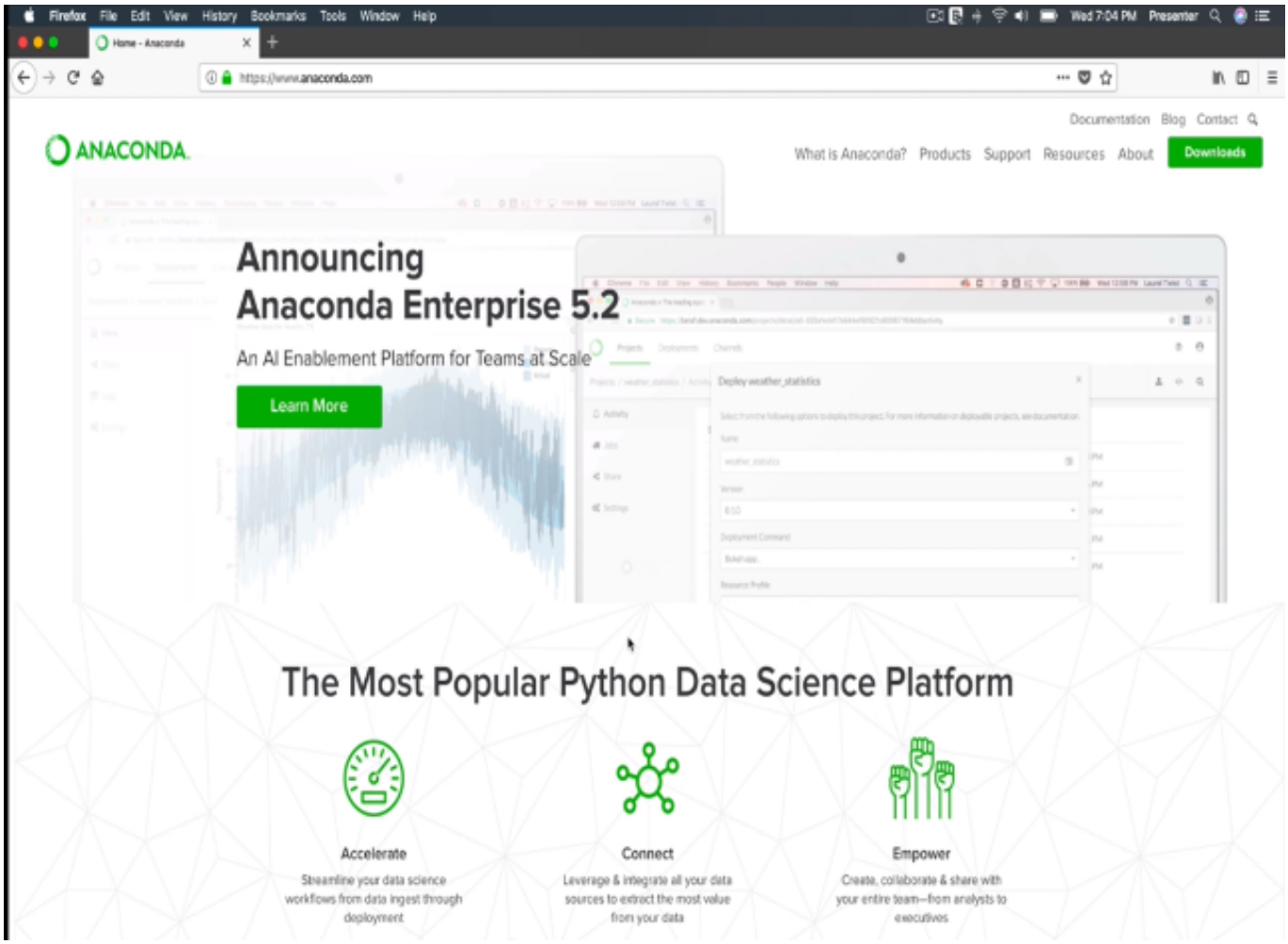
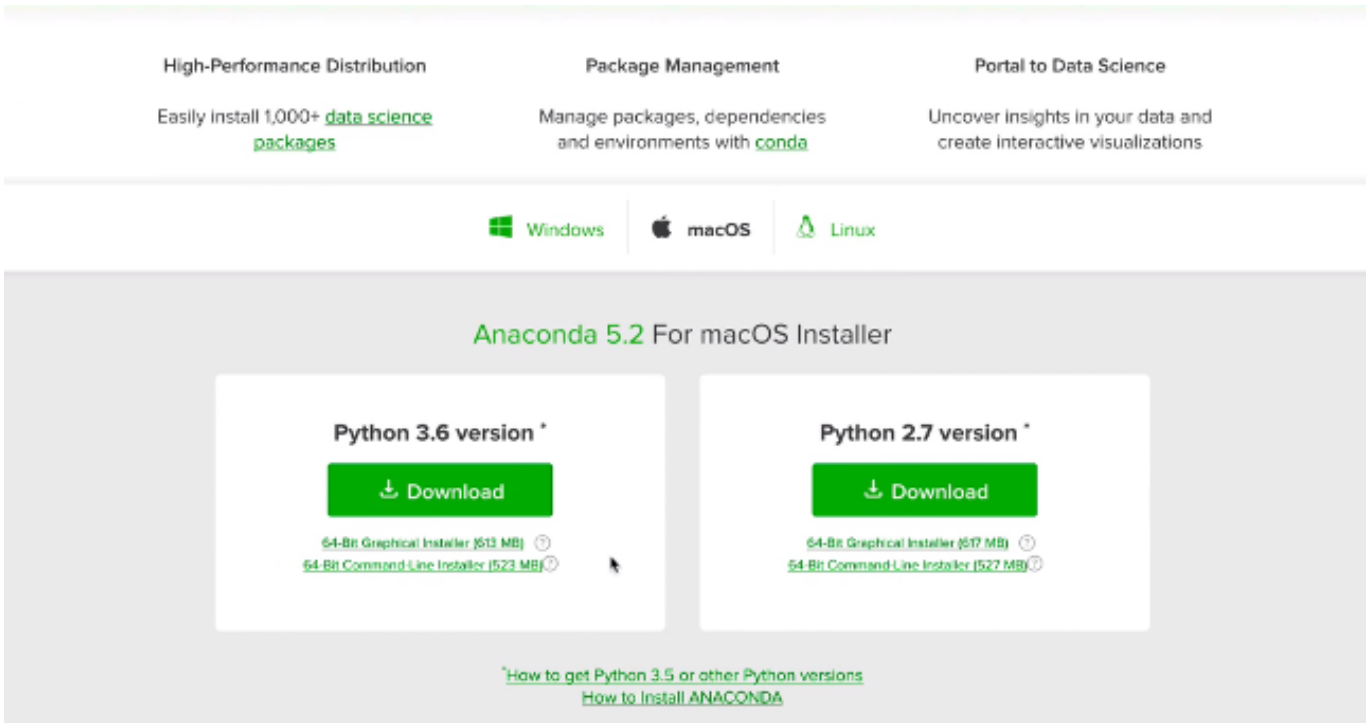
In this video, lets get started getting our entire development environment setup. This includes all dependencies and the dependency manager. We are going to use this really wonderful tool called **Anaconda** (www.anaconda.com) to manage all our **Python dependencies**.



Anaconda comes with a wonderful UI that we can download and use. Anaconda contains the dependencies into environments, which can be exported and imported easily. For our projects, we will provide you with a ready environment which you can import and use. Anaconda, under the hood, will take care of this for us.

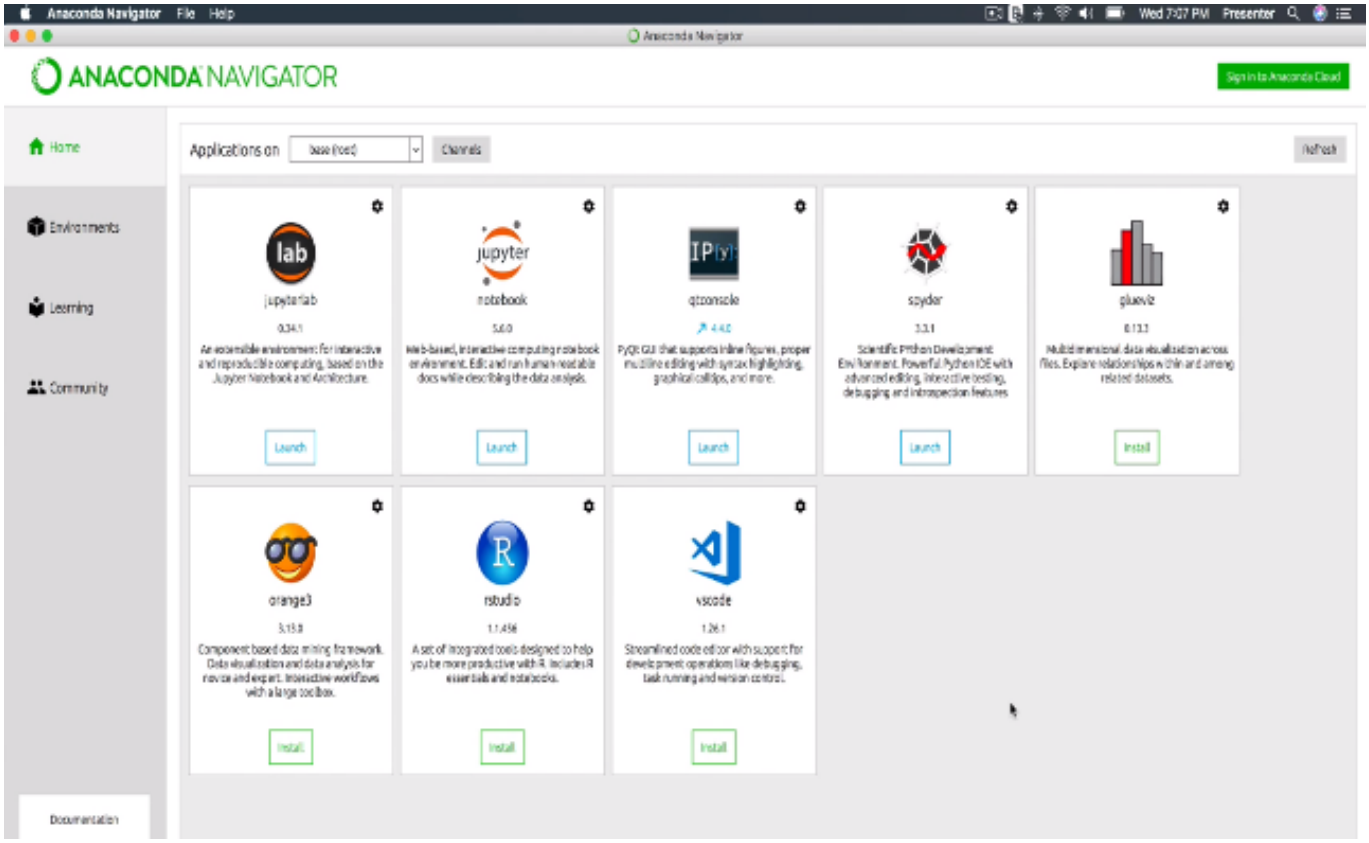
Lets go to www.anaconda.com and click on **Downloads**.

Now download the correct download for your operating system setup.

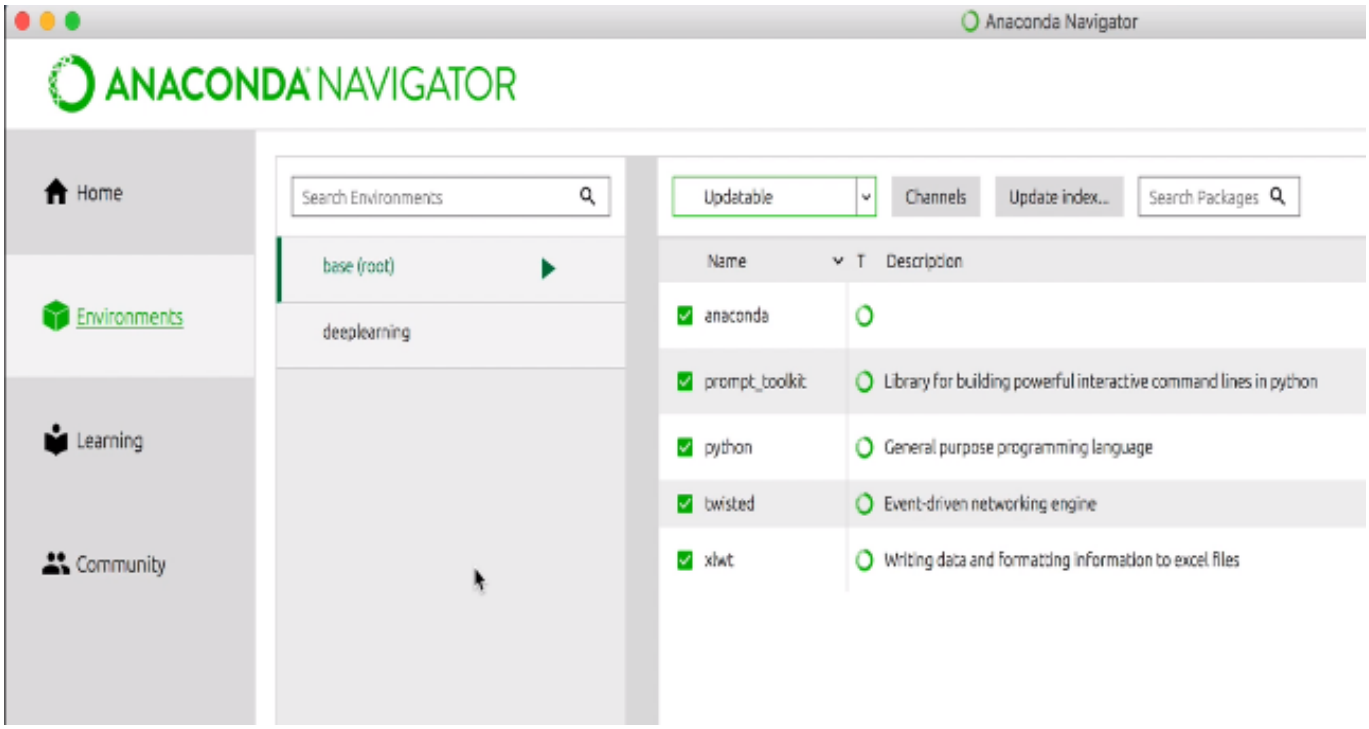


Make sure you pick the **Python 3.6** version. Click Download. It will download the correct package (macOS) or executable (Window, Linux). Lets proceed with the install to install Anaconda.

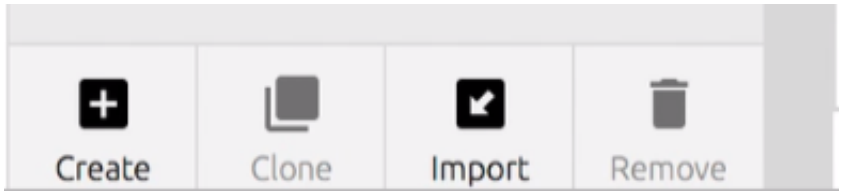
We get this application called **Anaconda Navigator**. When you run it, it would look something like:



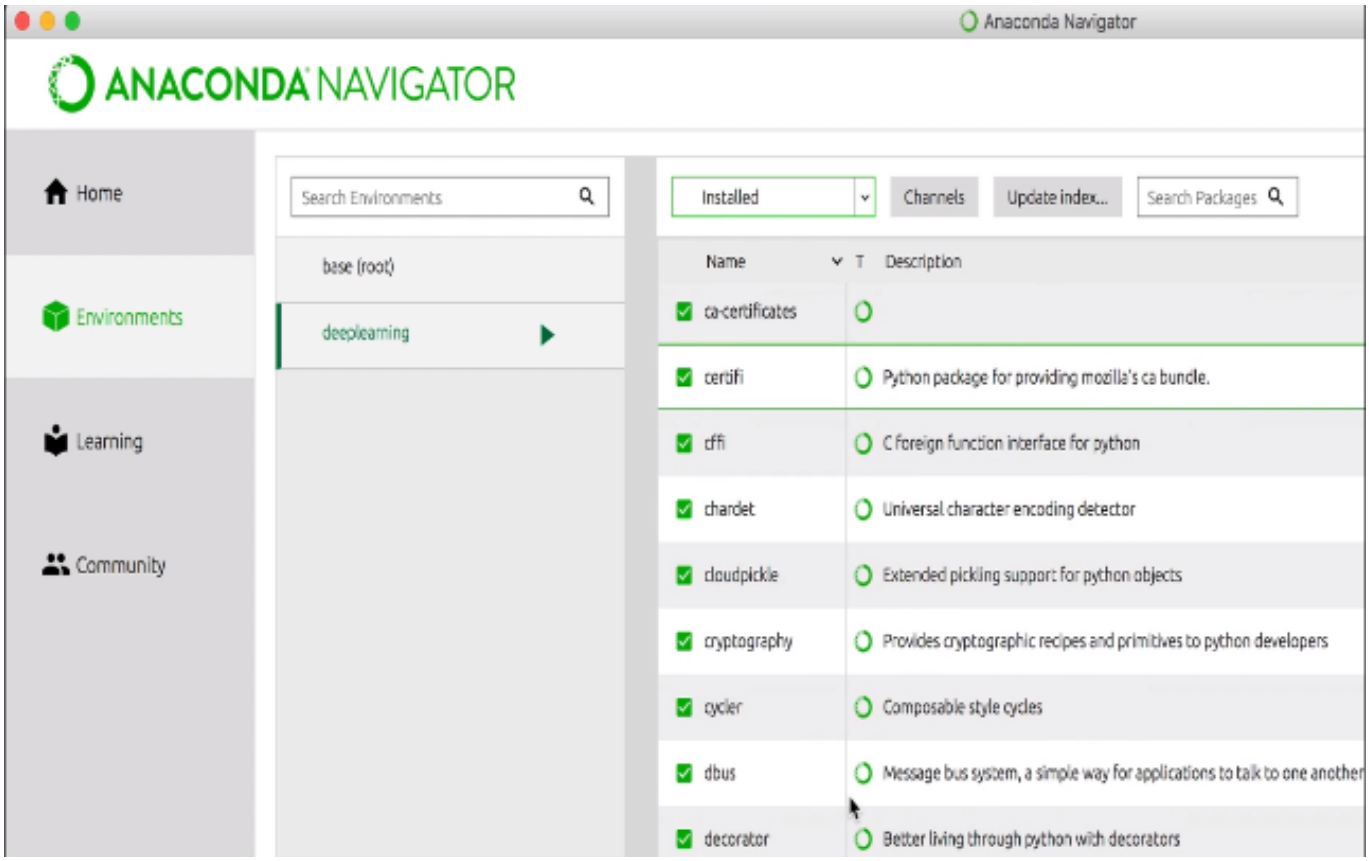
Lets click on **Environments**.



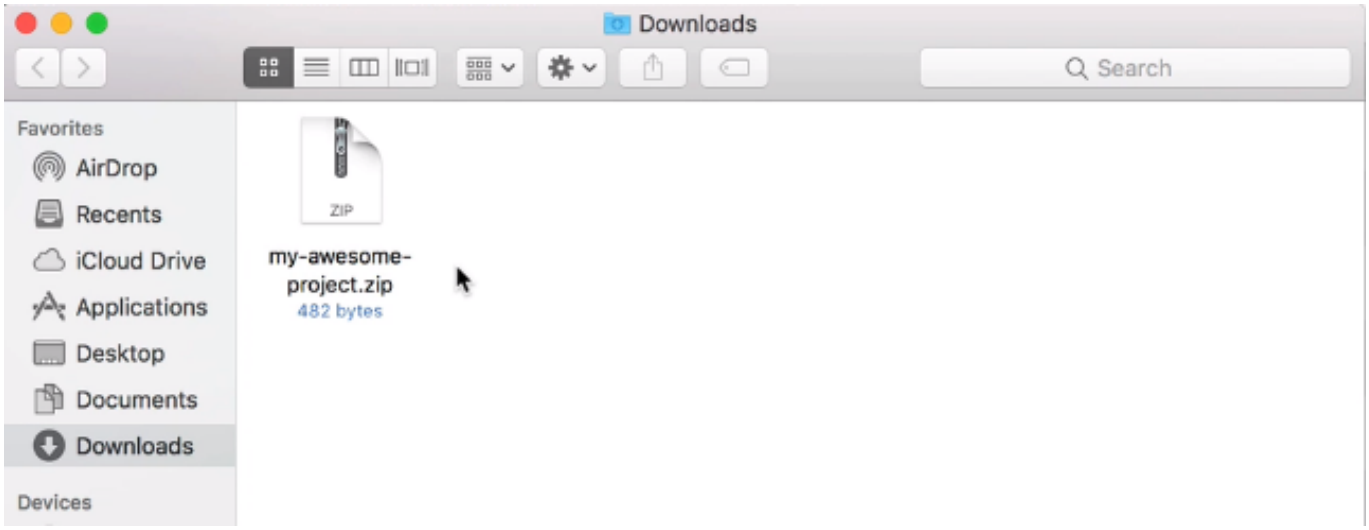
We already have an environment for **deeplearning**. The base(root) environment is present by default, which is seldom used directly. Different environments are created for different projects since the dependencies could be unique to each project. At the bottom there are buttons to create or import environments.



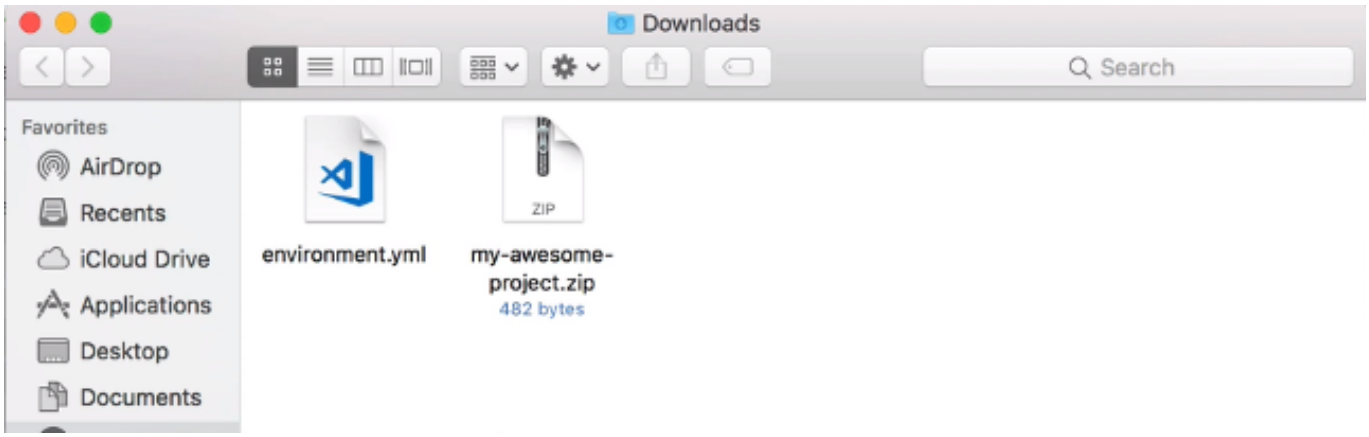
To change the environment to *deeplearning*, we simply click on it.



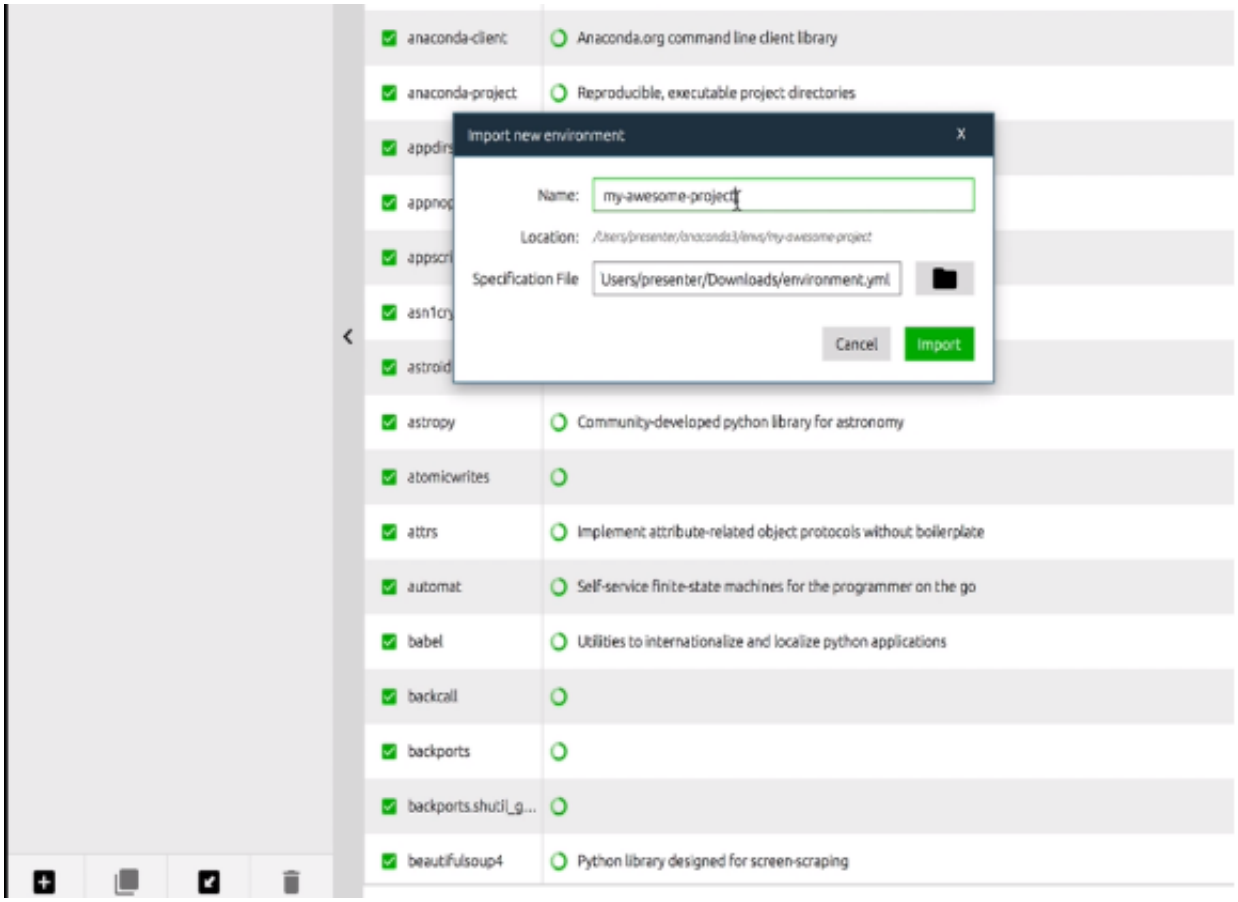
To see the packages installed inside *deeplearning*, we choose Installed from the drop down above. You could create any environments you want. However, as mentioned above, we will be providing you with a file **(my-awesome-project.zip**) that you can simply import. This will create an environment and download and install all required packages on your computer.



Extracting this file yields a single file – *environment.yml*, as shown below.

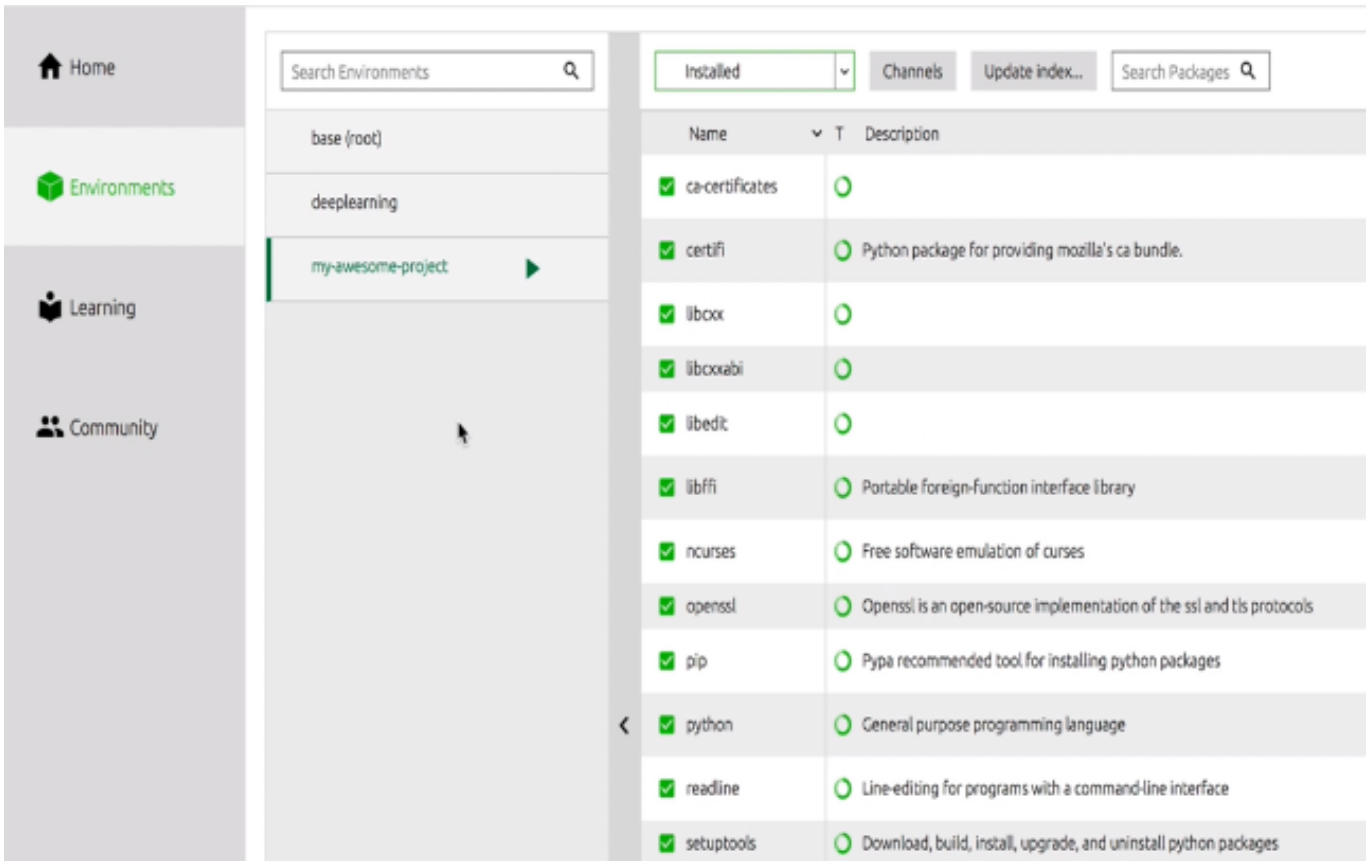


This file contains the names and version numbers of all Python packages in this environment. Importing this file is easy to do. Click on the import icon (shown above) and choose the *environment.yml*, extracted above.

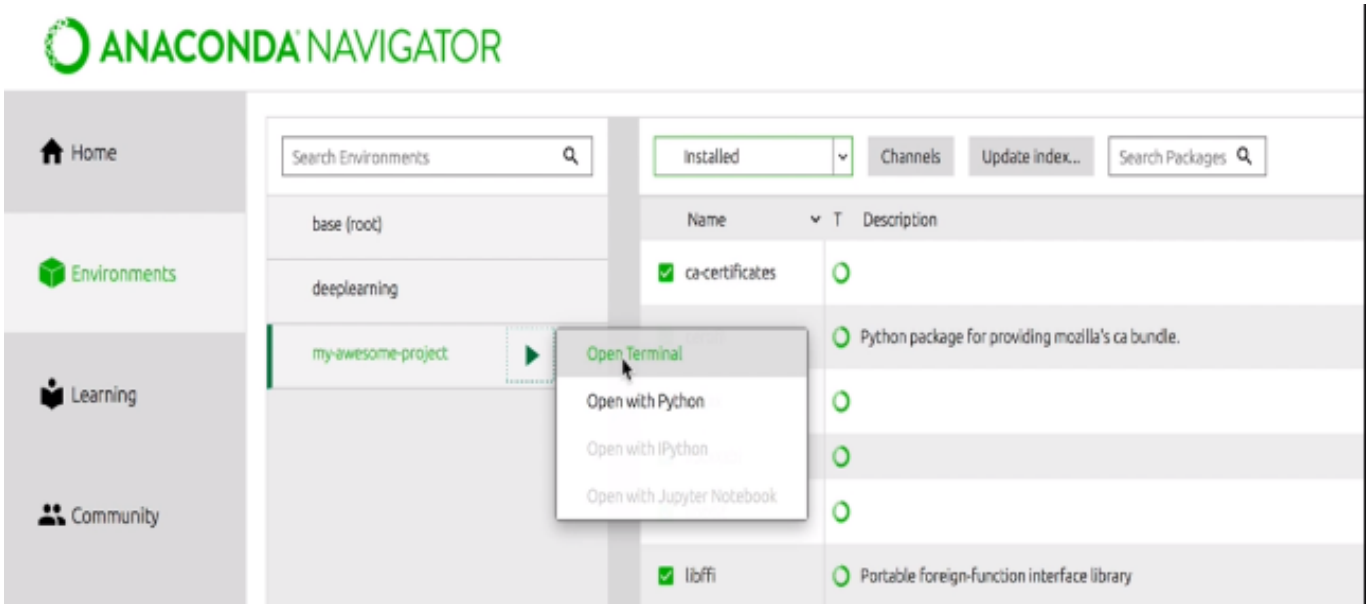


Click Import. In a few minutes, the environment with all packages will be downloaded and installed.

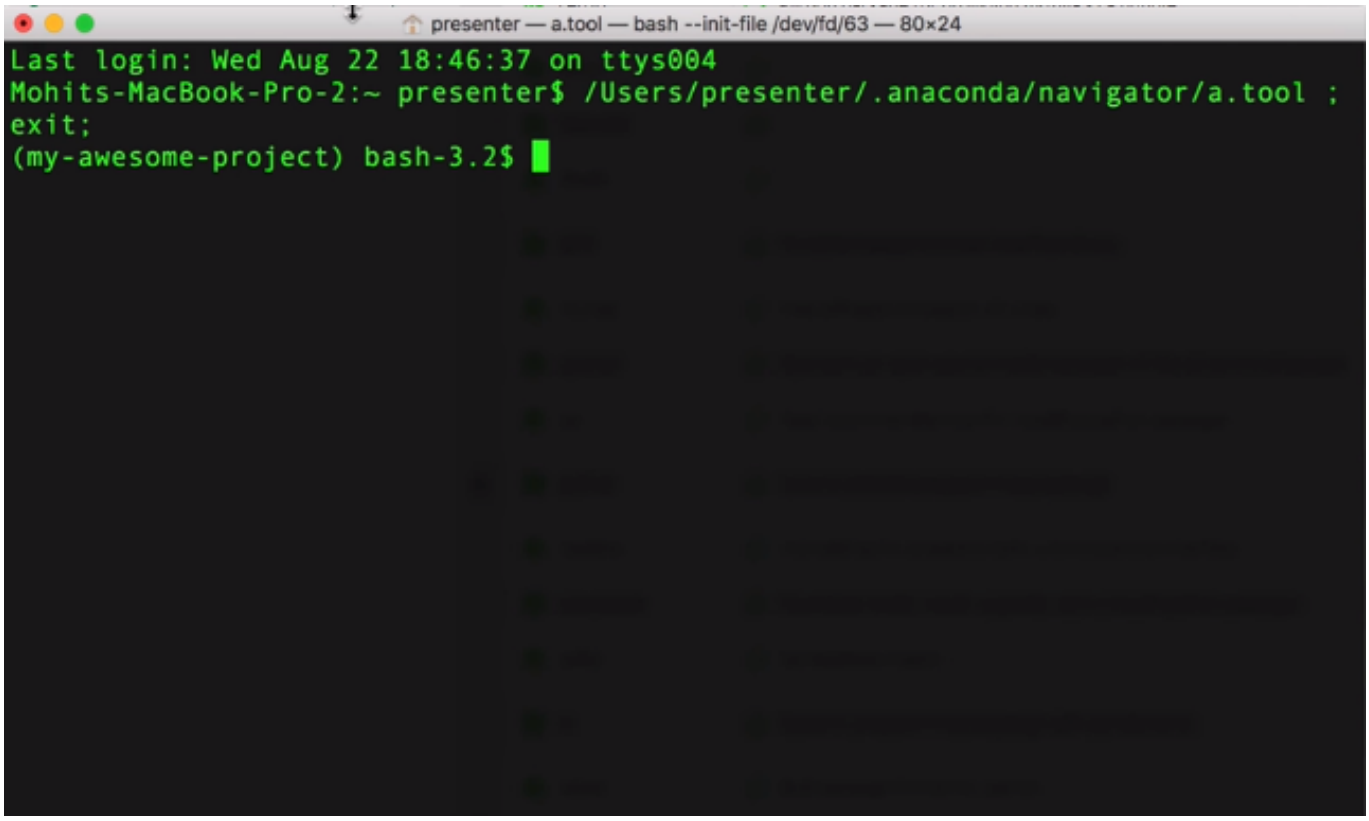
Here it is.



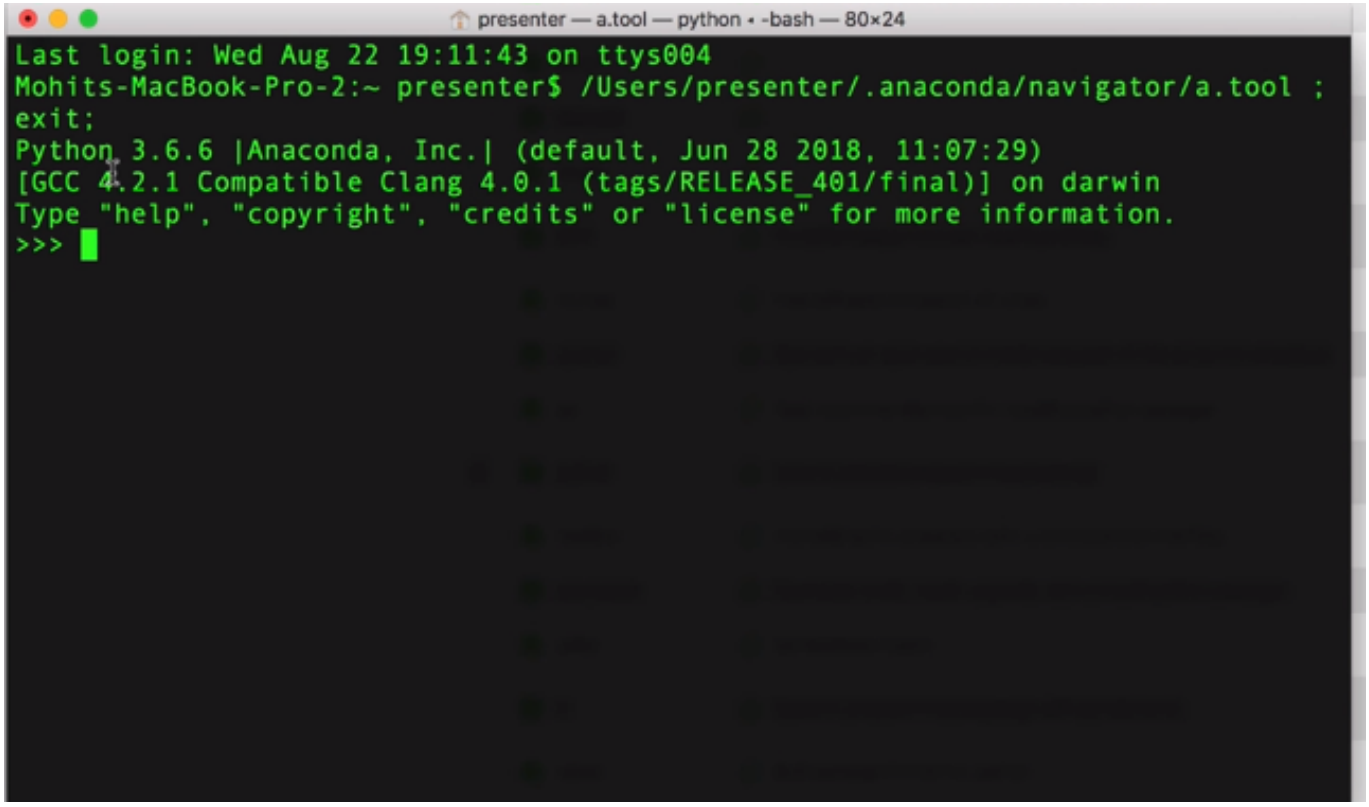
Lets click on the little green arrow next to the environment name. We see 2 menu options. Lets first click on **“Open Terminal”**.



We get a command prompt (OS specific), that is already setup to use the environment we just created.



Lets close this and click on the **“Open With Python”** menu option show above. It will open the Python interpreter with the correct version.



I will be able to access any package installed into my environment in this Python environment.

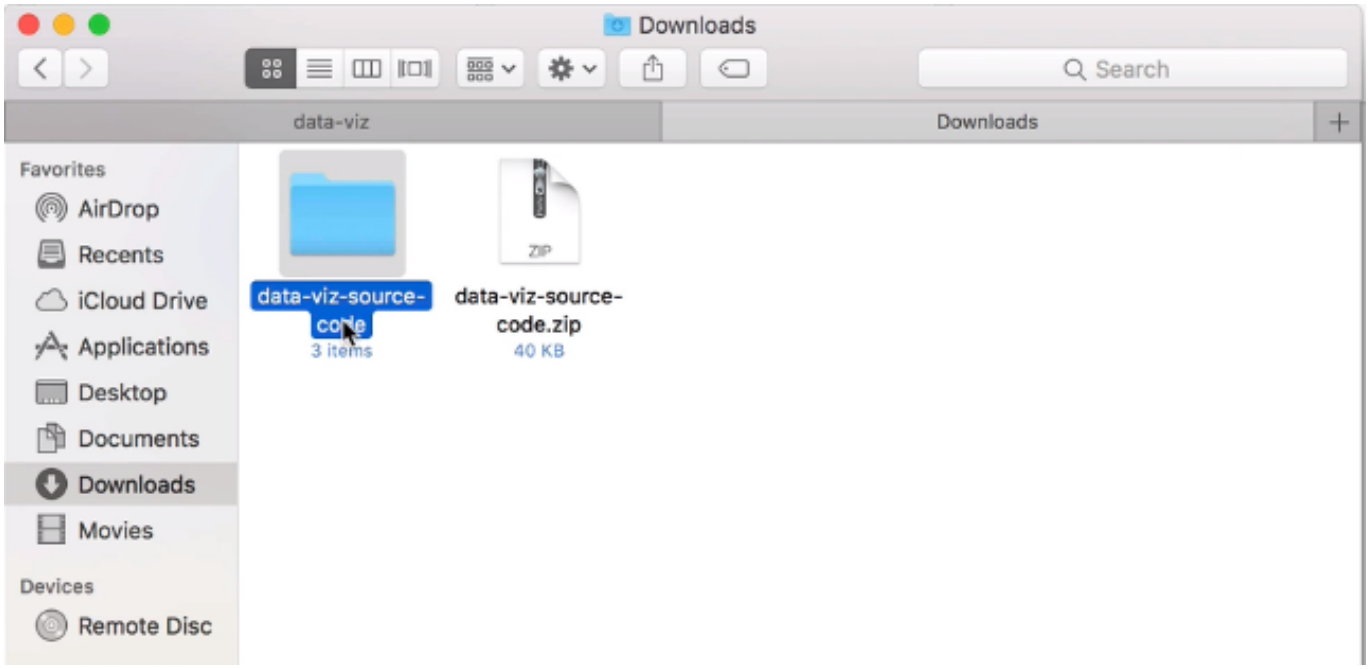
Summary

This was a quick overview of how to install Anaconda and how we can import environments. We'll be using this going forward as it is a great tool to manage dependencies in a simple way. All we have to do is import the environment file and Anaconda will point Python to the correct folder for all dependencies.

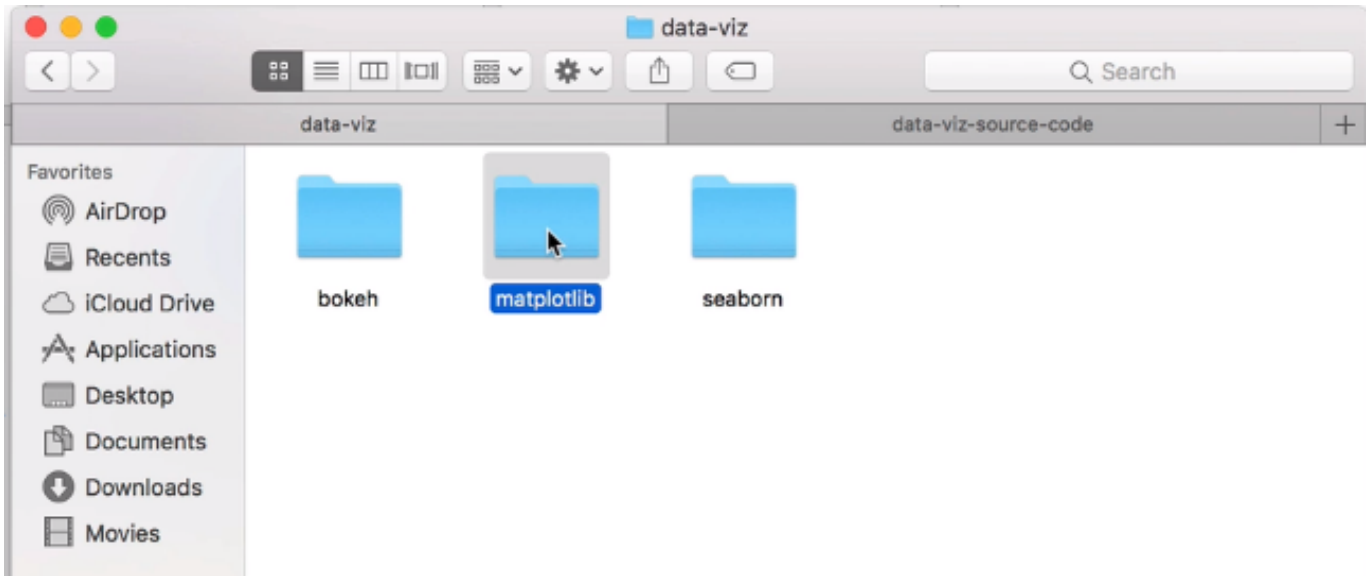
In this video, we are going to be looking for 2 of the more common plots – the column and bar plots. There is a very small difference between the two and *matplotlib* gives us a way to use an almost identical API for both these plots.

First download the environment file and import it. We showed you how to do that in the last video. Next make sure you have the correct environment – **data-viz** (can be selected from the drop-down).

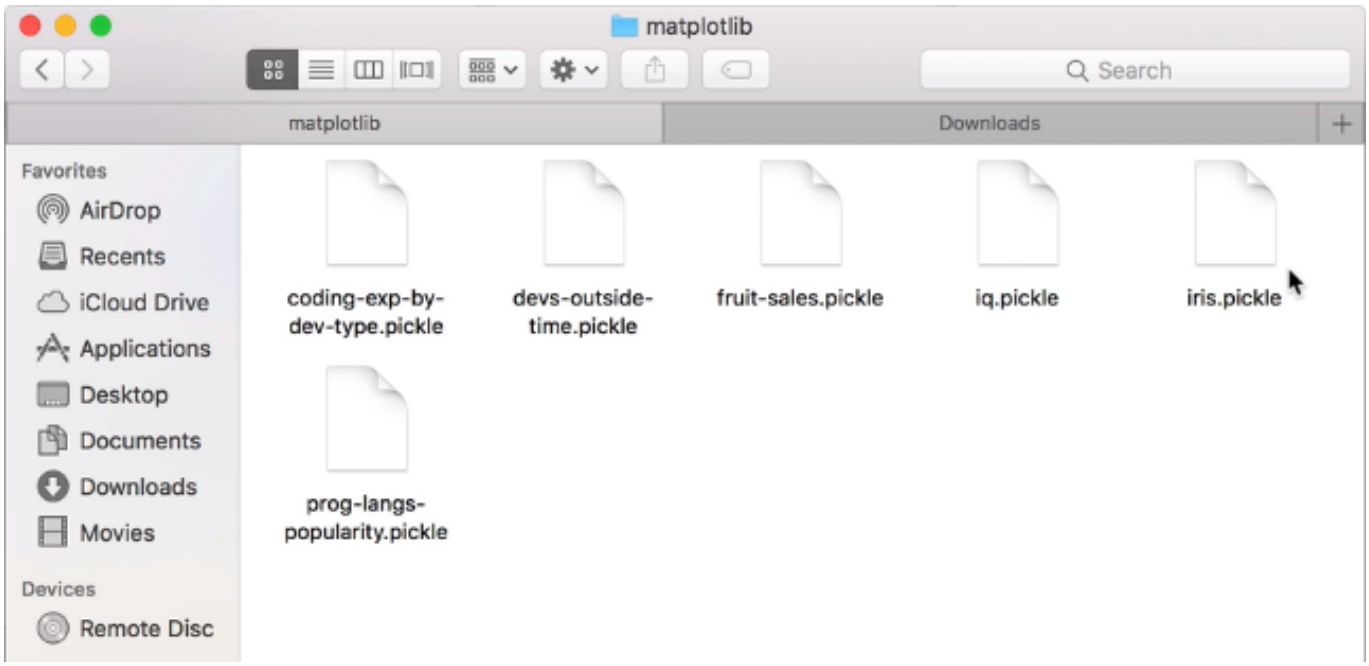
Next download the *data-viz-source-code.zip* file and unzip it.



Lets double click on this unzipped file and copy the folders within, into your working directory (*data-viz* in my setup).

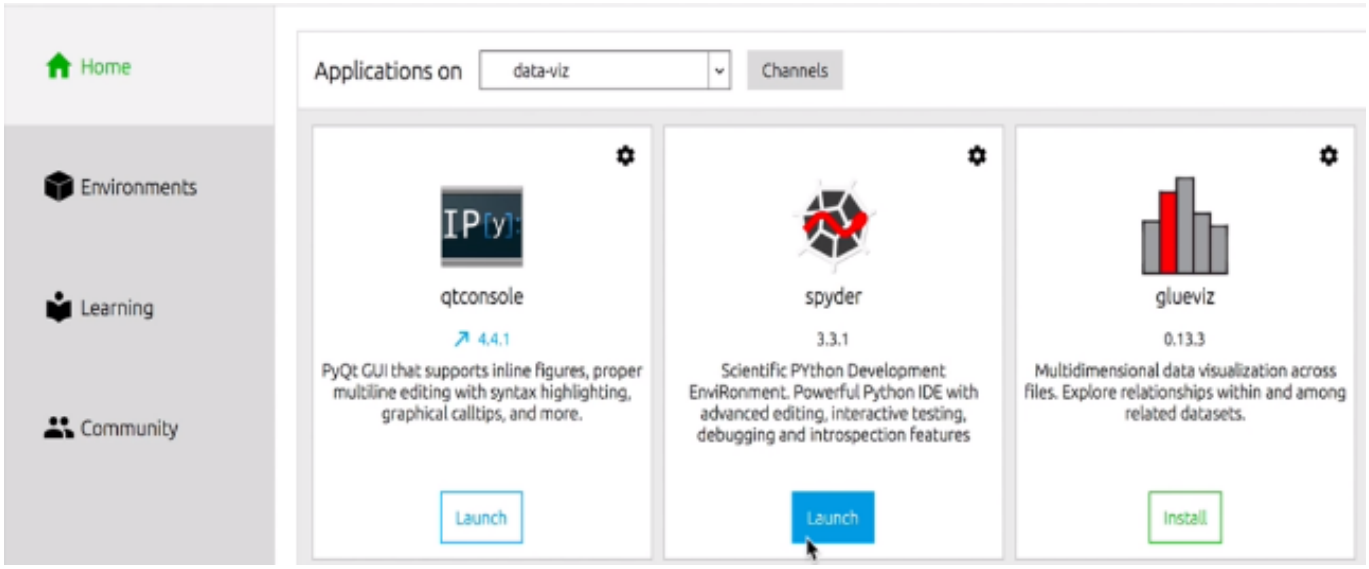


These 3 folders contain all the data we need. e.g. The *matplotlib* folder contains:

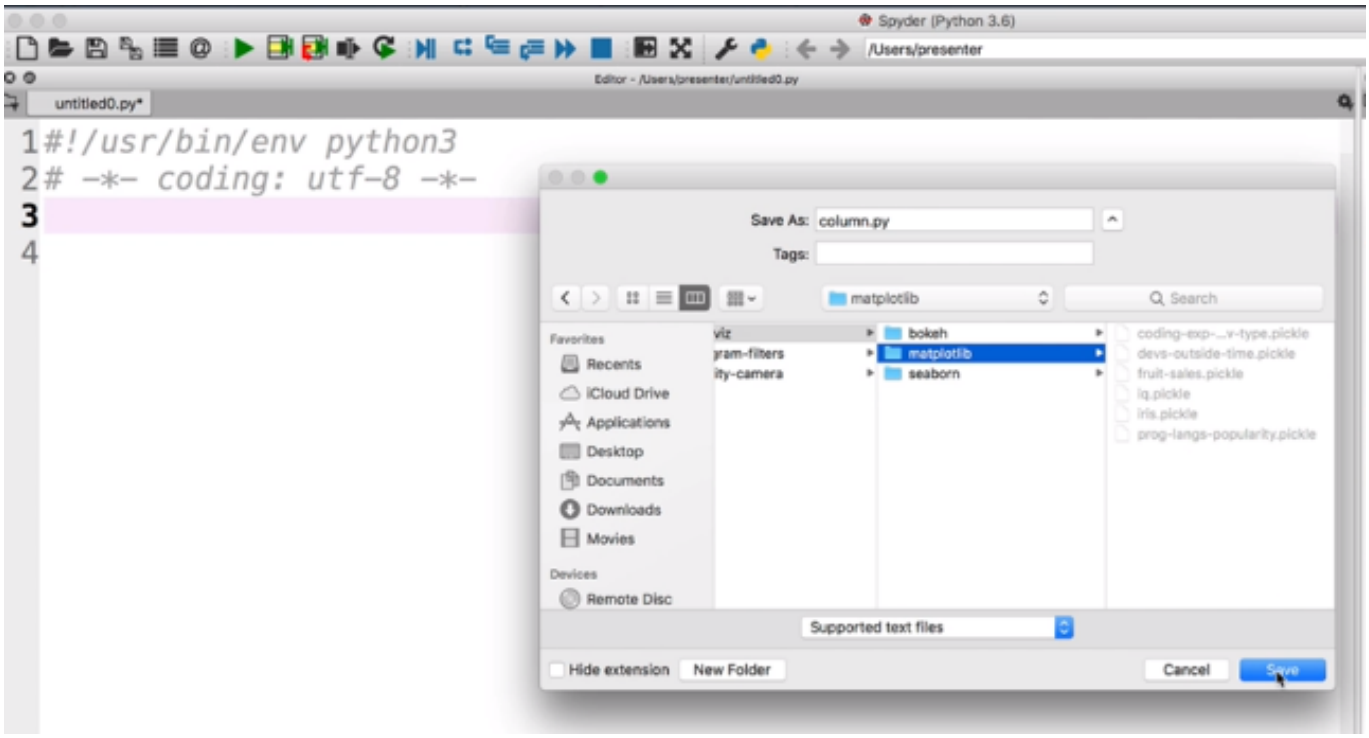


The folder contains a number of *.pickle* files. Python provides the pickle serialization format as a way to load and save Python objects efficiently.

Lets start coding. Lets make sure we have chosen the correct environment and launch Spyder (our development tool).



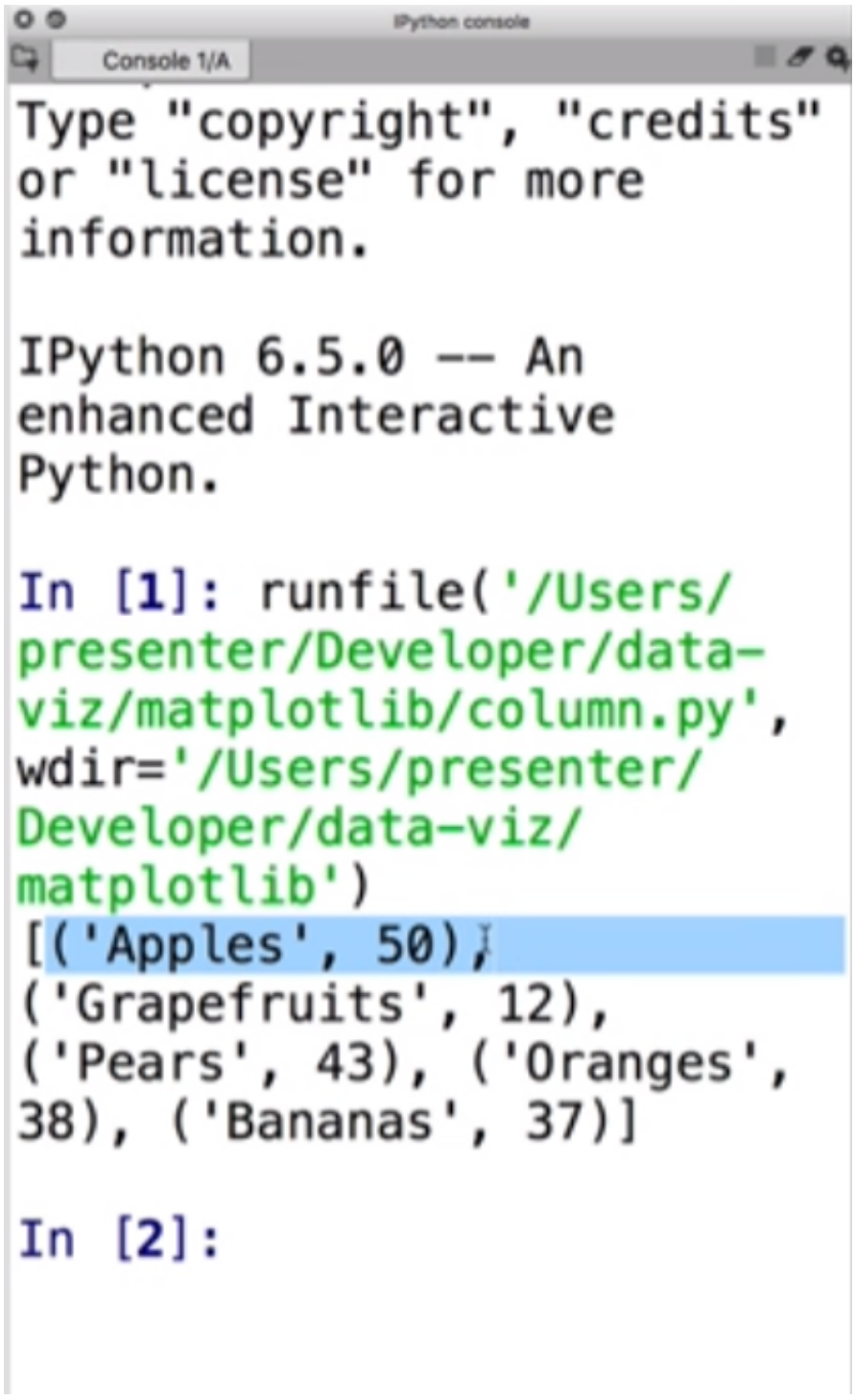
Lets create a new Python script called *column.py* in the same folder as the pickle files (the *matplotlib* folder).



Lets import the relevant packages.

```
# import matplotlib.pyplot as an alias
import matplotlib.pyplot as plt

# python object serialization library
import pickle

# load data using with block (f is closed automatically after the block)
# rb means “read binary data”
with open ('fruit-sales.pickle', 'rb') as f :
    data = pickle.load(f)

print (data)
```

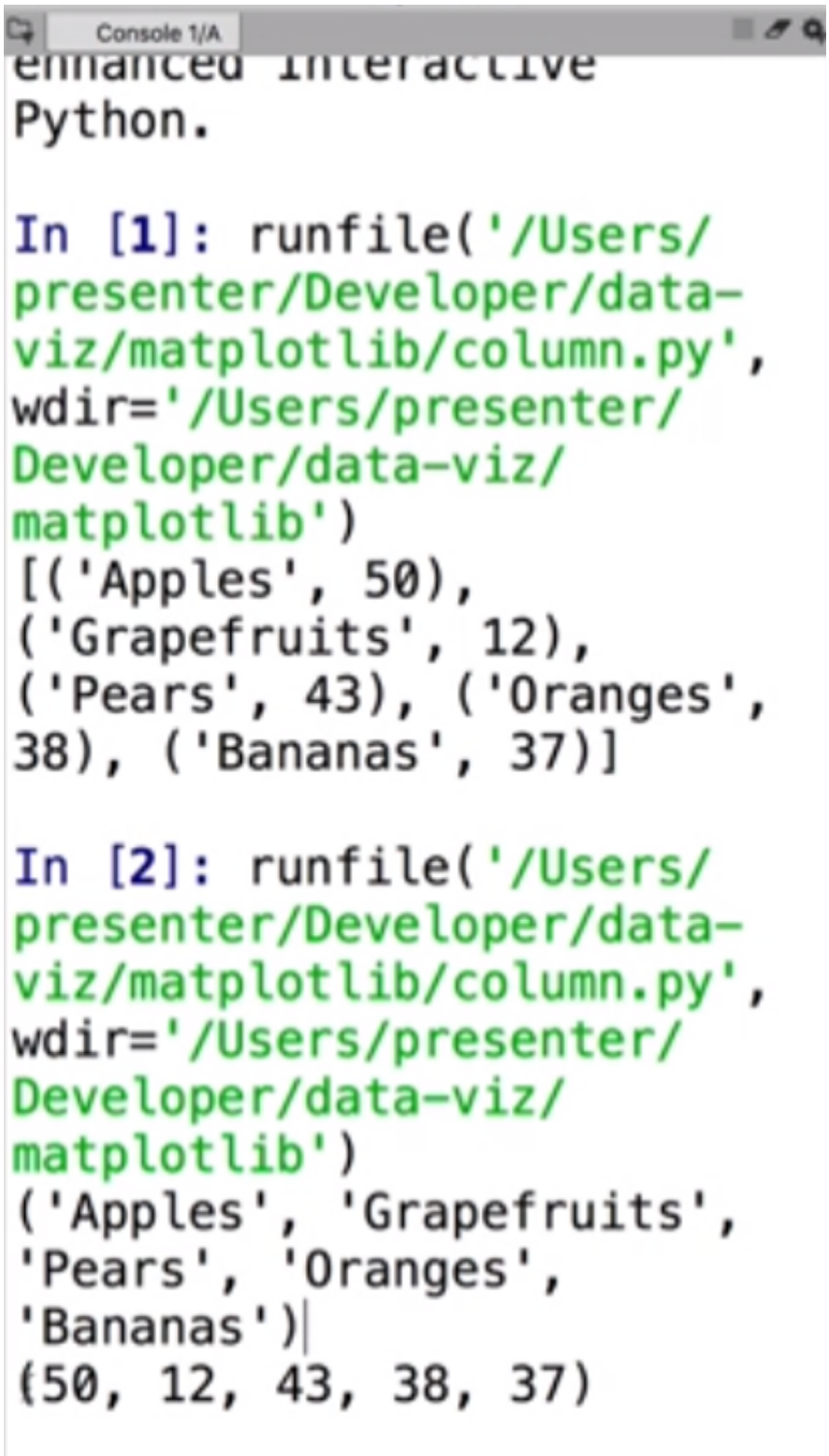
Lets run this code to see the data.



We see a tuple of elements each with the name of the fruit and the quantity sold. To make this easier to work with, lets split this tuple into 2 separate lists – one with the names and one with the numerical values.

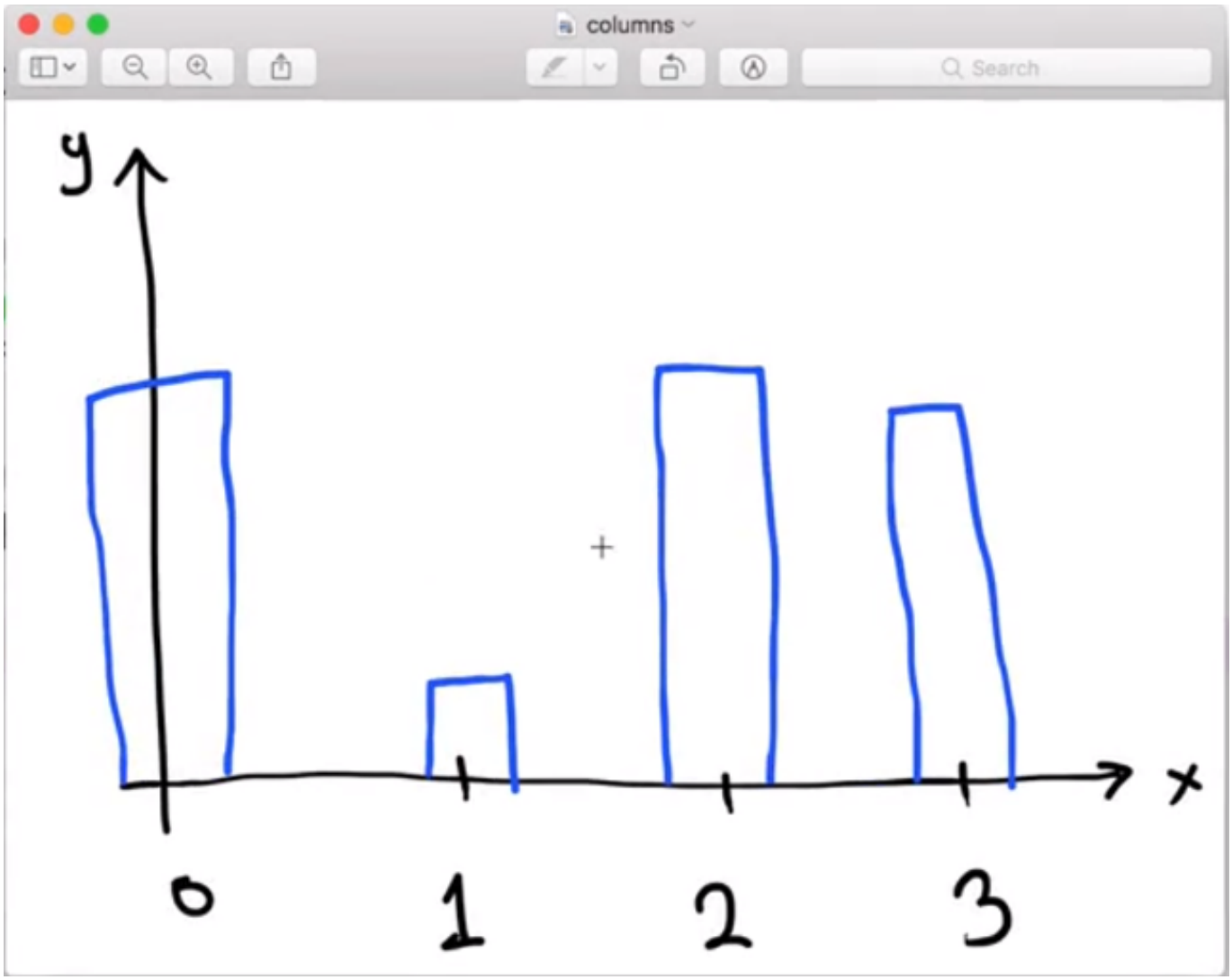
```
#splitting a list into 2 lists
fruit, num_sold = zip(*data)
print (fruit)
print (num_sold)
```


Lets run the code.



We can see the data in 2 different tuples now.

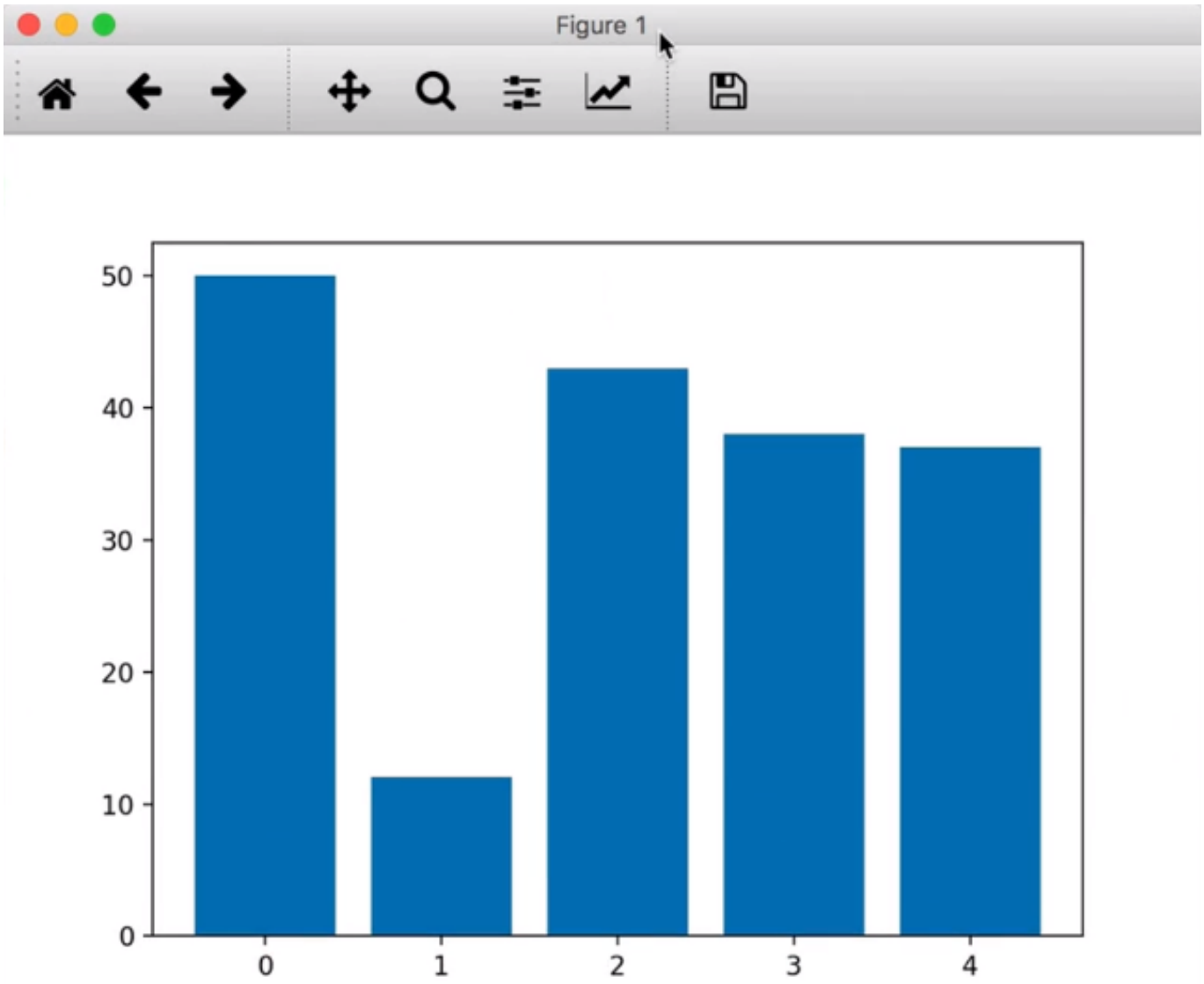
Let continue adding code to make the column plot. We need to tell *matplotlib* where to position the bars i.e. give it x-coordinates as in the diagram below.



The bars are at positions 0, 1, 2, 3. Lets create a list containing these values.

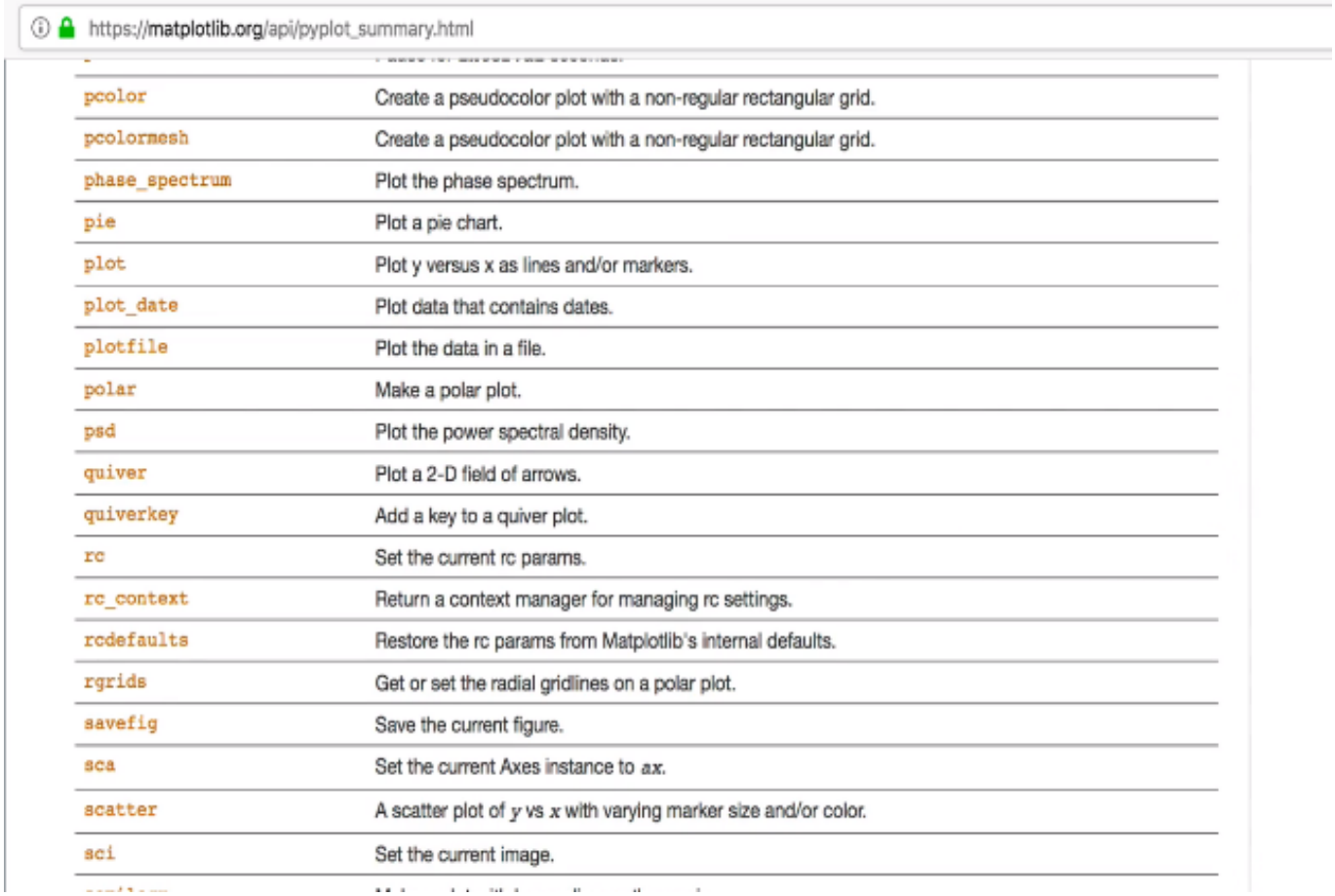
```
# list from 0 – the number of fruit
bar_coords = range(len(fruit))

# tell matplotlib to plot this

# second argument specifies the height of the bars
plt.bar(bar_coords, num_sold)

# show plot
plt.show()
```

Lets run this code.



As expected we see the bars centered around 0, 1, 2, 3, 4.

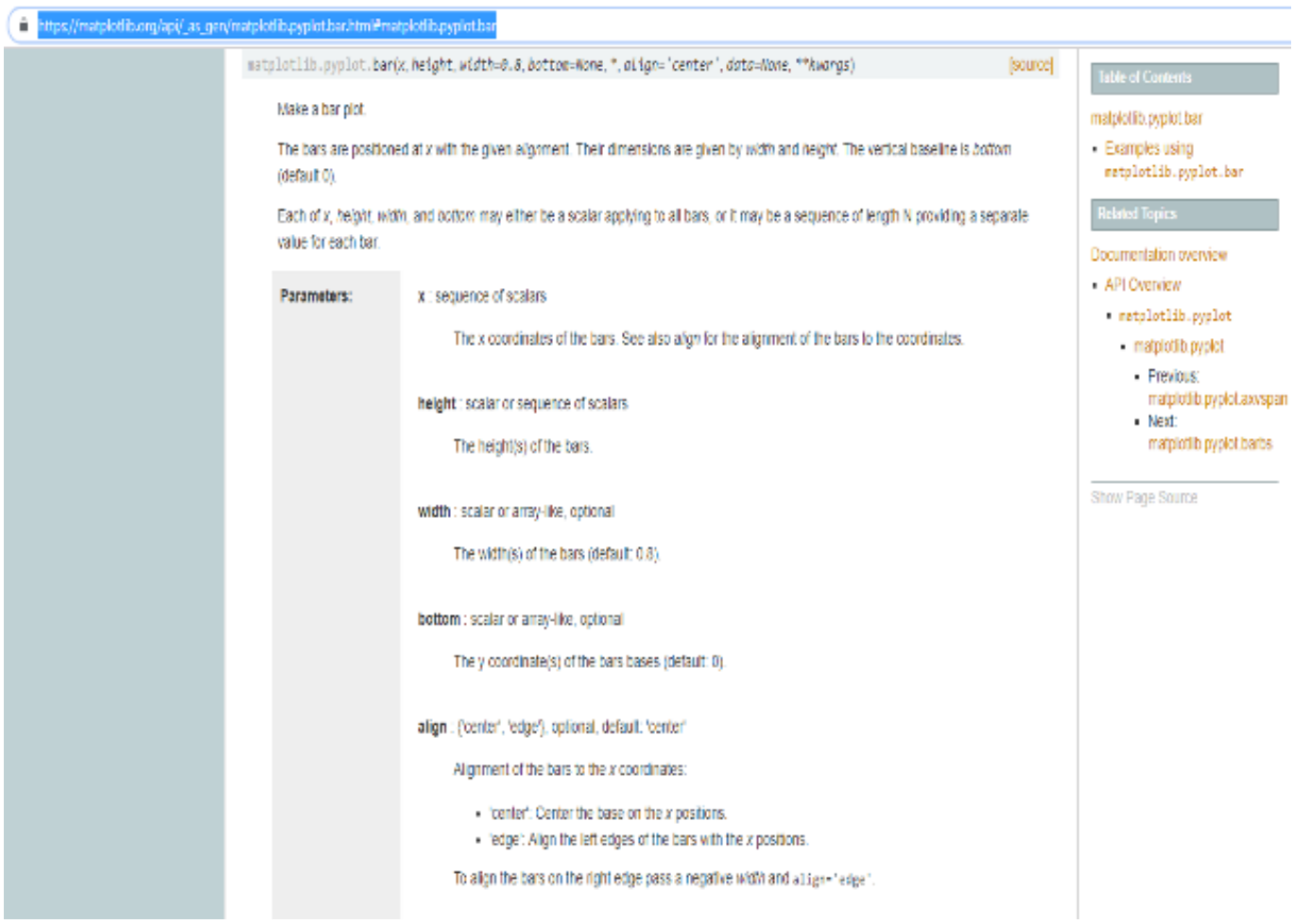
The matplotlib documentation has a list of all functions we can use to produce various kinds of plots.



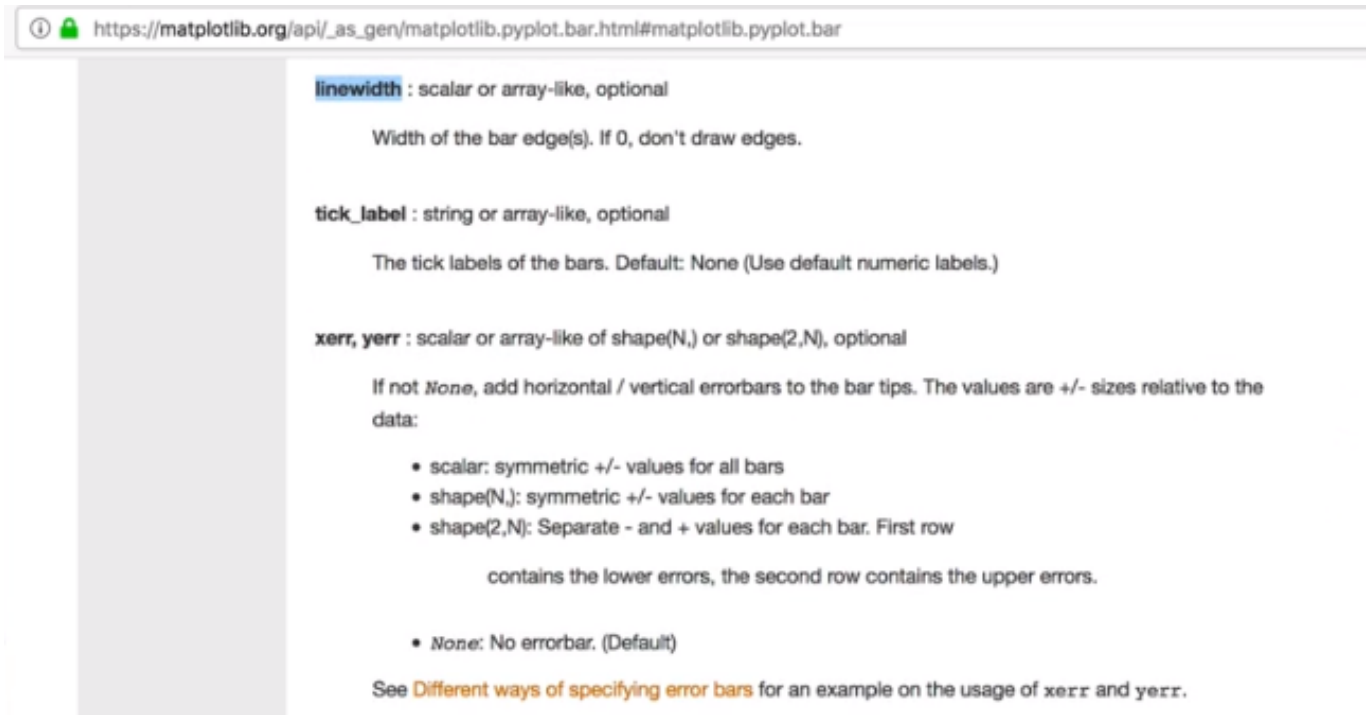
pcolor	Create a pseudocolor plot with a non-regular rectangular grid.
pcolormesh	Create a pseudocolor plot with a non-regular rectangular grid.
phase_spectrum	Plot the phase spectrum.
pie	Plot a pie chart.
plot	Plot y versus x as lines and/or markers.
plot_date	Plot data that contains dates.
plotfile	Plot the data in a file.
polar	Make a polar plot.
psd	Plot the power spectral density.
quiver	Plot a 2-D field of arrows.
quiverkey	Add a key to a quiver plot.
rc	Set the current rc params.
rc_context	Return a context manager for managing rc settings.
rcdefaults	Restore the rc params from Matplotlib's internal defaults.
rgrids	Get or set the radial gridlines on a polar plot.
savefig	Save the current figure.
sca	Set the current Axes instance to *ax*.
scatter	A scatter plot of *y* vs *x* with varying marker size and/or color.
sci	Set the current image.

As we can see, there are APIs for all kinds of plots – histograms, spectrograms, 3D plots, contour plots etc.

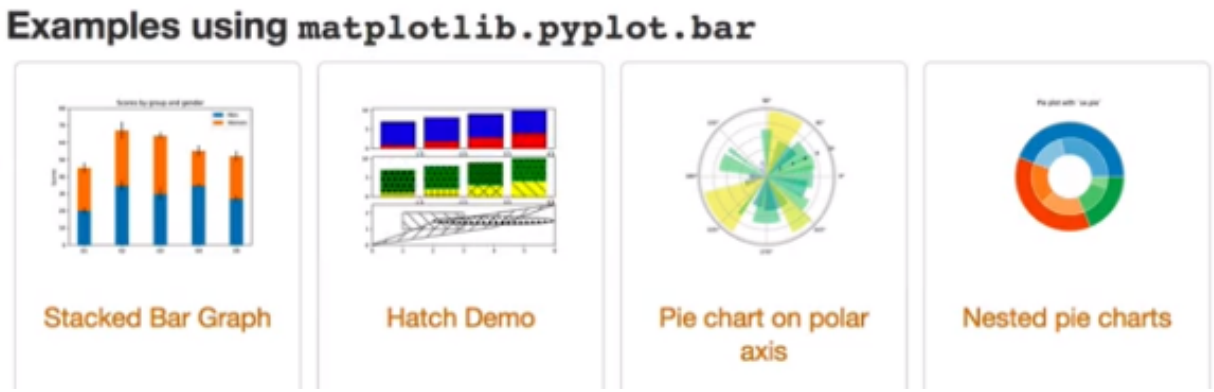
Lets look at the API documentation for bar plots.



We can see the parameters above. Scrolling down there are other parameters we can change such as color, linewidth etc.



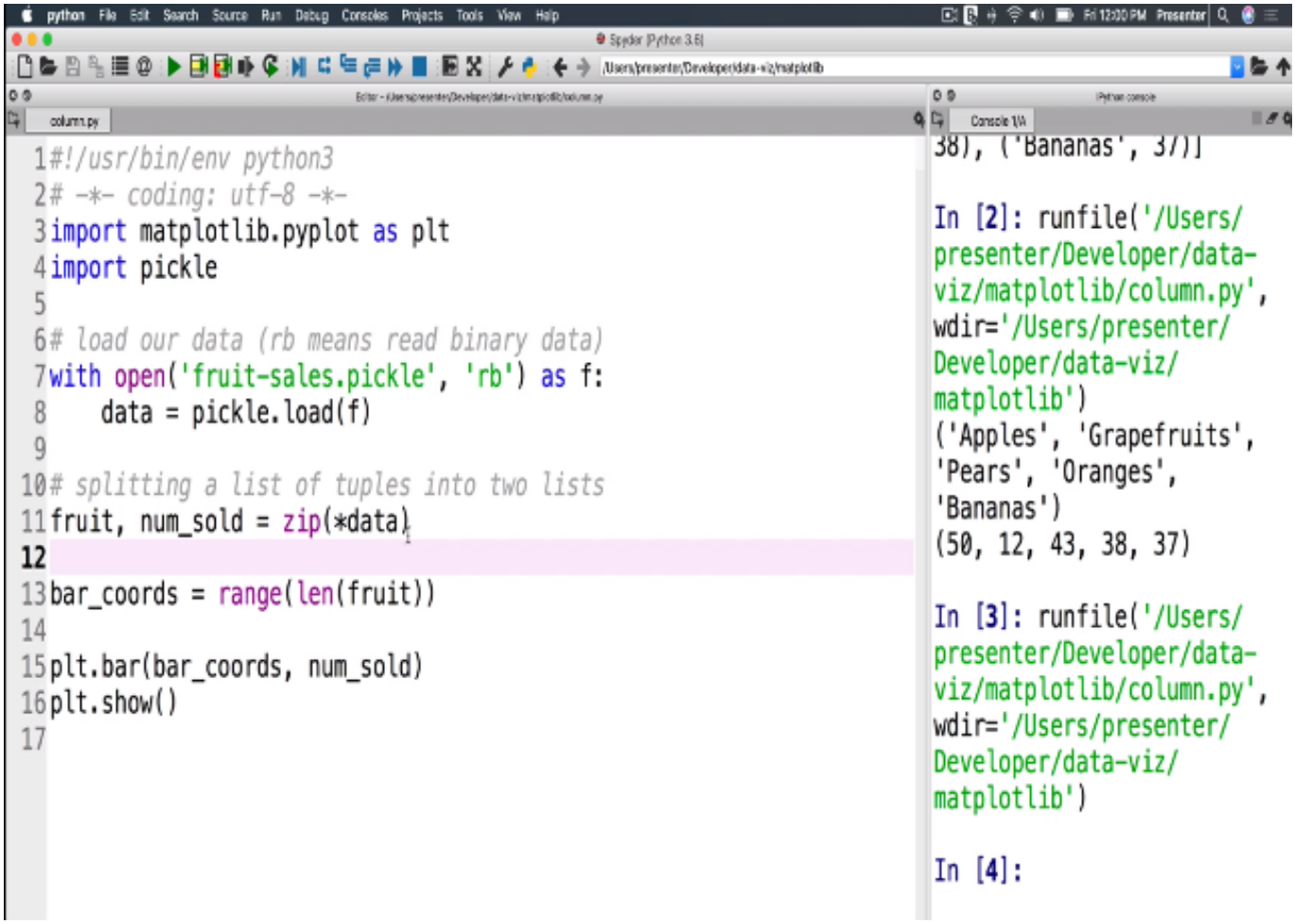
At the bottom of the page are some examples of how to use the bar plot API.



Our column chart works, but does not really look nice. Lets add a few features like a plot title, axis labels. Lets do that in the next video.

In this video, we are going to make our plot look nicer by adding labels on the axes, the title. New code segments will be marked as # NEW.

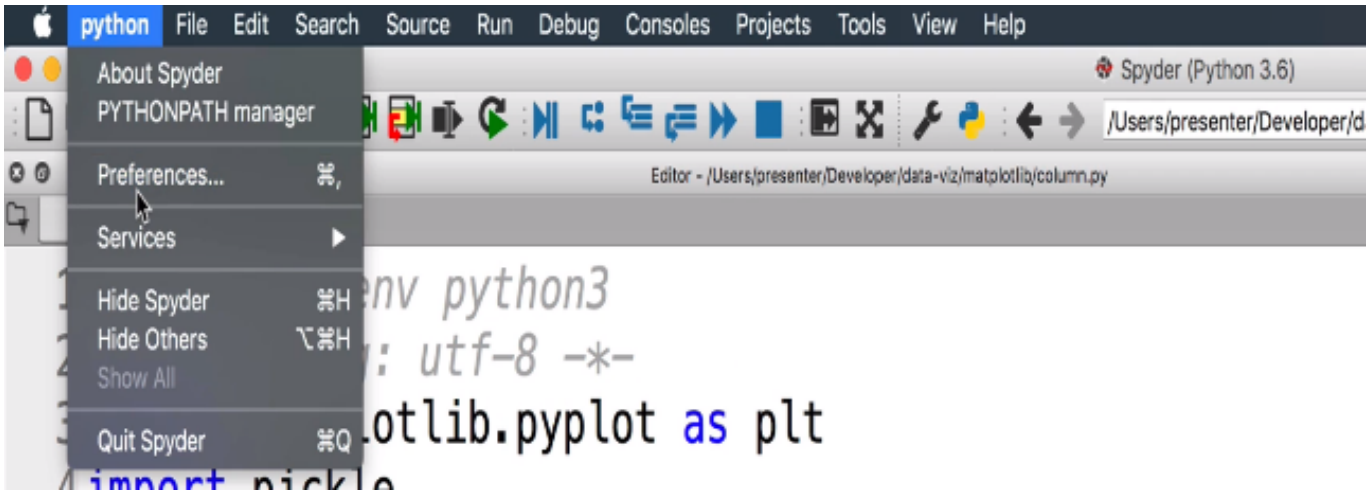
Lets start with the code we wrote for the plot so far.



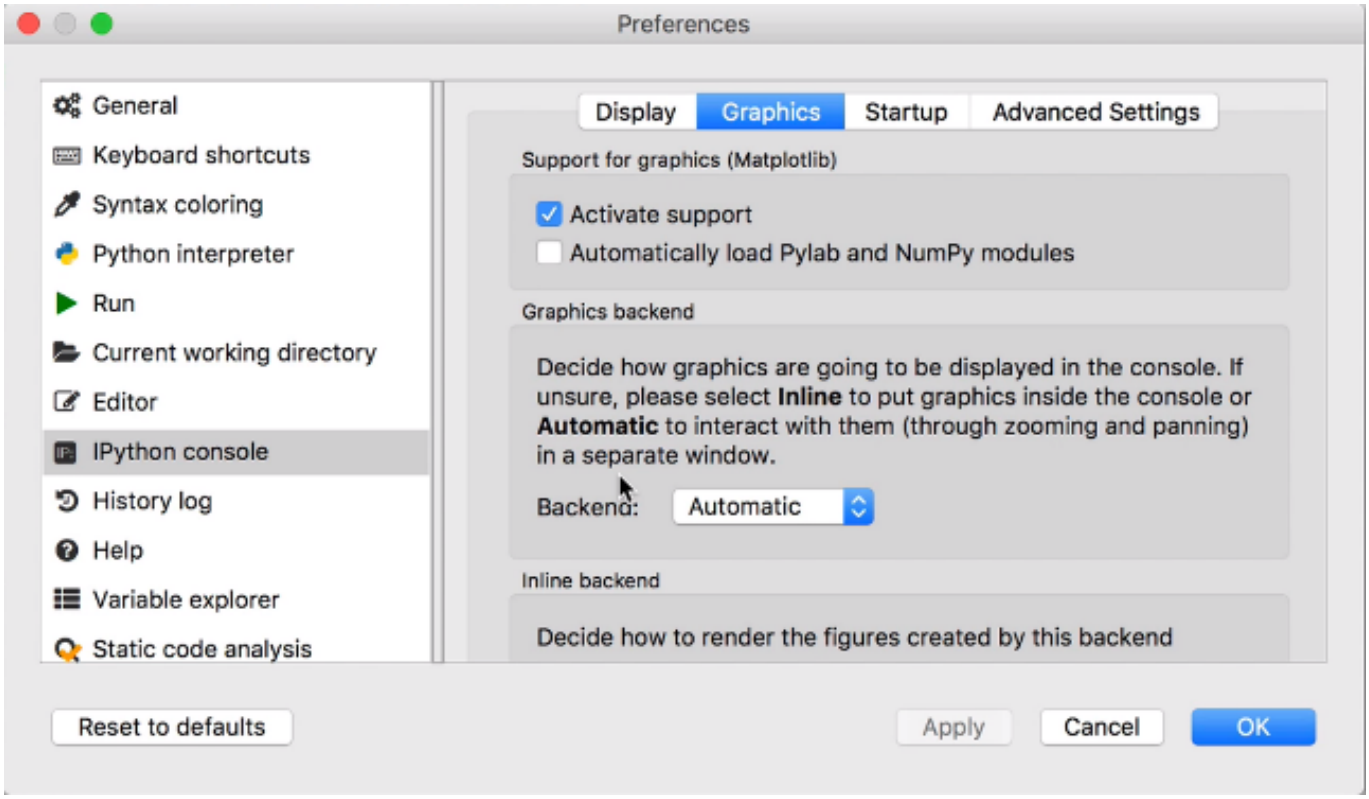
There are 2 ways we can view these plots. One was in a separate window as we showed in the previous video.

The other way is to have it display inside the *iPython* console. This is convenient when you don't want to launch a separate window for the plot.

To do this, we need to change some preferences in *Spyder*. Open the **Preferences** dialog as follows :



In the **Preferences** window, go to IPython console => Graphics.



If the **Backend** option is set to **Automatic**, the plot will open in a separate window with buttons that you can use to manipulate the plot. If you select **Inline**, the plot will show in the console but there won't be any buttons to manipulate the plot with. After making a change to this setting, you'll need to quit Spyder and restart it.

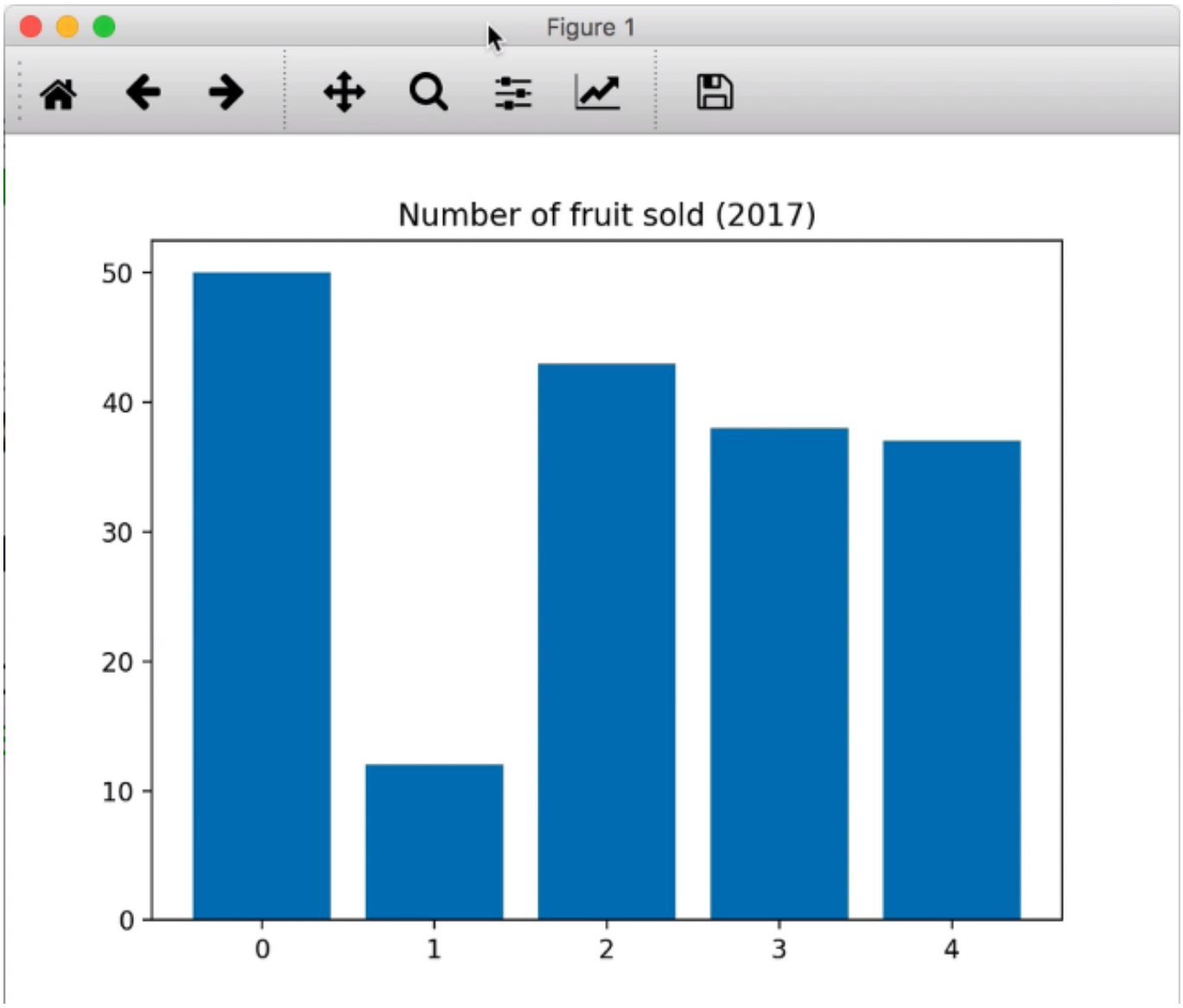
Lets start making our chart look a little better.

```
# import matplotlib.pyplot as an alias
import matplotlib.pyplot as plt

# python object serialization library
import pickle
```



```
# load data using with block (f is closed automatically after the block)
# rb means “read binary data”
with open ('fruit-sales.pickle', 'rb') as f :
    data = pickle.load(f)

#splitting a list into 2 lists
fruit, num_sold = zip(*data)

# list from 0 – the number of fruit
bar_coords = range(len(fruit))

# tell matplotlib to plot this

# second argument specifies the height of the bars
plt.bar(bar_coords, num_sold)

# NEW
# add plot title
plt.title(' No of fruit sold(2017)')

# show plot
plt.show()
```

Running the code, we get:

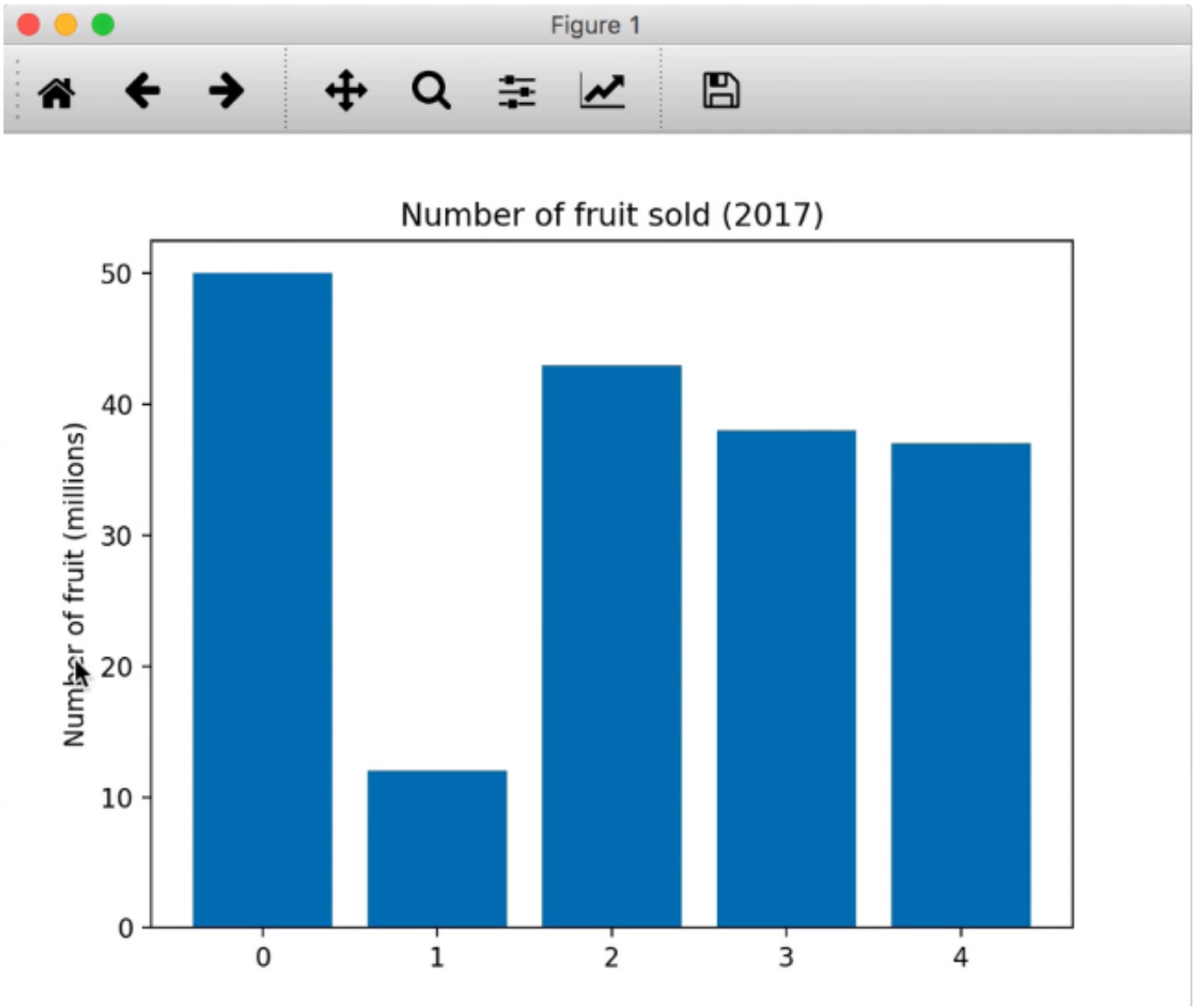


We see the plot title we just added.

Lets set an **axis label** for the **y-axis**.

```
# NEW
plt.ylabel('Number of fruit(millions)')
```

Running the code after the above addition, we see.

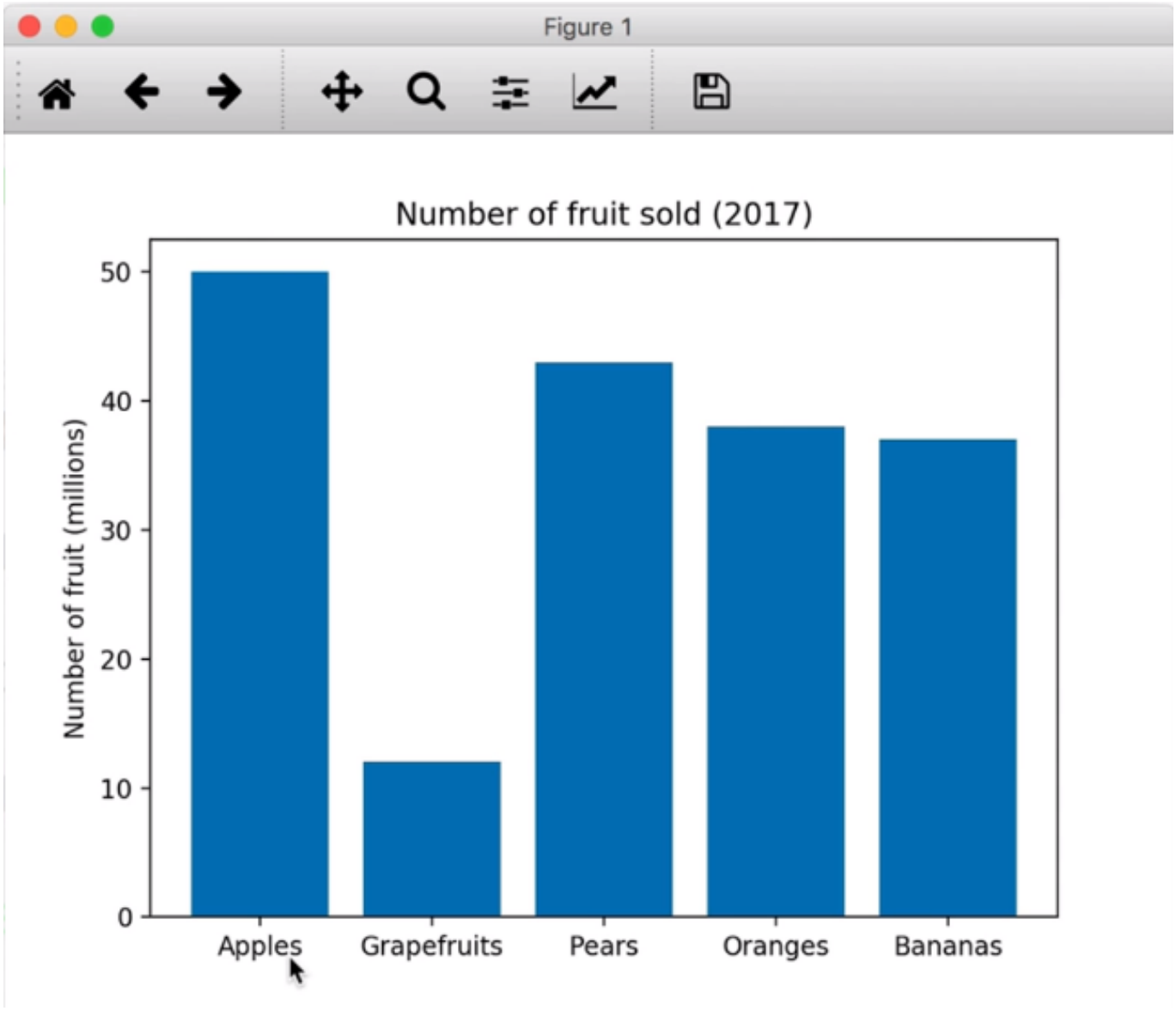


We see the label along the y-axis. The pyplot documentation will show ways we can rotate this label by 180 degrees.

We can setup the x-axis so that instead of 0, 1, 2, 3, 4, we can use the actual fruit. We use the matplotlib *xticks* function. The fruits are split in order with the *zip* function. so we don't have to worry about the correct order.

```
# NEW
# replace bar_coords with fruit names
plt.xticks(bar_coords, fruit)
```

When we run this code, we get:



We see the fruit names on the x-axis. Apples sold the most. Grapefruits did not sell as much. So column bar charts allow us to look at the data and infer information from it. The x-axis has categories and bar charts are very good to depict *categorical* data. The term bar charts and column charts are used interchangeably.

Summary

So we saw how to add annotations such a title, y-label etc. to any *matplotlib* chart we work with. Like *ylabel*, we have *xlabel* that lets you label the x-axis. We choose not to do it since we have fruit names on the x-axis.

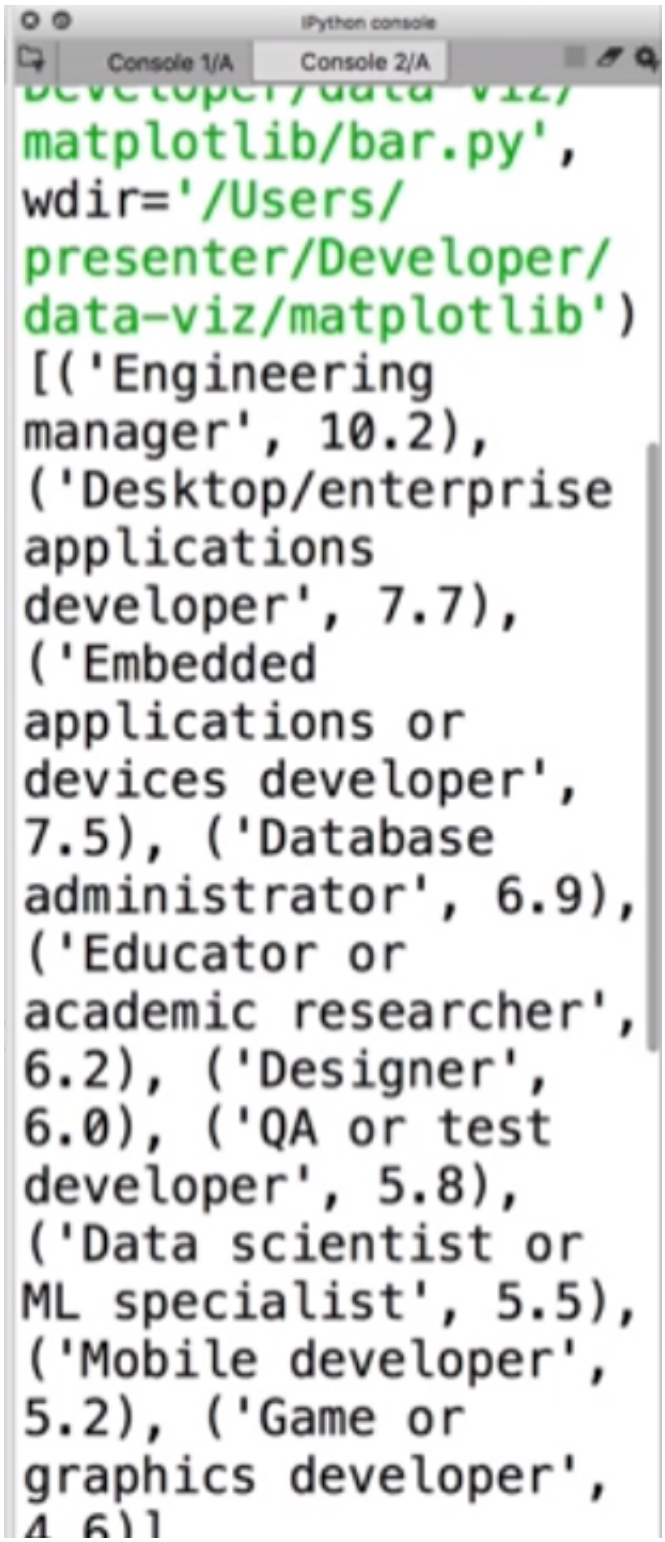
In this video, we are going to look at a horizontal bar chart and some real data. I have some data taken from the Stack Overflow developer survey. Stack Overflow is a website where developers can ask coding questions and they are answered community style. They release a developer survey every year, so I pulled some results from the 2017 survey and we are going to visualize this data.

The data we are going to be looking at is years of coding experience organized by the type of developer you are. So if you are say a mobile developer, we can see how many years on average developers in this category has.

Lets create a new python script *bar.py* in the **matplotlib** directory. We can thus directly read in our data.

```
import matplotlib.pyplot as plt
import pickle

# load data
with open('coding-exp-by-dev-type.pickle', 'rb') as f :
    data = pickle.load(f)

print (data)
```

Lets run this code to view the data.

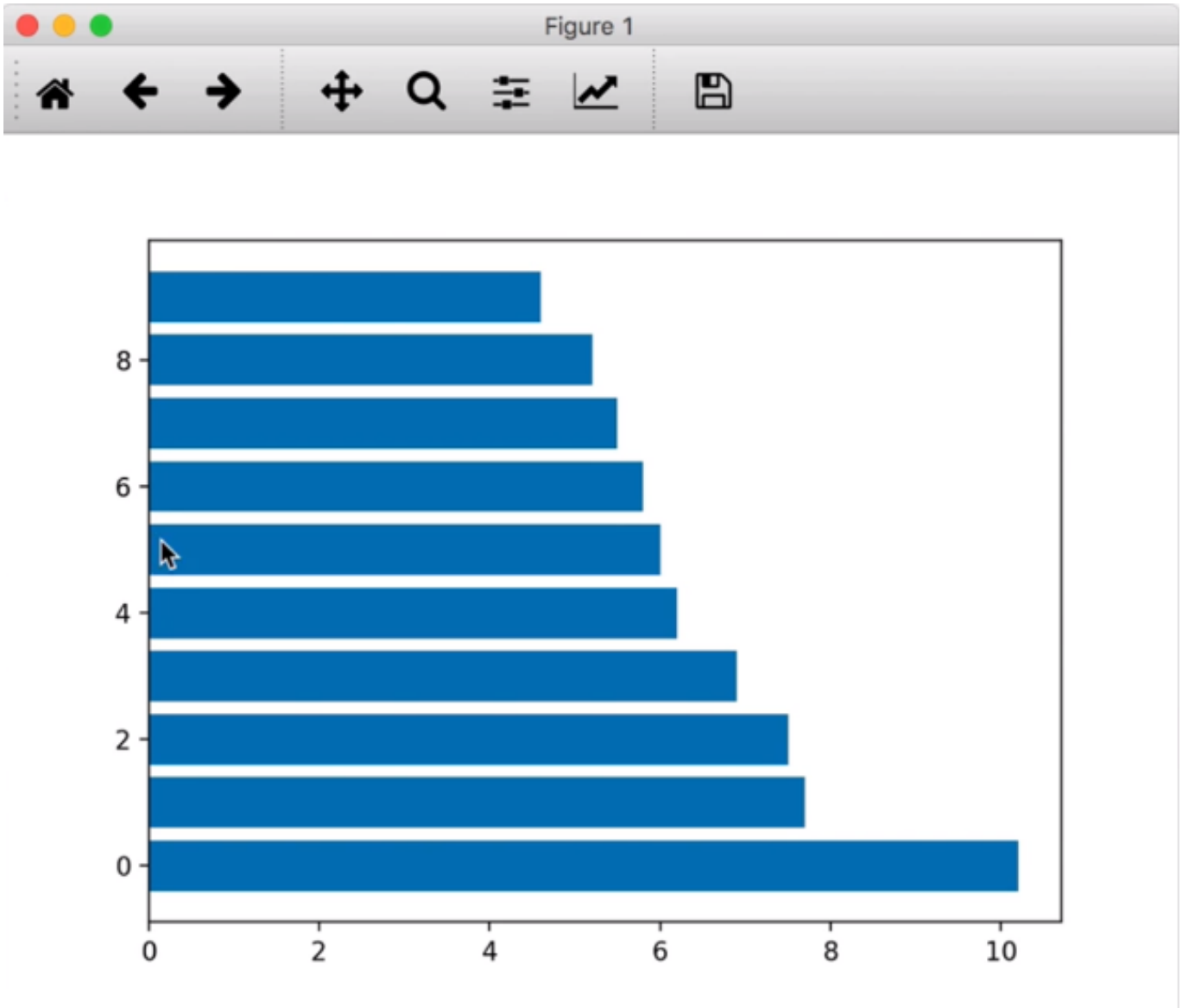


Like the column chart, let split this data into 2 same-sized lists.

```
dev_types, years_exp = zip(*data)

# bars on y-axis
bar_coords = range(len(dev_types))

# horizontal bar chart
```

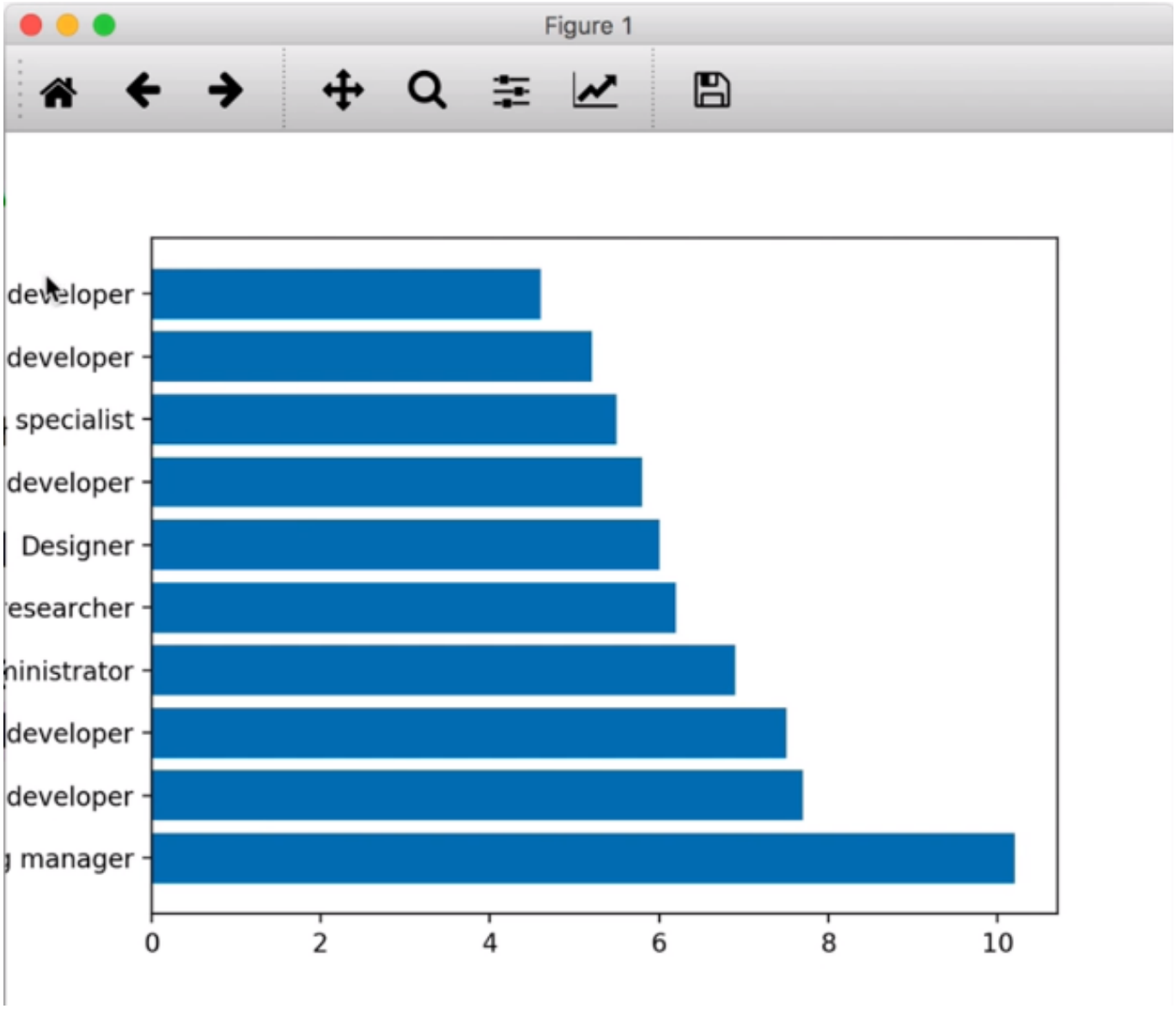


```
plt.barh(bar_coords, years_exp)
plt.show()
```



Lets replace the y-labels with the actual titles

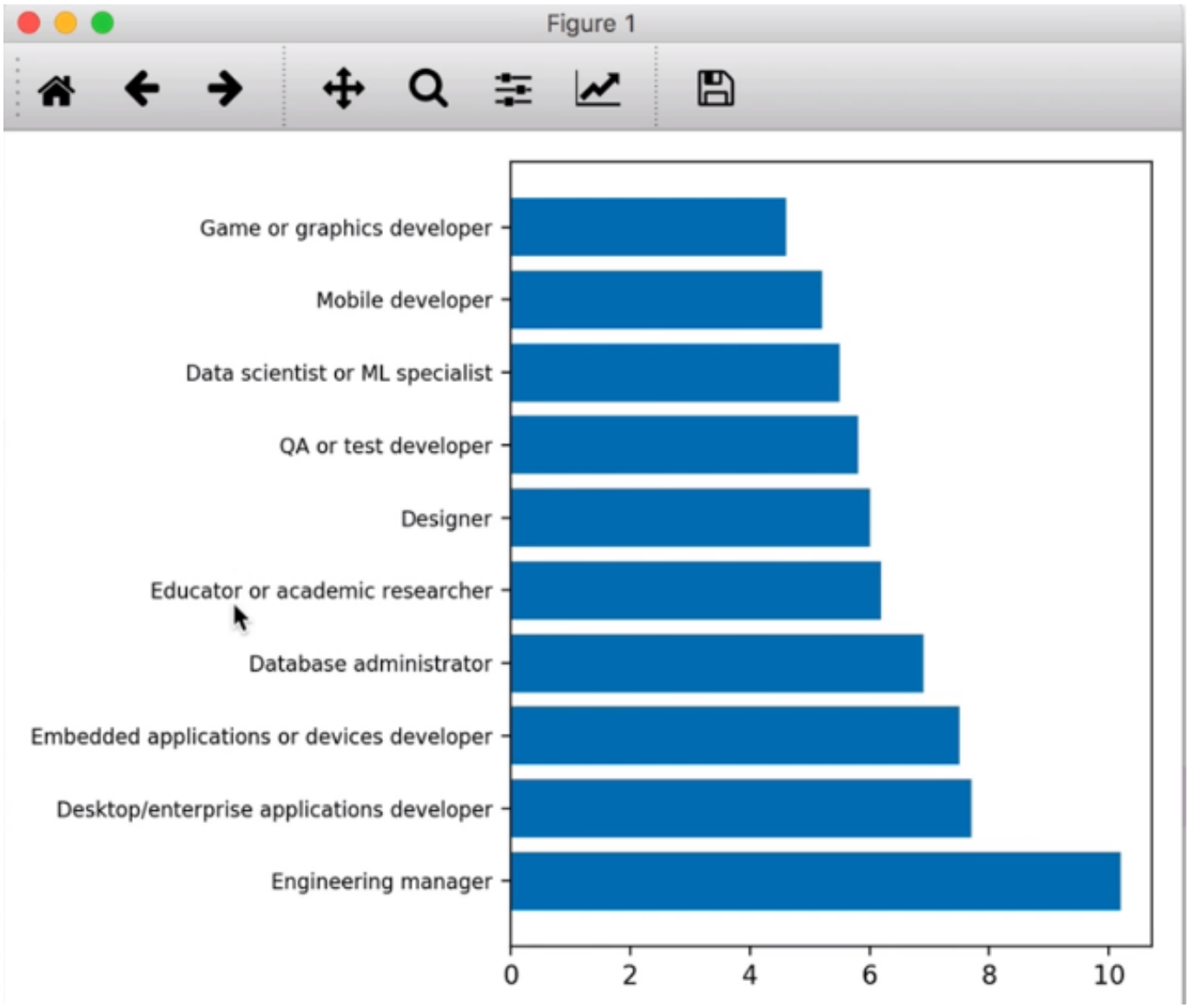
```
plt.yticks(bar_coords, dev_types)
```



The labels are a little cramped, so lets reduce the font size and remove white space.

```
plt.yticks(bar_coords, dev_types, fontsize=8)

# remove white spaces
plt.tight_layout()
```

Let's run this code.

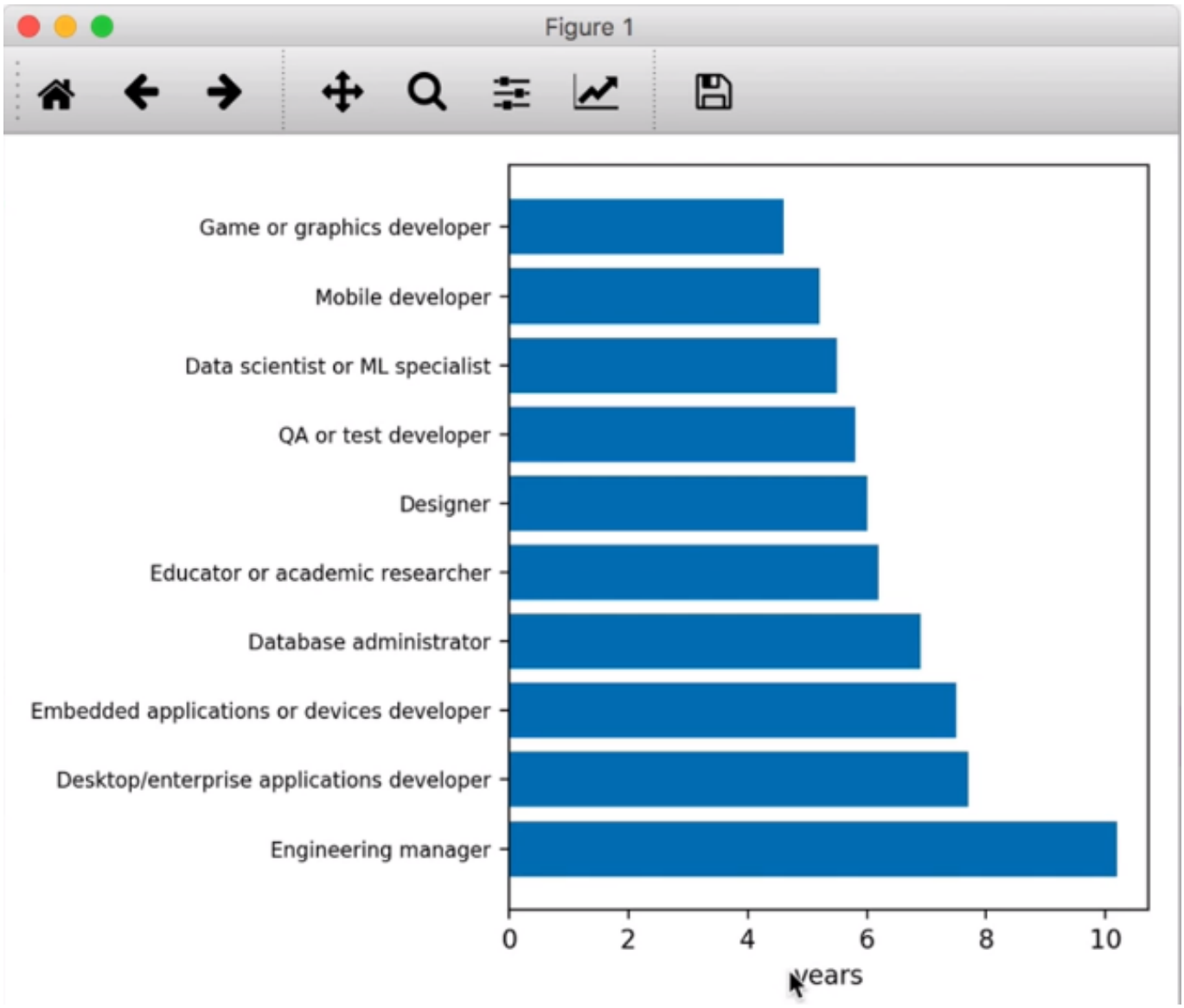


The text is now clearly visible.

Lets label the x-axis so that it is clear what it represents.

```
plt.xlabel('years')
```

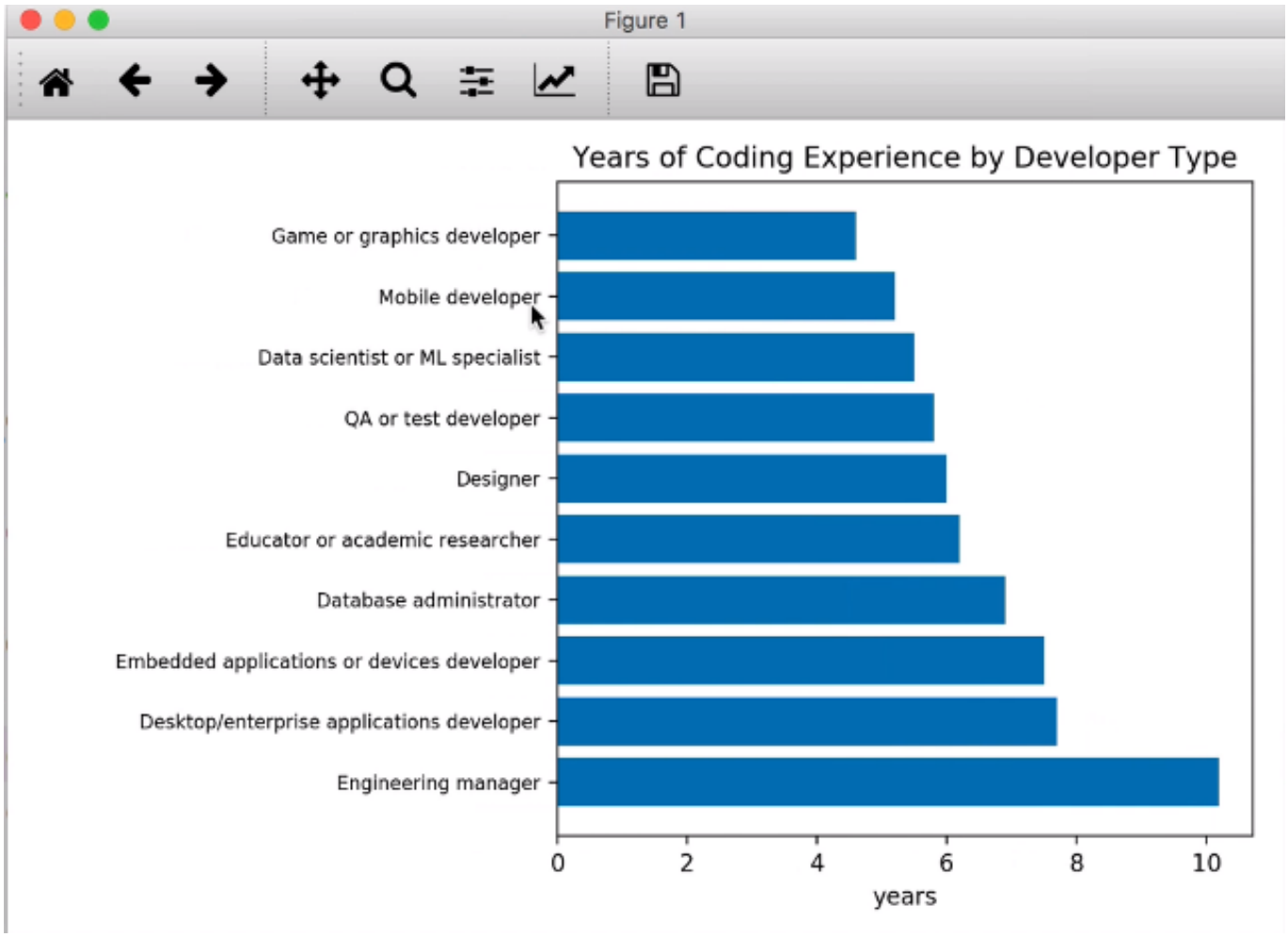
Let's run this code.



We see that the x-axis label says years.

Lets add a plot title.

```
plt.title('Years of Coding Experience by Developer Type')
```

We see that the title “Engineering Manager” has the most amount of coding experience. This could be because they start off being a software developer and move on to being an engineering manager. Similarly a Game or graphics developer has about 4.5 years experience.

If this data were put in a column chart, the title would all overlap. This is something to be aware of when deciding whether to use a column or bar chart.

Summary

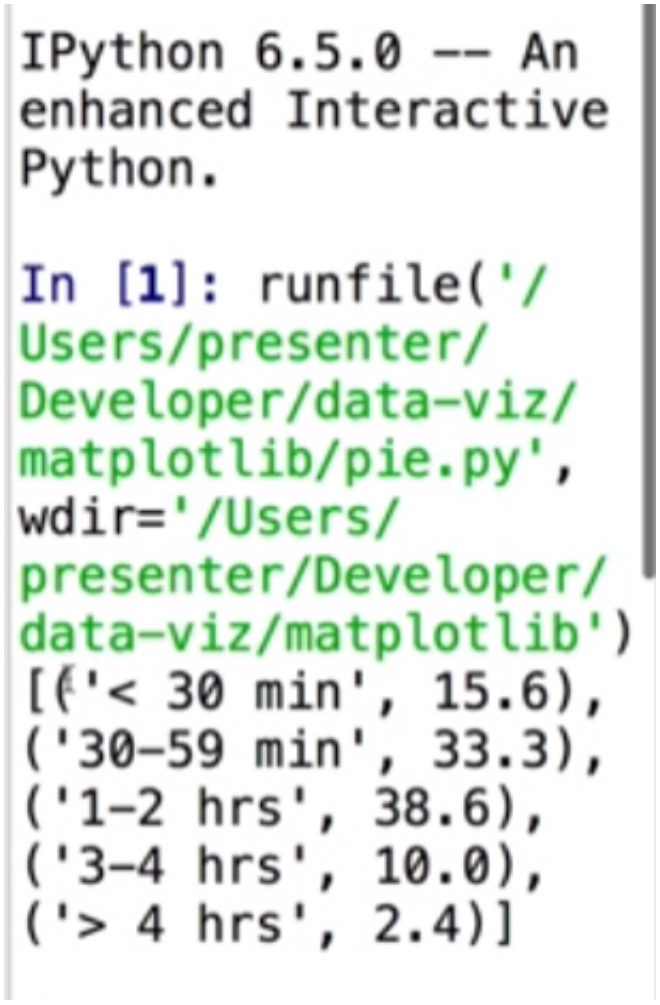
In this video, we use a horizontal bar chart to depict data and the syntax is almost identical to a column chart, except we flip x and y using (*barh*) and label the y-axis. We also introduced a new way to condense our graph and a couple of other parameters.

In this video, we are going to look at how to show a Pie Chart in matplotlib. We'll also look at some of the options we can apply to a pie chart to make it look nice.

Lets open Spyder and start coding. We are displaying the percentage of responses i.e. of all the people that responded to the Stack Overflow developer survey, what percentage of them say that they spend an hour outside or 4 hours outside everyday. This kind of data is nice to visualize as a Pie Chart.

```
import matplotlib.pyplot as plt
import pickle

# load data
with open('devs-outside-time.pickle', 'rb')  as f :
    data = pickle.load(f)

print (data)
```

Let's run this code.

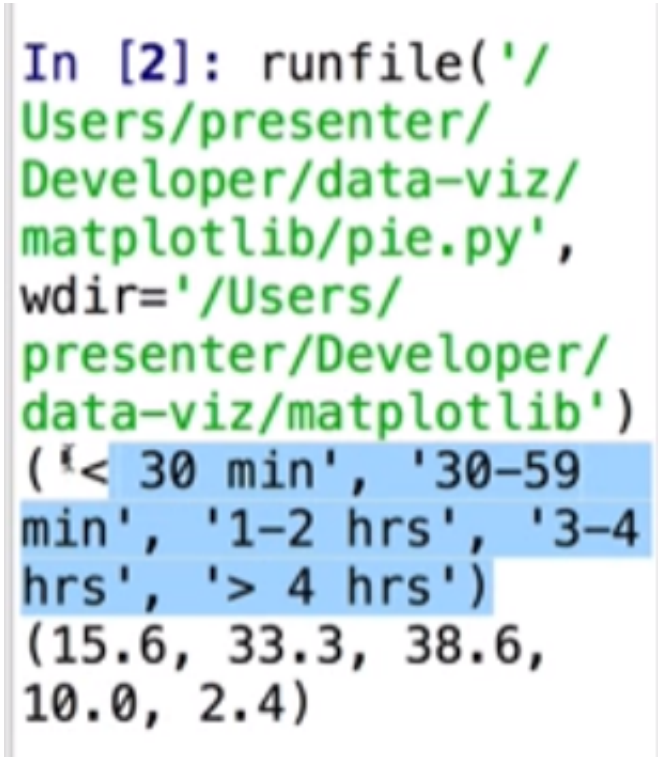


We get a set of tuples. Each tuple has 2 values – the amount of time and the actual percentage of people.

We now split the data into 2 lists.

```
# split into 2 lists
time, responses = zip(*data)
print (time)
print (responses)
```

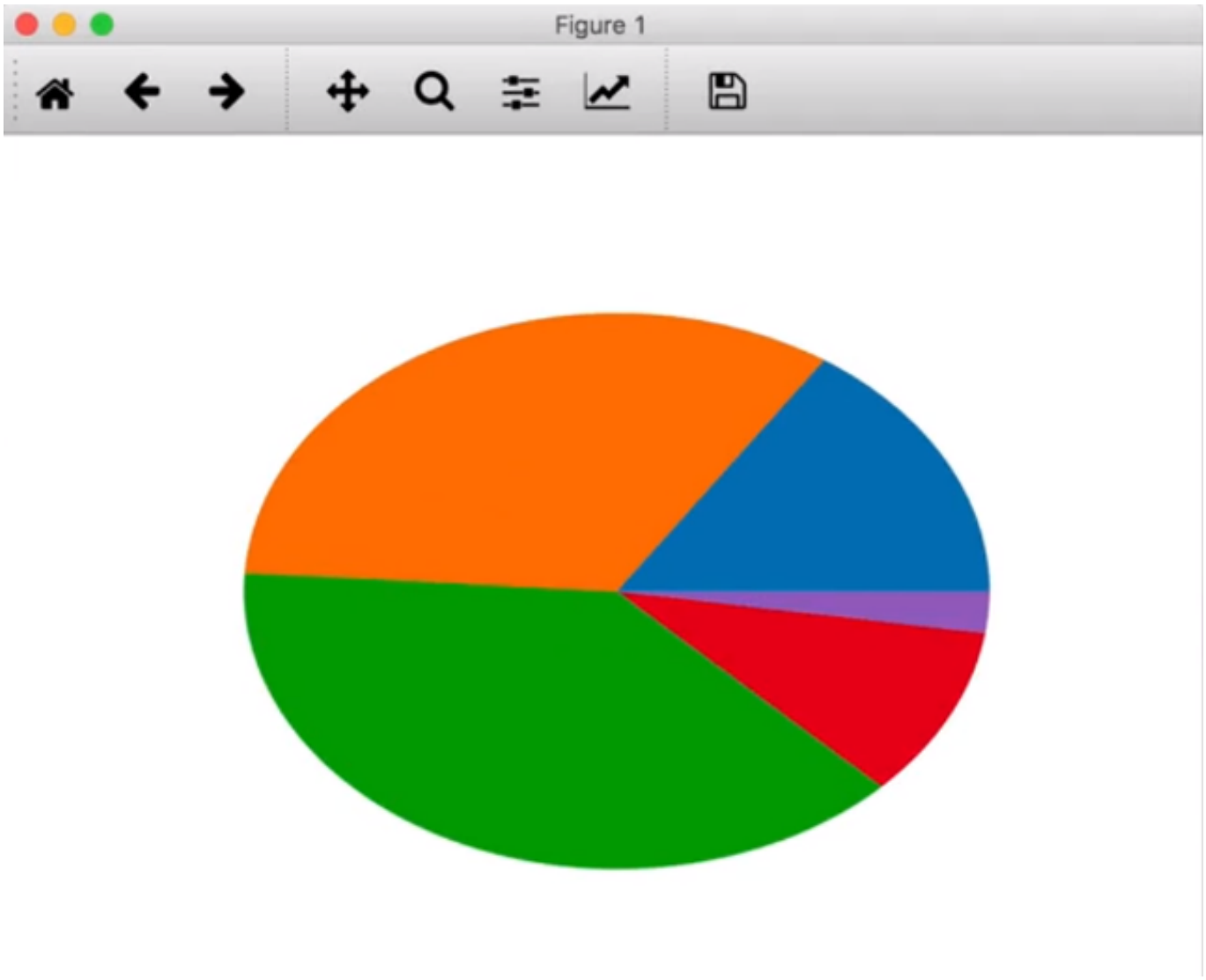
Let’s run this code.



Let’s display our pie chart.

```
plt.pie(responses)
plt.show()
```

Let’s run this code.

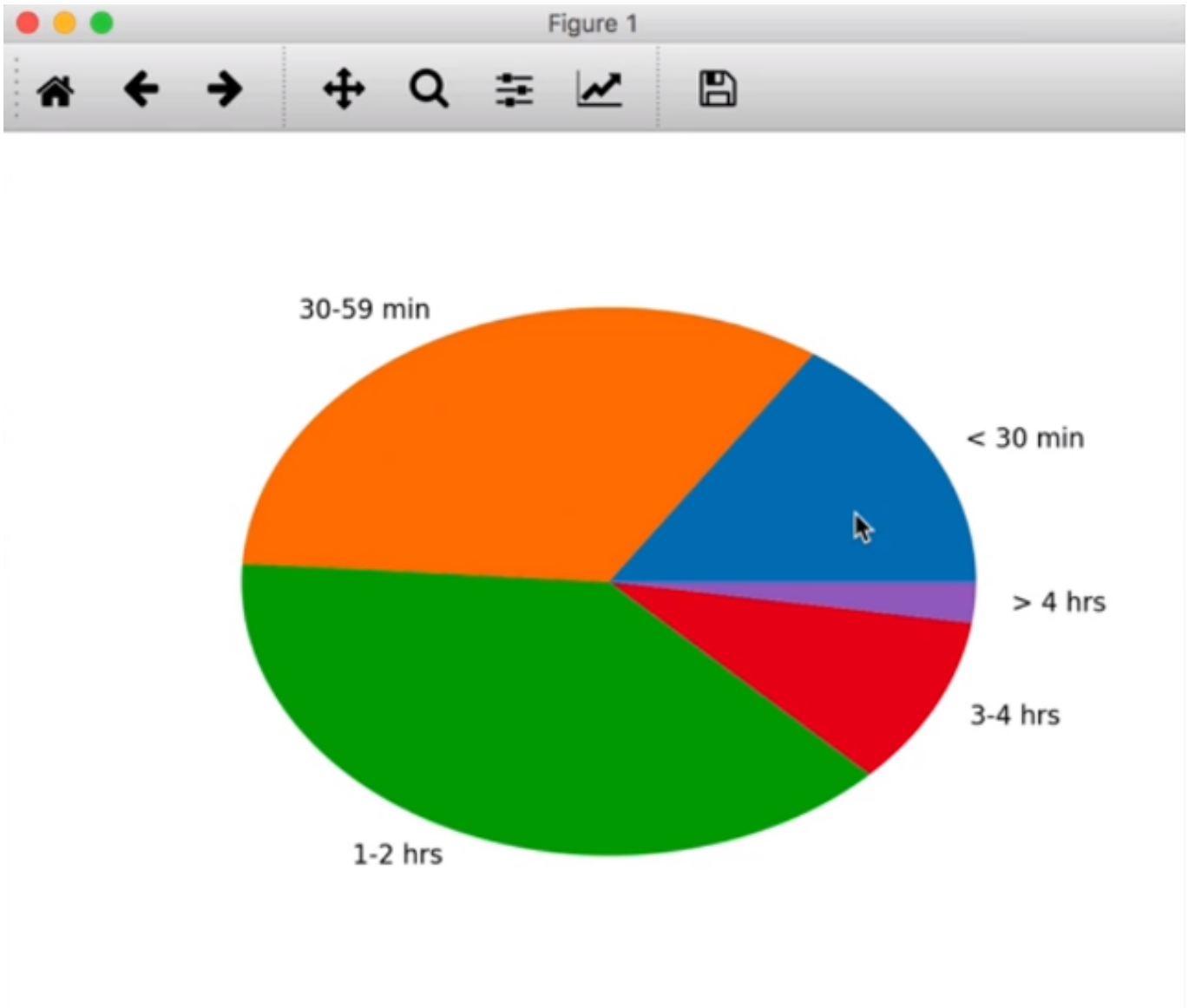


We have a pie chart but nothing that tells me what each of these sections are. Also, the chart is more oval than circular.

Let's **modify** the code above to add labels.

```
plt.pie(responses, labels=time)
```

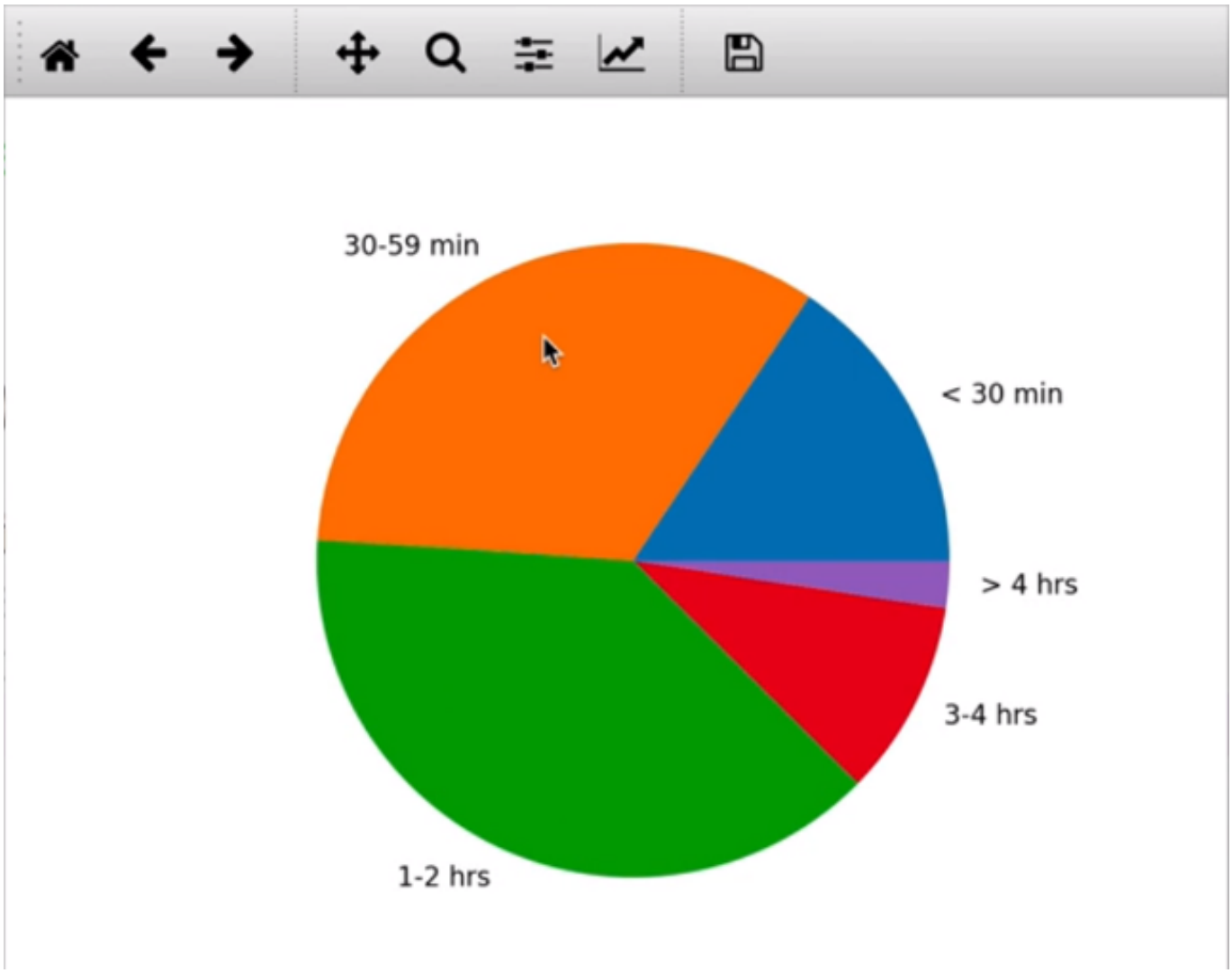
Lets run the code.



Lets make the display into a circle.

```
# force the x/y axes to have the same scale
# circle instead of an ellipse

plt.axis('equal')
```

Let’s run this code.

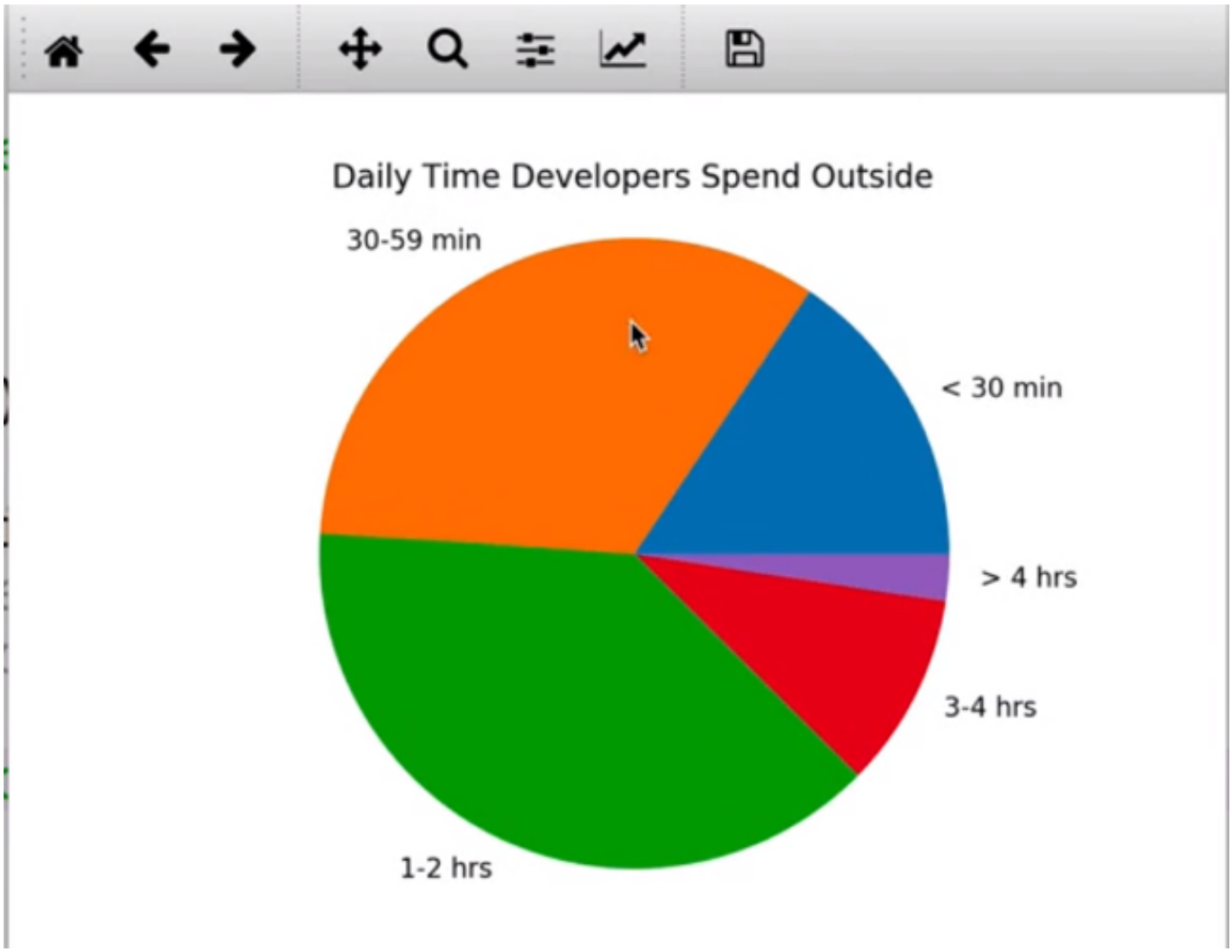


We have a circle.

Lets put a title on the plot.

```
plt.title('Daily Time Developer Spend Outside')
```

Let’s run this code.



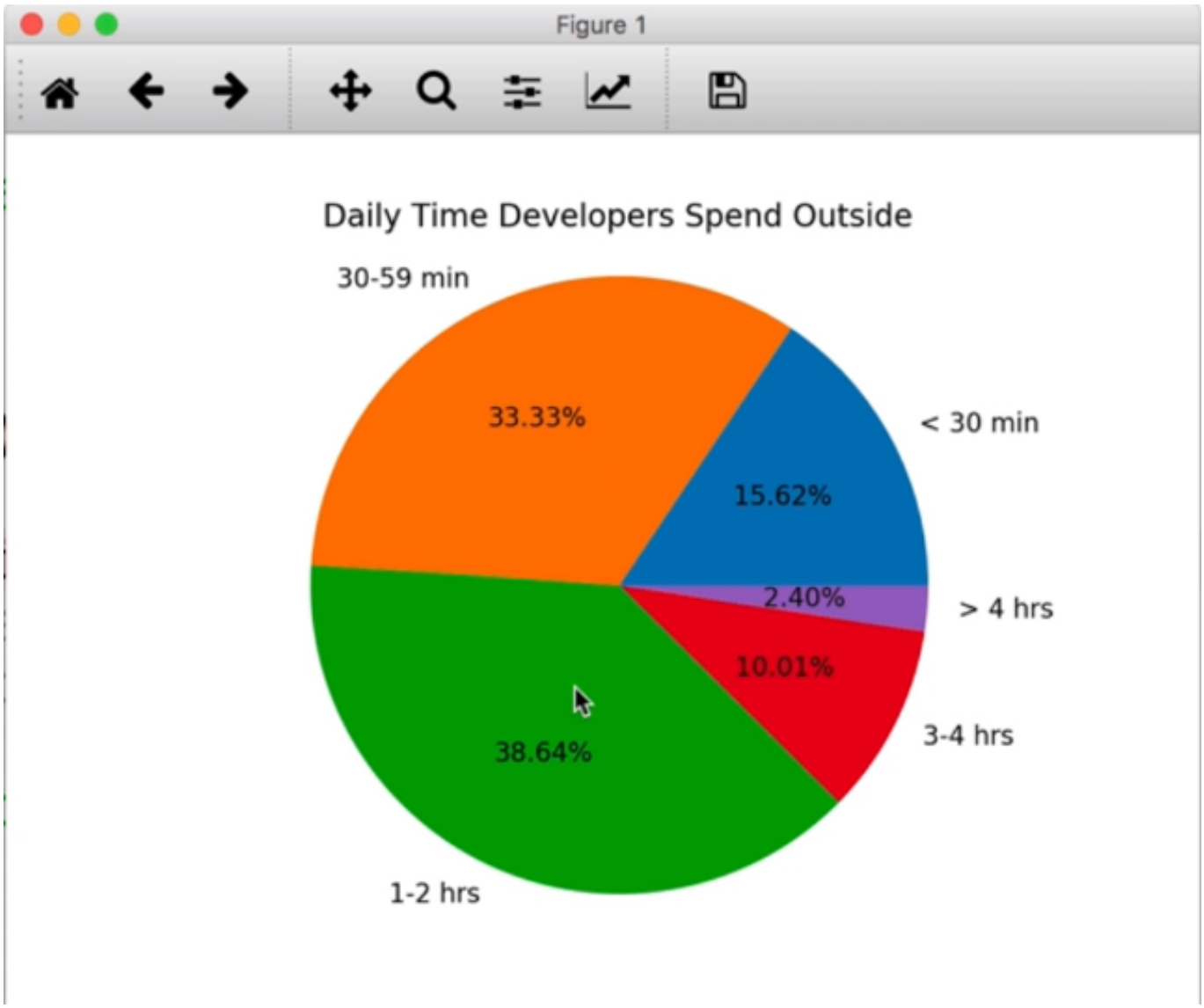
We can see that the majority of developers spend 1-2 hrs outside, slightly fewer spend 30-59 mins outside, very few developers spend > 4 hrs outside.

Lets add code to add the actual sizes inside the chart slices. Right now we are just comparing the relative size. Lets **modify** the above code.

```
# add format to display in slices – 2 decimal places
plt.pie(responses, labels=time, autopct='%.2f%%')
```

NOTE : %% is used to display the % sign after the number.

Let’s run this code.

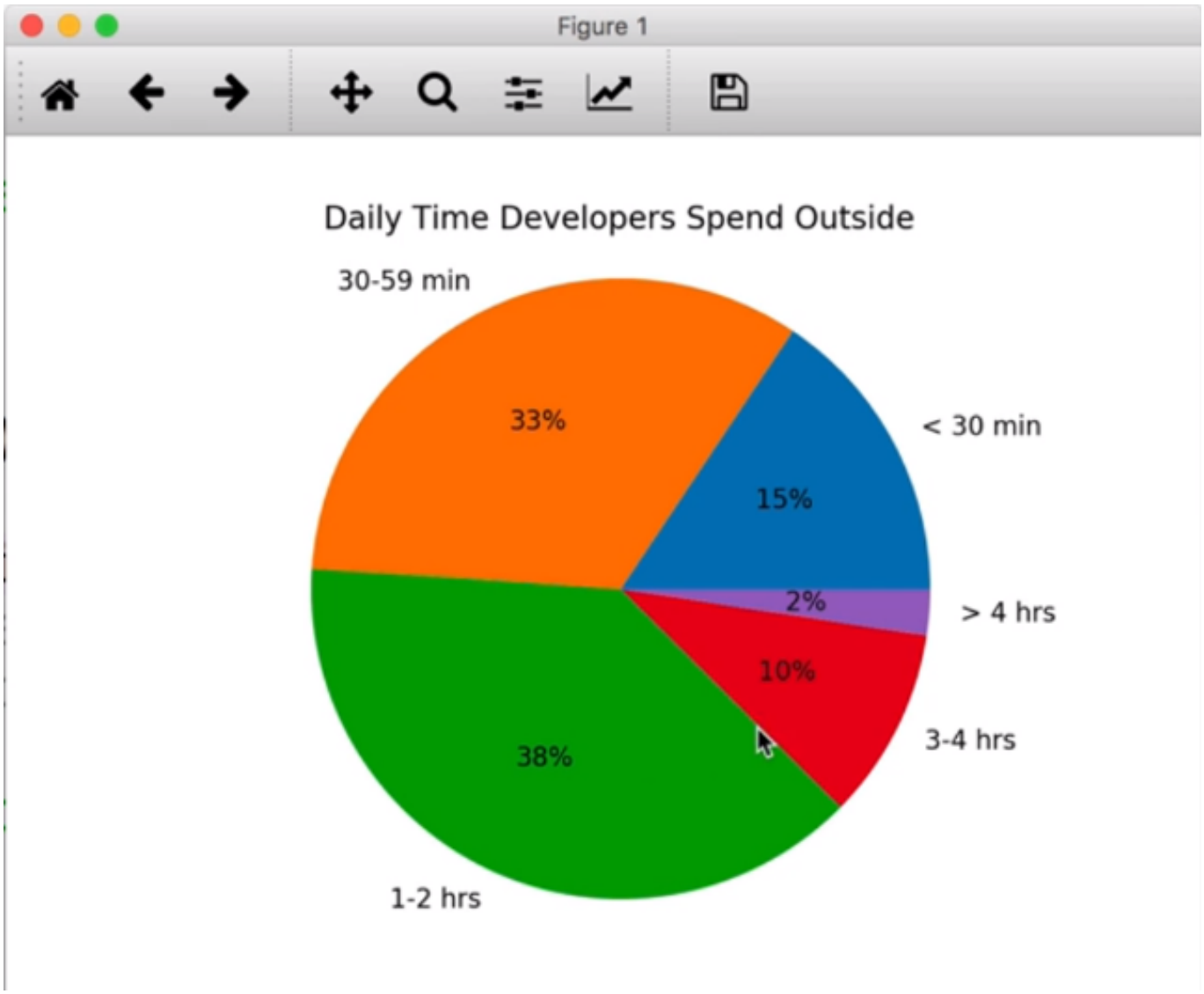


Now we can clearly see that while 38.64% of the developers spend 1-2 hrs outside, only 33.33% of the developers spend 30-59 minutes outside.

Lets change the display format above to an integer.

```
plt.pie(responses, labels=time, autopct='%d%%')
```

Let’s run this code.



We see the percentages displayed in integers. You can look at various formats in the official Python documentation.

Pie charts are useful when there are sections of a whole that you want to display. For example sections of your monthly budget that go into different expenditures could be displayed effectively with a pie chart. You can represent the data just as in the tuples above and matplotlib *pie* will know how to format and display the data as parts of a whole.

In this video, we are going to be looking at **line charts**. The data we are going to be using is ranking of programming languages in terms of popularity. I have pulled some data ranking Java, Python, C and C++ and we can see that it is actually a function of time i.e. the rankings change over time. This kind of data i.e. *time series data* is the kind of data that can be depicted really well using *line charts*.

Lets write some code.

```
import matplotlib.pyplot as plt
import pickle

# load data
with open('prog-langs-popularity.pickle', 'rb' as f) :
    data = pickle.load(f)

print (data)
```

Let’s look at our data by running this code.

```
In [1]: runfile('/
Users/presenter/
Developer/data-viz/
matplotlib/line.py',
wdir='/Users/
presenter/Developer/
data-viz/matplotlib')
[('Java', [(2018, 1),
(2013, 2), (2008, 1),
(2003, 1), (1998,
16)]), ('C', [(2018,
2), (2013, 1), (2008,
2), (2003, 2), (1998,
1), (1993, 1), (1988,
1)]), ('C++', [(2018,
3), (2013, 4), (2008,
3), (2003, 3), (1998,
2), (1993, 2), (1988,
4)]), ('Python',
[(2018, 4), (2013,
7), (2008, 6), (2003,
11), (1998, 23),
(1993, 17)])]
```

Our data is a bit complicated. We have a list of tuples that each look like:

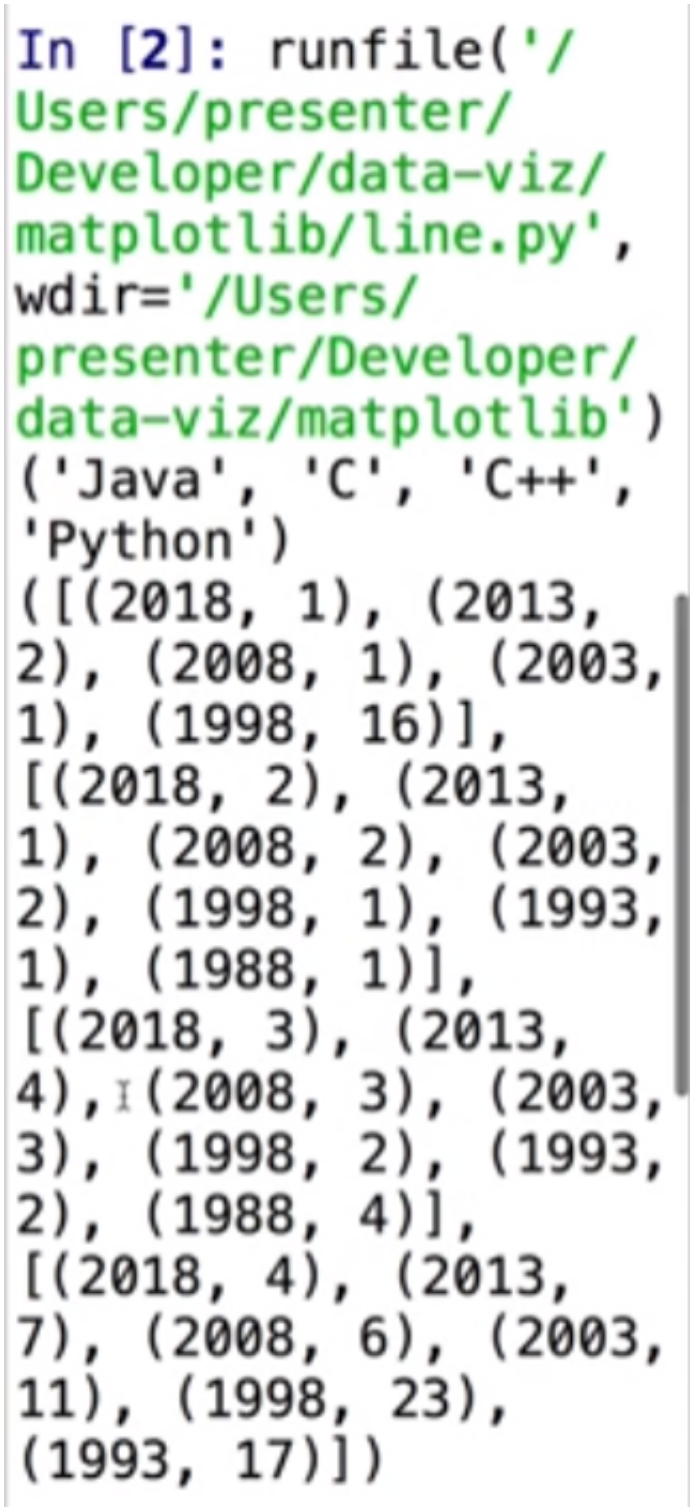
`(Java, [(2018, 1), (2013, 2), (2008, 1), (2003, 1), (1998, 16)])`

So we have **Java** which was ranked **1st** in 2018, 2008, 2003. It was ranked **2nd** in 2013 and **16th** in 1998.

At first we will only be visualizing the popularity of Java and later we'll look at multi-line plots with legends. Lets separate this data into 2 lists.


```
languages, rankings = zip(*data)

print (languages)
print (rankings)
```

Let's run this code.

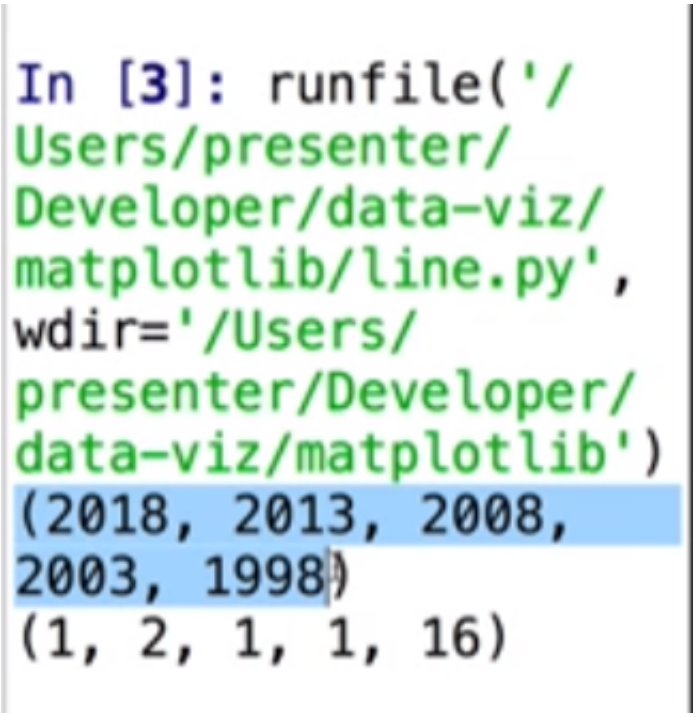


Our first list is the list of languages – **('Java', 'C', 'C++', 'Python')**. Our second list is a tuple of list elements, each element similar to **[(2018, 1), (2013, 2), (2008, 1), (2003, 1), (1998, 16)]**.

Lets get **only** the **Java** years and ranks.

```
# splitting Java years and ranks (split Java data into 2 lists)

java_years,  java_ranks = zip(*rankings[0])
print (java_years)
print (java_ranks)
```

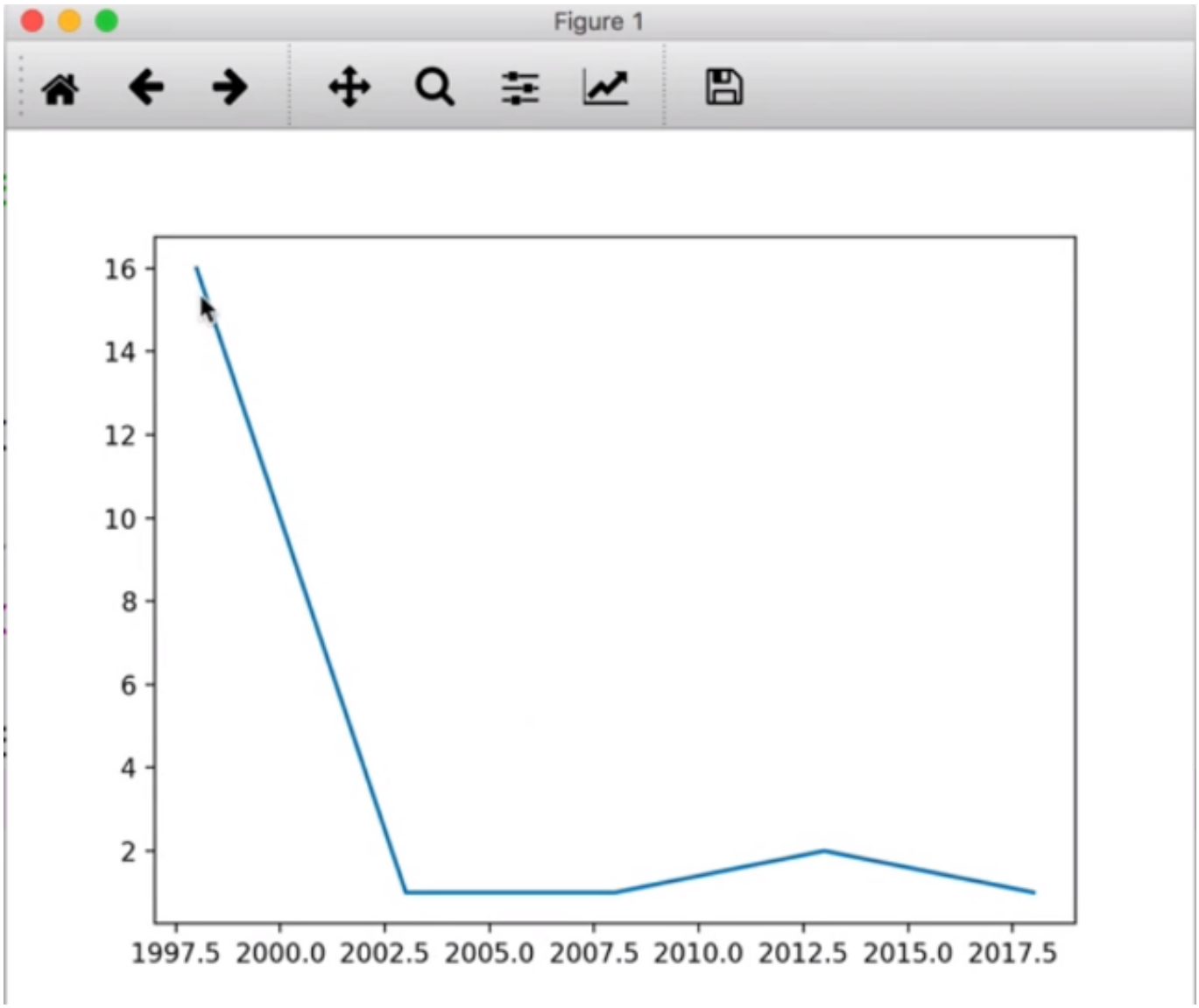
Let's run this code.



First we get the **Java years and then the rankings**.

Lets do a **line plot** of this data.

```
plt.plot(java_years, java_ranks)
plt.show()
```

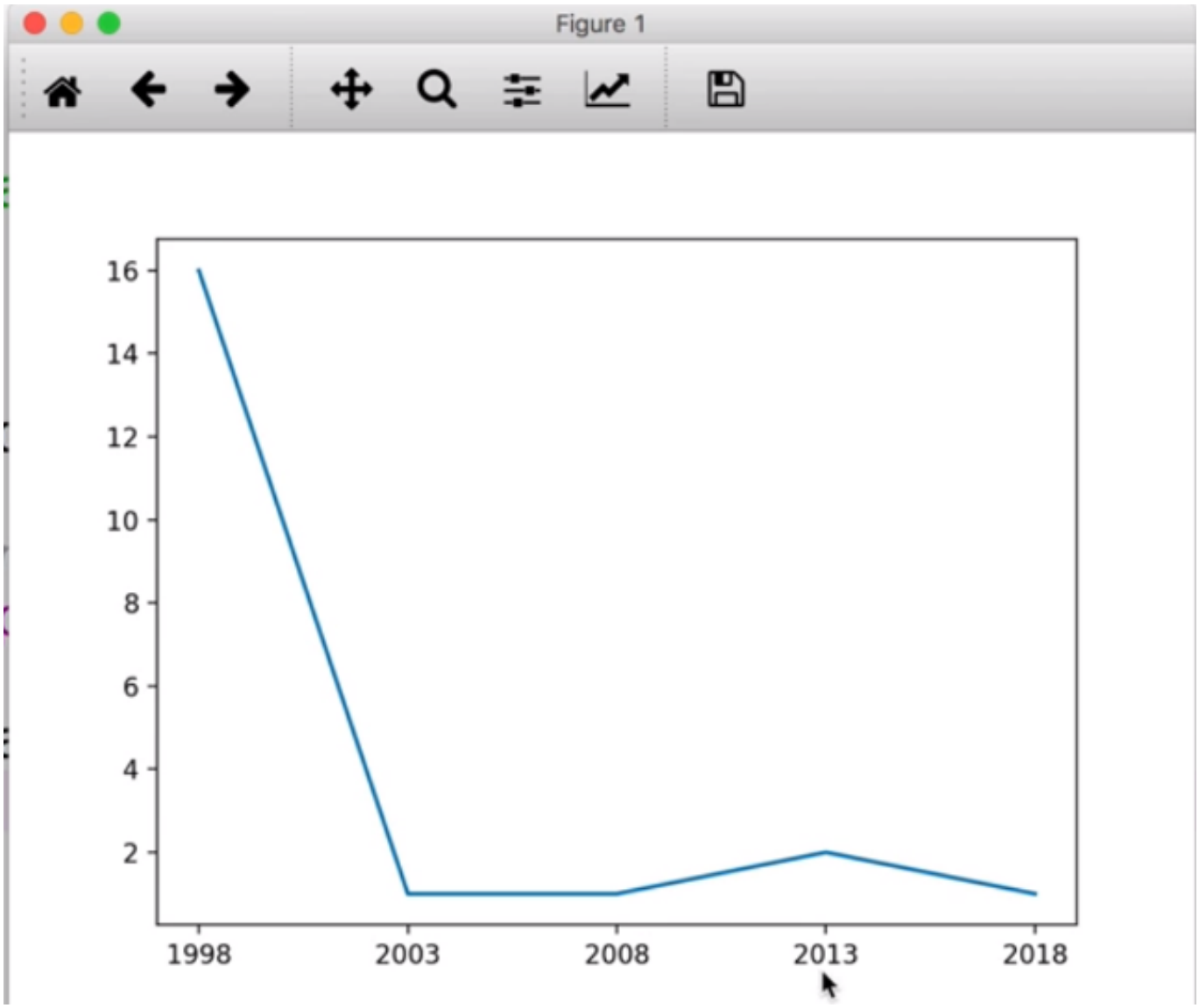
The **rank** is displayed on the y-axis and the **year** on the x-axis. The plot tells us that early on, Java was not ranks very highly. As the years progressed, it has jumped dramatically and is currently ranked quite high.

Notice the years are decimal values. We can force *matplotlib* to force the x-value to be exactly equal to the years.

Lets add this line before *plt.show()* above.

```
plt.xticks(java_years)
```

Running this code, we get a cleaner plot.

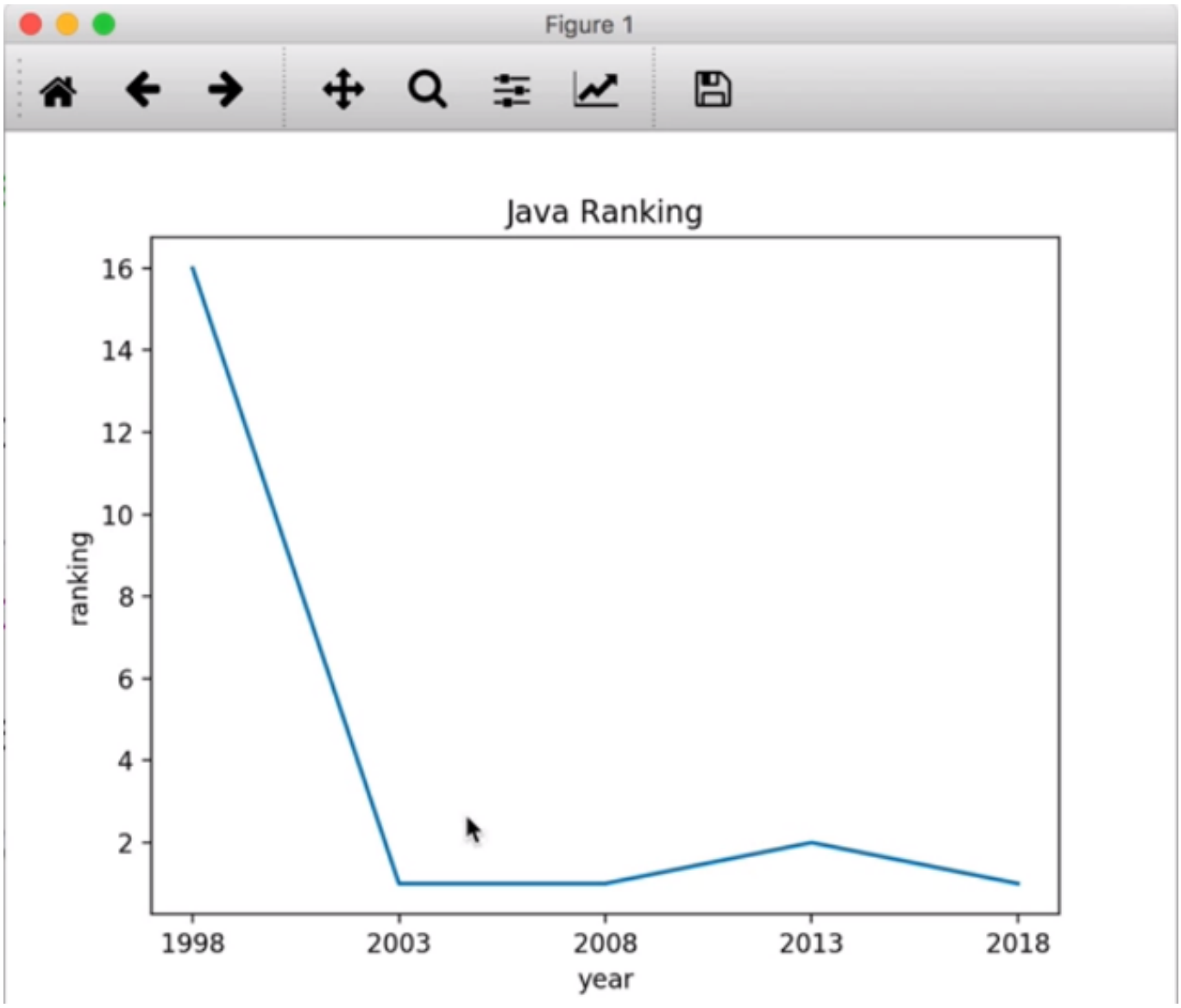


CHALLENGE : Write code to label the **x-axis (with 'year'), y-axis (with 'ranking')** and add a plot **title ('Java Ranking')**.

Add the following code before the *plt.show()* call.

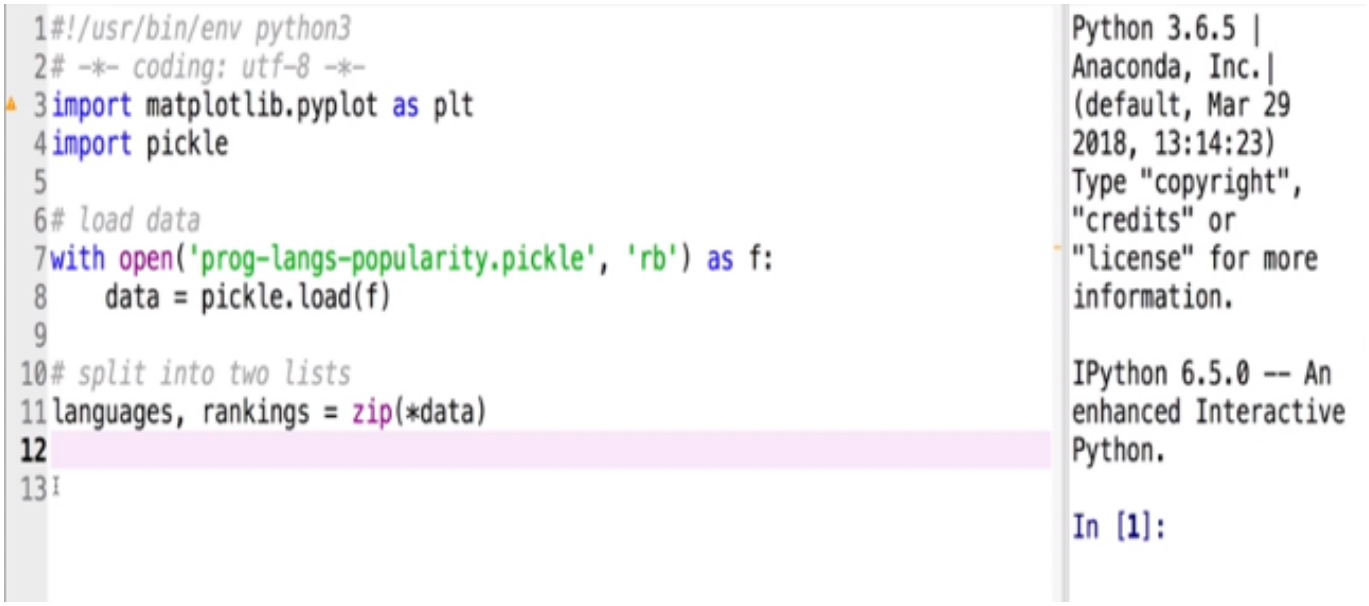
```
plt.xlabel('year')
plt.ylabel('ranking')
plt.title('Java Ranking')
```

Let's run the code to display our plot.



We can see that in 1998, Java had a rank of 16 but has fluctuated around 1 and 2 since 2003.

In the video, we will look at how to plot multiple lines.

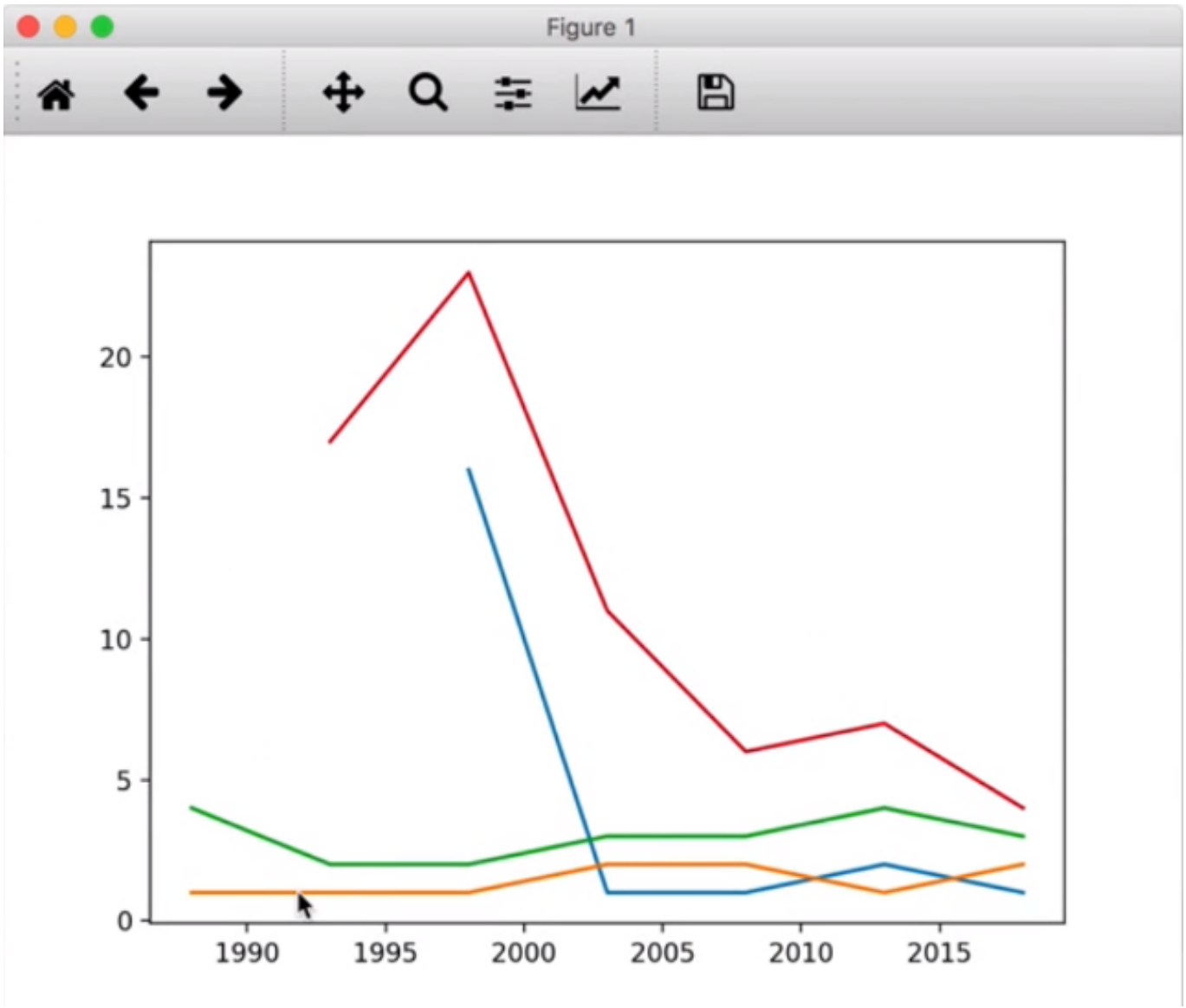


We've copied the code above from the previous lesson's *line.py*, renamed it to *multiline.py*. Here, i want to plot the data for all of the languages, not just Java.

In *matplotlib*, every time you call *plt.plot()*, you create a new line. So i need to iterate over all the languages and call *plt.plot()* for each of them.

```
# iterate over all the languages and call plot() on their data.

for i in range (len(languages)) :
    # for each language, split their data into years and rankings lists
    years, rankings = zip(*rankings[i])
    plt.plot(years, ranks)

plt.show()
```

Let's run this code.

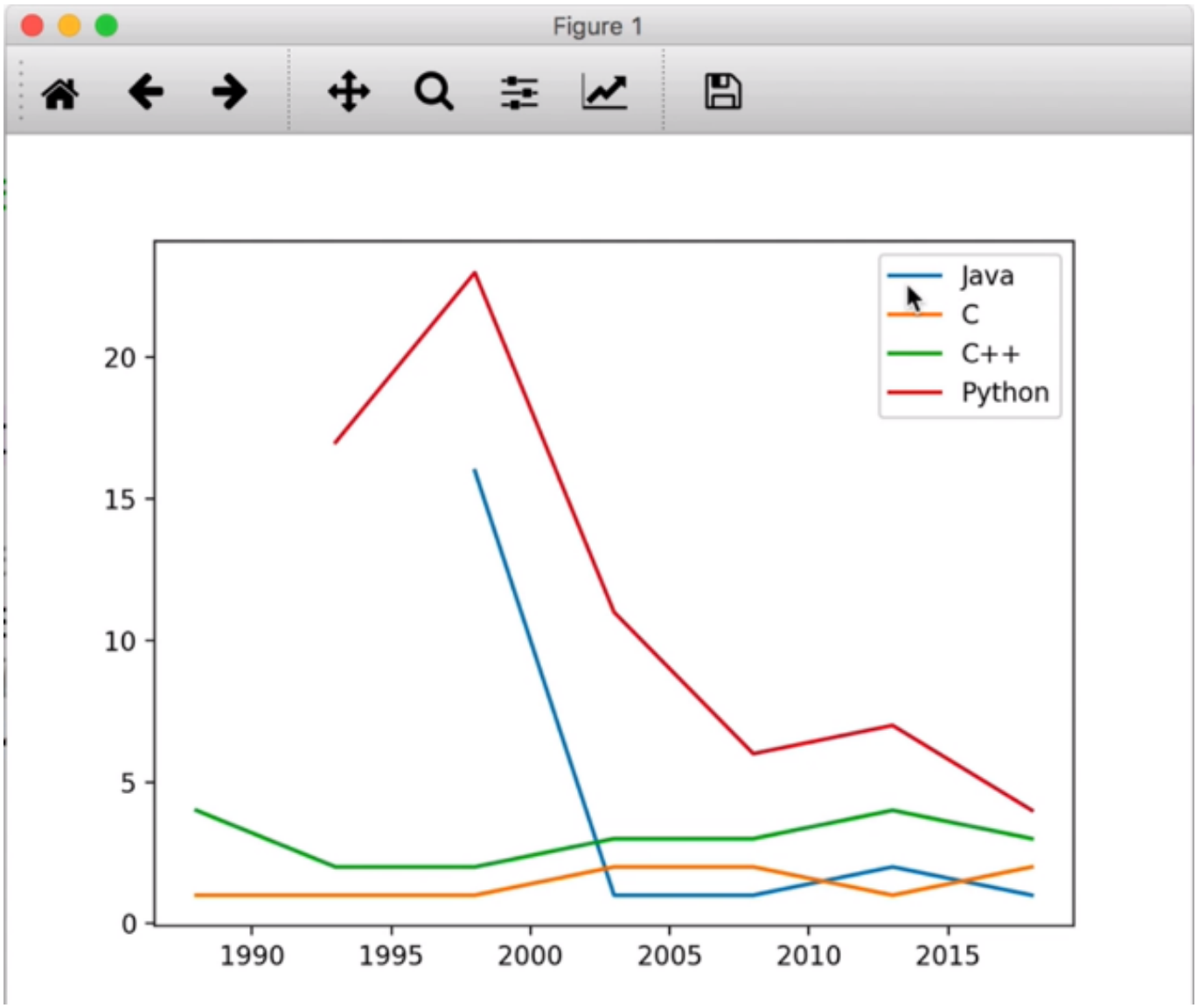


We can see different lines on this plot, colored differently.

Lets add a legend so we know which line corresponds to which language.

```
# languages ordering retained by zip
plt.legend(languages)
```

Let’s run this code.

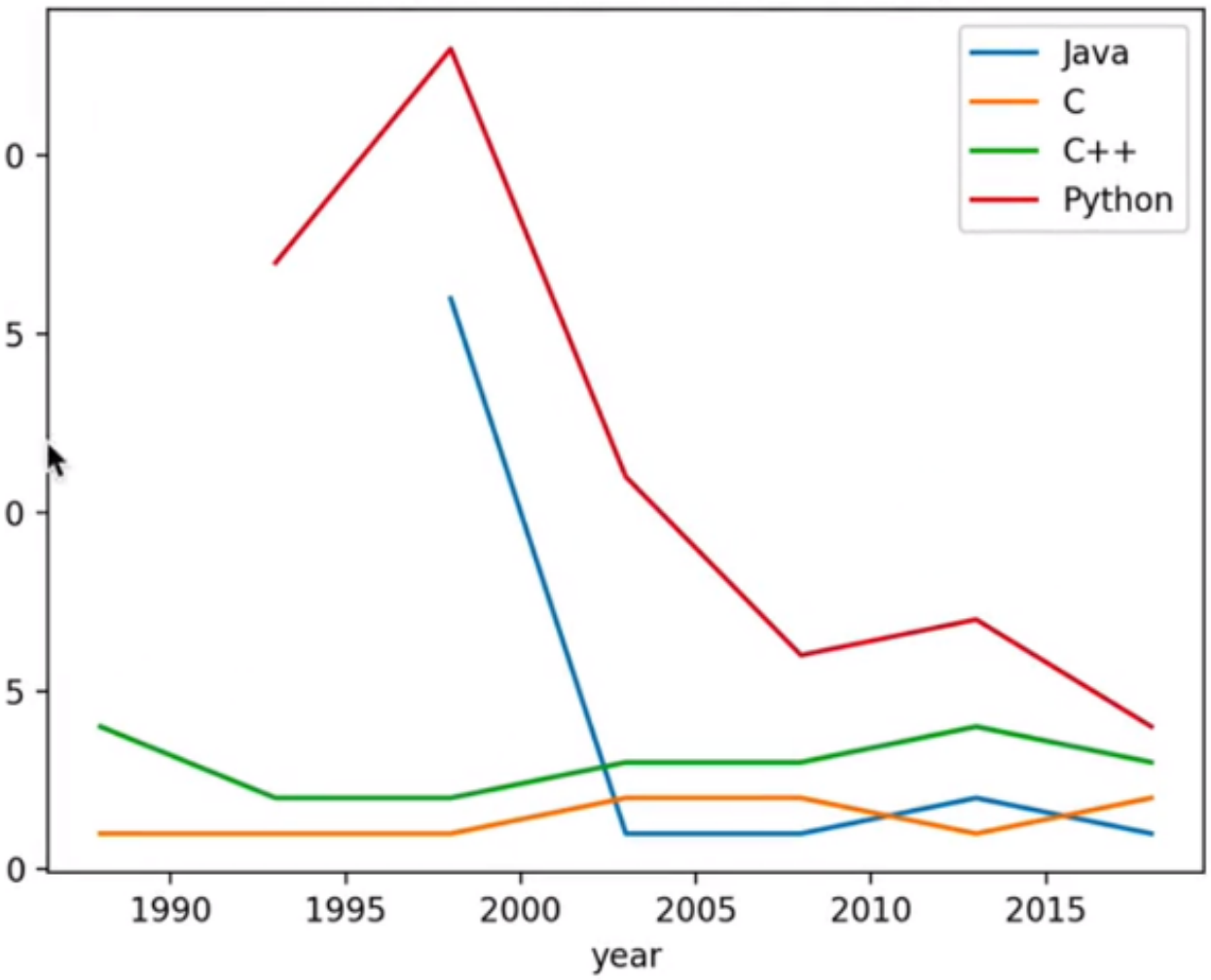


So **Java** was around 16, then fluctuates around 1 and 2. **C**, on the other hand, stays fairly consistent around 1 or 2, **C++** is consistent around 3 or 4. **Python** was popular, then lost popularity and has now become more popular.

EXERCISE : Write code to label our axes. Add this code **before** *plt.show()* above.

```
# x-axis= year, y-axis=ranking, title=Rankings of Programming Languages
plt.xlabel('year')
plt.ylabel('ranking')
plt.title('Rankings of  Programming Languages')
```

Lets run this code.



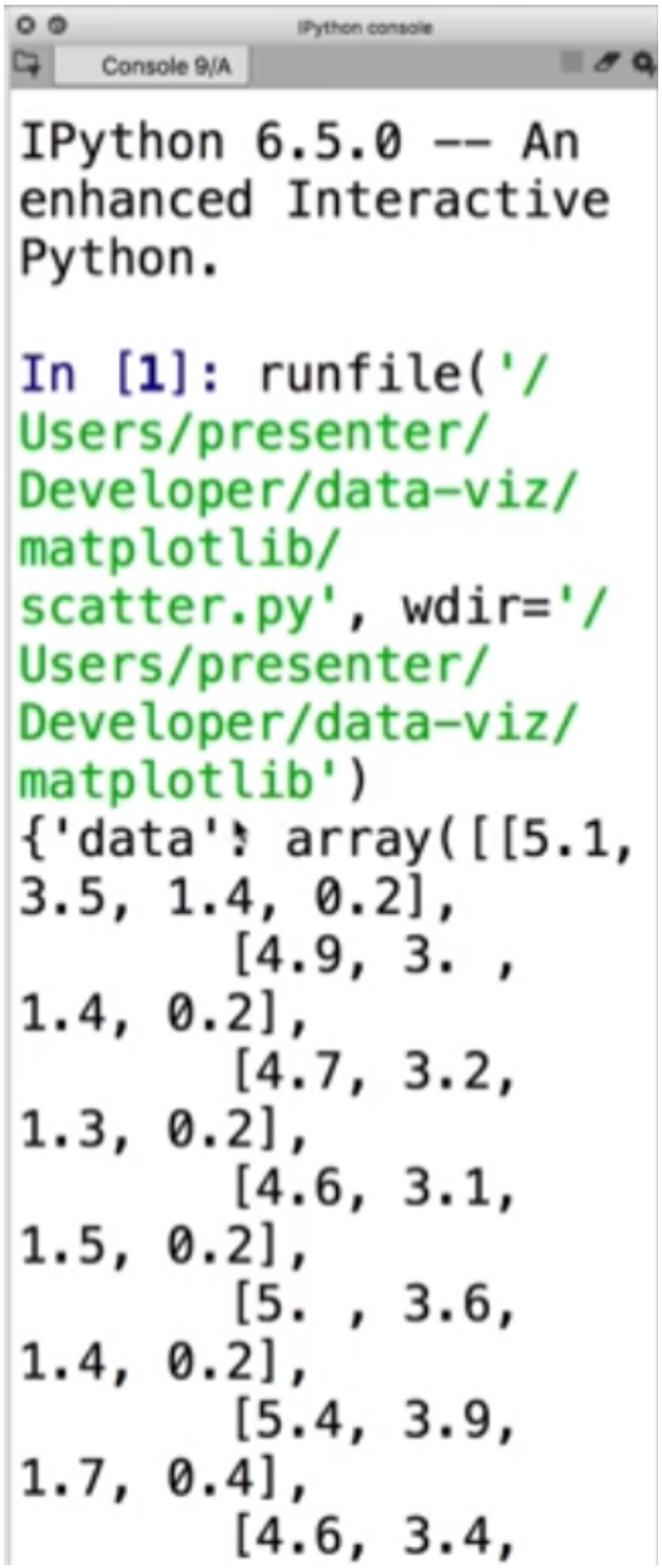
We can see our labeled axis and the title.

Summary

Line plots work great for plots for data in a time series or with a time element and we can plot multiple lines on a single plot. Here we can compare different programming languages. We achieve that by calling *plt.plot()* multiple times. Each time, it creates a new line.

In this video, I am going to show you how to display a **scatter plot** in *matplotlib*. The data set we are going to be using is a famous data set called Iris. This data has to do with flowers. We measure the **petal length, width, the sepal length and width** of a particular **species** of flowers. This data set has been used widely. The author used it to demonstrate his dimensionality and discriminate analysis techniques. We're going to be plotting some of that data to see what it looks like.

```
import matplotlib.pyplot as plt
import pickle

# load data
with open('iris.pickle', 'rb') as f:
    iris = pickle.load(f)

print (iris)
```

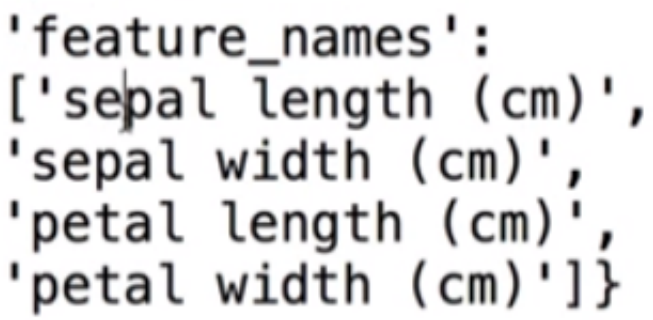


Lets understand the data. The '*data*' key is a table where each **row** represents a different **flower** being measured. Each of the **columns** represents a different **property** of the flower being measured.

e.g. Lets look at the list [5.1, 3.5, 1.4, 0.2]. Here 5.1 is the **sepal length**, 3.5 is the **sepal width**, 1.4

is the **petal length**, 0.2 is the **petal width**.

Scrolling down the above output, we see this key called *feature_names*.

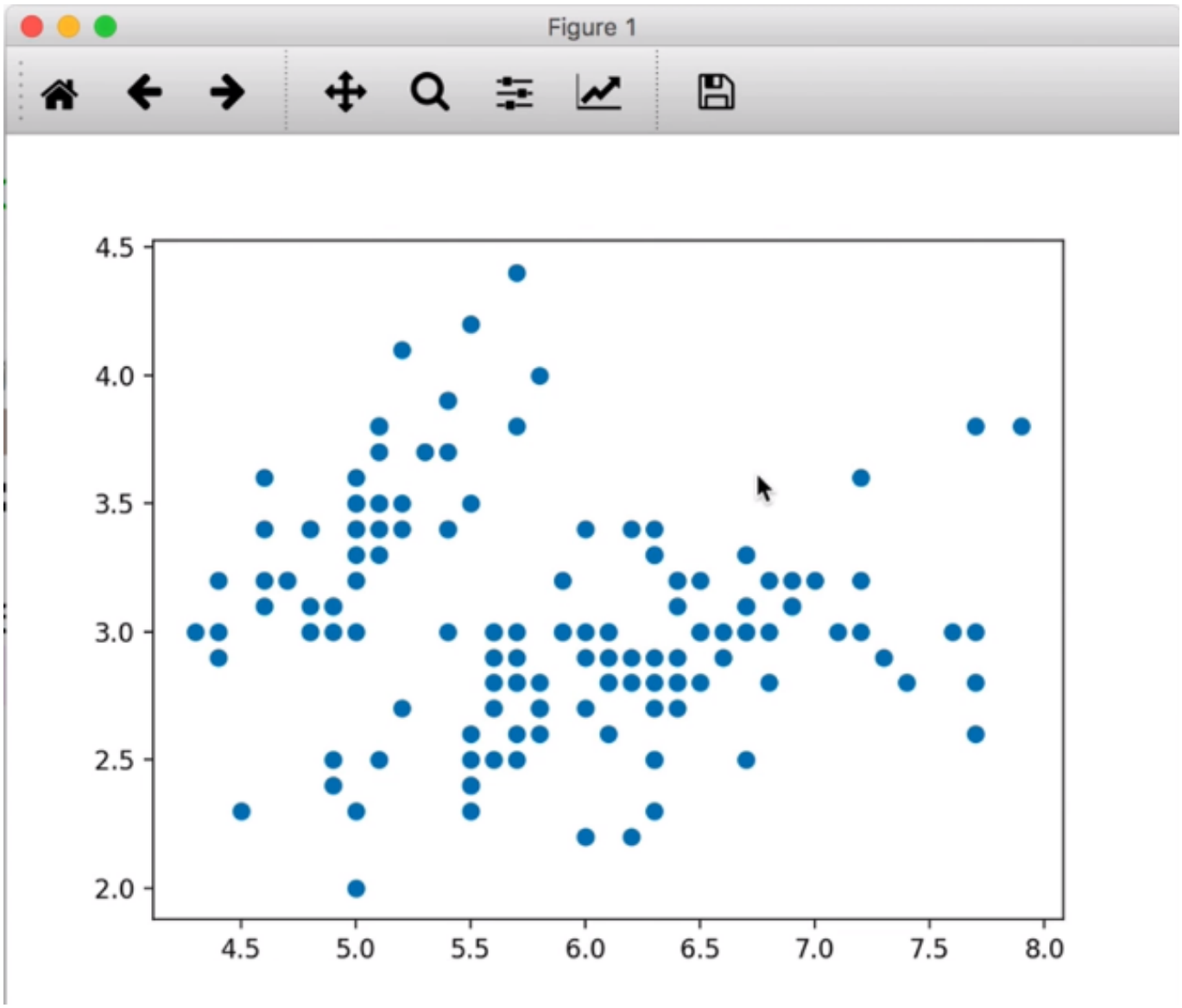


We will be plotting the **sepal length and sepal width**.

Lets get these columns from our data. Our data has 150 rows and 4 columns. 'sepal length' is the first column and 'sepal width' is the second column.

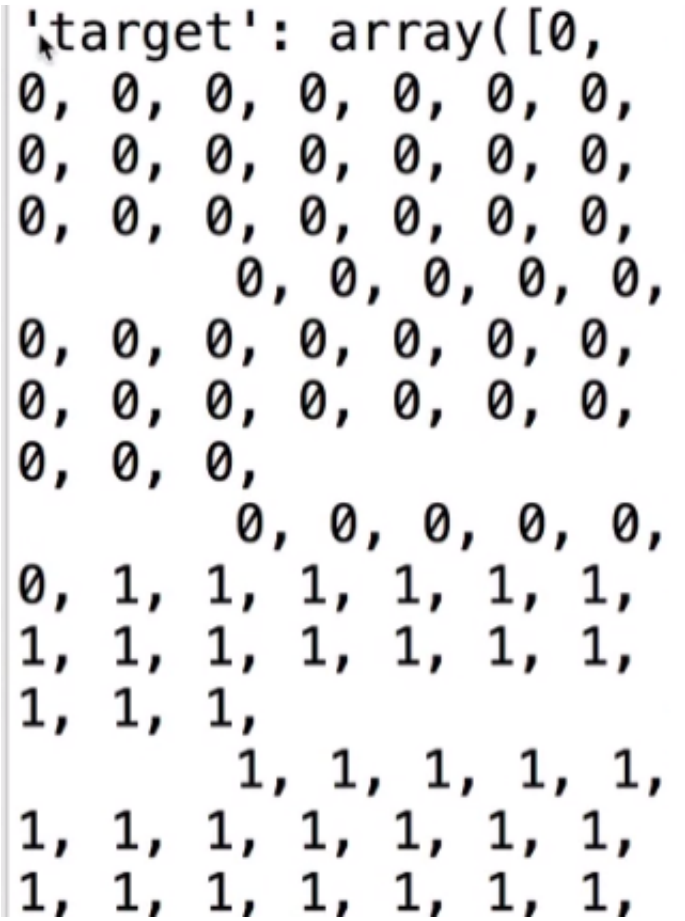
```
# extract first, second columns from the data table (get all of the rows)
sepal_length = iris['data'][:, 0]
sepal_width = iris['data'][:, 1]

# scatter plot
plt.scatter(sepal_length, sepal_width)

# We expect to see 150 data points, each point representing 2 different properties of
 1 flower.
plt.show()
```

Lets run this code.

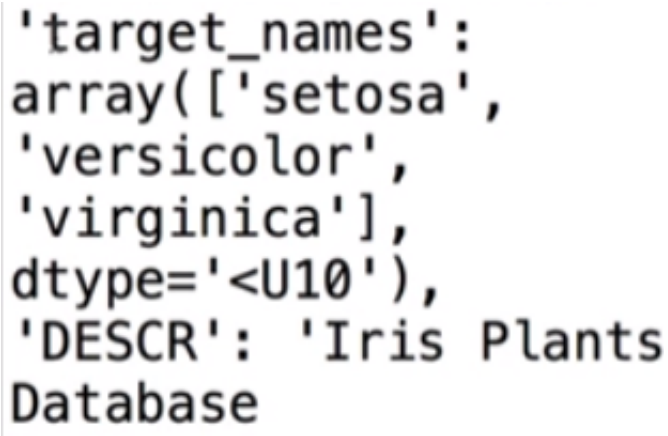


The Iris data set is a collection of 3 different species of flowers. However I don't know from the above plot, which point represents which species. I would like them to be colored differently based on their species.

Going back to our data output, we see key called '**target**'.



These show the different classes of flowers using the values 0, 1, 2. To see what they represent, lets look at the data. It contains a key called **target_names** that represents the 3 different types of flowers.



Lets extract the target array from our data. Add this code just **above** the *plt.scatter()* call above.

```
classes = iris['target']
```

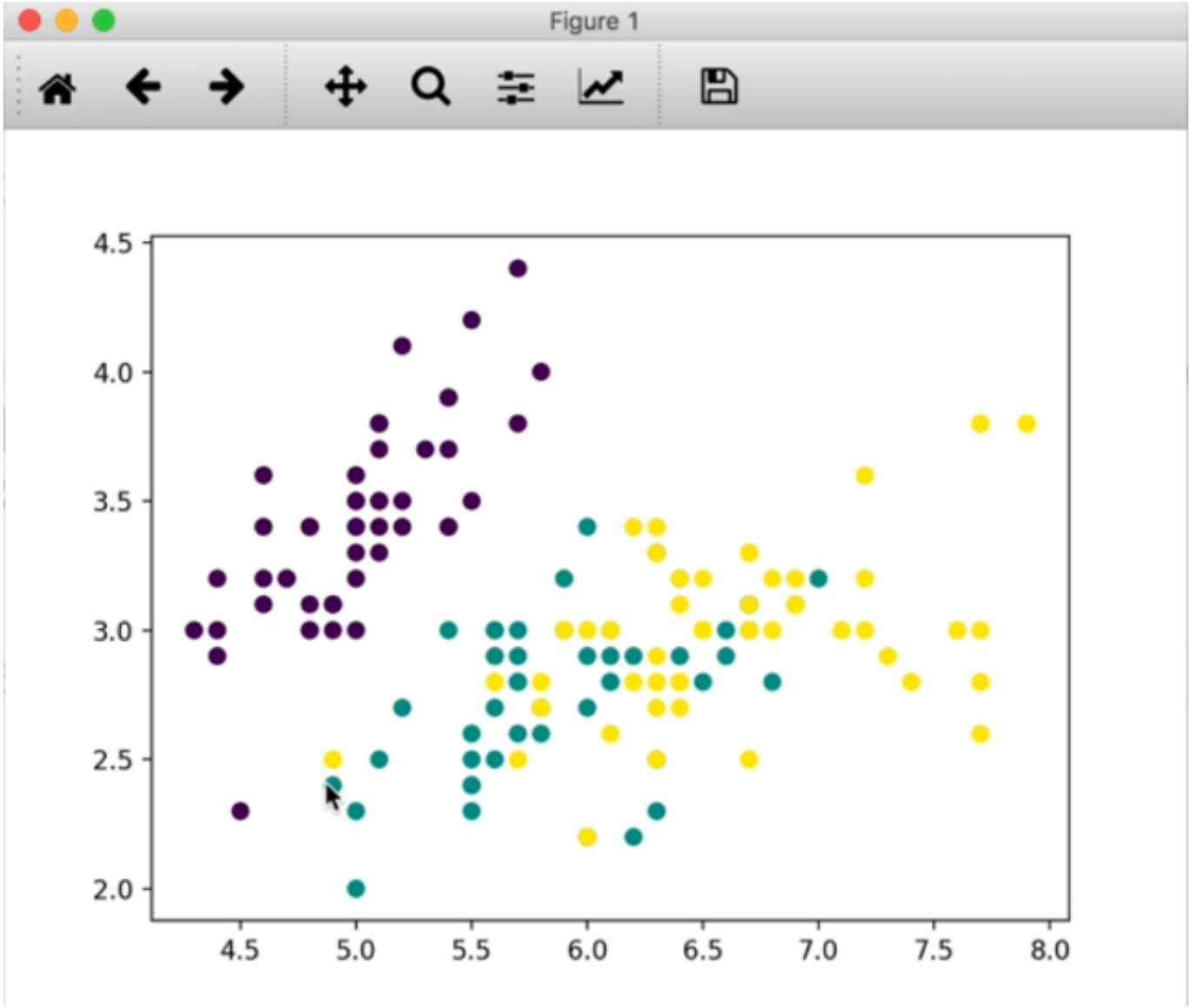
Let's **modify** the call to *plt.scatter()* as follows.

```
# 150 points implies 150 targets
```



```
plt.scatter(sepal_length, sepal_width, c=classes)
```

Let's run this code.

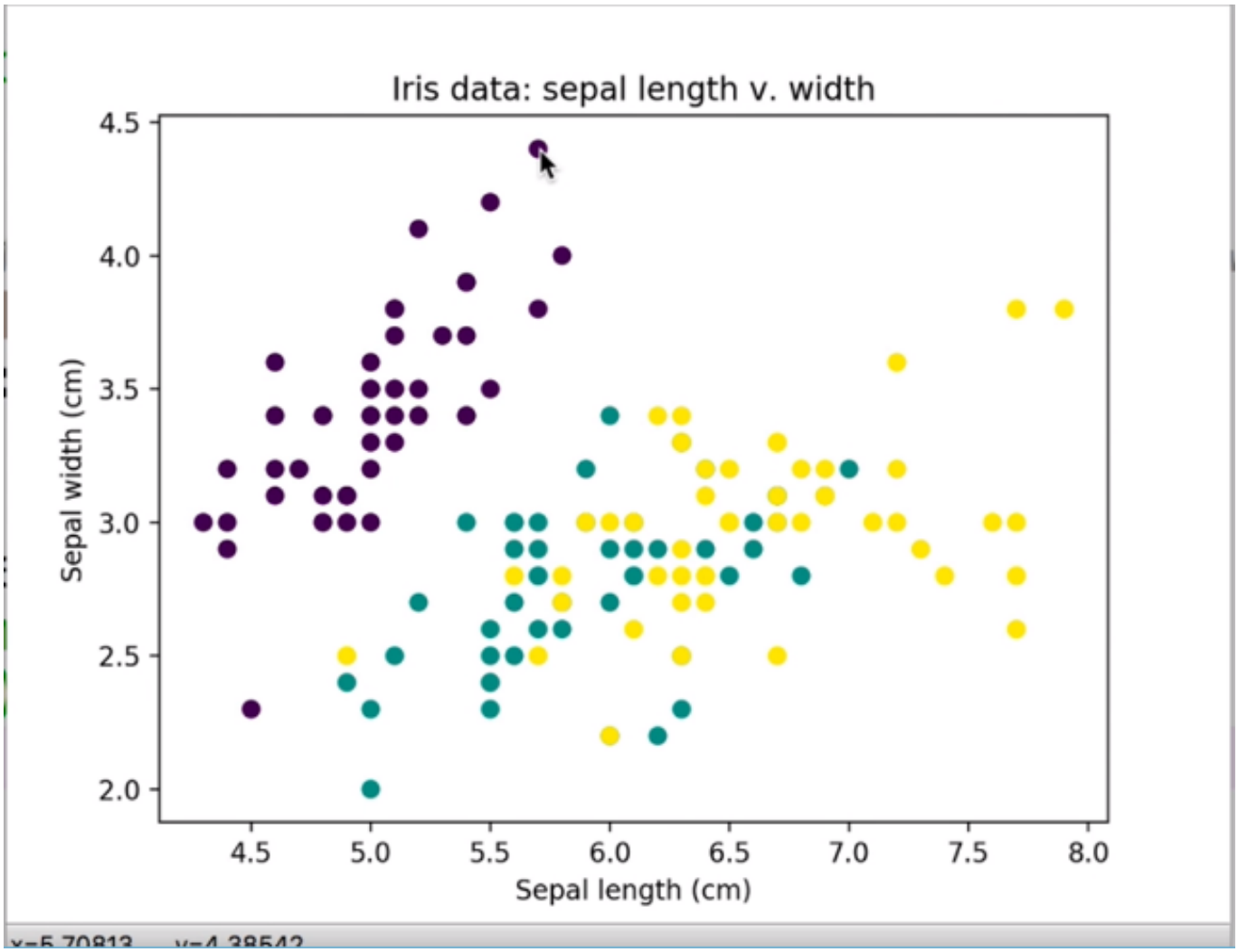


We see the points colored differently based on their class.

Let's label our axes.

```
plt.xlabel('Sepal length(cm)')
plt.ylabel('Sepal width(cm)')
plt.title('Iris data: sepal length v. width')
```

Let's run this code.

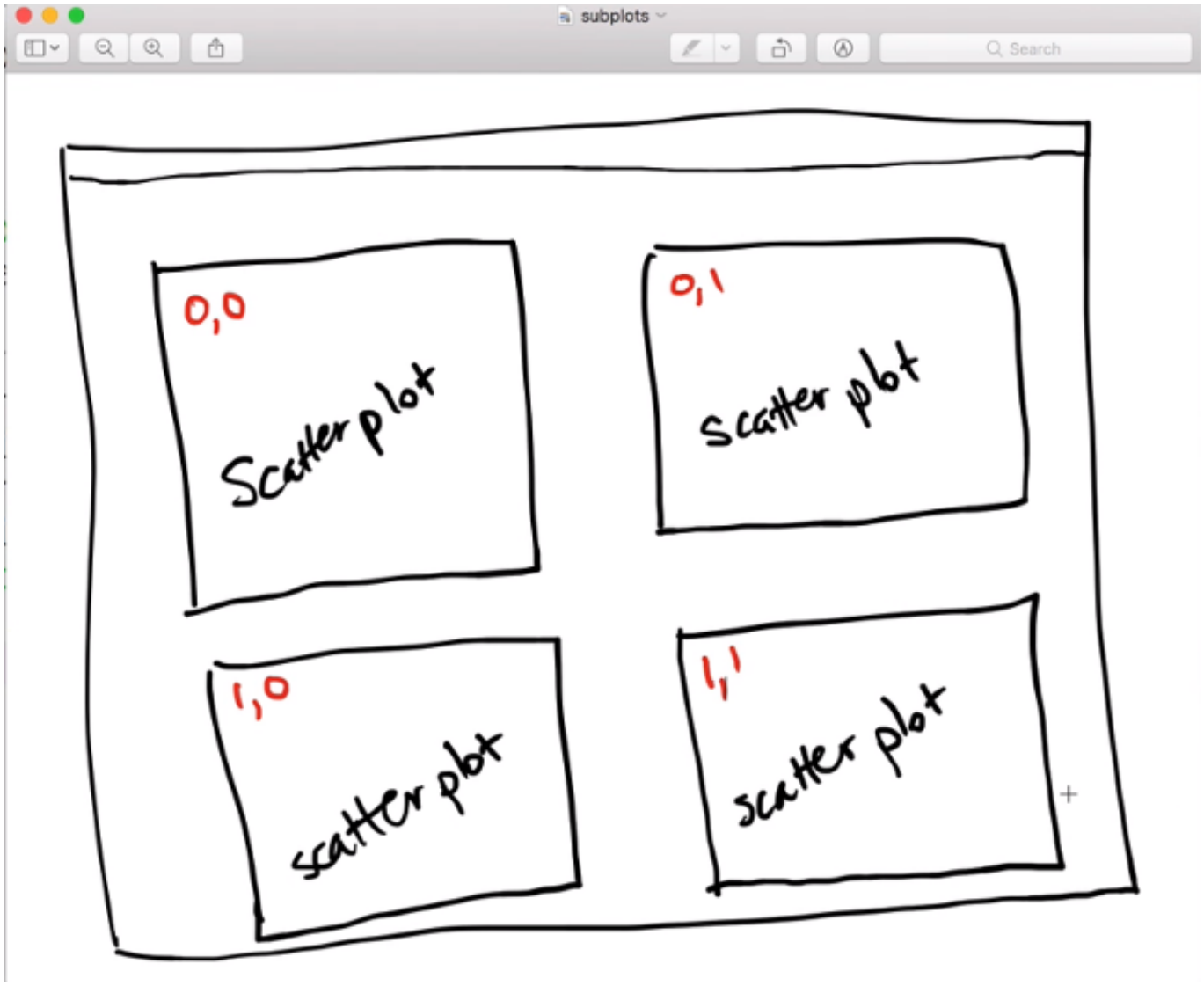


If we highlight a point, we can see its x and y coordinates below the plot. In the above plot, the highlighted point has a sepal length of 5.708 cms and a sepal width of 4.385 cms. We can analyze the data using this plot. For example, the purple colored points represent flowers that have **longer** sepal **width** than sepal length.

Summary

We looked at **scatter** plots using *matplotlib*. Scatter plots work for different kinds of data including time series data. Here we are using them to plot 2 different feature of a data set. Each point in the above plot represents a flower.

In this video, I want to explain how to create multiple plots as depicted in the diagram below.



It's sometimes useful to have plots organized in this grid-like fashion and each plot will show different types of data.

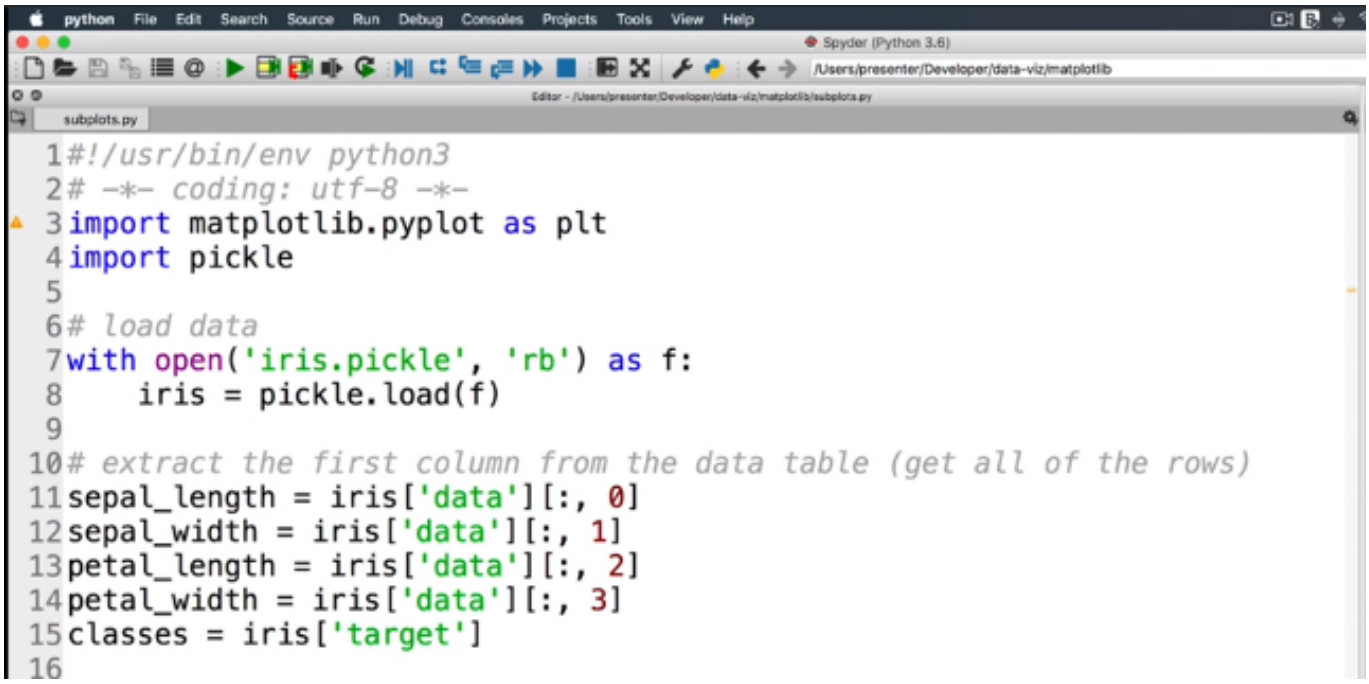
We are going to use the Iris data set. In this data set, there are 4 different features for each flowers – *sepal length, sepal width, petal length and petal width*. We can use this grid display to show different plots e.g. **sepal length Vs sepal width**, petal length Vs petal width etc.

Sub Plots

Each of the smaller plots inside the larger 2×2 plot (depicted above) is called a sub-plot. *Matplotlib* creates this ***axes*** object which allows to index into and access a particular plot. For example, in the diagram above, 0,0 depicts the 0th – row and 0th column. Similarly 1,0 depicts the 1st row and 0th column.

Though all the plots in the above diagram are labeled scatter plots, they don't have to be.

Let's write code to generate these sub-plots. Let's start with the following code copied over from the scatter plot lesson.

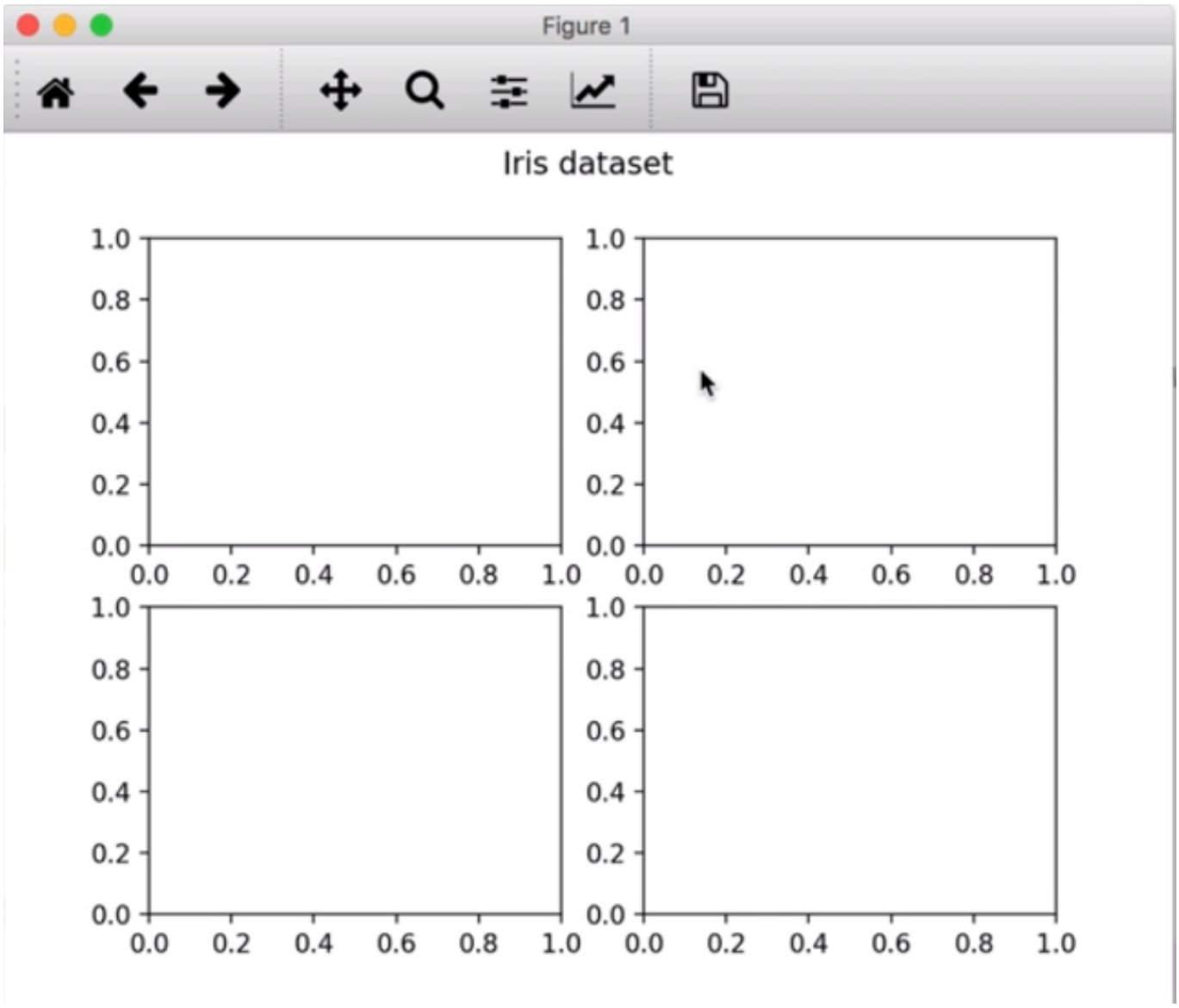


The code above extracts the sepal length, sepal width, petal length and petal width from the data.

Lets now create a 2×2 grid.

```
# fig - handle to the entire window
# axes – handle to individual sub-plots
fig, axes = plt.subplots(2, 2)

# title for entire plot
fig.suptitle('Iris dataset')
plt.show()
```

Let’s run this code.

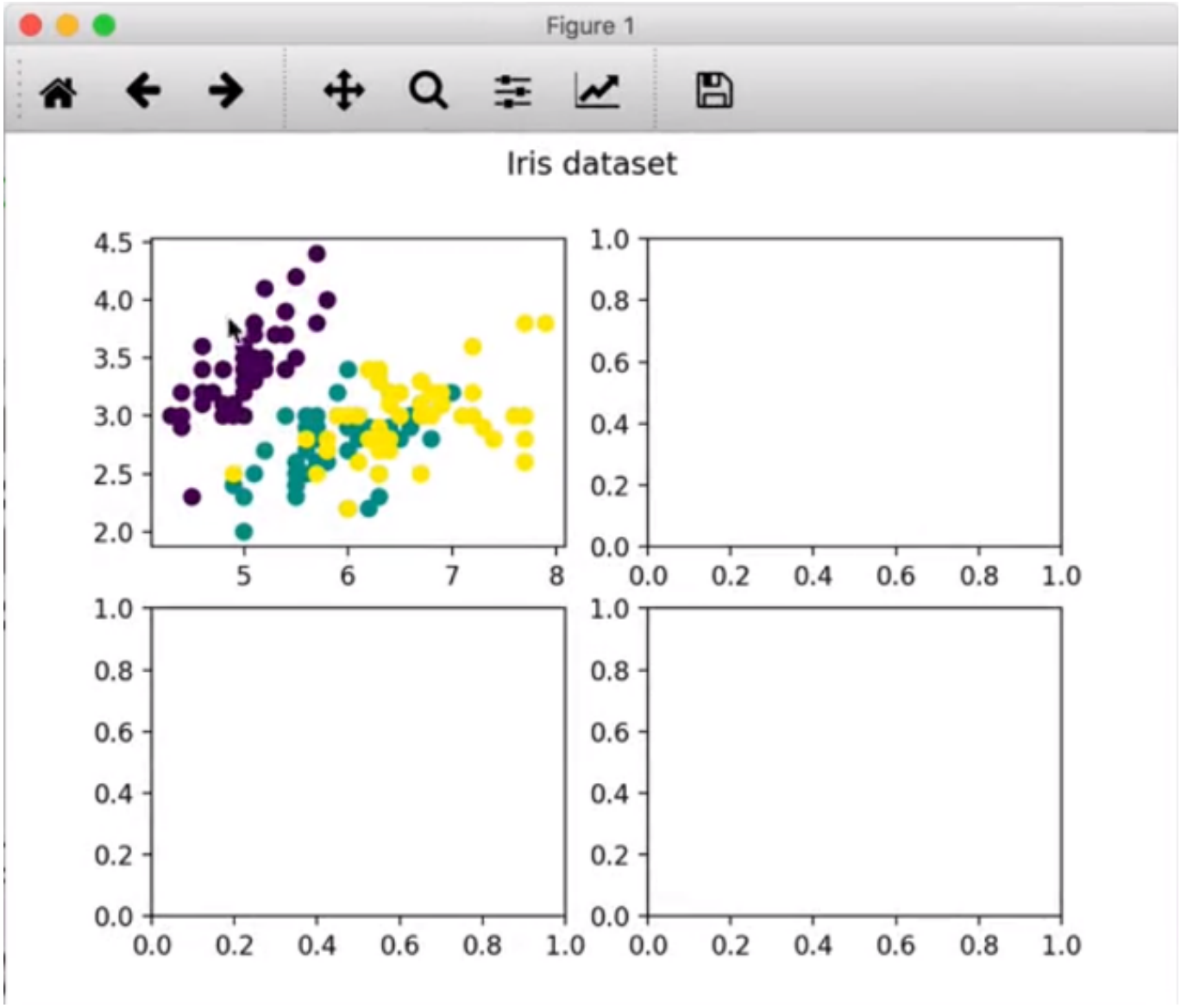


We see the 2×2 grid with the title.

Lets populate this with data. Add this code right after the *plt.subplots()* call.

```
# top left – plot sepal length Vs sepal width

axes[0,0].scatter(sepal_length, sepal_width, c=classes)
```

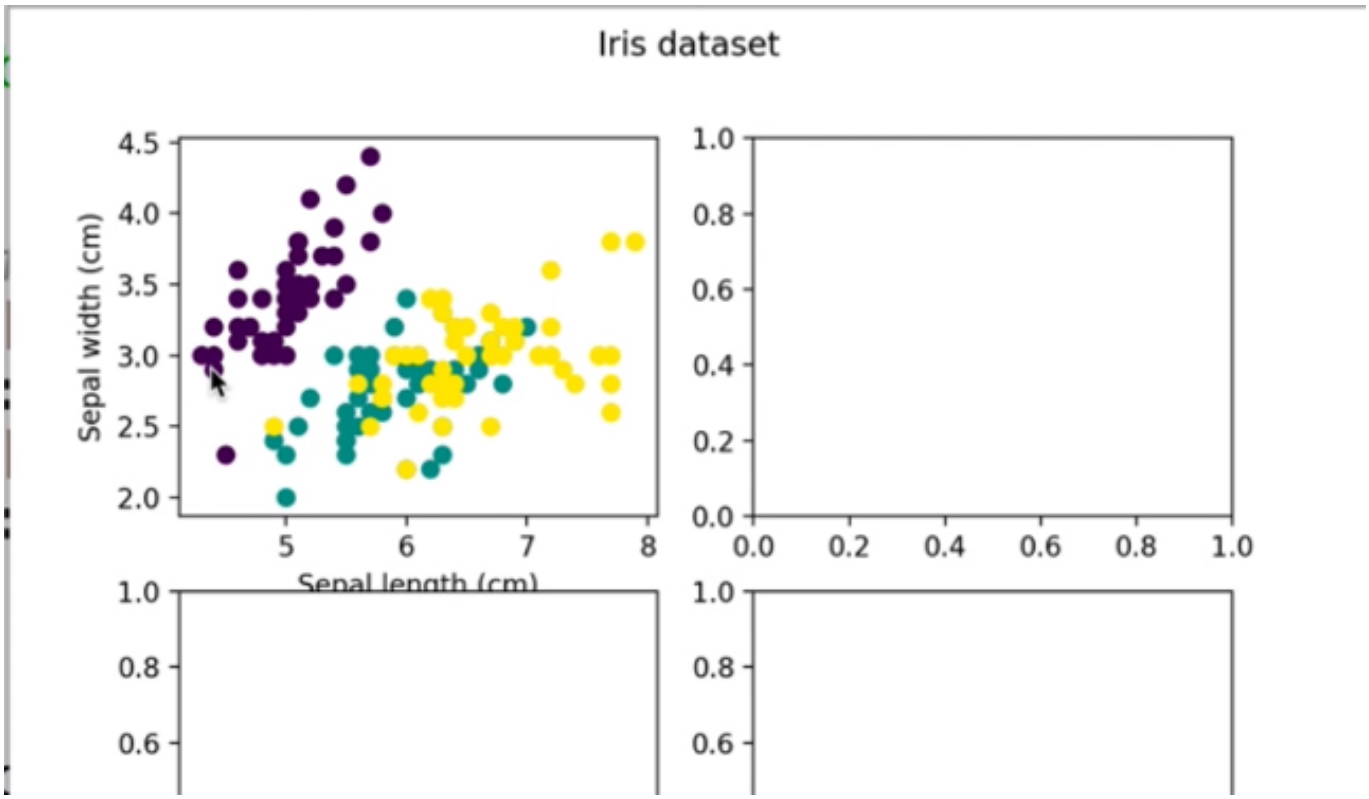
Let’s run this code.



We see the chart populated in the top-left sub-plot. Lets label the axes of the sub-plot.

```
axes[0,0].set_xlabel('Sepal length (cm)')
axes[0,0].set_ylabel('Sepal width (cm)')
```

Let’s run this code.

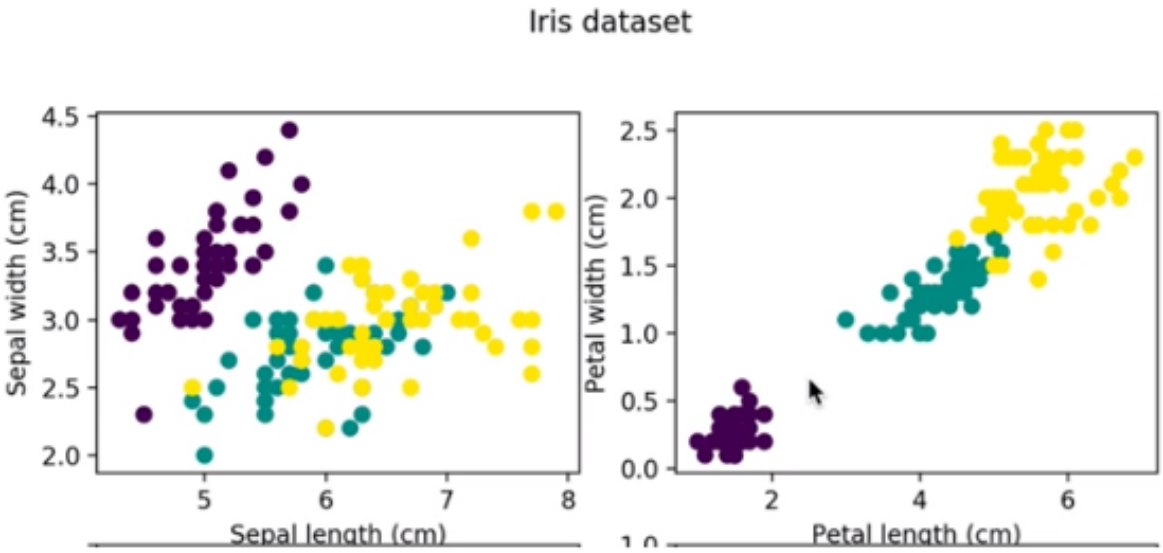


We can see the axis labels displayed on the first sub plot.

EXERCISE : Write the code that displays the sub plot in the 1st row, 2nd column. We want to plot the petal length Vs the petal width.

```
# top right  – plot petal length Vs petal width

axes[0,1].scatter(petal_length, petal_width, c=classes)
axes[0,1].set_xlabel('Petal length (cm)')
axes[0,1].set_ylabel('Petal width (cm)')
```

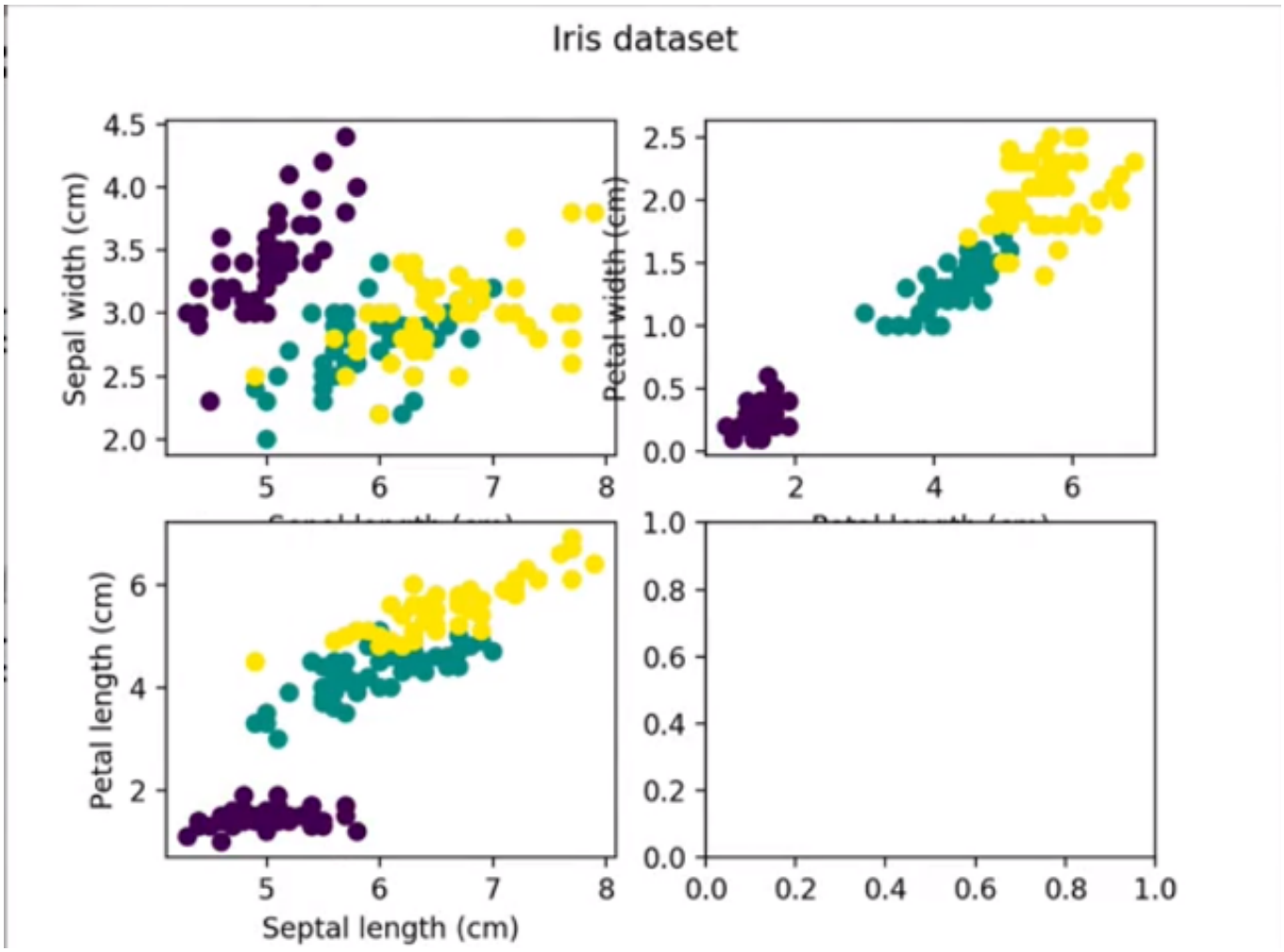
Let’s run this code.



EXERCISE : Let's now plot the bottom left chart (2nd row, 1st column). We want to plot sepal length Vs the petal length.

```
# bottom left  – plot sepal length Vs petal length

axes[1,0].scatter(sepal_length, petal_length, c=classes)
axes[1,0].set_xlabel('Sepal length (cm)')
axes[1,0].set_ylabel('Petal length  (cm)')
```

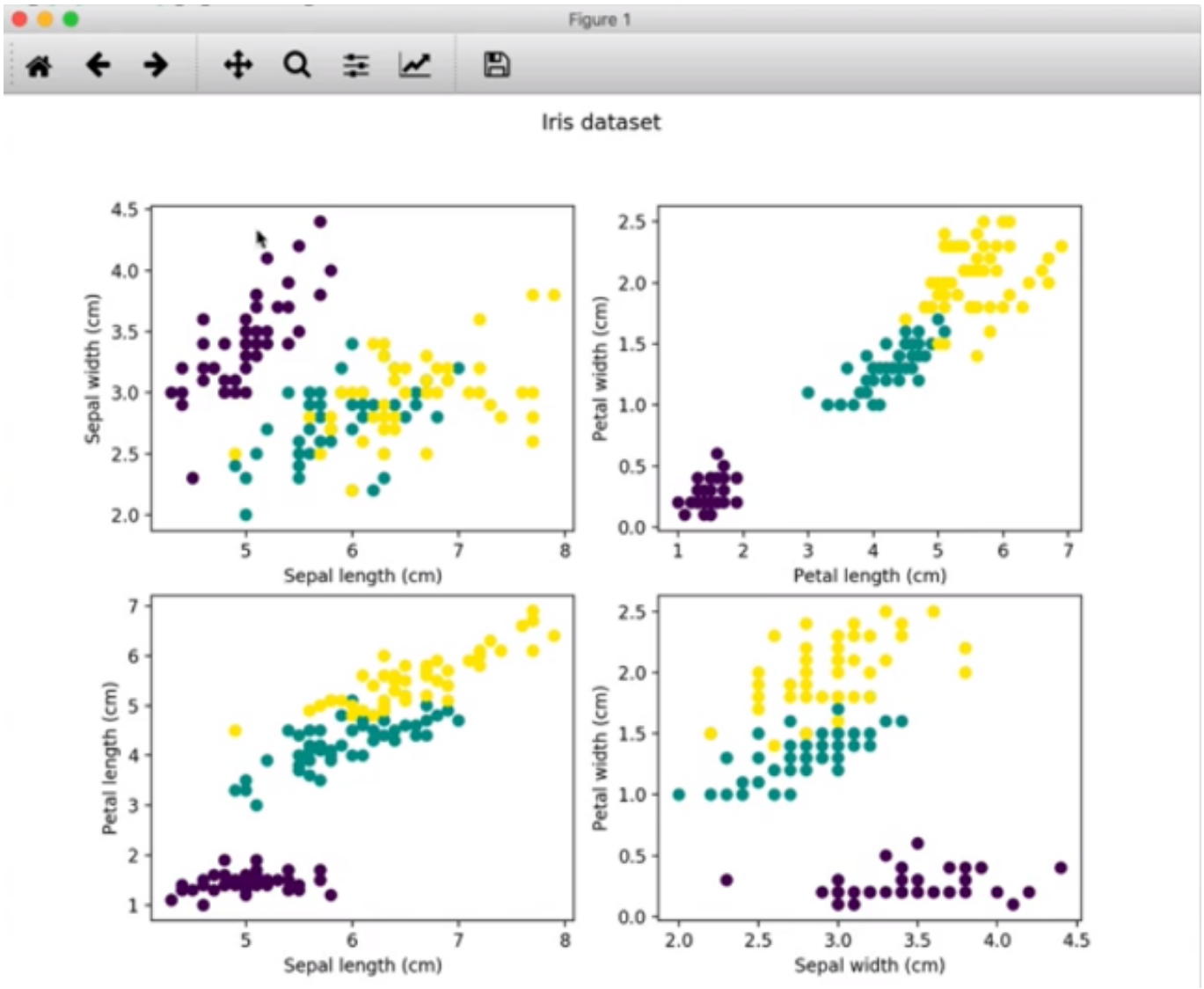
Let's run this code.



EXERCISE : Let's now plot the bottom right chart (2nd row, 2nd column). We want to plot sepal width Vs the petal width.

```
# bottom left  – plot sepal width Vs petal width

axes[1,1].scatter(sepal_width, petal_width, c=classes)
axes[1,1].set_xlabel('Sepal width (cm)')
axes[1,1].set_ylabel('Petal width  (cm)')
```

Let's run this code.



Now I can see all the sub-plots with labeled axes. We see different views of the same underlying data.

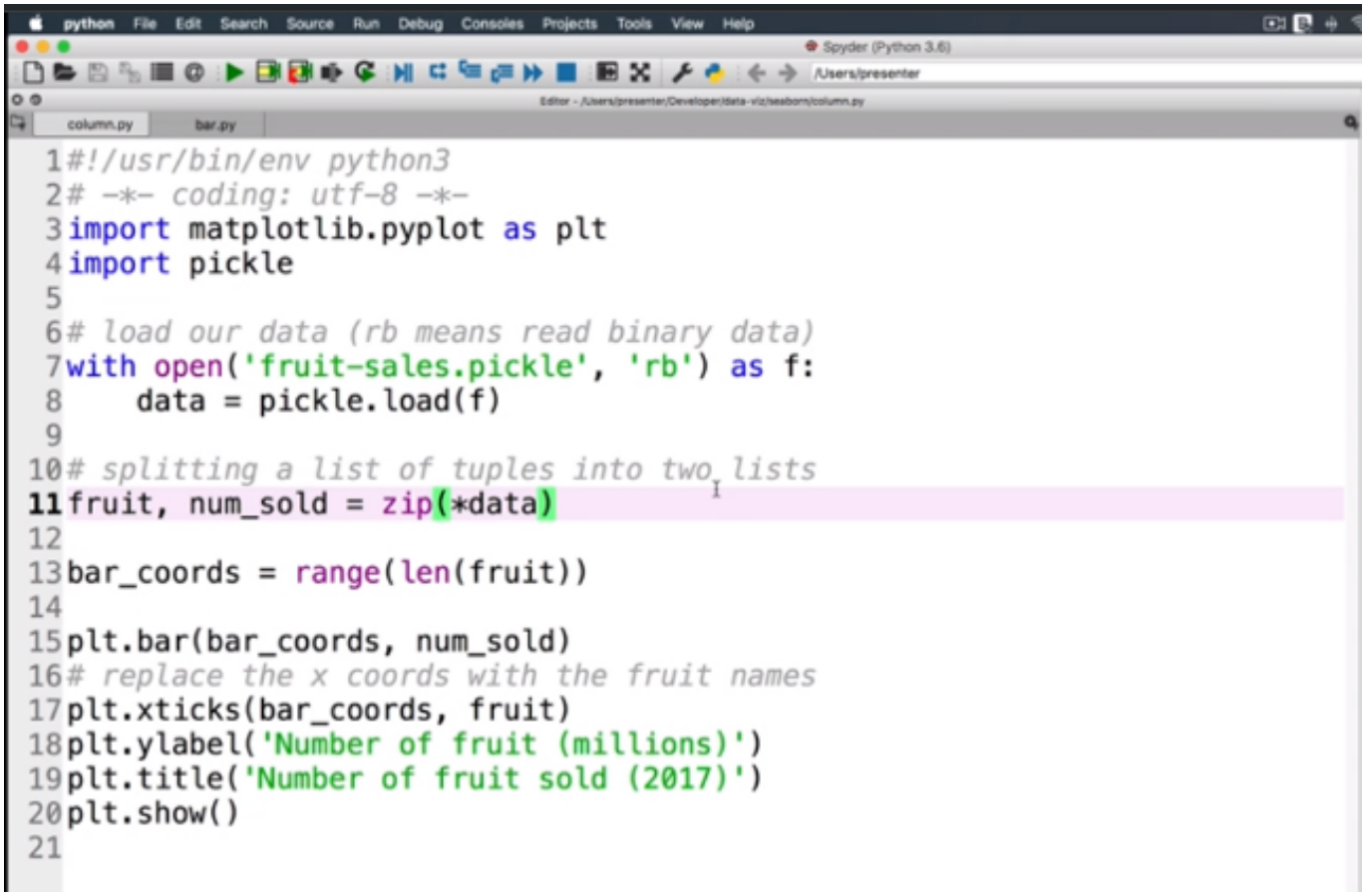
If I were working on a classification problem, the 2nd subplot (top-right) would tell me that the purple class had shorter petal length and petal widths.

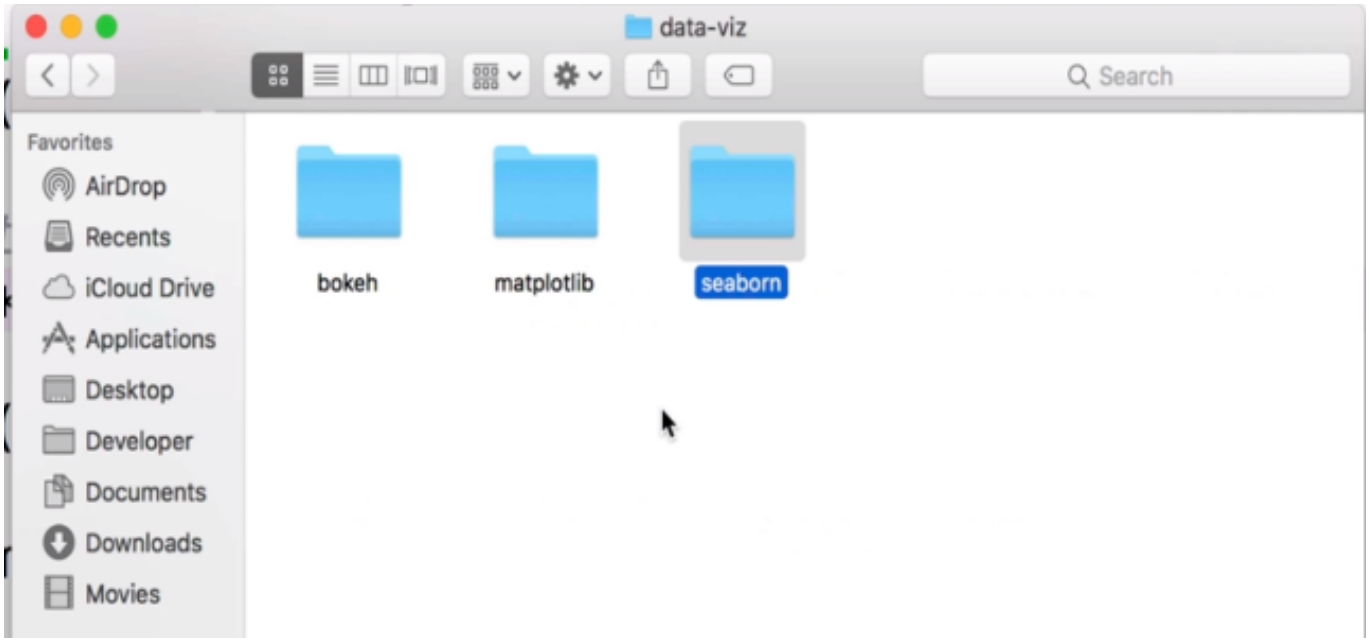
Summary

We demonstrated how to plot multiple charts inside one figure in *matplotlib*. We used the *plt.subplots()* function to build a grid of a specified number of row and columns. We use the returned *axes* object to index into the plot. The 1st index is the row and the 2nd is the column. We can do any kind of plot in these sub-plots. In the example above, we choose to do a scatter plot. The axes index decides which part of the grid the plot displays in. The functions *set_xlabel* and *set_ylabel* are used to label the sub-plots. The *figure* object is used to change the title of the entire plot.

In this video, we are going to going to look at a plotting library called Seaborn . Seaborn uses *matplotlib* under the hood and provides a different API for plotting similar charts. In this video, we will look specifically at column and bar charts. While people have their preferences of one library over another, it is good to know both APIs.

I've copied over *column.py* and *bar.py* (that we wrote using *matplotlib*) into a seaborn folder created for seaborn code. We will be using the same data as before.

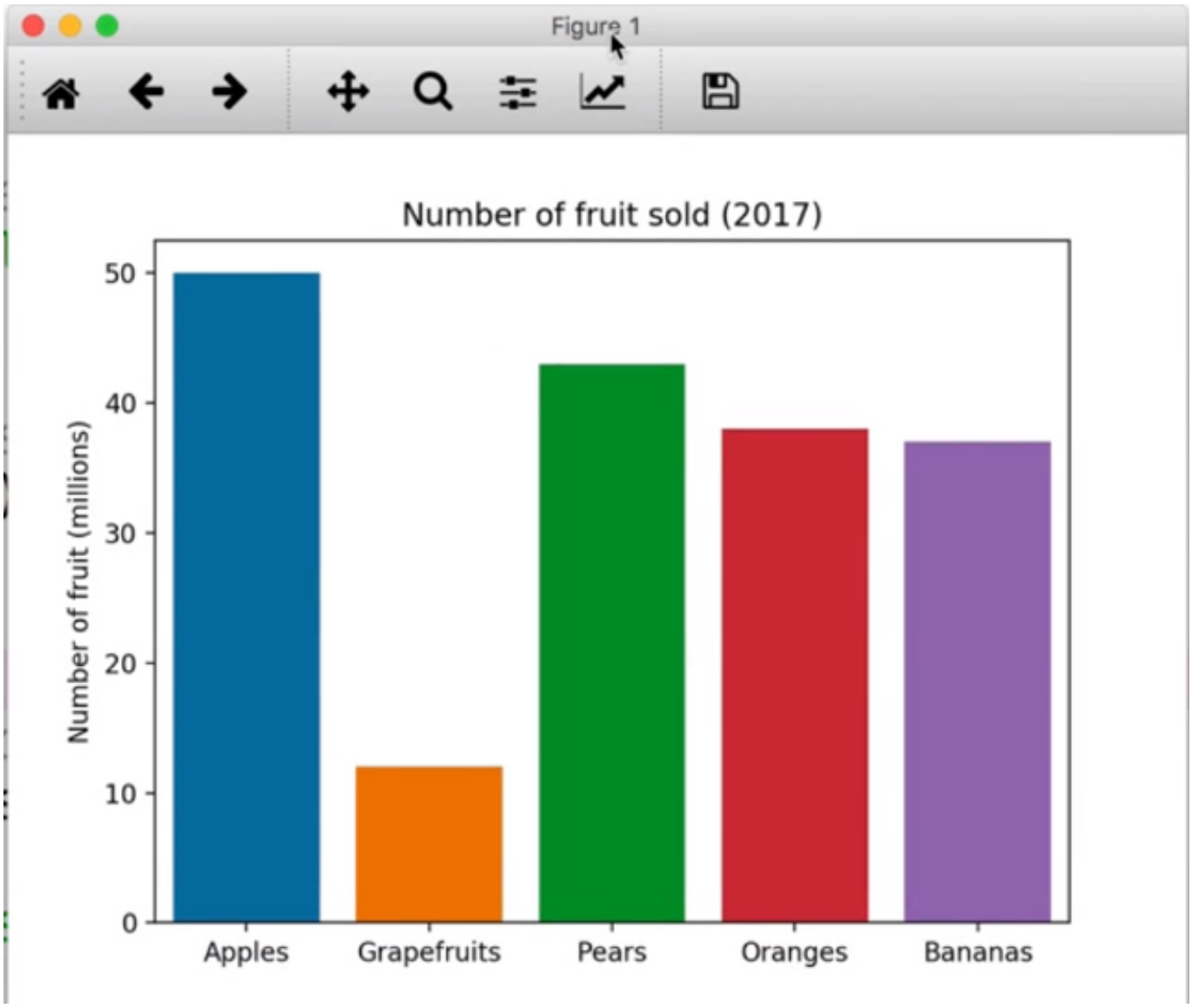




Lets start with *column.py* first.

```
# add this line to the imports
import matplotlib.pyplot as plt
import seaborn as sns    # add this
import pickle

# load data
with open ('fruit-sales.pickle', 'rb') as f :
    data = pickle.load(f)

# seaborn computes the bar coordinates for us
# so comment out this line
#matplotlib: bar_coords = range(len(fruit))

Likewise comment out the line below
# matplotlib: plt.xticks(bar_coords, fruit)

# Seaborn works with lists, so convert tuples to lists.

# NEW – below the zip(*data) statement
fruit = list(fruit)
num_sold = list(num_sold)

# NEW add code for our bar plot.
sns.barplot(x=fruit, y=num_sold)

# Since seaborn uses matplotlib under the hood, all the plotting statements will cont
inue to work.
plt.ylabel('Number of fruit (millions)')
plt.title('Number of fruit sold (2017)')
plt.show()
```

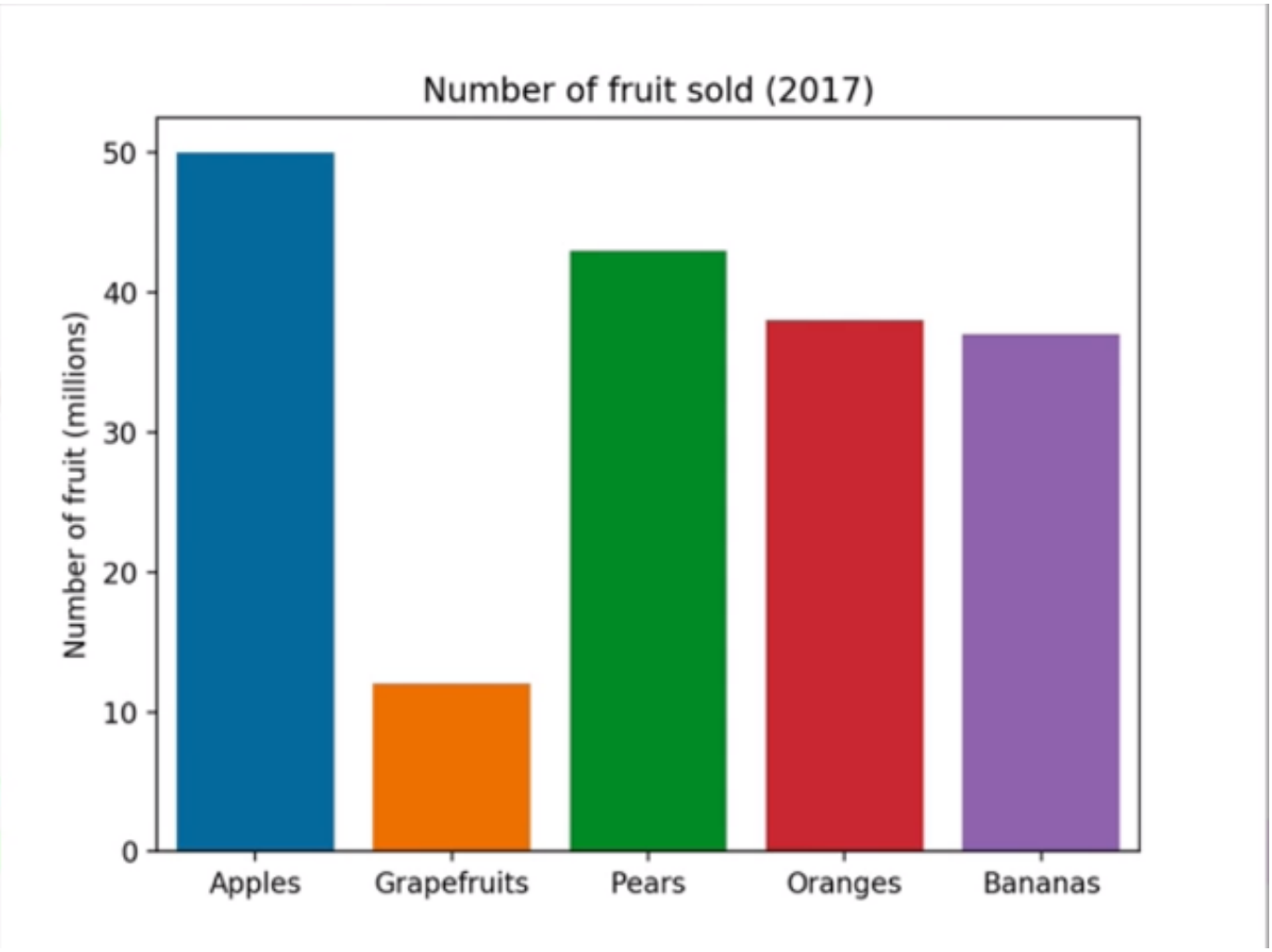
Let’s run this code.



The Seaborn plot has colored the plot separately for us.

There is a Seaborn way of setting axis labels and title. Lets replace our existing code with this.

```
# NEW - Modify the code below
axes = sns.barplot(x=fruit, y=num_sold)
axes.set_title('Number of fruit sold (2017)')
axes.set_ylabel('Number of fruit (millions)')
```

We get the same plot using the Seaborn APIs.

Now lets do a bar chart using Seaborn.

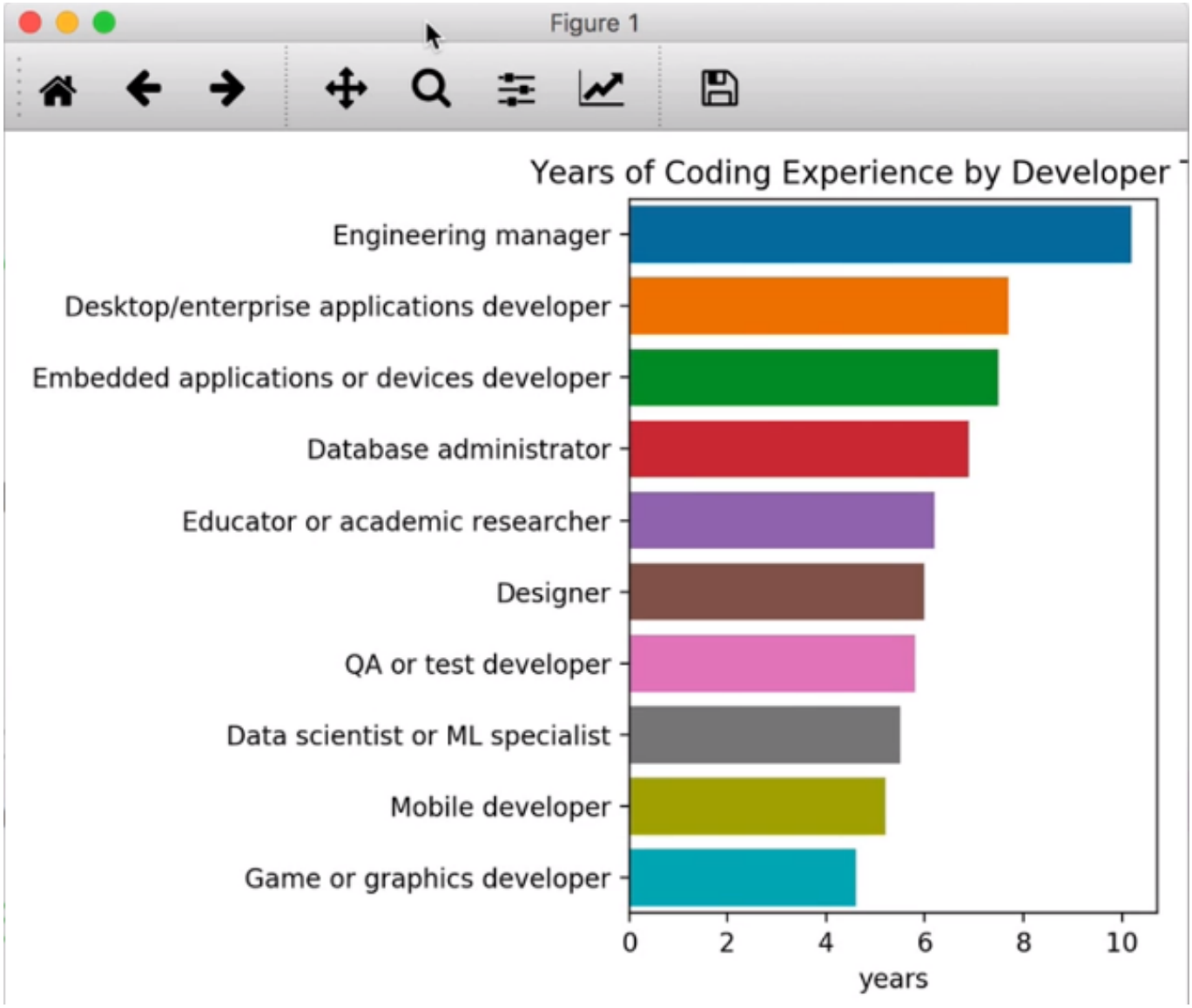
Lets start with *bar.py* from the *matplotlib* code.


```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import pickle

# load data
with open('coding-exp-by-dev-type.pickle', 'rb') as f:
    data = pickle.load(f)

# split into two lists
dev_types, years_exp = zip(*data)

bar_coords = range(len(dev_types))

plt.barh(bar_coords, years_exp)
plt.xlabel('years')
plt.title('Years of Coding Experience by Developer Type')
plt.yticks(bar_coords, dev_types, fontsize=8)
plt.tight_layout()
plt.show()

```

We want to convert this to a Seaborn plot.

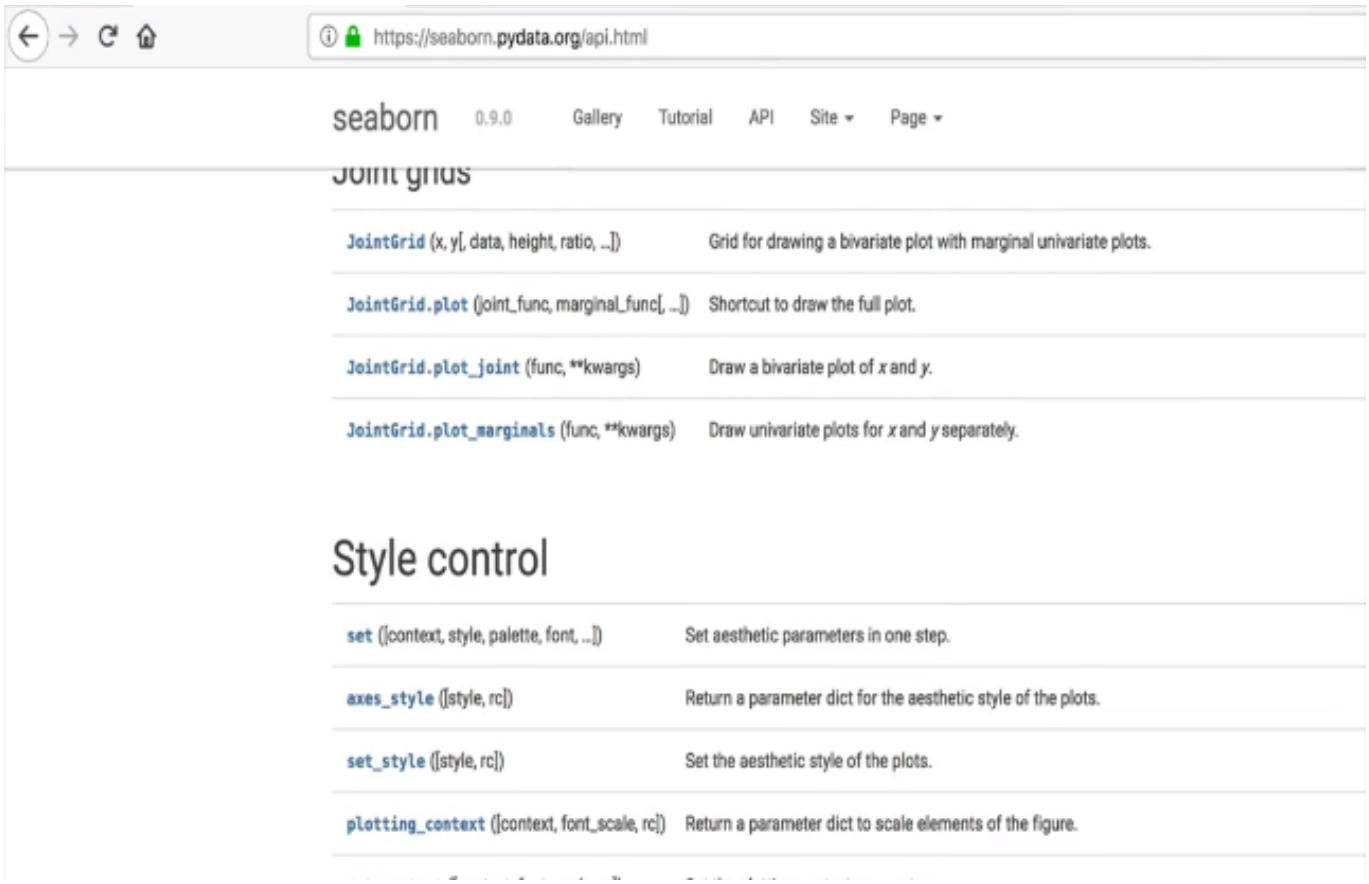
```
import matplotlib.pyplot as plt
import pickle

# load data
with open('coding-exp-by-dev-type.pickle', 'rb') as f :
    data = pickle.load(f)

dev_types, years_exp = zip(*data)

# NEW - convert tuple to list
dev_types = list(dev_types)
years_exp = list(years_exp)

# COMMENT OUT
#bar_coords = range(len(dev_types))

# COMMENT OUT
#matplotlib: plt.barh(bar_coords, years_exp)

# COMMENT OUT
#plt.yticks(bar_coords, dev_types, fontsize=8)

# NEW
sns.barplot(y=dev_types, x=years_exp)
plt.xlabel('years')
plt.title('Years of Coding Experience by Developer Type')

plt.show()
```


Let’s run this code.



Documentation

The Seaborn documentation has good information about the various plots it supports. There are various sections based on various functionality.



There are style controls, utility functions to load data sets. There is a section on bar plots that we can look at.

seaborn.**barplot**(*x=None, y=None, hue=None, data=None, order=None, hue_order=None, estimator=<function mean>, ci=95, n_boot=1000, units=None, orient=None, color=None, palette=None, saturation=0.75, errcolor='.26', errwidth=None, capsize=None, dodge=True, ax=None, **kwargs*)

Show point estimates and confidence intervals as rectangular bars.

A bar plot represents an estimate of central tendency for a numeric variable with the height of each rectangle and provides some indication of the uncertainty around that estimate using error bars. Bar plots include 0 in the quantitative axis range, and they are a good choice when 0 is a meaningful value for the quantitative variable, and you want to make comparisons against it.

For datasets where 0 is not a meaningful value, a point plot will allow you to focus on differences between levels of one or more categorical variables.

It is also important to keep in mind that a bar plot shows only the mean (or other estimator) value, but in many cases it may be more informative to show the distribution of values at each level of the categorical variables. In that case, other approaches such as a box or violin plot may be more appropriate.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, ... n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

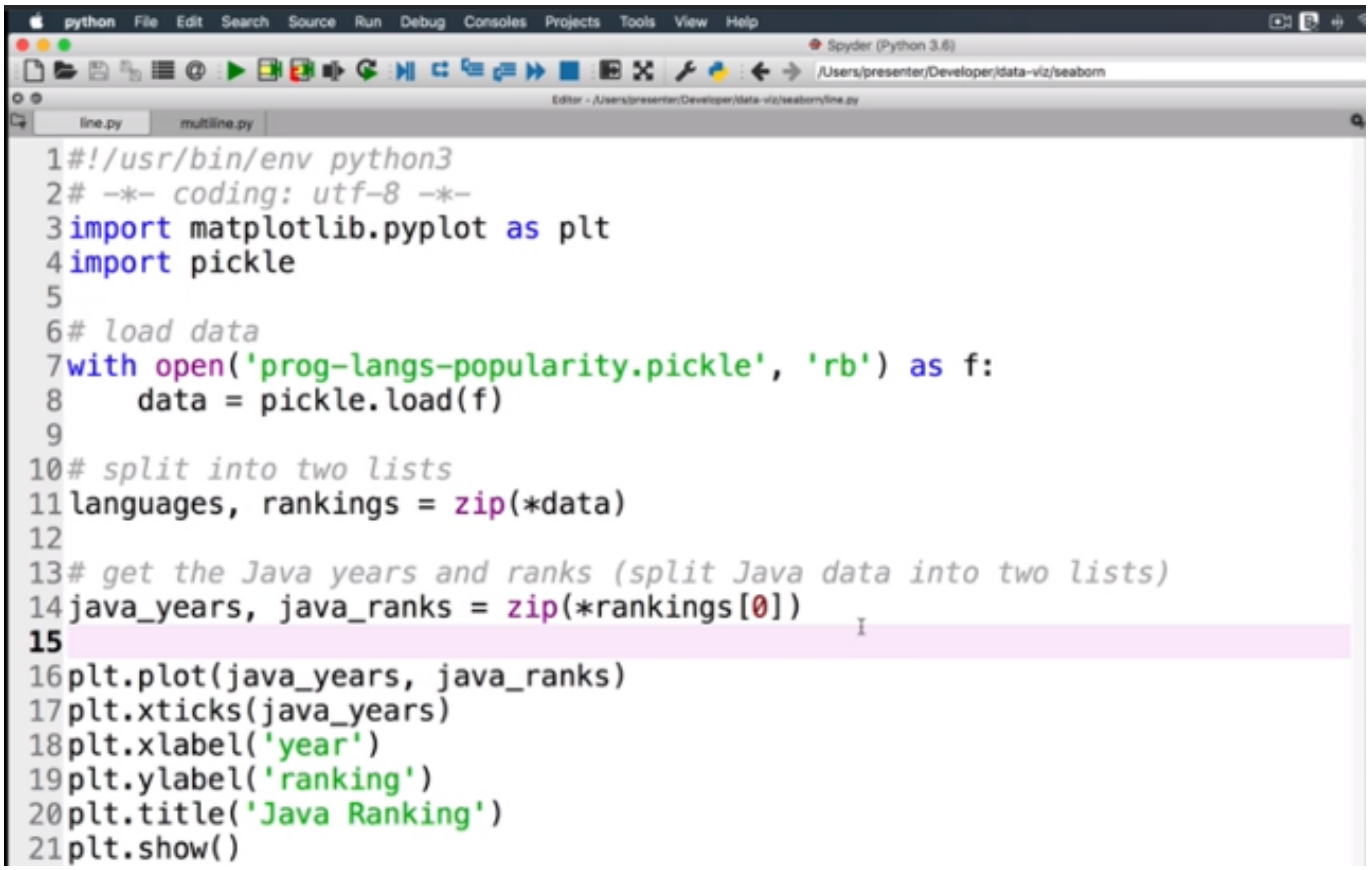
Parameters: **x, y, hue** : names of variables in `data` or vector data, optional

Inputs for plotting long-form data. See examples for interpretation.

The API documentation shows how we can manipulate the plot in various ways – change its saturation, change color palettes. We can use the API documentation to see what plot customization options are available to us.

In this lesson, we are going to see how to do **line** plots and **multiline** plots in Seaborn.

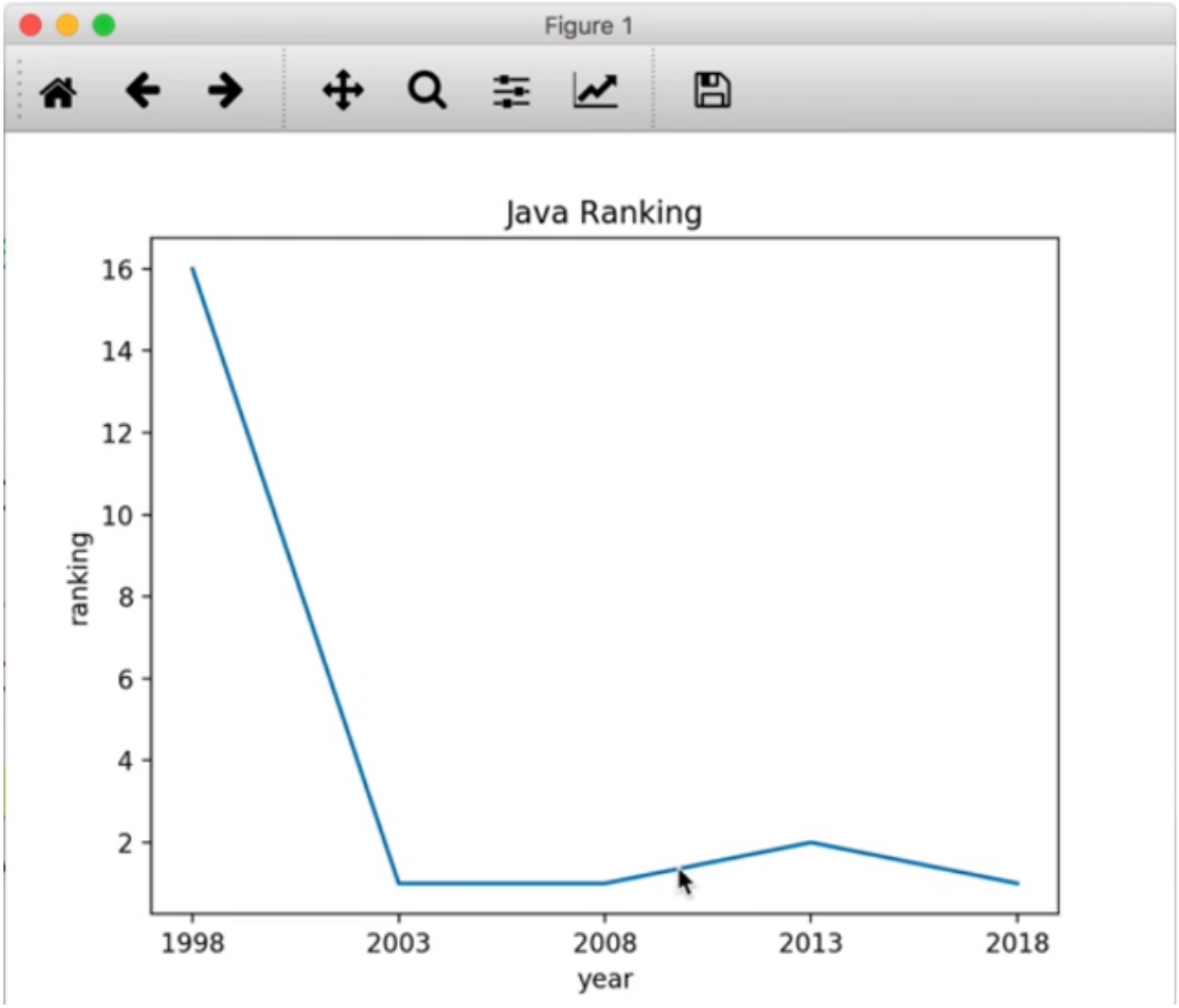
Let's start with the code in *line.py* (from a previous lesson).



New code will be marked as # NEW.

```
# NEW
import seaborn as sns

Let change the plotting code to Seaborn style.

# matplotlib: plt.plot(java_years, java_ranks)
sns.lineplot(java_years, java_ranks)
```

Let's run this code.

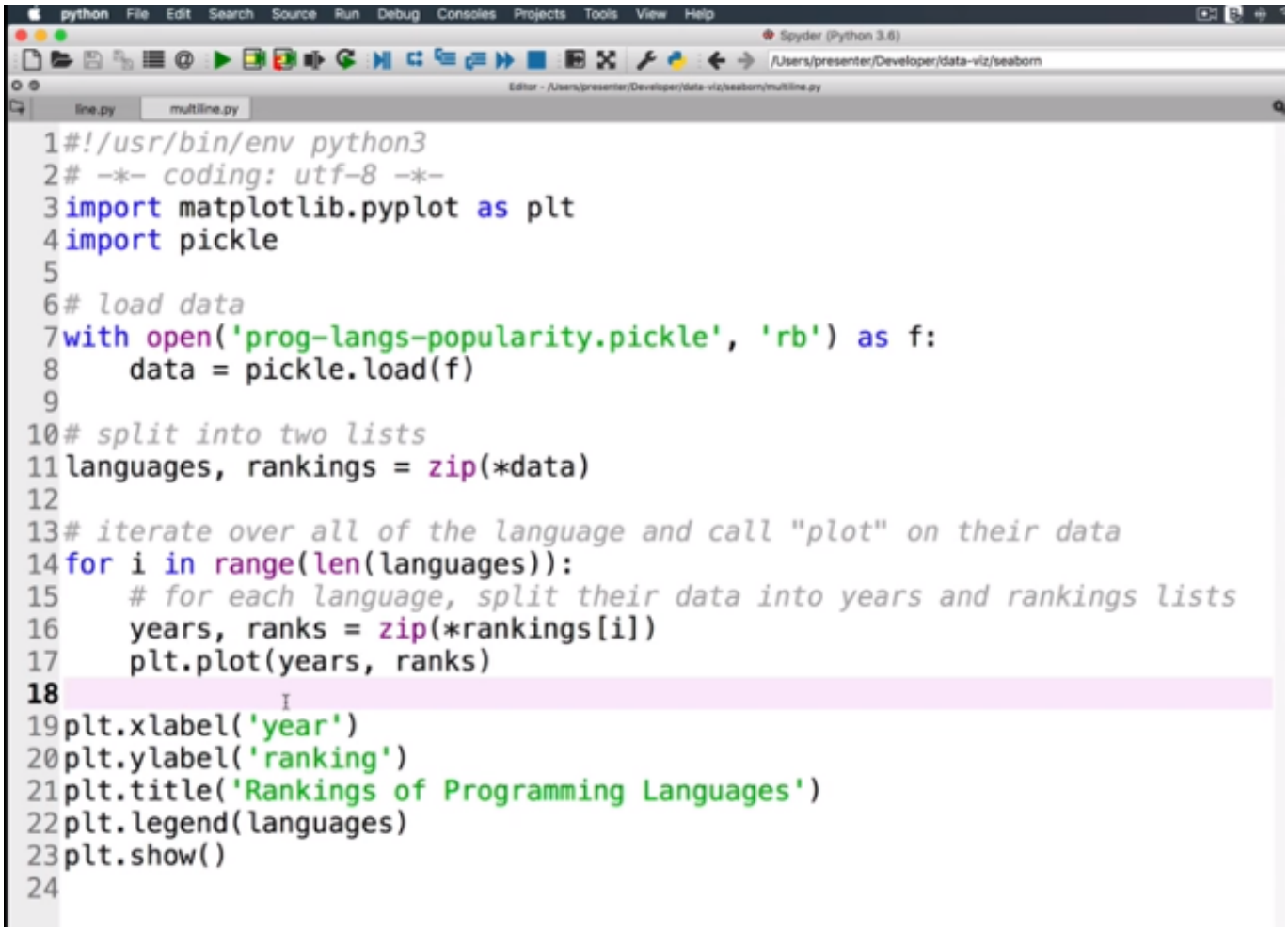


This is pretty much identical to the *matplotlib* plot we saw earlier.

The lineplot API documentation () provides many ways to customize the line plot.

Multiline plot

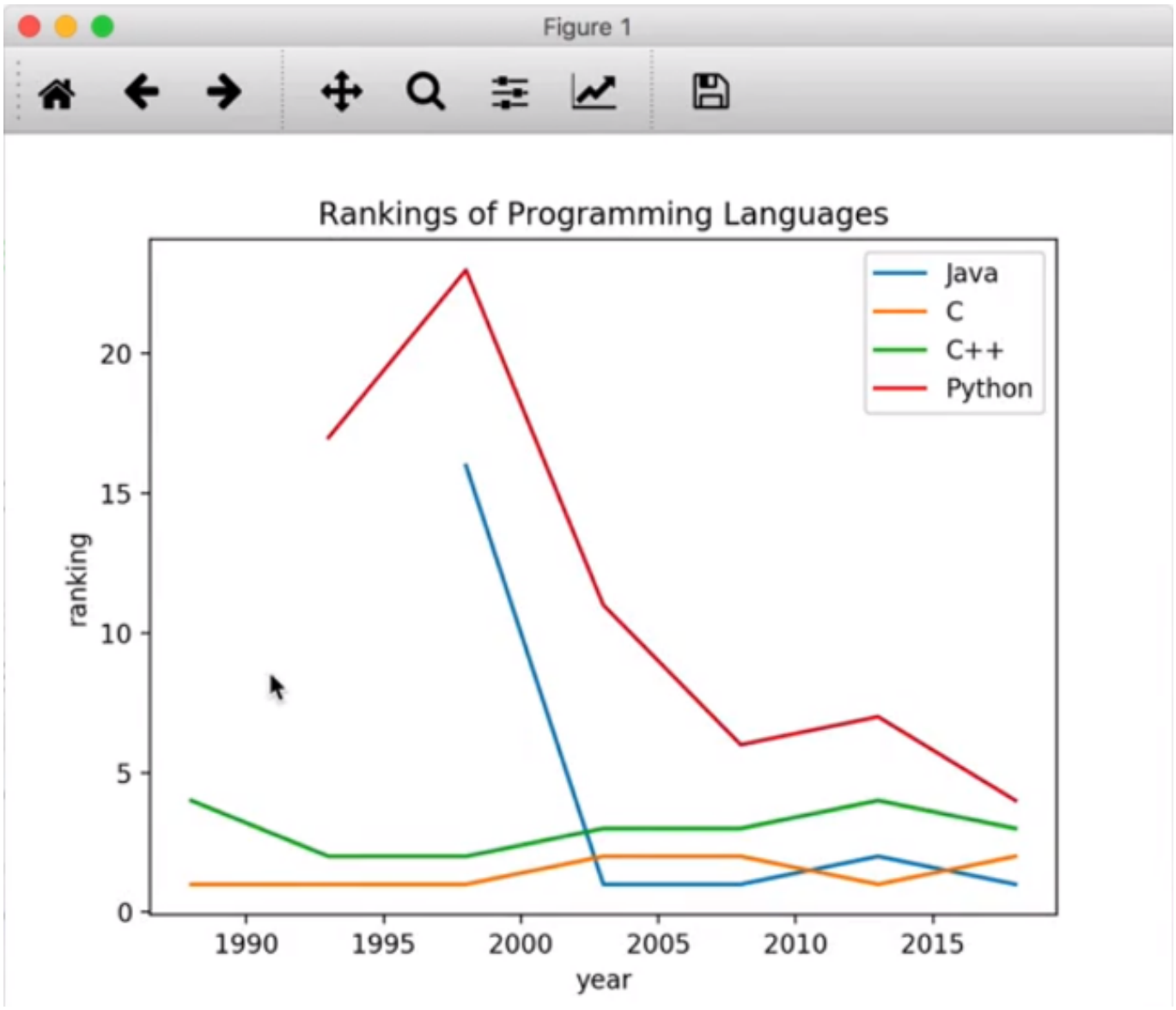
Let's plot a multi line plot using Seaborn. Lets start with copy of *multiline.py* (Matplotlib).



CHALLENGE : Change this code to use Seaborn APIs.

```
# NEW
import seaborn as sns

# change the plot() call as follows

#matplotlib: plt.plot(years, ranks)

#NEW
sns.lineplot(years, ranks)
```

Let’s run this code.

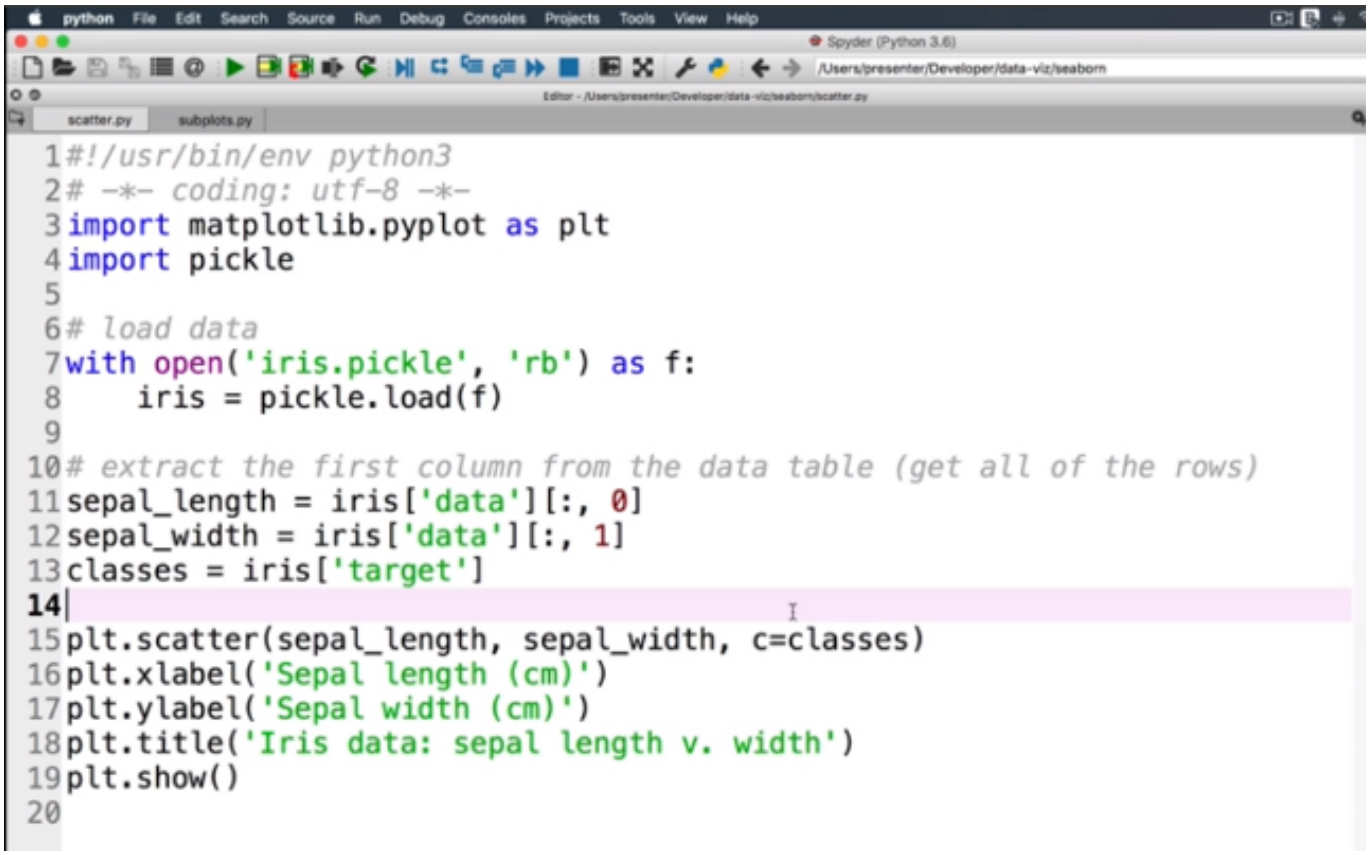


Summary

We see that most changes from matplotlib to seaborn are **1-line changes**. The plots we get from using seaborn are almost identical to those produced by matplotlib.

In this video, we are going to look a how to do **scatter plots** and multiple **sub plots** in **Seaborn**.

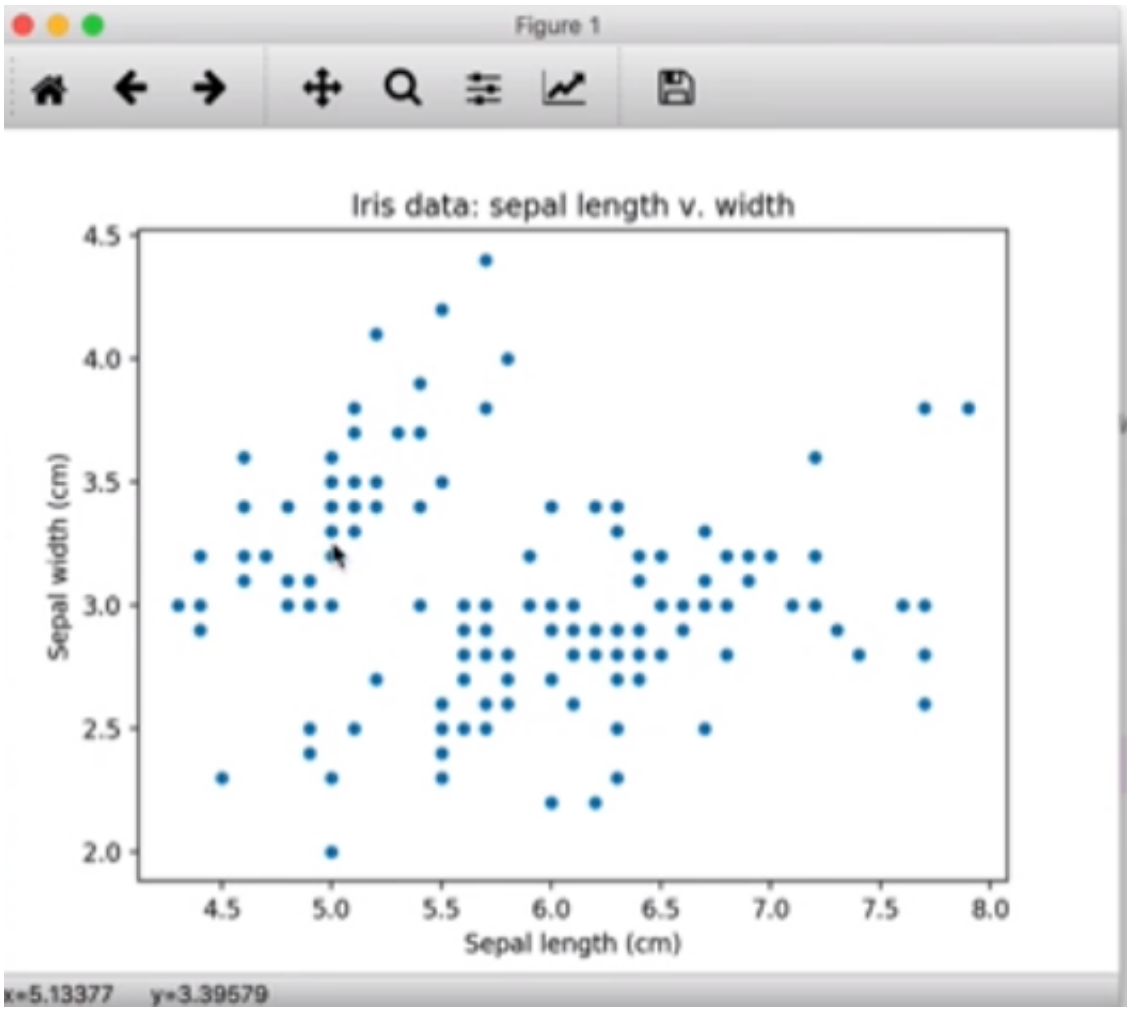
Let's start with the code we wrote earlier.



Let's first look at how to do a **scatter plo**t in Seaborn. Lets add Seaborn code to the above code.

```
#NEW
import seaborn to sns

#NEW
# matplotlib: plt.scatter(sepal_length, sepal_width, c=classes)
sns.scatterplot(sepal_length, sepal_width, )
```

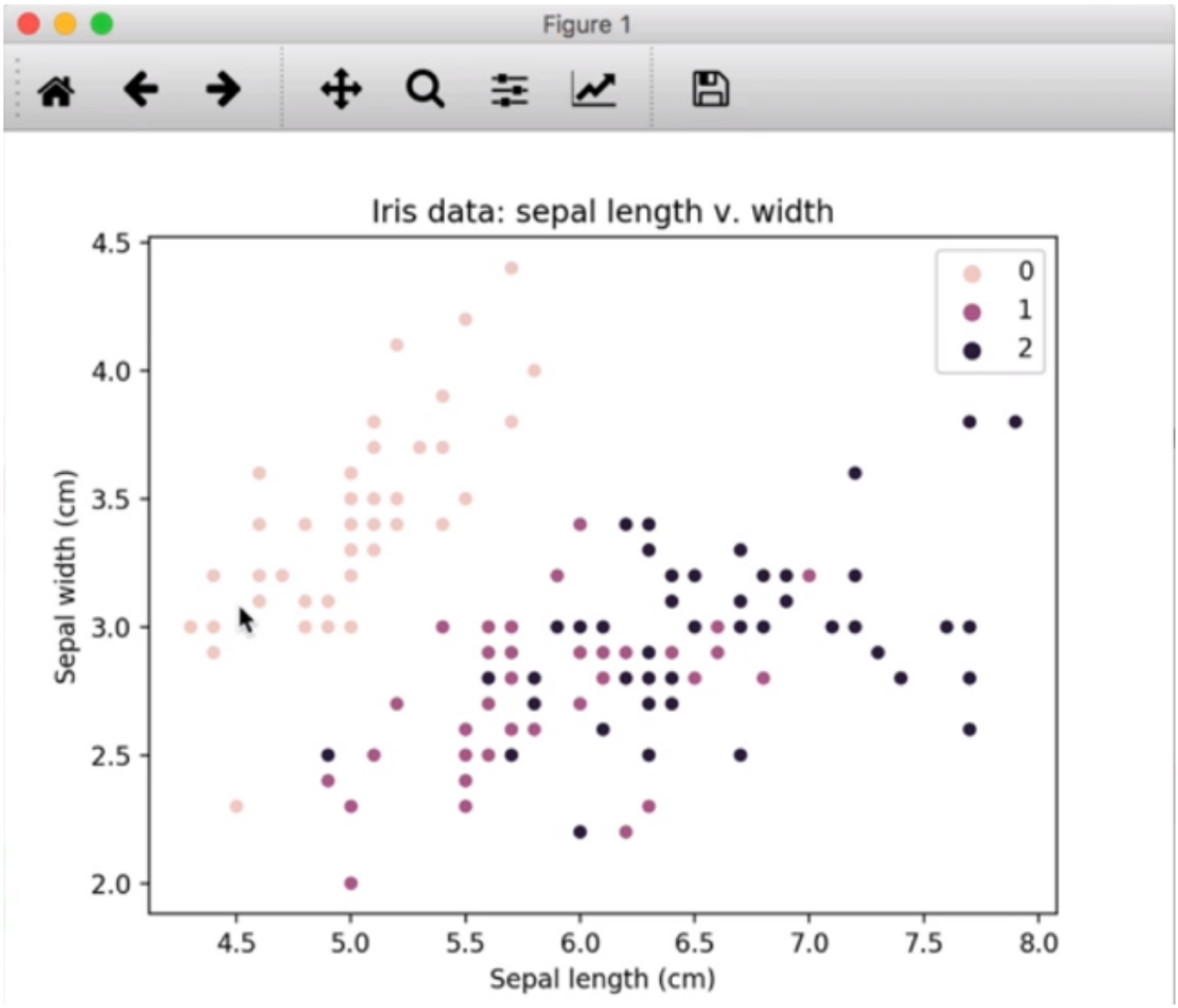
Let's run this code.



The plot looks very similar to that produced by *matplotlib*.

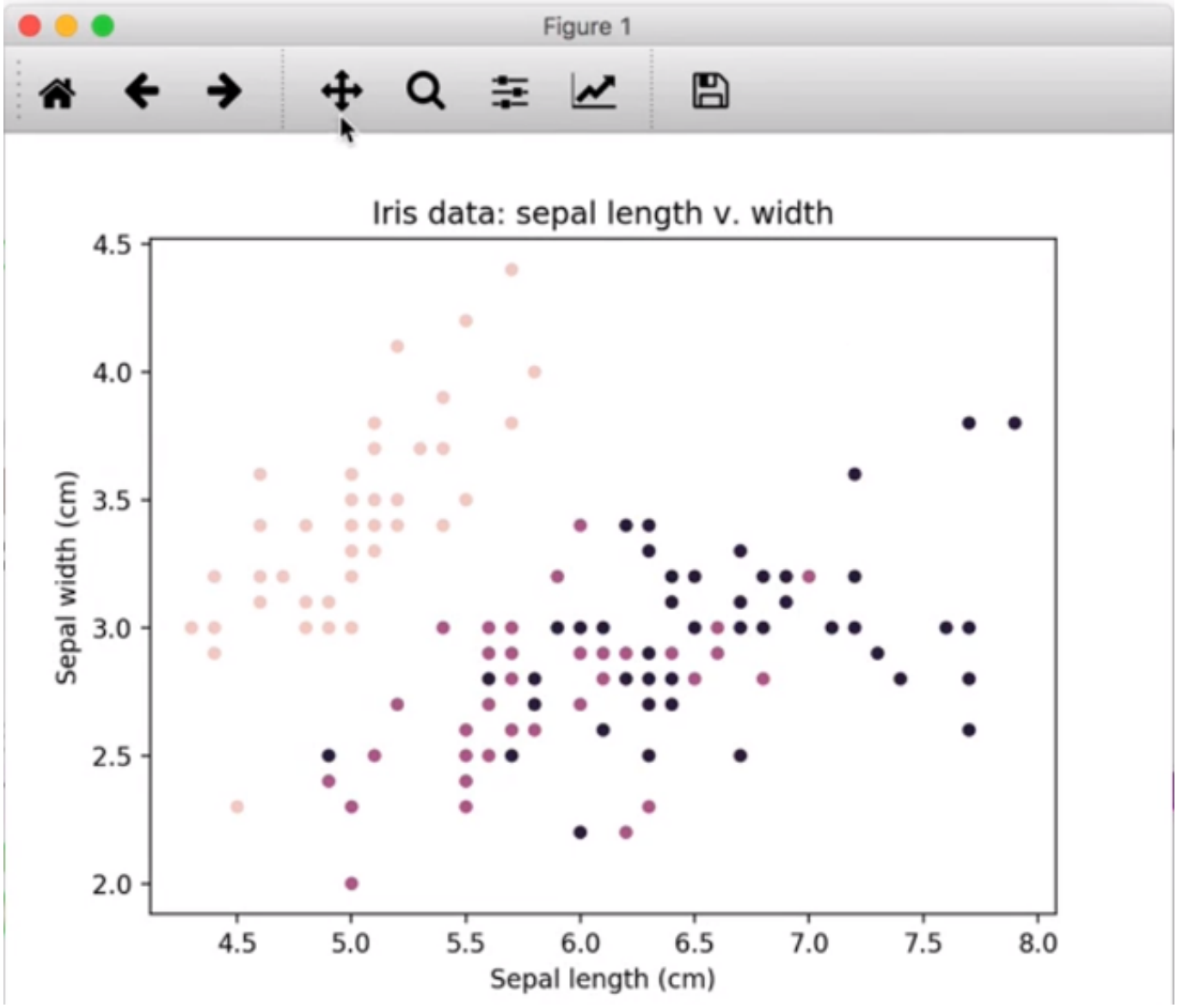
Let's color the points based on the class. Let's modify the code above.

```
sns.scatterplot(sepal_length, sepal_width, hue = classes)
```

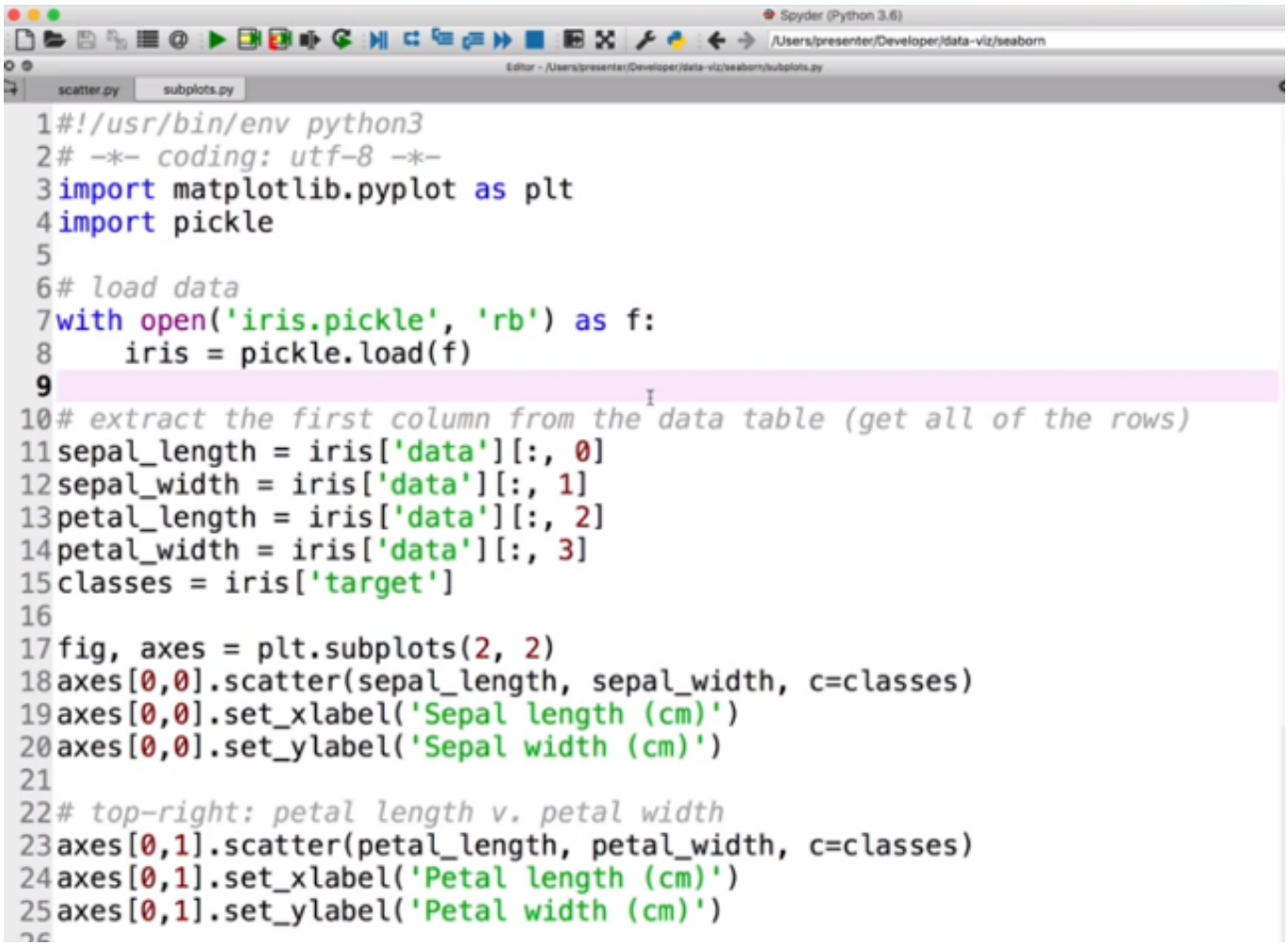



We get a colored plot along with a legend. Lets get rid of the legend.

```
sns.scatterplot(sepal_length, sepal_width, hue = classes, legend=False)
```

Let's now get the **sub plot** code working. Let's copy over the previous sub plot code (*matplotlib*).



Let's add the Seaborn code.

```
#NEW
import seaborn as sns

# changes from line 18 above
# matplotlib: axes[0, 0].scatter(sepal_length, sepal_width, c=classes)

# create scatter plot, put it on axes[0,0]
# analogous to axes[0,0] in matplotlib
sns.scatterplot(sepal_length, sepal_width, hue=classes, legend=False, ax=axes[0,0])
axes
```

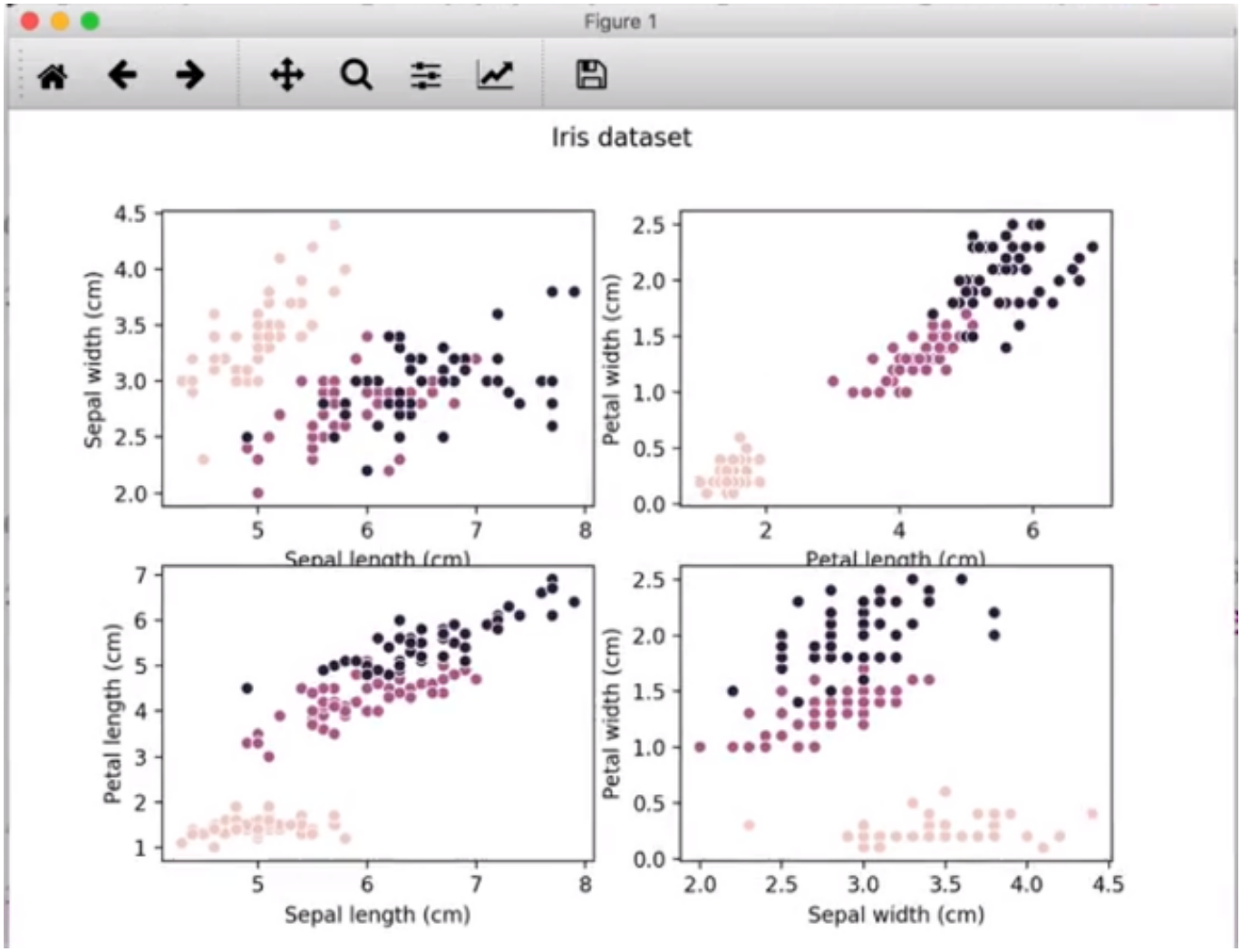
CHALLENGE : Replace the lines of code below the above block, replacing matplotlib code with seaborn code.

```
# matplotlib: axes[0, 1].scatter(petal_length, petal_width, c=classes)

# create scatter plot, put it on axes[0,1]
# analogous to axes[0,1] in matplotlib
sns.scatterplot(petal_length, petal_width, hue=classes, legend=False, ax=axes[0,1])
```



```
# matplotlib: axes[1, 0].scatter(sepal_length, petal_length, c=classes)

# create scatter plot, put it on axes[1,0]
# analogous to axes[1,0] in matplotlib
sns.scatterplot(sepal_length, petal_length, hue=classes, legend=False, ax=axes[1,0])
# matplotlib: axes[1, 1].scatter(sepal_width, petal_width, c=classes)

# create scatter plot, put it on axes[1,1]
# analogous to axes[1,1] in matplotlib
sns.scatterplot(sepal_width, petal_width, hue=classes, legend=False, ax=axes[1,1])
```

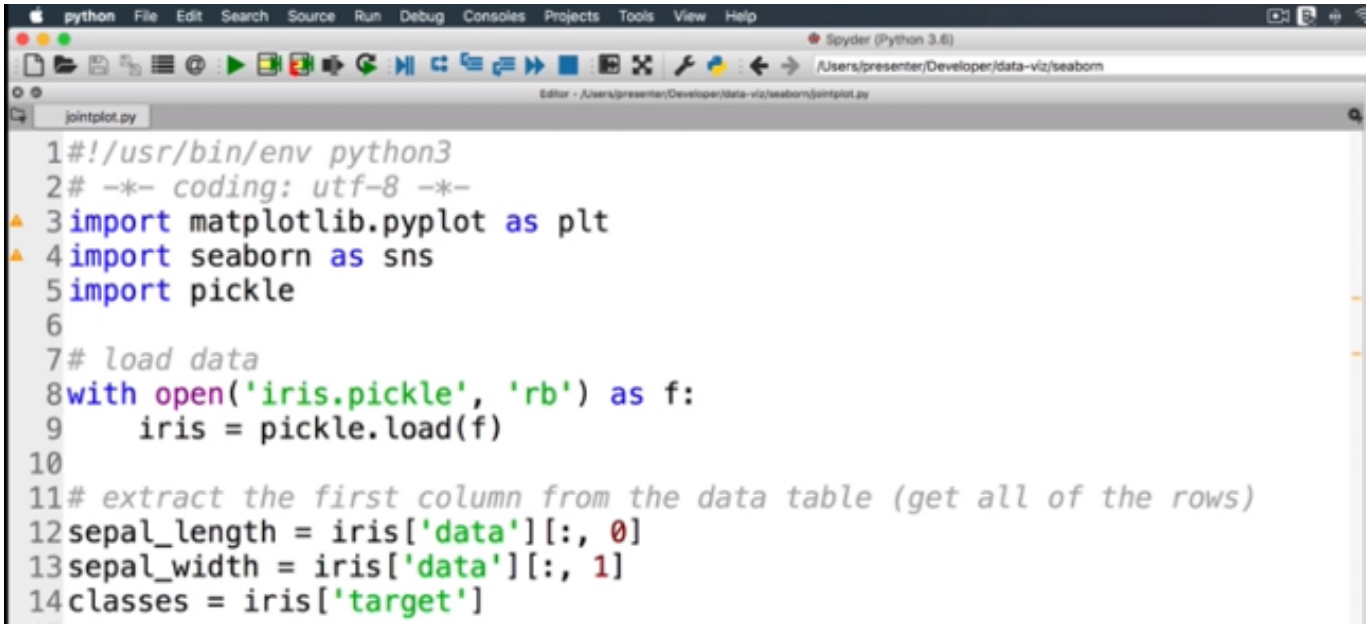
Let's run the code.



We see all the sub plots, now rendered by Seaborn code.

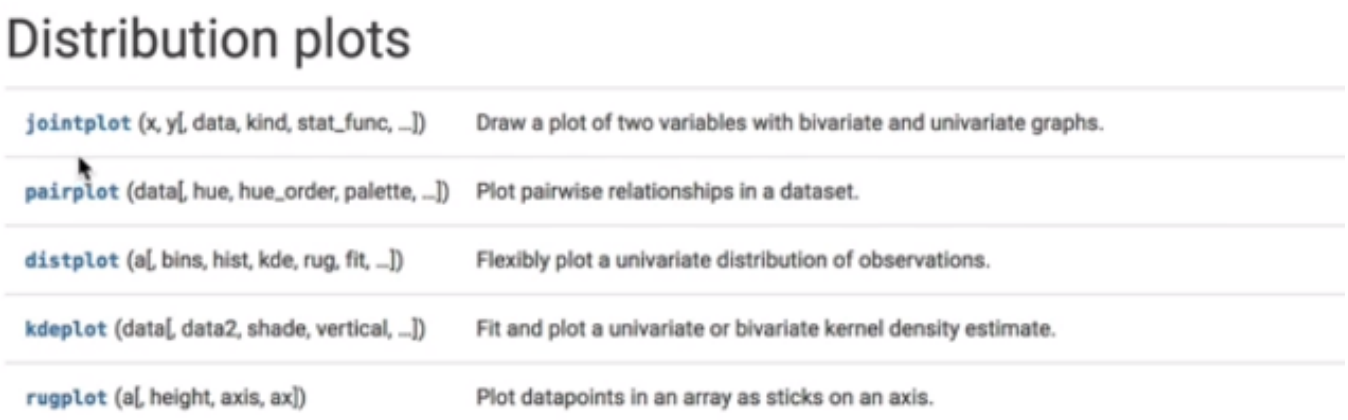
In this code, we will look at a plot called **joint-plot** in Seaborn. It's a very useful plot, similar to a scatter plot – very versatile in terms of the way it shows the scattering.

We've copied the Seaborn scatter plot code after removing the plotting code.



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import seaborn as sns
import pickle

# load data
with open('iris.pickle', 'rb') as f:
    iris = pickle.load(f)

# extract the first column from the data table (get all of the rows)
sepal_length = iris['data'][:, 0]
sepal_width = iris['data'][:, 1]
classes = iris['target']
```

The Seaborn documentation reveals a joint plot. It allows us to look at 2 features of a piece of data in an interesting way.



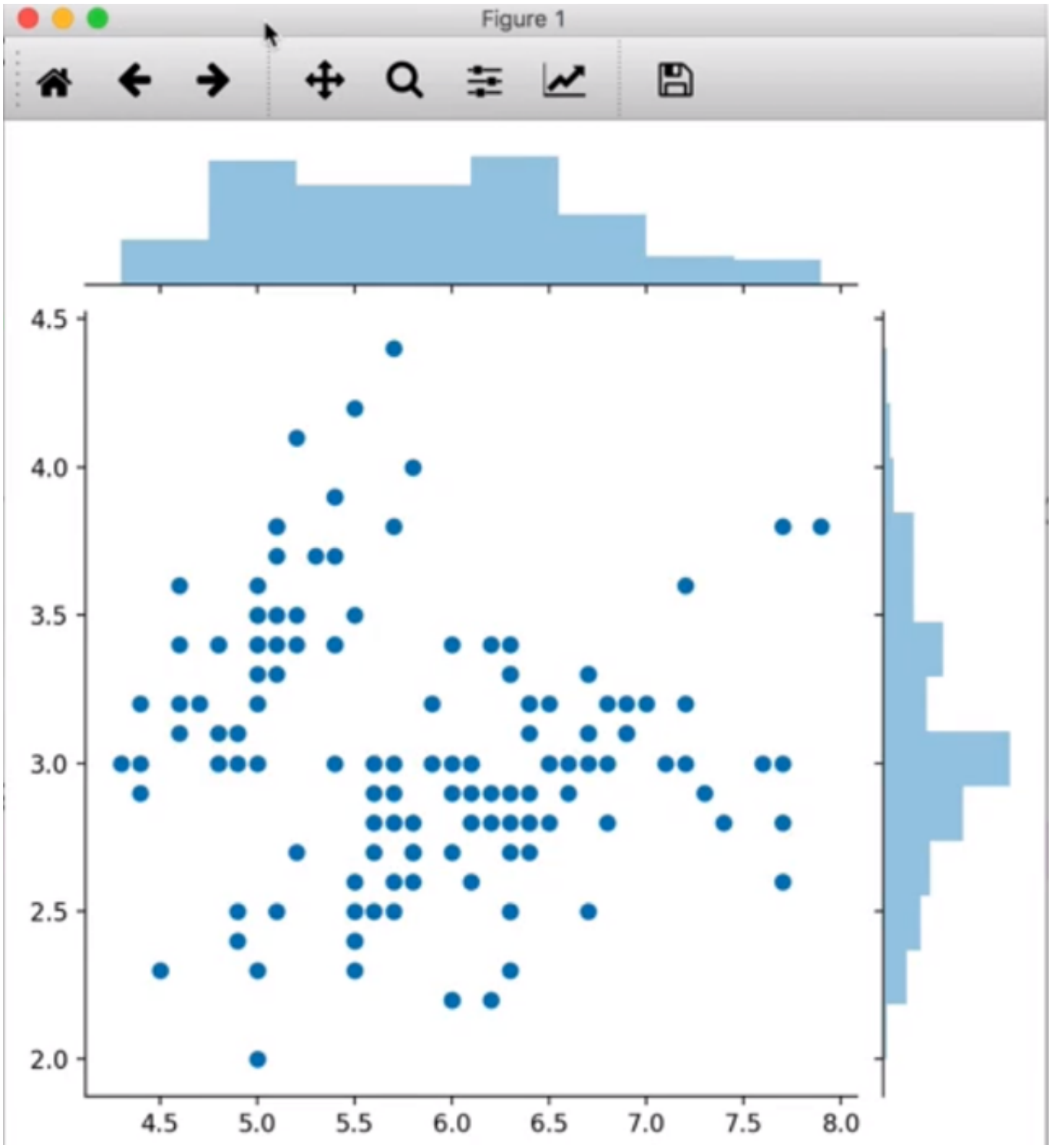
Distribution plots

jointplot (x, y[, data, kind, stat_func, ...])	Draw a plot of two variables with bivariate and univariate graphs.
pairplot (data[, hue, hue_order, palette, ...])	Plot pairwise relationships in a dataset.
distplot (a[, bins, hist, kde, rug, fit, ...])	Flexibly plot a univariate distribution of observations.
kdeplot (data[, data2, shade, vertical, ...])	Fit and plot a univariate or bivariate kernel density estimate.
rugplot (a[, height, axis, ax])	Plot datapoints in an array as sticks on an axis.

Lets code a joint plot.

```
sns.jointplot(sepal_length, sepal_width)
plt.show()
```

Let's run this code.



Along with the scatter plot, we see the **histograms** on the vertical and horizontal axes. It has **bins** and we can see how many points are in each bin. For example, the bin corresponding to 3.0 on the y-axis seems to be the bin of highest density (largest number of points). Similarly the vertical slice corresponding to 5 on the x-axis seems to have the highest number of point. It's hard to see these without the axes labels.

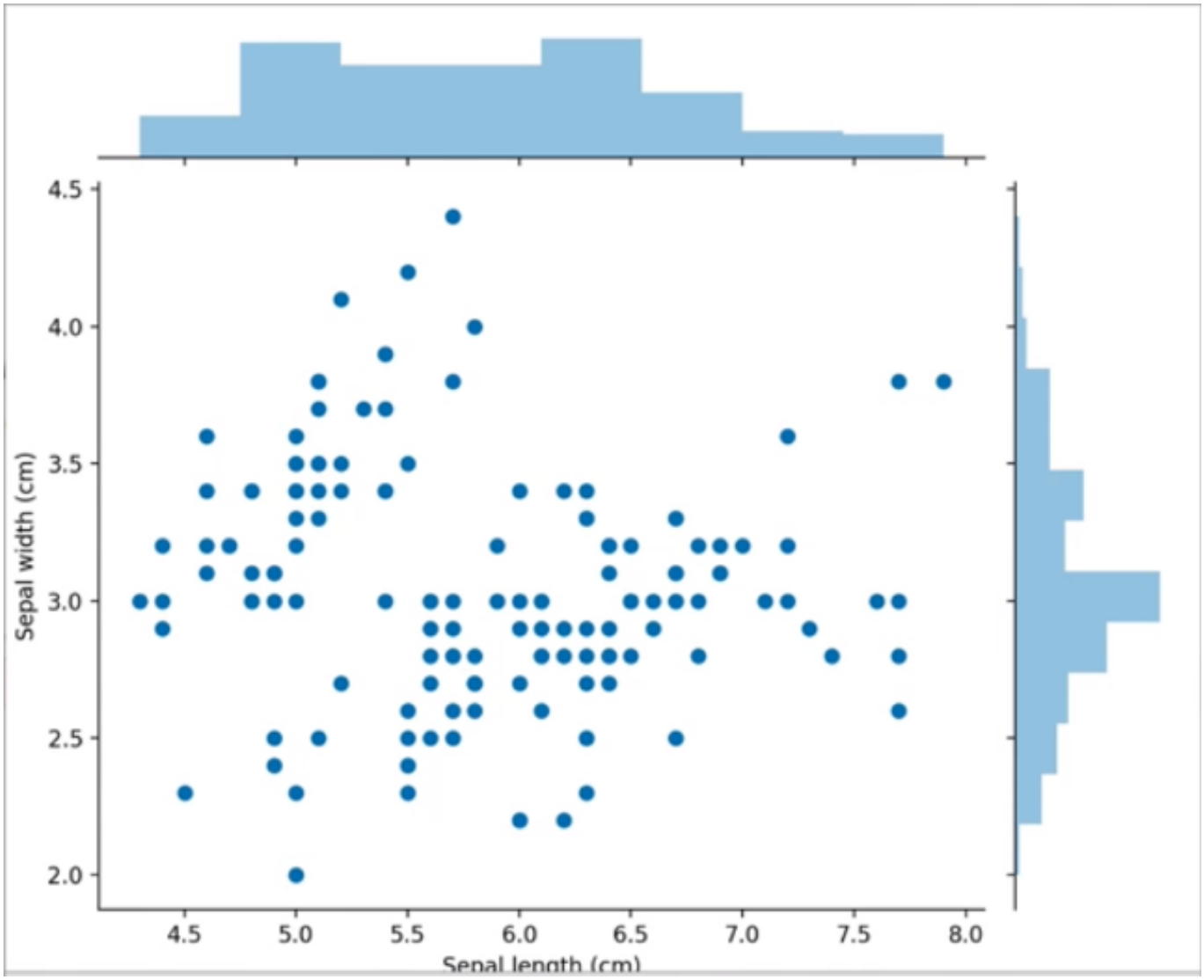
Lets add axes on this plot. Make the following changes prior to the *plt.show()* call.

```
# get a handle to the axes
```



```
axes = sns.jointplot(sepal_length, sepal_width)
axes.set_axis_labels('Sepal length (cm)',  'Sepal width (cm)')
```

Let's run this code.



We see the plot axes. This plot gives us additional useful additional information (density) about the 2 features.

We can see that the central part of the plot has the most density.

We can have **different** kinds of plots. We do this by using the ***kind*** parameter.

Parameters: **x, y** : strings or vectors

> Data or names of variables in `data` .

data : DataFrame, optional

> DataFrame when `x` and `y` are variable names.

kind : { "scatter" | "reg" | "resid" | "kde" | "hex" }, optional

> Kind of plot to draw.

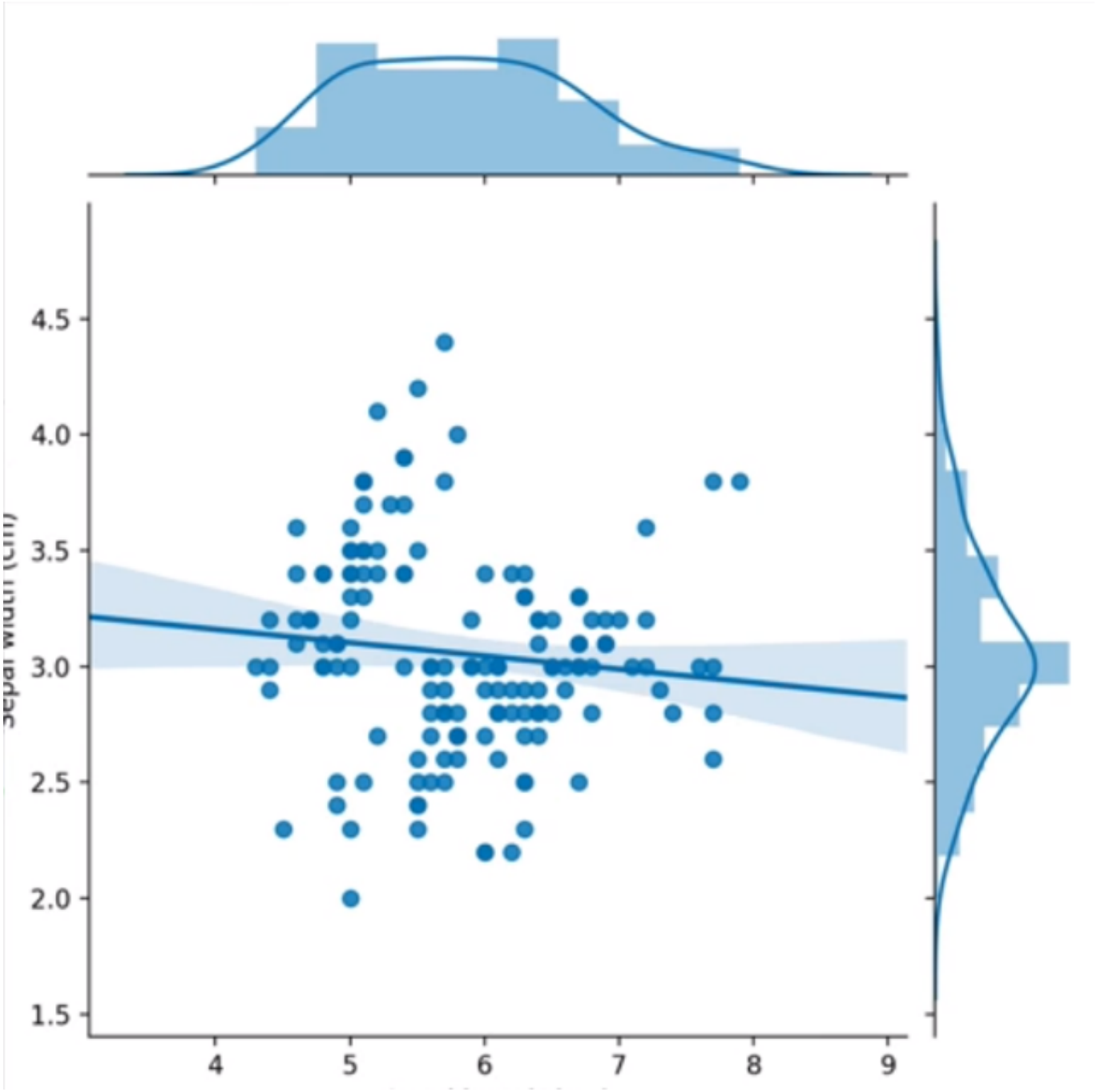
stat_func : callable or None, optional

> *Deprecated*

Let's modify the code above to see what these type of plots represent.

```
#regular plot
axes = sns.jointplot(sepal_length, sepal_width, kind='reg')
```

Let's run this code.



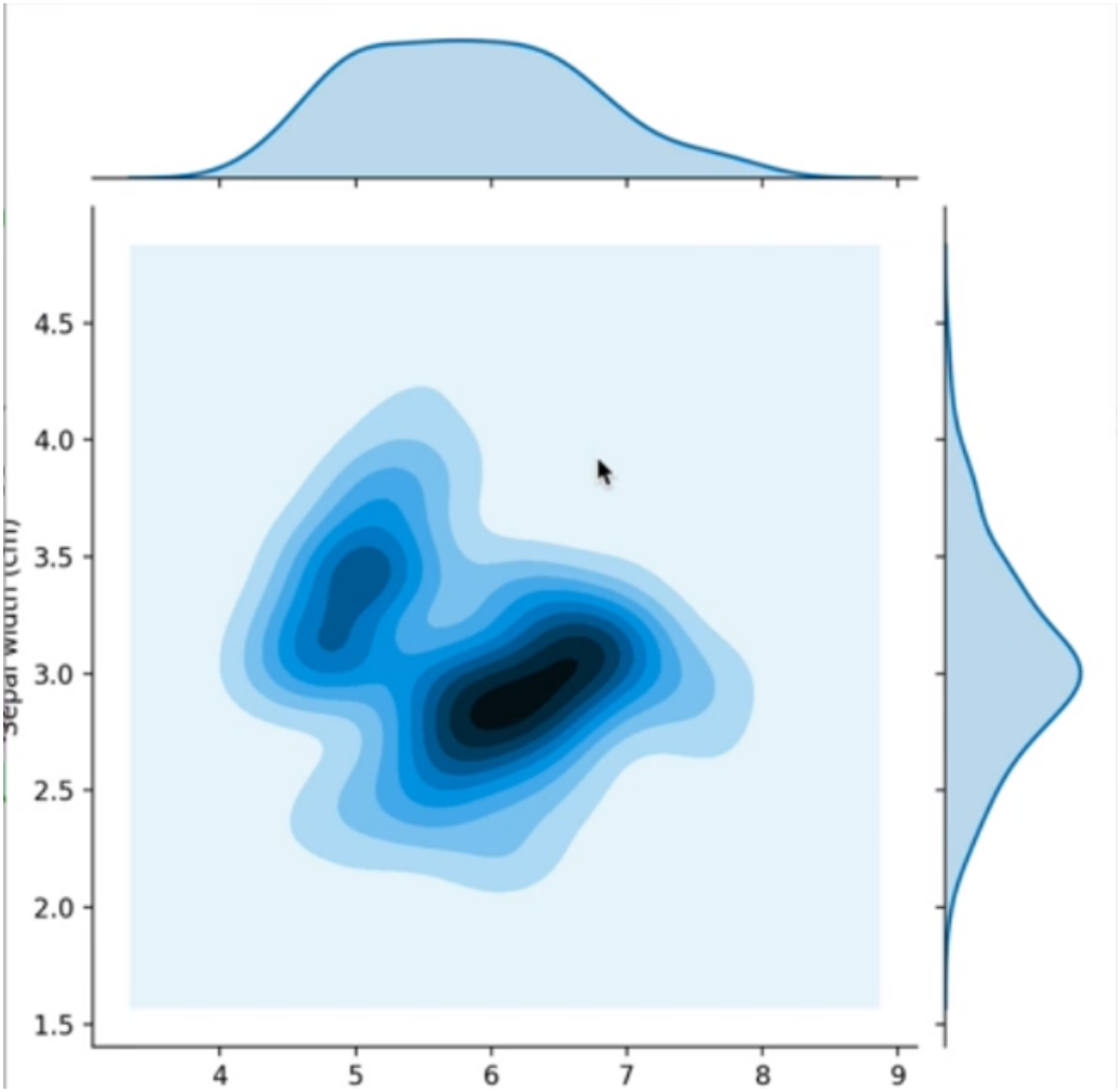
We see that in addition to getting a scatter plot, it is trying to fit the line of best fit. We also see the data distributions on the histograms.

For example, the histogram on the right has a peak at sepal_width around 3.0. The mean would be around sepal_length around 5.0.

Let's try the **kde** plot.

```
# kde plot
axes = sns.jointplot(sepal_length, sepal_width, kind='kde')
```


Let's run this code.

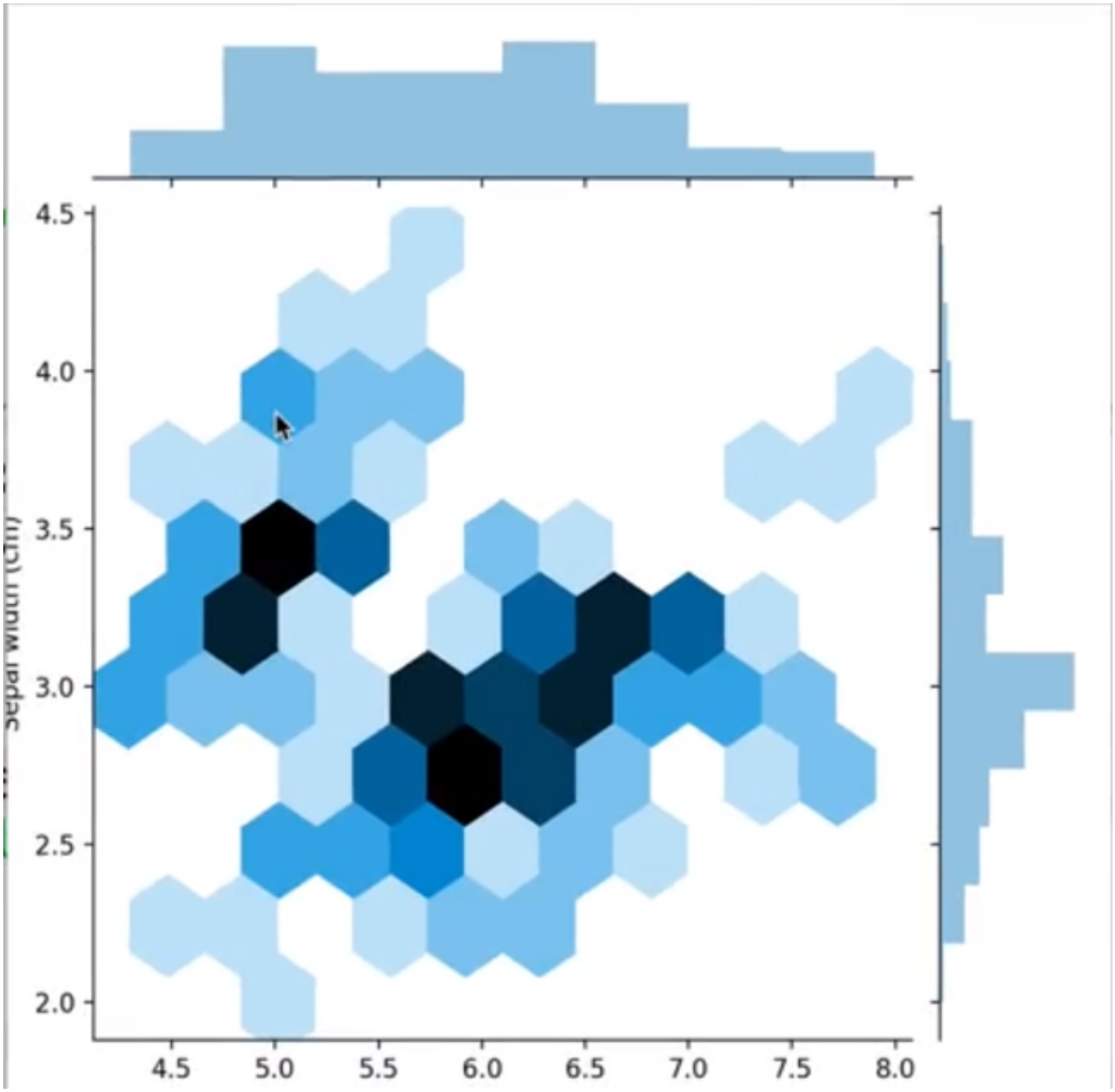


We see a **point density** now as opposed to individual points. The contours that represent various regions where point are concentrated. The lighter the color, the less points there are in that region. The darker region have the most points. We see that the darker regions correspond to the center of the histograms on the axes.

Let's now look at a **hex plot**. Lets modify the code above.

```
# hex plot
axes = sns.jointplot(sepal_length, sepal_width, kind='hex')
```


Running the code, we get:



We get hexagons that also represent the distribution of the points. White means there are no points in that region. As in the previous plots, the highest density points are in the same 2 regions represented by the darker hexagons.

EXERCISE : You can look at some other features from the Iris data set in a similar fashion. Go back to the lesson where we did multi plots and change the code to use join-plots as opposed to scatter plots and observe the distributions. For example plot *petal_length* and *petal_width* and observe the distribution.

The join-plots are a great starting point to do further statistical analysis.

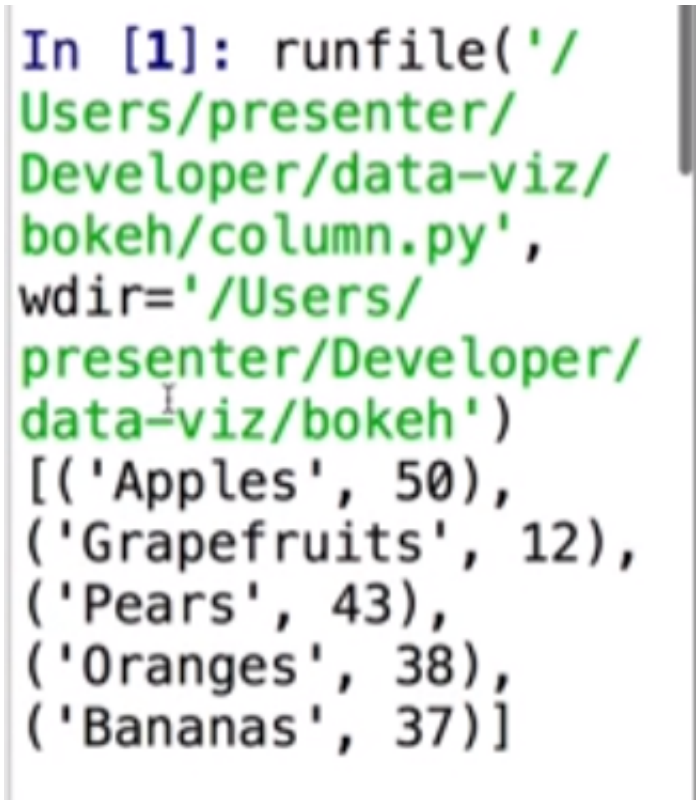
In this video, we are going to introduce a new framework, called Bokeh. Its a different kind of plotting framework compared to Seaborn or Matplotlib. Instead of displaying the plot directly, Bokeh creates a HTML page (file) with all the content embedded. You can do more interactive things with Bokeh than with the other frameworks. We can embedd these Bokeh plots in other web application. We are going to look a a couple of plots and how they can be used interactively.

We can do things like link 2 scatter plots in a way that when we move around in one plot, it does the corresponding motions in the other one.

Lets start with a simple column plot.

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

import pickle

# load data
with open ('fruit-sales.pickle', 'rb') as f :
    data = pickle.load()

print (data)
```

Let's run this code.



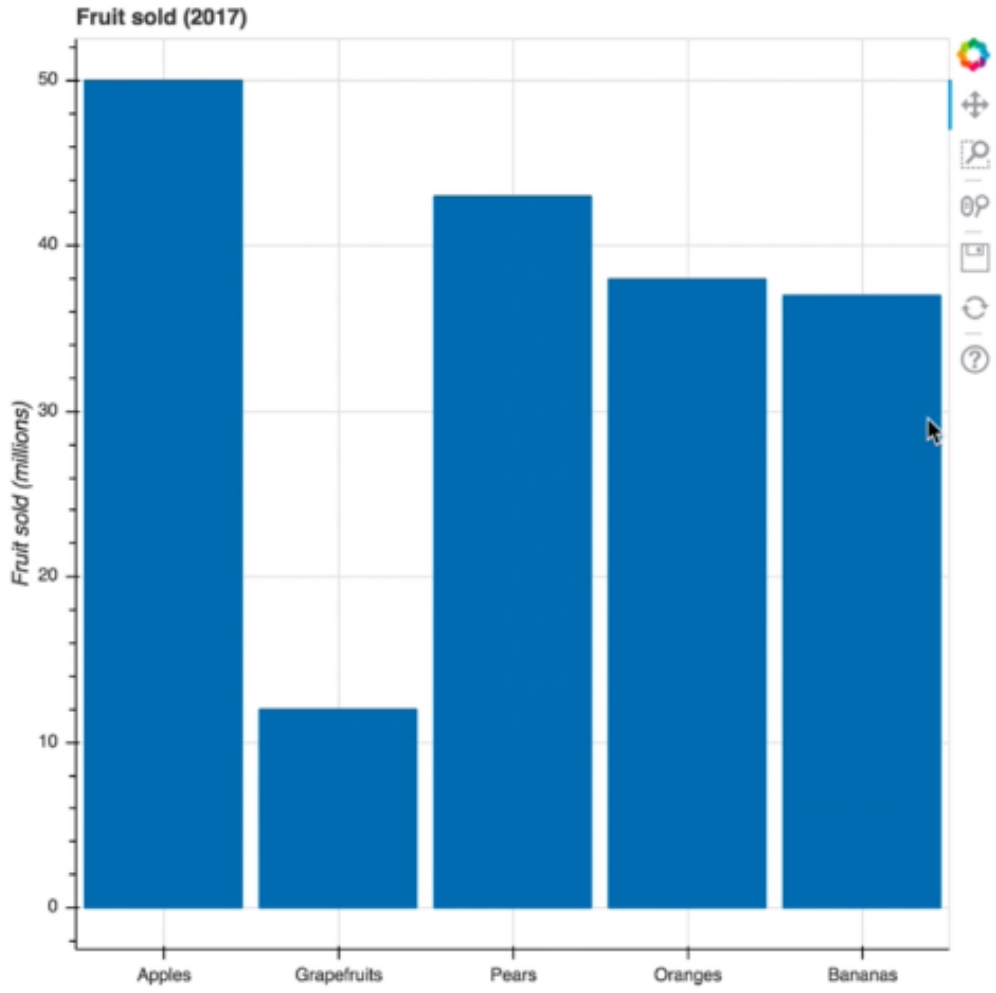
The data is a list of tuples, each containing the fruit name and the number sold.

```
# split data into 2 lists
fruit, num_sold = zip(*data)
```



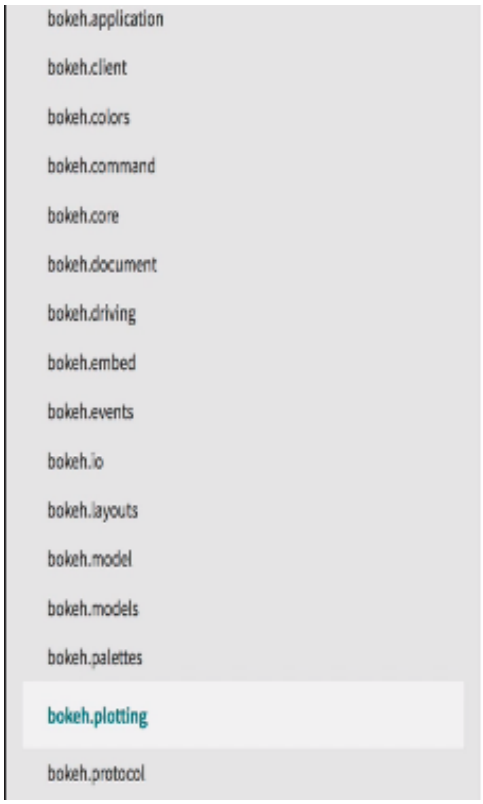
```
# create a plot and then manipulate that

# x-range shows fruit names
plot = figure(x_range=fruit, y_axis_label='Fruit sold (millions)', title='Fruit sold
(2017')

# vertical bars
plot.vbar(x=fruit, top=num_sold, width=0.9)

show(plot)
```

Let’s run this code. This will launch a web browser instance.



We see a toolbar to the top right. It has zoom controls, a icon to save the plot etc.

Lets’s look at the Bokeh API documentation for the *figure()* API we used above ( ).



Create a new `Figure` for plotting.

Figure objects have many glyph methods that can be used to draw vectorized graphical glyphs:

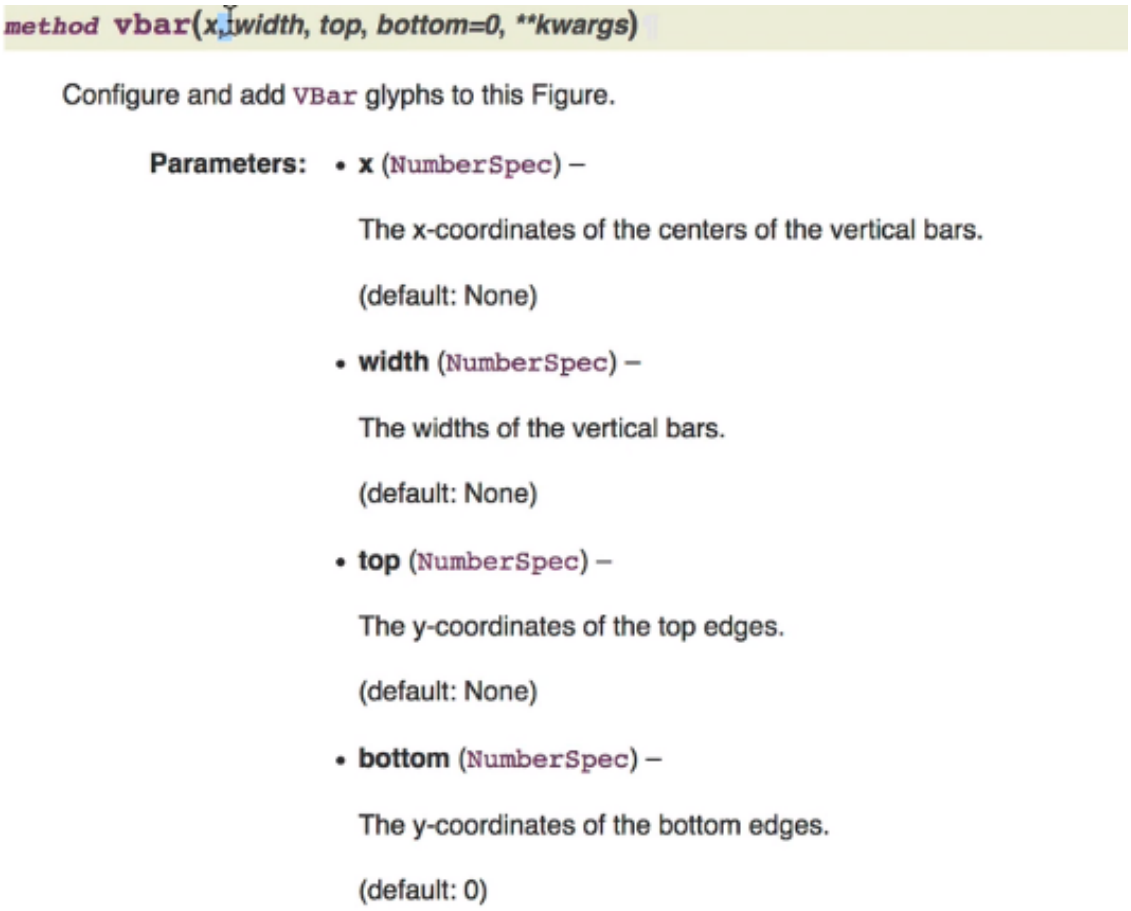
- annular_wedge()
- annulus()
- arc()
- asterisk()
- bezier()
- circle()
- circle_cross()
- circle_x()
- cross()
- diamond()
- diamond_cross()
- ellipse()
- hbar()
- hex()
- hex_tile()
- image()
- image_rgba()
- image_url()
- inverted_triangle()
- line()
- multi_line()
- oval()
- patch()
- patches()
- quad()
- quadratic()
- ray()
- rect()
- segment()
- square()
- square_cross()
- square_x()
- step()
- text()
- triangle()
- vbar()
- wedge()
- x()

There are also two specialized methods for stacking bars:

- hbar_stack()
- vbar_stack()

As you can see there are various methods to customize the figure such as setting the x-axis label, y-axis label etc.

Clicking on *vbar()*, we get the API reference.



method **vbar**(*x, width, top, bottom=0, **kwargs*)

Configure and add `VBar` glyphs to this Figure.

Parameters:

- **x** (`NumberSpec`) –

  The x-coordinates of the centers of the vertical bars.

  (default: None)

- **width** (`NumberSpec`) –

  The widths of the vertical bars.

  (default: None)

- **top** (`NumberSpec`) –

  The y-coordinates of the top edges.

  (default: None)

- **bottom** (`NumberSpec`) –

  The y-coordinates of the bottom edges.

  (default: 0)

It shows us the various parameters we can use. We can change the color of the bars, the width of the bars etc.

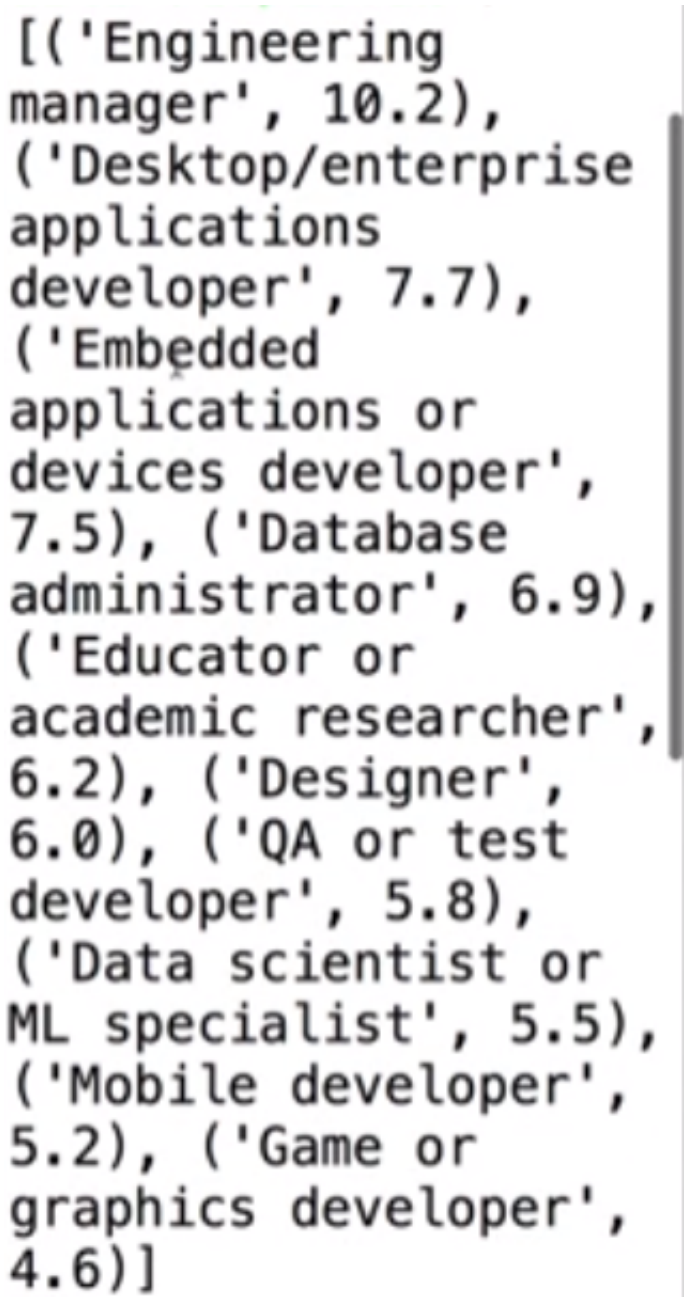
This is the first step in our foray with Bokeh. In future lessons, we will look at more kinds of charts and adding interactivity.

In this video, we are going to see how to create a **bar chart (horizontal bars)**. We'll also save the generated HTML file in the same folder so we can copy ad paste it wherever we need.

Let's get started.

```
from bokeh.io import show, output_file
from bokeh.plotting import figure
import pickle

# saves chart in HTML file
output_file('bar.html')

with open('coding-exp-by-dev-type.pickle', 'rb') as f :
    data = pickle.load(f)

print(data)
```

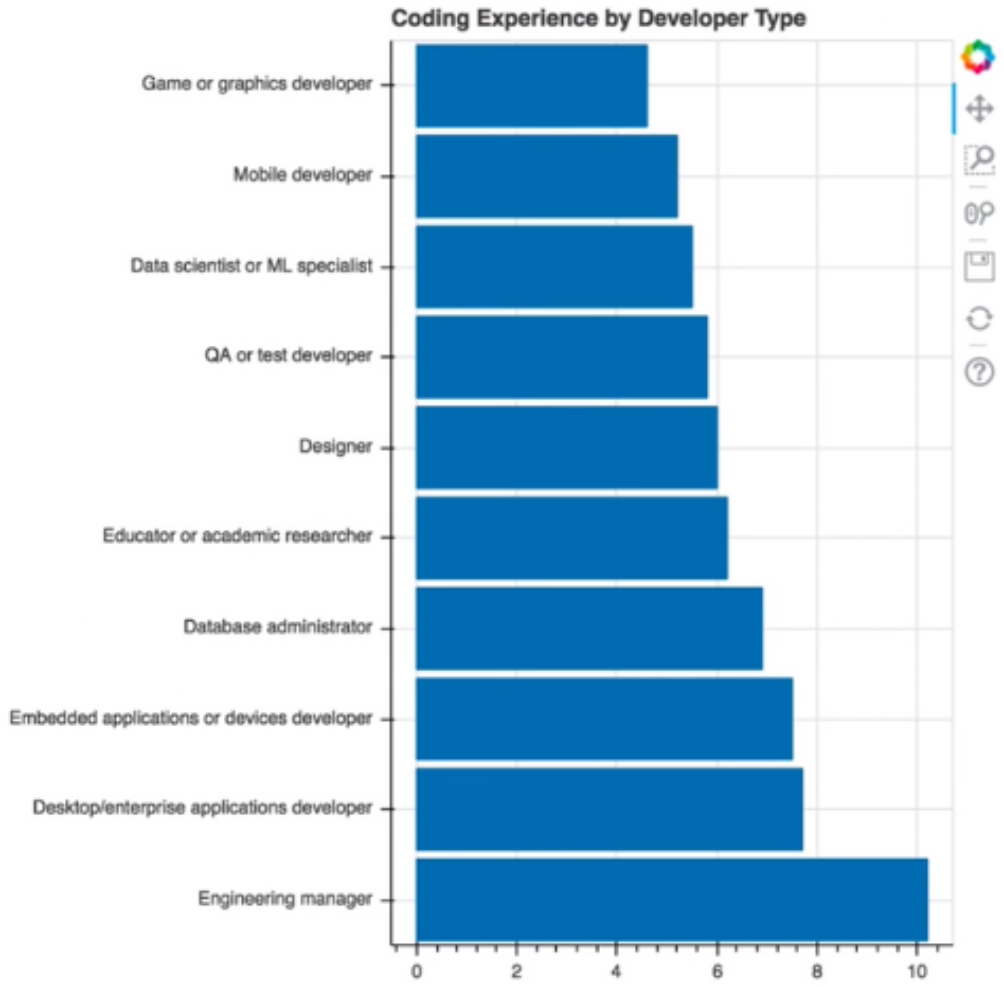
Let's run this to view the data.



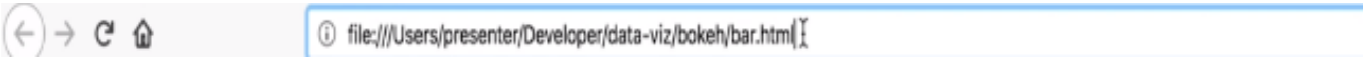
Let's split this data.

```
dev_types, years_exp = zip(*data)

# create figure
plot = figure(y_range=dev_types, x_axis_label = 'years', title='Coding Experience By Developer Type')
# bars oriented left -> right
plot.hbar(y=dev_types, right=years_exp, height=0.9 )

show(plot)
```

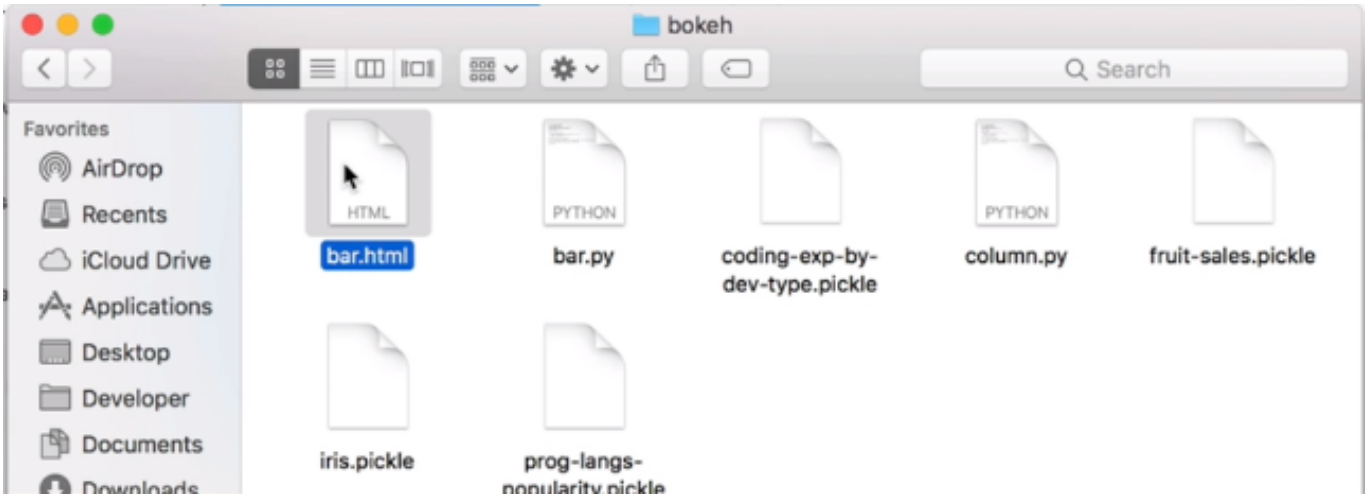
Let's run this code.



We have the years on the x-axis, the developer types on the y-axis. As the URL in the browser indicate, the plot is saved to *bar.html*.



Looking in the folder, we find the file:

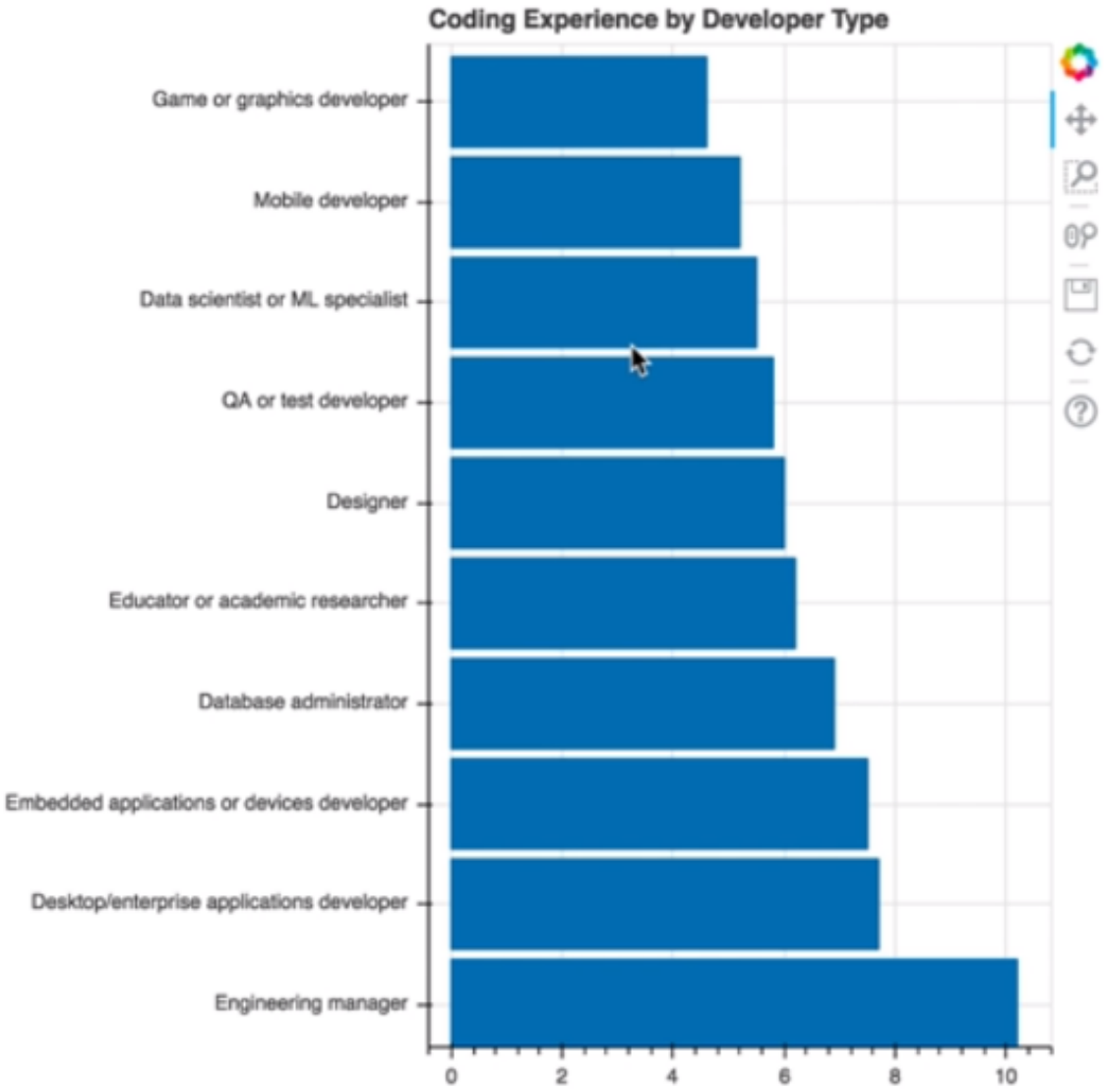


We can now open this file as needed, in a browser.

Summary

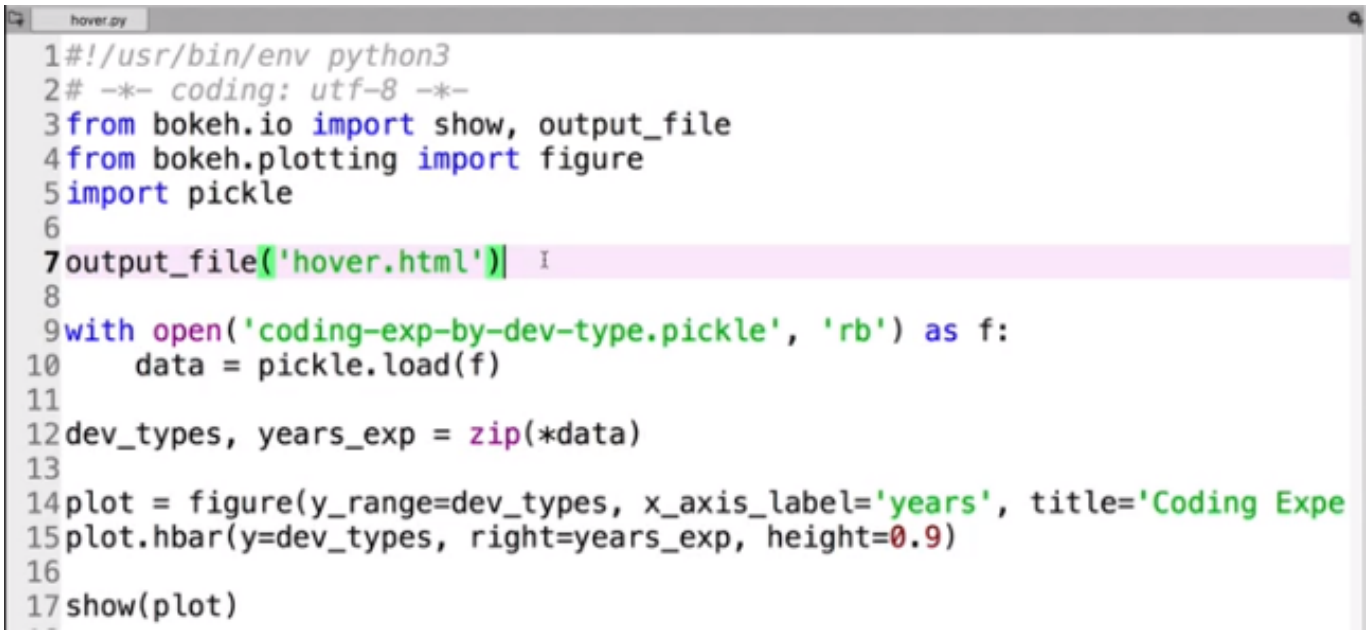
This is how we create a bar plot in Bokeh. We have to remember to flip x and y and remember the difference between *vbar* and *hbar*. We can save the output file in the same directory using the *output_file* function.

In this video, we are going to learn how to add **interactivity** to our bar chart.



Let’s add a **tool tip** to display the years of experience in the above chart.

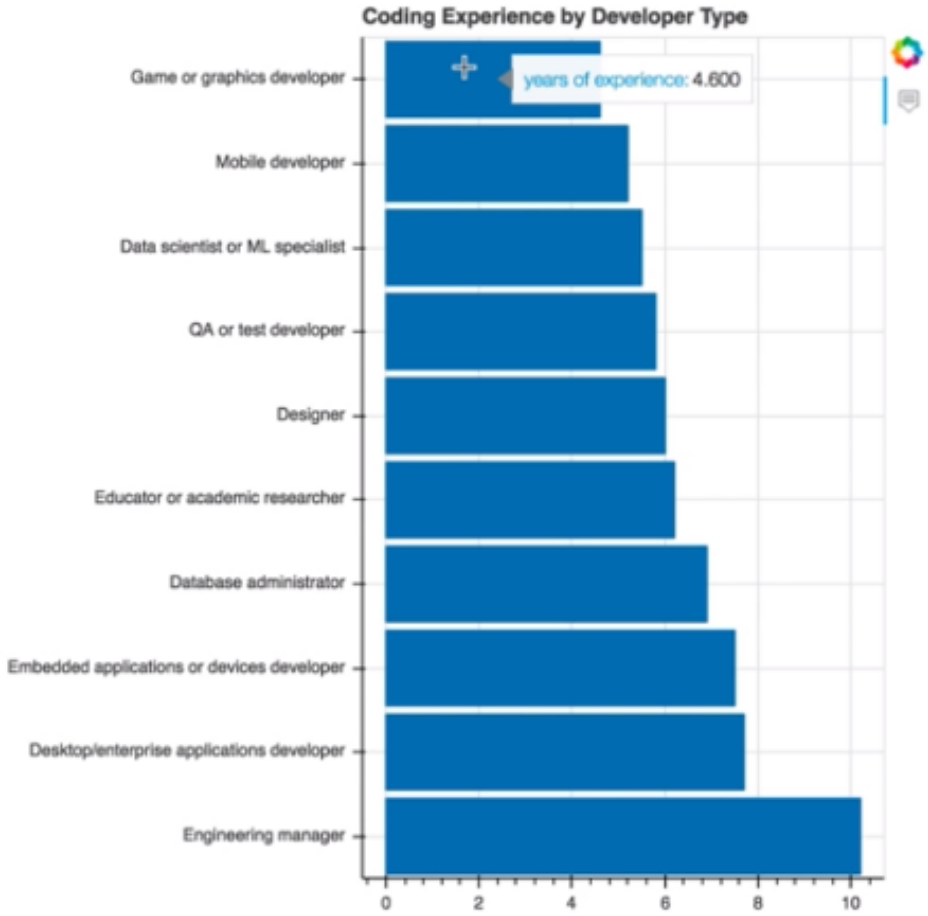
Let’s start with the code from the previous video.



We have renamed the output file to *hover.html*. New code is marked #NEW.

```
from bokeh.io import show, output_file
from bokeh.plotting import figure
import pickle

# saves chart in HTML file
output_file('hover.html')

with open('coding-exp-by-dev-type.pickle', 'rb') as f :
    data = pickle.load(f)

dev_types, years_exp = zip(*data)

#NEW
data source =  {'dev_types': dev_types, 'years_exp': years_exp}

#NEW tool tips format : “years of experience : actual value”
TOOLTIPS = [('years of experience', '@years_exp')]

# create figure
plot = figure(y_range=dev_types, x_axis_label = 'years', title='Coding Experience By
Developer Type', tools='hover', tooltips=TOOLTIPS)

# NEW: set data source, keys so it looks up values in data source dictionary
plot.hbar(y='dev_types', right='years_exp', height=0.9, source=data_source )

show(plot)
```

Let’s run this code.



As we hover over the bar, we can see the tool tip. The documentation shows us ways to customize the tool tips.

Summary

In this video, we saw how to add more interactivity to the Bokeh plot.

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

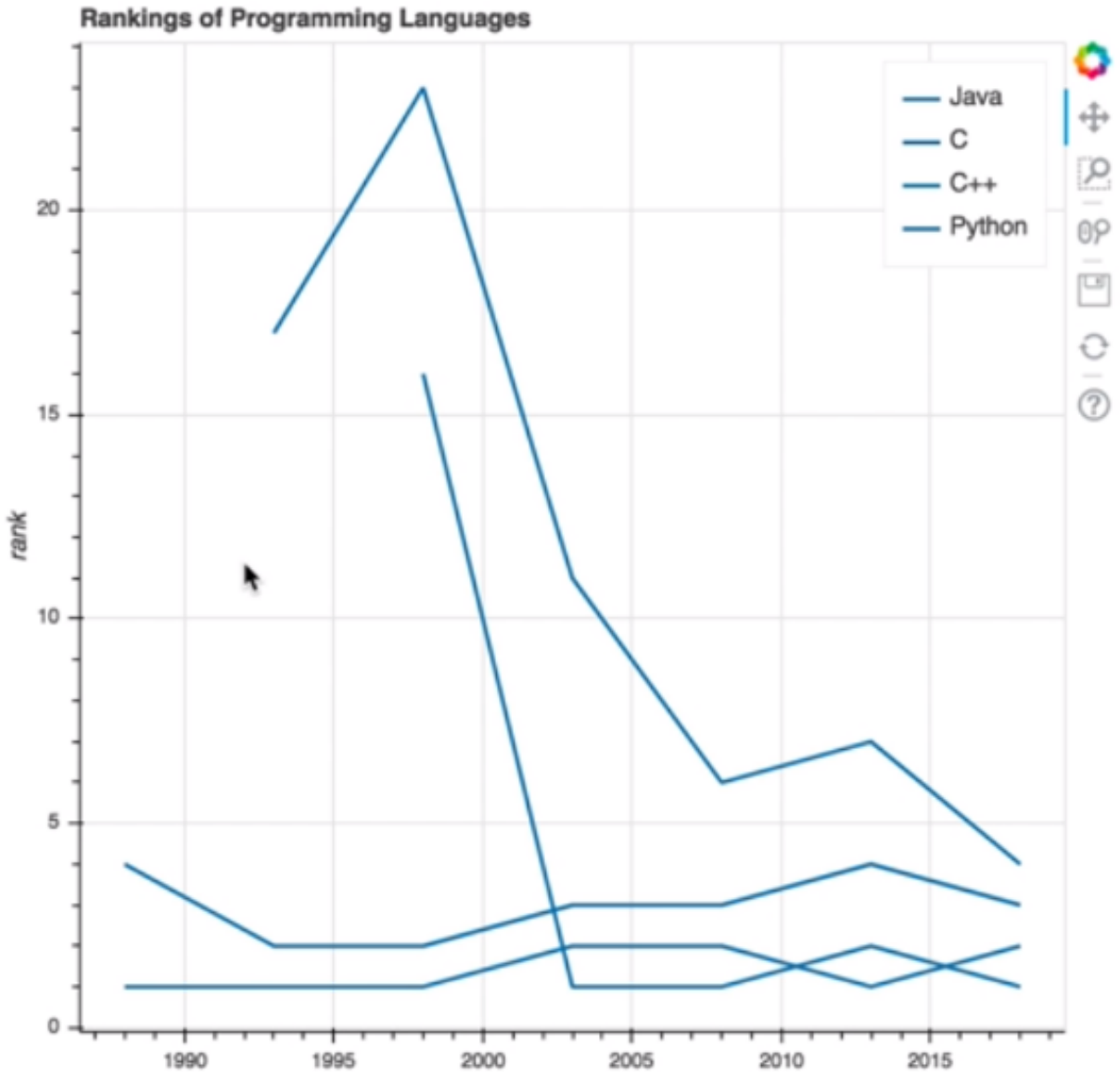
In this video, we will look at multi-line plots in Bokeh. Let's start with code from the previous video and then add code to iterate over the various rankings, creating a line for each programming language.

The following code has been updated, and differs from the video:

```
from bokeh.io import show, output_file
from bokeh.plotting import figure
import pickle

# saves chart in HTML file
output_file('multiline.html')

with open('prog-langs-popularity.pickle', 'rb') as f:
    data = pickle.load(f)

languages, rankings = zip(*data)

# create figure
fig = figure(x_axis_label = 'year', y_axis_label = 'rank', title='Rankings Of Program
ming Languages')

# iterate over rankings
for i in range(len(languages)):
    years, ranks = zip(*rankings[i])
    fig.line(years, ranks, line_width=2, legend_label=languages[i])

show(fig)
```

Note that 'legend' is deprecated and has been replaced by **'legend_label'**.

Let's run this code.



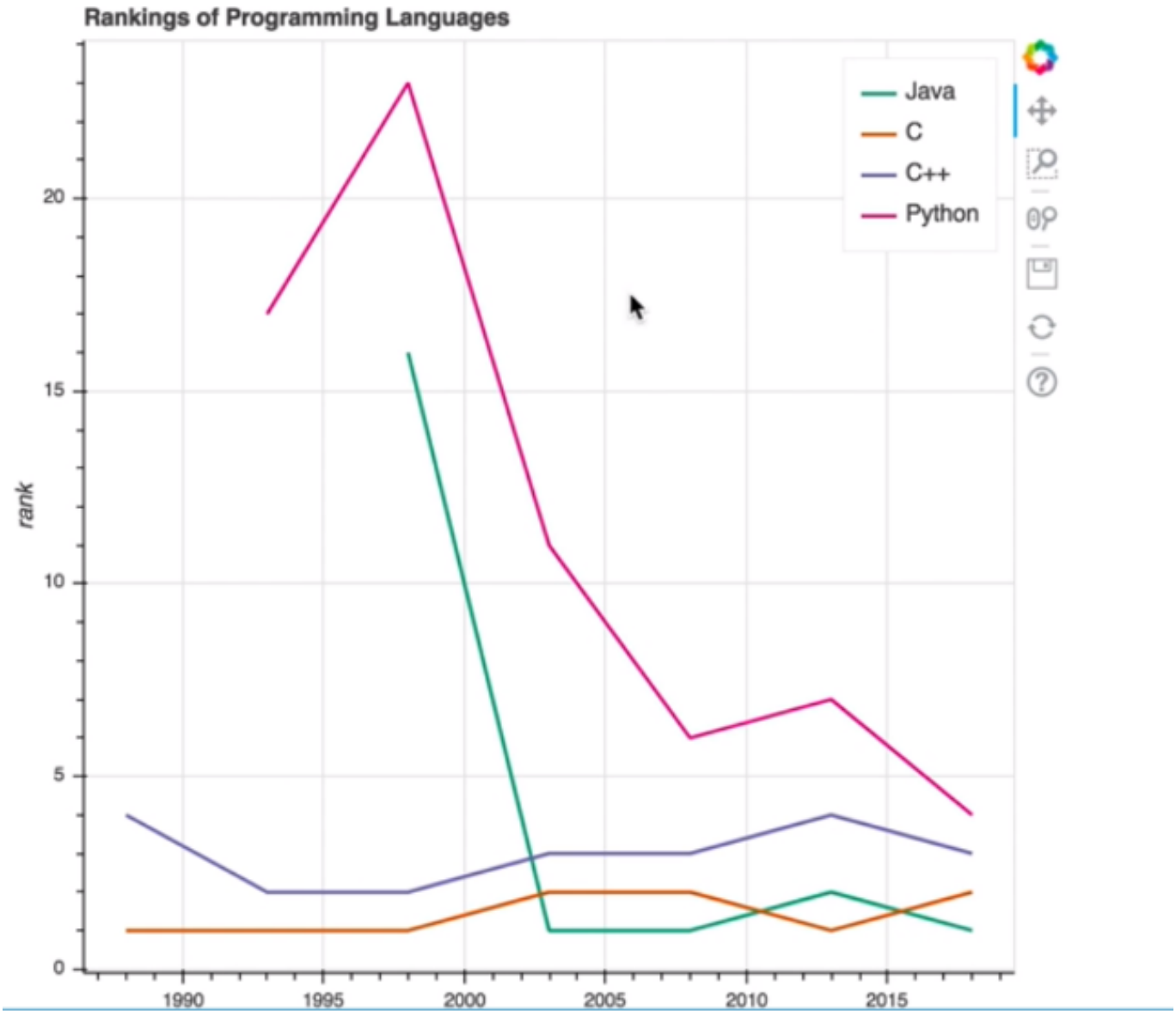
All of the lines have the same color. Let's change this so each programming language is depicted by a **different** color.

The following code has been updated, and differs from the video:

```
# add to imports
from bokeh.palettes import Dark2_5 as palette    # 5 colors in palette

#modify line below – legend and color for this particular line.
fig.line(years, ranks, line_width=2, legend_label=languages[i], color=palette[i])
```

NOTE: Bokeh's documentation lists several palettes that we can use for such plots.

Let's run this code.



Now we see that each of these lines are colored different. We can use the tools(right top) to **pan** around, **zoom** the plot, **save** it etc.

Summary

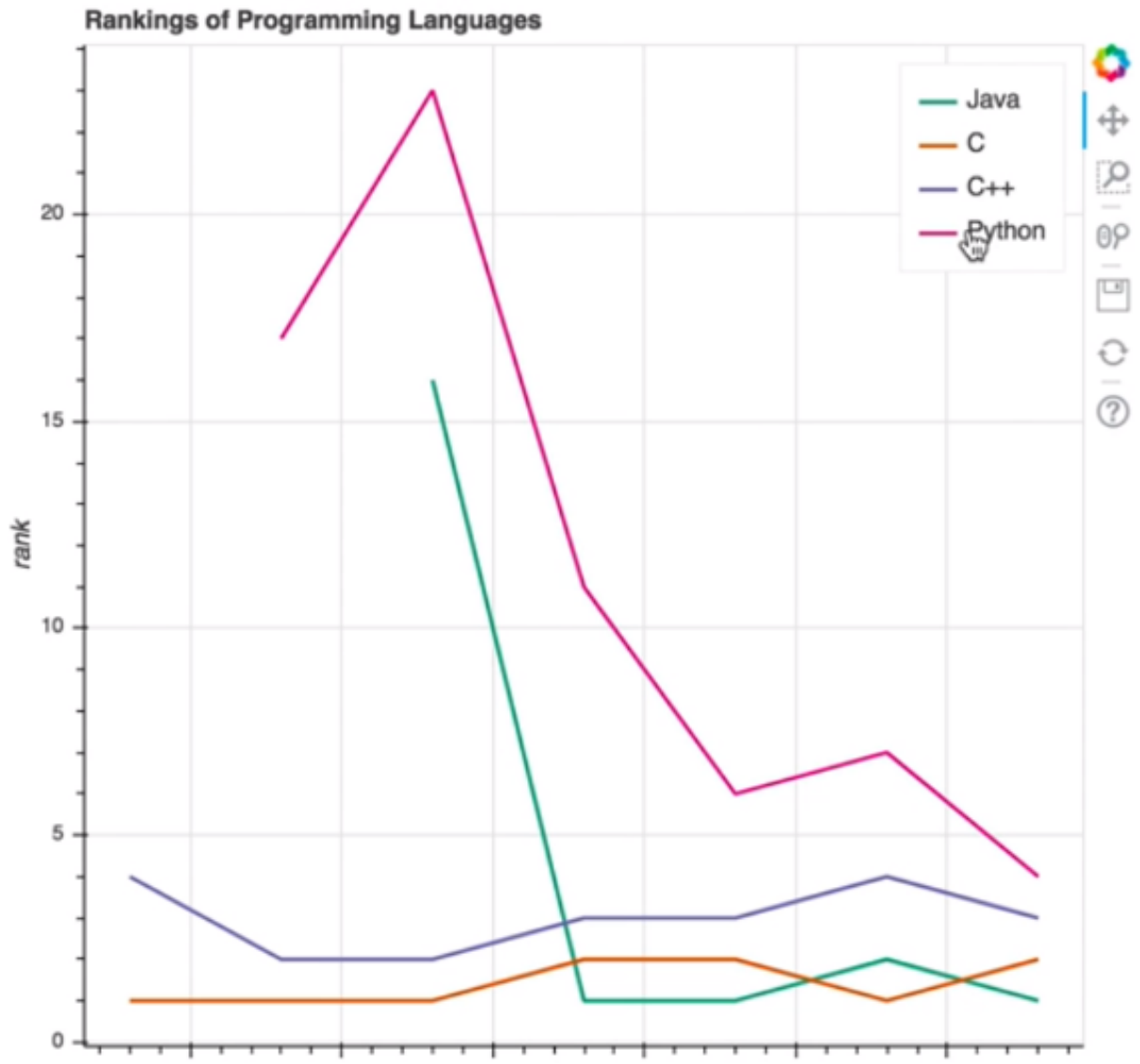
This is how we create a multi-line plot in Bokeh. We use the *fig.line()* method, specifying the x and y axes, the line width, the legend and the colors. Each time we call *fig.line()*, we create a new line.

In this video, we are going to add some interaction to the legend (New code is marked as #NEW). By clicking on the legend, I can hide different series of the line plot. This is a one-liner in Bokeh.

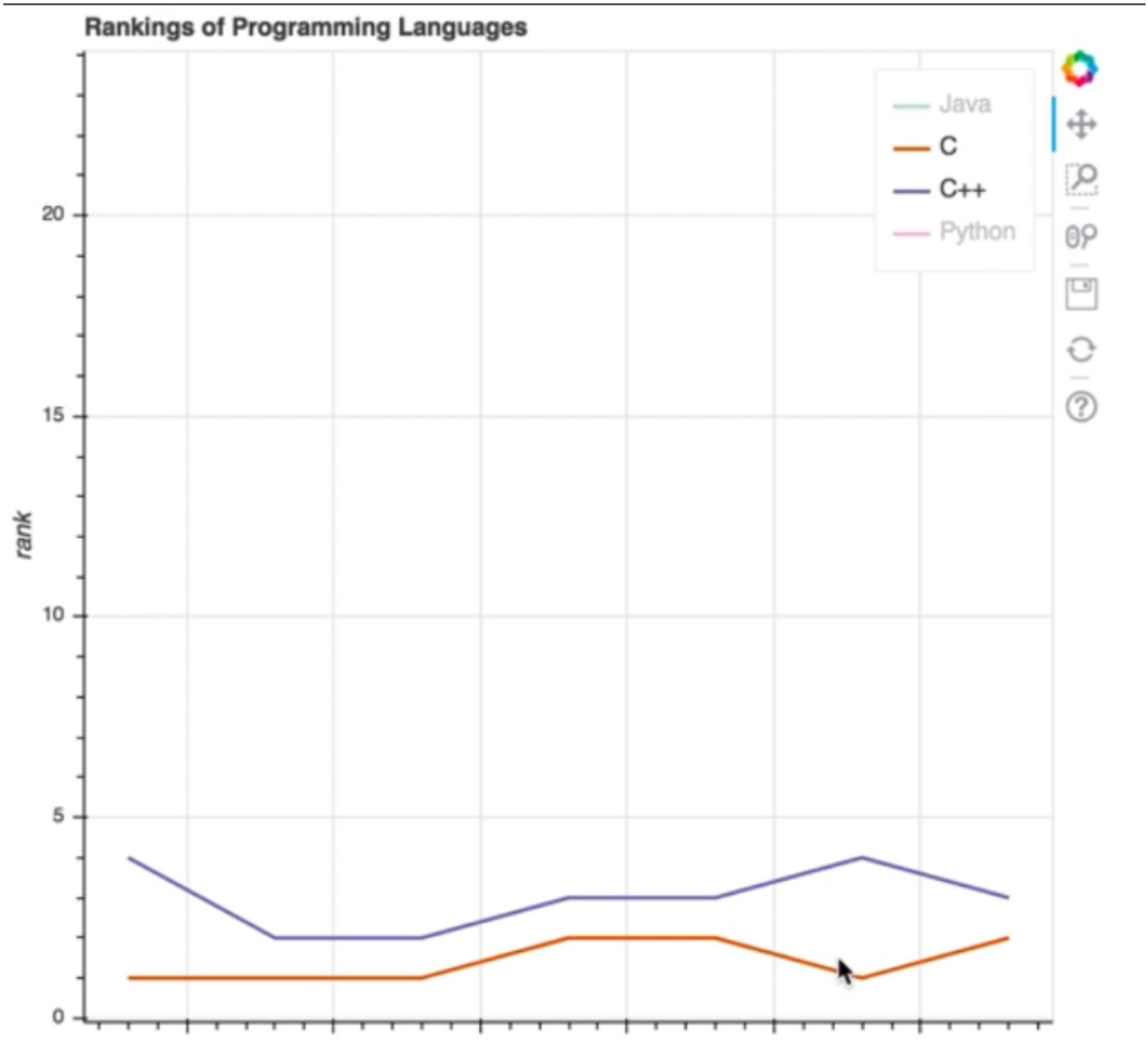
Let's start with the code from the previous video.

```
from bokeh.io import show, output_file
from bokeh.plotting import figure
#NEW
from bokeh.palettes import Dark2_5 as palette
import pickle

# saves chart in HTML file
output_file('multiline.html')

with open('prog-langs-popularity.pickle', 'rb') as f :
    data = pickle.load(f)

language, rankings = zip(*data)

fig = figure (x_axis_label='year', y_axis_label='rank', title='Rankings of  Programmi
ng Languages ')

for i in range (len(languages)) :
    years, ranks = zip(*rankings[i])
    fig.line(years, ranks, line_width=2, legend=languages[i], color=palette[i])

# NEW interactive legend – legend is clickable
fig.legend.click_policy = 'hide'
```

Let's run this code.



By clicking on the legend, we can make the some lines go away. Let’s look only at C and C++ by clicking on Java and Python to make them disappear.

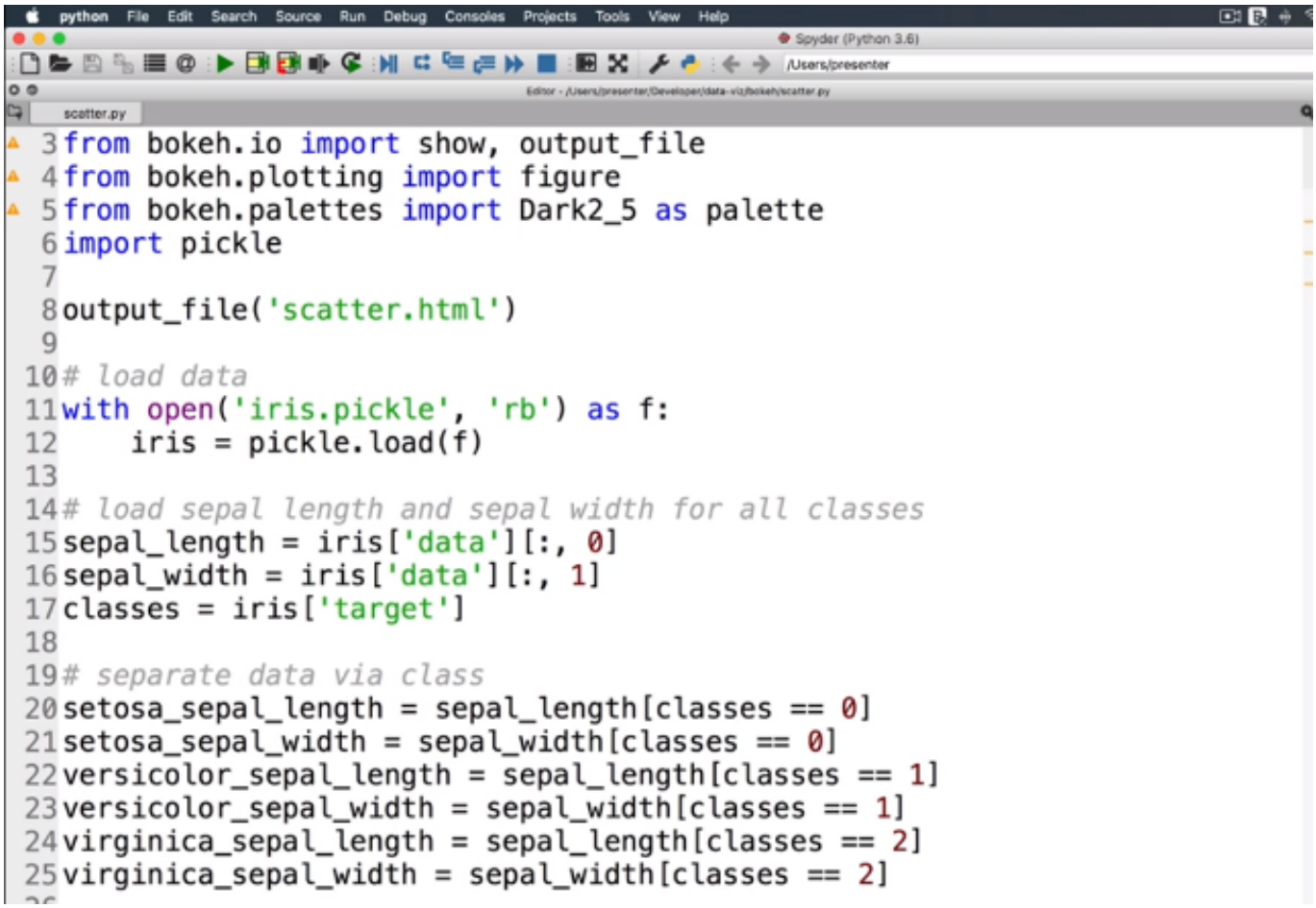


C tends to rank higher than C++.

Summary

This is a very useful tool to focus our attention on specific series of interest in the plot. It makes the plot interactive. We then simply click on a legend value to show hide the corresponding series. And all we did was to add a single of line of code to the plot.

In this video, we'll look at how to plot a scatter plot using Bokeh. Go into the ZIP file and look at *scatter.py* as there is a lot of setup that we've already done to extract the data and separate the data into different classes. The **Iris** data set has 3 different classes – corresponding to 3 different species of flowers i.e. *setosa*, *versicolor*, *virginica*. We extract the sepal length and sepal width for each of these classes.

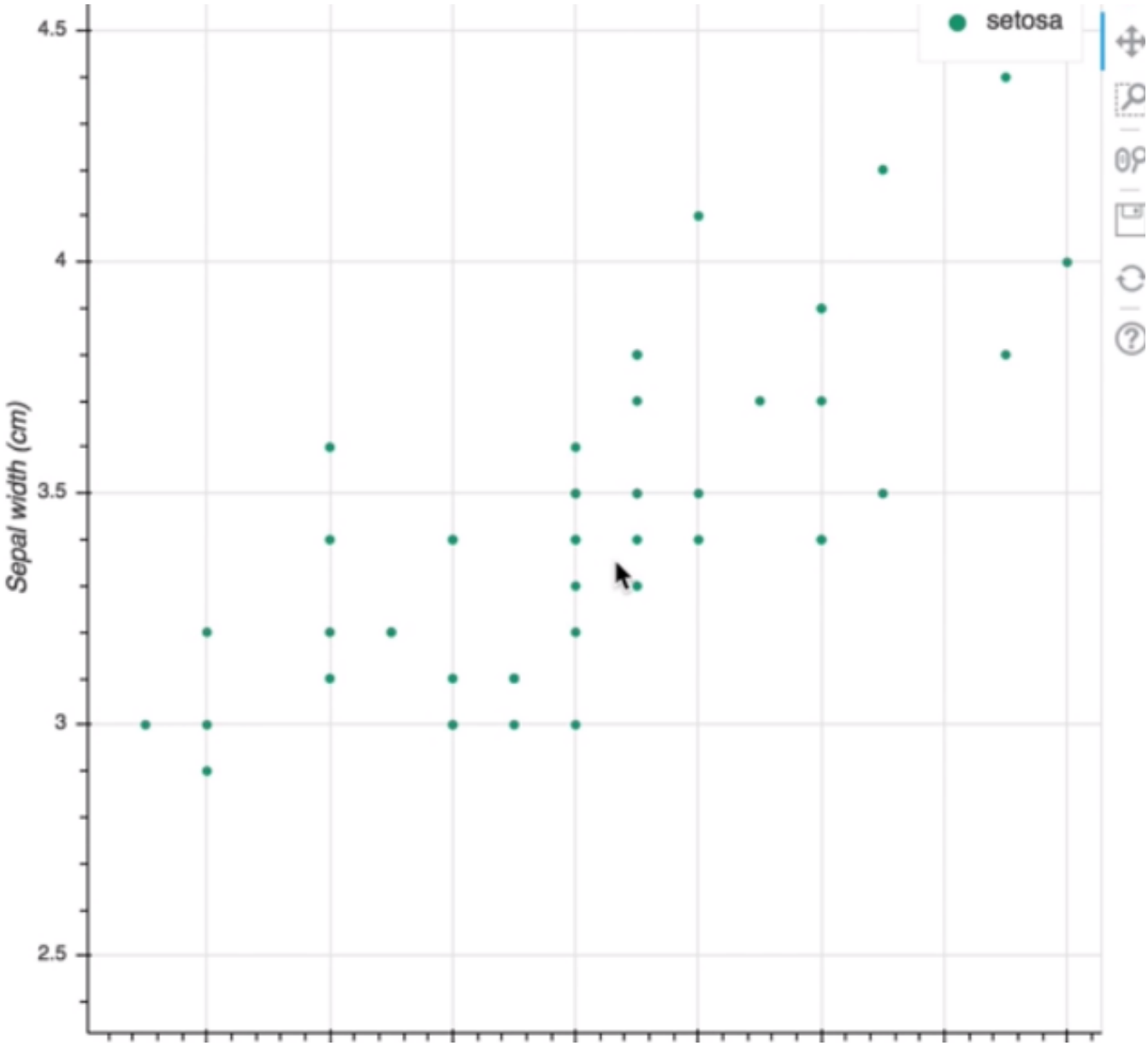


There are 150 data points, 50 of each class.

Let's add code to this to setup the figure and get the axes.

```
fig = figure(x_axis_label='Sepal length (cm)', y_axis_label='Sepal width (cm)')

# 3 plots – 1 each for each class (sepal length Vs sepal width).
# circle glyph – plot setosa lengths, widths
fig.circle(sentosa_sepal_length, sentosa_sepal_width, color=palette[0], legend='setos
a')

# show
show(fig)
```

Let's run this code.

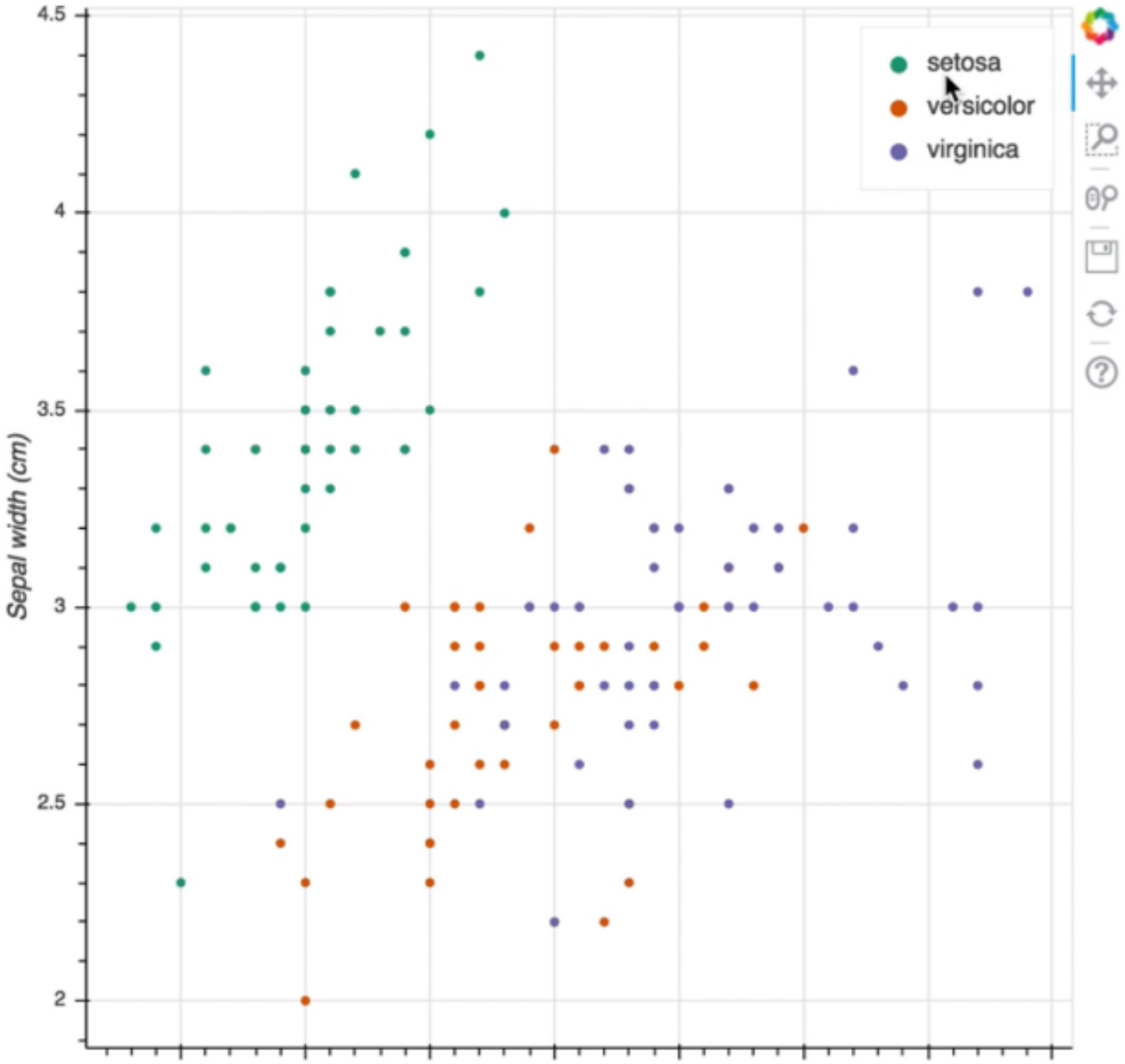


We see our scatter plot.

CHALLENGE : Plot the *versicolor*,sepal length Vs width and the *virginica* sepal length Vs width.

```
# versicolor sepal length Vs sepal width
fig.circle(versicolor_sepal_length, versicolor_sepal_width, color=palette[1], legend=
'versicolor')

# virginica sepal length Vs sepal width
fig.circle(virginica_sepal_length, virginica_sepal_width, color=palette[2], legend='v
irginica')
```

Let’s run this code.

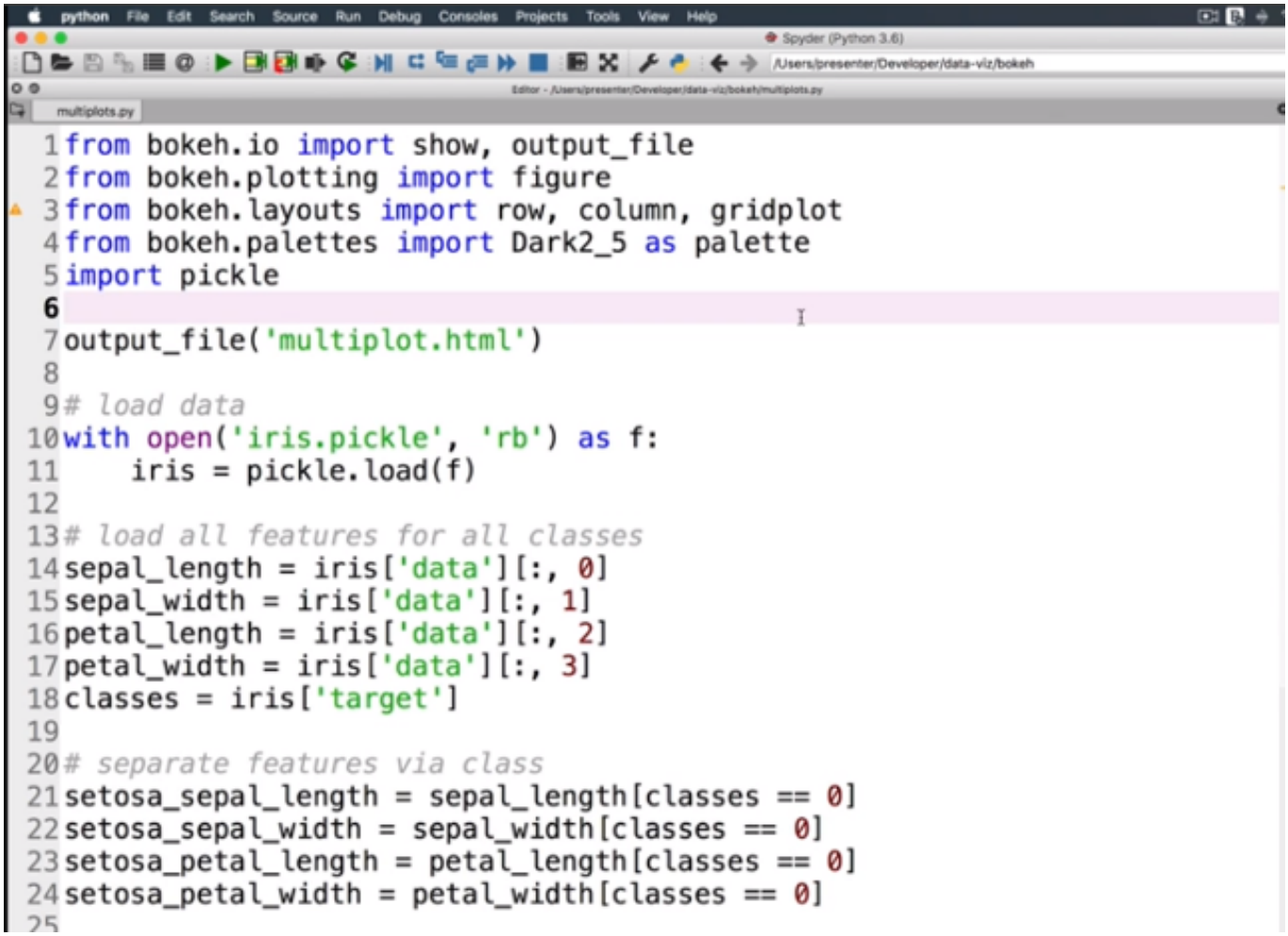


We can see the different classes depicted in different colors.

Suppose we wanted to make this into a classification problem, the plot shows us that most of the *sentosa* flowers (green color) are confined to the left side of the plot. It is easy to imagine a line separating the *sentosa* flowers from the other flowers. However it is trickier to separate the *versicolor* and *virginica* flower from each other.

Circle glyphs are not the only kind of glyphs we can use. Looking at the documentation, we can see other kinds of markers like *square*, *line* glyphs etc.

In this video, we are going to look at how to make **multiple** plots in Bokeh. We will look at different kinds of plots and how we can arrange them. Open the code ZIP file and extract *multiplots.py*.



```
1 from bokeh.io import show, output_file
2 from bokeh.plotting import figure
3 from bokeh.layouts import row, column, gridplot
4 from bokeh.palettes import Dark2_5 as palette
5 import pickle
6
7 output_file('multiplot.html')
8
9 # load data
10 with open('iris.pickle', 'rb') as f:
11     iris = pickle.load(f)
12
13 # load all features for all classes
14 sepal_length = iris['data'][:, 0]
15 sepal_width = iris['data'][:, 1]
16 petal_length = iris['data'][:, 2]
17 petal_width = iris['data'][:, 3]
18 classes = iris['target']
19
20 # separate features via class
21 setosa_sepal_length = sepal_length[classes == 0]
22 setosa_sepal_width = sepal_width[classes == 0]
23 setosa_petal_length = petal_length[classes == 0]
24 setosa_petal_width = petal_width[classes == 0]
25
```

We already have code to import the relevant packages, load the data, load the features for all the 3 classes – sentosa, versicolor, virginica. At the bottom, we have code to create the figure and scatter plots for the 3 classes. This code has been taken from *scatterplot.py*.

Let's create multiple plots. All the magic is in the *show()* function. The recipe for creating multiple plots is to create multiple figures and tell the show function how to arrange these figures.

Let's rename the above **fig** object to **fig1**.

```
fig1 = figure(x_axis_label='Sepal length (cm)', y_axis_label='Sepal width
fig1.circle(setosa_sepal_length, setosa_sepal_width, color=palette[0], le
fig1.circle(versicolor_sepal_length, versicolor_sepal_width, color=palett
fig1.circle(virginica_sepal_length, virginica_sepal_width, color=palette[
```

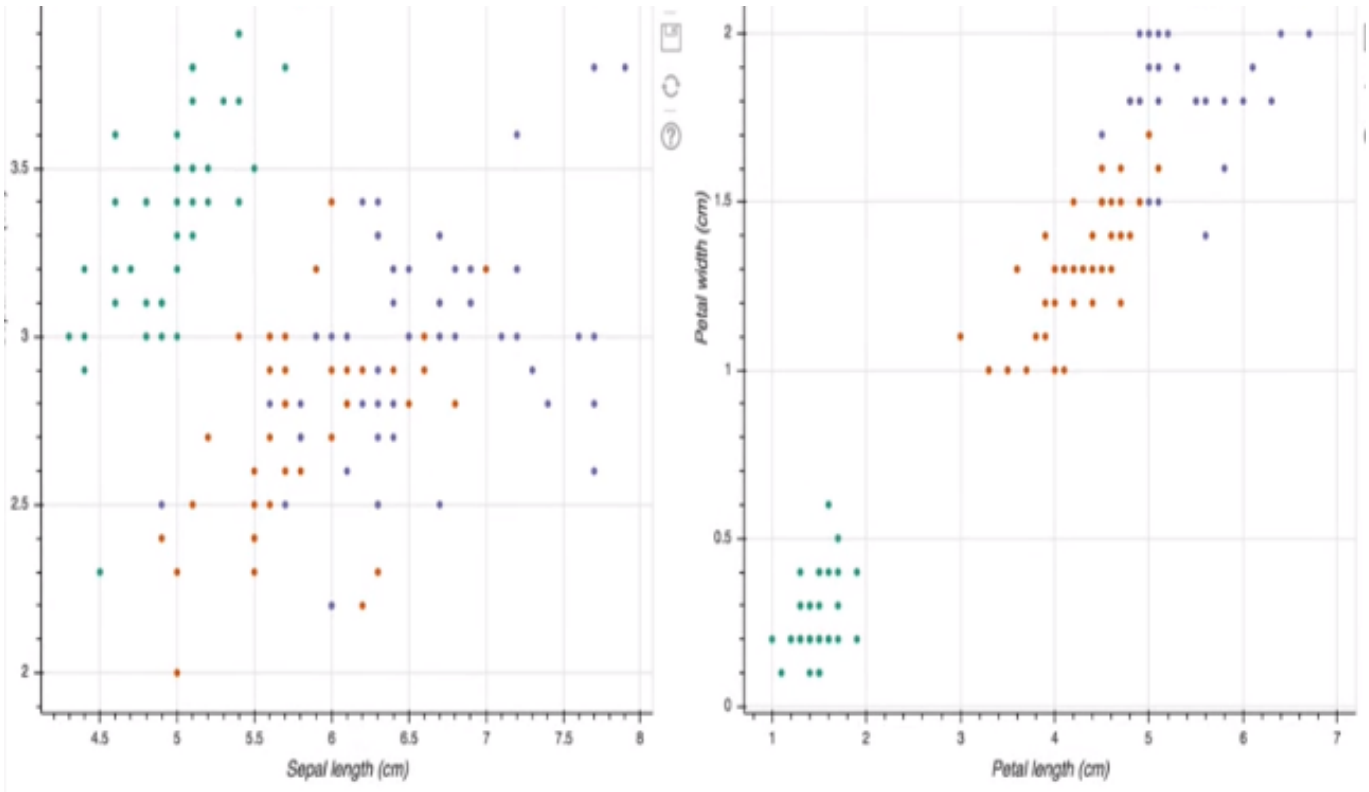
Let's create another figure.

```
# NEW
fig2 = figure(x_axis_label='Petal length (cm)', y_axis_label='Petal width (cm)')
fig2.circle(sentosa_petal_length, sentosa_petal_width, color=palette[0], legend='seto
sa')
fig2.circle(versicolor_petal_length, versicolor_petal_width, color=palette[1], legend
```



```
='versicolor')
fig2.circle(virginica_petal_length, virginica_petal_width, color=palette[2], legend='
virginica')

# the plots we want to show in a row
show(row([fig1, fig2]) )
```

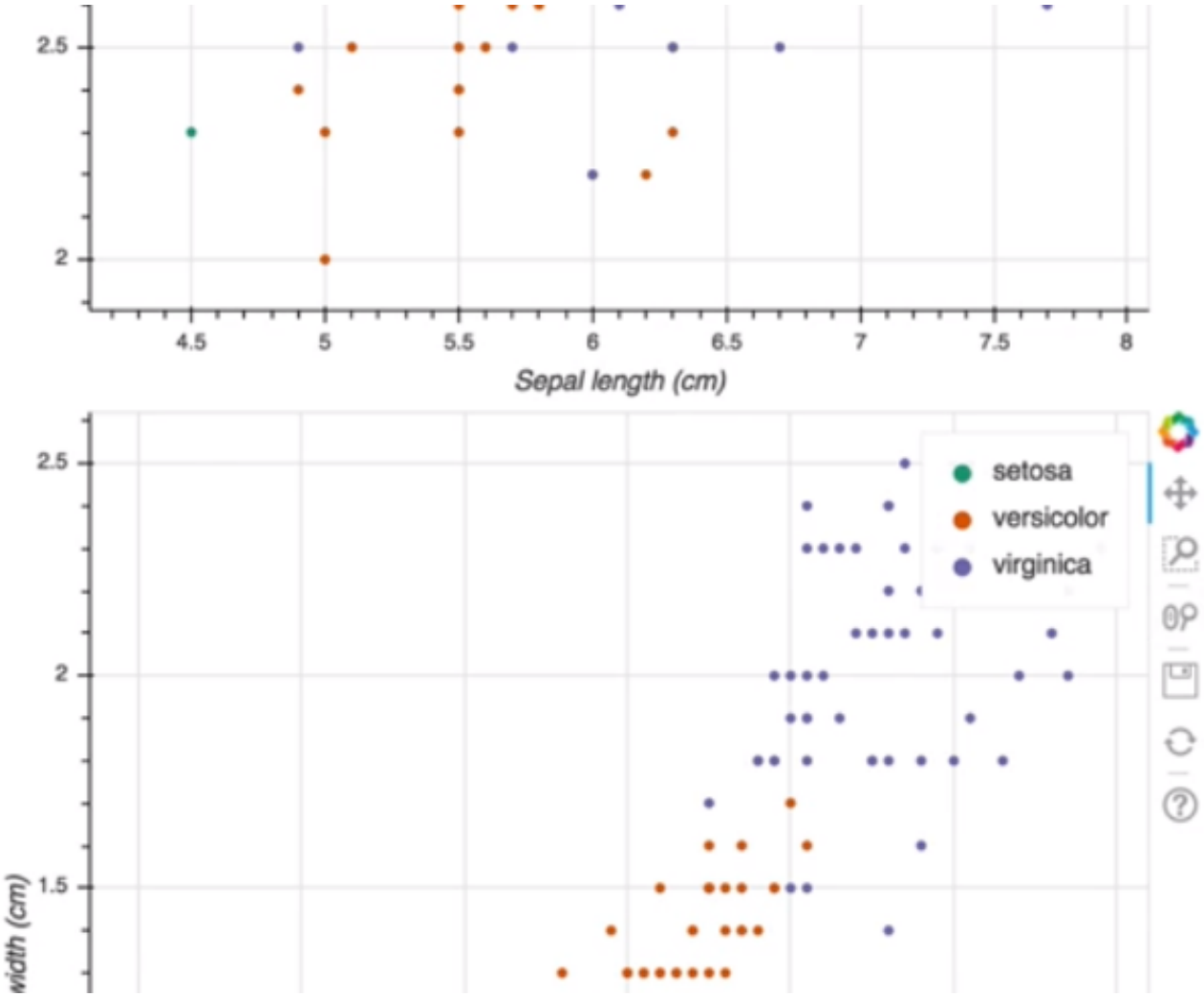
Let's run this code.



We can see the plots arranged in a row. We can do this for any number of plots. Let's plot them in a column instead.

```
# the plots we want to show in a column
show(column([fig1, fig2]) )
```

Let's run this code. Now we see the figures arranged in a column.

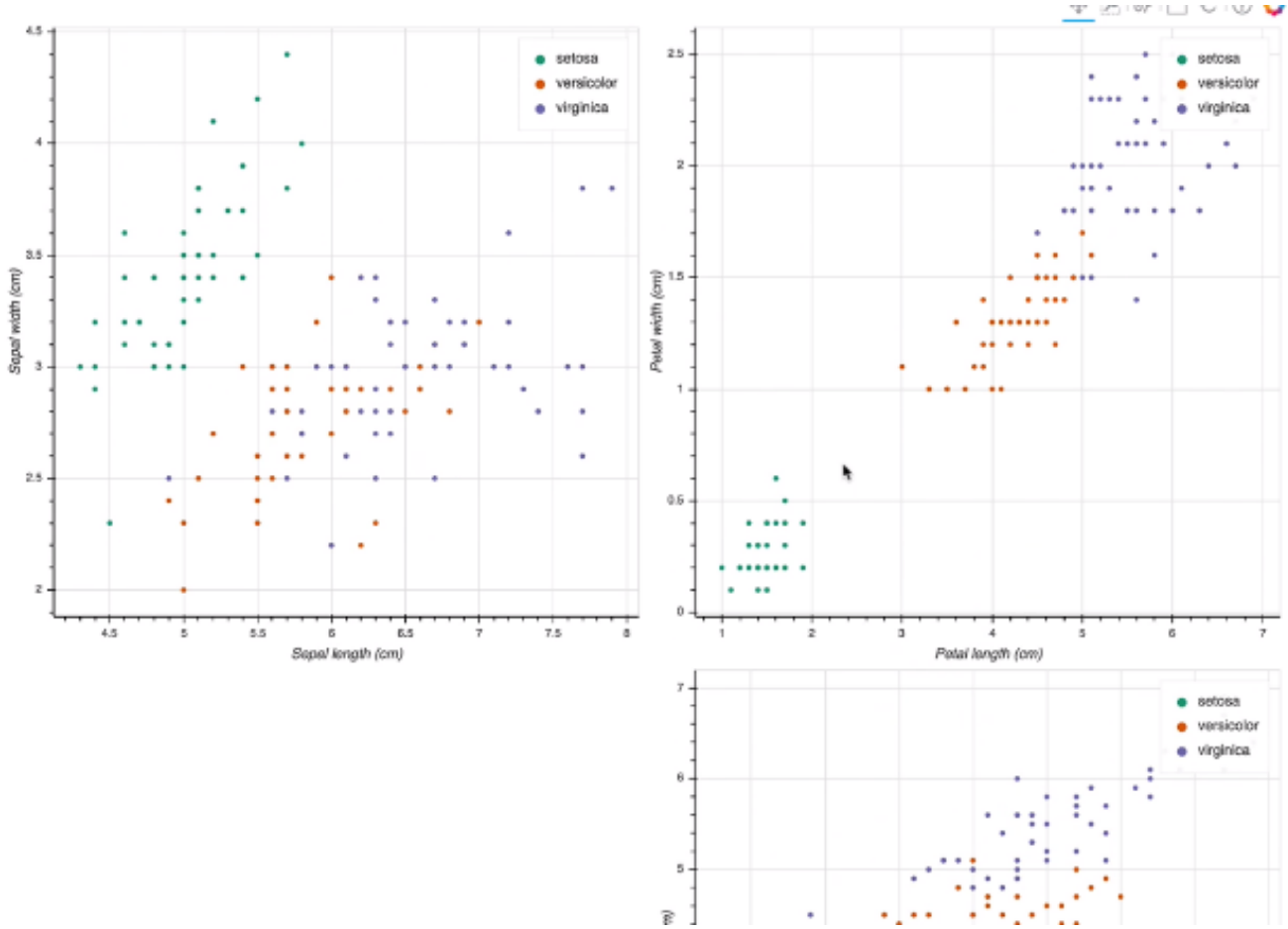


Let's now discuss **grid** plots. Let's add one more figure here.

CHALLENGE : Add code to plot sepal length Vs the petal length

```
# NEW
fig3 = figure(x_axis_label='Sepal length (cm)', y_axis_label='Petal length (cm)')
fig3.circle(sentosa_sepal_length, sentosa_petal_length, color=palette[0], legend='set
osa')
fig3.circle(versicolor_sepal_length, versicolor_petal_length, color=palette[1], legen
d='versicolor')
fig3.circle(virginica_sepal_length, virginica_petal_length, color=palette[2], legend=
'virginica')

# grid plot – pass in list of lists
# put fig3 under fig2
show(gridplot([[fig1, fig2], [None, fig3]]))
```

Let's run this code.



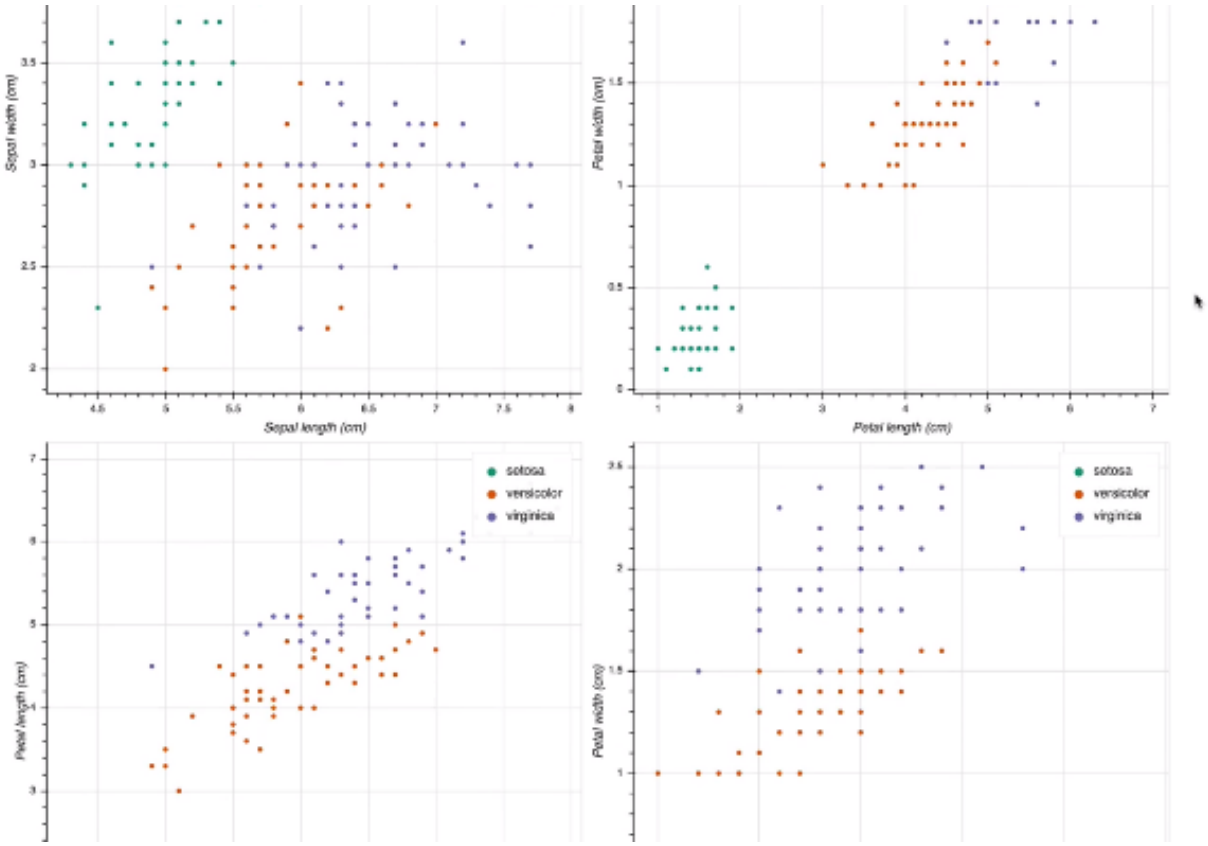
We see a row-column arrangement of the plot. The plot below fig1 is a **blank** plot since we have specified **None** for that position.

Let's create another plot.

CHALLENGE : Write code to plot sepal width Vs the petal width.

```
# NEW
fig4 = figure(x_axis_label='Sepal width (cm)', y_axis_label='Petal width (cm)')
fig4.circle(sentosa_sepal_width, sentosa_petal_width, color=palette[0], legend='setosa')
fig4.circle(versicolor_sepal_width, versicolor_petal_width, color=palette[1], legend='versicolor')
fig4.circle(virginica_sepal_width, virginica_petal_width, color=palette[2], legend='virginica')

# put fig3 under fig1, fig4 under fig2
show(gridplot([[fig1, fig2], [fig3, fig4]]))
```

Let's run this code.



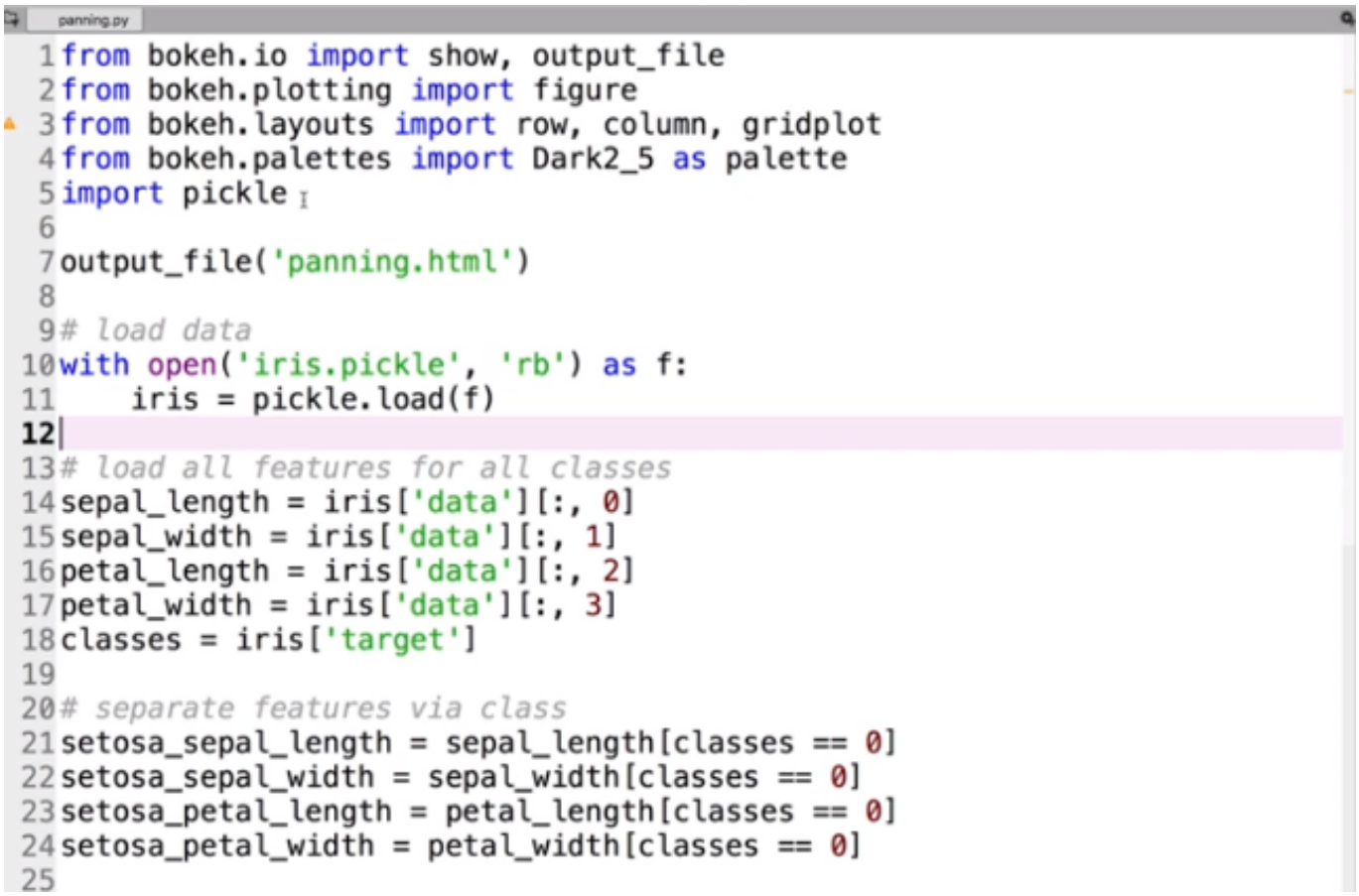
We see the plots arranged in a 2×2 grid.

Summary

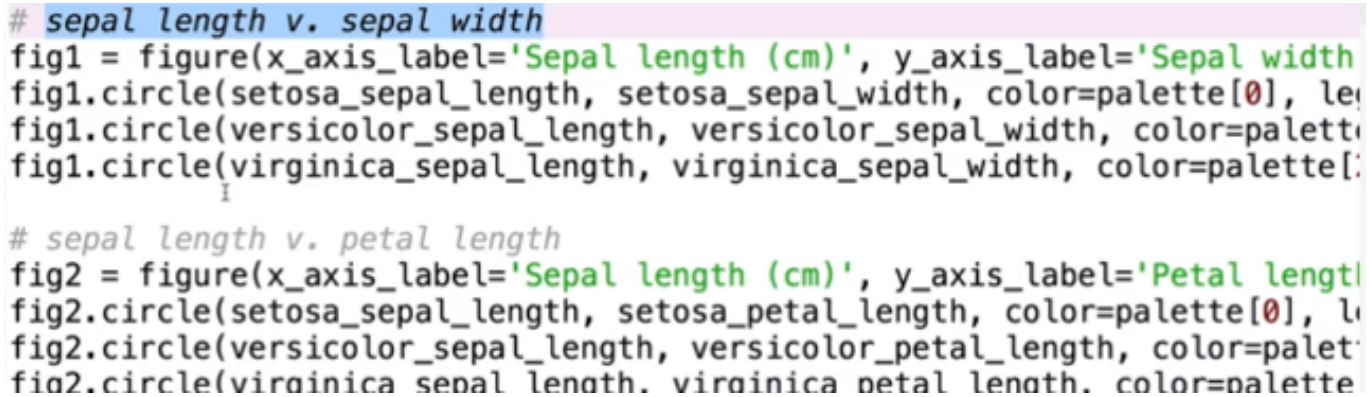
This is how we create multi plots in Bokeh. We create a figure object for each plot, and the use functions like *row*, *column*, *gridplot* in the *show()* function to arrange the figures in a kind of tiling fashion.

In this video, we will look at how we can link our axes together, so that we get **linked panning** i.e. if we move a certain distance in one grid, the plot will move the same distance in the another grid, along a particular axis.

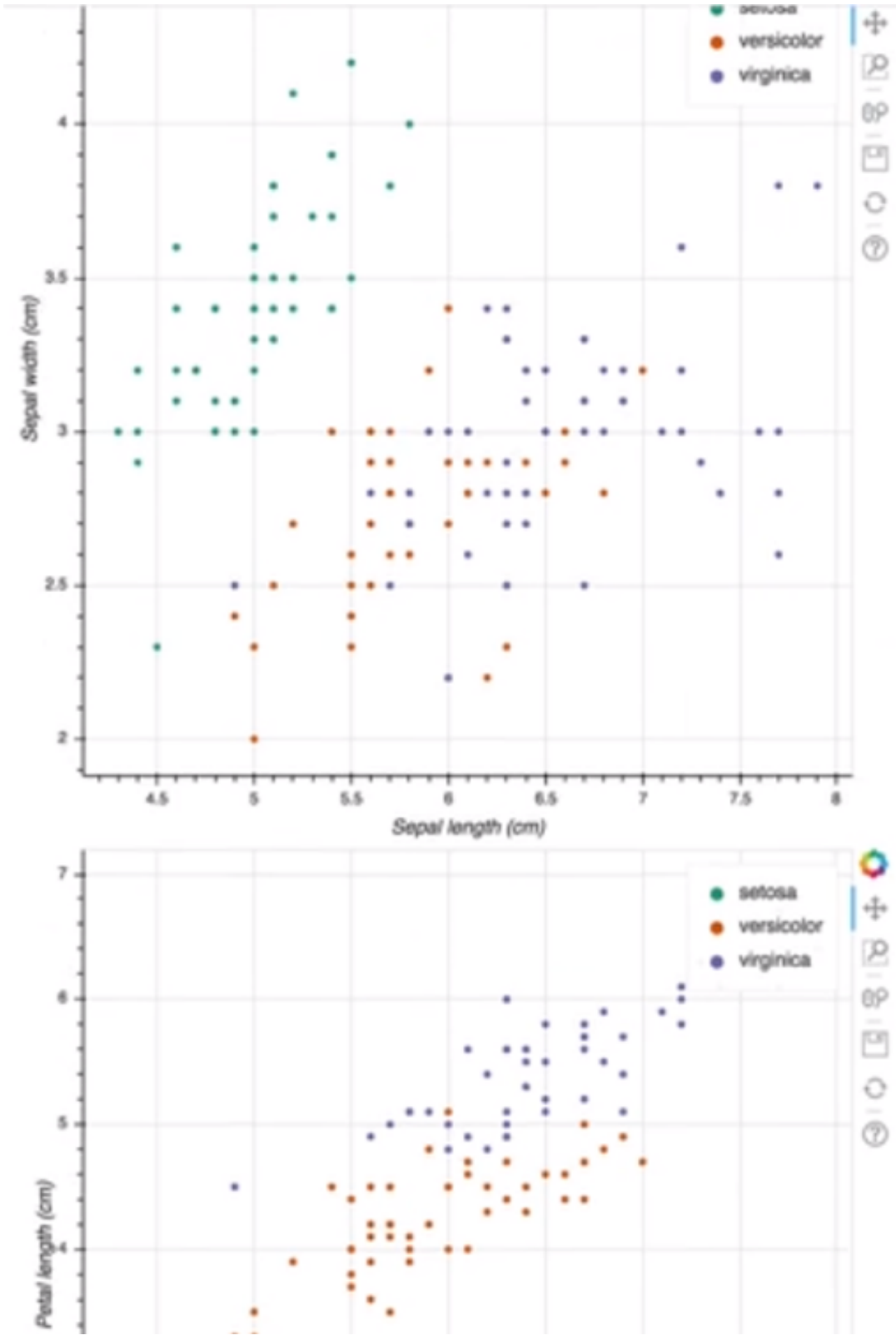
Download the source code and open this file called *panning.py*.



Going to the bottom of the code, we see 2 plots already in there i.e. sepal length Vs sepal width and sepal length Vs petal length.

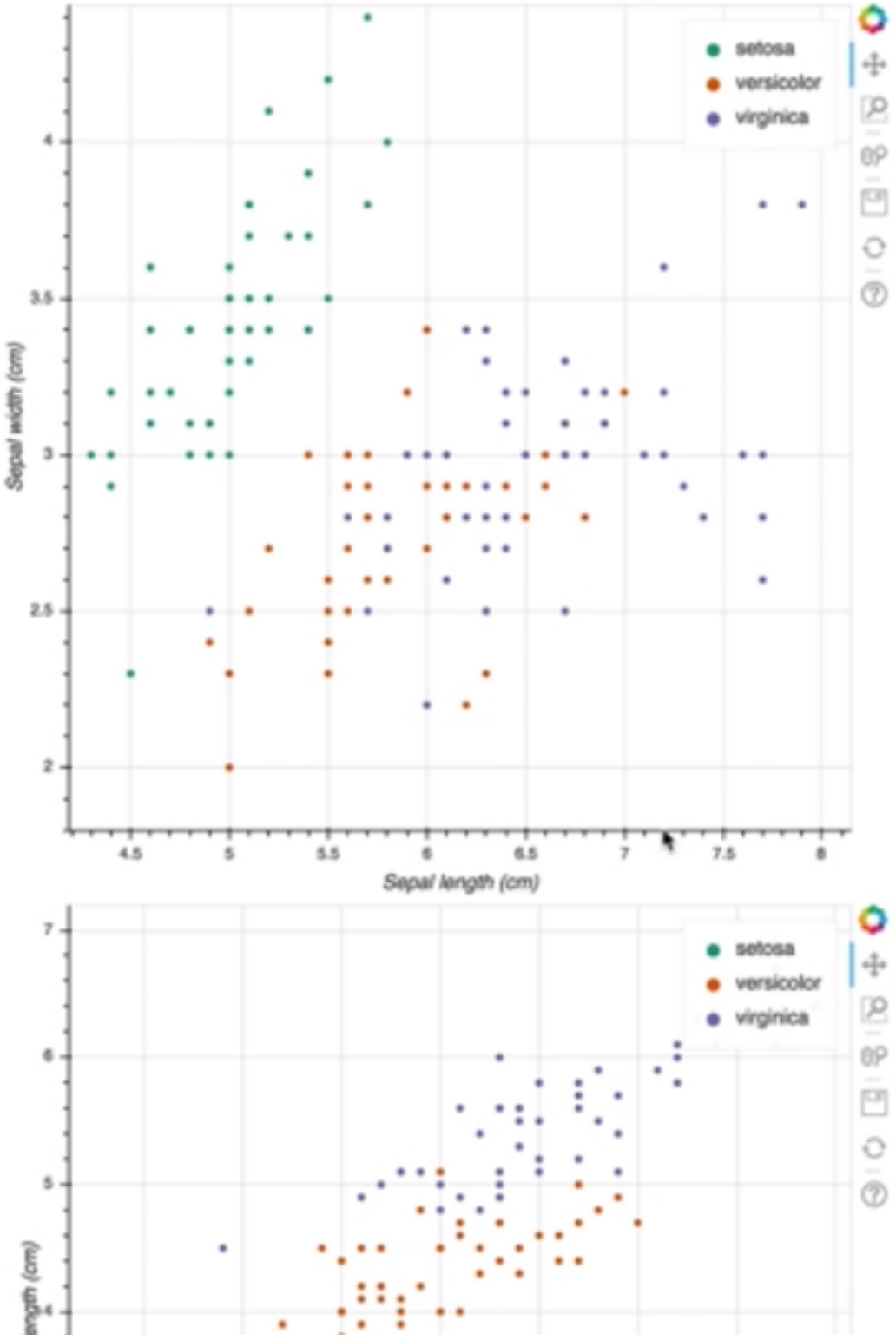


Let's run this code.



The x-axes of these plots are the same.

Let’s move the top plot. We notice that the plot below does not change.



I want to make it so that when I move the top plot along the x-axis, the plot below also to move i tandem. I want these figures to have a linked x-axis. We can easily achieve this in Bokeh, by adding a single argument to the *figure()* method.

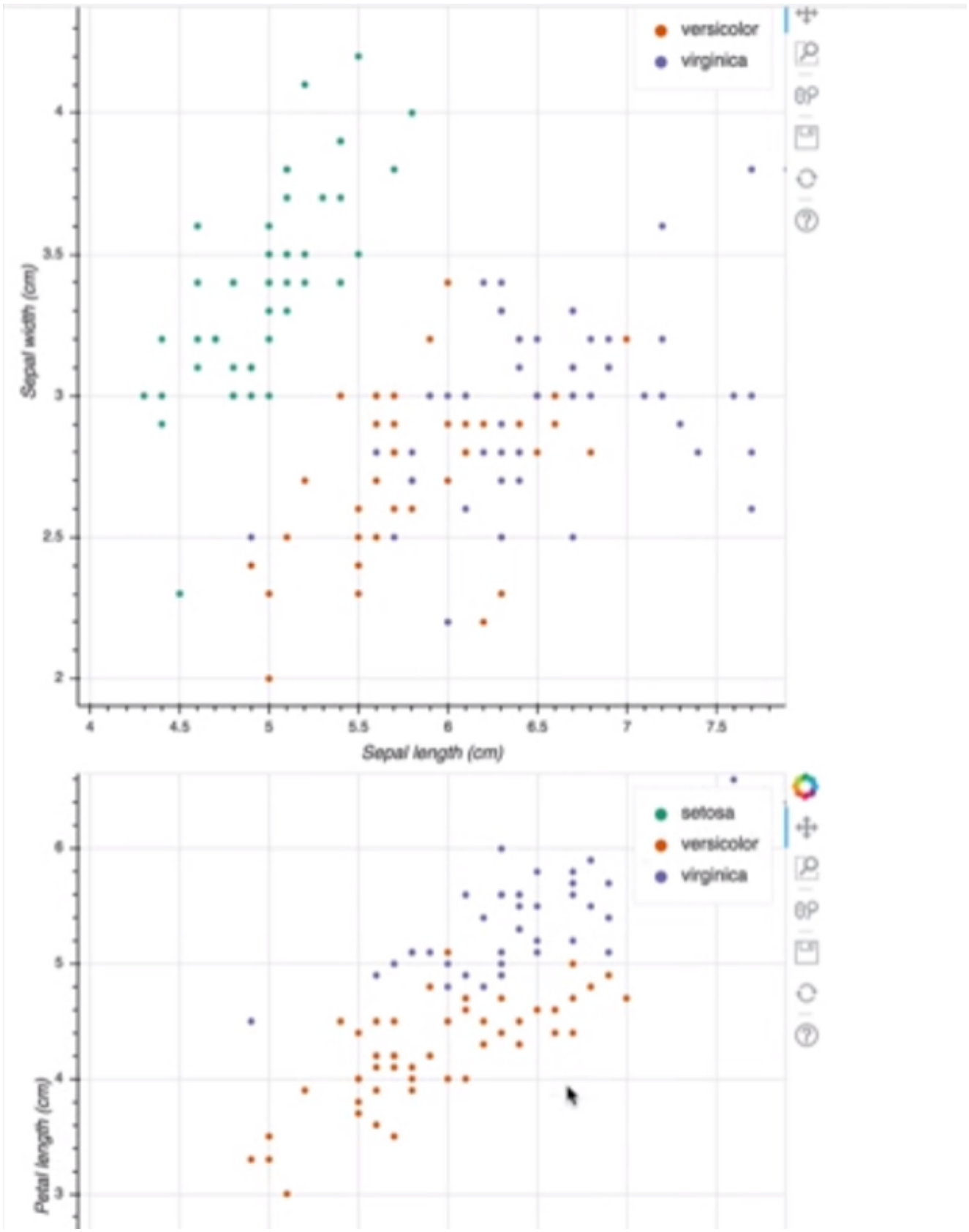
Let's modify this code above.

```
fig2 = figure(x_axis_label='Sepal length (cm)', y_axis_label='Petal length (cm)', x_r
```

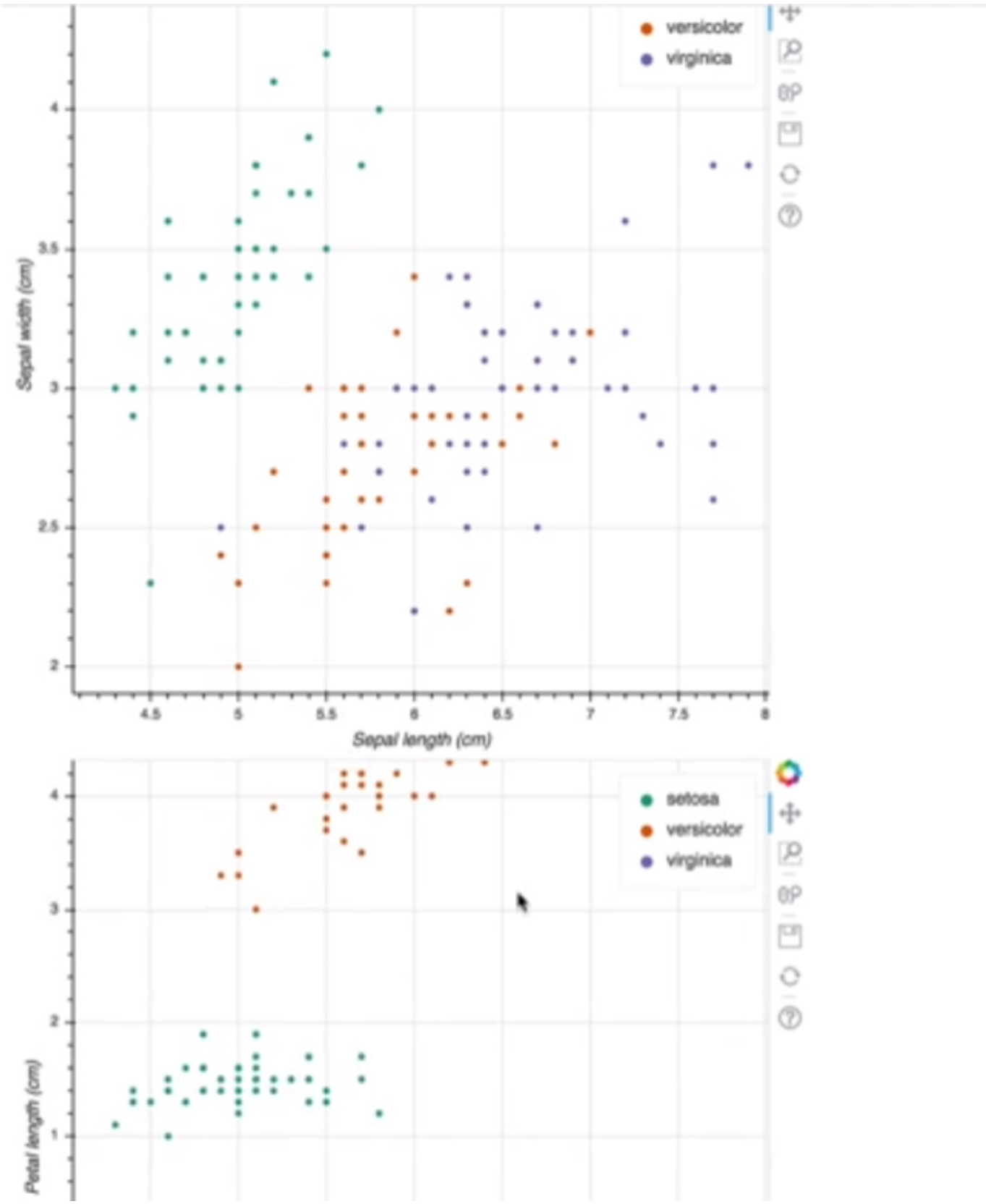


```
ange=fig1.x_range)
```

Let’s run this code. We get the same plot as above. However now when I move along the x-axis of one of the plots, the x-axis of the other plot moves as well.



However, when we move the y-axis in one figure, the other figure does not change.



This is because we haven't linked the y-axes. In the above figures, it may not make much sense to link the y-axes as they depict **different** things whereas the x-axes both show the sepal length. In general though, we can link any number of axes of any number of figures together.

Summary

All it takes to link axes of figures together is a single argument to the *figure()* method.