Interactive Visualization with Python

Bruno Gonçalves

www.data4sci.com/newsletter

viz4sci.substack.com

https://github.com/DataForScience/InteractiveViz

Question

- What's your job title?
    - Data Scientist
    - Data Engineer
    - Statistician
    - Researcher
    - Business Analyst
    - Software Engineer
    - Other

Question

- How experienced are you in Python?
    - Beginner (<1 year)
    - Intermediate (1-5 years)
    - Expert (5+ years)

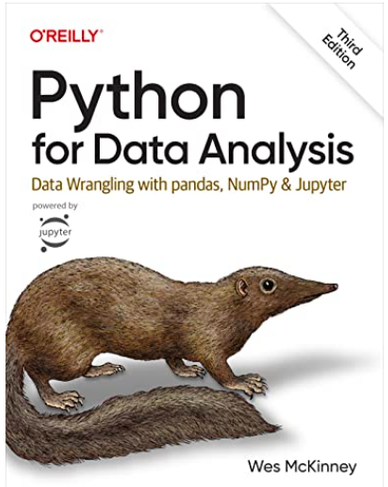
Question

- How did you hear about this webinar?
    - O'Reilly Platform
    - Newsletter
    - Website
    - Previous event
    - Other?

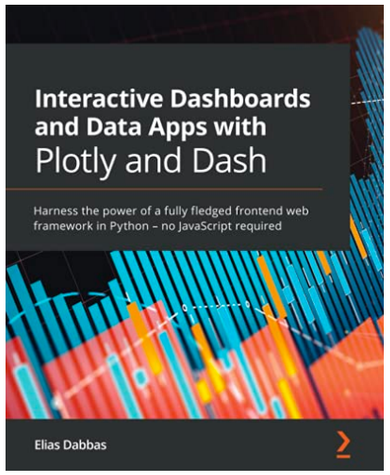
Table of Contents

References

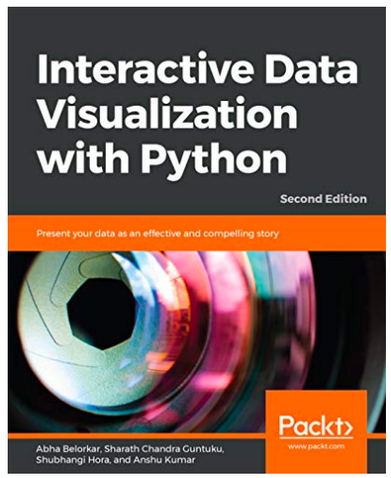
viz4sci.substack.com



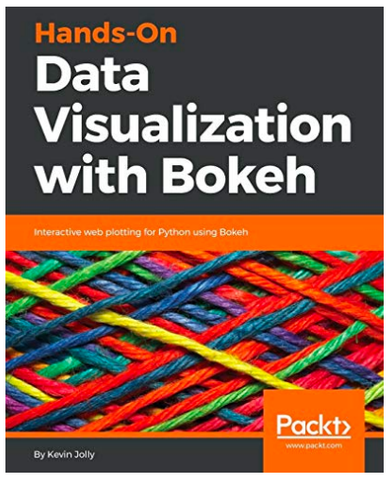
https://amzn.to/3lJ0VsL



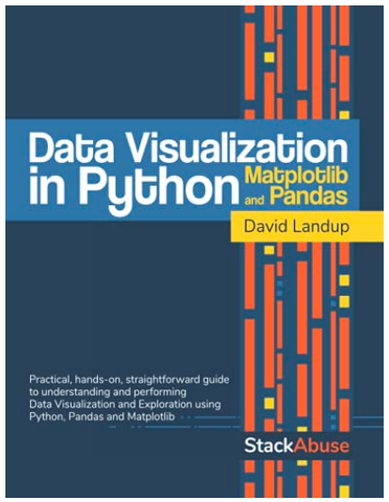
https://amzn.to/3zdiBj7



https://amzn.to/40DuWZG



https://amzn.to/3KcvTme



https://amzn.to/3nuCake

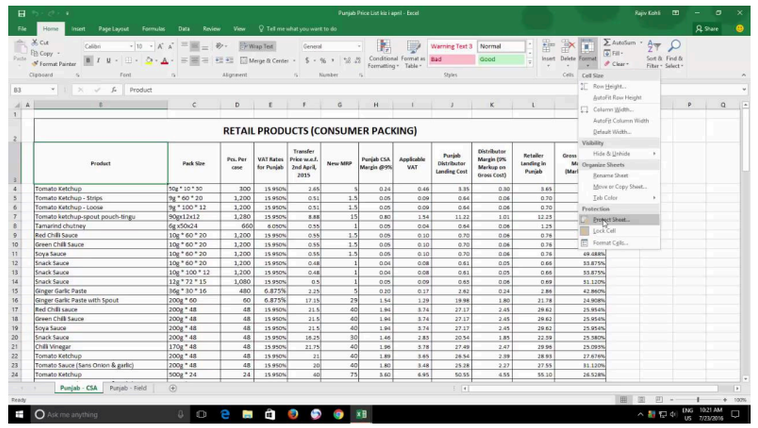
1. Visualization with Pandas

Series and Data Frames

- A Series is conceptually equivalent to a **numpy** array with some extra information (column and row names, etc)

- A DataFrame can be thought of as the union of several Series, with names associated.

- Similarly, you can think of a DataFrame as an Excel **sheet** and of a Series as an individual **column**



- A minimal DataFrame implementation would be a dict where each element is a list

Series

	id
0	23
1	42
2	12
3	86

+

Series

	Name
0	“Bob”
1	“Karen”
2	“Kate”
3	“Bill”

=

DataFrame

	id	Name
0	23	“Bob”
1	42	“Karen”
2	12	“Kate”
3	86	“Bill”

Indexing and Slicing

- pandas supports various ways of indexing the contents of a DataFrame
- [<column name>] - select a given column by it's name. column names can also be used as field names
- .loc[<row name>] - select a given row by it's (Index) name
- .iloc[<position>] - select a given row by it's position in the Index, starting at 0
- .loc[<row name>, <column name>] - select an individual element by row and column name
- .iloc[<row position>, <row position>] - select an individual element by row and column position (starting at 0)
- .iloc also supports ranges and slices similarly to python lists or numpy arrays

Importing and Exporting Data

- pandas has powerful methods to read and write data from multiple sources
  - pd.read_csv() - Read a comma-separated values file
    - sep=' ' - Define the separator to use. ',' is the default
    - header=0 - Row number to use as column names
  - pd.read_excel() - Read an Excel file
    - sheet_name - The sheet name to load
  - pd.read_html() - Read tabular data from a URL (or local html file)
  - pd.read_pickle() - Read a Pickle file
- Each of these functions accepts a large number of options and parameters controlling its behavior. Use help(<function name>) to explore further.
- Each read_*() function has a complementary to_*() function to write out a DataFrame to disk. The to_*() functions are members of the DataFrame object

Transformed values

- We often need to apply some simple transformation to the values in our original Series. Pandas offers several methods to accomplish this goal:
  - map(func) - Map values of Series according to input correspondence.
    - func - function, dict or Series to use for mapping
  - transform(func) - Transform each column/row of the DataFrame using a function. Output must have the same shape as the original
    - axis = 0 - apply function to columns
    - axis = 1 - apply function to rows
- apply(func) - Apply a function along an axis of the DataFrame
  - Similar to transform() but without the limitation of preserving the shape

groupby

- Sometimes we need to calculate statistics for subsets of our data
- groupby() allows us to group data based on the values of a column
- groupby() returns a GroupBy object that supports several aggregations functions, including:
    - max()/min()/mean()/median()
    - transform()/apply()
    - sum()/cumsum()
    - prod()/cumprod()
    - quantile()
- Each of these functions is applied to the contents of each group

Pivot Tables

- Pandas provides an extremely flexible pivot table implementation
- pd.pivot_table() - Creates a spreadsheet-style pivot table as a DataFrame.
- Three important arguments:
    - values - column to aggregate
    - index - Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
    - columns - Keys to group by on the pivot table column.
    - aggfunc - Function to use for aggregation. Defaults to np.mean()
- index, columns and aggfunc can also be lists of keys or functions to use.

merge/join

- merge() and join() allows us to perform database-style join operation by columns or indexes (rows)

- Some of the most important arguments are common to both methods:

    - on - Column(s) to use for joining, otherwise join on index. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

    - how - Type of join to perform: {'left', 'right', 'outer', 'inner'}

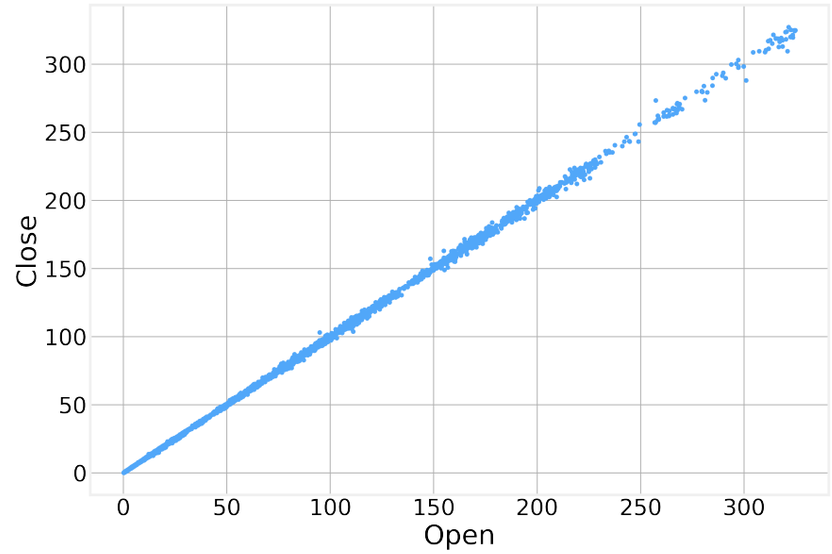
- merge() is more sophisticated and flexible. It also allows us to specify:

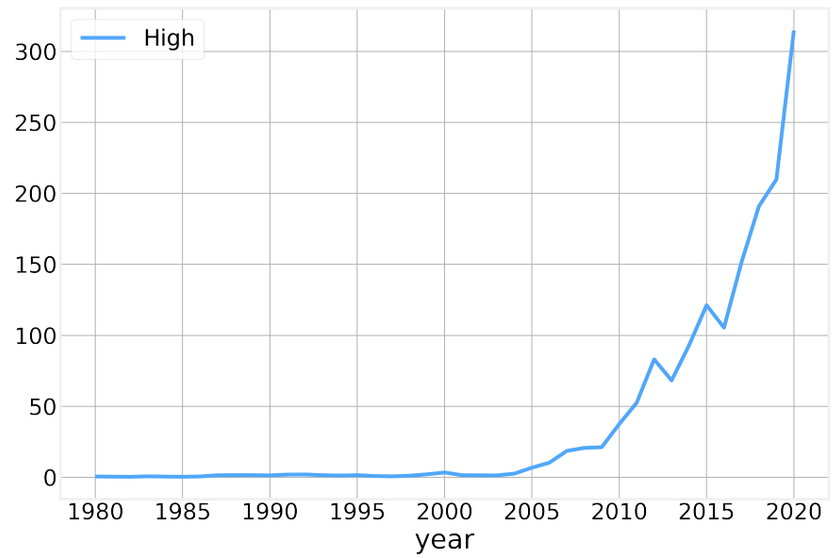
    - left_on/right_on - Field names to join on for each DataFrame

    - left_index/right_index - Whether or not to use the left/right index as join key(s)

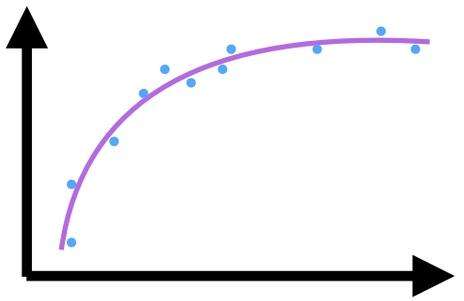
plotting

- Finally, pandas also provides a simple interface for basic plotting through the plot() function
- By specifying some basic parameters the variables plotted and even the kind of plot can be easily modified:
  - **x/y** -column name to use for the **x/y** axis
  - **kind** - type of plot
    - ‘line’ - line plot (default)
    - ‘bar’/‘barh’ - vertical/horizontal bar plot
    - ‘hist’ - histogram
    - ‘box’ - boxplot
    - ‘pie’ - pie plot
    - ‘scatter’ - scatter plot
- The plot() function returns a matplotlib Axes object





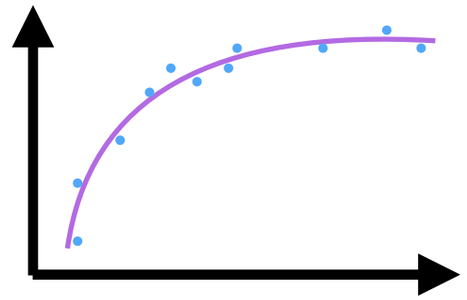
Basic Plotting



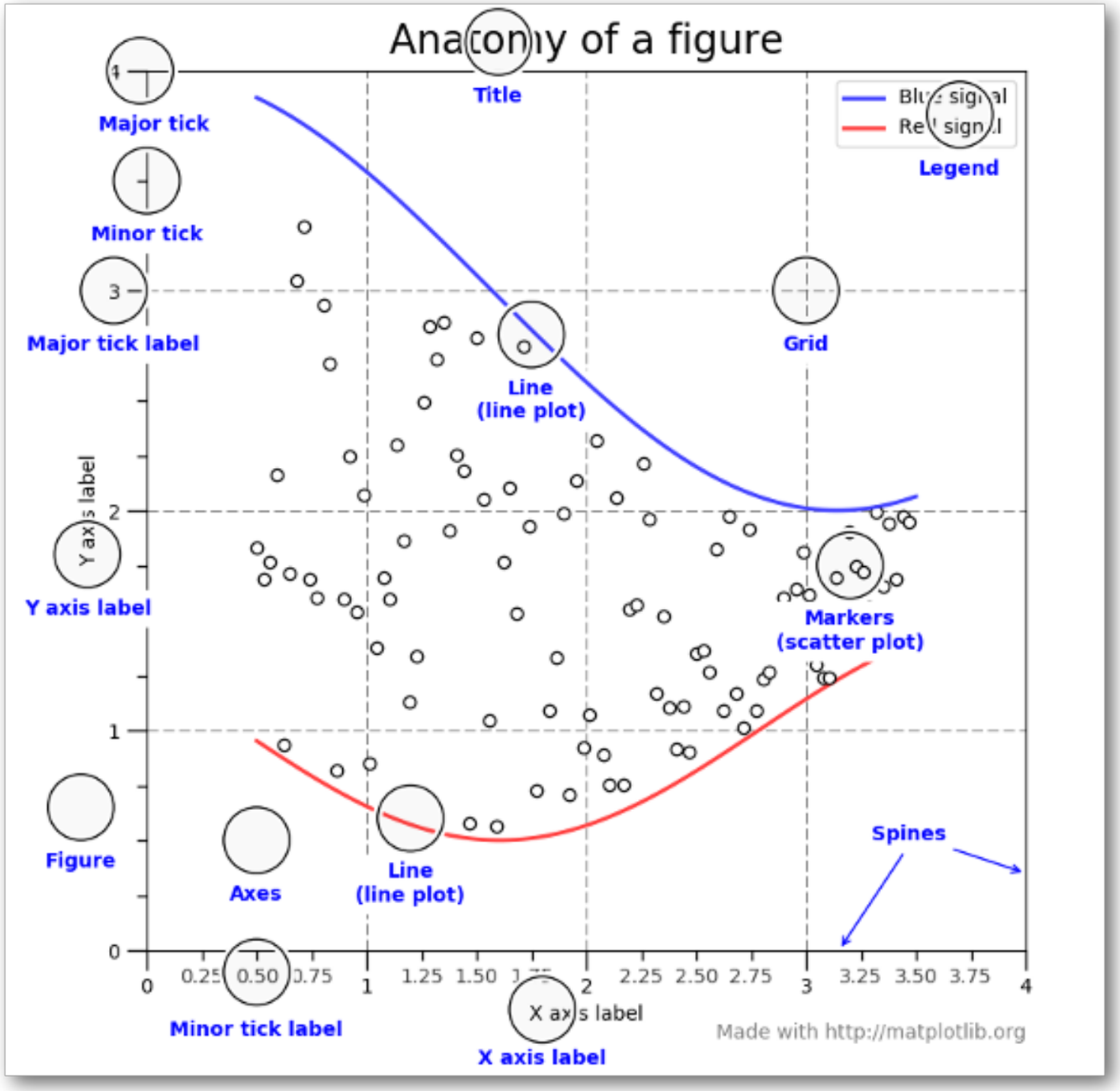
matplotlib.org/3.1.0/

- Matplotlib uses an object oriented structure following an intuitive notation
- Each Axes object contains one or more Axis objects.
- A Figure is a set of one or more Axes.
- Each Axes is associated with exactly one Figure and each set of Markers is associated with exactly one Axes.
- In other words, Markers/Lines represent a dataset that is plotted against one or more Axis. An Axes object is (effectively) a subplot of a Figure.

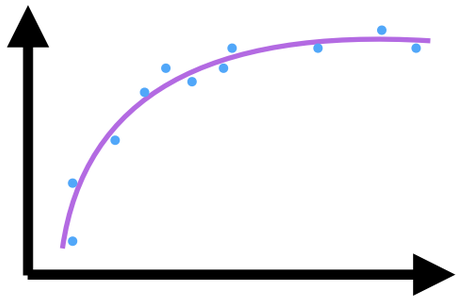
Basic Plotting - Programmatically!



matplotlib.org/3.1.0/



Basic Plotting - Programmatically!

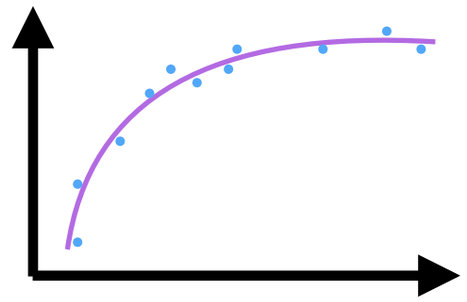


matplotlib.org/3.1.0/

- While the Figure object controls the way in which the figure is displayed.
  - .gca() - Get the current Axes, creating one if necessary
  - .show() - Show the final figure
  - .savefig("filename.ext", dpi=300) - Save the figure to "filename.ext" where ".ext" defines the format the saved image ()

```
filetypes = {'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw
RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'jpg': 'Joint
Photographic Experts Group', 'jpeg': 'Joint Photographic Experts Group', 'tif': 'Tagged Image File
Format', 'tiff': 'Tagged Image File Format'}
```

Basic Plotting - Programmatically!



matplotlib.org/3.1.0/

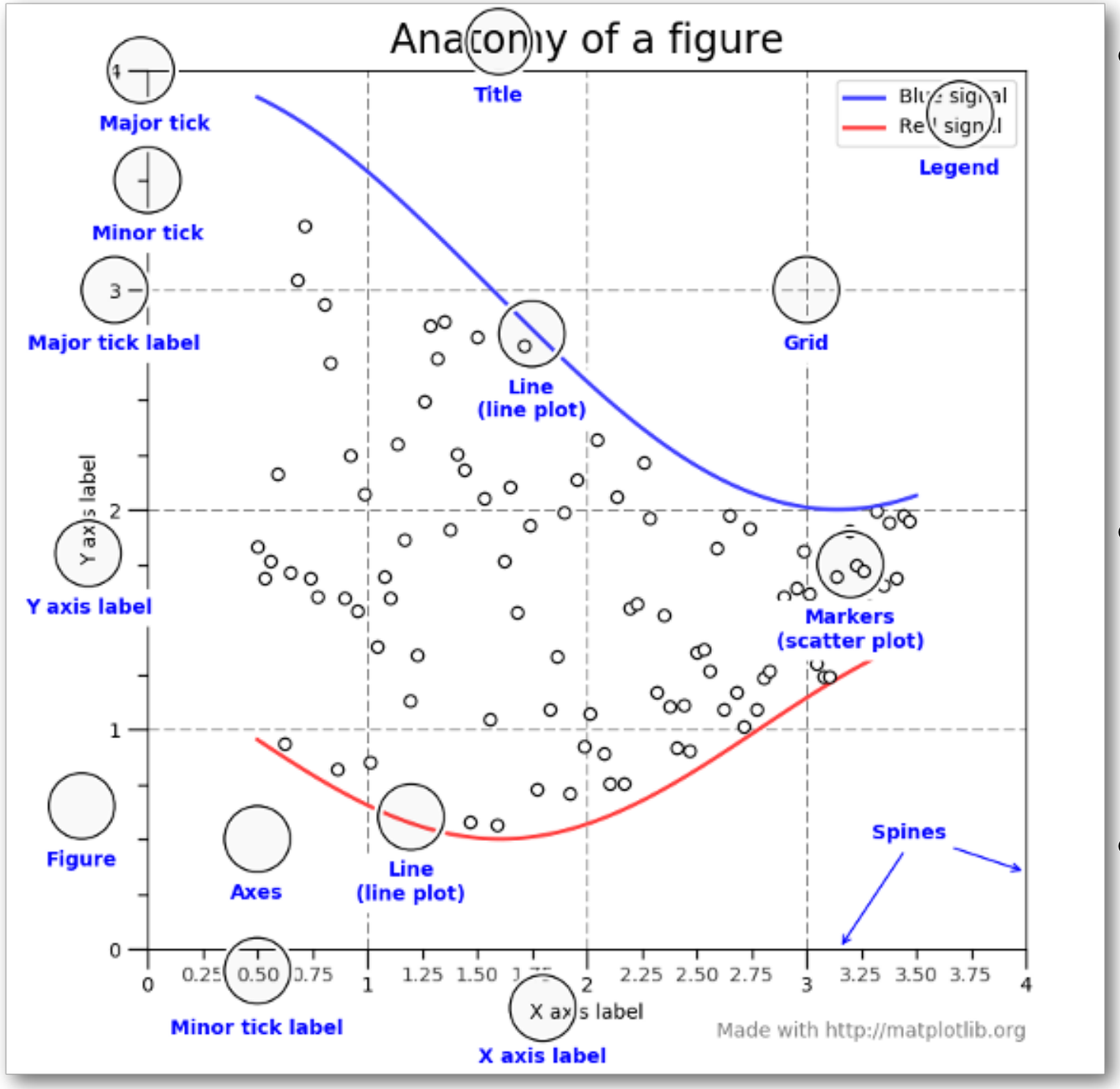
- Import the pyplot module from matplotlib

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

- Axes have several methods of interest:
  - .plot(x, y) - Make a scatter or line plot from a list of x, y coordinates.
  - .imshow(mat) - Plot a matrix as if it were an image. Element 0,0 is plotted in the top right corner.
  - .bar(x, y) - Make a bar plot where x is a list of the lower left coordinates of each bar and y is the respective height.
  - .pie(values, labels=labels) - Produce a pie plot out of a list of values list and labeled with labels
  - .savefig(filename) - Write the current figure as an static image

matplotlib - decorations

matplotlib.org/3.1.0/



- The respective functions are named in an intuitive way, Every Axes object has as methods:
  - .set_xlabel(label)
  - .set_ylabel(label)
  - .set_title(title)
- And axis limits can be set using:
  - .set_xlim(xmin, xmax)
  - .set_ylim(ymin, ymax)
- Tick marks and labels are set using:
  - .set_xticks(ticks)/.set_yticks(ticks)
  - .set_xticklabels(labels)/.set_yticklabels(labels)

matplotlib - Images

matplotlib.org/3.1.0/

- plt.imshow(fig) - Display an image on a set of axes.
- plt.imshow(fig, extent=(xllcorner, xurcorner, yllcorner, yurcorner), zorder=-1)
- fig can be any matrix of numbers.
- Further plotting can occur by simply using the functions described above

Code - Pandas

https://github.com/DataForScience/InteractiveViz

2. matplotlib animations

The matplotlib Animation API

https://matplotlib.org/stable/api/animation_api.html

- matplotlib supports simple animations through the Animation API
- Two main Animation classes:
  - FuncAnimation - generate an animation by repeatedly calling a given function func() to draw each frame.
  - ArtistAnimation - creates an animation by using a fixed set of Artist objects.
- FuncAnimation is the simplest and most intuitive approach, so it's the approach we will follow here
- Animations can be displayed in a pop-up window, a notebook cell, or saved as a movie or GIF file

FuncAnimation

https://matplotlib.org/stable/api/animation_api.html

- FuncAnimation takes several important parameters:
  - fig - a handle to the figure object being used
  - func - the function that plots each frame. This function should take the frame number as the first argument.
  - frames=None - the number of frames or a list of frames to generate
  - interval - number of milliseconds between frames
  - init_func=None - a function that creates the initial, empty, figure. Must return a list of Artist objects that will be updated in each frame
  - fargs=None - Any further parameters required by func()
- Instantiating the FuncAnimation object returns an animation object that we can use to generate the animation.
- Before we can watch the animation, we must call a writer method

FuncAnimation

https://matplotlib.org/stable/api/animation_api.html

```
fig = plt.figure()
ax = plt.axes(xlim=(0, 4), ylim=(-2, 2))
line, = ax.plot([], [], lw=3)

def init():
    line.set_data([], [])
    return line,

def animate(i):
    x = np.linspace(0, 4, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,


anim = FuncAnimation(fig, animate, init_func=init,
                                   frames=200, interval=20, blit=True,
                                   repeat=True)
```

- Instantiating the FuncAnimation object returns an animation object that we can use to generate the animation.

- Before we can watch the animation, we must call a writer method

Animation writers

https://holypython.com/how-to-save-matplotlib-animations-the-ultimate-guide/

- There are three possible writer methods:
  - Animation.save() - Save the animation as a movie, frame by frame
    - The file format can be specified by using the file extension

```
15 # Save the animation as a GIF
16 anim.save('data/vulcano.gif')
```

    - A more robust alternative is to provide a FFMpegWriter instance, which supports a wider range of formats

```
15 writervideo = FFMpegWriter(fps=60)
16 anim.save("data/vulcano.mp4", writer=writervideo)
```

    - This might require installing ffmpeg separately (https://ffmpeg.org/download.html)
  - Animation.to_jshtm() - Generate HTML representation of the animation.
  - Animation.to_html5_video() - Convert the animation to an HTML5 <video> tag with the video encoded directly inside the tag. This is ideal for integrating into Jupyter notebooks

Animation.to_html5_video()

- It is possible to view the animation directly within the notebook
- Embedding the animation within the notebook is a three step process:
    - Generate the html5 video using Animation.to_html5_video()
    - Generate the html embedding using jupyters display.HTML()
    - Display the video by calling display.display()

```
20 # Display the animation as an HTML video
21 video = anim.to_html5_video()
22 html = display.HTML(video)
23 display.display(html)
```

Code - matplotlib

https://github.com/DataForScience/InteractiveViz

3. Jupyter ipywidgets

ipywidgets

https://ipywidgets.readthedocs.io/en/stable/index.html

- Jupyter Widgets are interactive browser controls for Jupyter notebooks
- Widgets come in a wide range of forms:
  - Numeric widgets
  - Boolean widgets
  - Selection widgets
  - String widgets
  - Image
  - Button
  - Output
  - Date/Color pickers
  - Color picker
  - File Upload
- Interactive widgets allow users to visualize and manipulate data in intuitive and easy ways.

Simple widget use

https://ipywidgets.readthedocs.io/en/7.x/examples/Using%20Interact.html

- The simplest way to use the widgets is to use the ipywidgets.interact() function decorator
- @ipywidgets.interact - automatically generates UI controls based on the arguments (and default values) of the function to which it is applied
    - string - Text
    - boolean - Checkbox
    - integer/float - IntSlider/FloatSlider
    - list/dict - Dropdown - The dict allows the dropdown box to display one value (the key) while passing a different value to the function (the value)
- Fixed arguments to the function can be specified with the fixed() function as the default value.
- The return value of the function will be displayed automatically, making it particularly useful for functions that produce visualizations
- @ipywidgets.interact_manual - is similar to @ipywidgets.interact but it adds an extra Button to execute the function instead of automatically executing it as soon as any of the parameters change

Simple widget use

https://ipywidgets.readthedocs.io/en/7.x/examples/Using%20Interact.html

- Adding the @widgets.interact decorator directly above the function definition produces a simple GUI to manipulate the function arguments

```
1 @widgets.interact
2 def plot_continent(cont = cont_dict, log_scale=True, font_size=font_sizes):
```

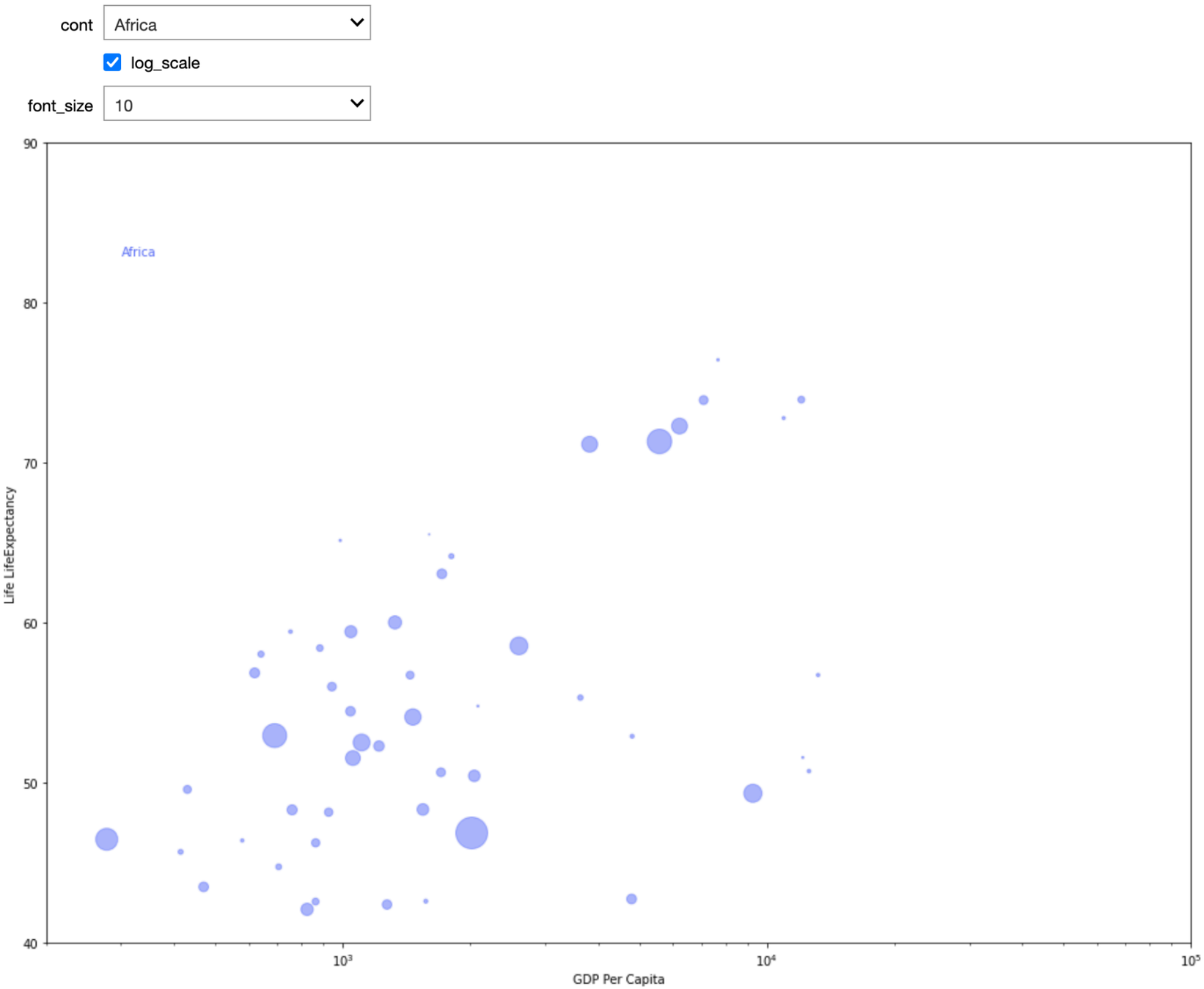
- This function takes three arguments, a dictionary, a boolean, and a. list, resulting in:

immediately above the output of the function.

- Each GUI element is labeled with the argument name it controls

Simple widget use

https://ipywidgets.readthedocs.io/en/7.x/examples/Using%20Interact.html



Simple widget use

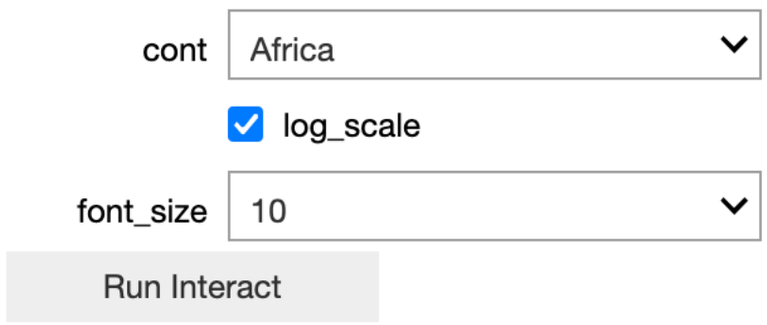
https://ipywidgets.readthedocs.io/en/7.x/examples/Using%20Interact.html

- We can also use both widgets.interact() and the widgets.interact_manual() as functions, where the first argument is the function we want it to control. Every other argument is an argument that our function takes.

- The previous example could also be produced using

```
widgets.interact(plot_continent,
                 cont = cont_dict,
                 log_scale=True,
                 font_size=font_sizes);
```

- If instead, we switch to widgets.interact_manual() we obtain an extra button to manually execute the function



Widget customizing

- We can also manually instantiate the widget we wish to use, and pass it directly as the default value to the respective function argument.

- A custom version of our font size selector can be created using:

```
font_slider = widgets.IntSlider(
    value=18,
    min=10,
    max=40,
    step=2,
    description='Font size:',
    disabled=False,

    # Should the figure be updated as the value is being changed?
    continuous_update=False,
    orientation='horizontal',
    #readout=True,

    # Show only decimal values
    readout_format='d'
)
```

- And passed directly to the function:

```
widgets.interact(plot_continent,
                 cont = cont_dict,
                 log_scale=True,
                 font_size=font_slider);
```

Widget customizing

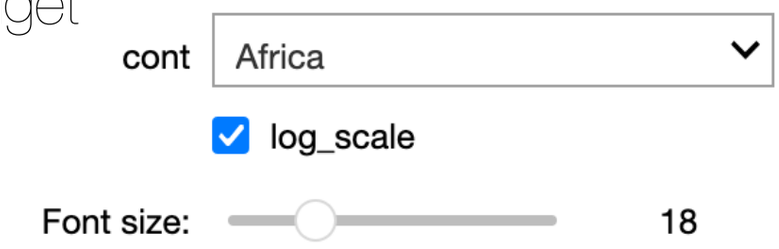
- We can also manually instantiate the widget we wish to use, and pass it directly as the default value to the respective function argument.

- A custom version of our font size selector can be created using:

```
font_slider = widgets.IntSlider(
    value=18,
    min=10,
    max=40,
    step=2,
    description='Font size:',
    disabled=False,

    # Should the figure be updated as the value is being changed?
    continuous_update=False,
    orientation='horizontal',
    #readout=True,

    # Show only decimal values
    readout_format='d'
)
```

- And passed directly to the function, to produce the custom widget

```
widgets.interact(plot_continent,
                 cont = cont_dict,
                 log_scale=True,
                 font_size=font_slider);
```



Connecting widgets

https://ipywidgets.readthedocs.io/en/7.x/examples/Using%20Interact.html

- Sometimes we want to create dependencies between the values of the different widgets
- Each widget has an observe() method that allows us to specify a handler function to be called whenever one of its specific traits (value, color, size, etc) changes.
- The handler function can then interact and modify other widgets as it sees fit.
- We can instruct the continent Dropdown widget to modify the maximum font size of the font IntSlider whenever its value changes by simply using:

```
23 def update_font_range(*args):
24     new_font_slider.max = 20+2.0 * continent_widget.value
25
26 # Tell the continent widget to call update_font_range
27 # whenever it's value changes
28 continent_widget.observe(update_font_range, 'value')
```

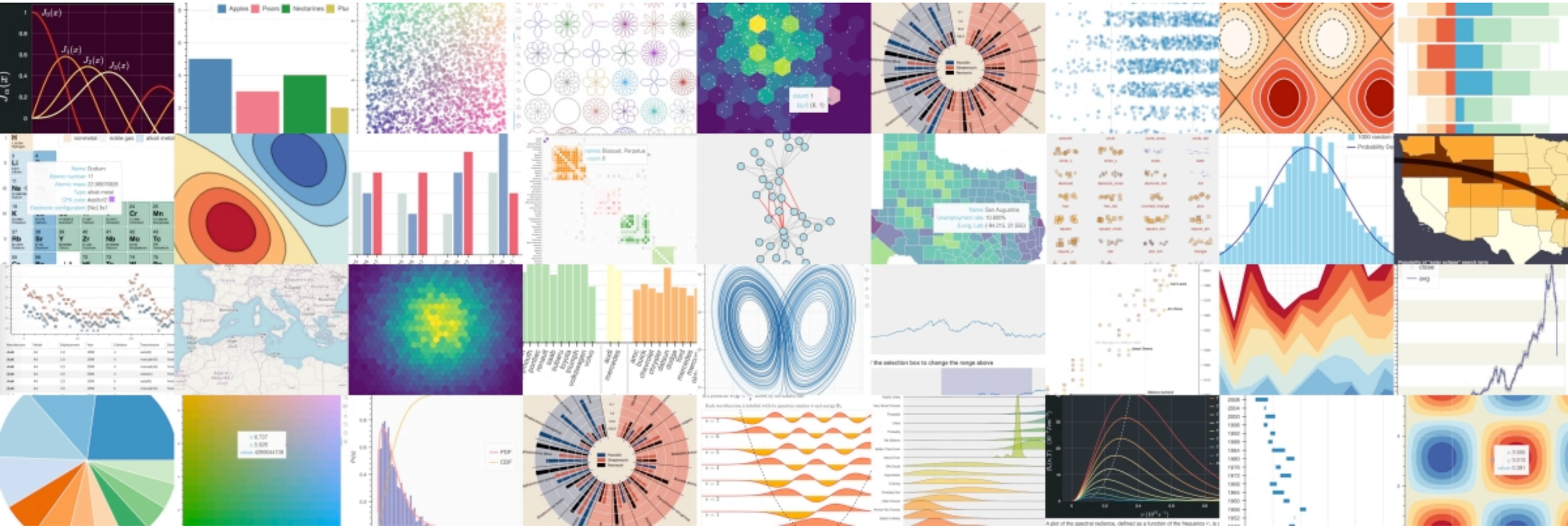
Code - ipywidgets

https://github.com/DataForScience/InteractiveViz

4. Bokeh

bokeh.org

- Interactive visualization library for Python
- Javascript backend that supports all modern browsers
- Optimized to Interactively Explore Data in Notebooks
- Supports Streaming Data



- bokeh is powerful but its structure can sometimes complex
- It relies on a layered structured of submodules:
  - bokeh.models - responsible for handling the JSON data created by the JavaScript backend. models are fairly simple
  - bokeh.plotting - mid-level interface focusing on Matplotlib like features. It handles the creation of axes, grids, and tools. It contains the figure(), function, our workhorse
  - bokeh.io - functions to handle the input/output, such as output_file(), output_notebook(), and show() functions.
  - bokeh.palettes - color palettes like Viridis256, Category20c, Spectral6, GnBu3, OrRd3, etc
  - bokeh.layouts - functions to layout multiple figures in the same plot

Basic Plotting

- The first step with any Bokeh plot is to generate a figure object

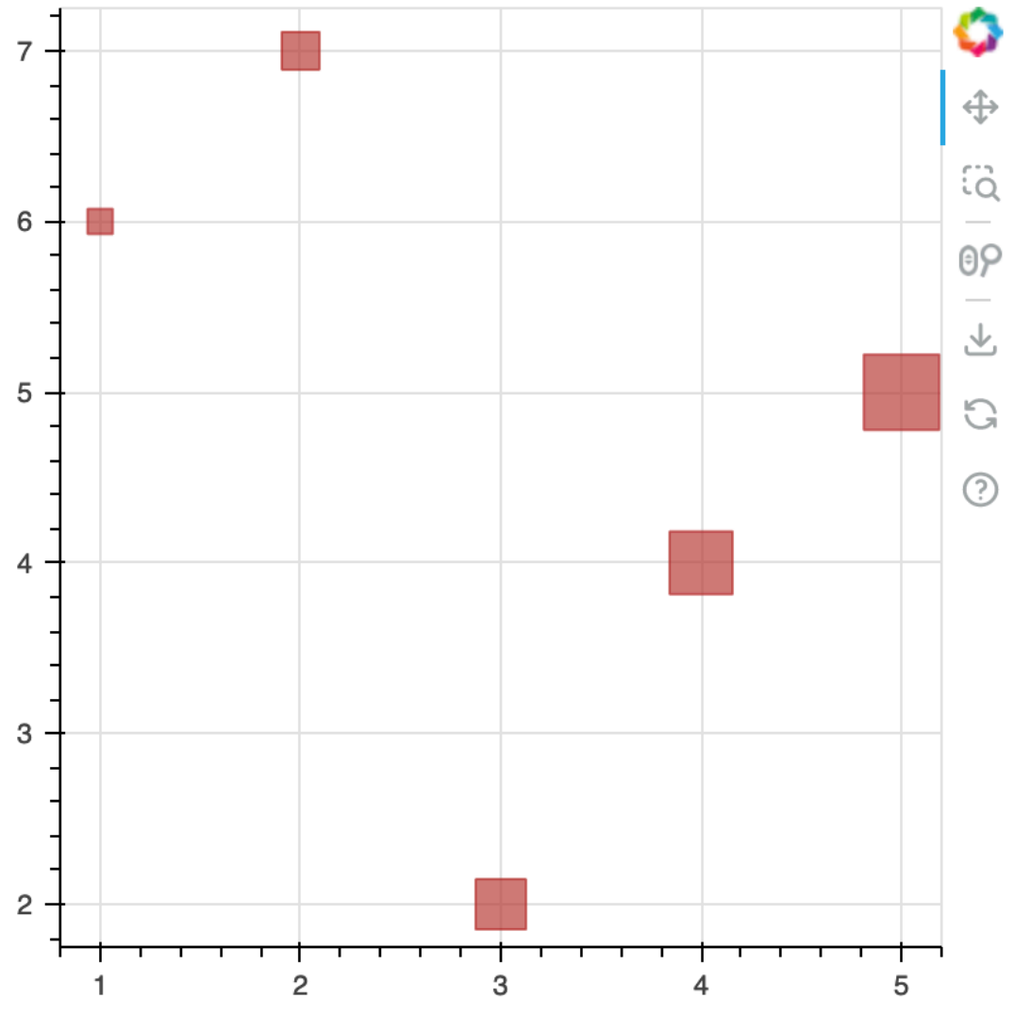
```
1 p = figure(width=400, height=400)
```

- Most plotting functions exist as methods of the figure object
  - p.circle() - Scatter plot using circles
  - p.square() - Scatter plot using squares
  - p.line() - Line plot
  - p.wedge() - Pie plot
  - p.annular_wedge() - Donut plot
  - p.vbar() / p.hbar() - Vertical and Horizontal bars
  - p.vbar_stack() / p.hbar_stack() - Stacked Vertical and Horizontal bars
- Figures are only displayed when you call show(p)

Basic Plotting

- Simple figures can be generated by calling any of the methods above

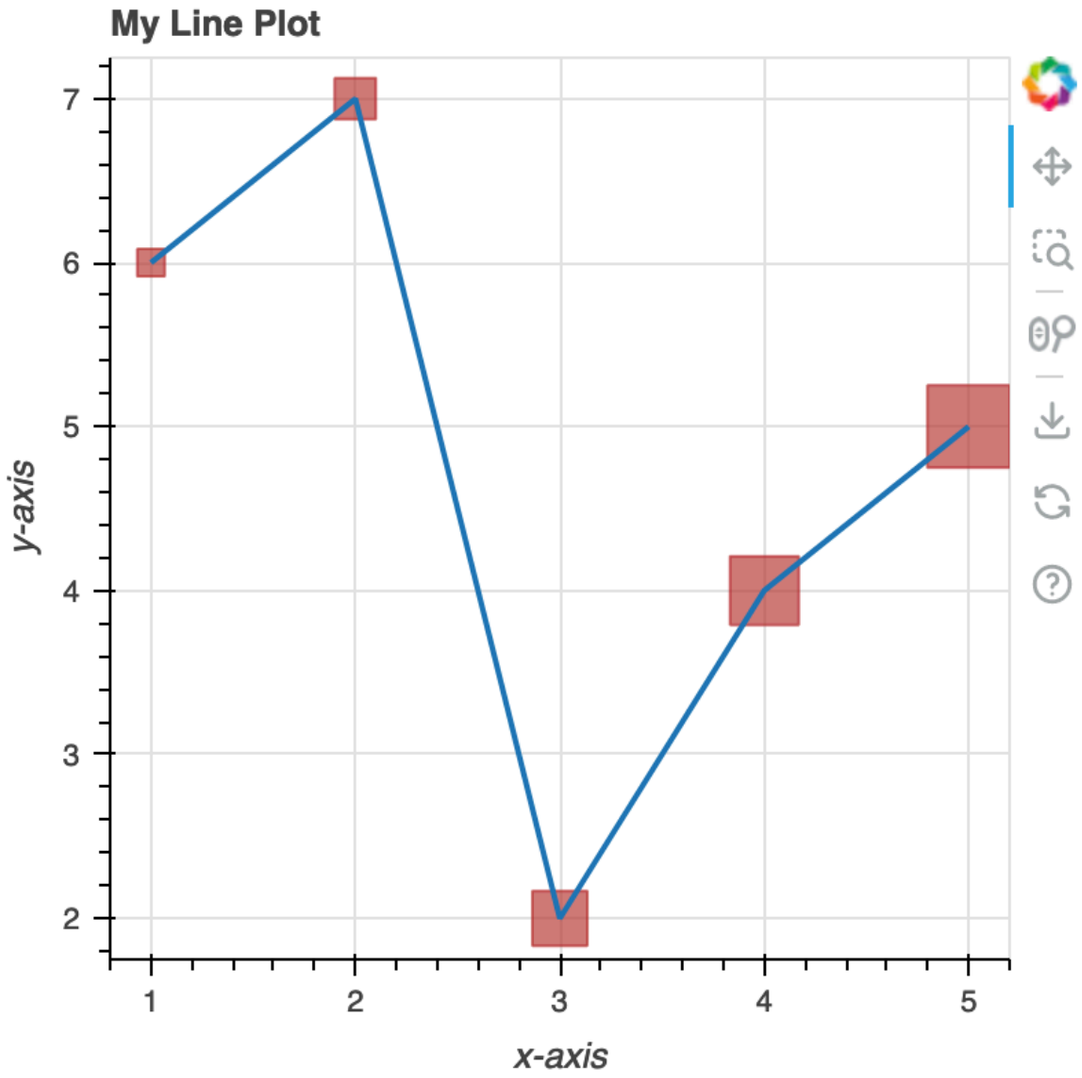
```
p = figure(width=400, height=400)

# Use a different size for each symbol
p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
         size=[10, 15, 20, 25, 30],
         color="firebrick", alpha=0.6)
show(p)
```



Basic Plotting

- Simple figures can be generated by calling any of the methods above
- More complex figures can be generated by calling multiple methods

```
p = figure(width=400, height=400, title="My Line Plot")

# add a line renderer
p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
         size=[10, 15, 20, 25, 30],
         color="firebrick", alpha=0.6)
p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
p.xaxis.axis_label = 'x-axis'
p.yaxis.axis_label = 'y-axis'

show(p)
```

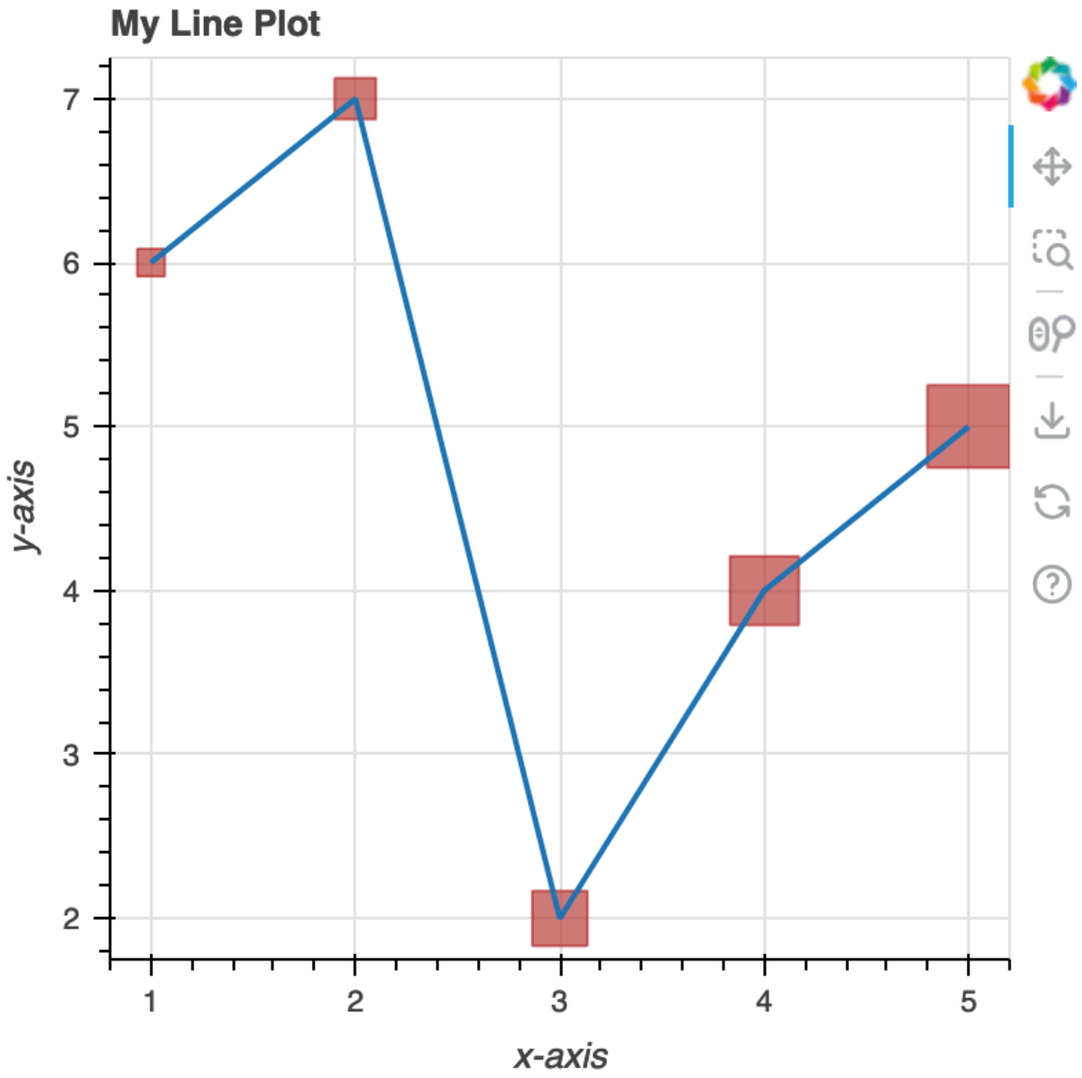


Basic Plotting

- Simple figures can be generated by calling any of the methods above
- More complex figures can be generated by calling multiple methods

```
p = figure(width=400, height=400, title="My Line Plot")

# add a line renderer
p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
         size=[10, 15, 20, 25, 30],
         color="firebrick", alpha=0.6)
p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
p.xaxis.axis_label = 'x-axis'
p.yaxis.axis_label = 'y-axis'

show(p)
```

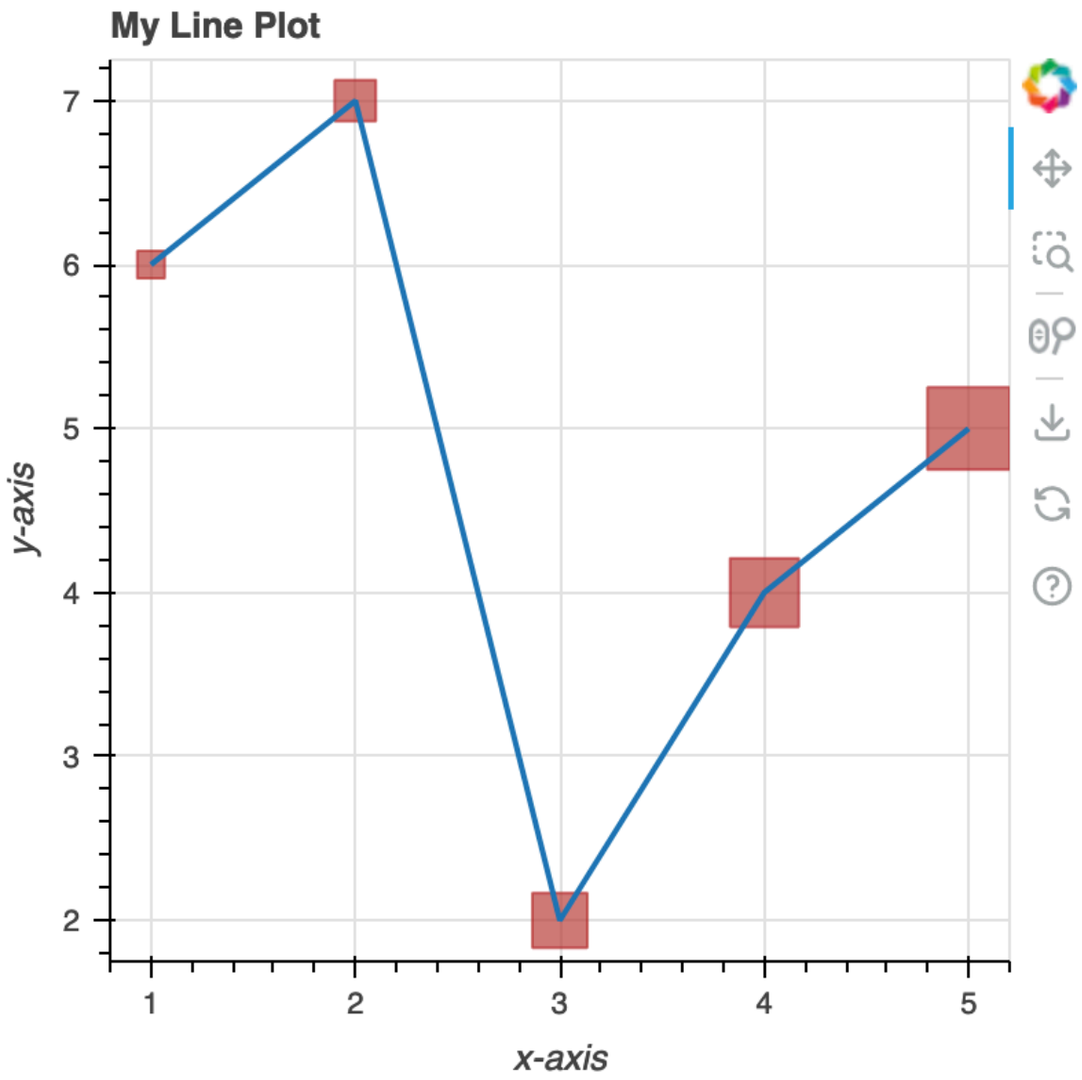
- Axis labels can be added by simple assignment



Basic Plotting

- Simple figures can be generated by calling any of the methods above
- More complex figures can be generated by calling multiple methods

```
1  p = figure(width=400, height=400, title="My Line Plot")
2
3  # add a line renderer
4  p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
5           size=[10, 15, 20, 25, 30],
6           color="firebrick", alpha=0.6)
7  p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
8  p.xaxis.axis_label = 'x-axis'
9  p.yaxis.axis_label = 'y-axis'
10
11 show(p)
```



- Axis labels can be added by simple assignment
- The toolbar on the right allows us to zoom in, pan, select specific points, download a copy of the image, etc

Pie and Donut charts

- p.wedge() and p.annular_wedge() take a similar set of arguments:
  - x / y - Location on which to generate the plot
  - start_angle / end_angle - start and end angle of each slice
  - legend_field - the field (column) name to use for the legend
  - source - the data source
- The main difference is that p.wedge() takes a single radius argument, while p.annular_wedge() takes both an inner_radius and an outer_radius
- Complex visualizations often rely on complex datasets. Bokeh makes it easy to use pandas DataFrames as data sources, where values can be referred to by column names.

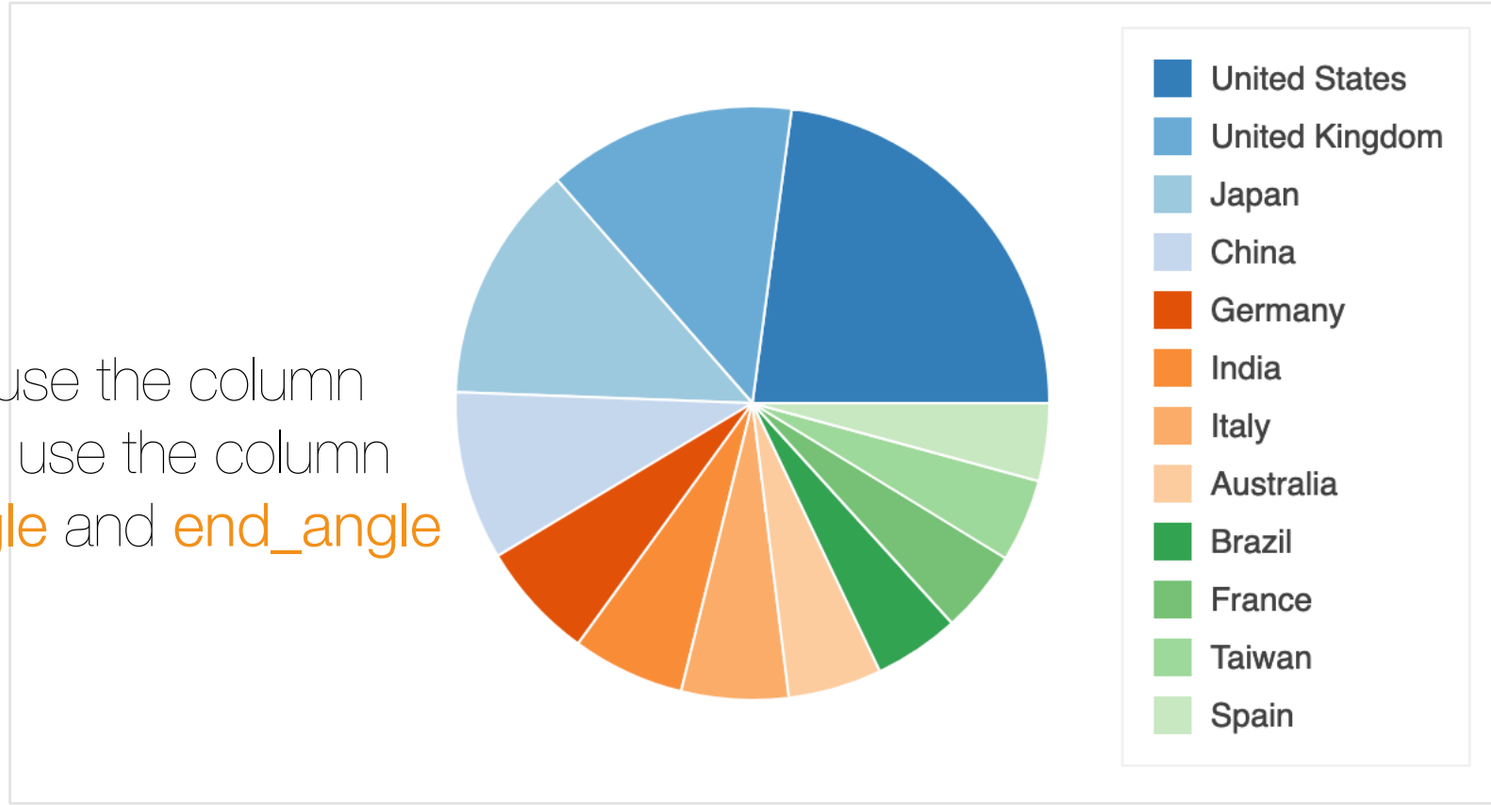
Pie and Donut charts

- Using this DataFrame we can generate a Pie plot using:

```
p = figure(height=350, title="Pie Chart", toolbar_location=None,
           tools="hover", tooltips="@country: @value")

p.wedge(x=0, y=1, radius=0.4,

        # use cumsum to cumulatively sum the values for start and end angles
        start_angle=cumsum('angle', include_zero=True),
        end_angle=cumsum('angle'),
        line_color="white",
        fill_color='color',
        legend_field='country', source=data)

p.axis.axis_label=None
p.axis.visible=False
p.grid.grid_line_color = None

show(p)
```

	country	value	color	angle
0	United States	157	#3182bd	1.437988
1	United Kingdom	93	#6baed6	0.851802
2	Japan	89	#9ecae1	0.815165
3	China	63	#c6dbef	0.577027
4	Germany	44	#e6550d	0.403003
5	India	42	#fd8d3c	0.384685
6	Italy	40	#fdae6b	0.366366
7	Australia	35	#fdd0a2	0.320571
8	Brazil	32	#31a354	0.293093
9	France	31	#74c476	0.283934
10	Taiwan	31	#a1d99b	0.283934
11	Spain	29	#c7e9c0	0.265616

- where we specify that we want to use the column ‘color’, for the wedge colors, and use the column ‘angle’ to compute the start_angle and end_angle

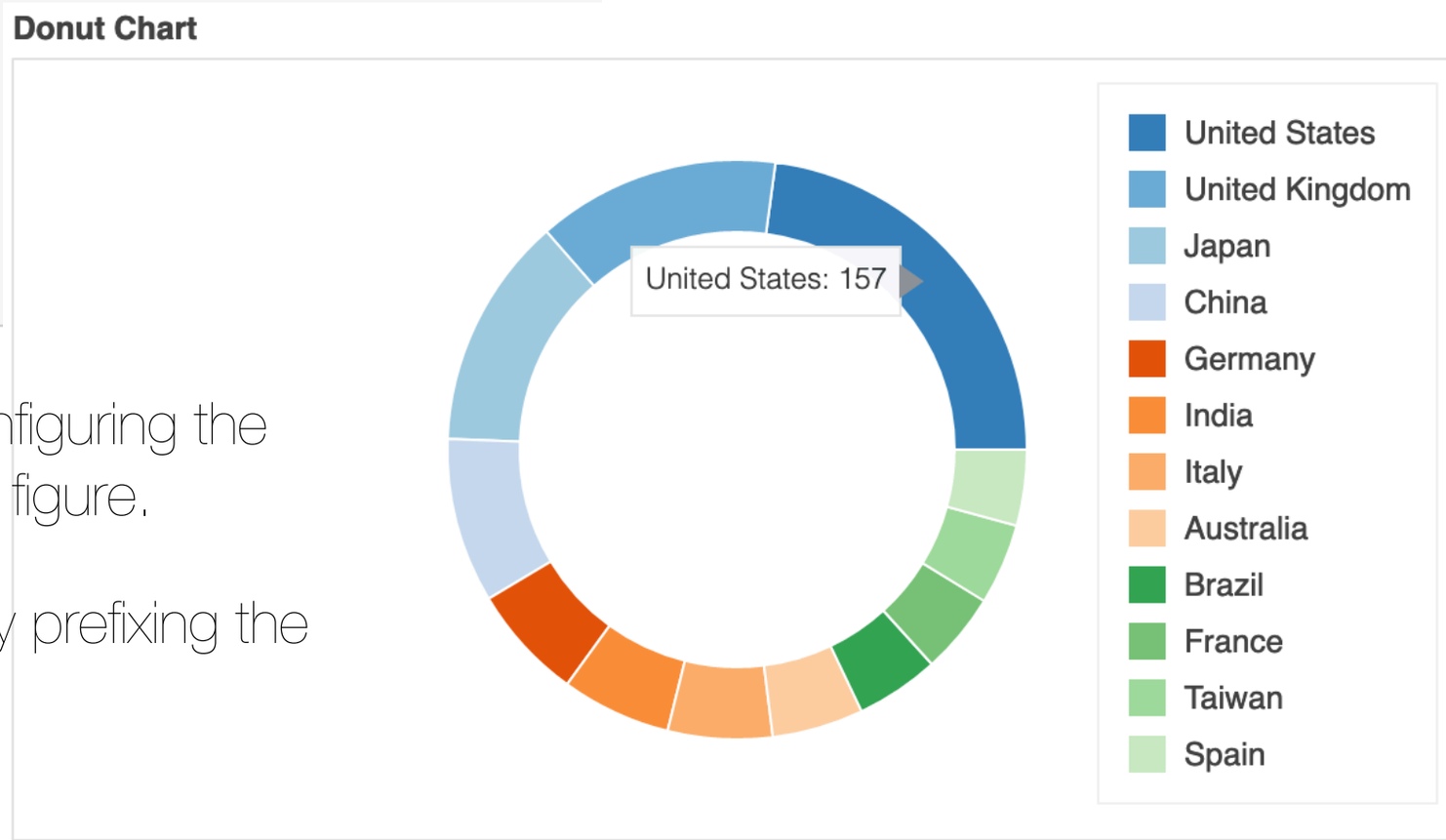


Pie and Donut charts

- And a donut plot using:

```
p = figure(height=350, title="Donut Chart", toolbar_location=None,
           tools="hover", tooltips="@country: @value")

p.annular_wedge(x=0, y=1,
        inner_radius=0.3, outer_radius=0.4,

        # use cumsum to cumulatively sum the values for start and end angles
        start_angle=cumsum('angle', include_zero=True),
        end_angle=cumsum('angle'),
        line_color="white", fill_color='color',
        legend_field='country', source=data)

p.axis.axis_label=None
p.axis.visible=False
p.grid.grid_line_color = None

show(p)
```

	country	value	color	angle
0	United States	157	#3182bd	1.437988
1	United Kingdom	93	#6baed6	0.851802
2	Japan	89	#9ecae1	0.815165
3	China	63	#c6dbef	0.577027
4	Germany	44	#e6550d	0.403003
5	India	42	#fd8d3c	0.384685
6	Italy	40	#fdae6b	0.366366
7	Australia	35	#fdd0a2	0.320571
8	Brazil	32	#31a354	0.293093
9	France	31	#74c476	0.283934
10	Taiwan	31	#a1d99b	0.283934
11	Spain	29	#c7e9c0	0.265616



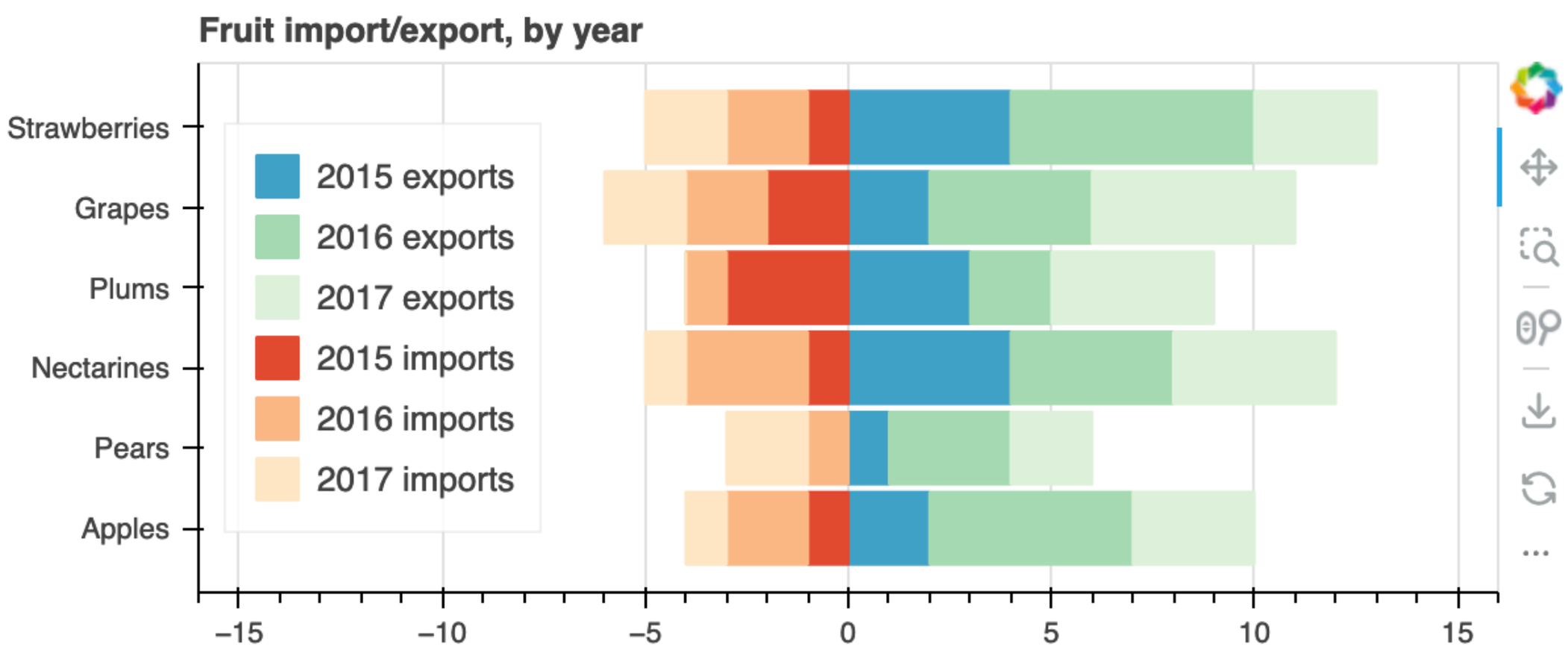
- In both cases, we add tooltips by configuring the hover tool, when we setup the basic figure.

- source columns can be referred to by prefixing the column name with the @ symbol

Advanced Plotting

- Sophisticated plots can be generated with just a few lines of straightforward code

```
p = figure(y_range=fruits, height=250, x_range=(-16, 16),
           title="Fruit import/export, by year")

p.hbar_stack(years, y='fruits', height=0.9, color=GnBu3,
             source=ColumnDataSource(exports),
             legend_label=["%s exports" % x for x in years])

p.hbar_stack(years, y='fruits', height=0.9, color=OrRd3,
             source=ColumnDataSource(imports),
             legend_label=["%s imports" % x for x in years])

p.y_range.range_padding = 0.1
p.ygrid.grid_line_color = None
p.legend.location = "center_left"

show(p)
```

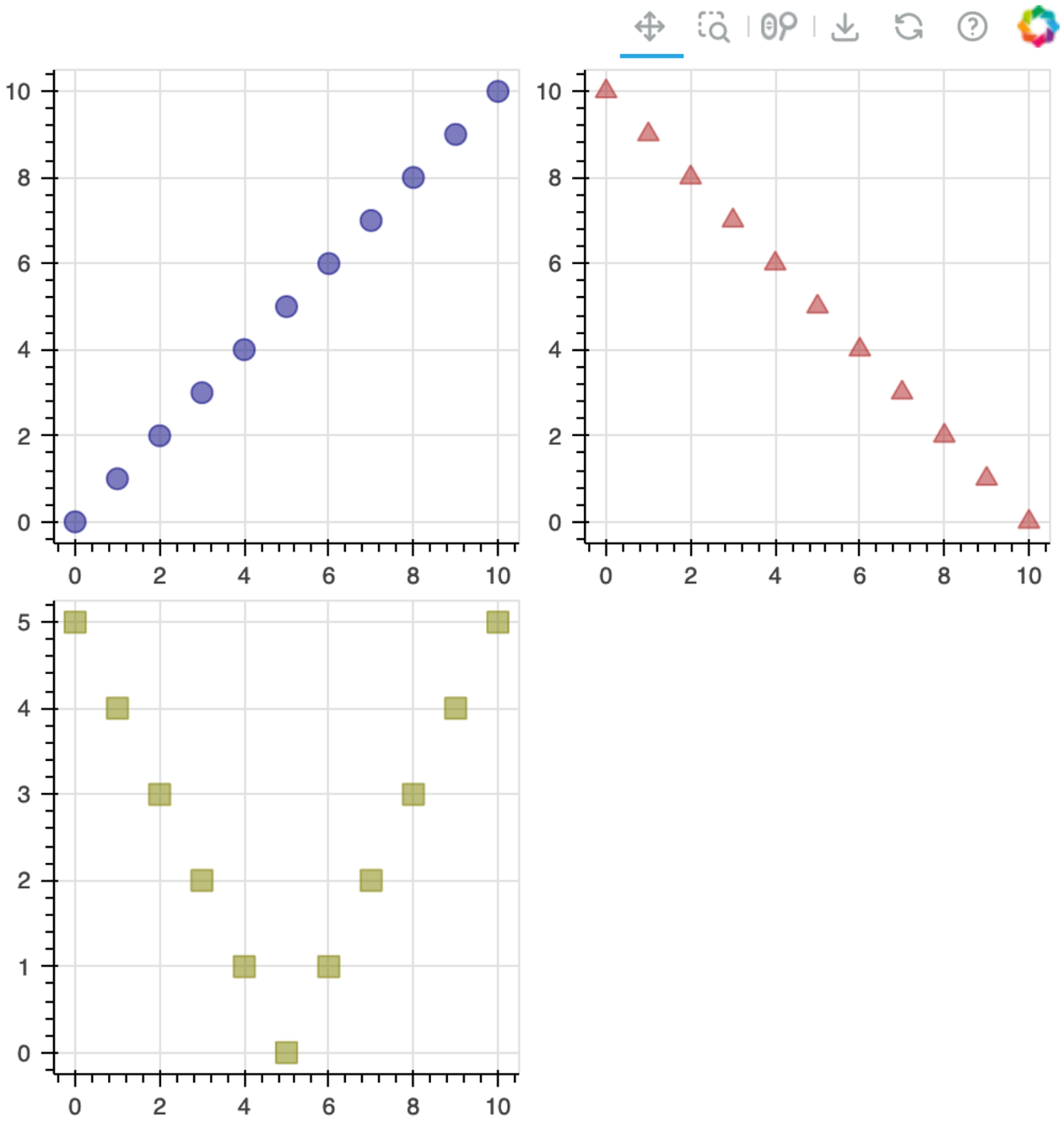


Multiple plots

- Multiple figure objects can be put together using the gridplot() command

- gridplot() takes a 2D array of image objects as the main argument

```
16 # put all the plots in a gridplot
17 p = gridplot([[s1, s2], [s3, None]])
```



Linking plots

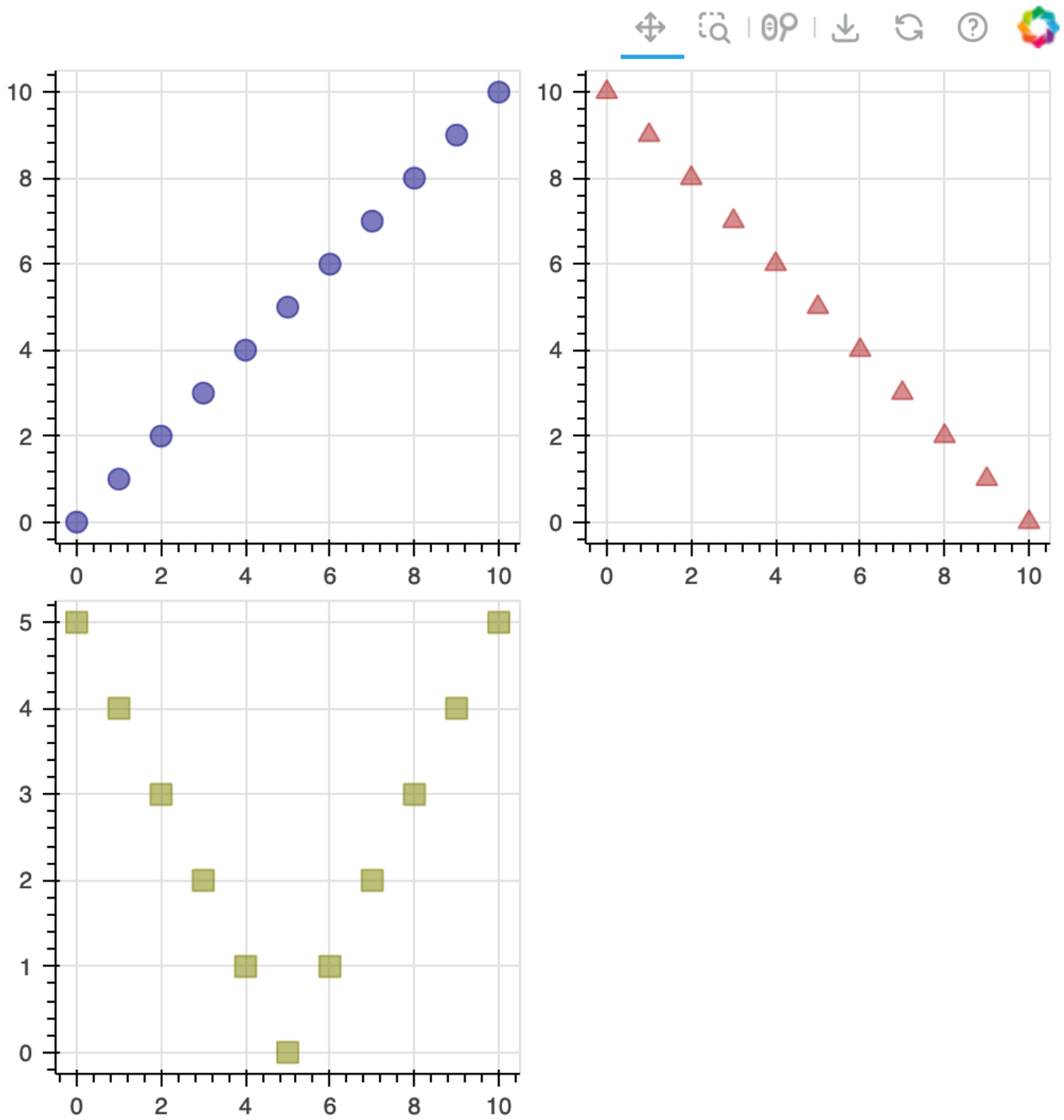
- Multiple figure objects can be put together using the gridplot() command

- gridplot() takes a 2D array of image objects as the main argument

```
16 # put all the plots in a gridplot
17 p = gridplot([[s1, s2], [s3, None]])
```

- Each subplot can be generated and configured separately

- Plots can be connected by passing references to other objects' settings



Linking plots

- Multiple figure objects can be put together using the gridplot() command

- gridplot() takes a 2D array of image objects as the main argument

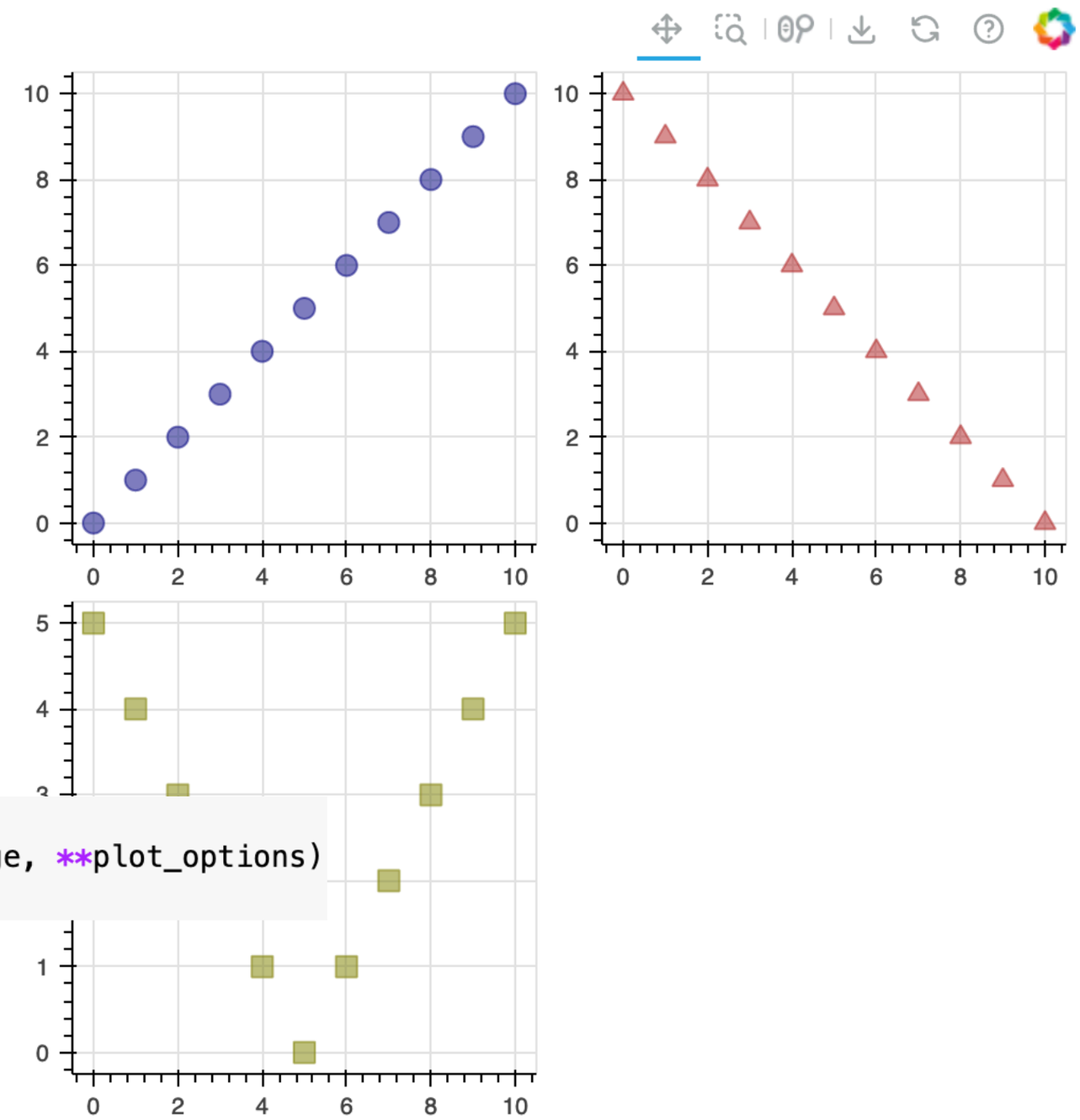
```
16 # put all the plots in a gridplot
17 p = gridplot([[s1, s2], [s3, None]])
```

- Each subplot can be generated and configured separately

- Plots can be connected by passing references to other objects' settings:

```
10 # create a new plot and share both ranges
11 s2 = figure(x_range=s1.x_range, y_range=s1.y_range, **plot_options)
12 s2.triangle(x, y1, size=10, color="firebrick")
```

- In this way, any change to one of the plots quickly reflects in all others



Linking plots

- Multiple figure objects can be put together using the gridplot() command

- gridplot() takes a 2D array of image objects as the main argument

```
16 # put all the plots in a gridplot
17 p = gridplot([[s1, s2], [s3, None]])
```

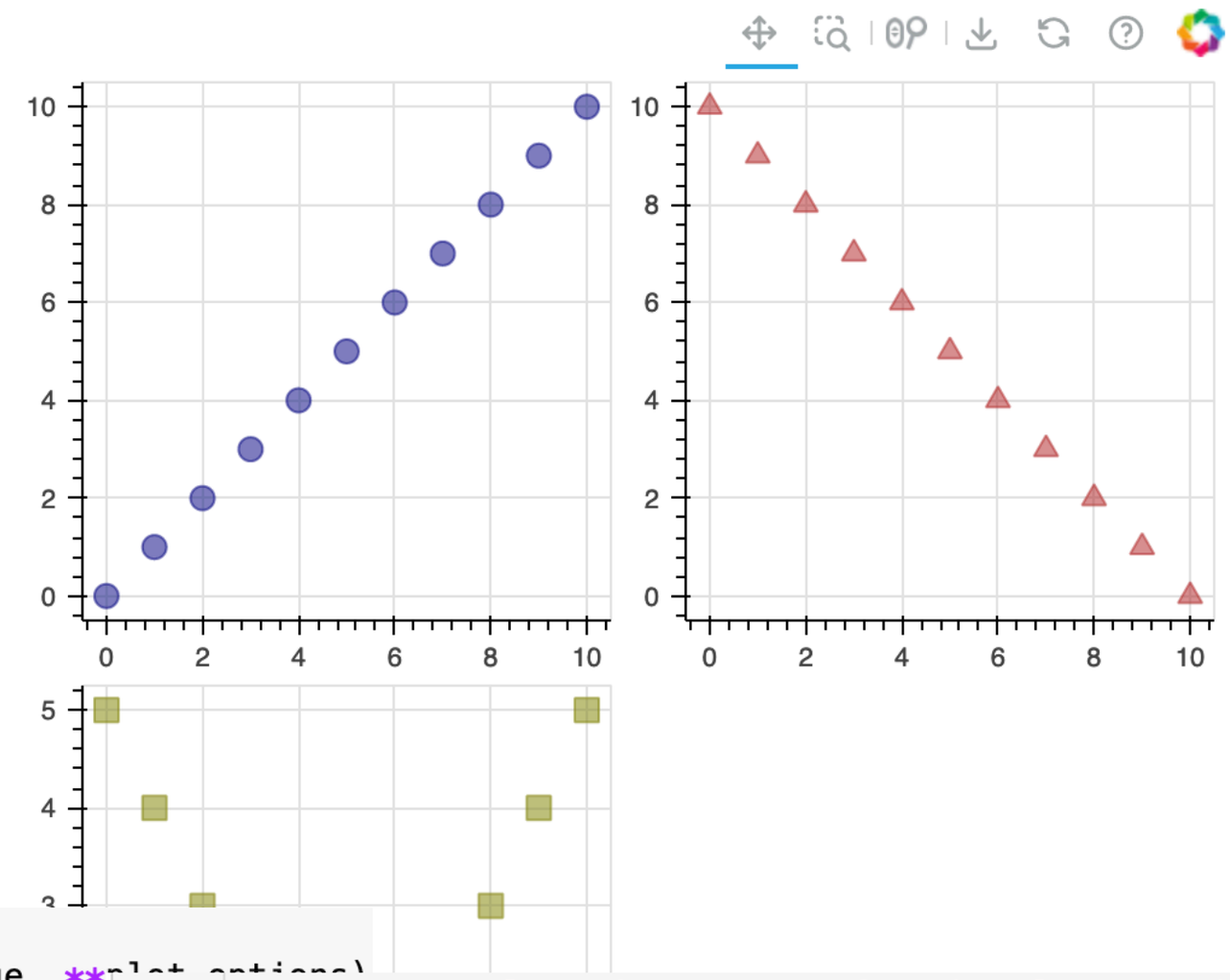
- Each subplot can be generated and configured separately

- Plots can be connected by passing references to other objects' settings:

```
10 # create a new plot and share both ranges
11 s2 = figure(x_range=s1.x_range, y_range=s1.y_range, **plot_options)
12 s2.triangle(x, y1, size=10, color="firebrick")
```

- In this way, any change to one of the plots quickly reflects in all others

- Or by sharing a common data source




```
4  # create a column data source for the plots to share
5  source = ColumnDataSource(data=dict(x=x, y0=y0, y1=y1))
6
7  TOOLS = "box_select,lasso_select,reset,help"
8
9  # create a new plot and add a renderer
10 left = figure(tools=TOOLS, width=300, height=300)
11 left.circle('x', 'y0', source=source)
12
13 # create another new plot and add a renderer
14 right = figure(tools=TOOLS, width=300, height=300)
15 right.circle('x', 'y1', source=source)
```

Plot tools

https://docs.bokeh.org/en/latest/docs/user_guide/interaction/tools.html

- Bokeh plots come with a toolbar to help us interact with them
- Plot tools are of several types:
    - Pan/Drag tools
    - Click/Tap tools
    - Scroll/Pinch tools
    - Action tools
    - Inspector tools

Plot tools

https://docs.bokeh.org/en/latest/docs/user_guide/interaction/tools.html

- Bokeh plots come with a toolbar to help us interact with them
- Plot tools are of three types:
  - Pan/Drag tools
    - BoxSelectTool - Select all points within a box
    - BoxZoomTool - Zoom into a box
    - LassoSelectTool - Select all points within an arbitrary area
    - PanTool - Pan the plot
  - Click/Tap tools
  - Scroll/Pinch tools
  - Action tools
  - Inspector tools

Plot tools

https://docs.bokeh.org/en/latest/docs/user_guide/interaction/tools.html

- Bokeh plots come with a toolbar to help us interact with them
- Plot tools are of three types:
  - Pan/Drag tools
  - Click/Tap tools
    - PolySelectTool - Select all points within an arbitrary polygon
    - TapTool - Select individual data points
  - Scroll/Pinch tools
    - WheelZoomTool - Zoom the plot in and out
    - WheelPanTool - Pans the plot along a specified direction
  - Action tools
  - Inspector tools

Plot tools

https://docs.bokeh.org/en/latest/docs/user_guide/interaction/tools.html

- Bokeh plots come with a toolbar to help us interact with them
- Plot tools are of three types:
  - Pan/Drag tools
  - Click/Tap tools
  - Scroll/Pinch tools
  - Action tools
    - UndoTool - Restores the previous state of the plot
    - RedoTool - Reverses the last action performed by the undo tool.
    - ResetTool - Restores the plot ranges to their original values
    - SaveTool - Save a PNG image of the plot.
  - Inspector tools
    - CrosshairTool - Draws a crosshair annotation over the plot
    - HoverTool - Displays a customizable tool tip

Plot tools

https://docs.bokeh.org/en/latest/docs/user_guide/interaction/tools.html

- You can manually configure which tools you want to have enabled in your plot by using the tools argument of the figure object.

```
p = figure(height=350, title="Pie Chart", toolbar_location=None,
           tools="hover", tooltips="@country: @value")
```

- You can also set the toolbar_location argument to be None, to disable it or “above”, “below”, “left”, or "right" to specify which side of the plot to display it in.
- The hover tool also requires you to specify the format of the tooltip displayed. This can be specified by the tooltips argument as a string and it can refer to values in the data source by specifying the column name preceded by a @

Networks

https://docs.bokeh.org/en/latest/docs/examples/topics/graph/from_networkx.html

- Bokeh has good support for visualizing networks described in NetworkX
- To generate a bokeh graph from a NetworkX object, use from_networkx() with the NetworkX object and the layout algorithm you want to use
- The resulting object must then be appended to the list of renderers of our plot

```
# Create a Bokeh graph from the NetworkX input using nx.spring_layout
graph = from_networkx(G, nx.spring_layout, scale=1.8, center=(0,0))
plot.renderers.append(graph)
```

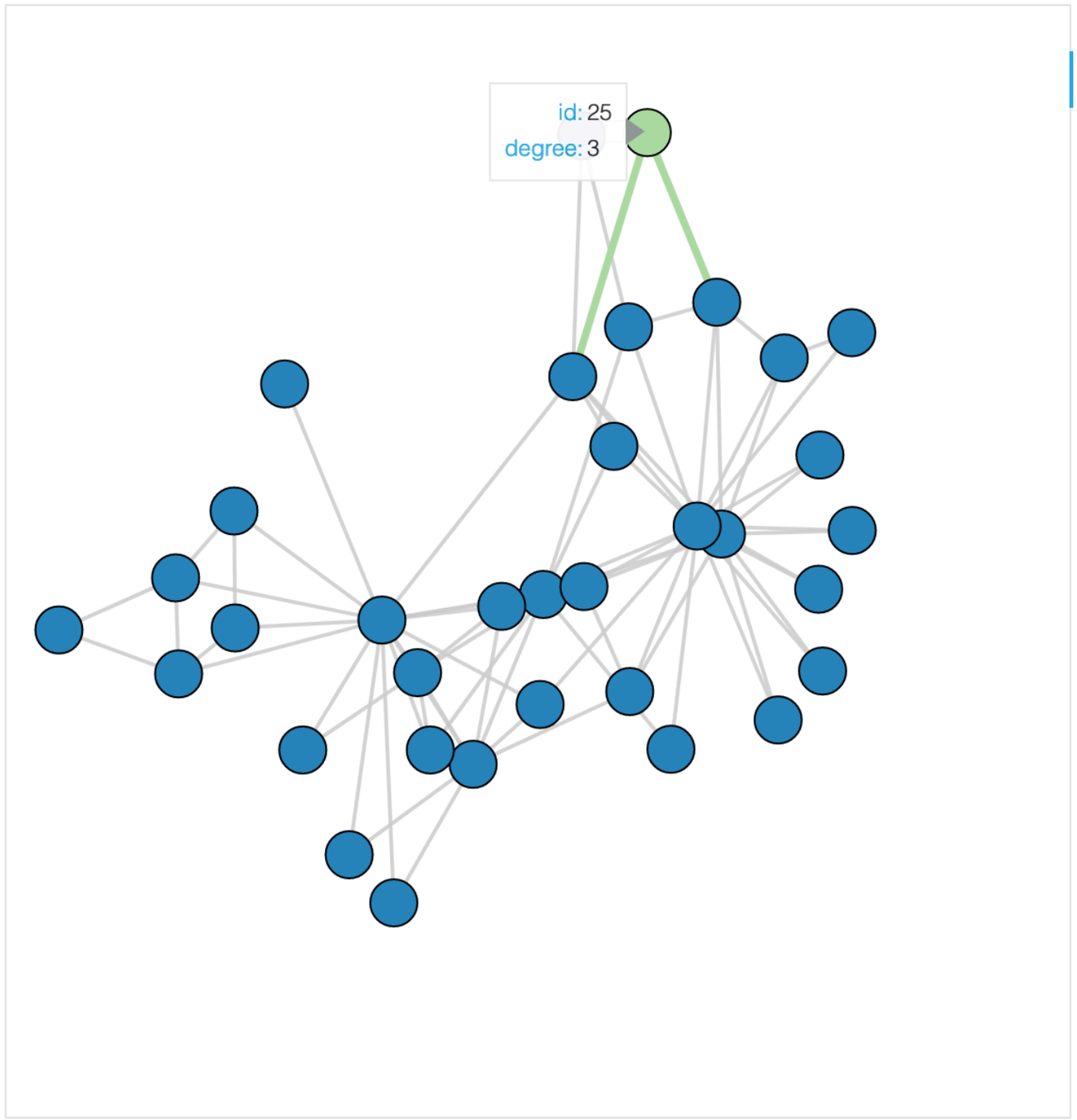
- We can also specify display settings for nodes and edges

```
# Blue circles for nodes, and light grey lines for edges
graph.node_renderer.glyph = Circle(size=25, fill_color='#2b83ba')
graph.edge_renderer.glyph = MultiLine(line_color="#cccccc",
                                      line_alpha=0.8, line_width=2)
```

- Specify tooltips for nodes and edges, etc

Networks

https://docs.bokeh.org/en/latest/docs/examples/topics/graph/from_networkx.html



Code - Bokeh

https://github.com/DataForScience/InteractiveViz

5. Plotly

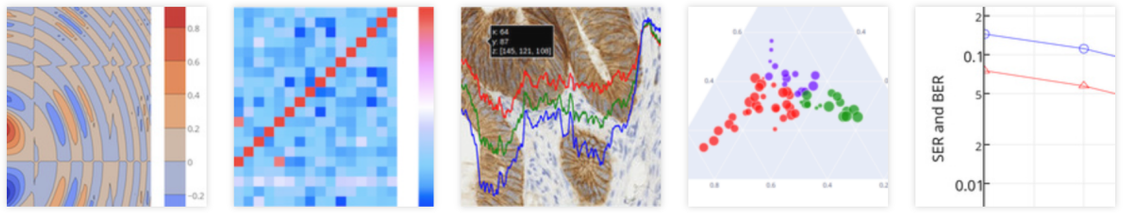
plotly

plotly.com

- Founded in 2012, in Montreal, Canada
- plotly.py - interactive, open-source library for creating graphs powered by the plotly.js JavaScript library
- Supports a huge range of interactive graphs across a large number of applications and fields
- Integrated into Dash, a framework for building web-based analytic applications.

Scientific Charts

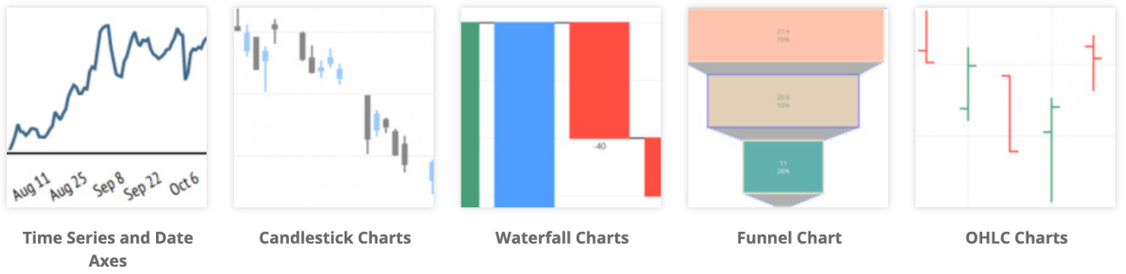
More Scientific Charts »



Contour Plots | Heatmaps | Imshow | Ternary Plots | Log Plots

Financial Charts

More Financial Charts »



Time Series and Date Axes | Candlestick Charts | Waterfall Charts | Funnel Chart | OHLC Charts

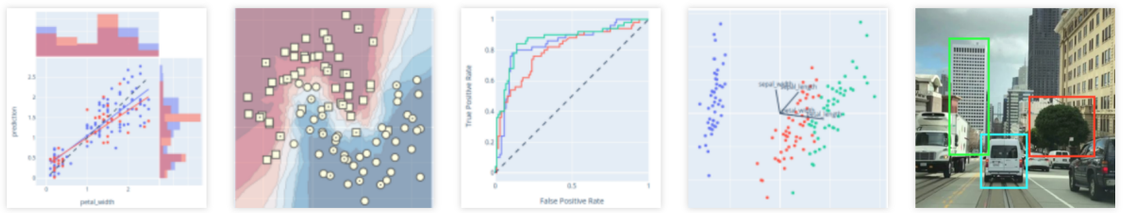
Maps

More Maps »

Mapbox Choropleth Maps | Lines on Mapbox | Filled Area on Maps | Bubble Maps | Mapbox Density Heatmap

Artificial Intelligence and Machine Learning

More AI and ML »



ML Regression | kNN Classification | ROC and PR Curves | PCA Visualization | AI/ML Apps with Dash

- plotly relies on three main modules:
  - plotly.express - to generate entire figures at once
  - plotly.graph_objects - to generate individual graph components
  - plotly.figure_factory - to create multi-plot figures
- plotly.express supports a wide range of figures, each with its own dedicated method
  - Basics: scatter(), line(), area(), bar(), funnel(), timeline()
  - Part-of-Whole: pie(), sunburst(), treemap(), icicle(), funnel_area()
  - 1D Distributions: histogram(), box(), violin(), strip(), ecdf()
  - 2D Distributions: density_heatmap(), density_contour()
  - Matrix or Image Input: imshow()
  - 3-Dimensional: scatter_3d(), line_3d()
  - Multidimensional: scatter_matrix(), parallel_coordinates(), parallel_categories()
  - Tile Maps: scatter_mapbox(), line_mapbox(), choropleth_mapbox(), density_mapbox()
  - Polar Charts: scatter_polar(), line_polar(), bar_polar()

plotly express

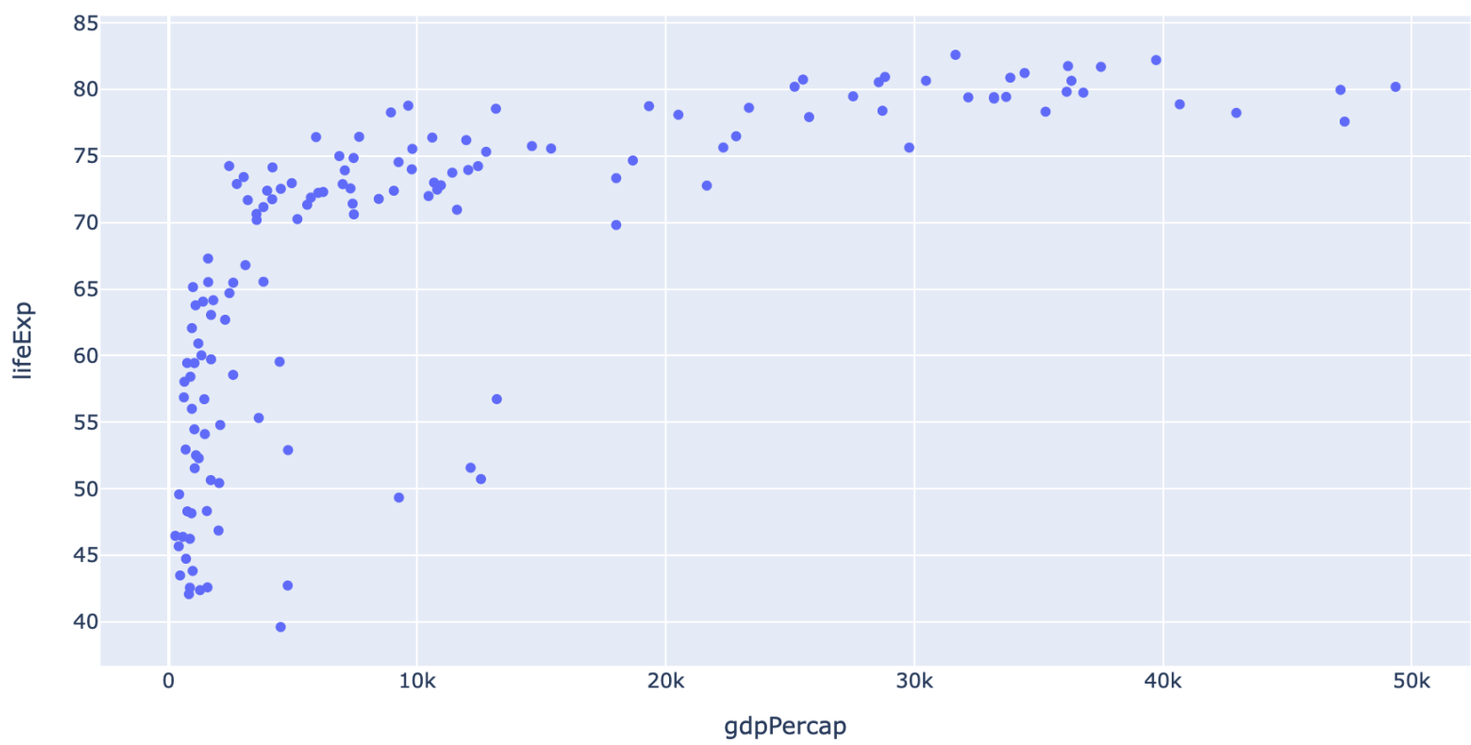
- Each function takes a pandas DataFrame as an argument, and you can refer directly to column names for each of the parameters

- Figures don't display until the show() method is called on the figure object

- A simple plot can be generated using:

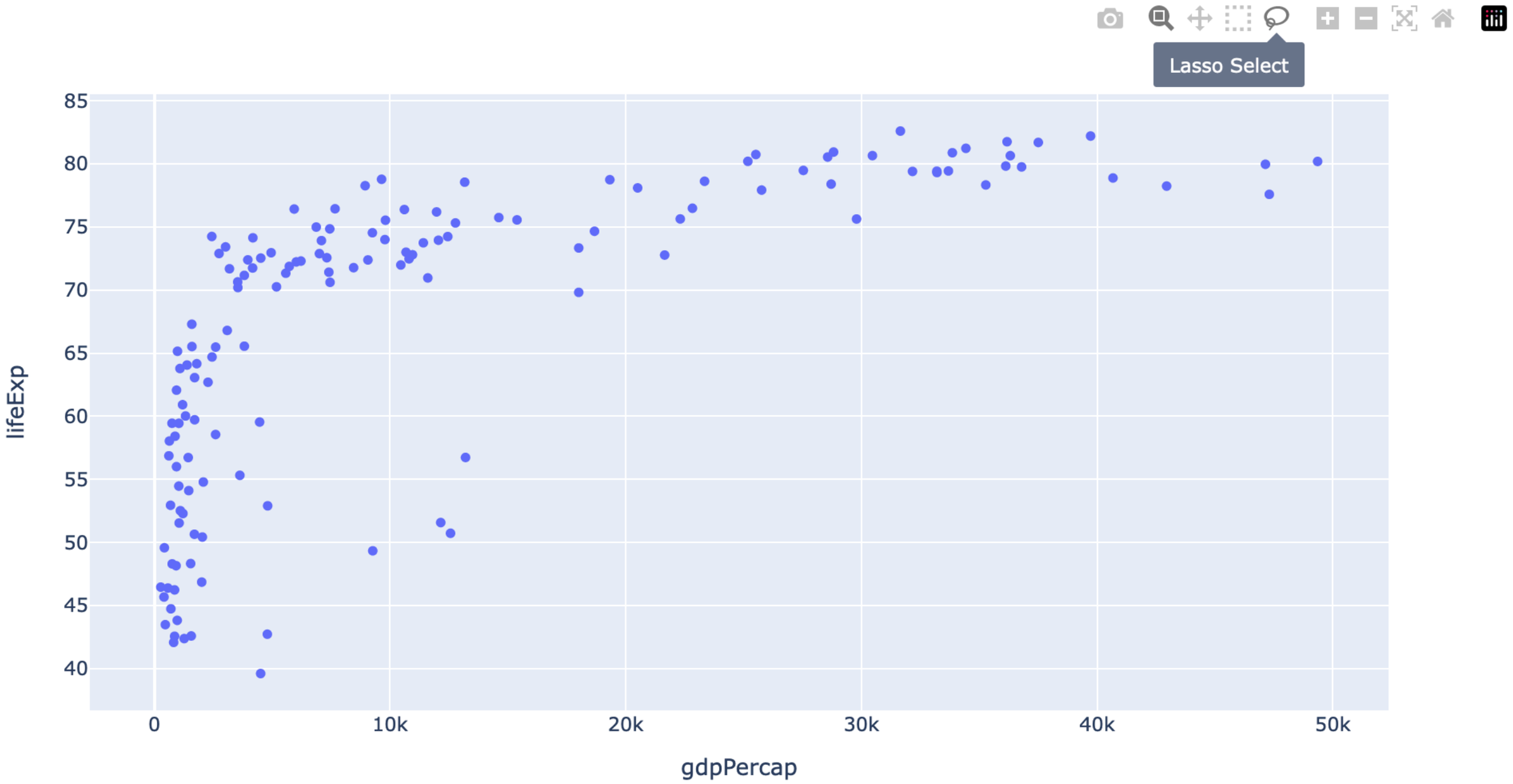
```
fig = px.scatter(gapminder, x="gdpPercap", y="lifeExp")
fig.show()
```

- Resulting in



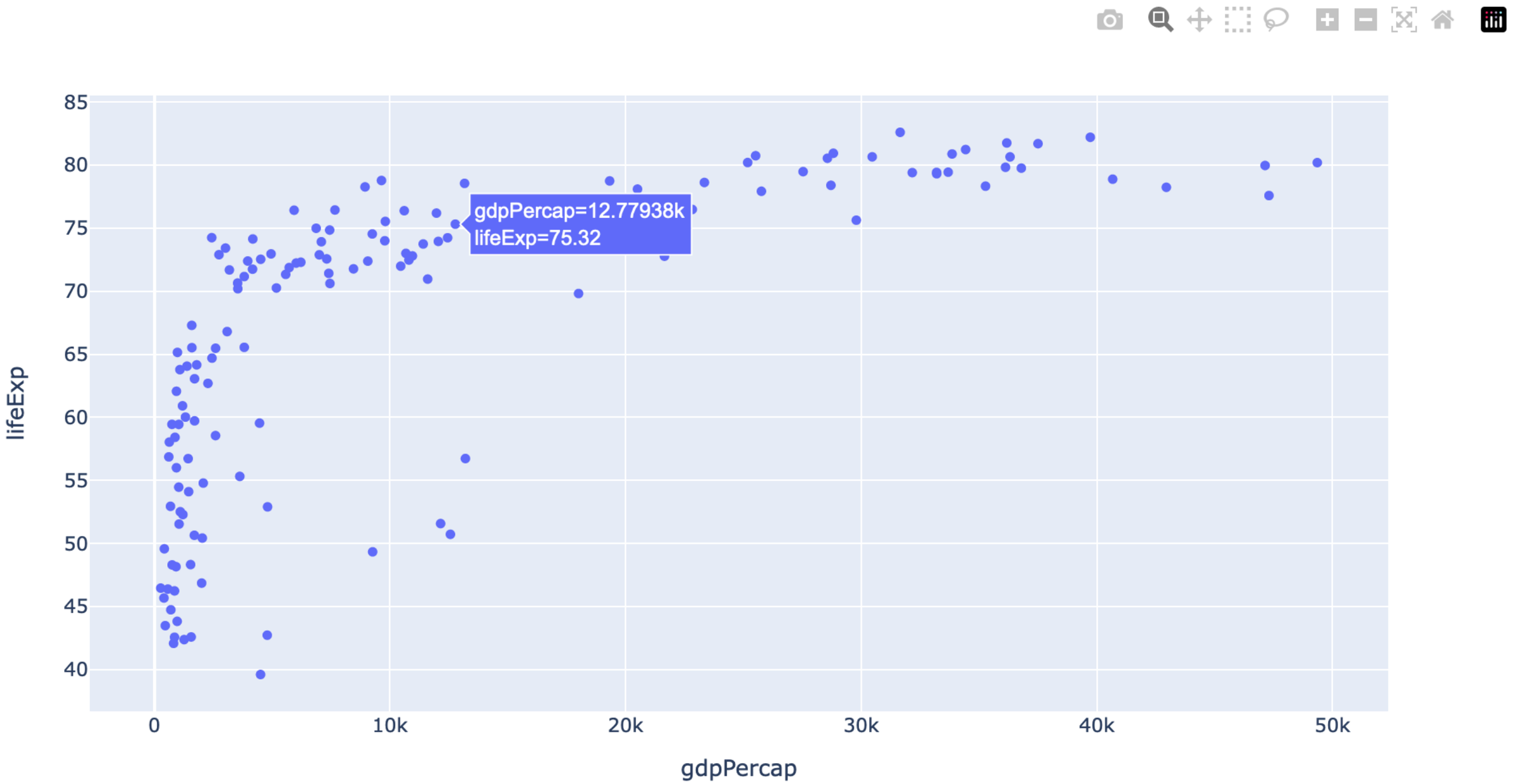
plotly express

- plotly.express plots also include a toolbar with a set of tools similar to what we saw in Bokeh



plotly express

- plotly.express plots also include tooltips with the values for each point in the chart



plotly express

- Each plotting function has a wide range of options that we can use to customize the resulting plot

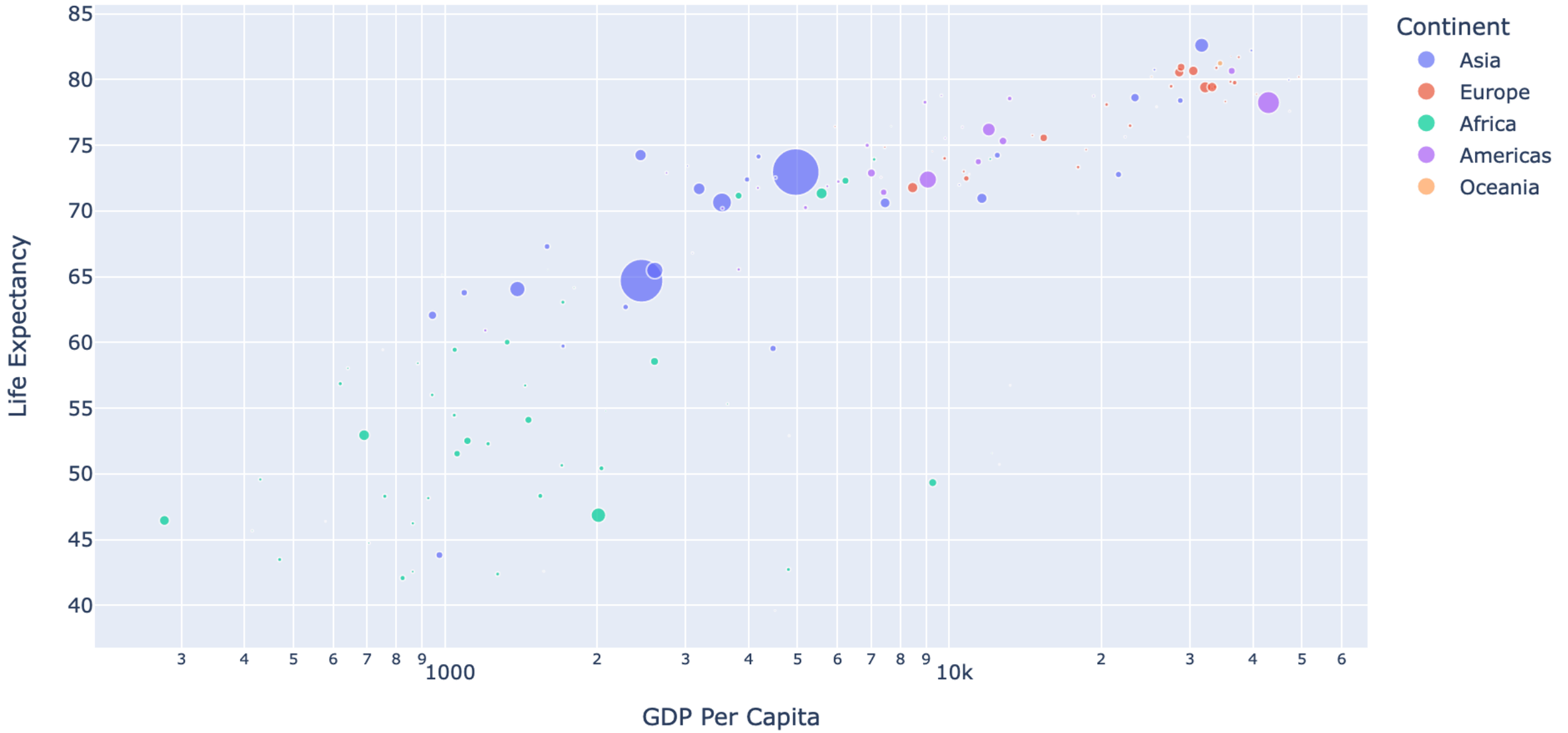
- The labels parameter specifies how to rename the names of the columns in our DataFrame when displaying the respective values, such as in axis labels, legends, tooltips, etc.

```
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True)
fig.show()
```

- Resulting in a powerful and flexible interface

plotly express

```python
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True)
fig.show()
```



plotly express

- plotly.express is optimized to generate pre-formatted plotly plots with minimal lines of code; (similar to what Seaborn does for matplotlib).

- To generate more complex plots, we must use the plotly.graph_objects interface, which allows us to overlay multiple plotly.express plots into single figure

- To overlay two or more plotly.express, we must instantiate a go.Figure object where the data parameter is set to the sum of the data field of the plots we wish to combine

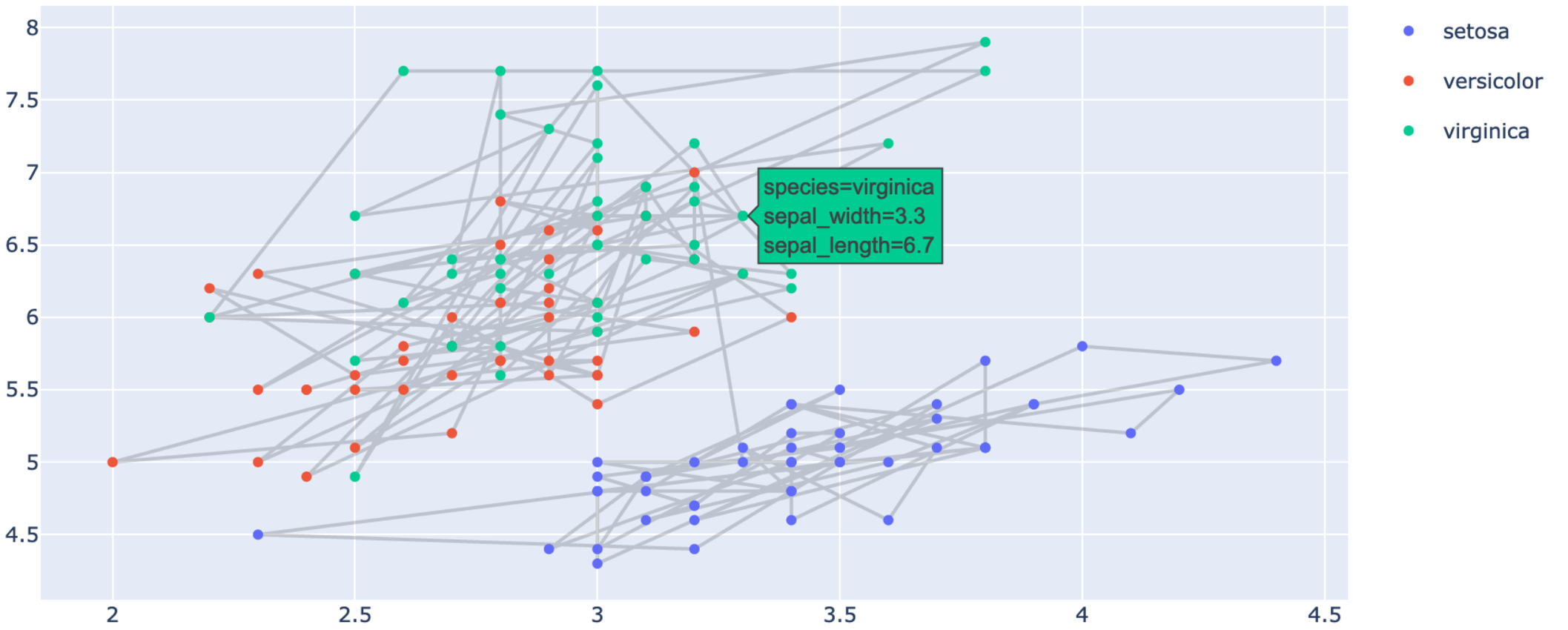
- For example, we can generate a simple scatter plot with lines connecting all the dots, by using:

```
fig1 = px.line(iris, x="sepal_width", y="sepal_length")
fig1.update_traces(line=dict(color = 'rgba(50,50,50,0.2)'))

fig2 = px.scatter(iris, x="sepal_width", y="sepal_length", color="species")

fig3 = go.Figure(data=fig1.data + fig2.data)
fig3.show()
```

plotly express

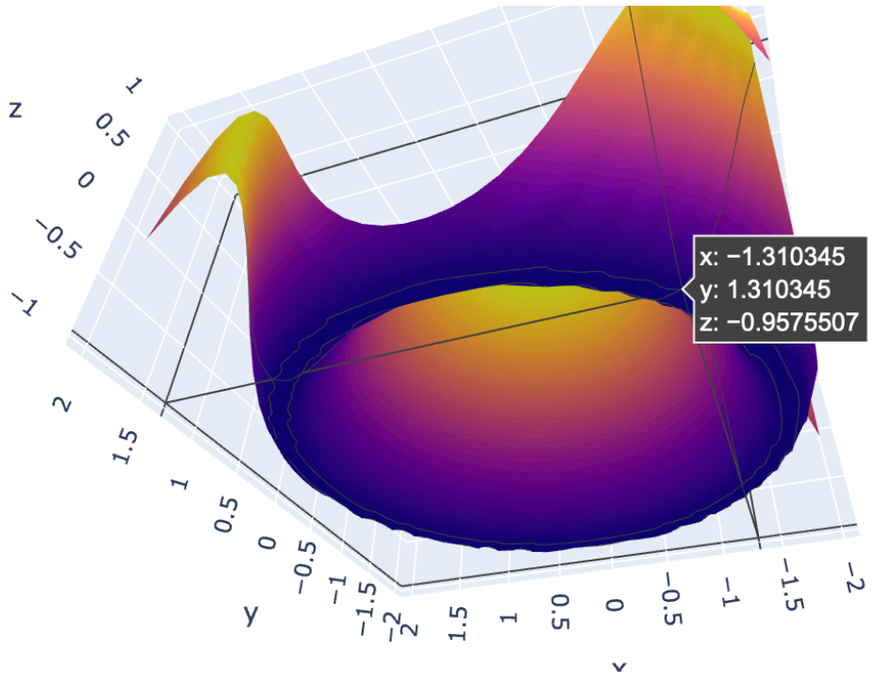
```
1 fig1 = px.line(iris, x="sepal_width", y="sepal_length")
2 fig1.update_traces(line=dict(color = 'rgba(50,50,50,0.2)'))
3
4 fig2 = px.scatter(iris, x="sepal_width", y="sepal_length", color="species")
5
6 fig3 = go.Figure(data=fig1.data + fig2.data)
7 fig3.show()
```

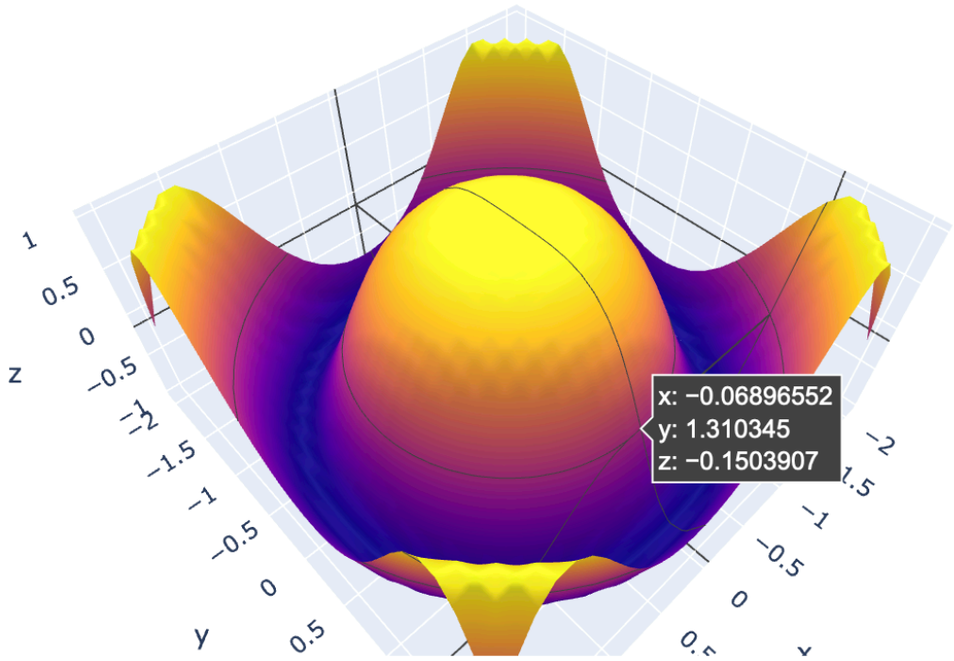
- Resulting in:



3D plotting

- plotly has extensive support for 3D plots using:
  - px.line_3d() - 3D lines
  - px.scatter_3d() - 3D scatter plot
  - go.Surface() - 3D Surface plot
- In the case of 3D plots, we can easily rotate the image in 3D by simply dragging the mouse





Animated Plots

https://plotly.com/python/animations/

- Support for animations is straightforward.
- We can animate scatter and bar plotly.express plots by providing two extra arguments:
  - animation_frame - the name of the column in the data frame that splits the data into individual frames
  - animation_group - the name of the column specifying the group of values that will be animated
- When these arguments are provided, Play, Stop and Timeline elements are added to the bottom of the figure that allows us to animate the plot

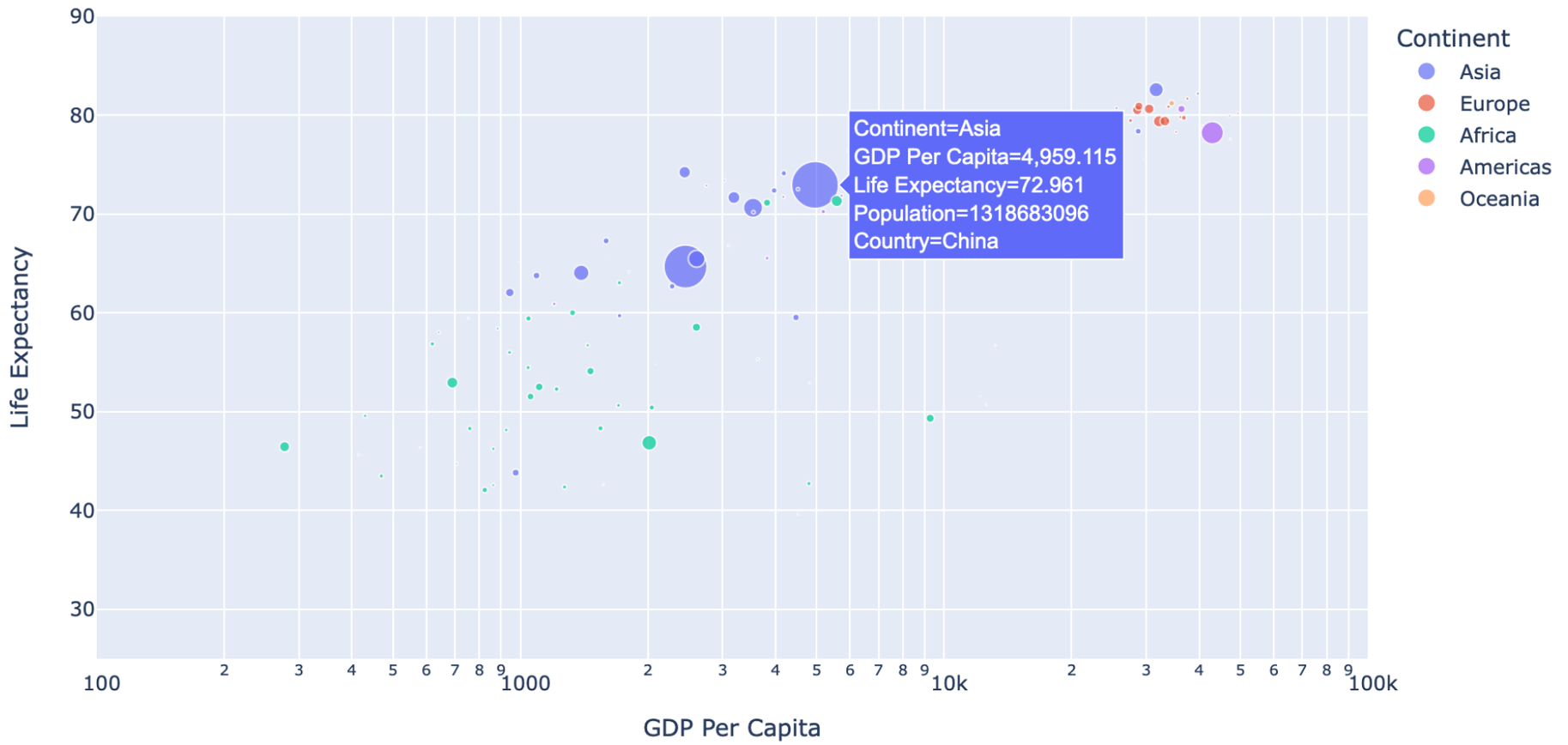
Animated Plots

https://plotly.com/python/animations/

- A gap minder plot for single year can be generated using:

```
fig = px.scatter(gapminder_2007,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True,
                 range_x=[100,100000], range_y=[25,90])
fig.show()
```

- Resulting in:



Animated Plots

https://plotly.com/python/animations/

- To animate it, we just need a data frame with multiple years, and properly setting the animation_frame and animation_group parameters

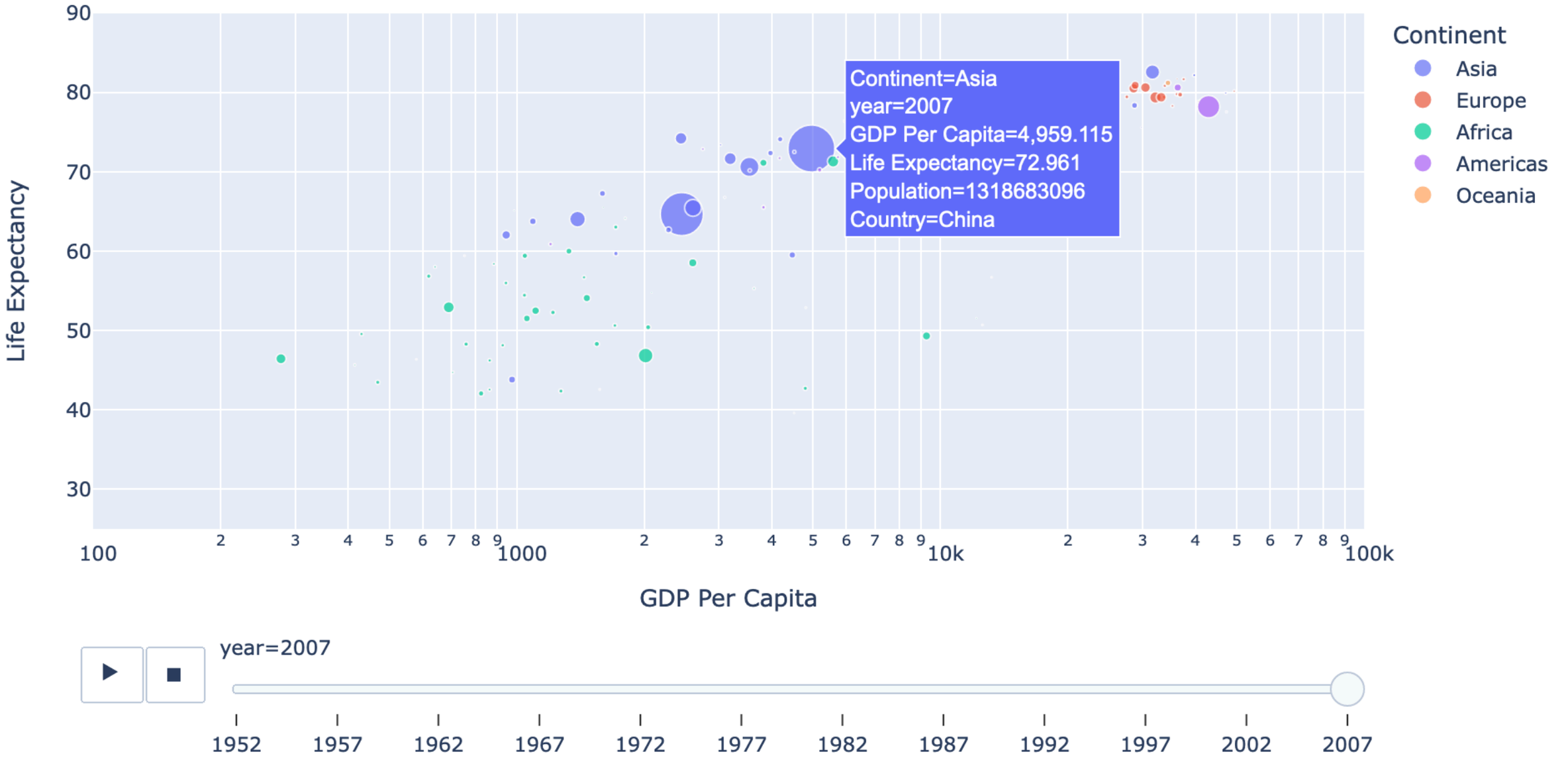
```python
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 animation_frame="year", animation_group="country",
                 log_x=True,
                 range_x=[100,100000], range_y=[25,90]
                )
fig.show()
```

- Which adds the playback controls that allow us to playback the animation

Animated Plots

https://plotly.com/python/animations/

- Resulting in:



Unfortunately, plotly does not support exporting the animation as a video file

Code - Plotly

https://github.com/DataForScience/InteractiveViz

Question

- How was the technical level?
    - 1 — Too Low (too many details)
    - 2 — Low
    - 3 — Just Right
    - 4 — High
    - 5 — Too High (not enough details)

Question

- How was the level of Python code/explanations?
    - 1 — Too Low (too many details)
    - 2 — Low
    - 3 — Just Right
    - 4 — High
    - 5 — Too High (not enough details)

Events

graphs4sci.substack.com

Time Series for Everyone

Apr 11, 2023 - 10am-2pm (PST)

Probability and Statistics for Everyone

Apr 25, 2023 - 10am-2pm (PST

Graphs for Data Science

May 10, 2023 - 10am-2pm (PST)

graphs4sci.substack.com

VIDEO

Times Series Analysis for Everyone

★★★★★ 1 review

By Bruno Goncalves

https://bit.ly/Timeseries_LL

TIME TO COMPLETE:
6h

TOPICS:
Time Series

PUBLISHED BY:
Pearson

PUBLICATION DATE:
November 2021

Start

6 Hours of Video Instruction

The perfect introduction to time-based analytics

Overview

Times Series Analysis for Everyone LiveLessons covers the fundamental tools and techniques for the analysis of time series data. These lessons introduce you to the basic concepts, ideas, and algorithms necessary to develop your own time series applications in a step-by-step, intuitive fashion. The lessons follow a gradual progression, from the more specific to the more abstract, taking you from the very basics to some of the most recent and sophisticated algorithms by leveraging the statsmodels, arch, and Keras state-of-the-art models.

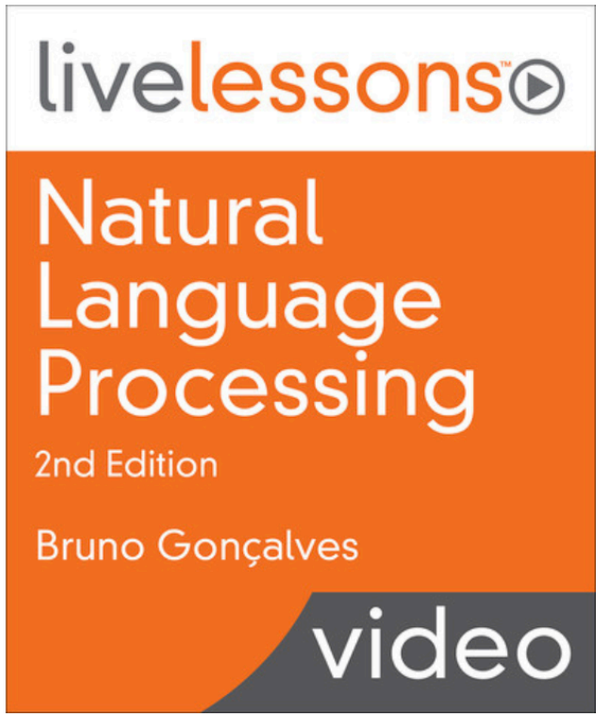
VIDEO

Natural Language Processing, 2nd Edition

Write the first review

By Bruno Goncalves

https://bit.ly/NLP_LL



TIME TO COMPLETE:
5h 23m

TOPICS:
Natural Language Processing

PUBLISHED BY:
Addison-Wesley Professional

PUBLICATION DATE:
October 2021

Start

5 Hours of Video Instruction

Overview

Natural Language Processing LiveLessons covers thefundamentals of Natural Language Processing in a simple and intuitive way,empowering you to add NLP to your toolkit. Using the powerful NLTK package, itgradually moves from the basics of text representation, cleaning, topicdetection, regular expressions, and sentiment analysis before moving on to theKeras deep learning framework to explore more advanced topics such as textclassification and sequence-to-sequence models. After successfully completingthese lessons you'll be equipped with a fundamental and practical understandingof state-of-the-art Natural Language Processing tools and algorithms.

END