

<https://github-wiki-see.page/m/brendensong/IBM-Data-Science-Professional-Certificate/wiki>

https://github.com/brendensong/IBM-Data-Science-Professional-Certificate/tree/main/8_Data%20Visualization%20with%20Python

Welcome

Hello everyone and welcome to **data visualization with Python**. I'm Alex Akison, a data scientist at IBM, and I'm your instructor for this course. Throughout this course we're gonna learn how to create meaningful, effective, and aesthetically pleasing data visuals and plots in python using Matplotlib and a couple of other libraries namely Seaborn and Folium.

This course will consist of three modules. In **module 1**, we will briefly discuss data visualization and some of the best practices to keep in mind when creating data visuals. We will then learn about Matplotlib: its history, architecture, and the three layers that form its architecture. We will also learn about the data set that we will use throughout the course in these lectures as well as the hands-on sessions. We will essentially be working with a data set that was curated by the United Nations on immigration from different countries to Canada from 1980 to 2013.

Then we will start learning how to use Matplotlib to create plots and visuals, and we will start off with line plots. Now, we will generate the majority of our plots and visualizations in this course using data stored in pandas dataframes. For those of you who don't know what pandas is, pandas is a python library for data manipulation and analysis.

So before we start building visualizations and plots, we will take a brief crash course on pandas and learn how to use it to read data from csv files like the one shown here into what is called a pandas dataframe like the one shown here.

Now, if you are interested in learning more about the pandas library, we actually cover it in much more detail in our next course in this specialization which is Data Analysis with Python, so make sure to complete the next course in this specialization.

In **module 2**, we will continue on with a few more basic data visualizations such as area plots, histograms, and bar charts, and learn how to use Matplotlib to create them and even create different versions of these plots. We will also cover a set of specialized visualizations such as pie charts, box plots, scatter plots, and bubble plots, and we will learn how to create them still using Matplotlib.

In **module 3**, we will learn about more advanced visuals such as waffle charts that provide a fine-grained view of the proportions of different categories in a dataset. We will also learn about word clouds that depict word frequency or importance in a body of text. Also, in this module, we will explore another library, seaborn, which is built on top of Matplotlib to simplify the process of creating plots and visuals, and we will get a taste of its effectiveness through the creation of regression plots.

Finally, in this module, we will explore another library, folium, which was built primarily to visualize geospatial data. So, we will learn how to create maps of different regions of the world, superimpose markers of different shapes on top of maps, and learn how to create choropleth maps.

Before I conclude this video, let me stress one thing. Data visualization is best learned through hands-on exercises and sessions. Therefore, don't worry if you find some of the videos to be short. The labs and the hands-on sessions are very thorough and cover a lot of the concepts that are discussed in the videos in much more detail, so it is very important that you complete the labs and the hands-on sessions, although they are ungraded components of the course. I hope that you remember this and keep it in mind as you progress in this course. After completing this course, you'll be able to use different visualization libraries in Python namely, Matplotlib, seaborn, and folium to create expressive visual representations of your data for different purposes.

General Information

"A picture is worth a thousand words". We are all familiar with this expression. It especially applies when trying to explain the insight obtained from the analysis of increasingly large datasets. Data visualization plays an essential role in the representation of both small and large-scale data.

One of the key skills of a data scientist is the ability to tell a compelling story, visualizing data and findings in an approachable and stimulating way. Learning how to leverage a software tool to visualize data will also enable you to extract information, better understand the data, and make more effective decisions.

The main goal of this Data Visualization with Python course is to teach you how to take data that at first glance has little meaning and present that data in a form that makes sense to people. Various techniques have been developed for presenting data visually but in this course, we will be using several data visualization libraries in Python, namely Matplotlib, Seaborn, and Folium.

This course is self-paced, can be taken at any time, and audited for free as many times as you wish. If you are taking the course validated there is only **ONE** chance to pass the course, but multiple attempts per question (see the [Grading Scheme](#) section for details).

Learning Objectives

In this course you will learn about:

- Data visualization and some of the best practices when creating plots and visuals.
- The history and architecture of *Matplotlib*, and how to do basic plotting with Matplotlib.
- Generating different visualization tools using *Matplotlib* such as line plots, area plots, histograms, bar charts, box plots, and pie charts.
- *Seaborn*, another data visualization library in Python, and how to use it to create attractive statistical graphics.
- *Folium*, and how to use it to create maps and visualize geospatial data.

Syllabus

Module 1 - Introduction to Visualization Tools

- Introduction to Data Visualization
- Introduction to Matplotlib
- Basic Plotting with Matplotlib
- Dataset on Immigration to Canada
- Line Plots
- Hands-on Lab: Introduction to Matplotlib and Line Plots
- Graded Quiz
- Optional Reading: Download Jupyter Notebook

Module 2 - Basic and Specialized Visualization Tools

- Area Plots
- Histograms
- Bar Charts
- Hands-on Lab: Basic Visualization Tools
- Optional: Download Jupyter Notebook

Module 3: Specialized Visualization Tools

- Pie Charts
- Box Plots
- Scatter Plots
- Hands-on Lab: Specialized Visualization Tools
- Graded Quiz
- Optional: Download Jupyter Notebook

Module 4 - Advanced Visualization Tools

- Waffle Charts
- Word Clouds
- Seaborn and Regression Plots
- Hands-on Lab: Advanced Visualization Tools
- Optional: Download Jupyter Notebook

Module 5 - Creating Maps and Visualizing Geospatial Data

- Introduction to Folium
- Maps with Markers
- Choropleth Maps
- Hands-on Lab: Generating Maps in Python
- Graded Quiz
- Optional: Download Jupyter Notebook

Module 6 - Creating Dashboards with Plotly and Dash

- Dashboarding Overview
- Additional Resources for Dashboards
- Introduction to Plotly
- Additional Resources for Plotly
- Hands-on Lab: Plotly Basics - Scatter, Line, Bar, Bubble, Histogram, Pie, Sunburst
- Introduction to Dash
- Additional Resources for Dash
- Hands-on Lab: Dash Basics - HTML and Core Components, Multiple Charts
- Make Dashboards Interactive
- Additional Resources for Interactive Dashboards
- Add Interactivity: User Input and Callbacks
- Hands-on Lab: Flight Delay Time Statistics Dashboard
- Lesson Summary
- Graded Quiz

Final Assignment

- Final Assignment
- Peer Graded Assignment

Final Exam

- Final Exam

Grading Scheme

1. The minimum passing mark for the **course** is 70% with the following weights:
 - Graded Quizzes - %60
 - Final Assignment - 15%
 - Final Exam - 25%
2. Though Review Questions and the Final Exam have a passing mark of 50% respectively, the only grade that matters is the overall grade for the **course**.
3. Review Questions have no time limit. You are encouraged to review the course material to find the answers. Please remember that the Review Questions are worth 50% of your final mark.
4. The final exam has a 1-hour time limit.
5. Attempts are per **question** in both, the Review Questions and the Final Exam:
 - One attempt - For True/False questions
 - Two attempts - For any question other than True/False
6. There are no penalties for incorrect attempts.
7. Clicking the "**Submit**" button when it appears, means your submission is **FINAL**. You will **NOT** be able to resubmit your answer for that question ever again.
8. Check your grades in the course at any time by clicking on the "Progress" tab.

Module Introduction

In this module, you will learn about data visualization and some of the best practices to keep in mind when creating plots and visuals. You will also learn about the history and the architecture of Matplotlib and learn about basic plotting with Matplotlib. In addition, you will learn about the dataset on immigration to Canada, which will be used extensively throughout the course. Finally, you will briefly learn how to read CSV files into a pandas dataframe, process and manipulate the data in the dataframe, and generate line plots using Matplotlib.

Learning Objectives

In this lesson you will learn about:

- Describe the importance of data visualization
- Relate the history of Matplotlib and its architecture
- Apply Matplotlib to create plots using Jupyter notebooks
- Read CSV files into a Pandas DataFrame; process and manipulate the data in the DataFrame; and generate line plots using Matplotlib

Introduction to Data Visualization

First module of the data visualization with Python . We'll introduce **data visualization** and go over an example of transforming a given visual into one which is more effective attractive and impactive.

One might ask why would I need to learn how to visualize data well data visualization is a way to show a complex data in a form that is graphical and easy to understand. This can be especially useful when one is trying to explore the data and getting acquainted with it. Also since a picture is worth a thousand words then plots and graphs can be very effective in conveying a clear description of the data especially when disclosing findings to an audience or sharing the data with other pure data scientists. Also they can be very valuable when it comes to supporting any recommendations you make to clients managers or other decision-makers in your field.

Darkhorse analytics is a company that spun out of a research lab at the University of Alberta in 2008 and has done fascinating work on data visualization. Darkhorse analytics specializes in quantitative consulting in several areas including data visualization and geo spatial analysis.

Their approach when creating a visual revolves around **three key points**:

1. Less is more effective . 2. Less is more attractive and 3. Less is more impactive. In other words, **any feature or design you incorporate in your plot to make it more attractive or pleasing should support the message that the plot is meant to get across and not distract from it.**

Let's take a look at an example. Here is a **pie chart of what looks like people's preferences when it comes to different types of pig meat**. The charts message is almost half of the people surveyed preferred bacon over the other types of pig meat but I'm sure that almost all of you agree that there is a lot going on in this pie chart and we're not even sure it features such as the blue background or 3d orientation are meant to convey anything. In fact these additional unnecessary features distract from the main message and can be confusing to the audience.

Let's apply darkhorse analytics approach to transform this into a visual that's more effective attractive and impactive as I mentioned earlier the message here is that people are most likely to choose bacon over other types of pig meat so let's get rid of everything that can be distracting.

From this core message the first thing is let's get rid of the blue background and the gray background. let's also get rid of borders as they do not convey any extra information. also let's drop the redundant legend since the pie chart is already color coded 3d isn't adding any extra information so let's say bye to it.Text bolding is also unnecessary and let's get rid of the different colors and the wedges whoa what just happened. Well let's stick in the lines to make them more meaningful now this looks a little familiar yes this is a bar graph after all one with horizontal bars and finally let's emphasize bacon so that it stands out among the other types of pig meat.

Now let's just oppose the pie chart and the bar graph and compare which is better and easy to understand I hope that we anonymously agree that the bar graph is the better of the two. It is simple cleaner less distracting and much easier to read in fact pie charts have recently come under fire from data visualization experts who argue that they are relevant only in the rarest of circumstances **bar graphs and charts on the other hand are argued to be far superior ways to quickly get a message across** but don't worry about this for now we will come back to this point when we learn how to create pie charts and bar graphs with matplotlib for more similar and interesting examples check out darkhorse analytics website they have a couple more examples on how to clean bar graphs and maps of geospatial data all these examples reinforce the concept of less is more effective attractive and impactive

Data Visualization with Python

Introduction to Data Visualization

Why Build Visuals?

1. For exploratory data analysis
2. Communicate data clearly
3. Share unbiased representation of data
4. Use them to support recommendations to different stakeholders

Best Practices

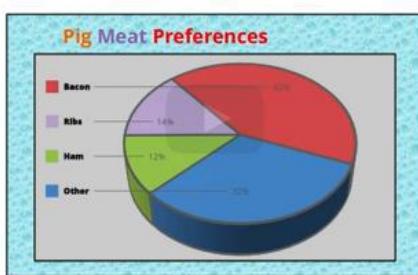
When creating a visual, always remember:

1. Less is more effective
2. Less is more attractive
3. Less is more impactful

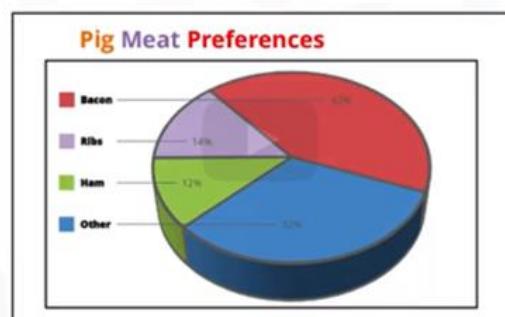


Three key points: less is more effective it is more attractive and it is more impactful. In other words any feature or design you incorporate in your plot to make it more attractive or pleasing should support the message that the plot is meant to get across and not distract from it.

Example

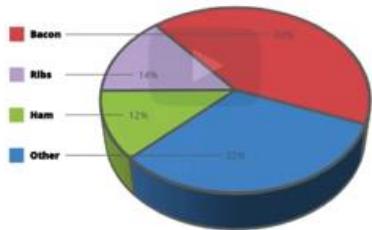


Example – Remove Background



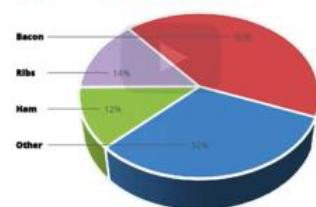
Example – Remove Borders

Pig Meat Preferences



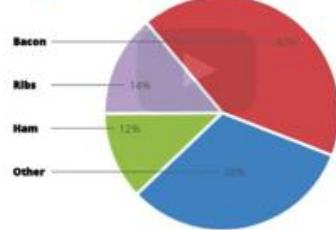
Example – Remove Redundant Legend

Pig Meat Preferences



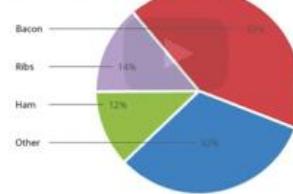
Example – Remove 3D

Pig Meat Preferences



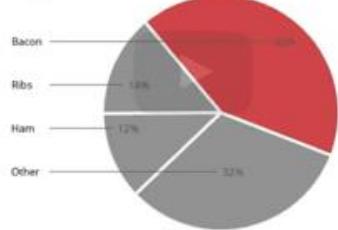
Example – Remove Text Bolding

Pig Meat Preferences



Example – Reduce Color

Pig Meat Preferences



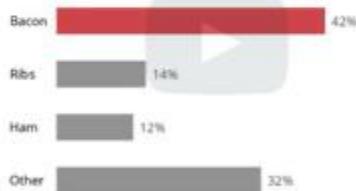
Example – Remove Wedges

Pig Meat Preferences



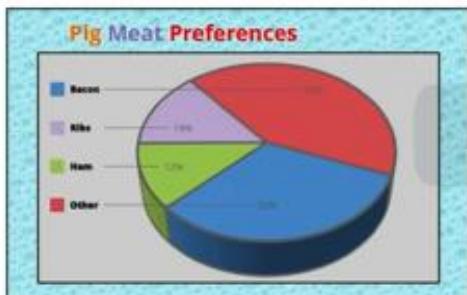
Example – Emphasize Bacon

Pig Meat Preferences



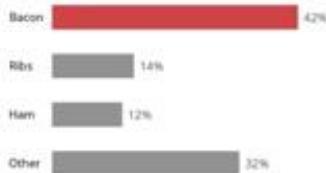
Comparison

Before



After

Pig Meat Preferences



The bar graph is the better of the two it is simple cleaner less distracting and much easier to read.

* bar graphs and charts are argued to be far superior ways to quickly get a message across

More Examples

For more examples, check out:

<http://www.darkhorseanalytics.com/>

Introduction to Matplotlib

In this video, we will start learning about Matplotlib. This video will focus on the history of Matplotlib and its architecture.

History:

Matplotlib is one of the most widely used, if not the most popular data visualization library in Python.

- It was created by John Hunter, who was a neurobiologist and was part of a research team that was working on analyzing Electrocorticography signals, ECoG for short. The team was using a proprietary software for the analysis. However, they had only one license and were taking turns in using it. So in order to overcome this limitation, John set out to replace the proprietary software with a MATLAB based version that could be utilized by him and his teammates, and that could be extended by multiple investigators. As a result, Matplotlib was originally developed as an ECoG visualization tool, and just like MATLAB.
- Matplotlib was equipped with a scripting interface for quick and easy generation of graphics, represented by pyplot. We will learn more about this in a moment. Now Matplotlib's architecture is composed of three main layers: the back-end layer, the artist layer where much of the heavy lifting happens and is usually the appropriate programming paradigm when writing a web application server, or a UI application, or perhaps a script to be shared with other developers, and the scripting layer, which is the appropriate layer for everyday purposes and is considered a lighter scripting interface to simplify common tasks and for a quick and easy generation of graphics and plots.

Now let's go into each layer in a little more details. So the back-end layer

- has three built-in abstract interface classes:
 - FigureCanvas, which defines and encompasses the area on which the figure is drawn.
 - Renderer, an instance of the renderer class knows how to draw on the figure canvas.
 - Event, which handles user inputs such as keyboard strokes and mouse clicks.

Moving on to the artist layer. It is composed of one main object, which is the artist. The artist is the object that knows how to take the Renderer and use it to put ink on the canvas. Everything you see on a Matplotlib figure is an artist instance. The title, the lines, the tick labels, the images, and so on, all correspond to an individual artist. There are two types of Artist objects.

- The first type is the primitive type, such as a line, a rectangle, a circle, or text. And
- the second type is the composite type, such as the figure or the axes. The top-level Matplotlib object that contains and manages all of the elements in a given graphic is the

Figure artist, and the most important composite artist is the axes because it is where most of the Matplotlib API plotting methods are defined, including methods to create and manipulate the ticks, the axis lines, the grid or the plot background.

Now it is important to note that each composite artist may contain other composite artists as well as primitive artists. So a figure artist for example would contain an axis artist as well as a rectangle or text artists. Now let's put the artist layer to use and see how we can use it to generate a graphic.

Example: let's try to generate a histogram of 10,000 random numbers using the artist layer. First we import the figure canvas from the backend backend underscore agg and attach the figure artist to it. Note that agg stands for anti grain geometry which is a high-performance library that produces attractive images.

Then we import the Numpy library to generate the random numbers. Next we create an axes artist. The axes artist is added automatically to the figure axes container, Fig.axes. And note here that (111) is from the MATLAB convention so it creates a grid with one row and one column and uses the first cell in that grid for the location of the new axes.

Then we call the axes method `hist`, to generate the histogram. `Hist` creates a sequence of rectangle artists for each histogram bar and adds them to the axes container. Here 100 means create 100 bins. Finally, we decorate the figure with a title and we save it.

Now this is the generated histogram and so this is how we use the artist layer to generate a graphic. As for the scripting layer, it was developed for scientists who are not professional programmers, and I'm sure you agree with me based on the histogram that we just created that the artist layer is syntactically heavy as it is meant for developers and not for individuals whose goal is to perform quick exploratory analysis of some data.

Matplotlib's scripting layer is essentially the Matplotlib.pyplot interface, which automates the process of defining a canvas and defining a figure artist instance and connecting them. So let's see how the same code that we used earlier using the artist layer to generate a histogram of 10,000 random numbers would now look like.

So first we import the pyplot interface and you can see how all the methods associated with creating the histogram and other artist objects and manipulating them whether it is the `hist` method or showing the figure are part of the pyplot interface. If you're interested in learning more about the history of Matplotlib and its architecture, this link will take you to a chapter written by the creators of Matplotlib themselves. It is definitely a recommended read.

Data Visualization with Python

Introduction to Matplotlib

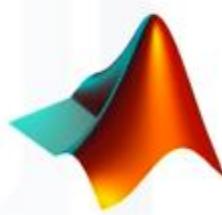
Matplotlib - History



John Hunter (1968 – 2012)



EEG/ECOG Visualization Tool



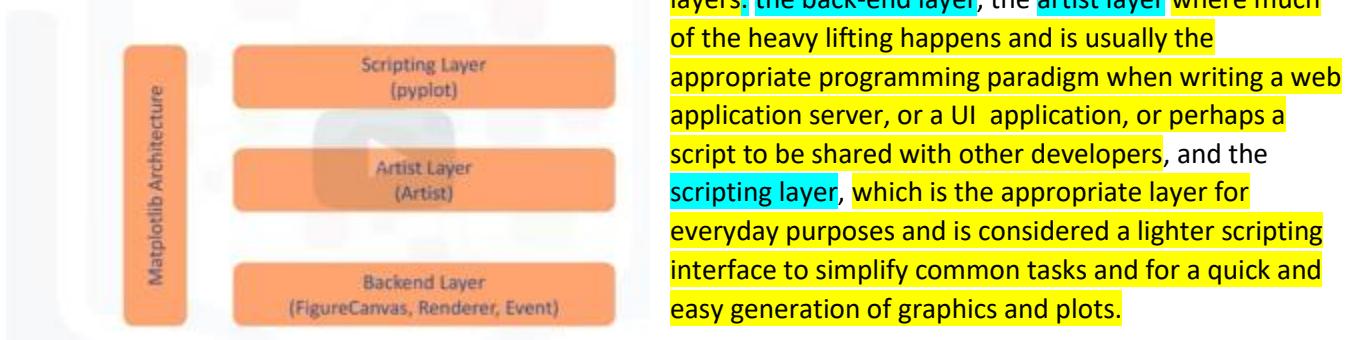
Analogous to Matlab scripting interface

It was created by John Hunter, who was a neurobiologist and was part of a research team that was working on analyzing Electrocorticography signals, ECoG for short. The team was using a proprietary software for the analysis. However they had only one license and were taking turns in using it. So in order to overcome this limitation, John set out to replace the

proprietary software with a MATLAB

based version that could be utilized by him and his teammates, and that could be extended by multiple investigators. As a result, Matplotlib was originally developed as an ECoG visualization tool, and just like MATLAB, Matplotlib was equipped with a scripting interface for quick and easy generation of graphics, represented by pyplot.

Matplotlib Architecture



Matplotlib's architecture is composed of three main layers: the back-end layer, the artist layer where much of the heavy lifting happens and is usually the appropriate programming paradigm when writing a web application server, or a UI application, or perhaps a script to be shared with other developers, and the scripting layer, which is the appropriate layer for everyday purposes and is considered a lighter scripting interface to simplify common tasks and for a quick and easy generation of graphics and plots.

Backend Layer

Has three built-in abstract interface classes:

1. FigureCanvas: **matplotlib.backend_bases.FigureCanvas**
 - Encompasses the area onto which the figure is drawn
2. Renderer: **matplotlib.backend_bases.Renderer**
 - Knows how to draw on the FigureCanvas
3. Event: **matplotlib.backend_bases.Event**
 - Handles user inputs such as keyboard strokes and mouse clicks

Artist Layer

- Comprised of one main object – **Artist**:
 - Knows how to use the Renderer to draw on the canvas.
- Title, lines, tick labels, and images, all correspond to individual **Artist** instances.
- Two types of **Artist** objects:
 1. **Primitive**: Line2D, Rectangle, Circle, and Text.
 2. **Composite**: Axis, Tick, Axes, and Figure
- Each *composite* artist may contain other *composite* artists as well as *primitive* artists.

Putting the Artist Layer to Use

- Let's try to generate a histogram of some data using the **Artist layer**:

```
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas # import FigureCanvas
from matplotlib.figure import Figure # import Figure artist
fig = Figure()
canvas = FigureCanvas(fig)

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

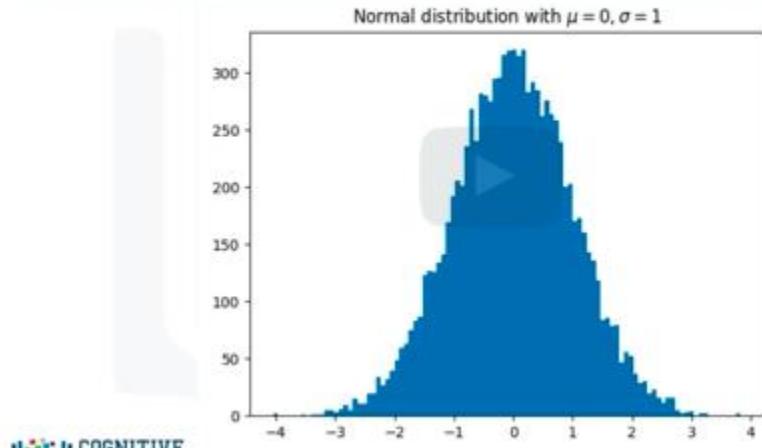
ax = fig.add_subplot(111) # create an axes artist
ax.hist(x, 100) # generate a histogram of the 10000 numbers

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

Let's try to generate a histogram of 10,000 random numbers using the artist layer. First we import the figure canvas from the **backend backend underscore agg** and attach the figure artist to it. Note that **agg** stands for anti grain geometry which is a high-performance library that produces attractive images. Then we import the Numpy library to generate the

random numbers. Next we create an axes artist. The axes artist is added automatically to the figure axes container, `Fig.axes`. And note here that (111) is from the MATLAB convention so it creates a grid with one row and one column and uses the first cell in that grid for the location of the new axes. Then we call the axes method `hist`, to generate the histogram. **Hist** creates a sequence of rectangle artists for each histogram bar and adds them to the axes container. Here 100 means create 100 bins. Finally, we decorate the figure with a title and we save it

Putting the Artist Layer to Use



Scripting Layer

- Comprised mainly of **pyplot**, a scripting interface that is lighter than the **Artist** layer.
- Let's see how we can generate the same histogram of 10000 random values using the **pyplot** interface.

```
import matplotlib.pyplot as plt
import numpy as np

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

Further Reading

You can find more information about Matplotlib: its history and architecture, by following the link below:

<http://www.aosabook.org/en/matplotlib.html>

Basic Plotting with Matplotlib

In this video, we will learn how to use Matplotlib to create plots, and we will do so using the Jupyter notebook as our environment.

Now Matplotlib is a well-established data visualization library that is well supported in different environments such as in Python scripts, in the iPython shell, web application servers, in graphical user interface toolkits as well as the Jupyter notebook.

Now for those of you who don't know what the Jupyter notebook is, it's an open source web application that allows you to create and share documents that contain live code visualizations and some explanatory text as well. Jupyter has some specialized support for Matplotlib and so if you start a Jupyter notebook, all you have to do is import Matplotlib and you're ready to go.

In this course, we will be working mostly with the scripting interface. In other words, we will learn how to create almost all of the visualization tools using the scripting interface. As we proceed in the course, you will appreciate the power of this interface when you find out that you can literally create almost all of the conventional visualization tools such as histograms, bar charts, box plots and many others using one function only:

the plot function. Let's start with an example. Let's first import the scripting interface as plt, and let's plot a circular mark at the position (5, 5), so x equals 5 and y equals 5. Notice how the plot was generated within the browser and not in a separate window for example. If the plot gets generated in a new window then you can enforce generating plots within the browser using what's called a magic function. A magic function starts with % Matplotlib, and to enforce plots to be rendered within the browser, you pass in inline as the backend.

Matplotlib has a number of different backends available. One limitation of this backend is that you cannot modify a figure once it's rendered. So after rendering the above figure, there is no way for us to add, for example, a figure title or label its axes. You will need to generate a new plot and add a title and the axes labels before calling the show function. A backend that overcomes this limitation is the notebook backend.

With the notebook backend in place, if a plt function is called, it checks if an active figure exists, and any functions you call will be applied to this active figure. If a figure does not exist, it renders a new figure. So when we call the plt.plot function to plot a circular mark at position (5, 5), the backend checks if an active figure exists. Since there isn't an active figure, it generates a figure and adds a circular mark to position (5, 5). And what is beautiful about this back end is that now we can easily add a title for example or labels to the axes after the plot was rendered, without the need to regenerate the figure.

Finally, another thing that is great about Matplotlib is that pandas also has a built-in implementation of it. Therefore, plotting in pandas is as simple as calling the plot function on a given pandas series or dataframe. So, say we have a data frame of the number of immigrants from India and China to Canada from 1980 to 1996 and say we're interested in generating a line plot of these data. All we have to do is call the plot function on this dataframe which we called India_China_df and set the parameter kind to line and there you have it: a line plot of the data in the data frame. Plotting a histogram of the data is not any different.

So say we would like to plot a histogram of the India column in our dataframe. All we have to do is call the plot function on that column and set the parameter kind to hist, for histogram. And there you have it. A histogram of the number of Indian immigrants to Canada from 1980 to 1996. This concludes our video on basic plotting with Matplotlib.

Data Visualization with Python

Basic Plotting with Matplotlib

Matplotlib – Jupyter Notebook



Matplotlib is a well-established data visualization library that is well supported in different environments such as in Python scripts, in the iPython shell, web application servers, in graphical user interface toolkits as well as the Jupyter notebook. It's an open source web application that allows you to create and share documents that contain live code visualizations and some explanatory text as well. Jupyter has some specialized support for Matplotlib and so if you start a Jupyter notebook, all you have to do is import Matplotlib.

Matplotlib – Jupyter Notebook

A screenshot of a Jupyter Notebook interface. The top bar shows "jupyter Matplotlib - Jupyter Notebook Last Checkpoint: 3 minutes ago (unsaved changes)". The menu includes File, Edit, View, Insert, Cell, Kernel, Help. The toolbar includes CellToolbar. The main area shows two code cells:

```
In [1]: import matplotlib as mpl  
In [2]: print('matplotlib version: ', mpl.__version__)  
'matplotlib version: ', '2.0.0'
```

The second cell's output is displayed below it.

1.

Matplotlib – Plot Function

A screenshot of a Jupyter Notebook interface. The top bar shows "jupyter Matplotlib - Jupyter Notebook Last Checkpoint: 39 minutes ago (unsaved changes)". The menu includes File, Edit, View, Insert, Cell, Kernel, Help. The toolbar includes CellToolbar. The main area shows two code cells:

```
In [1]: import matplotlib.pyplot as plt  
In [2]: plt.plot(5, 5, 'o')  
plt.show()
```

The second cell's output is a scatter plot with a single blue dot at the coordinates (5, 5) on a white background with axes from 4.0 to 6.0.

Let's first import the scripting interface as plt, and let's plot a circular mark at the position (5, 5), so x equals 5 and y equals 5. Notice how the plot was generated within the browser and not in a separate window for example.

Matplotlib Backends - Inline

A screenshot of a Jupyter Notebook interface. The top menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with various icons. The code cell contains two lines of Python code:

```
In [1]: import matplotlib.pyplot as plt  
In [2]: plt.plot(5, 5, 'o')
```

The plot window shows a single blue circular marker at the coordinates (5, 5) on a grid from 4.8 to 5.2 on both axes.

If the plot gets generated in a new window then you can enforce generating plots within the browser using what's called a **magic function**. A magic function starts with `% Matplotlib`, and to enforce plots to be rendered within the browser, you pass in `inline` as the backend. Matplotlib has a number of different backends available. One limitation of this backend is that you cannot modify a figure once it's rendered. So after rendering the above figure, there is no way for us to add, for example, a figure title or label its axes.

Matplotlib Backends - Inline

A screenshot of a Jupyter Notebook interface. The top menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with various icons. The code cell contains two lines of Python code:

```
In [1]: import matplotlib.pyplot as plt  
In [2]: plt.plot(5, 5, 'o')  
plt.show()
```

The plot window shows a single blue circular marker at the coordinates (5, 5) on a grid from 4.8 to 5.2 on both axes. To the right of the plot, the output cell shows:

```
In [3]: plt.ylabel("Y")  
plt.xlabel("X")  
Out[3]: <matplotlib.text.Text at 0x110480010>
```

You will need to generate a new plot and add a title and the axes labels before calling the `show` function.

Matplotlib Backends - Notebook

A screenshot of a Jupyter Notebook interface. The top menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with various icons. The code cell contains two lines of Python code:

```
In [1]: import matplotlib.pyplot as plt  
In [2]: plt.plot(5, 5, 'o')
```

The plot window shows a single blue circular marker at the coordinates (5, 5) on a grid from 4.8 to 5.2 on both axes. Below the plot, a message indicates:

```
Figure 1
```

To the right of the plot, the output cell shows:

```
Out[2]: <matplotlib.lines.Line2D at 0x110794c790>
```

A backend that overcomes this limitation is the **notebook backend**. With the notebook backend in place, if a `plt` function is called, it checks if an active figure exists, and any functions you call will be applied to this active figure. If a figure does not exist, it renders a new figure. So when we call the `plt.plot` function to plot a circular mark at position (5, 5), the backend checks if an active figure exists. Since there isn't an active figure, it generates a figure and adds a circular mark to position (5, 5).

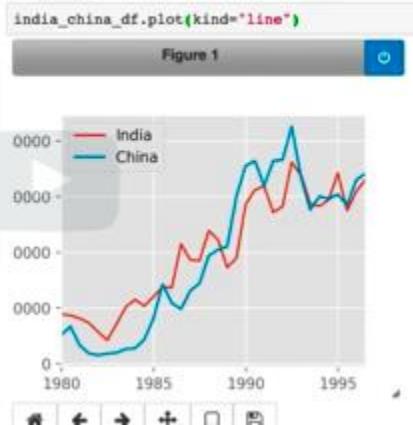
Matplotlib Backends – Notebook



And what is beautiful about this back end is that now we can easily add a title for example or labels to the axes after the plot was rendered, without the need to regenerate the figure.

Matplotlib – Pandas

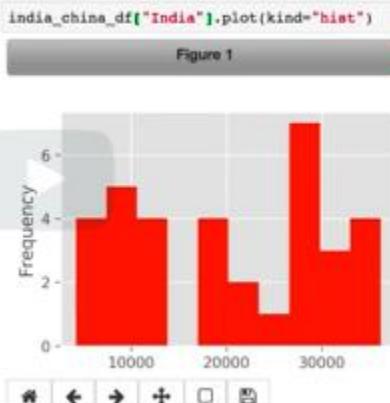
	India	China
1980	8880	5123
1981	8670	6682
1982	8147	3308
1983	7338	1863
1984	5704	1527



Matplotlib is that pandas also has a built-in implementation of it. Therefore, plotting in pandas is as simple as calling the plot function on a given pandas series or dataframe.

Matplotlib – Pandas

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



Dataset on Immigration to Canada

We will learn more about the dataset that we will be using throughout the course. The population division of the United Nations compiled immigration data pertaining to 45 countries. The data consist of the total number of immigrants from all over the world to each of the 45 countries as well as other metadata pertaining to the immigrants countries of origin.

In this course, we will focus on immigration to Canada and we will work primarily with the data set involving immigration to the great white north. Here is a snapshot of the UN data on immigration to Canada in the form of an excel file. As you can see, the first twenty rows contain textual data about the UN Department and other irrelevant information. Row 21 contains the labels of the columns. Following that each row represents a country and contains metadata about the country such as what continent it resides in, what region it belongs to, and whether the region is developing or developed. Each row also contains the total number of immigrants from that country for the years 1980 all the way to 2013.

Throughout this course, we will be using pandas for any analysis of the data before creating any visualizations. So in order to start creating different types of plots of the data, whether for exploratory analysis or for presentation, we will need to import the data into a pandas dataframe. To do that, we will need to import the pandas library as well as the xlrd library, which is required to extract data from Excel spreadsheet files. Then we call the pandas function read_excel to read the data into a pandas dataframe. And let's name this dataframe df_can. Notice how we're skipping the first twenty rows to read only the data corresponding to each country.

If you want to confirm that you have imported your data correctly, in pandas, you can always use the head function to display the first five rows of the dataframe. So, if we call this function on our dataframe, df_can, here is the output. As you can see, the output of the head function looks correct with the columns having the correct labels and each row representing a country and containing the total number of immigrants from that country. This concludes our video on the immigration to Canada dataset.

Data Visualization with Python

Dataset on Immigration to Canada

Dataset

- The Population Division of the United Nations compiled data pertaining to 45 countries.
- For each country, annual data on the flows of international migrants is reported in addition to other metadata.
- We will primarily work with a United Nations data on immigration to Canada.

Immigration Data to Canada

The screenshot shows a portion of an Excel spreadsheet from the United Nations Population Division. The title bar includes the UN logo and the text "United Nations Population Division Department of Economic and Social Affairs". Below the title, it says "International Migration Flows to and from Selected Countries: The 2015 Revision" and "POP/DB/MIG/FlowRev/2015". A note at the bottom states "December 2015 - Copyright © 2015 by United Nations. All rights reserved" and "Suggested citation: United Nations, Department of Economic and Social Affairs, Population Division (2015). International Migration Flows to and from Selected Countries: The 2015 Revision. (United Nations database)." The data starts with a header row and several rows of data for immigrants to Canada by citizenship and origin.

Classification	Coverage	Origin/Destination	Major area	REG	Region	Development region	DEV	DevName	1980	1981	1982	1983	1984	1985
Immigrants	Foreigners	Afghanistan	935 Asia	5501	Southern Asia	902	Developing regions	16	39	39	47	71	340	
Immigrants	Foreigners	Albania	908 Europe	925	Southern Europe	901	Developed regions	1	0	0	0	0	0	
Immigrants	Foreigners	Algeria	903 Africa	912	Northern Africa	902	Developing regions	80	67	71	69	63	44	
Immigrants	Foreigners	American Samoa	909 Oceania	957	Polynesia	902	Developing regions	0	1	0	0	0	0	
Immigrants	Environment	Andorra	one Europe	926	Northern Europe	901	Developed regions	n	n	n	n	n	n	

Read Data into Pandas Dataframe

```
import numpy as np # useful for many scientific computing in Python
import pandas as pd # primary data structure library
from __future__ import print_function # adds compatibility to python 2
```

```
# install xlrd
!pip install xlrd

print('xlrd installed!')
```

```
df_can = pd.read_excel(
    'https://ibm.box.com/shared/static/lwl90pt9zpy5bd1ptyg2aw15awomz9pu.xlsx',
    sheetname='Canada by Citizenship',
    skiprows=range(20),
    skip_footer = 2)
```

We will need to import the data into a pandas dataframe. To do that, we will need to import the pandas library as well as the **xlrd library**, which is required to extract data from Excel spreadsheet files. Then we call the pandas function `read_excel` to read the data into a pandas dataframe. And let's name this dataframe `df_can`. Notice how we're skipping the first twenty rows to read only the data corresponding to each country.

Display Dataframe

```
df_can.head()
```

The screenshot shows the first five rows of a Pandas DataFrame named `df_can`. The columns are labeled: Type, Coverage, OdName, AREA, AreaName, REG, RegName, DEV, DevName, 1980, ..., 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013. The data consists of 15 rows of immigrant counts for various countries and regions.

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Immigrants	Foreigners	Afghanistan	935 Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1748	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908 Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903 Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909 Oceania	957	Polynesia	902	Developing regions	0	...	0	1	0	0	0	0	0	0	0	0
4	Immigrants	Foreigners	Andorra	one Europe	926	Northern Europe	901	Developed regions	0	...	0	1	1	0	0	0	0	1	1	1

To confirm that you have imported your data correctly, in pandas, you can always use the `head` function to display the first five rows of the dataframe. So, if we call this function on our dataframe, `df_can`, here is the output. As you can see, the output of the `head` function looks correct with the columns having the correct labels and each row representing a country and containing the total number of immigrants from that country.

Line Plots



In this video, things will start getting more exciting. We will generate our first visualization tool: the **line plot**. So what is a line plot? As its name suggests, it is a plot in the form of a series of data points connected by straight line segments. It is one of the most basic type of chart and is common in many fields not just data science. The more important question is when to use line plots. The best use case for a line plot is when you have a continuous dataset and you're interested in visualizing the data over a period of time.

As an example, say we're interested in the trend of immigrants from Haiti to Canada. We can generate a line plot and the resulting figure will depict the trend of Haitian immigrants to Canada from 1980 to 2013. Based on this line plot, we can then research for justifications of obvious anomalies or changes. So in this example, we see that there is a spike of immigration from Haiti to Canada in 2010.

A quick Google search for major events in Haiti in 2010 would return the tragic earthquake that took place in 2010, and therefore this influx of immigration to Canada was mainly due to that tragic earthquake. Okay, now, how can we generate this line plot? Before we go over the code to do that,

let's do a quick recap of our dataset. Each row represents a country and contains metadata about the country such as where it is located geographically, and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

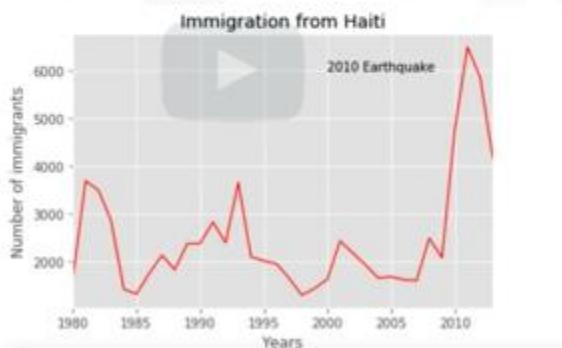
Now let's process the dataframe so that the country name becomes the index of each row. This should make querying specific countries easier. Also let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan, it is 58,639, total, and for Albania it is 15,699, and so on.

And let's name our dataframe `df_canada`. So now that we know how our data is stored in the dataframe, `df_canada`, let's generate the line plot corresponding to immigration from Haiti. First, we import Matplotlib as `mpl` and its scripting interface as `plt`. Then, we call the `plot` function on the row corresponding to Haiti, and we set `kind` equals `line` to generate a line plot. Note that we used `years` which is a list containing string format of years from 1980 to 2013 in order to exclude the column of total immigration that we added. Then to complete the figure, we give it a title, and we label its axes.

Finally, we call the `show` function to display the figure. Note that this is the code to generate the line plot using the magic function `%matplotlib` with the inline backend. And there you have it: a line plot that depicts immigration from Haiti to Canada from 1980 to 2013.

Line Plots

A line plot is a type of plot which displays information as a series of data points called 'markers' connected by straight line segments.



Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REQ	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

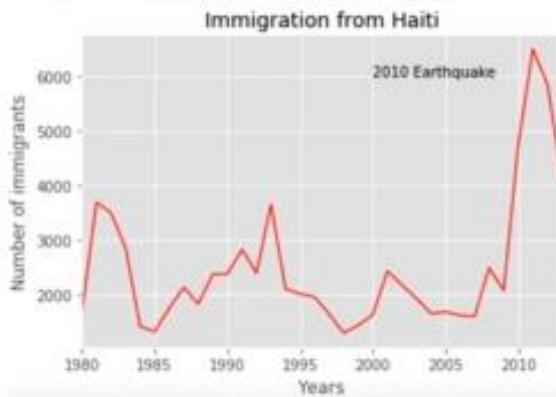
Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

Creating Line Plots

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

years = list(map(str, range(1980, 2014)))
df_canada.loc['Haiti', years].plot(kind = 'line')
plt.title('Immigration from Haiti')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')
plt.show()
```



of total immigration that we added. Then to complete the figure, we give it a title, and we label its axes. Finally, we call the show function to display the figure. Note that this is the code to generate the line plot using the magic function %matplotlib with the inline backend.

First, we import Matplotlib as mpl and its scripting interface as plt. Then, we call the plot function on the row corresponding to Haiti, and we set kind equals line to generate a line plot. Note that we used years which is a list containing string format of years from 1980 to 2013 in order to exclude the column

8.1.Introduction to Data Visualization Tools

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

Learning Objectives

- Describe the importance of data visualization
 - Relate the history of Matplotlib and its architecture
 - Apply Matplotlib to create plots using Jupyter notebooks
 - Read csv files into a Pandas DataFrame; process and manipulate the data in the DataFrame; and generate line plots using Matplotlib
-
- [Introduction to Data Visualization](#)
 - [Lab 1: Introduction to Matplotlib and Line Plots](#)

<https://www.coursera.org/professional-certificates/ibm-data-science>

© 2021 Coursera Inc. All rights reserved.

8.1.1.Introduction to Data Visualization

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Introduction to Data Visualization Tools

LATEST SUBMISSION GRADE 100%

Question 1

Matplotlib was created by John Hunter, an American neurobiologist, and was originally developed as an EEG/ECoG visualization tool.

- True.
- False.

Correct

Question 2

Using the inline backend, you can modify a figure after it is rendered.

- True.
- False.

Correct. You CANNOT modify a figure after it is rendered using the inline backend.

Question 3

`%matplotlib inline` is an example of Matplotlib magic functions.

- True.
- False.

Correct. A sign of a magic function is that it starts with a "%matplotlib" and inline is a Matplotlib backend

Data Visualization

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Create Data Visualization with Python
- Use various Python libraries for visualization

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 Basics for Data Science](#) and [Analyzing Data with Python](#).

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 [Analyzing Data with Python](#).

Table of Contents

1. [Exploring Datasets with *pandas*](#0)
1.1 [The Dataset: Immigration to Canada from 1980 to 2013](#2)
1.2 [*pandas* Basics](#4)
1.3 [*pandas* Intermediate: Indexing and Selection](#6)
2\. [Visualizing Data using Matplotlib](#8)
2.1 [Matplotlib: Standard Python Visualization Library](#10)
3\. [Line Plots](#12)

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>.

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.

Reporting country	Classification	Type	Coverage	International Migration Flows to and from Selected Countries: The 2015 Revision																				
				1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996				
Afghanistan	Residence	Immigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—				
Australia	Residence	Immigrants	Both	90860	85600	92340	100510	96360	93440	92490	97770	104770	120040	137470	143710	143660	140420	141680	149360	198260	176640			
Australia	Residence	Immigrants	Both	194290	212680	195200	153370	153330	172550	196680	221620	253860	238050	234060	237240	220460	187940	221920	25940	281330	266220			
Austria	Citizenship	Emigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Austria	Citizenship	Emigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	46725	48244		
Austria	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	12830	13227	
Austria	Citizenship	Immigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	50035	49636	
Austria	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—		
Austria	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	69830	79122	
Azerbaijan	Residential	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	16033	13151	15703
Azerbaijan	Residence	Immigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	6222	5781	7528
Belarus	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—		
Belgium	Citizenship	Emigrants	Citizens	13326	20325	21497	21090	20582	20481	21110	22253	16244	18079	15937	1802	13256	13616	14422	14442	16384	18250			
Belgium	Citizenship	Emigrants	Foreigners	36887	36970	37207	36170	32747	30431	29509	31017	28861	24737	24375	31617	24697	29412	32462	31745	30616	32710			
Belgium	Citizenship	Immigrants	Citizens	7834	7979	8479	8310	9843	9500	9663	9655	10253	10620	12193	13330	11713	10707	10182	9812	9638	9609			
Belgium	Citizenship	Immigrants	Foreigners	39746	33907	29498	28477	29854	28659	29466	31468	35084	39338	41783	4312	48344	47716	45614	45614	45614	45614			
Belgium	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Belgium	Residence	Emigrants	Both	54694	49298	44609	42657	47032	47042	48939	49750	48484	54189	62662	67490	66763	63749	—	—	—	—			
Bulgaria	Citizenship	Emigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Bulgaria	Citizenship	Emigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Bulgaria	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Bulgaria	Citizenship	Immigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Bulgaria	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Bulgaria	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Canada	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Canada	Citizenship	Immigrants	Foreigners	143137	128641	121175	89186	88272	84346	99351	152075	161585	191550	216448	232799	254783	2546635	224381	212863	228070	216636			
Croatia	Citizenship	Emigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Croatia	Citizenship	Emigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Croatia	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Croatia	Citizenship	Immigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Croatia	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Croatia	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Citizenship	Emigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Citizenship	Emigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Citizenship	Immigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Cyprus	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Citizenship	Emigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Citizenship	Emigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Citizenship	Immigrants	Citizens	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Citizenship	Immigrants	Foreigners	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Czech Republic	Residence	Emigrants	Both	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
Denmark	Citizenship	Emigrants	Citizens	17979	18640	17991	16849	16893	17692	18665	19981	23893	25447	25623	22167	22657	22510	23819	23627	24355	24355			
Denmark	Citizenship	Emigrants	Foreigners	11845	15377	13814	8333	8171	3275	10268	12455	8073	8645	10185	9081	2814	10891	10898	10898	10898	10898			
Denmark	Citizenship	Immigrants	Citizens	14526	14513	15295	15858	15742	16112	18389	16239	14605	19180	21050	21445	21893	22021	23984	24041	22918	22918			
Denmark	Citizenship	Immigrants	Foreigners	15282	12982	12606	11433	12900	16219	20592	18217	16756	16906	17739	19744	18639	19623	20460	28238	28914	26953			

The Canada Immigration dataset can be fetched from [here](#).

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

```
! pip3 install openpyxl
```

Now we are ready to read in our data.

```
In [2]:  
df_can = pd.read_excel(  
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',  
    sheet_name='Canada by Citizenship',  
    skiprows=range(20),  
    skipfooter=2)
```

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 follows: df_can.head(10)
```

```
Out[3]:  
      Type Coverage   OdName AREA AreaName REG  RegName  DEV DevName 1980 ... 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013  
0 Immigrants Foreigners Afghanistan 935 Asia 5501 Southern Asia 902 Developing regions 16 ... 2978 3436 3009 2652 2111 1746 1758 2203 2635 2004  
1 Immigrants Foreigners Albania 908 Europe 925 Southern Europe 901 Developed regions 1 ... 1450 1223 856 702 560 716 561 539 620 603  
2 Immigrants Foreigners Algeria 903 Africa 912 Northern Africa 902 Developing regions 80 ... 3616 3626 4807 3623 4005 5393 4752 4325 3774 4331  
3 Immigrants Foreigners American Samoa 909 Oceania 957 Polynesia 902 Developing regions 0 ... 0 0 1 0 0 0 0 0 0 0 0  
4 Immigrants Foreigners Andorra 908 Europe 925 Southern Europe 901 Developed regions 0 ... 0 0 1 1 0 0 0 0 0 1 1
```

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	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
190	Immigrants	Foreigners	Viet Nam	935	Asia	920	South-Eastern Asia	902	Developing regions	1191	...	1816	1852	3153	2574	1784	2171	1942	1723	1731	2112
191	Immigrants	Foreigners	Western Sahara	903	Africa	912	Northern Africa	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	0
192	Immigrants	Foreigners	Yemen	935	Asia	922	Western Asia	902	Developing regions	1	...	124	161	140	122	133	128	211	160	174	217
193	Immigrants	Foreigners	Zambia	903	Africa	910	Eastern Africa	902	Developing regions	11	...	56	91	77	71	64	60	102	69	46	59
194	Immigrants	Foreigners	Zimbabwe	903	Africa	910	Eastern Africa	902	Developing regions	72	...	1450	615	454	663	611	508	494	434	437	407

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.

This method can be used to get a short summary of the dataframe.

In [10]:

```
df_can.info(verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 195 entries, 0 to 194
Columns: 43 entries, Type to 2013
dtypes: int64(37), object(6)
memory usage: 65.6+ KB
```

To get the list of column headers we can call upon the data frame's `columns` instance variable.

In [11]:

```
df_can.columns
```

Out[11]:

```
Index(['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` instance variables.

In [12]:

```
df_can.index
```

Out[12]:

```
RangeIndex(start=0, stop=195, step=1)
```

Note: The default type of instance variables `index` and `columns` are **NOT** `list`.

```
In [13]: 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 [14]: df_can.columns.tolist()
```

```
Out[14]: ['Type',
 'Coverage',
 'OdmName',
 'AREA',
 'AreaName',
 'REQ',
 '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]
```

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

```
Out[15]: [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]
```

```
In [16]: 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` instance variable of it.

```
In [17]: # size of dataframe (rows, columns)
df_can.shape
```

```
Out[17]: (195, 43)
```

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

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

```
In [18]: # 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[18]:
```

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	1223	856	702	560	716	561	539	620	603

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

```
Out[19]: Index(['Country', 'Continent', 'Region', 'DevName',  
               1980,  
               1981, 1982, 1983, 1984, 1985,  
               1986, 1987, 1988, 1989, 1990,  
               1991, 1992, 1993, 1994, 1995,  
               1996, 1997, 1998, 1999, 2000,  
               2001, 2002, 2003, 2004, 2005,  
               2006, 2007, 2008, 2009, 2010,  
               2011, 2012, 2013],  
              dtype='object')
```

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

```
In [20]: 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 [21]: df_can.isnull().sum()
```

```
Out[21]: 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 [22]: df_can.describe()
```

```
Out[22]:
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2005
count	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	...	195.000000
mean	508.394872	566.989744	534.723077	387.435897	376.497436	358.861538	441.271795	691.133333	714.389744	843.241026	...	1320.292308
std	1949.588546	2152.643752	1866.997511	1204.333597	1198.246371	1079.309600	1225.576630	2109.205607	2443.606788	2555.048874	...	4425.957828
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500000	0.500000	1.000000	1.000000	...	28.500000
50%	13.000000	10.000000	11.000000	12.000000	13.000000	17.000000	18.000000	26.000000	34.000000	44.000000	...	210.000000
75%	251.500000	295.500000	275.000000	173.000000	181.000000	197.000000	254.000000	434.000000	409.000000	508.500000	...	832.000000
max	22045.000000	24796.000000	20620.000000	10015.000000	10170.000000	9564.000000	9470.000000	21337.000000	27359.000000	23795.000000	...	42584.000000

8 rows × 35 columns

2006	2007	2008	2009	2010	2011	2012	2013	Total
195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000	195.000000
1266.958974	1191.820513	1246.394872	1275.733333	1420.287179	1262.533333	1313.958974	1320.702564	32867.451282
3926.717747	3443.542409	3694.573544	3829.630424	4462.946328	4030.084313	4247.555161	4237.951988	91785.498686
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
25.000000	31.000000	31.000000	36.000000	40.500000	37.500000	42.500000	45.000000	952.000000
218.000000	198.000000	205.000000	214.000000	211.000000	179.000000	233.000000	213.000000	5018.000000
842.000000	899.000000	934.500000	888.000000	932.000000	772.000000	783.000000	796.000000	22239.500000
33848.000000	28742.000000	30037.000000	29622.000000	38617.000000	36765.000000	34315.000000	34129.000000	691904.000000

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

```
Out[23]: 0      Afghanistan
1      Albania
2      Algeria
3      American Samoa
4      Andorra
...
190      Viet Nam
191      Western Sahara
192      Yemen
193      Zambia
194      Zimbabwe
Name: Country, Length: 195, dtype: object
```

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

```
In [24]: df_can[['Country', 1980, 1981, 1982, 1983, 1984, 1985]] # returns a dataframe
# notice that 'Country' is string, and the years are integers.
# for the sake of consistency, we will convert all column names to string later on.
```

```
Out[24]:    Country  1980  1981  1982  1983  1984  1985
0      Afghanistan  16   39   39   47   71  340
1      Albania      1    0    0    0    0    0
2      Algeria     80   67   71   69   63   44
3      American Samoa  0    1    0    0    0    0
4      Andorra      0    0    0    0    0    0
...
190      Viet Nam  1191  1829  2162  3404  7583  5907
191      Western Sahara  0    0    0    0    0    0
192      Yemen       1    2    1    6    0    18
193      Zambia      11   17   11    7   16    9
194      Zimbabwe    72  114  102   44   32   29
```

195 rows × 7 columns

Select Row

There are main **2 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 [25]: 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 [26]: df_can.head(3)
```

```
Out[26]:
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439

3 rows × 38 columns

```
In [27]: # 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 [28]: # 1. the full row data (all columns)
df_can.loc['Japan']
```

```
Out[28]: Continent      Asia
Region        Eastern Asia
DevName   Developed regions
1980            701
1981            756
1982            598
1983            309
1984            246
1985            198
1986            248
1987            422
1988            324
1989            494
1990            379
1991            506
1992            605
1993            907
1994            956
1995            826
1996            994
1997            924
1998            897
1999            1083
2000            1010
2001            1092
2002            806
2003            817
2004            973
2005            1067
2006            1212
2007            1250
2008            1284
2009            1194
2010            1168
2011            1265
2012            1214
2013            982
Total          27707
Name: Japan, dtype: object
```

```
In [29]: # alternate methods
df_can.iloc[87]
```

```
Out[29]:
```

	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
	Asia	Eastern Asia	Developed regions	701	756	598	309	246	198	248	422	324	494	379	506	605	907	956	826	994	924	897	1083	1010	1092	806	817	973	1067	1212	1250	1284	1194	1168	1265	1214	982	27707
																																					Name: Japan, dtype: object	

```
In [30]:
```

```
df_can[df_can.index == 'Japan']
```

```
Out[30]:
```

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Japan	Asia	Eastern Asia	Developed regions	701	756	598	309	246	198	248	...	1067	1212	1250	1284	1194	1168	1265	1214	982	27707

1 rows × 38 columns

```
In [31]:
```

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

```
Out[31]:
```

```
982
```

```
In [32]: # alternate method  
# year 2013 is the last column, with a positional index of 36  
df_can.iloc[87, 36]
```

```
Out[32]: 982
```

```
In [33]: # 3. for years 1980 to 1985  
df_can.loc['Japan', [1980, 1981, 1982, 1983, 1984, 1985]]
```

```
Out[33]: 1980    701  
1981    756  
1982    598  
1983    309  
1984    246  
1984    246  
Name: Japan, dtype: object
```

```
In [34]: # Alternative Method  
df_can.iloc[87, [3, 4, 5, 6, 7, 8]]
```

```
Out[34]: 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 with the 2013th positional index.

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

```
In [38]: 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
```

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

```
Out[38]: [None,  
None,  
None]
```

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

```
Out[39]: ['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 [40]: # 1. create the condition boolean series
condition = df_can['Continent'] == 'Asia'
print(condition)
```

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

```
In [41]: # 2. pass this condition into the DataFrame
df_can[condition]
```

Out[41]:		Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
	Armenia	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	224	218	198	205	267	252	236	258	207	3310
	Azerbaijan	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	359	236	203	125	165	209	138	161	57	2649
	Bahrain	Asia	Western Asia	Developing regions	0	2	1	1	1	3	0	...	12	12	22	9	35	28	21	39	32	475
	Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	4171	4014	2897	2939	2104	4721	2694	2640	3789	65568
	Bhutan	Asia	Southern Asia	Developing regions	0	0	0	0	1	0	0	...	5	10	7	36	865	1464	1879	1075	487	5876
	Brunei Darussalam	Asia	South-Eastern Asia	Developing regions	79	6	8	2	2	4	12	...	4	5	11	10	5	12	6	3	6	600
	Cambodia	Asia	South-Eastern Asia	Developing regions	12	19	26	33	10	7	8	...	370	529	460	354	203	200	196	233	288	6538
	China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1960	...	42584	33518	27642	30037	29622	30391	28502	33024	34129	659962
	China, Hong Kong Special Administrative Region	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	729	712	674	897	657	623	591	728	774	9327
	China, Macao Special Administrative Region	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	21	32	16	12	21	21	13	33	29	284
	Cyprus	Asia	Western Asia	Developing regions	132	128	84	46	46	43	48	...	7	9	4	7	6	18	6	12	16	1126
	Democratic People's Republic of Korea	Asia	Eastern Asia	Developing regions	1	1	3	1	4	3	0	...	14	10	7	19	11	45	97	66	17	388
	Georgia	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	114	125	132	112	128	126	139	147	125	2068
	India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210	33848	28742	28261	29456	34235	27509	30933	33087	691904
	Indonesia	Asia	South-Eastern Asia	Developing regions	186	178	252	115	123	100	127	...	632	613	657	661	504	712	390	395	387	13150
	Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	5837	7480	6974	6475	6580	7477	7479	7534	11291	175923
	Iraq	Asia	Western Asia	Developing regions	262	245	260	380	428	231	265	...	2226	1788	2406	3543	5450	5941	6196	4041	4918	69789
	Israel	Asia	Western Asia	Developing regions	1403	1711	1334	541	446	680	1212	...	2446	2625	2401	2562	2316	2755	1970	2134	1945	66508
	Japan	Asia	Eastern Asia	Developed regions	701	756	598	309	246	198	248	...	1067	1212	1250	1284	1194	1168	1265	1214	982	27707

Jordan	Asia	Western Asia	Developing regions	177	160	155	113	102	179	181	...	1940	1827	1421	1581	1235	1831	1635	1206	1255	35406
Kazakhstan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	506	408	436	394	431	377	381	462	348	8490
Kuwait	Asia	Western Asia	Developing regions	1	0	8	2	1	4	4	...	66	35	62	53	68	67	58	73	48	2025
Kyrgyzstan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	173	161	135	168	173	157	159	278	123	2353
Lao People's Democratic Republic	Asia	South-Eastern Asia	Developing regions	11	6	16	16	7	17	21	...	42	74	53	32	39	54	22	25	15	1089
Lebanon	Asia	Western Asia	Developing regions	1409	1119	1159	789	1253	1683	2576	...	3709	3802	3467	3566	3077	3432	3072	1614	2172	115359
Malaysia	Asia	South-Eastern Asia	Developing regions	786	816	813	448	384	374	425	...	593	580	600	658	640	802	409	358	204	24417
Maldives	Asia	Southern Asia	Developing regions	0	0	0	1	0	0	0	...	0	0	2	1	7	4	3	1	1	30
Mongolia	Asia	Eastern Asia	Developing regions	0	0	0	0	0	0	0	...	59	64	82	59	118	169	103	68	99	952
Myanmar	Asia	South-Eastern Asia	Developing regions	80	62	46	31	41	23	18	...	210	953	1887	975	1153	556	368	193	262	9245
Nepal	Asia	Southern Asia	Developing regions	1	1	6	1	2	4	13	...	607	540	511	581	561	1392	1129	1185	1308	10222
Oman	Asia	Western Asia	Developing regions	0	0	0	8	0	0	0	...	14	18	16	10	7	14	10	13	11	224
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314	13127	10124	8994	7217	6811	7468	11227	12603	241600

Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	18139	18400	19837	24887	28573	38617	36765	34315	29544	511391
Qatar	Asia	Western Asia	Developing regions	0	0	0	0	0	0	1	...	11	2	5	9	6	18	3	14	6	157
Republic of Korea	Asia	Eastern Asia	Developing regions	1011	1456	1572	1081	847	962	1208	...	5832	6215	5920	7294	5874	5537	4588	5316	4509	142581
Saudi Arabia	Asia	Western Asia	Developing regions	0	0	1	4	1	2	5	...	198	252	188	249	246	330	278	286	267	3425
Singapore	Asia	South-Eastern Asia	Developing regions	241	301	337	169	128	139	205	...	392	298	690	734	366	805	219	146	141	14579
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	4930	4714	4123	4756	4547	4422	3309	3338	2394	148358
State of Palestine	Asia	Western Asia	Developing regions	0	0	0	0	0	0	0	...	453	627	441	481	400	654	555	533	462	6512
Syrian Arab Republic	Asia	Western Asia	Developing regions	315	419	409	269	264	385	493	...	1458	1145	1056	919	917	1039	1005	650	1009	31485
Tajikistan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	85	46	44	15	50	52	47	34	39	503
Thailand	Asia	South-Eastern Asia	Developing regions	56	53	113	65	82	66	78	...	575	500	487	519	512	499	396	296	400	9174
Turkey	Asia	Western Asia	Developing regions	481	874	706	280	338	202	257	...	2065	1638	1463	1122	1238	1492	1257	1068	729	31781
Turkmenistan	Asia	Central Asia	Developing	0	0	0	0	0	0	0	...	40	26	37	13	20	30	20	20	14	310
United Arab Emirates	Asia	Western Asia	Developing regions	0	2	2	1	2	0	5	...	31	42	37	33	37	86	60	54	46	836
Uzbekistan	Asia	Central Asia	Developing regions	0	0	0	0	0	0	0	...	330	262	284	215	288	289	162	235	167	3368
Viet Nam	Asia	South-Eastern Asia	Developing regions	1191	1829	2162	3404	7583	5907	2741	...	1852	3153	2574	1784	2171	1942	1723	1731	2112	97146
Yemen	Asia	Western Asia	Developing regions	1	2	1	6	0	18	7	...	161	140	122	133	128	211	160	174	217	2985

49 rows × 38 columns

In [42]:

```
# we can pass multiple criteria in the same line.
# Let's filter for AreaName = Asia and RegName = Southern Asia

df_can[(df_can['Continent']=='Asia') & (df_can['Region']=='Southern Asia')]

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

Out[42]:		Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
	Bangladesh	Asia	Southern Asia	Developing regions	83	84	86	81	98	92	486	...	4171	4014	2897	2939	2104	4721	2694	2640	3789	65568
	Bhutan	Asia	Southern Asia	Developing regions	0	0	0	0	1	0	0	...	5	10	7	36	865	1464	1879	1075	487	5876
	India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210	33848	28742	28261	29456	34235	27509	30933	33087	691904
	Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	5837	7480	6974	6475	6580	7477	7479	7534	11291	175923
	Maldives	Asia	Southern Asia	Developing regions	0	0	0	1	0	0	0	...	0	0	2	1	7	4	3	1	1	30
	Nepal	Asia	Southern Asia	Developing regions	1	1	6	1	2	4	13	...	607	540	511	581	561	1392	1129	1185	1308	10222
	Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314	13127	10124	8994	7217	6811	7468	11227	12603	241600
	Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	4930	4714	4123	4756	4547	4422	3309	3338	2394	148358

9 rows × 38 columns

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

In [43]:

```
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[43]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total	
	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699

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](#). 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 [44]: # we are using the inline backend  
%matplotlib inline
```

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

*optional: check if Matplotlib is loaded.

```
In [45]: print('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

```
Matplotlib version:  3.3.4
```

*optional: apply a style to Matplotlib.

```
In [46]: print(plt.style.available)  
mpl.style.use(['ggplot']) # optional: for ggplot-like style
```

```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn',  
'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted',  
'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-  
whitegrid', 'tableau-colorblind10']
```

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](#)

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.

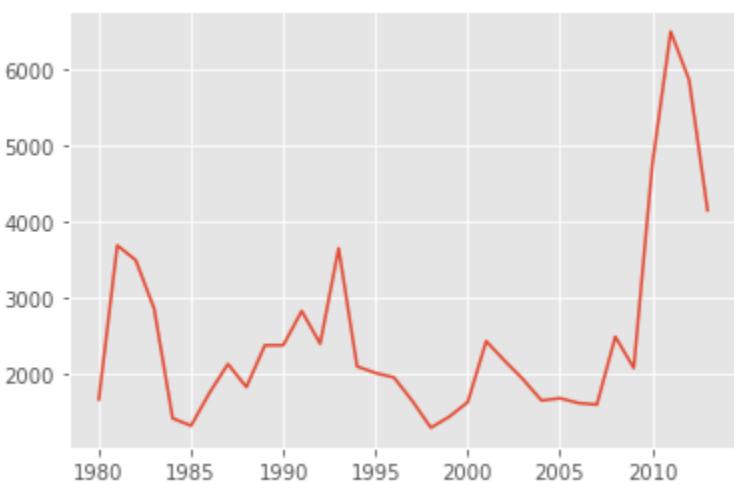
```
In [47]: haiti = df_can.loc['Haiti', years] # passing in years 1980 - 2013 to exclude the 'total' column  
haiti.head()
```

```
Out[47]: 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 [48]: haiti.plot()
```

```
Out[48]: <AxesSubplot:>
```



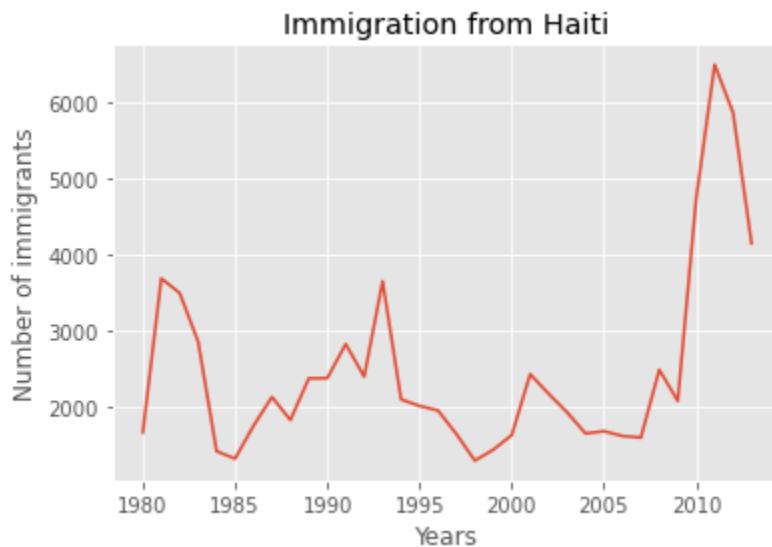
pandas automatically populated the x-axis with the index values (years), and the y-axis with the column values (population). However, notice how the years were not displayed because they are of type *string*. Therefore, let's change the type of the index values to *integer* for plotting.

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

```
In [49]: 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
```



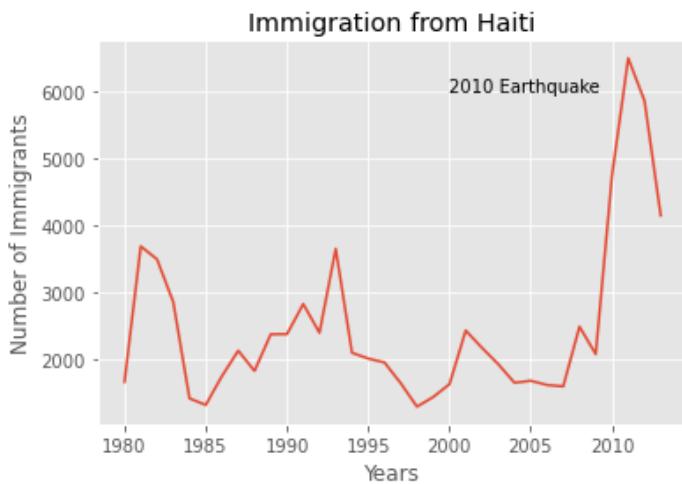
We can clearly notice how number of immigrants from Haiti spiked up from 2010 as Canada stepped up its efforts to accept refugees from Haiti. Let's annotate this spike in the plot by using the plt.text() method.

```
In [54]: 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 the dataframe.

```
Out[55]:
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
India	8880	8670	8147	7338	5704	4211	7150	10189	11522	10343	...	28235	36210	33848	28742	28261	29456	34235	27509	30933	33087
China	5123	6682	3308	1863	1527	1816	1960	2643	2758	4323	...	36619	42584	33518	27642	30037	29622	30391	28502	33024	34129

2 rows × 34 columns

Click here for a sample python solution ````python #The correct answer is: df_CI = df_can.loc[['India', 'China'], years] df_CI ```

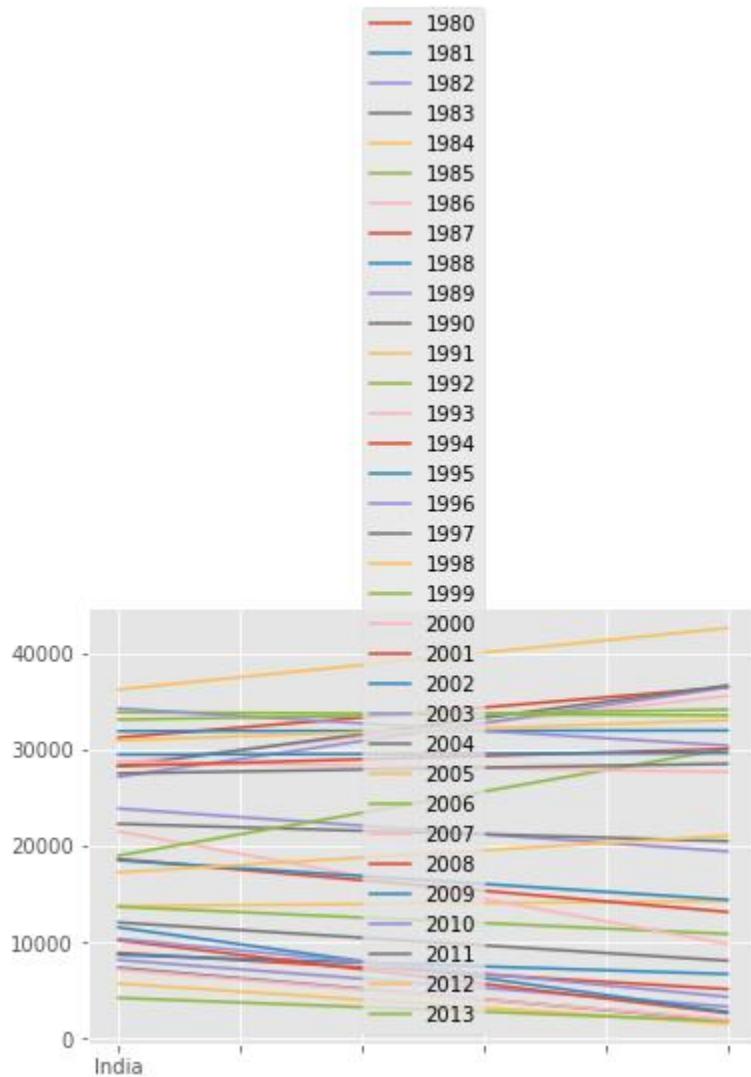
Step 2: Plot graph. We will explicitly specify line plot by passing in kind parameter to plot().

```
In [57]:
```

```
### type your answer here
df_CI.plot(kind='line')
```

```
Out[57]: <AxesSubplot:>
```

```
Out[57]: <AxesSubplot:>
```



Click here for a sample python solution ````python #The correct answer is: df_CI.plot(kind='line')````

That doesn't look right...

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

```
In [58]: df_CI = df_CI.transpose()  
df_CI.head()
```

```
Out[58]:
```

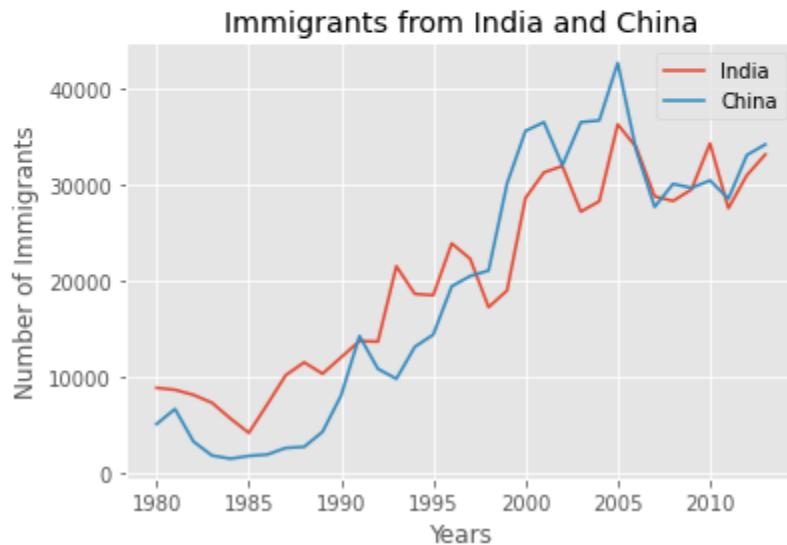
	India	China
1980	8880	5123
1981	8670	6682
1982	8147	3308
1983	7338	1863
1984	5704	1527

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 [59]: df_CI.index = df_CI.index.map(int)
df_CI.plot(kind='line')

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

plt.show()
```



Click here for a sample python solution ````python #The correct answer is: 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() ````

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 [62]: inplace = True  
df_can.sort_values(by='Total', ascending=False, axis=0, inplace=True)  
  
df_top5 = df_can.head(5)  
  
df_top5 = df_top5[years].transpose()  
  
print(df_top5)
```

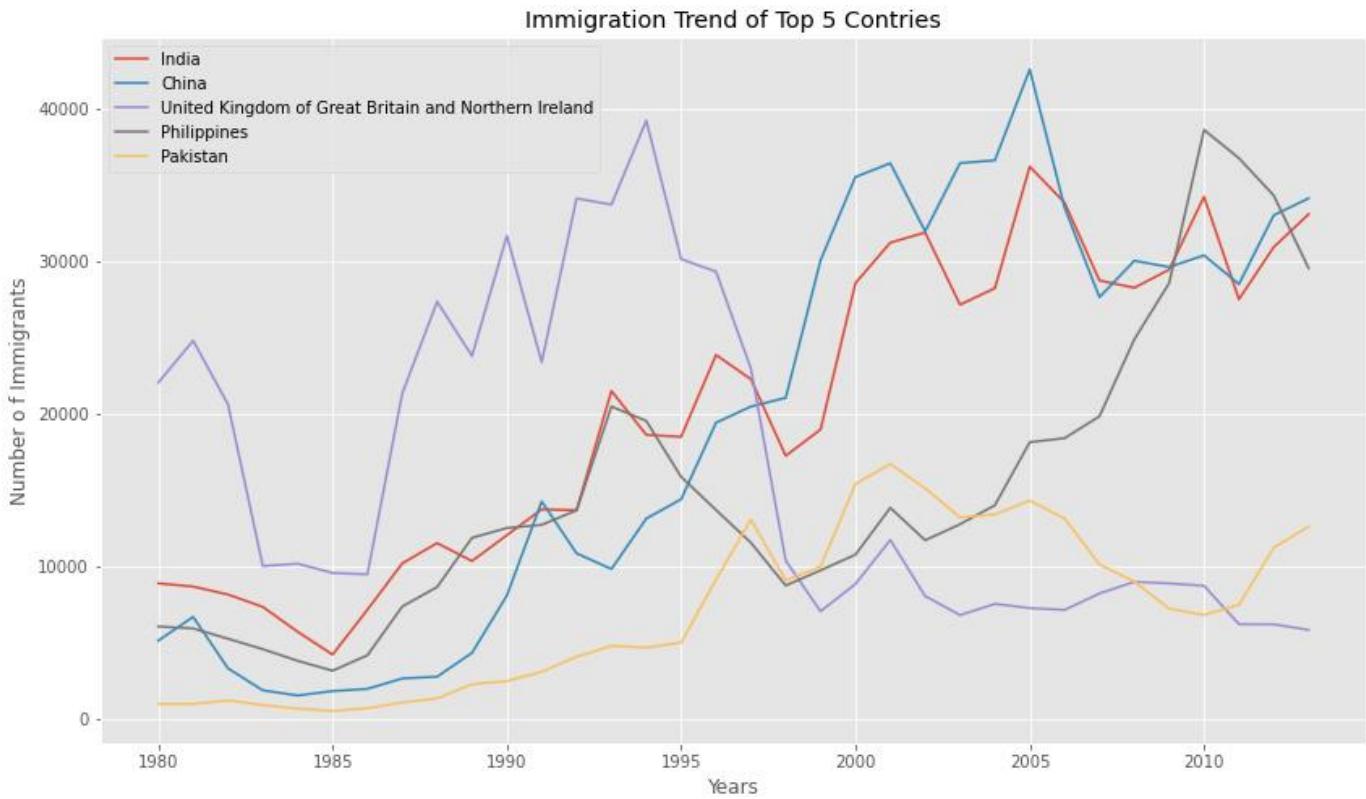
	India	China	United Kingdom of Great Britain and Northern Ireland
1980	8880	5123	22045
1981	8670	6682	24796
1982	8147	3308	20620
1983	7338	1863	10015
1984	5704	1527	10170
1985	4211	1816	9564
1986	7150	1960	9470
1987	10189	2643	21337
1988	11522	2758	27359
1989	10343	4323	23795
1990	12041	8076	31668
1991	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

```
In [65]: df_top5.index = df_top5.index.map(int)
df_top5.plot(kind='line', figsize=(14, 8))

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

plt.show()
```



Click here for a sample python solution ````python #The correct answer is: #Step 1: Get the dataset. Recall that we created a Total column that calculates cumulative immigration by country. #We will sort on this column to get our top 5 countries using pandas sort_values() method. inplace = True # paramemter saves the changes to the original df_can dataframe df_can.sort_values(by='Total', ascending=False, axis=0, inplace=True) # get the top 5 entries df_top5 = df_can.head(5) # transpose the dataframe df_top5 = df_top5[years].transpose() print(df_top5) #Step 2: Plot the dataframe. To make the plot more readeable, we will change the size using the 'figsize' parameter. df_top5.index = df_top5.index.map(int) # 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() ````

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
- **bard** 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

Question 1

1/1 point (graded)

Using the inline backend, you can modify a figure after it is rendered.

True

False



Answer

Correct: Correct. You cannot modify a figure after it is rendered using the inline backend.

Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

Question 2

1/1 point (graded)

Which of the following are examples of Matplotlib magic functions? Choose all that apply.

%matplotlib inline

#matplotlib notebook

\$matplotlib outline

%matplotlib notebook

#matplotlib inline



Submit

You have used 1 of 2 attempts

✓ Correct (1/1 point)

Question 3

1/1 point (graded)

Matplotlib was created by John Hunter, an American neurobiologist, and was originally developed as an EEG/ECoG visualization tool.

True

False



Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

Module 2 - Basic and Specialized Visualization Tools

- Area Plots
- Histograms
- Bar Charts
- Hands-on Lab: Basic Visualization Tools
- Optional: Download Jupyter Notebook

Module Introduction

In this module, you learn about area plots and how to create them with Matplotlib, histograms and how to create them with Matplotlib, bar charts, and how to create them with Matplotlib, pie charts, and how to create them with Matplotlib, box plots and how to create them with Matplotlib, and scatter plots and bubble plots and how to create them with Matplotlib.

Learning Objectives

In this lesson you will learn about:

- Explain how to generate an area plot using Matplotlib
- Describe why and how to create histograms using Matplotlib
- Explain how to create bar charts
- Create pie charts using Matplotlib
- Create a box plot using Matplotlib

Area Plots

We will learn about another visualization tool: the area plot, which is actually an extension of the line plot that we learned about in an earlier video. So what is an area plot? An area plot also known as an area chart or graph is a type of plot that depicts accumulated totals using numbers or percentages over time. It is based on the line plot and is commonly used when trying to compare two or more quantities.

So how can we generate an area plot with Matplotlib? Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

Now let's process the dataframe so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also, let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan, it is 58,639, total, and for Albania, it is 15,699 and so on, and let's name our data frame df_canada. So now that we know how our data is stored in the dataframe, df_canada, let's try to generate area plots for the countries with the highest number of immigration to Canada. We can try to find these countries by sorting our dataframe in descending order of cumulative total immigration from 1980 to 2013. We use the sort_values function to sort our dataframe in descending order and here is the result. So it turns out that India followed by China then the UK, Philippines, and Pakistan are the top five countries with the highest number of immigration to Canada.

So can we now go ahead and generate the area plots using the first five rows of this dataframe? Not quite yet. First we need to create a new dataframe of only these five countries and we need to exclude the total column. More importantly, to generate the area plots for these countries, we need the years to be plotted on the horizontal axis and the annual immigration to be plotted on the vertical axis. Note that Matplotlib plots the indices of a dataframe on the horizontal axis, and with the dataframe as shown, the countries will be plotted on the horizontal axis. So to fix this, we need to take the transpose of the dataframe.

Let's see how we can do this. After we sort our dataframe in descending order of cumulative annual immigration, we create a new dataframe of the top five countries and let's call it df_top5. We then select only the columns representing the years 1980 to 2013 in order to exclude the total column before applying the transpose method. The resulting dataframe is exactly what we want, with five columns where each column represents one of the top five countries, and the years being the indices. Now we can go ahead and call the plot function on dataframe df_top5 to generate the area plots. To do that, first we import Matplotlib as mpl and its scripting interface as plt. Then we call the plot function on the dataframe df_top5 and, we set kind equals area to generate an area plot.

Then to complete the figure we give it a title and we label its axes. Finally we call the show function to display the figure. Note that here we're generating the area plot using the inline backend. And there you have it: an area plot that depicts the immigration trend of the five countries with the highest immigration to Canada from 1980 to 2013. In the lab session, we explore area plots in more details, so make sure to complete this module's lab session.

Data Visualization with Python

Area Plots

Area Plot

- Also known as area chart or area graph.
- Commonly used to represent cumulated totals using numbers or percentages over time.
- Is based on the line plot.

Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

Dataset - Processed

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58636
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	6	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

df_canada

Top five countries with the highest number of immigration to Canada

Steps:

1.

Generating Area Plots

```
df_canada.sort_values(['Total'], ascending = False, axis = 0, inplace = True)
```

Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210	33848	28742	28261	29456	34235	27509	30933	33087	691904
China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1900	...	42584	33518	27642	30037	29622	30391	28502	33024	34129	659962
United Kingdom of Great Britain and Northern Ireland	Europe	Northern Europe	Developed regions	22045	24796	20620	10015	10170	9564	9470	...	7258	7140	8216	8979	8876	8724	6204	6195	5827	551500
Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	18139	18400	19837	24887	28573	38617	36765	34315	29544	511391
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314	13137	10134	8904	7217	6811	7468	11227	12603	341600

India China United Kingdom Philippines Pakistan

2.

Generating Area Plots

```
years = list(map(str, range(1980, 2014)))
df_canada.sort_values(['Total'], ascending = False, axis = 0, inplace = True)
df_top5 = df_canada.head()
df_top5 = df_top5[years].transpose()
```

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

3.

Area Plots

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

df_top5.plot(kind='area')

plt.title('Immigration trend of top 5 countries')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')

plt.show()
```



4.

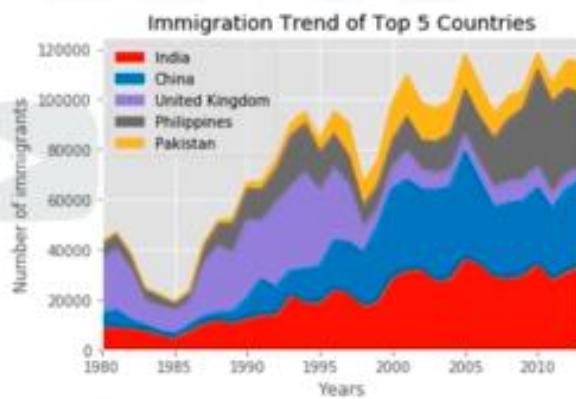
Area Plots

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

df_top5.plot(kind='area')

plt.title('Immigration trend of top 5 countries')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')

plt.show()
```



Histograms

We will learn about another visualization tool: the histogram, and we will learn how to create it using Matplotlib. Let's start by defining what a histogram is. A histogram is

- a way of representing the frequency distribution of a numeric dataset.
- The way it works is it partitions the spread of the numeric data into bins, assigns each datapoint in the dataset to a bin, and then counts the number of datapoints that have been assigned to each bin.
- So the vertical axis is actually the frequency or the number of datapoints in each bin.

For example, let's say the range of the numeric values in the dataset is 34,129. Now, the first step in creating the histogram is partitioning the horizontal axis in, say, ten bins of equal width, and then we construct the histogram by counting how many datapoints have a value that is between the limits of the first bin, the second bin, the third bin, and so on. Say the number of datapoints having a value between 0 and 3,413 is 175. Then we draw a bar of that height for this bin. We repeat the same thing for all the other bins, and if no datapoints fall into a bin then that bin would have a bar of height 0.

So how do we create a histogram using Matplotlib? Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

Now let's process the dataframe so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan for example, it is 58,639, total, and for Albania it is 15,699, and so on. And let's name our dataframe df_canada. So now that we know how our data is stored in the dataframe df_canada, say we're interested in visualizing the distribution of immigrants to Canada in the year 2013.

The simplest way to do that is to generate a histogram of the data in column 2013, and let's see how we can do that with Matplotlib. First, we import Matplotlib as mpl and its scripting interface as plt. Then we call the plot function on the data in column 2013 and we set kind equals hist to generate a histogram. Then to complete the figure, we give it a title and we label its axes. Finally, we call the show function to display the figure. And there you have it: A histogram that depicts the distribution of immigration to Canada in 2013, but notice how the bins are not aligned with the tick marks on the horizontal axis. This can make the histogram hard to read. So let's try to fix this in order to make our histogram more effective. One way to solve this issue is to borrow the histogram function from the Numpy library. So as usual we start by importing Matplotlib and its scripting interface, but this time we also import the Numpy library. Then we call the Numpy histogram function on the data in column 2013. What this function is going to do is it is going to partition the spread of the data in column 2013 into 10 bins of equal width, compute the number of datapoints that fall in each bin, and then return this frequency of each bin which we're calling count here and the bin edges which we're calling bin_edges.

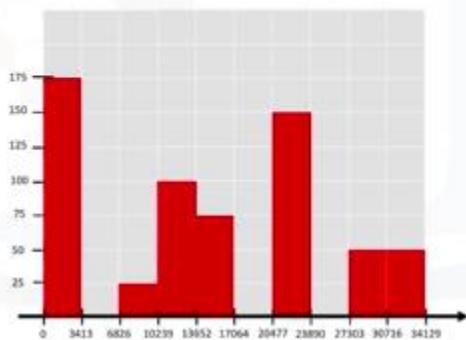
We then pass these bin edges as an additional parameter in our plot function to generate the histogram. And there you go: A nice looking histogram whose bin edges are aligned with the tick marks on the horizontal axis.

Data Visualization with Python

Histograms

Histogram

A histogram is a way of representing the frequency distribution of a variable.



COGNITIVE

Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	20
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	66
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	43
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Poynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3823	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

df_canada

Visualizing the distribution of immigrants to Canada in the year 2013.

Steps:

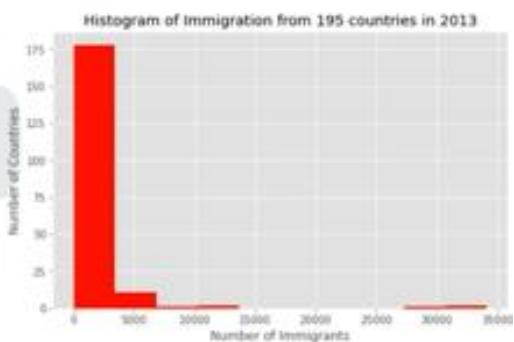
1. Using Matplotlib

Histograms

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

df_canada['2013'].plot(kind='hist')

plt.title('Histogram of Immigration from 195 countries in 2013')
plt.ylabel('Number of Countries')
plt.xlabel('Number of Immigrants')
plt.show()
```



2.

Notice how the bins are not aligned with the tick marks on the horizontal axis. This can make the histogram hard to read.

Histograms

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

count, bin_edges = np.histogram(df_canada['2013'])

df_canada['2013'].plot(kind='hist', xticks = bin_edges)

plt.title('Histogram of Immigration from 195 countries in 2013')
plt.ylabel('Number of Countries')
plt.xlabel('Number of Immigrants')

plt.show()
```

One way to solve this issue is to borrow the histogram function from the Numpy library. So as usual we start by importing Matplotlib and its scripting interface, but this time we also import the Numpy library. Then we call the Numpy histogram function on the data in column 2013. What this function is going to do is it is going to partition the spread of the data in column 2013 into 10 bins of equal width, compute the number of datapoints that fall in each bin, and then return this frequency of each bin which we're calling count here and the bin edges which we're calling bin_edges. We then pass these bin edges as an additional parameter in our plot function to generate the histogram.

3.

Histograms

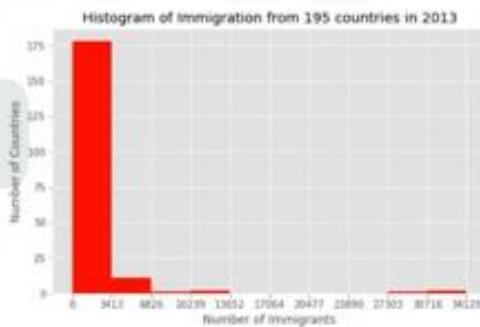
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

count, bin_edges = np.histogram(df_canada['2013'])

df_canada['2013'].plot(kind='hist', xticks = bin_edges)

plt.title('Histogram of Immigration from 195 countries in 2013')
plt.ylabel('Number of Countries')
plt.xlabel('Number of Immigrants')

plt.show()
```



Histogram whose bin edges are aligned with the tick marks on the horizontal axis.

Bar Charts

We will learn about an additional visualization tool, namely the bar chart, and learn how to create it using Matplotlib. A **bar chart** is

- a very popular visualization tool. Unlike a histogram,
- a bar chart also known as a bar graph is a type of plot where the length of each bar is proportional to the value of the item that it represents.
- It is commonly used to compare the values of a variable at a given point in time.

For example, say we're interested in visualizing in a discrete fashion how immigration from Iceland to Canada looked like from 1980 to 2013. One way to do that is by building a bar chart where the height of the bar represents the total immigration from Iceland to Canada in a particular year. So how do we do that with Matplotlib. Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

Now let's process the dataframe so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also, let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan for example, it is 58,639, total, and for Albania it is 15,699 and so on. And let's name our dataframe, `df_canada`.

So now that we know how our data is stored in the dataframe, `df_canada`, let's see how we can use Matplotlib to generate a bar chart to visualize how immigration from Iceland to Canada looked like from 1980 to 2013. As usual, we start by importing Matplotlib and its scripting interface. Then we use the years variable to create a new dataframe; let's name it `df_iceland`, which includes the data pertaining to annual immigration from Iceland to Canada and excluding the total column. Then we call the `plot` function on `df_iceland`, and we set `kind` equals `bar` to generate a bar chart. Then to complete the figure we give it a title, and we label its axes.

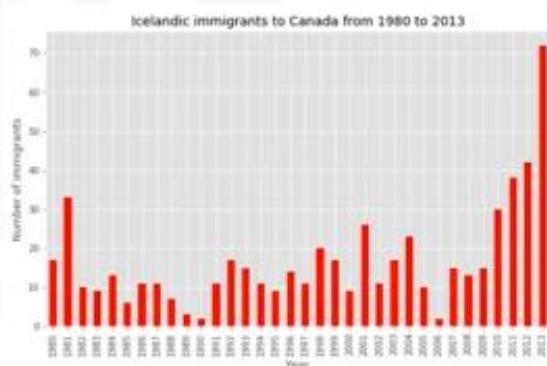
Finally, we call the `show` function to display the figure. And there you have it: A bar chart that depicts the immigration from Iceland to Canada from 1980 to 2013. By examining the bar chart, we notice that immigration to Canada from Iceland has seen an increasing trend since 2010. I'm sure that the curious among you are already wondering who the culprit behind this increasing trend is. In the lab session, we reveal the reason and we also learn how to create a bar chart with horizontal bars

Data Visualization with Python

Bar Charts

Bar Chart

Unlike a histogram, a bar chart is commonly used to compare the values of a variable at a given point in time.



COGNITIVE

Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

df_canada

Visualizing how immigration from Iceland to Canada looked like from 1980 to 2013

Steps:

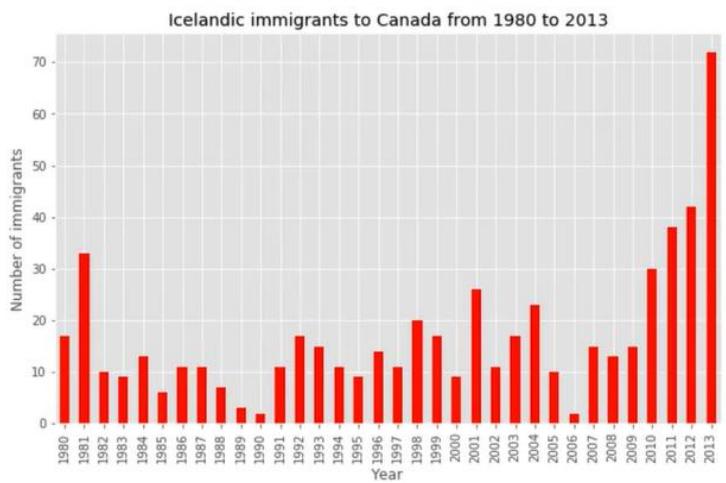
1. Using Matplotlib

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

years = list(map(str, range(1980, 2014)))
df_iceland = df_canada.loc['Iceland', years]

df_iceland.plot(kind='bar')

plt.title('Icelandic immigrants to Canada from 1980 to 2013')
plt.xlabel('Year')
plt.ylabel('Number of immigrants')
plt.show()
```



8.2.Basic and Specialized Visualization Tools

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

In this module, you learn about area plots and how to create them with Matplotlib, histograms and how to create them with Matplotlib, bar charts, and how to create them with Matplotlib, pie charts, and how to create them with Matplotlib, box plots and how to create them with Matplotlib, and scatter plots and bubble plots and how to create them with Matplotlib.

Learning Objectives

- Explain how to generate an area plot using Matplotlib
 - Describe why and how to create histograms using Matplotlib
 - Explain how to create bar charts
 - Create pie charts using Matplotlib
 - Create a box plot using Matplotlib
-

- [Visualization Tools](#)
- [Lab 2-1: Basic Visualization Tools](#)
- [Lab 2-2: Specialized Visualization Tools](#)

8.2.1. Visualization Tools

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Basic Visualization Tools

LATEST SUBMISSION GRADE 100%

Question 1

Area plots are unstacked by default.

- **True.**
- **False.**

Correct.

Question 2

The following code will create a histogram of a *pandas* series, `series_data`, and align the bin edges with the horizontal tick marks.

```
count, bin_edges = np.histogram(series_data)
series_data.plot(kind='hist', xticks = count, bin_edges)
```

- **True.**
- **False.**

Correct.

Question 3

The following code will create a horizontal bar chart of the data in a *pandas* dataframe, `question`.

```
question.plot(type='bar', rot=90)
```

- **True.**
- **False.**

Correct.

Specialized Visualization Tools

Box Plots

- Minimum (Q0 or 0th percentile): the lowest data point excluding any outliers
- First quartile (Q1 or 25th percentile): also known as the lower quartile , is the median of the lower half of the dataset
- Median (Q2 or 50th percentile): the middle value of the dataset
- Third quartile (Q3 or 75th percentile): also known as the upper quartile , is the median of the upper half of the dataset
- Maximum (Q4 or 100th percentile): the largest data point excluding any outliers
- Outliers: individual dots that occur outside the upper and lower extremes
- Interquartile range (IQR) : is the distance between the upper and lower quartiles.
 -

Specialized Visualization Tools

LATEST SUBMISSION GRADE 100%

Question 1

What do the letters in the box plot above represent?

- A = Median, B = Third Quartile, C = Mean, D = Inter Quartile Range, E = Lower Quartile, and F = Outliers
- A = Mean, B = Upper Mean Quartile, C = Lower Mean Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers
- A = Mean, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Maximum
- A = Median, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers
- A = Mean, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers

Correct.

Question 2

What is the correct combination of function and parameter to create a box plot in Matplotlib?

- ~~Function = box, and Parameter = type with value = "plot"~~
- ~~Function = boxplot, and Parameter = type with value = "plot"~~
- ~~Function = plot, and Parameter = type with value = "box"~~
- ~~Function = plot, and Parameter = kind with value = "boxplot"~~
- **Function = plot, and Parameter = kind with value = "box"**

Correct.

Question 3

Which of the lines of code below will create the following scatter plot, given the *pandas* dataframe, df_total?

```
import matplotlib.pyplot as plt  
  
df_total.plot(kind='scatter', x='year', y='total')  
  
plt.title('Total Immigrant population to Canada from 1980 - 2013')  
plt.xlabel ('Year')  
plt.ylabel('Number of Immigrants')
```

Correct.



Area Plots, Histograms, and Bar Plots

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Create additional labs namely area plots, histogram and bar charts

Table of Contents

1. [Exploring Datasets with *pandas*](#0)
2. [Downloading and Prepping Data](#2)
3. [Visualizing Data using Matplotlib](#4)
4. [Area Plots](#6)
5. [Histograms](#8)
6. [Bar Charts](#10)

Exploring Datasets with *pandas* and Matplotlib

Toolkits: The course heavily relies on **pandas** and **Numpy** for data wrangling, analysis, and visualization. The primary plotting library that we are exploring 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. For this lesson, we will focus on the Canadian Immigration data.

Downloading and Prepping Data

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

```
! pip3 install openpyxl
```

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

In [2]:

```
df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2)
print('Data downloaded and read into a dataframe!')
```

Data downloaded and read into a dataframe!

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

In [3]:

```
df_can.head()
```

Out[3]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	0
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 43 columns

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

In [5]:

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

(195, 43)

Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to Introduction to Matplotlib and Line Plots lab for the rational and detailed description of the changes.

1. Clean up the dataset to remove columns that are not informative to us for visualization (eg. Type, AREA, REG).

```
In [6]: df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)

# Let's view the first five elements and see how the dataframe was changed
df_can.head()
```

Out[6]:

	OdName	AreaName	RegName	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	1223	856	702	560	716	561	539	620	603
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
4	Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 38 columns

Notice how the columns Type, Coverage, AREA, REG, and DEV got removed from the dataframe.

2. Rename some of the columns so that they make sense.

```
In [7]: df_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)

# Let's view the first five elements and see how the dataframe was changed
df_can.head()
```

Out[7]:

	Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	...	1450	1223	856	702	560	716	561	539	620	603
2	Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
4	Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 38 columns

Notice how the column names now make much more sense, even to an outsider.

3. For consistency, ensure that all column labels of type string.

```
In [8]: # Let's examine the types of the column labels
all(isinstance(column, str) for column in df_can.columns)
```

Out[8]: False

Notice how the above line of code returned `False` when we tested if all the column labels are of type `string`. So let's change them all to `string` type.

```
In [9]: df_can.columns = list(map(str, df_can.columns))

# Let's check the column labels types now
all(isinstance(column, str) for column in df_can.columns)
```

Out[9]: True

4. Set the country name as index - useful for quickly looking up countries using .loc method.

```
In [10]: df_can.set_index('Country', inplace=True)

# Let's view the first five elements and see how the dataframe was changed
df_can.head()
```

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1450	1223	856	702	560	716	561	539	620	603
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	0	1	0	0	0	0	0	0	0
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	0	1	1	0	0	0	0	1	1

5 rows × 37 columns

Notice now the country names now serve as indices.

5. Add total column.

```
In [11]: df_can['Total'] = df_can.sum(axis=1)

# Let's view the first five elements and see how the dataframe was changed
df_can.head()
```

Out[11]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	6	
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

5 rows × 38 columns

Now the dataframe has an extra column that presents the total number of immigrants from each country in the dataset from 1980 - 2013. So if we print the dimension of the data, we get:

```
In [12]: print('data dimensions:', df_can.shape)
```

```
data dimensions: (195, 38)
```

So now our dataframe has 38 columns instead of 37 columns that we had before.

```
In [13]: # finally, let's create a list of years from 1980 - 2013
# this will come in handy when we start plotting the data
years = list(map(str, range(1980, 2014)))

years
```

```
Out[13]: ['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']
```

Visualizing Data using Matplotlib

Import the matplotlib library.

```
In [15]: # use the inline backend to generate the plots within the browser
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt

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

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

Matplotlib version: 3.3.4

Area Plots

In the last module, we created a line plot that visualized the top 5 countries that contributed the most immigrants to Canada from 1980 to 2013. With a little modification to the code, we can visualize this plot as a cumulative plot, also known as a **Stacked Line Plot or Area plot**.

```
In [19]: df_can.sort_values(['Total'], ascending=False, axis=0, inplace=True)

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

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

df_top5.head()
```

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

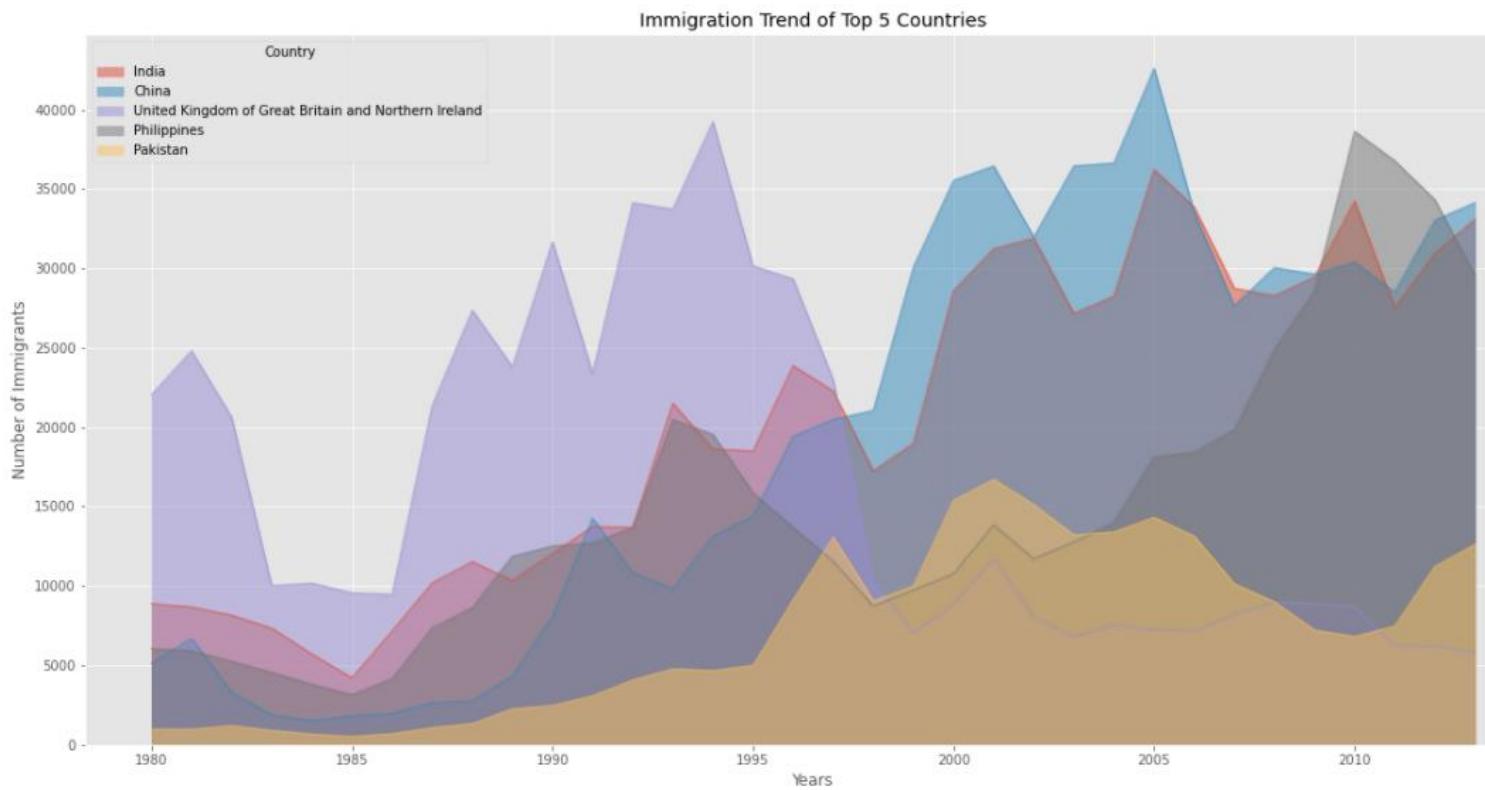
Area plots are stacked by default. And to produce a stacked area plot, each column must be either all positive or all negative values (any NaN, i.e. not a number, values will default to 0). To produce an unstacked plot, set parameter stacked to value False.

In [20]:

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

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

plt.show()
```

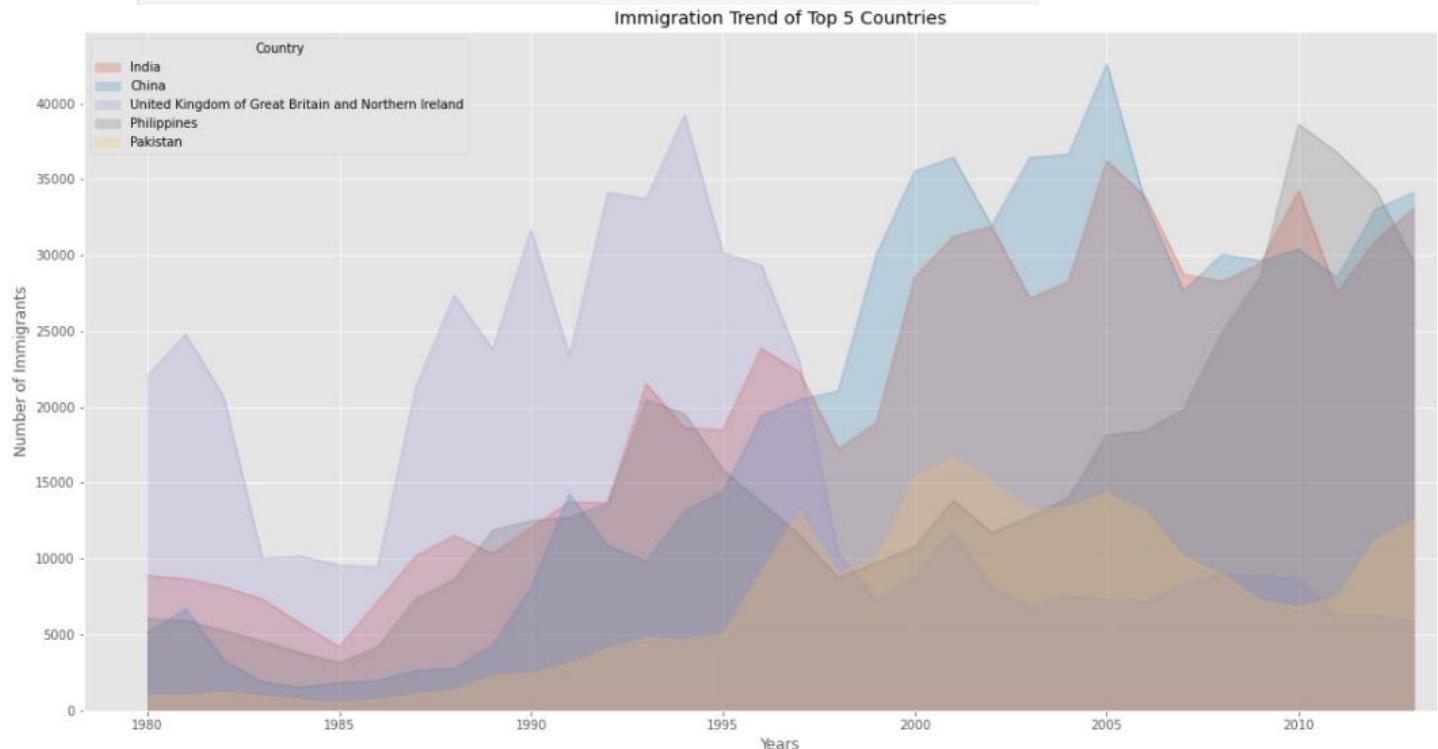


The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the `alpha` parameter.

```
In [21]: df_top5.plot(kind='area',
                   alpha=0.25, # 0 - 1, default value alpha = 0.5
                   stacked=False,
                   figsize=(20, 10))

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

plt.show()
```



Two types of plotting

As we discussed in the video lectures, there are two styles/options of plotting with matplotlib, plotting using the Artist layer and plotting using the scripting layer.

****Option 1: Scripting layer (procedural method) - using matplotlib.pyplot as 'plt' ****

You can use plt i.e. matplotlib.pyplot and add more elements by calling different methods procedurally; for example, plt.title(...) to add title or plt.xlabel(...) to add label to the x-axis.

```
# Option 1: This is what we have been using so far
df_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))
plt.title('Immigration trend of top 5 countries')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')
```

**Option 2: Artist layer (Object oriented method) - using an `Axes` instance from Matplotlib (preferred) **

You can use an `Axes` instance of your current plot and store it in a variable (eg. `ax`). You can add more elements by calling methods with a little change in syntax (by adding "`set_`" to the previous methods). For example, use `ax.set_title()` instead of `plt.title()` to add title, or `ax.set_xlabel()` instead of `plt.xlabel()` to add label to the x-axis.

This option sometimes is more transparent and flexible to use for advanced plots (in particular when having multiple plots, as you will see later).

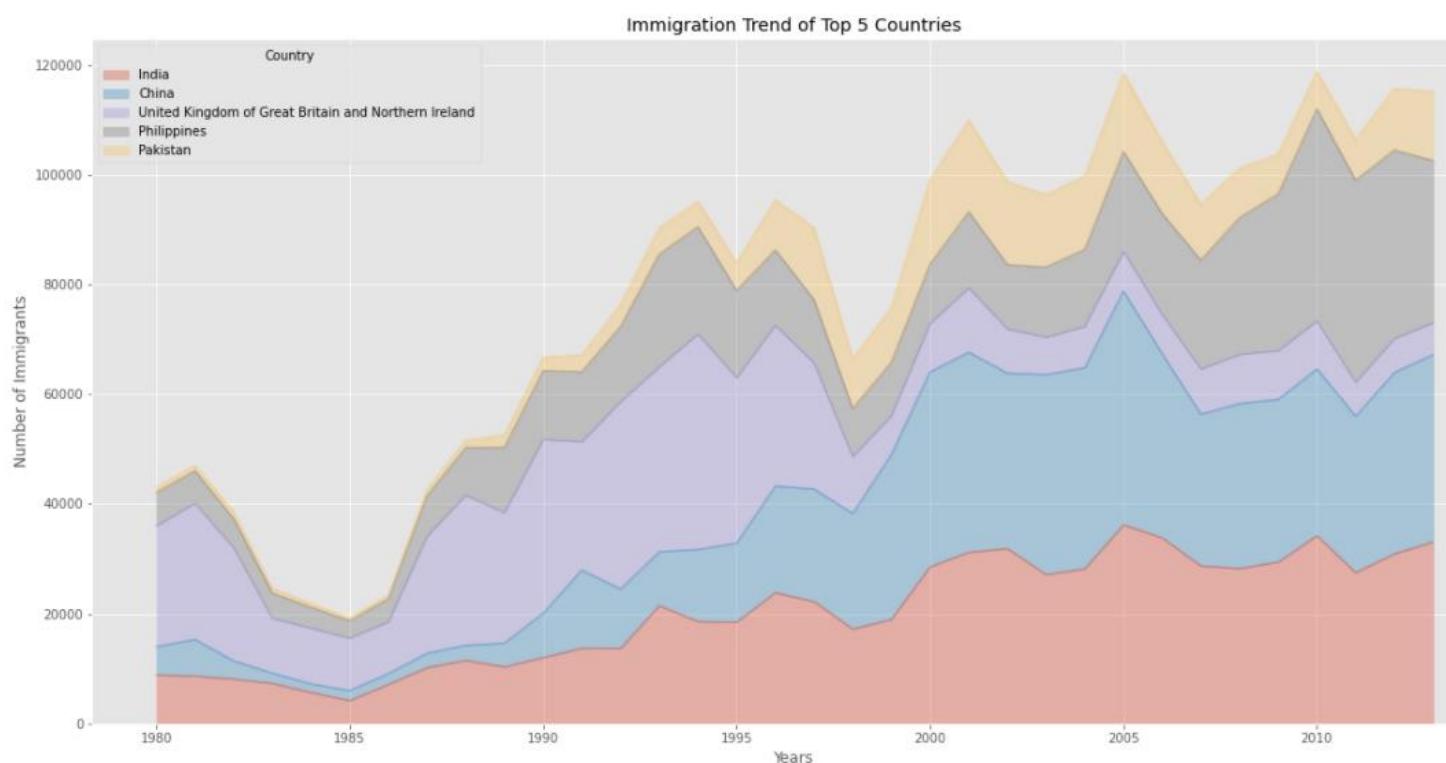
In this course, we will stick to the **scripting layer**, except for some advanced visualizations where we will need to use the **artist layer** to manipulate advanced aspects of the plots.

In [22]:

```
# option 2: preferred option with more flexibility
ax = df_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))

ax.set_title('Immigration Trend of Top 5 Countries')
ax.set_ylabel('Number of Immigrants')
ax.set_xlabel('Years')
```

Out[22]:



Question: Use the scripting layer to create a stacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.45.

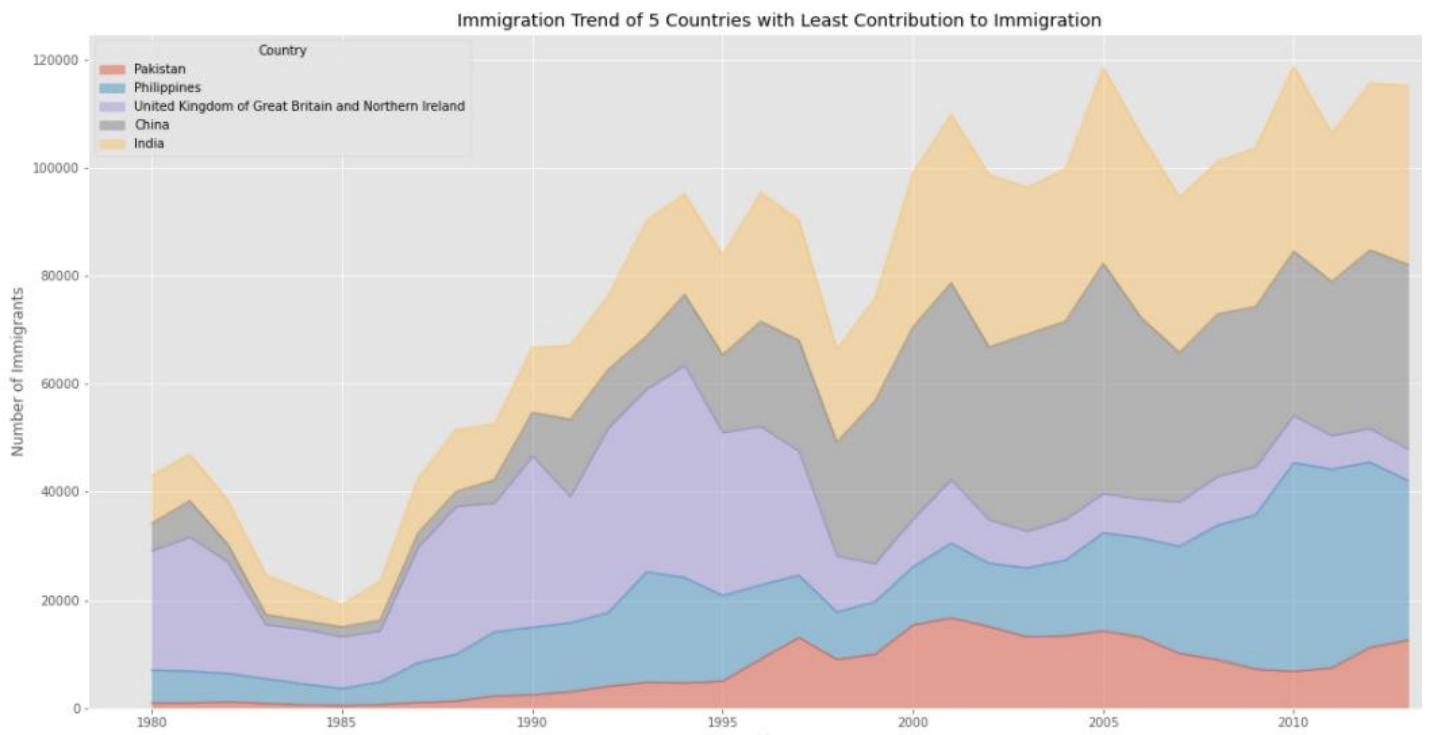
```
In [30]: df_bottom5 = df_can.tail(5)

# transpose the dataframe
df_bottom5 = df_bottom5[years].transpose()

df_bottom5.index = df_bottom5.index.map(int)
df_bottom5.plot(kind='area', alpha=0.45, figsize=(20, 10))

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

plt.show()
```



Click here for a sample python solution `python #The correct answer is: # get the 5 countries with the least contribution df_least5 = df_can.tail(5) # transpose the dataframe df_least5 = df_least5[years].transpose() df_least5.head() df_least5.index = df_least5.index.map(int) # let's change the index values of df_least5 to type integer for plotting df_least5.plot(kind='area', alpha=0.45, figsize=(20, 10)) plt.title('Immigration Trend of 5 Countries with Least Contribution to Immigration') plt.ylabel('Number of Immigrants') plt.xlabel('Years') plt.show()`

Question: Use the artist layer to create an unstacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.55.

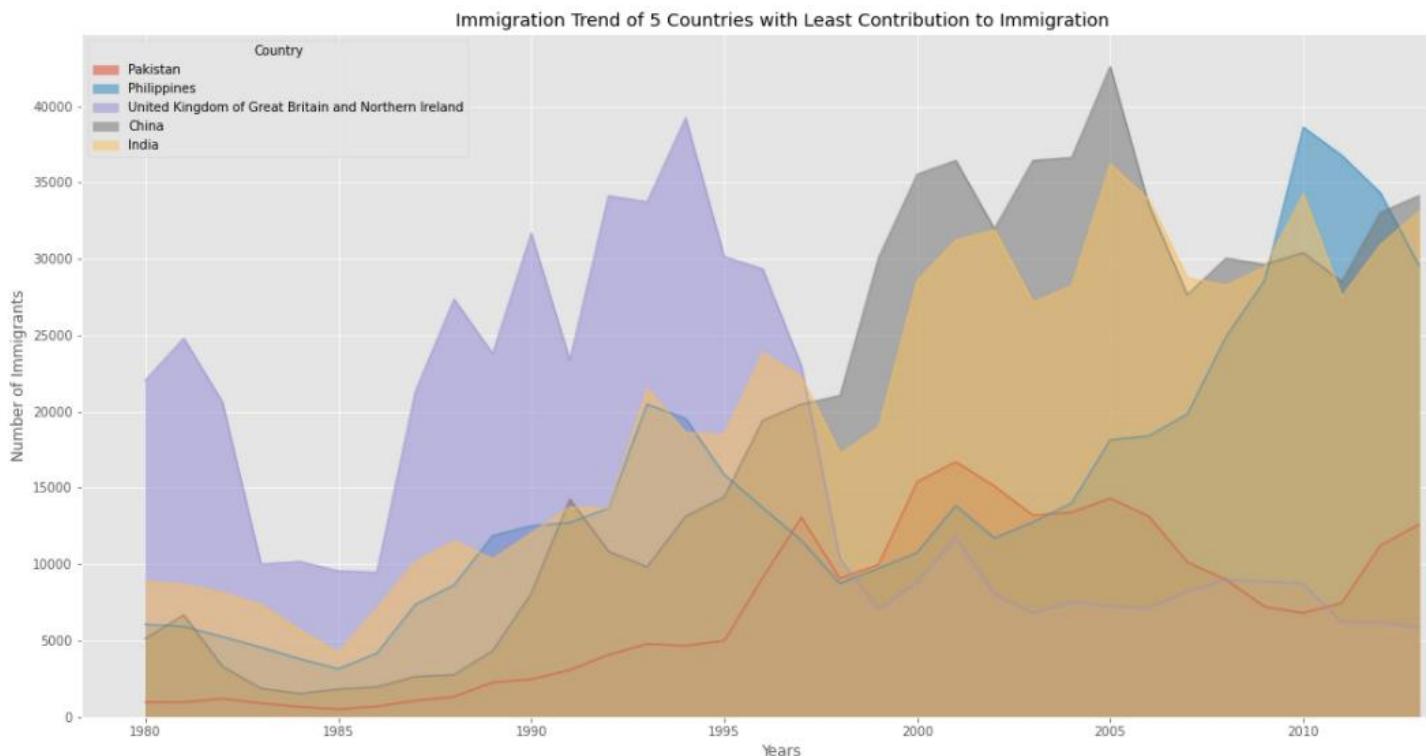
```
In [31]: df_bottom5 = df_can.tail(5)

# transpose the dataframe
df_bottom5 = df_bottom5[years].transpose()

df_bottom5.index = df_bottom5.index.map(int)
df_bottom5.plot(kind='area', stacked=False, alpha=0.55, figsize=(20, 10))

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

plt.show()
```



Click here for a sample python solution ````python #The correct answer is: # get the 5 countries with the least contribution df_least5 = df_can.tail(5) # transpose the dataframe df_least5 = df_least5[years].transpose() df_least5.head() df_least5.index = df_least5.index.map(int) # let's change the index values of df_least5 to type integer for plotting ax = df_least5.plot(kind='area', alpha=0.55, stacked=False, figsize=(20, 10)) ax.set_title('Immigration Trend of 5 Countries with Least Contribution to Immigration') ax.set_ylabel('Number of Immigrants') ax.set_xlabel('Years') ````

Histograms

A histogram is a way of representing the *frequency* distribution of numeric dataset. The way it works is it partitions the x-axis into *bins*, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So the y-axis is the frequency or the number of data points in each bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.

Question: What is the frequency distribution of the number (population) of new immigrants from the various countries to Canada in 2013?

Before we proceed with creating the histogram plot, let's first examine the data split into intervals. To do this, we will use **Numpy**'s histogram method to get the bin ranges and frequency counts as follows:

```
In [32]: # Let's quickly view the 2013 data  
df_can['2013'].head()
```

```
Out[32]: Country  
Palau          0  
Western Sahara 0  
Marshall Islands 0  
New Caledonia   2  
San Marino      0  
Name: 2013, dtype: int64
```

```
In [33]: # np.histogram returns 2 values  
count, bin_edges = np.histogram(df_can['2013'])  
  
print(count) # frequency count  
print(bin_edges) # bin ranges, default = 10 bins
```

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

By default, the histogram method breaks up the dataset into 10 bins. The figure below summarizes the bin ranges and the frequency distribution of immigration in 2013. We can see that in 2013:

- 178 countries contributed between 0 to 3412.9 immigrants
- 11 countries contributed between 3412.9 to 6825.8 immigrants
- 1 country contributed between 6825.8 to 10238.7 immigrants, and so on..

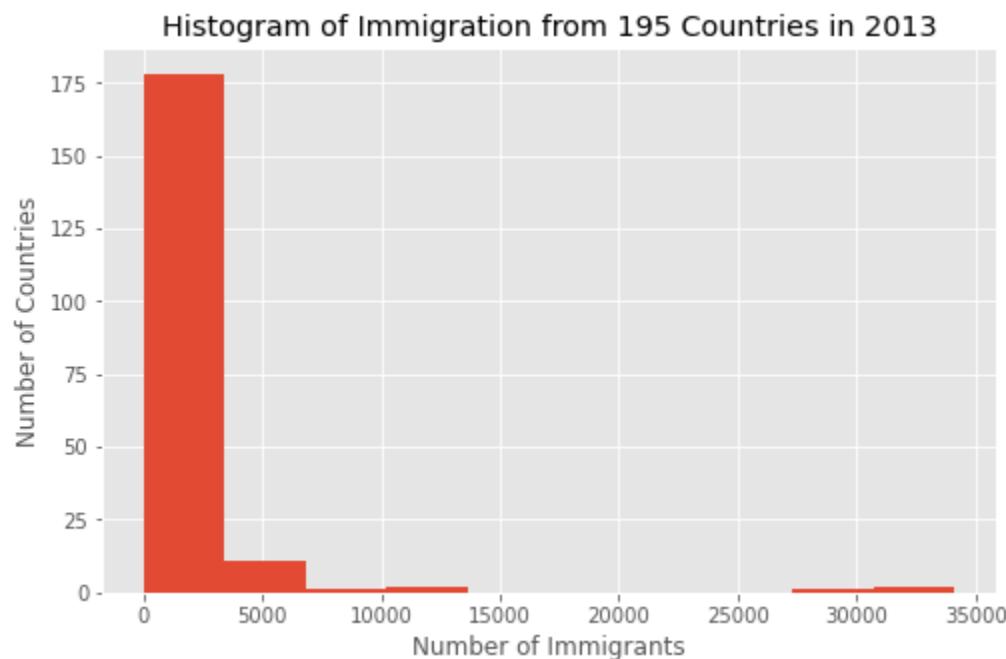
	Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
Range	0. to 3412.9	3412.9 to 6825.8	6825.8 to 10238.7	10238.7 to 13651.6	13651.6 to 17064.5	17064.5 to 20477.4	20477.4 to 23890.3	23890.3 to 27303.2	27303.2 to 30716.1	30716.1 to 34129.
Frequency	178	11	1	2	0	0	0	0	1	2

We can easily graph this distribution by passing kind=hist to plot().

```
In [34]: df_can['2013'].plot(kind='hist', figsize=(8, 5))
```

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

plt.show()
```



In the above plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the aforementioned population.

Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a `xticks` keyword that contains the list of the bin sizes, as follows:

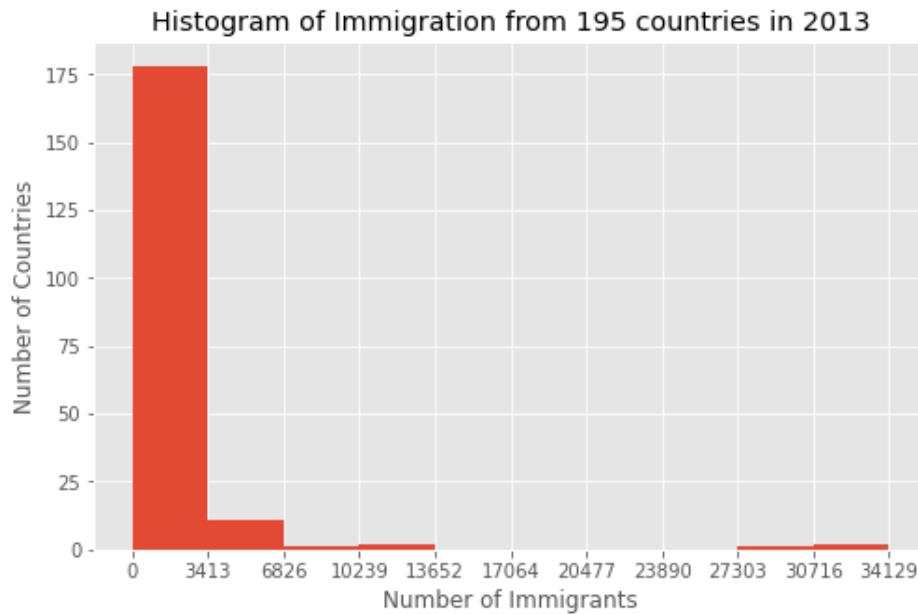
In [35]:

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

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

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

plt.show()
```



Side Note: We could use `df_can['2013'].plot.hist()`, instead. In fact, throughout this lesson, using `some_data.plot(kind='type_plot', ...)` is equivalent to `some_data.plot.type_plot(...)`. That is, passing the type of the plot as argument or method behaves the same.

See the *pandas* documentation for more info <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.plot.html>.

We can also plot multiple histograms on the same plot. For example, let's try to answer the following questions using a histogram.

Question: What is the immigration distribution for Denmark, Norway, and Sweden for years 1980 - 2013?

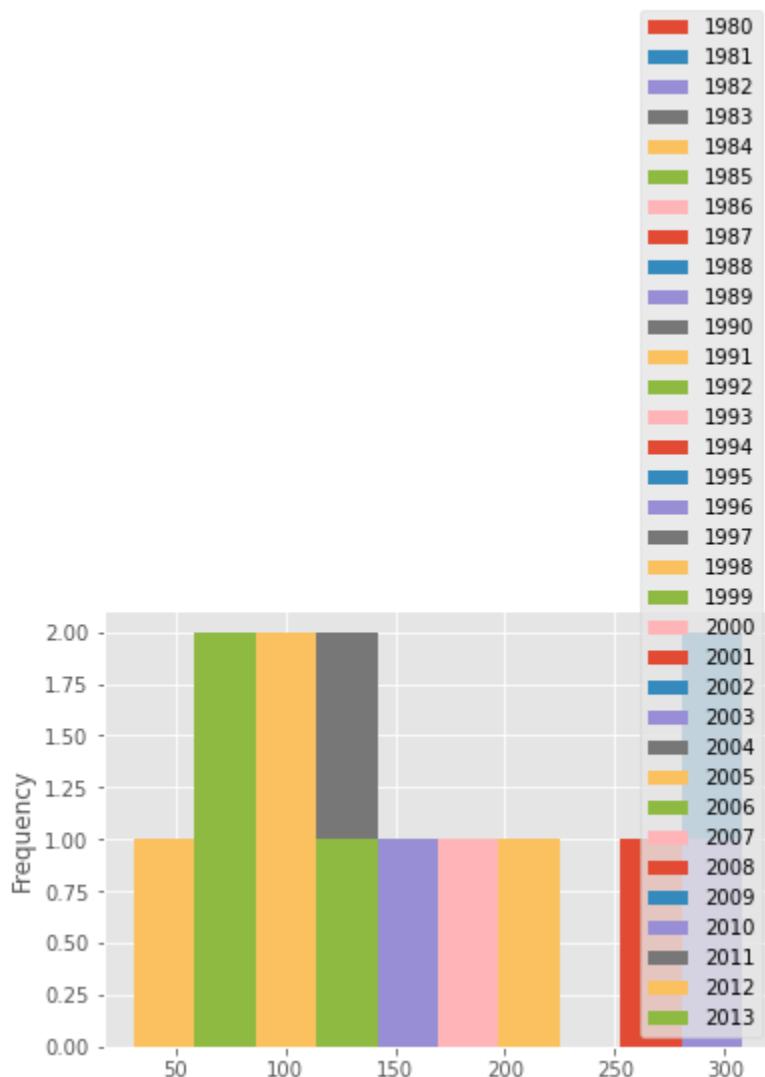
```
In [36]: # let's quickly view the dataset
df_can.loc[['Denmark', 'Norway', 'Sweden'], years]
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Country																					
Denmark	272	293	299	106	93	73	93	109	129	129	...	89	62	101	97	108	81	92	93	94	81
Norway	116	77	106	51	31	54	56	80	73	76	...	73	57	53	73	66	75	46	49	53	59
Sweden	281	308	222	176	128	158	187	198	171	182	...	129	205	139	193	165	167	159	134	140	140

3 rows × 34 columns

```
In [37]: # generate histogram
df_can.loc[['Denmark', 'Norway', 'Sweden'], years].plot.hist()
```

Out[37]: <AxesSubplot:ylabel='Frequency'>



That does not look right!

Don't worry, you'll often come across situations like this when creating plots. The solution often lies in how the underlying dataset is structured.

Instead of plotting the population frequency distribution of the population for the 3 countries, *pandas* instead plotted the population frequency distribution for the years.

This can be easily fixed by first transposing the dataset, and then plotting as shown below.

In [38]:

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

Out[38]:

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

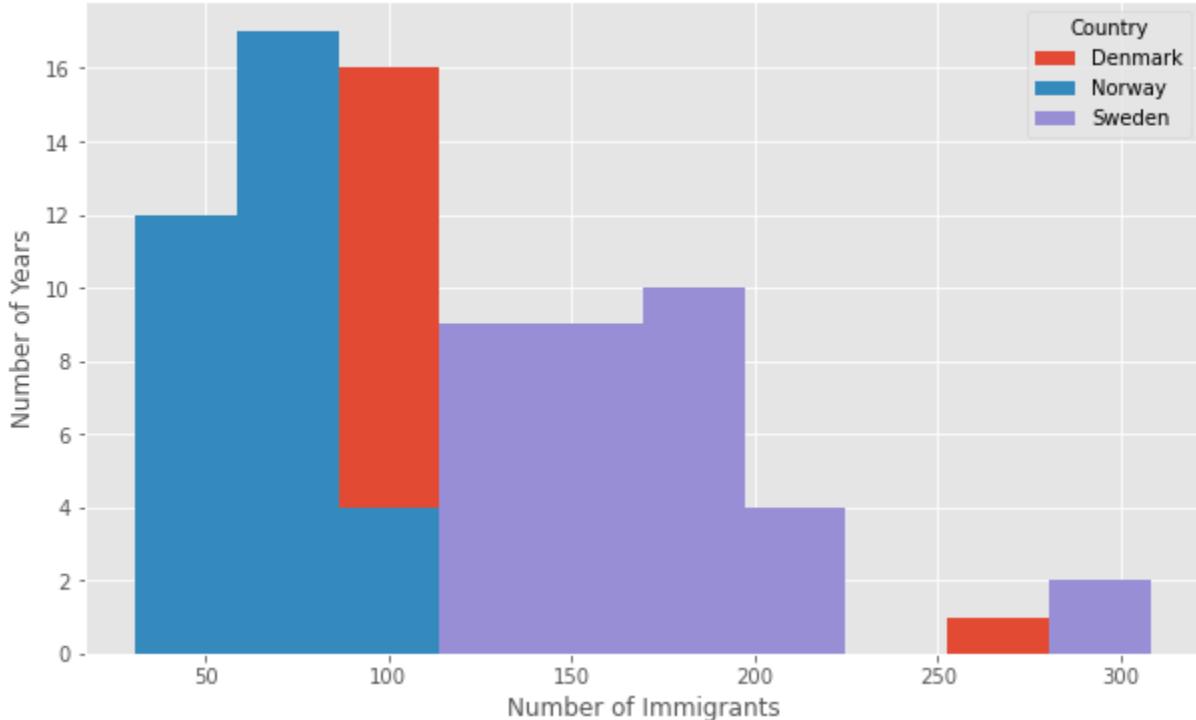
In [39]:

```
# generate histogram
df_t.plot(kind='hist', figsize=(10, 6))

plt.title('Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()
```

Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013



Let's make a few modifications to improve the impact and aesthetics of the previous plot:

- increase the bin size to 15 by passing in bins parameter;
- set transparency to 60% by passing in alpha parameter;
- label the x-axis by passing in x-label parameter;
- change the colors of the plots by passing in color parameter.

```
In [40]: # Let's get the x-tick values
count, bin_edges = np.histogram(df_t, 15)

# un-stacked histogram
df_t.plot(kind = 'hist',
           figsize=(10, 6),
           bins=15,
           alpha=0.6,
           xticks=bin_edges,
           color=['coral', 'darkslateblue', 'mediumseagreen']
         )

plt.title('Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()
```

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

```

import matplotlib

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

```

If we do not want the plots to overlap each other, we can stack them using the stacked parameter. Let's also adjust the min and max x-axis labels to remove the extra gap on the edges of the plot. We can pass a tuple (min,max) using the xlim parameter, as shown below.

```

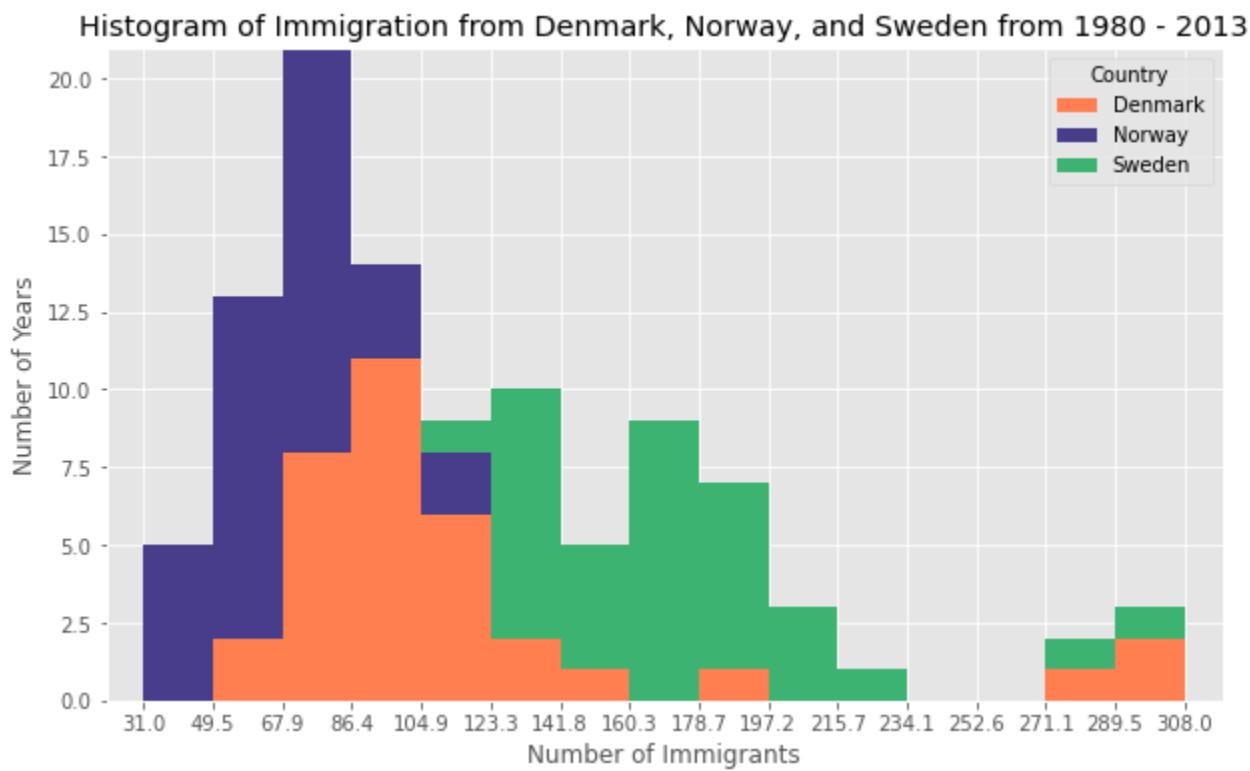
In [41]: count, bin_edges = np.histogram(df_t, 15)
xmin = bin_edges[0] - 10 # first bin value is 31.0, adding buffer of 10 for aesthetic purposes
xmax = bin_edges[-1] + 10 # last bin value is 308.0, adding buffer of 10 for aesthetic purposes

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

plt.title('Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()

```



Question: Use the scripting layer to display the immigration distribution for Greece, Albania, and Bulgaria for years 1980 - 2013? Use an overlapping plot with 15 bins and a transparency value of 0.35.

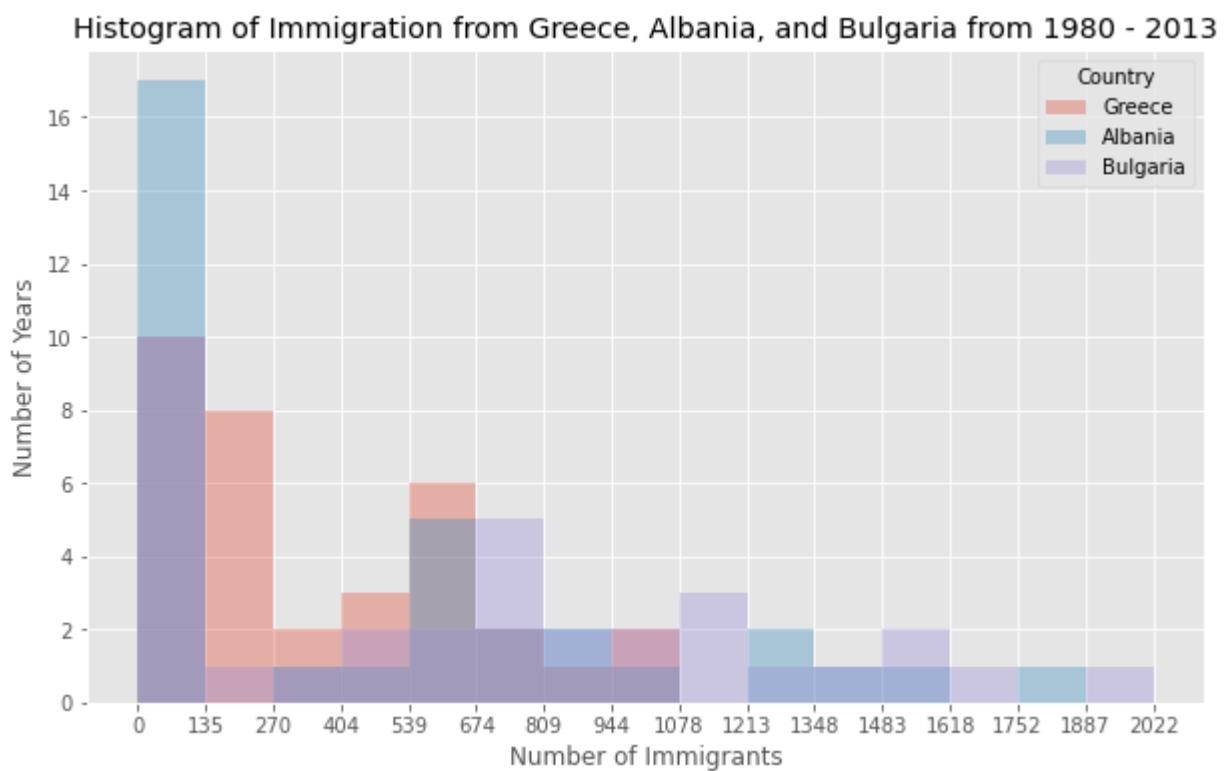
```
In [44]: df_GAB = df_can.loc[['Greece', 'Albania', 'Bulgaria'], years].transpose()

count, bin_edges = np.histogram(df_GAB, 15)

df_GAB.plot(kind = 'hist',
            figsize=(10, 6),
            bins=15,
            alpha=0.35,
            xticks=bin_edges
            )

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

plt.show()
```



Click here for a sample python solution ````python #The correct answer is: # create a dataframe of the countries of interest (cof) df_cof = df_can.loc[['Greece', 'Albania', 'Bulgaria'], years] # transpose the dataframe df_cof = df_cof.transpose() # let's get the x-tick values count, bin_edges = np.histogram(df_cof, 15) # Un-stacked Histogram df_cof.plot(kind = 'hist', figsize=(10, 6), bins=15, alpha=0.35, xticks=bin_edges, color=['coral', 'darkslateblue', 'mediumseagreen']) plt.title('Histogram of Immigration from Greece, Albania, and Bulgaria from 1980 - 2013') plt.ylabel('Number of Years') plt.xlabel('Number of Immigrants') plt.show() ````

Bar Charts (Dataframe)

A bar plot is a way of representing data where the *length* of the bars represents the magnitude/size of the feature/variable. Bar graphs usually represent numerical and categorical variables grouped in intervals.

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

- kind=bar creates a *vertical* bar plot
- kind=barr creates a *horizontal* bar plot

Vertical bar plot

In vertical bar graphs, the x-axis is used for labelling, and the length of bars on the y-axis corresponds to the magnitude of the variable being measured. Vertical bar graphs are particularly useful in analyzing time series data. One disadvantage is that they lack space for text labelling at the foot of each bar.

Let's start off by analyzing the effect of Iceland's Financial Crisis:

The 2008 - 2011 Icelandic Financial Crisis was a major economic and political event in Iceland. Relative to the size of its economy, Iceland's systemic banking collapse was the largest experienced by any country in economic history. The crisis led to a severe economic depression in 2008 - 2011 and significant political unrest.

Question: Let's compare the number of Icelandic immigrants (country = 'Iceland') to Canada from year 1980 to 2013.

```
In [45]: # step 1: get the data
df_iceland = df_can.loc['Iceland', years]
df_iceland.head()
```

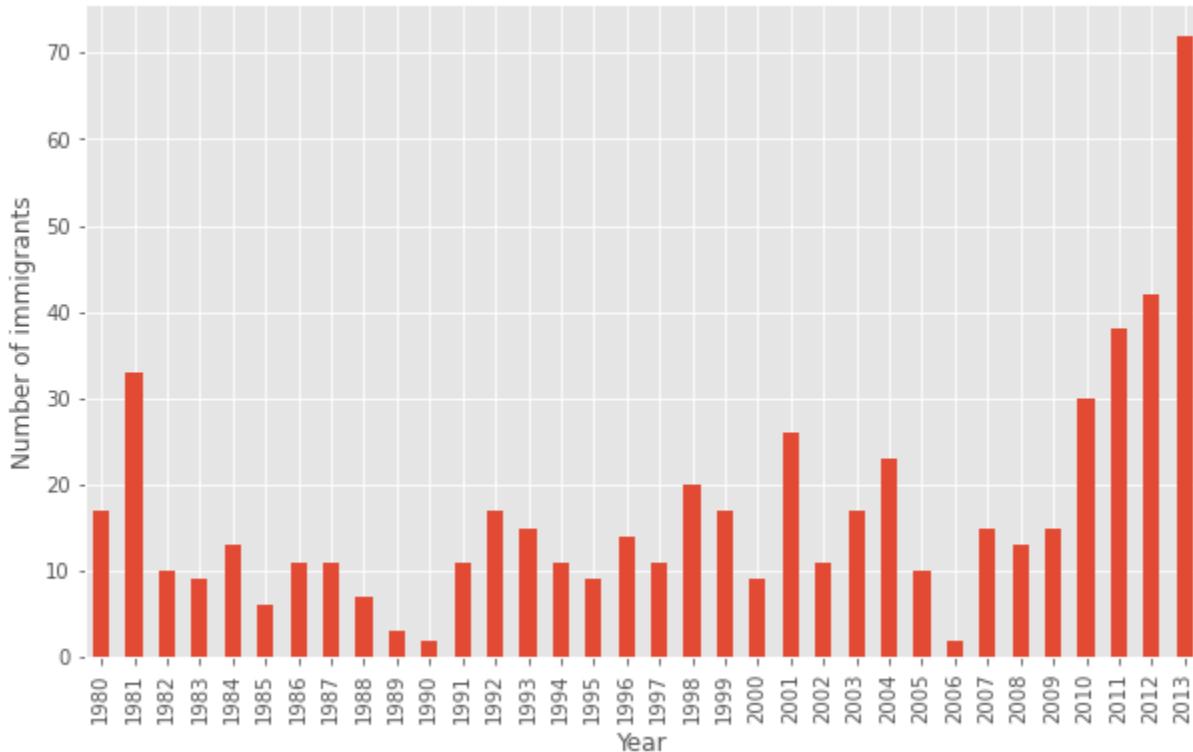
```
Out[45]: 1980    17
1981    33
1982    10
1983     9
1984    13
Name: Iceland, dtype: object
```

```
In [46]: # step 2: plot data
df_iceland.plot(kind='bar', figsize=(10, 6))

plt.xlabel('Year') # add to x-label to the plot
plt.ylabel('Number of immigrants') # add y-label to the plot
plt.title('Icelandic immigrants to Canada from 1980 to 2013') # add title to the plot

plt.show()
```

Icelandic immigrants to Canada from 1980 to 2013



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

Let's annotate this on the plot using the `annotate` method of the **scripting layer** or the **pyplot interface**. We will pass in the following parameters:

- `s`: str, the text of annotation.
- `xy`: Tuple specifying the (x,y) point to annotate (in this case, end point of arrow).
- `xytext`: Tuple specifying the (x,y) point to place the text (in this case, start point of arrow).
- `xycoords`: The coordinate system that `xy` is given in - 'data' uses the coordinate system of the object being annotated (default).
- `arrowprops`: Takes a dictionary of properties to draw the arrow:
 - `arrowstyle`: Specifies the arrow style, '`->`' is standard arrow.
 - `connectionstyle`: Specifies the connection type. `arc3` is a straight line.
 - `color`: Specifies color of arrow.
 - `lw`: Specifies the line width.

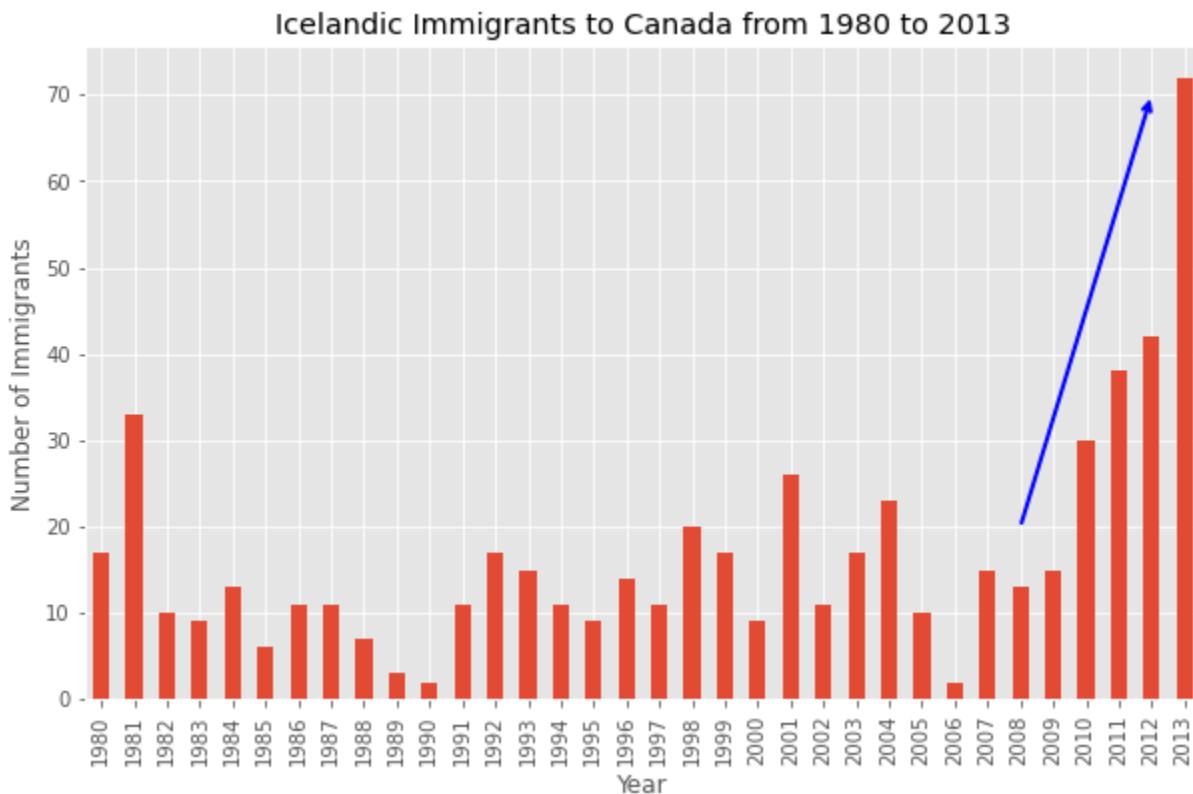
I encourage you to read the Matplotlib documentation for more details on annotations: http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.annotate.

```
In [47]: df_iceland.plot(kind='bar', figsize=(10, 6), rot=90) # rotate the xticks(labelled points on x-axis) by 90 degrees

plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
plt.title('Icelandic Immigrants to Canada from 1980 to 2013')

# Annotate arrow
plt.annotate('', # s: str. Will leave it blank for no text
            xy=(32, 70), # place head of the arrow at point (year 2012 , pop 70)
            xytext=(28, 20), # place base of the arrow at point (year 2008 , pop 20)
            xycoords='data', # will use the coordinate system of the object being annotated
            arrowprops=dict(arrowstyle='->', connectionstyle='arc3', color='blue', lw=2)
            )

plt.show()
```



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

- rotation: rotation angle of text in degrees (counter clockwise)
- va: vertical alignment of text ['center' | 'top' | 'bottom' | 'baseline']
- ha: horizontal alignment of text ['center' | 'right' | 'left']

```
In [48]: df_iceland.plot(kind='bar', figsize=(10, 6), rot=90)

plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
plt.title('Icelandic Immigrants to Canada from 1980 to 2013')

# Annotate arrow
plt.annotate('', # s: str. will leave it blank for no text
            xy=(32, 70), # place head of the arrow at point (year 2012 , pop 70)
            xytext=(28, 20), # place base of the arrow at point (year 2008 , pop 20)
            xycoords='data', # will use the coordinate system of the object being annotated
            arrowprops=dict(arrowstyle='->', connectionstyle='arc3', color='blue', lw=2)
            )

# Annotate Text
plt.annotate('2008 - 2011 Financial Crisis', # text to display
            xy=(28, 30), # start the text at at point (year 2008 , pop 30)
            rotation=72.5, # based on trial and error to match the arrow
            va='bottom', # want the text to be vertically 'bottom' aligned
            ha='left', # want the text to be horizontally 'left' aligned.
            )

plt.show()
```



Horizontal Bar Plot

Sometimes it is more practical to represent the data horizontally, especially if you need more room for labelling the bars. In horizontal bar graphs, the y-axis is used for labelling, and the length of bars on the x-axis corresponds to the magnitude of the variable being measured. As you will see, there is more room on the y-axis to label categorical variables.

Question: Using the scripting later and the df_can dataset, create a *horizontal* bar plot showing the *total* number of immigrants to Canada from the top 15 countries, for the period 1980 - 2013. Label each country with the total immigrant count.

Step 1: Get the data pertaining to the top 15 countries.

```
In [49]: df_can.sort_values(by='Total', ascending=False, inplace=True)  
df_top15 = df_can['Total'].head(15)  
df_top15
```

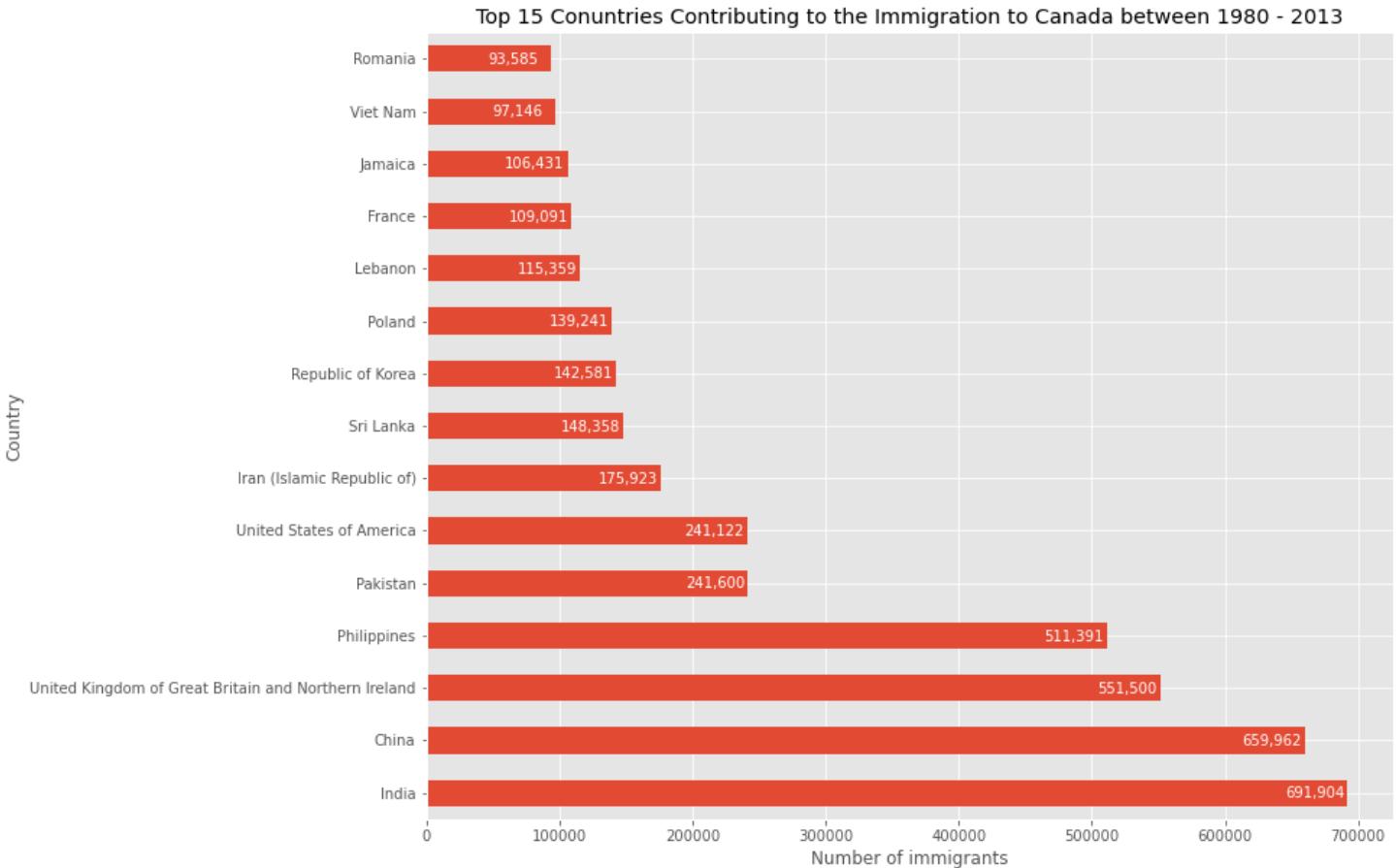
```
Out[49]: Country  
India 691904  
China 659962  
United Kingdom of Great Britain and Northern Ireland 551500  
Philippines 511391  
Pakistan 241600  
United States of America 241122  
Iran (Islamic Republic of) 175923  
Sri Lanka 148358  
Republic of Korea 142581  
Poland 139241  
Lebanon 115359  
France 109091  
Jamaica 106431  
Viet Nam 97146  
Romania 93585  
Name: Total, dtype: int64
```

Click here for a sample python solution ``python #The correct answer is: # sort dataframe on 'Total' column (descending) df_can.sort_values(by='Total', ascending=True, inplace=True) # get top 15 countries df_top15 = df_can['Total'].tail(15) df_top15 ````

Step 2: Plot data:

1. Use kind='barh' to generate a bar chart with horizontal bars.
2. Make sure to choose a good size for the plot and to label your axes and to give the plot a title.
3. Loop through the countries and annotate the immigrant population using the annotate function of the scripting interface.

```
In [57]: df_top15.plot(kind='barh', figsize=(12, 10))  
  
plt.xlabel('Number of immigrants')  
plt.title('Top 15 Countries Contributing to the Immigration to Canada between 1980 - 2013')  
  
for index, value in enumerate(df_top15):  
    label = format(int(value), ',') # format int with commas  
    # place text at the end of bar (subtracting 47000 from x, and 0.1 from y to make it fit within the bar)  
    plt.annotate(label, xy=(value - 47000, index - 0.1), color='white')  
  
plt.show()
```



Click here for a sample python solution ````python #The correct answer is: # generate plot df_top15.plot(kind='barh', figsize=(12, 12), color='steelblue') plt.xlabel('Number of Immigrants') plt.title('Top 15 Countries Contributing to the Immigration to Canada between 1980 - 2013') # annotate value labels to each country for index, value in enumerate(df_top15): label = format(int(value), ',') # format int with commas # place text at the end of bar (subtracting 47000 from x, and 0.1 from y to make it fit within the bar) plt.annotate(label, xy=(value - 47000, index - 0.10), color='white') plt.show() ````

Thank you for completing this lab!

Review Question 1

0/1 point (graded)

Area plots are stacked by default.

True

False



Answer

Incorrect: Incorec

Submit

You have used 1 of 1 attempt

Review Question 2

1/1 point (graded)

Given a *pandas* series, **series_data**, which of the following will create a histogram of *series_data* and align the bin edges with the horizontal tick marks?

count, bin_edges = np.histogram(series_data)
series_data.plot(kind='hist', xticks = count, bin_edges)

count, bin_edges = np.histogram(series_data)
series_data.plot(kind='hist', xticks = count)

count, bin_edges = np.histogram(series_data)
series_data.plot(kind='hist', xticks = bin_edges)

series_data.plot(kind='hist')



Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

Save

Review Question 3

1/1 point (graded)

Given a *pandas* dataframe, **question**, which of the following will create a horizontal barchart of the data in **question**?

- question.plot(type='bar', rot=90)
- question.plot(kind='bar', orientation='horizontal')
- question.plot(kind='barh')
- question.plot(kind='bar')



Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

S

Learning Objectives

In this lesson you will learn about:

- Pie charts, and how to create them with Matplotlib.
- Box plots, and how to create them with Matplotlib.
- Scatter plots and bubble plots, and how to create them with Matplotlib.

Pie Charts

We will learn about another visualization tool: the pie chart, and we will learn how to create it using Matplotlib. So what is a pie chart? A pie chart is a circular statistical graphic divided into slices to illustrate numerical proportion. For example, here is a pie chart of the Canadian federal election back in 2015 where the Liberals in red won more than 50% of the seats in the House of Commons. That is why the red color occupies more than half of the circle.

So how do we create a pie chart with Matplotlib? Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

Now let's process the dataframe so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan for example, it is 58,639, total, and for Albania, it is 15,699 and so on. And let's name our dataframe df_canada.

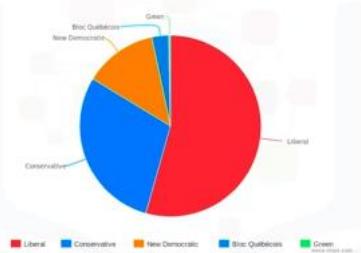
So now that we know how our data is stored in the dataframe df_canada, say we're interested in visualizing a breakdown of immigration to Canada continent wise. The first step is to group our data by continent using the continent column, and we use pandas for this. We call the pandas groupby function on df_canada, and we sum the number of immigrants from the countries that belong to the same continent. Here is the resulting dataframe, and let's name it df_continents. The resulting dataframe has six rows, each representing a continent and 35 columns representing the years from 1980 to 2013 plus the cumulative sum of immigration for each continent. And now we're ready to start creating our pie chart.

We start with the usual, importing Matplotlib as mpl and its scripting layer the pyplot interface as plt. Then we call the plot function on column total of the dataframe df_continents and we set kind equals pie to generate a pie chart. Then to complete the figure we give it a title. Finally we call the show function to display the figure. And there you have it: A pie chart that depicts each continent's proportion of immigration to Canada from 1980 to 2013. In the lab session, we will go through the process of creating a very professional-looking and aesthetically pleasing pie chart and transform the pie chart that we just created into one that looks like this. So make sure to complete this module's lab session. One last comment on pie charts. There are some very vocal opponents to

the use of pie charts under any circumstances. Most argue that pie charts fail to accurately display data with any consistency. Bar charts are much better when it comes to representing the data in a consistent way and getting the message across. If you're interested in learning more about the arguments against pie charts, here is a link to a very interesting article that discusses very clearly the flaws of pie charts

Pie Charts

Pie Chart



Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

df_canada

Pie Chart

```
df_continents = df_canada.groupby('Continent', axis = 0).sum()
```

Continent	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	... 2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Africa	3951	4363	3819	2671	2639	2650	3782	7494	7552	9894	... 27523	29188	28284	29860	34534	40882	35441	38063	38543	618948
Asia	31025	34314	30214	24696	27274	23850	28739	43203	47454	60256	... 159253	149054	133459	138894	141434	163645	146884	152218	156073	3317794
Europe	39780	44802	42720	24638	22287	20844	24370	46698	54726	60893	... 35855	33053	33495	34892	35078	33425	26778	29177	28691	1410947
Latin America and the Caribbean	13081	15215	16789	15427	13678	15171	21179	28471	21924	25090	... 24747	24676	26011	26547	26887	28818	27856	27173	24950	765148
Northern America	9378	10303	9074	7100	6661	6543	7074	7705	6469	6790	... 8384	9813	9463	10190	8995	8142	7677	7892	8503	241142
Oceania	1942	1839	1675	1018	878	920	904	1200	1181	1539	... 1585	1473	1693	1834	1800	1834	1548	1679	1775	55174

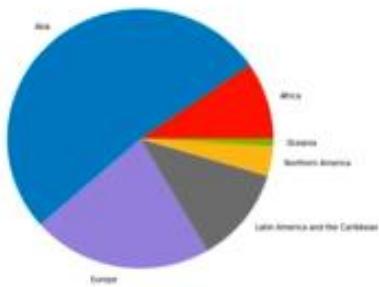
Visualizing a breakdown of immigration to Canada continent wise

Pie Chart

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

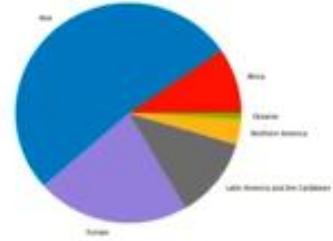
```
df_continents['Total'].plot(kind='pie')
plt.title('Immigration to Canada by Continent [1980-2013]')
plt.show()
```

Immigration to Canada by Continent [1980-2013]

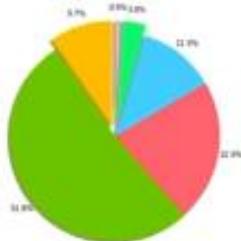


Pie Charts

Immigration to Canada by Continent [1980-2013]



Immigration to Canada by Continent [1980-2013]



Box Plots

We will learn about another visualization tool, the **Boxplot** and how to create one using matplotlib. So, what is a boxplot? A **boxplot** is a **way of statistically representing the distribution of given data through 5 main dimensions**.

- The **first dimension is minimum, which is the smallest number in the sorted data**. Its value can be obtained by subtracting 1.5 times the IQR where IQR is interquartile range from the first quartile.
- The **second dimension is first quartile which is 25% of the way through the sorted data**. In other words, 1/4 of the data points are less than this value.
- The **third dimension is median, which is the median of the sorted data**.
- The **fourth dimension is third quartile, which is 75% of the way through the sorted data**. In other words, 3/4 of the data points are less than this value.
- And the **final dimension is maximum, which is the highest number in the sorted data where maximum equals third quartile summed with 1.5 multiplied by IQR**.
- Finally, **boxplots also display outliers as individual dots that occur outside the upper and lower extremes**.

Now let's see how we can create a boxplot with Matplotlib. Before we go over the code to do that. Let's do a quick recap of our data set. Recall that each row represents a country and contains metadata about the country, such as where it is located geographically, and whether it is developing or developed.

Each row contains numerical figures of annual immigration from that country to Canada from 1980 to 2013. Now let's process the data frame so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also, let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So, for Afghanistan, for example, it is 58,639 total and for Albania it is 15,699. And so on. And let's name our data frame, DF_Canada. So now that we know how our data is stored in the Dataframe DF_Canada, say we're interested in creating a boxplot to visualize immigration from Japan to Canada.

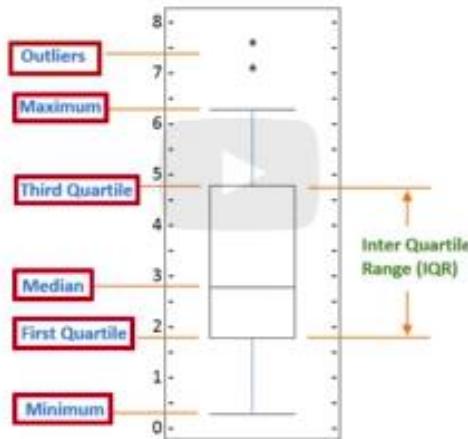
As with other tools that we learned so far, we start by importing Matplotlib as MPL and the pyplot interface as PLT. Then we create a new data frame of the data pertaining to Japan and we're excluding the column total using the years variable. Then we transpose the resulting data frame to make it in the correct format to create the Boxplot.

Let's name this new data frame DF_Japan. Following that, we call the plot function on DF_Japan and we set kind equals box to generate a boxplot. Then, to complete the figure, we give it a title and we label the vertical axis. Finally, we call the show function to display the figure and there you have it, a boxplot that provides a pleasing distribution of Japanese immigration to Canada from 1980 to 2013. In the lab session we explore boxplots in more detail and learn how to create multiple boxplots as well as horizontal boxplots

Data Visualization with Python

Box Plots

Box Plots



The first dimension is minimum, which is the smallest number in the sorted data. Its value can be obtained by subtracting 1.5 times the IQR where IQR is interquartile range from the first quartile. The second dimension is first quartile which is 25% of the way through the sorted data. In other words, 1/4 of the data points are less than this value. The third dimension is median, which is the median of the sorted data. The fourth dimension is third quartile, which is 75% of the way through the sorted data. In other words, 3/4 of the data points are less than this value. And the final dimension is maximum, which is the highest number in the sorted data where maximum equals third quartile summed with 1.5 multiplied by IQR. Finally, boxplots also display outliers as individual dots that occur outside the upper and lower extremes.

Visualize immigration from Japan to Canada

Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	... 2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	... 2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	... 1450	1223	856	702	560	716	561	539	620	603
2	immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	60	... 3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	... 0	0	1	0	0	0	0	0	0	
4	immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	... 0	0	1	1	0	0	0	1	1	

Dataset - Processed

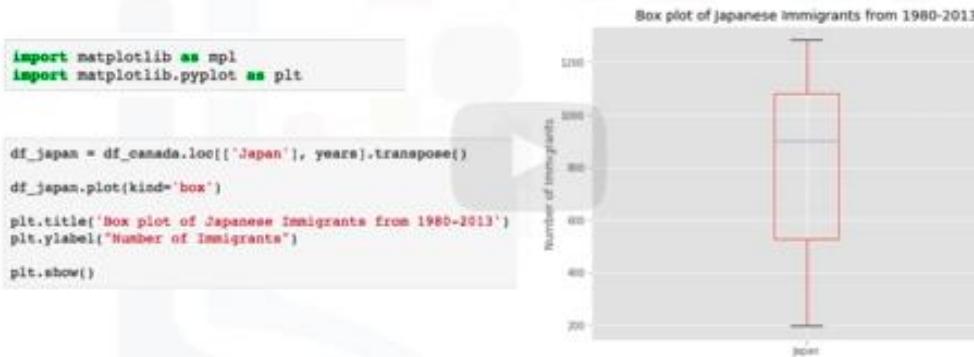
	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

Box Plots

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

df_japan = df_canada.loc[['Japan'], years].transpose()
df_japan.plot(kind='box')
plt.title('Box plot of Japanese Immigrants from 1980-2013')
plt.ylabel("Number of Immigrants")
plt.show()
```

Box Plots



Scatter Plots

We will learn about an additional visualization tool: the scatter plot, and we will learn how to create it using Matplotlib. So what is a scatter plot? A scatter plot is a type of plot that displays values pertaining to typically two variables against each other. Usually it is a dependent variable to be plotted against an independent variable in order to determine if any correlation between the two variables exists.

For example, here is a scatter plot of income versus education and by looking at the plotted data one can conclude that an individual with more years of education is likely to earn a higher income than an individual with fewer years of education. So how can we create a scatterplot with Matplotlib? Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

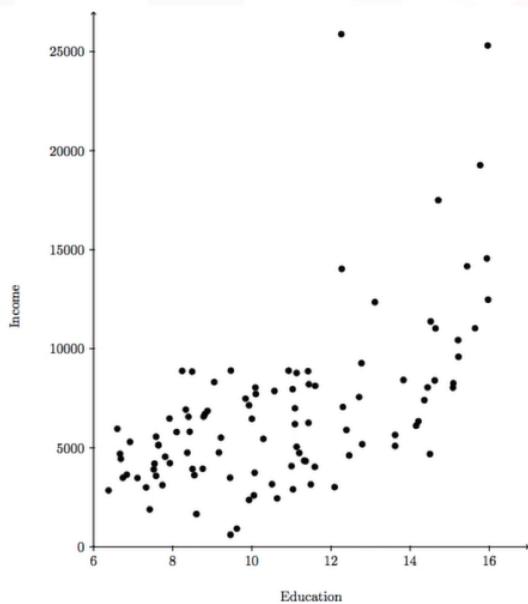
Now let's process the dataframe so that the country name becomes the index of each row. This should make retrieving rows pertaining to specific countries a lot easier. Also let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan for example, it is 58,639, total, and for Albania it is 15,699 and so on. And let's name our dataframe df_canada. So now that we know how our data is stored in the dataframe, df_canada, say we're interested in plotting a scatter plot of the total annual immigration to Canada from 1980 to 2013. To be able to do that, we first need to create a new dataframe that shows each year and the corresponding total number of immigration from all the countries worldwide as shown here. Let's name this new dataframe, df_total. In the lab session, we will walk together through the process of creating df_total from df_canada, so make sure to complete this module's lab session. Then we proceed as usual. We import Matplotlib as mpl and its scripting layer, the pyplot interface, as plt. Then we call the plot function on the data frame df_total and we set kind equals scatter to generate a scatter plot.

Now unlike the other data visualization tools we're only passing the kind parameter was enough to generate the plot, with scatter plots we also need to pass the variable to be plotted on the horizontal axis as the x-parameter and the variable to be plotted on the vertical axis as the y-parameter. In this case, we're passing column year as the x-parameter and column total as the y-parameter. Then to complete the figure we give it a title and we label its axes. Finally, we call the show function to display the figure. And there you have it. A scatter plot that shows total immigration to Canada from countries all over the world from 1980 to 2013. The scatter plot clearly depicts an overall rising trend of immigration with time. In the lab session we explore scatter plots in more details and learn about a very interesting variation of this scatter plot, a plot called the bubble plot, and we learn how to create it using Matplotlib

Data Visualization with Python

Scatter Plots

Scatter Plots



DATA COGNITIVE

Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	1	15

Scatter plot that shows total immigration to Canada from countries all over the world from 1980 to 2013

Scatter Plots

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

df_total.plot(
    kind='scatter',
    x='year',
    y='total',
)
plt.title('Total Immigrant population to Canada from 1980 - 2013')
plt.xlabel ('Year')
plt.ylabel('Number of Immigrants')

plt.show()
```

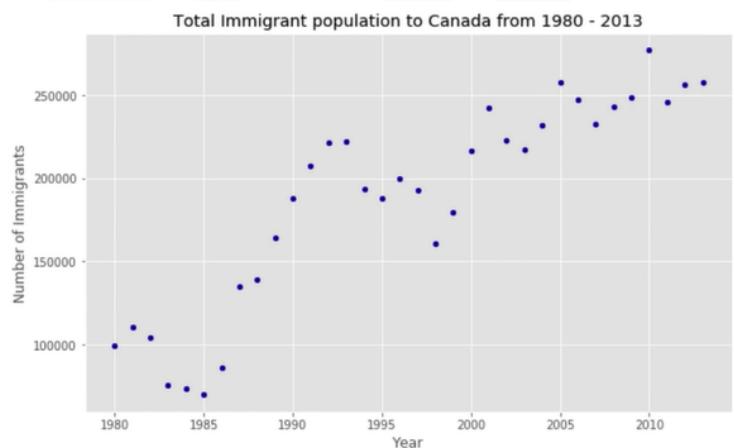
df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.
.	.

Scatter Plots

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

df_total.plot(
    kind='scatter',
    x='year',
    y='total',
)
plt.title('Total Immigrant population to Canada from 1980 - 2013')
plt.xlabel ('Year')
plt.ylabel('Number of Immigrants')

plt.show()
```





Pie Charts, Box Plots, Scatter Plots, and Bubble Plots

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Explore Matplotlib library further
- Create pie charts, box plots, scatter plots and bubble charts

Table of Contents

1. [Exploring Datasets with *pandas](#0)
2. [Downloading and Prepping Data](#2)
3. [Visualizing Data using Matplotlib](#4)
4. [Pie Charts](#6)
5. [Box Plots](#8)
6. [Scatter Plots](#10)
7. [Bubble Plots](#12)

Exploring Datasets with *pandas* and Matplotlib

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.

Downloading and Prepping Data

Import primary modules.

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

```
! pip3 install openpyxl
```

Download the dataset and read it into a pandas dataframe.

```
In [2]: df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2
)
print('Data downloaded and read into a dataframe!')
```

Data downloaded and read into a dataframe!

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

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

```
Out[3]:
```

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 43 columns

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

```
In [4]: # print the dimensions of the dataframe
print(df_can.shape)
```

(195, 43)

Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to *Introduction to Matplotlib and Line Plots and Area Plots, Histograms, and Bar Plots* for a detailed description of this preprocessing.

```
In [5]:
# clean up the dataset to remove unnecessary columns (eg. REG)
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)

# Let's rename the columns so that they make sense
df_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)

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

# set the country name as index - useful for quickly looking up countries using .loc method
df_can.set_index('Country', inplace=True)

# add total column
df_can['Total'] = df_can.sum(axis=1)

# years that we will be using in this lesson - useful for plotting later on
years = list(map(str, range(1980, 2014)))
print('data dimensions:', df_can.shape)

data dimensions: (195, 38)
```

Visualizing Data using Matplotlib

Import `Matplotlib`.

```
In [6]:
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt

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

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

Matplotlib version: 3.3.4

Pie Charts

A pie chart is a circular graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the `kind=pie` keyword.

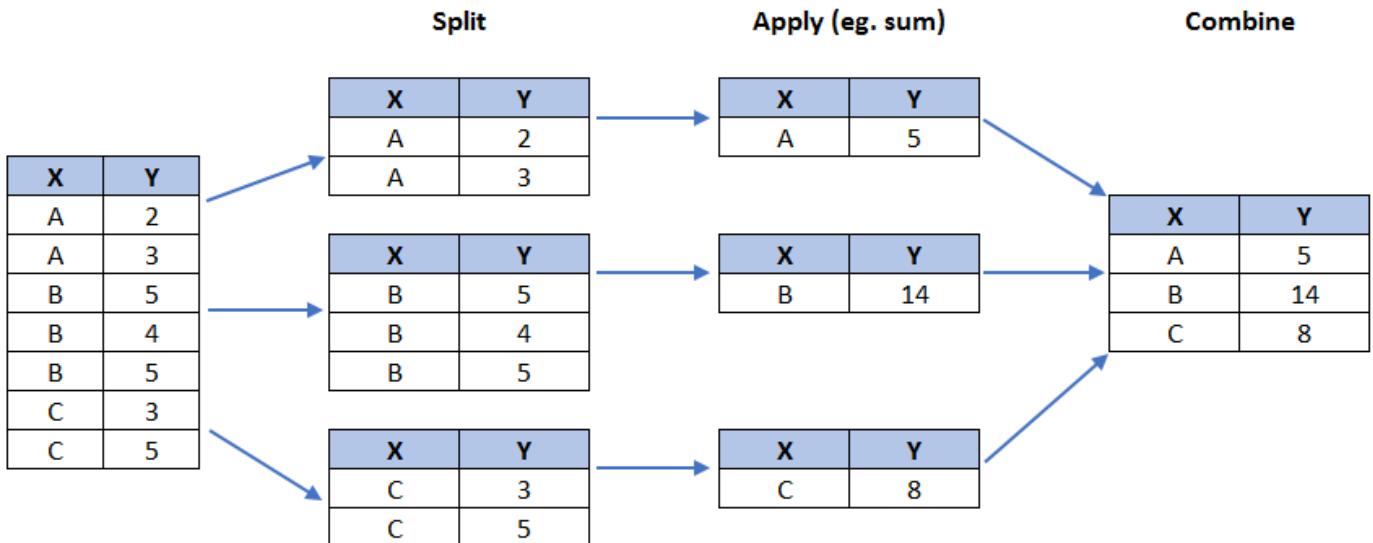
Let's use a pie chart to explore the proportion (percentage) of new immigrants grouped by continents for the entire time period from 1980 to 2013.

Step 1: Gather data.

We will use `pandas` groupby method to summarize the immigration data by Continent. The general process of groupby involves the following steps:

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

3. Combine: Combining the results into a data structure.



In [7]:

```
# group countries by continents and apply sum() function
df_continents = df_can.groupby('Continent', axis=0).sum()

# note: the output of the groupby method is a 'groupby' object.
# we can not use it further until we apply a function (eg .sum())
print(type(df_can.groupby('Continent', axis=0)))

df_continents.head()
```

pandas.core.groupby.generic.DataFrameGroupBy

```
Out[7]:
```

Continent	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total	
Africa	3951	4363	3819	2671	2639	2650	3782	7494	7552	9894	...	27523	29188	28284	29890	34534	40892	35441	38083	38543	618948
Asia	31025	34314	30214	24696	27274	23850	28739	43203	47454	60256	...	159253	149054	133459	139894	141434	163845	146894	152218	155075	3317794
Europe	39760	44802	42720	24638	22287	20844	24370	46698	54726	60893	...	35955	33053	33495	34692	35078	33425	26778	29177	28691	1410947
Latin America and the Caribbean	13081	15215	16769	15427	13678	15171	21179	28471	21924	25060	...	24747	24676	26011	26547	26867	28818	27856	27173	24950	765148
Northern America	9378	10030	9074	7100	6661	6543	7074	7705	6469	6790	...	8394	9613	9463	10190	8995	8142	7677	7892	8503	241142

5 rows × 35 columns

Step 2: Plot the data. We will pass in kind = 'pie' keyword, along with the following additional parameters:

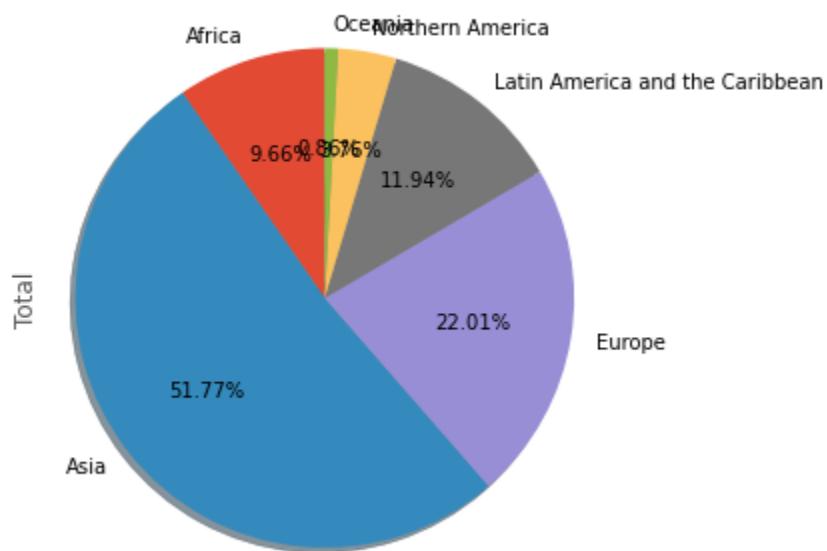
- `autopct` - is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%`.
- `startangle` - rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.
- `shadow` - Draws a shadow beneath the pie (to give a 3D feel).

```
In [8]: # autopct create %, start angle represent starting point
df_continents['Total'].plot(kind='pie',
                           figsize=(5, 6),
                           autopct='%1.2f%%', # add in percentages
                           startangle=90,      # start angle 90° (Africa)
                           shadow=True,        # add shadow
                           )

plt.title('Immigration to Canada by Continent [1980 - 2013]')
plt.axis('equal') # Sets the pie chart to look like a circle.

plt.show()
```

Immigration to Canada by Continent [1980 - 2013]



The above visual is not very clear, the numbers and text overlap in some instances. Let's make a few modifications to improve the visuals:

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

```
In [9]: colors_list = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue', 'lightgreen', 'pink']
explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each continent with which to offset each wedge.

df_continents['Total'].plot(kind='pie',
                            figsize=(15, 6),
                            autopct='%1.2f%%',
                            startangle=55,
                            shadow=True,
                            labels=None,           # turn off labels on pie chart
                            pctdistance=1.12,      # the ratio between the center of each pie slice and the start of the text generated by autopct
                            colors=colors_list,    # add custom colors
                            explode=explode_list) # 'explode' lowest 3 continents
)

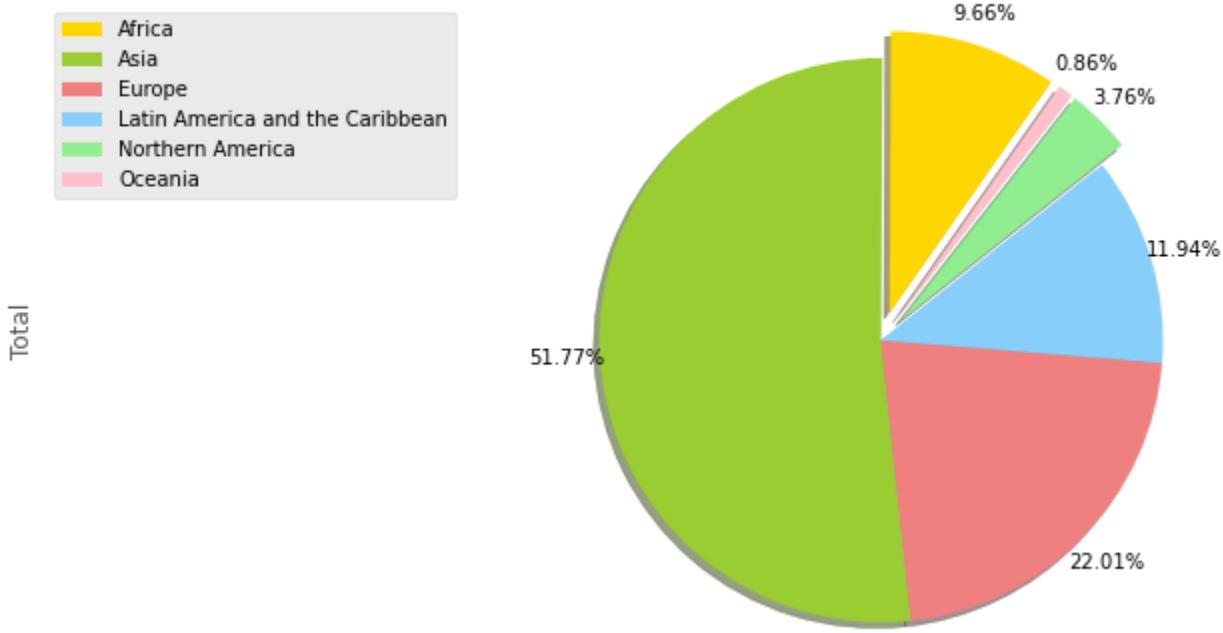
# scale the title up by 12% to match pctdistance
plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.12)

plt.axis('equal')

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

plt.show()
```

Immigration to Canada by Continent [1980 - 2013]



Question: Using a pie chart, explore the proportion (percentage) of new immigrants grouped by continents in the year 2013.

Note: You might need to play with the explode values in order to fix any overlapping slice values.

In

[10]:

```
explode_list = [0, 0, 0, 0.1, 0.1, 0.2]

df_continents['2013'].plot(kind='pie',
                           figsize=(15, 6),
                           autopct='%1.1f%%',
                           startangle=90,
                           shadow=True,
                           labels=None,
                           pctdistance=1.12,
                           explode=explode_list
                           )

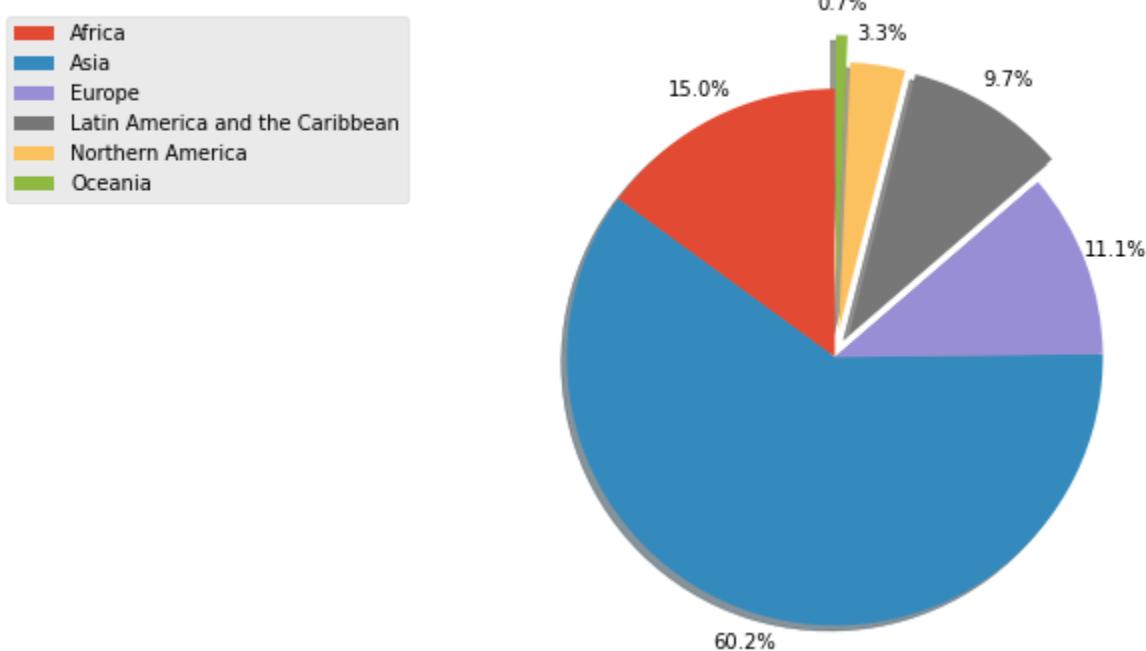
plt.title('Immigration to Canada by Continent in 2013', y=1.1)
plt.axis('equal')

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

plt.show()
```

Immigration to Canada by Continent in 2013

2013

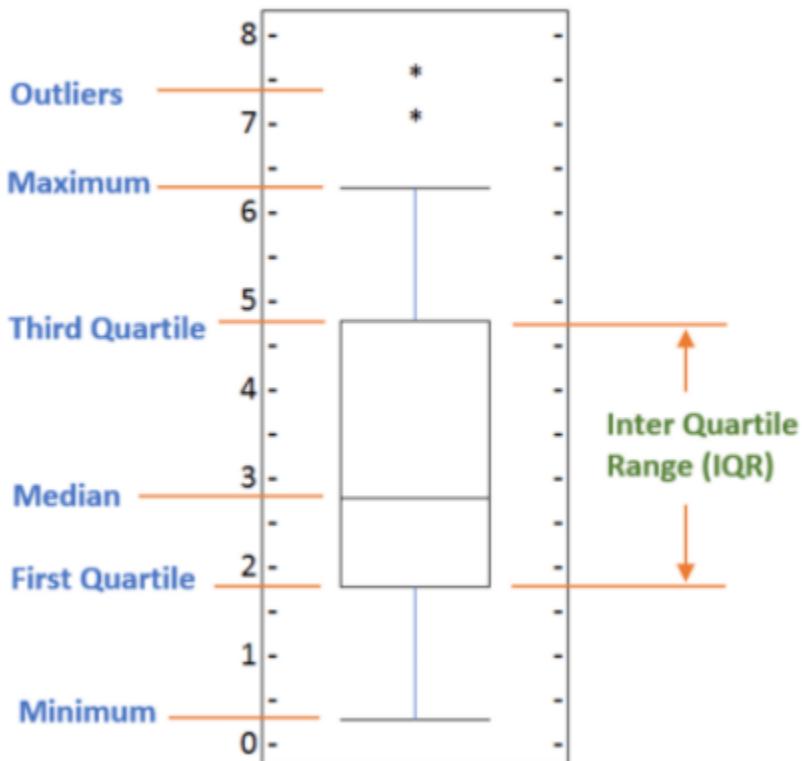


Click here for a sample python solution ````python #The correct answer is: explode_list = [0.0, 0, 0, 0.1, 0.1, 0.2] # ratio for each continent with which to offset each wedge. df_continents['2013'].plot(kind='pie', figsize=(15, 6), autopct='%1.1f%%', startangle=90, shadow=True, labels=None, # turn off labels on pie chart pctdistance=1.12, # the ratio between the pie center and start of text label explode=explode_list # 'explode' lowest 3 continents) # scale the title up by 12% to match pctdistance plt.title('Immigration to Canada by Continent in 2013', y=1.12) plt.axis('equal') # add legend plt.legend(labels=df_continents.index, loc='upper left') # show plot plt.show() ````

Box Plots

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

- **Minimum:** The smallest number in the dataset excluding the outliers.
- **First quartile:** Middle number between the minimum and the median.
- **Second quartile (Median):** Middle number of the (sorted) dataset.
- **Third quartile:** Middle number between median and maximum.
- **Maximum:** The largest number in the dataset excluding the outliers.



To make a boxplot, we can use kind=box in plot method invoked on a *pandas* series or dataframe.

Let's plot the box plot for the Japanese immigrants between 1980 - 2013.

Step 1: Get the subset of the dataset. Even though we are extracting the data for just one country, we will obtain it as a dataframe. This will help us with calling the `dataframe.describe()` method to view the percentiles.

```
In [11]: # to get a dataframe, place extra square brackets around 'Japan'.
df_japan = df_can.loc[['Japan'], years].transpose()
df_japan.head()
```

```
Out[11]: Country Japan
```

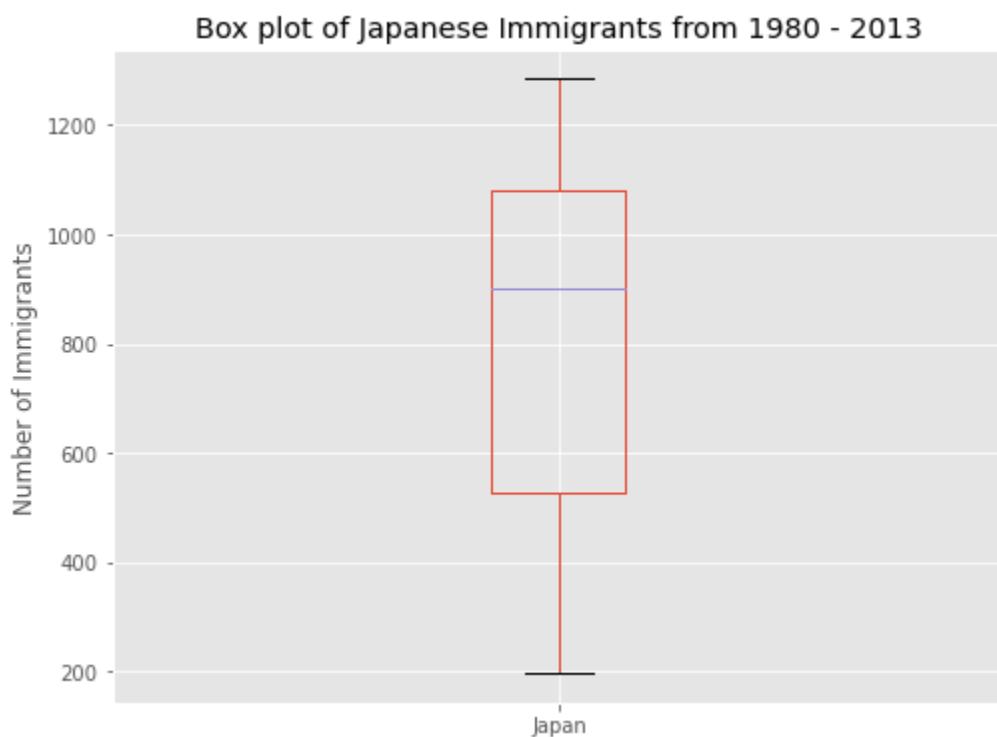
1980	701
1981	756
1982	598
1983	309
1984	246

Step 2: Plot by passing in `kind='box'`.

```
In [12]: df_japan.plot(kind='box', figsize=(8, 6))

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

plt.show()
```



We can immediately make a few key observations from the plot above:

1. The minimum number of immigrants is around 200 (min), maximum number is around 1300 (max), and median number of immigrants is around 900 (median).
2. 25% of the years for period 1980 - 2013 had an annual immigrant count of ~500 or fewer (First quartile).
3. 75% of the years for period 1980 - 2013 had an annual immigrant count of ~1100 or fewer (Third quartile).

We can view the actual numbers by calling the describe() method on the dataframe.

Out[13]:	Country	Japan
	count	34.000000
	mean	814.911765
	std	337.219771
	min	198.000000
	25%	529.000000
	50%	902.000000
	75%	1079.000000
	max	1284.000000

One of the key benefits of box plots is comparing the distribution of multiple datasets. In one of the previous labs, we observed that China and India had very similar immigration trends. Let's analyze these two countries further using box plots.

Question: Compare the distribution of the number of new immigrants from India and China for the period 1980 - 2013.

Step 1: Get the dataset for China and India and call the dataframe df_CI.

```
In [14]: df_CI = df_can.loc[['China', 'India'], years].transpose()
df_CI.head()
```

Out[14]:	Country	China	India
	1980	5123	8880
	1981	6682	8670
	1982	3308	8147
	1983	1863	7338
	1984	1527	5704

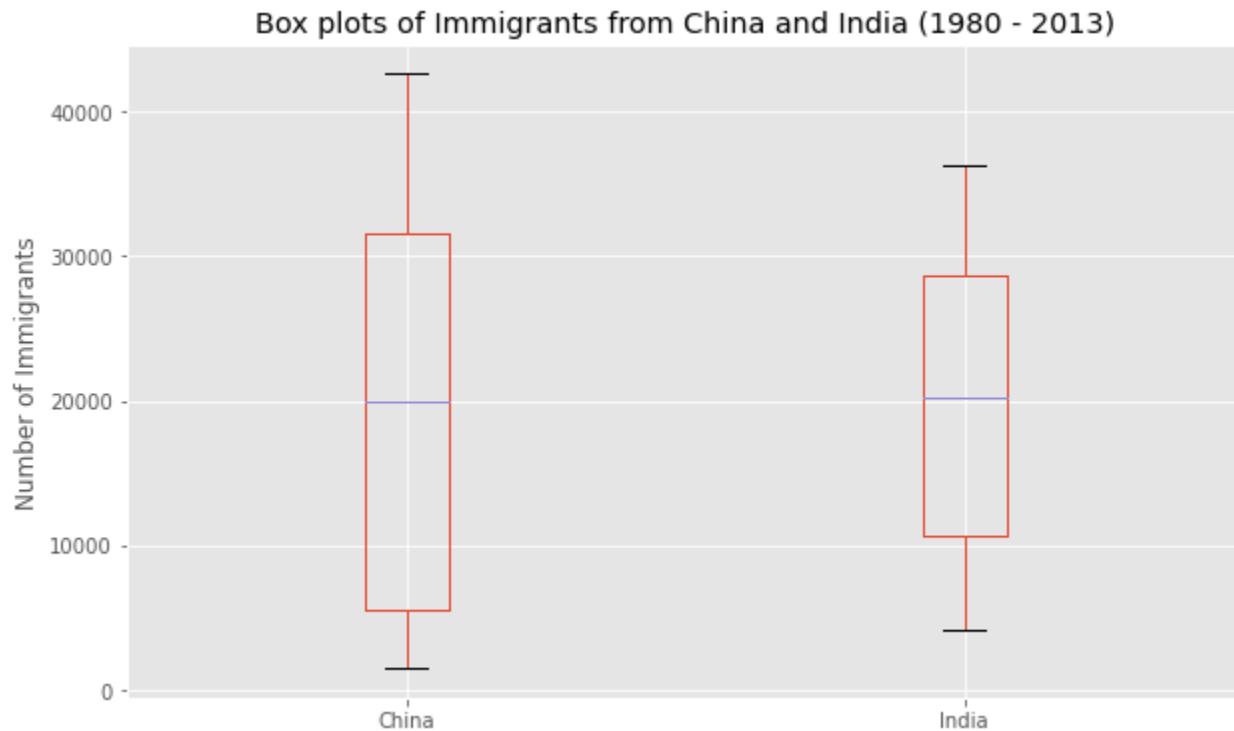
Click here for a sample python solution ``python #The correct answer is: df_CI.describe() ``

Step 2: Plot data.

```
In [16]: df_CI.plot(kind='box', figsize=(10, 6))

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

plt.show()
```



Click here for a sample python solution `python #The correct answer is: df_CI.plot(kind='box', figsize=(10, 7)) plt.title('Box plots of Immigrants from China and India (1980 - 2013)') plt.ylabel('Number of Immigrants') plt.show()`

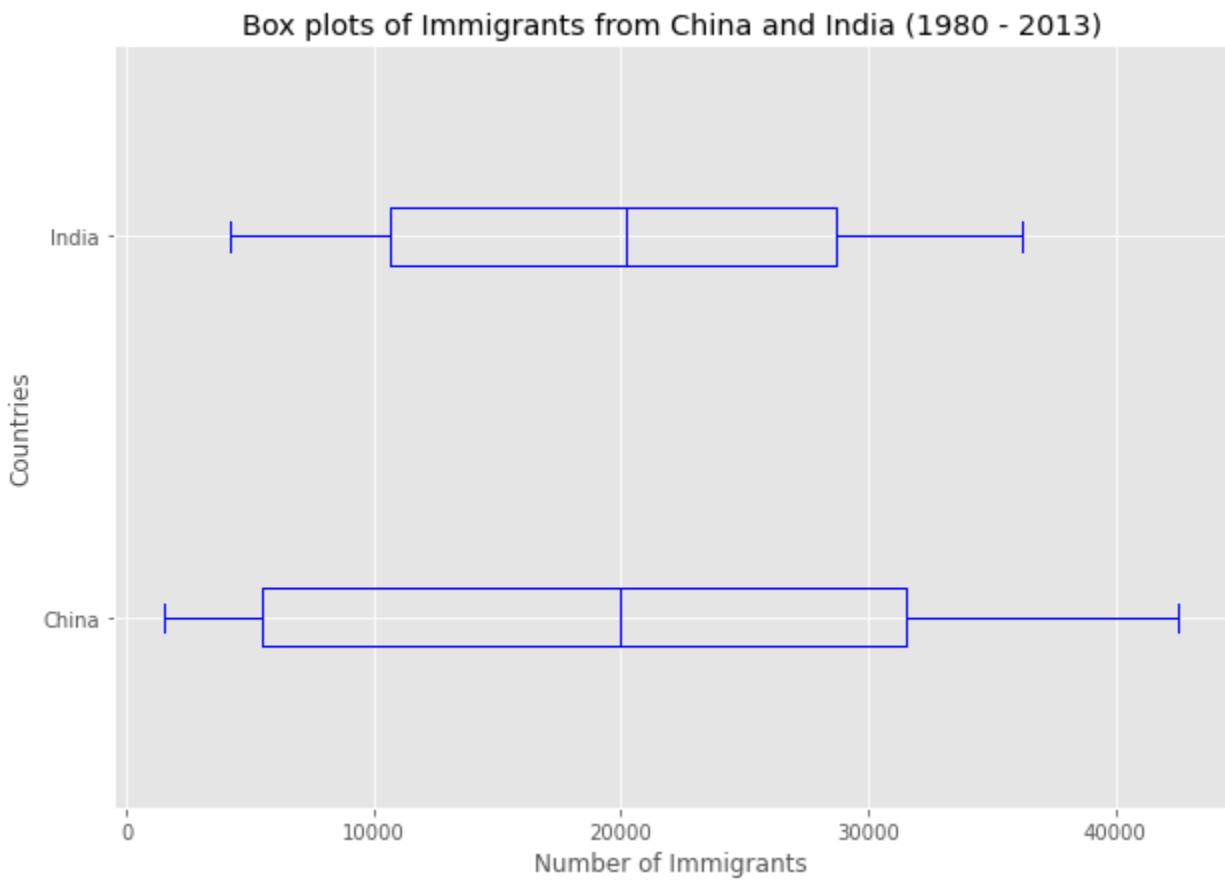
We can observe that, while both countries have around the same median immigrant population (~20,000), China's immigrant population range is more spread out than India's. The maximum population from India for any year (36,210) is around 15% lower than the maximum population from China (42,584).

If you prefer to create horizontal box plots, you can pass the `vert` parameter in the `plot` function and assign it to `False`. You can also specify a different color in case you are not a big fan of the default red color.

```
In [17]: # horizontal box plots
df_CI.plot(kind='box', figsize=(10, 7), color='blue', vert=False)

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

plt.show()
```



Subplots

Often times we might want to plot multiple plots within the same figure. For example, we might want to perform a side by side comparison of the box plot with the line plot of China and India's immigration.

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

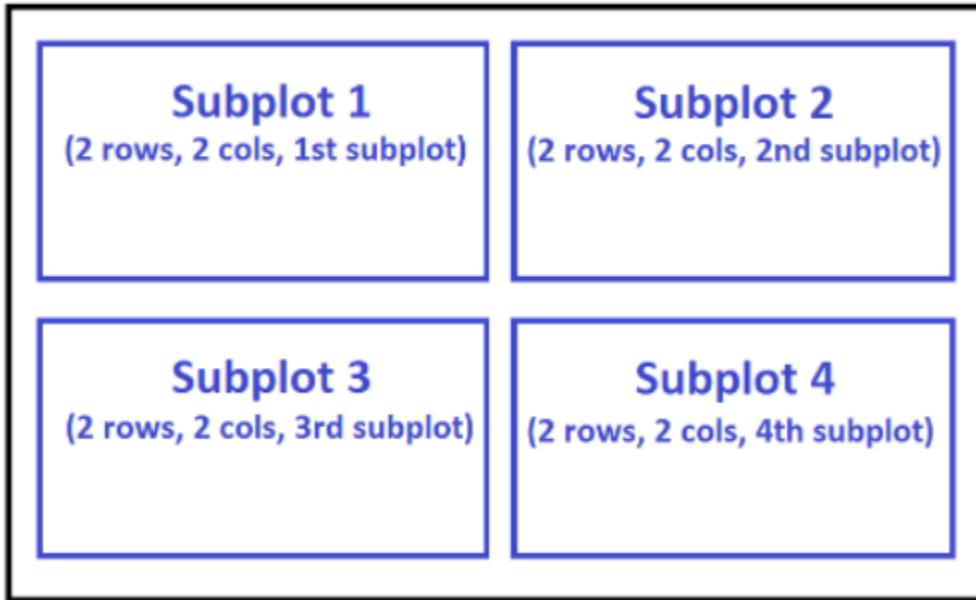
Typical syntax is :

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

Where

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

Figure



We can then specify which subplot to place each plot by passing in the `ax` parameter in `plot()` method as follows:

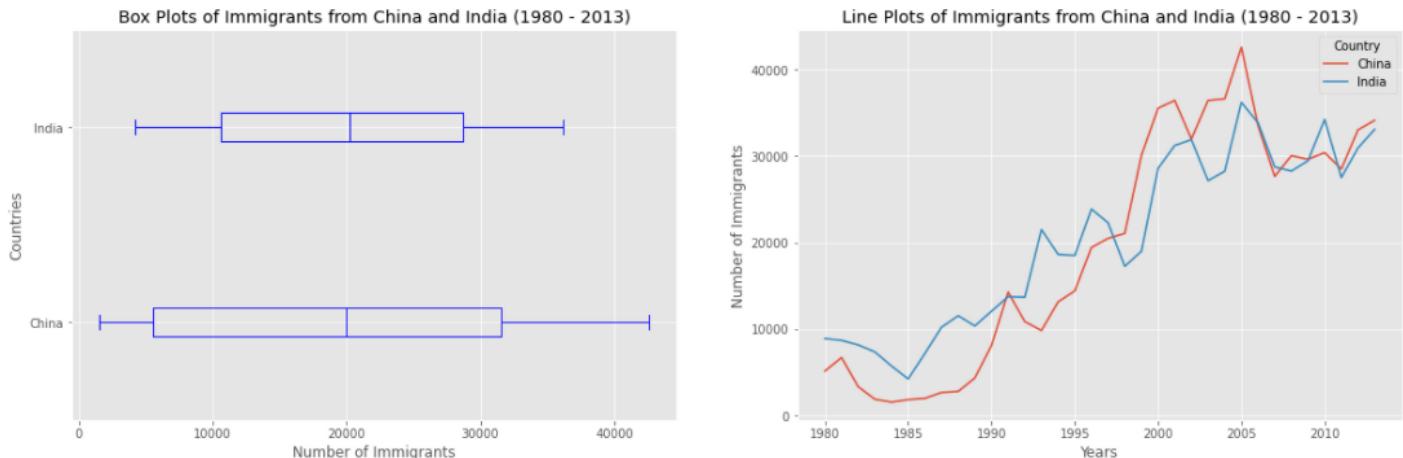
```
In [18]: fig = plt.figure() # create figure

ax0 = fig.add_subplot(1, 2, 1) # add subplot 1 (1 row, 2 columns, first plot)
ax1 = fig.add_subplot(1, 2, 2) # add subplot 2 (1 row, 2 columns, second plot). See tip below**

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

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

plt.show()
```



Tip regarding subplot convention

In the case when nrows, ncols, and plot_number are all less than 10, a convenience exists such that a 3-digit number can be given instead, where the hundreds represent nrows, the tens represent ncols and the units represent plot_number. For instance,

```
subplot(211) == subplot(2, 1, 1)
```

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 rows by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).

Let's try something a little more advanced.

Previously we identified the top 15 countries based on total immigration from 1980 - 2013.

Question: Create a box plot to visualize the distribution of the top 15 countries (based on total immigration) grouped by the decades 1980s, 1990s, and 2000s.

Step 1: Get the dataset. Get the top 15 countries based on Total immigrant population. Name the dataframe **df_top15**.

In [19]:	df_top15 = df_can.sort_values(['Total'], ascending=False, axis=0).head(15) df_top15																																																																																																																																																											
Out[19]:	<table border="1"> <thead> <tr> <th>Country</th><th>Continent</th><th>Region</th><th>DevName</th><th>1980</th><th>1981</th><th>1982</th><th>1983</th><th>1984</th><th>1985</th><th>1986</th><th>...</th><th>2005</th><th>2006</th><th>2007</th><th>2008</th><th>2009</th><th>2010</th><th>2011</th><th>2012</th><th>2013</th><th>Total</th></tr> </thead> <tbody> <tr> <td>India</td><td>Asia</td><td>Southern Asia</td><td>Developing regions</td><td>8880</td><td>8670</td><td>8147</td><td>7338</td><td>5704</td><td>4211</td><td>7150</td><td>...</td><td>36210</td><td>33848</td><td>28742</td><td>28261</td><td>29456</td><td>34235</td><td>27509</td><td>30933</td><td>33087</td><td>691904</td></tr> <tr> <td>China</td><td>Asia</td><td>Eastern Asia</td><td>Developing regions</td><td>5123</td><td>6682</td><td>3308</td><td>1863</td><td>1527</td><td>1816</td><td>1960</td><td>...</td><td>42584</td><td>33518</td><td>27642</td><td>30037</td><td>29622</td><td>30391</td><td>28502</td><td>33024</td><td>34129</td><td>659962</td></tr> <tr> <td>United Kingdom of Great Britain and Northern Ireland</td><td>Europe</td><td>Northern Europe</td><td>Developed regions</td><td>22045</td><td>24796</td><td>20620</td><td>10015</td><td>10170</td><td>9564</td><td>9470</td><td>...</td><td>7258</td><td>7140</td><td>8216</td><td>8979</td><td>8876</td><td>8724</td><td>6204</td><td>6195</td><td>5827</td><td>551500</td></tr> <tr> <td>Philippines</td><td>Asia</td><td>South-Eastern Asia</td><td>Developing regions</td><td>6051</td><td>5921</td><td>5249</td><td>4562</td><td>3801</td><td>3150</td><td>4166</td><td>...</td><td>18139</td><td>18400</td><td>19837</td><td>24887</td><td>28573</td><td>38617</td><td>36765</td><td>34315</td><td>29544</td><td>511391</td></tr> <tr> <td>Pakistan</td><td>Asia</td><td>Southern Asia</td><td>Developing regions</td><td>978</td><td>972</td><td>1201</td><td>900</td><td>668</td><td>514</td><td>691</td><td>...</td><td>14314</td><td>13127</td><td>10124</td><td>8994</td><td>7217</td><td>6811</td><td>7468</td><td>11227</td><td>12603</td><td>241600</td></tr> </tbody> </table>																								Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total	India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210	33848	28742	28261	29456	34235	27509	30933	33087	691904	China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1960	...	42584	33518	27642	30037	29622	30391	28502	33024	34129	659962	United Kingdom of Great Britain and Northern Ireland	Europe	Northern Europe	Developed regions	22045	24796	20620	10015	10170	9564	9470	...	7258	7140	8216	8979	8876	8724	6204	6195	5827	551500	Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	18139	18400	19837	24887	28573	38617	36765	34315	29544	511391	Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314	13127	10124	8994	7217	6811	7468	11227	12603	241600
Country	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total																																																																																																																																							
India	Asia	Southern Asia	Developing regions	8880	8670	8147	7338	5704	4211	7150	...	36210	33848	28742	28261	29456	34235	27509	30933	33087	691904																																																																																																																																							
China	Asia	Eastern Asia	Developing regions	5123	6682	3308	1863	1527	1816	1960	...	42584	33518	27642	30037	29622	30391	28502	33024	34129	659962																																																																																																																																							
United Kingdom of Great Britain and Northern Ireland	Europe	Northern Europe	Developed regions	22045	24796	20620	10015	10170	9564	9470	...	7258	7140	8216	8979	8876	8724	6204	6195	5827	551500																																																																																																																																							
Philippines	Asia	South-Eastern Asia	Developing regions	6051	5921	5249	4562	3801	3150	4166	...	18139	18400	19837	24887	28573	38617	36765	34315	29544	511391																																																																																																																																							
Pakistan	Asia	Southern Asia	Developing regions	978	972	1201	900	668	514	691	...	14314	13127	10124	8994	7217	6811	7468	11227	12603	241600																																																																																																																																							

United States of America	Northern America	Northern America	Developed regions	9378	10030	9074	7100	6661	6543	7074	...	8394	9613	9463	10190	8995	8142	7676	7891	8501	241122
Iran (Islamic Republic of)	Asia	Southern Asia	Developing regions	1172	1429	1822	1592	1977	1648	1794	...	5837	7480	6974	6475	6580	7477	7479	7534	11291	175923
Sri Lanka	Asia	Southern Asia	Developing regions	185	371	290	197	1086	845	1838	...	4930	4714	4123	4756	4547	4422	3309	3338	2394	148358
Republic of Korea	Asia	Eastern Asia	Developing regions	1011	1456	1572	1081	847	962	1208	...	5832	6215	5920	7294	5874	5537	4588	5316	4509	142581
Poland	Europe	Eastern Europe	Developed regions	863	2930	5881	4546	3588	2819	4808	...	1405	1263	1235	1267	1013	795	720	779	852	139241
Lebanon	Asia	Western Asia	Developing regions	1409	1119	1159	789	1253	1683	2576	...	3709	3802	3467	3566	3077	3432	3072	1614	2172	115359
France	Europe	Western Europe	Developed regions	1729	2027	2219	1490	1169	1177	1298	...	4429	4002	4290	4532	5051	4646	4080	6280	5623	109091
Jamaica	Latin America and the Caribbean	Caribbean	Developing regions	3198	2634	2661	2455	2508	2938	4649	...	1945	1722	2141	2334	2456	2321	2059	2182	2479	106431
Viet Nam	Asia	South-Eastern Asia	Developing regions	1191	1829	2162	3404	7583	5907	2741	...	1852	3153	2574	1784	2171	1942	1723	1731	2112	97146
Romania	Europe	Eastern Europe	Developed regions	375	438	583	543	524	604	656	...	5048	4468	3834	2837	2076	1922	1776	1588	1512	93585

15 rows × 38 columns

Click here for a sample python solution ````python #The correct answer is: df_top15 = df_can.sort_values(['Total'], ascending=False, axis=0).head(15) df_top15 ````

Step 2: Create a new dataframe which contains the aggregate for each decade. One way to do that:

1. Create a list of all years in decades 80's, 90's, and 00's.
2. Slice the original dataframe df_can to create a series for each decade and sum across all years for each country.
3. Merge the three series into a new data frame. Call your dataframe **new_df**.

In [20]:

```
# create a list of all years in decades 80's, 90's, and 00's
years_80s = list(map(str, range(1980, 1990)))
years_90s = list(map(str, range(1990, 2000)))
years_00s = list(map(str, range(2000, 2010)))

# slice the original dataframe df_can to create a series for each decade
df_80s = df_top15.loc[:, years_80s].sum(axis=1)
df_90s = df_top15.loc[:, years_90s].sum(axis=1)
df_00s = df_top15.loc[:, years_00s].sum(axis=1)

# merge the three series into a new data frame
new_df = pd.DataFrame({'1980s': df_80s, '1990s': df_90s, '2000s': df_00s})

# display dataframe
new_df.head()
```

Out[20]:

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

Click here for a sample python solution [``python #The correct answer is: # create a list of all years in decades 80's, 90's, and 00's
years_80s = list\(map\(str, range\(1980, 1990\)\)\) years_90s = list\(map\(str, range\(1990, 2000\)\)\) years_00s = list\(map\(str, range\(2000, 2010\)\)\) # slice the original dataframe df_can to create a series for each decade df_80s = df_top15.loc\[:, years_80s\].sum\(axis=1\)
df_90s = df_top15.loc\[:, years_90s\].sum\(axis=1\) df_00s = df_top15.loc\[:, years_00s\].sum\(axis=1\) # merge the three series into a new data frame new_df = pd.DataFrame\({'1980s': df_80s, '1990s': df_90s, '2000s':df_00s}\) # display dataframe new_df.head\(\) ````](#)

Let's learn more about the statistics associated with the dataframe using the describe() method.

In [21]:

```
new_df.describe()
```

Out[21]:

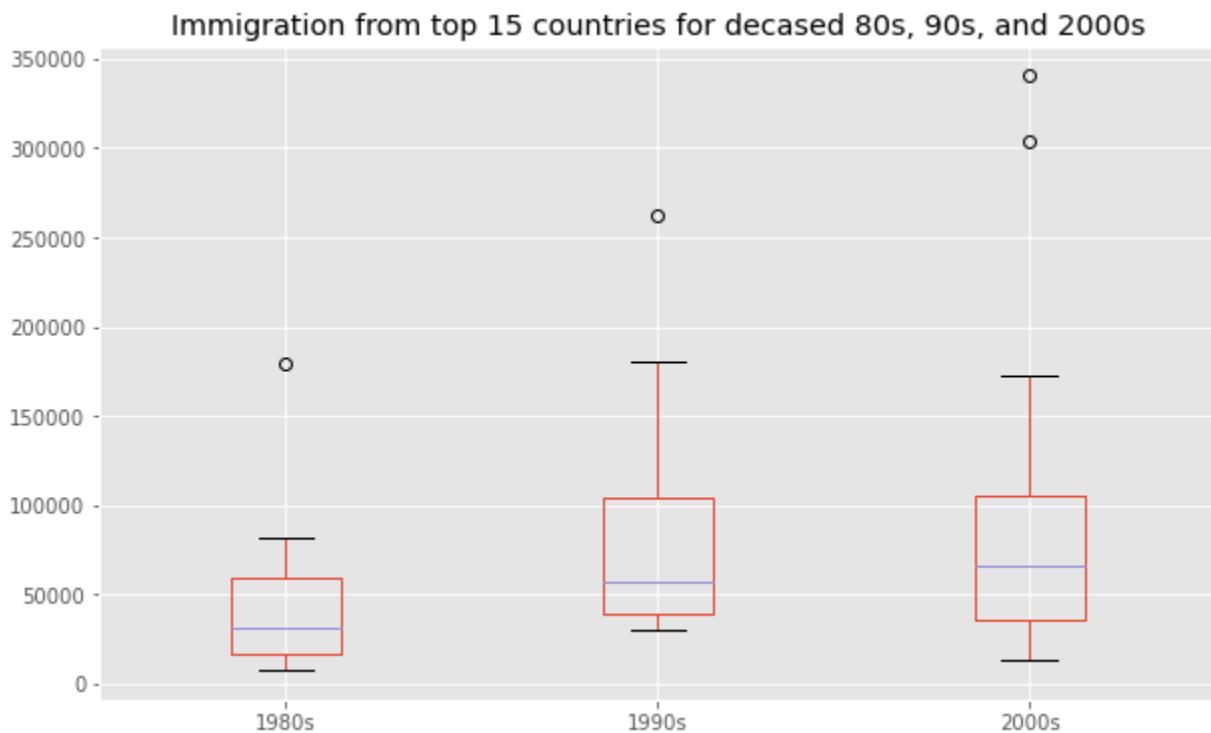
	1980s	1990s	2000s
count	15.000000	15.000000	15.000000
mean	44418.333333	85594.666667	97471.533333
std	44190.676455	68237.560246	100583.204205
min	7613.000000	30028.000000	13629.000000
25%	16698.000000	39259.000000	36101.500000
50%	30638.000000	56915.000000	65794.000000
75%	59183.000000	104451.500000	105505.500000
max	179171.000000	261966.000000	340385.000000

In [22]:

```
new_df.plot(kind='box', figsize=(10, 6))

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

plt.show()
```



Click here for a sample python solution ````python #The correct answer is: new_df.plot(kind='box', figsize=(10, 6)) plt.title('Immigration from top 15 countries for decades 80s, 90s and 2000s') plt.show() ````

Note how the box plot differs from the summary table created. The box plot scans the data and identifies the outliers. In order to be an outlier, the data value must be:

- larger than Q3 by at least 1.5 times the interquartile range (IQR), or,
- smaller than Q1 by at least 1.5 times the IQR.

Let's look at decade 2000s as an example:

- Q1 (25%) = 36,101.5
- Q3 (75%) = 105,505.5
- IQR = Q3 - Q1 = 69,404

Using the definition of outlier, any value that is greater than Q3 by 1.5 times IQR will be flagged as outlier.

Outlier > $105,505.5 + (1.5 * 69,404)$

Outlier > 209,611.5

```
In [23]: new_df = new_df.reset_index()
new_df[new_df['2000s'] > 209611.5]
```

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

Click here for a sample python solution ````python #The correct answer is: new_df=new_df.reset_index()
new_df[new_df['2000s']> 209611.5] ````
China and India are both considered as outliers since their population for the decade exceeds 209,611.5.

The box plot is an advanced visualizaiton tool, and there are many options and customizations that exceed the scope of this lab. Please refer to [Matplotlib documentation](#) on box plots for more information.

Scatter Plots

A scatter plot (2D) is a useful method of comparing variables against each other. Scatter plots look similar to line plots in that they both map independent and dependent variables on a 2D graph. While the data points are connected together by a line in a line plot, they are not connected in a scatter plot. The data in a scatter plot is considered to express a trend. With further analysis using tools like regression, we can mathematically calculate this relationship and use it to predict trends outside the dataset.

Let's start by exploring the following:

Using a scatter plot, let's visualize the trend of total immigrantion to Canada (all countries combined) for the years 1980 - 2013.

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

```
In [24]: # we can use the sum() method to get the total population per year  
df_tot = pd.DataFrame(df_can[years].sum(axis=0))  
  
# change the years to type int (useful for regression later on)  
df_tot.index = map(int, df_tot.index)  
  
# reset the index to put in back in as a column in the df_tot dataframe  
df_tot.reset_index(inplace = True)  
  
# rename columns  
df_tot.columns = ['year', 'total']  
  
# view the final dataframe  
df_tot.head()
```

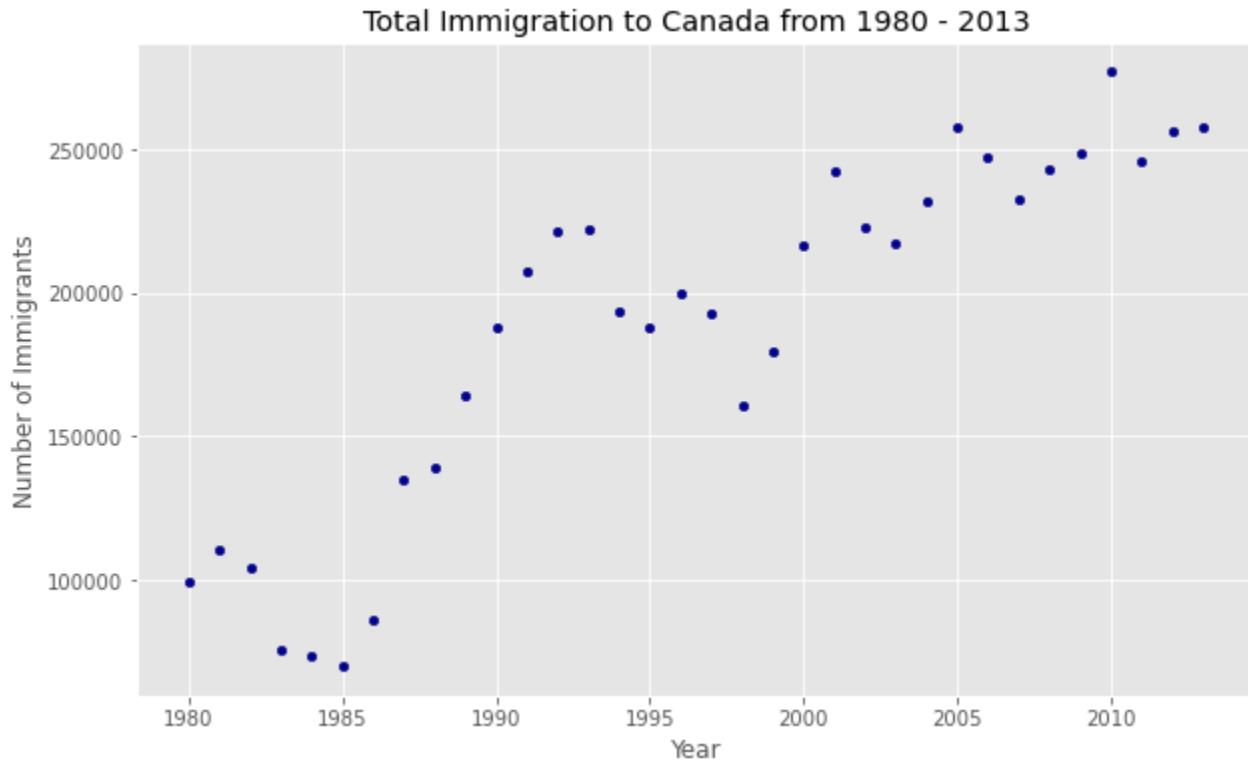
```
Out[24]:    year    total  
0   1980    99137  
1   1981   110563  
2   1982   104271  
3   1983    75550  
4   1984    73417
```

Step 2: Plot the data. In Matplotlib, we can create a scatter plot set by passing in kind='scatter' as plot argument. We will also need to pass in x and y keywords to specify the columns that go on the x- and the y-axis.

```
In [25]: df_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')

plt.title('Total Immigration to Canada from 1980 - 2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')

plt.show()
```



Notice how the scatter plot does not connect the data points together. We can clearly observe an upward trend in the data: as the years go by, the total number of immigrants increases. We can mathematically analyze this upward trend using a regression line (line of best fit).

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

Step 1: Get the equation of line of best fit. We will use **Numpy**'s `polyfit()` method by passing in the following:

- x: x-coordinates of the data.
- y: y-coordinates of the data.
- deg: Degree of fitting polynomial. 1 = linear, 2 = quadratic, and so on.

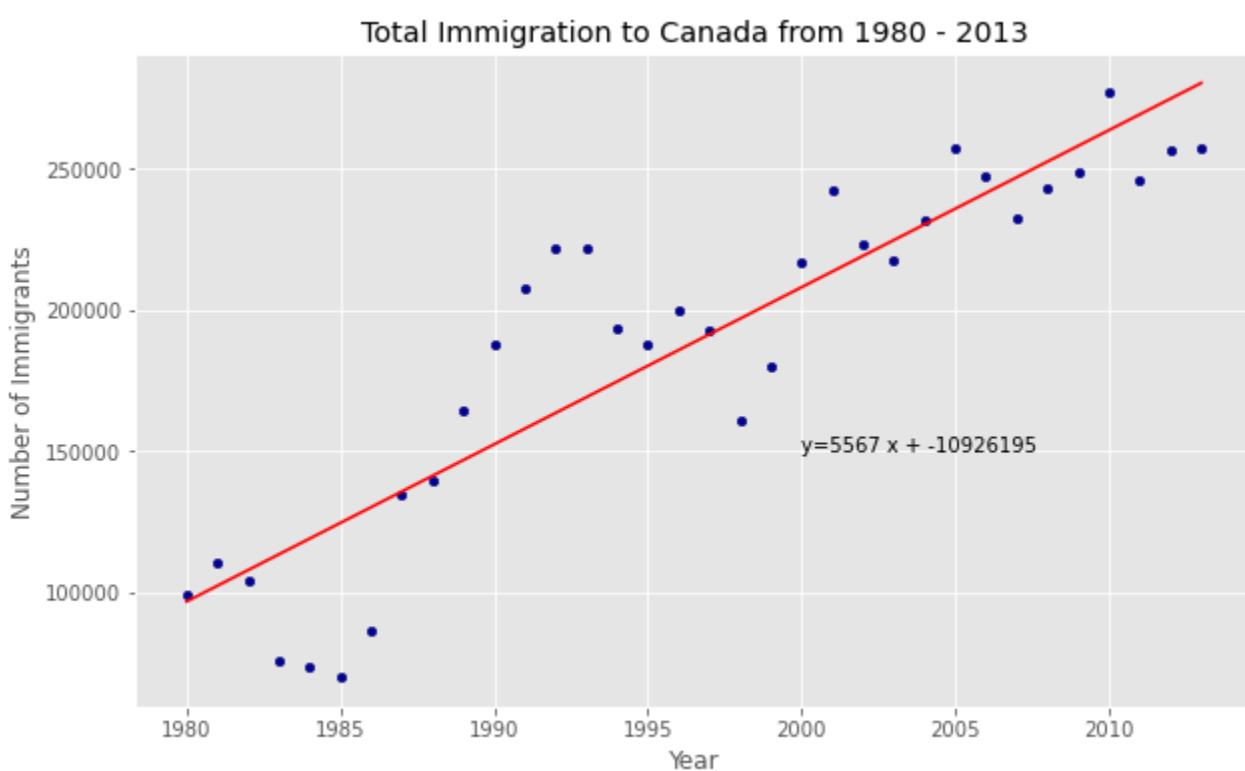
```
In [26]:  
x = df_tot['year']      # year on x-axis  
y = df_tot['total']     # total on y-axis  
fit = np.polyfit(x, y, deg=1)  
  
fit
```

```
Out[26]: array([ 5.56709228e+03, -1.09261952e+07])
```

The output is an array with the polynomial coefficients, highest powers first. Since we are plotting a linear regression $y = a * x + b$, our output has 2 elements [5.56709228e+03, -1.09261952e+07] with the slope in position 0 and intercept in position 1.

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

```
In [27]:  
df_tot.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')  
  
plt.title('Total Immigration to Canada from 1980 - 2013')  
plt.xlabel('Year')  
plt.ylabel('Number of Immigrants')  
  
# plot line of best fit  
plt.plot(x, fit[0] * x + fit[1], color='red') # recall that x is the Years  
plt.annotate('y={0:.0f} x + {1:.0f}'.format(fit[0], fit[1]), xy=(2000, 150000))  
  
plt.show()  
  
# print out the line of best fit  
'No. Immigrants = {0:.0f} * Year + {1:.0f}'.format(fit[0], fit[1])
```



'No. Immigrants = 5567 * Year + -10926195'

Using the equation of line of best fit, we can estimate the number of immigrants in 2015:

$$\text{No. Immigrants} = 5567 * \text{Year} - 10926195$$

$$\text{No. Immigrants} = 5567 * 2015 - 10926195$$

$$\text{No. Immigrants} = 291,310$$

When compared to the actual from Citizenship and Immigration Canada's (CIC) [2016 Annual Report](#), we see that Canada accepted 271,845 immigrants in 2015. Our estimated value of 291,310 is within 7% of the actual number, which is pretty good considering our original data came from United Nations (and might differ slightly from CIC data).

As a side note, we can observe that immigration took a dip around 1993 - 1997. Further analysis into the topic revealed that in 1993 Canada introduced Bill C-86 which introduced revisions to the refugee determination system, mostly restrictive. Further amendments to the Immigration Regulations cancelled the sponsorship required for "assisted relatives" and reduced the points awarded to them, making it more difficult for family members (other than nuclear family) to immigrate to Canada. These restrictive measures had a direct impact on the immigration numbers for the next several years.

Question: Create a scatter plot of the total immigration from Denmark, Norway, and Sweden to Canada from 1980 to 2013?

Step 1: Get the data:

1. Create a dataframe that consists of the numbers associated with Denmark, Norway, and Sweden only. Name it `df_countries`.

2. Sum the immigration numbers across all three countries for each year and turn the result into a dataframe. Name this new dataframe **df_total**.
3. Reset the index in place.
4. Rename the columns to **year** and **total**.
5. Display the resulting dataframe.

Out[27]: `'No. Immigrants = 5567 * Year + -10926195'`

In [28]:

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

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

# reset index in place
df_total.reset_index(inplace=True)

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

# change column year from string to int to create scatter plot
df_total['year'] = df_total['year'].astype(int)

# show resulting dataframe
df_total.head()
```

Out[28]:

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

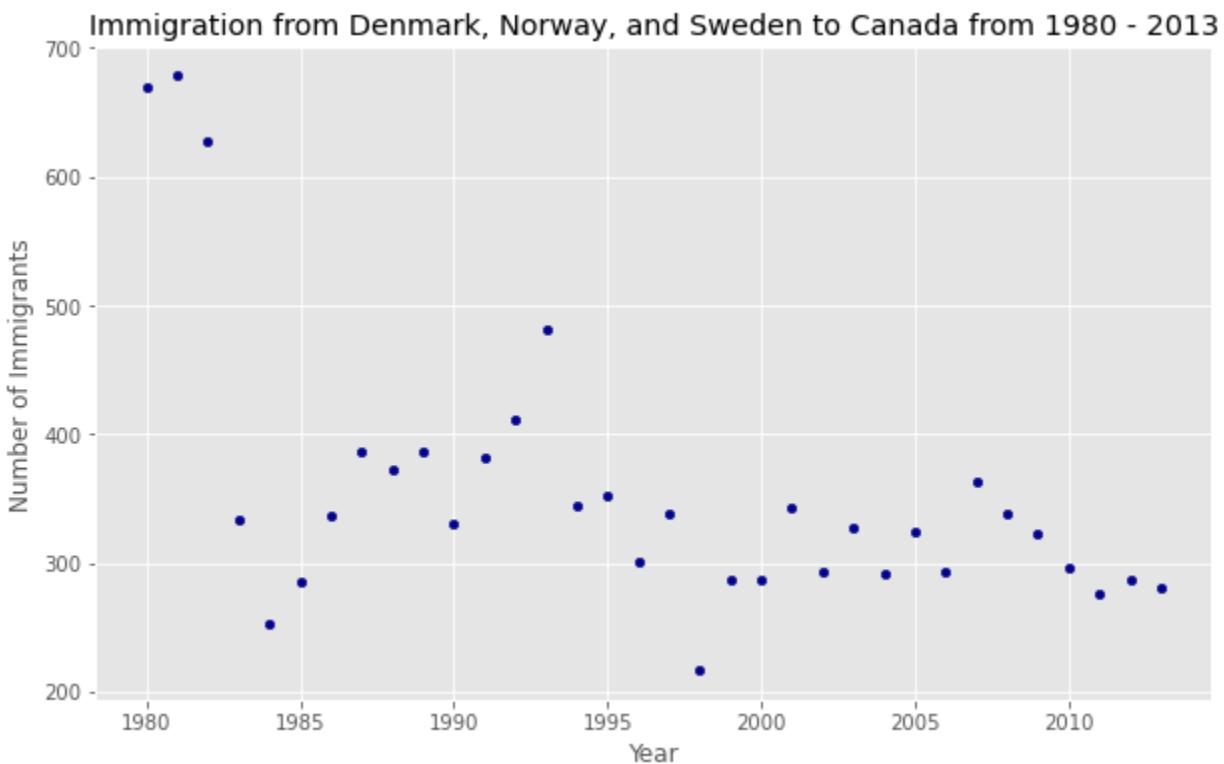
Click here for a sample python solution `'''python #The correct answer is: # create df_countries dataframe df_countries = df_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose() # create df_total by summing across three countries for each year df_total = pd.DataFrame(df_countries.sum(axis=1)) # reset index in place df_total.reset_index(inplace=True) # rename columns df_total.columns = ['year', 'total'] # change column year from string to int to create scatter plot df_total['year'] = df_total['year'].astype(int) # show resulting dataframe df_total.head()'''`

Step 2: Generate the scatter plot by plotting the total versus year in **df_total**.

```
In [29]: # generate scatter plot
df_total.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue')

# add title and label to axes
plt.title('Immigration from Denmark, Norway, and Sweden to Canada from 1980 - 2013')
plt.xlabel('Year')
plt.ylabel('Number of Immigrants')

# show plot
plt.show()
```



Click here for a sample python solution ``python #The correct answer is:
`df_total.plot(kind='scatter', x='year', y='total', figsize=(10, 6), color='darkblue') # add title and label to axes
plt.title('Immigration from Denmark, Norway, and Sweden to Canada from 1980 - 2013') plt.xlabel('Year')
plt.ylabel('Number of Immigrants') # show plot plt.show()```

Bubble Plots

A **bubble plot** is a variation of the scatter plot that displays three dimensions of data (x, y, z). The data points are replaced with bubbles, and the size of the bubble is determined by the third variable z, also known as the weight. In matplotlib, we can pass in an array or scalar to the parameter s to plot(), that contains the weight of each point.

Let's start by analyzing the effect of Argentina's great depression.

Argentina suffered a great depression from 1998 to 2002, which caused widespread unemployment, riots, the fall of the government, and a default on the country's foreign debt. In terms of income, over 50% of Argentines were poor, and seven out of ten Argentine children were poor at the depth of the crisis in 2002.

Let's analyze the effect of this crisis, and compare Argentina's immigration to that of its neighbour Brazil. Let's do that using a bubble plot of immigration from Brazil and Argentina for the years 1980 - 2013. We will set the weights for the bubble as the *normalized* value of the population for each year.

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

```
In [30]: # transposed dataframe
df_can_t = df_can[years].transpose()

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

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

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

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

Out[30]:

Country	Year	Afghanistan	Albania	Algeria	American Samoa	Andorra	Angola	Antigua and Barbuda	Argentina	Armenia	...	United States of America	Uruguay	Uzbekistan	Vanuatu
0	1980	16	1	80	0	0	1	0	368	0	...	9378	128	0	0
1	1981	39	0	67	1	0	3	0	426	0	...	10030	132	0	0
2	1982	39	0	71	0	0	6	0	626	0	...	9074	146	0	0
3	1983	47	0	69	0	0	6	0	241	0	...	7100	105	0	0
4	1984	71	0	63	0	0	4	42	237	0	...	6661	90	0	0

5 rows × 196 columns

Venezuela (Bolivarian Republic of)	Viet Nam	Western Sahara	Yemen	Zambia	Zimbabwe
103	1191	0	1	11	72
117	1829	0	2	17	114
174	2162	0	1	11	102
124	3404	0	6	7	44
142	7583	0	0	16	32

Step 2: Create the normalized weights.

There are several methods of normalizations in statistics, each with its own use. In this case, we will use [feature scaling](#) to bring all values into the range [0, 1]. The general formula is:

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

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

```
In [31]: # normalize Brazil data
norm_brazil = (df_can_t['Brazil'] - df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max() - df_can_t['Brazil'].min())

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

Step 3: Plot the data.

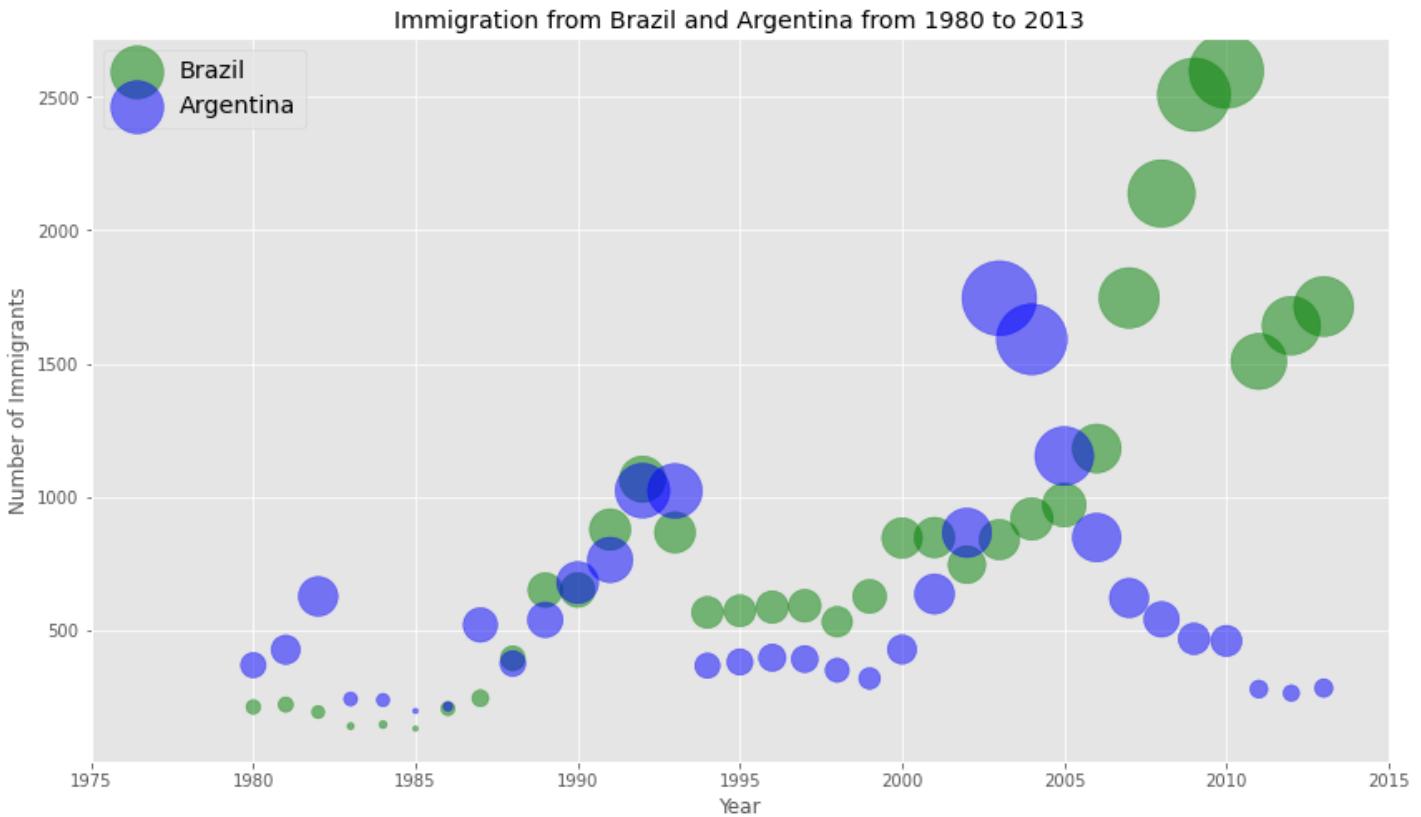
- To plot two different scatter plots in one plot, we can include the axes one plot into the other by passing it via the ax parameter.
- We will also pass in the weights using the s parameter. Given that the normalized weights are between 0-1, they won't be visible on the plot. Therefore, we will:
 - multiply weights by 2000 to scale it up on the graph, and,
 - add 10 to compensate for the min value (which has a 0 weight and therefore scale with $\times 2000$).

```
In [32]: # Brazil
ax0 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='Brazil',
                     figsize=(14, 8),
                     alpha=0.5, # transparency
                     color='green',
                     s=norm_brazil * 2000 + 10, # pass in weights
                     xlim=(1975, 2015)
                    )

# Argentina
ax1 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='Argentina',
                     alpha=0.5,
                     color="blue",
                     s=norm_argentina * 2000 + 10,
                     ax=ax0
                    )

ax0.set_ylabel('Number of Immigrants')
ax0.set_title('Immigration from Brazil and Argentina from 1980 to 2013')
ax0.legend(['Brazil', 'Argentina'], loc='upper left', fontsize='x-large')
```

Out[32]: <matplotlib.legend.Legend at 0x7f6aa1e46748>



The size of the bubble corresponds to the magnitude of immigrating population for that year, compared to the 1980 - 2013 data. The larger the bubble is, the more immigrants are in that year.

From the plot above, we can see a corresponding increase in immigration from Argentina during the 1998 - 2002 great depression. We can also observe a similar spike around 1985 to 1993. In fact, Argentina had suffered a great depression from 1974 to 1990, just before the onset of 1998 - 2002 great depression.

On a similar note, Brazil suffered the *Samba Effect* where the Brazilian real (currency) dropped nearly 35% in 1999. There was a fear of a South American financial crisis as many South American countries were heavily dependent on industrial exports from Brazil. The Brazilian government subsequently adopted an austerity program, and the economy slowly recovered over the years, culminating in a surge in 2010. The immigration data reflect these events.

Question: Previously in this lab, we created box plots to compare immigration from China and India to Canada. Create bubble plots of immigration from China and India to visualize any differences with time from 1980 to 2013. You can use `df_can_t` that we defined and used in the previous example.

Step 1: Normalize the data pertaining to China and India.

```
In [33]: # normalized Chinese data
norm_china = (df_can_t['China'] - df_can_t['China'].min()) / (df_can_t['China'].max() - df_can_t['China'].min())
# normalized Indian data
norm_india = (df_can_t['India'] - df_can_t['India'].min()) / (df_can_t['India'].max() - df_can_t['India'].min())
```

Click here for a sample python solution ````python #The correct answer is: # normalized Chinese data norm_china = (df_can_t['China'] - df_can_t['China'].min()) / (df_can_t['China'].max() - df_can_t['China'].min()) # normalized Indian data norm_india = (df_can_t['India'] - df_can_t['India'].min()) / (df_can_t['India'].max() - df_can_t['India'].min()) ````

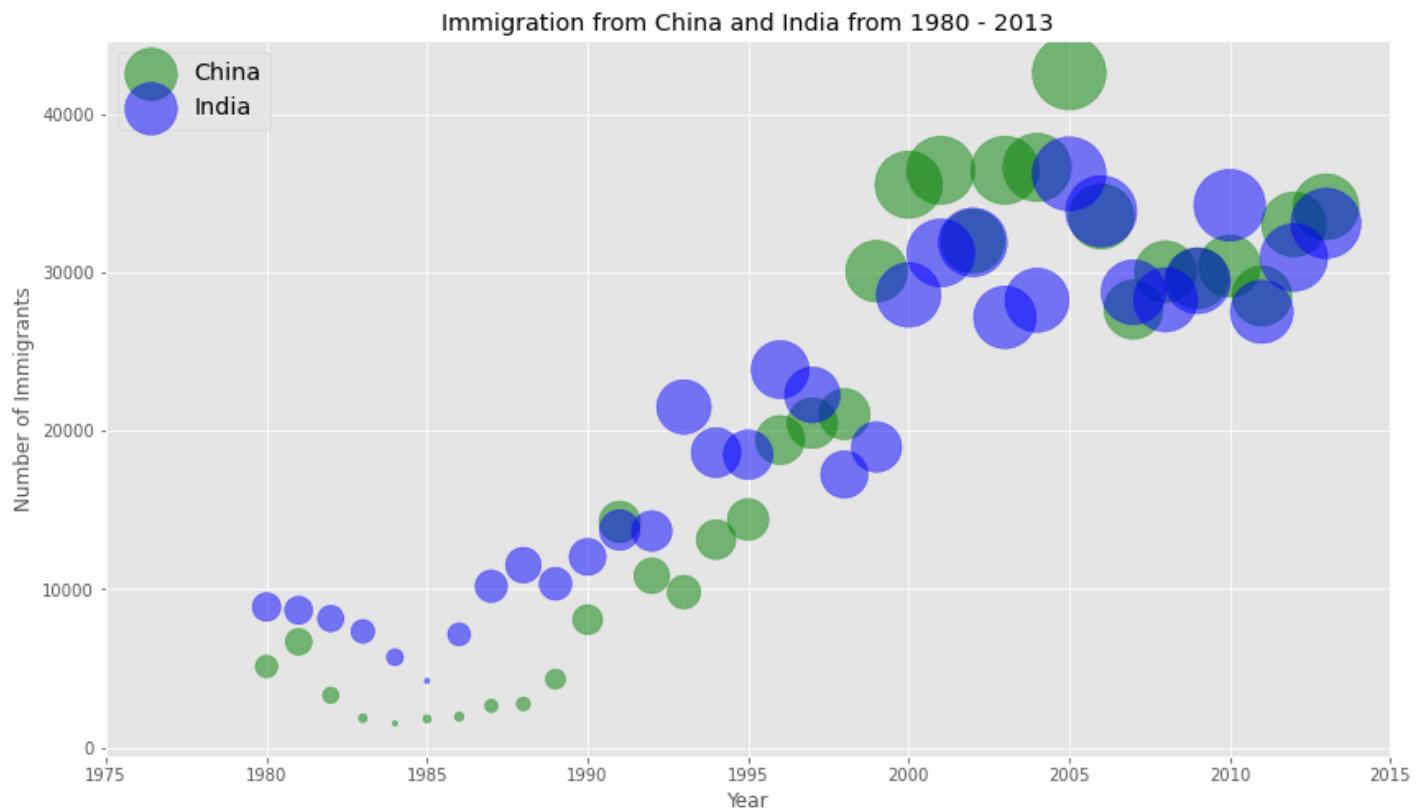
Step 2: Generate the bubble plots.

```
In [34]: # China
ax0 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='China',
                     figsize=(14, 8),
                     alpha=0.5,
                     color='green',
                     s=norm_china * 2000 + 10, # pass in weights
                     xlim=(1975, 2015)
                    )

# India
ax1 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='India',
                     alpha=0.5,
                     color="blue",
                     s=norm_india * 2000 + 10,
                     ax = ax0
                    )

ax0.set_ylabel('Number of Immigrants')
ax0.set_title('Immigration from China and India from 1980 - 2013')
ax0.legend(['China', 'India'], loc='upper left', fontsize='x-large')
```

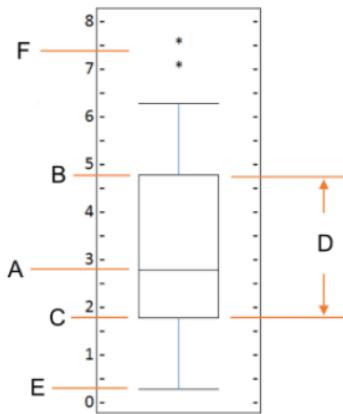
Out[34]: <matplotlib.legend.Legend at 0x7f6aa1f405f8>



Click here for a sample python solution ````python #The correct answer is: # China ax0 = df_can_t.plot(kind='scatter', x='Year', y='China', figsize=(14, 8), alpha=0.5, # transparency color='green', s=norm_china * 2000 + 10, # pass in weights xlim=(1975, 2015)) # India ax1 = df_can_t.plot(kind='scatter', x='Year', y='India', alpha=0.5, color="blue", s=norm_india * 2000 + 10, ax = ax0) ax0.set_ylabel('Number of Immigrants') ax0.set_title('Immigration from China and India from 1980 - 2013') ax0.legend(['China', 'India'], loc='upper left', fontsize='x-large') ````

Review Question 1

1 point possible (graded)



What do the letters in the box plot above represent?

- A = Mean, B = Upper Mean Quartile, C = Lower Mean Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers
- A = Mean, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers
- A = Median, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Outliers
- A = Median, B = Third Quartile, C = Mean, D = Inter Quartile Range, E = Lower Quartile, and F = Outliers
- A = Mean, B = Third Quartile, C = First Quartile, D = Inter Quartile Range, E = Minimum, and F = Maximum



Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

S:

✓ Correct (1/1 point)

Review Question 2

1/1 point (graded)

What is the correct combination of function and parameter to create a box plot in Matplotlib?

- Function = box, and Parameter = type, with value = "plot"
- Function = boxplot, and Parameter = type, with value = "plot"
- Function = plot, and Parameter = type, with value = "box"
- Function = plot, and Parameter = kind, with value = "boxplot"
- Function = plot, and Parameter = kind, with value = "box"



Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

✓ Correct (1/1 point)

Module Introduction

In this module, you will learn about advanced visualization tools such as waffle charts and word clouds and how to create them. You will also learn about seaborn, which is another visualization library, and how to use it to generate attractive regression plots. In addition, you will learn about Folium, which is another visualization library, designed especially for visualizing geospatial data. Finally, you will learn how to use Folium to create maps of different regions of the world and how to superimpose markers on top of a map, and how to create choropleth maps.

Learning Objectives

In this lesson you will learn about:

- Apply advanced visualization tools to create waffle charts and word clouds.
- Use Seaborn with Matplotlib to generate attractive regression plots.
- Explain how to use the Folium, for visualizing geospatial data.
- Use Folium to create maps and superpose markers.
- Create Choropleth Maps with Folium.

Waffle Charts

We will learn about what some consider an advanced visualization tool, namely the waffle chart. So what is a waffle chart? A waffle chart is a great way to visualize data in relation to a whole or to highlight progress against a given threshold.

For example, say immigration from Scandinavia to Canada is comprised only of immigration from Denmark, Norway, and Sweden, and we're interested in visualizing the contribution of each of these countries to the Scandinavian immigration to Canada.

The main idea here is for a given waffle chart whose desired height and width are defined, the contribution of each country is transformed into a number of tiles that is proportional to the country's contribution to the total, so that more the contribution the more the tiles, resulting in what resembles a waffle when combined. Hence the name waffle chart.

Unfortunately, Matplotlib does not have a built-in function to create waffle charts. Therefore, in the lab session, I'll walk you through the process of creating your own Python function to create a waffle chart.

Data Visualization with Python

Waffle Charts

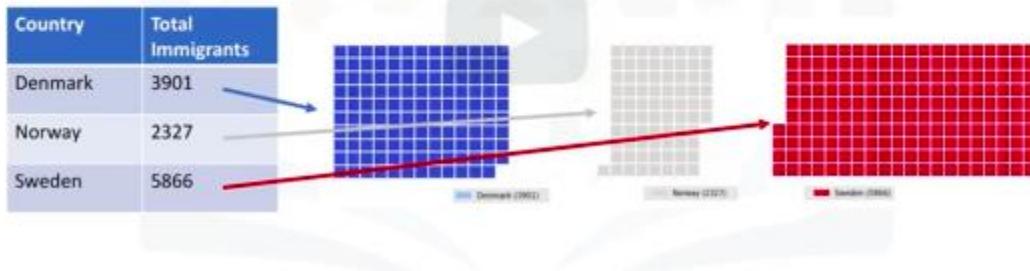
Waffle Charts

- A waffle chart is an interesting visualization that is normally created to display progress toward goals.

Country	Total Immigrants
Denmark	3901
Norway	2327
Sweden	5866

Waffle Charts

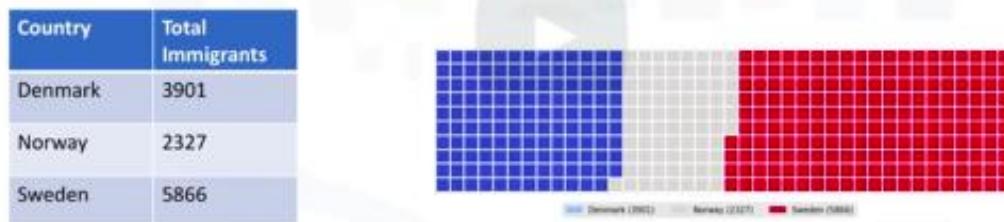
- A waffle chart is an interesting visualization that is normally created to display progress toward goals.



The main idea here is for a given waffle chart whose desired height and width are defined, the contribution of each country is transformed into a number of tiles that is proportional to the country's contribution to the total, so that more the contribution the more the tiles, resulting in what resembles a waffle when combined. Hence the name waffle chart.

Waffle Charts

- A waffle chart is an interesting visualization that is normally created to display progress toward goals.



The contribution of each country is transformed into a number of tiles that is proportional to the country's contribution to the total, so that more the contribution the more the tiles, resulting in what resembles a waffle when combined.

Word Cloud

We will learn about another advanced visualization tool: the **word cloud**. So what is a word cloud?

- A word cloud is simply a depiction of the importance of different words in the body of text.
- A word cloud works in a simple way; the more a specific word appears in a source of textual data the bigger and bolder it appears in the world cloud. So given some text data on recruitment, for example, we generate a cloud of words like this. This cloud is telling us that words such as recruitment, talent, candidates, and so on, are the words that really stand out in these text documents.
- And assuming that we didn't know anything about the content of these documents, a word cloud can be very useful to assign a topic to some unknown textual data.

Unfortunately, just like waffle charts, Matplotlib does not have a built-in function to generate word clouds. However, luckily a Python library for cloud word generation that was created by Andreas Mueller is publicly available. So, in the lab session we will learn how to use Mueller's word cloud generator, and we will also create interesting word clouds superimposed on different background images. So make sure to complete this module's lab session. And with this, we conclude our video on word clouds

Data Visualization with Python

Word Clouds

Word Clouds

- A Word cloud is a depiction of the frequency of different words in some textual data.

Bandsky 2
Faculty requirement for

After leaving Ravis, I must attend college because it is definitely a requirement for becoming a veterinarian. In fact, a bachelor's degree is necessary in order to even enter a veterinarian program. One must also possess excellent communication, leadership, public speaking, and organizational skills. I have put a lot of thought and consideration into college, and I have decided that I would like to go to the University of Illinois. It is a wonderful school, and they even have a graduate program designed for students who want to become veterinarians. Once I have completed a veterinarian program, I will be able to pursue my dream career. This career provides numerous benefits, the first of which is salary. The average veterinarian salary is \$60,000 a year, a salary that would definitely allow me to live a comfortable life. Secondly, it is a rewarding job. This job would provide me with the satisfaction of knowing that I am helping or saving an animal's life. Finally, becoming a veterinarian would assure me a lifetime of happiness. I know I would love going to job every day, because I would be working with what I love most: animals.

In summary, when I graduate from Ravis, I plan to go to college to become a veterinarian. I love animals and I want to do anything that I can to help them. I know I am only a freshman, but I also know that I am growing up quickly. As Frits Boelke quotes, "Life moves pretty fast. If you don't stop and look around once in a while, you could miss it!"

pretty fast. If you don't stop.



A word cloud works in a simple way; the more a specific word appears in a source of textual data the bigger and bolder it appears in the world cloud.

Seaborn and Regression Plots

We will learn about a new visualization library in Python, which is Seaborn. Although Seaborn is another data visualization library, it is actually based on Matplotlib.

- It was built primarily to provide a high-level interface for drawing attractive statistical graphics, such as regression plots, box plots, and so on.
- Seaborn makes creating plots very efficient. Therefore with
- Seaborn you can generate plots with code that is 5 times less than with Matplotlib.

Let's see how we can use Seaborn to create a statistical graphic. Let's look into regression plots. Let's say we have a dataframe called df_total of total immigration to Canada from 1980 to 2013 with the year in one column and the corresponding total immigration in another column, and say we're interested in creating a scatter plot along with a regression line to highlight any trends in the data.

With Seaborn, you can do all this with literally one line of code. The way to do this, we first import Seaborn and let's import it as sns. Then, we call the Seaborn regplot function. We basically tell it to use the dataframe df_total and to plot the column year on the horizontal axis and the column total on the vertical axis. And the output of this one line of code is a scatter plot with a regression line and not just that, but also 95% confidence interval.

Isn't that really amazing? Seaborn's regplot function also accepts additional parameters for any personal customization. So you can change the color for example using the color parameter. Let's go ahead and change the color to green. Also, you can change the marker shape as well using the marker parameter. Let's go ahead and change the shape of our markers to a + marker instead of the default circular marker. In the lab session, we explore regression plots with Seaborn in more details, so make sure to complete this module's lab session. And with this we conclude our short introduction to Seaborn and regression plots.

Data Visualization with Python

Seaborn and Regression Plots

Seaborn

- Seaborn is a Python visualization library based on Matplotlib.
- Visuals that need ~20 lines of code using Matplotlib to be created, with seaborn, the number of lines of code is reduced by 5-fold.
- Let's look at creating scatter plots with regression lines for example.

It was built primarily to provide a high-level interface for drawing attractive statistical graphics, such as regression plots, box plots, and so on. Seaborn makes creating plots very efficient. Therefore, with Seaborn you can generate plots with code that is 5 times less than with Matplotlib. Let's see how we can use Seaborn to create a statistical graphic

Regression Plots

df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.
.	.

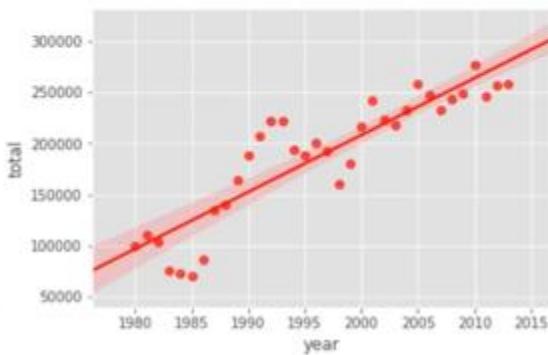
df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.
.	.

```
import seaborn as sns  
ax = sns.regplot(x='year', y='total', data=df_tot)
```

Regression Plots

df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.
.	.

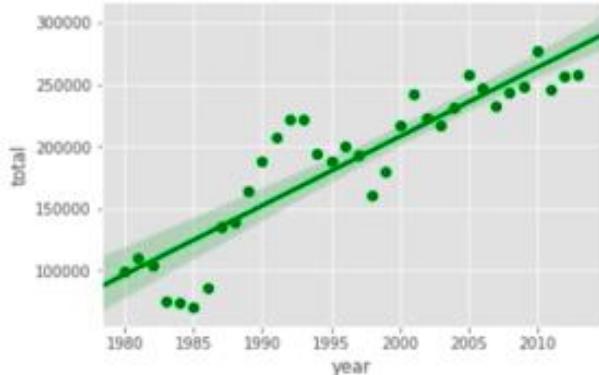
```
import seaborn as sns  
ax = sns.regplot(x='year', y='total', data=df_tot)
```



Regression Plots

df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.

```
import seaborn as sns  
ax = sns.regplot(x='year', y='total', data=df_tot,  
color='green')
```



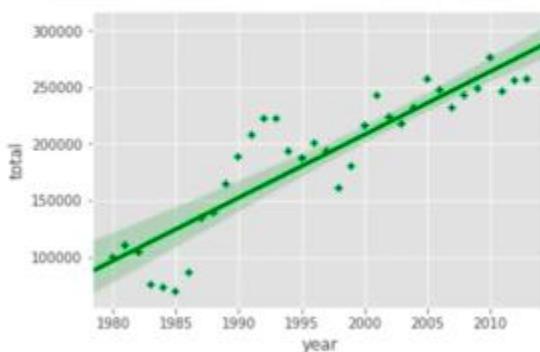
COGNITIVE CLASS.ai

With Seaborn, you can do all this with literally one line of code. The way to do this, we first import Seaborn and let's import it as sns. Then, we call the Seaborn regplot function. We basically tell it to use the dataframe df_total and to plot the column year on the horizontal axis and the column total on the vertical axis. And the output of this one line of code is a scatter plot with a regression line and not just that, but also 95% confidence interval.

Regression Plots

df_total	
year	total
1980	99137
1981	110563
1982	104271
1983	75550
1984	73417
.	.

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



COGNITIVE CLASS.ai

you can change the color for example using the color parameter. Let's go ahead and change the color to green. Also, you can change the marker shape as well using the marker parameter. Let's go ahead and change the shape of our markers to a + marker instead of the default circular marker.

8.3.Advanced Visualizations and Geospatial Data

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

In this module, you will learn about advanced visualization tools such as waffle charts and word clouds and how to create them. You will also learn about seaborn, which is another visualization library, and how to use it to generate attractive regression plots. In addition, you will learn about Folium, which is another visualization library, designed especially for visualizing geospatial data. Finally, you will learn how to use Folium to create maps of different regions of the world and how to superimpose markers on top of a map, and how to create choropleth maps.

Learning Objectives

- Apply advanced visualization tools to create waffle charts and word clouds.
 - Use Seaborn with Matplotlib to generate attractive regression plots.
 - Explain how to use the Folium, for visualizing geospatial data.
 - Use Folium to create maps and superpose markers.
 - Create Choropleth Maps with Folium.
-

- [Advanced Visualization Tools & Visualizing Geospatial Data](#)
- [Lab 3-1: Advanced Visualization Tools](#)
- [Lab 3-2: Creating Maps and Visualizing Geospatial Data](#)

8.3.1.Advanced Visualization Tools & Visualizing Geospatial Data

Advanced Visualization Tools

Seaborn and Regression Plots

Seaborn

- Seaborn is a Python visualization library based on Matplotlib.
- Visuals that need ~20 lines of code using Matplotlib to be created, with seaborn, the number of lines of code is reduced by 5-fold.

Regression Plots

- Example:

```
import seaborn as sns  
ax = sns.regplot(x='year', y='total', data=df_total, color='green', marker='+')
```

Advanced Visualization Tools

LATEST SUBMISSION GRADE 100%

Question 1

Seaborn is a Python visualization library that provides a high-level interface for drawing attractive statistical graphics, such as regression plots and box plots.

- True
- False

Correct.

Question 2

The following code

```
import seaborn as sns
```

```
ax = sns.regplot(x="year", y="total", data=data_df, color="green", marker="+")
```

creates the following regression plot.

- **True**
- **False**

Correct.

Question 3

In Python, creating a waffle chart is straightforward since we can easily create one using the scripting layer of Matplotlib.

- **True**
- **False**

Correct.

Visualizing Geospatial Data

Introduction to Folium

What is Folium?

- Folium is a powerful Python library that helps you create several types of Leaflet maps.
- It enables both the binding of data to a map for choropleth visualizations as well as passing visualizations as markers on the map.
- The library has a number of built-in tilesets from OpenStreetMap, Mapbox, and Stamen, and supports custom tilesets with Mapbox API keys.

Creating a World Map

```
# define the world map
world_map = folium.Map()

# display world map
world_map
```

Creating a Map of Canada

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

# dispaly world map
world_map
```

Map Styles - Stamen Toner

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

# dispaly map
world_map
```

Map Styles - Stamen Terrain

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

# dispaly map
world_map
```

Maps with Markers

Label the Marker

```
# generate map of Canada
canada_map = folium.Map(
    location=[56.130, -106.35],
    zoom_start=4
)

## add a red marker to Ontario
# create a feature group
ontario = folium.map.FeatureGroup()

# style the feature group
ontario.add_child(
    folium.features.CircleMarker(
        [51.25, -85.32], radium=5,
        color='red', fill_color='Red'
    )
)

# add the feature group to the map
canada_map.add_child(ontario)

# label the marker
folium.Marker([51.25, -85.32],
    popup='Ontario').add_to(canada_map)

# display map
canada_map
```

Choropleth Maps

Choropleth Maps

Geojson File

```
{  
    "type": "FeatureCollection",  
    "features": [  
        {  
            "type": "Feature",  
            "properties": {  
                "name": "Brunei"  
            },  
            "geometry": {  
                "type": "Polygon",  
                "coordinates": [  
                    [  
                        [114.204017, 4.525874], [114.599961, 4.900011], [115.45071, 5.44773],  
                        [115.4057, 4.955228], [115.347461, 4.316636], [114.869557, 4.348314],  
                        [114.659596, 4.007637], [114.204017, 4.525874]  
                    ]  
                ]  
            },  
            "id": "BRN"  
        },  
    ],  
}
```

Creating the Map

```
# create a plain world map  
world_map = folium.Map(  
    zoom_start=2,  
    title="Mapbox Bright"  
)  
  
## geojson file  
world_geo = r'world_countries.json'  
  
# generate choropleth map using the total  
# population of each country to Canada from  
# 1980 to 2013  
world_map.choropleth(  
    geo_path=world_geo,  
    data=df_canada,  
    columns=['Country', 'Total'],  
    key_on='feature.properties.name',  
    fill_color='YlOrRd',  
    legend_name='Immigration to Canada'  
)  
  
# display map  
world_map
```

Visualizing Geospatial Data

LATEST SUBMISSION GRADE 100%

Question 1

You cluster markers, superimposed onto a map in Folium, using a *marker cluster* object.

- True
- False

Correct.

Question 2

The following code will generate a map of Spain, displaying its hill shading and natural vegetation.

```
folium.Map(location=[40.4637, -3.7492], zoom_start=6, tiles='Stamen Terrain')
```

- True
- False

Correct.

Question 3

A choropleth map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map.

- True
- False

Correct.

<https://www.coursera.org/professional-certificates/ibm-data-science>

© 2021 Coursera Inc. All rights reserved.

Waffle Charts, Word Clouds, and Regression Plots

Estimated time needed: **30** 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

1. [Exploring Datasets with *pandas](#0)
2. [Downloading and Prepping Data](#2)
3. [Visualizing Data using Matplotlib](#4)
4. [Waffle Charts](#6)
5. [Word Clouds](#8)
6. [Regression Plots](#10)

Exploring Datasets with *pandas* and Matplotlib

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.

Downloading and Prepping Data

Import Primary Modules:

In [2]:

```
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
```

Let's download and import our primary Canadian Immigration dataset using *pandas*'s `read_excel()` method. Normally, before we can do that, we would need to download a module which *pandas* requires reading in Excel files. This module was **openpyxl** (formerly **xlrd**). For your convenience, we have pre-installed this module, so you would not have to worry about that. Otherwise, you would need to run the following line of code to install the **openpyxl** module:

```
! pip3 install openpyxl
```

Download the dataset and read it into a *pandas* dataframe:

In [3]:

```
df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2)

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

Data downloaded and read into a dataframe!

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

In [4]:

```
df_can.head()
```

Out[4]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polyynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	0
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	0	1	1

5 rows × 43 columns

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

In [5]:

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

(195, 43)

Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to *Introduction to Matplotlib and Line Plots* and *Area Plots, Histograms, and Bar Plots* for a detailed description of this preprocessing.

```
In [6]: # clean up the dataset to remove unnecessary columns (eg. REG)
df_can.drop(['AREA','REG','DEV','Type','Coverage'], axis = 1, inplace = True)

# Let's rename the columns so that they make sense
df_can.rename (columns = {'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace = True)

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

# set the country name as index - useful for quickly looking up countries using .loc method
df_can.set_index('Country', inplace = True)

# add total column
df_can['Total'] = df_can.sum (axis = 1)

# years that we will be using in this Lesson - useful for plotting later on
years = list(map(str, range(1980, 2014)))
print ('data dimensions:', df_can.shape)

data dimensions: (195, 38)
```

Visualizing Data using Matplotlib

Import and setup matplotlib:

```
In [7]: %matplotlib inline

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

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

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

Matplotlib version: 3.3.4

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.

```
In [8]: # Let's create a new dataframe for these three countries
df_dsn = df_can.loc[['Denmark', 'Norway', 'Sweden'], :]

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

Out[8]:

	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Denmark	Europe	Northern Europe	Developed regions	272	293	299	106	93	73	93	...	62	101	97	108	81	92	93	94	81	3901
Norway	Europe	Northern Europe	Developed regions	116	77	106	51	31	54	56	...	57	53	73	66	75	46	49	53	59	2327
Sweden	Europe	Northern Europe	Developed regions	281	308	222	176	128	158	187	...	205	139	193	165	167	159	134	140	140	5866

3 rows × 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 determining the proportion of each category with respect to the total.

```
In [9]: # compute the proportion of each category with respect to the total
total_values = df_dsn['Total'].sum()
category_proportions = df_dsn['Total'] / total_values

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

Out[9]: Category Proportion

Country	Category Proportion
Denmark	0.322557
Norway	0.192409
Sweden	0.485034

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

```
In [10]: width = 40 # width of chart
height = 10 # height of chart

total_num_tiles = width * height # total number of tiles

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

Total number of tiles is 400.

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

```
In [11]: # compute the number of tiles for each category  
tiles_per_category = (category_proportions * total_num_tiles).round().astype(int)  
  
# print out number of tiles per category  
pd.DataFrame({"Number of tiles": tiles_per_category})
```

Out[11]: **Number of tiles**

Country	Number of tiles
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.

```
In [12]: # 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.

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

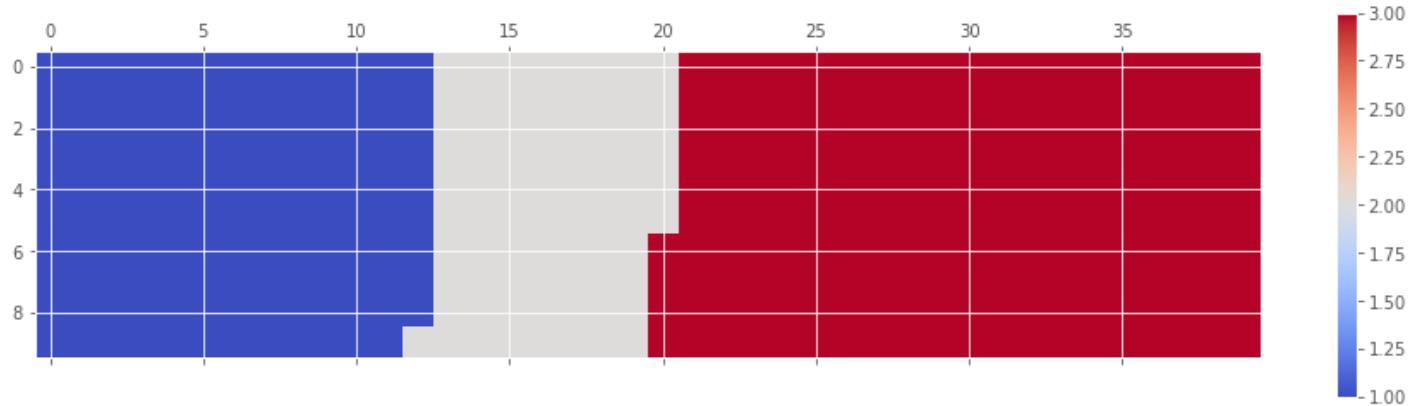
Step 5. Map the waffle chart matrix into a visual.

In [14]:

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

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

<Figure size 432x288 with 0 Axes>



Step 6. Prettify the chart.

In [15]:

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

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

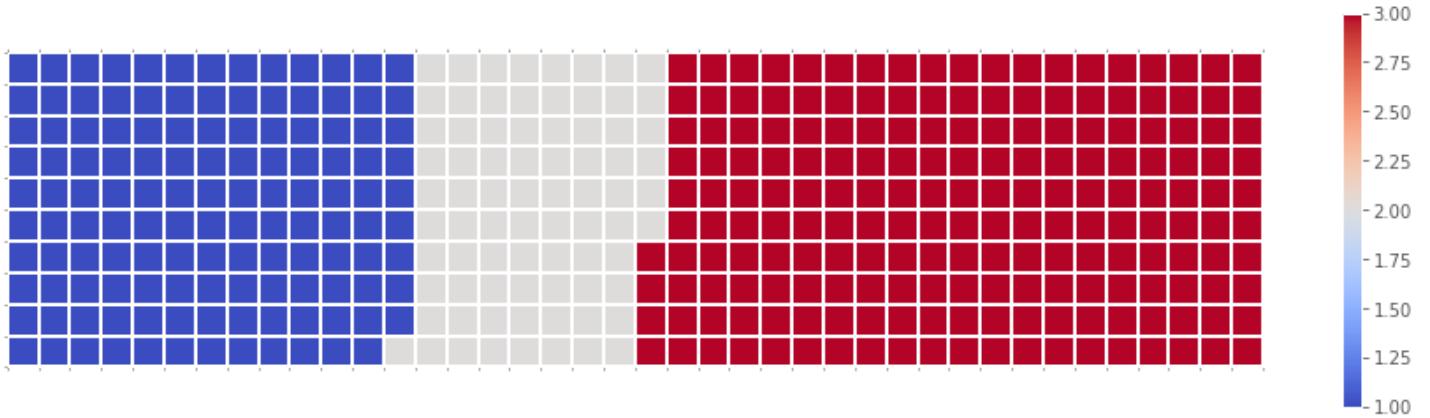
# get the axis
ax = plt.gca()

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

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

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

<Figure size 432x288 with 0 Axes>



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

```
In [16]: # 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)

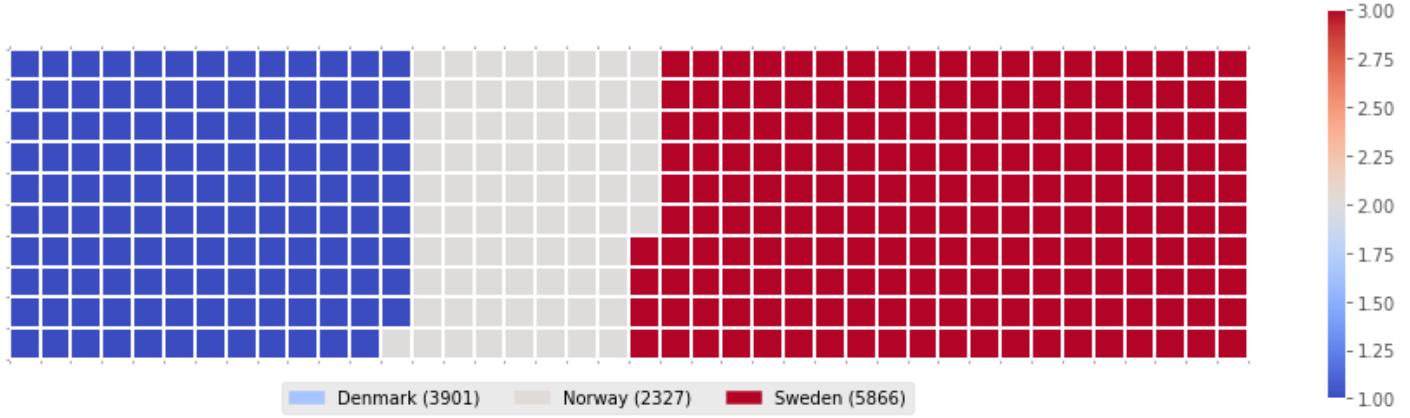
plt.xticks([])
plt.yticks([])

# 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 432x288 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.

```
In [17]: 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 category    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 gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-', linewidth=2)

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

```

```

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

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

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

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

```
In [18]: width = 40 # width of chart
height = 10 # height of chart

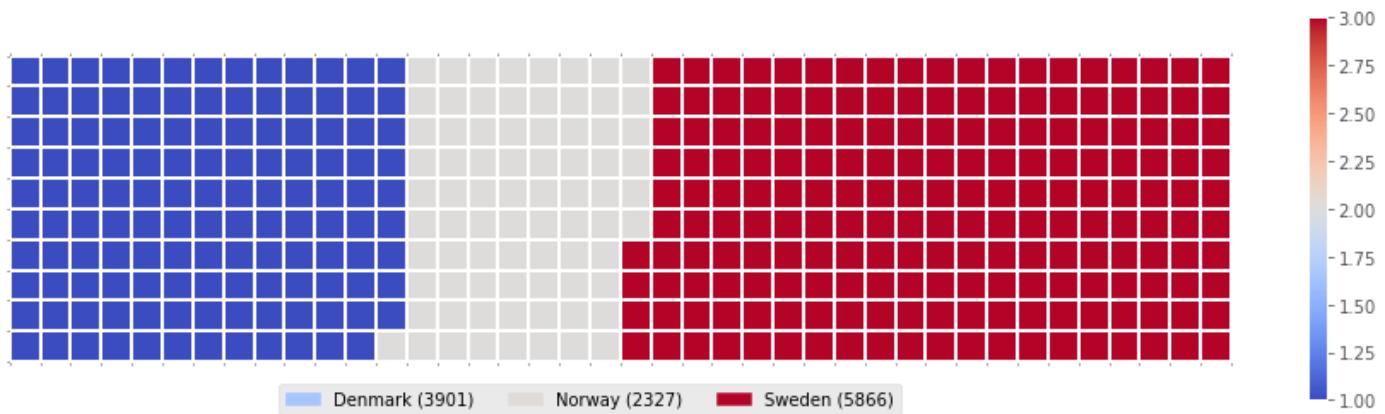
categories = df_dsn.index.values # categories
values = df_dsn['Total'] # correponding values of categories

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

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

```
In [19]: create_waffle_chart(categories, values, height, width, colormap)
```

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



There seems to be a new Python package for generating waffle charts called [PyWaffle](#), but it looks like the repository is still being built. But feel free to check it out and play with it.

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.

```
In [20]: # install wordcloud
! pip install wordcloud

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

print ('Wordcloud is installed and imported!')

Collecting wordcloud
  Downloading https://files.pythonhosted.org/packages/05/e7/52e4bef8e2e3499f6e96cc8ff7e0902a40b95014143b062acde4ff8b9fc8/wordcloud-1.8.1-cp36-cp36m-manylinux1_x86_64.whl (366kB) | 368kB 29.2MB/s eta 0:00:01
Requirement already satisfied: pillow in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from wordcloud) (8.3.1)
Requirement already satisfied: matplotlib in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from wordcloud) (3.3.4)
Requirement already satisfied: numpy>=1.6.1 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from wordcloud) (1.19.5)
Requirement already satisfied: pyparsing!=2.0.4,!>=2.1.2,!>=2.1.6,>=2.0.3 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from matplotlib>wordcloud) (2.4.7)
Requirement already satisfied: python-dateutil>=2.1 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from matplotlib>wordcloud) (2.8.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from matplotlib>wordcloud) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/cycler-0.10.0-py3.6.egg (from matplotlib>wordcloud) (0.10.0)
Requirement already satisfied: six>=1.5 in /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages (from python-dateutil>=2.1->matplotlib>wordcloud) (1.15.0)
Installing collected packages: wordcloud
Successfully installed wordcloud-1.8.1
Wordcloud is installed and imported!
```

Word clouds are commonly used to perform high-level analysis and visualization of text data. Accordingly, 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.

```
In [21]: 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/IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork/')


```

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

```
In [22]: 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.

```
In [23]: # instantiate a word cloud object
alice_wc = WordCloud(
    background_color='white',
    max_words=2000,
    stopwords=stopwords
)

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

```
Out[23]: <wordcloud.wordcloud.WordCloud at 0x7fbab54a8e10>
```

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

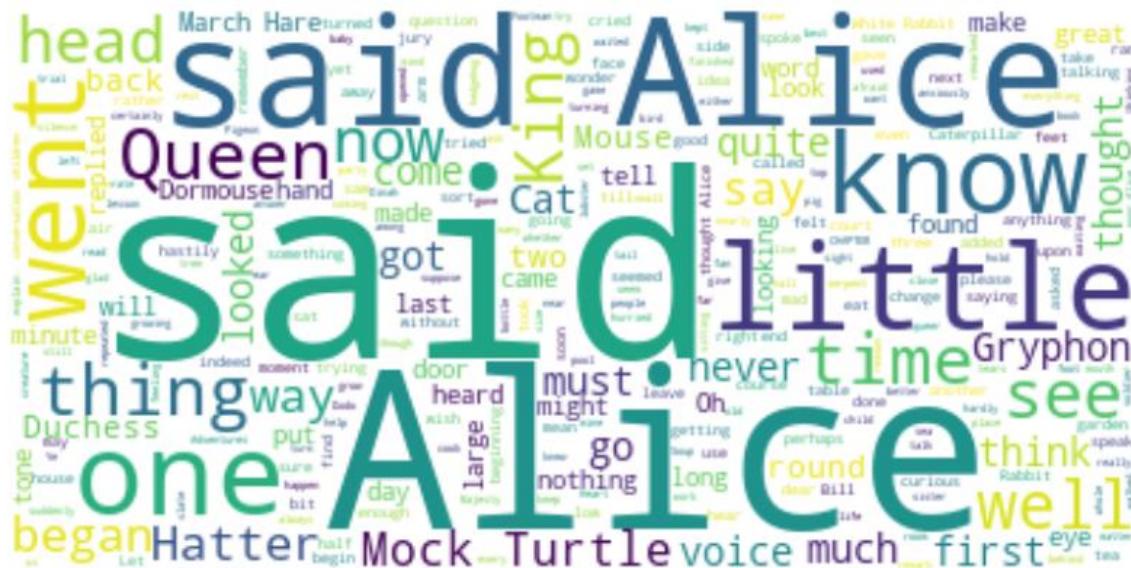
```
In [24]: # display the word cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```



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.

```
In [25]: 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.

```
In [26]: 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 `wordcloud` 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`.

```
In [27]: # 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-D
```

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

```
In [28]: 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.

```
In [29]: # 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()
```



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.

In [30]: `df_can.head()`

Out[30]:	Continent	Region	DevName	1980	1981	1982	1983	1984	1985	1986	...	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Country																					
Afghanistan	Asia	Southern Asia	Developing regions	16	39	39	47	71	340	496	...	3436	3009	2652	2111	1746	1758	2203	2635	2004	58639
Albania	Europe	Southern Europe	Developed regions	1	0	0	0	0	0	1	...	1223	856	702	560	716	561	539	620	603	15699
Algeria	Africa	Northern Africa	Developing regions	80	67	71	69	63	44	69	...	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
American Samoa	Oceania	Polynesia	Developing regions	0	1	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	6
Andorra	Europe	Southern Europe	Developed regions	0	0	0	0	0	0	0	2	...	0	1	1	0	0	0	0	1	15

5 rows × 38 columns

And what was the total immigration from 1980 to 2013?

In [31]: `total_immigration = df_can['Total'].sum()
total_immigration`

Out[31]: 6409153

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

In [32]:

```
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
```

Out[32]: '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 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.

```
In [33]: # create the word cloud  
wordcloud = WordCloud(background_color='white').generate(word_string)  
  
print('Word cloud created!')
```

Word cloud created!

```
In [34]: # 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!

Regression Plots

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

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

Let's first install *seaborn*

```
In [35]: # install seaborn  
! pip3 install seaborn  
  
# import library  
import seaborn as sns  
  
print('Seaborn installed and imported!')
```

```

Collecting seaborn
  Downloading seaborn-0.11.2-py3-none-any.whl (292 kB)
    |██████████| 292 kB 28.7 MB/s eta 0:00:01
Collecting pandas>=0.23
  Downloading pandas-1.3.2-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (11.5 MB)
    |██████████| 11.5 MB 63.3 MB/s eta 0:00:01
Collecting numpy>=1.15
  Downloading numpy-1.21.2-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
    |██████████| 15.8 MB 57.4 MB/s eta 0:00:01
Collecting matplotlib>=2.2
  Downloading matplotlib-3.4.3-cp38-cp38-manylinux1_x86_64.whl (10.3 MB)
    |██████████| 10.3 MB 39.6 MB/s eta 0:00:01
Collecting scipy>=1.0
  Downloading scipy-1.7.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (28.4 MB)
    |██████████| 28.4 MB 60.0 MB/s eta 0:00:01
Collecting pytz>=2017.3
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Requirement already satisfied: python-dateutil>=2.7.3 in /home/jupyterlab/conda/lib/python3.8/site-packages (from pandas>=0.23->seaborn) (2.8.1)
Collecting cycler>=0.10
  Downloading cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Requirement already satisfied: pyparsing>=2.2.1 in /home/jupyterlab/conda/lib/python3.8/site-packages (from matplotlib>=2.2->seaborn) (2.4.7)
Collecting kiwisolver>=1.0.1
  Downloading kiwisolver-1.3.2-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (1.2 MB)
    |██████████| 1.2 MB 74.8 MB/s eta 0:00:01
Collecting pillow>=6.2.0
  Downloading Pillow-8.3.2-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.0 MB)
    |██████████| 3.0 MB 60.9 MB/s eta 0:00:01
Requirement already satisfied: six>=1.5 in /home/jupyterlab/conda/lib/python3.8/site-packages (from python-dateutil>=2.7.3->pandas>=0.23->seaborn) (1.1.5)
Installing collected packages: pytz, numpy, pandas, cycler, kiwisolver, pillow, matplotlib, scipy, seaborn
Successfully installed cycler-0.10.0 kiwisolver-1.3.2 matplotlib-3.4.3 numpy-1.21.2 pandas-1.3.2 pillow-8.3.2 pytz-2021.1 scipy-1.7.1 seaborn-0.11.2
Seaborn installed and imported!

```

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

```

In [36]: # 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']

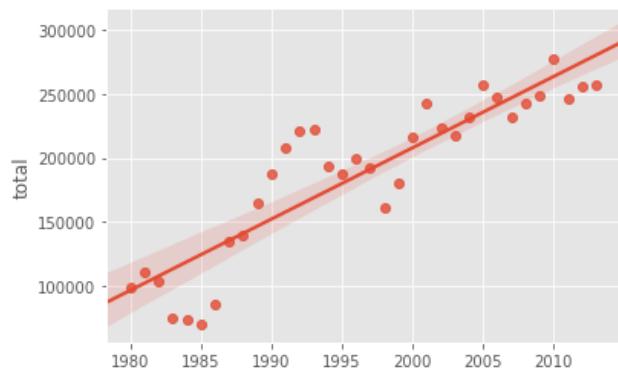
# view the final dataframe
df_tot.head()

```

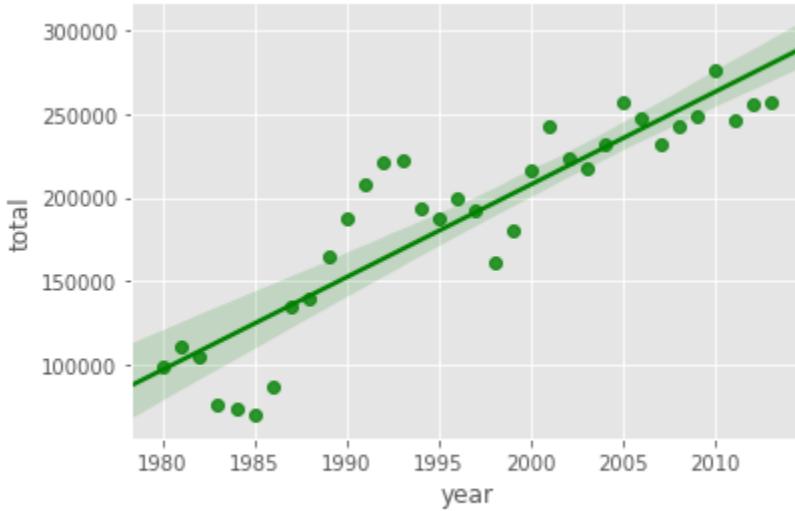
	year	total
0	1980.0	99137
1	1981.0	110563
2	1982.0	104271
3	1983.0	75550
4	1984.0	73417

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

```
In [38]:  
sns.regplot(x='year', y='total', data=df_tot)  
plt.show()
```

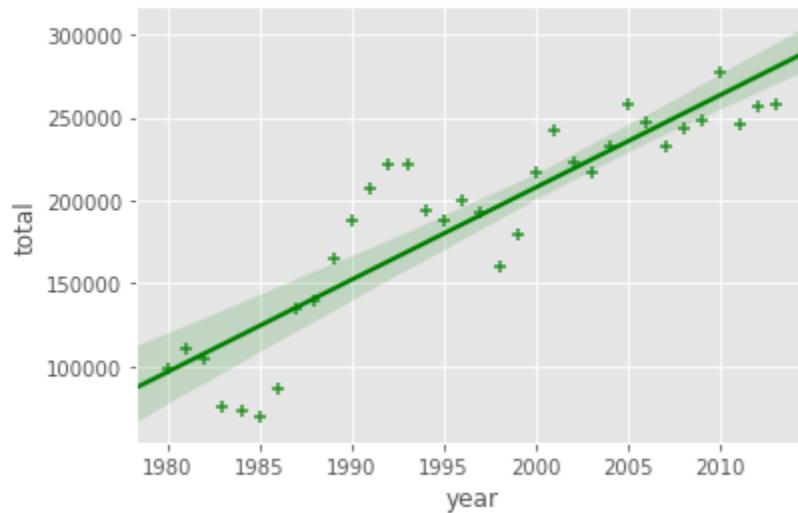


This is not magic; it is *seaborn*! You can also customize the color of the scatter plot and regression line. Let's change the color to green.



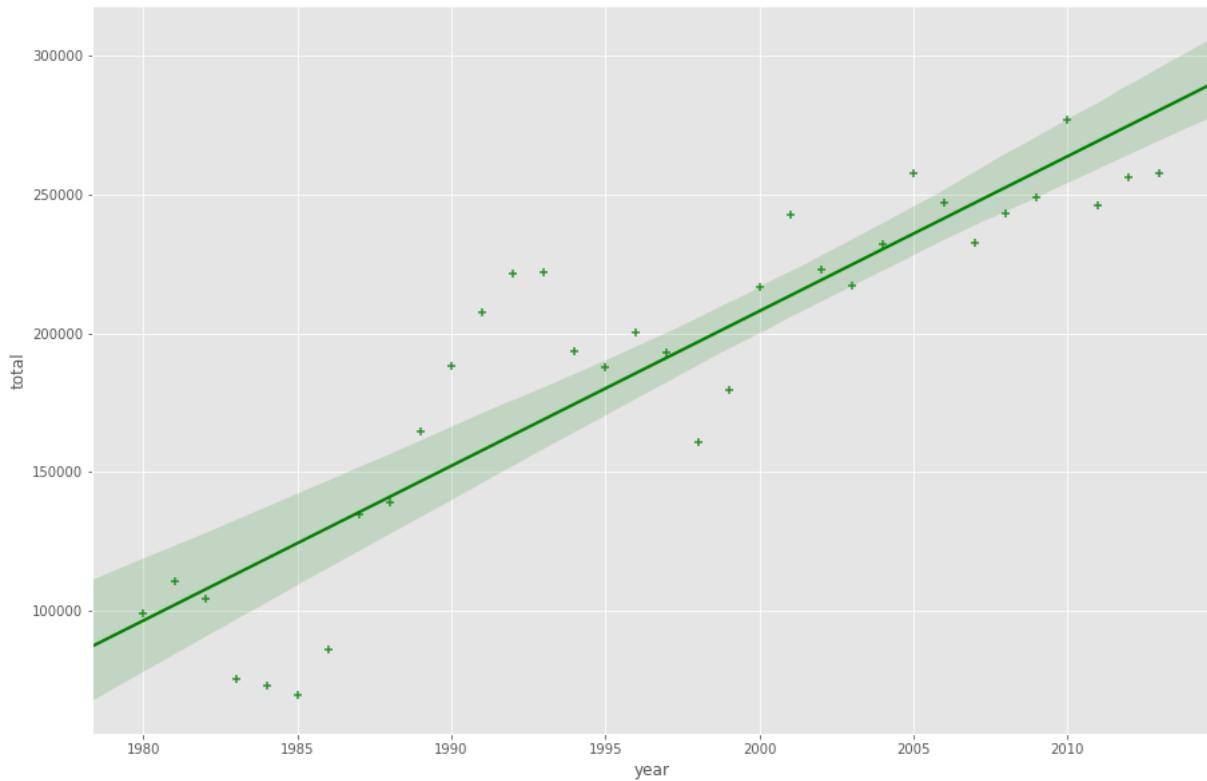
You can always customize the marker shape, so instead of circular markers, let's use `+`.

```
In [40]: 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.

```
In [42]: plt.figure(figsize=(15, 10))
sns.regplot(x='year', y='total', data=df_tot, color='green', marker='+')
plt.show()
```

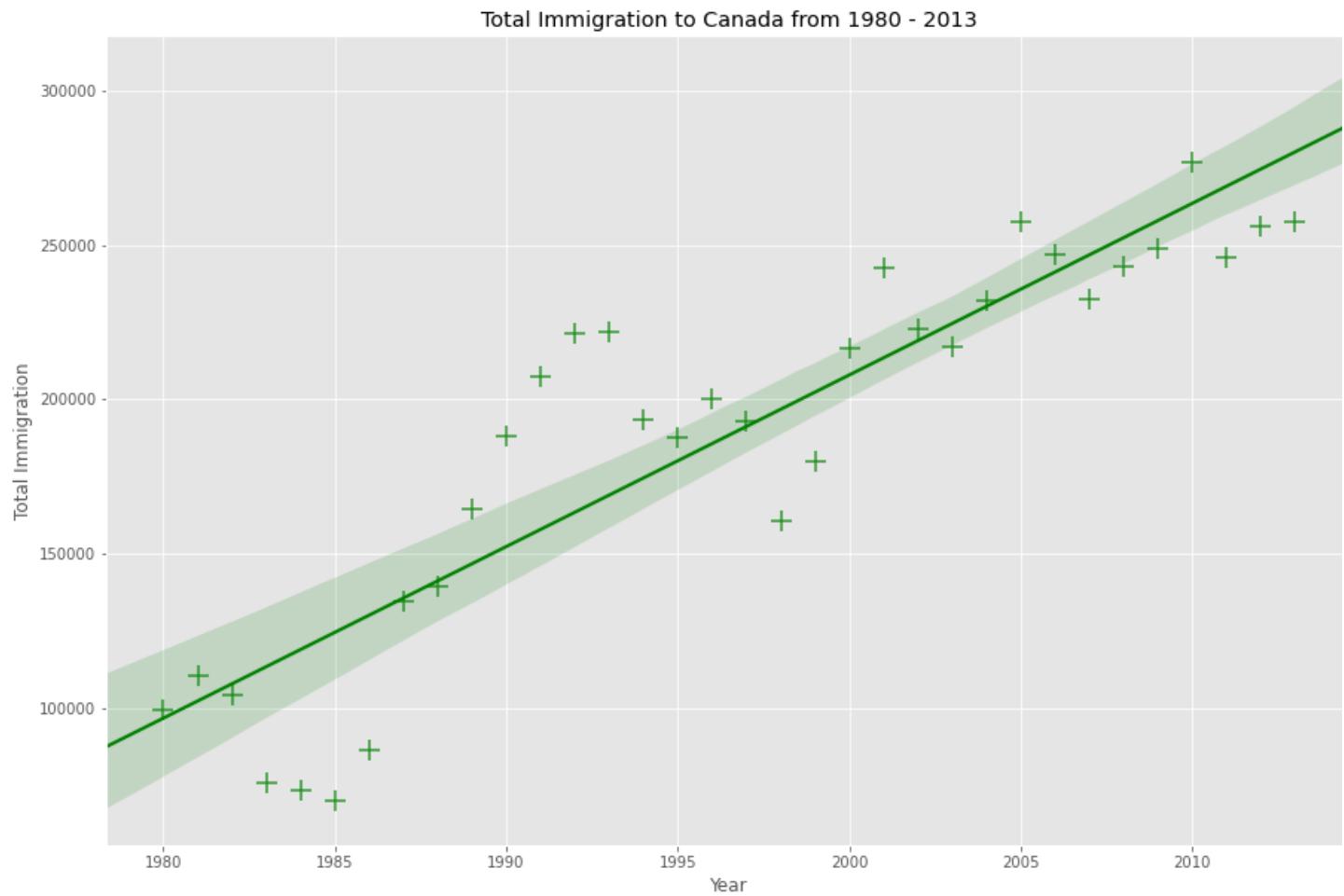


And let's increase the size of markers so they match the new size of the figure, and add a title and x- and y-labels.

In [43]:

```
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()
```

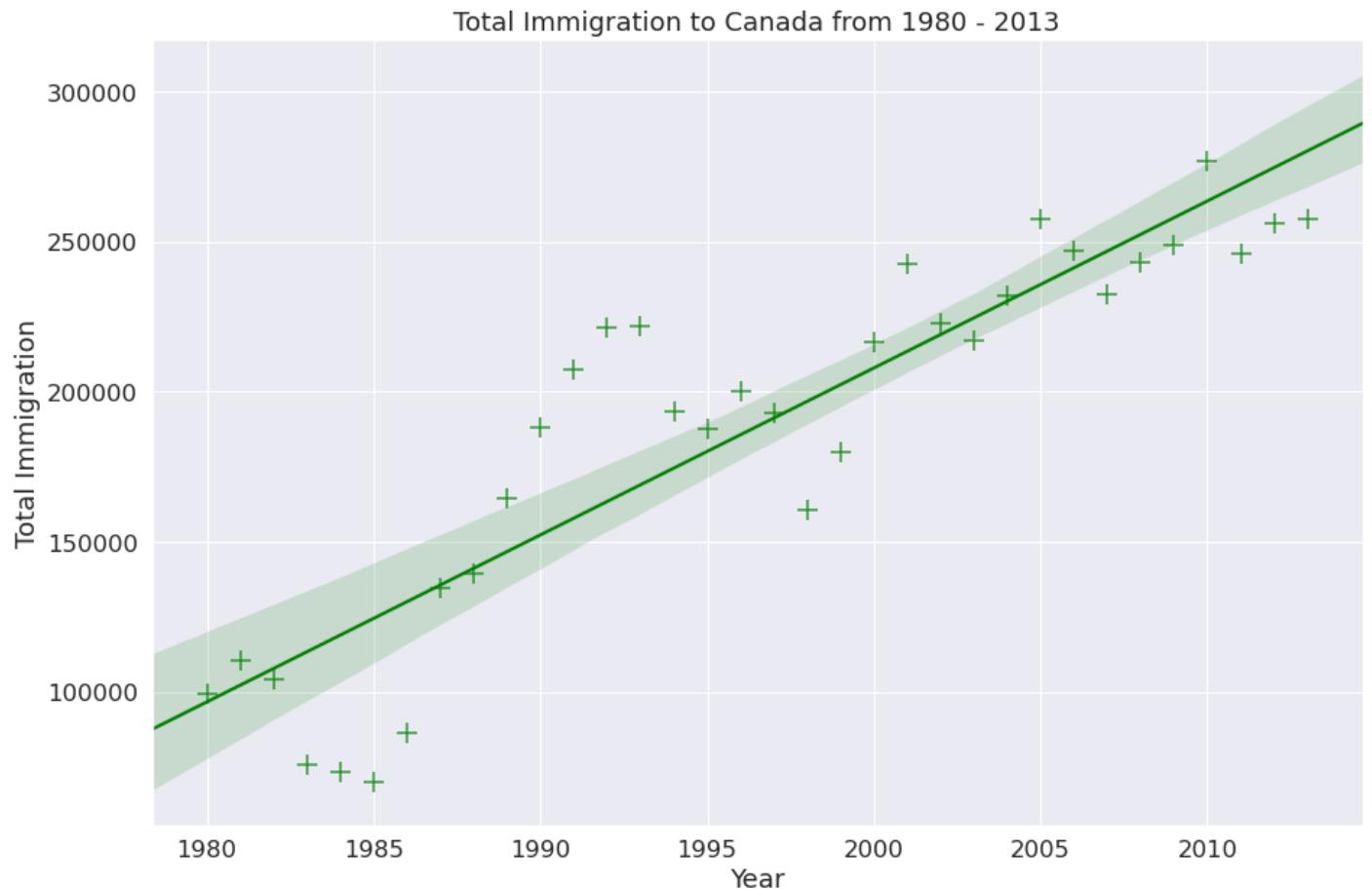


And finally increase the font size of the tickmark labels, the title, and the x- and y-labels so they don't feel left out!

```
In [44]: 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()
```



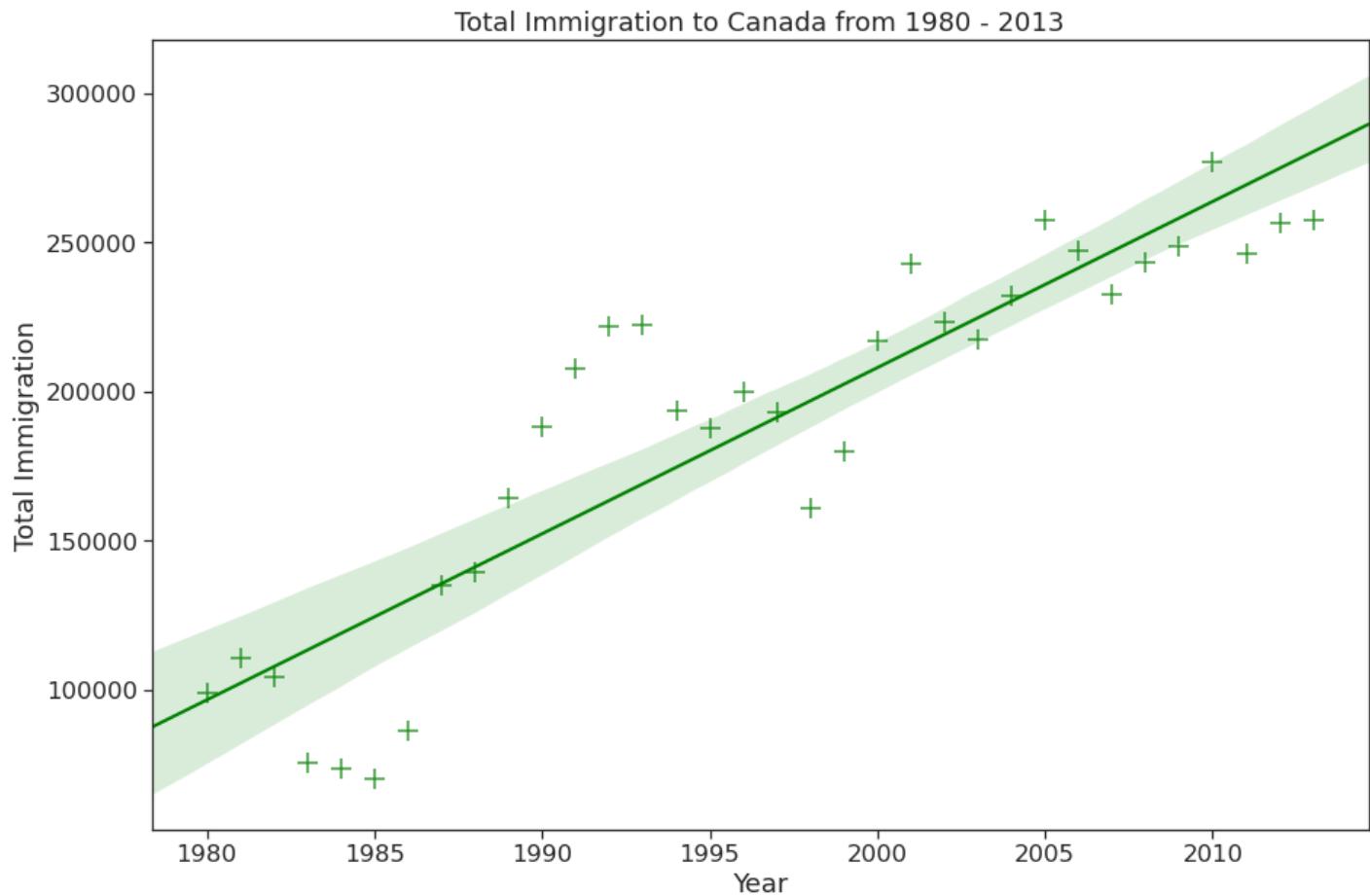
Amazing! A complete scatter plot with a regression fit with 5 lines of code only. Isn't this really amazing?

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

```
In [45]: 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()
```

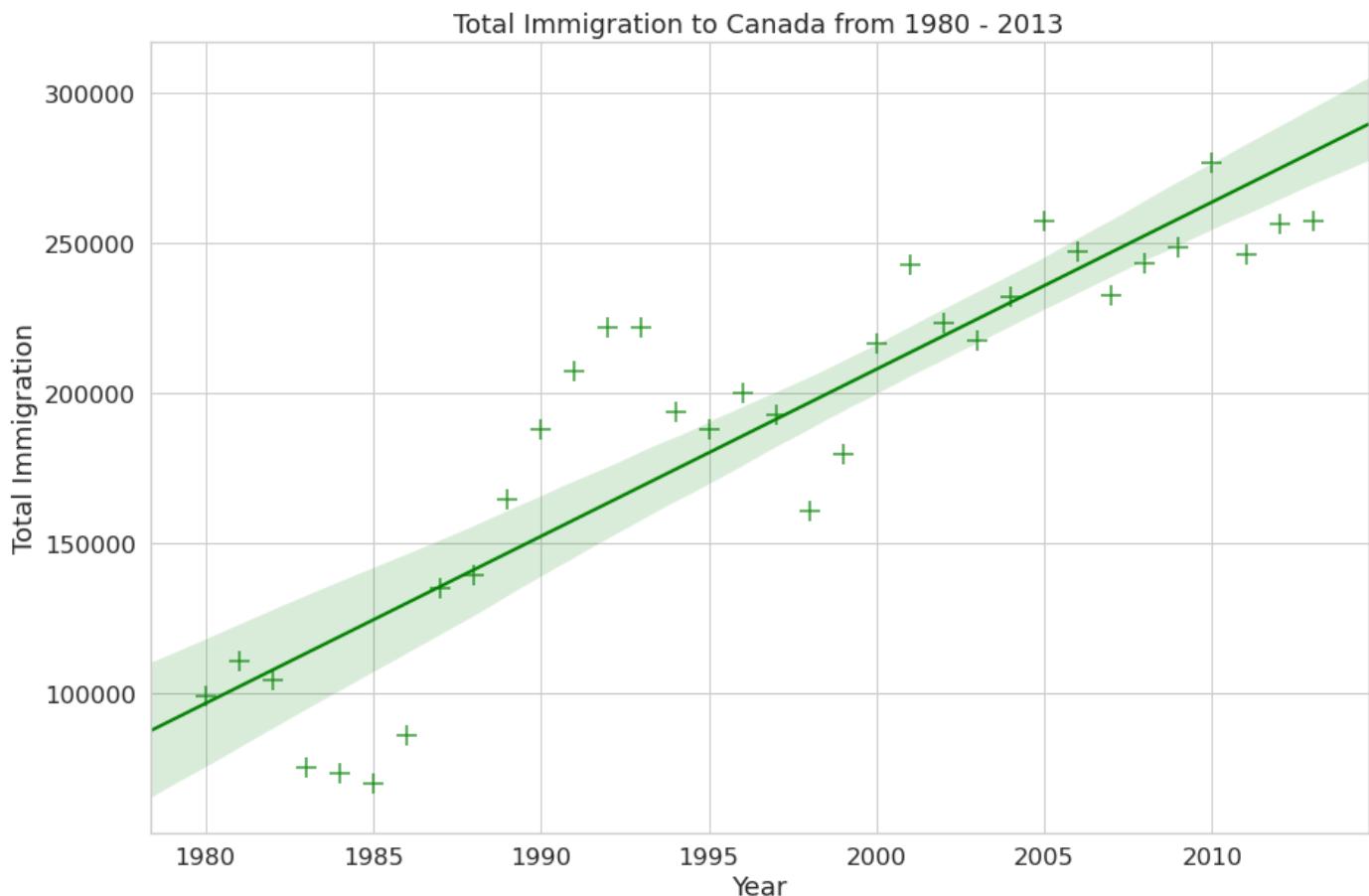


Or to a white background with gridlines.

```
In [46]: 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.

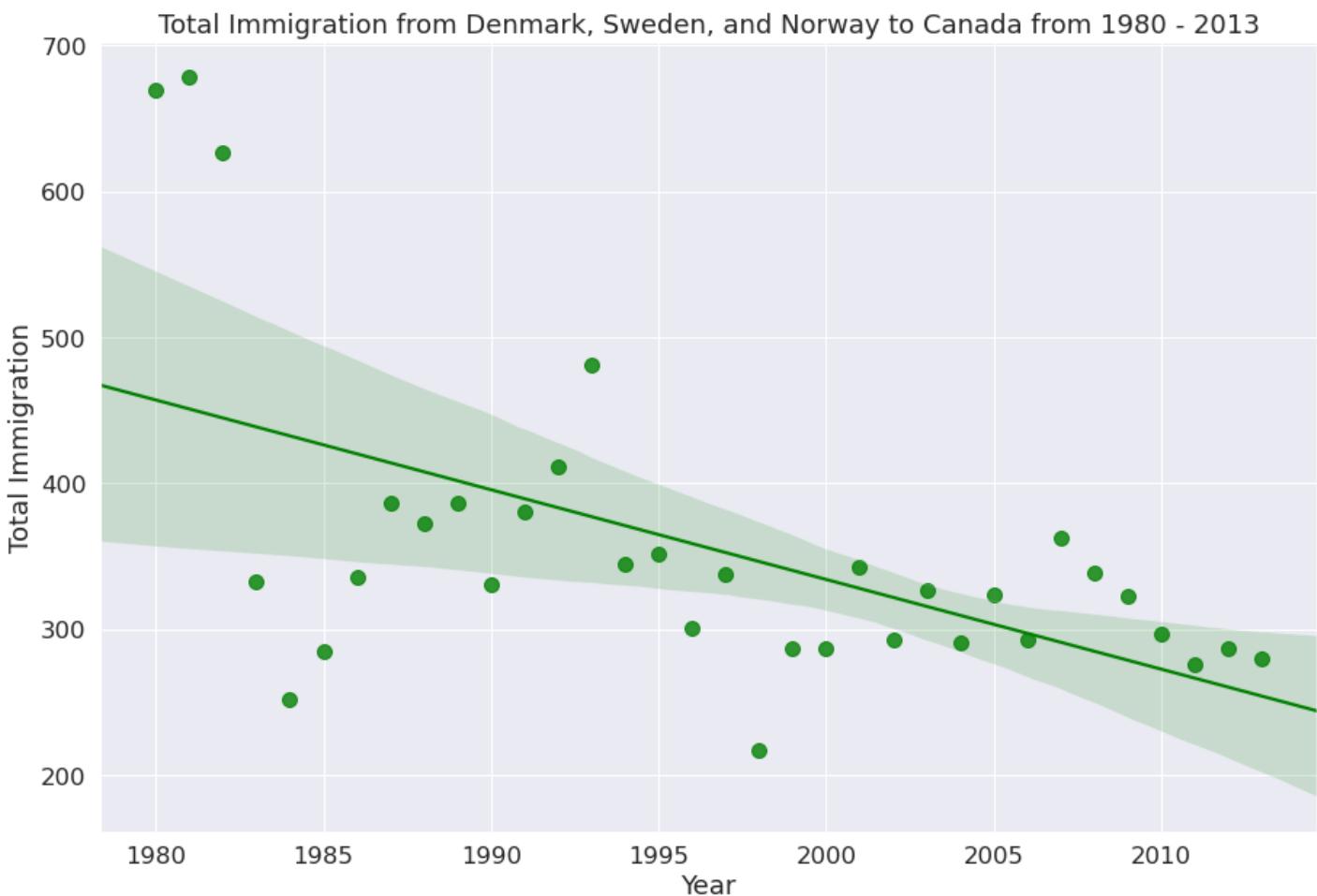
```
In [47]: df_countries = df_can.loc[['Denmark', 'Sweden', 'Norway'], years].transpose()
df_total = pd.DataFrame(df_countries.sum(axis=1))

df_total.reset_index(inplace=True)
df_total.columns = ['year', 'total']
df_total['year'] = df_total['year'].astype(int)

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

sns.set(font_scale=1.5)

ax = sns.regplot(x='year', y='total', data=df_total, color='green', scatter_kws={'s':100})
ax.set_xlabel('Year', ylabel='Total Immigration')
ax.set_title('Total Immigration from Denmark, Sweden, and Norway to Canada from 1980 - 2013')
plt.show()
```



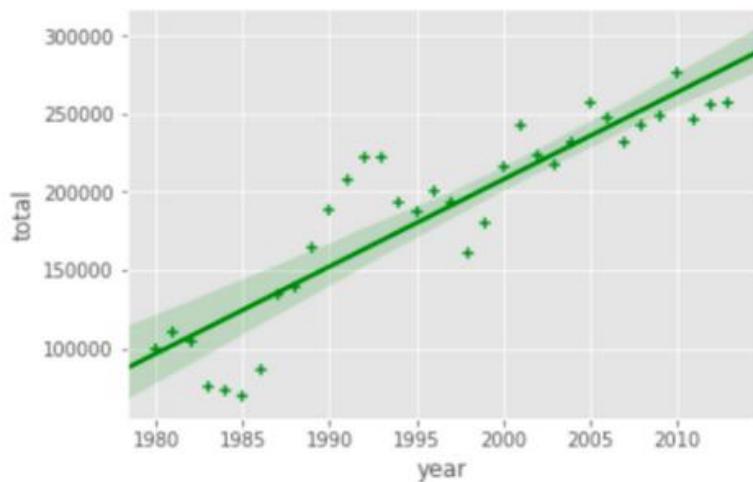
Click here for a sample python solution `python #The correct answer is: # create df_countries dataframe df_countries = df_can.loc[['Denmark', 'Norway', 'Sweden'], years].transpose() # create df_total by summing across three countries for each year df_total = pd.DataFrame(df_countries.sum(axis=1)) # reset index in place df_total.reset_index(inplace=True) # rename columns df_total.columns = ['year', 'total'] # change column year from string to int to create scatter plot df_total['year'] = df_total['year'].astype(int) # define figure size plt.figure(figsize=(15, 10)) # define background style and font size sns.set(font_scale=1.5) sns.set_style('whitegrid') # generate plot and add title and axes labels ax = sns.regplot(x='year', y='total', data=df_total, color='green', marker='+', scatter_kws={'s': 200}) ax.set(xlabel='Year', ylabel='Total Immigration') ax.set_title('Total Immigrationn from Denmark, Sweden, and Norway to Canada from 1980 - 2013')`

Thank you for completing this lab!

Review Question 1

1/1 point (graded)

Which of the choices below will create the following regression line plot, given a *pandas* dataframe, **data_dataframe**?



- import seaborn as sns
 ax = sns.regplot(x="year", y="total", data=data_dataframe, color="green")
- data_dataframe.plot(kind="regression", color="green", marker="+")
- import seaborn as sns
 ax = sns.regplot(x="year", y="total", data=data_dataframe, color="green", marker="+")

- data_dataframe.plot(kind="regplot", color="green", marker="+")

- import seaborn as sns
 ax = sns.regplot(x="total", y="year", data=data_dataframe, color="green")

**Answer**

Correct: Correct

Submit

You have used 1 of 2 attempts

- Correct (1/1 point)

Review Question 2

1/1 point (graded)

In Python, creating a waffle chart is straightforward since we can easily create one using the scripting layer of Matplotlib.

- True

- False

**Answer**

Correct: Correct

Submit

You have used 1 of 1 attempt

Module 5 - Creating Maps and Visualizing Geospatial Data

- Introduction to Folium
- Maps with Markers
- Choropleth Maps
- Hands-on Lab: Generating Maps in Python
- Graded Quiz
- Optional: Download Jupyter Notebook

Learning Objectives

In this lesson you will learn about:

- Folium, a data visualization library in Python.
- Creating maps of different regions of the world and how to superimpose markers on top of a map.
- Creating Choropleth maps with Folium.

Introduction to Folium

We will learn about a very interesting data visualization library in Python which is [Folium](#).

- Folium is a powerful data visualization library in Python that was built primarily to help people visualize geospatial data.
- With Folium, you can create a map of any location in the world as long as you know its latitude and longitude values.
- You can also create a map and superimpose markers as well as clusters of markers on top of the map for cool and very interesting visualizations.
- You can also create maps of different styles such as street level map, stamen map, and a couple others which we will look into in just a moment.

Creating a world map with Folium is pretty straightforward. You simply call the map function and that is all. What is really interesting about the maps created by Folium is that they are interactive, so you can zoom in and out after the map is rendered, which is a super useful feature.

The default map style is the open street map, which shows a street view of an area when you're zoomed in and shows the borders of the world countries when you're zoomed all the way out. Now let's create a world map centred around Canada. To do that, we pass in the latitude and the longitude values of Canada using the location parameter and with Folium you can set the initial zoom level using the zoom start parameter. Now I say initial because you can easily change the zoom level after the map is rendered by zooming in or zooming out.

You can play with this parameter to figure out what the initial zoom level looks like for different values. Now, let's set the zoom level for our map of Canada to 4. And there you go. Here is a world map centred around Canada.

Another amazing feature of Folium is that you can create different map styles using the tiles parameter. Let's create a stamen toner map of Canada. This style is great for visualizing and exploring river meanders and coastal zones. Another style is stamen terrain.

Let's create a map of Canada in stamen terrain. This style is great for visualizing hill shading and natural vegetation colors

Data Visualization with Python

Introduction to Folium

What is Folium?

- Folium is a powerful Python library that helps you create several types of Leaflet maps.
- It enables both the binding of data to a map for choropleth visualizations as well as passing visualizations as markers on the map.
- The library has a number of built-in tilesets from OpenStreetMap, Mapbox, and Stamen, and supports custom tilesets with Mapbox API keys.

Steps on how to create a map?

1.

Creating a World Map

```
# define the world map
world_map = folium.Map()

# display world map
world_map
```



Simply call the map function and that is all. What is really interesting about the maps created by Folium is that they are interactive, so you can zoom in and out after the map is rendered, which is a super useful feature. The default map style is the open street map, which shows a street view of an area when you're zoomed in and shows the borders of the world countries when you're zoomed all the way out.

2.

Creating a Map of Canada

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

# display world map
world_map
```

Creating a Map of Canada

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

# display world map
world_map
```

3.

Creating a Map of Canada

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

# display world map
world_map
```



Map Styles – Stamen Toner

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

# display map
world_map
```



Can create different map styles using the `tiles parameter`. This style is great for visualizing and exploring river meanders and coastal zones.

Map Styles – Stamen Terrain

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

# display map
world_map
```



Map of Canada in `stamen terrain`. This style is great for visualizing hill shading and natural vegetation colors

Maps with Markers

We will continue working with the Folium library and learn how to superimpose markers on top of a map for interesting visualizations. In the previous video, we learned how to create a world map centred around Canada, so let's create this map again and name it Canada_map this time.

Ontario is a Canadian province and contains about 40 percent of the Canadian population. It is considered Canada's most populous province. Let's see how we can add a circular mark to the centre of Ontario. To do that, we need to create what is called a feature group.

Let's go ahead and create a feature group named Ontario. Now when a feature group is created, it is empty and that means what's next is to start creating what is called children and adding them to the feature group. So let's create a child in the form of a red circular mark located at the centre of the Ontario province. We specify the location of the child by passing in its latitude and longitude values. And once we're done adding children to the feature group, we add the featured group to the map. And there you have it: A red circular mark superimposed on top of the map and added to the centre of the province of Ontario. Now, it would be nice if we could actually label this marker in order to let other people know what it actually represents. To do that, we simply use the marker function and the pop up parameter to pass in whatever text we want to add to this marker. And there you go:

Now our marker displays Ontario when clicked on. In the lab session, we will look into a real-world example and explore crime rate in San Francisco. We will create a map of San Francisco and superimpose thousands of these markers on top of the map. Not just that but I'll show you how you can also create clusters of markers in order to make your map look less congested.

Data Visualization with Python

Maps with Markers

Add a Marker

```
# generate map of Canada
canada_map = folium.Map(
    location=[56.130, -106.35],
    zoom_start=4
)
```

```
# display map
```



Add a Marker

```
# generate map of Canada
canada_map = folium.Map(
    location=[56.130, -106.35],
    zoom_start=4
)

## add a red marker to Ontario

# create a feature group
ontario = folium.map.FeatureGroup()

# style the feature group
ontario.add_child(
    folium.features.CircleMarker(
        [51.25, -85.32], radius = 5,
        color = 'red', fill_color = 'Red'
    )
)

# display map
canada_map
```



Add a Marker

```
# generate map of Canada
canada_map = folium.Map(
    location=[56.130, -106.35],
    zoom_start=4
)

## add a red marker to Ontario

# create a feature group
ontario = folium.map.FeatureGroup()

# style the feature group
ontario.add_child(
    folium.features.CircleMarker(
        [51.25, -85.32], radius = 5,
        color = 'red', fill_color = 'Red'
    )
)

# add the feature group to the map
canada_map.add_child(ontario)

# display map
canada_map
```



To Label the Marker (red dot)

Label the Marker

```
# generate map of Canada
canada_map = folium.Map(
    location=[56.130, -106.35],
    zoom_start=4
)

## add a red marker to Ontario

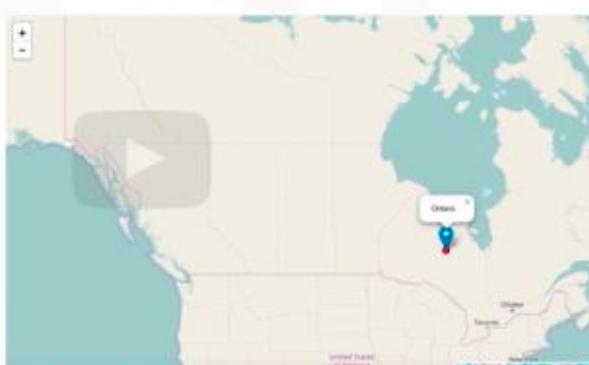
# create a feature group
ontario = folium.map.FeatureGroup()

# style the feature group
ontario.add_child(
    folium.features.CircleMarker(
        [51.25, -85.32], radius = 5,
        color = 'red', fill_color = 'Red'
    )
)

# add the feature group to the map
canada_map.add_child(ontario)

# label the marker
folium.Marker([51.25, -85.32],
    popup='Ontario').add_to(canada_map)

# display map
canada_map
```



Choropleth Maps

We will learn how to create a special type of map called **choropleth map with Folium**. I'm sure that most of you have seen maps similar to this one and this one. These are what we call choropleth maps.

So what is a choropleth map? A **choropleth map** is

- a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per capita income.
- **The higher the measurement the darker the color.**

So, the map to the left is a choropleth map of the world showing infant mortality rate per 1000 births. The darker the color the higher the infant mortality rate. According to the map, African countries have very high infant mortality rates with some of them reporting a rate that is higher than 160 per 1000 births.

Similarly, the map to the right is a choropleth map of the US showing population per square mile by state. Again, the darker the color the higher the population. According to the map, states in the eastern part of the US tend to be more populous than states in the western part, with California being an exception. In order to create a choropleth map of a region of interest,

Folium requires a Geo JSON file that includes geospatial data of the region.

For a choropleth map of the world, we would need a Geo JSON file that lists each country along with any geospatial data to define its borders and boundaries. Here is an example of what the Geo JSON file would include about each country.

The example here pertains to the country Brunei. As you can see, the file includes the country's name, it's ID, geometry shape, and the coordinates that define the country's borders and boundaries. So let's see how we can create a choropleth map of the world like this one showing immigration to Canada. Before we go over the code to do that, let's do a quick recap of our dataset. Recall that each row represents a country and contains metadata about the country such as where it is located geographically and whether it is developing or developed. Each row also contains numerical figures of annual immigration from that country to Canada from 1980 to 2013.

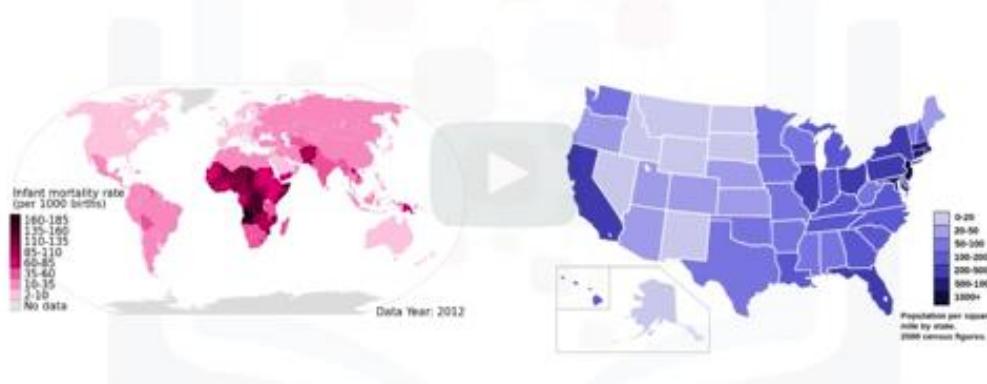
Now let's process the data, and let's add an extra column which represents the cumulative sum of annual immigration from each country from 1980 to 2013. So for Afghanistan for example, it is 58,639, total, and for Albania it is 15,699, and so on. And let's name our dataframe `df_Canada`. So now that we know how our data is stored in the dataframe, `df_Canada`, let's see how we can generate a choropleth map of the world showing immigration to Canada. We should be experts now in creating world maps with Folium.

So let's go ahead and create a world map, but this time let's use the mapbox bright tiles set. The result is a nice world map displaying the name of every country. Now to convert this map into a choropleth map, we first define a variable that points to our Geo JSON file. Then we apply the choropleth function to our world map and we tell it to use the columns "Country" and "Total" in our `df_Canada` dataframe, and to use the country names to look up the geospatial information about each country in the Geo JSON file. And there you have it: A choropleth map of Canada showing the intensity of immigration from different countries worldwide

Data Visualization with Python

Choropleth Maps

Choropleth Maps



A **choropleth map** is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per capita income. The higher the measurement the darker the color.

So, the map to the left is a choropleth map of the world showing infant mortality rate per 1000 births. The darker the color the higher the infant mortality rate. According to the map, African countries have very high infant mortality rates with some of them reporting a rate that is higher than 160 per 1000 births.

Similarly, the map to the right is a choropleth map of the US showing population per square mile by state. Again, the darker the color the higher the population. According to the map, states in the eastern part of the US tend to be more populous than states in the western part, with California being an exception.

Geojson File

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "name": "Brunei"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              [114.284017, 4.535874], [114.599961, 4.988611], [115.458711, 5.44771],
              [115.4857, 4.955228], [115.347461, 4.316636], [114.889557, 4.348314],
              [114.659596, 4.088767], [114.204817, 4.525874]
            ]
          ]
        ]
      }
    }
  ],
  "id": "BRN"
}
```

The **Geo JSON file** would include about each country. The example here pertains to the country Brunei. As you can see, the file includes the country's name, it's ID, geometry shape, and the coordinates that define the country's borders and boundaries.

Creating the Map



Dataset - Recap

Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	

Dataset - Processed

Type	Coverage	Country	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total	
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2978	3436	3009	2652	2111	1746	1758	2203	2635	2004	58839
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	1450	1223	856	702	560	716	561	539	620	603	15699
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3616	3626	4807	3623	4005	5393	4752	4325	3774	4331	69439
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	1	0	0	0	0	0	0	6	
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	0	0	1	1	0	0	0	1	1	15	

df_canada

Creating the Map

```
# create a plain world map
world_map = folium.Map(
    zoom_start=2,
    tiles='Mapbox Bright'
)

## geojson file
world_geo = r'world_countries.json'

# generate choropleth map using the total
# population of each country to Canada from
# 1980 to 2013
world_map.choropleth(
    geo_path=world_geo,
    data=df_canada,
    columns=['Country', 'Total'],
    key_on='feature.properties.name',
    fill_color='YlOrRd',
    legend_name='Immigration to Canada'
)

# display map
world_map
```



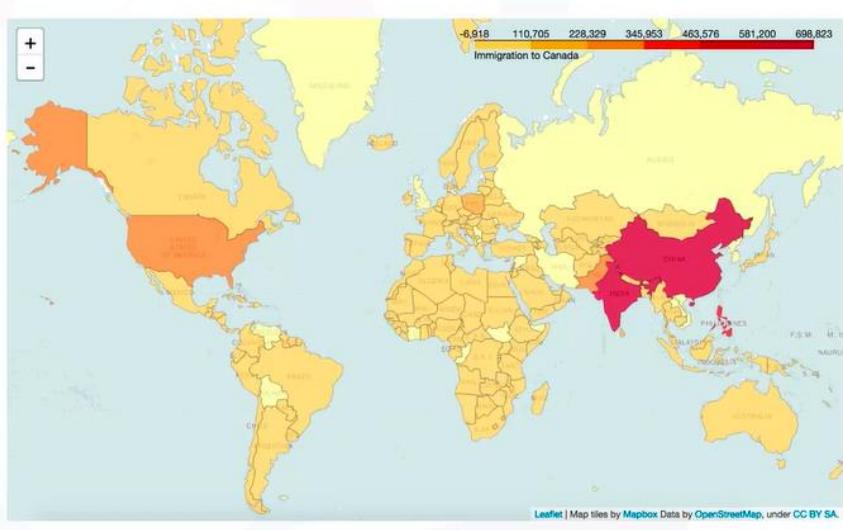
Creating the Map

```
# create a plain world map
world_map = folium.Map(
    zoom_start=2,
    tiles='Mapbox Bright'
)

## geojson file
world_geo = r'world_countries.json'

# generate choropleth map using the total
# population of each country to Canada from
# 1980 to 2013
world_map.choropleth(
    geo_path=world_geo,
    data=df_canada,
    columns=['Country', 'Total'],
    key_on='feature.properties.name',
    fill_color='YlOrRd',
    legend_name='Immigration to Canada'
)

# display map
world_map
```



A choropleth map of Canada showing the intensity of immigration from different countries worldwide.

3-2-Generating-Maps-in-Python

Lab 3-2: Creating Maps and Visualizing Geospatial Data



Generating Maps with Python

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Visualize geospatial data with Folium

Introduction

In this lab, we will learn how to create maps for different objectives. To do that, we will part ways with Matplotlib and work with another Python visualization library, namely **Folium**. What is nice about **Folium** is that it was developed for the sole purpose of visualizing geospatial data. While other libraries are available to visualize geospatial data, such as **plotly**, they might have a cap on how many API calls you can make within a defined time frame. **Folium**, on the other hand, is completely free.

Table of Contents

1. [Exploring Datasets with *pandas*](#0)
2. [Downloading and Prepping Data](#2)
3. [Introduction to Folium](#4)
4. [Map with Markers](#6)
5. [Choropleth Maps](#8)

Exploring Datasets with *pandas* and Matplotlib

Toolkits: This lab heavily relies on *pandas* and *Numpy* for data wrangling, analysis, and visualization. The primary plotting library we will explore in this lab is [Folium](#).

Datasets:

1. San Francisco Police Department Incidents for the year 2016 - [Police Department Incidents](#) from San Francisco public data portal. Incidents derived from San Francisco Police Department (SFPD) Crime Incident Reporting system. Updated daily, showing data for the entire year of 2016. Address and location has been anonymized by moving to mid-block or to an intersection.

- Immigration to Canada from 1980 to 2013 - [International migration flows to and from selected countries - The 2015 revision](#) from United Nation's website. The dataset contains annual data on the flows of international migrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. For this lesson, we will focus on the Canadian Immigration data

Downloading and Prepping Data

Import Primary Modules:

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

Introduction to Folium

Folium is a powerful Python library that helps you create several types of Leaflet maps. The fact that the Folium results are interactive makes this library very useful for dashboard building.

From the official Folium documentation page:

Folium builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library. Manipulate your data in Python, then visualize it in on a Leaflet map via Folium.

Folium makes it easy to visualize data that's been manipulated in Python on an interactive Leaflet map. It enables both the binding of data to a map for choropleth visualizations as well as passing Vincent/Vega visualizations as markers on the map.

The library has a number of built-in tilesets from OpenStreetMap, Mapbox, and Stamen, and supports custom tilesets with Mapbox or Cloudmade API keys. Folium supports both GeoJSON and TopoJSON overlays, as well as the binding of data to those overlays to create choropleth maps with color-brewer color schemes.

Let's install Folium

Folium is not available by default. So, we first need to install it before we are able to import it.

```
In [2]: !conda install -c conda-forge folium=0.5.0 --yes
import folium

print('Folium installed and imported!')

Collecting package metadata (current_repotdata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

environment location: /opt/conda/envs/Python-3.8-main

added / updated specs:
- folium=0.5.0

The following packages will be downloaded:

package          |      build
-----|-----
altair-4.1.0     |      py_1      614 KB  conda-forge
branca-0.4.2     |  pyhd8ed1ab_0   26 KB  conda-forge
certifi-2021.10.8 |  py38h578d9bd_1  145 KB  conda-forge
folium-0.5.0      |      py_0      45 KB  conda-forge
python_abi-3.8    |      2_cp38     4 KB   conda-forge
vincent-0.4.4     |      py_1      28 KB  conda-forge
widgetsnbextension-3.5.1 |  py38h578d9bd_4  1.8 MB  conda-forge
-----|-----
Total:           2.6 MB

The following NEW packages will be INSTALLED:

altair          conda-forge/noarch::altair-4.1.0-py_1
branca          conda-forge/noarch::branca-0.4.2-pyhd8ed1ab_0
folium          conda-forge/noarch::folium-0.5.0-py_0
python_abi       conda-forge/linux-64::python_abi-3.8-2_cp38
vincent         conda-forge/noarch::vincent-0.4.4-py_1

The following packages will be UPDATED:

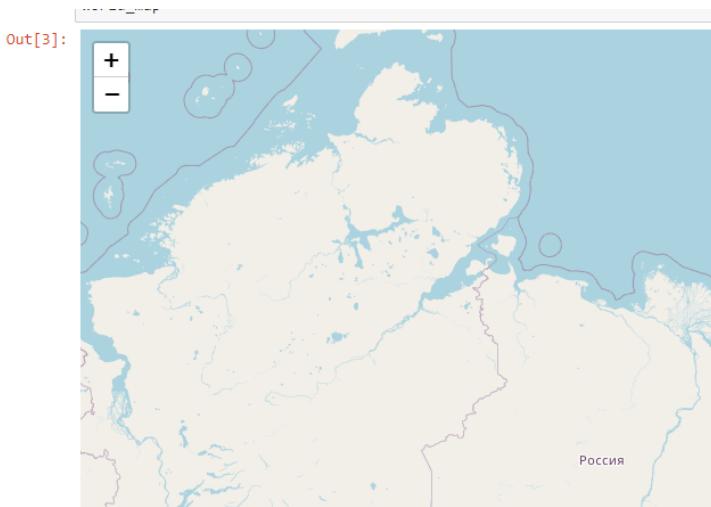
certifi          pkgs/main::certifi-2021.10.8-py38h06a~ --> conda-forge::certifi-2021.10.8-py38h578d9bd_1
widgetsnbextension pkgs/main::widgetsnbextension-3.5.1-p~ --> conda-forge::widgetsnbextension-3.5.1-py38h578d9bd_4

Downloading and Extracting Packages
folium-0.5.0      | 45 KB  | #####| 100%
branca-0.4.2      | 26 KB  | #####| 100%
widgetsnbextension-3 | 1.8 MB | #####| 100%
vincent-0.4.4     | 28 KB  | #####| 100%
certifi-2021.10.8 | 145 KB | #####| 100%
altair-4.1.0       | 614 KB | #####| 100%
python_abi-3.8    | 4 KB   | #####| 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: | Exception while loading config file /var/pod/.ws/ax-ext/config/wscloud/jupyter_notebook_config.py
Traceback (most recent call last):
  File "/opt/conda/envs/Python-3.8-main/lib/python3.8/site-packages/traitlets/config/application.py", line 737, in _load_config_files
    config = loader.load_config()
  File "/opt/conda/envs/Python-3.8-main/lib/python3.8/site-packages/traitlets/config/loader.py", line 616, in load_config
    self._read_file_as_dict()
  File "/opt/conda/envs/Python-3.8-main/lib/python3.8/site-packages/traitlets/config/loader.py", line 648, in _read_file_dict
    exec(compile(f.read(), conf_filename, 'exec'), namespace, namespace)
  File "/var/pod/.ws/ax-ext/config/wscloud/jupyter_notebook_config.py", line 17, in <module>
    from cdsax_jupyter_extensions.ax_log import ax_log_request
ModuleNotFoundError: No module named 'cdsax_jupyter_extensions'
Enabling notebook extension jupyter-js-widgets/extension...
- Validating: OK

done
Folium installed and imported!
```

Generating the world map is straightforward in **Folium**. You simply create a **Folium Map** object, and then you display it. What is attractive about **Folium** maps is that they are interactive, so you can zoom into any region of interest despite the initial zoom level.

```
In [3]: # define the world map  
world_map = folium.Map()  
  
# display world map  
world_map
```



Go ahead. Try zooming in and out of the rendered map above.

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

All locations on a map are defined by their respective *Latitude* and *Longitude* values. So you can create a map and pass in a center of *Latitude* and *Longitude* values of **[0, 0]**.

For a defined center, you can also define the initial zoom level into that location when the map is rendered. **The higher the zoom level the more the map is zoomed into the center.**

Let's create a map centered around Canada and play with the zoom level to see how it affects the rendered map.

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

Out[4]:

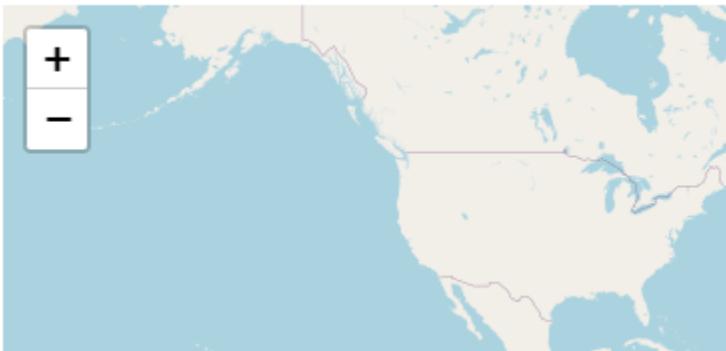


As you can see, the higher the zoom level the more the map is zoomed into the given center.

Question: Create a map of Mexico with a zoom level of 4.

```
In [6]: mexico_latitude = 23.6345  
mexico_longitude = -102.5528  
  
world_map = folium.Map(location=[mexico_latitude, mexico_longitude], zoom_start=4)  
  
world_map
```

Out[6]:



[Click here for a sample python solution](#)

Another cool feature of **Folium** is that you can generate different map styles.

A. Stamen Toner Maps

These are high-contrast B+W (black and white) maps. They are perfect for data mashups and exploring river meanders and coastal zones.

Let's create a Stamen Toner map of Canada with a zoom level of 4.

```
In [7]: # create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Toner')

# display map
world_map
```

Out[7]:



Feel free to zoom in and out to see how this style compares to the default one.

B. Stamen Terrain Maps

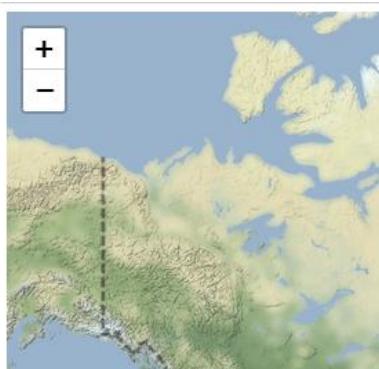
These are maps that feature hill shading and natural vegetation colors. They showcase advanced labeling and linework generalization of dual-carriageway roads.

Let's create a Stamen Terrain map of Canada with zoom level 4.

```
In [8]: # create a Stamen Toner map of the world centered around Canada
world_map = folium.Map(location=[56.130, -106.35], zoom_start=4, tiles='Stamen Terrain')

# display map
world_map
```

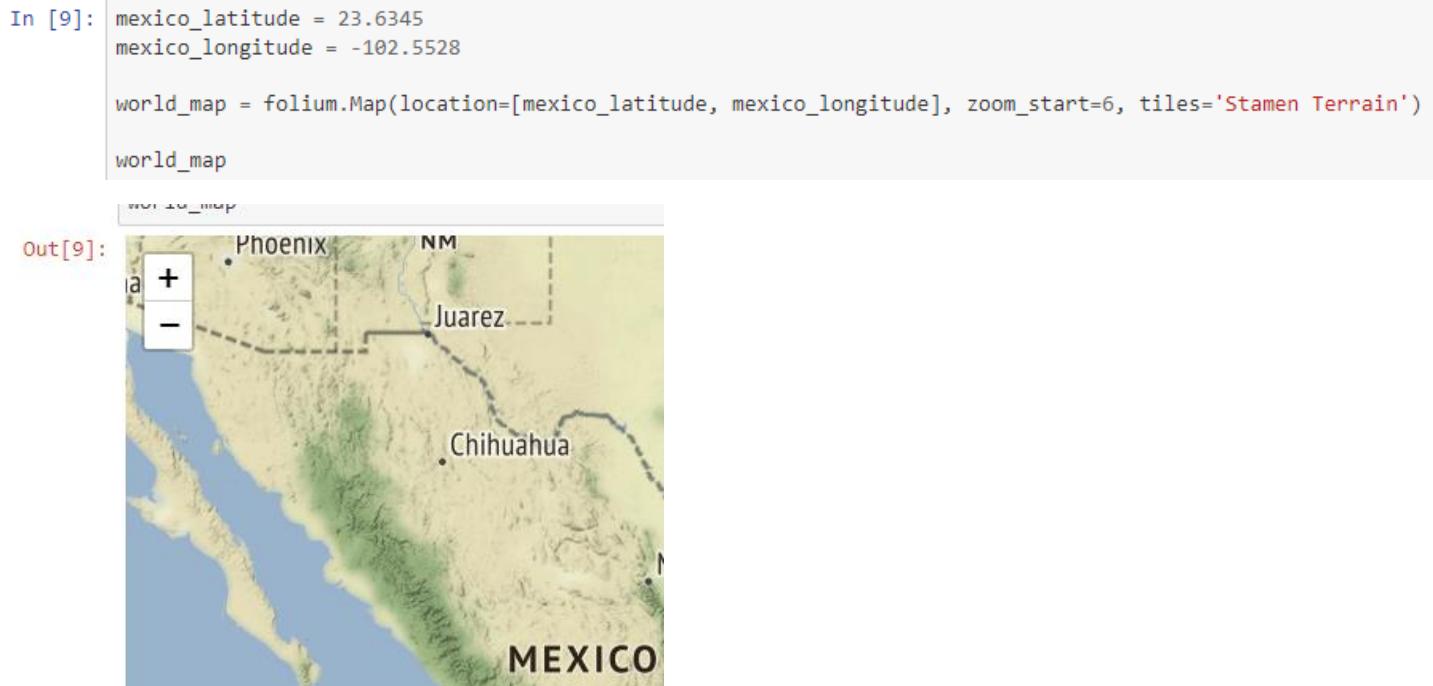
Out[8]:



Feel free to zoom in and out to see how this style compares to Stamen Toner, and the default style.

Zoom in and notice how the borders start showing as you zoom in, and the displayed country names are in English.

Question: Create a map of Mexico to visualize its hill shading and natural vegetation. Use a zoom level of 6.



[Click here for a sample python solution](#)

Maps with Markers

Let's download and import the data on police department incidents using *pandas read_csv()* method.

Download the dataset and read it into a *pandas* dataframe:

```
In [10]: df_incidents = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Police_Department_Incidents_-_Previous_Year__2016_.csv')
print('Dataset downloaded and read into a pandas dataframe!')
```

Dataset downloaded and read into a pandas dataframe!

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

```
In [11]: df_incidents.head()
```

Out[11]:

	IncidentNum	Category	Descript	DayOfWeek	Date	Time	PdDistrict	Resolution	Address	X	Y
0	120058272	WEAPON LAWS	POSS OF PROHIBITED WEAPON	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 Block of BRYANT ST	-122.403405	37.775421
1	120058272	WEAPON LAWS	FIREARM, LOADED, IN VEHICLE, POSSESSION OR USE	Friday	01/29/2016 12:00:00 AM	11:00	SOUTHERN	ARREST, BOOKED	800 Block of BRYANT ST	-122.403405	37.775421
2	141059263	WARRANTS	WARRANT ARREST	Monday	04/25/2016 12:00:00 AM	14:59	BAYVIEW	ARREST, BOOKED	KEITH ST / SHAFTER AV	-122.388856	37.729981
3	160013662	NON-CRIMINAL	LOST PROPERTY	Tuesday	01/05/2016 12:00:00 AM	23:50	TENDERLOIN	NONE	JONES ST / OFARRELL ST	-122.412971	37.785788
4	160002740	NON-CRIMINAL	LOST PROPERTY	Friday	01/01/2016 12:00:00 AM	00:30	MISSION	NONE	16TH ST / MISSION ST	-122.419672	37.765050

Location	PdId
(37.775420706711, -122.403404791479)	12005827212120
(37.775420706711, -122.403404791479)	12005827212168
(37.7299809672996, -122.388856204292)	14105926363010
(37.7857883766888, -122.412970537591)	16001366271000
(37.7650501214668, -122.419671780296)	16000274071000

So each row consists of **13 features**:

1. **IncidentNum**: Incident Number
2. **Category**: Category of crime or incident
3. **Descript**: Description of the crime or incident
4. **DayOfWeek**: The day of week on which the incident occurred
5. **Date**: The Date on which the incident occurred
6. **Time**: The time of day on which the incident occurred
7. **PdDistrict**: The police department district
8. **Resolution**: The resolution of the crime in terms whether the perpetrator was arrested or not
9. **Address**: The closest address to where the incident took place
10. **X**: The longitude value of the crime location
11. **Y**: The latitude value of the crime location
12. **Location**: A tuple of the latitude and the longitude values
13. **PdId**: The police department ID

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

```
In [12]: df_incidents.shape
```

```
Out[12]: (150500, 13)
```

So the dataframe consists of 150,500 crimes, which took place in the year 2016. In order to reduce computational cost, let's just work with the first 100 incidents in this dataset.

```
In [13]: # get the first 100 crimes in the df_incidents dataframe
limit = 100
df_incidents = df_incidents.iloc[0:limit, :]
```

Let's confirm that our dataframe now consists only of 100 crimes.

```
In [14]: df_incidents.shape
```

```
Out[14]: (100, 13)
```

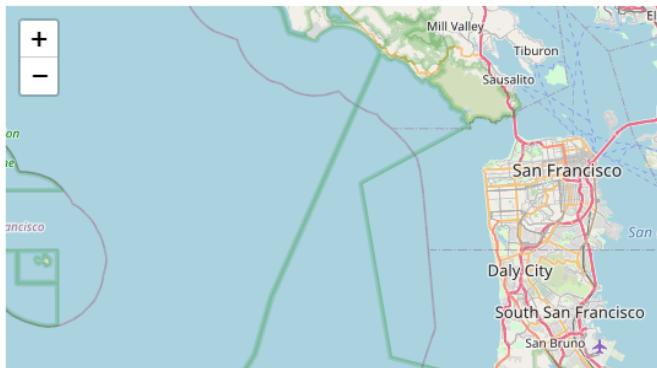
Now that we reduced the data a little, let's visualize where these crimes took place in the city of San Francisco. We will use the default style, and we will initialize the zoom level to 12.

```
In [15]: # San Francisco Latitude and Longitude values
latitude = 37.77
longitude = -122.42
```

```
In [16]: # create map and display it
sanfran_map = folium.Map(location=[latitude, longitude], zoom_start=12)

# display the map of San Francisco
sanfran_map
```

Out[16]:

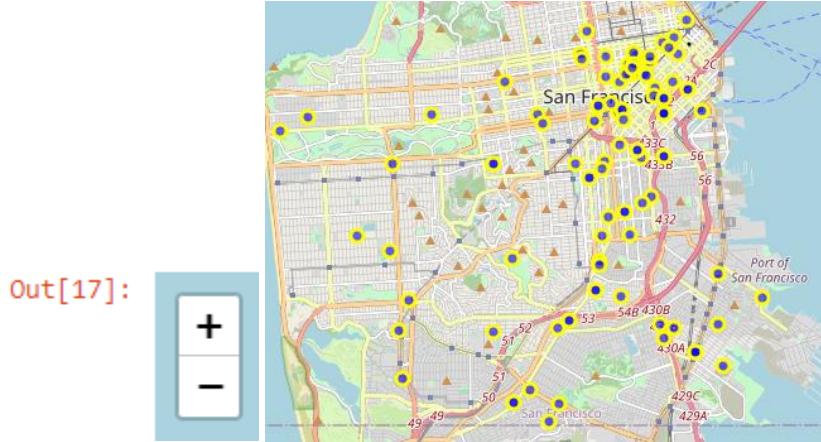


Now let's superimpose the locations of the crimes onto the map. The way to do that in **Folium** is to create a **feature group** with its own features and style and then add it to the `sanfran_map`.

```
In [17]: # instantiate a feature group for the incidents in the dataframe
incidents = folium.map.FeatureGroup()

# Loop through the 100 crimes and add each to the incidents feature group
for lat, lng, in zip(df_incidents.Y, df_incidents.X):
    incidents.add_child(
        folium.features.CircleMarker(
            [lat, lng],
            radius=5, # define how big you want the circle markers to be
            color='yellow',
            fill=True,
            fill_color='blue',
            fill_opacity=0.6
        )
    )

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



You can also add some pop-up text that would get displayed when you hover over a marker. Let's make each marker display the category of the crime when hovered over.

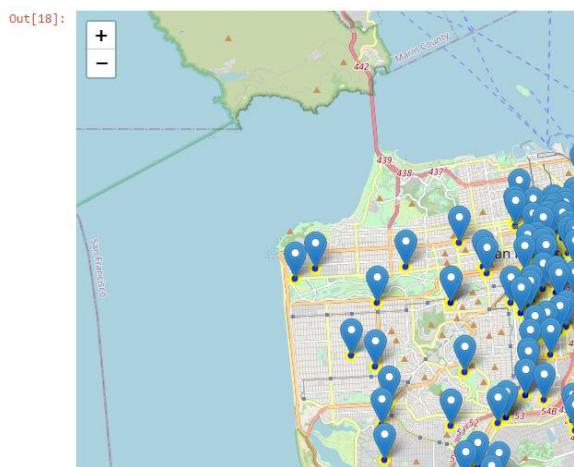
```
In [18]: # instantiate a feature group for the incidents in the dataframe
incidents = folium.map.FeatureGroup()

# Loop through the 100 crimes and add each to the incidents feature group
for lat, lng, in zip(df_incidents.Y, df_incidents.X):
    incidents.add_child(
        folium.features.CircleMarker(
            [lat, lng],
            radius=5, # define how big you want the circle markers to be
            color='yellow',
            fill=True,
            fill_color='blue',
            fill_opacity=0.6
        )
    )

# add pop-up text to each marker on the map
latitudes = list(df_incidents.Y)
longitudes = list(df_incidents.X)
labels = list(df_incidents.Category)

for lat, lng, label in zip(latitudes, longitudes, labels):
    folium.Marker([lat, lng], popup=label).add_to(sanfran_map)

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



Isn't this really cool? Now you are able to know what crime category occurred at each marker.

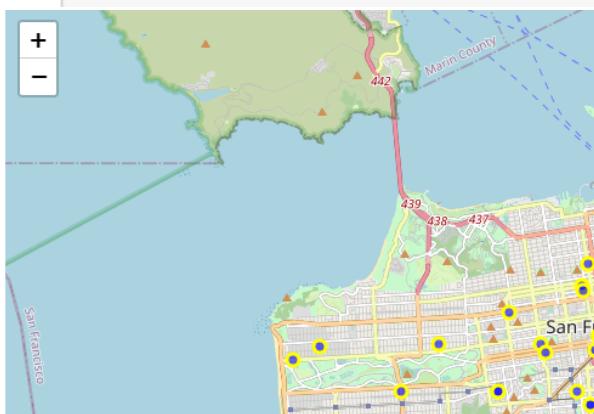
If you find the map to be so **congested** will all these markers, there are two remedies to this problem. The simpler solution is to remove these location markers and just add the text to the circle markers themselves as follows:

```
In [19]: # create map and display it
sanfran_map = folium.Map(location=[latitude, longitude], zoom_start=12)

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

# show map
sanfran_map
```

Out[19]:



The other proper remedy is to group the markers into different clusters. Each cluster is then represented by the number of crimes in each neighborhood. These clusters can be thought of as pockets of San Francisco which you can then analyze separately.

To implement this, we start off by instantiating a `MarkerCluster` object and adding all the data points in the dataframe to this object.

```
In [20]: from folium import plugins

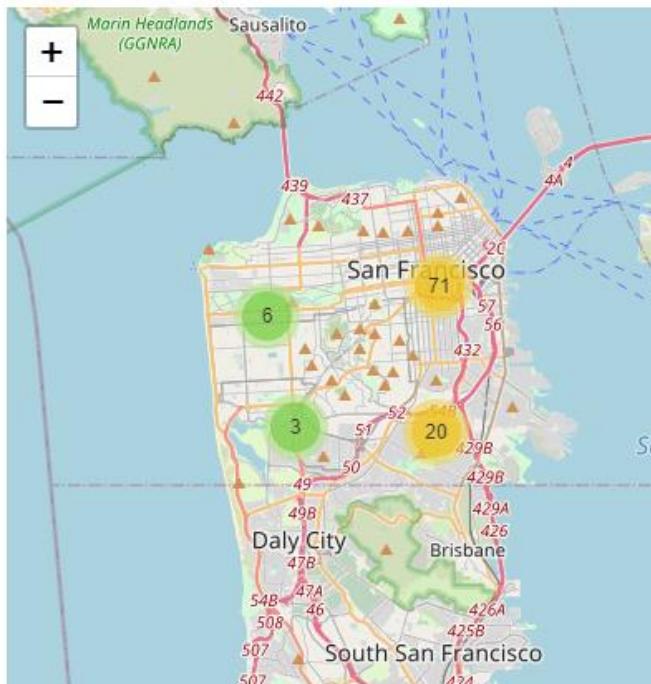
# Let's start again with a clean copy of the map of San Francisco
sanfran_map = folium.Map(location = [latitude, longitude], zoom_start = 12)

# instantiate a mark cluster object for the incidents in the dataframe
incidents = plugins.MarkerCluster().add_to(sanfran_map)

# Loop through the dataframe and add each data point to the mark cluster
for lat, lng, label, in zip(df_incidents.Y, df_incidents.X, df_incidents.Category):
    folium.Marker(
        location=[lat, lng],
        icon=None,
        popup=label,
    ).add_to(incidents)

# display map
sanfran_map
```

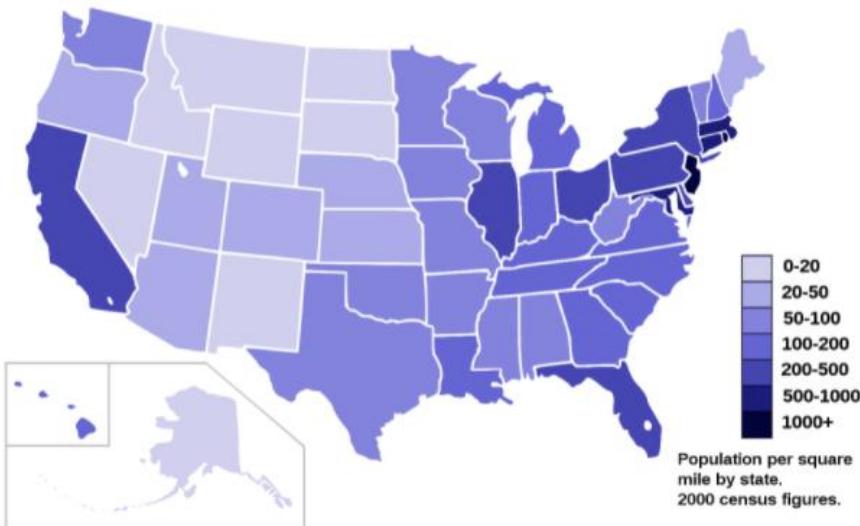
Out[20]:



Notice how when you zoom out all the way, all markers are grouped into one cluster, *the global cluster*, of 100 markers or crimes, which is the total number of crimes in our dataframe. Once you start zooming in, the *global cluster* will start breaking up into smaller clusters. Zooming in all the way will result in individual markers.

Choropleth Maps

A Choropleth map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per-capita income. The choropleth map provides an easy way to visualize how a measurement varies across a geographic area, or it shows the level of variability within a region. Below is a Choropleth map of the US depicting the population by square mile per state.



Now, let's **create our own Choropleth map of the world depicting immigration from various countries to Canada.**

Let's first download and import our primary Canadian immigration dataset using *pandas* `read_excel()` method. Normally, before we can do that, we would need to download a module which *pandas* requires reading in Excel files. This module was **openpyxl** (formerly **xlrd**). For your convenience, we have pre-installed this module, so you would not have to worry about that. Otherwise, you would need to run the following line of code to install the **openpyxl** module:

```
! pip3 install openpyxl
```

Download the dataset and read it into a *pandas* dataframe:

```
In [21]: df_can = pd.read_excel(
    'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20
Files/Canada.xlsx',
    sheet_name='Canada by Citizenship',
    skiprows=range(20),
    skipfooter=2)

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

Data downloaded and read into a dataframe!
```

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

Out[22]:

	Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName	1980	...	2012	2013	Unnamed: 43
0	Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions	16	...	2635	2004	NaN
1	Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions	1	...	620	603	NaN
2	Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions	80	...	3774	4331	NaN
3	Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions	0	...	0	0	NaN
4	Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions	0	...	1	1	NaN

5 rows × 51 columns

Unnamed: 43	Unnamed: 44	Unnamed: 45	Unnamed: 46	Unnamed: 47	Unnamed: 48	Unnamed: 49	Unna
NaN	NaN						
NaN	NaN						
NaN	NaN						
NaN	NaN						
NaN	NaN						

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

In [23]: # print the dimensions of the dataframe
print(df_can.shape)

(195, 51)

Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to *Introduction to Matplotlib and Line Plots* and *Area Plots, Histograms, and Bar Plots* notebooks for a detailed description of this preprocessing.

```
In [24]: # clean up the dataset to remove unnecessary columns (eg. REG)
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'], axis=1, inplace=True)

# let's rename the columns so that they make sense
df_can.rename(columns={'OdName':'Country', 'AreaName':'Continent','RegName':'Region'}, inplace=True)

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

# add total column
df_can['Total'] = df_can.sum(axis=1)

# years that we will be using in this Lesson - useful for plotting later on
years = list(map(str, range(1980, 2014)))
print ('data dimensions:', df_can.shape)

data dimensions: (195, 47)
```

Let's take a look at the first five items of our cleaned dataframe.

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

Out[25]:

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

5 rows × 47 columns

Unnamed: 43	Unnamed: 44	Unnamed: 45	Unnamed: 46	Unnamed: 47	Unnamed: 48	Unnamed: 49	Unnamed: 50	Total
NaN	58639.0							
NaN	15699.0							
NaN	69439.0							
NaN	6.0							
NaN	15.0							

In order to create a Choropleth map, we need a **GeoJSON file** that defines the areas/boundaries of the state, county, or **country that we are interested in**. In our case, since we are endeavoring to create a world map, we want a GeoJSON that defines the boundaries of all world countries. For your convenience, we will be providing you with this file, so let's go ahead and download it. Let's name it **world_countries.json**.

```
In [26]: # download countries geojson file
! wget --quiet https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/world_countries.json

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

GeoJSON file downloaded!

Now that we have the GeoJSON file, let's create a world map, centered around **[0, 0]** latitude and longitude values, with an initial zoom level of 2.

```
In [27]: world_geo = r'world_countries.json' # geojson file

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

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

1. **geo_data**, which is the **GeoJSON file**.
2. **data**, which is the dataframe containing the data.
3. **columns**, which represents the columns in the dataframe that will be used to create the Choropleth map.
4. **key_on**, which is the key or variable in the GeoJSON file that contains the name of the variable of interest. To determine that, you will need to open the GeoJSON file using any text editor and note the name of the key or variable that contains the name of the countries, since the countries are our variable of interest. In this case, **name** is the key in the GeoJSON file that contains the name of the

5. countries. Note that this key is case_sensitive, so you need to pass exactly as it exists in the GeoJSON file.

```
In [28]: # generate choropleth map using the total immigration of each country to Canada from 1980 to 2013
world_map.choropleth(
    geo_data=world_geo,
    data=df_can,
    columns=['Country', 'Total'],
    key_on='feature.properties.name',
    fill_color='YlOrRd',
    fill_opacity=0.7,
    line_opacity=0.2,
    legend_name='Immigration to Canada'
)

# display map
world_map
```

Out[28]:



As per our Choropleth map legend, the darker the color of a country and the closer the color to red, the higher the number of immigrants from that country. Accordingly, the highest immigration over the course of 33 years (from 1980 to 2013) was from China, India, and the Philippines, followed by Poland, Pakistan, and interestingly, the US.

Notice how the legend is displaying a negative boundary or threshold. Let's fix that by defining our own thresholds and starting with 0 instead of -6,918!

```
In [29]: world_geo = r'world_countries.json'

# create a numpy array of length 6 and has linear spacing from the minimum total immigration to the maximum total immigration
threshold_scale = np.linspace(df_can['Total'].min(),
                               df_can['Total'].max(),
                               6, dtype=int)
threshold_scale = threshold_scale.tolist() # change the numpy array to a list
threshold_scale[-1] = threshold_scale[-1] + 1 # make sure that the last value of the list is greater than the maximum immigration
n

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

Out[29]:



Much better now! Feel free to play around with the data and perhaps create Choropleth maps for individuals years, or perhaps decades, and see how they compare with the entire period from 1980 to 2013.

Thank you for completing this lab!

Review Question 1

1/1 point (graded)

If you are interested in generating a map of Spain to visualize its hill shading and natural vegetation, which of the following lines of code will create the right map for you? (Try and code yourself before answering this question.)

folium.Map(location=[40.4637, 3.7492], zoom_start=6, tiles='Stamen Terrain')

folium.Map(location=[40.4637, 3.7492], zoom_start=6)

folium.Map(location=[40.4637, -3.7492], zoom_start=6, tiles='Stamen Terrain')

folium.Map(location=[40.4637, 3.7492], zoom_start=6, tiles='Stamen Toner')

folium.Map(location=[-40.4637, -3.7492], zoom_start=6, tiles='Stamen Toner')



Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

Save

Review Question 2

1/1 point (graded)

You cluster markers superimposed onto a map in Folium using a *feature group* object.

False

True



Submit

You have used 1 of 1 attempt

Correct (1/1 point)

Review question 3

1/1 point (graded)

A choropleth map is a thematic map in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map.

True

False



Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

Module 6 - Creating Dashboards with Plotly and Dash

- Dashboarding Overview
- Additional Resources for Dashboards
- Introduction to Plotly
- Additional Resources for Plotly
- Hands-on Lab: Plotly Basics - Scatter, Line, Bar, Bubble, Histogram, Pie, Sunburst
- Introduction to Dash
- Additional Resources for Dash
- Hands-on Lab: Dash Basics - HTML and Core Components, Multiple Charts
- Make Dashboards Interactive
- Additional Resources for Interactive Dashboards
- Add Interactivity: User Input and Callbacks
- Hands-on Lab: Flight Delay Time Statistics Dashboard
- Lesson Summary
- Graded Quiz

Final Assignment

- Final Assignment
- Peer Graded Assignment

Final Exam

- Final Exam

Module Introduction

In this module, you will get started with dashboard creation using the Plotly library. You will create a dashboard with the theme "US Domestic Airline Flights Performance". You will do this using a US airline reporting carrier on-time performance dataset, Plotly, and Dash concepts learned throughout the course. Hands-on labs will follow each concept to make you comfortable with using the library. Reading lists will reference additional resources to learn more about the concepts covered.

Learning Objectives

- Identify high-level popular Python dashboarding tools.
- Demonstrate basic Plotly, Plotly.graph_objects, and Plotly express commands.
- Demonstrate using Dash and basic Dash components (core and HTML).
- Demonstrate adding different dashboard elements including text boxes, dropdowns, graphs, and others.
- Apply interactivity to dash core and HTML components.
- Describe how a dashboard can be used to answer critical business questions.

Dashboarding Overview

We are going to see how an interactive data application can help improve business performance, and the tools available for building the application. With real-time visuals on the dashboard, understanding business moving parts becomes easy.

Based on the report type and data, suitable graphs and charts can be created in one central location. This provides an easy way for stakeholders to understand what is going right, wrong, and what needs to be improved. Getting the big-picture in one place can help businesses make informed decisions. This improves business performance.

In general, the best dashboards answer critical business questions. Let's say you are assigned a task to monitor and report the performance of domestic US flights. Following are the yearly review report items. The top 10 airline carrier in the year 2019 in terms of number of flights The number of flights in 2019 split by month

And the number of travelers from California state to other states split by distance group Let's look at the two ways of presenting the report.

For this type 1 report, the information is presented through tables, with inference from tables documented for reference. For report type 2 we are presenting the same report in the dashboard format. Hovering over each chart will provide details about the data points. In the bottom sunburst chart, you can click on different numbers, drill down into levels and get detailed information about each segment.

Can you observe the difference in the presentation of the findings? What if we need to get the report on the real-time data, not the static data? Also, presenting the result using tables and documents is time-consuming, less visually appealing, and more difficult to comprehend.

A data scientist should be able to create and deliver a story around the finding in a way stakeholders can easily understand. With that in mind, dashboards are the way to go.

Let's take a look at web-based dashboarding tool options available in Python.

Dash. is a

- python framework for building web analytic applications.
- It is written on top of Flask, Plotly.js, and React.js.
- Dash is well-suited for building data visualization apps with highly custom user interfaces.

Panel

- works with visualizations from Bokeh, Matplotlib, HoloViews, and many other Python plotting libraries, making them instantly viewable either individually or when combined with interactive widgets that control them.
- Panel works equally well in Jupyter Notebooks, for creating quick data-exploration tools. Panel can also be used in standalone deployed apps and dashboards, allowing you to easily switch between those contexts as needed.

Voila

- turns Jupyter notebooks into standalone web applications.
- It can be used with separate layout tools like jupyter-flex or templates like voila-vuetify.

Streamlit

- can easily turn data scripts into shareable web apps with 3 main principles:
- embrace python scripting, treat widgets as variables, and reuse data and computation.

There are other tools that can be used for dashboarding:

Bokeh

- is a plotting library, a widget and app library.
- It acts as a server for both plots and dashboards.

Panel is one of the web-based dashboarding tools built on Bokeh.

ipywidgets provides an array of Jupyter-compatible widgets and an interface supported by many Python libraries, but sharing as a dashboard requires a separate deployable server like Voila.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Bowtie allows users to build dashboards in pure Python.

Flask is a Python-backed web server that can be used to build arbitrary web sites, including those with Python plots that function as flask dashboards.

Learn more about the tools from the source link. In this course, we will be focusing on Dash.

Dashboards and Interactive Data Applications

Dashboard

- Real-time visuals
- Understand business moving parts
- Visually track, analyze, and display key performance indicators (KPI)
- Take informed decisions and improve performance
- Reduced hours of analyzing

Best dashboards answer important business questions.

Scenario

Monitor and report US airline performance

Requested report items

1. Top 10 airline carrier in year 2019 in terms of number of flights
2. Number of flights in 2019 split by month
3. Number of travelers from California (CA) to other states split by distance group

2 Ways of Presenting the Report

Type 1 - without dashboard

Reporting_Airline	Flights
WN	13972
DL	10241
AA	9853
OO	8515
UA	6430
YX	3416
MQ	3368
B6	3170
OH	2969
AS	2783

Month	Flights
1	6125
2	5578
3	6553
4	6282
5	6441
6	6583
7	6798
8	6753
9	6140
10	6533
11	6280
12	6550

DestState	DistanceGroup	Flights
AK	10	4
AL	8	1
AR	7	1
AZ	2	361
AZ	3	220
AZ	4	9
CA	1	283
CA	2	2256
CA	3	36
CO	3	19
CO	4	401
CO	5	2
CT	11	2
FL	9	51
FL	10	90
FL	11	26
GA	8	114
GA	9	65
HI	10	168
HI	11	116
ID	2	7
ID	3	60
IL	7	169
IL	8	154
IN	8	16
KY	8	13
KY	9	5
LA	7	33
LA	8	14
MA	11	128

Type 2 – with dashboard



For report type 2 we are presenting the same report in the **dashboard format**. Hovering over each chart will provide details about the data points. In the bottom sunburst chart, you can click on different numbers, drill down into levels and get detailed information about each segment.

Web-based Dashboarding

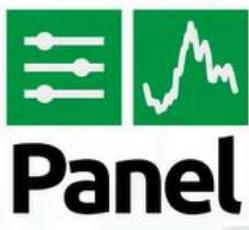
1.

Dash from Plotly



- **python framework for building web analytic applications.**
- **It is written on top of Flask, Plotly.js, and React.js.**
- **Dash is well-suited for building data visualization apps with highly custom user interfaces.**

2.



- works with visualizations from Bokeh, Matplotlib, HoloViews, and many other Python plotting libraries, making them instantly viewable either individually or when combined with interactive widgets that control them.
- works equally well in Jupyter Notebooks, for creating quick data-exploration tools. Panel can also be used in standalone deployed apps and dashboards, allowing you to easily switch between those contexts as needed.

3.



- turns Jupyter notebooks into standalone web applications.
- It can be used with separate layout tools like jupyter-flex or templates like voila-vuetify.

4.



- can easily turn data scripts into shareable web apps with 3 main principles:
- embrace python scripting, treat widgets as variables, and reuse data and computation.

Web-based Dashboarding

Dash from Plotly
 plotly | Dash



Dashboard Tools

1.



- is a plotting library, a widget and app library.
- It acts as a server for both plots and dashboards.
- Panel is one of the web-based dashboarding tools built on Bokeh.

2.



ipywidgets provides an array of Jupyter-compatible widgets and an interface supported by many Python libraries, but sharing as a dashboard requires a separate deployable server like Voila.

3.



Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

4.



Bowtie allows users to build dashboards in pure Python.

5.



Flask is a Python-backed web server that can be used to build arbitrary web sites, including those with Python plots that function as flask dashboards.

<https://pyviz.org/dashboarding/>

Dashboarding tools

Just about any Python library can be used to create a “static” PNG, SVG, HTML, or other output that can be pasted into a presentation, sent in an email, published as a figure in a paper, and so on. Many people also want or need to create “live” Python-backed applications or dashboards that a user can interact with to explore or analyze some data. Python offers several libraries for this purpose. The four main tools designed specifically for web-based dashboarding in Python are:

- [Dash](#) (from [Plotly](#)); see the [blog post](#)
- [Panel](#) (from [Anaconda](#)); see the [blog post](#)
- [Voila](#) (from [QuantStack](#)); see the [blog post](#); used with separate layout tools like [jupyter-flex](#) or templates like [voila-vuetify](#).
- [Streamlit](#); see the [blog post](#)

You can see comparisons of these tools in:

- [Streamlit vs Dash vs Voilà vs Panel — Battle of The Python Dashboarding Giants](#) 30 Mar 2021 Stephen Kilcommis. Comparing Streamlit, Dash, Voilà, and Panel for dashboarding. Links to more detailed explorations for each library individually.
- [Are Dashboards for Me?](#) 7 Jul 2020 Dan Lester. Overview of Python and R dashboard tools, including Voila, ipywidgets, binder, Shiny, Dash, Streamlit, Bokeh, and Panel.

There are also other tools that can be used for some aspects of dashboarding as well as many other tasks:

- [Bokeh](#) is a plotting library, a widget and app library, and a server for both plots and dashboards. [Panel](#) is built on Bokeh, providing a higher-level toolkit specifically focused on app and dashboard creation and supporting multiple plotting libraries (not just Bokeh).
- [ipywidgets](#) provides a wide array of Jupyter-compatible widgets and an interface supported by many Python libraries, but sharing as a dashboard requires a separate deployable server like [Voila](#).
- [matplotlib](#) supports many different backends, including several native GUI toolkit interfaces such as Qt that can be used for building arbitrarily complex native applications that can be used instead of a web-based dashboard like those above.
- [Bowtie](#) (from Jacques Kvam) allows users to build dashboards in pure Python.
- [flask](#) is a Python-backed web server that can be used to build arbitrary web sites, including those with Python plots that then function as [flask dashboards](#), but is not specifically set up to make dashboarding easier.

John Snow's data journalism: the cholera map that changed the world

John Snow's map of cholera outbreaks from nineteenth century London changed how we saw a disease - and gave data journalists a model of how to work today

[Interactive map](#)

[Download the data](#)

[More data journalism and data visualisations from the Guardian](#)



John Snow's cholera map of Soho. Click image to embiggen

How often does a map change the world? In 1854, one produced by Doctor John Snow, altered it forever.

In the world of the 1850s, cholera was believed to be spread by [miasma in the air](#), germs were not yet understood and the sudden and serious outbreak of cholera in London's Soho was a mystery.

So Snow did something data journalists often do now: he mapped the cases. The map essentially represented each death as a bar, and you can see them in the smaller image above.



It became apparent that the cases were clustered around the pump in Broad (now Broadwick) street.

There were some outliers though and Snow wrote that:

In some of the instance , where the deaths are scattered a little further from the rest on the map, the malady was probably contracted at a nearer point to the pump

One 59-year-old woman sent daily for water from the Broad street pump because she liked its taste.
Wrote Snow:

I was informed by this lady's son that she had not been in the neighbourhood of Broad Street for many months. A cart went from broad Street to West End every day and it was the custom to take out a large bottle of the water from the pump in Broad Street, as she preferred it. The water was taken on Thursday 31st August., and she drank of it in the evening, and also on Friday. She was seized with cholera on the evening of the latter day, and died on Saturday

At a local brewery, the workers were allowed all the beer they could drink - it was believed they didn't drink water at all. But it had its own water supply too and there were consequently fewer cases.

In nearby Poland street, a workhouse was surrounded by cases but appeared unaffected: this was because, again, it had its own water supply.

It turned out that the water for the pump was polluted by sewage from a nearby cesspit where a baby's nappy contaminated with cholera had been dumped. But he didn't just produce a map; it was one part of a detailed statistical analysis.

As the [Public Health Perspectives blog says](#), it changed how we see data visualisations, and how we see microbes. Snow was born 200 years ago this week and is the [subject of an exhibiton at the London School of Hygiene and Tropical Medicine](#).

But how would those deaths look for a data journalist today?

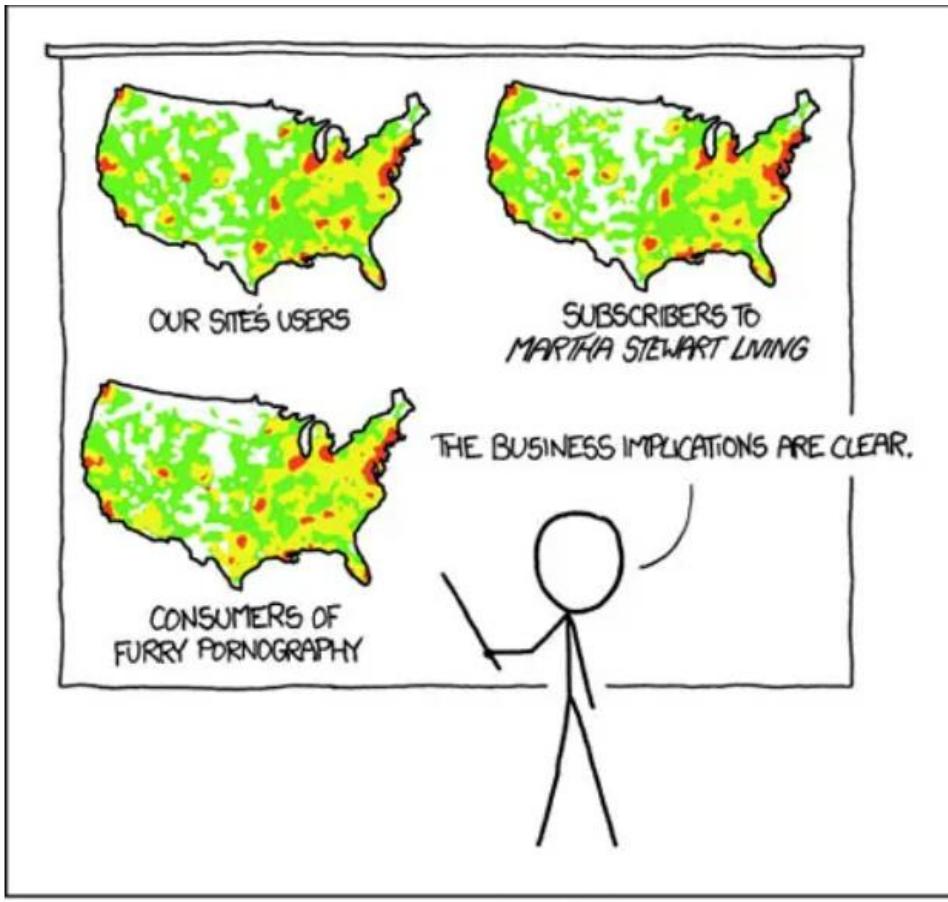
Thanks to [Robin Wilson at Southampton University, we have the data](#). Robin painstakingly georeferenced every cholera death and pump location, so we could recreate the map on a modern layout of London. We wondered what would happen if we tried to recreate the map using a modern tool, opting to try [CartoDB](#), using the the lovely [Stamen](#) 'toner' projection to at least keep the background in common with Snow's London.

An interactive version

 Water pumps  Cholera deaths

Cholera map key Photograph: Guardian

As XKCD have pointed out, heatmaps or dotmaps have flaws, not least that they tend to show where the people are.



PET PEEVE #208:
GEOGRAPHIC PROFILE MAPS WHICH ARE
BASICALLY JUST POPULATION MAPS

▣ XKCD on heatmaps. Image: XKCD

And the alternative is usually to aggregate the data, so that you could show, say, the incidence of cholera by geographical area - a choropleth. But in this case, would that have worked?

The cluster of dots around the Broad street pump were what alerted Snow to the cause of the outbreak.

Edward Tufte is interesting on this. He points out that

The big problem is that dot maps fail to take into account the number of people living in an area and at risk to get a disease ... Snow's dot map does not assess varying densities of population in the area around the pump

But, as Tufte points out, this part of Soho was incredibly thickly populated. And "aggregations by area can sometimes mask and even distort the true story of the data". A choropleth map of the area might show that there was a cluster of cholera cases, but it might not, depending on where the boundaries are drawn. Mark Monmonier, author of How to lie with maps has examined this.

But there's another key point here: in the event of an outbreak like this now, it's inconceivable that the government would publish the data on grounds of privacy; that the victims' addresses were personal data.

As data journalists, we agonise over how to represent the true impact of an event. [Maps](#) are often the first thing to reach for because it's easy: the tools are now just so easy to use and so much data is geographic. Although they are often mightily popular with readers, it's probably not always the right choice. Trying harder to show the data in different ways is an honourable objective.

But when they work, maps can tell a story in a language that everyone can understand.

Maybe Snow's map had such a huge impact on its own because it was simply a great data visualisation.

Robin Wilson has given us links to the data below. What can you do with it?

Introduction to Plotly

Plotly is an interactive, open source plotting library that supports over 40 unique chart types. It is available in Python, R and Javascript.

Plotly python is build on top of Plotly Javascript library and includes chart types like statistical, financial, maps, scientific, and 3-dimensional data . The web based visualizations created using Plotly python can be displayed in Jupyter notebook, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash.

The focus of this lesson will be on two of the Plotly sub-modules:

Plotly Graph Objects and Plotly Express.

Plotly Graph Objects is

- the low-level interface to figures, traces, and layout.
- The Plotly graph objects module provides an automatically generated hierarchy of classes (figures, traces, and layout) called graph objects.
- These graph objects are used for representing figures with a top-level class `plotly.graph_objects.Figure`.

Plotly express

- is a high-level wrapper for Plotly.
- It is a recommended starting point for creating most common figures provided by Plotly using a more simple syntax.
- It uses graph objects internally.

Let's see how to use `plotly.graph_objects` sub-module by creating a simple line chart.

First, import the required packages. Here we are importing graph objects as go. Then, generate sample data using numpy. The Plotly.graph contains a JSON object which has a dictionary structure. Since we imported `plotly graph object` as go in the previous slide, `go` will be the JSON object.

The Chart can be plotted by up-dating the values of the go object keywords. We will create the figure by adding a scatter type trace.

Next the layout of the figure is updated using the "update layout" method. Here, we are updating the x-axis, y-axis, and chart title. This is the plotted figure.

Now, we will create the same line chart using Plotly express. In Plotly express, the entire line chart can be created using a single command. Visualization is automatically interactive. Plotly express makes visualization easy to create and modify.

It's time to play with the plotly library. Next is going to be a lab session. We will be using the airline reporting dataset from data asset exchange to demonstrate how to use Plotly graph objects and Express for creating charts.

Here is a quick overview of the airline reporting dataset. The Reporting Carrier On-Time Performance Dataset contains information on approximately 200 million domestic US flights reported to the United States Bureau of Transportation Statistics.

The dataset contains basic information about each flight (such as date, time, departure airport, arrival airport) and, if applicable, the amount of time the flight was delayed and information about the reason for the delay.

Introduction to Plotly

Plotly – An Overview

- Interactive, open-source plotting library
- Supports over 40 unique chart types
- Includes chart types like statistical, financial, maps, scientific, and 3-dimensional
- Visualizations can be displayed in Jupyter notebook, saved to HTML files, or can be used in developing Python-built web applications

Plotly Sub-modules

- Plotly Graph Objects: Low-level interface to figures, traces, and layout
`plotly.graph_objects.Figure`
- Plotly Express: High-level wrapper

How to use `plotly.graph_objects` sub-module by creating a simple line chart?

1.

Using `plotly.graph_objects`

```
# Import required packages
import plotly.graph_objects as go
import plotly.express as px
import numpy as np

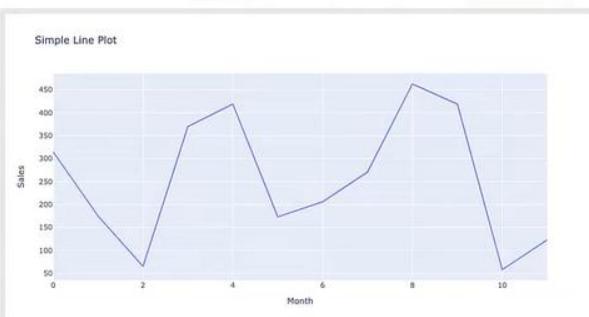
# Set random seed for reproducibility
np.random.seed(10)
x = np.arange(12)
# Create random y values
y = np.random.randint(50, 500, size=12)
```

2.

Using `plotly.graph_objects`

```
# Line Plot using graph objects.
# Plotly.graph contains a JSON object which has a structure of dict.
# Here, `go` is the plotly JSON object.
# Updating values of `go` object keywords, chart can be plotted.
# Create figure and add trace (scatter)

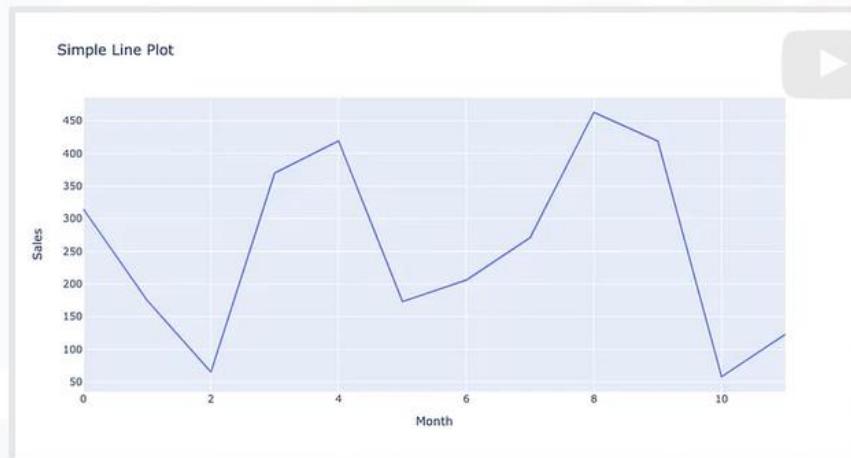
fig = go.Figure(data=go.Scatter(x=x, y=y))
fig.update_layout(title='Simple Line Plot', xaxis_title='Month', yaxis_title='Sales')
fig.show()
```



3.

Using plotly.express

```
# Entire line chart can be created in a single command  
fig = px.line(x=x, y=y, title='Simple Line Plot', labels=dict(x='Month', y='Sales'))  
fig.show()
```



In Plotly express, the entire line chart can be created using a single command. Visualization is automatically interactive. Plotly express makes visualization easy to create and modify.

Dataset for Lab

Data Asset eXchange

Explore useful and relevant data sets for enterprise data science

Dataset | CSV

Airline Reporting Carrier On-Time Performance Dataset

November 23, 2020 →

Dataset | CSV

COVID-19 Questions

October 1, 2020 →

Dataset for Lab

The screenshot shows the IBM Developer website with a sidebar on the left containing links for Artificial intelligence, Articles, Courses, Datasets, Models, Code Patterns, Podcasts, Open Projects, Tutorials, Videos, Community, and More resources. The main content area displays the 'Airline Reporting Carrier On-Time Performance Dataset'. It includes a summary, an 'Overview' section with a detailed description of the dataset, and a 'Dataset Metadata' table.

Field	Value
Format	CSV
License	CDLA-Sharing
Domain	Time Series
Number of Records	194,385,636 flights

Dataset for Lab

The screenshot shows the GDX Dataset Preview page for the same dataset. It features a navigation bar with links for GDX Dataset Preview, Notebook Preview, Run Notebook in Watson Studio, and Dataset Homepage. The main content area displays the dataset's metadata, including its name, a preview section, and a glossary.

Dataset Metadata	Dataset Preview	Dataset Glossary
Format	CSV	
License	CDLA-Sharing	
Domain	Time Series	
Number of Records	194,385,636 flights	
Data Split	NA	
Size	81 GB	
Data Origin	Bureau of Transportation Statistics	
Dataset Version	Version 1 – June 25, 2020	
Dataset Coverage	Location: United States Dates: 1987 through 2020	
Business Use Case	Aviation: Flight delay prediction	

The Reporting Carrier On-Time Performance Dataset contains information on approximately 200 million domestic US flights reported to the United States Bureau of Transportation Statistics. The dataset contains basic information about each flight (such as date, time, departure airport, arrival airport) and, if applicable, the amount of time the flight was delayed and information about the reason for the delay

Reading: Additional Resources for Plotly

 [Bookmark this page](#)

To learn more about using Plotly to create dashboards, explore

[Plotly Python](#)

[Plotly graph objects with example](#)

[Plotly express](#)

[API reference](#)

Here are additional useful resources:

[Plotly cheatsheet](#)

[Plotly community](#)

[Related blogs](#)

[Open-source datasets](#)

<https://plotly.com/python/getting-started/>

[Python > Getting Started with Plotly](#)

[Suggest an edit to this page](#)

Getting Started with Plotly in Python

Getting Started with Plotly for Python.

New to Plotly?

Overview

The [plotly Python library](#) is an interactive, [open-source](#) plotting library that supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases.

Built on top of the Plotly JavaScript library ([plotly.js](#)), plotly enables Python users to create beautiful interactive web-based visualizations that can be displayed in Jupyter notebooks, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash. The plotly Python library is sometimes referred to as "plotly.py" to differentiate it from the JavaScript library.

Thanks to deep integration with our [Kaleido](#) image export utility, plotly also provides great support for non-web contexts including desktop editors (e.g. QtConsole, Spyder, PyCharm) and static document publishing (e.g. exporting notebooks to PDF with high-quality vector images).

This Getting Started guide explains how to install plotly and related optional pages. Once you've installed, you can use our documentation in three main ways:

1. You jump right in to **examples** of how to make [basic charts](#), [statistical charts](#), [scientific charts](#), [financial charts](#), [maps](#), and [3-dimensional charts](#).
2. If you prefer to learn about the **fundamentals** of the library first, you can read about [the structure of figures](#), [how to create and update figures](#), [how to display figures](#), [how to theme figures with templates](#), [how to export figures to various formats](#) and about [Plotly Express](#), [the high-level API](#) for doing all of the above.
3. You can check out our exhaustive **reference** guides: the [Python API reference](#) or the [Figure Reference](#)

For information on using Python to build web applications containing plotly figures, see the [Dash User Guide](#). We also encourage you to join the [Plotly Community Forum](#) if you want help with anything related to plotly.

Installation

plotly may be installed using pip:

```
$ pip install plotly==5.6.0
```

or conda:

```
$ conda install -c plotly plotly=5.6.0
```

This package contains everything you need to write figures to standalone HTML files.

Note: **No internet connection, account, or payment is required to use plotly.py.** Prior to version 4, this library could operate in either an "online" or "offline" mode. The documentation tended to emphasize the online mode, where graphs get published to the Chart Studio web service. In version 4, all "online" functionality was removed from the plotly package and is

now available as the separate, optional, chart-studio package (See below). **plotly.py version 4** is "offline" only, and does not include any functionality for uploading figures or data to cloud services.

```
import plotly.express as px
fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])
fig.write_html('first_figure.html', auto_open=True)
```

Plotly charts in Dash

[Dash](#) is the best way to build analytical apps in Python using Plotly figures. To run the app below, run pip install dash, click "Download" to get the code and run python app.py.

Get started with [the official Dash docs](#) and learn how to effortlessly [style & deploy](#) apps like this with [Dash Enterprise](#).

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objects as go

app = dash.Dash(__name__)

app.layout = html.Div([
    html.P("Color:"),  

    dcc.Dropdown(
        id="dropdown",
        options=[
            {'label': x, 'value': x}
            for x in ['Gold', 'MediumTurquoise', 'LightCoral'],
        ],
        value='Gold',
        clearable=False,
    ),
    dcc.Graph(id="graph"),
])

@app.callback(
    Output("graph", "figure"),
    [Input("dropdown", "value")])
def display_color(color):
    fig = go.Figure(
        data=go.Bar(y=[2, 3, 1], marker_color=color))
    return fig
```



```
app.run_server(debug=True)
```

Color:

Gold



JupyterLab Support

For use in [JupyterLab](#), install the jupyterlab and ipywidgets packages using pip:

```
$ pip install "jupyterlab>=3" "ipywidgets>=7.6"
```

or conda:

```
$ conda install "jupyterlab>=3" "ipywidgets>=7.6"
```

You'll need jupyter-dash to add widgets such as sliders, dropdowns, and buttons to Plotly charts in JupyterLab.

Install [jupyter-dash](#) using pip:

```
$ pip install jupyter-dash
```

or conda:

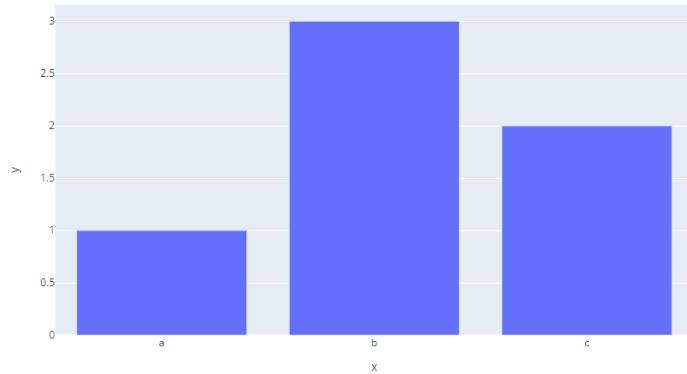
```
$ conda install -c conda-forge -c plotly jupyter-dash
```

These packages contain everything you need to run JupyterLab...

```
$ jupyter lab
```

and display plotly figures inline using the `plotly_mimetype` renderer...

```
import plotly.express as px
fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])
fig.show()
```



or using `FigureWidget` objects.

```
import plotly.express as px
fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])

import plotly.graph_objects as go
fig_widget = go.FigureWidget(fig)
fig_widget
```

The instructions above apply to JupyterLab 3.x. **For JupyterLab 2 or earlier**, run the following commands to install the required JupyterLab extensions (note that this will require [node](#) to be installed):

```
# JupyterLab 2.x renderer support jupyter labextension install jupyterlab-plotly@5.6.0 @jupyter-widgets/jupyterlab-manager
```

Please check out our [Troubleshooting guide](#) if you run into any problems with JupyterLab, particularly if you are using multiple python environments inside Jupyter.

See [Displaying Figures in Python](#) for more information on the renderers framework, and see [Plotly FigureWidget Overview](#) for more information on using `FigureWidget`.

Jupyter Notebook Support

For use in the classic [Jupyter Notebook](#), install the `notebook` and `ipywidgets` packages using pip:

```
$ pip install "notebook>=5.3" "ipywidgets>=7.5"
```

or conda:

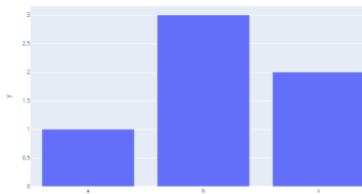
```
$ conda install "notebook>=5.3" "ipywidgets>=7.5"
```

These packages contain everything you need to run a Jupyter notebook...

```
$ jupyter notebook
```

and display plotly figures inline using the notebook renderer...

```
import plotly.express as px
fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])
fig.show()
```



```
import plotly.express as px
fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])

import plotly.graph_objects as go
fig_widget = go.FigureWidget(fig)
fig_widget
```

See [Displaying Figures in Python](#) for more information on the renderers framework, and see [Plotly FigureWidget Overview](#) for more information on using FigureWidget.

Static Image Export

plotly.py supports [static image export](#), using either the [kaleido](#) package (recommended, supported as of plotly version 4.9) or the [orca](#) command line utility (legacy as of plotly version 4.9).

Kaleido

The [kaleido](#) package has no dependencies and can be installed using pip...

```
$ pip install -U kaleido
```

or conda.

```
$ conda install -c plotly python-kaleido
```

Orca

While Kaleido is now the recommended image export approach because it is easier to install and more widely compatible, [static image export](#) can also be supported by the legacy [orca](#) command line utility and the [psutil](#) Python package.

These dependencies can both be installed using conda:

```
conda install -c plotly plotly-orca==1.3.1 psutil
```

Or, psutil can be installed using pip...

```
pip install psutil
```

and orca can be installed according to the instructions in the [orca README](#).

Extended Geo Support

Some plotly.py features rely on fairly large geographic shape files. The county choropleth figure factory is one such example. These shape files are distributed as a separate plotly-geo package. This package can be installed using pip...

```
$ pip install plotly-geo==1.0.0
```

or conda.

```
$ conda install -c plotly plotly-geo=1.0.0
```

See [USA County Choropleth Maps in Python](#) for more information on the county choropleth figure factory.

Chart Studio Support

The chart-studio package can be used to upload plotly figures to Plotly's Chart Studio Cloud or On-Prem services. This package can be installed using pip...

```
$ pip install chart-studio==1.1.0
```

or conda.

```
$ conda install -c plotly chart-studio=1.1.0
```

Note: This package is optional, and if it is not installed it is not possible for figures to be uploaded to the Chart Studio cloud service.

Where to next?

Once you've installed, you can use our documentation in three main ways:

1. You jump right in to **examples** of how to make [basic charts](#), [statistical charts](#), [scientific charts](#), [financial charts](#), [maps](#), and [3-dimensional charts](#).
2. If you prefer to learn about the **fundamentals** of the library first, you can read about [the structure of figures](#), [how to create and update figures](#), [how to display figures](#), [how to theme figures with templates](#), [how to export figures to various formats](#) and about [Plotly Express](#), [the high-level API](#) for doing all of the above.
3. You can check out our exhaustive **reference** guides: the [Python API reference](#) or the [Figure Reference](#)

For information on using Python to build web applications containing plotly figures, see the [Dash User Guide](#).

We also encourage you to join the [Plotly Community Forum](#) if you want help with anything related to plotly.

What About Dash?

[Dash](#) is an open-source framework for building analytical applications, with no Javascript required, and it is tightly integrated with the Plotly graphing library.

Learn about how to install Dash at <https://dash.plot.ly/installation>.

Everywhere in this page that you see fig.show(), you can display the same figure in a Dash application by passing it to the figure argument of the [Graph component](#) from the built-in dash_core_components package like this:

```
import plotly.graph_objects as go # or plotly.express as px
fig = go.Figure() # or any Plotly Express function e.g. px.bar(...)
# fig.add_trace( ... )
# fig.update_layout( ... )

import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()
app.layout = html.Div([
    dcc.Graph(figure=fig)
])

app.run_server(debug=True, use_reloader=False) # Turn off reloader if inside Jupyter
```

<https://plotly.com/python/graph-objects/>

Graph Objects in Python

Python classes that represent parts of a figure.

New to Plotly?

What Are Graph Objects?

The figures created, manipulated and rendered by the plotly Python library are [represented by tree-like data structures](#) which are automatically serialized to JSON for rendering by the Plotly.js JavaScript library. These trees are composed of named nodes called "attributes", with their structure defined by the Plotly.js figure schema, which is available in [machine-readable form](#). The **plotly.graph_objects module (typically imported as go)** contains an [automatically-generated hierarchy of Python classes](#) which represent non-leaf nodes in this figure schema. The term "graph objects" refers to instances of these classes.

The primary classes defined in the plotly.graph_objects module are [Figure](#) and an [ipywidgets-compatible variant called FigureWidget](#), which both represent entire figures. Instances of these classes have many convenience methods for Pythonically [manipulating their attributes](#) (e.g. `.update_layout()` or `.add_trace()`), which all accept ["magic underscore" notation](#) as well as [rendering them](#) (e.g. `.show()`) and [exporting them to various formats](#) (e.g. `.to_json()` or `.write_image()` or `.write_html()`).

Note: the functions in [Plotly Express](#), which is the recommended entry-point into the plotly library, are all built on top of graph objects, and all return instances of `plotly.graph_objects.Figure`.

Every non-leaf attribute of a figure is represented by an instance of a class in the `plotly.graph_objects` hierarchy. For example, a figure `fig` can have an attribute `layout.margin`, which contains attributes `t`, `l`, `b` and `r` which are leaves of the tree: they have no children. The field at `fig.layout` is an object of class [plotly.graph_objects.Layout](#) and `fig.layout.margin` is an object of class `plotly.graph_objects.layout.Margin` which represents the margin node, and it has fields `t`, `l`, `b` and `r`, containing the values of the respective leaf-nodes. Note that specifying all of these values can be done without creating intermediate objects using ["magic underscore" notation](#): `go.Figure(layout_margin=dict(t=10, b=10, r=10, l=10))`.

The objects contained in the list which is the [value of the attribute data are called "traces"](#), and can be of one of more than 40 possible types, each of which has a corresponding class in `plotly.graph_objects`. For example, traces of type `scatter` are represented by instances of the class `plotly.graph_objects.Scatter`. This means that a figure constructed as `go.Figure(data=[go.Scatter(x=[1,2], y=[3,4])])` will have the JSON representation `{"data": [{"type": "scatter", "x": [1,2], "y": [3,4]}]}`.

Graph Objects Compared to Dictionaries

Graph objects have several benefits compared to plain Python dictionaries:

1. Graph objects provide precise data validation. If you provide an invalid property name or an invalid property value as the key to a graph object, an exception will be raised with a helpful error message describing the problem. This is not the case if you use plain Python dictionaries and lists to build your figures.
2. Graph objects contain descriptions of each valid property as Python docstrings, with a [full API reference available](#). You can use these docstrings in the development environment of your choice to learn about the available properties as an alternative to consulting the online [Full Reference](#).
3. Properties of graph objects can be accessed using both dictionary-style key lookup (e.g. `fig["layout"]`) or class-style property access (e.g. `fig.layout`).
4. Graph objects support higher-level convenience functions for making updates to already constructed figures (`.update_layout()`, `.add_trace()` etc).
5. Graph object constructors and update methods accept "magic underscores" (e.g. `go.Figure(layout_title_text="The Title")` rather than `dict(layout=dict(title=dict(text="The Title")))`) for more compact code.
6. Graph objects support attached rendering (`.show()`) and exporting functions (`.write_image()`) that automatically invoke the appropriate functions from [the plotly.io module](#).

When to use Graph Objects vs Plotly Express

The recommended way to create figures is using the [functions in the plotly.express module, collectively known as Plotly Express](#), which all return instances of `plotly.graph_objects.Figure`, so every figure produced with the `plotly` library actually uses graph objects under the hood, unless manually constructed out of dictionaries.

That said, certain kinds of figures are not yet possible to create with Plotly Express, such as figures that use certain 3D trace-types like [mesh](#) or [isosurface](#). In addition, certain figures are cumbersome to create by starting from a figure created with Plotly Express, for example figures with [subplots of different types](#), [dual-axis plots](#), or [faceted plots](#) with multiple different types of traces. To construct such figures, it can be easier to start from an empty `plotly.graph_objects.Figure` object (or one configured with subplots via the [make_subplots\(\) function](#)) and progressively add traces and update attributes as above.

Every `plotly` documentation page lists the Plotly Express option at the top if a Plotly Express function exists to make the kind of chart in question, and then the graph objects version below.

Note that the figures produced by Plotly Express **in a single function-call** are [easy to customize at creation-time](#), and to [manipulate after creation](#) using the `update_*` and `add_*` methods.

Comparing Graph Objects and Plotly Express

The figures produced by Plotly Express can always be built from the ground up using graph objects, but this approach typically takes **5-100 lines of code rather than 1**.

Here is a simple example of how to produce the same figure object from the same data, once with Plotly Express and once without. The data in this example is in "long form" but [Plotly Express also accepts data in "wide form"](#) and the line-count savings from Plotly Express over graph objects are comparable. More complex figures such as [sunbursts](#), [parallel coordinates](#), [facet plots](#) or [animations](#) require many more lines of figure-specific graph objects code, whereas switching from one representation to another with Plotly Express usually involves changing just a few characters.

```
import pandas as pd

df = pd.DataFrame({
    "Fruit": ["Apples", "Oranges", "Bananas", "Apples", "Oranges", "Bananas"],
    "Contestant": ["Alex", "Alex", "Alex", "Jordan", "Jordan", "Jordan"],
    "Number Eaten": [2, 1, 3, 1, 3, 2],
})

# Plotly Express

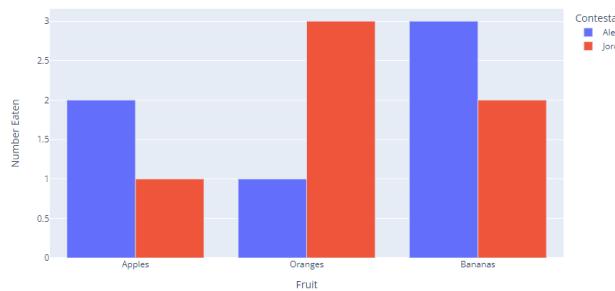
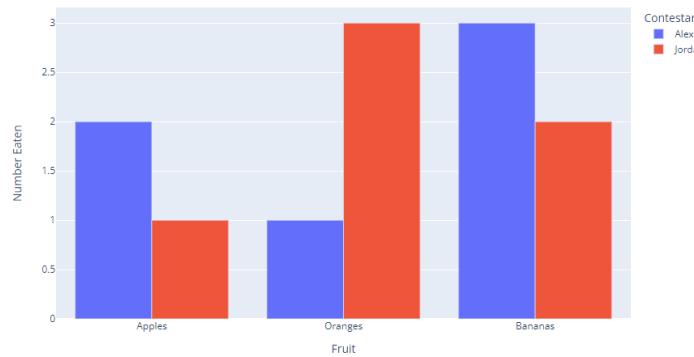
import plotly.express as px

fig = px.bar(df, x="Fruit", y="Number Eaten", color="Contestant", barmode="group")
fig.show()

# Graph Objects

import plotly.graph_objects as go

fig = go.Figure()
for contestant, group in df.groupby("Contestant"):
    fig.add_trace(go.Bar(x=group["Fruit"], y=group["Number Eaten"], name=contestant,
        hovertemplate="Contestant=%s<br>Fruit=%{x}<br>Number Eaten=%{y}<extra></extra>% contestant"))
fig.update_layout(legend_title_text = "Contestant")
fig.update_xaxes(title_text="Fruit")
fig.update_yaxes(title_text="Number Eaten")
fig.show()
```



What About Dash?

[Dash](#) is an open-source framework for building analytical applications, with no Javascript required, and it is tightly integrated with the Plotly graphing library.

Learn about how to install Dash at <https://dash.plot.ly/installation>.

Everywhere in this page that you see `fig.show()`, you can display the same figure in a Dash application by passing it to the `figure` argument of the [Graph component](#) from the built-in `dash_core_components` package like this:

```
import plotly.graph_objects as go # or plotly.express as px
fig = go.Figure() # or any Plotly Express function e.g. px.bar(...)
# fig.add_trace( ... )
# fig.update_layout( ... )

import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()
app.layout = html.Div([
    dcc.Graph(figure=fig)
])

app.run_server(debug=True, use_reloader=False) # Turn off reloader if inside Jupyter
```

<https://plotly.com/python/plotly-express/>

Plotly Express in Python

Plotly Express is a terse, consistent, high-level API for creating figures.

New to Plotly?

Overview

The `plotly.express` module (usually imported as `px`) contains functions that can create entire figures at once, and is referred to as Plotly Express or PX. Plotly Express is a built-in part of the `plotly` library, and is the recommended starting point for creating most common figures. Every Plotly Express function uses [graph objects](#) internally and returns a `plotly.graph_objects.Figure` instance. Throughout the `plotly` documentation, you will find the Plotly Express way of building figures at the top of any applicable page, followed by a section on how to use graph objects to build similar figures. Any figure created in a single function call with Plotly Express could be created using graph objects alone, but with between 5 and 100 times more code.

Plotly Express provides [more than 30 functions for creating different types of figures](#). The API for these functions was carefully designed to be as consistent and easy to learn as possible, making it easy to switch from a scatter plot to a bar chart to a histogram to a sunburst chart throughout a data exploration session. *Scroll down for a gallery of Plotly Express plots, each made in a single function call.*

Here is a talk from the [SciPy 2021 conference](#) that gives a good introduction to Plotly Express and [Dash](#):



Plotly Express currently includes the following functions:

- **Basics:** [scatter](#), [line](#), [area](#), [bar](#), [funnel](#), [timeline](#)
- **Part-of-Whole:** [pie](#), [sunburst](#), [treemap](#), [icicle](#), [funnel_area](#)
- **1D Distributions:** [histogram](#), [box](#), [violin](#), [strip](#), [ecdf](#)

- **2D Distributions:** [density heatmap](#), [density contour](#)
- **Matrix or Image Input:** [imshow](#)
- **3-Dimensional:** [scatter 3d](#), [line 3d](#)
- **Multidimensional:** [scatter matrix](#), [parallel coordinates](#), [parallel categories](#)
- **Tile Maps:** [scatter mapbox](#), [line mapbox](#), [choropleth mapbox](#), [density mapbox](#)
- **Outline Maps:** [scatter geo](#), [line geo](#), [choropleth](#)
- **Polar Charts:** [scatter polar](#), [line polar](#), [bar polar](#)
- **Ternary Charts:** [scatter ternary](#), [line ternary](#)

High-Level Features

The Plotly Express API in general offers the following features:

- **A single entry point into plotly:** just import `plotly.express` as `px` and get access to [all the plotting functions](#), plus [built-in demo datasets under px.data](#) and [built-in color scales and sequences under px.color](#). Every PX function returns a `plotly.graph_objects.Figure` object, so you can edit it using all the same methods like [update_layout](#) and [add_trace](#).
- **Sensible, Overridable Defaults:** PX functions will infer sensible defaults wherever possible, and will always let you override them.
- **Flexible Input Formats:** PX functions [accept input in a variety of formats](#), from lists and dicts to [long-form or wide-form Pandas DataFrames](#) to [numpy arrays and xarrays](#) to [GeoPandas GeoDataFrames](#).
- **Automatic Trace and Layout configuration:** PX functions will create one [trace](#) per animation frame for each unique combination of data values mapped to discrete color, symbol, line-dash, facet-row and/or facet-column. Traces' [legendgroup](#) and [showlegend attributes](#) are set such that only one legend item appears per unique combination of discrete color, symbol and/or line-dash. Traces are automatically linked to a correctly-configured [subplot of the appropriate type](#).
- **Automatic Figure Labelling:** PX functions [label axes, legends and colorbars](#) based in the input DataFrame or xarray, and provide [extra control with the labels argument](#).
- **Automatic Hover Labels:** PX functions populate the hover-label using the labels mentioned above, and provide [extra control with the hover_name and hover_data arguments](#).
- **Styling Control:** PX functions [read styling information from the default figure template](#), and support commonly-needed [cosmetic controls like category_orders and color_discrete_map](#) to precisely control categorical variables.
- **Uniform Color Handling:** PX functions automatically switch between [continuous](#) and [categorical color](#) based on the input type.

- **Faceting:** the 2D-cartesian plotting functions support [row, column and wrapped facetting with facet_row, facet_col and facet_col_wrap arguments](#).
- **Marginal Plots:** the 2D-cartesian plotting functions support [marginal distribution plots](#) with the marginal, marginal_x and marginal_y arguments.
- **A Pandas backend:** the 2D-cartesian plotting functions are available as [a Pandas plotting backend](#) so you can call them via df.plot().
- **Trendlines:** px.scatter supports [built-in trendlines with accessible model output](#).
- **Animations:** many PX functions support [simple animation support via the animation_frame and animation_group arguments](#).
- **Automatic WebGL switching:** for sufficiently large scatter plots, PX will automatically [use WebGL for hardware-accelerated rendering](#).

Plotly Express in Dash

[Dash](#) is the best way to build analytical apps in Python using Plotly figures. To run the app below, run pip install dash, click "Download" to get the code and run python app.py.

Get started with [the official Dash docs](#) and learn how to effortlessly [style & deploy](#) apps like this with [Dash Enterprise](#).

```

import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.express as px

df = px.data.iris()
all_dims = ['sepal_length', 'sepal_width',
            'petal_length', 'petal_width']

app = dash.Dash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="dropdown",
        options=[{"label": x, "value": x}
                 for x in all_dims],
        value=all_dims[:2],
        multi=True
    ),
    dcc.Graph(id="splom"),
])

```

`@app.callback(`

```

    Output("splom", "figure"),
    [Input("dropdown", "value")])
def update_bar_chart(dims):
    fig = px.scatter_matrix(
        df, dimensions=dims, color="species")
    return fig

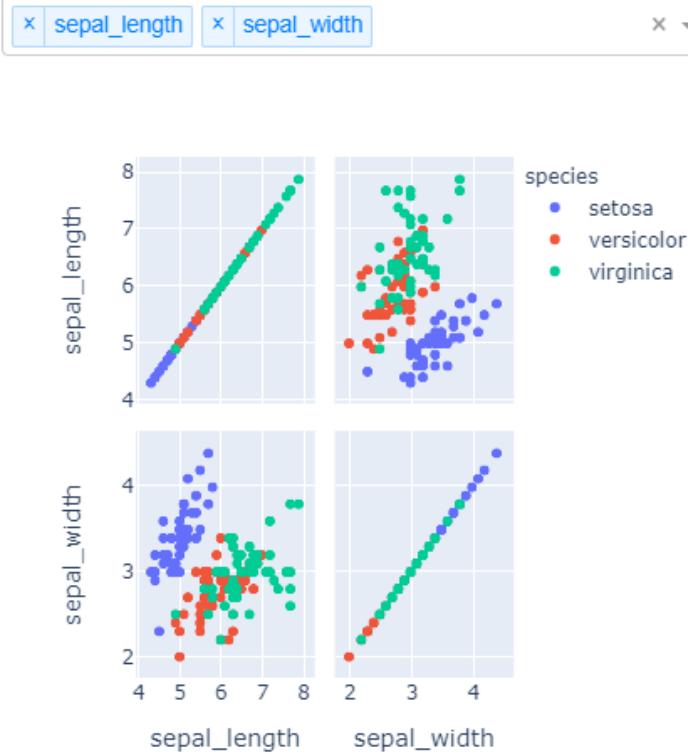
```

```

app.run_server(debug=True)

```

DOWNLOAD



Gallery

The following set of figures is just a sampling of what can be done with Plotly Express.

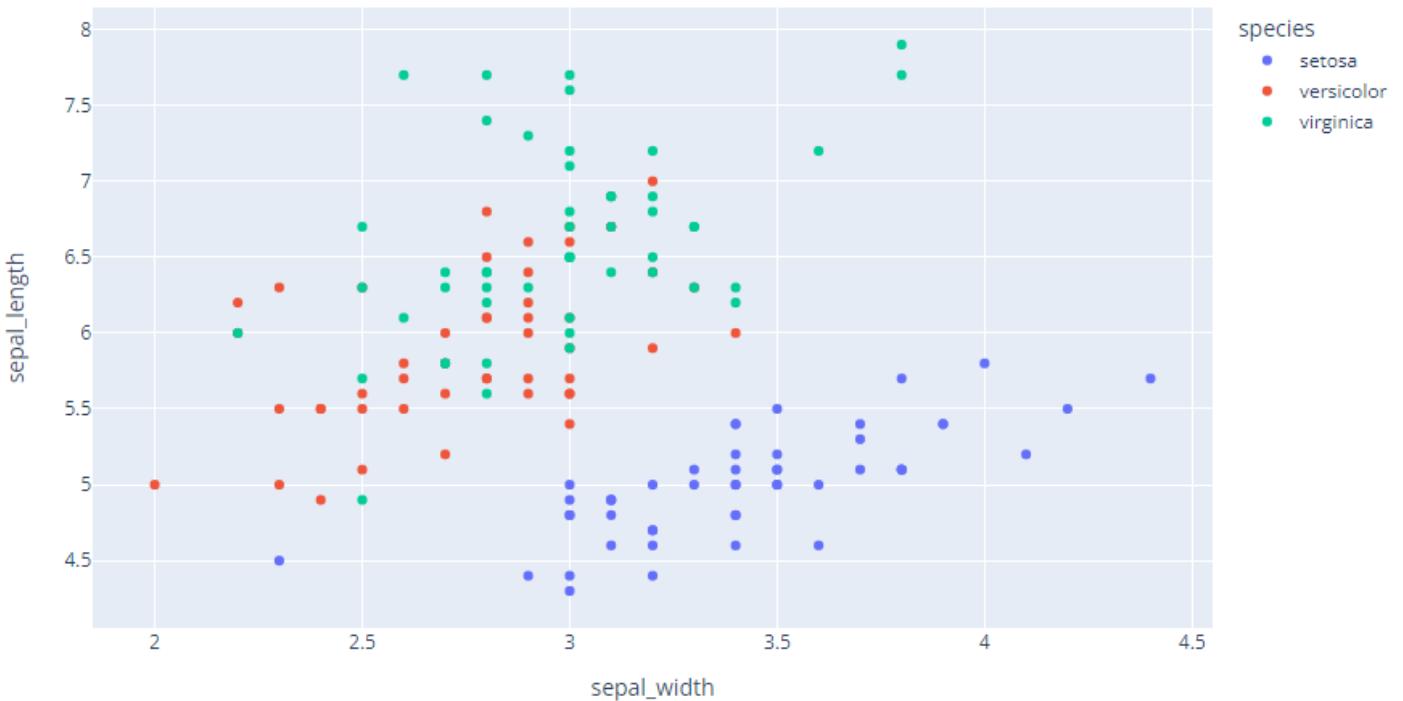
Scatter, Line, Area and Bar Charts

Read more about [scatter plots](#) and [discrete color](#).

```

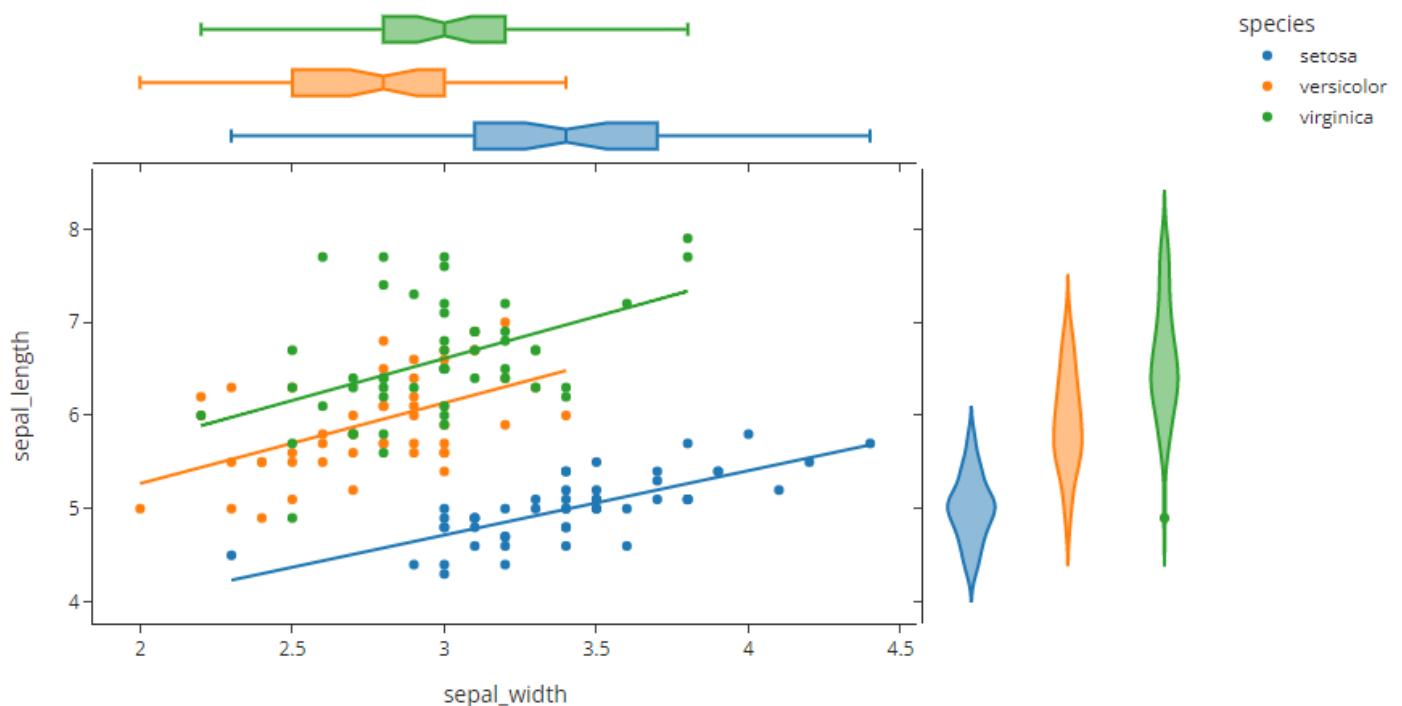
import plotly.express as px
df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species")
fig.show()

```



Read more about [trendlines](#) and [templates](#) and [marginal distribution plots](#).

```
import plotly.express as px
df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species", marginal_y="violin",
                 marginal_x="box", trendline="ols", template="simple_white")
fig.show()
```



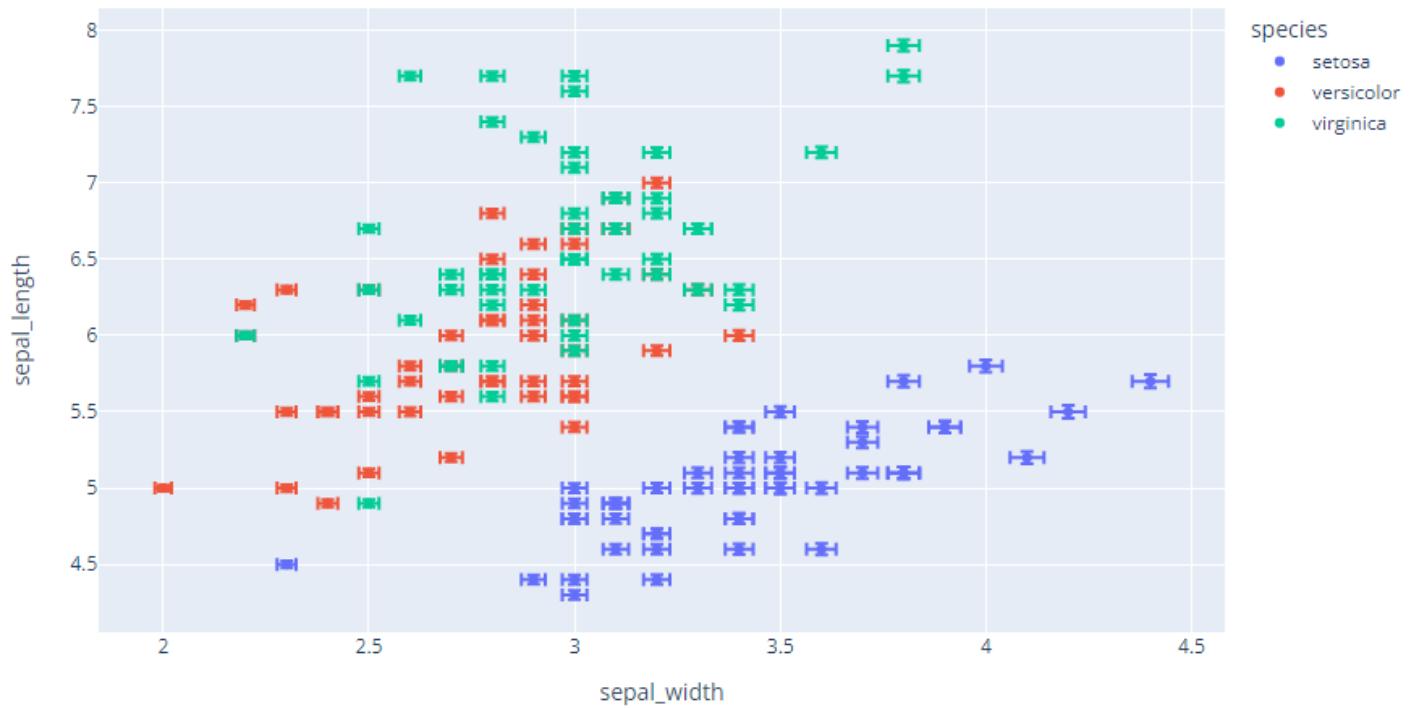
[Read more about error bars.](#)

```
import plotly.express as px
df = px.data.iris()
df["e"] = df["sepal_width"]/100
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species", error_x="e", error_y="e")
fig.show()
```

```

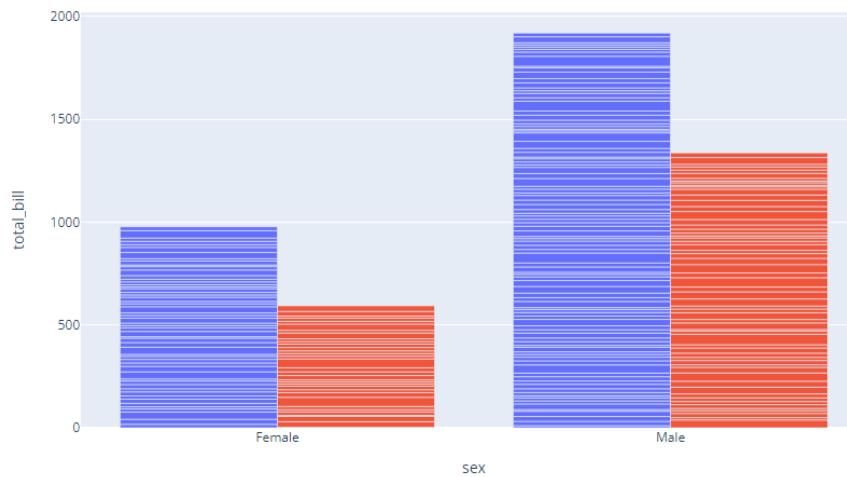
import plotly.express as px
df = px.data.iris()
df["e"] = df["sepal_width"]/100
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species", error_x="e", error_y="e")
fig.show()

```



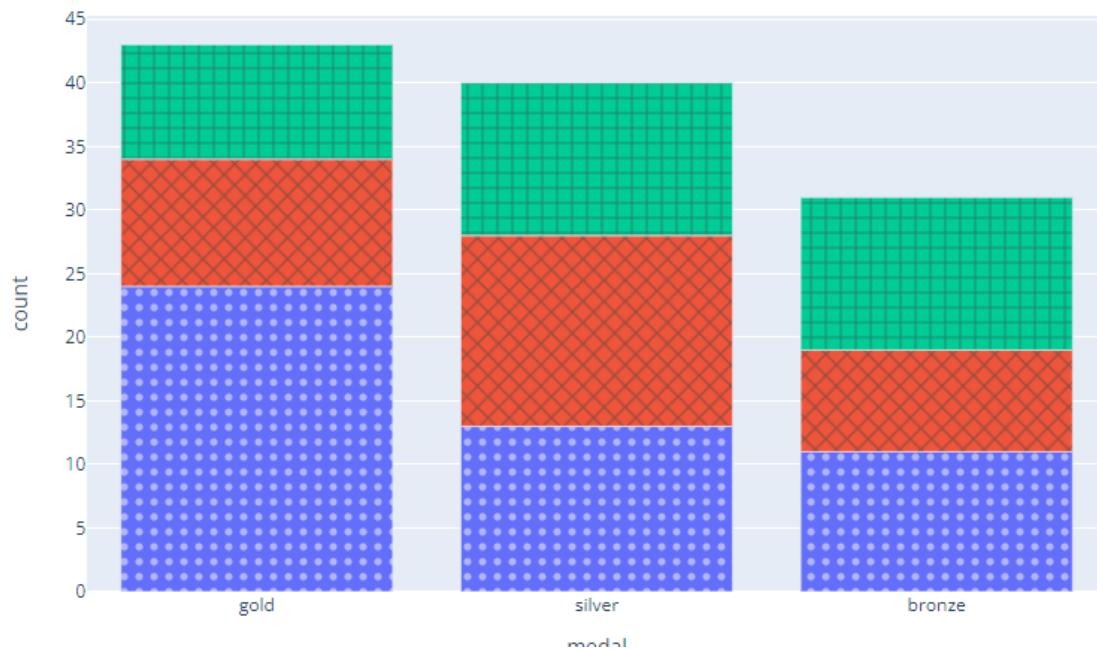
Read more about [bar charts](#).

```
import plotly.express as px
df = px.data.tips()
fig = px.bar(df, x="sex", y="total_bill", color="smoker", barmode="group")
fig.show()
```



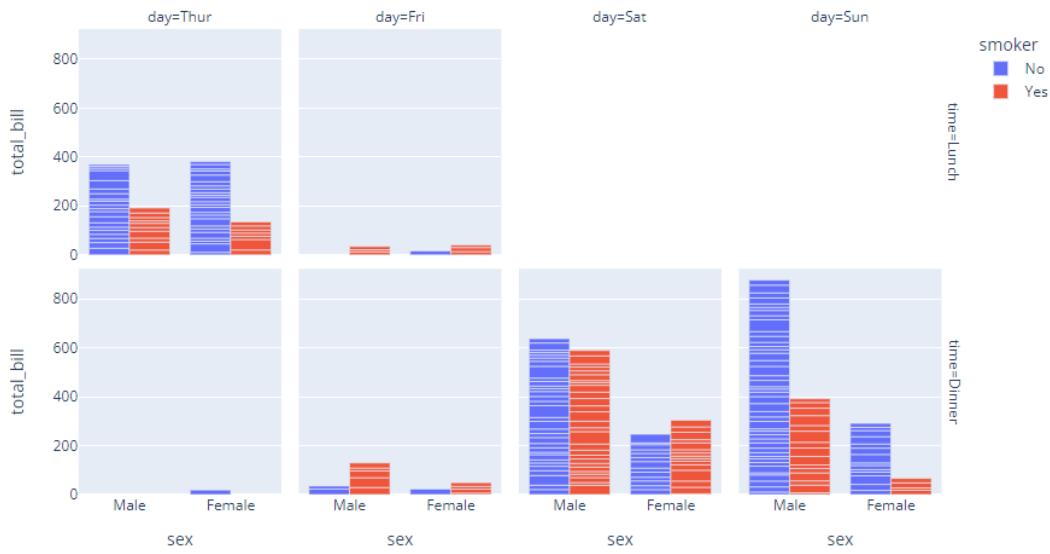
```
import plotly.express as px
df = px.data.medals_long()

fig = px.bar(df, x="medal", y="count", color="nation",
             pattern_shape="nation", pattern_shape_sequence=[".", "x", "+"])
fig.show()
```



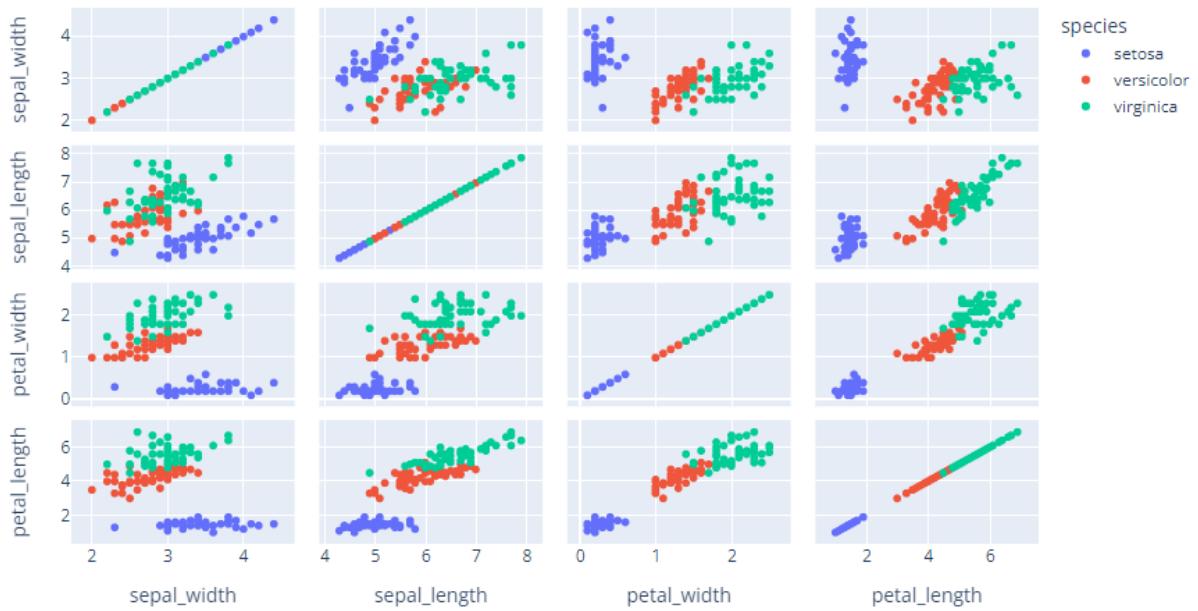
[Read more about facet plots.](#)

```
import plotly.express as px
df = px.data.tips()
fig = px.bar(df, x="sex", y="total_bill", color="smoker", barmode="group", facet_row="time", facet_col="day",
             category_orders={"day": ["Thur", "Fri", "Sat", "Sun"], "time": ["Lunch", "Dinner"]})
fig.show()
```



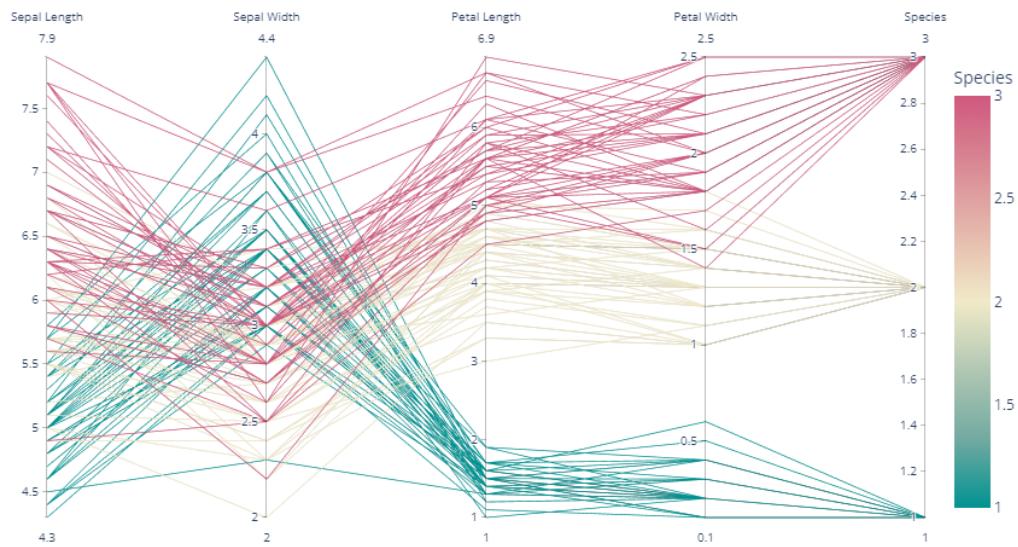
[Read more about scatterplot matrices \(SPLOMs\).](#)

```
import plotly.express as px
df = px.data.iris()
fig = px.scatter_matrix(df, dimensions=["sepal_width", "sepal_length", "petal_width", "petal_length"], color="species")
fig.show()
```

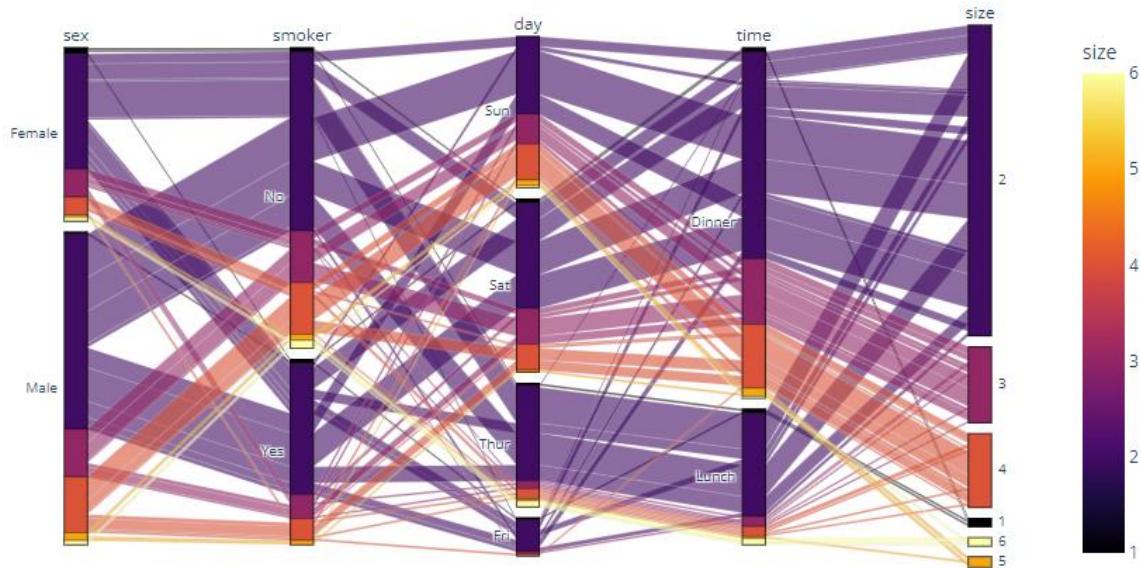


Read more about [parallel coordinates](#) and [parallel categories](#), as well as [continuous color](#).

```
import plotly.express as px
df = px.data.iris()
fig = px.parallel_coordinates(df, color="species_id", labels={"species_id": "Species",
    "sepal_width": "Sepal Width", "sepal_length": "Sepal Length",
    "petal_width": "Petal Width", "petal_length": "Petal Length", },
    color_continuous_scale=px.colors.diverging.Tealrose, color_continuous_midpoint=2)
fig.show()
```

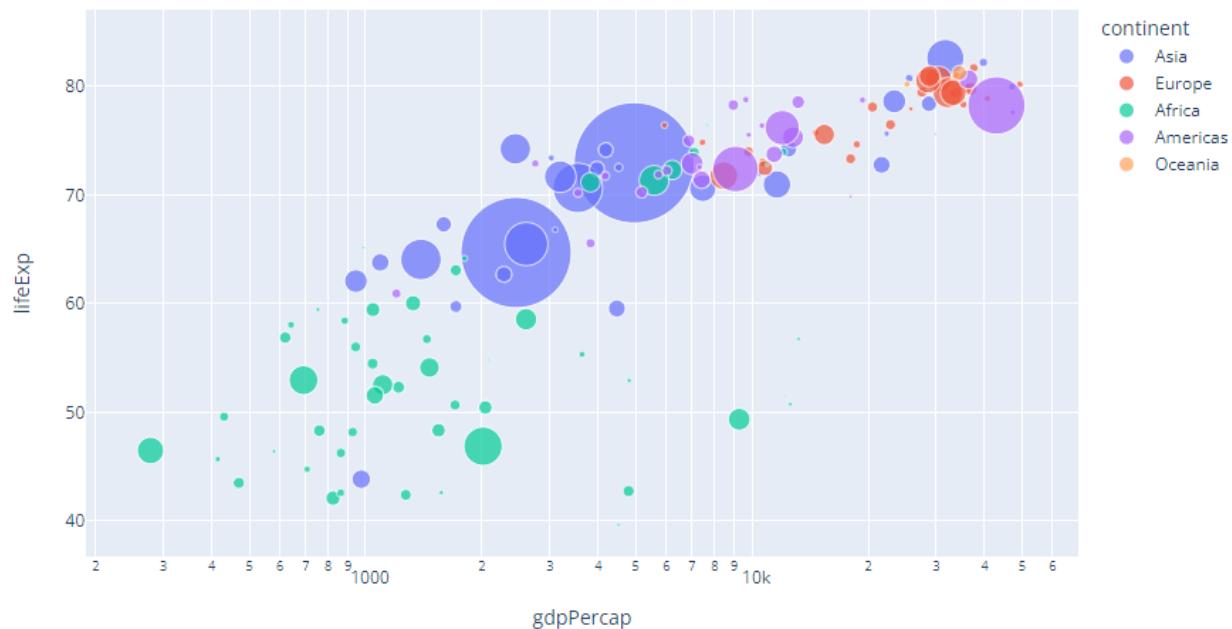


```
import plotly.express as px
df = px.data.tips()
fig = px.parallel_categories(df, color="size", color_continuous_scale=px.colors.sequential.Inferno)
fig.show()
```



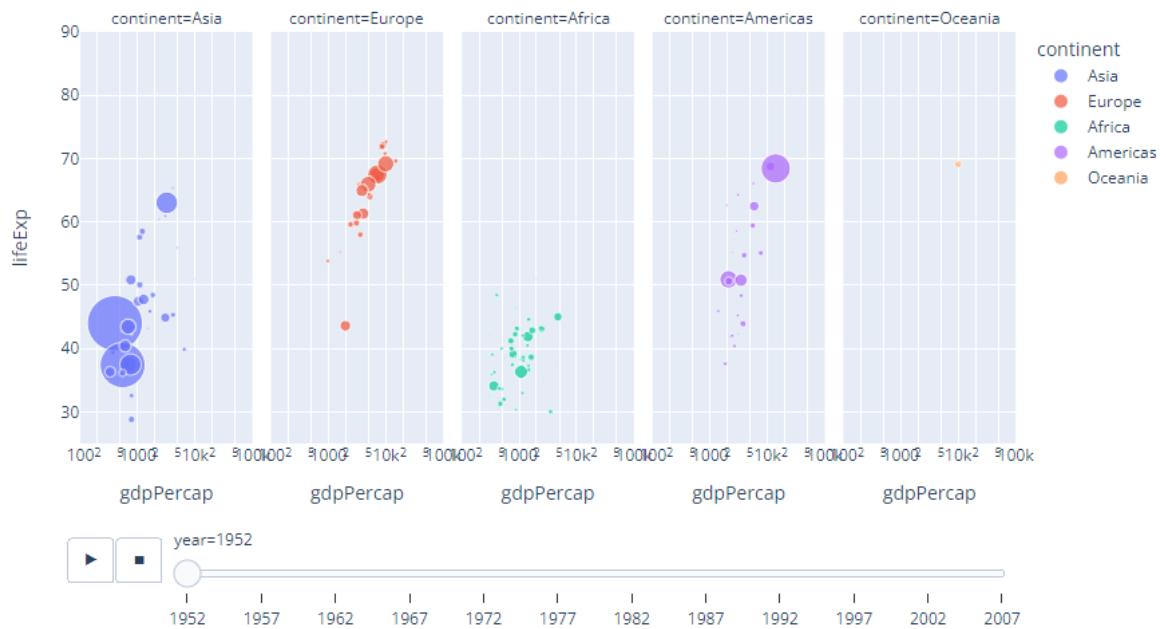
[Read more about hover labels.](#)

```
import plotly.express as px
df = px.data.gapminder()
fig = px.scatter(df.query("year==2007"), x="gdpPercap", y="lifeExp", size="pop", color="continent",
                 hover_name="country", log_x=True, size_max=60)
fig.show()
```



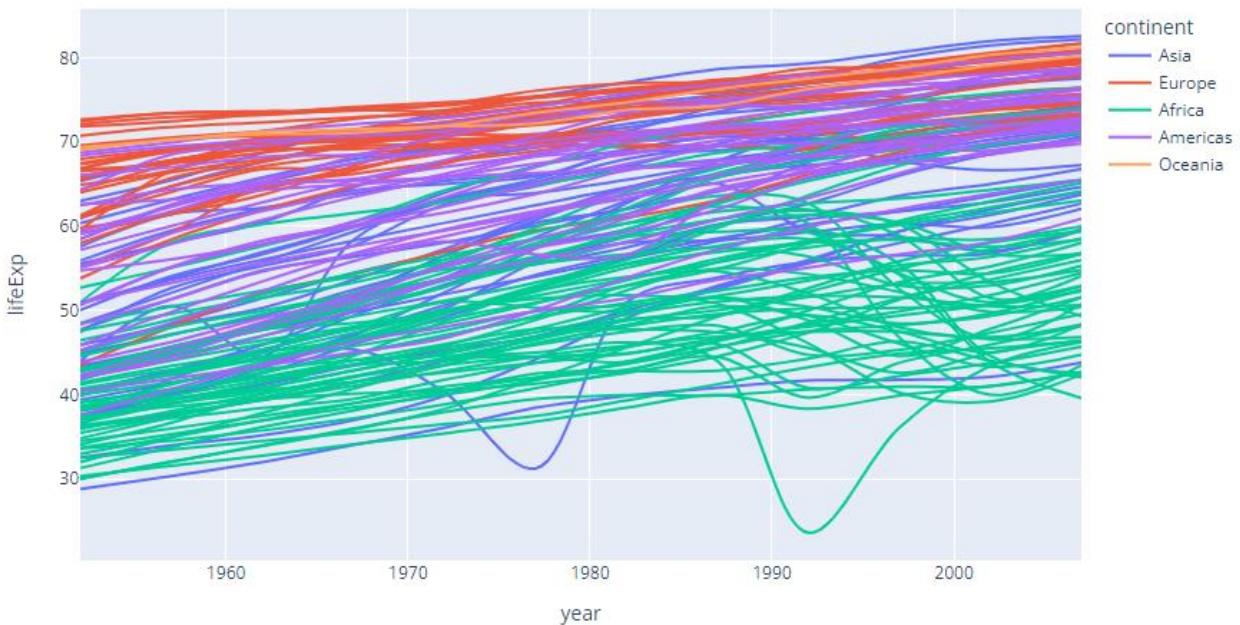
[Read more about animations.](#)

```
import plotly.express as px
df = px.data.gapminder()
fig = px.scatter(df, x="gdpPercap", y="lifeExp", animation_frame="year", animation_group="country",
                 size="pop", color="continent", hover_name="country", facet_col="continent",
                 log_x=True, size_max=45, range_x=[100,100000], range_y=[25,90])
fig.show()
```



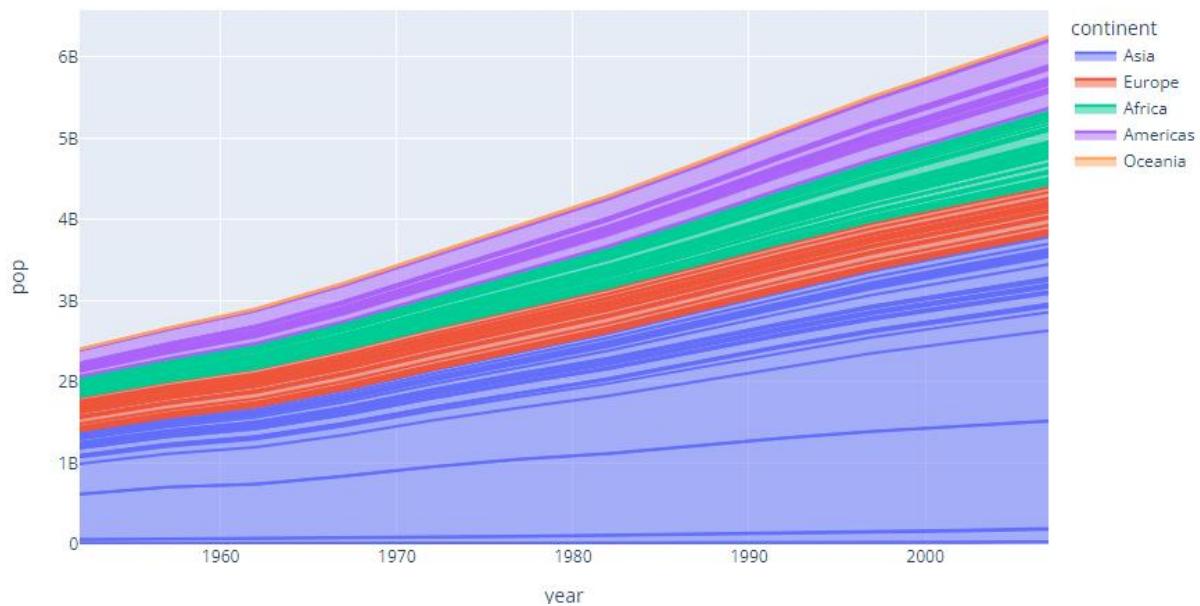
[Read more about line charts.](#)

```
import plotly.express as px
df = px.data.gapminder()
fig = px.line(df, x="year", y="lifeExp", color="continent", line_group="country", hover_name="country",
              line_shape="spline", render_mode="svg")
fig.show()
```



[Read more about area charts.](#)

```
import plotly.express as px
df = px.data.gapminder()
fig = px.area(df, x="year", y="pop", color="continent", line_group="country")
fig.show()
```

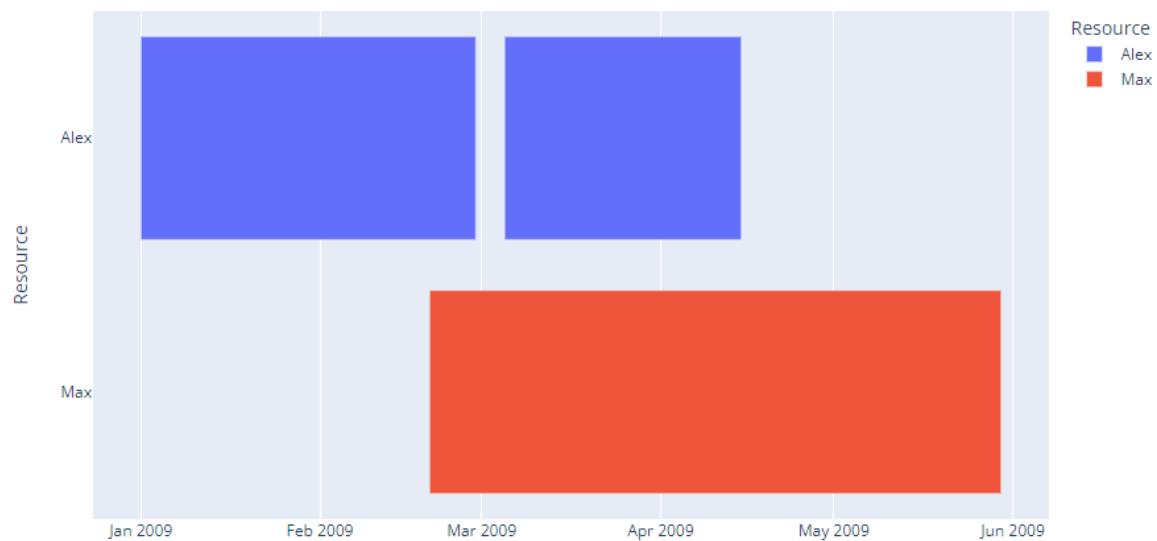


[Read more about timeline/Gantt charts.](#)

```
import plotly.express as px
import pandas as pd

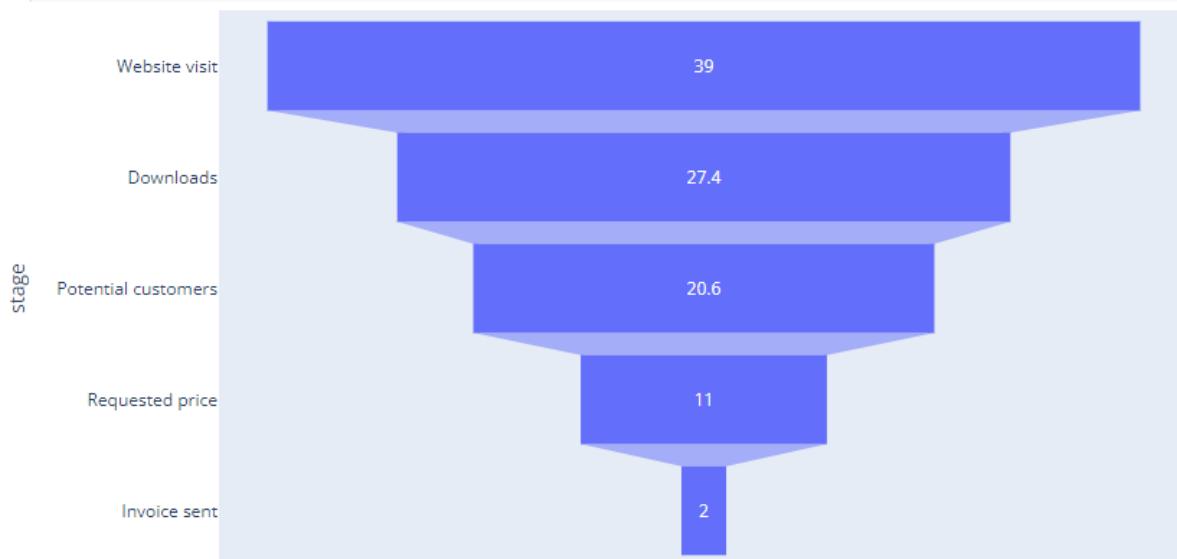
df = pd.DataFrame([
    dict(Task="Job A", Start='2009-01-01', Finish='2009-02-28', Resource="Alex"),
    dict(Task="Job B", Start='2009-03-05', Finish='2009-04-15', Resource="Alex"),
    dict(Task="Job C", Start='2009-02-20', Finish='2009-05-30', Resource="Max")
])

fig = px.timeline(df, x_start="Start", x_end="Finish", y="Resource", color="Resource")
fig.show()
```



[Read more about funnel charts.](#)

```
import plotly.express as px
data = dict(
    number=[39, 27.4, 20.6, 11, 2],
    stage=["Website visit", "Downloads", "Potential customers", "Requested price", "Invoice sent"])
fig = px.funnel(data, x='number', y='stage')
fig.show()
```

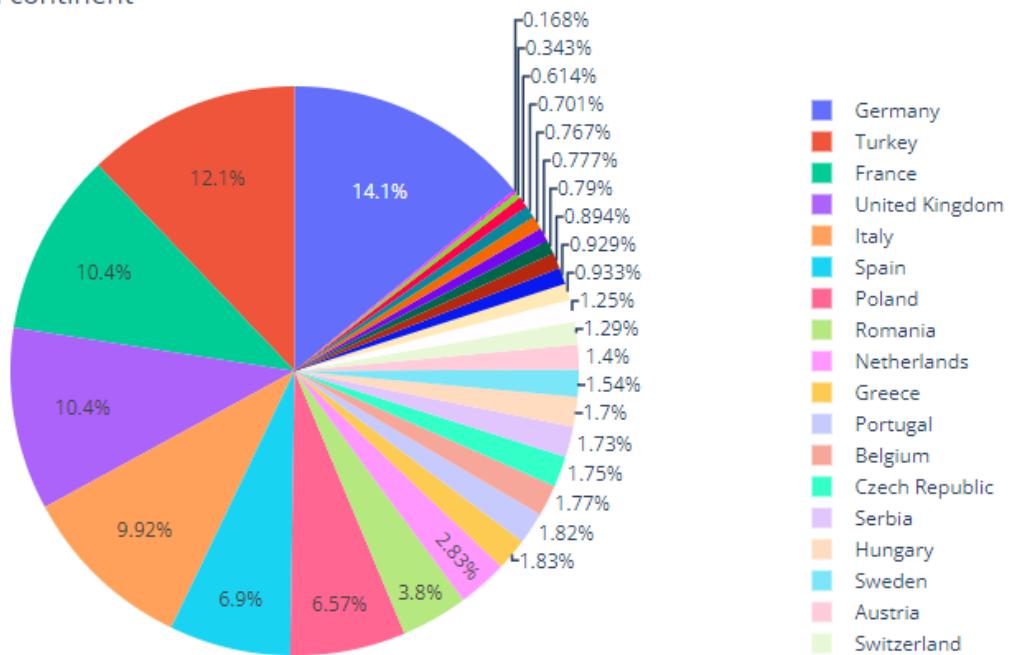


Part to Whole Charts

[Read more about pie charts.](#)

```
import plotly.express as px
df = px.data.gapminder().query("year == 2007").query("continent == 'Europe'")
df.loc[df['pop'] < 2.e6, 'country'] = 'Other countries' # Represent only Large countries
fig = px.pie(df, values='pop', names='country', title='Population of European continent')
fig.show()
```

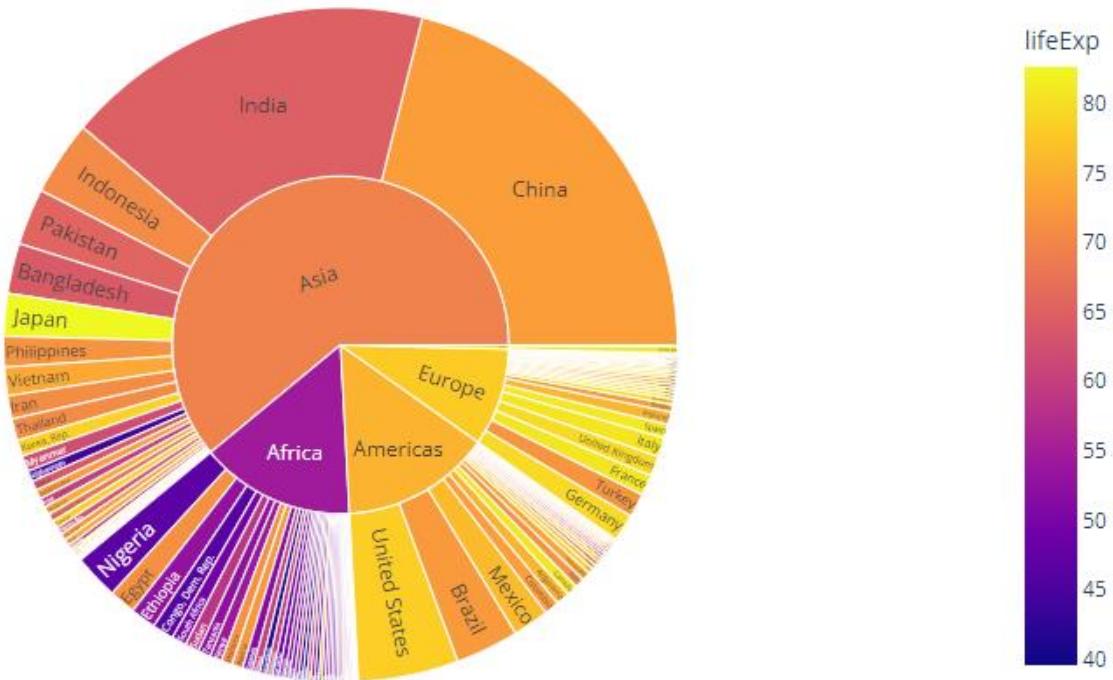
Population of European continent



Read more about [sunburst charts](#).

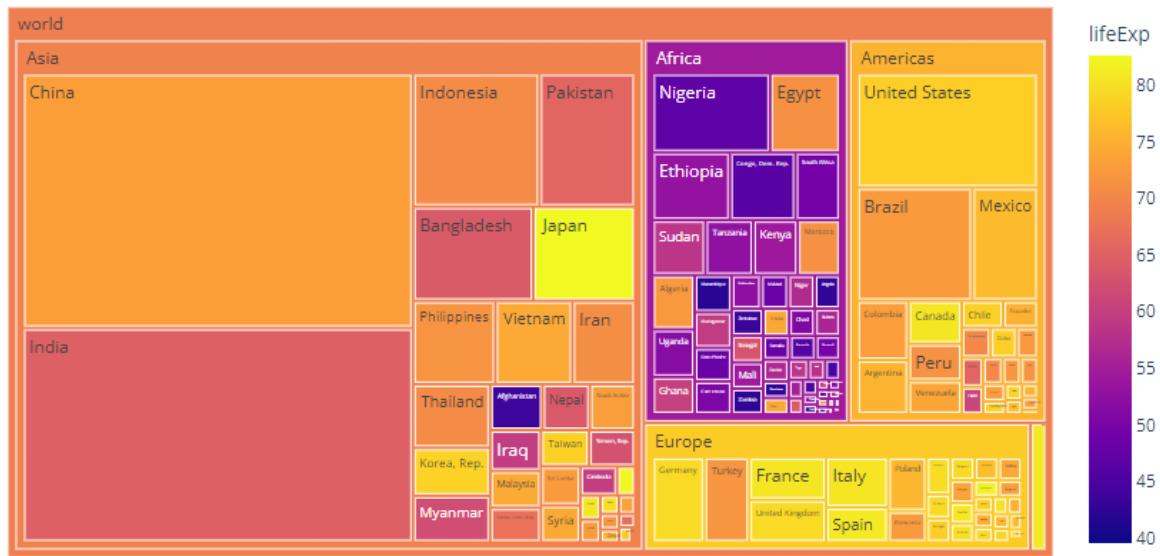
```
import plotly.express as px

df = px.data.gapminder().query("year == 2007")
fig = px.sunburst(df, path=['continent', 'country'], values='pop',
                   color='lifeExp', hover_data=['iso_alpha'])
fig.show()
```



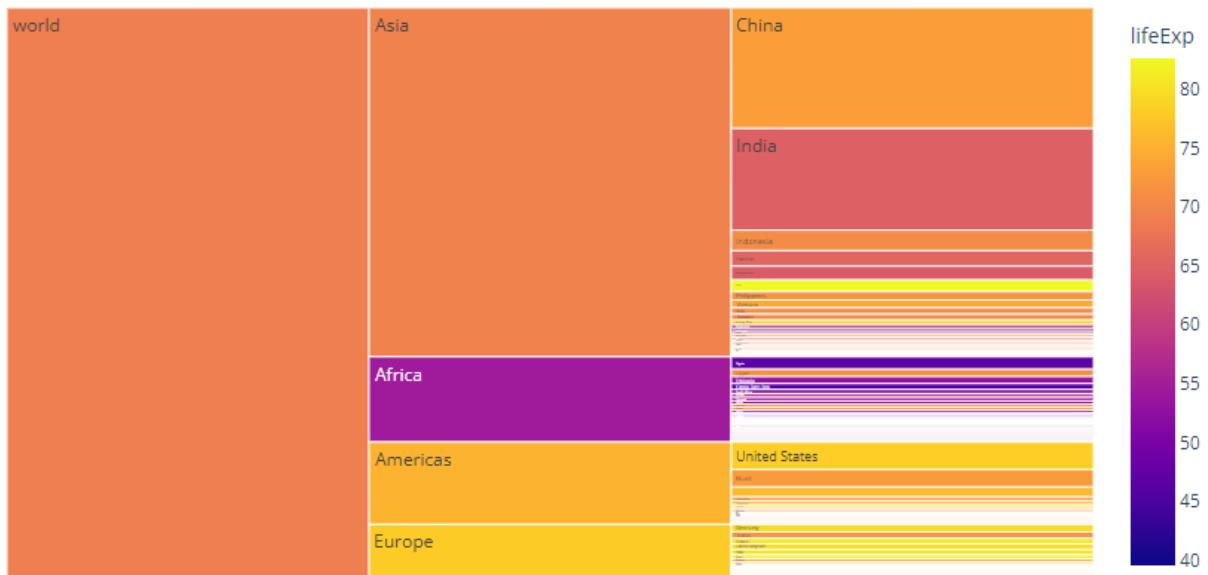
Read more about [treemaps](#).

```
import plotly.express as px
import numpy as np
df = px.data.gapminder().query("year == 2007")
fig = px.treemap(df, path=[px.Constant('world'), 'continent', 'country'], values='pop',
                  color='lifeExp', hover_data=['iso_alpha'])
fig.show()
```



[Read more about icicle charts.](#)

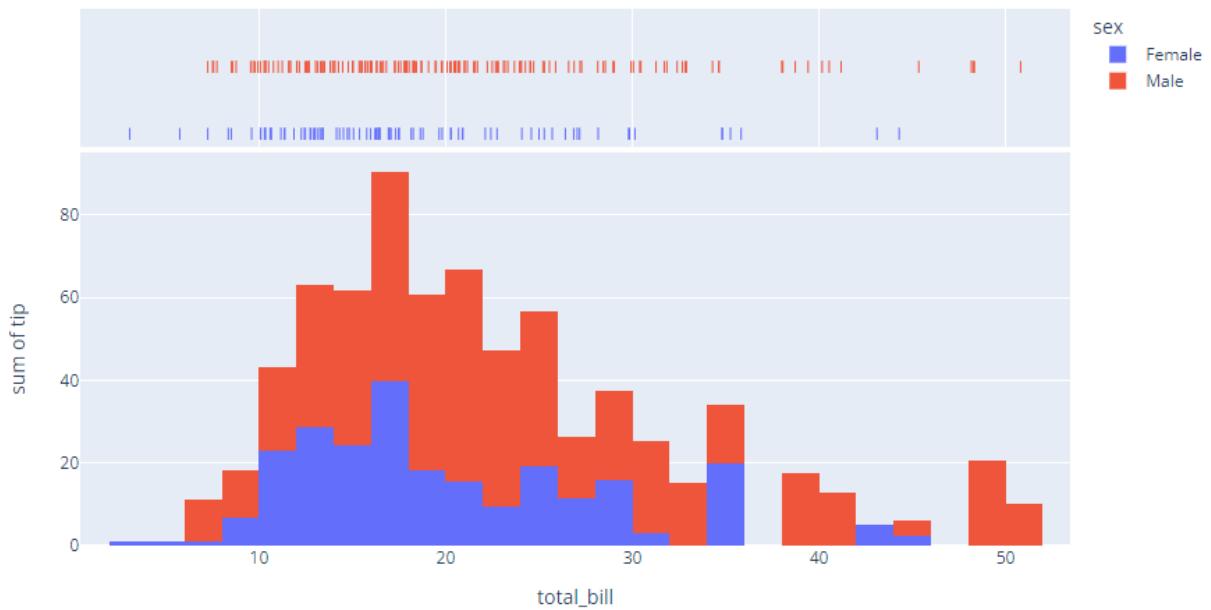
```
import plotly.express as px
import numpy as np
df = px.data.gapminder().query("year == 2007")
fig = px.icicle(df, path=[px.Constant('world'), 'continent', 'country'], values='pop',
                 color='lifeExp', hover_data=['iso_alpha'])
fig.show()
```



Distributions

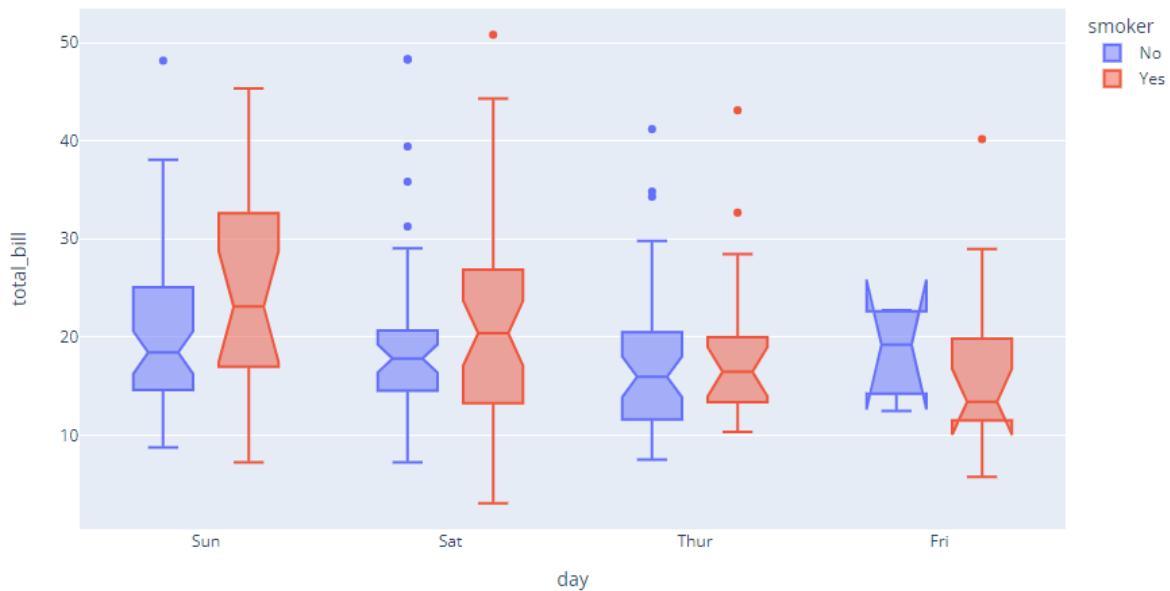
[Read more about histograms.](#)

```
import plotly.express as px
df = px.data.tips()
fig = px.histogram(df, x="total_bill", y="tip", color="sex", marginal="rug", hover_data=df.columns)
fig.show()
```



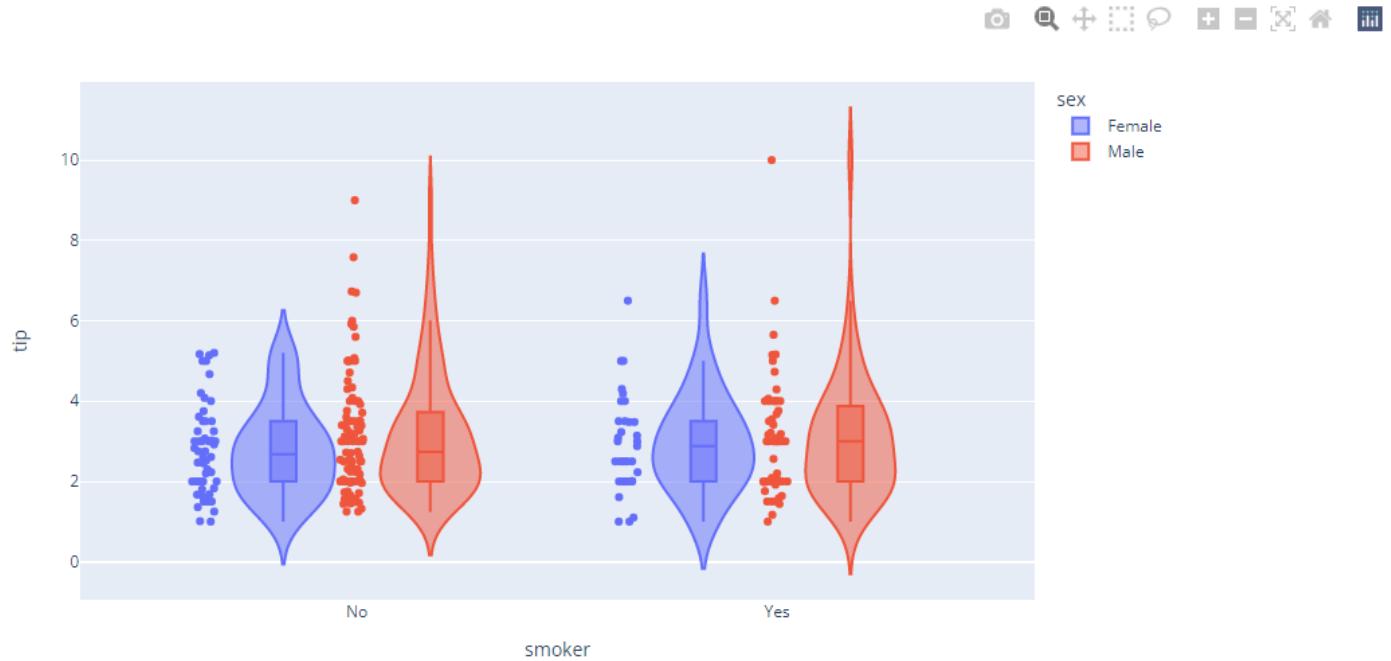
Read more about [box plots](#).

```
import plotly.express as px
df = px.data.tips()
fig = px.box(df, x="day", y="total_bill", color="smoker", notched=True)
fig.show()
```



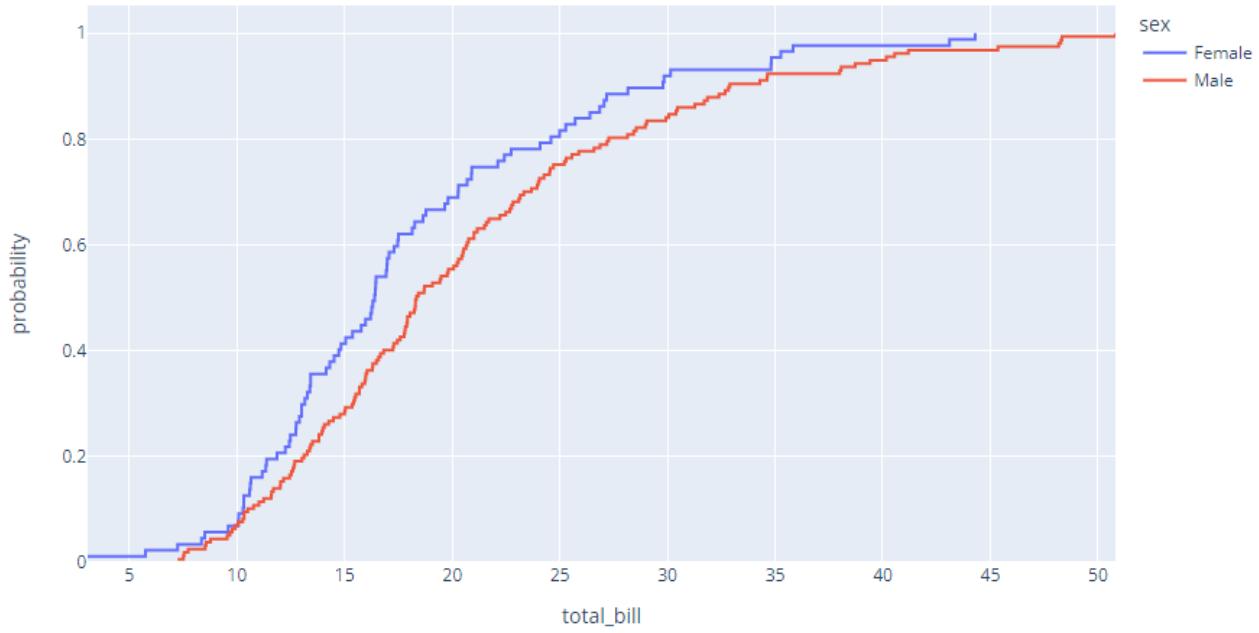
[Read more about violin plots.](#)

```
import plotly.express as px
df = px.data.tips()
fig = px.violin(df, y="tip", x="smoker", color="sex", box=True, points="all", hover_data=df.columns)
fig.show()
```



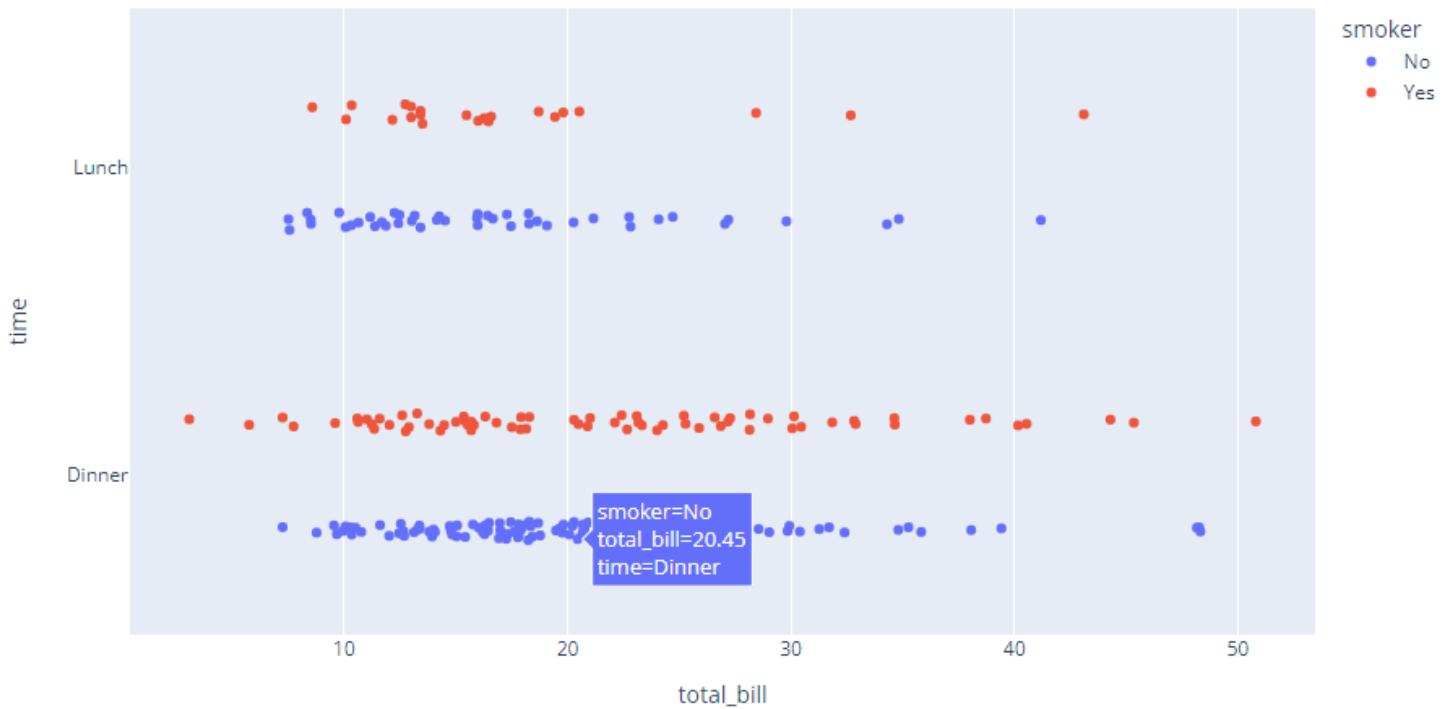
[Read more about Empirical Cumulative Distribution Function \(ECDF\) charts.](#)

```
import plotly.express as px
df = px.data.tips()
fig = px.ecdf(df, x="total_bill", color="sex")
fig.show()
```



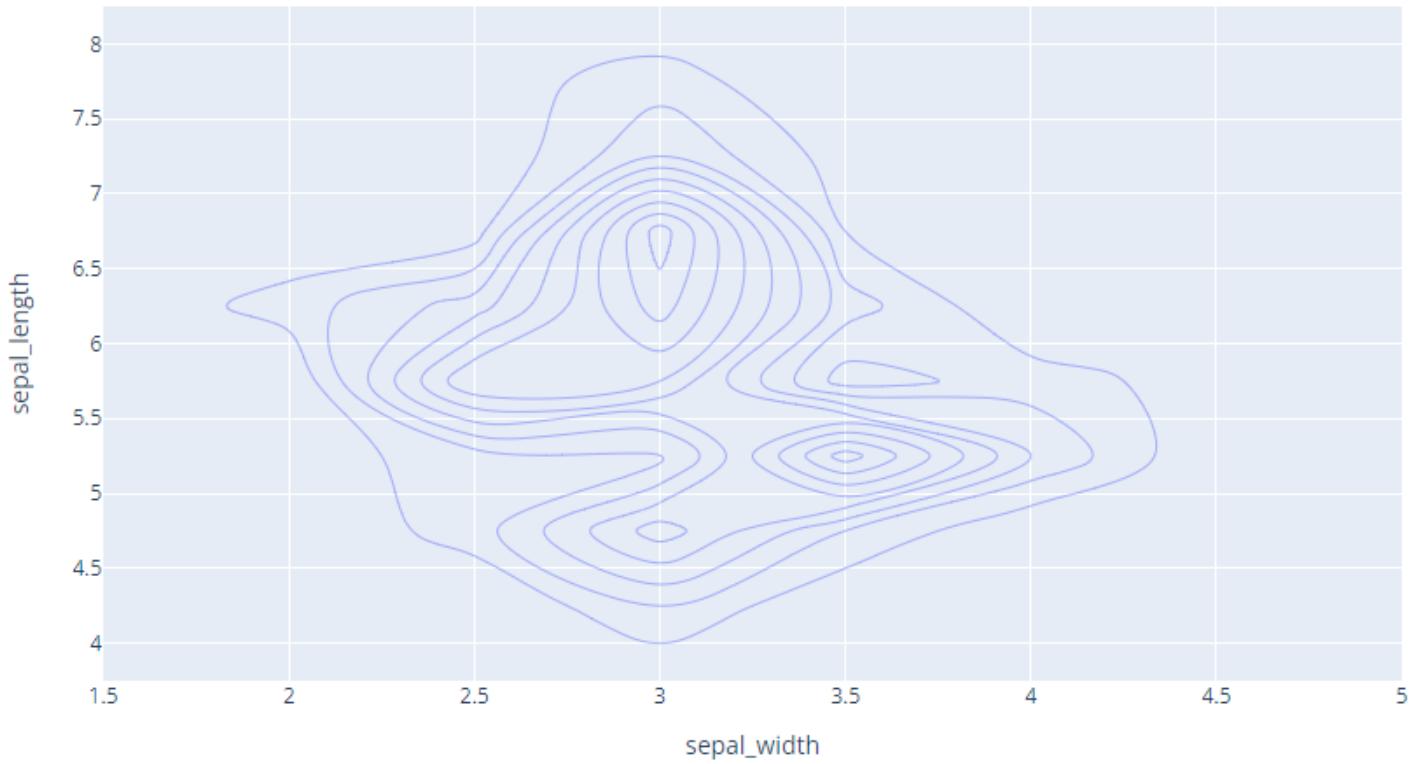
Read more about [strip charts](#).

```
import plotly.express as px
df = px.data.tips()
fig = px.strip(df, x="total_bill", y="time", orientation="h", color="smoker")
fig.show()
```



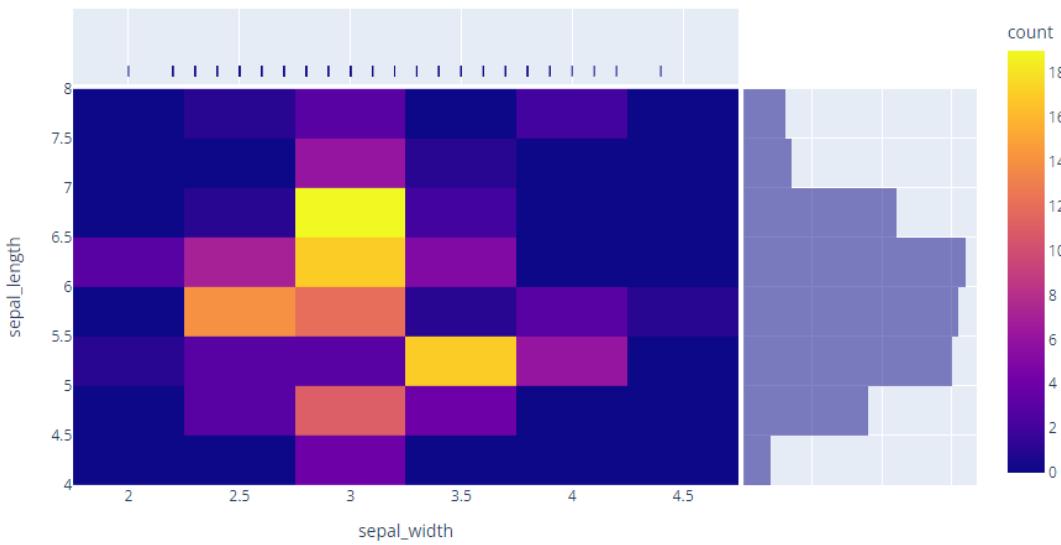
Read more about [density contours, also known as 2D histogram contours](#).

```
import plotly.express as px
df = px.data.iris()
fig = px.density_contour(df, x="sepal_width", y="sepal_length")
fig.show()
```



Read more about [density heatmaps](#), also known as 2D histograms.

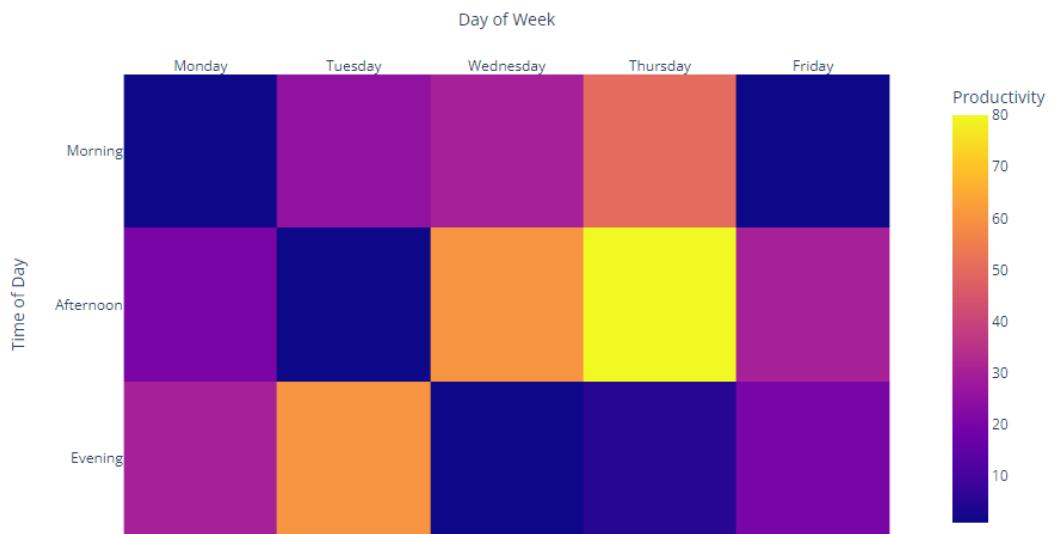
```
import plotly.express as px
df = px.data.iris()
fig = px.density_heatmap(df, x="sepal_width", y="sepal_length", marginal_x="rug", marginal_y="histogram")
fig.show()
```



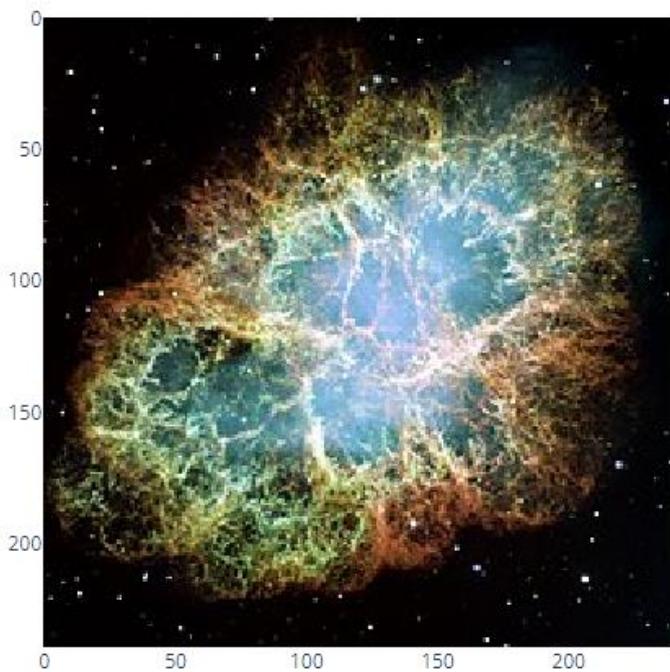
Images and Heatmaps

Read more about [heatmaps and images](#).

```
import plotly.express as px
data=[[1, 25, 30, 50, 1], [20, 1, 60, 80, 30], [30, 60, 1, 5, 20]]
fig = px.imshow(data,
                 labels=dict(x="Day of Week", y="Time of Day", color="Productivity"),
                 x=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'],
                 y=['Morning', 'Afternoon', 'Evening']
                )
fig.update_xaxes(side="top")
fig.show()
```



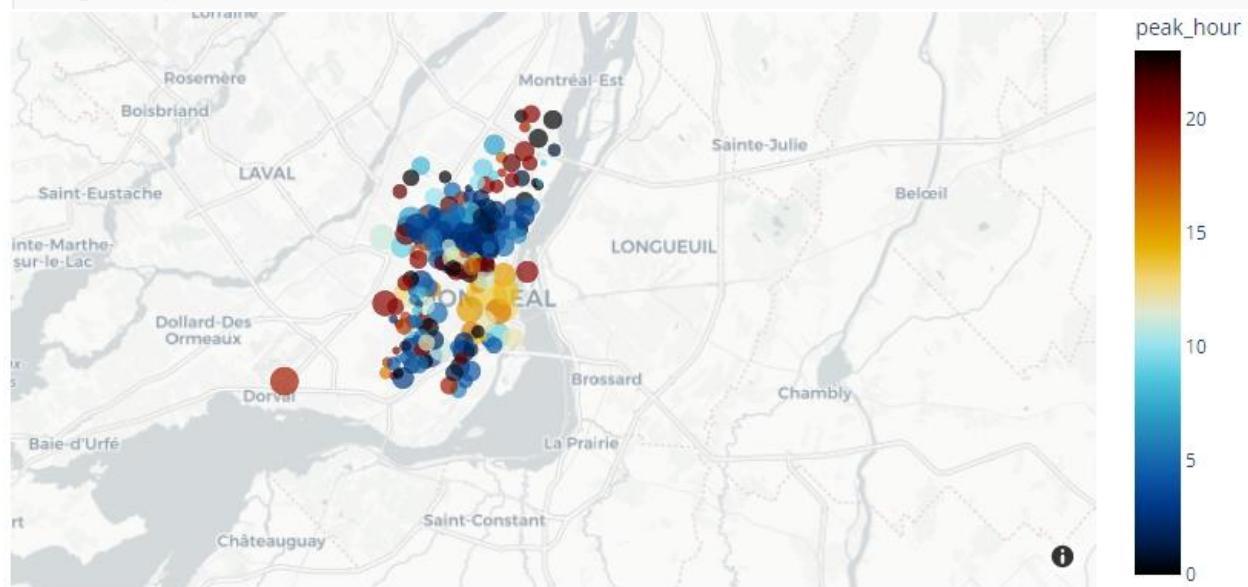
```
import plotly.express as px
from skimage import io
img = io.imread('https://upload.wikimedia.org/wikipedia/commons/thumb/0/00/Crab_Nebula.jpg/240px-Crab_Nebula.jpg')
fig = px.imshow(img)
fig.show()
```



Tile Maps

[Read more about tile maps and point on tile maps.](#)

```
import plotly.express as px
df = px.data.carshare()
fig = px.scatter_mapbox(df, lat="centroid_lat", lon="centroid_lon", color="peak_hour", size="car_hours",
                        color_continuous_scale=px.colors.cyclical.IceFire, size_max=15, zoom=10,
                        mapbox_style="carto-positron")
fig.show()
```

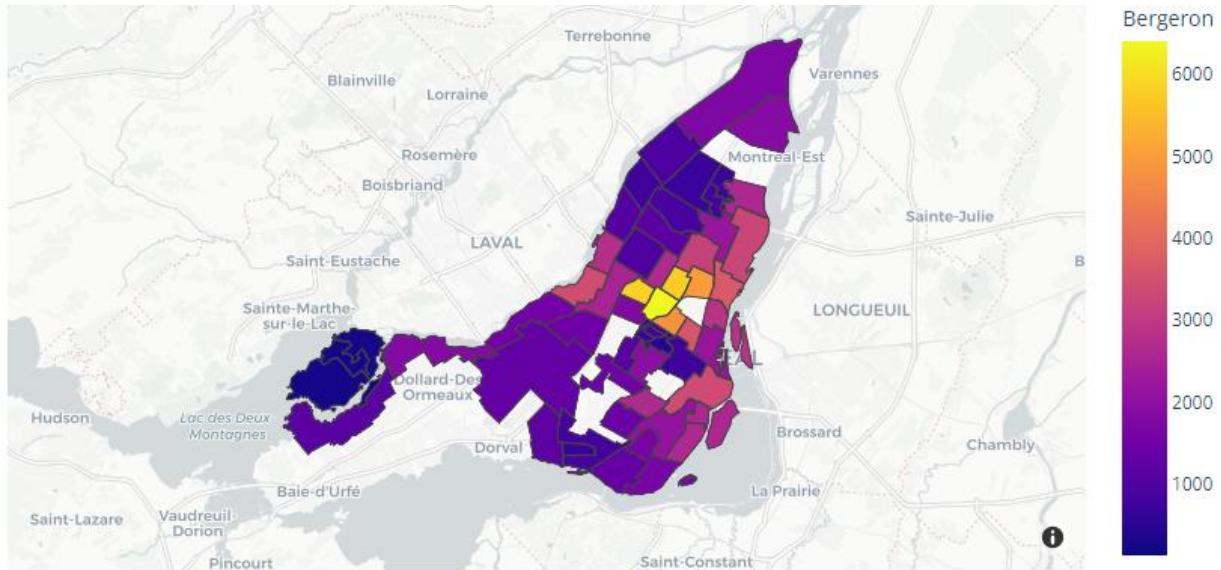


[Read more about tile map GeoJSON choropleths.](#)

```
import plotly.express as px

df = px.data.election()
geojson = px.data.election_geojson()

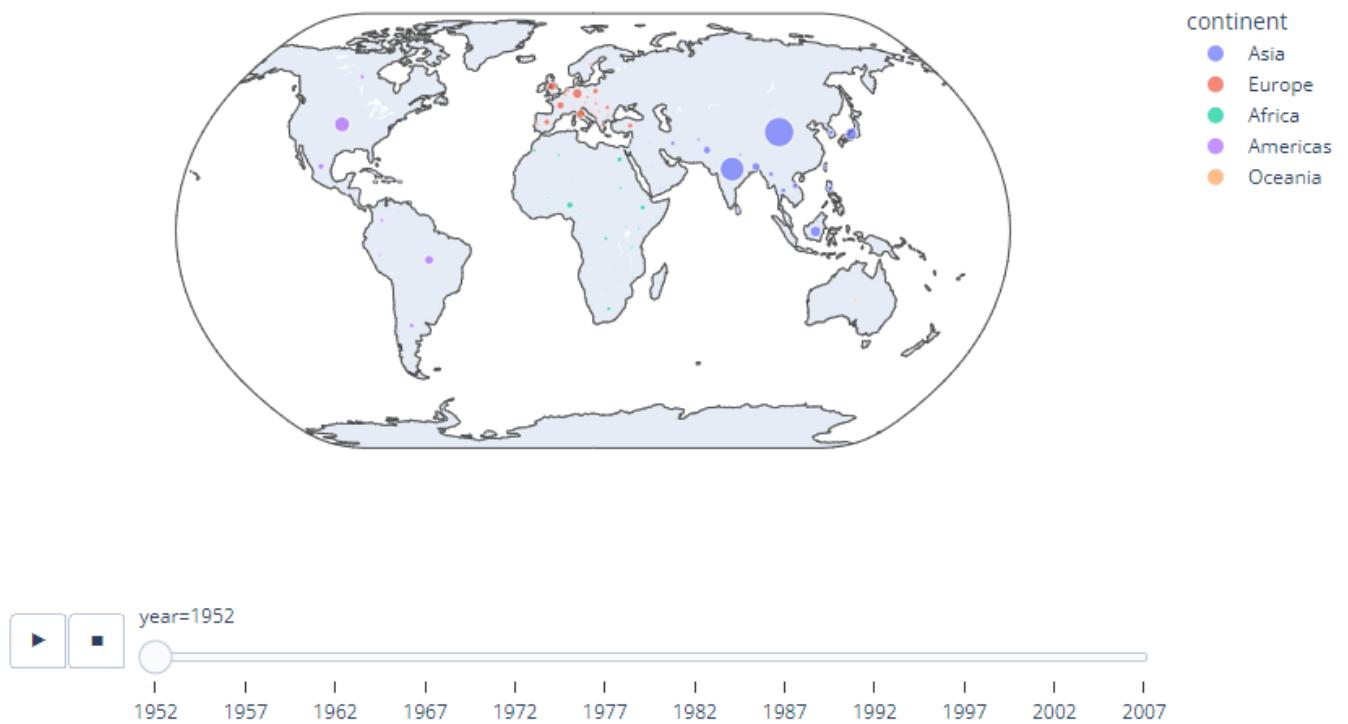
fig = px.choropleth_mapbox(df, geojson=geojson, color="Bergeron",
                           locations="district", featureidkey="properties.district",
                           center={"lat": 45.5517, "lon": -73.7073},
                           mapbox_style="carto-positron", zoom=9)
fig.show()
```



Outline Maps

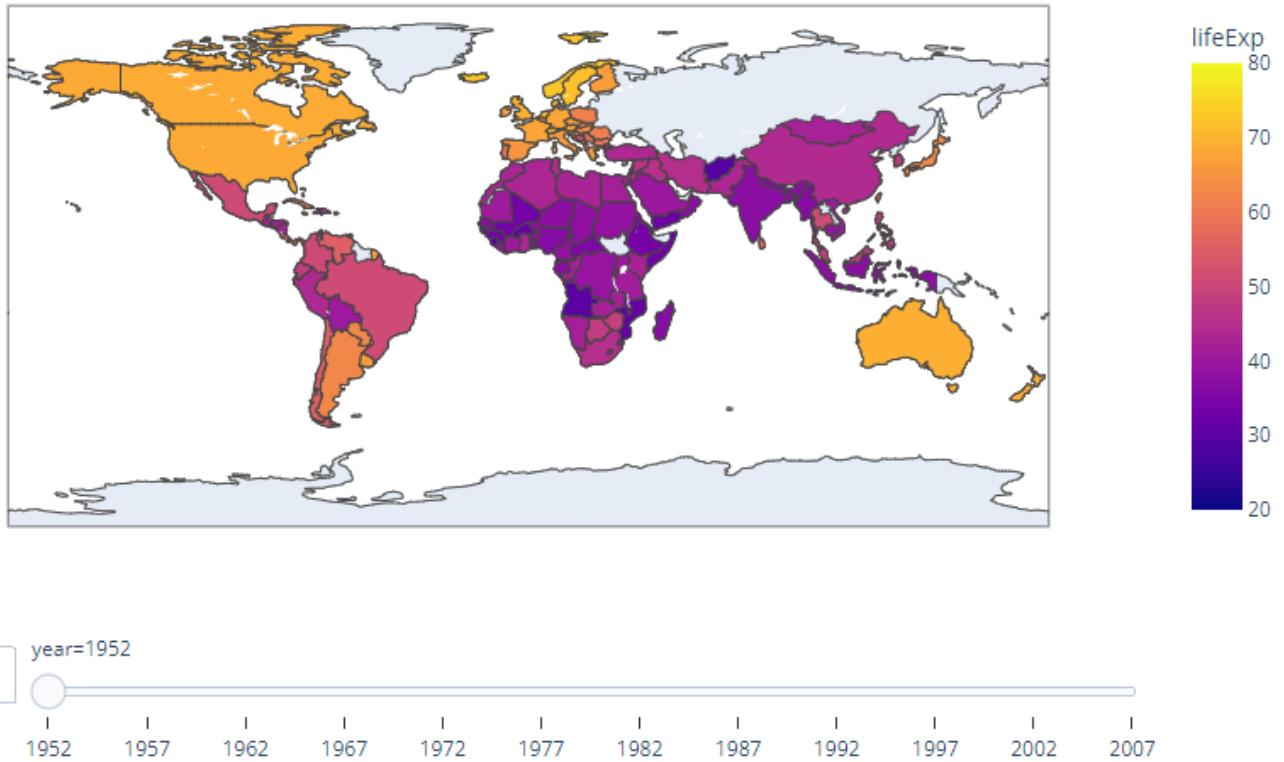
Read more about [outline symbol maps](#).

```
import plotly.express as px
df = px.data.gapminder()
fig = px.scatter_geo(df, locations="iso_alpha", color="continent", hover_name="country",
                     animation_frame="year", projection="natural earth")
fig.show()
```



[Read more about choropleth maps.](#)

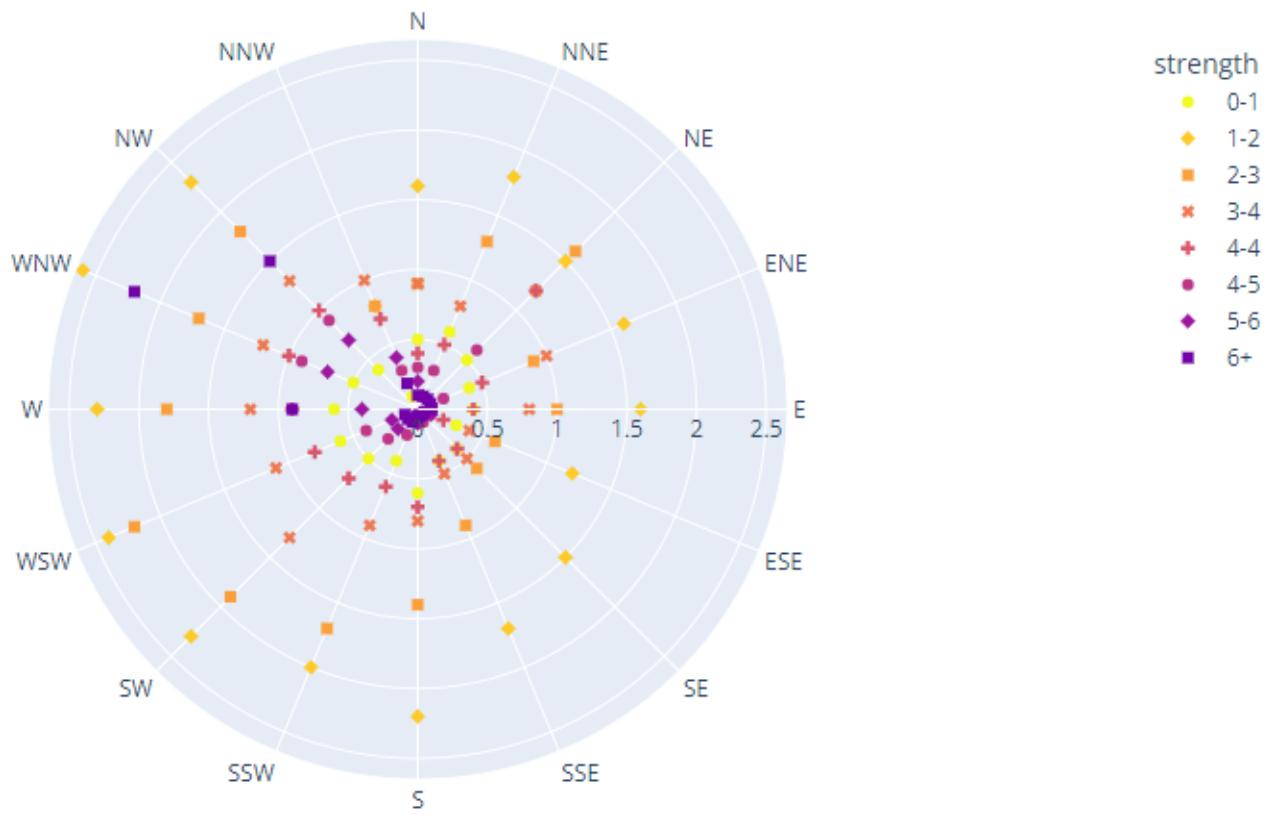
```
import plotly.express as px
df = px.data.gapminder()
fig = px.choropleth(df, locations="iso_alpha", color="lifeExp", hover_name="country", animation_frame="year", range_color=[20,80])
fig.show()
```



Polar Coordinates

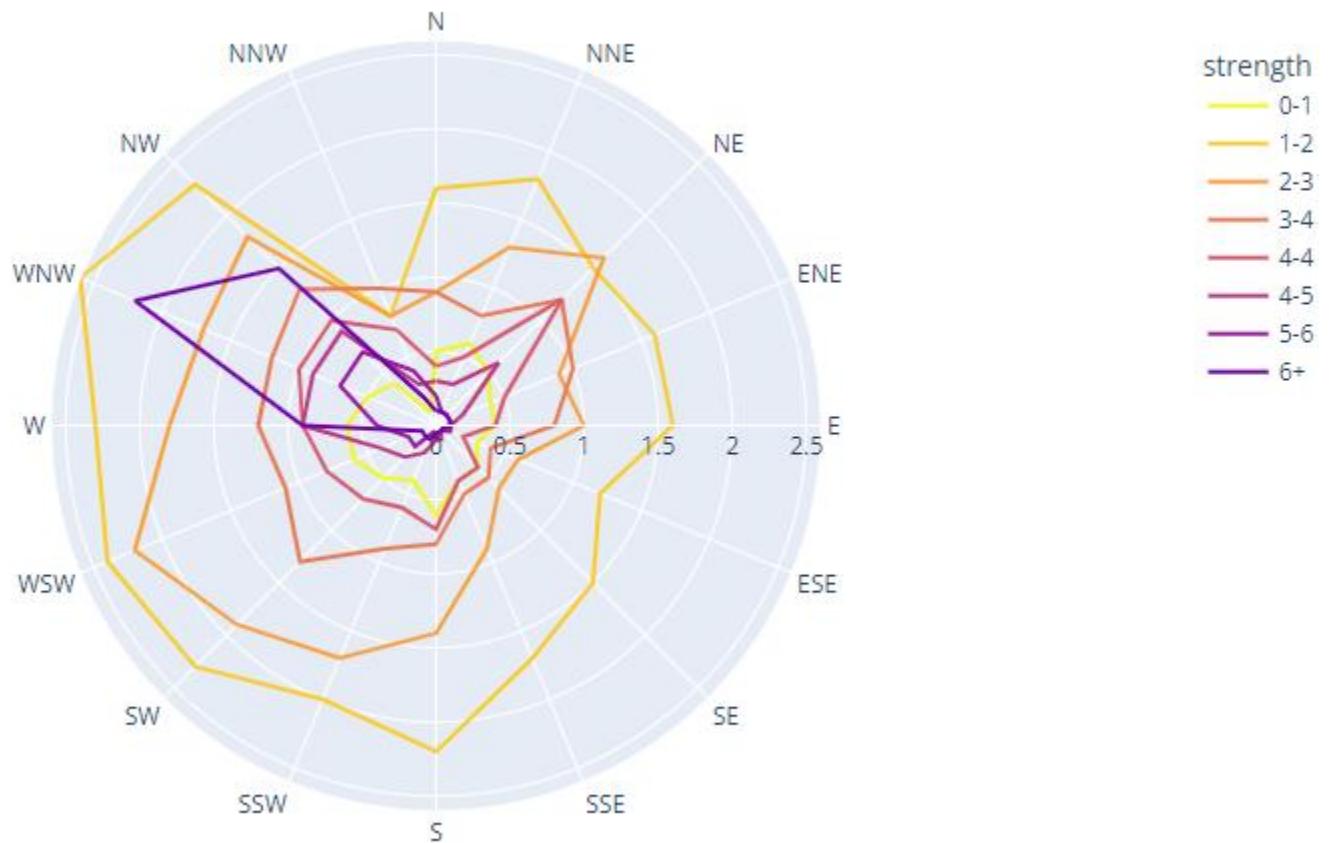
[Read more about polar plots.](#)

```
import plotly.express as px
df = px.data.wind()
fig = px.scatter_polar(df, r="frequency", theta="direction", color="strength", symbol="strength",
                      color_discrete_sequence=px.colors.sequential.Plasma_r)
fig.show()
```



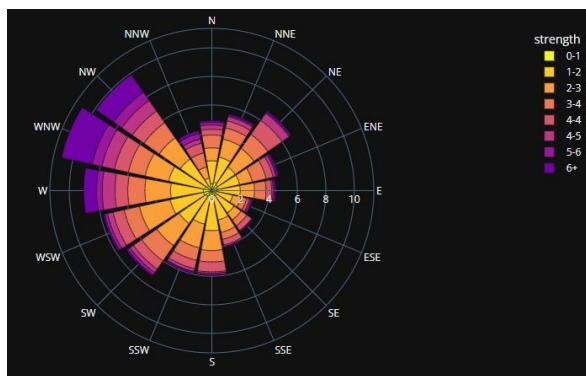
Read more about [radar charts](#).

```
import plotly.express as px
df = px.data.wind()
fig = px.line_polar(df, r="frequency", theta="direction", color="strength", line_close=True,
                     color_discrete_sequence=px.colors.sequential.Plasma_r)
fig.show()
```



[Read more about polar bar charts.](#)

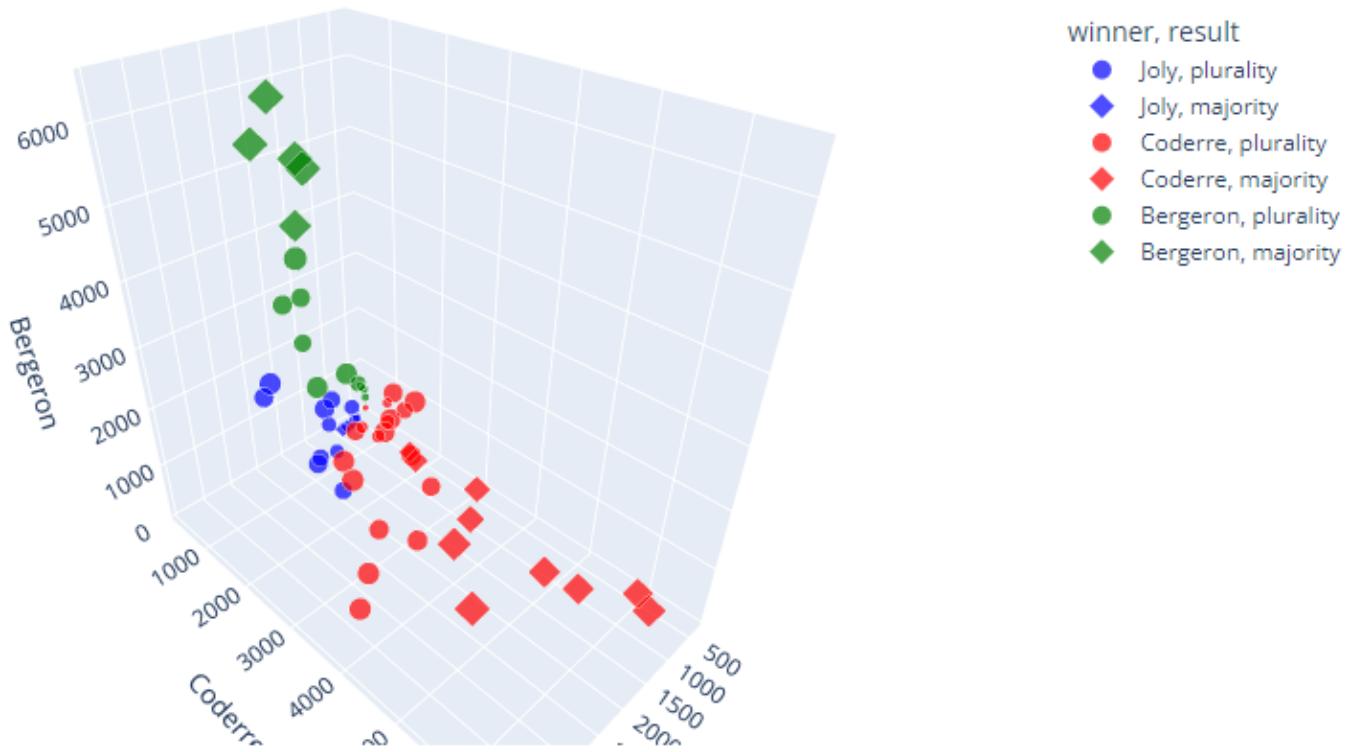
```
import plotly.express as px
df = px.data.wind()
fig = px.bar_polar(df, r="frequency", theta="direction", color="strength", template="plotly_dark",
                    color_discrete_sequence= px.colors.sequential.Plasma_r)
fig.show()
```



3D Coordinates

[Read more about 3D scatter plots.](#)

```
import plotly.express as px
df = px.data.election()
fig = px.scatter_3d(df, x="Joly", y="Coderre", z="Bergeron", color="winner", size="total", hover_name="district",
                     symbol="result", color_discrete_map = {"Joly": "blue", "Bergeron": "green", "Coderre": "red"})
fig.show()
```



Ternary Coordinates

[Read more about ternary charts.](#)

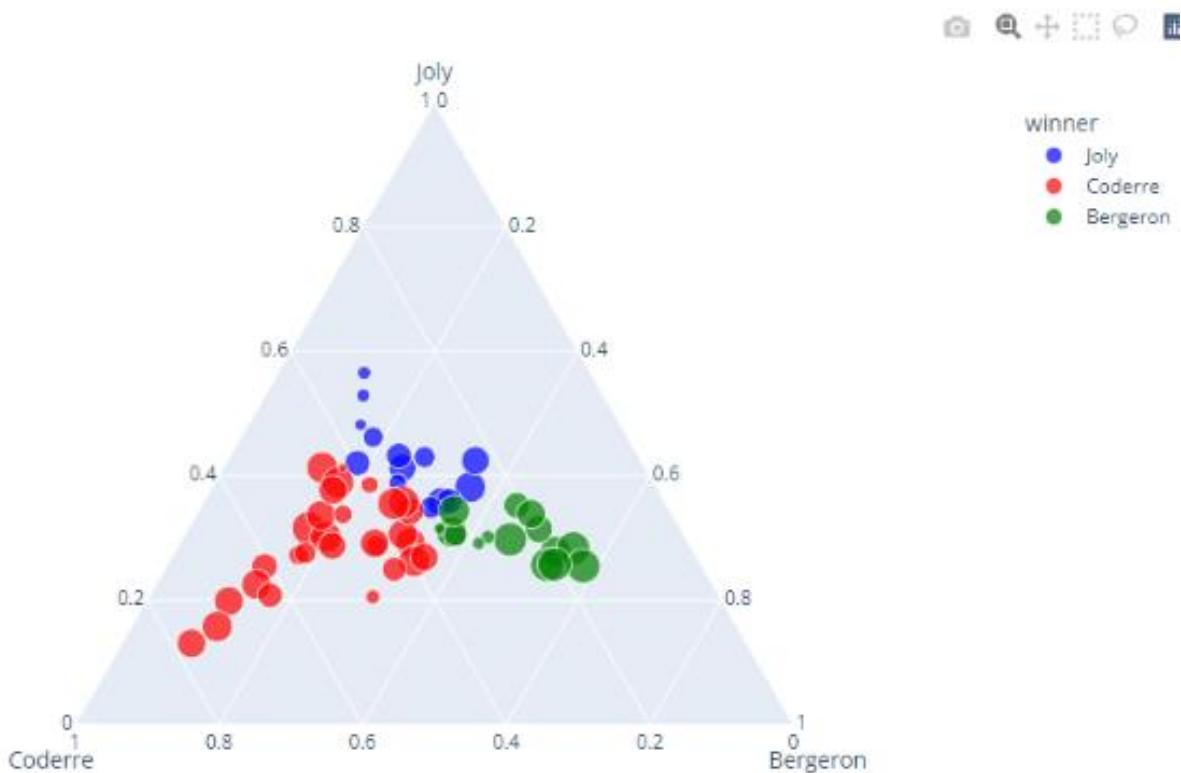
```
import plotly.express as px
df = px.data.election()
fig = px.scatter_ternary(df, a="Joly", b="Coderre", c="Bergeron", color="winner", size="total", hover_name="district",
                        size_max=15, color_discrete_map = {"Joly": "blue", "Bergeron": "green", "Coderre": "red"} )
fig.show()
```

What About Dash?

[Dash](#) is an open-source framework for building analytical applications, with no Javascript required, and it is tightly integrated with the Plotly graphing library.

Learn about how to install Dash at <https://dash.plot.ly/installation>.

Everywhere in this page that you see `fig.show()`, you can display the same figure in a Dash application by passing it to the `figure` argument of the [Graph component](#) from the built-in `dash_core_components` package like this:



Python API reference for plotly

This is the reference of plotly's API. Also see [plotly's documentation website](#).

Submodules

- [Plotly Express: high-level interface for data visualization](#)
- [Graph Objects: low-level interface to figures, traces and layout](#)
- [Subplots: helper function for layout out multi-plot figures](#)
- [Figure Factories: helper methods for building specific complex charts](#)
- [I/O: low-level interface for displaying, reading and writing figures](#)
- [plotly.colors](#): colorscales and utility functions
- [plotly.data](#): built-in datasets for demonstration, educational and test purposes

Full reference list

- [plotly.express: high-level interface for data visualization](#)
 - [plotly.express.scatter](#)
 - [plotly.express.scatter_3d](#)
 - [plotly.express.scatter_polar](#)
 - [plotly.express.scatter_ternary](#)
 - [plotly.express.scatter_mapbox](#)
 - [plotly.express.scatter_geo](#)
 - [plotly.express.line](#)
 - [plotly.express.line_3d](#)
 - [plotly.express.line_polar](#)
 - [plotly.express.line_ternary](#)
 - [plotly.express.line_mapbox](#)
 - [plotly.express.line_geo](#)
 - [plotly.express.area](#)
 - [plotly.express.bar](#)
 - [plotly.express.timeline](#)
 - [plotly.express.bar_polar](#)

- [plotly.express.violin](#)
- [plotly.express.box](#)
- [plotly.express.ecdf](#)
- [plotly.express.strip](#)
- [plotly.express.histogram](#)
- [plotly.express.pie](#)
- [plotly.express.treemap](#)
- [plotly.express.sunburst](#)
- [plotly.express.icicle](#)
- [plotly.express.funnel](#)
- [plotly.express.funnel_area](#)
- [plotly.express.scatter_matrix](#)
- [plotly.express.parallel_coordinates](#)
- [plotly.express.parallel_categories](#)
- [plotly.express.choropleth](#)
- [plotly.express.choropleth_mapbox](#)
- [plotly.express.density_contour](#)
- [plotly.express.density_heatmap](#)
- [plotly.express.density_mapbox](#)
- [plotly.express.imshow](#)
- [plotly.express.set_mapbox_access_token](#)
- [plotly.express.get_trendline_results](#)
- [plotly.express subpackages](#)
 - [plotly.express.data_package](#)
 - [plotly.express.colors_package](#)
 - [plotly.express.trendline_functions_package](#)
- [plotly.graph_objects: low-level interface to figures, traces and layout](#)
 - [Figure](#)

- [`plotly.graph_objects.Figure`](#)
- [Layout](#)
 - [`plotly.graph_objects.Layout`](#)
 - [`plotly.graph_objects.layout`](#)
- [Simple Traces](#)
 - [`plotly.graph_objects.Scatter`](#)
 - [`plotly.graph_objects.scatter`](#)
 - [`plotly.graph_objects.Scattergl`](#)
 - [`plotly.graph_objects.scattergl`](#)
 - [`plotly.graph_objects.Bar`](#)
 - [`plotly.graph_objects.bar`](#)
 - [`plotly.graph_objects.Pie`](#)
 - [`plotly.graph_objects.pie`](#)
 - [`plotly.graph_objects.Heatmap`](#)
 - [`plotly.graph_objects.heatmap`](#)
 - [`plotly.graph_objects.Heatmapgl`](#)
 - [`plotly.graph_objects.heatmapgl`](#)
 - [`plotly.graph_objects.Image`](#)
 - [`plotly.graph_objects.image`](#)
 - [`plotly.graph_objects.Contour`](#)
 - [`plotly.graph_objects.contour`](#)
 - [`plotly.graph_objects.Table`](#)
 - [`plotly.graph_objects.table`](#)
- [Distribution Traces](#)
 - [`plotly.graph_objects.Box`](#)
 - [`plotly.graph_objects.box`](#)
 - [`plotly.graph_objects.Violin`](#)
 - [`plotly.graph_objects.violin`](#)

- [`plotly.graph_objects.Histogram`](#)
- [`plotly.graph_objects.histogram`](#)
- [`plotly.graph_objects.Histogram2d`](#)
- [`plotly.graph_objects.histogram2d`](#)
- [`plotly.graph_objects.Histogram2dContour`](#)
- [`plotly.graph_objects.histogram2dcontour`](#)
- [Finance Traces](#)
 - [`plotly.graph_objects.Ohlc`](#)
 - [`plotly.graph_objects.ohlc`](#)
 - [`plotly.graph_objects.Candlestick`](#)
 - [`plotly.graph_objects.candlestick`](#)
 - [`plotly.graph_objects.Waterfall`](#)
 - [`plotly.graph_objects.waterfall`](#)
 - [`plotly.graph_objects.Funnel`](#)
 - [`plotly.graph_objects.funnel`](#)
 - [`plotly.graph_objects.Funnelarea`](#)
 - [`plotly.graph_objects.funnelarea`](#)
 - [`plotly.graph_objects.Indicator`](#)
 - [`plotly.graph_objects.indicator`](#)
 - [`plotly.graph_objects.Scatter3d`](#)
 - [`plotly.graph_objects.scatter3d`](#)
 - [`plotly.graph_objects.Surface`](#)
 - [`plotly.graph_objects.surface`](#)
 - [`plotly.graph_objects.Mesh3d`](#)
 - [`plotly.graph_objects.mesh3d`](#)
 - [`plotly.graph_objects.Cone`](#)
 - [`plotly.graph_objects.cone`](#)
 - [`plotly.graph_objects.Streamtube`](#)

- [`plotly.graph_objects.streamtube`](#)
- [`plotly.graph_objects.Volume`](#)
- [`plotly.graph_objects.volume`](#)
- [`plotly.graph_objects.Isosurface`](#)
- [`plotly.graph_objects.isosurface`](#)
- [Map Traces](#)
 - [`plotly.graph_objects.Scattergeo`](#)
 - [`plotly.graph_objects.scattergeo`](#)
 - [`plotly.graph_objects.Choropleth`](#)
 - [`plotly.graph_objects.choropleth`](#)
 - [`plotly.graph_objects.Scattermapbox`](#)
 - [`plotly.graph_objects.scattermapbox`](#)
 - [`plotly.graph_objects.Choroplethmapbox`](#)
 - [`plotly.graph_objects.choroplethmapbox`](#)
 - [`plotly.graph_objects.Densitymapbox`](#)
 - [`plotly.graph_objects.densitymapbox`](#)
- [Specialized Traces](#)
 - [`plotly.graph_objects.Scatterpolar`](#)
 - [`plotly.graph_objects.scatterpolar`](#)
 - [`plotly.graph_objects.Scatterpolargl`](#)
 - [`plotly.graph_objects.scatterpolargl`](#)
 - [`plotly.graph_objects.Barpolar`](#)
 - [`plotly.graph_objects.barpolar`](#)
 - [`plotly.graph_objects.Scatterternary`](#)
 - [`plotly.graph_objects.scatterternary`](#)
 - [`plotly.graph_objects.Sunburst`](#)
 - [`plotly.graph_objects.sunburst`](#)
 - [`plotly.graph_objects.Treemap`](#)

- [`plotly.graph_objects.treemap`](#)
- [`plotly.graph_objects.icicle`](#)
- [`plotly.graph_objects.icicle`](#)
- [`plotly.graph_objects.Sankey`](#)
- [`plotly.graph_objects.sankey`](#)
- [`plotly.graph_objects.Splom`](#)
- [`plotly.graph_objects.splom`](#)
- [`plotly.graph_objects.Parcats`](#)
- [`plotly.graph_objects.parcats`](#)
- [`plotly.graph_objects.Parcoords`](#)
- [`plotly.graph_objects.parcoords`](#)
- [`plotly.graph_objects.Carpet`](#)
- [`plotly.graph_objects.carpet`](#)
- [`plotly.graph_objects.Scattercarpet`](#)
- [`plotly.graph_objects.scattercarpet`](#)
- [`plotly.graph_objects.Contourcarpet`](#)
- [`plotly.graph_objects.contourcarpet`](#)
- [`plotly.subplots`](#): helper function for laying out multi-plot figures
 - [`plotly.subplots.make_subplots`](#)
- [`plotly.figure_factory`](#): helper methods for building specific complex charts
 - [`plotly.figure_factory.create_2d_density`](#)
 - [`plotly.figure_factory.create.annotated_heatmap`](#)
 - [`plotly.figure_factory.create_bullet`](#)
 - [`plotly.figure_factory.create_candlestick`](#)
 - [`plotly.figure_factory.create_choropleth`](#)
 - [`plotly.figure_factory.create_dendrogram`](#)
 - [`plotly.figure_factory.create_distplot`](#)
 - [`plotly.figure_factory.create_facet_grid`](#)

- [`plotly.figure_factory.create_gantt`](#)
- [`plotly.figure_factory.create_hexbin_mapbox`](#)
- [`plotly.figure_factory.create_ohlc`](#)
- [`plotly.figure_factory.create_quiver`](#)
- [`plotly.figure_factory.create_scatterplotmatrix`](#)
- [`plotly.figure_factory.create_streamline`](#)
- [`plotly.figure_factory.create_table`](#)
- [`plotly.figure_factory.create_ternary_contour`](#)
- [`plotly.figure_factory.create_trisurf`](#)
- [`plotly.figure_factory.create_violin`](#)
- [`plotly.io`: low-level interface for displaying, reading and writing figures](#)
 - [`plotly.io.to_image`](#)
 - [`plotly.io.write_image`](#)
 - [`plotly.io.to_json`](#)
 - [`plotly.io.from_json`](#)
 - [`plotly.io.read_json`](#)
 - [`plotly.io.write_json`](#)
 - [`plotly.io.templates`](#)
 - [`plotly.io.to_templated`](#)
 - [`plotly.io.to_html`](#)
 - [`plotly.io.write_html`](#)
 - [`plotly.io.renderers`](#)
 - [`plotly.io.show`](#)

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

<https://developer.ibm.com/exchanges/data/>

Data Asset eXchange

Explore useful and relevant data sets for enterprise data science

Dataset | Various formats

Project CodeNet

May 5, 2021



Dataset | CSV

NOAA Weather Data -
JFK Airport

August 11, 2020



Dataset | CSV

Airline Reporting Carrier
On-Time Performance
Dataset

November 23, 2020



Dataset | IOB format

Groningen Meaning Bank
- Modified

May 14, 2020



Dataset | CSV

Fashion-MNIST

August 17, 2020



Dataset | CSV

COVID-19 Questions

October 1, 2020



Dataset | JPG, JSON

PubLavNet

Dataset | WAV

TensorFlow Speech

Dataset | PNG, JSON

PubTabNet

8.4.Creating Dashboards with Plotly and Dash

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 1 revision

In this module you will get started with dashboard creation using the Plotly library. You will create a dashboard with a theme 'US Domestic Airline Flights Performance'. You will do this using a US airline reporting carrier on-time performance dataset, Plotly, and Dash concepts learned throughout the course. Hands-on labs will follow each concept to make you comfortable with using the library. Reading lists will reference additional resources to learn more about the concepts covered.

Learning Objectives

- Identify high-level popular Python dashboarding tools.
- Demonstrate basic Plotly, Plotly.graph_objects, and Plotly express commands.
- Demonstrate using Dash and basic Dash components (core and HTML).
- Demonstrate adding different dashboard elements including text boxes, dropdown, graphs, and others.
- Apply interactivity to dash core and HTML components.
- Describe how a dashboard can be used to answer critical business questions.

-
- [Creating Dashboards with Plotly and Dash](#)
 - [Lab 4-1: Plotly basics: scatter, line, bar, bubble, histogram, pie, sunburst](#)
 - [Lab 4-2: Dash basics: HTML and core components](#)
 - [Lab 4-3: Add interactivity: user input and callbacks](#)
 - [Lab 4-4: Flight Delay Time Statistics Dashboard](#)

8.4.1.Creating Dashboards with Plotly And Dash

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Module Overview and Learning Objectives

As the saying goes, A picture worth thousand words. Data visualization through dashboards will help you uncover information from data that are hidden and democratize the understanding of the extracted information.

In this topic, you will create a dashboard with theme US Domestic Airline Flights Performance. You will do this using a US airline reporting carrier on-time performance dataset, plotly, and dash concepts learned throughout the course.

In this module, you will learn

- How a dashboard can be used to answer critical business questions.
- What high-level overview of popular dashboarding tools available in python.
- How to use basic Plotly, `plotly.graph_objects`, and `plotly express`.
- How to use Dash and basic overview of dash components (core and HTML).
- How to add different elements (like text box, dropdown, graphs, etc) to the dashboard.
- How to add interactivity to dash core and HTML components.

This module will help you get started with dashboard creation using the Plotly library. Hands-on labs will follow each concept to make you comfortable with using the library.

Reading lists will reference additional resources to learn more about the concepts covered.

Dashboarding Overview

Dashboard

- Real-time visuals
- Understand business moving parts
- Visually track, analyze, and display key performance indicators (KPI)
- Take informed decisions and improve performance
- Reduced hours of analyzing

Best dashboards answer important business questions.

Scenario

Monitor and report US airline performance *Requested report items*

1. Top 10 airline carrier in year 2019 in terms of number of flights
2. Number of flights in 2019 split by month
3. Number of travelers from California (CA) to other states split by distance group

Introduction to Plotly

Plotly - An Overview

- Interactive, open-source plotting library
- Supports over 40 unique chart types
- Includes chart types like statistical, financial, maps, scientific, and 3-dimensional
- Visualizations can be displayed in Jupyter notebook, saved to HTML files, or can be used in developing Python-built web applications

Plotly Sub-modules

- Plotly Graph Objects: Low-level interface to figures, traces, and layout
`plotly.graph_objects.Figure`
- Plotly Express: High-level wrapper

Using `plotly.graph_objects`

```
# Import required packages
import plotly.graph_objects as go
import plotly.express as px
import numpy as np

# Set random seed for reproducibility
np.random.seed(10)
x = np.arange(12)
# Create random y values
y = np.random.randint(50, 500, size=12)
```

```
# Line Plot using graph objects
# Plotly.graph contains a JSON object which has a structure of dict
# Here, 'go' is the plotly JSON object
# Updating values of 'go' object keywords, chart can be plotted
# Create figure and add trace (scatter)

fig = go.Figure(data=go.Scatter(x=x,y=y))
fig.update_layout(title='Simple Line Plot', xaxis_title='Month', yaxis_title='Sales')
fig.show()
```

- Create the same line chart using Plotly express

```
# Entire line chart can be created in a single command
fig = px.line(x=x, y=y, title='Simple Line Plot',
               labels=dict(x='Month', y='Sales'))
fig.show()
```

Additional Resources for Plotly

To learn more about using Plotly to create dashboards, explore

[Plotly python](#)

[Plotly graph objects with example](#)

[Plotly express](#)

[API reference](#)

Here are additional useful resources:

[Plotly cheatsheet](#)

[Plotly community](#)

[Related blogs](#)

[Open-source datasets](#)

Introduction to Dash

Dash - An Overview

- Open-source User Interface Python library from Plotly
- Easy to build GUI
- Declarative and Reactive
- Rendered in web browser and can be deployed to servers
- Inherently cross-platform and mobile ready

Dash Components

- Core components

```
import dash_core_components as dcc
```

- HTML components

```
import dash_html_components as html
```

HTML Components

- Component for every HTML tag
- Keyword arguments describe the HTML attributes like style, className, and id

Core Components

- Higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library
- Example: Creating a slider, input area, check items, datepicker and so on

Additional Resources for Dash

To learn more about Dash, explore

[Complete dash user guide](#)

[Dash core components](#)

[Dash HTML components](#)

[Dash community forum](#)

[Related blogs](#)

Make dashboards interactive

Dash - Callbacks

- Callback function is a python function that are automatically called by Dash whenever an input component's property changes.
- `**@app.callback**`

Dash - Callback Function

```
@app.callback(Output, Input)
def callback_function
    ....
    ....
    ....
    return some_result
```

Callback with one input

```
# Import required packages
import pandas as pd
import plotly.express as px
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output

# Read the data
airline_data = pd.read_csv('airline_2m.csv', encoding='ISO-8859-1',
                           dtype={'Div1Airport': str, 'Div1TailNum': str,
                                  'Div2Airport': str, 'Div2TailNum': str})

app = dash.Dash()
# Design dash app layout
app.layout = html.Div(children=[html.H1('Airline Dashboard',
                                         style={'text-align': 'Center', 'color': colors['text'],
                                                'font-size': 40}),
                                 html.Div(['Input: ', dcc.Input(id='input-yr', value='2010',
                                                               type='number', style={'height': '50px', 'font-size': 35}),
                                         style={'font-size': 40}),
                                 html.Br(),
                                 html.Br(),
                                 html.Div(dcc.Graph(id='bar-plot')),
                               ])
@app.callback(Output(component_id='bar-plot', component_property='figure'),
              Input(component_id='input-yr', component_property='value'))

def get_graph(entered_year):
    # Select data
    df = airline_data[airline_data['Year']==int(entered_year)]
    # Top 10 airline carrier in terms of number of flights
    g1 = df.groupby(['Reporting_Airline'])['Flights'].sum().nlargest(10).reset_index()
    # Plot the graph
    fig1 = px.bar(g1, x='Reporting_Airline', y='Flights',
                  title='Top 10 airline carrier in year' + str(entered_year)
                  + ' in terms of number of flights')
    fig1.update_layout()
    return fig1
if __name__ == '__main__':
    app.run_server(port=8002, host='127.0.0.1', debug=True)
```

Callback with two inputs

```
app = dash.Dash()
#Design dash app layout
app.layout = html.Div(children=[html.H1('Airline Dashboard',
                                         style={'textAlign': 'Center', 'color': colors['text'],
                                                 'font-size': 40}),
                                 html.Div(['Year: ', dcc.Input(id='input-yr', value='2010',
                                                               type='number', style={'height':'50px', 'font-size': 35}.)],
                                         style={'font-size': 40}),
                                 html.Div(['State Abbreviation: ',
                                         dcc.Input(id='input-ab', value='AL',
                                                               type='text', style={'height':'50px', 'font-size': 35}.)],
                                         style={'font-size': 40}),
                                 html.Br(),
                                 html.Br(),
                                 html.Div(dcc.Graph(id='bar-plot')),
                               ])
]

@app.callback(Output(component_id='bar-plot', component_property='figure'),
              [Input(component_id='input-yr', component_property='value'),
               Input(component_id='input-ab', component_property='value')])

def get_graph(entered_year, entered_state):
    # Select data
    df = airline_data[(airline_data['Year']==int(entered_year) &
                       (airline_data['OriginState']==entered_state))]
    # Top 10 airline carrier in terms of number of flights
    g1 = df.groupby(['Reporting_Airline'])['Flights'].sum().nlargest(10).reset_index()
    # Plot the graph
    fig1 = px.bar(g1, x='Reporting_Airline', y='Flights',
                  title='Top 10 airline carrier in year' + str(entered_year)
                  + ' in terms of number of flights')
    fig1.update_layout()
    return fig1

if __name__ == '__main__':
    app.run_server(port=8002, host='127.0.0.1', debug=True)
```

References

- Dash Basic Callbacks: <https://dash.plotly.com/basic-callbacks>
- Core Components: <https://dash.plotly.com/dash-core-components>
- HTML Components: <https://dash.plotly.com/dash-html-components>

Additional Resources for Interactive Dashboards

To learn more about making interactive dashboards in Dash, visit

[Python decorators reference 1](#)

[Python decorators reference 2](#)

[Callbacks with example](#)

[Dash app gallery](#)

[Dash community components](#)

Lesson Summary

- Best dashboards answer critical business questions. It will help business make informed decisions, thereby improving performance.
- Dashboards can produce real-time visuals.
- Plotly is an interactive, open-source plotting library that supports over 40 chart types.
- The web based visualizations created using Plotly python can be displayed in Jupyter notebook, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash.
- Plotly Graph Objects is the low-level interface to figures, traces, and layout whereas plotly express is a high-level wrapper for Plotly.
- Dash is an Open-Source User Interface Python library for creating reactive, web-based applications. It is both enterprise-ready and a first-class member of Plotly's open-source tools.
- Core and HTML are the two components of dash.
- The `dash_html_components` library has a component for every HTML tag.
- The `dash_core_components` describe higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library.
- A callback function is a python function that is automatically called by Dash whenever an input component's property changes. Callback function is decorated with `@app.callback` decorator.
- Callback decorator function takes two parameters: Input and Output. Input and Output to the callback function will have component id and component property. Multiple inputs or outputs should be enclosed inside either a list or tuple.



Basic Plotly Charts

Estimated time needed: 30 minutes

Objectives

In this lab, you will learn about creating plotly charts using `plotly.graph_objects` and `plotly.express`.

Learn more about:

- [Plotly python](#)
- [Plotly Graph Objects](#)
- [Plotly Express](#)
- Handling data using [Pandas](#)

We will be using the [airline dataset](#) from [Data Asset eXchange](#).

Airline Reporting Carrier On-Time Performance Dataset

The Reporting Carrier On-Time Performance Dataset contains information on approximately 200 million domestic US flights reported to the United States Bureau of Transportation Statistics. The dataset contains basic information about each flight (such as date, time, departure airport, arrival airport) and, if applicable, the amount of time the flight was delayed and information about the reason for the delay. This dataset can be used to predict the likelihood of a flight arriving on time.

Preview data, dataset metadata, and data glossary [here](#).

```
In [1]: # Import required libraries
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

Read Data

```
In [2]: # Read the airline data into pandas dataframe
airline_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/airline_data.csv',
                           encoding = "ISO-8859-1",
                           dtype={'Div1Airport': str, 'Div1TailNum': str,
                                  'Div2Airport': str, 'Div2TailNum': str})
```

```
In [3]: # Preview the first 5 lines of the Loaded data
airline_data.head()
```

Out[3]:

	Unnamed: 0	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_Reportin
0	1295781	1998	2	4	2	4	1998-04-02	AS	19930
1	1125375	2013	2	5	13	1	2013-05-13	EV	20366
2	118824	1993	3	9	25	6	1993-09-25	UA	19977
3	634825	1994	4	11	12	6	1994-11-12	HP	19991
4	1888125	2017	3	8	17	4	2017-08-17	UA	19977

5 rows × 110 columns

DOT_ID_Reportin	IATA_CODE_Reportin	...	Div4WheelsOff	Div4TailNum	Div5Airport	Div5AirportID
19930	AS	...	NaN	NaN	NaN	NaN
20366	EV	...	NaN	NaN	NaN	NaN
19977	UA	...	NaN	NaN	NaN	NaN
19991	HP	...	NaN	NaN	NaN	NaN
19977	UA	...	NaN	NaN	NaN	NaN

Div5AirportSeqID	Div5WheelsOn	Div5TotalGTime	Div5LongestGTime	Div5WheelsOff	Div5TailNum
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

```
In [4]: # Shape of the data
airline_data.shape

Out[4]: (27000, 110)

In [5]: # Randomly sample 500 data points. Setting the random state to be 42 so that we get same result.
data = airline_data.sample(n=500, random_state=42)

In [6]: # Get the shape of the trimmed data
data.shape

Out[6]: (500, 110)
```

Lab structure

plotly.graph_objects

1. Review scatter plot creation

Theme: How departure time changes with respect to airport distance

2. **To do** - Create line plot

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

plotly.express

1. Review bar chart creation

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

2. **To do** - Create bubble chart

Theme: Get number of flights as per reporting airline

3. **To do** - Create histogram

Theme: Get distribution of arrival delay

4. Review pie chart

Theme: Proportion of distance group by month (month indicated by numbers)

5. **To do** - Create sunburst chart

Theme: Hierarchical view in othe order of month and destination state holding value of number of flights

plotly.graph_objects

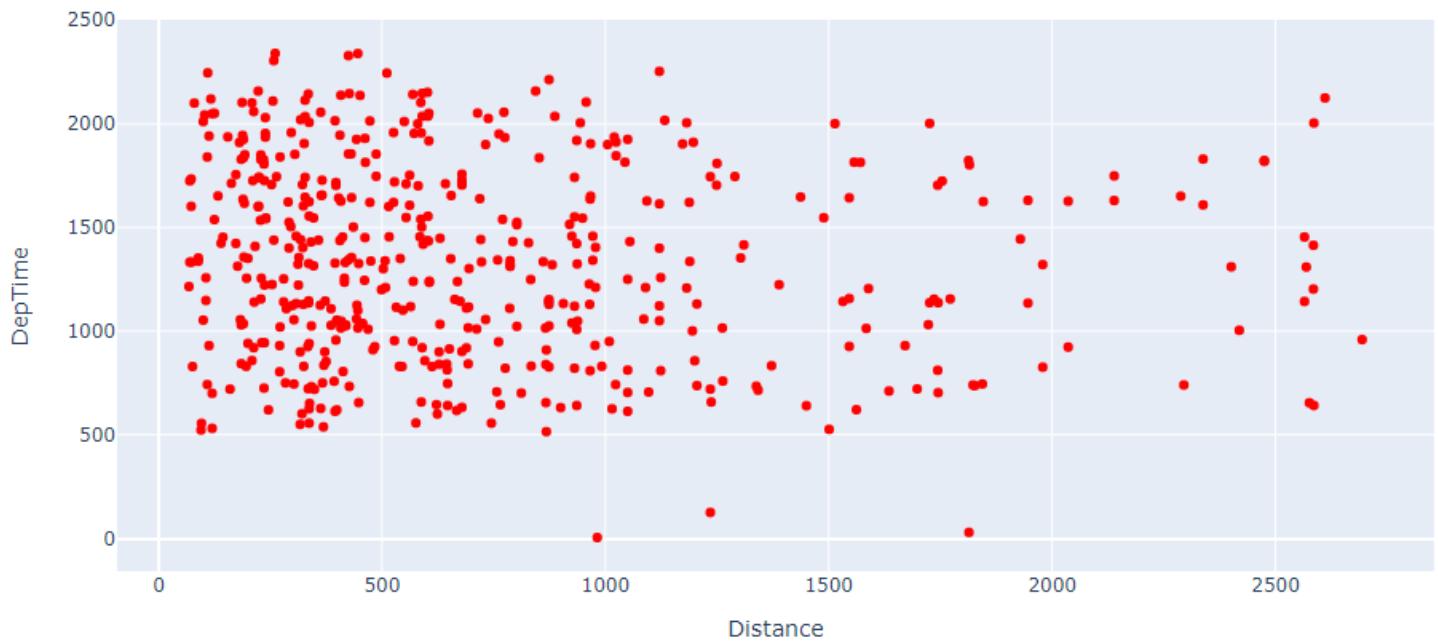
1. Scatter Plot

Learn more about usage of scatter plot [here](#)

Idea: How departure time changes with respect to airport distance

```
In [7]: # First we create a figure using go.Figure and adding trace to it through go.scatter
fig = go.Figure(data=go.Scatter(x=data['Distance'], y=data['DepTime'], mode='markers', marker=dict(color='red')))
# Updating layout through `update_layout`. Here we are adding title to the plot and providing title to x and y axis.
fig.update_layout(title='Distance vs Departure Time', xaxis_title='Distance', yaxis_title='DepTime')
# Display the figure
fig.show()
```

Distance vs Departure Time



2. Line Plot

Learn more about line plot [here](#)

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

```
In [8]: # Group the data by Month and compute average over arrival delay time.  
line_data = data.groupby('Month')['ArrDelay'].mean().reset_index()
```

```
In [9]: # Display the data  
line_data
```

Out[9]:

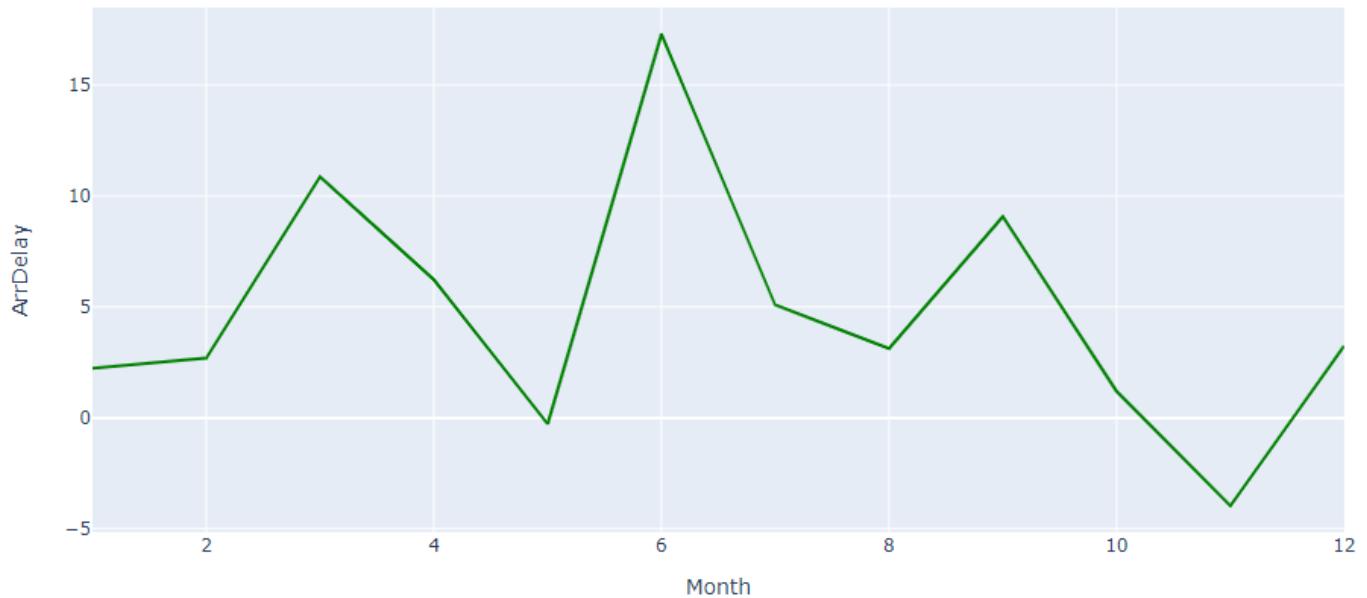
	Month	ArrDelay
0	1	2.232558
1	2	2.687500
2	3	10.868421
3	4	6.229167
4	5	-0.279070
5	6	17.310345
6	7	5.088889
7	8	3.121951
8	9	9.081081
9	10	1.200000
10	11	-3.975000
11	12	3.240741

To do:

- Create a line plot with x-axis being the month and y-axis being computed average delay time. Update plot title, xaxis, and yaxis title.
- Hint: Scatter and line plot vary by updating mode parameter.

```
In [13]: fig = go.Figure(data=go.Scatter(x=line_data['Month'], y=line_data['ArrDelay'], mode='lines', marker=dict(color='green'))  
fig.update_layout(title='Month vs Average Flight Delay Time', xaxis_title='Month', yaxis_title='ArrDelay')  
fig.show()
```

Month vs Average Flight Delay Time



plotly.express

1. Bar Chart

Learn more about bar chart [here](#)

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

```
In [17]: # Group the data by destination state and reporting airline. Compute total number of flights in each combination  
bar_data = data.groupby('DestState')['Flights'].sum().reset_index()
```

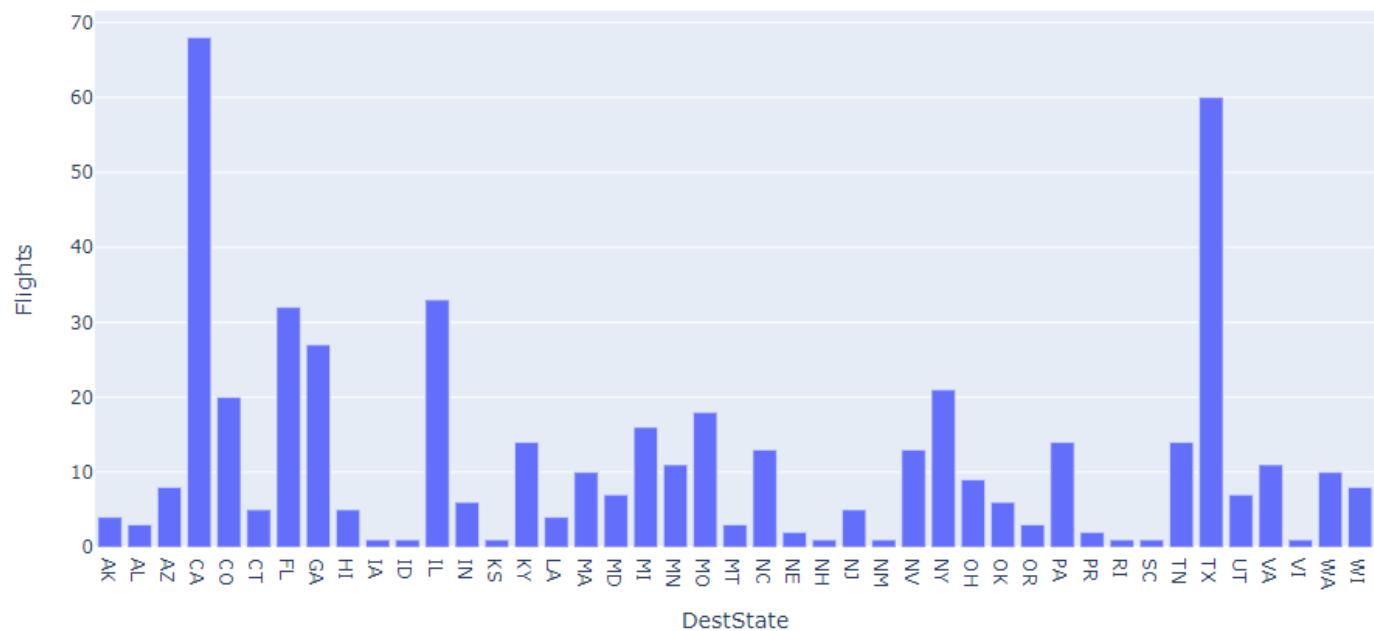
```
In [41]: # Display the data  
bar_data
```

Out[41]:

	DestState	Flights
0	AK	4.0
1	AL	3.0
2	AZ	8.0
3	CA	68.0
4	CO	20.0
5	CT	5.0
6	FL	32.0
7	GA	27.0
8	HI	5.0
9	IA	1.0
10	ID	1.0
11	IL	33.0
12	IN	6.0
13	KS	1.0
14	KY	14.0
15	LA	4.0
16	MA	10.0
17	MD	7.0
18	MI	16.0
19	MN	11.0
20	MO	18.0
21	MT	3.0
22	NC	13.0
23	NE	2.0
24	NH	1.0
25	NJ	5.0
26	NM	1.0
27	NV	13.0
28	NY	21.0
29	OH	9.0
30	OK	6.0
31	OR	3.0
32	PA	14.0
33	PR	2.0
34	RI	1.0
35	SC	1.0
36	TN	14.0
37	TX	60.0
38	UT	7.0
39	VA	11.0
40	VI	1.0
41	WA	10.0
42	WI	8.0

```
In [19]: # Use plotly express bar chart function px.bar. Provide input data, x and y axis variable, and title of the chart.  
# This will give total number of flights to the destination state.  
fig = px.bar(bar_data, x="DestState", y="Flights", title='Total number of flights to the destination state split by reporting airline')  
fig.show()
```

Total number of flights to the destination state split by reporting airline



2. Bubble Chart

Learn more about bubble chart [here](#)

Idea: Get number of flights as per reporting airline

```
In [20]: # Group the data by reporting airline and get number of flights
bub_data = data.groupby('Reporting_Airline')['Flights'].sum().reset_index()
```

```
In [43]: bub_data
```

Out[43]:

	Reporting_Airline	Flights
0	9E	5.0
1	AA	57.0
2	AS	14.0
3	B6	10.0
4	CO	12.0
5	DL	66.0
6	EA	4.0
7	EV	11.0
8	F9	4.0
9	FL	3.0
10	HA	3.0
11	HP	7.0
12	KH	1.0
13	MQ	27.0
14	NK	3.0
15	NW	26.0
16	OH	8.0
17	OO	28.0
18	PA (1)	1.0

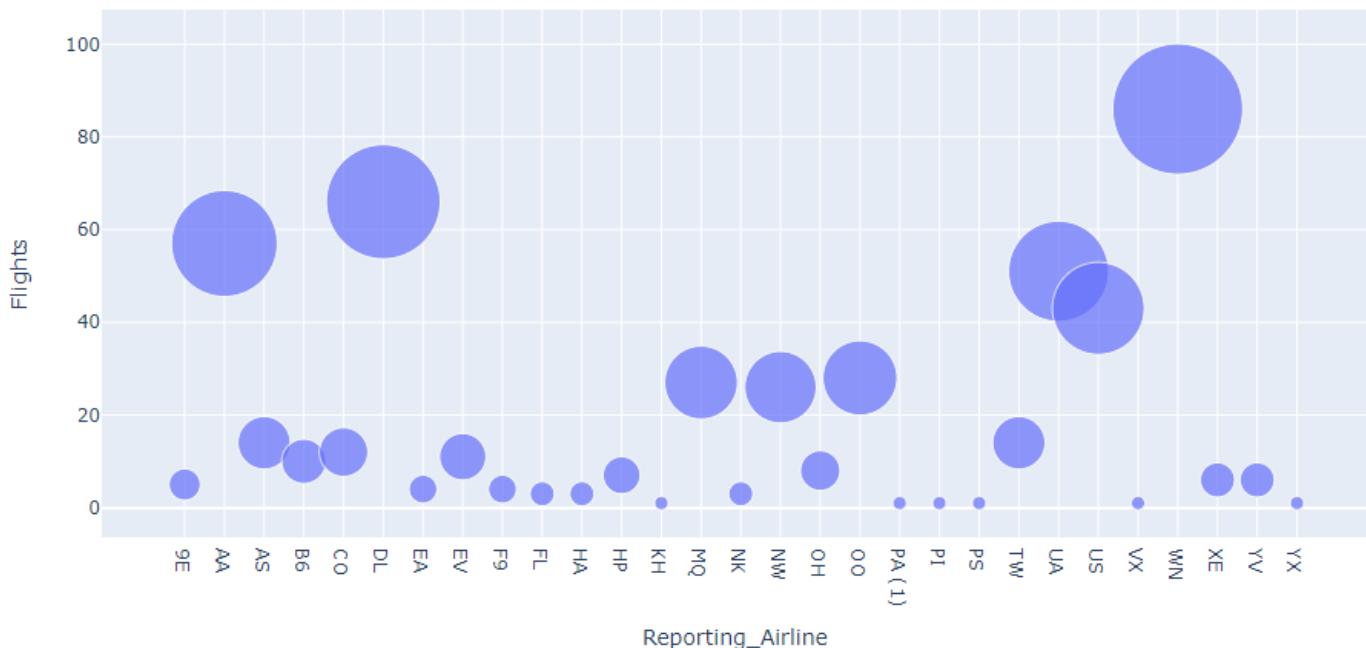
19	PI	1.0
20	PS	1.0
21	TW	14.0
22	UA	51.0
23	US	43.0
24	VX	1.0
25	WN	86.0
26	XE	6.0
27	YV	6.0
28	YX	1.0

To do

- Create a bubble chart using the bub_data with x-axis being reporting airline and y-axis being flights.
- Provide title to the chart
- Update size of the bubble based on the number of flights. Use size parameter.
- Update name of the hover tooltip to reporting_airline using hover_name parameter.

```
In [23]: fig = px.scatter(bub_data, x='Reporting_Airline', y='Flights', size='Flights',
                      hover_name='Reporting_Airline', title='Reporting Airline vs Number of Flights', size_max=60)
fig.show()
```

Reporting Airline vs Number of Flights



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

Histogram

Learn more about histogram [here](#)

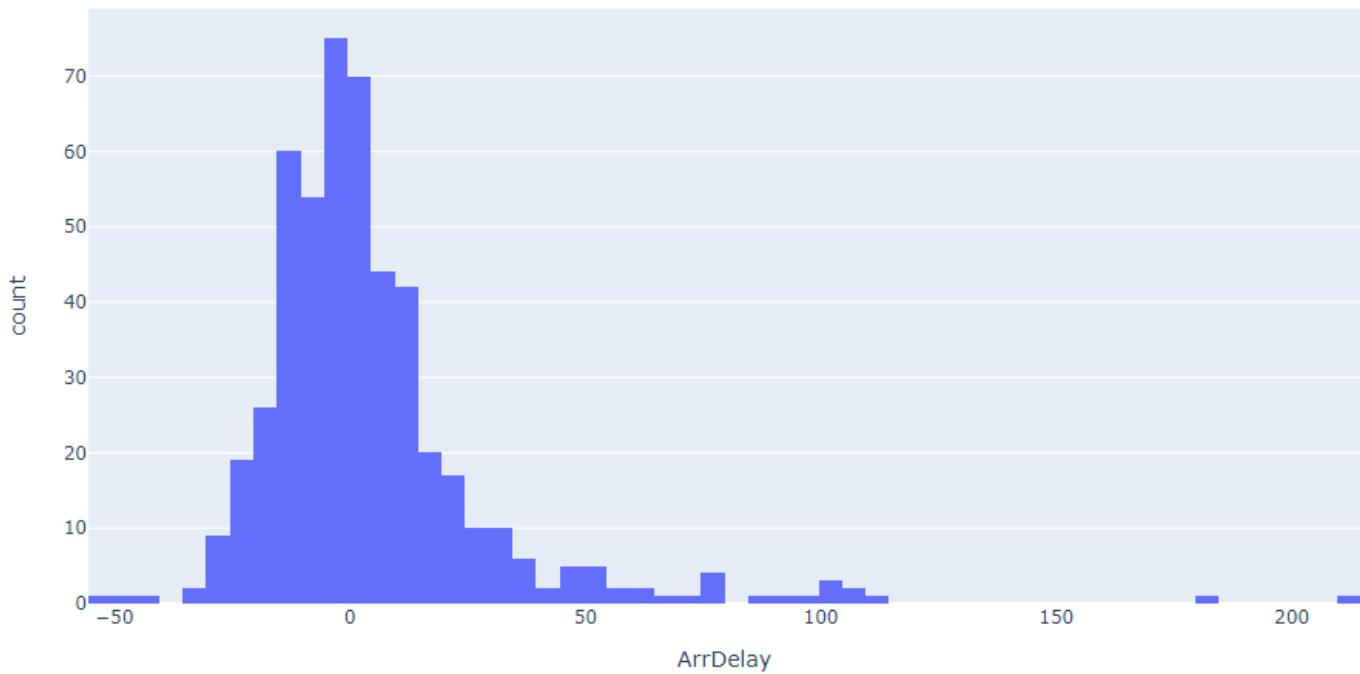
Idea: Get distribution of arrival delay

```
In [24]: # Set missing values to 0  
data['ArrDelay'] = data['ArrDelay'].fillna(0)
```

To do

- Use px.histogram and pass the dataset.
- Pass ArrDelay to x parameter.

```
In [26]: fig= px.histogram(data, x='ArrDelay')  
fig.show()
```



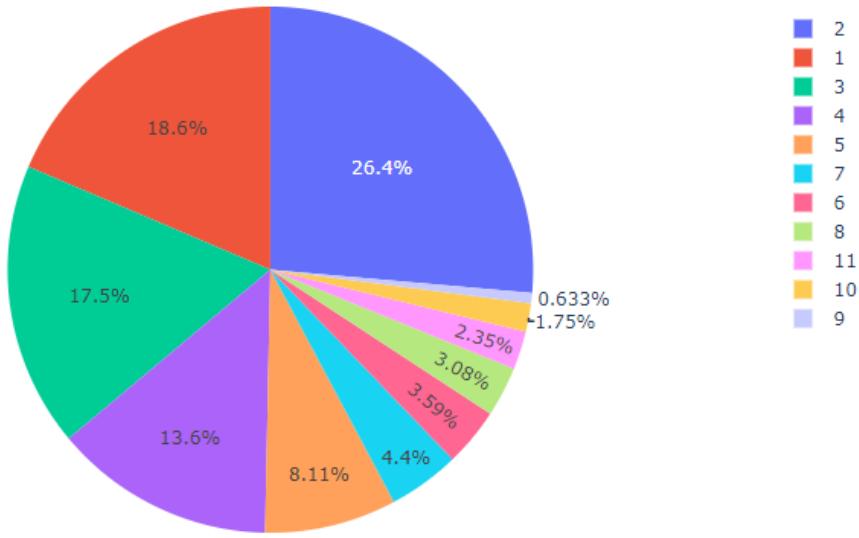
Pie Chart

Learn more about pie chart [here](#)

Idea: Proportion of distance group by month (month indicated by numbers)

```
In [28]: # Use px.pie function to create the chart. Input dataset.  
# Values parameter will set values associated to the sector. 'Month' feature is passed to it.  
# Labels for the sector are passed to the `names` parameter.  
fig = px.pie(data, values='Month', names='DistanceGroup', title='Distance group proportion by month')  
fig.show()
```

Distance group proportion by month



Sunburst Charts

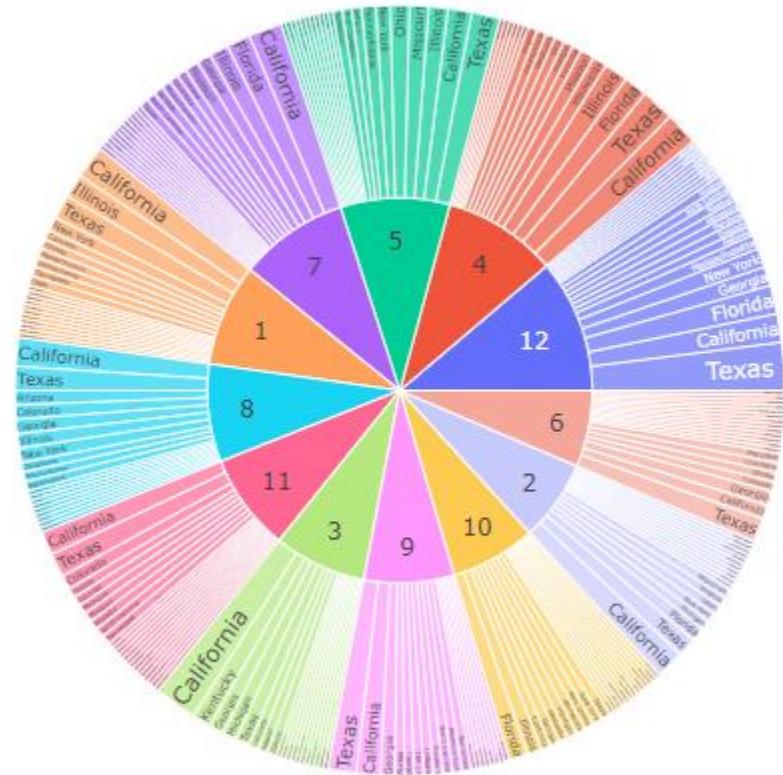
Learn more about sunburst chart [here](#)

Idea: Hierarchical view in othe order of month and destination state holding value of number of flights

To do

- Create sunburst chart using px.sunburst.
- Define hierarchy of sectors from root to leaves in path parameter. Here, we go from Month to DestStateName feature.
- Set sector values in values parameter. Here, we can pass in Flights feature.
- Show the figure.

```
In [29]: fig = px.sunburst(data, path=['Month', 'DestStateName'], values='Flights')
fig.show()
```



Summary

Congratulations for completing your first lab.

In this lab, you have learnt how to use `plotly.graph_objects` and `plotly.express` for creating plots and charts.

Introduction to Dash

Dash is a

- Open-Source User Interface Python library for creating reactive, web-based applications.
- It is enterprise-ready and a first-class member of Plotly's open-source tools.
- Dash applications are web servers running Flask and communicating JSON packets over HTTP requests. Dash's frontend renders components using React.js.
- It is easy to build a Graphical User Interface using dash as it abstracts all technologies required to build the applications. Dash is Declarative and Reactive.
- Dash output can be rendered in web browser and can be deployed to servers. Dash uses a simple reactive decorator for binding code to the UI. This is inherently mobile and cross-platform ready.

Let's say you are planning to create an application to answer a business question.

1. As a first step, you need to determine the layout of the application.
2. Decide which chart to use and where to place for example. This is called 'layout' part in dash.
3. The second part is to add interactivity to the application.

There are two components of Dash

1. First is 'Core components'

We can import core components as dcc using this import statement

2. 'HTML Components'

We can import html components as html using this import statement Let's explore these further.

The `dash_html_components` library has a component for every HTML tag. You can compose your layout using Python structures with the `dash-html-components` library.

The `dash_html_components` library provides classes for all of the HTML tags. The keyword arguments describe the HTML attributes like style, className, and id. No knowledge of HTML or CSS is required but can help in styling the dashboards.

Let's see an example of how to use HTML components.

We start by creating a dash application. From here we create division in our application layout and then adding components to it. In the outer layout division, we first provide a name for our application using the HTML heading component H1. The style parameter is used to change the font color, size and border of the heading.

Next, we add paragraph content to the page using a HTML paragraph component P. Division can be created inside the outer division. Here we are providing division content as 'This is a new division` and styling it using style parameter components. To put all this together, in the application layout create a HTML division and add components. Multiple divisions can be added to the outer application layout.

The dash_core_components describe higher-level components that are interactive and generated with JavaScript, HTML, and CSS through the React.js library. Some example of core components are Creating a slider, input area, check items, and datepicker.

You can explore other components using the reference link provided at the end of the slide. Let's see how to add a slider and dropdown to the application. For the dropdown, we use the dcc.dropdown component. We will create a dropdown list under the options parameter as a dictionary. `Label` will hold the dropdown display label name and `value` will hold the value of the label. We can also provide a default dropdown display label using `value` parameter. For the slider, we use the dcc.slider component and provide min and max value of the slider. The `marks` parameter is used for adding a slide marker and `value` parameter for adding default value.

Introduction to Dash



Dash – An Overview

- Open-source User Interface Python library from Plotly
- Easy to build GUI
- Declarative and Reactive
- Rendered in web browser and can be deployed to servers
- Inherently cross-platform and mobile ready

Dash Components

- Core components

```
import dash_core_components as dcc
```

- HTML Components

```
import dash_html_components as html
```

HTML Components

- Component for every HTML tag
- Keyword arguments describe the HTML attributes like style, className, and id

Example of Creating a Dashboard using HTML component

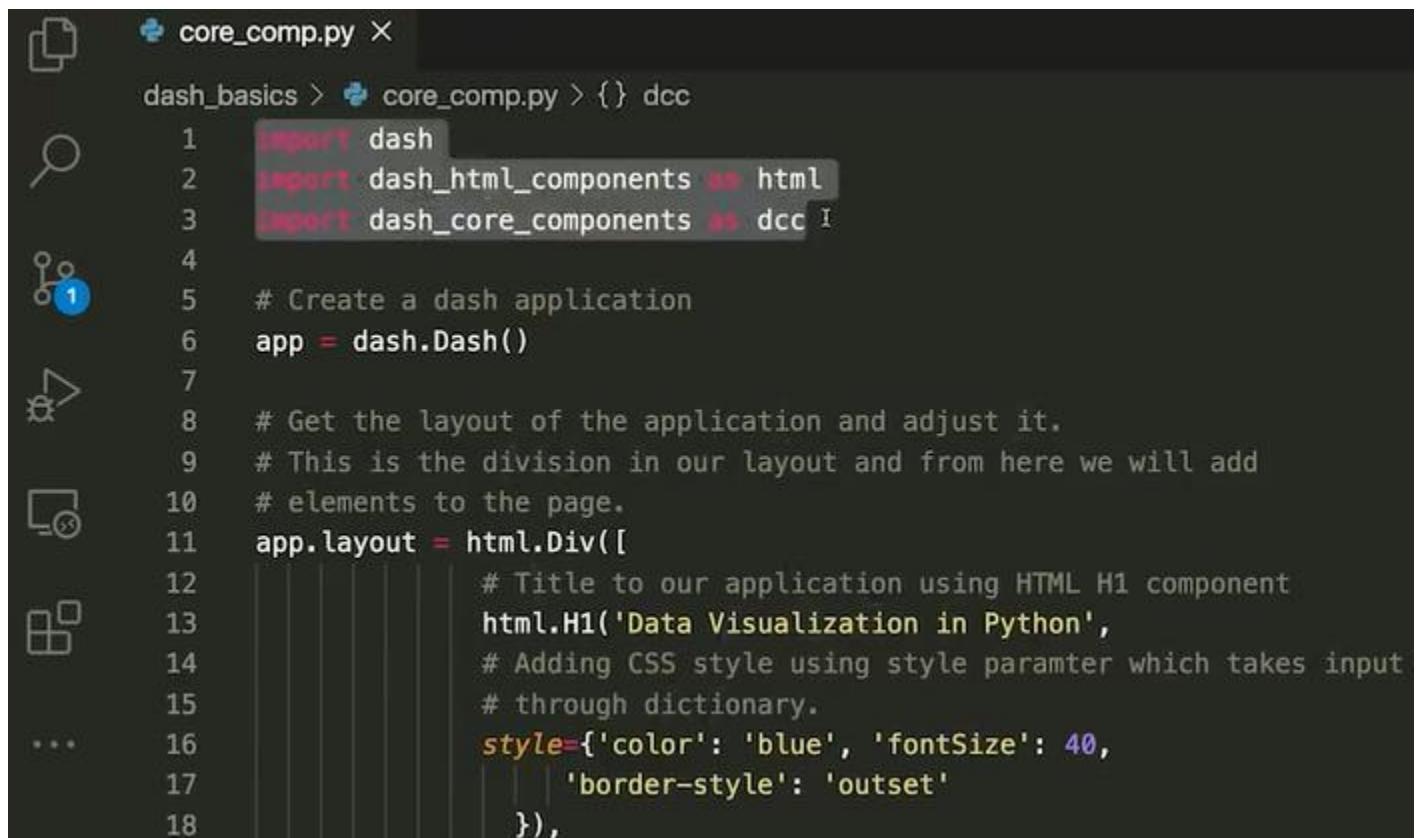
The screenshot shows a Jupyter Notebook interface with a dark theme. On the left, there's a sidebar with various icons for file operations like copy, paste, search, and refresh. The main area displays a Python script named `html_comp.py`. The code uses the `dash` library to create a dashboard application. It includes imports for `dash` and `dash_html_components`, creates a `dash.Dash()` application, and defines its layout using `html.Div` and `html.H1` components. The code also adds a paragraph with specific styling and creates a new division with red text and a double border. At the bottom, the terminal shows the code being run in a Python 3.7.9 64-bit environment, and the status bar indicates the code is at line 1, column 1, with 2 spaces and UTF-8 encoding.

```
html_comp.py
import dash
import dash_html_components as html
# Create a dash application
app = dash.Dash()
# Get the layout of the application and adjust it.
# This is the division in our layout and from here we will add
# elements to the page.
app.layout = html.Div([
    # Title to our application using HTML H1 component
    html.H1('Data Visualization in Python'),
    # Adding CSS style using style parameter which takes input
    # through dictionary.
    style={'color': 'blue', 'fontSize': 40,
           'border-style': 'outset'
    }),
    # Adding paragraph component inside layout division
    html.P('This video is about dash basics',
           style={'fontSize': 30
    }),
    # Create division inside the application layout outer division
    html.Div(['This is a new division'],
            style={'border-style': 'ridge', 'border-color': 'blue'})
]
)
if __name__ == '__main__':
    # Grab the application and run the server
    app.run_server(port=8002, host='127.0.0.1', debug=True)
```

Core Components

- Higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library
- Example: Creating a slider, input area, check items, datepicker and so on

How to add a slicer and a dropdown to the application



The screenshot shows a code editor window with a dark theme. On the left, there are several icons: a file icon, a magnifying glass, a circular icon with a '1', a play button, and a refresh/circular arrow icon. The main area displays a Python script named 'core_comp.py'. The code imports 'dash', 'dash_html_components' as 'html', and 'dash_core_components' as 'dcc'. It creates a Dash application and sets its layout to a single HTML Div containing an H1 title with a specific color and font size.

```
core_comp.py ×

dash_basics > core_comp.py > {} dcc
1 import dash
2 import dash_html_components as html
3 import dash_core_components as dcc
4
5 # Create a dash application
6 app = dash.Dash()
7
8 # Get the layout of the application and adjust it.
9 # This is the division in our layout and from here we will add
10 # elements to the page.
11 app.layout = html.Div([
12     # Title to our application using HTML H1 component
13     html.H1('Data Visualization in Python',
14         # Adding CSS style using style parameter which takes input
15         # through dictionary.
16         style={'color': 'blue', 'fontSize': 40,
17                'border-style': 'outset'
18             }),
```

```

19     # Adding dropdown
20     html.Label('Dropdown'),
21     dcc.Dropdown(
22         options=[
23             {'label': 'Option 1', 'value': '1'},
24             {'label': 'Option 2', 'value': '2'},
25             {'label': 'Option 3', 'value': '3'}
26         ],
27         value='3'
28     ),
29     # Adding Slider
30     dcc.Slider(
31         min=0,
32         max=5,
33         marks={i: '{}'.format(i) for i in range(5)},
34         value=2,
35     )
36     ],
37 )
38
39 if __name__ == '__main__':
40     # Grab the application and run the server
41     app.run_server(port=8002, host='127.0.0.1', debug=True)
42

```

Adding Dropdown

For the dropdown, we use the `dcc.dropdown` component. We will create a dropdown list under the `options` parameter as a dictionary. `Label` will hold the dropdown display label name and `value` will hold the value of the label. We can also provide a default dropdown display label using `value` parameter.

Adding Slider

For the slider, we use the `dcc.slider` component and provide min and max value of the slider. The `marks` parameter is used for adding a slide marker and `value` parameter for adding default value.

Reading: Additional Resources for Dash

 [Bookmark this page](#)

To learn more about Dash, explore:

[Complete dash user guide](#)

[Dash core components](#)

[Dash HTML components](#)

[Dash community forum](#)

[Related blogs](#)

Important! Must go to below to create a dashboard

To learn comprehensively on how to create a dashboard (tutorial) and install dash go to

1.Dash Python User Guide <https://dash.plotly.com/>

What's Dash? Installation Dash Tutorial etc.

2.Dash Core Components <https://dash.plotly.com/dash-core-components>

Dash ships with supercharged components for interactive user interfaces.

The Dash Core Components module (dash.dcc) can be imported and used with from dash import dcc and gives you access to many interactive components, including, dropdowns, checklists, and sliders.

The dcc module is part of Dash and you'll find the source for it in the [Dash GitHub repo](#).

For production Dash apps, the Dash Core Components styling & layout should be managed with Dash Enterprise [Design Kit](#).

3. Dash HTML Components <https://dash.plotly.com/dash-html-components>

Dash is a web application framework that provides pure Python abstraction around HTML, CSS, and JavaScript.

Instead of writing HTML or using an HTML templating engine, you compose your layout using Python with the Dash HTML Components module (dash.html).

The Dash HTML Components module is part of Dash and you'll find the source for it in the [Dash GitHub repo](#).

For production Dash apps, styling and layout of Dash HTML Components should be managed with Dash Enterprise [Design Kit](#).

Here is an example of a simple HTML structure:

8.4.3.Dash basics: HTML and core components

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Objectives

After completing the lab you will be able to:

- Create a dash application layout
- Add HTML H1, P, and Div components
- Add core graph component
- Add multiple charts

Dataset Used

[Airline Reporting Carrier On-Time Performance](#) dataset from [Data Asset eXchange](#)

Let's start creating dash application

Goal

Create a dashboard that displays the percentage of flights running under specific distance group. Distance group is the distance intervals, every 250 miles, for flight segment. If the flight covers to 500 miles, it will be under distance group 2 (250 miles + 250 miles).

Expected Output

Below is the expected result from the lab. Our dashboard application consists of three components:

- Title of the application
- Description of the application
- Chart conveying the proportion of distance group by month

To do:

1. Import required libraries and read the dataset
2. Create an application layout
3. Add title to the dashboard using HTML H1 component
4. Add a paragraph about the chart using HTML P component
5. Add the pie chart above using core graph component

6. Run the app

TASK 1 - Data Preparation

Let's start with

- Importing necessary libraries
- Reading and sampling 500 random data points
- Get the chart ready

Copy the below code to the dash_basics.py script and review the code.

```
# Import required packages
import pandas as pd
import plotly.express as px
import dash
import dash_html_components as html
import dash_core_components as dcc

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

# Randomly sample 500 data points. Setting the random state to be 42 so that we get same result.
data = airline_data.sample(n=500, random_state=42)

# Pie Chart Creation
fig = px.pie(data, values='Flights', names='DistanceGroup', title='Distance group proportion by flights')
```

TASK 2 - Create dash application and get the layout skeleton

Next, we create a skeleton for our dash application. Our dashboard application has three components as seen before:

- Title of the application
- Description of the application
- Chart conveying the proportion of distance group by month

Mapping to the respective Dash HTML tags:

- Title added using html.H1() tag
- Description added using html.P() tag
- Chart added using dcc.Graph() tag

Copy the below code to the dash_basics.py script and review the structure.

NOTE: Copy below the current code

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

# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1 component
# Add description about the graph using HTML P (paragraph) component
# Finally, add graph component.
app.layout = html.Div(children=[html.H1(),
                                html.P(),
                                dcc.Graph(),

                               ])

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

TASK 3 - Add the application title

Update the `html.H1()` tag to hold the application title.

- Application title is Airline Dashboard
- Use style parameter provided below to make the title center aligned, with color code #503D36, and font-size as 40

Update the `html.H1()` tag to hold the application title.

- Application title is `Airline Dashboard`
- Use style parameter provided below to make the title `center` aligned, with color code `#503D36`, and font-size as `40`

```
'Airline Dashboard',
style={'textAlign': 'center', 'color': '#503D36', 'font-size': 40}
```

After updating the `html.H1()` with the application title, the `app.layout` will look like:

```
app.layout = html.Div(children=[html.H1('Airline Dashboard',
                                         style={'textAlign': 'center',
                                                'color': '#503D36',
                                                'font-size': 40}),
                                 html.P(),
                                 dcc.Graph(),
                               ])
```

Update the `html.P()` tag to hold the description of the application.

- Description is `Proportion of distance group (250 mile distance interval group) by flights.`
- Use style parameter to make the description `center` aligned and with color `#F57241`.

```
html.P('Proportion of distance group (250 mile distance interval group) by flights.',  
       style={'textAlign': 'center', 'color': '#F57241'}),
```

After updating the `html.H1()` with the application title, the `app.layout` will look like:

```
app.layout = html.Div(children=[html.H1('Airline Dashboard',  
                                      style={'textAlign': 'center',  
                                             'color': '#503D36',  
                                             'font-size': 40}),  
                           html.P('Proportion of distance group (250 mile distance interval group) by flights.',  
                                  style={'textAlign': 'center', 'color': '#F57241'}),  
                           dcc.Graph(),  
                           ])
```

TASK 5 - Update the graph

Update `figure` parameter of `dcc.Graph()` component to add the pie chart. We have created pie chart and assigned it to `fig`. Let's use that to update the `figure` parameter.

```
dcc.Graph(figure=fig)
```

After updating the `dcc.Graph()` with the application title, the `app.layout` will look like:

```
app.layout = html.Div(children=[html.H1('Airline Dashboard',
                                         style={'textAlign': 'center',
                                                 'color': '#503D36',
                                                 'font-size': 40}),
                                 html.P('Proportion of distance group (250 mile distance interval group) by flights.',
                                       style={'textAlign': 'center', 'color': '#F57241'}),
                                 dcc.Graph(figure=fig),
                               ])
```

TASK 6 - Run the application

- Run the python file using the following command in the terminal

```
python3 dash_basics.py
```

The app will open in a new browser tab like below:

<https://www.coursera.org/professional-certificates/ibm-data-science>

© 2021 Coursera Inc. All rights reserved.

Make Dashboards Interactive

We will see how to connect core and HTML components using callbacks.

- A callback function is a python function that is automatically called by Dash whenever an input component's property changes.
- Callback function is decorated with `decorator`.

So what this decorator tells Dash? Basically, whenever there is a change in the input component value, callback function wrapped by the decorator is called followed by the update to the output component children in the application layout.

Let's look at the callback function skeleton. First, create a function that will perform operations to return the desired result for the output component.

Decorate the callback function with `@app.callback decorator`.

This takes two parameters. **Output** : This sets result returned from the callback function to a component id

Input: This set input provided to the callback function to a component id From here we will connect input and output to desired properties.

We will see this in action with an example using the airline data. The use case here is to extract the top 10 airline carriers in the provided input year selected by the number of flights. Based on the input year, the output will change.

First, we import the required packages. As seen before, we will import pandas, dash, dash core, and HTML components. The new entry here is dash dependencies. From dash dependencies, we will import input and output that we will use in the callback function. We read the airline data into the pandas dataframe. We load our dataframe at the start of the app and can be read inside the callback function.

We will start designing the dash application layout by adding components. First, we will provide the title to the dash app using the HTML heading component H1 and style it using the style parameter. Next, we will add an HTML division and text input Core component. In-Dash, the inputs and outputs of the application are simply the properties of a particular component.

In this example, our input is the "value" property of the component that has the ID "input-yr". By default, the value has 2010. We will update this value in our callback function. Lastly, we will add a division with a graph core component. The core component has `as id`, which we will update inside the callback function. Note the component ids.

We will add a callback decorator . Input to the callback will be the component with id `and property` Output to the callback will be the component with id `and property` Component_id and component_property keywords are optional and are included here for clarity. Next, we will define the callback function The entered year will be the input.

Using the year we extract the required information from data. Finally, the application layout graph is updated. Lastly, we will run the application. This is the output of the code. Our initial input year is 2010. Note that as we update the year, graph is updated for that year.

The second example is a callback with two inputs. It is similar to the one input callback except for a few changes. We will add a division with one more text inputs with the component id input-ab Now we will add the new input with component id to the decorator inside the list.

Next, we will define callback function . This takes the entered year and the entered state as inputs parameters. Computation is performed to extract the information and the application layout is updated with the graph. This is the output of the code. Our initial input year is 2010 and state is AL which is Alabama. As I update the year and state, you can observe that the graph is updated in parallel.

Dash Interactive Components

Dash – Callbacks

Callback function is a python function that are automatically called by Dash whenever an input component's property changes.

Decorator

`@app.callback`

So what this decorator tells Dash? Basically, whenever there is a change in the input component value, callback function wrapped by the decorator is called followed by the update to the output component children in the application layout.

Dash - Callback Function

```
@app.callback(Output, Input)
def callback_function:
    ....
    ....
    ....
    return some_result
```

This takes two parameters. Output : This sets result returned from the callback function to a component id Input: This set input provided to the callback function to a component id From here we will connect input and output to desired properties.

Extract the top 10 airline carriers in the provided input year selected by the number of flights. Based on the input year, the output will change.

Steps:

1.

Callback with one input

```
# Import required packages
import pandas as pd
import plotly.express as px
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
```

First, we import the required packages. As seen before, we will import pandas, dash, dash core, and HTML components. The new entry here is dash dependencies. From dash dependencies, we will import input and output that we will use in the callback function.

Callback with one input

```
# Import required packages
import pandas as pd
import plotly.express as px
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output

# Read the data
airline_data = pd.read_csv('airline_2m.csv',
                           encoding = "ISO-8859-1",
                           dtype={'Div1Airport': str,
                                  'Div1TailNum': str,
                                  'Div2Airport': str,
                                  'Div2TailNum': str})
```

We read the airline data into the pandas dataframe. We load our dataframe at the start of the app and can be read inside the callback function.

Callback with one input

```
app = dash.Dash()
# Design dash app layout
app.layout = html.Div(children=[ html.H1('Airline Dashboard',
    style={'textAlign': 'center', 'color': colors['text'],
    'font-size': 40}),
    html.Div(["Input: ", dcc.Input(id='input-yr', value='2010',
    type='number', style={'height':'50px', 'font-size': 35}),],
    style={'font-size': 40}),
    html.Br(),
    html.Br(),
    html.Div(dcc.Graph(id='bar-plot')),
])

```

We will start designing the dash application layout by adding components. First, we will provide the title to the dash app using the HTML heading component H1 and style it using the style parameter. Next, we will add an HTML division and text input Core component. In-Dash, the inputs and outputs of the application are simply the properties of a particular component.

In this example, our input is the "value" property of the component that has the ID "input-yr". By default, the value has 2010. We will update this value in our callback function. Lastly, we will add a division with a graph core component. The core component has as id, which we will update inside the callback function. Note the component ids.

Callback with one input

```
@app.callback( Output(component_id='bar-plot', component_property='figure'),
                Input(component_id='input-yr', component_property='value'))

def get_graph(entered_year):
    # Select data
    df = airline_data[airline_data['Year']==int(entered_year)]
    # Top 10 airline carrier in terms of number of flights
    g1 = df.groupby(['Reporting_Airline'])['Flights'].sum().nlargest(10).reset_index()
    # Plot the graph
    fig1 = px.bar(g1, x='Reporting_Airline', y='Flights', title='Top 10 airline carrier in
                                                year ' + str(entered_year) + ' in terms of number of flights')
    fig1.update_layout()
    return fig1

if __name__ == '__main__':
    app.run_server(port=8002, host='127.0.0.1', debug=True)
```

We will add a callback decorator . Input to the callback will be the component with id and property Output to the callback will be the component with id and property Component_id and component_property keywords are optional and are included here for clarity. Next, we will define the callback function The entered year will be the input. Using the year we extract the required information from data. Finally, the application layout graph is updated. Lastly, we will run the application.

Callback with one input



This is the output of the code. Our initial input year is 2010. Note that as we update the year, graph is updated for that year.

3.

Callback with two inputs

```
app = dash.Dash()
# Design dash app layout
app.layout = html.Div(children=[ html.H1('Airline Dashboard', style={'textAlign': 'center',
    'color': colors['text'], 'font-size': 40}),
    html.Div(["Year: ", dcc.Input(id='input-yr', value='2010',
        type='number', style={'height':'50px', 'font-size': 35}),
    ], style={'font-size': 40}),
    html.Div(["State Abbreviation: ", dcc.Input(id='input-ab',
        value='AL', type='text', style={'height':'50px',
        'font-size': 35})], style={'font-size': 40}),
    html.Br(),
    html.Br(),
    html.Div(dcc.Graph(id='bar-plot')),
    ])
])
```

The second example is a **callback with two inputs**. It is similar to the one input callback except for a few changes. We will add a division with one more text inputs with the component id **input-ab**

4.

Callback with two inputs

```
@app.callback( Output(component_id='bar-plot', component_property='figure'),
    [ Input(component_id='input-yr', component_property='value'),
      Input(component_id='input-ab', component_property='value')])

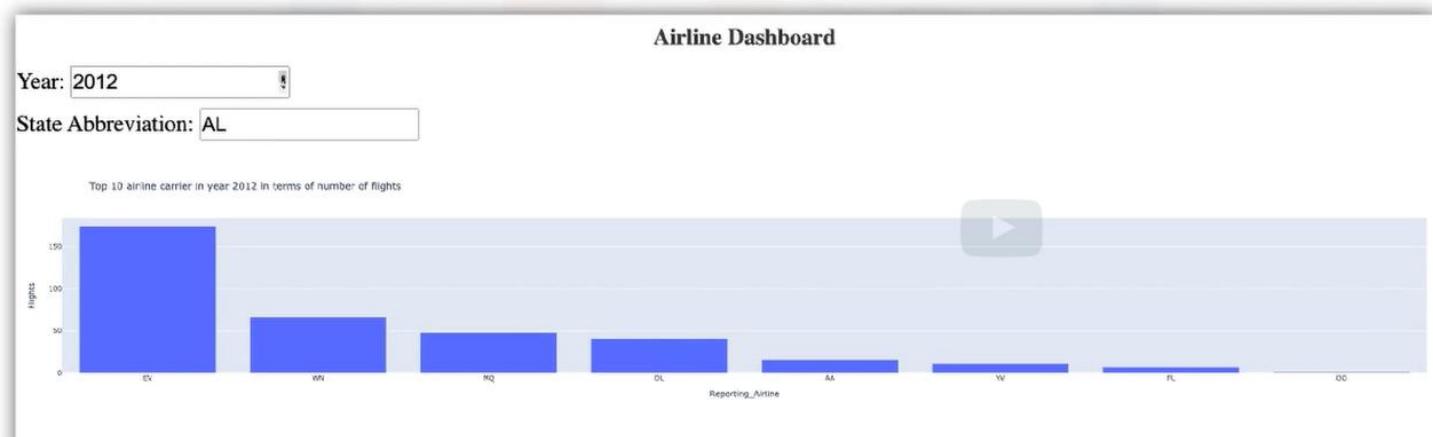
def get_graph(entered_year, entered_state):
    # Select data
    df = airline_data[ (airline_data['Year']==int(entered_year)) &
                      (airline_data['OriginState'] == entered_state)]
    # Top 10 airline carrier in terms of number of flights
    .....
    fig1.update_layout()
    return fig1
if __name__ == '__main__':
    app.run_server(port=8002, host='127.0.0.1', debug=True)
```

with the graph.

Now we will add the new input with component id to the decorator inside the list. Next, we will define callback function. This takes the entered year and the entered state as inputs parameters. Computation is performed to extract the information and the application layout is updated

5.

Callback with two inputs



This is the output of the code. Our initial input year is 2010 and state is AL which is Alabama. As I update the year and state, you can observe that the graph is updated in parallel.

Reading: Additional Resources for Interactive Dashboards

 [Bookmark this page](#)

To learn more about making interactive dashboards in Dash, visit:

[Python decorators reference 1](#)

[Python decorators reference 2](#)

[Callbacks with example](#)

[Dash app gallery](#)

[Dash community components](#)

<https://realpython.com/primer-on-python-decorators/>

Table of Contents

- Functions
 - First-Class Objects
 - Inner Functions
 - Returning Functions From Functions
- Simple Decorators
 - Syntactic Sugar!
 - Reusing Decorators
 - Decorating Functions With Arguments
 - Returning Values From Decorated Functions
 - Who Are You, Really?
- A Few Real World Examples
 - Timing Functions
 - Debugging Code
 - Slowing Down Code
 - Registering Plugins
 - Is the User Logged In?
- Fancy Decorators
 - Decorating Classes
 - Nesting Decorators
 - Decorators With Arguments
 - Both Please, But Never Mind the Bread
 - Stateful Decorators
 - Classes as Decorators
- More Real World Examples
 - Slowing Down Code, Revisited
 - Creating Singletons
 - Caching Return Values
 - Adding Information About Units
 - Validating JSON
- Conclusion
- Further Reading

<https://www.python.org/dev/peps/pep-0318/#current-syntax>

PEP 318 -- Decorators for Functions and Methods

Contents

- [Warning](#)[Warning](#)[Warning](#)
- [Abstract](#)
- [Motivation](#)
 - [Why Is This So Hard?](#)
- [Background](#)
- [On the name 'Decorator'](#)
- [Design Goals](#)
- [Current Syntax](#)
- [Syntax Alternatives](#)
 - [Decorator Location](#)
 - [Syntax forms](#)
 - [Why @?](#)
- [Current Implementation, History](#)
 - [Community Consensus](#)
- [Examples](#)
- [\(No longer\) Open Issues](#)
- [Copyright](#)

<https://dash.plotly.com/basic-callbacks>

Basic Dash Callbacks

This is the 3rd chapter of the [Dash Tutorial](#). The [previous chapter](#) covered the Dash app layout and the [next chapter](#) covers interactive graphing. Just getting started? Make sure to [install the necessary dependencies](#).

In the [previous chapter](#) we learned that `app.layout` describes what the app looks like and is a hierarchical tree of components. The Dash HTML Components (`dash.html`) module provides classes for all of the HTML tags, and the keyword arguments describe the HTML attributes like `style`, `className`, and `id`. The Dash Core Components (`dash.dcc`) module generates higher-level components like controls and graphs.

This chapter describes how to make your Dash apps using *callback functions*: functions that are automatically called by Dash whenever an input component's property changes, in order to update some property in another component (the output).

For optimum user-interaction and chart loading performance, production Dash applications should consider the [Job Queue](#), [HPC](#), [Datashader](#), and [horizontal scaling](#) capabilities of Dash Enterprise.

Let's get started with a simple example of an interactive Dash app.

Simple Interactive Dash App

If you're using Dash Enterprise's [Data Science Workspaces](#), copy & paste the below code into your Workspace ([see video](#)).

```
from dash import Dash, dcc, html, Input, Output

app = Dash( name )

app.layout = html.Div([
    html.H6("Change the value in the text box to see callbacks in action!"),
    html.Div([
        "Input: ",
        dcc.Input(id='my-input', value='initial value', type='text')
    ]),
    html.Br(),
    html.Div(id='my-output'),
])

@app.callback(
    Output(component_id='my-output', component_property='children'),
    Input(component_id='my-input', component_property='value')
)
def update_output_div(input_value):
    return f'Output: {input_value}'

if __name__ == '__main__':
    app.run_server(debug=True)
```

<https://dash.gallery/Portal/>

Dash Enterprise App Gallery

This public instance of the 🏰 Dash Enterprise 🏰 app manager runs >60 Dash apps for 100s of concurrent users on Azure Kubernetes Service. Click on a Dash app's name to below for more information. For the open-source demos, the [Python & R source code](#) can be found on GitHub. For apps using [Design Kit](#) or [Snapshot Engine](#), reach out to [get a demo](#).

[Aerospace](#) | [Automotive](#) | [Energy](#) | [Finance](#) | [Manufacturing](#) | [Medical Imaging](#) |
[Pharma](#) | [Retail](#) | [Sports Analytics](#)

Examples:



<https://plotly.com/dash-community-components/>

Dash Components created by the world's largest open-source community for ML & data science web apps

With >2M Python users, Dash is the most downloaded, trusted framework for building ML & data science web apps.

Plotly maintains a suite of [Core](#), [DAQ](#), [Bio](#), and [HTML components](#) that can be easily downloaded and imported into any Python Dash app. For advanced users, Dash also provides a framework that easily converts [React.js](#) components into Python classes that are compatible with the Dash ecosystem. The Dash community has utilized this compatibility by building incredible components, with utilities ranging from styling to data visualization to server-side caching.

Scroll down to see some Dash components that were recently [shared by the Dash community](#) and/or [listed on PyPI](#). We're always adding to this list. So, if you come across a component that we've missed or if you want to share [your own component](#), contact us!

[CONTACT US](#)

Dash Components

Objectives

- Create a dash application layout
- Add HTML H1, P, and Div components
- Add core graph component
- Add multiple charts

Dataset Used

[Airline Reporting Carrier On-Time Performance](#) dataset from [Data Asset eXchange](#)

Lab Questions

We will be using the same pie and sunburst chart theme from Plotly basics lab.

Theme for Pie Chart

Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).

Theme for Sunburst Chart

Hierarchical view in othe order of month and destination state holding value of number of flights.

```
[1]: # Import required packages
import pandas as pd
import plotly.express as px
```

Load the data

```
[4]: # Read the airline data into pandas dataframe
# Read the airline data into pandas dataframe
airline_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/airline_data.csv',
                           encoding = "ISO-8859-1",
                           dtype={'Div1Airport': str, 'Div1TailNum': str,
                                  'Div2Airport': str, 'Div2TailNum': str})
```

```
[3]: # Preview the first 5 Lines of the Loaded data..
airline_data.head()
```

[3]:

	Unnamed: 0	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	Reporting_Airline	DOT_ID_Reported_Airline	IATA_CODE_Reported_Airline	...
0	1295781	1998	2	4	2	4	1998-04-02	AS	19930	AS	...
1	1125375	2013	2	5	13	1	2013-05-13	EV	20366	EV	...
2	118824	1993	3	9	25	6	1993-09-25	UA	19977	UA	...
3	634825	1994	4	11	12	6	1994-11-12	HP	19991	HP	...
4	1888125	2017	3	8	17	4	2017-08-17	UA	19977	UA	...

5 rows x 110 columns

Div5AirportSeqID	Div5WheelsOn	Div5TotalGTime	Div5LongestGTime	Div5WheelsOff	Div5TailNum
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

[5]: # Shape of the data
airline_data.shape

[5]: (27000, 110)

[6]: # Randomly sample 500 data points. Setting the random state to be 42 so that we get same result.
data = airline_data.sample(n=500, random_state=42)

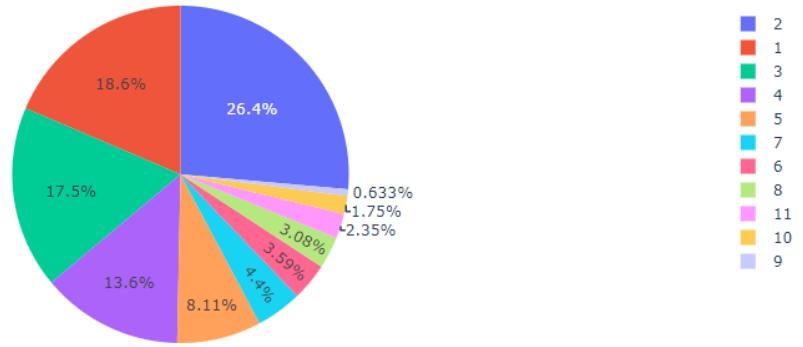
[7]: # Get the shape of the trimmed data
data.shape

[7]: (500, 110)

Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).

[8]: # Pie Chart Creation
fig = px.pie(data, values='Month', names='DistanceGroup', title='Distance group proportion by month')
fig.show()

Distance group proportion by month



Let's start creating dash application

Theme

Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).

To do:

1. Import required libraries and create an application layout
2. Add title to the dashboard using HTML H1 component
3. Add a paragraph about the chart using HTML P component
4. Add the pie chart created above using core graph component
5. Run the app

Hints

General examples can be found [here](#).

- For step 1 (only review), this is very specific to running app from Jupyterlab.
 - For Jupyterlab, we will be using jupyter-dash library. Adding from jupyter_dash import JupyterDash import statement.
 - Instead of creating dash application using app = dash.Dash(), we will be using app = JupyterDash(__name__).
- For step 2,
 - [Plotly H1 HTML Component](#)
 - Title as Airline Performance Dashboard
 - Use style parameter and make the title center aligned, with color code #503D36, and font-size as 40. Check More about HTML section [here](#).

- For step 3,
 - [Plotly Paragraph Component](#)
 - Paragraph as Proportion of distance group (250 mile distance interval group) by month (month indicated by numbers).
 - Use style parameter to make the description center aligned and with color #F57241.
- For step 4, refer [dcc.Graph](#) component usage.
- For step 5, you can refer examples provided [here](#).

NOTE: Run the solution cell multiple times if you are not seeing the result.

App Skeleton

```
import dash

from jupyter_dash import JupyterDash

app = JupyterDash(__name__)
JupyterDash.infer_jupyter_proxy_config()

app.layout = html.Div(children=[html.H1('.....'),
                               html.P('.....'),
                               dcc.Graph('.....')
                           ])
if __name__ == '__main__':
    app.run_server(mode="inline", host="localhost")
```

```
[9]: # Import required libraries
import dash
from dash import dcc
from dash import html
from jupyter_dash import JupyterDash
```

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

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

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

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

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

<!-- The answer is below:

```
# Import required libraries
import dash
import dash_html_components as html
import dash_core_components as dcc
from jupyter_dash import JupyterDash

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

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

```

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

<!-- The answer is below:

# Import required libraries
import dash
import dash_html_components as html
import dash_core_components as dcc
from jupyter_dash import JupyterDash

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

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

```

Congratulations for completing your dash basics lab.

In this lab, you have learnt how to use dash HTML and core components for creating dashboard.

Question 1

1/1 point (graded)

Plotly express is a _____ wrapper

Low-level

High-level



Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

Question 2

1/1 point (graded)

@app_callback is the callback decorator.

True

False



Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

Question 3

1/1 point (graded)

Choose correct way of adding callback decorator:

`@app.callback[Output(component_id='bar-plot', component_property='figure'), Input(component_id='input-yr', component_property='value')]`

`@app.callback(Output(component_id='bar-plot', component_property='figure'), Input(component_id='input-yr', component_property='value'))`

`@app.callback(Output{component_id='bar-plot', component_property='figure'}, Input{component_id='input-yr', component_property='value'})`



Submit

You have used 2 of 2 attempts

✓ Correct (1/1 point)

8.4.4.Add interactivity: user input and callbacks

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [2 revisions](#)

Objectives

After completing the lab you will be able to:

- Work with Dash Callbacks

Dataset Used

Airline Reporting Carrier On-Time Performance dataset from [Data Asset eXchange](#)

Let's start creating dash application

Theme

Extract average monthly arrival delay time and see how it changes over the year. Year range is from 2010 to 2020.

Expected Output

Below is the expected result from the lab. Our dashboard application consists of three components:

- Title of the application
- Component to enter input year
- Chart conveying the average monthly arrival delay

To do:

1. Import required libraries and read the dataset
2. Create an application layout
3. Add title to the dashboard application using HTML H1 component
4. Add an input text box using core input component
5. Add the line chart using core graph component
6. Run the app

Let's start with

- Importing necessary libraries
- Reading the data

Copy the below code to the `dash_interactivity.py` script and review the code.

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

# Read the airline data into pandas dataframe
airline_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DATA010/Datasets/Div1Div2.csv',
                           encoding = "ISO-8859-1",
                           dtype={'Div1Airport': str, 'Div1TailNum': str,
                                  'Div2Airport': str, 'Div2TailNum': str})
```

TASK 2 - Create dash application and get the layout skeleton

Next, we create a skeleton for our dash application. Our dashboard application layout has three components as seen before:

- Title of the application
- Component to enter input year inside a layout division
- Chart conveying the average monthly arrival delay inside a layout division

Mapping to the respective Dash HTML tags:

- Title added using `html.H1()` tag
- Layout division added using `html.Div()` and input component added using `dcc.Input()` tag inside the layout division.
- Layout division added using `html.Div()` and chart added using `dcc.Graph()` tag inside the layout division.

Copy the below code to the `dash_interactivity.py` script and review the structure.

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

# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1 component
# Add a html.Div and core input text component
# Finally, add graph component.
app.layout = html.Div(children=[html.H1(),
                                html.Div(["Input Year", dcc.Input(),],
                                         style={}),
                                html.Br(),
                                html.Br(),
                                html.Div(),
                                ])
```

TASK 3 - Update layout components

Application title

- Heading reference: [Plotly H1 HTML Component](#)
- Title as `Airline Performance Dashboard`
- Use `style` parameter and make the title `center` aligned, with color code `#503D36`, and font-size as `40`. Check `More about HTML` section [here](#).

Input component

- Update `dcc.Input` component `id` as `input-year`, default `value` as `2010`, and `type` as `number`. Use `style` parameter and assign height of the input box to be `50px` and font-size to be `35`.
- Use `style` parameter and assign font-size as `40` for the whole division.

Output component

- Add `dcc.Graph()` component to the second division.
- Update `dcc.Graph` component `id` as `line-plot`.

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

# Get the layout of the application and adjust it.
# Create an outer division using html.Div and add title to the dashboard using html.H1 component
# Add a html.Div and core input text component
# Finally, add graph component.
app.layout = html.Div(children=[html.H1('Airline Performance Dashboard',
                                         style={'textAlign': 'Center', 'color': '#503D36', 'font-size': 40}),
                                 html.Div(["Input Year: ", dcc.Input(id='input-year', value='2010',
                                         type='number', style={'height':'50px', 'font-size':35}),],
                                         style={'font-size': 40}),
                                 html.Br(),
                                 html.Br(),
                                 html.Div(dcc.Graph(id='line-plot')),
                             ])
```

TASK 4 - Add the application callback function

The core idea of this application is to get year as user input and update the dashboard in real-time. We will be using `callback` function for the same.

Steps:

- Define the callback decorator
- Define the callback function that uses the input provided to perform the computation
- Create graph and return it as an output
- Run the application

Copy the below code to the `dash_interactivity.py` script and review the structure.

```
# add callback decorator
@app.callback(Output(),
              Input())

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

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

    #
    fig = go.Figure(data=)
    fig.update_layout()
    return fig

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

TASK 5 - Update the callback function

Callback decorator

- Refer examples provided [here](#)
- Update output component id parameter with the id provided in the `dcc.Graph()` component and component property as `figure`.
- Update input component id parameter with the id provided in the `dcc.Input()` component and component property as `value`.

Callback function

- Update `data` parameter of the `go.Figure()` with the scatter plot. Refer [here](#). Sample syntax below:

```
go.Scatter(x='----',y='----',mode='----',marker='----')
```

- Update `x` as `line_data['Month']`, `y` as `line_data['ArrDelay']`, `mode` as `lines`, and `marker` as `dict(color='green')`.
- Update `fig.update_layout` with title, `xaxis_title`, and `yaxis_title` parameters.
 - Title as `Month vs Average Flight Delay Time`
 - `xaxis_title` as `Month`
 - `yaxis_title` as `ArrDelay`

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

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

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

    fig = go.Figure(data=go.Scatter(x=line_data['Month'], y=line_data['ArrDelay'], mode='lines', marker=dict(color='green')))
    fig.update_layout(title='Month vs Average Flight Delay Time', xaxis_title='Month', yaxis_title='ArrDelay')
    return fig

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

= go.Figure(data=go.Scatter(x=line_data['Month'], y=line_data['ArrDelay'], mode='lines', marker=dict(color='green')))
.update_layout(title='Month vs Average Flight Delay Time', xaxis_title='Month', yaxis_title='ArrDelay')
urn fig
```

TASK 6 - Run the application

8.4.5.Flight Delay Time Statistics Dashboard

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Objectives

After completing the lab you will be able to:

- Know how to add multiple graphs to the dashboard
- Work with Dash Callbacks to handle multiple outputs

Dataset Used

[Airline Reporting Carrier On-Time Performance](#) dataset from [Data Asset eXchange](#)

Let's start creating dash application

Theme

Analyze flight delays in a dashboard.

Dashboard Components

- Monthly average carrier delay by reporting airline for the given year.
- Monthly average weather delay by reporting airline for the given year.
- Monthly average national air system delay by reporting airline for the given year.
- Monthly average security delay by reporting airline for the given year.
- Monthly average late aircraft delay by reporting airline for the given year.

NOTE: Year range should be between 2010 and 2020

Expected Output

Below is the expected result from the lab. Our dashboard application consists of three components:

- Title of the application
- Component to enter input year
- 5 Charts conveying the different types of flight delay. Chart section is divided into three segments.
 - Carrier and Weather delay in the first segment
 - National air system and Security delay in the second segment
 - Late aircraft delay in the third segment

To do:

- Design layout for the application.
- Create a callback function. Add callback decorator, define inputs and outputs.
- Review the helper function that performs computation on the provided inputs.
- Create 5 line graphs.
- Run the application.

TASK 1 - Read the data

Let's start with

- Importing necessary libraries
- Reading the data

Copy the below code to the `flight_delay.py` script and review the code.

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

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

TASK 2 - Create dash application and get the layout skeleton

Next, we create a skeleton for our dash application. Our dashboard application layout has three components as seen before:

- Title of the application
- Component to enter input year inside a layout division
- 5 Charts conveying the different types of flight delay

Mapping to the respective Dash HTML tags:

- Title added using `html.H1()` tag
- Layout division added using `html.Div()` and input component added using `dcc.Input()` tag inside the layout division.
- 5 charts split into three segments. Each segment has a layout division added using `html.Div()` and chart added using `dcc.Graph()` tag inside the layout division.

Copy the below code to the `flight_delay.py` script and review the structure.

```

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

# Build dash app layout
app.layout = html.Div(children=[ html.H1(),
                                  html.Div(["Input Year: ", dcc.Input()],
                                           style={'font-size': 30}),
                                  html.Br(),
                                  html.Br(),
                                  html.Div([
                                      html.Div(),
                                      html.Div()
                                  ], style={'display': 'flex'}),

                                  html.Div([
                                      html.Div(),
                                      html.Div()
                                  ], style={'display': 'flex'}),

                                  html.Div(, style={'width':'65%'})
                               ])

```

NOTE: We are using display as `flex` for two outer divisions to get graphs side by side in a row.

TASK 3 - Update layout components

Application title

- Title as `Flight Delay Time Statistics`, align text as `center`, color as `#503D36`, and font size as `30`.

Input component

- Update `dcc.Input` component `id` as `input-year`, default `value` as `2010`, and `type` as `number`. Use `style` parameter and assign height of the input box to be `35px` and font-size to be `30`.

Output component - Segment 1

Segment 1 is the first `html.Div()`. We have two inner division where first two graphs will be placed.

Skeleton

```
html.Div([
    html.Div(),
    html.Div()
], style={'display': 'flex'}),
```

First inner division

- Add `dcc.Graph()` component.
- Update `dcc.Graph` component `id` as `carrier-plot`.

Second inner division

- Add `dcc.Graph()` component.
- Update `dcc.Graph` component `id` as `weather-plot`.

Output component - Segment 2

Segment 2 is the second `html.Div()`. We have two inner division where the next two graphs will be placed.

Skeleton

```
html.Div([
    html.Div(),
    html.Div()
], style={'display': 'flex'}),
```

First inner division

- Add `dcc.Graph()` component.
- Update `dcc.Graph` component `id` as `nas-plot`.

Second inner division

- Add `dcc.Graph()` component.
- Update `dcc.Graph` component `id` as `security-plot`.

Output component - Segment 3

Segment 3 is the last `html.Div()`.

Skeleton

```
html.Div(, style={'width': '65%'})
```

- Add `dcc.Graph()` component to the first inner division.
- Update `dcc.Graph` component `id` as `late-plot`.

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

# Build dash app layout
app.layout = html.Div(children=[ html.H1('Flight Delay Time Statistics',
                                             style={'textAlign': 'center', 'color': '#503D36',
                                             'font-size': 30}),
                                   html.Div(["Input Year: ", dcc.Input(id='input-year', value='2010',
                                             type='number', style={'height':'35px', 'font-size': 30}),],
                                             style={'font-size': 30}),
                                   html.Br(),
                                   html.Br(),
                                   # Segment 1
                                   html.Div([
                                       html.Div(dcc.Graph(id='carrier-plot')),
                                       html.Div(dcc.Graph(id='weather-plot'))
                                   ], style={'display': 'flex'}),
                                   # Segment 2
                                   html.Div([
                                       html.Div(dcc.Graph(id='nas-plot')),
                                       html.Div(dcc.Graph(id='security-plot'))
                                   ], style={'display': 'flex'}),
                                   # Segment 3
                                   html.Div(dcc.Graph(id='late-plot'), style={'width': '65%'})
                               ])
```

```

"""
Compute_info function description

This function takes in airline data and selected year as an input and performs computation for creating charts and plots.

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

Returns:
    Computed average dataframes for carrier delay, weather delay, NAS delay, security delay, and late aircraft delay.

"""

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

```

TASK 5 - Add the application callback function

The core idea of this application is to get year as user input and update the dashboard in real-time. We will be using `callback` function for the same.

Steps:

- Define the callback decorator
- Define the callback function that uses the input provided to perform the computation
- Create graph and return it as an output
- Run the application

Copy the below code to the `flight_delay.py` script and review the structure.

NOTE: Copy below the current code

```

# Callback decorator
@app.callback( [
    Output(component_id='carrier-plot', component_property='figure'),
    ...
    ...
    ...
    ...
    ],
    Input(...))

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

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

    # Line plot for carrier delay
    carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')
    # Line plot for weather delay
    weather_fig = -----
    # Line plot for nas delay
    nas_fig = -----
    # Line plot for security delay
    sec_fig = -----
    # Line plot for late aircraft delay
    late_fig = -----

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

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

.car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')

```

TASK 6 - Update the callback function

Callback decorator

- Refer examples provided [here](#)
- We have 5 output components added in a list. Update output component id parameter with the ids provided in the `dcc.Graph()` component and set the component property as `figure`. One sample has been added to the skeleton.
- Update input component id parameter with the id provided in the `dcc.Input()` component and component property as `value`.

Callback function

Next is to update the `get_graph` function. We have already added a function `compute_info` that will perform computation on the data using the input.

Mapping the returned value from the function `compute_info` to graph:

- `avg_car` - input for carrier delay
- `avg_weather` - input for weather delay
- `avg_NAS` - input for NAS delay
- `avg_sec` - input for security delay
- `avg_late` - input for late aircraft delay

Code has been provided for plotting carrier delay. Follow the same process and use the above mapping to get plots for other 4 delays.

Refer [here](#) to know how your python code should look like.

```

"""Callback Function
Function that returns fugures using the provided input year.

Arguments:
    entered_year: Input year provided by the user.

Returns:
    List of figures computed using the provided helper function `compute_info` .

"""

# Callback decorator
@app.callback( [
    Output(component_id='carrier-plot', component_property='figure'),
    Output(component_id='weather-plot', component_property='figure'),
    Output(component_id='nas-plot', component_property='figure'),
    Output(component_id='security-plot', component_property='figure'),
    Output(component_id='late-plot', component_property='figure')
],
    Input(component_id='input-year', component_property='value'))
# Computation to callback function and return graph
def get_graph(entered_year):

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

    # Line plot for carrier delay
    carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay')
    # Line plot for weather delay
    weather_fig = px.line(avg_weather, x='Month', y='WeatherDelay', color='Reporting_Airline', title='Average weather delay')
    # Line plot for nas delay
    nas_fig = px.line(avg_NAS, x='Month', y='NASDelay', color='Reporting_Airline', title='Average NAS delay time (minutes)')
    # Line plot for security delay
    sec_fig = px.line(avg_sec, x='Month', y='SecurityDelay', color='Reporting_Airline', title='Average security delay')
    # Line plot for late aircraft delay
    late_fig = px.line(avg_late, x='Month', y='LateAircraftDelay', color='Reporting_Airline', title='Average late aircraft delay')

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

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

```

```
creating graph from the data
_sec, avg_late = compute_info(airline_data, entered_year)

Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')

x='Month', y='WeatherDelay', color='Reporting_Airline', title='Average weather delay time (minutes) by airline')

n', y='NASDelay', color='Reporting_Airline', title='Average NAS delay time (minutes) by airline')

h', y='SecurityDelay', color='Reporting_Airline', title='Average security delay time (minutes) by airline')

y
nth', y='LateAircraftDelay', color='Reporting_Airline', title='Average late aircraft delay time (minutes) by airline')

as_fig, sec_fig, late_fig]
```

TASK 7 - Run the application

Reading: Summary

- Best dashboards answer critical business questions. It will help businesses make informed decisions, thereby improving performance.
- Dashboards can produce real-time visuals.
- Plotly is an interactive, open-source plotting library that supports over 40 chart types.
- The web-based visualizations created using Plotly python can be displayed in Jupyter notebook, saved to standalone HTML files, or served as part of pure Python-built web applications using Dash.
- Plotly Graph Objects is the low-level interface to figures, traces, and layout whereas Plotly express is a high-level wrapper for Plotly.
- Dash is an Open-Source User Interface Python library for creating reactive, web-based applications. It is both enterprise-ready and a first-class member of Plotly's open-source tools.
- Core and HTML are the two components of Dash.
- The dash_html_components library has a component for every HTML tag.
- The dash_core_components describe higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library.
- A callback function is a python function that is automatically called by Dash whenever an input component's property changes. Callback function is decorated with `app.callback` decorator.
- Callback decorator function takes two parameters: Input and Output. Input and Output to the callback function will have component id and component property. Multiple inputs or outputs should be enclosed inside either a list or tuple.

8.5.Final Project & Exam

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [1 revision](#)

Final Assignment

Code

<https://github.com/brendensong/IBM-Data-Science-Professional-Certificate/blob/main/8%20Data%20Visualization%20with%20Python/5%20Peer%20Graded%20Assignment%20Questions.py>

225 lines (188 sloc) | 12.3 KB

Raw Blame

```
1 # Import required libraries
2 import pandas as pd
3 import dash
4 #import dash_html_components as html
5 #import dash_core_components as dcc
6 from dash import html
7 from dash import dcc
8 from dash.dependencies import Input, Output, State
9 import plotly.graph_objects as go
10 import plotly.express as px
11 from dash import no_update
12
13
14 # Create a dash application
15 app = dash.Dash(__name__)
16
17 # REVIEW1: Clear the layout and do not display exception till callback gets executed
18 app.config.suppress_callback_exceptions = True
19
20 # Read the airline data into pandas dataframe
21 airline_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/airline_data.csv')
22 encoding = "ISO-8859-1",
23 dtype={'Div1Airport': str, 'Div1TailNum': str,
24 'Div2Airport': str, 'Div2TailNum': str})
25
26
27 # List of years
28 year_list = [i for i in range(2005, 2021, 1)]
29
30 """Compute graph data for creating yearly airline performance report
```

```

31
32 Function that takes airline data as input and create 5 dataframes based on the grouping condition to be used for plotting charts and graphs.
33
34 Argument:
35
36     df: Filtered dataframe
37
38 Returns:
39     Dataframes to create graph.
40 """
41 def compute_data_choice_1(df):
42     # Cancellation Category Count
43     bar_data = df.groupby(['Month','CancellationCode'])['Flights'].sum().reset_index()
44     # Average flight time by reporting airline
45     line_data = df.groupby(['Month','Reporting_Airline'])['AirTime'].mean().reset_index()
46     # Diverted Airport Landings
47     div_data = df[df['DivAirportLandings'] != 0.0]
48     # Source state count
49     map_data = df.groupby(['OriginState'])['Flights'].sum().reset_index()
50     # Destination state count
51     tree_data = df.groupby(['DestState', 'Reporting_Airline'])['Flights'].sum().reset_index()
52     return bar_data, line_data, div_data, map_data, tree_data
53
54
55 """Compute graph data for creating yearly airline delay report
56
57 This function takes in airline data and selected year as an input and performs computation for creating charts and plots.
58
59 Arguments:
60     df: Input airline data.

```

```

61
62 Returns:
63     Computed average dataframes for carrier delay, weather delay, NAS delay, security delay, and late aircraft delay.
64 """
65 def compute_data_choice_2(df):
66     # Compute delay averages
67     avg_car = df.groupby(['Month','Reporting_Airline'])['CarrierDelay'].mean().reset_index()
68     avg_weather = df.groupby(['Month','Reporting_Airline'])['WeatherDelay'].mean().reset_index()
69     avg_NAS = df.groupby(['Month','Reporting_Airline'])['NASDelay'].mean().reset_index()
70     avg_sec = df.groupby(['Month','Reporting_Airline'])['SecurityDelay'].mean().reset_index()
71     avg_late = df.groupby(['Month','Reporting_Airline'])['LateAircraftDelay'].mean().reset_index()
72     return avg_car, avg_weather, avg_NAS, avg_sec, avg_late
73
74
75 # Application layout
76 app.layout = html.Div(children=[
77     # TASK1: Add title to the dashboard
78     # Enter your code below. Make sure you have correct formatting.
79     html.H1('US Domestic Airline Flights Performance',
80             style={'textAlign': 'center', 'color': '#503D36', 'font-size': 24}),
81     # REVIEW2: Dropdown creation
82     # Create an outer division
83     html.Div([
84         # Add an division
85         html.Div([
86             # Create an division for adding dropdown helper text for report type
87             html.Div(
88                 [
89                     html.H2('Report Type:', style={'margin-right': '2em'}),
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

),
# TASK2: Add a dropdown
# Enter your code below. Make sure you have correct formatting.
dcc.Dropdown(id='input-type',
            options=[
                {'label': 'Yearly Airline Performance Report', 'value': 'OPT1'},
                {'label': 'Yearly Airline Delay Report', 'value': 'OPT2'}
            ],
            placeholder='Select a report type',
            style={'text-align-last': 'center', 'font-size': '20px', 'width': '80%', 'padding': '3px'})
# Place them next to each other using the division style
], style={'display': 'flex'}),

# Add next division
html.Div([
    # Create an division for adding dropdown helper text for choosing year
    html.Div(
        [
            html.H2('Choose Year:', style={'margin-right': '2em'})
        ]
    ),
    dcc.Dropdown(id='input-year',
                # Update dropdown values using list comprehension
                options=[{'label': i, 'value': i} for i in year_list],
                placeholder="Select a year",
                style={'width': '80%', 'padding': '3px', 'font-size': '20px', 'text-align-last': 'center'}),
    # Place them next to each other using the division style
    ], style={'display': 'flex'}),
])

```

```

121         # Add Computed graphs
122         # REVIEW3: Observe how we add an empty division and providing an id that will be updated during callback
123         html.Div([ ], id='plot1'),
124
125         html.Div([
126             html.Div([ ], id='plot2'),
127             html.Div([ ], id='plot3')
128         ], style={'display': 'flex'}),
129
130         # TASK3: Add a division with two empty divisions inside. See above disvision for example.
131         # Enter your code below. Make sure you have correct formatting.
132         #html.Div([ ], id='plot1'),
133
134         html.Div([
135             html.Div([ ], id='plot4'),
136             html.Div([ ], id='plot5')
137         ], style={'display': 'flex'})
138     ])
139
140     # Callback function definition
141     # TASK4: Add 5 ouput components
142     # Enter your code below. Make sure you have correct formatting.
143     @app.callback( [Output(component_id='plot1', component_property='children'),
144                     Output(component_id='plot2', component_property='children'),
145                     Output(component_id='plot3', component_property='children'),
146                     Output(component_id='plot4', component_property='children'),
147                     Output(component_id='plot5', component_property='children')],
148                 [Input(component_id='input-type', component_property='value'),
149                  Input(component_id='input-year', component_property='value')],
150                 # REVIEW4: Holding output state till user enters all the form information. In this case, it will be chart type and year
151                 [State("plot1", "children"), State("plot2", "children"),
152                  State("plot3", "children"), State("plot4", "children"),
153                  State("plot5", "children")]
154             ])
155     # Add computation to callback function and return graph
156     def get_graph(chart, year, children1, children2, c3, c4, c5):
157

```

```

158     # Select data
159     df = airline_data[airline_data['Year']==int(year)]
160
161     if chart == 'OPT1':
162         # Compute required information for creating graph from the data
163         bar_data, line_data, div_data, map_data, tree_data = compute_data_choice_1(df)
164
165         # Number of flights under different cancellation categories
166         bar_fig = px.bar(bar_data, x='Month', y='Flights', color='CancellationCode', title='Monthly Flight Cancellation')
167
168         # TASK5: Average flight time by reporting airline
169         # Enter your code below. Make sure you have correct formatting.
170         line_fig = px.line(line_data, x='Month', y='AirTime', color='Reporting_Airline', title='Average monthly flight time (minutes) by airline')
171
172         # Percentage of diverted airport landings per reporting airline
173         pie_fig = px.pie(div_data, values='Flights', names='Reporting_Airline', title='% of flights by reporting airline')
174
175         # REVIEW5: Number of flights flying from each state using choropleth
176         map_fig = px.choropleth(map_data, # Input data
177             locations='OriginState',
178             color='Flights',
179             hover_data=['OriginState', 'Flights'],
180             locationmode = 'USA-states', # Set to plot as US States
181             color_continuous_scale='GnBu',
182             range_color=[0, map_data['Flights'].max()])
183         map_fig.update_layout(
184             title_text = 'Number of flights from origin state',
185             geo_scope='usa') # Plot only the USA instead of globe
186
187         # TASK6: Number of flights flying to each state from each reporting airline
188         # Enter your code below. Make sure you have correct formatting.
189         tree_fig = px.treemap(tree_data, path=['DestState', 'Reporting_Airline'],
190             values='Flights',
191             color='Flights',
192             color_continuous_scale='RdBu',
193             title='Flight count by airline to destination state'
194             )
195

```

```

195
196
197     # REVIEW6: Return dcc.Graph component to the empty division
198     return [dcc.Graph(figure=tree_fig),
199             dcc.Graph(figure=pie_fig),
200             dcc.Graph(figure=map_fig),
201             dcc.Graph(figure=bar_fig),
202             dcc.Graph(figure=line_fig)
203             ]
204
205 else:
206     # REVIEW7: This covers chart type 2 and we have completed this exercise under Flight Delay Time Statistics Dashboard section
207     # Compute required information for creating graph from the data
208     avg_car, avg_weather, avg_NAS, avg_sec, avg_late = compute_data_choice_2(df)
209
210     # Create graph
211     carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')
212     weather_fig = px.line(avg_weather, x='Month', y='WeatherDelay', color='Reporting_Airline', title='Average weather delay time (minutes) by airline')
213     nas_fig = px.line(avg_NAS, x='Month', y='NASDelay', color='Reporting_Airline', title='Average NAS delay time (minutes) by airline')
214     sec_fig = px.line(avg_sec, x='Month', y='SecurityDelay', color='Reporting_Airline', title='Average security delay time (minutes) by airline')
215     late_fig = px.line(avg_late, x='Month', y='LateAircraftDelay', color='Reporting_Airline', title='Average late aircraft delay time (minutes) by airline')
216
217     return[dcc.Graph(figure=carrier_fig),
218            dcc.Graph(figure=weather_fig),
219            dcc.Graph(figure=nas_fig),
220            dcc.Graph(figure=sec_fig),
221            dcc.Graph(figure=late_fig)]
222
223 # Run the app
224 if __name__ == '__main__':
225     app.run_server()

```

```

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

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

# REVIEW1: Clear the layout and do not display exception till callback gets executed
app.config.suppress_callback_exceptions = True

# Read the airline data into pandas dataframe
airline_data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetworkDataScienceSkillsInPython/Module%205/Airline%20Performance%20Metrics/airline_data.csv',
                           encoding = "ISO-8859-1",
                           dtype={'Div1Airport': str, 'Div1TailNum': str,
                                  'Div2Airport': str, 'Div2TailNum': str})

# List of years
year_list = [i for i in range(2005, 2021, 1)]

"""Compute graph data for creating yearly airline performance report

Function that takes airline data as input and create 5 dataframes based on the grouping condition to be used for plotting.

Argument:
    df: Filtered dataframe

Returns:
    Dataframes to create graph.
"""

def compute_data_choice_1(df):
    # Cancellation Category Count
    bar_data = df.groupby(['Month','CancellationCode'])['Flights'].sum().reset_index()
    # Average flight time by reporting airline
    line_data = df.groupby(['Month','Reporting_Airline'])['AirTime'].mean().reset_index()
    # Diverted Airport Landings
    div_data = df[df['DivAirportLandings'] != 0.0]
    # Source state count
    map_data = df.groupby(['OriginState'])['Flights'].sum().reset_index()
    # Destination state count
    tree_data = df.groupby(['DestState', 'Reporting_Airline'])['Flights'].sum().reset_index()
    return bar_data, line_data, div_data, map_data, tree_data

"""Compute graph data for creating yearly airline delay report

This function takes in airline data and selected year as an input and performs computation for creating charts and plots.
"""

```

```

Arguments:
    df: Input airline data.

Returns:
    Computed average dataframes for carrier delay, weather delay, NAS delay, security delay, and late aircraft delay.
"""

def compute_data_choice_2(df):
    # Compute delay averages
    avg_car = df.groupby(['Month', 'Reporting_Airline'])['CarrierDelay'].mean().reset_index()
    avg_weather = df.groupby(['Month', 'Reporting_Airline'])['WeatherDelay'].mean().reset_index()
    avg_NAS = df.groupby(['Month', 'Reporting_Airline'])['NASDelay'].mean().reset_index()
    avg_sec = df.groupby(['Month', 'Reporting_Airline'])['SecurityDelay'].mean().reset_index()
    avg_late = df.groupby(['Month', 'Reporting_Airline'])['LateAircraftDelay'].mean().reset_index()
    return avg_car, avg_weather, avg_NAS, avg_sec, avg_late

# Application layout
app.layout = html.Div(children=[
    # TASK1: Add title to the dashboard
    # Enter your code below. Make sure you have correct formatting.
    html.H1('US Domestic Airline Flights Performance',
           style={'textAlign': 'center', 'color': '#503D36', 'font-size': 24}),
    # REVIEW2: Dropdown creation
    # Create an outer division
    html.Div([
        # Add an division
        html.Div([
            # Create an division for adding dropdown helper text for report type
            html.Div(
                [
                    html.H2('Report Type:', style={'margin-right': '2em'}),
                ]
            ),
            # TASK2: Add a dropdown
            # Enter your code below. Make sure you have correct formatting.
            dcc.Dropdown(id='input-type',
                         options=[
                             {'label': 'Yearly Airline Performance Report', 'value': 'OPT1'},
                             {'label': 'Yearly Airline Delay Report', 'value': 'OPT2'},
                         ],
                         placeholder='Select a report type',
                         style={'text-align-last': 'center', 'font-size': '20px', 'width': '150px'}
            )
        ], style={'display': 'flex'})
    ])
])

```

```

# Add next division
html.Div([
    # Create an division for adding dropdown helper text for choosing year
    html.Div(
        [
            html.H2('Choose Year:', style={'margin-right': '2em'})
        ]
    ),
    dcc.Dropdown(id='input-year',
        # Update dropdown values using list comprehension
        options=[{'label': i, 'value': i} for i in year_list],
        placeholder="Select a year",
        style={'width': '80%', 'padding': '3px', 'font-size': '20px', 'text-align-last': 'center'}),
        # Place them next to each other using the division style
        ], style={'display': 'flex'}),
]),
]),

# Add Computed graphs
# REVIEW3: Observe how we add an empty division and providing an id that will be updated during callback
html.Div([ ], id='plot1'),

html.Div([
    html.Div([ ], id='plot2'),
    html.Div([ ], id='plot3')
], style={'display': 'flex'}),

# TASK3: Add a division with two empty divisions inside. See above division for example.
# Enter your code below. Make sure you have correct formatting.
#html.Div([ ], id='plot1'),

html.Div([
    html.Div([ ], id='plot4'),
    html.Div([ ], id='plot5')
], style={'display': 'flex'}))
])

```

```

# Callback function definition
# TASK4: Add 5 output components
# Enter your code below. Make sure you have correct formatting.
@app.callback( [Output(component_id='plot1', component_property='children'),
    Output(component_id='plot2', component_property='children'),
    Output(component_id='plot3', component_property='children'),
    Output(component_id='plot4', component_property='children'),
    Output(component_id='plot5', component_property='children')],
    [Input(component_id='input-type', component_property='value'),
    Input(component_id='input-year', component_property='value')], 
    # REVIEW4: Holding output state till user enters all the form information. In this case, it will be changed
    [State("plot1", "children"), State("plot2", "children"),
    State("plot3", "children"), State("plot4", "children"),
    State("plot5", "children")]
)
# Add computation to callback function and return graph
def get_graph(chart, year, children1, children2, c3, c4, c5):

    # Select data
    df = airline_data[airline_data['Year']==int(year)]

    if chart == 'OPT1':
        # Compute required information for creating graph from the data
        bar_data, line_data, div_data, map_data, tree_data = compute_data_choice_1(df)

        # Number of flights under different cancellation categories
        bar_fig = px.bar(bar_data, x='Month', y='Flights', color='CancellationCode', title='Monthly Flight Cancellations')

        # TASK5: Average flight time by reporting airline
        # Enter your code below. Make sure you have correct formatting.
        line_fig = px.line(line_data, x='Month', y='AirTime', color='Reporting_Airline', title='Average monthly flight time by reporting airline')

        # Percentage of diverted airport landings per reporting airline
        pie_fig = px.pie(div_data, values='Flights', names='Reporting_Airline', title='% of flights by reporting airline')

        # REVIEW5: Number of flights flying from each state using choropleth
        map_fig = px.choropleth(map_data, # Input data
            locations='OriginState',
            color='Flights',
            hover_data=['OriginState', 'Flights'],
            locationmode = 'USA-states', # Set to plot as US States
            color_continuous_scale='GnBu',
            range_color=[0, map_data['Flights'].max()])
        map_fig.update_layout(
            title_text = 'Number of flights from origin state',
            geo_scope='usa') # Plot only the USA instead of globe

        # TASK6: Number of flights flying to each state from each reporting airline
        # Enter your code below. Make sure you have correct formatting.
        tree_fig = px.treemap(tree_data, path=['DestState', 'Reporting_Airline'],
            values='Flights',
            color='Flights',
            color_continuous_scale='RdBu',
            title='Flight count by airline to destination state')
    )

```

```

else:
    # REVIEW7: This covers chart type 2 and we have completed this exercise under Flight Delay Time Statistics
    # Compute required information for creating graph from the data
    avg_car, avg_weather, avg_NAS, avg_sec, avg_late = compute_data_choice_2(df)

    # Create graph
    carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')
    weather_fig = px.line(avg_weather, x='Month', y='WeatherDelay', color='Reporting_Airline', title='Average weather delay time (minutes) by airline')
    nas_fig = px.line(avg_NAS, x='Month', y='NASDelay', color='Reporting_Airline', title='Average NAS delay time (minutes) by airline')
    sec_fig = px.line(avg_sec, x='Month', y='SecurityDelay', color='Reporting_Airline', title='Average security delay time (minutes) by airline')
    late_fig = px.line(avg_late, x='Month', y='LateAircraftDelay', color='Reporting_Airline', title='Average late aircraft delay time (minutes) by airline')

    return[dcc.Graph(figure=carrier_fig),
           dcc.Graph(figure=weather_fig),
           dcc.Graph(figure=nas_fig),
           dcc.Graph(figure=sec_fig),
           dcc.Graph(figure=late_fig)]

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

```

2 and we have completed this exercise under Flight Delay Time Statistics Dashboard section
 creating graph from the data
`avg_car, avg_weather, avg_NAS, avg_sec, avg_late = compute_data_choice_2(df)`

`carrier_fig = px.line(avg_car, x='Month', y='CarrierDelay', color='Reporting_Airline', title='Average carrier delay time (minutes) by airline')`
`weather_fig = px.line(avg_weather, x='Month', y='WeatherDelay', color='Reporting_Airline', title='Average weather delay time (minutes) by airline')`
`nas_fig = px.line(avg_NAS, x='Month', y='NASDelay', color='Reporting_Airline', title='Average NAS delay time (minutes) by airline')`
`sec_fig = px.line(avg_sec, x='Month', y='SecurityDelay', color='Reporting_Airline', title='Average security delay time (minutes) by airline')`
`late_fig = px.line(avg_late, x='Month', y='LateAircraftDelay', color='Reporting_Airline', title='Average late aircraft delay time (minutes) by airline')`

`g),`
`g),`

Screenshots

Final Exam

LATEST SUBMISSION GRADE 100%

Question 1

According to the author in the video, what does Dark Horse Analytics state are the 3 best practices for creating a visual?

- Less is not effective; Less is not attractive; Less is not impactive.
- Less is more effective; Less is more attractive; Less is more impactive.
- Less is more effective; Less is not attractive; Less is more impactive.
- None of the above.

Correct

Question 1-1

Which of the following is not true regarding data visualizations?

- Supports recommendations to different stakeholders.
- Shares unbiased representation of data.
- Explores a given dataset.
- Trains and tests a machine learning algorithm.

Correct

Question 2

What are the layers that make up the Matplotlib architecture?

- ~~FigureCanvas Layer, Renderer Layer, and Artist Layer.~~
- ~~Figure Layer, Artist Layer, and Scripting Layer.~~
- ~~Backend Layer, FigureCanvas Layer, Renderer Layer, Artist Layer, and Scripting Layer.~~
- ~~Backend Layer, Artist Layer, and Scripting Layer.~~
- ~~Backend_Bases Layer, Artist Layer, Scripting Layer.~~

Correct

Question 3

The following code uses what layer to create a stacked area plot of the data in the *pandas* dataframe, *area_df*?

```
ax = area_df.plot(kind='area', figsize=(20, 10))
ax.set_title('Plot Title')
ax.set_ylabel('Vertical Axis Label')
ax.set_xlabel('Horizontal Axis Label')
```

- ~~Artist layer~~
- ~~Scripting layer~~
- ~~Backend Layer~~
- ~~None of the above~~

Correct

Question 3-1

Which of the following codes uses the scripting layer to create a stacked area plot of the data in the *pandas* dataframe, `area_df`?

```
**import matplotlib.pyplot as plt
area_df.plot(kind='area', figsize=(20, 10))
plt.title('Plot Title')
plt.ylabel('Vertical Axis Label')
plt.xlabel('Horizontal Axis Label')
plt.show()**
```

- `~~ax = area_df.plot(kind='area', figsize=(20, 10)) ax.title('Plot Title') ax.ylabel('Vertical Axis Label') ax.xlabel('Horizontal Axis Label')~~`
- `~~ax = area_df.plot(type='area', figsize=(20, 10)) ax.set_title('Plot Title') ax.set_ylabel('Vertical Axis Label') ax.set_xlabel('Horizontal Axis Label')~~`
- None of the above

Correct

Question 4

The following code will create a stacked area plot of the data in the *pandas* dataframe, `area_df`, with a transparency value of 0.35?

```
import matplotlib.pyplot as plt
transparency = 0.35
area_df.plot(kind='area', alpha=transparency, figsize=(20, 10))
plt.title('Plot Title')
plt.ylabel('Vertical Axis Label')
plt.xlabel('Horizontal Axis Label')
plt.show()
```

- True
- False

Correct

Question 4-1

Which of the following codes will create an unstacked area plot of the data in the *pandas* dataframe, *area_df*, with a transparency value of 0.55?

```
**transparency = 0.55
ax = area_df.plot(kind='area', alpha=transparency, stacked=False, figsize=(20, 10))
ax.set_title('Plot Title')
ax.set_ylabel('Vertical Axis Label')
ax.set_xlabel('Horizontal Axis Label')**
```

- ~~import matplotlib.pyplot as plt transparency = 1 - 0.55 area_df.plot(kind='area', alpha=transparency, stacked=False, figsize=(20, 10)) plt.title('Plot Title') plt.ylabel('Vertical Axis Label') plt.xlabel('Horizontal Axis Label') plt.show()~~
- ~~import matplotlib.pyplot as plt area_df.plot(kind='area', stacked=False, figsize=(20, 10)) plt.title('Plot Title') plt.ylabel('Vertical Axis Label') plt.xlabel('Horizontal Axis Label') plt.show()~~
- ~~transparency = 0.55 ax = area_df.plot(kind='area', alpha=transparency, stacked=False, figsize=(20, 10))
ax.title('Plot Title') ax.ylabel('Vertical Axis Label') ax.xlabel('Horizontal Axis Label')~~

Correct

Question 5

What is a circular graphic that displays numeric proportions by dividing a circle into proportional slices?

- Bar chart
- Radial column chart
- Pie chart
- Table chart

Correct

Question 5

What is a circular graphic that displays numeric proportions by dividing a circle into proportional slices?

- Bar chart
- Radial column chart
- Pie chart
- Table chart

Correct

Question 6

A _____ is a variation of the scatter plot that displays three dimensions of data.

- Heatmap
- Bubble plot
- Scatter map
- Bar chart

Correct

Question 7

A waffle chart is great way to visualize data in relation to a whole, or to do what?

- Identify variables that have an impact on a topic of interest.
- Highlight progress against a given threshold.
- Show frequency or importance.
- Summarize a set of data measured on an interval scale.

Correct

Question 8

What is a depiction of the meaningful words in some textual data, where the more a specific word appears in the text, the bigger and bolder it appears?

- A Regression Plot
- A Waffle Chart
- A Box Plot
- A Word Cloud

Correct

Question 9

Which of the following are tile styles of Folium maps?

- Mapbox Control Room
- Stamen Watercolor
- Stamen Terrain
- OpenStreetMap
- All of the above

Correct

Question 9-1

Which of the followings are correct statements regarding Folium?

- Folium builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library.
- Folium is a powerful Python library that helps you create several type of Leaflet maps.
- The Folium results are interactive, which makes this library very useful for dashboard building.
- Folium is available by default and does not need to be installed.

Correct

Question 10

What is the correct tile style for maps that are high contrast black and white, that are perfect for data mashups and exploring river meanders and coastal zones?

- Mapbox Control Room
- Stamen Watercolor
- Stamen Toner
- Stamen Terrain

Correct

Question 10-1

Which of the followings are true regarding the Stamen Terrain tile style for Folium maps?

- Features natural vegetation colors.
- Features hill shading.
- Showcases advanced labeling and linework generalization of dual-carriageway roads.
- Is perfect for data mashups and exploring river meanders and coastal zones.

Correct

Question 11

Plotly visualizations can be displayed in which of the following ways

- Displayed in Jupyter notebook
- Saved to HTML files
- Served as a pure python-build applications using Dash
- All of the above

Correct

Question 12

Dash components are

- HTML
- CSS
- Core

Correct

<https://github.com/brendensong/IBM-Data-Science-Professional-Certificate>

<https://github.com/brendensong/IBM-Data-Science-Professional-Certificate/wiki/8.3.Advanced-Visualizations-and-Geospatial-Data>

