

CHAPTER 7

Data Visualization with matplotlib

After discussing in the previous chapters Python libraries that were responsible for data processing, now it is time for you to see a library that takes care of visualization. This library is matplotlib.

Data visualization is often underestimated in data analysis, but it is actually a very important factor because incorrect or inefficient data representation can ruin an otherwise excellent analysis. In this chapter, you will discover the various aspects of the matplotlib library, including how it is structured, and how to maximize the potential that it offers.

The matplotlib Library

matplotlib is a Python library specializing in the development of two-dimensional charts (including 3D charts). In recent years, it has been widespread in scientific and engineering circles (<http://matplotlib.org>).

Among all the features that have made it the most used tool in the graphical representation of data, there are a few that stand out:

- Extreme simplicity in its use
- Gradual development and interactive data visualization
- Expressions and text in LaTeX
- Greater control over graphic elements
- Export to many formats, such as PNG, PDF, SVG, and EPS

matplotlib is designed to reproduce as much as possible an environment similar to MATLAB in terms of both graphical view and syntactic form. This approach has proved successful, as it has been able to exploit the experience of software (MATLAB) that has been on the market for several years and is now widespread in all professional technical-scientific circles. Not only is matplotlib based on a scheme known and quite familiar to most experts in the field, but also it also exploits those optimizations that over the years have led to a deducibility and simplicity in its use, which makes this library also an excellent choice for those approaching data visualization for the first time, especially those without any experience with applications such as MATLAB or similar.

In addition to simplicity and deducibility, the matplotlib library inherited *interactivity* from MATLAB as well. That is, the analyst can insert command after command to control the gradual development of a graphical representation of data. This mode is well suited to the more interactive approaches of Python as the IPython QtConsole and IPython Notebook (see Chapter 2), thus providing an environment for data analysis that has little to envy from other tools such as Mathematica, IDL, or MATLAB.

The genius of those who developed this beautiful library was to use and incorporate the good things currently available and in use in science. This is not only limited, as we have seen, to the operating mode of MATLAB and similar, but also to models of textual formatting of scientific expressions and symbols represented by LaTeX. Because of its great capacity for display and presentation of scientific expressions, LaTeX has been an irreplaceable element in any scientific publication or documentation, where the need to visually represent expressions like integrals, summations, and derivatives is mandatory. Therefore matplotlib integrates this remarkable instrument in order to improve the representative capacity of charts.

In addition, you must not forget that matplotlib is not a separate application but a library of a programming language like Python. So it also takes full advantage of the potential that programming languages offer. matplotlib looks like a graphics library that allows you to programmatically manage the graphic elements that make up a chart so that the graphical display can be controlled in its entirety. The ability to program the graphical representation allows management of the reproducibility of the data representation across multiple environments and especially when you make changes or when the data is updated.

Moreover, since matplotlib is a Python library, it allows you to exploit the full potential of other libraries available to any developer that implements with this language. In fact, with regard to data analysis, matplotlib normally cooperates with a set of other libraries such as NumPy and pandas, but many other libraries can be integrated without any problem.

Finally, graphical representations obtained through encoding with this library can be exported in the most common graphic formats (such as PNG and SVG) and then be used in other applications, documentation, web pages, etc.

Installation

There are many options for installing the matplotlib library. If you choose to use a distribution of packages like Anaconda or Enthought Canopy, installing the matplotlib package is very simple. For example, with the conda package manager, you have to enter the following:

```
conda install matplotlib
```

If you want to directly install this package, the commands to insert vary depending on the operating system.

On Debian-Ubuntu Linux systems, use this:

```
sudo apt-get install python-matplotlib
```

On Fedora-Redhat Linux systems, use this:

```
sudo yum install python-matplotlib
```

On Windows or MacOS, you should use pip for installing matplotlib.

The IPython and IPython QtConsole

In order to get familiar with all the tools provided by the Python world, I chose to use IPython both from a terminal and from the QtConsole. This is because IPython allows you to exploit the interactivity of its enhanced terminal and, as you will see, IPython QtConsole also allows you to integrate graphics directly inside the console.

To run an IPython session, simply run the following command:

```
ipython
```

```
Python 3.6.3 (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]  
Type "copyright", "credits" or "license" for more information.
```

```
IPython 3.6.3 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]:
```

Whereas if you want to run the Jupyter QtConsole with the ability to display graphics within the line commands of the session, you use:

```
jupyter qtconsole
```

A window with a new open IPython session will immediately appear on the screen, as shown in Figure 7-1.

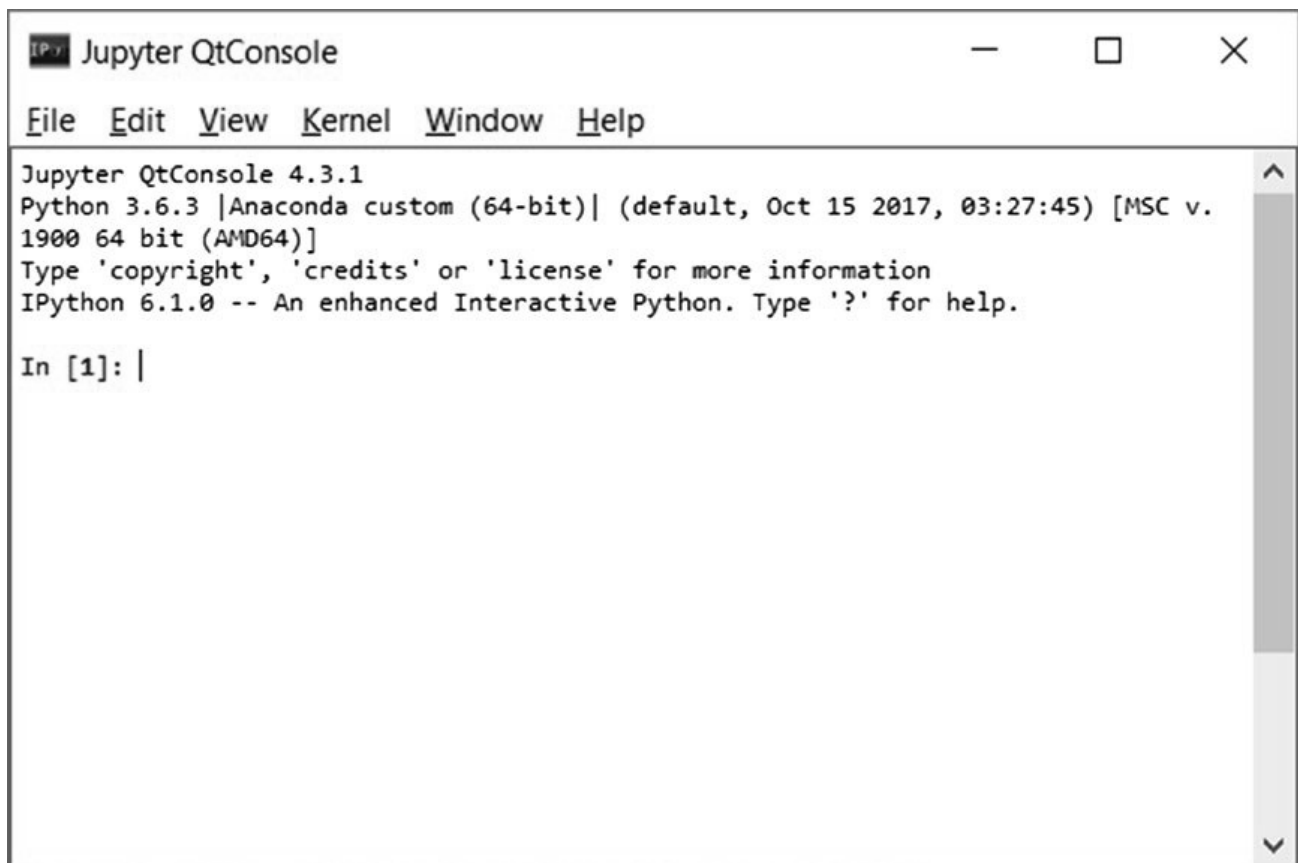


Figure 7-1. The IPython QtConsole

However, if you want to continue using a standard Python session you are free to do so. If you do not like working with IPython and want to continue to use Python from the terminal, all the examples in this chapter will still be valid.

The matplotlib Architecture

One of the key tasks that matplotlib must take on is provide a set of functions and tools that allow representation and manipulation of a *Figure* (the main object), along with all internal objects of which it is composed. However, matplotlib not only deals with graphics but also provides all the tools for the event handling and the ability to animate graphics. So, thanks to these additional features, matplotlib proves to be capable of producing interactive charts based on the events triggered by pressing a key on the keyboard or on mouse movement.

The architecture of matplotlib is logically structured into three layers, which are placed at three different levels (see Figure 7-2). The communication is unidirectional, that is, each layer can communicate with the underlying layer, while the lower layers cannot communicate with the top ones.

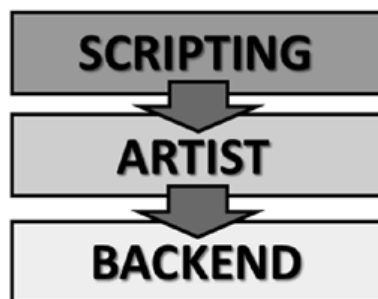


Figure 7-2. *The three layers of the matplotlib architecture*

The three layers are as follows:

- Scripting
- Artist
- Backend

Backend Layer

In the diagram of the matplotlib architecture, the layer that works at the lowest level is the *Backend* layer. This layer contains the matplotlib APIs, a set of classes that play the role of implementation of the graphic elements at a low level.

- FigureCanvas is the object that embodies the concept of drawing area.
- Renderer is the object that draws on FigureCanvas.
- Event is the object that handles user inputs (keyboard and mouse events).

Artist Layer

As an intermediate layer, we have a layer called *Artist*. All the elements that make up a chart, such as the title, axis labels, markers, etc., are instances of the Artist object. Each of these instances plays its role within a hierarchical structure (as shown in Figure 7-3).

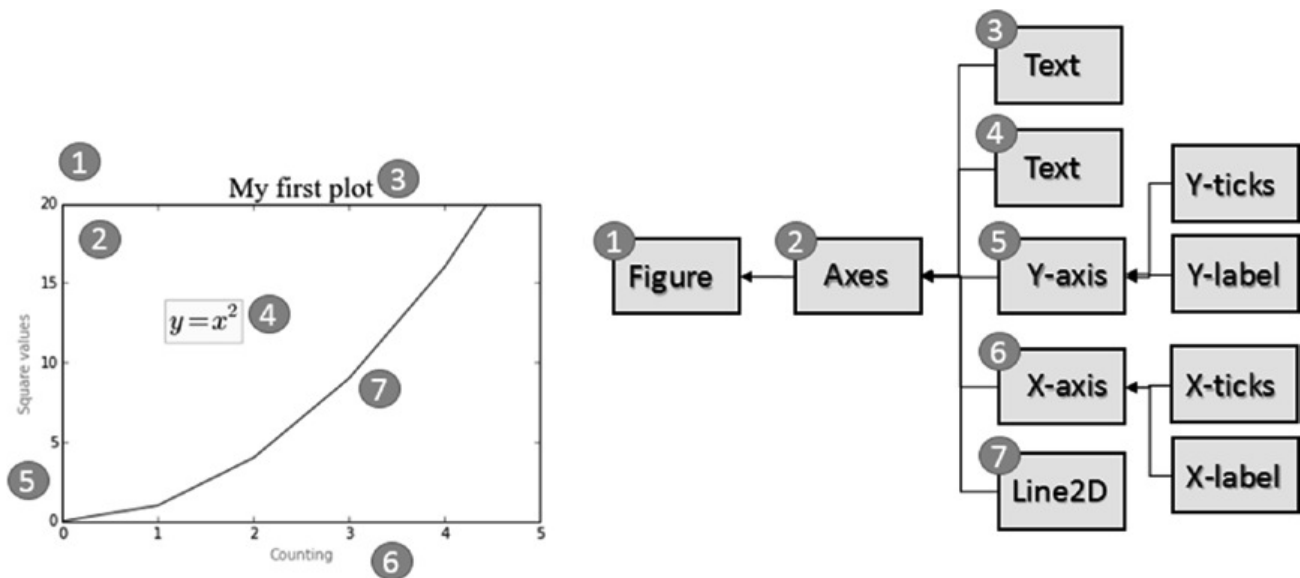


Figure 7-3. Each element of a chart corresponds to an instance of Artist structured in a hierarchy

There are two Artist classes: primitive and composite.

- The *primitive artists* are individual objects that constitute the basic elements to form a graphical representation in a plot, for example a `Line2D`, or as a geometric figure such as a `Rectangle` or `Circle`, or even pieces of text.
- The *composite artists* are those graphic elements present in a chart that are composed of several base elements, namely, the primitive artists. Composite artists are for example the `Axis`, `Ticks`, `Axes`, and `Figure` (see Figure 7-4).

Generally, working at this level you will have to deal often with objects in higher hierarchy as `Figure`, `Axes`, and `Axis`. So it is important to fully understand what these objects are and what role they play within the graphical representation. Figure 7-4 shows the three main Artist objects (composite artists) that are generally used in all implementations performed at this level.

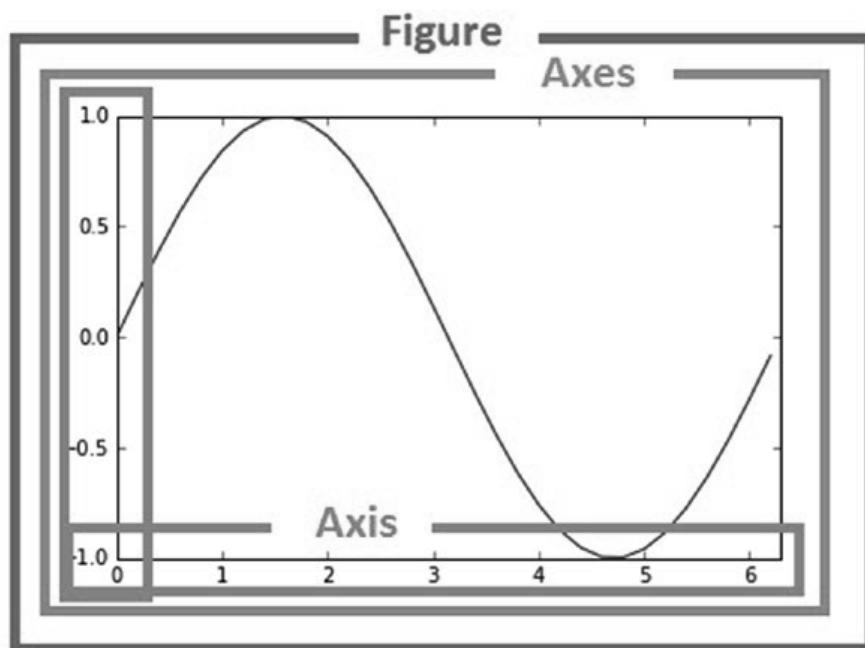


Figure 7-4. The three main artist objects in the hierarchy of the Artist layer

- *Figure* is the object with the highest level in the hierarchy. It corresponds to the entire graphical representation and generally can contain many `Axes`.

- *Axes* is generally what you mean as plot or chart. Each Axis object belongs to only one Figure, and is characterized by two Artist Axis (three in the three-dimensional case). Other objects, such as the title, the x label, and the y label, belong to this composite artist.
- *Axis* objects that take into account the numerical values to be represented on Axes, define the limits and manage the ticks (the mark on the axes) and tick labels (the label text represented on each tick). The position of the tick is adjusted by an object called a *Locator* while the formatting tick label is regulated by an object called a *Formatter*.

Scripting Layer (pyplot)

Artist classes and their related functions (the matplotlib API) are particularly suitable to all developers, especially for those who work on web application servers or develop the GUI. But for purposes of calculation, and in particular for the analysis and visualization of data, the scripting layer is best. This layer consists of an interface called *pyplot*.

pylab and pyplot

In general there is talk of *pylab* and *pyplot*. But what is the difference between these two packages? Pylab is a module that is installed along with matplotlib, while pyplot is an internal module of matplotlib. Often you will find references to one or the other approach.

```
from pylab import *

and

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

Pylab combines the functionality of pyplot with the capabilities of NumPy in a single namespace, and therefore you do not need to import NumPy separately. Furthermore, if you import pylab, pyplot and NumPy functions can be called directly without any reference to a module (namespace), making the environment more similar to MATLAB.


```
plot(x,y)
array([1,2,3,4])
```

Instead of

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

The *pyplot* package provides the classic Python interface for programming the matplotlib library, has its own namespace, and requires the import of the NumPy package separately. This approach is the one chosen for this book; it is the main topic of this chapter; and it will be used for the rest of the book. In fact this choice is shared and approved by most Python developers.

pyplot

The pyplot module is a collection of command-style functions that allow you to use matplotlib much like MATLAB. Each pyplot function will operate or make some changes to the Figure object, for example, the creation of the Figure itself, the creation of a plotting area, representation of a line, decoration of the plot with a label, etc.

Pyplot also is *stateful*, in that it tracks the status of the current figure and its plotting area. The functions called act on the current figure.

A Simple Interactive Chart

To get familiar with the matplotlib library and in a particular way with Pyplot, you will start creating a simple interactive chart. Using matplotlib, this operation is very simple; in fact, you can achieve it using only three lines of code.

But first you need to import the pyplot package and rename it as plt.

```
In [1]: import matplotlib.pyplot as plt
```

In Python, the constructors generally are not necessary; everything is already implicitly defined. In fact when you import the package, the plt object with all its graphics capabilities have already been instantiated and ready to use. In fact, you simply use the plot() function to pass the values to be plotted.

Thus, you can simply pass the values that you want to represent as a sequence of integers.

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

```
Out[2]: [<matplotlib.lines.Line2D at 0xa3eb438>]
```

As you can see, a `Line2D` object has been generated. The object is a line that represents the linear trend of the points included in the chart.

Now it is all set. You just have to give the command to show the plot using the `show()` function.

```
In [3]: plt.show()
```

The result will be the one shown in Figure 7-5. It looks just a window, called the *plotting window*, with a toolbar and the plot represented within it, just as with MATLAB.

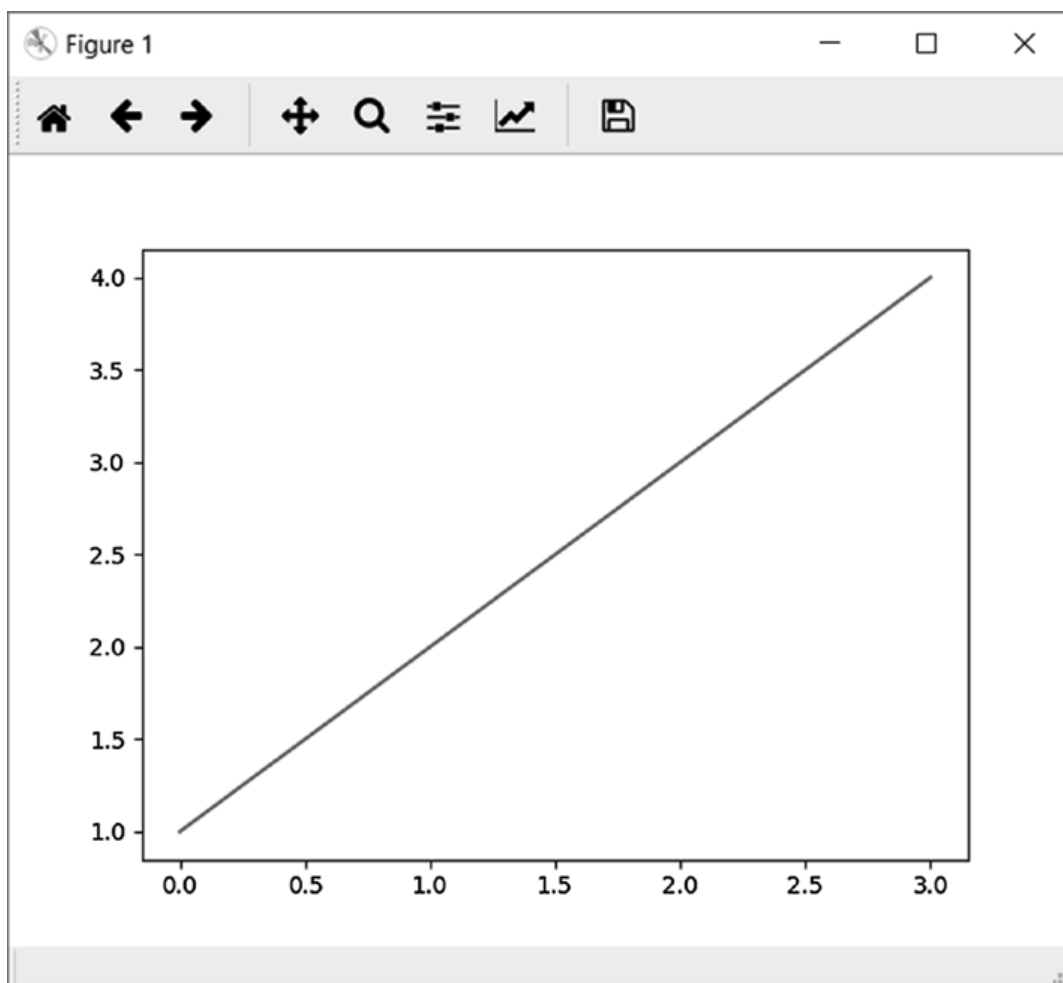

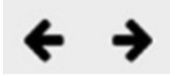







Figure 7-5. *The plotting window*

The Plotting Window

The plotting window is characterized by a toolbar at the top in which there are a series of buttons.

-  Resets the original view
-  Goes to the previous/next view
-  Pans axes with left mouse, zoom with right
-  Zooms to rectangle
-  Configures subplots
-  Saves/exports the figure
-  Edits the axis, curve, and image parameters

The code entered into the IPython console corresponds on the Python console to the following series of commands:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4])
[<matplotlib.lines.Line2D at 0x000000007DABFD0>]
>>> plt.show()
```

If you are using the IPython QtConsole, you may have noticed that after calling the `plot()` function the chart is displayed directly without explicitly invoking the `show()` function (see Figure 7-6).

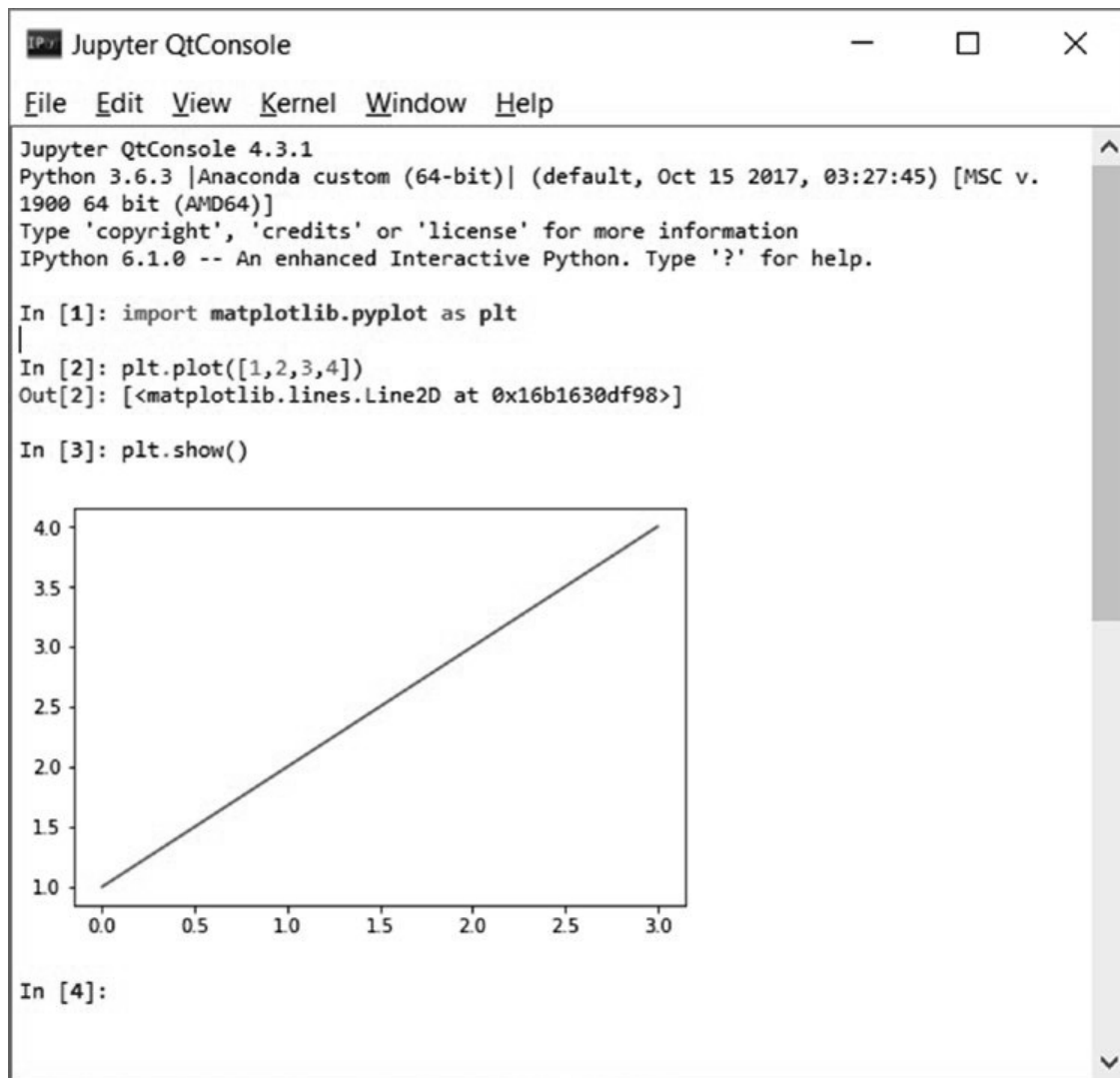


Figure 7-6. The QtConsole shows the chart directly as output

If you pass only a list of numbers or an array to the `plt.plot()` function, matplotlib assumes it is the sequence of y values of the chart, and it associates them to the natural sequence of values x: 0,1,2,3,

Generally a plot represents value pairs (x, y), so if you want to define a chart correctly, you must define two arrays, the first containing the values on the x-axis and the second containing the values on the y-axis. Moreover, the `plot()` function can accept a third argument, which describes the specifics of how you want the point to be represented on the chart.

Set the Properties of the Plot

As you can see in Figure 7-6, the points were represented by a blue line. In fact, if you do not specify otherwise, the plot is represented taking into account a default configuration of the `plt.plot()` function:

- The size of the axes matches perfectly with the range of the input data
- There is neither a title nor axis labels
- There is no legend
- A blue line connecting the points is drawn

Therefore you need to change this representation to have a real plot in which each pair of values (x, y) is represented by a red dot (see Figure 7-7).

If you're working on IPython, close the window to get back to the active prompt for entering new commands. Then you have to call back the `show()` function to observe the changes made to the plot.

```
In [4]: plt.plot([1,2,3,4],[1,4,9,16], 'ro')  
Out[4]: [<matplotlib.lines.Line2D at 0x93e6898>]
```

```
In [5]: plt.show()
```

Instead, if you're working on Jupyter QtConsole you see a different plot for each new command you enter.

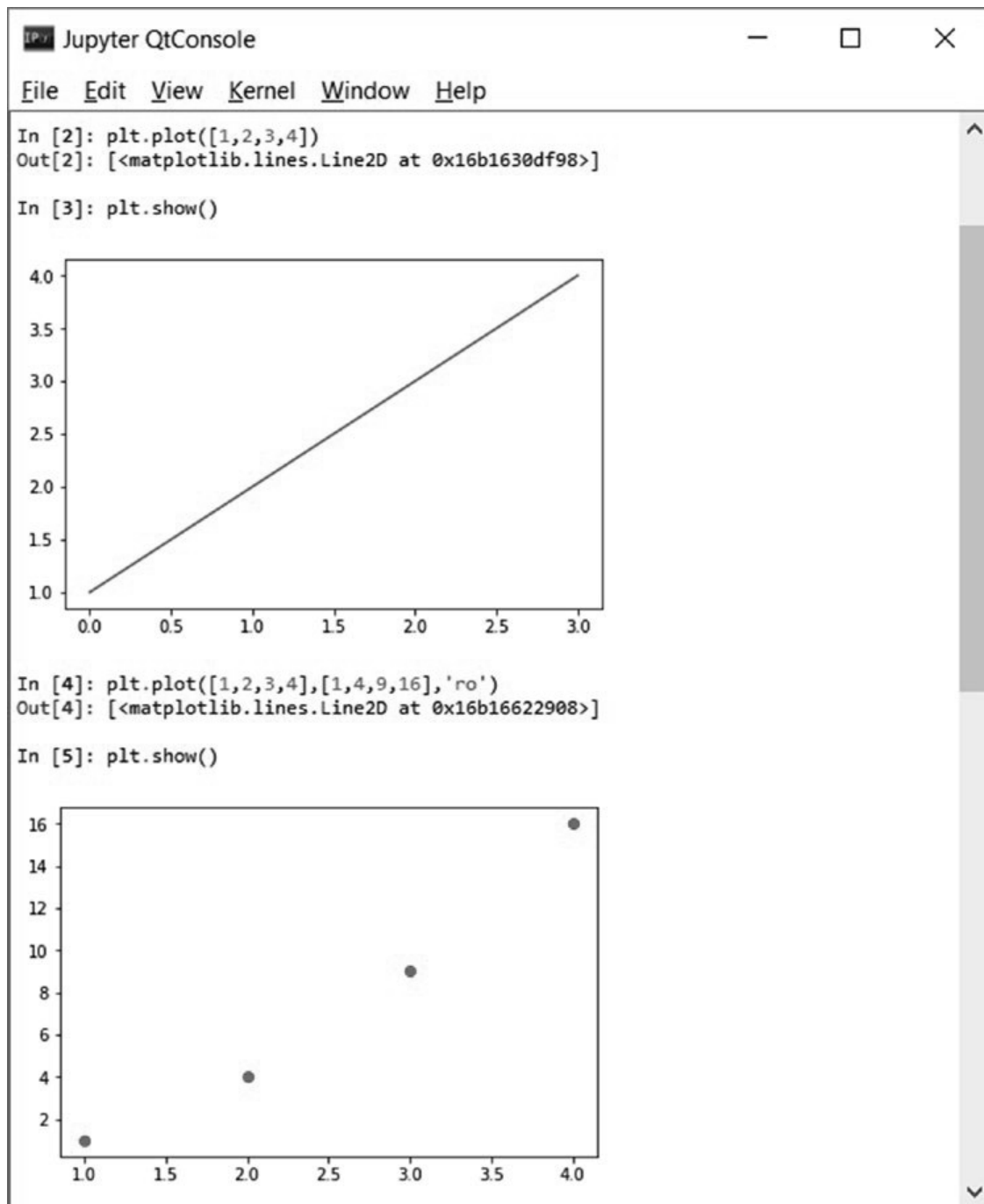


Figure 7-7. The pairs of (x,y) values are represented in the plot by red circles

Note At this point in the book, you already have a very clear idea about the difference between the various environments. To avoid confusion from this point, I will consider the IPython QtConsole as the sole development environment.

You can define the range both on the x-axis and on the y-axis by defining the details of a list `[xmin, xmax, ymin, ymax]` and then passing it as an argument to the `axis()` function.

Note In the IPython QtConsole, to generate a chart it is sometimes necessary to enter more rows of commands. To avoid generating a chart every time you press Enter (start a new line) along with losing the setting previously specified, you have to press Ctrl+Enter. When you want to finally generate the chart, just press Enter twice.

You can set several properties, one of which is the title that can be entered using the `title()` function.

```
In [4]: plt.axis([0,5,0,20])
...: plt.title('My first plot')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x97f1c18>]
```

In Figure 7-8 you can see how the new settings made the plot more readable. In fact, the end points of the dataset are now represented within the plot rather than at the edges. Also the title of the plot is now visible at the top.

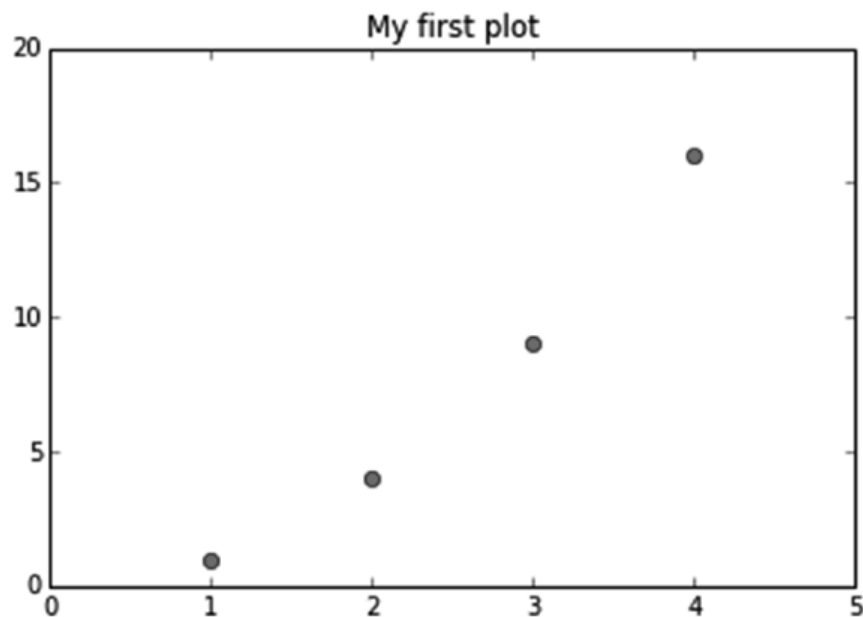


Figure 7-8. The plot after the properties have been set

matplotlib and NumPy

Even the matplotlib library, despite being a fully graphical library, has its foundation as the NumPy library. In fact, you have seen so far how to pass lists as arguments, both to represent the data and to set the extremes of the axes. Actually, these lists have been converted internally in NumPy arrays.

Therefore, you can directly enter NumPy arrays as input data. This array of data, which have been processed by pandas, can be directly used with matplotlib without further processing.

As an example, you see how it is possible to plot three different trends in the same plot (see Figure 7-9). You can choose for this example the `sin()` function belonging to the `math` module. So you will need to import it. To generate points following a sinusoidal trend, you will use the NumPy library. Generate a series of points on the x-axis using the `arange()` function, while for the values on the y-axis you will use the `map()` function to apply the `sin()` function on all the items of the array (without using a `for` loop).

```
In [5]: import math
In [6]: import numpy as np
In [7]: t = np.arange(0,2.5,0.1)
...: y1 = np.sin(math.pi*t)
...: y2 = np.sin(math.pi*t+math.pi/2)
...: y3 = np.sin(math.pi*t-math.pi/2)
In [8]: plt.plot(t,y1,'b*',t,y2,'g^',t,y3,'ys')
Out[8]:
[<matplotlib.lines.Line2D at 0xcbd2e48>,
 <matplotlib.lines.Line2D at 0xcbe10b8>,
 <matplotlib.lines.Line2D at 0xcbe15c0>]
```

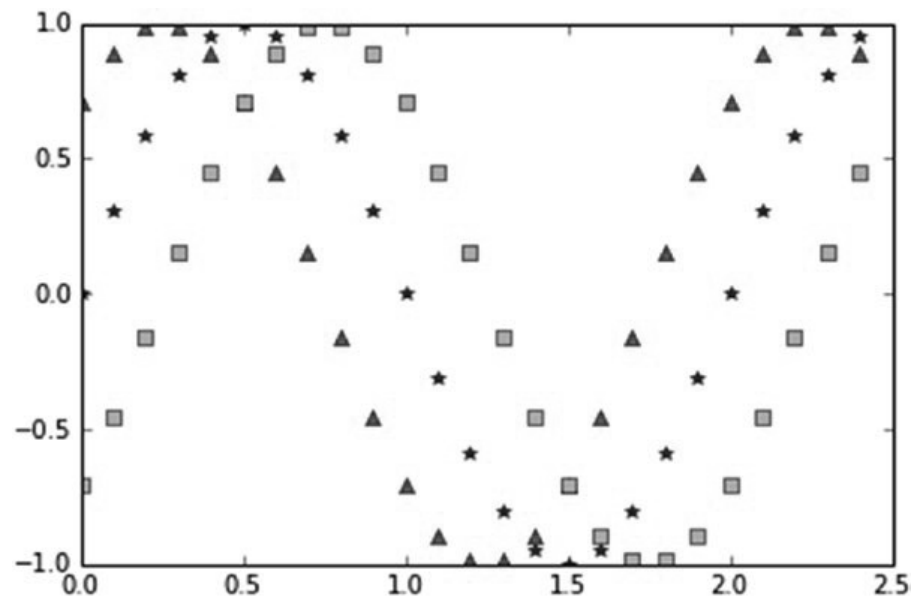



Figure 7-9. Three sinusoidal trends phase-shifted by $\pi / 4$ represented by markers

Note If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, insert the `plt.show()` command at the end of the code to obtain the chart shown in Figure 7-10.

As you can see in Figure 7-9, the plot represents the three different temporal trends with three different colors and markers. In these cases, when the trend of a function is so obvious, the plot is perhaps not the most appropriate representation, but it is better to use the lines (see Figure 7-10). To differentiate the three trends with something other than color, you can use the pattern composed of different combinations of dots and dashes (- and .).

```
In [9]: plt.plot(t,y1,'b--',t,y2,'g',t,y3,'r-.')
```

```
Out[9]:
```

```
[<matplotlib.lines.Line2D at 0xd1eb550>,
 <matplotlib.lines.Line2D at 0xd1eb780>,
 <matplotlib.lines.Line2D at 0xd1ebd68>]
```

Note If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, insert the `plt.show()` command at the end of the code to obtain the chart shown in Figure 7-10.

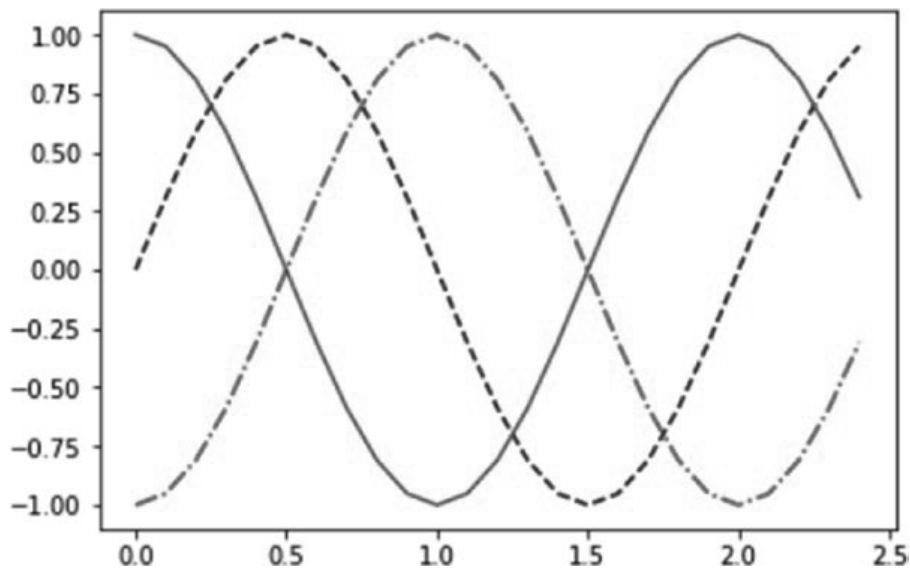


Figure 7-10. This chart represents the three sinusoidal patterns with colored lines

Using the kwargs

The objects that make up a chart have many attributes that characterize them. These attributes are all default values, but can be set through the use of *keyword args*, often referred as *kwargs*.

These keywords are passed as arguments to functions. In reference documentation of the various functions of the matplotlib library, you will always find them referred to as *kwargs* in the last position. For example the `plot()` function that you are using in these examples is referred to in the following way.

```
matplotlib.pyplot.plot(*args, **kwargs)
```

For a practical example, the thickness of a line can be changed if you set the `linewidth` keyword (see Figure 7-11).

```
In [10]: plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=2.0)
Out[10]: [<matplotlib.lines.Line2D at 0xc909da0>]
```

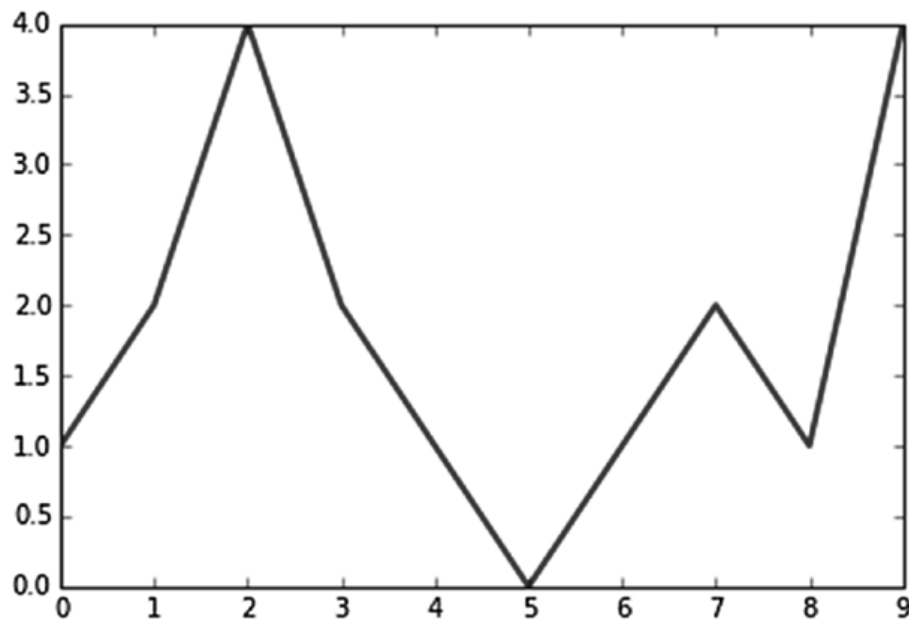


Figure 7-11. *The thickness of a line can be set directly from the `plot()` function*

Working with Multiple Figures and Axes

So far you have seen how all pyplot commands are routed to the display of a single figure. Actually, matplotlib allows you to manage multiple figures simultaneously, and within each figure, it offers the ability to view different plots defined as subplots.

So when you are working with pyplot, you must always keep in mind the concept of the current Figure and current Axes (that is, the plot shown within the figure).

Now you will see an example where two subplots are represented in a single figure. The `subplot()` function, in addition to subdividing the figure in different drawing areas, is used to focus the commands on a specific subplot.

The argument passed to the `subplot()` function sets the mode of subdivision and determines which is the current subplot. The current subplot will be the only figure that will be affected by the commands. The argument of the `subplot()` function is composed of three integers. The first number defines how many parts the figure is split into vertically. The second number defines how many parts the figure is divided into horizontally. The third issue selects which is the current subplot on which you can direct commands.

Now you will display two sinusoidal trends (sine and cosine) and the best way to do that is to divide the canvas vertically in two horizontal subplots (as shown in Figure 7-12). So the numbers to pass as an argument are 211 and 212.

```

In [11]: t = np.arange(0,5,0.1)
... : y1 = np.sin(2*np.pi*t)
... : y2 = np.sin(2*np.pi*t)
In [12]: plt.subplot(211)
...: plt.plot(t,y1,'b-.')
...: plt.subplot(212)
...: plt.plot(t,y2,'r--')
Out[12]: [<matplotlib.lines.Line2D at 0xd47f518>]

```

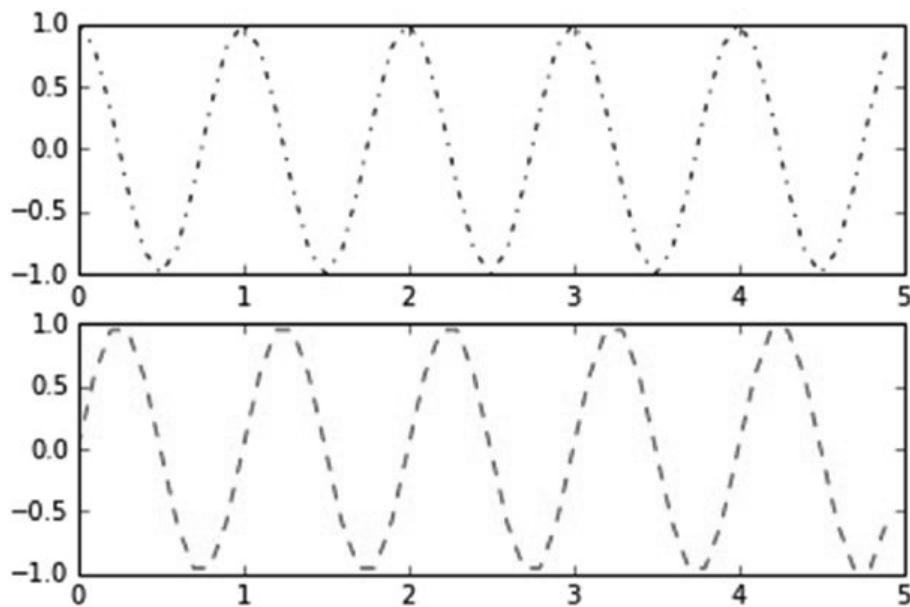


Figure 7-12. The figure has been divided into two horizontal subplots

Now you do the same thing by dividing the figure in two vertical subplots. The numbers to be passed as arguments to the `subplot()` function are 121 and 122 (as shown in Figure 7-13).

```

In [ ]: t = np.arange(0.,1.,0.05)
...: y1 = np.sin(2*np.pi*t)
...: y2 = np.cos(2*np.pi*t)
In [ ]: plt.subplot(121)
...: plt.plot(t,y1,'b-.')
...: plt.subplot(122)
...: plt.plot(t,y2,'r--')
Out[94]: [<matplotlib.lines.Line2D at 0xed0c208>]

```

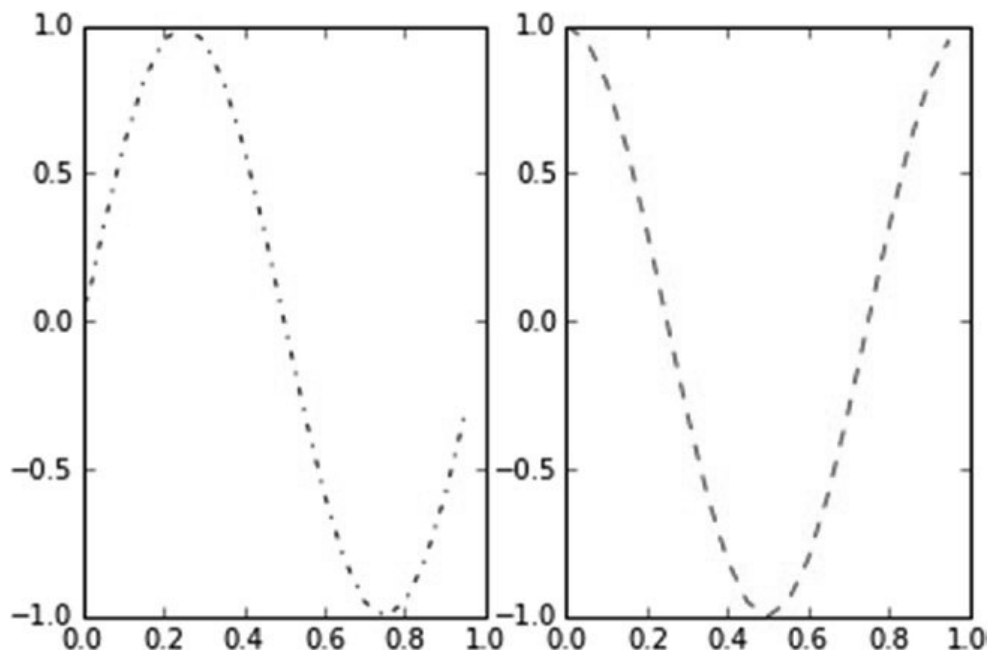


Figure 7-13. The figure has been divided into two vertical subplots

Adding Elements to the Chart

In order to make a chart more informative, many times it is not enough to represent the data using lines or markers and assign the range of values using two axes. In fact, there are many other elements that can be added to a chart in order to enrich it with additional information.

In this section you will see how to add elements to the chart as text labels, a legend, and so on.

Adding Text

You've already seen how you can add the title to a chart with the `title()` function. Two other textual indications you can add the *axis labels*. This is possible through the use of two other specific functions, called `xlabel()` and `ylabel()`. These functions take as an argument a string, which will be the shown text.

Note Command lines forming the code to represent your chart are growing in number. You do not need to rewrite all the commands each time, but using the arrow keys on the keyboard, you can call up the list of commands previously passed and edit them by adding new rows (in the text are indicated in bold).

Now add two axis labels to the chart. They will describe which kind of value is assigned to each axis (as shown in Figure 7-14).

```
In [10]: plt.axis([0,5,0,20])
...: plt.title('My first plot')
...: plt.xlabel('Counting')
...: plt.ylabel('Square values')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[10]: [<matplotlib.lines.Line2D at 0x990f3c8>]
```

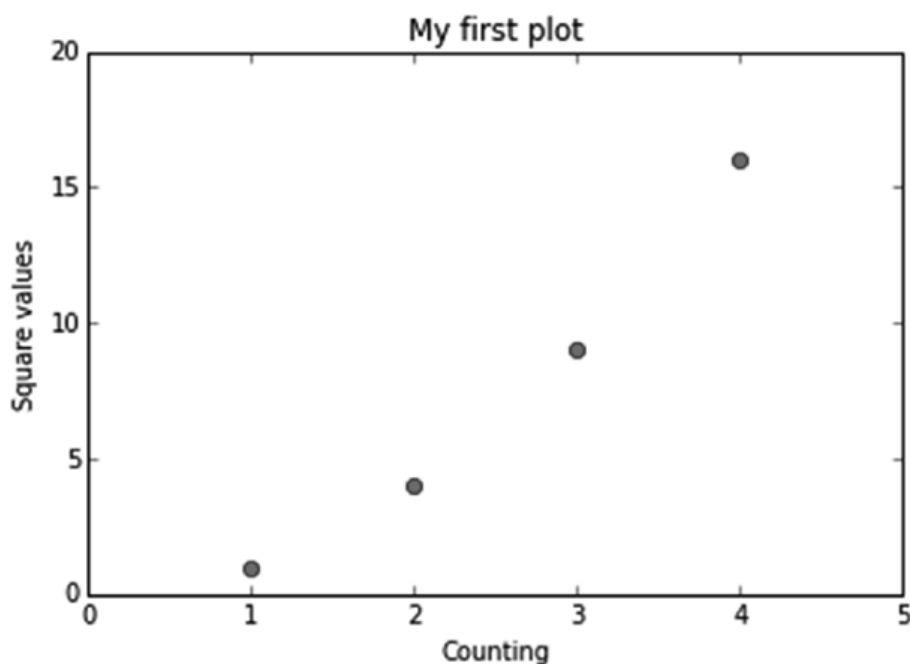


Figure 7-14. A plot is more informative by adding axis labels

Thanks to the keywords, you can change the characteristics of the text. For example, you can modify the title by changing the font and increasing the size of the characters. You can also modify the color of the axis labels to accentuate the title of the plot (as shown in Figure 7-15).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[116]: [<matplotlib.lines.Line2D at 0x11f17470>]
```

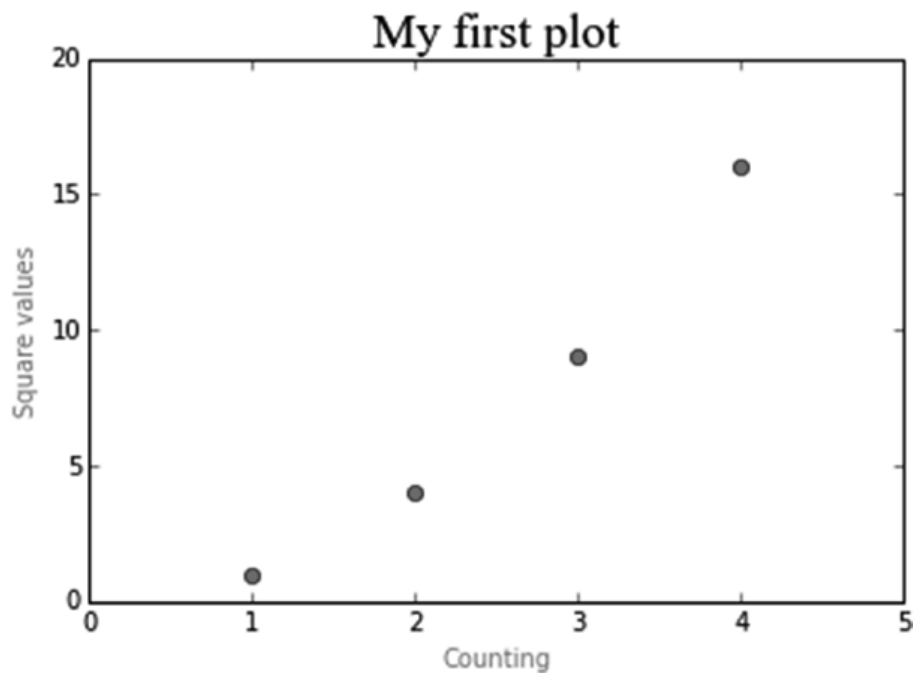


Figure 7-15. *The text can be modified by setting the keywords*

But matplotlib is not limited to this: pyplot allows you to add text to any position within a chart. This feature is performed by a specific function called `text()`.

```
text(x,y,s, fontdict=None, **kwargs)
```

The first two arguments are the coordinates of the location where you want to place the text. `s` is the string of text to be added, and `fontdict` (optional) is the font that you want to use. Finally, you can add the keywords.

Add the label to each point of the plot. Because the first two arguments to the `text()` function are the coordinates of the graph, you have to use the coordinates of the four points of the plot shifted slightly on the y-axis.

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [matplotlib.lines.Line2D at 0x10f76898]
```

As you can see in Figure 7-16, now each point of the plot has a label.

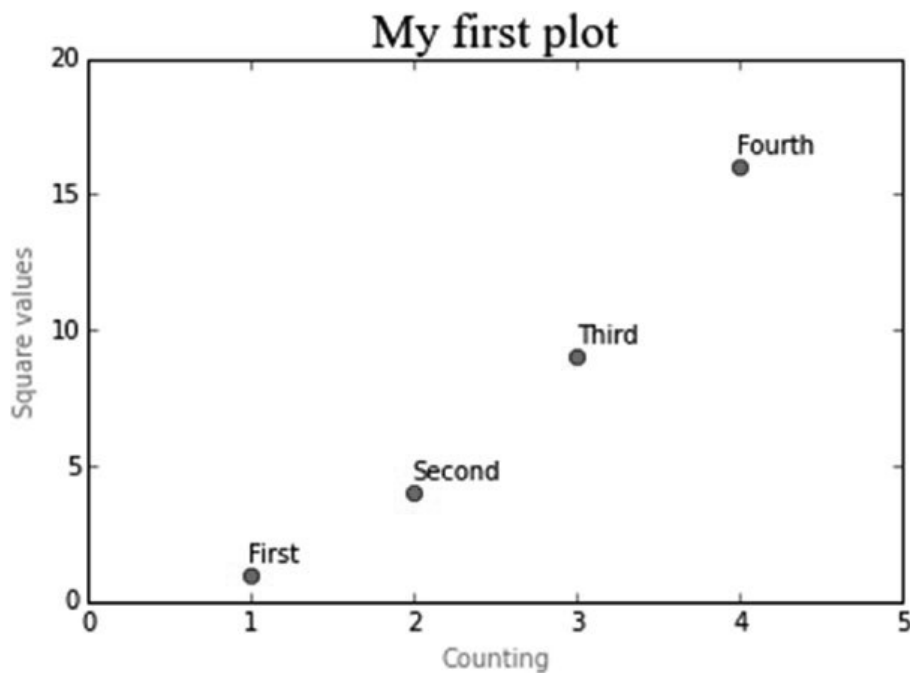


Figure 7-16. Every point of the plot has an informative label

Since matplotlib is a graphics library designed to be used in scientific circles, it must be able to exploit the full potential of scientific language, including mathematical expressions. matplotlib offers the possibility to integrate LaTeX expressions, thereby allowing you to insert mathematical expressions within the chart.

To do this, you can add a LaTeX expression to the text, enclosing it between two \$ characters. The interpreter will recognize them as LaTeX expressions and convert them to the corresponding graphic, which can be a mathematical expression, a formula, mathematical characters, or just Greek letters. Generally you have to precede the string containing LaTeX expressions with an `r`, which indicates raw text, in order to avoid unintended escape sequences.

Here, you can also use the keywords to further enrich the text to be shown in the plot. Therefore, as an example, you can add the formula describing the trend followed by the point of the plot and enclose it in a colored bounding box (see Figure 7-17).


```

In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')

```

```
Out[130]: [<matplotlib.lines.Line2D at 0x13920860>]
```

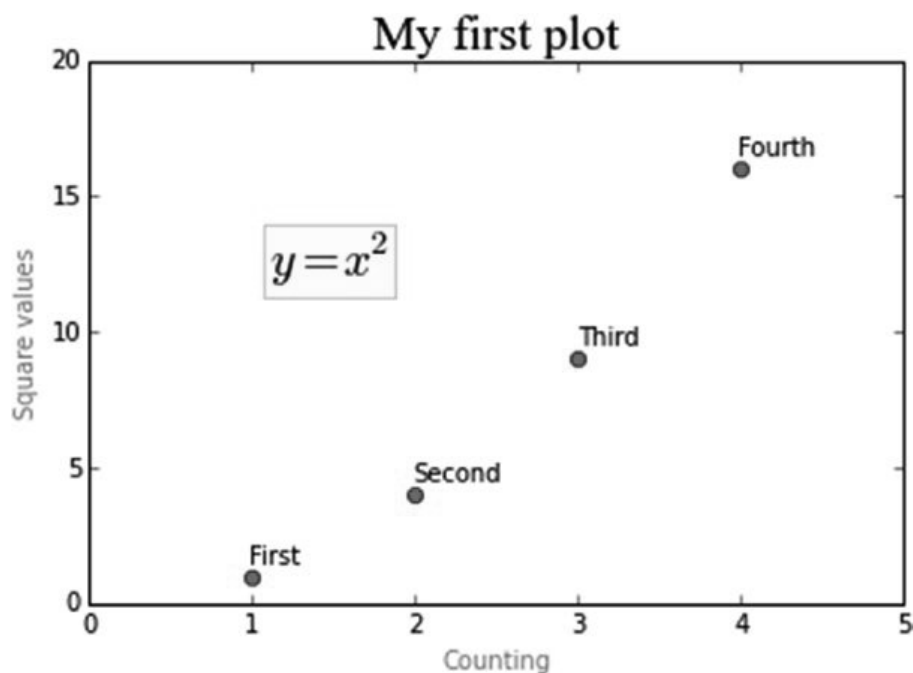


Figure 7-17. Any mathematical expression can be seen in the context of a chart

To get a complete view on the potential offered by LaTeX, consult Appendix A of this book.

Adding a Grid

Another element you can add to a plot is a grid. Often its addition is necessary in order to better understand the position occupied by each point on the chart.

Adding a grid to a chart is a very simple operation: just add the `grid()` function, passing `True` as an argument (see Figure 7-18).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

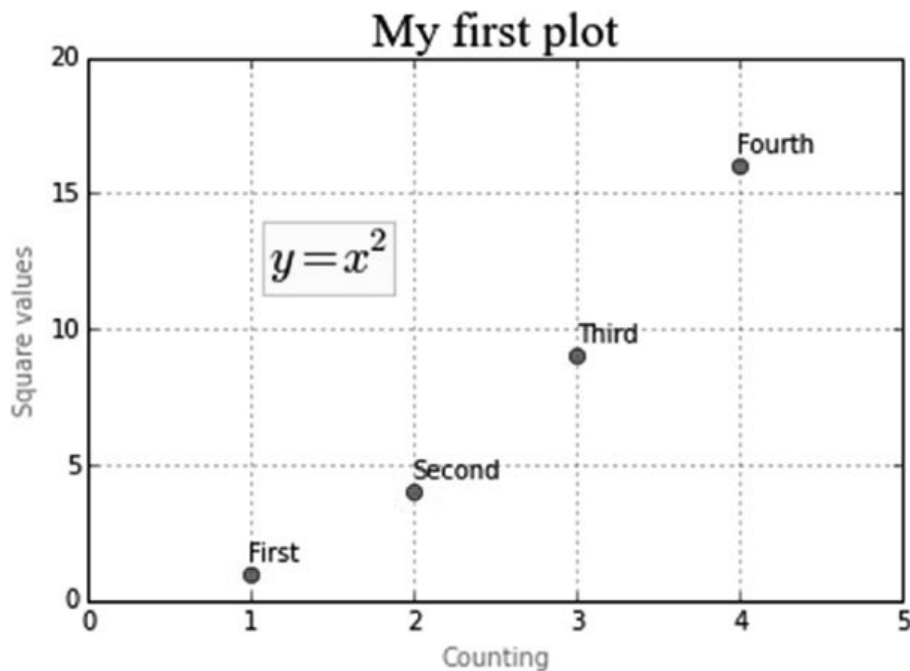


Figure 7-18. A grid makes it easier to read the values of the data points represented on a chart

Adding a Legend

Another very important component that should be present in any chart is the legend. pyplot also provides a specific function for this type of object: `legend()`.

Add a legend to your chart with the `legend()` function and a string indicating the words with which you want the series to be shown. In this example, you assign the First series name to the input data array (see Figure 7-19).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.legend(['First series'])
```

Out[156]: <matplotlib.legend.Legend at 0x16377550>

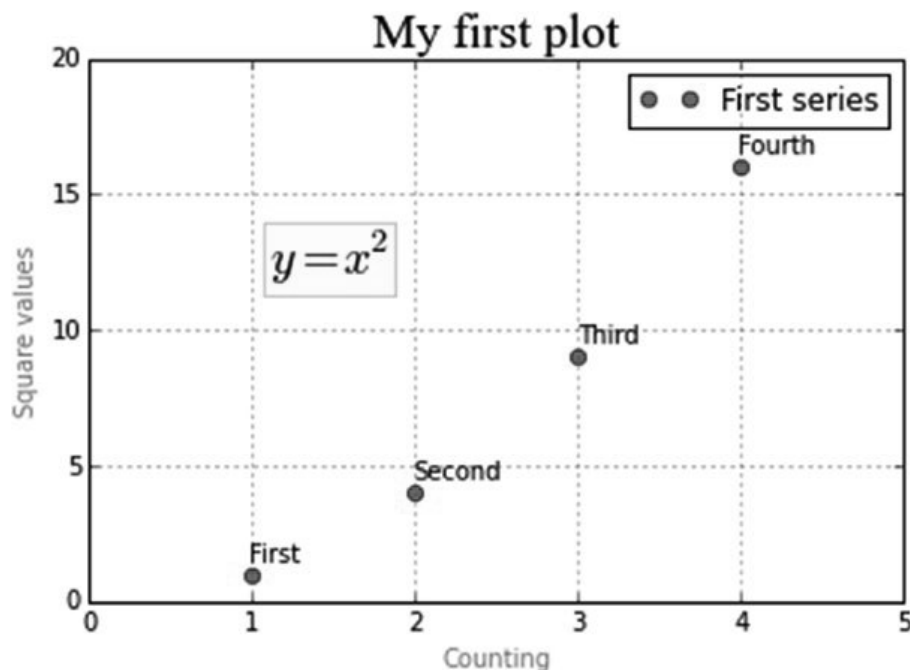


Figure 7-19. A legend is added in the upper-right corner by default

As you can see in Figure 7-19, the legend is added in the upper-right corner by default. Again if you want to change this behavior you will need to add a few kwargs. For example, the position occupied by the legend is set by assigning numbers from 0 to 10 to the `loc` kwarg. Each of these numbers characterizes one of the corners of the chart (see Table 7-1). A value of 1 is the default, that is, the upper-right corner. In the next example, you will move the legend in the upper-left corner so it will not overlap with the points represented in the plot.

Table 7-1. *The Possible Values for the `loc` Keyword*

Location Code	Location String
0	best
1	upper-right
2	upper-left
3	lower-right
4	lower-left
5	right
6	center-left
7	center-right
8	lower-center
9	upper-center
10	center

Before you begin to modify the code to move the legend, I want to add a small notice. Generally, the legends are used to indicate the definition of a series to the reader via a label associated with a color and/or a marker that distinguishes it in the plot. So far in the examples, you have used a single series that was expressed by a single `plot()` function. Now, you have to focus on a more general case in which the same plot shows more series simultaneously. Each series in the chart will be characterized by a specific color and a specific marker (see Figure 7-20). In terms of code, instead, each series will be characterized by a call to the `plot()` function and the order in which they are defined will correspond to the order of the text labels passed as an argument to the `legend()` function.

```

In [ ]: import matplotlib.pyplot as plt
...: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
...: 'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
...: plt.legend(['First series','Second series','Third series'],loc=2)
Out[170]: <matplotlib.legend.Legend at 0x1828d7b8>

```

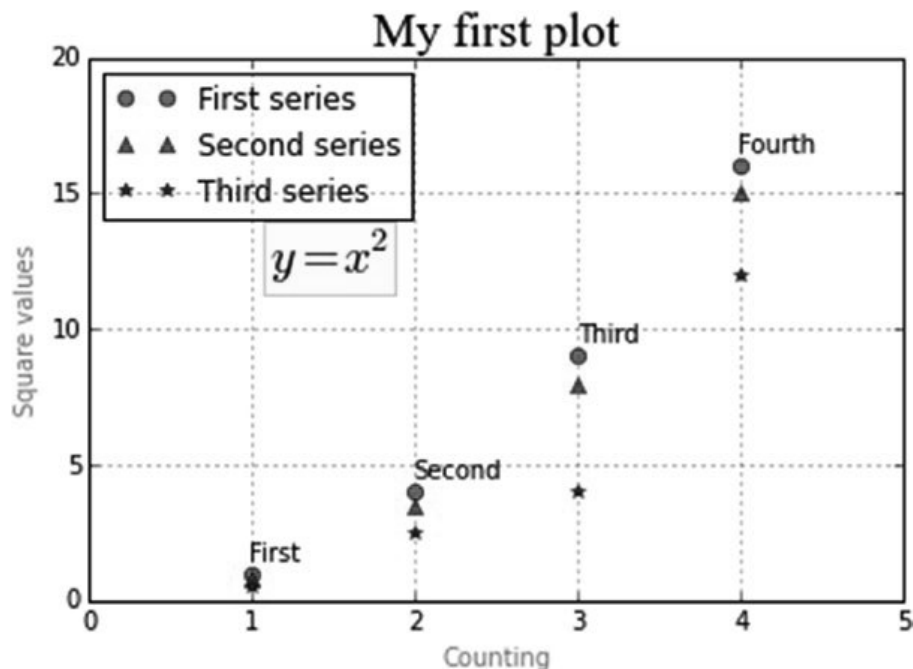


Figure 7-20. A legend is necessary in every multiseries chart

Saving Your Charts

In this section you will learn how to save your chart in different ways depending on your needs. If you need to reproduce your chart in different notebooks or Python sessions, or reuse them in future projects, it is a good practice to save the Python code. On the other hand, if you need to make reports or presentations, it can be very useful to save your chart as an image. Moreover, it is possible to save your chart as a HTML page, and this could be very useful when you need to share your work on Web.

Saving the Code

As you can see from the examples in the previous sections, the code concerning the representation of a single chart is growing into a fair number of rows. Once you think you've reached a good point in your development process, you can choose to save all rows of code in a `.py` file that you can recall at any time.

You can use the magic command `save%` followed by the name of the file you want to save followed by the number of input prompts containing the row of code that you want to save. If all the code is written in only one prompt, as your case, you have to add only its number; otherwise if you want to save the code written in many prompts, for example from 10 to 20, you have to indicate this range with the two numbers separated by a `-`, that is, `10-20`.

In your case, you would save the Python code underlying the representation of your first chart contained into the input prompt with the number 171.

```
In [171]: import matplotlib.pyplot as plt
...
```

You need to insert the following command to save the code into a new `.py` file.

```
%save my_first_chart 171
```

After you launch the command, you will find the `my_first_chart.py` file in your working directory (see Listing 7-1).

Listing 7-1. my_first_chart.py

```
# coding: utf-8
import matplotlib.pyplot as plt
plt.axis([0,5,0,20])
plt.title('My first plot',fontsize=20,fontname='Times New Roman')
plt.xlabel('Counting',color='gray')
plt.ylabel('Square values',color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'Second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
'alpha':0.2})
plt.grid(True)
plt.plot([1,2,3,4],[1,4,9,16],'ro')
plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
plt.legend(['First series','Second series','Third series'],loc=2)
```

Later, when you open a new IPython session, you will have your chart and start to change the code at the point where you had saved it by entering the following command:

```
ipython qtconsole --matplotlib inline -m my_first_chart.py
```

Or you can reload the entire code in a single prompt in the QtConsole using the magic command `%load`.

```
%load my_first_chart.py
```

Or you can run it during a session with the magic command `%run`.

```
%run my_first_chart.py
```

Note On my system, this command works only after launching the two previous commands.

Converting Your Session to an HTML File

Using the IPython QtConsole, you can convert all the code and graphics present in your current session to an HTML page. Simply choose *File-->Save to HTML/XHTML* from the menu (as shown in Figure 7-21).

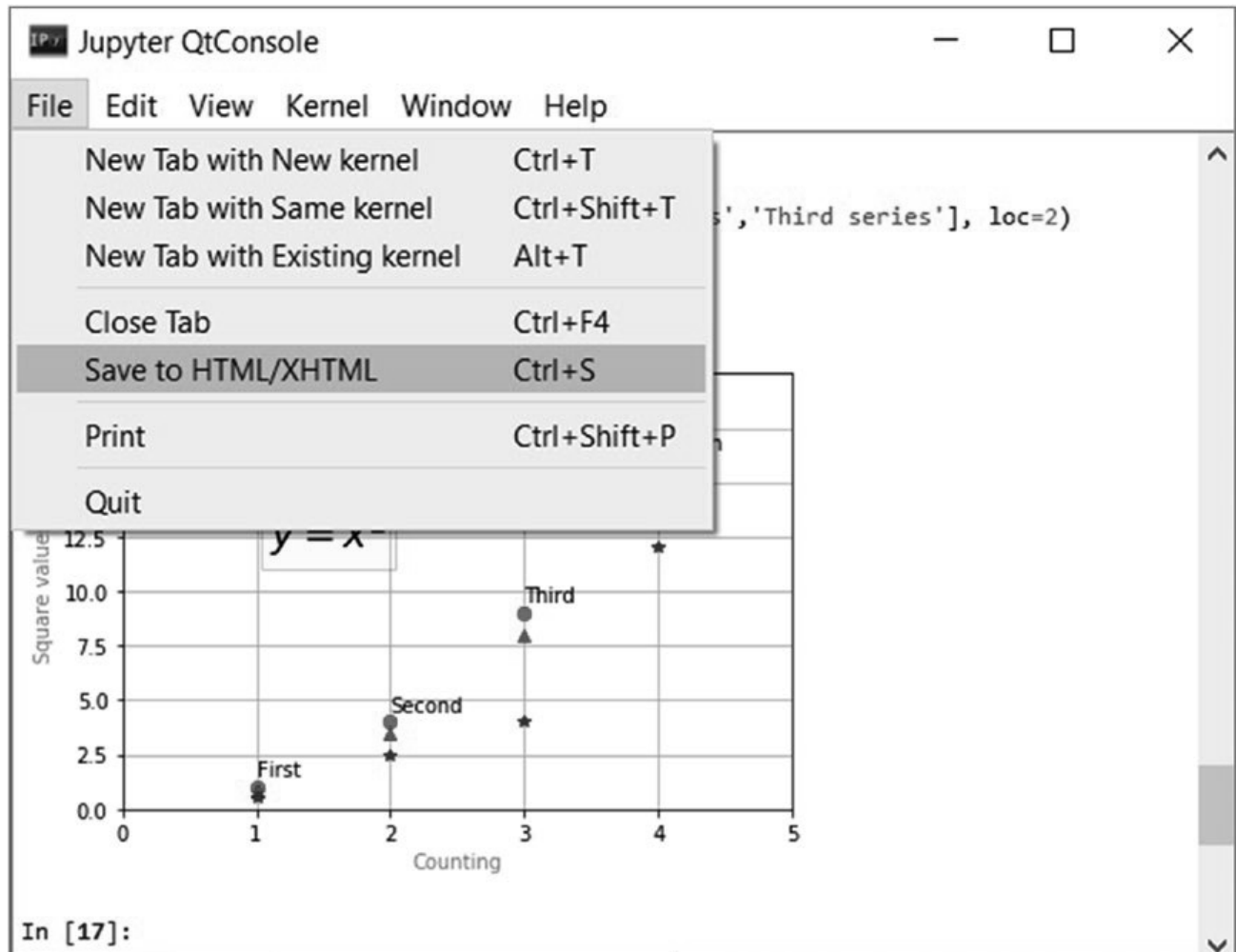


Figure 7-21. You can save your current session as a web page

You will be asked to save your session in two different formats: HTML and XHTML. The difference between the two formats is based on the image conversion type. If you select HTML as the output file format, the images contained in your session will be converted to PNG format. If you select XHTML as the output file format instead, the images will be converted to SVG format.

In this example, save your session as an HTML file and name it `my_session.html`, as shown in Figure 7-22.

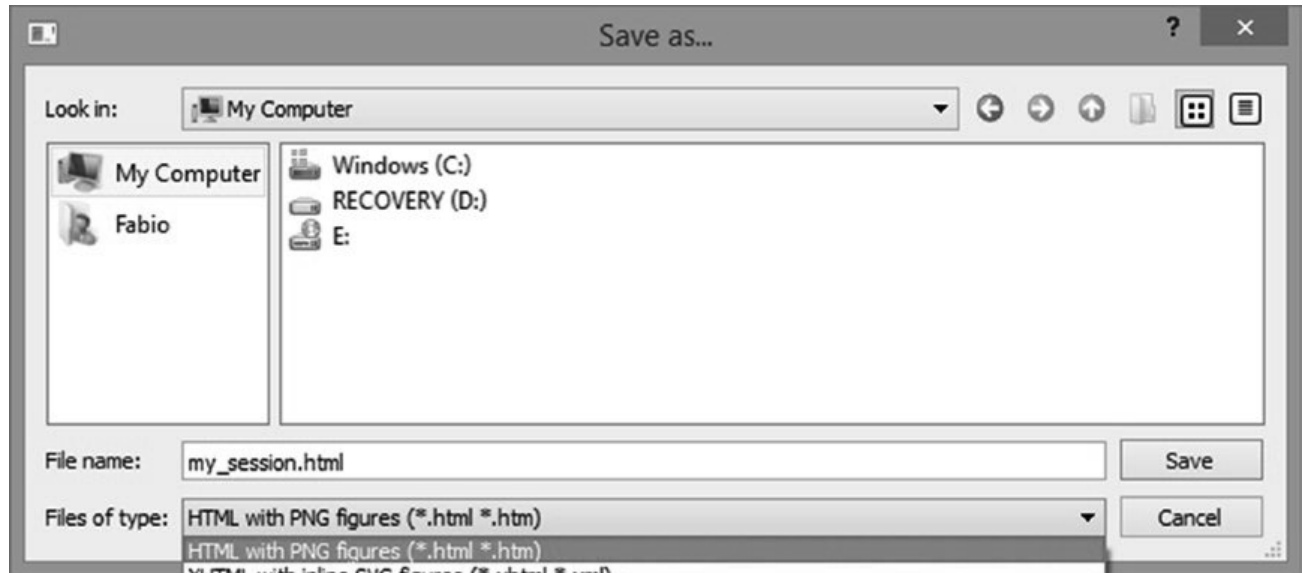


Figure 7-22. You can select the type of file between HTML and XHTML

At this point, you will be asked if you want to save your images in an external directory or inline (see Figure 7-23).

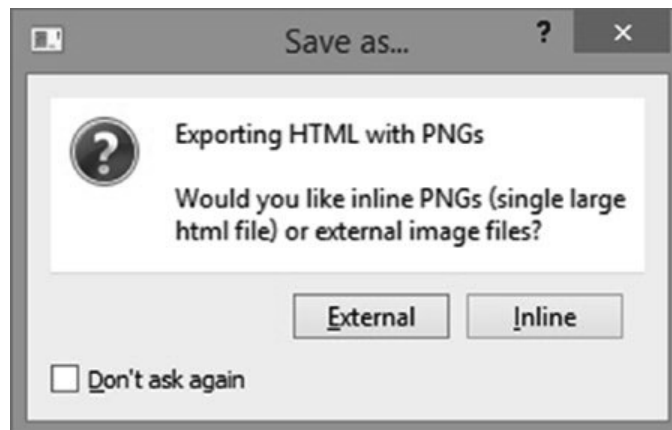


Figure 7-23. You can choose between creating external image files and embedding the PNG format directly into the HTML page

By choosing the external option, the images will be collected into a directory called `my_session_files`. By choosing inline, the graphical information concerning the image is embedded into the HTML code.

Saving Your Chart Directly as an Image

If you are interested in saving only the figure of a chart as an image file, ignoring all the code you've written during the session, this is also possible. In fact, thanks to the `savefig()` function, you can directly save the chart in a PNG format, although you should take care to add this function to the end of the same series of commands (otherwise you'll get a blank PNG file).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
...: plt.legend(['First series','Second series','Third series'],loc=2)
...: plt.savefig('my_chart.png')
```

Executing the previous code, a new file will be created in your working directory. This file will be named `my_chart.png` and will contain the image of your chart.

Handling Date Values

One of the most common problems encountered when doing data analysis is handling data of the date-time type. Displaying that data along an axis (normally the x-axis) can be problematic, especially when managing ticks (see Figure 7-24).

Take for example the display of a linear chart with a dataset of eight points in which you have to represent date values on the x-axis with the following format: day-month-year.

```
In [ ]: import datetime
...: import numpy as np
...: import matplotlib.pyplot as plt
...: events = [datetime.date(2015,1,23),datetime.
...:           date(2015,1,28),datetime.date(2015,2,3),datetime.
...:           date(2015,2,21),datetime.date(2015,3,15),datetime.
...:           date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
...: readings = [12,22,25,20,18,15,17,14]
...: plt.plot(events,readings)
Out[83]: [<matplotlib.lines.Line2D at 0x12666400>]
```

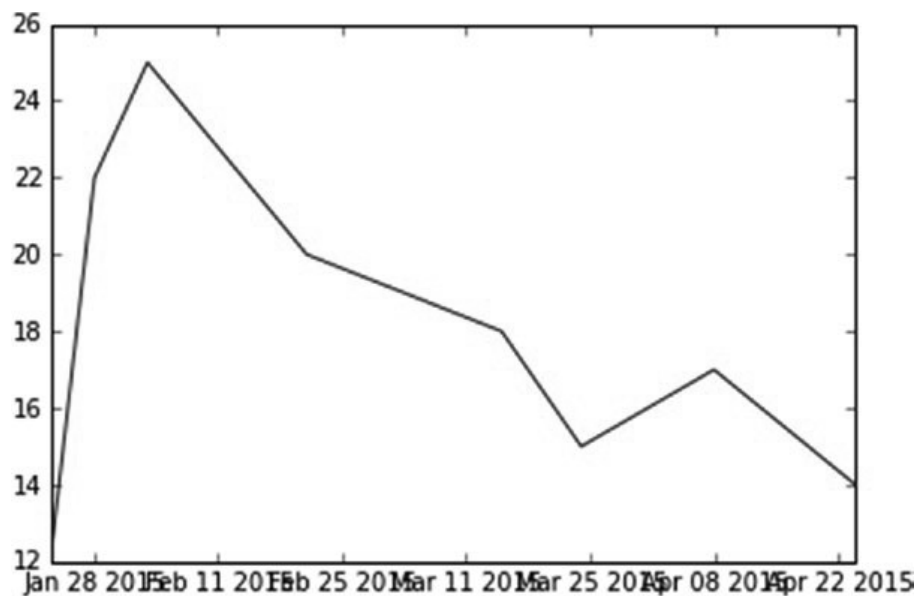


Figure 7-24. *If not handled, displaying date-time values can be problematic*

As you can see in Figure 7-24, automatic management of ticks, and especially the tick labels, can be a disaster. The dates expressed in this way are difficult to read, there are no clear time intervals elapsed between one point and another, and there is also overlap.

To manage dates it is therefore advisable to define a time scale with appropriate objects. First you need to import `matplotlib.dates`, a module specialized for this type of data. Then you define the scales of the times, as in this case, a scale of days and one of the months, through the `MonthLocator()` and `DayLocator()` functions. In these cases, the formatting is also very important, and to avoid overlap or unnecessary references, you have to limit the tick labels to the essential, which in this case is year-month. This format can be passed as an argument to the `DateFormatter()` function.

After you defined the two scales, one for the days and one for the months, you can set two different kinds of ticks on the x-axis, using the `set_major_locator()` and `set_minor_locator()` functions on the `xaxis` object. Instead, to set the text format of the tick labels referred to the months you have to use the `set_major_formatter()` function.

Changing all these settings you finally obtain the plot as shown in Figure 7-25.

```
In [ ]: import datetime
...: import numpy as np
...: import matplotlib.pyplot as plt
...: import matplotlib.dates as mdates
...: months = mdates.MonthLocator()
...: days = mdates.DayLocator()
...: timeFmt = mdates.DateFormatter('%Y-%m')
...: events = [datetime.date(2015,1,23),datetime.
               date(2015,1,28),datetime.date(2015,2,3),datetime.
               date(2015,2,21),datetime.date(2015,3,15),datetime.
               date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
readings = [12,22,25,20,18,15,17,14]
...: fig, ax = plt.subplots()
...: plt.plot(events,readings)
...: ax.xaxis.set_major_locator(months)
...: ax.xaxis.set_major_formatter(timeFmt)
...: ax.xaxis.set_minor_locator(days)
```

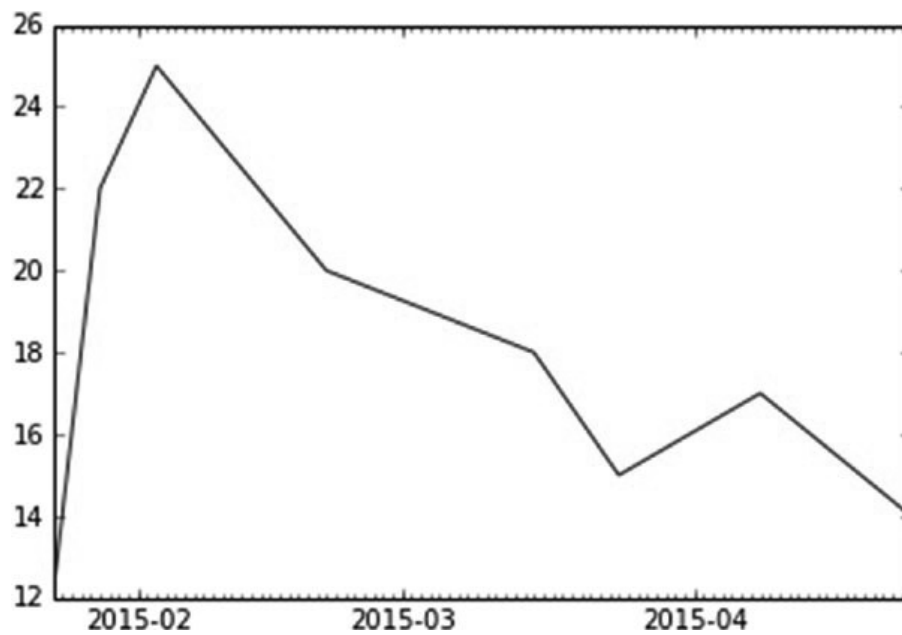


Figure 7-25. Now the tick labels of the x-axis refer only to the months, making the plot more readable

Chart Typology

In the previous sections you saw a number of examples relating to the architecture of the matplotlib library. Now that you are familiar with the use of the main graphic elements in a chart, it is time to see a series of examples treating different types of charts, starting from the most common ones such as linear charts, bar charts, and pie charts, up to a discussion about some that are more sophisticated but commonly used nonetheless.

This part of the chapter is very important since the purpose of this library is the visualization of the results produced by data analysis. Thus, knowing how to choose the proper type of chart is a fundamental choice. Remember that excellent data analysis represented incorrectly can lead to a wrong interpretation of the experimental results.

Line Charts

Among all the chart types, the linear chart is the simplest. A line chart is a sequence of data points connected by a line. Each data point consists of a pair of values (x,y), which will be reported in the chart according to the scale of values of the two axes (x and y).

By way of example, you can begin to plot the points generated by a mathematical function. Then, you can consider a generic mathematical function such as this:

$$y = \sin(3 * x) / x$$

Therefore, if you want to create a sequence of data points, you need to create two NumPy arrays. First you create an array containing the x values to be referred to the x-axis. In order to define a sequence of increasing values you will use the `np.arange()` function. Since the function is sinusoidal you should refer to values that are multiples and submultiples of the Greek pi (`np.pi`). Then, using these sequence of values, you can obtain the y values applying the `np.sin()` function directly to these values (thanks to NumPy!).

After all this, you have only to plot them by calling the `plot()` function. You will obtain a line chart, as shown in Figure 7-26.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: plt.plot(x,y)
Out[393]: [<matplotlib.lines.Line2D at 0x22404358>]
```

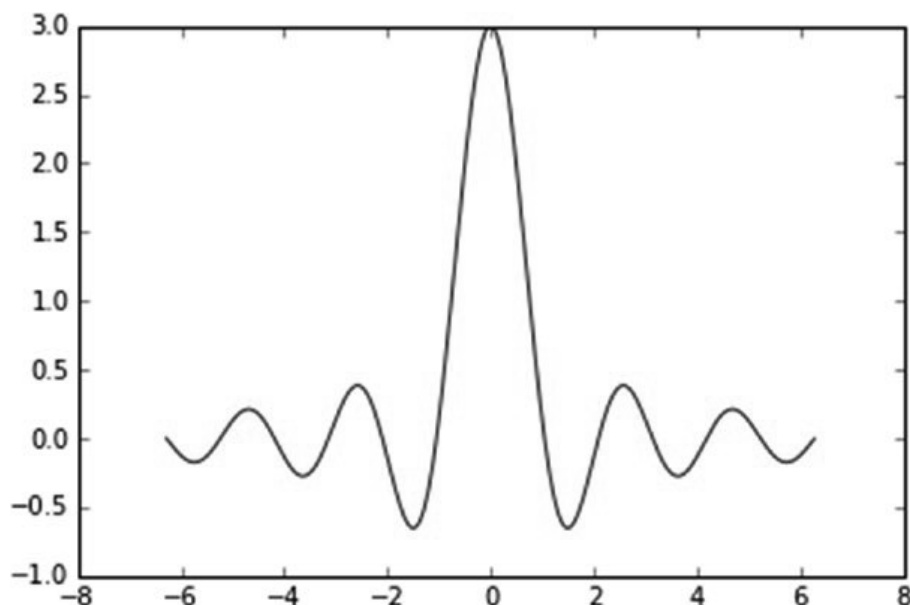


Figure 7-26. A mathematical function represented in a line chart

Now you can extend the case in which you want to display a family of functions, such as this:

$$y = \sin(n * x) / x$$

varying the parameter n .

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(3*x)/x
...: plt.plot(x,y)
...: plt.plot(x,y2)
...: plt.plot(x,y3)
```

As you can see in Figure 7-27, a different color is automatically assigned to each line. All the plots are represented on the same scale; that is, the data points of each series refer to the same x-axis and y-axis. This is because each call of the `plot()` function takes into account the previous calls to same function, so the Figure applies the changes keeping memory of the previous commands until the Figure is not displayed (using `show()` with Python and Enter with the IPython QtConsole).

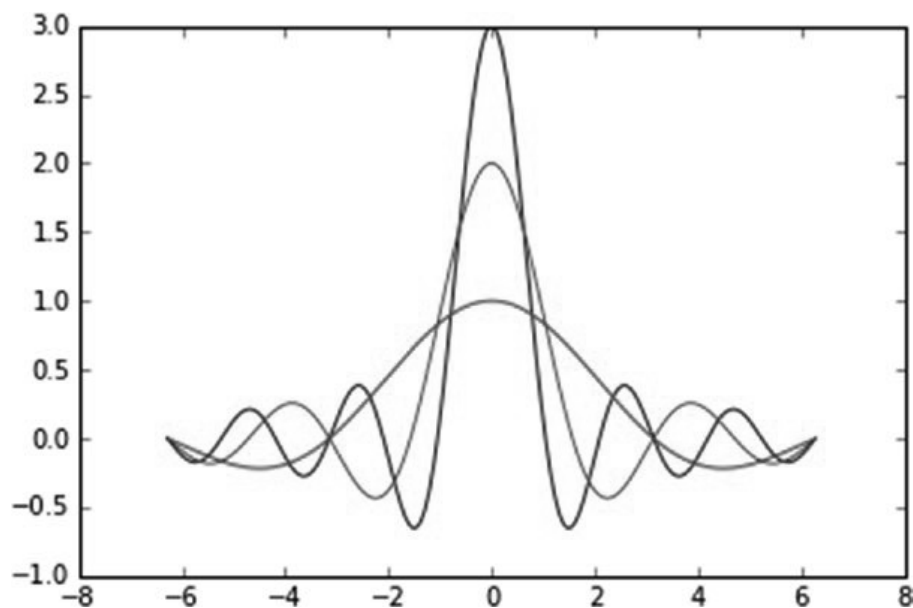


Figure 7-27. Three different series are drawn with different colors in the same chart

As you saw in the previous sections, regardless of the default settings, you can select the type of stroke, color, etc. As the third argument of the `plot()` function you can specify some codes that correspond to the color (see Table 7-2) and other codes that correspond to line styles, all included in the same string. Another possibility is to use two kwargs separately, `color` to define the color, and `linestyle` to define the stroke (see Figure 7-28).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(3*x)/x
...: plt.plot(x,y,'k--',linewidth=3)
...: plt.plot(x,y2,'m-.')
...: plt.plot(x,y3,color='#87a3cc',linestyle='--')
```

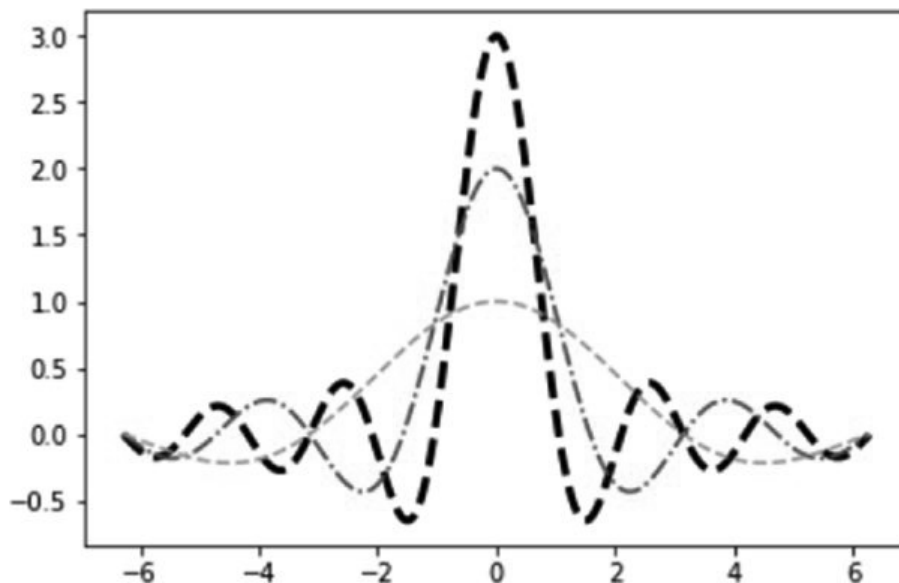


Figure 7-28. You can define colors and line styles using character codes

Table 7-2. *Color Codes*

Code	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

You have just defined a range from -2π to 2π on the x-axis, but by default, values on ticks are shown in numerical form. Therefore you need to replace the numerical values with multiple of π . You can also replace the ticks on the y-axis. To do all this, you have to use `xticks()` and `yticks()` functions, passing to each of them two lists of values. The first list contains values corresponding to the positions where the ticks are to be placed, and the second contains the tick labels. In this particular case, you have to use strings containing LaTeX format in order to correctly display the symbol π . Remember to define them within two `$` characters and to add a `r` as the prefix.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi, 2*np.pi, 0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(x)/x
...: plt.plot(x, y, color='b')
...: plt.plot(x, y2, color='r')
...: plt.plot(x, y3, color='g')
...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
...:             [r'$-2\pi$', r'$-\pi$', r'$0$', r'$+\pi$', r'$+2\pi$'])
```

```
...: plt.yticks([-1,0,1,2,3],
               [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
```

Out[423]:

```
([<matplotlib.axis.YTick at 0x26877ac8>,
  <matplotlib.axis.YTick at 0x271d26d8>,
  <matplotlib.axis.YTick at 0x273c7f98>,
  <matplotlib.axis.YTick at 0x273cc470>,
  <matplotlib.axis.YTick at 0x273cc9e8>],
 <a list of 5 Text yticklabel objects>)
```

In the end, you will get a clean and pleasant line chart showing Greek characters, as in Figure 7-29.

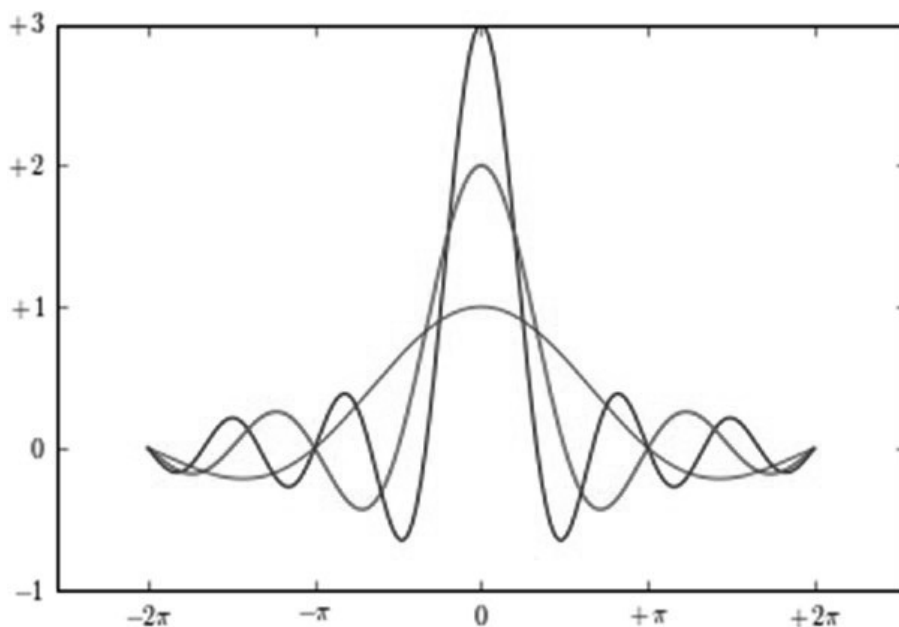


Figure 7-29. The tick label can be improved adding text with LaTeX format

In all the linear charts you have seen so far, you always have the x-axis and y-axis placed at the edge of the figure (corresponding to the sides of the bounding border box). Another way of displaying axes is to have the two axes passing through the origin (0, 0), i.e., the two Cartesian axes.

To do this, you must first capture the Axes object through the `gca()` function. Then through this object, you can select each of the four sides making up the bounding box, specifying for each one its position: right, left, bottom, and top. Crop the sides that do not match any axis (right and bottom) using the `set_color()` function and indicating none for color. Then, the sides corresponding to the x- and y-axes are moved to pass through the origin (0,0) with the `set_position()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi, 2*np.pi, 0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(x)/x
...: plt.plot(x, y, color='b')
...: plt.plot(x, y2, color='r')
...: plt.plot(x, y3, color='g')
...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
...:             [r'$-2\pi$', r'$-\pi$', r'$0$', r'$+\pi$', r'$+2\pi$'])
...: plt.yticks([-1, 0, +1, +2, +3],
...:             [r'$-1$', r'$0$', r'$+1$', r'$+2$', r'$+3$'])
...: ax = plt.gca()
...: ax.spines['right'].set_color('none')
...: ax.spines['top'].set_color('none')
...: ax.xaxis.set_ticks_position('bottom')
...: ax.spines['bottom'].set_position(('data', 0))
...: ax.yaxis.set_ticks_position('left')
...: ax.spines['left'].set_position(('data', 0))
```

Now the chart will show the two axes crossing in the middle of the figure, that is, the origin of the Cartesian axes, as shown in Figure 7-30.

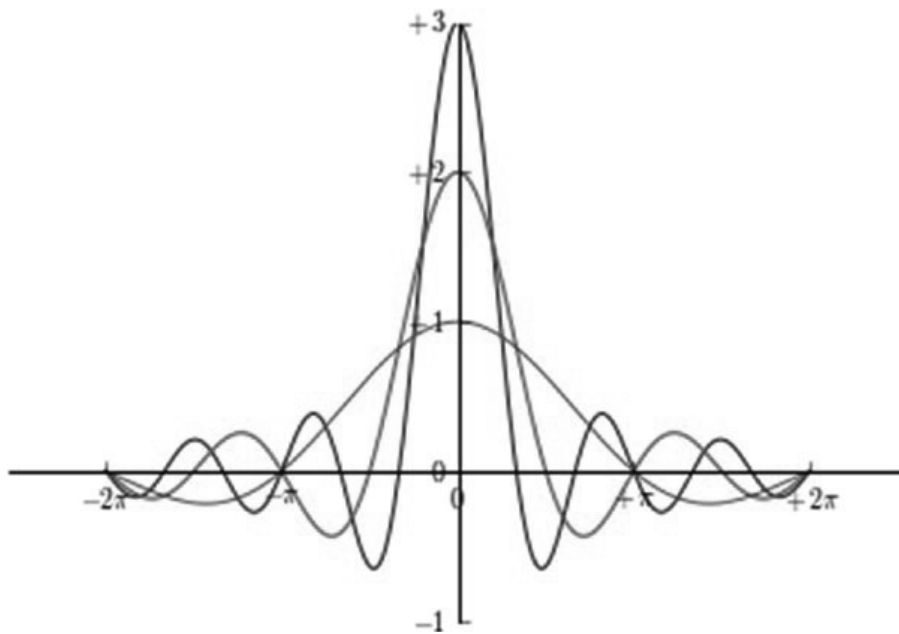


Figure 7-30. The chart shows two Cartesian axes

Often, it is very useful to be able to specify a particular point of the line using a notation and optionally add an arrow to better indicate the position of the point. For example, this notation may be a LaTeX expression, such as the formula for the limit of the function $\sin x/x$ with x tends to 0.

In this regard, matplotlib provides a function called `annotate()`, which is especially useful in these cases, even if the numerous kwargs needed to obtain a good result can make its settings quite complex. The first argument is the string to be represented containing the expression in LaTeX; then you can add the various kwargs. The point of the chart to note is indicated by a list containing the coordinates of the point $[x, y]$ passed to the `xy` kwarg. The distance of the textual notation from the point to be highlighted is defined by the `xytext` kwarg and represented by means of a curved arrow whose characteristics are defined in the `arrowprops` kwarg.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi, 2*np.pi, 0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(x)/x
...: plt.plot(x, y, color='b')
...: plt.plot(x, y2, color='r')
```

```

...: plt.plot(x,y3,color='g')
...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
    [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
...: plt.yticks([-1,0,+1,+2,+3],
    [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
...: plt.annotate(r'$\lim_{x\to 0}\frac{\sin(x)}{x}= 1$', xy=[0,1],
    xycoords='data',xytext=[30,30],fontsize=16,textcoords='offset points',
    arrowprops=dict(arrowstyle="->",connectionstyle="arc3,rad=.2"))
...: ax = plt.gca()
...: ax.spines['right'].set_color('none')
...: ax.spines['top'].set_color('none')
...: ax.xaxis.set_ticks_position('bottom')
...: ax.spines['bottom'].set_position(('data',0))
...: ax.yaxis.set_ticks_position('left')
...: ax.spines['left'].set_position(('data',0))

```

Running this code, you will get the chart with the mathematical notation of the limit, which is the point shown by the arrow in Figure 7-31.

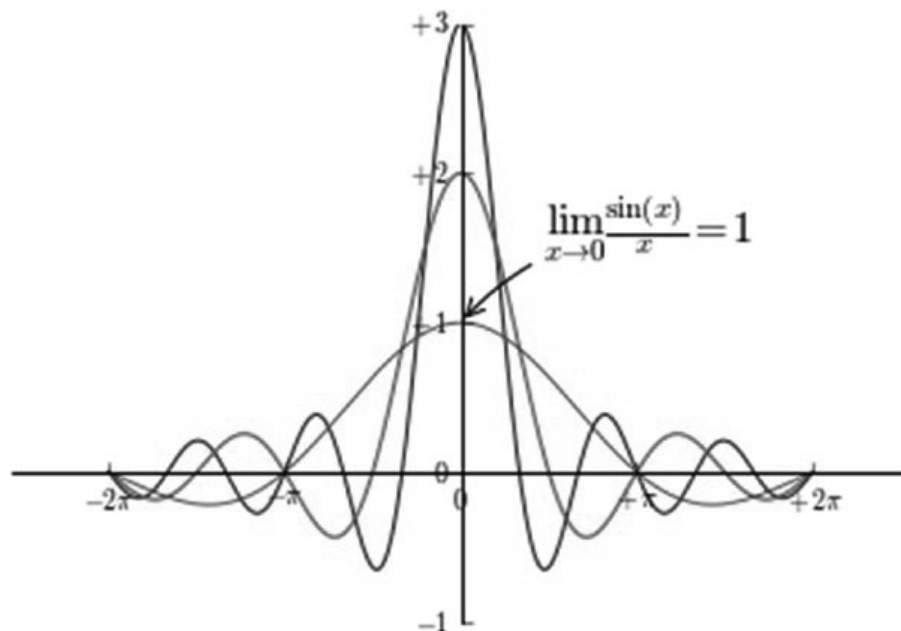


Figure 7-31. Mathematical expressions can be added to a chart with the `annotate()` function

Line Charts with pandas

Moving to more practical cases, or at least more closely related to data analysis, now is the time to see how easy it is to apply the matplotlib library to the dataframes of the pandas library. The visualization of the data in a dataframe as a linear chart is a very simple operation. It is sufficient to pass the dataframe as an argument to the `plot()` function to obtain a multiseries linear chart (see Figure 7-32).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: x = np.arange(5)
...: plt.axis([0,5,0,7])
...: plt.plot(x,df)
...: plt.legend(data, loc=2)
```

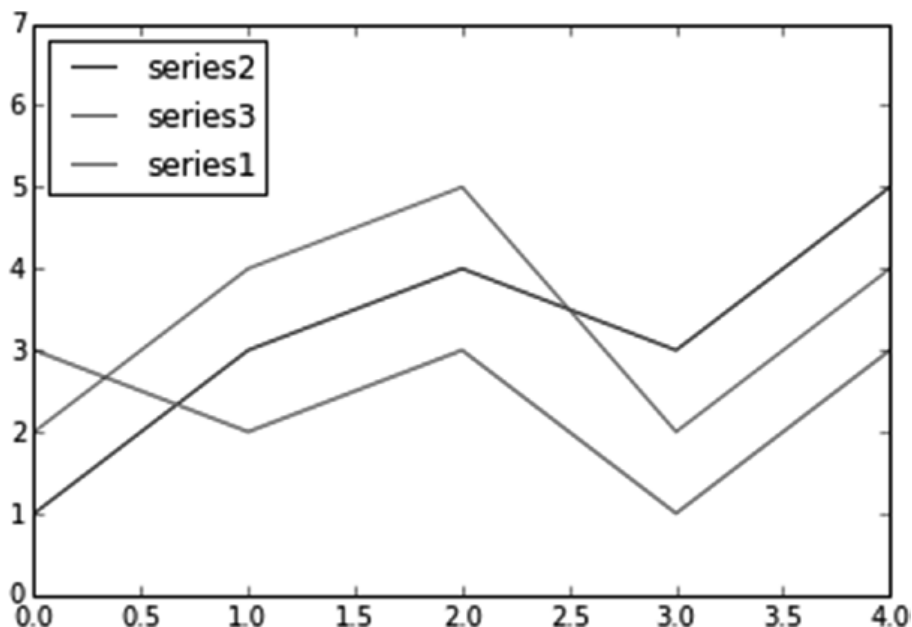


Figure 7-32. The multiseries line chart displays the data within a pandas dataframe

Histograms

A *histogram* consists of adjacent rectangles erected on the x-axis, split into discrete intervals called *bins*, and with an area proportional to the frequency of the occurrences for that bin. This kind of visualization is commonly used in statistical studies about distribution of samples.

In order to represent a histogram, pyplot provides a special function called `hist()`. This graphic function also has a feature that other functions producing charts do not have. The `hist()` function, in addition to drawing the histogram, returns a tuple of values that are the results of the calculation of the histogram. In fact the `hist()` function can also implement the calculation of the histogram, that is, it is sufficient to provide a series of samples of values as an argument and the number of bins in which to be divided, and it will take care of dividing the range of samples in many intervals (bins), and then calculate the occurrences for each bin. The result of this operation, in addition to being shown in graphical form (see Figure 7-33), will be returned in the form of a tuple.

(n, bins, patches)

To understand this operation, a practical example is best. Then you can generate a population of 100 random values from 0 to 100 using the `random.randint()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: pop = np.random.randint(0,100,100)
...: pop
```

Out[]:

```
array([32, 14, 55, 33, 54, 85, 35, 50, 91, 54, 44, 74, 77,  6, 77, 74,  2,
       54, 14, 30, 80, 70,  6, 37, 62, 68, 88,  4, 35, 97, 50, 85, 19, 90,
       65, 86, 29, 99, 15, 48, 67, 96, 81, 34, 43, 41, 21, 79, 96, 56, 68,
       49, 43, 93, 63, 26,  4, 21, 19, 64, 16, 47, 57,  5, 12, 28,  7, 75,
        6, 33, 92, 44, 23, 11, 61, 40,  5, 91, 34, 58, 48, 75, 10, 39, 77,
       70, 84, 95, 46, 81, 27,  6, 83,  9, 79, 39, 90, 77, 94, 29])
```

Now, create the histogram of these samples by passing as an argument the `hist()` function. For example, you want to divide the occurrences in 20 bins (if not specified, the default value is 10 bins) and to do that you have to use the kwarg `bin` (as shown in Figure 7-33).

```
In [ ]: n,bins,patches = plt.hist(pop,bins=20)
```

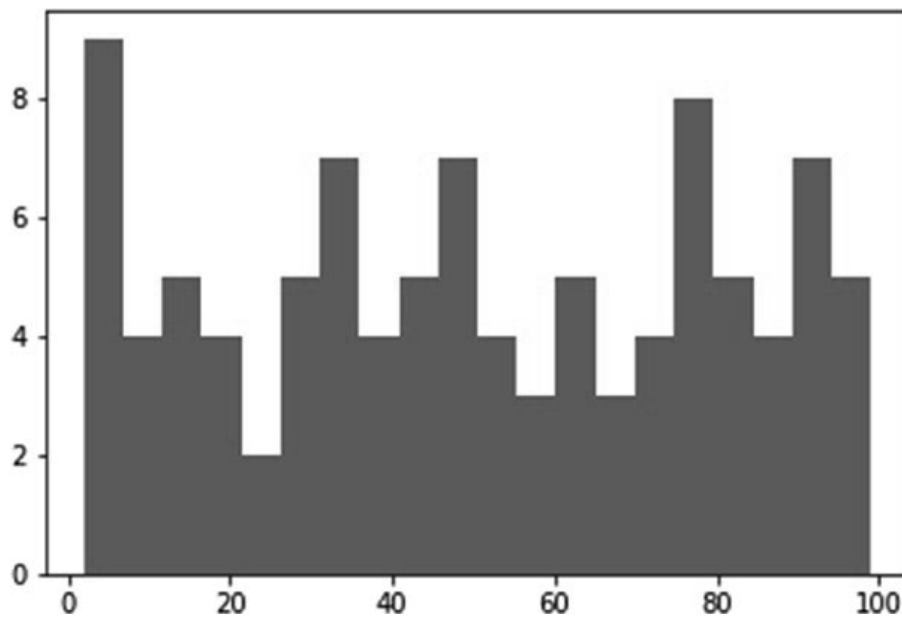


Figure 7-33. The histogram shows the occurrences in each bin

Bar Charts

Another very common type of chart is the bar chart. It is very similar to a histogram but in this case the x-axis is not used to reference numerical values but categories. The realization of the bar chart is very simple with matplotlib, using the `bar()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: index = [0,1,2,3,4]
...: values = [5,7,3,4,6]
...: plt.bar(index,values)
Out[15]: <Container object of 5 artists>
```

With this few rows of code, you will obtain a bar chart as shown in Figure 7-34.

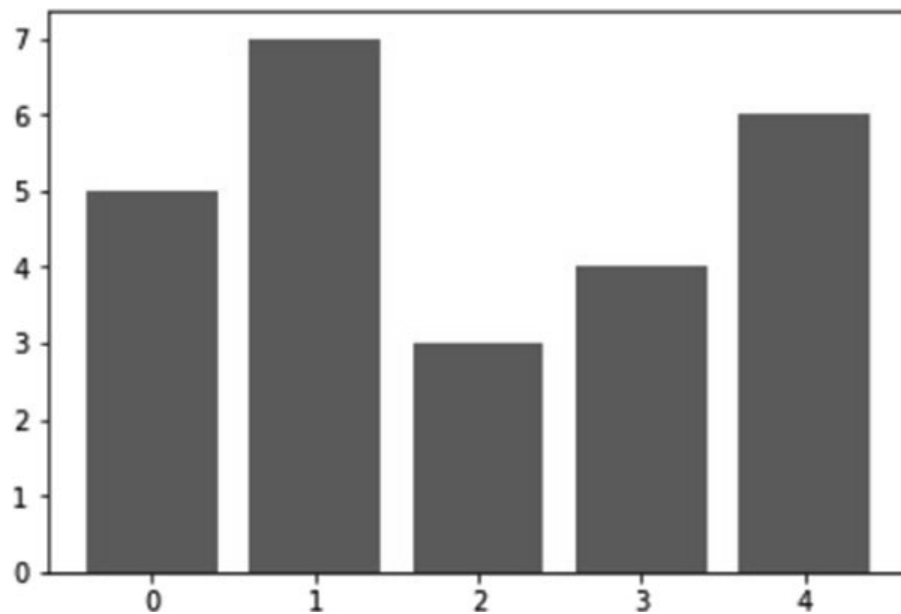


Figure 7-34. *The simplest bar chart with matplotlib*

If you look at Figure 7-34 you can see that the indices are drawn on the x-axis at the beginning of each bar. Actually, because each bar corresponds to a category, it would be better if you specify the categories through the tick label, defined by a list of strings passed to the `xticks()` function. As for the location of these tick labels, you have to pass a list containing the values corresponding to their positions on the x-axis as the first argument of the `xticks()` function. At the end you will get a bar chart, as shown in Figure 7-35.

```
In [ ]: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: plt.bar(index,values1)
...: plt.xticks(index+0.4,['A','B','C','D','E'])
```

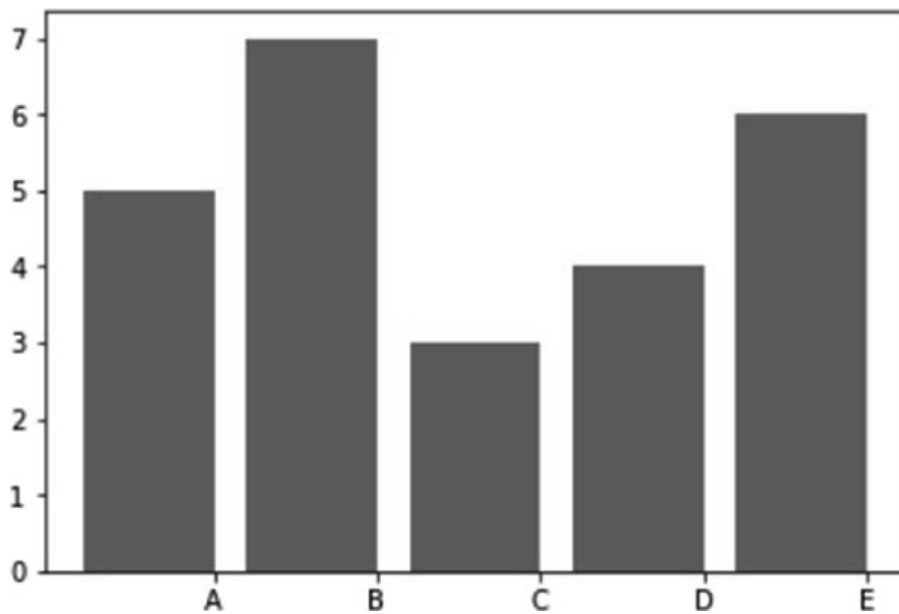


Figure 7-35. A simple bar chart with categories on the x-axis

Actually there are many other steps you can take to further refine the bar chart. Each of these finishes is set by adding a specific kwarg as an argument in the `bar()` function. For example, you can add the standard deviation values of the bar through the `yerr` kwarg along with a list containing the standard deviations. This kwarg is usually combined with another kwarg called `error_kw`, which, in turn, accepts other kwargs specialized for representing error bars. Two very specific kwargs used in this case are `eColor`, which specifies the color of the error bars, and `capsize`, which defines the width of the transverse lines that mark the ends of the error bars.

Another kwarg that you can use is `alpha`, which indicates the degree of transparency of the colored bar. Alpha is a value ranging from 0 to 1. When this value is 0 the object is completely transparent to become gradually more significant with the increase of the value, until arriving at 1, at which the color is fully represented.

As usual, the use of a legend is recommended, so in this case you should use a kwarg called `label` to identify the series that you are representing.

At the end you will get a bar chart with error bars, as shown in Figure 7-36.

```
In [ ]: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: std1 = [0.8,1,0.4,0.9,1.3]
...: plt.title('A Bar Chart')
```

```

...: plt.bar(index, values1, yerr=std1, error_kw={'ecolor': '0.1',
...:      'capsize': 6}, alpha=0.7, label='First')
...: plt.xticks(index+0.4, ['A', 'B', 'C', 'D', 'E'])
...: plt.legend(loc=2)

```

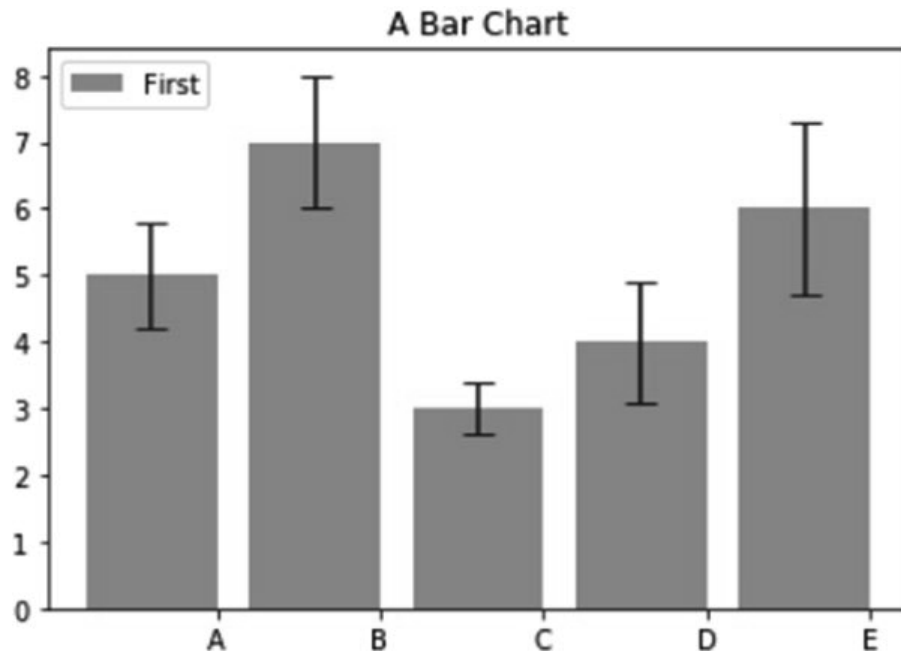


Figure 7-36. A bar chart with error bars

Horizontal Bar Charts

So far you have seen the bar chart oriented vertically. There are also bar charts oriented horizontally. This mode is implemented by a special function called `barh()`. The arguments and the kwargs valid for the `bar()` function remain the same for this function. The only change that you have to take into account is that the roles of the axes are reversed. Now, the categories are represented on the y-axis and the numerical values are shown on the x-axis (see Figure 7-37).

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: std1 = [0.8,1,0.4,0.9,1.3]
...: plt.title('A Horizontal Bar Chart')

```

```

...: plt.barh(index, values1, xerr=std1, error_kw={'ecolor': '0.1',
...: 'capsize': 6}, alpha=0.7, label='First')
...: plt.yticks(index+0.4, ['A', 'B', 'C', 'D', 'E'])
...: plt.legend(loc=5)

```

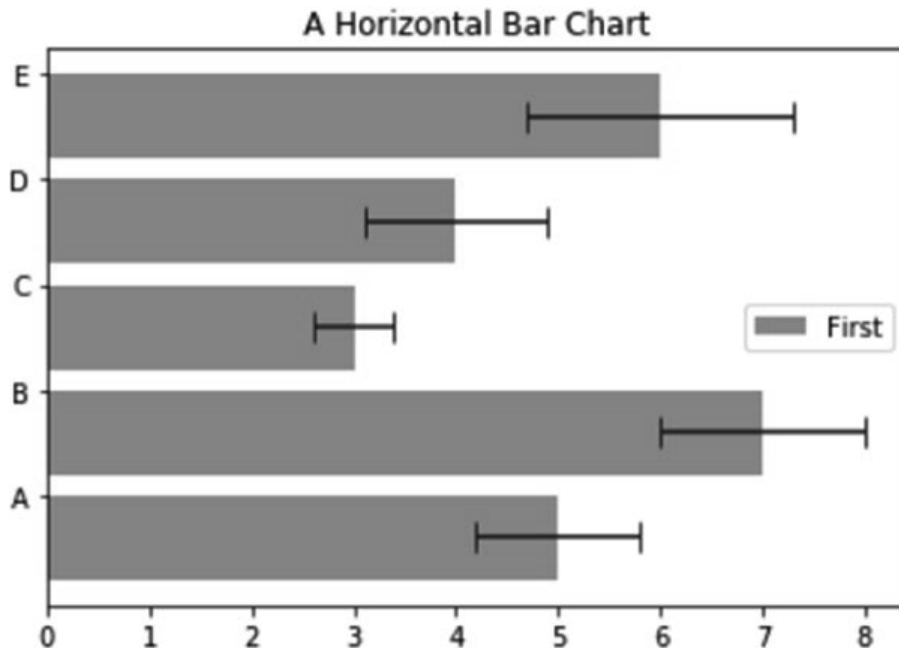


Figure 7-37. A simple horizontal bar chart

Multiserial Bar Charts

As line charts, bar charts also generally are used to simultaneously display larger series of values. But in this case it is necessary to make some clarifications on how to structure a multiseries bar chart. So far you have defined a sequence of indexes, each corresponding to a bar, to be assigned to the x-axis. These indices should represent categories. In this case, however, you have more bars that must share the same category.

One approach used to overcome this problem is to divide the space occupied by an index (for convenience its width is 1) in as many parts as are the bars sharing that index and that we want to display. Moreover, it is advisable to add space, which will serve as a gap to separate a category with respect to the next (as shown in Figure 7-38).

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: values2 = [6,6,4,5,7]
...: values3 = [5,6,5,4,6]
...: bw = 0.3
...: plt.axis([0,5,0,8])
...: plt.title('A Multiseries Bar Chart',fontsize=20)
...: plt.bar(index,values1,bw,color='b')
...: plt.bar(index+bw,values2,bw,color='g')
...: plt.bar(index+2*bw,values3,bw,color='r')
...: plt.xticks(index+1.5*bw,['A','B','C','D','E'])

```

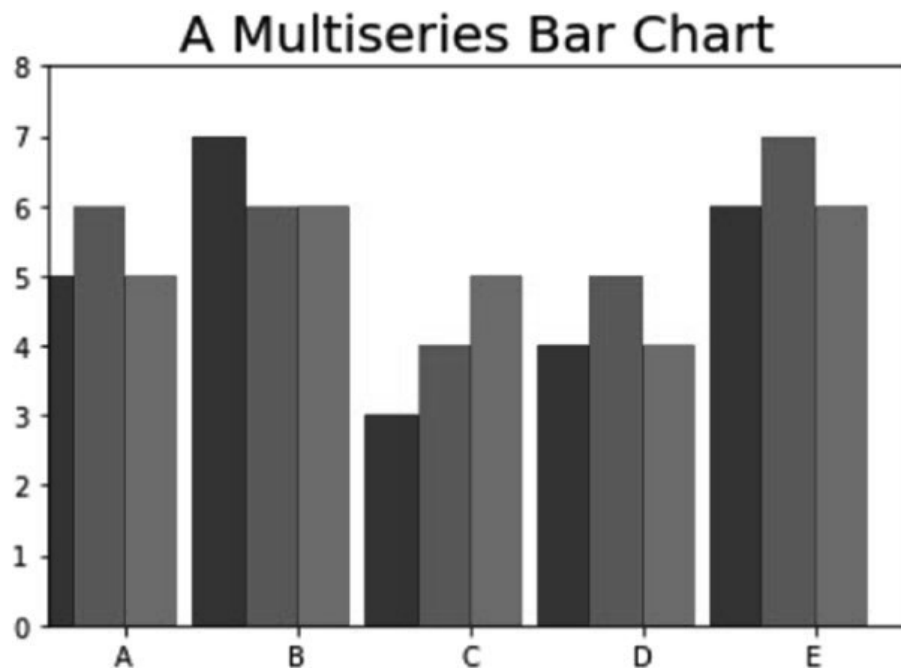


Figure 7-38. A multiseries bar chart displaying three series

Regarding the multiserie horizontal bar chart (see Figure 7-39), things are very similar. You have to replace the `bar()` function with the corresponding `barh()` function and remember to replace the `xticks()` function with the `yticks()` function. You need to reverse the range of values covered by the axes in the `axis()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: values2 = [6,6,4,5,7]
...: values3 = [5,6,5,4,6]
...: bw = 0.3
...: plt.axis([0,8,0,5])
...: plt.title('A Multiserie Horizontal Bar Chart',fontsize=20)
...: plt.barh(index,values1,bw,color='b')
...: plt.barh(index+bw,values2,bw,color='g')
...: plt.barh(index+2*bw,values3,bw,color='r')
...: plt.yticks(index+0.4,['A','B','C','D','E'])
```

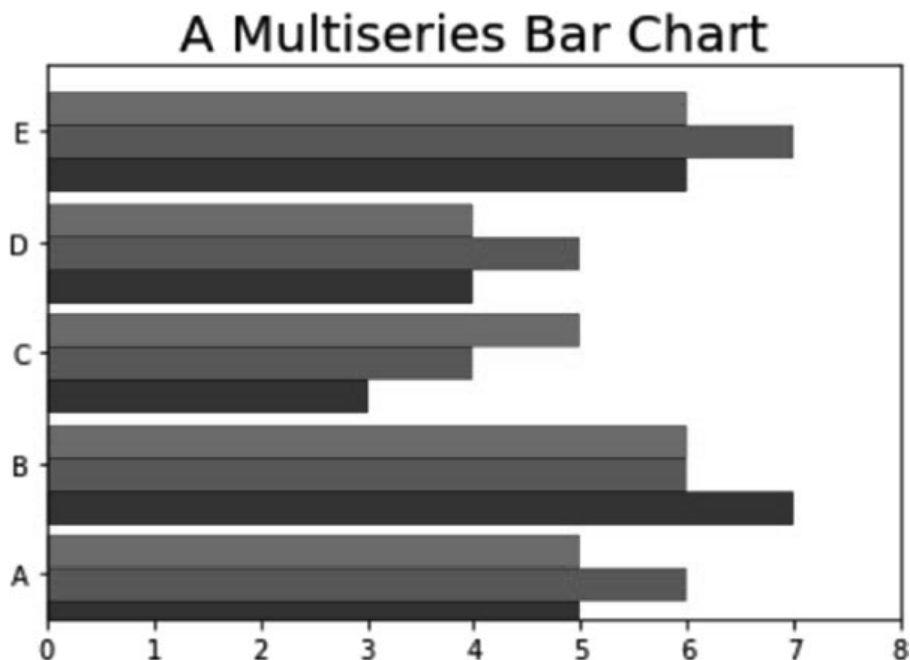


Figure 7-39. A multiserie horizontal bar chart

Multiseries Bar Charts with pandas Dataframe

As you saw in the line charts, the matplotlib library also provides the ability to directly represent the dataframe objects containing the results of data analysis in the form of bar charts. And even here it does it quickly, directly, and automatically. The only thing you need to do is use the `plot()` function applied to the dataframe object and specify inside a kwarg called `kind` to which you have to assign the type of chart you want to represent, which in this case is `bar`. Thus, without specifying any other settings, you will get the bar chart shown in Figure 7-40.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df.plot(kind='bar')
```

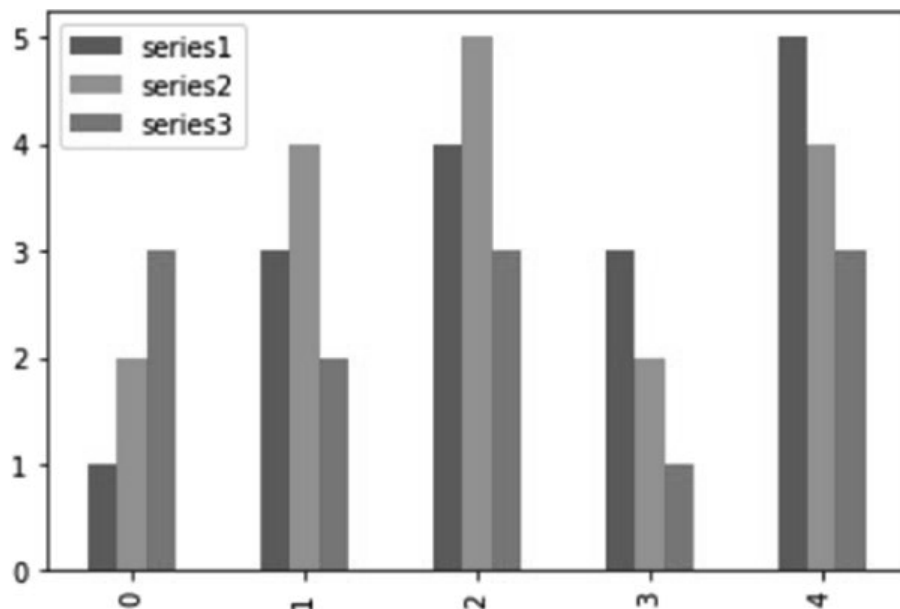


Figure 7-40. The values in a dataframe can be directly displayed as a bar chart

However, if you want to get more control, or if your case requires it, you can still extract portions of the dataframe as NumPy arrays and use them as illustrated in the previous examples in this section. That is, by passing them separately as arguments to the matplotlib functions.

Moreover, regarding the horizontal bar chart, the same rules can be applied, but remember to set `barh` as the value of the `kind` kwarg. You'll get a multiseried horizontal bar chart, as shown in Figure 7-41.

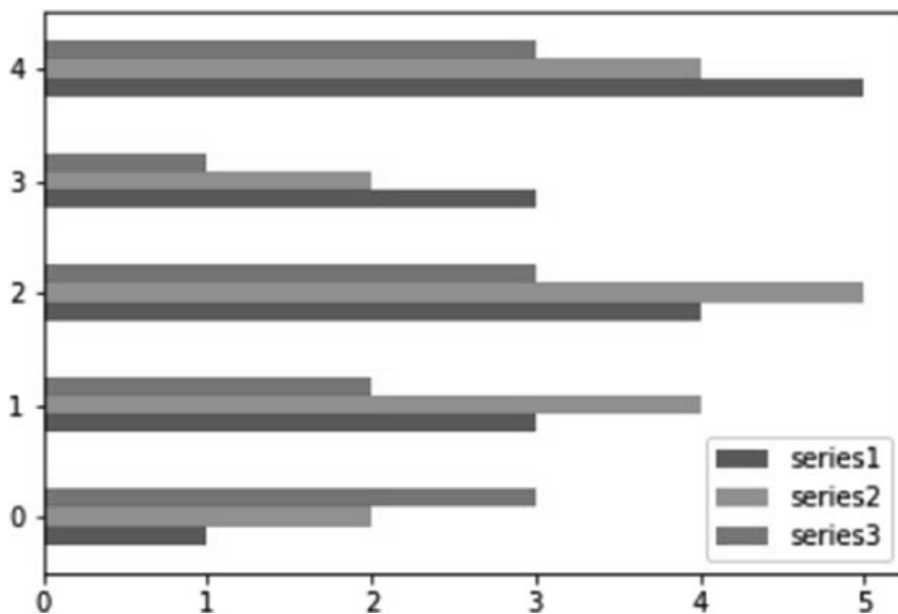


Figure 7-41. A horizontal bar chart could be a valid alternative to visualize your dataframe values

Multiseries Stacked Bar Charts

Another form to represent a multiseries bar chart is in the stacked form, in which the bars are stacked one on the other. This is especially useful when you want to show the total value obtained by the sum of all the bars.

To transform a simple multiseries bar chart in a stacked one, you add the `bottom` kwarg to each `bar()` function. Each series must be assigned to the corresponding `bottom` kwarg. At the end you will obtain the stacked bar chart, as shown in Figure 7-42.


```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: index = np.arange(4)
...: plt.axis([-0.5,3.5,0,15])
...: plt.title('A Multiseries Stacked Bar Chart')
...: plt.bar(index,series1,color='r')
...: plt.bar(index,series2,color='b',bottom=series1)
...: plt.bar(index,series3,color='g',bottom=(series2+series1))
...: plt.xticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])

```

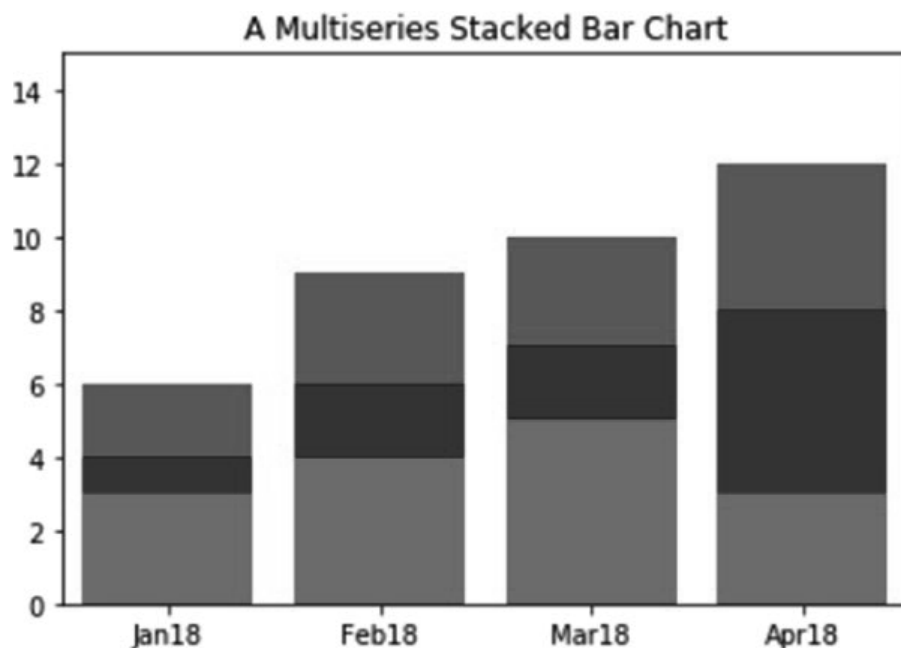


Figure 7-42. *A multiseries stacked bar*

Here too, in order to create the equivalent horizontal stacked bar chart, you need to replace the `bar()` function with `barh()` function, being careful to change the other parameters as well. Indeed the `xticks()` function should be replaced with the `yticks()` function because the labels of the categories must now be reported on the y-axis. After making all these changes, you will obtain the horizontal stacked bar chart as shown in Figure 7-43.

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(4)
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: plt.axis([0,15,-0.5,3.5])
...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
...: plt.barh(index,series1,color='r')
...: plt.barh(index,series2,color='g',left=series1)
...: plt.barh(index,series3,color='b',left=(series1+series2))
...: plt.yticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])

```

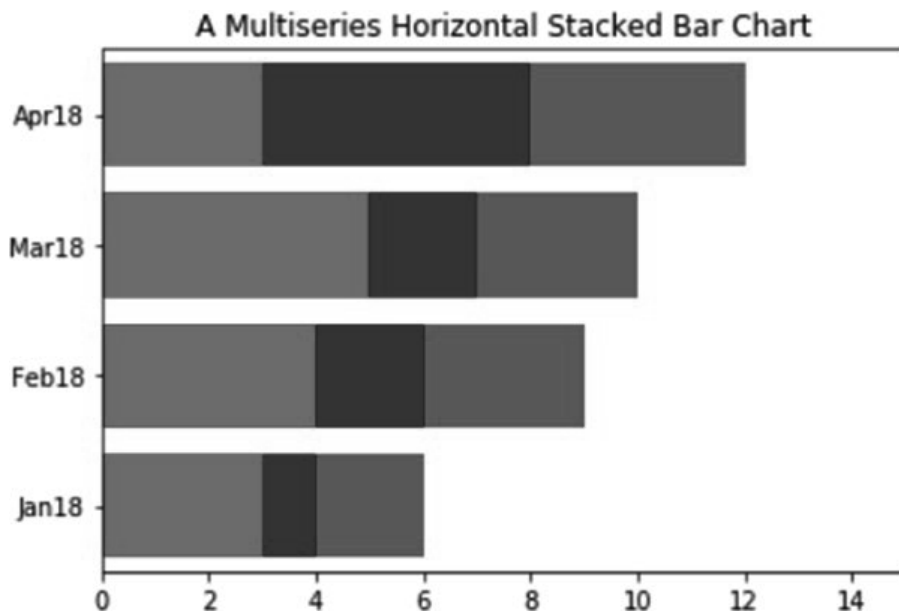


Figure 7-43. A multiseries horizontal stacked bar chart

So far the various series have been distinguished by using different colors. Another mode of distinction between the various series is to use hatches that allow you to fill the various bars with strokes drawn in a different way. To do this, you have first to set the color of the bar as white and then you have to use the hatch kwarg to define how the hatch is to be set. The various hatches have codes distinguishable among these characters (|, /, -, \, *, -) corresponding to the line style filling the bar. The more a symbol is replicated, the denser the lines forming the hatch will be. For example, /// is more dense than //, which is more dense than / (see Figure 7-44).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(4)
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: plt.axis([0,15,-0.5,3.5])
...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
...: plt.barh(index,series1,color='w',hatch='xx')
...: plt.barh(index,series2,color='w',hatch='///', left=series1)
...: plt.barh(index,series3,color='w',hatch='\\\\\\\\',left=(series1+series2))
...: plt.yticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])

Out[453]:
([<matplotlib.axis.YTick at 0x2a9f0748>,
 <matplotlib.axis.YTick at 0x2a9e1f98>,
 <matplotlib.axis.YTick at 0x2ac06518>,
 <matplotlib.axis.YTick at 0x2ac52128>],
 <a list of 4 Text yticklabel objects>)
```

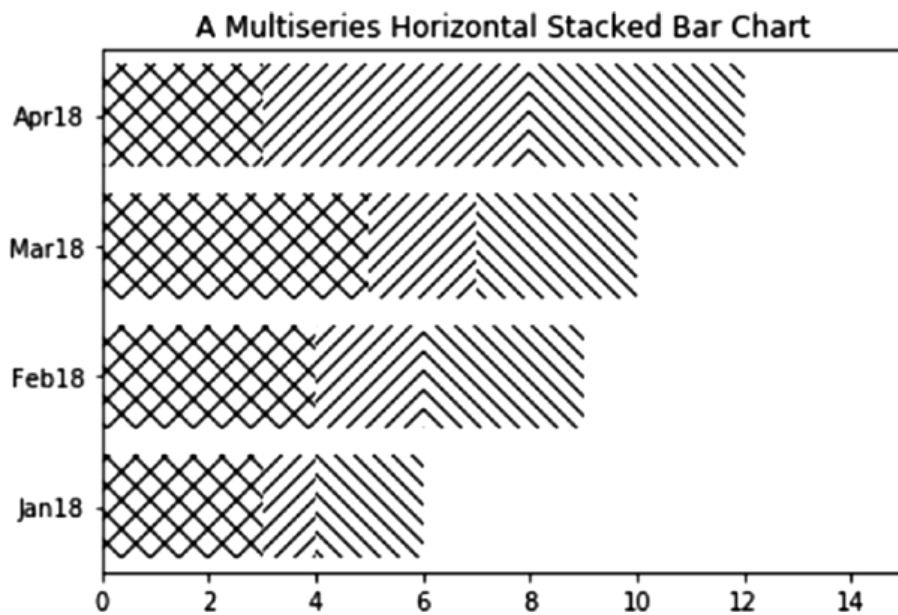


Figure 7-44. The stacked bars can be distinguished by their hatches

Stacked Bar Charts with a pandas Dataframe

Also with regard to stacked bar charts, it is very simple to directly represent the values contained in the dataframe object by using the `plot()` function. You need only to add as an argument the `stacked` kwarg set to `True` (see Figure 7-45).

```
In [ ]: import matplotlib.pyplot as plt
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df.plot(kind='bar', stacked=True)
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0xcda8f98>
```

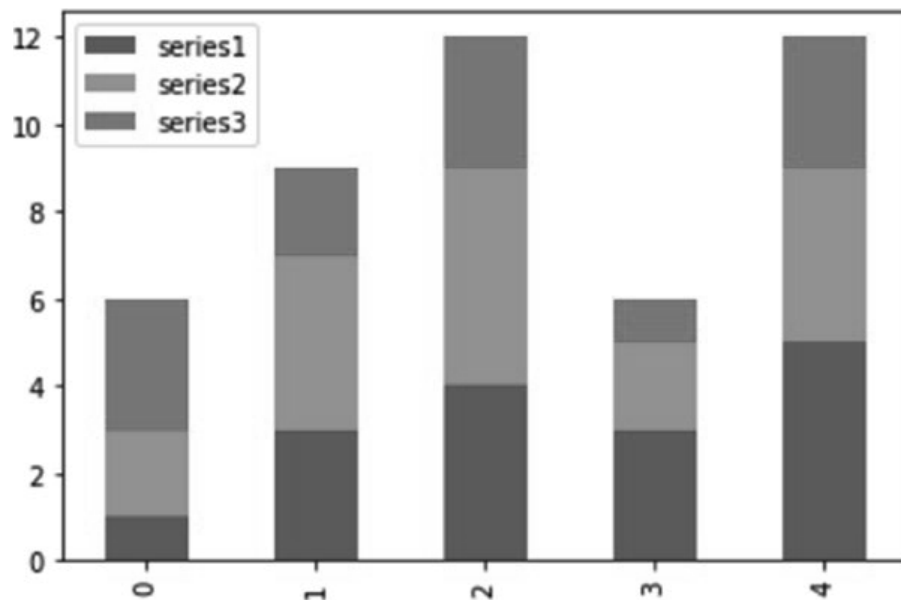


Figure 7-45. The values of a dataframe can be directly displayed as a stacked bar chart

Other Bar Chart Representations

Another type of very useful representation is that of a bar chart for comparison, where two series of values sharing the same categories are compared by placing the bars in opposite directions along the y-axis. In order to do this, you have to put the y values of one of the two series in a negative form. Also in this example, you will see the possibility of coloring the inner color of the bars in a different way. In fact, you can do this by setting the two different colors on a specific kwarg: `facecolor`.

Furthermore, in this example, you will see how to add the y value with a label at the end of each bar. This could be useful to increase the readability of the bar chart. You can do this using a for loop in which the `text()` function will show the y value. You can adjust the label position with the two kwargs called `ha` and `va`, which control the horizontal and vertical alignment, respectively. The result will be the chart shown in Figure 7-46.

```
In [ ]: import matplotlib.pyplot as plt
...: x0 = np.arange(8)
...: y1 = np.array([1,3,4,6,4,3,2,1])
...: y2 = np.array([1,2,5,4,3,3,2,1])
...: plt.ylim(-7,7)
```

```

...: plt.bar(x0,y1,0.9,facecolor='r')
...: plt.bar(x0,-y2,0.9,facecolor='b')
...: plt.xticks(())
...: plt.grid(True)
...: for x, y in zip(x0, y1):
...:     plt.text(x + 0.4, y + 0.05, '%d' % y, ha='center', va= 'bottom')
...:
...: for x, y in zip(x0, y2):
...:     plt.text(x + 0.4, -y - 0.05, '%d' % y, ha='center', va= 'top')

```

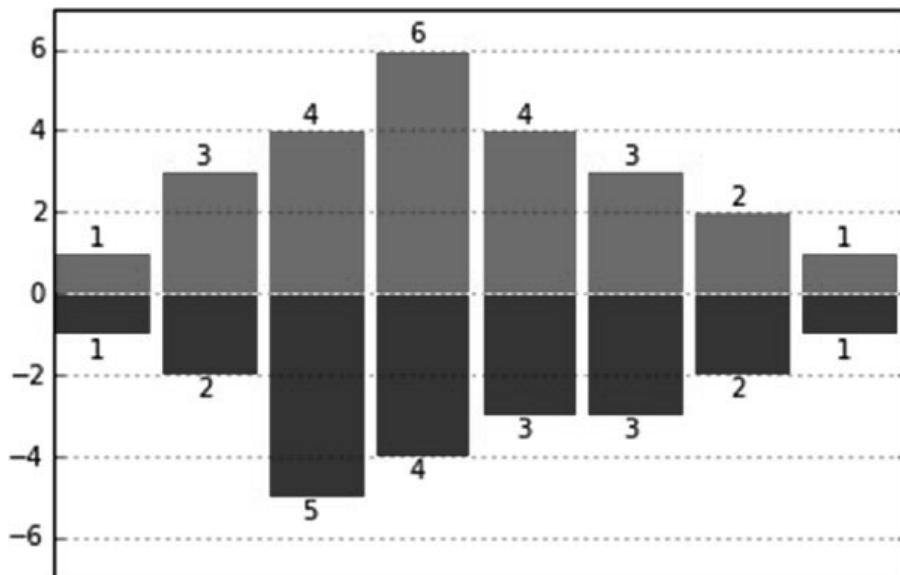


Figure 7-46. Two series can be compared using this kind of bar chart

Pie Charts

An alternative way to display data to the bar charts is the pie chart, easily obtainable using the `pie()` function.

Even for this type of function, you pass as the main argument a list containing the values to be displayed. I chose the percentages (their sum is 100), but you can use any kind of value. It will be up to the `pie()` function to inherently calculate the percentage occupied by each value.

Also with this type of representation, you need to define some key features making use of the kwargs. For example, if you want to define the sequence of the colors, which will be assigned to the sequence of input values correspondingly, you have to use the `colors` kwarg. Therefore, you have to assign a list of strings, each containing the name of the desired color. Another important feature is to add labels to each slice of the pie. To do this, you have to use the `labels` kwarg to which you will assign a list of strings containing the labels to be displayed in sequence.

In addition, in order to draw the pie chart in a perfectly spherical way, you have to add the `axis()` function to the end, specifying the string `'equal'` as an argument. You will get a pie chart as shown in Figure 7-47.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia', 'Samsung', 'Apple', 'Lumia']
...: values = [10, 30, 45, 15]
...: colors = ['yellow', 'green', 'red', 'blue']
...: plt.pie(values, labels=labels, colors=colors)
...: plt.axis('equal')
```

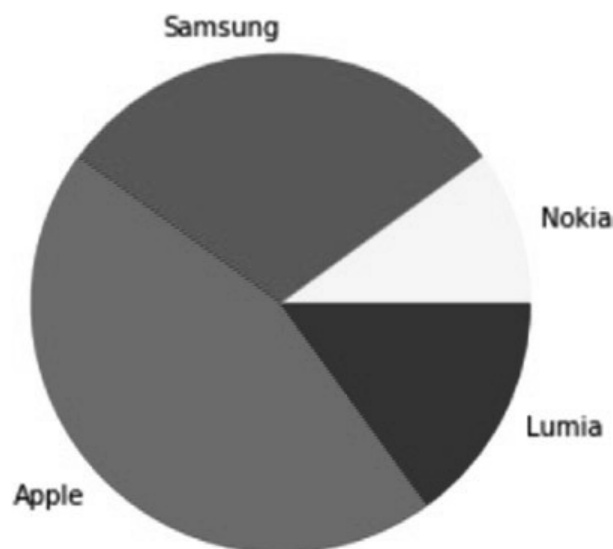


Figure 7-47. A very simple pie chart

To add complexity to the pie chart, you can draw it with a slice extracted from the pie. This is usually done when you want to focus on a specific slice. In this case, for example, you would highlight the slice referring to Nokia. In order to do this, there is a special kwarg named `explode`. It is nothing but a sequence of float values of 0 or 1, where 1 corresponds to the fully extended slice and 0 corresponds to slices completely in the pie. All intermediate values correspond to an intermediate degree of extraction (see Figure 7-48).

You can also add a title to the pie chart with the `title()` function. You can also adjust the angle of rotation of the pie by adding the `startangle` kwarg, which takes an integer value between 0 and 360, which are the degrees of rotation precisely (0 is the default value).

The modified chart should appear as shown in Figure 7-48.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia','Samsung','Apple','Lumia']
...: values = [10,30,45,15]
...: colors = ['yellow','green','red','blue']
...: explode = [0.3,0,0,0]
...: plt.title('A Pie Chart')
...: plt.pie(values,labels=labels,colors=colors,explode=explode,
startangle=180)
...: plt.axis('equal')
```

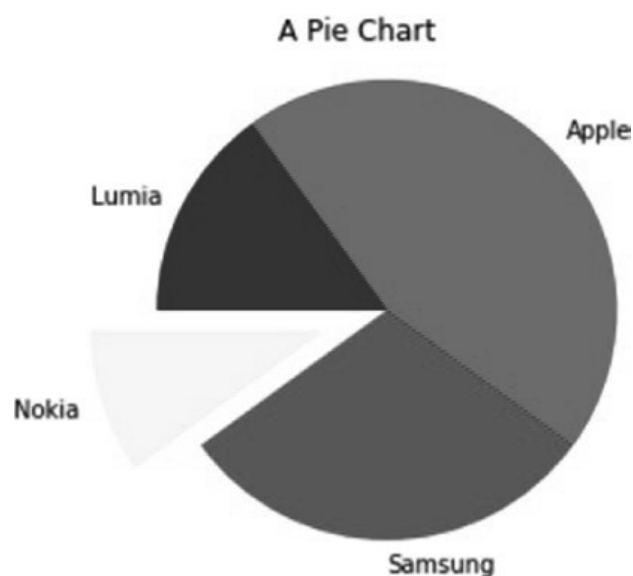


Figure 7-48. A more advanced pie chart

But the possible additions that you can insert in a pie chart do not end here. For example, a pie chart does not have axes with ticks and so it is difficult to imagine the perfect percentage represented by each slice. To overcome this, you can use the `autopct` kwarg, which adds to the center of each slice a text label showing the corresponding value.

If you want to make it an even more appealing image, you can add a shadow with the `shadow` kwarg setting it to `True`. In the end you will get a pie chart as shown in Figure 7-49.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia', 'Samsung', 'Apple', 'Lumia']
...: values = [10, 30, 45, 15]
...: colors = ['yellow', 'green', 'red', 'blue']
...: explode = [0.3, 0, 0, 0]
...: plt.title('A Pie Chart')
...: plt.pie(values, labels=labels, colors=colors, explode=explode,
...:         shadow=True, autopct='%1.1f%%', startangle=180)
...: plt.axis('equal')
```

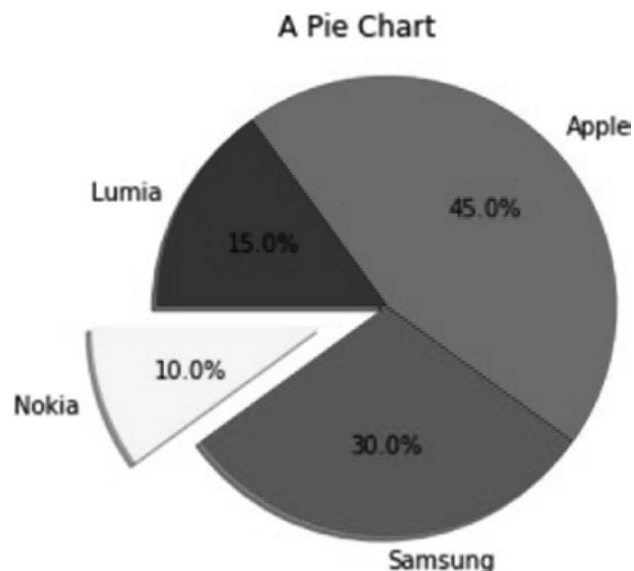


Figure 7-49. *An even more advanced pie chart*

Pie Charts with a pandas Dataframe

Even for the pie chart, you can represent the values contained within a dataframe object. In this case, however, the pie chart can represent only one series at a time, so in this example you will display only the values of the first series specifying `df['series1']`. You have to specify the type of chart you want to represent through the `kind` kwarg in the `plot()` function, which in this case is `pie`. Furthermore, because you want to represent a pie chart as perfectly circular, it is necessary that you add the `figsize` kwarg. At the end you will obtain a pie chart as shown in Figure 7-50.

```
In [ ]: import matplotlib.pyplot as plt
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df['series1'].plot(kind='pie',figsize=(6,6))
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0xe1ba710>
```

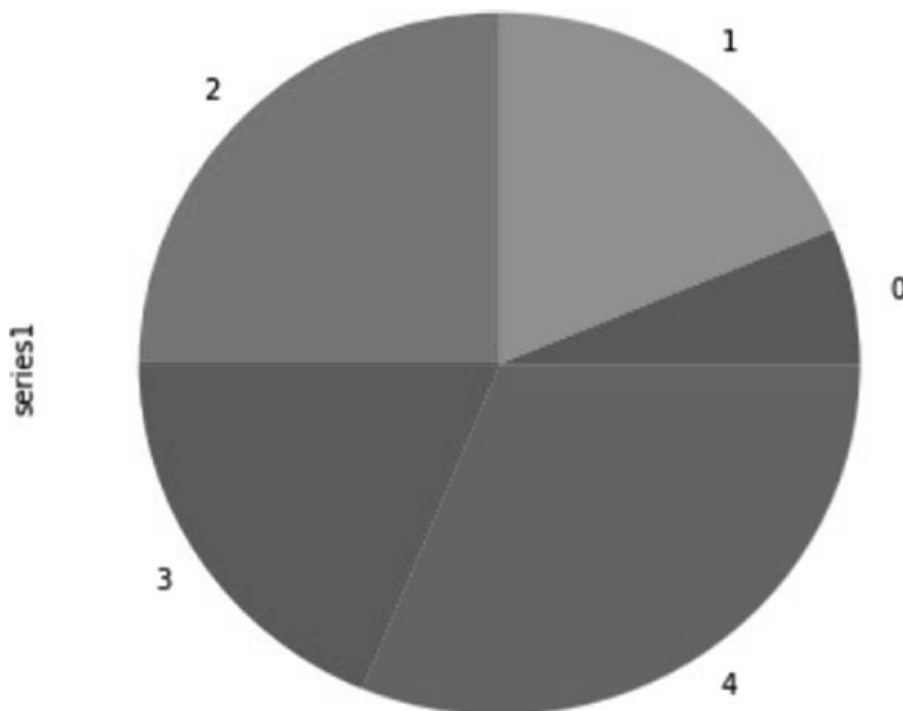


Figure 7-50. The values in a pandas dataframe can be directly drawn as a pie chart

Advanced Charts

In addition to the more classical charts such as bar charts or pie charts, you might want to represent your results in an alternative ways. On the Internet and in various publications there are many examples in which many alternative graphics solutions are discussed and proposed, some really brilliant and captivating. This section only shows some graphic representations; a more detailed discussion about this topic is beyond the purpose of this book. You can use this section as an introduction to a world that is constantly expanding: data visualization.

Contour Plots

A quite common type of chart in the scientific world is the *contour plot* or *contour map*. This visualization is in fact suitable for displaying three-dimensional surfaces through a contour map composed of curves closed showing the points on the surface that are located at the same level, or that have the same z value.

Although visually the contour plot is a very complex structure, its implementation is not so difficult, thanks to the matplotlib library. First, you need the function $z = f(x, y)$ for generating a three-dimensional surface. Then, once you have defined a range of values x, y that will define the area of the map to be displayed, you can calculate the z values for each pair (x, y) , applying the function $f(x, y)$ just defined in order to obtain a matrix of z values. Finally, thanks to the `contour()` function, you can generate the contour map of the surface. It is often desirable to add also a color map along with a contour map. That is, the areas delimited by the curves of level are filled by a color gradient, defined by a color map. For example, as in Figure 7-51, you may indicate negative values with increasingly dark shades of blue, and move to yellow and then red with the increase of positive values.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: dx = 0.01; dy = 0.01
...: x = np.arange(-2.0,2.0,dx)
...: y = np.arange(-2.0,2.0,dy)
...: X,Y = np.meshgrid(x,y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
```

```

...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8)
...: plt.clabel(C, inline=1, fontsize=10)

```

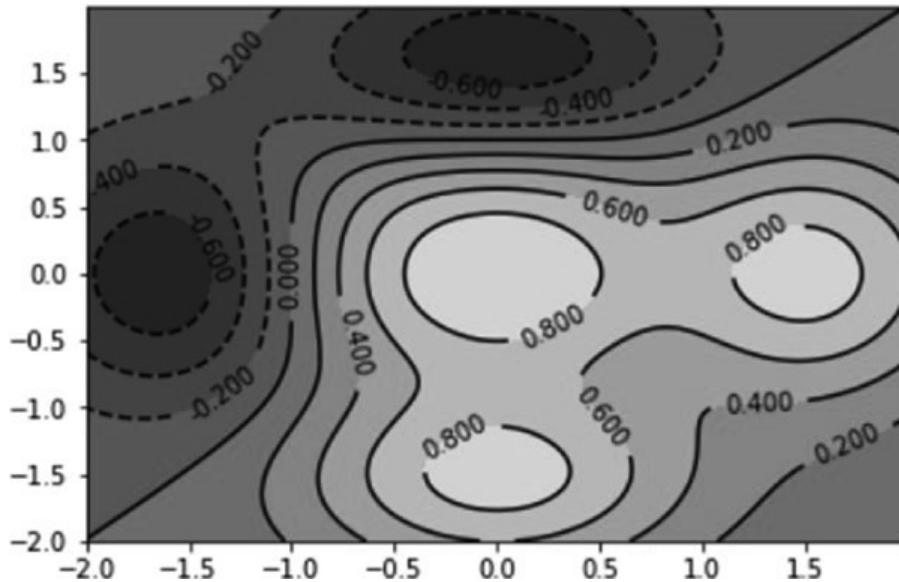


Figure 7-51. A contour map can describe the z values of a surface

The standard color gradient (color map) is represented in Figure 7-51. Actually you choose among a large number of color maps available just specifying them with the `cmap` kwarg.

Furthermore, when you have to deal with this kind of representation, adding a color scale as a reference to the side of the graph is almost a must. This is possible by simply adding the `colorbar()` function at the end of the code. In Figure 7-52 you can see another example of color map that starts from black, passes through red, then turns yellow until reaching white for the highest values. This color map is `plt.cm.hot`.

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: dx = 0.01; dy = 0.01
...: x = np.arange(-2.0,2.0,dx)
...: y = np.arange(-2.0,2.0,dy)
...: X,Y = np.meshgrid(x,y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)

```

```

...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8,cmap=plt.cm.hot)
...: plt.clabel(C, inline=1, fontsize=10)
...: plt.colorbar()

```

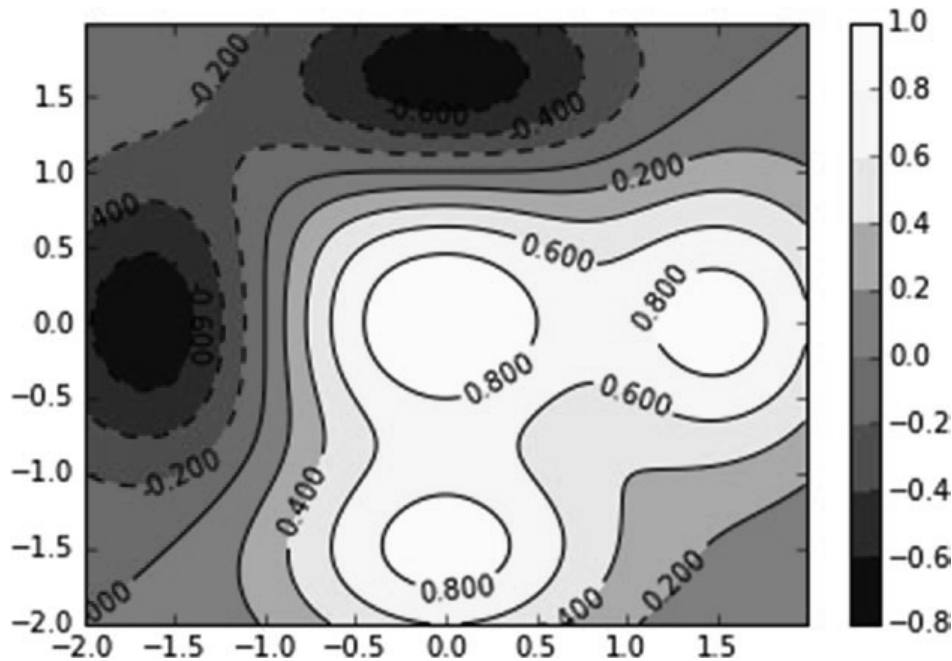


Figure 7-52. The “hot” color map gradient gives an attractive look to the contour map

Polar Charts

Another type of advanced chart that is popular is the *polar chart*. This type of chart is characterized by a series of sectors that extend radially; each of these areas will occupy a certain angle. Thus you can display two different values assigning them to the magnitudes that characterize the polar chart: the extension of the radius r and the angle θ occupied by the sector. These in fact are the polar coordinates (r, θ) , an alternative way of representing functions at the coordinate axes. From the graphical point of view, you could imagine it as a kind of chart that has characteristics both of the pie chart and of the bar chart. In fact as the pie chart, the angle of each sector gives percentage information represented by that category with respect to the total. As for the bar chart, the radial extension is the numerical value of that category.

So far we have used the standard set of colors using single characters as the color code (e.g., r to indicate red). In fact you can use any sequence of colors you want. You have to define a list of string values that contain RGB codes in the `#rrggbb` format corresponding to the colors you want.

Oddly, to get a polar chart you have to use the `bar()` function and pass the list containing the angles θ and a list of the radial extension of each sector. The result will be a polar chart, as shown in Figure 7-53.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: N = 8
...: theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
...: radii = np.array([4, 7, 5, 3, 1, 5, 6, 7])
...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
...: colors = np.array(['#4bb2c5', '#c5b47f', '#EAA228', '#579575',
...:                   '#839557', '#958c12', '#953579', '#4b5de4'])
...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0,
...:                 color=colors)
```

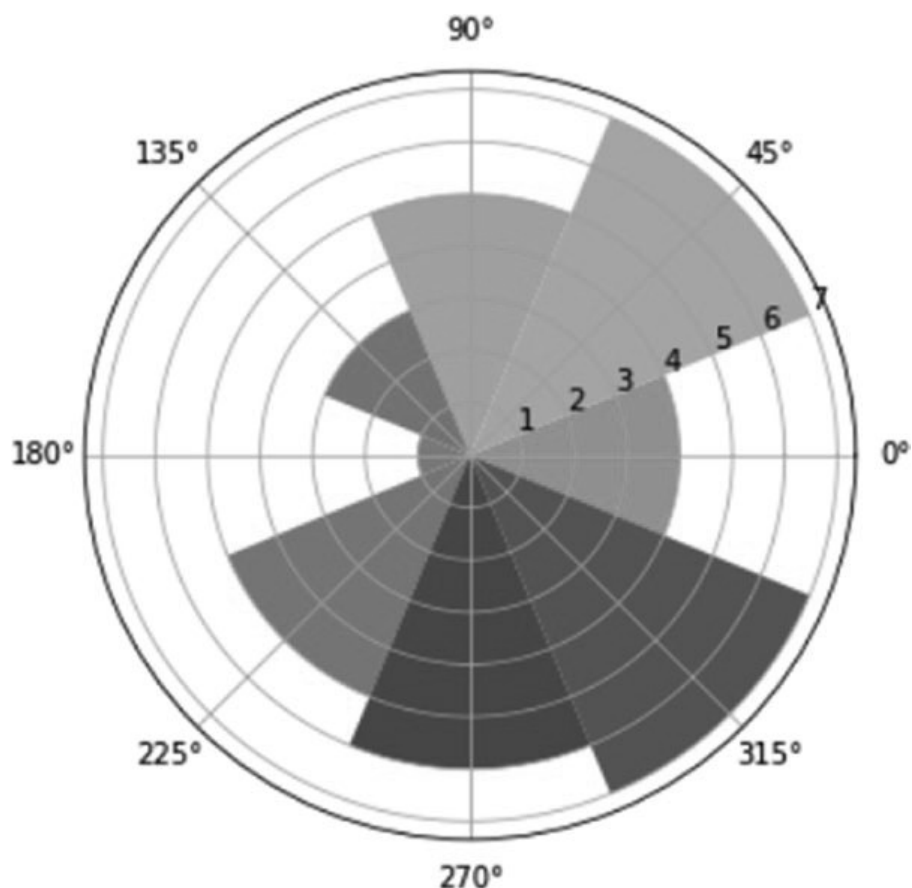


Figure 7-53. A polar chart

In this example, you have defined the sequence of colors using the format `#rrggbb`, but you can also specify a sequence of colors as strings with their actual name (see Figure 7-54).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: N = 8
...: theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
...: radii = np.array([4, 7, 5, 3, 1, 5, 6, 7])
...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
...: colors = np.array(['lightgreen', 'darkred', 'navy', 'brown',
...:                   'violet', 'plum', 'yellow', 'darkgreen'])
...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0,
...:                 color=colors)
```

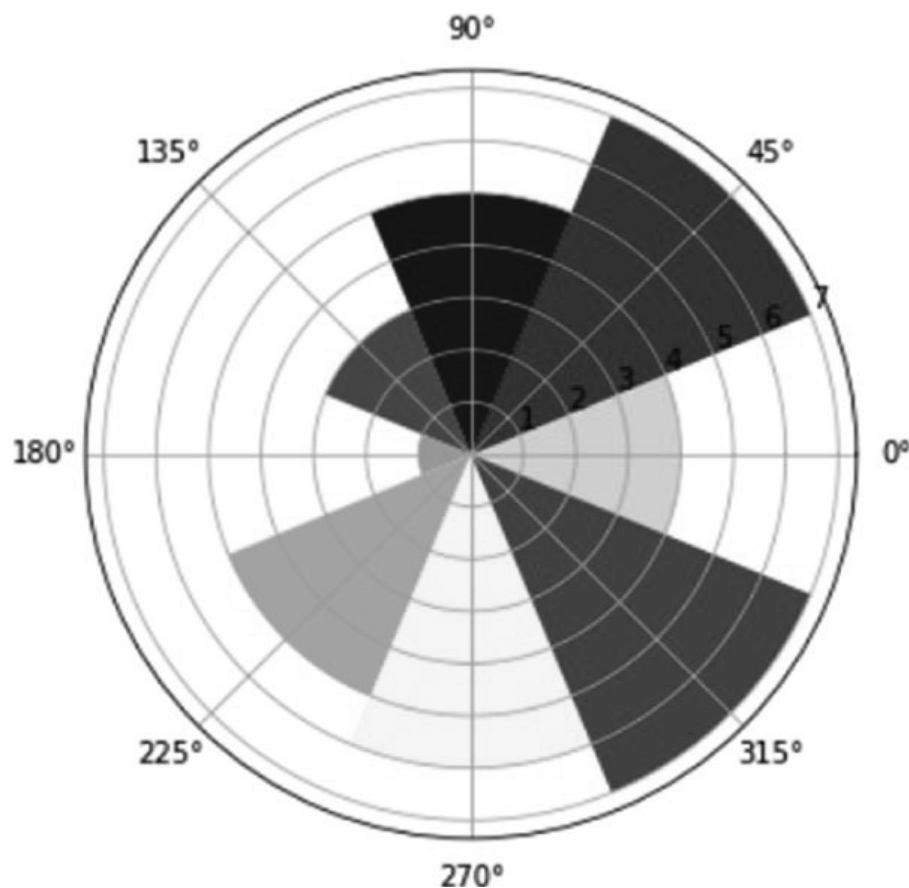


Figure 7-54. A polar chart with another sequence of colors

The mplot3d Toolkit

The `mplot3d` toolkit is included with all standard installations of `matplotlib` and allows you to extend the capabilities of visualization to 3D data. If the figure is displayed in a separate window, you can rotate the axes of the three-dimensional representation with the mouse.

With this package you are still using the `Figure` object, only that instead of the `Axes` object you will define a new kind of object, called `Axes3D`, and introduced by this toolkit. Thus, you need to add a new import to the code, if you want to use the `Axes3D` object.

```
from mpl_toolkits.mplot3d import Axes3D
```

3D Surfaces

In a previous section, you used the contour plot to represent the three-dimensional surfaces through the level lines. Using the `mplot3D` package, surfaces can be drawn directly in 3D. In this example, you will use the same function $z = f(x, y)$ you have used in the contour map.

Once you have calculated the `meshgrid`, you can view the surface with the `plot_surface()` function. A three-dimensional blue surface will appear, as in Figure 7-55.

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
...: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: X = np.arange(-2,2,0.1)
...: Y = np.arange(-2,2,0.1)
...: X,Y = np.meshgrid(X,Y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1)
```

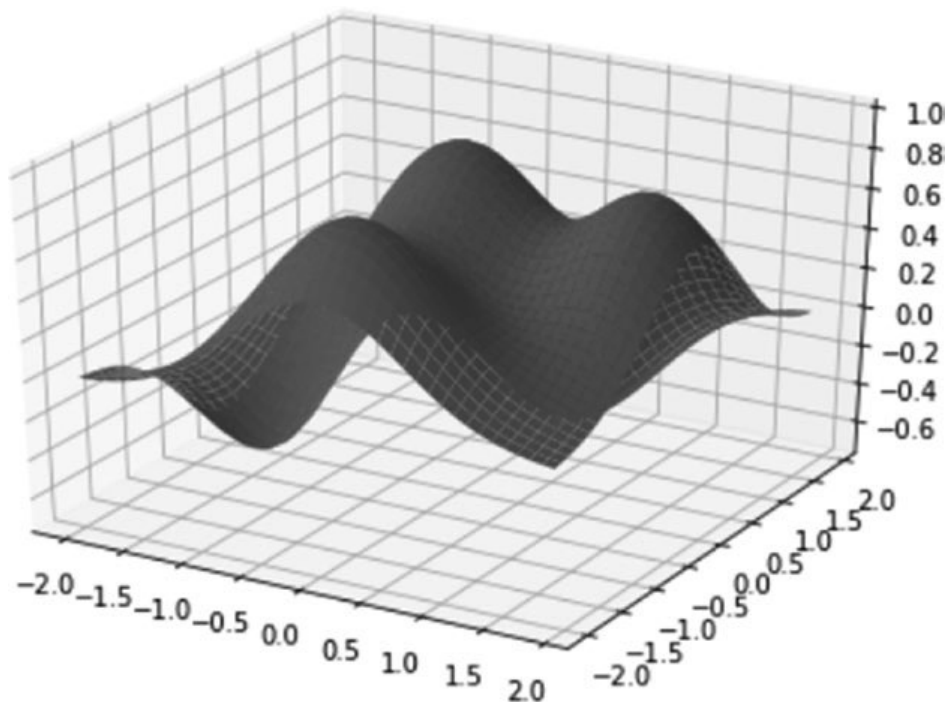



Figure 7-55. A 3D surface can be represented with the *mplot3d* toolkit

A 3D surface stands out most by changing the color map, for example by setting the `cmap` kwarg. You can also rotate the surface using the `view_init()` function. In fact, this function adjusts the view point from which you see the surface, changing the two kwargs called `elev` and `azim`. Through their combination you can get the surface displayed from any angle. The first kwarg adjusts the height at which the surface is seen, while `azim` adjusts the angle of rotation of the surface.

For instance, you can change the color map using `plt.cm.hot` and moving the view point to `elev=30` and `azim=125`. The result is shown in Figure 7-56.

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
...: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: X = np.arange(-2,2,0.1)
...: Y = np.arange(-2,2,0.1)
...: X,Y = np.meshgrid(X,Y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1, cmap=plt.cm.hot)
...: ax.view_init(elev=30,azim=125)
```

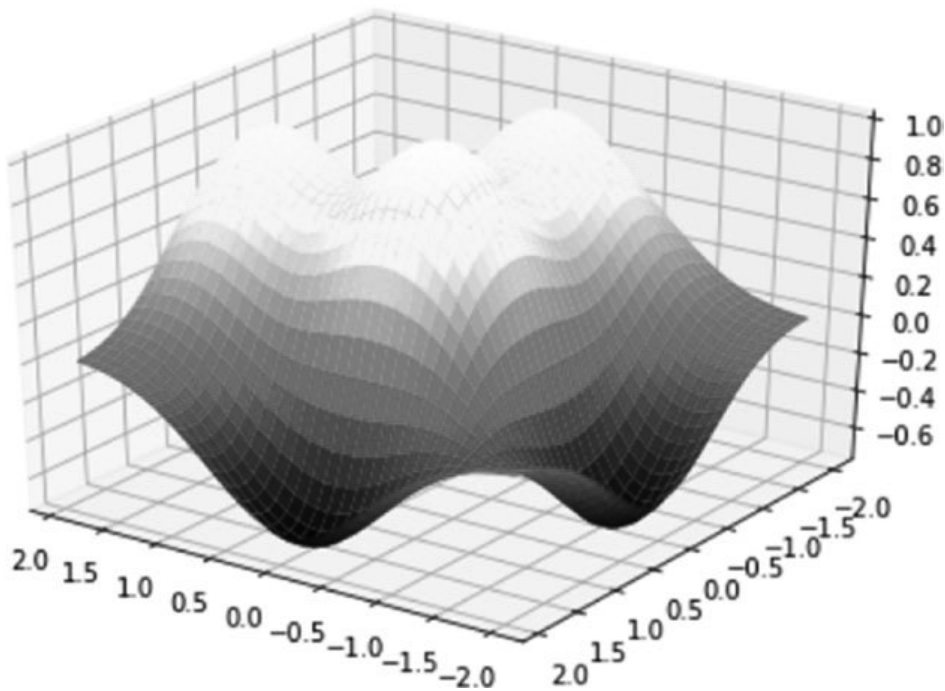


Figure 7-56. The 3D surface can be rotated and observed from a higher viewpoint

Scatter Plots in 3D

The mode most used among all 3D views remains the 3D scatter plot. With this type of visualization, you can identify if the points follow particular trends, but above all if they tend to cluster.

In this case, you will use the `scatter()` function as the 2D case but applied on the `Axes3D` object. By doing this, you can visualize different series, expressed by the calls to the `scatter()` function, all together in the same 3D representation (see Figure 7-57).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: from mpl_toolkits.mplot3d import Axes3D
...: xs = np.random.randint(30,40,100)
...: ys = np.random.randint(20,30,100)
...: zs = np.random.randint(10,20,100)
...: xs2 = np.random.randint(50,60,100)
...: ys2 = np.random.randint(30,40,100)
...: zs2 = np.random.randint(50,70,100)
...: xs3 = np.random.randint(10,30,100)
```

```

...: ys3 = np.random.randint(40,50,100)
...: zs3 = np.random.randint(40,50,100)
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.scatter(xs,ys,zs)
...: ax.scatter(xs2,ys2,zs2,c='r',marker='^')
...: ax.scatter(xs3,ys3,zs3,c='g',marker='*')
...: ax.set_xlabel('X Label')
...: ax.set_ylabel('Y Label')
...: ax.set_zlabel('Z Label')
Out[34]: <matplotlib.text.Text at 0xe1c2438>

```

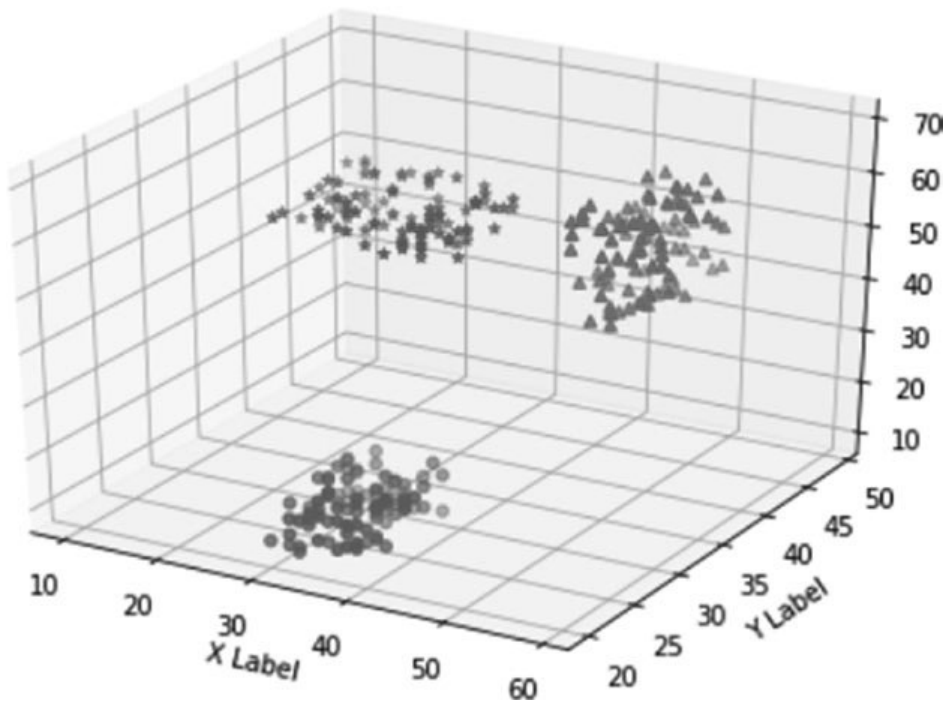


Figure 7-57. This 3D scatter plot shows three different clusters

Bar Charts in 3D

Another type of 3D plot widely used in data analysis is the 3D bar chart. Also in this case, you use the `bar()` function applied to the object `Axes3D`. If you define multiple series, you can accumulate several calls to the `bar()` function in the same 3D visualization (see Figure 7-58).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: from mpl_toolkits.mplot3d import Axes3D
...: x = np.arange(8)
...: y = np.random.randint(0,10,8)
...: y2 = y + np.random.randint(0,3,8)
...: y3 = y2 + np.random.randint(0,3,8)
...: y4 = y3 + np.random.randint(0,3,8)
...: y5 = y4 + np.random.randint(0,3,8)
...: clr = ['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557',
...:        '#958c12', '#953579', '#4b5de4']
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.bar(x,y,0,zdir='y',color=clr)
...: ax.bar(x,y2,10,zdir='y',color=clr)
...: ax.bar(x,y3,20,zdir='y',color=clr)
...: ax.bar(x,y4,30,zdir='y',color=clr)
...: ax.bar(x,y5,40,zdir='y',color=clr)
...: ax.set_xlabel('X Axis')
...: ax.set_ylabel('Y Axis')
...: ax.set_zlabel('Z Axis')
...: ax.view_init(elev=40)
```

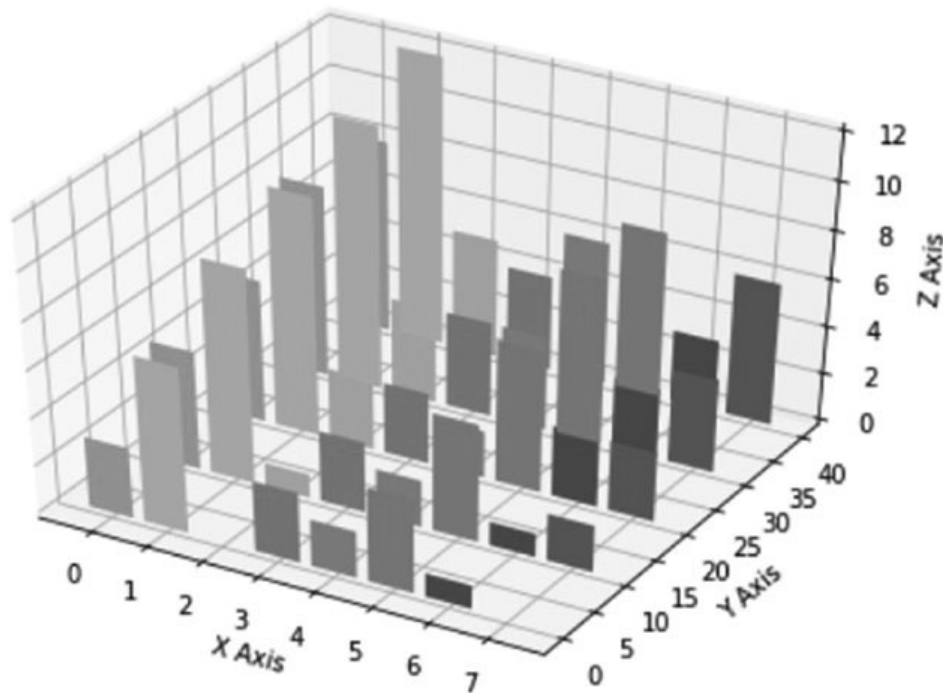


Figure 7-58. A 3D bar chart

Multi-Panel Plots

So far you’ve had the chance to see different ways of representing data through a chart. You saw how to see more charts in the same figure by separating them with subplots. In this section, you will deepen your understanding of this topic by analyzing more complex cases.

Display Subplots Within Other Subplots

Now an even more advanced method will be explained: the ability to view charts within others, enclosed within frames. Since we are talking of frames, i.e., Axes objects, you need to separate the main Axes (i.e., the general chart) from the frame you want to add that will be another instance of Axes. To do this, you use the `figures()` function to get the Figure object on which you will define two different Axes objects using the `add_axes()` function. See the result of this example in Figure 7-59.

```
In [ ]: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
```

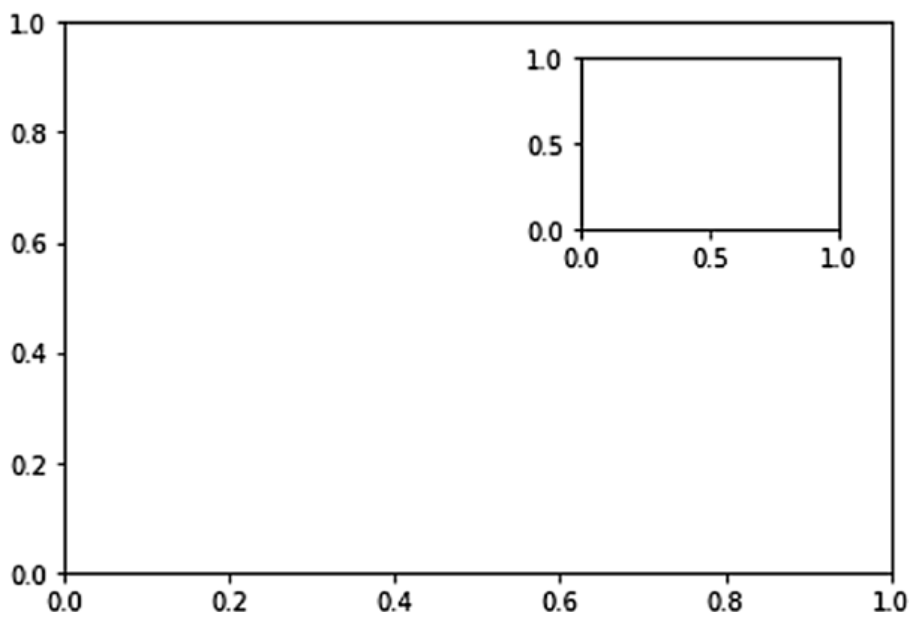


Figure 7-59. A subplot is displayed within another plot

To better understand the effect of this mode of display, you can fill the previous Axes with real data, as shown in Figure 7-60.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: fig = plt.figure()
...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
...: x1 = np.arange(10)
...: y1 = np.array([1,2,7,1,5,2,4,2,3,1])
...: x2 = np.arange(10)
...: y2 = np.array([1,3,4,5,4,5,2,6,4,3])
...: ax.plot(x1,y1)
...: inner_ax.plot(x2,y2)
Out[95]: [<matplotlib.lines.Line2D at 0x14acf6d8>]
```

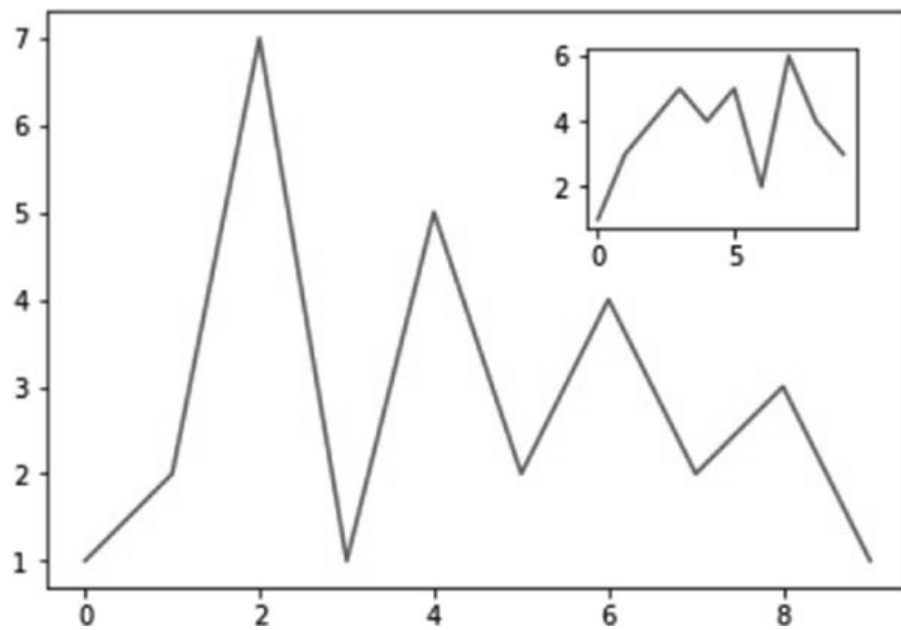


Figure 7-60. *A more realistic visualization of a subplot within another plot*

Grids of Subplots

You have already seen the creation of subplots. It is quite simple to add subplots using the `subplots()` function and by dividing a plot into sectors. `matplotlib` allows you to manage even more complex cases using another function called `GridSpec()`. This subdivision allows splitting the drawing area into a grid of sub-areas, to which you can assign one or more of them to each subplot, so that in the end you can obtain subplots with different sizes and orientations, as you can see in Figure 7-61.

```
In [ ]: import matplotlib.pyplot as plt
...: gs = plt.GridSpec(3,3)
...: fig = plt.figure(figsize=(6,6))
...: fig.add_subplot(gs[1,:2])
...: fig.add_subplot(gs[0,:2])
...: fig.add_subplot(gs[2,0])
...: fig.add_subplot(gs[:2,2])
...: fig.add_subplot(gs[2,1:])
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x12717438>
```

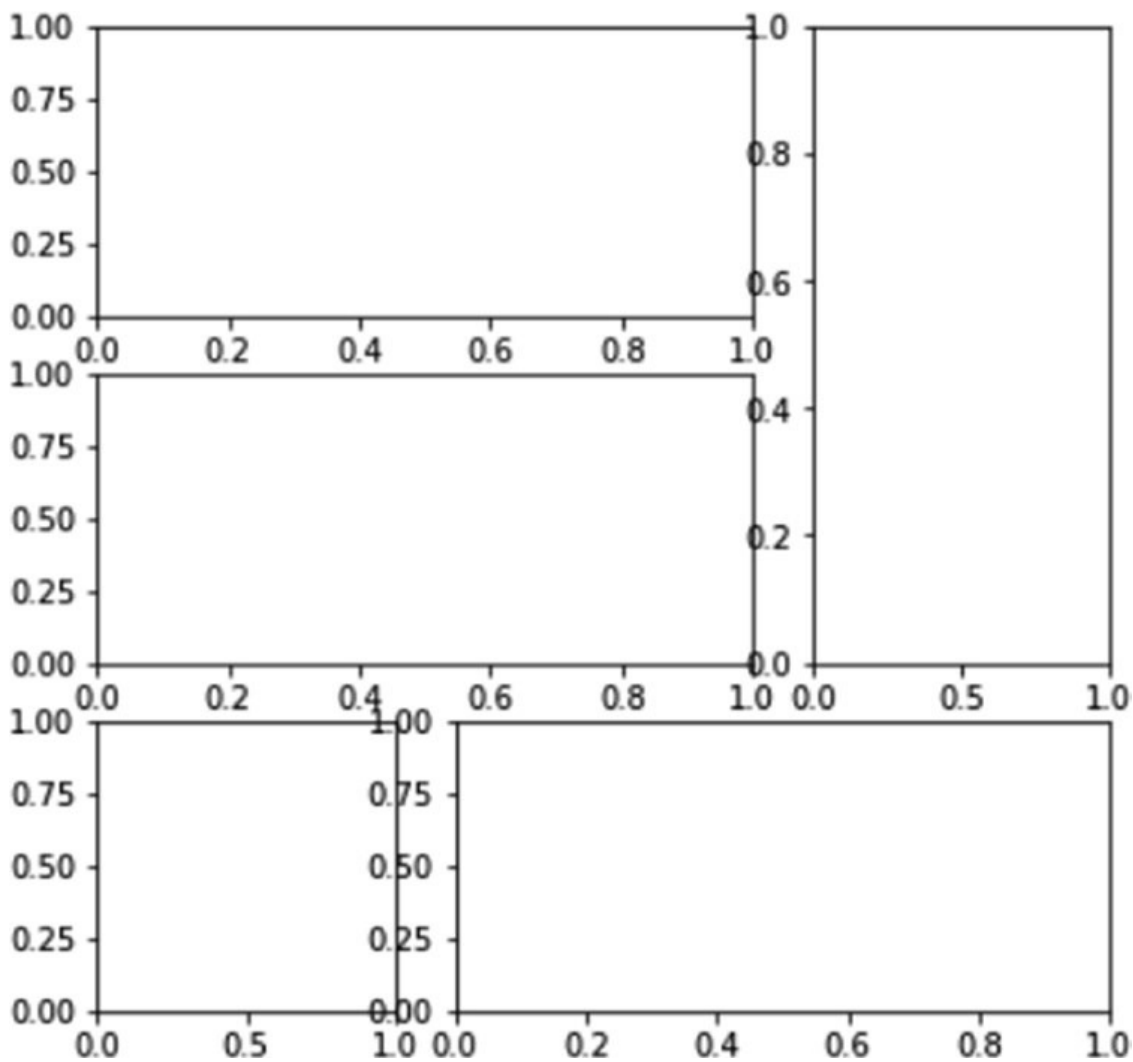


Figure 7-61. Subplots with different sizes can be defined on a grid of sub-areas

Now that it's clear to you how to manage the grid by assigning the various sectors to the subplot, it's time to see how to use these subplots. In fact, you can use the Axes object returned by each `add_subplot()` function to call the `plot()` function to draw the corresponding plot (see Figure 7-62).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: gs = plt.GridSpec(3,3)
...: fig = plt.figure(figsize=(6,6))
...: x1 = np.array([1,3,2,5])
...: y1 = np.array([4,3,7,2])
...: x2 = np.arange(5)
```



```

...: y2 = np.array([3,2,4,6,4])
...: s1 = fig.add_subplot(gs[1,:2])
...: s1.plot(x,y,'r')
...: s2 = fig.add_subplot(gs[0,:2])
...: s2.bar(x2,y2)
...: s3 = fig.add_subplot(gs[2,0])
...: s3.barh(x2,y2,color='g')
...: s4 = fig.add_subplot(gs[:2,2])
...: s4.plot(x2,y2,'k')
...: s5 = fig.add_subplot(gs[2,1:])
...: s5.plot(x1,y1,'b^',x2,y2,'yo')

```

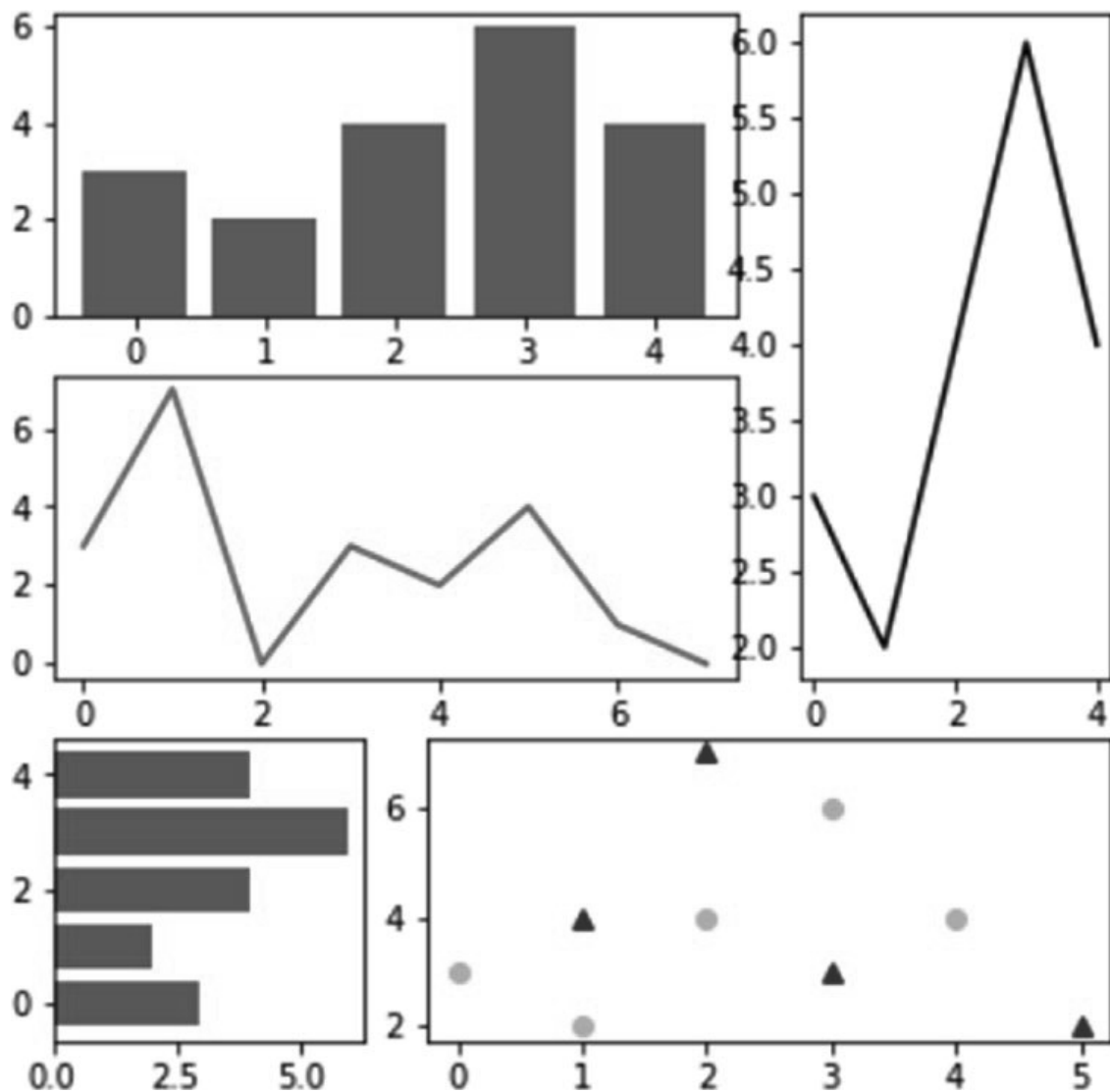


Figure 7-62. A grid of subplots can display many plots at the same time

Conclusions

In this chapter, you received all the fundamental aspects of the matplotlib library, and through a series of examples, you have learned about the basic tools for handling data visualization. You have become familiar with various examples of how to develop different types of charts with a few lines of code.

With this chapter, we conclude the part about the libraries that provides the basic tools to perform data analysis. In the next chapter, you will begin to treat topics most closely related to data analysis.