# Waffle Charts, Word Clouds, and Regression Plots

Estimated time needed: **40** minutes

## Objectives

After completing this lab you will be able to:

- Create Word cloud and Waffle charts
- Create regression plots with Seaborn library

## Table of Contents

# Import Libraries

```
#Import and setup matplotlib:
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches # needed for waffle Charts

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

#Import Primary Modules:
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

#install seaborn and wordcloud
!pip install seaborn wordcloud

#import seaborn
import seaborn as sns

#import wordcloud
import wordcloud

# check for latest version of Matplotlib and seaborn
print ('Matplotlib version: ', mpl.__version__) # >= 2.0.0
print('Seaborn version: ', sns.__version__)
print('WordCloud version: ', wordcloud.__version__)
```

```
Requirement already satisfied: seaborn in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (0.9.0)
Requirement already satisfied: wordcloud in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (1.9.3)
Requirement already satisfied: matplotlib>=1.4.3 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
seaborn) (3.5.3)
Requirement already satisfied: numpy>=1.9.3 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
seaborn) (1.21.6)
Requirement already satisfied: pandas>=0.15.2 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
seaborn) (1.3.5)
Requirement already satisfied: scipy>=0.14.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
seaborn) (1.7.3)
Requirement already satisfied: pillow in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
wordcloud) (9.5.0)
Requirement already satisfied: cycler>=0.10 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (23.1)
Requirement already satisfied: pyparsing>=2.2.1 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib>=1.4.3->seaborn) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
pandas>=0.15.2->seaborn) (2023.3)
Requirement already satisfied: typing-extensions in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
kiwisolver>=1.0.1->matplotlib>=1.4.3->seaborn) (4.5.0)
Requirement already satisfied: six>=1.5 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
python-dateutil>=2.7->matplotlib>=1.4.3->seaborn) (1.16.0)
Matplotlib version:  3.5.3
Seaborn version:  0.9.0
WordCloud version:  1.9.3
```

# Fetching Data

Toolkits: The course heavily relies on *pandas* and *Numpy* for data wrangling, analysis, and visualization. The primary plotting library we will explore in the course is Matplotlib.

Dataset: Immigration to Canada from 1980 to 2013 - International migration flows to and from selected countries - The 2015 revision from United Nation's website

The dataset contains annual data on the flows of international migrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. In this lab, we will focus on the Canadian Immigration data and use the *already cleaned dataset*. You can refer to the lab on data pre-processing wherein this dataset is cleaned for a quick refresh your Panads skill Data pre-processing with Pandas

Download the Canadian Immigration dataset and read it into a *pandas* dataframe.

```
df_can = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-
SkillsNetwork/Data%20Files/Canada.csv')

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

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

Let's take a look at the first five items in our dataset

```
df_can.head()
```

```
          Country Continent           Region             DevName  1980
1981  \
0     Afghanistan      Asia    Southern Asia  Developing regions    16
39
1         Albania    Europe  Southern Europe   Developed regions     1
0
2         Algeria    Africa  Northern Africa  Developing regions    80
67
3  American Samoa   Oceania        Polynesia  Developing regions     0
1
4         Andorra    Europe  Southern Europe   Developed regions     0
0

   1982  1983  1984  1985  ...  2005  2006  2007  2008  2009  2010
2011  \
0    39    47    71   340  ...  3436  3009  2652  2111  1746  1758
2203
1     0     0     0     0  ...  1223   856   702   560   716   561
539
2    71    69    63    44  ...  3626  4807  3623  4005  5393  4752
4325
```

```
3     0     0     0     0  ...     0     1     0     0     0     0
0
4     0     0     0     0  ...     0     1     1     0     0     0
0

   2012  2013  Total
0  2635  2004  58639
1   620   603  15699
2  3774  4331  69439
3     0     0      6
4     1     1     15

[5 rows x 39 columns]
```

Let's find out how many entries there are in our dataset

```
# print the dimensions of the dataframe
print(df_can.shape)

(195, 39)

#set Country as index
df_can.set_index('Country', inplace=True)
df_can.head()

               Continent           Region             DevName  1980
1981  \
Country

Afghanistan         Asia    Southern Asia  Developing regions    16
39
Albania           Europe  Southern Europe   Developed regions     1
0
Algeria           Africa  Northern Africa  Developing regions    80
67
American Samoa   Oceania        Polynesia  Developing regions     0
1
Andorra           Europe  Southern Europe   Developed regions     0
0

                1982  1983  1984  1985  1986  ...  2005  2006  2007
2008  \
Country                                       ...

Afghanistan       39    47    71   340   496  ...  3436  3009  2652
2111
Albania            0     0     0     0     1  ...  1223   856   702
560
Algeria           71    69    63    44    69  ...  3626  4807  3623
4005
```

```
American Samoa     0     0     0     0     0  ...      0     1     0
0
Andorra            0     0     0     0     2  ...      0     1     1
0

                2009  2010  2011  2012  2013  Total
Country
Afghanistan     1746  1758  2203  2635  2004  58639
Albania          716   561   539   620   603  15699
Algeria         5393  4752  4325  3774  4331  69439
American Samoa     0     0     0     0     0      6
Andorra            0     0     0     1     1     15

[5 rows x 38 columns]
```

# Waffle Charts

A `waffle chart` is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an Excel dashboard.

Let's revisit the previous case study about Denmark, Norway, and Sweden.

```
# 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

        Continent           Region             DevName  1980  1981
1982  1983  \
Country

Denmark    Europe  Northern Europe  Developed regions   272   293
299   106
Norway     Europe  Northern Europe  Developed regions   116    77
106    51
Sweden     Europe  Northern Europe  Developed regions   281   308
222   176

         1984  1985  1986  ...  2005  2006  2007  2008  2009  2010
2011  \
Country                    ...

Denmark    93    73    93  ...    62   101    97   108    81    92
93
Norway     31    54    56  ...    57    53    73    66    75    46
49
```

```
Sweden     128   158   187  ...   205   139   193   165   167   159
134

         2012  2013  Total
Country
Denmark    94    81   3901
Norway     53    59   2327
Sweden    140   140   5866

[3 rows x 38 columns]
```

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.

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

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

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

         Category Proportion
Country
Denmark             0.322557
Norway              0.192409
Sweden              0.485034
```

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

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

total_num_tiles = width * height # total number of tiles

print(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 determe it respective number of tiles

```
# 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})
```

```
         Number of tiles
Country
Denmark              129
Norway                77
Sweden               194
```

Based on the calculated proportions, Denmark will occupy 129 tiles of the `waffle` chart, Norway will occupy 77 tiles, and Sweden will occupy 194 tiles.

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

```
# initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width), 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 corresponding allocated tiles...
        if tile_index > sum(tiles_per_category[0:category_index]):
            # ...proceed to the next category
            category_index += 1

        # set the class value to an integer, which increases with
class
        waffle_chart[row, col] = category_index

print ('Waffle chart populated!')
```

```
Waffle chart populated!
```

Let's take a peek at how the matrix looks like.

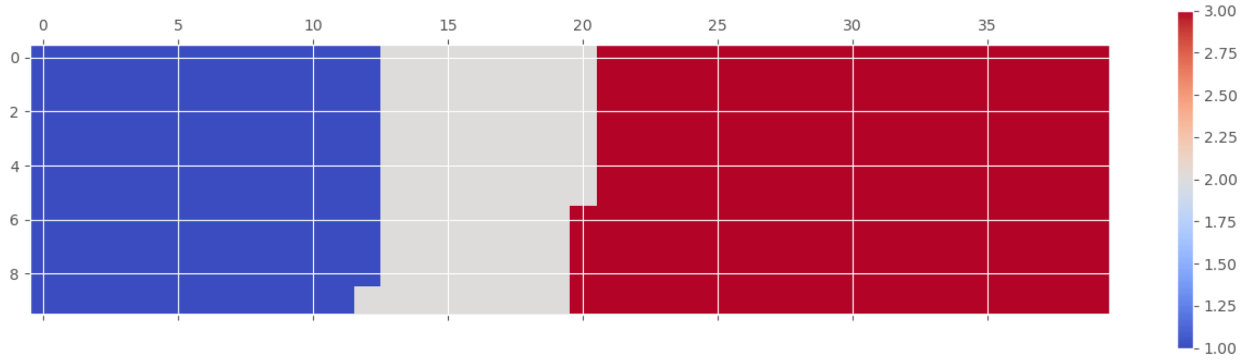
```
waffle_chart
```

```
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, 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=uint64)
```

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.

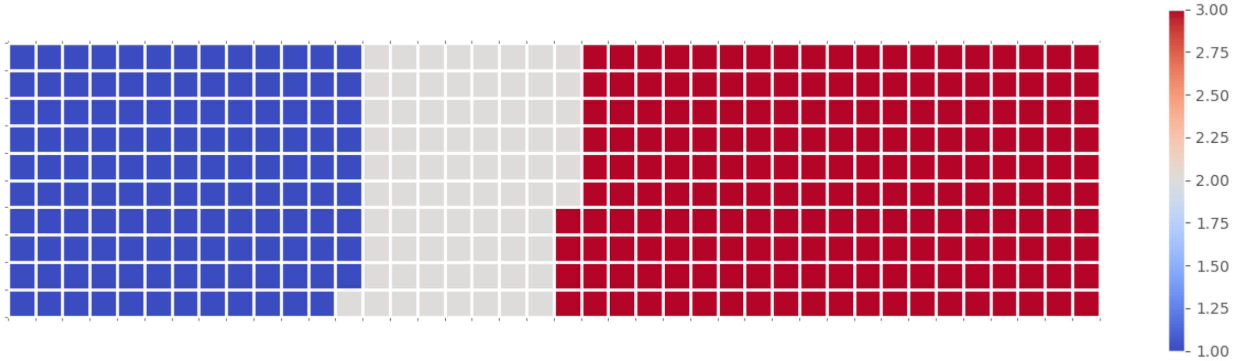
**Step 5.** Map the `waffle` chart matrix into a visual.

```
# 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 640x480 with 0 Axes>
```



**Step 6.** Prettify the chart.

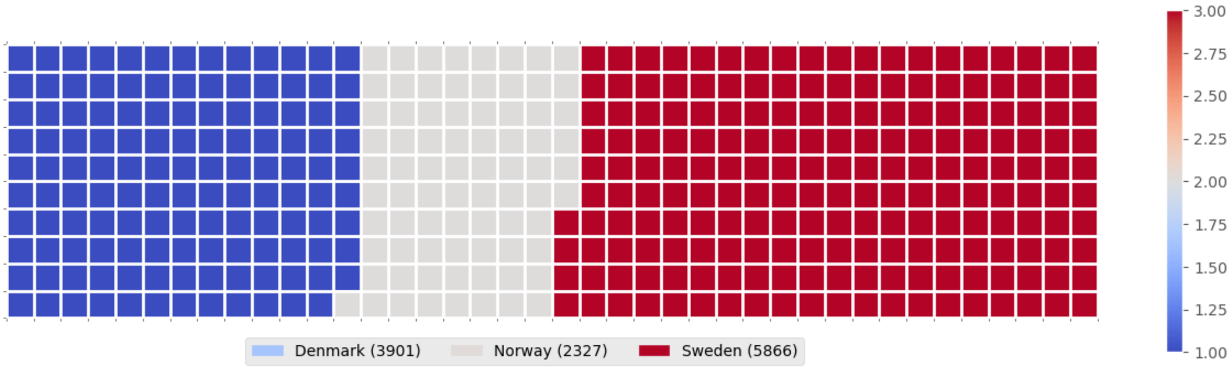
```
# instantiate a new figure object
fig = plt.figure()

# use matshow to display the waffle chart
colormap = plt.cm.coolwarm
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# get the axis
ax = plt.gca()

# set minor ticks
ax.set_xticks(np.arange(-.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-.5, (height), 1), minor=True)

# add gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-', linewidth=2)

plt.xticks([])
plt.yticks([])
plt.show()

<Figure size 640x480 with 0 Axes>
```



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

```
# instantiate a new figure object
fig = plt.figure()

# use matshow to display the waffle chart
colormap = plt.cm.coolwarm
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# get the axis
ax = plt.gca()

# set minor ticks
ax.set_xticks(np.arange(-.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-.5, (height), 1), minor=True)

# add gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-', linewidth=2)

plt.xticks([])
plt.yticks([])

# compute cumulative sum of individual categories to match color
schemes between chart and legend
values_cumsum = np.cumsum(df_dsn['Total'])
total_values = values_cumsum[len(values_cumsum) - 1]

# create legend
legend_handles = []
for i, category in enumerate(df_dsn.index.values):
    label_str = category + ' (' + str(df_dsn['Total'][i]) + ')'
    color_val = colormap(float(values_cumsum[i])/total_values)
    legend_handles.append(mpatches.Patch(color=color_val,
label=label_str))

# add legend to chart
plt.legend(handles=legend_handles,
           loc='lower center',
           ncol=len(df_dsn.index.values),
           bbox_to_anchor=(0., -0.2, 0.95, .1)
          )
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



And there you go! What a good looking *delicious* `waffle` chart, don't you think?

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.

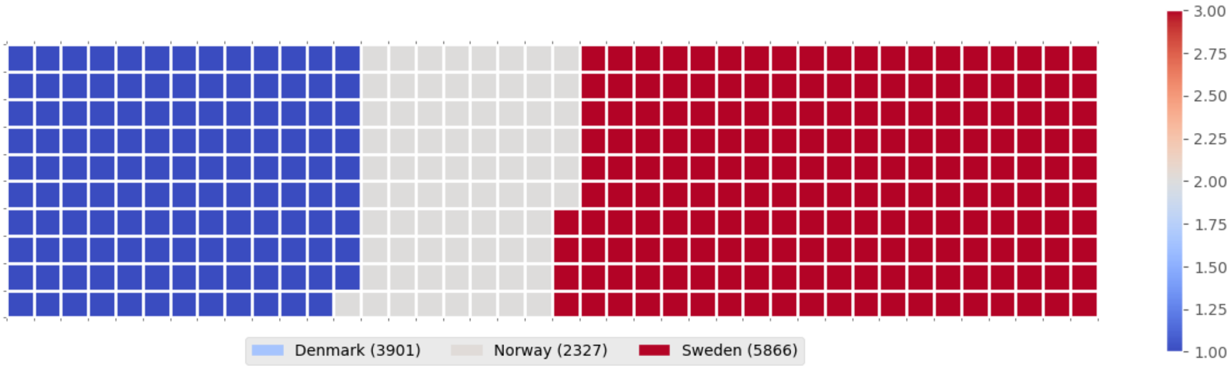
```
def create_waffle_chart(categories, values, height, width, colormap,
value_sign=''):

    # 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 category_proportions]

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

    # 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 chart and legend
    values_cumsum = np.cumsum(values)
    total_values = values_cumsum[len(values_cumsum) - 1]

    # create legend
    legend_handles = []
    for i, category in enumerate(categories):
        if value_sign == '%':
            label_str = category + ' (' + str(values[i]) + value_sign
+ ')'
        else:
            label_str = category + ' (' + value_sign + str(values[i])
+ ')'

        color_val = colormap(float(values_cumsum[i])/total_values)
        legend_handles.append(mpatches.Patch(color=color_val,
label=label_str))

    # add legend to chart
    plt.legend(
        handles=legend_handles,
        loc='lower center',
        ncol=len(categories),
        bbox_to_anchor=(0., -0.2, 0.95, .1)
    )
    plt.show()
```

Now to create a `waffle` chart, all we have to do is call the function `create_waffle_chart`. Let's define the input parameters:

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

categories = df_dsn.index.values # categories
values = df_dsn['Total'] # correponding values of categories

colormap = plt.cm.coolwarm # color map class
```

And now let's call our function to create a `waffle` chart.

```
create_waffle_chart(categories, values, height, width, colormap)

Total number of tiles is 400
Denmark: 129
Norway: 77
Sweden: 194

<Figure size 640x480 with 0 Axes>
```

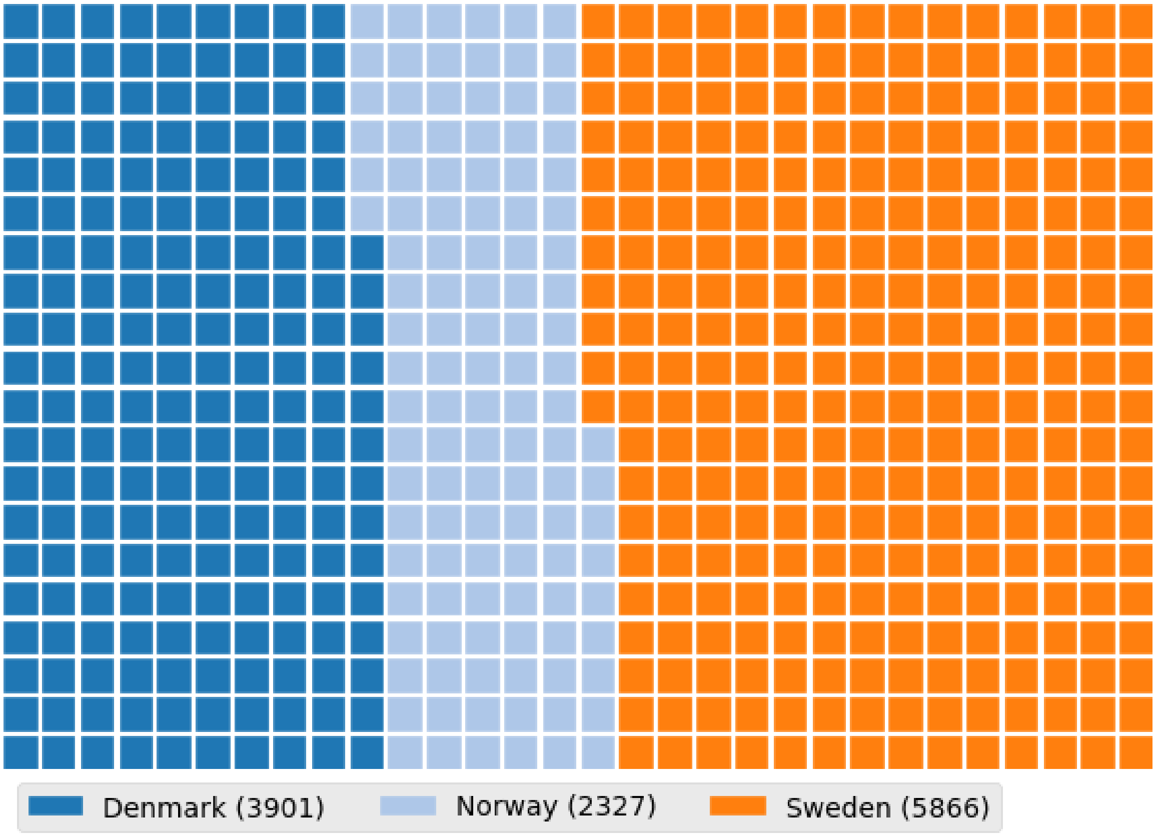


There seems to be a new Python package for generating `waffle charts` called PyWaffle, Let's create the same waffle chart with **pywaffle** now

```
#install pywaffle
!pip install pywaffle

Requirement already satisfied: pywaffle in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (1.1.1)
Requirement already satisfied: fontawesomefree in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
pywaffle) (6.5.1)
Requirement already satisfied: matplotlib in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
pywaffle) (3.5.3)
Requirement already satisfied: cycler>=0.10 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (1.4.4)
Requirement already satisfied: numpy>=1.17 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (1.21.6)
Requirement already satisfied: packaging>=20.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (23.1)
Requirement already satisfied: pillow>=6.2.0 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (9.5.0)
Requirement already satisfied: pyparsing>=2.2.1 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
matplotlib->pywaffle) (2.8.2)
```

```
Requirement already satisfied: typing-extensions in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
kiwisolver>=1.0.1->matplotlib->pywaffle) (4.5.0)
Requirement already satisfied: six>=1.5 in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from
python-dateutil>=2.7->matplotlib->pywaffle) (1.16.0)
```

```
#import Waffle from pywaffle
from pywaffle import Waffle

#Set up the Waffle chart figure

fig = plt.figure(FigureClass = Waffle,
                 rows = 20, columns = 30, #pass the number of rows and
columns for the waffle
                 values = df_dsn['Total'], #pass the data to be used
for display
                 cmap_name = 'tab20', #color scheme
                 legend = {'labels': [f"{k} ({v})" for k, v in
zip(df_dsn.index.values,df_dsn.Total)],
                            'loc': 'lower left', 'bbox_to_anchor':(0,-
0.1),'ncol': 3}
                 #notice the use of list comprehension for creating
labels
                 #from index and total of the dataset
                )

#Display the waffle chart
plt.show()
```



**Question:** Create a Waffle chart to dispaly the proportion of China and India total immigrant contribution.

```
# let's create a new dataframe for these three countries
df_CI = df_can.loc[['China', 'India'], :]

# let's take a look at our dataframe
df_CI

        Continent          Region                DevName  1980  1981  1982
1983  \
Country

China        Asia    Eastern Asia  Developing regions  5123  6682  3308
1863
India        Asia   Southern Asia  Developing regions  8880  8670  8147
7338

         1984  1985  1986  ...   2005   2006   2007   2008   2009
2010  \
Country                    ...

China    1527  1816  1960  ...  42584  33518  27642  30037  29622
30391
```

```
India    5704  4211  7150  ...  36210  33848  28742  28261  29456
34235

          2011   2012   2013   Total
Country
China    28502  33024  34129  659962
India    27509  30933  33087  691904

[2 rows x 38 columns]
```

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

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

```
         Category Proportion
Country
China               0.488186
India               0.511814
```

```python
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.
```

```python
# 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})
```

```
         Number of tiles
Country
China                195
India                205
```

```python
# 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 corresponding allocated tiles...
        if tile_index > sum(tiles_per_category[0:category_index]):
            # ...proceed to the next category
            category_index += 1

        # set the class value to an integer, which increases with
class
        waffle_chart[row, col] = category_index

print ('Waffle chart populated!')
```

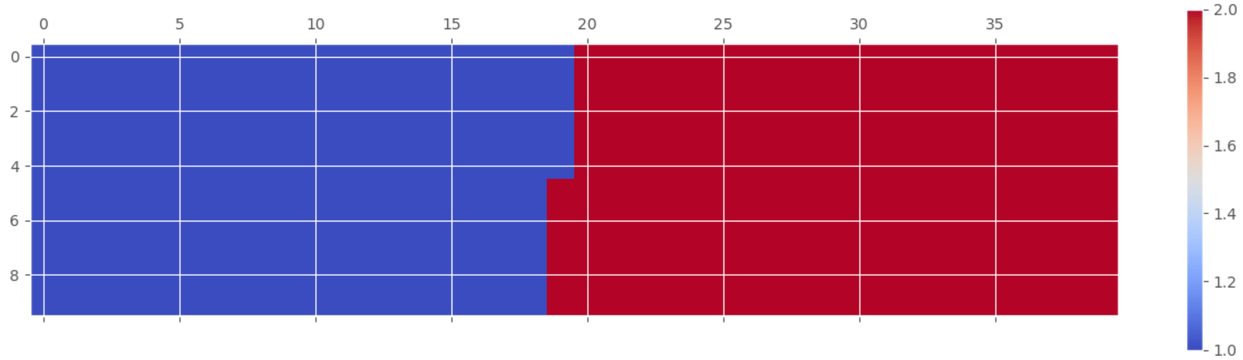
```
Waffle chart populated!
```

```
waffle_chart
```

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]],
      dtype=uint64)
```

```
# 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()
```

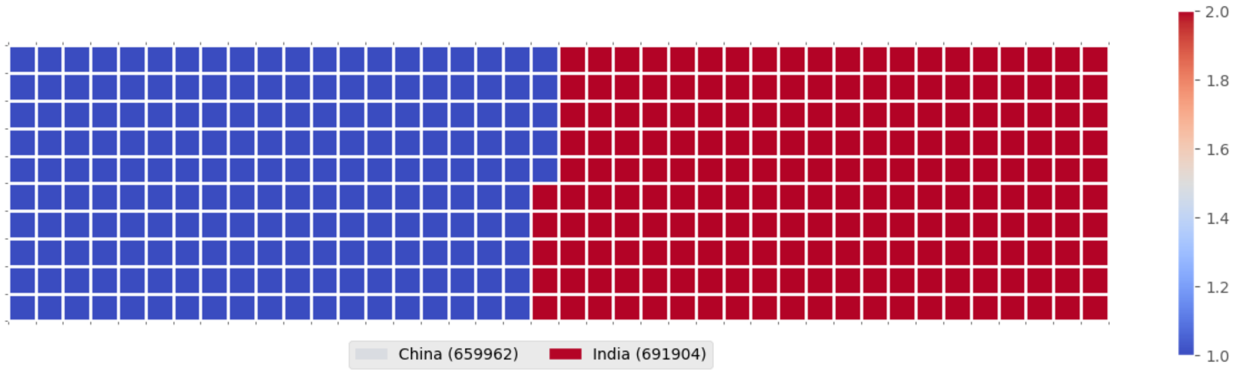
```
<Figure size 640x480 with 0 Axes>
```



```
# 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 and legend
values_cumsum = np.cumsum(df_CI['Total'])
total_values = values_cumsum[len(values_cumsum) - 1]

# create legend
```

```
legend_handles = []
for i, category in enumerate(df_CI.index.values):
    label_str = category + ' (' + str(df_CI['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 640x480 with 0 Axes>
```



# Word Clouds

`Word` clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

Luckily, a Python package already exists in Python for generating `word` clouds. The package, called `word_cloud` was developed by **Andreas Mueller**. You can learn more about the package by following this link.

Let's use this package to learn how to generate a word cloud for a given text document.

First, let's install the package.

```
#import package and its set of stopwords
from wordcloud import WordCloud, STOPWORDS

print ('Wordcloud imported!')
```

```
Wordcloud imported!
```

`Word` clouds are commonly used to perform high-level analysis and visualization of text data. Accordinly, let's digress from the immigration dataset and work with an example that involves analyzing text data. Let's try to analyze a short novel written by **Lewis Carroll** titled *Alice's Adventures in Wonderland*. Let's go ahead and download a *.txt* file of the novel.

```
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-storage.appdomain.cloud/
IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/
alice_novel.txt').read().decode("utf-8")
print('Done')
```

```
Done
```

Next, let's use the stopwords that we imported from `word_cloud`. We use the function *set* to remove any redundant stopwords.

```
stopwords = set(STOPWORDS)
```

Create a word cloud object and generate a word cloud. For simplicity, let's generate a word cloud using only the first 2000 words in the novel.

```
#if you get attribute error while generating worldcloud, upgrade
Pillow and numpy using below code
%pip install --upgrade Pillow
%pip install --upgrade numpy

#After upgrading, Restart the kernel, delete the above upgrading code
cell and rerun the codes from the start.
```

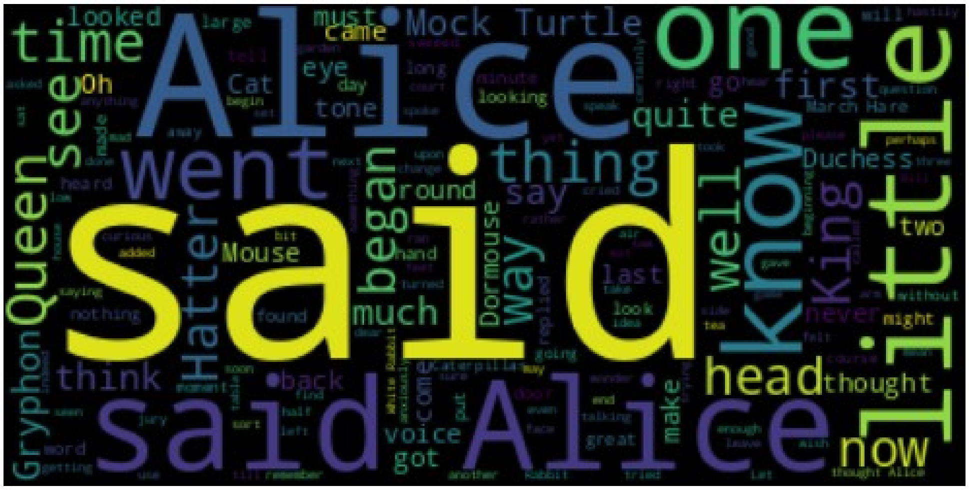
```
Requirement already satisfied: Pillow in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (9.5.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: numpy in
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages
(1.21.6)
Note: you may need to restart the kernel to use updated packages.
```

```
# instantiate a word cloud object
alice_wc = WordCloud()

# generate the word cloud
alice_wc.generate(alice_novel)
```

```
<wordcloud.wordcloud.WordCloud at 0x7f8c52079f50>
```

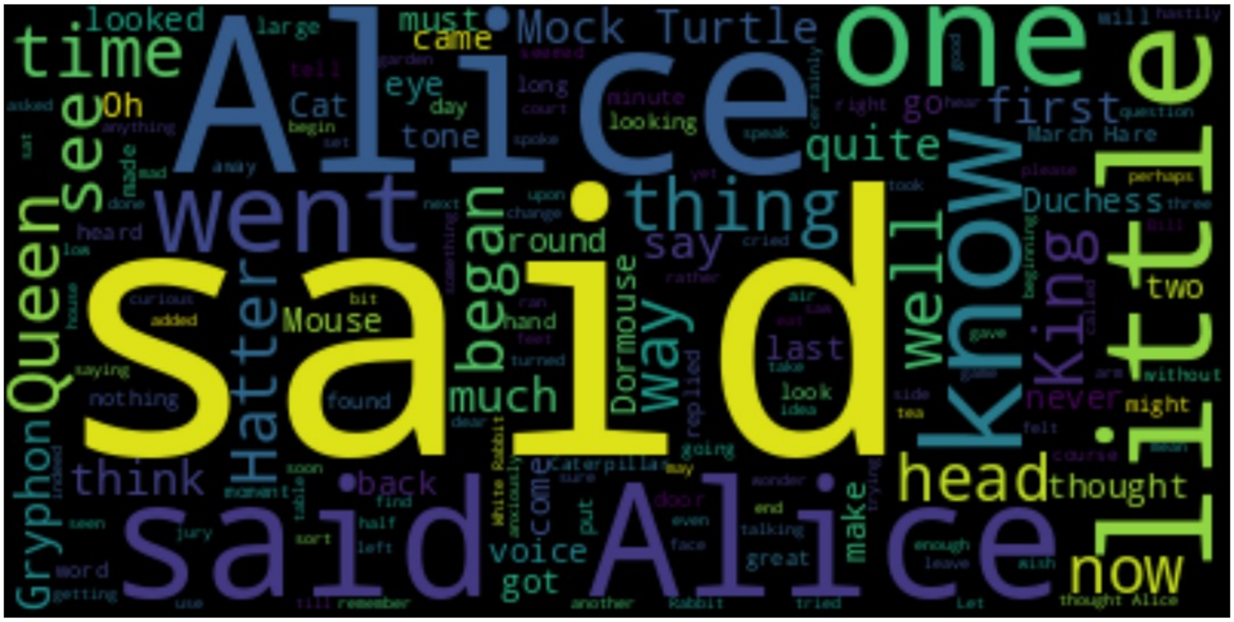
Awesome! Now that the `word` cloud is created, let's visualize it.

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



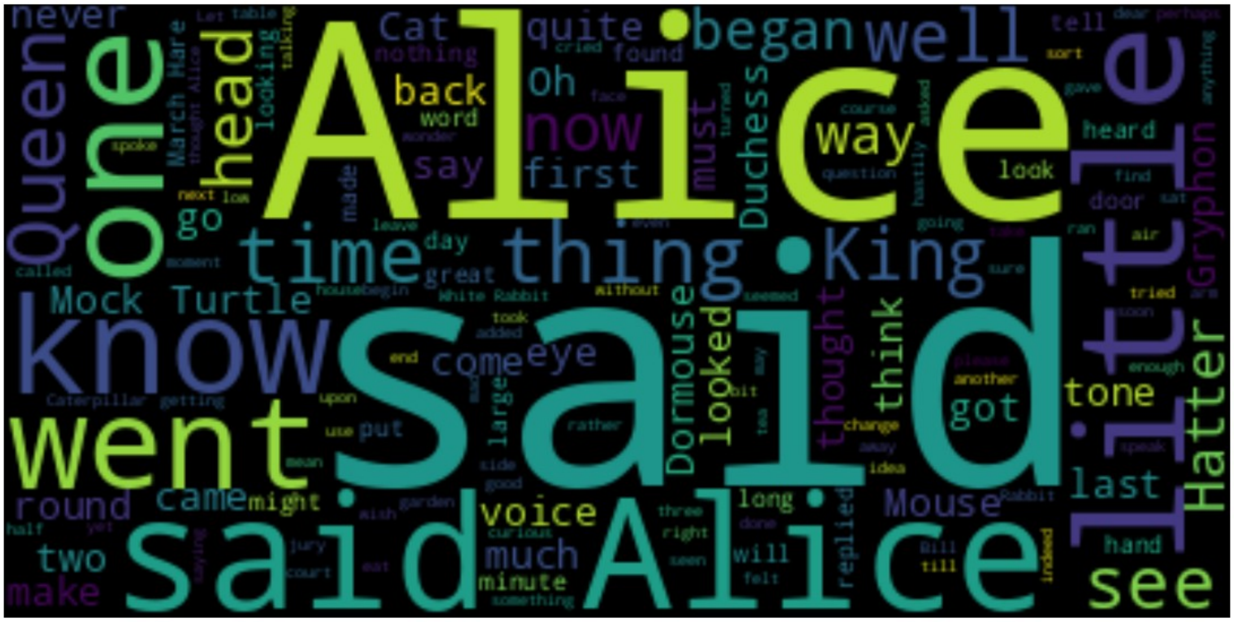
Interesting! So in the first 2000 words in the novel, the most common words are **Alice**, **said**, **little**, **Queen**, and so on. Let's resize the cloud so that we can see the less frequent words a little better.

```
fig = plt.figure(figsize=(14, 18))

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



Much better! However, **said** isn't really an informative word. So let's add it to our stopwords and re-generate the cloud.

```
stopwords.add('said') # add the words said to stopwords

# re-generate the word cloud
alice_wc.generate(alice_novel)

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

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



Excellent! This looks really interesting! Another cool thing you can implement with the `word_cloud` package is superimposing the words onto a mask of any shape. Let's use a mask of Alice and her rabbit. We already created the mask for you, so let's go ahead and download it and call it *alice_mask.png*.

```
#save mask to alice_mask
alice_mask = np.array(Image.open(urllib.request.urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/labs/Module%204/images/alice_mask.png')))
```

Let's take a look at how the mask looks like.

```
fig = plt.figure(figsize=(14, 18))

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

Shaping the `word` cloud according to the mask is straightforward using `word_cloud` package. For simplicity, we will continue using the first 2000 words in the novel.

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

# generate the word cloud
alice_wc.generate(alice_novel)

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

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

getting
looked
way
round
tell
King
beginning
little
thing
Hatter
Mouse
Caterpillar
Cat
must
never
hand
Gryphon
thought
know
well
come
tone
great
see
went
Alice
head
Dormouse
Duchess
began
quite
think
Mock Turtle
one
time
Queen
now
back
two
voice
found
first
much
make
day
minute
last
go
say
got
door
March Hare
White Rabbit
Rabbit
Oh
large
came
heard
might
looking
replied
long
made
spoke
three
half
tail
use

Really impressive!

Unfortunately, our immigration data does not have any text data, but where there is a will there is a way. Let's generate sample text data from our immigration dataset, say text data of 90 words.

Let's recall how our data looks like.

```
df_can.head()

               Continent           Region             DevName  1980
1981  \
Country

Afghanistan         Asia    Southern Asia  Developing regions    16
39
Albania           Europe  Southern Europe   Developed regions     1
0
Algeria           Africa  Northern Africa  Developing regions    80
67
American Samoa   Oceania        Polynesia  Developing regions     0
1
Andorra           Europe  Southern Europe   Developed regions     0
0

                1982  1983  1984  1985  1986  ...  2005  2006  2007
2008  \
Country                                       ...

Afghanistan       39    47    71   340   496  ...  3436  3009  2652
2111
Albania            0     0     0     0     1  ...  1223   856   702
560
Algeria           71    69    63    44    69  ...  3626  4807  3623
4005
American Samoa     0     0     0     0     0  ...     0     1     0
0
Andorra            0     0     0     0     2  ...     0     1     1
0

                2009  2010  2011  2012  2013  Total
Country
Afghanistan     1746  1758  2203  2635  2004  58639
Albania          716   561   539   620   603  15699
Algeria         5393  4752  4325  3774  4331  69439
American Samoa     0     0     0     0     0      6
Andorra            0     0     0     1     1     15

[5 rows x 38 columns]
```

And what was the total immigration from 1980 to 2013?

```
total_immigration = df_can['Total'].sum()
total_immigration

6409153
```

Using countries with single-word names, let's duplicate each country's name based on how much they contribute to the total immigration.

```
max_words = 90
word_string = ''
for country in df_can.index.values:
     # check if country's name is a single-word name
    if country.count(" ") == 0:
        repeat_num_times = int(df_can.loc[country, 'Total'] / 
total_immigration * max_words)
        word_string = word_string + ((country + ' ') * 
repeat_num_times)

# display the generated text
word_string
```

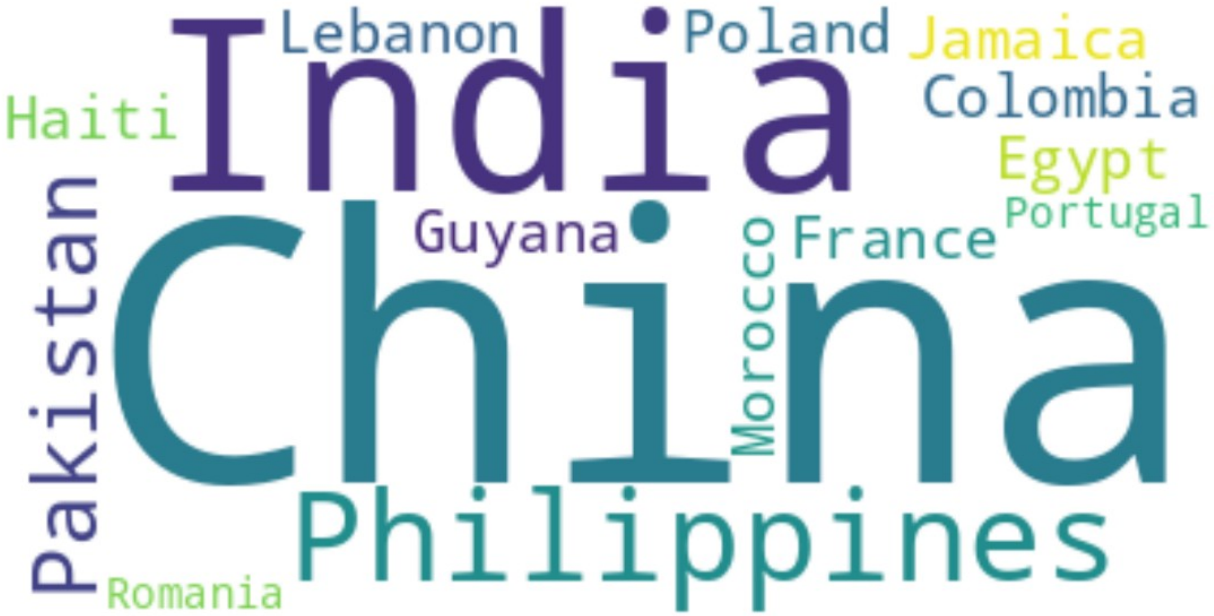
```
'China China China China China China China China China Colombia Egypt 
France Guyana Haiti India India India India India India India India 
India Jamaica Lebanon Morocco Pakistan Pakistan Pakistan Philippines 
Philippines Philippines Philippines Philippines Philippines 
Philippines Poland Portugal Romania '
```

We are not dealing with any stopwords here, so there is no need to pass them when creating the word cloud.

```
# create the word cloud
wordcloud = WordCloud(background_color='white').generate(word_string)

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

```
Word cloud created!
```

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

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



According to the above word cloud, it looks like the majority of the people who immigrated came from one of 15 countries that are displayed by the word cloud. One cool visual that you could build, is perhaps using the map of Canada and a mask and superimposing the word cloud on top of the map of Canada. That would be an interesting visual to build!

# Plotting with Seaborn

> Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics. You can learn more about *seaborn* by following this link and more about *seaborn* regression plots by following this link.

In lab *Pie Charts, Box Plots, Scatter Plots, and Bubble Plots*, we learned how to create a scatter plot and then fit a regression line. It took ~20 lines of code to create the scatter plot along with the regression fit. In this final section, we will explore *seaborn* and see how efficient it is to create regression lines and fits using this library!

## Categorical Plots

In our data 'df_can', let's find out how many continents are mentioned

```
df_can['Continent'].unique()
```

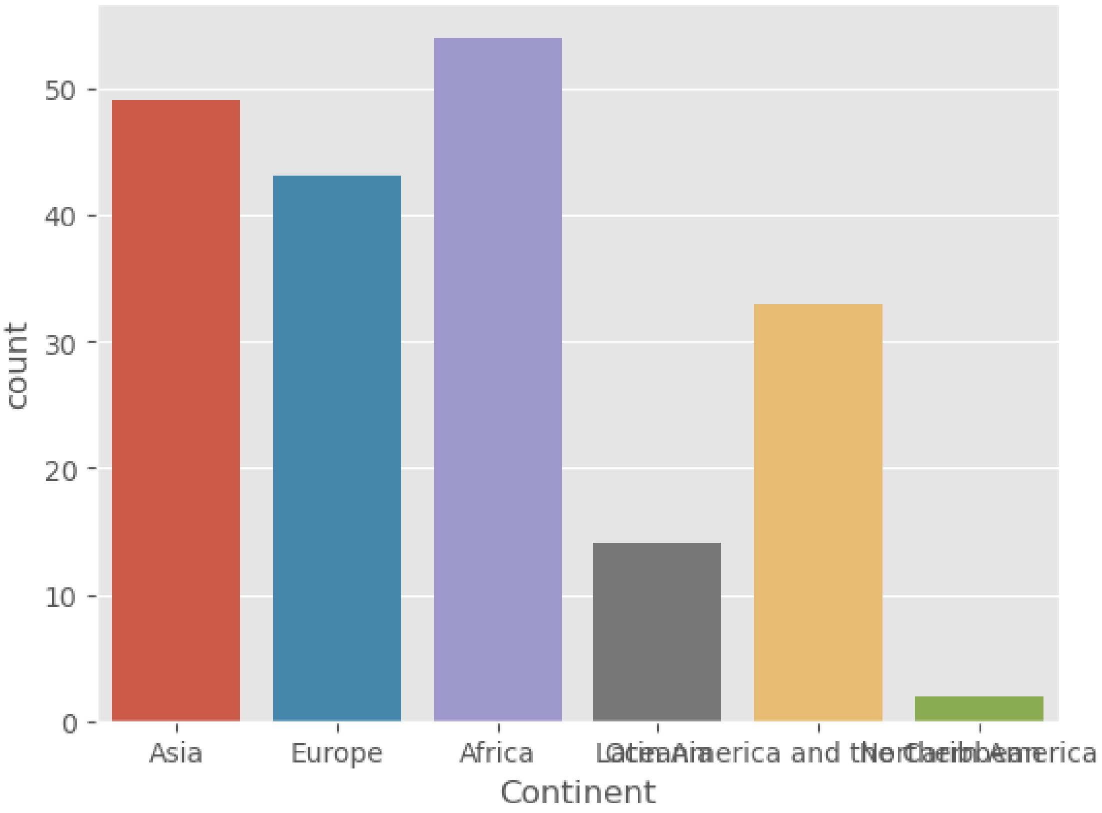
```
array(['Asia', 'Europe', 'Africa', 'Oceania',
       'Latin America and the Caribbean', 'Northern America'],
      dtype=object)
```

## countplot

**A count plot can be thought of as a histogram across a categorical, instead of quantitative, variable.** Let's find the count of Continents in the data 'df_can' using countplot on 'Continent'

```
sns.countplot(x='Continent', data=df_can)
```

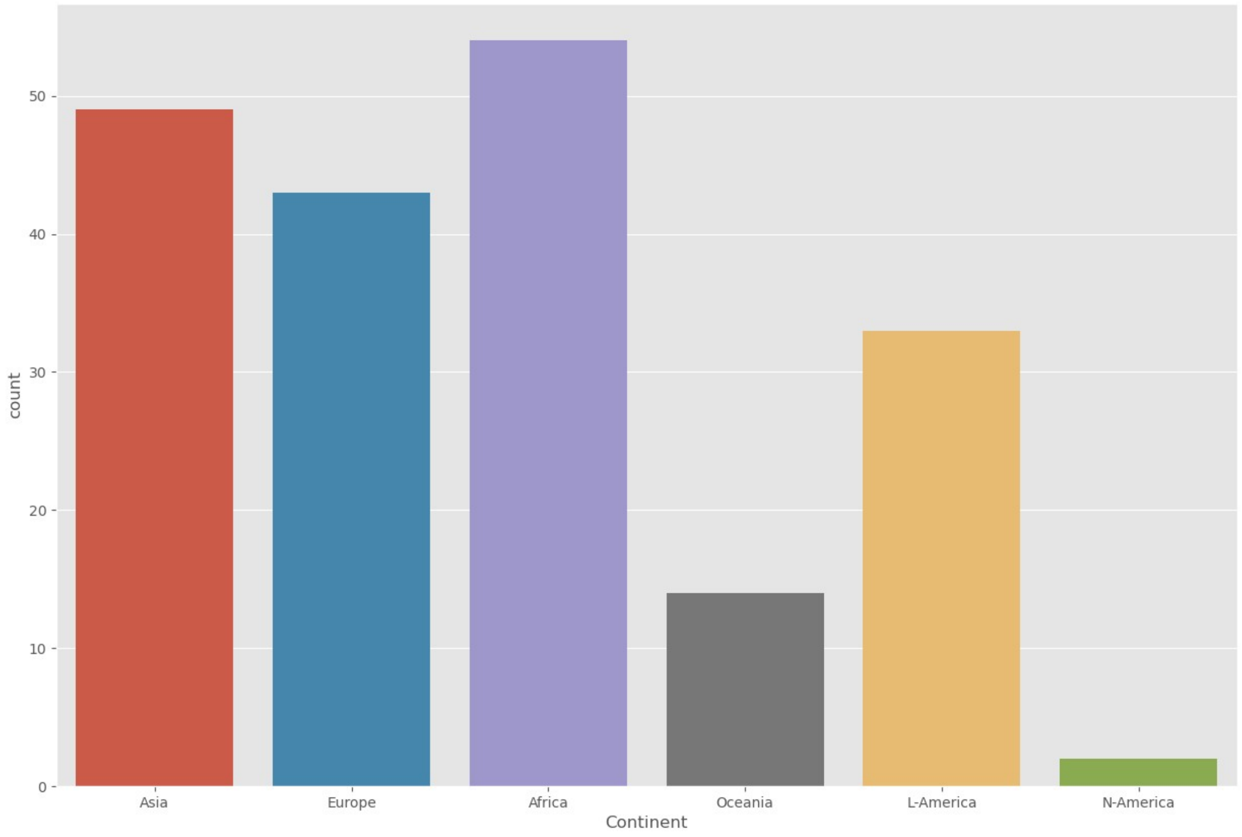
```
<AxesSubplot:xlabel='Continent', ylabel='count'>
```



The labels on the x-axis doesnot look as expected.Let's try to replace the 'Latin America and the Caribbean' with and "L-America", 'Northern America' with "N-America", and change the figure size and then display the plot again

```
df_can1 = df_can.replace('Latin America and the Caribbean', 'L-America')
df_can1 = df_can1.replace('Northern America', 'N-America')

plt.figure(figsize=(15, 10))
sns.countplot(x='Continent', data=df_can1)
```

```
<AxesSubplot:xlabel='Continent', ylabel='count'>
```



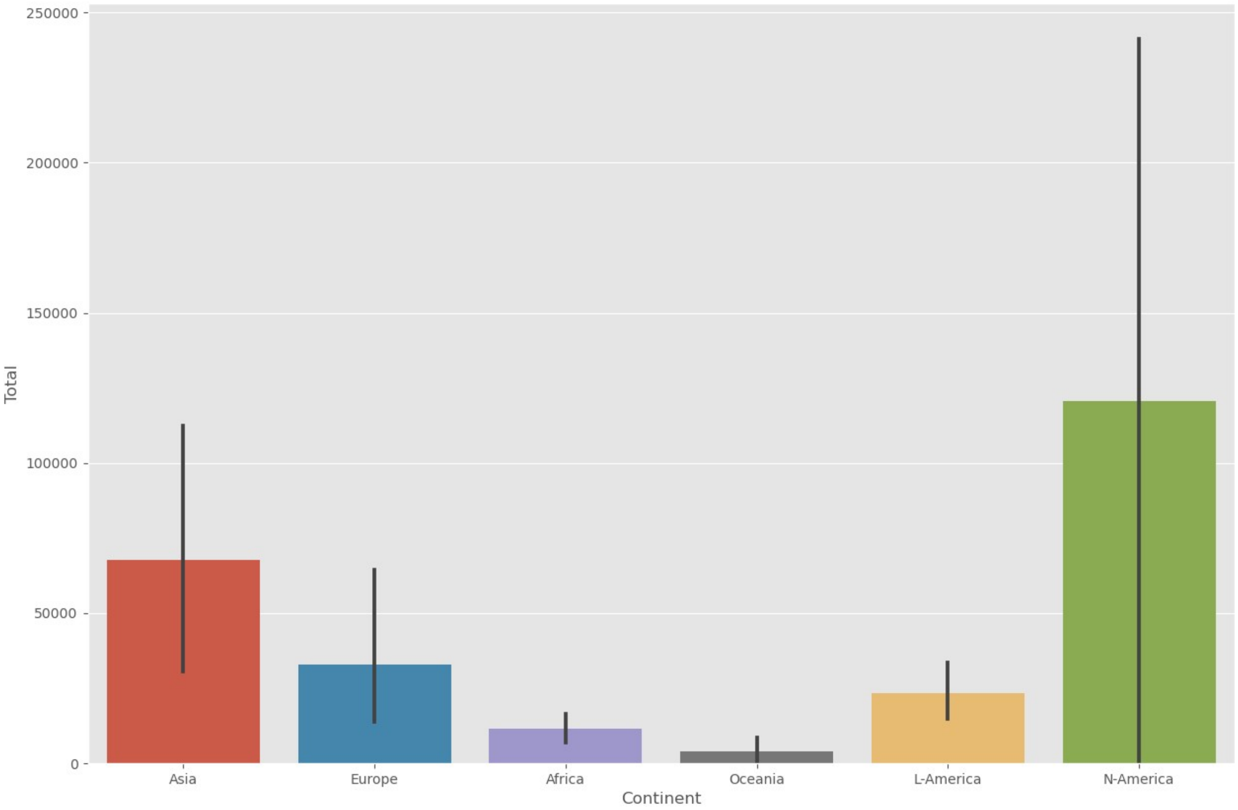
Much better!

## Barplot

**This plot will perform the Groupby on a categorical varaible and plot aggregated values, with confidence intervals.** Let's plot the total immigrants Continent-wise

```
plt.figure(figsize=(15, 10))
sns.barplot(x='Continent', y='Total', data=df_can1)
```

```
<AxesSubplot:xlabel='Continent', ylabel='Total'>
```



You can verify the values by performing the groupby on the Total and Continent for mean()

```
df_Can2=df_can1.groupby('Continent')['Total'].mean()
df_Can2

Continent
Africa         11462.000000
Asia           67710.081633
Europe         32812.720930
L-America      23186.303030
N-America     120571.000000
Oceania         3941.000000
Name: Total, dtype: float64
```

Create a new dataframe that stores that total number of landed immigrants to Canada per year from 1980 to 2013.

# Regression Plot

With *seaborn*, generating a regression plot is as simple as calling the **regplot** function.

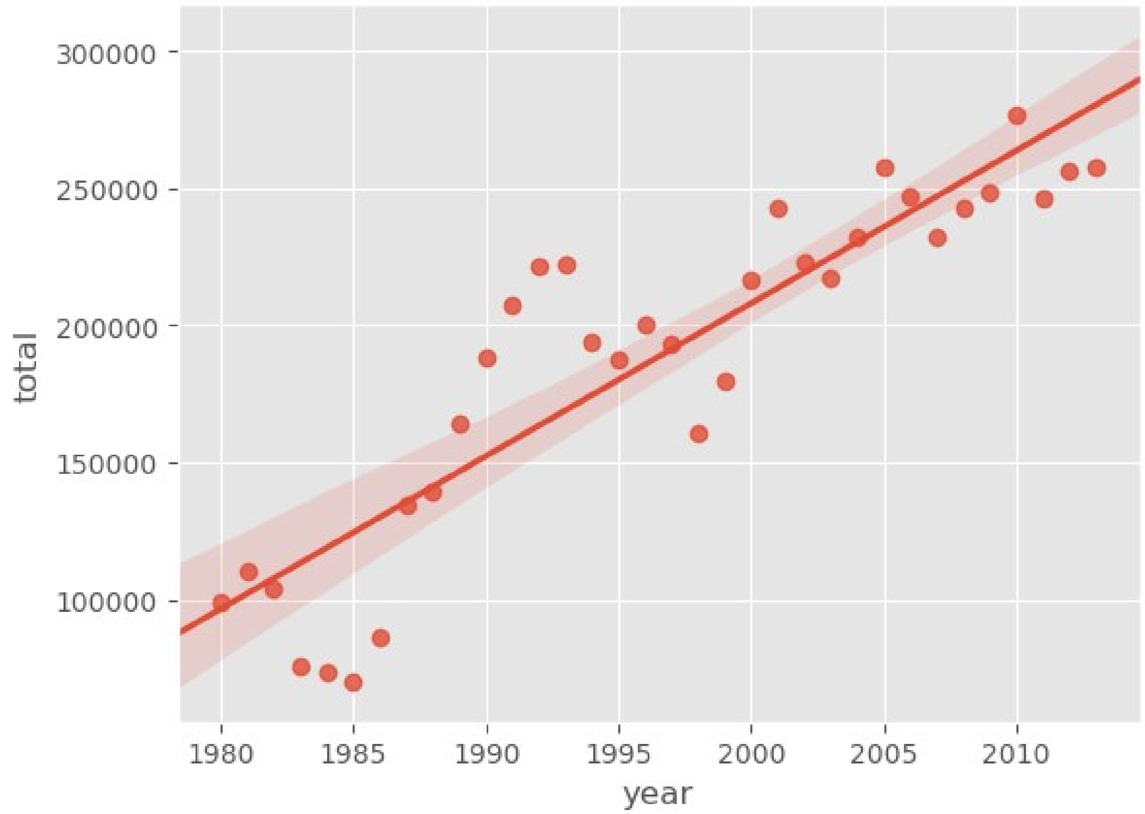
```
years = list(map(str, range(1980, 2014)))
# we can use the sum() method to get the total population per year
```

```
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()
```

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

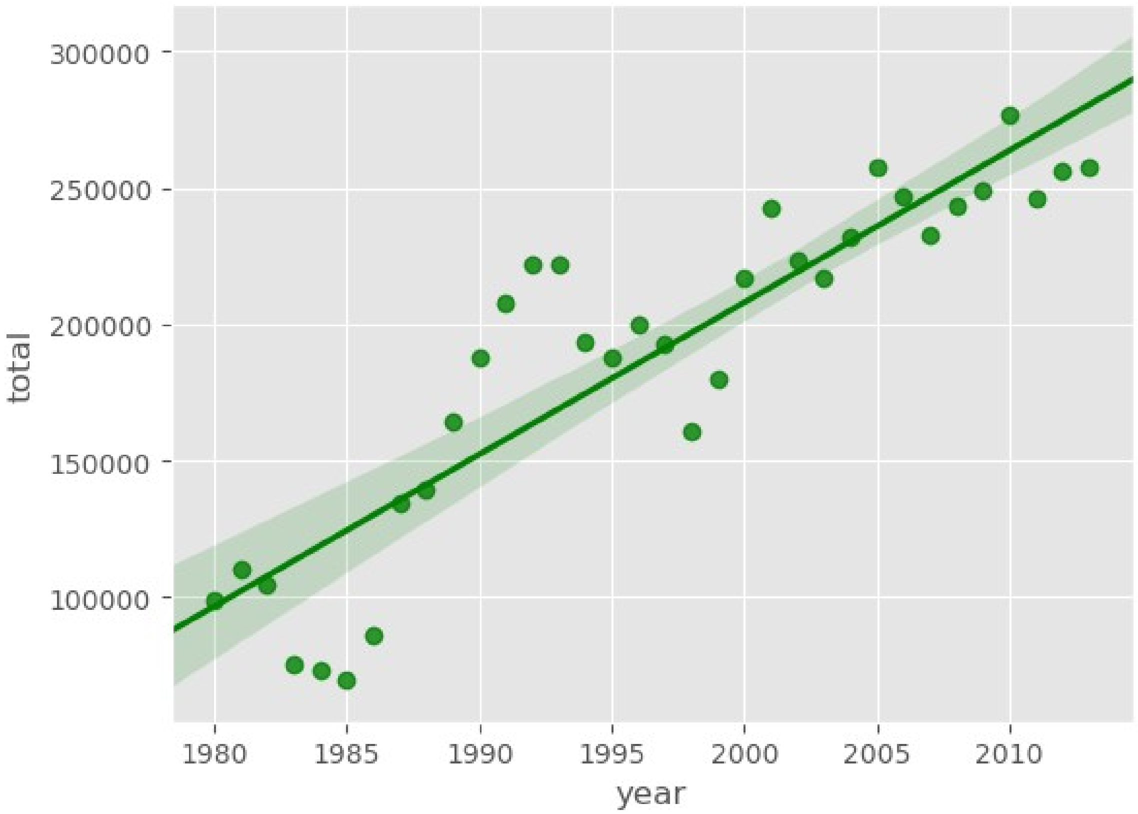
```
#seaborn is already imported at the start of this lab
sns.regplot(x='year', y='total', data=df_tot)
```

```
<AxesSubplot:xlabel='year', ylabel='total'>
```



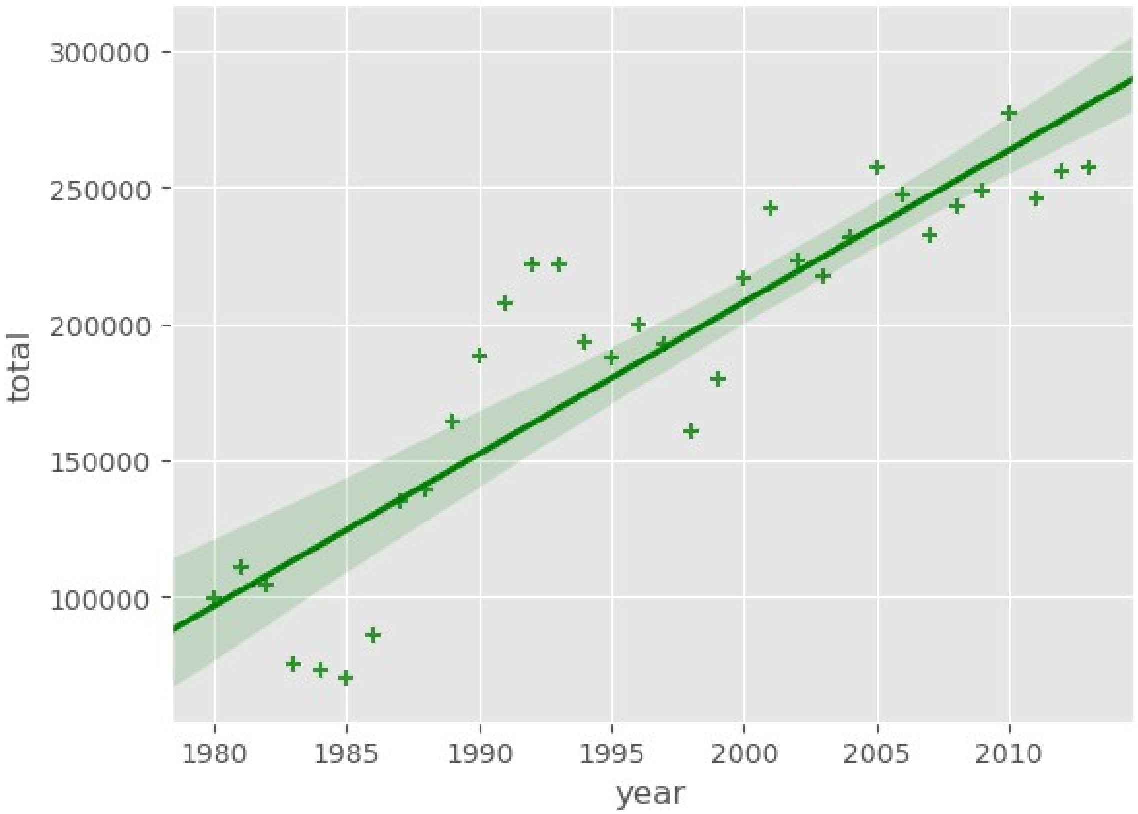
This is not magic; it is *seaborn*! You can also customize the color of the scatter plot and regression line. Let's change the color to green.

```
sns.regplot(x='year', y='total', data=df_tot, color='green')
plt.show()
```



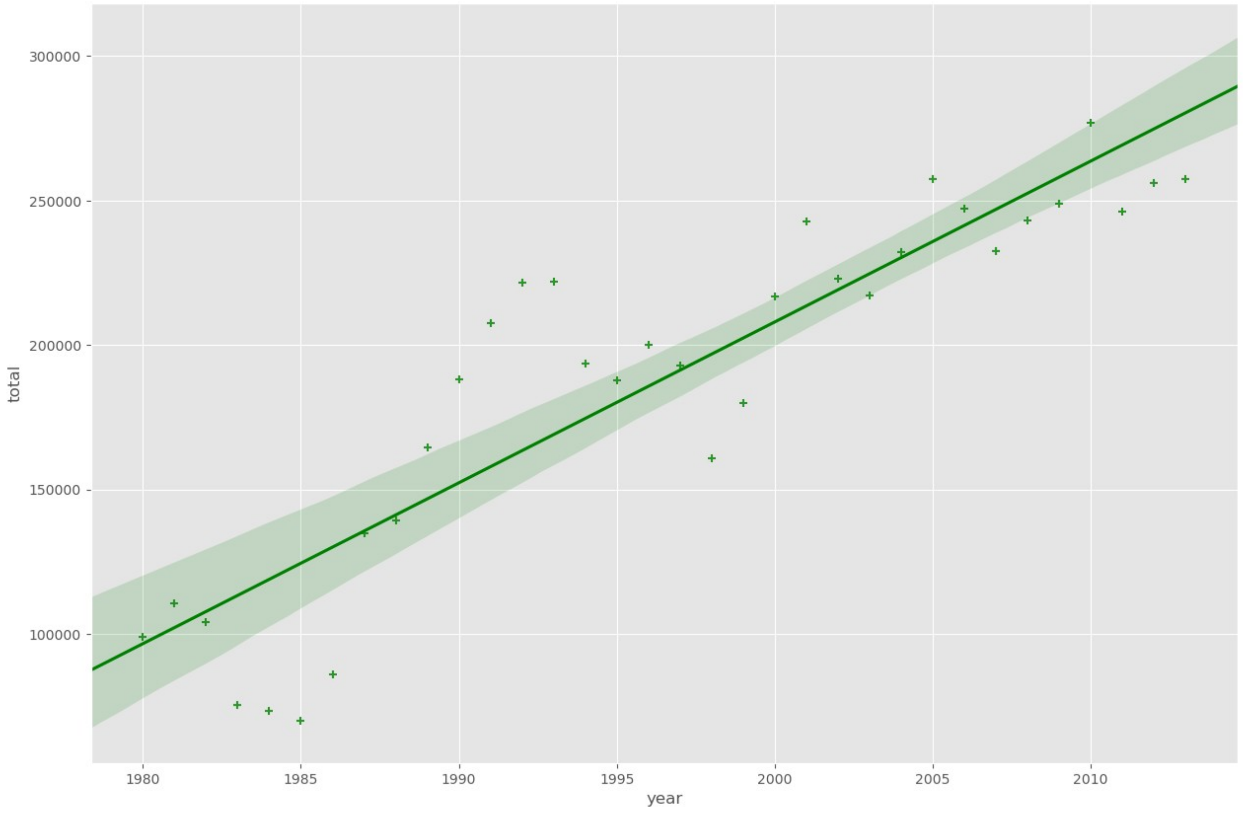
You can always customize the marker shape, so instead of circular markers, let's use `+`.

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



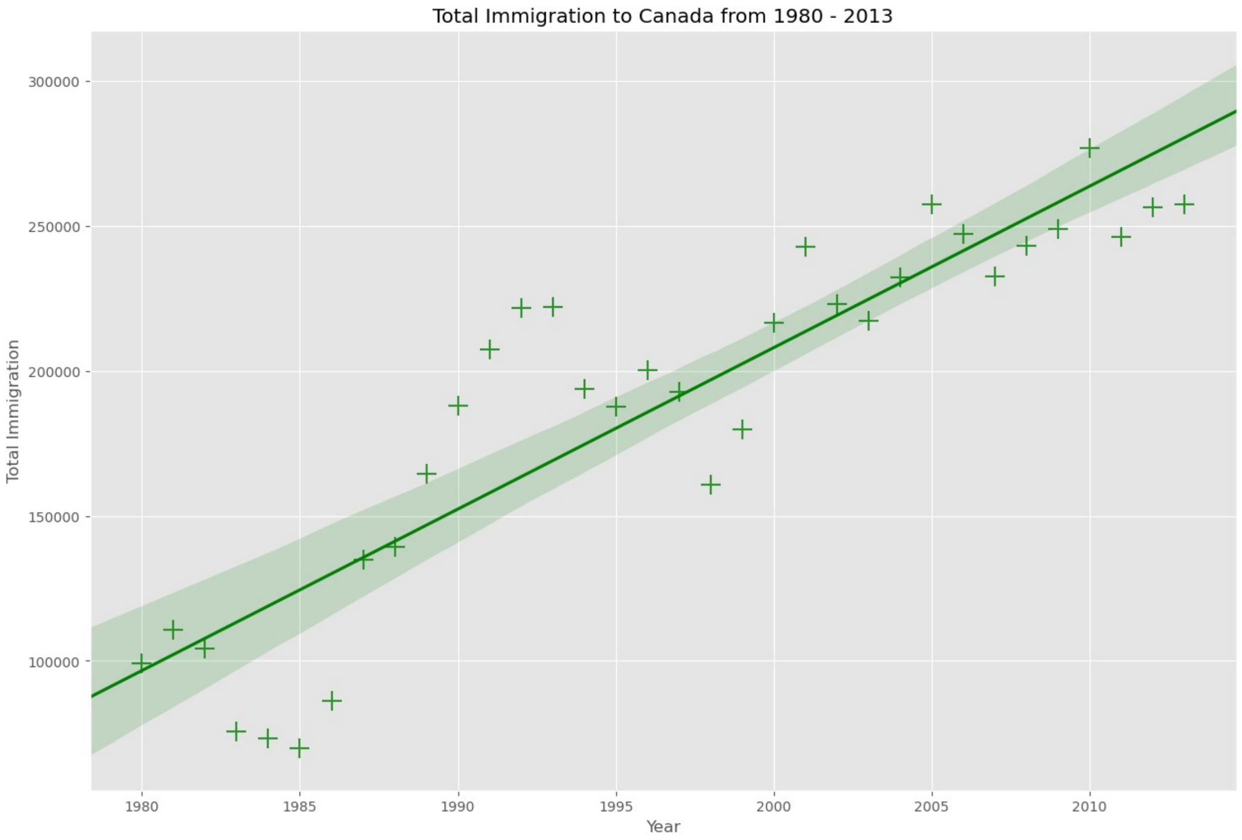
Let's blow up the plot a little so that it is more appealing to the sight.

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



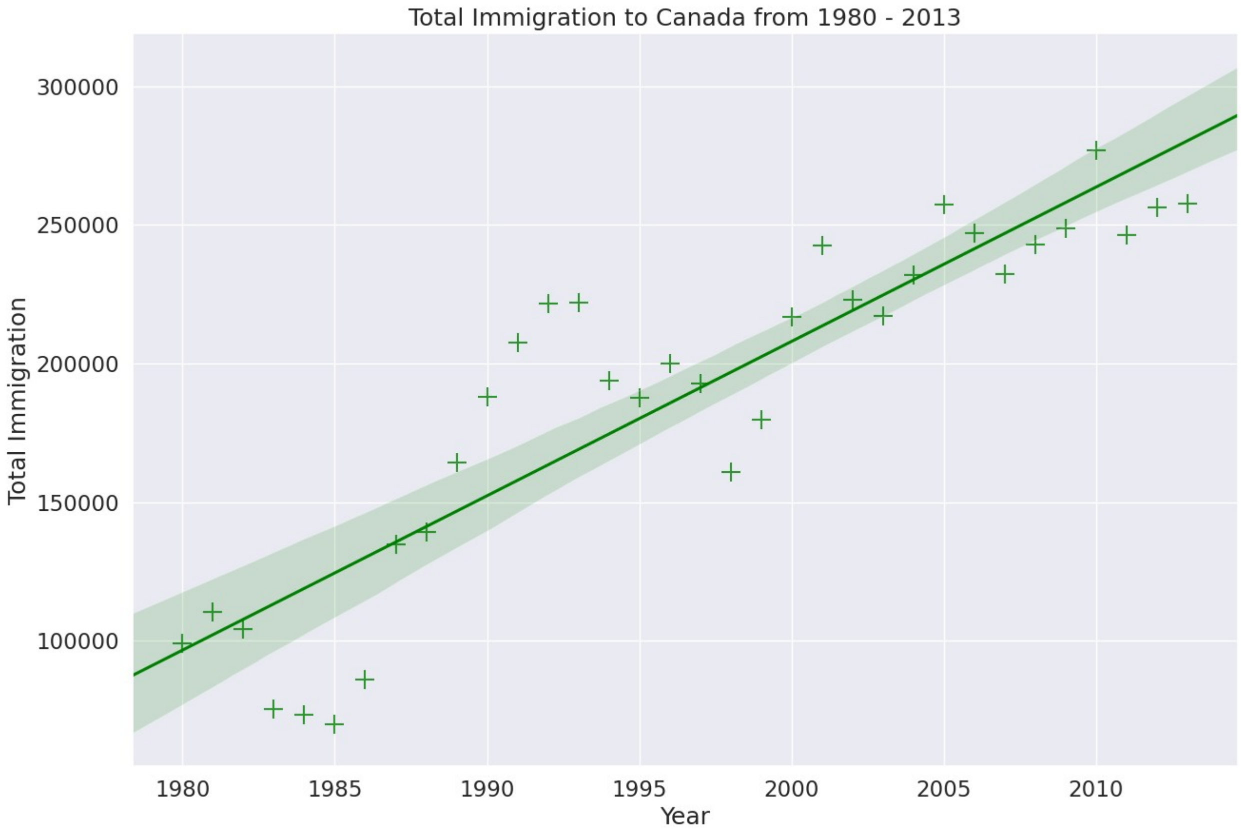
And let's increase the size of markers so they match the new size of the figure, and add a title and x- and y-labels.

```
plt.figure(figsize=(15, 10))
ax = sns.regplot(x='year', y='total', data=df_tot, color='green', marker='+', scatter_kws={'s': 200})

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



And finally increase the font size of the tickmark labels, the title, and the x- and y-labels so they don't feel left out!

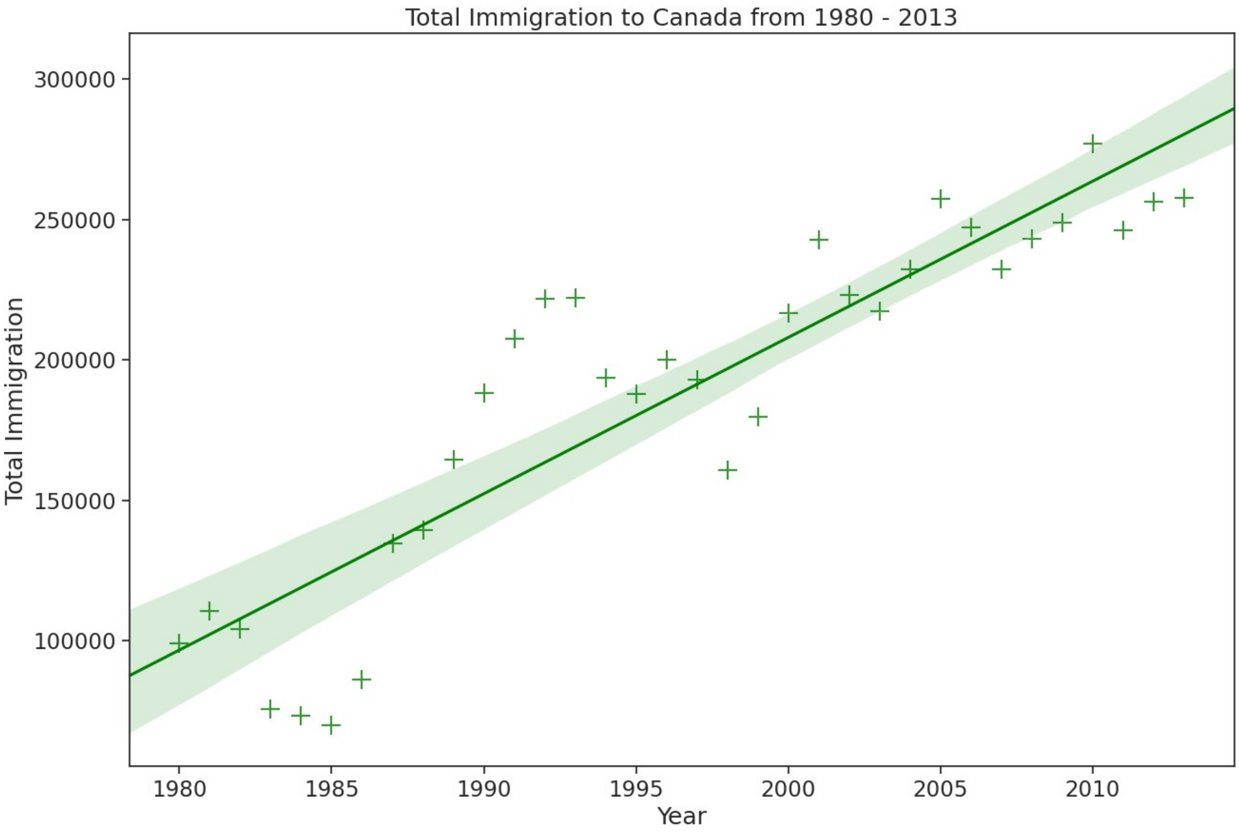
```
plt.figure(figsize=(15, 10))

sns.set(font_scale=1.5)

ax = sns.regplot(x='year', y='total', data=df_tot, color='green', 
marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



Amazing! A complete scatter plot with a regression fit with 5 lines of code only. Isn't this really amazing?

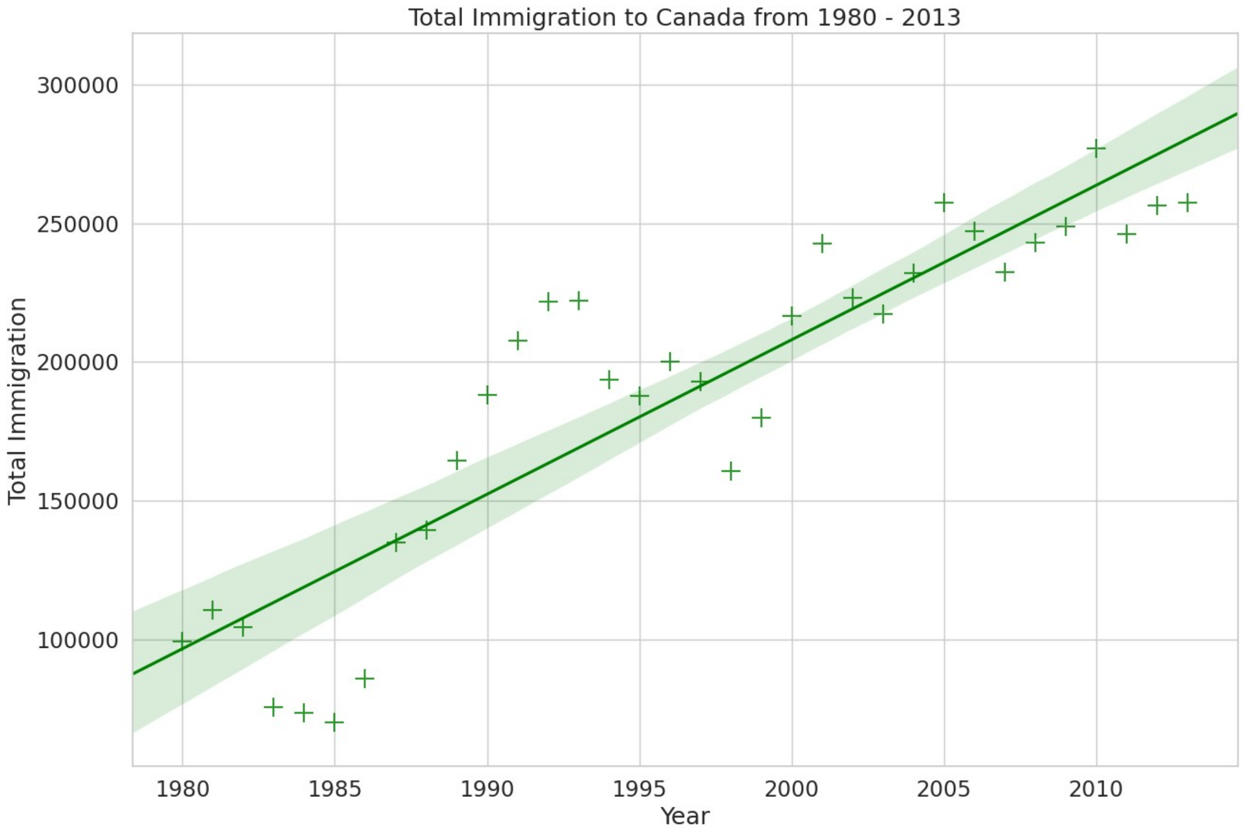
If you are not a big fan of the purple background, you can easily change the style to a white plain background.

```
plt.figure(figsize=(15, 10))

sns.set(font_scale=1.5)
sns.set_style('ticks')  # change background to white background

ax = sns.regplot(x='year', y='total', data=df_tot, color='green',
marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



Or to a white background with gridlines.

```
plt.figure(figsize=(15, 10))

sns.set(font_scale=1.5)
sns.set_style('whitegrid')

ax = sns.regplot(x='year', y='total', data=df_tot, color='green',
marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



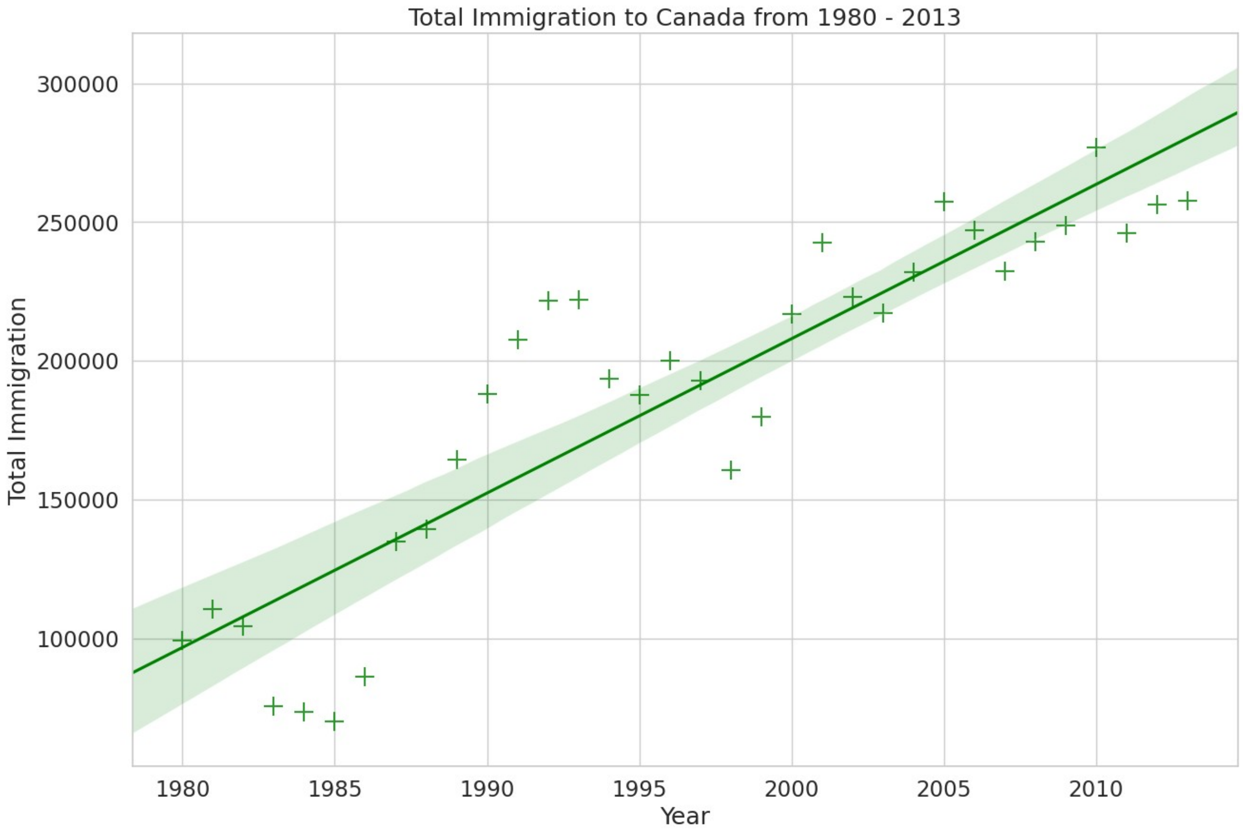
**Question**: Use seaborn to create a scatter plot with a regression line to visualize the total immigration from Denmark, Sweden, and Norway to Canada from 1980 to 2013.

```
# let's create a new dataframe for these three countries
df_countries  = df_can.loc[['Denmark', 'Norway', 'Sweden'],
years].transpose()

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

```
Country  Denmark  Norway  Sweden
1980         272     116     281
1981         293      77     308
1982         299     106     222
1983         106      51     176
1984          93      31     128
1985          73      54     158
1986          93      56     187
1987         109      80     198
1988         129      73     171
1989         129      76     182
1990         118      83     130
1991         111     103     167
1992         158      74     179
1993         186      92     203
```

```
1994          93      60     192
1995         111      65     176
1996          70      70     161
1997          83     104     151
1998          63      31     123
1999          81      36     170
2000          93      56     138
2001          81      78     184
2002          70      74     149
2003          89      77     161
2004          89      73     129
2005          62      57     205
2006         101      53     139
2007          97      73     193
2008         108      66     165
2009          81      75     167
2010          92      46     159
2011          93      49     134
2012          94      53     140
2013          81      59     140
```

```
# 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)

### type your answer here
plt.figure(figsize=(15, 10))

sns.set(font_scale=1.5)
sns.set_style('whitegrid')

ax = sns.regplot(x='year', y='total', data=df_tot, color='green',
marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')
plt.show()
```



Thank you for completing this lab!

## Author

Alex Aklson Dr. Pooja

## Change Log

| Date (YYYY-MM-DD) | Version | Changed By | Change Description |
|---|---|---|---|
| 2023-07-07 | 2.7 | Dr. Pooja | wordcloud, sns, piplite,pywaffle issue resolved |
| 2023-06-11 | 2.6 | Dr. Pooja | Clean data link, pywaffle,Categorical plots included |
| 2021-05-19 | 2.3 | Weiqing Wang | Fixed typos and code spells |
| 2021-01-21 | 2.2 | Lakshmi Holla | Updated TOC markdown cell |
| 2020-11-03 | 2.1 | Lakshmi Holla | Changed URL of excel file |
| 2020-08-27 | 2.0 | Lavanya | Moved lab to course repo in GitLab |