<https://www.youtube.com/watch?v=Lsue2gEM9D0&list=PLblh5JKOoLUICTaGLRoHQDuF_7q2GfuJF&index=28>



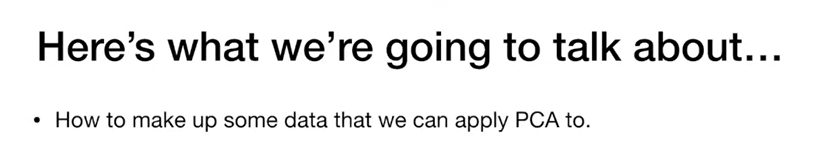
Today we're gonna be talking about how to do PCA and Python.

It's gonna be clearly explained.

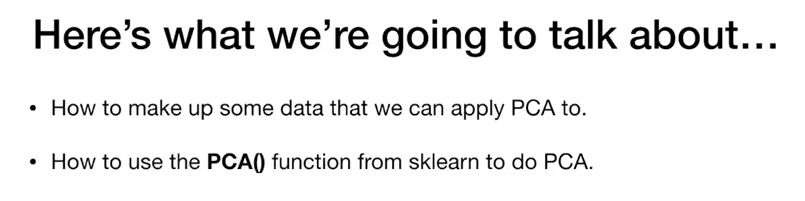
However before I get started I need to have a big shout out for Chris yoga zimsky, he wrote the code that I'm gonna be talking about and he's the reason why the stat quest exists to begin with.



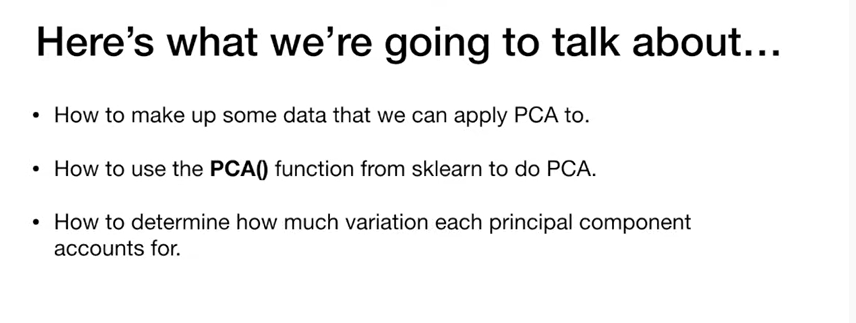
Here's what we're going to talk about.



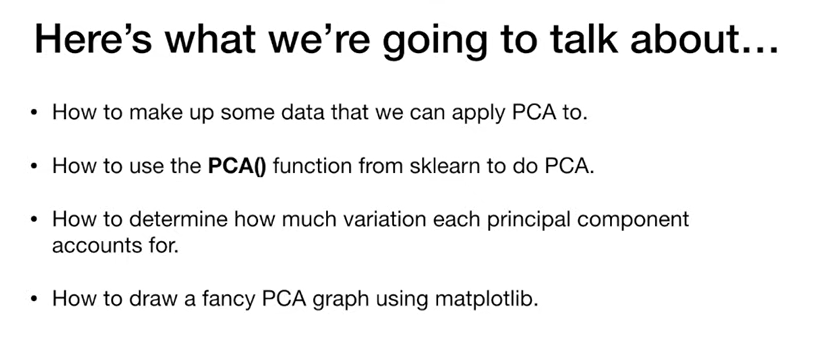
First we'll talk about how to make up some data that we can apply PCA to.



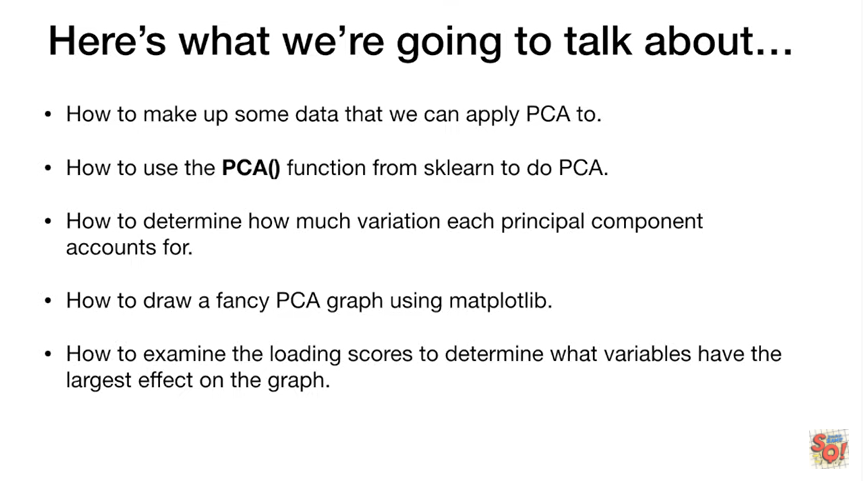
Then we'll talk about how to use the PC a function from scikit-learn to do PCA.



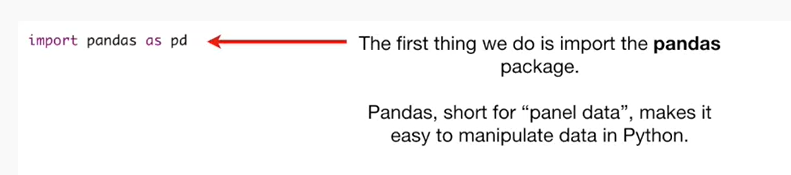
We'll also talk about how to determine how much variation each principal component accounts for.



We'll talk about how to draw a fancy PCA graph using matplotlib.

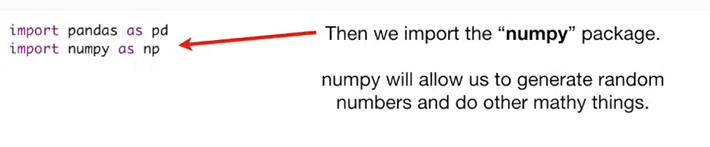


And lastly we'll talk about how to examine the loading scores to determine what variables have the largest effect on the graph.



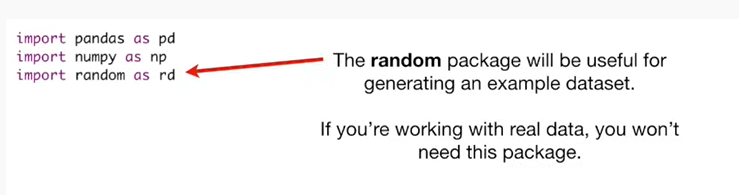
The first thing we do is import the pandas package.

Pandas, short for panel data makes it easy to manipulate data in python.



Then we import the numpy package.

Numpy will allow us to generate random numbers and do other mathy things.

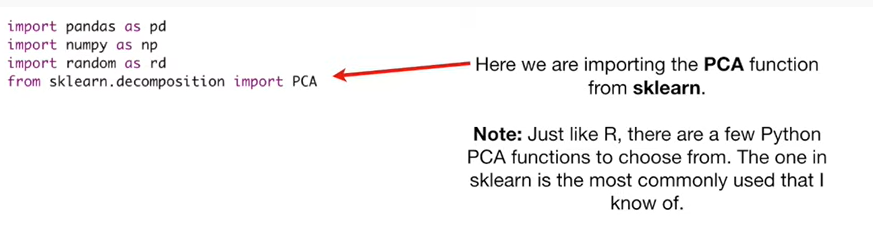


The random package will be useful for generating an example dataset.

If you're working with real data you won't need this package.



Here we are importing the PC a function from scikit-learn.

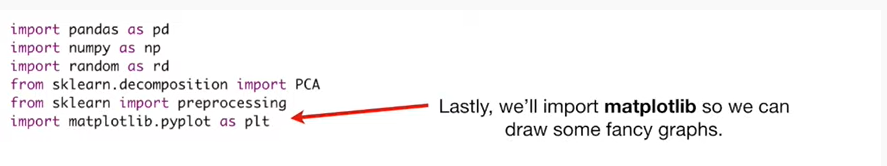


Note : just like R there are a few Python PCA functions to choose from.

The one in scikit-learn is the most commonly used that I know of.



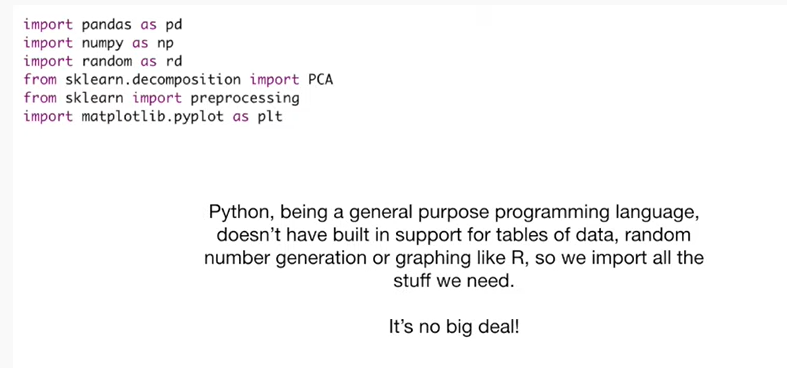
The pre-processing package from psych had learned gives us functions for scaling the data before performing PCA.



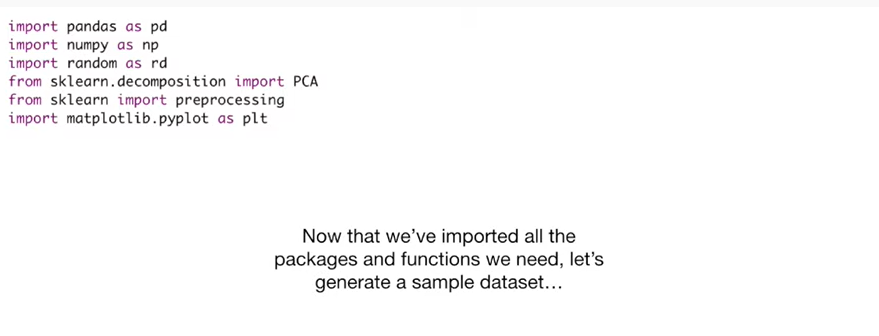
Lastly we'll import matplotlib so we can draw some fancy graphs.



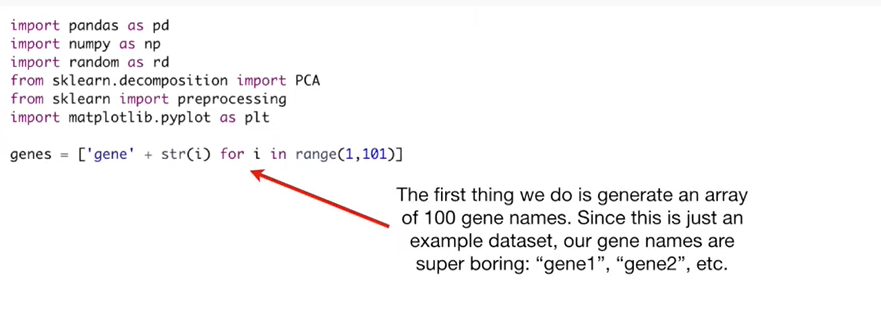
Python, being a general-purpose programming language doesn't have built-in support for tables of dat,a random number generation or graphing like R, so we import all the stuff we need.



It's no big deal !

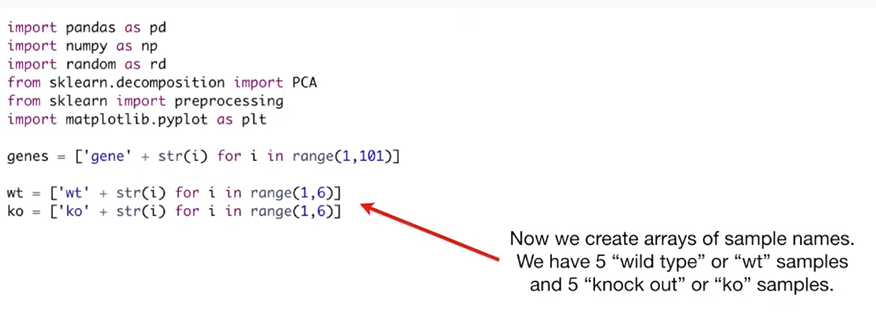


Now that we've imported all the packages and functions we need let's generate a sample data set.



The first thing we do is generate an array of a hundred gene names.

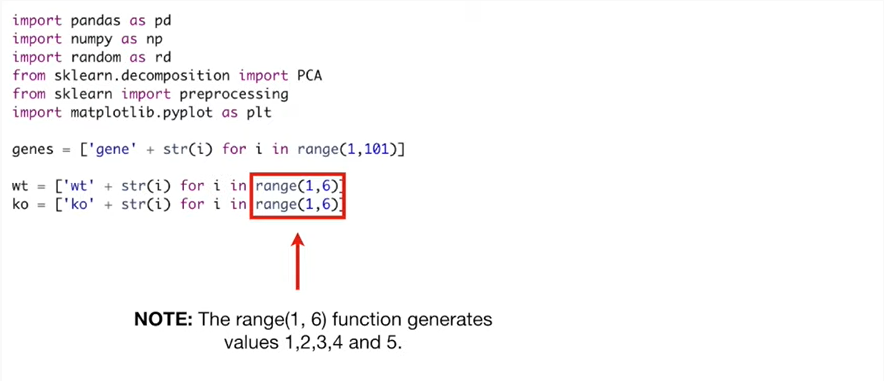
Since this is just an example data set our gene names are super boring gene one, gene two, etc.



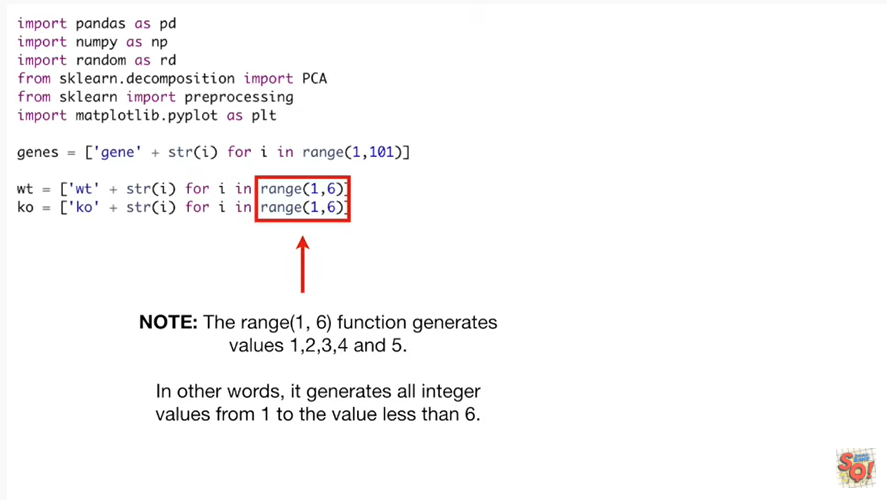
Now we create arrays of sample names

We have five wild-type or WT samples and five knockout or ko samples.

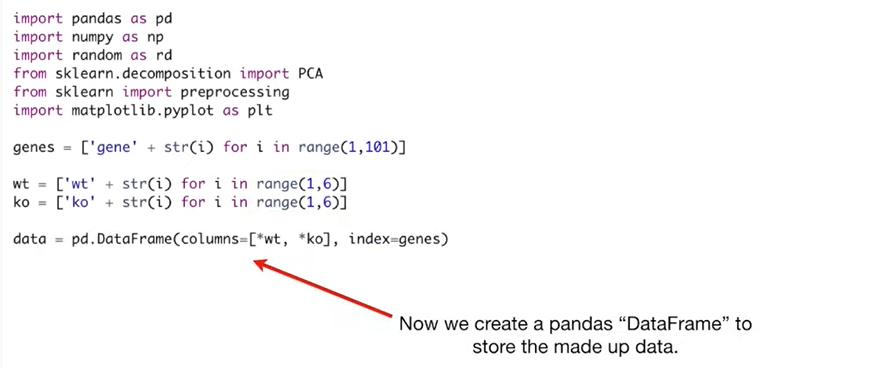
If you don't know what wild-type or knockout samples are don't worry about it just know that there's two different types of samples.



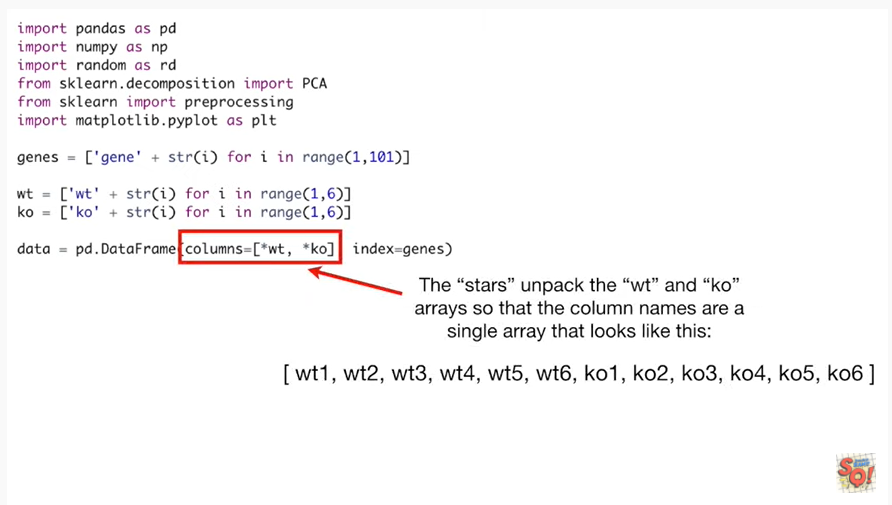
Note : the range one comma six function generates values one two three four and five.



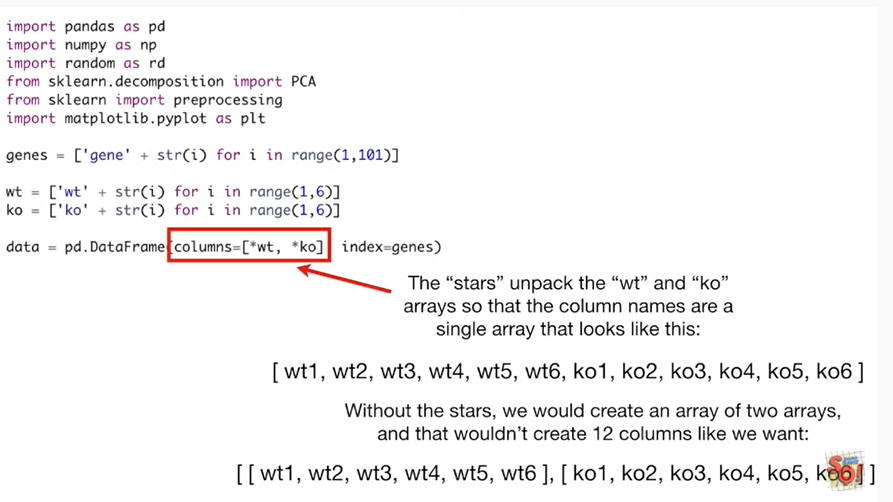
In other words it generates all integer values from 1 up to the value less than six.



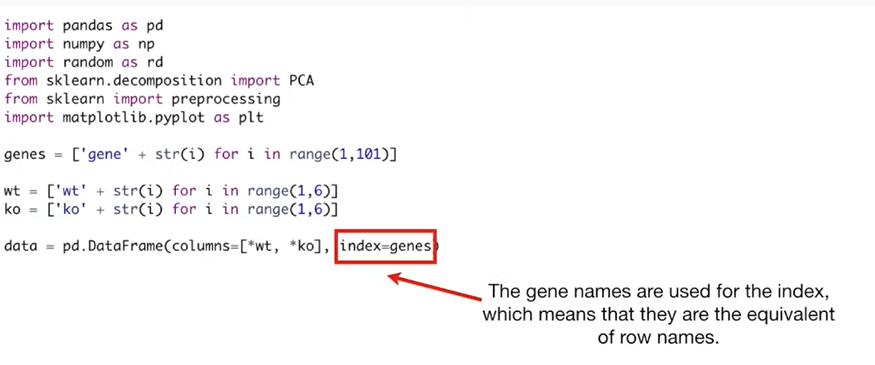
Now we create a panda's data frame to store the made-up data.



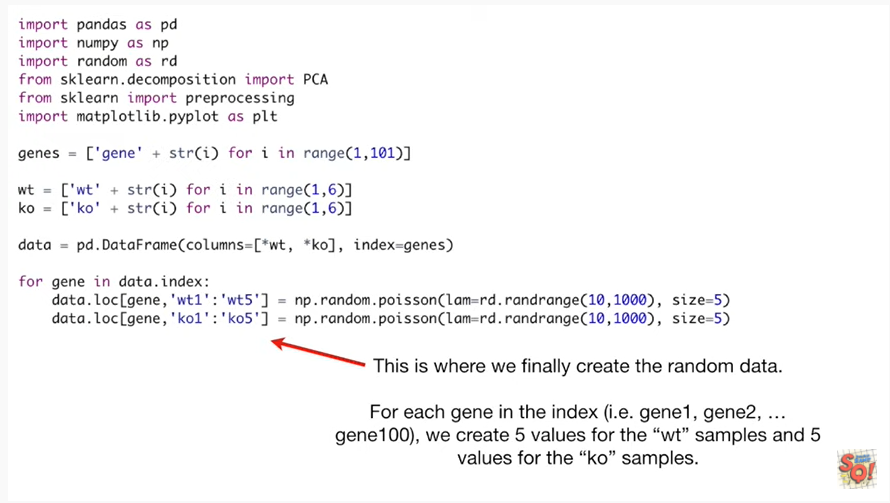
The stars unpack the WT and ko arrays so that the column names are a single array that looks like this.



Without the stars we would create an array of two arrays and that wouldn't create twelve columns like we want.

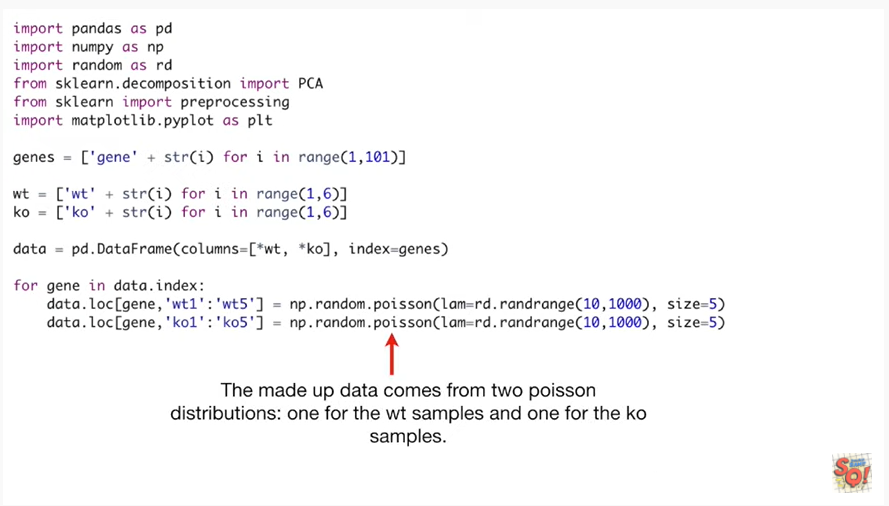


The gene names are used for the index which means that they are the equivalent of row names.



This is where we finally create the random data.

For each gene in the index ie gene one gene two dot dot all the way to gene 100 we create five values for the WT samples and five values for the KO samples.



The made up data comes from two Poisson distributions one for the WT samples and one for the ko samples.

Note : RNA sequencing aficionados will recognize that the Poisson distribution isn't the right one to use for RNA seek data.

No big deal, this is just a made-up dataset.

So don't worry about it.

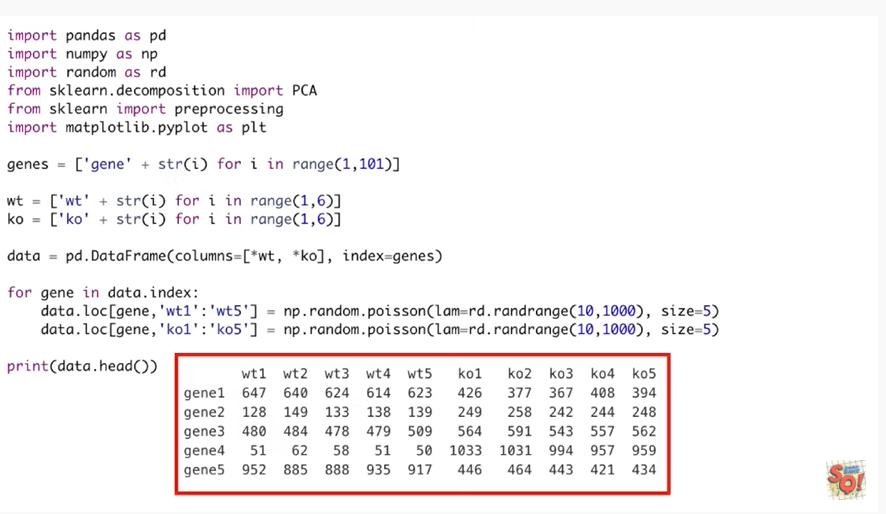


For each gene we select a new mean for the Poisson distribution.

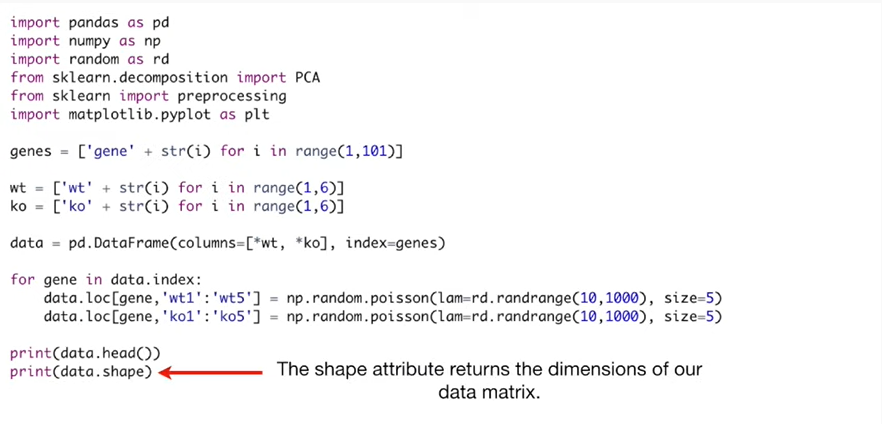
The means can vary between 10 and 1000.



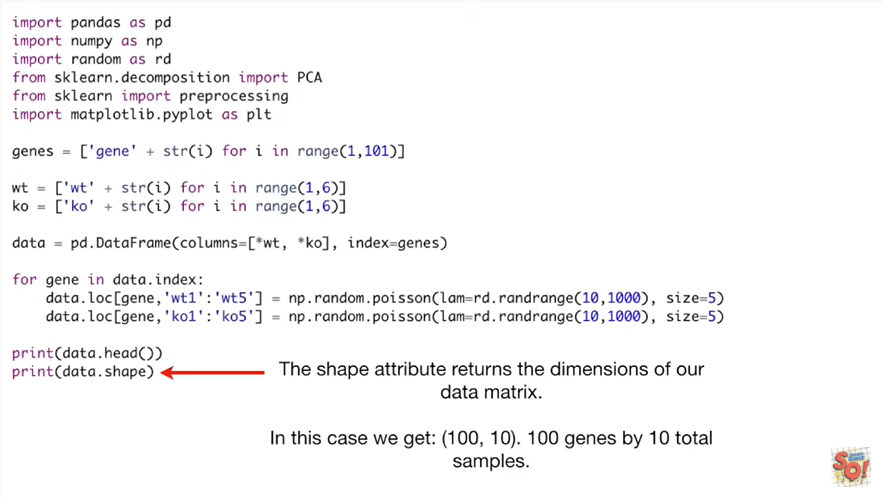
The head method for our data frame returns the first five rows of data.



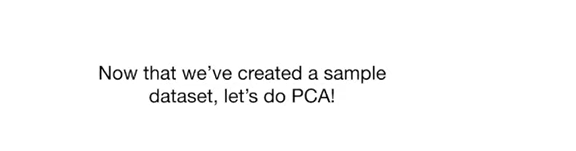
This is useful for verifying that the data look the way we expect they should.



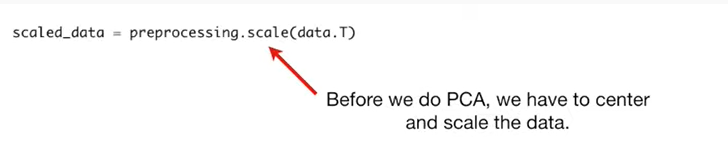
The shape attribute returns the dimensions of our data matrix.



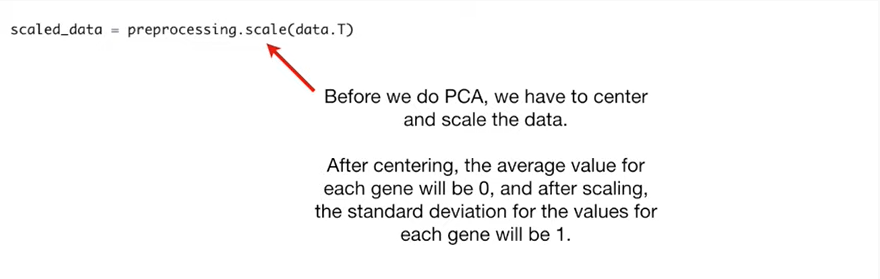
In this case we get a hundred comma 10 100 genes by 10 total samples.



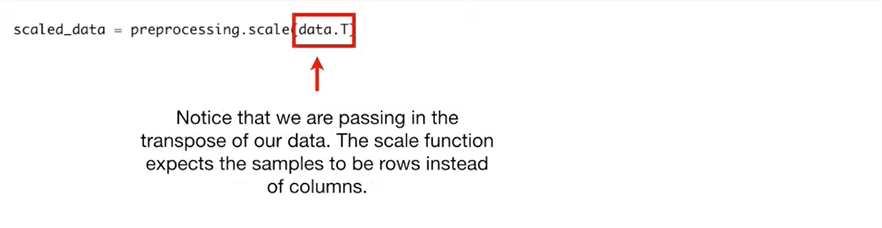
Now that we've created a sample data set let's do PCA !



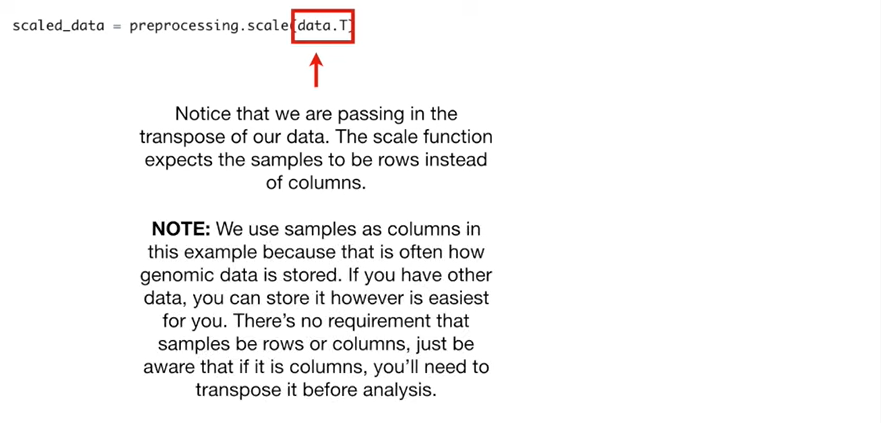
Actually before we do PCA we have to Center and scale the data.



After centering the average value for each gene will be zero and after scaling the standard deviation for the values for each gene will be 1.



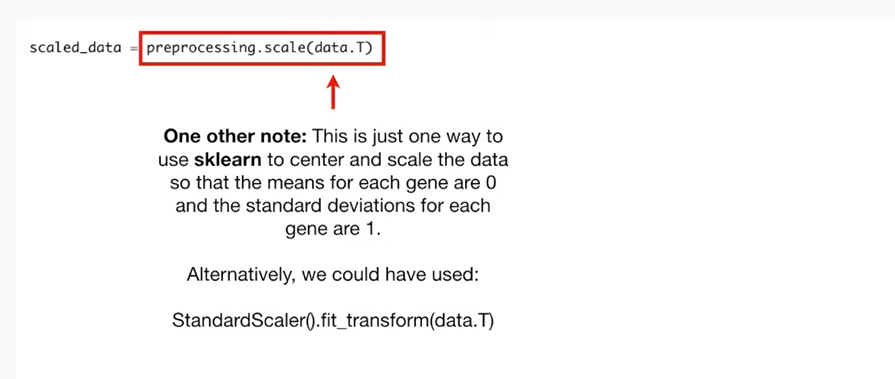
Notice that we are passing in the transpose of our data the scale function expects the samples to be in rows instead of columns.



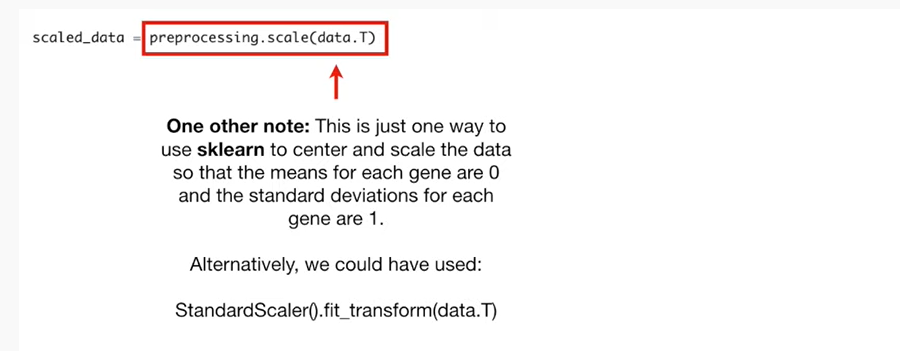
Note : we use samples as columns in this example because that is often how genomic data is stored.

If you have other data you can store it however is easiest for you.

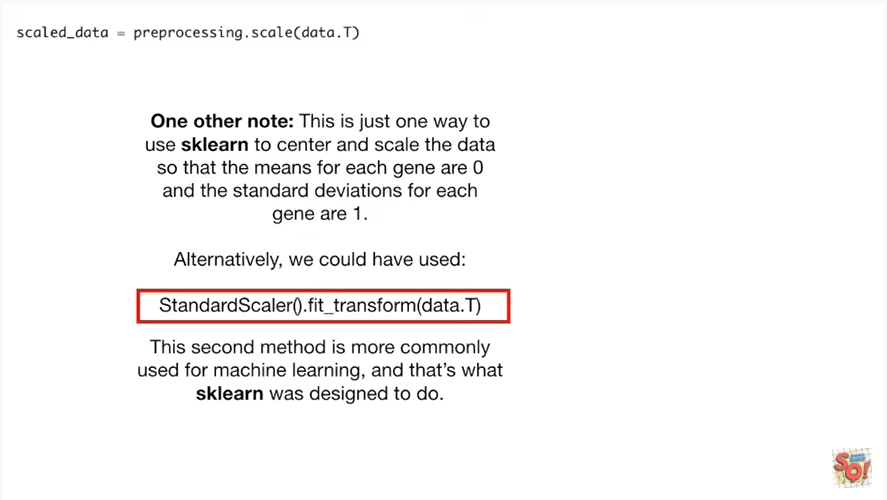
There's no requirement that the samples be rows or columns just be aware that if it is columns you'll need to transpose it before analysis.



One other note this is just one way to use scikit-learn to Center and scale the data so that the means for each gene are 0 and the standard deviations for each gene are 1.

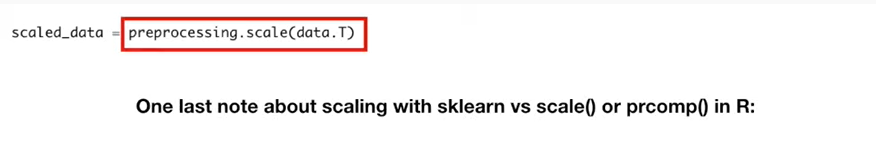


Alternatively we could have use : standard scalar fit underscore transform.

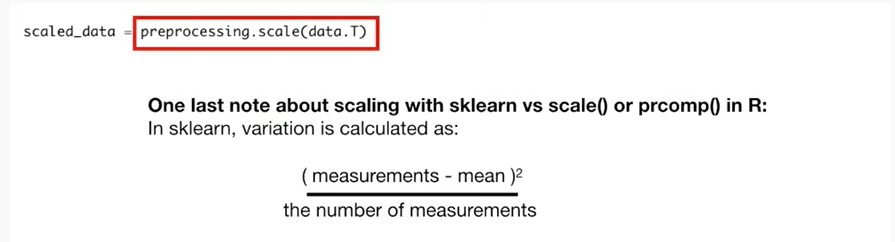


This second method is more commonly used for machine learning and that's what scikit-learn was designed to do.

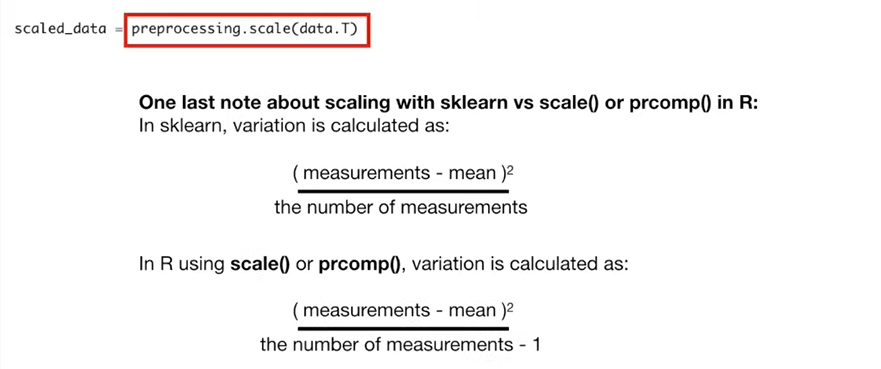
That's why it's available.



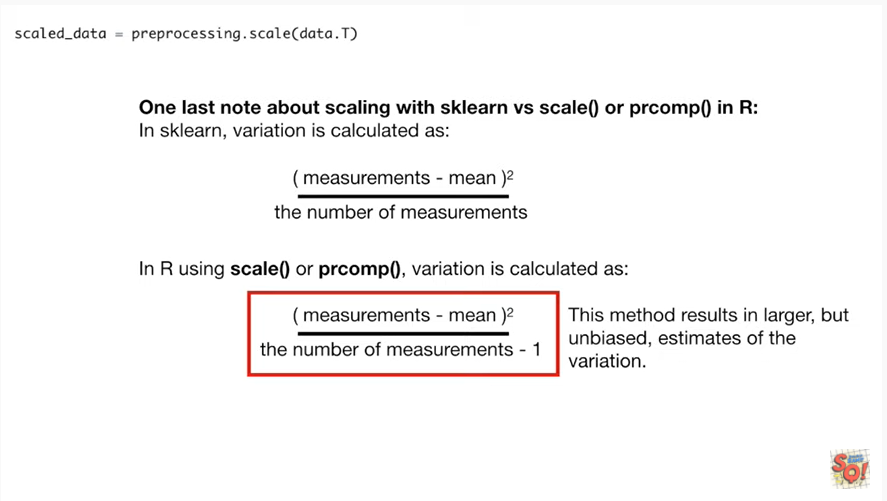
One last note about scaling with scikit-learn versus using scale() or prcomp() in R to do the scaling.



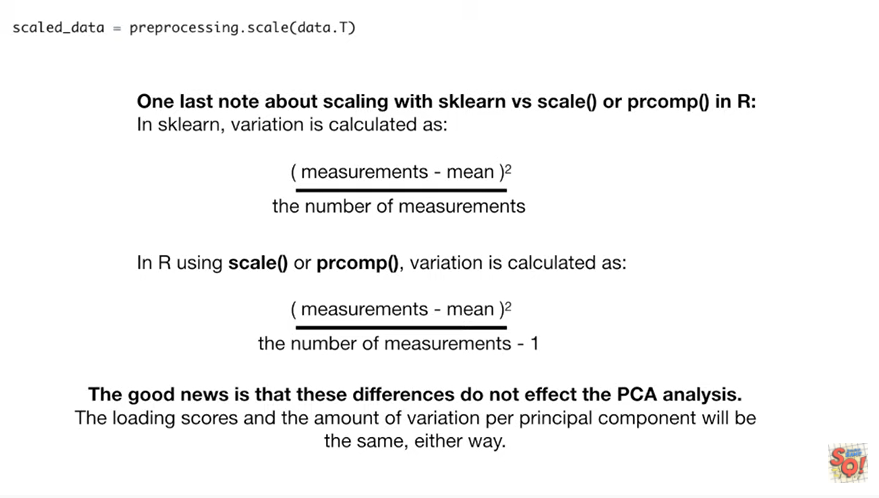
In scikit-learn variation is calculated as the measurements minus the mean squared divided by the number of measurements.



In contrast in our using scale or per comp variation is calculated as the measurements minus the mean squared divided by the number of measurements minus one.

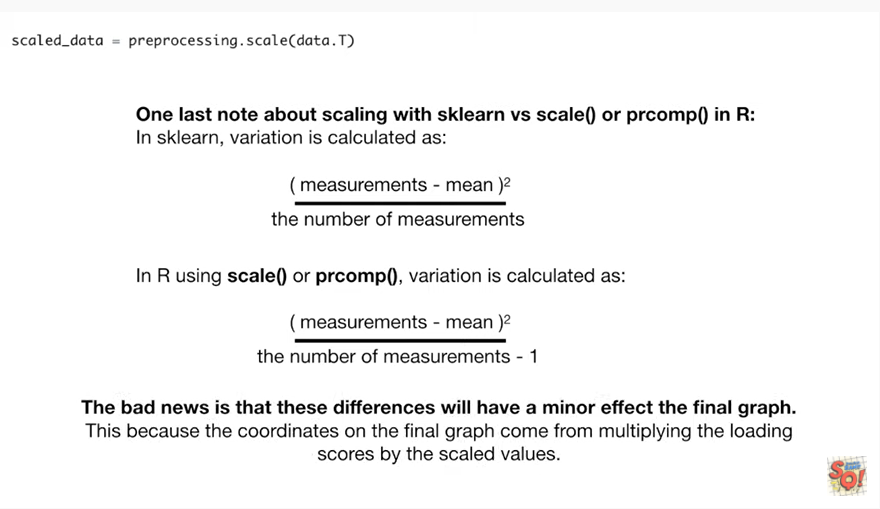


This method results in larger but unbiased estimates of the variation.



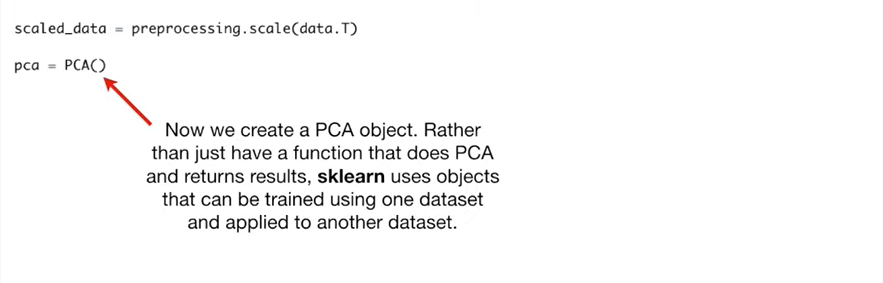
The good news is that these differences do not affect the PCA analysis.

The loading scores and the amount of variation per principal component will be the same, either way.



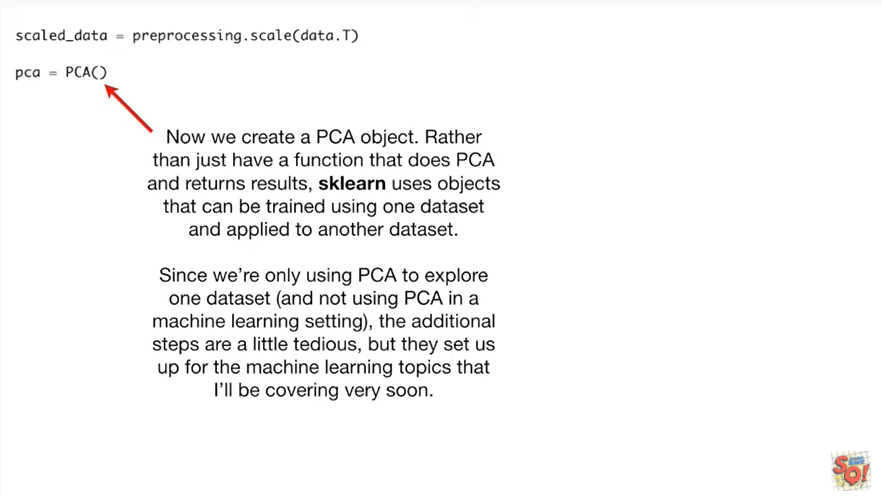
The bad news is that these differences will have a minor effect on the final graph.

This is because the coordinates on the final graph come from multiplying the loading scores by the scaled values.

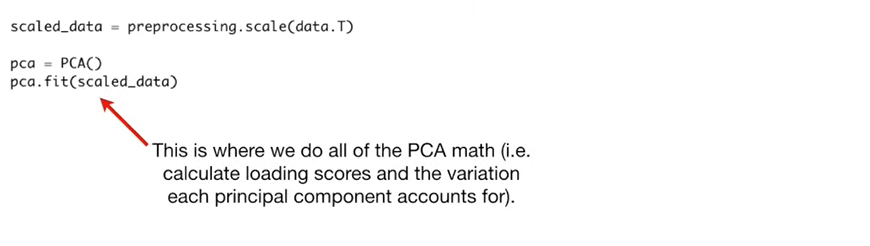


Now we create a PCA object.

Rather than just have a function that does PCA and returns results scikit-learn uses objects that can be trained using one data set and applied to another data set.

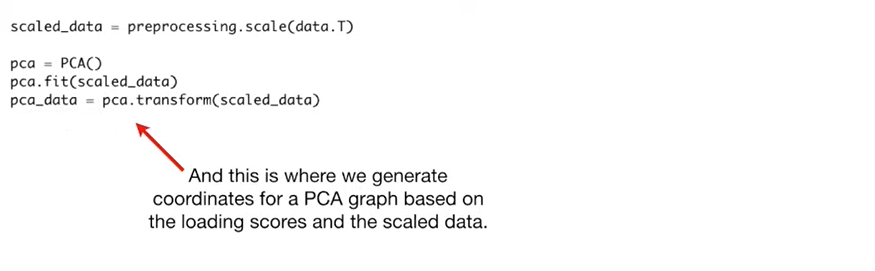


Since we're only using PCA to explore one data set and not using PCA in a machine learning setting the additional steps are a little tedious but they set us up for the machine learning topics that I'll be covering very soon.

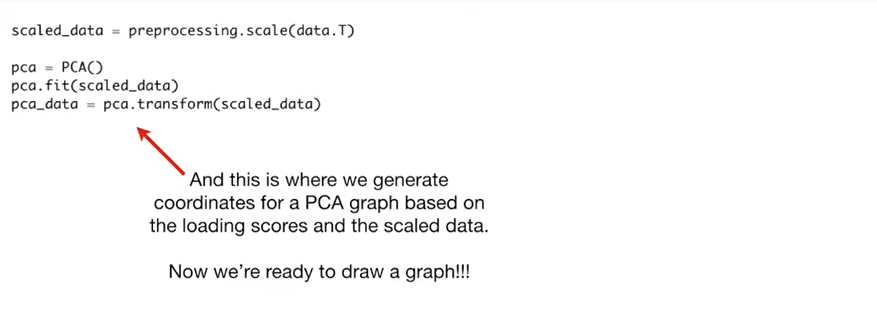


Then we call the fit method on the scale data.

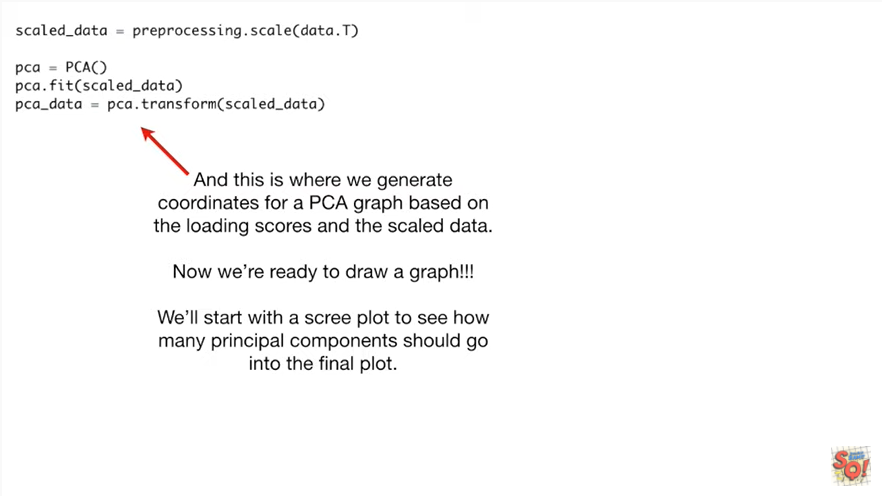
This is where we do all the PCA math ie we calculate the loading scores and the variation each principal component accounts for.



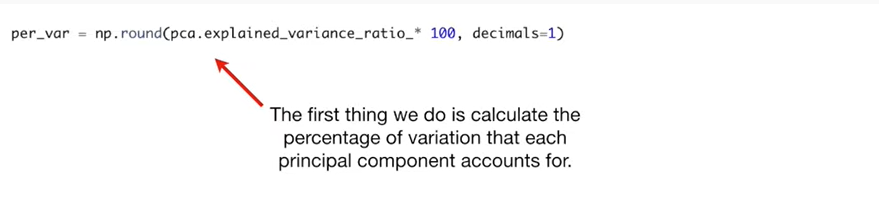
Lastly this is where we generate coordinates for a PCA graph based on the loading scores in the scaled data.



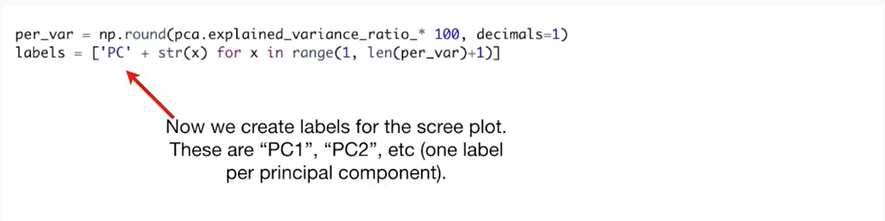
Now we're ready to draw a graph !!!



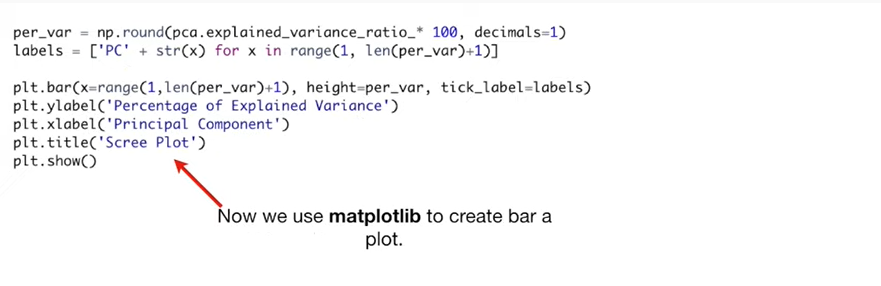
We'll start with a scree plot to see how many principle components should go into the final plot.



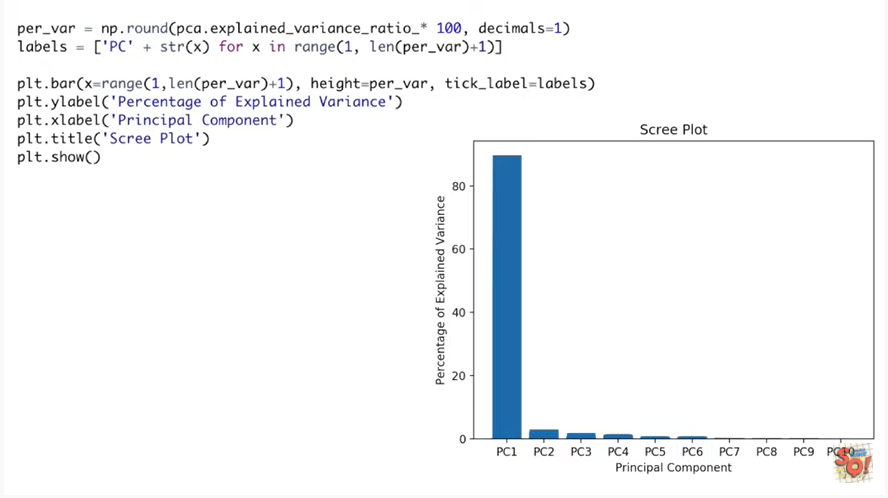
The first thing we do is calculate the percentage of variation that each principal component accounts for.



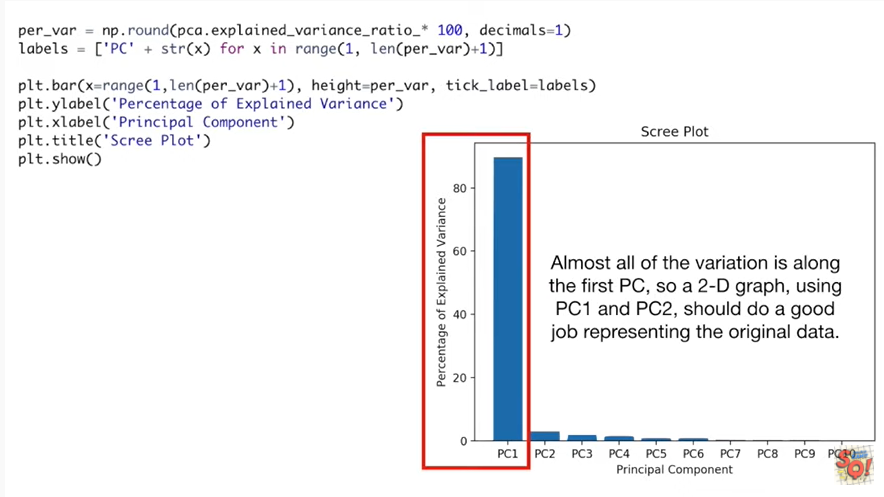
Now we create labels for the scree plot these are pc-1 pc2 etc (one label per principal component).



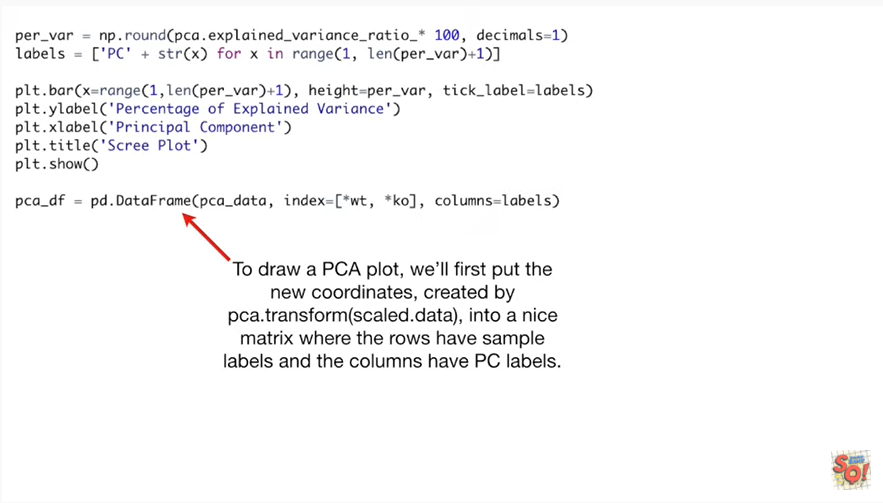
Now we use matplotlib to create a bar plot.



And here's what the scree plot looks like with all the fancy annotation that we added using the y label X label and title methods.



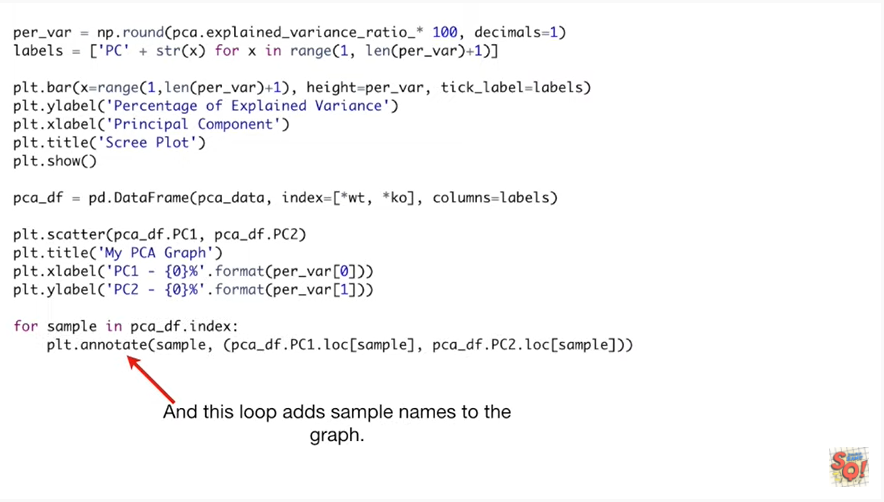
Almost all of the variation is along the first principal component so a 2d graph using pc1 & pc2 should do a good job representing the original data.



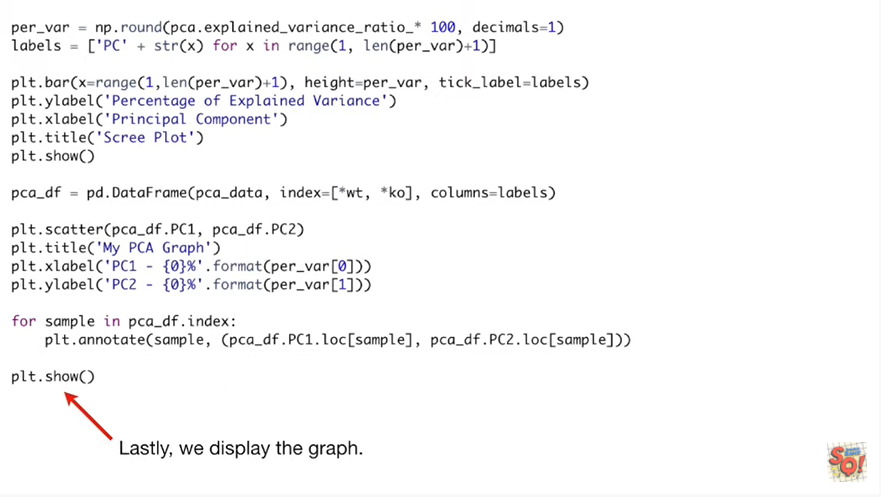
To draw a PCA plot well first put the new coordinates created by PCA transform of the scaled data into a nice matrix where the rows have sample labels and the columns have PC labels.



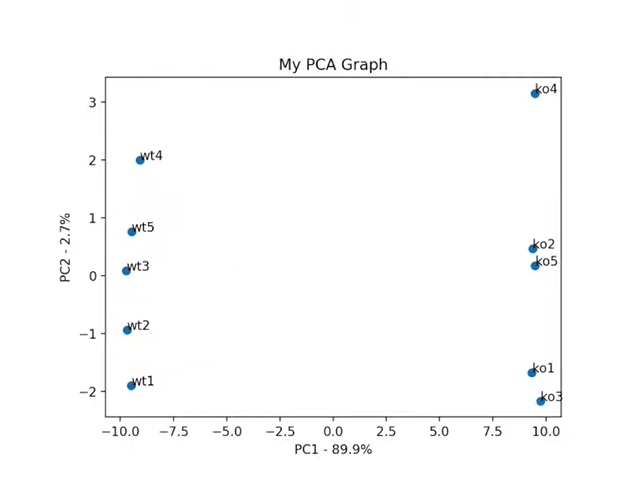
These commands draw chatter plot with the title and nice access labels.



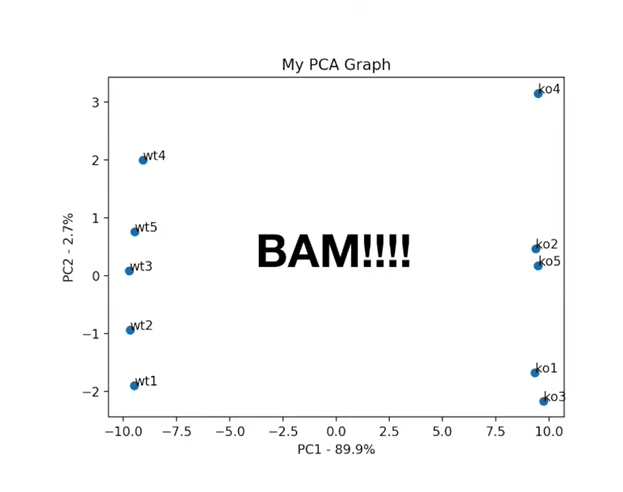
And this loop adds sample names to the graph.



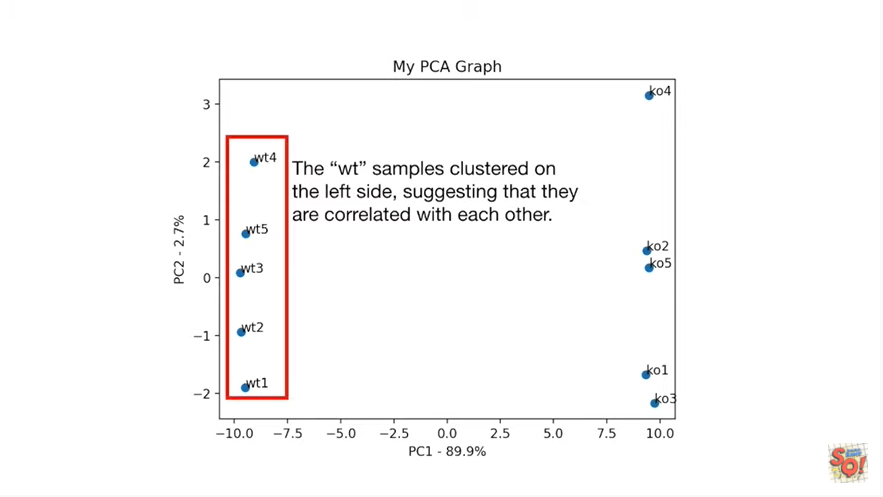
Lastly we display the graph.



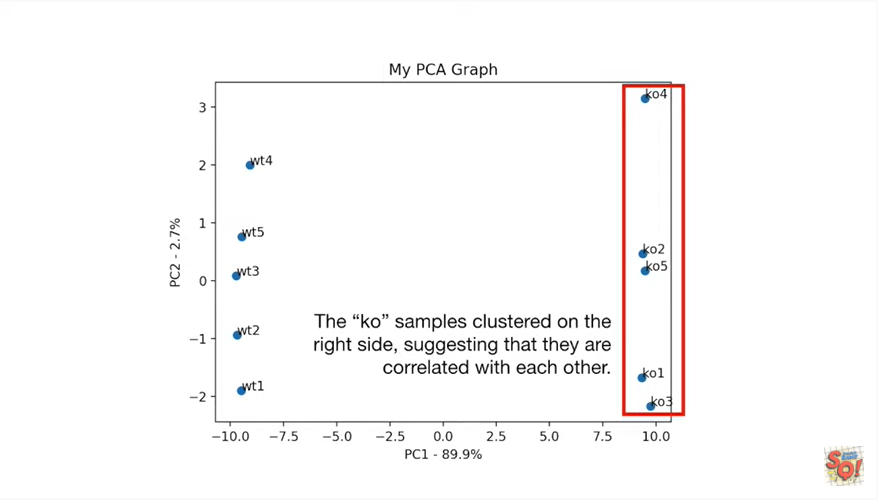
There it is wow so awesome.



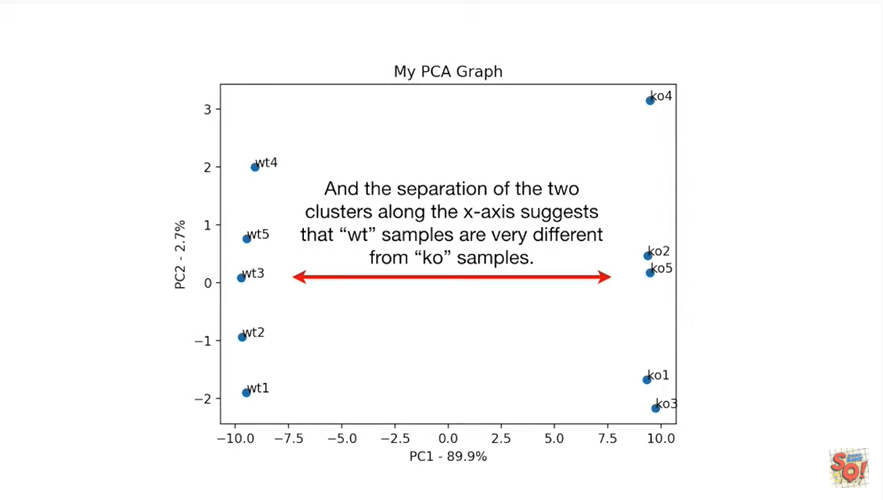
BAM !!!!



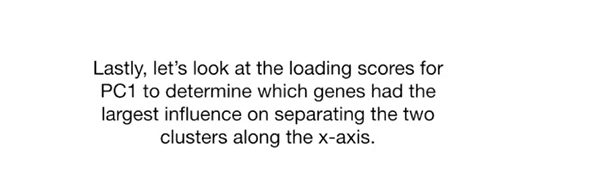
The WT samples clustered on the left side suggesting that they are correlated with each other.



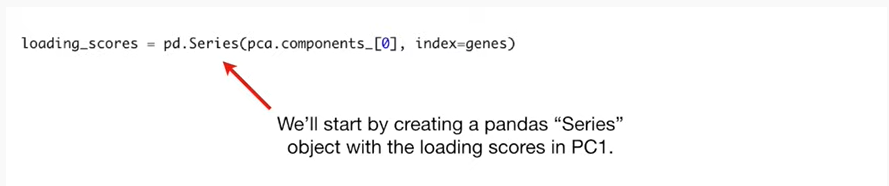
The KO samples clustered on the right side suggesting that they are correlated with each other.



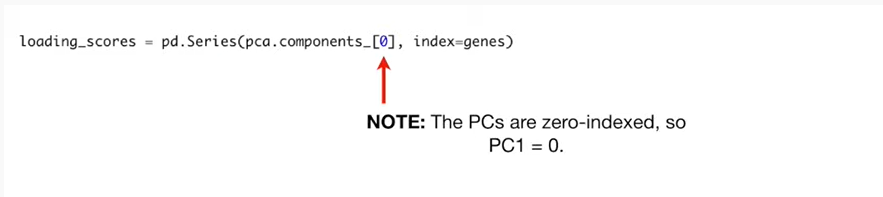
And the separation of the two clusters along the x-axis suggests that the wild-type samples are very different from the knockout samples.



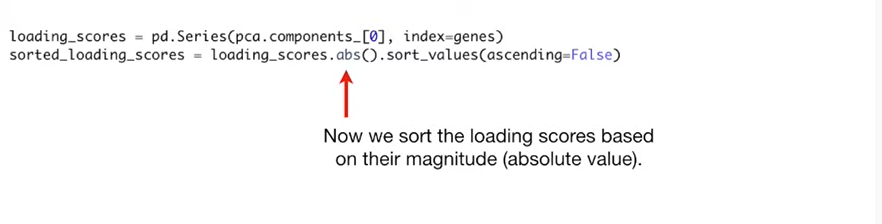
Lastly let's look at the loading scores for PC one to determine which genes had the largest influence on separating the two clusters along the x axis.



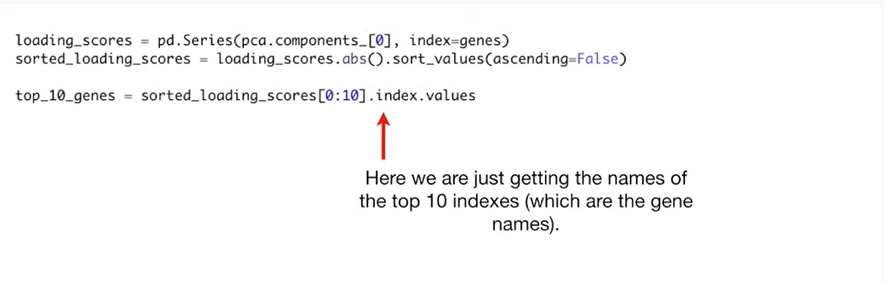
We'll start by creating a panda's series object with the loading scores in principal component one.



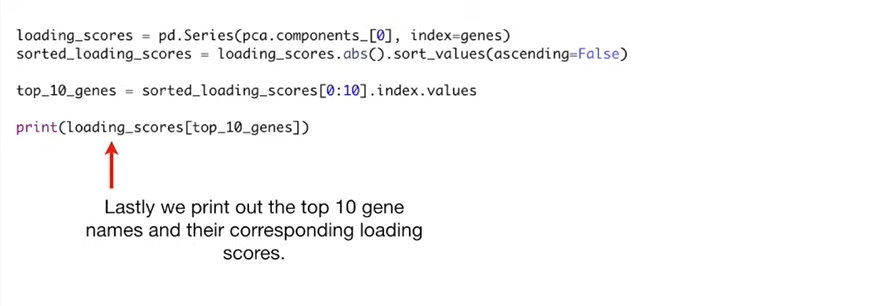
Note the principal components are zero index so PC one equals zero.



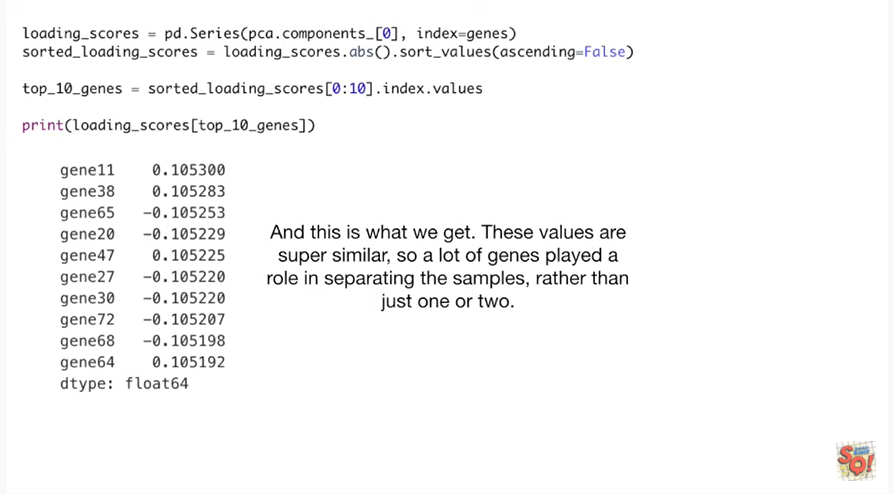
Now we sort the loading scores based on their magnitude aka we take the absolute value.



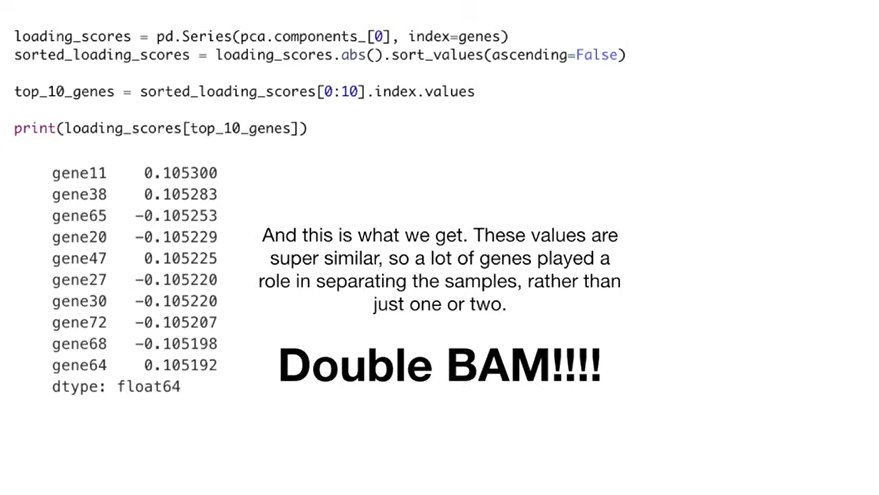
Here we are just getting the names of the top ten indexes which are the gene names.



Lastly we print out the top ten gene names and their corresponding loading scores.



And this is what we get these values are super similar so a lot of genes played a role in separating the samples rather than just one or two .



Double bam !!!!