16

THE INFOVIS, SCIVIS, AND DASHBOARDING LIBRARIES

Visualizing data is an integral part of science. Humans are visual creatures by nature, and viewing data graphically is more efficient and intuitive than reading through lists of strings or numbers. Effective plots help you to clean, prepare, and explore data. You can use them to reveal outliers and spurious samples, identify patterns, and compare datasets. Perhaps most important, they help you to communicate clearly with others and convey your ideas in an easily consumed manner. It's little wonder that graphics have been called the "pinnacle of communication."

Data visualization is a very broad category that includes everything from simple charts used for data exploration and reporting, to complex interactive web applications that operate in real time. With Python, you

can easily cover this range. In fact, when it comes to creating graphics, Python suffers from an embarrassment of riches. With more than 40 different plotting libraries, there's something for everyone. But that's part of the problem.

Wading through Python's plotting APIs is exhausting. Users can be overwhelmed by all the choices, which cover a wide range of functionality, both unique and overlapping. As a result, they usually focus more on learning APIs than on their real job: exploring their data. In fact, this book was inspired by conversations with other scientists who were frustrated by this very problem.

Another issue with Python's plotting libraries is that the vast majority force you to write code to create even the simplest of visualizations. Compare this to software like Tableau or Excel, in which sensible, attractive graphs require just a few clicks of a mouse with little cognitive burden on the user.

Fortunately, many users share similar needs, and with a little forethought you can avoid going down suboptimal paths. In general, this involves selecting a high-level tool that covers the most common tasks succinctly and conveniently, typically by providing a simpler API on top of an existing tool.

In the sections that follow, we'll take a broad look at some of Python's most popular and useful plotting and dashboarding libraries. Then, we'll review some logical questions that should help guide you to the best plotting library, or libraries, for your needs.

NOTE *The plotting examples in this chapter are intended to demonstrate the complexity of the code and the types of plots produced. You're not expected to run the code snippets, as many of the libraries discussed do not come preinstalled with Anaconda. But if you do want to test them for yourself, you can find installation instructions in the product web pages. I recommend that you install them all in a dedicated conda environment (see Chapter 2), rather than dump them in your* `base` *environment.*

InfoVis and SciVis Libraries

We can divide visualizations into three main categories: *InfoVis*, *SciVis*, and *GeoVis* (Figure 16-1). InfoVis, short for *Information Visualization*, refers to 2D or simple 3D static or interactive representations of data. Common examples are statistical plots such as pie charts and histograms. SciVis, short for *Scientific Visualization*, refers to graphical representations of physically situated data. These visualizations are designed to provide insight into the data, especially when it's studied by novel and unconventional means. Examples are magnetic resonance imaging (MRI) and simulations of turbulent fluid flow. GeoVis, short for *Geovisualization*, refers to the analysis of geospatial (geographically located) data through static and interactive visualization. Examples include satellite imagery and map creation.

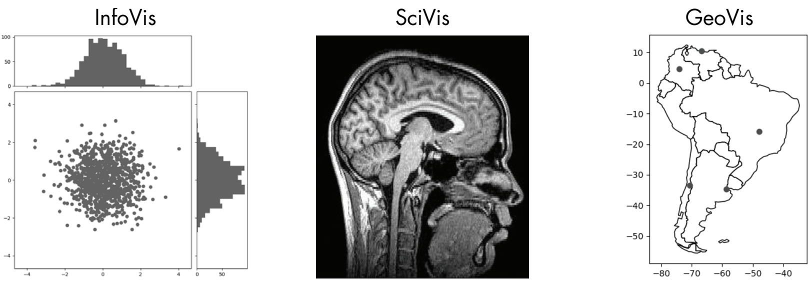


Figure 16-1: Three visualization categories with examples

Tables 16-1 lists some of Python's more important InfoVis and SciVis plotting libraries. We'll take a closer look at some of these in the sections that follow before turning to the dashboard libraries. Then, in Chapter 17, we'll repeat this exercise for the GeoVis libraries.

Table 16-1: Python's Major InfoVis and SciVis Libraries

Type	Library	Description	Website
	Matplotlib	Publication-quality 2D and simple 3D plots	*https://matplotlib.org/*
	seaborn	Matplotlib wrapper for easier, prettier plots	*https://seaborn.pydata.org/*
	pandas	Matplotlib wrapper for easy DataFrame plotting	*http://pandas.pydata.org/*
	Altair	Easy and simple 2D plots for small datasets	*https://altair-viz.github.io/*
	ggplot	Simple "grammar of graphics" plots with pandas	*https://yhat.github.io/ggpy/*
InfoVis	Bokeh	Web interactivity tool with large or streaming datasets	*https://bokeh.org/*
	Chartify	Bokeh wrapper for easier charting	*https://github.com/spotify/chartify/*
	Plotly	Dynamic, interactive graphics for web apps	*https://plotly.com/python/*
	HoloViews	Viz data structures usable by many libraries	*http://holoviews.org/*
	hvPlot	Easy interactive plotting library built on HoloViews/Bokeh	*https://hvplot.holoviz.org/*
	Datashader	Tools for rasterizing giant datasets for easy visualization	*https://datashader.org/*
	VTK	Visualization toolkit for 3D computer graphics	*https://vtk.org/*
SciVis	Mayavi	3D scientific visualization tool with interactivity	*https://docs.enthought.com/mayavi/*
	ParaView	3D scientific visualization tool with interactivity	*https://www.paraview.org/*

NOTE *If you're curious about how we got into this mess, take a few minutes to look at James Bednar's blog post "Python Data Visualization 2018: Why So Many Libraries?" (*https://www.anaconda.com/blog/python-data-visualization-2018-why-so-many-libraries/*). You should also check out his ebook,* Python Data Visualization*, and* PyViz *site (*https://pyviz.org/*), which are designed to help users decide on the best open source Python data visualization tools for their purposes, with links, overviews, comparisons, examples, and exhaustive tool lists.*

Matplotlib

The *Matplotlib* library is an open source, comprehensive library for creating manuscript-quality static, animated, and interactive visualizations in Python. These are mainly 2D plots, such as bar charts, pie charts, scatterplots, and so on, though some 3D plotting is possible (Figure 16-2). Matplotlib is almost 20 years old and was designed to provide early versions of Python with a familiar MATLAB-type interface. MATLAB is a proprietary scientific programming language that has been displaced in popularity by Python.

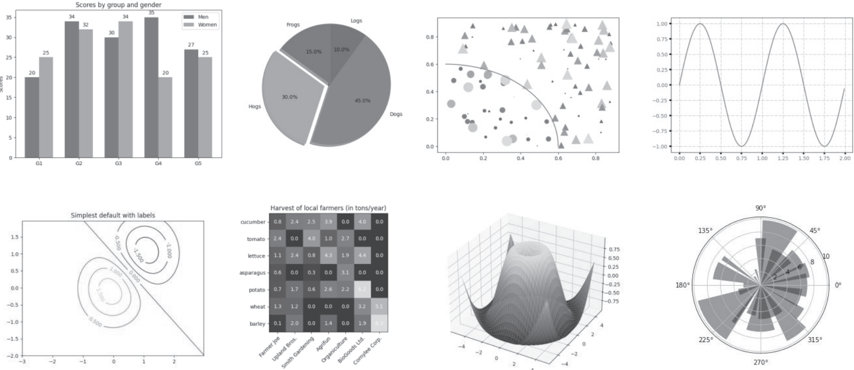


Figure 16-2: A small sampling of Matplotlib plot types (courtesy of https://matplotlib.org/*)*

Matplotlib's focus is on creating static images for use in publications and interactive figures for data exploration and analysis. These interactive figures use GUI toolkits like Qt, rather than web applications. The library comes preinstalled with Anaconda.

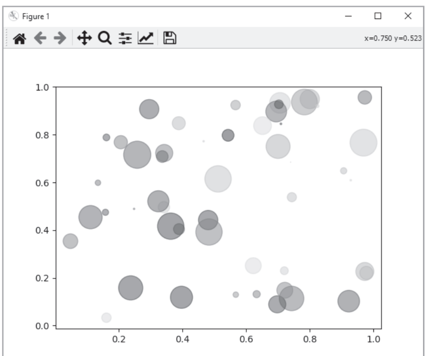
Matplotlib is the King, Grandaddy, and Big Kahuna of Python visualization. It's a massive, exhaustive library, and many alternative products are built on top of it, just as others are built on NumPy (including Matplotlib). Likewise, the internal visualization tools of libraries like pandas leverage Matplotlib methods.

The Matplotlib motto is that it "makes easy things easy and hard things possible." It works on all operating systems and handles all the common image formats. It has broad functionality, allowing you to build just about any kind of chart you can imagine, and it's very compatible with other popular science libraries like pandas, NumPy, and scikit-learn, thanks to collaborations between the Matplotlib and IPython communities.

Matplotlib is a powerful but low-level plotting engine. This means that you have lots of flexibility and options for precisely controlling plots by assembling them component by component. But this freedom comes with complexity. When creating anything beyond a simple plot, your code can become ugly, dense, and tedious.

The unfriendliness of Matplotlib's API is offset somewhat by its popularity and maturity. A simple online search will yield example code for just about any plot that you want to make. Its greatest resource is undoubtably the Matplotlib *gallery* (*https://matplotlib.org/gallery/index.html/*), a "cookbook" of code recipes for making a huge variety of plots.

Other issues with Matplotlib are the appearance and "explorable nature" of its plots. Although Matplotlib plots come with interactive features like zooming, panning, saving, and posting the cursor's *location* (Figure 16-3), they are somewhat antiquated compared to what's directly available in more modern libraries.



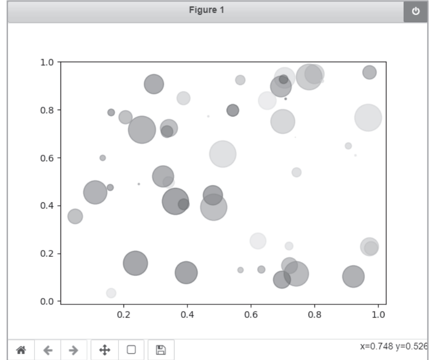


Figure 16-3: Matplotlib plot in an external Qt window (left) versus inline in a Jupyter notebook (right)

By default, Matplotlib's interactivity is designed to work in *external* windows rather than *inline* on the same screen as your code. You can force inline interactivity in Jupyter Notebook and JupyterLab, but the results can be buggy. For example, the Save button might simply open a blank web page rather than downloading the plot. Other libraries also provide more intelligent cursor hovering capabilities that can display custom information about posted data.

As a testament to Matplotlib's dominance and usefulness, a number of external packages extend or build on Matplotlib functionality (see *https://matplotlib.org/3.2.1/thirdpartypackages/*). Two of these, *mpldatacursor* and *mplcursors*, let you add *some* interactive data cursor functionality to plots using only a few lines of code.

Likewise, there are add-on visualization toolkits that rely on Matplotlib under the hood. One of the most important is *seaborn*, which is designed to simplify plotting and to generate more attractive plots than those produced by Matplotlib's defaults. Both seaborn and pandas are wrappers over Matplotlib, which lets you access some of Matplotlib's methods with less code.

seaborn

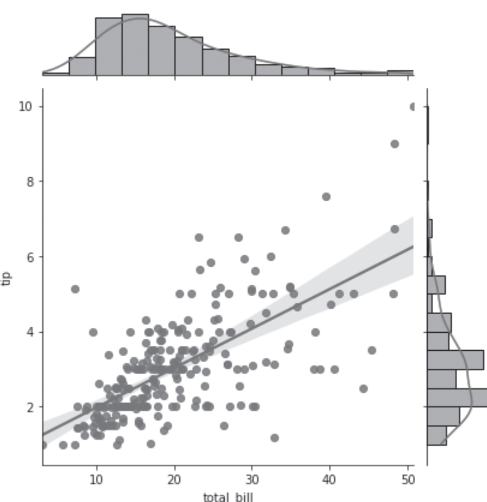
The *seaborn* library is a free, open source visualization library built on Matplotlib. It provides a higher-level (that is, easier-to-use) interface for drawing attractive and informative *statistical* graphics such as bar charts, scatterplots, histograms, and so on. It also comes with built-in functions for density estimators, confidence bounds, and regression functions. Not surprisingly, it's well integrated with data structures in pandas and NumPy. Seaborn comes preinstalled with Anaconda.

A goal of seaborn is to make visualization a central part of exploring and understanding data through the use of dataset-oriented plotting functions. It makes default plots more attractive and supports the building of complex visualizations. It helps reveal data patterns through the use of high-level multiplot grids and different color pallets (visit *https://seaborn.pydata.org/examples/index.html* for some examples).

Seaborn is designed to work well with the popular DataFrame objects in pandas, and you can easily assign column names to the plot axes. It's also considered preferrable to Matplotlib for making multidimensional plots.

In the example that follows, the last line of code generated an attractive scatterplot including a linear regression line with 95 percent confidence interval, marginal histograms, and distributions:

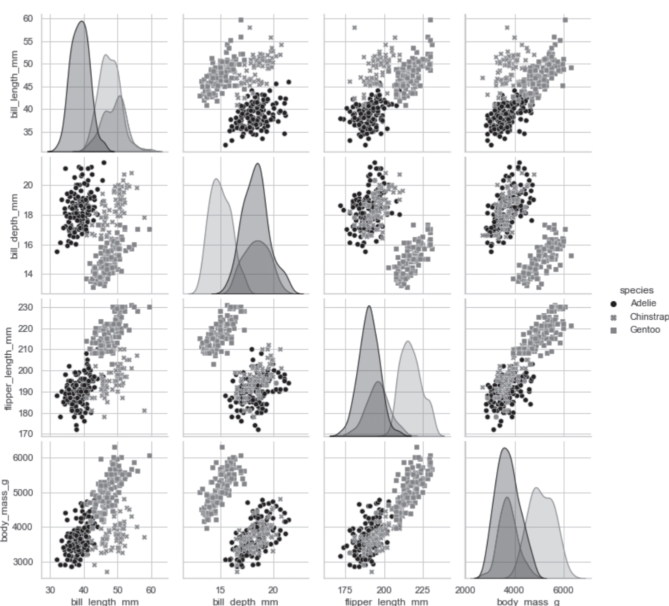
```
import seaborn as sns
tips = sns.load_dataset('tips')
sns.jointplot(data=tips, x='total_bill', y='tip', kind='reg');
```



One of the best features of seaborn is the *pairplot*. This built-in plot type lets you explore the pairwise relationships in an entire dataset in one figure, with the option of viewing histograms, layered kernel density estimates, scatterplots, and more. Following is an example of a pairplot created using

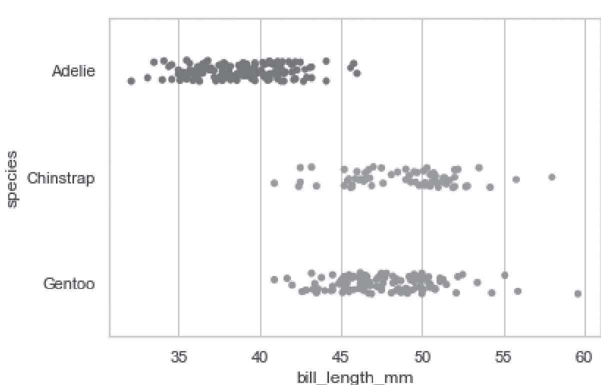
the Palmer Archipelago dataset for identifying penguin species. The data is loaded as a pandas DataFrame (see the pandas section in Chapter 15 for an overview of the pandas library).

```
import seaborn as sns
penguins = sns.load_dataset('penguins')
sns.pairplot(data=penguins, hue='species', markers=['o', 'X', 's']);
```



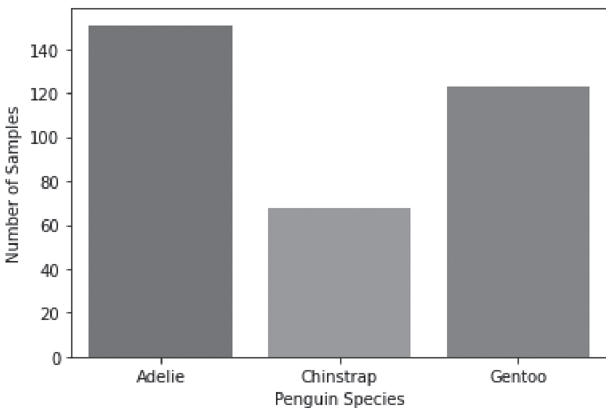
Another built-in plot type, *stripplot*, is a scatterplot in which one variable is categorical. It's perfect for comparing the lengths of bills among penguin species:

```
sns.set_theme(style='whitegrid')
strip = sns.stripplot(x='bill_length_mm', y='species', data=penguins);
```



Unlike Matplotlib, seaborn lets you manipulate data *during* the plotting operation. For example, you can calculate the number of body mass samples in the penguins dataset by calling the built-in length function (`len`) from within the `barplot()` method:

```
bar = sns.barplot(data=penguins, x='species', y='body_mass_g', estimator=len)
bar.set(xlabel='Penguin Species', ylabel='Number of Samples');
```



Let's take a look at how easy it is to customize a plot using seaborn. Table 16-2 lists the top 10 countries most affected by COVID-19 (based on number of cases) in roughly the first year of the virus's spread. The Fatality Rate column lists the number of deaths per 100 confirmed cases. The Deaths per 100,000 column calculates deaths based on a country's general population.

Table 16-2: COVID-19 Statistics

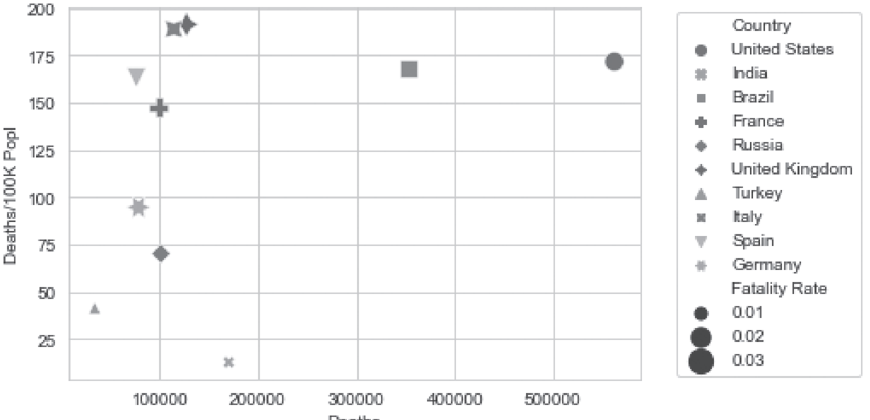
Country	Region	Cases	Deaths	Deaths/100K popl	Fatality rate
United States	North America	31,197,873	562,066	171.80	0.018
India	Asia	13,527,717	170,179	12.58	0.013
Brazil	Latin America	13,482,023	353,137	168.59	0.026
France	Europe	5,119,585	98,909	147.65	0.019
Russia	Asia	4,589,209	101,282	70.10	0.022
UK	Europe	4,384,610	127,331	191.51	0.029
Turkey	Middle East	3,849,011	33,939	41.23	0.009
Italy	Europe	3,769,814	114,254	189.06	0.030
Spain	Europe	3,347,512	76,328	163.36	0.023
Germany	Europe	3,012,158	78,500	94.66	0.026

Source: *https://coronavirus.jhu.edu/data/mortality*

Let's save Table 16-2 as a comma-separated value (*.csv*) file and use it with seaborn to look at the relationship among deaths, the death rate per 100,000 people, and the fatality rate:

```
import pandas as pd
import seaborn as sns

sns.set_style('whitegrid')
df = pd.read_csv('johns_hopkins_covid_stats_apr_2021.csv')
scatter = sns.scatterplot(data=df,
                          x='Deaths',
                          y='Deaths/100K Popl',
                          hue='Country',
                          style='Country',
                          size='Fatality Rate',
                          sizes=(50, 200))
scatter.legend(loc='center right', bbox_to_anchor=(1.4, 0.5), ncol=1);
```



After importing pandas and seaborn, you set the style of the plot to give it a white background with gridlines. The data, in *.csv* format, is then loaded as a pandas DataFrame named `df`. Creating a scatterplot (`scatter`) takes a single command. The marker color (`hue`) and shape (`style`) are based on the country and their size reflects the fatality rate, with a size range of 50 to 200. You finish by creating a legend and calling the plot. Note how, by using the DataFrame column names from Table 16-2, the code is easy to read and understand.

Despite being an abstraction layer on top of Matplotlib, seaborn provides access to underlying Matplotlib objects, so you can still achieve precise control over your plots. Of course, you'll need to know Matplotlib to some degree to tweak the seaborn defaults in this manner.

Seaborn plots are considered more attractive, and thus better for publications and presentations, than those produced by Matplotlib. It's a good choice if all you want are static images made with simpler code and better defaults.

NOTE *Even if you choose to use Matplotlib instead of the seaborn wrapper, you can still import seaborn and use its themes to improve the visual appearance of your plots. For examples, see* https://www.python-graph-gallery.com/106-seaborn-style-on-matplotlib-plot *and* https://seaborn.pydata.org/generated/seaborn.set_theme.html?highlight=themes.

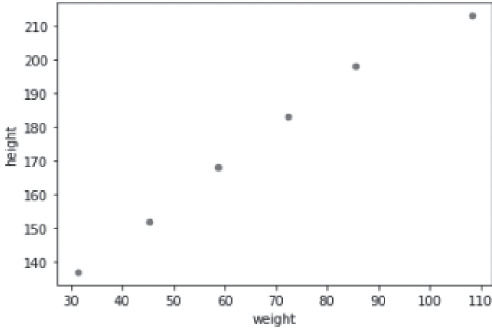
The pandas Plotting API

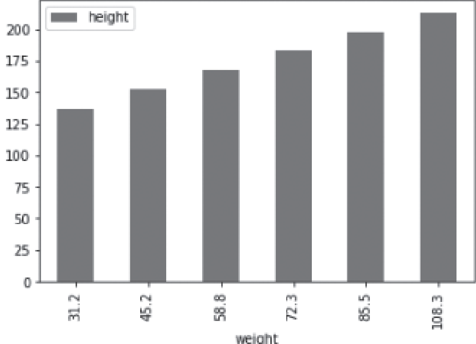
The pandas library discussed in the previous chapter has its own plotting API, `Pandas.plot()` (*https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html*). This API has emerged as a de facto standard for creating 2D charts because it can use Matplotlib and many other libraries as its plotting backend. This makes it possible to learn one set of plotting commands using pandas and then apply them with a wide range of libraries for static or interactive plots.

Plotting in pandas is arguably the easiest way to create visualizations using Python. It's especially good at quick "throwaway" plots for data exploration. Let's take a look:

```
import pandas as pd

female_ht_vs_wt = {'height': [137, 152, 168, 183, 198, 213],
                   'weight': [31.2, 45.2, 58.8, 72.3, 85.5, 108.3]}
df = pd.DataFrame(female_ht_vs_wt)
df.plot(kind='scatter', x='weight', y='height')
df.plot.bar('weight');
```



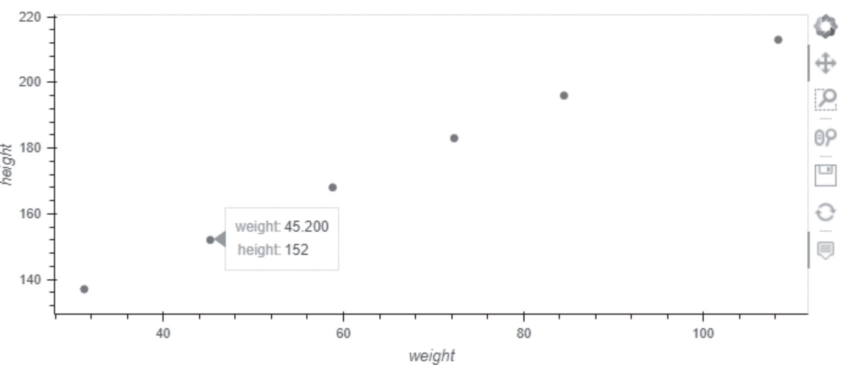


After importing pandas and making a Python dictionary of some measurements of female height verses weight, we turn the dictionary into a pandas DataFrame. The last two lines of code can then immediately build two plots! What could be easier?

The plots are very plain and lack any kind of interactivity, but never fear, pandas plays well with the other plotting libraries. With little effort, you can switch to an alternative plotting tool for additional functionality. By changing the plotting backend for pandas to HoloViews, a library we'll discuss shortly, you can produce an interactive plot that lets you zoom, pan, save, and hover the cursor over points to see their values. Here's an example of the code and its results:

```
import pandas as pd

pd.options.plotting.backend = 'holoviews'
female_ht_vs_wt = {'height': [137, 152, 168, 183, 196, 213],
                   'weight': [31.2, 45.2, 58.8, 72.3, 84.5, 108.3]}
df = pd.DataFrame(female_ht_vs_wt)
df.plot(kind='scatter', x='weight', y='height')
```



Note that, despite changing the plotting library, you didn't need to change a single line of the original plotting code. To see some other drop-in replacements for the `Pandas .plot()` API, see *https://pyviz.org/high-level/index.html#pandas-plot-api/*.

Altair

Altair is an open source statistical visualization library in Python that's closely aligned with pandas DataFrames. It's popular with people looking for a quick way to visualize small datasets.

Altair handles a lot of plotting details automatically, letting you focus on what you want to do rather than the button-pushing "how to do it" part. Much like the female height-verses-weight example in the previous section, you only need to link your data columns to encoding channels, such as the

x- and y-axes, to make a plot. But this ease of use comes with a few downsides. The plots are not as customizable as those made in Matplotlib, and there's no 3D plotting capability.

On the other hand, all Altair plots can be made interactive, meaning that you can zoom, pan, highlight plot regions, update linked charts with the selected data, enable *tooltips* that let you hover the cursor over points for detailed information, and so on. Altair visualizations require a JavaScript frontend to display charts and so should be used with Jupyter notebooks or an integrated development environment (IDE) with notebook support.

Unlike Matplotlib and other *imperative* plotting libraries that build plots step by step with no intermediate stages, Altair is *declarative* by nature, and generates a plot object, in JSON format, from which the plot can be reconstituted. JSON, short for JavaScript Object Notation, is a file and data interchange format that uses human-readable text to store and transmit data objects. Thus, Altair does not produce plots consisting of pixels, but plots consisting of data plus a visualization specification.

Because declarative plotting objects store your data and associated metadata, it's easy to manipulate the data during the plot render command or visualize it alongside or overlaid with other data. It can also result in very large visualization file sizes or entire datasets stored in your Jupyter notebook. Although there are some workarounds to help you manage memory and performance issues, the library's documentation recommends plotting no more than 5,000 rows of data (see *https://altair-viz.github.io/user_guide/faq.html#altair-faq-large-notebook/*).

Another drawback of using JSON is that it can be hacked if used with untrusted services or untrusted browsers. This can make the hosting web application vulnerable to a variety of attacks.

Bokeh

Bokeh is an open source visualization library that supports the creation of interactive, web-ready plots from very large or streaming datasets. Bokeh (pronounced "BO-kay") takes plots defined using Python and automatically renders them in a web browser using HTML and JavaScript, the dominant programming languages used for interactive web pages. It's one of the better-maintained and supported libraries and comes preinstalled with Anaconda.

Bokeh can output JSON objects, HTML documents, or interactive web applications. It has a three-level interface that provides increasing control over plots, from the simple and quick to the painstakingly detailed. However, unlike Matplotlib, Bokeh does not have high-level methods for some common diagrams such as pie charts, donut charts, or histograms. This requires extra work and the use of additional libraries such as NumPy. Support for 3D plotting is also limited. Thus, from a practical standpoint,

Bokeh's native API is mainly useful for publishing plots as part of a web app or HTML/JavaScript-based report, or for when you need to generate highly interactive plots or dashboards.

Bokeh works well in Jupyter notebooks and lets you use *themes*, for which you stipulate up front how you want your plots to look, such as font sizes, axis ticks, legends, and so on. Plots also come with a toolbar (Figure 16-4) for interactivity, including zooming, panning, and saving.

- Use the **pan tool** to move the graph within your plot.
- Use the **box zoom tool** to zoom into an area of your plot.
- Use the **wheel zoom tool** to zoom in and out with a mouse wheel.
- Use the **save tool** to export the current view of your plot as a PNG file.
- Use the **reset tool** to return your view to the plot's default settings.
- Use the **help symbol** to learn more about the tools available in Bokeh.

Figure 16-4: The Bokeh plot toolbar (courtesy of https://bokeh.org/*)*

Finally, if you keep your data in pandas, you can use a library called Pandas-Bokeh (*https://github.com/PatrikHlobil/Pandas-Bokeh/*), which consumes pandas data objects directly and renders them using Bokeh. This results in a higher-level, easier-to-use interface than Bokeh alone. Other high-level APIs built on Bokeh include HoloViews, hvPlot, and Chartify for plotting, and Panel for creating dashboards. We'll look at most of these later in the chapter.

Plotly

Plotly is an open source web-based toolkit for making interactive, publication-quality graphics. It's similar to Bokeh in that it builds interactive plots, generating the required JavaScript from Python. And like Bokeh and Matplotlib, Plotly is a core Python library on which multiple higher-level libraries are built.

Plotly graphs are stored in the JSON data format. This makes them portable and readable using scripts of other programming languages such as R, Julia, MATLAB, and more. Its web-based visualizations can be displayed in Jupyter notebooks, saved as standalone HTML files, or incorporated into web applications. Because Plotly uses JSON, it suffers similar memory and security issues as Altair (see "Altair" on page 429).

Unlike Matplotlib and seaborn, Plotly is focused on creating dynamic, interactive graphics in Python for embedding in web apps. You can create basic plots as well as more unique contour plots, dendrograms, and 3D charts (Figure 16-5).

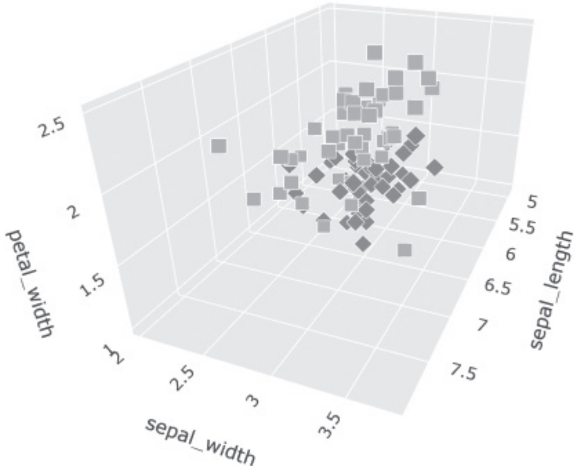


Figure 16-5: A 3D scatterplot made with Plotly Express

Figure 16-6 shows an example of a 3D mesh. You can even display LaTeX equations in legends and titles.

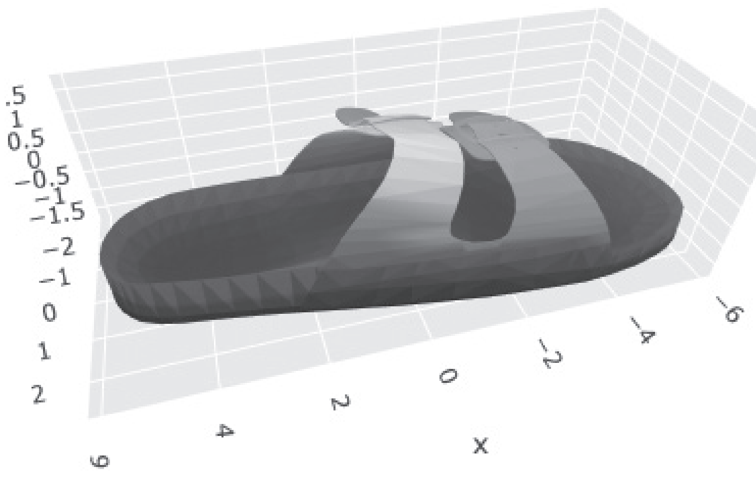


Figure 16-6: A sandal plotted as a 3D mesh in Plotly/Dash

Plotly also recognizes sliders, filters, and mouseover and cursor-click events. With only a few lines of code, you can create attractive interactive plots that save you time when exploring datasets and can be easily modified and exported. The toolkit also permits complex visualizations of multiple sources, in contrast to products like Tableau, which accept only one data table as input per chart.

Plotly is written in JavaScript and powers *Dash* (*https://dash.plotly.com/introduction*), an open source Python framework for building web analytic applications (called dashboards). Dash is written on top of Plotly.js and greatly simplifies the building of highly customized dashboards in Python. These apps are rendered in a web browser and can be deployed to servers and shared through URLs. Dash is cross-platform and mobile ready. We'll look at Dash a little more in "Dashboards" on page 445.

Plotly also comes with a high-level, more intuitive API called *Plotly Express* (*https://plotly.com/python/plotly-express/*) that provides shorthand syntax for creating entire figures at once. It has more than 30 functions for creating different types of graphics, each carefully designed to be as consistent and easy to learn as possible, allowing you to effortlessly switch from a scatterplot to a bar chart to a sunburst chart, and so on throughout a data exploration session. As such, Plotly Express is the recommended starting point for creating common figures with Plotly.

Plotly Express charts are easy to style so that they do really useful things. Suppose that you want to look at monthly rainfall totals over a two-decade period and see how the months of August and October compare to the rest. With Plotly Express, you can easily highlight the lines for these months so that they stand out. And with the interactive toolbar, you can toggle spike lines and the hover feature to query values (Figure 16-7).

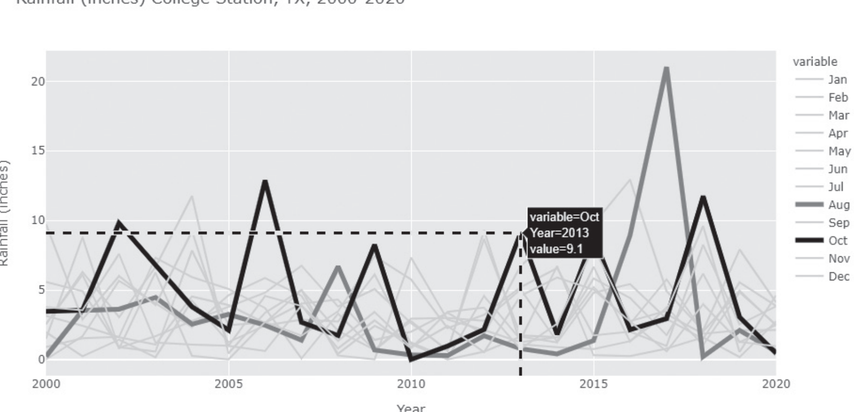


Figure 16-7: A Plotly Express line chart with highlighted lines, spike lines, and hover box

Another useful feature of Plotly Express is that legends are "alive." Click a category in a legend once and you temporarily remove it from the plot. Click it twice and all other lines will vanish, leaving that category isolated. This was done for the August (`Aug`) category in Figure 16-8. You can even animate the plot to see how things change over time. What a great way to untangle confusing "spaghetti" plots!

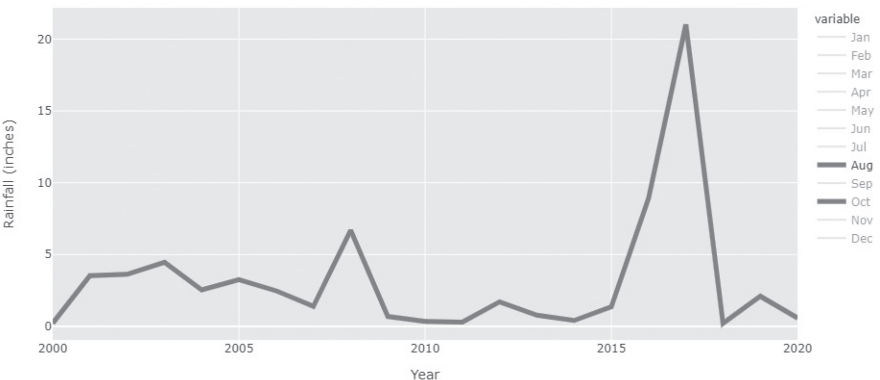
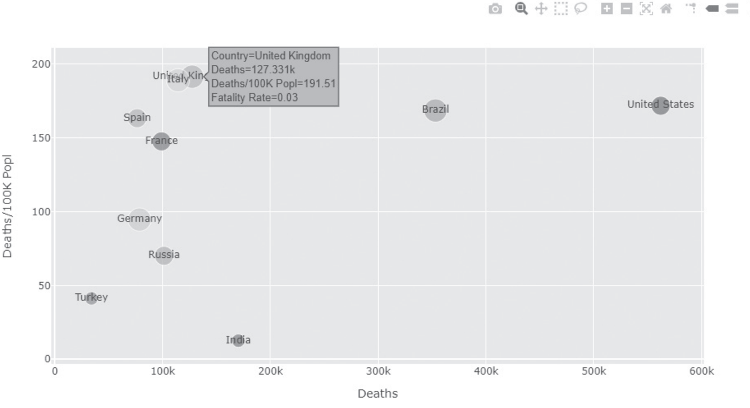


Figure 16-8: Double-clicking a legend category isolates that category by removing the other data.

Let's revisit the COVID-19 dataset that captures fatality statistics from the first year of the virus's spread. You'll want to compare the code and results that follow to the seaborn example on page 427.

```
import pandas as pd
import plotly.express as px

df = pd.read_csv('johns_hopkins_covid_stats_apr_2021.csv')
fig = px.scatter(data_frame=df,
                 x='Deaths',
                 y='Deaths/100K Popl',
                 color='Country',
                 size='Fatality Rate',
                 text='Country')
fig.update_layout(showlegend=False)
fig.show()
```

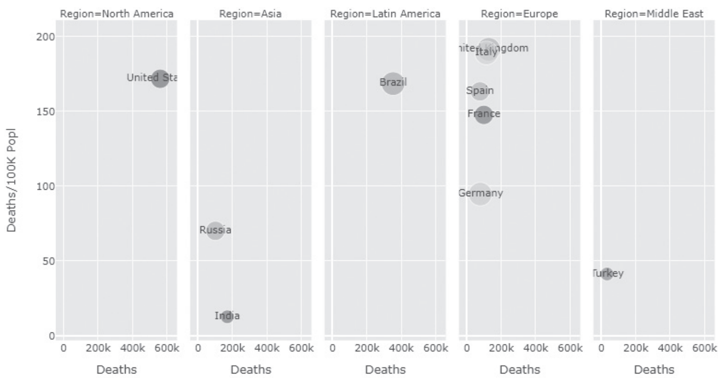


Like the previous seaborn code, it's very readable and easy to understand. Also note that Plotly Express has a specific parameter called `data_frame` that lets you know without a doubt that it's built for working with pandas.

A nice feature here is that you can easily post the country name over the markers, letting you use a consistent marker shape for easy size comparisons. You don't get the automatic "size" legend that you get with seaborn, but Plotly Express makes up for this by automatically permitting mouseover events, as shown in the plot for the United Kingdom.

Another useful Plotly Express feature is the *facet plot*, which lets you view the previous scatterplot by geographical region:

```
--snip--
fig = px.scatter(data_frame=df,
                 x='Deaths',
                 y='Deaths/100K Popl',
                 color='Country',
                 size='Fatality Rate',
                 text='Country',
               ❶ facet_col='Region')
fig.update_layout(showlegend=False)
fig.show()
```



We did this by adding a single argument ❶ to the `px.scatter()` method.

Plotly Express is designed mainly for exploratory data analysis. Your data must be in very specific formats (it's targeted at pandas DataFrames), your overall ability to customize plots is limited, and you might have trouble putting the visualizations into a presentation. To be able to do everything you'll probably want to do, you'll need to occasionally drop down into the full Plotly API or use Plotly Express in conjunction with other libraries like Matplotlib or seaborn.

There also exists an independent third-party wrapper library around Plotly called *cufflinks* (*https://github.com/santosjorge/cufflinks/*) that provides bindings between Plotly and pandas. This helps you create plots from pandas DataFrames using the `Pandas.plot()` interface but with Plotly output.

Both Plotly and Plotly Express facilitate building charts for the web directly from pandas DataFrames. And plots you create in Jupyter notebooks can essentially be copied and pasted into a Dash app for quick implementation of a dashboard. You can see an example of some scientific charts built with Plotly at *https://plotly.com/python/scientific-charts/*.

HoloViews

HoloViews is an open source library (note that I didn't say *plotting* library) designed to make visualization simple by abstracting away the process of plotting. HoloViews makes it easier to visualize data interactively by providing a set of declarative plotting objects that store your data with associated metadata. The goal is to support the entire life cycle of scientific research, from initial exploration to publication to reproduction of the work and new extensions.

HoloViews lets you combine various container types into data structures for visually exploring data. Some example container types are *Layout*, for displaying elements side by side as separate subplots; *Overlay*, for displaying elements on top of one another; and *DynamicMap*, for dynamic plots that automatically update and respond to user interactions. To appreciate the DynamicMap container, check out *https://holoviews.org/user_guide/Streaming_Data.html* and *https://holoviews.org/user_guide/Responding_to_Events.html* to view animated examples.

HoloViews generates final plots using a proper plotting library such as Matplotlib, Plotly, or Bokeh, as a backend. This lets you focus on your data rather than waste time writing plotting code. And as a plotting "middleman," HoloViews integrates well with libraries like seaborn and pandas and is particularly useful for visualizing large datasets—up to billions—using libraries like *Dask* and *Datashader* (such as *https://holoviz.org/tutorial/Plotting.html*).

One vision of Python's plotting future is to use a set of libraries to streamline the process of working with small and large datasets in a web browser (Figure 16-9). This would include doing exploratory analysis, making simple widget-based tools, or building full-featured dashboards.

Figure 16-9: The HoloViz-maintained libraries (courtesy of holoviz.org*)*

In this coordinated effort, HoloViews and GeoViews provide a single, concise, and high-level API for libraries like Matplotlib, Bokeh, Datashader, Cartopy, and Plotly. Panel provides a unified approach to dashboarding, and Datashader allows for the plotting of very large datasets. Param supports declaring user-relevant parameters for working with widgets inside or outside of a notebook context. This arrangement permits you to easily switch between backends without having to learn commands for each new plotting library.

Recognizing that a typical figure is an object composed of many visual representations combined together, HoloViews makes it trivial to compose elements in the two most common ways: concatenating multiple representations into a single figure or overlaying visual elements within the same set of axes. When making multiplot figures, HoloViews helps by automatically linking axes and selections across each figure. It's also useful for creating charts that update dynamically, especially those using sliders. With the Bokeh backend, you can combine various widgets with zooming and panning tools to aid data exploration.

Let's take a look at a Jupyter Notebook example, adapted from the HoloViews gallery (*https://holoviews.org/gallery/index.html*), that uses both HoloViews and Panel to generate a plot. For data, we'll again use the Palmer Archipelago dataset that quantifies the morphologic variations among three penguin species. Thanks to Panel, you'll be able to use drop-down menus to switch out and decorate the displayed data inside the single plot.

```
import seaborn as sns  # For access to penguins dataset.
import holoviews as hv
import panel as pn, panel.widgets as pnw
hv.extension('bokeh')

❶ hv.opts.defaults(hv.opts.Points(height=400, width=500,
                                  legend_position='right',
                                  show_grid=True))

penguins = sns.load_dataset('penguins')
columns = penguins.columns
discrete = [x for x in columns if penguins[x].dtype == object]
continuous = [x for x in columns if x not in discrete]
❷ x = pnw.Select(name='X-Axis', value='bill_length_mm', options=continuous)
y = pnw.Select(name='Y-Axis', value='bill_depth_mm', options=continuous)
size = pnw.Select(name='Size', value='None', options=['None'] + continuous)
color = pnw.Select(name='Color', value='None',
                   options=['None'] + ['species'] + ['island'] + ['sex'])
@pn.depends(x.param.value, y.param.value,
            color.param.value, size.param.value)

❸ def create_figure(x, y, color, size):
    opts = dict(cmap='Category10', line_color='black')
    if color != 'None':
        opts['color'] = color
    if size != 'None':
        opts['size'] = hv.dim(size).norm() * 20
    return hv.Points(penguins, [x, y], label="{} vs {}".
                     format(x.title(), y.title())).opts(**opts)

widgets = pn.WidgetBox(x, y, color, size, width=200)
pn.Row(widgets, create_figure).servable('Cross-selector')
```

After importing seaborn (for the data), HoloViews, and Panel, you tell HoloViews which plotting library to use. Bokeh is the default, but you can

easily change this to Matplotlib or Plotly by changing the line to `hv.extension('matplotlib')` or `hv.extension('plotly')`. Most of the time, changing the backend doesn't require any change to the rest of the code.

The next line ❶ is optional but demonstrates a nice feature of HoloViews: the ability to set your own defaults for how you want your plots to look. In this case, you set the size of the figure, position of the legend, and background grid to be used for all scatterplots.

Next, you load the penguins dataset, which conveniently ships with the seaborn library as a pandas DataFrame. To provide the user with menu choices, go through the columns in the `penguins` DataFrame and assign the contents to either a list called `discrete` or a list called `continuous`. The `discrete` list holds objects, such as species name, island name, or the penguin's sex. The `continuous` list is for numerical data, like the bill lengths and bill depths.

Starting at ❷, you must specify what choices the Panel widget will show for the x- and y-axes and the marker size and color, including the default options for what's initially shown. After this, you define a function to create the figure ❸ and return a HoloViews *Points* element. The final two lines create the figure with the menu widgets.

The output from this program is shown in Figure 16-10. Note the pull-down menus along the left side of the plot and the interactive toolbar along the right. Because we set the `size` and `color` default values to `'None'`, the points all look the same.

You can now use the menu widgets to color the points by species (Figure 16-11), which generates a legend at the lower-right corner of the plot. Setting the size option to body mass allows you to qualitatively incorporate a third measurement into the 2D scatterplot. Now you can see that the Gentoo species is clearly larger than the other two.

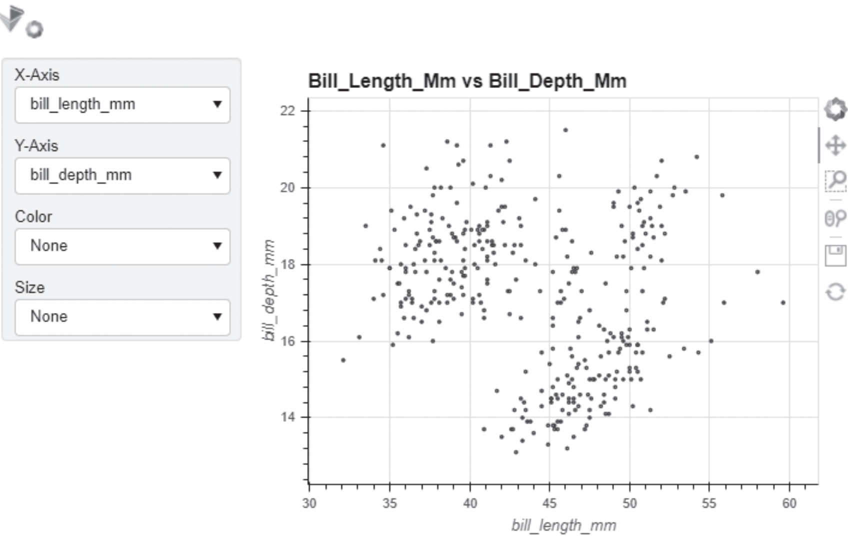


Figure 16-10: Bill depth versus bill length for three different penguin species

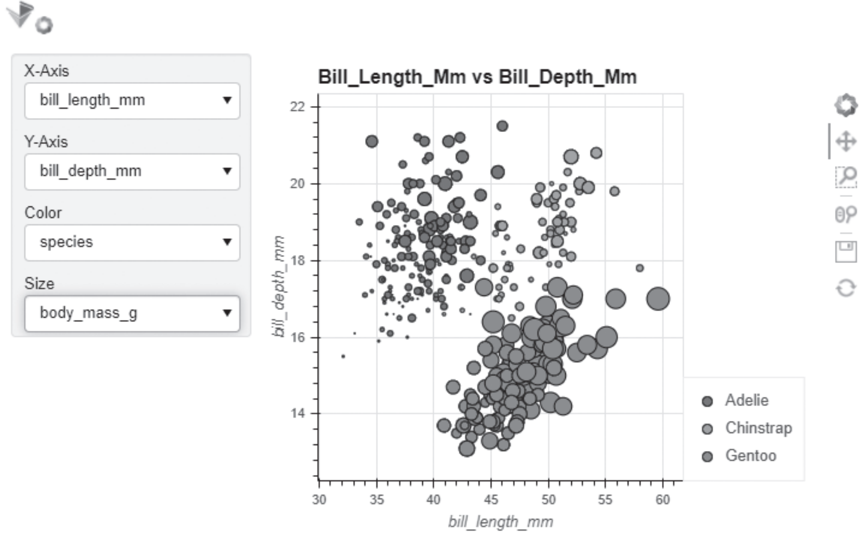


Figure 16-11: Bill depth versus bill length, colored by species and sized by body mass

In Figure 16-12, we've used the drop-down menus to change out both the data and size parameters. As you can see, this is a great way to interactively explore and familiarize yourself with a dataset without generating lots of plots.

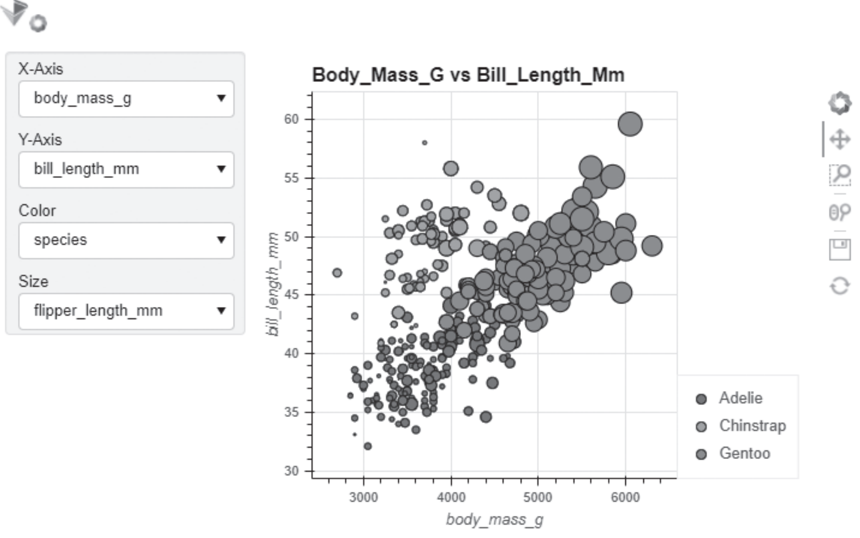


Figure 16-12: Bill length versus body mass, colored by species and sized by flipper length

A key point here is that the code references the DataFrame to make a HoloViews *Points element*. This object is basically the DataFrame plus knowledge of what goes on the x- and y-axes. This makes the DataFrame plottable. But unlike plot objects in other libraries, the `hv.Points` element holds

onto your raw data. This makes it usable later in a processing pipeline (for a dynamic demonstration, see the HoloViews Showcase at *http://holoviews.org/Tutorials/Showcase.html*).

Just as Plotly has Plotly Express, the HoloViz libraries have *hvPlot*, a simpler plotting alternative built on top of HoloViews. This fully interactive high-level API complements the primarily static plots available from libraries built on Matplotlib, such as pandas and GeoPandas, that require support from additional libraries for interactive web-based plotting. It's designed for the PyData ecosystem and its core data containers, which allow users to work with a wide array of data types (Figure 16-13).

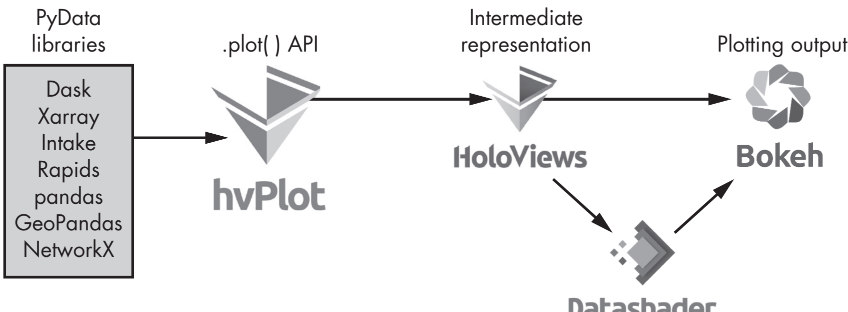
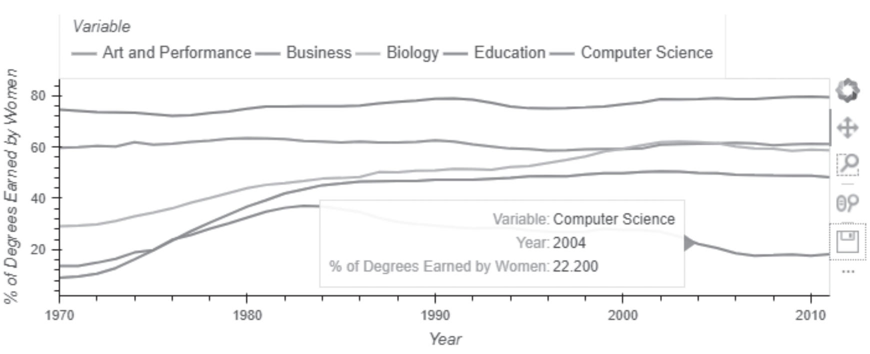


Figure 16-13: The hvPlot library provides a high-level plotting API for HoloViews.

The hvPlot library's interactive Bokeh-based API supports panning, zooming, hovering, and clickable/selectable legends. In the following example, hvPlot is used in conjunction with pandas to produce an interactive plot:

```
import hvplot.pandas
from bokeh.sampledata.degrees import data as degrees

degrees.hvplot.line(x='Year', y=['Art and Performance',
                                 'Business', 'Biology',
                                 'Education', 'Computer Science'],
                    value_label='% of Degrees Earned by Women',
                    legend='top')
```



This is just as simple as plotting in pandas, but note the toolbar along the right side of the chart with icons for panning, zooming, saving, and hovering. The latter lets you query the graph details using the cursor, as shown by the pop-up window for the computer science variable. These options aren't available when plotting from native pandas.

For more on these libraries, check out *HoloViz* (*https://holoviz.org/*), the coordinated effort to make browser-based data visualization in Python easier to use, easier to learn, and more powerful.

Datashader

Datashader is an open source library designed for visualizing very large datasets. Rather than passing the entire dataset from the Python server to a browser for rendering, Datashader rasterizes (pixelates) it to a much smaller heatmap or image, which is then transferred for rendering. Whereas popular libraries like Matplotlib can suffer from performance issues with only 100,000 points, Datashader can handle hundreds of millions, even billions, of them. For example, Figure 16-14 plots 300 million data points.

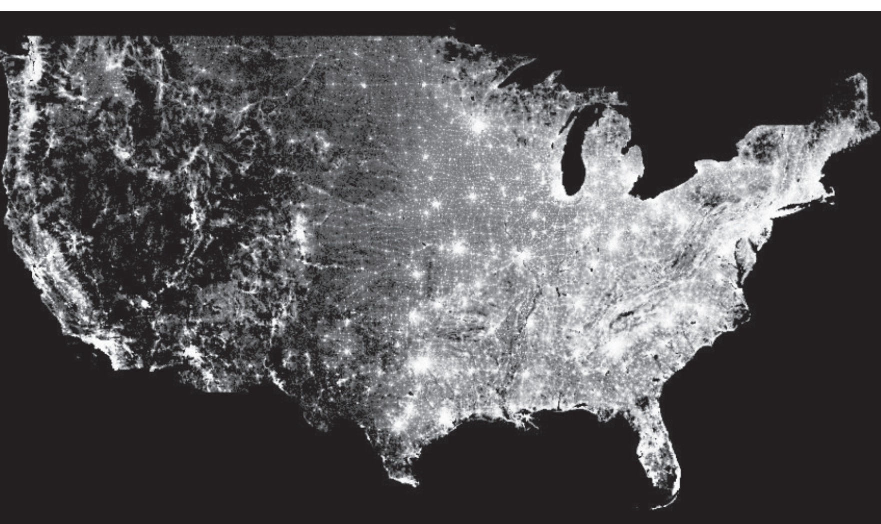


Figure 16-14: A Datashader-created plot of 300 million data points from the 2010 census (courtesy of Datashader)

Datashader makes it possible to work with very large datasets on standard hardware such as your laptop. Although the computationally intensive steps are written in Python, they're transparently compiled to machine code using a tool called *Numba* (*https://numba.pydata.org/*) and distributed across multiple processors using Dask.

The Datashader documentation highlights the tool's function in a preprocessing stage for plotting. What this means is that Datashader is often used with other plotting libraries to perform the heavy lifting associated with large datasets. Thus, although it's more focused on performance and efficiency than on directly generating basic statistical plots, it can work with

other tools to help you plot large datasets—say, in a scattergram—by handling the common over-posting of points problem, where the density of the distributed points is obscured (Figure 16-15).

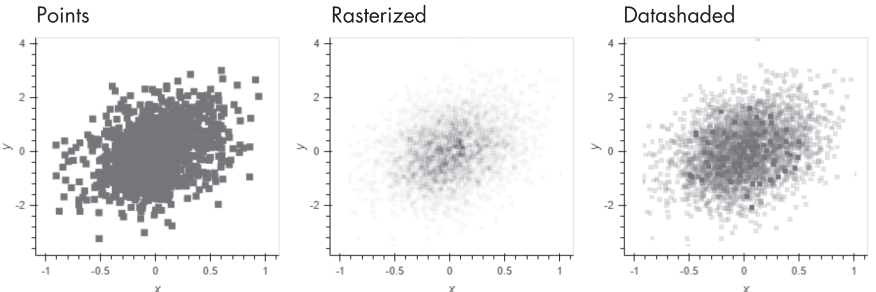


Figure 16-15: Datashader (right) handles over-posted points well (courtesy https://holoviews.org/*).*

In another example, imagine that you're using Bokeh to copy your data directly into the browser so that a user can interact with the data even without a live Python process running. If the dataset contains millions or billions of samples, you'll run up against the limitations of the web browser. But with Datashader, you can prerender this huge dataset into a fixed-size raster image that captures the data's distribution. Bokeh's interactive plot can then dynamically re-render these images when zooming and panning, making it easier to work with the huge dataset in the web browser (Figure 16-16).

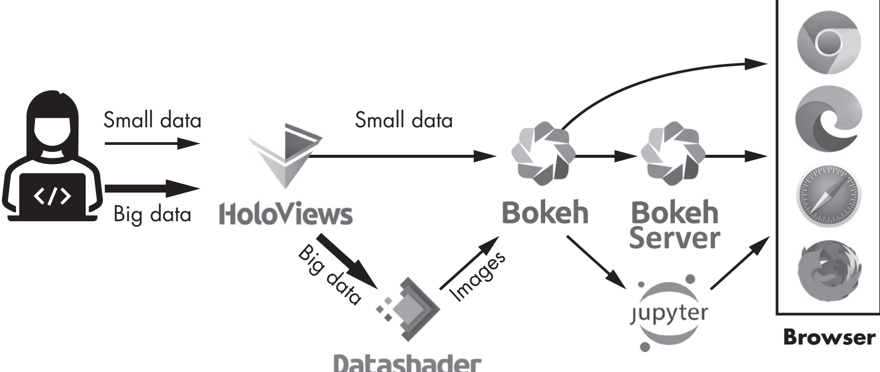


Figure 16-6: Generating interactive Datashader-based plots using HoloViews + Bokeh (courtesy of https://datashader.org/*)*

You can see a fantastic instance of Datashader in action in the "gerrymandering" example at *https://examples.pyviz.org/*. Working in concert with HoloViews and multiple plotting libraries, Datashader produces a map of

Houston's population, color-coded by ethnicity, that turns plotting into fine art, with a gorgeous watercolor-like rendering that has to be seen in color to be appreciated.

For a nice example of using Datashader with statistical plots, see *https://holoviews.org/user_guide/Large_Data.html*. Peter Wang, co-creator of Datashader, gives an easily digestible video overview of the library at *https://www.youtube.com/watch?v=fB3cUrwxMVY/*.

In all of these examples, be aware that you'll lose some interactivity with Datashader. You'll still be able to zoom and pan, but mouseover events and the like will no longer work without special support, because the browser doesn't hold all of your datapoints ready for inspection. In return, you'll be able to visualize millions of datapoints without watching your computer grind to a halt.

Mayavi and ParaView

A common scientific practice is to visualize point clouds, such as those you might find in a Light Detection and Ranging (LIDAR) scan. General-purpose workhorse libraries like Matplotlib are capable of performing this task to a certain degree, but performance deteriorates quickly when interactively visualizing point clouds and other 3D plots. Matplotlib, for example, will be slow and might even crash your computer if you try to interact with a large number of samples. Even if the 3D representations successfully render, they won't look very nice, and you'll probably have trouble understanding what you see.

Datashader can help with this, but for graphics-intensive 3D and 4D visualizations such as those used for physical processes, you need a dedicated library like Mayavi (pronounced MA-ya-vee) that can handle *physically situated* regular and irregularly gridded data. This discriminates Mayavi from Datashader somewhat, as the latter is focused more on visualizations of information in *arbitrary spaces*, not necessarily the three-dimensional physical world.

Mayavi2 is an open source, general-purpose, cross-platform tool for 3D scientific data visualization. It's been designed with scripting and extensibility in mind from the ground up. You can import Mayavi2 into a Python script and use it as a simple plotting library like Matplotlib. It also provides an application (Figure 16-17) that is usable by itself.

Mayavi2 is written in Python, uses powerful Visualization Toolkit (VTK) libraries, and provides a GUI via Tkinter. It's cross-platform and runs on any platform where both Python and VTK are available (almost any Unix, macOS, or Windows systems). To a limited extent, you can use Mayavi in Jupyter notebooks. To see some examples of Mayavi2 plots, visit the gallery at *https://docs.enthought.com/mayavi/mayavi/auto/examples.html*.

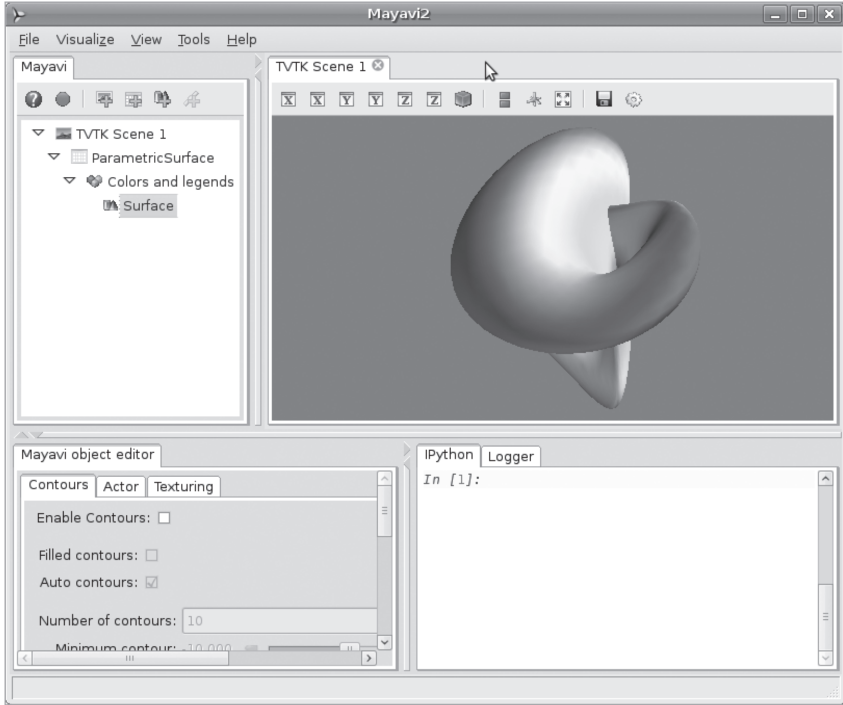


Figure 16-7: Mayavi2 application for 3D visualization. Note the Python console in the lower-right corner.

An alternative to Mayavi2 is *ParaView* (Figure 2-18). Although designed for 3D, it does 2D as well, is very interactive, and has a Python scripting interface.

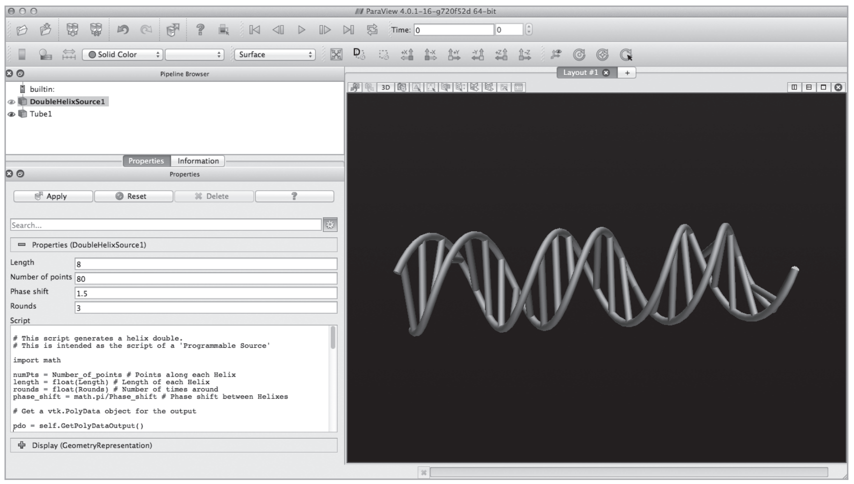


Figure 16-8: ParaView application for 3D visualization. Note the Python console in the lower-left corner.

ParaView was developed by Sandia National Laboratories, whereas Mayavi is a product of Enthought, whose Canopy distribution is a direct competitor of Anaconda.

Dashboards

A *dashboard* is a type of easy-to-read interactive GUI, often presented in real time. Dashboards are usually displayed on a single web page linked to a database, which allows the displayed information to be constantly updated. Example scientific dashboards include weather stations, earthquake monitoring, and spacecraft tracking (Figure 16-19).

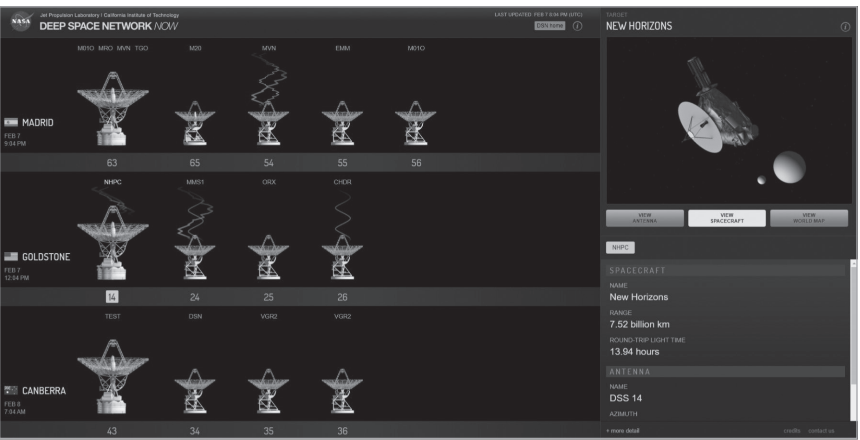


Figure 16-9: NASA spacecraft tracking dashboard (courtesy of https://www.nasa.gov*)*

Dashboards can really open up the usability and interactivity of your data, especially for nontechnical users. They also make the data accessible from anywhere, as long as you have an internet connection. This can be important when collaborating with external parties or providing results to scattered stakeholders.

Dashboards need to perform multiple tasks like analyzing and visualizing data, listening for and accepting user requests, and returning web pages via a web server. You can cobble together different libraries to handle these, or you can just use a dedicated dashboarding library.

Python supports higher-level web-based dashboarding with five main libraries: Dash, Streamlit, Voilà, Panel, and Bokeh (Table 16-4). These libraries let you create dashboards with pure Python, so you don't have to learn the underlying enabling languages like JavaScript and HTML. We looked at Bokeh earlier, so here we'll focus on the other four.

Table 16-4: Python's Most Important Dashboarding Libraries

Library	Description	Website
Plotly Dash	Advanced production-grade/enterprise dashboards	*https://plotly.com/dash/*
Streamlit	Fast and easy web apps from multiple plotting libraries	*https://streamlit.io/*
Voilà	Jupyter notebook rendering as stand-alone web apps	*https://voila.readthedocs.io/*
Panel	Interactive web apps with nearly any library	*https://panel.holoviz.org/*
Bokeh	Web interactivity with large or streaming datasets	*https://bokeh.org/*

Before we take a quick look at these four tools, note that it's possible to do some aspects of dashboarding in other libraries. The plotting stalwart Matplotlib supports several GUI toolkit interfaces, such as Qt, that can generate native applications you can use as an alternative to a web-based dashboard. Whereas several libraries make use of JavaScript to help build dashboards, *Bowtie* (*https://bowtie-py.readthedocs.io/*) lets you build them using pure Python. You can use *ipywidgets* with Jupyter Notebook to build a dashboard, but you need to use a separate deployable server, like Voilà, to share it.

For more insight, PyViz hosts a page on dashboarding that includes blog posts, links to comparison articles, and lists of alternative or supporting tools. You can find it at *https://pyviz.org/dashboarding/*.

NOTE *Bokeh, which we looked at previously, includes a widget and app library and a server for both plots and dashboards. It also supports live streaming of large datasets. However, if you intend to develop complex data visuals with Bokeh, you'll need some knowledge of JavaScript. Panel is built on Bokeh, just as seaborn is built on Matplotlib, and in the same way provides a higher-level toolkit to make dashboarding easier. It also supports multiple plotting libraries in addition to Bokeh.*

Dash

Dash is an open source Python framework developed by Plotly as a complete solution for deploying web analytic applications. Dash is built on Plotly.js, React.js, and *Flask* (a lower-level framework for building web apps from the ground up). Dash apps are rendered in a web browser deployed to servers and shared through a URL. This makes Dash platform agnostic and mobile ready. In 2020, Plotly released *JupyterDash* (*https://github.com/plotly/jupyter-dash/*), a new library designed for building Dash apps from Jupyter environments.

With Dash, it's possible to build a responsive, custom interface with pure Python in just a few hours. *Responsive*, by the way, means that the web page will render well on a variety of devices and screen sizes. Dash uses simple patterns to abstract away much of the dashboard-building process,

such as generating the required JavaScript, React components, HTML, and server API. In fact, you can basically copy and paste Plotly graphs straight from a Jupyter notebook into a Dash app.

As far as how your dashboard looks, Dash provides an attractive out-of-the-box default stylesheet but also allows you to easily add third-party styling. *Dash-bootstrap-components* (*https://dash-bootstrap-components.opensource.faculty.ai/*) is an open source library that makes it easier to build consistently styled apps with complex, responsive layouts. You can also use any of the themes from Bootswatch themes (*https://www.bootstrapcdn.com/bootswatch/*). These time-saving add-ons will let you build professional-looking dashboards with little effort.

Because of its relative maturity, expanding user community, and adoption by large enterprise organizations, Dash now has a large library of specialized modules, a host of repositories, and great documentation and tutorials to aid with the construction of customized dashboards. Whereas most scientists might aim to produce simple single-page dashboards, Dash can also build multipage, scalable, high-performance dashboards capable of incorporating organization style guides in the final layouts. This is a distinguishing feature of Dash versus friendlier tools like Streamlit and Voilà.

On the flip side, Dash is primarily designed for Plotly, though it's possible to use other third-party plotting libraries (see *https://github.com/plotly/dash-alternative-viz-demo/*). Dash also requires you to work with HTML and Cascading Style Sheets (CSS) syntax, which isn't something Python users generally want to do. This has led to the development of simpler tools, like Streamlit, which we'll look at next.

Streamlit

Streamlit is a relatively new open source library for quickly building attractive dashboard web applications. As an all-in-one tool, it addresses web serving as well as data analysis.

Streamlit's simple API lets you concentrate on your data analysis and visualization rather than on frontend and backend technology issues. Sharing and deploying is fast and easy, and the learning curve is arguably the shortest of any of Python's dashboarding tools. As a result, Streamlit's popularity has risen rapidly, and new features are constantly being added.

Whereas Dash focuses on production and enterprise settings, Streamlit is designed for rapid prototyping. It lets you do more with less code, and unlike Dash, which is designed to work primarily with Plotly, Streamlit lets you easily mix and match plots from multiple libraries, including Plotly, Altair, Bokeh, seaborn, and Matplotlib. This gives you the option to choose the best tool for the particular plotting job and allows contributing team members to use their preferred plotting library.

For existing Python scripts, Streamlit is arguably the best way to quickly and easily turn them into interactive dashboards. However, it provides no support for Jupyter Notebook, and you'll encounter some friction moving your code into Streamlit. On the other hand, it's very compatible with major libraries like scikit-learn, TensorFlow/Keras, NumPy, OpenCV, PyTorch,

pandas, and more. If you're happy with Streamlit's design defaults and don't need to do a lot of customization, it's a great choice for getting a dashboard up and running quickly.

Voilà

Voilà is an open source library that lets you quickly convert a Jupyter notebook into a stand-alone interactive dashboard sharable with others. As a thin layer built over Jupyter, it represents a very specific use case rather than a complete dashboarding solution.

Voilà allows nontechnical people associated with your project to use your Jupyter notebooks without having to know Python or Jupyter or have them installed on their computer. And if you already have a notebook with all the interactivity you need, it's the shortest path to turning your work into a dashboard.

Voilà is mostly about rendering. A common approach is to add interactivity (widgets) to a Jupyter notebook using a Python library like bqplot, Plotly, or ipywidgets, all of which are supported by Voilà. (We looked at ipywidgets in Chapter 5 on Jupyter Notebook.) You might then need to format the notebook to suppress and hide unused code and markdowns.

Voilà runs the code in the notebook, collects the outputs, and converts them to HTML. By default, the notebook code cells are hidden from view. The outputs are displayed vertically in the order in which they appear in the notebook (Figure 16-20), but you can use *widget layout templates* to change the position of the cell outputs, for example, by dragging them into a horizontal configuration. The page is then saved as a web application where the widgets on the page have access to the underlying Jupyter kernel.

At this point, the dashboard is only on your computer. For others to have access, you need to deploy your dashboard on the cloud using a public cloud computing platform such as Binder, Heroku, Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Cloud, or Microsoft Azure.

Binder, a free open source web application for managing digital repositories, is one of the most accessible ways to deploy Voilà applications. Use cases involve workshops, scientific workflows, and streamlined sharing among teams. Heroku (*https://www.heroku.com/*) is also a good choice for the less tech-savvy and those with limited budgets. It manages the supporting hardware and server infrastructure allowing you to focus on perfecting your app. On the downside, the app might run slowly due to low network performance. You can see more deployment options at *https://voila.readthedocs.io/en/stable/deploy.html*.

Voilà produces dashboards broadly similar to Streamlit and can be simpler to use, assuming that you already have a Jupyter notebook ready to go. Jupyter aficionados will also appreciate that Voilà shares Jupyter's widget library, whereas Streamlit requires you to learn its own set of custom widgets. You can see some example dashboards at *https://voila-gallery.org/*.

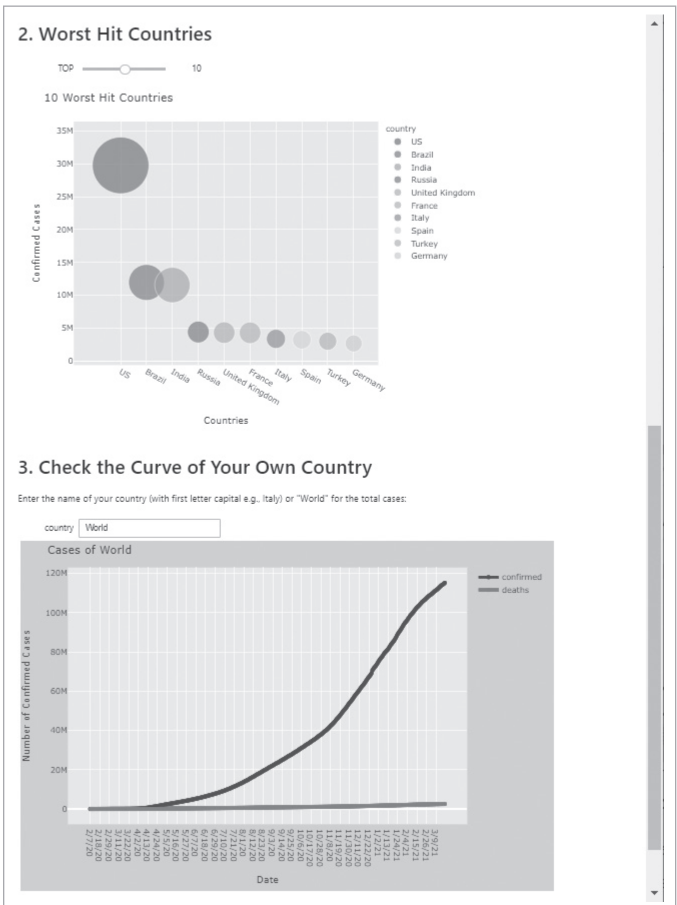


Figure 16-20: Dashboard elements retain Jupyter Notebook arrangement (courtesy of https://voila-gallery.org).

Panel

Panel is an open source Python library that lets you create custom interactive web apps and dashboards by connecting user-defined widgets to plots, images, tables, or text. Created and supported by Anaconda, Panel is part of the HoloViz family of unified plotting tools (see Figure 16-9) and uses the Bokeh server.

Panel helps support your entire workflow so that you never need to commit to only one way of using your data and your analyses, and you don't need to rewrite your code just to make it usable in a different way. You can move seamlessly from exploring data, creating reproducible steps, and telling a story in a notebook to creating a dashboard for a target audience, or even creating a notebook from a dashboard.

Panel automatically creates frontends based on Python syntax without requiring you to write in HTML or create style sheets with CSS. It

integrates better with Jupyter Notebook than Dash or Streamlit. It's arguably the next choice if you're already using Jupyter Notebook, and Voilà is not flexible enough for your needs.

Like Streamlit, Panel works with visualizations from multiple libraries, including Bokeh, Matplotlib, HoloViews, and more, making them instantly viewable either individually or when combined with interactive widgets that control them. Being integrated with the HoloViz family, including GeoViews, Panel is especially good for handling geospatial data.

Panel objects are reactive, immediately updating to reflect changes to their state. This makes it easy to compose viewable objects and link them into simple one-off apps to do a specific exploratory task. You then can reuse the same objects in more complex combinations to build more ambitious apps. You can also share information between multiple pages so that you can build full-featured multipage apps. To see some example dashboards and how Panel works with multiple plotting libraries, visit *https://panel.holoviz.org/gallery/index.html*.

Choosing a Plotting Library

Even the simplest plotting libraries in Python require a bit of time and effort to learn, so you can't realistically learn them all. But with so many plotting choices available, how do you choose among them?

The throwaway answer is that it depends on what you're trying to do. But there's more to it than that. You need to look beyond your immediate needs. What will you be doing next year? What are your teammates and clients using? How do you position yourself for the long term, to reduce the number of libraries you need to learn?

The following sections are designed to help you choose the best library, or combination of libraries, for you. They include the libraries we've discussed so far and address the following criteria:

Size of dataset The number of data points you need to plot

Types of plots The types of plots you plan to make, from statistical charts to complex 3D visualizations

Format The way you plan to present the data, such as static plots, Jupyter notebooks, interactive dashboards, and so on

Versatility A library's range of capabilities, such as ease of use, the ability to make sophisticated plots, and dashboarding support

Maturity The age of the library

For the first four criteria, we'll look at native, out-of-the-box functionality. Although it's always possible to extend the capabilities of a given library by using another library (for example, to enable interactivity), the assumption here is that the average user will want to avoid these types of complications.

And remember, we're only discussing a subset of the most popular plotting libraries. If you have highly specialized requirements, you'll need to perform an online search to find the most appropriate tool available.

Size of Dataset

The most important starting consideration for choosing a plotting library is the size of the datasets that you plan to use. In today's world of big data, you can't afford poor performance or memory issues during visualization. Although there are ways to decimate and otherwise manipulate large datasets so that they behave as smaller sets, you generally want to avoid this if possible.

Figure 16-21 presents a rough range of data sizes that you can practically plot with different libraries. These are more *relative* than *absolute*, as maximum limits can depend on the type of plot you're making, the hardware you're using, browser performance, whether you're working in a Jupyter notebook, and so on.

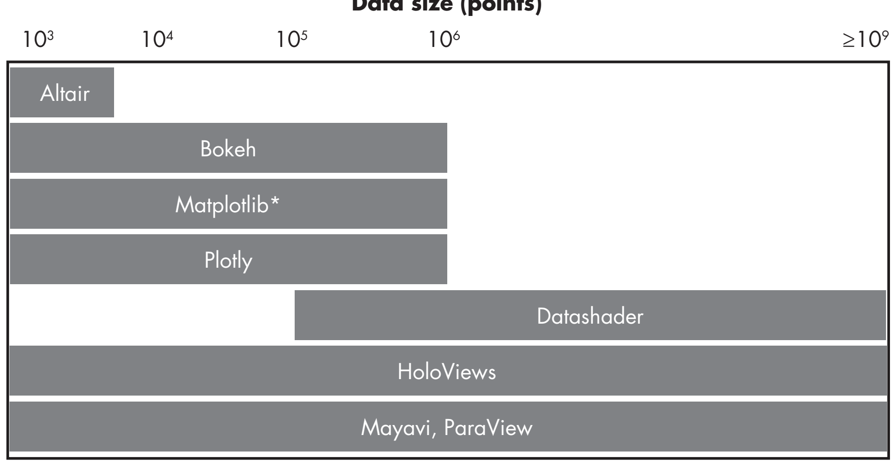


Figure 16-21: InfoVis and SciVis libraries versus size of dataset (in number of samples)

Most of the InfoVis libraries we've discussed can plot somewhere between a hundred thousand and a million data points. Bokeh supports both Canvas- and WebGL-based plotting, and the default Canvas plotting limit may be in the hundreds of thousands. But if the WebGL JavaScript API (*https://get.webgl.org/*) is used for Bokeh, assuming it's supported for the particular type of plot involved, the limit should be similar to that for Matplotlib and Plotly.

Larger datasets require Datashader, which renders plots as images. The SciVis libraries Mayavi and ParaView can handle billions of samples using compiled data libraries and native GUI apps. Because HoloViews can use Matplotlib, Bokeh, or Plotly as its plotting backend, as well as use Datashader, it can theoretically cover the whole range shown in Figure 16-21.

Types of Plots

Knowing the types of plots that you plan to make, along with their degree of interactivity, will help you in selecting the most user-friendly tool for your needs. Figure 16-22 shows the capabilities of plotting libraries, with simple statistical plots on the left and complex 3D visualizations on the right.

Figure 16-22: InfoVis and SciVis libraries versus type of plot

All of the InfoVis libraries can handle statistical plotting. Even the SciVis tools Mayavi and ParaView have this capability to some extent, though they're hardly the best choice. Likewise, although several InfoVis libraries can generate 3D scatterplots (Figure 16-5) and meshes (Figures 16-2 and 16-6), you still need Mayavi or ParaView for high-performance visualization of large and complex 3D plots (such as Figures 16-17 and 16-18). Of the three major plotting libraries, only Bokeh has no built-in 3D capability, though it can be extended by installing other libraries.

Format

Knowing how you will present your visualizations will help you choose a library while keeping things as simple as possible. With the exception of the specialty products like Mayavi, ParaView, and the dashboarding tools, you can use most libraries to generate static plots and images to print or use in a report. You'll want to verify that you can output the smooth SVG format if you need it, though most support this option. Figure 16-23 shows more sophisticated options, ranging from Jupyter notebooks to highly interactive web applications viewed in a browser.

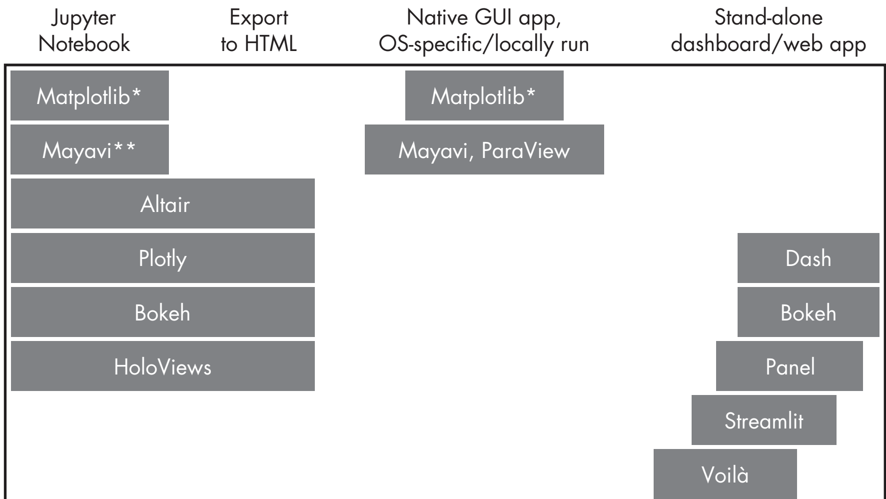


Figure 16-23: The InfoVis and SciVis libraries versus publishing format

The dashboarding libraries are displayed so that the simplest, least flexible ones are shifted to the left and the more powerful and customizable are shifted to the right. Voilà, for example, works only with Jupyter Notebook, whereas Dash can produce enterprise-level visualizations. Bokeh operates over *WebSockets*, a library for maintaining a persistent connection between a client and server, allowing for constantly connected sessions that you can easily use for multiple back-and-forth interactions.

Versatility

Sometimes organically and sometimes by design, plotting libraries grow into "families" of a sort (Figure 16-24). The Plotly family, for example, has Plotly Express for quick and simple plotting, and Dash for dashboarding. In similar fashion, HoloViews has hvPlot and Panel, and pandas and seaborn make plotting with Matplotlib as easy as possible. With a truly versatile family, you can quickly produce plots using simple syntax, drop down into the core library to add sophisticated elements, and seamlessly share the result as a dashboard on the web.

Even though it's possible to mix and match these to a point, having to learn the syntax for multiple libraries is not very appealing. Both Plotly and HoloViews give you full built-in soup-to-nuts functionality, but that doesn't mean you're limited to just two options. The Matplotlib family can "adopt" a dashboarding library, such as Streamlit, Panel, or Voilà, whereas Chartify, Pandas-Bokeh, and hvPlot can serve as an "easy" option for Bokeh.

Simplicity	Sophistication	Dashboarding
Altair		
pandas/seaborn	Matplotlib	
Plotly Express	Plotly	Dash
hvPlot	HoloViews	Panel
Chartify	Bokeh	Bokeh
	Mayavi, ParaView	

Figure 16-24: Versatility of the InfoVis and SciVis libraries

Maturity

Figure 16-25 captures the relative age of the plotting libraries. The longer a library has been around, the more likely it is to be reliable, well documented, and have an established user base that produces helpful tutorials, example galleries, and extensions. Over time, users encounter bugs, learn usage patterns, and share their experiences. As a result, you'll be able to find answers to most questions at help sites like Stack Overflow (*https://stackoverflow.com/*).

Paraview, Matplotlib, and pandas have been around for a long time, whereas libraries like Voilà and Panel are more recent. Keep in mind that maturity is a somewhat scalable criterion. Wildly popular libraries will mature quickly. A good example of this is the newer dashboarding libraries Dash and Streamlit, with rapidly growing user bases constantly adding new features and supplementing the documentation.

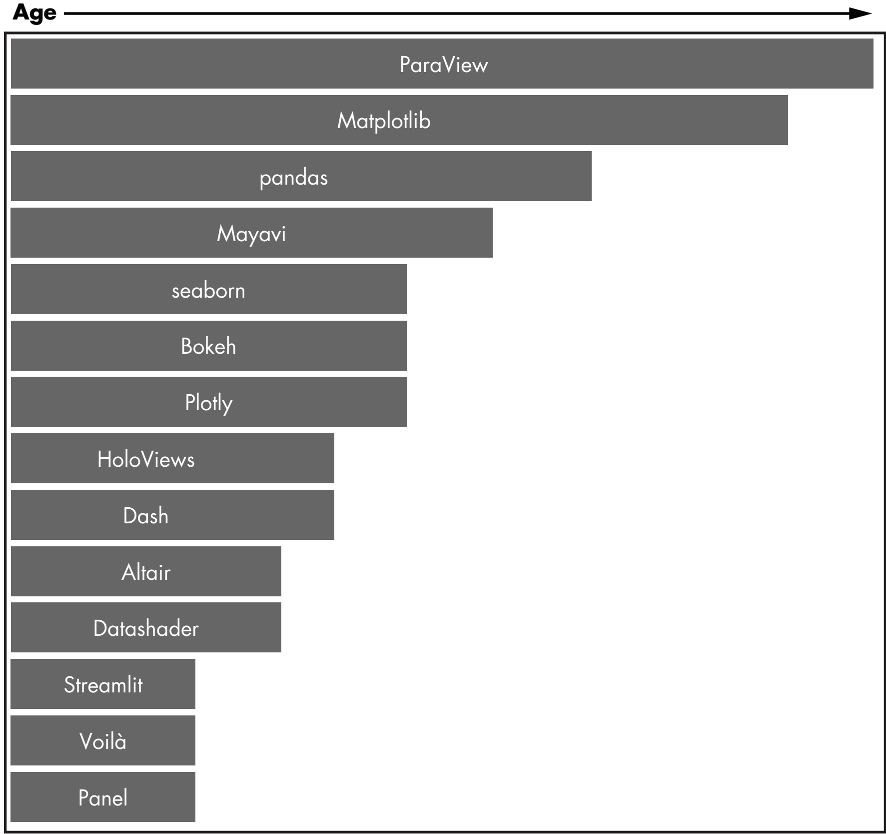


Figure 16-25: Relative age of the InfoVis and SciVis libraries

Making the Final Choice

Although it's true that the best plotting library might be dependent on your use case as well as your background and skill level, no one wants to jump from tool to tool with each new project. Still, there's a good chance you won't be able to get by with a single visualization library, especially if you need to do a range of things, including visualizing complicated 3D simulations.

If you expect to use Python *a lot*, you should look for a library, such as Matplotlib, Plotly, or the HoloViz family, that covers as much area as possible in Figures 16-21 through 16-25. These libraries may be more difficult to learn, but it will be worth it in the long run.

The case for learning Matplotlib is always strong due to its maturity, versatility, good integration with the ecosystem, and the fact that so many other libraries are built upon it. As a default plotting tool, it's a safe choice, but if you strongly favor a simpler library, all is not lost. As mentioned previously, Figures 16-21 through 16-24 assume that you're using the *native capability* of the posted libraries. They further assume that you want functionality, like zooming and panning, to work out of the box. But many

other libraries exist that, with little effort, can *extend* their native capabilities. Earlier, you saw how, with one extra line of code, HoloViews could add interactivity to the static plots generated by the pandas plotting API.

With Anaconda, it's easy to install plotting libraries and play with them in Jupyter Notebook. You should take the time to experiment a little using online tutorials. If you find that you prefer a fairly simple library or one not discussed here, search for libraries that can add any missing capability. You may be able to cobble together a Frankenstein product that perfectly fits your needs.

As a final comment: the HoloViz concept is intriguing. Its goal is to provide a unified, consistent, and forward-looking plotting solution for Python. It's worth serious consideration, especially if you have a long career ahead of you.

NOTE *After you choose a plotting library, you'll still need to pick a type of plot to use with your data. A great place to start is the From Data to Viz website at* https://www.data-to-viz.com/. *Here you'll find a decision tree that will help you determine the most appropriate chart based on the format of your dataset. You'll also find a Caveats page that will help you understand and avoid some of the most common data presentation mistakes.*

Summary

In this chapter, we reviewed the InfoVis libraries, used for 2D or simple 3D static or interactive representations of data, as well as the more sophisticated SciVis libraries, used for graphical representations of physically situated data. Because the InfoVis libraries address common displays such as bar charts and scattergrams, there are many libraries from which to choose.

The most popular InfoVis library is Matplotlib. Due to its maturity and flexibility, other plotting libraries, like seaborn, "wrap" Matplotlib to make it easier to use and to provide additional themes and styles. Newer plotting libraries such as Bokeh, Plotly, and Holoviews, provide much of the functionality of Matplotlib but also focus on web apps and the building of interactive dashboards. Other tools, like Datashader, address the need to efficiently plot large volumes of data.

The choice of a go-to plotting library is a personal one influenced by the tasks that you need to complete and the effort you're willing to apply. Because most users will want to focus on learning as few packages as possible, the best solution is to choose a plotting "family" that provides broad coverage of plot types, formats, dataset sizes, and so on. This will need to be weighed against the value of a mature (but possibly disjointed) solution that comes with lots of support versus newer, less well-documented libraries that try to provide a seamless, holistic approach that will stand the test of time.