**Preparing your figures to share with others**

This chapter will focus on creating visualizations that you can share with others and incorporate into automated data analysis pipelines. We'll start with customization of figure styles. Previously, you saw that you can change the appearance of individual elements of the figure, such as the line color, or marker shapes.

**Changing plot style**

Here, we'll change the overall style of the figure. To see what that means, let's look at one of the figures we created in a previous lesson. This figure shows the average temperatures in Seattle and Austin as a function of the months of the year. This is what it looks like per default. 

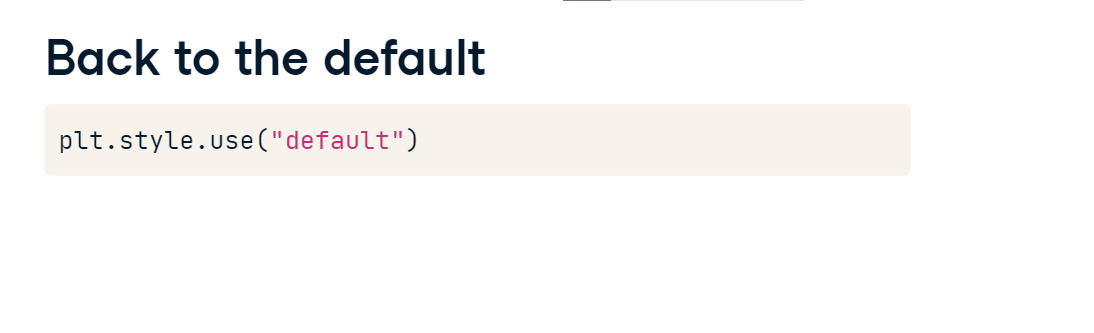
**Choosing a style**

If instead, we add this line of code before the plotting code, the figure style will look completely different. The style we chose here emulates the style of the R library ggplot. Maybe you know this library and this looks familiar to you, or you can learn about ggplot in a DataCamp course devoted to this library. Either way, you will notice that the setting of the style didn't change the appearance of just one element in the figure. Rather, it changed multiple elements: the colors are different, the fonts used in the text are different, and there is an added gray background that creates a faint white grid marking the x-axis and y-axis tick locations within the plot area. Furthermore, this style will now apply to all of the figures in this session, until you change it by choosing another style.



**Back to the default**

For example, to go back to the default style, you would run plt-dot-style-dot-use "default".



**The available styles**

Matplotlib contains implementations of several different styles and you can see the different styles available by going to this webpage, which contains a series of visualizations that have each been created using one of the available styles.



**The "bmh" style**

For example, this is what you get if you use "bmh" as the style.

**Seaborn styles**

This is what you get if you select "seaborn-colorblind". In fact, if you visit the documentation web-page, you will see that there are several available styles that are named after the Seaborn software library. This is a software library for statistical visualization that is based on Matplotlib, and Matplotlib adopted back several of the styles developed there. You can learn more about Seaborn in other DataCamp courses.



**Guidelines for choosing plotting style**

How would you choose which style to use? If your goal is primarily to communicate with others, think about how they might see it. Dark backgrounds are generally discouraged as they are less visible, so only use them if you have a good reason to do so. If colors are important, consider using a colorblind-friendly style, such as "seaborn-colorblind" or "tableau-colorblind10". These are designed to retain color differences even when viewed by colorblind individuals. That might sound like a minor consideration, but approximately 1 out of 20 individuals is colorblind. Figures that are designed for use on websites have different considerations than figures in printed reports. For example, if someone is going to print out your figures, you might want to use less ink. That is, avoid colored backgrounds, like the background that appears in the "ggplot" style that we demonstrated before. If the printer used is likely to be black-and-white, consider using the "grayscale" style. This will retain the differences you see on your screen when printed out in a black-and-white printer.

**Sharing your visualizations with others**

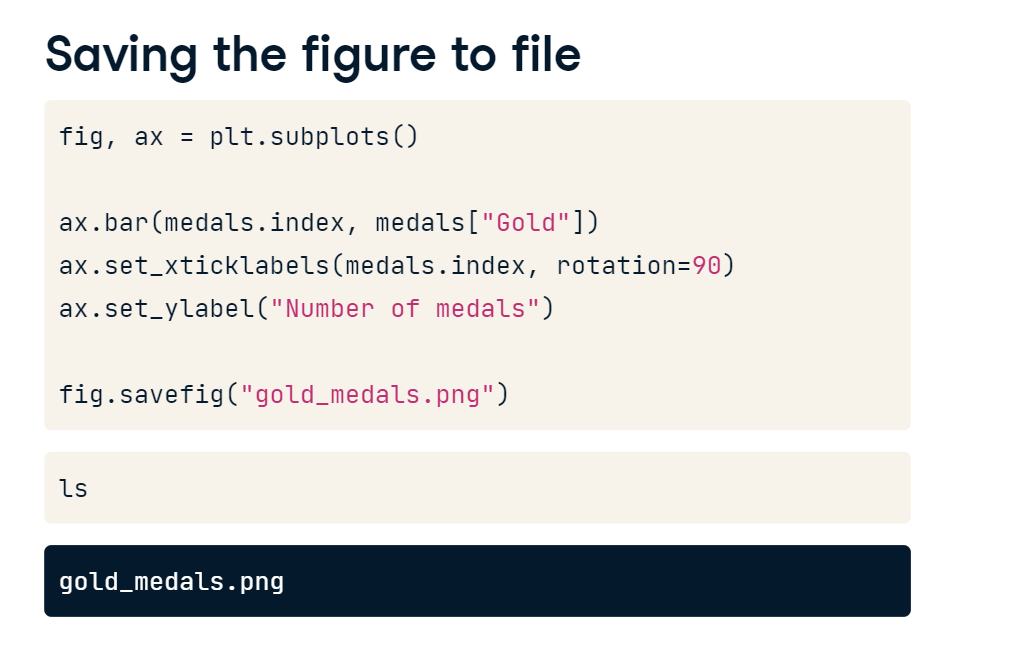
After you have created your visualizations, you are ready to share them with your collaborators, colleagues, and with others. Here, we will show how you would go about doing final customizations to your figures, and saving them in an appropriate format.

**A figure to share**

Take for example this figure that you previously created to display data about the number of gold medals that each of several countries won in the 2016 Olympic Games. When you previously ran this code, it displayed the figure on your screen when you called the plt-dot-show method at the end of this code.

**Saving the figure to file**

Now, we replace the call to plt-dot-show with a call to the Figure object's savefig method. We provide a file-name as input to the function. If we do this, the figure will no longer appear on our screen, but instead appear as a file on our file-system called "gold-underscore-medals-dot-png". In the interactive Python shell that we are using here, we can call the unix ls function, which gives us a listing of the files in the present working directory. In this case, only the file that we created is present. We can then share this file that now contains the visualization with others.



**Different file formats**

In the previous slide, we saved the figure as a PNG file. This file format provides lossless compression of your image. That means that the image will retain high quality, but will also take up relatively large amounts of diskspace or bandwidth. You can choose other file formats, depending on your need. For example, if the image is going to be part of a website, you might want to choose the jpg format used here, instead. This format uses lossy compression, and can be used to create figures that take up less diskspace and less bandwidth. You can control how small the resulting file will be, and the degree of loss of quality, by setting the quality key-word argument. This will be a number between 1 and 100, but you should avoid values above 95, because at that point the compression is no longer effective. Choosing the svg file-format will produce a vector graphics file where different elements can be edited in detail by advanced graphics software, such as Gimp or Adobe Illustrator. If you need to edit the figure after producing it, this might be a good choice.



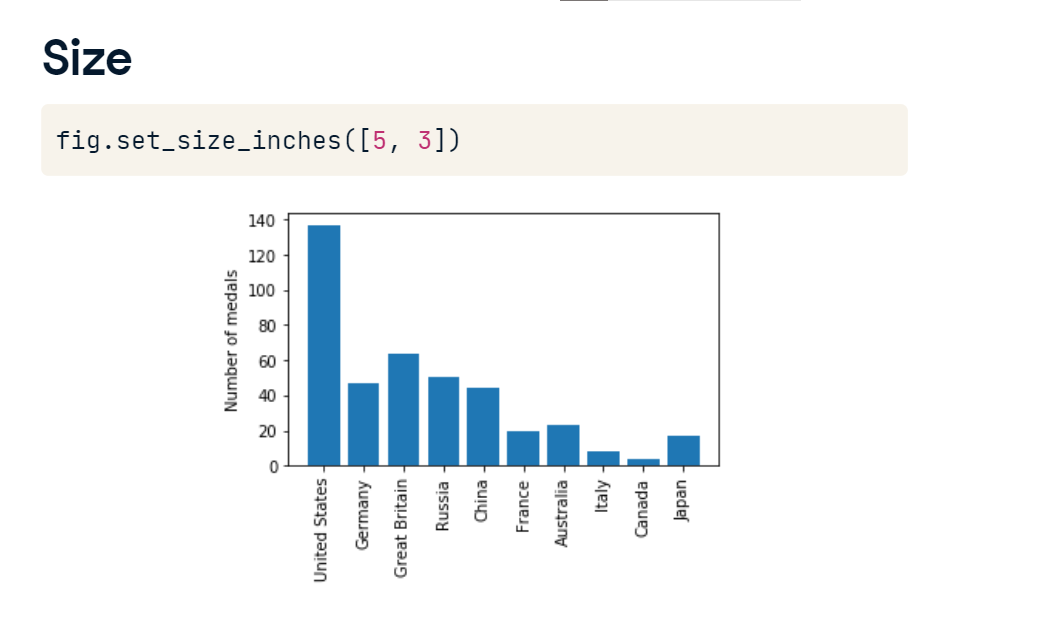
**Resolution**

Another key-word that you can use to control the quality of the images that you produce is the dpi key-word argument. This stands for dots per inch. The higher this number, the more densely the image will be rendered. If you set this number to 300, for example, this will render a fairly high-quality resolution of your image to file. Of course, the higher the resolution that you ask for, the larger the file-size will be.



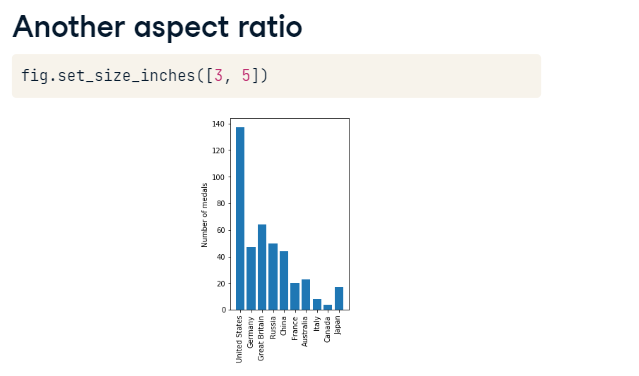
**Size**

Finally, another thing that you might want to control is the size of the figure. To control this, the Figure object also has a function called set-underscore-size-underscore-inches. This function takes a sequence of numbers. The first number sets the width of the figure on the page and the second number sets the height of the figure. So setting the size would also determine the aspect ratio of the figure. For example, you can set your figure to be wide and short



**Another aspect ratio**

or long and narrow, like here.



**Automating figures from data**

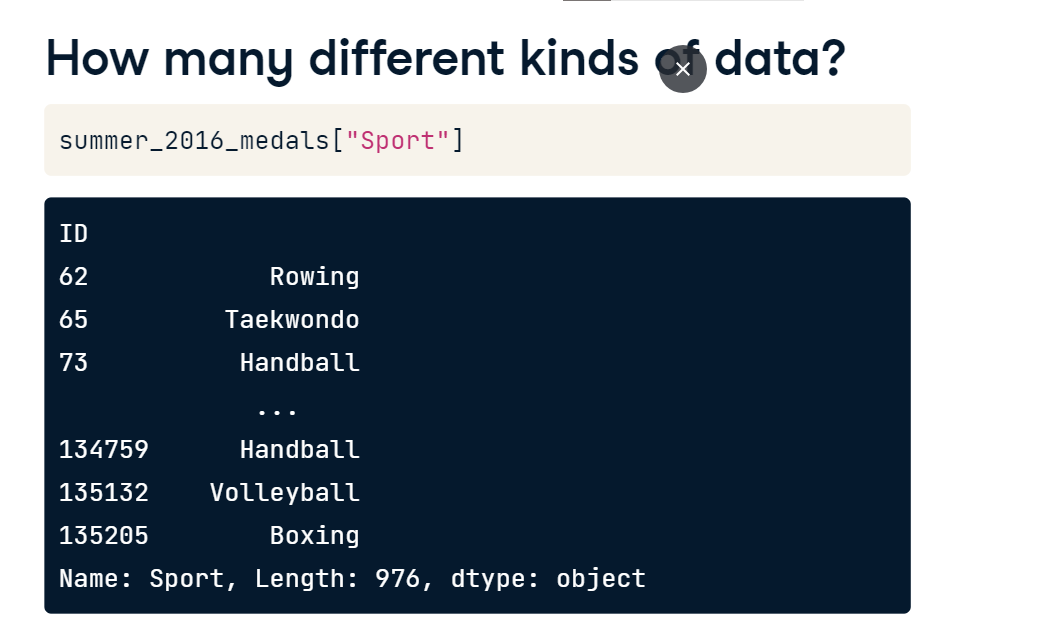
One of the strengths of Matplotlib is that, when programmed correctly, it can flexibly adapt to the inputs that are provided.

**Why automate?**

This means that you can write functions and programs that automatically adjust what they are doing based on the input data. Why would you want to automate figure creation based on the data? Automation makes it easier to do more. It also allows you to be faster. This is one of the major benefits of using a programming language like Python and software libraries such as Matplotlib, over tools that require you to interact with a graphical user interface every time you want to create a new figure. Inspecting the incoming data and changing the behavior of the program based on the data provides flexibility, as well as robustness. Finally, an automatic program that adjusts to the data provides reproducible behavior across different runs.

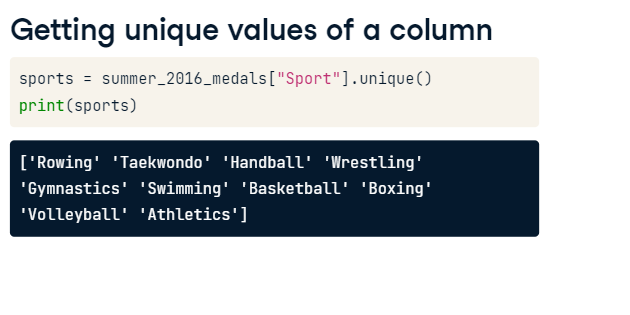
**How many different kinds of data?**

Let's see what that means for Matplotlib. Consider the data about Olympic medal winners that we've looked at before. Until now, we always looked at two different branches of sports and compared them to each other, but what if we get a new data file, and we don't know how many different sports branches are included in the data? For example, what if we had a data-frame with hundreds of rows and a "Sport" column that indicates which branch of sport each row belongs to.



**Getting unique values of a column**

A column in a pandas DataFrame is a pandas Series object, so we can get the list of different sports present in the data by calling the unique method of that column. This tells us that there are 10 different branches of sport here.



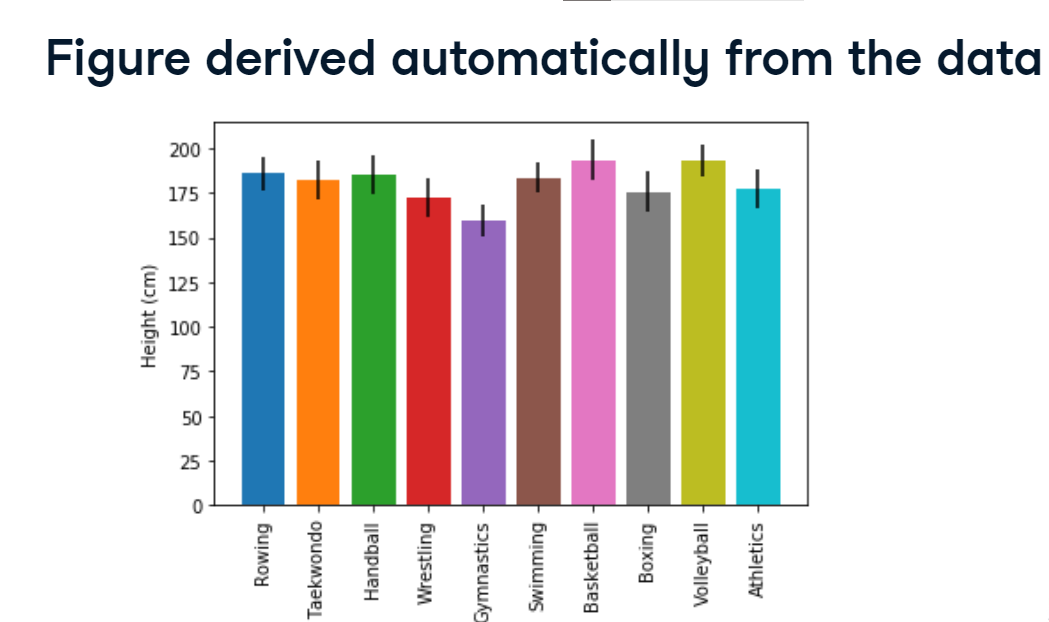
**Bar-chart of heights for all sports**

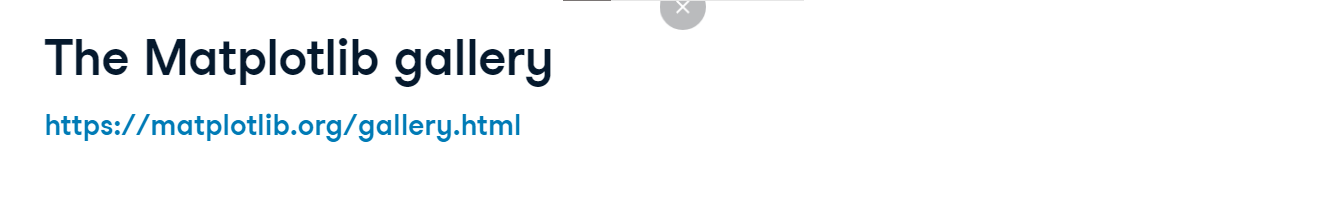
Let's say that we would like to visualize the height of athletes in each one of the sports, with a standard deviation error bar. Given that we don't know in advance how many sports there are in the DataFrame, once we've extracted the unique values, we can loop over them. In each iteration through, we set a loop variable called sport to be equal to one of these unique values. We then create a smaller DataFrame, that we call sport-underscore-d-f, by selecting the rows in which the "Sport" column is equal to the sport selected in this iteration. We can call the bar method of the Axes we created for this plot. As before, it is called with the string that holds the name of the sport as the first argument, the mean method of the "Height" column is set to be the height of the bar and an error bar is set to be equal to the standard deviation of the values in the column. After iterating over all of the sports, we exit the loop. We can then set the y-label to indicate the meaning of the height of each bar and we can set the x-axis tick labels to be equal to the names of the sports. As we did with the country names in the stacked bar chart that you saw in a previous lesson, we rotate these labels 90 degrees, so that they don't run over each other.

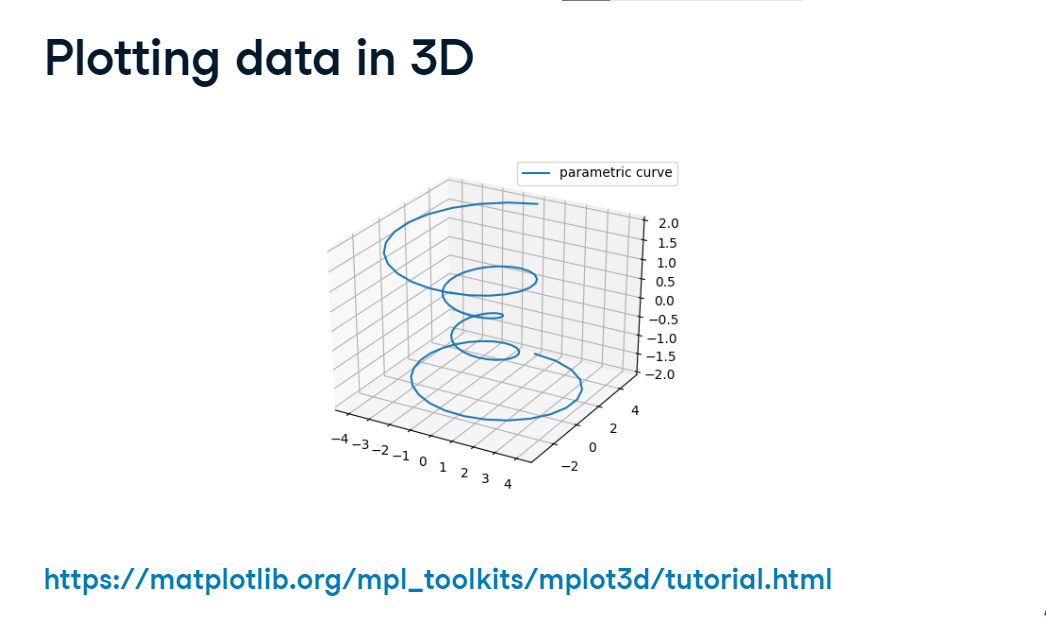


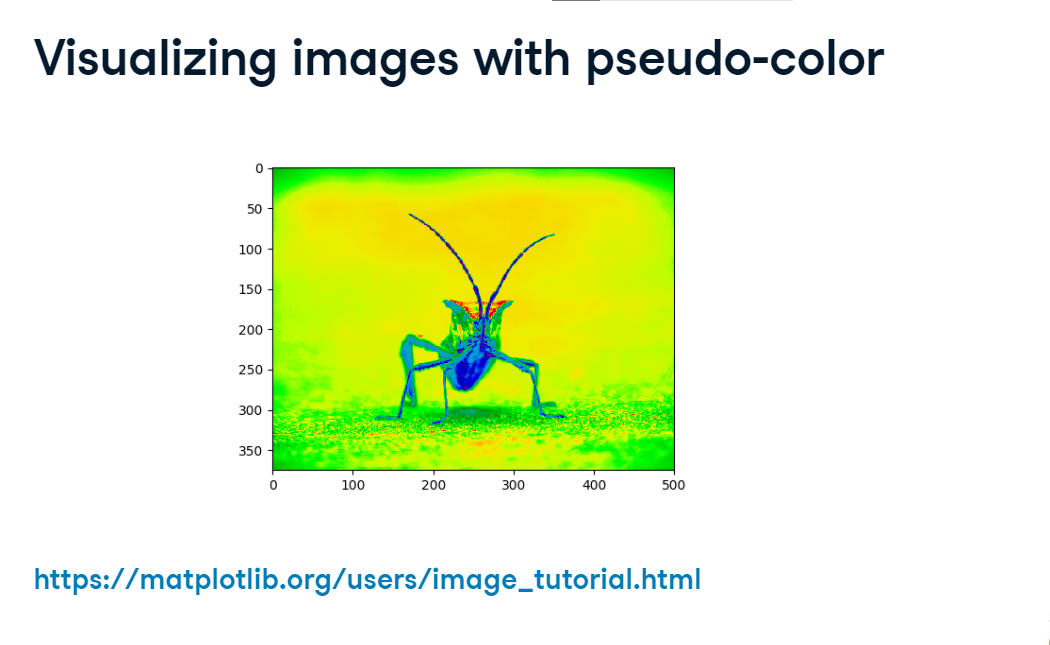
**Figure derived automatically from the data**

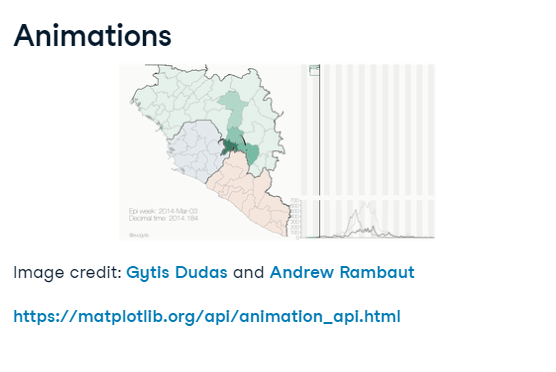
This is what this figure would look like. Importantly, at no point during the creation of this figure did we need to know how many different sports are recorded in the DataFrame. Our code would automatically add bars or reduce the number of bars, depending on the input data.

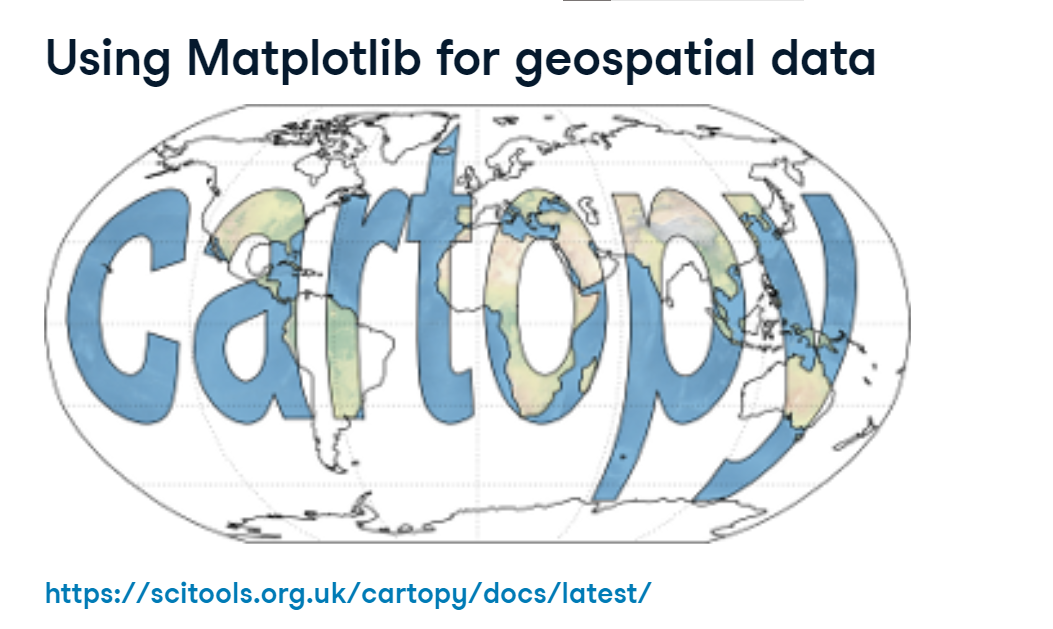


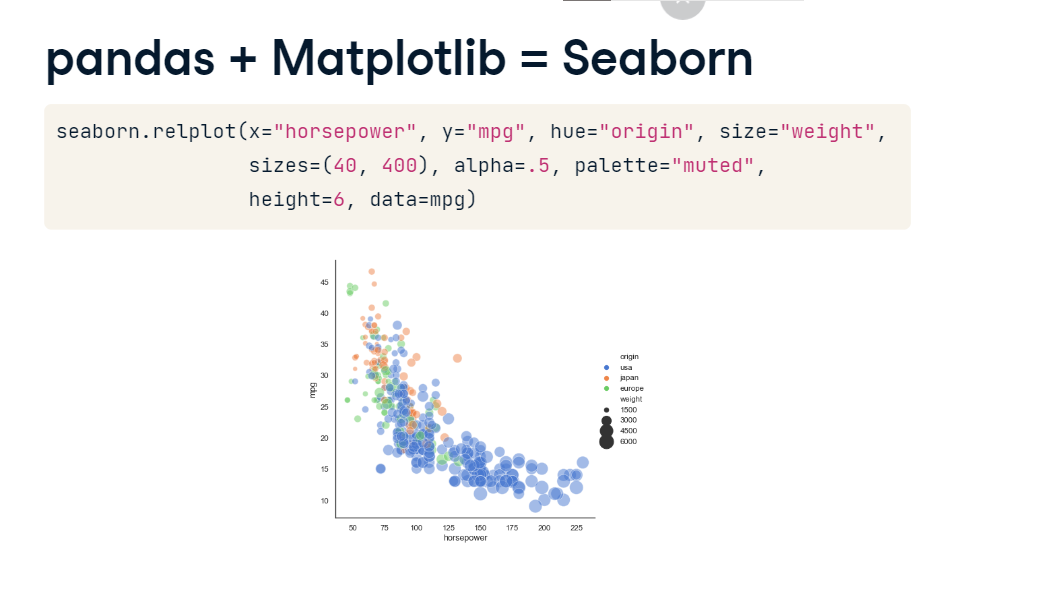












**Where to go next**

Congratulations! You have completed this introduction to Matplotlib. And yet, we have only scratched the surface in what Matplotlib can do.

**The Matplotlib gallery**

One way to learn about other kinds of visualizations that you can create with Matplotlib is to visit the online gallery of examples on the Matplotlib website, at this URL.

The gallery contains several dozen examples of figures that you can create with Matplotlib. If you click on one of the figures, you will land in a page

**Example page with code**

that contains not only a larger version of the example, but also the full Python code that would generate this example from scratch. If you are interested in creating a visualization that is a variation on this example, you can start by copying over this example code and editing it to fit your particular use-case. This is much better than starting from scratch!

**Plotting data in 3D**

Here are a few of the things that you might want to do next. In this course we always visualized data using the two dimensions of the page, but you can also extend your capability to visualize data, by adding perspective to your visualizations to make them appear three-dimensional. For example, here is a parametric curve through a three-dimensional space. In this web page, you can learn more about creating three-dimensional visualizations.

**Visualizing images with pseudo-color**

Another capability of Matplotlib is visualizing data from images. For example, here is an image visualized using pseudo-color, where each value in the image is translated into a color. You can learn more about working with images in this URL.

**Animations**

You might remember this visualization that I showed you in the very first lesson of this course. It used one more dimension, time, by varying the display through animation. You can create animations by creating multiple frames of the movie, each as its own visualization, and then stitching them together into a movie using tools such as Quicktime, but Matplotlib also has its own interface for creating animations. You can learn about this interface at this URL.

**Using Matplotlib for geospatial data**

There are multiple software packages that extend Matplotlib's capability to a variety of different kinds of data. For example, Cartopy extends Matplotlib to be used with geospatial data, such as maps.

**pandas + Matplotlib = Seaborn**

03:10 - 03:52

Another library that extends Matplotlib is Seaborn. This library creates very sophisticated statistical visualizations from pandas data structures, such as DataFrames. The nice thing about Seaborn is that you can create elegant and sophisticated visualizations of your data with very little code. For example, this code would create this visualization that encodes the fuel efficiency of cars as a function of their horsepower, but also encodes the country in which the car was manufactured, using the color of the bubbles, as well as their weight, using the size of each bubble.

**Seaborn example gallery**

Seaborn also has an extensive example gallery that you can visit in this URL.

**Good luck visualizing your data!**

So as you can see, there is a lot more to learn about data visualization. But thanks to this course, you have already taken your first step along the path to visualizing your data in Python, using Matplotlib. Good luck visualizing your data!