WMG
THE UNIVERSITY OF WARWICK

Data Visualisation in Python

Table of Contents

1. Data Visualisation in Python 2
1.1. Visualisations with *Matplotlib* 2
1.2. Common Plots in *Matplotlib* 10
1.3. Customising Plots in *Matplotlib* 13
1.4. Recommended resources for further reading on *Matplotlib* 17
1.5. Visualisation with *Pandas* 17
1.6. Visualisation with *Seaborn* 23
1.7. Facet Grids and Matrix Plots in *Seaborn* 53
1.8. Summary 56

1. Data Visualisation in Python

Written by Liping Zheng.

As we covered in the module, data visualisation is the graphical representation of information and data, which can be applied to remove the noise from data, find out the pattern and trend and highlight the useful information to tell stories. Data visualisation can be used for data exploration to help us understand data and quickly get some initial insight from big amount of data or be used to present data analysis result to facilitate decision making. We have been using Tableau to do data visualisation, and here we are going to introduce how we can visualise data using Python.

Python is powerful in terms of data visualisation since it offers multiple great visualisation libraries with different features to fulfil your needs. There are some popular and commonly used plotting libraries in Python, including *Matplotlib*, *Seaborn*, *ggplot*, built-in visualisation in *Pandas* and *Plotly* (among others). In this section, we will introduce *Matplotlib*, *Pandas* visualisation and *Seaborn* (arguably the most commonly used in the data science community).

1.1. Visualisations with *Matplotlib*

Matplotlib is the most popular plotting library in Python. It's originally created by John D. Hunter and has become an active developer community. It provides MATLAB like interface so you might feel natural with *Matplotlib* if you are familiar with MATLAB. *Matplotlib* is an excellent library for both 2D and 3D graphical plotting, and it allows to create animated and interactive visualisation as well. Here we will introduce some basic examples using *Matplotlib*, and I strongly recommend you to explore the *Matplotlib* official webpage (http://matplotlib.org/).

Matplotlib is included as standard in the Anaconda distribution. Should you be using another flavour or Python you will need to install the library:

```
pip install matplotlib
```

After the installation, you need to import the *pyplot* module, which is conventionally done using the name *plt*:

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

You'll also need to use this line to see plots in the Jupyter notebook:

```
%matplotlib inline
```

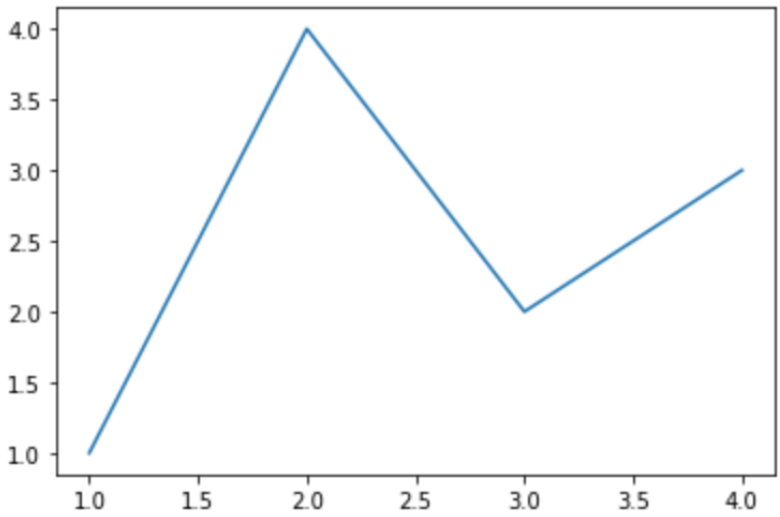
Note, this line is only used for Jupyter Notebooked or other products such as the Jupyter QT Console. If you are using other editor, you'll just use plt.show() at the end of all your plotting commands to have the figure pop up in another window.

Next let's see some simple examples and be familiar with some basic commands of *Matplotlib*. Like many other plotting libraries or languages, *matplotlib.pyplot* module allows to perform the plot directly without explicitly creating an axe. For example, you can create two list as x and y in an axe and graph them directly with `plt.plot()` as shown in **Figure 1**.

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
```

Figure 1: Simple example of visualisation using plt.plot( )

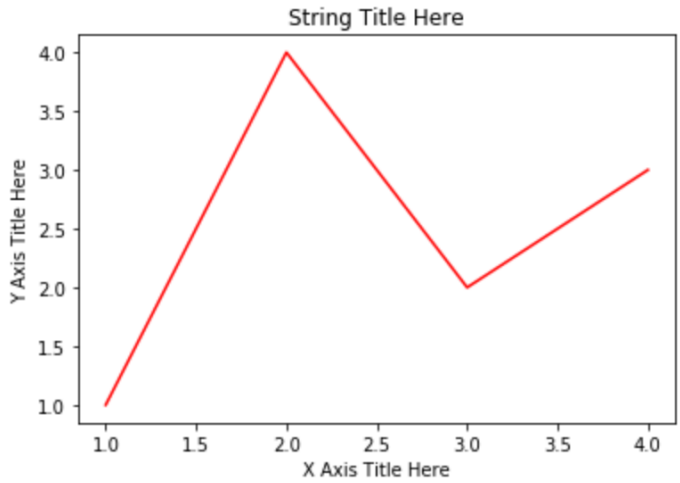
```
In [4]: plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
Out[4]: [<matplotlib.lines.Line2D at 0x2b76a6e0320>]
```



The above is showing one of the approaches to use *Matplotlib*. This approach relies on *pyplot* to automatically create and manage the figures and axes and use *pyplot* functions for plotting. You can set up colour for the plot, as well as the axis labels and title. This is demonstrated in **Figure 2**.

Figure 2: Formatting the plot

```
In [6]:  plt.plot([1, 2, 3, 4], [1, 4, 2, 3], 'r') # 'r' is the colour red
         plt.xlabel('X Axis Title Here')
         plt.ylabel('Y Axis Title Here')
         plt.title('String Title Here')
         plt.show()
```



You can also create multi-plots on the same canvas by either using *plt.subplot()* to create two plots in grid or using *plt.plot()* sequentially with different data to create combo plot. This is shown in **Figure 3** and **Figure 4**.

Figure 3: Multi-plotting example #1

```
In [8]:  # plt.subplot(nrows, ncols, plot_number)
         plt.subplot(1,2,1) # We create 1 row with two columns, and here is the plot 1
         plt.plot([1, 2, 3, 4], [1, 4, 2, 3], 'r--') # '--'means the shape of the line
         plt.subplot(1,2,2) # We create 1 row with two columns, and here is the plot 2
         plt.plot( [1, 4, 2, 3],[1, 2, 3, 4], 'g*-'); # '*-'means the shape of the line
```

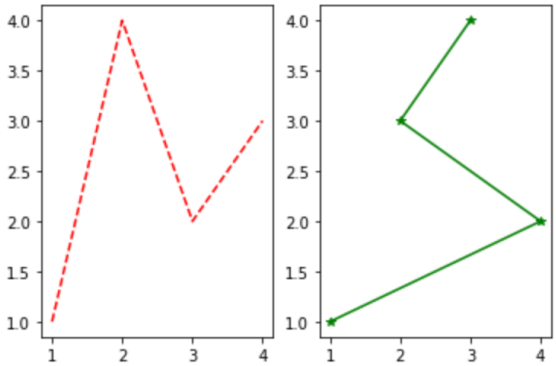
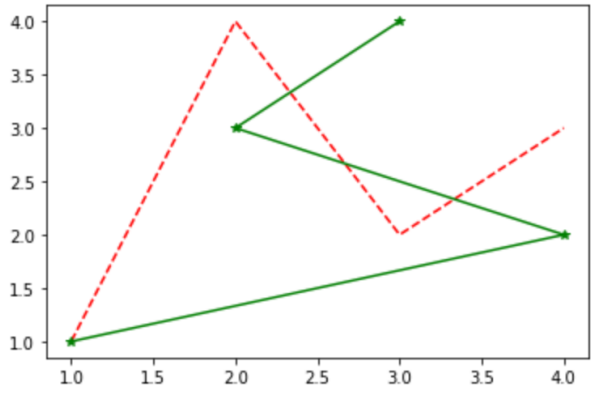


Figure 4: Multi-plotting example #2

```
In [9]: plt.plot([1, 2, 3, 4], [1, 4, 2, 3], 'r--')
        plt.plot( [1, 4, 2, 3],[1, 2, 3, 4], 'g*-')
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x2b76a8802b0>]
```



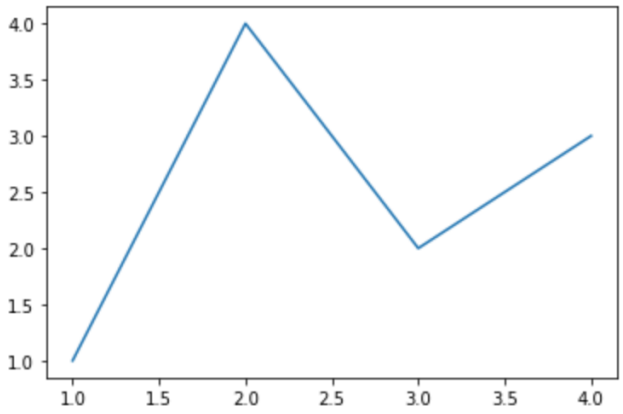
There is another way to use *Matplotlib* by explicitly creating figures and axes and call methods on them. This is what we would commonly call an object-oriented approach. In the practice, this method is more commonly used since this is nicer when dealing with a canvas that has multiple plots on it.

Generally, *Matplotlib* graphs your data on figure(s), each of which can contain one or more Axes. The simplest way to create a figure with an axe is using *plt.subplots()* and then use *ax.plot()* to draw some data on the axes. The above example can be also written by this way with the same output:

Figure 5: Create figures and axes using an object-orientated approach

```
In [5]: fig, ax = plt.subplots()  # Create a figure containing a single axes.
        ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Plot some data on the axes.
```

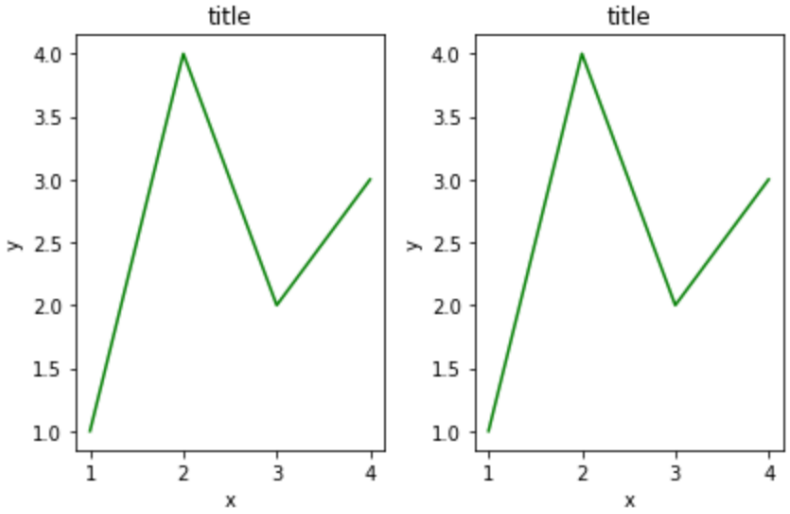
```
Out[5]: [<matplotlib.lines.Line2D at 0x2b76a5af710>]
```



You can also create multi-plots in this object-orientated approach:

Figure 6: Creating multi-plots using an object-orientated approach

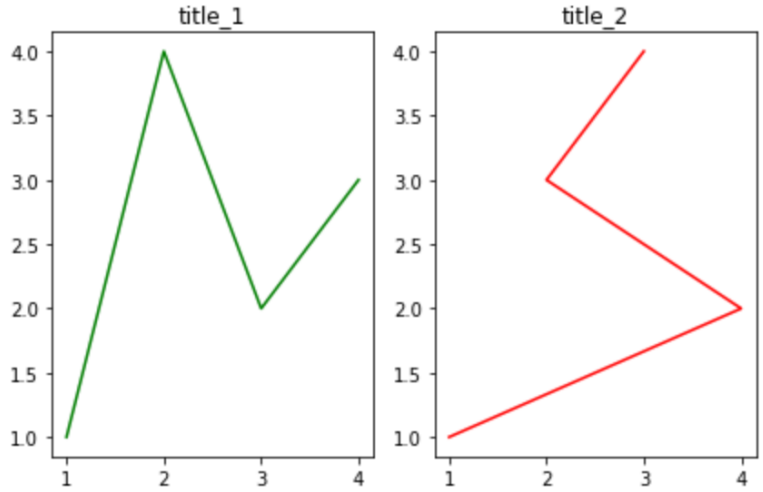
In [19]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot([1, 2, 3, 4], [1, 4, 2, 3], 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig # Display the figure object
plt.tight_layout() # avoid overlapping content
```



Another (more useful!) variant on this would be to use different data on each plot, as shown in **Figure 7**.

Figure 7: Creating multi-plot figures with different data

```
In [20]:
fig, (ax1, ax2) = plt.subplots(1, 2)

ax1.plot([1, 2, 3, 4], [1, 4, 2, 3], 'g')
ax1.set_title('title_1')
ax2.plot([1, 4, 2, 3],[1, 2, 3, 4],'r')
ax2.set_title('title_2')

fig # Display the figure object
plt.tight_layout() # avoid overlapping content
```

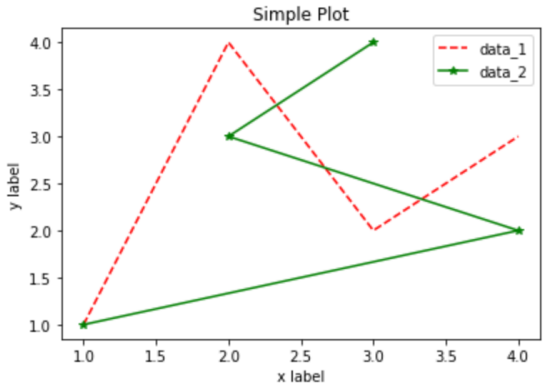


We can also create a single figure with both sets of data (using the same axes):

Figure 8: Creating a figure with two datasets on the same axes

```
In [35]:
fig, ax = plt.subplots()  # Create a figure and an axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3], 'r--',label='data_1')  # Plot some data on the axes.
ax.plot([1, 4, 2, 3],[1, 2, 3, 4], 'g*-',label='data_2')  # Plot more data on the axes...
ax.set_xlabel('x label')  # Add an x-label to the axes.
ax.set_ylabel('y label')  # Add a y-label to the axes.
ax.set_title("Simple Plot")  # Add a title to the axes.
ax.legend()  # Add a legend.
```

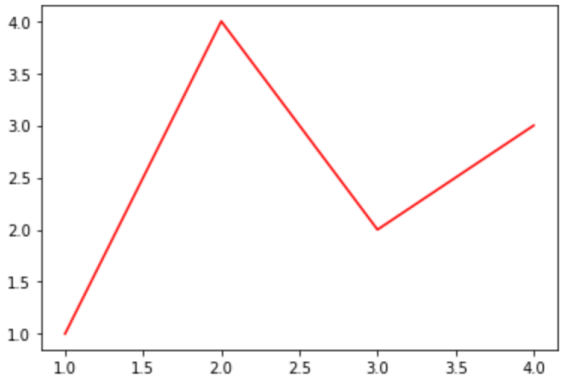
```
Out[35]: <matplotlib.legend.Legend at 0x2b76c30d5c0>
```



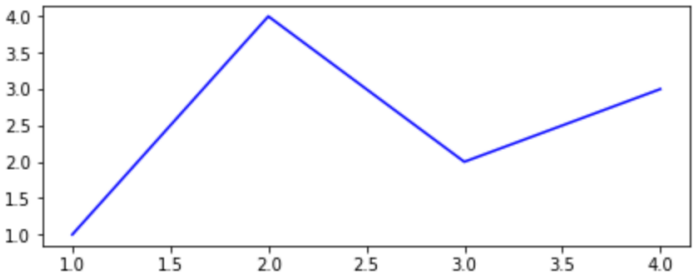
It is typically more convenient to create the axes at the same time as the figure, but they can also be added later in the code, allowing for more complex axes and layouts:

Figure 9: Creating figures and axes separately

```
In [12]: # Create Figure (empty canvas)
         fig = plt.figure()

         # Add set of axes to figure
         axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)

         # Plot on that set of axes
         axes.plot([1, 2, 3, 4], [1, 4, 2, 3], 'r') # r is colour red

Out[12]: [<matplotlib.lines.Line2D at 0x2b76aa72c50>]
```



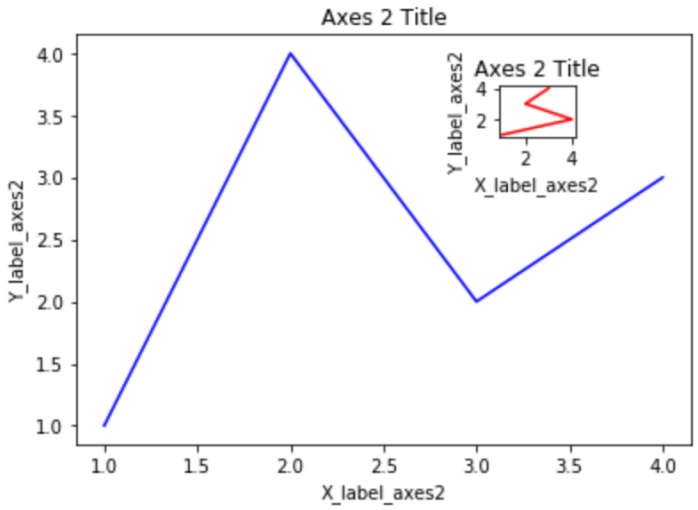

```
In [21]: fig = plt.figure()
         axes = fig.add_axes([0.1, 0.1,0.9, 0.5])
         # Plot on that set of axes
         axes.plot([1, 2, 3, 4], [1, 4, 2, 3], 'b') # b is colour blue

Out[21]: [<matplotlib.lines.Line2D at 0x2b76aca0198>]
```



We can plot axes inside other axes (e.g. a chart inside a chart) as shown in **Figure 10**.

Figure 10: Inserting an axes inside an axes

In [32]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.65, 0.7, 0.1, 0.1]) # inset axes

# Larger Figure Axes 1
axes1.plot([1, 2, 3, 4], [1, 4, 2, 3], 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot([1, 4, 2, 3],[1, 2, 3, 4], 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title');
```



After seeing some examples, let's explore in more depth the components of a *Matplotlib* chart.

Figure is the whole chart which can contain any number of axes but at least one. The below is the easiest way to create a figure with *pyplot*:

```
fig = plt.figure()  # an empty figure with no Axes

fig, ax = plt.subplots()  # a figure with a single Axes

fig, axs = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes
```

Axes are what we call "a plot". The axe contains two (or three in the case of 3D graph) axis objects (x,y or x,y,z).

Axis is the number-line-like object (e.g., x-axis and y-axis in a 2D graph). They take care of setting the graph limits and generating the ticks (the marks on the axis) and tick labels (strings labelling the ticks).

1.2. Common Plots in *Matplotlib*

Alongside line charts, which we demonstrated in section 6.1, *Matplotlib* has a wide array of in-built chart types. In this part, we will use the Happiness report data (available from the Github). First, let's import the data using the approaches discussed in chapter five.

Figure 11: Importing the Happiness dataset

```
In [36]: import pandas as pd
         happiness = pd.read_csv('Happiness report-2015.csv')
         happiness.head()
```

Out[36]:

	Country	Region	Happiness Rank	Happiness Score	Standard Error	Economy (GDP per Capita)	Family	Health (Life Expectancy)	Freedom	Trust (Government Corruption)	Generosity	Dystopia Residual
0	Switzerland	Western Europe	1	7.587	0.03411	1.39651	1.34951	0.94143	0.66557	0.41978	0.29678	2.51738
1	Iceland	Western Europe	2	7.561	0.04884	1.30232	1.40223	0.94784	0.62877	0.14145	0.43630	2.70201
2	Denmark	Western Europe	3	7.527	0.03328	1.32548	1.36058	0.87464	0.64938	0.48357	0.34139	2.49204
3	Norway	Western Europe	4	7.522	0.03880	1.45900	1.33095	0.88521	0.66973	0.36503	0.34699	2.46531
4	Canada	North America	5	7.427	0.03553	1.32629	1.32261	0.90563	0.63297	0.32957	0.45811	2.45176

Since in the original dataset the column names include spaces and brackets, it would be sensible to fix the column names to make it easier to call the column name later when creating plots.

Figure 12: The transformed Dataframe

```
In [39]: happiness.columns = happiness.columns.str.strip().str.lower().str.replace(' ', '_').str.replace('(', '').str.replace(')', '')
         happiness.head()
```

Out[39]:

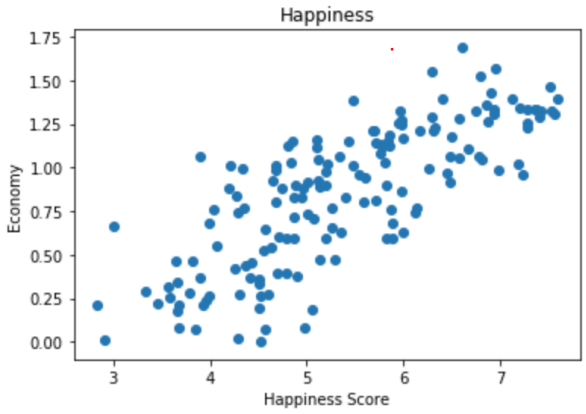
	country	region	happiness_rank	happiness_score	standard_error	economy_gdp_per_capita	family	health_life_expectancy	freedom	trust_govern
0	Switzerland	Western Europe	1	7.587	0.03411	1.39651	1.34951	0.94143	0.66557	
1	Iceland	Western Europe	2	7.561	0.04884	1.30232	1.40223	0.94784	0.62877	
2	Denmark	Western Europe	3	7.527	0.03328	1.32548	1.36058	0.87464	0.64938	
3	Norway	Western Europe	4	7.522	0.03880	1.45900	1.33095	0.88521	0.66973	
4	Canada	North America	5	7.427	0.03553	1.32629	1.32261	0.90563	0.63297	

We can use *Matplotlib* to create a scatter plot to examine the relationship between happiness score and economy. This is easy since we already know how to create figure, and all we need do is replace `ax.plot()` with `ax.scatter()`.

Figure 13: Creating a scatter plot

```
In [40]:
# create a figure and axis
fig, ax = plt.subplots()

# scatter the sepal_length against the sepal_width
ax.scatter(happiness["happiness_score"], happiness["economy_gdp_per_capita"])
# set a title and labels
ax.set_title('Happiness')
ax.set_xlabel('Happiness Score')
ax.set_ylabel('Economy')
```

```
Out[40]: Text(0, 0.5, 'Economy')
```

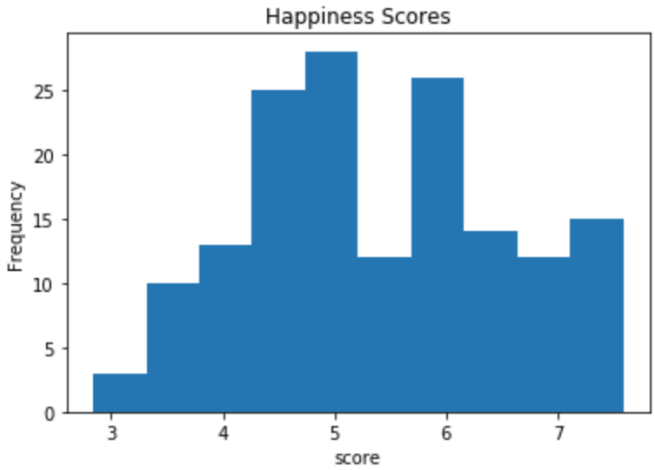


We can also create a Histogram using the `hist()` method. Histogram is useful for checking the distribution of the data. **Figure 14** shows the frequency distribution of the happiness score for different countries:

Figure 14: Creating a histogram

```
In [44]:
# create figure and axis
fig, ax = plt.subplots()
# plot histogram
ax.hist(happiness["happiness_score"])
# set title and labels
ax.set_title('Happiness Scores')
ax.set_xlabel('score')
ax.set_ylabel('Frequency')
```

```
Out[44]: Text(0, 0.5, 'Frequency')
```

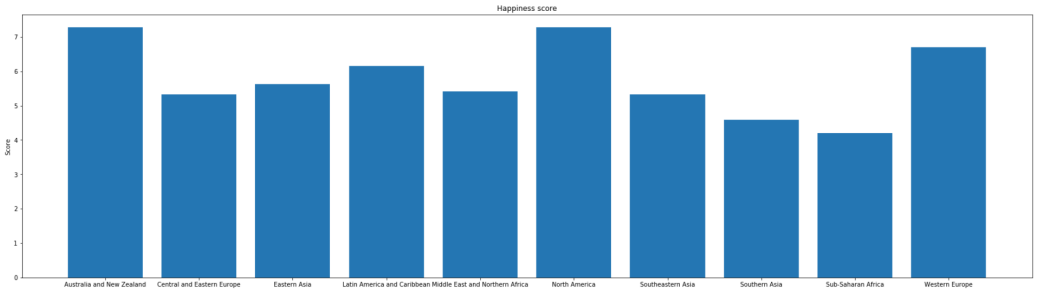


A bar chart can be created using the `bar()` method. A bar chart is useful for categorical data that doesn't have a lot of different categories (e.g. less than 30) because else it can get quite messy. Here we use a bar chart to show the happiness score of different regions. Because the bar chart isn't automatically calculating the average happiness score, we group the data first to get the x (regions) and y (mean happiness score based on region) data (**Figure 15**).

Figure 15: Creating a bar chart

```
In [62]:
data = happiness.groupby('region', as_index=False).agg({"happiness_score": "mean"})

# create a figure and axis
fig, ax = plt.subplots(figsize=(30,8))

# create bar chart
ax.bar(data["region"], data["happiness_score"])
# set title and labels
ax.set_title('Happiness score')
ax.set_ylabel('Score')
```

```
Out[62]: Text(0, 0.5, 'Score')
```



1.3. Customising Plots in *Matplotlib*

Matplotlib allows figure size and DPI (dots per inch) to be specified when the object is created. Figure size is tuple of width and height of the figure in inches, while DPI is effectively the number of pixels per inch. You can use the `figsize` and `dpi` keyword argument. For example:

```
fig = plt.figure (figsize=(4,6), dpi=100)
```

Or

```
fig, axes = plt.subplots(figsize=(4,6),dpi=100)
```

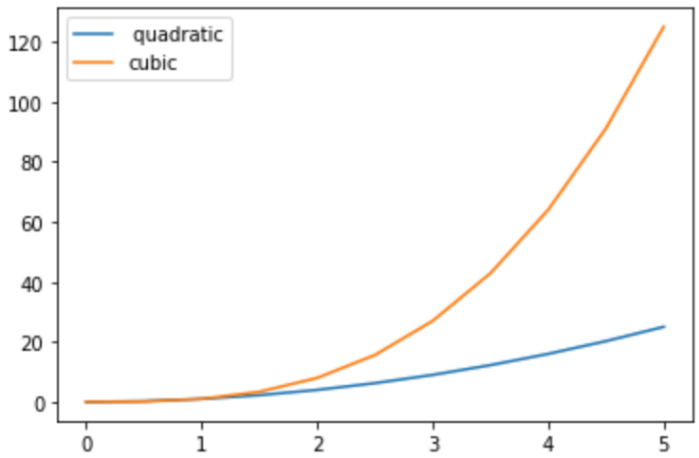
As mentioned in the previous examples, all the axis labels, titles and legends can be added to the figure. To set the figure title, use the `set_title` method in the axes instance: `ax.set_title("title")`. To set the x and y axis labels, use the methods `set_xlabel` and `set_ylabel` in the axes instance: `ax.set_xlabel("x"); ax.set_ylabel("y")`. To add legends, it's a bit different. You need to add the `label = "label text"` keyword argument first when plots or other objects are added to the figure, and then use the legend method without arguments to add the legend to the figure. Below is the example of setting legends:

```
Fig,ax = plt.subplots()

ax.plot(x, x**2, label=" quadratic")

ax.plot(x, x**3, label="cubic")

ax.legend()
```

Figure 16: Setting the legend for a chart

```
In [65]:
import numpy as np
x = np.linspace(0, 5, 11)

Fig,ax = plt.subplots()

ax.plot(x, x**2, label=" quadratic")
ax.plot(x, x**3, label="cubic")
ax.legend()
```

```
Out[65]: <matplotlib.legend.Legend at 0x2b77164f6a0>
```



Note that you can also adjust where in the figure the legend is to be drawn. In the above example, it's default setting but sometimes it might overlap with the plot. Here we can use loc keyword argument to make change. The commonly used is loc =0, which means the Matplotlib will decide the optimal location. But there are also more options to choose. See the documentation page for details. Here are some common examples for you:

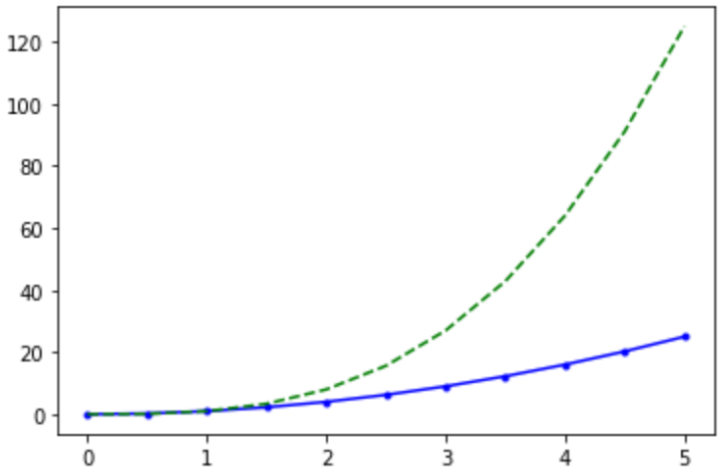
```
ax.legend(loc=0) # let matplotlib decide the optimal location

ax.legend(loc=1) # upper right corner

ax.legend(loc=2) # upper left corner

ax.legend(loc=3) # lower left corner

ax.legend(loc=4) # lower right corner
```

There are different ways to set up colour for your plot. You can use MATLAB like syntax by adding colour symbols when plotting. For example, we use "*b*" to mean blue while "*r*" for red and "*g*" for green. After the colour symbol you can set the line style as well. For example, "*b.-*" means blue line with dots while "*b* --" means blue dashed line.

Figure 17: Setting the line colour and style (method #1)

```
In [67]: # MATLAB style line color and style
         fig, ax = plt.subplots()
         ax.plot(x, x**2, 'b.-') # blue line with dots
         ax.plot(x, x**3, 'g--') # green dashed line
```

```
Out[67]: [<matplotlib.lines.Line2D at 0x2b77130b208>]
```

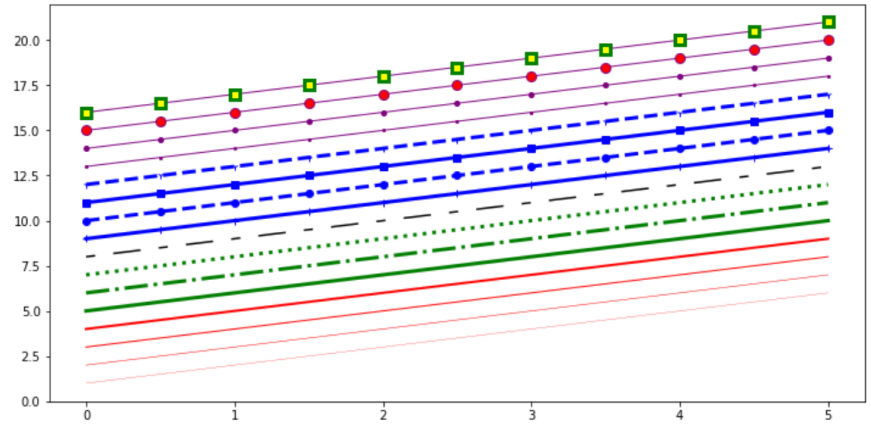


You can also define colours by their name or RGB hex codes and optionally provide an `alpha` value (alpha indicates opacity) using colour and alpha keyword arguments.

In addition to colour setting, you can also use `linewidth` or `lw` keyword argument to change the line width and use the `linestyle` or `ls` keyword argument to change the line style.
Furthermore, you can use marker keyword argument to change the shape of the line.

Figure 18: Setting the line colour and style (method #2)

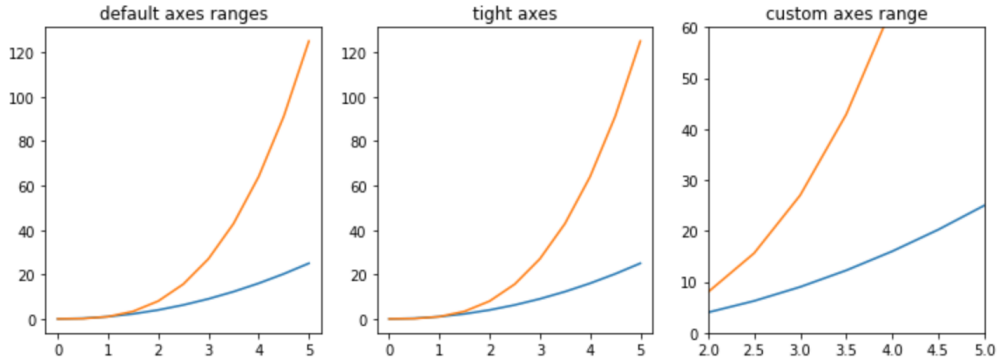
```
In [69]: fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)

# possible linestype options '-', '--', '-.', ':', 'steps'
ax.plot(x, x+5, color="green", lw=3, linestyle='-')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4', ...
ax.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```



You can configure the range of the axis by using the `set_xlim` and `set_ylim` methods in the axes object or using `axis('tight')` to automatically generate tightly fitted axes ranges:

Figure 19: Setting the range of axes

```
In [70]:
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



To save a figure to a file we can use the `savefig` method in the Figure class. *Matplotlib* can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. You can also optionally specify the DPI and choose between different output formats.

```
fig.savefig("filename.png")
```

```
fig.savefig("filename.png", dpi=200)
```

1.4. Recommended resources for further reading on *Matplotlib*

http://www.matplotlib.org - The project web page for matplotlib.

https://github.com/matplotlib/matplotlib - The source code for matplotlib.

http://matplotlib.org/gallery.html - A large gallery showcasing various types of plots matplotlib can create. Highly recommended!

1.5. Visualisation with *Pandas*

Pandas, as we explored in chapter five, is a widely-used library that provides data structures such as Dataframes. It also provides data analysis tools like the visualisation that we are going to cover in this section. *Pandas* built-in visualisation is built on *Matplotlib* and it's very easy to use. It utilises a higher level API than *Matplotlib* which allows us to use shorter code to get the same results. Before you start doing visualisation with *Pandas*, make sure you have installed and imported the

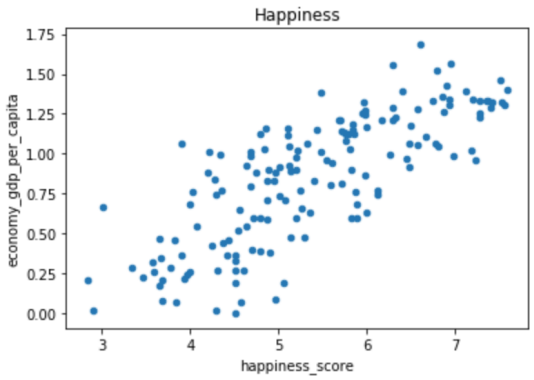
package. If you have followed the previous chapter, or are using the Anaconda distribution of Python, this should already be the case.

When we were creating sample plots in *Matplotlib*, we imported the happiness report dataset into a Dataframe. As we are already using *Pandas* to create the Dataframe, it is probably more efficient to keep using this package rather than switch out to *Matplotlib*. Again, we can create line charts, bar charts, scatter plots and histograms directly with Pandas without creating figure and axes. In this section, some examples will be given using the same dataset we used in section 6.2 (the Happiness Report-2015).

To create a scatter plot, we can use `df.plot.scatter()` to quickly create scatter plot based on the Dataframe. All we need to do is to specify the columns we wish to use as x and y, and optionally you can pass the title name.

Figure 20: Creating a scatter plot using Pandas

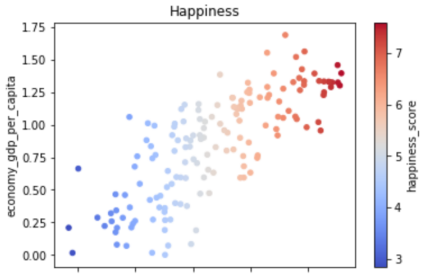
```
In [72]: happiness.plot.scatter(x="happiness_score",y="economy_gdp_per_capita",title='Happiness')
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x2b77152e128>
```



In addition, you can use `c` to set the colour of the scatter plots based on another column and use `cmap` to indicate colour map to use. For all the colour maps, check out: http://matplotlib.org/users/colormaps.html.

Figure 21: Setting a colourmap for a scatter plot in Pandas

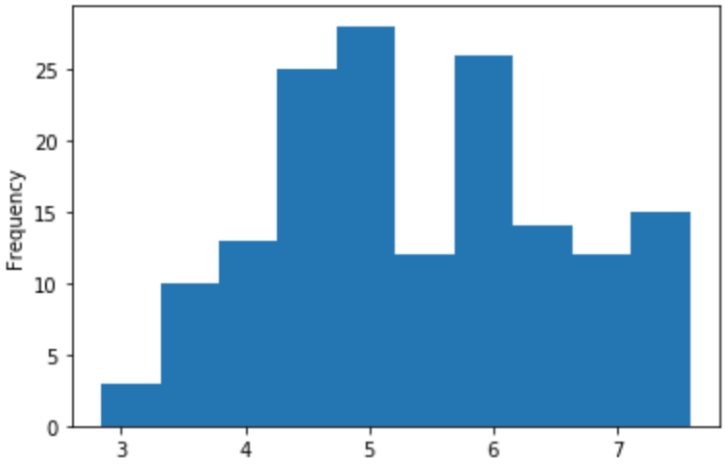
```
In [75]: happiness.plot.scatter(x="happiness_score",y="economy_gdp_per_capita",c= "happiness_score", cmap='coolwarm',title='Happiness')
Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0x2b772b44ef0>
```



Histograms can also be easily created using `df[column_name].plot.hist()`:

Figure 22: Creating a histogram in Pandas

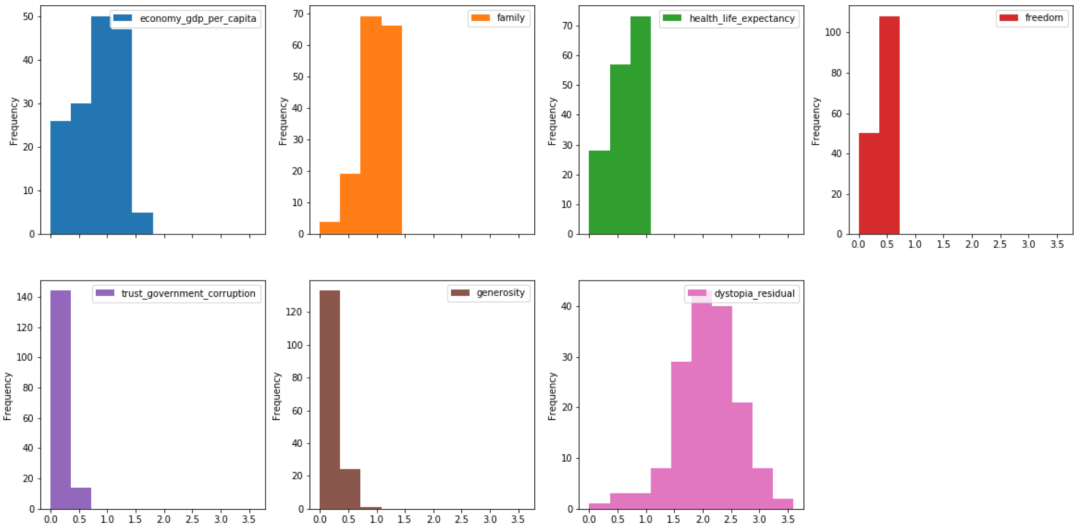
```
In [76]: happiness["happiness_score"].plot.hist()
Out[76]: <matplotlib.axes._subplots.AxesSubplot at 0x2b771b5afd0>
```



You can also easily create multiple histograms using the dataset. The subplots argument specifies that we want a separate plot for each feature and the layout specifies the number of plots per row and column.

Figure 23: Creating multiple histograms in Pandas

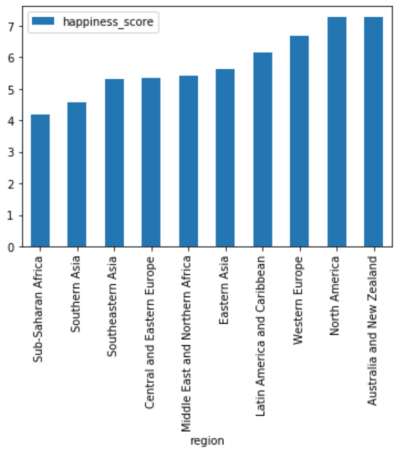
```
In [82]: df = happiness.drop(['country','region','happiness_rank','happiness_score','standard_error'],axis=1)
         df.plot.hist(subplots=True, layout=(2,4), figsize=(20, 10))

Out[82]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000002B7740F51D0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B7741070F0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B77412A518>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B774159978>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000002B77418CDD8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B7741C9240>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B7741FB6A0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000002B77422DB38>]],
                dtype=object)
```



To create bar plot, we can use `df.plot.bar()` method. First, we group the data by the regions, sort the values from smallest to the biggest and then show them in a bar chart.

Figure 24: Creating a bar chart in Pandas

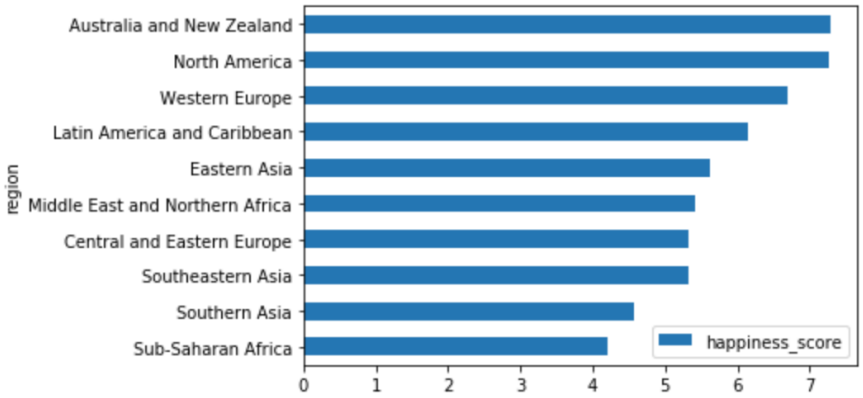
```
In [88]: data = happiness.groupby('region', as_index=True).agg({"happiness_score": "mean"}).sort_values(by='happiness_score')

         data.plot.bar()

Out[88]: <matplotlib.axes._subplots.AxesSubplot at 0x2b774917fd0>
```



You can change the vertical bar chart to a horizontal bar chart using `df.plot.barh()` method.

Figure 25: Creating a horizontal bar chart in Pandas

```
In [89]: data.plot.barh()
Out[89]: <matplotlib.axes._subplots.AxesSubplot at 0x2b774bfab70>
```

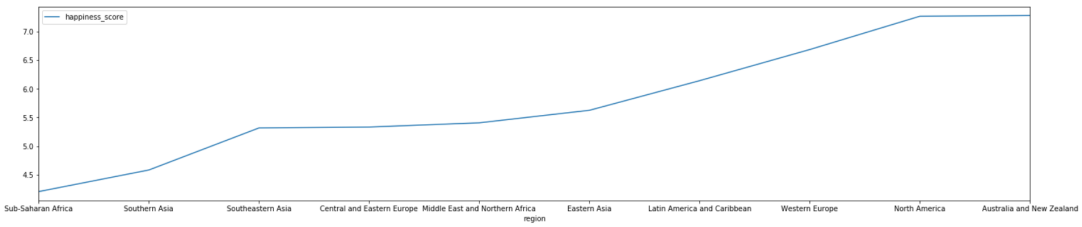


You can also set the colour for your bar chart using `color = [colour name]`.

Line charts in *Pandas* can be easily created using the `df.plot.line()` method. In *Pandas*, if you don't specify which column you want to create the line chart, it automatically plots all available numeric columns of the whole dataset.

Figure 26: Creating a line chart in Pandas

```
In [98]: data.plot.line(figsize=(25,5))
Out[98]: <matplotlib.axes._subplots.AxesSubplot at 0x2b776601e80>
```



Note: this is not the best example as the dataset is more suited to bar charts (and similar) than line charts. A line chart would be more suited to use-cases such as time-series data.

There are also more other chart types that are available in Pandas built-in visualisation. Here are the types that might be commonly used in the practice:

1. df.plot.area
2. df.plot.barh

3. df.plot.density
4. df.plot.hist
5. df.plot.line
6. df.plot.scatter
7. df.plot.bar
8. df.plot.box
9. df.plot.hexbin
10. df.plot.kde
11. df.plot.pie

You can also just call `df.plot(kind='hist')` or replace that kind argument with any of the key terms shown in the list above (e.g. `'box'`, `'barh'`, etc.)

Similar to what can do in *Matplotlib*, you can have settings for your plot with *Pandas* visualisation, such as setting titles, legends, axis labels etc. There are different ways to realise these. You can set the arguments directly when you are creating the plot. Here is an example:

```
data.plot(kind='barh',color='red',xlim=(0,8),title='Happiness_report',figsize=(5,5), legend=False)
```

For more details of the arguments, you can explore `pandas.DataFrame.plot` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html).

Additionally, you can also stick with the approaches used for *Matplotlib* for setting or formatting your plots. For example:

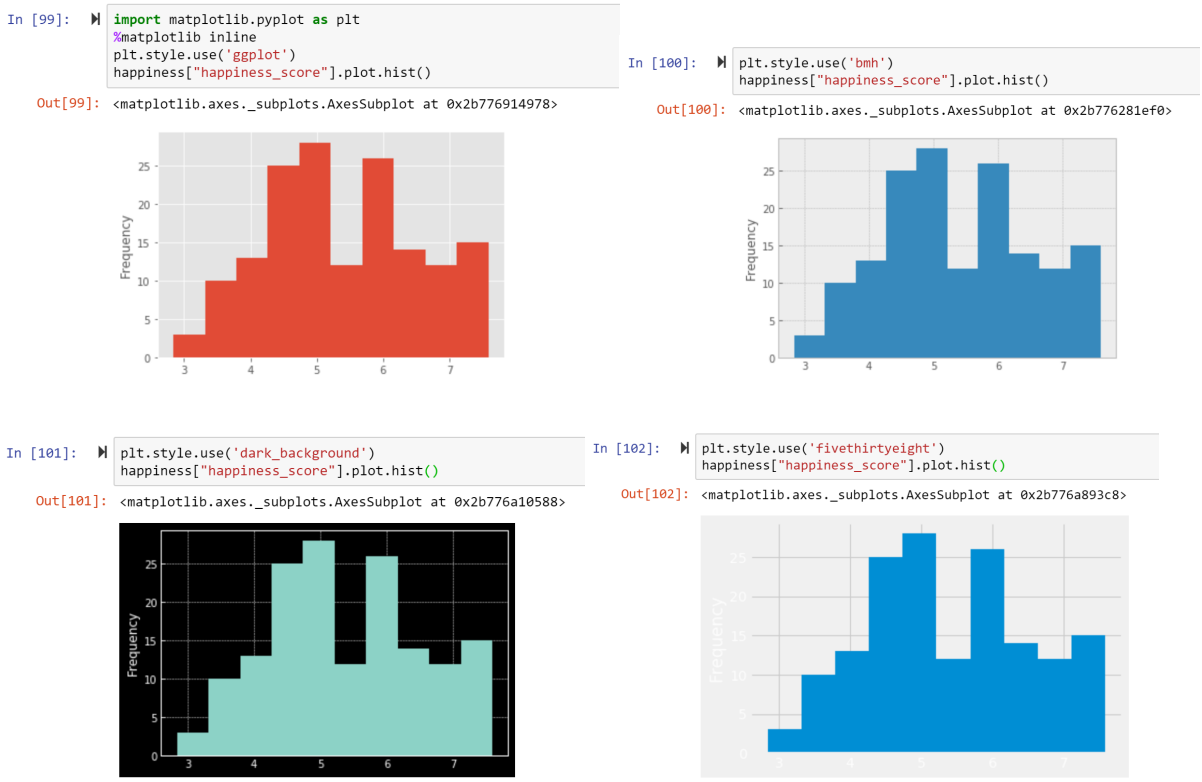
```
data.plot(kind='barh',legend=False, figsize=(5,5),color='red')
plt.title(Happiness Report',color = 'black')
plt.xticks(color = 'black')
plt.yticks(color = 'black')
plt.xlabel('Happiness_score',color = 'black')
plt.ylabel(Region',color = 'black')
plt.savefig('bar_happiness.png')
```

Furthermore, you can use style sheets from Matplotlib to make your plots look nicer. These style sheets include `plot_bmh`, `plot_fivethirtyeight`, `plot_ggplot` and more

(https://matplotlib.org/gallery.html#style_sheets). They basically create a set of style rules that your plots follow. By set up the style you can keep all your visualisation consistent. Here are some examples of the styles:

```
plt.style.use('ggplot')

plt.style.use('bmh')

plt.style.use('dark_background')

plt.style.use('fivethirtyeight')
```

Figure 27: Style examples for Pandas plots



1.6. Visualisation with *Seaborn*

Seaborn is a Python data visualization library also based on *Matplotlib*. It provides a high-level interface for drawing attractive and informative statistical graphics. It can be more effective than *Matplotlib* by using only one line to create one graph while using *Matplotlib* you need to write multiple lines. It builds on top of matplotlib and integrates closely with *Pandas* data structures.

Before you start using *Seaborn*, you need to import the library first (note: *Seaborn* is also included automatically in the Anaconda distribution but will need to be manually installed on other versions of Python).

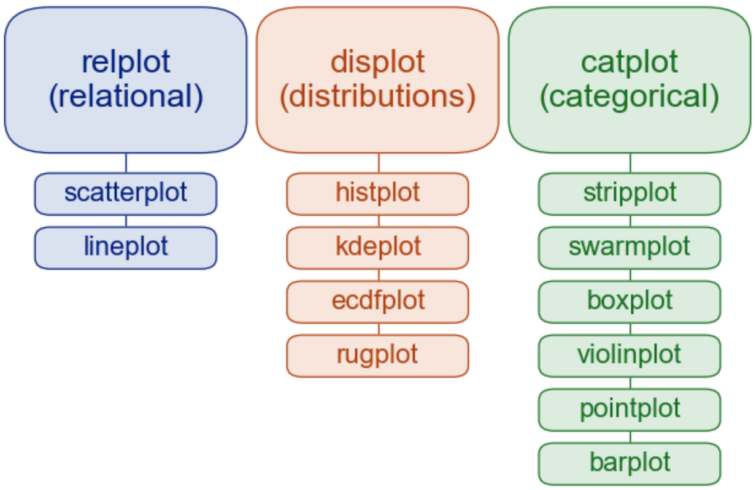
```
import seaborn as sns

%matplotlib inline
```

Seaborn provide some built-in dataset and we can use some sample dataset to practice using `sns.load_dataset()` method. Please check https://seaborn.pydata.org/generated/seaborn.load_dataset.html to see how you can load the dataset and check https://github.com/mwaskom/seaborn-data to see what datasets are available for you to use. In this tutorial, we will use some sample datasets from *Seaborn*. (If you don't want to work with sample data, you can use `pd.read_csv()` to import any datasets that you want to work with.)

Seaborn has a set of plotting functions. Before we go deeper to specific features offered by each function, we will have an overview of the seaborn plotting functions. Most of the documentations in seaborn are structured around three modules, namely distribution module, relational module, and categorical module.

In addition, it's noted that seaborn functions can be classified as "figure-level" or "axes-level". Each module has a single figure-level function, which offers a unitary interface to its various axes-level functions. The organisation of this is summarised in:

Figure 28: Classification of Seaborn functions



Here `sns.relplot()`, `sns.displot()` and `sns.catplot()` are the figure-level function for the relational, distribution and categorical module respectively. For example, `displot()` draws histogram in default mode using the following code:

```
penguins = sns.load_dataset("penguins")
```

```
sns.displot(data=penguins, x="flipper_length_mm", hue="species",
multiple="stack")
```

If you want to draw a kernel density plot, instead of using `sns.kdeplot()`, you use kind keyword argument within `displot()`, like this:

```
sns.displot(data=penguins, x="flipper_length_mm", hue="species",
multiple="stack", kind="kde")
```

The above examples show figure-level function in *Seaborn*. If instead you use something like `sns.kdeplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack")`, you are creating an axes-level function.

The most useful feature offered by the figure-level functions is that they can easily create figures with multiple subplots. For example, instead of stacking the three distributions for each species of penguins in the same axes, we can "facet" them by plotting each distribution across the columns of the figure:

```
sns.displot(data=penguins, x="flipper_length_mm", hue="species",
col="species")
```

Generally, the figure-level functions wrap their axes-level counterparts and pass the kind-specific keyword arguments (such as the bin size for a histogram) down to the underlying function, which means with figure-level functions there is less flexibility. However, axes-level functions can make self-contained plots. Here we will come back to the part on how we create figure and axes in *Matplotlib*. So, we need to import *Matplotlib* to use together with seaborn. Here is an example:

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

```
%matplotlib inline
```

```
figure, axs = plt.subplots(1, 2, figsize=(8, 4),
gridspec_kw=dict(width_ratios=[4, 3])) # create figure and axes
```

```
sns.scatterplot(data=penguins, x="flipper_length_mm", y="bill_length_mm",
hue="species", ax=axs[0]) #axe 1
```

```
sns.histplot(data=penguins, x="species", hue="species", shrink=.8,
alpha=.8, legend=False, ax=axs[1]) #axe 2
```

```
figure.tight_layout()
```

In this way, it's much easier for you to adjust what you want for each axe in your plot. To sum up, figure-level functions can create a cleaner plot, but if you want to make a complex figure with multiple different plot "kinds" it is recommended to create the figure and axes using *Matplotlib* first and the to fill in individual components using axes-level functions in *Seaborn*.

Next, we will introduce different kinds of visualisation that can be realised using seaborn. It will include distribution, relational and categorical plotting. In addition, we will have a look at how to create multi-plot grids and matrix plotting. Here we will use the built-in dataset from *Seaborn* called "tipis.csv" (the same dataset we used in chapter five).

```
tips = sns.load_dataset('tips')

tips.head()
```

Figure 29: Importing a built-in dataset in Seaborn

```
In [35]: # Load an example dataset
         tips = sns.load_dataset("tips")
         tips.head()
```

Out[35]:

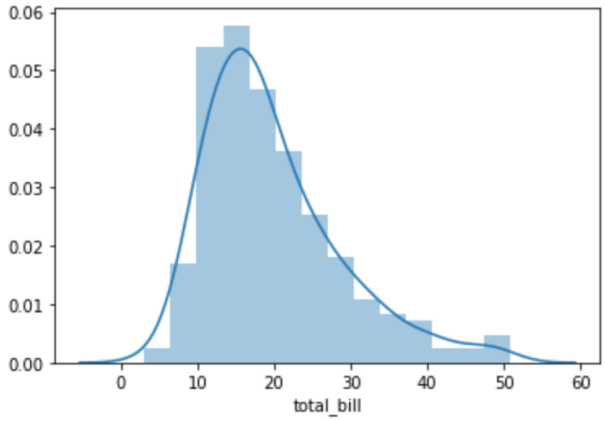
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

There are different kinds of plots that allow us to visualise the distribution of the dataset. Here we will try axes-level functions including `distplot()`, `histplot()`, `kdeplot()` and also figure-level function `displot()`. Furthermore, `jointplot()` will be shown on how to match up two histogram for bivariate data.

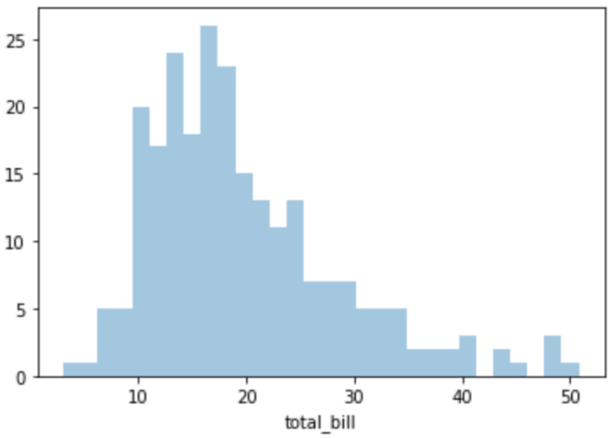
We can simply use `distplot()` to draw the histogram of the data with kernel density plot. To remove the kde layer, you can add the `kde=False` argument inside the `distplot()` to only have the histogram.

Figure 30: Creating histograms using the `distplot()` method

```
In [2]: sns.distplot(tips['total_bill'])
Out[2]: <matplotlib.axes._subplots.AxesSubplot at 0x17a465156a0>
```




```
In [6]: sns.distplot(tips['total_bill'],kde=False,bins=30)
C:\Users\Ssoph\Anaconda3\lib\site-packages\seaborn\distribut
and will be removed in a future version. Please adapt your c
r flexibility) or `histplot` (an axes-level function for his
  warnings.warn(msg, FutureWarning)
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5338a9550>
```

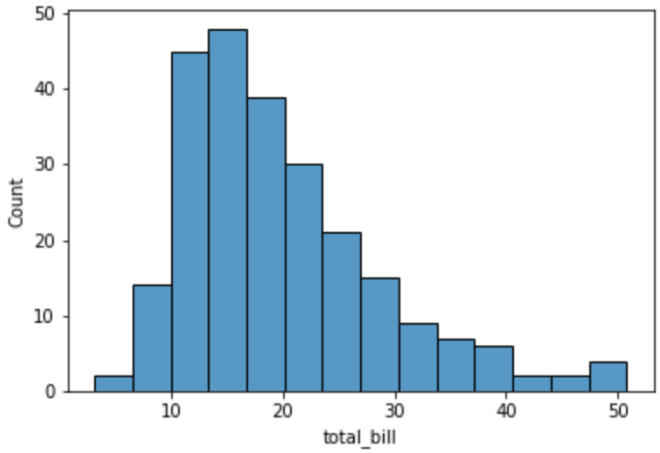


However, in the most recent version of seaborn, `distplot()` is deprecated, meaning it will be removed in a future version. `histplot()` is more recommended to use in this case. Next we will show the example with `histplot()`. Please note that you may get the error when you are running `histplot()` for the first time (as shown in figure). That is because your version of *Seaborn* is not uptodate. What you need to do is just to upgrade your *Seaborn* version to "0.11.1",

following the instructions in chapter four, and then you can restart the kennel and run the code again as shown in **Figure 31**.

Figure 31: Creating a histogram using the `histplot()` method

```
In [4]: sns.histplot(tips['total_bill'])
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x1c532446550>
```

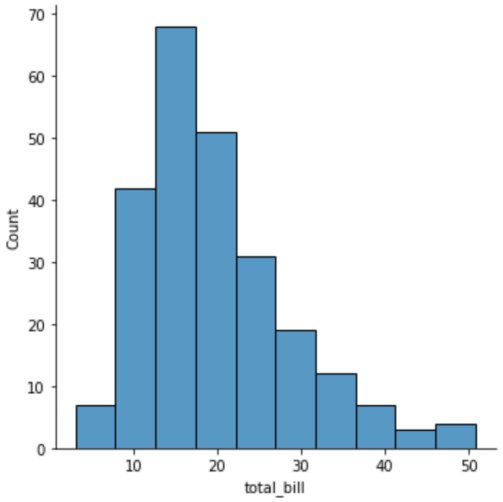


Of course, as we mentioned previously, you can also use the figure-level function `displot()` to draw the histogram which looks almost the same as `histplot()`.

By default, `displot()`/`hisplot()` chooses an automatic bin size based on the variance of the data and the number of observations. However, it is advisable to check your impressions of the distributions are consistent across different bin sizes. You can easily adjust the bin size in *Seaborn* by using then `bins` or `binwidth` keyword argument.

Figure 32: Adjusting bin sizes in Seaborn histograms

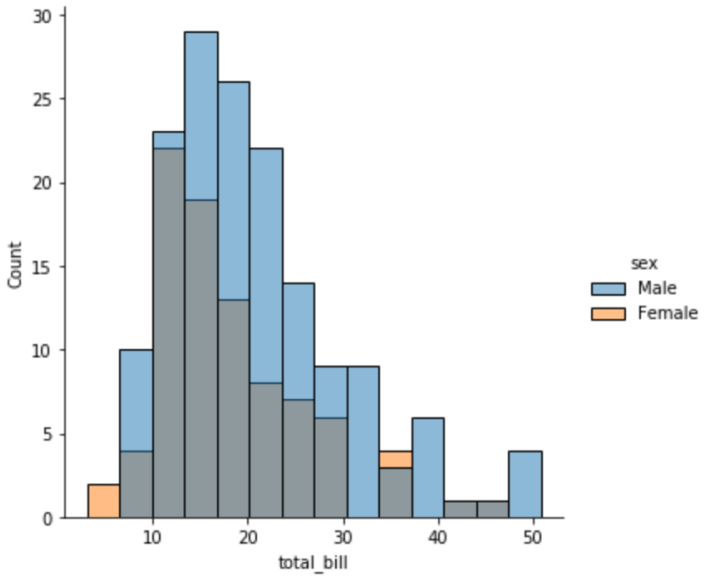
```
In [8]: sns.displot(tips['total_bill'],bins=10)
Out[8]: <seaborn.axisgrid.FacetGrid at 0x1c5339c9dd8>
```



`displot()` and `histplot()` provide support for conditional subsetting via the hue semantic. Assigning a variable to hue will draw a separate histogram for each of its unique values and distinguish them by colour.

Figure 33: Conditional subsetting of a histogram

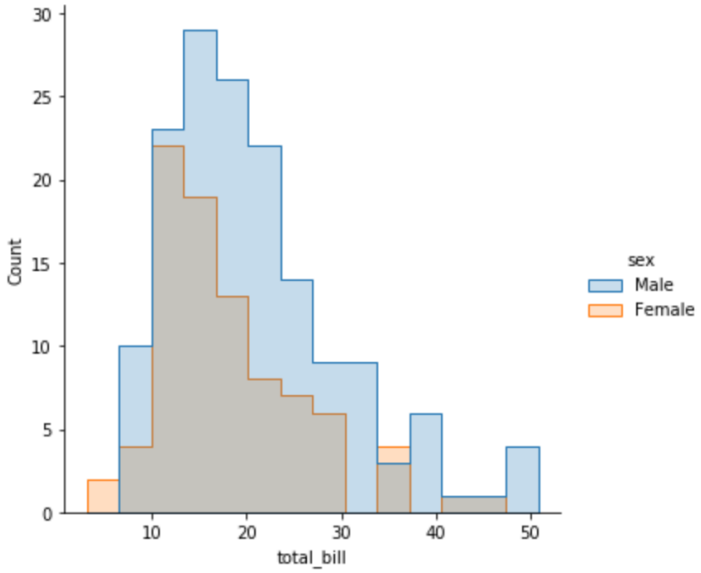
```
In [10]: sns.displot(tips,x='total_bill',hue='sex')
Out[10]: <seaborn.axisgrid.FacetGrid at 0x1c533c6eb38>
```



By default, the different histograms are "layered" on top of each other and, in some cases, they may be difficult to distinguish. One option is to change the visual representation of the histogram from a bar plot to a "step" plot:

Figure 34: Changing the visual representation of a histogram (step plot)

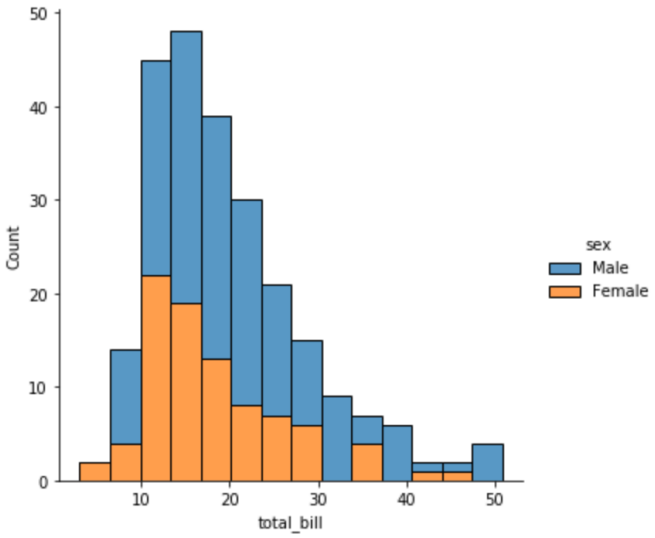
```
In [11]: sns.displot(tips,x='total_bill',hue='sex',element='step')
Out[11]: <seaborn.axisgrid.FacetGrid at 0x1c533d72dd8>
```



Alternatively, instead of layering each bar, they can be "stacked", or moved vertically. In this plot, the outline of the full histogram will match the plot with only a single variable:

Figure 35: Changing the visual representation of a histogram (stacked plot)

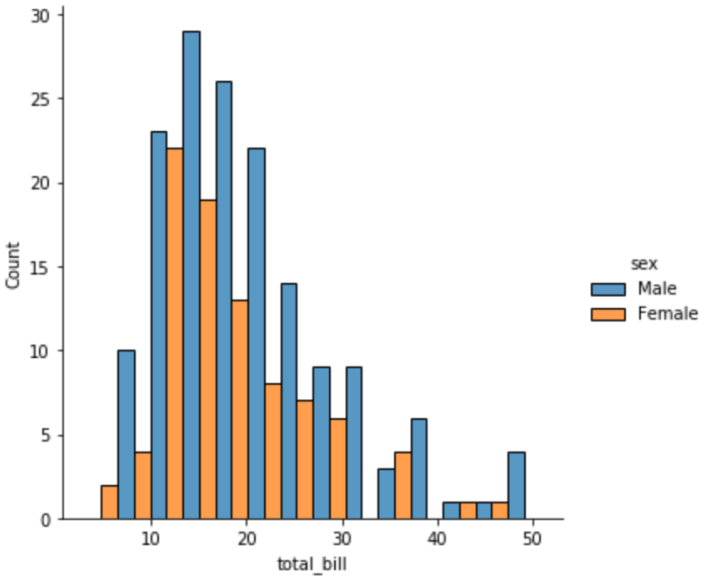
```
In [12]: sns.displot(tips,x='total_bill',hue='sex',multiple='stack')
Out[12]: <seaborn.axisgrid.FacetGrid at 0x1c533da4eb8>
```



Another option is to "dodge" the bars, which moves them horizontally and reduces their width. This ensures that there are no overlaps and that the bars remain comparable in terms of height. However, this only works well when the categorical variable has a small number of levels:

Figure 36: Changing the visual representation of a histogram (clustered plot)

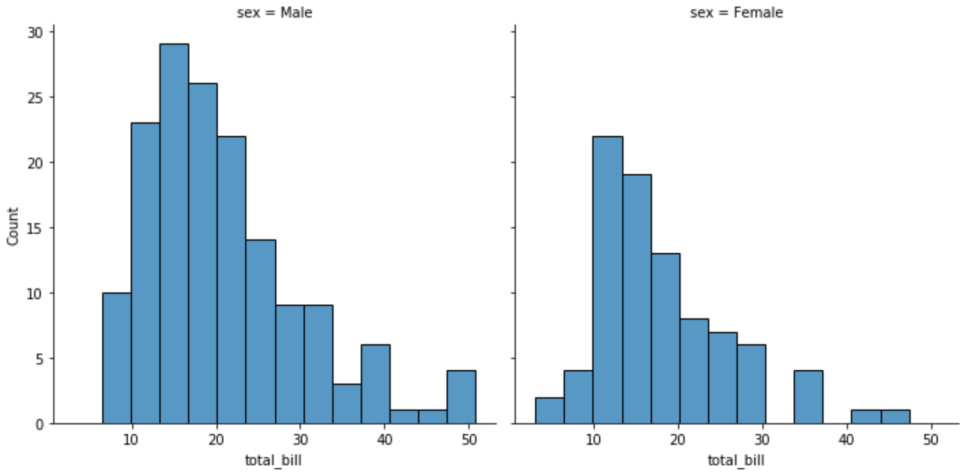
```
In [14]: sns.displot(tips,x='total_bill',hue='sex',multiple='dodge')
Out[14]: <seaborn.axisgrid.FacetGrid at 0x1c533f70b70>
```



Since `displot()` is a figure-level plot, it's possible to draw individual distribution in separate subplot based on the second variable (in this case it's the "sex") instead of using hue.

Figure 37: Creating a multi-histogram in Seaborn

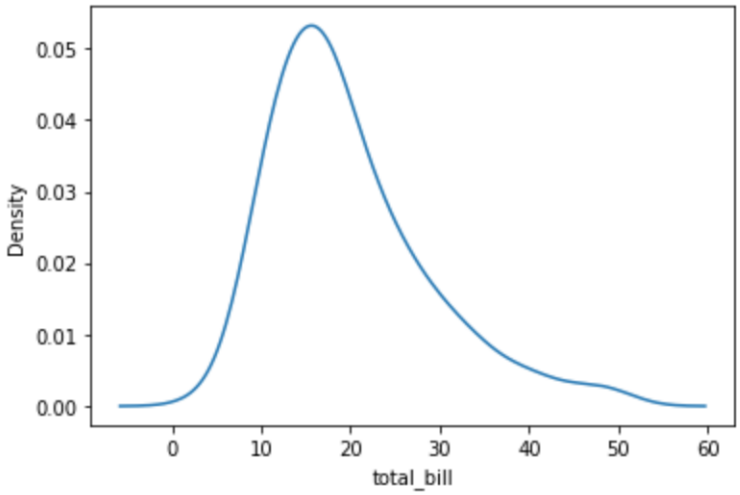
```
In [13]: sns.displot(tips,x='total_bill',col='sex',multiple='dodge')
Out[13]: <seaborn.axisgrid.FacetGrid at 0x1c533d69e80>
```



A Kernel Density Plot (KDE) replaces every single observation with a Gaussian (Normal) distribution centred around that value. For example:

Figure 38: Creating a KDE plot in Seaborn (method #1)

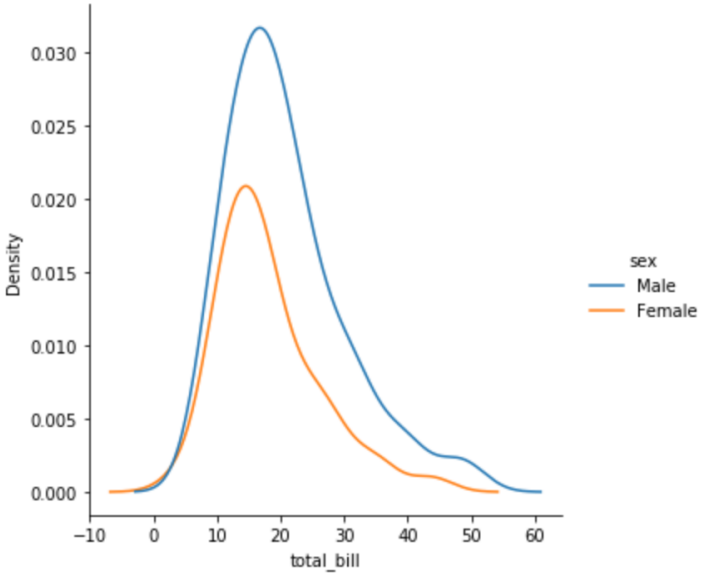
```
In [15]: sns.kdeplot(tips['total_bill'])
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1c534030c18>
```



Or you can use `displot()` adding the `kind` parameter to draw the KDE plot:

Figure 39: Creating a KDE plot in Seaborn (method #2)

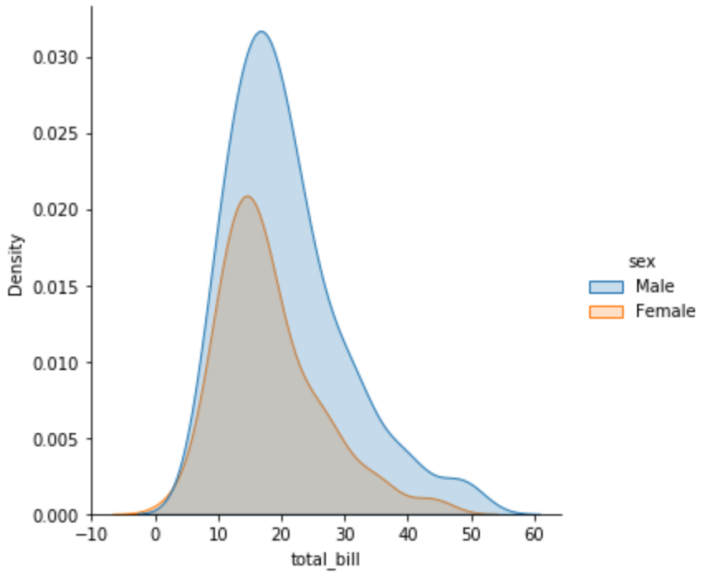
```
In [17]: sns.displot(tips,x='total_bill',hue='sex',kind='kde')
Out[17]: <seaborn.axisgrid.FacetGrid at 0x1c5340da710>
```



As with histograms, if you assign a hue variable, a separate density estimate will be computed for each level of that variable. You can also fill the area for each KDE:

Figure 40: Assigning hue to a KDE plot

```
In [19]: sns.displot(tips,x='total_bill',hue='sex',kind='kde',fill=True)
Out[19]: <seaborn.axisgrid.FacetGrid at 0x1c53514cba8>
```

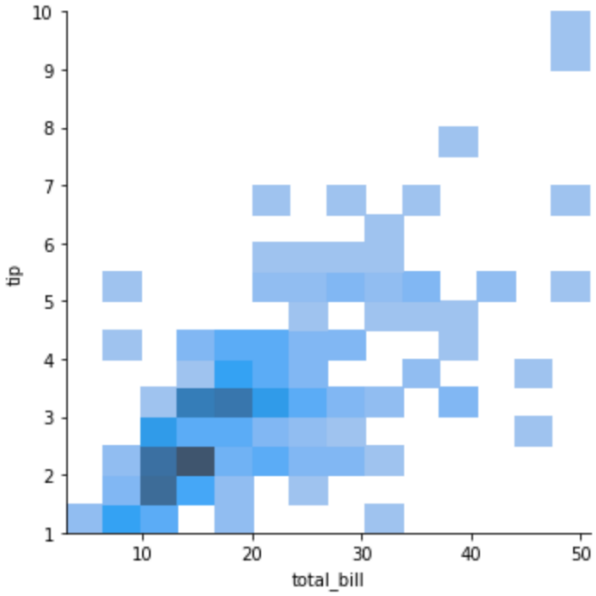


In addition, as with the histogram, you can stack the kernel (using `multiple = 'stack'`) so that you can see the full picture of the distribution as a whole.

All above are the examples of univariate distributions (single variable). Next, we will see some examples on how to visualise bivariate distributions. It's pretty straight-forward that we can simply assign y in `displot()` to draw the bivariate distribution. For example:

Figure 41: Visualising bivariate distributions in Seaborn

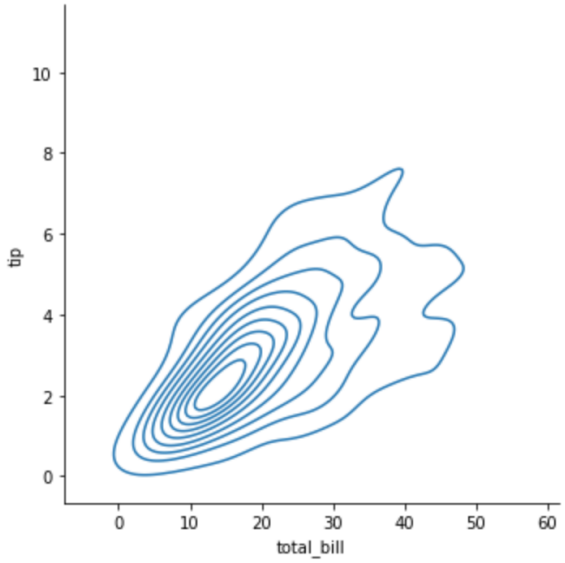
```
In [20]: sns.displot(tips,x='total_bill',y='tip')
Out[20]: <seaborn.axisgrid.FacetGrid at 0x1c5351ef978>
```



A bivariate histogram bins the data within rectangles that tile the plot and then shows the count of observations within each rectangle with the fill colour (analogous to a heatmap). Similarly, a bivariate KDE plot smoothes the (x, y) observations with a 2D Gaussian:

Figure 42: Visualising a bivariate distribution with KDE in Seaborn

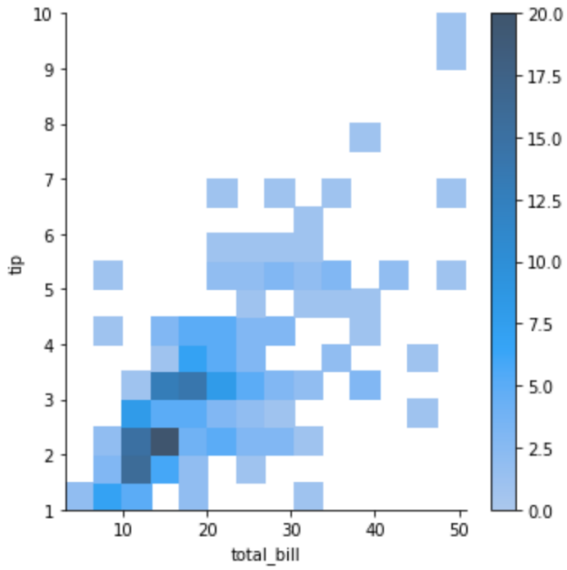
```
In [21]: sns.displot(tips,x='total_bill',y='tip',kind='kde')
Out[21]: <seaborn.axisgrid.FacetGrid at 0x1c5351e3710>
```



To better interpret the graph, you can add a colour bar to show the mapping between counts and a colour intensity:

Figure 43: Adding colour representation to a bivariate distribution visualisation

```
In [22]: sns.displot(tips,x='total_bill',y='tip',cbar=True)
Out[22]: <seaborn.axisgrid.FacetGrid at 0x1c5352ff208>
```

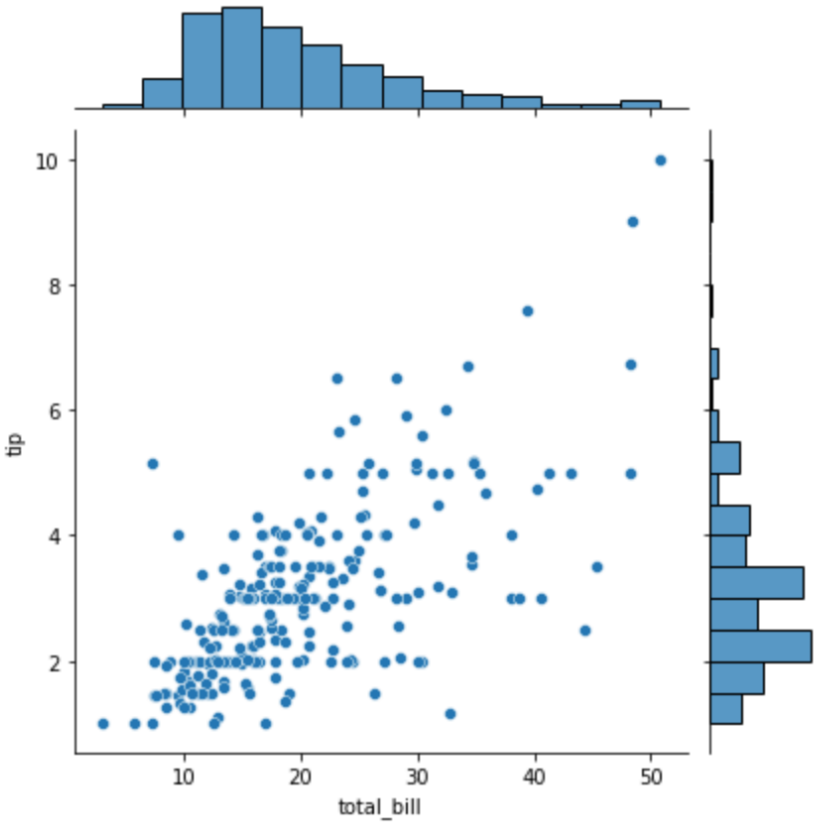


As with a univariate distribution plotting, you can assign a hue variable to plot multiple heatmap using different colours.

In the bivariate distribution, we will visualise the distribution combining two variables. But if we still want to see the single distribution of each variable, we can use `jointplot()` to get both the bivariate distribution and univariate distribution.

Figure 44: Creating joint distribution using the `jointplot()` method

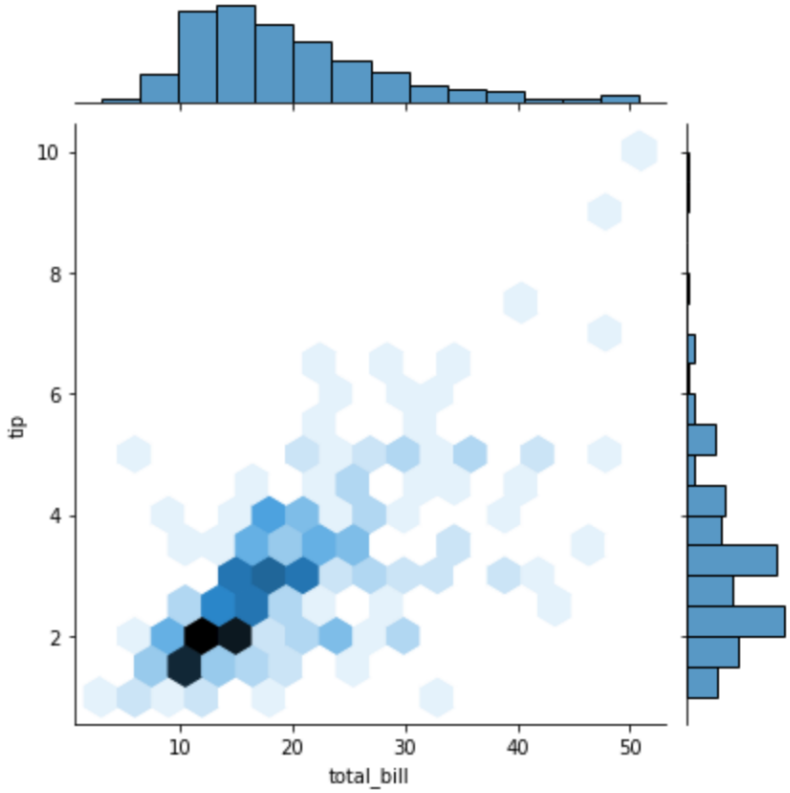
```
In [25]: sns.jointplot(data=tips,x='total_bill',y='tip')
Out[25]: <seaborn.axisgrid.JointGrid at 0x1c5355c20b8>
```



By default, `jointplot()` will display the bivariate distribution, using `scatterplot()`, and the marginal distribution, using `histplot()`. You can change the scatter chart to another type:

Figure 45: Changing plot type in the joint distribution plot

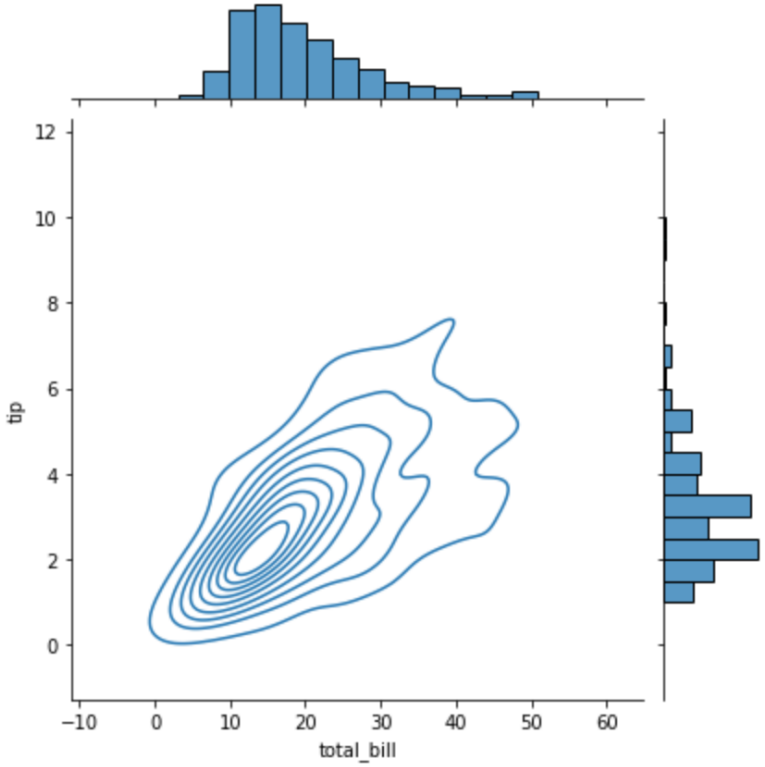
```
In [26]: sns.jointplot(data=tips,x='total_bill',y='tip',kind='hex')
Out[26]: <seaborn.axisgrid.JointGrid at 0x1c535717be0>
```



Instead of using `jointplot()`, you can use `JointGrid()` to have more flexibility to add any plots you want to the joint distribution plot. For example:

Figure 46: Creating a joint distribution using the `JointGrid()` method

```
In [29]: g = sns.JointGrid(data=tips,x='total_bill',y='tip')
         g.plot_joint(sns.kdeplot)
         g.plot_marginals(sns.histplot)

Out[29]: <seaborn.axisgrid.JointGrid at 0x1c5368b2358>
```



In a lot of cases, it's useful to visualise the data to understand the relationship between variables in the dataset. In *Seaborn*, `relplot()` is commonly used and it's figure-level functions as we mentioned before. Alternatively you can use `scatterplot()` and `lineplot()` which are axes-level functions. **Figure 47***: Creating a scatter plot using the `relplot()` method* demonstrates the `relplot()` method, while **Figure 48** demonstrates the `scatterplot()` method.

Figure 47: Creating a scatter plot using the `relplot()` method

```
In [30]: sns.relplot(data=tips, x="total_bill", y="tip")
Out[30]: <seaborn.axisgrid.FacetGrid at 0x1c536a38908>
```

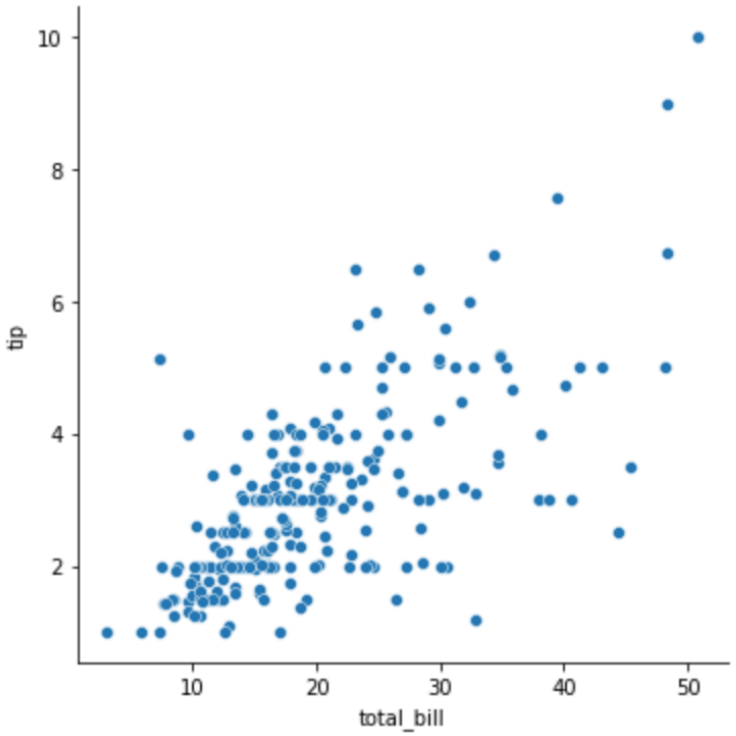
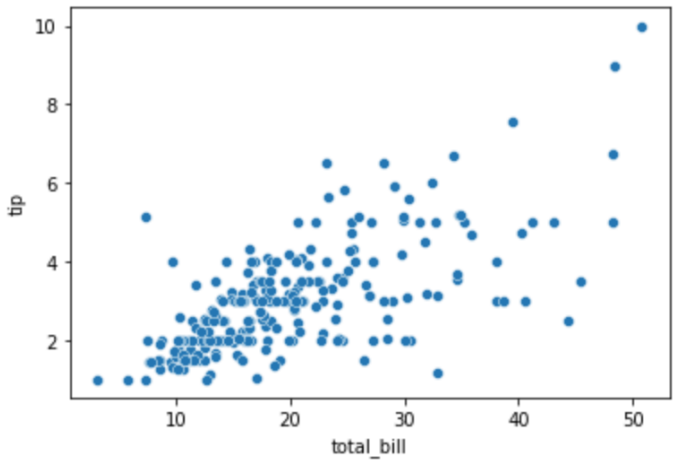


Figure 48: Creating a scatter plot using the `scatterplot()` method

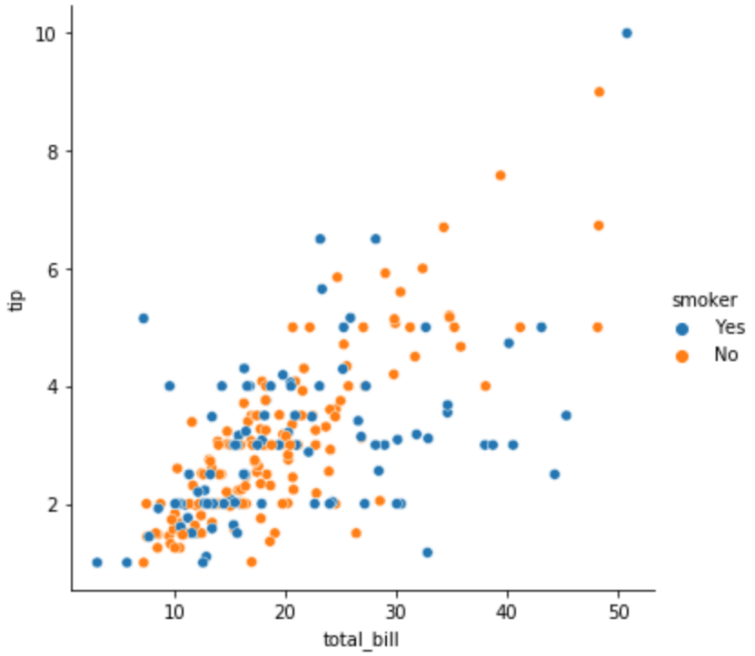
```
In [31]: sns.scatterplot(data=tips, x="total_bill", y="tip")
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x1c536aa5ac8>
```



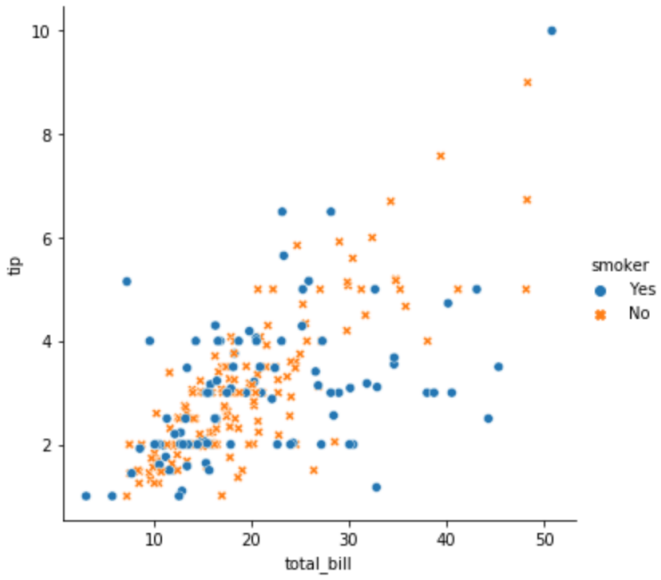
As with distribution plotting, you can assign `hue` and/or `style` to add more dimension to the plot. For example:

Figure 49: Assigning hue and style to a scatter plot

```
In [32]: sns.relplot(data=tips, x="total_bill", y="tip",hue='smoker')
Out[32]: <seaborn.axisgrid.FacetGrid at 0x1c536afb828>
```



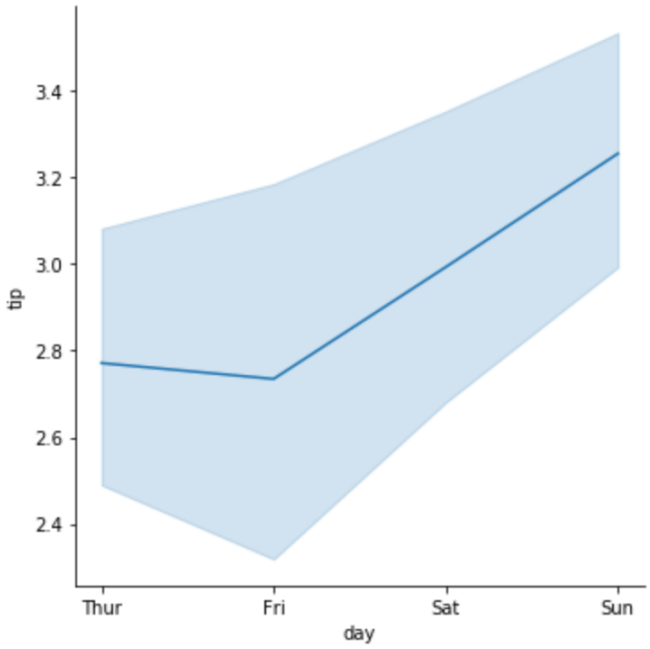

```
In [33]: sns.relplot(data=tips, x="total_bill", y="tip",hue='smoker',style='smoker')
Out[33]: <seaborn.axisgrid.FacetGrid at 0x1c536b6aa20>
```



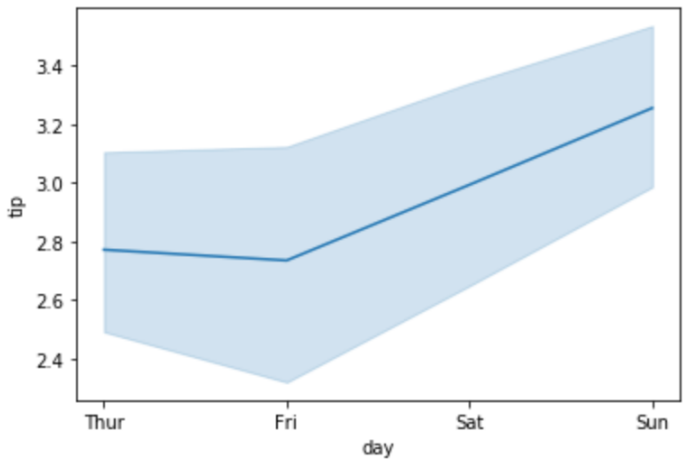
A scatter plot can be very effective to check the relationship between two variables. However, if you want to understand the changes in a variable caused by time or another similarly continuous variable, a line plot would be more suited. Here we can see the tips dataset by different days of the week by via a line chart. You can use either `relplot()` with `kind='line'` or `lineplot()`:

Figure 50: Creating a line chart using `relplot()` anf `lineplot()` methods

```
In [34]: sns.relplot(data=tips, x="day", y="tip",kind='line')
Out[34]: <seaborn.axisgrid.FacetGrid at 0x1c536bdf550>
```



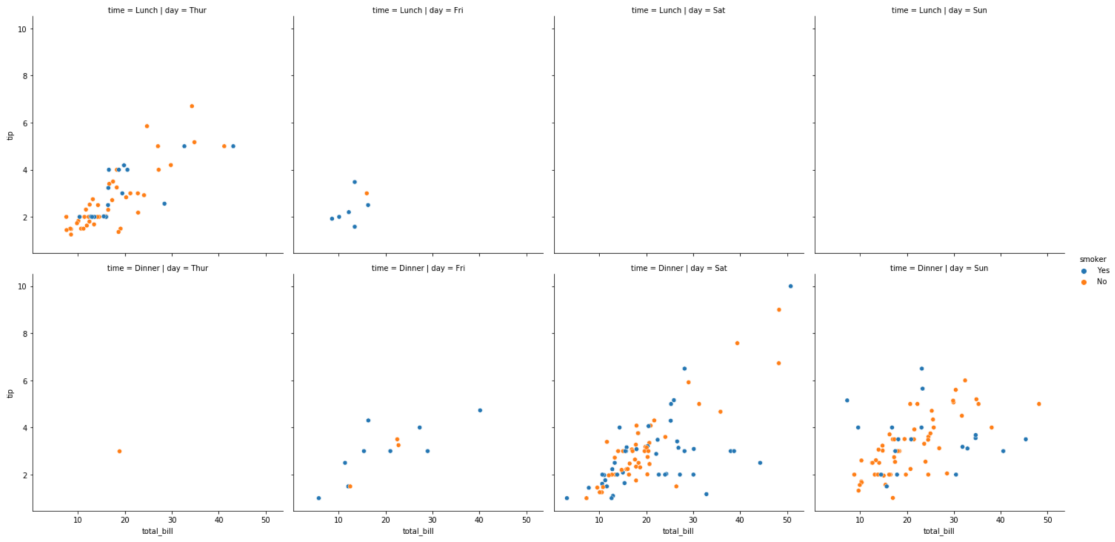

```
In [36]: sns.lineplot(data=tips, x="day", y="tip")
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x1c536c56f60>
```



Similarly, you can set up `hue` or `style` for your line chart as we do for the scatter plot and distribution plot. You can also create multiple plots to show the multiple relationships by using `relplot()`. In this way, you can incorporate more information into your visualisation:

Figure 51: Multiple relational plots

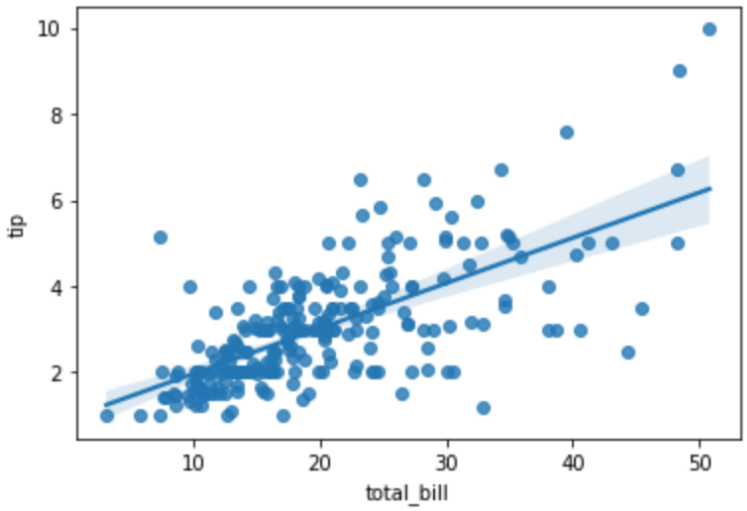
```
In [39]: sns.relplot(x="total_bill", y="tip", hue="smoker",col="day", row='time',data=tips)
Out[39]: <seaborn.axisgrid.FacetGrid at 0x1c536d02908>
```



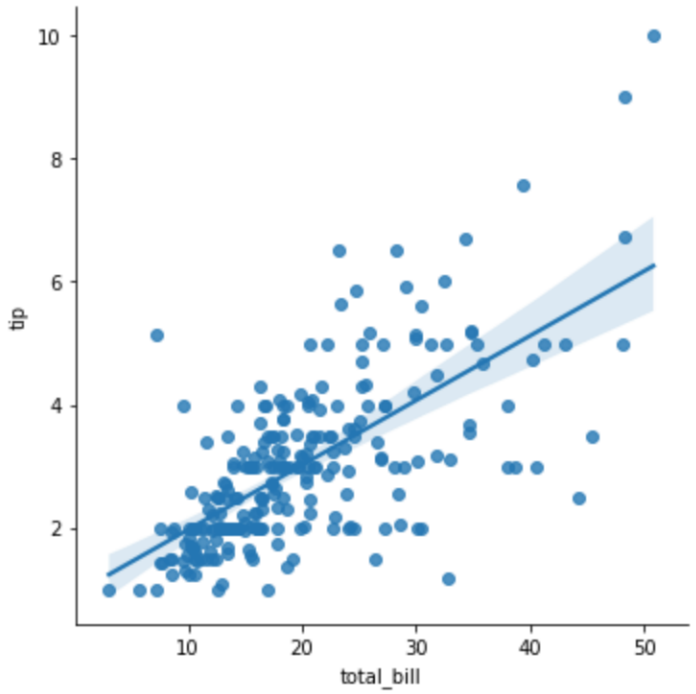
In the scatter plot, we can see the 'total bill' and 'tips' have potential relationship. To show this in a more direct way, we can build a regression model (as we did in chapter five) and visualise this using `regplot()` or `lmplot()`, both of which are functions used to draw linear regression plots. Both draw a scatterplot of two variables x and y, and then fit the regression model y ~ x, plotting the regression line and a 95% confidence interval. Here are the examples:

Figure 52: Creating regression plots using the `regplot()` and `lmplot()` methods

```
In [40]: sns.regplot(x="total_bill", y="tip", data=tips)
Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x1c537604518>
```



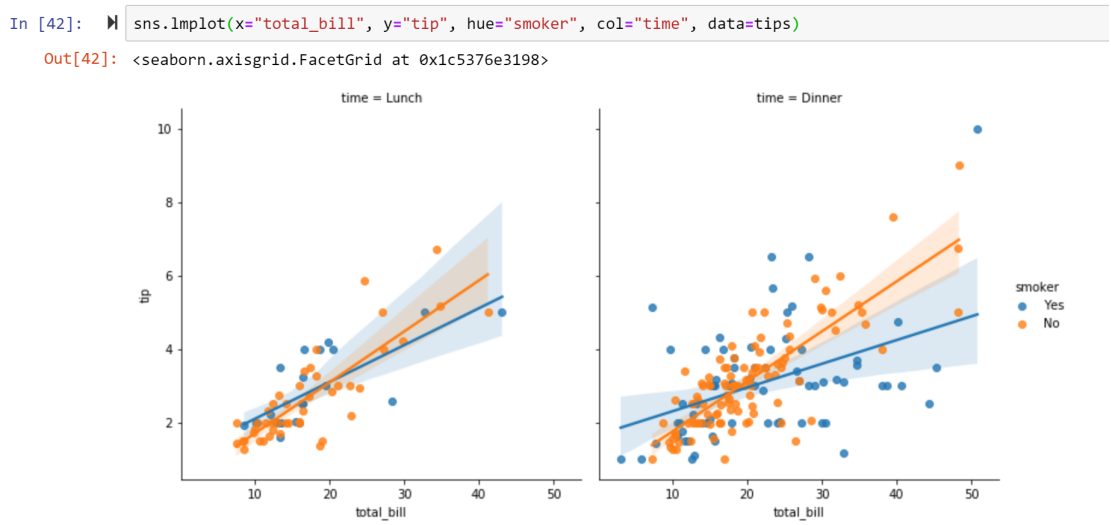

```
In [41]: sns.lmplot(x="total_bill", y="tip", data=tips)
Out[41]: <seaborn.axisgrid.FacetGrid at 0x1c537674fd0>
```



You can see here the results from both functions are almost the same. But they have slight difference. Basically, `regplot()` performs a simple linear regression model fit and plot.

`lmplot()` combines `regplot()` and `FacetGrid`. The `FacetGrid` class helps visualise the distribution of one variable as well as the relationship between multiple variables separately within subsets of your dataset using multiple panels. Thus, `lmplot()` is more computationally intensive and is intended as a convenient interface to show a linear regression on "faceted" plots that allow you to explore interactions with up to three categorical variables. For example:

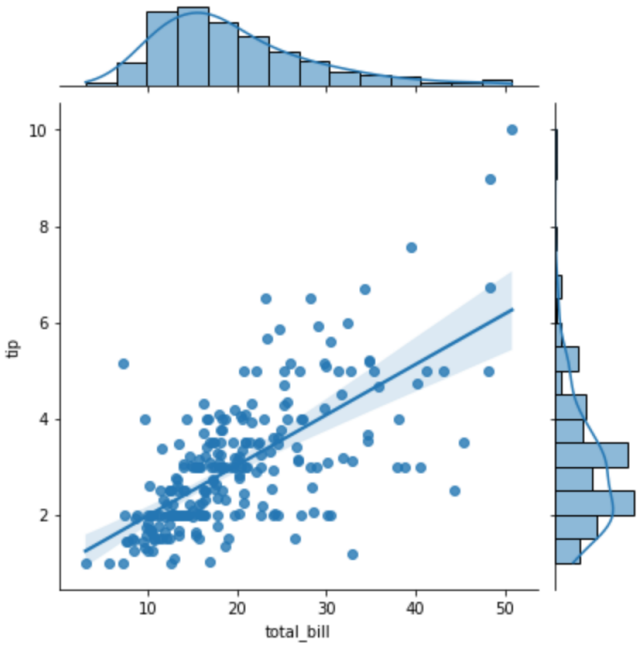
Figure 53: Showing regression on facet plots using `lmplot()`



You can also use `jointplot()` in this case to join the regression plot and the distribution plot as what we mentioned in the distribution module:

Figure 54: Joining a regression plot with a distribution plot

```
In [43]: sns.jointplot(x="total_bill", y="tip", data=tips, kind="reg")
Out[43]: <seaborn.axisgrid.JointGrid at 0x1c5377a7b38>
```



If the main variable is categorical, then categorical functions would be useful to visualise categorical data. Generally, there are three kinds of categorical plots that are available in seaborn, which includes categorical scatter plots (`stripplot()` and `swamplot()`), categorical distribution plots (`boxplot()`, `violinplot()`, `boxenplot()`) and categorical estimate plots (`pointplot()`, `barplot()`, `countplot()`). All this plots can be replaced as `catplot()` which is figure-level functions similar to `relplot()` and `displot()`. Next, we will give some examples in each type of categorical plots.

Categorical scatterplots include strip plot and swarm plot. The default representation of the data in `catplot()` uses a `stripplot`. Therefore, we can use `catplot()` to draw strip plot directly or use `catplot()` by assigning `kind='swarm'` to draw swarm scatter plot. Optionally, we can use `stripplot()` or `swarmplot()` instead.

The `stripplot()` will draw a scatterplot where one variable is categorical. A strip plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.

The `swarmplot()` is similar to `stripplot()`, but the points are adjusted (only along the categorical axis) so that they don't overlap. This gives a better representation of the distribution

of values, although it does not scale as well to large numbers of observations (both in terms of the ability to show all the points and in terms of the computation needed to arrange them).

Figure 55: Creating strip plot using `catplot()` method

```
In [47]: sns.catplot(x="day", y="total_bill", data=tips,jitter=False)
Out[47]: <seaborn.axisgrid.FacetGrid at 0x1c537758f60>
```

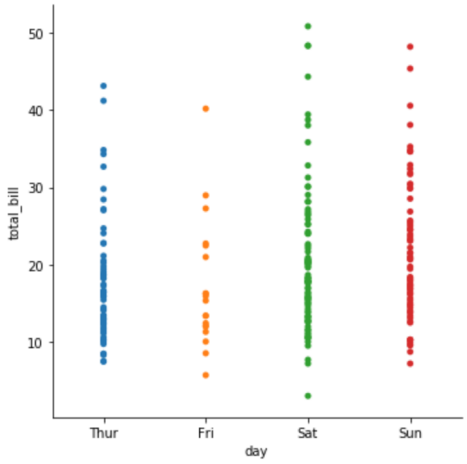
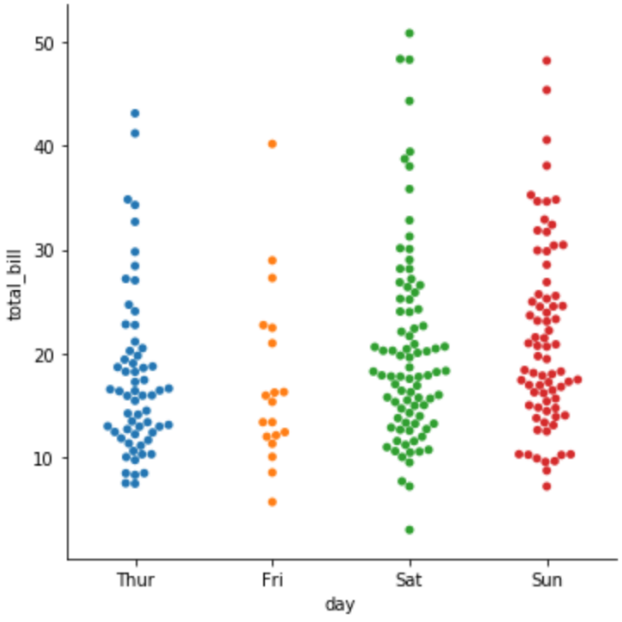


Figure 56: Creating swarm plot using `catplot()` method

```
In [48]: sns.catplot(x="day", y="total_bill", data=tips,kind='swarm')
Out[48]: <seaborn.axisgrid.FacetGrid at 0x1c5357e20b8>
```



Like a relational plot and distribution plot, you can assign hue to the categorical scatter plot to have more dimensions in the chart. When using `displot()`, `histplot()` or `kdeplot()`, we

usually draw the univariate or bivariate distributions. But if we want to compare the distributions among different "categories", then categorical distribution plots will be more informative. The commonly used categorical distribution plots are boxplot and violin plot. To draw boxplot and violin plot, you can either using `catplot()` by setting the kind or using `boxplot()` and `violinplot()` respectively. Here are the examples using the same dataset:

Figure 57: Creating boxplot using `catplot()` method

```
In [53]: sns.catplot(x="day", y="total_bill", data=tips,kind='box')
Out[53]: <seaborn.axisgrid.FacetGrid at 0x1c536da15f8>
```

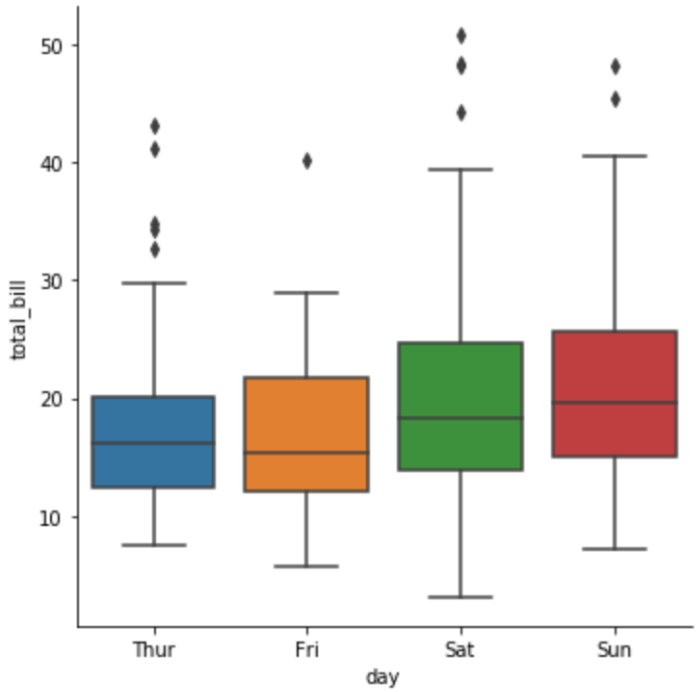
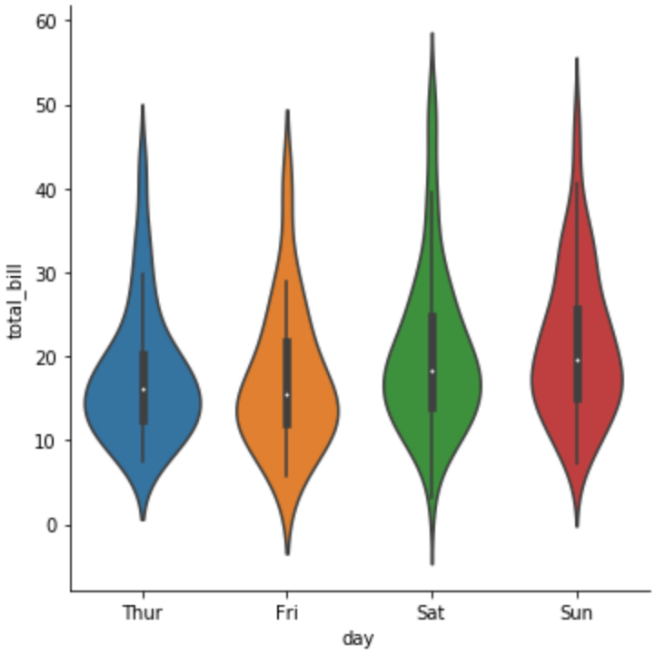


Figure 58: Creating violin plot using `catplot()` method

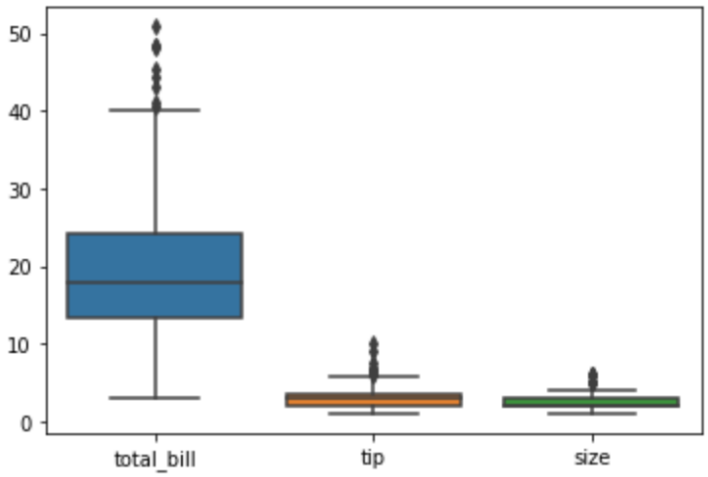
```
In [55]: sns.catplot(x="day", y="total_bill", data=tips, kind='violin')
Out[55]: <seaborn.axisgrid.FacetGrid at 0x1c536f8ca20>
```



You can also visualise the distribution of all the columns (with numerical data) in the dataset (if you want a vertical view of the data) by using `orient='h'` assignment:

Figure 59: Drawing a boxplot for the whole dataset

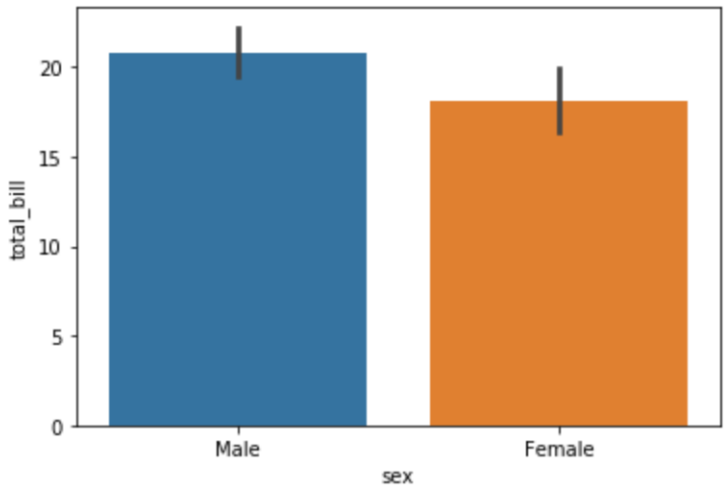
```
In [56]: sns.boxplot(data=tips)
Out[56]: <matplotlib.axes._subplots.AxesSubplot at 0x1c536f82470>
```



Rather than showing distribution of the data, you might want to show an estimate of the central tendency of the value (e.g. sum, mean etc.) The most common plot type for this purpose is bar plot or count plot (bar plot showing counts of the observation) in *Seaborn*. Similar to the other categorical plots, we can use either figure-level function (`sns.catplot()` with `kind='bar'` or `sns.barplot()`/`sns.countplot()`). Below are some examples using the same dataset, but now focusing on the amount of bill from customers of different gender.

Figure 60: Creating bar chart using `barplot()` method

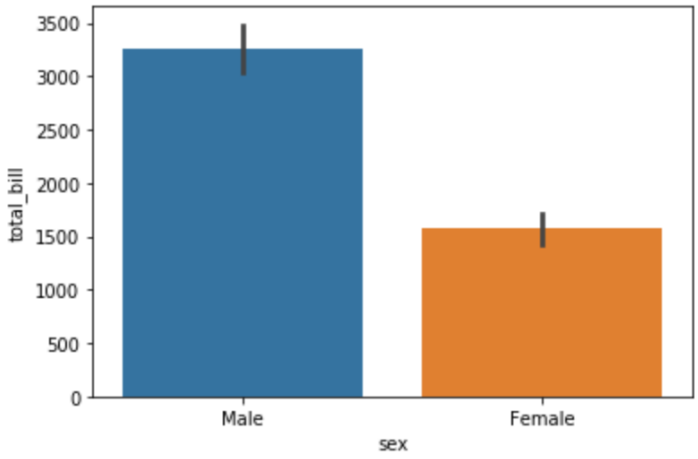
```
In [57]: sns.barplot(x='sex',y='total_bill',data=tips)
Out[57]: <matplotlib.axes._subplots.AxesSubplot at 0x1c53708eb00>
```



When there are multiple observations in each category, it also uses bootstrapping to compute a confidence interval around the estimate, which is plotted using error bars. And by default, the estimate takes the mean of each category. So here the chart shows the average amount of bill for each gender. If you want to change the mean to the sum or any other estimator, you can assign estimator to change (the estimator can be assigned with any statistical function to estimate within each categorical bin and you can import *NumPy* to do that):

Figure 61: Changing the estimator for the bar chart

```
In [58]: sns.barplot(x='sex',y='total_bill',data=tips,estimator=sum)
Out[58]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5371074a8>
```



And as we mentioned, the error bar shows the confidence interval (by default `ci=95`) and you can remove it, adjust it, or change it to standard deviation (`ci='sd'`) as you want:

Figure 62: Adjust the confidence interval to 70% on a bar chart

```
In [60]: sns.barplot(x='sex',y='total_bill',data=tips,ci=70)
Out[60]: <matplotlib.axes._subplots.AxesSubplot at 0x1c5371ad080>
```

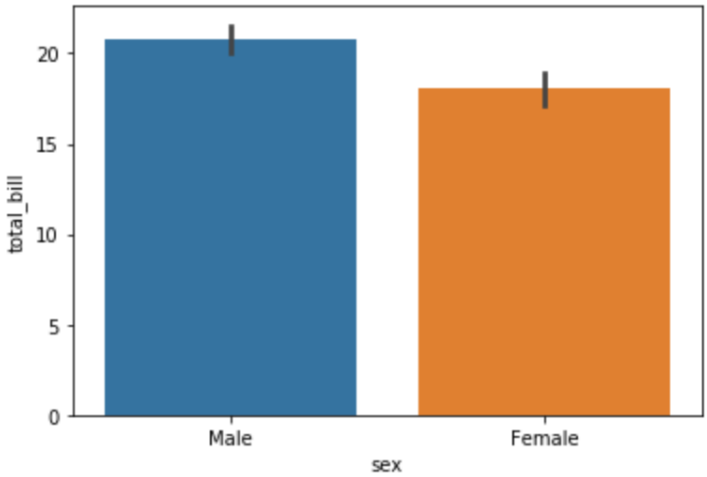
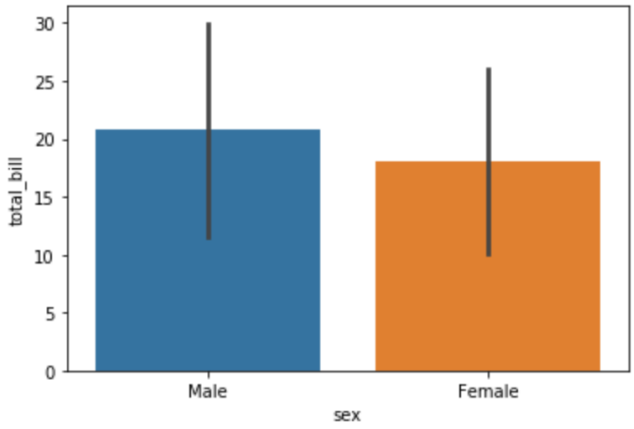


Figure 63: Adjust the confidence interval to standard deviation on a bar chart

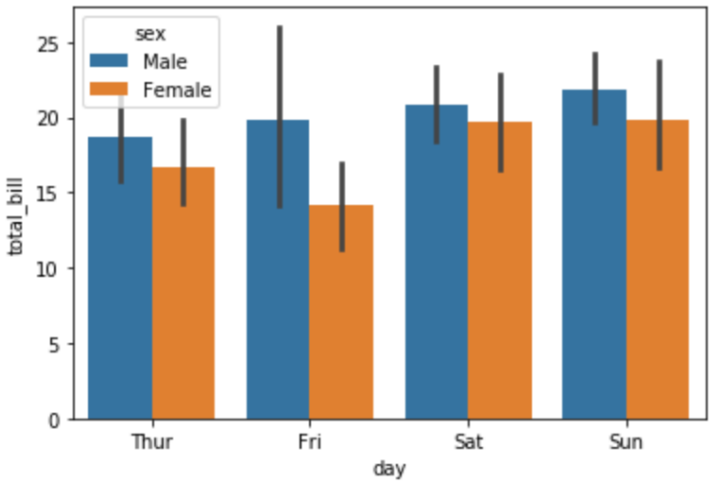
```
In [61]: sns.barplot(x='sex',y='total_bill',data=tips,ci='sd')
Out[61]: <matplotlib.axes._subplots.AxesSubplot at 0x1c537201588>
```



Like other plots, you can set up `hue` for your bar chart to create a clustered column bar chart (by default) or stacked bar chart (`dodge=False`) when assigning hue:

Figure 64: Assigning hue to the bar chart

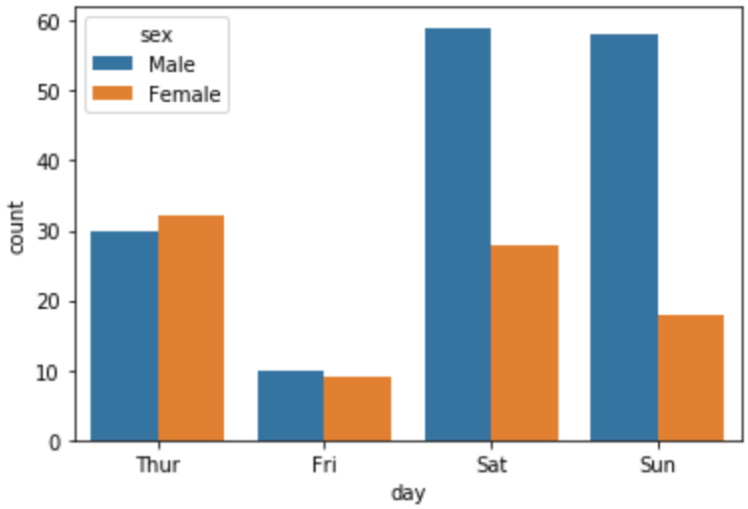
```
In [62]: sns.barplot(x='day',y='total_bill',data=tips,hue='sex')
Out[62]: <matplotlib.axes._subplots.AxesSubplot at 0x1c53726b0f0>
```



If you just want to show the number of the observations instead you can use `countplot()` as a special case for bar chart.

Figure 65: Creating a count plot using `countplot()` method

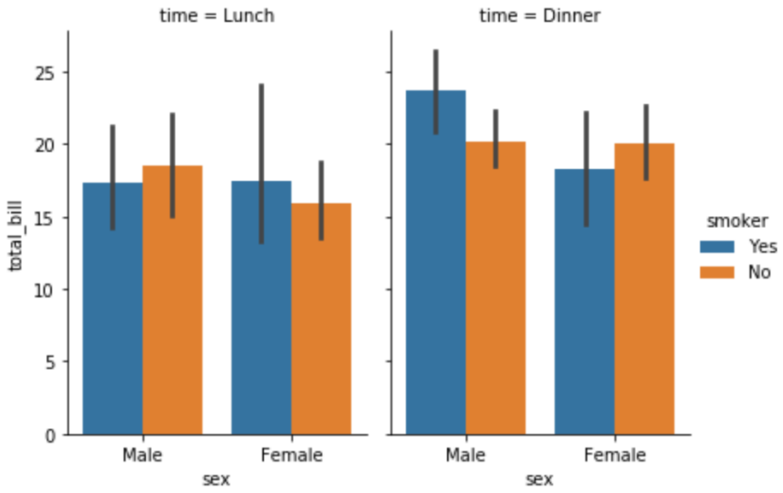
```
In [68]: sns.countplot(x='day',data=tips,hue='sex')
Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0x1c537a676d8>
```



As with other categorial plots, you can also use `catplot()` as a good way for you to create facet grid that allowing you to create multiple plots:

Figure 66: Creating bar plot using `catplot()` method

```
In [69]: sns.catplot(x="sex", y="total_bill",
                     hue="smoker", col="time",
                     data=tips, kind="bar",
                     height=4, aspect=.7);
```



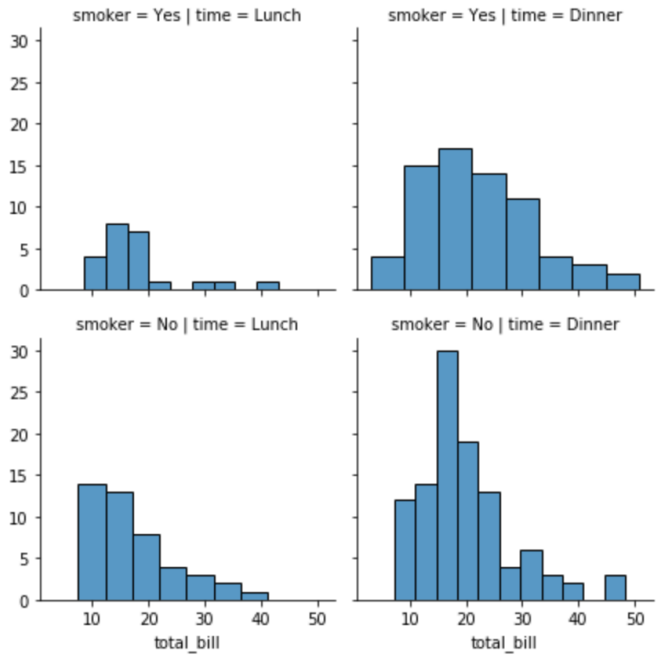
The above are the most commonly used charts/graphs that you can employ using *Seaborn*. Many are ultimately similar in terms of their coding. In the last part of this section, we will cover some examples of the visualisation of both grids and matrices.

1.7. Facet Grids and Matrix Plots in *Seaborn*

We have covered in the previous section primarily figure-level functions and `jointplot`. Here we will introduce another function called `FacetGrid`, which will also create multiple plots.

Figure 67: Creating multiple plots with `FacetGrid()`

```
In [76]: g = sns.FacetGrid(tips, col="time", row="smoker") # create the gird first
         g.map(sns.histplot, "total_bill") # map the plots
Out[76]: <seaborn.axisgrid.FacetGrid at 0x1c53a9b8390>
```

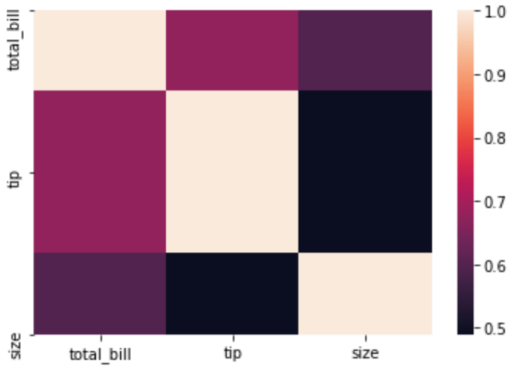


There is also some other gird visualisation like pair grid plots in *Seaborn*, but we are not going to cover here. You can explore the web page if you are interested in this: https://seaborn.pydata.org/tutorial/axis_grids.html.

Next, we will see how we can visualise matrix plots such as heatmaps in *Seaborn*. We didn't cover this kind of plots in *Matplotlib* and *Pandas* because using seaborn would be much easier. Here is the example to create a heatmap in *Seaborn*:

Figure 68: Creating a heatmap in Seaborn

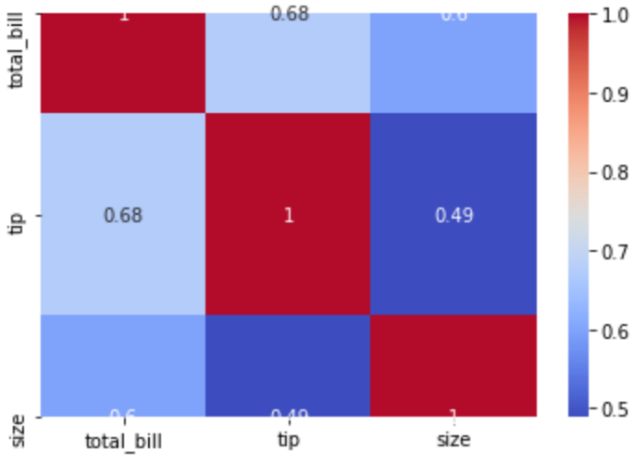
```
In [77]: tips.corr() # Matrix form for correlation data
         sns.heatmap(tips.corr()) # visualise the heatmap based on the correlation matrix
Out[77]: <matplotlib.axes._subplots.AxesSubplot at 0x1c53bb0a2b0>
```



You can set up colour and annotation for your heatmap to make it more informative by adding argument:

Figure 69: Setting colours and annotations for our heatmap

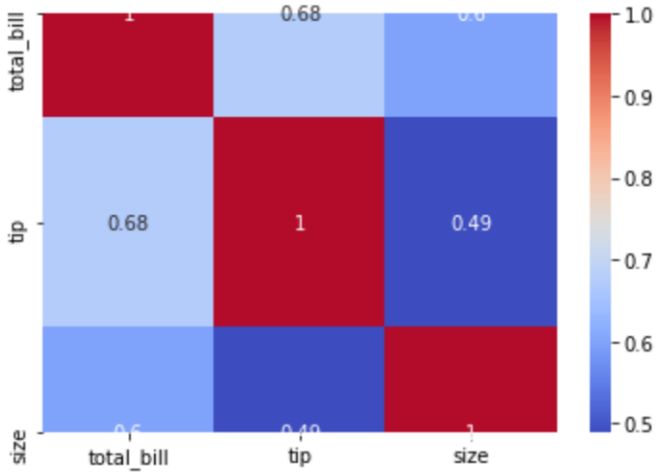
```
In [78]: sns.heatmap(tips.corr(),cmap='coolwarm',annot=True)
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x1c53bbbd1d0>
```



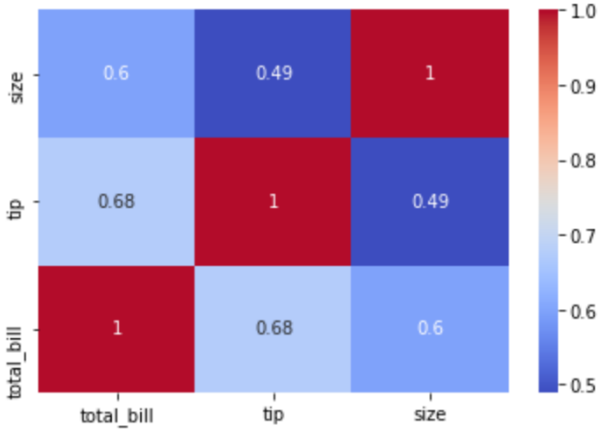
You may notice that the heatmap is not displayed properly as the first and last row is obscured. We can fix it by adjusting the y axis limit to make it better. You can check the original `ylim` values and then set the new one:

Figure 70: Adjusting the size of the heatmap

```
In [79]: ax=sns.heatmap(tips.corr(),cmap='coolwarm',annot=True)
         ax.get_ylim()

Out[79]: (2.5, 0.5)
```




```
In [83]: ax=sns.heatmap(tips.corr(),cmap='coolwarm',annot=True)
         ax.set_ylim(0,3)

Out[83]: (0, 3)
```



Before creating charts, we can set modify the theme. Check out https://seaborn.pydata.org/generated/seaborn.set_theme.html to understand how you can set the parameters. Alternatively you can use the `sns.set_style()` method to set the aesthetic style of the plots. For "`Style`" can choose `None` or one of `darkgrid`, `whitegrid`, `dark`, `white` or `ticks`. Here we will stick with the default theme.

```
# Apply the default theme

sns.set_theme()
```

or

```
sns.set_style('whitegrid')
```

When you are doing your plotting, you can adjust your plots as you want. The customising can be done through the argument set for each function and you can take a look at the documentation to understand how you can assign the relevant keyword arguments. In addition, you can do this through *Matplotlib*. You can create the figure and axes and then set the parameters as we covered in the *Matplotlib* section. For the aesthetic requirement especially the colour palettes, please check https://seaborn.pydata.org/tutorial.html# to see what are available for you to choose.

1.8. Summary

To wrap up, we have covered three main ways of implementing visualisation in Python, all of which are underpinned by *Matplotlib*. Thus, I would recommend you learn *Matplotlib* first and then the other libraries are much easier to learn since they are sharing some APIs. In addition, this tutorial mostly covers the introductory content of each library by extracting some key aspects from the official library documentation. If you want to go deeper or create more complex visualisations, you can refer to the documentation or with more advanced tutorials from the official web pages. TL;DR the content here covers many of the most common applications, but as with anything you can go much deeper into this subject.