Module 6
Video Transcripts

Video 1: Importing and Exporting Data (4:47)

In this week, we're going to cover several things from importing data to export it, to manipulate it, and plot it. This first notebook, we will be exploring how to import and export data. We'll be importing the IBM Employee Attrition dataset. That will help us in the future notebooks as well. We will look at the top five rows of that dataset, and we will export the data as a '.csv'. CSV meaning Comma-separated values, a very common format in data science.

So, let's get started by importing the IBM Employee Attrition dataset. For that one, I 'import pandas as pd', and we're going to import a library called 'pathlib'. And from 'pathlib', we're going to 'import Path'. This is a very cool thing to do things in Python. I mean, you can, don't do that, and just put, there's a string here. But the cool thing is that this will work in Windows, Mac, and Linux, and you don't have to change the code to be able to run in this operating system. So, this is just to create a 'Path', where I'm storing my data.

I'm storing my data in a folder called 'data', and the name of the dataset is 'employee_attrition.csv'. I have my 'file_path', stored in the 'file_path' variable. I will use pandas function 'read_csv' to read my data, and I'm going to store that in the 'df' DataFrame. This is a variable called 'df'. It's a DataFrame, which is a type of data pandas created with columns and rows, like a spreadsheet for Python.

To run it again, we'll use Ctrl-Enter, and now, we run it. If you want to look at the top five rows of our dataset, we can use the function 'head( )'. The function 'head( )' will show us by default the first five rows of our dataset. Let's run it. As you can see here, we have several columns, 26 to be specific. And we're shown here only five-row because this is what we set with 'head'. We can change this by passing the argument inside of 'head' and saying, I want to see '10' columns, and you can see more, or you can see just '2' columns, and that's it.

Again, but by default, we have five. We have 26 columns, but a lot of them are not shown, because pandas decided that if it showed every one of the columns, it was going to look not that good. So, this is a default behavior for pandas, but you can change it, and we'll do it in the future. Next, let's export our dataset as a '.csv'. Again, this may sound weird at the moment, because we're importing some data, we're showing it, and we're exporting it again.

But in the future, it will be important, because we may change the data, and if we want to have, and want to store that information in our computer, we will need to have it in a CSV or in JSON or a different format. A big difference between the 'read_csv' and the 'to_csv' function that we're going to use here is that the 'read_csv' function is a method for the pandas module, but the 'to_csv' is a method for the DataFrame.

So, you will need to pass this as a function of the DataFrame, and in here as a method or of the module, pandas. What you have to pass into CSV is a path where you're going to store the data, and I'm going to store that in the as the 'employee_attrition3.csv', and it going to be in the 'data' folder. You do this, and now you have it. You can check that you have your information because if you go to home, you go to data, you will be seen both of the data.

Also recap, in this notebook, we learned how to import data with pandas. We learned how to use 'pathlib', a very cool library that can help us define a 'Path' that will work for Mac, Linux, and Windows, and we learned how to use the 'head( )' function to view the first five rows of our dataset. And finally, we also learned how to export the data we have as a '.csv'.

****

Video 2: Introduction to Pandas Objects Part 1: Series (5:18)

Welcome to video two. In this video, we'll be covering several Python and pandas objects you need to understand to be able to work with data with the library pandas. We will start by defining some objects in pandas, and we will go into detail to Series, DataFrames, and then some common functionality between the DataFrames and Series. So, the first two and most important objects we have in pandas to store and manipulate data are a Series and a DataFrame.

There are more types of objects in the library, but in general, you will be always using the Series and the DataFrame. Here's a good definition about the structures of Series and DataFrames, and it says that "The best way to think about a pandas data structure is as flexible container for lower dimensional data. For example, a DataFrame is a container for Series, and that Series is a container for scalars.

We would like to be able to insert and remove objects from these containers in a dictionary-like fashion." So, you can go here to this link if you want more information about that. And we get started by importing 'pandas as pd'. This is default, this is not something you have to do, but it is very recommended because this is the way you're going to be seeing every code in this course and in the web. Also, we will be importing 'numpy as np'.

It's another default we have, and 'from pathlib import Path' to be able to read data from our computer. So, as I mentioned before, a Series is a pandas object that stores one-dimensional data. Normally, we store in a Series one-dimensional arrays or of index data. Well, this is going to be important, pandas DataFrame are indexed, and this will be important in later videos and when you want to manipulate the information you have both in a DataFrame and in a Series. Let me get started by creating a Series, and we can do that in several ways.

One of the simplest way of doing that is by passing a 'list', a normal Python 'list'. So, the data variable will be creating a panda Series. For that, I'm going to be using 'pd.Series', and I'm going to pass a normal 'list'. Then I'm going to be creating a Series by passing to 'pd.Series' a numpy array. As you can see, it is not the same in the creation because for once, we need to pass a 'list', and the other one's going to pass an array, but it's going to look almost the same in the end.

Let me print it here. I'm going to print it, and as you can see here, we have a panda Series that contains a 'float'. So, the 'dtype' here is the type of the Series. So, this is a 'float' type of Series. As you can see here, we have float64 that is not a common Python type, and this is before the people at pandas are using the types for NumPy, and a lot of the types in NumPy comes from C and C++. Right now we have an indexed set of information that's the definition of a Series, but we can give it a name.

We can transform the way we index the Series, for that we're going to provide an 'index'. To do it, I'm going to be passing here a 'list' of letters from 'a' to 'g'. It's important if you want to do it in this way, you need to pass as much letters as data points you have. So, we have

'10, 20, 30, 40, 50, 60, and 70'. And 'my_index', it's going to be a 'list' containing this information. I mean, we are doing this because when we pass a string into the 'list', we're casting the string, This will be transformed into ' ['a' , 'b' ,...] '.

After we create a Series, the difference between this code and the above code is that we have to pass the 'index', and the 'index' will be the variable 'my_index'. And we run this, so as you can see here, in the beginning, we have, by default, from '0' to n minus one numbers. And now, we have from 'a' to 'g' because we defined this was going to be our index.

We can also do this by using a dictionary, and we can specify the values in the index in a more understandable structure. Let me run this, and as you can see, the keys in the dictionary are transformed into the index, and the values in the dictionary are transformed into the actual content of the Series.

****

Video 3: Introduction to Pandas Objects Part 2: DataFrames (3:52)

Let's now go to DataFrames. While Series are good to store one-dimensional arrays of information, a DataFrame is a combination of Series in one object. It can be defined as a DataFrame. So, you need to store in a DataFrame. I mean, it's useful to store two-dimensional types of information or more dimension. You can think of this as a two-dimensional array being an ordered sequence of aligned one-dimensional columns. So, this is the same as saying that a DataFrame is a sequence of aligned Series.

Here, the word "align" means that they have the same index. They share the same index. Let's get started by creating a DataFrame with a list of lists, where each inner list is a row or of a scalar data values. So, I have a list of lists. And in here, you have, 'Paul', 'Kristen', and 'Stanley'. This will be transformed later into a column, '32', '22', and '9' will be another column. 'Software Engineer', 'Data scientist', and 'Dog' will be another column.

Well, let me run this, and as you can see, I don't have a name for my columns because I just passed the information for the DataFrame and not the columns. If you want to pass the name of the columns when you are creating the DataFrame with the function 'pd.Dataframe( )', you'd not only pass the 'rows', you will always be passing the 'columns'. These 'columns' here are the names of my 'column'.

This is a list with strings, and each one of these strings will be transformed into the name of the column. And after that, I'm also passing the 'index'. If you don't do this by default, it will be 0, 1, and 2 in this case. Well, I'm just going to change that to 'P1', 'P2', and 'A1'. As you can see here, we have named our columns, and we have changed the default index. Again, you can leverage the dictionary type to conveniently specify column labels, and the associated data in that column.

For that, I'm going to be creating all of the things we did there, aside from the index in one dictionary. So, the dictionary will be for the keys of a dictionary will be transforming to the 'Names' of the columns, and the lists on the right side are the values, those will be the actual content of that column. As you can see here, this is an easier way of doing the same. You can change the column and index labels after creation by setting them as attributes of DataFrame objects.

In here, I'm going to be creating the 'DataFrame' by passing the dictionary. Then I'm going to be passing the names of the columns. And then, I'm going to be transforming the index.

But how I'm doing it here? I'm just saying 'df.columns', and 'df.index', and this is I'm overriding the attribute columns.

If I say 'df.columns', I will be seeing the columns, so I'm accessing the name of the columns. If I say 'df.index', I'm accessing the index. So, when I pass something to that, I can change the default. Well, let me do it here. I'm going to run it. And as you can see here, I mean, I just created some weird names for the columns, but this works.

****

Video 4: Introduction to Pandas Objects Part 3: Common Functionality (7:29)

Normally, the way you use pandas is that you will be passing through Series and DataFrames, and you have some common functionality between both of them. We will be focused later on that in the course because these are more complex, but keep in mind that we have this symmetry between a Series and a DataFrames in pandas. So, in this case, this is what we're going to do. We're going to create a Series called 'ser', and we're going to create a DataFrame called 'df'. The Series is going to be just a 'list' of information.

It's going to be just letters from 'a' to 'p', and the DataFrame will be the same as before when I read the 'employee_attrition' IBM dataset. Let's just do this. With Series and for pandas, we saw that you could use the 'head( )' function to view the first five rows, or how many rows you wanted for the DataFrame. That also works for Series. So, this is the common functionality I was talking about. There are some functions that will work for both DataFrame and for the Series. So, 'ser.head( )' will give me the five first rows of our Series.

You also have one function that's very interesting, it's called 'tail( )'. And 'tail' is just, it's almost the same as 'head', but it will be showing the last five rows by default. You can also give an argument of how many rows you want to see. So, I'm going to put '3', so I'm going to be seeing the last three rows of my DataFrame 'df', and here you have it. You can select values from a series, and also for a DataFrame, by using the '[ ]', as you will do if there will be a Python list.

So, let me say that I want to select the second index, so it's going to be 1, 2, 3. So, it's going to be 0, 1, 2. It's going to be the third element of my Series. Sorry. And you can do that by just passing the index like you will be doing in a list, and you could also do this slicing as the way we learned for list, the same. And say '2:6', and it will be the same form the second index to the sixth element, without counting the sixth one. So, it's going to be from '2' to '5'. Then we run it, and the first one is c because you can check here that 0, 1, 2.

So, the third element is the number 2 index is going to be the c letter, and from '2' to '6', without counting the c, is from 2 to 5. So, we have c, d, e, f. You can do similar things with the DataFrame, but something interesting will happen here. If you just select like this, let's say I want to select the 'Age' column from the DataFrame. Because c here is the first column, we have. You can do that by passing the name of the column between the '[ ]'. But take a look at the 'type'. The 'type' here of 'age' is a Series.

So, when you are slicing a simple, or selecting a column with this type of formatting when you pass '[ ]' and the name of the column, what you actually will be getting is a Series. And you can show that, because when we call 'age', what we're seeing here is a Series, and not we're not seeing a DataFrame. Again, you can get 'rows'; I mean this 'rows' variable will be selected from the second row to the sixth row, without including the sixth row, again, for the DataFrame 'df'. And if you 'print' that, you will be getting a DataFrame, but this is something different.

If you're selecting like this, you're getting a Series. If you're selecting rows like this, you're getting a DataFrame. There's two quick functions I want to show. They're very helpful to understand the way our data is structured. The first one is called 'shape', and 'shape', what we'll do is we'll tell you the dimensions of your information. So, if you apply 'shape' to a Series, you'll be saying, so by default, 'shape' will be returning a tuple, where the first element of the tuple will be how many rows you have, and the right part how many columns you have because a Series is a one-dimensional type of information we'll only have in rows.

So, you have 16 rows in the Series. If you apply that to the DataFrame, you will be seeing that you have 1470 rows and 26 columns. So, by default, the first element of a tuple from 'shape' is rows, second element are the columns. I'm sorry. Finally, let's look at the 'describe( )' function. The 'describe( )' function is very interesting, and it will give you some statistics about your data. 'describe( )', it's a very interesting way of getting our first glance of how your data is a structure in the means of mathematics.

Let's say you want to get the mean, the minimum value, the maximum value, the percentiles, all of that will be given to you by 'describe( )'. If we run 'describe( )' for a Series, we'll be getting, by default, the count, how many are there, how many are unique, the top, and the frequency. If we do that for the DataFrame, we'll get much more information. We will be getting the count, mean, standard deviation, minimum, percentiles, and maximum for each one of the columns.

This is very good information that's talking about the way our data is shaped and is structured. As a recap, we learned several things about Python objects. We learned that the most important objects we have in Python are the Series and the DataFrame. We learned how to create Series. We learned how to create DataFrames. After that, we learned how to shape and transform our DataFrames by accessing the different attributes they have. And finally, we learned some common functionality between DataFrames and a Series.

Where we explored the functions 'head( )', 'tail( )', we learned how to select, and slice our Series, and DataFrames. And finally, we saw two interesting things. The one is the attribute 'shape', that will tell you about the shape of your information, meaning how many rows and columns you have. And finally, the 'describe( )' function, that will tell that will give you some statistics about your data.

****

Video 5: Indexing and Selecting Data Part 1 (7:42)

Let's go more in-depth now about how to index and select data from DataFrames. Again, if you're familiar with SQL or Excel spreadsheet, you know that you will want to select and get information in various ways. If you want to have some columns there, or some rows here, or maybe a combination of both. In the last video, we saw very briefly how we can do that for a Series and DataFrames. And in this video, we'll be covering the basic functions we have, and the basic ways we have to get informations as columns or rows in DataFrames.

You've seen a lot of different functions, but the most important ones are 'loc[ ]' and 'iloc[ ]'. So, the outline will be when we try selecting a data for DataFrames. We're going to be choosing our data, that's going to be the 'employee_attrition' DataFrame. And we're going to be selecting columns with the '[ ]'. We're going to be selecting data with the function 'loc[ ]', and we're going to select some data with the function 'iloc[ ]', and I'm going to give some comments about 'iloc[ ]'. So, let's get started with our data.

So, let's just read the information again by importing 'pandas' and 'pathlib', and then let's see our data very quickly with 'head( )'. We've done this before, so we're okay with that, and let's try to select information with just '[ ]'. As I mentioned in the last video, if you use this simple '[ ]', for selecting something for our DataFrame, you'll be getting a Series, and you can check that here.

So, I'm getting only the 'Age' column as a Series in the variable 'age'. If you pass a 'list' and you can select several and multiple columns, and if you pass that with '[[ ]]', something interesting will happen, and that is how you will be getting a DataFrame. So, '[ ]', you'll be getting a pandas Series, '[[ ]]', you'll be getting a pandas DataFrame. You can get creative with this comprehensions to select columns dynamically. And in this case here, I'm just going to list comprehension, select all of the columns that start with 'Years'.

So, if a column starts with the word 'Years', I'm going to be selecting that, and for that, I'm going to be doing '[[col for col in df.columns if col.startswith('Years')]]'. This is a function we have in pandas columns to just see if the column starts with several or one letter. So, when I run this, as you can see here, I have all of the columns that starts with the word 'Years'. If you want to do this in a more formal way, pandas has a function, it's actually an attribute called 'loc[ ]', and this provides a label-based access of your information and data.

In most cases, 'rows' are labeled with the four row index that start with 0, and will increment by 1 for each one of the rows, and the columns are labeled with descriptive strings, like the names of the column. So, if we're specifying a row, we'll be using the interior value, and when we will be specifying a column, we'll be using this string like distance from home. You will always access 'loc' with '[ ]'; however, it accepts different types of inputs, and will return different structured responses accordingly.

Let's get started with individual row selection with 'loc[row]'. So, in this case, we'll simply pass an integer as a row label, and we'll be selecting a row of our information. With that, I say 'df.loc', and I'm going to select the row number '17' here. As you can see here, this will give me a pandas Series. This is kind of the same as just we saw with only the numbers. But, in this case, if you remember back then when we just used the '[ ]', we got a pandas DataFrame. Now, we're getting a pandas Series.

So, now you can also select rows with '[start : finish]'. Like you will do in a normal list. In this case, only one interesting thing is that the both values of the slice will be included, unlike when we use list, that last index was not included in the selection. So, if I say this, and we'll be selecting from the row number '12' to the row number '15', with both '12' and '15' in the selection. We run this. As you can see here, this will give me a pandas DataFrame, not a pandas Series.

You can select a single value with the idea 'loc[ ]', and then you pass a 'row' and a 'column'. So, if you have the second parameter in here as a string, pandas will be assuming that you want to get the row number '17' out of the column 'Age', and only the column 'Age'. So, you will be getting here the 'age_ of_employee' in the '_17' position. For this, if you run this, it will give you just the value. Not a pandas DataFrame, not a pandas Series, it will be normally be giving you a type 'numpy' object.

So, in this case, we have an 'int64' 'numpy' type object. You can select multiple rows, and multiple columns with this interesting way, where in the left part, you pass 'rows'. And remember that in this DataFrame, the 'rows' are indexed by numbers. So, from 0 to a 1,000 and something more, and you have the columns that're named. So, they're strings. But in the first part, you will be getting this format here is saying I want to get from the row '25' to the row '30', including both of them, and I want to get the columns 'Age' and the column 'Environment Satisfaction'.

So, if I run this, something interesting is that this column is also saying here from "Age': 'Environmental Satisfaction", so all of the columns in that range will be included in the slice. Something interesting that you can do here is that you can also pass not a single. I mean, in this case, we used a column, because that will be getting us from '25:30', including all of them, and from "Age': 'Environmental Satisfaction", including all of them. But you can select specific rows and columns by just passing a list with the items.

So, if you pass a list of indexes here, it will give me the row number '13', the row number '75', '22', and '11'. And take a look here, and you have the ordering to this that you decided there. So, pandas will not order your DataFrame if you don't say that, and I'm only getting here the columns 'Age', 'Attrition', and 'HourlyRate'.

****

Video 6: Indexing and Selecting Data Part 2 (7:01)

Something very interested is that 'loc[ ]' can also accept a different type of input. If you pass like a list-like collection of boolean values, it'll return a DataFrame including rows that can respond to True values in the input list. So, let me create a simple DataFrame here with the columns 'Name' and 'Age'. 'Name' will be having the values '['Joe', 'Alice', 'Steve',' and "Jennie']'. And 'Age' will be having the values '[33, 39, 22', and '42]'. So, what I'm doing here is, I'm creating this 'loc[ ]' thing here. I'm passing '[True, True, False, True]' to just get the people in my DataFrame that are 'over_30'.

If I run this, what's going to happen here is that, this '[True, True, False, True]' is just excluding the third row, because in that third row, you have that information of a person who's less than '30' years old. So, I mean, this was depending on you to know every one of the rows very well. So, if you don't know that, I mean, you will not be constructing a list of boolean values manually. So, it will be much more simple if we can create a like a boolean conditional function that will give us everyone who's above '30'.

So, what we do here in this case is that, we're going to say 'example.Age', 'example' is the name of our DataFrame, and the operation is '> 30'. So, as you can see here, again, if you 'print' 'is_over_30', right now, this is giving you all of this same operation here, where you created by hand '[True, True, False, True]'. It was given to you by pandas with this function here. So, if we want to apply that, and I mean this is getting just the '[True, True, False, True]'. If you want to get the information, you have to put that inside of 'loc[ ]'.

So, 'over_30', you say 'example.loc', and you pass a condition inside of that, but it will only give you information of people which are older than 30 years old. That's it. But this is much better than this. Because in this case we hard-coded everyone of the rows. But if we don't know that, and what if we have 2,000 rows? We won't want to be doing this by hand. So, this conditional function is much better than that. Again, this will become much more powerful as we're dealing with larger datasets. So, in this case, we're going to do several things.

We're going to be having a DataFrame called 'filtered', it's going to contain people older than '30' years old. And after that, we're going to have a different DataFrame with only '40' year-olds or lower. And in the end, we're going to just put people that 'Travel_Rarely', here in the column here. So, I'm going to run this right now. I'm using this for the DataFrame 'df', okay? Okay, so this is not for the 'example' DataFrame. I'm using the 'employee_attrition'.

So, I'm only getting people older than 30, younger than 40, and which the variable 'BusinessTravel == 'Travel_Rarely''. So, this is what we have here. You can check here that the DataFrame 'df', contains 1470 rows. And now, our new DataFrame only contained 423 rows. This is because in the filter process we get rid of a lot of information. So, people that are not in the 30, 40 range of age are discarded. And people which 'BusinessTravel' is not the same as "Travel_Rarely" are not going to be there as well.

Alongside with the function 'loc[ ]', we have the 'iloc[ ]' attribute. This is a very analogous way to 'loc[ ]'; however it operate strictly on positional integer values for locating rows and columns. It's very important to note that we use integer values for row previously when we were using 'loc[ ]'; however that was only to match the data type of a row index in the DataFrame. We will use a string value for this example, to demonstrate the difference between 'loc' and 'iloc'.

So, let me create again the DataFrame. I'm going to name this 'df2', going to have two columns, and the index is going to be '('abcd')'. So, it works there. So, if I try to use 'loc' with '[0, 1,' and '2]', this will 'throw' a 'KeyError, because those values aren’t present in the row index'. Now, the index says, the index is not an integer. It's actually a string. So, I can use 'loc[ ]' like this, and it will work. However with 'iloc[ ]', we can always use integers in order to retrieve the rows and the columns by their position.

So, if you try to do this with changing 'iloc[ ]' to 'loc[ ]' it's not going to work, but if you want to maintain working with integers, you can use 'iloc[ ]', and you can select things by their integer position in the DataFrame. But this will work perfectly. As a recap, we saw different ways of indexing and selecting DataFrames. We read our information, and we tried different ways of selecting and slicing data. First we started with '[ ]'. Then we went into the very useful 'loc[ ]' function. We learned how to select only rows, select several rows, select a single value, and select multiple rows and columns with 'loc[ ]'.

Then we start learning how to select rows with a condition, and we passed that with a boolean operation inside of 'loc[ ]'. And in that way, we can filter the DataFrame we have. In the last part, we learn how to use 'iloc[ ]'. And 'iloc[ ]' is very useful when you want to keep using integers instead of the actual name of the index, if for some reason the index is not an integer, you will not be able to select and slice the information with 'loc[ ]', but with 'iloc[ ]', you will be able to do it.

****

Video 7: Editing DataFrames Part 1: Setting Columns (7:07)

In this video, we'll be learning how to edit data in DataFrames. DataFrames are editable. That means that you can change the information that they already have. So, now we'll be learning to do three important things; one, how to set columns; two, how to transform columns; and, finally, how to set data with the 'loc[ ]' attribute. So, let's start by importing 'pandas' and 'numpy', also 'pathlib' to be able to read our information, and let's just see the information we have inside of that.

Again, we're using the 'employee_ attribution' DataFrame, that will guide us through the process of learning all of this. And remember that through all of the examples, you will see code sections that will be starting with '.copy( )', and the rest of the example will then typically work with a 'df' variable. This 'copy( )' thing is to copy the contents of our DataFrame to a local variable so that the example don't interfere with each other. So, this copy thing we want to do, if you want to copy the information, it's like you copy the lists.

But now we're copying DataFrames. So, when we do some examples, they don't interfere. So, don't worry that much about that. It's just for the sake of the demonstration. Let's start with setting columns. All columns in the DataFrame can be accessed with the ' . ' or the '[ ]' like 'df.Age' or 'df['Age']', and they can be assigned in the same way. DataFrames can be thought of as collection of Series, as we've said in other videos. And the pandas libraries support adding and replacing in the DataFrame.

So, let's say we want to add a new column. How would we do that? Again, when you see this copy thing, I'm not going to talk about this, again. It's just to copy the DataFrame, so we don't interfere with the examples. So, I'm going to create a new variable with a 'range'. In this 'range' here, we'll be generating data from '(0, 1470)'. Remember that this '1470' will not be included, and what I will be doing is that I'm going to be adding a 'new_column'. How am I going to do that? I'm going to say 'df', and inside of the '[ ]', I'm going to say '['new_column']'.

So, what this will do is that it will transform my DataFrame 'df' to have a new column in the last part of it called 'new_column', and the information will be, all of the range of numbers from '(0, 1470)'. So, as you can see, and I mentioned the 'new_column' was placed in the end of the DataFrame, in the right edge of the DataFrame, and this is by default what a 'new_column' creation does. You can also replace a DataFrame, and let's say we want to create a new one that contains the same information as the other one.

So, I want to create a new column called 'Gender' that is based on the 'Age' column. In this case, this is not a good idea because 'Age' is going to be a number, and 'Gender' should be male and female. So, this is going to be a weird column, but this is just again for the sake of a demonstration. So, if you say 'df.Gender', this is going to be the same as saying 'df['Gender']'. I'm going to be creating a new column. But, in this case, instead of passing new data, I'm just copying the information from the 'Age' column.

Again, as you can see, now we have the Gender. Let's say here Gender; it's not directly seen here or maybe, okay. Here it is. So, Gender, what this is doing is replacing the information. So, let's go to the original DataFrame. So, Gender was saying Female and Male. So, for this demonstration, we already have the Gender column, but now what we did was replace the information from Gender, and we put their information from Age. So, now Age is there twice.

As you can see here, Age is here and in here, sorry, so Age is here, and Gender is not showing Female and Male. It's just showing the same information as Gender because that's what we did. You can also remove columns. This may be very important because, when you're trying to do machine learning and we'll see that later in the course, you don't want to use every column that you have in the DataFrame. We will discover how to choose the best columns. But, for this moment, sometimes you want to explicitly remove a column.

And, to do that, we are going to use the function 'drop( )'. This 'drop( )' function will return a new copy of the DataFrame with the 'drop( )' entity. It will not mutate the original DataFrame. So, this is a different type of behavior because the way we were doing things so far is that, when we copy the information, or we added a new column, it was changing the actual DataFrame. Now, it's giving me a new copy.

So, for me to get that information inside of 'df' and if I don't want to have a different name for that, I can say 'df = df.drop', and in this case, I'm going to be dropping 'Gender' and 'Age'. But, if I don't put this 'df' here, when I see 'df', it's not going to have that transformation. So, you want to do that. You are on this and you don't see Age or Gender in the DataFrame. You can always use 'loc[ ]' and 'iloc[ ]' in order to select information and get a new look of the data you have.

For instance, let's say you want to drop all but the first four columns, and a good way on doing that is by just selecting the information with 'iloc[ ]'. I mean, if you don't know the names of a column or you don't want to type them, and you want to use the integer representation or number of each one of the columns, you can use 'iloc[ ]', and in the first part of 'iloc[ ]' we're passing rows.

So, if we put nothing ':' nothing, that means we're getting all the rows. And in the right part, we're getting from the column number '0' to the column number '4' with '0' and '4' in the range, and I'm just getting the information between those columns. So, this is what we're getting, and we're getting here, all of this information, all of these columns, and this is a very interesting thing to do if you don't want to select columns by their names.

****

Video 8: Editing DataFrames Part 2: Transforming Columns (4:24)

You can also transform columns. And sometimes it's something you want to do, perhaps you want to standardize the 'Gender' column in this example. And instead of 'Male' and 'Female', we just want 'm' and 'f'. So the question here is, how we can change the data in that column to match what we want? One of the ways of doing this is with the '.map( )' operator, Map is a universal concept in programming, and what it does is that it involves taking a collection of data as an input, applying a function there to each one of the elements, and returning all of the return values of that function in a new collection.

In our case, we would like to create a function that can return the values in the Gender column to either 'm' or 'f' and return that into a new column of data. A '.map( )' function on the column series will do that for us. So, I'm going to create here, with the 'lambda' operator, a new column called 'new_gender'. So, for that, I'm going to get the 'Gender' column, I'm going to be applying the 'map( )', and the 'lambda' function that we'll be using is this one here. So, 'lambda g:', that says 'g' will be the content of each one of the rows.

So, it's going to put ' 'f' ' if the row says 'Female', or 'm' otherwise, and for that, this is going to give us a Series, and what you want to do now is to assign a new column to the Gender column, as we did before. So, 'df.Gender = new_gender', and if we see now have, you can see that here Gender has changed, and now it contains the standardized information for female and male. You can also do arithmetic operations with columns. Let's take a look at another example for transforming information with this.

Let's say we want to identify people who have worked in their entire career at this company. If their "TotalWorkingYears" is equal to their "YearsAtCompany" value, then, I mean, we can take the assumption that that person has only worked in this company. And then, we want the value in the Lifer column to be equal to true. So, I'm going to be creating a new column called 'Lifer' that is True if the person has worked in the company for all their career and False otherwise. Again, this is an assumption.

I mean, we can have some mistaken information in these columns, but we're going to assume that if the "TotalWorkingYears" equal to "YearsAtCompany", that they have worked all their life in the company. But to do that, I'm going to be passing a boolean operator here, saying that 'TotalWorkingYears' has to match 'YearsAtCompany', so when we do that, we will be having a series of boolean values, True and False, and I'm going to assign that into a new variable called '[Lifer]'.

And now, in the last part of our DataFrame, we will have False and True. True meaning that this person here will have worked their entire career in life in this company. Again, we can do more things in here. Let's say we want to create some new variables by summing or multiplying, or difference we have already in our DataFrame. In this case, I'm going to be creating a new variable called '['unfair_compensation']', that's going to be the multiplication of the columns 'DailyRate' and 'JobLevel'.

And I'm going to be creating a new column called '['BusinessTravel+Gender']', that is going to be the string concatenation of 'BusinessTravel' and 'Gender'. So, I'm going to run this, and as you can see here, we have this new columns now, unfair_compensation, and BusinessTravel+Gender. That's this concatenation. So, these are some operations you can do, you can think of whatever you want to do, and you can apply maps( ), or you can apply the simple operations to them.

****

Video 9: Editing DataFrames Part 3: Setting Data with loc[] (2:04)

Now, for the last part of the notebook, you can also set data with 'loc[ ]'. Remember how useful was the 'loc[ ]' attribute for reading our DataFrame? So, now it also works and also it's useful to setting new data in the DataFrame. First, let's build a sample DataFrame to demonstrate how this works. So, I'm going to create a new DataFrame. I'm going to call that 'sdf'. And the 'data' is going to be '[[False for j in columns] for i in range(0, 10)]', so it's going to be a list of "False" for our DataFrame, and the names is going to be just 'abcd' and 'e' and 'f'.

So, this is what we did. We just created a DataFrame of False values with name in the columns a to f, and now, let's say I'm going to print this 'loc[0, 'a']'. '[0, 'a']' is this one right here. It is False, and what if we want to change that False into True? So, you can do that with 'loc[ ]' saying, I want to change the value inside of the '0' row. So, the first row in the column "a", and I want that to be equal to 'True'. So, you say this, and this operation will transform your DataFrame.

So, this is going to take advantage of the readable nature of DataFrames because it's going to change that. As you can see here, it was False, and now it is True. Let's say we want to have the whole third row as True. We can state that 'loc[3]' is, remember this is selecting rows, and this is just changing the whole third row into all Trues. We can also say, I want to change for the row number '[7' in the columns '['a', 'c', 'f']]', and I want all of them to be 'True'. So, I do this, and I'm changing it. In the 7 column in a, in c, and in f, we have True.

Instead of passing a single value to be assigned, you can pass data that matches the shape of your query to set the exact data. So, if you pass a list that contains one, two, three, four, five, six elements inside of it, you can change the whole row by saying I want to have True, False, True, False, True, False; and this should happen here. As you can see right now in row number 9, we have all False. And now, I'm selecting row number '9', and I'm changing that to '[True, False, True, False, True, False]'. And, as you can see here, it worked perfectly.

Finally, let's 'use the slice operation to select rows 3 and 4, and columns d, e and f'. So, what I'm doing here is I'm selecting the rows '3' and '4', and I'm selecting all the columns from "d' : 'f", so that means 'd', 'e', and 'f'. And I'm going to be changing that to '[['a', 'a', 'a'], ['b', 'b', 'b']]'. So, what it did is that it selected a portion of my DataFrame, and it helped me change that.

Take a look that we pass a list of lists here, and that's to be able to change all of the items inside of the operation, inside of a row and inside of a column. As a recap, in this notebook, we learned how to set information in our DataFrames. We learned how to set new columns, how to add new columns, how to replace content in existing columns. We learned also how to remove columns with a 'drop( )' operator.

We then find a way to select columns by only their index and ID, and we use 'iloc[ ]' for that. Then we learned how to transform columns to perform different operations on there, and we use the '.map()' operator to do that. And we also learned how to apply arithmetic operations to our DataFrame with the use of simple multiplication sums, and subtractions for that. Finally, we learned how to set data with 'loc[ ]' and that's very useful if you want to set a specific column, a specific row or specific value contained in a row and column.

****

Video 10: Combining DataFrames (9:40)

In this video, we will show you how to combine DataFrames. In particular, we will explain to you how to perform the union and the join. Before doing that, let's think about why we want to combine DataFrames. Well, this may be very useful if, for some reason you have data that is collected in different DataFrames, and you want to join and combine this data in order to perform, perhaps, a more complete analysis. Let's observe a simple example. I begin by importing pandas as usual. And here, I have defined three different data frames. DF, the first names one, DF, first names two, and DF last names that contains the student ID, the first name, and the last name of some students.

Let's have a look at these DataFrames. DF first names one contains the student ID and the first name of Daniel and Alfredo. DF first name two contains the names and the student ID of Gertrude and Ying, and the last name DataFrame contains the student ID and the last name of some of those students. One of the ways that we can use in order to combine DataFrame is the union. The union essentially allows you to combine DataFrames on top of each other in the following way. Suppose that I wanted to combine the student first name one and first name two DataFrames.

I could do so by defining a new DataFrame that I would call first names and then define these to be the union of first name one and first name two by using the function concat in the following way. I would pass to concat the name of the two data frames that I want to merge as a list. So, in this case, it would be DF first name one and DF first name two. Next, because I wanted to combine these data frames on top of each other, I would simply reset the index. So, I would just say index drop equals true. What this reset index function does, it essentially resets the index in both data frame, so that the indexes are not repeated, in case they were. Let's have a look at the DataFrame that we have just obtained, and you see that what I've obtained, it's a DataFrame that contains a student ID and first name, and all the information of the students that I have.

Also, observe that the index in the new DataFrame starts from zero and ends in three. You can also combine DataFrames by merging them. We can do by using the merge function in Python. This function takes four arguments. The first DataFrame, which is the one that you would like on the left of your resulting DataFrame. The second DataFrame, which is the one you would like on the right of your resulting DataFrame, and then it takes two additional arguments. How, it's a parameter that defines what kind of merge that you want to make, and on, it's a parameter that tells you which column or indices you want to join on. Let's look at an example.

The first type of merge that I want to talk about, it's the inner join. In the inner join, we will lose all the rows that don't have a match in the other DataFrames' key column. If you look at this picture, you see that I have a left-side DataFrame and the right-side DataFrame, and if I merge them using the ID column, then only the rows with index three or four will be kept in the resulting DataFrame.

Let's have a look. Here I print again, the DF first names DataFrame and the DF last names DataFrame. I want to join them by using the command merge in pandas by passing the DF for first names DataFrames. The DF last name DataFrame, and I want to merge them on the student ID column. And I want to perform an inner join. Therefore, I set the parameter, how equals the inner. You see that the resulting DataFrame has only three rows, because only the student with these student IDs were present in both DataFrames. The other type of merge that I want to talk about, it's the outer join.

The outer join works in a similar way to the inner join, except that now all the pair-wise combination of rows from both DataFrames are kept. If you observe this picture, you see that in the resulting DataFrame, all the rows from the left side and the right side DataFrames are kept, and the ones that didn't have a value get filled with nans. Again, we can look at an example using pandas.

So, I want to join, again DF first names, DF last names. I'm going to use again, the column student ID as my index column, and then I want to perform now a outer join. So, I set the parameter how equal to outer. You see here that the resulting DataFrame has all the possible combinations of student ID, and some of the entries in this resulting DataFrame as are filled with nans. The third type of join we're going to see today, it's the left join. The left join, again, it's really similar to the inner join and the outer joins, except that now all the rows of the left side DataFrames will be kept, the rows that don't have a value in the right side DataFrame will be filled with nans. The syntax in pandas is very similar to the one of the inner and outer join.

So, we can say PD. merge, and then DF first names, DF last names. I'm going to use, again, student ID as my index column, and then, I want to set the argument how equals to left. You see that the resulting DataFrames as all the rows that were present on the DF first names DataFrames, and there is one value in the column last name that is filled with nan. Finally, we are going to look to the right join.

The right join, it's very similar to the left join, but now, all the rows on the right side DataFrame will be kept, and if the left side DataFrames has some missing values for the same index, those values will be filled with nans. The syntax, again, it's very similar to the one we've already seen. So, we can just say PD.merge, DF first names, DF last names. I'm going to use, again, student ID as my index column, and then, I can set the argument how equals to right. You see now that all the entries that were in DF last names are filled, and there is one nan value in the column first.

In summary, in this video, we saw that we can combine DataFrames in two different ways. We can concatenate them across rows or column by using the function concat, or we can use merge to combine data on columns or indices. We can perform different types of merge, the inner, outer, left, and right one.

****

Video 11: Reshaping DataFrames (10:17)

In this video, we'll show you how to reshape data in dataframe using Pandas. In particular, we will show you how to pivot and melt dataframe. Let's begin by pivoting dataframe. The pivot function is used to reshape data in a given dataframe by giving an index and column values. The pivot function takes 3 parameters. An index, which tells us which columns be used in order to order the new rows in our new dataframe, the columns which tells us which columns should be used in order to create the new column in our dataframe, and the values, which tells us which column should be used in order to fill the values in our new dataframe.

Let's begin. In this code cell, we have imported Pandas as usual and we have to find the data frame df that contains the year, the average, the student name, and the age of some students. Suppose now that I wanted to reshape this dataframe by using the function pivot. What I would do is say "df" and then I would apply to it the function pivot. I would specify the index of my new dataframe to be for example, year. I would specify the columns to be equal to for example, average. And I would specify the values to be equal to, for example, student name.

Let's see what I obtain when I run this code cell. You see that I obtain a dataframe that has a very different structure from the original one. The index in the new data frame is year and because year in the original dataframe only had entries 1 and 2, you see that the new data frame has only 2 rows, 1 and 2. Because we used the average column in order to create the new columns in my dataframe and the average column had only entries A, B, and C in the original dataframe, you see that my new dataframe has again 3 columns, A, B, and C. And finally, observe that because we used student name to fill the new dataframe, the student names are now positioned in the corresponding order in the new dataframe.

For example, in the original dataframe, John was in year one and had an A average and therefore in the new dataframe, John is positioned in the entry that corresponds to year one, column A. Let's look at another example. Here I'm again applying the function pivot to my dataframe. I want to use the same index so year. I want to use the same columns so I'm going to use average again. But now I'm going to fill the values with age. When I run this code cell, you see that I obtain a new dataframe. The structure of the index and the columns in this new dataframe is exactly the same as the one I obtained before but now the values are filled with the entries that were in the column age. Again, observe for example that Joe is 22 years old and is in year 2 and has an average of B.

Therefore, 22 is positioned in my new dataframe in the row that has index 2 and average B. Another way of reshaping dataframes in Pandas is my melting them. We can do so by using the Pandas function melt. This function helps us transforming a dataframe from a wide format to a long format and it takes 2 types of argument, the id_vars, which identifies the identifier variables and the value_vars, which defines the value variables that will be used in our new dataframe. Let's observe an example. Here we have defined a dataframe df2 that describes the job position, the salary, the start date, and the performance rating of some professions. Suppose now that I want to perform a melt on this dataframe.

Here is how it goes. I would say "df2.melt" and then I would specify which variables I want to use as id variables. In this case, I can choose them to be, for example, the job position. Next, I could specify which columns I want to use as value variables, so I would say "value_vars is equal to, for example, start date." Observe the data frame that I obtain. On the id variable, I have the job position. The variable column contains now that entry that are, sorry, contains as entries the name of the value variables that I specified before, so start date.

And in the column value, I have the value that were in the start date in the original dataframe. For example, the start date for this data scientist was 2017 and therefore the corresponding value in the new dataframe is filled with 2017 for a data scientist that started during that year. The function melt will also assume all columns are value_vars if they are included as id_vars. We can also decide to include more than one value_vars in our definition when we use the function melt.

Let's see an example. Here I am defining a new dataframe df2.melt that will be equal to the old dataframe to which I apply the function melt by specifying id_vars equals to job position. Value_vars will now be equal to- start date- and salary. And I can also decide to change the name of the variable and value column in my dataframe by specifying the following parameters. I could say "var_name is equal to Variable Column" And next, I could say "value_name is equal to Value Column". Now, let's see what happens when I print my new melted dataframe. Okay, you see that job position is again being used as identifier variable. In the variable column as entries, I have the name of the columns that I've specified in the melt function, so only start date and salary.

The value column is still with the entries from the column start date and salary from the original dataframe. As we would expect, the resulting dataframe has a number of rows that is equal to the number of rows of the original dataframe multiplied by the number of features included in value_vars. Here I have defined our next statement that it will print an okay message if the length, so if the number of rows of df2_melt is equal to the number of rows of the original dataframe times 2 because remember I only have 2 features here otherwise; it will print an error message. Great. So, in this video, we have learned that we can reshape data frames in 2 different ways. We can use the pivoting, or we can use the function melt.

****

Video 12: Grouping and Aggregating Data (8:06)

In this video, we'll show you how to group and aggregate data in Pandas. In particular, we will show you how to split data into groups and how to perform aggregation operations on your data. We begin, as usual by importing the necessary libraries so we import pandas as pd and for this exercise, we will use a dataset that contains some information about the NBA league. We read this dataset by using the read_csv function to obtain the data frame df. Finally, we use the command head to visualize the first file rows. You see now that this data frame contains the information about the name, the team, position, age, college, and salary of some NBA players. It might be convenient sometimes to split our data into groups based on some criteria. We can do so by using the function groupby in Pandas.

Let's try. Suppose, for example, that I wanted to split or group the data in my data frame based on which team each player belongs to. Then I would say team is df.groupby and then as argument I would pass the column that I want to use as criteria in order to split my data, so in this case, it would be team. Next, if I were interested to see what is the first value in each group that I've created, I can just do so by using the first attribute on the team data frame that I have obtained. You see now that what I obtain, it's a new data frame. The first column, it's labeled team and it has all the teams that are present in the NBA league and the corresponding attribute are the information of the first player that appeared in the original data frame and that belonged to that team.

We can also print some information about all the players that belong to a certain group. For example, if I wanted to print all the values contained in the Boston Celtic group, I would do so by saying "team.get_group" and then I would pass as argument to this function the team I want to extract, and so in this case, it would be Boston Celtics. You see now that what I obtain it's a data frame that contains all the information of those players that belong to the Boston Celtics. We can also use the groupby function to form groups based on multiple categories. For example, I could decide to create a data frame that grouped the players based on team and position.

So here I am defining a team_position data frame and next I will use the groupby function on my original data frame but now I would pass the labels team and position as a list to the function groupby like so. Next, I can decide to again print the first value in each group to have a look at what my resulting data frame looks like and so I would say team position and I would use again the function first and you see that what I obtain it's a data frame where first of all all the players are grouped based on their team, so we have the Atlanta Hawks first and the Washington Wizards last.

Next, the players are grouped based on the position they played and next we have all the remaining information for each player. Now that we've learned how to group the data in our data frame, we can also see how to perform some aggregation operations on it. This can be done in Pandas by using the function aggregate. It allows us to perform some aggregation operations across data. For example, with this function, we could compute the sum, the min or the max of the data contained in certain columns. Let's look at an example.

Suppose I was interested in finding the sum and maximum of all the numerical columns in my df dataframe. I would do so by typing df and then aggregate and then I would pass to this function a list containing the name of the operation that I want to compute on my columns, so in this case, it would be sum and then max. You see that what I obtain, it's a data frame that contains the same columns that were in my original data frame but only the ones that had a numerical type of data in it and the rows labeled sum and max because those are the operation that I wanted to compute with my aggregate functions. We can also decide to apply different aggregation functions across different columns.

For example, if I was interested in retrieving the max and min of the age and weight of the players but only the sum of the column salary, I could do so by passing to the function aggregate a dictionary that has keys, the name of the columns I want to perform the aggregation operations on, and as values, a list containing the name of the operations that I want to perform. Observe this code cell. Here, I want to compute only the max and the min on the columns age and weight and I only want to compute the sum on the column salary. When I run this code cell, what I obtain is a data frame that contains some NaN values and that is because I decided that I didn't want to compute the max and the min, for example, in the column salary.

In summary, in this video we saw that we can set the data in our data frame by using the function groupby and that we can group the data by selecting one or multiple columns. Next, we also saw that we can perform aggregation operation on our data frame by using the function aggregate conveniently get some statistical information about our data such as the sum, the min, or the max.

****

---END---