Chapter 1)

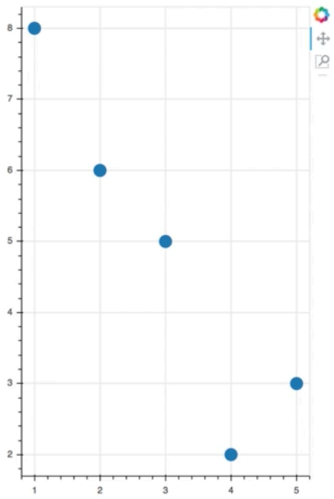
Note)

What are Glyphs

- Visual shapes
  - circles, squares, triangles
  - rectangles, lines, wedges
- With properties attached to data
  - coordinates (x,y)
  - size, color, transparency

Typical usage

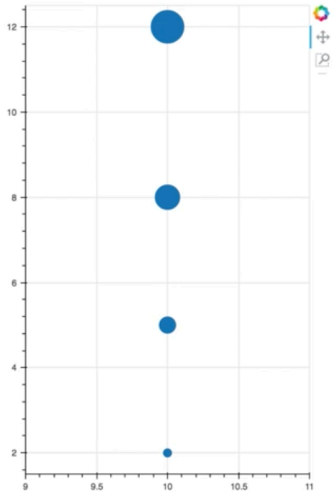
```
In [1]: from bokeh.io import output_file, show

In [2]: from bokeh.plotting import figure

In [3]: plot = figure(plot_width=400, tools='pan,box_zoom')

In [4]: plot.circle([1,2,3,4,5], [8,6,5,2,3])

In [5]: output_file('circle.html')

In [6]: show(plot)
```



Glyph properties

- Lists, arrays, sequences of values
- Single fixed values

```
In [1]: plot = figure()

In [2]: plot.circle(x=10, y=[2,5,8,12], size=[10,20,30,40])
```



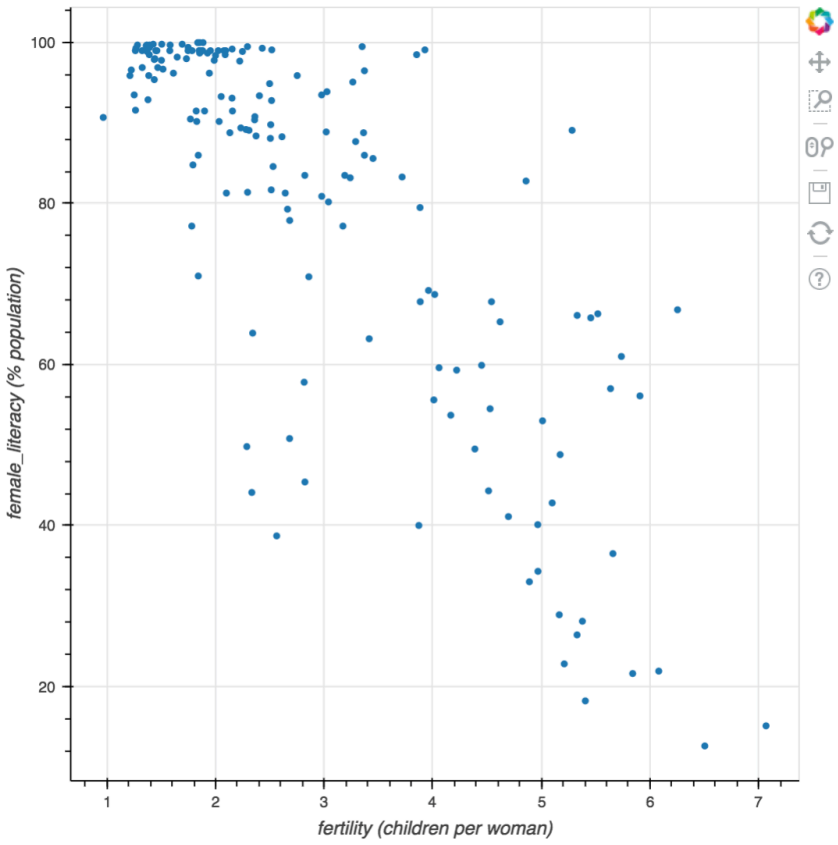
1)
A simple scatter plot

In this example, you're going to make a scatter plot of female literacy vs fertility using data from the [European Environmental Agency](). This dataset highlights that countries with low female literacy have high birthrates. The x-axis data has been loaded for you as `fertility` and the y-axis data has been loaded as `female_literacy`.
Your job is to create a figure, assign x-axis and y-axis labels, and plot `female_literacy` vs `fertility` using the circle glyph.
After you have created the figure, in this exercise and the ones to follow, play around with it! Explore the different options available to you on the tab to the right, such as "Pan", "Box Zoom", and "Wheel Zoom". You can click on the question mark sign for more details on any of these tools.
Note: You may have to scroll down to view the lower portion of the figure.

- Import the `figure` function from `bokeh.plotting`, and the `output_file`and `show` functions from `bokeh.io`.
- Create the figure `p` with `figure()`. It has two parameters: `x_axis_label`and `y_axis_label`.
- Add a circle glyph to the figure `p` using the function `p.circle()` where the inputs are, in order, the x-axis data and y-axis data.
- Use the `output_file()` function to specify the name `'fert_lit.html'` for the output file.
- Create and display the output file using `show()` and passing in the figure `p`.

```
# Import figure from bokeh.plotting
from bokeh.plotting import figure

# Import output_file and show from bokeh.io
from bokeh.io import output_file, show
```

```
# Create the figure: p
p = figure(x_axis_label='fertility (children per woman)',
y_axis_label='female_literacy (% population)')

# Add a circle glyph to the figure p
p.circle(x=fertility, y=female_literacy)

# Call the output_file() function and specify the name of the file
output_file('fert_lit.html')

# Display the plot
show(p)
```



2)

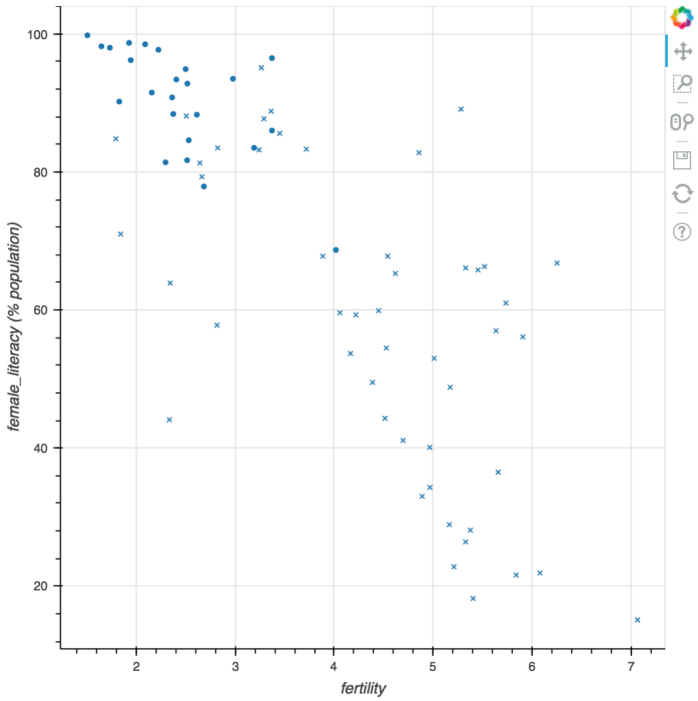
A scatter plot with different shapes

By calling multiple glyph functions on the same figure object, we can overlay multiple data sets in the same figure.
In this exercise, you will plot female literacy vs fertility for two different regions, Africa and Latin America. Each set of x and y data has been loaded separately for

you as `fertility_africa`, `female_literacy_africa`, `fertility_latinamerica`, and `female_literacy_latinamerica`.
Your job is to plot the Latin America data with the `circle()` glyph, and the Africa data with the `x()` glyph.
`figure` has already been imported for you from `bokeh.plotting`.

- Create the figure `p` with the `figure()` function. It has two parameters: `x_axis_label` and `y_axis_label`.
- Add a circle glyph to the figure `p` using the function `p.circle()` where the inputs are the x and y data from Latin America: `fertility_latinamerica` and `female_literacy_latinamerica`.
- Add an x glyph to the figure `p` using the function `p.x()` where the inputs are the x and y data from Africa: `fertility_africa` and `female_literacy_africa`

```
# Create the figure: p
p = figure(x_axis_label='fertility', y_axis_label='female_literacy (% population)')

# Add a circle glyph to the figure p
p.circle(fertility_latinamerica,female_literacy_latinamerica)

# Add an x glyph to the figure p
p.x(fertility_africa,female_literacy_africa)

# Specify the name of the file
output_file('fert_lit_separate.html')

# Display the plot
show(p)
```



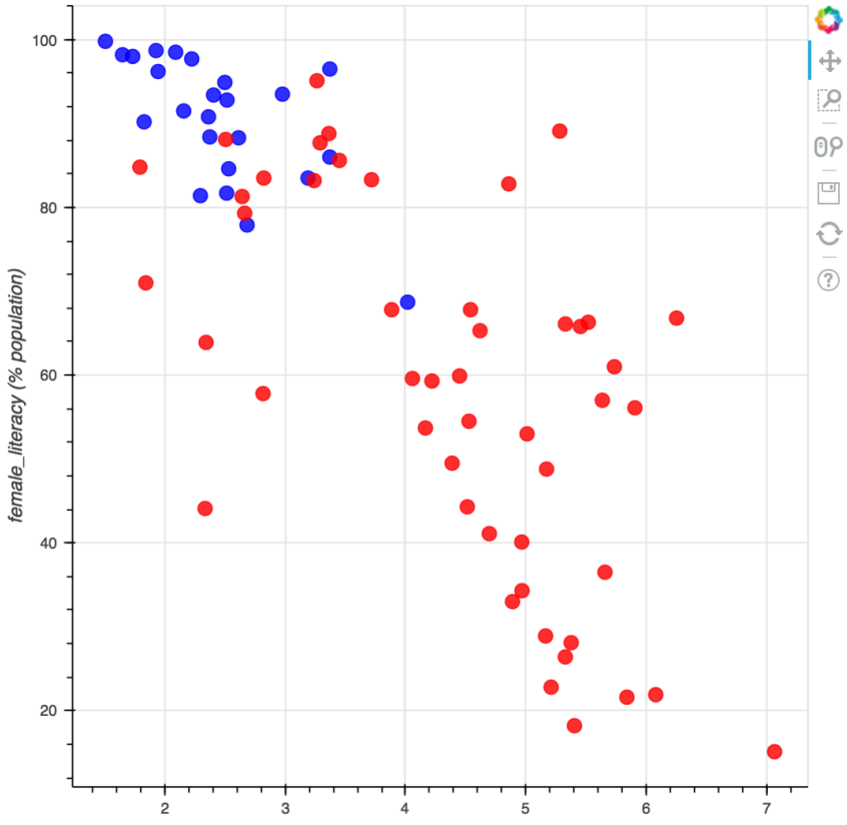
3)
Customizing your scatter plots

The three most important arguments to customize scatter glyphs are `color`, `size`, and `alpha`. Bokeh accepts colors as hexadecimal strings, tuples of RGB values between 0 and 255, and any of the 147 **CSS color names**. Size values are supplied in screen space units with 100 meaning the size of the entire figure. The `alpha` parameter controls transparency. It takes in floating point numbers between 0.0, meaning completely transparent, and 1.0, meaning completely opaque.
In this exercise, you'll plot female literacy vs fertility for Africa and Latin America as red and blue circle glyphs, respectively.

- Using the Latin America data (`fertility_latinamerica` and `female_literacy_latinamerica`), add a `blue` circle glyph of `size=10` and `alpha=0.8` to the figure `p`. To do this, you will need to specify the `color`, `size` and `alpha` keyword arguments inside `p.circle()`.
- Using the Africa data (`fertility_africa` and `female_literacy_africa`), add a `red` circle glyph of `size=10` and `alpha=0.8` to the figure `p`

```
# Create the figure: p
p = figure(x_axis_label='fertility (children per woman)',
y_axis_label='female_literacy (% population)')

# Add a blue circle glyph to the figure p
p.circle(fertility_latinamerica, female_literacy_latinamerica, size=10,
alpha=0.8, color='blue')
```

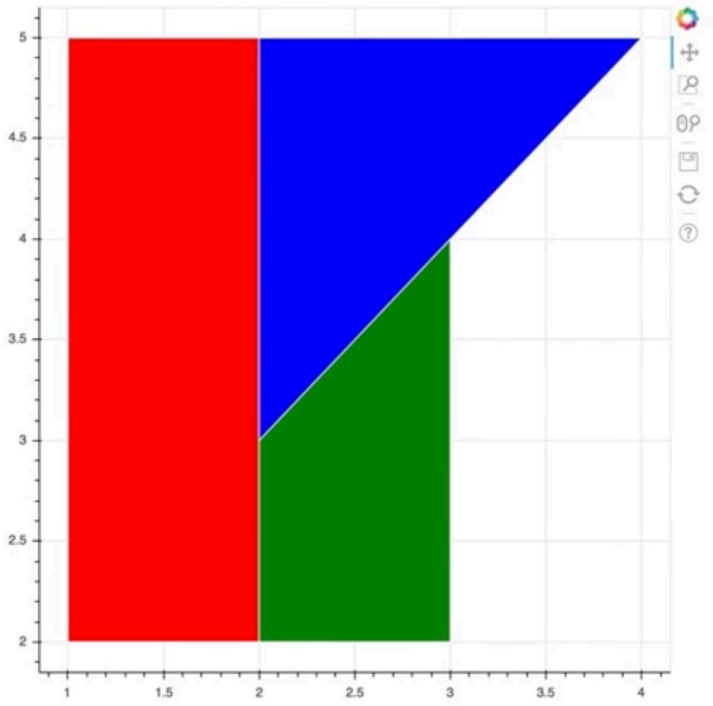
```
# Add a red circle glyph to the figure p
p.circle(fertility_africa,female_literacy_africa,size=10,alpha=0.8,color='red')

# Specify the name of the file
output_file('fert_lit_separate_colors.html')

# Display the plot
show(p)
```



Note)

Patches

```
In [1]: from bokeh.io import output_file, show

In [2]: from bokeh.plotting import figure

In [3]: xs = [ [1,1,2,2], [2,2,4], [2,2,3,3] ]

In [4]: ys = [ [2,5,5,2], [3,5,5], [2,3,4,2] ]

In [5]: plot = figure()

In [6]: plot.patches(xs, ys,
   ...:              fill_color=
   ...:              ['red', 'blue','green'],
   ...:              line_color='white')

In [7]: output_file('patches.html')

In [8]: show(plot)
```



4)

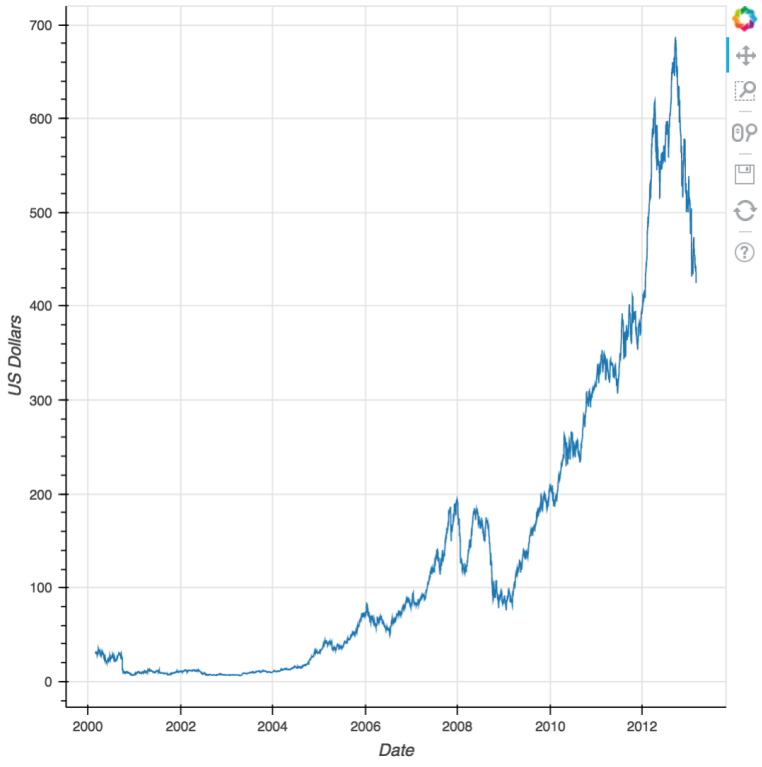
Lines

We can draw lines on Bokeh plots with the `line()` glyph function.
In this exercise, you'll plot the daily adjusted closing price of Apple Inc.'s stock (AAPL) from 2000 to 2013.

The data points are provided for you as lists. `date` is a list of **datetime objects**to plot on the x-axis and `price` is a list of prices to plot on the y-axis.
Since we are plotting dates on the x-axis, you must
add `x_axis_type='datetime'` when creating the figure object.

- Import the `figure` function from `bokeh.plotting`.
- Create a figure `p` using the `figure()` function with `x_axis_type` set to `'datetime'`. The other two parameters are `x_axis_label` and `y_axis_label`.
- Plot `date` and `price` along the x- and y-axes using `p.line()`

```
# Import figure from bokeh.plotting
from bokeh.plotting import figure

# Create a figure with x_axis_type="datetime": p
p = figure(x_axis_type='datetime', x_axis_label='Date', y_axis_label='US Dollars')

# Plot date along the x axis and price along the y axis
p.line(date,price)

# Specify the name of the output file and show the result
output_file('line.html')
show(p)
```

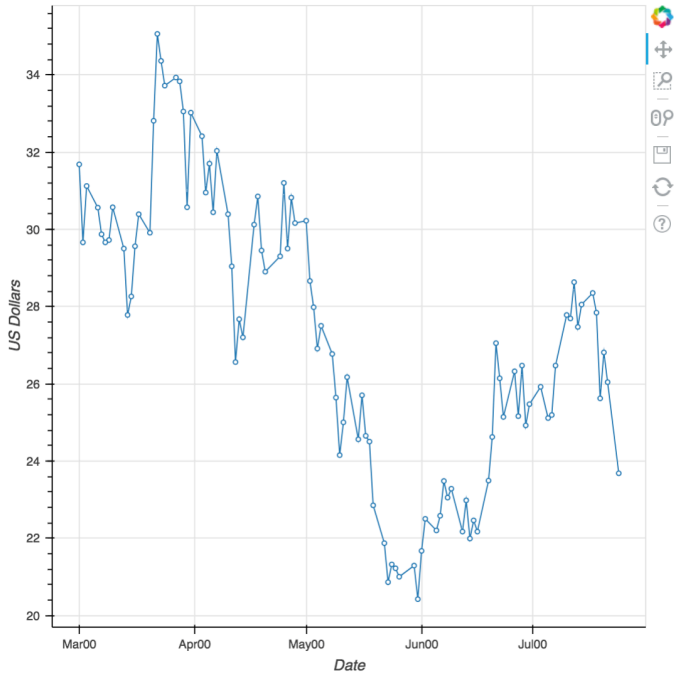


5)
Lines and markers

Lines and markers can be combined by plotting them separately using the same data points.
In this exercise, you'll plot a line and circle glyph for the AAPL stock prices. Further, you'll adjust the `fill_color` keyword argument of the `circle()`glyph function while leaving the `line_color` at the default value.
The `date` and `price` lists are provided. The Bokeh figure object `p` that you created in the previous exercise has also been provided.

- Plot `date` along the x-axis and `price` along the y-axis with `p.line()`.
- With `date` on the x-axis and `price` on the y-axis, use `p.circle()` to add a `'white'` circle glyph of size `4`. To do this, you will need to specify the `fill_color` and `size` arguments

```
# Import figure from bokeh.plotting
from bokeh.plotting import figure

# Create a figure with x_axis_type='datetime': p
p = figure(x_axis_type='datetime', x_axis_label='Date', y_axis_label='US Dollars')

# Plot date along the x-axis and price along the y-axis
p.line(date,price)

# With date on the x-axis and price on the y-axis, add a white circle glyph of size 4
p.circle(date, price, fill_color='white', size=4)

# Specify the name of the output file and show the result
output_file('line.html')
show(p)
```



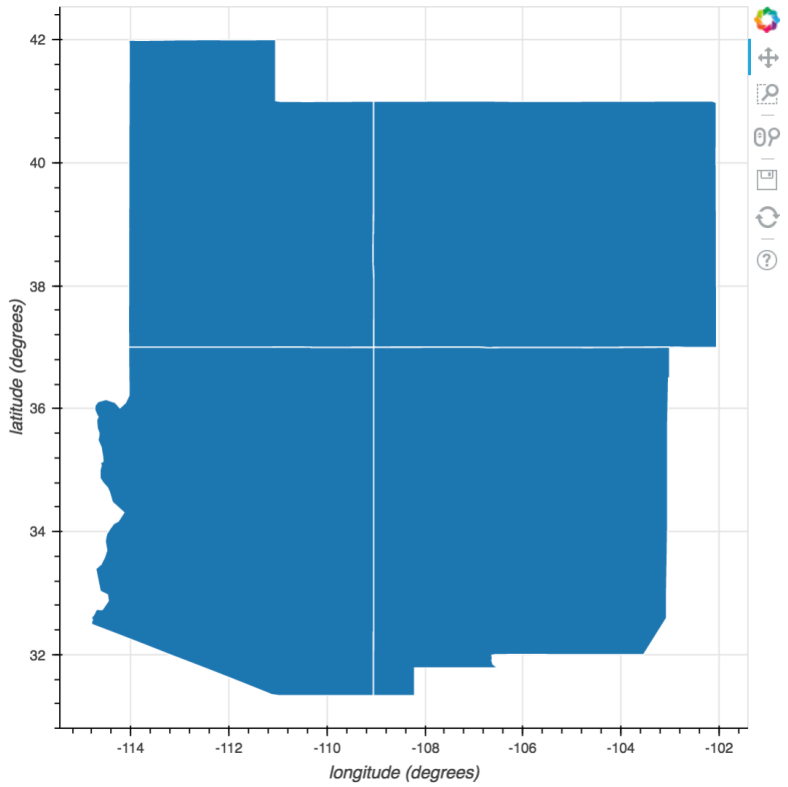
6)
Patches

In Bokeh, extended geometrical shapes can be plotted by using the `patches()`glyph function. The patches glyph takes as input a list-of-lists collection of numeric values specifying the vertices in x and y directions of each distinct patch to plot.
In this exercise, you will plot the state borders of Arizona, Colorado, New Mexico and Utah. The latitude and longitude vertices for each state have been prepared as lists.
Your job is to plot longitude on the x-axis and latitude on the y-axis. The figure object has been created for you as `p`.

- Create a list of the longitude positions for each state as `x`. This has already been done for you.
- Create a list of the latitude positions for each state as `y`. The variable names for the latitude positions are `az_lats`, `co_lats`, `nm_lats`, and `ut_lats`.
- Use `p.patches()` to add the patches glyph to the figure `p`. Supply the `x`and `y` lists as arguments along with a `line_color` of `'white'`

```
# Create a list of az_lons, co_lons, nm_lons and ut_lons: x
x = [az_lons, co_lons, nm_lons, ut_lons]

# Create a list of az_lats, co_lats, nm_lats and ut_lats: y
y = [az_lats,co_lats,nm_lats,ut_lats]

# Add patches to figure p with line_color=white for x and y
p.patches(x,y,line_color='white')
```

# Specify the name of the output file and show the result
output_file('four_corners.html')
show(p)

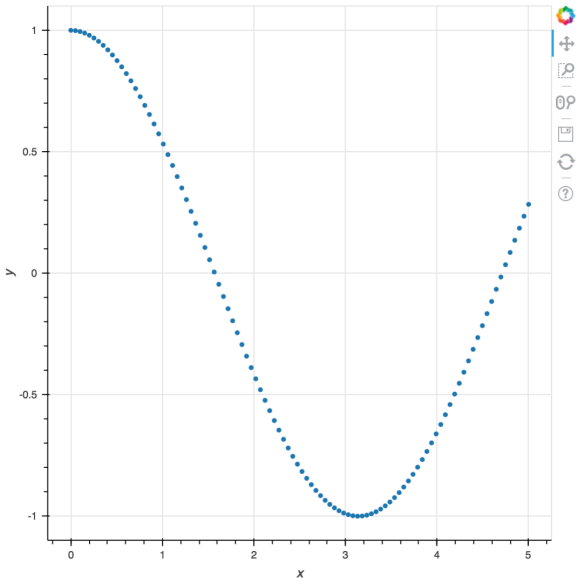


7)
Plotting data from NumPy arrays

In the previous exercises, you made plots using data stored in lists. You learned that Bokeh can plot both numbers and datetime objects.
In this exercise, you'll generate NumPy arrays using `np.linspace()` and `np.cos()` and plot them using the circle glyph.
`np.linspace()` is a function that returns an array of evenly spaced numbers over a specified interval. For example, `np.linspace(0, 10, 5)` returns an array of `5` evenly spaced samples calculated over the interval `[0, 10]`. `np.cos(x)` calculates the element-wise cosine of some array `x`.
For more information on NumPy functions, you can refer to the **NumPy User Guide** and **NumPy Reference**.
The figure `p` has been provided for you.

- Import `numpy` as `np`.
- Create an array `x` using `np.linspace()` with `0`, `5`, and `100` as inputs.
- Create an array `y` using `np.cos()` with `x` as input.
- Add circles at `x` and `y` using `p.circle()`.

```
# Import numpy as np
import numpy as np

# Create array using np.linspace: x
x = np.linspace(0,5,100)

# Create array using np.cos: y
y = np.cos(x)

# Add circles at x and y
p.circle(x,y)

# Specify the name of the output file and show the result
output_file('numpy.html')
show(p)
```



8)

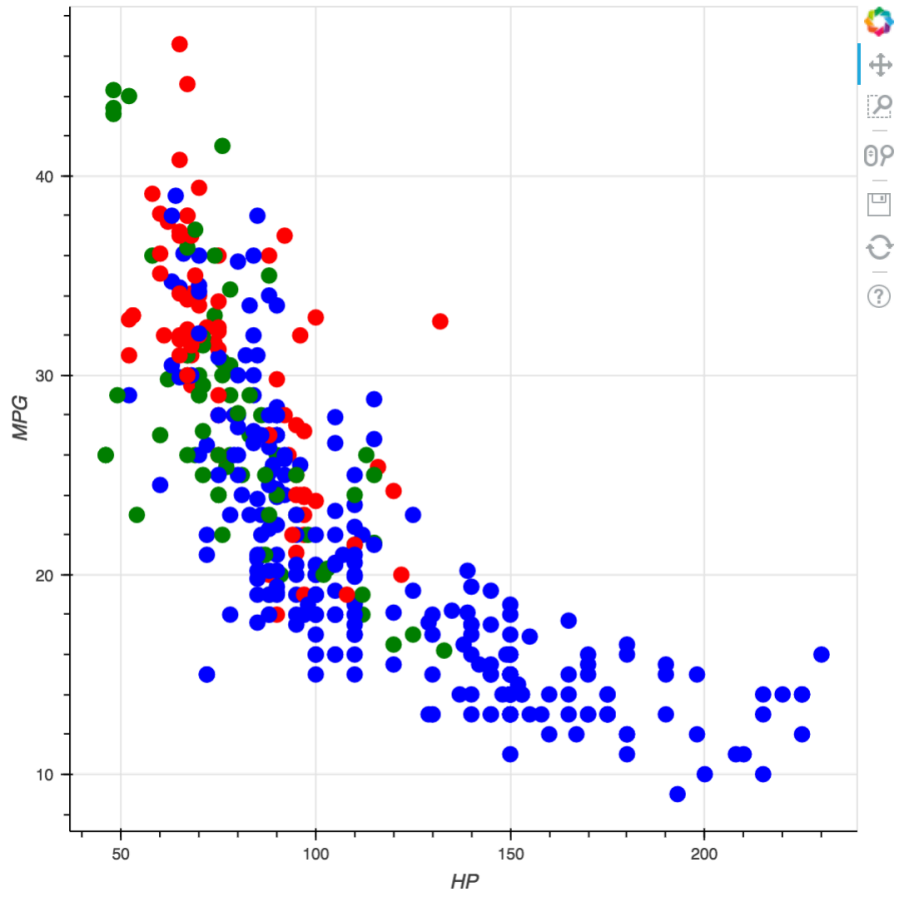
Plotting data from Pandas DataFrames

You can create Bokeh plots from Pandas DataFrames by passing column selections to the glyph functions.
Bokeh can plot floating point numbers, integers, and datetime data types. In this example, you will read a CSV file containing information on 392 automobiles manufactured in the US, Europe and Asia from 1970 to 1982.
The CSV file is provided for you as `'auto.csv'`.
Your job is to plot miles-per-gallon (`mpg`) vs horsepower (`hp`) by passing Pandas column selections into the `p.circle()` function. Additionally, each glyph will be colored according to values in the `color` column.

- Import `pandas` as `pd`.

- Use the `read_csv()` function of `pandas` to read in `'auto.csv'` and store it in the DataFrame `df`.
- Import `figure` from `bokeh.plotting`.
- Use the `figure()` function to create a figure `p` with the x-axis labeled `'HP'` and the y-axis labeled `'MPG'`.
- Plot `mpg` (on the y-axis) vs `hp` (on the x-axis) by `color` using `p.circle()`. Note that the x-axis should be specified before the y-axis inside `p.circle()`. You will need to use Pandas DataFrame indexing to pass in the columns. For example, to access the `color` column, you can use `df['color']`, and then pass it in as an argument to the `color` parameter of `p.circle()`. Also specify a `size` of `10`

```
# Import pandas as pd
import pandas as pd

# Read in the CSV file: df
df = pd.read_csv('auto.csv')

# Import figure from bokeh.plotting
from bokeh.plotting import figure

# Create the figure: p
p = figure(x_axis_label='HP', y_axis_label='MPG')

# Plot mpg vs hp by color
p.circle(df['hp'],df['mpg'],size=10,color=df['color'])

# Specify the name of the output file and show the result
output_file('auto-df.html')
show(p)
```



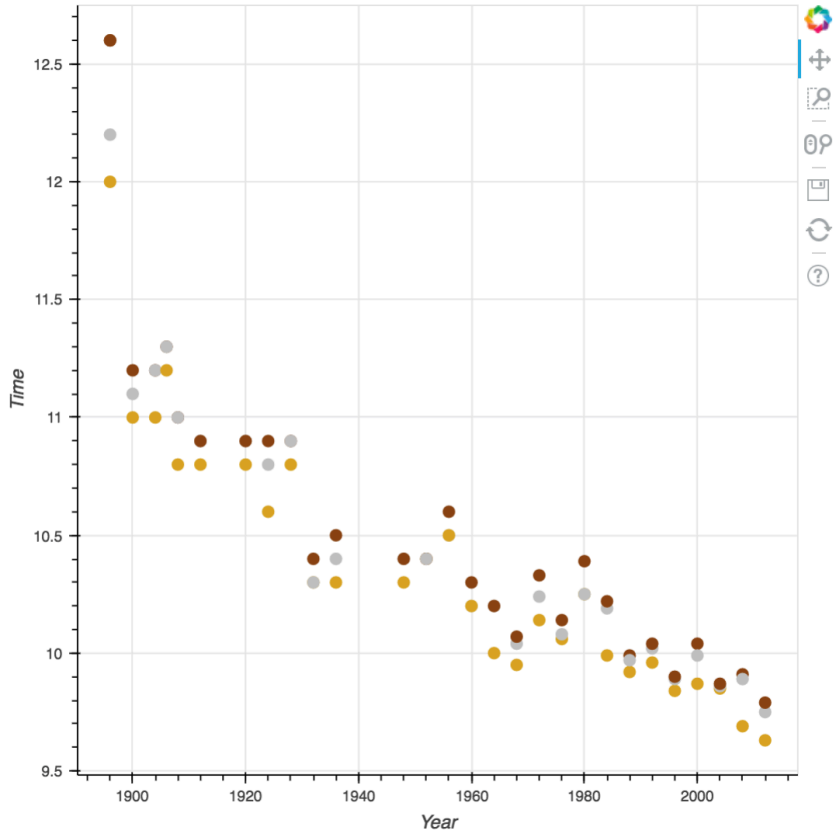
9)

The Bokeh ColumnDataSource (continued)

You can create a `ColumnDataSource` object directly from a Pandas DataFrame by passing the DataFrame to the class initializer.
In this exercise, we have imported pandas as `pd` and read in a data set containing all Olympic medals awarded in the 100 meter sprint from 1896 to 2012.
A `color` column has been added indicating the CSS colorname we wish to use in the plot for every data point.
Your job is to import the `ColumnDataSource` class, create a new `ColumnDataSource` object from the DataFrame `df`, and plot circle glyphs with `'Year'` on the x-axis and `'Time'` on the y-axis. Color each glyph by the `color` column.
The figure object `p` has already been created for you.

- Import the `ColumnDataSource` class from `bokeh.plotting`.
- Use the `ColumnDataSource()` function to make a new ColumnDataSource object called `source` from the DataFrame `df`.
- Use `p.circle()` to plot circle glyphs of `size=8` on the figure `p` with `'Year'` on the x-axis and `'Time'` on the y-axis. Be sure to also specify `source=source` and `color='color'` so that the ColumnDataSource object is used and each glyph is colored by the `color` column

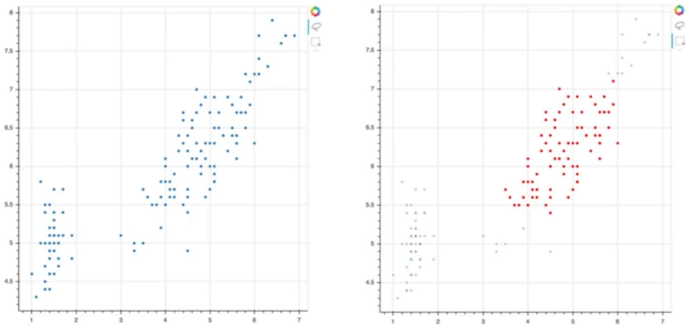
```
# Import the ColumnDataSource class from bokeh.plotting
from bokeh.plotting import ColumnDataSource

# Create a ColumnDataSource from df: source
source = ColumnDataSource(df)

# Add circle glyphs to the figure p
p.circle('Year','Time',source = source,color='color',size=8)

# Specify the name of the output file and show the result
output_file('sprint.html')
show(p)
```



Note)

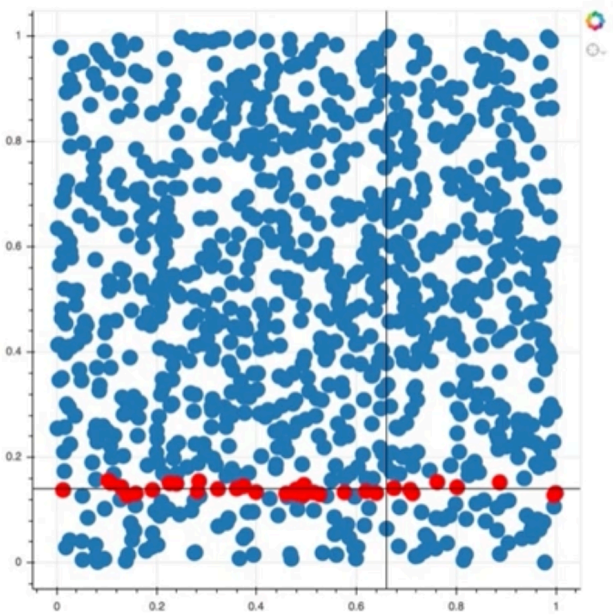
Selection appearance

```
In [1]: plot = figure(tools='box_select, lasso_select')

In [2]: plot.circle(petal_length, sepal_length,
   ...:             selection_color='red',
   ...:             nonselection_fill_alpha=0.2,
   ...:             nonselection_fill_color='grey')
```



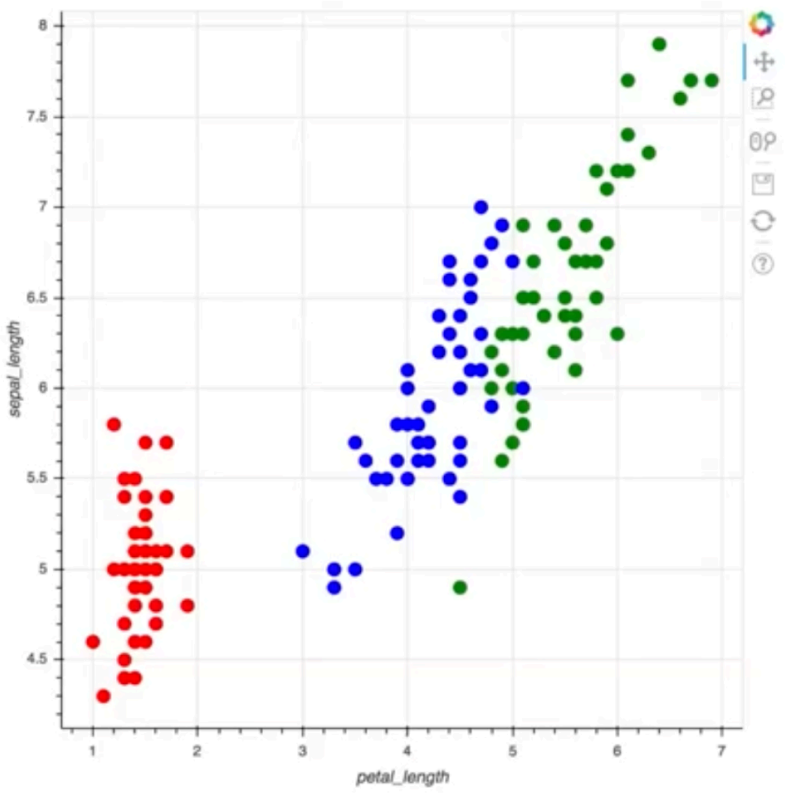
Hover appearance

```
In [1]: from bokeh.models import HoverTool

In [2]: hover = HoverTool(tooltips=None, mode='hline')

In [3]: plot = figure(tools=[hover, 'crosshair'])

In [4]: # x and y are lists of random points

In [5]: plot.circle(x, y, size=15, hover_color='red')
```



Color mapping

```
In [1]: from bokeh.models import CategoricalColorMapper

In [2]: mapper = CategoricalColorMapper(
   ...:          factors=['setosa', 'virginica',
   ...:                   'versicolor'],
   ...:          palette=['red', 'green', 'blue'])

In [3]: plot = figure(x_axis_label='petal_length',
   ...:               y_axis_label='sepal_length')

In [4]: plot.circle('petal_length', 'sepal_length',
   ...:             size=10, source=source,
   ...:             color={'field': 'species',
   ...:                    'transform': mapper})
```



10)

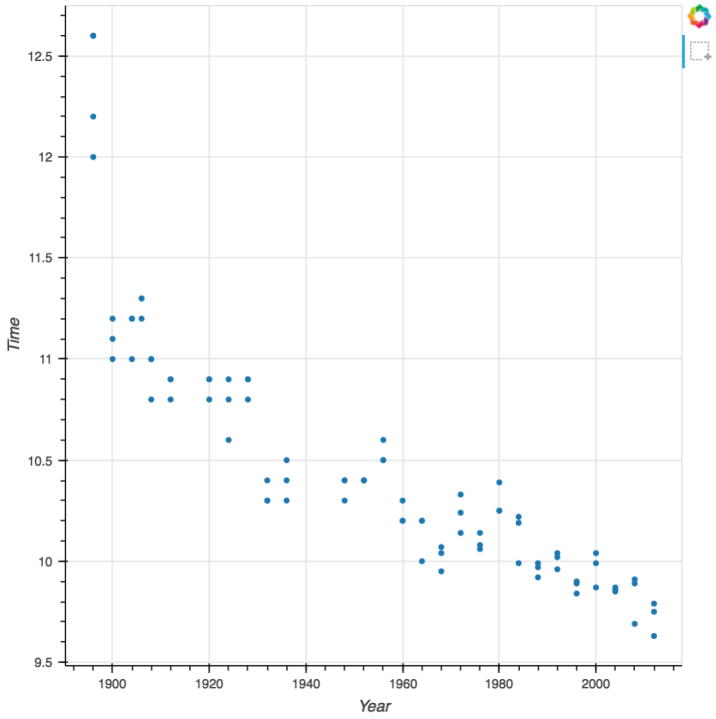
Selection and non-selection glyphs

In this exercise, you're going to add the `box_select` tool to a figure and change the selected and non-selected circle glyph properties so that selected glyphs are red and non-selected glyphs are transparent blue.

You'll use the ColumnDataSource object of the Olympic Sprint dataset you made in the last exercise. It is provided to you with the name `source`.
After you have created the figure, be sure to experiment with the Box Select tool you added! As in previous exercises, you may have to scroll down to view the lower portion of the figure.

- Create a figure `p` with an x-axis label of `'Year'`, y-axis label of `'Time'`, and the `'box_select'` tool. To add the `'box_select'` tool, you have to specify the keyword argument `tools='box_select'` inside the `figure()`function.
- Now that you have added `'box_select'` to `p`, add in circle glyphs with `p.circle()` such that the selected glyphs are red and non-selected glyphs are transparent blue. This can be done by specifying `'red'` as the argument to `selection_color` and `0.1` to `nonselection_alpha`. Remember to also pass in the arguments for the `x` (`'Year'`), `y` (`'Time'`), and `source`parameters of `p.circle()`

```
# Create a figure with the "box_select" tool: p
p = figure(tools='box_select',x_axis_label='Year',y_axis_label = 'Time')

# Add circle glyphs to the figure p with the selected and non-selected properties
p.circle(selection_color='red',nonselection_alpha=0.1,x ='Year',y='Time',source = source)

# Specify the name of the output file and show the result
output_file('selection_glyph.html')
show(p)
```

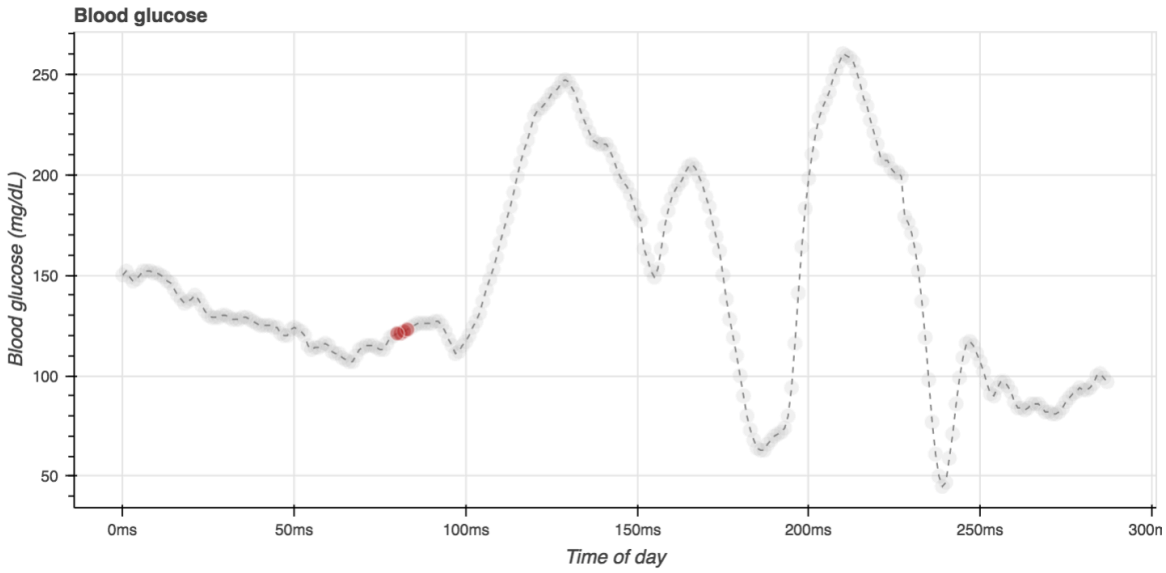


11)
Hover glyphs

Now let's practice using and customizing the hover tool.
In this exercise, you're going to plot the blood glucose levels for an unknown patient. The blood glucose levels were recorded every 5 minutes on October 7th starting at 3 minutes past midnight.
The date and time of each measurement are provided to you as `x` and the blood glucose levels in mg/dL are provided as `y`.
A bokeh figure is also provided in the workspace as `p`.
Your job is to add a circle glyph that will appear red when the mouse is hovered near the data points. You will also add a customized hover tool object to the plot.
When you're done, play around with the hover tool you just created! Notice how the points where your mouse hovers over turn red.

- Import `HoverTool` from `bokeh.models`.
- Add a circle glyph to the existing figure `p` for `x` and `y` with a `size` of `10`, `fill_color` of `'grey'`, alpha of `0.1`, `line_color` of `None`,`hover_fill_color` of `'firebrick'`, `hover_alpha` of `0.5`, and `hover_line_color` of `'white'`.
- Use the `HoverTool()` function to create a HoverTool called `hover` with `tooltips=None` and `mode='vline'`.
- Add the HoverTool `hover` to the figure `p` using the `p.add_tools()`function

```
# import the HoverTool
from bokeh.models import HoverTool

# Add circle glyphs to figure p
p.circle(x, y, size=10,
        fill_color='grey', alpha=0.1, line_color=None,
        hover_fill_color='firebrick', hover_alpha=0.5,
        hover_line_color='white')

# Create a HoverTool: hover
hover = HoverTool(tooltips=None,mode='vline')

# Add the hover tool to the figure p
p.add_tools(hover)

# Specify the name of the output file and show the result
output_file('hover_glyph.html')
show(p)
```



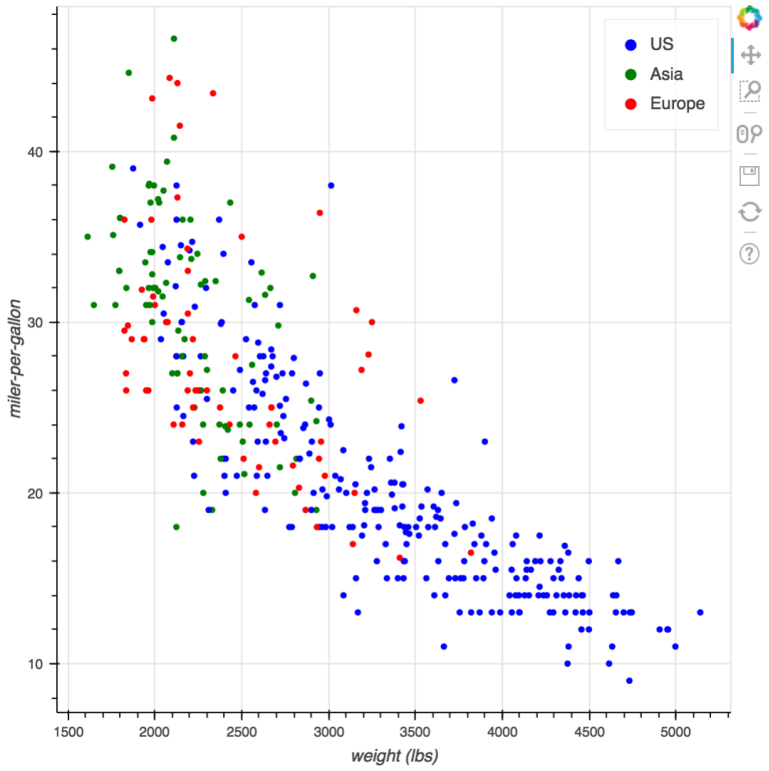
12)
Colormapping

The final glyph customization we'll practice is using the CategoricalColorMapper to color each glyph by a categorical property.
Here, you're going to use the automobile dataset to plot miles-per-gallon vs weight and color each circle glyph by the region where the automobile was manufactured.
The `origin` column will be used in the ColorMapper to color automobiles manufactured in the US as blue, Europe as red and Asia as green.
The automobile data set is provided to you as a Pandas DataFrame called `df`. The figure is provided for you as `p`.

- Import `CategoricalColorMapper` from `bokeh.models`.
- Convert the DataFrame `df` to a ColumnDataSource called `source`. This has already been done for you.
- Make a CategoricalColorMapper object called `color_mapper` with the `CategoricalColorMapper()` function. It has two parameters here: `factors`and `palette`.
- Add a `circle` glyph to the figure `p` to plot `'mpg'` (on the y-axis) vs `'weight'` (on the x-axis). Remember to pass in `source` and `'origin'` as arguments to `source` and `legend`. For the `color` parameter, use `dict(field='origin', transform=color_mapper)`.

```
#Import CategoricalColorMapper from bokeh.models
from bokeh.models import CategoricalColorMapper

# Convert df to a ColumnDataSource: source
source = ColumnDataSource(df)

# Make a CategoricalColorMapper object: color_mapper
```

```
color_mapper = CategoricalColorMapper(factors=['Europe', 'Asia', 'US'],
                                      palette=['red', 'green', 'blue'])

# Add a circle glyph to the figure p
p.circle('weight', 'mpg', source=source,
         color=dict(field='origin',transform=color_mapper),
         legend='origin')

# Specify the name of the output file and show the result
output_file('colormap.html')
show(p)
```

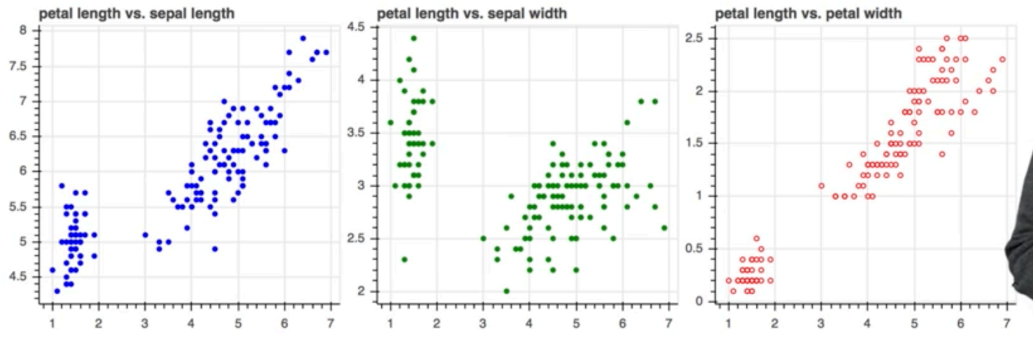


Chapter 2)

Note)

Rows of plots

```
In [1]: from bokeh.layouts import row

In [2]: layout = row(p1, p2, p3)

In [3]: output_file('row.html')

In [4]: show(layout)
```



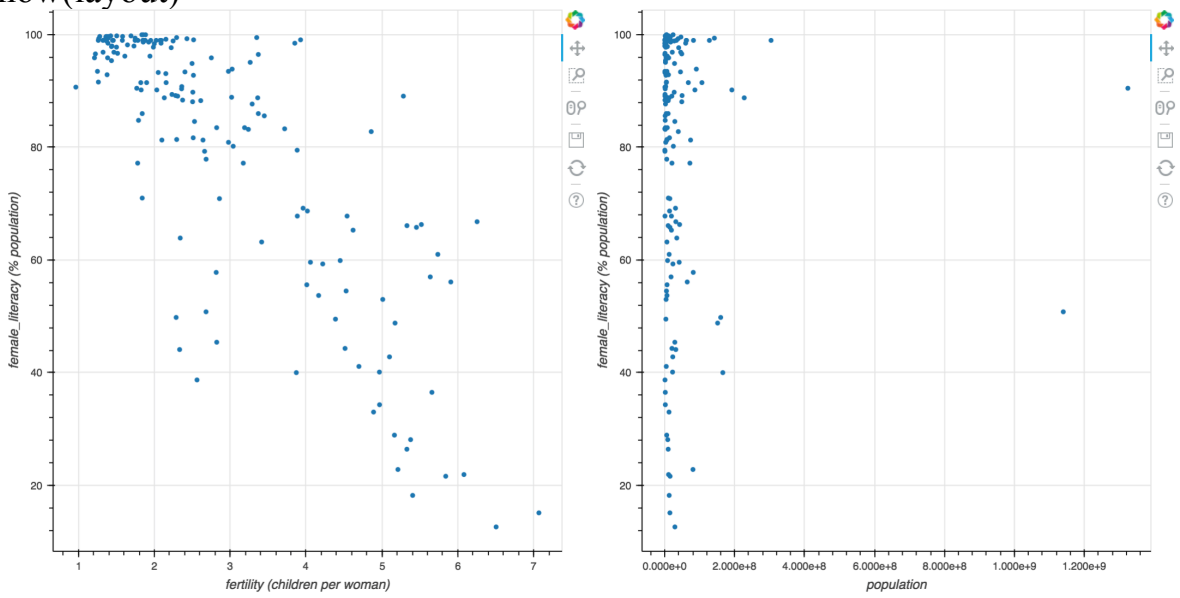
1)

Creating rows of plots

Layouts are collections of Bokeh figure objects.
In this exercise, you're going to create two plots from the Literacy and Birth Rate data set to plot fertility vs female literacy and population vs female literacy.
By using the `row()` method, you'll create a single layout of the two figures.
Remember, as in the previous chapter, once you have created your figures, you can interact with them in various ways.
In this exercise, you may have to scroll sideways to view both figures in the row layout. Alternatively, you can view the figures in a new window by clicking on the expand icon to the right of the "Bokeh plot" tab.

- Import `row` from the `bokeh.layouts` module.
- Create a new figure `p1` using the `figure()` function and specifying the two parameters `x_axis_label` and `y_axis_label`.
- Add a circle glyph to `p1`. The x-axis data is `fertility` and y-axis data is `female_literacy`. Be sure to also specify `source=source`.
- Create a new figure `p2` using the `figure()` function and specifying the two parameters `x_axis_label` and `y_axis_label`.
- Add a `circle()` glyph to `p2` and specify the `x_axis_label` and `y_axis_label` parameters.
- Put `p1` and `p2` into a horizontal layout using `row()`.

```
# Import row from bokeh.layouts
from bokeh.layouts import row

# Create the first figure: p1
p1 = figure(x_axis_label='fertility (children per woman)',
y_axis_label='female_literacy (% population)')

# Add a circle glyph to p1
p1.circle('fertility', 'female_literacy', source=source)

# Create the second figure: p2
p2 = figure(x_axis_label='population', y_axis_label='female_literacy (%
population)')

# Add a circle glyph to p2
p2.circle('population','female_literacy',source=source)

# Put p1 and p2 into a horizontal row: layout
layout = row(p1,p2)

# Specify the name of the output_file and show the result
output_file('fert_row.html')
show(layout)
```



2)

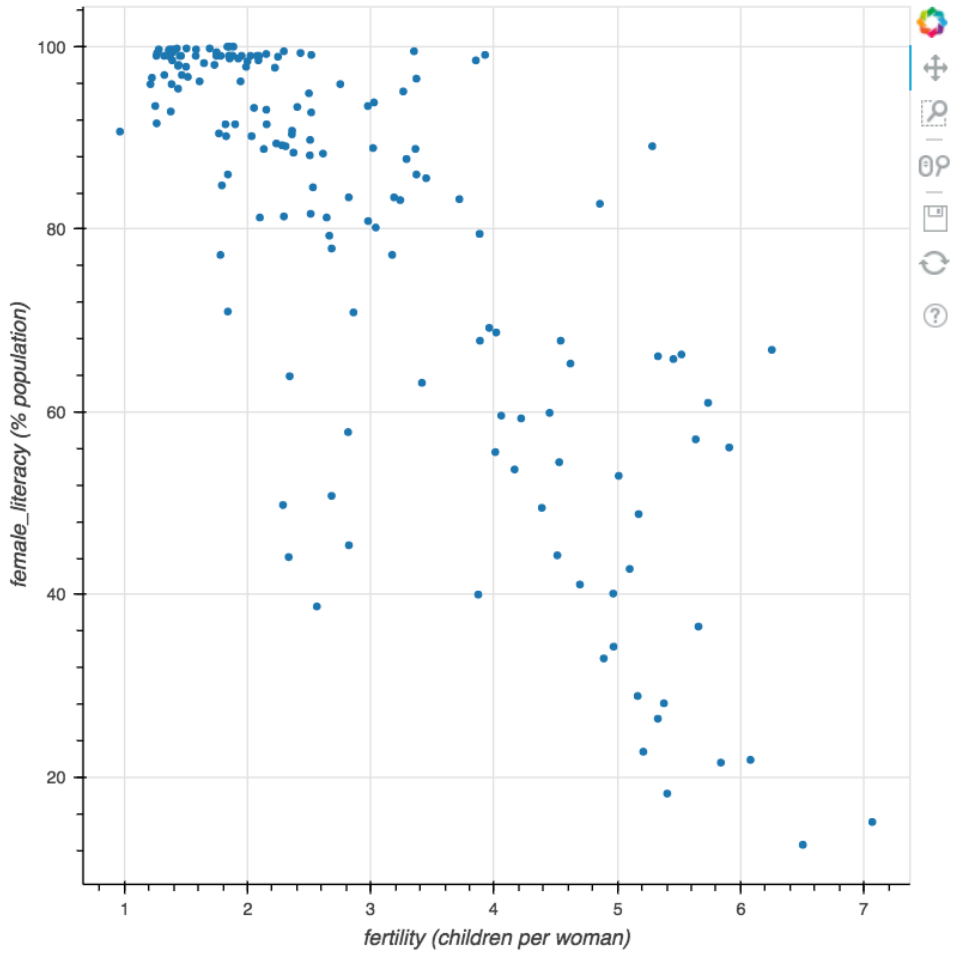
Creating columns of plots

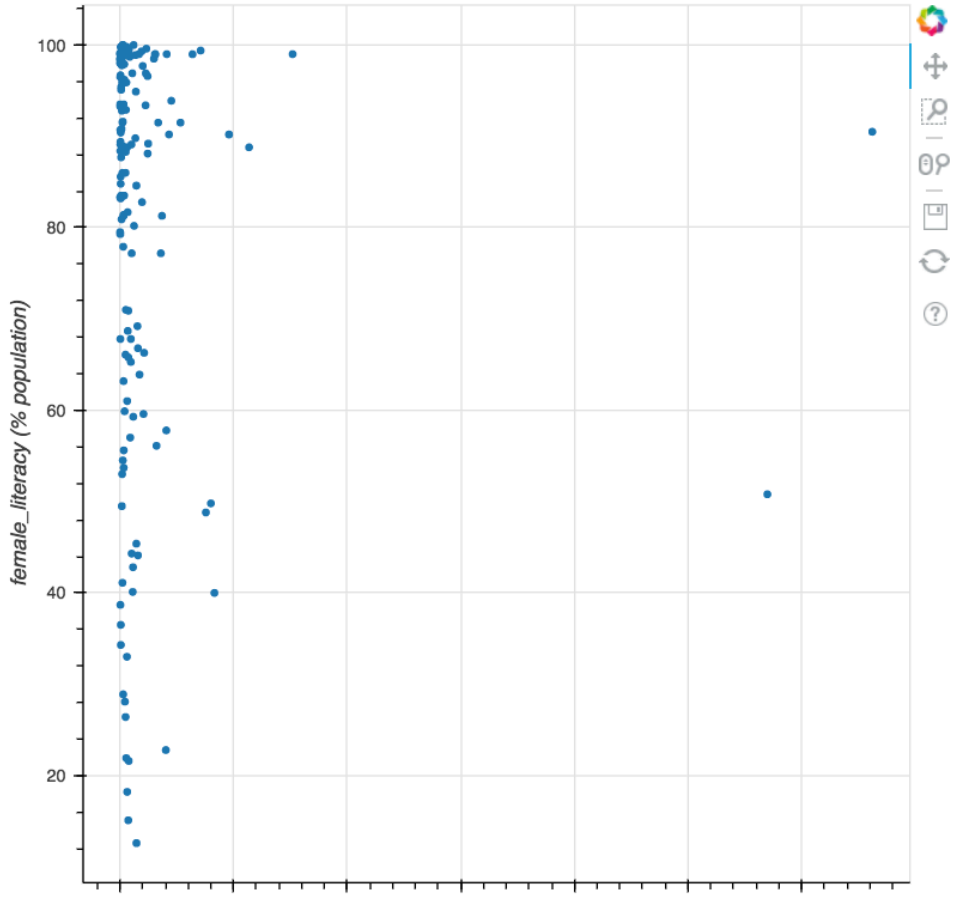
In this exercise, you're going to use the `column()` function to create a single column layout of the two plots you created in the previous exercise.

Figure `p1` has been created for you.
In this exercise and the ones to follow, you may have to scroll down to view the lower portion of the figure.

- Import `column` from the `bokeh.layouts` module.
- The figure `p1` has been created for you. Create a new figure `p2` with an x-axis label of `'population'` and y-axis label of `'female_literacy (% population)'`.
- Add a circle glyph to the figure `p2`.
- Put `p1` and `p2` into a vertical layout using `column()`.

```
# Import column from the bokeh.layouts module
from bokeh.layouts import column

# Create a blank figure: p1
p1 = figure(x_axis_label='fertility (children per woman)',
y_axis_label='female_literacy (% population)')

# Add circle scatter to the figure p1
p1.circle('fertility', 'female_literacy', source=source)

# Create a new blank figure: p2
p2 = figure(x_axis_label='population',y_axis_label='female_literacy (%
population)')

# Add circle scatter to the figure p2

p2.circle('population','female_literacy',source=source)
# Put plots p1 and p2 in a column: layout
layout = column(p1,p2)

# Specify the name of the output_file and show the result
output_file('fert_column.html')
show(layout)
```


female_literacy (% population)
fertility (children per woman)
100
80
60
40
20
1
2
3
4
5
6
7


female_literacy (% population)
100
80
60
40
20

3)
Nesting rows and columns of plots

You can create nested layouts of plots by combining row and column layouts.
In this exercise, you'll make a 3-plot layout in two rows using the auto-mpg data set.
Three plots have been created for you of average mpg vs year, mpg vs hp, and mpg vs weight.
Your job is to use the `column()` and `row()` functions to make a two-row layout where the first row will have only the average mpg vs year plot and the second row will have mpg vs hp and mpg vs weight plots as columns.
By using the `sizing_mode` argument, you can scale the widths to fill the whole figure.

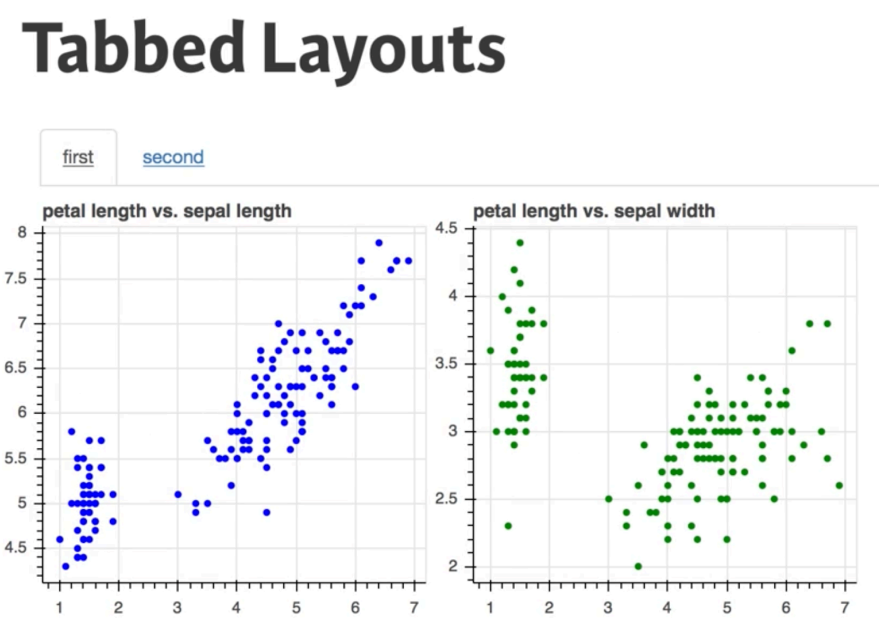
- Import `row` and `column` from `bokeh.layouts`.
- Create a column layout called `row2` with the figures `mpg_hp` and `mpg_weight` in a list and set `sizing_mode='scale_width'`.
- Create a row layout called `layout` with the figure `avg_mpg` and the column layout `row2` in a list and set `sizing_mode='scale_width'`

```
# Import column and row from bokeh.layouts
from bokeh.layouts import row, column

# Make a column layout that will be used as the second row: row2
row2 = column([mpg_hp,mpg_weight], sizing_mode='scale_width')

# Make a row layout that includes the above column layout: layout
layout = row([avg_mpg,row2], sizing_mode='scale_width')

# Specify the name of the output_file and show the result
output_file('layout_custom.html')
show(layout)
```

Note)

Tabbed Layouts

```
In [1]: from bokeh.models.widgets import Tabs, Panel

In [2]: # Create a Panel with a title for each tab

In [3]: first = Panel(child=row(p1, p2), title='first')

In [4]: second = Panel(child=row(p3), title='second')

In [5]: # Put the Panels in a Tabs object

In [6]: tabs = Tabs(tabs=[first, second])

In [7]: output_file('tabbed.html')

In [8]: show(layout)
```



4)

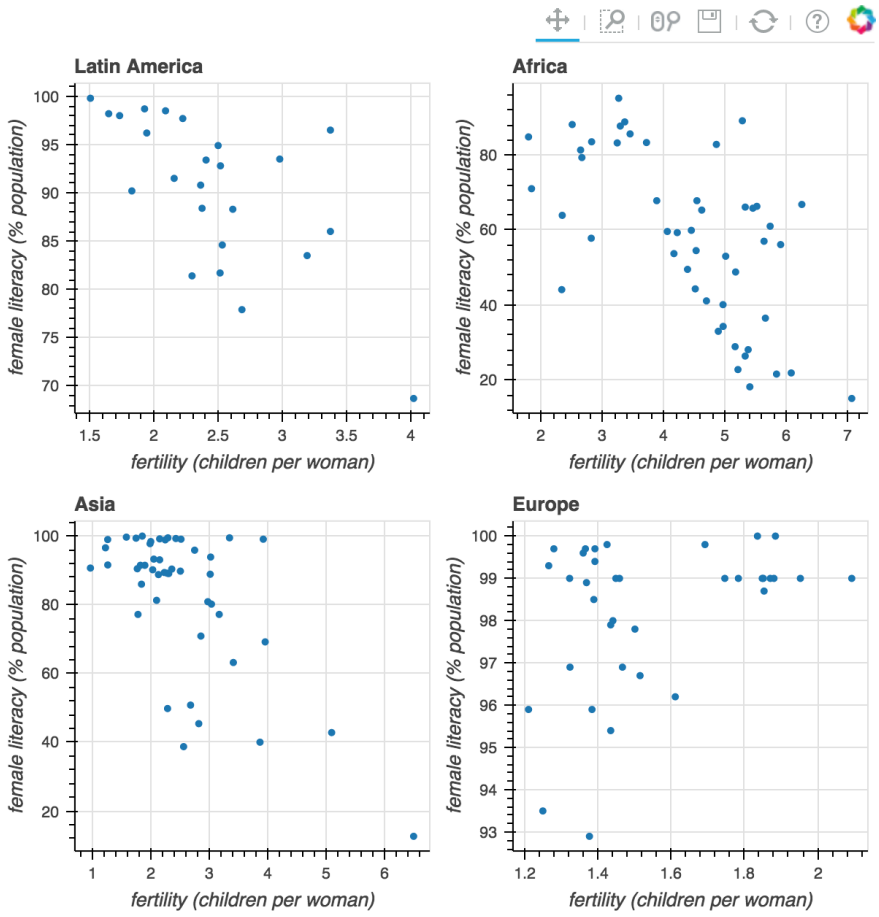
Creating gridded layouts

Regular grids of Bokeh plots can be generated with `gridplot`.
In this example, you're going to display four plots of fertility vs female literacy for four regions: Latin America, Africa, Asia and Europe.
Your job is to create a list-of-lists for the four Bokeh plots that have been provided to you as `p1`, `p2`, `p3` and `p4`. The list-of-lists defines the row and column placement of each plot.

- Import `gridplot` from the `bokeh.layouts` module.
- Create a list called `row1` containing plots `p1` and `p2`.

- Create a list called `row2` containing plots `p3` and `p4`.
- Create a gridplot using `row1` and `row2`. You will have to pass in `row1` and `row2` in the form of a list.

```
# Import gridplot from bokeh.layouts
from bokeh.layouts import gridplot

# Create a list containing plots p1 and p2: row1
row1 = [p1,p2]

# Create a list containing plots p3 and p4: row2
row2 = [p3,p4]

# Create a gridplot using row1 and row2: layout
layout = gridplot([row1,row2])

# Specify the name of the output_file and show the result
output_file('grid.html')
show(layout)
```



5)
Starting tabbed layouts

Tabbed layouts can be created in Pandas by placing plots or layouts in Panels.
In this exercise, you'll take the four fertility vs female literacy plots from the last exercise and make a `Panel()` for each.
No figure will be generated in this exercise. Instead, you will use these panels in the next exercise to build and display a tabbed layout.

- Import `Panel` from `bokeh.models.widgets`.
- Create a new panel `tab1` with child `p1` and a title of `'Latin America'`.
- Create a new panel `tab2` with child `p2` and a title of `'Africa'`.
- Create a new panel `tab3` with child `p3` and a title of `'Asia'`.
- Create a new panel `tab4` with child `p4` and a title of `'Europe'`

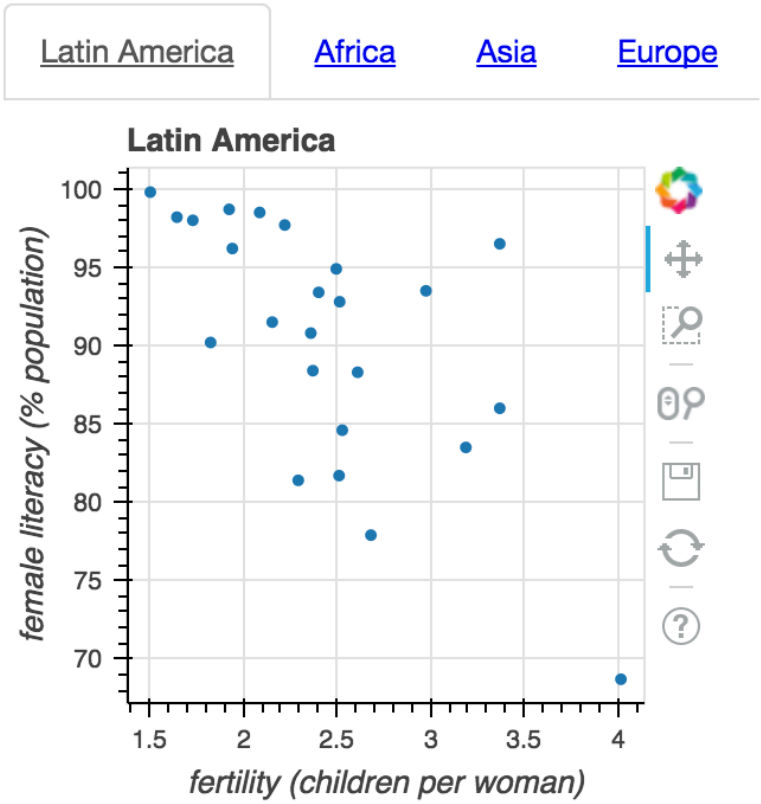
```
# Import Panel from bokeh.models.widgets
from bokeh.models.widgets import Panel

# Create tab1 from plot p1: tab1
tab1 = Panel(child=p1, title='Latin America')

# Create tab2 from plot p2: tab2
tab2 = Panel(child=p2, title='Africa')

# Create tab3 from plot p3: tab3
tab3 = Panel(child=p3, title='Asia')

# Create tab4 from plot p4: tab4
tab4 = Panel(child=p4, title='Europe')
```

6)
Displaying tabbed layouts

Tabbed layouts are collections of Panel objects. Using the figures and Panels from the previous two exercises, you'll create a tabbed layout to change the region in the fertility vs female literacy plots.
Your job is to create the layout using `Tabs()` and assign the `tabs`keyword argument to your list of Panels. The Panels have been created for you as `tab1`, `tab2`, `tab3` and `tab4`.
After you've displayed the figure, explore the tabs you just added! The "Pan", "Box Zoom" and "Wheel Zoom" tools are also all available as before.

- Import `Tabs` from `bokeh.models.widgets`.
- Create a `Tabs` layout called `layout` with `tab1`, `tab2`, `tab3`, and `tab4`.

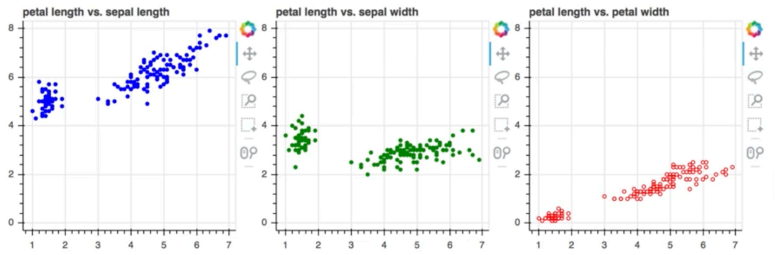
```
# Import Tabs from bokeh.models.widgets
from bokeh.models.widgets import Tabs

# Create a Tabs layout: layout
layout = Tabs(tabs=[tab1, tab2, tab3, tab4])

# Specify the name of the output_file and show the result
output_file('tabs.html')
show(layout)
```



Note)

Linking axes

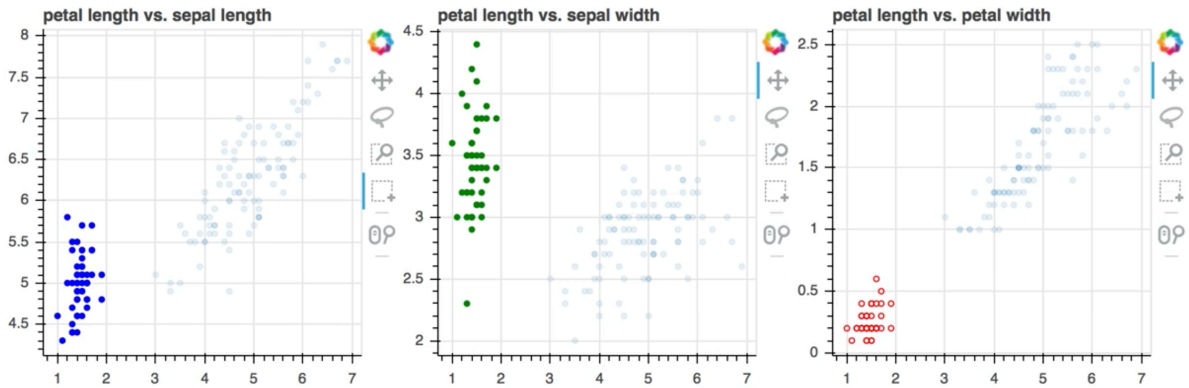
```
In [1]: p3.x_range = p2.x_range = p1.x_range

In [2]: p3.y_range = p2.y_range = p1.y_range
```



Linking selections

```
In [1]: p1 = figure(title='petal length vs. sepal length')

In [2]: p1.circle('petal_length', 'sepal_length',
   ...:           color='blue', source=source)

In [3]: p2 = figure(title='petal length vs. sepal width')

In [4]: p2.circle('petal_length', 'sepal_width',
   ...:           color='green', source=source)

In [5]: p3 = figure(title='petal length vs. petal width')

In [6]: p3.circle('petal_length', 'petal_width',
   ...:           line_color='red', fill_color=None,
   ...:           source=source)
```

Linking selections

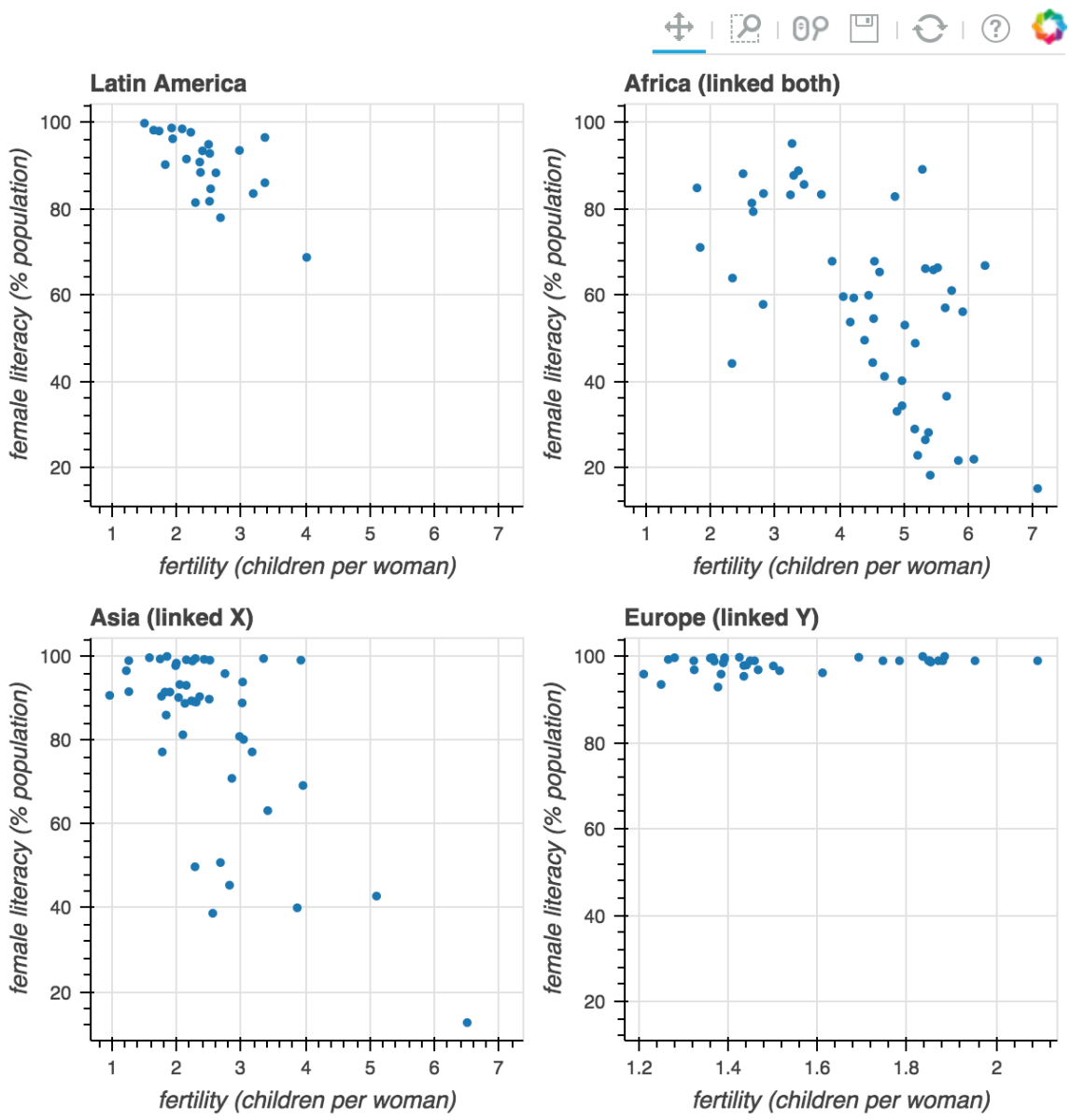


7)
Linked axes

Linking axes between plots is achieved by sharing `range` objects.
In this exercise, you'll link four plots of female literacy vs fertility so that when one plot is zoomed or dragged, one or more of the other plots will respond.
The four plots `p1`, `p2`, `p3` and `p4` along with the `layout` that you created in the last section have been provided for you.
Your job is link `p1` with the three other plots by assignment of the `.x_range` and `.y_range` attributes.
After you have linked the axes, explore the plots by clicking and dragging along the x or y axes of any of the plots, and notice how the linked plots change together.

- Link the `x_range` of `p2` to `p1`.
- Link the `y_range` of `p2` to `p1`.
- Link the `x_range` of `p3` to `p1`.
- Link the `y_range` of `p4` to `p1`.

```
# Link the x_range of p2 to p1: p2.x_range
p2.x_range = p1.x_range

# Link the y_range of p2 to p1: p2.y_range
p2.y_range = p1.y_range

# Link the x_range of p3 to p1: p3.x_range
p3.x_range = p1.x_range

# Link the y_range of p4 to p1: p4.y_range
p4.y_range = p1.y_range

# Specify the name of the output_file and show the result
output_file('linked_range.html')
show(layout)
```



8)

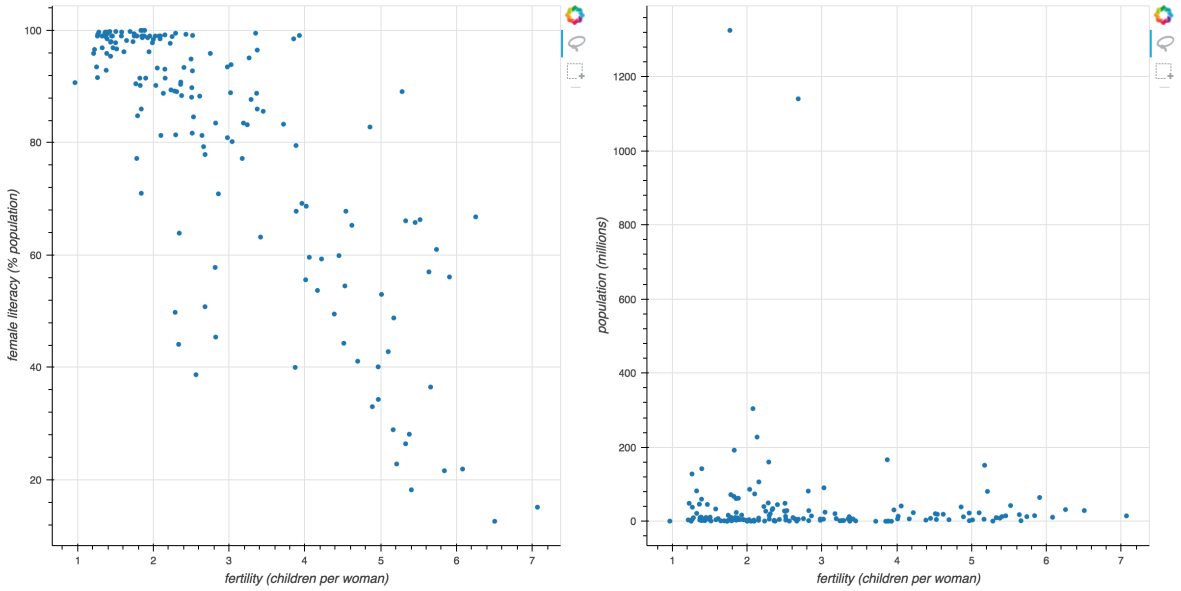
Linked brushing

By sharing the same `ColumnDataSource` object between multiple plots, selection tools like BoxSelect and LassoSelect will highlight points in both plots that share a row in the ColumnDataSource.
In this exercise, you'll plot female literacy vs fertility and population vs fertility in two plots using the same ColumnDataSource.
After you have built the figure, experiment with the Lasso Select and Box Select tools. Use your mouse to drag a box or lasso around points in one figure, and notice how points in the other figure that share a row in the ColumnDataSource also get highlighted.

Before experimenting with the Lasso Select, however, click the Bokeh plot pop-out icon to pop out the figure so that you can definitely see everything that you're doing.

- Create a `ColumnDataSource` object called `source` from the `data`DataFrame.
- Create a new figure `p1` using the `figure()` function. In addition to specifying the parameters `x_axis_label` and `y_axis_label`, you will also have to specify the BoxSelect and LassoSelect selection tools with `tools='box_select,lasso_select'`.
- Add a circle glyph to `p1`. The x-axis data is `fertility` and y-axis data is `female literacy`. Be sure to also specify `source=source`.
- Create a second figure `p2` similar to how you created `p1`.
- Add a circle glyph to `p2`. The x-axis data is `fertility` and y-axis data is `population`. Be sure to also specify `source=source`.

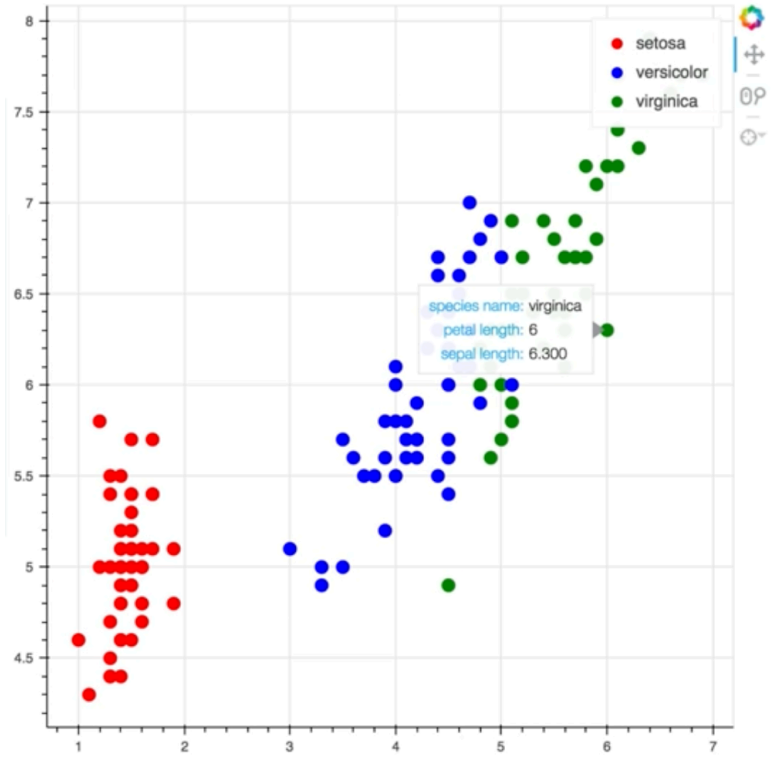
```
# Create ColumnDataSource: source
source = ColumnDataSource(data)

# Create the first figure: p1
p1 = figure(x_axis_label='fertility (children per woman)', y_axis_label='female
literacy (% population)',
           tools='box_select,lasso_select')

# Add a circle glyph to p1
p1.circle('fertility','female literacy',source = source)

# Create the second figure: p2
p2 = figure(x_axis_label='fertility (children per woman)',
y_axis_label='population (millions)',
           tools='box_select,lasso_select')

# Add a circle glyph to p2
p2.circle('fertility','population',source=source)

# Create row layout of figures p1 and p2: layout
layout = row(p1,p2)

# Specify the name of the output_file and show the result
output_file('linked_brush.html')
show(layout)
```



Note)

Hover Tooltips

```
In [1]: from bokeh.models import HoverTool

In [2]: hover = HoverTool(tooltips=[
   ...:          ('species name', '@species'),
   ...:          ('petal length', '@petal_length'),
   ...:          ('sepal length', '@sepal_length'),
   ...:          ])

In [3]: plot = figure(tools=[hover, 'pan',
   ...:                 'wheel_zoom'])
```



9)

How to create legends

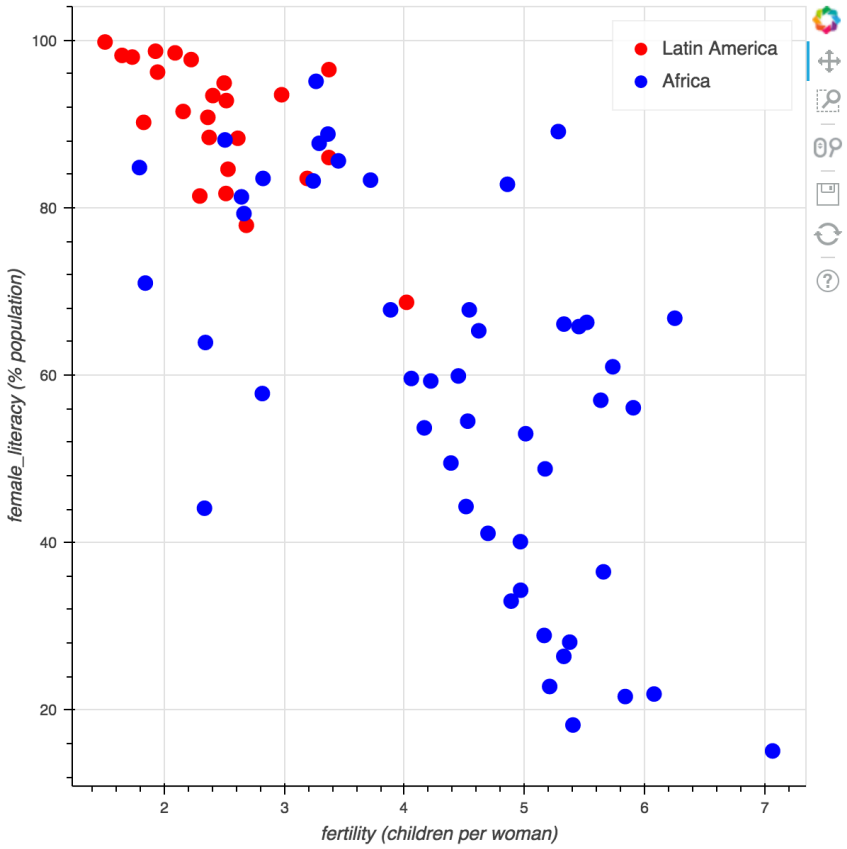
Legends can be added to any glyph by using the `legend` keyword argument.
In this exercise, you will plot two `circle` glyphs for female literacy vs fertility in Africa and Latin America.
Two ColumnDataSources called `latin_america` and `africa` have been provided.
Your job is to plot two `circle` glyphs for these two objects with `fertility` on the x axis and `female_literacy` on the y axis and add the `legend` values. The figure `p` has been provided for you.

- Add a `red` circle glyph to the figure `p` using the `latin_america`ColumnDataSource. Specify a `size` of `10` and `legend` of `Latin America`.
- Add a `blue` circle glyph to the figure `p` using the `africa`ColumnDataSource. Specify a `size` of `10` and `legend` of `Africa`

# Add the first circle glyph to the figure p
p.circle('fertility', 'female_literacy', source=latin_america, size=10, color='red', legend='Latin America')

# Add the second circle glyph to the figure p
p.circle('fertility', 'female_literacy', source=africa, size=10, color='blue', legend='Africa')

# Specify the name of the output_file and show the result
output_file('fert_lit_groups.html')
show(p)



10)
Positioning and styling legends

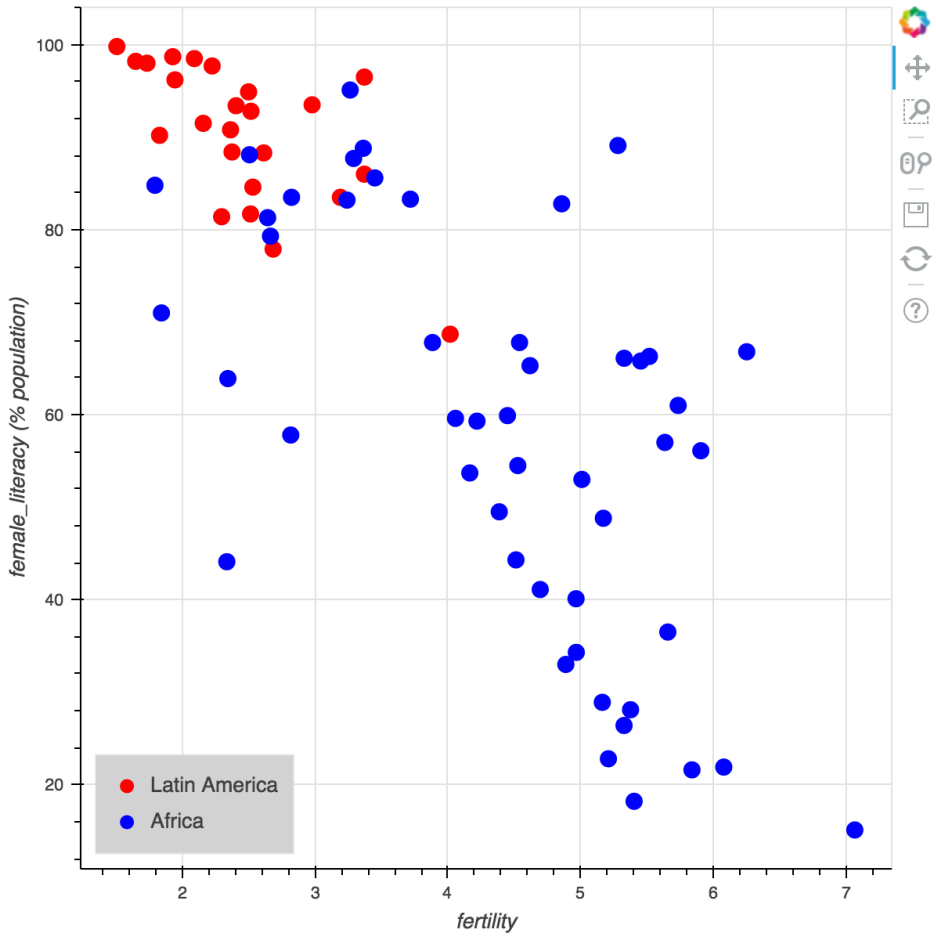
Properties of the legend can be changed by using the `legend` member attribute of a Bokeh figure after the glyphs have been plotted.
In this exercise, you'll adjust the background color and legend location of the female literacy vs fertility plot from the previous exercise.
The figure object `p` has been created for you along with the circle glyphs.

- Use `p.legend.location` to adjust the legend location to be on the `'bottom_left'`.
- Use `p.legend.background_fill_color` to set the background color of the legend to `'lightgray'`

# Assign the legend to the bottom left: p.legend.location
p.legend.location ='bottom_left'

Fill the legend background with the color 'lightgray':
p.legend.background_fill_color

```
p.legend.background_fill_color = 'lightgray'

# Specify the name of the output_file and show the result
output_file('fert_lit_groups.html')
show(p)
```



11)

Hover tooltips for exposing details

When configuring hover tools, certain pre-defined fields such as mouse position or glyph index can be accessed with `$` -prefixed names, for example `$x` , `$index` . But tooltips can display values from arbitrary columns in a `ColumnDataSource` .

What is the correct format to display values from a column `"sales"` in a hover tooltip?

ANSWER THE QUESTION 50XP

Possible Answers

- `&{sales}` press 1
- `%sales%` press 2
- `@sales` press 3

12)

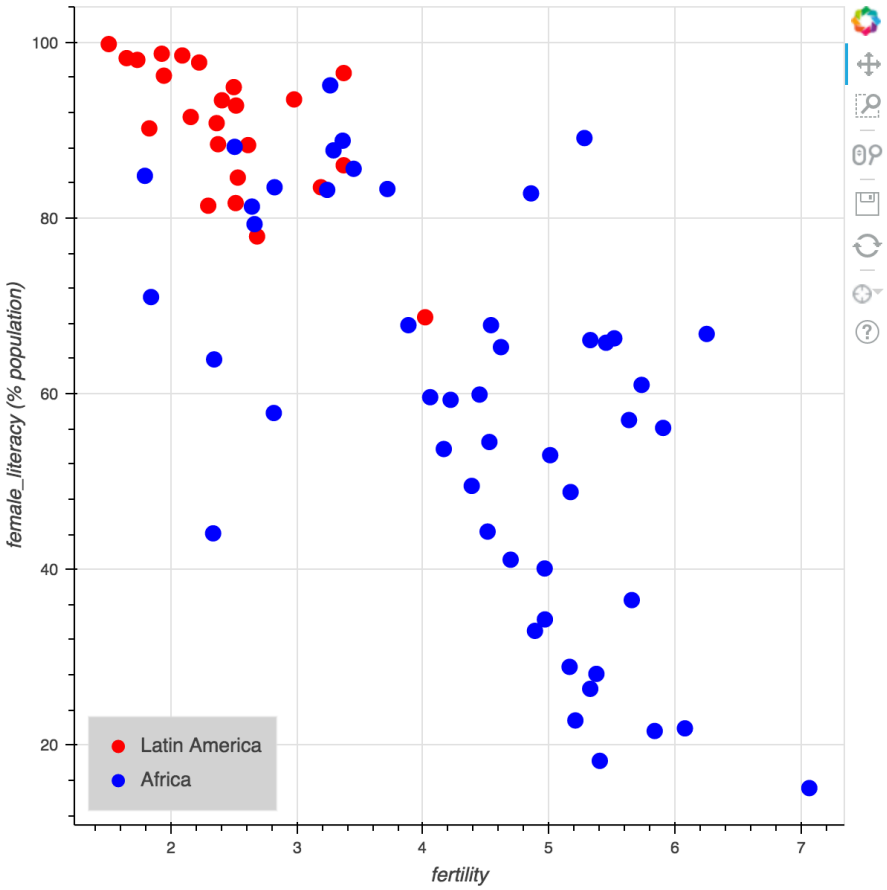
Adding a hover tooltip

Working with the `HoverTool` is easy for data stored in a ColumnDataSource.
In this exercise, you will create a `HoverTool` object and display the country for each circle glyph in the figure that you created in the last exercise. This is done by assigning the `tooltips` keyword argument to a list-of-tuples specifying the label and the column of values from the ColumnDataSource using the `@` operator.
The figure object has been prepared for you as `p`.
After you have added the hover tooltip to the figure, be sure to interact with it by hovering your mouse over each point to see which country it represents.

- Import the `HoverTool` class from `bokeh.models`.
- Use the `HoverTool()` function to create a `HoverTool` object called `hover` and set the `tooltips` argument to be `[('Country','@Country')]`.
- Use `p.add_tools()` with your `HoverTool` object to add it to the figure.

```
# Import HoverTool from bokeh.models
from bokeh.models import HoverTool

# Create a HoverTool object: hover
hover = HoverTool(tooltips = [('Country','@Country')])

# Add the HoverTool object to figure p
p.add_tools(hover)

# Specify the name of the output_file and show the result
```

```
output_file('hover.html')
show(p)
```



Chapter 3)

Note)

Basic App Outline

outline.py

```
from bokeh.io import curdoc

# Create plots and widgets

# Add callbacks

# Arrange plots and widgets in layouts

curdoc().add_root(layout)
```

Running Bokeh Applications

Run single module apps at the shell or Windows command prompt:

```
bokeh serve --show myapp.py
```

“Directory” style apps run similarly:

```
bokeh serve --show myappdir/
```

1)

Using the current document

Let's get started with building an interactive Bokeh app. This typically begins with importing the `curdoc`, or "current document", function from `bokeh.io`. This current document will eventually hold all the plots, controls, and layouts that you create. Your job in this exercise is to use this function to add a single plot to your application.

In the video, Bryan described the process for running a Bokeh app using the `bokeh serve` command line tool. In this chapter and the one that follows, the DataCamp environment does this for you behind the scenes. Notice that your code is part of

a `script.py` file. When you hit 'Submit Answer', you'll see in the IPython Shell that we call `bokeh serve script.py` for you.
Remember, as in the previous chapters, that there are different options available for you to interact with your plots, and as before, you may have to scroll down to view the lower portion of the plots.

- Import `curdoc` from `bokeh.io` and `figure` from `bokeh.plotting`.
- Create a new plot called `plot` using the `figure()` function.
- Add a line to the plot using `[1,2,3,4,5]` as the `x` coordinates and `[2,5,4,6,7]` as the `y` coordinates.
- Add the `plot` to the current document using `curdoc().add_root()`. It needs to be passed in as an argument to `add_root()`

```
# Perform necessary imports
from bokeh.io import curdoc
from bokeh.plotting import figure

# Create a new plot: plot
plot = figure()

# Add a line to the plot
plot.line([1,2,3,4,5],[2,5,4,6,7])

# Add the plot to the current document
curdoc().add_root(plot)
```

2)
Add a single slider

In the previous exercise, you added a single plot to the "current document" of your application. In this exercise, you'll practice adding a layout to your current document.
Your job here is to create a single slider, use it to create a widgetbox layout, and then add this layout to the current document.
The slider you create here cannot be used for much, but in the later exercises, you'll use it to update your plots!

- Import `curdoc` from `bokeh.io`, `widgetbox` from `bokeh.layouts`, and `Slider` from `bokeh.models`.
- Create a slider called `slider` by using the `Slider()` function and specifying the parameters `title`, `start`, `end`, `step`, and `value`.
- Use the slider to create a widgetbox layout called `layout`.
- Add the layout to the current document using `curdoc().add_root()`. It needs to be passed in as an argument to `add_root()`

```
# Perform the necessary imports
from bokeh.io import curdoc
from bokeh.layouts import widgetbox
from bokeh.models import Slider

# Create a slider: slider
slider = Slider(title='my slider', start=0, end=10, step=0.1, value=2)

# Create a widgetbox layout: layout
layout = widgetbox(slider)

# Add the layout to the current document
curdoc().add_root(layout)
```

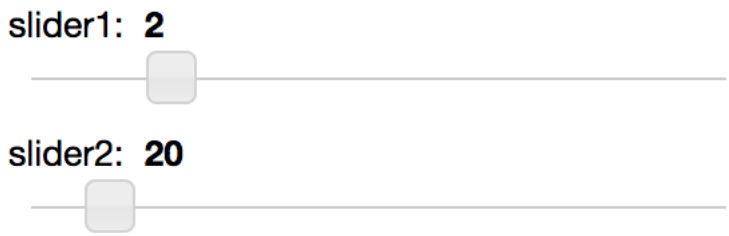
3)
Multiple sliders in one document

Having added a single slider in a widgetbox layout to your current document, you'll now add multiple sliders into the current document.
Your job in this exercise is to create two sliders, add them to a widgetbox layout, and then add the layout into the current document.

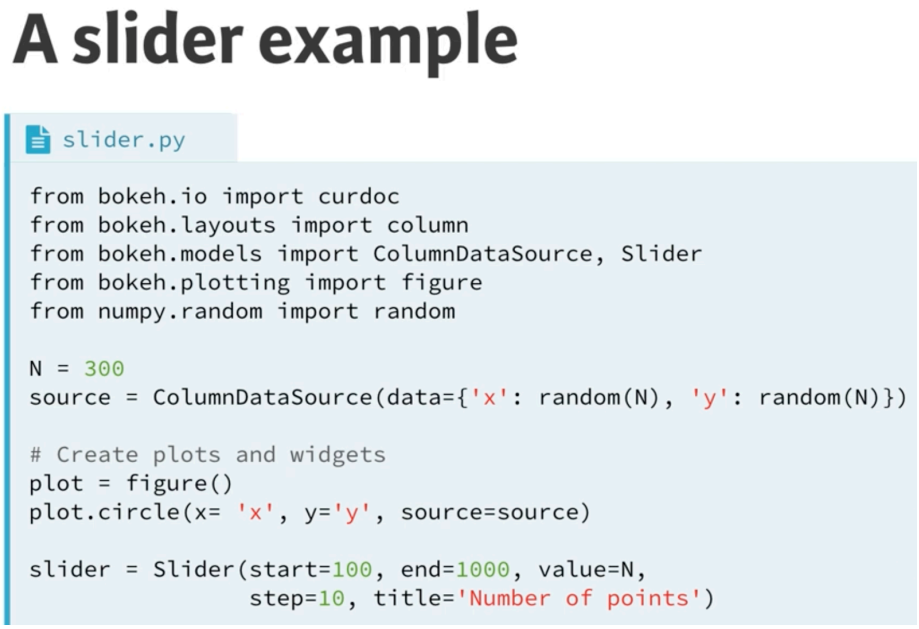
- Create the first slider, `slider1`, using the `Slider()` function. Give it a title of `'slider1'`. Have it `start` at `0`, `end` at `10`, with a `step` of `0.1` and initial `value` of `2`.
- Create the second slider, `slider2`, using the `Slider()` function. Give it a title of `'slider2'`. Have it `start` at `10`, `end` at `100`, with a `step` of `1` and initial `value` of `20`.
- Use `slider1` and `slider2` to create a widgetbox layout called `layout`.
- Add the layout to the current document using `curdoc().add_root()`

```
# Perform necessary imports
from bokeh.io import curdoc
from bokeh.layouts import widgetbox
from bokeh.models import Slider

# Create first slider: slider1
slider1 = Slider(title='slider1',start=0,end=10,step=0.1,value=2)
```

```
# Create second slider: slider2
slider2 = Slider(title='slider2',start=10,end=100,step=1,value=20)

# Add slider1 and slider2 to a widgetbox
layout = widgetbox(slider1,slider2)

# Add the layout to the current document
curdoc().add_root(layout)
```



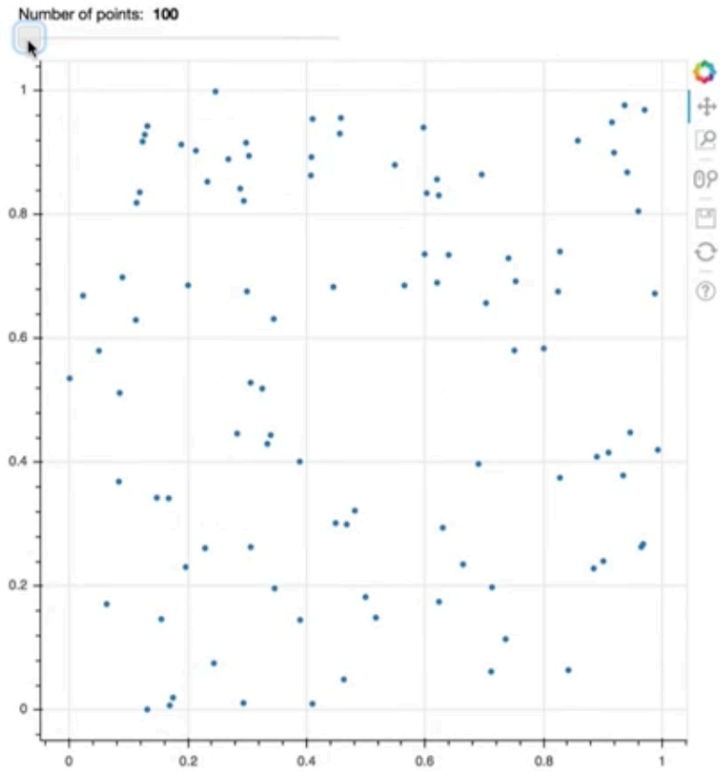
Note)



A slider example

slider.py

```
# (continued)

# Add callback to widgets
def callback(attr, old, new):
    N = slider.value
    source.data={'x': random(N), 'y': random(N)}
slider.on_change('value', callback)

# Arrange plots and widgets in layouts
layout = column(slider, plot)

curdoc().add_root(layout)
```



4)

How to combine Bokeh models into layouts

Let's begin making a Bokeh application that has a simple slider and plot, that also updates the plot based on the slider.
In this exercise, your job is to first explicitly create a ColumnDataSource. You'll then combine a plot and a slider into a single column layout, and add it to the current document.

After you are done, notice how in the figure you generate, the slider will not actually update the plot, because a widget callback has not been defined. You'll learn how to update the plot using widget callbacks in the next exercise.
All the necessary modules have been imported for you. The plot is available in the workspace as `plot`, and the slider is available as `slider`.

- Create a ColumnDataSource called `source`. Explicitly specify the `data` parameter of `ColumnDataSource()` with `{'x': x, 'y': y}`.
- Add a line to the figure `plot`, with `'x'` and `'y'` from the ColumnDataSource.
- Combine the slider and the plot into a column layout called `layout`. Be sure to first create a widgetbox layout using `widgetbox()` with `slider` and pass that into the `column()` function along with `plot`

```
# Create ColumnDataSource: source
source = ColumnDataSource(data={'x':x,'y':y})

# Add a line to the plot
plot.line('x', 'y', source=source)

# Create a column layout: layout
layout = column(widgetbox(slider),plot)

# Add the layout to the current document
curdoc().add_root(layout)
```

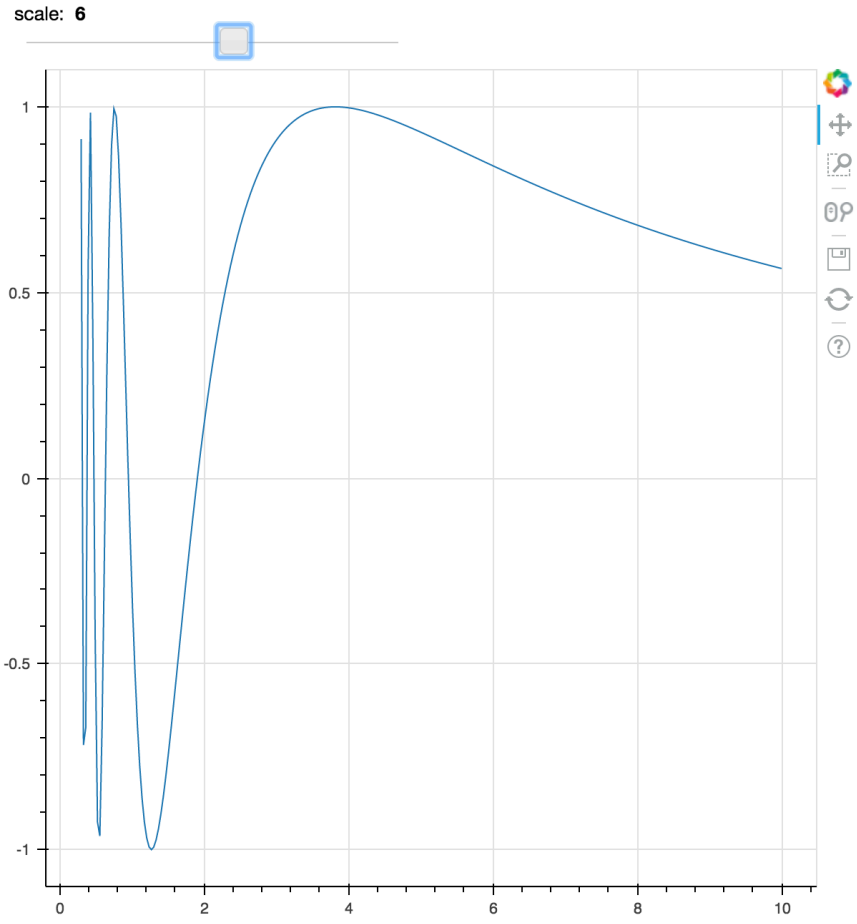
5)
Learn about widget callbacks

You'll now learn how to use widget callbacks to update the state of a Bokeh application, and in turn, the data that is presented to the user.
Your job in this exercise is to use the slider's `on_change()` function to update the plot's data from the previous example. NumPy's `sin()`function will be used to update the y-axis data of the plot.
Now that you have added a widget callback, notice how as you move the slider of your app, the figure also updates!

- Define a callback function `callback` with the parameters `attr`, `old`, `new`.
- Read the current value of `slider` as a variable `scale`. You can do this using `slider.value`.
- Compute values for the updated y using `np.sin(scale/x)`.
- Update `source.data` with the new data dictionary.
- Attach the callback to the `'value'` property of `slider`. This can be done using `on_change()` and passing in `'value'` and `callback`

```
# Define a callback function: callback
```

```
def callback(attr, old, new):

    # Read the current value of the slider: scale
    scale = slider.value

    # Compute the updated y using np.sin(scale/x): new_y
    new_y = np.sin(scale/x)

    # Update source with the new data values
    source.data = {'x': x, 'y': new_y}

# Attach the callback to the 'value' property of slider
slider.on_change('value',callback)

# Create layout and add to current document
layout = column(widgetbox(slider), plot)
curdoc().add_root(layout)
```



Note)

A Select example

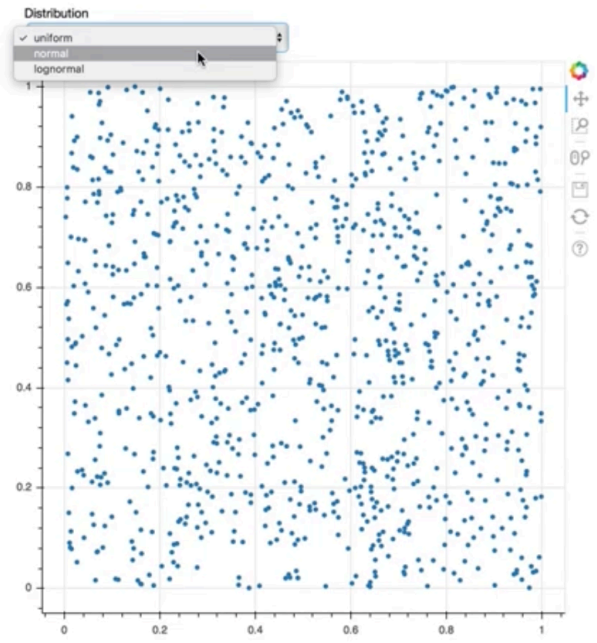
select.py

```
from bokeh.io import curdoc
from bokeh.layouts import column
from bokeh.models import ColumnDataSource, Select
from bokeh.plotting import figure
from numpy.random import random, normal, lognormal

N = 1000
source = ColumnDataSource(data={'x': random(N), 'y': random(N)})

# Create plots and widgets
plot = figure()
plot.circle(x='x', y='y', source=source)

menu = Select(options=['uniform', 'normal', 'lognormal'],
              value='uniform', title='Distribution')
```

A Select example

select.py

```
# (continued)

# Add callback to widgets
def callback(attr, old, new):
    if   menu.value == 'uniform': f = random
    elif menu.value == 'normal':  f = normal
    else:                         f = lognormal
    source.data={'x': f(size=N), 'y': f(size=N)}
menu.on_change('value', callback)

# Arrange plots and widgets in layouts
layout = column(menu, plot)

curdoc().add_root(layout)
```



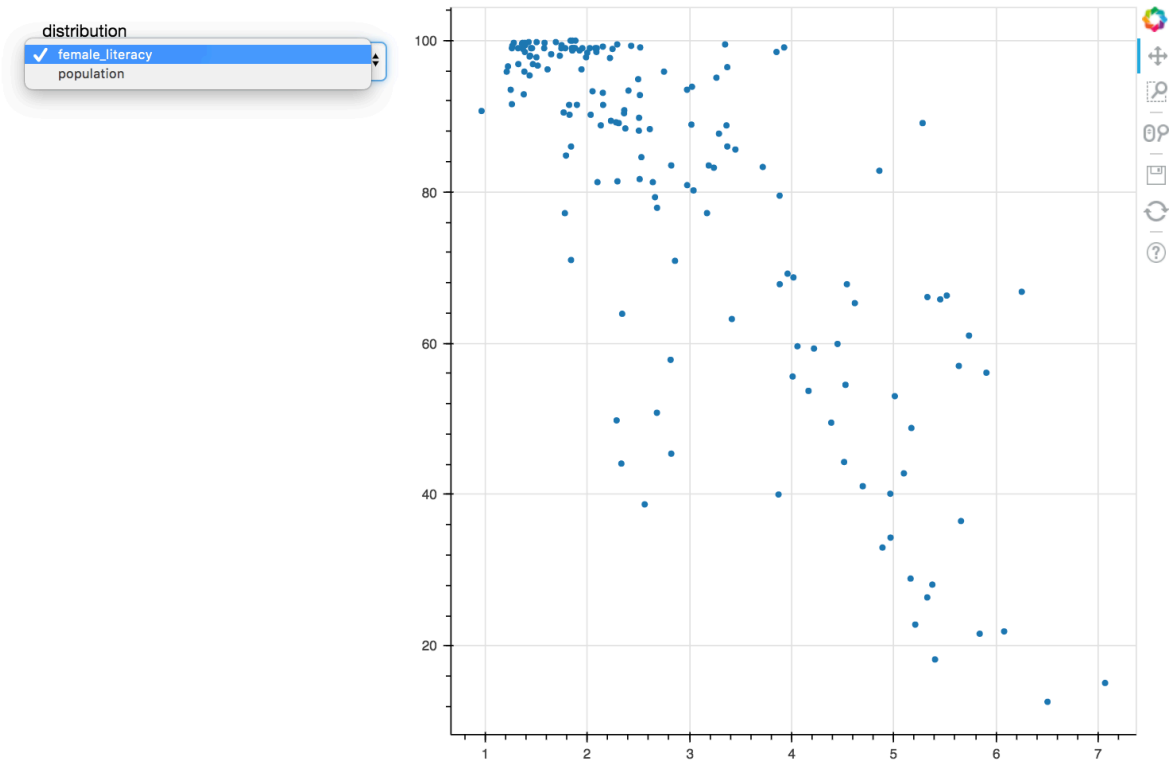
6)
Updating data sources from dropdown callbacks

You'll now learn to update the plot's data using a drop down menu instead of a slider. This would allow users to do things like select between different data sources to view.
The ColumnDataSource `source` has been created for you along with the plot. Your job in this exercise is to add a drop down menu to update the plot's data.
All necessary modules have been imported for you.

- Define a callback function called `update_plot` with the parameters `attr`, `old`, `new`.
    - If the `new` selection is `female_literacy`, update the `y` value of the ColumnDataSource to `female_literacy`. Else, `y` should be `population`.
    - `x` remains `fertility` in both cases.
- Create a dropdown select widget using `Select()`. Specify the parameters `title`, `options`, and `value`.
  The `options` are `'female_literacy'` and `'population'`, while the `value` is `'female_literacy'`.
- Attach the callback to the `'value'` property of `select`. This can be done using `on_change()` and passing in `'value'` and `update_plot`

```
# Perform necessary imports
from bokeh.models import ColumnDataSource, Select

# Create ColumnDataSource: source
source = ColumnDataSource(data={
    'x' : fertility,
    'y' : female_literacy
})

# Create a new plot: plot
plot = figure()

# Add circles to the plot
plot.circle('x', 'y', source=source)

# Define a callback function: update_plot
def update_plot(attr, old, new):
    # If the new Selection is 'female_literacy', update 'y' to female_literacy
    if new == 'female_literacy':
        source.data = {
            'x' : fertility,
            'y' : female_literacy
```

```
    }
# Else, update 'y' to population
else:
    source.data = {
        'x' : fertility,
        'y' : population
    }

# Create a dropdown Select widget: select
select = Select(title="distribution", options=['female_literacy', 'population'],
value='female_literacy')

# Attach the update_plot callback to the 'value' property of select
select.on_change('value', update_plot)

# Create layout and add to current document
layout = row(select, plot)
curdoc().add_root(layout)
```



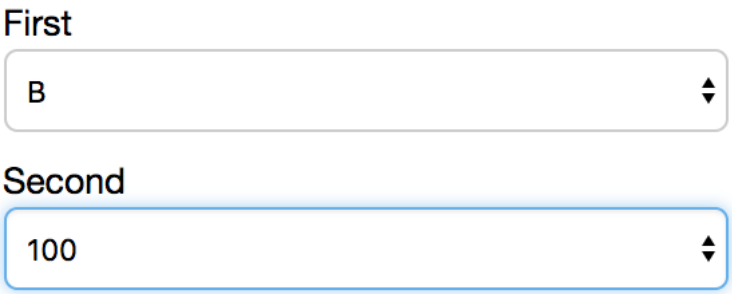
7)
Synchronize two dropdowns

Here, you'll practice using a dropdown callback to update another dropdown's options. This will allow you to customize your applications even further and is a powerful addition to your toolbox.
Your job in this exercise is to create two dropdown select widgets and then define a callback such that one dropdown is used to update the other dropdown.
All modules necessary have been imported.

- Create `select1`, the first dropdown select widget. Specify the parameters `title`, `options`, and `value`.
- Create `select2`, the second dropdown select widget. Specify the parameters `title`, `options`, and `value`.
- Inside the callback function, if `select1` equals `'A'`, update the options of `select2` to `['1', '2', '3']` and set its value to `'1'`.
- If `select1` does not equal `'A'`, update the options of `select2` to `['100', '200', '300']` and set its value to `'100'`.
- Attach the callback to the `'value'` property of `select1`. This can be done using `on_change()` and passing in `'value'` and `callback`

```
# Create two dropdown Select widgets: select1, select2
select1 = Select(title='First', options=['A', 'B'], value='A')
select2 = Select(title='Second', options=['1', '2', '3'], value='1')

# Define a callback function: callback
def callback(attr, old, new):
    # If select1 is 'A'
    if select1.value == 'A':
        # Set select2 options to ['1', '2', '3']
        select2.options = ['1','2','3']

        # Set select2 value to '1'
        select2.value = '1'
    else:
        # Set select2 options to ['100', '200', '300']
        select2.options = ['100', '200', '300']

        # Set select2 value to '100'
        select2.value = '100'

# Attach the callback to the 'value' property of select1
select1.on_change('value', callback)

# Create layout and add to current document
```

```
layout = widgetbox(select1, select2)
curdoc().add_root(layout)
```



Note)

Button callbacks

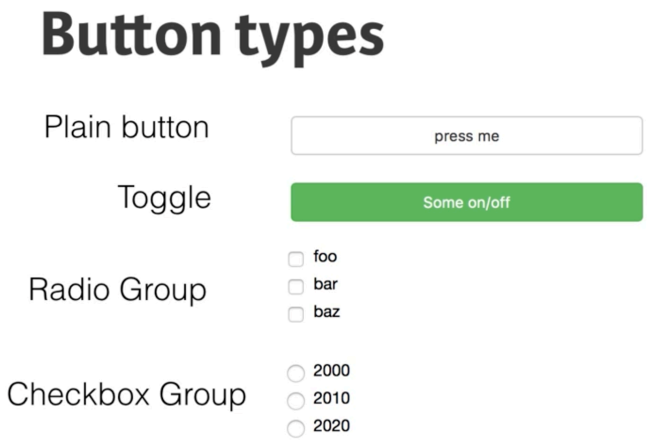
select.py

```
from bokeh.models import Button

button = Button(label='press me')

def update():

    # Do something interesting

button.on_click(update)
```

Button types

select.py

```
from bokeh.models import CheckboxGroup, RadioGroup, Toggle

toggle = Toggle(label='Some on/off', button_type='success')

checkbox = CheckboxGroup(labels=['foo', 'bar', 'baz'])

radio = RadioGroup(labels=['2000', '2010', '2020'])

def callback(active):
    # Active tells which button is active
```



8)
Button widgets

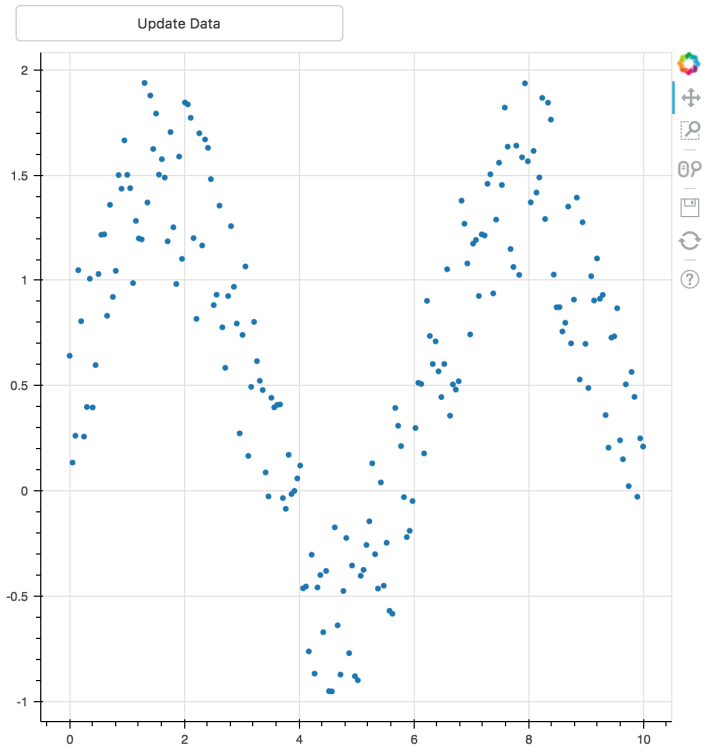
It's time to practice adding buttons to your interactive visualizations. Your job in this exercise is to create a button and use its `on_click()` method to update a plot.
All necessary modules have been imported for you. In addition, the ColumnDataSource with data `x` and `y` as well as the figure have been created for you and are available in the workspace as `source` and `plot`.
When you're done, be sure to interact with the button you just added to your plot, and notice how it updates the data!

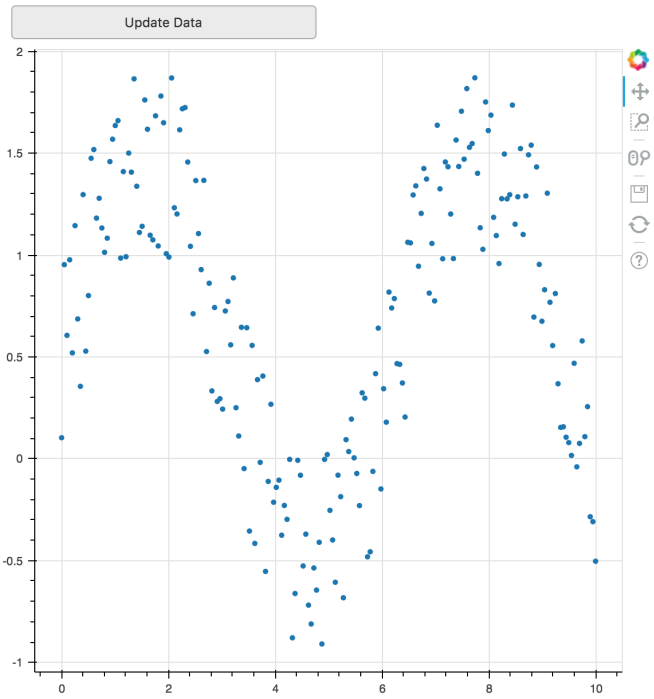
- Create a button called `button` using the function `Button()` with the label `'Update Data'`.
- Define an update callback `update()` with no arguments.

- Compute new y values using the code provided.
- Update the ColumnDataSource data dictionary `source.data` with the new y value.
- Add the update callback to the button using `on_click()`

```
# Create a Button with label 'Update Data'
button = Button(label='Update Data')

# Define an update callback with no arguments: update
def update():

    # Compute new y values: y
    y = np.sin(x) + np.random.random(N)

    # Update the ColumnDataSource data dictionary
    source.data['y'] = y

# Add the update callback to the button
button.on_click(update)
```

```
# Create layout and add to current document
layout = column(widgetbox(button), plot)
curdoc().add_root(layout)
```



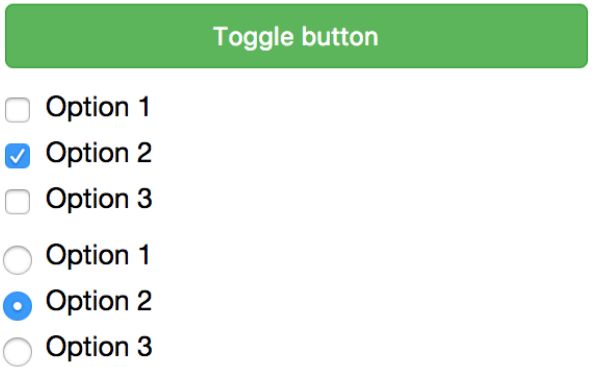


9)
Button styles

You can also get really creative with your `Button` widgets.
In this exercise, you'll practice using CheckboxGroup, RadioGroup, and Toggle to add multiple `Button` widgets with different styles.
`curdoc` and `widgetbox` have already been imported for you.

- Import `CheckboxGroup`, `RadioGroup`, `Toggle` from `bokeh.models`.
- Add a Toggle called `toggle` using the `Toggle()` function with `button_type` `'success'` and `label` `'Toggle button'`.
- Add a CheckboxGroup called `checkbox` using the `CheckboxGroup()`function with `labels=['Option 1', 'Option 2', 'Option 3']`.
- Add a RadioGroup called `radio` using the `RadioGroup()` function with `labels=['Option 1', 'Option 2', 'Option 3']`.
- Add the widgetbox containing the Toggle `toggle`, CheckboxGroup `checkbox`, and RadioGroup `radio` to the current document

```
# Import CheckboxGroup, RadioGroup, Toggle from bokeh.models
from bokeh.models import CheckboxGroup,RadioGroup,Toggle

# Add a Toggle: toggle
toggle = Toggle(button_type='success',label='Toggle button')

# Add a CheckboxGroup: checkbox
checkbox = CheckboxGroup(labels=['Option 1','Option 2','Option 3'])

# Add a RadioGroup: radio
radio = RadioGroup(labels=['Option 1','Option 2','Option 3'])

# Add widgetbox(toggle, checkbox, radio) to the current document
curdoc().add_root(widgetbox(toggle,checkbox,radio))
```



Bokeh Application Hosting

https://anaconda.org

Chapter 4)

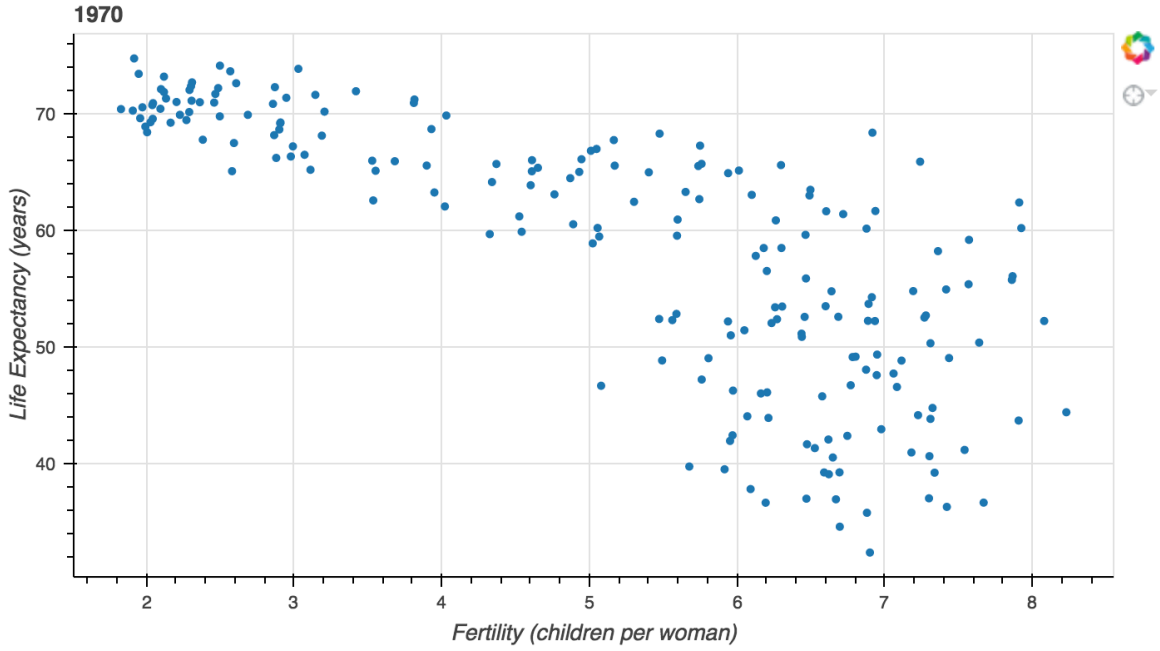
1)
Some exploratory plots of the data

Here, you'll continue your Exploratory Data Analysis by making a simple plot of Life Expectancy vs Fertility for the year 1970.
Your job is to import the relevant Bokeh modules and then prepare a `ColumnDataSource` object with the `fertility`, `life` and `Country` columns, where you only select the rows with the index value 1970.
Remember, as with the figures you generated in previous chapters, you can interact with your figures here with a variety of tools.

- Import `output_file` and `show` from `bokeh.io`, `figure` from `bokeh.plotting`, and `HoverTool` and `ColumnDataSource` from `bokeh.models`.
- Make a `ColumnDataSource` called `source` with `x` set to the `fertility` column, `y` set to the `life` column and `country` set to the `Country` column. For all columns, select the rows with index value `1970`. This can be done using `data.loc[1970].column_name`

```
# Perform necessary imports
from bokeh.io import output_file, show
from bokeh.plotting import figure
from bokeh.models import HoverTool, ColumnDataSource

# Make the ColumnDataSource: source
source = ColumnDataSource(data={
    'x'       : data.loc[1970].fertility,
    'y'       : data.loc[1970].life,
    'country' : data.loc[1970].Country
})

# Create the figure: p
p = figure(title='1970', x_axis_label='Fertility (children per woman)',
y_axis_label='Life Expectancy (years)',
           plot_height=400, plot_width=700,
           tools=[HoverTool(tooltips='@country')])

# Add a circle glyph to the figure p
p.circle(x='x', y='y', source=source)

# Output the file and show the figure
output_file('gapminder.html')
```

show(p)

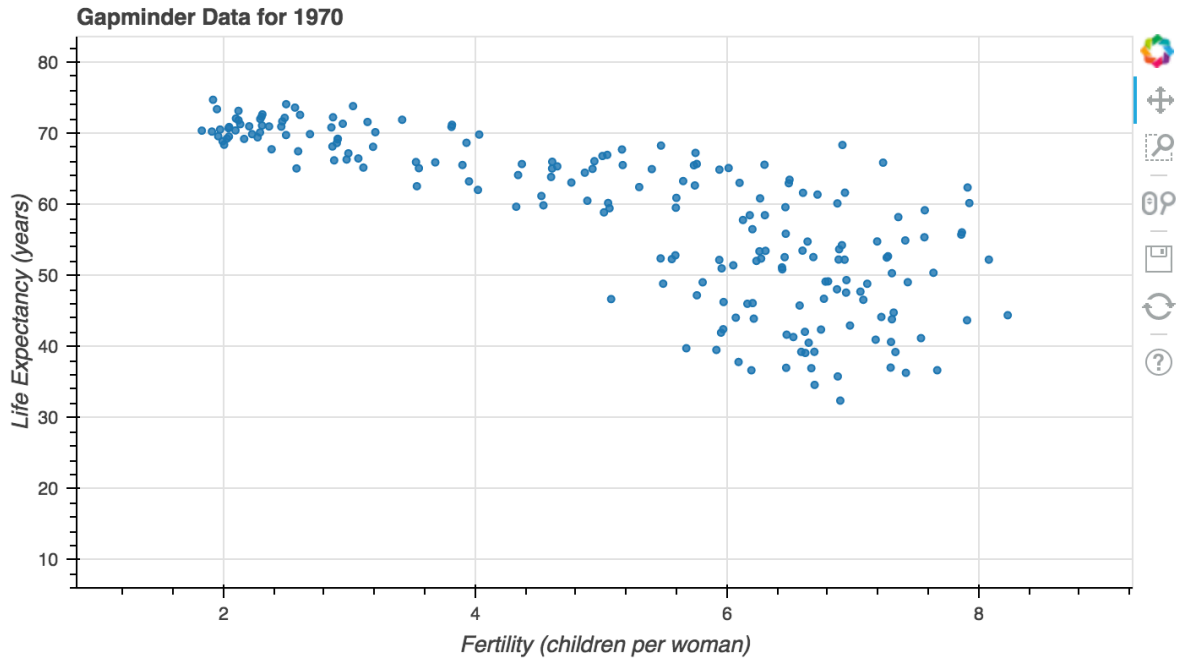


2)
Beginning with just a plot

Let's get started on the Gapminder app. Your job is to make the `ColumnDataSource` object, prepare the plot, and add circles for Life expectancy vs Fertility. You'll also set x and y ranges for the axes.
As in the previous chapter, the DataCamp environment executes the `bokeh serve` command to run the app for you. When you hit 'Submit Answer', you'll see in the IPython Shell that `bokeh serve script.py`gets called to run the app. This is something to keep in mind when you are creating your own interactive visualizations outside of the DataCamp environment.

- Make a `ColumnDataSource` object called `source` with `'x'`, `'y'`, `'country'`, `'pop'` and `'region'` keys. The Pandas selections are provided for you.
- Save the minimum and maximum values of the life expectancy column `data.life` as `ymin` and `ymax`. As a guide, you can refer to the way we saved the minimum and maximum values of the fertility column `data.fertility` as `xmin` and `xmax`.
- Create a plot called `plot()` by specifying the `title`, setting `plot_height` to `400`, `plot_width` to `700`, and adding the `x_range` and `y_range` parameters.
- Add circle glyphs to the plot. Specify an `fill_alpha` of `0.8` and `source=source`.

```
# Import the necessary modules
from bokeh.io import curdoc
from bokeh.models import ColumnDataSource
from bokeh.plotting import figure

# Make the ColumnDataSource: source
source = ColumnDataSource(data={
    'x'       : data.loc[1970].fertility,
    'y'       : data.loc[1970].life,
    'country'      : data.loc[1970].Country,
    'pop'      : (data.loc[1970].population / 20000000) + 2,
    'region'      : data.loc[1970].region,
})

# Save the minimum and maximum values of the fertility column: xmin, xmax
xmin, xmax = min(data.fertility), max(data.fertility)

# Save the minimum and maximum values of the life expectancy column: ymin,
ymax
ymin, ymax = min(data.life), max(data.life)

# Create the figure: plot
plot = figure(title='Gapminder Data for 1970', plot_height=400,
plot_width=700,
            x_range=(xmin, xmax), y_range=(ymin, ymax))

# Add circle glyphs to the plot
plot.circle(x='x', y='y', fill_alpha=0.8, source=source)

# Set the x-axis label
plot.xaxis.axis_label ='Fertility (children per woman)'

# Set the y-axis label
plot.yaxis.axis_label = 'Life Expectancy (years)'

# Add the plot to the current document and add a title
curdoc().add_root(plot)
curdoc().title = 'Gapminder'
```



3)

Enhancing the plot with some shading

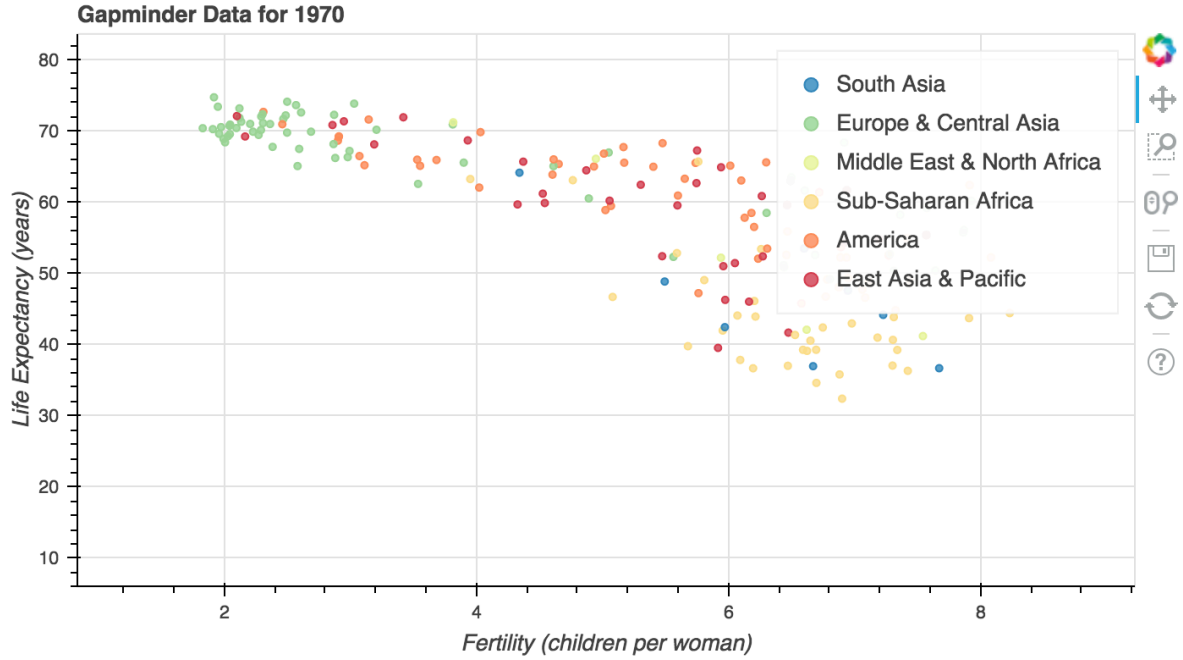
Now that you have the base plot ready, you can enhance it by coloring each circle glyph by continent.
Your job is to make a list of the unique regions from the data frame, prepare a `ColorMapper`, and add it to the circle glyph.

- Make a list of the unique values from the `region` column. You can use the `unique()` and `tolist()` methods on `data.region` to do this.
- Import `CategoricalColorMapper` from `bokeh.models` and the `Spectral6` palette from `bokeh.palettes`.
- Use the `CategoricalColorMapper()` function to make a color mapper called `color_mapper` with `factors=regions_list` and `palette=Spectral6`.
- Add the color mapper to the circle glyph as a dictionary with `dict(field='region', transform=color_mapper)` as the argument passed to the `color` parameter of `plot.circle()`. Also set the `legend` parameter to be the `'region'`.
- Set the `legend.location` attribute of `plot` to `'top_right'`

# Make a list of the unique values from the region column: regions_list
regions_list = data.region.unique().tolist()

# Import CategoricalColorMapper from bokeh.models and the Spectral6 palette from bokeh.palettes
from bokeh.models import CategoricalColorMapper
from bokeh.palettes import Spectral6

```
# Make a color mapper: color_mapper
color_mapper = CategoricalColorMapper(factors=regions_list,
palette=Spectral6)

# Add the color mapper to the circle glyph
plot.circle(x='x', y='y', fill_alpha=0.8, source=source,
            color=dict(field='region', transform=color_mapper), legend='region')

# Set the legend.location attribute of the plot to 'top_right'
plot.legend.location = 'top_right'

# Add the plot to the current document and add the title
curdoc().add_root(plot)
curdoc().title = 'Gapminder'
```



4)

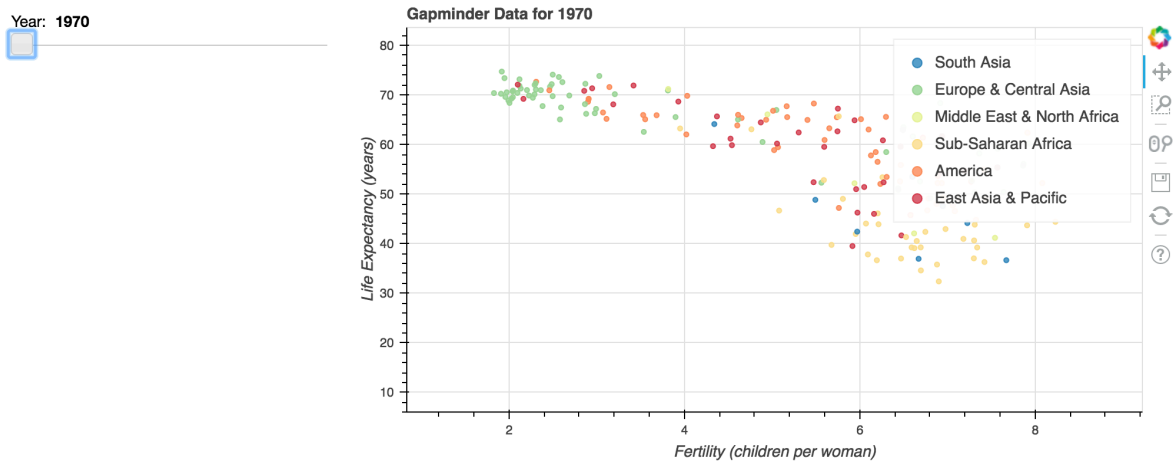
Adding a slider to vary the year

Until now, we've been plotting data only for 1970. In this exercise, you'll add a slider to your plot to change the year being plotted. To do this, you'll create an `update_plot()` function and associate it with a slider to select values between 1970 and 2010.

After you are done, you may have to scroll to the right to view the entire plot. As you play around with the slider, notice that the title of the plot is not updated along with the year. This is something you'll fix in the next exercise!

- Import the `widgetbox` and `row` functions from `bokeh.layouts`, and the `Slider` function from `bokeh.models`.
- Define the `update_plot` callback function with parameters `attr`, `old` and `new`.
- Set the `yr` name to `slider.value` and set `source.data = new_data`.
- Make a slider object called `slider` using the `Slider()` function with a `start` year of `1970`, `end` year of `2010`, `step` of `1`, `value` of `1970`, and `title` of `'Year'`.
- Attach the callback to the `'value'` property of slider. This can be done using `on_change()` and passing in `'value'` and `update_plot`.
- Make a row `layout` of `widgetbox(slider)` and `plot` and add it to the current document.

```
# Import the necessary modules
from bokeh.layouts import widgetbox, row
from bokeh.models import Slider

# Define the callback function: update_plot
def update_plot(attr, old, new):
    # set the `yr` name to `slider.value` and `source.data = new_data`
    yr = slider.value
    new_data = {
        'x'       : data.loc[yr].fertility,
        'y'       : data.loc[yr].life,
        'country' : data.loc[yr].Country,
        'pop'     : (data.loc[yr].population / 20000000) + 2,
        'region'  : data.loc[yr].region,
    }
    source.data = new_data


# Make a slider object: slider
slider = Slider(start=1970,end=2010,step=1,value=1970,title='Year')

# Attach the callback to the 'value' property of slider
slider.on_change('value',update_plot)

# Make a row layout of widgetbox(slider) and plot and add it to the current
document
layout = row(widgetbox(slider), plot)
curdoc().add_root(layout)
```



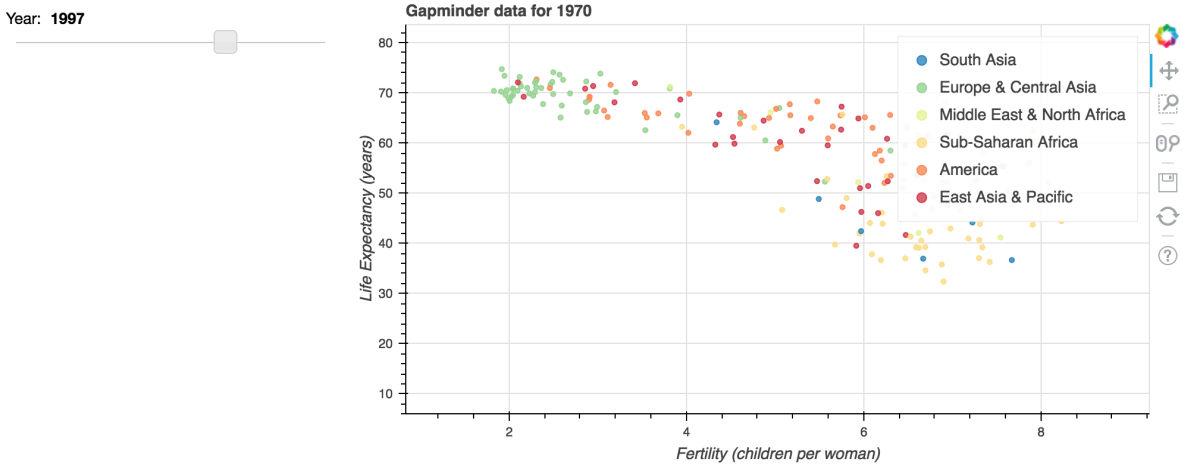
5)
Customizing based on user input

Remember how in the plot from the previous exercise, the title did not update along with the slider? In this exercise, you'll fix this.
In Python, you can format strings by specifying placeholders with the `%` keyword. For example, if you have a string `company = 'DataCamp'`, you can use `print('%s' % company)` to print `DataCamp`. Placeholders are useful when you are printing values that are not static, such as the value of the year slider. You can specify a placeholder for a number with `%d`. Here, when you're updating the plot title inside your callback function, you should make use of a placeholder so that the year displayed is in accordance with the value of the year slider.
In addition to updating the plot title, you'll also create the callback function and slider as you did in the previous exercise, so you get a chance to practice these concepts further.
All necessary modules have been imported for you, and as in the previous exercise, you may have to scroll to the right to view the entire figure.

- Define the `update_plot` callback function with parameters `attr`, `old` and `new`.
- Inside `update_plot()`, assign the value of the slider, `slider.value`, to `yr` and set `source.data = new_data`.
- Inside `update_plot()`, specify `plot.title.text` to update the plot title and add it to the figure. You want the plot to update based on the value of the slider, which you have assigned above to `yr`. Make use of the placeholder syntax provided for you.
- Make a slider object called `slider` using the `Slider()` function with a `start` year of `1970`, `end` year of `2010`, `step` of `1`, `value` of `1970`, and `title` of `'Year'`.
- Attach the callback to the `'value'` property of slider. This can be done using `on_change()` and passing in `'value'` and `update_plot`.

# Define the callback function: update_plot
def update_plot(attr, old, new):

```
    # Assign the value of the slider: yr
    yr = slider.value
    # Set new_data
    new_data = {
        'x'       : data.loc[yr].fertility,
        'y'       : data.loc[yr].life,
        'country' : data.loc[yr].Country,
        'pop'     : (data.loc[yr].population / 20000000) + 2,
        'region'  : data.loc[yr].region,
    }
    # Assign new_data to: source.data
    source.data = new_data

    # Add title to figure: plot.title.text
    plot.title.text = 'Gapminder data for %d'%yr

# Make a slider object: slider
slider = Slider(start=1970,end=2010,step=1,value=1970,title='Year')

# Attach the callback to the 'value' property of slider
slider.on_change('value',update_plot)

# Make a row layout of widgetbox(slider) and plot and add it to the current
document
layout = row(widgetbox(slider), plot)
curdoc().add_root(layout)
```



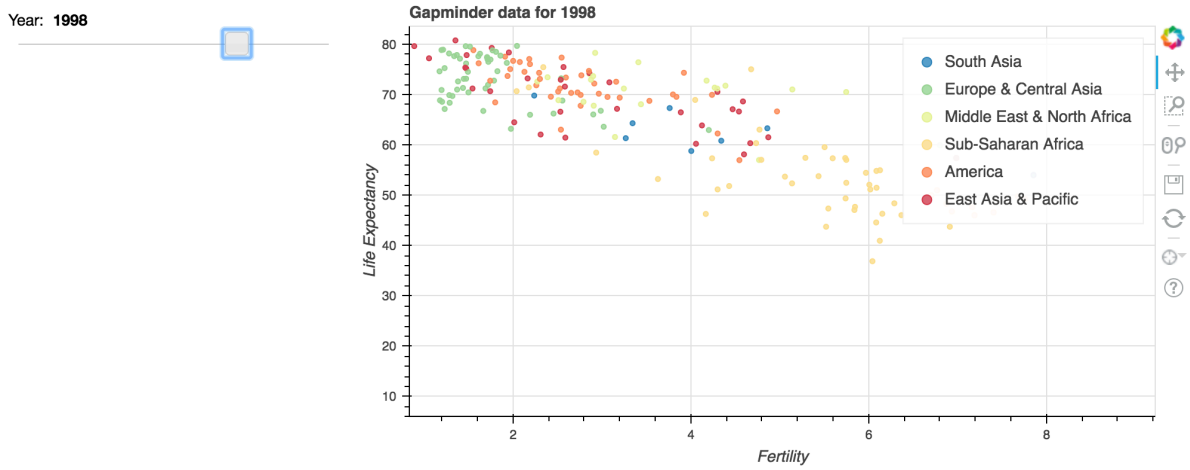
6)
Adding a hover tool

In this exercise, you'll practice adding a hover tool to drill down into data column values and display more detailed information about each scatter point.

After you're done, experiment with the hover tool and see how it displays the name of the country when your mouse hovers over a point!
The figure and slider have been created for you and are available in the workspace as `plot` and `slider`.

- Import `HoverTool` from `bokeh.models`.
- Create a HoverTool object called `hover` with `tooltips=[('Country', '@country')]`.
- Add the HoverTool object you created to the `plot` using `add_tools()`.
- Create a row `layout` using `widgetbox(slider)` and `plot`.

```
# Import HoverTool from bokeh.models
from bokeh.models import HoverTool

# Create a HoverTool: hover
hover = HoverTool(tooltips=[('Country','@country')])

# Add the HoverTool to the plot
plot.add_tools(hover)
# Create layout: layout
layout = row(widgetbox(slider),plot)

# Add layout to current document
curdoc().add_root(layout)
```



7)
Adding dropdowns to the app

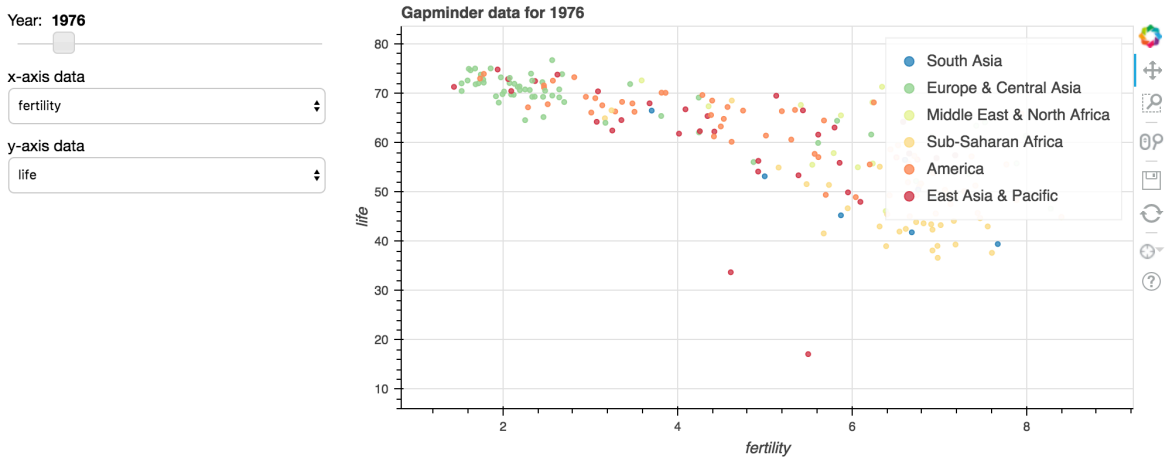
As a final step in enhancing your application, in this exercise you'll add dropdowns for interactively selecting different data features. In combination with the hover tool you added in the previous exercise, as well as the slider to change the year, you'll have a powerful app that allows you to interactively and quickly extract some great insights from the dataset!

All necessary modules have been imported, and the previous code you wrote is taken care off. In the provided sample code, the dropdown for selecting features on the x-axis has been added for you. Using this as a reference, your job in this final exercise is to add a dropdown menu for selecting features on the y-axis.
Take a moment, after you are done, to enjoy exploring the visualization by experimenting with the hover tools, sliders, and dropdown menus that you have learned how to implement in this course.

- Inside the `update_plot()` callback function, read in the current value of the `y` dropdown, `y_select`.
- Use `plot.yaxis.axis_label` to label the y-axis as `y`.
- Set the `start` and `end` range of the y-axis of `plot`.
- Specify the parameters of the `y_select` dropdown widget: `options`, `value`, and `title`. The default `value` should be `'life'`.
- Attach the callback to the `'value'` property of `y_select`. This can be done using `on_change()` and passing in `'value'` and `update_plot`.

```
# Define the callback: update_plot
def update_plot(attr, old, new):
    # Read the current value off the slider and 2 dropdowns: yr, x, y
    yr = slider.value
    x = x_select.value
    y = y_select.value
    # Label axes of plot
    plot.xaxis.axis_label = x
    plot.yaxis.axis_label = y
    # Set new_data
    new_data = {
        'x'       : data.loc[yr][x],
        'y'       : data.loc[yr][y],
        'country' : data.loc[yr].Country,
        'pop'     : (data.loc[yr].population / 20000000) + 2,
        'region'  : data.loc[yr].region,
    }
    # Assign new_data to source.data
    source.data = new_data

    # Set the range of all axes
    plot.x_range.start = min(data[x])
    plot.x_range.end = max(data[x])
    plot.y_range.start = min(data[y])
    plot.y_range.end = max(data[y])

    # Add title to plot
    plot.title.text = 'Gapminder data for %d' % yr
```

```
# Create a dropdown slider widget: slider
slider = Slider(start=1970, end=2010, step=1, value=1970, title='Year')

# Attach the callback to the 'value' property of slider
slider.on_change('value', update_plot)

# Create a dropdown Select widget for the x data: x_select
x_select = Select(
    options=['fertility', 'life', 'child_mortality', 'gdp'],
    value='fertility',
    title='x-axis data'
)

# Attach the update_plot callback to the 'value' property of x_select
x_select.on_change('value', update_plot)

# Create a dropdown Select widget for the y data: y_select
y_select = Select(
    options=['fertility', 'life', 'child_mortality', 'gdp'],
    value='life',
    title='y-axis data'
)

# Attach the update_plot callback to the 'value' property of y_select
y_select.on_change('value', update_plot)

# Create layout and add to current document
layout = row(widgetbox(slider, x_select, y_select), plot)
curdoc().add_root(layout)
```



8)