# Building Data Visualizations Using Matplotlib

by Janani Ravi

Start Course

Bookmark Add to Channel Download Course



# Course Overview

## Course Overview

(Music playing) Hi, my name is Janani Ravi and welcome to this course on Building Data Visualizations Using Matplotlib. A little about myself, I have a master's in Electrical Engineering from Stanford, and have worked at companies such as Microsoft, Google, and Flipkart. At Google I was one of the first engineers working on real-time collaborative editing in Google Docs, and I hold four patents for its underlying technologies. I currently work on my own startup, Loonycorn, a studio for high quality video content. In this course, we'll understand the basic components which make up a plot, and see how we can tweak parameters and attributes to have the visualizations customized to exactly how we want them to be. We start off by understanding the basic APIs available in Matplotlib and where they are used. We'll build basic plots and learn how to customize the display colors and other attributes of these plots. We'll work with plots which have multiple axes and also with watermarks. We'll then move onto building intermediate and advanced plots, drawing shapes and Bezier curves, using text and other annotations to highlight plot elements, and normalizing this case that we use on the X as well as the Y axis. We'll then use some real world data to visualize statistical data, such as mean, median, mode, and outliers in our data. We'll cover box plots, violin plots, histograms, pie charts, stem and stat plots, and autocorrelation graphs. This course will allow you to explore all of the nitty-gritty that Matplotlib

has to offer, so you can build production-ready visuals who embed with your UI or to display within reports and presentations.

# Working with the Matplotlib and Pyplot APIs

## Module Overview

Hi, and welcome to this course on Building Data Visualizations Using Matplotlib. If you're a data analyst or a data scientist and you worked with Python for any length of time, chances are you've used Matplotlib for your visualizations. Matplotlib is one of the most important packages for data visualizations, and an important reason for that is its seamless integration with other Python packages in the PyData stack. NumPy, Pandas, and even the SciKit learn libraries integrate smoothly with Matplotlib. It's possible to produce very cool-looking graphs and charts with Matplotlib without really understanding what exactly is going on, and that's what this course looks to address. In this very first module, we'll start off with the basic anatomy of a figure in Matplotlib, what it's made up of, and how you can exercise granular control over each part of this figure. This course is very, very hands-on, and you'll learn all of the features of Matplotlib with real code, basic plots, labels, titles, markers, and watermarks. This is what we'll cover in this module. We'll also see how we can build and customize multiple plots on the same figure using the axes and subplot components of Matplotlib.

## Course Outline and Prerequisites

Before we dive into the actual contents of this course, let's see some of the prereqs that you need to have in order to make the most of your learning. This course assumes that you have a basic working knowledge of the Python programming language. All demos in this course will be in Python 3. This course also assumes that you're comfortable working with the Jupyter iPython notebook for Python. Jupyter notebooks are an interactive shell available on the browser that allows you to execute code and view results right there onscreen in front of you. A basic understanding of NumPy arrays will also be helpful for this course because all of our data will be expressed in the form of these NumPy arrays. In case you want to brush up on your Python programming or want to have a deeper understanding of NumPy, here are some courses on

Pluralsight that I recommend. Python: Getting Started and Python Fundamentals are great beginner courses on Python. Working with Multidimensional Data Using NumPy will give you a good overview of the NumPy library in Python. This course starts with first principals. We'll understand where Matplotlib fits in the PyData ecosystem and the basic anatomy of a figure in Matplotlib. We'll move onto basic plotting techniques, see how we can plot titles, labels, lines, markers, and watermarks. Once we have a sound understanding of the basics, we can move onto actual plots. We'll see how we can plot shapes and curves, use text and annotations in order to enhance our visualizations, we'll see how we can use different units on the same axis, and we'll see how we can scale the representation of a particular axis to get the right information we want displayed onscreen. We'll then move onto using the built-in plots and graphs that Matplotlib has to offer. We'll study the boxplot and the ylim plot and understand why we would choose to use one over the other. We'll see how we can build histograms and pie charts, stem charts, which will allow us to see data from the previous record. We'll also use Matplotlib to visualize autocorrelations and explore stackplots.

## Introduction to Matplotlib

Businesses and organizations collect a huge amount of data nowadays. There are sensors all over the world collecting climate data. There are sensors on a car which predict how the car is going to move. There is just data that organizations collect from the clicks on their website. All of this data holds some key insights and visualizations are the easiest way to absorb insights that you might extract from this data. As a data analyst or a data scientist, when you're presenting information to others on your team or in your organization, you want to actually set this up in the form of a graph or a chart so that it's memorable. Visualizations are also the easiest way to remember complicated information. Visualizations are not just useful to showcase the end results of your analysis, visualizations in fact form an important step in data exploration. When you get a huge dataset, in order to figure out what kind of insights you can extract from it, what kind of relationships exist between the variables, you need visualization. This is an important reason why the PyData stack contains so many libraries such as NumPy, Pandas, Seaborn, and Matplotlib that allows you to help develop an intuition for relationships in data. You can play around and slice and dice your data in different ways to see what's in there. In the real world, data visualization is an important precursor to higher level data analysis, especially using machine learning or deep learning techniques. Machine learning or ML algorithms are algorithms that learn from the data that is fed into them. How do I know what features affect my output? I'd want to visualize it first before I actually feed into my ML algorithms blindly. And it is this key need for visualization that

Matplotlib tries to fulfull. Matplotlib is a Python 2D or 2-dimensional plotting library which produces publication quality figures. The figures are of a very high quality, they can be printed out or embedded within other applications in order to convey information. A powerful feature of Matplotlib is the fact that it works across a variety of hardcopy formats and it also embeds into a number of different platforms and backends. Matplotlib is an important and integral part of the PyData stack. The PyData stack is a variety of Python packages that is used for data analysis and data science. There are a number of different packages which exist within this, NumPy, Pandas, statsmodels, SciKit-learn and SciKit-image; all of these are part of the PyData stack, and Matplotlib integrates seamlessly with them. Any data expressed in the form of a NumPy array can be passed in directly into a Matplotlib visualization. You can select those columns which we are interested in within a Pandas dataframe and use them with Matplotlib to visualize the information within. Matplotlib itself is not a monolithic package; it's made up of several modules, each of which perform a different logical group of functionality. Matplotlib, pyplot, pylab, and the object-level APIs call Matplotlib APIs. Object-level APIs form the code building blocks of our visualization. These allow us to exercise very fine-grained control over how the data is displayed. You can control individual axes that fix on each axes, grid lines, everything. The pyplot module is a higher-level extraction which works on data directly. These are the APIs that we'll use the most often. We'll use pyplot to set up the data that we want to visualize and then use the lower-level APIs to customize our graph. Another module which forms an important component of Matplotlib is pylab. It's a convenience module which imports portions of the Matplotlib library and NumPy to give users a Matlab-like access to functionality available in Matplotlib. And here is where we come to a little bit of history about Matplotlib. Matplotlib has the term Matlab within it because it was originally conceived as a way to allow Python users access to Matlab-like graphs and visualizations. If you've worked with Matlab before, Matplotlib should be very simple and straightforward for you, and you probably already know that Matlab excels at making nice-looking plots and build them easily. There are a couple of drawbacks to using Matlab, though. Matlab is not free, which means you have to pay for its license. It's difficult to scale to a huge dataset. And finally, as a programming language, Matlab is not the language of choice for data scientists and data analysts; it's rather difficult to use. And this is where there is the need for something like Matplotlib. It emulates Matlab's plotting capabilities and brings all of these capabilities to Python developers. The best things about Matlab plotting is now available in Python. This Python plotting library is very popular because of how easy it is to create plots in it. It's also embeddable in that each of your visualizations can be part of a GUI application that you're building in Python. It can also be used across platforms. We'll use it with Jupyter notebook.

It can be part of a regular Python script, it can be within a Python GUI application, or it can be printed out in a publication or hard copy format.

## Anatomy of a Figure

Matplotlib is made up of three top-level components. The first of these is Pylab. This module allows users to work with Matplotlib as thought they were working with Matlab. This module is especially useful for developers and data scientists who are moving over from Matlab into the world of Python. For this course, we assume that you are already Python developers, which is why we won't focus at all on the Pylab module. When you're working with Matplotlib, you'll often hear the term front-end. The term front-end collectively refers to those classes in Matplotlib that abstract the actual plotting mechanics from the developer. These are the high-level APIs that make plotting easy. The front-end does not change when you change the actual implementation of how the graphing works. The front-end is a high-level abstraction that makes plotting easy. Because Matplotlib works across so many different platforms, the front-end code and high-level APIs have to be translated to some kind of back-end implementation. Back-end refers to those renderers that transform the front-end representation to a hardcopy or display device. Depending on where you're using Matplotlib, you can choose a particular back-end to work with your front-end. That's, of course, if you have the rest of the drivers, et cetera, for the back-end pre-installed on your machine. Here is a simple tree structure that represents Matplotlib object hierarchy. At the very root of this tree is the figure class. The figure can be thought of as a top-level container class which contains within it the access and the subplots on which you do the actual plotting. You can use the PyPlot APIs within a figure. These operates at higher levels and provide abstractions for you to draw complex virtualizations. You can also use the lower level Matplotlib APIs within a figure to exercise fine-grained control over how your plot actually displays. Let's take a look at the visual anatomy of a Matplotlib figure so that we know the right terms to use for each component. Remember we can exercise control over each of these components in Matplotlib. At the top level, this is the figure which is the overall window or page in which the actual plotting happens. All of the operations that you perform on Matplotlib will be within this figure. Contained within this top-level figure container are the axes. Axes are where the actual plotting or graphing operations happen. Anything that you see displayed, a pie chart or a bar chart, is within axes. Matplotlib is a 2D plotting library, which means every axes is made up of 2 components, the X axis and the Y axis. These are the axes that you're familiar with, used to represent your data points. And for each of your axes, whether it's X axis or the Y axis, they contain ticks, the location of those ticks, and labels associated with the axes. An essential feature

of Matplotlib is the fact that a single figure can contain more than one axes. This is what allows us to lay out multiple plots in the same figure. Here is the basic anatomy of a Matplotlib figure. Here are terms that you should be familiar with. A figure can be subdivided into grids, and grids are what allow us to see where a particular data point or a line is located at. A figure in Matplotlib is made up of one or more axes, and every axis is made up of an X axis and a Y axis. An axis might contain major ticks, which are these large dashes, which align with the grid of the figure, and help you parse where exactly a data point is plotted on your axes. In addition to major ticks, the smaller horizontal lines are minor ticks. These represent intermediate range values on either the X axis or the Y axis. A visualization that you have on a Matplotlib figure could be a line plot. A line plot is simply a sequence of data points that have been connected using a line or curve of some kind. All your visualization could comprise of a number of data points that are significant in some way. These points are represented by markers. Every axes within a figure has a title associated with it, which explains what exactly that visualization is supposed to convey, what is the information that it represents. If you have multiple lines or markers within your graph, you'll also need some kind of legend which explains what each line, what each color, what each marker stands for. All of these components form the basic anatomy of a figure in Matplotlib, and each of these components can be individually customized.

## Non-interactive Mode

Let's get hands-on right away. We'll start off with our first demo where we'll understand how Matplotlib works in the non-interactive mode. Matplotlib has two modes of functioning, the interactive mode and the non-interactive mode. And if you're using Jupyter notebooks, chances are that you'll work in the interactive mode of operation. However, it's important to understand how the non-interactive mode works, and that's what we'll cover here. This is the directory where I'm going to be writing all of my code. This is my current working directory. All of my code will be written in Python 3, so I'm going to start off a new Python 3 notebook. I'm going to give this a descriptive name so that I know what this demo stands for. This is the non-interactive mode demo. If you're using Jupyter notebooks within your Anaconda distribution, you don't need to do anything extra to set up the Matplotlib package on your local machine. However, if you do not have Matplotlib installed, you must run a pip install, either on your terminal window or within your Jupyter notebook. If you're running pip install Matplotlib within Jupyter notebook, don't forget the exclamation point just before the pip word. I import the Matplotlib library and the version that I have installed, which is the latest at the time of this recording is 2. 1. 0. A former check on your machine to see what mode Matplotlib is currently running in. If Matplotlib is an interactive mode,

is_interactive will be, too. The default mode for your Matplotlib depends on how you install the Matplotlib libraries. If you use the installation that came along with the Anaconda distribution, chances are you'll start off in non-interactive mode. If you use pip install, chances are you'll start off in the interactive mode. You can see that my interactive mode is currently set to false. Now, what I found by trying on different machines was that this is actually a little buggy. Interactive mode might be set to false, but you're actually in interactive mode, and if you especially got the Matplotlib libraries using a pip install, you will stay in interactive mode and you can't turn it off, so beware of this. You can set the interactive mode explicitly using this function, matplotlib. interactive, and pass in the False as an input argument, or you can use the matplotlibr. pyplot library to turn off interactive mode by calling plt. ioff. At this point, I can reconfirm that I am in non-interactive mode. Now for your setup, if you find that you can't turn interactive mode off, do not worry, all the demos in this course are done using interactive mode, but you should understand how non-interactive mode works, which is why we've especially set up this demo for you. Let's understand the differences between non-interactive and interactive mode by plotting a simple line. We just pass in two lists. The first list contains X coordinates, the next list contains the corresponding Y coordinates. When we call the plot function in non-interactive mode, the line object representing our visualization is created, but there is nothing plotted to screen. This line connects our data points together, but you can't actually see the visual. In non-interactive mode, you have to explicitly call the plot. show function in order to display your graph onscreen. The graph is constructed behind the scenes to display it called plt. show. Notice that our first Matplotlib graph has plotted the individual data points, which make up our X and Y coordinates. All of these points are plotted and Matplotlib has drawn a line connecting all of these points. In this particular notebook, I'm going to now turn on interactive mode by calling matplotlib. interactive and passing in True. You can also enable interactive mode by calling the plt. ion function. Now, it's not recommended within the same Jupyter notebook to turn interactive modes on and off when you're working in production. This causes some confusion as to how exactly a plot will behave, which is why it's not recommended. It's just something for you to be aware of. Now that we are in interactive mode, let's call the same plot function with the same dataset as before to plot our line. And notice here that the graph is displayed onscreen right away in interactive mode. As soon as the plot is drawn and as soon as you executed that cell, you see the visual, there is no reason to call plt. show explicitly. If you're working in production when you're using Matplotlib in a GUI application or within a script, it's advisable to use the non-interactive mode because the non-interactive mode can optimize the plotting of your visuals. In interactive mode, whatever you've plotted you see it right away onscreen. But for this particular course, the interactive mode makes much more sense, which is why we'll use that.

## Interactive Backends

In this demo, we'll see how we can configure Matplotlib to work with an interactive back-end. We've spoken of Matplotlib backends earlier. Backends are essentially renderers that display our visualization to either a device or a hardcopy format. Here we'll see how we can configure a Matplotlib backend. You can use the %matplotlib --list command to list all the backends that are available. The list that you see on your screen may not be exactly the same as the list that I have because it varies depending on your environment, and many of these backends can't be used out of the box. There might be additional packages or drivers that you need to install before you can use them. Check to see whether you're in interactive or non-interactive mode for your Matplotlib. I'm currently in non-interactive mode. I'm going to set it to be interactive mode before I change my backend. I'll set matplotlib. interactive to True. If you're already in interactive mode, you don't need to do anything. Let's check what backend we are currently using here by calling matplotlib. get_backend, and you can see it's some kind of inline backend that comes along with the Jupyter notebook. I've installed the Anaconda distribution of the Jupyter notebook on my Mac machine, and this is my default backend. Yours might be a little different, and it doesn't matter. Of the backends that work with Matplotlib, certain backends are set to be interactive backends and others are non-interactive. Interactive backends allow you to update your plots and visuals on the fly. These are typically used with your display device such as a computer. Hardcopy backends are non-interactive backends. They render your file out to an image or to some other kind of disk format. Nbagg is an interactive backend that's available on my machine. I'm going to set my Matplotlib to use this backend. All of the demos from here are in will use Matplotlib in interactive mode, import the PyPlot library, and alias it as plt. You can turn on interactive mode by calling plt. ion and confirm that you're indeed running in interactive mode by calling is_interactive. Let's plot a simple graph. This is the same graph that we've used before. We are currently using an interactive backend in interactive mode, and notice in the resulting display, you see a bunch of levers or buttons onscreen that will allow us to change the graph on the fly. If you click on this on/off button at the top right, this will turn off the interactive nature of this plot. This functionality that you see with all the buttons onscreen is only available on an interactive backend, such as nbagg. Let's use the zoom in button to zoom into our plot. After you click on this button, you can draw a rectangle onscreen and then zoom into this rectangle to zoom into this particular view on your visuals. Clicking on the back button here will take you back to your previous view. There are a number of things here that you can play around with. This is the interactive mode of Matplotlib with an interactive backend. Let's say we update the title of this plot by calling the title function on our PyPlot library. Once you execute this code, you see nothing below this cell. That's because

we've updated the currently active plot. When you use an interactive backend, unless you explicitly instantiate a new figure, all the changes that you make are to the currently active plot. When you're exploring data, when you're playing around to find out what relationships exist in your data, interactive backends are extremely useful; you want to view what changes you make in real time.

## Basic Plots

We are now ready to tackle basic plotting functionality in Matplotlib. Let's set up the imports first. The imports that we use across our demos will be pretty straightforward. We'll always use the Matplotlib library that offers us low-level control over our plotting. We'll use the PyPlot higher level abstraction to build plots. We'll use NumPy to deal with data. I won't repeat this again for any other demo; we'll always be working with Matplotlib in interactive mode. So the plt. show that you see within each execution cell is not really needed if you're in interactive mode. Let's plot the simplest of all graphs, and we'll start from there. We pass in the X and Y coordinate for our data points in the form of a list, a list of X coordinates and a list of Y coordinates. And here is the resulting visualization. Matplotlib has plotted these four points onscreen and connected them using a line. This is the default representation. By default, Matplotlib draws the line in blue color. Now if you want to change the color of this line, you can specify the same plot by passing in the color input argument here. We've specified the color in the form of a string. Color is equal to red, and the line displayed is red. If you want to give the viewer more information about what this plot represents, you need to specify labels for your X and Y axes so that they know what data is being plotted. This you can do by calling the xlabel and ylabel functions on your PyPlot module. Our labels are super simple here. The xlabel simply says x. We want it to be displayed with a font size of 15 and we want the label to be green in color. All of these customizations are input arguments to the xlabel function. Similarly, we specify the same customizations as input argument to the ylabel function. The ylabel simply says 2 multiplied by x, and we give it a font size and the color in which we want it displayed. Notice that by default, Matplotlib adjusts the ylabel to align with the axis. You can call the plt function on the PyPlot module by passing in just one list of coordinates. If only one list is passed in, it's assumed that those coordinates are Y coordinates, the corresponding X coordinates are simply the index values of these Y elements. In the resulting visualization, you can see that a Y value of 4 corresponds to an X value of 0, a Y value of 8 corresponds to an X value of 1, 12 is 2, and 16 is 3. Let's generate some random X values here in order to set up some more plots. We do this by calling the np. linspace function. The linspace function in NumPy is used to generate evenly spaced values between a start value, which is 0

here. The upper range is indicated by the stop input argument, which here is n, and the number of data points that we generate is 50, specified by the num input argument. We'll use these randomly generated X values to plot a sine in Matplotlib. The NumPy np. sin x will generate the sine value for each corresponding X value. So our xlabel will be the x element and the ylabel will be the corresponding sine values as you can see in the resulting visualizations. Make sure you label your x and y values accordingly so that the user knows what representation they are seeing, what visualization they are looking at. Matplotlib allows you to exercise very fine-grained control over your figure. Here we'll configure the tick parameters for our Y axis. The first input argument says which axis this tick_params apply to, the axis is the y axis, and we want the ticks to be red in color. They're really small here, but if you look closely, they'll be red in color. The labels for the ticks are also configurable. You can specify the color in which you want the labels to display, and the font size. These are extra, extra large tick labels. We can use the same tick_params functions to configure the X axis. This time we specified a different set of parameters. We basically said that bottom should be False and labelbottom should be False, which means no ticks on the x axis and no corresponding labels, and that's what you see here. Let's start off with the same sign curve that we have seen before. We have the labels for the corresponding X and Y axes. In addition, we want to specify a label for the curve as a whole, which we'll do by passing an unlabeled parameter to our plot function. If you now specify a legend for your plot, which you can do by calling the legend function on pyplot, the legend will automatically pick up the label that's associated with your curve. Notice that the sine curve here is represented by a blue line, exactly what we'd expect. In addition to a legend, we also want to set up a title for this plot explaining what relationship this represents. This is easily done by calling the title function on pyplot and passing in the text that we want displayed as the title. Let's say you have a really large plot which covers a wide range of values on the X and Y axes, but only a portion of this plot is really important and you want to highlight or zoom into that portion. You can do this by specifying the xlim parameters for the limits on your X axis. We plotted the same sine curve that we saw earlier, but we limited the X axis to display only those points which are between 1 and 5. We've zoomed into that portion of the X axis. Just like the X limits, to limit the range of X values, we can also specify Y limits to limit the range of Y values as well. Here our Y limits are from -1 to 0. 5, and only that portion of our plot is displayed onscreen. You can use the xlim and ylim functions on pyplot in order to generate mirror images of your plot as well. By switching the xlim parameters, we invert the plot along a vertical axis. Here our X limits are 5, 1, which is different from the earlier 1, 5, and you can see the resulting plot has been flipped along a vertical line drawn through x=3. This is the center of our range. You can compare this figure and the previous one, and you'll notice by simply changing the X limits, we created a mirror image of the original figure. We can flip our

figure along a horizontal line as well, this time by interchanging the Y limits. The Y limits are now 0. 5, -1, which is a reverse of the Y limits that we saw previously. You can see the new Y limits along the Y axis, and you can see that the plot has been inverted along the horizontal line.

## Lines and Markers

So far we've only seen a visualization which draws a line on our Matplotlib graph. Let's see how we can work with lines, as well as markers. We start off with the same sine curve that we worked with earlier. We'll generate some random data for the X axis between the 0 and 10, 50 points, and then we'll plot X and the sine value of X. After plotting our sine curve, we call the plt. show function. This is not needed if you're running in interactive mode. I've left it just in there in case we have people working on non-interactive mode. And here is our single sine we've plotted with the default Matplotlib colors. Now this same plot can be used for multiple visualizations. You can add multiple curves by specifying them both. We invoke the pyplot plt function and for the same X values we add the sine curve, as well as the cosine curve here, and both of them will be drawn on the same graph. When you call the plt function twice with different values, Matplotlib is smart enough to know that this represents a different dataset, which is why it plots it in different colors, and the legend is updated accordingly. Matplotlib has picked the colors from the default palette that it has. 0. We've assigned a title to this plot as well; Playing with Plots is our title. Now let's customize the colors in which we view this visualization. We can specify the colors in different formats, including hex codes. Here we've used two different formats that are the most commonly used. Our sine curve will be in the green color, we've specified the full name of the color here, and the cosine curve will be in the magenta color. We use a short form, m is for magenta. The curve is set up as you have specified, and the legend is also updated accordingly. Let's use the NumPy library to generate some random numbers here. We'll generate 20 random numbers and plot them using Matplotlib. We simply pass in the random array. These form the Y coordinates. The X coordinates will be equal to the index of the corresponding Y coordinates. There is no particular order to the random numbers that we've generated. Matplotlib has plotted the data points and connected all of them using a green line. The green color was what we specified. We can customize the line that connects our data points by specifying the linestyle attribute. Linestyle is equal to colon is the notation for our dotted line. The default line as we saw earlier was solid. Linestyle is equal to colon will draw a dotted line connecting our data points. The dotted line is also in green. You can draw a dashed line if you specify linestyle is equal to dash dash. That is the notation for a dash line. You can customize the thickness of each line drawn by specifying the linewidth input argument. Here linewidth is equal to 3, which means the line drawn is thicker than

the default. The default linewidth is set to 1. Now so far our focus has been on the line connecting the data points, but what if the data points are the ones that are significant? We want to highlight them in some way. We can do this by using the marker attribute. Marker is equal to d will use the diamond shaped marker for each of our data points. Each diamond shaped marker in our plot is also green in color. The color attribute applies to both the line, as well as the markers that we'll draw. We can make our markers more prominent by specifying a size for the marker using the markersize input argument. The default size of the marker is 6. A markersize of 10 makes them display more prominently on our plot. The markersize input argument increases the width, as well as the height of the diamonds. If you only want to plot the markers and not the connecting line, you can specify linestyle is equal to None. This will get rid of the line entirely and you're left with only the markers. So far we've only used the plot function from pyplot. The pyplot model also offers the scatter function, allowing you to plot scatter plots. Scatter plots are used to model bivariate distributions or relationships between two variables, one on the X axis and one on the Y axis. When you use the scatter function from pyplot, the markers themselves are plotted by default. There is no connecting line.

## Figures and Axes

A figure in Matplotlib is a high-level container which represents the window or the drawing area where your plots are drawn. Let's see how we can work with figures here. You can instantiate a figure by calling pyplt. figure. This is the function that instantiates a figure, and you can draw axes within the figure. The add_axes function on the figure adds one set of X and Y axes to the figure. The input arguments that we pass into this add_axes function in the form of a list denotes how exactly this axis will be drawn within the larger figure container. The figure that we've instantiated represents the entire container on which this XY axes are drawn. The first input argument to the add_axes represents where in the figure the left of the axes will be positioned. Zero refers to the extreme left of the figure, 1 is the extreme right of the figure, so you can see that any portion in between the figure is represented by a fraction between 0 and 1. The second input argument refers to where the bottom axis is positioned. Zero is the bottom of the figure, 1 is the top of the figure. For the X axis to be drawn anywhere in the middle of this canvas, we have to have this value be a fraction between 0 and 1. The third argument here represents the width of the axis in terms of the fraction of the entire figure with a value of 1 means that this axes fills the width of the entire figure. And finally, the fourth and last argument is the height of the axes in terms of the fraction of the figure height. Here a value of 1 means the height of this axes fills up the entire height of the figure container. You can see that the Python object stored in the ax variable is of

type matplotlib. axes. You can draw multiple axes within the same figure. This is extremely useful if you want to set up 2 visualizations side by side for comparison. You simply call the add_axes function twice on the same figure, and we'll get a resulting visualization with 2 axes. Both of these axes have their left input argument set to 0, which means they start at the very left of the figure. Their bottom input argument is different, though, which means they are positioned one on top of the other. For the axes represented by variable ax2, the value is 0, which means this is drawn at the bottom. For the axes represented by the variable ax1, this is 0. 6. The bottom axes of this plot is 0. 6 units from the bottom of the figure as a whole. From the third input arguments you can see that the first axis spans the entire width of the figure. The input argument is 1. And the second axis spans only 80% of the entire width of the figure. Here the input argument is 0. 8. And finally, each of these axes cover only 40% of the height of the overall figure. Go ahead and generate random X variables from 0 to 10, and we'll plot the sine and cos curves that we saw earlier, but this time each curve will be on a different axis. The plt function for a Matplotlib figure is available on each axes. Ax1. plot will plot the sine curve on the first of our visualization, and ax2. plot will plot the cos curve on the second of our visualizations. If you want to associate X and Y labels with each of these plots, you can do so by setting the labels for each axis in the figure. The set_xlabel function called on ax1 will set the xlabel for the first plot, and the set_ylabel will set the corresponding ylabel. Notice that each of these labels can be customized individually by passing in the correct input arguments. Similarly, ax2. set_xlabel and set_ylabel will set up the X and Y labels for our second plot. Each axes can have its own set of customizations for its labels. Sometimes you may not want your plots to display side by side or one on top of the other; it's possible to have a larger plot and an inset smaller plot. You can have overlapping axes as well. In our code here, the first call to the fig. add_axes function sets up the larger axes which spans the width and the height of the figure. The second call to fig. add_axes sets up the smaller axes, which forms an inset on the larger plot. You can see that the smaller axes starts midway through the overall figure and its height and width is just 40% of the overall figure. So far we've set up each of our figures using the default height and width that Matplotlib uses. You can explicitly define figure height and width in inches. Another way to specify multiple axes within a Matplotlib figure, but this time laid out in a matrix format, is to use the subplot functionality. The fig. add_subplot function takes in a number like what you see onscreen. Here we've passed in 221. This basically means it'll set up a 2 x 2 grid to our overall figure, and the subplot that we add will be to the first position in the grid. In a 2 x 2 grid, the first position refers to the first cell at the top left of the figure. Because there's just one subplot in this figure, it's hard to see the 2 x 2 grid. When we add most subplots, you'll be able to see the matrix grid into which this figure has been divided. Here the ax1 variable holds an object of type AxesSubplot. An AxesSubplot is actually a

derived class of the axes class that we saw earlier. An AxesSubplot is basically an axes drawn within a grid. Let's set up the same subplot as before, but this time we plot a simple line within it. This is the same line that we've seen several times before. To the same figure that contains our line visualization, we add another subplot in the second grid position. The second grid is the one to the top right in a 2 x 2 matrix. This second subplot will have the sine curve, and if you see the resulting plots, we see 2 axes within the larger figure, the first with the line and the second with the sine curve. The input argument 222 indicates that this axis is within a 2 x 2 grid in the second cell position to the top right. Here is the same figure as the previous one with one additional change. We have a third set of axes where we draw the cos curve. This is within our 2 x 2 grid in the third position. And here is where the resulting 2 x 2 grid becomes clear. Here we have the first subplot at position 1. The second sine curve subplot that we added at position 2 in our grid, which is to the top right of our 2 x 2 matrix. And here is our third subplotted position, 3, the bottom left of our 2 x 2 matrix. The subplots need not necessarily be drawn in order. We can have the third subplot occupy the fourth position in our grid. You can choose to draw the cos curve at 224 instead of 223. When we set up the subplot grid, we can position our graphs in any cell within this matrix. The add subplot function only accepts valid positions within your matrix grid. Let's say you try to add a plot to the fifth position in a 2 x 2 grid. This is an error, and you can see the ValueError thrown onscreen here. The error message clearly states that the position num should be a number between 1 and 4, both inclusive. Matplotlib offers another way to specify subplots within a figure by using the subplot2grid function. The input argument to the subplot2grid function is 2 tuples. The first of these tuples indicates the matrix representation of the grid. In our case, we have a 2 x 3 grid as you can see. The second tuple that we pass into the subplot2grid function indicates the row and column values for this particular subplot. This subplot2grid function is used when you want to customize your grid further. Let's say you want a particular graph to span more than row or more than one column, then you can use the rowspan and colspan attribute for the subplot2grid function. Let's see what the resulting plot looks like. Here is our first subplot with the sine curve in our grid with 2 rows and 3 columns. This subplot is drawn in the 0 row and 0 column. And here is our cos curve in the 0 row and first column. Row and column indices start at 0. Our straight line plot is in the 0 row and second column, but it has a rowspan of 2, that means it spans both of the rows of our matrix. The yaxis of this plot is also configured to have its tick position on the right side of the plot. And here is the last plot on this grid. It's positioned at row 1, column 0, but it spans 2 columns. Typically if you're using Matplotlib for a simple visualization which has just one plot, you'll get a figure and one set of axes by simply calling the pyplt. subplots function. This will return an instance of a figure. It'll plot one axis within this figure and return that instance to you in the ax variable. The type of the figure object is a

Matplotlib figure, and the type of the ax object is AxesSubplot. This is the same object that's returned when we call the fig. add_subplot function.

## Watermarks

In this demo, we'll see how we can set up image and text watermarks for our Matplotlib plots. Here is a simple Matplotlib line plot built using the plot functionality in pyplot. We are very familiar with this. Let's go ahead and add a watermark to this. You can call the ax. text function to add this watermark. The first input argument along with the ha and va arguments specify where exactly that watermark will be positioned. You can see that the bottom left of our watermark is at coordinates 1, 4. Here is our watermark text. We want this plot to not be distributed outside our company. It's in fontsize 30 with the color red. Watermarks are typically specified with a low opacity value as compared with the rest of the figures so that they fade into the background. The opacity here is set by the alpha parameter and is just 0. 5. The same watermark can be positioned differently by changing the ha and va input arguments. Ha stands for horizontal alignment, which is to the right, and va for vertical alignment, which is the top. Here the top right of our watermark is at coordinates 1, 4. Let's generate some X values. We'll use these in some visualizations below. We can set up a Matplotlib figure and add a specific subplot at a particular position in a matrix grid and associate a watermark with that subplot as well. Here the larger figure has been divided into a 2 x 2 matrix, and this watermark is for 1 subplot in this figure. We set up the same 2 x 2 matrix with a subplot and a watermark associated with that subplot. We'll then add two more subplots within this figure showing the sine and the cos curve for our randomly generated X values. And in the resulting figure, you can see 3 subplots, but only 1 of these at position 1 has a watermark associated with the Do not distribute watermark. This clearly brings across the fact that the remaining two visualizations may be shared, but the first one is private to your team or your organization. If your entire figure contains plots that are private to your company, you can associate the watermark with the figure as a whole by calling the fig. text function. Figure level watermarks are used to indicate that all of the visualizations within this figure belong to a particular team or a company. All of these are property of Pluralsight in our example. The input arguments to a figure level watermark are exactly the same as axes level watermarks, except that X and Y coordinates are specified in terms of fractions of the figure width and height. The bottom left of our watermark is 0. 1 away from the left edge of the figure, and about halfway through, half the height from the bottom of our figure. We'll now read in an image and use an image as our watermark. We require the image and cbook modules from Matplotlib. Our current working directly where we are writing all of our code is in project/matplotlib, and within this directory we

have the Pluralsight logo. This logo is what we'll specify as our watermark. We first need to read in this image file, which we'll do using cbook. get_sample_data. We then set up the same subplot grid that we used earlier, but this time the figure level watermark is an image, the Pluralsight image that we just read in. We use the fig. figimage function to set up this watermark. Figimage is the function that we have to call on the figure object in order to set up an image as our watermark as you see displayed onscreen. This takes in the image that we read in using the Matplotlib image library. The other input arguments to the figimage function you are familiar with. What's interesting here is the zorder. The zorder refers to the Z axis is the axis which is set to emanate from your screen. A zordre of 3 basically indicates that your Pluralsight image will appear above all the lines and labels in your figure.

## Visualizing Stock Data

So far we've been focused on understanding Matplotlib functionality. Let's bring all that we've learned in this module together by using some real world data. We'll use the Pandas library to read in the stocks. csv file that we have under the dataset folder in the current working directory. The Pandas package is an extremely useful one if you're a data analyst or a data scientist. It allows you to manipulate data in a tabular format with rows and columns. If you don't have Pandas installed on your machine, you can get it using a simple pip install. When you use Pandas, data is read in a tabular format and stored in a dataframe. If you call the head function on your stock_data dataframe, you can quickly explore what data is present within our CSV file that we just read in. This file contains some stock information that I downloaded from yahoo. com. It contains the closing prices of each stock on the first trading day of each month for 10 years, from January 2007 to January 2017. The date column in this Pandas dataframe is read in as a string. We convert this to a Python datetime format by calling pd. to_datetime. It's hard to see here when the data is just displayed, but the date is now in the Python datetime format and can be manipulated as such. Let's add a figure with two axes which will help us compare some stock prices. We want a larger plot and an inset smaller plot. In the resulting visualization, you can see that axis 1 is larger and fills the entire width and height of the figure. Axis 2 is smaller inset within axis 1. Let's now plot the closing prices of AAPL stock on axis 1. This is the larger axis. The title for this axis as a whole is AAPL vs IBM. We're going to plot the closing prices for IBM stock in the inset of this larger plot. And here is a graph showing stock price movements of AAPL stock over the last 10 years plotted in the green color. Because axis 1 fills the entire width and height of this figure, we have set the title on axis 1. This is AAPL vs IBM, where IBM is going to be plotted on the inset graph. Here is the same figure as before with the data for AAPL being plotted in green on

the larger axis. We have the data for IBM plotted in the blue color on the smaller inset axis. And this comparison visualization is now complete with the IBM stock price movement in the inset and AAPL stock price movement in the larger plot. You can now do a quick comparison to see how these stocks have moved. Now if you're actually performing stock analysis, you might want to view how the stock prices of multiple stocks have moved side by side. Let's say you want to see four different companies you're interested in and view their stock movements all in the same figure. That's possible to set up using Matplotlib. Let's add our first axis where we plot the stock price for Microsoft. In our 2 x 2 grid, this plot will be drawn in the first position. Here is our resulting figure which as of now has just one subplot. We have a title for the figure as a whole, Stock price comparison 2007-2017. We've added an axis to the first cell of our 2 x 2 grid and this cell contains the stock price movement for Microsoft over the last 10 years. We can now add the stock price for Google stock in the second cell of our 2 x 2 matrix. Make sure you specify that Google is what we want to plot and we've plotted this in the purple color. We then plot the closing stock price for Starbucks in the third cell of our 2 x 2 grid. This plot is in the magenta color. Notice that we are plotting each of these stocks in a different color so that they stand out in the resulting visualization. And finally, we plot the stock price movement for Chevron in the last cell of our 2 x 2 grid. This is in the orange color. Now we have a single figure with four subplots allowing us make stock price movement comparisons side by side. And on this note, we come to the very end of this introductory module on Matplotlib. Matplotlib is a powerful visualization library which is very well integrated with other packages in the PyData stack, such as NumPy, Pandas, statsmodel, and so on. In this module, we first studied the basic anatomy of a Matplotlib figure and how we could perform very granular customizations of different parts of a figure using Matplotlib APIs. This module introduced us to basic plotting functionality using the pyplot module in Matplotlib. We set up simple plots, labels, titles, markers, and even watermarks using both X as well as images. We studied in detail figures, axes, and subplots, and saw how we could set up multiple plots within the same figure. In the next module, we'll focus on plotting techniques and see how we can use Matplotlib to build basic, intermediate, and advanced plots.

# Building Basic, Intermediate, and Advanced Plots with Matplotlib

## Module Overview

Hi and welcome to this module where we'll see how we can build basic, intermediate, and advanced plots with Matplotlib. If you have a visualization that uses some kind of shape, you can use Matplotlib to plot these shapes on a graph. Matplotlib can plot extremely complex shapes, arbitrary polygons, and Bezier curves that are curves plotted according to a mathematical formula. Often when you plot a graph or a visualization, there might be a single data point or multiple data points that are interesting and you want to annotate or highlight it in some way. We'll see how we can add text annotations to graphs to convey information. In this module, we'll also cover how you can use Matplotlib's twin axis functionality in order to plot different units on the same axis by making a mirror copy or a twin of the original axis. By default, Matplotlib uses the linear scale on both its X and Y axis. You can change the scales on the specific axis in order to better display your data.

## Plotting Shapes

Matplotlib has a special module called patches that allows you to plot arbitrary shapes on graphs. These range from very simple shapes, such as rectangles, and go on to circles and complex shapes that can be drawn using polygons, as well as curves. Instantiate a Matplotlib figure and subplot in one statement by calling the plot. subplots function. You can use the axes to add a patch of any shape. A patch in Matplotlib is basically a 2D object. Any two-dimensional object can be thought of as a patch. The resulting graph has a single rectangle drawn onscreen. We've added a rectangle patch, which is one of the built-in patches that Matplotlib offers. The input arguments that you'd pass into the rectangle object in the patches modules are the same as what you would use to specify an axis within a figure. We specify a fraction of the axis as the left edge of the rectangle and the bottom edge of the rectangle. Both the left, as well as the bottom edge of the rectangle that you've drawn are 0 point away from the corresponding axis. Next is the width of our rectangle, which we have set to 0. 5. This is the fractional width of the rectangle in relation to the axis as a whole. The next input argument is the fractional height of the rectangle in relation to the axis as a whole. This, once again, is 0. 5. Fill is equal to False basically indicates that this rectangle is not filled with any color, only the outliners visible. There is no color fill. By default, Matplotlib draws a rectangle and fills it in the blue color. Notice something interesting about the X and Y axes here? Each axis is just one 1 unit long, but the X axis appears longer than the Y axis. This is because of the default aspect ratio that Matplotlib uses. If you want to change the aspect ratio, you can call the ax. set_aspect function. Setting the aspect input argument to be equal will display the resulting figure as a square. You can see that the graph shows up as a square. Once

you have access to a patch object, you can customize these patches very granularly. Here the fill color will be yellow and the edge, or the border, color will be green for our resulting patch. If you want to draw your shapes with a little design within them, you can specify the hatch input argument to each patch that you set up within your plot. We have two rectangles drawn on this plot here. One of them has the hatch pattern set to dot, and the other to the double backslash. And here is our plot with two rectangles at two different positions. One of them has the dotted pattern and the other has the striped pattern. Notice how we use a for loop to plot two patches of rectangles within our plot. The for loop iterates over a list where the first object is an instance of the rectangle patch object. This is the one with the dotted pattern. And the second object in this list is another rectangle patch, this time with the back striped pattern. The variable p references each of these objects in turn and we add them to the same axis at different positions. We use the same for loop trick to add four rectangle patches to our figure. This time we change the alpha input argument for each of our rectangle patches. Alpha changes the transparency or the opacity of our rectangles. We get four rectangles with four transparency values. But something should have jumped out at you; we can just see three of the four rectangles onscreen here. Let's observe, this is the first rectangle at this position, one with opacity alpha set to none. This is the second rectangle with opacity alpha set to 1. The opacity 1 and none mean the same thing. Both of these are perfectly opaque rectangles. Here is a third rectangle at a different position with an alpha value of 0. 6, and the fourth rectangle is not present within this plot. The left edge of this fourth rectangle is at position 1, which is at the extreme right of the figure, so it's not part of the figure here. We need to fix this, and we can do that by specifying the X limits. We've extended the limits of the X axis to be from 0 to 1. 5. Our fourth rectangle from earlier with its left edge at position 1 is now visible on our screen. This has an alpha value of 0. 1, and notice the X limits of our X axis, it goes all the way up to 1. 5. The fill color for any shape is specified using the facecolor input argument, and there are different ways to specify the color for any shape. Notice here facecolor None, none in quotes, red, and a hex value. When you set the facecolor to be the Python value None, Matplotlib uses the default colors in its palette. The first color in this palette is blue, and that's what Matplotlib has chosen. When you set the facecolor parameter to the string none, that means no color at all. There is no color fill in this rectangle. In fact, you can't even see where it is. You can specify colors using color names as well, such as facecolor is equal to red, blue, green, yellow, and so on. The facecolor input argument also accepts a hexadecimal value for color specification. This is the hex code for blue. These represents rgb values for our colors. If you want to have hollow rectangles drawn with only the border colors of the rectangles specified, you'll see fill is equal to False and specify an edge color. And these edge colors can accept the same formatting as the facecolor input argument. When

you use a patch from the patches module, the edges, or the borders, can be treated as lines and formatted accordingly. Here are the various linestyles that you can specify. Your linestyles can be solid, dashed, dashed and dots alternate, or just dotted. It's not just for rectangles. A patch of any shape can be drawn using these linestyles. The patches module in Matplotlib has built-in classes for all of the common shapes that you might use within your plot. Circles are another type of patch object. The code that you see here draws four circles with different patterns and positions on the screen. Let's look at each of these circles in turn. The patches. Circle class takes in a number of input arguments to draw the circle. As you might imagine, the input arguments that we pass in to instantiate a circle will be different from those that we use with the rectangle. The first input argument is a tuple of which the first field refers to the X coordinate of the center. The X coordinate is specified in terms of the fractional width of the figure. The second field of the tuple is the Y coordinate of the center. This is specified as a fraction of the height of the figure. The second input argument to a circle is the radius of the circle, once again, specified as a fraction. As you can see from the circles onscreen, you can customize the facecolor, edge color, the hatch pattern for any shape that you draw using patches. The patches module in Matplotlib does not restrict to just drawing ordinary common shapes. You can draw any kind of polygon by using the patches. Polygon class. In order to draw a polygon, you need to specify the X, Y coordinates of every vertex of your polygon. Here is a five-sided polygon. Each of these coordinates represents one vertex of our resulting polygon. Here is the first vertex at 0. 1, 0. 1, and here is the last vertex at 0. 4, 0. 3. By default, Matplotlib assumes that you want to close your polygon, and it simply connects the last vertex with the first vertex. If you want to explicitly leave your polygon open, you can specify an input argument to your polygon class, closed=False. You can see here that this results in the same polygon as before, except that the last vertex is not connected to the first. Let's close out this demo by taking a look at one last shape that you might find useful, the patches. Arrow shape. If you think about it, an arrow is basically a polygon with seven sides. The arrow shape takes in just two X, Y coordinates. The first is the center of the base of the arrow. This is at 0. 1, 0. 2 in our demo. The second input argument is the center of the base of the arrowhead, which is at 0. 7, 0. 7. By tweaking these coordinates, you can have the arrow point in any way that you want.

## Visualizing a Bezier Curve

Matplotlib can also be used to plot Bezier curves. These are curves that can be drawn using a mathematical formula. Let's intuitively understand how Bezier curves work before we see it in Matplotlib. Don't worry, we won't be performing any math here. In the last demo, we worked with

shapes, and if you think about it, any vector shape that you draw onscreen is basically made up of a series of points. Here is a vector shape. These are made up of a series of curves, and a curve is defined by a number of different points. The points that you see highlighted on this curve are special in the fact that these are the points that cause this vector shape to actually curve. You can imagine handles on these points, and if you move these handles, you can shape the curve a little differently. And such a shape where control points control the shape of the resulting vector is a Bezier curve. A Bezier curve is defined by these control points, and for different types of curves, you have a different number of control points. The number of control points varies based on the type of curve. At a minimum, a Bezier curve has to have at least two control points. This is a linear curve, and by linear curve we mean a line which simply connects these two control points. Let's say you're working with three control points on a plane. The curve that you get is a quadratic curve. And the curve that is generated using four control points is a cubic curve. Of all of these curves, let's work with a specific example. We'll work with the quadratic curve, which has three control points. This term control point can be used for all positions of the Bezier curve. The Bezier curve is shaped in such a manner that the final curve can pass through all three of these control points. But we use a single term, control point, for these specific positions which help draw a Bezier curve. A control point is made up of two different components, the handle and the anchor. Notice that the anchor is a position actually on the curve, and the handle extends outwards from the anchor and can be used to move that control point. The handles are basically the positions that we use to influence the curvature of the resulting curve. The handles move those anchor positions in such a way that a curve is drawn. Anchors are fixed positions that describe the start and end points of the curve. You'll understand handles and anchors a little better when you see how we draw a curve using these control points. A quadratic curve has three control points, and you can draw an edge connecting each control point to the next control point. We'll now interpolate along each leg of the polygon. Here is a polygon with two legs. We'll set up a point on each edge and interpolate along that edge. We'll set up two points in blue here, point A and point B. Point A is on the edge between control points 1 and 3, and point B is on the edge between control points 3 and 2. We'll draw a line in blue connecting both of these points and we'll define a third point called k on this line. The point k will move along the line connecting A and B. The movement of this point k is what will draw out our Bezier curve. Let's first define how these points will move. Point A will move from control point 1 to control point 3 along the orange dotted line. Point B will move along the dotted line connecting points 3 and 2. Point k, which is on the line between A and B, will only move along this blue line. We can now visualize how the movements of these points will trace out our Bezier curve in space. Notice the movement of point k is along a curve. That is our Bezier curve. A Bezier curve will be defined by the movement of

point k in space. If you draw a line through the path that k followed in space, this is a quadratic Bezier curve, drawn using three control points. And this is an intuitive understanding of how we can use mathematical formulas to draw Bezier curves. Now, Bezier curves find a wide range of users, especially in graphical applications. Bezier curves were publicized about 50-60 years ago by the French engineer Pierre Bezier. They are named after him. He used it to design automobile bodies at the French automobile maker Renault. Bezier curves today are widely used in graphical application, especially computer graphics and other kinds of vector graphics. They're also widely used in animation and typography. If you've design fonts or if you use any of the Adobe Suite of products, chances are you've worked with Bezier curves. Bezier curves in the modern world also find widespread use in web development where they are used to design fonts. TrueType fonts are a special category of fonts that are designed using quadratic Bezier curves.

## Drawing Paths and Bezier Curves

Now that you've intuitively understood how a Bezier curve is plotted, let's try it using Matplotlib. If you want Matplotlib to plot any path, any arbitrary path, you will use a special class from within your patches module called the PathPatch. The PathPatch is instantiated using a Path object. The Path object takes in a list of vertices and a bunch of codes which define the actions we have to perform to move from one vertex to another. When we connect the individual vertices by performing the action specified in the code, that's how we generate a curve or a line. Let's take a look at the resulting visualization and then understand the Path object and see how it was used to produce the set of arbitrary lines. The very first vertex is at 0. 1, 0. 1, that's what you see highlighted here onscreen, and the first action is Path. MOVETO. So think of a cursor as moving to the vertex 0. 1, 0. 1. This is the vertex where we start drawing our path. The next vertex in this path is at the position 0. 8, 0. 8, and the code associated with this transition is the Path. LINETO. Imagine drawing a line from the first vertex to the second vertex, so this is a line from 0. 1, 0. 1 to 0. 8, 0. 8. Here are the coordinates of the third vertex, and you can also see it highlighted in the final shape onscreen. The code is Path. LINETO. So the LINETO action has been invoked. We connect the previous vertex in the path to this current vertex by drawing a line. And here is the last vertex in our path at position 0. 4, 0. 2. And the Path. LINETO code ensures that we draw a line from the previous vertex to this current vertex. If you observe the vertices and the various codes that we specified, the path that was generated was an open path. Our polygon wasn't closed by default. If you want the path traced out to be a closed path rather than an open-ended path, you specify the same set of vertices as before, but the last code that you specified, the one associated with the very last vertex, will be CLOSEPOLY rather than the LINETO. And here is what

the resulting shape looks like. The shape looks a little different because the CLOSEPOLY changed what action had to be performed. Instead of drawing a line from the third vertex specified at 0. 8, 0. 1 to the next vertex position in the list, the polygon was closed by drawing a line from the third vertex back to the first vertex. This is because every action or code is associated with an edge. The last code basically said close the polygon, and the last vertex was ignored. We'll use the same set of vertices as before, but this time instead of plotting a straight line, we'll plot a quadratic Bezier curve by using the CURVE3 code. The CURVE3 indicates that we use three control points and we know that three control points generates a quadratic Bezier curve. And here is the resulting shape with a Bezier curve. The first thing to notice here is that there is a straight line which connects vertex 1 to vertex 2. This is exactly the same as in the earlier paths we saw. This is because of the path. LINETO. This is the action, or the code, that has been specified to connect the vertex 1 and 2. The next code after the LINETO is the CURVE3. Whenever a CURVE3 is encountered, Matplotlib plots a Bezier curve. It uses the three vertices as you see highlighted onscreen as control points for this curve. The first and the third vertex are used as endpoints, and the central vertex highlighted in blue is used as a control point. Now Matplotlib expects as many curves as there are vertices in your list, which is why we specify the path. CURVE3 again at the very end. This last code or action in our list is to move the cursor and to complete the Bezier curve that we have drawn. In fact, instead of specifying the CURVE3 code once again at the very end, we can simply put in the path. MOVETO. This is the same Bezier curve that will be plotted as we saw earlier. Since the first CURVE3 uses the last 3 vertices, the last action can simply be a MOVETO, which completes join the Bezier curve. And you can see the path. MOVETO has produced the same curve that we saw earlier.

## Annotations

It's often useful when you set up a visualization to highlight certain significant results that you might want to point out to your team or organization. This is where we'll use annotations. We start this demo off with a simple plot, which plots X variables against the corresponding Y variables, and their result is just a straight line. We've seen this plot several times before. There's nothing new here. Hypothetically, let's assume that the minimum value that this plot has is something significant and we want to highlight this in some way. We can use the annotate function to add an annotation to point to the minimum value. This is a text annotation, and the text that is written out to screen is min value. The xy input argument, or the annotate function, points to what data point is being highlighted or annotated. The minimum value here is 1, 2. The xytext input argument points to the X and Y coordinates of the bottom left of our text annotation.

Here the bottom left of the min value text is at 1. 5, 2. 0. The fact that this is an arrow annotation is defined by the arrowprops input argument. The arrowprops is a simple dictionary and the only value that we have specified within this dictionary is that the arrow should be green in color. The arrow in our annotation will always point to the XY coordinate that we want to highlight from the text annotation that we've added. Let's say the XY text annotation is now at a different position, 1, 3. The arrow direction will change accordingly as you see onscreen. The base of the arrow is always at the bottom left of the text and the head of the arrow is at the point we want to highlight. You can customize this annotation by specifying other properties in the arrowprops dictionary. Our arrow here is yellow in color with a green outline, and its transparency is 0. 3. Let's say this minimum value is significant in some way. We want to point to that arrow. We also want to highlight the particular data point. In addition to the ax. annotate function, we need to specify an additional plotting function which simply marks the point that is significant. We use the same plot function that we have seen earlier to plot a red circular mark on the line at point 2, 4. This point is significant in some way. The combination of the red marker and our text and arrow annotation serves to highlight this point. Notice one thing here. The arrow overlaps with the marker and that's not really a neat layout. It affects visibility. This is a common problem and very easily solved using Matplotlib. You can shrink your arrow as a whole. It'll still point to the same point, but you can specify a shrink factor within your arrowprops, which will shrink the arrow at its end by 10%. And here you can see the same highlight of the arrow does not overlap with the point. We'll now use the NumPy library and generate some random values and assign them to the variables x1, y1, and x2, y2. X1, y1 contains X and Y points which skew towards the bottom left of our plot, and x2, y2 contains X and Y points that skew towards the top right of our plot. We'll use the scatter function to set up a scatter plot for X and Y values. Both the sets of X and Y values you can see that one set is in red and the second set is plotted in green. We'll now add some stylized text annotations on our plot in order to explain the significance of these red and green dots. There's a lot of code here. Don't worry, we'll take a look at the resulting visualization and see how the individual elements are set up. You can see from the resulting plot that the red dots represent sample A and the green dots sample B. We have wrapped both of the text annotations that we have set up in a bounding box, which we've assigned as a dictionary to the bbox_props variable. The boxstyle property for this bounding box is square, which is basically a rectangle. The facecolor is white, and the boxes are transparent. Their alpha value is 0. 5. The transparency allows us to see the markers that lie below our text annotation. We call the text function on our axes to plot the individual text annotations. Here is the text function for sample A. We specified the position, the text of the annotation, the horizontal and vertical alignment, the size of our font, and the style of the box. And here is the second text annotation for this plot, the one for sample

B. The position and the text for this annotation is different. Everything else remains the same. In the next example, we'll set up the same scatter plot as before, along with the sample A and sample B text annotations. In addition, we want to show some direction of movement, which is significant within the scatter plot. For this, we specify text within an arrow box. Instead of a simple bounding box that is a square or a rectangle, you can have text embedded within an arrow box, and the text will indicate why this direction is such. Specify the box properties as a dictionary, the boxstyle is rarrow, indicating it's a right-pointing arrow. The text annotation that we add to the plot simply says direction. And it has the arrow box props. And here is the resulting visualization, which has the original text annotations for sample A and B, and a directional arrow. The coordinates of this directional arrow represents the center of the arrow. The center of the arrow is at 0, 0. The rotation input argument to this text annotation specifies how the text itself is rotated. The text is at an angle of 60 degrees to the horizontal, and the arrow orients itself along the text. This text is embedded in the arrow box, which shapes itself to match the size of the text. The look and feel of this arrow box can be customized using the properties that we specified in the arrowprops dictionary.

## Scales

So far, the X and Y axes that we have specified have always used the linear scale to represent their range information. We can have each of these axes use a different scale. Let's see how we can set that up in Matplotlib. We'll generate some random data which forms our Y coordinates using the np. random. uniform function. This will have a low value of 0, a high value of 1000, and we'll generate 1000 data points. We'll sort our Y values before plotting them so that the Y coordinates are in ascending order. The X coordinates are simply integers starting from 0 to the number of Y coordinates, also in increasing order. We use the simple plot function to plot X values against Y values. We enable the grid lines in our plot by calling the plot. grid function and passing in true. By default, Matplotlib uses the linear scale for both the axes. You can explicitly specify the scale by calling plt. yscale and pass in linear. This ensures that the Y axis will be plotted in the linear scale. This is just an explicit specification that's the default anyway. And here is the resultant plot. We are familiar with drawing these kinds of graphs. Notice that the plot has gridlines that we specified. Pay close attention to the range of values on the X axis, as well as on the Y axis, and you can see that both of these use the linear scale. We plot the same data as before, but this time we want to express the Y axis in the logarithmic scale. We pass in log as an input argument to the Y scale function. You can see that the resulting shape of the curve has changed, and if you observe the Y axis, the ticks are at values 10 raised to 0, 10 raised to 1, 10

raised to 2, and so on. The default base for your logarithmic scale is the base 10. You can change the base of your logarithmic scale. Simply call plt. yscale, specify the logarithmic scale and say basey should be equal to 2. We get the exponential curve as before and observe the range of values on the Y axis. They are now 2 raised to 0, 2 raised to the power 2, 2 raised to the power 4, and so on. Our Y values are now expressed in terms of log to the base 2. You can scale each of the individual axes separately. Here we have specified the logarithmic scale for both the X and Y axis. The X axis is base2 and the Y axis is to the base 10. The tick values on both of these axes have been updated to represent the logarithmic scale.

## Twin Axis

In this demo, we'll see how we can set up multiple axes on the same plot using Matplotlib's twin axes functionality. All the demos here on in through to the end of the course will mostly use real-world datasets. Here is some weather data for the city of Austin. The original dataset was present at kaggle. com, and you can download it from there if you want to. Read this dataset in into a Pandas dataframe and let's explore what the data looks like. This dataset contains weather information for the city of Austin in Texas over a period of four years. We won't be using all of the data available here, though. Two of the columns that we are interested in are the date and the high temperature value for a day. If you're interested in this dataset, it contains a bunch of other information as well, such as sea level, humidity, pressure, and so on. The third column of data that we'll use in this demo is the average wind speed as you can see highlighted here. Let's trim our dataframe to only include those columns that we are interested in, that is date, average temperature, and the average wind speed. We also limit the data under consideration to 30 days so that we're not working with a huge dataset. And here is the 30 days of data that we'll be visualizing using a twin axis. We are going to build up this plot which uses a twin axis in an incremental fashion so that you can what changes are introduced to our graph at each step. We instantiate a new Matplotlib figure, we set its height to be 6 inches, and width to be 12 inches. The X axis in our plot represents date information. We set date to be the xlabel. We'll use the tick_params function to disable ticks on the X axis and the corresponding tick labels are disabled as well. And here is what the X axis looks like. The xlabel has been set to Date, and you can see that the bottom axis has no ticks and no tick labels. The Y axis of our plot represents the average temperature during the day in Fahrenheit. If you see the resulting plot, you'll see that the ylabel is set to temperature and in brackets F for Fahrenheit. The label is red in color and its size is extra large. Correspondingly, you can see that the ticks of the label are also red in color and the size of the tick labels are large. Having customized our plot, we are now ready to plot the average

temperature per day in Austin using a simple line plot. We have some additional information in our Pandas dataframe that we want to add to this plot. This is the average wind speed for each of these days. We'll now plot them on a twin axis. We want to create a twin of the Y axis which uses the same X axis that we have set up earlier. The ax_wind is generated using the twinx function, which we call on our temperature axis. We haven't plotted any data for wind speed yet. You can see the resulting plot has a twin axis on the right-hand side of the screen. The twin axis range is from 0 to 1 because we haven't explicitly configured any values there yet. The parameters for the twin axis can be configured individually as well. We set a ylabel on the wind axis and also the tick parameters to look a certain way. And the resulting graph will be drawn using these customizations on our twin axis. Once the twin axis has been customized, we are now ready to plot the average wind speed data, which we'll plot on the ax_wind axis. The wind data uses the same X values, and hence the same X axis as our temperature data. The Y coordinates are different, though. We've plotted two different pieces of information on the same plot. They have the same X values, but different Y values, and the different Y values are represented on twin axes. Let's see another example of how we can use the twin axes. This time we'll use it to represent the same data, the temperature information for each of these days, but we'll use different units on each of our axes. Fahrenheit on the left and Celsius on the right. Our original dataset contains a temperature expressed in Fahrenheit. We use this conversion function to convert temperature values to Celsius. You'll find that much of the code that we write for this example is exactly what we've seen before. We set up the temperature axis expressed in Fahrenheit and plot average temperatures for each of our dates. We'll create a twin axes by calling the twinx function as before, but this twin axis will be used to represent temperature in Celsius. Since we're only representing temperature information in this graph, we plot our data in some neutral colors, so we'll use the green color that is separate from the colors that we specify on the twin axes. After setting up our twin axes, which contains temperature in Celsius, we need to figure out the Y limits for this axis. The Y limits for this new twin axis will be the same as the Y limits of the original axis of the temperature expressed in Fahrenheit, except that for the twin axis the temperature has to be expressed in Celsius. We'll perform the conversion next once we get the Y limits. We set the Y limits on the new temperature and Celsius axis, we use the utility function that we had set up earlier to convert Fahrenheit values to Celsius. We can now set up custom configuration for the labels and text for our new temperature in Celsius axis, exactly like we did earlier for our wind speed twinned axis. And here is our resulting visualization. The twinned axis contains the same temperature expressed in Celsius this time. We just have the one piece of data that we plotted, and the twinned axis represents temperature in different units. And this twin axis demo brings us to the very end of this module. We started off this module by seeing how we could use Matplotlib

to plot complex shapes, such as polygons and Bezier curves. We then studied how we could highlight significant portions of our visualizations by using text annotations on graphs. We also covered how we could use the twin axis functionality to plot different units or different representations on the same graph. The twin axis is used in situations where data shared a common axis, but the second axis is completely different. We also saw how we could transform the scales for our X and Y axes by specifying scale parameters to Matplotlib. So far, we've seen how we can exercise very granular control over how our plots look. In the next module, we'll focus on higher-level plotting extractions. We'll see how we can set up the box plot, the violin plot, the stem plot, plots for autocorrelations, and so on.

# Visualizing Statistical Data with Matplotlib

## Module Overview

Hi, and welcome to this module where we'll visualize statistical data in Matplotlib. So far we worked on very granular components where we configured how a plot looks and feels when we finally view it. Here we'll work on many real world datasets in order to visualize these datasets using different kinds of interesting plots. We start off by seeing how we can use box plots and violin plots to represent information and see when we would choose to use the violin plot over the box plot. Histograms and pie charts are two of the most common types of plots used to represent information. Matplotlib has built-in functions for both of these. If you're working with time series data, you often want to figure out correlations between your dataset, or if you have just one dataset, you want to see whether it's autocorrelated. We'll see how we can plot autocorrelated data using Matplotlib. We'll also see how stem plots work, which allow us to plot deltas from the previous record or from a baseline. We'll close this module out by taking a look at how we can set up stack plots in Matplotlib, which stacks information one on top of the other so that we can see their relative significance or importance.

## Visualizing and Customizing a Boxplot

In this demo, we'll first get introduced to the box plot and then to the violin plot. This is used to view important statistical information about any dataset that you're working with. We'll generate

some random numbers to view. Using box plot first, we use these random numbers to first understand the box plot and then use it with real data. We use the np. random. randint function to generate 20 integers ranging from 0 to the number 20. We'll then sort these 20 randomly generated integers in the ascending order. Here is what our resulting dataset looks like. A simple box plot can be plotted using the boxplot function on your pyplot library. Pass in your X dataset, and here is the box plot representation of your data. A box plot is a really interesting visualization that gives you a number of pieces of information. The actual box here represents those values which are between the 25th and 75th percentiles in your dataset. The number 5 is at the 25th percentile and the number around 16 is at the 75th percentile. The central horizontal line that is drawn here onscreen is the median value of our data. The median of our randomly generated data is around 11. 5. A box plot typically has gaps on either end. These gaps represent the range of values, excluding outliers, in our dataset. If there are outliers, they're not included within these gaps. The horizontal bars, which you see connecting the caps of our box plot with the actual box, are the whiskers of our box plot. The length of the whiskers represent the range of the data that are below the 25th percentile and above the 75th percentile. Let's add another point to our randomly generated integers, which is beyond the original range of the data. This is the integer 22 and the highest value in our previous dataset was 20. Matplotlib does not consider the addition of this new value 22 to be an outlier; instead, the top gap of the dataset has extended to include the value 22 and the median of our data has also changed. In order to see how outliers are represented in a box plot, let's add in two integers to our dataset, which are well beyond the highest value that we have so far. We add in 37 and 40. These are clear outliers in our dataset, and you can see that these are represented by special markers. These markers are called fliers within the box plot. The box plot will represent any outliers in the data on either end using these fliers. Matplotlib draws the box plot vertically oriented by default. If you want the box plot to be horizontal, you can pass in the vert input argument and set it to False. If the median value represented on your box plot is significant and you want to draw attention to it, you can add the notch parameter and set it to True. This will indent the median as you see onscreen. By default, Matplotlib will always display the outlier values using the marker representation. If you don't want to see the fliers, you can simply set showfliers to False. Here is the resultant box plot. It's the same as before, but the outliers have been discarded from your final representation. The default box plot that Matplotlib draws is a Line2D object, which only allows you to customize the edges of the plot. By setting patch_artist=True when you instantiate your box plot, we turn it into a 2-dimensional patch. And patches in Matplotlib you already know allow for very fine-grained customizations. The patches object which represents our box plot contains a number of different components, such as boxes, caps, means, medians, whiskers, and so on. Setting the patch_artist

input argument to True allows us to access our box plot as a 2D patch. This 2D patch is filled with a default blue color. Our box plot patch is basically just a dictionary that allows us to access the individual components of this 2D patch using the square brackets. Let's access all of the individual components and understand what they mean. Our box plot contain just one box, there were two whiskers and two caps, one median, and two fliers, but both these fliers are represented using one object, which is why number of fliers is set to one. If we access the box plot as a 2D patch, we can customize each of these components separately. Here we access the only box in our box plot at index position 0 and set its facecolor to yellow, edgecolor to maroon, and fill it using a dotted designed pattern as specified by our hatch input argument. Similarly, we can style the whiskers in our box plot individually by accessing the whiskers by index. The whisker at position 0 is the lower whisker, and the whisker at position 1 is the upper whisker. They can be styled separately and individually. Every individual component in our box plot can be customized. We access the fliers in our box plot. Remember, one flier object represents both of our outlier data, and we set its marker to be a diamond in the blue color. The median value in our box plot can be customized as well. Here we've changed the line style to be a dashed line, and we've increased the width of our median to be 3 pixels.

## Using Boxplots to Plot Exam Scores Data

Let's import the Pandas library and use the box plot with some real world data. This is a dataset that can be downloaded at this site and contains marks for several students in three different subjects. Here is the dataframe that we get as a result of reading in the CSV file. It contains a bunch of different information about students. We're only interested in three columns of data, which contain the math score, reading score, and the writing score for these students. We'll draw box plots to represent this information. We'll trim the Pandas dataframe to include the data only from these three columns. You can call the describe function on the dataframe to get a quick feel for this data. You can see that there are 100 records for 100 students. The mean values for the math, reading, and writing scores are between 67 and 69. Matplotlib plots data represented in the form of NumPy arrays. We can convert our Pandas dataframe into a NumPy array by calling the np. array function and passing an exam_scores dataframe. The resulting array contains three categories of data for scores in each subject. And this is how we typically use box plots in the real world. We pass in the exam_scores_array to the box plot function, and here we have three box plots representing scores in each of the three subjects. These box plots allow us to quickly compare how the students are doing in each of these three subjects. In order to customize the visual look and feel of this box plot, we need to access the box plot as a patch. We do this by

passing in patch_artist=True. Let's print out information about the various components of our box plot patch, the number of boxes, whiskers, caps, medians, and fliers. Our exam scores box plot has three boxes. That makes sense, one for each category. You have a total of six whiskers, two whiskers for each box. The number of caps which indicate the range of our dataset is also equal to six, two caps for each subject, the number of medians is three, one for each box, and the number of fliers is three, one for each category or each subject scores. We want to represent these boxes in specific colors. Let's specify the colors. Maybe these are brand colors for our organizations. Blue, grey, and lawngreen in a list. We instantiate a box plot and access it as a patch object. We can then iterate through all the boxes that are present in our box plot and set its facecolor. The facecolor, or the fill color for each of our boxes, is set to a different color from our colors array. We've also colored the caps as well. The top caps are set to the same color as the box itself. This box plot is not complete, though. It's still missing some information. If you take a look at the X axis, you can see three categories, but we have no idea which box plot represents which subject scores. You can specify the correct labels for the X axis in our box plot by calling plt. xsticks. The categories 1, 2, and 3 correspond to Math, Reading, and Writing scores. Once you plot these xlabels, it's pretty obvious that box plots are used to visualize data which are grouped into categories.

## Violin Plots

Let's work with the same exam score data that we set up in an earlier clip, but this time we plot the same data using a violin plot instead of a box plot. A violin plot represents the exact same information as a box plot, except that it's called a violin plot because of the shape of the resulting visualization. The shape of a violin plot conveys additional information to anyone who's viewing this graph. This displays the density of the dataset over the entire range of values. For example, if you take a look at the first violin plot, you can see that it's very thick around the score 70, which means most of the data points for our math score are clustered around 70. Unlike with box plots, by default, violin plots don't display median information. In order to view median information in violin plots, you need to explicitly pass in the showmedians input argument and set it to true. This is the same violin plot as before, but the medians are represented using a horizontal line. We've also enhanced the readability of our plot by using the xticks function to specify labels for our X axis. The first plot is for the math scores, the second for reading, and the third for writing scores. You can horizontally orient your violin plots by passing in the vert input argument. Vert is set to False for horizontal orientation. In the horizontal representation of our violin plot, we need to specify categories on the Y axis rather than the X axis. The labels for our math, reading, and

writing score categories are applied using the yticks function. Violin plots are patches. They're basically curves that are used to represent data density. So when you instantiate a violin plot, you get a patch object as its return value by default. Here are all the components that make up a violin plot: Bodies, cbars, cmaxes, cmedians, and cmins. Only the bodies component of each violin plot can be configured and customized individually. Here we've set the facecolor for each of our bodies to the colors that we had set up earlier in the colors array. You can see that each violin body in our violin plot is now in a different color. When you're customizing the other components of a violin plot, you have to apply the same customization for all of the shapes that you've plotted within a violin plot. The cmaxes, cmins, cbars, and cmedians, these customizations apply to all of the violin shapes. Here you can see in the result that the max ranges for all of our violin bodies are maroon in color. The min range in each violin body is black in color. The bars which connect each of the range ends are with the linestyle dotted. And the medians are depicted using a thick four pixel line. You can customize the legend that you add to box or violin plots by specifying what bodies or boxes you want to represent within the legend. Here we've set up a legend only for bodies 0 and 1, only 2 out of the 3 datasets that we have plotted on this plot. The scores for math and reading are represented in our legend. If you're displaying only a subset of data in our legend, make sure you indicate the right labels. We are only showing math and reading scores, as you can see here. And the position configuration of our legend says that the legend should display in the upper left corner.

## Histograms

One of the most common kinds of visualizations that you're likely to use when you have a dataset are histograms. In order to plot histograms, you need to divide your X axis into ranges or bins and specify the Y axis as the frequency or number of data points that are present in each bin. We'll work with data that's contained in the national_parks. csv file under our datasets folder. The data assembled in this file has been put together from the original dataset that's available on this website that you see onscreen. If you explore the data in the resulting dataframe, you can see that it has four columns of information. The first is the year, and the remaining three columns contain the number of annual visitors to each of these U. S. National Parks, the badlands, the Grand Canyon, and the Bryce Canyon. If you call the describe function on the dataframe, that should quickly show you that there are a total of 57 records for 57 years of data. Let's plot a histogram of the number of visitors to the Grand Canyon. This we can do by calling the hist function on our pyplot library and passing in the column that contains the number of Grand Canyon visitors in our Pandas dataframe. The facecolor and the edgecolor input argument allow

us to customize how our histogram looks. Our histograms are in the cyanne color with a blue outline. We can also specify the number of bins that we want our dataset to be divided into. The entire range which represents the number of annual visitors to the Grand Canyon have been divided into 10 bins. The height of each of these histogram bars represents the number of data points that fall into each bin. From the tallest histogram here you can see that there were 16 years where the number of annual visitors was between 4. 25 and 4. 75 million. The width of a single bin is about 0. 5 million or 500, 000 visitors. Any Matplotlib histogram representation has three different components. N, the frequency of each bin, bins, that is the number of bins, and patches, the shapes that define the histogram bar for each bin. If you print out the values for each of these components, you can see that n is a 10 element array, and each element of the array is the frequency of the number of data points in that bin. For example, the frequency of the first bin is 5, the second bin is 9, and the frequency of the highest histogram bar is 16. The bins component is a list which contains the middle value for each bin. The first bin has a middle value of roughly 1. 253 million, the second bin at 1. 75 million, and so on. The patches component is a list of patch objects where each patch represents one histogram bar. Histograms are typically used in two different ways. One way is to have the frequency counts on the Y axis, another way is instead of frequency, we plot the probability density for each bin on the Y axis. We can customize a histogram to plot probability densities by specifying True for the density parameter. The resulting histogram looks exactly the same as the previous frequency-based histogram, except for the Y axis. The Y axis now has probability densities. You can see that the probability density of the tallest histogram bar is about 5. 6 into 10 to the power minus 7. From the X axis, we can tell that the interval for each bin is about 0. 5 million, or 500, 000, so 500, 000 multiplied by 5. 6 multiplied by 10 to the power minus 7 is 0. 28. The probability density for the highest histogram bar is 28%. Histograms can be plotted with cumulative values as well. In a cumulative histogram, each bar represents the frequency count for that bin, plus the frequency counts of all the bins that came before it. You can see that the histogram bars are monotonically increasing here. Let's plot the histogram for the Grand Canyon visitors as before. In addition, we want to add a second histogram to our plot, which defines the visitors to Bryce Canyon as well. The bars of the second histogram are yellow in color. The number of bins is different; there are just 8 bins in this histogram. The second histogram has been placed on top of the first histogram, and in fact, hides some information from the first histogram. The histogram that's plotted second will appear on top. So if you switch the ordering of the two histograms so that Grand Canyon visitors are plotted second, the blue bars will appear on top of the yellow bars. If you are overlaying histograms in this manner, it's good practice to specify the opacity of the second histogram and make it more transparent. Here the

alpha value for the second histogram is 0. 3. It's more transparent. It does not hide the bars of the first histogram.

## Pie Charts

Another very popular visualization mechanism, especially when you're working with percentage data that sum to 100, are pie charts. Let's see how we can construct pie charts with Matplotlib. We are going to read in data from a CSV file. This is called sector_weighting. csv. Assume that this contains the breakdown of the investment portfolio of some mutual fund. The mutual fund holds company stocks from different categories, such as general financial, beverages, life insurance, tobacco, travel, and so on. The percentage values represent what percent of their fund is invested in these categories. These categories all sum up to 100. A pie chart can be formed by calling the pie function on the pyplot library. Simply specify the data that you want to plot within the pie chart and the labels that you want to use. You can specify the columns in a Pandas dataframe, and here is our resulting pie chart. The resulting pie chart is a little skewed because of the default aspect ratio that Matplotlib uses. It isn't a perfect circle. But all of the labels and their percentage contribution towards the investment portfolio are specified here. If you want to display your pie chart in the form of a perfect circle, you specify the axis function, and the input argument to that should be equal. Plt. axis equal will display the pie in the form of a perfect circle. Let's customize the colors in this pie chart by setting up a list of colors. These could be your organization's brand colors. You plot the pie chart as before, you pass in the list of colors as a part of the color's input argument. Colors sets the color for each sector, and the autopct represents the format of the displayed percentage values. The colors in this pie chart are completely different. It's drawing from the list of colors that we had initialized earlier. And you can see the labels on each pie wedge. These are the percentage values for each wedge of the pie, and they're formatted to two decimal places. If you notice the wedges in the resulting pie chart, you can see that the first sector, that is General Financial, starts at the 3:00 position. The next sector is then laid out counter-clockwise. The next sector is the Beverages sector, then Life Insurance, and so on. You can customize how the pie wedges are laid out by specifying the startangle input argument. The startangle of 90 degrees will start the first wedge at the 12:00 position. The counterclock argument is set to False, and the wedges will be laid out clockwise. Let's see the resulting pie chart. Startangle of 90 degrees starts off our first wedge at the 12:00 position, and counterclock=False will lay out our sectors clockwise around the pie. Specific sectors in our pie chart can be highlighted by using the explode functionality. Explode allows us to highlight a wedge by separating it from the pie by a fraction of the pie's radius. Here we specify a list of

explode fractions for each wedge in our pie chart. While creating our pie chart, we pass in the explode input argument and set it to our explode list that we initialized earlier. Here is the resulting pie. An explode of 0. 1 extracted Beverages wedge just a little bit. The Beverages wedge was pulled out by a value equal to 10 percent of the radius of the pie. This explode fraction was applied to the second element in our pie, which is the Beverages sector. The explode fraction value of 0. 3 is applied to the Tobacco wedge, and you can see that the Tobacco wedge has been exploded, or pulled out even further from the pie. There are three components that make up any pie chart in Matplotlib, the wedges, the texts, and the autotexts. Let's print them out to screen and see what they are. Wedges are a list of patch objects where each patch objet represents one sector in our pie chart. The texts component refers to all the text labels in our pie chart, along with their position in the chart. The texts here contains all of the sectors in our mutual fund portfolio. Autotexts represent the numeric values that correspond to each wedge in our pie. Autotexts is available only if autopct is set for the pie chart. Only then are autotexts displayed. Each wedge, texts, and autotexts in our pie chart can be individually styled. Here we've styled the wedges, texts, and autotexts corresponding to the Beverages sector. The Beverages wedge is in the blue color with the 2 pixel border. The text associated with it is cursive, and the autotext is in bold with the size of 15.

## Autocorrelation

When you're working with time series data that is the statistics data available across a period of time, sooner or later you're going to come across the term autocorrelation. In order to understand what autocorrelation means, let's start off with the term correlation. Correlation is basically a measure that expresses the relationship between two items or two variables or two datasets expressed in time series. In the term autocorrelation, the only additional term that you see here is auto, which stands for self, and autocorrelation basically means the relationship of any variable with itself. Autocorrelation for time series data measures the relationship between a variable's current value and any of its past or future values. For example, let's assume that you're predicting the weather and it has rained today. Once you have that piece of information, you know that it's more likely that it's going to rain tomorrow. The weather for two consecutive days tend to be highly correlated. For example, if it was sunny today, you know that it's less likely to rain tomorrow. The weather may not change suddenly. I know that this is not true for several parts of the world, but you know that this is true in general. Let's see how we actually plot the autocorrelation for any time series data. Consider the X and Y axes as you see onscreen here. The X axis represents time. The data in its original form may be made up of a series of data points

representing different values over time. Now if you use the same series once again, you'll shift this entire series, you'll lag this over one or more time periods. Here is the same time series lagged by, say, one time period. You can lag this further and get the same time series over a different time period. Let's assume that the green dots represent data at time period t, and the maroon dots represent data at time period t-k. Now the value of k might be different. There is no constraint on the value of k; k can be equal to 1, 2, 3, anything. The correlation between time series data in its original form and the same time series data lagged by 1 or more time periods is called autocorrelation. The value of autocorrelation is a fraction that ranges between -1 and 1, -1 at one end of the range, and +1 at the other end of the range. An autocorrelation value of +1 indicates perfect positive correlation between the 2 time series. An autocorrelation value of -1, on the other hand, indicates perfect negative correlation, or the datasets are not correlated at all.

## Plotting Autocorrelations in Time Series Data

In this demo, we'll see how we can explore autocorrelations that exist within time series data using Matplotlib visualizations. We'll work with National Parks data as we have done before. This time we'll read in data from grand_canyon_visits. csv. This dataset contains the number of visitors to the Grand Canyon National Park on a monthly basis. This file was created using data originally available at this website linked that you see here. If you explore the dataframe created, by reading in the CSV file you'll see that it has two columns. The first column refers to the month, and the second column contains the number of visitors to the Grand Canyon for that month. If you call the describe function on this Pandas dataframe, you can see that this dataframe contains a total of 84 records representing about 7 years of visit data. When you're working with the autocorrelation function in Matplotlib, large numbers may result in an overflow and the autocorrelation cannot be calculated. To avoid this, we divide the NumVisits values by 1000. This is a common trick typically used when your data contains large numbers which might overflow during a mathematical operation. Here is the resulting dataframe, the same original records, all 84 of them, the NumVisits column is now represented in units of thousands. Plotting the autocorrelation graph in Matplotlib is very straightforward. We call the acorr function on our pyplot library and pass in the NumVisits. This is the time series data for which we want to plot autocorrelation. The maxlags input argument is 20. And the resulting visualization will show us the autocorrelation values for this time series data. The maxlags is 20. You can see that the X axis ranges from -20 lags up to 20 lags. Notice that the tallest line here corresponds to a correlation value of 1. This is at timelag 0. Any time series data is highly correlated with itself. The correlation with the same dataset will be perfect, have a value of 1 as you see onscreen. You can also see that correlation values are very

high when the data is lagged by just 1 time period. One time period here refers to 1 month, that means the number of visitors to the Grand Canyon for any month is a good prediction of how many visitors the Grand Canyon will have in the next month. Other peaks in this correlation data that you can see here are at the lag of +12 or -12 in the series. This means that there is some seasonality to the number of visitors to the Grand Canyon. The number of visitors the Grand Canyon had in the July of 2015 is a good predictor of the number of visitors who will visit in the July of 2016. The number of bars that you see in this autocorrelation graph is determined by the maxlags parameters. You can see the correlation values for all lags ranging from -20 to +20. Any autocorrelation plot in Matplotlib is made up of four different components, lags, the correlation values, the vertical lines, and the horizontal line. In our case, we have 40 elements in our lags list, 1 corresponding to each time lag ranging from -20 going up to +20. C contains the correlation values corresponding to each time lag. At the very center corresponding to time lag 0, we have correlation value 1. The vlines and the hlines are line objects corresponding to the vertical bars and the horizontal bars.

## Stacked Plots and Stem Plots

In this demo, we'll see how we can visualize data using stacked plots in Matplotlib. We'll work with the same National Park data that we are very familiar with. We'll read in from national_parks. csv. This file contains annual visitor information for three national parks in the U. S. This file is compiled using data available at this website here. The first column in the dataframe that we just read in contains year information and the remaining three columns contains the number of annual visitors to the badlands, Grand Canyon, and Bryce Canyon National Parks. When we represent this data using a stacked plot on the X axis, we'll plot the year information, access the year column, and place it in the X variable. We use NumPy's vstack method to create a vertical stack of our data. This is a two-dimensional array with each series stacked one on top of another. We have the badlands, the Grand Canyon, and the Bryce Canyon. All of this data will be in the form of arrays stacked one on top of the other. We have the number of annual visitors to the badlands at the very top, then the annual visitors to the Grand Canyon, and finally, the number of annual visitors to Bryce Canyon. Set up the labels associated with each dataset in the form of a list. Assign this list to the labels variable. These are the labels for our stack plot. A stacked plot can be drawn by calling the stackplot function on our pyplot library. We pass in the X values, the Y values, and the labels for our X values. And here is the resulting stack plot. You can see that the number of visitors for each national park is represented using a different color, and we have a legend indicating what color corresponds to what national park. These default colors have been picked

by Matplotlib. If you want to customize the colors in which your stack plot is displayed, you can specify a list of colors and pass this in as an input argument to your stack plot. The stackplot function takes in a colors input argument, point that our list. The edge color for each of our stacks is gray. That is our boundary, or border. You can see here that we have the same stack plot as earlier with a gray boundary for each stack and the colors are sandy brown, tomato, and sky blue. The differences in the number of annual visitors to each of these national parks can be visualized using an interesting visualization called a stem plot. Let's modify our national parks data to show the difference in the number of visitors from the previous year. We can call the diff function on a Pandas dataframe. This is a single function call that will allow you to get the difference in value for each cell from the corresponding value in the previous row. So the dataframe will have the same column values, except that the values in each cell will now have the delta, so the differences from the previous record. Here is the dataframe that was generated by calling the diff function. You can see that the very first record has all NaN values because there isn't a previous record for it to compute the difference. Stem plots are a great visualization tool to use to analyze fluctuating data, data that fluctuates up and down from one time period to the next. We simply call the stem function on the pyplot library and pass in the year information on the X axis and the difference information on the Y axis. We'll plot only the differences for badlands. And here is the resulting stem plot. There is a horizontal line at 0, and we have stems emanating from this horizontal line. The stems go up for positive deltas, they go down for negative deltas. And on this note we come to the very end of this module and this course on Matplotlib. In this module, we focused on understanding the nuances of the box plot and the violin plot and understood why we would choose to use one representation over another. If you want to see the density of points at any value, you'll use the violin plot. We used Matplotlib to plot common visualizations, such as histograms and pie charts, and we also saw how we could customize the look and feel of these plots. We then understood what autocorrelated data means and plotted autocorrelations on time series data. We also used stem plots in order to view changes from a baseline. And finally, we learned how we could generate and customize stack plots using Matplotlib. If you're interested in data analysis and visualization, there are other interesting courses on Pluralsight that you could watch. Visualizing Statistical Data Using Seaborn introduces the Seaborn Python package. Seaborn is a high-level abstraction built on top of Matplotlib. It allows you to set up very complex visualizations in a simple and intuitive manner. If you're interested in data visualization as a precursor to machine learning, here is a good ML course for you to start off with, Building Machine Learning Models in Python with SciKit-Learn will introduce you to machine learning using the scikit-learn Python package, a very popular open source package for ML models. If you're interested in other libraries in the PyData stack, Working with Multidimensional Data Using

NumPy is a good one for you to watch. On this note, we come to the very end of this course. I'll say good-bye. Thank you for listening!

## Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

## Course info

| | |
|---|---|
| Level | Beginner |
| Rating | ★★★★½ (19) |
| My rating | ★★★★★ |
| Duration | 2h 8m |
| Released | 14 Aug 2018 |

## Share course