Module 3: Introduction to Data Analysis

Video Transcripts

Video 1: The Data Science Lifecycle

As motivation, let's consider a straightforward example of a machine learning problem. Suppose we're working for a real estate investment company and we want to be able to predict the sale price for houses. Let's consider a real house that recently sold in the city of Berkeley. This particular house has two bedrooms, one bathroom, was built in 1914, and is around 1,000 square feet or around 90 square meters. In principle, it should be possible to take the available data on the number of bedrooms, bathrooms, the location, the quality of the house's construction, and generate a reasonably good estimate. It turns out that this house sold for just a bit more than $1.6 million.

Building a model to solve this problem of predicting the house price is really a two-step process. The first step is to acquire and clean data so that it could be fed to a model. The second step is to train a model using that data. In the real world, business leaders and policymakers engage in a less linear process than these two sequential steps.

To give some insight, I'd like to briefly introduce the data science lifecycle, a concept from a data science course I teach at Berkeley. In the data science lifecycle, we sometimes start with a question. For example, what will be the sale price of a given house? Or it could be a bigger picture question like, how did the racially discriminatory housing practices of the early 20th century in the United States affect housing prices today. We then go and

find data which we must then understand. What are the rows, what are the columns of the dataset, and how much of the data is actually useful? What data are we missing? For example, after a first pass analysis of our housing data, we might then go and find crime statistics to help improve our model.

Our ultimate goal is to understand something about the world and make actionable recommendations or decisions. I should note there are actually two different entry points into this lifecycle. Sometimes we start with data, but no clear question. For example, we might have a bunch of climate data and we want to make useful predictions about how the climate will change and make useful recommendations to steer or prevent these changes. Or maybe we have the entire transcripts of all sessions of the United States Congress. And we simply want to dig around to find something interesting.

Whether you're simply training an ML model or you're pursuing a project that encompasses the entire data science lifecycle. There are a number of important fundamental skills that we'll be covering in the next few modules. In this module, we'll be covering five such skills: reading, querying, visualizing, aggregating, and filtering our data.

Video 2: pandas Basics

For this module, I'm going to do a great deal of live coding, so you can see my thought process as I work. So let's work our way through this Jupyter Notebook. I have here a bunch of notes which are just reminders to me what we're going to try and do and I'll be filling in as we go.

So the very first thing we're going to do is we're going to use pandas to read the GDP file that we're going to be working with today. And I'm going to show you what this dataframe looks like. So first, I'm gonna look at just the

first five rows of this table. And we see that, well, it appears to be the GDP for the country of Afghanistan in the years 2000 through 2006. We can also look at the last five rows with tail. Here we see Zimbabwe is at the end. We can also get a random sample if we use df.sample. So here are some various entities around the world and their GDPs and various years.

Now, one thing that I find a little annoying about this dataset we're working with is that the GDP is in these gigantic numbers. So what I'm going to do is just create a new column, which will be the GDP, but now in just billions of dollars. So to do so, I'm going to take this column right here. I'm going to divide it by 1 billion and I'm going to assign the result to df["gdp"] That's a million, that's a billion. And I'm gonna use a little trick you may not know, where I can put in underscores optionally, to make it easier to read.

Now that I've done that, I look at my table. I see there's a new column, which is just the GDP in billions. And that will be a little easier to think about instead of having to mentally consider, what is "e+09?" I just know this is billions. OK?

So, now that I've done that, I want to show you how we can do some querying on a dataset. So let's suppose we want to see only the GDP of China. So to do that, we can use some syntax you've seen earlier, where I'll say I only want rows where the entity is China. And so when I do this, I get my result. There is China's GDP and all the dataset from 1960 to 2017. OK?

Now one thing we did not cover earlier is what exactly is going on here. So let me just mention that this, right here, returns a series of true false values for each row, where the result is false if the entity is not China and true if it is. So, for example, if I take this here and I say, give me rows 1775 through 1785, let's say. We can see that there are some trues and some falses. Let's

see, 1775. Here we go. So we can see that somewhere in the middle, there's the crossover point where we go from true to false in our table. And so, in effect, whenever we're saying (df[“Entity”] == “China”), we're providing an array of Booleans as an argument to the bracket operator of df.

Now that's a little more advanced than we really need to know for our class. But I wanted to mention it, in case it's helpful to you in the future. Let's suppose we only want to see the GDP of China in the year 2017. So in that case, we can do something similar. We can say, give me only where the entity is China. Then I'm going to say and where df year is 2017.

So if I do this right here, I'll get an error. Well, two errors actually. So one of them is I need to make sure that this is in quotes. And the other is whenever you try and "and" two of these arrays of Booleans. Because of the order of operations, I need to actually put parentheses around this. Because otherwise what it's trying to do is and df year with the word China. So basically I'm saying, please complete this operation first, the formation of my Boolean array, and then the and operator will combine them. Once I've done that, I get the GDP of only China in the year 2017.

Now this syntax is a little awkward for querying. So there's another nice way to query a dataframe, which is called query, which is a great name. So, for example, if I say df query and I do entity equal equals China, like so, I'll get only the GDP for China. And I can add other conditions here with, if you've ever seen the programming language SQL, it's a SQL like syntax. So I can say, for example here, a year equals 2017. And this more concise code, in my opinion, allows me to get back the GDP for China in just that year.

Now as one last little trick, in terms of how to query and find rows of interest. You may find yourself wanting to know the GDP, not just of a

specific country, but an entire list. So here are six of the world's largest economies. And if I want to just see these top six economies, I can do the following. I can say top6 = df(df[“Entity”].isin(list_of_countries)] , and that will give me my top six countries. And so if I look at, we just pick a random sample. We'll see rows that are only from China, United States, Japan, and the others.

Now instead of using this syntax, there is a way to do that with query as well, that I'll be using throughout the class. And so to do that, we'll say top6 = df.query. We will say entity equals. And then we'll put an at sign, or sorry, entity in. And then @list_of_countries. And we do that, we'll get the same result as above. So now again, if we do top6.sample(5), we will get entries from our list. OK? So I'll tend to use this style more than this style. But both of these approaches are perfectly valid.

Video 3: Introduction to Visualization Libraries

We've just seen how we can understand a dataframe using query or selection criteria. But now let's see how we can use visualization to understand data.

We're going to look at a few different ways of visualizing data from our World Bank GDP dataframe. And rather than doing live coding in this section, I'm just going to show you some code snippets in the results. That's because here, the coding process itself isn't particularly interesting.

So first, let's suppose we want to plot just the GDP of these six countries from the year 2017 that we had in our previous example. One way to do this would be to use the pandas built-in plotting library, which you saw briefly earlier. To do this, we call a plotting function on the top6 dataframe itself,

giving the x, the y, and the kind of plot that we want. In this case, I'm saying x is Entity, y is gdp, and the kind of plot that I want is a bar plot, as opposed to a line plot or something else.

For reference, in the corner of this slide, I've linked the dataframe plotting libraries documentation, if you'd like to learn more. But personally, I'm actually not a big fan of what the pandas libraries outputs in terms of plots. I think they're pretty ugly, honestly, and they lack the interactivity that we'll see with, for example, Plotly.

So that brings us to our second plotting library, Seaborn. Seaborn is a popular library, especially for doing statistical plots, but it can also do standard plot types, like bar plots, line plots, and so forth. Now one nice thing about Seaborn that I want to show you right off the bat, is that it supports a number of really great color schemes that maximally differentiate each of the bars in our plot. And will tend to be more accessible for colorblind readers. Seaborn, like many other plotting libraries, is built on top of Matplotlib. We're not going to go over Matplotlib in our class, but advanced use of Seaborn to make your figures look just right, requires Matplotlib knowledge.

So as an example, in this plot, I have China, Germany, India, Japan, the United Kingdom, and the United States. And you'll notice that the levels for the United Kingdom and the United States are overlapping and we don't really want that. So if we want to fix that, it's not a Seaborn-specific command, but instead, we're going to use the Matplotlib xticks command. I can say the rotation equals 45, as shown here. And now all the labels will be at a 45 degree angle. But if you wanted them to be at a 90 degree angle or some other angle, you could do that too.

Now this is not a visualization class, so I'm not going to cover Matplotlib any further, especially because the primary library that we'll be using in this class is Plotly.

Plotly is an up-and-coming, but really powerful, visualization library that's my own personal favorite. After having used Matplotlib-based visualizations for nearly a decade, I've become a Plotly convert. There are a number of reasons that I love Plotly. One is that the plot it creates are often, but not always, better looking than what you'll get out of pandas or Seaborn. Another reason is that the documentation is generally quite good. And third, the plots, by default, they include these really nice interactivity features that I personally find useful when trying to understand and present data.

Now there are some downsides to Plotly. One is that the statistics visualization support is not as strong as, for example, in the Seaborn library. And we'll see at least one example of that in a future lecture video. Another downside is that Plotly is just not as popular as some of these other more established libraries. And so if you go online, you'll find there are fewer posts and generally a smaller community to help guide you towards creating exactly the figures that you want. If I had to guess though, I think Plotly was going to continue to grow in popularity and I expect that it will dominate, perhaps in the next few years.

Let's see a Plotly example. When I run this code, the plot that we get back looks pretty similar to the Seaborn plot from before. It has those same 45-degree angle labels and it has these nice bright colors for each of the different countries. Now, I personally don't really like this bright color palette. And so, if you'd like to try out one of the other color schemes that you can get out of Plotly, there's plenty out there. For example, here's one

called qualitative G10 that I like, which you'll notice looks a lot like the color palette that we had in Seaborn.

You'll get an opportunity to explore these three different plotting libraries and the nuances of these plotting libraries in this week's exercises. So I'm not going to go into all the nitty-gritty syntax details here. Instead, I'd like to re-emphasize that none of these libraries are objectively better than the others. Depending on the organization that you're working with or the project that you're working on, different libraries may be appropriate. By getting a taste of each of these three different libraries, you'll be able to remain flexible and adaptive when trying to visualize data.

Video 4: Aggregation Operations

Often when we're working with these real-world datasets, we need to do some kind of complex operations on that data, either to clean it or understand it, or to create another data table that we're going to feed into a model. Now I know that's a bit of a mouthful.

So let me give you some examples of questions we might ask that would require us to do this kind of preprocessing on our dataframe. For example, let's suppose we need to know the total GDP of the world in each year. Right now we only have a row for each country. So we need to think, how do we somehow combine those? Or let's say we want to know what fraction of the world GDP did each country generate each year?

For example, in 1960, was the United States 5% of the world economy? Half? We'll see when we look at the data. Or you might ask, how has each country's GDP grown since 1960? For all of these queries, it turns out that a

key part of coming up with an efficient and simple solution is to use what we'll see is called a groupby operation in pandas.

Now, at first you're going to find the style of programming a little uncomfortable. The code we're going to write is not going to involve any loops or any iterations and instead is going to be based on aggregate operations. This style of programming is a little bit different and it's known as the declarative paradigm. Here we're just going to say what we want. And then the programming language under the hood is going to do all the iteration for us. It's really convenient and nice once you've gotten some practice. But it's going to take some time to wrap your mind around.

Let's start with that first question of finding the GDP in each year. So what I want to do is I want to, first of all, let's grab the GDP for the year 1960. And that's just a starting point. Year equals 1960. And then we can, in principle, just use our usual sum function in Python. And that will give me back, of course, the GDP of the entire world in the year 1960, although I need to first make sure we are only looking at the GDP column. So I do that. I get back 9,025. In other words, the world GDP in 1960 was $9 trillion. OK? Now remember our goal was to find the GDP of the entire world in every year. So we could, in principle, write some code. So we say for i in a range. Let's say 1960 to 2000, as an arbitrary cutoff point. And then we could say, let's print out the GDP in year i… is... And to that we will add exactly what we have up here. We'll convert this thing to a string. We will make this an arbitrary parameter.

```
print(f"gdp in year {i} is: " + str(sum(df.query("Year == 1960")["gdp"])))
```

And if we do that and run our code, we get back a result. So that's the GDP in every year.

But this code right here, I'm just not a big fan of it for a number of reasons. One is, well, the output I'm getting, it is just a bunch of printed out statements. It is not a table, it's not a dataframe. So I could read, sure. I could tell a story with it, but it's not really in a format that's amenable to future analysis. The other thing is, well, if you were to try and write this code, it's very easy to stumble when putting this thing together. And it also explicitly loops. It's just way too complicated, this piece of code.

So what I want to show you is how to use groupby operations in order to make this code much more concise and also give us back our data in a nice format. So to do that, I am going to take my data. I'm going to use the somewhat mysterious for now function called groupby, and I'm going to say, I want to group my data by the year. Now, just at a very high level, you can think about it as pandas is being told, let's treat this data in a year wise manner. And then what I'm going to do is I'm going to say let's aggregate the data in each year by summing it.

So when I run this right here, I'll get back a dataframe instead of a bunch of printed out statements that have the results that I want. Here is the total GDP in dollars. And here's the total GDP of the world in 1960, in billions of dollars. So we can see that we get our result back in a nice format. And the code was really short and didn't involve any looping and didn't require any thought. OK.

Now, I should note that when you call, agg(sum), you get back everything that can be summed. So here I have the GDP in dollars and billions. And so I can say, well, you know what, I only really need the GDP. So if I do that, I get back a series showing the total world GDP in those years. And if I want to get back a dataframe, I should use double braces notation instead of single

braces. So you can see the difference visually here. So single braces gives me a series. Double braces gets me a dataframe. And we're not really going to go into the details in our course of the difference between a series and a dataframe. But for your purposes, a double braces format is what you usually want. This is the type of table we're operating with. OK?

So this new code, we're going to be picking this apart and really working to understand it. But before we do that, let's actually tell ourselves a little story using this data. So if I go down here, I'm going to use the Plotly library. And I am going to make a line plot showing this data. OK? So now, from that original raw data, I have the world GDP since 1960, and I can see its various moods and how it's grown over time. We can even see the great recession, here briefly on this world GDP chart. OK?

Here's another question. We might ask ourselves,. what was the largest GDP of any country in a given year? And, well, it turns out we can do code that is almost exactly the same as before. So if I just grab the code we had here where we were summing, the only difference on this next line is max. And so what that's going to do is give me back a dataframe showing the country with the largest GDP in the world. So it was 3 trillion in 1960, 17 trillion in 2017. And as before, we can plot and see the size of the world's largest economy over time. Now since this economy happens to be the United States, the Great Recession happens to be a little more visible than in our earlier plot, but it's a fairly similar shape.

How about the smallest economy in every year? Well, same deal, if I aggregate by min, I see that the world's smallest economy was 0.1 billions, or a 100 million in 1960. And then there's this funny artifact in the data, that in 1990, the world's smallest economy suddenly got much smaller. And so

we'll pick apart that strange mystery, but it's a lesson that in the data science lifecycle, sometimes just exploring our data, it brings up interesting questions. We didn't originally set out to think about the world's smallest economy. But while exploring the data, we found an interesting anomaly which maybe is worth considering later.

Now before we get to that puzzle, which we'll do in a little bit, I'll show you a little bit more sophisticated thing that you can do with the aggregation operations. So let's suppose we want to know the ratio of the largest to smallest GDP in a year. Basically the ratio of these two plots, OK? So there's a couple of ways you can do that. One is to take the largest GDPs and set them aside. I'm actually just going to grab the code from above. Here are the largest GDPs will be if we agg by max, the smallest GDPs will be if we agg by min, and the ratio of the largest to smallest will simply be the largest GDP's dataframe divided by the smallest GDP's dataframe. And by the miracle of pandas, this actually works. And so what it's doing, if we just look at one of these dataframes briefly, it's taking the two dataframes and dividing each of the corresponding elements. So the 1960 values get matched and divided. And so as a result, we end up with the ratio here. So the ratio of the world's largest economy to the smallest was 31,174 times larger. So that's a much larger economy.

Now, we can, if we'd like, do this all in one line. So instead of saying, hey, I want to explicitly set aside the largest GDPs and the smallest ones and then divide them. We can actually also do that with the aggregation function. So this is a little more advanced, but let me give you a flavor of how this works. Which, by the way, you'll have plenty of practice with on our homework. That's what I'm going to do is create a new function that is not the max, it is not the min, it is not the sum, it is the ratio of largest to smallest. So, if

someone gives me a series of values, I will say largest_value = max(s). smallest_value = min(s). And then I'm going to return the largest value divided by the smallest value. And this is just a regular old function. So if I do a ratio of largest to smallest and I do something like, give me an array of the values. Let's do like 5, 10, 15, 20. So I need to import numpy. If I do this. Largest, typo there. I get back four, because the ratio of the largest value to the smallest is four.

So this is just a generic function that takes a series of values and gives you back the largest divided by the smallest. And so by aggregate, my original series with this function, I will get back the exact same dataframe as when we manually divided the largest GDP by the smallest GDP. Bottom line here, this agg function is incredibly powerful and can take any arbitrary function to aggregate our data.

Video 5: A Visual Depiction of Aggregation Operations

Let's see an animation that showcases exactly how groupby and agg work. When we call groupby entity, you can imagine that pandas reorganizes our data, so that rows with the same entity are clustered together. Now in reality, nothing actually gets moved around, but it's a convenient fiction that makes the groupby operation easier to understand. Effectively, we can think of our original dataframe as being broken into sub dataframes, each of which contains all rows corresponding to a single entity. If we'd grouped by multiple columns, for example, groupby entity comma year. Then each sub dataframe would be a group of rows, where all of the selected columns are equal. For example, all of the rows with Afghanistan and 1977 would be part of a sub dataframe.

When we then call agg(sum) on the results of the groupby operation, pandas will look at each sub dataframe and will compute a single value for each column that represents the entire dataframe. So for example, if we pick sum as our argument to agg, the so-called representative of each column will be the sum. So for the China sub dataframe, the year value after calling agg(sum) is 5,967 and the GDP ratio value is 86.88. Here, the year value of 5,967 is just the sum of all years in the sub dataframe. And the GDP ratio here is likewise the sum of all GDP ratio values in the sub dataframe. After computing these values for China, pandas moves on to India. We again get 5,967. And that's because 1960 plus 1990 plus 2017 is 5,967. Our representative GDP ratio value for this time is also the sum, or 23.61.

Then lastly, pandas computes the representative values for the year and the GDP ratio for the USA sub dataframe. This time we get 5,967 and 9.56, respectively. I should note that these summed year and GDP ratio values are not useful. Summing three years is not a valuable quantity, nor is the sum of the ratios of GDP since 1960. This example here exists purely to showcase how the groupby and agg functions work and the resulting table has no practical value.

We could have chosen a different function to aggregate our data. So, for example, if we used ratio of largest to smallest, now the representative of each column will be the ratio of the largest to the smallest, as defined in this function at the bottom right. Now this time, the representatives for the year and GDP ratio are 2017 divided by 1960, or 1.029 and 79.4, divided by 1, or 79.4. Repeating this process for the India and USA sub dataframes, we get the value shown. Now as before, this output table is not useful for any practical purpose and is provided simply so you can see how groupby and agg work on a specific example.

Video 6: Sorting and Aggregating

Let's now consider this interesting mystery, that the world's smallest economy suddenly got smaller in 1990. So to resolve this question, we might do the following. So, the naive approach is, let's just take the entire dataframe, group it by year, and then agg(min). And this time, instead of looking at only the GDP data, I'm going to leave all of the columns intact. So in other words, the entity in this code column we haven't really been using. They will also stay. And when I do that, I see that in 1960, we see the word Algeria, and we see a GDP of 0.098736.

So, you might say, oh OK, well maybe the answer to our mysterious puzzle has just switched from being Algeria to Afghanistan. But the bad news is, these are not the GDPs of Algeria in the year 1960. So in other words, if I go up here, and I say, df.query("Entity == "Algeria"), switch my quote styles. There we go, oops. Alright, so to entity equals Algeria. One more, sorry. Alright, there we go. Algeria, here, in 1960 had a GDP of 27 billion. And up here, we see 0.098736. You might think about what's special about the names Algeria and Afghanistan that have led to this problem. So the issue is that what groupby is doing is actually computing the minimum entity in the year 1960. And since Algeria came alphabetically earliest, it's the one getting picked. So in other words, this naive approach, while it seems appealing, you just throw in the minimum function. It's not going to do it for us here.

So how do we actually figure out what was the smallest economy in the year 1960? Well, actually, in this case, it's a really clever trick and something you'll see all over the place. So here, what I'm going to do is take my values, and I'm going to, or my dataframe. And then I'm going to sort it by the GDP.

OK? So what did I just do? Well, my original table was 8,869 rows, and now it is sorted in increasing order by the GDP. So the very smallest economy was Tuvalu in the year 1990. And the very largest economy was the United States in the year 2017. And now, what I'm going to do is I'm going to consider each year, and I'm going to get the very first value in that year. And since we know that the dataframe is ordered by GDP, then if we get the first thing with a given year, we'll know it is the smallest thing. OK? So for example, let's create a function here that gets the first item. So to do that, I'm just going to say return s.iloc(0). And so for example, if I say, let's do df.sorted, or sort_values("gdp"). And then I take that, and I group it by year. And then I take that, and I call agg(get_first_item). What I'll get back is a table where we're getting the first thing within each sub dataframe. So everything that starts with 1960, the first row in that is Belize, with this certain GDP.

Now this is a little mind-bending, that this actually works. And so it's certainly something you'll need to practice on the homework. OK? And so looking at all this data, we can start to conjecture that actually what happens is that Belize was the smallest economy. But at some point, Tuvalu maybe entered the dataset and wasn't there before. So what we can do here, is look at Tuvalu and get a sense of, let's do query('Entity == "Tuvalu"'). So if we do this, we see that indeed Tuvalu, the entity, only existed starting in 1990. And so the reason we saw that sudden plummet was just because Tuvalu was suddenly in the dataset.

Now I do want to show you that we don't actually really need to create our own function, that gets the first item of a series. And that there's another way that we can create the result we wish. So here, instead of saying, agg(get_first_item), there is a built-in function called first that gets the first

item in a list. And by the way, there's also a max, and a min, and a sum. And so, anytime in pandas, whenever you do, say, agg(sum), this is such a common operation, that there is a built-in .sum function, so just be aware of that. And so, if you look at code that's available out there online, often people will write it as sum, instead of agg sum. But realize that whenever somebody says, .sum, it's just shorthand. OK? Alright, so we solved our mystery. And we used a clever trick involving sorting, and then aggregating by first.

Video 7: Indexing

As another demonstration of our incredibly powerful groupby feature in pandas, let's consider a harder problem to solve, which is, what fraction of the world GDP did each country generate in a given year? So for example, in the year 1995, what fraction of the world's economy did the United States occupy? So let's look at our dataframe again to reflect on and think forward about how we're going to try and solve this problem.

So here are all of the world's economies. And we want to know what fraction of the total world economy this eight billion represents. OK? So what we could do, you might imagine, is let's just compute the total_gdp for every year, like we did before. And then it's just a matter of saying, OK, well Afghanistan, you're eight. The total world GDP was 9000 billion. And so we'll just divide those. So ideally, you would like it that I could just say df / total_gdp. But we get back an answer, which is just nonsense. We get back a bunch of NaNs, and by the way, NaN means not a number. Now the reason this is happening, is that the total_gdp here, it doesn't have the same structure as our original data. You'll notice that it's indexed by the year, whereas our original dataframe is indexed by some arbitrary number. And

so, as a result, when it tries to divide these two tables, it gets confused. And so it ends up producing a bunch of not a number values, OK?

So to fix this, let's start with just a simple example, looking only at China. So if I take the world GDP, or all of the GDPs, and we query only where entity equals China. OK? We'll get back, of course, only the Chinese GDPs. Now one thing you can do, which is quite useful often, especially when I want to divide dataframes, is I can set the index. I can say, I don't want this to be indexed by some arbitrary row number from the original CSV file. I want it to be indexed by entity. Now we're not going to go through the precise definition of an index, but I think its purpose and its use will be clear. So notice what happened there, when I set the index to entity. Now, entity is the way that the data is organized, and all of the data shares the same entity. In particular, China. OK? If instead, I set the index to year. Now it actually does what I'd like it to do, for our example. Now, the data is indexed by the year, which is the same as our original total_gdp table. And in fact, this groupby operation, because you're thinking about the data in a year-wise manner, it will automatically set the index to year when it aggregates our data. OK?

So now that I have my data indexed by year, if we now compute the ratio here, it actually works, right? Now, we see that in 1960, China's economy was 0.104. It was 1.4% of the world's economy. Whereas in 2017, it was 12.9% of the world's economy. You'll notice I still get these NaN values here. And that's because what's happening is, it's trying to divide entity China by the corresponding value in this table that doesn't exist. And so we end up just getting name. OK? And so, what we'll probably want to do is only be grabbing this GDP value. But what we do get, what's really nice about this, is the actual value that we want. So now, let's just try plotting this again, because we like to tell stories with our data. So if I take this whole

thing and I look at only the GDP, and I see px.line. So if I do this, I get back the GDP in, the fraction of the world's GDP that China occupies in each of these years, right? So we can see that China, until roughly 1990, had an economy which was somewhere around 2% of the world's economy. But since then, has dramatically increased as a fraction of the world's economy.

Now, we can take our original dataframe, just to showcase something really important. OK, so here's df. Whenever I set the index of a dataframe, when I say df.set_index("Year"), you'll see that it does set the index to year. But if we look back at df now, so if I look at df again, it has not changed. So in other words, if I say set_index. Let's do it again in place. Set the index to year. And then I ask for df. It's not actually changing anything. And so the important point I want to raise here, is that set_index doesn't change your original dataframe, it instead creates a copy that is indexed by what you specify. So just in case you're working on the homework, or on your own projects, and you're like, why isn't set_index working? It's because you need to say, you know, some new dataframe equals set_index, OK, so just be aware of that.

So now what I want to do lastly, is I want to. Remember, we don't want just China. OK? Our goal was to get every single country's economy, the ratio of the world GDP that it occupies. And so to do that, we're going to need to do something a little trickier. OK? So here, I am going to, in order to do this, I need to actually do a multi-indexing. So I'm going to take my original dataframe, and I'm going to set the index, not just to year, OK? I'm also going to set the index to entity. Now you might say, that seems strange, how can you have two indices? Well you can, it's called a multi-index. And now, when pandas is manipulating this table, it's thinking about it in both a

year and an entity-wise way. There are two different quantities that it is considering as important for organizing the data. OK?

And so now, if I take this data here, and I divide it by the total_gdp, it's actually going to work exactly the way I want. So just to remind you, total_gdp looks like this. It's the world GDP indexed by year. And then df_indexed_by_year_and_entity is all the data indexed by both. And so if I take this dataframe here, and I divide it by this dataframe here, pandas is actually smart enough to line up the years, ignore the entities in this table, because they don't exist, and give me the exact result we want. And don't worry, you'll get plenty of practice with this on the homework. But what we see here is exactly the result I want. Now I have a table with 8,000 rows. This is everything. This is all countries, in all years. And we know what fraction of the world's economy they occupy, all in this relatively simple piece of code. OK?

And so now, what I'm going to do is take this original, or this resulting dataframe, and I'm going to reset its index. When I do that, I end up with a table, that's in my original format, where here's my original row number from the CSV. And we're back to where we started, but now with this gdp column, which represents the ratio, OK? So here, if I do px.line(share_of_world_economy). x = "Year", y = "Entity". Sorry, y = "gdp", color = "Entity". I'll be able to see a big table showing the world GDP of every country, and every year, in terms of its relative share of the world economy. OK? Now let's suppose this is too messy and I only want to see the top six countries, maybe. Remember that we can do something like, df. So let's get only share_of_world_economy.query, "Entity in @list_of_countries". OK, and then we do the top share. Now we can see the relative size of the top countries in the world. And so here, you can see that clearly China, among

these top six countries, has just grown so much faster than other countries in the list.

Now I should note, there are some funny features in this data. And again, one of the really important things, or one of the lessons that I'm trying to impart to you, is that we need to work with data and understand it through visualization and querying, because we want to look out for funny anomalies. So one that I'll just happen to notice here, is that Germany does not exist in the data set until 1970. And so, when Japan's GDP seemingly drops in 1970, and the United States as well. You need to be aware that, well, in the year 1970, it appears additional countries are added to this dataset, like Germany, which can skew the results. And so one of the most difficult tasks when doing any kind of machine learning or data science is really making sure you have nice, clean data. And doing these kind of exploratory visualizations can really help you understand what's going on with your data.

Video 8: Another Groupby and Aggregation Example

As yet another example of how we can use groupby to solve interesting problems, I'm inspired by the observation we had before, that China has grown significantly more than other top six economies. So here are the raw values for the total GDP of these top six countries over time. And what we'll observe is that China, in the year 2017, which was the latest data which I was able to download, relative to its 1960 value, is much, much larger than say, the ratio between the United Kingdom and its 1960 value. So I want to actually capture that and show by what fraction each economy has grown in a given year, and see which country has grown the most. So I'm first

going to show you how we might naively approach it, and where we run into some trouble. And we'll have a neat trick in order to get around our problem.

So first, I'm going to take our top six all-time economies. And I'm doing that by just querying the list of countries we had before, from this list. OK? So now, what you might imagine I could do, in order to get the growth of each economy. I would just say, OK, well, I'm going to set the index to be entity and year. And then I'll also take my data and group it by only the entity, and get its earliest value. And since my CSV file was ordered by year, it means they'll end up getting all my earliest years as a table. So in other words, the naive solution is, let's just take this table here, and divide it by this table. And we know that the way pandas will work is, it will find the matching China row here. And it will just divide this GDP across all of these China values and we'll get back the result we want.

Now, while that will in principle work, there's one flaw. And it's that the year that we see as earliest for Germany is 1970. So in other words, if we tried to actually do this, right here. Let's take this top table and divide it by the bottom table. Then the Germany data would be a little skewed because it would be comparing, not Germany's current economy to its 1960 value, but to its 1970 value. And so you might end up with an analysis which is deceptive in some way. So what I'm going to do to avoid this problem, is I'm going to, instead of forming this table as the denominator for my fraction, I'm going to create a new table, which is only the 1960 values. OK?

So there's a number of ways I can do that, for example, I can take my GDP values and I can say, give me only 1960. And then I'm gonna set the index to the entity. OK? And so what we'll get back here. Let's see, df.query. Sorry, say df.query, we get back only the values for 1960. And let's just keep only

the GDP. That's all we really care about. OK? So rather than having this table up here, which has values for our top six, which includes values that may not be from 1960, now it's only the 1960 values. OK? So if we did query in "Entity in @list_of_countries". OK, you'll see that we only have China, India, Japan, United Kingdom, and the United States. Germany is gone. OK. So this table is cleaner, it only has the 1960 values.

And so now, if I take my table, that we have up here, which is indexed by entity and year. OK, so I'm going take that, and put it here as my numerator. And then I'm going to divide it by this new table, with only the 1960 values. I get a GDP ratio. Let's see. So I need to also say, only the GDP value. I'll do single braces here. So if I do this, I get the GDP ratio. So now that I have that numerator and denominator set up appropriately, now I see that in 1960, China's economy was one times as big as it was in 1960. That's what you expect, right? In 1961, its GDP declined, in 1962 it declined yet further, and then began a recovery. We can see some history here actually. Likewise, in 2013, the United States' economy was five times as large as its 1960 value. In 2017, it was 5.622 times its largest value.

And so now, if we try to plot this data, so we take our top six by entity and year. OK, now we can finally tell our story. We can say x equals, let's say year, y equals gdp_ratio, that new column we created. And the color is our entity. Let's scroll down a little bit. OK, if we try to do this, we have an error. Now this error, I was expecting, and it's because we have not reset our index. It says, the x you've provided, year, is not a column in our dataframe. You need to give me one of the columns of your dataframe. And the problem here, is we've failed to reset the index.

So earlier I showed you how to reset the index. And so what I'm going to do here, is take top six by entity and year. And I'm going to reset its index. And I need to actually assign a copy. I need to do that in order to make a change. And now, if I try to actually plot my data again, it will finally work. So now we'll get to see our top countries, all visible. And so we see that among those top six economies, China clearly has vastly exceeded its 1960 value with an economy 79 times as large as it was. Whereas these other countries have grown, but not by nearly as much. And it makes sense, the countries which were considered developing, developed more. And you'll notice that Germany is missing because well, we don't have a 1960 value.

And so here, the trick was, the key thing we covered in this video, is the idea of creating a separate temporary table, which was only the 1960 GDPs, and then dividing by that table. Which allowed us to answer what I think of as a pretty interesting question.

Video 9: Filtering

Here, we looked at the amount by which the largest six economies in the world grew since 1960. But I actually want to know which countries grew the most. OK? So I want to include countries that may not be absolutely large, compared to the United States or Japan or, or India. I just want to know who grew the most.

To do that, I'm going to basically do the exact same code I did above. Here is a giant block of code that is the same as we saw before, except that now I'm doing it with the entire dataframe, instead of the top six. So I'm not going to talk through this code because it is identical to the above. And so what we see is that, for example, Afghanistan, it has a gdp_ratio of NaN, because we don't have a 1960 value. Germany will be the same. It has a

NaN value because it has no 1960 value. However, countries like Zimbabwe, India, the United States, we do have a 1960 value available, so there'll be a number here.

Now, one trick that I often use, though you need to be careful with it, is I can take this table up here. And I can say, just drop any row that contains a NaN. And if I do dropna, it'll just say, if any value in a row is not a number, or is null, any kind of weird value, then I get back a table, which no longer has my NaNs. OK? So if I do this, I get back only 5,202 rows, or 5,212 rows. And so we've dropped quite a lot of rows because a significant number of our countries, we just didn't have 1960 values.

Using various query operations, we could actually see which countries are missing. And that might be important to us if we're compiling a report or trying to make some kind of decision, and we want to know which countries we've left out of the report. Now I'm not going to do that here because that's just the same techniques we've covered before. And so instead, I'm going to move forward with my question, which is, I want to know which countries have grown the most.

So here, if I do px.line, and I say, I want x to be year, y to be gdp_ratio, and color to be entity. I now get the full picture. This is the fraction, sorry, the ratio of the current size of each country's economy to its 1960 value. And we see that China, yes, it has grown quite a lot. But there are actually a number of other countries that have grown quite a lot as well. For example, South Korea, and Singapore. They have grown tremendously since their 1960 value. And again, that makes sense because in the 1960s they were developing economies, and now they're advanced, highly built-up economies. We see Malaysia, we see Thailand, and perhaps surprisingly, at

the very top, we have Botswana. I actually didn't know, by the way, that Botswana was the country whose GDP had grown the most since 1960, until I put together this example. And then we have a long tail of a bunch of other countries. And by the magic of Plotly, you can go exploring this if you'd like. It could be fun to do. And you can see what you can learn about the world.

Now let's suppose I only want to filter, or generate a plot of countries which have grown by at least ten times since their 1960 value. That's going to be the last thing I'm going to do here. So I'm gonna ignore all of these countries because I just want to start with those which were ten times as large as their 1960 value. So one way I could do this, is I could take this dataframe, and I could say only give me, for example, where, in the year 2017, the gdp_ratio was > 10. And I'll be left with only 26 countries. And so, I could, for example, then pull out a list of entities and, look at what they are. So South Korea, Thailand, Turkey, and so forth. And I would know these are the countries that have grown the most.

But I want to actually generate this plot again, but only showing these top lines. So there's a number of different ways to do that. But I'm going to show you a piece of syntax we haven't seen yet, that will make this very, very convenient for further manipulation. So I don't want just the year 2017. I want all the years. But I want to base whether or not to keep a given country, only on its 2017 GDP ratio, OK? So what I'm going to do here, is create a function, which returns true if the maximum GDP ratio is greater than ten. Otherwise, it returns false. So I'm assuming someone's giving me a dataframe here. And I'm going to actually get the gdp_ratio from whatever you give me. Yes, I'll call this actually df, that's a better name for it. Here

we're going to say, largest_gdp_ratio equals the max. And then, if the largest gdp_ratio is greater than ten, I will return true. Otherwise, I'll return false.

I should note, by the way, this is sort of a bad programming habit. It's actually better to just say, return largest_gdp_ratio > 10. But if you have a habit of writing it the other way, that's OK. But I'm going to do it this way because it's more common in practice.

So once I've written this function, I can then take my dataframe, and I can group it by entity. And then I'm going to say, I only want to keep countries, I'm going to filter out countries. And I'm only going to keep the ones whose max GDP ratio is greater than ten. This is an incredibly powerful function that if I do this, I get back a small error, gdp_ratio. Let's see. Oh, I need to give it, sorry. I need to use the dataframe that actually has the gdp_ratio in it, sorry. So if I do this right here, now, I get ratio greater than ten. And if I look at this table that's returned, I see only those countries, Bangladesh, Turkey, and so forth. And what's so cool about this, is you'll notice that Bangladesh 1962 gets kept, even though its GDP ratio is only 1.11. And that's because we're considering the countries where the maximum GDP ratio is greater than ten. So say, the 2017 value. So if I look at ratio_greater_than_10, Bangladesh. Let's see, entity equals Bangladesh. OK, and I look at the result, I can see that, in more recent years, it did actually pass that ten threshold. Actually, just barely, right. And so, as a result, the entire set of all of those rows is kept. And we'll see an animation in a moment that shows that.

And so lastly, what I'm able to do is actually generate my plot. And now you'll see I've got the exact same plot as before, but now I'm only keeping countries which have grown by at least ten times since 1960. With the

caveat, of course, that there are some countries that are missing because our data is not enough. And so, that would be another thing you need to do in a real-world setting. Where you'd want to go and look and see which rows actually got lost here. But as a first pass, as a way to understand our data, this is a great plot and it tells us something about the world.

Let's see an animation that showcases exactly how groupby and filter work together. Now as before, groupby("Entity") effectively breaks our original dataframe into sub dataframes for each existing entity value. When we call filter, pandas creates one new dataframe according to the following rule. For each of our sub dataframes, pandas computes the results of the function passed to the filter operation. If this result is true, the filter command keeps that sub dataframe. But if the result is false, the filter command will reject that sub dataframe. Ultimately, the output of the filter command will be the union of all of the sub dataframes that were kept.

So as an example, for this China sub dataframe, pandas computes the max_gdp_ratio, which is 79.4. And since that value exceeds ten, the China sub dataframe will be kept in the output. For the India sub dataframe, pandas now looks at the max_gdp_ratio, which is 19.2. And since that is also bigger than ten, the sub dataframe is also kept. By contrast, for the USA sub dataframe, the max_gdp_ratio is 5.62. Since this value is less than ten, this sub dataframe is rejected. And thus, in the end, the return value of filter is a new dataframe, which has all the rows that we got from those two sub dataframes. In total, that'll be six rows. Note that this process has no effect on our original dataframe. In other words, the rejected rows are not removed from the original dataframe, upon which we called groupby.

Video 10: Conclusion

Let's briefly summarize what we did in this module. In preparation for future machine learning work, we learned today how to read, query, visualize, aggregate, and filter tabular data using pandas. During the videos, we only saw CSV files, but in the real world, you will often deal with other formats. While we won't discuss these other formats explicitly and these videos, the same basic principles from this module will work with those formats. And the pandas library makes it usually easier to read such files.

We got some basic practice with visualizations and we used the pandas, Seaborn and Plotly visualization libraries. Our most challenging topic for this module was the use of groupby operations to aggregate and filter data.

For those of you with lots of prior programming experience, you'll be tempted to use code that explicitly loops over your data. It is highly unlikely that that is the right approach for data analysis. Instead, I urge you to resist the temptation to write loops and instead rely on the much cleaner, much more efficient groupby capabilities that pandas provides. If groupby operations are new to you, it will take some time to gain familiarity and comfort with such operations.

And finally, we also saw how to set and reset the indices of dataframes, allowing for convenient operations between dataframes. For example, dividing one dataframe by another. That's all for this module. And I'll see you next time.