Chapter 9: Plotting and Visualization

Dinh Viet Hoang

Department of Computer Science
Faculty of Information Technology
DATCOM Lab

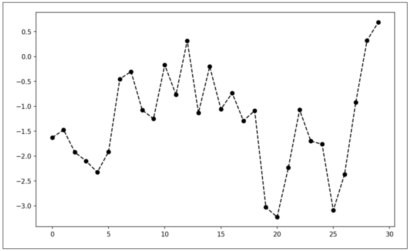
June 17, 2024

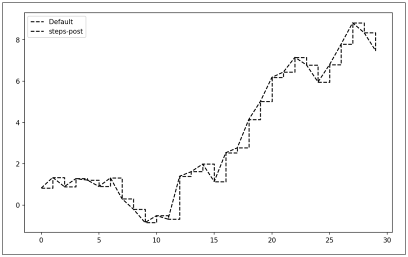
Outline

Introduction

Introduction

Making informative visualizations (sometimes called plots) is one of the most important tasks in data analysis. Python has many add-on libraries for making static or dynamic visualizations, but we will be mainly focused on matplotlib and libraries that build on top of it.





Essential concept

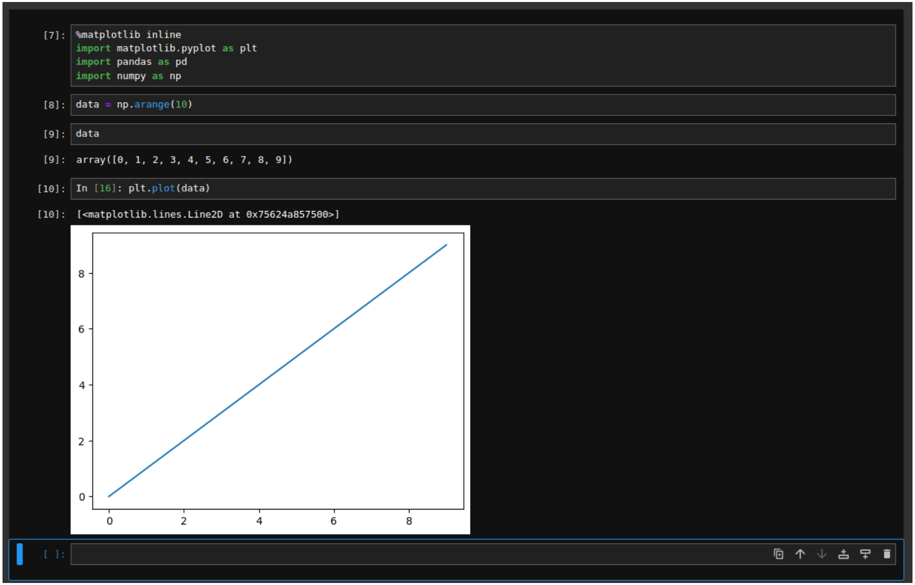
9.1 A Brief matplotlib API Primer

With matplotlib, we use the following import convention:

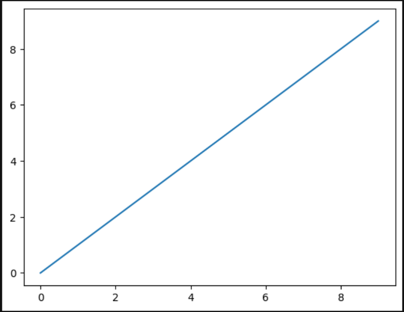
```python
import matplotlib.pyplot as plt
```

After running %matplotlib notebook in Jupyter (or simply %matplotlib in IPython), we can try creating a simple plot.

9.1 A Brief matplotlib API Primer



```
[7]: %matplotlib inline
     import matplotlib.pyplot as plt
     import pandas as pd
     import numpy as np

[8]: data = np.arange(10)

[9]: data

[9]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[10]: In [16]: plt.plot(data)

[10]: [<matplotlib.lines.Line2D at 0x75624a857500>]
```



```
[ ]:
```

9.1.1 Figures and Subplots

Plots in matplotlib reside within a Figure object. You can create a new figure with plt.figure:

```
    fig = plt.figure()
```

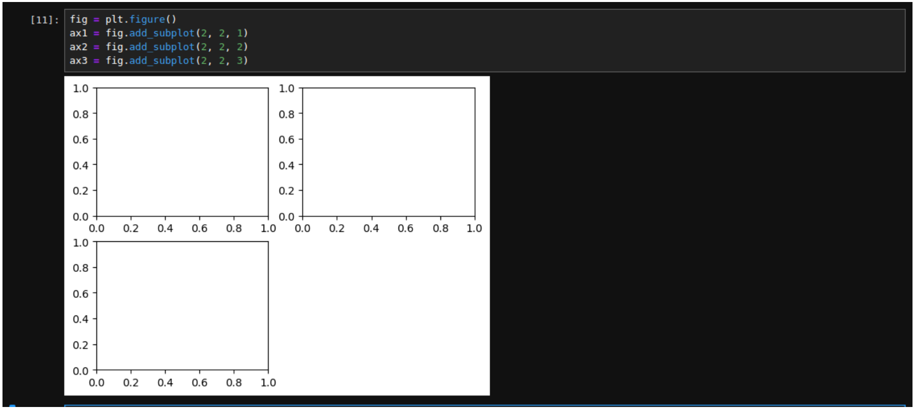
plt.figure has a number of options; notably, figsize will guarantee the figure has a certain size and aspect ratio if saved to disk. You can't make a plot with a blank figure. You have to create one or more subplots using `add_subplot`:

```
    ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2 x 2 (so up to four plots in total), and we are selecting the first of four subplots (numbered from 1),(third argument).

9.1.1 Figures and Subplots

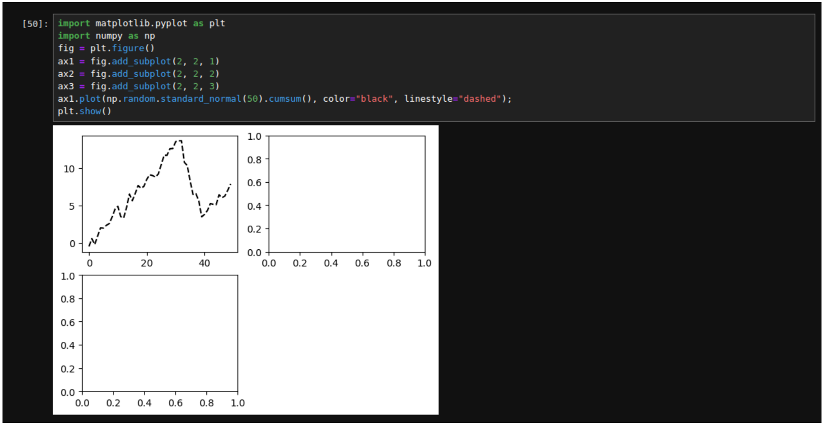
Here we run all of these commands in the same cell:



```
[11]: fig = plt.figure()
      ax1 = fig.add_subplot(2, 2, 1)
      ax2 = fig.add_subplot(2, 2, 2)
      ax3 = fig.add_subplot(2, 2, 3)
```

9.1.1 Figures and Subplots

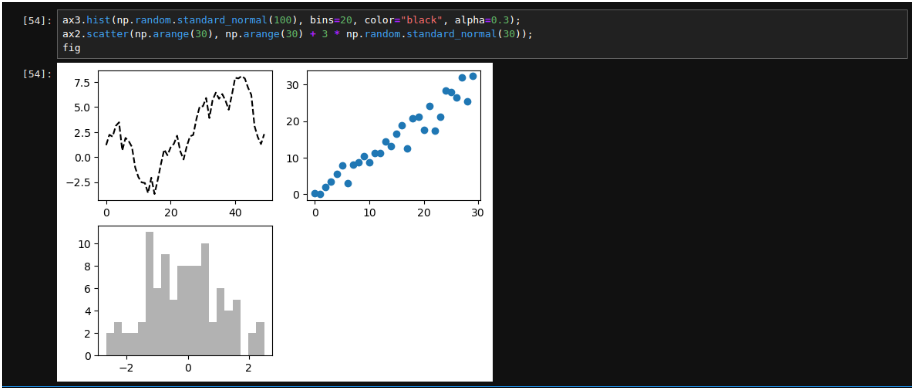
These plot axis objects have various methods that create different types of plots, and it is preferred to use the axis methods over the top-level plotting functions like `p0lt.plot`. For example, we could make a line plot with the plot method:



```
[50]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure()
      ax1 = fig.add_subplot(2, 2, 1)
      ax2 = fig.add_subplot(2, 2, 2)
      ax3 = fig.add_subplot(2, 2, 3)
      ax1.plot(np.random.standard_normal(50).cumsum(), color="black", linestyle="dashed");
      plt.show()
```

9.1.1 Figures and Subplots

The additional options instruct matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` here are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance method:

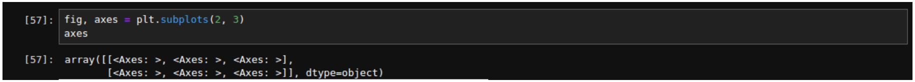


```
[54]: ax3.hist(np.random.standard_normal(100), bins=20, color="black", alpha=0.3);
      ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.standard_normal(30));
      fig
```

```
[54]:
```

9.1.1 Figures and Subplots

To make creating a grid of subplots more convenient, matplotlib includes a plt.sub plots method that creates a new figure and returns a NumPy array containing the created subplot objects:



```
[57]: fig, axes = plt.subplots(2, 3)
      axes

[57]: array([[<Axes: >, <Axes: >, <Axes: >],
             [<Axes: >, <Axes: >, <Axes: >]], dtype=object)
```

The axes array can then be indexed like a two-dimensional array; for example, axes[0, 1] refers to the subplot in the top row at the center. You can also indicate that subplots should have the same x- or y-axis using `sharex` and `sharey`, respectively. This can be useful when you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently.

9.1.1 Figures and Subplots

See Table 9-1 for more on this method.

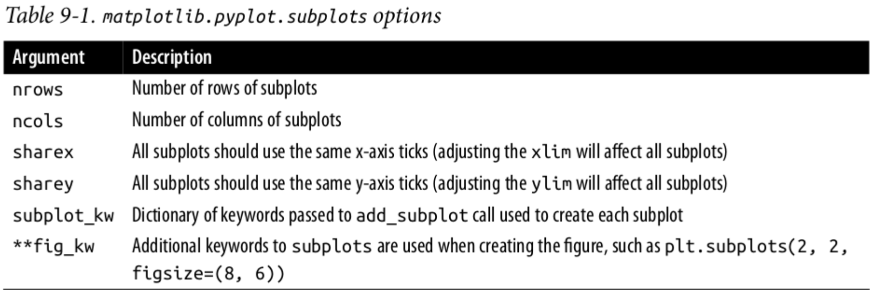


Table 9-1. `matplotlib.pyplot.subplots` options

Argument	Description
`nrows`	Number of rows of subplots
`ncols`	Number of columns of subplots
`sharex`	All subplots should use the same x-axis ticks (adjusting the `xlim` will affect all subplots)
`sharey`	All subplots should use the same y-axis ticks (adjusting the `ylim` will affect all subplots)
`subplot_kw`	Dictionary of keywords passed to `add_subplot` call used to create each subplot
`**fig_kw`	Additional keywords to `subplots` are used when creating the figure, such as `plt.subplots(2, 2, figsize=(8, 6))`

9.1.1 Figures and Subplots

Adjusting the spacing around subplots

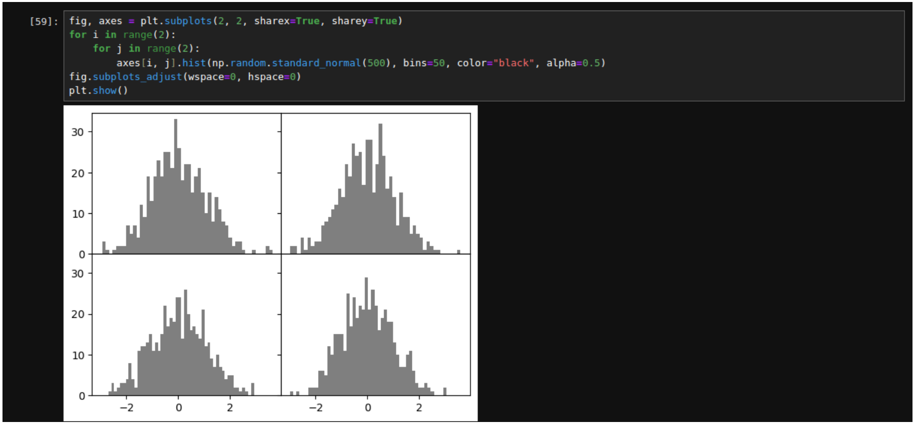
By default, matplotlib leaves a certain amount of padding around the outside of the subplots and in spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. You can change the spacing using the `subplots_adjust` method on `Figure` objects:

```
subplots_adjust(left=None, bottom=None,
    right=None, top=None, wspace=None,
    hspace=None)
```

`wspace` and `hspace` control the percent of the figure width and figure height, respectively, to use as spacing between subplots.

9.1.1 Figures and Subplots

There is an example:



```
[59]: fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
      for i in range(2):
          for j in range(2):
              axes[i, j].hist(np.random.standard_normal(500), bins=50, color="black", alpha=0.5)
      fig.subplots_adjust(wspace=0, hspace=0)
      plt.show()
```

9.1.2 Colors, Markers, and Line Styles

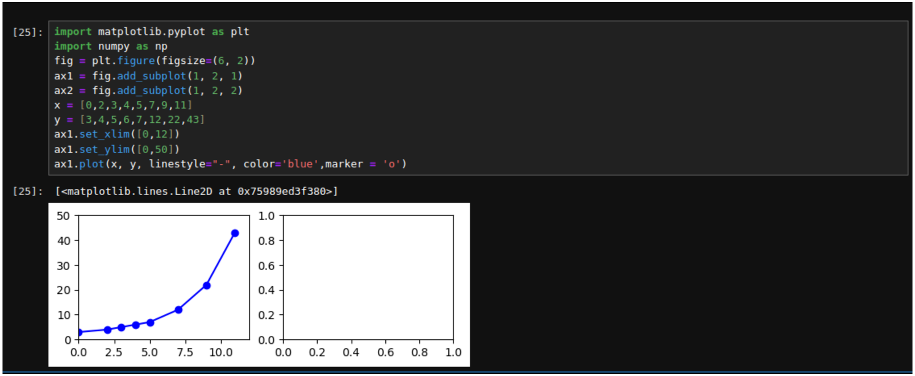
matplotlib's line plot function accepts arrays of x and y coordinates and optional color styling options. For example, to plot x versus y with green dashes, you would execute:

```
ax.plot(x, y, linestyle="--", color="green")
```

A number of color names are provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., "#CECECE"). You can see some of the supported line styles by looking at the docstring for plt.plot (use plt.plot? in IPython or Jupyter). A more comprehensive reference is available in the online documentation.

9.1.2 Colors, Markers, and Line Styles

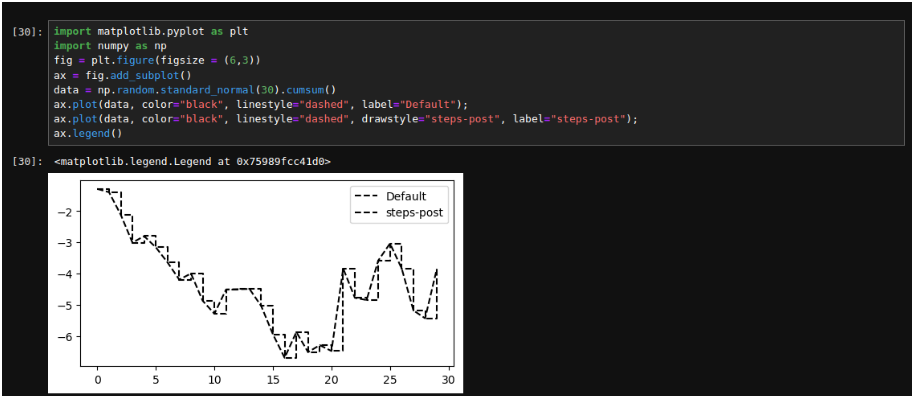
Line plots can additionally have markers to highlight the actual data points. Since matplotlib's plot function creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be supplied as an additional styling option:



```
[25]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure(figsize=(6, 2))
      ax1 = fig.add_subplot(1, 2, 1)
      ax2 = fig.add_subplot(1, 2, 2)
      x = [0,2,3,4,5,7,9,11]
      y = [3,4,5,6,7,12,22,43]
      ax1.set_xlim([0,12])
      ax1.set_ylim([0,50])
      ax1.plot(x, y, linestyle="-", color='blue',marker = 'o')

[25]: [<matplotlib.lines.Line2D at 0x75989ed3f380>]
```

9.1.2 Colors, Markers, and Line Styles

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option:



```
[30]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure(figsize = (6,3))
      ax = fig.add_subplot()
      data = np.random.standard_normal(30).cumsum()
      ax.plot(data, color="black", linestyle="dashed", label="Default");
      ax.plot(data, color="black", linestyle="dashed", drawstyle="steps-post", label="steps-post");
      ax.legend()

[30]: <matplotlib.legend.Legend at 0x75989fcc41d0>
```

9.1.3 Ticks, Labels, and Legends

Most kinds of plot decorations can be accessed through methods on matplotlib axes objects. This includes methods like `xlim`, `xticks`, and `xticklabels`. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

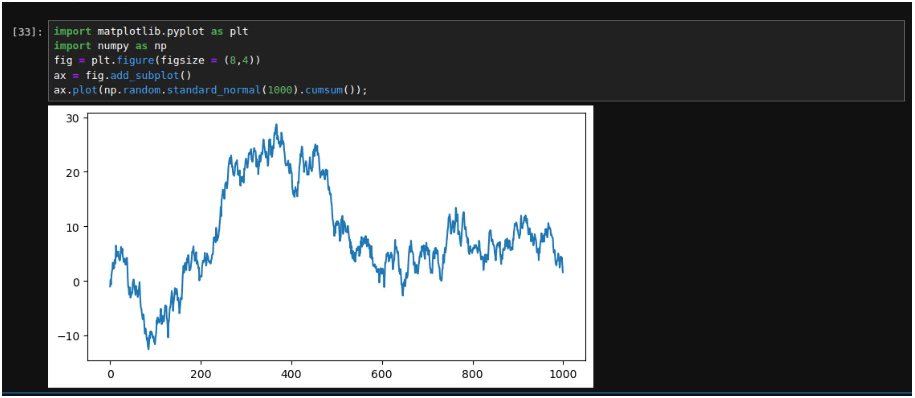
- Called with no arguments returns the current parameter value (e.g., `ax.xlim()` returns the current xaxis plotting range)
- Called with parameters sets the parameter value (e.g., `ax.xlim([0, 10])` sets the xaxis range to 0 to 10)

All such methods act on the active or most recently created AxesSubplot. Each corresponds to two methods on the subplot object itself; in the case of `xlim`, these are `ax.get_xlim` and `ax.set_xlim`.

9.1.3 Ticks, Labels, and Legends

Setting the title, axis labels, ticks, and tick labels

To illustrate customizing the axes, we will create a simple figure and plot of a random walk:



```
[33]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure(figsize = (8,4))
      ax = fig.add_subplot()
      ax.plot(np.random.standard_normal(1000).cumsum());
```

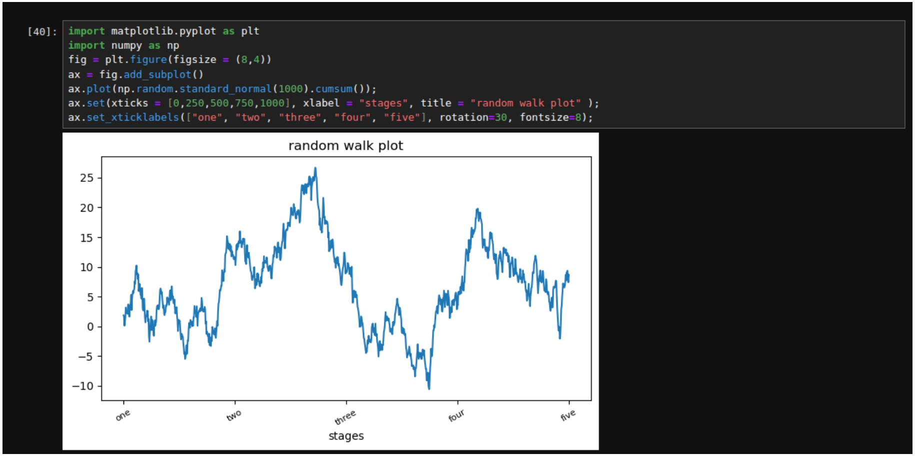
9.1.3 Ticks, Labels, and Legends

To change the x-axis ticks, it's easiest to use set_xticks and set_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using set_xticklabels:

```
ax.set_xticks([0, 250, 500, 750, 1000]);
ax.set_xticklabels(["one", "two", "three",
   "four", "five"], rotation=30, fontsize=8);
```

9.1.3 Ticks, Labels, and Legends

The rotation option sets the x tick labels at a 30-degree rotation. Lastly, `set_xlabel` gives a name to the x-axis, and `set_title` is the subplot title:

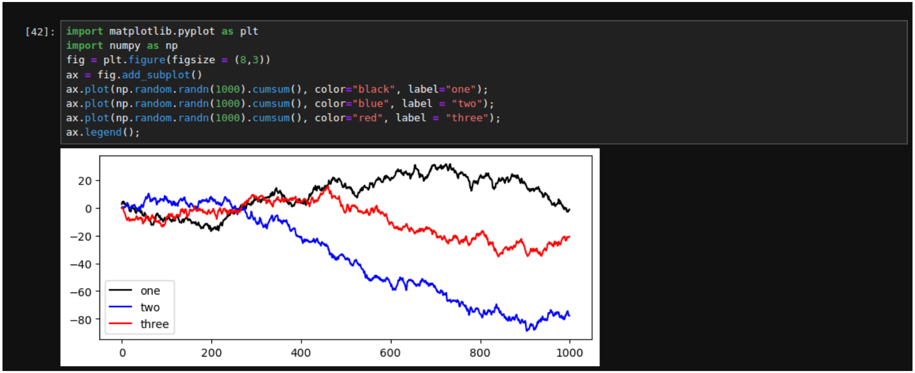


```
[40]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure(figsize = (8,4))
      ax = fig.add_subplot()
      ax.plot(np.random.standard_normal(1000).cumsum());
      ax.set(xticks = [0,250,500,750,1000], xlabel = "stages", title = "random walk plot" );
      ax.set_xticklabels(["one", "two", "three", "four", "five"], rotation=30, fontsize=8);
```

9.1.3 Ticks, Labels, and Legends

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the label argument when adding each piece of the plot. Once you have done this, you can call `ax.legend()` to automatically create a legend:



```
[42]: import matplotlib.pyplot as plt
      import numpy as np
      fig = plt.figure(figsize = (8,3))
      ax = fig.add_subplot()
      ax.plot(np.random.randn(1000).cumsum(), color="black", label="one");
      ax.plot(np.random.randn(1000).cumsum(), color="blue", label = "two");
      ax.plot(np.random.randn(1000).cumsum(), color="red", label = "three");
      ax.legend();
```

9.1.3 Ticks, Labels, and Legends

The legend method has several other choices for the location `loc` argument. See the docstring (with ax.legend?) for more information.

The loc legend option tells matplotlib where to place the plot. The default is "best", which tries to choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or label= `"_nolegend_"`.

9.1.4 Annotations and Drawing on a Subplot

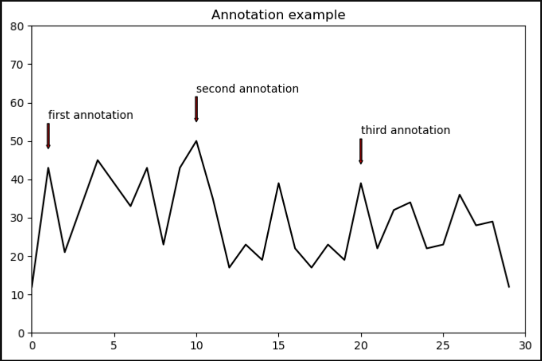
In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes. You can add annotations and text using the text, arrow, and annotate functions. text draws text at given coordinates (x, y) on the plot with optional custom styling:

```
ax.text(x, y, "Hello world!",
   family="monospace", fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. Let's see an example:

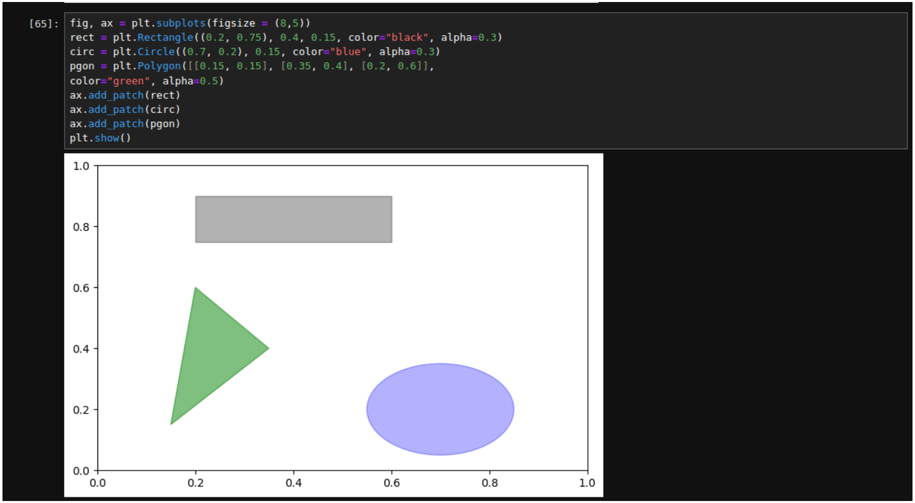
9.1.4 Annotations and Drawing on a Subplot

```
[62]: import pandas as pd
fig, ax = plt.subplots(figsize = (8,5))
data = pd.Series([12,43,21,33,45,39,33, 43,23, 43, 50, 35, 17, 23 ,19 ,39,22,17, 23 ,19 ,39,22, 32, 34, 22, 23, 36, 28, 29, 12])
data.plot(ax=ax, color="black")
annot = [ (1, "first annotation"), (10, "second annotation"), (20, "third annotation")]
for i, label in annot :
    ax.annotate(label, xy = (i, data[i] + 5), xytext = (i, data[i] + 15), arrowprops=dict(facecolor="red", headwidth=3, width=1.5,
headlength=3, shrink = 0), horizontalalignment="left", verticalalignment="top")
ax.set_xlim([0, 30])
ax.set_ylim([0, 80])
ax.set_title("Annotation example")
plt.show();
```



9.1.4 Annotations and Drawing on a Subplot

To add a shape to a plot, you create the patch object and add it to a subplot ax by passing the patch to `ax.add_patch`



```
[65]: fig, ax = plt.subplots(figsize = (8,5))
      rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color="black", alpha=0.3)
      circ = plt.Circle((0.7, 0.2), 0.15, color="blue", alpha=0.3)
      pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
      color="green", alpha=0.5)
      ax.add_patch(rect)
      ax.add_patch(circ)
      ax.add_patch(pgon)
      plt.show()
```

9.1.5 Saving Plots to File

You can save the active figure to file using the figure object's `savefig` instance method. For example, to save an SVG version of a figure, you need only type:

```
fig.savefig("figpath.svg")
```

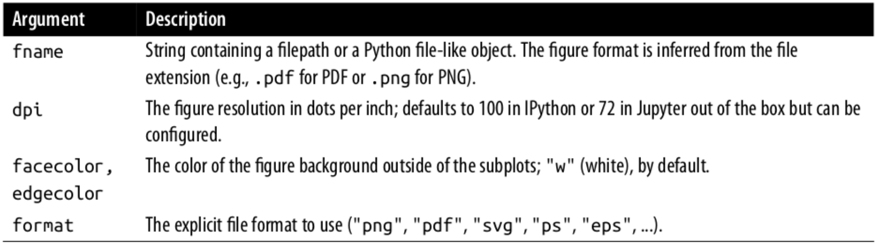
The file type is inferred from the file extension. So if you used .pdf instead, you would get a PDF. One important option that I use frequently for publishing graphics is dpi, which controls the dots-per-inch resolution. To get the same plot as a PNG at 400 DPI, you would do:

```
fig.savefig("figpath.png", dpi=400)
```

9.1.5 Saving Plots to File

See Table 9-2 for a list of some other options for savefig. For a comprehensive listing, refer to the docstring in IPython or Jupyter.

Table 9-2. Some `fig.savefig` options



Argument	Description
`fname`	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension (e.g., `.pdf` for PDF or `.png` for PNG).
`dpi`	The figure resolution in dots per inch; defaults to 100 in IPython or 72 in Jupyter out of the box but can be configured.
`facecolor`, `edgecolor`	The color of the figure background outside of the subplots; `"w"` (white), by default.
`format`	The explicit file format to use (`"png"`, `"pdf"`, `"svg"`, `"ps"`, `"eps"`, ...).

9.1.6 matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primar- ily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to modify the configuration programmatically from Python is to use the `rc` method; for example, to set the global default figure size to be 10 x 10, you could enter:

```
plt.rc("figure", figsize=(10, 10))
```

All of the current configuration settings are found in the `plt.rcParams` dictionary, and they can be restored to their default values by calling the `plt.rcdefaults()` function.

9.1.6 matplotlib Configuration

The first argument to `rc` is the component you wish to customize, such as `"figure"`, `"axes"`, `"xtick"`, `"ytick"`, `"grid"`, `"legend"`, or many others. After that can follow a sequence of keyword arguments indicating the new parameters. A convenient way to write down the options in your program is as a dictionary:

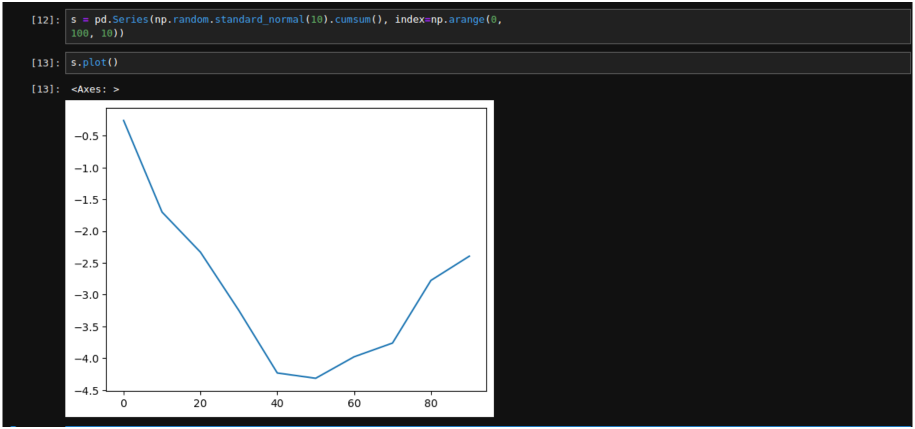
```
plt.rc("font", family="monospace",
   weight="bold", size=8)
```

9.2 Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base components: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations. In pandas, we may have multiple columns of data, along with row and column labels. pandas itself has built-in methods that simplify creating visualizations from DataFrame and Series objects. Another library is `seaborn`, a high-level statistical graphics library built on matplotlib. seaborn simplifies creating many common visualization types.

9.2.1 Line Plots

Series and DataFrame have a plot attribute for making some basic plot types. By default, plot() makes line plots:



```
[12]: s = pd.Series(np.random.standard_normal(10).cumsum(), index=np.arange(0,
      100, 10))

[13]: s.plot()

[13]: <Axes: >
```

9.2.1 Line Plots

The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing `use_index = False`. The x-axis ticks and limits can be adjusted with the texttxticks and `xlim` options, and the y-axis respectively with `yticks` and `ylim`.See Table 9-3 for a partial listing of plot options.

9.2.1 Line Plots

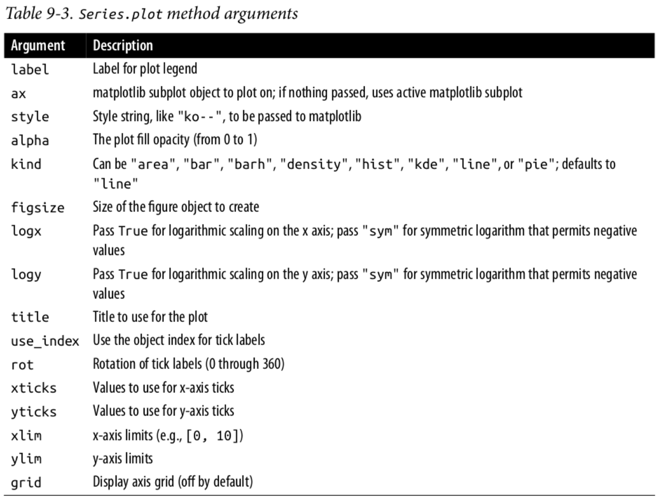
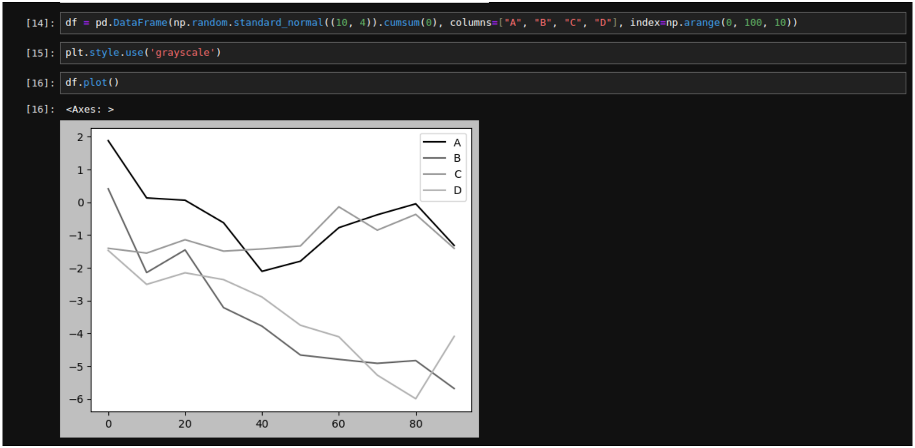


Table 9-3. `Series.plot` method arguments

Argument	Description
`label`	Label for plot legend
`ax`	matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot
`style`	Style string, like `"ko--"`, to be passed to matplotlib
`alpha`	The plot fill opacity (from 0 to 1)
`kind`	Can be `"area"`, `"bar"`, `"barh"`, `"density"`, `"hist"`, `"kde"`, `"line"`, or `"pie"`; defaults to `"line"`
`figsize`	Size of the figure object to create
`logx`	Pass `True` for logarithmic scaling on the x axis; pass `"sym"` for symmetric logarithm that permits negative values
`logy`	Pass `True` for logarithmic scaling on the y axis; pass `"sym"` for symmetric logarithm that permits negative values
`title`	Title to use for the plot
`use_index`	Use the object index for tick labels
`rot`	Rotation of tick labels (0 through 360)
`xticks`	Values to use for x-axis ticks
`yticks`	Values to use for y-axis ticks
`xlim`	x-axis limits (e.g., `[0, 10]`)
`ylim`	y-axis limits
`grid`	Display axis grid (off by default)

9.2.1 Line Plots

DataFrame's plot method plots each of its columns as a different line on the same subplot, creating a legend automatically:



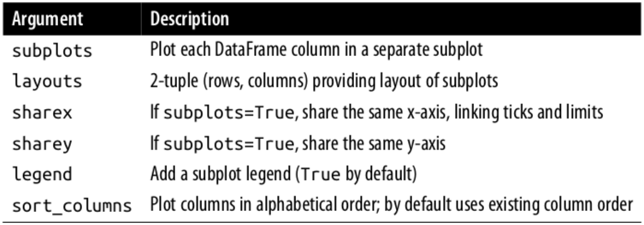
```
[14]: df = pd.DataFrame(np.random.standard_normal((10, 4)).cumsum(0), columns=["A", "B", "C", "D"], index=np.arange(0, 100, 10))

[15]: plt.style.use('grayscale')

[16]: df.plot()

[16]: <Axes: >
```

9.2.1 Line Plots

The plot attribute contains a "family" of methods for different plot types. For example, df.plot() is equivalent to df.plot.line().

DataFrame has a number of options allowing some flexibility for how the columns are handled, for example, whether to plot them all on the same subplot or to create separate subplots. See Table 9-4 for more on these.

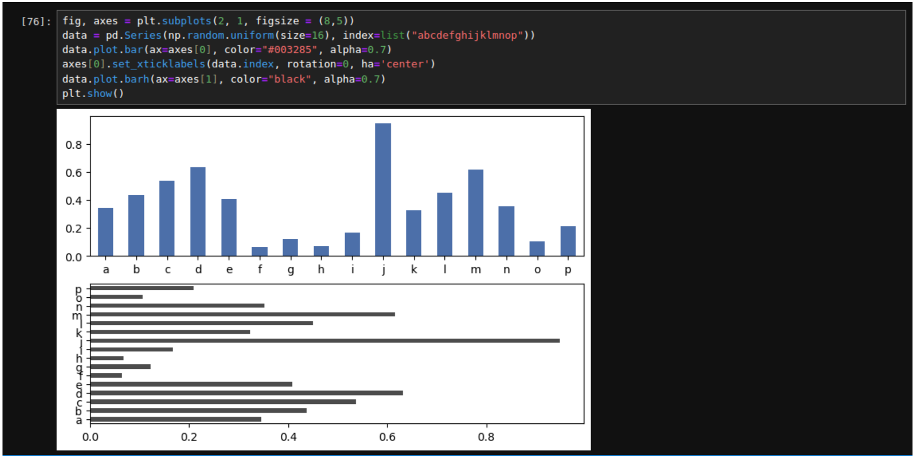
Table 9-4. DataFrame-specific plot arguments



Argument	Description
`subplots`	Plot each DataFrame column in a separate subplot
`layouts`	2-tuple (rows, columns) providing layout of subplots
`sharex`	If `subplots=True`, share the same x-axis, linking ticks and limits
`sharey`	If `subplots=True`, share the same y-axis
`legend`	Add a subplot legend (`True` by default)
`sort_columns`	Plot columns in alphabetical order; by default uses existing column order

9.2.2 Bar Plots

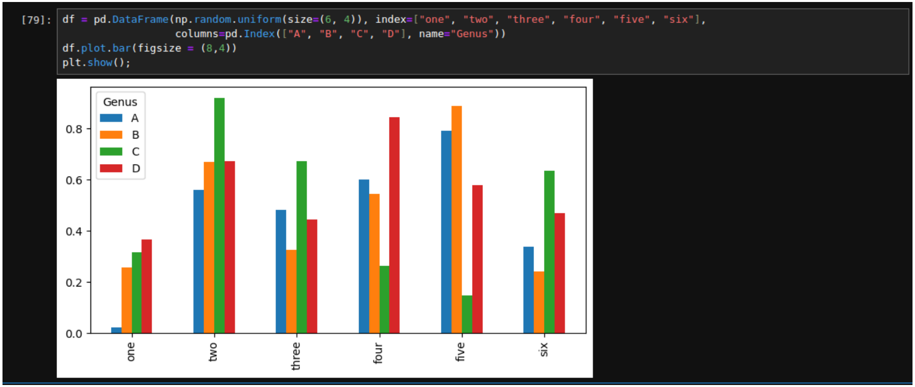
The `plot.bar()` and `plot.barh()` make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the x (`bar`) or y (`barh`) ticks:



```
[76]: fig, axes = plt.subplots(2, 1, figsize = (8,5))
      data = pd.Series(np.random.uniform(size=16), index=list("abcdefghijklmnop"))
      data.plot.bar(ax=axes[0], color="#003285", alpha=0.7)
      axes[0].set_xticklabels(data.index, rotation=0, ha='center')
      data.plot.barh(ax=axes[1], color="black", alpha=0.7)
      plt.show()
```

9.2.2 Bar Plots

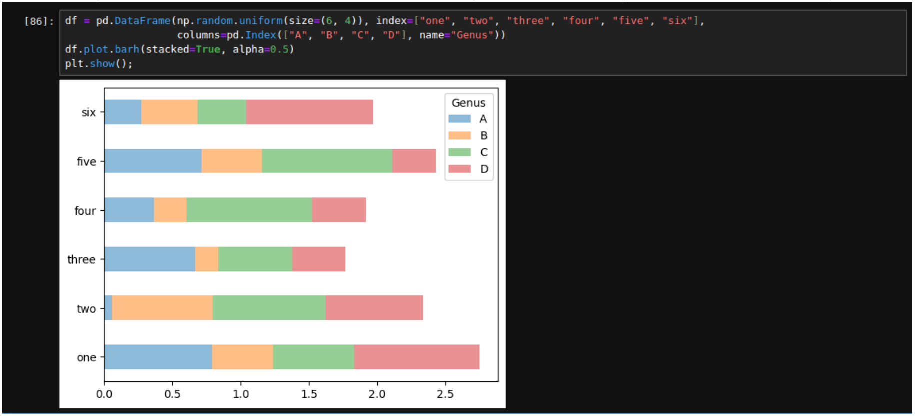
With a DataFrame, bar plots group the values in each row in bars, side by side, for each value:



```
[79]: df = pd.DataFrame(np.random.uniform(size=(6, 4)), index=["one", "two", "three", "four", "five", "six"],
                        columns=pd.Index(["A", "B", "C", "D"], name="Genus"))
      df.plot.bar(figsize = (8,4))
      plt.show();
```

9.2.2 Bar Plots

We create stacked bar plots from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together horizontally:

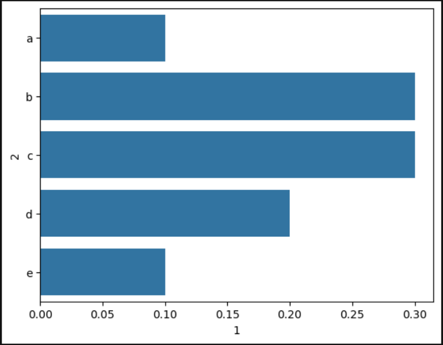


```
[86]: df = pd.DataFrame(np.random.uniform(size=(6, 4)), index=["one", "two", "three", "four", "five", "six"],
                        columns=pd.Index(["A", "B", "C", "D"], name="Genus"))
      df.plot.barh(stacked=True, alpha=0.5)
      plt.show();
```

9.2.2 Bar Plots

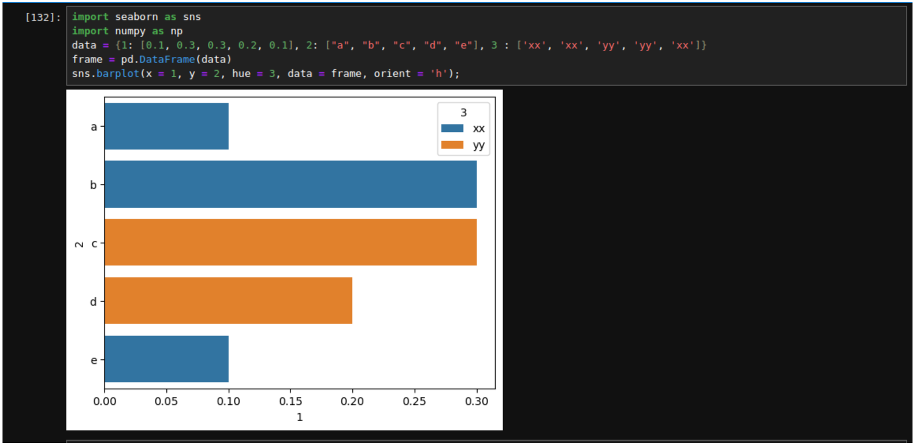
With data that requires aggregation or summarization before making a plot, using the seaborn package can make things much simpler:

```
[131]: import seaborn as sns
       import numpy as np
       data = {1: [0.1, 0.3, 0.3, 0.2, 0.1], 2: ["a", "b", "c", "d", "e"]}
       frame = pd.DataFrame(data)
       sns.barplot(x = 1, y = 2, data = frame, orient = 'h');
```



9.2.2 Bar Plots

`seaborn.barplot` has a `hue` option that enables us to split by an additional categorical value:



```
[132]: import seaborn as sns
       import numpy as np
       data = {1: [0.1, 0.3, 0.3, 0.2, 0.1], 2: ["a", "b", "c", "d", "e"], 3 : ['xx', 'xx', 'yy', 'yy', 'xx']}
       frame = pd.DataFrame(data)
       sns.barplot(x = 1, y = 2, hue = 3, data = frame, orient = 'h');
```

9.2.2 Bar Plots

Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors. You can switch between different plot appearances using `seaborn.set_style`:

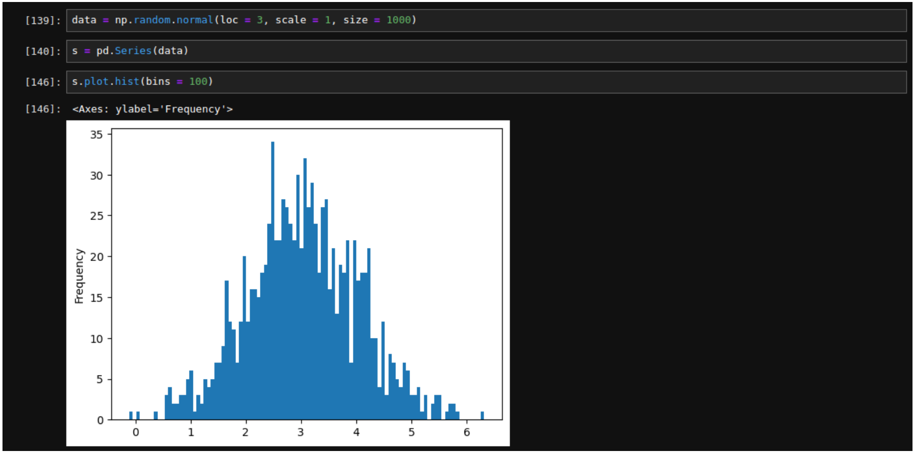
```
sns.set_style("whitegrid")
```

When producing plots for black-and-white print medium, you may find it useful to set a greyscale color palette, like so:

```
sns.set_palette("Greys_r")
```

9.2.3 Histograms and Density Plots

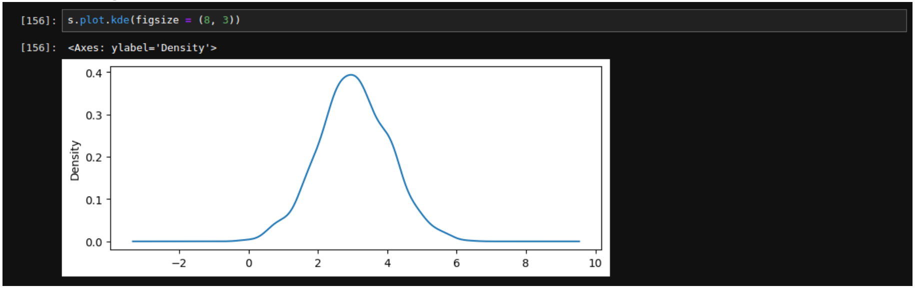
A histogram is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Let's see an example:



```
[139]: data = np.random.normal(loc = 3, scale = 1, size = 1000)

[140]: s = pd.Series(data)

[146]: s.plot.hist(bins = 100)

[146]: <Axes: ylabel='Frequency'>
```

9.2.3 Histograms and Density Plots

A related plot type is a density plot, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. The usual procedure is to approximate this distribution as a mixture of "kernels"-that is, simpler distributions like the normal distribution. Thus, density plots are also known as kernel density estimate (KDE) plots. Using `plot.density/plot.kde` makes a density plot using the conventional mixture-of-normals estimate:



```
[156]: s.plot.kde(figsize = (8, 3))

[156]: <Axes: ylabel='Density'>
```

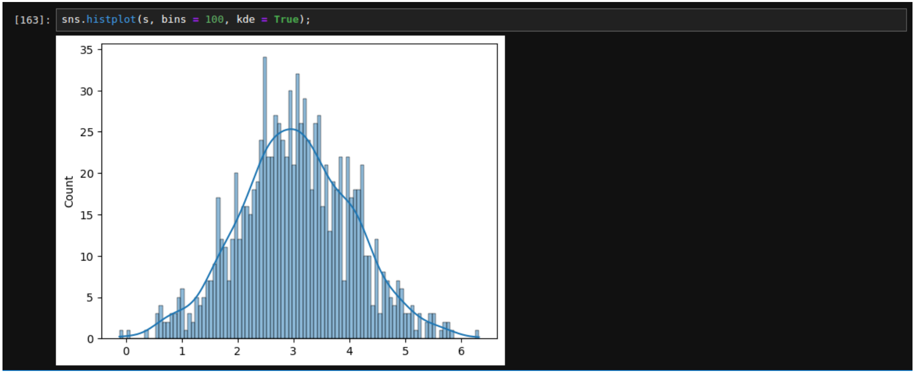
9.2.3 Histograms and Density Plots

This kind of plot requires SciPy, so if you do not have it installed already, you can pause and do that now (if you are using conda):

```
conda install scipy
```

9.2.3 Histograms and Density Plots

`seaborn` makes histograms and density plots even easier through its `histplot` method, which can plot both a histogram and a continuous density estimate simultaneously:

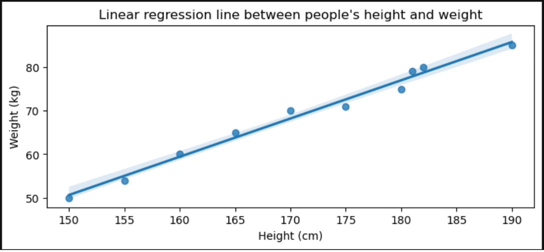


```
[163]: sns.histplot(s, bins = 100, kde = True);
```

9.2.4 Scatter or Point Plots

Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series. For example, if we have some data about height and weight of a group of people, we can then use seaborn's `regplot` method, which makes a scatter plot and fits a linear regression line between height and weight:

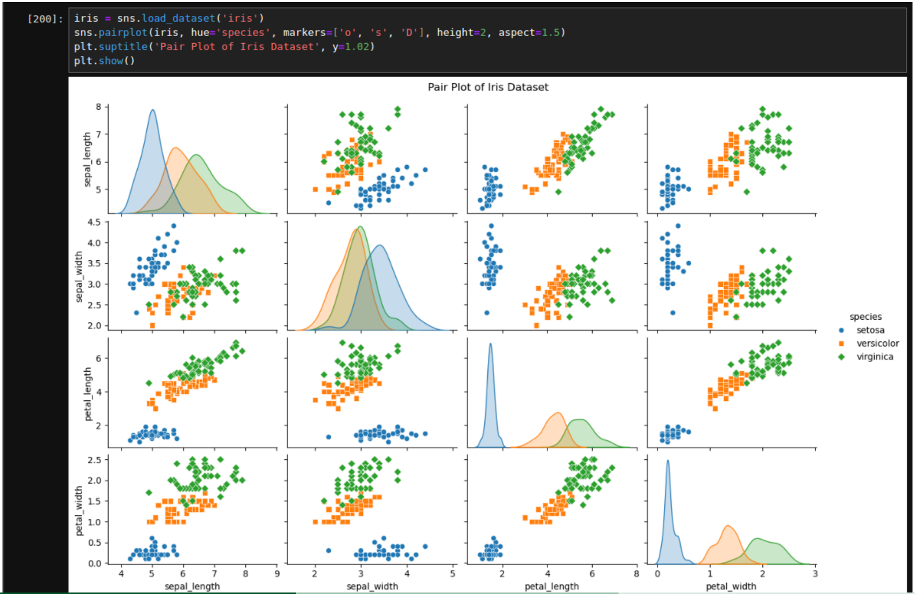
```
[189]: fig,ax = plt.subplots(figsize = (8,3))
       data = pd.DataFrame({'Height': [150, 155, 160, 165, 170, 175, 180, 181, 182, 190],
                            'Weight': [50, 54, 60, 65, 70, 71, 75, 79, 80, 85] })
       sns.regplot(x='Height', y='Weight', data=data)
       ax.set(xlabel = 'Height (cm)', ylabel = 'Weight (kg)', title = 'Linear regression line between people\'s height and weight')
       plt.show()
```



9.2.4 Scatter or Point Plots

In exploratory data analysis, it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so seaborn has a convenient `pairplot` function that supports placing histograms or density estimates of each variable along the diagonal:

9.2.4 Scatter or Point Plots



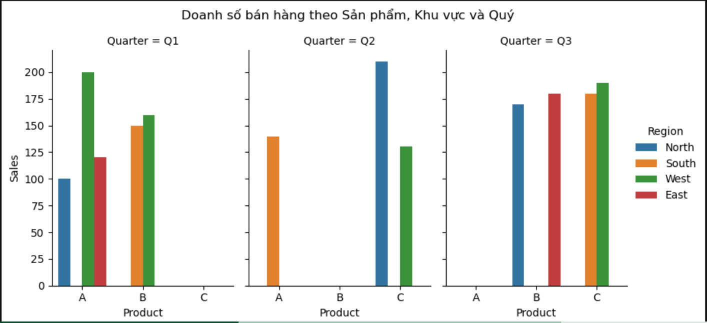
```
[200]: iris = sns.load_dataset('iris')
       sns.pairplot(iris, hue='species', markers=['o', 's', 'D'], height=2, aspect=1.5)
       plt.suptitle('Pair Plot of Iris Dataset', y=1.02)
       plt.show()
```

9.2.5 Facet Grids and Categorical Data

What about datasets where we have additional grouping dimensions? One way to visualize data with many categorical variables is to use a facet grid, which is a two-dimensional layout of plots where the data is split across the plots on each axis based on the distinct values of a certain variable. seaborn has a useful built-in function `catplot` that simplifies making many kinds of faceted plots split by ategorical variables:

9.2.5 Facet Grids and Categorical Data

```
•[10]: import seaborn as sns
       import matplotlib.pyplot as plt
       import pandas as pd
       data = {'Product': ['A', 'B', 'A', 'C', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C'],
               'Region': ['North', 'South', 'West', 'West', 'North', 'South', 'East', 'West', 'North', 'South', 'East', 'West'],
               'Quarter': ['Q1', 'Q1', 'Q1', 'Q2', 'Q3', 'Q3', 'Q1', 'Q1', 'Q2', 'Q2', 'Q3', 'Q3'],
               'Sales': [100, 150, 200, 130, 170, 180, 120, 160, 210, 140, 180, 190]   }
       df = pd.DataFrame(data)
       g = sns.catplot(
           data=df,
           x='Product', y='Sales',
           hue='Region', col='Quarter',
           kind='bar',
           height=4, aspect=0.7
       )
       g.fig.suptitle('Sales due to product, region and quarter', y=1.05)
       plt.show()
```



9.2.5 Facet Grids and Categorical Data

`catplot` supports other plot types that may be useful depending on what you are trying to display. See seaborn/catplot documentation for more.

9.2.5 Facet Grids and Categorical Data

You can create your own facet grid plots using the more general `seaborn.FacetGrid` class. See the seaborn documentation for more.

9.3 Other Python Visualization Tools

As is common with open source, there many options for creating graphics in Python (too many to list). Since 2010, much development effort has been focused on creating interactive graphics for publication on the web. With tools like Altair, Bokeh, and Plotly, it's now possible to specify dynamic, interactive graphics in Python that are intended for use with web browsers.

For creating static graphics for print or web, I recommend using matplotlib and libraries that build on matplotlib, like pandas and seaborn, for your needs. For other data visualization requirements, it may be useful to learn how to use one of the other available tools. I encourage you to explore the ecosystem as it continues to evolve and innovate into the future.

Conclusion

The goal of this chapter was to get your feet wet with some basic data visualization using pandas, matplotlib, and seaborn. If visually communicating the results of data analysis is important in your work, I encourage you to seek out resources to learn more about effective data visualization. It is an active field of research, and you can practice with many excellent learning resources available online.

Thank you