Week 13

Video Transcripts

Video 1 (6:28): Data cleaning with pandas (Part 1)

Today, what we're going to do is we're going to... we're going to take a look at how we can use Pandas as our starting point, really, for analyzing data and visualizing data in Python—so very useful package. And, we'll see it's very flexible and can do a lot of stuff with it. So, what we're going to do actually is we'll work with this file over here that... that I've got that's called, 'nyc_311_data_subset.csv'. This is a subset of data that's available on New York City's open source—open data project, and what this contains is every complaint that is called in to New York's 311 number, is recorded, and registered in this file. What I've done is–the file is huge, there's lots of—there are many, many fields and the data goes back to 2010, so there's many, many gigabytes of data sitting out there—what I've done is I've taken a subset of this data.

I've taken a few columns, not very many—and two months of data. I think it's September and October of last year, of 2016, and that's about it. So, even that's quite...quite a lot—so, it's...it's more than a...more than a 100 megabytes in size. And, I have a zip file that is loaded up on your—on our course site, which you should probably download and work with. You can download it, unzip it, and keep it in the same directory as where your Jupyter Notebook is running from. So, that's where we're going to start with. So, the first thing of course, when you work with data is you've got to get the data.

So, let's say that's our data file. We'll import Pandas as 'pd' and NumPy as 'np', and we're going to use the function called, 'read_csv', which is a very flexible file—very flexible function and can read data from, you know, multiple places. Though it's called csv, it's actually very, very flexible in reading stuff from almost anywhere. And, the documentation is that URL is provided over there. So, let's read that and all we need to do really is to say, "Read csv and give it the name of the data file." So, we do that and it's going to take a little bit of time because it's a very, very large file and—not that long, okay! So—did it much faster here than my machine. And, we get this data out here. And, what we can see is that our data contains several columns. There's a 'Unique key' column, a 'Created Date' column, a 'Closed Date' column, 'Agency', 'Incidence Zip', 'Borough', 'Latitude', and 'Longitude'. So, this data is pretty clear what it is. 'Created Date' is the date at which the complaint was received—and the date and time the complaint was received, and that's recorded over here.

'Closed Date' is the date and time that the complaint was handled effectively and closed. The 'Agency' is the agency responsible, and there are many different agencies. This is the Department of Sanitation in New York. There's the New York Police Department. There's the Housing Department. There's the Board of Education. There's all kinds of stuff, right! 'Incident Zip' is the zip code for which the complaint was, for...for where the complaint is. So, for example, if you have a heating complaint and you report it to the city, then the zip code of your apartment is what's going to show up over there. The 'Borough' is the borough of New York City. There are five boroughs in New York City–and Queens, Brooklyn, Manhattan,

Staten Island, and the Bronx. So, that's the 'Borough'. It tells you which borough it was in. And then, we have the 'Latitude' and 'Longitude' of the location of the complaint. So, if—for example, if it's a heating complaint, then that's latitude and longitude of your apartment, and that's what's going to show up over there. So, that's the thing here. And, we see one of them is called, 'Unique Key'. So, it's kind of nice to see that it's– first of all, it says unique, which is very helpful because we know it's unique.
But, we can take a look at our data itself, and see what the kind of data we have. So, this tells us here that we have various columns, so 'Unique Key' etc. It tells us how many rows of data are there. Remember, this is two months' data. So, you can imagine how much data—what size of data we would have, if we had downloaded everything in 2010 to whatever today's date is. So, there are 971,000 records of 'Unique Key'. We find that some of them like, 'Closed Date' has fewer records. So, we can see that over here. That's a 'Closed Date', and that's 'NaN' because we have two months' of data, and though, of course, this data should be more than that but, this—what this tells us is this complaint, whatever it is that was recorded on this date, hasn't been closed as of the time that I downloaded the data from there, right! So, that's—so, those are NaNs. The agency is, well, everything has an agency, sometimes zips are missing, and there could be many reasons for that. The 'Borough' is always there. And, the 'Latitude' and 'Longitude' could be missing as well, right! For example, maybe if it's a moving violation or something, then you don't exactly know where it is recorded. Maybe, I don't really know. So, we see that this is our—the 'data.info()' command tells us what the structure of our data file is, of our Panda data frame is, because we loaded all of this into—I should have pointed that out, It's all got loaded into a Panda's data frame, right, so, it's the data frame here, this thing with columns and this thing and all that stuff. And, it looks like from our data file—it looks like we already said the unique key—the name tells us it's a unique key and it looks like it really is a unique key, therefore, what we can do is we can make that into an index. It's not going to be really useful for us, but it's a good idea when you're loading a data file to see what the index is because then, if you want to access some, you know data, really, really quickly, you can use the index. Also, Pandas allows us to maintain multiple data frames, and then use SQL, like Joins to join them together. So, knowing the key for—on the index for a particular data frame is be really, really useful, right! We saw that in SQL. So, we can think of Pandas as—it's a table, so multiple tables. You can use an SQL like syntax to access data across multiple tables, if you want to. So, we'll make that that. And then, to do that all we need to do is to say, "Hey! When you're reading the csv, we are saying, 'Hey, the index call is unique key.'" All right! That's what we can say there. So, let's look at the first ten records here, yes! We have a nice key over there, and that's good, okay!

Video 2 (6:15): Data cleaning with pandas (Part 2)

Now, the next thing it tells us—if you are reading this stuff here, is it says–if we can—if I can highlight this, it says, column four has mixed types. What that means is that what—when Pandas is reading a csv file or reading data from anywhere, it tries to guess the type of that column. When it says mixed types, then what it means is it's unable to figure out what the exact type is. And, it—some would be integers, some would be strings, and it's not a single, uniform type, which is not so good for us because

remember, if we want to do analysis, we want...we want...we want to use an NP array kind of structure, which means that we want the data to be of the same type in...in a particular column. So, let's see what the mixed type stuff has to do. What...what's going on with the mixed types in Incidence Zip. Column four is Incidence Zip. That's one of the mixed types. And, the best way to do that is to use this function called 'unique'. What 'unique' does is it pulls out unique values of—in a column. So, we get this unique stuff and we look at it, and we find that there's some—there's a NaN over here, right! Then, we have zips as strings, many zips as strings, right! And, let's scroll down further. We find that sometimes in United States all zips are five digits, but sometimes we find that there's five digits followed by dash, followed by a four-digit number, right! So, that's another problem, right, and this says something, because the postal service uses these extra four digits for more precise identification of locations.
So, that shows up in some places. I guess, if the person filing the complaint has provided it, then we have a zip 'UNKNOWN', which is not very helpful. We can scroll down further and we look and see. They're all strings here. Here's some strings. Although they are strings, some strings have a decimal point in them. So, we've got to get rid of that because zips don't have decimal points. Then, we find that in some places, it actually identifies them as integer—as floating point numbers. So, we've got a mixed data type coming from this stuff here, that sometimes they are strings, sometimes they are floating point numbers. And, notice that there's the question mark here that we need to get rid of, as well. We scroll further down, further down, and see all the same problems, and then, there's a 'JFK'. We've got a mix of stuff here. And, our—roughly what we are saying is that sometimes a zip is a float, and other times it's an 'str'. It's sometimes zip codes that are represented as floats, especially the ones that start with a 0, are missing the first digit.
So, we look at this part for example. This is a four-digit zip, which is not correct. It's a five-digit zip, and the reason it's a four-digit zip is because it is represented as a floating point number, and the zero that came in front, which has vanished, because floating point numbers don't have zeros in front. So, that's the—though...though in our—when reading it came as a string, but somewhere along the way, it got entered as a number. The reason for this probably is the way it was entered in the database. Maybe, there are multiple entry points, bulk—you know, maybe it's entered on the web, it's entered on an application at the 311 person's desk, it's entered later somewhere, or in a file directly, I don't really know. But, there...there are probably multiple ways in which the data is entered. And, we're seeing that this data is messy because of those kind of reasons.
And, we have to clean up that mess because we have to deal with this data. So, we've got this...this. So, we have this. We have missing–the 0 missing. We have the four-digit extension. We have—somewhere in this stuff I saw there was a zip code '0'. Let's assume it's there. And, what the heck is that? You know, '0' zip code. And then, we have NaNs, and there's question mark. There's an 'UNKNOWN', there's a 'JFK'. There's some random stuff sitting inside, and we've got to figure out what to do with those, as well. So, what we need to do is we need to...we need to clean the data. And, cleaning the data is really a bit of an art. It's a mix of a science and an art, I should say. And, the reasons it's an art is because while cleaning the data you want to make sure you don't lose anything of value. It's easy to say, "Hey, let's just get rid of everything that doesn't make sense." That—that's a...that's problematic, but that—and you might throw out something important at the same time, as you're cleaning this stuff out.

So, it's a bit of an art and a science. So, we...we should keep that in mind as we go through this process here. So, the first thing we want to do is–what do we do with this 'JFK', 'UNKNOWN', etc? Ideally, we should probably keep 'JFK'. The reason is that 'JFK' is a J—John F Kennedy National Airport. It's a well-recognized entity. So, we should probably keep it, and maybe give it a dummy zip code or something like that, and keep it in our database. But for simplicity, I'm going to throw it out, right! But, that's one of the considerations that you should think about. So, what you're going to do is anything that we decide is bad, we're going to throw the entire record out, okay! We're going to say we don't want that record there at all. And, that may not be the right thing to do. Maybe we should convert them to NaNs and keep those records. But for simplicity, we're just going to do that. Keep...keep that in mind, right! Then, we want to make sure that everything is in the same format. And, we're going to convert everything–that's a zip, so we will convert everything into strings. The idea of converting into strings is that for one thing, it will keep a zero in front. Though, it doesn't matter in our case because New York City doesn't have any zip codes with zero in the front, but still, just for, I guess, for the future we'll just say, "All right! Zip codes are strings. So, we'll keep them as strings." And, we've got to decide what to do with the 'missing values' and the 'NaNs'. So, what we're going to do is we're going to take all the bad data and make them NaNs. And then, we're going to make sure that everything is a...is a string, and it's a proper integer like one, two, one, one, one or something like that. And then, what–decide what to do with missing values, the 'NaNs'. We make a decision about that. And, the decision we're going to make is we just throw them out. So, —and one other thing we'll do is, for now, is we'll also drop all zips that are less than '10000' and greater than '19999'. The reason for that is that New York City zips are all within this range. And, we're going to assume that for now, we don't care about any incident that is recorded–that is in a zip code that is not in New York City.

Video 3 (7:04): Data cleaning with pandas (Part 3)

So, how do we do this? Well, the easiest way to do things is always is to write a function that does it, right! So, we're going to write a function called, 'fix_zip'. That will take the zip code and output something, in whatever format, the zip code that generated by a file here. There are many of these zip codes, and return a zip code in a five-digit string that is hopefully a valid one, okay! So, that's our goal here, and this is a very straightforward function. So, let me just walkthrough these function—steps quickly. So, what we'll do is first we know that our zips are a mix of strings and integers, right—strings...strings and floating point numbers. And, we know that the strings that are not convertible into integers are bad zips because they have like, UNKNOWN and, you know, all that kind of stuff in them. So, what we'll do is we'll start by saying, "All right, we'll try to convert an 'input_zip' first into floating point number, right!", because we can't directly convert a string of this sort into an integer. So, we first convert to a floating point number, and then convert that into an integer. So, if we succeed in this step, right—so this is our thing.
If we get a 'input_zip' over here, then that's good. If we don't get an 'input_zip', then we couldn't do the conversion. Then, we want to check whether the reason, the conversion failed was because it was a five-digit zip followed by a dash, followed by a four-digit extension. So, what we do then is we do another

'try', and the try-excepts are really useful. So, we're going to get practice with them. We try again to take the zip and split it on a dash, and then get the first part, that is, what is before the dash, right! So, we have this over here. What we're going to do is, we're going to split it on the dash here and then, extract this part of the—from the...from the zip code, right, '55486'. So, we get that. And, we try to convert that into an integer. If that succeeds, then great. Otherwise, we just return a NaN. And the NaN...NaN, in—we use the NumPy NaN to represent a NaN. If we add this—so...so this is our 'try'. So, if this succeeds, right! If this succeeds, if we get beyond this point, then what we have is either a...a 'input_zip' that is actually an integer, right! Or, an 'input_zip' that is a NaN, one or the other. There's nothing else we're going to get here. So, if we get to the point where it's integer, then we check– see whether it's less than this or greater than this, which is our New York City zip code limits.

And, if it's not, then we return a NaN there, because we don't want to handle any zip codes that are not in New York City, and then, if that succeeds, that means if our zip code is both an integer as well as between '10000' and '19999', then we convert into a string and return it. So, that's our 'fix_zip'. So, let's just take a look at whether 'fix_zip' works or not. So, we can run this. We test it. And, let's try with this. That worked, right! It takes '11211'. Let's try dash 0072. That worked. Let's try, 'UNKNOWN' That returned a NaN. So, it looks like we've...we've we have a working proposition over here. So, that's our 'fix_zip' function. And now, what we want to do is we want to take that 'Incident Zip' column. So, if you—the column—zips are in this column that's labeled, 'Incident Zip'.

And, if you recall from our last week's lecture, the way you extract a column is, you just use the dictionary syntax here. So we get this, 'Incident Zip' column. So, what we are going to do is we're going to take the 'Incident Zip' column and use this function 'apply' to apply 'fix_zip' onto every element of that column. So, 'apply' is a really nice function to use in Pandas, and you—in the Panda dataframe you use it a lot, when you're working with data. Well, what you can do is you can take a column, and then take a function, and apply that function to each element in that column, and...and do some conversion on it. So, what this does is, it says—takes our Incident Zip, and I'm going to just take this here. Insert. So, if we look at data 'Incident Zip'. So, this is what it looks like, right! We got all this stuff here, with NaNs and numbers, and things like that. And, what I want to do is I want to take this and 'apply' 'fix_zip' to it. I get this revised look at it, right, which gives us all our zips in nice format over there, with NaNs and things, where it doesn't have to be, right! So, this is what it converts to.

But, of course, we are not saving this anywhere, so we need to actually save this. And, to save this, we want to do this over here so that we have a location on the left-hand side. Remember, I should have an assignment to save this stuff there. So, we do this, and that takes—converts a new 'Incident Zip' column. And then, I look at the unique values. And, I notice they're all nicely str'd, four-digit stuff. So, we are in good shape here. And finally, we can get rid of all the nos... all the rows that have a NaN. And for this, what we are going to do is we're going to use this function called, 'notnull'. So, what we are doing here is, this is a way of selecting a bunch of rows from a dataframe. What we are saying is take the dataframe data, which is our data that we've read in, okay! And, include only those rows that have this thing to be true, right, this condition to be true. And, the condition we are saying is that the value of the 'Incident Zip' cell in that row should not be null. Because 'notnull' is a function that just says not null, right! So, what this will do is it'll select the rows that are in the column that are–and now if you look at our selected rows that have a not, a non-NaN in the Incident Zip column.

And, if you look at this data info, what we get now is '910907', which is hopefully a bit lower than what we had before. Let's go back up. '971063', right! So, we lost a few rows, quite a few rows actually in our—by...by removing the NaN through Incident Zips.

Video 4 (9:06): Data cleaning with pandas (Part 4)

When we look at the columns in–for 'Closed Data', 'Latitude' and 'Longitude', they all have missing values. We can take a look at the info again. And, it tells us that this 829000 versus 910000, and obviously, there are missing values here. Similarly, for 'Latitude' and 'Longitude', we have missing values as well. So, let's get rid of all of them, you know, what the heck. We don't really want missing values. So, we'll throw them all out. And this, we can do very simply by using the same procedure that we used for 'Incident Zip'. But, here we're going to use a logical operator in the middle of the conditions. And, in the Pandas world, the logical operator that we use in conditions is not the a-n-d or o-r or n-o-t, as in Python. But we use an ampersand as an and.
This would be an or. I am sorry. This would be an or, the pipe character, and this would be a not, the exclamation point. So, that's how that works inside Pandas. So, what we're going to say here is that we want to include only those lines that have a 'notnull' value for 'latitude', a 'notnull' value for 'longitude', and 'notnull' value for 'Closed Date'. So that should take care of that. Let's run this and look at the info here. And now, we're down to 806000, and they're all the same. So now, we have a dataframe that contains data that is all notnull, right! We've hopefully got rid of all the null values.
So, that's the next step. The next thing we can do is we can start looking at the 'Borough' data and see–our goal is to clean the data. So, what we really want is a dataframe that contains data that is meaningful to us. So, we want to look at every column to see whether the data in that thing makes sense or not. So, let's take a look at the second column, the 'Borough' column, and another column, 'Borough' column, we look at that, and we look at the unique values there. And, we find that we have six values, even though there are only five Boroughs. And, there's one 'Borough' that says, 'Unspecified', right! So, that's a problem for us, right! What are–what is an unspecified Borough? So, we...we can of course remove it. But, before removing anything, we should also look at the data itself to make sure whether that removal is not going to cause us problems down the road. And, what are the problems down the road? You don't want to remove data systematically from your dataframe.
Because otherwise your analysis is going to miss out on some systemic property that is useful, right! So, you typically want to see that if you're removing data–that the... that the data is occurring reasonably randomly inside it– the...the rather the problematic data is occurring reasonably randomly inside your data. If that is the case, then you can get rid of it. It's not going to affect your analysis. However, if there's...if there's systematic reason, why we have 'Unspecified' in our database, then we should probably keep it, okay, because we want some– may be something useful that we get out of it. So, let's see what we get for here. So, what I want to do is, I'm going to say, "All right, show me the data that has the value of Borough unspecified." So, this is what we are saying here, 'data' 'Borough' equals equals 'Unspecified'. And, we are extracting this from the data. And from there, we are going to look at the two

columns 'Agency' and 'Incident Zip', just to see whether there is some, whether the 'Unspecified' 'Borough' occurs systematically across 'Agency' or 'Incidence Zip', right!

So, that's the goal here. In reality, like I said, there's something like I think, 60 or 70 columns in that...in that database. So, you really need to look at it in more– with the entire data set. But, we are working with a smaller data set because of limitations of a, the speed of processing, and the fact that we have to upload and download all this data. So, look at this here. And, we run this through here. And, we look at this and we look as–just looking, eyeballing it, it looks like there's lots of 'NYPD' in this thing here, right! Lots of them are 'NYPD'. So, we want to find out now, is that really the case. So, let's take a closer look at this. So, what we'll do is we're going to use the 'groupby' function—and we'll look at this in more detail later, but the 'groupby' function, what it does is, it groups data in a dataframe by groups, which is somewhat the SQL groupby that we saw last week.

It does pretty much the same thing. So, what we're going to do is, we're going to look for the—extract the data for 'Unspecified' 'Borough', then group it by the 'Agency' so that we get the 'Agency' groups by 'NYPD', DOS, DOE, and Department of Sanitation, etc., and come out with a count by groups. Okay! So, we can take a look and see whether we have some kind of a bias here in this. So, we look through this stuff here, and we find that we've got—well, this is all—it's going to be the same in every column, but we have '67' column, '67' rows for the...the housing department, that's 'DHS', I think it's housing, while I'm not so sure, and '725' for the New York Police Department. So, clearly there is a bias here of some sort, because '725' cases are 'NYPD' cases. So, the New York Police Department tends to do a lot of unspecified stuff.

What we want to do next is then find out, is this a significant proportion of 'NYPD' cases. Maybe, if it's a small proportion, we can still ignore it. And I–like I said, there's a–there very many more columns here. So then maybe, if we had all the columns, we might find that there are certain kinds of 'NYPD' complaints that come with unspecified Borough, like maybe they're 'TLC' complaints, complaints on moving violations of a certain type, okay, something like that. We don't know, right! We don't have all the data. But that will be what we would want to see. We want to see whether there is some systematic reason for that.

But right now, all we know is that hey, we've got '725' cases of 'Unspecified' 'Boroughs' for 'NYPD', and we want to find out are these a significant proportion of NYPD complaints or not. So, let's see if we can figure that out. So, what we'll do then is we'll say, "All right, we want to find the total 'NYPD' complaints." So, let me run this piece by piece so we can take a look at each step. Comment that. Comment that. Comment that. And, I'll run this. So this–what this does is, it's–we are counting here the number of cases where the 'Agency' is 'NYPD'. Very straightforward. So, let's see what that is. Complaints total. It does this '274408'. So, we got '725' out of that. It doesn't look like a huge amount but, let's get the exact number–exact percentage. So, here what we're going to look at is, how many are–let's–[inaudible] run that. So, it's '725'. So, we've got–these are the cases where we have 'NYPD' and 'Unspecified' 'Borough'. And then, we can divide the first number, sorry, the second the number by the first number to figure out the percent of cases where 'NYPD' is 'Unspecified' and it is 'NYPD', so, percentage of 'NYPD' complaints that are 'Unspecified', that have an 'Unspecified' 'Borough'.

And, that's '0.26' percent. It's not a huge number, and like I said, what we would like to do of course is take this data and look at the entire data set and then, see whether there is some kind of systematic

reason for these '0.26' percent cases. But, we don't have all that. So, what we'll do is, we'll just chuck them, okay! We'll throw them out. And–because for one thing, it's going to be hard to explain that when we are presenting our results to management or whatever. We'll say Unspecified Borough, and they'll say what the heck is that. So, what we'll do is, we'll take our data and say include only those elements in our data where the 'Borough' is not equal to 'Unspecified'. So, this is where 'Borough' is Manhattan, Queens, Brooklyn, Bronx or Staten Island. So, this node is our not sign over here, all right, it says not equal to. This is actually pretty common in computer programming languages. Python is a little bit of an exception with that A and D stuff. So that takes us through to that. And, let's take a look at our thing now. So, it's down below. And, what do we have now? 'data' dot 'info'. And, we are now down to 805000 from 970000 approximately. So, we lost quite a bit of data in this process. So, we have...we need to be careful, right! We have lost all, I think, like almost 170000 lines of data. So that's pretty hefty chunk of our data, 10 percent, more than 10. So, we should be little bit careful when we think about this. If you're losing a lot of data, we're going to lose a lot of meaning in our analysis as well, right!

Video 5 (6:26): Data cleaning with pandas (Part 5)

Let's take a look at how we can deal with time in a dataframe. Dates and times are best converted to 'datetime'. The reason is very straightforward. Because once we have 'datetime' objects, we can do reason—we can reason about time. We can subtract two pieces of time. We can add time, we can do all kinds of stuff with it. You can't really do that with time that comes as, in the form of 2016, blah, blah, because—or ten, 26, 2016 because, simply because it's not the—you don't have the timeline, the linear timeline that we live with in life, right! So, since we...we don't want to do—so we want to deal with time as 'datetime' objects, the first thing we need to do, is take our 'Created Date', which is currently a string, and figure out how to convert that to time.
So, let's take a look at our data once again, just to be sure we are on the right page here. So, I'll do 'data' dot 'info'. And, here we see that 'Created Date' and 'Closed Date' are—just says, nonnull...nonnull object, right! So, a nonnull object is basically just date—a string. And, we want to make sure that it's a string, but none of them are null, right! That's what this is saying here. So, we want to take this thing and convert into a 'datetime' object. So, let's see how we can do that. Well again, we have a column. The column contains time as strings. And, we want to take each element of that time as string, and convert it into a 'datetime' object. So, what we can do is we can use the 'apply' function again. The 'apply' function, remember, takes a column– takes every value in the column and applies a function to it. And, we're going to apply to this–each time element, we're going to apply this special function called lambda function, which we haven't talked about before. lambda functions are called–are also known as anonymous functions. They're just like functions. They are exactly like our fix_zip function, except that they don't have a name, and they are created at the point where they're used, right!
So, they are not like precreated functions, but they're created at the point that they're used. So here, we are saying–so you can think of lambda as a kind of the name of the function, except that it's not really a name. And then, you give it an argument. And the argument here is 'x', right! So in–normally, we would write something like this. We probably say 'def' 'eggs' 'x' and then, do 'stuff' and then, 'return x' or

something, right, 'x' or some...some transformation of 'x'. Let's say, 'x' star 'x', right, some kind of transformation of 'x'. So, that's what a function is like. So here, in a...in a lambda, anonymous function, we start with the keyword lambda that tells Python that we're dealing with an anonymous function. We give it the argument that we want to 'x' that is same as this 'x' over here and then, put a colon and then, we put a transformation. So clearly, you can see the transformation for this to be effective has to be reasonably simple. Otherwise, you're going to get really messy in this stuff. And usually, lambda functions are used for simple transformations. So, our transformation here is we're going to call this 'strptime' function. And, you might recall from our second week, when we looked at 'datetime' objects that the 'strptime'...'strptime', s-t-r-p time function what it does is, it takes a string, 'x', and takes another string, which is a format, and applies that format to 'x' and converts it into a 'datetime' object. So, since our data is of the sort like this, the format of our date, dates are in the string format are month slash day slash year there's a space, then there's hours, colon, minutes, colon seconds another space and then, the 'AM' after that. So, the format we need is to correspond to that and, that's what this does here. It says we have a month, percent 'm' followed by the slash that we see in this here, this slash, followed by the day, followed by the slash, followed by a percent 'Y', which is the year– and this says the year is in four-digit format, the uppercase 'Y' and then, there's another space that's this space over here– format has to be exactly what it is, right– that space and then, percent 'I'. Percent 'I' says that the hours are given in AM, PM format, twelve-hour format, right! And, we want to convert–the datetime keeps hours, it keeps it in 24-hout format. So, we want to convert that into that. So we say, "Hey, that's in 12-hour format". So, that's percent 'I'." That's the minutes. That's the seconds. The colon is a separator. We see the colon as a separator. It's just matching exactly what this looks like, right! It's the same structure. Followed by another space, which is the space over here between–before AM or before PM.

And then, percent 'P', which signifies that we're going to get an AM or a PM in this location, the letters AM, PM in that location. And, that–we're going to apply this conversion. So, it's going to take each string x of this type, example like this, that's the value of x, apply this format to it and convert it to a 'datetime', and do it for the entire created date thing. So let's check that. And, once they're done, we can look at the top 20 created dates and see what it does. It's a little bit slow, this one. It's working through a lot of stuff. And now, we get this. We notice it's a different looking thing. And, we can check and see if that is indeed the case by looking at 'data' dot 'info'. So now, it says that Created Date is a datetime, 64-bit datetime in– object, right! So, this is...this is the type that we want. This is the special Pandas datetime structure, okay! So, we're going to do the same thing for 'Closed Date'. So, let's run that. And, we know that there are no NaNs, because we got rid of them. You could have had NaNs in which case, you would need to do, probably write a function to do the conversion. Because it's not going to work as well.

Video 6 (7:13): Data cleaning with pandas (Part 6)

And once, we do that then, we get a new data set, which looks something like this. And, you now have these dates, Closed Dates blah, blah all that kind of stuff. So, what we want to do, of course, is we—if

you're looking at complaints, and we're looking at created time and closed time, Created Date and Closed Date, the time it was created and the time it got closed then, we would like to know, one probably one reasonable set—direction of analysis is to see how long it takes to close these complaints, and see whether there are some systematic reasons for delays or whatever. I mean, if...if you were managing the 311 system, that's what we would like to know, like where does it take longer. Where does it take shorter? Like, what's going on? Are we closing stuff in a timely fashion? You know, those kind of things, right! So, what we're going to do is we want to essentially look at the time that it takes to close it, which is this time, the close time minus the open time.

That tells us how long the complaint was open and unresolved in our system. So, that becomes easy now because we have–our times here are datetime objects. All we need to do is to take 'Closed Date' minus 'Created Date', and we can create a new column. Remember, in Pandas, you can create new columns just by adding a name to it. So, this is going to create a new column in data called 'processing time', and that's the difference between these two. So, let's run that. And, we can take a look at this and see statistics over there. So, 'describe', remember tells us, describes the–for any numerical or calculatible column, because datetimes are not technically numerical. It can do some calculations. So, this tells us that the mean time to close a complaint was five days, five hours, 11 point five three' minutes, right! 11 five three, so five days, zero hours, five minutes and 11 point five three seconds. The standard deviation is 12 days, etc. The minimum is 134–negative 134 days. Well, that's a problem, right! We can't really be closing complaints before they occur.

That doesn't make any sense. But, we'll come back to that. The maximum is '148 days'. That also sounds like a lot. We will check that out. And we–this 'describe' also gives us the quartiles– we saw that last week again, but just sort of remind you again what it...what it tells us, is that the first quartile, the bottom quartile, that is the fastest closer time was between negative 134 days and two hours, 34 minutes. The second was between two hours, 34 minutes and 21 hours, ten minutes. It looks pretty good, actually, 50 percent of them, assuming that this negative 134 makes sense, are closed within...within a day, right! So, that's not so bad. Then, the next one is '4 days' and then, the last one is '148 days', '4 days' to '148 days'. The next one is 21 hours to '4 days', and the last the...the slowest closures are from '4 days' to '148 days'. So, that's our very quick way of looking at this stuff here. What we want to do now is we want to look at the odd stuff. The odd stuff is–what the heck is negative processing time? And, since our data is for two months, we want to see whether '148 days' is actually sensible or not. Two answers 60 or 61 or 62 days, right! Not more than that. So, and this is September and October, so it's 61 days. So, we want to see whether '148 days'–is, you know, something wrong with that? Is it bad data? Or, is it really the correct thing?

So, let's start by looking at the negative processing time. And, we can take a look at it that–very straightforward again, by looking at our–from our data, extracting the rows that have a 'timedelta' less than zero, right! That's all the negative stuff. So, let's take a look at that. And, we get this here and we find that you know the– this is very confusing. Because, it tells us that the complaint here for example was on 20th of October, this one, and it closed on 19th of October. Most likely when thinking of this, looking at this data, most likely what happened is that the closing date and opening date got–creation date and closing date got interchanged, that will be my guess. However, we don't know that. And, this is something that as a data analyst you probably need to go back to the client or the person, who gave you

the data and see, is that what's going on. If that's going–that's what's going on, then you should interchange these two values for these– for any negative thing here. If there's something else going on, then, you know, we have to take care of that. For now, because we don't have access to the client, to the 311 data's people, we are going to say, "All right, we will probably just dump them." Okay!
But in reality, when you're cleaning data, you have to go back and try to figure out why something has happened– why something strange is going on in the database. So, let's take a–look at the other side. That is the stuff that is 148 days and over. So, the...the maximum that we saw here was '148 days', 13 hours, this one here, '148 days', 13 hours, and ten minutes. So, let's take a look at that...that what's more than 148 days, right! So, we say 'timedelta' of 148 days, and we extract that data from our data set, and we find that two such cases. And looking at this, we see that it's opened on September first and closed on January 28th. So, that is September, October, November, December. That's about a 120. Another 20, 148. That looks reasonable, right! I don't see anything wrong with that. Similarly, this opened on September first and closed on October, on January 27$_{th}$ of the next year. So, this also probably also is a valid date. So, we'll say, "All right! We will take care of–148 days seems okay." So, we are okay with this stuff, right!
So, anything that took longer than 60 days is probably all right in our database. So, upper end makes sense, in other words, and the negative times basically don't, right. Oops, the negative times don't, but the upper end makes sense. So, what we need to explore this, like I've said, and I want to reiterate that really we shouldn't be throwing data out willy-nilly the way I am doing over here. We need to understand the data better, and go back to our–the person who gave us the data or the organization that gathered data, and understand why there is something inside it. Because, we never want to remove data that is– will be useful in our analysis. Because, we're going to get a biased analysis. Then, it's better to have no analysis than to have biased results. You know, that's the bottom line, okay! So, you really need to work on this very carefully. So, what we'll do is we will–for now just throw the negatives, and say that we only will include data that has time greater than equal to zero.
I mean, actually probably should make zero–one second, but–or greater than zero, but whatever. So, we do that and we get a new database here. And, we can do 'data' dot oh. I can't do that again.

Video 7 (4:42): Data cleaning with pandas (Part 7)

So, what we'll do at this point now is we write a function that incorporates all the changes that we made in our thing over here, and this is a very, very important step, though it sounds trivial. The reason is that you're going to be working with the data again and again, and again. That's not like you take a piece of data and you sit down, and you press a few buttons and write a couple scripts, and you produce nice reports and send them back there again. Data analysis is an art as well as a science. You have to–you'll be doing things repeatedly, looking at results and then trying to figure out why they don't make sense, and then going back and doing them again, you know, all that kind of stuff. And, you don't want to go through all these data cleaning steps that we have in our—we've done today.
We don't want to go through them again and again, and again. So, what you want to do is you want to write a function that does that for you and then, you can call that function whatever you need it. So,

let's just walk through that very quickly and see what we did here. So, our function says, 'read_311_data'. That's what we're going to call it here and we give it a data file. We do our imports over here. This is an important step because you want functions to be independent. That means that they should be able to work and you...you shouldn't have to be importing stuff outside the function. So, whatever function—whatever imports you need inside a function, always keep them in the function. That's an—in Python, it's not going to hurt you in any way. So, you...you should do that. Then, we have a 'fix_zip', which is a function we wrote here. And, notice we've included this function inside the function. It's packaged together. As far as we are concerned, we're going to use 'fix_zip' inside 311 data. So, wherever we get—whenever we want to import data from the 311 site, and we could do this again, and again, and again, right, because the data keeps changing every day then, what we want to do is we want to make sure that 'fix_zip' is inside it.

We might need to modify it at some point later, but for now, we want to make sure we always include that. Then, we've got that. Then, we read the file. Then, we've make sure we get the key there. We fix the zip by applying 'fix_zip', right! That's what we did next. We take out any roles that have NaNs. And, notice this is a little bit different from what I did earlier. So, what I did earlier was that I was explicitly saying, "Hey, if you have a row that has a NaN, if you have Created Date that has a NaN or a Closed Date that has a NaN or a Incident Zip that has a NaN, I was specifying those explicitly." All right! Now, what I'm saying over here is that–you–Pandas [inaudible] has one nice function called 'dropna'. And, what that does is you can tell it, "Hey, go into my dataframe and any row that has a NaN any– in any column should be dropped." Okay, there's a–options are how equals 'any' and how equals 'all'. So, in– if you say how equals 'all', then it'll drop a row that has all NaNs, but keep rows that have one or more, one or two or three NaNs. And, if you say 'any', if a row has even one NaN in it in any of the columns, it'll drop it. So, it's a convenient way of getting rid of NaN data. So, like I said, bearing in mind that we may not want to do that.

We have—for example, if are looking only at processing time, then we don't care if it Incident Zip is a NaN or not, if we have the complete Created Date and Closed Date. So, you want to—I mean, this is a sort of a very—it's like a taking a hatchet to your data, but, you know, so you want to use it very carefully. Then, we want to get rid of Boroughs that are 'Unspecified', so we get rid of those. Any Borough that says 'Unspecified', we get rid of it. We do the datetime conversions and compute—so these are our datetime conversions. This one creates 'Created Date' 'Closed Date', just what we did before. And then, we compute the processing time over here and once you got all that we get rid of a negative processing times and return the completed dataframe, okay! So that's what we have here, and we can test this out. And we are running it, oops! I should probably run that We're still getting the error, because this other–the warning because we read it first, and then drop the stuff. And, once we do this, we're doing all our processing, which is why it's a little bit slower now because it's doing—creating the...the processing times, converting dates, which was kind of slow, if you remember, and you know all that stuff. And, we should see that result here, and we finally get this over here. We find the file—we now have a dataframe that contains 'Created Date' as datetimes, a 'Closed Date' datetime, processing time as a timedelta, and then everything else as either strings or floating point numbers.

Video 8 (9:16): Data visualization: Part 1

The idea there is that we take our data, we look at, examine it in as much detail as possible, and that really means getting down and looking at the actual values, looking at what the data is telling us in terms of what it contains, and then trying to figure out what kinds of anomalies we're finding in the data you know, data that looks problematic, and then removing any data that is problematic but is not going to bias our results, That's our goal with the data cleaning process. So once we've done that, we can actually start visualizing the data and looking at it for purposes of presenting our results or figuring out what the data is trying to tell us. So what we're going to look at today is how we can visualize data with the...with the very basic Python tools on data visualization. And before we do anything of course, we need to read the data. Fortunately, we have the very nice function that we wrote and we can just call that, so include the function on top of this file here called data visualization, and we can load that function and read our data and that's going to take a minute or two, and while that does that, we will look at what we do next. So there are a lot of stuff we can do, and we'll start with a very simple, but very, very effective way of looking at data which is taking your data and plotting it on Google Maps. That's—everyone understands Google Maps. So if you ever need to present data to anybody else and you present it on Google Maps, they're going figure out what the heck—you know they are going to see what's going on.

So what we'll do is we have data here for two months of 311 complaints, and these complaints are across the city. And the data contains latitudes and longitudes for every data item, right, for every complaint. And what we want to do is, we can take a look at that here now. That's actually run. So we have latitudes and longitudes here. This tells us the location of the complaint, right, and a fairly precise location. So what we can do is we can say, "All right, let's see how these complaints are distributed across the city." Are there regions of the city where the complaints are more frequent and are there regions of the city where they are less frequent. And then, maybe that will tell us— give us some insight into how these complaints occur. And if for example, we were at the city and we were allocating resources to handle complaints and they needed physical resources on the ground so to speak, we can allocate those resources based upon the density of complaints across the city. So we have latitude and longitude. The first thing you want to do of course is to install this library called GM plot.

So the library we're going to use for this which is Google Map plot, I guess, is GM plot. So you want to just install that. So let me install that here. I don't have it on this machine. Right, so that got installed, so you also want to do that. And remember, the installation of library is very straight forward in Python. You do a pip install and give the name. Though I have an upgrade over here, I don't need it in this case when I'm doing a first-time installation. The dash dash upgrade is when you want to upgrade from a previous version to a new version, all right! So in this case—in this particular machine it's a clean installation and it's not my machine. This is a machine provided for the class. So...so that's there.

And what we can do now we will—what we want to do is we want to provide a—draw finally a heatmap that helps us see the relative concentration of complaints using the latitudes and longitudes. So what we're going to do is set up the map itself. So GM plot is pretty straight forward. What it says is that you are going to do a Google Map. Now Google Maps is the entire world, right! So we want our Google Map to be centered at a particular location. The two ways of using GM plot to center the...the map, and what

we're going to do is—you can use either method. One method is to actually provide a latitude and a longitude. And if you—then the map center will be there. The other method is to give a location and if...if the...if GM plot understands the location, then it will center it right there, right!

So New York it understands very well and it centers it over there. And then the next thing you can do is you can tell it what your zoom factor is, and this might require some trial and error to get it right. So for example, I'm using a city, so in a city I want to zoom in quite a bit because I want to zoom into the city itself. What if I want to say am doing a heatmap for the United States, then I probably want to zoom out so I can see the whole United States on the screen. I don't just want to see New York City on the screen. So this is going to be centered on New York with a zoom factor of ten. Like I said, we have see if that makes sense or not. And a lot depends on various factors on that stuff there. So let's do that. So we've got the map centered. The next thing to do is to actually generate the heatmap itself and this is actually really, really simple. We have two vectors, a latitude vector and a longitude vector. We've removed all lines that contain even one NA from our table, right, from our data frame. So that means that every row has a latitude as well as a longitude and there is data in there, right! We haven't checked for bad latitudes and longitudes. It's conceivable that there's a mistake. You know, rather than a latitude of 40.7, this...this should be negative 74, actually, because it's New York. It's west of the Greenwich—of the international whatever that thing is, date line. Is it date line? No, it's called the time line, the Greenwich Median Time line.

So this is—where has it gone? Here. So we've got latitude of 40.744, negative 70.73, etc. So it's conceivable that some of these are bad. I don't know. I haven't checked. But we should probably be checking those as well. So what we would like to do and this will be something you want to think about is—look for a bounding box. A bounding box is a frame that you can put on a location and say that all points in that location are between this latitude and this longitude, right! So—or given a latitude to the south, a latitude to the north, a longitude to the east and a longitude to the west and you know it's in that place there. So that way you can check and see if there is a bad data entered. People make mistakes when entering data. You know, it's not uncommon. So you want to make sure of that. But for now we will assume that it's all good. So here we take the data latitude which is our latitude column, data longitude, which is our longitude column, and draw a heatmap. That's about it. Once you have written the file to—using 'GMap'dot'draw', you have written the file to insert inside HTML, you can view this. Just be really careful. You can view it on Chrome.

It doesn't view on Safari and I'm not sure about other browsers. So on Chrome it will show up like this, and we get a nice little view of our heatmap. So what this is showing, like I said, it shows you a bunch of dots. And the concentration of the dots, as it increases, you get a—as the density increases you get a redder color. It goes from very yellow to green to red and shows you how the density increases. So this tells us if you look at this map here, we can see the density is quiet around here. There is a peripheral density around in Queens. There's a slightly lower density in this area of Queens. Staten Island has a very low density except near the ferry, where the Staten Island ferry is. So probably something to do with the ferry I guess. I don't really know. And that's the thing in here. And we can zoom a little bit into this map here. It's of course Google Maps so it's completely interactive. And we can zoom in a bit. It's a bit slow to render, but it is there. And we get a little better view of what's going on here. Because what we see is that— if you look at this thing over here, we can see that we've got the red spots are in upper

Manhattan. And I don't know if you guys can see this on your screen, but upper Manhattan is where it says 9A, 9A, that area is upper Manhattan. Some red spots there. There's red spots in the Bronx. And then in lower Manhattan towards the east river or down here, there's the red spots. And then Brooklyn a little bit. And a lot of this is also to do with the fact that the months that I have taken are winter months.

And there likely are more heating complaints, and a lot of the city-owned houses are in upper Manhattan, the Bronx, and in eastern Brooklyn. So that's where we tend to see—and lower Manhattan and eastern Brooklyn. So this is where we tend to see the complaints, right! So that's probably something to do with that. What we would like to do of course is take all seven or eight years of data and look at the heatmap for that, but that's going to be a very, very intensive process. So that's the idea here. And getting back to this, so a heatmap is a very very quick and convenient way of conveying information to whoever is interested in that information.

Video 9 (7:01): Data visualization: Part 2

Now let's do some grouping operations. The grouping operations are probably the easiest way of getting sense of your data. You have some categorical variables and you want to group your data by those categorical variables, compute things like means, the count, you know, the standard deviation and those kinds of things by group. So that you get some sense for how your—what are the differences across different categories inside your data. So, in our data we have two major categories that we can think of. There are Agencies and there are Boroughs. So, we can start by grouping data by Borough. So, the first thing you want to do is, of course, since you're going to draw lots of graphs, is to in-line our matplotlib stuff so that it shows up in the screen. And then, we group the data by Borough. So, we say 'data' dot 'group' by 'borough' and that's going to group it, as simple as that. We compute the size of each group, so it's going to count the number of elements in each group.

And then, we plot it and say we wanted to do a bar graph, and then we get this nice little bar graph that tells us that the largest complaint— number of complaints is in Brooklyn by quite a order of magnitude 250,000. And then, we have Queens, Manhattan in second place pretty much close to each other, with the Bronx pretty close to that, and Staten Island way down with very few complaints. It makes sense because Staten Island, though it's a fairly large Borough actually has a very low population relative to...relative to the other Boroughs, right! But this does tell us that Brooklyn, you know, is a place where there are lots of complaints and it will be nice to see what they are. Actually, in the previous demonstration, I looked at the complaints for Brooklyn and a lot of them are noise-related. Apparently, in Brooklyn noise is a big deal. But we don't have all that data. I've eliminated that data because of the size issue here. So, if you have the time, you should probably go to the 311 site, download the larger data set and explore that. But, for now we can...we can just work with this.

So, Brooklyn has got the highest. So that tells us straight away that Brooklyn is, you know, a big complaint zone with tons of complaints there. We can look at them by Agency, and that may be more interesting also. So again, we do the same thing, we group by Agency and we plot it, and we get this over here, with many more agencies of course, so this tells us that most complaints are New York

City...New York Police Department complaints, followed by the housing—HPD which is the housing department I think, complaints. And, that is probably again due to the fact that we have winter months, so they are more likely to be heat-related complaints. Actually we have September and October, so we might have some air conditioner complaints and then heat-related complaints. But housing complaints are up there and then transportation complaints and energy protection and you know, all that kind of stuff. So, this one gives us an indication of what sort of complaints we're getting. In general, if you get a— if you're running an agency of like 311 and you find that you—two of your agencies are skewing all the results, perhaps you should have a separate department that just takes care of that stuff. But anyway, that's not our problem.

So, we have Agencies and we have Boroughs. So, the next thing you might want to do is, if I want to see like for which Agencies are—for which Boroughs which Agencies are more likely to be complaint zones, like for example, NYPD is huge, but what if NYPD is huge only in Manhattan but not anywhere else or only in Queens and not anywhere else, so we want to see whether there are some differences in the way complaints occur by Agency across different Boroughs. So, we can combine them into a single group by— and what you—when you want to do a multiple group by multiple things, what you're going to get is a two-layered grouping. So, this says here group by Agency and then group by Borough. So, we're going to get a two-layered grouping, where it will first group Agency, and within Agency, it will group everything by Borough, okay! So, we get agency NYPD, Borough one, Borough two, Borough three, Borough four, Borough five. Agency DOT, Borough one, Borough two, Borough three, Borough four, Borough five, that kind of stuff, right! And we can do that and we can draw a nice little graph for this, and we get this graph here. And we see, oh my goodness gracious, we can't even read this stuff, right! This is densely populated. And even, if you could spread it out and read it, it's very hard to make...make out differences across this because what this is really saying is NYPD Manhattan, NYPD Staten Island, that kind of stuff, right, which is kind of meaningless.

So, what we want to do is what matplotlib allows us to do is it allows us to unstack this so that we can now plot them, so that we can group them by one or the other, right! So, let's unstack this and take a look at it. So unstack this, and what I get is, I get for each Borough—for each Agency—because Agency is my first level, right! So remember, I take every Agency and I—within the Agency I group by Borough, right! So, I have the first group by Agency and then inside that I group by Borough. So, what I'm going to get is for every Agency, I'm going to get the Borough grouping. So now, we get a slightly different graph here, which is a lot more readable. And, we can see there are you know—it tells us for each— for NYPD it tells us the Borough... relative Borough complaints—number of complaints. And, what we can do is, still it's a pretty small graph, so what we can do is we can make it bigger. And to make it bigger, we use this fix size parameter to our plot function. And what the fix size parameter does is it tells you how big you want the figure, and you can again play around with these numbers to see what kind of size you get. And we give it a title, which is kind of nice. And we'll do exactly what we did before, except a slightly larger figure. And we get a nice big figure here that looks like this. It's a bit too big I think. So what I can do is I can make it a little bit smaller, make it 12, 12. Run that again, and see if it fits in the screen.

Yep. That's a little bit better, right! I can actually read it. So this—looking at this now, we get some more information. So, it tells us that Manhattan is green and Brooklyn is orange, Bronx is blue. So if I look at NYPD complaints, I find that the most number of complaints are in Brooklyn in— and the second most

complaints in Brooklyn are—the second most complaints are in Queens on the New York Police Department, right! Whereas if I look at HPD, then we see the most complaints are in Brooklyn followed by the Bronx, followed by Manhattan and then Queens is way down there, you know, far fewer complaints for HPD, and so on and so forth. So, I can actually go and look at each Agency or flip it around and look at each Borough and see if there are differences. So to flip it around, all I would need to do is to replace— to interchange these two. So instead of Agency, Borough I would have Borough, Agency. And that flips them around and I can see the—I would see a graph that was by Borough, but it will show you the bars here would be for each Agency, and that's it. So, this is another way of looking at the data and trying to get some sense out of it.

Video 10 (11:33): Groupby function pandas

Let's digress a little bit and take a more detailed look at the Pandas 'groupby' function. It's a very powerful function. It's very useful to understand how that works. And, you can do a lot with it, right! We...we're doing fairly simple stuff, but we can actually do quite a lot with it. So, let's say we start with a little dataframe. And, dataframe here is–contains—let's take a look at the dataframe. It contains four columns, 'Age', 'Author', 'Country', and 'Gender'. So, it's for various authors. And, what we do is, we've got four authors in our dataframe, 'George Orwell', 'John Steinbeck', 'Pearl Buck', and 'Agatha Christie'. And we know which country they come from the 'UK', 'USA', 'USA', and 'UK' And, they have genders available, and ages—I just made up–I think–I suspect when I made these, I think they're the age that they actually died at.
So, they're all no longer with us, so to speak. And that's the day–the age they died at, right! So, we got that. Now, we can group these by country. So, that's–we know how to do that. So, we have a column called 'Country'. It's a categorical column. So, all we need to do is to say, 'writers' dot 'groupby' 'Country', and we get the–let me close this, and we get the groups by country, right! So, we can take a look at what these groups look like, and just to see here. The country is 'UK' and the groups are integers, '0' and '3'. So, these '0' and '3' refer to the...in…in the index column. We don't have a specific index column here. So, it's telling us that 'UK' is row is '0' and it's row '3', which is what it is. And 'USA' is row '1', and row '2', which is where it is. So, that's just grouping them by the index values, right! So, which...which makes sense. That's what it should do. We can do some basic analysis on the groups.
So, we can see, what is the first element in each group. So, that gives us the 'grouped' dot 'first', tells us that 'George Orwell' and 'John Steinbeck' are the first elements in each–in the–in each of the groups, right, the 'UK' group and the 'USA' group. So, this is our groups, 'UK' and 'USA', the first item there. We can look at the last item, and it'll tell us it's 'Agatha Christie' in the 'UK' group and 'Pearl Buck' in the 'USA' group, not that really helps us that much, but it's a good way of looking at the data just to see examples, right! When you have large numbers of groups, you can do a 'grouped' dot 'first' to see what other items in each group, what kind of...kind of items are in each group. So, you can get a result. More interesting would be, we could add the groups and get a sum of the groups. What that does, in this case, is our—the column that we have that is summable is the 'Age' column, right! So, what we can do is, we can—what this we'll do is it'll add up the 'Age' columns, and 'Age' columns here are '46' plus '85' for 'UK',

and '66' plus '80' for 'USA', and we get the sum here. We can find the mean, and that's the mean of the sum. And we can apply any function, though we already have the 'sum' function, I can apply any function to it. And when we apply the function, what it's doing is, it's applying the function to the row and then, returning the summary of the other stuff there.
So, we get here, 'UK' is '131'. The author–both the authors are included here, and both the genders are included here. So, we get a kind of summary of this whole thing, through the 'apply' function. We're only applying it on the group, and the group contains all the data for that group, okay! That's the idea here. So, we've already seen the groups, so that's good. We can group by multiple columns. We saw that too. So, we want to group by country and by gender. So, we do that. And, we get a grouping now that is grouped by country and gender. So, we get 'UK', female, and that contains row '3', which was 'Agatha Christie', and 'UK', male, which contains row '0', which was 'George Orwell', this one here. And 'UK'– 'USA', female, which was 'Pearl Buck', and that was row '2'. And 'USA', male that was 'John Steinbeck' and that's Row '1'. So, that's a multi-level grouping. And, we can group by we can–well, we can do even better things than that. We...we have been grouping here in our data. So, if you look at my data again– let me just insert a set above, writers. So, I've been grouping by 'Country', 'Gender'– these are categorical columns, and we have age, which is the numerical column. I might want to...might want to group by age groups, you know, so, in the sense that I–like let's say I have a column with multiple, with a continuous variable and I might want group them by values of the variable, less than ten, greater than ten, that kind of–greater than ten, less than 20, that kind of stuff. What I can do is, I can write a function. So, here what I am going to do in this example is, I want to group by age groups. I want to say group in such a way that the age is less than 30, or the age is less than 60, okay! I would have a young age–a young person, a middle person, and whether they died young, they died in middle age, or they died in old age. So, they died at the age of less than 30–when they died, then they died when they were young, which actually is nobody here. If the age is less than 60 when they died, then they're middle aged, and that's George Orwell. And if otherwise, they died in old age, okay! That's the idea here. So, what I do is I write a function called 'age_groups'. And what I want to do is, I want to take this value, '46', for example, and decide on the group based on the...on the value itself. So, what I need to do is, I need to pass to age groups a dataframe, that is, the entire dataframe, writers, with that's where I'm grouping from. I need to pass the index of a row. So, the row index would be '0', and the name of the column that I want to group on– so, if I'm grouping on age, this column will be 'Age', okay! So, what this–these two together will do is, they are going to identify a single cell in the data–in my dataframe.
And now, I get a–so, I get a dataframe and a single set on the dataframe, and I look at the value of that single cell, and that determines–depending on that value, I return the grouping that I want for that cell, whether the young, middle or old. Does that make sense? Right! So, again, what I'm doing is, getting a single value for the cell, and using that single value to decide on when–what kind of group it falls into. And, notice that the labels I'm giving are my own. I can give whatever labels I want over here. So, I can create my own groupings. So, when I do this, and I run the function, so what I want to take is, take 'writers' and I want to apply 'groupby'. And for 'groupby', I want to apply this function here, where I vary the index, because I want to go row by row. Remember, when I group stuff, I group them in groups of rows. You know, I'm taking subsets of the rows and including them in groups. So, I want to take every row, extract from that row the specific cell that I'm looking at from the dataframe, right! This is the

dataframe, and this is the specific cell I'm looking at. And, notice that since I'm including the dataframe here, it...it could be an independent dataframe. It doesn't have to be the same dataframe. I could be grouping one dataframe by values in another dataframe, perfectly alright, okay! And I get this...this thing over there. So, now what–so, this gives me the cell. So, this will–this function will return 'young', 'middle', or 'old', and depending on that, the data will be grouped in 'young', 'middle', or 'old'.
So, let's run this and we get here—we don't have any 'young' age person because '46' is the lowest here. So, we get–'middle' is row '0', which is 'George Orwell'. And, 'old' is everybody else, which are 'John Steinbeck', 'Pearl Buck', and 'Agatha Christie'. So, that's a very convenient way of grouping stuff together. And we can–to take another example, we can also group, you know—just take a...take a different example here. Let's say, we have a bunch of people and—let me run this thing here. Oops! I need to import NumPy. I should have done that up front. I don't know why I didn't do that. 'import NumPy as np', 'import pandas as pd', just to be on the safe side. So, here what I'm doing is, I want to combine a few things over here, since it's worth looking at this, in more detail. So, what I'm doing is, I'm creating a dataframe that has five columns, 'a', 'b', 'c', 'd', 'e', and five rows. And each row, has an index, 'Joe', 'Steve', the name of a person. And, I'm randomly putting numbers between positive '5' and '5', inside this thing here, right! So, these are all random numbers. And, if I run this again, of course, I am going to get a completely different set over there. So, I'm just creating this stuff here, calling the 'np' dot 'random' function, which we looked at the NumPy random last time. So, I'm looking at a random number, normally distributed between '5' and–sorry. I'm...I'm sorry, I should have–I misstated completely over there. I'm looking at a random of '5' by '5' matrix of random numbers that are normally distributed with a mean zero, and a standard deviation of one. So, that's still the numbers I get. So, we see here, we've got 1.14, 0.79, negative 0.44, etc., right! So, what we want to do is, we want to write a function that takes three arguments again, a dataframe index and a column, and it returns a grouping for that row.
So, we want to group stuff, if it's greater than zero, it goes in 'Group 1', otherwise it goes...goes in 'Group 2'. Okay! So, the reason I'm doing this is because we get more numbers and we can actually do some statistical stuff on it, right! So, let's see, this is again, straightforward. We pass it the dataframe—the function, the data frame, an index and a column, right! And we give it the index and the column to zero in on a single cell, right, using the 'loc' function. And, we say if that's greater than zero, then we return 'Group 1', otherwise return 'Group 2'. So, let's run that and create the groups. So, we get these groups. 'Group 1' has 'Joe', 'Steve', 'Jim'. 'Group 2' has 'Wes' and 'Travis'. And now, we can actually compute statistics of these groups. This is kind of nice for us. So, we've got all this stuff here. So, I can–I'm just going to recreate those groups here, by using the function. I haven't saved it anywhere. So, I'm going to use this group creation and then run mean or standard deviation on it, and actually, get data, right! So, get...get stats as a single value. So, this tells us that 'Group 1' has a mean of 1.15, Group 2, a negative 0.97, and a standard deviation of 0.37, and 'Group 2' has a standard deviation of 0.73. So, you can already see that if you were like, for example, looking at say, historical returns on a...on a stock and we wanted to group them by perhaps the industry group or...or even by, let's say the...the–let's say the trading volume, and you want to look the returns, or even maybe some—like the one month prior return, or like a lagging one month return with a...with a say a two month return or one month return

two months ago, with a one month return last month, and then compare the mean and standard deviation of the returns, we could do that quite easily using this method over here.
So, grouping is a...is a very very versatile function to–very versatile function to use in dataframes. It's versatile because you can group by anything at all, and once you have the groups, you can find mean, standard deviations, or apply any function that you want to that grouping to do comparison across groups.

Video 11 (11:04): Data visualization: Part 3

Next take, let's take a look at the time angle in this. So, we really have the creation date for every incident. We can do things like building an incident for the number of incidents by month. This would be very useful if we had all of our data, if we had data running from '2010' to today, we could look at the number of incidents per month and see there's some kind of seasonality in these industries. We could see seasonality by Agency, seasonality by 'Borough'. We could do a lot of stuff for that. We only have two months' data, so we can't really do that kind of stuff, but if you have the time and a fast enough Internet connection to download the entire data and a powerful enough computer—actually it doesn't require a great deal of power, some kind of, you know, you probably need like four gigabytes RAM or something like that to process all that stuff in a reasonable amount of time.
You can actually run this through the entire data set. So, we're going to need to do some data manipulation for this. So, we...we want to do the month…the number of incidents by month. So, what we have is data that is in datetime format, and what we want to do is, we want to extract from that the...the year and the month. So we, what we can do here is—luckily for us, they're all, you know, '2010', '11, '12', they are in increasing order. So, we can actually just extract everything as a YYYYMM format from this thing here. And to do that, we take our created date column and apply this function to it, 'strftime' is a function that takes a datetime object and converts it into a string. And so we're going to convert our datetime object into a percent Y, percent M string. It's as simple as that. And, we do that and we get a new column that is 'data YYYYMM' and 'data YYYYMM' because it's a new column here. And let's take a look at it.
Oops, I didn't run that. String the extraction. And now we see we have this nice column with the strings in it, right! So, that's essentially what we do. We manipulate columns, we change them. We do stuff with them, so we manipulate this column and got a new one and now we can do a group by... by using 'yyyymm' and 'agency'. And, let's do a graph here and we saw earlier that '15, 15' was a bit too big. So, I want to make this '12, 12' and we run that and we get this by Agency. So, this tells us 'NYPD' has the highest here. But in '11', the brown line which is housing department has gone up. '12' brown line has shot up even more in the winter months. So, they are the colder months. People are making, I guess turning on their radiators and finding that they don't work. So, we see the brown line goes from September, it shoots up in October and November and then shoots really high in December, right! This is when people are realizing the heating doesn't work and then they're complaining about it. Then it falls again in January I guess because by then if you haven't got the heating working, you probably can't live there anymore. It would be just too cold. So that's something over here. If you have more months, my

guess is in summer, this brown line will be really really low because there won't be as many heating complaints but there could be other complaints, we don't know.
Maybe there are other complaints that occur in...in city housing in summer, maybe air conditioning complaints go up. I don't know, right! So, that's worth looking at. And then, otherwise, the 'NYPD' complaints tend to go down in winter for some reason. Actually, I'm not really sure whether these are—maybe, it's hard to see though, right! This could be 'NYPD', it could be 'DOE'. I'm not sure which one it is because this...this one is actually a little bit darker than that. So, we need to take a look at it by examining each column and seeing whether they are really the same or not. Probably 'NYPD'. We saw 'NYPD' had the most number of complaints. So, we can do a seasonal analysis by that.
We can look at the agencies and see which agencies have the highest ones. So, here I really want to just show you how we can draw multiple graphs, you know. So, like for example here, we group by Agency, look at the size and then we can call this function sort values which is a Pandas function. And, what sort values does is it sorts the values that it gets inside from—. Here we get a size, so the size is going to be the number of records in each...for each...in each group, Agency group. And, we can sort those and we can see that 'NYPD' has the highest, '273'. For the 'HPD', '244', and 'DOE' is really down there. So, none of those are 'DOE'. They're too low for that and we can set ascending equal to false, if you want it to be in descending order, and to true if you want it to be, by default it's set to true, so that it would be in ascending order. We can plot this. This gives us a graph like this. If we have many many of these things, we can try to restrict them if you want but for now, let's do 'agency' 'borough', 'unstack' this, right! So, this is our 'agency' 'borough' setup, and this tells us what we've done is we've taken the grouping of 'Agency' and 'Borough'. We compute the size here and unstack it and then we look at the data here.
So this tells us, since we have the count, this tells us, for each 'Borough', what the count is for each agency, and interestingly 3, 11 agency there as well. Okay, so we've got all the counts here. And we can...we can graph this, but what we... what we could do is, we could create a bunch of graphs, so like for example, we have five Boroughs, right, and we want to see each 'Borough' next to each other. So, what we can do is, we can create five graphs, one for each Borough and in each Borough's graph we show the agency plots and then we put these side by side. So, you get a nice set of—like a bunch of subplots and you get a...a very visual view of how these Boroughs differ in terms of the agencies that they complain to, right!
So, how do we do that? Well, what we want is, a set of side-by-side graphs, right, plots in this case. So, we want to have five plots. So, we do them in two plots in row one, two plots in row two and one plot in row three but our frame has to have six boxes because it has to be symmetrical, right! So, we need two two two. So, what we're going to do is, we're going to say we have two columns in our graph plot. And you know what, let me draw it and we can take a look at it. We get something like this, right! So, we say, we have '2' columns and '3' rows. So, the '3' rows are—we have 'Bronx' Brooklyn' in row one, 'Manhattan' 'Queens' in row two and 'Staten Island' in row three. So, this is an easy way of comparing stuff. So, we set up that. Then, we set up my 'matplotlib' dot 'pyplot' to create a...a...a figure and the figure contains subplots. So subplots, each of these is a subplot. 'Bronx', 'Brooklyn' these are subplots. And, each subplot is in a particular, each subplot will be in a particular location. and we tell it how many subplots we want.

So, there are two columns and three rows of subplots and the entire figure is '12, 12' in size. Again, that's something you can control. So, it will fit all six subplots into three times two, that is, subplots into a 12 by 12 figure, right! And then, we go through our subplots by—we enumerate the agency, 'borough iteritems'. Let's take a look at what that is. That will make it pretty clear here. So, it's all above. It's going to essentially enumerate all of the stuff that we have here. And that's that, bum bum bum. Okay, let me do it like this. For 'i', label, call in enumerate agency borough iteritems is an iterator. It's not going to have values. Enumerate this. Print 'i', label...label, call. So, you see what happens here, right! So, we get 'i' is a '0' and then we get the columns for that are the label is Bronx and the agencies are all of these. These are the calls, right!

So, we're getting '0', 'Bronx', '3', '1', '1'. '0', 'Bronx', 'DCA'. '0', 'Bronx', 'DEP', etc. Okay, so we get this whole setup over here. So, that's what this is producing for us, the enumerate. So, this gives us our ways by the 'i', 'label' and column. So, we set up the axes and what this tells us is, where for each 'i', where the plot is going to go. So, that's what this one tells us here, right! and this locates it, right! This is a division of the two and the remainder divided by two. And then, we sort the values of the, within this here. So, we get these as the values that we are sorting, right! The agency counts, so we sort those values and we pick the first five. Okay, that's what this is saying here. Let me slice it, right! So, we pick the first five. And then, we put inside a bar chart and give it the right access location and set the title for the access.

For each one, it will have the title of the label, which is 'Bronx' or 'Brooklyn' or 'Manhattan' or whatever, and which was this over here, right! This is the label, 'Staten Island'. Right, so that's a label. Remember, this was, I was getting 'i' That's 'i', that's label and that's col, okay! So, in our thing here, 'i, label, col'. So, we get that and then set the label to that. And then, just, you know, that's it. It'll...it'll automatically display it and it comes up here and we get this stuff here, we get this. So, we can look at this thing and try to see whether there are some differences across Boroughs. And, we see that here in 'Bronx', in 'Brooklyn' and 'Manhattan' and 'Queens', the 'NYPD' has the most number of complaints but in Bronx, the 'NYPD' is actually lower than the housing department complaints. 'HPD' is high in 'Brooklyn', 'Manhattan'—'Brooklyn', the 'Bronx', 'Manhattan' and 'Queens', but in 'Staten Island' where there are very few public housing, New York city houses, it's relatively low, right! So, that's what we can look at the Boroughs and figure out which agencies are featured higher in terms of number of complaints and which are fewer.

Video 12 (5:14): Data visualization: Part 4

The last thing we want to do–look at is the processing time, because processing time is probably one, you know, an important statistic for these guys to figure out how well they're doing with handling complaints. So, we can–what we can do is, we could group them by 'Borough'. You know, so that's what we're doing here. We get–take our data, and extract the 'processing time' column and the 'Borough' column, and then group it by 'Borough'. Then, we can choose a 'describe' to take a look at it, and it tells us that by 'Borough', 'Bronx' had a mean of '5 days', 11 hours, 'Brooklyn' '5 days' one hour, 'Manhattan' '5 days' seven hours, 'Queens' '4 days' 22 hours, and 'Staten Island' '5 days' zero hours. So, 'Queens'

apparently has the fastest processing time, right, in all...in all this setup here. But in general, the–one of the limitations in Pandas is that when you try to use datetime or timedelta objects for computing means as, standard deviations, it doesn't work very well. So, a better way of dealing with time in...in this maybe is to convert it into floating point numbers. So, what we'll do is, we'll convert these '5 days', '12 days', etc., these ones, into floating point numbers, which would be like 12 point something, five point something, that kind of thing. And that turns out to be fairly straightforward again, so what we do is, we'll apply this function, which is a NumPy function, called timedelta, and we're going to convert this into units of one day that– day units.

The 'D' here indicates that we want it to be days. So, we're going to take our 'x's'. 'x's' in this case are timedelta, like '4 days', 15 hours, 24 minutes, 28 seconds, and divide it by this conversion factor, which is going to convert into days. We could have hours. We could have minutes. We could have seconds. But, we want to use days here because that's easier to use. And, we apply this function to every element in 'processing time', and we'll put it into a 'float_time' function–float time column. We could take processing time and get rid of it, but sometimes it's useful to keep that as well. And these are decisions you make based on how much data you have, and you know, whether you're going to end up using lots of memory or not. So, in our case, this is not too much, right! So, we could just use that. So, we create a new column called 'float_time', which is really just the floating point version of time. And then, take a look at that data. And once we do that, it becomes relatively easy to compute statistics. So, here we have '15 days', zero hours, becomes 15.02, '8 days' 21 hours becomes 8.91, and your n–NumPy has nicely done that for us.

So, we have a nice float point stuff there. And now, we can easily compute statistics on this. So, we could say, do the same thing, take data and extract the float time and the agency, and then group by agency, and sort the values, and we get these here. It tells us that the maximum time that it takes to close is, for 'EDC'–I'm not sure what that is, Fire department, 32 hours. And the fastest are 'NYPD' is actually just 0.20, right!

It's a, you know, maybe in an hour or so, everything gets done, 0.20 of a day. And then, we could...we could compute statistics on this. We could do means, standard deviations, whatever we feel like, or we could just do–bin the stuff here, and we get this times binned by what...what...what we are doing is, you are computing a–drawing a histogram that has 50 bins that has taken the time and converted it into 50 bins by values, right! So, these are by values. So, you take the time, which varies here from zero days to 148 days, and divide the zero to 148 into 50 equal lengths, and then count the number of incidents in each bucket, right! So, you've got 50 buckets, and each bucket you call another incidence. And, we find that you have it's–this is a kind of a nice graph we do because almost everything is getting done in the first bucket itself, right! So, this is–if you look at this, the...the proportion of elements that are getting cleared up in the first bucket, is like much higher than anything else.

So, this is a sort of nice one. We can try a little bit more final thing. I'm not sure if this is going to work, but let's try that 500 buckets, so we find a little bit more variation here, in the initial thing. So, it...it means that we don't really have many cases that linger inside this thing here. And, as a final note over here, there are other visualization libraries that you can use. We're not going to cover them in this class, but there's Seaborn, there's Bokeh, and there's Plotly. Seaborn is a really nice library for visual–visually presenting your analysis results. Bokeh has interactive graphs, which is kind of nice. And it's–interactive

it's means that you can– like if you have this graph here, you can actually move it around, zoom into it, do all kinds of nice stuff with it.

And, Plotly is a very nice library for drawing maps and all kinds of things. The only thing is, Plotly, to get the upper level, you have to pay for it. Bokeh and Seaborn are...are free open source, free...free packages, essentially. So, that ends our brief introduction into data visualization and data cleaning.