Analytics Vidhya

Source: Pixabay

The Simplest Way to Create Visualizations in

Python Isn't With matplotlib.

Creating Sleek & Easy Plots Directly From Pandas

Andre Ye Follow
Jul 5 · 6 min read ★

matplotlib is generally considered to be the simplest way to create visualizations in Python, and it has formed the basis for many other plotting libraries like seaborn. However, there's an argument to be made that creating simple, quick, and elegant plots for analysis is better done with pandas, the popular data manipulation library.

There are many advantages to plotting directly from pandas:

- It's just faster — fewer lines of code, less code that needs to be written, fewer libraries that need to be re-loaded. Visualization is essential to data analysis, so there is no reason why one shouldn't actively try to make the process of producing a good one faster. Plus, pandas does a lot of inference in what we want it to plot, so it can visualize what we want it to in many cases without explicitly declaring them.
- It's easier to directly incorporate many of pandas' handy data manipulation functions involving Series and DataFrames, like differencing and rolling means.
- It's actually *easier* to create plots directly from pandas. Converting between bar charts, stacked bar charts, and horizontal bar charts is just a matter of changing a parameter value. Creating subplots and manipulating them is also just a few characters away. Pandas carries much of the hard work in visualizing.

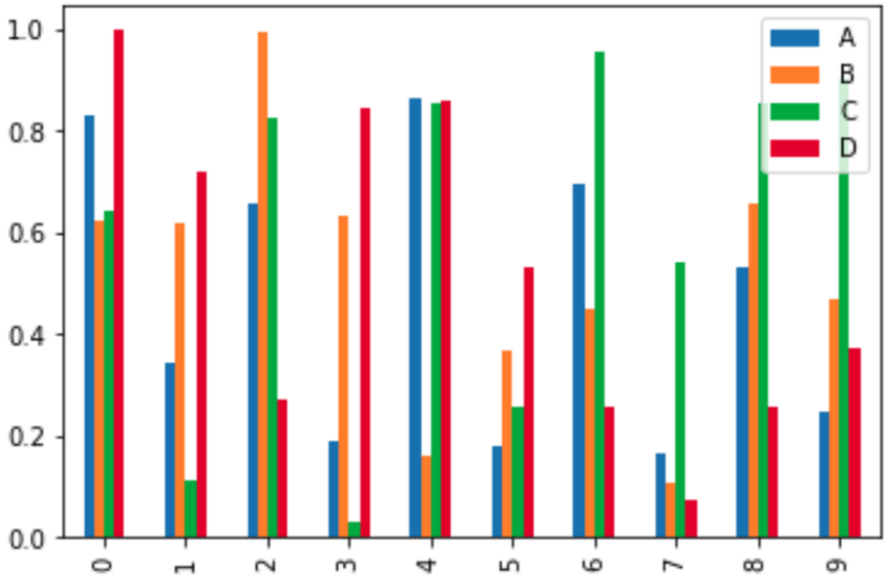
And for the technical: yes, pandas' plotting runs on matplotlib infrastructure, and loading other matplotlib items or parameters on top of a pandas-created graph can improve it. Instead, pandas simply provides a convenient and more direct *interface* to connect the data to the visualization.

Consider, for instance, the following DataFrame randomly generated with four columns and ten rows.

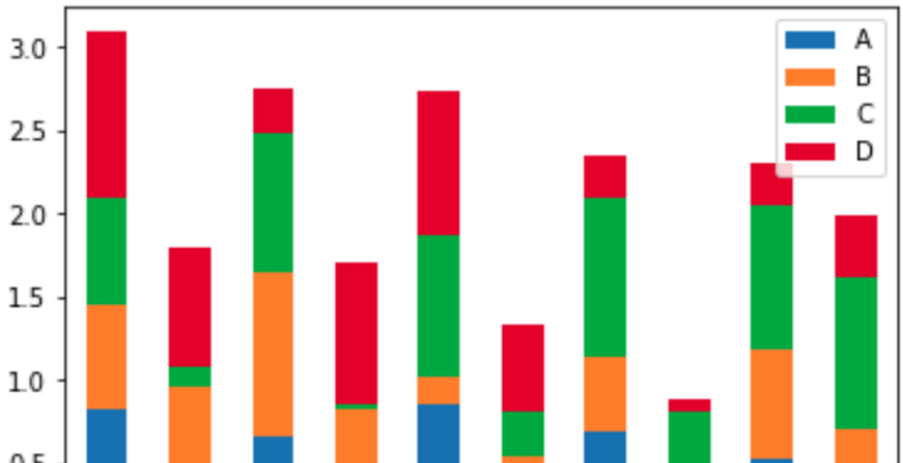
	A	B	C	D
0	0.828853	0.621988	0.641688	0.997204

1	0.341554	0.619444	0.114360	0.719739
2	0.658874	0.993483	0.824533	0.272312
3	0.190003	0.635236	0.033074	0.844827
4	0.862987	0.158771	0.855682	0.858407

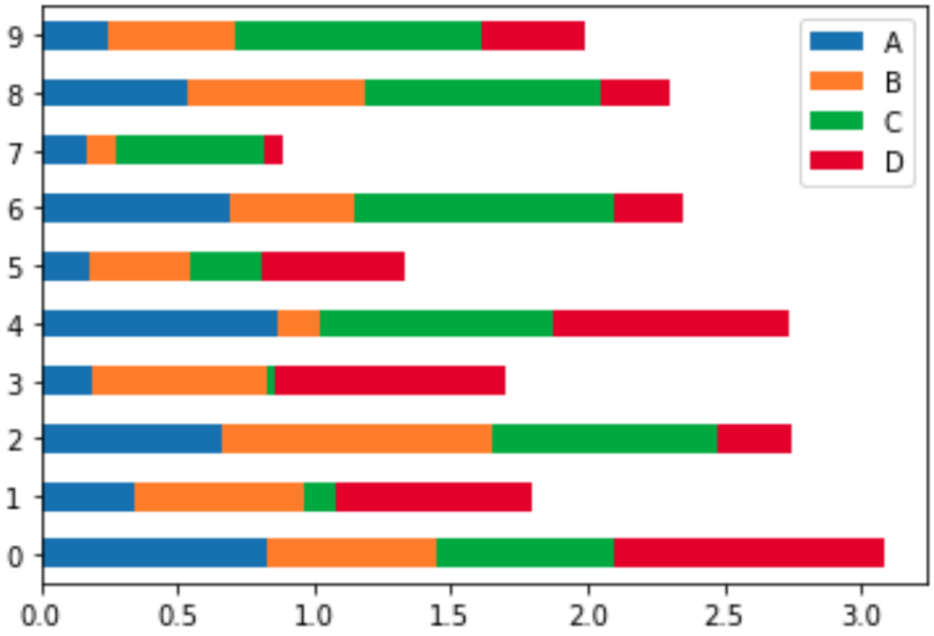
One can simply plot out the values for each of these columns per row simply with `data.plot.bar();`, where `data` should be replaced with the name of the DataFrame. Note that adding the semicolon (`;`) after the statement removes the cell from outputting other prints (that is, something like `<matplotlib.axes._subplots.AxesSubplot at 0x7f455982a490>`).



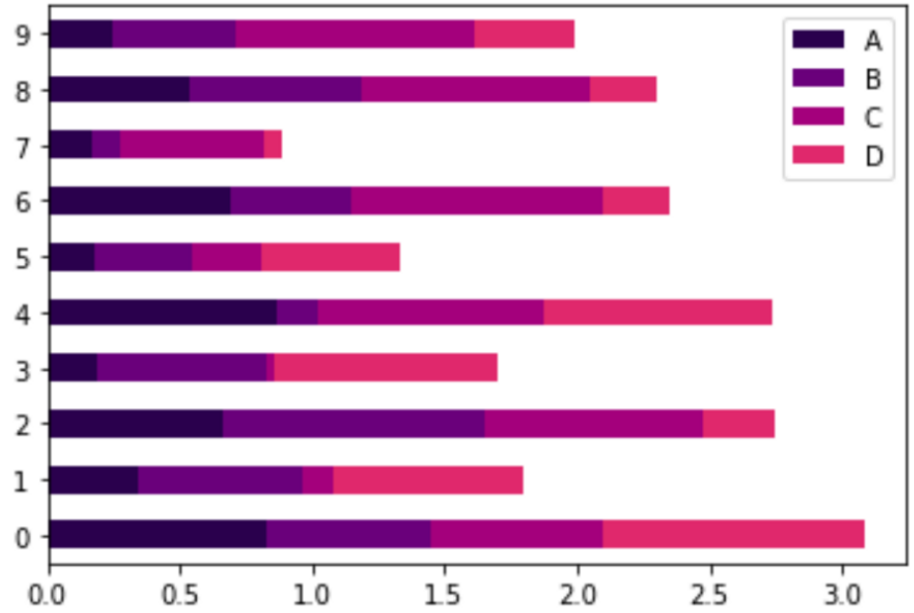
Alternatively, try adding a parameter `stacked=True` — this is a very easy way to create a stacked bar chart straight from the data source.



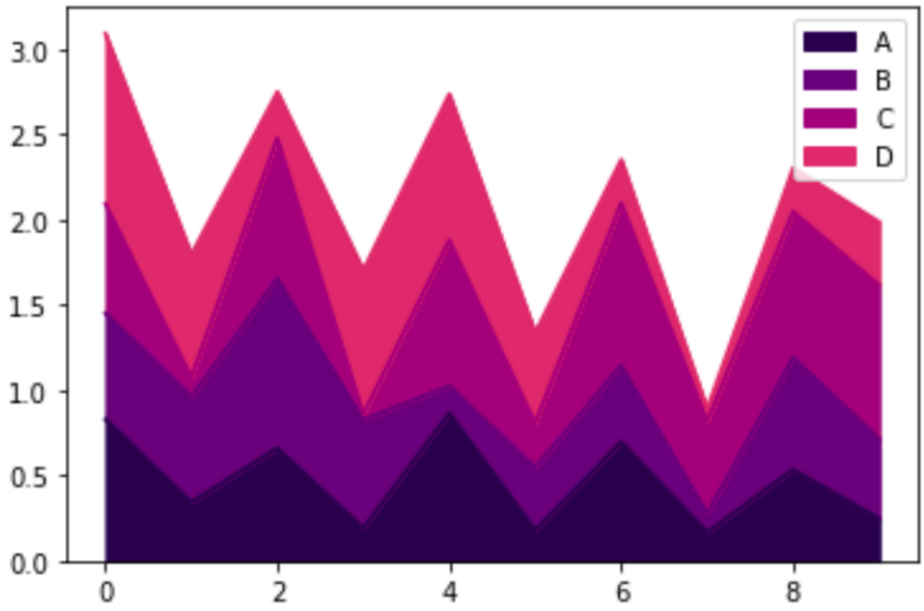
Or, try using `barh`, which draws horizontal bars: `data.plot.barh(stacked=True)`. All these variants can be easily created with only one line of code because they establish a direct flow with the data.



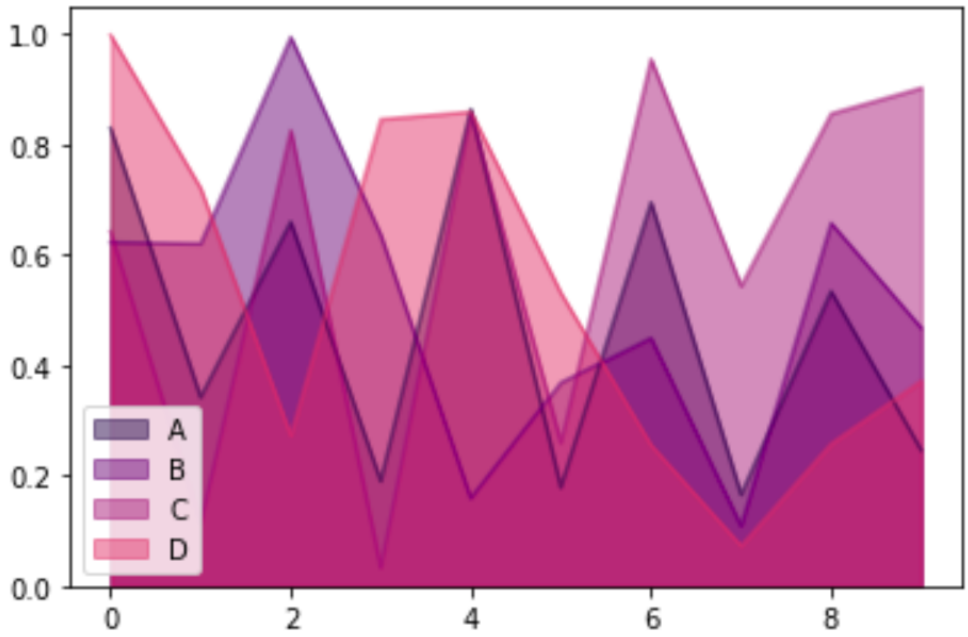
If you choose to, you could change the general palette of the plot by adding `sns.set_palette('magma')` before the code for the plot. Alternatively, you could pass in a color map argument into the plot.



Another method of displaying this type of data would be an area chart, using `data.plot.area();`.



Parameters within the code can be adjusted as they would normally would with a matplotlib or seaborn model. In the case of `data.plot.area(stacked=False);`, the parameter `alpha` (transparency) is set to 0.5 by default, but can be manually adjusted.

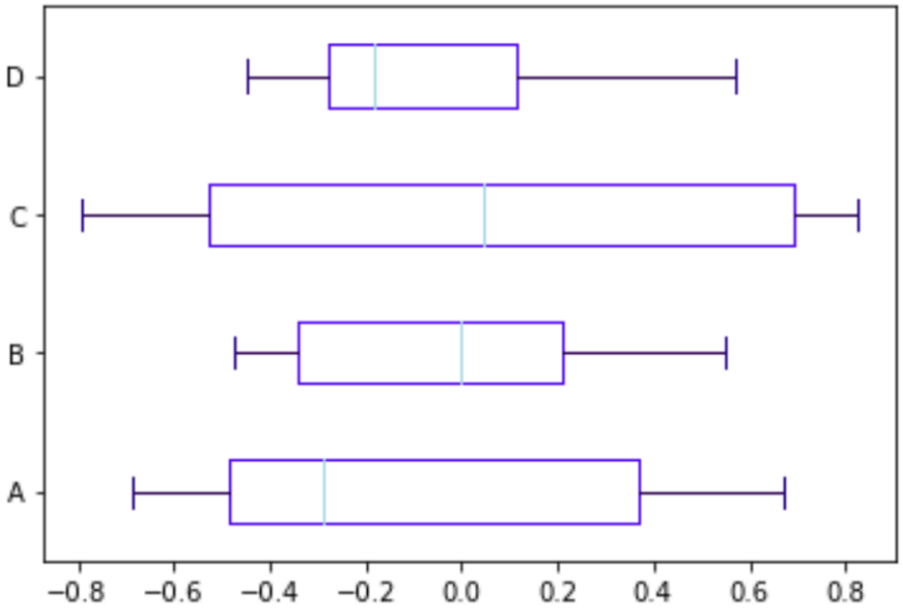


One of the major benefits of using pandas directly is that many of pandas' helpful DataFrame manipulations can be directly used. For example, consider the result of `data.diff()`, which simply takes the difference between one row and the row before it (hence the presence of NaN in the first row). This is helpful in many time series

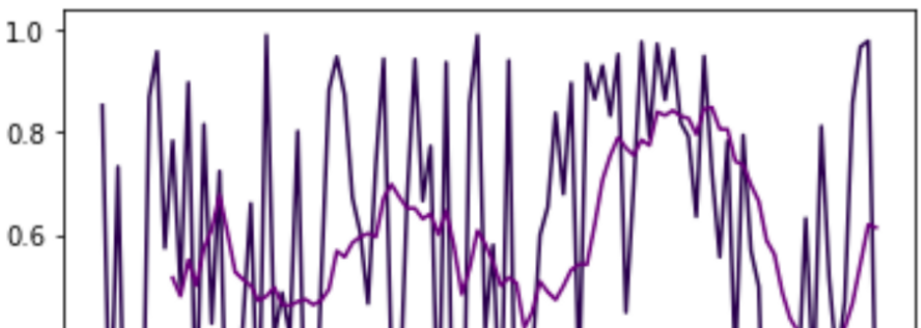
applications.

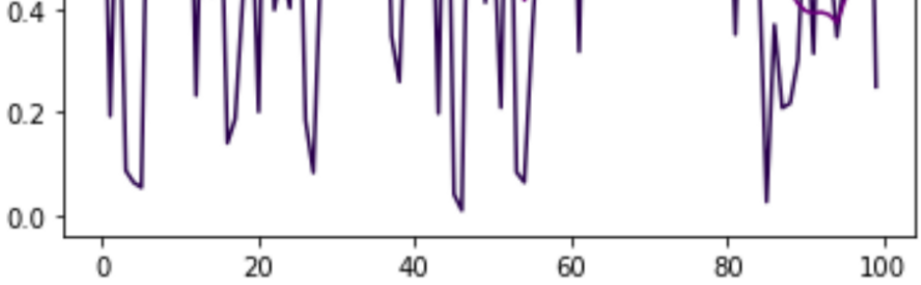
	A	B	C	D
0	NaN	NaN	NaN	NaN
1	-0.487299	-0.002544	-0.527328	-0.277464
2	0.317320	0.374038	0.710174	-0.447428
3	-0.468871	-0.358247	-0.791459	0.572515
4	0.672985	-0.476466	0.822608	0.013580

For example, consider the following code, which plots out the differenced data, as well as demonstrating the usage of various parameters in pandas plotting, in this case, color:



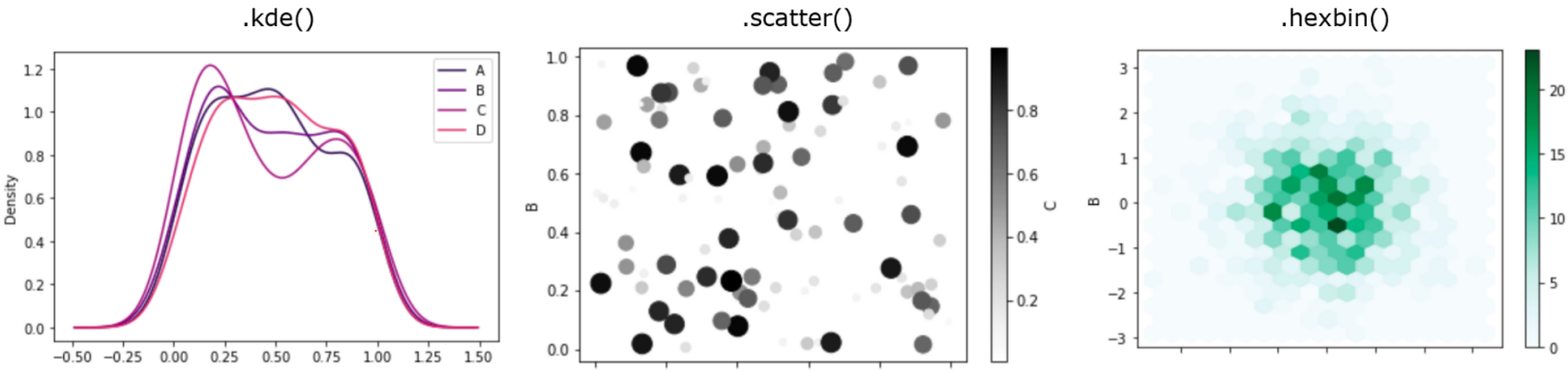
Another application of pandas' handy data manipulation functions is with `.rolling().mean()`, which takes the average rolling mean, a common statistical method to reduce the noisiness of data that averages moving windows of data.





There are various other types of plots that can be created directly from the data:

- `kde` or `density` for density plots
- `scatter` for scatter plots
- `hexbin` for hexagonal bin plots



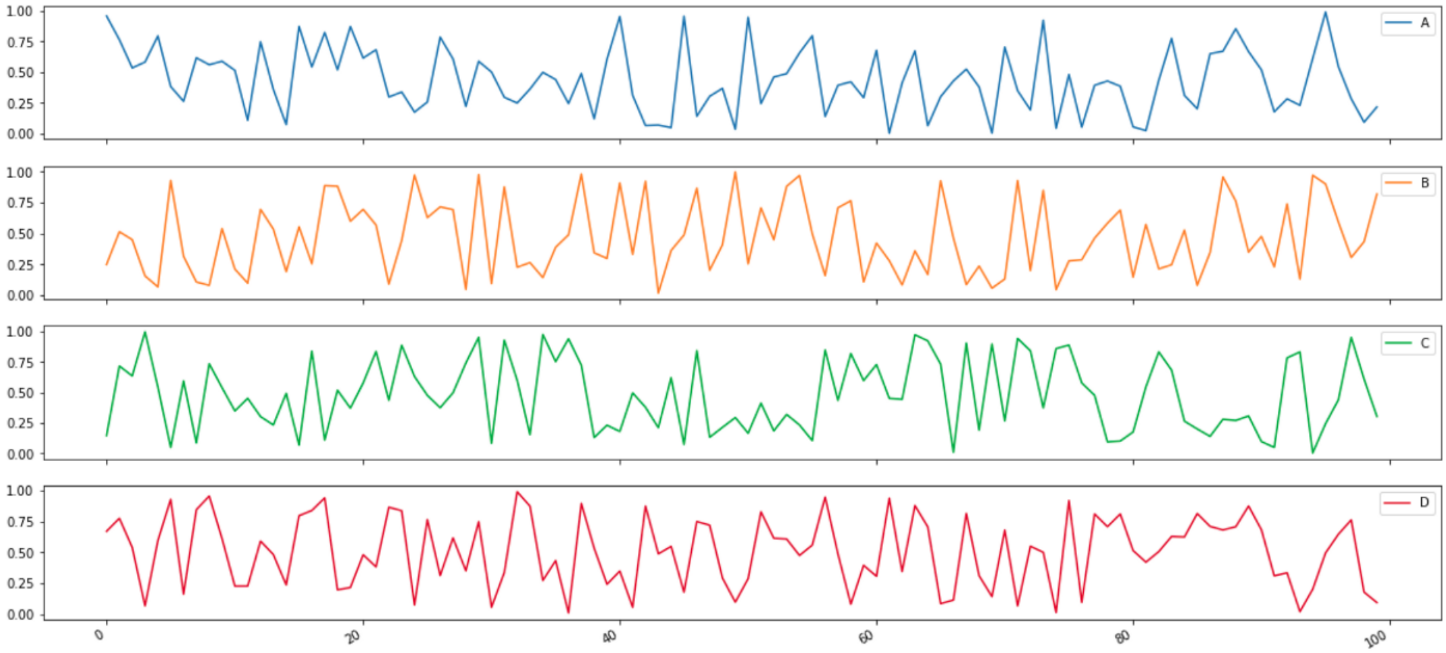
Note: Although color for `.plot.scatter()` was greyscale by default, one could have passed in a color map argument. All plots have a `figsize=(x,y)` argument as well to allow for control of the size of the outputted figure. Putting a semicolon (`;`) after each plotting line allows for multiple outputs in Jupyter Notebook.

One example of pandas doing heavy lifting in plotting for you is with subplots. By enabling `subplots=True`, pandas automatically create subplots based on the columns. For instance, consider the following generated DataFrame, which has two columns (`X` and `Y`), as well as five rows (indices `A`, `B`, `C`, `D`, `E`) — this pie chart will generate two pie charts, each with five sections.

Normally, you would need to create two subplots by hand. Especially in a case where you would want to generate several subplots, one can imagine how it would be helpful to directly use pandas plotting methods.

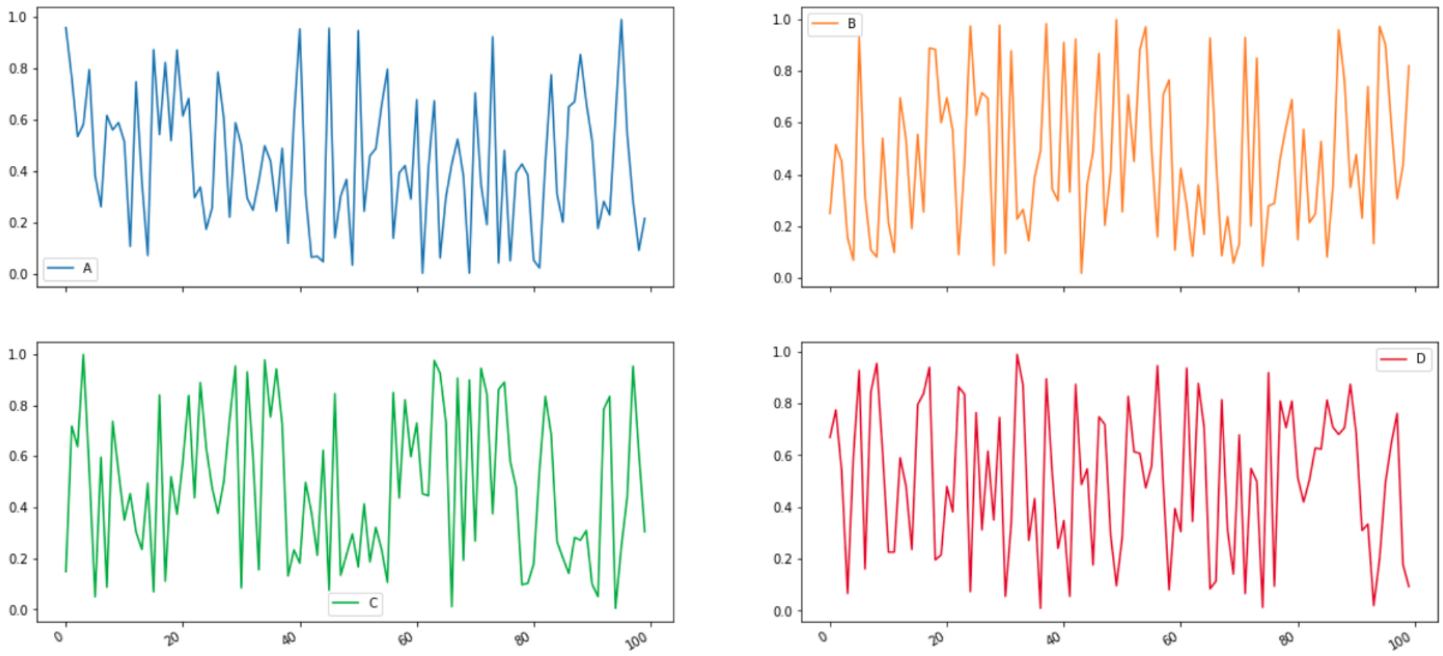
Other parameters for the pie chart include `labels=['label1','label2']`, which adds custom labels to the slices; `colors=['red','green']`, which specifies the color of each slice; `autopct='%.2f'`, which determines the percent labels and the degree of truncation of decimals; and `fontsize=20`, which determines the size of the labels.

As another example of the convenience of subplots in direct-pandas plotting, consider plotting line data (the default when using `.plot()`):



Consider the result when adding a parameter `layout=(2, 2)` in the line of code that plots the visualization (after `subplots=True`): pandas automatically formats the subplots in a format according to the layout. The dimensions of each subplot are determined by the `figsize` argument, which specifies the size of the "master-plot" that

encompasses all subplots.



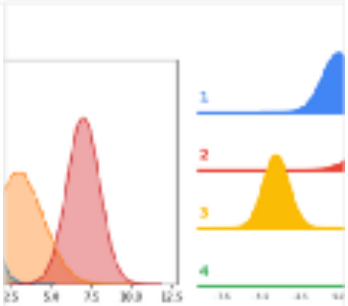
Given the extensive amount of parameters pandas offers in direct plotting — from error bars to offering table displays, there is almost no loss in freedom in terms of what visualizations you can create. From offering simple plot variants to easy subplot management, pandas has so much to offer plotting.

The next time you're producing a simple data analysis plot, try out pandas! You may be shocked how much more efficiently you are able to visualize.

Sorry, But sns.distplot() Just Isn't Good Enough. This is, Though

Beautiful Plots Are Never Created in One Line of Code

towardsdatascience.com



Your Ultimate Python Visualization Cheat-Sheet

Create beautiful, customizable plots easily

towardsdatascience.com

All images created by Author.

Data Science | Visualization | Data Analysis | Programming | Computer Science