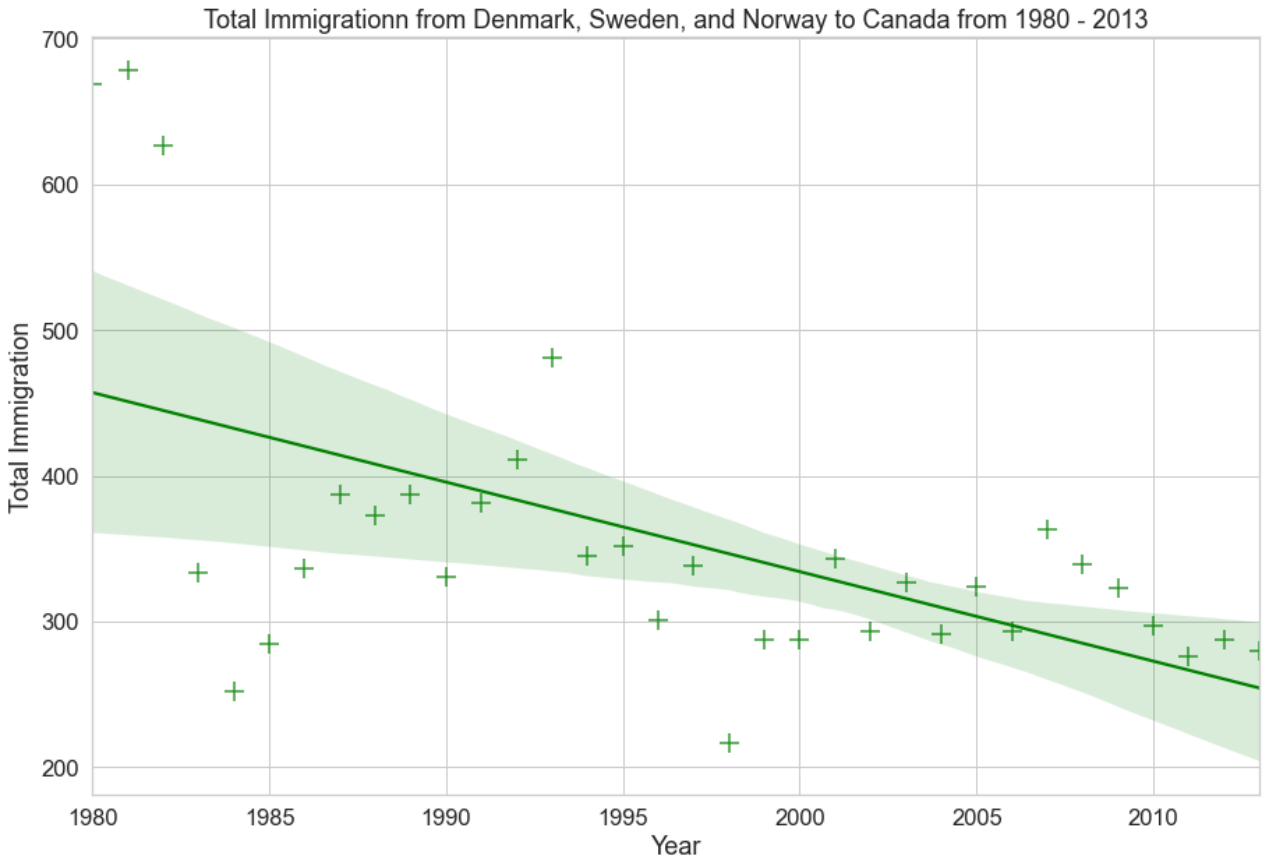


In [141…

```
# use seaborn to create a scatterplot with regression line to visualize total immigrati
# create df_countries dataframe
df_countries = df_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose()

# create df_total by summing across three countries for each year
df_total = pd.DataFrame(df_countries.sum(axis=1))

# 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)

# define figure size
plt.figure(figsize=(15, 10))

# define background style and font size
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
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigrationn from Denmark, Sweden, and Norway to Canada from 1980 -
```

Out[141…

```
Text(0.5, 1.0, 'Total Immigrationn from Denmark, Sweden, and Norway to Canada from 1980
- 2013')
```



Intro to Folium

Folium:

- a powerful data visualization library in python that was built primarily to help visualize geospatial data in an interactive way
- you can create a map of any location in the world as long as you know the longitude & latitude
- helps create several types of leaflet maps
- street level map, stamen map, world map

Syntax (world map):

- world_map = folium.Map(location = [], zoom_start = #])
- world_map

Syntax (stamen toner):

- world_map = folium.Map(location = [], zoom-start = #, tiles = 'Stamen Toner'])
- world_map

Syntax (stamen terrain):

- world_map = folium.Map(location = [], zoom-start = #, tiles = 'Stamen Terrain'])
- world_map

Maps with Markers

Markers:

- You can superimpose markers on top of a map using Folium
- Create a featuregroup
- Create what is called children and add them to the group
- Then add the featuregroup to the map
- use the marker function and the pop-up parameters to pass in whatever text we need to the marker

Choropleth Map

Choropleth Maps:

- You can create choropleth maps with Folium
- a choropleth map is a thematic map in which areas are shaded in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per capita income.
- the higher the measurement, the darker the color
- in order to create a choropleth map, folium requires a Geo JSON file that includes the geospatial data of the region.

Syntax:

- world_map = folium.Map(zoom_start = 2, tiles = 'Mapbox Bright')
- world_geo = r'geo_JSON_file.json'
- world_map.choropleth(geo_path = world_geo, data = df, columns = [col1, col2], key_on = 'feature.properties.name', fill_color = 'YlOrRd', legend_name = 'name')
- world_map

Lab: Visualizing Geospatial Data

In [142…
```
# !pip3 install folium
```

In [143…
```
# import library
import folium

print('Folium installed and imported!')
```

```
Folium installed and imported!
```

In [144…
```
# define the world map
world_map = folium.Map()

# display world map
world_map
```

Out[144… Make this Notebook Trusted to load map: File -> Trust Notebook

You can customize this default definition of the world map by specifying the centre of your map, and the initial 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 initial 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.

In [145…

```
# define the world map centered around Canada with a low zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4)

# display world map
world_map
```

Out[145… Make this Notebook Trusted to load map: File -> Trust Notebook

In [146…

```
# define the world map centered around Canada with a higher zoom level
world_map = folium.Map(location=[56.130, -106.35], zoom_start=8)

# display world map
world_map
```

Out[146… Make this Notebook Trusted to load map: File -> Trust Notebook

In [147…

```
# create a map of mexico with zoom level 4
#define Mexico's geolocation coordinates
mexico_latitude = 23.6345
mexico_longitude = -102.5528

# define the world map centered around Canada with a higher zoom level
mexico_map = folium.Map(location=[mexico_latitude, mexico_longitude], zoom_start=4)

# display world map
mexico_map
```

Out[147… Make this Notebook Trusted to load map: File -> Trust Notebook

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.

In [148…
```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Toner')

# display map
world_map
```

Out[148… Make this Notebook Trusted to load map: File -> Trust Notebook

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.

In [149…
```
# create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Terrain'

# display map
world_map
```

Out[149… Make this Notebook Trusted to load map: File -> Trust Notebook

In [150…

```
# create stamen terrain of mexico
#define Mexico's geolocation coordinates
mexico_latitude = 23.6345
mexico_longitude = -102.5528

# define the world map centered around Canada with a higher zoom level
mexico_map = folium.Map(location=[mexico_latitude, mexico_longitude], zoom_start=6, til

# display world map
mexico_map
```

Out[150… Make this Notebook Trusted to load map: File -> Trust Notebook

Maps with Markers

In [151…
```
# download dataset on police department incidents
df_incidents = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomai

print('Dataset downloaded and read into a pandas dataframe!')
```

```
Dataset downloaded and read into a pandas dataframe!
```

In [152…
```
df_incidents.head()
```

Out[152…

	IncidntNum	Category	Descript	DayOfWeek	Date	Time	PdDistrict	Resolution	Ad
0	120058272	WEAPON LAWS	POSS OF PROHIBITED WEAPON	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 BR
1	120058272	WEAPON LAWS	FIREARM, LOADED, IN VEHICLE, POSSESSION OR USE	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 BR
2	141059263	WARRANTS	WARRANT ARREST	Monday	04/25/2016 12:00:00 AM	14:59	BAYVIEW	ARREST, BOOKED	KEI SHA
3	160013662	NON-CRIMINAL	LOST PROPERTY	Tuesday	01/05/2016 12:00:00 AM	23:50	TENDERLOIN	NONE	JON OFAF
4	160002740	NON-CRIMINAL	LOST PROPERTY	Friday	01/01/2016 12:00:00 AM	00:30	MISSION	NONE	16TH MIS

In [153…
```
df_incidents.shape
```

Out[153…
```
(150500, 13)
```

In [154…
```
# get the first 100 crimes in the df_incidents dataframe
limit = 100
df_incidents = df_incidents.iloc[0:limit, :]
```

In [155…
```
# confirm it consists of 100 crimes
df_incidents.shape
```

Out[155…
```
(100, 13)
```

In [156…
```
# San Francisco latitude and longitude values
latitude = 37.77
longitude = -122.42
```

In [157…
```
# create map and display it
sanfran_map = folium.Map(location=[latitude, longitude], zoom_start=12)

# display the map of San Francisco
sanfran_map
```

Out[157… Make this Notebook Trusted to load map: File -> Trust Notebook

In [158…
```
# superimpose the locations of the crimes onto the 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 how big you want the circle markers to be
            color='yellow',
            fill=True,
            fill_color='blue',
            fill_opacity=0.6
        )
    )

# add incidents to map
sanfran_map.add_child(incidents)
```

Out[158… Make this Notebook Trusted to load map: File -> Trust Notebook

In [159…

```
# add pop up text to the 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 how big you want the circle markers to be
            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)
```

Out[159… Make this Notebook Trusted to load map: File -> Trust Notebook

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:

In [160…

```
# create map and display it
sanfran_map = folium.Map(location=[latitude, longitude], zoom_start=12)

# loop through the 100 crimes and add each to the map
for lat, lng, label in zip(df_incidents.Y, df_incidents.X, df_incidents.Category):
    folium.features.CircleMarker(
        [lat, lng],
        radius=5, # define how big you want the circle markers to be
        color='yellow',
        fill=True,
        popup=label,
        fill_color='blue',
        fill_opacity=0.6
    ).add_to(sanfran_map)

# show map
sanfran_map
```

Out[160… Make this Notebook Trusted to load map: File -> Trust Notebook

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.

In [161…

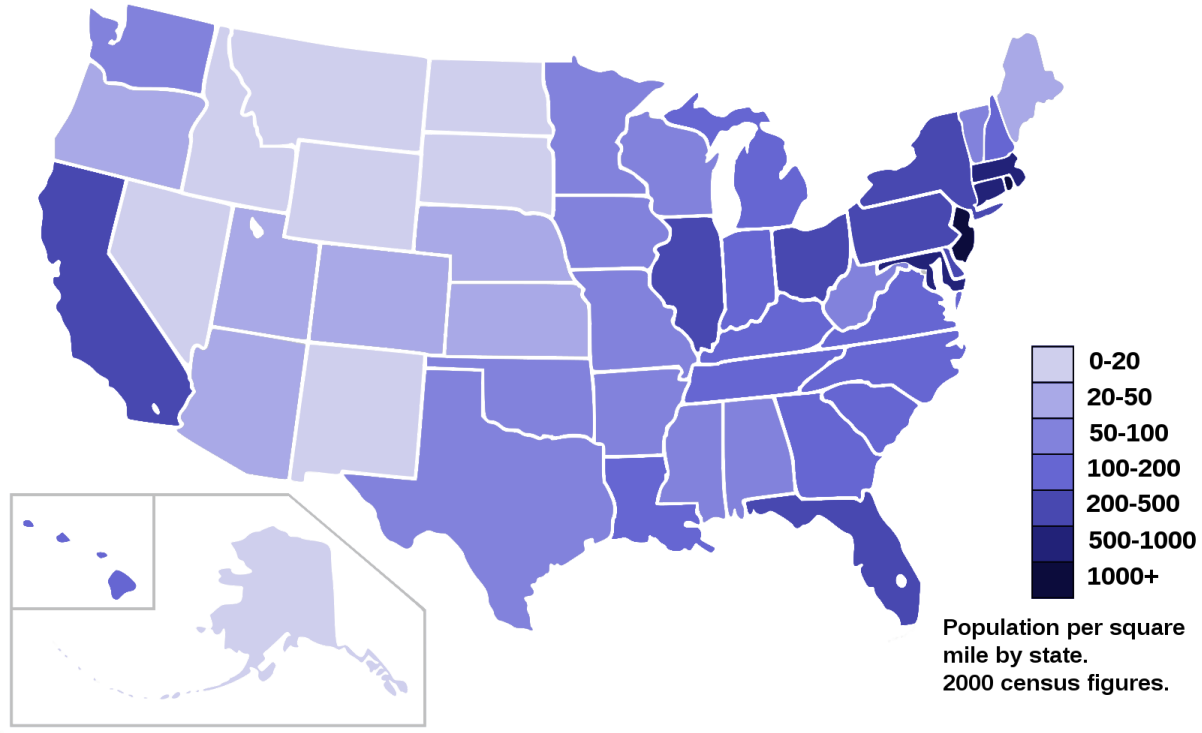
```
from folium import plugins

# 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 through the dataframe and 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)

# display map
sanfran_map
```

Out[161… Make this Notebook Trusted to load map: File -> Trust Notebook

Choropleth Maps

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. Normally, before we can do that, we would need to download a module which *pandas* requires reading in Excel files. This module was **openpyxl** (formerly **xlrd**). For your convenience, we have pre-installed this module, so you would not have to worry about that. Otherwise, you would need to run the following line of code to install the **openpyxl** module:

``` ! pip3 install openpyxl

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. For your convenience, we will be providing you with this file, so let's go ahead and download it. Let's name it **world_countries.json**.

In [162…
```
download countries geojson file
! wget --quiet https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDe

print('GeoJSON file downloaded!')
```

```
GeoJSON file downloaded!
```

In [163…
```
# get dataset
df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSki
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
```
```

```
    skipfooter=2)

print('Data downloaded and read into a dataframe!')
```

```
Data downloaded and read into a dataframe!
```

In [164…

```
# 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'},

# for sake of consistency, let's also make all column labels of type string
df_can.columns = list(map(str, df_can.columns))

# 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)
```

```
data dimensions: (195, 47)
```

In [165…

```
# create world map centered at [0,0] with zoom level 2
world_geo = r'world_countries.json' # geojson file

# create a plain world map
world_map = folium.Map(location=[0, 0], zoom_start=2)
```

In [166…

```
world_map
```

Out[166… Make this Notebook Trusted to load map: File -> Trust Notebook

And now to create a `Choropleth` map, we will use the *choropleth* method with the following main

parameters:

1. `geo_data` , which is the GeoJSON file.
2. `data` , which is the dataframe containing the data.
3. `columns` , which represents the columns in the dataframe that will be used to create the `Choropleth` map.
4. `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.

In [167…
```
df_can.head()
```

Out[167…

	Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	...	2013	Unna
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2004	
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	603	
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	4331	
3	American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	...	0	
4	Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	1	

5 rows × 47 columns

In [168…
```
# drop unnamed columns
df_can.drop(['Unnamed: 43', "Unnamed: 44", "Unnamed: 45", "Unnamed: 46", "Unnamed: 47",
df_can.head(3)
```

Out[168…

	Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	...	2005	2006
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	3436	3009
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1223	856
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3626	4807

3 rows × 39 columns

In [169…
```
# generate choropleth map using the total immigration of each country to Canada from 19
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'
)

# display map
world_map
```

```
c:\users\orgil\appdata\local\programs\python\python39\lib\site-packages\folium\folium.p
y:409: FutureWarning: The choropleth  method has been deprecated. Instead use the new Ch
oropleth class, which has the same arguments. See the example notebook 'GeoJSON_and_chor
opleth' for how to do this.
  warnings.warn(
```

Out[169… Make this Notebook Trusted to load map: File -> Trust Notebook

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!

In [170…
```
world_geo = r'world_countries.json'

# create a numpy array of length 6 and has linear spacing from the minimum total immigr
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 li

# let Folium determine the scale.
world_map = folium.Map(location=[0, 0], zoom_start=2)
world_map.choropleth(
    geo_data=world_geo,
    data=df_can,
    columns=['Country', 'Total'],
    key_on='feature.properties.name',
    threshold_scale=threshold_scale,
    fill_color='YlOrRd',
    fill_opacity=0.7,
    line_opacity=0.2,
    legend_name='Immigration to Canada',
    reset=True
)
world_map
```

Out[170… Make this Notebook Trusted to load map: File -> Trust Notebook

Week 4: Creating Dashboards with Plotly and Dash

Dashboarding

Dashboard:

- real-time visuals
- understand business moving parts
- visually track, analyze, and display key performance indicators (KPI)
- make informed decisions and improve performance
- reduce hours of analyzing

Web-based Dashboarding options in Python:

- Dash from Plotly
- Panel
- Voila
- Streamlit

Dashboard Tools:

- Bokeh
- ipywidgets
- matplotlib
- bowtie
- flask

Dashboarding tools

Data Journalism

Plotly

Plotly:

- interactive, open source plotting library
- supports over 40 unique plot types such as:
  - statistical,
  - financial
  - maps
  - scientific
  - 3D
- Visualizations can be displayed in jupyter notebook, saved to HTML files, or can be used in developing python-built web apps

Plotly Sub-Modules:

- Plotly Graph Objects:
  - low-level interface to figures, traces, and layouts
  - plotly.graph_objects.Figure
- Plotly Express:
  - high-level wrapper for Plotly Graph Objects

getting started with plotly

graph objects in python

plotly express

python API reference for plotly

plotly cheatsheet

data asset exchange

Lab: Plotly Basics

- scatter
- line
- bar
- bubble
- histogram
- pie
- sunburst

In [171…
```
# Import required libraries
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

In [172…
```
# Read the airline data into pandas dataframe
airline_data =  pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdoma
                            encoding = "ISO-8859-1",
                            dtype={'Div1Airport': str, 'Div1TailNum': str,
                                   'Div2Airport': str, 'Div2TailNum': str})
```

In [173…
```
# Preview the first 5 lines of the loaded data
airline_data.head()
```

Out[173…

	Unnamed: 0	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_I
0	1295781	1998	2	4	2	4	1998-04-02	AS	
1	1125375	2013	2	5	13	1	2013-05-13	EV	
2	118824	1993	3	9	25	6	1993-09-25	UA	
3	634825	1994	4	11	12	6	1994-11-12	HP	
4	1888125	2017	3	8	17	4	2017-08-17	UA	

5 rows × 110 columns

In [174…
```
# Shape of the data
airline_data.shape
```

Out[174… `(27000, 110)`

In [175…
```
# Randomly sample 500 data points. Setting the random state to be 42 so that we get sam
data = airline_data.sample(n=500, random_state=42)
```

In [176…
```
# Get the shape of the trimmed data
data.shape
```

Out[176… (500, 110)

PLOTLY.GRAPH_OBJECTS

Scatter Plot

How departure time changes with respect to airport distance:

In [177…
```
# First we create a figure using go.Figure and adding trace to it through go.scatter
fig = go.Figure(data=go.Scatter(x=data['Distance'], y=data['DepTime'], mode='markers',

# Updating layout through `update_layout`. Here we are adding title to the plot and pro
fig.update_layout(title='Distance vs Departure Time', xaxis_title='Distance', yaxis_tit

# Display the figure
fig.show()
```

Line Plot

Extract average monthly arrival delay time and see how it changes over the year:

In [178…
```
# Group the data by Month and compute average over arrival delay time.
line_data = data.groupby('Month')['ArrDelay'].mean().reset_index()
```

In [179…
```
# Display the data
line_data
```

Out[179…

	Month	ArrDelay
0	1	2.232558
1	2	2.687500
2	3	10.868421
3	4	6.229167
4	5	-0.279070
5	6	17.310345
6	7	5.088889
7	8	3.121951
8	9	9.081081
9	10	1.200000
10	11	-3.975000
11	12	3.240741

In [180…
```
# create a line plot with x-axis = month, y-axis = computed average delay time
# note: scatter and line plot vary by updating the 'mode'
fig = go.Figure(data=go.Scatter(x=line_data['Month'], y=line_data['ArrDelay'], mode='li

# update plot title and axis titles
fig.update_layout(title='Month vs Average Flight Delay Time', xaxis_title='Month', yaxi
fig.show()
```

PLOTLY.EXPRESS

Bar Chart

Extract the number of flights from a specific airline that goes to a destination:

In [181…
```
# Group the data by destination state and reporting airline. Compute total number of fl
bar_data = data.groupby(['DestState'])['Flights'].sum().reset_index()
```

In [182…
```
# Display the data
bar_data
```

Out[182…

	DestState	Flights
0	AK	4.0
1	AL	3.0
2	AZ	8.0
3	CA	68.0
4	CO	20.0
5	CT	5.0
6	FL	32.0
7	GA	27.0
8	HI	5.0
9	IA	1.0
10	ID	1.0
11	IL	33.0
12	IN	6.0
13	KS	1.0
14	KY	14.0

	DestState	Flights
15	LA	4.0
16	MA	10.0
17	MD	7.0
18	MI	16.0
19	MN	11.0
20	MO	18.0
21	MT	3.0
22	NC	13.0
23	NE	2.0
24	NH	1.0
25	NJ	5.0
26	NM	1.0
27	NV	13.0
28	NY	21.0
29	OH	9.0
30	OK	6.0
31	OR	3.0
32	PA	14.0
33	PR	2.0
34	RI	1.0
35	SC	1.0
36	TN	14.0
37	TX	60.0
38	UT	7.0
39	VA	11.0
40	VI	1.0
41	WA	10.0
42	WI	8.0

In [183…

```
# Use plotly express bar chart function px.bar. Provide input data, x and y axis variab
# This will give total number of flights to the destination state.
fig = px.bar(bar_data, x="DestState", y="Flights", title='Total number of flights to th
fig.show()
```

Bubble Chart

Get the number of flights as per reporting airline:

In [184…
```
# Group the data by reporting airline and get number of flights
bub_data = data.groupby('Reporting_Airline')['Flights'].sum().reset_index()
```

In [185…
```
# view the data
bub_data
```

Out[185…

	Reporting_Airline	Flights
0	9E	5.0
1	AA	57.0
2	AS	14.0
3	B6	10.0
4	CO	12.0
5	DL	66.0
6	EA	4.0
7	EV	11.0
8	F9	4.0

	Reporting_Airline	Flights
9	FL	3.0
10	HA	3.0
11	HP	7.0
12	KH	1.0
13	MQ	27.0
14	NK	3.0
15	NW	26.0
16	OH	8.0
17	OO	28.0
18	PA (1)	1.0
19	PI	1.0
20	PS	1.0
21	TW	14.0
22	UA	51.0
23	US	43.0
24	VX	1.0
25	WN	86.0
26	XE	6.0
27	YV	6.0
28	YX	1.0

In [186…

```
# create bubble chart using bub_data with x-axis = reporting airline, y-axis = flights
# provide title to the chart
# update the size of the bubbles based on number of flights using 'size'
# update name of the hover tooltip to reporting_airline using hover_name parameter
fig = px.scatter(bub_data, x="Reporting_Airline", y="Flights", size="Flights",
                 hover_name="Reporting_Airline", title='Reporting Airline vs Number of
fig.show()
```

Histogram

Get distribution of arrival delay:

In [187…
```
# Set missing values to 0
data['ArrDelay'] = data['ArrDelay'].fillna(0)
```

In [188…
```
# use px.histogram and pass the dataset
# use x = ArrDelay
fig = px.histogram(data, x="ArrDelay")
fig.show()
```

Pie Chart

Get the proportion of distance group by month (month indicated by numbers):

In [189…

```
# Use px.pie function to create the chart. Input dataset.
# Values parameter will set values associated to the sector. 'Month' feature is passed
# labels for the sector are passed to the `names` parameter.
fig = px.pie(data, values='Month', names='DistanceGroup', title='Distance group proport
fig.show()
```

Sunburst Charts

Get a hierarchical view in the order of month and destination state holding value of number of flights:

In [190…

```
# create sunburst chart with px.sunburst
# define hierarchy of sectors from root to leaves in 'path'. here, use month to deststa
# set sector values in value parameter (flights)
```

```
fig = px.sunburst(data, path=['Month', 'DestStateName'], values='Flights')
fig.show()
```

Dash

Dash:

- an open-source user interface python library from plotly
- easy to build GUI
- declarative and reactive
- rendered in web browser and can be deployed to servers
- inherently cross-platform and mobile-ready

Dash Components:

- Core Components
  - import dash_core_components as dcc
  - describe higher level components that are interactive and are generated with javascript, html, and css through the react.js library
  - some core components include:
    - creating a slider, input area, check items, datepicker, etc

- HTML components
  - import dash_html_components as html
  - has a component for every HTML tag
  - keyword arguments describe the HTML attributes like:
    - style, classname, id

dash user guide

dash core components

dash html components

Lab: Dash Basics

- HTML components
- Core components

In [191…
```
# ! pip3 install pandas dash
```

In [192…
```
# Import required packages
import pandas as pd
import plotly.express as px
```

In [193…
```
# Read the airline data into pandas dataframe
# Read the airline data into pandas dataframe
airline_data =  pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdoma
                            encoding = "ISO-8859-1",
                            dtype={'Div1Airport': str, 'Div1TailNum': str,
                                   'Div2Airport': str, 'Div2TailNum': str})
```

In [194…
```
# Preview the first 5 lines of the loaded data
airline_data.head()
```

Out[194…

	Unnamed: 0	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_I
0	1295781	1998	2	4	2	4	1998-04-02	AS	
1	1125375	2013	2	5	13	1	2013-05-13	EV	
2	118824	1993	3	9	25	6	1993-09-25	UA	
3	634825	1994	4	11	12	6	1994-11-12	HP	
4	1888125	2017	3	8	17	4	2017-08-17	UA	

5 rows × 110 columns

In [195…
```
# Shape of the data
airline_data.shape
```

Out[195… (27000, 110)

In [196…
```
# Randomly sample 500 data points. Setting the random state to be 42 so that we get sam
data = airline_data.sample(n=500, random_state=42)
```

In [197…
```
# Get the shape of the trimmed data
data.shape
```

Out[197… (500, 110)

In [198…
```
# get proportion of distance group (250 mile distance interval) by month (month indicat
# Pie Chart Creation
fig = px.pie(data, values='Month', names='DistanceGroup', title='Distance group proport
fig.show()
```

In [ ]:
```
# Import required libraries
```

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from jupyter_dash import JupyterDash
```

In [201…

```
# ! pip3 install jupyter_dash
```

In [ ]:

```
JupyterDash.infer_jupyter_proxy_config()
```

In [ ]:

```
# needs to be run again in a separate cell due to a jupyterdash bug
JupyterDash.infer_jupyter_proxy_config()
```

In [ ]:

```
# Create a dash application
app = JupyterDash(__name__)

# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1
# Add description about the graph using HTML P (paragraph) component
# Finally, add graph component.
app.layout = html.Div(children=[html.H1('Airline Dashboard',
                                        style={'textAlign': 'center',
                                               'color': '#503D36',
                                                'font-size': 40}),
                                html.P('Proportion of distance group (250 mile distance
                                       style={'textAlign':'center', 'color': '#F57241'
                                dcc.Graph(figure=fig)])
if __name__ == '__main__':
    app.run_server(mode="inline", host="localhost")
```

Interactive Dashboards

Dash Callbacks

- callback function is a python function that is automatically called by dash whenever an input component's property changes
  - decorator
  - @app.callback

Python decorators

Python decorators 2

dash callbacks

dash app library

Lab: Interactivity

- user input
- callbacks

In [ ]:

```
# Import required libraries
import pandas as pd
import plotly.graph_objects as go
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output

# Create a dash application
app = dash.Dash(__name__)

# Read the airline data into pandas dataframe
airline_data =  pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdoma
                            encoding = "ISO-8859-1",
                            dtype={'Div1Airport': str, 'Div1TailNum': str,
                                   'Div2Airport': str, 'Div2TailNum': str})

# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1
# Add a html.Div and core input text component
# Finally, add graph component.
app.layout = html.Div(children=[ html.H1('Airline Performance Dashboard',
                                 style={'textAlign': 'center', 'color': '#503D36',
                                 'font-size': 40}),
                                 html.Div(["Input Year: ", dcc.Input(id='input-year', va
                                 type='number', style={'height':'50px', 'font-size': 35}
                                 style={'font-size': 40}),
                                 html.Br(),
                                 html.Br(),
                                 html.Div(dcc.Graph(id='line-plot')),
                                 ])

# add callback decorator
@app.callback( Output(component_id='line-plot', component_property='figure'),
               Input(component_id='input-year', component_property='value'))

# Add computation to callback function and return graph
def get_graph(entered_year):
    # Select 2019 data
    df =  airline_data[airline_data['Year']==int(entered_year)]

    # Group the data by Month and compute average over arrival delay time.
    line_data = df.groupby('Month')['ArrDelay'].mean().reset_index()

    fig = go.Figure(data=go.Scatter(x=line_data['Month'], y=line_data['ArrDelay'], mode
    fig.update_layout(title='Month vs Average Flight Delay Time', xaxis_title='Month',
    return fig

# Run the app
if __name__ == '__main__':
    app.run_server()
```

In [ ]:

```
# Import required libraries
import pandas as pd
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
```

```
from jupyter_dash import JupyterDash
import plotly.graph_objects as go
```

In [ ]:
```
JupyterDash.infer_jupyter_proxy_config()
```

In [ ]:
```
 # needs to be run again in a separate cell due to a jupyterdash bug
JupyterDash.infer_jupyter_proxy_config()
```

In [ ]:
```
# Create a dash application
app = JupyterDash(__name__)

# Read the airline data into pandas dataframe
airline_data =  pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdoma
                            encoding = "ISO-8859-1",
                            dtype={'Div1Airport': str, 'Div1TailNum': str,
                                   'Div2Airport': str, 'Div2TailNum': str})

# To do
# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1
# Add a html.Div and core input text component
# Finally, add graph component.


# add callback decorator

# Add computation to callback function and return graph
def get_graph(entered_year):


    return fig

# Run the app
if __name__ == '__main__':
    app.run_server(mode='jupyterlab')
```

Lab: Flight Delay Time Statistics Dashboard

In [ ]:
```
# Import required libraries
import pandas as pd
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
from jupyter_dash import JupyterDash
import plotly.express as px

# Create a dash application
app = JupyterDash(__name__)
JupyterDash.infer_jupyter_proxy_config()


# Read the airline data into pandas dataframe
airline_data =  pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdoma
```

```
                                encoding = "ISO-8859-1",
                                dtype={'Div1Airport': str, 'Div1TailNum': str,
                                       'Div2Airport': str, 'Div2TailNum': str})

""" Compute_info function description

This function takes in airline data and selected year as an input and performs computat

Arguments:
    airline_data: Input airline data.
    entered_year: Input year for which computation needs to be performed.

Returns:
    Computed average dataframes for carrier delay, weather delay, NAS delay, security d

"""
def compute_info(airline_data, entered_year):
    # Select data
    df =  airline_data[airline_data['Year']==int(entered_year)]
    # Compute delay averages
    avg_car = df.groupby(['Month','Reporting_Airline'])['CarrierDelay'].mean().reset_in
    avg_weather = df.groupby(['Month','Reporting_Airline'])['WeatherDelay'].mean().rese
    avg_NAS = df.groupby(['Month','Reporting_Airline'])['NASDelay'].mean().reset_index(
    avg_sec = df.groupby(['Month','Reporting_Airline'])['SecurityDelay'].mean().reset_i
    avg_late = df.groupby(['Month','Reporting_Airline'])['LateAircraftDelay'].mean().re
    return avg_car, avg_weather, avg_NAS, avg_sec, avg_late


# Build dash app layout
app.layout = html.Div(children=[ html.H1('Flight Delay Time Statistics',
                                style={'textAlign': 'center', 'color': '#503D36',
                                'font-size': 30}),
                                html.Div(["Input Year: ", dcc.Input(id='input-year', va
                                type='number', style={'height':'35px', 'font-size': 30}
                                style={'font-size': 30}),
                                html.Br(),
                                html.Br(),
                                html.Div([
                                        html.Div(dcc.Graph(id='carrier-plot')),
                                        html.Div(dcc.Graph(id='weather-plot'))
                                ], style={'display': 'flex'}),

                                html.Div([
                                        html.Div(dcc.Graph(id='nas-plot')),
                                        html.Div(dcc.Graph(id='security-plot'))
                                ], style={'display': 'flex'}),

                                html.Div(dcc.Graph(id='late-plot'), style={'width':'65%


                                ])

"""Callback Function

Function that returns fugures using the provided input year.

Arguments:

    entered_year: Input year provided by the user.

Returns:
```

```
    List of figures computed using the provided helper function `compute_info`.
"""
# Callback decorator
@app.callback( [
               Output(component_id='carrier-plot', component_property='figure'),
               Output(component_id='weather-plot', component_property='figure'),
               Output(component_id='nas-plot', component_property='figure'),
               Output(component_id='security-plot', component_property='figure'),
               Output(component_id='late-plot', component_property='figure')
               ],
               Input(component_id='input-year', component_property='value'))
# Computation to callback function and return graph
def get_graph(entered_year):

    # Compute required information for creating graph from the data
    avg_car, avg_weather, avg_NAS, avg_sec, avg_late = compute_info(airline_data, enter

    # Create graph
    carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airlin
    weather_fig = px.line(avg_weather, x='Month', y='WeatherDelay', color='Reporting_Ai
    nas_fig = px.line(avg_NAS, x='Month', y='NASDelay', color='Reporting_Airline', titl
    sec_fig = px.line(avg_sec, x='Month', y='SecurityDelay', color='Reporting_Airline',
    late_fig = px.line(avg_late, x='Month', y='LateAircraftDelay', color='Reporting_Air

    return[carrier_fig, weather_fig, nas_fig, sec_fig, late_fig]



# Run the app
if __name__ == '__main__':
    app.run_server(mode="inline", host="localhost", port=7645, debug=True)
```

In [ ]: