



(<https://cognitiveclass.ai>)

Introduction to Matplotlib and Line Plots

pandas Basics

The first thing we'll do is import two key data analysis modules: *pandas* and **Numpy**.

```
In [1]: import numpy as np # useful for many scientific computing in Python
import pandas as pd # primary data structure library
```

Let's 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 to read in excel files. This module is **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 **xlrd** module:

```
!conda install -c anaconda xlrd --yes
```

Now we are ready to read in our data.

```
In [2]: df_can = pd.read_excel('https://s3-api.us-gio.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DV0101EN/labs/Data_Files/Canada.xlsx',
                                sheet_name='Canada by Citizenship',
                                skiprows=range(20),
                                skipfooter=2)

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

Data read into a pandas dataframe!

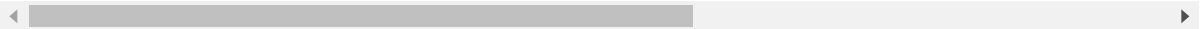
Let's view the top 5 rows of the dataset using the `head()` function.

```
In [3]: df_can.head()
# tip: You can specify the number of rows you'd like to see as follow
s: df_can.head(10)
```

Out[3]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	19
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	:
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	{
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	

5 rows × 43 columns



We can also view the bottom 5 rows of the dataset using the `tail()` function.

```
In [4]: df_can.tail()
```

Out[4]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	19
190	Immigrants	Foreigners	Viet Nam	935	Asia	920	South-Eastern Asia	902	Developing regions	1:
191	Immigrants	Foreigners	Western Sahara	903	Africa	912	Northern Africa	902	Developing regions	
192	Immigrants	Foreigners	Yemen	935	Asia	922	Western Asia	902	Developing regions	
193	Immigrants	Foreigners	Zambia	903	Africa	910	Eastern Africa	902	Developing regions	
194	Immigrants	Foreigners	Zimbabwe	903	Africa	910	Eastern Africa	902	Developing regions	

5 rows × 43 columns



When analyzing a dataset, it's always a good idea to start by getting basic information about your dataframe. We can do this by using the `info()` method.

```
In [11]: df_can.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Data columns (total 43 columns):
Type      195 non-null object
Coverage  195 non-null object
OdName    195 non-null object
AREA      195 non-null int64
AreaName  195 non-null object
REG        195 non-null int64
RegName    195 non-null object
DEV        195 non-null int64
DevName    195 non-null object
1980       195 non-null int64
1981       195 non-null int64
1982       195 non-null int64
1983       195 non-null int64
1984       195 non-null int64
1985       195 non-null int64
1986       195 non-null int64
1987       195 non-null int64
1988       195 non-null int64
1989       195 non-null int64
1990       195 non-null int64
1991       195 non-null int64
1992       195 non-null int64
1993       195 non-null int64
1994       195 non-null int64
1995       195 non-null int64
1996       195 non-null int64
1997       195 non-null int64
1998       195 non-null int64
1999       195 non-null int64
2000       195 non-null int64
2001       195 non-null int64
2002       195 non-null int64
2003       195 non-null int64
2004       195 non-null int64
2005       195 non-null int64
2006       195 non-null int64
2007       195 non-null int64
2008       195 non-null int64
2009       195 non-null int64
2010       195 non-null int64
2011       195 non-null int64
2012       195 non-null int64
2013       195 non-null int64
dtypes: int64(37), object(6)
memory usage: 65.6+ KB
```

To get the list of column headers we can call upon the dataframe's `.columns` parameter.

```
In [12]: df_can.columns.values
```

```
Out[12]: array(['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], dtype=object)
```

Similarly, to get the list of indices we use the `.index` parameter.

```
In [13]: df_can.index.values
```

```
Out[13]: array([ 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])
```

Note: The default type of index and columns is NOT list.

```
In [14]: print(type(df_can.columns))
print(type(df_can.index))

<class 'pandas.core.indexes.base.Index'>
<class 'pandas.core.indexes.range.RangeIndex'>
```

To get the index and columns as lists, we can use the `tolist()` method.

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

print (type(df_can.columns.tolist()))
print (type(df_can.index.tolist()))

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

To view the dimensions of the dataframe, we use the `.shape` parameter.

```
In [16]: # size of dataframe (rows, columns)
df_can.shape
```

```
Out[16]: (195, 43)
```

Note: The main types stored in *pandas* objects are *float*, *int*, *bool*, *datetime64[ns]* and *datetime64[ns, tz]* (in $\geq 0.17.0$), *timedelta[ns]*, *category* (in $\geq 0.15.0$), and *object* (string). In addition these dtypes have item sizes, e.g. *int64* and *int32*.

Let's clean the data set to remove a few unnecessary columns. We can use *pandas* `drop()` method as follows:

```
In [17]: # in pandas axis=0 represents rows (default) and axis=1 represents columns.
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)
df_can.head(2)
```

```
Out[17]:
```

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450

2 rows × 38 columns



Let's rename the columns so that they make sense. We can use `rename()` method by passing in a dictionary of old and new names as follows:

```
In [18]: df_can.rename(columns={'OdName': 'Country', 'AreaName': 'Continent', 'RegionName': 'Region'}, inplace=True)
df_can.columns
```

```
Out[18]: Index([ 'Country', 'Continent', 'Region', 'DevName', 198
0,
1981, 1982, 1983, 1984, 198
5,
1986, 1987, 1988, 1989, 199
0,
1991, 1992, 1993, 1994, 199
5,
1996, 1997, 1998, 1999, 200
0,
2001, 2002, 2003, 2004, 200
5,
2006, 2007, 2008, 2009, 201
0,
2011, 2012, 2013],
dtype='object')
```

We will also add a 'Total' column that sums up the total immigrants by country over the entire period 1980 - 2013, as follows:

```
In [19]: df_can['Total'] = df_can.sum(axis=1)
```

We can check to see how many null objects we have in the dataset as follows:

```
In [20]: df_can.isnull().sum()
```

```
Out[20]: 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
```

Finally, let's view a quick summary of each column in our dataframe using the `describe()` method.

```
In [21]: df_can.describe()
```

```
Out[21]:
```

	1980	1981	1982	1983	1984	1985	
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195
mean	508.394872	566.989744	534.723077	387.435897	376.497436	358.861538	408
std	1949.588546	2152.643752	1866.997511	1204.333597	1198.246371	1079.309600	1204
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
50%	13.000000	10.000000	11.000000	12.000000	13.000000	17.000000	13
75%	251.500000	295.500000	275.000000	173.000000	181.000000	197.000000	251
max	22045.000000	24796.000000	20620.000000	10015.000000	10170.000000	9564.000000	945

8 rows × 35 columns

pandas Intermediate: Indexing and Selection (slicing)

Select Column

There are two ways to filter on a column name:

Method 1: Quick and easy, but only works if the column name does NOT have spaces or special characters.

```
df.column_name
    (returns series)
```

Method 2: More robust, and can filter on multiple columns.

```
df['column']
    (returns series)

df[['column 1', 'column 2']]
    (returns dataframe)
```

Example: Let's try filtering on the list of countries ('Country').


```
In [22]: df_can.Country.head() # returns a series
```

```
Out[22]: 0      Afghanistan
1      Albania
2      Algeria
3  American Samoa
4      Andorra
Name: Country, dtype: object
```

Let's try filtering on the list of countries ('OdName') and the data for years: 1980 - 1985.

```
In [23]: df_can[['Country', 1980, 1981, 1982, 1983, 1984, 1985]].head() # 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[23]:
```

	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

Select Row

There are main 3 ways to select rows:

```
df.loc[label]
#filters by the labels of the index/column
df.iloc[index]
#filters by the positions of the index/column
```

Before we proceed, notice that the default index of the dataset is a numeric range from 0 to 194. This makes it very difficult to do a query by a specific country. For example to search for data on Japan, we need to know the corresponding index value.

This can be fixed very easily by setting the 'Country' column as the index using `set_index()` method.

```
In [24]: 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_index()
```

In [25]: `df_can.head(3)`

Out[25]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2013
Country												
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	34
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3

3 rows × 38 columns

In [26]: `# optional: to remove the name of the index`
`df_can.index.name = None`

Example: Let's view the number of immigrants from Japan (row 87) for the following scenarios:

1. The full row data (all columns)
2. For year 2013
3. For years 1980 to 1985

```
In [32]: # 1. the full row data (all columns)
print(df_can.loc['Japan'].head())
print("-----")
# alternate methods
print(df_can.iloc[87].head())
print("-----")
print(df_can[df_can.index == 'Japan'].T.squeeze().head())
```

```
Continent          Asia
Region            Eastern Asia
DevName    Developed regions
1980                701
1981                756
Name: Japan, dtype: object
-----
```

```
Continent          Asia
Region            Eastern Asia
DevName    Developed regions
1980                701
1981                756
Name: Japan, dtype: object
-----
```

```
Continent          Asia
Region            Eastern Asia
DevName    Developed regions
1980                701
1981                756
Name: Japan, dtype: object
```

```
In [33]: # 2. for year 2013
print(df_can.loc['Japan', 2013])

# alternate method
print(df_can.iloc[87, 36]) # year 2013 is the last column, with a pos
itional index of 36
```

```
982
982
```

```
In [34]: # 3. for years 1980 to 1985
print(df_can.loc['Japan', [1980, 1981, 1982, 1983, 1984, 1984]])
print(df_can.iloc[87, [3, 4, 5, 6, 7, 8]])
```

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

Name: Japan, dtype: object

1980	701
1981	756
1982	598
1983	309
1984	246
1985	198

Name: Japan, dtype: object

Column names that are integers (such as the years) might introduce some confusion. For example, when we are referencing the year 2013, one might confuse that when the 2013th positional index.

To avoid this ambiguity, let's convert the column names into strings: '1980' to '2013'.

```
In [35]: df_can.columns = list(map(str, df_can.columns))
# [print (type(x)) for x in df_can.columns.values] #<-- uncomment to
# check type of column headers
```

Since we converted the years to string, let's declare a variable that will allow us to easily call upon the full range of years:

```
In [37]: # useful for plotting later on  
years = list(map(str, range(1980, 2014)))  
years
```

```
Out[37]: ['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']
```

Filtering based on a criteria

To filter the dataframe based on a condition, we simply pass the condition as a boolean vector.

For example, Let's filter the dataframe to show the data on Asian countries (AreaName = Asia).

```
In [38]: # 1. create the condition boolean series
condition = df_can['Continent'] == 'Asia'
print(condition.head())
```

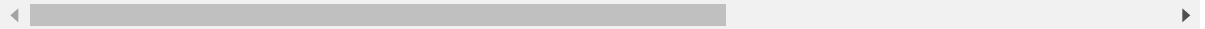
```
Afghanistan      True
Albania           False
Algeria           False
American Samoa   False
Andorra           False
Name: Continent, dtype: bool
```

```
In [39]: # 2. pass this condition into the dataframe
df_can[condition].head()
```

Out[39]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2016
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	34
Armenia	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	;
Azerbaijan	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	;
Bahrain	Asia	Western Asia	Developing regions	0	2	1	1	1	3	0	...	
Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	4;

5 rows × 38 columns



```
In [40]: # we can pass mutliple 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 A
sia')]

# note: When using 'and' and 'or' operators, pandas requires we use
'&' and '|' instead of 'and' and 'or'
# don't forget to enclose the two conditions in parentheses
```

Out[40]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	
Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	
Bhutan	Asia	Southern Asia	Developing regions	0	0	0	0	1	0	0	...	
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	30
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	!
Maldives	Asia	Southern Asia	Developing regions	0	0	0	1	0	0	0	...	
Nepal	Asia	Southern Asia	Developing regions	1	1	6	1	2	4	13	...	
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	1.
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	

9 rows × 38 columns



Before we proceed: let's review the changes we have made to our dataframe.

```
In [41]: 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[41]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2013
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	34
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1

2 rows × 38 columns

Visualizing Data using Matplotlib

Matplotlib: Standard Python Visualization Library

The primary plotting library we will explore in the course is [Matplotlib \(http://matplotlib.org/\)](http://matplotlib.org/). As mentioned on their website:

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

If you are aspiring to create impactful visualization with python, Matplotlib is an essential tool to have at your disposal.

Matplotlib.Pyplot

One of the core aspects of Matplotlib is `matplotlib.pyplot`. It is Matplotlib's scripting layer which we studied in details in the videos about Matplotlib. Recall that it is a collection of command style functions that make Matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In this lab, we will work with the scripting layer to learn how to generate line plots. In future labs, we will get to work with the Artist layer as well to experiment first hand how it differs from the scripting layer.

Let's start by importing Matplotlib and Matplotlib.pyplot as follows:

```
In [42]: # we are using the inline backend
%matplotlib inline

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

*optional: check if Matplotlib is loaded.

```
In [43]: print ('Matplotlib version: ', mpl.__version__) # >= 2.0.0

Matplotlib version: 3.1.0
```

*optional: apply a style to Matplotlib.

```
In [44]: print(plt.style.available)
mpl.style.use(['ggplot']) # optional: for ggplot-like style

['seaborn-dark-palette', 'seaborn-white', 'seaborn-poster', 'seaborn-
dark', 'fast', 'dark_background', 'seaborn-colorblind', 'seaborn-tal
k', 'tableau-colorblind10', 'seaborn-deep', 'seaborn-darkgrid', 'seab
orn-paper', 'seaborn-pastel', 'seaborn-muted', 'seaborn', 'Solarize_L
ight2', 'seaborn-whitegrid', 'bmh', 'seaborn-bright', 'classic', '_cl
assic_test', 'grayscale', 'ggplot', 'fivethirtyeight', 'seaborn-noteb
ook', 'seaborn-ticks']
```

Plotting in *pandas*

Fortunately, *pandas* has a built-in implementation of Matplotlib that we can use. Plotting in *pandas* is as simple as appending a `.plot()` method to a series or dataframe.

Documentation:

- [Plotting with Series](http://pandas.pydata.org/pandas-docs/stable/api.html#plotting) (<http://pandas.pydata.org/pandas-docs/stable/api.html#plotting>)
- [Plotting with Dataframes](http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-plotting) (<http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-plotting>)

Line Pots (Series/Dataframe)

What is a line plot and why use it?

A line chart or line plot is a type of plot which displays information as a series of data points called 'markers' connected by straight line segments. It is a basic type of chart common in many fields. Use line plot when you have a continuous data set. These are best suited for trend-based visualizations of data over a period of time.

Let's start with a case study:

In 2010, Haiti suffered a catastrophic magnitude 7.0 earthquake. The quake caused widespread devastation and loss of life and about three million people were affected by this natural disaster. As part of Canada's humanitarian effort, the Government of Canada stepped up its effort in accepting refugees from Haiti. We can quickly visualize this effort using a Line plot:

Question: Plot a line graph of immigration from Haiti using `df.plot()`.

First, we will extract the data series for Haiti.

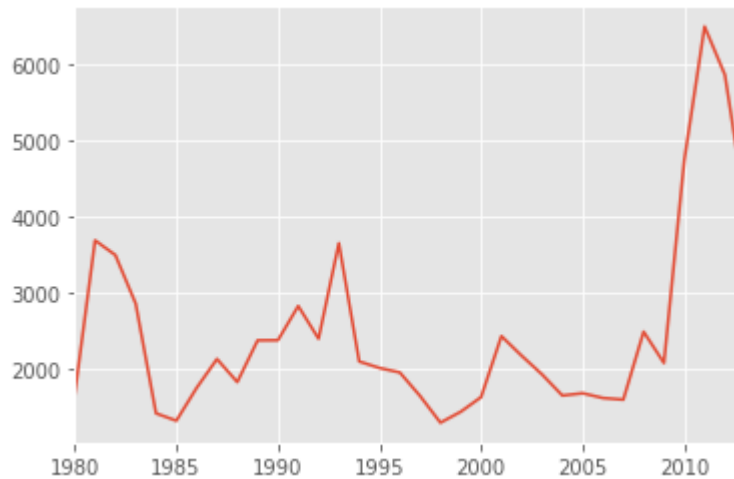
```
In [45]: haiti = df_can.loc['Haiti', years] # passing in years 1980 - 2013 to
         exclude the 'total' column
         haiti.head()
```

```
Out[45]: 1980    1666
         1981    3692
         1982    3498
         1983    2860
         1984    1418
         Name: Haiti, dtype: object
```

Next, we will plot a line plot by appending `.plot()` to the `haiti` dataframe.

```
In [46]: haiti.plot()
```

```
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd076bb1438>
```



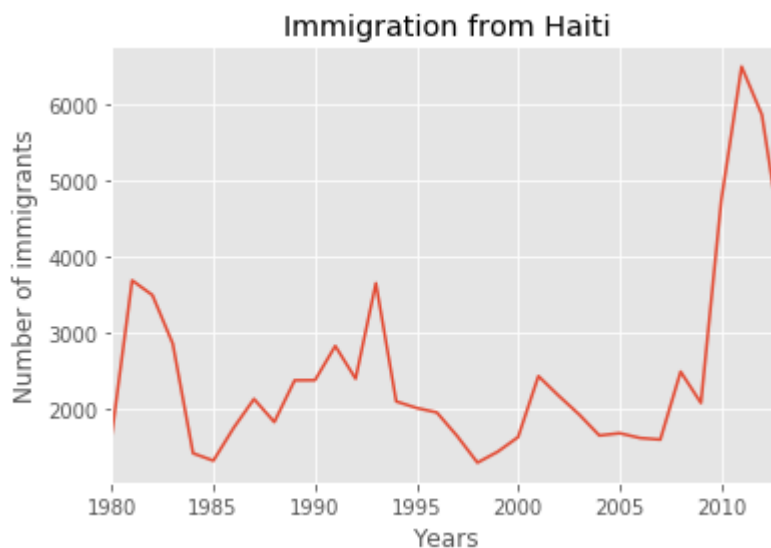
pandas automatically populated the x-axis with the index values (years), and the y-axis with the column values (population). However, notice how the years were not displayed because they are of type *string*. Therefore, let's change the type of the index values to *integer* for plotting.

Also, let's label the x and y axis using `plt.title()`, `plt.ylabel()`, and `plt.xlabel()` as follows:

```
In [47]: haiti.index = haiti.index.map(int) # let's change the index values of
Haiti to type integer for plotting
haiti.plot(kind='line')

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
```



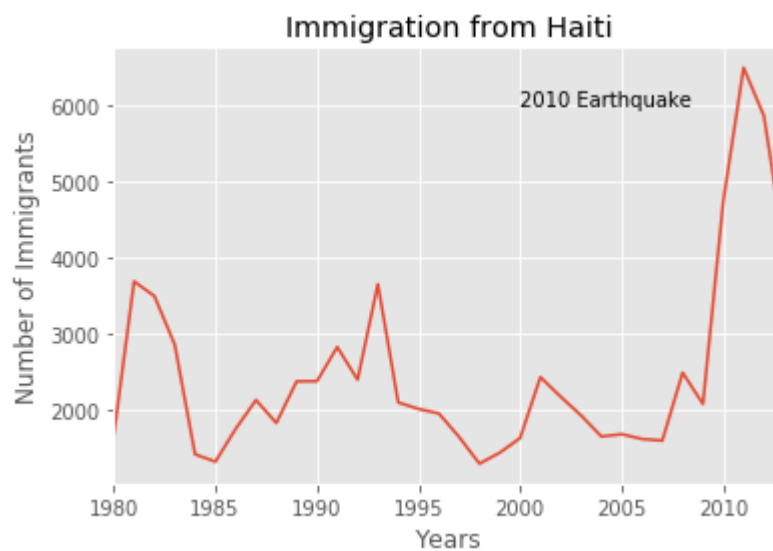
We can clearly notice how number of immigrants from Haiti spiked up from 2010 as Canada stepped up its efforts to accept refugees from Haiti. Let's annotate this spike in the plot by using the `plt.text()` method.

```
In [48]: 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.

We can easily add more countries to line plot to make meaningful comparisons immigration from different countries.

Question: Let's compare the number of immigrants from India and China from 1980 to 2013.

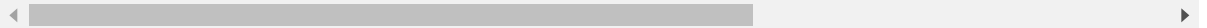
Step 1: Get the data set for China and India, and display dataframe.

```
In [49]: ### type your answer here
df_CI = df_can.loc[['India', 'China'], years]
df_CI.head()
```

Out[49]:

	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	33841
China	5123	6682	3308	1863	1527	1816	1960	2643	2758	4323	...	36619	42584	33542

2 rows × 34 columns



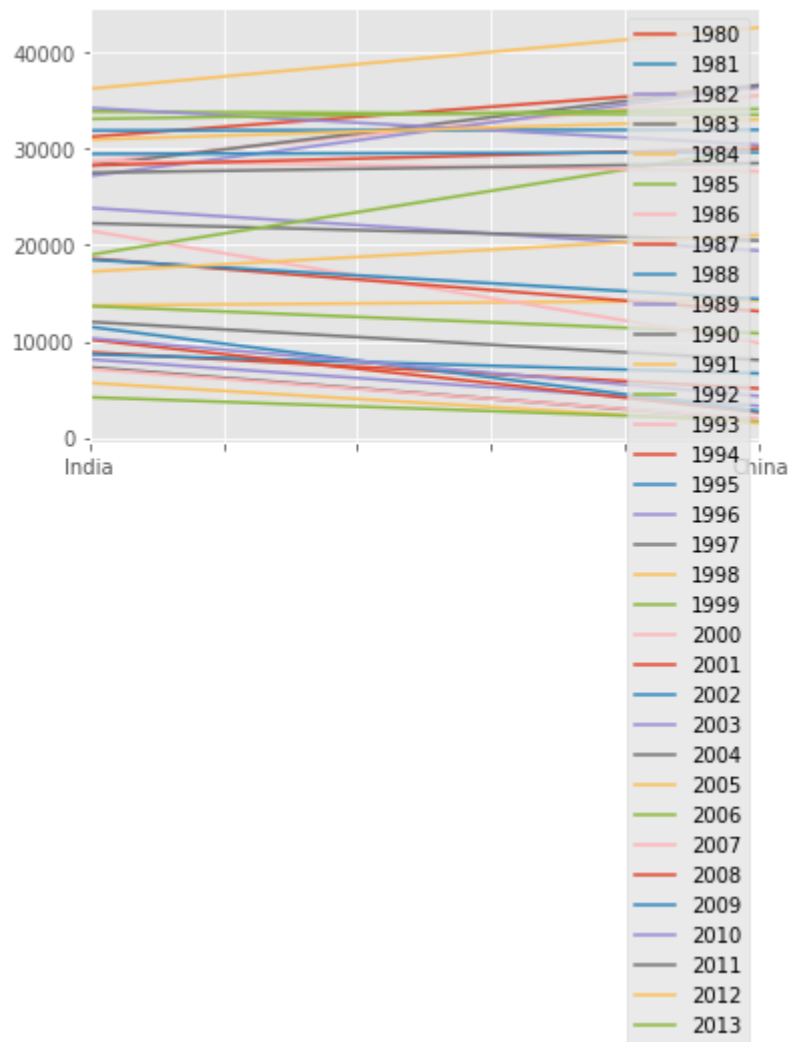
Double-click **here** for the solution.

Step 2: Plot graph. We will explicitly specify line plot by passing in `kind` parameter to `plot()`.

In [50]: *### type your answer here*

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

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd075cc9710>



Double-click [here](#) for the solution.

That doesn't look right...

Recall that *pandas* plots the indices on the x-axis and the columns as individual lines on the y-axis. Since `df_CI` is a dataframe with the `country` as the index and `years` as the columns, we must first transpose the dataframe using `transpose()` method to swap the row and columns.

```
In [51]: df_CI = df_CI.transpose()
df_CI.head()
```

Out[51]:

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

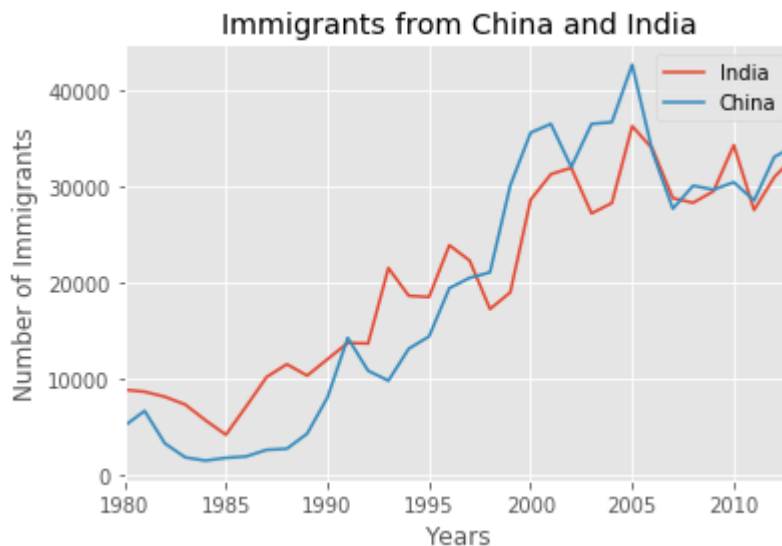
pandas will automatically graph the two countries on the same graph. Go ahead and plot the new transposed dataframe. Make sure to add a title to the plot and label the axes.

```
In [52]: ### type your answer here
```

```
df_CI.index = df_CI.index.map(int) # let's change the index values of
df_CI to type integer for plotting
df_CI.plot(kind='line')

plt.title('Immigrants from China and India')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```



Double-click **here** for the solution.

From the above plot, we can observe that the China and India have very similar immigration trends through the years.

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.

Question: Compare the trend of top 5 countries that contributed the most to immigration to Canada.


```
In [53]: ### type your answer here

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)

df_top5.index = df_top5.index.map(int) # let's change the index values of df_top5 to type integer for plotting
df_top5.plot(kind='line', figsize=(14, 8)) # pass a tuple (x, y) size

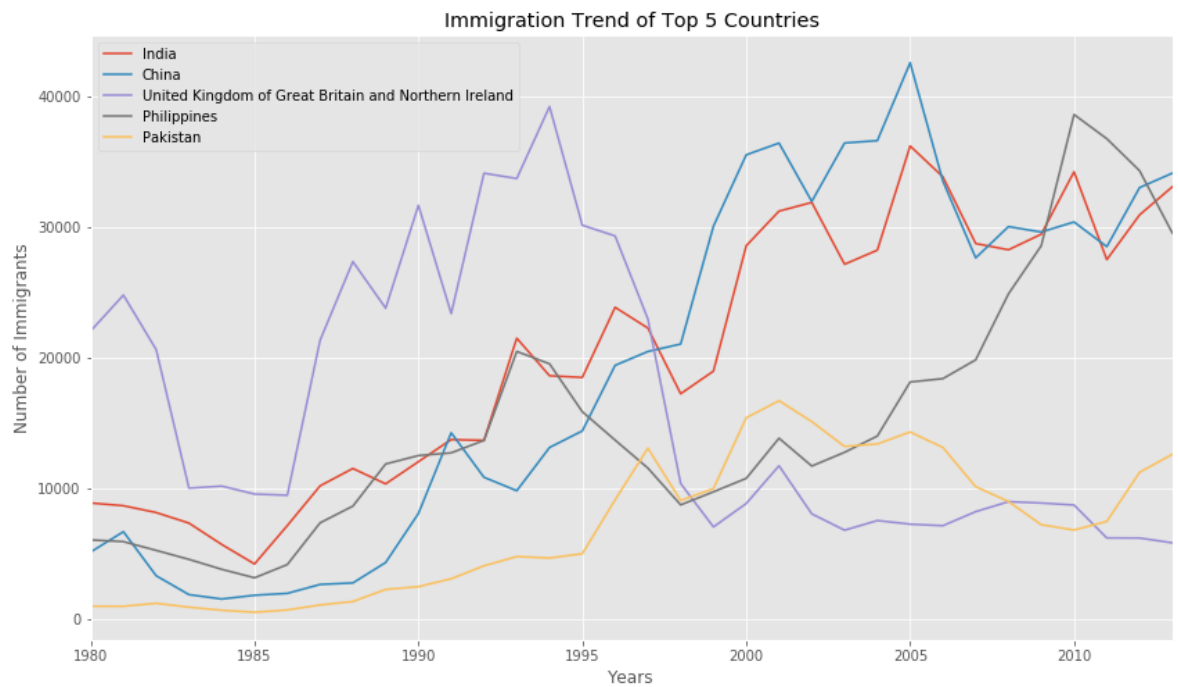
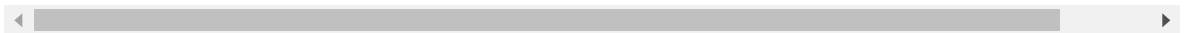
plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```

	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

	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



Double-click [here](#) for the solution.

Other Plots

Congratulations! you have learned how to wrangle data with python and create a line plot with Matplotlib. There are many other plotting styles available other than the default Line plot, all of which can be accessed by passing `kind` keyword to `plot()`. The full list of available plots are as follows:

- `bar` for vertical bar plots
- `barh` for horizontal bar plots
- `hist` for histogram
- `box` for boxplot
- `kde` or `density` for density plots
- `area` for area plots
- `pie` for pie plots
- `scatter` for scatter plots
- `hexbin` for hexbin plot

Thank you for completing this lab!

This notebook was originally created by [Jay Rajasekharan](https://www.linkedin.com/in/jayrajasekharan) with contributions from [Ehsan M. Kermani](https://www.linkedin.com/in/ehsanmkermani), and [Slobodan Markovic](https://www.linkedin.com/in/slobodan-markovic).

This notebook was recently revised by [Alex Aklson](https://www.linkedin.com/in/aklson/). I hope you found this lab session interesting. Feel free to contact me if you have any questions!

This notebook is part of a course on **Coursera** called *Data Visualization with Python*. If you accessed this notebook outside the course, you can take this course online by clicking [here](http://cocl.us/DV0101EN_Coursera_Week1_LAB1).

Copyright © 2019 [Cognitive Class](https://cognitiveclass.ai/?utm_source=bducopyrightlink&utm_medium=dswb&utm_campaign=bdu). This notebook and its source code are released under the terms of the [MIT License](https://bigdatauniversity.com/mit-license/).

In []: