02-bokeh-basic-plotting

October 29, 2023

we are covering the `bokeh.plotting` interface. This interface is a "mid-level" interface, and the main idea can be described by the statement:

Starting from simple default figures (with sensible default tools, grids and axes), add markers and other shapes whose visual attributes are tied to directly data.

We will see that it is possible to customize and change all of the defaults, but having them means that it is possible to get up and running very quickly.

1 Imports and Setup

When using the `bokeh.plotting` interface, there are a few common imports: * Use the `figure` function to create new plot objects to work with. * Call the functions `output_file` or `output_notebook` (possibly in combination) to tell Bokeh how to display or save output. * Execute `show` and `save` to display or save plots and layouts.

[ ]:
```
import numpy as np # we will use this later, so import it now

from bokeh.io import output_notebook, show
from bokeh.plotting import figure
```

In this case, we are in the Jupyter notebook, so we will call `output_notebook()` below. We only need to call this once, and all subsequent calls to `show()` will display inline in the notebook.

[ ]:
```
output_notebook()
```

If everything is working, you should see a Bokeh logo and a message like *"BokehJS 1.4.0 successfully loaded."* as the output.

This notebook uses Bokeh sample data. If you haven't downloaded it already, this can be downloaded by running the following:

[ ]:
```
import bokeh.sampledata
bokeh.sampledata.download()
```

2 Scatter Plots

Bokeh can draw many types of visual shapes (called *glyphs*), including lines, bars, patches, hex tiles and more. One of the most common visualization tasks is to draw a scatter plot of data using small *marker* glyphs to represent each point.

In this section you will see how to use Bokeh's various marker glyphs to create simple scatter plots.

The basic outline is: * create a blank figure: `p = figure(...)` * call a glyph method such as `p.circle` on the figure * `show` the figure

Execute the cell below to create a small scatter plot with circle glyphs:

```
[ ]: # create a new plot with default tools, using figure
     p = figure(width=400, height=400)

     # add a circle renderer with x and y coordinates, size, color, and alpha
     p.circle([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], size=25, line_color="navy",␣
      ↪fill_color="orange", fill_alpha=0.5, line_width=2)
     #p.circle([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], radius=.2, line_color="navy",␣
      ↪fill_color="orange", fill_alpha=0.9)

     show(p) # show the results
```

In the output above, you can see the effect of the different options for `line_color`, `fill_alpha`, etc. Try changing some of these values and re-executing the cell to update the plot.

All Bokeh scatter markers accept `size` (measured in screen space units) as a property. Circles in particular also have `radius` (measured in "data" space units).

```
[ ]: # EXERCISE: Try changing the example above to set a `radius` value instead of␣
      ↪`size`
```

To scatter square markers instead of circles, you can use the `square` method on figures.

```
[ ]: # create a new plot using figure
     p = figure(width=400, height=400)

     # add a square renderer with a size, color, alpha, and sizes
     p.diamond([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], size=[10, 15, 20, 25,␣
      ↪30],line_color="black", color="firebrick", alpha=0.6, line_width=2)

     show(p) # show the results
```

Note that in the example above, we are also specifying different sizes for each individual marker. **In general, all of a glyph's properties can be "vectorized" in this fashion.** Also note that we have passed `color` as a shorthand to set both the line and fill colors easily at the same time. This is a convenience specific to `bokeh.plotting`.

There are many marker types available in Bokeh, you can see details and example plots for all of them in the reference guide by clicking on entries in the list below:

- asterisk()
- circle()
- circle_cross()

- circle_x()
- cross()
- diamond()
- diamond_cross()
- hex()
- inverted_triangle()
- square()
- square_cross()
- square_x()
- triangle()
- x()

[ ]:
```
# EXERCISE: Make a scatter plot using the "autompg" dataset

from bokeh.sampledata.autompg import autompg as df # run df.head() to inspect
```

3 Line Plots

Another common visualization task is the drawing of line plots. This can be accomplished in Bokeh by calling the `p.line(...)` glyph method as shown below.

[ ]:
```
# create a new plot (with a title) using figure
p = figure(width=400, height=400, title="My Line Plot")

# add a line renderer
p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)

show(p) # show the results
```

In addition to `line_width`, there are other options such as `line_color` or `line_dash` that can be set. Try setting some of the other properties of line and re-running the cell above.

3.0.1 Datetime axes

It's often the case that timeseries data is represented by drawing lines. Let's look at an example using the "glucose" data set, which is available in a Pandas dataframe:

[ ]:
```
from bokeh.sampledata.glucose import data
data.head()
```

We'd like to plot a subset of this data, and have a nice datetime axis as well. We can ask Bokeh for a datetime axis by passing `x_axis_type="datetime"` to the call to `figure`. This is shown below, as well as configuration of a some other options such as plot dimensions, axis titles, and grid line properies.

[ ]:
```
# reduce data size to one week
week = data.loc['2010-10-01':'2010-10-08']
```

```
p = figure(x_axis_type="datetime", title="Glocose Range", height=350, width=800)
p.xgrid.grid_line_color="None"
p.ygrid.grid_line_alpha=0.5
p.xaxis.axis_label = 'Time'
p.yaxis.axis_label = 'Value'

p.line(week.index, week.glucose)

show(p)
```

[ ]:
```
# EXERCISE: Look at the AAPL data from bokeh.sampledata.stocks and create a␣
 ↪line plot using it
from bokeh.sampledata.stocks import AAPL

# AAPL.keys()
# dict_keys(['date', 'open', 'high', 'low', 'close', 'volume', 'adj_close'])

dates = np.array(AAPL['date'], dtype=np.datetime64) # convert date strings to␣
 ↪real datetimes
```

4 Hex Tiling

Bokeh supports drawing low level hex tilings using axial coordinates and the `hex_tile` method, as described in the Hex Tiles section of the User's Guide. However, one of the most common uses of hex tilings is to visualize binning. Bokeh encapsulates this common operation in the `hexbin` function, whose output can be passed directly to `hex_tile` as seen below.

[ ]:
```
from bokeh.palettes import Viridis256
from bokeh.util.hex import hexbin

n = 50000
x = np.random.standard_normal(n)
y = np.random.standard_normal(n)

bins = hexbin(x, y, 0.1)

# color map the bins by hand, will see how to use linear_cmap later
color = [Viridis256[int(i)] for i in bins.counts/max(bins.counts)*255]

# match_aspect ensures neither dimension is squished, regardless of the plot␣
 ↪size
p = figure(tools="wheel_zoom,reset", match_aspect=True,␣
 ↪background_fill_color='#440154')
p.grid.visible = False

p.hex_tile(bins.q, bins.r, size=0.4, line_color=None, fill_color=color)
```

```
show(p)
```

[ ]:
```
# Exercise: Experiment with the size parameter to hexbin, and using different␣
↪data as input
```

5 Images

Another common task is to display images, which might represent heat maps, or sensor data of some sort. Bokeh provides two glyph methods for displaying images:

- `image` which can be used, together with a palette, to show colormapped 2d data in a plot
- `image_rgba` which can be used to display raw RGBA pixel data in a plot.

The first example below shows how to call `image` with a 2d array and a palette

[ ]:
```
N = 500
x = np.linspace(0, 10, N)
y = np.linspace(0, 10, N)
xx, yy = np.meshgrid(x, y)

img = np.sin(xx)*np.cos(yy)

p = figure(x_range=(0, 10), y_range=(0, 10))

# must give a vector of image data for image parameter
p.image(image=[img], x=0, y=0, dw=10, dh=10, palette="Spectral11")

show(p)
```

A palette can be any list of colors, or one of the named built-in palettes, which can be seen in the bokeh.palettes reference guide. Try changing the palette, or the array data and re-running the cell above.

The next example shows how to use the `image_rgba` method to display raw RGBA data (created with help from NumPy).

[ ]:
```
from __future__ import division
import numpy as np

N = 20
img = np.empty((N,N), dtype=np.uint32)

# use an array view to set each RGBA channel individiually
view = img.view(dtype=np.uint8).reshape((N, N, 4))
for i in range(N):
    for j in range(N):
        view[i, j, 0] = int(i/N*255) # red
        view[i, j, 1] = 158          # green
        view[i, j, 2] = int(j/N*255) # blue
        view[i, j, 3] = 255          # alpha

# create a new plot (with a fixed range) using figure
p = figure(x_range=[0,10], y_range=[0,10])

# add an RGBA image renderer
p.image_rgba(image=[img], x=[0], y=[0], dw=[10], dh=[10])

show(p)
```

Try changing the RGBA data and re-running the cell above.

6 Other Kinds of Glyphs

Bokeh supports many other kinds of glyphs. You can click on the User Guide links below to see how to create plots with these glyphs using the `bokeh.plotting` interface.

- Ovals and Ellipses
- Segments and Rays
- Wedges and Arcs
- Specialized Curves

We will cover various kinds of Bar plots (e.g. with stacking and grouping) using Bars and Rectangles much more extensively in the Bar and Categorical Data Plots chapter of this tutorial.

[ ]:
```
# EXERCISE: Plot some of the other glyph types, following the examples in the␣
↪User Guide.
```

7 Plots with Multiple Glyphs

Finally, it should be noted that is possible to combine more than one glyph on a single figure. When multiple calls to glyph methods happen on a single figure, the glyphs are draw in the order called, as shown below.

[ ]:
```
# set up some data
x = [1, 2, 3, 4, 5]
y = [6, 7, 8, 7, 3]

# create a new plot with figure
p = figure(width=400, height=400)

# add both a line and circles on the same plot
p.line(x, y, line_width=2)
p.circle(x, y, fill_color="white", size=8)

show(p) # show the results
```

[ ]:
```
# EXERCISE: create your own plot combining multiple glyphs together
```