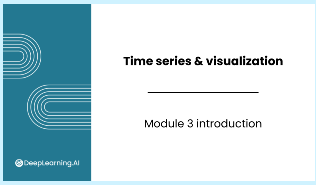
DAG C3M3 scripts

Last updated: Nov 22, 2024 , 11:28am

Video title	lecture	Isabel	Sean	slides	alpha
L0v1 – Module 3 introduction	-	✅	✅	✅	✅
L1v1 – DateTimes	✅	✅	✅	✅	✅
L1v2 – Using DateTimes as indices	✅	✅	✅	✅	✅
L1v3 – Line charts & moving average	✅	✅	✅	✅	✅
L1v4 – Percent change	✅	✅	✅	✅	✅
L1v5 – Segmentation & grouping	✅	✅	✅	✅	✅
L1v6 – Multiple line charts – no reshaping	✅	✅	✅	✅	✅
L1v7 – Method chaining	✅	✅	✅	✅	✅
L2v1 – matplotlib	✅	✅	✅	✅	✅
L2v2 – Text and annotations	✅	✅	✅	✅	✅
L2v3 – Date and axis formatting	✅	✅	✅	✅	✅
L2v4 – Multiple line charts – reshaping	✅	✅	✅	✅	✅
L2v5 – Column charts	✅	✅	✅	✅	✅
L2v6 – Scatter plots	✅	✅	✅	✅	✅
L2v7 – Grouped column charts	✅	✅	✅	✅	✅
L3V1 – Plotting with Seaborn	✅	✅	✅	✅	❌
L3V2 – Subplots	✅	✅	✅	✅	❌
L3V3 – Box plots	✅	✅	✅	✅	❌
L3V4 – Histograms and rugplots	✅	✅	✅	✅	❌
L3V5 – Other charts	✅	✅	✅	❌	❌

Introduction

L0V1 – Module 3 introduction

Visual	Script
🎙️ TH  Time series & visualization Module 3 introduction DeepLearning.AI	Welcome to Module 3: Time Series Analysis and Data Visualization! In this module, you'll learn to analyze time series data as well as master two essential libraries for data visualization: Matplotlib for customization, and Seaborn for enhanced visual appeal. Throughout this module, you'll analyze real-world stock market data from the top 50 largest companies to track market changes over time. The first lesson covers working with time series data in Python. You'll explore DateTime data types, learn to use them as indices, and create line charts with moving averages. You'll also calculate percent changes and segment data to extract meaningful patterns. The second lesson focuses on creating customized visualizations using Matplotlib. You'll enhance charts through annotations and formatting, then build complex visualizations like scatter plots and grouped column charts. In the final lesson, you'll explore advanced visualization with Seaborn. You'll create appealing charts with minimal code, compare distributions with subplots, and master specialized plots like box plots, histograms, and more. By the end of this module, you'll be able to create professional-quality visualizations to derive meaningful insights from your data. Follow me to the first lesson to begin exploring DateTime data types in Python. See you there!

Lesson 1 – Time series

L1v1 – DateTimes

Visual	Script
🎙️ TH Data structures and descriptive statistics DateTimes DeepLearning.AI	What's one important date in your life? I always remember November 2, 2016, the day my beloved hometown baseball team, the Chicago Cubs, won the World Series, breaking a 108-year drought of no championships. Dates are special to us, and time series data is crucial for many business problems. But dates are actually quite complex to represent — because they contain a lot of information.
	You've already worked with a lot of **[CLICK]** **data types**. You've seen how **[CLICK]** numbers and **[CLICK]** strings are represented **[CLICK]** in Python, as

	well as how **[CLICK]** pandas allows you to store data in rows and columns, with **[CLICK]** Series and **[CLICK]** DataFrames. You'll see in a moment that dates get their own special data type **[CLICK]** in both Python and Pandas: the **[CLICK]** DateTime type. It's a particular **[CLICK]** representation data type that represents the **[CLICK]** year, **[CLICK]** month, **[CLICK]** day, **[CLICK]** hour, **[CLICK]** minute, **[CLICK]** second, and even **[CLICK]** microsecond of a particular time. You could represent all those values in a string, like **[CLICK]** this, or in separate integers, like **[CLICK]** this, but the **[CLICK]** DateTime bundles everything together for convenience. DateTimes **[CLICK]** comes bundled with special methods. For example, if you create a DateTime, you can isolate the different components of the date, like **[CLICK]** day,**[CLICK]** hour, or **[CLICK]** second. You can also use the **[CLICK]** `weekday()` method to get the day of the week, like Monday. There are a lot more methods too. These are operations that **[CLICK]** wouldn't make sense to use on any old number like 3 or string like 'banana'.
🖥️ Screencast 🔗 **c3m3l1v1 datetimes**	Say you want to examine the change in stock prices for the top 20 largest companies in the S and P 500 from 2014 to 2024. Just for some context, you have your notebook "stock price analysis," and you have your csv file "stock underscore prices dot csv". These files are in the same folder. Again, you won't need to do this in Coursera. First thing, import pandas as pd, and then create your data frame by reading that CSV file: stock prices dot csv. Now all that data is loaded into your notebook. Here are the first five rows using df dot head. The unnamed column appears to be the index, with features for the date, the ticker, which is the name of the stock, the open price and so on. Each row represents an observation of one stock on one particular day. It appears this data is sorted in some way, because it's all Apple (AAPL is Apple), or maybe all electronics. To explore some additional companies in the data set, you can sample 10 rows, and you see some other stocks in here, like Walmart, JP Morgan, and some others that you may not be familiar with. One thing you may want to check out is the d-types, or the data types, in your data frame. Looks like date is an object. To explore in more detail, you can select the date column, remember this is a series, and then select the first value from that result, just like you would with a list by using bracket zero. It looks like it's represented in year, month, and day, so the 2nd of January, 2014, and the type of this value is a string. It's not a special date type, it's just a string. If you try to get the weekday of that date, you get an error. You can convert the date column from a string to a datetime, and you can do that using the P D dot to_datetime function. What are you going to call this

function on? Whatever you wanna turn into a datetime. In this case, you can select the entire column as a series and change all of those values to a datetime in one operation.

One hitch: if you just run that code, you get a new series, but this command doesn't actually modify your original data frame. The type of df[“date”][0] is still a string. To convert that column and actually save it, you can assign it to a new column. You could save it to the old column, too, date, if you'd like, but to preserve the original column, it’s best to create a new column, DateTime. That way if you make a mistake your original data isn’t overwritten.

Now, if you take a look at D F dot DateTime. This command selects just one column out of the data frame. Taking the first five values, the result you get shows that the type is DateTime. If you look at the first value, you get a timestamp of the date with zero o'clock associated with it. There wasn’t a time in the original data, so you just get a date and a default time of midnight.

Now, because these values are date times, you can do cool date-related operations with them. For example, you can find the weekday: three. Weekday starts at zero for Monday, and so three is Thursday!

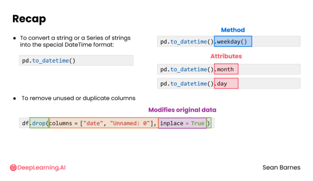
You can also do stuff like, what is the month? This is just an attribute, or a quick question about the datetime, so no parentheses, and you get the month was one, corresponding to January. You can do dot day, dot second, and so on. So, earlier it was maybe a little ambiguous whether this was January 2nd or February 1st, just looking at the string, but storing the information this way in a date makes it very clear what this data means.

Now, the DataFrame is quite large: 139,000 rows, so you may want to drop (or remove) some of the extra columns.

You can do that with the D F dot drop method. Since you've created “datetime”, you may want to drop the “date” column. You'll use a named argument columns, which can be a list. So what columns do you want to drop? Date, and it looks like unnamed zero could probably get dropped as well, since you already have the index. It’s a duplicate.

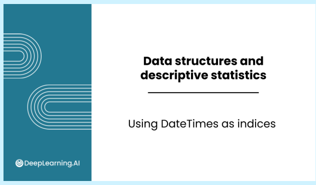
By default, just like dot to_datetime, drop creates a new data structure rather than modifying the old one. If you want to just remove the columns without duplicating the data frame, you can use the named argument in place equals true. This argument modifies the original data frame directly, rather than creating a new data frame with the modification, leaving the original data frame unchanged.

Now if you take a look at D F dot head, now you have this nice clean dataset with no date and no unnamed zero column.

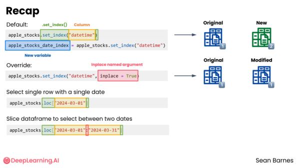

Recap
• To convert a string or a Series of strings into the special DateTime format:
`pd.to_datetime()`
Method
`pd.to_datetime().weekday()`
Attributes
`pd.to_datetime().month`
`pd.to_datetime().day`
• To remove unused or duplicate columns
Modifies original data
`df.drop(columns = ["date", "Unnamed: 0"], inplace = True)`
DeepLearning.AI
Sean Barnes
	To quickly recap, you can use **[CLICK]** PD dot to_datetime **[CLICK]** to convert a string or a Series of strings into the special DateTime format. Then you can use methods like **[CLICK]** dot weekday and attributes like **[CLICK]** dot month or **[CLICK]** dot day to perform operations on those dates.
	
	You also saw that you can use the **[CLICK]** dot drop method **[CLICK]** to remove unused or duplicate **[CLICK]** columns, and you saw that the **[CLICK]** inplace argument, when set to true, **[CLICK]** modifies your original data rather than creating a new data frame.

🎙️ TH	Once you've converted some of the dates as strings into proper DateTimes, you can use those dates as indices for your DataFrame, replacing the default integer indices. Follow me to the next video to see how.

L1v2 – Using DateTimes as indices

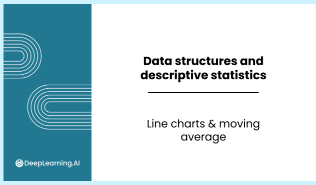
Visual	**Script**
🎙️ TH  Data structures and descriptive statistics Using DateTimes as indices DeepLearning.AI	Dates have a natural order to them, and you can organize your data frame in a way that takes advantage of that natural order. Specifically, you may be interested in setting the index of your data frame to be the datetime you just created, rather than a plain number.
🖥️ Screencast 🔗 c3m3l1v2 datetimes as indices	One interesting thing you can do with datetimes is use the datetime as the index for your data, instead of having an integer as the index, which is the default configuration. That can be really useful if each date uniquely identifies a row. For example, say you wanted to examine only the Apple stocks. You could use the date time as the index for each row. First, filter the data to only the stock prices for Apple stock. Select from the data frame where ticker equals AAPL. Now if you look at apple stocks dot sample(10), these are daily records for Apple stock, so it looks like that operation worked correctly. So, what you can do now is set the date time as the index. Start with your data frame, you only want to use this on apple stocks because you don't want to change your original data frame, where this operation wouldn't be as appropriate. Dot set index, and then the name of a column. It could be any of these columns in theory, but date time is the one you want to choose. Just like to_datetime and drop, by default set Index creates a new data frame. So it duplicates the old data frame and does some operations on it to make dateTime the index. You have two options to deal with that: you can either

	save this to a new variable or you can just use the named argument inPlace equals true. Now inPlace equals true is a little bit dangerous because you're overwriting your previous data, but in this case, it's okay, because you can always rerun the previous cell here to essentially refresh the Apple data. If you take a look at apple stocks dot head, the Index has been replaced by the datetime column. That's really cool because now you can select specific dates and slice your data frame using those dates. For example, look at apple stocks dot tail: tail is the opposite of head and allows you to get the last few rows of the data frame. The last rows are from October of 2024, so now you know the range of available dates. Say you want to access the 1st of March, 2020, as a baseline, maybe before the pandemic. You could say Apple Stocks dot lock, and then select the date that you want, using a string. In this case, you want to select 2020, 03, 01. Lock is similar to iloc, which you saw previously. Iloc is for integer location, but in this case you changed your index from integers to datetimes. Instead of integer location, you can just use the location specified by a given date. Shift enter, that gives you all the information about that date. It was Apple stock, of course; the high price was 180 dollars for that day, and so on. You can also now **slice** your data using these dates. For example, if you wanted to select all the rows from March 2020, to see the effects of the pandemic, you could start with what you had before, but now use the slicing operator, the colon, to select from the start through the end of March. March has 31 days, so you can just borrow that same date and change the day to 31. The result is a data frame because you selected multiple rows. Now, you can examine how the Apple stock price behaved throughout the month of March 2020! You can see a couple of dates are missing, so there is not necessarily a measurement every day. For example, the 9th and 10th are missing, which correspond to weekend days when the stock market is closed.


Recap
Default:
`apple_stocks.set_index("datetime")`
`apple_stocks_date_index = apple_stocks.set_index("datetime")`
Override:
`apple_stocks.set_index("datetime", inplace = True)`
Select single row with a single date
`apple_stocks.loc["2024-03-01"]`
Slice dataframe to select between two dates
`apple_stocks.loc["2024-03-01":"2024-03-31"]`
DeepLearning.AI
Sean Barnes
 .set_index()
Column
New variable
Inplace named argument
Original
New
Original
Modified
1
2
1
1
	To recap the tools you just learned, you first saw the **[CLICK]** set index method, which takes a **[CLICK]** column name as an argument. By **[CLICK]** default this method will **[CLICK]** create a brand new data frame rather than modifying the old one. So, you can save it into a new variable, like **[CLICK]** apple stocks date index. That leaves your original data frame intact.
	You also saw that you can use the **[CLICK]** inplace equals True named argument to **[CLICK]** override that default behavior. Inplace equals true means the set index method will **[CLICK]** modify the data frame directly rather than creating a copy first. Be mindful when you're using it because if you mess something up, you'll have to go back and execute the previous steps to reset your data.

	You also used the **[CLICK]** dot loc method to select rows based on the datetime index. You can **[CLICK]** select a single row with a **[CLICK]** single date, or you can **[CLICK]** slice your data frame using the **[CLICK]** colon to select a range between **[CLICK]** two dates.
🎙️ TH	Excellent work with datetime indices! You won't always need them when analyzing time series data, but if they fit your data, they can make selection a breeze. Now that you're comfortable using date times, follow me to the next video to see how to describe time series data in Python. I'll see you there!

L1v3 – Line charts & moving average

Visual	**Script**
🎙️ TH  Data structures and descriptive statistics Line charts & moving average DeepLearning.AI	You learned how to conduct a ton of cross-sectional analyses in Python in the previous module. So what are your options for time series data? You can start with graphing and calculating moving averages
🖥️ Screencast 🔗[c3m3l1v3 line charts and moving average]()	Now that you have your apple stock price time series, you'll want to analyze the trends over time. You may think to try dot describe, and that does give you some information, like the highest high price is $237, the lowest low price is apparently $17, which is kind of wild. However, describe treats this data as cross sectional, so it doesn't help you understand how the data is trending over time. Remember, the first thing in time series analysis is visualizing the data. You can start by graphing the closing price. Because your index is a date time, you will automatically get a really nice looking line graph when you are working with a single column. So for example, if you want to plot close, the price at the end of the day, essentially; first, select the close column. If you take a look at this result, it's a series, right? You can see that the date time as the index gives you this nice X and Y axis structure for the data. You can then add dot plot, and, this time, kind equals line. Shift enter, and you get this nice line chart, and you can see this big upward trend in the stock after 2020. The stock has really gone up in value since around 2020. There also appears to have been a lot more volatility in the stock since that period as well. You can also graph volume. Volume is the number of shares in the stock that were traded on that particular day. Volume dot plot, then kind equals line. This is a busy chart; it's pretty difficult to see the trend, but it seems to be moving

downward over time.

You may be interested in calculating the moving average for volume to help smooth out the noise in the time series. Remember the moving average smooths out the data and helps you reduce the impact of these spikes. To calculate the moving average, start with the column you're interested in, which in this case is apple stocks volume, and you want to use the dot rolling method. Rolling creates subsets of your data that have a certain window length. Use the named argument window equals some number. For example, each of your observations is a day, so maybe you want to calculate the average over 14 days or two weeks.

If you just shift enter there, you don't get any kind of rolling average, you just get this rolling output, which represents all of these subsets of your data. So, the rolling method takes care of just the grouping of the rows, in this case into sets of 14. In order to actually do something useful, you're going to need to apply some kind of aggregation function. For example, you can calculate a simple moving average using dot mean.

Like many methods in pandas, this command won't directly add a new column to your dataset. It's just going to generate a new series. So you'll want to save that into a variable. Rolling 14 day average. Now, if you take a look at the first 14 values here, what do you expect to see? **[pause for thought]** Well, the first 13 cells are empty because you don't have 14 days of data to average, but after that, you get the first moving average value. You can assign this result to your data frame by creating a new column, 14 day volume, and saving the rolling 14 day average series into it.

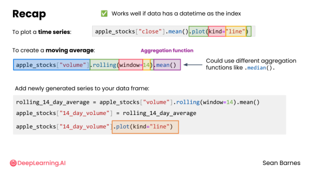
Now if you look at apple stocks dot head, with the first 14 values at least, that new column has been added to your data frame, which is very cool! You can plot this column in a line chart. First, select the 14 day average column, dot plot, kind equals line.

You can see that this line graph is much smoother, with fewer spikes, but there is still some noise in the data.

You can also create a monthly average with 30 days. This is a very long piece of code; it just does all the selecting, grouping, averaging, and plotting steps in one step. This is called chaining, because you are executing multiple steps in one line of code, chained together as one single command. You can see that compared with the 14 day moving average, the 30 day moving average is even smoother.

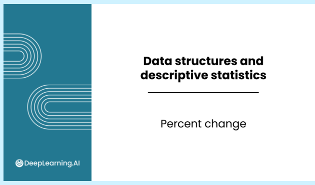
And this 90 day average is even smoother than that, so you're starting to really distill the trend of your time series data, without all of the distracting noise. These types of longer-term moving averages are often used in stock price and

	volume analysis because the variability is often significant from day to day.

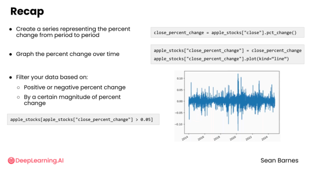

Recap
✅ Works well if data has a datetime as the index
To plot a **time series**: `apple_stocks["close"].mean().plot(kind="line")`
To create a **moving average**:
`apple_stocks["volume"].rolling(window=14).mean()` Could use different aggregation functions like `.median()`.
Add newly generated series to your data frame:
`rolling_14_day_average = apple_stocks["volume"].rolling(window=14).mean()`
`apple_stocks["14_day_volume"] = rolling_14_day_average`
`apple_stocks["14_day_volume"].plot(kind="line")`
DeepLearning.AI Sean Barnes
	Recapping what you just learned, you saw that you can use the **[CLICK]** dot plot method with the named argument **[CLICK]** kind equals **[CLICK]** line to **[CLICK]** plot a time series chart. This command **[CLICK]** works well if your data has a datetime as the index rather than just an integer.
	
	You also saw how **[CLICK]** to create a moving average. First, select the column you want to create a moving average for, like the **[CLICK]** volume column. Then, use the **[CLICK]** dot rolling method with a **[CLICK]** number for the **[CLICK]** window named argument. You saw 14 day, 30 day, and 90 day averages, which are appropriate choices to smooth out daily observations. Finally, you'll need to apply an aggregation function. For a moving **average**, you'll want to use **[CLICK]** dot mean, but you could also use different aggregation functions like **[CLICK]** median.
	
	Then, you can **[CLICK]** add that newly generated series to your data frame, if you'd like, as well as plot it in a line chart using **[CLICK]** dot plot kind equals line.

🎙️ TH	Moving averages are a great tool for smoothing out noisy, time series data. It's quite fast to implement in Python compared with a spreadsheet. Follow me to the next video to see how to implement a percent change column.

L1v4 – Percent change

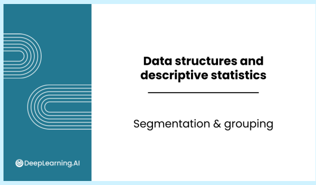
Visual	**Script**
🎙️ TH  Data structures and descriptive statistics Percent change DeepLearning.AI	One powerful tool you have for time series data is the percent change, which helps you identify changes compared with the previous time period.
🖥️ Screencast 🔗 c3m3l1v4 percent change	Now, focusing back on the closing price for a moment, the percent change will tell you whether the stock went up or down compared with the previous day. To calculate the percent change, first select your column, close, which is the closing price, and the method you want to use is pct for percent, underscore change. Save this result to a variable. Now, what do you expect to see when you look at percent change? Will it have the same number of empty cells as the moving average did? **[pause for thought]** Only the first cell is empty because there's no previous time to compare it with. The second day showed a slight decrease relative to the first, the next day had a slight increase, and so on.

	Now you can create a new column in your data frame, Apple Stocks. Then you can plot that percent change over time as well, by specifying kind equals line. You can see the percent change over time. It fluctuates quite a bit; eyeballing it, the mean seems to be near to zero with both positive and negative spikes representing large daily changes. Now, percent change can be useful for, for example, filtering your data. Here is a quick example. You can filter the time series for Apple stocks to where the close percent change it's greater than... what might be interesting here? Say, 0.05, so the closing price went up more than 5% compared with the previous day. That might be pretty interesting. There are not a whole lot of these days; you can see all of the rows here. It looks like those days are pretty spread out over time, except there are a bunch of them in 2020. You can see that there are 26 days in total that had more than a 5 percent increase compared with the previous day. You could examine these days further to see what events may have precipitated these sharp spikes in the trading price.

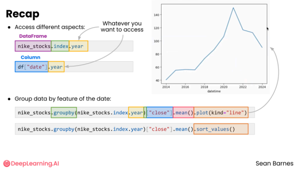

Recap
- Create a series representing the percent change from period to period
`close_percent_change = apple_stocks["close"].pct_change()`
- Graph the percent change over time
`apple_stocks["close_percent_change"] = close_percent_change`
`apple_stocks["close_percent_change"].plot(kind="line")`
 0.10
0.05
0.00
−0.05
−0.10
2014
2016
2018
2020
2022
2024
- Filter your data based on:
○ Positive or negative percent change
○ By a certain magnitude of percent change
`apple_stocks[apple_stocks["close_percent_change"] > 0.05]`
DeepLearning.AI
Sean Barnes
	Playing that back, you can use the **[CLICK]** dot P C T change method to **[CLICK]** create a series representing the percent change from period to period. In the demo just now, you saw the process of creating a new column to represent the percent change in closing price from the previous day.
	
	You can also **[CLICK]** graph the percent change over time, or **[CLICK]** filter your data based on **[CLICK]** positive or negative percent change, or **[CLICK]** by a certain magnitude of percent change, like the days with **[CLICK]** 5% or greater increase in closing price compared with the previous day.

🎙️ TH	Once you've examined the price and volume fluctuations in one stock, you may be interested in segmenting the time series based on different features, like the category.

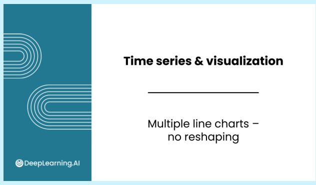
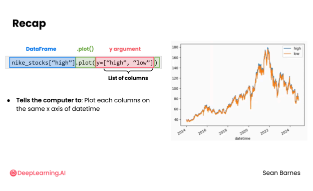
L1v5 – Segmentation & grouping

Visual	**Script**
🎙️ TH  Data structures and descriptive statistics Segmentation & grouping DeepLearning.AI	You have a lot of options for segmenting or grouping your time series data. You can segment by categorical features, like a stock's market sector, and you can also group by features of time, like the mean monthly or yearly price.
🖥️ Screencast 🔗 c3m3l1v5	Returning to the original data frame with all of the stocks, let's check out a different company. Looking at the 20 different companies, you may recognize some of these here, like Meta and Tesla. Let's go ahead and take a look at

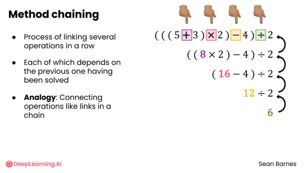
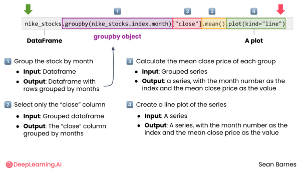
segmentation and grouping	Nike, the sports apparel and shoe retailer. Nike stocks equals df, selecting from the original data frame, where the ticker equals “N K E”. Go ahead and sample 10 of these randomly, and it looks like they are indeed all Nike stocks. Now, set the date time to be the index so you can work with this time series data more easily. Use the set index method, datetime, in place equals true, and looks good. Go ahead and plot the closed price with a line graph. And you can see there's a big surge in 2021, 2022, then it falls back down, and then in late 2024, the price is roughly on par with pre pandemic levels. This stock has a different trend than Apple’s. One thing you may be interested in is grouping this data by the year or the month or some other component of the date. For example, you may be interested in grouping the Nike stock price, by the year. How can you access the year from somewhere in this data? **[pause for thought]** From the date time. You have your data frame, Nike Stocks, dot index, so index is this date time here, which represents how you access each value. Dot year. So year is an attribute of the date time series. Shift enter, you can see that you start out with 2014, all the way to 2024. So you can group by the year. You can also group by the month, if you're interested in that as well, starting in January and ending in October. Let’s start with the year first to get the overall trend. You can take your data frame, dot group by, and use this value for the year. Notice that you can group by a specific column of a data frame, or input a series that specifies the group by category. The requirement is that if you do input a series, it has to be the same length as the data frame that you are grouping. Now remember, if you hit shift enter, this is going to give you that weird group by result. You have to actually do some kind of calculation to see any values. Select the close price and dot describe. What are the features of the close price for all of those years? There's roughly the same number of values for each year and the mean close price seems to follow the same overall trend as you saw in the data. If you have these columns grouped by the year, what could you plot? Well, instead of saying dot describe, you can use any of these methods up here. So quantile or min, mean, count, and so on. For example, you could say dot mean, and that will give you the mean value for each year. Then you can even dot plot kind equals line and that will give you this simplified yearly trend.

	Say you want to group the data by month; maybe you think that the stock will go up towards the end of the year because, that's when people are buying more shoes as gifts during the holidays. Borrowing the code you had before, can you see how you would change this code to group the data by the month? **[pause for thought]** You want to do nike stocks dot index dot month, instead of dot year, shift enter, and you can see in general the lowest average value is in the summer months, and the stock value peaks in the later months of the year. Going back to your original data frame again, with all the stocks, you could also group these stocks by the sector. So df dot group by sector, and what do you want to do? Maybe you want to select the close price from there and average it for each sector. It looks like Healthcare Services has the highest average stock price, and Electronics seems to have the lowest. Remember, these are the 20 largest companies in the S&P 500, so healthcare stocks are not necessarily the most expensive stocks overall. Taking this operation one step further, you could also sort these averages using the sort values method. You may think, well, what column do I sort by? But since you only have one column here, You can just say sort values with no arguments, and that will sort by the mean price here. So finance is actually the lowest, I said electronics earlier, but it's actually finance, so sorting helps a lot. Electronics, telecom, at the bottom, and then materials and processing and healthcare services are at the top.
 Recap • Access different aspects: DataFrame nike_stocks.index.year Column df["date"].year • Group data by feature of the date: nike_stocks.groupby(nike_stocks.index.year)["close"].mean().plot(kind="line") nike_stocks.groupby(nike_stocks.index.year)["close"].mean().sort_values() DeepLearning.AI Sean Barnes Whatever you want to access 140 120 100 80 60 40 2014 2016 2018 2020 2022 2024 datetime	To recap, you can **[CLICK]** access different aspects of the datetime in your time series using the name of your **[CLICK]** data frame, **[CLICK]** dot index, then **[CLICK]** dot year or dot month or whatever attribute you'd like to access. Alternatively, if your data is stored in a column, like **[CLICK]** df["date"], you can just select that column, then **[CLICK]** dot year or dot month, **[CLICK]** et cetera. Then, you can **[CLICK]** group your data by that feature of the date, like grouping by year or month. You can use the **[CLICK]** dot group by method you saw previously, have the **[CLICK]** year for example as the argument to that function. Then you can select a **[CLICK]** column to apply dot describe or **[CLICK]** dot mean. And lastly, you can then **[CLICK]** graph one of these results in a line chart, or **[CLICK]** sort the results in ascending or descending order.
🎙️ TH	Segmenting time series, like segmenting stocks by sector or by some time period like a year or month, are useful strategies for extracting multiple series in your data. Follow me to the next video to see how to plot those series in a multiple line chart!

L1v6 – Multiple line charts – no reshaping

Visual	Script
🎙️ TH  Time series & visualization Multiple line charts – no reshaping DeepLearning.AI	Your strategy for creating a multiple line graph depends on what you're trying to accomplish. You'll see in a moment that most of the work is really in making sure your data is in the right format for plotting.
🖥️ Screencast 🔗 **c3m3l1v6 multiple line charts no reshaping**	You saw one way to compare different values across categories. You may also be interested in graphing multiple lines on the same chart, which you can do with Pandas. Remind yourself of your dataset. You can do a relatively straightforward plot by taking one stock and graphing multiple values. For example, if you wanted to continue examining Nike stocks, you might be interested in the difference between the high and low prices. You can graph those together to see how closely they correlate. To plot one line which you did previously, you used Nike stocks, then you would select the column you want to plot like high, for example, dot plot. In the case of plotting multiple lines from the same data frame, you'll want to start with the data frame rather than the column, which is a series. Because then you have access to all of the columns associated with each unique date time. So, to plot multiple lines, say high and low, start with Nike Stocks dot plot. And then you can use the y named argument with a list of column names that you want to plot. So high and low, for example. Make sure your parentheses are closed and matching. Shift enter. Now you get a graph with multiple lines. You can see that the high and low prices correlate very, very closely with each other.
 Recap `nike_stocks["high"].plot(y=["high", "low"])` - **Tells the computer to**: Plot each columns on the same x axis of datetime DeepLearning.AI Sean Barnes	To recap, you started with the **[CLICK]** data frame, **[CLICK]** dot plot, then for the **[CLICK]** named argument y, whatever you want on the y axis, you used a **[CLICK]** list of columns. That **[CLICK]** tells the computer to **[CLICK]** plot each of these columns on the same x axis of datetime.
🎙️ TH	You've learned a lot of great tools for analyzing and plotting time series data. Follow me to the next video to process some of the complex commands you've seen so far in this module. It will give you some tools to break down some of the long lines of code you've encountered. I'll see you there!

L1v7 – Method chaining

Visual	Script
🎙️ TH **Data structures and descriptive statistics** Method chaining DeepLearning.AI	In the previous few videos, you saw some pretty intimidating looking lines of code that involved calling multiple methods in a row. You'll see this strategy – called method chaining – a lot as you continue to work with Python, you'll see where it can help you make your code more efficient to write!
Expression: (((5+3)×2)−4)÷2  **Method chaining** - Process of linking several operations in a row - Each of which depends on the previous one having been solved - **Analogy**: Connecting operations like links in a chain $(((5 \boxed{+} 3) \boxed{\times} 2) \boxed{-} 4) \boxed{\div} 2$ $((8 \times 2) - 4) \div 2$ $(16 - 4) \div 2$ $12 \div 2$ 6 DeepLearning.AI Sean Barnes	Check out this math expression. How would you go about evaluating it? **[pause for thought]** You have **[CLICK]** four operations you need to complete. You need to resolve the **[CLICK]** addition operation first before you can do anything else. So 5 plus 3 equals **[CLICK]** 8. You've reduced the complexity of this expression and now you can proceed with the **[CLICK]** multiplication. 8 times 2 is **[CLICK]** 16, so again you've resolved another level of complexity, and can now subtract four, which gives you **[CLICK]** 12, and **[CLICK]** divide by two, which gives you **[CLICK]** 6. In code, this **[CLICK]** process of linking several operations in a row, **[CLICK]** each of which depends on the **[CLICK]** previous one having been solved, is called method chaining. It's called chaining because you're **[CLICK]** connecting operations like links in a chain; each link depends on the previous one.
Code: nike_stocks.groupby(nike_stocks.index.month)["close"].mean().plot(kind="line")  `nike_stocks.groupby(nike_stocks.index.month)["close"].mean().plot(kind="line")` 1 Group the stock by month - **Input**: Dataframe - **Output**: Dataframe with rows grouped by months 2 Select only the "close" column - **Input**: Grouped dataframe - **Output**: The "close" column grouped by months 3 Calculate the mean close price of each group - **Input**: Grouped series - **Output**: a series, with the month number as the index and the mean close price as the value 4 Create a line plot of the series - **Input**: A series - **Output**: A series, with the month number as the index and the mean close price as the value DeepLearning.AI Sean Barnes	Let's break down this line of code together, which you saw in a previous video. When you're reading and trying to understand this code, think of it as happening from **[CLICK]** left to **[CLICK]** right. Try to focus on your data types. The entire point of this line of code is to take some data, execute several commands on it, and end up with some type of result, in this case, a visualization. You start with the leftmost data type – a **[CLICK]** data frame – and end up with the output of the rightmost comman – a **[CLICK]** plot. This line of code has **[CLICK]** four steps, and each one needs to be resolved before moving on to the next one. - The first step is to **[CLICK]** group the stocks by month. The input to this step is a **[CLICK]** data frame and the output is a **[CLICK]** data frame with the rows grouped by months, also called a **[CLICK]** "groupby object". So you have 12 groups, one for each month. - Second, **[CLICK]** select only the "close" column. The input is a **[CLICK]** grouped data frame and the output is a series: just the **[CLICK]** "close" column grouped by months, so 12 groups. - Then, **[CLICK]** calculate the mean closing price of each of these twelve groups. The input is a **[CLICK]** grouped series and the output is **[CLICK]** a series, with the month number as the index and the mean closing price as the value. - Finally, **[CLICK]** create a line plot of the resulting series. The input is the

[CLICK] series from the previous step, and the output is **[CLICK]** a plot that can be displayed.

Each step takes the data from the previous one and transforms it further.

```
# Step 1: group stocks by month
stocks_by_month = nike_stocks.groupby(nike_stocks.index.month)

# Step 2: select close column
stocks_by_month_close = stocks_by_month["close"]

# Step 3: calculate mean close price for each group
stocks_by_month_close_mean = stocks_by_month_close.mean()

# Step 4: plot mean close price
stocks_by_month_close_mean.plot(kind = "line")
```

Using variables for each step:
- Useful if you needed to save intermediate step
- More flexible but takes longer

Method chaining:
- Get to plot as quickly as possible
- Not very flexible
- If you wanted to stop at intermediate step, you wouldn't be able to

DeepLearning.AI

Sean Barnes

Here's an alternate way you could have written this code, **[CLICK]** using variables for each step.

```Python
stocks_by_month = nike_stocks.groupby(nike_stocks.index.month) # Step 1: group stocks by month
stocks_by_month_close = stocks_by_month["close"] # Step 2: select close column
stocks_by_month_close_mean = stocks_by_month_close.mean() # Step 3: calculate mean close price for each group
stocks_by_month_close_mean.plot(kind="line") # Step 4: plot mean close price
```

All these variables could be **[CLICK]** useful if you needed to save these intermediate steps for later. For example, maybe you want to calculate the median and standard deviation of stocks by month close.

But, you don't really need to save these intermediate steps each time. Think about **[CLICK]** method chaining as taking an **[CLICK]** express train, in this case from data frame to plot. On the express train, you buy your ticket, board the train at the dataframe, and arrive at your destination: the plot. That's great if you're just trying to **[CLICK]** get to the plot as quickly as possible. It's **[CLICK]** not very flexible, though, so **[CLICK]** if you realized you wanted to stop at an intermediate step, you wouldn't be able to.

The approach with variables is more like the **[CLICK]** local train that stops at each stop. It's **[CLICK]** more flexible but takes longer. You can get off at any particular point if you need to, like if you wanted to calculate the median and standard deviation of closing prices by month. But if you don't have to stop anywhere, method chaining can get the job done quickly.

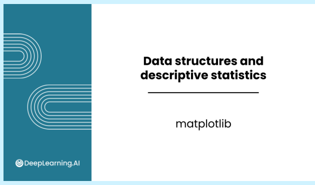
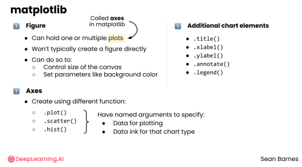
🎙️ TH

Chained methods can be challenging to read at first, but the more you work with them, the better you'll get at understanding them. They can make your coding quick and efficient, especially if you don't have a need to store intermediate results.

That takes you to the end of this lesson! Up next, you'll complete the practice assignment and practice lab for this lesson. Once you're done, follow me to the next lesson to create and customize visualizations using a popular Python library: matplotlib. See you there!

Lesson 2 – Creating & customizing visualizations with matplotlib

L2v1 – matplotlib

Visual	**Script**
🎙️ TH  Data structures and descriptive statistics matplotlib DeepLearning.AI	Before you start creating your own visualizations with matplotlib, you'll need to understand its design philosophy.
 **matplotlib** **Figure** • Can hold one or multiple plots • Won't typically create a figure directly • Can do so to: ○ Control size of the canvas ○ Set parameters like background color Called **axes** in matplotlib **Additional chart elements** • `.title()` • `.xlabel()` • `.ylabel()` • `.annotate()` • `.legend()` **Axes** • Create using different function: ○ `.plot()` ○ `.scatter()` ○ `.hist()` Have named arguments to specify: • Data for plotting • Data ink for that chart type DeepLearning.AI Sean Barnes	Matplotlib visualizations are made up of a few core components. First, the **[CLICK]** figure, which you can kind of think of like a canvas for drawing. A figure **[CLICK]** can hold one plot or multiple plots. It's essentially a container. Plots are **[CLICK]** called axes in matplotlib. Each "Axes" has its own set of plot elements, like labels, lines, legends, a grid, and so on. You **[CLICK]** won't typically need to create a figure directly, but you **[CLICK]** can do so in order to **[CLICK]** control the size of the canvas, or **[CLICK]** set other parameters like the background color. You'll create a set of **[CLICK]** "Axes" using **[CLICK]** different functions that create what you'd think of as graphs or charts, like **[CLICK]** .plot(), **[CLICK]** .scatter(), **[CLICK]** .hist(), and so on. These functions also **[CLICK]** have named arguments for you to specify the **[CLICK]** data for plotting as well as the **[CLICK]** data ink for that chart type, like the axes, markers, colors, line style, and other arguments. Then you can also layer on **[CLICK]** additional chart elements, with functions like **[CLICK]** .title(), **[CLICK]** .xlabel(), **[CLICK]** .ylabel(), **[CLICK]** .annotate(), **[CLICK]** .legend() and so on. Let's see that in action.
🖥️ Screencast 🔗 [c3m3l2v1 matplotlib]()	Let's start with a visualization of Netflix stock prices. First, import Pandas as PD and read in the same stock prices data from before. Then, you're going to conduct some of the same transformations that you did previously, such as selecting only the Netflix stocks, changing the date column to a date time and assigning that to the index, and then dropping the unnamed and date columns. You get something very similar to the data you had previously, except this data represents the Netflix stock price time series rather than Apple or Nike. Let's say you want to plot the adjusted close price directly with pandas. You can say Netflix stocks, select adjusted close, dot plot kind equals line, and that will give you this graph of Netflix's stock price over time. You can see that the beginning of 2022 was a really tough time, but Netflix has gradually grown and

surpassed its previous peak stock price.

Now, this is a pretty bare chart. You can enhance it using matplotlib. So let's borrow this line of code, and import matplotlib dot pyplot as P L T. Remember, as P L T creates an alias; this is a very, very common alias for matplotlib, it just makes the name shorter.

You can start with the same chart as you had before, and now add more layers to it. If you look at this chart, there's already a figure here, or a canvas where you can draw a plot, and there's already a plot. You can enhance this plot using, for example, PLT dot title: this is going to stack a title on top of this graph as another layer. Maybe adjusted close price for Netflix, 2014 to 2024.

You can also say PLT dot x label to add or update the x-axis label. There's already a label there, date, but maybe you want it to be capitalized. PLT dot y label could be adjusted closing price. Now you've layered on these other elements to your chart that just make it a little nicer and easier to interpret.

Another interesting thing you can do for a line chart is to label different points with an annotation. Suppose that you wanted to highlight the highest adjusted closing price, you could use PLT dot annotate. To find the highest point, you could sort by adjusted close. That gives you these points. The highest is October 21st, 2024. Borrow the same code as before, minus the import, since you only need to import once per notebook.

Now you can add PLT dot annotate. Notice that you have to stack these functions every time to create your chart in a new cell. Let's say you're adding the highest point, text equals peak, and then you can say the x, y coordinates you're interested in. The peak was at these x, y coordinates. Date is a string, adjusted close is a float, and just like you would in math, you want to enclose the x-y point in parentheses. Shift enter! It overlaps a little with the edge of the chart, but you do have the peak, or the highest point, labeled.

If you are concerned about readability and being able to measure the adjusted closing price for all of these different dates, you could also add a grid. PLT dot grid. The first argument is true: yes, you want a grid. You also have a named argument, “which.” Which sets of gridlines would you like to display? You can pick major, minor, or both, so pick both. There is also a named argument for which axes you want to display gridlines for, which defaults to both the x- and y-axes. Shift Enter, and you get a grid for both x and y axes. Every two years on the x-axis, and every $100 on the y-axis.

One last cool thing about Matplotlib is you can now save your figure to a file. Use PLT dot save fig.The only argument you need is the filename you want to save it to, including the file extension, or type. The figure will be saved as an image in the same location as your notebook. Maybe you want to save this

	image to a file called netflix line chart dot png. Shift Enter. And now you have this nice image that you can include in a report or a presentation.
🎙️ TH	Matplotlib adds a lot of utility and power to your visualizations. It's great for customization, whereas your pandas-only visualizations were great for just getting the chart on the page. Follow me to the next video to see some great enhancements for line charts.

L2v2 – Text and annotations

Visual	**Script**
🎙️ TH	You've seen how to create and annotate a chart in matplotlib, but so far its visual design and organization left something to be desired! Matplotlib has a ton of customization options. Let's explore these starting with tools for text and annotations
🖥️ Screencast 🔗 **c3m3l2v2 text and annotations**	In the previous video, you developed this line chart showing the adjusted close price for Netflix over a decade, labeled the peak, and added gridlines. Let's finesse the design of this graph. First, for any method with text, like title, xlabel, ylabel, or annotate, you can also adjust the font size. You can use the named argument font size equals 14, to make the font a little bit larger, in this case for the title. And same thing for the x and y axis labels. You might want to adjust the font size to be a little bit larger. Now, briefly, for the line, you can change a lot of aspects by adding more arguments to this plot method. For example, maybe you want the line to be in the Netflix brand color. You can use the color named argument. You have a couple of options for how to specify color. You can just use a word like "red", though that's not exactly the Netflix red. You can also use a hex code, like this one. If you're not familiar with hex codes, they specify the amount of red, green, and blue in the color. So, that hex code gives you the Netflix red. You already saw that the default for color is blue, so if you leave out a lot of these named arguments, like color, font size, and so on, you'll just get the default settings for a matplotlib chart. Another option you have is the line width. Maybe you want line width equals 2, so a little bit thicker. If you do 0.5, that gives you a thinner line width. The thinner width does reveal more detail, so that's an option that level of detail is important to the story. I'll put the line width back at 2. For the title, in addition to the font size, you might want to make this bold and

set it apart from the chart so it's more legible, so you can say font weight equals bold in quotes, and maybe you want pad equals 15, that just gives you a little more space, called “padding”, between the chart and title.

Let's take a look at this annotation. You saw in the previous video how it's kind of overlapping the area of the chart. There’s also room to make it more useful, since you can already see the highest value on the chart.

First, update this label to actually tell you the highest value. All time high. You can use backslash n for a new line. All time high, $772.07.

Right now, you have this larger piece of text and it's extending past the side of the plot. X Y is the argument that allows you to specify the point you are actually annotating. But the location of the text is separate from the location of the point that you're annotating. You have another argument, X Y text, which is also a set of X Y coordinates, that allows you to move this text around so that it's not going beyond the chart.

It looks like you’ll want to move this annotation to the left and down. You can say, negative 100, negative 30, just as a guess. These numbers are in points, which is a very small unit of distance. The first number adjusts the x value, while the second adjusts the y value.

You'll also want to use the named argument text coords equals offset points, which tells the computer to interpret this X Y text argument as offsetting the location of this text box from the annotated point rather than giving it an absolute value. Shift enter, it looks like you moved the box down correctly, but it's a little too far to the left.

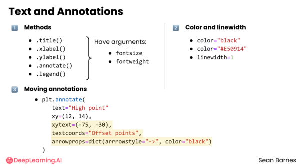
Remember that the first value is the x value. So maybe you want negative… well, not negative 750! Negative 75 looks quite nice.

Now, since you have moved this text away from the annotated point, you may want to indicate this point it’s referring to. Matplotlib’s annotate function has a feature to add an arrow, but it's a little complicated to use.

So taking it over to chat GPT, you can say modify this code to add an arrow pointing to the point from the text. And paste in the code.

So it looks like you can use the arrow props parameter. Arrow props stands for arrow properties, and it uses a dictionary of properties, which personally I can never remember. You can just go ahead and hit copy code. Head back over to your notebook and paste in the expanded code.

Shift enter, and there you go, you get a cute little arrow indicating the all time high price that corresponds with the annotation.

 **Text and Annotations** 1 **Methods** • .title() • .xlabel() • .ylabel() • .annotate() • .legend() Have arguments: • fontsize • fontweight 2 **Color and linewidth** • color="black" • color="#E50914" • linewidth=1 3 **Moving annotations** • plt.annotate(text="High point" xy=(12, 14), xytext=(-75, -30), textcoords="Offset points", arrowprops=dict(arrowstyle="->", color="black")) DeepLearning.AI Sean Barnes	To recap, you saw some methods for styling text and basic chart elements, as well as creating annotations. For text, you saw that **[click]** methods like **[click]** title, xlabel, and others with text have **[click]** options like **[click]** fontsize and fontweight to adjust the size and style of the font. **[click]** Then, you saw how to style the color of the line chart, using the **[click]** color argument, with a color name or a hex code and how to set the linewidth using the **[click]** linewidth argument. You also saw that you can **[click]** move the annotation text independently of the point you're annotating. To do that, you used the **[click]** xytext argument to specify how you wanted to move the text relative to that point, in this case 75 points to the left and 30 points down. You needed to specify that the **[click]** textcoords argument was "offset points" so you didn't end up setting an absolute value for the text location. And you used an LLM to create a nice looking **[click]** arrow to point to the annotated location on the graph.
🎙️ TH	Once you've labeled your graph nicely, adjusted the appearance of the line, and created a visually appealing annotation, you'll want to customize your axes to better show the time series information you're working with. Follow me to the next video to see how.

L2v3 – Date and axis formatting

Visual	**Script**
🎙️ TH	Dates are a complex data type, but they're important to get right in a time series visualization. Matplotlib gives you options for formatting them that can come in handy.
🖥️ Screencast 🔗 c3m3l2v2 date and axis formatting	When you're working with dates specifically, one thing you may want to do is change the x axis, which is the date. To make the graph easier to read, you might want to display every year on the x axis, rather than every other year. Dates are a pretty complex type of data. Matplotlib has helpful date tools that you can import so you don't have to start from scratch. It's very common to call them M dates. Go ahead and borrow the code you had from earlier. Now, in order to change the axes easily, you'll want to have a variable for it. Remember that in matplotlib, the axes are basically a plot. So when you create the plot, you actually want to save the plot to a variable.

You can just say A X equals, and that's just going to save the plot into this variable A X, for axes. Running this cell, the same plot is displayed, but now you have the axes saved to this variable A X. So now you can change aspects of the axes plot more easily.

Now you can adjust the x axis to show each year. You wanna say A X, so you're taking the axes, dot x axis, because you want to select the x axis. Then setMajorLocator, and then you want the locator to be M dates dot yearLocator.

Set allows you to change what is actually on the axis, as opposed to just getting some information about it. Major refers to the major, or primary, tick marks. Right now the plot doesn’t have any minor tick marks. Locator means you’re looking to change where the information is displayed. So, you're actually saying, here is where I want those major axis ticks to be. For that, you're gonna use the years, so M dates dot year locator with no arguments will place ticks automatically at yearly intervals.

Shift Enter, and now you have each year plotted on the x axis.

Notice that your grid updated to be shown for each year as well. So this new axis makes it easy to compare the start of 2020 and the start of 2021, and you can see that the first half of 2022 saw a sharp decline in the stock price.

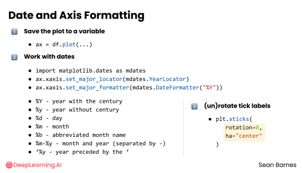
The last thing you may be interested in doing here is changing the x axis to show just the last two characters of the year,14, 15, and so on, just to make it a little more readable, since you already have the years in the title. You can copy and paste your code from the previous cell; now you want to use a very similar line of code to the locator. Grab the x axis, and this time you want to set major formatter.

Just like with the locator, set means you’re changing something. You're doing it for the major tick marks, and now you just wanna change the formatting, not their position. So you can say M dates dot date formatter. And then you have some options that are specified in a string. It’s actually a special type of string called a date formatter. You’ll learn more about the options in a moment. As one example, you could enter percent capital Y. Percent starts a format code, and the capital Y specifies a 4-digit year for each value.

Oh, ran into an error there because I misspelled dates.

So capital Y gives you the exact same format as you had before. Lowercase Y just gives you the last two digits. And if you want to add an apostrophe at the beginning, running this cell will give you the apostrophe and then the last two digits of the year.

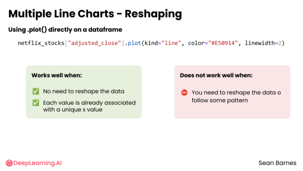
	Finally, because these dates are shorter, you may want to change the rotation and alignment of these values. Right now they're quite rotated, which is no longer necessary because there’s plenty of room. To adjust the rotation of the x axis labels, you can enter plt dot x ticks, rotation equals 0. Now you have a little bit of an easier time reading these. Then you’ll want to horizontally align the labels. Right now they're kind of right aligned which is a little harder to read. In the same line of code for x ticks here, you have another named argument, ha for horizontal alignment equals center. Run this cell, and now you have a really nice looking chart, great annotation, and the x axis is very, very clear!


Date and Axis Formatting
1️⃣ **Save the plot to a variable**
• `ax = df.plot(...)`
2️⃣ **Work with dates**
• `import matplotlib.dates as mdates`
• `ax.xaxis.set_major_locator(mdates.YearLocator)`
• `ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))`
• `%Y - year with the century`
• `%y - year without century`
• `%d - day`
• `%m - month`
• `%b - abbreviated month name`
• `%m-%y - month and year (separated by -)`
• `‘%y - year preceded by the ‘`
3️⃣ **(un)rotate tick labels**
• `plt.xticks(`
`rotation=0,`
`ha="center"`
`)`
DeepLearning.AI
Sean Barnes
	Summing up, you learned how to **[click]** save the result of a plot method into a variable so you can change the axes later.
	
	You saw that to **[click]** work with dates, **[click]** you can import matplotlib dot dates. You used the **[click]** set major locator with m dates dot year locator to add each year on the x axis, and you used **[click]** set major formatter to control how the dates were formatted on that axis.
	
	You saw a moment ago this date formatter string with percent signs. You used the **[click]** % capital y and **[click]** % lowercase y date codes, which show the year with or without the century. You have other options too, like **[click]** % m to show month as a zero-padded number, or **[click]** % d for day, or **[click]** % b for abbreviated month name. You can also combine these together, for example by showing **[click]** % m % y to show the month and year on the x axis. And you used a **[click]** single apostrophe to add the apostrophe character directly to the format.
	
	Finally, **[click]** you used the P L T dot x ticks method to unrotate the labels and center them horizontally.

🎙️ TH	You’ve learned a lot about matplotlib plotting functions. In order to unlock more complex plots, you’ll need to get familiar with reshaping your data; in other words, preparing the right rows and columns for plotting. Follow me to the next video to see how to do that with a multiple line chart.

L2v4 – Multiple line charts – reshaping

Visual	**Script**
🎙️ TH	Often the difficulty in plotting your data is in getting it into the right format for the plot, rather than in specifying the plotting methods. You saw previously how to graph multiple columns in the same line chart, but what if you wanted

<table>
<tr><td></td><td>to graph multiple stocks instead?</td></tr>
<tr><td>

Multiple Line Charts - Reshaping

Using .plot() directly on a dataframe

```
netflix_stocks["adjusted_close"].plot(kind="line", color="#E50914", linewidth=2)
```

Works well when:

✅ No need to reshape the data
✅ Each value is already associated with a unique x value

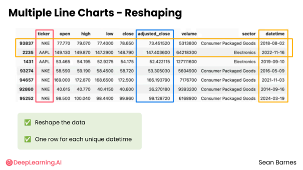
Does not work well when:

⛔ You need to reshape the data o follow some pattern

DeepLearning.AI

Sean Barnes</td><td>In an earlier video, you <b>[click]</b> used the dot plot method on your dataframe directly with a set of columns for the y named argument.

<b>[click]</b> That approach worked because you <b>[click]</b> didn’t need to reshape your data. <b>[click]</b> Each value you want to plot on the y axis was already associated with some unique x value – the date time.

However, <b>[click]</b> if you want to create a more complex plot, you’ll sometimes need to <b>[click]</b> reshape your data so that it follows this pattern – the datetime as the index, with each column containing some data you want to plot.</td></tr>
<tr><td>

Multiple Line Charts - Reshaping

<table>
<tr><th></th><th>ticker</th><th>open</th><th>high</th><th>low</th><th>close</th><th>adjusted_close</th><th>volume</th><th>sector</th><th>datetime</th></tr>
<tr><td>93837</td><td>NKE</td><td>77.770</td><td>79.070</td><td>77.4000</td><td>78.650</td><td>73.451520</td><td>5313800</td><td>Consumer Packaged Goods</td><td>2018-08-02</td></tr>
<tr><td>2235</td><td>AAPL</td><td>149.130</td><td>149.870</td><td>147.2900</td><td>148.790</td><td>147.403600</td><td>64218300</td><td>Electronics</td><td>2022-11-16</td></tr>
<tr><td>1431</td><td>AAPL</td><td>53.465</td><td>54.195</td><td>52.9275</td><td>54.175</td><td>52.422115</td><td>127111600</td><td>Electronics</td><td>2019-09-10</td></tr>
<tr><td>93274</td><td>NKE</td><td>58.590</td><td>59.190</td><td>58.4500</td><td>58.720</td><td>53.305030</td><td>5604900</td><td>Consumer Packaged Goods</td><td>2016-05-09</td></tr>
<tr><td>94657</td><td>NKE</td><td>169.000</td><td>172.870</td><td>168.6500</td><td>172.500</td><td>166.193790</td><td>7176700</td><td>Consumer Packaged Goods</td><td>2021-11-03</td></tr>
<tr><td>92860</td><td>NKE</td><td>40.615</td><td>40.770</td><td>40.4150</td><td>40.600</td><td>36.270180</td><td>9393200</td><td>Consumer Packaged Goods</td><td>2014-09-16</td></tr>
<tr><td>95252</td><td>NKE</td><td>98.500</td><td>100.040</td><td>98.4400</td><td>99.960</td><td>99.128720</td><td>6168900</td><td>Consumer Packaged Goods</td><td>2024-03-19</td></tr>
</table>

✅ Reshape the data
✅ One row for each unique datetime

DeepLearning.AI

Sean Barnes</td><td>As an example, say you want to plot the adjusted close price for Apple and Nike stocks, with Apple and Nike getting their own line on the chart. That way you can compare the adjusted close price for each company.

However, <b>[click]</b> if you take a look at the original data frame, your data isn’t yet set up properly. If you just look at the <b>[click]</b> datetime, <b>[click]</b> ticker, and adjusted_close <b>[click]</b> column, you have a situation where each unique day has two rows, so date time is repeated. And the adjusted_close for Nike and Apple appear in different rows.

What you need in order to graph this properly is to somehow <b>[click]</b> reshape this data so that there is only <b>[click]</b> one row for each unique date time, and the columns in that row are the adjusted_close price for apple, and the adjusted_close price for nike.

Let’s take a look at how to do that.</td></tr>
<tr><td>🖥️ Screencast

🔗 <u>c3m3l2v4 multiple line charts with reshaping</u></td><td>Let’s go back to your original data frame, to work again with the Apple and Nike stocks rather than the Netflix stocks. Using D F ticker dot unique, remember you had all of these different stocks available to you in the data set. Say you wanted to graph the performance of Apple against the performance of Nike. You already looked at those separately and saw that they were pretty different, but what if you want to graph them together?

The first step is to filter for the tickers you're interested in. In this case, you have a list, it's not just one: Apple and Nike. Now you can filter your data frame using both of these tickers. Previously you saw how to filter for one ticker using df, df ticker equals equals something like Apple.

To filter for multiple tickers, you can use the “is in” method. You have something very similar to this previous command. Take the data frame. Select from it, where the ticker is in this list “tickers.”</td></tr>
</table>

What's going to happen in this line of code is the computer is going to take the series for ticker, which is one column, look at each value (apple, U P S, and so on), and check if that value is in this list of tickers, which in this case is Apple and Nike. So if the ticker is Apple or Nike, it will be kept in this filtered data frame. If not, it will be thrown out. You can call this filtered data frame Apple and Nike Stocks. Great.

Now, what you may want to do is check the tickers in Apple and Nike stocks, just to see if that operation happened successfully. So, running this line of code to get all the unique values in the column ticker, what do you expect? Ah, I had to fix a typo here in the Apple stock ticker, but you should expect only two tickers, Apple and Nike. So the “is in” method will allow you to select multiple columns using a list.

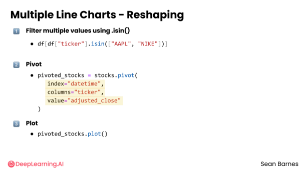
Now that you've filtered for only the Apple and Nike stocks, you need one more step to set up your data, which is to pivot the data. Start by sampling from this data frame so you can get a mix of values. If you're interested in the adjusted close for Apple and Nike, then what you really want to have is these columns: datetime, Apple, Nike. And then you would have some date, like August 2nd, 2018, and some price for Apple, say 10, and some price for Nike, like 5, whatever it is. If you can imagine two lines being plotted on the same graph, this format would be ideal because you have two y values for each x value; the two y values correspond to the two lines you’re plotting: Apple and Nike.

In order to reshape your data in this way, you need to pivot your data. And pivot is a little bit different from a pivot table. A pivot table summarizes your data, but the pivot method is going to reorganize your data across rows, columns, and values.

How would you do that? You need to say apple and nike stocks dot pivot. For each row, or the index, you want to have the datetime. For the columns, you want to have the ticker. You've only got two unique values in ticker, so now you're just going to have two columns in your dataset: apple and Nike. And for the values, you want to have the adjusted close.

You’ll want to save this into a variable like pivoted stocks because it creates a new data frame rather than changing the old one. It’s a very long line of code, I know, but if you take a look at pivoted stocks dot head, now you have exactly the format of data you wanted. For each date, you can then graph the adjusted close price for Apple and the adjusted close price for Nike.

Now you can just say pivoted stocks dot plot shift enter, and you get this really nice looking graph. You can see that both stocks were increasing in the first half of the graph, but then they started to diverge. Apple continued going up while Nike started to trend downwards. Now you can continue by adding axis

	labels, annotations, and so on to your plot, but you can see that most of the work here was in reshaping the data to get it in the form that you needed.
 Multiple Line Charts - Reshaping 1️⃣ Filter multiple values using .isin() • `df[df["ticker"].isin(["AAPL", "NIKE"])]` 2️⃣ Pivot • `pivoted_stocks = stocks.pivot( index="datetime", columns="ticker", value="adjusted_close" )` 3️⃣ Plot • `pivoted_stocks.plot()` DeepLearning.AI Sean Barnes	To quickly recap, **[click]** you used the is in method to filter for multiple values in a column. That method checks if a particular value is in a list you specify. You also used **[click]** pivot with **[click]** datetime as the index, **[click]** ticker as the columns, and **[click]** adjusted close as the values to reshape your data, with the goal of having multiple y values for each x value. Once you reshaped your data, you could call **[click]** dot plot on your data frame and get a nice looking plot that you can continue to finesse with matplotlib.
🎙️ TH	Excellent work reshaping your data to plot multiple series in one chart. You've been working with line charts for a while now, but matplotlib is great for all kinds of plots. In the next video, you'll see how to create a column chart. I'll see you there!

L2v5 – Column charts

Visual	**Script**
🎙️ TH	Line charts are great, but matplotlib has a lot more capabilities beyond time series data. One of your go-to chart types is the column chart, so let's see how to create them.
🖥️ Screencast 🔗 **c3m3l2v5 column charts**	Say you want to plot the volume of stock trades for all finance sector stocks on July 2nd, 2018. Just pick a random date for this exercise, any date. A column chart is appropriate because you have a numerical variable, volume, over a categorical variable, the ticker for each finance sector stock. First thing, you want to make sure that your index is the date time, so you can select from it correctly. You've already dropped the date and unnamed columns, and here are the first five rows in this data. Next, you want to select the finance stocks, so the stocks where sector equals finance. If you look at the ticker dot unique, you get the four different finance stocks: Bank of America, JP Morgan, Morgan Stanley, and Wells Fargo. You can call these your fin stocks. Go ahead and sample 10 of those. You can see you've selected only the stocks that you expected based on the tickers and they all appear to be finance. Your next step is to filter it down for the date. Call this July 2nd stocks. Which will be fin stocks dot loke, and then select 2018, 7, 2. Oh, it should be fin not fine stocks. Now, how many rows do you expect in July 2nd stocks? **[pause for thought]**

Well, it's only four rows and that's because now each row is unique to the date and the ticker.

To make your column chart, start with July 2nd stocks. You want to plot on the data frame because if you select the column here, volume, like this, now you only have access to the volume and date columns, but not the ticker column. You need the ticker column to be able to label your columns. So dot plot, kind equals bar. Remember, there's no column chart. X is ticker and Y is volume. Now you have this nice looking bar chart. It looks great!

Now it’s time to jazz it up! P L T dot title, say, finance stock volume July 2nd, 2018. Then you can add a line of code to capitalize the ticker for the x axis label, and to add a y-axis label for trading volume, and maybe you want to increase the font size to 16 and 12 here. This is starting to look pretty nice.

Now, you may notice this color legend is pretty redundant. You don't actually need to distinguish between multiple series in this chart. So you can remove that using the named argument legend equals false in the initial plot command.

What else might you want to do? The x axis labels could be rotated from vertical to horizontal. There’s enough room for each label to fit. Use P L T dot x ticks, rotation equals zero, horizontal alignment equals center, just to make sure they're centered. And there you go, that looks much nicer.

Now, to have different colors for each of these tickers. First, you want to create a list of colors. Let's use the brand colors for each one. You can go and look these up online. Use the named argument color equals bank colors. It's a little confusing because color is singular while bank colors is plural. If you're going to set the color of each bar manually, you need to make sure there are enough colors in this list for the number of bars.

Now each bar is in the main brand color for that bank. Looking at it though, these colors are pretty similar. You could also make a list of more differentiated colors and use that color list instead. It's up to you.

You may also notice that this y axis for trading volume is in scientific notation with one to the seventh power. Some stakeholders may find that notation difficult to interpret. One option you have is to set the Y axis tick labels to be in non-scientific notation. Use P L T dot tick label format, style equals plain. And you can say axis equals y so it applies to the volume axis.

This is possibly a little bit easier to read, but the natural next step is to add commas. That’s a more complicated operation. The perfect time to head over to your L L M! How do I add commas to the Y-axis labels in this chart?

	It's gonna tell you to use funk formatter. Import it from mat plot lib dot ticker, define a new function with commas and so on. It shows you what the output should be. You can look this code over, but the proof is in the output. My suggestion would be to just copy that code, paste it in, and see what happens. This code displays a strange result with two different plots. Don't fret! Try to troubleshoot. Go back to the LLM, say this isn't working, then tell it what the problem is as precisely as possible. I see an empty set of axes followed by the plot I want, but without the commas in the Y labels. Okay, so it writes some more code, then explains the changes, but you don't need to read all that. Go ahead and try one more time. You can remove the imports, since these have already run. And there you go, that looks much nicer! So now you have the trading volume, 10, 20, 30, 40 million on the y axis. The last thing you may want to do is order these columns by volume. You already know how to sort. Grab the code from before, plus July 2nd stocks dot sort values by volume. Ascending is the default, in place equals true. That way you can use the same variable name. Now you get a nice sorted list of values, and a pretty good looking graph in my opinion.

Column charts
1️⃣ **Selecting the data**
• `fin_stocks = df[df["sector"]=="Finance"]`
• `july_2nd_stocks = fin_stocks.loc["2018-07-02"]`
2️⃣ **Adjustments to .plot()**
• `july_2nd_stocks.plot(`
`kind="bar", x="ticker", y="volume",`
`legend=False,`
`color=list_of_colors`
`)`
3️⃣ **Labels**
• `plt.ticklabel_format(style="plain", axis="y")`
DeepLearning.AI
Sean Barnes
	You started this demo by **[click]** selecting the data you wanted to plot in your column chart. You'll want to have a numerical feature to plot over a categorical feature.
	Then, **[click]** you used **[click]** legend equals false to remove a redundant legend from your plot, plus the **[click]** named argument color to add a list of custom colors to the chart.
	You also saw how to **[click]** change the ticklabel format for the y axis to be plain numbers rather than scientific notation, and used an LLM to create a custom y axis label format that made those large numbers easier to read.

🎙️ TH	Once you're ready to explore relationships in your data, you'll want to create scatter plots. Follow me to the next video to see how it's done.

L2v6 – Scatter plots

Visual	**Script**
🎙️ TH	Matplotlib allows you to make highly customized scatterplots that explore the relationships between features in your dataset.
🖥️ Screencast 🔗 c3m3l2v6 scatter	Starting from your data frame again you may be interested in calculating relationships between the prices of different stocks. One interesting line of inquiry is to look at the difference between the close and open prices of Apple

plots	and Google stocks, basically how much did the stock gain or lose that day. First, you’ll select the Apple and Google stocks, then create columns for the Apple and Google differences per day. Then you can go about plotting with a scatterplot. Call this filtered data frame Apple Google to keep things simple. Select from the data frame where the ticker is in a list. You need a list because you want multiple stocks: both Apple and Google. There are a lot of parentheses, brackets, and such. If you want, you can instead use a variable for this list: columns equals Apple, Google. If the ticker is in columns, then you're going to store that in the data frame Apple Google. Sample 10 rows to view it, and you just have Apple and Google stocks, which is exactly what you need. Now, to create a new column, you're going to say Apple Google. Then pick the column name. Start with Apple Diff. This creates a column called Apple Diff. How much money did the stock gain or lose each day? Select from Apple Google, where from that data frame, ticker equals Apple. Select the close column, and you can just perform simple arithmetic minus that same thing, except the open price. This piece of code takes the Apple Google data frame, which just has the Apple and Google stocks, it selects all of the rows where the ticker is Apple, and takes the close price, minus the open price for those same stocks and saves it in the Apple diff column. Minor typo. You can view some of those rows and you see this Apple diff column. So on August 30th, 2022, Apple lost about $3. You can do the same for the Google column, hit enter, now you have Apple Diff and Google Diff. To plot a scatterplot, start with the data frame Apple Google dot plot, this time kind equals scatter. If you just run this cell now, the computer won’t know what to put on the x and y axes. So for x, apple diff, for y, google diff, and you get this simple scatter plot. To zhuzh it up a little bit, you can start by making these points a little more transparent so you can see where they're overlapping. Alpha equals anywhere from 0 to 1; lower means more transparent points. 1 is just the original plot. So 0.3 is nice because you can more clearly see this dense mass right around this central area. You may also want to add a title and axis titles. One thing you may notice is that the x and y axis, though they’re on the same scale, their ticks aren’t spaced the same way, so it may be a little bit difficult to compare. Remember to mess with the axes, you need to save the plot into a variable, so A X equals that plot, shift enter, again you get the same plot, but now you can use A X to change the axes.

	Borrow this code here. You're going to need to import the ticker tools. So import matplotlib dot ticker as ticker. These tools just help you manage what goes on the axes with more specificity. Start with ax dot x axis dot set Major Locator. You used this method before when you were working with dates to make sure the x axis showed every year. Remember it changes what major ticks are displayed on the x axis. This time, use ticker dot multiple Locator. And say you want the x axis to have ticks every two dollars, the same as the y axis. Nice, now you get a little more granularity. Multiple locator is very useful if you include a grid. For P L T grid in this first version, the grid is unevenly spaced horizontally and vertically. If you add P L T dot grid here, now it's perfectly evenly spaced. Notice that the grid isn't perfectly aligned on the edges. You can control this behavior with the x lim and y lim functions. Borrow this code so that you can compare. P L T dot x lim, maybe you want negative 8 to 14. That gives you the range of values on the x axis. Then, you can add P L T dot Y lim, maybe from negative 8 to 8. Now the plot perfectly lines up with the grid on the edges. Wrapping up, maybe you want to set the color of your markers to purple. You can also use the marker named argument, which is a funny one. Circles with “o” are the default. You can use the carrot for triangles, or X to get exes.

Scatter plots
1 **Plot**
• `aapl_googl.plot(`
`kind="scatter", x="AAPL_diff", y="GOOGL_diff",`
`color="purple", alpha=0.5, marker="^",`
`)`
• `plt.grid(alpha=0.5, marker="^")`
2 **set_major_locator**
• `import matplotlib.ticker as ticker`
• `ax.xaxis.set_major_locator(ticker.MultipleLocator(2))`
3 **Setting limits**
• `plt.xlim(-2, 2)`
DeepLearning.AI
Sean Barnes
	To recap, you saw how to use the **[click]** dot plot method with **[click]** kind equals scatter to create a scatterplot. If you’re calling plot on a data frame, like you saw in the demo just now, you’ll need to **[click]** set the columns for x and y.
	
	You saw that you can **[click]** control the **[click]** color of the markers using color, the **[click]** transparency using alpha, and the **[click]** marker style using marker, with options like **[click]** carat for triangle or **[click]** x for exes. The alpha named argument can control the transparency for other chart elements as well, like P L T dot grid.
	
	You saw another use for the **[click]** set major locator method, this time by first **[click]** importing matplotlib dot ticker as ticker, then using **[click]** ticker dot Multiple Locator to create an evenly spaced grid every 2 dollars.
	
	Finally, you used **[click]** P L T dot x lim and y lim to set the axis limits, squaring up your plot. You can also use these methods to focus on a particular slice of the plot, for example only those differences between -2 and 2 dollars.

🎙️ TH	Great work with scatter plots. Follow me to the next video to see how to plot

	grouped and stacked column charts for segmentation analysis.

L2v7 – Grouped column charts

Visual	Script
🎙️ TH	Grouped column charts allow you to compare a numerical feature across multiple categorical data. You'll need to take some time to set up your data first before plotting.
🖥️ Screencast 🔗 **c3m3l2v7 grouped column charts**	Say you're interested in the average trading volume per year, and you want to compare both within and across different stocks. Let's take three stocks: Amazon, Meta, and Disney. You'll want to create a grouped column chart to look at the average trading volume as it changes throughout the years across these three different stocks. First you're going to want to select all three tickers. Then you'll need to group by each ticker and year, then aggregate by calculating the mean of the volume feature. Then plot with a bar chart. Starting by selecting the three columns, you can call this filtered df so you don't have to list out all the columns in the variable name. You can take df, where the ticker is in this list of columns right here. Make sure these values are strings. Use filtered df dot sample to check out the resulting data frame. Now you just have this filtered data frame of those selected stocks. Your next step is to group by each ticker and year. Just call this grouped by ticker and year. That's going to be the filtered Data Frame dot group By, and now you want to group by two columns. You can actually put a list here as the argument to group by. You want to group by the ticker, and you want to group by the year. So filtered DF dot index, because you have the Date as the index dot year. Since your date index is a Pandas datetime, you can use the dot year attribute to get just the year. This is a group by object, if you remember, and so you can't actually see it. It's an intermediate data structure. You'll need to do the next step first: aggregate the volume. Select the volume column and then add dot mean. If you take a look at a sample of ten of those, you have the ticker and the year, and the average volume for that ticker and the year. This feature is called a multi index in Pandas. If you try graphing this directly, plot kind equals bar, now you get these double indices and they're all separated into their own bars. So Amazon 2014, Amazon 2015, 2016, and so on. Same for Disney and Meta, so this isn't a true grouped bar chart, although you're quite close. What you'll need to do is

unstack this data.

Notice that you have two layers of indices here, the ticker and the date. You want to use the unstack method to pivot out the years into columns. Check out Average Volume dot unstack on its own: you have this really nice setup where each ticker has a value of the average volume for each year.

If you go ahead and plot that as a bar chart, now you get this nice chart of the years over time. Well, nice may be a little generous here. You'll definitely need a title and axis titles. There are a lot of colors happening. And maybe these labels can be rotated to be horizontal. The Legend should be moved outside the chart; although it's not obscuring the data, it is in an awkward location. So let's get started with that.

Here are some of the low hanging fruit to improve the chart, with the title, labels, and rotating the x ticks.

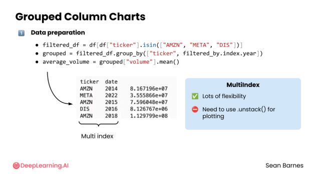
In order to move the legend outside the chart, you're gonna say P L T dot legend, which is a new function. Right now, the title is date but you really want to title it year. Then, use the argument b-box to anchor. B-box stands for bounding box, or the invisible box that contains the legend. So the question is, where do you put this box relative to the axes, or relative to the plot?

This function takes in an argument that specifies a point of x y coordinates. Let's try 1.05, 1. And 1.05, a number greater than one, is going to put the bounding box outside the chart to the right. Now you get a nicer location for your legend.

Finally, the color scheme is a little gaudy. You'll see in the next lesson that the Seaborn package, which is similar to matplotlib, has many more options for color scheme. The only option you really have in this case however is to manually set the colors for all 10 years.

I'm guessing you don't want to write all that, so you can just use chat G P T. Create a list of some progressively darker hex codes in pink. Copy that list. Borrow the same block as before. Now you get this result. Well, maybe these aren't successively darker shades of pink. But according to what ChadGBT can see, I suppose they are. All in all, it does have more of a progressive color scheme.

Looking at the chart overall, you can see that Amazon was trading the most compared to Meta and Disney. Disney's volume had an overall upward trend. Meta is a mixed bag, while Amazon peaked more towards the middle of this decade.

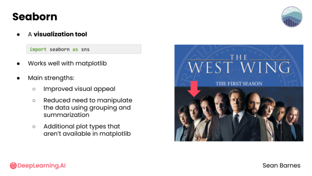

Grouped Column Charts
1️⃣ Data preparation
- `filtered_df = df[df["ticker"].isin(["AMZN", "META", "DIS"])]`
- `grouped = filtered_df.group_by(["ticker", filtered_by.index.year])`
- `average_volume = grouped["volume"].mean()`
`ticker  date`
`AMZN    2014    8.167196e+07`
`META    2022    3.555866e+07`
`AMZN    2015    7.596048e+07`
`DIS     2016    8.126767e+06`
`AMZN    2018    1.129799e+08`
Multi index
MultiIndex
✅ Lots of flexibility
⛔ Need to use .unstack() for plotting
DeepLearning.AI
Sean Barnes
	You saw the successive steps **[click]** used to create a grouped column chart. First, **[click]** you selected the columns of interest. Then, **[click]** you grouped by one or more features, in the case of the example both ticker and the year of the index. Then **[click]** you calculated the mean of the volume column for each ticker and year.
	
	[click] That sequence of steps gave you a **[click]** Series with a **[click]** MultiIndex, in other words, it has two stacks or layers of indices – both the ticker and the year. The multi index feature gives you **[click]** a lot of flexibility to create unique rows in your data, but in order to plot it correctly, you'll need to use the **[click]** .unstack() method.

	Here's how the unstack method works. It's very related to pivot methods. If you have two rows of indices, like ticker and year, it pivots this second set of indices to become columns instead. Now you end up with a single index, and the rest of the data captured in the columns of the unstacked data frame. Once you've unstacked your data, you can use dot plot kind equals bar and the columns are grouped by index automatically. You also saw how to move your legend relative to your axes, or your plot, using the bbox to anchor argument with an x y coordinate. An x coordinate greater than 1 moved the legend outside the plot. You also saw that you can give your legend a title.
🎙️ TH	That's it for matplotlib features! You've learned a ton of tools for customizing your plots and creating many different types. You've made it to the end of this lesson. Once you've completed the practice assignment and practice lab, follow me to the next lesson to check out visualization eye candy with a new package: Seaborn!

Lesson 3 – Visualization eye candy with Seaborn

L3V1 – Plotting with Seaborn

Visual	Script
🎙️ TH Time series & visualization Plotting with Seaborn DeepLearning.AI	So far, you've encountered a variety of plots and seen how to create customizations for those plots using matplotlib. There's one more data visualization tool you need to learn: Seaborn.

 **Seboran** - A **visualization tool** `import seaborn as sns` - Works well with matplotlib - Main strengths: ◦ Improved visual appeal ◦ Reduced need to manipulate the data using grouping and summarization ◦ Additional plot types that aren't available in matplotlib DeepLearning.AI Sean Barnes	Seaborn is **[CLICK]** a visualization tool that you can **[CLICK]** import into your code that **[CLICK]** works well with matplotlib. Its **[CLICK]** main strengths are **[CLICK]** improved visual appeal, **[CLICK]** reduced need to manipulate the data using grouping and summarization, and its **[CLICK]** additional plot types that aren't available through matplotlib. If you're wondering, Seaborn's unusual name comes from a character in **[CLICK]** The West Wing, whom Seaborn's creator took creative inspiration from. Let's compare creating a grouped bar chart in matplotlib and Seaborn so you can get a sense of Seaborn's strengths.
🖥️ Screencast 🔗 **c3m3l3v1 plotting with seaborn**	Working with the same stock prices data from the previous lesson, you used this code here to create a grouped column chart of Amazon, Meta, and Disney stock volume by the year. You ended up with this plot. Now, there are a couple things that are a little frustrating about this plot. First, it's a little bit ugly. And second, you have to do a lot of fumbling around with the data in order to get it in the right format for this plot, which doesn't seem like that unusual of a plot. However! You have other options for creating this type of plot. Let's do so with Seaborn. First, import Seaborn as S N S, its typical alias. Those are actually the initials of the Seaborn character in The West Wing. Since you've already created this variable selected Stocks, which contains just the filtered data frame, if you sample ten of those stocks, you just get Meta, Disney, and Amazon. You can use Seaborn to plot it directly. You're going to start with S N S for Seaborn, and use the bar plot function. The first argument is always the data frame. Selected stocks. That's what you're plotting. Now, what do you want on the x axis? You want the ticker. x equals ticker. On the y axis, you want the volume. There's no grouping, there's no mean, there's no stacking, you're just creating a bar plot. Shift enter, and you get this nice bar plot where the mean volume has already been calculated, in this case across all years. In order to separate out the volume by year, you're going to use the hue named argument. So just like you have here in the original matplotlib plot, the year isn't another axis, it's just the color of the bar. Hence the "hue" named argument. So hue equals and then the same thing that you use to group by year originally. Selected stocks dot index dot year. You get this really nice looking bar plot with the years in increasingly darker hues. This is the power of Seaborn. You get a really beautiful graph without too much effort.

Now, there's a lot that you can still do here.

Borrow this line of code and just add on to it. Since Seaborn is built on top of matplotlib,you can stack matplotlib and Seaborn commands together. So matplotlib or plt functions like title, xlabel, and ylabel work perfectly well with Seaborn. You may also want to change the title of the legend to year. Its position seems fine for now.

You can change the color palette as well, which is one of the coolest things about Seaborn. Add a new named argument to your plotting function, palette. The value of this argument should be the name of a color palette. There are a ton of these available to you, you’ll need to look up their names.

Here are some of them. Blues, or Blues Reverse, B R B G, and so on. Say palette equals blues, shift enter, and now you get a nice increasing hue of blue. Also try B U G N, that gives you a nice green color palette.

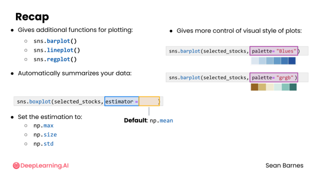
If you're really into that truly categorical look, you could do dark 2. Notice that the legend now moved because it was a little bit too long. In this case, it's showing every single year since the colors are so different, whereas if you went back to blues, for example, it only shows you every other year because you can infer the years in between because it’s a sequential colormap.

Let's use blues D, which is a darker palette, and that way you can just see the first year 2014 a little more easily. Notice these also have an error bar, If you'd like to remove those, you can say error bar equals none, that gives you a cleaner plot.

Now, Seaborn’s setup also makes it easier to quickly try out some other plots. Instead of volume, maybe you want to visualize the open price, for example, and that’s a quick change.

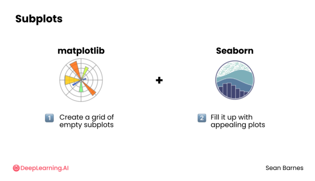
Now, by default, this function plots the mean price or the mean volume, but you can use other estimators as well. You’ll want to import Num Py as np. It turns out that way down under the hood, Seaborn, Matplotlib, Pandas, they're all operating on Num Py, so the mean, max, standard deviation functions, those actually all come from Num Py.

For the volume graph, by default, the estimator is np dot mean. If you specify it, you just get the exact same graph you had before. You can also do np dot max, which gives you the highest trade volume of each year. You could also do the size or the number of values in each year. Those were very similar, which makes sense because there are roughly the same number of trading days in the year, except 2024 has fewer elements because the data stops in October.


Recap
• Gives additional functions for plotting:
○ `sns.barplot()`
○ `sns.lineplot()`
○ `sns.regplot()`
• Automatically summarizes your data:
`sns.boxplot(selected_stocks, estimator = )`
• Set the estimation to:
○ `np.max`
○ `np.size`
○ `np.std`
Default: `np.mean`
• Gives more control of visual style of plots:
`sns.barplot(selected_stocks, palette= "Blues")`
`sns.barplot(selected_stocks, palette= "grgb" )`
DeepLearning.AI
Sean Barnes
	To sum up, Seaborn has a few key capabilities. First, it **[CLICK]** gives you additional functions for plotting, including **[CLICK]** `barplot`, which you saw in the demo, as well as **[CLICK]** `lineplot`, **[CLICK]** `regplot`, and many others you'll see throughout this lesson.
	
	Seaborn **[CLICK]** automatically summarizes your data, rather than making you go through the groupby and summarization steps manually. You saw that the default **[CLICK]** estimator was **[CLICK]** `np.mean`, which takes the mean of each group, but that you can also **[CLICK]** set the estimator to your liking, with options like **[CLICK]** `np.max` and **[CLICK]** `np.size`. You can even use **[CLICK]** `np.std` and other measures of variability.
	
	You also saw that Seaborn **[CLICK]** gives you more control of the visual style of your plots. You used the `palette` named argument with options like **[CLICK]** "blues", **[CLICK]** "BrBG" and more to improve the aesthetics of your plot.

🎙️ TH	That's the basics of Seaborn. It can be a lot of fun to create such visually appealing graphs. Seaborn lends itself to experimentation! Follow me to the next video to see how to create many side by side plots at once.

L3V2 – Subplots

Visual	Script
🎙️ TH	Often, one chart isn't enough. If you have 15 stocks of interest in your data, you may want to graph their adjusted close price side by side, on separate plots, rather than cramming 15 lines into one chart.
 Subplots matplotlib + Seaborn 1️⃣ Create a grid of empty subplots 2️⃣ Fill it up with appealing plots DeepLearning.AI Sean Barnes	You'll want to combine the strengths of **[CLICK]** matplotlib and **[CLICK]** Seaborn for this operation. You'll first want to **[CLICK]** create a grid of empty subplots using matplotlib, then **[CLICK]** fill it up with appealing Seaborn plots.
🖥️ Screencast 🔗 [c3m3l3v2 subplots]()	Now, say you have this list of tickers and you want to create a line plot for each one showing the adjusted close price over time. To create many plots in one image, you can use matplotlib's subplots function. First, manually set the figure size. The figure is like the canvas, and you want to make sure that the canvas is large enough to draw all these individual plots. You want to create three plots. How do you want to arrange them? One option is one row with three plots. Start with plot dot figure. Then you want to say fig size equals, and then the height and width. This is in inches. You can start with 5 by 5 for one plot, it's typically good, and you can play around with this. But since you have one row, the width can be 15, so 3 plots times 5 inches, and the height can just be 5.

Now, you're going to say plt dot subplot with three arguments. The rows, columns, and which plot you're actively creating. So there's only one row, there's three columns, and you're creating the first plot.

Now you can filter your data frame. So filtered df, tickers zero, which corresponds to the Amazon ticker. Then you can create S N S dot lineplot with the filtered df, x as the year,and y as the adjusted close price.

Now you can copy this code and basically do the exact same thing again. You can do Meta next. You just need to change two things. You're doing the second subplot, and you want tickers one, which is the second ticker that corresponds to Meta. That's it.

Copy this again. Tickers two. And now you're creating the third plot. Shift Enter, and you get these three line plots next to each other and they're all in one figure. Cool! If you want to save these, you can say plt dot save fig, and then maybe line plots dot PNG. Now you actually have all three of those in the same plot, which is super convenient.

Notice the repeated code when creating these figures. If you find yourself copying and pasting, you can use a loop. Let's transform this code to use a loop. Keep the figure size the same, keep the tickers the same.

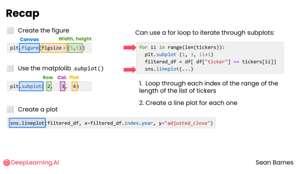
Now, you just want to iterate through this list. You can say for i i in range, length of tickers. That's going to take you all the way through the list of tickers. You don't want to just iterate through the tickers themselves, because you actually need a number to tell matplotlib which plot you’re creating. So take plt dot subplot, which one is that going to be? You have to start at one, so you need to do i i plus one, because remember the first index is zero.

For the second row, again, you're going to filter the data frame to the tickers position i i.

Then you want to do the line plot. No changes to this line of code. Now you can remove all these repeated lines of code, and remove save fig as well; you don't need to save it right now. So what's going to happen here is this for loop is going to run for the values ii equal to 0, 1, and 2. It will select the appropriate subplot and filter to the associated ticker to visualize, and you get the exact same plot you have before.

Now, you can plot a ton of things with this loop. It will work for any list of tickers. For example, you have all these tickers. Let's just take nine of those from the middle, copy and paste that same code as before. Now, what would you change about this code? You need to change a couple things. First, the figure size, you want 15 by 15, because now you have three rows and three

	columns. And same here, plot dot subplot 3, 3, that's going to give you three plots in a row and three plots in a column. Shift Enter, you get these nice line graphs here. All nine of them together. You can add more things to your loop as well. For example, you can say plot dot title, and then grab the ticker, for example. Here's Meta, which you saw before. And NVIDIA's wild rise since 2022, since the advent of large language models powered by their graphics processing units.


Recap
□ Create the figure
`plt.figure(figsize = (5,5))`
□ Use the matplolib .subplot()
`plt.subplot( 2, 3, 4)`
□ Create a plot
`sns.lineplot(filtered_df, x=filtered_df.index.year, y="adjusted_close")`
Can use a for loop to iterate through subplots:
`for ii in range(len(tickers)):`
`  plt.subplot (1, 3, ii+1)`
`  filtered_df = df[ df["ticker"] == tickers[ii]]`
`  sns.lineplot(...)`
1. Loop through each index of the range of the length of the list of tickers
2. Create a line plot for each one
DeepLearning.AI
Sean Barnes
 	
	To recap, subplots are useful when you want to combine several plots in a graph, especially when those plots are similar. You'll need to explicitly **[CLICK]** create the figure, which acts as the **[CLICK]** canvas, with the named argument **[CLICK]** figsize specifying the **[CLICK]** width and height of the entire figure. You saw that about 5 inches by 5 inches gives you a good size, but you can experiment with other sizes too.
	Then you saw that you need to **[CLICK]** use the matplotlib subplots function, with three arguments: the number of **[CLICK]** rows, the number of **[CLICK]** columns, and the **[CLICK]** plot you're currently creating. Calling this function right before creating a plot will position it in the correct place in the image. For example, `plt.subplots(2,3,4)` **[CLICK]** will position the next plot as the first in the second row, with each row having three plots.
	After calling the subplot function, you can **[CLICK]** create a plot as you normally would! You can use a combination of **[CLICK]** seaborn and matplotlib functions.
	You also saw that you **[CLICK]** can use a for loop to iterate through the subplots, You used a **[CLICK]** loop through each index of the range of the length of the list of tickers to **[CLICK]** create a line plot for each one.

🎙️ TH	Subplots are powerful for viewing many plots together, in particular when you want to create the same type of plot for many features or subsets of your data quickly. Now that you've seen how to create subplots, follow me to the next two videos where you'll apply this technique to a new Seaborn plotting function for box plots!

L3V3 – Box plots

Visual	Script
🎙️ TH	Seaborn gives you some great options for visualizing the distribution of your data and making it look pretty. While boxplots would have been challenging to make with matplotlib, Seaborn makes them easier.
🖥️ Screencast	Seaborn makes box plots much easier compared with matplotlib. For

🔗 **c3m3l3v3 box plots**

example, say you wanted to plot the distribution of the adjusted close price for Meta stock. First, filter your data frame for just the Meta ticker.

Then you can use S N S dot boxplot. Your first argument is always the data frame. Then you can just start with x equals, adjusted close. As simple as that. Now what happens if you do y equals adjusted close? Then you get a vertical box plot, which is probably not as nice to look at as the horizontal one, but it can be useful in certain situations.

What other customizations do you have here? One really popular option is despine, which removes some of these outer boundaries, or spines, from the plot figure. By default, it will remove the top and right boundaries, but you could also have it remove the other ones as well.

For example, left equals true will remove the left boundary. It's a little counterintuitive, but true means to remove it. You're saying yes to despine. Bottom equals true, shift enter. And now you get this clean plot without any unnecessary boundaries, which increases your data-ink ratio. Despine is a great option for boxplots because the y axis here doesn't mean anything.

You may also notice that the boxplot is pretty thick. One way to address that is to change the figure size to make it wider and flatter. This is a matplotlib function, P L T dot figure, which is what you saw in the previous video. You want to say fig size equals height and width. For width, maybe you want 10, and for height, maybe you want 5. So twice as wide as it is high. Shift enter, and then you get this nice wide plot.

You could create the same plot for volume. They're both looking to be relatively skewed, but volume definitely has a much more skewed distribution where a couple of days really stand out.

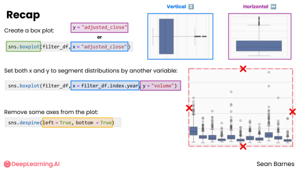
Now you can also plot multiple box plots on the same plot. For example, you previously had volume by year as well as ticker. If you add the year to the x-axisl, you get these side by side boxplots. Same thing here, you might want to despine left equals true, bottom equals true. And again, maybe you want to make the figure a bit wider. Just so you can see everything more clearly.

Give this a title using PLT dot title, and then apply some axis labels.

Now, if you want to give each of these a color, you might think to just set the year to be the hue, rather than x. Shift Enter. That does have this colorful effect but it also removes the years from the x axis because it's interpreting these all as coming from one category.

So instead of doing that, you could just give it a palette. Here's one of the ones that I like; it gives you this nice blue to gold transition. You could use whatever

	you want to experiment with. There are a lot of these. You can also just set the color individually, like color equals sky blue, for example, or a hex code. That gives all the boxes the same color, which in this case is fine since you already have the x-axis ticks to differentiate each distribution. You can see that in 2022, that was where the highest volume outliers occurred, with some others occurring in 2018.

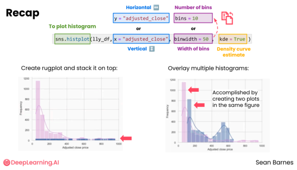

Recap
Create a box plot: `y = "adjusted_close"` or `sns.boxplot(filter_df, x = "adjusted_close")`
Set both x and y to segment distributions by another variable:
`sns.boxplot(filter_df, x = filter_df.index.year, y = "volume")`
Remove some axes from the plot:
`sns.despine(left = True, bottom = True)`
 Vertical ↕️
Horizontal ↔️
DeepLearning.AI
Sean Barnes
	To recap, you saw the **[CLICK]** `sns.boxplot()` function which **[CLICK]** creates a box plot. If you set a value to **[CLICK]** x, you'll get a **[CLICK]** horizontal boxplot, and **[CLICK]** y gives you a **[CLICK]** vertical box plot. You also saw that you can **[CLICK]** set both **[CLICK]** x and **[CLICK]** y to **[CLICK]** segment your distributions by another variable, like year.
	
	You also saw the **[CLICK]** `sns.despine` function, which works especially well with Seaborn plots like the boxplot, lineplot, and so on. It allows you to **[CLICK]** remove some boundaries from the plot, and by default will remove the **[CLICK]** top and right boundaries. You can also use the named arguments **[CLICK]** left and bottom to remove **[CLICK]** other boundaries as well.

🎙️ TH	As you've learned previously, the box plot isn't your only tool for visualizing distributions! Follow me to the next video to see how to create histograms and a new complementary plot type in Seaborn.

L3V4 – Histograms and rugplots

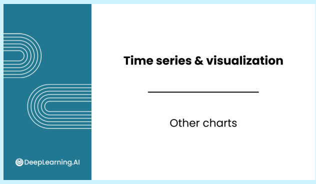
Visual	**Script**
🎙️ TH Time series & visualization Histograms and rugplots 	Histograms are a go-to plot type for visualizing the distributions of your features. Seaborn gives you great looking plots out of the box, and you can use matplotlib to further customize them.
🖥️ Screencast 🔗 c3m3l3v4 histograms and rug plots	Say you want to have a histogram of the adjusted close price for, let's pick Eli Lilly Co., which is a very large pharmaceutical company and their ticker is LLY. First filter the data frame for ticker equals LLY. S N S dot histplot is the histogram function. Tthe first argument is always the data frame, LLY DF. And for X, you can have the adjusted close. Now you get a graph of the distribution of adjusted closing prices. You can then add your matplotlib titles and axis labels. By the way, if you change the input to y equals adjusted close, then you get a horizontal histogram, but the vertical one is maybe a little bit easier to read. You have a lot of options for arguments to the hist plot. For example, you have

	an automatic number of bins here that Seaborn picks for you, but you can also set the number of bins, using bins equals 10. You can also choose the width of the bins. Suppose you want each bin to cover a $50 or a $25 range. So instead of bins equals 10, you can say bin width equals, and then the desired increment, like 25 or 50. 50 looks pretty good. 25 looks like the original perhaps. Maybe stick with 50 for now. You also have another available argument KDE. Let's just take a look at KDE equals true. This gives you the estimated density curve. The computer is estimating the probability density of the distribution of adjusted closing prices. So is another possible option if you're interested in a smoother representation of the distribution. You can also change the color as well. For example, color equals sky blue. Perhaps that's a little light. Maybe plum? Actually, plum is a nice one here. It also may bother you. Well, it bothers me at least, that these lines don't line up perfectly on the edge, so you can call S N S dot despine, to remove the outer boundaries, and left equals true to remove the left one as well. Now you basically have the exact same plot, but the edges looks quite nice. Now another option you have, and let's just borrow this code so you can start fresh, is to add a rug plot. Let's do S N S dot rugplot. A rugplot complements a histogram really well, because it displays all of the individual values in addition to the binned frequency that a histogram visualizes. So you get the benefit of visualizing the shape of the distribution and the location of the individual data points. You can just stack this plot right on top of the previous plot. S N S dot rugplot. Start with the data frame, set x equal to the adjusted closing price, and now you can see the individual values. Now what other options do you have here? You can the rug data points a color. For example, dark violet. That looks complimentary. The only other thing you might want to do is change the height. The default height is 0.025. Maybe you want it to be a little taller, let’s try 0.04. Something else I noticed is this graph is just a little small, so you could use P L T dot figure to adjust the figure size. Fig size equals, and try something larger than 5 by 5, for example, 8 by 8. Then maybe 0.03 is a better height for the rug plot. Not too shabby. Actually, with the larger plot, the default height looks quite nice. One last thing with histograms is that you can plot multiple of them together. Say you want to plot the distribution of Eli Lilly close prices against Thermo Fisher Scientific. This is another pharmaceutical biotech company.

	Filter the data frame for the TMO ticker, and then you can just go ahead and add another hist plot. Just like you stacked the rug plot on top here, you can just stack another histogram here too. You can keep all the same parameters except the color, which I'll remove and allow the default. Interesting, this stock price has a different distribution. The minimum price is higher for Thermo Fisher and it has a bit of a bimodal distribution. In fact, you could look at that pretty quickly with S N S dot boxplot, to see if it comes from different years. Yes, you can see that the price really went up during the COVID-19 market recovery, hence the two distinct peaks in the histogram plot above.
 Recap To plot histogram `sns.histplot(lly_df, x = "adjusted_close", binwidth = 50, kde = True )` Horizontal: `y = "adjusted_close"` or Vertical: `x = "adjusted_close"` Number of bins: `bins = 10` or Width of bins: `binwidth = 50` Density curve estimate: `kde = True` Create rugplot and stack it on top: Overlay multiple histograms: Accomplished by creating two plots in the same figure DeepLearning.AI Sean Barnes Frequency Adjusted close price 0 200 400 600 800 1000 1200	To play that back, you saw that **[CLICK]** S N S dot histplot allows you to quickly **[CLICK]** graph histograms. You can set the **[CLICK]** x **[CLICK]** or **[CLICK]** y arguments for a **[CLICK]** vertical or **[CLICK]** horizontal histogram. You can use the **[CLICK]** bins argument to set the **[CLICK]** number of bins, **[CLICK]** or **[CLICK]** binwidth to set the **[CLICK]** width of the bins instead. Binwidth **[CLICK]** overrides bins, so you'll only need to use one of them. You were also able to use the **[CLICK]** K D E argument to create a **[CLICK]** density curve estimate. You also saw how to **[CLICK]**create a rugplot and stack it on top of your existing histogram. The rugplot is a disaggregated plot that gives you insight into how the **[CLICK]** individual values are situated within the aggregated distribution shown by the histogram. Finally, you learned how to **[CLICK]** overlay multiple histograms on the same chart to compare distributions. This overlay effect can be **[CLICK]** accomplished just by creating **[CLICK]** two plots in the same figure.
🎙️ TH	Excellent work with histograms! You've learned the core chart types that will make up the vast majority of your data visualization work, including line charts, column charts, scatter plots, histograms, and more. In the next video, you'll explore some of the other weird and wonderful charts Seaborn has to offer! I'll see you there.

L3V5 – Other charts

Visual	Script
🎙️ TH	Seaborn offers many cool plot types beyond column and line charts, like heatmaps, violin plots, swarm plots, and more. You may not have a ton of occasions to use them, but they often serve a niche purpose in your visualization toolkit.

 Time series & visualization Other charts DeepLearning.AI	
🖥️ Screencast 🔗 **c3m3l3v5 other charts**	Let's take a look at three very cool visualizations you could create just to show off what Seaborn can do! This is a heat map of the correlations between various different variables. For example, between the stock volume and the open price in this square. You can see that a lot of these variables have a correlation of 1 with each other. The high, low, open, and close prices are essentially perfectly correlated, but the correlation between price and volume is a little bit different. There's a negative 0.2 correlation, so there's a negative relationship, but it's relatively weak. You can see that it is very close between all the different prices other than the adjusted close. It's a little bit lower of a correlation, not quite as clear a relationship, but still very close. Another cool plot you can do is a violin plot. Violin plot is similar to a box plot in that it visualizes the distribution of the data, but it also provides additional insight about where the mass of data is located. Where is most of this data in this distribution? You have these five different sectors, and they each have their own color and their own distribution as far as the closing price. You can see that consumer packaged goods has a much narrower range of closing prices, which might be worth investigating. Biotech, tech, and retail all seem to have very similar distributions, although for retail, there seems to be almost two peaks in the distribution, whereas tech and biotech both have these smooth curves in this area of the distribution. Another option is a regression plot. This is a plot of the closing price over the volume of trades. It's basically a scatter plot, but you can also add this line of best fit, which shows you the relationship between these two features. You already saw earlier this is a negative correlation of about negative point two. So there is some kind of relationship here, but it's a weak relationship. With the line of best fit, you can more easily predict for any given volume, what the close price might be. So those are just three of the many cool plots that you can plot with Seaborn.
🎙️ TH	That's a wrap on visualization in Python! You've learned all the core tools you'll need to create beautiful and functional visualizations, from Pandas to matplotlib to Seaborn. These three libraries form the backbone of data visualization in Python, and you'll rarely find the need to reach for something

	else. Coming up, you’ll complete the graded assignment and lab for this module. In the lab, you’ll work with oceanography data about Australia’s coral reefs to better understand trends in ocean conditions. You’ll analyze time series data tracking the clarity and amount of chlorophyll in the water, as well as create visualizations for that data using both matplotlib and Seaborn. Once you’ve finished the graded assignment and lab, follow me to the next and final module of this course, which is all about inferential statistics, including confidence intervals, hypothesis testing, and linear regression. I’ll see you there!