**Pandas, Part 1**

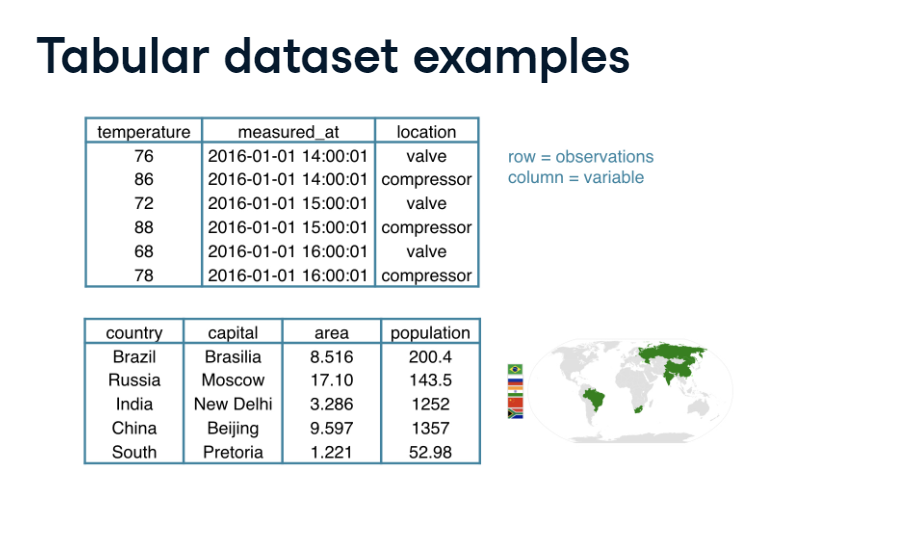
As a data scientist, you'll often be working with tons of data. The form of this data can vary greatly, but pretty often, you can boil it down to a tabular structure, that is, in the form of a table like in a spreadsheet. Let's have a look at some examples.

**Tabular dataset examples**

Suppose you're working in a chemical plant and have a ton of temperature measurements to analyze. This data can come in the following form:

**Tabular dataset examples**

every row is a measurement, or an observation, and for each observation, there are different variables. For each measurement, there's of course the temperature, but also the date and time of the measurement, and the location. Another example: you have collected information on the so-called BRICS countries, Brazil, Russia, India, China and South Africa. You can again build a table with this data, like this. Each row is an observation and represents a country. Each observation has the same variables: the country name, the capital, the area in millions of square kilometers and the population in millions.



**Datasets in Python**

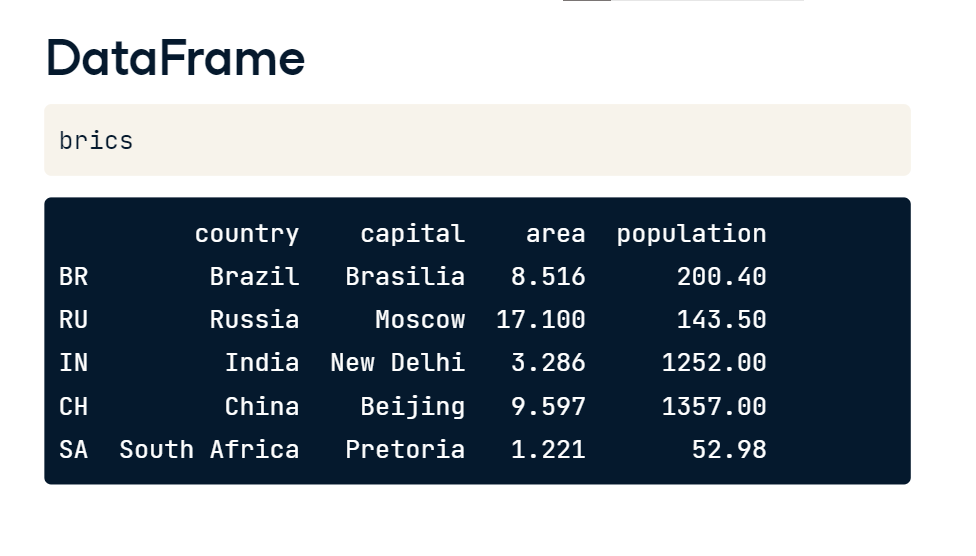
To start working on this data in Python, you'll need some kind of rectangular data structure. That's easy, we already know one! The 2D NumPy array, right? Well, it's an option, but not necessarily the best one. In the two examples we covered, there are different data types and NumPy arrays are not great at handling these. In the BRICS example,

**Datasets in Python**

the area and population are floats, while the country and capital are strings, for example. Your datasets will typically comprise different data types, so we need a tool that's better suited for the job. To easily and efficiently handle this data, there's the Pandas package. Pandas is a high level data manipulation tool developed by Wes McKinney, built on the NumPy package. Compared to NumPy, it's more high level, making it very interesting for data scientists all over the world. In pandas, we store the tabular data like the brics table here in an object called a DataFrame. Have a look at the Pandas DataFrame version of the BRICS data I showed you before:

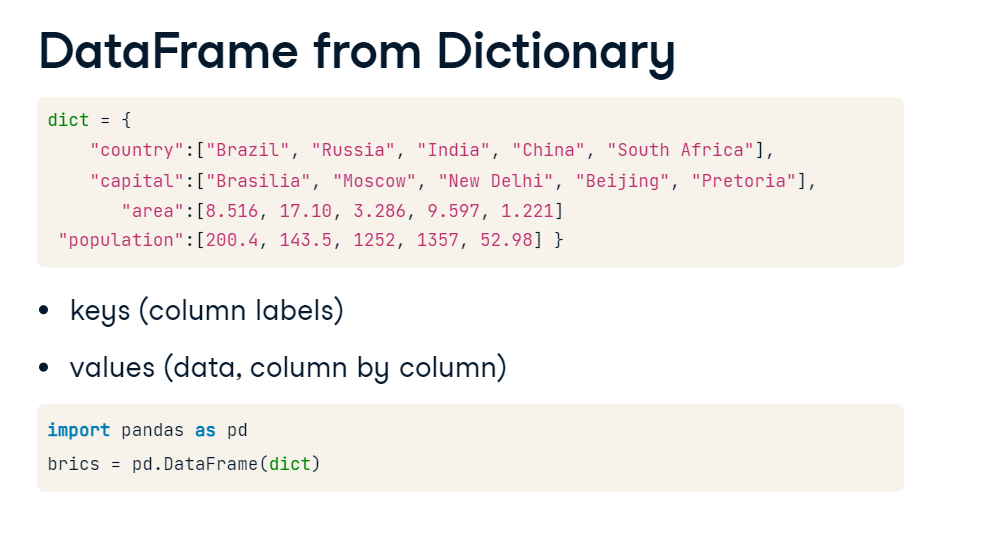
**DataFrame**

You see a similar structure: the rows represent the observations, and the columns represent the variables. Also notice that each row has a unique row label: BR for Brazil, RU for Russia, and so on. The columns, or variables, also have labels: country, population, and so on. Notice that the values in the different columns have different types. This is all great news, but how can we create this DataFrame in the first place? Well, there are different ways.



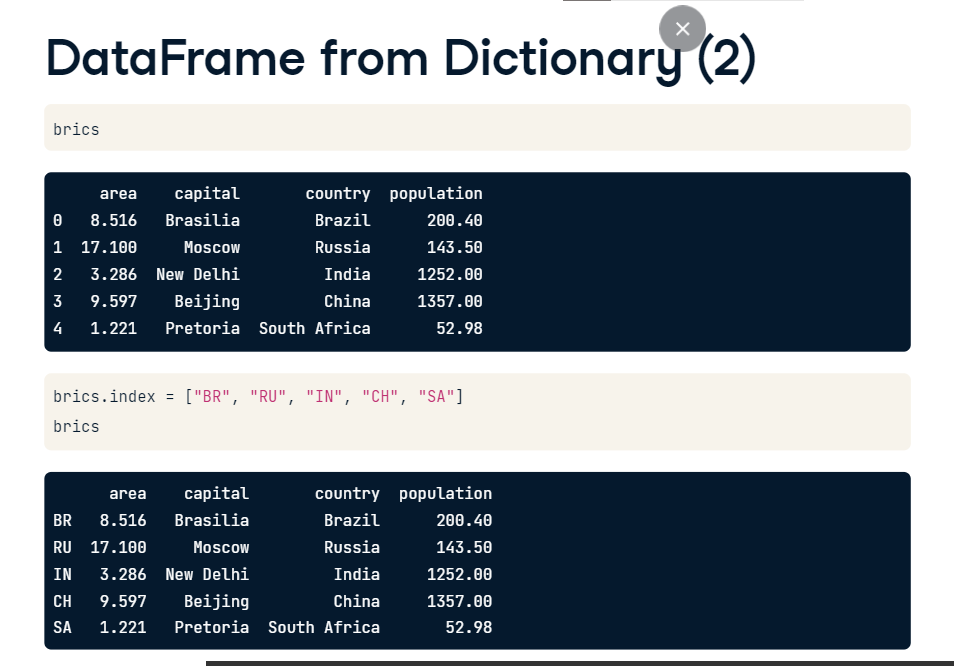
**DataFrame from Dictionary**

First of all, you can build it manually, starting from a dictionary. Using the distinctive curly brackets, we create key value pairs. The keys are the column labels, and the values are the corresponding columns, in list form. After importing the pandas package as pd, you can create a DataFrame from the dictionary using pd (dot) DataFrame.



**DataFrame from Dictionary (2)**

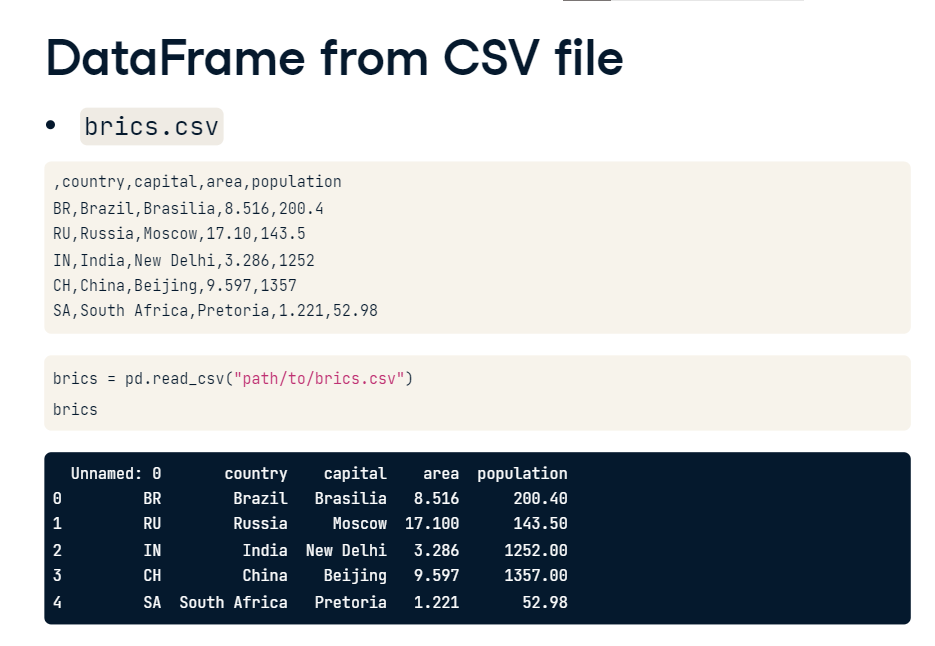
If you check out brics now, we're almost there. Pandas assigned some automatic row labels, 0 up to 4. To specify them manually, you can set the index attribute of brics to a list with the correct labels. The resulting brics DataFrame is the same one as you saw before. Using a dictionary approach is fine, but what if you're working with tons of data, which is typically the case as a data scientist? Well, you won't build the DataFrame manually. Instead, you import data from an external file that contains all this data.



**DataFrame from CSV file**

Suppose the brics data that I showed you before comes in the form of a CSV file called brics.csv. By the way, CSV is short for comma separated values. 

Let's try to import this data into Python using Pandas read\_csv function. You pass the path to the csv file as an argument. If you now print brics, there's still something wrong. The row labels are seen as a column in their own right. To solve this, we'll have to tell the read\_csv function that the first column contains the row indexes. You do this by setting the index\_col argument, like this.



This time brics contains the DataFrame we started with in this video. It nicely contains the row and column labels. The read\_csv function features many more arguments that allow you to customize your data import, make sure to check out its documentation



**Pandas, Part 2**

**brics**

In the previous video, we created the DataFrame brics, containing basic data on the BRICS countries. Here it is again. The code here makes sure that the rows and columns are given appropriate labels. This is important to make accessing columns, rows and single elements in your DataFrame easy. Now that's exactly what I'll show you in this video, what a coincidence!

**Index and select data**

There are numerous ways in which you can index and select data from DataFrames, so we'll take this step by step. First, I'm going to talk about how to use square brackets; next, I'm going to tell you about advanced data access methods, loc and iloc, that make Pandas extra powerful.

**Column Access [ ]**

Suppose that you only want to select the country column from brics. How to do this with square brackets? Well, you type brics, and then the column label inside square brackets. Python prints out the entire column, together with the row labels. But there's something strange here. The last line says Name: country, dtype: object. We're clearly not dealing with a regular DataFrame here. Let's find out about the type of the object that gets returned, with the type function,



**Column Access [ ]**

with the type function, as follows. Okay, so we're dealing with a Pandas Series here. In a simplified sense, you can think of the Series as a 1-dimensional array that can be labeled, just like the DataFrame. Otherwise put, if you paste together a bunch of Series, you can create a DataFrame.



If you want to select the country column but keep the data in a DataFrame, you'll need double square brackets, like this. If you check out the type of this fellow,

it's a good old DataFrame, this time with only one column though.



You can perfectly extend this call to select two columns, country and capital, for example. If you look at it from a different angle, you're actually putting a list with column labels inside another set of square brackets, and end up with a 'sub DataFrame', containing only the country and capital columns. You can also use the same square brackets to select rows from a DataFrame.



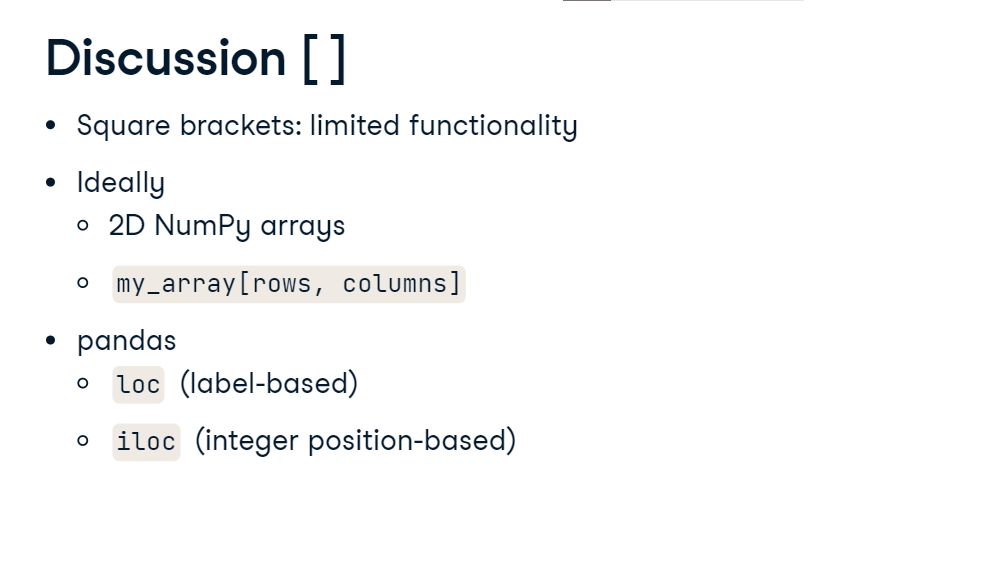
**Row Access [ ]**

The way to do it is by specifying a slice. To get the second, third and fourth rows of brics, we use the slice 1 colon 4. Remember that the end of the slice is exclusive and that the index starts at zero.

Here are the row indexes so that you see what's happening.



**Discussion [ ]**

These square brackets work, but it only offers limited functionality. Ideally, we'd want something similar to 2D NumPy arrays. There, you also used square brackets, the index or slice before the comma referred to the rows, the index or slice after the comma referred to the columns. If we want to do a similar thing with Pandas, we have to extend our toolbox with the loc and iloc functions. loc is a technique to select parts of your data based on labels, iloc is position based. Let's start with loc first. 

**Row Access loc**

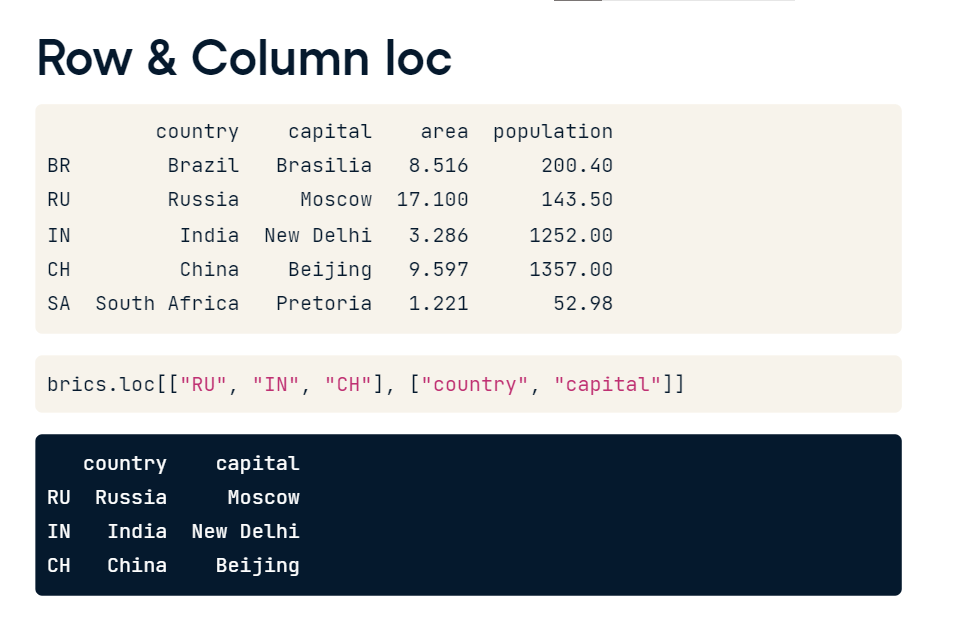
Let's have another look at the brics DataFrame, and try to get the row for Russia. This is how it's done. You put the label of the row of interest in square brackets after loc. Again, we get a Pandas Series, containing all the row's information, rather inconveniently shown on different lines. 

To get a DataFrame, we have to put the "RU" string inside another pair of brackets.

We can also select multiple rows at the same time. Suppose you want to also include India and China. This will do the trick; simply add some more row labels to the list. This was only selecting entire rows, that's something you could also do with the basic square brackets. The difference here is that you can extend your selection with a comma and a specification of the columns of interest. 

**Row & Column loc**

Let's extend the previous call to only include the country and capital columns. We add a comma, and a list of column labels we want to keep. The intersection gets returned. Of course, you can also use loc to select all rows but only a specific number of columns.



Simply replace the first list that specifies the row labels with a colon, a slice going from beginning to end. This time, the intersection spans all rows, but only two columns. 

**Recap**

So, let's take a step back: simple square brackets work fine if you want to get columns; to get rows, you can use slicing. The loc function is more versatile: you can select rows, columns, but also rows and columns at the same time. When you use loc, subsetting becomes remarkable similar to how you subsetted 2D NumPy arrays. The only difference is that you use row labels with loc, not the positions of the elements. If you want to subset Pandas DataFrames based on their position, or index, you'll need the iloc function.



**Row Access iloc**

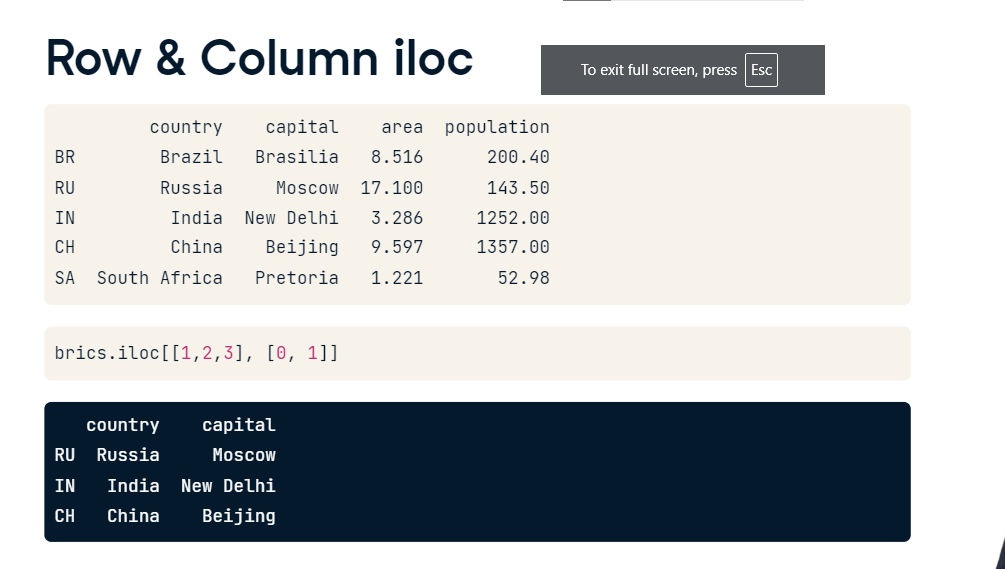
Let's cover the same examples as with loc, and start with getting the row for Russia. In loc, you use the "RU" string in double square brackets, to get a DataFrame, like here. In iloc, you use the index 1 instead of RU. The results are exactly the same. 

To get the rows for Russia, India and China, which was coded like this when using loc, you can now use a list with the index 1 to 3. Again, the same result. 

**Row & Column iloc**

To in addition only keep the country and capital column, which we did as follows with loc,

we put the indexes 0 and 1 in a list after the comma, referring to the country and capital column.



Finally, you can keep all rows and keep only the country and capital column in a similar fashion. With loc, this is how it's done.

For iloc, it's like this. loc and iloc are pretty similar, the only difference is how you refer to columns and rows. I know all of this could be a lot to take in, so lets get you coding yourself to master indexing and selecting data. Keep learning by doing!

