Module 6 Part 2: Visualization

This module consists of 2 parts:

- **Part 1** - Descriptive Statistics
- **Part 2** - Visualization

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

This is **Part 2** of the module. We will work with the Python visualization library `matplotlib` and look briefly into the `Seaborn` library.

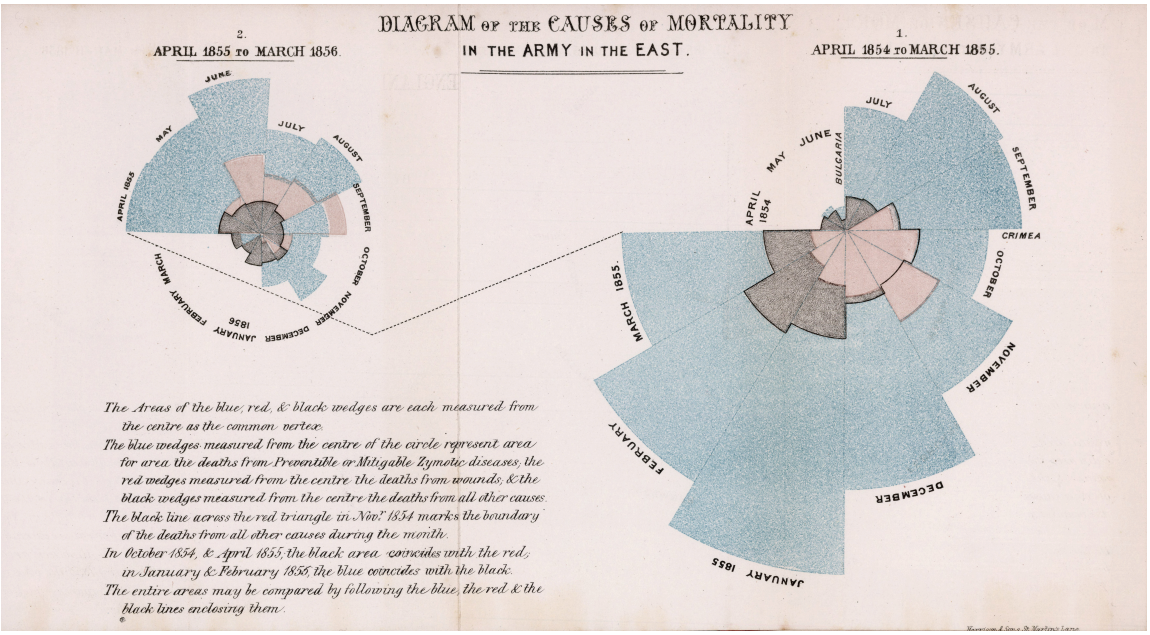
Table of Contents

History of Visualization

A good graphic can tell a story, bring a lump to the throat, even change policies." (Worth a Thousand Words, 2013).

By visualizing data, we can spot trends, identify outliers that need further analysis, and identify spikes in the data that require investigation and explanation. Good visualization tells the story.

One of the first visualizations known is the famous "Diagram of the causes of mortality in the army in the East" by Florence Nightingale (Florence Nightingale, n.d.). Florence is remembered today as "the mother of modern nursing". She was a nurse during the Crimean War (1853-1856). She realized that there were more soldiers dying from diseases than from wounds. She changed the practices in the hospitals and managed to reduce the death rate from 42% to 2%. Her detailed notes helped her create what is now considered a fine example of the power of visualization — the "Rose Diagram" which showed that epidemic disease was responsible for more British deaths in the course of the Crimean War than battlefield wounds.



Picture 1. Nightingale's Rose (Image Source: Wikimedia Commons. Public Domain.)

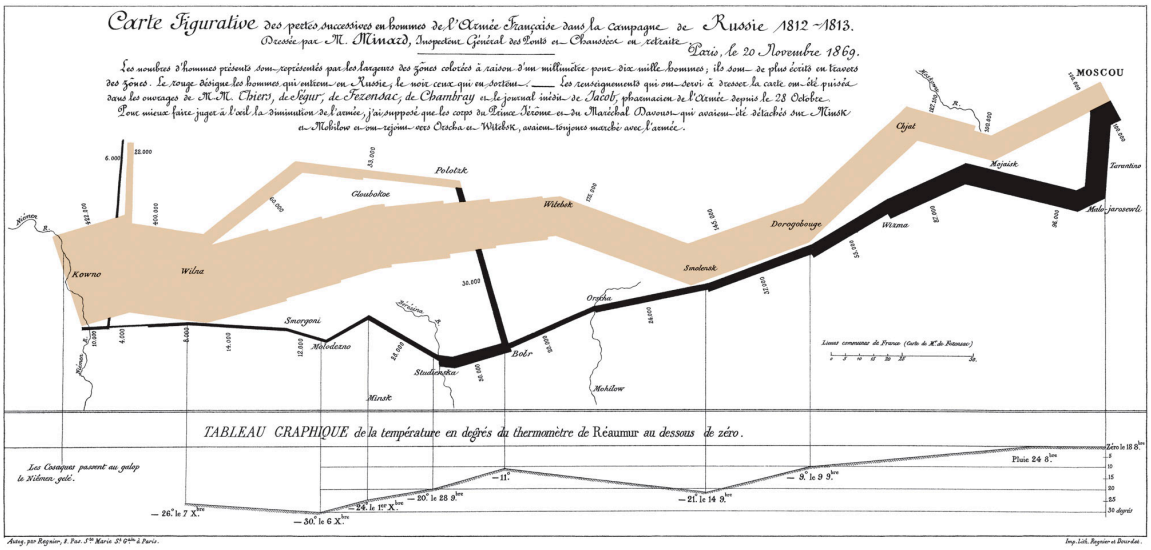
This graphic indicates the number of deaths that occurred from preventable diseases (in blue), those that were the results of wounds (in red), and those due to other causes (in black).

The next example of a powerful visualization is the map of the Napoleon's campaign of 1812. It was created by Charles Joseph Minard (Charles Joseph Minard, n.d.), a French civil engineer who made significant contributions in the field of statistical graphics.

Minard's map of the Napoleon's 1812 campaign depicts the advance into (1812) and retreat from (1813) Russia by Napoleon's Grande Armée. Minard's graphic is quite unique because it combines multiple dimensions of data into one single graphic. It is a chart and a map. It shows six types of information: geography, time, temperature, the course and direction of the army's movement, and the number of soldiers remaining. The widths of the brown (advancing) and black (retreating) lines represent the size of the army, each millimeter of line is 10,000 men. Napoleon entered Russia with 442,000 men, reached Moscow with only 100,000 soldiers and returned from Russia with just 10,000 left.

Statistician Edward Tufte (Edward Tufte, n.d.) described this chart as "The best statistical graphic ever drawn" in his famous book "The Visual Display of Quantitative Information".

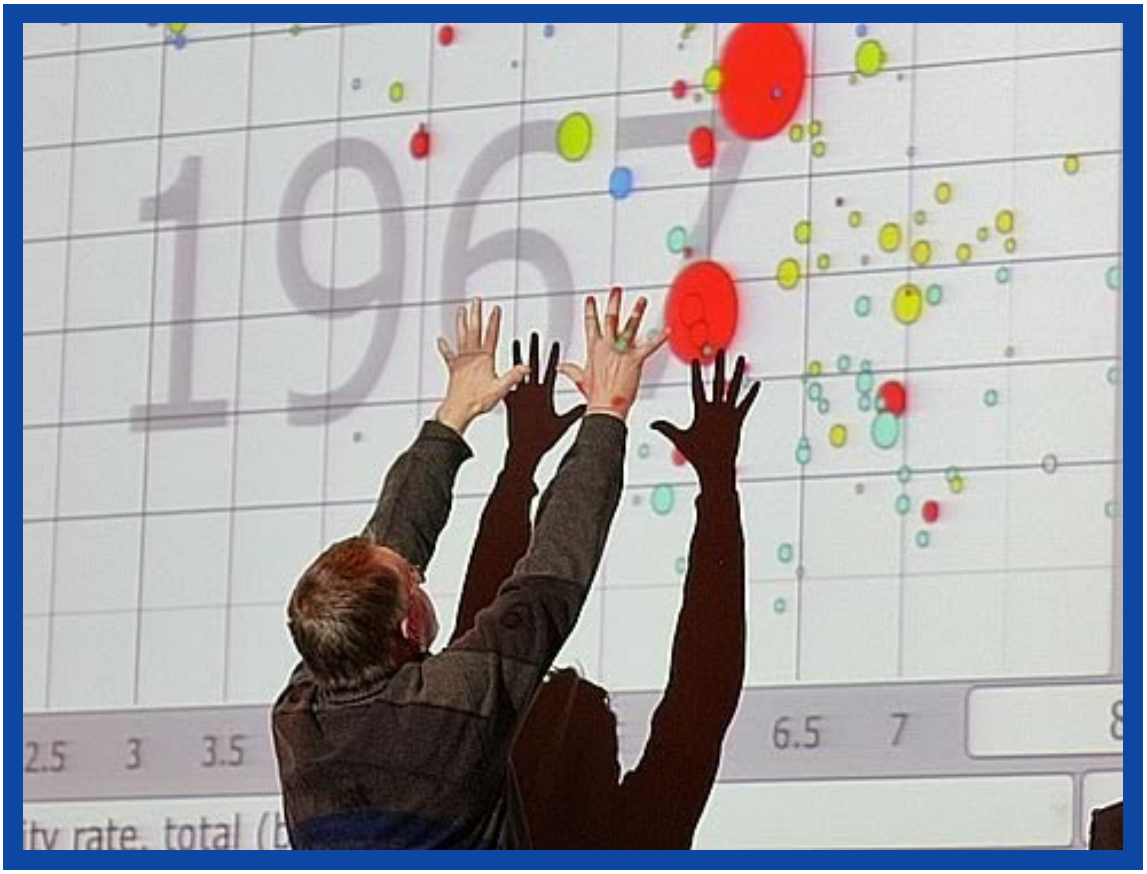
In addition to the brown and black lines we described above, Minard also showed the distance traveled, temperature, latitude and longitude, direction of travel, and location relative to specific dates.



Picture 2. Charles Minard's map of Napoleon's Russian campaign of 1812 (Image Source: Wikimedia Commons. Public Domain.)

We highly recommend that you look into Edward Tufte's book to learn the best practices of statistical graphic design. His book entitled "The Visual Display of Quantitative Information" is considered "a classic book on statistical graphics, charts, tables." See the book here: https://www.edwardtufte.com/tufte/books_vdqi.

One of the best TED talks on the visualization of the statistical data was done by Hans Rosling on the topic of global health and economics. The entire talk is very interesting and provides a great example of data analysis. If you don't have time right now to watch the entire presentation you can watch only the first 7 minutes and come back to it later:



Visualization with Matplotlib

`Matplotlib` is Python's plotting library, created in 2002 by John D. Hunter, an American neurobiologist. In 2008, John wrote "Matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays." (Hunter et al, 2018).

The latest version of Matplotlib, version 2.2.2, supports 3-D plotting with `mplot3d` toolkit, a large number of third party packages that extend and build on Matplotlib functionality, such as `seaborn` and `HoloViews` to name just two of them, and two projection and mapping toolkits, `Basemap` and `Cartopy`.

Matplotlib is very easy to use, it is fast and reliable, and provides excellent `TeX` formatting system support.

The home page of the Matplotlib library is https://matplotlib.org/ where you will find information about the latest release of the library, documentation, plot samples, and other useful links.

To start using the `matplotlib` library, we need to import it. Usually, Anaconda includes `matplotlib` pre-installed — all we need to do is to write the `import` statement in the notebook before we use the library to visualize data.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
# This will ignore the warnings
import warnings
warnings.filterwarnings("ignore")
```

What is `pyplot`?

`pyplot` is `matplotlib`'s module that provides an interface that allows us to implicitly and automatically create plots. It allows us to define any and all parameters of the plot, such as, for example, titles for the axes, grid, type of the plot, colour of the line, etc.

You might also notice the following line of code: `%matplotlib inline` in the cell above. This line of code will tell `Jupyter` that we want the output of `matplotlib` commands, the graphs, to be displayed within the notebook, not in a separate window.

We can easily change the "style" of the plot by specifying the style sheet we want to use. The `style` package offers several predefined styles and allows you to define your own.

You can see the list of available built-in styles with the following command:

In [2]:
```
print(plt.style.available)
```

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

In this notebook, we will use `style='ggplot'` which emulates the look and feel of `ggplot`, a very popular plotting package for R.

In [3]:
```
plt.style.use(style='ggplot')
```

Simple line plot

To create our first graph, we can use randomly generated data. We will create an array of random normally distributed data points. This array will be used for the Y axis.

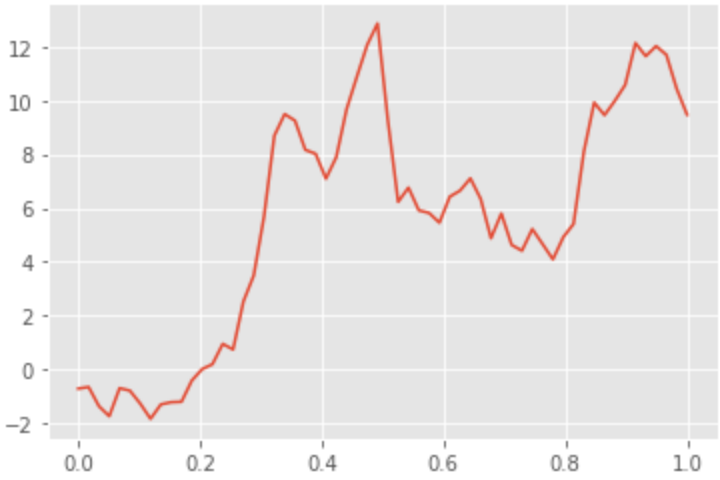
In [4]:
```
# X axis - 60 evenly spaced numbers between 0 and 1
# (refer to NumPy module of this course)
```

```
x = np.linspace(0, 1, 60)

y = np.random.randn(len(x)).cumsum()
y
```

Out[4]:
```
array([-0.70215346, -0.64381412, -1.35587164, -1.72795298, -0.68643787,
       -0.77914859, -1.25851092, -1.82523274, -1.29015997, -1.20760351,
       -1.19191407, -0.40296613,  0.02041882,  0.20385829,  0.95840159,
        0.7486838 ,  2.53385837,  3.49781768,  5.68975371,  8.69292731,
        9.49186225,  9.24728854,  8.16467927,  8.01655805,  7.09433159,
        7.89223787,  9.67211833, 10.88168007, 12.06357572, 12.85717151,
        9.26652955,  6.23188901,  6.76269552,  5.91631337,  5.81988549,
        5.4584044 ,  6.42467087,  6.65023881,  7.10517449,  6.34089966,
        4.88169914,  5.80094627,  4.63260464,  4.41797168,  5.22425666,
        4.66353552,  4.09722468,  4.91997462,  5.41387098,  8.10219945,
        9.92077651,  9.45232013,  9.98493155, 10.57373626, 12.12871831,
       11.64263453, 12.01910248, 11.69056076, 10.42792649,  9.47027536])
```

In [5]:
```
# Now plotting:

plt.plot(x,y)
```

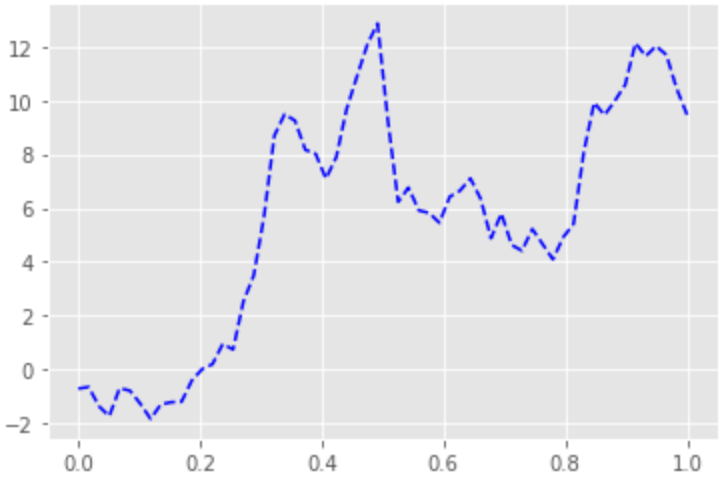
Out[5]:
```
[<matplotlib.lines.Line2D at 0x11344eb38>]
```



This is a very simple line plot of the data we have. We can change the colour and the type of the line. For example, if we want the line to be a blue, dashed line rather than a red, solid line, we need to add the corresponding parameters to the `plot()` function:

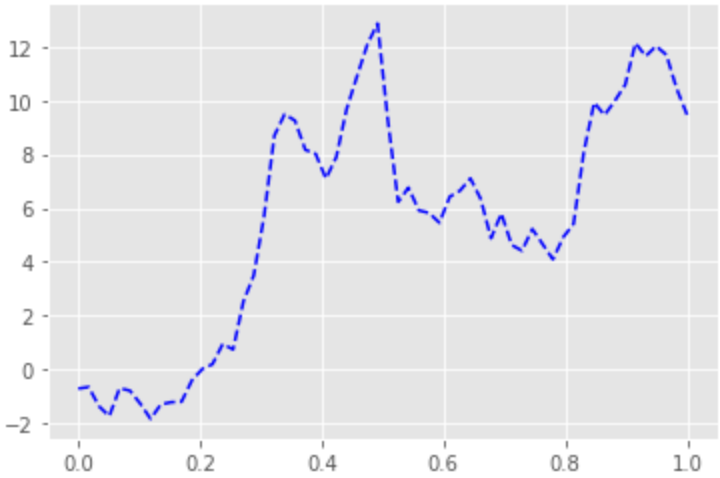
In [6]:
```
plt.plot(x,y, color = 'blue', linestyle = '--')
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x1134b2898>]
```



In [7]:
```
# The following line of code will generate the same graph as above:

plt.plot(x,y, color = 'b', linestyle = 'dashed')
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x11362a2e8>]
```



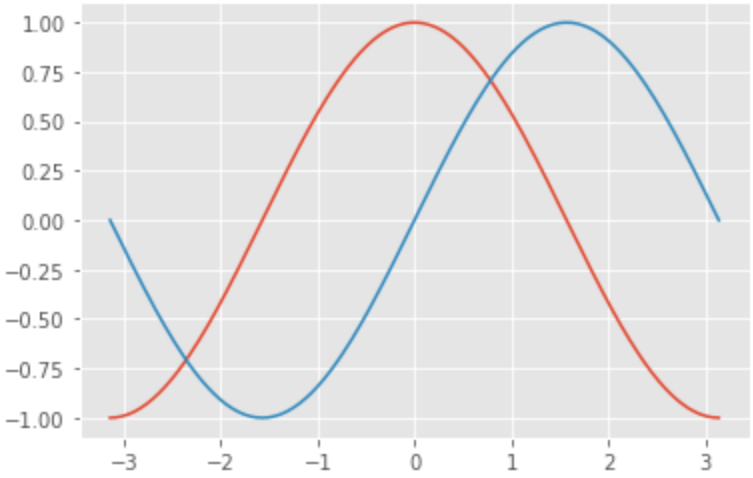
A detailed description of all parameters for `plot()` and their possible values can be found on the `matplotlib.pyplot.plot()` documentation page.

Let's create a new graph with the following data points:

- X is 200 equally spaced points from -pi to pi (to include pi, we will use the `endpoint=True` parameter)
- the plot will depict two lines, the sine and cosine graphs as functions of X
- both lines will be plotted on the same graph

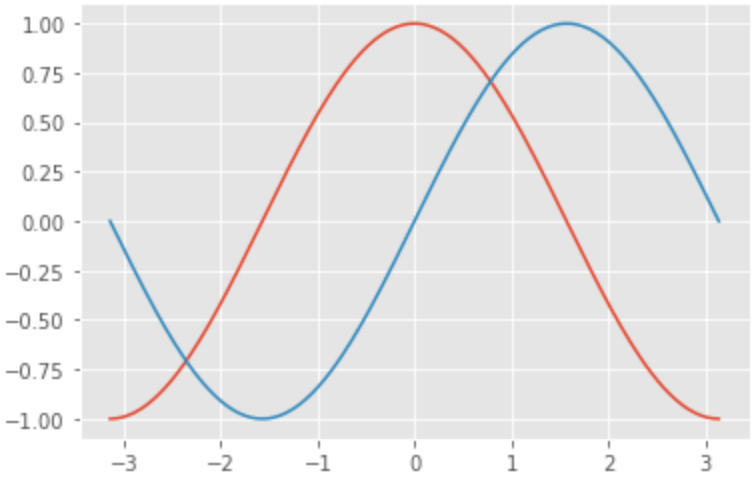
In [8]:
```
x_sincon = np.linspace(-np.pi, np.pi, 200, endpoint=True)
cosine, sine = np.cos(x_sincon), np.sin(x_sincon)

# To plot both functions on the same graph, we simply list them one after another
plt.plot(x_sincon, cosine)    # plotting cosine function, red line
plt.plot(x_sincon, sine)      # plotting sine function, blue line
```

Out[8]: [<matplotlib.lines.Line2D at 0x113495f60>]



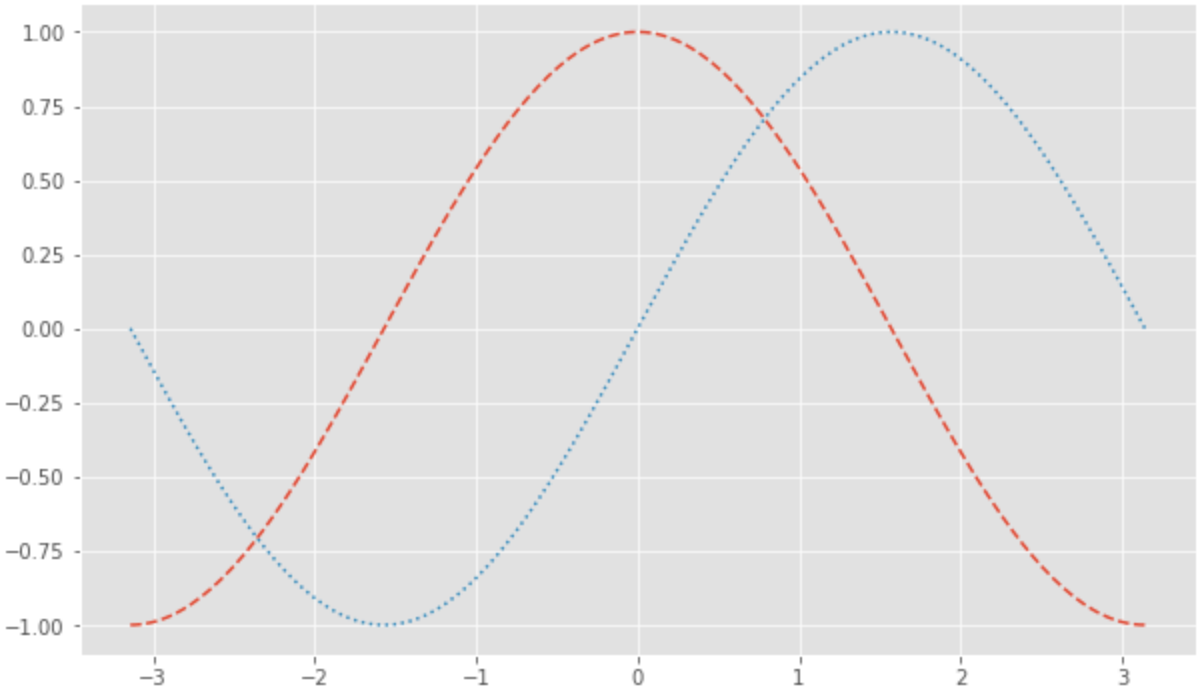
In [9]:
```
# This way will work too:

plt.plot(x_sincon, cosine, x_sincon, sine)
```

Out[9]: [<matplotlib.lines.Line2D at 0x1138a9be0>,
<matplotlib.lines.Line2D at 0x1138a9d30>]



In [10]:
```
# We can make the plot a little bigger
# and we can change the style of the lines

plt.figure(figsize=(10,6))
plt.plot(x_sincon, cosine, linestyle='dashed')     # plotting cosine function, red
plt.plot(x_sincon, sine, linestyle='dotted')       # plotting sine function, blue d
```

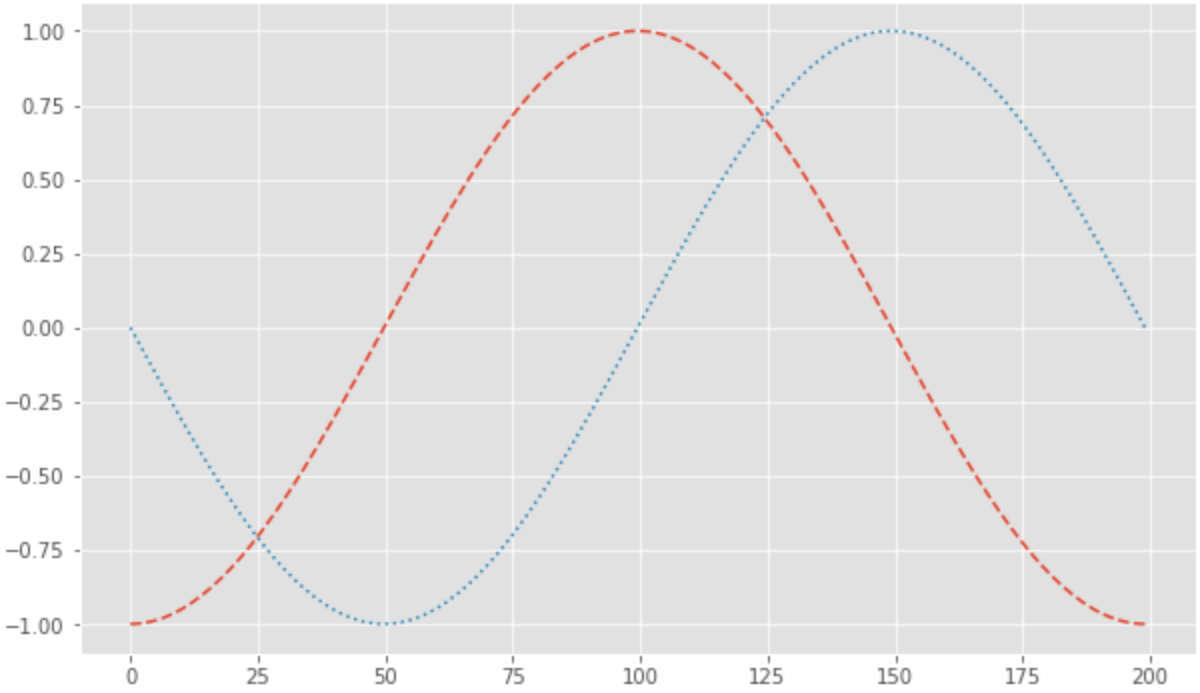
Out[10]: [<matplotlib.lines.Line2D at 0x113646898>]



If we don't provide an X axis, matplotlib will use the default X values, starting from 0. Please note there is a simplier way of specifying the style of the line, simply using `'--'` for a dashed line and `':'` for a dotted line:

In [11]:
```
plt.figure(figsize=(10,6))
plt.plot(cosine, '--')   # no X axis is provided, just the Y function
plt.plot(sine, ':')     # same as for the first graph
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x1139d14e0>]
```



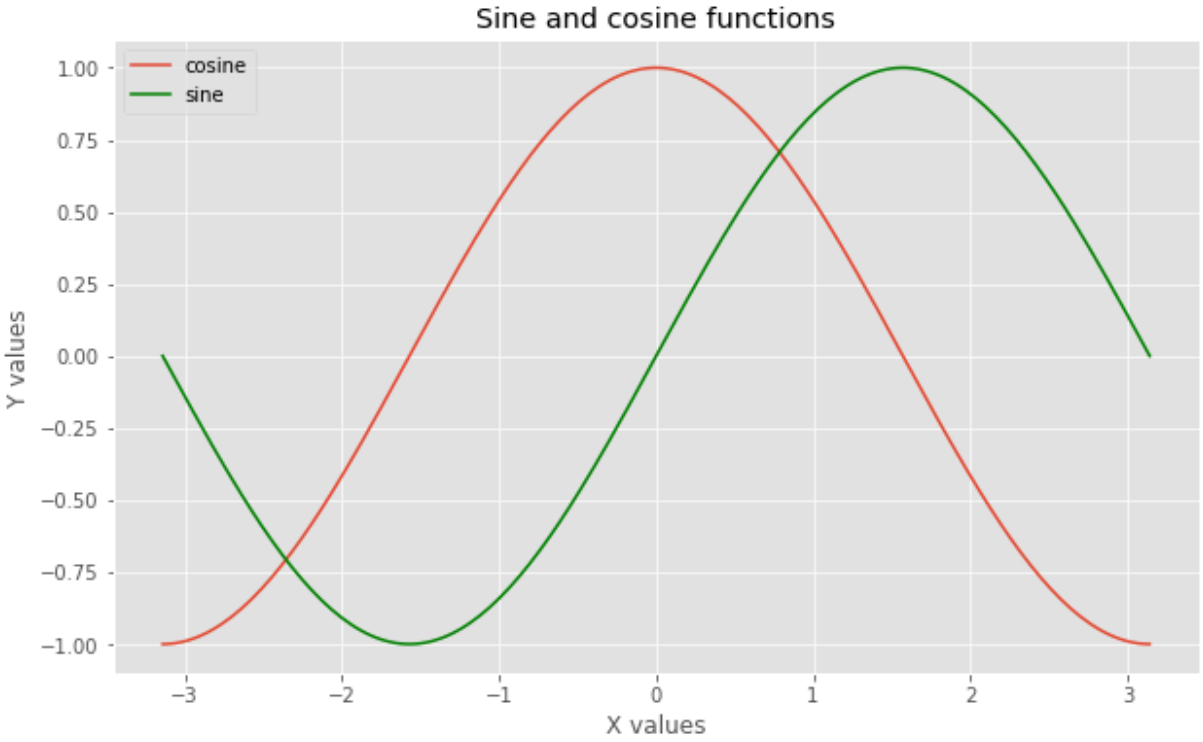
We can also add the following to the plot:

- Labels for the X and Y axes
- Title of the plot
- A legend for the lines

You will also see below that as we start adding parameters describing the lines, like labels or colours, we have to use a separate `plot()` for each of the lines.

In [12]:
```
plt.figure(figsize=(10,6))

plt.plot(x_sincon, cosine, label = 'cosine')                # adding label paramet
plt.plot(x_sincon, sine, label = 'sine', color = 'green')   # adding the color par

plt.ylabel("Y values")   # title of the Y axis
plt.xlabel("X values")   # title of the X axis
plt.title("Sine and cosine functions")   # plot title, will be displayed above the
plt.legend(loc='upper left')             # whether the legend has to be dispayed on
```

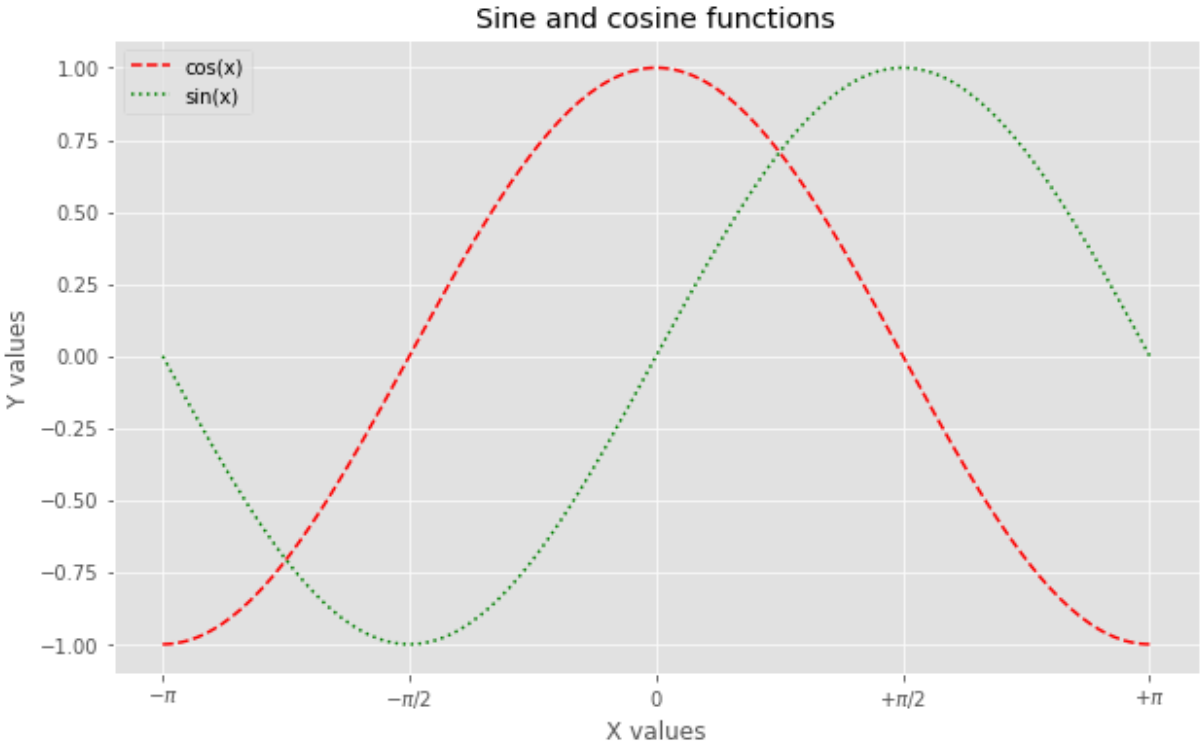
Out[12]:
```
<matplotlib.legend.Legend at 0x113c45588>
```



We can also update the ticks for the X axis and show $ pi $ values on the X axis instead of the current scale of integers from -3 to 3. We will also label the X ticks using LaTeX formatting. Finally, we will combine the `color` and `linestyle` parameters into a single parameter for simplicity.

In [13]:
```
plt.figure(figsize=(10,6))

plt.plot(x_sincon, cosine, '--r', label = 'cos(x)')
plt.plot(x_sincon, sine, ':g', label = 'sin(x)', color = 'green')

plt.ylabel("Y values")
```

```
plt.xlabel("X values")
plt.title("Sine and cosine functions")

plt.legend(loc='upper left')
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
       [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

plt.show();
```



Histograms

Let's go back to the histogram that we were plotting in Part 1 of this module and break it down in more detail to understand how it was created and what parameters were specified for the histogram plot.

We created two histograms to demonstrate the impact of different standard deviation values on distribution. In other words, how the variance in the data described by the standard deviation affects the distribution. A low standard deviation indicates that the values of a random variable are close to the expected value, while a high standard deviation indicates that the values are spread out over a wider range of values.

In our previous example, one of the histograms was plotted with `sigma = 10` and the other one, with `sigma = 30`. Let's plot both histograms on the same plot for better visualization.

First, we define the main variables we will need:

In [14]:
```
mu = 100        # mean of distribution
sigma10 = 10    # standard deviation of distribution, first value of sigma = 10
sigma30 = 30    # standard deviation of distribution, second value of sigma = 30

# Creating two arrays of 10,000 random numbers each with the same mean = 100
# but different standard deviations:
x10 = np.random.normal(loc=mu, scale=sigma10, size=10000)
x30 = np.random.normal(loc=mu, scale=sigma30, size=10000)

# Another way of getting the same result as the lines of code above:
# x10 = mu + sigma10 * np.random.randn(10000)
# x30 = mu + sigma30 * np.random.randn(10000)
```

The values that the `hist()` function return are:

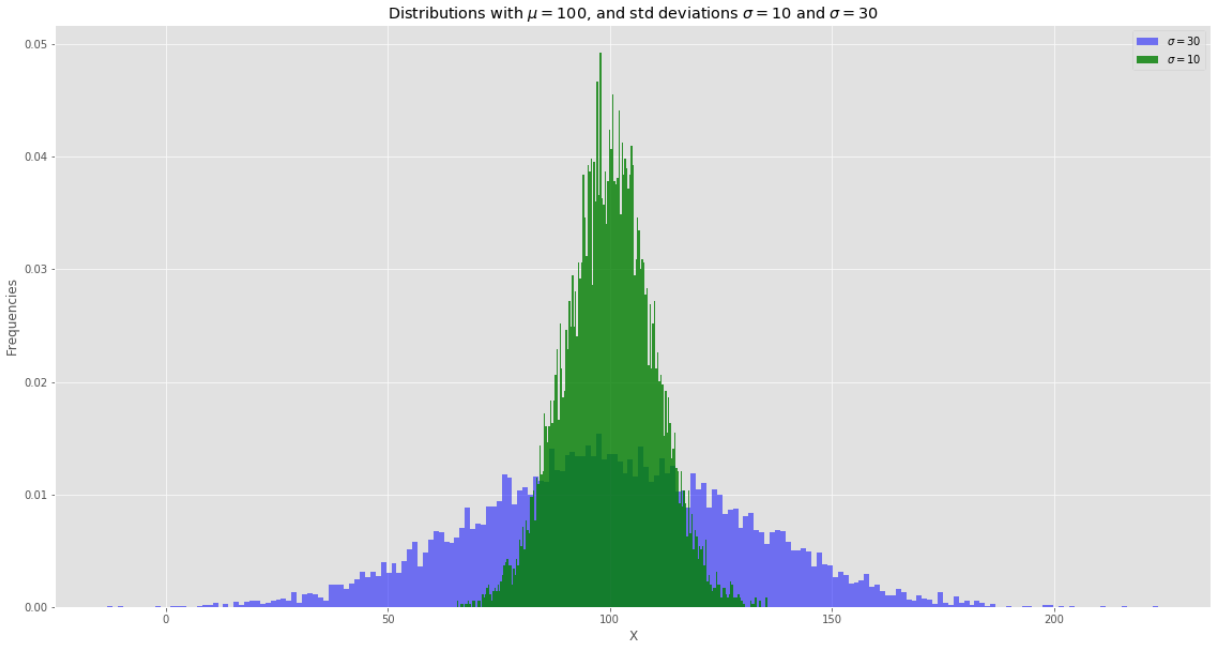
- `n` - values of histogram bins, the number of counts in each bin of the histogram
- `bins` - array of edges of the bins
- `patches` - list of individual patches used to create the histogram, i.e. a collection of rectangles. The patches can be used to change the properties of individual bars.

For a detailed description of all parameters for the `hist()` function of matplotlib, please refer to the documentation.

We will create a green graph to depict the distribution of the values of the variable with standard deviation $sigma = 10$, and a blue graph to show the distribution with $ sigma = 30 $.

All other parameters used in the graph are exactly the same.

In [15]:
```
# Size of the histogram plot:
plt.figure(figsize=(20,10))

# Number of bins to be created
num_bins = 200

n, bins, patches = plt.hist(x30, num_bins, density=1, facecolor='b', alpha=0.5, lab
n, bins, patches = plt.hist(x10, num_bins, density=1, facecolor='green', alpha=0.8,

plt.title("Distributions with $\mu=100$, and std deviations $\sigma=10$ and $\sigma
plt.xlabel("X")
plt.ylabel("Frequencies")
plt.legend()

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

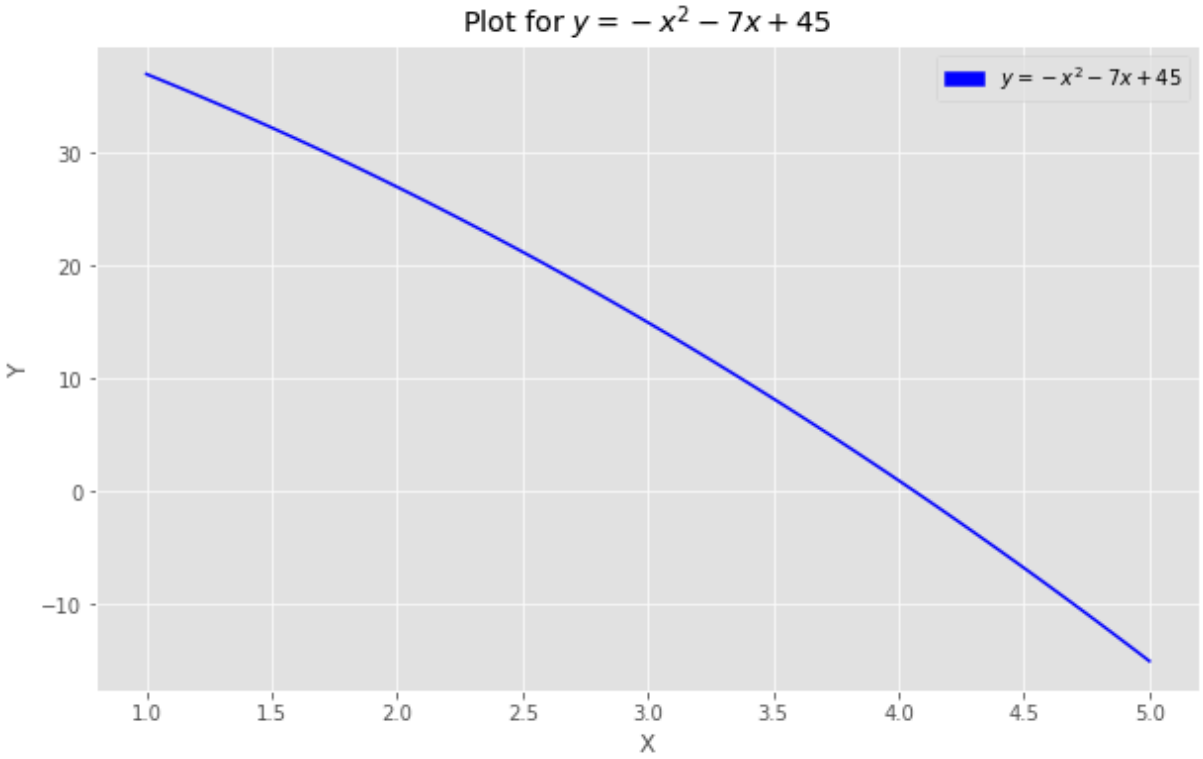


Saving plots as a file

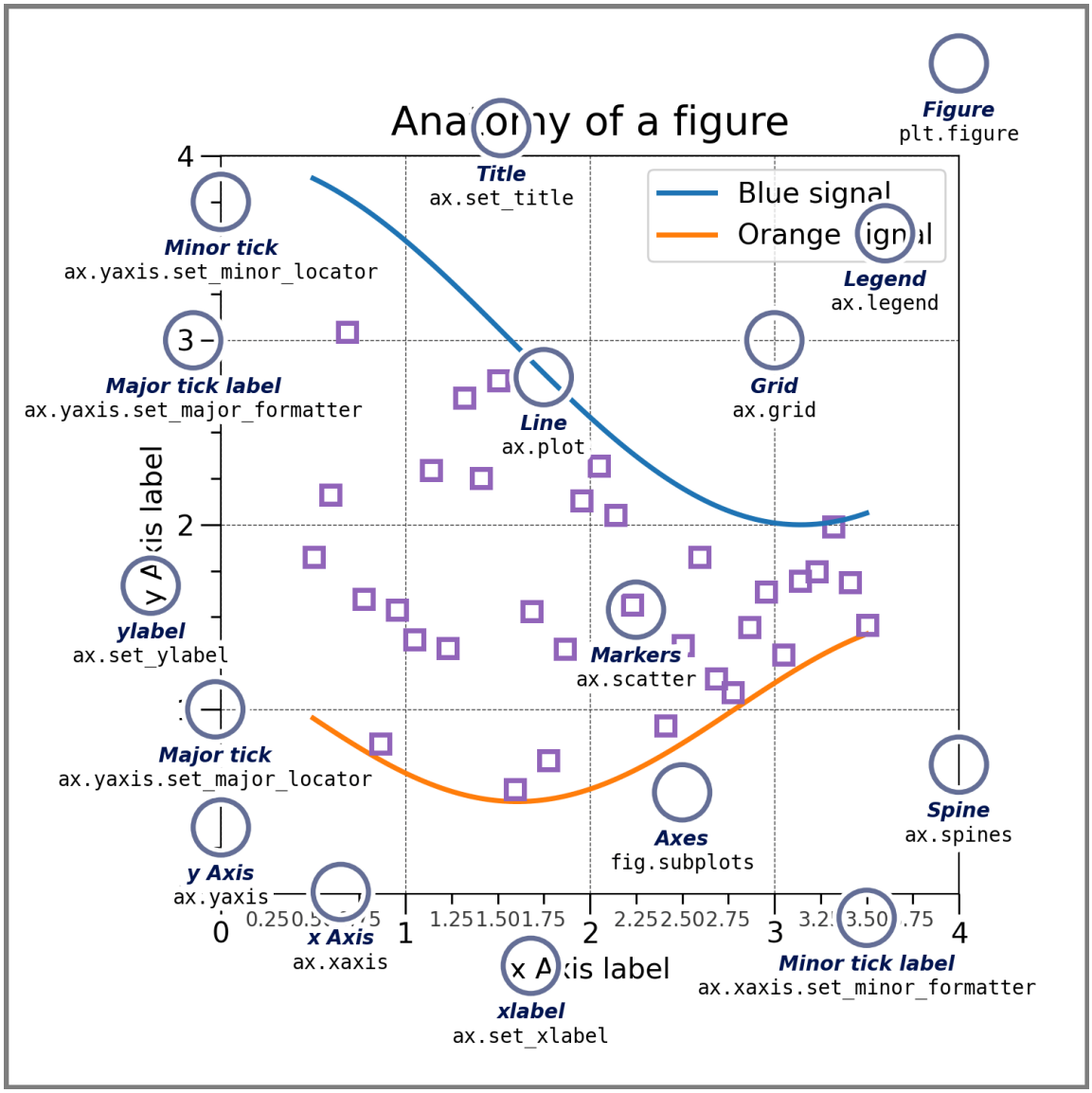
If there is a need to save a plot as an image file, in addition to displaying the graph within the notebook, we can easily do that using the `savefig()` function. In the example below, we are generating a line graph based on the formula $y = -x^2 - 7x + 45$, where X is an array of 100 data points from 1 to 5, inclusive. We also use the `patches` library that will allow us to customize the legend of the graph (displayed in the top right corner).

In [16]:
```
import matplotlib.patches as mpatches
plt.figure(figsize=(10,6))

X = np.linspace(1, 5, 100, endpoint=True)
Y = -X ** 2 - 7 * X + 45

plt.plot(X, Y, color = 'b')

plt.title("Plot for $y = -x^2 - 7x + 45$")

# Simple labels for X and Y axes
plt.xlabel("X")
plt.ylabel("Y")

# Make sure that we display the grid on the plot
plt.grid(True)

# The legend patch in the top right corner
blue_patch = mpatches.Patch(color="blue", label="$y = -x^2 - 7x + 45$")
plt.legend(handles=[blue_patch])

# This will save the graph as PDF file as a file on the file system,
# in the same folder where the notebook is
```

```
plt.savefig("pic1.pdf")
plt.show()
```



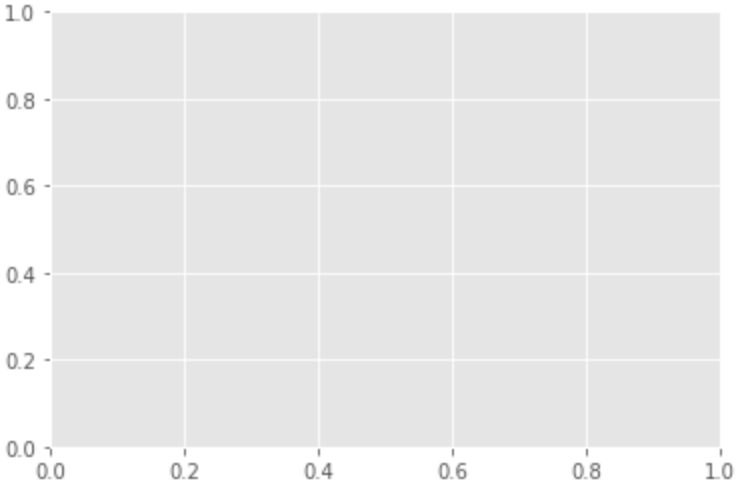
From the examples we used so far, we can see that the plot in Matplotlib is a collection of objects: the *figure* as a container which holds *axes*, *tick marks*, plot *lines*, *legends*, and other objects which we can easily customize for the specific graph we are drawing. This is how matplotlib presents this concept:



Picture 3: Anatomy of a Figure

We can create an empty figure just by declaring the `figure` object as a container and `axes`. Matplotlib will use the default values for axes:

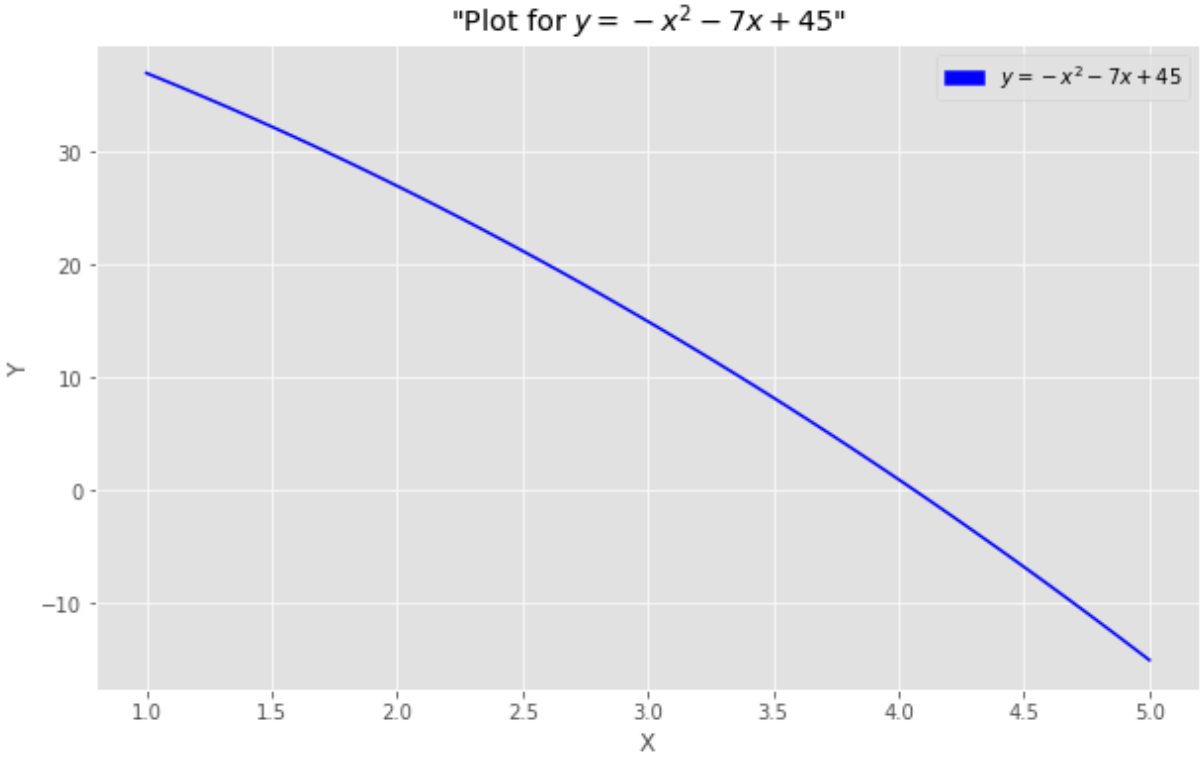
In [17]:
```
fig = plt.figure()
ax = plt.axes()
```



This is the object-oriented matplotlib's interface which allows us, for example, to use a single function, `set()`, to define all parameters for axes. Here is how we can create the previous graph using this approach:

In [18]:
```
plt.figure(figsize=(10,6))

x_formula = np.linspace(1, 5, 100, endpoint=True)
y_formula = -x_formula ** 2 - 7 * x_formula + 45

ax = plt.axes()
ax.plot(x_formula, y_formula, 'b')
ax.set(xlabel='X', ylabel='Y',
       title='"Plot for $y = -x^2 - 7x + 45$"')

# Make sure that we display the grid on the plot
plt.grid(True)

# The legend patch in the top right corner
blue_patch = mpatches.Patch(color="blue", label="$y = -x^2 - 7x + 45$")
plt.legend(handles=[blue_patch])
```

Out[18]:
```
<matplotlib.legend.Legend at 0x114bbb860>
```



Subplots and bar charts

Subplots

In the upcoming section, we will plot a bar chart. We will use `figure` and `axes` objects, just as we have already seen. We will also use another very important functionality of matplotlib: **subplots** which will allow us to create multiple plots in one figure.

Subplots can be created with the `subplot()` function, which takes 3 arguments:

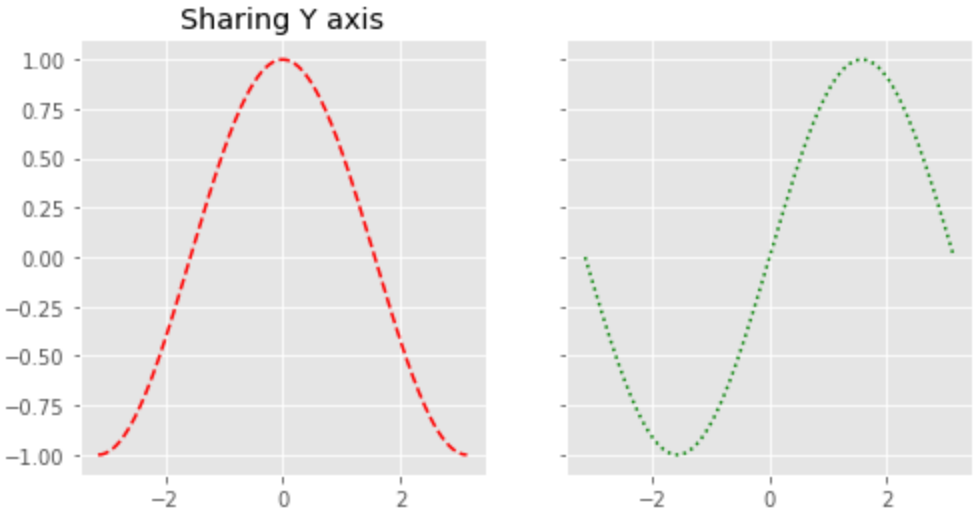
1. Number of rows
2. Mumber of columns of the subplot grid
3. Index of the plot to be created

The `subplots()` function returns `figure` and `axes` objects.

Here is a simple example where we will draw the sine and cosine functions on two separate subplots:

In [19]:
```
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(8, 4))  # one row and two
ax1.plot(x_sincon, cosine, '--r', label = 'cos(x)')
ax1.set_title('Sharing Y axis')
ax2.plot(x_sincon, sine, ':g', label = 'sin(x)', color = 'green')
```

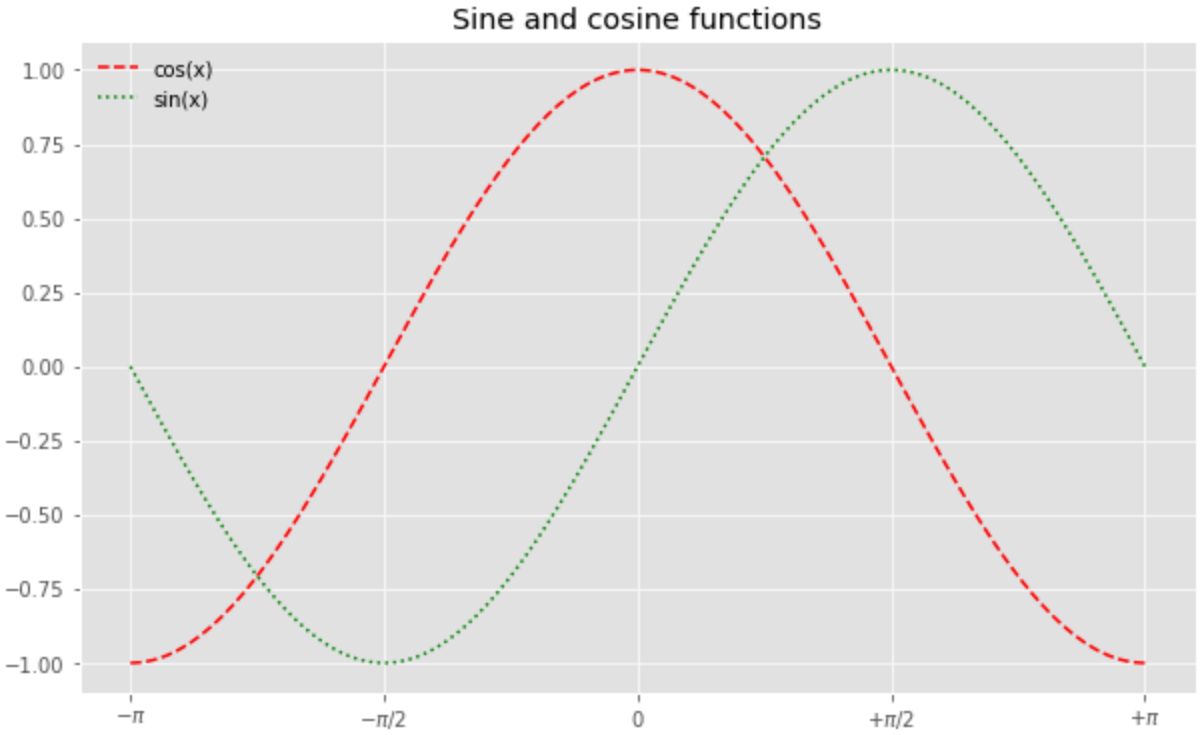
Out[19]:
```
[<matplotlib.lines.Line2D at 0x1147a0e48>]
```



In the example above, we used the `sharey` keyword which allows us to specify the relationships between Y axes of the subplots. There is also a keyword `sharex` which allows to do the same for `X` axes.

We can use the same approach to re-produce the graph we were plotting in one of the previous sections with both functions within the same coordinate system. If we do not provide the number of columns and rows for the `subplots()` function, then matplotlib assumes that we want to use a single row with one column, hence plotting both lines together.

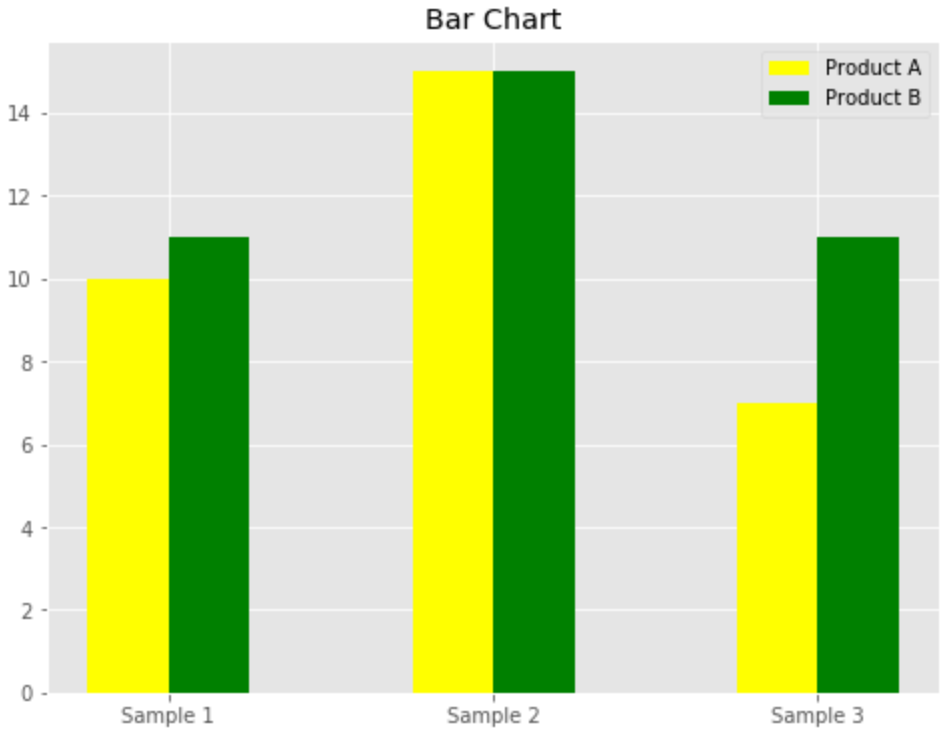
In [20]:
```
f, ax = plt.subplots(figsize=(10, 6))  # one row and two columns of subplots
ax.plot(x_sincon, cosine, '--r', label = 'cos(x)')
ax.plot(x_sincon, sine, ':g', label = 'sin(x)', color = 'green')
ax.legend(frameon=False, loc='upper left')

# setting plot title and X axes ticks
ax.set(title = 'Sine and cosine functions',
      xticks = [-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
      xticklabels = [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$']);
```



Bar charts

In the example below, we are creating two bar charts and plotting them within the same coordinate system, using one subplot. The example uses random numbers, just to illustrate how to create a bar chart. It represents three sets of observations of two parameters. For example, it could be the number of items of product A and product B purchased in three different stores which we need to draw side by side.

In [21]:
```
# Bar Chart

n_groups = 3                # number of groups of numbers
productA = (10, 15, 7)     # product A
productB = (11, 15, 11)    # product B

ind = np.arange(n_groups)   # the X locations for the groups of data
width = 0.25                # width of the bar
fig, ax = plt.subplots(figsize=(8, 6))   # drawing both sets in one subplot

prodAchart = ax.bar(ind, productA, width, color='yellow', align='center')
prodBchart = ax.bar(ind+width, productB, width, color='green', align='center')

# add some text for labels, title and axes ticks
ax.set(title = 'Bar Chart', xticks = ind + width/2,
      xticklabels = ['Sample 1', 'Sample 2', 'Sample 3'])

ax.legend((prodAchart[0], prodBchart[0]), ('Product A', 'Product B'))

plt.show()
```



EXERCISE 1: Plotting provincial support data

In the **pandas** module, we worked with provincial support data. Let's visualize this data and demonstrate the power of visual exploration.

In [22]:
```
prov_support = pd.read_csv('pandas_ex1.csv',
                           sep=',',
                           skiprows=1,
                           header=None,
                           names=['province_name','province','2016','2017','2018'],
                           index_col='province')

prov_support
```

Out[22]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

TASK: Create a bar chart to plot the data above. Group the bars by the provinces and territories. Each group will have three bars, one per year.

In [23]:
```
# <<< your code goes here >>>
```

In [ ]:

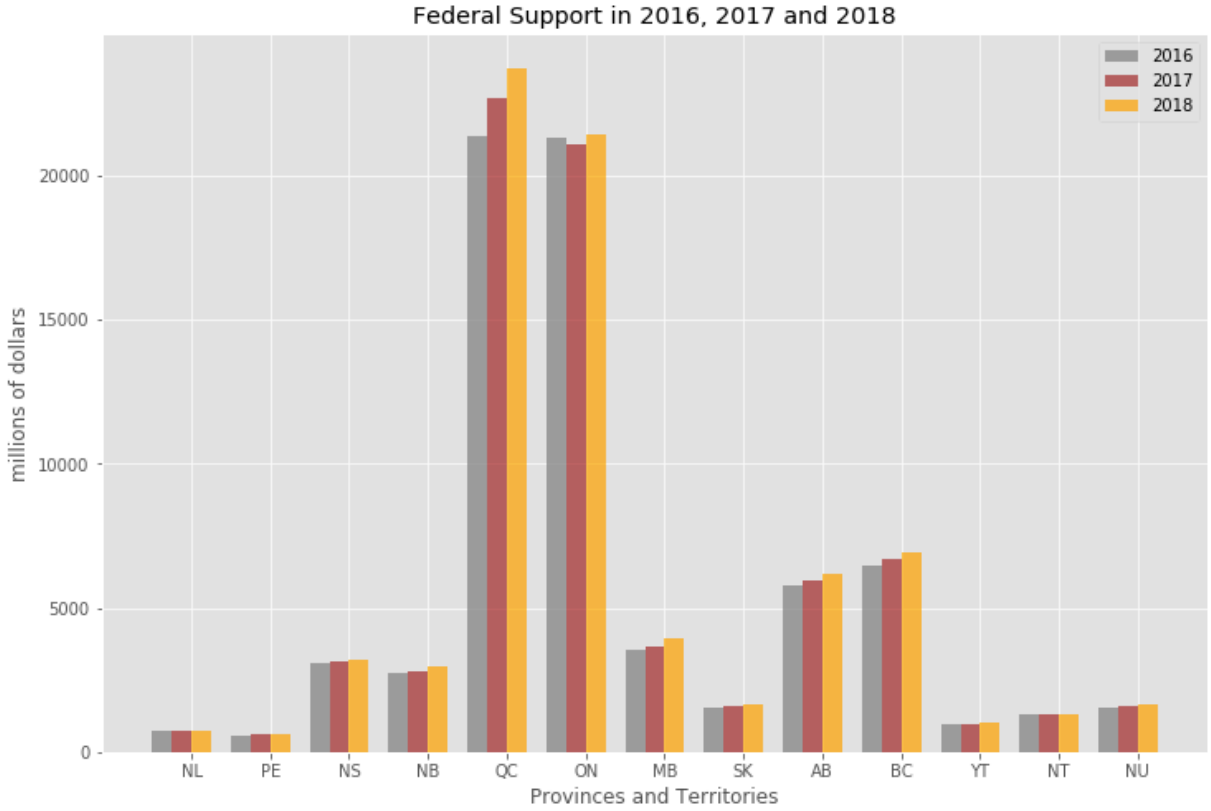
In [ ]:

In [ ]:

In [ ]:

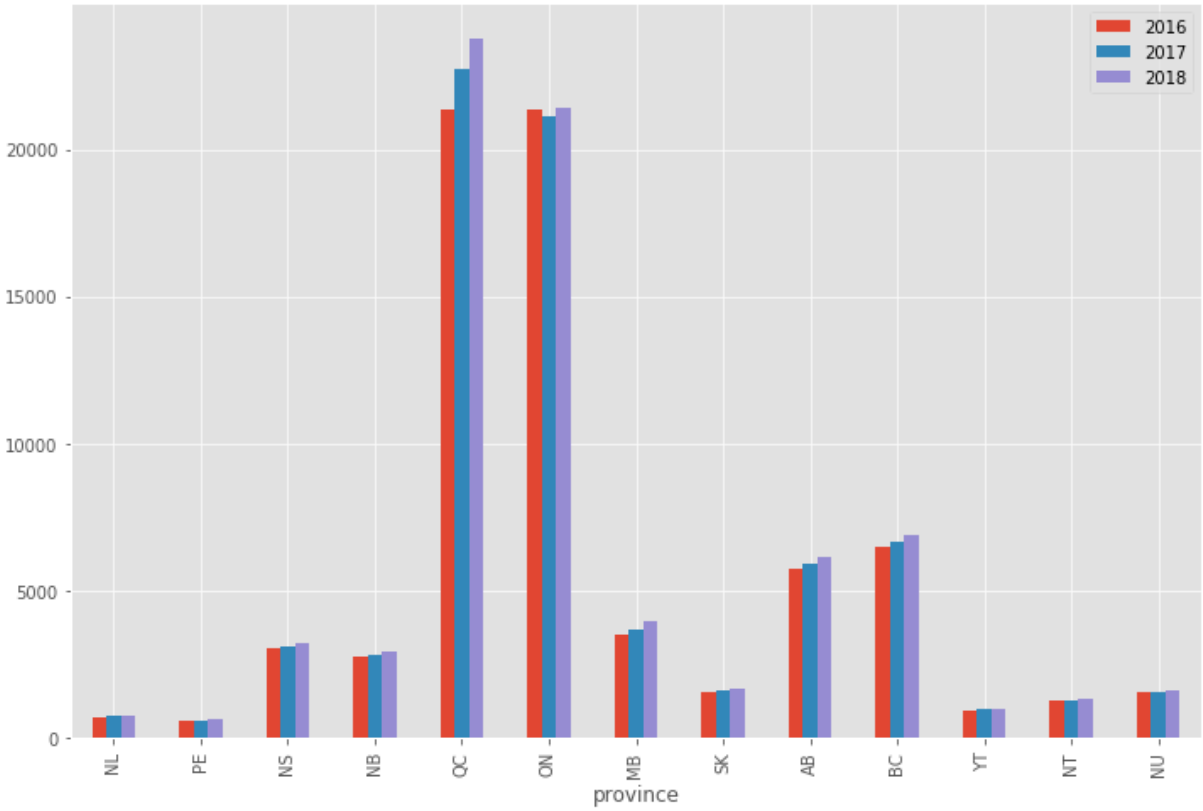
In [23]:
```
# SOLUTION

# Setting the positions and width for the bars
positions = list(range(len(prov_support.index)))
width = 0.25

# plotting the bars
fix, ax = plt.subplots(figsize=(12,8))

# plotting 2016 bars
plt.bar(positions, prov_support['2016'],
        width, color = 'gray', alpha = 0.7, label='2016')
# plotting 2017 and 2018 bars, making sure the bars do not overlap
```

```
plt.bar([p + width for p in positions], prov_support['2017'],
        width, color ='brown', alpha = 0.7, label='2017')
plt.bar([p + width*2 for p in positions], prov_support['2018'],
        width, color = 'orange', alpha = 0.7, label='2018')

ax.set(title = 'Federal Support in 2016, 2017 and 2018',
       ylabel = 'millions of dollars', xlabel = "Provinces and Territories",
       xticks = [p+1.5*width for p in positions], xticklabels = prov_support.index)

plt.legend(['2016','2017','2018'], loc = 'upper right')
plt.show()
```



Your solution might be different from the one we provided in this notebook, as there are multiple ways to draw this bar chart. For example, if all you need to do is to visualize the data to see the trends and understand what further processing is required, you can use the following quick and simple solution:

In [24]:
```
ax_provinces = prov_support[['2016', '2017', '2018']].plot(kind='bar', legend = Tru
plt.show();
```



NOTE: If you want to prevent matplotlib from including a description of the object drawn, for example, `[[<matplotlib.axis.XTick at 0x112333da0>,...]]`, add a semi-colon at the end:

```
plt.show();
```

or:

```
ax.set(title = 'Sine and cosine functions', xticks = [-np.pi, -np.pi/2, 0, np.pi/2, np.pi], xticklabels = [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$']);
```

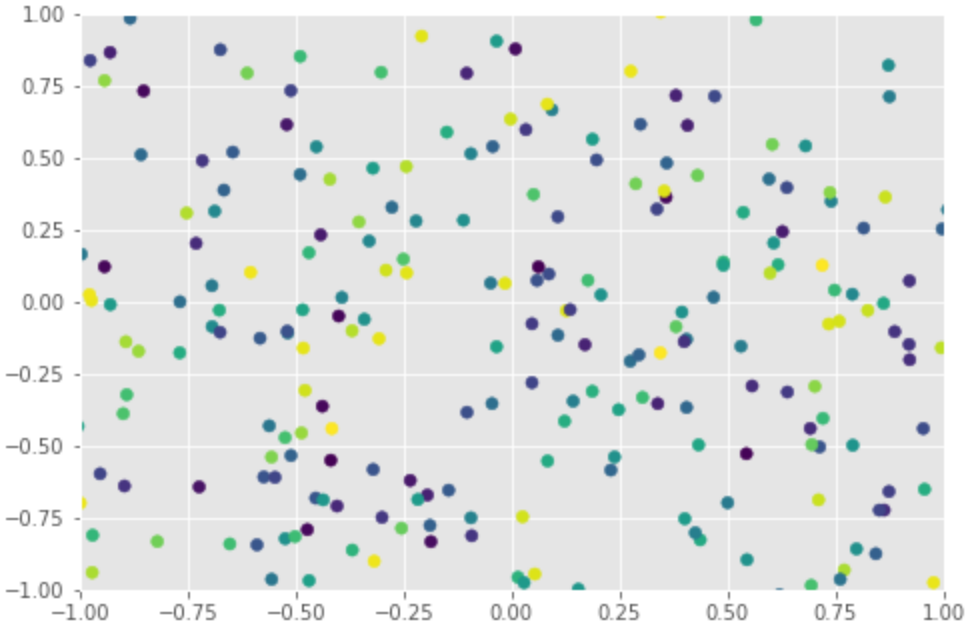
(note the `;` after the last closing bracket)

Scatter plot

A scatter plot is a commonly used visualization and is very easy to create with matplotlib. For example, below is a simple plot which draws 500 random dots within a `1 x 1` figure. The colours for the dots are also randomly selected from the standard matplotlib colour scheme.

In [25]:
```
# Scatter Plot

n = 500      # number of dots
x_scatter = np.random.normal(0, 1, n)
y_scatter = np.random.normal(0, 1, n)
colors = np.random.rand(n)
```

```
# axes([left, bottom, width, height])
plt.axes([-1, -1, 1, 1])
plt.scatter(x_scatter, y_scatter, c = colors)

# limits the values of x and y axes
plt.xlim(-1, 1)
plt.ylim(-1, 1)

plt.show()
```



NOTE: If we don't want matplotlib to draw any ticks on the plot, we can use the following function calls: `plt.xticks(())` and `plt.yticks(())` before `plt.show()`.

EXERCISE 2: Plotting iris flowers

In this exercise we will plot the data from the iris flower data set which you are familiar with from the pandas module. Reading the data into the DataFrame:

In [26]:
```
iris = pd.read_csv('iris.data', sep=',',
                   header=None,  # the data file does not contain a header
                   names=['sepal length','sepal width','petal length','petal width'
                  )

iris.head()
```

Out[26]:

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

TASK: Create a scatter plot as follows:

- Plot "sepal length" attribute on X axes
- Plot "sepal width" attribute on Y axes
- The colour of the dots should reflect the class of the flower

In [28]:
```
# << Your code goes here >>
```

In [ ]:

In [ ]:

In [ ]:

SOLUTION:

Before we start plotting, we need to deal with `class` being a categorical variable. In order to use this variable as a colour code for plotting let's convert the flower description into a code. We can do this a couple different ways. Here is the approach we will take:

- First, we will convert the `class` column into a *categorical* variable.
- Then we will use the `cat.codes` method to replace the descriptive name of the class with the category

In [27]:
```
iris['class'] = iris['class'].astype("category").cat.codes
iris.head()
```

Out[27]:

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

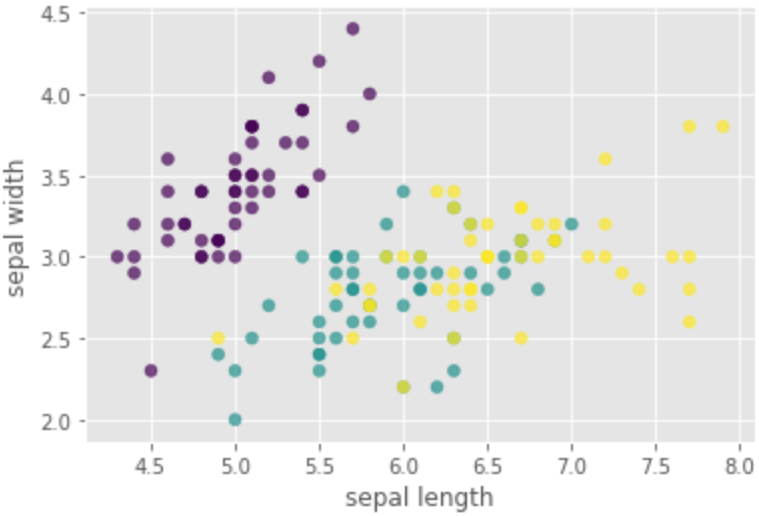
In [28]:
```
iris['class'].unique()
```

Out[28]:
```
array([0, 1, 2])
```

Good. Now we have codes for the three classes of iris flowers and we can plot.

In [29]:
```
plt.scatter(iris['sepal length'], iris['sepal width'], alpha=0.7,
            c=iris['class'], cmap='viridis')
plt.xlabel(iris.columns.values[0])
plt.ylabel(iris.columns.values[1])
```

Out[29]:
```
Text(0, 0.5, 'sepal width')
```

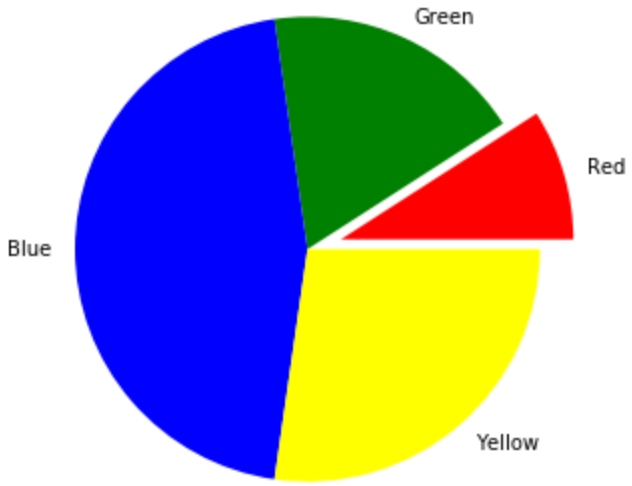


Now you can add a legend to this plot and you are done.

Pie chart

The last type of plot that we will review in this section is a pie chart.

In [30]:
```
# Pie Chart

# number of sections on the chart
N = 5
data = [1, 2, 5, 3]

# to ensure axes are square
plt.axes([0, 0, 0.9, 0.9])
# or axes(aspect=1)

# explode specifies the fraction of the radius with which to offset each wedge
plt.pie(data, explode = (0.15, 0, 0, 0),
        labels = ("Red", "Green", "Blue", "Yellow"),
        colors = ["red", "green", "blue", "yellow"])
plt.axis('equal')

plt.show()
```



Seaborn Library - Quick Overview

Seaborn is an open source Python data visualization library built on top of `matplotlib`. It provides a high-level interface for drawing statistical graphs and integrates very well with pandas DataFrames, allowing us to easily visualize data from DataFrames.

Because `seaborn` is built on top of the `matplotlib` library, it is very easy to use. In this section, we will provide a couple of examples of the graphs built using the `seaborn` library, and encourage you to use the tutorial on the `seaborn` site to learn more about this package.

To start using the library, we need to import it. By generally accepted convention among the Python community, `seaborn` is imported as `sns`:

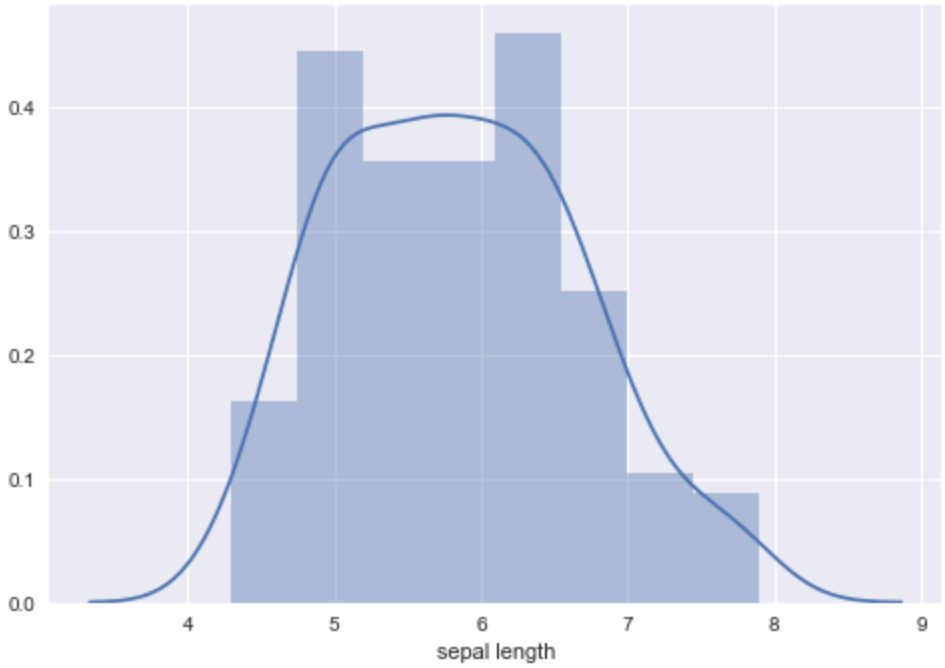
In [31]:
```
import seaborn as sns
# next line of code sets the graph style to a seaborn style
sns.set()
```

To plot a histogram, we will use function `distplot()`. By default, this function will plot the histogram with a kernel density estimate (KDE) line.

Let's plot the `sepal width` parameter from the iris dataset (for all classes). Before we draw the graph, let's reload the DataFrame from the source file:

In [32]:
```
iris = pd.read_csv('iris.data', sep=',',
                   header=None,  # the data file does not contain a header
                   names=['sepal length','sepal width','petal length','petal width'
                  )

sns.distplot(iris['sepal length'])
```

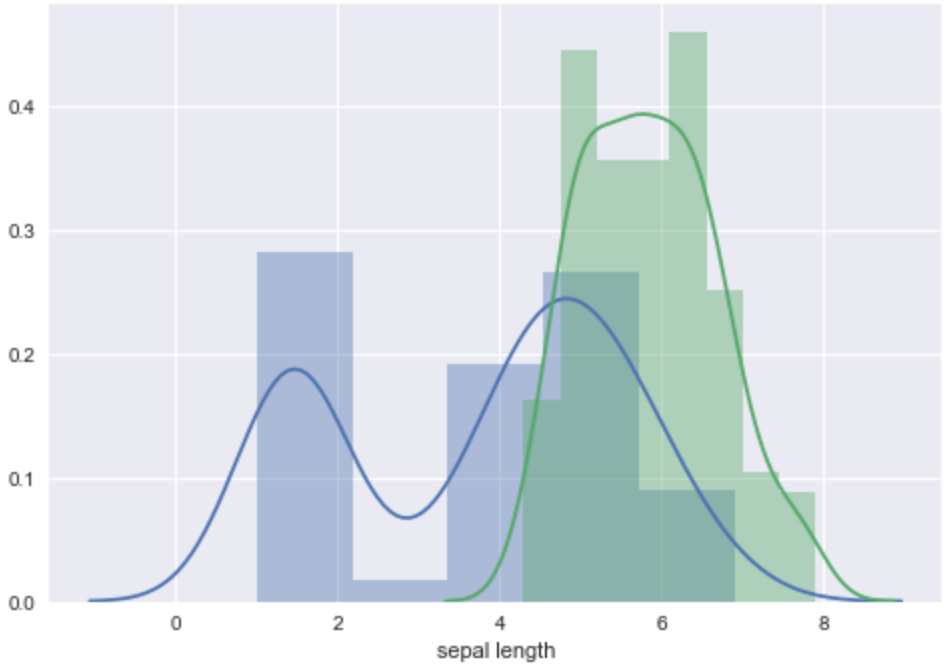
Out[32]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x117902470>
```



We can also combine multiple histograms in a single plot:

In [33]:
```
sns.distplot(iris['petal length'])
sns.distplot(iris['sepal length'])
```

Out[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11797b2b0>
```

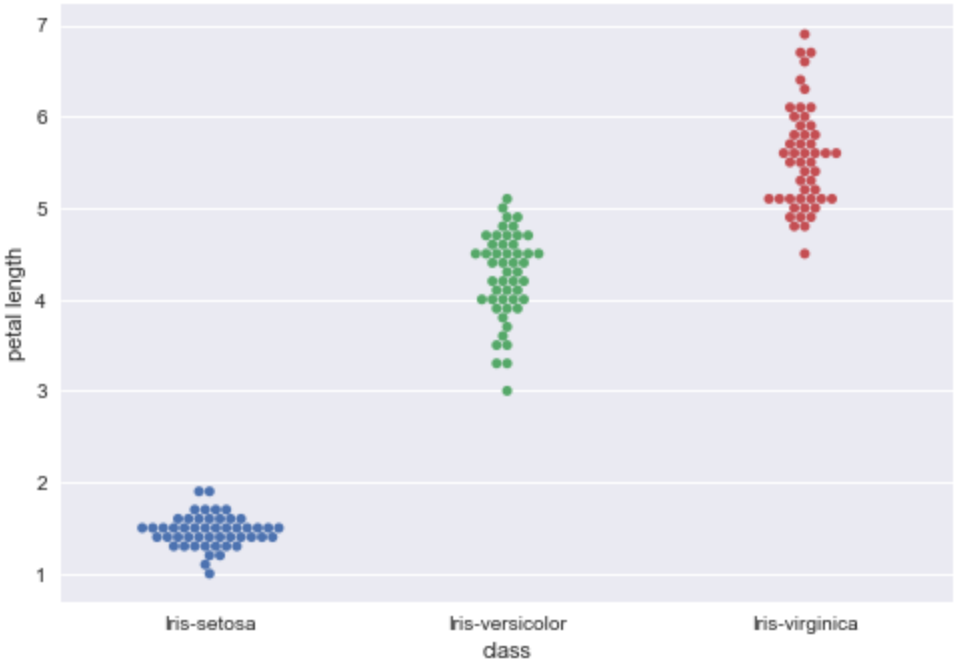


One interesting and useful plot in the `seaborn` library is `swarmplot()`. It is useful for datasets containing categorical variables. In the iris dataset the categorical variable is `class`. For example, we can draw a plot which will show us `petal length` data points per class of the flower.

In this plot, the points along the categorical axis are adjusted so that they don't overlap. This will give us a good visual representation of clusters of data points:

In [34]:
```
sns.swarmplot(x=iris['class'], y=iris['petal length'], data=iris)
```

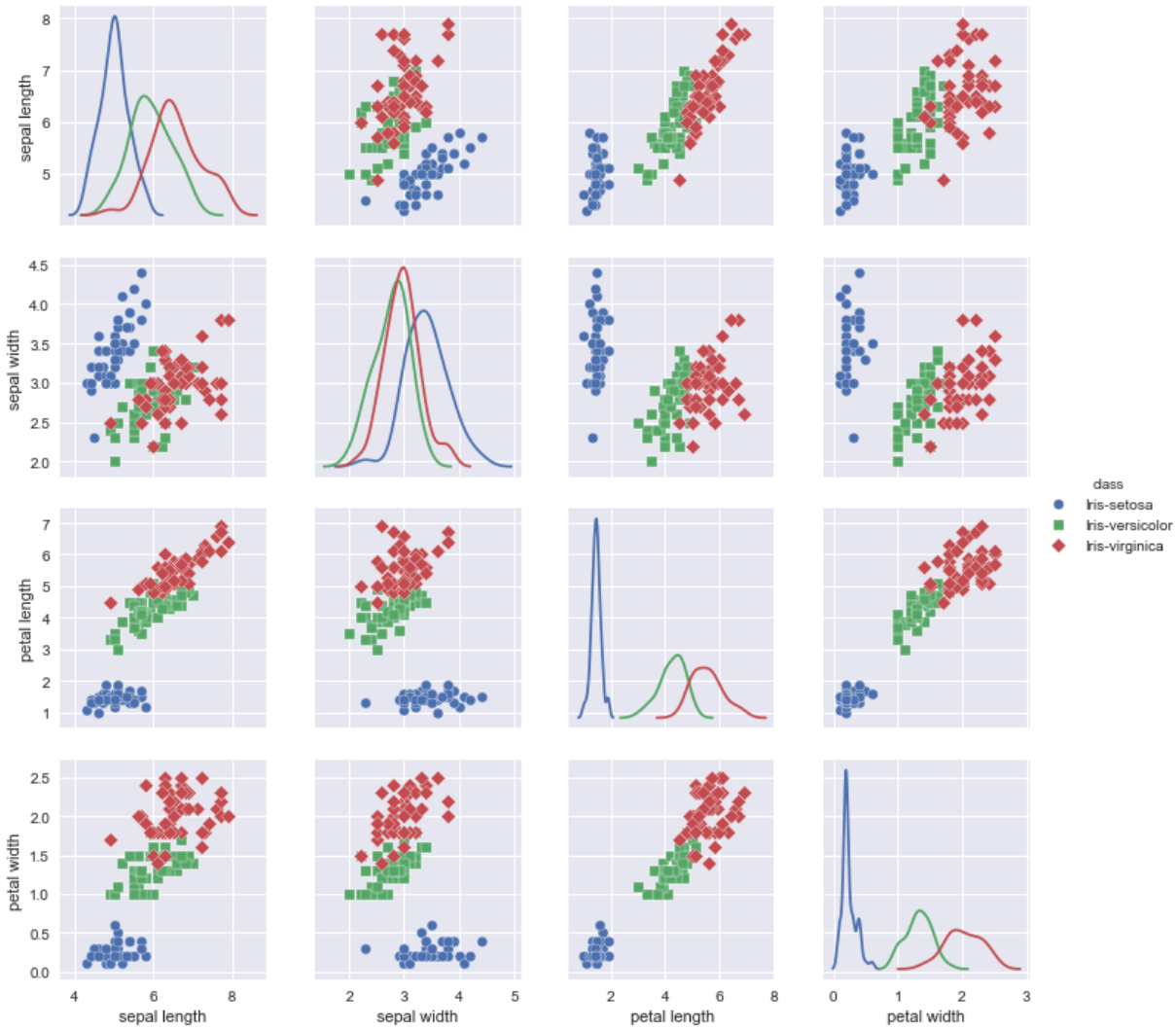
Out[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x117ab8ba8>
```



The last `seaborn` graph that we want to mention is the `pairplot()`. It draws a matrix of pairwise relationships in a dataset. Here is the `pairplot` for the iris dataset:

In [35]:
```
sns.pairplot(iris, hue="class", diag_kind = 'kde', markers=["o", "s", "D"])
```

Out[35]:
```
<seaborn.axisgrid.PairGrid at 0x11797b0b8>
```



This concludes the quick overview of the `seaborn` library. We encourage you to review the tutorial on the `seaborn` website and use this library to create beautiful visualizations in your notebooks.

End of Module

You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.

References

British Crown (2018). Cartopy Introduction. Retrieved July 18, 2018 from https://scitools.org.uk/cartopy/docs/latest/

Charles Joseph Minard (n.d.) in Wikipedia. Retrieved July 18, 2018 from https://en.wikipedia.org/wiki/Charles_Joseph_Minard

Worth a Thousand Words. (2013, Oct. 7). The Economist Newspaper Limited. Retrieved from https://www.economist.com/christmas-specials/2013/10/07/worth-a-thousand-words

Edward Tufte (n.d.) in Wikipedia. Retrieved July 18, 2018 from https://en.wikipedia.org/wiki/Edward_Tufte

Florence Nightingale (n.d.) in Wikipedia. Retrieved July 18, 2018 from https://en.wikipedia.org/wiki/Florence_Nightingale

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 History. Retrieved July 18, 2018 from https://matplotlib.org/users/history.html

Wickham, H. (2016) ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. Retrieved from: https://ggplot2.tidyverse.org/

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 Index. Retrieved July 18, 2018 from https://matplotlib.org/index.html

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 matplotlib.pyplot.hist. Retrieved July 18, 2018 from https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 matplotlib.pyplot.plot. Retrieved July 18, 2018 from https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 matplotlib.pyplot.subplots. Retrieved July 18, 2018 from https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 showcase example code: anatomy.py. Retrieved July 18, 2018 from https://matplotlib.org/examples/showcase/anatomy.html

Hunter, J., Dale, D., Firing, E., Droettboom, M., & Matplotlib development team (2018). Matplotlib Version 2.2.2 Toolkits. Retrieved July 18, 2018 from https://matplotlib.org/mpl_toolkits/index.html

John D. Hunter (n.d.) in Wikipedia. Retrieved July 18, 2018 from https://en.wikipedia.org/wiki/John_D._Hunter

Pandas 0.23.3 documentation (n.d.) Categorical Data. Retrieved July 18, 2018 from https://pandas.pydata.org/pandas-docs/stable/categorical.html

PyViz Developers (2018). HoloViews. Retrieved July 18, 2018 from http://holoviews.org/

TeX (n.d.) in Wikipedia. Retrieved July 18, 2018 from https://en.wikipedia.org/wiki/TeX

Tufte, E. (2001). The Visual Display of Quanitiative Information, 2nd Ed. Cheshire, Connecticut: Graphics Press. https://www.edwardtufte.com/tufte/books_vdqi

Waskom, M. (2018). Seaborn: statistical data visualization. Retrieved July 18, 2018 from http://seaborn.pydata.org/

Waskom, M. (2018). Seaborn 0.9.0. seaborn.distplot. Retrieved from: https://seaborn.pydata.org/generated/seaborn.distplot.html#seaborn.distplot

Kernel density estimation in Wikipedia. (2018) Retrieved from https://en.wikipedia.org/wiki/Kernel_density_estimation.

Waskom, M. (2018). Seaborn 0.9.0. seaborn.swarmplot. Retrieved from: https://seaborn.pydata.org/generated/seaborn.swarmplot.html

Waskom, M. (2018). Seaborn 0.9.0. seaborn.pairplot. Retrieved from: https://seaborn.pydata.org/generated/seaborn.pairplot.html

Whitaker, J. (2011). Welcome to the Matplotlib Basemap Toolkit documentation. Retrieved July 18, 2018 from https://matplotlib.org/basemap/

In [ ]: