5QQMN534: Algorithmic Finance

Week4: Data Visualisation

Yves Hilpisch - Python for Finance 2nd Edition 2019: Chapter 7

Agenda

- Data Visualisation
- Static 2D Plotting
  - One Dimensional Data Sets
  - Two Dimensional Data Sets
  - Other plot styles
- Static 3D Plotting
- Interactive 2D Plotting
  - Basic Plots
  - Financial Plots (OHLC Candlesticks, Bollinger Bands, RSI)
- Conclusion

Data Visualization

Use a picture. It's worth a thousand words.
Arthur Brisbane (1911)

- This chapter is about the basic visualization capabilities of the `matplotlib`.
- Although there are more visualization packages available, `matplotlib` has established itself as the benchmark and, in many situations, a robust and reliable visualization tool.
- It is both easy to use for standard plots and flexible when it comes to more complex plots and customizations.
- In addition, it is tightly integrated with `NumPy` and `pandas` and the data structures they provide. `matplotlib` only allows for the generation of plots in the form of bitmaps (for example, in PNG or JPG format).
- On the other hand, modern web technologies — based, for example, on the Data-Driven Documents (D3.js) standard — allow for nice interactive and also embeddable plots (interactive, for example, in that one can zoom in to inspect certain areas in greater detail). A package that makes it convenient to create such D3.js plots with Python is `plotly`.
- A smaller additional library, called `Cufflinks`, tightly integrates `plotly` with `pandas DataFrame` objects and allows for the creation of popular financial plots (such as candlestick charts).

Data Visualization

- This chapter mainly covers the following topics:

"Static 2D Plotting"

This section introduces `matplotlib` and presents a selection of typical 2D plots, from the most simple to some more advanced ones with two scales or different subplots.

"Static 3D Plotting"

Based on `matplotlib`, a selection of 3D plots useful for certain financial applications are presented in this section.

"Interactive 2D Plotting"

This section introduces `plotly` and `Cufflinks` to create interactive 2D plots. Making use of the `QuantFigure` feature of `Cufflinks`, this section is also about typical financial plots used, for example, in technical stock analysis.

- This chapter cannot be comprehensive with regard to data visualization with Python, `matplotlib`, or `plotly`, but it provides a number of examples for the basic and important capabilities of these packages for finance.
- Other examples are also found in later chapters. For instance, Chapter 8 shows in more depth how to visualize financial time series data with the `pandas` library.

Static 2D Plotting

- Before creating the sample data and starting to plot, some imports and customizations:

```
In [1]: import matplotlib as mpl                    ❶

In [2]: mpl.__version__                             ❷
Out[2]: '3.0.0'

In [3]: import matplotlib.pyplot as plt             ❸

In [4]: plt.style.use('seaborn')                    ❹

In [5]: mpl.rcParams['font.family'] = 'serif'       ❺

In [6]: %matplotlib inline
```

❶ Imports `matplotlib` with the usual abbreviation `mpl`.

❷ The version of `matplotlib` used.

❸ Imports the main plotting (sub)package with the usual abbreviation `plt`.

❹ Sets the plotting style to `seaborn`.

❺ Sets the font to be `serif` in all plots.

One-Dimensional Data Sets1

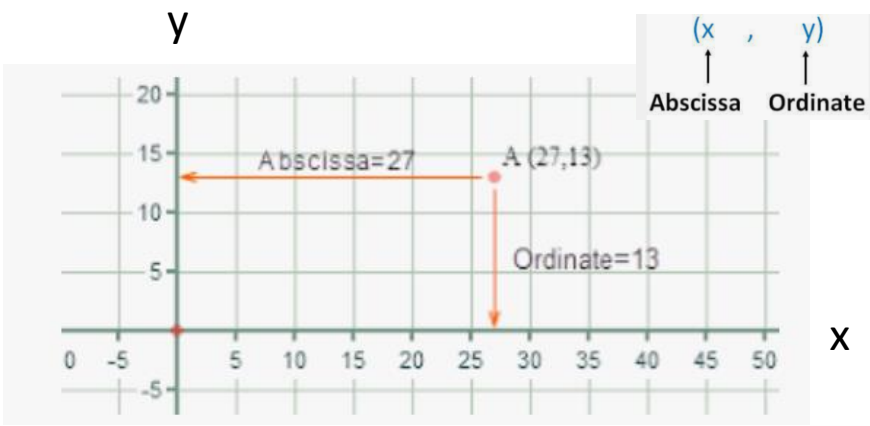
- The most fundamental, but nevertheless quite powerful, plotting function is `plt.plot()`.
- In principle, it needs two sets of numbers:

x values

A `list` or an array containing the *x* coordinates (values of the abscissa)

y values

A `list` or an array containing the *y* coordinates (values of the ordinate)



Abscissa: (in a system of coordinates) the distance from a point to the vertical or *y* -axis, measured parallel to the horizontal or *x* -axis; the *x* -coordinate.
Ordinate: a straight line from any point drawn parallel to one coordinate axis and meeting the other, especially a coordinate measured parallel to the vertical.

The number of *x* and *y* values provided must match, of course. Consider the following code, whose output is presented in Figure 7-1: (Next slide)

```
In [7]: import numpy as np

In [8]: np.random.seed(1000)          ❶

In [9]: y = np.random.standard_normal(20)   ❷

In [10]: x = np.arange(len(y))        ❸
         plt.plot(x, y);              ❹
```

❶ Fixes the seed for the random number generator for reproducibility.

❷ Draws the random numbers (*y* values).

❸ Fixes the integers (*x* values).

❹ Calls the `plt.plot()` function with the `x` and `y` objects.

One-Dimensional Data Sets2

Figure 7-1. Plot given x and y values

One-Dimensional Data Sets3

- `plt.plot()` notices when an `ndarray` object is passed. In this case, there is no need to provide the "extra" information of the *x* values.
- If one only provides the *y* values, `plt.plot()` takes the index values as the respective *x* values. Therefore, the following single line of code generates exactly the same output (see Figure 7-2):


```
In [11]: plt.plot(y);
```

Figure 7-2. Plot given data as an ndarray object

One-Dimensional Data Sets4

NUMPY ARRAYS AND MATPLOTLIB

You can simply pass `NumPy ndarray` objects to `matplotlib` functions. `matplotlib` is able to interpret the data structures for simplified plotting. However, be careful to not pass a too large and/or complex array.

- Since the majority of the `ndarray` methods return an `ndarray` object, one can also pass the object with a method (or even multiple methods, in some cases) attached.
- By calling the `cumsum()` method on the `ndarray` object with the sample data, one gets the cumulative sum of this data and, as to be expected, a different output (see Figure 7-3):

```
In [12]: plt.plot(y.cumsum());
```

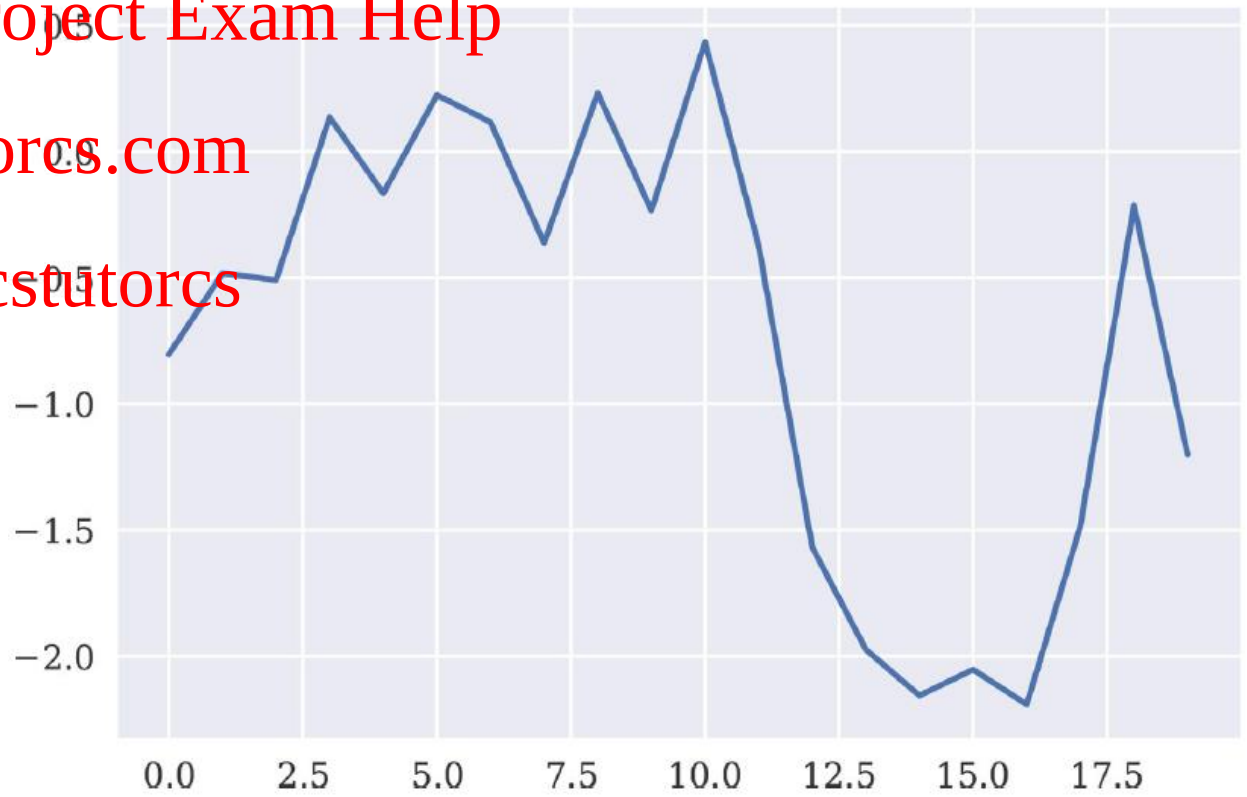


Figure 7-3. Plot given an ndarray object with a method attached

One-Dimensional Data Sets5

- In general, the default plotting style does not satisfy typical requirements for reports, publications, etc.
- For example, one might want to customize the font used (e.g., for compatibility with LaTeX fonts), to have labels at the axes, or to plot a grid for better readability.
- This is where plotting styles come into play.
- In addition, `matplotlib` offers a large number of functions to customize the plotting style. Some are easily accessible; for others one has to dig a bit deeper.
- Easily accessible, for example, are those functions that manipulate the axes and those that relate to grids and labels (see Figure 7-4):

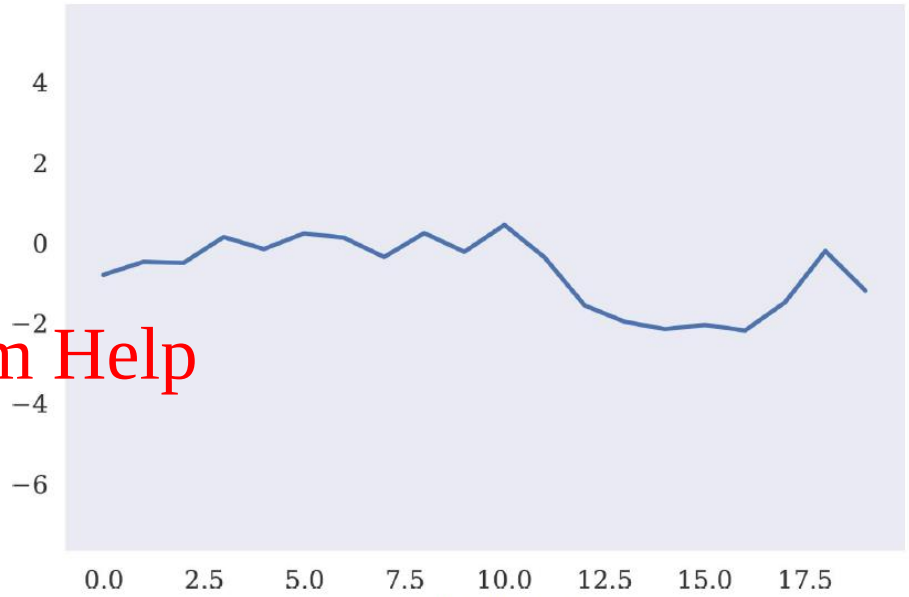


Figure 7-4. Plot without grid

```
In [13]: plt.plot(y.cumsum())
         plt.grid(False)    ❶
         plt.axis('equal'); ❷
```

❶ Turns off the grid.

❷ Leads to equal scaling for the two axes.

One-Dimensional Data Sets6

Other options for `plt.axis()` are given in Table 7-1, the majority of which have to be passed as a `str` object.

Table 7-1. Options for plt.axis()

Parameter	Description
Empty	Returns current axis limits
`off`	Turns axis lines and labels off
`equal`	Leads to equal scaling
`scaled`	Produces equal scaling via dimension changes
`tight`	Makes all data visible (tightens limits)
`image`	Makes all data visible (with data limits)
`[xmin, xmax, ymin, ymax]`	Sets limits to given (`list` of) values

option : *bool or str*

If a bool, turns axis lines and labels on or off. If a string, possible values are:

Value	Description
'on'	Turn on axis lines and labels. Same as `True`.
'off'	Turn off axis lines and labels. Same as `False`.
'equal'	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scaled'	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight'	Set limits just large enough to show all data, then disable further autoscaling.
'auto'	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'square'	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

One-Dimensional Data Sets7

- In addition, one can directly set the minimum and maximum values of each axis by using `plt.xlim()` and `plt.ylim()`.
- The following code provides an example whose output is shown in Figure 7-5:

```
In [14]: plt.plot(y.cumsum())
         plt.xlim(-1, 20)
         plt.ylim(np.min(y.cumsum()) - 1,
                  np.max(y.cumsum()) + 1);
```


Figure 7-5. Plot with custom axis limits

One-Dimensional Data Sets8

- For the sake of better readability, a plot usually contains a number of labels - e.g., a title and labels describing the nature of the *x* and *y* values.
- These are added by the functions `plt.title()`, `plt.xlabel()`, and `plt.ylabel()`, respectively.
- By default, `plot()` plots continuous lines, even if discrete data points are provided.
- The plotting of discrete points is accomplished by choosing a different style option.
- Figure 7-6 (next slide) overlays (red) points and a (blue) line with line width of 1.5 points:


```
In [15]: plt.figure(figsize=(10, 6))          ❶
         plt.plot(y.cumsum(), 'b', lw=1.5)    ❷
         plt.plot(y.cumsum(), 'ro')           ❸
         plt.xlabel('index')                  ❹
         plt.ylabel('value')                  ❺
         plt.title('A Simple Plot');          ❻
```

❶ Increases the size of the figure.

❷ Plots the data as a line in blue with line width of 1.5 points.

❸ Plots the data as red (thick) dots.

❹ Places a label on the x-axis.

❺ Places a label on the y-axis.

❻ Places a title.

One-Dimensional Data Sets9

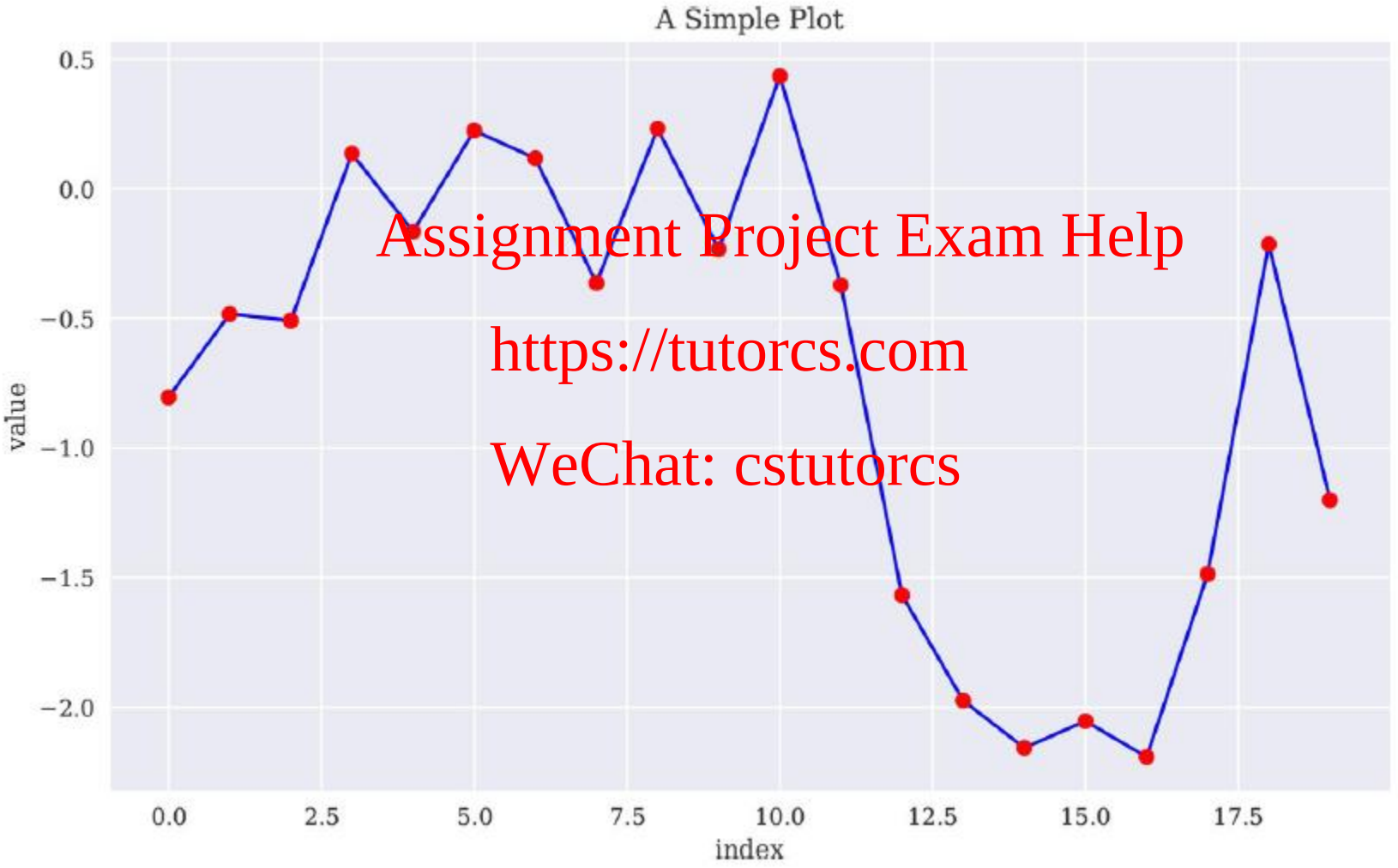


Figure 7-6. Plot with typical labels

One-Dimensional Data Sets10

- By default, `plt.plot()` supports the color abbreviations in Table 7-2.
- In terms of line and/or point styles, `plt.plot()` supports the characters listed in Table 7-3.

- Any color abbreviation can be combined with any style character.
- In this way, one can make sure that different data sets are easily distinguished.
- The plotting style is also reflected in the legend.

Table 7-2. Standard color abbreviations

Character	Color
b	Blue
g	Green
r	Red
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Table 7-3. Standard style characters

Character	Symbol
-	Solid line style
--	Dashed line style
-.	Dash-dot line style
:	Dotted line style
.	Point marker
,	Pixel marker
o	Circle marker
v	Triangle_down marker
^	Triangle_up marker
<	Triangle_left marker
>	Triangle_right marker
1	Tri_down marker
2	Tri_up marker
3	Tri_left marker
4	Tri_right marker
s	Square marker

Character	Symbol
h	Hexagon1 marker
H	Hexagon2 marker
+	Plus marker
x	X marker
D	Diamond marker
d	Thin diamond marker
|	Vline marker
_	Hline marker

Two-Dimensional Data Sets1

- Plotting one-dimensional data can be considered a special case.
- In general, data sets will consist of multiple separate subsets of data.
- The handling of such data sets follows the same rules with `matplotlib` as with one dimensional data.
- However, a number of additional issues might arise in such a context.
- For example, two data sets might have such a different scaling that they cannot be plotted using the same y- and/or x-axis scaling.
- Another issue might be that one might want to visualize two different data sets in different ways, e.g., one by a line plot and the other by a bar plot.
- The following code generates a two-dimensional sample data set as a `NumPy ndarray` object of shape with standard normally distributed pseudo-random numbers.
- On this array, the method `cumsum()` is called to calculate the cumulative sum of the sample data along axis 0 (i.e., the first dimension):


```
In [16]: y = np.random.standard_normal((20, 2)).cumsum(axis=0)
```

Two-Dimensional Data Sets2

- In general, one can also pass such two-dimensional arrays to `plt.plot()`.
- It will then automatically interpret the contained data as separate data sets (along axis 1, i.e., the second dimension). A respective plot is shown in Figure 7-7:

```
In [17]: plt.figure(figsize=(10, 6))
         plt.plot(y, lw=1.5)
         plt.plot(y, 'ro')
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('A Simple Plot');
```

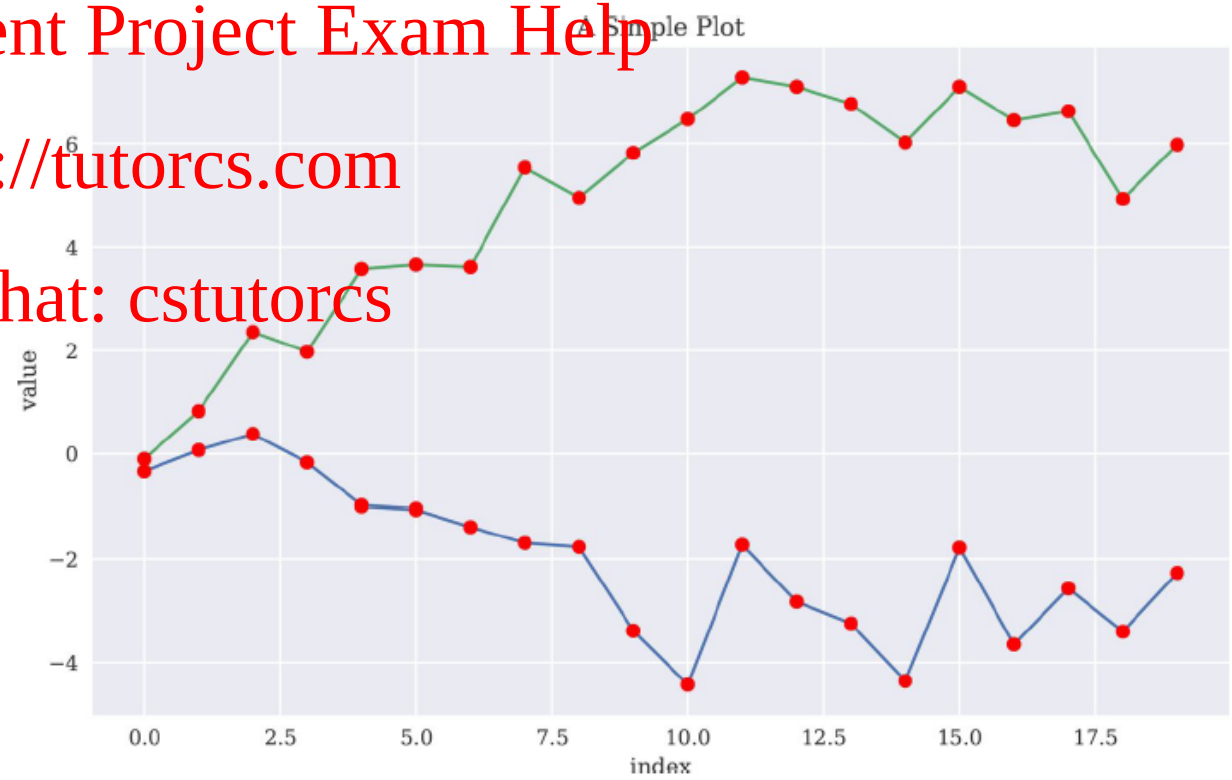



Figure 7-7. Plot with two data sets

Two-Dimensional Data Sets3

- In such a case, further annotations might be helpful to better read the plot.
- You can add individual labels to each data set and have them listed in the legend.
- The function `plt.legend()` accepts different locality parameters. `0` stands for *best location*, in the sense that as little data as possible is hidden by the legend.
- Figure 7-8 (next slide) shows the plot of the two data sets, this time with a legend.
- In the generating code, the `ndarray` object is not passed as a whole but the two data subsets (`y[:, 0]` and `y[:, 1]`) are accessed separately, which allows you to attach individual labels to them:


```
In [18]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], lw=1.5, label='1st')  ❶
         plt.plot(y[:, 1], lw=1.5, label='2nd')  ❶
         plt.plot(y, 'ro')
         plt.legend(loc=0)  ❷
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('A Simple Plot');
```

❶ Defines labels for the data subsets.

❷ Places a legend in the “best” location.

Two-Dimensional Data Sets4

- Further location options for `plt.legend()` include those presented in Table 7-4.

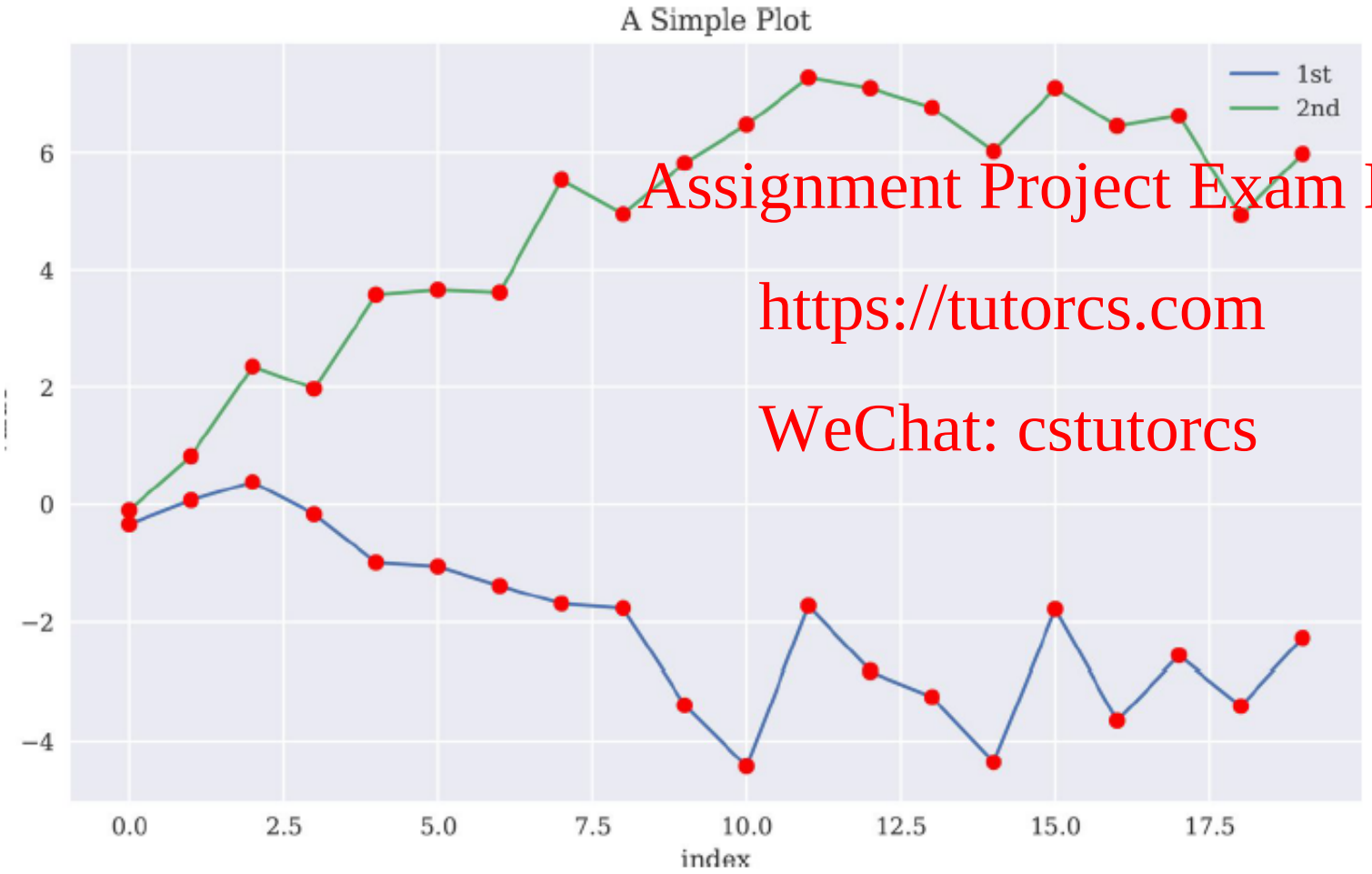


Figure 7-8. Plot with labeled data sets

Table 7-4. Options for plt.legend()

Loc	Description
Default	Upper right
0	Best possible
1	Upper right
2	Upper left
3	Lower left
4	Lower right
5	Right
6	Center left
7	Center right
8	Lower center
9	Upper center
10	Center

Two-Dimensional Data Sets5

- Multiple data sets with a similar scaling, like simulated paths for the same financial risk factor, can be plotted using a single y-axis.
- However, often data sets show rather different scalings and the plotting of such data with a single y-scale generally leads to a significant loss of visual information.
- To illustrate the effect, the following example scales the first of the two data subsets by a factor of 100 and plots the data again (see Figure 7-9) (next slide):


```
In [19]: y[:, 0] = y[:, 0] * 100  ❶

In [20]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 1], lw=1.5, label='2nd')
         plt.plot(y, 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('A Simple Plot');
```

❶ Rescales the first data subset.

Two-Dimensional Data Sets6

- Inspection of Figure 7-9 reveals that the first data set is still “visually readable,” while the second data set now looks like a straight line with the new scaling of the y-axis.
- In a sense, information about the second data set now gets “visually lost.”
- There are two basic approaches to resolve this problem through means of plotting, as opposed to adjusting the data (e.g., through rescaling):
  - Use of two y-axes (left/right)
  - Use of two subplots (upper/lower, left/right)

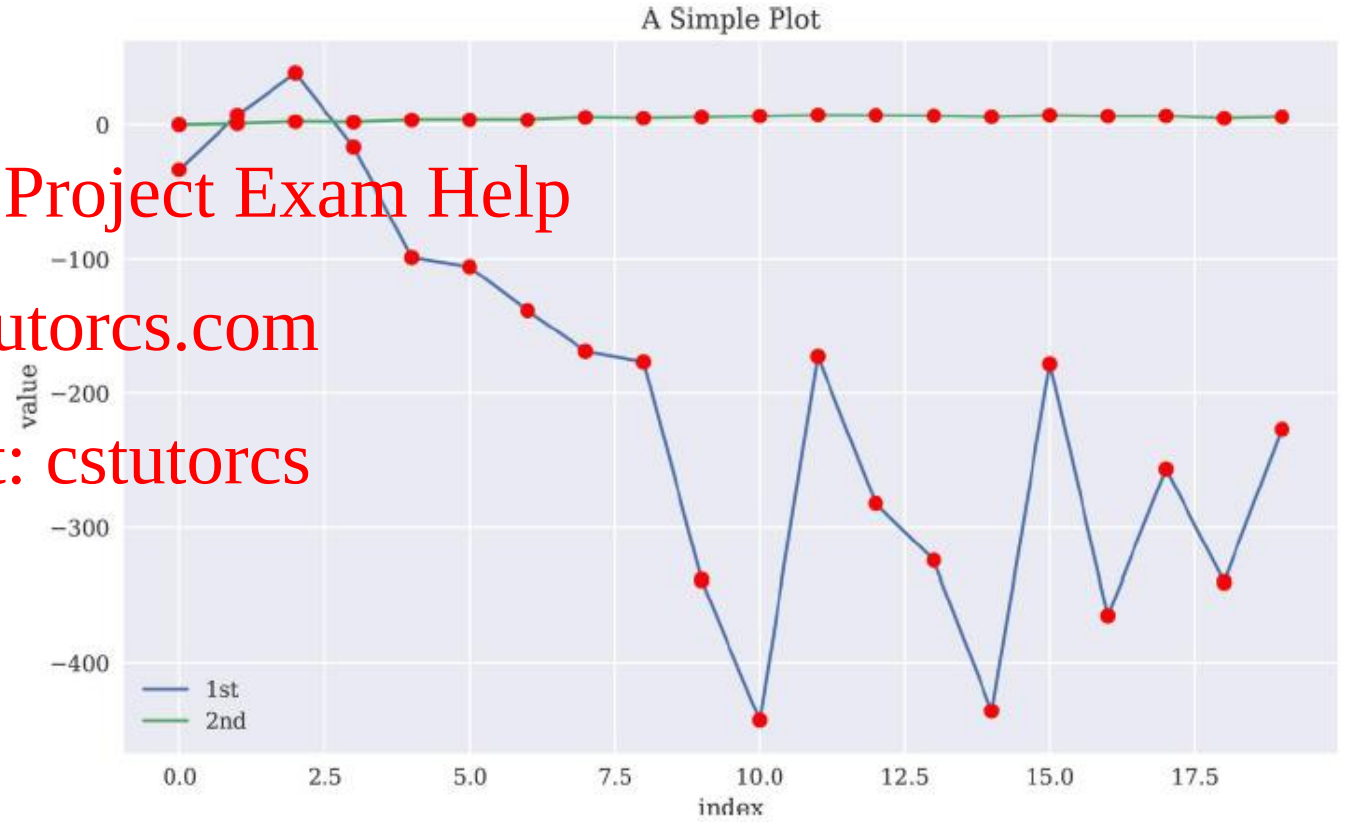


Figure 7-9. Plot with two differently scaled data sets

Two-Dimensional Data Sets7

- The following example introduces a second y-axis to the plot.
- Figure 7-10 (next slide) now has two different y-axes.
- The left y-axis is for the first data set while the right y-axis is for the second.
- Consequently, there are also two legends:


```
In [21]: fig, ax1 = plt.subplots()
         plt.plot(y[:, 0], 'b', lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=8)
         plt.xlabel('index')

         plt.ylabel('value 1st')
         plt.title('A Simple Plot')
         ax2 = ax1.twinx()
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value 2nd');
```

❶ Defines the figure and axis objects.

❷ Creates a second axis object that shares the x-axis.

The key lines of code are those that help manage the axes:

```
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
```

matplotlib.axes.Axes.twinx¶

`Axes.twinx()` [source]

Create a twin Axes sharing the xaxis.

Create a new Axes with an invisible x-axis and an independent y-axis positioned opposite to the original one (i.e. at right). The x-axis autoscale setting will be inherited from the original Axes. To ensure that the tick marks of both y-axes align, see `LinearLocator`.

Returns: **Axes**

The newly created Axes instance

matplotlib.pyplot.subplots

`matplotlib.pyplot.subplots(nrows=1, ncols=1, *, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)` [source]

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Two-Dimensional Data Sets8

- By using the `plt.subplots()` function, one gets direct access to the underlying plotting objects (the figure, subplots, etc.).
- It allows one, for example, to generate a second subplot that shares the x-axis with the first subplot.
- In Figure 7-10 (), then, the two subplots actually *overlay* each other.

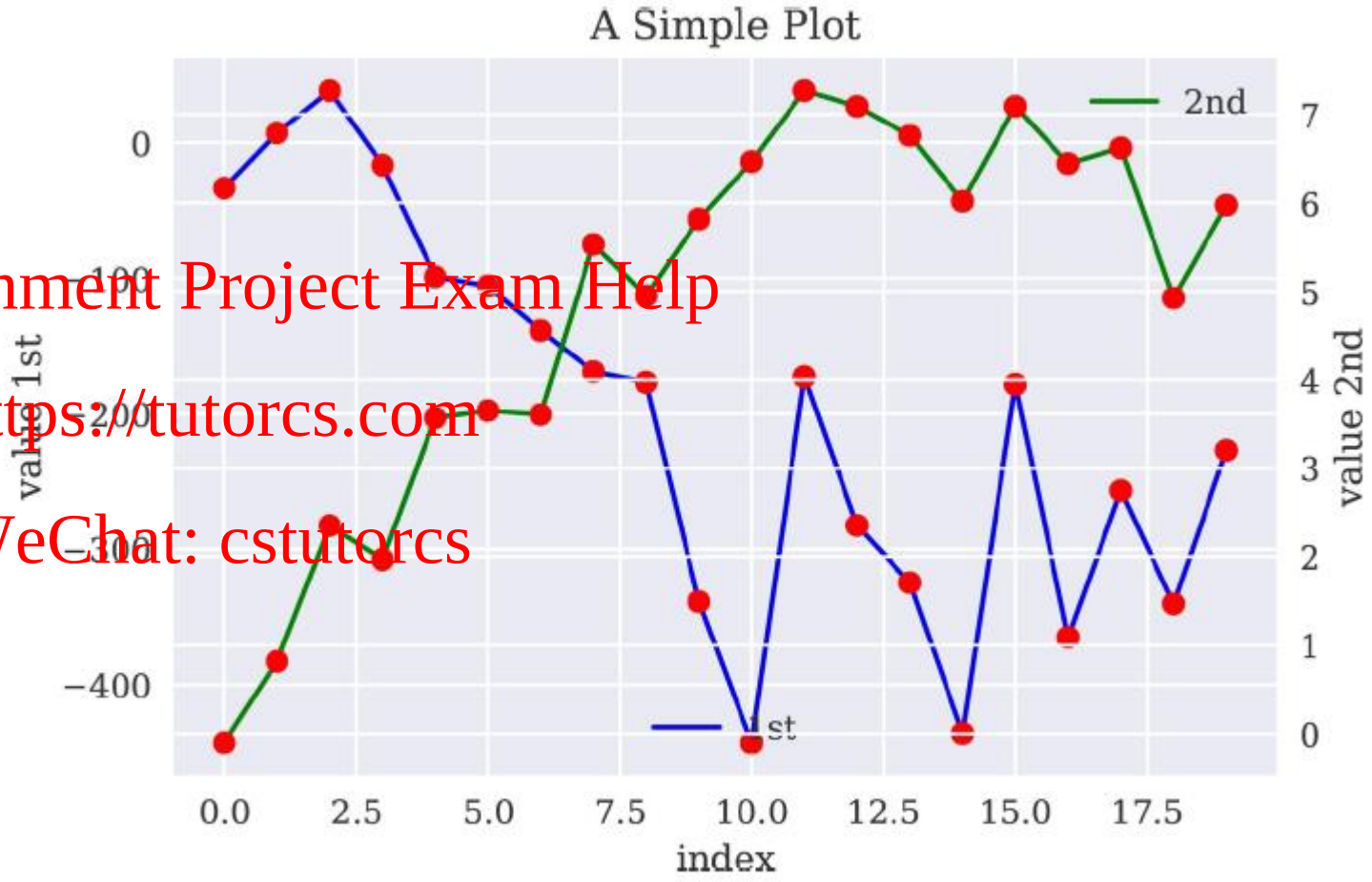


Figure 7-10. Plot with two data sets and two y-axes

Two-Dimensional Data Sets9

- Next, consider the case of two *separate* subplots.
- This option gives even more freedom to handle the two data sets, as Figure 7-11 (next slide) illustrates:


```
In [22]: plt.figure(figsize=(10, 6))
         plt.subplot(211)  ❶
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.ylabel('value')
         plt.title('A Simple Plot')
         plt.subplot(212)  ❷
         plt.plot(y[:, 1], 'g', lw=1.5, label='2nd')
         plt.plot(y[:, 1], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value');
```

❶ Defines the upper subplot 1.

❷ Defines the lower subplot 2.

Two-Dimensional Data Sets10

- The placing of subplots in a `matplotlib` figure object is accomplished by the use of a special coordinate system.
- `plt.subplot()` takes as arguments three integers for `numrows`, `numcols`, and `fignum` (either separated by commas or not). `numrows` specifies the number of *rows*, `numcols` the number of *columns*, and `fignum` the number of the *subplot*, starting with 1 and ending with `numrows * numcols`.
- For example, a figure with nine equally sized subplots would have `numrows=3`, `numcols=3`, and `fignum=1,2,...,9`.
- The lower-right subplot would have the following "coordinates": `plt.subplot(3, 3, 9)`.
- Sometimes, it might be necessary or desired to choose two different plot types to visualize such data.
- With the subplot approach one has the freedom to combine arbitrary kinds of plots that `matplotlib` offers

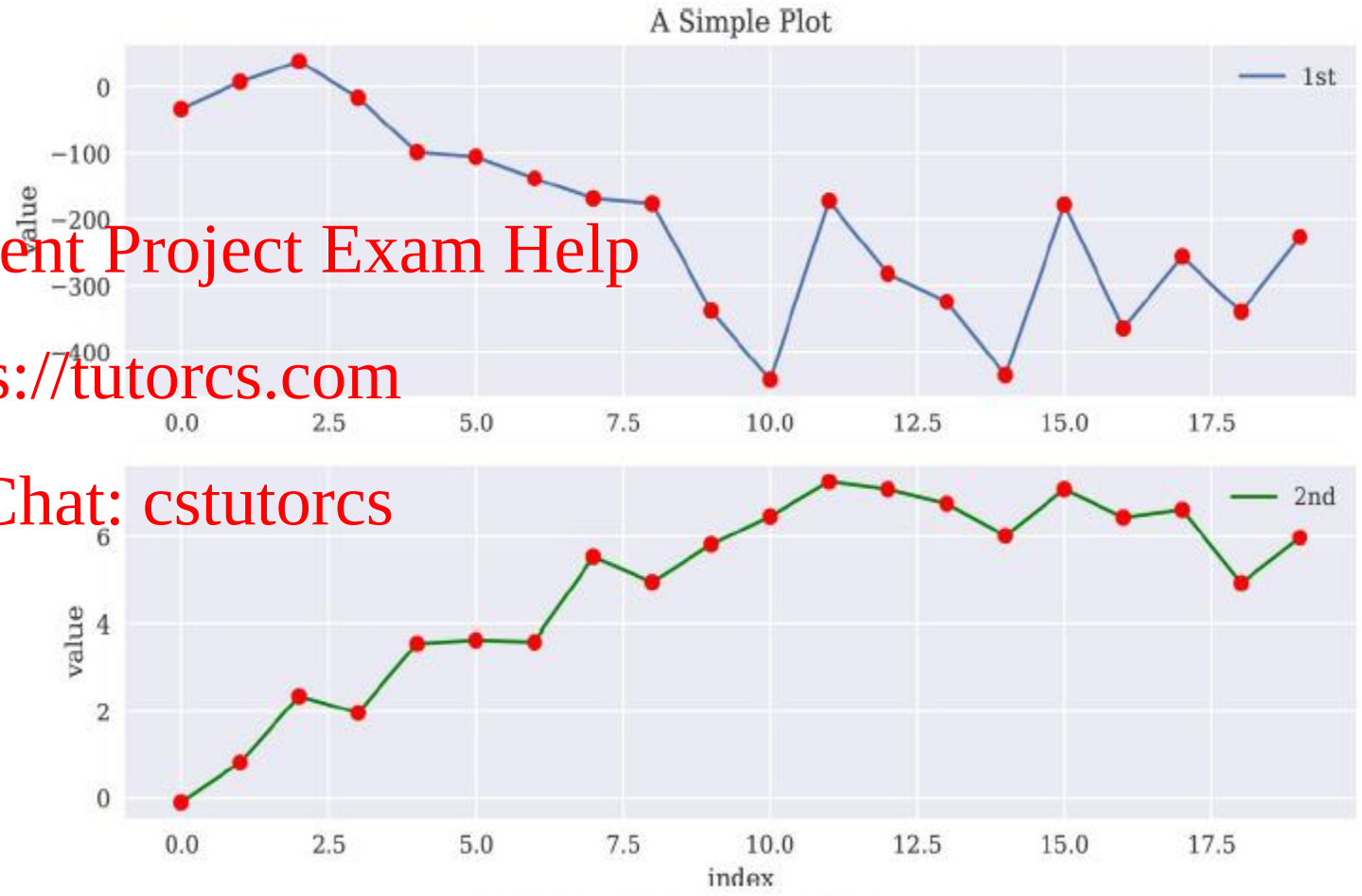


Figure 7-11. Plot with two subplots

Matplotlib Gallery: https://matplotlib.org/2.0.2/gallery.html

Two-Dimensional Data Sets11

- Figure 7-12 combines a line/point plot with a bar chart:

```
In [23]: plt.figure(figsize=(10, 6))
         plt.subplot(121)
         plt.plot(y[:, 0], lw=1.5, label='1st')
         plt.plot(y[:, 0], 'ro')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.ylabel('value')
         plt.title('1st Data Set')
         plt.subplot(122)
         plt.bar(np.arange(len(y)), y[:, 1], width=0.5,
                 color='g', label='2nd')
         plt.legend(loc=0)
         plt.xlabel('index')
         plt.title('2nd Data Set');
```

❶

Creates a `bar` subplot.

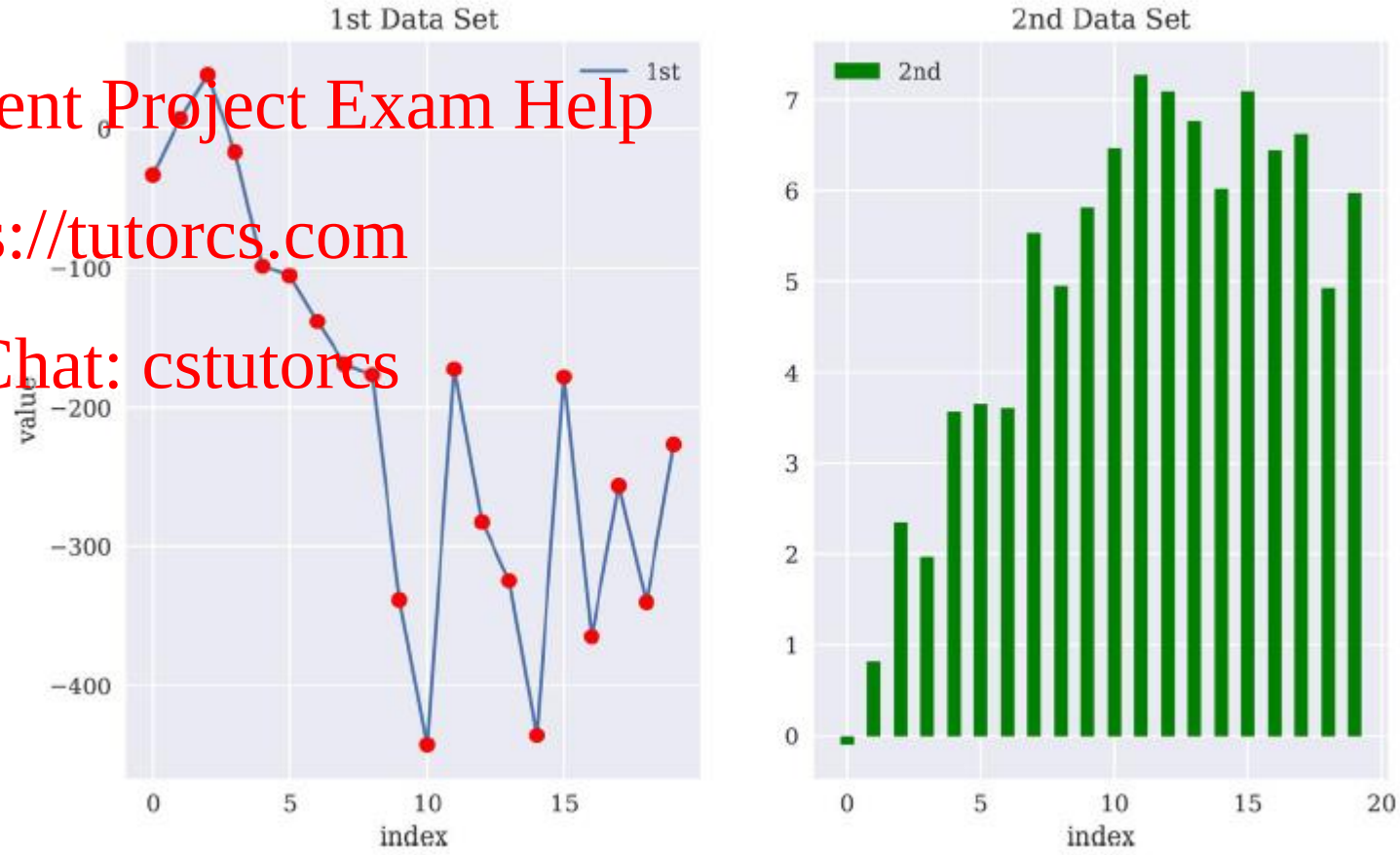


Figure 7-12. Plot combining line/point subplot with bar subplot

Other Plot Styles1: Introduction

- When it comes to two-dimensional plotting, line and point plots are probably the most important ones in finance; this is because many data sets embody time series data, which generally is visualized by such plots.
- Chapter 8 addresses financial time series data in detail.
- However, this section sticks with a two-dimensional data set of random numbers and illustrates some alternative, and for financial applications useful, visualization approaches.
- The first is the *scatter plot*, where the values of one data set serve as the *x* values for the other data set.

Other Plot Styles2: Scatterplot

- Figure 7-13 shows such a plot.
- This plot type might be used, for example, for plotting the returns of one financial time series against those of another one.
- This example uses a new two dimensional data set with some more data:

```
In [24]: y = np.random.standard_normal((1000, 2))

In [25]: plt.figure(figsize=(10, 6))
         plt.plot(y[:, 0], y[:, 1], 'ro')
         plt.xlabel('1st')
         plt.ylabel('2nd')
         plt.title('Scatter Plot');
```

❶ Creates a larger data set with random numbers.

❷ Scatter plot produced via the `plt.plot()` function.

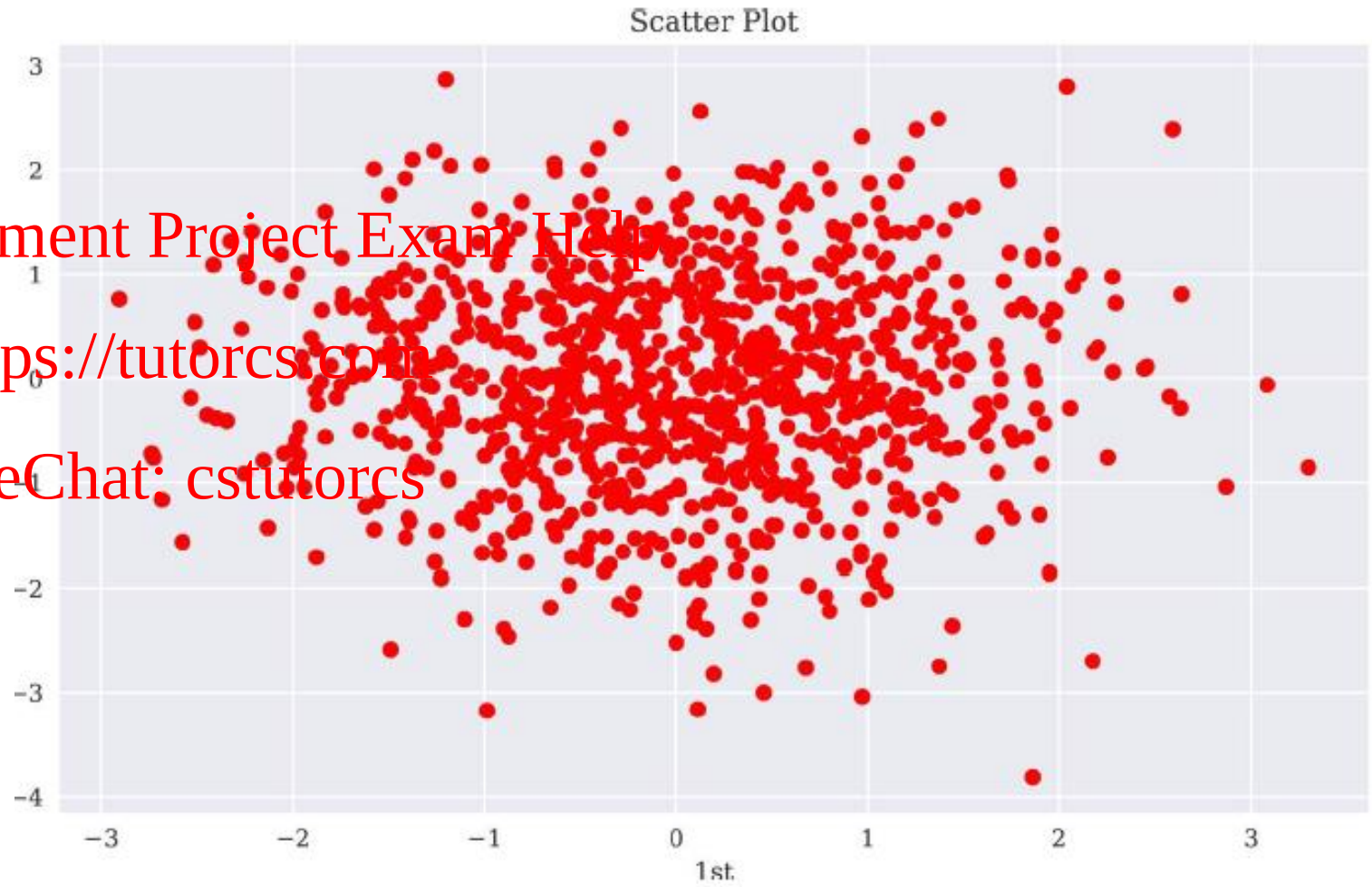


Figure 7-13. Scatter plot via plt.plot() function

Other Plot Styles3: Scatterplot

- `matplotlib` also provides a specific function to generate scatter plots. It
- basically works in the same way, but provides some additional features.
- Figure 7-14 shows the corresponding scatter plot to Figure 7-13, this time
- generated using the `plt.scatter()` function:


```
In [26]: plt.figure(figsize=(10, 6))
         plt.scatter(y[:, 0], y[:, 1], marker='o')
         plt.xlabel('1st')
         plt.ylabel('2nd')
         plt.title('Scatter Plot');
```

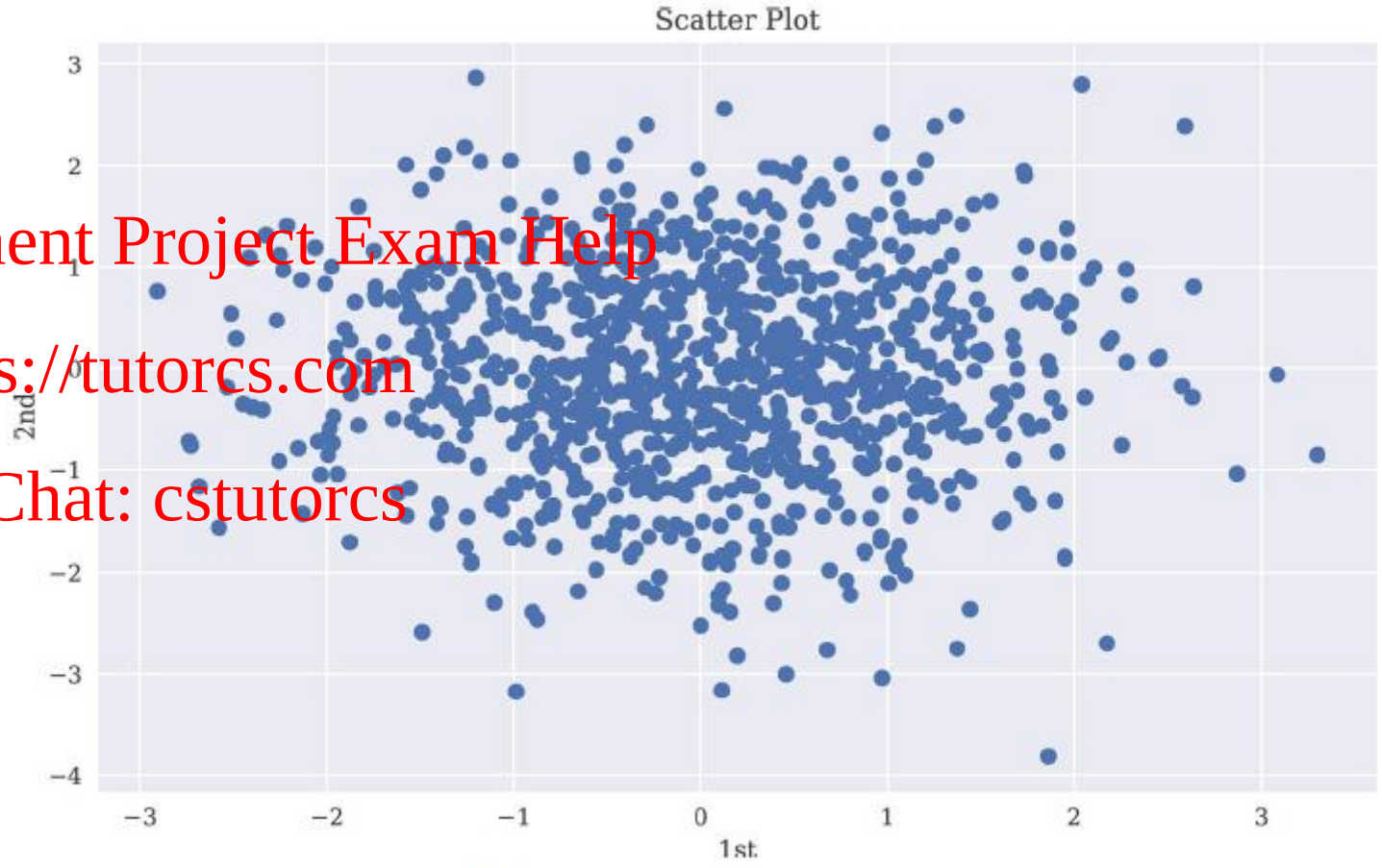


Figure 7-14. Scatter plot via plt.scatter() function

❶ Scatter plot produced via the `plt.scatter()` function.

Other Plot Styles4: Scatterplot with colour bar

- Among other things, the `plt.scatter()` plotting function allows the addition of a third dimension, which can be visualized through different colors and be described by the use of a color bar.
- Figure 7-15 shows a scatter plot where there is a third dimension illustrated by different colors of the single dots and with a color bar as a legend for the colors.
- To this end, the following code generates a third data set with random data, this time consisting of integers between 0 and 10:


```
In [27]: c = np.random.randint(0, 10, len(y))

In [28]: plt.figure(figsize=(10, 6))
         plt.scatter(y[:, 0], y[:, 1],
                     c=c,      ❶
                     cmap='coolwarm',  ❷
                     marker='o')  ❸
         plt.colorbar()
         plt.xlabel('1st')
         plt.ylabel('2nd')
         plt.title('Scatter Plot');
```

❶ Includes the third data set.

❷ Chooses the color map.

❸ Defines the marker to be a thick dot.

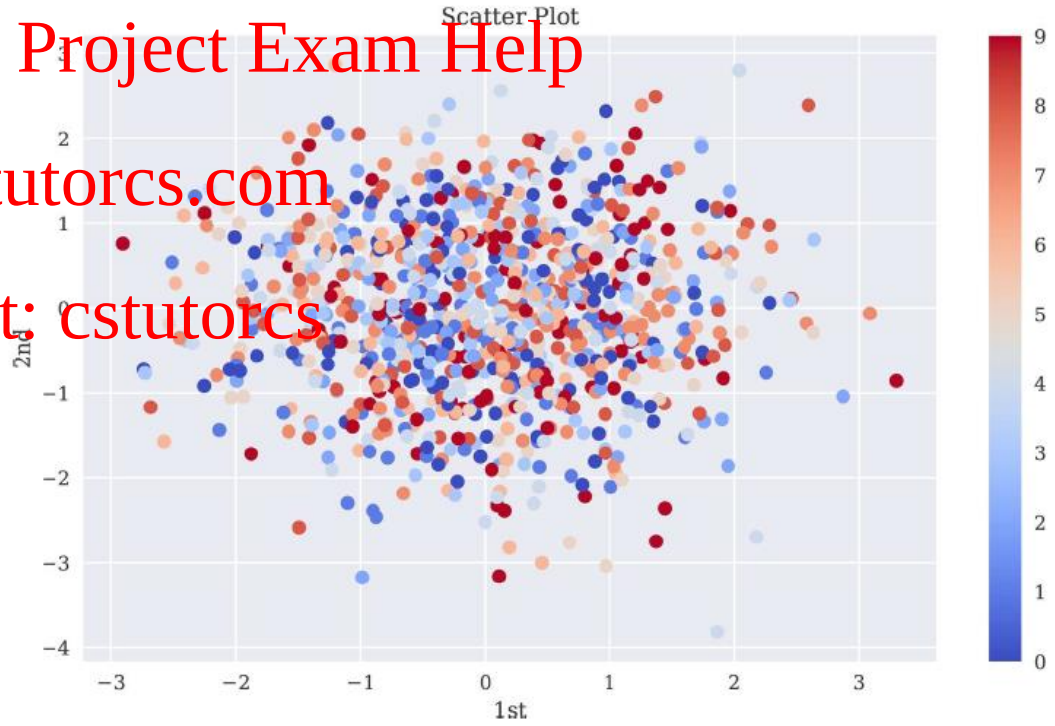


Figure 7-15. Scatter plot with third dimension

Names of colormaps:
https://www.kite.com/python/docs/matplotlib.pyplot.colormaps

Other Plot Styles5: Histogram

- Another type of plot, the *histogram*, is also often used in the context of financial returns.
- Figure 7-16 puts the frequency values of the two data sets next to each other in the same plot:


```
In [29]: plt.figure(figsize=(10, 6))
         plt.hist(y, label=['1st', '2nd'], bins=25)  ❶
         plt.legend(loc=0)
         plt.xlabel('value')
         plt.ylabel('frequency')
         plt.title('Histogram');
```

Figure 7-16. Histogram for two data sets

❶ Histogram plot produced via the `plt.hist()` function.

Other Plot Styles6: Histogram parameters

- Since the histogram is such an important plot type for financial applications,
- let's take a closer look at the use of `plt.hist()`.
- The following example illustrates the parameters that are supported:


```
plt.hist(x, bins=10, range=None, normed=False, weights=None, cumulative=False,
bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None,
log=False, color=None, label=None, stacked=False, hold=None, **kwargs)
```

Table 7-5 provides a description of the main parameters of the `plt.hist()` function.

Table 7-5. Parameters for plt.hist()

Parameter	Description
`x`	`list` object(s), `ndarray` object
`bins`	Number of bins
`range`	Lower and upper range of bins
`normed`	Norming such that integral value is 1
`weights`	Weights for every value in `x`
`cumulative`	Every bin contains the counts of the lower bins
`histtype`	Options (strings): `bar`, `barstacked`, `step`, `stepfilled`
`align`	Options (strings): `left`, `mid`, `right`
`orientation`	Options (strings): `horizontal`, `vertical`
`rwidth`	Relative width of the bars
`log`	Log scale
`color`	Color per data set (array-like)
`label`	String or sequence of strings for labels
`stacked`	Stacks multiple data sets

Other Plot Styles7:Stacked histogram

- Figure 7-17 shows a similar plot
- This time, the data of the two data sets is stacked in the histogram:

```
In [30]: plt.figure(figsize=(10, 6))
         plt.hist(y, label=['1st', '2nd'], color=['b', 'g'],
                  stacked=True, bins=20, alpha=0.5)
         plt.legend(loc=0)
         plt.xlabel('value')
         plt.ylabel('frequency')
         plt.title('Histogram');
```

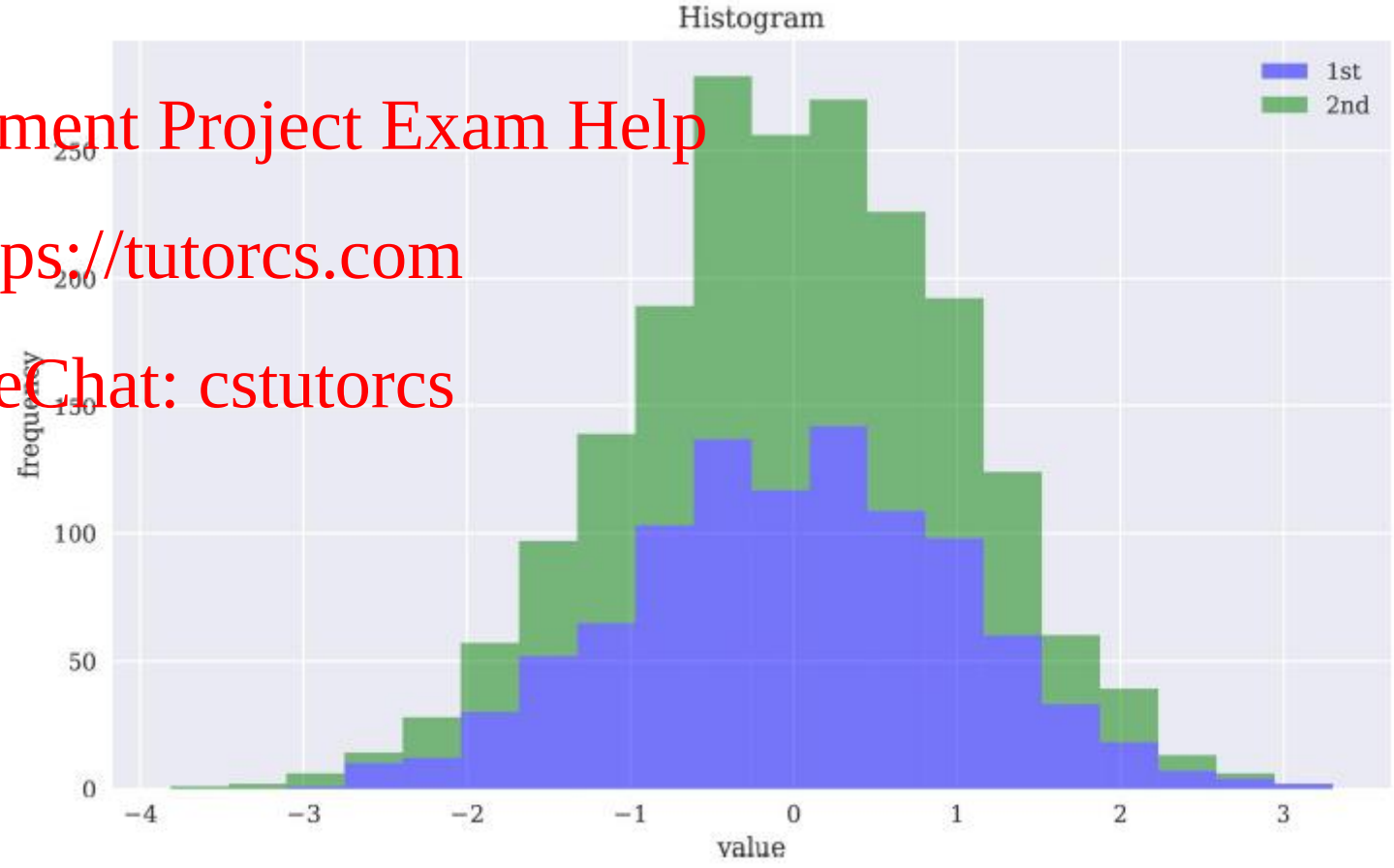



Figure 7-17. Stacked histogram for two data sets

Other Plot Styles8: Boxplot

- Another useful plot type is the *boxplot*.
- Similar to the histogram, the boxplot allows both a concise overview of the characteristics of a data set and easy comparison of multiple data sets.
- Figure 7-18 shows such a plot for our data sets:


```
In [31]: fig, ax = plt.subplots(figsize=(10, 6))
         plt.boxplot(y)                          ❶
         plt.setp(ax, xticklabels=['1st', '2nd'])  ❷
         plt.xlabel('data set')
         plt.ylabel('value')
         plt.title('Boxplot');
```

❶ Boxplot produced via the `plt.boxplot()` function.

❷ Sets individual x labels.

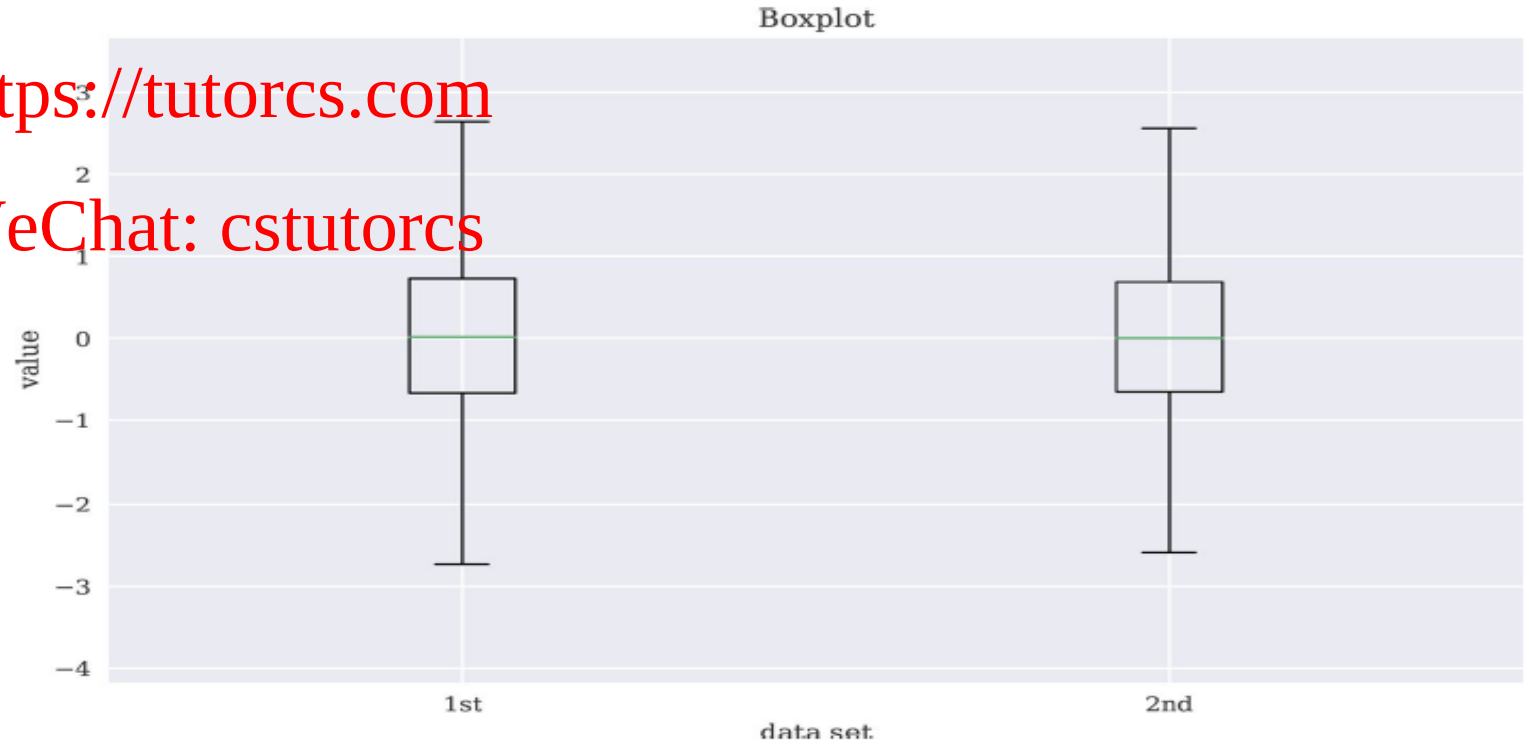


Figure 7-18. Boxplot for two data sets

Demo:
https://matplotlib.org/stable/gallery/pyplots/boxplot_demo_pyplot.html

Other Plot Styles8: Interpreting Boxplot

This last example uses the function `plt.setp()`, which sets properties for a (set of) plotting instance(s).
For example, consider a line plot generated by:

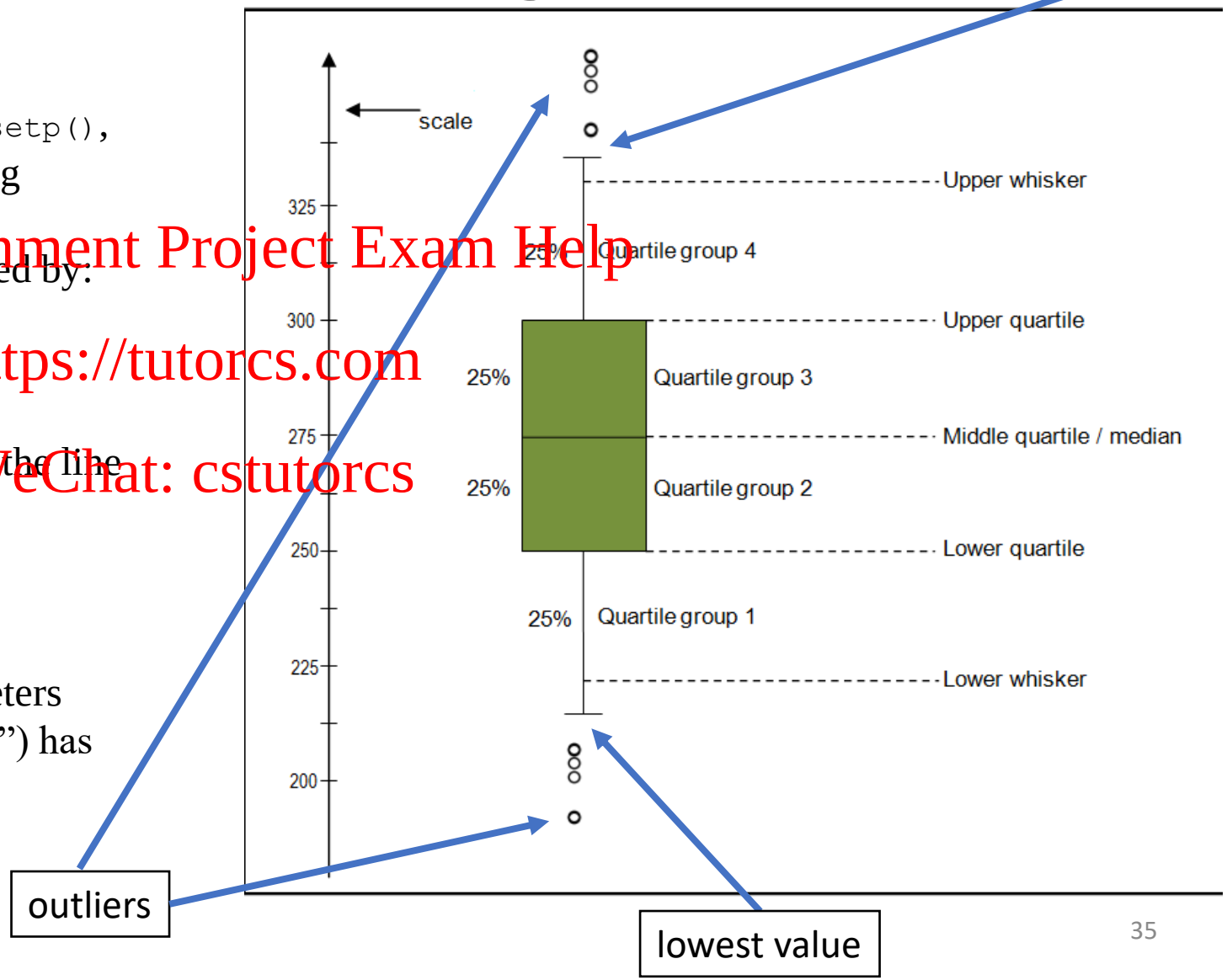
```
line = plt.plot(data, 'r')
```

The following code changes the style of the line to "dashed":

```
plt.setp(line, linestyle='--')
```

This way, one can easily change parameters after the plotting instance ("artist object") has been generated.

Parameter (showfliers=False) is to not show outliers



Static 3D Plotting1

- There are not too many fields in finance that really benefit from visualization in three dimensions.
- However, one application area is volatility surfaces showing implied volatilities simultaneously for a number of times-to maturity and strikes of the traded options used.
- Appendix B shows an example of value and vega surfaces being visualized for a European call option.
- In what follows, the code artificially generates a plot that resembles a volatility surface. To this end, consider the parameters:
  - *Strike values* between 50 and 150
  - *Times-to-maturity* between 0.5 and 2.5 years
- This provides a two-dimensional coordinate system.
- The `NumPy np.meshgrid()` function can generate such a system out of two one dimensional `ndarray` objects:

```
In [34]: strike = np.linspace(50, 150, 24)  ❶

In [35]: ttm = np.linspace(0.5, 2.5, 24)  ❷

In [36]: strike, ttm = np.meshgrid(strike, ttm)  ❸

In [37]: strike[:2].round(1)  ❸
Out[37]: array([[ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,  84.8,
                  89.1,  93.5,  97.8, 102.2, 106.5, 110.9, 115.2, 119.6, 123.9,
                 128.3, 132.6, 137. , 141.3, 145.7, 150. ],
                [ 50. ,  54.3,  58.7,  63. ,  67.4,  71.7,  76.1,  80.4,  84.8,
                  89.1,  93.5,  97.8, 102.2, 106.5, 110.9, 115.2, 119.6, 123.9,
                 128.3, 132.6, 137. , 141.3, 145.7, 150. ]])

In [38]: iv = (strike - 100) ** 2 / (100 * strike) / ttm  ❹

In [39]: iv[:5, :3]  ❹
Out[39]: array([[1.        , 0.76695652, 0.58132045],
                [0.65165185, 0.65333333, 0.4951989 ],
                [0.74193548, 0.56903226, 0.43130227],
                [0.65714286, 0.504     , 0.38201058],
                [0.59724359, 0.45230769, 0.34283001]])
```


❶ The `ndarray` object with the strike values.

❷ The `ndarray` object with the time-to-maturity values.

❸ The two two-dimensional `ndarray` objects (grids) created.

❹ The dummy implied volatility values.

Static 3D Plotting2: 3D Surface Plot Example1

The plot resulting from the following code is shown in Figure 7-20:

```
In [40]: from mpl_toolkits.mplot3d import Axes3D ❶
         fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d') ❷
         surf = ax.plot_surface(strike, ttm, iv, rstride=2, cstride=2,
                                cmap=plt.cm.coolwarm, linewidth=0.5,
                                antialiased=True) ❸
         ax.set_xlabel('strike') ❹
         ax.set_ylabel('time-to-maturity') ❺
         ax.set_zlabel('implied volatility') ❻
         fig.colorbar(surf, shrink=0.5, aspect=5); ❼
```

❶ Imports the relevant 3D plotting features, which is required although `Axes3D` is not directly used.

❷ Sets up a canvas for 3D plotting.

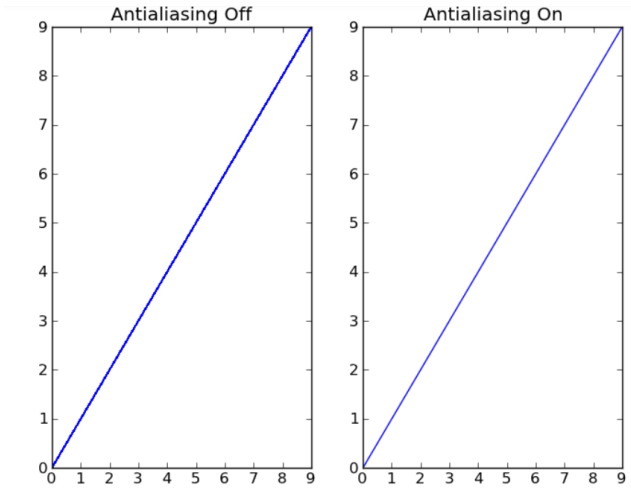
❸ Creates the 3D plot.

❹ Sets the x-axis label.

❺ Sets the y-axis label.

❻ Sets the z-axis label.

❼ Creates a color bar.



Note

To maximize rendering speed consider setting *rstride* and *cstride* to divisors of the number of rows minus 1 and columns minus 1 respectively. For example, given 51 rows rstride can be any of the divisors of 50.

Similarly, a setting of *rstride* and *cstride* equal to 1 (or *rcount* and *ccount* equal the number of rows and columns) can use the optimized path.

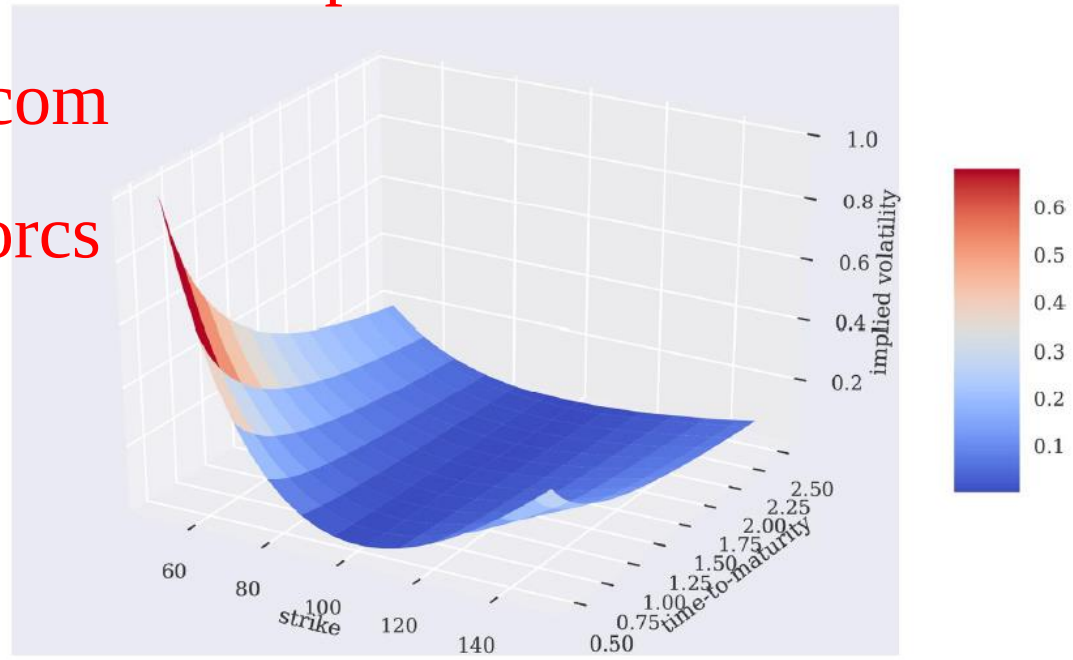
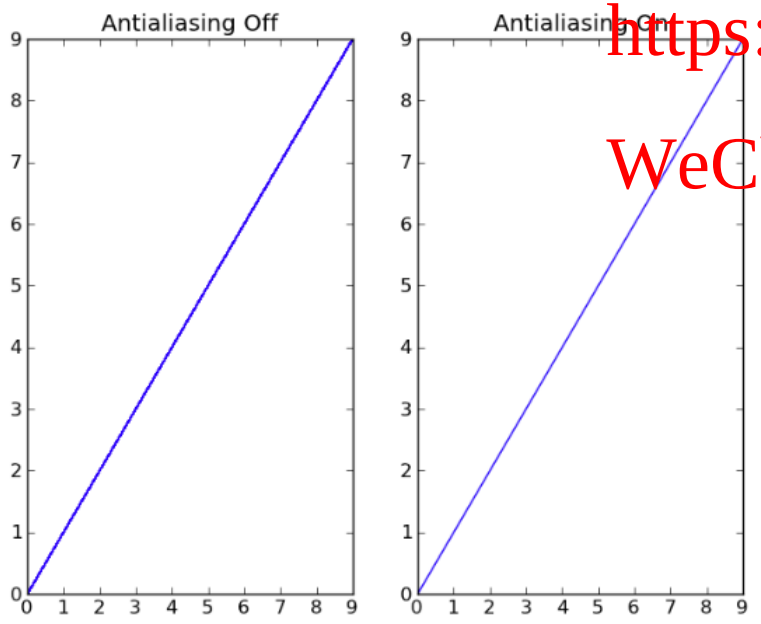


Figure 7-20. 3D surface plot for (dummy) implied volatilities

Non-antialiased plotting will be faster, so if you're plotting a large amount of data, it can be worthwhile to turn it off

Static 3D Plotting3: plot_surface() parameters

- Table 7-6 provides a description of the different parameters the `plt.plot_surface()` function can take.



Non-antialiased plotting will be faster, so if you're plotting a large amount of data, it can be worthwhile to turn it off.

Table 7-6. Parameters for plot_surface()

Parameter	Description
`X, Y, Z`	Data values as 2D arrays
`rstride`	Array row stride (step size)
`cstride`	Array column stride (step size)
`color`	Color of the surface patches
`cmap`	Color map for the surface patches
`facecolors`	Face colors for the individual patches
`norm`	Instance of `Normalize` to map values to colors
`vmin`	Minimum value to map
`vmax`	Maximum value to map
`shade`	Whether to shade the face colors

Static 3D Plotting4 3D Surface Plot Exampl2

- As with two-dimensional plots, the line style can be replaced by single points or, as in what follows, single triangles.
- Figure 7-21 plots the same data as a 3D scatter plot but now also with a different viewing angle, using the `view_init()` method to set it:

```
In [41]: fig = plt.figure(figsize=(10, 6))
         ax = fig.add_subplot(111, projection='3d')
         ax.view_init(30, 60)  ❶
         ax.scatter(strike, ttm, iv, zdir='z', s=25,
                    c='b', marker='^')  ❷
         ax.set_xlabel('strike')
         ax.set_ylabel('time-to-maturity')
         ax.set_zlabel('implied volatility');
```


❶ Sets the viewing angle.

❷ Creates a 3D scatter plot.

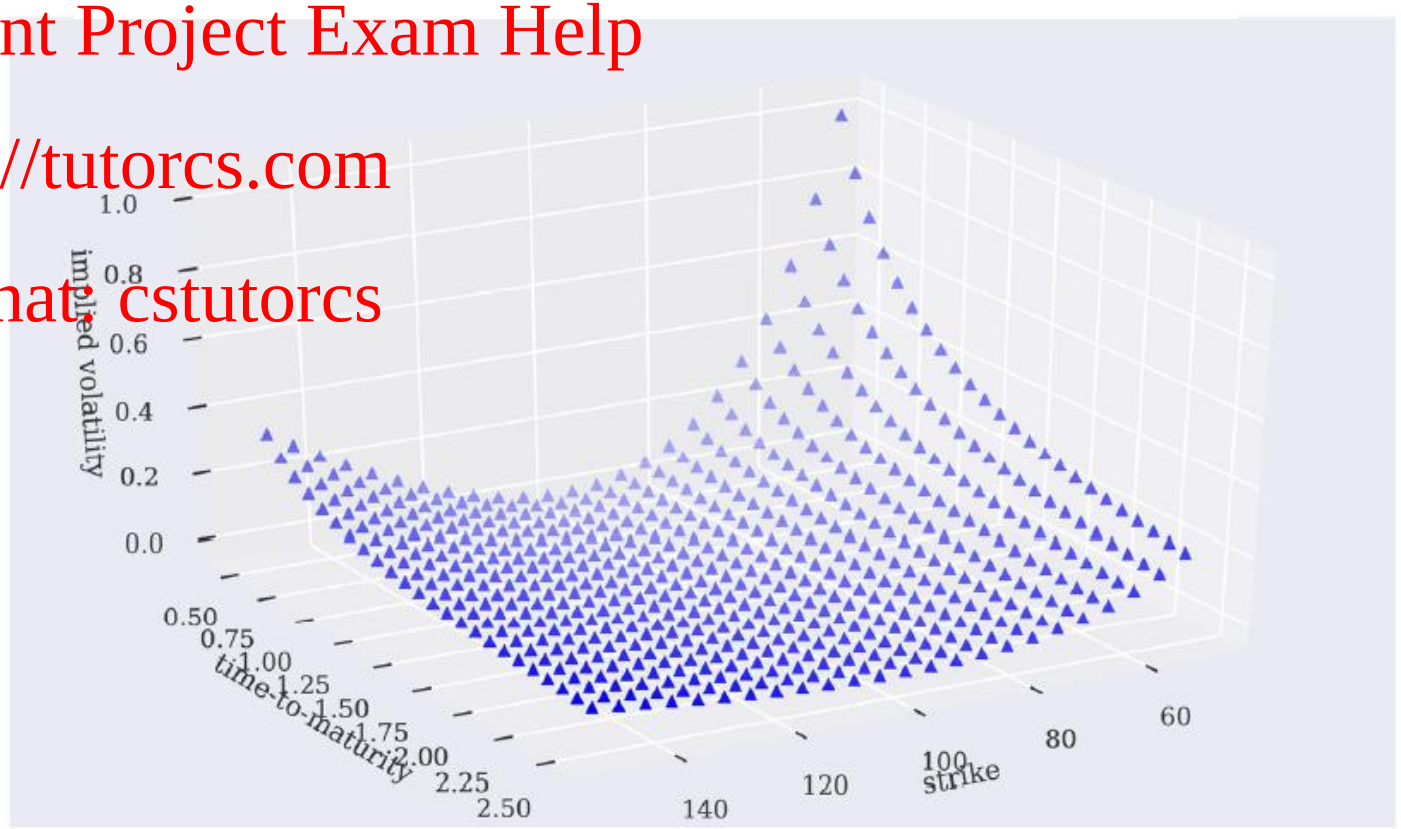


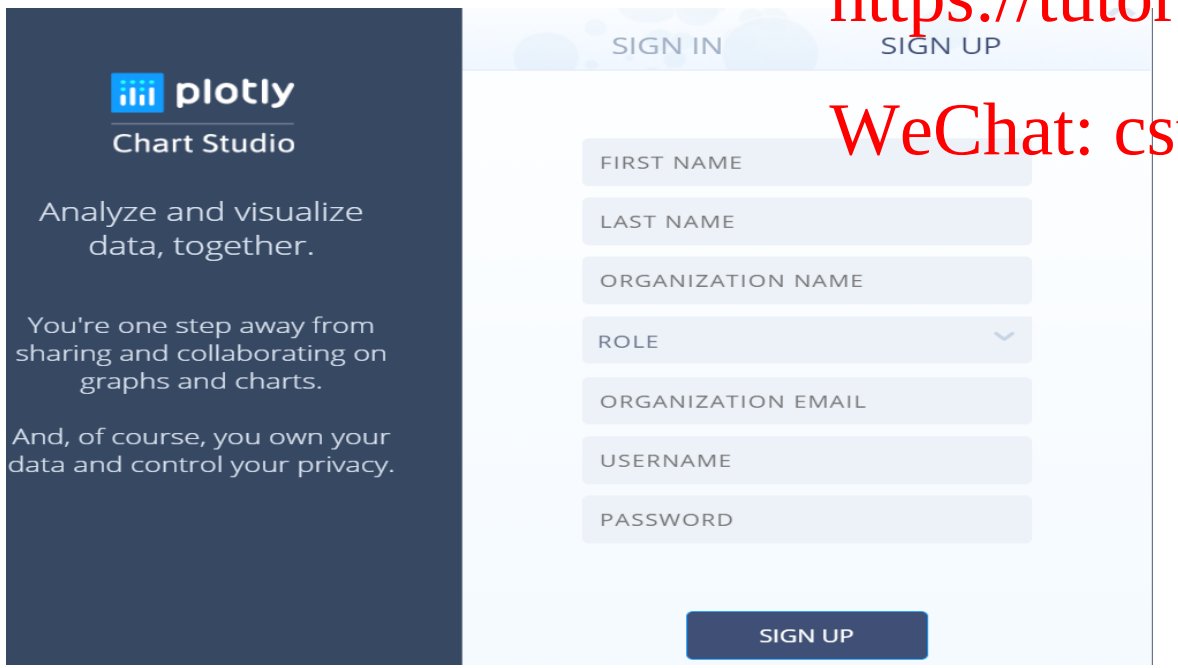
Figure 7-21. 3D scatter plot for (dummy) implied volatilities

Interactive 2D Plotting

- `matplotlib` allows you to create plots that are static bitmap objects or of PDF format. Nowadays, there are many libraries available to create interactive plots based on the D3.js standard. Such plots enable zooming in and out, hover effects for data inspection, and more. They can in general also be easily embedded in web pages.
- A popular platform and plotting library is `plotly`. It is dedicated to visualization for data science and is in widespread use in the data science community. Major benefits of `plotly` are its tight integration with the Python ecosystem and the ease of use — in particular when combined with `pandas DataFrame` objects and the wrapper package `Cufflinks`.
- For some functionality, a free account is required. Once the credentials are granted they should be stored locally for permanent use. For details, see the "Getting Started with Plotly for Python" guide.
- This section focuses on selected aspects only, in that `Cufflinks` is used exclusively to create interactive plots from data stored in `DataFrame` objects.

Register for Plotly

- **Free Account here:** https://chart-studio.plotly.com/Auth/login/#/
- This allows you more functionality.
- Enter Kings College University on Organisation Name and Student as Role.
- Enter your university email.
- Confirm the account creation on your email activation link.



Python Getting started with Plotly: manualhttps://plotly.com/python/getting-started/

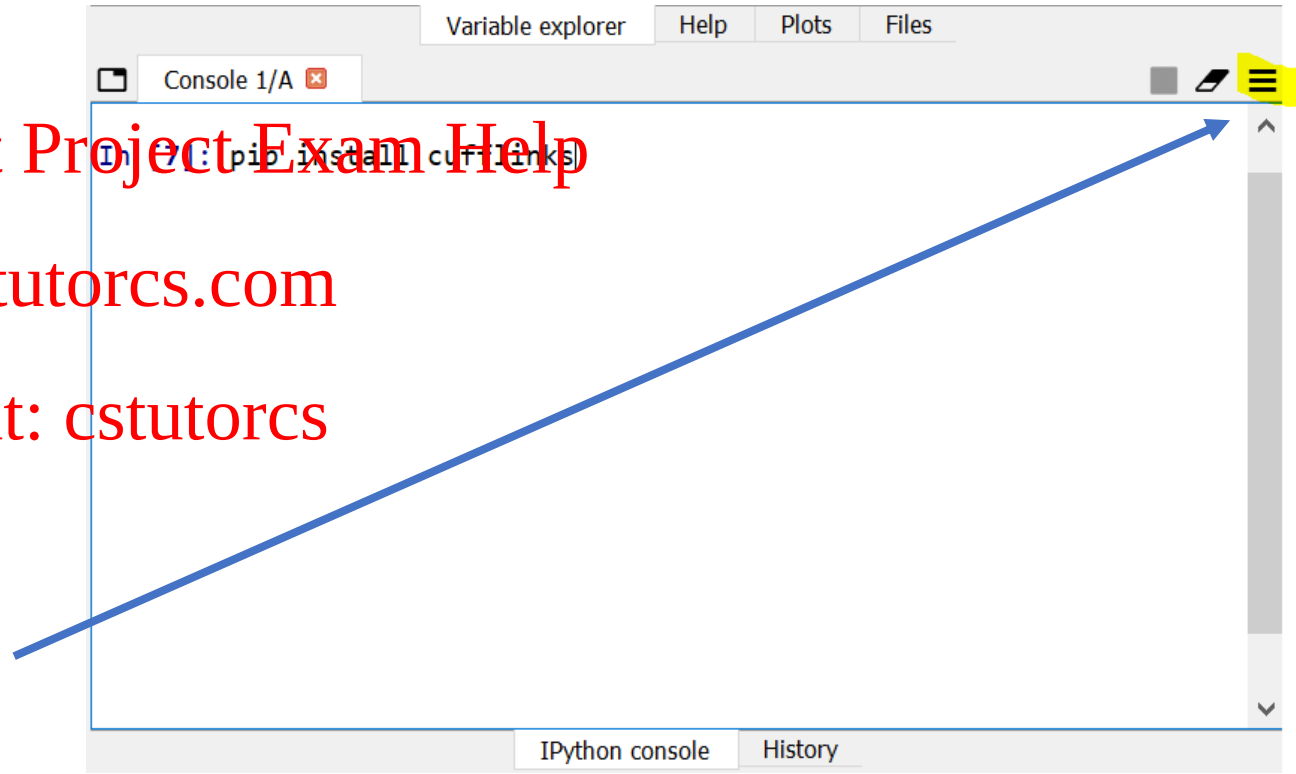
Install Cufflinks library

This library binds the power of plotly with the flexibility of pandas for easy plotting.

https://pypi.org/project/cufflinks/

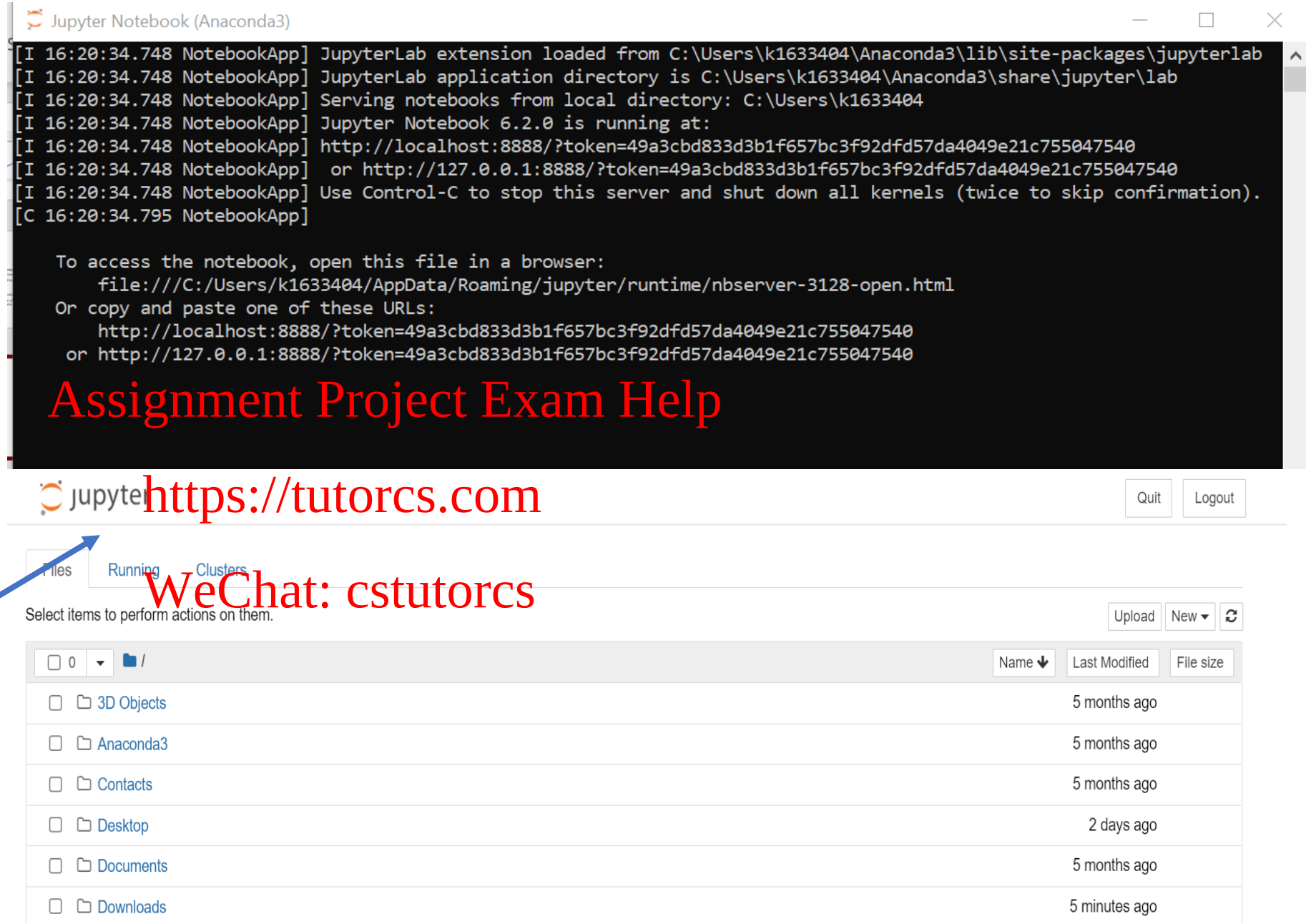
https://github.com/santosjorge/cufflinks

1. In Spyder IPython type pip install cufflinks
2. Hit enter
3. Wait for install to complete. You will get a notification like successfully installed cufflinks.
4. Then click highlight yellow drop down below
5. Click restart kernel
6. Wait for it to restart
7. Once you see the next In [] you can continue coding



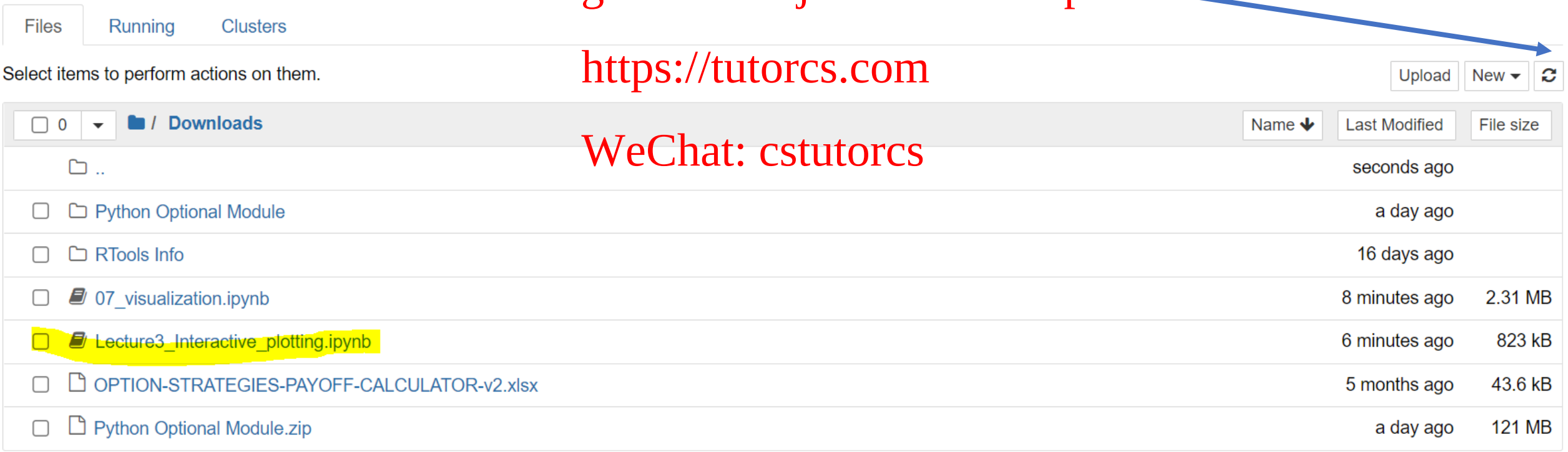
Jupyter Notebooks1

1. Go to Start Menu
2. Click on Anaconda and load **Jupyter Notebook**
3. This will load a black window in background(**Always** keep open)
4. It will also load a prompt to load a browser window Like chrome.



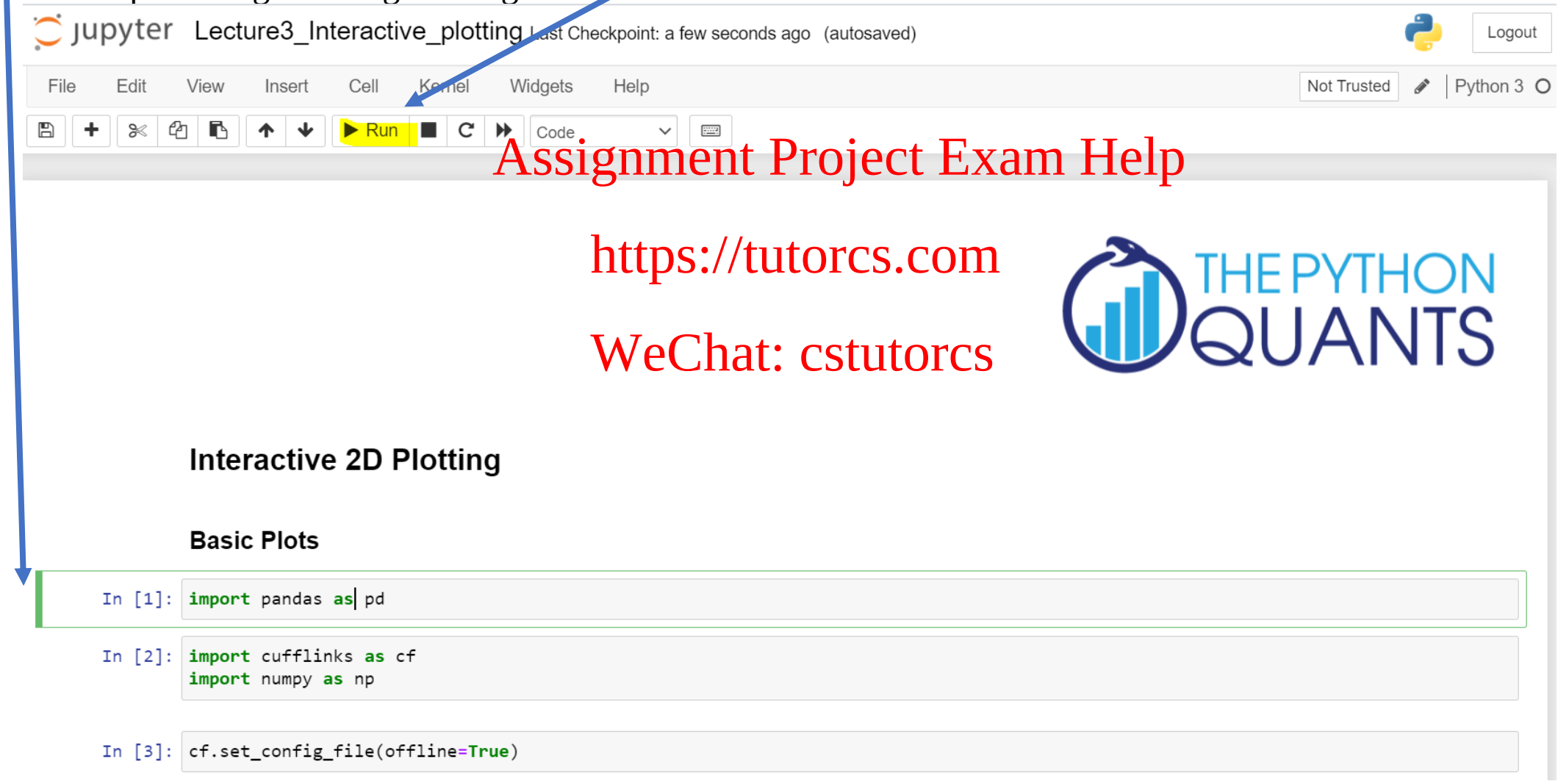
Jupyter Notebooks2

- Copy the lecture code entitled Lecture4_Interactive_plotting.ipynb to your downloads folder for instance
- Click refresh button
- Click on the file highlighted in yellow



Running Jupyter Code

- Navigate to first cell and click Run (highlighted yellow)
- Keep clicking Run to go through each section



Plotly and iplot

- **Jupyter notebook** is required to graph iplot.
- iplot is an **interactive** plot.
- **Plotly** takes Python code and makes **beautiful** looking JavaScript plots.
- They let you have a lot of control over how these plots look and they let you zoom, show information on hover and toggle data to be viewed on the chart.

Basic Plots1

- To get started from within a Jupyter Notebook context, some imports are required and the *notebook mode* should be turned on:

```
In [42]: import pandas as pd

In [43]: import cufflinks as cf
In [44]: import plotly.offline as plyo

In [45]: plyo.init_notebook_mode(connected=True)
```

❶ Imports `Cufflinks`.

❷ Imports the offline plotting capabilities of `plotly`.

❸ Turns on the notebook plotting mode.

REMOTE OR LOCAL RENDERING

With `plotly`, there is also the option to get the plots rendered on the `plotly` servers. However, the notebook mode is generally much faster, in particular when dealing with larger data sets. That said, some functionality, like the streaming plot service of `plotly`, is only available via communication with the server.

Basic Plots2: Data

- The examples that follow rely again on pseudo-random numbers, this time stored in a `DataFrame` object with `DatetimeIndex` (i.e., as time series data):

```
In [46]: a = np.random.standard_normal((250, 5)).cumsum(axis=0)  ❶

In [47]: index = pd.date_range('2019-1-1',  ❷
                               freq='B',  ❸
                               periods=len(a))  ❹

In [48]: df = pd.DataFrame(100 + 5 * a,  ❺
                           columns=list('abcde'),  ❻
                           index=index)  ❼

In [49]: df.head()  ❽
Out[49]:                    a          b          c         d          e
2019-01-01  109.037535  98.693865  104.474094  96.878857  100.621936
2019-01-02  107.598242  97.005738  106.789189  97.966552  100.175313
2019-01-03  101.639668  100.332253  103.183500  99.747869  107.902901
2019-01-04   98.500363  101.208283  100.966242  94.023898  104.387256
2019-01-07   93.941632  103.319168  105.674012  95.891062   86.547934
```

❶ The standard normally distributed pseudo-random numbers.

❷ The start date for the `DatetimeIndex` object.

❸ The frequency ("business daily").

❹ The number of periods needed.

❺ A linear transform of the raw data.

❻ The column headers as single characters.

❼ The `DatetimeIndex` object.

❽ The first five rows of data.

Basic Plots3: Cufflinks Interactive Line Plot1

- `Cufflinks` adds a new method to the `DataFrame` class: `df.iplot()`.
- This method uses `plotly` in the backend to create interactive plots.
- The code examples in this section all make use of the option to download the interactive plot as a static bitmap, which in turn is embedded in the text.
- In the Jupyter Notebook environment, the created plots are all interactive.
- The result of the following code is shown in Figure 7-23 (next slide):


```
In [50]: plyo.iplot(                    ❶
             df.iplot(asFigure=True),   ❷
             # image='png',             ❸
             filename='ply_01'          ❹
         )
```

❶ This makes use of the offline (notebook mode) capabilities of `plotly`.

❷ The `df.iplot()` method is called with parameter `asFigure=True` to allow for local plotting and embedding.

❸ The `image` option provides in addition a static bitmap version of the plot.

❹ The filename for the bitmap to be saved is specified (the file type extension is added automatically).

Basic Plots4: Cufflinks Interactive Line Plot2

Figure 7-22. Line plot for time series data with plotly, pandas, and Cufflinks

Basic Plots5: Cufflinks Interactive Line Plot3

- As with `matplotlib` in general and with the `pandas` plotting functionality, there are multiple parameters available to customize such plots (see Figure 7-23) (next slide) :

```
In [51]: plyo.iplot(
             df[['a', 'b']].iplot(asFigure=True,
                      theme='polar',  ❶
                      title='A Time Series Plot',  ❷
                      xTitle='date',  ❸
                      yTitle='value',  ❹
                      mode={'a': 'markers', 'b': 'lines+markers'},  ❺
                      symbol={'a': 'circle', 'b': 'diamond'},  ❻
                      size=3.5,  ❼
                      colors={'a': 'blue', 'b': 'magenta'},  ❽
                             ),
             # image='png',
             filename='ply_02'
         )
```

❶ Selects a theme (plotting style) for the plot.

❷ Adds a title.

❸ Adds an x-axis label.

❹ Adds a y-axis label.

❺ Defines the plotting *mode* (line, marker, etc.) by column.

❻ Defines the symbols to be used as markers by column.

❼ Fixes the size for all markers.

❽ Specifies the plotting color by column.

Basic Plots6: Cufflinks Interactive Line Plot4

Figure 7-23. Line plot for two columns of the DataFrame object with customizations

Basic Plots7: Cufflinks Interactive Bar Plot1

- Similar to `matplotlib`, `plotly` allows for a number of different plotting types.
- Plotting types available via `Cufflinks` are `chart`, `scatter`, `bar`, `box`, `spread`, `ratio`, `heatmap`, `surface`, `histogram`, `bubble`, `bubble3d`, `scatter3d`, `scattergeo`, `ohlc`, `candle`, `pie`, and `choropleth`.
- As an example of a plotting type different from a line plot, consider the histogram (see Figure 7-24) (next slide):


```
In [52]: plyo.iplot(
             df.iplot(kind='hist',    ❶
                      subplots=True,  ❷
                      bins=15,        ❸
                      asFigure=True),
             # image='png',
             filename='ply_03'
         )
```

❶ Specifies the plotting type.

❷ Requires separate subplots for every column.

❸ Sets the `bins` parameter (buckets to be used = bars to be plotted).

Basic Plots8: Cufflinks Interactive Bar Plot1

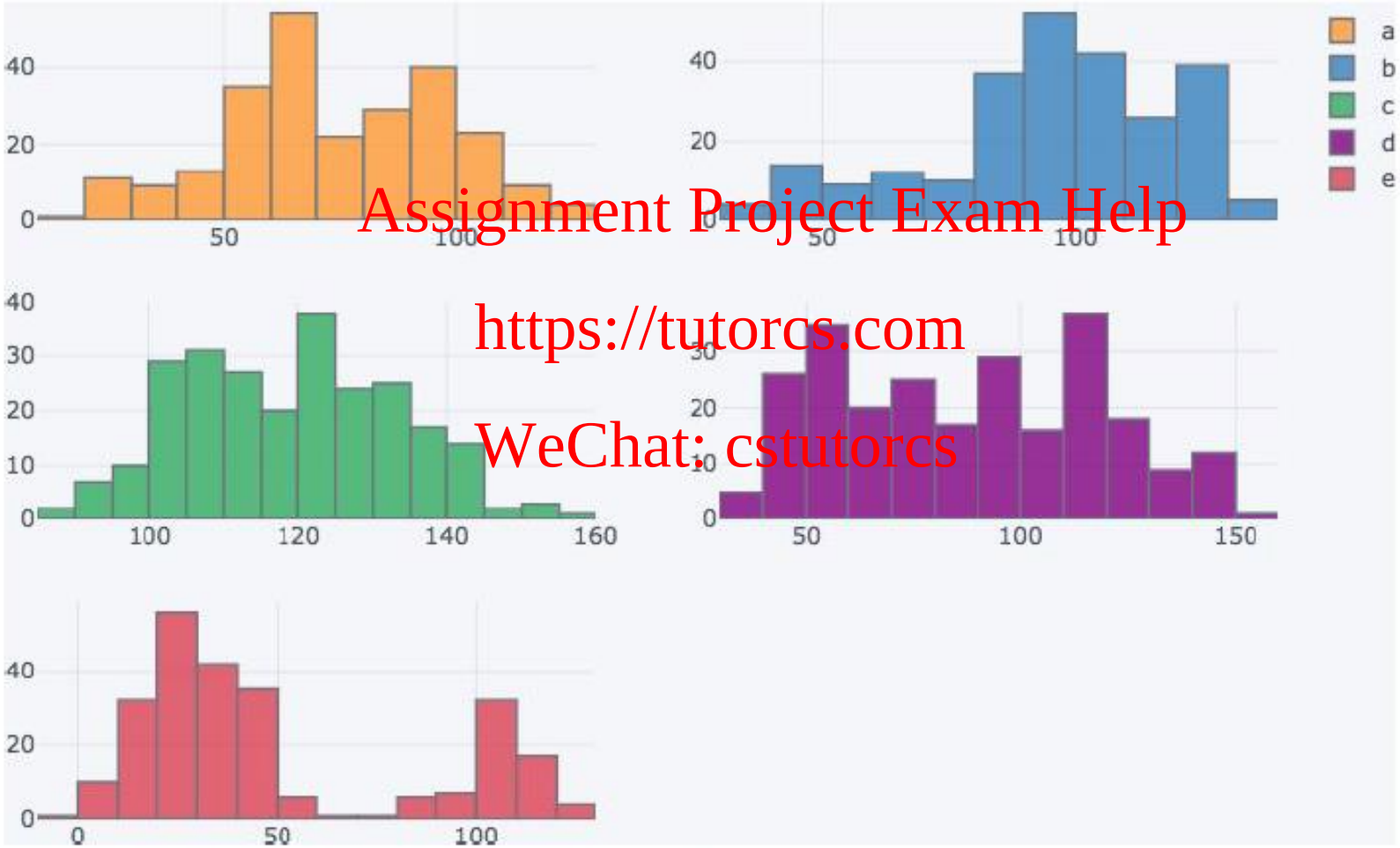
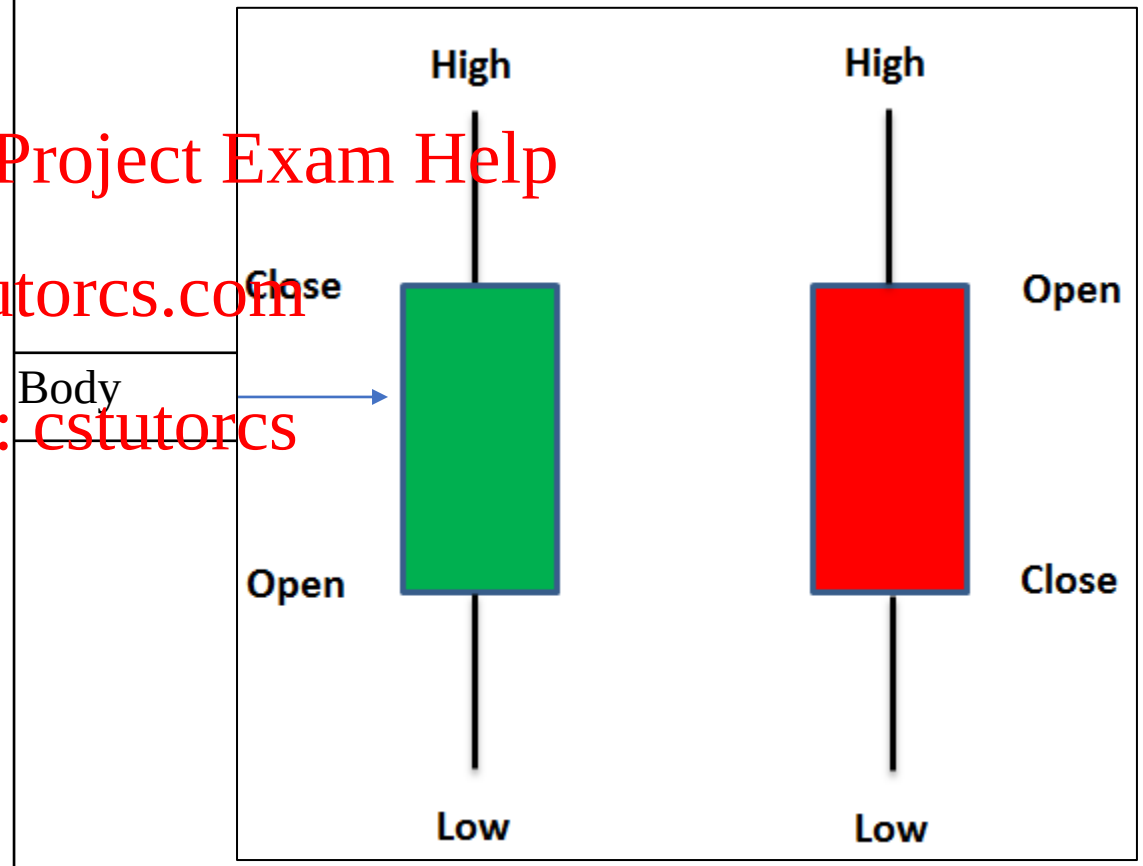


Figure 7-24. Histograms per column of the DataFrame object

Candlesticks (OHLC)

- Candlesticks show price action by visually representing the size of price moves with different colours.
- Traders use the candlesticks to make trading decisions based on regularly occurring patterns that help forecast price direction.
- Candlesticks show four price points **(open, close, high, and low)** throughout any specified time period.

- Green is a **positive** day **Close > Open**
- Red is a **negative** day **Close < Open**
- This colour scheme is typical in UK, European, USA markets
- This coliur scheme is opposite in Asian markets
- Note: Other financial platforms may use different default colour schemes. Logic is still the same.



Financial Plots1

- The combination of `plotly`, `Cufflinks`, and `pandas` proves particularly powerful when working with financial time series data. `Cufflinks` provides specialized functionality to create typical financial plots and to add typical financial charting elements, such as the Relative Strength Index (RSI), to name but one example.
- To this end, a persistent `QuantFig` object is created that can be plotted the same way as a `DataFrame` object with `Cufflinks`.
- This subsection uses a real financial data set, time series data for the
- EUR/USD exchange rate (source: FXCM Forex Capital Markets Ltd.):

```
In [54]: raw = pd.read_csv('../../source/fxcm_eur_usd_eod_data.csv',
                           index_col=0, parse_dates=True)  ❶

In [55]: raw.info()  ❷
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1547 entries, 2013-01-01 22:00:00 to 2017-12-31 22:0
Data columns (total 8 columns):
BidOpen     1547 non-null float64
BidHigh     1547 non-null float64
BidLow      1547 non-null float64
BidClose    1547 non-null float64
AskOpen     1547 non-null float64
AskHigh     1547 non-null float64
AskLow      1547 non-null float64
AskClose    1547 non-null float64
dtypes: float64(8)
memory usage: 108.8 KB

In [56]: quotes = raw[['AskOpen', 'AskHigh', 'AskLow', 'AskClose']]  ❸
         quotes = quotes.iloc[-60:]  ❹
         quotes.tail()  ❺
Out[56]:
                     AskOpen  AskHigh   AskLow  AskClose
2017-12-25 22:00:00  1.18667  1.18791  1.18467   1.18587
2017-12-26 22:00:00  1.18587  1.19104  1.18552   1.18885
2017-12-27 22:00:00  1.18885  1.19592  1.18885   1.19426
2017-12-28 22:00:00  1.19426  1.20256  1.19369   1.20092
2017-12-31 22:00:00  1.20092  1.20144  1.19994   1.20144
```

❶ Reads the financial data from a CSV file.

❷ The resulting `DataFrame` object consists of multiple columns and more than 1,500 data rows.

❸ Selects four columns from the `DataFrame` object (Open-High-Low-Close, or OHLC).

❹ Only a few data rows are used for the visualization.

❺ Returns the final five rows of the resulting data set `quotes`.

Financial Plots2

- During instantiation, the `QuantFig` object takes the `DataFrame` object as input and allows for some basic customization. Plotting the data stored in the `QuantFig` object `qf` then happens with the `qf.iplot()` method (see Figure 7- 25) (next slide):


```
In [57]: qf = cf.QuantFig(
                    quotes,  ❶
                    title='EUR/USD Exchange Rate',  ❷
                    legend='top',  ❸
                    name='EUR/USD'  ❹
                )

In [58]: plyo.iplot(
                qf.iplot(asFigure=True),
                # image='png',
                filename='qf_01'
            )
```

❶ The `DataFrame` object is passed to the `QuantFig` constructor.

❷ This adds a figure title.

❸ The legend is placed at the top of the plot.

❹ This gives the data set a name.

Figure 7-25. OHLC plot of EUR/USD data

Bollinger Bands

- A Bollinger Band is a technical analysis tool defined by a set of trendlines plotted two standard deviations (positively and negatively) away from a simple moving average (SMA) of a security's price, but which can be adjusted to user preferences.
- A **simple moving average (SMA)** is simply the sum of all previous closing days divided by the amount of days

- Bollinger Bands® are a technical analysis tool developed by John Bollinger for generating oversold or overbought signals.
- There are three lines that compose Bollinger Bands: A simple moving average (middle band) and an upper and lower band.
- The upper and lower bands are typically 2 standard deviations +/- from a 20-day simple moving average, but can be modified.

If Price > Upper Bollinger Band = Sell Signal

If Price < Lower Bollinger Band = Buy Signal

Take Profit Strategy 1: When Price hits MA

Take Profit Strategy 2: When Price crosses the opposite Bollinger Band

https://www.investopedia.com/terms/b/bollingerbands.asp

Financial Plots4

- Adding typical financial charting elements, such as Bollinger bands, is possible via different methods available for the `QuantFig` object (see Figure 7-26):

```
In [59]: qf.add_bollinger_bands(periods=15,
                                boll_std=2)

In [60]: plyo.iplot(qf.iplot(asFigure=True),
                    # image='png',
                    filename='qf_02'
                )
```

❶ The number of periods for the Bollinger band.

❷ The number of standard deviations to be used

Figure 7-26. OHLC plot of EUR/USD data with Bollinger band

Relative Strength Index (RSI)

- The relative strength index (RSI) is a momentum indicator used in technical analysis that measures the magnitude of recent price changes to evaluate overbought or oversold conditions in the price of a stock or other asset.

- The RSI is displayed as an oscillator (a line graph that moves between two extremes) and can have a reading from 0 to 100.

- Traditional interpretation and usage of the RSI are that values of 70 or above indicate that a security is becoming overbought or overvalued and may be primed for a trend reversal or corrective pullback in price.
- An RSI reading of 30 or below indicates an oversold or undervalued condition.

It is calculated using the following formula:

RSI = 100 - 100/(1 + RS*)

*Where RS = Average of x days' up closes / Average of x days' down closes.
. https://www.investopedia.com/terms/r/rsi.asp

Financial Plots5

- Certain financial indicators, such as RSI, may be added as a subplot (see Figure 7-27):

```
In [61]: qf.add_rsi(periods=14,  ❶
                    showbands=False)  ❷

In [62]: plyo.iplot(
             qf.iplot(asFigure=True),
             # image='png',
             filename='qf_03'
         )
```

❶ Fixes the RSI period.

❷ Does not show an upper or lower band.

Figure 7-27. OHLC plot of EUR/USD data with Bollinger band and RSI

Conclusion

- `matplotlib` can be considered both the benchmark and an all-rounder when it comes to data visualization in Python. It is tightly integrated with `NumPy` and `pandas`, and the basic functionality is easily and conveniently accessed.
- However, `matplotlib` is a mighty library with a somewhat complex API.
- This makes it impossible to give a broad overview of all the capabilities of `matplotlib` in this chapter.
- This chapter introduces the basic functions of `matplotlib` for 2D and 3D plotting useful in many financial contexts. Other chapters provide further examples of how to use the package for visualization.
- In addition, this chapter covers `plotly` in combination with `Cufflinks`.
- This combination makes the creation of interactive `D3.js` plots a convenient affair since only a single method call on a `DataFrame` object is necessary in general.
- All technicalities are taken care of in the backend.
- Furthermore, `Cufflinks` provides with the `QuantFig` object an easy way to create typical financial plots with popular financial indicators.

Further Resources

- A variety of resources for `matplotlib` can be found on the web, including:
- The home page, https://matplotlib.org/ which is probably the best starting point
- A gallery https://matplotlib.org/stable/gallery/index.html?highlight=gallery with many useful examples
- A tutorial for 2D plotting https://matplotlib.org/stable/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py
- A tutorial for 3D plotting https://matplotlib.org/stable/tutorials/toolkits/mplot3d.html#sphx-glr-tutorials-toolkits-mplot3d-py

- It has become kind of a standard routine to consult the gallery, look there for an appropriate visualization example, and start with the corresponding example code.
- The major resources for the `plotly` and `Cufflinks` packages are also online. These include:
- The `plotly` home page https://plotly.com/
- A tutorial to get started with `plotly` for Python https://plotly.com/python/getting-started/
- The `Cufflinks` GitHub page https://github.com/santosjorge/cufflinks