Data Visualization using Matplotlib


Badreesh Shetty · Follow
Published in Towards Data Science
11 min read · Nov 12, 2018

Data Visualization is an important part of business activities as organizations nowadays collect a huge amount of data. Sensors all over the world are collecting climate data, user data through clicks, car data for prediction of steering wheels etc. All of these data collected hold key insights for businesses and visualizations make these insights easy to interpret.

> *Data is only as good as it's presented.*

Why are visualizations important?

Visualizations are the easiest way to analyze and absorb information. Visuals help to easily understand the complex problem. They help in identifying patterns, relationships, and outliers in data. It helps in understanding business problems better and quickly. It helps to build a compelling story based on visuals. Insights gathered from the visuals help in building strategies for businesses. It is also a precursor to many high-level data analysis for Exploratory Data Analysis(EDA) and Machine Learning(ML).

> *Human beings are visual creatures. Countless studies show how our brain is wired for the visual, and processes everything faster when it is through the eye.*

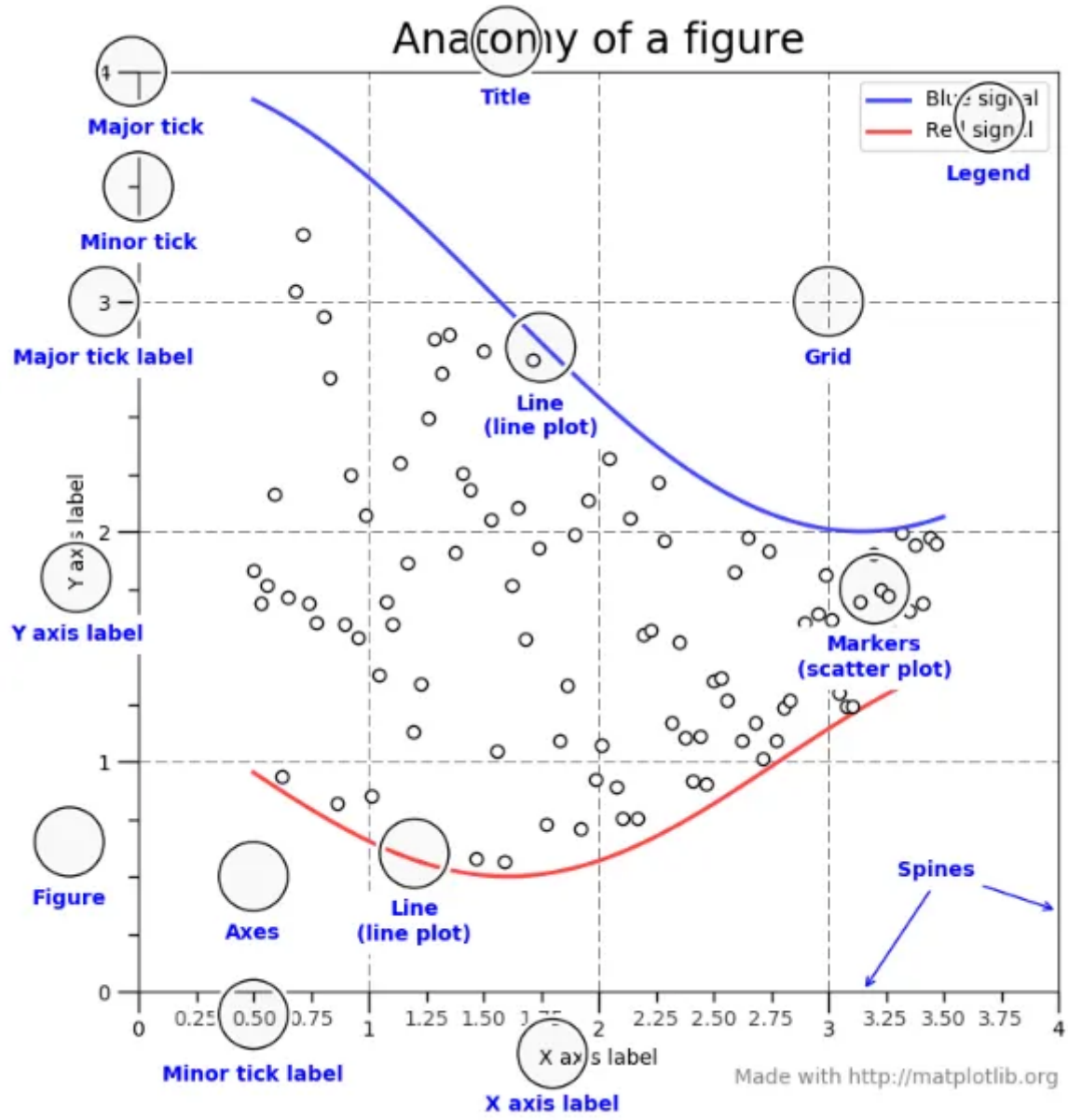
> “Even if your role does not directly involve the nuts and bolts of data science, it is useful to know what data visualization can do and how it is realized in the real world.”
>
> - Ramie Jacobson

Data visualizations in python can be done via many packages. We’ll be discussing of matplotlib package. It can be used in Python scripts, Jupyter notebook, and web application servers.

Matplotlib

Matplotlib is a 2-D plotting library that helps in visualizing figures. Matplotlib emulates Matlab like graphs and visualizations. Matlab is not free, is difficult to scale and as a programming language is tedious. So, matplotlib in Python is used as it is a robust, free and easy library for data visualization.

Anatomy of Matplotlib Figure



Anatomy of Matpotlib

The figure contains the overall window where plotting happens, contained within the figure are where actual graphs are plotted. Every Axes has an x-axis and y-axis for plotting. And contained within the axes are titles, ticks, labels associated with each axis. An essential figure of matplotlib is that we can more than axes in a figure which helps in building multiple plots, as shown below. In matplotlib, pyplot is used to create figures and change the characteristics of figures.

Installing Matplotlib

Type `!pip install matplotlib` in the Jupyter Notebook or if it doesn't work in cmd type `conda install -c conda-forge matplotlib` . This should work in most cases.

Things to follow

Plotting of Matplotlib is quite easy. Generally, while plotting they follow the same steps in each and every plot. Matplotlib has a module called pyplot which aids in plotting figure. The Jupyter notebook is used for running the plots. We `import matplotlib.pyplot as plt` for making it call the package module.

- Importing required libraries and dataset to plot using Pandas `pd.read_csv()`
- Extracting important parts for plots using conditions on Pandas Dataframes.
- `plt.plot()` for plotting line chart similarly in place of plot other functions are used for plotting. All plotting functions require data and it is provided in the function through parameters.
- `plot.xlabel` , `plt.ylabel` for labeling x and y-axis respectively.
- `plt.xticks` , `plt.yticks` for labeling x and y-axis observation tick points respectively.
- `plt.legend()` for signifying the observation variables.
- `plt.title()` for setting the title of the plot.
- `plot.show()` for displaying the plot.

Histogram

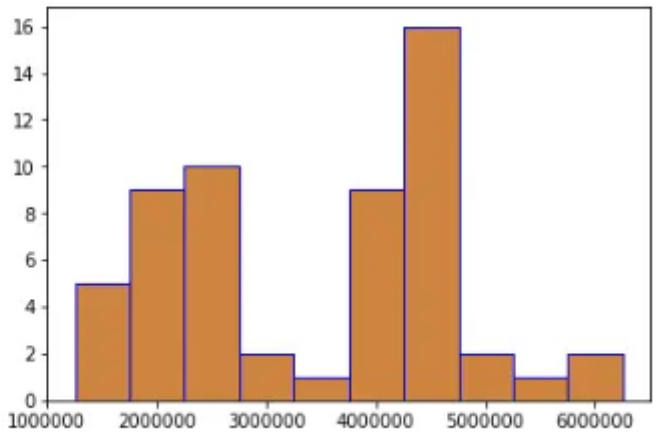
A histogram takes in a series of data and divides the data into a number of bins. It then plots the frequency data points in each bin (i.e. the interval of points). It is useful in understanding the count of data ranges.

When to use: We should use histogram when we need the count of the variable in a plot.

eg: Number of particular games sold in a store.

Histogram

In [3]:
```
plt.hist(np_data['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10)
plt.show()
```



From above we can see the histogram for GrandCanyon visitors in years. `plt.hist()` takes the first argument as numeric data in the horizontal axis i.e GrandCanyon visitor.bins=10 is used to create 10 bins between values of visitors in GrandCanyon.

Components of a histogram

- **n:** Contains the frequency of each bin
- **bins:** Represents the middle value of each bin
- **patches:** The Patch object for the rectangle shape representing each bar

In [4]:
```
n, bins, patches = plt.hist(np_data['GrandCanyon'],
                            facecolor='peru',
                            edgecolor='blue',
                            bins=10)

print('n: ', n)
print('bins: ', bins)
print('patches: ', patches)
```

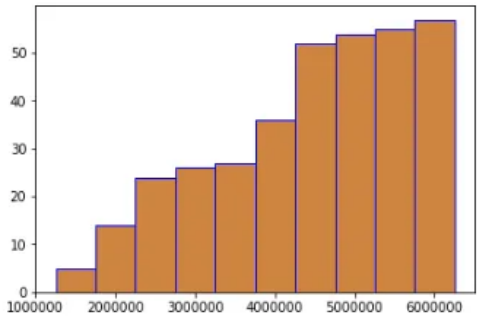
```
n:  [ 5.  9. 10.  2.  1.  9. 16.  2.  1.  2.]
bins:  [1253000.  1753123.8 2253247.6 2753371.4 3253495.2 3753619.  4253742.8
 4753866.6 5253990.4 5754114.2 6254238. ]
patches:  <a list of 10 Patch objects>
```

From above, we can see the components that make a histogram, n as the max values in each bin of histogram i.e 5,9, and so on.

The cumulative property

If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

In [5]:
```
plt.hist(np_data['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10,
         cumulative=True)
plt.show()
```

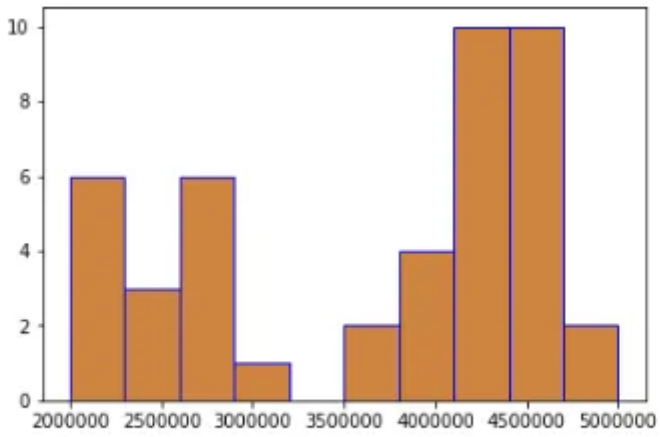


The cumulative property gives us the end added value and helps us understand the increase in value at each bin.

Check the histogram to a range of values

We only look at the data points within the range 2M-5M. This realigns the bins in the histogram

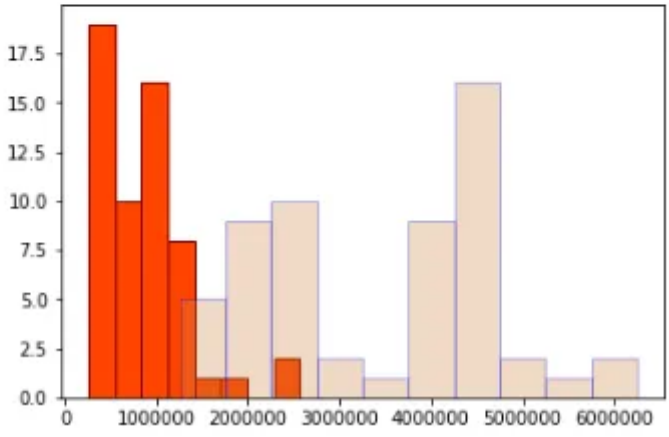
In [6]:
```
plt.hist(np_data['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10,
         range=(2000000, 5000000))

plt.show()
```



Range helps us in understanding value distribution between specified values.

Multiple histograms

In [7]:
```
plt.hist(np_data['BryceCanyon'],
         facecolor='orangered',
         edgecolor='maroon',
         bins=8)

plt.hist(np_data['GrandCanyon'],
         facecolor='peru',
         edgecolor='blue',
         bins=10,
         alpha = 0.3)

plt.show()
```



Multiple histograms are useful in understanding the distribution between 2 entity variables. We can see that GrandCanyon has comparably more visitors than BryceCanyon.

Implementation: Histogram

Pie Chart

It is a circular plot which is divided into slices to illustrate numerical proportion. The slice of a pie chart is to show the proportion of parts out of a whole.

When to use: Pie chart should be used seldom used as It is difficult to compare sections of the chart. Bar plot is used instead as comparing sections is easy.

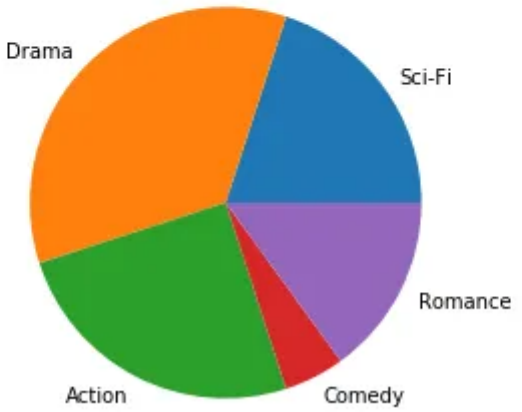
eg: Market share in Films.

Note: Pie Charts is not a good chart to illustrate information.

Pie Chart

Pie chart shows distribution of data based on proportion of pie occupied.

In [3]:
```
plt.pie(t_mov['Percentage'],
        labels=t_mov['Sector'])

plt.axis('equal')

plt.show()
```



Above, `plt.pie()` takes the numeric data as 1st argument i.e Percentage and labels to display as second argument i.e Sector. Ultimately, it shows the distribution of data in proportion to the pie.

Pie chart components

- **wedges:** A list of Patch objects representing each wedge
- **texts:** List of Text objects representing the labels
- **autotexts:** List of Text objects for the numeric values - this is only availabe if the autopct value for the pie chart is not None

In [4]:
```
wedges, texts, autotexts = plt.pie(t_mov['Percentage'],
                                   labels=t_mov['Sector'],
                                   autopct='%.2f')

plt.axis('equal')

print('Wedges: ', wedges)
print('Texts: ', texts)
print('Autotexts: ', autotexts)
```

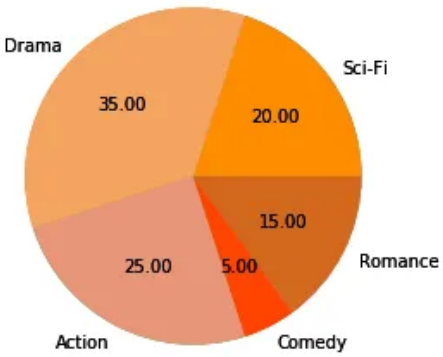
```
Wedges:  [<matplotlib.patches.Wedge object at 0x000001F40DD48BE0>, <matplotlib.patches.Wedge object at 0x00
tplotlib.patches.Wedge object at 0x000001F40DD51B00>, <matplotlib.patches.Wedge object at 0x000001F40DD5B2E
hes.Wedge object at 0x000001F40DD5BA90>]
Texts:  [Text(0.889919,0.646564,'Sci-Fi'), Text(-0.777817,0.777817,'Drama'), Text(-0.49939,-0.980107,'Actio
0.980107,'Comedy'), Text(0.980107,-0.49939,'Romance')]
Autotexts:  [Text(0.48541,0.352671,'20.00'), Text(-0.424264,0.424264,'35.00'), Text(-0.272394,-0.534604,'25
4,-0.534604,'5.00'), Text(0.534604,-0.272394,'15.00')]
```

From above we can the components that make a pie chart and it returns wedge object, text in labels and so on.

Pie chart customizations

We set the values for the colors and autopct properties. The latter sets the format for the values to be displayed.

In [5]:
```
colors = ['darkorange', 'sandybrown', 'darksalmon', 'orangered','chocolate']

plt.pie(t_mov['Percentage'],
        labels=t_mov['Sector'],
        colors=colors,
        autopct='%.2f')

plt.axis('equal')

plt.show()
```



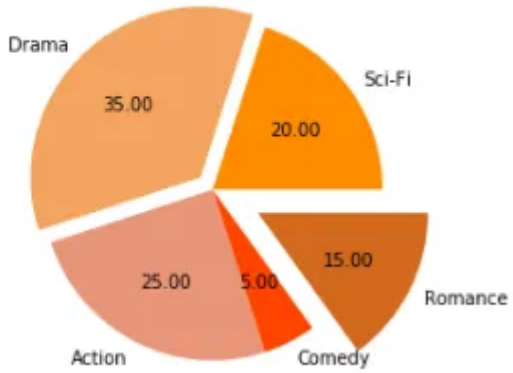
A pie chart can be easily customized and from above color and label values are formatted.

The explode property

To highlight a particular wedge of the pie chart, we use explode to separate it from the rest of the chart.

The value for "explode" represents the fraction of the radius with which to offset each wedge.

In [6]:
```
explode = (0, 0.1, 0, 0, 0.3)

plt.pie(t_mov['Percentage'],
        labels=t_mov['Sector'],
        colors=colors,
        autopct='%.2f',
        explode=explode)

plt.axis('equal')

plt.show()
```



From above explode is used to separate out points from the pie. Similar to a pizza piece being cut.

Implementation: Pie Chart

Time Series by line plot

Time series is a line plot and it is basically connecting data points with a straight line. It is useful in understanding the trend over time. It can explain the correlation between points by the trend. An upward trend means positive correlation and downward trend means a negative correlation. It mostly used in forecasting, monitoring models.

When to use: Time Series should be used when single or multiple variables are to be plotted over time.

eg: Stock Market Analysis of Companies, Weather Forecasting.

In [2]:
```
stock_data.head()
```

Out[2]:

	Date	AAPL	ADBE	CVX	GOOG	IBM	MDLZ	MSFT	NFLX	ORCL	SBUX
0	3-Jan-07	11.107141	38.869999	50.777351	251.001007	79.242500	17.519524	24.118483	3.258571	15.696321	15.752188
1	1-Feb-07	10.962033	39.250000	48.082939	224.949951	74.503204	16.019426	22.092464	3.218571	15.028588	13.930813
2	1-Mar-07	12.037377	41.700001	51.900383	229.309311	75.561348	16.009354	21.857189	3.312857	16.583584	14.138198
3	2-Apr-07	12.930043	41.560001	54.588032	235.925919	81.934280	16.924608	23.480597	3.167143	17.196436	13.984914
4	1-May-07	15.701322	44.060001	57.598267	249.204208	85.786057	17.111704	24.146753	3.128572	17.726965	12.988567

In [3]:
```
stock_data['Date'] = pd.to_datetime(stock_data['Date'])
stock_data.head()
```

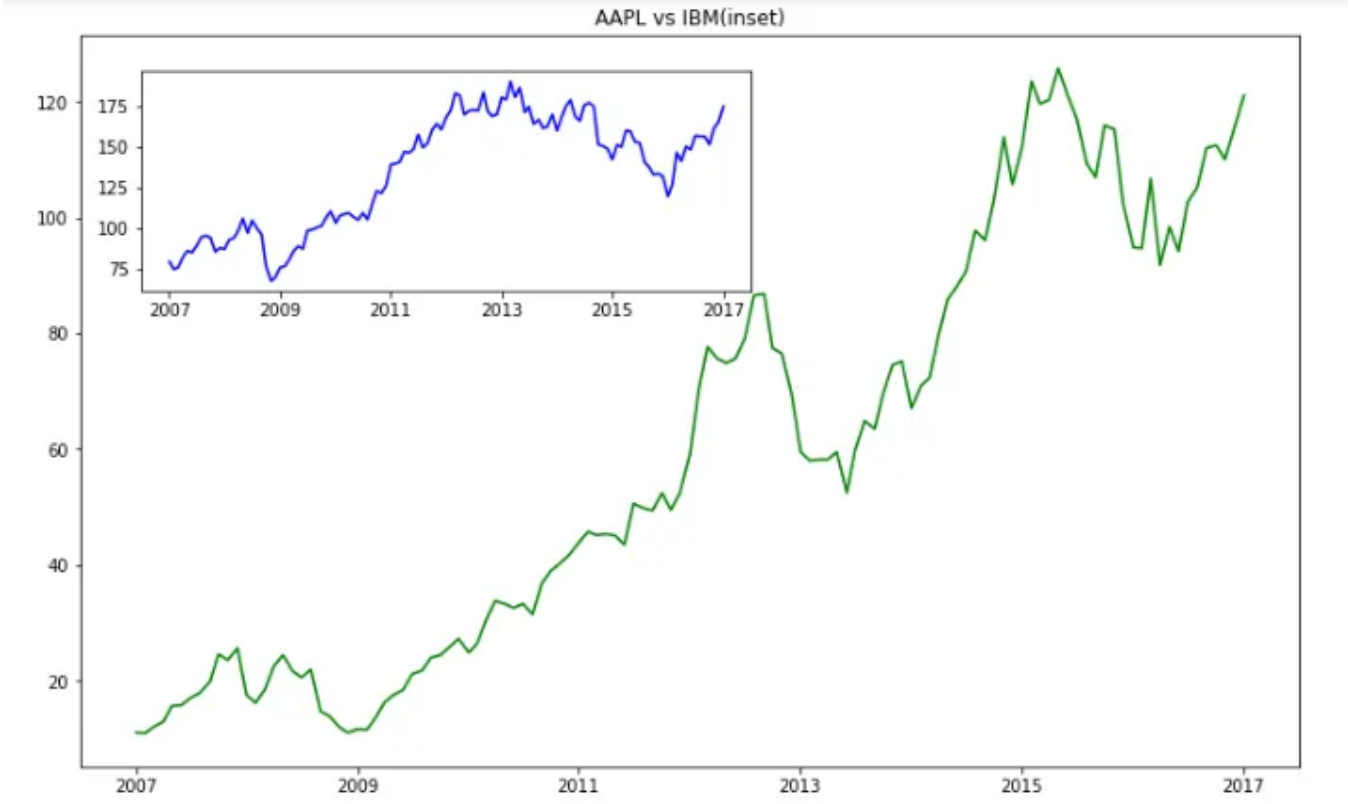
Out[3]:

	Date	AAPL	ADBE	CVX	GOOG	IBM	MDLZ	MSFT	NFLX	ORCL	SBUX
0	2007-01-03	11.107141	38.869999	50.777351	251.001007	79.242500	17.519524	24.118483	3.258571	15.696321	15.752188
1	2007-02-01	10.962033	39.250000	48.082939	224.949951	74.503204	16.019426	22.092464	3.218571	15.028588	13.930813
2	2007-03-01	12.037377	41.700001	51.900383	229.309311	75.561348	16.009354	21.857189	3.312857	16.583584	14.138198
3	2007-04-02	12.930043	41.560001	54.588032	235.925919	81.934280	16.924608	23.480597	3.167143	17.196436	13.984914
4	2007-05-01	15.701322	44.060001	57.598267	249.204208	85.786057	17.111704	24.146753	3.128572	17.726965	12.988567

First, Convert Date to pandas DateTime for easier plotting of data.

Compare Stock side-by-side

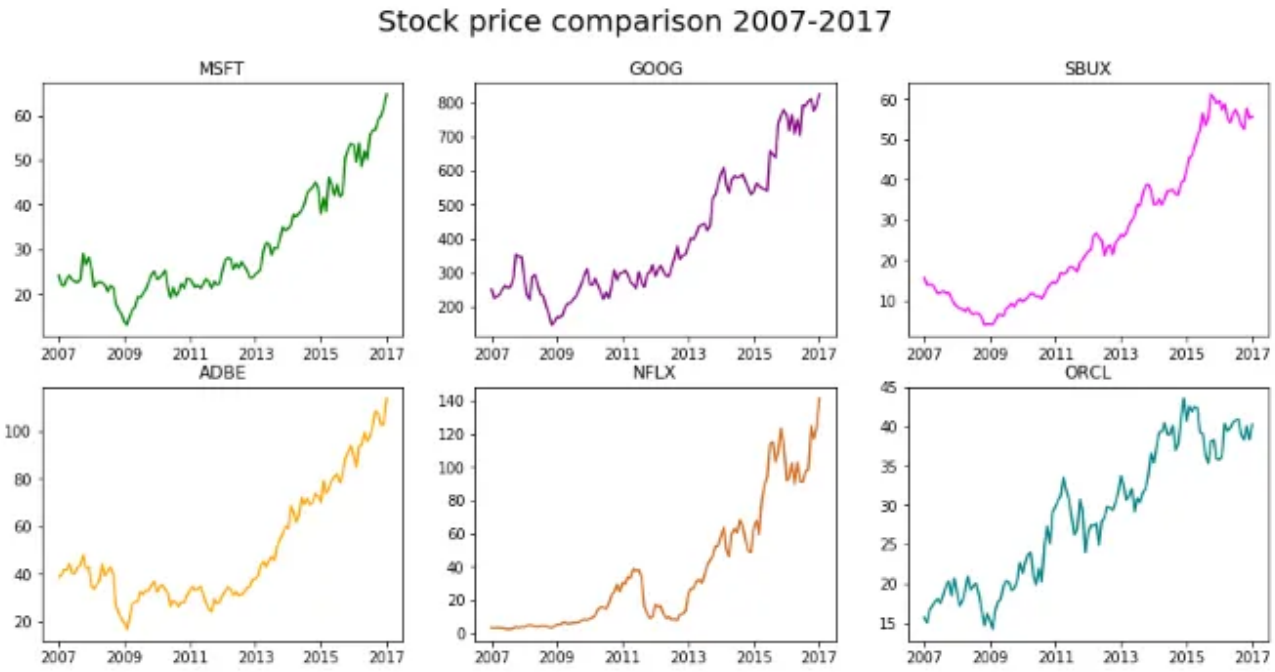
In [4]:
```
fig = plt.figure(figsize=(10,6))

ax1 = fig.add_axes([0, 0, 1, 1])
ax2 = fig.add_axes([0.05, 0.65, 0.5, 0.3])

ax1.set_title('AAPL vs IBM(inset)')

ax1.plot(stock_data['Date'],
         stock_data['AAPL'],
         color='green')

ax2.plot(stock_data['Date'],
         stock_data['IBM'],
         color='blue')
plt.show()
```



From above, `fig.add_axes` is used for plotting the canvas. Check this What are the differences between add_axes and add_subplot? to understand axes and subplots. `plt.plot()` takes the 1st argument as numeric data i.e Date and 2nd argument is to numeric stock data. AAPL Stock is considered as ax1 which is the outer figure and on ax2 IBM Stock is considered for plotting which is inset.

In [15]:

```
# Figsize for width and height of plot
fig = plt.figure(figsize=(15,7))

fig.suptitle('Stock price comparison 2007-2017',
             fontsize=20)

ax1 = fig.add_subplot(231)
ax1.set_title('MSFT')

ax1.plot(stock_data['Date'],
         stock_data['MSFT'],
         color='green')

ax2 = fig.add_subplot(232)
ax2.set_title('GOOG')

ax2.plot(stock_data['Date'],
         stock_data['GOOG'],
         color='purple')

ax3 = fig.add_subplot(233)
ax3.set_title('SBUX')

ax3.plot(stock_data['Date'],
         stock_data['SBUX'],
         color='magenta')

ax4 = fig.add_subplot(234)
ax4.set_title('ADBE')

ax4.plot(stock_data['Date'],
         stock_data['ADBE'],
         color='orange')

ax4 = fig.add_subplot(235)
ax4.set_title('NFLX')

ax4.plot(stock_data['Date'],
         stock_data['NFLX'],
         color='chocolate')

ax4 = fig.add_subplot(236)
ax4.set_title('ORCL')

ax4.plot(stock_data['Date'],
         stock_data['ORCL'],
         color='teal')

```



In the earlier figure,add_axes is used to used to add an axes to a figure whereas from above add_subplot adds multiple subplots to a figure. `fig.add_subplot(237)` cannot be done as there are only 6 subplots possible.

We can see that the tech company stocks are following an upward trend showing positive results for traders to invest in stocks.

Implementation: Time Series

Boxplot and Violinplot

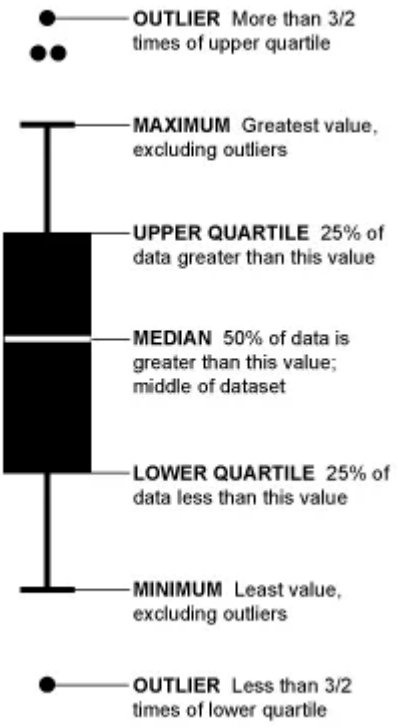
Boxplot

Boxplot gives a nice summary of the data. It helps in understanding our distribution better.

When to use: It should be used when we require to use the overall statistical information on the distribution of the data. It can be used to detect outliers in the data.

eg: Credit Score of Customer. We can get the max, min and much more information about the mark.

Understanding Boxplot



Source: How to Read and Use a Box-and-Whisker Plot

From the above diagram, the line that divides the box into 2 parts represents the median of the data. The end of the box shows the upper quartile(75%)and the start of the box represents the lower quartile(25%). Upper Quartile is also called 3rd quartile and similarly, Lower Quartile is also called as 1st quartile. The region between lower quartile and the upper quartile is called as Inter Quartile Range(IQR) and it is used to approximate the 50% spread in the middle data(75–25=50%). The maximum is the highest value in data, similarly minimum is the lowest value in data, it is also called as caps. The points outside the boxes and between the maximum and maximum are called as whiskers, they show the range of values in data. The extreme points are outliers to the data. A commonly used rule is that a value is an outlier if it's less than lower quartile-1.5 * IQR or high than the upper quartile + 1.5* IQR.

In [3]:
```
exam_scores = exam_data[['math score', 'reading score', 'writing score']]
exam_scores.head()
```

Out[3]:

	math score	reading score	writing score
0	69	61	58
1	47	65	69
2	66	52	53
3	88	89	82
4	62	82	76

In [4]:
```
exam_scores.describe()
```

Out[4]:

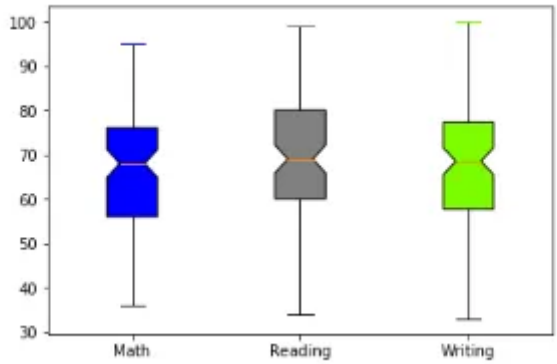
	math score	reading score	writing score
count	100.000000	100.000000	100.000000
mean	67.150000	69.180000	67.780000
std	12.797865	13.832607	14.874954
min	36.000000	34.000000	33.000000
25%	58.000000	60.000000	57.750000
50%	68.000000	69.000000	68.500000
75%	76.000000	80.000000	77.250000
max	95.000000	99.000000	100.000000

In [5]:
```
# For plotting in boxplot, we convert it to an array
exam_scores_array = np.array(exam_scores)
exam_scores_array
```

Out[5]:
```
array([[ 69,  61,  58],
       [ 47,  65,  69],
       [ 66,  52,  53],
       [ 88,  89,  82],
       [ 62,  82,  76],
       [ 47,  69,  60],
       [ 71,  66,  74],
       [ 57,  62,  60],
```

Box plots

In [6]:
```
colors = ['blue', 'grey', 'lawngreen']

bp = plt.boxplot(exam_scores_array,
                 patch_artist=True,
                 notch=True)

for i in range(len(bp['boxes'])):

    bp['boxes'][i].set(facecolor=colors[i])

    bp['caps'][2*i + 1].set(color=colors[i])

plt.xticks([1, 2, 3], ['Math', 'Reading', 'Writing'])

plt.show()
```



bp contains the boxplot components like boxes, whiskers, medians, caps. Seaborn another plotting library makes it easier to build custom plots than matplotlib. `patch_artist` makes the customization possible. `notch` makes the median look more prominent.

A caveat of using boxplot is the number of observations in the unique value is not defined, Jitter Plot in Seaborn can overcome this caveat or Violinplot is also useful

Violin plot

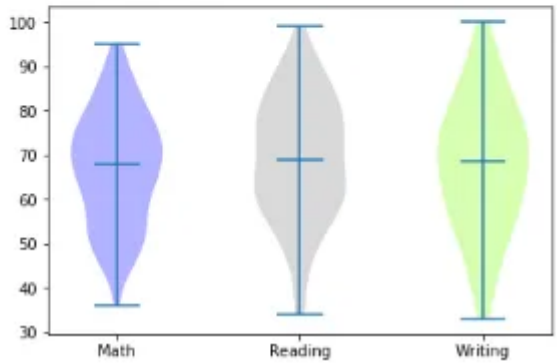
Violin plot is a better chart than boxplot as it gives a much broader understanding of the distribution. It resembles a violin and dense areas point the more distribution of data otherwise hidden by box plots

When to use: Its an extension to boxplot. It should be used when we require a better intuitive understanding of data.

Violin Plots

Similar to boxplots, except they can show the density of the data points around a particular value with their widths

In [7]:
```
vp = plt.violinplot(exam_scores_array,
                    showmedians=True)

plt.xticks([1, 2, 3], ['Math', 'Reading', 'Writing'])

for i in range(len(vp['bodies'])):
    vp['bodies'][i].set(facecolor=colors[i])

plt.show()
```



The density of points in the middle seems more as students tend to score around average mostly in the subjects.

Implementation: Boxplot & Violinplot

TwinAxis

TwinAxis helps in visualizing plotting 2 plots w.r.t to the y-axis and same x-axis.

When to use: It should when we require 2 plots or grouped data in the same direction.

Eg: Population, GDP data in the same x-axis (Date).

Plotting 2 Plots w.r.t the y-axis and same x-axis

```
In [2]:
austin_weather.head()
```

Out[2]:

	Date	TempHighF	TempAvgF	TempLowF	DewPointHighF	DewPointAvgF	DewPointLowF	HumidityHighPercent	HumidityAvgPercent	HumidityLowPercent
0	2013-12-21	74	60	45	67	49	43	93	75	57
1	2013-12-22	56	48	39	43	36	28	93	68	43
2	2013-12-23	58	45	32	31	27	23	76	52	27
3	2013-12-24	61	46	31	36	28	21	89	56	22
4	2013-12-25	58	50	41	44	40	36	86	71	56

5 rows × 21 columns

Extracting Date,Avg Temperature and Avg Wind Speed columns

```
In [3]:
austin_weather = austin_weather[['Date', 'TempAvgF', 'WindAvgMPH']].head(30)
austin_weather
```

Out[3]:

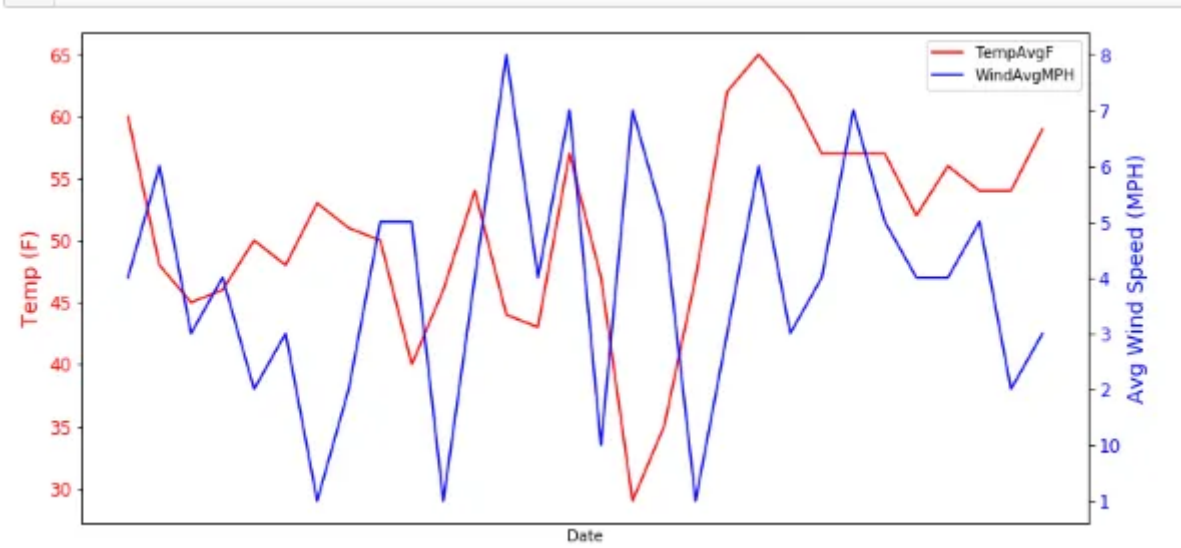
	Date	TempAvgF	WindAvgMPH
0	2013-12-21	60	4
1	2013-12-22	48	6

Extracting important details i.e Date for the x-axis, TempAvgF, and WindAvgMPH for the different y-axis.

In [4]:

```
 # Subplot for Plotting Figure
fig, ax_tempF = plt.subplots()

# Similar to fig=plt.figure(figsize=(12,6))
fig.set_figwidth(12)
fig.set_figheight(6)

# set x label which is common
ax_tempF.set_xlabel('Date')

# bottom= false disables ticks and labelbottom disables x-axis labels
ax_tempF.tick_params(axis = 'x',
                     bottom=False,
                     labelbottom=False)

# set left y-axis label
ax_tempF.set_ylabel('Temp (F)',
                    color='red',
                    size='x-large')

# set labelcolor and labelsize to the left Y-axis
ax_tempF.tick_params(axis='y',
                     labelcolor='red',
                     labelsize='large')

# plot AvgTemp on Y-axis to the left
ax_tempF.plot(austin_weather['Date'],
              austin_weather['TempAvgF'],
              color='red')

#ax_tempF.legend(0)

# twinx sets the same x-axis for both plots
ax_precip = ax_tempF.twinx()

#set Right y-axis label
ax_precip.set_ylabel('Avg Wind Speed (MPH)',
                     color='blue',
                     size='x-large')

# set labelcolor and labelsize to the Right Y-axis
ax_precip.tick_params(axis='y',
```

As we can there is only 1 axis, `twinx()` is used for twinning the x-axis and left y-axis is used for Temp and the right y-axis is used for WindMPH.



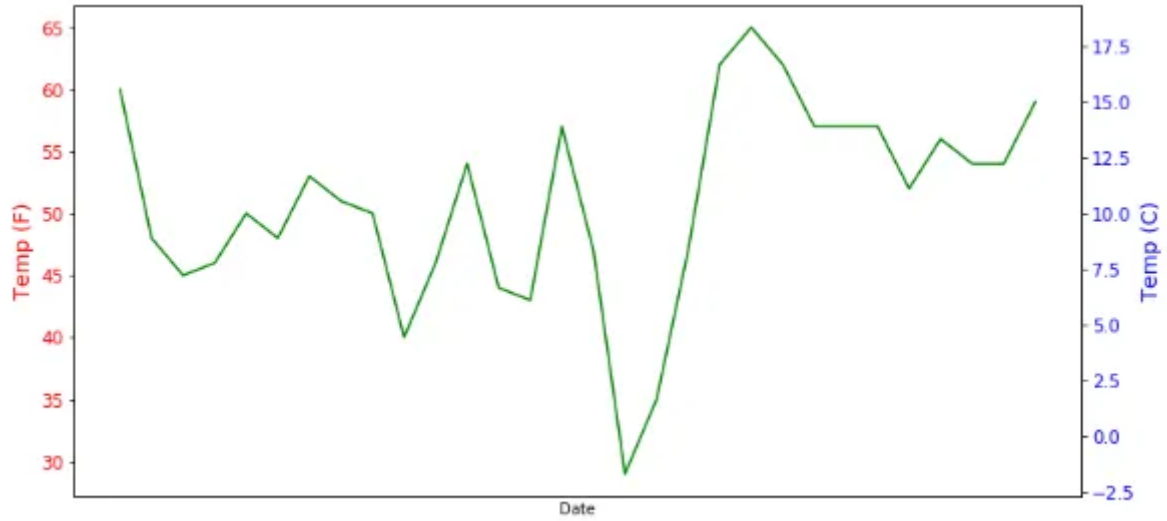
Plotting the same data in different units and the same x-axis

Function to convert from fahrenheit to celsius

In [5]:
```
def fahrenheit2celsius(f):
    return (f - 32)*5/9
```

In [6]:
```
 # Subplot for Plotting Figure
fig, ax_tempF = plt.subplots()

# Similar to fig=plt.figure(figsize=(12,6))
fig.set_figwidth(12)
fig.set_figheight(6)

# set x label which is common
ax_tempF.set_xlabel('Date')

# bottom= false disables ticks and labelbottom disables x-axis labels
ax_tempF.tick_params(axis = 'x',
                     bottom=False,
                     labelbottom=False)

# set left y-axis label
ax_tempF.set_ylabel('Temp (F)',
                    color='red',
                    size='x-large')

# set labelcolor and labelsize to the left Y-axis
ax_tempF.tick_params(axis='y',
                     labelcolor='red',
                     labelsize='large')

# plot AvgTemp on Y-axis to the left
ax_tempF.plot(austin_weather['Date'],
              austin_weather['TempAvgF'],
              color='green')

# twinx sets the same x-axis for both plots
ax_tempC = ax_tempF.twinx()

```

The function is defined for calculating different unit of data i.e convert from Fahrenheit to Celsius.



We can see that to the left y-axis Temp in Fahrenheit is plotted and to the right x-axis Temp in Celsius is plotted.

Implementation: TwinAxis

Stack Plot and Stem Plot

Stack Plot

Stack plot visualizes data in stacks and shows the distribution of data over time.

When to use: It is used for checking multiple variable area plots in a single plot.

Eg: It is useful in understanding the change of distribution in multiple variables over an interval.

```
In [2]:  1 np_data.head()
```

Out[2]:

	Year	Badlands	GrandCanyon	BryceCanyon
0	1961	833300	1253000	264800
1	1962	1044800	1447400	251000
2	1963	1074000	1539500	289500
3	1964	1079800	1576600	300300
4	1965	1091300	1689200	366800

```
In [3]:  1 x = np_data['Year']
```

```
In [4]:  1 y = np.vstack([np_data['Badlands'],
         2                np_data['GrandCanyon'],
         3                np_data['BryceCanyon']])
```

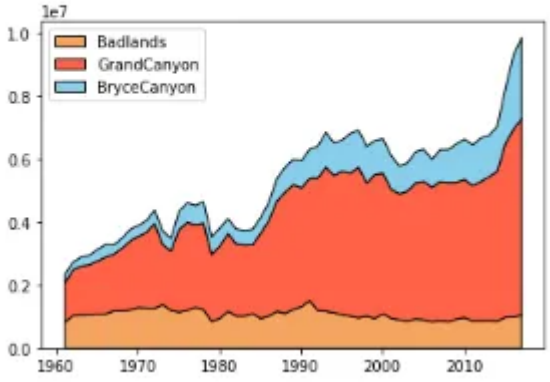
As stack plot requires stacking, it is done in using `np.vstack()`

Stack Plots

It is used stack plots and visualize data overtime.

In [5]:

```
# Labels for each stack
labels = ['Badlands',
          'GrandCanyon',
          'BryceCanyon']

# Colors for each stack
colors = ['sandybrown',
          'tomato',
          'skyblue']

#  Similar to pandas df.plot.area()
plt.stackplot(x, y,
              labels=labels,
              colors=colors,
              edgecolor='black')

# Plots legend to the upperLeft of Figure
plt.legend(loc=2)

plt.show()
```



`plt.stackplot` takes in 1st argument numeric data i.e year and 2nd argument the vertically stacked data i.e the Nationalparks.

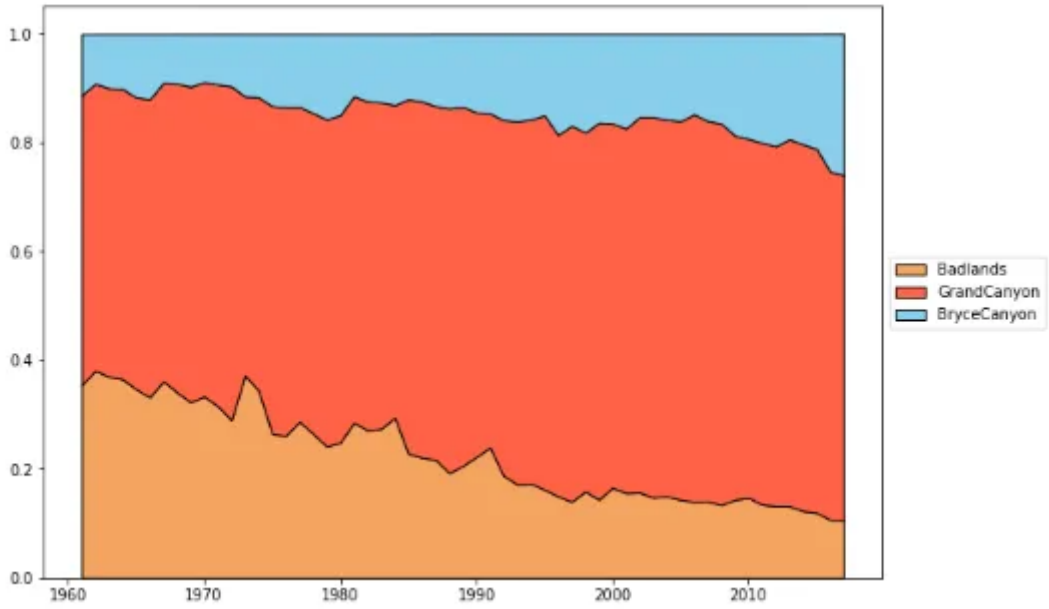
Percentage Stacked plot

Similar to stack plot but each data is converted into a percentage of distribution it holds.

Percentage Stacked Area Chart

Stack plots based on percentage in distribution.

In [6]:
```
plt.figure(figsize=(10,7))

data_perc = np_data.divide(np_data.sum(axis=1), axis=0)

plt.stackplot(x,
              data_perc["Badlands"],data_perc["GrandCanyon"],data_perc["BryceCanyon"],
              edgecolor='black',
              colors=colors,
              labels=labels)

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```



`data_prec` is used to divide the overall percentage into individual percentage distributions. s= `np_data.sum(axis=1)` calculates sum along columns, `np_data.divide(s,axis=0)` divides data along rows.

Stem Plot

Stemplot even takes negative values, so the difference is taken of data and is plotted over time.

When to use: It is similar to a stack plot but the difference helps in comparing the data points.

Stem Plots

Similar to Stack Plots but it helps us view negative numbers, i.e the difference in data over time.

In [7]:
```
np_stem= np_data.copy()
```

In [8]:
```
np_stem[['Badlands',
         'GrandCanyon',
         'BryceCanyon']] = np_data[['Badlands',
                                    'GrandCanyon',
                                    'BryceCanyon']].diff()

np_stem.head()
```

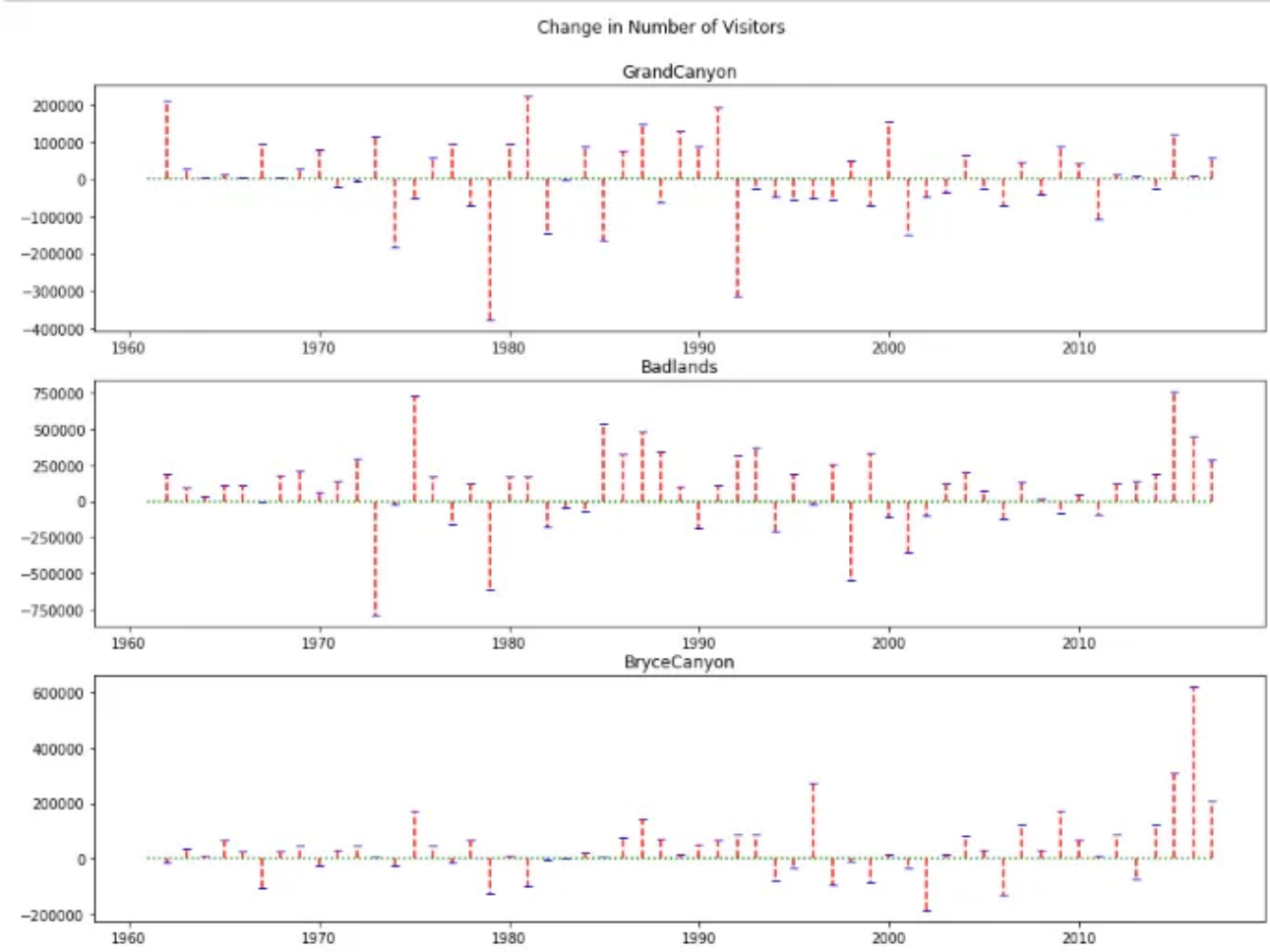
Out[8]:

	Year	Badlands	GrandCanyon	BryceCanyon
0	1961	NaN	NaN	NaN
1	1962	211500.0	194400.0	-13800.0
2	1963	29200.0	92100.0	38500.0
3	1964	5800.0	37100.0	10800.0
4	1965	11500.0	112600.0	66500.0

`diff()` is used to find the difference between previous data and is stored in another copy of the data. The first data point is NaN (Not a Number) as it doesn't contain any previous data for calculating the difference.

In [9]:
```
plt.figure(figsize=(15,11))

plt.suptitle('Change in Number of Visitors', y=0.94)

plt.subplot(311)
plt.stem(np_stem['Year'],
         np_stem['Badlands'],
         markerfmt = 'b_',
         linefmt = 'r--',
         basefmt = 'g:')
plt.title('GrandCanyon')

plt.subplot(312)
plt.stem(np_stem['Year'],
         np_stem['GrandCanyon'],
         markerfmt = 'b_',
         linefmt = 'r--',
         basefmt = 'g:')
plt.title('Badlands')

plt.subplot(313)
plt.stem(np_stem['Year'],
         np_stem['BryceCanyon'],
         markerfmt = 'b_',
         linefmt = 'r--',
         basefmt = 'g:')
plt.title('BryceCanyon')

plt.show()
```

(31n)Subplots are created to accommodate 3 rows 1 column subplots in the figure. `plt.stem()` takes the 1st argument as numeric data i.e year and 2nd argument as numeric data of the National Park visitors.



Implementation: Stack Plot & Stem Plot

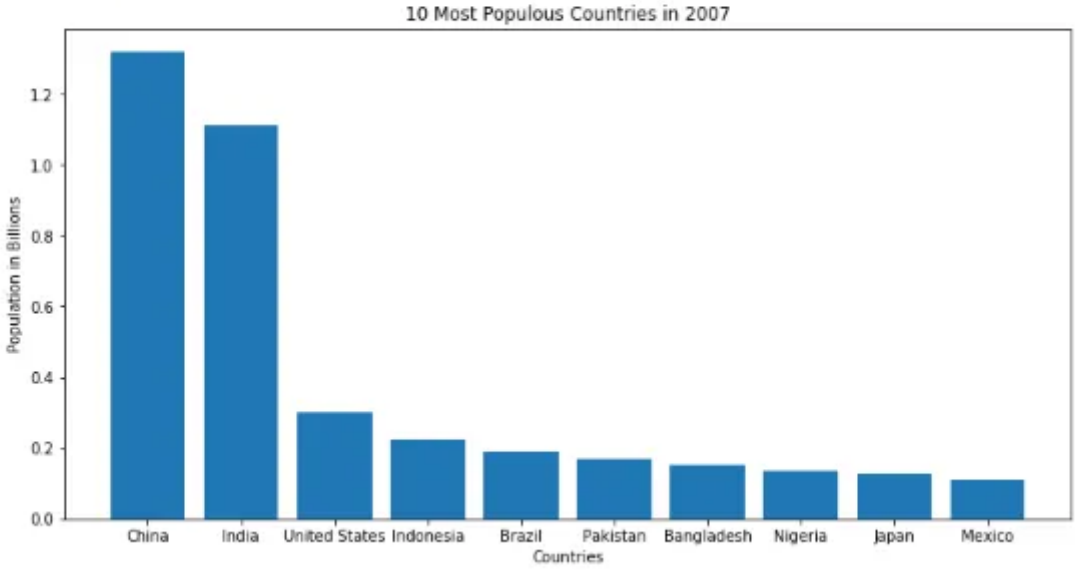
Bar Plot

Bar Plot shows the distribution of data over several groups. It is commonly confused with a histogram which only takes numerical data for plotting. It helps in comparing multiple numeric values.

When to use: It is used when to compare between several groups.

Eg: Student marks in an exam.

```
In [5]:
plt.figure(figsize=(12,6))

x=range(10)
plt.bar(x,top_10_p['population']/10**9)
plt.xticks(x,top_10_p['country'])
plt.xlabel('Countries')
plt.ylabel('Population in Billions')
plt.title('10 Most Populous Countries in 2007')
plt.show()
```



`plt.bar()` takes the 1st argument as labels in numeric format and 2nd argument for the value it represents w.r.t to the plots.

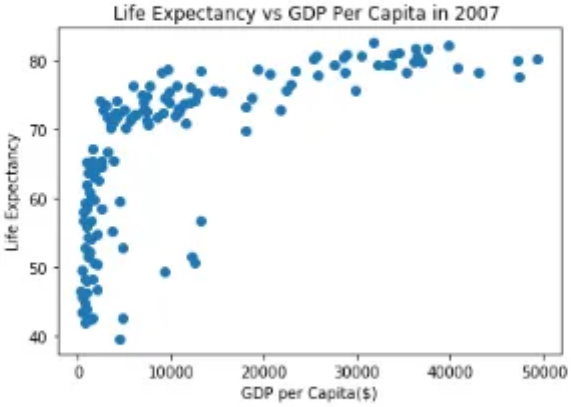
Implementation: Bar Plot

Scatter Plot

Scatter plot helps in visualizing 2 numeric variables. It helps in identifying the relationship of the data with each variable i.e correlation or trend patterns. It also helps in detecting outliers in the plot.

When to use: It is used in Machine learning concepts like regression, where x and y are continuous variables. It is also used in clustering scatters or outlier detection.

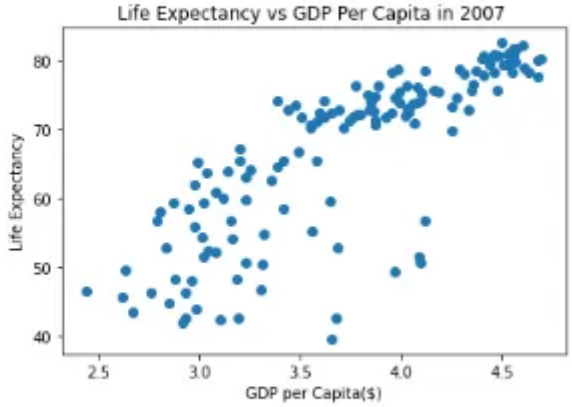
```
In [4]:
plt.scatter(data_2007['gdpPerCapita'],data_2007['lifeExpectancy'])
plt.title('Life Expectancy vs GDP Per Capita in 2007 ')
plt.xlabel('GDP per Capita($)')
plt.ylabel('Life Expectancy')
plt.show()
```




```
In [5]:
data_2007['gdpPerCapita'].corr(data_2007['lifeExpectancy'])
Out[5]: 0.6786623986777587
```

`plt.scatter()` takes 2 numeric arguments for scattering data points in the plot. It is similar to line plot except without the connected straight lines. By corr we mean correlation and it means that how correlated GDP is with life expectancy, as we can see that it is positive it means as GDP of a country increases, life expectancy too increases.

```
In [6]:
plt.scatter(np.log10(data_2007['gdpPerCapita']),data_2007['lifeExpectancy'])
plt.title('Life Expectancy vs GDP Per Capita in 2007 ')
plt.xlabel('GDP per Capita($)')
plt.ylabel('Life Expectancy')
plt.show()
```



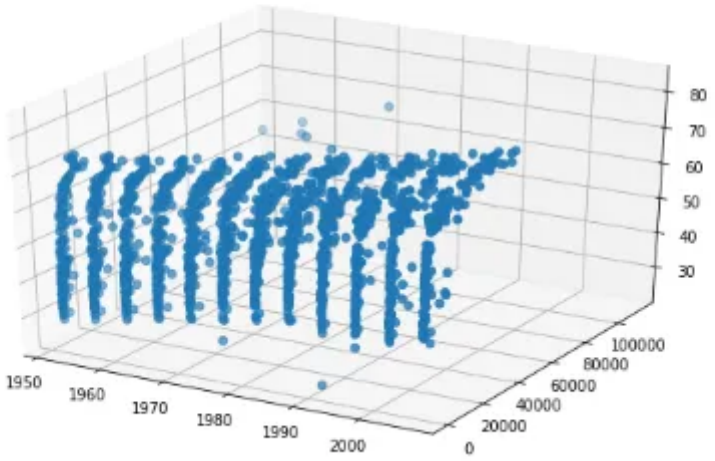

```
In [7]:
np.log10(data_2007['gdpPerCapita']).corr(data_2007['lifeExpectancy'])
Out[7]: 0.8089802514849209
```

By taking the log of GDP, we can there is a much better correlation as we can fit points better, it converts GDP in log scale i.e log($1000)=3.

3D Scatterplot

3D Scatterplot helps in visualizing 3 numerical variables in a three- dimensional plot.

```
In [8]:
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(countries['year'],countries['gdpPerCapita'],countries['lifeExpectancy'], s=30)
plt.show()
```



It is similar to scatter except we add 3 numerical variables this time. By looking at the plot we can make an inference that as the year and GDP increases, life expectancy too increases.

Implementation: Scatter Plot

Find the above code in this Github Repo.

Conclusion

In summary, we learned how to build data visualization plots using one numeric variable and multiple variables. We can now easily build plots for understanding our data intuitively through visualizations.

Data Visualization | Data Science | Matplotlib | Jupyter Notebook | Data Analysis