Start from Today – 4

Advanced Data Visualization

Visualization is a very integral part of data science. It helps in communicating a pattern or a relationship that cannot be seen by looking at raw data. It's easier for a person to remember a picture and recollect it as compared to lines of text. This holds true for data too.

We'll cover the following topics:

• Controlling the properties of a plot

• Combining multiple plots

• Styling your plots

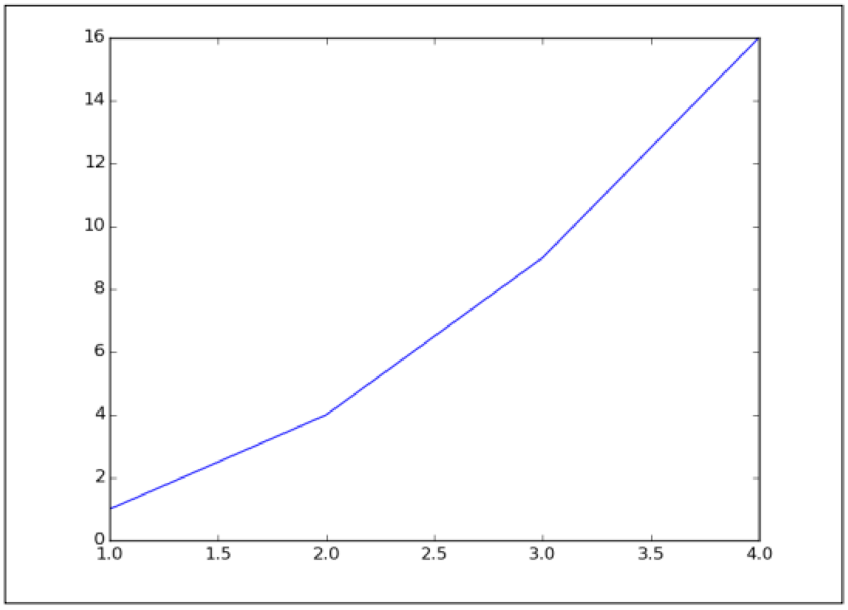
• Creating various advanced visualizations

Controlling the line properties of a chart

There are many properties of a line that can be set, such as the color, dashes, and several others. There are essentially three ways of doing this. Let's take a simple line chart as an example:

```
>>> plt.plot([1,2,3,4], [1,4,9,16])
>>> plt.show()
```

After the preceding code is executed we'll get the following output:

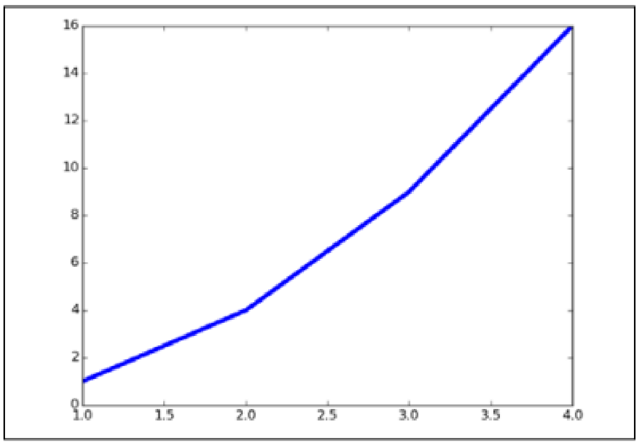


Using keyword arguments

We can use arguments within the plot function to set the property of the line:

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> import pandas.tools.rplot as rplot


>>> plt.plot([1, 2, 3, 4], [1, 4, 9, 16], linewidth=4.0)  # increasing
        # the line width
>>> plt.show()
```

After the preceding code is executed we'll get the following output:

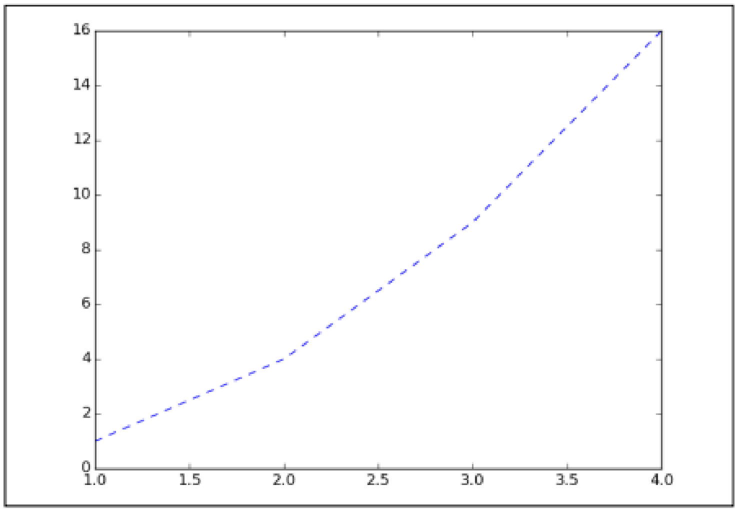


Using the setter methods

The plot function returns the list of line objects, for example `line1, line2 = plot(x1,y1,x2,y2)`. Using the line setter method of line objects we can define the property that needs to be set:

```
>>> line, = plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> line.set_linestyle('--') # Setting dashed lines
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



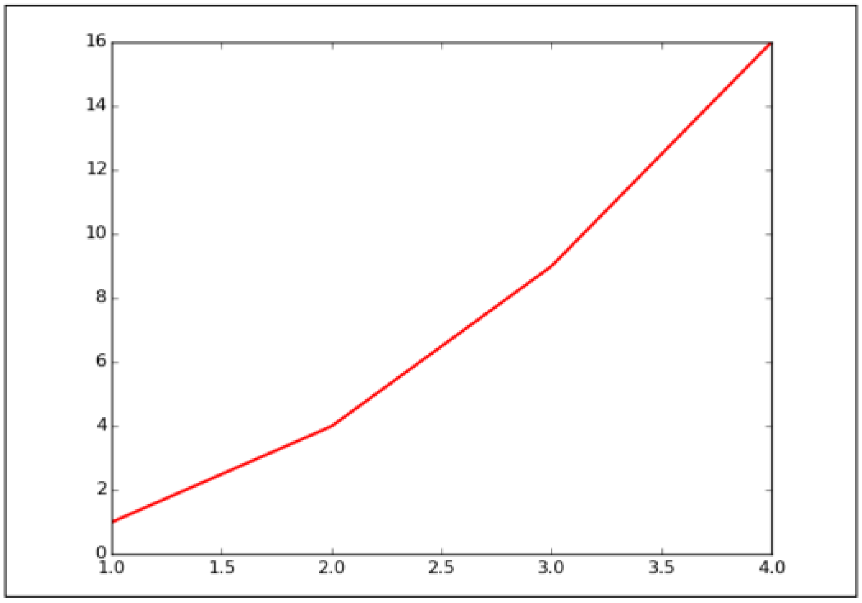
You can view the acceptable line style at `http://matplotlib.org/api/lines_api.html`.

Using the setp() command

The `setp()` command can be used to set multiple properties of a line:

```
>>> line, = plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> plt.setp(line, color='r', linewidth=2.0)  # setting the color
         # and width of the line
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



Creating multiple plots

One very useful feature of matplotlib is that it makes it easy to plot multiple plots, which can be compared to each other:

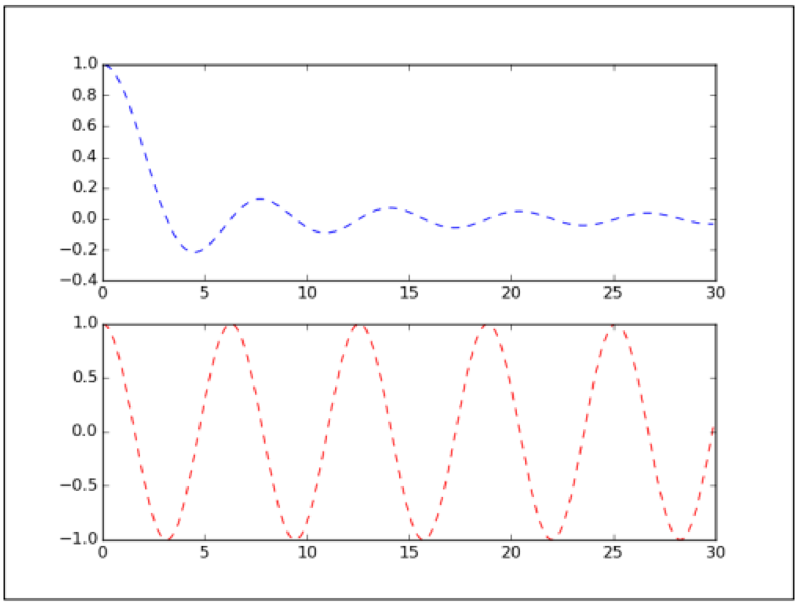
```
>>> p1 = np.arange(0.0, 30.0, 0.1)
```

```
>>> plt.subplot(211)
```

```
>>> plt.plot(p1, np.sin(p1)/p1, 'b--')
```

```
>>> plt.subplot(212)
>>> plt.plot(p1, np.cos(p1), 'r--')
>>> plt.show()
```

In the preceding code, we use a subplot function is used to plot multiple plots that need to be compared. A subplot with a value of `211` means that there will be two rows, one column, and one figure:

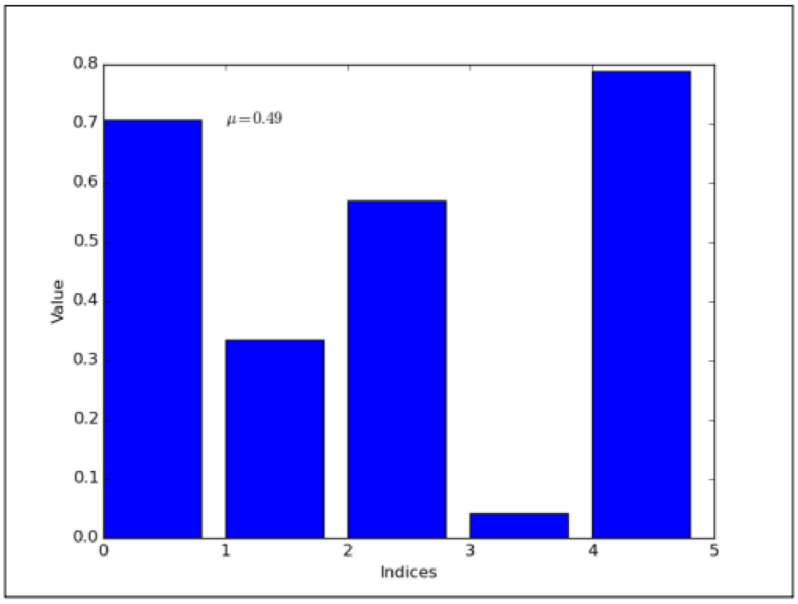


Playing with text

Adding text to your chart can be done by using a simple matplotlib function. You only have to use the `text()` command to add it to the chart:

```
>>> # Playing with text
>>> n = np.random.random_sample((5,))

>>> plt.bar(np.arange(len(n)), n)
>>> plt.xlabel('Indices')
>>> plt.ylabel('Value')


>>> plt.text(1, .7, r'$\mu=' + str(np.round(np.mean(n), 2)) + ' $')

>>> plt.show()
```

In the preceding code, the `text()` command is used to add text within the plot:

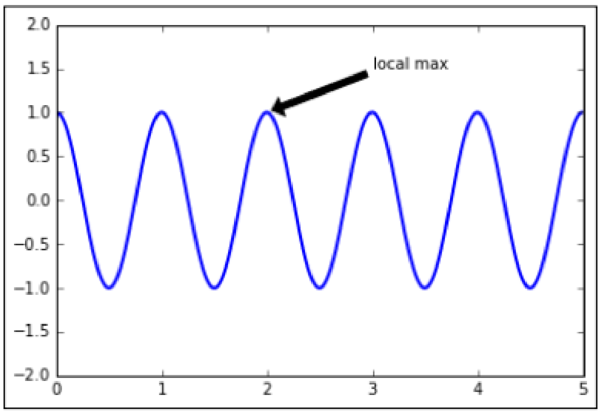


The first parameter takes the *x* axis value and the second parameter takes the *y* axis value. The third parameter is the text that needs to be added to the plot. The latex expression has been used to plot the `mu` mean within the plot.

A certain section of the chart can be annotated by using the annotate command. The annotate command will take the text, the position of the section of plot that needs to be pointed at, and the position of the text:

```
>>> ax = plt.subplot(111)
>>> t = np.arange(0.0, 5.0, 0.01)
>>> s = np.cos(2*np.pi*t)
>>> line, = plt.plot(t, s, lw=2)
>>> plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
                 arrowprops=dict(facecolor='black', shrink=0.05),
                 )
>>> plt.ylim(-2,2)
>>> plt.show()
```

After the preceding code is executed we'll get the following output:

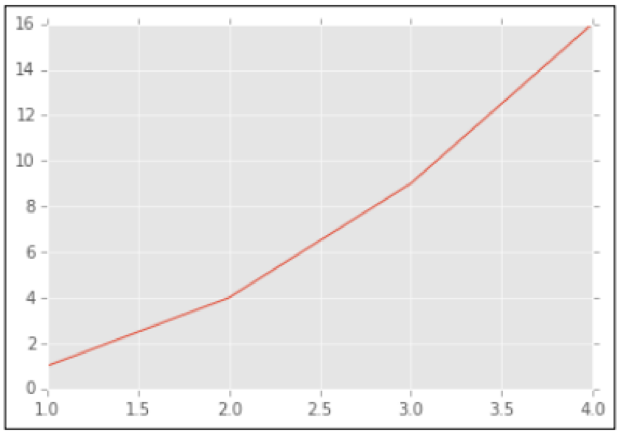


Styling your plots

The style package within the matplotlib library makes it easier to change the style of the plots that are being plotted. It is very easy to change to the famous `ggplot` style of the R language or use the Nate Silver's website `http://fivethirtyeight.com/` for `fivethirtyeight` style. The following example shows the plotting of a simple line chart with the `ggplot` style:

```
>>> plt.style.use('ggplot')
>>> plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> plt.show()
```

After the preceding code is executed we'll get the following output:

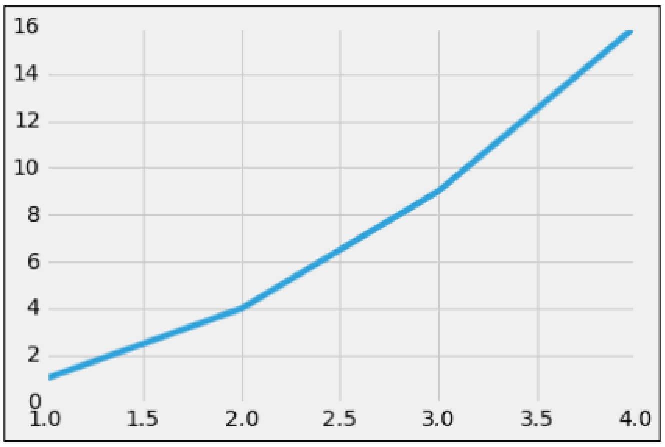


In the preceding code, `plt.style.use()` is used to set the style of the plot. It is a global set, so after it is executed, all the plots that follow will have the same style.

The following code gives the popular `fivethirtyeight` style, which is Nate Silver's website on **data journalism**, where his team write articles on various topics by applying data science:

```
>>> plt.style.use('fivethirtyeight')
>>> plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> plt.show()
```

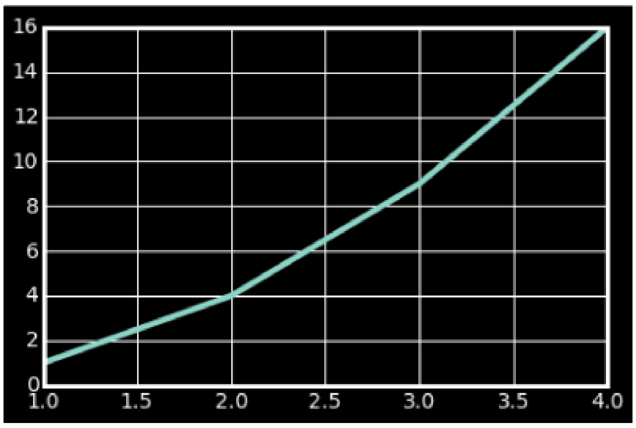
After the preceding code is executed we'll get the following output:



Sometimes, you just want a specific block of code to have a particular style and the rest of the plots in the code to have the default style. This can be achieved using the `plt.style.context` function and the style can be specified within it. Once the following code is executed, only the plot that is specified within it is plotted with the given style:

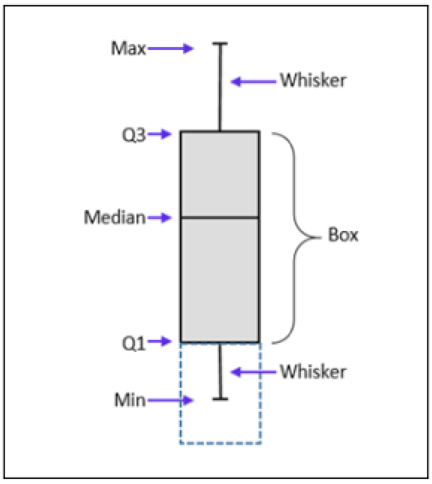
```
>>> with plt.style.context(('dark_background')):
        plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



Box plots

A box plot is a very good plot to understand the spread, median, and outliers of data:



The various parts of the preceding figure are explained as follows:

- **Q3**: This is the 75th percentile value of the data. It's also called the upper hinge.
- **Q1**: This is the 25th percentile value of the data. It's also called the lower hinge.
- **Box**: This is also called a step. It's the difference between the upper hinge and the lower hinge.

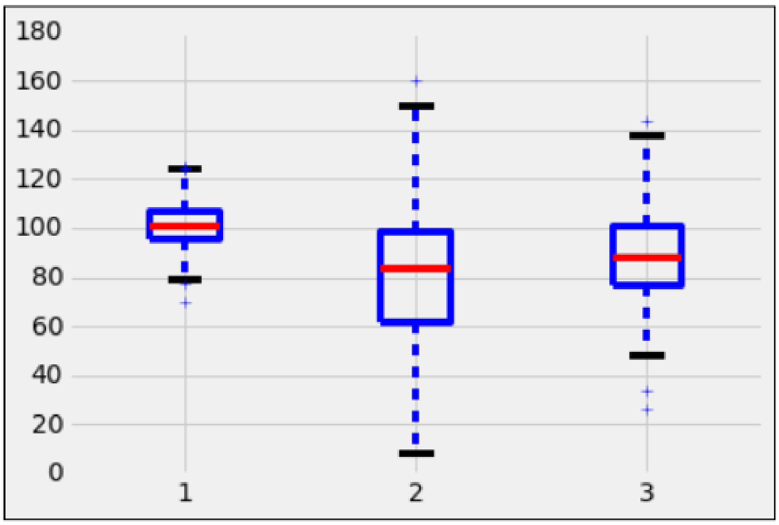
- **Median**: This is the midpoint of the data.
- **Max**: This is the upper inner fence. It is 1.5 times the step above **Q3**.
- **Min**: This is the lower inner fence. It is 1.5 times the step below **Q1**.

Any value that is greater than **Max** or lesser than **Min** is called an outlier, which is also known as a flier.

The following code will create some data, and by using the `boxplot` function we'll create box plots:

```
>>> ## Creating some data
>>> np.random.seed(10)
>>> box_data_1 = np.random.normal(100, 10, 200)
>>> box_data_2 = np.random.normal(80, 30, 200)
>>> box_data_3 = np.random.normal(90, 20, 200)

>>> ## Combining the different data in a list
>>> data_to_plot = [box_data_1, box_data_2, box_data_3]

>>> # Create the boxplot
>>> bp = plt.boxplot(data_to_plot)
```

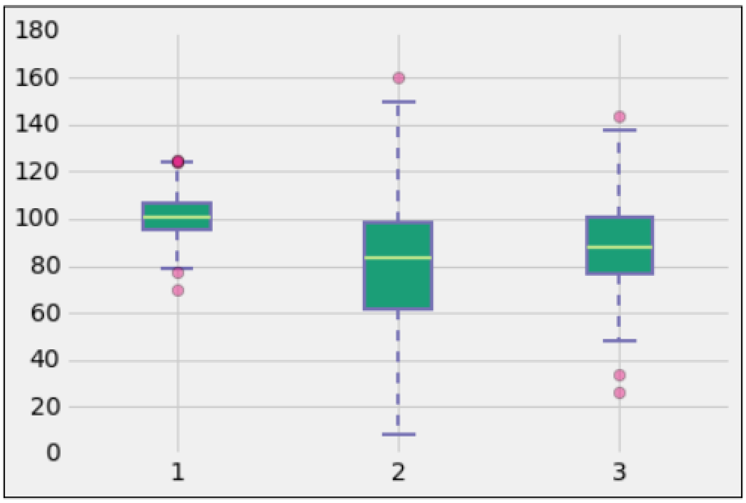
After the preceding code is executed we'll get the following output:



The bp variable in the boxplot function is a Python dictionary with key values such as boxes, whiskers, fliers, caps, and median. The values in the keys represent the different components of the box plot and their properties. The properties can be accessed and altered appropriately to style the box plot to your liking. The following code gives you an example of how to style your boxplot:

```
>>> ## add patch_artist=True option to ax.boxplot()
>>> ## to get fill color
>>> bp = plt.boxplot(data_to_plot, patch_artist=True)


>>> ## change outline color, fill color and linewidth of the boxes
>>> for box in bp['boxes']:
       # change outline color
       box.set( color='#7570b3', linewidth=2)
       # change fill color
       box.set( facecolor = '#1b9e77' )


>>> ## change color and linewidth of the whiskers
>>> for whisker in bp['whiskers']:
       whisker.set(color='#7570b3', linewidth=2)

>>> ## change color and linewidth of the caps
>>> for cap in bp['caps']:
        cap.set(color='#7570b3', linewidth=2)


>>> ## change color and linewidth of the medians
>>> for median in bp['medians']:
       median.set(color='#b2df8a', linewidth=2)


>>> ## change the style of fliers and their fill
>>> for flier in bp['fliers']:
       flier.set(marker='o', color='#e7298a', alpha=0.5)
```

In the preceding code, we take the key values of boxplots and set their properties in terms of color, line width, and face color. Similarly, we perform the same task for the other components, such as whiskers, caps, medians, and fliers.



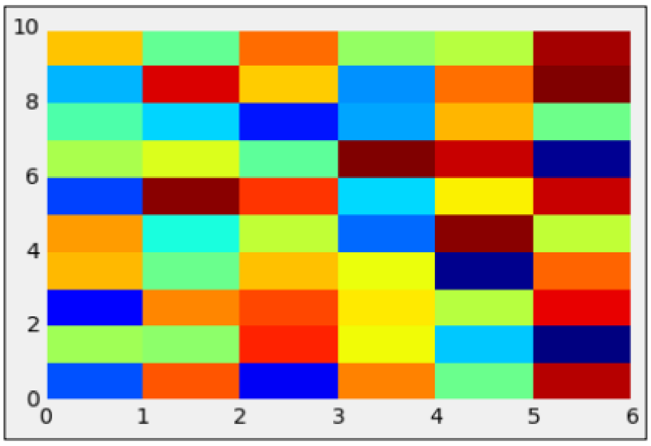
Heatmaps

A heatmap is a graphical representation where individual values of a matrix are represented as colors. A heatmap is very useful in visualizing the concentration of values between two dimensions of a matrix. This helps in finding patterns and gives a perspective of depth.

Let's start off by creating a basic heatmap between two dimensions. We'll create a 10 x 6 matrix of random values and visualize it as a heatmap:

```
>>> # Generate Data
>>> data = np.random.rand(10,6)
>>> rows = list('ZYXWVUTSRQ')  #Ylabel
>>> columns = list('ABCDEF')  #Xlabel

>>> #Basic Heat Map plot
>>> plt.pcolor(data)
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



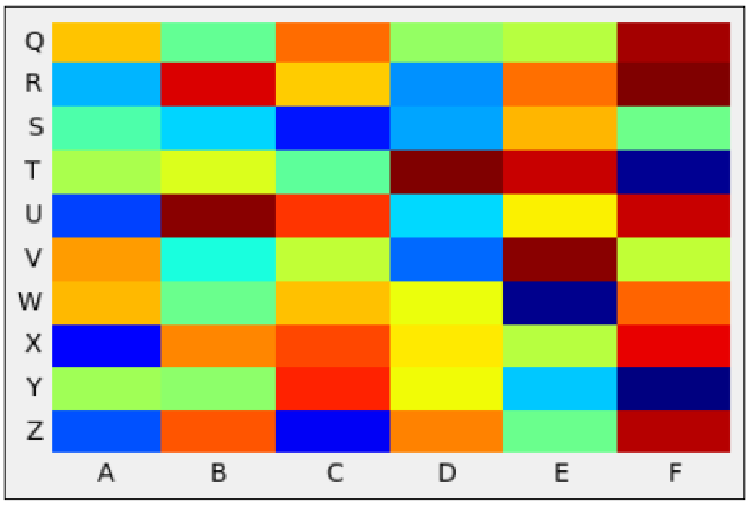
In the preceding code, we used the `pcolor()` function to create the heatmap colors. We'll now add labels to the heatmap:

```
>>> # Add Row/Column Labels
>>> plt.pcolor(data)
>>> plt.xticks(np.arange(0,6)+0.5,columns)
>>> plt.yticks(np.arange(0,10)+0.5,rows)
>>> plt.show()
```

In the preceding code, we used the `pcolor()` function to create the heatmap colors. We'll now add labels to the heatmap:

```
>>> # Add Row/Column Labels
>>> plt.pcolor(data)
>>> plt.xticks(np.arange(0,6)+0.5,columns)
>>> plt.yticks(np.arange(0,10)+0.5,rows)
>>> plt.show()
```

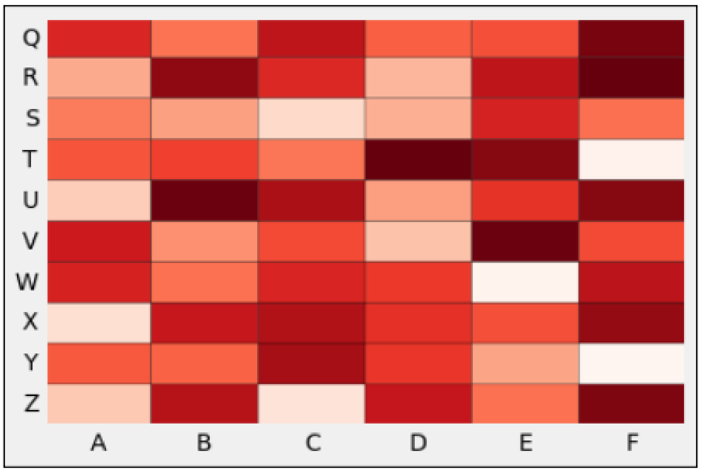
After the preceding code is executed we'll get the following output:



We'll now adjust the color of the heatmap to make it more visually representative. This will help us to understand the data:

```
>>> # Change color map
>>> plt.pcolor(data,cmap=plt.cm.Reds,edgecolors='k')
>>> plt.xticks(np.arange(0,6)+0.5,columns)
>>> plt.yticks(np.arange(0,10)+0.5,rows)
>>> plt.show()
```

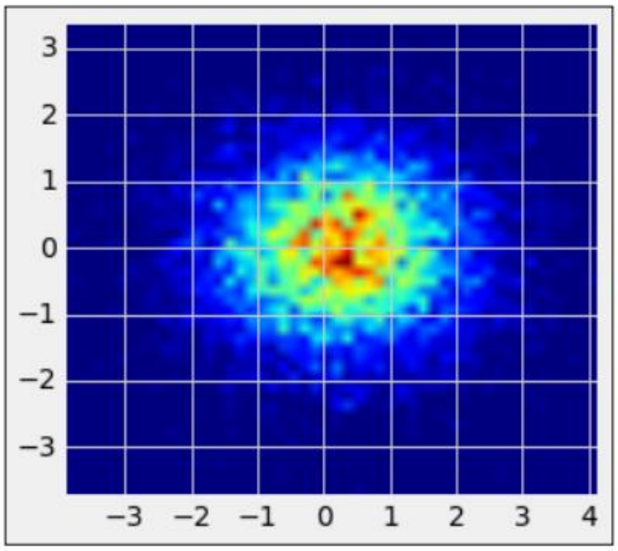
After the preceding code is executed we'll get the following output:



In some instances, there might be a huge number of values that need to be plotted on the heatmap. This can be done by binning the values first and then using the following code to plot it:

```
>>> # Generate some test data
>>> x = np.random.randn(8873)
>>> y = np.random.randn(8873)

>>> heatmap, xedges, yedges = np.histogram2d(x, y, bins=50)
>>> extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]


>>> plt.imshow(heatmap, extent=extent)
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



In the preceding code, the `histogram2d` function helped in binning the the 2D values. Post this, we feed the values to the heatmap to get the preceding plot. Since we used the `randn()`, the values generated were random normally distributed numbers, which means that the concentration of numbers will be more toward the mean. This can be seen in the preceding plot, which shows the center to be red and the exterior area to be blue.

Scatter plots with histograms

We can combine a simple scatter plot with histograms for each axis. These kinds of plots help us see the distribution of the values of each axis.

Let's generate some randomly distributed data for the two axes:

```
>>> from matplotlib.ticker import NullFormatter
>>> # the random data
>>> x = np.random.randn(1000)
>>> y = np.random.randn(1000)
```

A `NullFormatter` object is created, which will be used for eliminating the *x* and *y* labels of the histograms:

```
>>> nullfmt    = NullFormatter()         # no labels
```

The following code defines the size, height, and width of the scatter and histogram plots:

```
>>> # definitions for the axes
>>> left, width = 0.1, 0.65
>>> bottom, height = 0.1, 0.65
>>> bottom_h = left_h = left+width+0.02

>>> rect_scatter = [left, bottom, width, height]
>>> rect_histx = [left, bottom_h, width, 0.2]
>>> rect_histy = [left_h, bottom, 0.2, height]
```

Once the size and height are defined, the axes are plotted for the scatter plot as well as both the histograms:

```
>>> # start with a rectangular Figure
>>> plt.figure(1, figsize=(8,8))

>>> axScatter = plt.axes(rect_scatter)
>>> axHistx = plt.axes(rect_histx)
>>> axHisty = plt.axes(rect_histy)
```

The histograms' *x* and *y* axis labels are eliminated by using the `set_major_formatter` method, and by assigning the `NullFormatter` object to it, the scatter plot is plotted:

```
>>> # no labels
>>> axHistx.xaxis.set_major_formatter(nullfmt)
>>> axHisty.yaxis.set_major_formatter(nullfmt)

>>> # the scatter plot:
>>> axScatter.scatter(x, y)
```

The limits of the *x* and *y* axes are computed using the following code, where the max of the *x* and *y* values are taken. The max value is then divided by the bin, then one is added to it before it is again multiplied with the bin value. This is done so there is some space ahead of the `max` value:

```
>>> # now determine nice limits by hand:
>>> binwidth = 0.25
>>> xymax = np.max( [np.max(np.fabs(x)), np.max(np.fabs(y))] )
>>> lim = ( int(xymax/binwidth) + 1) * binwidth
```

The limit value that is calculated is then assigned to the `set_xlim` method of the `axScatter` object:

```
>>> axScatter.set_xlim( (-lim, lim) )
>>> axScatter.set_ylim( (-lim, lim) )
```

The `bins` variable creates a list of interval values, which will be used with the histograms:

```
>>> bins = np.arange(-lim, lim + binwidth, binwidth)
```

The histograms are plotted and the one that is horizontal is set using the orientation parameter:

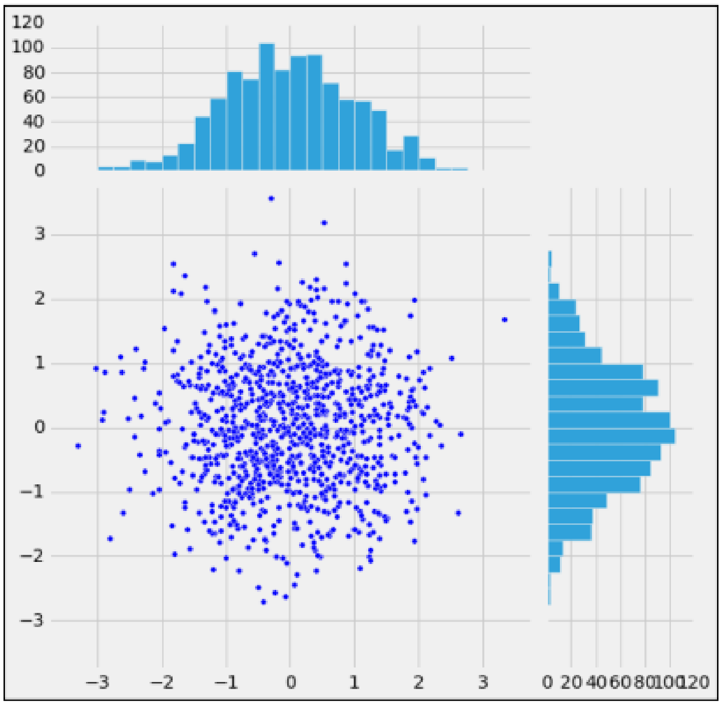
```
>>> axHistx.hist(x, bins=bins)
>>> axHisty.hist(y, bins=bins, orientation='horizontal')
```

The limit value of the scatter plot is fetched and then assigned to the limit methods of the histogram:

```
>>> axHistx.set_xlim( axScatter.get_xlim() )
>>> axHisty.set_ylim( axScatter.get_ylim() )

>>> plt.show()
```

```
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



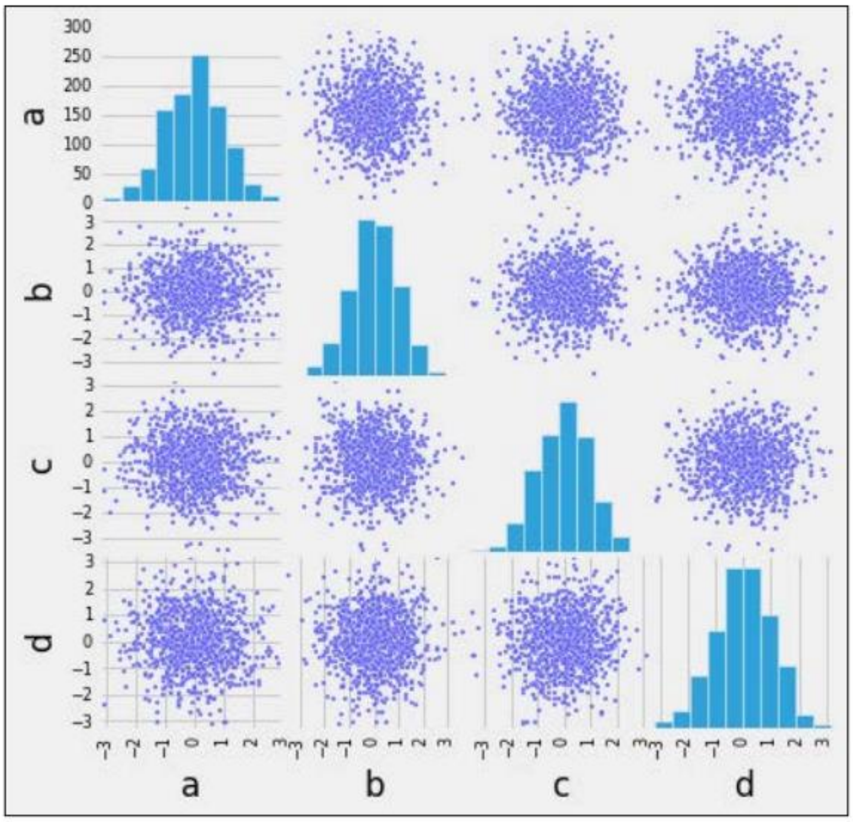
A scatter plot matrix

A scatter plot matrix can be formed for a collection of variables where each of the variables will be plotted against each other. The following code generates a DataFrame `df`, which consists of four columns with normally distributed random values and column names named from `a` to `d`:

```
>>> df = pd.DataFrame(np.random.randn(1000, 4),
         columns=['a', 'b', 'c', 'd'])
```

```
>>> spm = pd.tools.plotting.scatter_matrix(df, alpha=0.2,
        figsize=(6, 6), diagonal='hist')
```

After the preceding code is executed we'll get the following output:

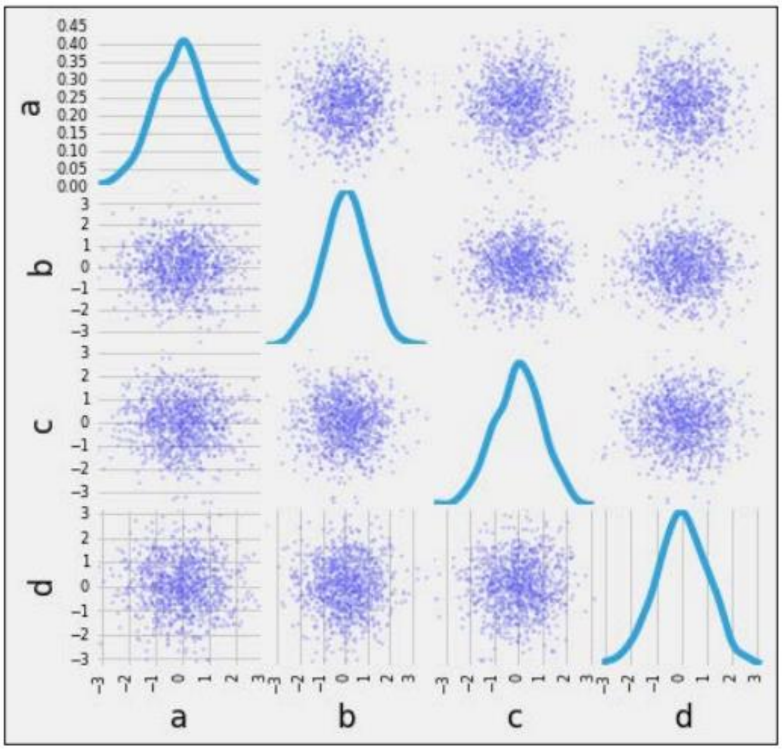


The `scatter_matrix()` function helps in plotting the preceding figure. It takes in the data frame object and the required parameters that are defined to customize the plot. You would have observed that the diagonal graph is defined as a histogram, which means that in the section of the plot matrix where the variable is against itself, a histogram is plotted.

Instead of the histogram, we can also use the kernel density estimation for the diagonal:

```
>>> spm = pd.tools.plotting.scatter_matrix(df, alpha=0.2,
         figsize=(6, 6), diagonal='kde')
```

After the preceding code is executed we'll get the following output:



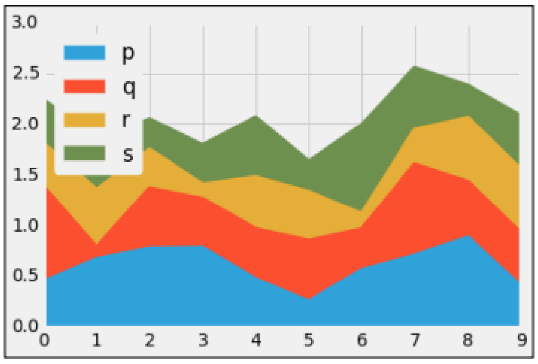
The kernel density estimation is a nonparametric way of estimating the probability density function of a random variable. It basically helps in understanding whether the data is normally distributed and the side toward which it is skewed.

Area plots

An area plot is useful for comparing the values of different factors across a range. The area plot can be stacked in nature, where the areas of the different factors are stacked on top of each other. The following code gives an example of a stacked area plot:

```
>>> df = pd.DataFrame(np.random.rand(10, 4),
        columns=['p', 'q', 'r', 's'])


>>> df.plot(kind='area');
```

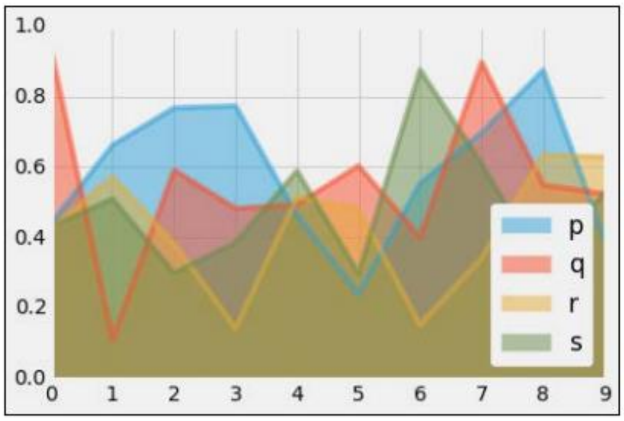
After the preceding code is executed we'll get the following output:



To remove the stack of area plot, you can use the following code:

```
>>> df.plot(kind='area', stacked=False);
```

After the preceding code is executed we'll get the following output:



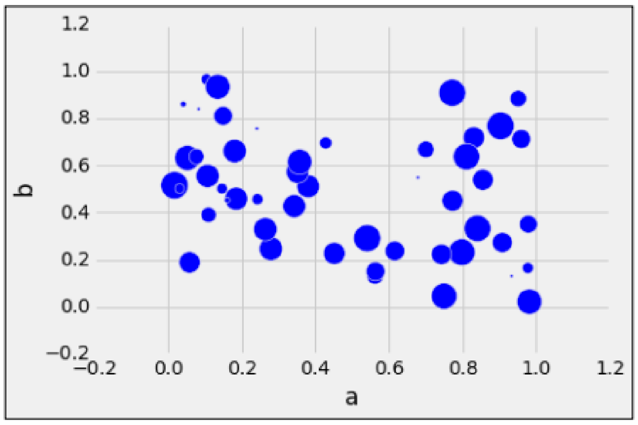
Bubble charts

A bubble chart is basically a scatter plot with an additional dimension. The additional dimension helps in setting the size of the bubble, which means that the greater the size of the bubble, the larger the value that represents the bubble. This kind of a chart helps in analyzing the data of three dimensions.

The following code creates a sample data of three variables and this data is then fed to the `plot()` method where its kind is mentioned as a scatter and s is the size of the bubble:

```
>>> plt.style.use('ggplot')
>>> df = pd.DataFrame(np.random.rand(50, 3), columns=['a', 'b', 'c'])
>>> df.plot(kind='scatter', x='a', y='b', s=df['c']*400);
```

After the preceding code is executed we'll get the following output:



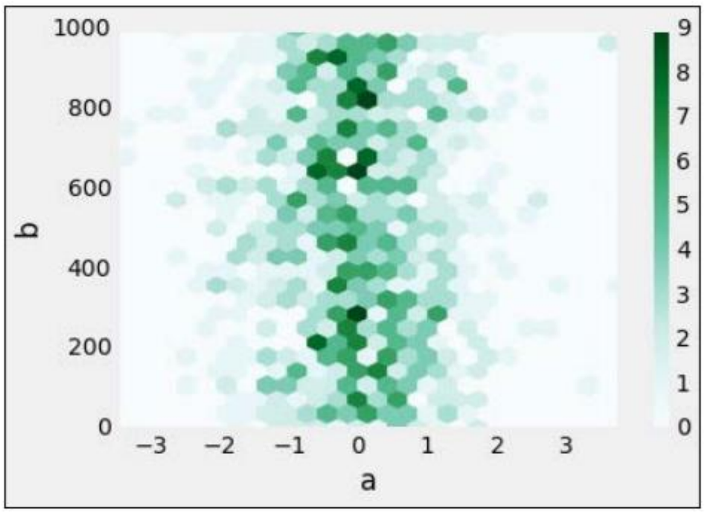
Hexagon bin plots

A hexagon bin plot can be created using the `DataFrame.plot()` function and `kind = 'hexbin'`. This kind of plot is really useful if your scatter plot is too dense to interpret. It helps in binning the spatial area of the chart and the intensity of the color that a hexagon can be interpreted as points being more concentrated in this area.

The following code helps in plotting the hexagon bin plot, and the structure of the code is similar to the previously discussed plots:

```
>>> df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

>>> df['b'] = df['b'] + np.arange(1000)

>>> df.plot(kind='hexbin', x='a', y='b', gridsize=25)
```

After the preceding code is executed we'll get the following output:



Trellis plots

A Trellis plot is a layout of smaller charts in a grid with consistent scales. Each smaller chart represents an item in a category, named conditions. The data displayed on each smaller chart is conditional for the items in the category.

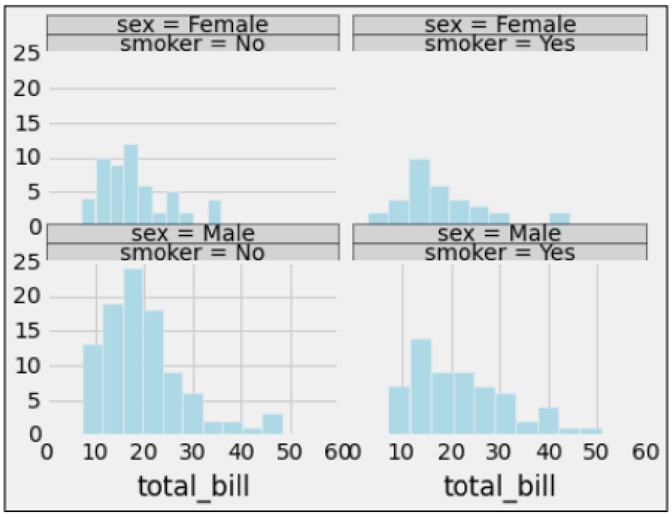
Trellis plots are useful for finding structures and patterns in complex data. The grid layout looks similar to a garden trellis, hence the name Trellis plots.

The following code helps in plotting a trellis chart where for each combination of sex and smoker/nonsmoker:

```
>>> tips_data = pd.read_csv('Data/tips.csv')
>>> plt.figure()
>>> plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
>>> plot.add(rplot.TrellisGrid(['sex', 'smoker']))
>>> plot.add(rplot.GeomHistogram())
>>> plot.render(plt.gcf())
```

After the preceding code is executed we'll get the following output:

After the preceding code is executed we'll get the following output:

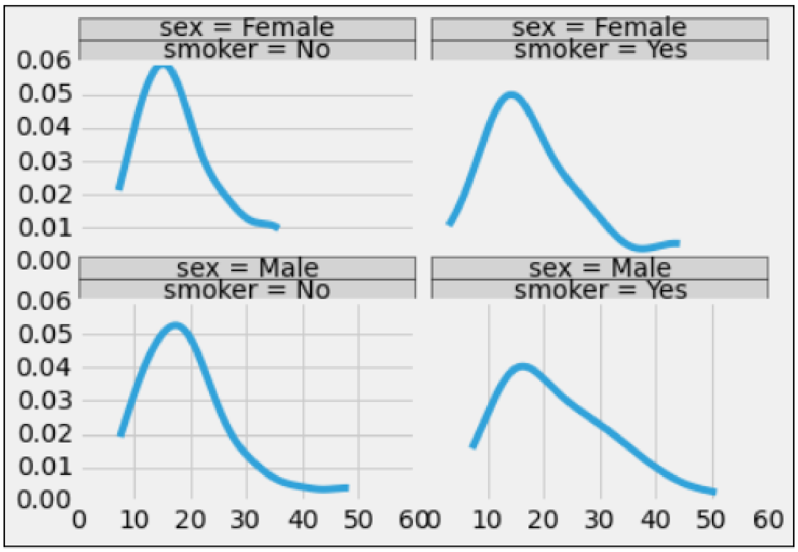


In the preceding code, `rplot.RPlot` takes the `tips_data` object. Also, the *x* and *y* axis values are defined. After this, the Trellis grid is defined based on the smoker and sex. In the end, we use `GeomHistogram()` to plot a histogram.

To change the Trellis plot to a kernel density estimate, we can use the following code:

```
>>> plt.figure()
>>> plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
>>> plot.add(rplot.TrellisGrid(['sex', 'smoker']))
>>> plot.add(rplot.GeomDensity())
>>> plot.render(plt.gcf())
```

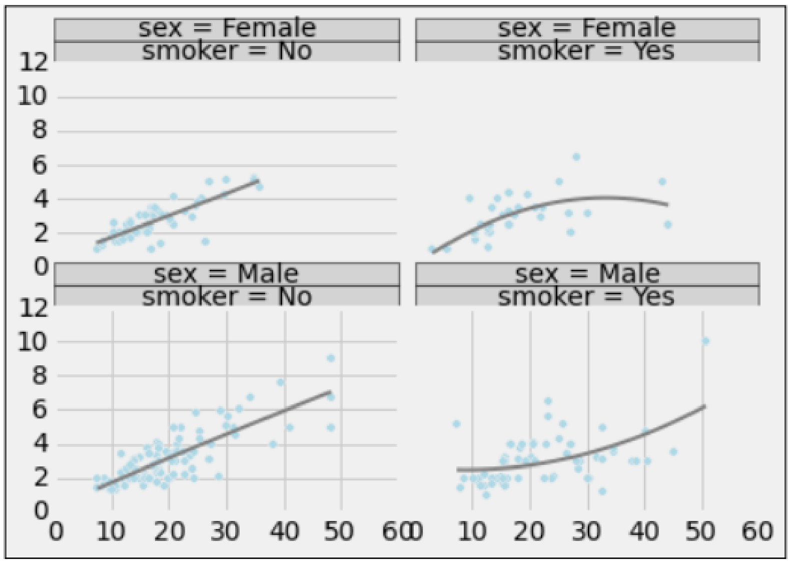
After the preceding code is executed we'll get the following output:



We could also have a scatter plot with a poly fit line on it:

```
>>> plt.figure()
>>> plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
>>> plot.add(rplot.TrellisGrid(['sex', 'smoker']))
>>> plot.add(rplot.GeomScatter())
>>> plot.add(rplot.GeomPolyFit(degree=2))
>>> plot.render(plt.gcf())
```

After the preceding code is executed we'll get the following output:

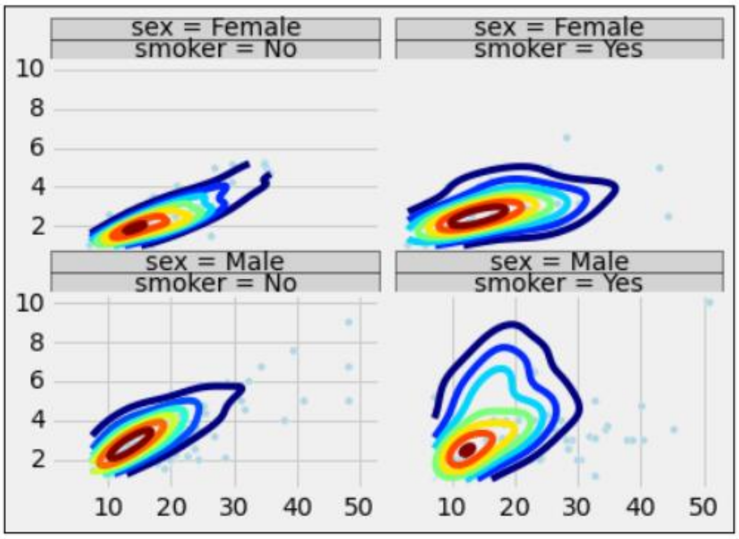


The code is similar to the previous example. The only difference is that `GeomScatter()` and `GeomPolyFit` are used to get the fit line on the plot.

The scatter plot can be combined with a 2D kernel density plot by using the following code:

```
>>> plt.figure()
>>> plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
>>> plot.add(rplot.TrellisGrid(['sex', 'smoker']))
>>> plot.add(rplot.GeomScatter())
>>> plot.add(rplot.GeomDensity2D())
>>> plot.render(plt.gcf())
```

After the preceding code is executed we'll get the following output:

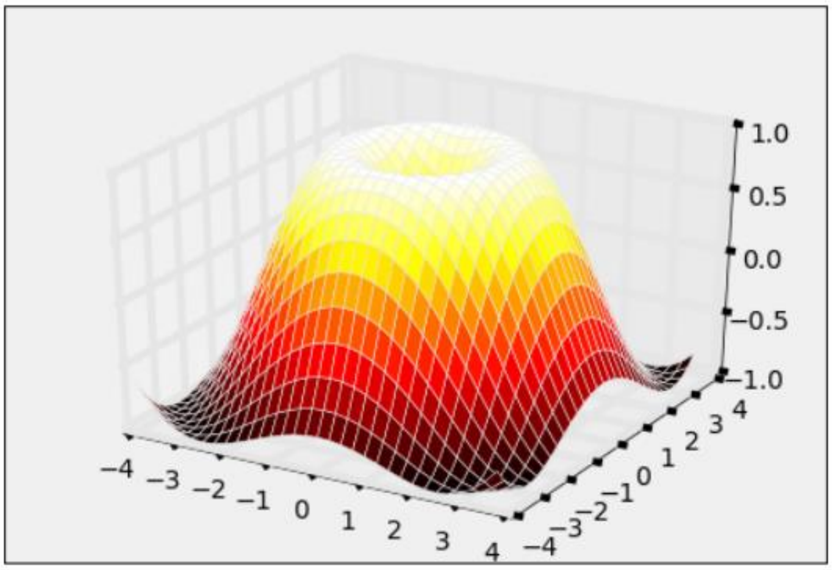


A 3D plot of a surface

We'll now plot a 3D plot, where the `Sin` function is plotted against the sum of the square values of the two axes:

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> X = np.arange(-4, 4, 0.25)
>>> Y = np.arange(-4, 4, 0.25)
>>> X, Y = np.meshgrid(X, Y)
>>> R = np.sqrt(X**2 + Y**2)
>>> Z = np.sin(R)
>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

After the preceding code is executed we'll get the following output:



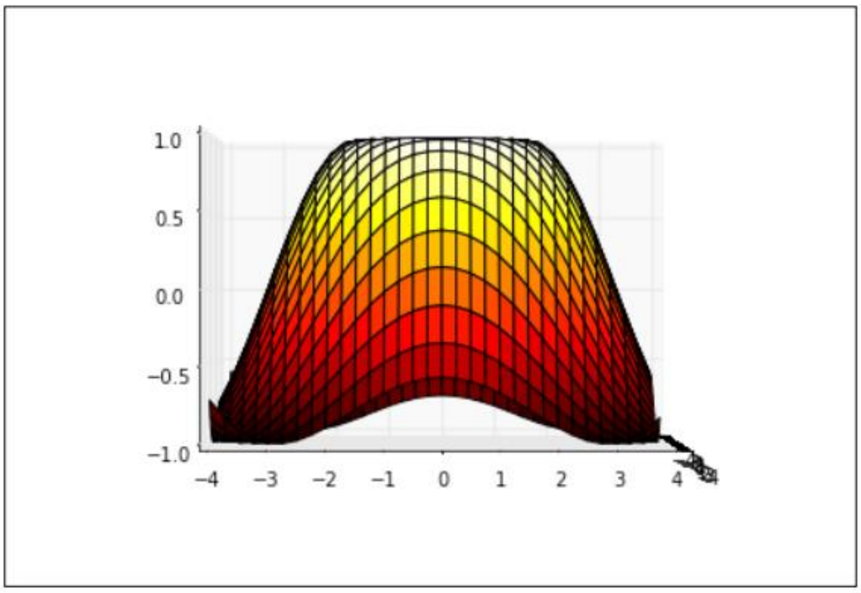
In the preceding code, we defined the *x* and *y* axes with values ranging from -4 to 4. We created a coordinate matrix with `meshgrid()`, then squared the values of *x* and *y*, and finally, summed them up. This was then fed to the `plot_surface` function. The `rstride` and `cstride` parameters in simple terms help in sizing the cell on the surface.

Let's adjust the view using `view_int`. The following is the view at `0` degree elevation and `0` degree angle:

```
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.view_init(elev=0., azim=0)
>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

After the preceding code is executed we'll get the following output:

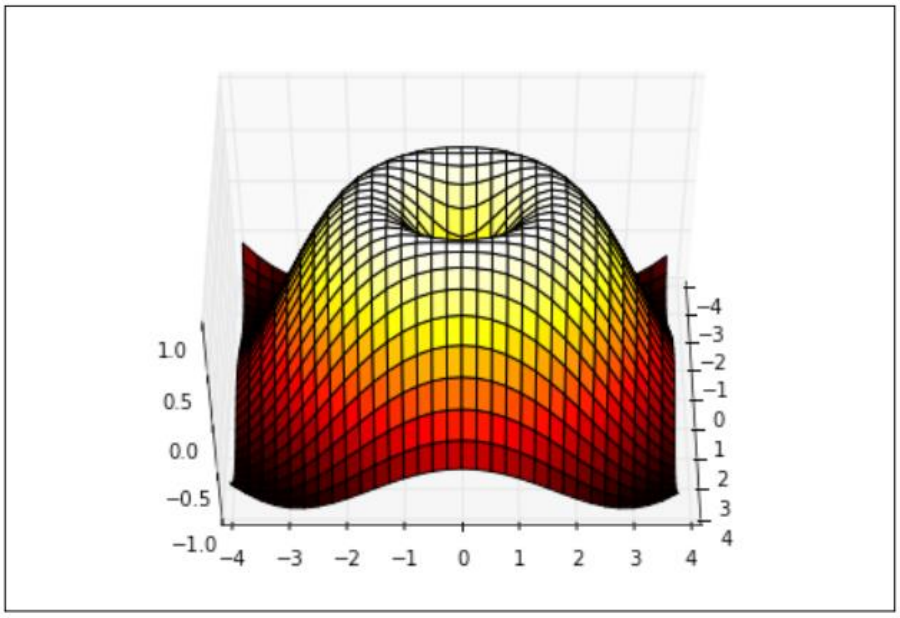
After the preceding code is executed we'll get the following output:



The following is the view at `50` degrees elevation:

```
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.view_init(elev=50., azim=0)
>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

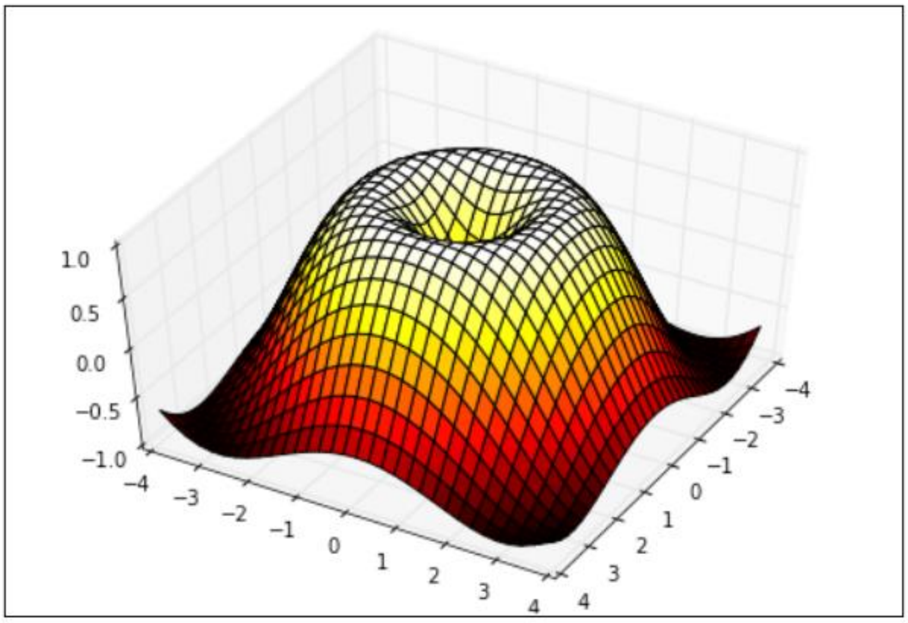
After the preceding code is executed we'll get the following output:



The following is the view at `50` degrees elevation and `30` degrees angle:

```
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.view_init(elev=50., azim=30)
>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```

After the preceding code is executed we'll get the following output:



1.0
0.5
0.0
−0.5
−1.0
−4
−3
−2
−1
0
1
2
3
4