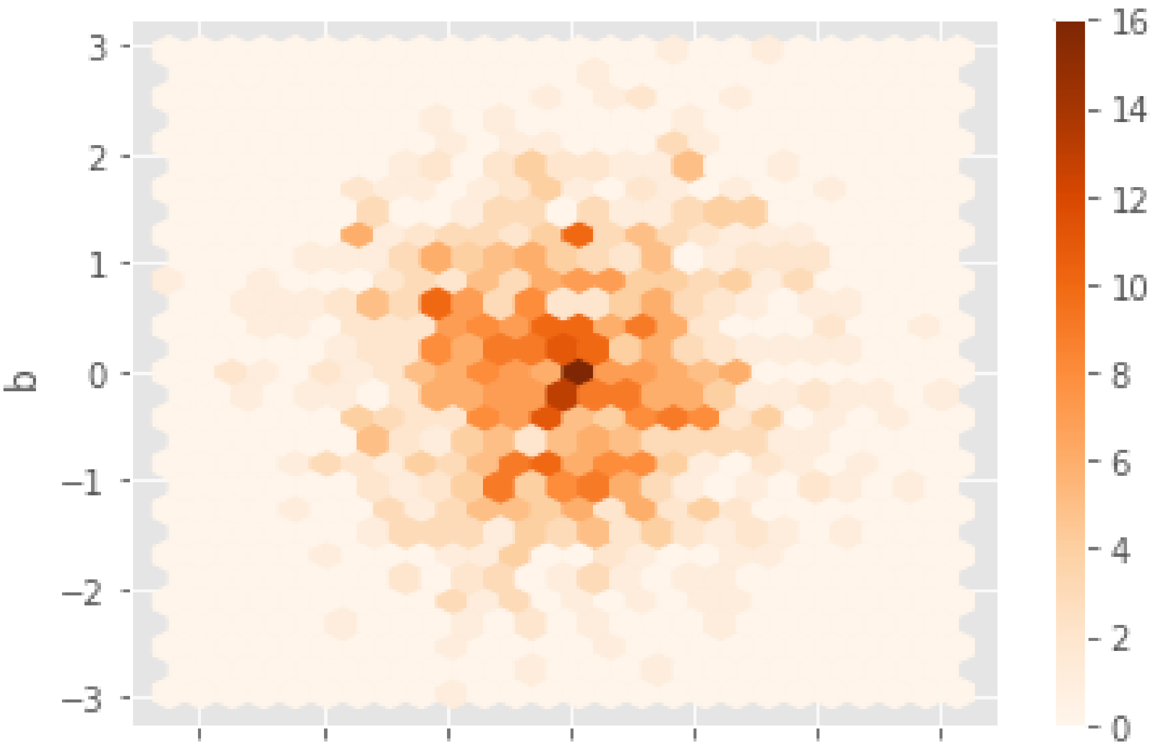
A Comprehensive Guide to Data Visualization with Python for Complete Beginners – Pandas Data Visualization

helloml.org/a-comprehensive-guide-to-data-visualization-with-python-for-complete-beginners-pandas-data-visualization/

By Ajit Singh October 6, 2021

This article is a part of the ‘Comprehensive guide to data visualization with Python for complete beginners’ series, with the previous articles covering an introduction to data visualization and hands-on data visualization with two of the most popular graphing tools in Python: Matplotlib and Seaborn. The links to the tutorials can be found here and here respectively for tutorials on Matplotlib and Seaborn.

In this article, we will explore another possibility for visualizing our plots, with **Pandas built-in data visualization**. As we know, Pandas is primarily used for data manipulation and analysis in Python. However, Pandas also has built-in capabilities for data visualization. It’s derived from Matplotlib but is baked into Pandas for easier usage. It is not as popular as compared to using other tools for data visualization but comes in handy if someone wants a glance at their data or wants quick data visualization rather than preparing highly customized, exhaustive graphs/plots. However, it is still capable of producing powerful visualizations if we add some Matplotlib customizations. So let’s dive straight in!



Example of graphing plots using Pandas built-in visualization: A **hexagonal bin plot**.

Setting up

Installing the Required Packages

We will be using **NumPy**, **Pandas**, and **Matplotlib** for this tutorial. If you do not have these installed, you can do so by simply typing in the following commands (Assuming you're working on a Jupyter notebook) in the Anaconda command prompt:

```
conda install numpy pandas matplotlib seaborn
```

If not, type in the following command in the terminal/command line with pip installed (You need to have a version of Python installed locally for this to work):

```
pip install numpy pandas matplotlib
```

Importing the Required Libraries

```
import numpy as np
import pandas as pd
%matplotlib inline
```

'**%matplotlib inline**' is a pre-defined **magic function** that enables our plot outputs to appear and be stored within the Jupyter notebook. If you're not using jupyter notebook, then you'll have to use **plt.show()** for output to appear in a graphical window.

Generating Data for Visualization

In this tutorial, we will not use any library's built-in datasets. Instead, we will create fake data ourselves for various kinds of plots.

We'll generate 2 different data frames populated with random numbers. One among these will be indexed with fake time series data which we will generate using **NumPy** (Because we'll also graph a line plot and line plots are generally used for visualizing trends over a period of time).

```
date = np.array('2001-01-01', dtype=np.datetime64)
date = date + np.arange(1000)
df1 = pd.DataFrame(np.random.randn(1000,4), index=date, columns='a b c d'.split())
```

Let's check the **head** of this dataframe:

```
df1.head()

                   a              b                     c             d
2001-01-01      -0.331089      -0.028812            -0.529715   -0.079998
2001-01-02      -0.323879       1.119133              0.029561   0.057041
2001-01-03       0.272296       0.821087            -0.203703    1.147628
2001-01-04       0.759472       -1.262508             1.009500  -0.951337
2001-01-05      -2.011348       0.752604            -1.630587    0.137959
```

Next, we will generate another dataframe with random data:

```
df2 = pd.DataFrame(np.random.random((10,4)),columns='a b c d'.split())
df2.head()
```

```
            a       b       c       d
0       0.875663        0.287279        0.894160        0.660192
1       0.910721        0.664171        0.355575        0.956805
2       0.446677        0.512392        0.981894        0.100121
3       0.730667        0.983717        0.595256        0.546215
4       0.725861        0.887893        0.227208        0.466892
```

Plot Types

There are many types of plots built-in into pandas. Most of these plots by nature are **statistical plots** (Plots which are used to present the results of some statistical analysis of the data rather than show the data themselves are called statistical plots. For eg: Box plot is a statistical plot that provides clear statistical inference like the mean, Inter Quartile Range, Range, etc.) whereas a plot such as a scatter plot is a conventional 2-D plot used to display data points.

Some Plot Types Built into Pandas

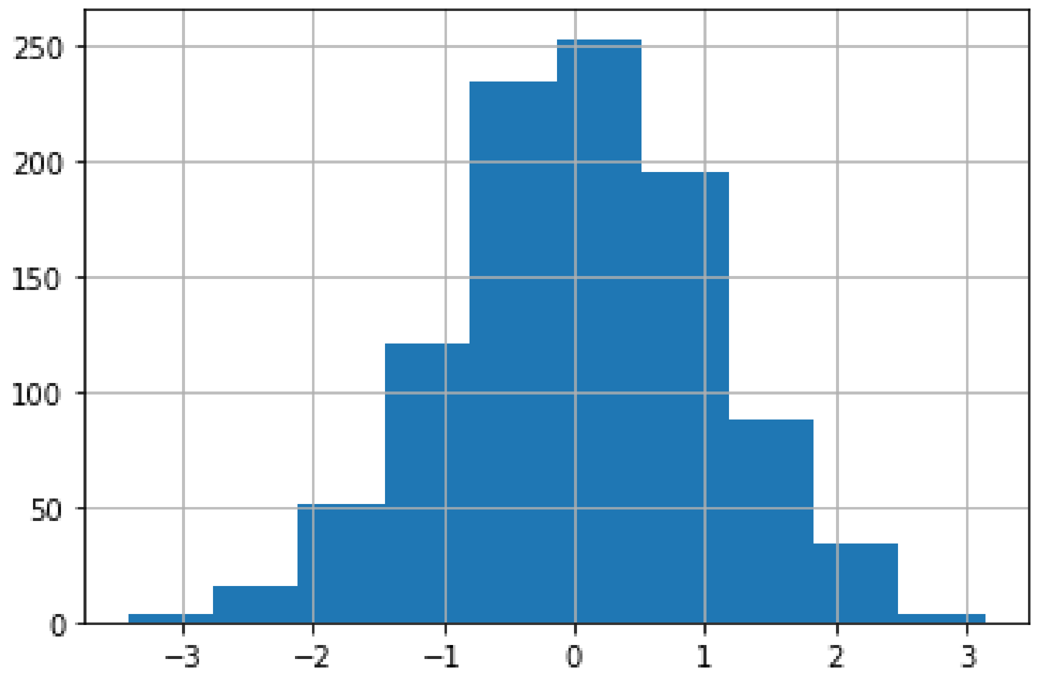
(**df.plot.plotname** is the standard convention to call a plot directly off the dataframe. We can also just call **df.plot(kind='plotname')** and replace the '**kind**' argument with any of the terms shown in the list below. For example, to plot a histogram, we can also use **df.plot(kind='hist')**.

- df.plot.area
- df.plot.barh
- df.plot.density
- df.plot.hist
- df.plot.line
- df.plot.scatter
- df.plot.bar
- df.plot.box
- df.plot.hexbin
- df.plot.kde
- df.plot.pie

Histogram

To plot a histogram, simply call the **pandas.DataFrame.plot.hist** function on the dataframe:

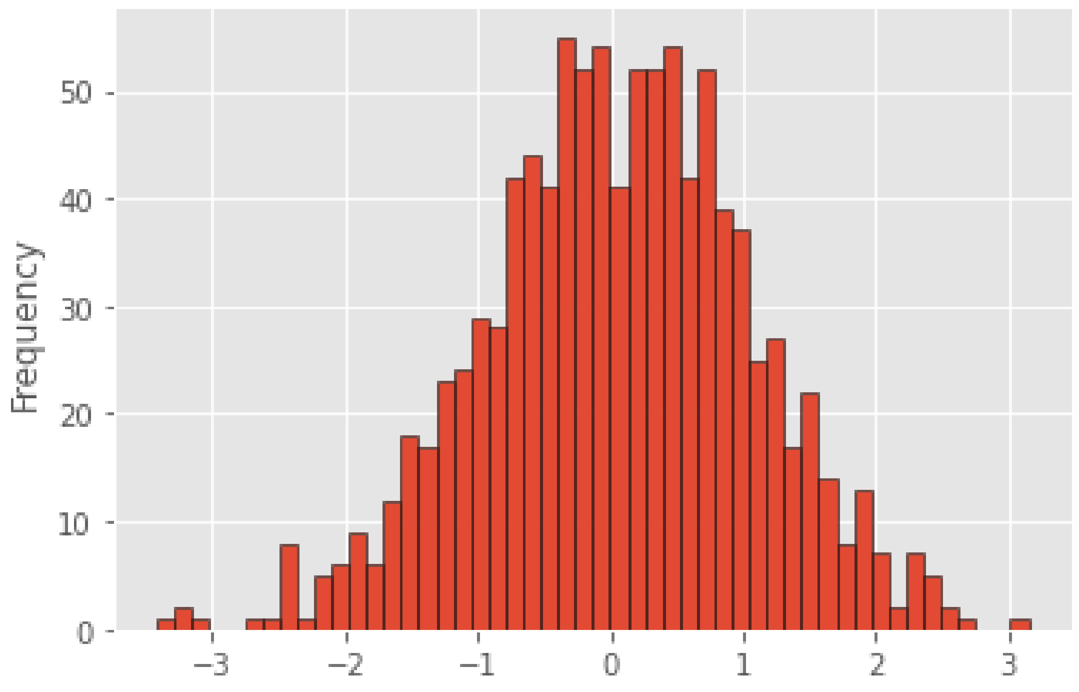
```
df1['a'].plot.hist()
```



Output

As we can see, the bars of the histogram are not well defined. Also, some people may not like the default background style of the graph. We had seen earlier that the built-in visualization in Pandas is just implementing Matplotlib under the hood. Hence, to customize our plots we will use Matplotlib commands and Matplotlib's style sheet to change the background as well. But first, we'll need to import Matplotlib. Here are some options to make our histogram plot look nicer if needed.

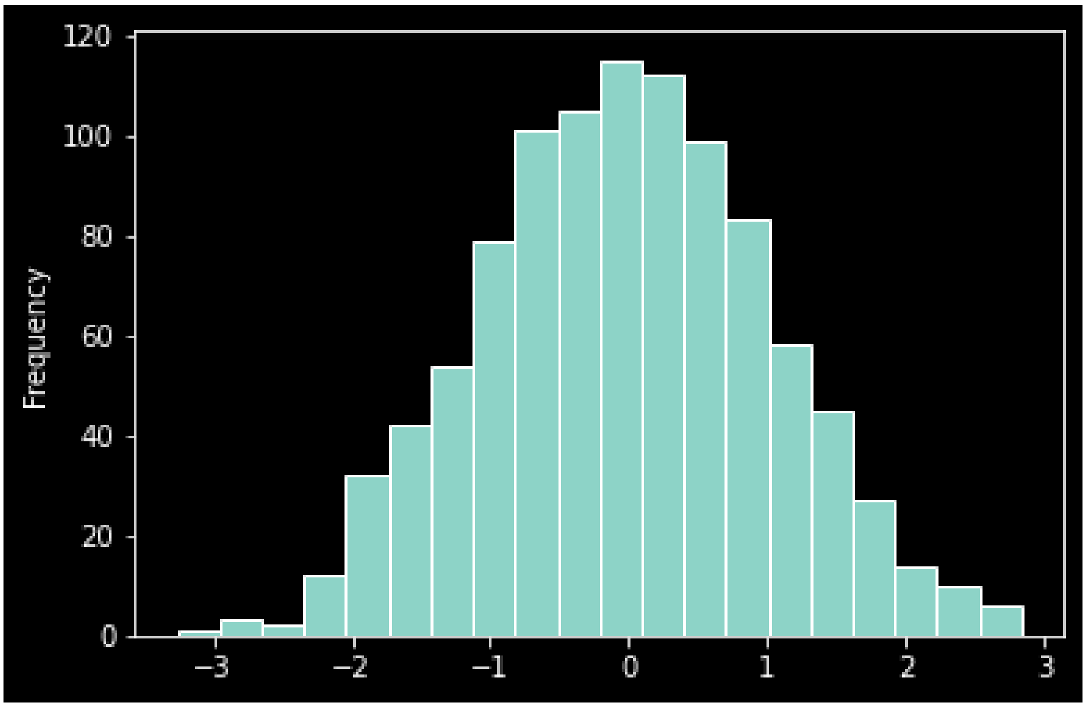
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')
df1['a'].plot.hist(bins=50, edgecolor='black')
```



Output

Let's reduce the number of bins and change the style sheet to '**dark_background**'.

```
plt.style.use('dark_background')
df1['a'].plot.hist(bins=20, edgecolor='white')
```



Output

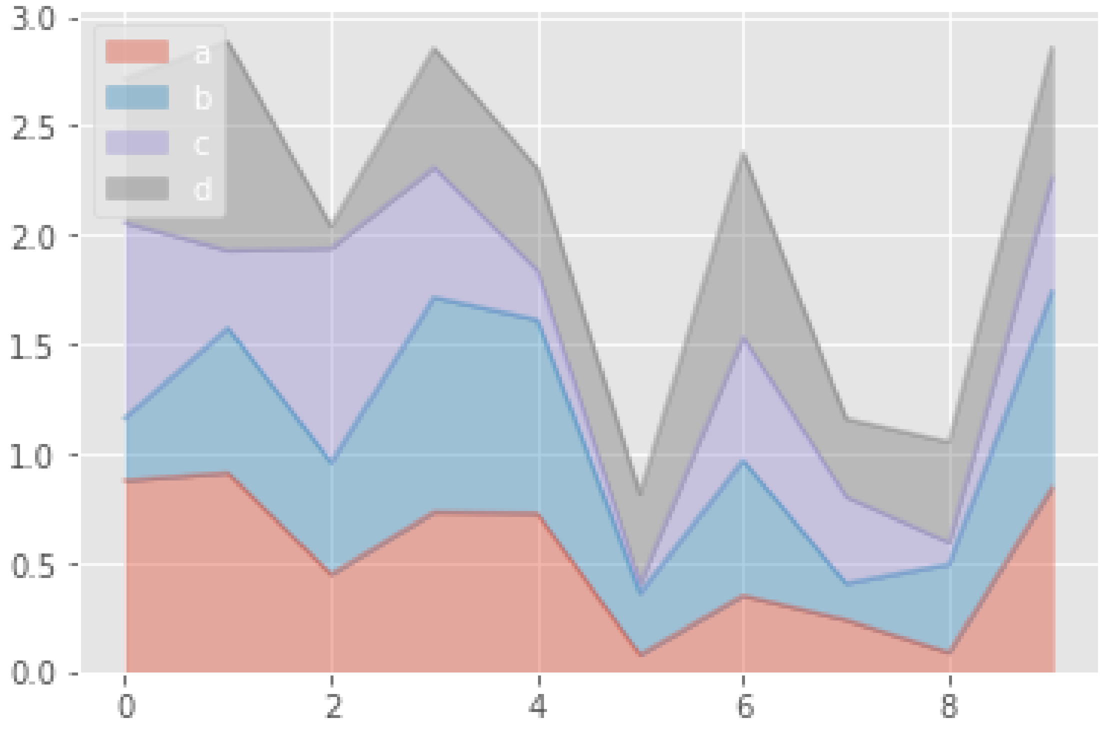
These 2 plots just touched the tip of the iceberg when it comes to customizing and tweaking plots using Matplotlib. Moving on, we will stick with the '**ggplot**' stylesheet provided by Matplotlib (A matter of my personal preference). Let's move ahead and go through most of the built-in plot types which we had listed earlier.

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

Area Plot

Area plots are charts that are based on bar and line plots. Instead of connecting our data points with a continuous line, we also fill in the area below that line with a solid colour. Thus, we represent the quantity in terms of region spanned. Area plots are useful for showing the total amounts of data covered by a variable over time.

```
df2.plot.area(alpha=0.4)
```



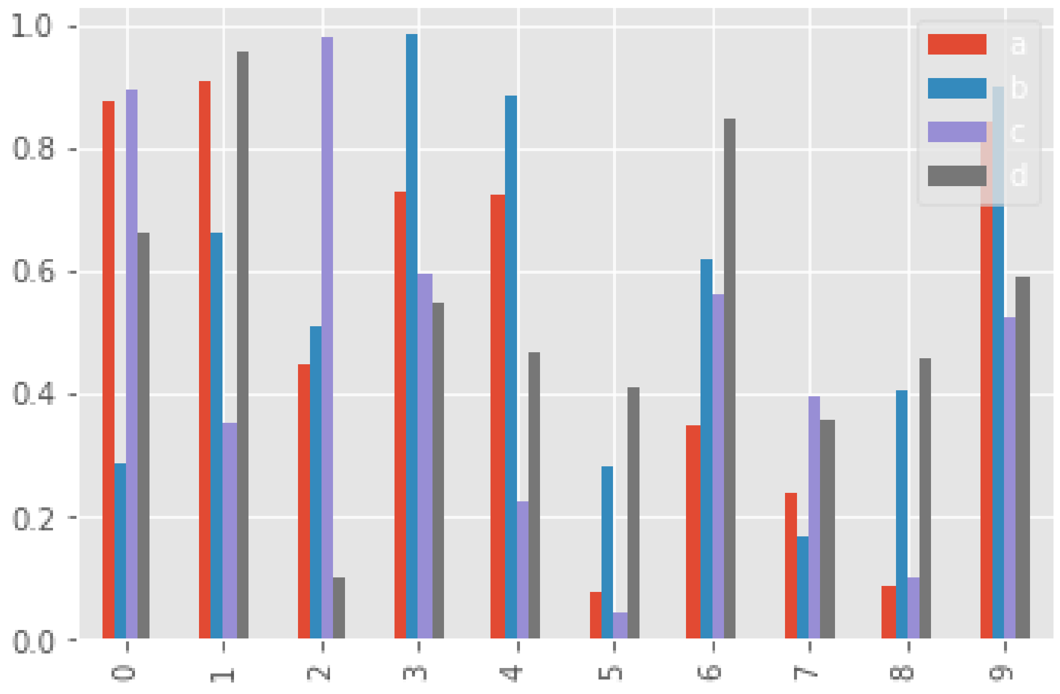
Output

Bar Plot

A bar plot is an example of a categorical plot. Categorical plots are the ones in which we plot a categorical column vs a numerical column or another categorical column.

A bar plot presents categorical data with rectangular bars with lengths proportional to the values that they represent. It is useful for showing comparisons.

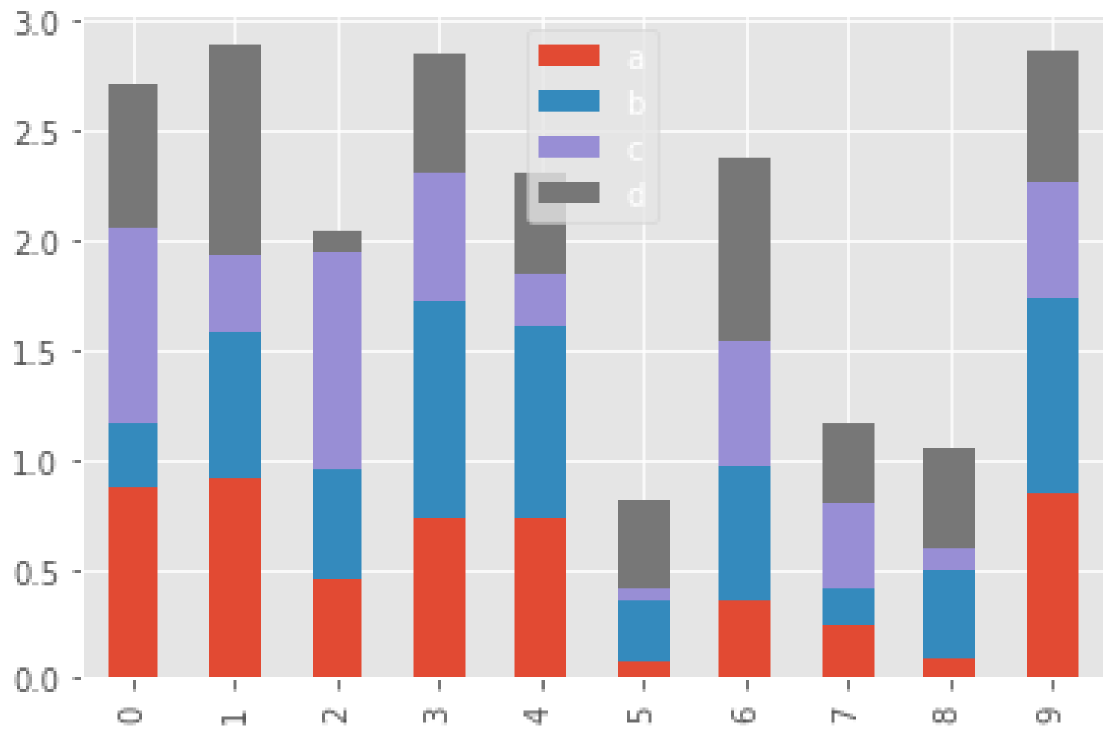
```
df2.plot.bar()
```



Output

As you can see, the bars for each index appear too constricted in the compact space. We also have the option to stack them on top of each other instead of displaying them side-by-side.

```
df2.plot.bar(stacked=True)
```

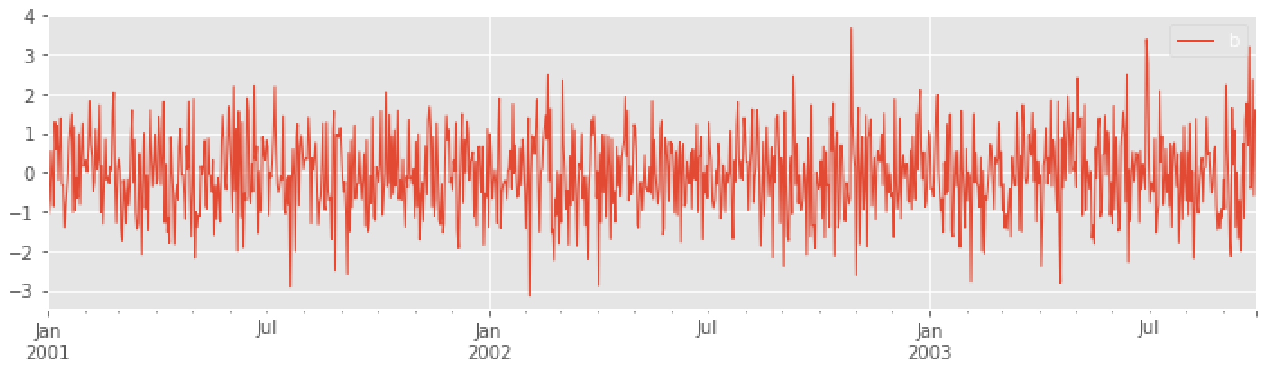


Output

Line Plot

Line plots are generally used to view changes in trends over time. We had already indexed our **df1** with time data. We'll also add some customizations (Changing the figure size and setting the line width to **1**).

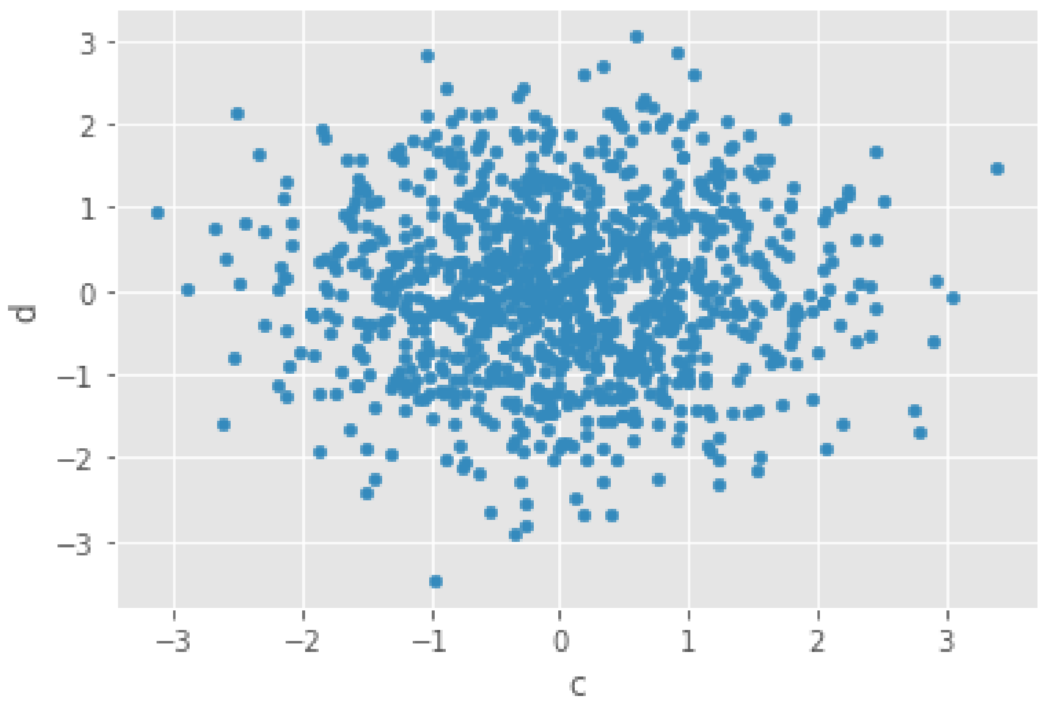
```
df1.plot.line(y='b',figsize=(12,3),lw=1)
```



Output

Scatter Plot

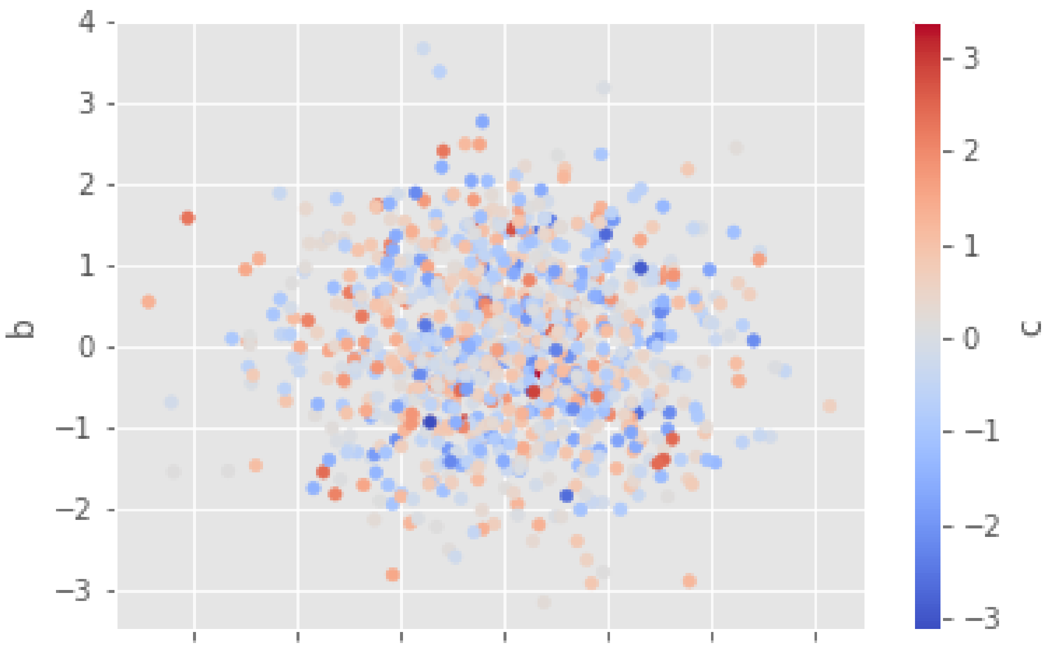
```
df1.plot.scatter(x='c',y='d')
```



Output

We can use the argument ‘**c**‘ to color the scatter points based on another column value. Use ‘**cmap**‘ to indicate the colormap to use.

```
df1.plot.scatter(x='a',y='b',c='c',cmap='coolwarm')
```

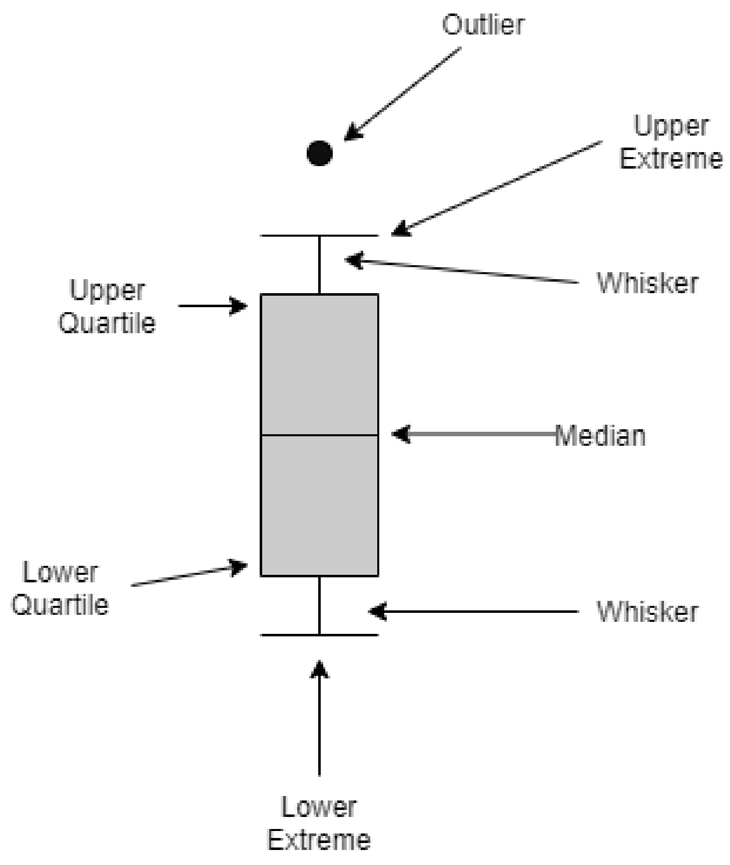


Output

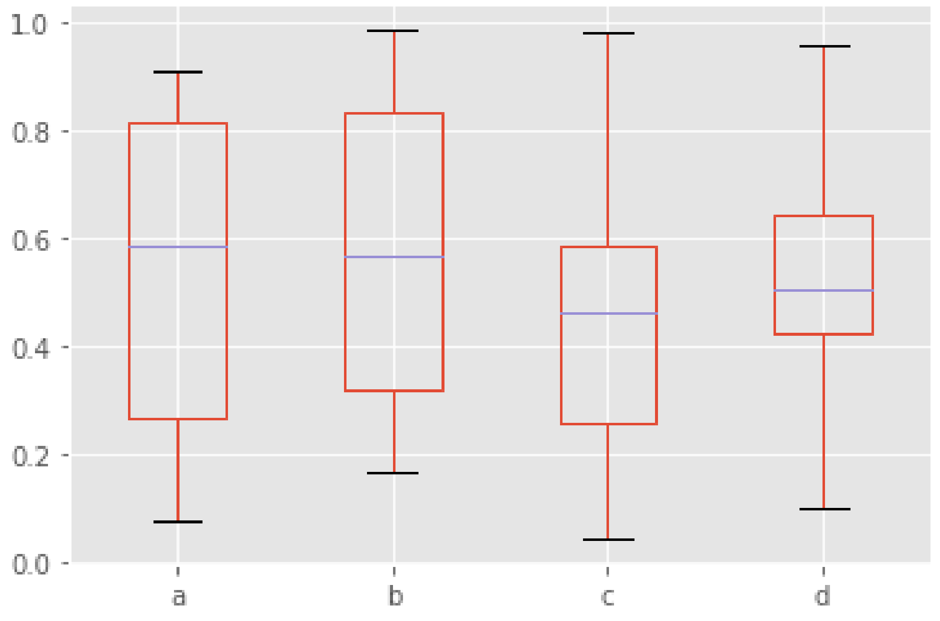
Box Plot

A box plot displays the five-number summary of a set of data. The five-number summary is the minimum, first quartile, median, third quartile, and maximum. The minimum and maximum points are represented by the dash in the horizontal axis at the edge of each plot. The lines extending parallel (inline) from the boxes which reach up to the extremes (upper and lower) are known as the “whiskers”, which indicate variability outside the upper and lower quartiles. For some plots, dots follow the maximum dash. Those points are the outliers. The outliers are in the same line as the whiskers. For more clarity, refer to the below figure:

```
df2.plot.box()
```



Explanation of Box-Plot

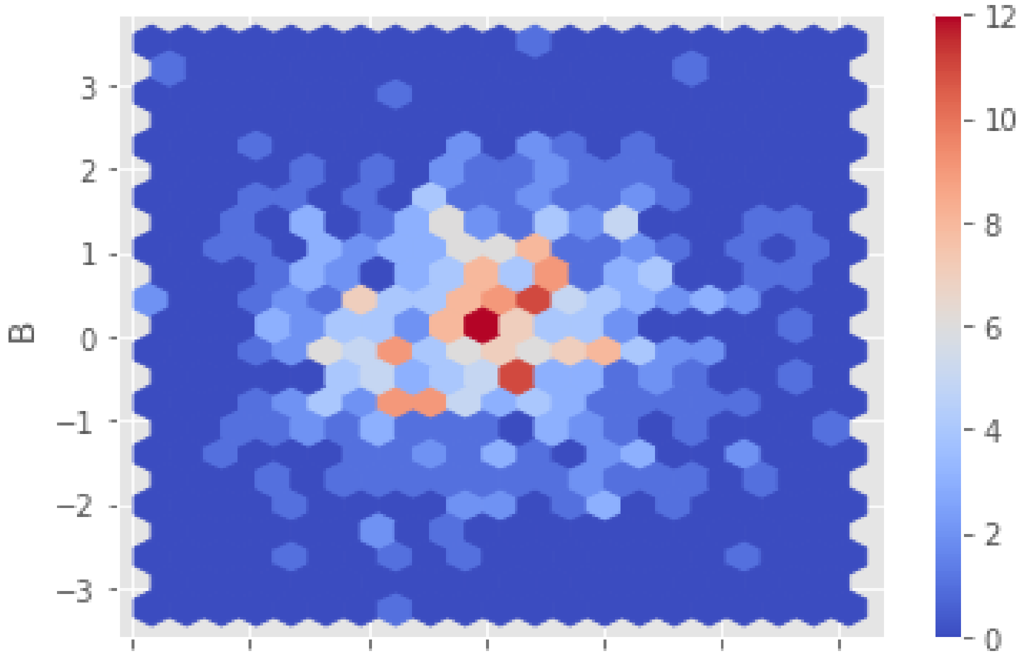


Output

Hexagonal Bin Plot

Hexagonal bin plots are alternative to scatter plots. They are used for bivariate data. This is a great resource for getting started with hexagonal binning. For our example, we are first creating dummy data of size (500×2) and then plotting the hexagonal bin plot for this dummy data. We are also setting the grid size to **20** and are using the commonly used colormap i.e. '**coolwarm**' :

```
df = pd.DataFrame(np.random.randn(500, 2), columns=['A', 'B'])
df.plot.hexbin(x='A',y='B',gridsize=20,cmap='coolwarm')
```

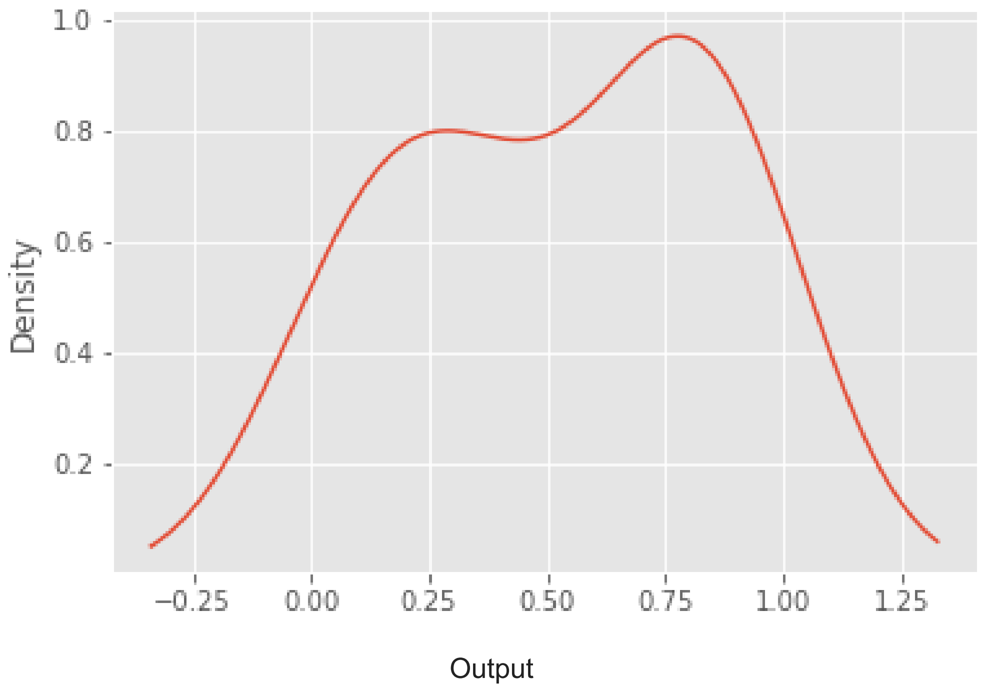


Output

Kernel Density Estimate (KDE) Plot

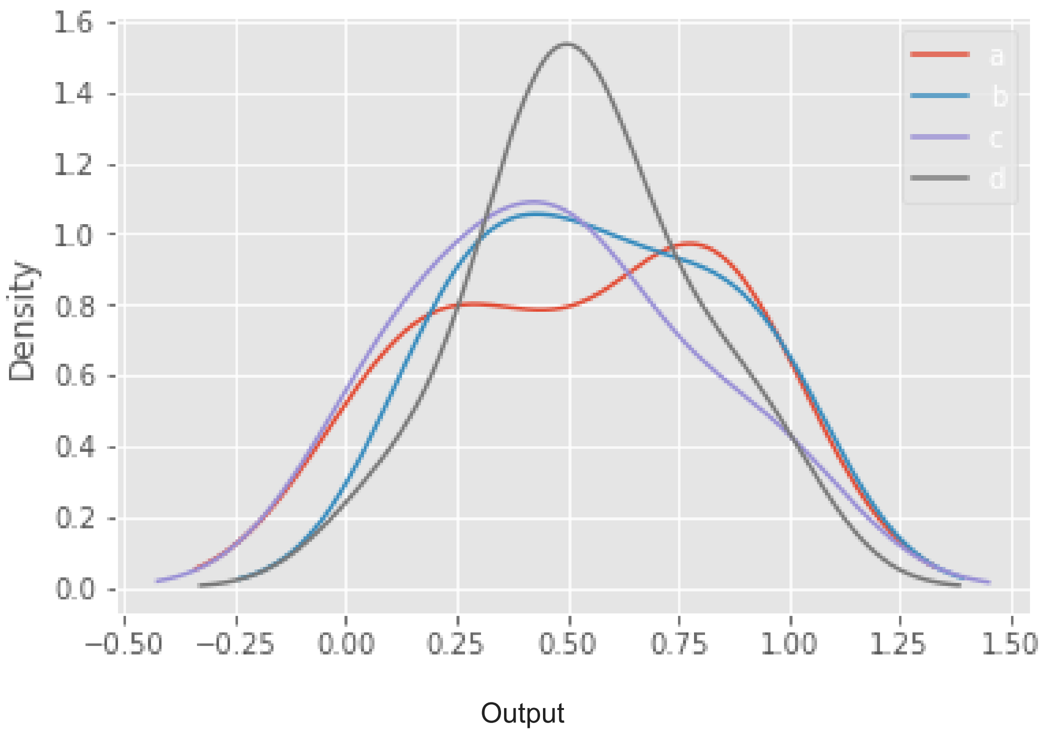
KDE plots allow us to graph the kernel density estimate for the distribution of a particular column. This is a great resource to learn more about kernel density estimation.

```
df2['a'].plot.kde()
```



Instead of calling df2.plot.**kde**(), we can also call df2.plot.**density**(). If we don't specify the column, it'll graph the KDE plot for every column present in the dataframe.

```
df2.plot.density()
```



That's it for today! Hopefully, now you know how to quickly create certain types of plots by calling methods directly off Pandas data frames without having to use another library. It's a lot easier than using Matplotlib directly. We may get slightly less control over our figures; However, the ease of use makes up for it. Also, we learned how to customize these plots by providing additional arguments of their parent matplotlib.plt (such as **bins** for histogram, using **edgecolor**, changing **stylesheet**, **figsize**, etc).