Week 12

Video Transcripts

Video 1 (8:05): Introduction to NumPy

In today's session, we're going to look at the basics of data analysis with Pandas and NumPy. Pandas and NumPy are two packages–modules that are available with Python and are very useful for data analysis. Python has an huge number of libraries for supporting data analysis. The main ones that we look at here are–the ones we are going to look at here are the main ones, but that doesn't mean that we're restricted to that, so it's a good idea to sort of, whenever you are doing some kind of analysis to...to use Google effectively to see what else is available, and whether there's something that'll fit your own particular need. So...so for–but the basics are that almost every library is going to use NumPy as its underlying, data structure so to speak, but what NumPy does is it supports numerical, and array operations, and it's very fast, it's very fast way of doing numerical, and array operations, so that's really– more stuff is built on that.

This Scipy, which is–does some scientific applications, and we're not going to worry too much about it, but it has—you know—a lot of capabilities for advanced, like, differential equations, and things like that. This Pandas, which supports data manipulation analysis, and Pandas is going to be our basic data structure, or Pandas will provide our basic data structures for most of what we will do in machine learning and analysis down the road. And then there are various visualization libraries, there are many of them, and they all have their own strengths, and weaknesses. So, it's always a good idea to look, and see what suits your particular application the best. So, matplotlib is the basic library, which sits inside Pandas, and Pandas are direct support for that. So, that's something we will do anyway. There's seaborne, there bokeh, and–which are two third-party libraries that are pretty good, and they have again–have their own strengths, and weaknesses, but you– can–we'll take a look at some of them along the way. This plotly, which is a really nice library for...for drawing graphs, and those kind of things. There's gmplot, which is a great library for putting maps very quickly, integrating your data with a Google Map. So, you know, that's Google Maps– everyone understands those, so it's always a good idea to think, if you have a mapping kind of application to see how you can put it up in a map.

So, the...the...the key here is that when we're talking about visualizing data, we can visualize data in many different ways. You can put a graph, a bar chart, a pie chart, or you can throw in a map, or you can throw in interactive map, that way you can sort of go in and out, and drill down, and get more information, no. There're many different ways of doing stuff, and each library has approaches the—this process in its own...in its own way. That means that it–like, some...some of them like bokeh is very good at interactive mapping. Gmplot is good at handling Google Maps. Plotly is good for almost any kind of thing, but also it does map–maps, you know, like if you want to have zip code based maps, you use shape files, Plotly integrates well with that.

There's another library called basemap that does that integration too, but probably a lot slower, but very nicely as well. So, it really depends on what you want to do. So, we'll take a look at a few examples in **our—as** the rest of this course, but keep in mind that there is no perfect library out there. It's not that you can take one library and use that for all your visualization needs. It's really depends on what you want to do. So, keep that in mind and be ready to Google. So, that's our introduction to the...the libraries. Why do we want to use NumPy? Now, NumPy is...is a good NumPy or NumPy, people use either term interchangeably—is...is...is very useful, because it's a fast mechanism for dealing with arrays, and matrices, and those kind of things. It's...it's fast, and it's also more space efficient than lists. Now, the key understanding here is that–the key reason for this of course is that a list is—can have different objects inside it.

So, you can have a list that mixes integers, strings, or even your own user-defined objects inside the same list. So, what that means is, that if I have a list, and I'm looking at this list over here, and if I say, if I have one, and then 'a', and then two, and then 'b', and I'm looking for the 'b', and I know I'm looking for, let's say, 'x' equals this. And looking for 'x3'. So, 'x3' is 'b'. But I can't get to this directly by saying that, "Hey, it's the element–the fourth element in the list," because I don't know how much space these other elements–the elements before it, these ones, are using, right?

I don't know, because they could be of any type at all. What NumPy does is it says, "Hey, you know, you can't mix this stuff up." So, NumPy array will contain one, two, three, four, and now if 'x' equals this, if I do 'x4'—'x3', I'm sorry, then it says "Hey, it's...it's element in the fourth place, they're integers." If an integer is in say, 32 bits, or 4 bytes or whatever, then I know that it is at byte number three times four, starting from there, right? So, yeah that's right, three times four, zero times four, blah, blah, blah. Right! three times four. So, it knows it's there, and can go directly to that and access it. So, that...that, you know, in a very intuitive sense tells us that it's going to be much faster than using a list.

The second thing that NumPy does is, it's...it's also written in underline–written in C. So, C, as we know, as much, is a very fast language. So we don't have–we don't really want to know C, because it's also very complicated language. So, NumPy written in C, so it's already faster, even its implementation is faster than what Python is, of course, many Python interpreters will actually be using C underlying it, but that's in principle, yes. It has good support for linear algebra, Fourier transforms, random numbers, all kinds of things. So, it's very useful for any mathematical or calculations that you might want to do, and with data analysis typically you do lots of mathematical calculations. The basic structure in NumPy is the idea of an array, so, with an array, we already talked about this, an array is sequential collection, just like a list.

The only difference is that it has like objects. They have to be the same data type. The same type. So, this makes indexed access faster, because we can figure out exactly where it is, and it's also more memory efficient, because you don't have to worry about the overhead of keeping different objects together. So, lists are designed for things like making fast insertions, you know, ideally the list is that it'll change very often. So since, it's going to change very often, you want to be able to do things like insert into stuff. So now list–roughly, in the list structure, you know, you have–a list contains, like, memory addresses of each element. You know, something like that, right? Some such thing. So, it's a...it's a—has a big overhead in just maintaining just itself. NumPy arrays don't have the same overhead, and so they are a little bit more space efficient. But I don't think that's the main reason. The main reason is the faster access. They're mutable, just like in the case of lists, you

can change an array. They are optimized for matrix operations, so–because, that's you know, all the functions are written in– with this idea that you have this like matrix or list or matrix array or whatever and is optimized for that. And it provides random number support. So, that's the reason usually want to use NumPy.

Video 2 (8:25): NumPy Part 1

So let's take a look at NumPy in our—in here—NumPy here and see what we can do with it. So this is going to be fairly hands-on today. Let's take a look at it. So the first thing we want to do is, we want to be able to create a NumPy array, and to create a NumPy array, what you do is you–well, of course, it's Python, so the first thing you always want to do is import NumPy. And by convention, NumPy is inverted–is imported with nickname or alias 'np'. So whenever–and it's useful to know this because whenever...whenever you're Googling stuff, you're going to find everything will say npdot, npdot, and that usually means it's a NumPy array.
So, we have a NumPy array here and we want to create one. So, here I'm going to give it a NumPy array and I'm going to give it this list as an input, and I want to convert it into an array.
So, I use the function array over here, and that creates a NumPy array over there, and we can see that the—why does it say 'x', should be 'ax', let me clean this out actually, 'ax'...'ax', and I run this and I see that we have here array–the NumPy array looks like a list–one, two, three, four, five–inside this location, and it has five elements.
So–because we are dealing with Python, it's going to use similar–the nice thing with Python is that it uses the same functions across the board for the same action, right?
That's the polymorphic component in Python. So, that's what we get here, and it's–we can see what type it is if we like. So I could say here, type 'ax', and it tells me it's a type NumPy dot 'nd' array, right! That's the array–basic array type inside NumPy. You can specify the type when you're–and this is very useful, for example, let's say you're reading data from a file and the data is all numbers, maybe pricing data on–for a stock or something like that. You want it to be read into the array and converted into a...a floating-point number. So, you want–any...any extreme–text stream that comes in– because typically when you get data from the internet or from a file, it's going to come in as a text stream, you want it to be converted into a number and you don't want to have to do this in two steps, and do an explicit conversion yourself because it's going to waste space, and...and time. So what you do is you can say, "hey, I'm reading the data in, and so I read it and I'm–in this example, of course I'm just reading it from a list, but this could be coming from a file or some other source, and I want it to be read in as an integer." So, it's going to convert everything into an integer and become a numerical thing.
I can do a floating point, I can do 'str' so, I can specify exactly what type of data the array has. So, if I run this, for example, it says the first one, 'xi', is integers, 'xf' is floating point numbers and it shows that as one dot, two dot, three dot because in this case there is no integer value, no...no values after the decimal point, and the third is string then we can see there are string. So, just to put a fine point to it.
Let's say, if I did this 2 point 2 and ran this, I'm getting an error because I'm trying to convert 2 point 2 into a integer and that doesn't work, right! So, that's—the...the thing you need to watch out for is that the–your conversion actually works, right! So, it's an important point over there. So, what we would typically do is if you're reading a stream as you would put it inside, a try accept and if there is missing,

noncovertantvalue, you convert it to a 'nan' and that will stay as a– an invalid value over there, all right! So, that's the step one.

So, what can we do with it? Well, you can do a lot of nice little things with it. So, if we have here, an array that I'm reading in this thing I can call these functions sum, mean, standard deviation directly on the array and I get the...the values from that. So, for any numerical array, you can call these functions and get the arrays. So, NumPy has built-in statistical support, you can then just use on it. So, I get some mean, and standard deviation of these things very...very conveniently. Of course, in many cases we want multidimensional arrays like matrices, and NumPy allows that too. So here, in this case, I have a list of lists, and each sublist is the– is a row in my matrix. A matrix has a bunch of columns and a bunch of rows, so, each– so this is a column. The entire three columns and each row has six elements. So, I run this thing here and I print 'ax' and I get six elements in the rows and—three– what am I saying— three rows and six columns, yeah, sorry. So, this is one row, second row, third row. So I have three rows and six columns in my data set here.

So that becomes my matrix over here. I can index it because obviously you want to access stuff over there. Indexing is very straightforward. We can index stuff so that we can go straight to our values. If I want to–for–the indexing is essentially you do– inside square brackets, you do– it's not chained indexing but you can do it 'ax1' comma three, so it will say go to row number one, which is this, column three, which is this, so we should get 13, and we get 13.0. So, that's pretty straightforward. Slicing is a little bit more complicated because, when you are slicing an array, a matrix, so, you might want to slice a matrix so that you pull out, you know, maybe this 12, 13, 22, 12, 13, and 22, 23, right? So, to do this, we–our slice needs to know a couple of things. It needs to know what rows slices we are taking, and what column slices we are taking. And of course they have to be meaningful, right?

So, for example, so here–what I– so to slice it, you specify the row slices, and the column slices independently–oops, sorry. And the column slices independently, right! So these are the– what we are saying is, slice it so that we get rows one through three and slice it so that we get columns two through four, okay! So if I run this, I get rows one, through three not inclusive–remember, the slicing only stops between row one and two. So I'm getting 10, 11, 12, 13, 14, 15; 20, 21, 22, 23, 24, 25. And the columns, I'm saying, "I want are two-four, so that's zero, one, two." So, that's 12, 13, 22, 23, right! So, I get this little subset over there. That's the idea there. So, of course, you have to be little bit careful because if I did– instead of this, if I did something like, let's say five colon, ah, that probably works, actually.

So, it's going to get that anyway. So...so, the idea here is you can use colons exactly like you do– the only thing is that you want to watch out for is that, you're going to be doing this the rows first, and the columns next. Okay! Given an array, you can reshape it. So the shape attribute of an array tells you what the shape is. So if I go here and do this, it tells me that my array has three rows, and six columns, okay! Which we know is what it has, so that's the shape of it. I can reshape it so that I get nine rows and two columns, and what it's going to do is, it's going to just go serially, so nine rows means that zero, one becomes row one; two, three becomes row two; four, five becomes row three; etcetera...etcetera, right! And I get a new set of data. So go serially– column, along the rows, and allocates the data accordingly.

Video 3 (9:57): NumPy Part 2

If I try to reshape it so that we get to 10, 3, then I have a problem, because the total size of the new array must be unchanged. So, here I have 6 times 3, that means 18 different elements. So, my shape must—must be...must be factorized into–to 18 or whatever, must be some factors of 18, right! So, those multiply—if...if you multiply the rows by the columns, you should get 18 basically. So, 10 times 3 is obviously 30, so I don't have 30 elements, so, that's not going to work, or I could 6 times 3 or 3 times 6, so, this would work, or 3 times 6, which it already is. But I can't do anything other than something that becomes 18, right! So...so, I have to have the number of elements must be the same. So, you can reshape it to anything, as long as the number of elements is the same, NumPy has a whole bunch of things for initializing arrays, So for example, if I want to create an empty array of ten elements, I do this. So, what this is saying is take all the elements from 0 through 9 and create an array out of that. So I get an 'np' array that is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, little bit like the range that we have in Python, right! So it's just like the range here in Python, we are saying an 'a' range. So, you're creating an array of that, and you can do–you can create a matrix as well. So, this creates a matrix. The thing for each row you specify that separately, so that's row one, and that's row two, and that becomes our new range over there. So, I could actually do here 2 comma, 10 and I get starting with 2, right! So it's just like the range that we have in–I could do 2, comma 10, comma 2, and that will work too. So just like the range that we have in Python. This is creating 'a' range over there.

You can create an array of one's, and that's just going to create an array of 1, 1, 1, 1, 1, which is very useful when you're creating for example, tick marks in a...in a—in...in a...in a matrix or something, and sorry in a graph, right! You want to create tick points. You can just create it by 1, 1, 1, 1, 1. And you get these tick points over there. You can do things like create the stuff but raise the entire array to the power of 2. So, here what this is going to do is step one, creates NumPy array from 0 through 9, and that each element in that array. So, this is doing it says take every element in the array and raise that to the power of 2, and we get an array that looks like this. So, this is really two steps, right! So, this is 'np' array, and then, when you do–take an array, and you apply a function I would–operator it like star, star, or plus or star or whatever it gets applied to every element in that array, and we get a new array that is the powers of two.

And finally, you can create an identity matrix which is useful in matrix multiplications and matrix operations, and identity matrix is that's going to create a 10 by 10 matrix here, with ones in the diagonal elements, and zeros everywhere else. You can do matrix multiplication. Matrix multiplication is again straightforward here. You say, create an array and you got a nice array over there, and you multiply it by—just do scalar multiplication by times 2 just like we did over there. So, every element gets multiplied by 2. Or you can take two different arrays and multiply it–get the dot product by multiplying them. You just have to make sure that the matrix shapes are correct for the multiplications. So, here we have 'ay' which is of type—let me just write that out here, just to be clear, the shape of 'ay' is two rows and ten columns. So, to multiply this we need to–the matrix that we get has to have ten rows, right! Essentially. So what we—what we—what we do here is I'm just reshaping it as 10, 2, and multiplying them together. But you don't have to, of course! You, typically you're not going to do that. You're going to have two matrices that you are independent that you want to multiply. But in this case, I'm just taking the same

thing and multiplying it, and we get a dot product, which tells us there's a two by two matrix, right? Exactly that. You can take a look at how well NumPy arrays work. How fast they are compared to lists. So for example, let's say I take an array of 'ax' over here, let me print that, take a look at it, so, 'ax' is a 2 by 10 matrix. And I'm going to take a transpose of that, and I can do a dot product of that, and I want to functionalize that. So...so here I have this over here. So what I do here is I'm creating a function that takes, so in this case, I have 'n' as ten which is what I did originally there. I want to create a function that creates an 'n', a 2 by 'n' matrix, right! Right! A 2 by 'n' matrix, and the first row in that matrix is powers of 2, and the second row is powers of 3, and then I create another 'ay' which is a transpose of that so, I can do the multiplication, and I'm going to time, the amount of time it takes to actually do the multiplication, right! So I import datetime, and from datetime, I want to get the...the now...the time now, do the multiplication and then get the time at the end of the multiplication and look at the difference, right! So if I, and...and return the, the time that it's taken to do the multiplication which is essentially this minus start–'n' minus start, right!

So I run this and I get, this tells me took 18 seconds to do this, 18 nanoseconds, milliseconds...milliseconds, I guess, right? So I can write...write a similar thing with Python lists. So in Python lists, I'm going to create a list called 'x' which has powers of two, for 'n' range, and list called 'y', and then create a...a...a matrix of that 'xy'. So this is similar to our, 'ax' is similar to the 'ax' over here, right! It's a list, and then, I'm going to transpose that. So, this is what I'm doing the transposition of it, bam...bam...bam, and I do the transposition of that thing, and now I'm going to actually multiply them. So for in the multiplication itself, if you...if you—the way matrix multiplication does happens is that you take each element in the row, and multiply it by the corresponding element so, you take say, row one. So, zero, zero in matrix 'x' is going to get multiplied by–it's going–the element in zero, zero is going to be multiplied by the element in zero, zero in 'y', and then element in zero one is going to get multiplied by one, zero in– zero, one in 'x' is going to get multiplied by one, zero in 'y', etcetera, right!

So, that's how you do simple matrix multiplication stuff here. So that's what this list comprehension thing is doing over here, and you can sort of analyze this yourself. I'm not going to go into this in detail. But you can analyze this for yourself over there as you go along with it. But this essentially does the multiplication, and our goal here is to see this is probably the best way to do it, with Python lists. I'm not sure, but I would think so. So our goal is to see how long that takes. So again, we have a start time here, and an end time here, and the difference is going to tell us how fast that works. So, if I looked at say ten over here, and ran this, it took–that took 20, and this took 22, right! So, what I want to do is, I want to see–compare the results of my ordering the multiplication with the twos. So I'm going to stake for 'n' in this list, for 10, 100, 1000, and 10,000, and compute the dot product using NumPy, and compute the dot product using lists, and then look at the comparison of how quickly they work. So looking at this here, we see that for NumPy, it's 14, 9, yeah, here. So we get 16, 8, 81, 56. And for the list, it's 16, a 100, 670,

36-216.

So, as we see that as the number increases, NumPy actually scales very well, right! It's...it's...it's–it scales really well, because 16, 8, 81, 56 actually 10,000 was even faster than a 1000, it's just, depends on what else is running in the program, right! So, but it's scaling really well, right! Whereas the list operation is... is actually increasing almost pretty much exponentially. It starts with 16, with a 100 it's a 100. With 1000 it's 670, and with 10,000 it's

36216, and we can run this again and take a look. What do we get here? 15,12,32,78,16,72,694,15,423. I mean, ideally you want to run this hundreds of times but you can see that it's, NumPy's far by far the faster of the two.

And even more important, it scales almost linearly with this....with this—if...if even better than linearly actually, right! So it's...it's a very fast way of calculating as compared to lists. So even if we could get a better way of doing the multiplication of lists than this, I don't really know, this is what I could figure out, you'd still have a problem with lists, okay! So, that's the main reason to use NumPy. It's much faster than anything else.

Video 4 (04:13): NumPy Part 3

The...the next thing in...in 'NumPy' is that there's a nice function called 'where' which works somewhat like an excel if you know...if you know an excel if, what an excel if says is: if, a condition is true, then you put a value in a...in a cell and if it's false, you put a different value in a cell. So, in excel if you will say, if...if say 'C1' greater than 20, then 'C' in—let's say, this was in...in the cell 'C2', you would say, if 'C1' greater than 20, you put a zero in 'C1', otherwise, you put a one in C—sorry, you put a zero in 'C two', otherwise you put a one in 'C two'. So, that's the idea in excel. So similarly, there's a 'where' function in 'NumPy'. What this says is go through the array 'ax', and for every 'ax' percent '2' equals equals '0', which in this sense means that the remainder of dividing 'ax' by two is zero, which means that the value inside that cell is an even number, you put a one, otherwise you put a zero. So, we have this as our initial array '0', '1', '2', '3', '4', '5'. So, what you want to do is, wherever the values are even, we put a '1'. And wherever they are odd, we put a '0', okay! And, that's the goal here. So, it's pretty much like that. So, you can think of it getting exactly that. So this is—yeah so, '0' is even, we get a '1', '1' is odd, we get a '0', '2' is even, we get a '1', etc...etc. So, we'll—the 'where' function is kind of important, we use—we'll use it a lot in Pandas, so, just be aware of that. But it's a...it's a very straightforward function, there's a condition, and we're looking at each element of the array, and we're creating a new array, a new 'NumPy' array, which have—which contains this value if this condition is true or—in...in the cell, and this value if it's false. So, the new array is going to have the same size, the same dimensions as the original array, and except that the values will be, whatever is here rather than the original values and the way we select them or choose what values to put in depends on this condition.

There is a—okay, we can—if...if you have an array that contains a bunch of nans, then you can use a function called nanmean 'x' scipy. Now, remember, we talked about scipy, it's scientific computing, and what nanmean does is, it ignores anything that is nan inside it, so, this actually is probably not a great idea here. 'NumPy' also has very good random number support, and there's a library called—module— called random inside it. So for example, if you want to get a disnormal a bunch of normally distributed values, we can call np dot random dot normal and give it a size ten, and we get back an array that contains a normally distributed values between negative three and positive—well, with a zero mean and one standard deviation, and so, standard normal form. We can also do—create a matrix in this stuff. So, if I go here and run this, I get a 100 by 100 matrix, so it's kind of very useful. You know, when you're doing certain simulations or things like that, you can create the matrix, and then run your simulation on that. You can give it any distribution that you like. So, if you want, an exponential random variable

drawn from an exponential distribution, this is going to pull that out, and each time I run this, I'm going to get a different one. So, you know, it's–giving me–you really want a series where they are exponentially distributed. So, if I run this here, I get these as my exponentially distributed random values, and finally, the thing that we often uses is getting a random integer from—drawn from a uniform distribution, and this is going to get from a uniform distribution from negative 10 to positive 10, and the size, it's going to give me nine times nine, or 81, different values for this thing here. So, that's the basics of 'NumPy'. It's a package–has many other things in it, but we are—we will stick with this stuff here.

Video 5(8.53): Introduction to pandas

Pandas is the basic package for data analysis in Python. It provides integrated data manipulation, and analysis. So, key here is that you can manipulate data, and some, you know, basic analytical capabilities, and act power really comes from the fact that it feeds into various machine learning libraries that you can then take the basic panda data structures, and feed them into all kinds of different packages that you can use for more powerful analysis. It's well-integrated with data visualization libraries. Pandas comes by default, you can use any matplotlib function in it. So, you get—the matplotlib libraries are already in...in-built into it, so a very quick visualization of graphs, etc., Pandas is really good. If you want to more complicated stuff, then you can use the other liberties we mentioned, but for the basic stuff it's really...really good. It also has built in time-series capabilities, which is very useful, we'll see that there. It's optimized for speed. It's got NumPy sitting underneath it, so, that's always a good thing, and it...it's—Pandas open-source so it comes with a bunch of libraries that are useful for getting data.
You can get it from csv files, from excel files, from HTML pages, grab all the tables, Google, from the World Bank, from the Federal Reserve. We could until recently get data from Yahoo, but it's an open- source library, so, Yahoo made a change to their underlying data structures, and HTML pages, and so currently this is not working, and that's the...you know the pitfalls of working in the open-source world, but it will—hopefully, it will get fixed, and someone will make it work sooner rather than later. There are workarounds around it. You can Google them but we will not focus on that right now. The main thing in Panda is that, it organizes data into two data objects, and that's the data structure at that you—that it works with. The...the essential idea is something called as a data frame, which is a two-dimensional table object in which you can recognize the columns, and can recognize the rows, and you can address columns, and rows by name rather than having to go through like zero, one, two, as we do in lists, and arrays, and NumPy arrays, and all that.
And each—so that's two-dimensional thing, and each...each column or each row is also a series...a series of one-dimensional array objects, which again has, you know, meta information built into it that's kind of useful. So this is the key thing with Pandas that we are interested in is how to work with series, and data frames because that's when we go to machine learning, and all that kind of stuff, we're going to be using Pandas data frames as our basic data structure. Each column corresponds to a named data series so you know, column has a meaning, and you can index every row, right! So every row also has a meaning. So you can think of time series, for example, you might have open, high, low, close, etc. prices, and then you have a date stamp, so the date stamp is a row index, and open, high, low, close become a

column name, and that becomes your data frame that you can use for analysis. So, that's how Pandas works. So, let's take a look at Pandas in practice and go to our Pandas notebook over here.

So, before we do anything else, let's make sure we have everything installed. So, the first thing we are— we will work with is that we need to make sure that we have these two libraries. Pandas comes installed in Anaconda's Python by default, so, we don't worry about Pandas. However, for reading data, there's a nice library called Pandas_datareader that we need to do—we need to download. And Pandas_datareader, for a HTML part, uses 'HTML5lib', but it uses this particular version of 'HTML5lib'.

So, the first thing you want to do is you want to download these two things, and make sure that your 'HTML5lib' that you working with is 1.0b8, okay. So, let's run these two. So, we have both these installed, and just to be on the safe side, after you install stuff, you know, it's not a bad idea to restart the kernel. So, if you restart the kernel in here, it will clear a lot of output as well, so that we don't have anything, if you restart the kernel, then you are sure that, you know, any changes—because when you install one library, it might be loading other modules and installing those as well or updating them or changing them, and what you want to make sure is that you're not in your current notebook—which is not going to the case in our case right now, but because we are starting with this, but if you install it later, for example, then you want to make sure that the current notebook has the latest library loaded inside it.

If it already loaded the library previously, it's not reload the new version just because you've got a new version in there. So, that's the goal there, right! So, once we've got the stuff here, we want to work with the inputs. I put all of the inputs up front here, but of course we can do them anywhere we feel like before we use them, But just to make sure we have all of them so, we...we of course need 'Pandas', so, that's our base library. We need 'Pandas_datareader'. This is for reading data from Google, getting HTML from tables that kind of stuff. We want to import 'NumPy' as 'np', right. And Pandas typically load with the alias 'pd'.

Again, this is just a convention, it's not a requirement. We are going to import 'matplotlib dot pyplot' and call it 'plt', just that's our alias for that. And we want to important 'datetime' and—as 'dt'. Well, we don't have to, we can—conventionally we don't really need to do that, but...shall I do that. What we'll do is we get rid of this because typically you don't need that, right! So, I know I have that there, and the only other thing that will we want to make sure is we use this command here called 'matplotlib inline'. The 'percentage' sign is a Geppetto command. It's telling the browser something that is meta function on the browser itself. So, Geppetto has a whole bunch of things you can do on the page. We have not really looked at that at all. We are focusing only on Python. But there's a lot of stuff you can do—that as well. So, one of the things we do is we want to tell it that any graphs that we draw using 'matplotlib'— and 'matplotlib' is part of Panda, so, Pandas can automatically access a matplotlib functions. But any graph we want to draw using matplotlib should show up on our browser itself. If you don't do this, then it's going to pop up a window somewhere behind and you won't know that it's there, and your browser will be waiting for you to complete that, it's kind of a mess. So, when you're working on Anaconda, it's a good idea to add this. But not if you are writing your code somewhere else like in NID, like perhaps Pycharm or something like that, then you don't need it. But if you are—or Eclipse, but if you're working in the Anaconda Geppetto notebook, then in the Geppetto notebook you should always do this because you want the graphs to show up right there in...in the output part of your thing.

So, this is the—these are the inputs that we want so, let's run this, and we've got it more imported, and let's start by looking at a data frame. So here, the data frame, like I said earlier, is a bunch of columns and a bunch of rows. And what we want to do is we want to give our columns names. So what we are saying is, take this data, which is these two rows here—'1', '2', '3' and '1', '2', '3' and give—there two rows and three columns and for each column we are going to give a name—give a...give a we're going to give a name to each column. So, column '0' has the name 'A', column '1' has the name 'B', column '2' has the name 'C', and we can look at this, and we get a data stream that looks like this. So here, what do we see? We see that Pandas has given column names, right, and its given row index—indices. So, since we didn't specify an index explicitly, it's made up an index of row numbers—'0', '1' etc., right! And that becomes our index over there, but this is the basic structure of a Pandas data frame. The idea is that the data is sitting in here, in this '1', '2', '3'; '1', '2', '3' stuff, and we've got a bunch of row indices, and a bunch of column indices that we can then use to access data, just like an excel spreadsheet. In excel spreadsheet, you have 'A', 'B', 'C' in the columns, and '0', '1', '2', '3' in the rows. So, you can think of Pandas as a...as a slightly more knowledgeable version of Excel, okay, and...and at least in terms of data structure.

Video 6 (9:45): Pandas

So, once you've got that, we have a data frame. We can start accessing rows and columns. So...so, let's just walk through some of these simple Pandas, things that we can do with it. So, we have—I'm going to create a data frame here with one, two, three rows, and each row has four elements. In 'r1', '00', '01', '02'. You can see these are just the locational spaces, '10', '11', '12', and '20', '21', '22' and 'r0', sorry 'r1', 'r2', and 'r3' are the row numbers that we're giving it. So, and we give the columns our row labels. So, 'r1', 'r2', and 'r3' correspond to the column row labels, and the rest are 'A', 'B', 'C', just like before. So, let's run this and print it–out the array. So, we do this here–we print–look at the way the data frame is. So, the data frame is sitting in this particular location, and now what I want to do is, I want to set the index of this data frame as 'r1', 'r2', and 'r3'. So, I tell it here–I call the function on data frame, I call the function set index, and I tell it that the index is corresponds to the column that is named 'row_label'. So, this 'row_label' came from here. Okay! And I am saying 'inplace equal to True', which means don't create a new data frame. Change this data frame so that it has the same—it's the same data frame but now has been reformatted. So, that the row number—rows have been taken out–row labels that have been taken out of the data frame, and put into the index instead.
So that's what this becomes like. Just to be clear on this, What if I did print 'df' here? Did you notice what I'm getting here is the row label is another column in it? Right! And 'A', 'B', 'C' is over there, right! So, it's–yeah–okay–so, these are our columns, but rather than getting—you know what, let me just try this separately, much clearer if I take this out–control C–So, here it is. So, we can see that row label is just another column. So, when I make it under index, it becomes an index that is no longer a column. It's no longer–we cannot longer refer to it by its column name, okay! So, it becomes row label just becomes an index for that table itself. So, that's really the difference there, but it's the same table. If I say 'inplace equal to False', then it's going to create a new data frame. My original data frame will not be—for

example, if I did this... if I did this, then notice my data frame is unchanged, right! My data frame is unchanged because the—I've created a new data frame. I could call this 'x' equal to 'this, and do 'print(x)', and see 'x' has 'row_label' as a separate index but 'df' is unchanged.

So, the 'inplace equal to True' is kind of helpful. If you don't want to change the create a new data frame, which is probably advisable. I mean, if you have a data frame that contains a three or four gigabytes of data, you don't want to create another data frame that contains three or four gigabytes of data, right! Or maybe you know 100 gigabytes of data. It's going to just use up memory for no reason at all. So, in place may be not a bad idea, right! So, given that we have this data frame here now, right!

Bum...bum...bum. Let me get back to this, 'inplace equal to True'. Run that. So, this is the data frame we are working with right now. So, the data frame now has three columns 'A', 'B', 'C' and rows that are indexed by 'r1', 'r2', and 'r3'. So, let's see how we can get data from this. Well, the nice thing is that you can access a column directly by just using this dictionary method of accessing data because the...the way Pandas works is, the columns are just dictionary elements, right! With—that are indexed by the name of the column. So, I give the—for the... the 'df' in my data frame, 'B' is the column name. I said 'df B' and I get the data that corresponds with that column.

And note that it comes with the index. I have 'r1', 'r2', and 'r3'. This is now a data series, right! And if I do 'type' of this, it says 'pandas, core, series.Series'. So, it's a data series. And if I did type of 'df', it's a 'pandas, core, frame.Dataframe', okay! That's the idea here. So we go back to this and do 'df(B)'. To get row data, I use this—this attribute called location, and use the index because the rows are also indexed by—they're like dictionaries indexed by the row index value. So, I want to say 'df.loc', 'r1' is going to give me the row that corresponds to 'r1'. So, it says column 'A' has the value 0. Column 'B' is a column 1. Column 'C' has a value 2, and this is another data series object, right! A data—pandas data series. I can also get rows by row number which is kind of useful because sometimes it's faster to get by row number rather than by indexing.

If you can get time series data, for example, you know they're arranged by row numbers so, you can just use a number rather than having to use the date or whatever every time, right! So, we do this. Then give me for 'iloc 0' which is a use of function 'iloc', which is indexed by integer rather than this, and this is going to be the same thing as if I had, let's say, for one, for example, the same values that I have for 'B', right! So it's the—Oh sorry–the rows, not the columns. So, this is the same value as 'r2'–so–oops! Yeah, so this is the same values as here. I apologize for that. Got a little bit confused there. Because we're getting 'iloc', we're getting the rows here. So, this corresponds to this...this is the same result. You can get multiple columns in a...in a—from the table. So, all you need to do is you give a list of column headers, and you get multiple columns there. So, note that this is a list. So, it's not like 'df' 'B' and 'A' in a...in a...in separately but inside a list here. So, this gives me this, like this: if I try doing something like this, and this is you know common mistake, so watch out for it, 'df''B' we know works. But if I did 'df''B' comma 'A', then that's not going to work, right! Because there's no key that corresponds to that. So, this has to be in the form of a list, okay! So, just watch out. This is a very common mistake in working with that—working with pandas—data frames. You can go and get a specific cell from the data frame. You can—to do that you want to use the 'loc' function, and..and the 'loc' attribute rather. You tell it which index you want to use, and which column you want to get data from.

So you can do that. And this will give us the item in here, which is saying 'r2' is the index, so it gets this row, and 'B' is the column. You get that one. You can try what's called chained indexing, and if I do this, this gives me the same result. You make that 'B' has one one, but there's a slight difference in that and you will see briefly, really soon, we'll see what the difference in that thing is. You can slice your—you can...you can slice your data frame just like we did with NumPy array, and the goal here is the–the basic procedure is very similar to—because remember, it's under—underlying it there is NumPy anyway. So, you can slice by—the rows by giving the row colon row thing here. So, we use 'df.loc', and we say we want row from 'r1' to 'r2', so we get these two rows. And we get '00', '01'. We don't get row three. Or you can choose a row-column combination, and that would be in this case, you want 'r1' to 'r2', and 'B' and 'C'. So, we get that little submatrix there, '01', '02', '11', '12'. So, next we're going to look at—this is the basics of Pandas so, the idea here just to recap a little bit that what we are doing is, we create a data frame, and from the data frame, we have multiple ways of accessing data from it. You can access columns by just giving the column name and using the dictionary structure by saying 'df' 'B''A', for example, inside a list will give you a subset of columns so that data frame or—if you say 'df''B', that gives you the one single column. And you can use the loc function to either get a single row or...or as a slicer, to get multiple rows, like here, or you can use it to drill down into a single column—a single data point by giving both the row as well as the row index, as well as the column name, and you get a thing, and again, you can slice—take a slice also that contains a slice of rows or a slice of columns. So, that's the basic structure.

Video 7 (8:23): Pandas datareader

Now we're going to take a look at the various ways with reading data from Pandas, using Pandas. We can get data from HTML tables and any web page. There's a nice way of doing that. You can get data from Google Finance, from Federal Reserve, from the World Bank, all kinds of places. So, we'll take a look at some of these things here, csv files, excel, excel files. I'm not going to do csv, and excel here. We will definitely cover that in other classes as examples. But just so you know that we can, or we can do that, if you don't already know that. So any HT...any HTML page that contains tables, we can use Pandas to extract the tables from that page. So, what are the tables on an HTML page? The tables contain the tag, this tag here, if I do insert, so, anything that's inside. So table tags typically contain, you know, a bunch of rows so, I have 'tr' and maybe column one sorry, 'td', 'c1' slash 'td'. What the heck happened here? Slash 'tr', so I get this thing here, and you can see this comes as a table like this, right! So this is a basic table structure. So, what Pandas does is it provides us with a function called 'read_html' that given this function here, the 'read_html', that given the URL, can go to the URL, and decode all the tables that are inside a table tags like this. So, it looks for the table tags, and then looks for the 'tr's and 'td's to get the table structure are, and takes that stuff, and shoves it into a bundle data frame because data frames are tables anyway, right! So that's the idea here. So...so you call the function rea...'read_html'. So I can run this, and I get this back here. I get a—my table back inside of 'df' list variable. What I'm getting back is a list of tables because any HTML page can contain multiple tables. So right now it says, I have just one particular table in that thing here.

So that is just one there. And—but it's a list containing one element. But it's a list to sort of extract the actual table itself, I need to index it to the 0th element. There's only one element. And I can print that, and I get this dataframe. So we've gone to the Bloomberg Markets Currencies page and pulled out the single table that is there, that contains currencies, values, changes, net change, time, and a two-day change which is land for some reason but, not sure why. But that's in the data itself. But so this is what it's...it's pulling out for us. So we get this inside a dataframe now, right! And we can see it's a dataframe if I did type 'df'. It tells me it's a Pandas dataframe. So that's a very convenient way of grabbing data from...from the internet. Any page that—any page you're interested in, contains data in the form of a table, you just call read HTML, give it the URL, and it's going to pull out all the tables from there. And if you note here that the—this has actually already for us, it's...it's just to make it clear, the structure clearer rather than using print, I'm going to use 'df'. It's–the column names it's figured out for us that the column names are inside the–that–the header column there contains currency, value, change, etcetera. So we already have that in a very usable structure for our things out there. If we with the index, it hasn't done, an index— it will not do automatically. But you can specify the index. Say that like–if I want this to be my index. The ERU-USD, USD-JPY. I can set the index to this, and set in place equals to, just make that. Now I get a nice little table here that has the thing properly done for us, and we can now use our 'loc' to extract values from here.

So this tells us that the Euro to Swiss Franc exchange rate current–currently is one point zero eight nine nine blah, blah, blah, right! So that's the—can–I—extracts specific currency rates or changes or whatever if you like. So change, that means point 0 0 2 you know, all that stuff. Even the time if I'm interested in the time like, you know it's kind of useful we are doing that so, this is a way—nice way of getting data from there. Next thing we want to do is we want to make sure that we talked a little bit about chained indexing earlier. Where we said we could—in this section here where I said that you could either go drill down to a specific cell by either giving 'r2' comma 'B' in loc or doing chained indexing where you do 'df' 'loc' 'r2', so you get your series, sorry, a data frame which contains the row information then you extract the 'B' column of that series which is only one element.

So, this is called chained indexing. And this is, you know just complete normal indexing, or whatever you want to call it. So the...the results are slightly different, and you want to see what happens when you do these two things is that when you do 'chained indexing, or Pandas does is it creates a copy of the data rather than the actual values. So, when you drill down, and you get to this value here for example: I'm using-here I take 'df dot loc' I first drill down and get the row that corresponds to 'EUR-USD', and then I get the value of the column chained. So, this is now a copy of what actually in there—in our dataframe. So now, if I look at, and let's say I store this currently in 'EUR-USD', I store that value somewhere I go, and I do 'loc', and I change 'ERU-USD', 'change', this is, I'm saying, get me this value, right! And change it to '1.0', okay! And if I print these things here, I find that—so it's telling me it's trying to send a copy of a slice from a database. So, what it's doing is it's changed this 'df' dot 'loc', and made it–this return a copy, and that copy is an ephemeral copy.

Because I haven't stored it anywhere, right! So I'm not storing it anywhere, its ephemeral copy. I change the value of ephemeral copy to one point zero. And then when I look and access the same value again or go back to the same location again inside my original table. I find that it's unchanged. And the reason is because I use chained indexing, it actually did that on a copy. So instead, I don't want to use 'chained

indexing'. I want to use direct indexing, and direct indexing one will get a copy when it works with actual locations. So now, here I do 'df' dot 'loc'. 'EUR-USD change'—so change. This is no longer a chained index. We are not using, we are not getting one thing first we are saying go, and get me that location.
So, here I go in, and I chain that to one point zero. So what this does is it creates a view of the database—of the data frame. So now, it's sort of pointing to the actual location of the thing inside my data frame itself, and if I change this, then I find that inside my data frame, it has actually changed. The copy that I made and I stored inside another variable, 'ERU-USD', that hasn't changed, and because that's copy of when I do an equal to here, it's actually creating a copy, but the original location has changed now because I did not use the chained indexing method to get it, that's the thing here. That's the— so that's the key here.
So you'll be really careful when you're doing the data frame. When you're working the data frame, you want to make sure that if you want to work with a copy, you're working with a copy. If you want to work with original, you work with original. To work with original, use this mechanism. To work with a copy, use this mechanism, and you'll be fine with that. As long as you're clear what you are doing, you'll be all right, okay! Just make sure that you get that.

Video 8 (4:59): Google finance data

So, let's take a look at now another more detailed example. Let's say we want to go into 'Google Finance', and this should be finance, and get data from there, right! So, we use 'pandas_datareader' for this. So, 'pandas_datareader' we're going to import that, and we're going to call it data or whatever for whatever we want to call it, and we're going to use date time, 'import date time', and we'll change that to 'date time'. So, what I want to do here is to pandas_datareader—what it does is it allows us to get— give it—you can give it a ticker, and you can give it a source. The source is in this case is 'Google', it could be Yahoo, it could be Fred, it could be the World Bank, you know, various alternatives available but we're going to get from Google, and we're going to get data for IBM, and we give it start time and an end time, and the start time and an end time have to be in date time format, okay! For that to be in the formats specified by the date time library which we look way back in week two, if you remember.
So we want to start from '2017', '1', '1'. Beginning of the year, and all the way up to whatever today's date is, right! And we call this function data, or data reader. Data reader is the function inside the 'pandas_datareader' library, and we give it our values, arguments, which are the ticker, the source Google, the start date, and the end date, and we run this, and we get data frame that... that would be the start and end date. We've got a data frame that contains our data, and we get all this data from Google which has open, high low, close, volume indexed by the date, okay! So, this is our complete set of data over there. There are documentation on that you can read it on this website, if you're interested. So, the date—the data is organized by time over here, and the index is our timeline. So, we can–do you know various times series kind of things with this, and that's what we're going to look at now. So, the first thing you want to do is we want to work on this stuff, and create–see whether we can create a new column with our...our data. So let's say, so for—we have all this stuff, and let's say we have a naive view of our trading view or something that we want to naively say that we have a open, and a close for every day, and we want to see in general, whether IBM our...our naive trading

philosophy is that if we can buy at the open, and sell at the close, then we're going to make money on the average, right! So, we want to see whether, the open is more often lower than the close or higher than the close, all right! That's our goal here.

So, what we'll do is we'll define a column called 'up days'. 'Up days' are days in which the close closes higher than the open. So for example, here we have '167' as the open. And '167.19' as the close so, that's an 'up day', okay! And here we have '169.25' as the open, and '168.70' as the close, so, that's a 'down day' because it closed down that day. So, this is a very naive view. Don't use it in practice, you're not going to make any money. So, what we want to do is we want to create a new column in our table...in our data frame over here, and the new column is going to be called 'UP', and it's have a one if the it's an up day, and a zero otherwise. So, here we can use—remember our Pandas columns are— underlying it is NumPy array. So, we can use the function 'where' that we saw in NumPy, 'np' dot 'where' and we say if the value of a cell in the closed column of our data frame is greater than the value of the cell in the open column of that...that row. Okay, we would go row by row. We look at the two column values in those rows. So for each row, we see whether this value is greater than that value. If it is, then we–in a new column is going to contain a one, and if it is not, then it's going to contain a zero, if it's less...less than or equal to it's going to contain a zero, and this becomes added to our data frame, and we now have an extra column in our data frame called 'UP'.

It's a one variable, it's an 'up day', and a zero variable it's a 'down day'. So, it's a very convenient way, and you can keep adding stuff to, to columns. So, it's a very convenient way of working with data because typically, you get data from somewhere, and then you want to create new composite elements that you know transform the data in some way, either by combining the data of, or some kind of function on it or something like that—and—for each element, and you—maybe using Lambda functions— and— creating new columns. So, you can just keep creating new columns and new—you know work with that.

Video 9 (07:22): Time series analysis

You know, you can take this stuff and we can add bunch of summary statistics to a thing here so this is a nice functional describe. What describe does is it describes every numerical column in your database, only for columns that contain numbers. So, it tells you how many elements there are and hundred and fourteen elements expressing anyways, I guess. It tells you the mean of the column, it tells you standard deviation, the minimum and it tells you the quantiles the...the quartiles, I'm sorry, of the what each quartile value is. So, this tells us that for the lowest 25 percent, the range is from one fifty—the range of open is from '150' to '156.265' for the '25' to '50' percent is from '156.265' to '170.365' from '50' to '75' it's '170' to '176' and the max is '182', I mean, this is kind of useful when we're doing our data—our machine learning kind of stuff but what it tells us off the bat is that the range here is the first quartile is 6 price points, the second quartile is 14 price points, the third quartile is 6 price points and the fourth quartile is 6 price points, right! So, the—this quartile here—the second quartile is sort of a mess, compared to everything else, in a nicely well behaved data, say, you will hopefully get equal ranges in each quartile but this is not that well behaved.

So, now we...we've...we can use this for quickly—looking at the statistics but less our interest is to...is to find out—is to see whether the—in general, if we buy at the open and sell at the close, we are going to make money. So, let's see what percent of the total days are up days and now here by doing this, we can see—we can actually do very quickly run functions on our data frames, right! That's a nice thing about it, right! So, we have our data frame and the data frame we have this—the 'Up' column right! So this becomes the 'Up' column. So, we can say for the 'Up' column compute the sum of all values, okay! So, it's going to add up all the ones, basically. Divided by the count of all items in the thing here, so the count is a '114'. So, we add up the number of times is a one by the number of fourteen by a '114' and we get the percent that we are interested in. Note that, you're doing a different day, so you probably have more than a '114' elements, okay! You should have anyway. So, this does that 48 percent of the time, we are going to be—we can, if you like, we can multiply this by a 100. So, we get a nice percent, 48.245 percent of the time, we're going to be getting the 'Close' being higher than the Open. So, if we—assuming we don't care about differences at this point, in general, we will lose money or we would have, historically, lost money if we had followed the buy at the 'Open', sell at the 'Close' policy, on the number of days. Of course, we might be I mean the—the strategy might still work because the up days may be more powerful than the down days but we don't know that right! So, just in principle, there's a simple analysis. So...so that's the goal here. Now, so what can we do, we can take a data frame, we can compute summary statistics on all numerical columns and we can apply our own transformations and operators on those values to get stuff from there. We can create new columns of using elements from the old columns which got a nice and do analysis on that as well, if you want to do that. So, now what else can we do with this? So, there's a lot of stuff we can do because we are working with time series data and we can see that time series data is ordered by time and that's where our data frame is structured. So, we want to be able to use some—do some time series analysis on it across the time. So, there's a bunch of functions that Pandas provides us, the first function is percent change what percent change does is given a column, so let's say a close column for example, given the close column, you can apply the percentage function—percent change function to that. So, what does it going to do is, going to give you for the 'Close', it's going to give you, let's say, that's a 'Close', so it's going to do the percent change for, I'm sorry', that's not that...that's the right thing. So, here the 'Close' is '167.19', '169.26'. So, we are going to get, we do a one day percent change, so the percent change, date change for '2017''01''03' is going to be unknown, since we don't know the previous day's close but for 04 of the percentage change is going to be 169.26 minus 167.19 divided by 167.19 and we going to get that value there so let's take a look at that. This is a one time period percent change and that's what we get here. We've conveniently done because this time three is data. Nice thing is that we can actually give it an argument and we can—some number, and it'll give us a 'n' day percent change, which means in this case like for example, a 2 day percent change would be 168.70 minus 167.19 2 days divided by 167.19. So, it will look at the price 'n' days before and computer percent change based on that. This is very useful for technical analysis.

They can do that, so we get initially—we get NaN's because they are we don't have enough—we don't have '13' days of data, once we have '13' days of data, we can start getting up percent changes. Pandas function can ignore NaN's, so wherever there are NaN's, it'll just ignore that. So, look, our percent change here for example, has a bunch of NaN's in the beginning and you want to compute the mean on

thecolumn.Soit'llgiveusameanwiththeNaN'signored,sothat'sanicethingaboutit.Nansarejust ignored byitandcanworkonthat.Wecancreaterollingwindows.Rollingwindowsareverynice becausewhatdo— whatyouwanttodois.So,inthepercentchange,whatwe'redoingis,welookata '13'daypercentchangeis takingtheprice.Today,Iamlookingattheprice'13'daysagoandlookingat the difference, right!
However, what you may want to do is, you won't might want to go to compute a moving averages acrossthis thing.So,youmightwanttocomputethe'21'daymovingaverageofa'13'daypricechange, sowhatyouwantto dois,youwanttoconstructwindowsof'13''13'dayseach,ohsorry,of'21'days each and for each '21' day window, we compute an average, then slide the window by one day, compute a new average, then slide thewindowbyoneday,newaverageetc.Pandashasarolling windowsfunction.Whatitdoesis,that it extractswindowssofromtheirdataseries,sowehavea data...dataframefromthedataframe,weextract theof'Close'column,wethecomputean'n'day percentchange,where'n'is'13'andwecompute'21'day rollingmeansonthat—rollingwindowson that.So,thisconstructsanobjectthatcontainsallthewindows.It doesn't,ifthegeneratorobject,it doesn'tactuallyconstructthewindowsbutitwillgiveusawindowswhenwe needit.Whendoweneed thewindows?So,whenyouwanttoconstruct,forexample,themean,sowhatthis goingtodois,it's going to construct the mean for the '13' day percent change or a '21' day rolling windows.
So,wewanttogetaninitialsetofNaN'sbecauseweneed'13'datapointsforthepercentchangeand '21'data pointsfortherollingwindows.So,weneedtoget'21''13'daypercentchanges.So,weneed '21' plus '13' data points before we start getting any values. So, you're gonna get those many NaN's first andthenthisisthe...the movingaveragethatwecanconstruct—wedoonthat.So,wecanconstruct manyofthese,forexample,Iam goingtoconstructherein'8'day,'13'day,'21'day,'34'dayor'55'day movingaverageona'13'daypercent changes.So,Igetaboutbunchofthemandthen,ifIwanttoplot them,allIneedtodoistousethefunctiondot plot.So,whatdotplotwilldois,it'lltakeadataseries,a singleseriesanddrawagraphofoutofit,right! So, I'mhereIamgoingtoplotthe'8'dayandthe'34' dayaverage.Justtakealookatit.And,wegetplotthatlooks somethinglikethis.So,thisisthemissing data.
The initial '21' plus '13' days which are NaNs, right! Because our data series contains NaNs and then this isthetwo movingaverages.Wecanseethatthe...the'20''34'daymovingaverageisthissmootherone andthe'8'day movingaverageisthenoisieronebecauseit'sshorterandyoucan,youknow,cansee that.So,weseethatthe '8'daymovingaverageisgoinglowallthewaydoneandthenitsortoftookup. So,Iguessthisis,whereIBMsis nowoutperformingitselfithistorically.So,that'sthebasicstuffwetake ourwhatwe'vedone,wecantakeour constructiondataframe,wecandosomeverybasictimeseries stuffwithit,likecomputepercentchangesfor differenttimeperiods,wecangetrollingwindows.On therollingwindows,wecanconstruct,wecandraw anythingwelike,themeans,standarddeviations mayberight otherfunctionsonitdoesn'treallymatterand youcandowhateveryoufeellikewiththat but that's what Pandas provides us.

Video 10 (9:51): Risk return analysis: example

WithPandaswecandosimplekindofanalysisveryveryquickly.Soforexample,hereIwanttolookata linear regressionexamplethatI'mgoingtodohereandthegoalhereissortofroughly,it'sa...it'sa contrived example.So,forgivemeforthatbuttherearevariouscompaniesthatworkinthesolarsector andsomeofthem don'tworkanymore.They'vemergedwithothercompaniesbutlet'ssaywehave threeofthesecompanies, FSLR,whichisFirstSolar,RGSEwhichisanothersolarcompany,and—Ithink it'scalledResourceCapitalor something.Iforgetthenameofit,andthenthere'sSCTYwhichisSolar City.

Theyarethreedifferentcompanies.Theyeither—eachcompanyeitherbuildsorleasessolarpanels. They have slightly different marketing strategies or business models so to speak and there's also a–an ETF, that's available, thatiscalled—thathasatickerTANwhichtradesonsolar.It'sasolarETF,Right!So,whatwewanttodoiswewant to look at our— our goal is to sort of use Pandas to try to figure out what a good investment strategy would be, if you have, you know, historical data on these companies. So, we're going to use—first we're going to look at things like correlations between the returns of these companies and the correlations between TAN and the three companies and then try to figure out what the risk reward profiles are and then say, "Hey! what should we do?" We've...we've money to invest and we want to sort of invest that money. So, it's...it's an exploration process and we're going to use Pandas to...to do this stuff here.

So,thefirstthingwewanttodoisactuallyextractthedata.So,Iamgoingtoextractdatahere fromJuly '1st', '2015'toJune'1st','2016'.Thereasonforthisis...isthatthedataisnotcompletebecausesome companiescame into existence later and some companies have gone out of existence in...in this example. I'm just going to work with this stuff here. So, I start with this, end with this, and I'm going to get the data from Google, and now because that I'm getting data from multiple items, multiple tickers, I'm going to give them all in a list. Earlier, there was only one ticker I can just give that ticker itself but I have four tickers here, so I'm going to give them all in a list. I have the start date and the end date, and all I want from that is the closed prices, from what I get back, I don't want anything other than the closed prices. I don't want to open high-lows and all that stuff and I take all this thing and I put it inside data frame called 'solar_df', and I got the data. I'm going to take a look at it and that's what the data I get, right! Indexed by date and with the columns correspond to the tickers that we have because I don't...I don't really need the closed price, right! So that's...that's my data frame. That's what I'm going to work with. Now, I'm looking at the returns on these—in the things. So, the returns I want to calculate here.

I'mgoingtocalculatetheonedaypercentchange.So,Iwanttoconvertintoanewdataframecalled 'rets'andthe 'rets'containstheone-daydata—one-daypercentchangeforeachofthesestocks,right! Forthe...forthedates given.Ihavethat.Now,Ican—simplestuffIcando.Icansay,"Letmelookateachstockandseehowitcorrelates withtheETF."Icandoitvisuallybydrawingascatterplot.So,Ido 'plt' dot 'scatter', 'plt' is 'matplotlib' function, remember? And, matplotlib library and saying, for matplotlib library, use the 'scatter function to scatter the returns on FSLR and TAN. And, looking at this, I can see there is, you know, reasonable correlation. There are some outliers, but all in all, it looks like, you know, reasonably correlated set up. TAN is the ETF and FSLR First Solar is the

solar—it sells solar panels. I can do a scatterplot between RGSE and then this is a lot messier. So, there is some vague correlation but it's very very messy, right! I know we can look at Solar City and TAN, and that's a little bit better but looking at it—yeah, it's also pretty well correlated not as neat as First Solar but—and the idea here is that we can look at correlations as numbers, and we can look at them as scatterplots.

Visuals are usually better when you're conveying information. So, this is one quick way of looking at visually trying to get this stuff or we can just go ahead and get the correlation matrix, and take a look at it. So, there's a function called 'corr', 'corr', which you apply to a data frame, any data frame, and it is going to pull out the correlations between the columns on the data frame. So, we get a—in our case, we have four columns. So, we get a four by four matrix, and just looking at this, we can see that our visuals are pretty good. It tells us First Solar to the ETF is '0.67'. CTY is a little bit lower, and RGSE is kind of low, right! So, we can look at this stuff and figure out, you know, what the correlations are. So, now we can do some very basic risk analysis here. What we're going to do is, we're going to draw a plot that...that plots the risk reward— risk return tradeoff for each of these four tickers that we have there.

So, let me just draw this and I'll show and explain this to you in a second. So, what we have in our thing here is that we've got a—the...the means we take our returns and we take our standard deviations and we compute the means and the standard deviation for each of the columns in this...in this thing here. So, if I insert this here, I do 'rets' dot 'mean'. I get four data points, right! And, if I do 'rets' dot 'std', I get four data points. So, what I'm doing is, I'm taking each data point and the returns are on the 'x' axis and the...the mean returns, expected return on the 'x' axis, and the standard deviation is on the right—on the 'y' axis. So, for TAN, for example, the standard deviation is '0.25'. So, that's over here and the mean was negative point zero two...zero two, so that is probably reads somewhere at '0.00'. '-0.005', so that's somewhere over there, right! And, then I'm—when you—of course you draw a graph, you want to label it because you want to show it to someone and they're not going to—want to see this without labels.

So, when—the—matplotlib you can have, you can just label it by saying 'x' label, 'y' label, and giving labels. So, we get expected returns here, and standard deviations over there by just using these values here, expected return and standard deviations there. And then, what...what I'm doing is, I'm going to zip this stuff up here and play the 'zip' function, and then take each point and draw little boxes around them. So, I get the name of the ticker there and you can read—you can see all the stuff is pretty straightforward actually. So, the—this tells you where everything is positioned, I have the type, I'm giving the color as yellow and I'm drawing little boxes around it. So, that's the box style round, and the pad is the space around RGSE. These are, you know, various things that we want to do for the–for making the graph look pretty. The easiest way to figure all this out, I could sit and talk about it, is to actually change stuff. So for example, if I change '0.5' to '0.9' here and ran this. Not very clear. Let me take '0.1' See, that's telling you the...the intensity of the color in this thing here, right! So, each—if you just try changing these things, and see what it looks like. But this is essentially the labelled stuff.

So, I've labelled each of these things here and that's getting labelled, right! Here, the label, right! So, that's getting labelled over there and it says TAN, FSLR, SCTY, RGSE, and we can see all that stuff.

Looking at this plot over here, it's pretty clear that if you look at our data, and this is a very—actually a very very illustrative visual when you think about it. Because it's telling us that the..the highest average return that you can expect is from FSLR, right! Because expected returns is on the 'x' axis. So, FSLR has the highest return and it has a slightly higher Standard Deviation than TAN. So, its risk is higher and the return is higher, right! On the other hand,

SCTY, and RGSE; SCTY gives you pretty much the same return as TAN with a much higher risk, and RGSE gives you a lower return with a higher risk. So, if you are a trader, you would definitely not want to invest in RGSE, and SCTY of—with the caveat that past performance doesn't reflect—doesn't necessarily mean you are going to have the same thing in the future. But based on the past, you would not want to—because investing in RGSE and SCTY doesn't give you any additional return but gives you a lot more risk, okay! RGSE actually get lower return. You might want to invest in FSLR because you get a higher risk. Depending on your risk averse—risk aversion, whatever level of risk aversion you have. You might want to either just say, "I'm willing to accept a slightly lower return and a lower risk with TAN or a slightly higher return with slightly higher risk with FSLR." And you can, you know, there are various matrix that you can use to figure our which way is better. But you can see this a very...very very clear graph because it really tells you what's—what the risk return profile looks like.

Video 11 (04:37): Regression example

The last thing we're going to look at is regression. How to do a regression on this data here? So for regression, it's very straightforward. You have a 'y', a dependent variable, and you have a bunch of 'X's, independent variables. So, what you want to do is, you want to specify the 'y' and the 'X's. You want to add an intercept because there could be 'y' equals 'alpha' plus 'beta one', 'X one', plus 'beta two', 'X two', all that stuff. And then, you want to model the regression, and then look at the results. So, there's a nice package called statsmodel which cont—models which contains many regression packages. We're going to use the ordinary least squares version of that thing there. So, let's start—just walk through these steps. So, first thing we do is we import from statsmodel 'api'. We import the...the 'api' from there. So that, we have and we have that as 'sm'. We're going to call that 'sm'. So, we got that. We want to construct our 'y' variable.
Our 'y' variable is this, the ETF. We want to see how much of the ETF's returns are explained by the various—the four different independent variables. The three different independent variable returns FSLR, RGSE, and SCTY. So, that's our 'y'. Our 'X'—the 'y' is only one, right! Usually we have only one dependent variable. We have a series of independent variables, so that will be put on a list over here and that becomes our 'X' variable. So, we have 'X' and we have a 'y'. We construct the model by calling the OLS function from constructor, really, from our statsmodel package. So, that's 'sm' dot 'OLS'. We tell it what the 'y' variable is and that's here. We tell it what the 'X' variable is and that's this. That becomes our extract from the data frame and we say, if there's any missing value, then just drop the entire row, right! Missing equals drop says, if you have any missing value and remember we have returns, so we're going to have some missing values, right! Just drop the entire row. So, this is constructing the model, and what you want to do now is, you want to run this function called 'fit' and this is going to be very standard thing we will be using, excuse me, down the road as we do more machine learning. The—what will you always be doing is constructing a model object and then getting a—calling a fit function to actually get the—to parametrize it. To get you know from our data, whatever our data is. So we should be using a trading testing sample, and all that before purposes of our little example here.
We're just going to fit the whole thing. So, what this does is now is, it actually constructs the line, the 'X'–'y' equals 'alpha' plus 'beta one' 'X one' plus 'beta two' 'X two'. It's going to estimate the 'alpha' and

the 'beta one', 'beta two', 'beta three' in this case. And once we do that, we can call it 'result' dot 'summary' to see the results and this is what we get here. So, this...this gives us our complete stuff over there, and what we're interested in is the 'R-squared' which is a nice '0.851' which is kind of a good score over there. We're interested in the 'P' values which is here, and we find the probability that this— only one that's reasonable actually is this one, FSLR, right!

We're not really interested in anything else, and so this, you know, is our very quick and dirty way of running a regression over there and you look at the results and figure it out and you got that. So, we...we will look at regression again later but for now, you know, we just know we can run this. So, once we've got that, we can actually look at our fitted values, and compare them with a simple plot that looks at our fitted values with the actual values. So, we have—the actuals are— so this tells us here, the actuals are—the fitted values are the blue curve, no, the fitted values are the orange curve, and the actual value are the blue curve, the nosier one is the actuals. And, so this tells us visually how good our fit is, and it looks pretty, not, doesn't look too bad, actually. I'm surprised. I thought it would be not so great but it doesn't look too bad. And, the reason it doesn't look too bad I think is because FSLR is so highly correlated with TAN that it probably explains why almost all the variants in the...in the...in this model, and you know we can see that. It's got a very high 'P' value in this thing here. So, that's pretty much it. So, that is our introduction to Pandas.