# Module 4: Fundamentals of Data Analysis

## Video Transcripts

### Video 1: Basic GDP and Population Join

Today we're going to build on our pandas knowledge. We're going to do some more sophisticated basic manipulations of data. And we're going to start by trying to expand on the stories we were telling last time.

So, recall that before we were looking at the GDP of various countries and we saw various trends — like that China is growing quite rapidly, that the United States is the largest economy, and so forth. Now, one thing we were unable to think about, was the relative per capita wealth of each country. So, if we look down here towards the bottom and I scroll around, I see countries like Belgium, or Switzerland, or the Netherlands — which we associate as relatively wealthy countries — and I want to capture that. And the way I'm going to do that, is I'm going to take that GDP value we had before and then simply divide it by the population of each country.

So, to do that, I'm going to read-in a population table that I've downloaded from online that is from the same source. It is this World Bank dataset. And this table, because it's from the same source, has a very similar format. We have the entity, we have a code for each country, we have a year, and then we have a total population value. Now like any dataset, I think it's always a really good idea to just look at the table and see what's interesting. And so I observe that, well, it includes populations going pretty far back. And we also see that it clearly is... that these are estimates. That Afghanistan in 1800 it says had a population of 3.28 million and that number stays fixed in the dataset for some number of years. So that's something to be aware of if we're doing some kind of a really important analysis, but for our purposes today, I'll just assume that the estimates are good enough.

Let's also try plotting this data again to get a sense of what it's like. And I'm just going to show the population of each country across time. So X is the year, Y is the population, and then the color will be the entity in question. So if I do this plot, I see that this data actually goes back 12,000 years to the year -10,000. And when I look at it, well, there's not much of a trend I can see here, because the data is so compressed. Other than the fact that, well, at some point in not too distant history, the human population did some kind of extreme hyper-exponential thing. The population has grown up and is transitioning to some presumably new steady-state. And we're living right through the middle of this incredibly crazy moment in history.

Now if I want to focus only on the part of history that we are more familiar with — since we're all relatively recently born — we can see the population of various entities since 1900. So here I see, well, all right, here's all that growth. but now just in the last century in change. And we look at the top entities. And we see, oh, okay, well the top entity is just "World". And so what this tells me is that this specific table has entities which are not countries, but are, for example, the whole world, or the continent of Asia, but it also includes specific countries. Okay? So just something to be aware of, though it's not going to change anything dramatically about our analysis, but we should know that these other bits of data are there.

Okay, so next what I'm going to do, is I'm actually going to take my GDPs and then divide them by populations. But the first thing I'm going to do is I'm going to rename this column, which I don't really like the name of. It's a little awkward to keep typing Gapminder, HYDE & UN. And so to fix that, I'm going to say pop.rename. And I'm going to take the name I currently have. Then the name I want. I'm going to put this inside of braces in order to say that this is a dictionary. And I'm going to say columns =. And if I do that, I get back a table which has a column named population. So it just makes life a little easier. Now I should note that the pop table did not actually change because just like set index and reset index, this creates a copy with the renamed column. So if I want the change to actually stick, I'm going to say take the pop variable and reassign it to be the new table. And if I do that, now, the result will actually change. Okay?

So now let's try to see how we can start doing some per capita adjustments. So let's look at the 2017 GDPs for the moment. And let's also look at the 2017 populations, right? So here, gdp2017 is going to be only those 2017 GDPs. And here are my populations. And now I actually want to form their ratio. Okay? So to do that, you might say, well, is there some kind of like setting index or division operation? But it turns out that the appropriate thing to do here, is to actually join the tables.

So there's a formal notion in database systems known as a join, where I'm going to combine these tables into one. So to do that, I'm going to say pd.merge. And then I need to enumerate which tables I want emerge and how. So I'm going to say my left table is my GDP table, this one right here. And since we're working with the 2017 table for the moment, I'm saying left equals GDP. And then right is going to be pop2017. That's this right table. Then I need to tell how to join. So I'm going to say I want to join on the "Entity" column and we'll see what that means in a moment.

But the basic idea is we want to line up rows based on the entity value. And then I'll say how = "outer". And we'll come back to what that means exactly. And once I do that, I end up with the single table, which has data from both datasets. So for example, we have Afghanistan and the year 2017 had a GDP of this and a population of this. And so the GDP per capita value can then be computed by saying gdp_and_pop_2017. And now I'm going to use the actual dollar amounts. And I'm going to divide by gdp_and_pop_2017 "population". So if I do this, I get back a table which now has these nice GDP per capita values.

And so now we have the results you want, but we do have some NaNs. So let's reflect a little bit on what exactly has happened with this table join.

## Video 2: More Basic Joins With a Diagram

Before we continue, I want to discuss briefly that how parameter that we used in the pd.merge function. You'll notice that the code that I used set the how parameter equal to outer. This is because the type of join we just performed is known as an outer join.

Consider the example shown here. Since we're joining on this Entity field, the pandas library finds all matching rows between these two columns. But sometimes we have a row in one table that has no match in the other. For example, the entity called Kosovo is in our GDP table, but it's not present in our population table. And so as a result, in the resulting join table, the population for Kosovo has no value. A similar thing happens with the GDP for the entity called Africa. Sometimes we think of one table as more important, and so we'll instead perform a left or a right join. For example, here we see the results of a left join. For every entity in our GDP table, which we used as our left table, there will definitely be at least one resulting row in the output table.

But notice that the Africa row is dropped entirely. And that's because Africa is not present in our left table. This type of join is useful, for example, if we're thinking of our GDP table as the important table and the population table is just a supplement that we'll use if information happens to be available. By performing a left join, we avoid introducing new entities, like Africa, into our analysis.

By contrast, observe the results of a right join. In this case, Kosovo is missing, but Africa is still present. Lastly, let's observe the results of an inner join. In this case, we only keep rows where the entity in question is present in both tables. Sometimes when we join, we may have values that are not unique. For example, in this left table, the name Jeb appears once, but in the right table it appears twice. And so, as a result, in the output table, we'll see two rows for Jeb. Things get even stranger if we have non-unique values in both tables. In this case, the resulting output table will have all possible combinations of the matching rows. So, in this example here, Jeb appears twice in the left table and three times in the right table. And so as a result, the output table actually has six rows for Jeb; one for each possible combination. For many practical purposes, you'll only ever be merging on tables with unique keys. So you might not encounter this somewhat counter-intuitive behavior.

Nonetheless, it's important to understand this, so that if this issue pops up unexpectedly, you can diagnose it. Now, I'm not going to spend more time talking about this, but you'll have an opportunity on the homework to explore this issue.

## Video 3: Joining by Multiple Fields

So what we've got so far, are the GDP per capita values for the year 2017. But I want to do better. I want to actually get the GDP per capita for all years since 1960. This code is actually going to feel pretty similar to before. But let's work our way through it one step at a time, okay?

Now, visually, if I want to think about what I've achieved just for the 2017 case, if I put together a little bar plot here, I can see the GDP per capita of various countries around the world. So for example, we see that Norway, Luxembourg, Switzerland, and so forth, Denmark and so on, they appear here and we see them on the high-end — as you might expect, if you're familiar with the relative prosperity of Europe. Now this isn't a particularly beautiful plot. And we could make it look nicer, but that's not my goal, right now. Okay? I'll notice by the way, that there are a number of other entities over here — like Latin America, or Tokelau — which do not have bars. And that's just a reflection of the fact that my table up here has these NaNs. And these NaNs are because the GDP table did not include some entities, like Yemen and World.

So, let's say we want to have not just the 2017 values, but values for all years. Well, in that case, rather than merging the gdp2017 table as before, and the pop2017 table as before, I'm going to merge on the entire gdp_and_pop tables. When I do so, I get back a really big table which will have kind of strange behavior. In particular, it has way more rows than I might expect. And if we look at it, it's just, it's strange. We have Afghanistan in 2002, but also in 1800, had a population of three two eight, and a GDP of this. It's like there's two year values on every row. And that seems very strange. And what's happening here is just what we saw in our join video earlier. So if we look at this gdp_and_pop result and we ask for where 'Entity == Afghanistan'. Okay, so we change this, sorry. I need single quotes, so I can do double-quotes here.

So here we get back Afghanistan. We see that there are 3,520 rows. Why so many? Well, if we look at our original table, which had 16 rows for the GDP of Afghanistan, and pop, our other original table, which has 220 rows. 220 times 16, as it turns out, is exactly the number of rows that we got in gdp_and_pop. In other words, we have duplicate entity values in our two tables. And so what we're seeing is all possible combinations, just like we saw on our diagram earlier.

So, what we'd like to do is join not just on the entity. We need to join on the entity and the year, right? So when we're trying to combine our GDP and population tables, we only want to combine the Afghanistan 2002 GDP data with the Afghanistan 2002 population data. And so luckily, it's actually relatively straightforward to join our tables in this way. So we'll say left = gdp, right = pop. And we'll say left_on =. And in this case we'll say "Entity" and "Year". Both of these things together will be the way that I'm going to join my tables. And then, just for the heck of it, I'll say we'll do an outer join. You could do an inner, or whatever else, but I'll do "outer". And so when I do that, I get back a much more sensible table with a much smaller number of rows. Okay?

So here we have Afghanistan with only one year. We don't have that very strange behavior where there's a double year. And now we have a table which is actually useful. You know what? I'm going to change this, by the way, to a left join, because really the GDP data is what I'm thinking of as the data I care about. The right table was just extra data when I want it. So I don't want to add all these new rows for World and North America. I just want the original countries for my GDP table. Okay, so now that I've done that, I can compute those values I've been seeking. I can take my GDP and I can divide it by population — as I already have filled out here. And once I've done that, I end up with a table which has the GDP per capita for each country in each year.

So now that I've done this, actually, you know what I forgot to do, I should say gdp_and_pop here. I notice there's too many rows. Notice there's 1,900,000. I forgot to assign my result here. Okay, so here we go. Now we have 8,869 rows, much better. And I see my GDP per capita for each country. Okay? So now that I've done that, I can now explore the history of various countries, like for example, Norway.

So here, this is the "gdp_per_capita" over time. Oops. I misplaced the parentheses. There we go. This is the GDP per capita over time of Norway. So Norway's GDP per capita has grown steadily over the years, but kind of leveled off since 2006. We look at say, "India". India was flat for a while, but it is a rapidly growing GDP per capita. If we look at say, "China", obviously of course, a lot of growth. If we look at, I don't know, let's pick another random country, let's say "Mexico". We see a different trend, right? Now I can also plot all countries. And this will give me the relative growth of, oops, I need to say "color = Entity". Otherwise it's just a big mess. And here we can see the history of the GDP per capita of a bunch of countries. Also a bit of a mess, but at least a mess we could try and untangle by looking at the data. This plot we put together does tell a story. It tells me, for example, that Luxembourg is a wealthy country.

What we don't see in this story, is how much wealthier, on a per capita basis, each country is than it was in 1960. So let's do that. This is going to be very similar to the previous module where we did this with the raw GDP values. It's just now we're going to do it with that normalized GDP per capita figure. So to do that, we're going to repeat that same technique, where we create a temporary dataframe with the 1960 values, and use that as our denominator. Now, since we've already seen this. I'm not going to live code it out. I'm just going to show you those steps again.

So as the numerator of our table, we're going to take our GDP and population join table, and we're going to set its index to be the Entity. Ultimately, our goal is to divide these GDP per capita values by the 1960 values. How do we do that? Well, we create a little table by querying the 1960 values. And here we end up with a table which has the 1960 values.

This is our temporary table that we're using as our denominator. And now each country's GDP per capita in a given year will be divided by this number. Same exact thing that we saw on that previous module. So once we do this, we're taking that gdp_per_capita value for each year and each country, dividing it by its 1960 value, and giving us back this table. So for example, in Zimbabwe in 2017, the GDP per capita was only 1.2 times as high as it was in 1960. In other words, while Zimbabwe's overall GDP may have grown more — because its population has grown — on a population adjusted basis the average person is not generating very much more wealth than they were in 1960.

Now I want to actually plot that so we can see that across all countries. And so first I'm going to reset the index, because we need to do that in order to use the x value as our year. So after resetting our index, we have table just like above, now with an arbitrary value for its index. Then I'm also going to drop the null rows because I don't want to include, for example, Afghanistan in here. Or I can't, because we do not have a gdp_per_cap ratio. Now just to reiterate, the reason I'm using the word ratio here, is this is the ratio of the GDP per capita to its 1960 value. So I drop those rows. And now we can finally plot. And what we see is a slightly different story.

So this is how much the wealth of various countries has grown over time. And we see, for example, that China, on a per capita basis, has an economy which is 37 times as large as that was in 1960. Below it, we see South Korea, we see Botswana, we see Singapore. And so I'll highlight Botswana as a key example.

You may recall at the end of the previous module we saw Botswana at the top of a similar plot. And that's because Botswana overall economy has grown more than China's. However, a significant fraction of that growth was because of a growth in population. If you adjust for population, China's actually on top of the pile in this measure. And so that tells you something about how the average person in a given country's experience differs from, say, their parents' or their grandparents' generation. And in that regard, we see China at the very top.

## Video 4: Scatterplot of GDP and Population

So we've seen between the previous module and this module two different ratios. In the previous module we saw a GDP ratio. In this module we've seen GDP per capita ratio. So let's also add a population ratio. Okay? And that's going to be a little confusing. So take your time as you digest this video, and of course you'll have a chance to explore it on the homework.

But what I'm doing here, is I'm just doing the exact same exercise that we did before with the GDP per capita, and I'm creating these other two ratios to their 1960 values. Okay, and we're going to reflect on them. So this code, I'm not even talking about it, because it's doing the same thing as before.

If we look at this GDP and population table, the important thing is I'm ending up with these three different ratios that each tell a different story about our different countries. So the GDP per capita ratio is on a population-adjusted basis. What is the GDP compared to 1960? When this value is large, roughly speaking, it means the average person is much wealthier than they were in the past. The population ratio is how many more people there are in a country than there were in 1960. So, for example, in Zimbabwe, the population is nearly four times what it was in the year 1960. Lastly, we have the GDP ratio. And that is, as a total sum of the economic output of a country, how much bigger is it since 1960? And so just taking this last row as an example, in Zimbabwe, the population is almost four times as large. The population is only about 1.2 times as productive in terms of wealth. And so the product of these two values is the overall change in the GDP of a country. Okay?

So let's explore some of these trends. So if we look, for example, at the population ratios of various countries — just to get a sense — we can see that, well, there are countries that have grown much more than others. So, Cote d'Ivoire has a population which is seven times what it was in 1960, which is a pretty amazing difference. We have Niger, Kenya, and so forth. And then we have a number of countries whose population ratios have barely changed since 1960 — a lot of Western Europe down here, right? And so there's so much to untangle with this data. And of course, people are studying this data who are specialists. So I'm not going to go into it in much detail. But I wanted to point out that with these group by operations, and with their divisions, and with their table joining, we actually have a pretty rich set of questions we can maybe try and probe.

Now in order to better visualize all of these stories simultaneously, let's broaden our horizon a bit by going beyond just a simple line plot. So I'm going to create a scatterplot here, which is going to show a number of these different ratios all at the same time. Okay? So what I'm going to do, is I'm going to say let's look at the GDP and population table. And I'm only going to look at the year 2017 for the moment. And then as X data, I'm going to do the GDP per capita ratio, and on the Y we'll do the population ratio. And if I do this, and I also wanna say color = "Entity". And if I do this, I get some stories about the world. Okay? What is the story I get?

Well out here we see there's some outliers. So we see China, for example, is a country whose population has grown somewhat modestly. It's on the low side, actually, among some of these dots, It's a little more than double than 1960. But the output per person is 37 times higher, and that is an extreme value. So of all the world's countries, China has had the biggest change in productivity since 1960. Over here we see South Korea. A similar story. Its economy has grown per person quite a lot, and its population hasn't increased very much. By contrast, over here, we see Cote d'Ivoire, which has a much larger population, but its productivity per person has not grown so much. We can see there's only a relatively small handful of countries whose GDP per capita ratios actually exceed 10. And these are the countries where the average person's life has improved, I would say, the most. That's not exactly true. We have to take into account things like purchasing power, but that's beyond the scope of our class.

Now I can actually show all three ratios at the same time and get an even richer story. So to do that, I'm going to take this exact code we have up here. And now all I've done is I've added a size to each marker. In this case, the size is going to be the "gdp_ratio". Okay? And so now the X and the Y's are exactly as before, but now the size of the marker shows the how much bigger the economy is today compared to its 1960 value. Now the size of the marker is just the product of these two things, because the population ratio times the GDP per capita ratio gives you the GDP ratio. But you can see that a lot of countries' economies, they're almost the same as they were in 1960. And we can actually diagnose why that is in various countries.

So in Liberia, there the GDP per capita is actually smaller than it was in1960. Much smaller. So the productivity of the average person has fallen. And while the prop, the population ratio is four times what it was, it has just barely made up to keep the GDP above its 1960 value. So Liberia, presumably — and again, I'm not a historian, I don't know how to digest this — but it opens up an interesting question that I'd be curious to know. Is the average person in Liberia worse off than 1960? This GDP per capita ratio suggests yes.

And so this kind of visualization that opens up a whole world of interesting questions that, again, you can potentially explore using machine learning or other techniques, whatever they may be.

## Video 5: Incorporating Life Expectancy Data

The only thing better than two tables of data is three. So we can take our joined GDP population table and start to bring in other measures, like, say, life expectancy.

So here I again collected life expectancy from the same entity as before. And it shows the life expectancy of a person in a given year. Now you notice some of these values are very small. Keep in mind that life expectancy includes infant mortality. So that's not to say that somebody who survives birth only lived to 27. It is heavily skewed by high infant mortality here. Okay?

Now given that I have this life expectancy table, I want to actually join it to my earlier table. I'm going to drop this code column, by the way, because I don't really need it. It's just the same three-letter code as before. So here I'm just pulling out only the "Entity", "Year", and "Life expectancy". And once I've done that, I'm going to join my tables. Now here the code is exactly as before, so I'm not going to retype it. But as my left table, I'm using gdp_and_pop. And on my right, I'm using life expectancy. And as before, I want to join on both the "Entity" and the "Year". And we're doing a left join, because we're thinking of the gdp_and_pop table as our primary table.

Once we've created this table, we have in each row, of course, all the data we had before. So Algeria in 1960 had a GDP of $27.43 billion, a population of this value. We have all of our per capita, sorry, all of our ratios. And, of course, since we're talking about the year 1960, they're all one, because all of them are exactly the same as in 1960. And then lastly, we have our life expectancy. So we're starting to have pretty high-dimensional data with a lot of columns. And that's pretty typical once you're trying to do a large data analysis project. So now that we have this, we can start to do other things with this data, like for example, creating a simple scatterplot. So here, I'm going to plot on the x-axis the "gdp_per_capita". And on the y-axis, I'll put our "Life expectancy". So if I do this, I end up with a plot that summarizes, in some sense, this implicit relationship between the GDP per capita and life expectancy.

And you'll actually notice a pretty interesting trend, which is that, well, as the GDP per capita goes up, life expectancy increases. And there's a large number of countries which have relatively low GDP per capitas, and have a wide range of life expectancies. But once a country has a GDP per capita above a certain level, life expectancy tends to be relatively long. Now I should note that this Y-axis starts at 55, and so it looks a little skewed. But the reason is that way we can see most of our data here, okay? And so we can poke around at various countries. But I won't do so here, but you can do so if you feel.

Now one thing that jumps out at me, is that if I were to plot this data on an axis where the x was logarithmic, I might expect to see almost some kind of interesting trend. This curviness, it feels kinda almost logarithmic in a way. And so just as an experiment, I decided to try it. And I'll just do this in order to show off the logarithmic x-axis feature of Plotly. So all I'm going to do, is use this exact same code as before. And I've added one new thing, which is I'm saying log_x = True. If I do this, I get the same basic plot as before, but now my GDP per capita is on a log scale. And here, interestingly, we see what's almost, looks almost like a linear relationship. Now, I don't know exactly why this is. I'm sure people have reflected on this. But it does say that maybe there's some kind of model you could build.

This already opens up an interesting machine learning problem. If I wanted to predict the life expectancy of a country based on its GDP per capita, on this logarithmic scale, a straight line model might do a pretty good job. I find that pretty fascinating. This is the kind of question, and the kind of analysis, that is opened up to you when you're able to manipulate these datasets and combine them.

## Video 6: Violin and Box Plots

So far we've seen a bunch of classic plots, line plots, bar plots, scatterplots. And so now let's start talking about some more statistical ways of looking at data. Now you've already seen histograms with the pandas library. But what I wanted to cover here now, is Plotly histograms, and Seaborn histograms, and some of the additional features we get out of Seaborn.

First of all, this line of code here it creates a histogram using Plotly. And so I feed it just this one-dimensional data. I'm not saying what the X's and the Y's are, I'm just giving it data. And it'll give me back a histogram. And this histogram is interactive in the sense that I can put my mouse over it. And it will tell me that among the world's countries, there are exactly 16 whose life expectancy value is between 60 and 64.99. I think that's pretty nice.

Now, one thing I don't like about this visualization, is that it does not have little lines around the bars. You can add them, but by default, for whatever reason, Plotly, at least right now, does not have them. Now just to show you what a histogram in Seaborn looks like, and how you create one, the code is sns.displot. Okay, that's a histogram in Seaborn. And when I do that, I end up with a plot that I personally like a little more than the Plotly version — other than the fact that it is not interactive. It has lines and just feels generally cleaner.

Now one of the great things about Seaborn histograms, is they support additional features that can sometimes be useful for understanding the distribution of data. So this displot line here, if I add to it kde = True — so this is literally the exact same code as above, but kde = True — it will overlay on the histogram something known as a kernel density estimate. Now I'm not going to explain from scratch what a kernel density estimate is, but you can think of it as a smoothed version of the histogram. So that it gives you, instead of these hard edges, it gives you a wiggly curve, giving you a rough distribution of the probability distribution of life expectancies. Okay? Now, there is a more formal definition for kernel density estimates that you can go learn about, but it's not in the scope of our course.

Another thing you can do, which is really nice with Seaborn, is you can create what's known as a rug plot. So if I say rug = True, we can actually also see the original data. Now it's pretty small. But, technically speaking, every single data point in our original dataset is on there. And so you get a sense of, okay, well, I know there were three different countries in this bin, but what what life expectancies were there? Well, there's one, two, three. And so we can see that one of them just barely missed that bin edge. And so a slightly different binning would have maybe had a lower value here of only two, instead of three. Okay? So you'll see those out there in the real-world, these kernel density estimates or rug plots from time to time. And they're very convenient to put together using Seaborn

By contrast, it's not very easy to do this in Plotly. Technically speaking, the Plotly library does support kernel density estimates and rug plots in conjunction with the histogram. But at least as of the current moment, it's really awkward. Though there are notes that in the future, the histogram library will natively support them. But at least as of the time I'm creating this, it does not actually support them. So I tend toward Seaborn when creating these kind of plots.

Now kernel density estimates open up a really interesting kind of plot you may not have seen before, known as a violin plot. So in Plotly, I'm going to do this one in Plotly, because I think it looks nicer than Seaborn. I can say px.violin and say give me the life expectancy data. So what I get out of this, is a really interesting plot that is really ultimately just the kernel density estimate of the life expectancy. So this curve right here is just a sideways version of this curve in Seaborn. And it gives you a sense of the distribution of life expectancies in countries. And we can see that the fattest part of the curve is around 77, and the skinniest part is down here towards 44. So this is just a sideways version of the kernel density estimate. You can think of it also as a smoothed, sideways histogram

Now you can also on top of these violin plots, plot the data. So I'm just going to take this exact code from above and add points = "all". And we'll see a side-by-side of the kernel density estimate with the original data. So here are a bunch of different countries. They're not labeled, but these are just where the data points are. And the Y value is the value, the actual life expectancy value. And then the X, right? You might say, why is this one over here versus here? It's just an arbitrarily chosen X point to spread out the data and make it easy to see. But there's no actual meaning of the X coordinate. It's just a randomly chosen value, so that they're not all stacked on top of each other. And so here we can get a sense of the distribution of our data and see that whereas the smoothed histogram indicates that in our distribution of randomly chosen countries, there could be some countries with a life expectancy of 89, or 87, or 88, in practice, there are no countries — at least at the moment — that actually attain that value. Even though the distribution would suggest that the underlying distribution actually could produce such countries. Okay?

Now we can take this data and actually augment it yet further. If we're not just happy with GDPs, and populations, and life expectancies, I can throw in continents as well. And I'm going to do this because often you'll see people create violin plots are other kinds of plots that are segmented by, say, continent. Right? I want to try and consider different categories. So here I read-in a data file which I created myself by just taking some random list of countries and continents, and then making it into a CSV. And that will allow us to associate, of course, each country with its continent. We can merge just as before. I'm not going to do any description of this code because it's exactly like we've seen before. And I get back a table where, now, each country and year is annotated by the appropriate continent

Once I've done that, I can do something like this. I want to look at only countries from these two continents. So just using the kind of query operations we've done before. And I'm going to color each of my violin plots by the appropriate continent. And so now I can see the relative life expectancies of countries in Africa versus Europe on a violin plot. And we can see that the distribution is much wider in range in Africa as compared to Europe. You could, in principle, add other countries. And you'll get a chance to explore this on our homework. Now, I think that violin plots can be a little messy in these kind of circumstances.

So violin plots have a very close cousin, known as a box plot. So a box plot, you can think of it, doesn't have all this extra information about these smoothed bends, and instead just displays all the data as raw boxes. Okay? So these boxes here, each of the lines has a specific meaning. So the very middle of the box shows the median of the available data. Q1 and Q3 are the quartile values. So that's the bottom quartile, sorry, the first and third quartiles. Then we have the upper and lower fence, which are related to these quartiles in ways that I will not describe right now. And then we also have the outliers here, which are the max and the min for that particular condition — the maximum and the minimum life expectancies — in this case, for Africa.

And lastly, because box plots, one thing I really like about them, is they allow you to really conveniently display, for example, data for all continents. So here is now a statistical plot showing the wrapped distribution of life expectancies for all the continents that I have available. That's a nice, concise picture of life expectancies. So if you're trying to understand the dataset, especially if you're trying to decide whether or not it's worth using, or are there other errors in the data set? Is there a story in this original dataset? Box plots can be a first, a great first pass, and I think make for really nice visualizations of the distribution of different categories.

## Video 7: Joint Plots

I'd like to briefly broaden our toolbox by showing how we can create marginal plots, or joint plots. So, the scatterplot feature of Plotly allows us to say, I want to also create a marginal plot on the y and the x-axis for the data that I'm plotting. I'll show you an example so it makes sense.

So when I create the scatterplot, I'm saying on x is my "gdp_per_capita", and on y is my "Life_expectancy". We see that same curve from before. But now, because we've told it to create marginals on the x and the y-axis, we also get the marginal distribution of each of my two different variables. So, here is the distribution of my GDP per capitas, which, as you can see, is skew on the low side. There's 55 countries in the lowest bracket. And on the y-axis, we have my marginal distribution of life expectancies. There are 20 countries with the highest life expectancies, then there's only five in this lowest bin. Okay? And so that can be helpful. Let's say you're trying to put together a very concise report when you're discussing the results of an analysis you've done. This kind of plot can allow people to easily see salient features. They might want to know, how many countries are there with long life expectancies or high GDPs? Now these marginal plots can be...there's many options.

So, for example, instead of a histogram, I can actually do a box plot up there. And so, again, we'll end up with the same plot as before, but now box plots. Now, I'm not quite sure why, but in Plotly, the box plots that it produces as marginals, they have a little pinched waist. But it's the same idea as before. We have the median, the first and third quartile, and so forth. So, we have here a marginal distribution again, so that people who are expert at looking at these plots, can immediately look at them and get some sense of the distribution of the data. I want to capture concisely on the actual plot, the density of points in a location, right? I see a lot of points here. I see only a few points here. I see no points in the corner.

So, there's something known as a density heat map, where, instead of a scatterplot, it's going to replace my points with, well, you'll see. Some colors, right? So here, this is basically the same plot as above. It's just that each one of these points contribute some color to its appropriate box. Okay? Now, I'm actually, this is the default color scheme. I don't know that I love the default color scheme, but I'm just rolling with it. And so each of these boxes represents basically a two-dimensional histogram bin. So, these are countries whose GDP per capita are between zero and $19,000, and whose life expectancy is between 75 and 80. There are 16 such countries. And so we see there's quite a lot of countries in this leftmost column. And then we also have, on the top row, a lot of countries that have long life expectancies and small, relatively larger GDP per capita. And then way on the edges, we have a small number of countries which are both very wealthy and live long. But this is the overall distribution. Now I don't think that this specific selection of bins is particularly informative, but on other datasets you might find this sort of thing useful.

I should note that another tool that I've used personally, is in Seaborn. So Seaborn also supports these marginal plots, where we can put on the x and y-axis, next to them, extra plots. And it also allows you to do these heat map type plots. One version that exists in Seaborn, but not in Plotly — just to show it off — is I can say kind = "hex" with this jointplot command. And it will do basically the same thing, except now these are little hexagon shapes. And for various reasons, hexagons might be a better choice than a square shape. But, again, little beyond the scope of our day. So I won't go into too many details. And we see the same marginal plots up here. So, it's possible in Seaborn, though I will tend towards using Plotly.

Now whether or not you actually end up using all these tools, just kinda depends on where you go from here after taking this class. But I did want to go out there just a little bit, so that you know these tools exist, in case they're useful to you.

## Video 8: String Operations and Data Cleaning

Another really important tool in your pandas toolbox, will be the ability to manipulate strings. So, let's say, for whatever reason, we only want to keep countries whose name includes the letters IN. To do that, I have this line of code, which I'll talk about in a moment. But after we run it, we get only countries that, you'll notice, have IN in the name.

How this works is we're saying: hey, table, I want you to consider the string library that's available to you. And consider also the entity column. Among those strings, only give me those which contain "in". That's the flavor of these. So you'll always have this funny .str.contains. And this is a library of so-called vectorized string functions that operate very efficiently.

Another example is startswith. So, if I want to find those who start with B, maybe I'll type it out so it feels real. We'll say, give me only those entities whose string — so we're saying use the string library — startswith, and then we say "B". Starts with, startswith. There we go. There's no space. No underscore. That's it. Alright. So we do startswith. And we get Bahamas, Burundi, and so forth, right? And so, if we go look at the documentation, I should note we've barely scratched the surface of what string functions exist. There's startswith, strip, swapcase, rsplit, and more. And so, obviously, we're only going to cover just a few of these. But I wanted to give you the same, or the basic, syntax that we need. As an example of one of the slightly different flavors of these, these ones, like startswith and contains, they give you back an array of true false values.

But there are also some that could be useful, for example, for uppercasing our entire dataset. So if I say give me the entities all in uppercase, the way to do that in code is .str.upper. I'll get back all caps Algeria, all caps Zimbabwe, and so forth. Now, the reason you might want to do that, is imagine you're trying to join two tables. One you've downloaded from a certain source that is in mixed case, and another that is in all caps. You can easily convert them all to uppercase and then it makes it easier to join your data. Now I should note that this .str.contain supports arbitrary strings. And so, as an example, if I say contains space, then we can use this to find, for example, countries who have a space in their name. So, that's just a tiny fraction of what the string library in pandas is capable of doing.

And so, in order to demonstrate a couple of new features and also give you a real-world context for why these are really useful, let's try and do a somewhat tricky task. So here what I'm going to do, is I want truly real-world data. So, I'm going to go to Wikipedia, and I'm going to look up Indian states on Wikipedia. Here are the states and union territories of India. I want this table, okay? And what I'm going to try and do, is I'm going to count the number of people who speak Telugu. Okay. I don't speak Telugu. I don't, but I know of its existence. So all these people, they'll count. These ones, they won't.

So to do this, I need this table. How am I going to do that? Well, there's conveniently on the web, people who've done things like convert Wiki table to CSV, okay? Some random guy's website. So this kind of programming a super common when doing data science and machine learning. Because you need to get data somehow and people have made it easy for you by creating websites like this. So I'm going to grab this right here. And I'm going to say Convert. And when I do that, assuming their website's working, which it is, I get a table. And look, here's that table I want. I'm going to download this file and I'm going to copy it to the appropriate directory. I won't show you all the file system manipulation, because you know how to do that, but that's the table I'm using. I literally did that process to create that file. Okay?

So, now that I've done that, I'm going to say read_csv, "indian_states", I think I called it. Hopefully that reads. pd.read_csv. And I get back the Indian states. So, I should note this file is exactly this table. It's just they didn't show you the step where I copied it to the folder and then renamed it to be called "indian_states.csv". So this is straight out of Wikipedia. And this is something that is so important and useful to be able to do, that I'm going to go through the steps, even though it is a little annoying. So what do I see? Well, I see a bunch of states, whatever this column is. I see kind of messy stuff in here, like this bracket 40. And if we go back and look at Wikipedia, this bracket 40 is the citations at the bottom of the page, right? So they're in our data, because it's messy real-world data.

So remember, our goal is we want to count up the people who speak Telugu in India. Now the first thing I want to do is rename some of the columns. I've omitted typing this out because it's too boring, but the official languages column is named Officiallanguages[39]. And, again, this 39 represents the fact that it's the 39th reference on that Wikipedia page. And so, if I rename these columns, I get back a dataframe which has the official languages labeled "official", the additional official languages labeled as "other", and the population labeled as "population".

Now, why are there two language columns here? Well, if we just go back to the original table, we can see that there's something called official languages and additional official languages. So, I don't know why. That's just whoever created this Wikipedia page decided to do. And so we're stuck with it. Okay? So now that we've done this, we're going to try and add up all of our folks. So, let's try and find only those states where people speak Telugu — either officially, or unofficially. So to do this, we can ask, for example, df. ["official"]. Let's see, isin, our site. .str.contains("Telugu"). Okay. So it could be the case that the official language contains Telugu, or it could be the case that the other languages include Telugu.

Okay, so now we've done this. We have a restricted set of states. None, okay, that's not good. Let's see. What do we got? Official Telugu. Other Telugu. Okay, so when we're debugging, I'm going to start simple. Okay, so those are the states which officially speak Telugu. I misspelled Telugu. There you go. All over the place. See I told you I don't speak Telugu. I cannot spell the language. Alright, so now that I've done that, I should hopefully get a list of states. So these are the two states where Telugu is spoken as the official language. And I want to now also get the ones where the other language is Telugu. Okay, now we have three states, all right?

And so we want to add up these population numbers. So let's try and add up those populations. So here it feels like we're almost done. And we can almost do this query. So, if I say "population" and we try to sum this, let's see what we get. Okay. It says, oh, unsupported operand type(s) for + 'int' and 'str'. So what's going on here? Okay? It seems a little strange. Or, you know what we can also do, is just say .sum. That should also work. Let's try that instead. Okay? Alright, something even weirder happened. Instead of an error, I get some absolutely giant number which has commas in the wrong place and it has this citation number. And so we need to step back and think. What is going on here? Well, we have the table here. And we can see that it's just the raw concatenation of these strings. So something has gone wrong here. And it is not interpreting these as numbers, but as strings, which are to be combined into one really big string.

So here's where I'm going to reveal something important. Whenever we have a dataframe, every column has a type. And here, all my types are object. Now what I want in order for some to behave the way I want it, I want the population not to be object, but some sort of numerical type. Now if you've never dealt with types in a language, this may seem a little mysterious. But the basic idea is that every column is annotated in a secret place with what kind of data it is. And that will affect how sum behaves. So we want to convert this column into numbers. Okay? So in order to do that, There's something called pd dot numeric. It's a function which takes a non-numeric table, or column, and converts it into a numerical. So I'm going to tell our dataframe: please convert my dataframe population — my column called "population" — of my dataframe into a number. However, when I try to run this, I run into some trouble. It says, Unable to parse string 49, 506,799. The problem is that the pd.to_numeric does not understand how to handle commas, somewhat surprisingly. So we need to get rid of them.

So, in order to do that, I'm going to say I want to replace the population column with a string line as replacement, where any comma is going to be replaced with nothing. So commas become nothing. That's the syntax. And so if I do that, and now I look at my data, we can see that my population numbers — note this is not just the Telugu states, this is the population — they are now numbers that don't have commas. And we still have this stuff, which is going to be a problem, but we've made things better. So now if we try to take this population column, and convert it into numbers, it will fail again. In this case, it fails because it doesn't understand what to do with brackets 48.

So you might say — and this is the reason I'm doing this on a real-world data — I'll just do that exact same thing, Josh. It's going to be no problem. I'm going to say df dot "population". Actually, first, I'm just going to show it on one line. .str.replace. I want to get rid of 48 with nothing. So if I do that, I get back all of my numbers. But now the 48 should be gone. However, if you look here, this funny thing, There's still brackets, right? I said, don't take this 48 and break brackets and get rid of it. And I'm left with just brackets. And, even weirder — and you're really not going to like this — the data has been manipulated in a funny way. Notice that the first row originally had a population which was 4,9,5,0, etc. Now, because of my replacement operation, 4 9 5 0 has become 9 5 0. The top digit is missing. This will cause you to pull your hair out. You will be so sad about why this is happening. This is a complete mystery and something that would be really, really hard to Google. And so that's why I'm introducing it here in this video.

The exact reason this happens, is because of something known as a regular expression, but it is beyond the scope of our class. So I'm not going to go into the full details. Instead, what I need to do — and this is just for your benefit, but this is something you'll need to dig in further in order to understand deeply. Because the brackets are a very special thing to this replace function, I need to put a backslash before them to escape them. Okay? So now you're probably feeling a little nervous about manipulating string data. But that's part of the fun of learning about these libraries. There's a lot of little details that can really bite you. If I instead do this backslash bracket, now, it's no longer taking away fours that were at the front, and my little braces here are gone. So, I'll show you the difference. That's the way you think you should do it.

But the problem there is if a number starts with a four or an eight, that number disappears and these braces stay. Whereas here, if I do this, we end up with keeping all of our values. Actually, now that I think about it, what this code does is it gets rid of any four and any eight, period. Right? So if any number had a four or eight in it then it's gone. But by adding these special backslashes before the braces, that weird behavior goes away. Now again, I have just dropped you in the deepest of possible deep ends. So I don't want you to panic about this. And I don't expect you to fully understand it, but nonetheless, I want you to see it. So that if you had those sort of problems, you at least have seen something vaguely like this. And the ultimate phrase, really, if you wanted to learn this well, would be regular expression. Okay, so great. So we're finally dug ourselves out of this hole. And once we do that, we get a population value, which will be nice.

Okay, so now, our dataframe. We've gotten rid of those weird brackets in the population corner that had 48. And we've also gotten rid of our commas and so now this time converting to numeric works. And specifically, what you'll see is that the result of this to_numeric function, it has a data type which is int64. Or, if we change the df population column to be the result of this, and then I ask for my data types, I see that the population is now considered to be a number, or an int64.

Now again, I am revealing things that are really pretty low-level, and are not necessarily super important to practitioners of machine learning and data science. Especially if you're just doing data analytics. But nonetheless, it's something that will pop up even if the actual details are a little messy. So now, finally, what we should be able to do is take our data and we can only get Telugu states. And hopefully we get back our sum. And indeed we do. And even though I misspelled Telugu here, we have an answer, and we know roughly how many people live in states in India where they speak Telugu.

## Video 9: Real-World High-Dimensional Data

The very last thing we're going to do in terms of programming today, is we're going to look at a real-world dataset. This is a housing dataset that I got from Kaggle. Now, if you're not familiar with Kaggle, it is a platform for machine learning and data science, that originally started out as a contest platform where people would basically show off their skills at doing machine learning. But these days it's actually a much bigger thing. I won't describe its entire history and what it does. But I will say that they have a lot of really great datasets, like this housing dataset

This housing dataset provides, for a very narrow region of time, a set of sales of houses. And so each row represents one sale. So for example, this top row, this is a house which sold in 2008. It was a Normal sale, whatever that means. And its sale price was $208,000. Down here we have a house that sold in 2006. It was an Abnormal sale, again, I don't know what that means, for a $140,000. Now the reason I want to show this example to you, that this is very typical of a real-world data science for machine learning problem where we have quite a large table. Here there are 81 columns, meaning there's 81 different dimensions of data that we might possibly consider. Now, each of these columns is something that could be potentially useful to us, if we're trying to tell a story about this data.

And what we're going to do, is we're going to be considering how the sale price varies as a function of these other properties. This is a very natural question if you've worked in real estate — I want to predict how much the household will sell for. Now, to understand what's in this dataset, we can look at the columns up top. But we'll notice that there are so many columns in this dataset that not even all are showing.

So if I want to see the columns, I'll say df.columns. And there I get a list of all my columns. So, this is a new piece of syntax we haven't seen before. So there are a lot of columns here, and they have somewhat mysterious names — some of them — like MSZoning or RoofMatl. It's probably roof material. We have bedroom above ground, and so forth. But if we wanted to really know what this dataset means, ideally, when you're given a big dataset like this in the real-world, someone will have annotated it and told you what the columns mean.

Indeed, it is the case that on Kaggle where I got this dataset, they have a description of the data called data_description.text. And here are the meanings of each of the different features that we have. So OverallCond is the the overall condition rating. ExterCond is the present condition of the material on the exterior. We have, for example, 1stFlrSF is first floor square feet, and so forth. GrLivArea, which we'll use later, is the above-ground living area. Now, ideally, these names would tell you right off the bat what they mean. But sometimes you need something like this to understand. So, for example, when I first saw GrLivArea, I assumed it meant gross living area, but actually it's above-ground living area. Okay. So those are our columns.

Now, whenever we have a real-world dataset, we often want to explore it to understand what's in it. And so, to give you an example of this, let's consider this "Street" feature here. So, if we look just at "Street" first, we'll see that the value Pave seems to appear a lot. And so, if we want to summarize this, a piece of syntax you may not have seen before is value_counts. value_counts will actually count the number of times Pave or anything else, shows up, and give you a concise summary of this field. When we do it, we see that paved streets represent 1,454, and GrVl, which probably means gravel, only six. And so what that tells me, is that if I were trying to build a model of sale prices for houses in this set, "Street" is probably not going to be very useful. Because Pave occurs almost in all the rows. Now we could do this with other features of this dataset, and I encourage you to try it. But that's the kind of thing I might do to figure out what features might be useful.

Alright. What I'm going to show you next, is we want to try next to show the sale price as a function of some of the other features. And so, looking at these features mean many of them could be useful. But to me, one of the most intuitive measures of a house's value is its square footage. So I want to know what are the different conditions, or what are the different columns, that contain information about the area of the house. We see like GarageArea and PoolArea. But what else is out there? So we could go to our data description, and then we could Control-F and look for the word area everywhere. We can also do that in code by saying columns, please tell me if you contain the word area. If they do, we'll get back True, and if they don't, we'll get back False. Now given one of these binary arrays, we can say df.columns and then index into it with that. That's just the same indexing trick we've been using since the very beginning. And when we do that, we get back LotArea, MasVnrArea, GrLivArea, etc. So we might be, for example, curious about what this means. And we see it means masonry veneer area in square feet. And if you're not a native English speaker, you'll probably have to look up what this means. So given these, I think the above-ground living area is likely to be something related to the sale price.

And so given that, I wanna make a plot. When I do this, I see that indeed, what do you know? Big houses tend to sell for more money. And so that says that in principle, a machine learning model could try to learn something. It could try to predict sale prices. Now that's only one of our 81 different dimensions of data. And so we might try and use other things, like the number of bathrooms, and the number of bedrooms, or whatever else. And so I'm going to show you what happens if we use, I don't know, say, the overall quality. So the "OverallQual", without having really dug into it, we don't know exactly what we're going to get. But when I do this, I'm going to be basically adding a third dimension to my plot. So instead of just the x, y-coordinate, I'm not also going to have the color. And I'll get back another plot. Now what does this tell me? What this tells me is that houses which are rated overall quality higher, tend to have higher sale prices. So you'll see that yellowy, orangey houses tend to be up here. And then lower-quality houses, in this blue purple range, they tend to be at the bottom. And so what that suggests to me, is that a model which uses both the living area and the quality could potentially do better.

You can add more dimensions to your plot if you really wanted. You could do like, for example, the size as the number of bathrooms. So say bathrooms, FullBath, and maybe you'll get something here. Maybe not. Okay. Alright. So we could see that, yeah, we know the number of FullBath. There's one bath there. This one has three bathrooms. And this one's a little harder to see, because my plots' getting a little busy. And so the point here, is sometimes it's hard to tell in advance exactly what's going to happen when you create these plots.

And as you add things like size to your plots, you don't always get a good story out of it. And so what we've got here is basically the beginnings of an analysis that we're going to be continuing to discuss in the future. And really, the key point here is that by exploring this data, and understanding it, we have the sense that we will be able to build effective machine learning models. And we'll have to wait until a future module in order to see how well it really works.

## Video 10: The Consequences of Using the Wrong Data

Much of what you're learning in this course is, appropriately, how to build models that can find hidden patterns in data, make predictions, or explain the world in a way that humans cannot otherwise. The excitement around machine learning, data science, whatever you wanna call it, is about the power of computational models to create some form of artificial intelligence. That's probably a big reason you're here. This realization, and the excitement, can lead many people to think about the technology and the algorithms as the answer to many questions. Too many.

I want to talk briefly about the other important ingredient: data. We're spending a lot of time on how to manage and structure data. But I want to spend a moment thinking about something much more basic. How do you find the right data? What is the right data for your problem? These issues are much more mundane, but these kinds of questions are central to any successful data science project. In fact, you can probably stop with simple regressions, or comparisons of means, and focus only on getting better data and you'd be better off. Not that I think we should, we can do both. We can have great data and find the right models — not the most complicated for our task.

One of the most important lessons in applying tools from data science, is that data science is a means, not an end. That is, you need to think about what you're trying to develop in context. Perhaps you're developing a piece of technology product. Or maybe you're trying to answer a strategic question, like how to set the price of something, or whether to pay for an advertisement, and where to place it. That's where you want to start. Too often data scientists have a hammer — or a random forest or deep learning — and go around looking for a nail. But start by asking what the problem is you need to solve. Then ask what data you have, and what data you'd like to have in an ideal world. Do all this before you even get to how you're going to develop your model.

I want to give you an example of how choosing the wrong data can cause problems — big problems, in this case — even if you do everything else right. The example comes from a fantastic paper in science by my colleague, Ziad Obermeyer, here at Berkeley, and some co-authors. They studied predictive algorithms in health. The scenario of huge excitement in health care — from large technology companies, health care companies to national health care systems, and many, many digital health startups. There's great enthusiasm that AI can help predict bad health outcomes, and allow doctors, and nurses, and others to intervene before that happens. It's really exciting. This can improve people's health and potentially save money for the system. It's a big opportunity. To put it in perspective, in the US, one third of deaths come from diseases that can be prevented by behavioral change. And the country spends almost 20 percent of GDP on health care. So, if we can figure out how to use data science to affect this, we can do well and we can do good, to say the least.

In the paper, they studied one widely-used algorithm that has been developed to do precisely this kind of thing. They used detailed data on health care claims — that's the data on what was done, that is transmitted to an insurer — and electronic medical records, the information your doctor gathers in a visit, to try to find out who was at very high risk of illness. This was done by developing a predictive model of risk and then assigning interventions — in this case, nurses who are available any time to support care to individuals above a threshold. To do this though, we need a measure of health risk. That is, we need the right data. Now the good news is, the company developing the algorithm, had detailed data on health care utilization for millions of people in the US. Now take a moment. If you want you can pause and think about how you'd try to do this. What data would you want?

The algorithm developers decided that the best metric for identifying illness was how much money was spent on individuals. This gives a uniform measure of total care used, assuming that sicker people need more health care. Based on this decision, they hired a huge team of data scientists and physicians to develop a prediction algorithm. They use incredibly detailed data on every visit to the doctor, lab values, hospitalizations, numerous other inputs, to predict the total amount of spending an individual would have in the future. The algorithm performed very well in terms of predictive performance. All of the metrics you have learned, or will learn, looked really good. Thus, they rolled this out commercially as a tool. And it was widely adopted. In fact, this algorithm, or similar ones, are estimated impact care for 200 million people in the US annually. They're big business. And they're potentially improving the care delivered to many of the sickest Americans and, similarly, in other countries. What's not to like? What the authors of the study found though, was a big problem. The algorithm was racially biased.

This of course alarmed the developers, the hospitals, the health insurers using it, and many others. But why? In this case, the bias stemmed from one seemingly simple decision: which data to use. Specifically, the decision to predict cost of care as an outcome. Let me walk you through that, because it illustrates how important these decisions are. For a variety of reasons related to the history of racism and the medical system, as well as social factors, black patients use, on average, less care. Thus, relying on cost — which is based on actually receiving care — systematically makes black patients look lower risk. At the same level of true illness, they use significantly less care.

To see this, let's look at a better measure of actual illness — the true number of chronic conditions. This plot captures figure 1 from the paper. The x-axis is a measure of risk based on the algorithm. And the y-axis is the number of chronic conditions. The vertical dotted lines give you the threshold beyond which action was taken based on the algorithm. The right one is the 97th percentile. And above this, substantial resources were targeted, and interventions were implemented. The other line, the 55th percentile, was used to alert primary care doctors of risk. The two curves plot the fitted relationship between risk, measured by the algorithm, and the chronic conditions for black and white patients separately. You can see very clearly where the problem comes from. At the same level of risk, that's any point on the x-axis, the black patients are systematically sicker. Remember, the algorithm was supposed to be telling you how sick someone is, regardless of race. In fact, if we look at the threshold for intervening, that's the 97th percentile, black patients have 27% more chronic conditions. This is also true looking at important biomarkers for health. Black patients have more severe hypertension, diabetes, renal failure, and anemia, and high cholesterol, for the same risk score. The bias in the algorithm has real impacts for how care is delivered to black patients. If you were to instead use chronic conditions as your outcome, the share of black patients assigned to the intervention program would have been 46.6%, almost half of them, compared to only 17% in the actual implementation.

The authors of the study were able to look at the actual variables selected in developing the algorithm, giving them more insight into how and why the bias arose. Race doesn't even enter into the algorithm at all. You might think this is helpful in avoiding bias. The issue, as we've discussed though, is cost. At a given level of health, measured by the number of chronic conditions, black patients spent less. A lot less $1,800 a year, on average, less. Thus, when we build an algorithm to predict cost, we get just that. A good prediction of cost. And if we call cost health, though, black patients look healthier than they actually are.

There is good news here too. The authors of the study worked with the algorithm developer to reduce bias. It turns out getting better data, or different data, is key. By using a better measure of actual health— not just cost — the bias is reduced by 84%. This is a specific case in a specific industry. However, I hope it makes clear why thinking about what you're studying and asking what the strategic goal is. Do you want to predict health or spending? And getting the right data matters for how algorithms are implemented and how they perform.

Before you start building and testing models, start by understanding the setting you're in, your goals, and finding the best data you can on what you want the model to predict, and the inputs to that prediction.

## Video 11: Conclusion

In this module, we got additional practice with our core data processing skills. These skills are invaluable for many purposes, including data science, data analytics, and preparing data for machine learning. Let's review some of the key ideas that we covered in this module. First, we saw how to join tables using pd.merge, and discussed the difference between outer, left, right, and inner joins.

We also saw how to join tables based on a single column, or on a combination of columns. We then spent a fair amount of time trying to visualize and tell stories about a combined dataset that was the result of joining GDP, population, life expectancy, and continent tables. Through those visualizations, we saw a number of new plotting techniques and functions provided by our various plotting libraries, including violin plots, rug plots, kernel density estimates, 2D heat maps, and marginal plots. We also saw how to manipulate text data, using the powerful .str commands provided by pandas.

And lastly, we wrapped up our analyses today by looking at a very high-dimensional house price data set. Here, by high-dimensional, we mean our dataset contained a large number of columns. Real-world data's often big, messy, and not very well documented. So, it's important to get some experience poking around in such datasets.

In closing, I'll note that this moment is a major milestone. We're now done with the foundational part of our course, Woo-hoo! Now there are still many fundamental data processing and data manipulation skills to learn. But what we have to do next is get started with machine learning. So, we'll see you next time.