

# DV0101EN-1-1-1-Introduction-to-Matplotlib-and-Line-Plots-py-v2.0

December 4, 2018

## Introduction to Matplotlib and Line Plots

### 0.1 Introduction

The aim of these labs is to introduce you to data visualization with Python as concrete and as consistent as possible. Speaking of consistency, because there is no *best* data visualization library available for Python - up to creating these labs - we have to introduce different libraries and show their benefits when we are discussing new visualization concepts. Doing so, we hope to make students well-rounded with visualization libraries and concepts so that they are able to judge and decide on the best visualization technique and tool for a given problem *and* audience.

Please make sure that you have completed the prerequisites for this course, namely **Python for Data Science** and **Data Analysis with Python**, which are part of this specialization.

**Note:** The majority of the plots and visualizations will be generated using data stored in *pandas* dataframes. Therefore, in this lab, we provide a brief crash course on *pandas*. However, if you are interested in learning more about the *pandas* library, detailed description and explanation of how to use it and how to clean, munge, and process data stored in a *pandas* dataframe are provided in our course **Data Analysis with Python**, which is also part of this specialization.

---

### 0.2 Table of Contents

1. Section ?? 1.1 Section ?? 1.2 Section ?? 1.3 Section ??
2. Section ?? 2.1 Section ??
3. Section ??

## 1 Exploring Datasets with *pandas*

*pandas* is an essential data analysis toolkit for Python. From their [website](#): *>pandas* is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** data analysis in Python.

The course heavily relies on *pandas* for data wrangling, analysis, and visualization. We encourage you to spend some time and familiarize yourself with the *pandas* API Reference: <http://pandas.pydata.org/pandas-docs/stable/api.html>.

## 1.1 The Dataset: Immigration to Canada from 1980 to 2013

Dataset Source: [International migration flows to and from selected countries - The 2015 revision](#).

The dataset contains annual data on the flows of international immigrants 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. The current version presents data pertaining to 45 countries.

In this lab, we will focus on the Canadian immigration data.

For sake of simplicity, Canada's immigration data has been extracted and uploaded to one of IBM servers. You can fetch the data from [here](#).

---

## 1.2 *pandas* Basics

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

```
In [ ]: 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 [ ]: df_can = pd.read_excel('https://ibm.box.com/shared/static/lw190pt9zpy5bd1ptyg2aw15awomz9
                                sheet_name='Canada by Citizenship',
                                skiprows=range(20),
                                skipfooter=2)

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

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

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

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

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

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 [ ]: df_can.info()
```

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

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

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

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

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

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

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

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

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

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

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

Note: The main types stored in *pandas* objects are *float*, *int*, *bool*, *datetime64[ns]* and *date-time64[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 [ ]: # in pandas axis=0 represents rows (default) and axis=1 represents columns.
        df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)
        df_can.head(2)
```

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

```
In [ ]: df_can.rename(columns={'OdName': 'Country', 'AreaName': 'Continent', 'RegName': 'Region'},
        df_can.columns
```

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

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

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

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

## 1.3 *pandas* Intermediate: Indexing and Selection (slicing)

### 1.3.1 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 [ ]: df_can.Country # returns a series
```

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

```
In [ ]: 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.
```

### 1.3.2 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 [ ]: 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 [ ]: df_can.head(3)
```

```
In [ ]: # 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 [ ]: # 1. the full row data (all columns)
        print(df_can.loc['Japan'])

        # alternate methods
        print(df_can.iloc[87])
        print(df_can[df_can.index == 'Japan'].T.squeeze())
```

```
In [ ]: # 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 positional index of 36
```

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

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 [ ]: 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
```

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

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

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

```
In [ ]: # we can pass mutltiple 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
```

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

```
In [ ]: print ('data dimensions:', df_can.shape)
        print(df_can.columns)
        df_can.head(2)
```

## 2 Visualizing Data using Matplotlib

### 2.1 Matplotlib: Standard Python Visualization Library

The primary plotting library we will explore in the course is [Matplotlib](#). 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.

#### 2.1.1 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 [ ]: # we are using the inline backend
        %matplotlib inline

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

\*optional: check if Matplotlib is loaded.

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

\*optional: apply a style to Matplotlib.

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

### 2.1.2 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](#) - [Plotting with Dataframes](#)

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

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

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

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

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

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

```
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 t

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

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

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

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

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



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

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

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

### 3.0.3 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

### 3.0.4 Thank you for completing this lab!

This notebook was originally created by [Jay Rajasekharan](#) with contributions from [Ehsan M. Kermani](#), and [Slobodan Markovic](#).

This notebook was recently revised by [Alex 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](#).

Copyright © 2018 [Cognitive Class](#). This notebook and its source code are released under the terms of the [MIT License](#).