**NumPy**

Wow, you've done well and by now, you are aware

**Lists Recap**

that the Python list is pretty powerful. A list can hold any type and can hold different types at the same time. You can also change, add and remove elements. This is wonderful, but one feature is missing, a feature that is super important for aspiring data scientists as yourself. When analyzing data, you'll often want to carry out operations over entire collections of values, and you want to do this fast. With lists, this is a problem.

**Illustration**

Let's retake the heights of your family and yourself. Suppose you've also asked for everybody's weight. It's not very polite, but everything for science, right? You end up with two lists, height, and weight. The first person is 1.73 meters tall and weighs 65.4 kilograms. If you now want to calculate the Body Mass Index for each family member, you'd hope that this call can work, making the calculations element-wise. Unfortunately, Python throws an error, because it has no idea how to do calculations on lists. You could solve this by going through each list element one after the other, and calculating the BMI for each person separately, but this is terribly inefficient and tiresome to write.

**Solution: NumPy**

A way more elegant solution is to use NumPy, or Numeric Python. It's a Python package that, among others, provides a alternative to the regular python list: the NumPy array. The NumPy array is pretty similar to the list, but has one additional feature: you can perform calculations over entire arrays. It's really easy, and super-fast as well. The NumPy package is already installed on DataCamp's servers, but if you want to work with it on your own system, go to the command line and execute pip3 install numpy.

Next,to actually use NumPy in your Python session, you can import the numpy package, like this. Let's start with creating a numpy array. You do this with NumPy's array function: the input is a regular Python list. I'm using array twice here, to create NumPy versions of the height and weight lists from before: np\_height and np\_weight: Let's try to calculate everybody's BMI with a single call again. This time, it worked fine: the calculations were performed element-wise. The first person's BMI was calculated by dividing the first element in np\_weight by the square of the first element in np\_height, the second person's BMI was calculated with the second height and weight elements, and so on.

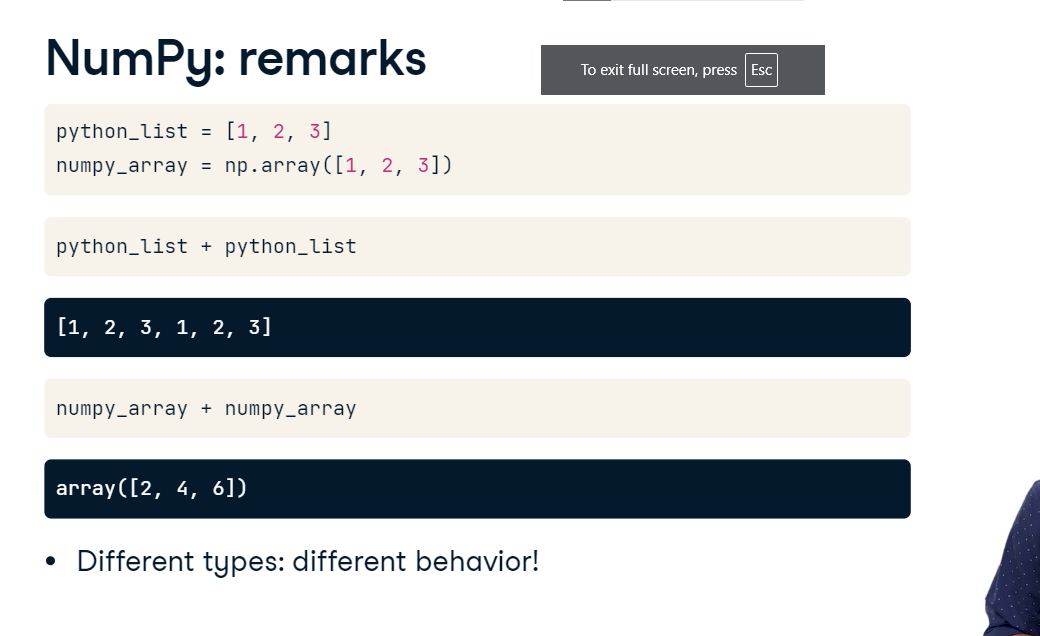
**Comparison**

Let's do a quick comparison here. First, we tried to do calculations with regular lists, like this, but this gave us an error, because Python doesn't now how to do calculations with lists like we want them to. Next, these regular lists where converted to NumPy arrays. The same operations now work without any problem: NumPy knows how to work with arrays as if they are single values, which is pretty awesome if you ask me.



**NumPy: remarks**

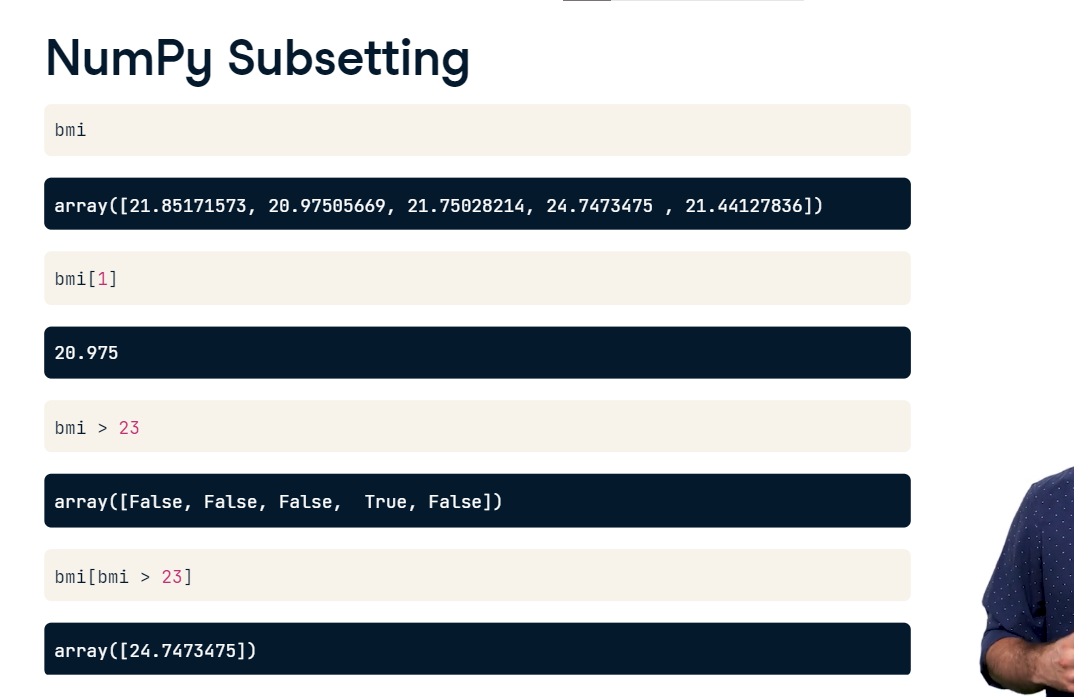
You should still pay attention, though. First of all, NumPy can do all of this so easily because it assumes that your NumPy array can only contain values of a single type. It's either an array of floats, either an array of booleans, and so on. If you do try to create an array with different types, like this for example, the resulting NumPy array will contain a single type, string in this case. The boolean and the float were both converted to strings. Second, you should know that a NumPy array is simply a new kind of Python type, like the float, string and list types from before. This means that it comes with its own methods, which can behave differently than you'd expect.



Take this Python list and this numpy array, for example. If you do python\_list + python\_list, the list elements are pasted together, generating a list with 6 elements. If you do this with the numpy arrays, on the other hand, Python will do an element-wise sum of the arrays. Just make sure to pay attention when you're juggling around with different Python types, because the outcomes can differ a lot! Apart from these subtleties,

**NumPy Subsetting**

you can work with NumPy arrays pretty much the same as you can with regular Python lists. When you want to get elements from your array, for example, you can use square brackets. Suppose you want to get the bmi for the second person, so at index 1. This will do the trick. Specifically for NumPy, there's also another way to do list subsetting: using an array of booleans. Say you want to get all BMI values in the bmi array that are over 23. A first step is using the greater than sign, like this: The result is a NumPy array containing booleans: True if the corresponding bmi is above 23, False if it's below. Next, you can use this boolean array inside square brackets to do subsetting. Only the elements in bmi that are above 23, so for which the corresponding boolean value is True, is selected. There's only one BMI that's above 23, so we end up with a NumPy array with a single value, that specific BMI. Using the result of a comparison to make a selection of your data is a very common way to get surprising insights.



**2D NumPy Arrays**

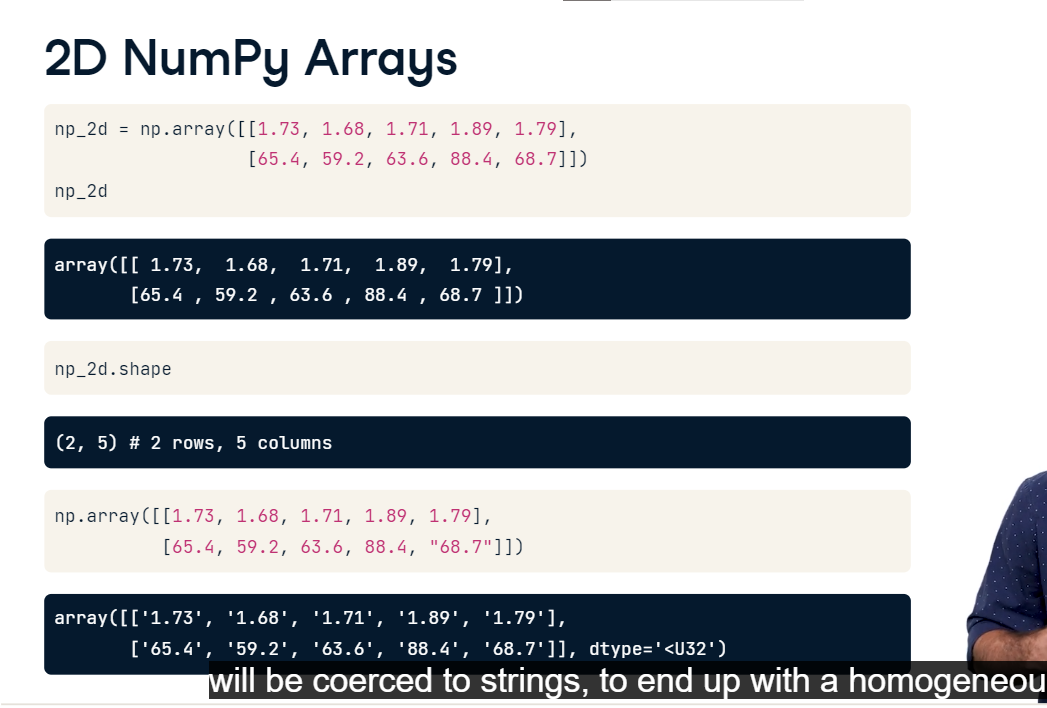
Well done you legend! Let's now recreate the numpy arrays from the previous video.

**Type of NumPy Arrays**

If you ask for the type of these arrays, Python tells you that they are numpy.ndarray. numpy dot tells you it's a type that was defined in the numpy package. ndarray stands for n-dimensional array. The arrays np\_height and np\_weight are one-dimensional arrays, but it's perfectly possible to create 2 dimensional, three dimensional, heck even seven dimensional arrays! Let's stick to 2 in this video though.

**2D NumPy Arrays**

You can create a 2D numpy array from a regular Python list of lists. Let's try to create one numpy array for all height and weight data of your family, like this. If you print out np\_2d now, you'll see that it is a rectangular data structure: Each sublist in the list, corresponds to a row in the two dimensional numpy array. From np\_2d.shape, you can see that we indeed have 2 rows and 5 columns. shape is a so-called attribute of the np2d array, that can give you more information about what the data structure looks like. Note that the syntax for accessing an attribute looks a bit like calling a method, but they are not the same! Remember that methods have round brackets after them, and, you can see here, attributes do not. Also for 2D arrays, the NumPy rule applies: an array can only contain a single type. If you change one float to be string, all the array elements will be coerced to strings, to end up with a homogeneous array.

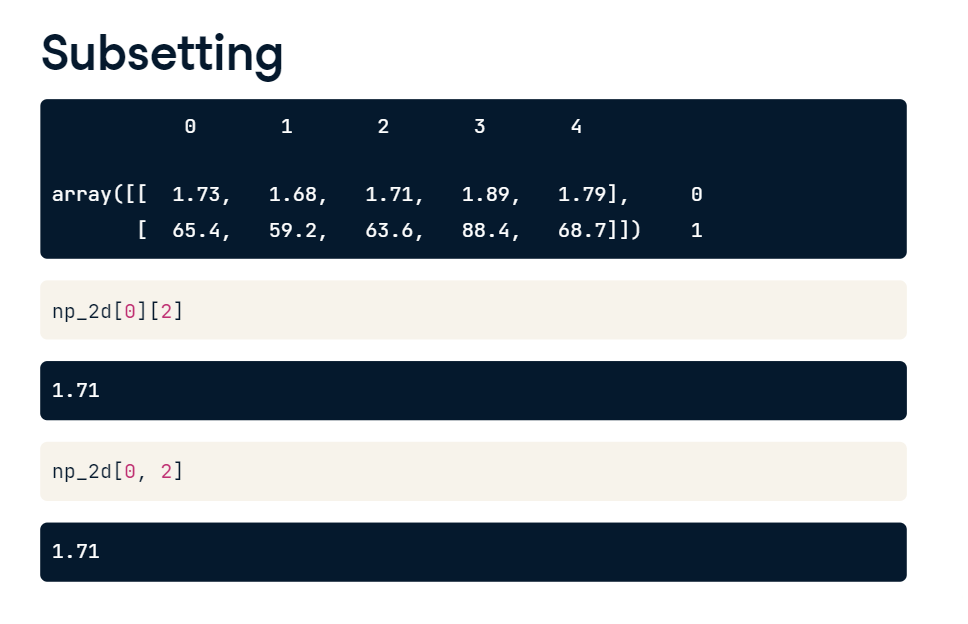


**Subsetting**

You can think of the 2D numpy array as an improved list of lists: you can perform calculations on the arrays, like I showed before, and you can do more advanced ways of subsetting. Suppose you want the first row, and then the third element in that row. To select the row, you need the index 0 in square brackets. Don't forget about zero indexing. To then select the third element, you can extend the same call with another pair of brackets, this time with the index 2,

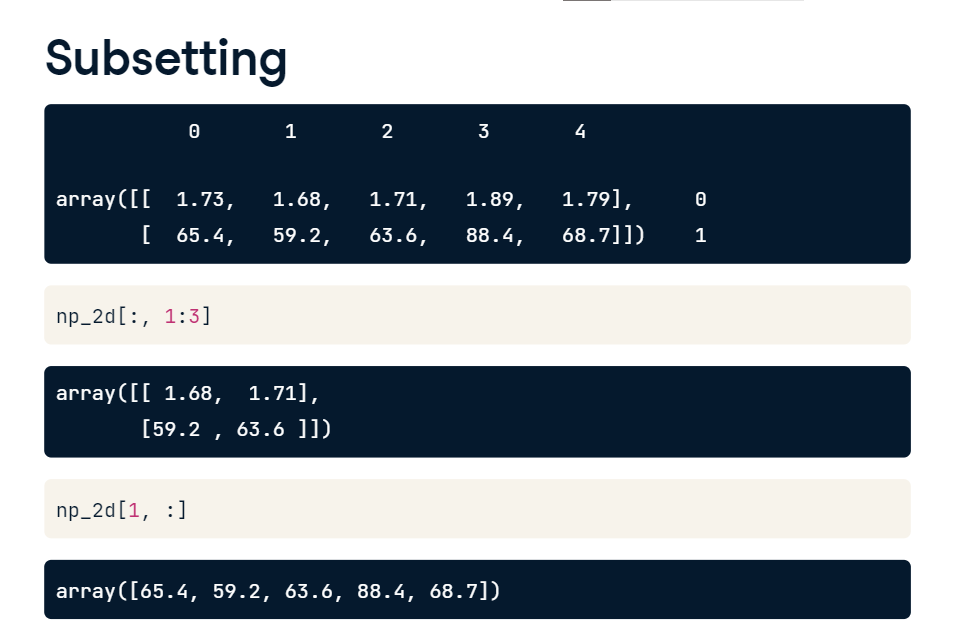
**Subsetting**

like this. Basically you're selecting the row, and then from that row do another selection. There's also an alternative way of subsetting, using single square brackets and a comma. This call returns the exact same value as before. The value before the comma specifies the row, the value after the comma specifies the column. The intersection of the rows and columns you specified, are returned.



Once you get used to it, this syntax is

Suppose you want to select the height and weight of the second and third family member. You want both rows, so you put in a colon before the comma. You only want the second and third column, so you put in the indices 1 to 3 after the comma. Remember that the third index is not included here. The intersection gives us a 2D array with 2 rows and 2 columns: Similarly, you can select the weight of all family members like this: you only want the second row, so put 1 before the comma. You want all columns, so you use a colon after the comma. The intersection gives us the entire second row. Finally, 2D numpy arrays enable you to do element-wise calculations, the same way you did it with 1D numpy arrays. That's something



**NumPy: Basic Statistics**

A typical first step in analyzing your data,

**Data analysis**

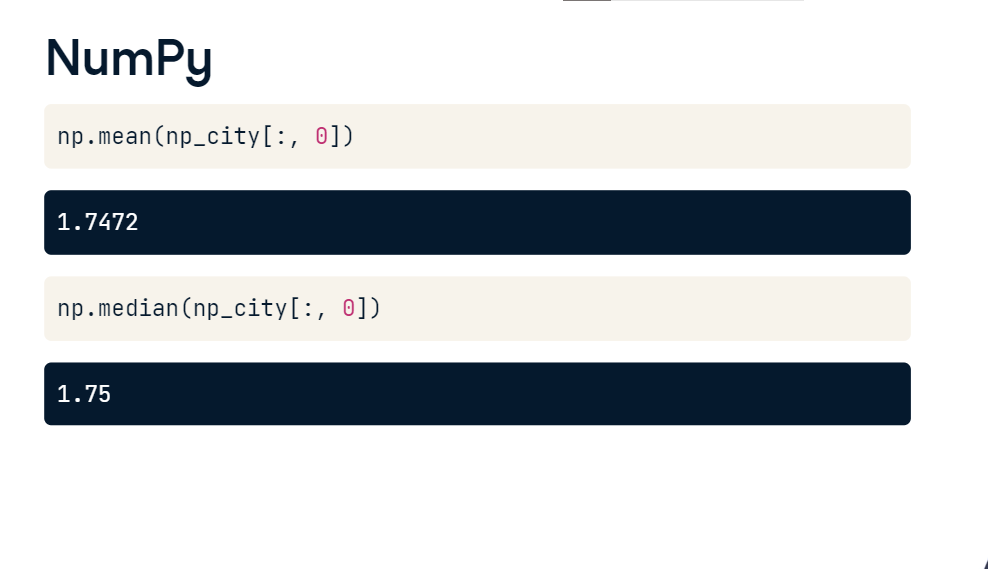
is getting to know your data in the first place. For the NumPy arrays from before, this is pretty easy, because it isn't a lot of data. However, as a data scientist, you'll be crunching thousands, if not millions or billions of numbers.

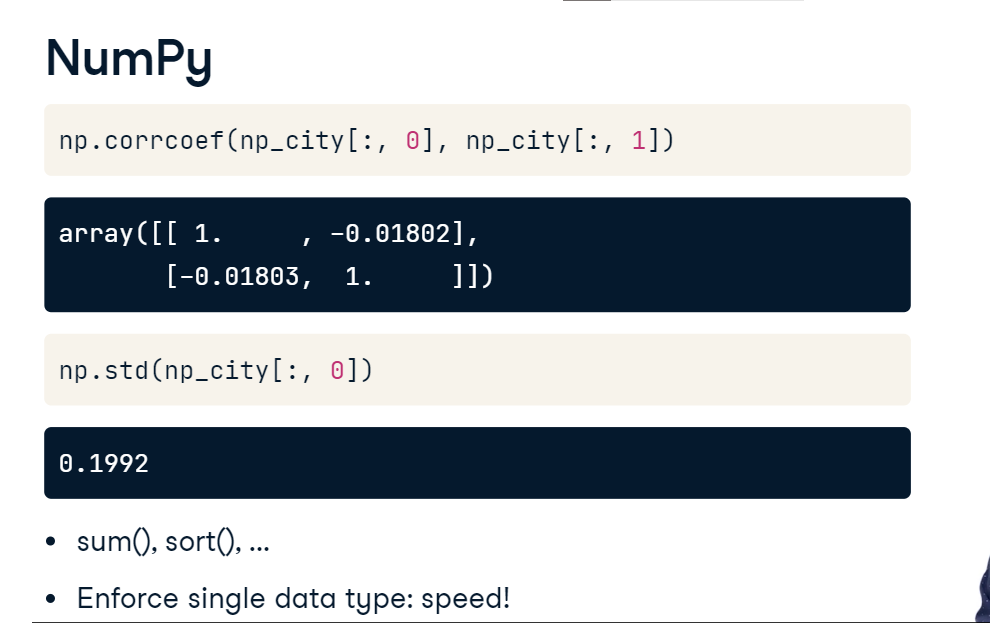
**City-wide survey**

Imagine you conduct a city-wide survey where you ask 5000 adults about their height and weight. You end up with something like this: a 2D numpy array, which I named np\_city, that has 5000 rows, corresponding to the 5000 people, and two columns, corresponding to the height and the weight. Simply staring at these numbers like a zombie won't give you any insights. What you can do, though, is generate summarizing

**NumPy**

Aside from an efficient data structure for number crunching, it happens that NumPy is also good at doing these kinds of things. For starters, you can try to find out the average height of these 5000 people, with NumPy's mean function. Because it's a function from the NumPy package, don't forget to start with np.. Of course, I first had to do a subsetting operation to get the height column from the 2D array. It appears that on average, people are 1.75 meters tall. What about the median height? This is the height of the middle person if you sort all persons from small to tall. Instead of writing complicated python code to figure this out, you can simply use NumPy's median function: You can do similar things for the weight column in np\_city. Often, these summarizing statistics will provide you with a "sanity check" of your data. If you end up with a average weight of 2000 kilograms, your measurements are most likely incorrect. Apart from mean and median, there's also other functions, like corrcoeff to check if for example height and weight are correlated, and std, for standard deviation. NumPy also features more basic functions, such as sum and sort, which also exist in the basic Python distribution. However, the big difference here is speed. Because NumPy enforces a single data type in an array, it can drastically speed up the calculations.



****

**Generate data**

Just a sidenote here: If you're wondering how I came up with the data in this video: We simulated it with NumPy functions! I sampled two random distributions 5000 times to create the height and weight arrays, and then used column\_stack to paste them together as two columns. Another awesome thing that NumPy can do! Another great tool to get some sense of your data is to visualize it, but that's something for the next course also

