Pandas Playbook: Visualization

by Reindert-Jan Ekker

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hi everyone. My name is Reindert-Jan Ekker, and welcome to this playbook about visualizing your data with Pandas. I'm a senior developer and freelance educator, and in this course, I'll teach you about visualizing your data with Pandas, the most popular Python framework for doing data science and data analysis. When working with a dataset, being able to make attractive and informative visualizations is extremely important, for example when you're exploring the data or when you want to communicate your results with others. This course gives you an in-depth understanding of how to create beautiful plots for your dataset using Pandas. You will learn how to create common plot types like bar plots, area plots, scatter plots, and many more, how to customize the way your plots look using line styles, color themes, configuring axes and legends, etc., etc., how to prepare your dataset for plotting, and how to follow best practices, and we'll also see some other plotting libraries besides Pandas, namely Seaborn and Bokeh. By the end of this course, you'll be able to create most plots you will need in a common work flow using Pandas, customizing it to show the data in exactly the way you want and making it attractive too. Before beginning the course, you should be familiar with the very basics of Python, Pandas, and data science. I hope you'll join me on this journey to learn how to visualize your data with Pandas with this playbook course at Pluralsight.

Course Introduction

Introduction

Hi. My name is Reindert-Jan Ekker, and welcome to the Pandas playbook about visualizing data. In this introduction module, I'll give you a short overview of what to expect from this course. I'll tell you what you already need to know and what you need to have installed on your system to be able to follow this course. And finally, I'll give you an overview of the modules in the course and the topics I cover in each module. So what can you expect from this course? Broadly speaking, I'm going to teach you how to plot your data using Pandas, and I'll do this from a technical point of view. And by that, I meant that I'll focus on how to write working code, what message to call in what order, etc., etc. I will assume that you already have a working knowledge of Pandas DataFrames and that the basics of data visualization are also familiar. So I will not explain, for example, what a scatter plot is, but I will teach you how to write the code to create scatter plots. One thing that will play quite an important role throughout this course is matplotlib. It's the Python visualization library that makes almost all our plotting possible. Now there are various confusing ways to work with matplotlib, and that's why I will take some time to make sure that you understand exactly how all of this works. Once you understand this, we will cover some more advanced ways to customize your plots and make them look good. I will also go into some other libraries besides matplotlib and Pandas, namely Seaborn and Bokeh. That means that there are some things I will not cover in this course. First of all, I will not give an introduction to Pandas or data structures like the Pandas DataFrame. I will assume you already know about these things. If you need a general introduction into these topics, please first watch the Pandas Introduction course. And this is also not an introductory course about data science or data visualization. I'll assume you know about datasets and analyzing them, and you should also be familiar with the basics of visualization. Common plot types like a histogram or a heatmap should not be new to you, and I will focus on how to make them, not on the question of what constitutes a good or appropriate visualization. Like I said, this course covers the material from a technical point of view, not a theoretical one. And some of the plots we'll make are not optimal from a data scientific view point, but they will serve just fine as examples of how to call matplotlib methods. So what should you already know before watching this? Well, of course, you need some basic Python knowledge. At the very least, you should have an understanding of the built-in data structures, like lists and dicts. You should be able to write functions, and you should have a minimal understanding of what classes and objects are. You should also have at least a tiny bit of knowledge of Pandas. Let's say you have played around with it a little bit, and you have an understanding of what a series, a DataFrame, and an index are. Actually it would be even better if you have some knowledge of how to manipulate and transform a DataFrame. If you don't know this, you might want to watch my other course titled Pandas Playbook: Manipulating Data before watching this one. Then there's the Jupyter Notebook, which is the tool I will use to write and run my code. You should have this installed and running, and you should know how to work with notebook cells, write, edit, and run code. In case this is new for you, there's a Pluralsight intro course about Jupyter as well. By the way, another tip from me, install Anaconda from anaconda. org. This will install Jupyter, numpy, and Pandas, as well as a wealth of other popular packages for data science and analysis. At the end of this course, I'll give a short overview of two visualization libraries other than Pandas, and you might want to install these. They're Seaborn, which has a focus on statistical visualization. and by the way, if you use Anaconda, this is included in there, so you won't have to install it. And we'll also take a look at Bokeh, which is aimed at creating visualizations for the web. Like Seaborn, this library is included in Anaconda. So if you use Anaconda, you won't have to install it separately. This course consists of five modules. In a moment, we'll start with making simple plots, and I will introduce you to the basic concepts of plotting with Pandas. In the module after that, I will teach you about the various plotting APIs that matplotlib and Pandas have to offer and how to make sense of them. After that, you're ready for the next module covering all kinds of advanced ways to customize reports. In the final two modules, I will give a brief introduction of two other plotting libraries called Seaborn and Bokeh. Very well. That gives you an overview of this course. Let's get started with the next module and learn about the basics of plotting with Pandas.

Making Simple Plots

Module Overview

Welcome to this module called Making Simple Plots. In this module, I'll give you a gentle introduction to the Pandas Plot API. I'll be focusing on making some very simple, straightforward plots, and we'll see what the various kinds of plots are that we can make through this Plot API. I'll also show you how to prepare your DataFrame for plotting. So let's go straight to the demo. I'll introduce you to the Pandas Plot API and mainly the DataFrame plot function. This allows us to plot the data in our DataFrame. And of course before we do that, we have to select what data we want to plot. Along the way, I'll showcase various kinds of plots that we can make with this plot function.

Demo: Introducing the Pandas Plot API

So here we are in a running Jupyter Notebook. As you can see, we have several CSV files here, which I will make available for download as the course materials, and there's also a Jupyter Notebook here. Let's look at it. Now when working with Pandas, the fastest and most convenient way to visualize your data is through the Pandas Plot API. Many everyday visualizations are very simple when you use Pandas. Now before we start, please note that I'm importing Pandas, and I'm assuming this is familiar to you, and I'm also adding another line, %matplotlib inline. Now this makes it possible to include our plots directly in the Jupyter Notebook. We'll talk more about this later. For now, let's create a DataFrame and make some plots. I'm importing a small dataset containing water temperatures and heights measured over the last two days of the coast of the Netherlands. Let's see what our data looks like. So we have three columns, one of which has type object and two that contain numbers, a temperature in floats and a height in ints. Now the most straightforward thing to do is to take our dataset and call plot on it. This plots all columns that contain plotable values. In this case, we see the water height going up and down with the tide, and the temperature staying more or less steady. Pandas generates the legend for us, which is the little box on the lower-right side, which shows the column names and the colors of the lines, and it also generates the axis and the ticks on the axis. Now the values on the X-axis are taken from the index of our DataFrame, and right now we have an index that simply goes 0, 1, 2, 3, etc., etc. for every row index, and that is what you see on the X-axis as well. This isn't very informative, and we'll see how to fix that later. By the way, we also see a very cryptic line of output here. This actually is the return values of the call to plots, which is a so-called AxesSubplot object. We can avoid this output by assigning this object to a variable. I'm going to say x = here. And then rerunning this cell, now we see just the plot and not this line of output anymore. Now storing the Axes object like this is actually very useful, and we'll see more of that soon. Now df. plot, by default, plots all columns in a single figure. But if we give it the argument subplots is true, this will put each column in its own little plot, and now we can see a little more detail on the temperature data. What I can also do is add a title. In many cases though, you'll want to select the subset of your data to plot. Let's say I want to plot only the temperature column, and we can simply select that column and plot it. Now in this case because we're plotting only one column, Pandas doesn't generate a legend, and that's why I like to add a title so at least you can see what we're plotting here. Now don't you think it would be nicer to have the actual timestamps on the X-axis. Now to do this, I first have to convert the datetime column from objects into actual datetime objects, and I can do this using the Pandas to_datetime function. So now we see that the type of the datetime column has changed from object into datetime64, and this means now I can plot the same line, but specify that I want the datetime column on the X-axis. And this already looks a little better. Now what I don't like is the fact that we see the column name on the bottom now. It says datetime, which I think is ugly. And I also don't really like the way the date and time are displayed here. Now we'll see how to fix both of those in a moment. Now maybe I'm more interested in the distribution of this data than how it changes over time. We can make a histogram instead of a line plot by saying plot. hist. But actually, I think this is quite ugly. So let's show the distribution with another plot, a box plot. For this we can say plot. box. So now you can see that the data is slightly skewed to the right. So now we've already seen three different types of plots, lines, box plots, and histograms. But the Plot API can make other plot types as well, and I'd like to show you some other examples. But to show you that, I need different datasets. Let's load nobel. csv, which contains info about all Nobel Prize winners, and let's try to visualize Nobel Prizes awarded by country. We do have a Country table here, so let's use it to count Nobel Prizes by country. For this, I can use the value_count function. Let's take a short look at the result here. As you can see, it shows every single country in the dataset and the amount of times that it shows up. So this is a very simple way to determine the Nobel Prize winners per country. Now one approach to show this would be to make a pie chart. Let's try this. I'm simply going to say. plot. pie, and the result here is not really nice. There are far too many countries and, as you may know, pie charts are usually not considered to be a very good visualization tool because, among other things, it's quite hard to judge the area of each slice of the pie. Instead, let's select the first 10 items from the list and assign it to a variable, top_countries. And now I can make a bar chart that only shows the Nobel Prizes for these 10 countries. And to make a bar chart, you probably guessed it, I can say plot. bar. Very well. Well I think this is a much nicer result than the pie chart. So that's a little introduction to the kind of things we can do with the Pandas Plot API. Now if I type df. plot and then a. and I press Tab, we see an overview of all the functions we can call. And we see that there are some plots that Pandas can make that we haven't seen yet. For an overview, let me take you to the Pandas documentation page about visualization. And if I scroll down here a bit, it gives us an overview of all the plot functions that are available. The ones I haven't shown yet are scatter, area, hexbin, and density plots, and most of those, I will show in this course. The ones I won't show, well you can look up the documentation right here.

Review: The Pandas Plot API

So we've just learned about the method df. plot. What it does is it creates a line plot where it includes data from each column in the DataFrame, at least each column that contains data that we can plot. The values on the X-axis are taken from the index of our DataFrame, and the values on the Y-axis are taken from the values in our columns. The legend is added automatically by Pandas, and it shows the various names of the columns together with the color of the line that each column is plotting. Now as you can see, there's some things that Pandas doesn't do. It doesn't set the title, and it also doesn't set labels for the X and Y-axis. So usually, you will want to do some customization. We've also seen two arguments that we can pass to plots. The first is title. So here I say, for example, title is Water measurements of the Dutch coast. And as you can see on the left side, that's the title that appears about the figure. The second argument I pass is subplots is True. And when I pass that, Pandas generates a separate subplot for each column in my DataFrame. And one of the nice things about this is that now the data in our columns doesn't have to share the same Y-axis. We've also seen that there's a set of plot functions that we can call. If we just call. plots, it gives us a line plot. But we can also say df. plot. bar, which gives us a bar plot as you can see on the left side. Or we can generate a box plot with. plot. box or a pie plot with. plot. pie, etc., etc.

Demo: Three Steps for Simple Plots

So let's see the next demo. I'm going to give you a rule of thumb for creating simple plots. There's three steps you usually want to take. First, you prepare your scripts by doing the right imports. Next, you prepare your DataFrame. And once you've prepared your data, of course, you want to call the right plotting function. So in most simple cases, there are three steps to successfully plotting your data. The first one is trivial, and we've already seen it in the previous demo, prepare your script for plotting, by which I mean import Pandas, say %matplotlib inline, and we also import matplotlib. pyplot as plt. Now both the first and the last line I will explain more about in the next module. For now, please just assume that this is necessary to plot with Pandas. Now the second step is not that easy. In fact, it is the most important step, and it is prepare your data. You see, when you prepare a DataFrame for plotting in the right way, you avoid a lot of misery, and you can create most plots with a single call to plots. Let me give some examples. Here I'm creating a DataFrame. It contains grades for a number of students in a class. The data looks like this. Now let's try to make some plots with this data. Let's start by plotting the mean score for every test. I can calculate the mean by saying grades. mean. And looking at the result here, it has each teach name as the index, and in the column, it has the averages, and this is perfect for plotting. So in this case, I've already prepared my data. Now the third step after preparing the script and preparing the data is to choose the correct plotting function. Let's start by calling plot, and I'm going to give it a title as well. Also, I shouldn't forget it's a best practice to assign the results value from this call to a variable. Now this honestly doesn't look very attractive. The test scores aren't related so I don't want to connect them with a line. So in this case, it's better to make a bar plot, so let's call plot. bar instead. Now similarly, we can plot the average grades per student. We only have add an argument, axis is 1, to the mean function and it will calculate the average per student. Of course, I have to change the title as well. And just because I can, I'm going to change this into a horizontal bar chart by saying barh. Now running this, so what you see here is just by thinking clearly about what my data looks like and seeing that I can create a Pandas series with exactly the right index and the right values, I can create exactly the plot I want, and I'll only have to call plot. barh. And, okay, I have to add a title. That's the only thing I have to add here. But basically, my plotting code is extremely simple. Now in this cases of these two bar plots, I had a series that was just ready to plot without any extra effort. But what if I want to present both the students grades and the averages in one plot? Let's think about what the data should look like. So for every student we want to show their score and their averages. Looking at the grades DataFrame, we already have grades per student. Let's add a new column for the averages. So now if I look at grades again, we have an extra column containing averages. If I plot this, now that's nice. It shows the grades for each student, as well as their average because, of course, in the index we have the student names, and we get a colored bar for every column and whatever columns is the average. But I'm missing the class average for each test, so I want another set of bars with the average score for each test. Looking at the DataFrame again, let's scroll up. Let's think for a moment. Taking the average for test_1 will take an average of our column. And taking the average for test_2 also calculates an average of our column. So if we take all averages for each test, this should add a row at the bottom. So I'm going to add a new row to the DataFrame with the average score per test. We can use grades. loc for this, and let's look at our data again. Very well. Here is our new row. So now that our DataFrame looks like this, I can scroll up, select the cell here, and rerun it. And now we see colored bars not just per student, but also for the whole class.

Demo: Three Steps with Data Transform

As another example, let's say we want the progress of some students plotted, and I want a line for each student that shows their score on each test, so I can see how they progressed. In other words, I want the test names on the X-axis and the grades on the Y-axis. Now the problem with our grades DataFrame is that it's exactly the wrong way around. The test names are in the column headers, but I want them to be in the index. Now the solution is to transpose our grades. I'm going to make a new variable for this, grades_t, and to it I will assign the transposition of our DataFrame. Also we don't need the averages anymore, so I'm going to drop those. Now because we just did a transposition, a whole class is now a column, and average is a row. So you can see that reflected in the two statements here. Now looking at grades_t, we have the grades per student in the columns and test names in the index. So we're ready to plot. And again, once we prepare our data correctly, we can make plotting very simple. Now the one thing I don't like here is the X-axis. It shows too many ticks and no values. We can fix this, but I'll go into that later.

Demo: Three Steps, Another Example

Let's look at one final example. We've already seen this plot before. I read the water. csv file, and I'm plotting every column in a single plot. Now this is an example of not preparing the data for plotting. The names of the columns in the legend are unclear, the temperature is scaled so that it shows basically no useful info at all, the X-axis is also not very informative. Now, of course, a quick fix is to use subplots is true as we've seen in the previous demo as well. But there are still too many things here I don't like. Now the solution for making a pretty plot usually is to create a new DataFrame specifically for the plot we want. So let's say I want to plot just the water height. Let's start by converting the datetime column to actual datetime objects. We've also already seen this before. So this gives us a series of real datetimes, which we can then use as an index on our data by saying DataFrame. set_index. Also, I never want to plot the datetime data, so let's drop that, and our plot data now looks like this. So we have an index with datetimes, and we have both a height and a temp column, and now we can plot. But let's make sure to only plot the height. So this is already slightly better, but I don't like the datetime label at the bottom. It's actually taken from the name of our index as you can see when I scroll up. This here is the name of our index, and it came from the original datetime column. We can change this to something nicer by assigning to index. name. And let's also show exactly what the values on the Y-axis mean. To do this, first let's rename the Height column to something that looks a little nicer as a label in a plot like this, Height in cm. Now when you plot only a single column, you don't get a legend automatically, so we need to add that by saying legend is true. So let's rerun this cell and, of course, now I get an error because the column I'm trying to plot isn't called height anymore. So I have to use the new name here as well. Let's copy/paste it and rerunning. Now the only thing I still don't like here is the way that dates and times are shown on the X-axis, but you see the legend. You see that it has a nice name for the column, etc., etc. The moral of the story is this. If you make sure to prepare your DataFrame correctly, your plotting can usually be done in a single, simple statement.

Review: Three Steps for Simple Plots

For most simple plots, you can make your life easy by following these three steps. You start by preparing the script, then you prepare your data, and then all you have to do is choose and call the right plotting function. So to prepare the script, you use these three lines. You start by saying %matplotlib inline, and this first line is very specific for the Jupyter Notebook. It allows displaying our plots inside the Jupyter Notebook. So if you're not using Pandas from inside a Jupyter Notebook, you should use a different way to specify the backend from matplotlib. And I'll get back to that in the next module. The second line is import pandas as pd, and I'm going to assume that this is familiar to you. And on the third line, we import matplotlib. pyplot as plt. Now we haven't actually used this import yet, but it's very important to use anyway because, as we will soon see, this gives us access to some very important functions that we need when making slightly more advanced plots. Then there's the more complicated step of preparing our data for plotting. Now I cannot give you exact steps of how to do this because it depends on what you want to plot. But I can give you five hints. First of all, you have to check is the data I want to plot in my columns? If it's not in my columns, but in my rows, or maybe it's spread out over several columns in a way that doesn't make it easy to plot, well probably you should transform your DataFrame first. Then if your data is organized neatly into columns, you can select the columns you actually want to plot. Of course, similarly, you have to select the rows you want to plot. So maybe you want to drop some rows, or maybe you want to just make a selection of specific rows. Then in most cases, you also want to create a suitable index, like we've seen, for example, with creating an index of timestamps. Now in some cases, for example with scatter plots, you don't need to create a suitable index, and I'll show you examples of that as well. Finally, to get a pretty plot, you want to have nice labels in your plot, and that means you have to rename your columns and your index so that they have names that show up nicely. Now remember that your column names show up in your legend, and the name of your index shows up as the label on the X-axis. And then finally, step 3 is choose the right plotting function and call it. So the bar plot we see here for example has been created by the plot. bar function. Now as a side note, a different way of saying plot. bar is just calling df. plot, but adding an argument, kind, and passing that as a value the string bar. So the names of the different plots in Pandas, like pie and box and scatter, etc., etc., you can either use these names as plotting functions, saying plots. py and plot. box, etc., etc. Or you can just call df. plot and saying kind is and passing the name of the plot type as a string.

Both of these are equivalent, and you'll find examples of both of these online. And that brings us to the end of this module. We've seen a short introduction of the Pandas Plot API, we've seen how to create various kinds of plots, and most importantly, we've taken some time to see how to prepare our data for plotting. So let's move on to the next module in which we'll explore matplotlib, the Jupyter Notebook, and how these work together with the Pandas Plot API.

Navigating the API Jungle: Matplotlib, Pandas, and the Jupyter Notebook

Introduction: The Three Plotting APIs

Hi. Welcome to this module where I'll teach you how to make sense of all the different ways to plot your data. So far, I've focused on showing you the short and simple way to plotting with Pandas. The core of it is df. plot, the Pandas plot function. Now in this example, I'm not just calling plot. I'm setting the title of the plot as well. But to make things slightly confusing, there are multiple ways to do the same thing. For example, this also sets the title, but here we use something called pyplot. Or yet another way to set the title would be ax. set_title. And you can actually use all three of these approaches in the same script. Although whether it will work does depend on what you import and in which order the various lines show up in your code and more. So the theme of this module is to help you make sense of this. We basically have three different ways to work with plots and to become proficient in data visualization with Pandas or to understand the examples you find on the internet, you need to understand these approaches and how they relate to each other. So what are the plotting APIs that we will cover in this course? To start, there's matplotlib, which is probably the most important Python visualization library because other libraries are built on top of it. So the Pandas plot function, for example, just calls matplotlib. Now matplotlib offers two ways to work with graphics, pyplots, which is kind of old-fashioned, and it has some drawbacks, but it is basically still used by pretty much everyone all the time. The other API that matplotlib offers it the matplotlib object-oriented API, which gives us figures and axes objects. Now generally speaking the OO interface and pyplot both offer the same functionality. They just have a different way of writing it down. But generally speaking, the object-oriented API is newer and more flexible in the way you can use it. Then we've already seen the Pandas Plot API. This is one we've used so far, but it's actually just a nice and friendly frontend for matplotlib. So basically, it just calls matplotlib behind the screens. Now although the plot function makes plotting simple, it only offers part of matplotlib's functionality. So in practice when you need to tweak your plot, you will need to actually switch to one of the two matplotlib APIs. So in this module, I will show and compare these three different ways to create your plots, and we'll see how they relate. This isn't just so you can write working code yourself, but also to better understand the many examples you'll find online, like onstackoverflow. Now later in this course, I'm also going to cover two more very popular APIs for plotting with Pandas, both of which I will cover in later modules. First of all Seaborn, which specializes in statistical visualization and which is based on matplotlib, and it offers a variety of plots that are not very easy to make with matplotlib or Pandas at all. Then there's Bokeh, which is focused on interactive visuals for the web and which as the only library here is not based on matplotlib. Now let me give you a short overview of what to expect in this module. In a moment, I'll start with showing you how to use pyplot, and, of course, we'll also see how it relates to Pandas. Then I'll talk about matplotlib backends, which enable us to embed plots in the Jupyter Notebook, add interactivity, and more. From there, we will continue to another API, which is the matplotlib OO API. And again, we'll see how to combine it with our Pandas code. Then I'll do a short demo of JupyterLab, which is the future of the Jupyter Notebook, but combining it with matplotlib works slightly differently. Finally, I'll give you an overview of best practices for writing visualization code.

Demo: Introducing Pyplot

So let's start with taking a look at matplotlib. pyplot. This is, in a way, the most old-fashioned and low-level API. To give you a good understanding of how matplotlib works, we will start by using it in a regular Python script without using the Jupyter Notebook and without using Pandas. Here's a little demo script I wrote. I'm going to start with a barebones example and build on from that. The code I'm showing you right now is supposed to run from the command line. So in other words, we don't need a notebook. So if you want to work along, you can just create a text file in any editor you like and all it pyplot-demo. py. I'm assuming that you're running with Anaconda installed and that Pandas, numpy, and matplotlib are set up correctly. The code starts with now familiar imports of numpy and pandas, and then there's this one, import matplotlib. pyplot as plt. This is the pyplot package, and we can use it for visualization. For now, I'm just generating a simple graph. I take the range of numbers from -10 to 10 and raise them to the 3rd power with the np. power function. This gives me a list of numbers, and I can pass that to plt. plot. So right now, I'm actually making a plot without Pandas at all. You might think of this approach as being a little more low-level than using a DataFrame. We're not running inside a notebook, and this means that I have to say plt. show to make the plot show up. So when I run this program from the command line, we see this window pop up. This is the effect of the call to plt. show. Exactly what this window looks like depends on your operating system and installation. Usually, you will be able to save your plot by clicking on the icon here. In any case, this window has its own so-called event loop, which means that the Python scripts will now not finish until we close this window. But before I close it, please note that the X-axis isn't correct. It runs from 0 to 20, but we generated our data for a range from -10 to 10. So let me close this window, and let's fix this. Let's store the x range in a variable called x and then apply the power method and store the result of that in y. Then I can call plt. plot with both x and y, setting both the X-axis and the data to plot. This is one of the main differences with the DataFrame plot function, which tries to set the X-axis to a sensible set of values taken from the DataFrame index. Numpy and matplotlib are not aware of DataFrames and their index, so we have to set the X-axis ourselves. An important thing you should know about pyplot is that it's stateful. It keeps an internal state so that it knows what the current figure is, and the functions we call will affect that current figure. For example, I could save the plot to a file by saying plt. savefig and then passing a filename. And this will save the plots that we just created by the call to plt. plot just before. Or maybe I want to change the title of this plot, but I cannot add the title as an argument to plt. plot. Instead, I do another function call, and this will set the title of the current figure. Or I can add a second call to plots, which will plot another line on the same axis. So here I create a new variable, y2, and to make it a little more interesting, I add a nice little polynomial, 13 * x2 - y - 1000. So plotting this will add a second line to the plot we already have. I can also very simply add a line style with a simple string syntax. So let me just make the second line red and the first line green. Good. Running this again, we have two lines now, a title and the correct values on the X-axis. So going back, we've seen so far that we have the plt package, which lets us make plots in a slightly different way than Pandas. If we do multiple calls to plots, it will remember what the current plot is, and we have to call plt. show, which will show the plot in a pop-up window. Now let's add one more thing. Let's say I want to make another plot after this one. The first thing I will have to do is to say plt. figure. This will create a new figure, and any following function calls to plt will affect this new figure. And this time, let's use Pandas. I'll read the familiar water. csv file, and just for the demo, let's not use the pandas. plot function, but plot it again with matplotlib. Now the call to plt. show I will leave at the bottom of the script because it will show both figures we have now created. Running the script, we again see a pop-up window, and in this case I see my new figure. And when I close it, we see the other figure as well. And the reason we saw the second figure first is because my operating system puts the newest window on top so that it covers the previous one. Of course, this might vary if you use a different system. Good. So the point of this demo was to show you that matplotlib is a complete visualization library in and of itself and that we can use it independently of Pandas and the Jupyter Notebook. Now let's move on and see how to apply this knowledge when we do have a notebook.

Demo: Jupyter Notebook and Matplotlib Backends

Now that we have a little more understanding of what matplotlib is and how it works, let's investigate how it can work together with the Jupyter Notebook. To make this work, we have to select a matplotlib backend, and we can do that through a special command. We'll see two examples of this, using the inline backend and using the notebook backend. So let's try to use the code from our script in a Jupyter Notebook. I cope/pasted everything here into this notebook cell, and when I try to run it, it doesn't actually work. It just shows some text. Now when we're working in a Jupyter Notebook, we want the plots to appear in the notebook itself, and we have to configure matplotlib to do this by setting the so-called backend. We can do this by saying %matplotlib and then selecting the backend. The best practice is to do this before you import matplotlib. So far, we've seen the inline backend, but what other values can we select here? Well, let me create another cell. And to see what backends there are, we can saw matplotlib -l, and this shows me a list of possible values. This list might be slightly different on your system, but generally speaking, most of these will show a pop-up window for each figure we create, and whether they will work depends on your system. For example, I could select osx, but that is meant to run on macOS, and I am on Linux, so that won't work. A backend that I know will work on my machine is qt, but actually most other backends will not work because I don't have the correct libraries installed. But let's focus on something that does work. We've already seen the inline backend, which will include the plots in the notebook itself. So running it, we see our plots embedded in the notebook. We can now remove the call to plt. show because the inline backend takes care of this for us. If I also change the last call to plot into a call to Pandas plots, this will automatically create a new figure and so we don't need to create a new figure either. So now we're running, get the same results, but with less code. Now an important subtlety is this. I'm calling plot on a DataFrame here, and I don't need to create a figure. But when I want to plot only the height column, this is calling plot not on a DataFrame, but on a series, and this will plot to the active figure. So when I run this, it adds the water height to the first figure. So in this case, I do need to call plt. figure to create a new, separate figure for my water height. Good. Now there's another backend you should know about, and it's called notebook. When I change this and rerun the cell, it doesn't seem to work at all. I just get a lot of empty space, and this is because switching backends in a script is not really supported. To make this work, I can simply restart the kernel by clicking Kernel at the top here and selecting Restart. And then let's rerun my cell, and we now see the same plots, but they are slightly larger so let me zoom out a little bit. And here we see some buttons we can use to interact with these plots. So this is a feature of the notebook backend. There's a button here to save the figure to a file and other buttons to zoom and pan. For example, if I select this button, I can select the part of my graph where my lines cross. In this way, I can alter the displayed data that will be shown in the notebook. By the way, I can only interact with a single image at a time. So if I want to interact with the second image, I first have to stop the interactivity here with the Close button, and we end up with a static image just like we had with the inline backend. And now I can, for example, resize the second image. And when I'm done, I can close this as well. Personally, I usually prefer the inline backend because it tends to work slightly faster, but the notebook backend certainly has its advantages.

Demo: The Matplotlib Object-oriented API

So we've learned about pyplot and about backends, and now it's time to talk about the second way to work with plots, which is the matplotlib object-oriented API. Now this API gives us the same functionality as pyplot, but in a more elegant and flexible way. And in this demo, we'll rewrite our simple program to use this API instead. I'm still working with the same code as before here, but I've split up the code into multiple cells so that I can run them separately. Now the first cell simply does my imports, and the second cell does the first plot. Now the third cell shows the second plot. Now suppose I want to do something to the second plot. Let's set the title of the plot, and I'm going to do this in a new cell. And let's do something else that you wouldn't usually do with Pandas, but with matplotlib instead. I'm adding a horizontal line to the plot. Now both of these statements are using the pyplot API, which affects the active figure. That's the figure from the previous cell. So when we run this, the plot above this cell changes. But the thing with notebooks is that you can run your cell in any order. So let's say I change something in my first cell. Just for fun, I'll change the plotting style to show only the data points and not connect them with a line. And when I run it, we see that the second figure is still my active plot, and that's where my output ends up. So this can become quite confusing. The pyplot interface with its active figure is not a very good idea to use in a notebook. Instead, we prefer to use the matplotlib object-oriented interface, which works slightly differently. The preferred way of plotting is like this. We start by calling plt. subplots. Although this is a call to a function from pyplot, it returns two objects, a figure and an axis. The figure is an object that represents the entire picture we're creating, and the axis object represents a single plot. Right now, we're only plotting to a single axis in a single figure. Now instead of depending on whatever plot is active, we can explicitly refer to the specific axis. So instead of plt. plot, I can now say ax. plot. And I can also change the call to title, although in this case we have to say ax. set_title. And if we want to save a figure, we have to use the figure object instead of the axis. So now rerunning, we now see that this cell plots into its own new figure again. Now I can add arguments to subplots to create multiple subplots inside the figure. I can pass it the number of plots vertically and horizontally. If I add the arguments 1 and 2, this will divide my figure into 2 plots horizontally. Vertically speaking, there will be only a single row of plots. If I do it this way, the variable ax will now contain a list of axis objects, each representing one of the subplots. So now if I want to plot to the first subplot, I have to select the first axis object by adding an index. And I'll add that to set_title too. And now that we're at it, let's add a title to the second subplot as well, and let's plot our second equation to the second axis. Now rerunning, we see two subplots here, and this is because we explicitly created two subplots with the call to subplots. And in my code, it's completely clear in every line which subplot I'm referring to. So if we do it this way, the result will be the same regardless of the order in which I run my cells. Now looking at the second cell, we're using the Pandas plot function here, let's see how to make this work with the object-oriented API as well. First, let's remove the call to plt. figure. Now remember that when I plot a whole DataFrame, this will create a new plot by default. So in this case, we don't need to create the new figure. But if we plot a series, this doesn't work the same way. So now we do need to create a new figure, and we do this in the same way as above, but now we pass the Axes object to the plot function because right now we're not calling plot on an axis, but we're calling plot on the DataFrame. Now we can tell it explicitly to which plots it has to send the output. Now the name of the argument is ax, so we say ax is ax. And then when I run it, we see a new figure with the data plotted. Now the code in the cell after this we can now change so that it uses ax. set_title, and we can draw the horizontal line on the axis object as well. And if we run this, again we see the plot above change. Now, of course, this code still very much depends on the order in which I run my cells because both cells above create a variable X that points to a different plot. To fix this, you might change the variable names so that each plot has its own unique name. But in fact, that would still cause problems. You see, referring back to an axis from a previous cell will not even work with the inline backend, and this is because the inline backend closes a plot immediately, which means that you can refer back to the plot, but it will not be rerendered. So if you would rerun this code with the inline backend instead of the notebook backend we have right now, you would see that this last cell here has no effect at all. So what's the solution? Well actually the best practice is to put all code that sets up a plot in a single cell, and this prevents all problems with the order in which you run cells, as well as any problems with the inline backend. We also prefer the object-oriented way of doing things because it makes more explicit what we are operating on, so to which plot are we sending our output, which figure are we are trying to change, etc., etc. So I'm moving the code from this last cell into the previous cell, and let's just remove these last cells. And now I can rerun this cell, and all the code is in a single cell as it should be. Very well so far for my introduction of the three different ways to work with matplotlib. For the final demo, I want to take a short look at JupyterLab.

Demo: JupyterLab and Matplotlib

You are probably aware that besides the Jupyter Notebook, there's a different, but similar project called JupyterLab. And actually, JupyterLab is meant to be the future of the Jupyter Notebook. But at the moment, it's still in development, and there are quite some issues, one of which is that the performance isn't always great yet. Nevertheless, we're going to take a short look at it and, of course, we'll talk about how to use it for plotting. It turns out that we need a different backend called widget, and there are some installation needed to get it working. We can start JupyterLab from the Anaconda Navigator, just as you would start the Jupyter Notebook. Or if you prefer the command line, you can start it by typing jupyter-lab, whichever you like best. In your browser, it looks like this. Now JupyterLab is really intended as the future of the notebook, and it contains lots of exciting new functionality. For example, we can start a terminal in a separate tab or a text editor, or I can just open one of my Python files in a text editor. But in a regular notebook, like the one we already created, we can also do lots of new things, like for example drag and dropping cells to reorder them. And there's a lot of new features, but this course is not about the features of JupyterLab. It's about visualization. And the thing is, JupyterLab works fine with the inline backend. But with the notebook backend, it doesn't. Now to show you, I have to restart the kernel of course, and then saying Run All Cells gives me an error message about JavaScript. Now JupyterLab is still very much in development, so the situation here changes a lot. But currently, the backend you want to use with JupyterLab is either inline or something called widget, which has basically the same functionality as the notebook backend, but to make it work, you will have to install a library called jupyter-matplotlib, which you can find here on GitHub. If you scroll down on this page, here are the installation instructions. So if you prefer to be on the cutting-edge and work with JupyterLab, make sure to follow these steps and use the widget backend instead of the notebook backend. Now, of course, I already did these installation instructions, and let me just show you that if I say widget here and restart my kernel and run all cells, I actually get my output here, including all the interactivity. Now we've learned a lot so far. Let's review.

Review

We started by looking at pyplots, which is usually imported as plt. Just about all functions we use with plt are stateful. Pyplot keeps track of what has happened before, and the functions we call will affect the currently active plot. Although this makes for short code, it turns out not to be the greatest way to write code for a notebook. Anyway with plt, we usually start by creating a figure with the plt. figure function, after which we can call plot to create some visuals. We can call any number of extra functions to customize the plot as well, like the title function you see here. Now when we're not working inside a notebook, you usually want the plots to be shown in a pop-up, and for that you call plt. show. An alternative way to work with matplotlib is the matplotlib object-oriented API. It offers the same functionality as pyplot, but in an arguably more elegant way. In this case, the usual way to get started is to call plt. subplots. This will create a Figure object and an Axis object and return these as well. You will want to store both of these in a variable so you can use these objects in the lines that follow. Now we can call subplots with the number of plots we want vertically and horizontally. So in the example here, I'm creating a figure with six subplots, 2 rows and 3 columns. A lot of the time, you only want single plots, and in that case, you can leave out both arguments altogether. After this line, we can call methods on the axis that were returned. In our example, we have six subplots, so the x variable here will contain a list of six Axis objects. I select the first one by saying ax 0 and then call plot on it. Of course, I can also customize my plot, for example, by calling set_title in it to set the title. Now one of the main advantages of pyplot is that here I can explicitly reference exactly the object I want to change. I don't depend on which plot happens to be active right now, and that's nice in case I'm working in a notebook. Then, of course, there's the Pandas plot function. This is the friendly Pandas frontend for all the matplotlib functions we just saw. Simply calling df. plot on a DataFrame will create a new figure and plot the DataFrame to it, plotting all columns, setting up the axis and the legend, etc., etc. But if you call plot on a series, it will not create a new plot. Instead, it will plot to the currently active axis or subplot. Now the effect of this depends on several things. Did you just make a different plot and is it still active? Then this will add to that plot. But we've also seen that the inline backend closes a plot automatically, so it will not be active when you run the next cell of your notebook. The notebook backend doesn't do that, so the effect of your code might differ between backends. So in general, it's best to call subplots, and then when plotting a series, explicitly pass the axis you want to plot to. This way you won't get any unexpected surprises. Now for the short overview of the backends we've seen. The most basic backend for using a notebook is inline, which just puts the output of the plot command into the notebook without any interactivity. You should realize that this calls plt. show for us, so you don't have to call it yourself. The inline backend also closes the figure right after you run your cell, so you cannot write code to change the plot in the next cell. You need to put all your code for a plot in a single cell. Then there's the notebook backend. It also calls plt. show, but it does more. It adds interactivity, a button to save the figure, buttons for zooming and panning, resizing the figure, etc. Now this only supports interaction with a single figure at a time. So make sure to press the Close button to end the interaction when you're done. We also saw that for JupyterLab, there's a different backend that offers the same interactivity as the notebook backend. To make this work, you will have to install the jupyter-matplotlib library and follow the installation instructions from their GitHub page.

Best Practices

Before we move onto the next module, let me give you some pointers on how to write code that works well in any notebook. Of course, many examples you find online will not follow this advice, but you should be able to rewrite any code you find to match these ideas. So here's how you should write your visualization code in my opinion. To start, you set the backend for matplotlib. Make sure you do this before you actually import matplotlib. When you do import matplotlib, you do it as shown here, import matplotlib. pyplot as plt. Then before you create any actual output, you create a figure and an axis by calling plt. subplots. So now you're ready to start plotting some data. If you set up your DataFrame correctly, you can usually simply call df. plot, and it will create a plot with several things already set up, like the X-axis taken from the index, etc. And here it's a good idea to actually specify the axis object we want to plot to. This makes sure we never accidentally plot to an axis we created before in another cell or anything like that. In the call to plot, we can also do some customization, like setting the title or the line color and several other things. But whenever you want to do anything more advanced, and there's actually quite some stuff that matplotlib can do, but that's not available through df. plot, so whenever you want to do something special, use the matplotlib object-oriented API. In other words, call a method on the Axis object or on the figure. This will give you access to the full functionality of matplotlib. Now in my opinion, you should not use pyplot at all. So the only thing we use the plt module for is to call plt. subplots. Anything else we do through the Axis and Figure objects. And finally, to make sure that your code works well with the inline backend, whenever you're creating a plot, it's good practice to group all the code for that specific plot together in a single notebook cell. And that brings us to the end of this module. I started by introducing pyplots, the old-fashioned and slightly quirky way to create plots with matplotlib. We learned how to use matplotlib in the Jupyter Notebook by setting a backend, and then we improved our code by using the object-oriented API instead of pyplots. I gave a short demo of JupyterLab and showed you how to set it up for plotting. And finally, I gave you some best practices for writing great visualization code. So by now, you have a good overview of the various ways to create plots and how they work. You should be able to understand most examples you find online. Let's move on to the next module and learn about some advanced plotting techniques.

Creating Advanced Plots with Pandas

Introduction

Hi, and welcome in this module where we'll learn a lot about all the ways in which we can create and customize plots with Pandas. In this module, we're going to do a lot of demos. Matplotlib offers a huge amount of functionality, and it's impossible to cover all of it. But I'll do my best to give you a good overview. We'll start by looking at some scenarios where we have a figure and we want to plot more than just a single series of values on it. There's two common cases that you need to be aware of, creating multiple subplots and plotting to them and sharing axes between them. And then there's a case where you have a bar or an area plot and you want to stack the various data series. Next, we'll look into styling. We'll see the predefined themes that come with matplotlib, as well as how to set line styles, and we'll take a closer look at scatter plots, which can do some special things that other plots cannot. A very important thing for every plot is that you set up the axes correctly so that they show the right information. So we'll look at that including the position of ticks, setting the right labels, controlling the range of axes, and setting the legends. And finally, the coolest thing of all, I'm going to introduce you to an awesome library that will make your plots interactive with almost no code at all. So let's go ahead and start with the first demo.

Demo: Subplots and Shared Axes

In the first demo, we'll take a look at creating subplots and sharing axes between those subplots. Here we are again in a new Jupyter Notebook, and my first cell contains the familiar code where I do my imports and set up the backend. And in the second cell, I read weather. csv. And as we can see in the first few lines here, it contains hourly measurements of temperature and pressure. Now what I want to do is to compare the progression of both temperature and pressure on two different days. So I want to take the data for two different days of the year and see how they differ. Here I start by selecting the month, which is either January or July. And for each month, I select the first day. This gives us all data for only January 1 and July 1. I then drop the DAY column because we won't need it. And now I do a little trick. I use the pivot method to transform my data. As you can see down here, this results in the time of day being in the index. The columns now have a multilevel index giving the temperature for each month and the pressure per month. Now if I just do a days. plot here, this isn't very helpful. We just see two lines at the top and two at the bottom. And this is because the values differ so very much. Pressures are in the tens of thousands, and temperatures are much, much lower than that. So we cannot easily put them both in the same figure. Let's instead try to use subplots. Now let me resize this. Of course I'm at a very low resolution so some of our plots are really small. Now this looks a little bit better. Let's disregard the legend for now and just look at the Y-axis here. Each plot has its own values, making them very hard to compare value-wise. One goes from 0 to 5, the other from 15 to 17, and the pressure axes also have completely different values. So this makes comparing them very difficult. Now what I want is to have the temperatures side by side and the pressure side by side. I can do this by adding an argument, layout, to the call to plots. And I pass a tuple with the number of rows and columns I want. So here I'm saying give me 2 rows of 2 plots each. And this already looks a bit better. But to make it even nicer, we can specify that we want the range of the Y-axis to be the same so that we can compare values. To do this, I say sharey is true. But now we have a familiar problem. Everything has a shared Y -xis, and we only see lines at the top and bottom. So actually we only want to share the Y-axis for the temps and for the pressures, but not for everything. And by the way, note that the X- axis in this case is already shared. This is what Pandas does by default, so there's no need to change that. The way to fix this is to create subplots explicitly through pyplot. Here again, I create 2 rows of 2 plots. Now when I create my subplots this way and I want to share the axis, I have to do this immediately in the call to subplots. But the nice thing is that now I can say that I want to share the Y-axis per row. So I get a shared Y-axis for the row with the temperatures and another one for pressures. And I'm also setting the X-axis as shared per column. Now what this returns is a list containing a list of axes per row. Let me show you. Running this, we see the empty subplots followed by the value of X. And we see that this is a list containing one list for the first row with two axes and another list with two axes for the second row. So what I'm going to do is I'm going to plot temperatures to the first row and pressures to the second row. And to do this, I get the temperature data by saying days TEMP. And because we have a multilevel index, this contains in itself two columns, one for the month of January and one for July. Calling plots will plot these two columns of data, and because we say subplots is true, it will plot to two subplots. Now as a second argument here, I pass the first element of my ax variable, which is a list with two subplots. In other words, we're plotting each month to one of the two subplots in the first row. On the other line with days PRESSURE, it works the same way, only we're plotting to the second row of subplots. So when I run this, this is a much better result. We now see the temperature on the first row and pressures on the second. The first month is in the first column, and the second month is in the second column. Also, the Y-axis is now shared for temperatures and for pressures, but not for all axes. One of the things we can now finally see is, for example, that our day in January was much colder, but the pressure was a bit higher. Also scrolling down, we have some ugly output text here, and this is actually the output of this final line here. Now in the past, I've said that you could store the return value of the function in a variable to suppress the output. But in this case, of course, I can store the return value, which is the axis we plot to. But we already have a reference to the axis. Now instead of saving the return value, what I can also do is add a semicolon at the end of the line. And rerunning, we now see no more text at the bottom here. Nice. One other thing I don't like is that every time when the figure is created, because I have such a small resolution, the text of the labels doesn't fit in the figure. It goes over the edges. And we can fix it by asking the figure to tighten its layout. So this fixes the spacing of the figure so that all the text of the labels fits inside the figure. Now again, I'm on a very small resolution, so all these subplots here don't look very beautiful. But if I resize them a bit, you'll see that it looks a bit better. Now finally, let's make some other things a little bit prettier. I'm going to remove the legends. Good. And let me copy some code in here. On every line here, I select a specific subplot and set some label. For the upper-left subplot, I set the ylabel to Temperature and the title to January. For the upper-right one, I set only the title to July, etc., etc. Now I have to make sure that tight_layout is my last call here because it has to fix all the spacing after applying these labels. Very well. So that's subplots for you. Let's move on to the next demo.

Demo: Stacking - Preparing the DataFrame

Let's take a closer look at bar plots and area plots. They can do something special called stacking. So let's take a look at the sales figures of some fictitious company in 2017. I want to see sales per month and total revenue, and we will see some different ways to plot this. So here we already have some code. I'm setting up the script as usual, and then I'm reading my CSV. I've added an argument, parse_dates, that tells the importer that the Date column contains dates. So when I call info, we can see that the Date column contains dates. And we also have an Amount and a Product column. Let's also look at the first couple of rows. We see that the sales are recorded per products. Now I want to plot the sales by month, so I'm going to start by extracting the month from the Date column, and we can do that by saying df Date. dt, which gives us access to the various properties of a datetime, and we ask for the month. And let's store this in a new column, Month. And let's take another look at our data. So now we have a column with month numbers. Now I want to plot the sales per month and per product. And to do that, I'll start by grouping the data per month and per products. Because I'm not interested in the Date column anymore, I'm going to tell the groupby to just look at the amount for each sale. And let's take the sum, and I'm going to store the group data in a new variable. Let's call it g. Okay, so this gives us a single series with a multilevel index of months and products. Now let's think for a moment. I want to make a plot with months on the X-axis, so we see the progression of sales through the year. So the X-axis should read 1 to 12 from left to right showing the progression of time. I then want to plot sales per product, so it would be nice to have a column with sales numbers per product. In other words, I want to move the products from the index into three columns, and there's an operation for this, and it's called unstack. Calling this on our DataFrame, we now have exactly the format I like, months as the index and products in the columns. But I'm actually not really interested in the sales themselves. I'm interested in the total revenue per product, and we can calculate that by taking the cumulative sum, by saying g. cumsum, and I'll store this in a new variable, revenue. So this shows my revenue as it grows per month through the year. What I don't like is that there are empty cells here because the NaN cells are skipped by the cumsum function. Let's replace these NaN values with 0s so that the cumsum will calculate values for those cells as well. Very well. That looks better.

Demo: Area Plots with and Without Stacking

Now one way to plot this data is to use an area plot. And this shows the various sales per month stacked on top of each other. So we can see that the total revenue climbs to almost 200, 000 euros, and this is nice. It gives a general impression of how the revenue has grown and how much each product has contributed to that. By the way, the text of the labels is clipped. It goes over the edge of the figure and, again, I can fix this by calling tight_layout. Of course, this means we first have to have a figure, so let's add that. And of course, now I have to reference the axis I just created in my call to plots. And now I can call tight_layout on the figure. So what an area plot does is it puts a line for each column, but adds them on top of each other and then fills in the regions between the lines. In this way, we get an idea of the total growth and how much each column contributes to the total. But if we want a better picture of the way the revenue grows per product, I can make the area plot without stacking by adding stacked is false to the function column. And let's also add a grid to make it easier to read exact amounts from the plots. So this plots to areas for each product over each other. And as you can see, these areas are made transparent so you can see each line even though they overlap. So now we can see that sales for product A were over 80, 000 euros and for product B around 60, 000, etc. Also, the revenue for product A grows steadily, while product B shows bumps every few months. Now personally, I like this plot better than the stacked one. But we cannot see the total revenue anymore, so let's add that as a line to the plot. I can calculate the total revenue by taking my original data, grouping by month, and taking the sum per month overall sales. And then for that, I say cumsum, which will calculate total revenue for everything. To plot this, I say total_revenue. plots, and I pass it the axis that we set up earlier because, of course, I wanted this line to end up in the same graph. And let's give it a line style as well. I'll make it red with dashes. We can use line styles from matplotlib here like we saw in the previous module, and I'll talk more about that in a moment. The result here is not what I was expecting. Apparently, matplotlib doesn't resize the Y-axis when I do this, and part of the red line falls out of the plot. To fix this, I can simply plot the total revenue first. And now, the Y-axis does get resized, and we see all the data in one plot. Another way to fix this is to explicitly set the range of the Y-axis, and we'll see that later. So now we have all the information I want in a single plot. And although the legend doesn't show information for the red line, we can fix that by adding legend is true to the first call to plot, and I want the label for this line in the legend to be All for All products. Good, that's better.

Demo: Bar Plots with and Without Stacking

Now let me show you another example of stacking. You see, not only area charts can be stacked. Bar charts can do this too. We're going to read a file with results from the Olympic Games. And to prepare our data, let me just copy some other code here, and I'm grouping our data by nationality and selecting only the number of medals for each country, and I'm summing that. I then sort the countries by most medals won in descending order, and from that I select the first 10 countries, and this gives me the following results. I can plot this DataFrame as a bar chart by saying top. plot. bar. And I'm specifying the colors for the columns by saying the first column should be colored gold and the second silver. Unfortunately, bronze is not a color that matplotlib understands, so I'll just take brown instead. And the result here shows me a regular bar chart, although the text for the X label falls out of the figure. Let me fix this quickly by saying fig. tight_layout. Now to give a different overview that shows the total number of medals per country, we can add an argument, stacked is true, to the plotting function. And there you have it. One difference between area and bar plots is that an area plot is stacked by default, while a bar plot isn't.

Review: Subplots and Stacking

Earlier in this course, I told you that it's a good idea to save the output of a call to plot in a variable. This has two reasons, to make the axis object available for later customization and to suppress an ugly line of output beneath the plots. Now that we know how to handle axis in more detail, we don't always need to save the Axis object after a call to plot because we usually already have the variable holding that reference. So instead, you can use a semicolon at the end of the line, and this is a Jupyter-specific thing that will also suppress the output. We saw that we can automatically create subplots by passing the argument subplots is true, and this will make a subplot for every column in the DataFrame. And you can set the layout of subplots by adding the number of rows and columns. So here I set it to 2 rows and 2 columns. Optionally, you can add arguments sharex is true or sharey is true to share the axis between all figures. As we saw in the demo, this is not always a good idea. In the plots shown on the right, the values on the Y-axis vary so hugely that when you share the Y-axis between all subplots, you will see nothing more than a line at the very bottom and one at the very top, and you'll lose all useful information. The X-axis is usually already shared when you use Pandas because usually all series you're plotting have the same index. Alternatively, you can set up the subplots when you call plt. subplots. Here again, you can set the number of rows and columns, as well as any axis you want to share. So here I share the Y-axis per row as you can see on the right side, the top row has shared Y-axis and the bottom row as well, and the X-axis is shared per column here. When you do this, the X variable will hold a list, and that list holds another level of lists, one per row. And in those lists, you will find the actual Axis objects for each subplot. So now to plot to the top-right plot, we do this. We get row 0 from the x variable and then get plot 1. This is the second plot from the first row, in other words the top-right plot. If you need even more control over your subplots, like giving them different sizes and positions, you should check out GridSpec. I added a link here to the documentation so you can check this out. We saw that an area plot is kind of like a line plot, but for every line it fills the region below it. By default, the plot is stacked, and that means that each region is added on top of the previous one. So the plot on the right here, which is the default behavior, shows the total revenue for all sales, and it goes up to about 200, 000. The plot on the left is unstacked. It just shows the actual values for each line. And in this case, one line can overlap each other, and that means that Pandas automatically adds an alpha value here to make sure the regions are transparent, and we can still see each region. Bar plots, on the other hand, are not stacked by default. So if you want this, you need to pass stacked is true.

Demo: Matplotlib Themes

Now let's move onto the next part of this module and learn how to make our plots look a little better. Let's talk for a moment about making our plots look good. Matplotlib gives us a lot of possibilities to change the way our plots look. We can change colors, fonts, and spacing, and we've only seen a tiny part of the possibilities. Actually, the functionality of matplotlib is so huge that it's basically impossible to explain it all in this course. But I can give you some pointers. First of all, matplotlib comes with a number of very nice themes installed by default. We can ask for a list of them by saying plt. style. available. So this is a list of available styles, and I'm going to try out some of these. And let's start with ggplot, which I think looks quite good. Opening one of the notebooks from previous demos, here I'm calling plt. style. use, and I pass the name of the style I want to us, in this case ggplot. I'll just rerun the entire notebook, and we see that this gives our plots a completely different style with a gray background, nicer fonts for labels, and a nice color map for the data. Or maybe you like the FiveThirtyEight style taken from the well-known statistics blog, FiveThirtyEight, or one of the many Seaborn styles, like Seaborn Pastel. So just take some time to play with these different styles and see what you like. To properly see what each style looks like, make sure to restart and rerun when you apply a style. If you don't do this, you will end up looking at a mix of two different style sets, which, of course, can be interesting. But it's not the way they are supposed to look.

Demo: Styling - Line Styles, Colors, and More

Let's take a short look at some other options we have to change the way our plots look. I'm going to read the sales figure CSV we have seen before in a demo about bar charts, and I'm going to do some of the same transformations. So first I extract the month number from the date. Next, I group by month and product, sum the Amount column, and on the result, I call unstack, and then I store everything in a variable called sales. Calling sales. head, we now have the monthly sales per product in three separate columns. Good. Let's put these figures as lines. Now in this case, I want to set up a style for plotting, but I don't want to apply this to the entire script, just to this one single plot. I can set up a so-called context, which applies to style only for the plotting done within a Python block, like this. So here I'm creating a style context with the style seaborn-dark. And inside there, let's create a plot and plot our data. I'm setting a size for this figure, and we can do this with the argument figsize, which accepts a tuple with width and the height in inches. So I'm setting this to 6 x 3 inches. So that's nice. You can control the exact size of your plot this way. And I'm also passing a parameter, style, which allows me to set the line styles for each column. I pass a dictionary mapping each column name to a style. So the style for column a is given by this string here, and this is a special matplotlib shorthand for specifying line styles. The colon means a dotted line, and the s means squares. And we see that the line is indeed dotted with squares that marked the values. And the second line is dashed and has triangles as markers, and the third has both dashes and dots and uses circles. For now, there's something wrong with my plots. I see some line segments missing from the blue line. And for the orange product b, there's no line at all. And the reason is that our data contains empty or null values. And Pandas will not plot a line when it encounters a null value. I could try to fill the missing values like this, which will fill each missing value with the value before it. But the problem here is when I run this, this gives the impression that there were sales in those months and that they matched the month before, and that's not the case. So here, I'm adding data points that don't actually exist. So instead, I'm going to drop the missing values from each column in turn. But this means I also need to plot each column in turn. And this means I'm going to write a loop. I'm going to start by moving the styles dictionary to the top of the cell. Then I can loop over this and plot each column in turn, and I will simply pass the styles dictionary here to set the style. And now the trick is that I will drop the empty value from each column before plotting each column. And the result, as you can see, is now correct. So apart from setting the general style and specific line styles, there's a lot of other things we can do. For example, let me just set two different titles. We can set a supertitle on the figure with suptitle and set a normal title for the plot with set_title. But in both cases, I can pass CSS-like styling. So here I'm setting the supertitle as bold and the subtitle as italic, and I'm making the subtitle slightly smaller as well. Then, let me add a label to the Y-axis, as well as a grid. The X-axis label doesn't fit inside the figure, so we have to call fig. tight_layout. Now to give the supertitle a little bit more space, I have to resize the figure. Again, this is because I'm currently on a very small resolution. So as you can see, there's a lot of options to actually style our figure. Now looking at our code again, there's one thing I don't like, and that's that we're setting the figure size here in the call to plot because this gets called three times. Instead, I can also set it at the top where we create the figure. So this has the same effect as setting the figure size in the call to plot. So you can do many more things like controlling spacing and margins around the plots and all the labels and the titles. And more importantly for this plot, we should fix the fact that the legend is missing. And I would also like to see the name of the months on the X-axis. I'll get back to that in a moment when I'll have a demo dedicated to axes and legends. For now, let's go on and talk about scatter plots.

Demo: Scatter Plots

One type of plot we didn't really look at yet is the scatter plot. It has some peculiarities you should know about, especially where it comes to colors and sizes. So there's a kind of plot we didn't really discuss yet, and that's scatter plots. Here I'm reading my athletes. csv file to show you some examples. Let's say I want to compare height and weight for athletes in various sports. Now in this case, instead of transforming my DataFrame, I'm going to break it up into four different DataFrames. So first I separate the men and women into two different variables, f and m, and then I'm going to select my sports. Let's say I want to look at tennis players and gymnasts. My idea is to create a scatter plot where I plot the height and weight of each person against each other, like this. So the thing with a scatter plot is that you usually plot two columns of your data against each other. And this means that the X-axis is not generated from your index, but that you need to specify which column will be on the X-axis and which one will be on the Y-axis. So this is the result. Here we see height and weight of female gymnasts. But now I want to add the other three groups into the same plots. First, let's create a figure and an axis. And then I'm going to make a list with properties for our plots. So here I have a list, and that list contains a tuple for every plot I want to make. Each tuple has the variable holding the data, so for example the male tennis players followed by the color I want to put them in and then the label that needs to be shown in the legend. So the male tennis players will be blue. The female gymnasts will be orange, etc. And I store all of this in a variable called plots. And now that I have this, I can set up a for loop that loops over this list, and from each tuple, it retrieves the data, the color, and the label, and then it makes a scatter plot. Of course, we pass the axis to plot 2, the X and Y columns to plot, which are weight and height, and I'm also passing an alpha value, which is the amount of transparence, and this is nice because we will be able to interpret multiple values plotted on top of each other. And finally, we pass the color to plot in, which again is taken from the list above. So this will be red, blue, orange, and green. And then in a similar way, we pass the label. So this will become the text of our legend. And running this, we see some nice results. Again, because of my tiny resolution, I'm going to have to resize this a bit. So here we see that the four groups have very distinct characteristics in terms of height and weight. The male tennis players are by far the tallest and heaviest; whereas the female tennis players are slightly taller than the male gymnasts, but not heavier. Because of the alpha setting, we can see where multiple points are plotted at the same location. For example, there are multiple tennis players with the same height and weight. We can also see that the legend contains the text we set as the label in the function call. going back to my code, let's fix the labels on this plot by calling set_xlabel, set_ylabel, and set_title. And here I've added a semicolon to the less line because this function returns the label, and if we don't add a semicolon, we'll see that as a line of text below the plots. And again, this semicolon will suppress that output. So for a second example, let me just show you one other nice thing we can do with the scatter plot. Let me plot only the male gymnasts for now. If I add a new argument here, c, and I pass in the name of a column, let's say gold, which holds the amount of gold medals an athlete has won, running this, the color of each data point is now determined by the amount of gold medals. But this is grayscale, and I think it's ugly. Fortunately, we can pass the name of a color map, and one of those is called plasma. By the way, if you want to know the color maps that come with matplotlib, simply Google for matplotlib color map, and one of the top links will take you to the documentation page that shows all the available color maps. Now rerunning this, we can now see who the winners of the gold medals are. Now let's just go figure the tight layout so we can read the labels. And it seems to me that if you are under 60 kg and are less than 1 m 72, you have a larger chance of winning a medal in tennis. So as you can see, there's quite a lot of nice possibilities with scatter plots. But in general, if you want to make a scatter plot, I find that Seaborn, another library which I will discuss in the next module, is much more powerful and friendly to work with. So in case you want to do anything more intricate than just a basic scatter plot, my advice is to look into Seaborn and use that instead.

Review: Styling

Matplotlib comes with a number of predefined styles, which can make your plots look better with a single line of code. The variable plt. style. available holds the list of available styles, so you can use this to play with and see which style you prefer. You can call plt. style. use with the name of a style, and from there on that style will be applied to all the plots in the scripts. Or you can call plt. style. context and use with statements to make the style apply only to a specific block of Python code. We can give the lines in our plots a variety of different styles, making them dotted or dashed and adding all kinds of shapes for the markers for our values. You can do this by passing the style of the lines for your columns as a dictionary to the plot function. If you're plotting only a single line, you can pass the style directly to style. Instead of going over all the different markers and styles here, let me give you two links that explain all the possibilities. The first link shows the various line styles and how to apply them, and this link lists all the marker shapes that matplotlib supports. You can set the size for a figure using the argument figsize and passing the width and height of the figure. This size is measured in inches. If you use subplots, it might be more elegant actually to set the figure size in the call to subplots, which works exactly the same way. Sometimes, especially when your labels have a lot of text, this text will flow outside of the figure. Matplotlib can fix this for you automatically if you call fig. tight_layout. Make sure to call this after doing all the layout for the figure. Matplotlib also lets us set default values for lots of things. As an example here, you see how we can set a default figure size, a default font-size for the title, and how to enable the tight_layout by default for all figures. The list of things you can configure is huge, and to get started, you can follow the link at the bottom. By the way, you can also put your default configuration in a text file so you won't have to run all these lines of code at the start of every notebook. All of this is explained in the documentation I've linked here.

Demo: Axes - Ticks and Tick Labels

A very important thing about plotting is how you're axes look. Are they readable? Do they give enough information? Let's see how to set up your axes correctly. We'll learn about ticks and tick labels, how to rotate tick labels, how to set the range of our axes, and how to correctly configure legends. A very important aspect of making a nice plot is to have your axes set up correctly. And if you do this right, not only will the plot be more attractive to look at, it will also be better at conveying information to your audience. There's three main things you may need to control for your axes, limits, which is where your axis starts and ends, ticks, which are the little indicator lines on the axes, and the tick labels, which is the text shown at each tick. So let's go ahead and make another plot. I'm reading in the now familiar sales. csv, and I'm creating a new column with month numbers, and then I do a groupby to calculate the monthly sales. Then I unstack to move the products into the columns. You can see what the data now looks like. The plotting code here should also be familiar now. We set up a style context, create a new plot, and I do a loop over the various columns, giving each their own style. All of this we've seen before. Now first, let's fix the ylabel and title. And to make everything fit, I'm going to call tight_layout. So this already looks a little bit better. But one thing I really don't like here is that matplotlib has decided to add only six ticks on the X-axis here. Now the ticks are these little indicators here, and because of my small resolution and image size, matplotlib only generates six ticks. I can change the position of the ticks by calling set_xticks on the X-axis and specifying a list of positions for the ticks. In this case, I simply want a tick for every whole number from 1 to 12. So I can use the function range and specify I want all numbers from 1 to 12. That looks a little bit better. But I would also like to change the text. I want to see the name of the month instead of the number. To do this, I can input the calendar module, and this has a list called month_name, which you can use to look up the name for each month. So here what I do is I take my index, which has all the month numbers, and I'm converting each month number into the name of the month by looking it up in the month_name list. And then the new list I get from that, I pass to ax. set_xticklabels. So this sets the tick labels for the X-axis. And when I run this, we do get the month names on the X-axis, but this is ugly. Let's rotate the text. I can add a parameter, rotation, and set it to 90. And now the text is vertical. By the way, we see that the labels now again go outside of the figure, and this is because I've added the code after figure. tight_layout. So this should be the last function call. Let's move it to the bottom here. Very well. And actually I prefer to have the text slanted, so let's set this to 30 degrees instead of 90. And although I think this looks better, now we have a different problem. The labels don't line up with the ticks. And to fix this, I add another parameter for the horizontal alignment called ha, and I set it to the value right, which means that I want to align the right side of the text with the tick. Good. So finally, we have a nicely formatted X-axis.

Demo: Axes - Limits and a Custom Tick Label Formatter

But what about the Y-axis? We have only four ticks here, going from 0 to 15, 000, and let's improve this. But first, I want to give the plot a little bit more space, and I'd like my axis to go up to a nice, round number. And we can set the range of the axis by saying ylim. So ylim means ylimits, and I'm passing a tuple with the start and end values of the axis. So let's say 0 to 20, 000. Nice. And as you can see, now we also have different ticks. There's now a new tick on the Y-axis for the value, 20, 000. And I'd like to have a little bit more control over the ticks and labels of the Y-axis. But the thing is, the plot is currently quite small because of the low resolution, and that means that for real display, I probably will want to resize it. And if I do that, we see that matplotlib automatically changes the number of ticks when I resize. And for the months I don't want this because I want all 12 months on the X-axis at any time. But for the Y-axis, I like this behavior. So I don't want to set a fixed list of tick positions for the Y-axis. Instead, let's add a function to make the labels nicer without breaking this nice behavior when I resize the plots. Here I have a little function that takes a tick as input and transforms it into the label I want. In this case, I divided by 1000 and put the letter K behind it. Then I import something called FuncFormatter from the matplotlib ticker module. And in the next line, I call ax. yaxis. set_major_formatter. And here I create a FuncFormatter object with the function I just created. And when we run this, we see some much nicer-looking labels on the Y-axis. And when I resize the plot, this even recreates the labels correctly when extra ticks are added. So how this works is that every time that my plot is rendered, for example F to resizing, matplotlib creates as many ticks for the Y-axis as it deems appropriate. Then, for each tick, it calls the formatter, which we have set to this FuncFormatter object, which in turn calls our function with two arguments, the value and position of the tick. And in this example, I'm only using the value of the tick to create a nicer-looking label. Now for a moment, let's take a look at the matplotlib ticker module. And this module provides us with a lot of functionality for generating ticks and labels for our axes. Here you see some classes listed that can generate ticks, for example the linear tick locator for evenly spaced ticks or the MultipleLocator for ticks at multiples of some value or the MaxNLocator, which can help you locate ticks at nice, round numbers. There are also some classes that can do tick formatting.

Here's a link to some example code that illustrates how to use these classes, as well as some extra documentation for formatting dates.

Demo: Customizing the Legend

Now the last thing to fix here is the fact that the legend is missing. If I just plot the DataFrame as a whole, this will give me a legend. But because I'm using a for loop to plot every product series separately, we get three calls to plot and we get no legend. Setting legend is true fixes this. Now if we want to change the text for each line, we can also do this by adding an argument label. And let's take the column name and prefix it with the word Product. And again, to make this look better, I have to resize it a little bit. Very well. So we now see that the legend looks better. Now for more control of the legend, there's also a method, ax. legend, which gives us a little bit more options. For example, we can set other labels, as well as a location for the legend. So let's change that to the upper-left corner, and I'm going to say that I want the text before showing the line styles. Very well. Now there's lots more you can do with axes and legends. For example, if you want to know more about how to customize your legends, you can Google matplotlib legend, and this takes you to the documentation of the legend method where you can see an overview of all the different options it takes.

Review: Working with Axes

We can set both the range of our axes and the position of the ticks when we call plot. For setting the limits, we pass an argument, xlim or ylim, depending on the axis. And we pass it a tuple with a start and end of the axis. So in this example, my X-axis will range from -2 to +4. In a similar way, we can set the ticks with an argument, xticks or yticks. And we pass it a list of positions where we want to see ticks. And this list doesn't have to be evenly spaced. In this example, I'm setting ticks at irregular intervals, and this is no problem at all. Now when you have a lot of ticks you need to place, of course you can use a Python function, like range, to generate a list of tick positions for you. And for more advanced use cases, there's the matplotlib. ticker module, which, as we've seen in the demo, contains a number of classes that can position your ticks for you. Tick labels cannot be set with df. plot. Usually it will use your index and the values of your data anyway. So it will set the tick labels and positions based on that. But it happens a lot that this is not what you want. And in that case, you can set a formatter function for your axis. You can see the function call at the bottom here. We first import the FuncFormatter class from matplotlib. ticker, and then we create the FuncFormatter object and set it as the formatter for our axes. We have to pass it a function that will actually do the formatting. So this function here takes a tick value and position and returns the corresponding label. In our example, it transforms the value of the tick into a formatted string. We can also control the way the legend looks. In the plot function, you can pass the label for the line you're plotting by adding an argument label. If you need more control over the layout of the legend, you can use the ax. legend method, which offers a lot of functionality. We can set the location of the plot, as well as a large amount of different layout options.

Demo: Creating Interactive Plots

And now we come to the final demo of this module. I'll show you a very cool library called Interact, which allows us to make our plots interactive with very little code. For the final demo in this module, let me show you something I think is really cool. There's a really awesome library called ipywidgets, and it allows us to create interactive notebooks in a very easy way. Before you can use it, you have to install ipywidgets. And to do that, you can go to the ipywidgets documentation at ipywidgets. readthedocs. io, and then you look at the install instructions. Now if you use Anaconda, follow the conda instructions. Otherwise, you can use pip. And once that's done, you can import ipywidgets. And we do that like this. From ipywidgets import interact and then import ipywidgets as widgets. So this function interact that I'm importing can make any other function interactive. As an example, here's a function f, taking two parameters, x and y, and it returns y*x. I can then call interact, pass the name of the function, and then I tell interact that I want to make the input of the function interactive. So let's say that I want to be able to set x in a range from -5 to 5 and y in a range from 0 to 10. Running this, we get two sliders, one for both inputs. And if I slide, let's I set x to -3 and y to 8, we see the result of the function here, x*y is now -24. And you can see that the value is updated while I move the controls. So don't you see that's just awesome? And of course, to suppress this line of output here, I can use a semicolon. Now if I set the value of x to a string in the call to interact, there will be text input. And this still works because Python lets me multiply strings with numbers as you can see. Or you can set x to a list, and this will create a drop-down. Now let's choose this to make an interactive widget that lets me explore my data. Now before I do this, please note that I'm using the inline backend at the top here because this automatically closes a plot after rendering it. Using the notebook backend will not do this, and you will slowly fill your memory with all the plots you make while playing with the controls. Also, the notebook backend tends to add and remove scroll bars all the time, and that makes it just slow and annoying. So we're going to use inline because it's simple and fast. So here's some code. I'm reading our athletes database, and then I have a function that can create a plot. It takes as arguments two sports and two genders, so we can select two subgroups of our dataset and plot them against each other. Inside the function, first I create my figure and select both subgroups by gender and sports. Then I plot both of them as a scatter plot with weight versus height. One group is plotted in red and one in blue. And I also set the labels that will end up in the legend. Now to make this interactive, first I create a list of all available sports by saying df sport. drop_duplicates, and I sort them as well. And then I call interact, and I pass the name of my function, make_plot. And then for all arguments of the function, I have to create a widget. Sport1 and sport2 get the list of all sports, so these will be drop-downs. And gender1 and gender2 also get lists with male and female, so those will be drop-downs too. And the result looks like this. I can select a sport. Let's get female triathlon players, very well, versus male cyclists. And we see our plot being updated in real time. So whenever I select a value in one of our drop-downs, Interact will call might make_plot function with new values, and a new plot is created. It's just that simple. So let's do another short example. Let's plot medals per sport and country. I'm calling groupby and calculating the sum per sport and country for all three types of medal. I think sort these values by the number of gold medals in descending order, and let's get the plot function. So here it is. It takes an argument n, which is how many of the top countries I want to see and the sport, which defaults to aquatics. In the function, we create the plots, and we set the figure size so that the width of the figure scales with the number of countries. And then I get the medals only for the sport that was selected and get the top 10 rows. Then I plot this as a stacked bar plot. Note the argument rot. It rotates the X-axis labels. We've seen this before through a call to the Axis object, but we can do it through plot as well. So I call interact, pass the function name, and set n to a range between 5 and 10 and sport to the list of all_sports. Running this, we can now ask for the top 10 countries in, let's say, gymnastics. Now when I move the slider, the plot immediately changes. And if you play around with this, you will find that this becomes quite slow at some point. The notebook will have to try and keep up with you moving the slider around. And it's all the time generating new plots. We can improve this a little bit by using a little piece of code. So here I set n to a widgets. IntSlider object, and when I do this, I cannot just set the min and max, but also set continuous updates to false. And this will make sure that the plot is only updated when we stop moving the slider and release the mouse button. So now I can move the slider around a lot, but the plot only changes when I let go. So that's it for the demos in this module. Let's go and review what we've learned.

Summary

And that's it for this overview of all the things you can do with Pandas and matplotlib. Honestly, we haven't even scratched the surface yet, but I think you'll have a pretty good picture of how things work now. In this module, we started by working with axes, we made some subplots, and saw how to share axes between them. We also saw that you can plot multiple series with bar and area plots and how to use stacking with these plots. We took some time to look at styling our plots with predefined styles, as well as learning about line styles and colors. And after that, we had a look at scatter plots and the functionality they have to offer. Then we investigated how to correctly set up our axes, and this included setting ticks and their labels, as well as the range of our axes. And we configured the legend of a plot too. Then in the last demo, I showed you how you can use the ipywidgets library to make really cool interactive plots with almost no work at all. I hope you enjoyed this module. Thanks for watching, and please join me in the next module where I will showcase another library for visualization called Seaborn.

Visualizing Statistical Properties with Seaborn

Introduction

Hello, and welcome to this module. I'll give you a brief overview of the popular visualization library, Seaborn. Seaborn focuses on statistical visualizations, and it can do many things. But generally speaking, it makes exploring the properties of a dataset very fast and convenient. It's based on matplotlib, so most concepts will be familiar to you by now. In this module, I will give you a short overview of what Seaborn can do. First, I will showcase relational plotting methods, showing the relation between two columns in our data, then categorical plots, which break down the dataset based on a categorical variable, and then I will very briefly go over some of the other cools things Seaborn can do, including calculating regression, generating pairplots, and heatmaps. Now in case this module leaves you wanting more, you can check out the in-depth course about Seaborn here on Pluralsight titled Visualizing Statistical Data Using Seaborn.

Demo: Relational Plots

So let me introduce you to Seaborn. We'll learn about two kinds of visuals it can generate, relational plots and categorical plots. So let me give you a few short pointers about how to use Seaborn. Here I have a new notebook, and of course I'm importing Seaborn, which by convention we do like this, import seaborn as sns. Now Seaborn is based on matplotlib, so Seaborn will import matplotlib itself in the background. Next, we call sns. set, and this does some configuration, like setting the matplotlib plotting style to the Seaborn theme, as well as setting the matplotlib backend, so we don't have to do that. Now let me also import Pandas, and I'm going to read the now familiar athletes dataset. And to make all of this a bit more manageable, I'm reducing the dataset a bit, so I'm only selecting athletes for these four sports, wresting, badminton, triathlon, and gymnastics. Good. Now the power of Seaborn is that it lets me very quickly explore relations in a dataset. In previous modules, we've done some comparisons of weight and height for various sports. Seaborn can generate plots to compare this with the function relplot, which it can call by says sns. replot, and then I pass our DataFrame as the date argument and the x and y column names as x and y. So so far, this looks pretty much like the function plot. scatter we've seen in the previous module. Let me show you. So this plots weights against heights. But one of the nice things about Seaborn is that it can make subplots for subsets of our data automatically. If I add an argument, col, and give it the name of column, let's say sport, Seaborn will generate a subplot for each value in this column. So now we have four subplots showing height versus weight for each of our sports. Now because of my low resolution, these plots are quite tiny. But I can change the layout by asking Seaborn to wrap every two columns into a new row. And let's use one of our categorical columns as a caller, so I'm going to say hue is sex. So this means the callers of our data points will now be determines by the value of the sex column. So here we now have four plots in two rows and two columns. For every sport, we have a subplot, the points are colored by sex, and we plot weight versus height. Now you may remember that to create a similar plot with Pandas and matplotlib, we need to create a for loop, work with axes and figures, etc. So this is where Seaborn really shines. It splits up our data into subsets and creates a subplot for each in a beautiful way. We can use categories for colors or for subplots simply by adding arguments. Now for a moment, let's look at the documentation for Seaborn. I'll go to the Seaborn home page at seaborn. pydata. org, and I'll click on API here at the top. And this lists the various Seaborn methods, and at the top here is relplot, the function we just saw. And these methods are grouped by the kinds of things we are comparing. So these first three methods here make relational plots, and relplot is described as a figure-level plot. There are two methods below it, scatterplot and lineplot, which are not figure-level. And the way to understand it is that each subplot of the figure-level plot will contain one of these, a scatter or a line plot. Scatter plot and line plot are not figure- level, but access-level plots. They create either a single scatter plot or a single line plot for a single axis. Relplot will create a figure with multiple subplots, and it will call these functions for each subplot. Looking back at our example, the call to replot sets up the figure and subplots in that figure for each value of the call argument, so for each sport. By default, it calls scatterplot for each, and that's why we see four scatter plots. I can change this by adding an argument, kind. And now we get line plots for each sport. Now of course for this dataset, this doesn't really make sense, but you get the point. By the way, the warning we get here, we can ignore at the moment. Now let me show you one other example of what relplot can do.

Demo: Relational Plots - Second Example

Here I'm reading the weather dataset, and I select only the month of August. Looking at the first few lines, we see that this dataset contains the temperature and the pressure measurement for every hour in every day in August. Now we can make relplots like this. Again, I pass the DataFrame as the data argument, column names for x and y, and I say that I want a line plot in this case. Again, we get a warning, which I can ignore, and here I'm plotting the temperature against the time. Now the interesting thing here is that this automatically takes the average over all points in the set. So for every hour on the X-axis, we now see the average temperature over the entire month, as well as the confidence interval for this average as the shaded area around the line. So here Seaborn actually does some very useful calculation for us, which otherwise we would have to do for ourselves.

Demo: Categorical Plots

Taking another look at the API docs, here you'll see a list of categorical plotting methods. And again, the first method, catplot, is listed as being figure-level. So again, this means it will call one of the other methods for each subset of our data. Clicking on caplots, here we see the list of eight kinds of plots we can make. And it's subdivided in scatterplots, distribution plots, and estimate plots. Let's try some of those. Back in my notebook here, I'm calling sns. catplot, again with the athletes data. But this time, it's important to understand that either x or y here has to be a column with categorical data. So in this case again, I'm going to split our data in subsets based on sports. Then I want to see the weight on the other axis, and let's color each point by the athlete's gender. Running this, we see that catplot makes one plot, so no subplots, and it plots the values we see for the weights of athletes by sport as a kind of scatter plot that gives us an impression of the distribution of these values. Like with relplots, I can use the kind argument to select a different plot type. So for example, I'm going to set it to boxen, which is a fancy type of box plot, and this gives a nice overview of the distribution of weight per sport and per gender. So the main difference between catplot and relplot is that relplot makes a subplot per category and then plots two dimensions against each other, whereas catplot makes a single plot and plots one dimension for each category.

Demo: Other Features

Now let's take a short look at some of the other nice features that Seaborn has to offer, like automatically doing regression on our data, generating a so-called pairplot, which gives an overview of the relations between all columns, and generating heatmaps. In this demo, I'll just quickly show some examples from the documentation of other Seaborn functionality. For example, Seaborn can automatically do a regression on your data and plot the result. The main function to do this is lmplot, which, if we click it and scroll down, we can see it creates a scatter plot of our data, and it also does linear regression and draws the result of that regression as a line in the graph together with the confidence interval. Of course, there are several ways in which you can configure the way this regression is done. So for example, if you need algorithmic regression, this is also possible, but I won't be going into that here. Like with the other plots we saw, you can break your data into subsets and automatically generate subplots for each subset. Another cool thing Seaborn can do is generate joint plots. And for that, there's the function jointplots. Now this plots two columns against each other like a scatter plot, but it also shows the distribution of each column as a histogram on the axes. You can customize this plot in various ways, like plotting regressions and kernel density estimates or using hexbins, etc., etc. But my favorite Seaborn method is pairplot, and what this does is it takes all columns from your data and plots them against each other. And this gives you a very quick overview of your dataset and how the various columns relate. This is very powerful, and it's one of the first things I might do when I need to get a feeling for a new dataset. Now the final thing I have to say is if we go back here under matrix plots, there's the function heatmap. And if you use this function, Seaborn can also create wonderful heatmaps for you.

Summary

So this was a very brief introductory overview of Seaborn, which is a visualization library focused on statistical plots. And it really makes exploring our data so much faster and easier. We had a quick look at how to create relational plots and categorical plots, as well as doing regression, generating a pair plot, and heatmaps. Thank you for watching. I hope to see you again in the next module in which I will introduce another library called Bokeh.

Creating Interactive Plots for the Web with Bokeh

Introduction

Hi, my name is Reindert-Jan Ekker, and welcome to this final module in which I will introduce another visualization library called Bokeh. So Bokeh is another library for visualization, but it's focused more on creating interactive components for the web. Bokeh is huge, and it has lots of interesting functionality. So in this module, I can only give you a brief overview of what it can do and how to start using it. We'll create just a few simple plots to give you a general feeling of how the library works. By the way, Bokeh is not based on matplotlib, so even though a lot of concepts are similar to what we already know, many things are also very different than we have seen so far. Now where Bokeh really shines is creating content for use on a website. You can easily export your plots to an HTML document, which will contain JavaScript code to create a figure and any interactivity. And we'll see how to do that in the demos.

Demo: Simple Plots with Bokeh

I'm going to start with an introduction demo that shows you how to get up and running with Bokeh. We'll create two simple plots, and we'll learn about two important concepts, cliffs and the column datasource class. So here's a Jupyter Notebook, and I start by importing Pandas. Then I import three functions from Bokeh that we need to make simple plots, figure, output_notebook, and show. And because Bokeh is not based on matplotlib, we don't need to set a matplotlib backend here. Instead, we start our script by calling output_notebook, which tells Bokeh to create output that the notebook can display. And then of course, I read a dataset. This holds the sales figures that we've seen before in other modules. You can see here what the data looks like. Now let's make line plots to show these sales. Like with the other libraries we've seen, we start by creating a figure. We do this by calling the figure method, and we can set many options here. As you can see, I'm setting a title, the labels for the axes, the range for the Y-axis, and the dimensions of the figure in pixels. And this function call returns a figure object, which we store as the variable p. Then we can call methods to draw things on this figure. The first call here is to p. line, which, of course, draws a line. We have to pass it the x and y coordinates of the points that the line will connect. The x coordinates in our case are given by the m column, which holds the number of each month, and the y coordinates is the column holding the sales for Product A for each month. I also add a legend for this line, and I'm setting the line width. Similarly, I can call p. square here to draw a square for each data point. And in the output scrolling down, we see a blue line for Product A with squares at each point. You can also see that I'm creating a line with circles for Product B and with triangles for Product C. So so far it's all pretty similar to matplotlib, although you should note that the things we draw, like lines and circles and triangles, are called glyphs in Bokeh, and these glyphs are the parts that make up our plots. So scrolling a little bit up again, here you see the code to create lines and circles for Product B and triangles and lines for Product C. And we can, of course, do a lot of customization. For example, I could change fonts and font colors and sizes. Or like I do here, I can customize the legends. In this case, I've moved the position to the top left of this plot. And finally, to actually create the plot, I have to call the show method and pass it the figure object. And this will cause the figure to actually show. So if I wouldn't have this line, we wouldn't see the figure here. Now you should also note that Bokeh automatically adds some controls to zoom and pan the figure or save it to a file, much like we can do with matplotlib. Of course, Bokeh can also create complex layouts with subplots and things like that, but let's keep it simple for now.

Demo: Simple Plots with Bokeh, Continued

So let's see another short example where I create a bar plot. And this bar plot will show the total sales per product. I start by calculating the total sales by taking the sum of our DataFrame after first dropping the month column because we don't want the sum of all the months. So this is what our totals variable now looks like. And then I create a figure, again setting the title and axes and the figure size, and then I call p. vbar. This draws vertical bars. So a bar for Bokeh is nothing special. It's just another glyph, like a line or a circle. And the vbar method works slightly differently. Instead of setting x and y, I have to set x and top. So the x position of each bar is given by x, and then the vertical top of each bar is given by top. So each bar, by default, starts at 0. This is the bottom. And then it reaches up to the top. So running this, so as you can see, a lot of things in Bokeh are similar to matplotlib, but still kind of different as well. Now let me show you something slightly more advanced. Suppose I want to change the colors of the bars here. And as soon as I want to do some real customization, I find that I need a so-called ColumnDataSource. So this is what I import right here. This class ColumnDataSource is quite central to Bokeh. It is what most plots will use to get their data from. To make use of all the power of Bokeh, it's best to wrap our data in a ColumnDataSource. Now our totals variable currently holds a series, but the ColumnDataSource only understands Pandas DataFrames. So first I create a new DataFrame from our totals variable, and then I wrap this DataFrame in a ColumnDataSource. And this then I store as the variable src. So from here, the Bokeh plots will take src as the source of their data. And, by the way, this is what our DataFrame looks like. Now consider the following code. I set up a figure, and I do a call to vbar just like before. But this time, I pass our src variable and the x and top arguments, and I take the name of the column that they will take their data from. So I no longer pass the data directly here to x and top. But the plot will take the data from each respective column in the datasource. And then I do something interesting. I pass an argument fill_color and give it as a value a call to factor_cmap. This is a Bokeh method that can transform values from the data into colors taken from a palette. You can see importing the palette and the factor_cmap method at the top of the cell. So this transforms the value a, b, and c from my index into colors from the palette. And my point here is not so much to make you understand exactly what is going on here. My point is that to do these kinds of things, you need a column datasource because these kinds of transformations, like most things in Bokeh, expect to work with a ColumnDataSource. So that's why normally you will want to wrap your DataFrame in a ColumnDataSource like I did above, and this will make Bokeh play nicely with your Pandas DataFrame. So running this, we see the same bar plot as before, but now with three nicely colored bars.

Demo: Creating Interactive Plots for the Web

Let's take a look at some of the strong points of Bokeh. We'll see how to add interactivity to a Bokeh plot and how to export it as source code that we can use on the web. So here we are again, and I start with slightly different imports. First, pandas, and then from bokey. plotting, I import figure, show, and now also ColumnDataSource. And this time instead of output_notebook, I import output_file. As the name implies, this tells Bokeh to send the output to a file instead of the notebook. I also import the color palette called Set2, and I call it Colors. Now like before, we then start the script by immediately calling the output_file function, and I pass a file name, sales_app. html. Then I proceed by reading the CSV file again, and the DataFrame I create from that, I'm immediately wrapping in ColumnDataSource, so we can then use it with Bokeh. And here's a small overview of what our data looks like, and this is the code I used to create my plots. It's very similar to what we've done before. First, I create a title, set the axes labels and the range and the plot size, and then I plot three lines, one for every product. But note that this time I used the src argument and passed our ColumnDataSource. So now I can set the x and y arguments with the name of the column to use, so m for the month and then a, b, or c for the actual products. I also set the legend, line_width, and for each line, I add a color from the palette. After setting the legend location, there's now something new here. I import something called

HoverTool, and this I do to show how easy can be to add interactivity to our plots. The HoverTool lets me add tooltips. I saw p. add_tools and pass it a new HoverTool object, and this HoverTool I tell to create tooltips that show four things, the month taken from the m column and each product taken from the columns a, b, and c. So these @ signs here followed by m, a, b, and c points to the columns in our datasource. Now of course, to actually create this figure, we call show. And then when I run this cell, because we set output_file at the top instead of output_notebook, the effect will be different. What happens is we see a new tab open, and this shows us a new file that has been created called sales_app. html like I configured at the start of my script, and this file contains my plots. And when I move the mouse over the data points, we see tooltips pop up with the exact values for each data point. Now I just think that's awesome. And by the way, if you would choose to output to a notebook method, this would also work inside the notebook. So this interactivity is not exclusively something that works in the file output only. It also works in the notebook. But the cool thing about the file output is that this is actually a self-contained HTML file with working JavaScript. So you could actually use this very file as a part of a website. Or if we go back to the notebook, if you want to embed your figure in a website, you can also use this little piece of code here. It imports a method called components, which returns a script and a div for the plot we just made. Both are pieces of source code. The script, as you can see here, is the JavaScript that will create our figure, and the div contains an HTML div element you can use to embed this figure in your HTML page. So with these two pieces of source code, you have everything you need to embed a working interactive Bokeh figure in your web page. And that's it for my short, informal intro to Bokeh. As I've said and as you can see here on the Bokeh home page, Bokeh can do many, many things. And to see some cool examples of various kinds of things it can do, you can check out the gallery here. It really shows you that Bokeh can be used to create complete interactive web applications for data visualization. So in our demo, I've only given you a first taste. There's so much more to explore. This is as far as I will go in this course, so let's wrap this up. And that's the end. Not only of this module, but of this entire course. We've just seen a brief introduction of Bokeh, including how to create some simple plots, how to create content for the web, and how to add interactivity. Thank you for watching this course. I hope you enjoyed it, and I wish you happy coding, and I hope that you learned something useful. This was Reindert-Jan Ekker for Pluralsight.

Course author

Reindert-Jan Ekker

After studying Artificial Intelligence, Reindert-Jan has worked in many different roles in the software development life cycle, from tester and developer to Scrum Master. Then he decided to go...

Course info

Level	Intermediate
Rating	★★★★★ (22)
My rating	★★★★★
Duration	2h 11m
Released	9 Oct 2018

Share course