**Loop Data Structures Part 1**

So you already saw how looping over lists and strings works, but what about those other data structures, such as dictionaries and NumPy arrays? Well, in both cases, you can use a similar for loop construct, but the way you define the "sequence" over which you're iterating will differ depending on the data structure.

**Dictionary**

Let's go back to our "world" dictionary, containing country names as keys and corresponding populations as values, shown here. How should we approach this if we want to print out the key and corresponding value for each key:value pair on a new line? Maybe like this, simply hoping that the key and value are correctly set? Unfortunately, we get an error. Python sees that you expect two values in every iteration, like enumerate did before when you wanted the index and value from a list element, but in this case, Python has no idea how to go about this.



We can fix this by calling the method items() on world. This will generate a key and value in each iteration. If you have a look at the printout, there's something strange: afghanistan comes first in world, but not in the printout. That's because dictionaries are inherently unordered: the order in which they're iterated over is not fixed, at least in Python 3.5.

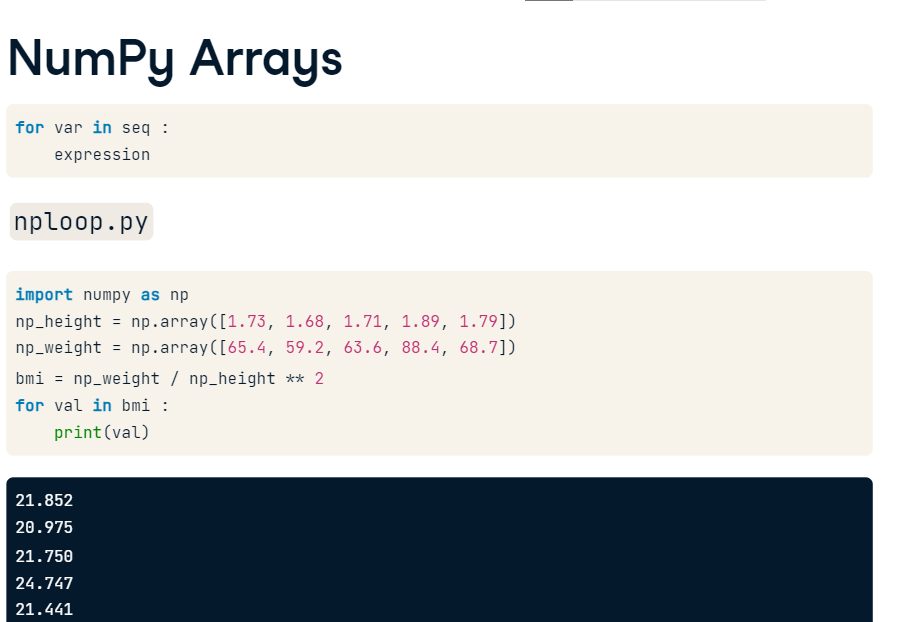


**Dictionary**

The names key and value are totally arbitrary by the way, I can also call these k and v, like here. The order does matter though. The first variable gets the key, the second one the value.

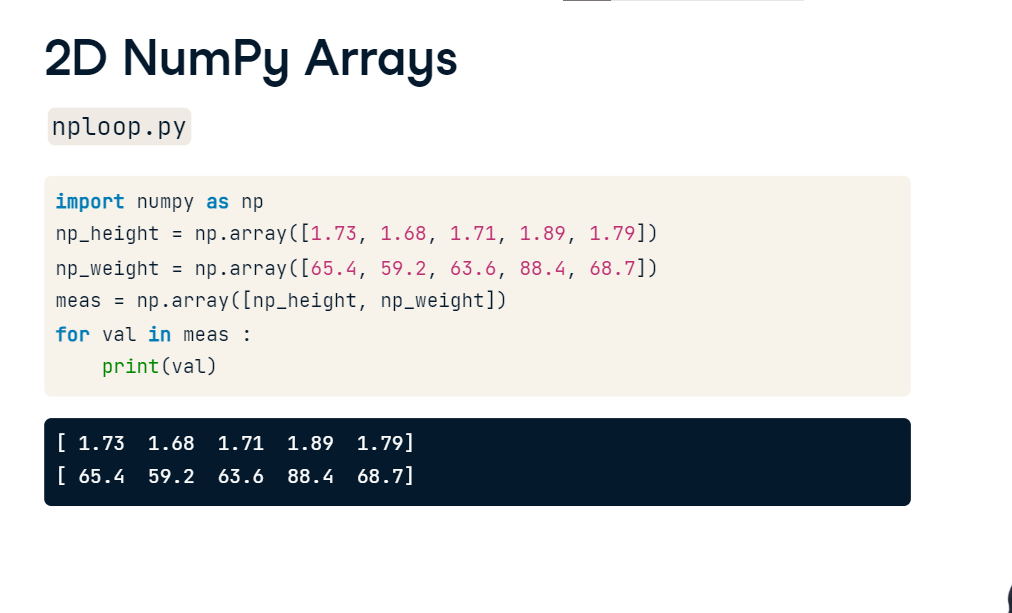
**NumPy Arrays**

Now for the NumPy array, that "data science equivalent" of the Python list I've been talking about quite a bit. Let's start from the bmi array that you already know -- here it is. It's pretty straightforward: the most basic for loop you can imagine already does the trick.



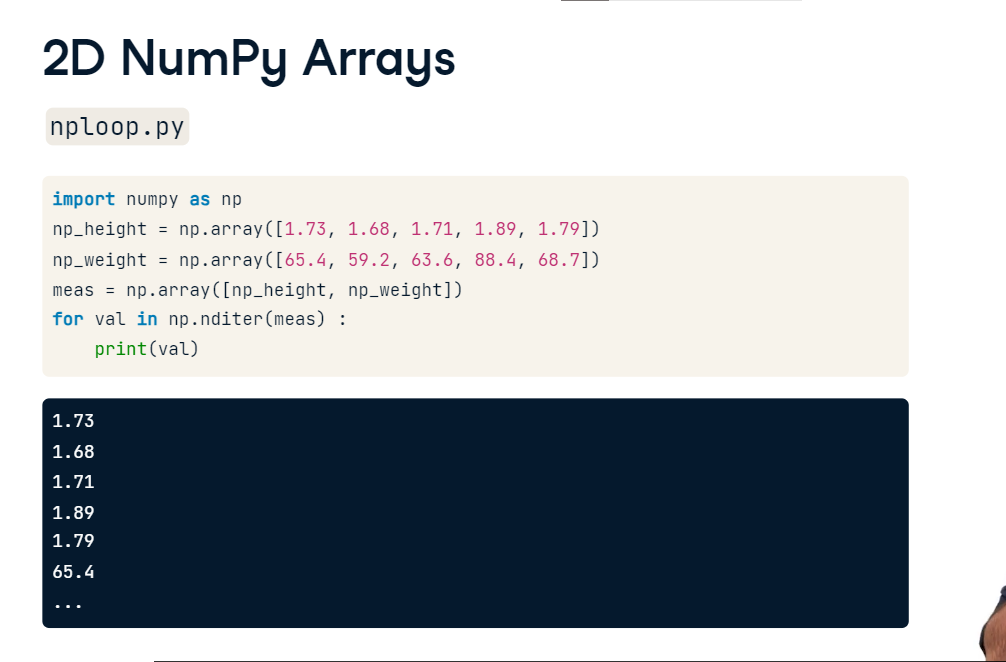
**2D NumPy Arrays**

Let's see if this also works with a 2D NumPy array. Here, I created meas, by combining the np\_height and np\_weight arrays. If we want to print out each element in this 2D array separately, the same basic for loop won't do the trick though. The 2D array is actually built up from an array of 1D arrays. The for loop simply prints out an entire array on each iteration.



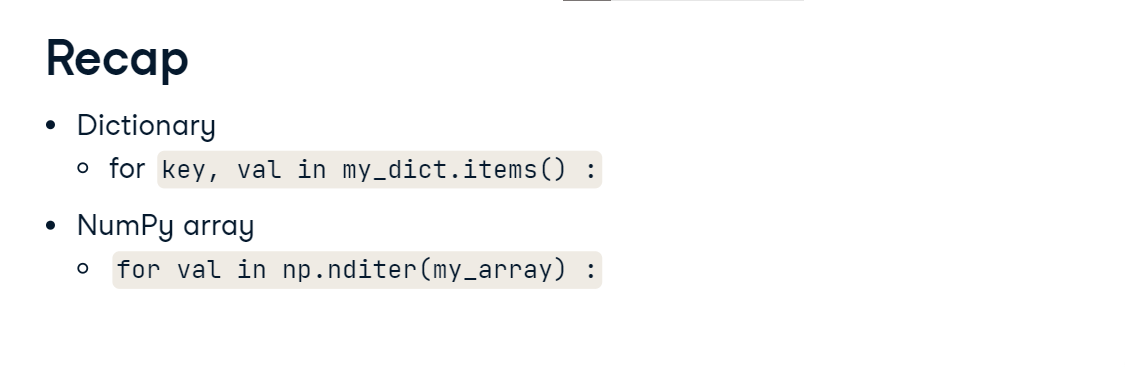
**2D NumPy Arrays**

To get every element of an array, you can use a NumPy function called nditer(). The input is the array you want to iterate over, meas in our case. This time, we get 10 printouts, first all the heights, then all the weights. Nice!



**Recap**

To recap: if you want to iterate over key-value pairs in a dictionary, use the items() method on the dictionary to define the sequence in the for loop. If you want to iterate over all elements in a NumPy array, you should use the nditer() function to specify the sequence. Pay attention here: dictionaries require a method, NumPy arrays use a function.



**Loop Data Structures Part 2**

There's one killer data structure out there that we haven't covered up to now when it comes to looping: the Pandas DataFrame.

**brics**

Let's go over the data on the brics countries one last time. These lines of code import it from the CSV file brics.csv. You can see its contents on the top right here, so you can follow along.

**for, first try**

If a Pandas DataFrame were to function the same way as a 2D NumPy array, then maybe a basic for loop like this, to print out each row, could work. Let's see what the output is. Well, this was rather unexpected. We simply got the column names. Also interesting, but not exactly what we want. In Pandas, you have to mention explicitly that you want to iterate over the rows.



**iterrows**

You do this by calling the iterrows method on the brics country, thus specifying another "sequence": The iterrows method looks at the DataFrame, and on each iteration generates two pieces of data: the label of the row and then the actual data in the row as a Pandas Series. Let's change the rest of the for loop to reflect this change: we store the row label as lab, and the row data as row. To understand what's happening, let's print lab and row seperately. In the first iteration, lab is BR, and row is this entire Pandas Series. Because this row variable on each iteration is a Series, you can easily select additional information from it using the subsetting techniques you learned about earlier.

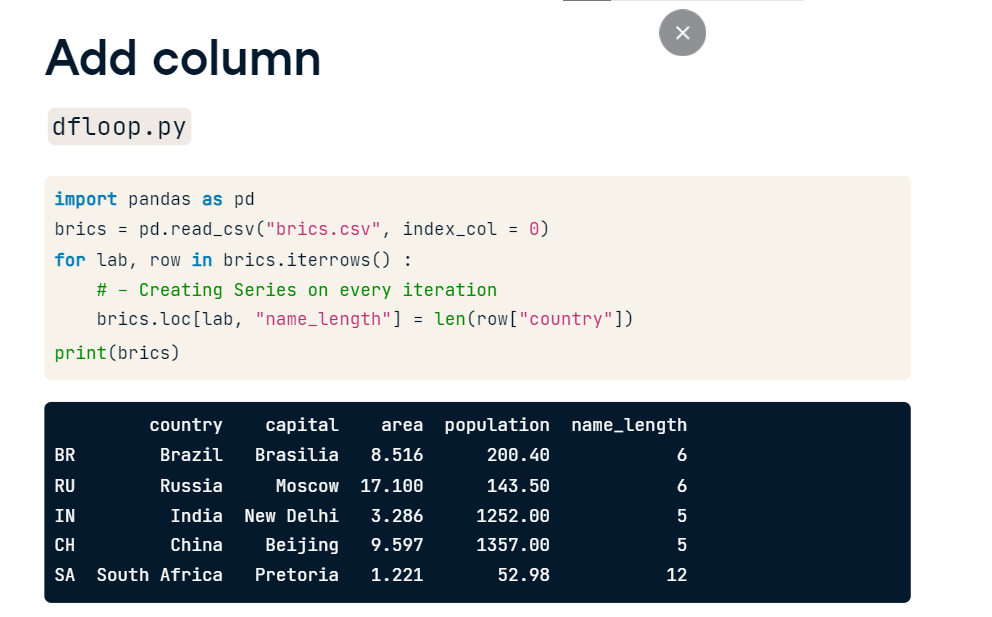


**Selective print**

Suppose you only want to print out the capital on each iteration: let's change the print statement as follows, printing the label and the capital together. You can take things further than simple printouts, though.

**Add column**

Let's add a new column to the brics DataFrame, named name\_length, containing the number of characters in the country's name. To do this, we'll need to blend together a lot of the things we've learned. It's pretty advanced stuff, so try to stay with me here. The specification of the for loop can be the same, because we'll need both the row label and the row data. Next, we can calculate the length of each country name by selecting the country column from row, and then passing it to the len() function, that determines the number of characters in a string. Finally, we'll have to add this new information to a new column, name\_length, at the appropriate location. I used loc here, which is label-based. To see whether we coded things correctly, I'm adding a printout of brics after the for loop, so without indentation. Running this scripts shows that it worked: there's a new column in there with the length of the country names. Nice, but not especially efficient, because you're creating a Series object on every iteration. For this small DataFrame that doesn't matter, but if you're doing funky stuff on a ginormous dataset, this loss in efficiency can become problematic.



**apply**

A way better approach if you want to calculate an entire DataFrame column by applying a function on a particular column in an element-wise fashion, is apply(). In this case, you don't even need a for loop. This is how it's done. Basically, you're selecting the country column from the brics DataFrame, and then, on this column, you apply the len function. Apply calls the len function with each country name as input and produces a new array, that you can easily store as a new column, "name\_length". This is way more efficient, and also easier to read, if you ask me. So, I've told you how to iterate DataFrames with a for loop, and how to do vectorized operations with the apply function. These skills will become very useful once you start transforming your own datasets!

