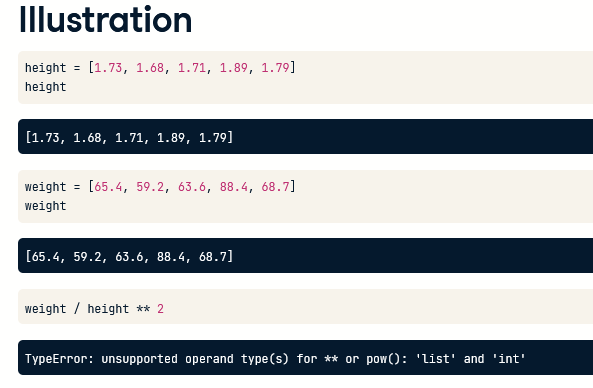
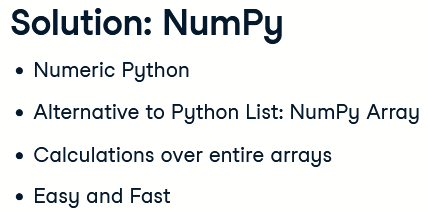
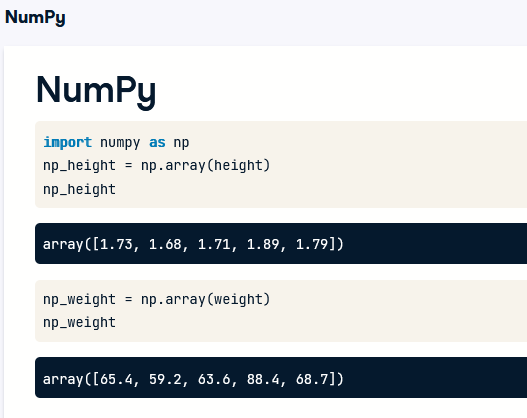
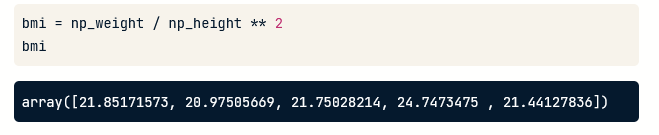
Lists have no idea how to do calculations:





Create NumPy array: input is regular Python list



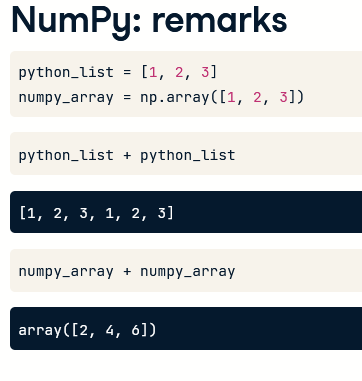


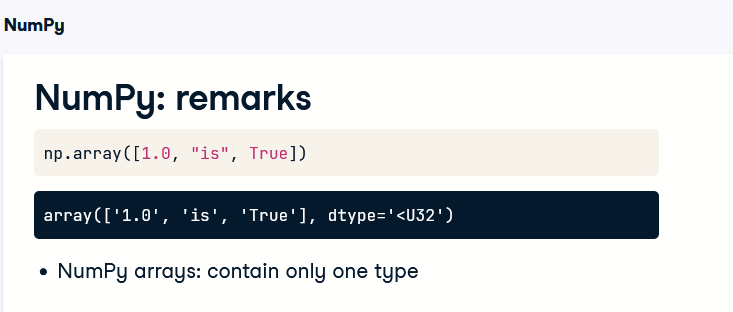


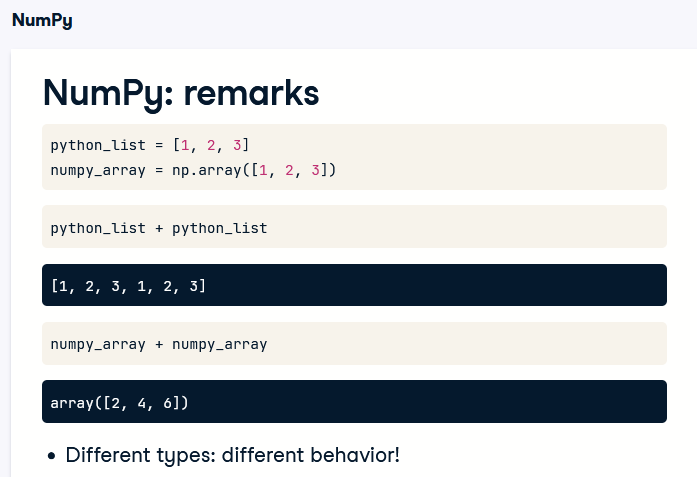
Mixed converts to one type

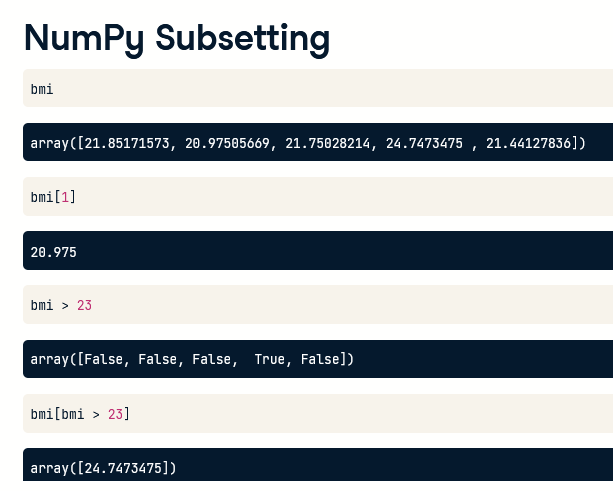
First of all, NumPy can do all of this so easily because it assumes that your NumPy array can only contain values of a single type. It's either an array of floats, either an array of booleans, and so on. If you do try to create an array with different types, like this for example, the resulting NumPy array will contain a single type, string in this case. The boolean and the float were both converted to strings.

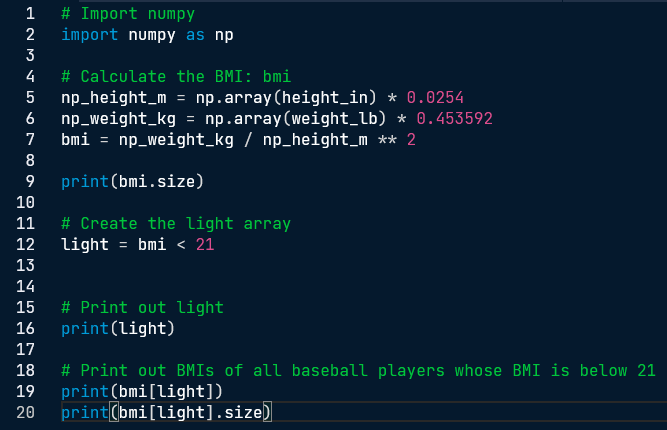
a NumPy array is simply a new kind of Python type, like the float, string and list types from before. This means that it comes with its own methods, which can behave differently than you'd expect



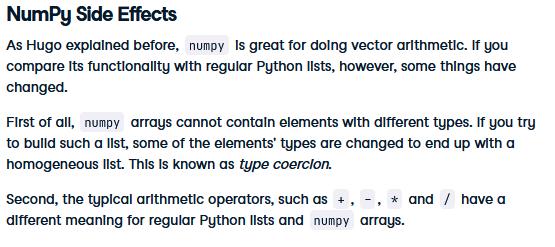


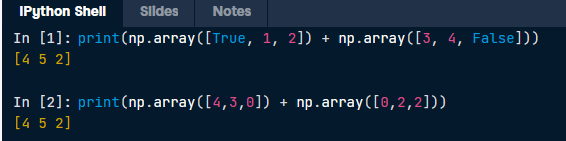




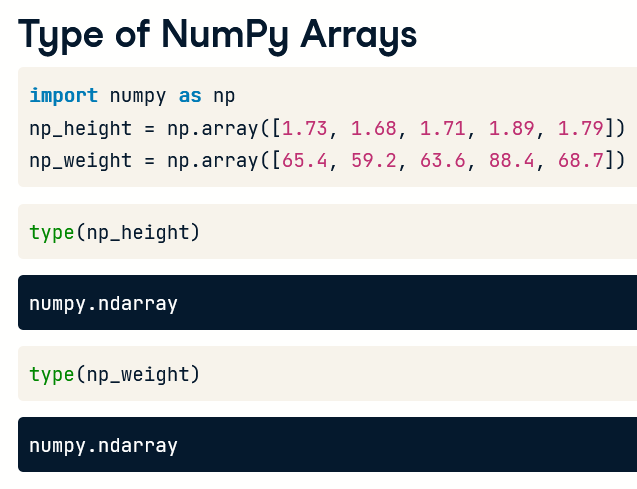


Type Coercion

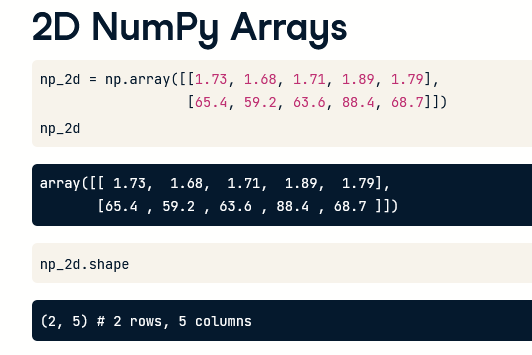






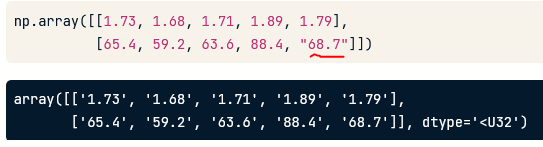


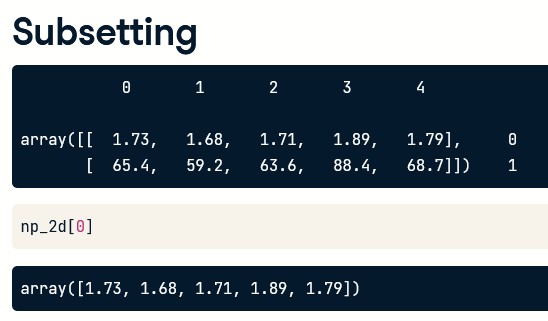
numpy dot tells you it's a type that was defined in the numpy package. ndarray stands for n-dimensional array.

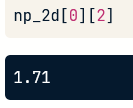


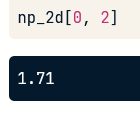
shape is a so-called attribute of the np2d array, that can give you more information about what the data structure looks like; attributes are not methods – methods have trailing ()

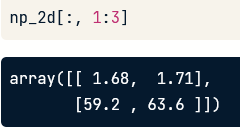
Coercion

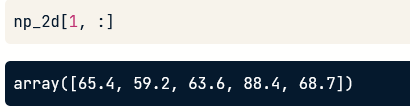


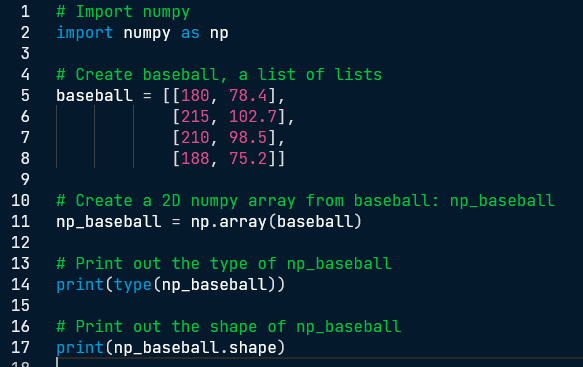


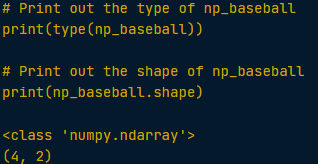




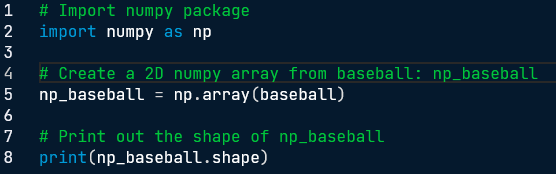








have another look at the MLB data and realize that it makes more sense to restructure all this information in a 2D numpy array. This array should have 1015 rows, corresponding to the 1015 baseball players you have information on, and 2 columns (for height and weight).





Have a look at the code below where the elements "a" and "c" are extracted from a list of lists.

# regular list of lists

x = [["a", "b"], ["c", "d"]]

[x[0][0], x[1][0]]

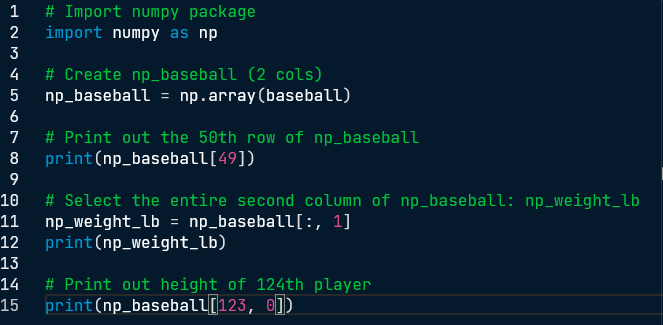
# numpy

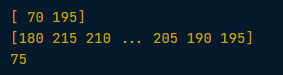
import numpy as np

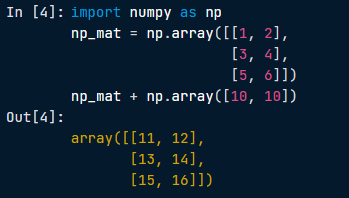
np\_x = np.array(x)

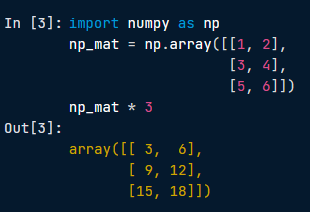
np\_x[:, 0]

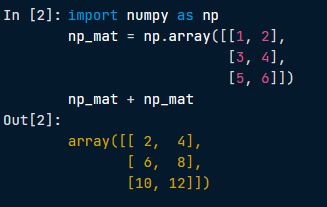
For regular Python lists, this is a real pain. For 2D numpy arrays, however, it's pretty intuitive! The indexes before the comma refer to the rows, while those after the comma refer to the columns. The”:” is for slicing; in this example, it tells Python to include all rows.





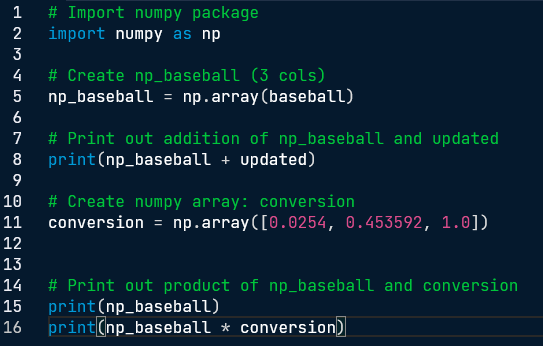






np\_baseball is coded for you; it's again a 2D numpy array with 3 columns representing height (in inches), weight (in pounds) and age (in years). baseball is available as a regular list of lists and updated is available as 2D numpy array.

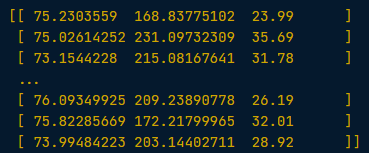
* You managed to get hold of the changes in height, weight and age of all baseball players. It is available as a 2D numpy array, updated. Add np\_baseball and updated and print out the result.
* You want to convert the units of height and weight to metric (meters and kilograms, respectively). As a first step, create a numpy array with three values: 0.0254, 0.453592 and 1. Name this array conversion.
* Multiply np\_baseball with conversion and print out the result.



np\_baseball



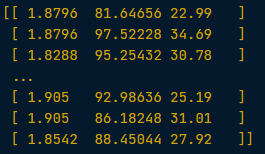
np\_baseball + updated

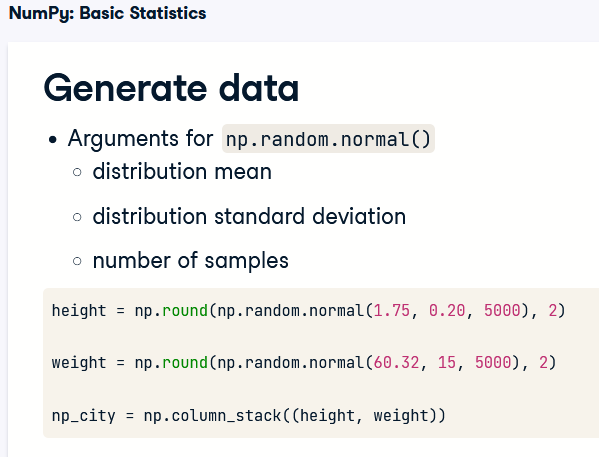


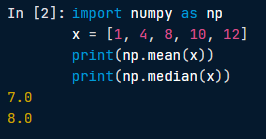
conversion

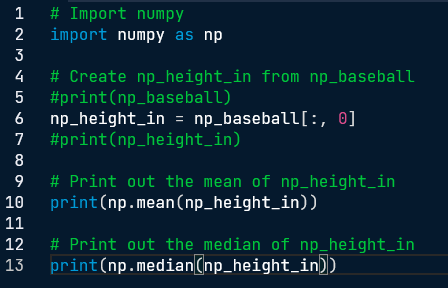


np\_baseball \* conversion



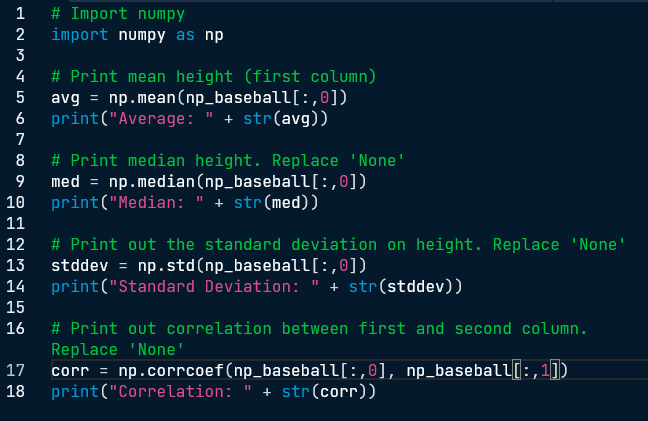


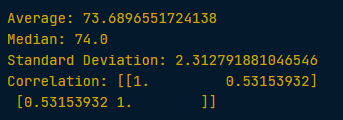






An average height of 1586 inches, that doesn't sound right, does it? However, the median does not seem affected by the outliers: 74 inches makes perfect sense. It's always a good idea to check both the median and the mean, to get an idea about the overall distribution of the entire dataset.





You've contacted FIFA for some data and they handed you two lists. The lists are the following:

positions = ['GK', 'M', 'A', 'D', ...]

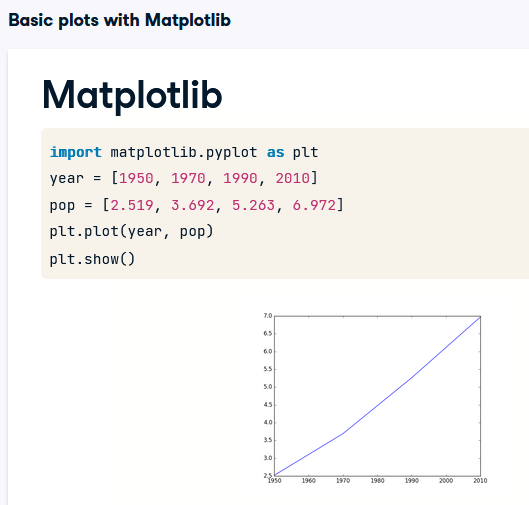
heights = [191, 184, 185, 180, ...]

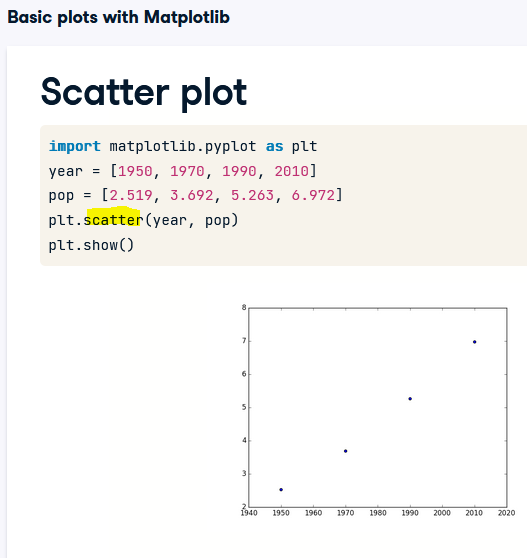
Each element in the lists corresponds to a player. The first list, positions, contains strings representing each player's position. The possible positions are: 'GK' (goalkeeper), 'M' (midfield), 'A' (attack) and 'D' (defense). The second list, heights, contains integers representing the height of the player in cm. The first player in the lists is a goalkeeper and is pretty tall (191 cm).

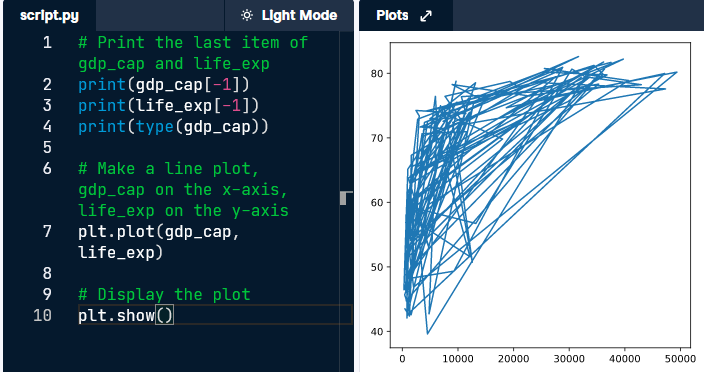
You're fairly confident that the median height of goalkeepers is higher than that of other players on the soccer field. Some of your friends don't believe you, so you are determined to show them using the data you received from FIFA and your newly acquired Python skills. heights and positions are available as lists



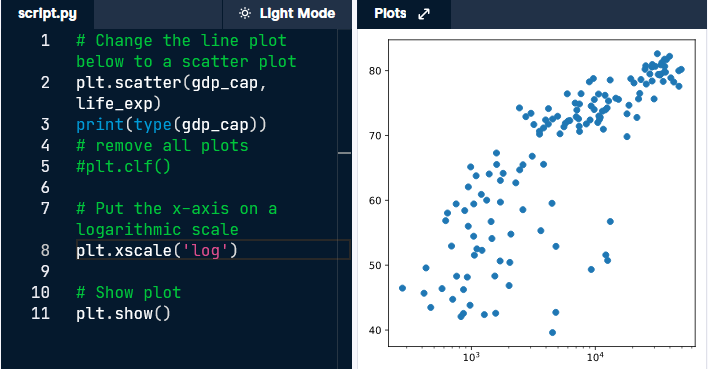


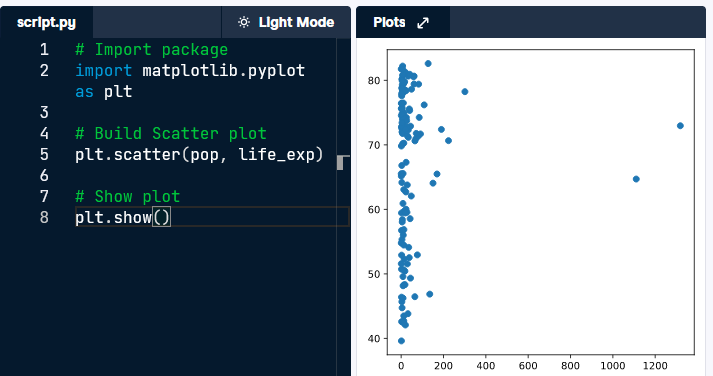






Well done, but this doesn't look right. Let's build a plot that makes more sense.

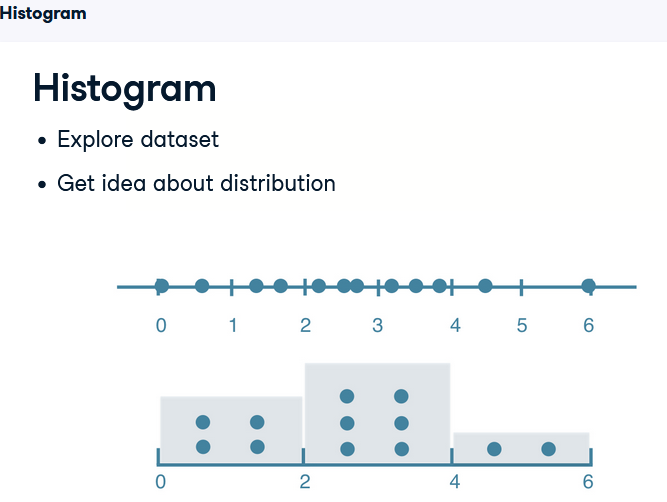


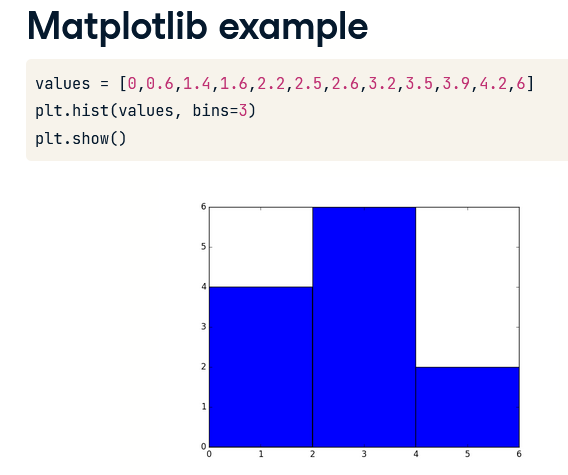


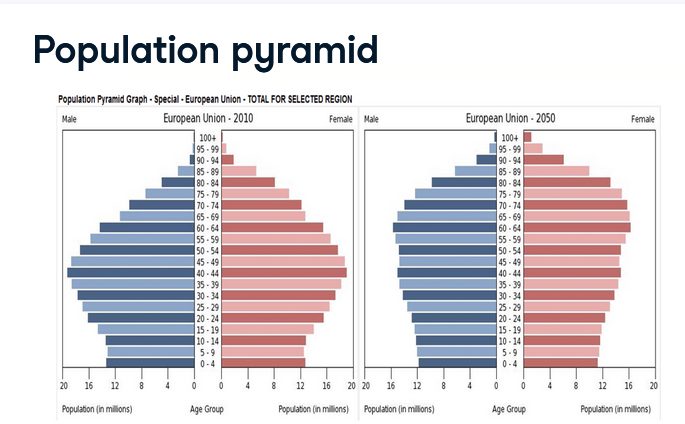
Do you think there's a relationship between population and life expectancy of a country? The list life\_exp from the previous exercise is already available. In addition, now also pop is available, listing the corresponding populations for the countries in 2007. The populations are in millions of people.

Do you see a correlation?

There's no clear relationship between population and life expectancy, which makes perfect sense.



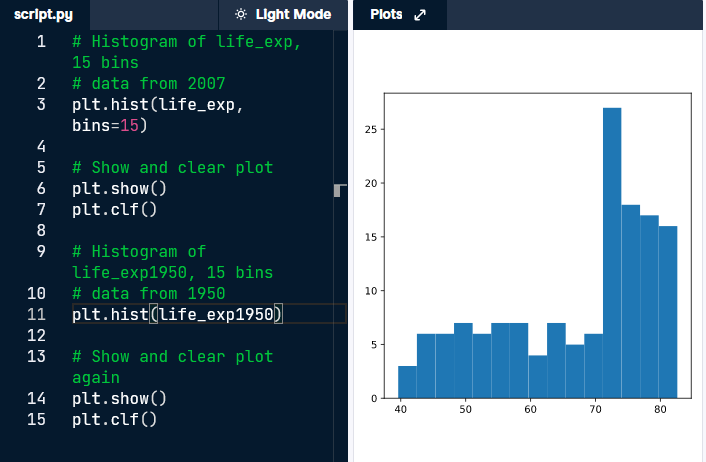




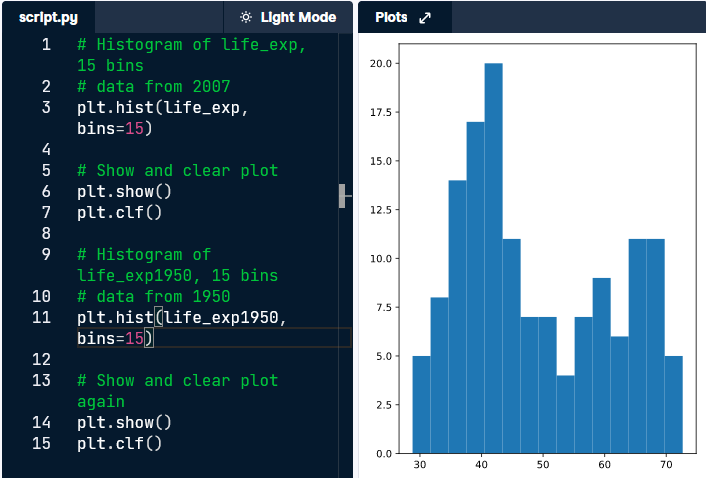
otice that the histograms are flipped 90 degrees; the bins are horizontal now. The bins are largest for the ages 40 to 44, where there are 20 million males and 20 million females. They are the so called baby boomers. These are figures of the year 2010. What do you think will have changed in 2050?

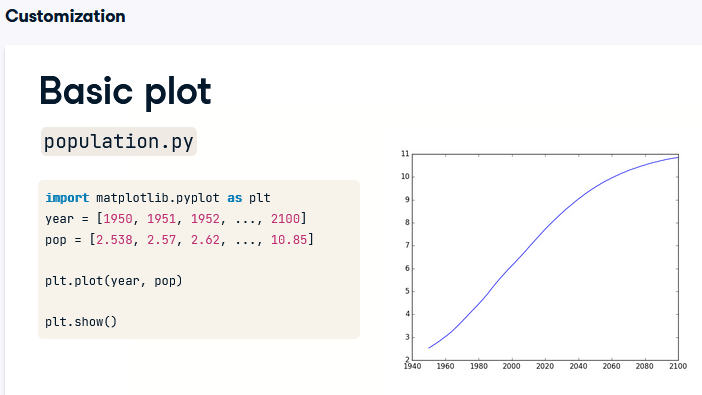
Let's have a look. The distribution is flatter, and the baby boom generation has gotten older. With the blink of an eye, you can easily see how demographics will be changing over time. That's the true power of histograms at work here!

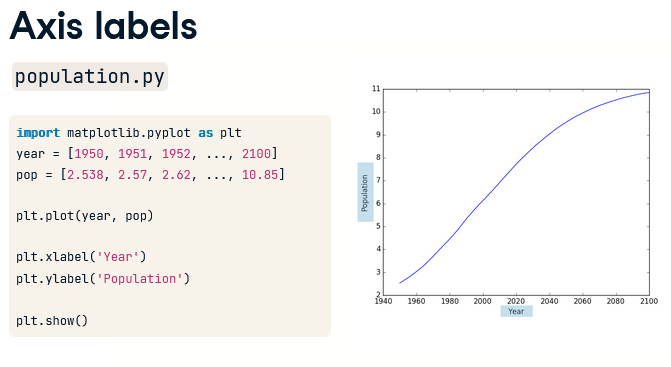
2007 life expectancy

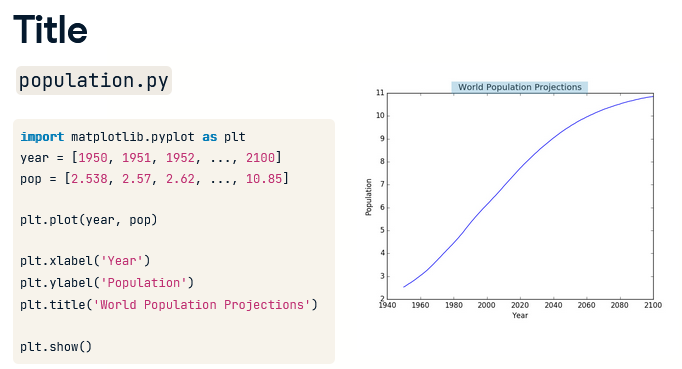


1950 life expectancy

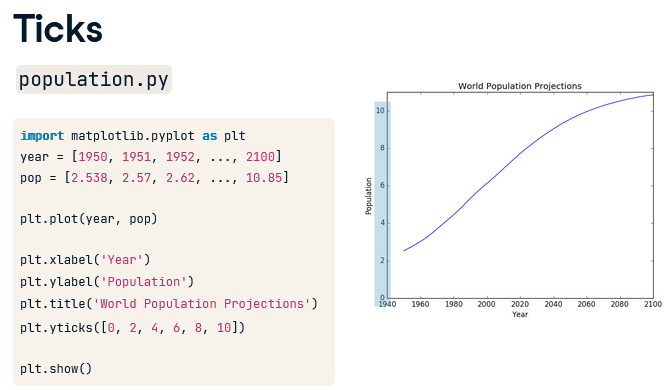




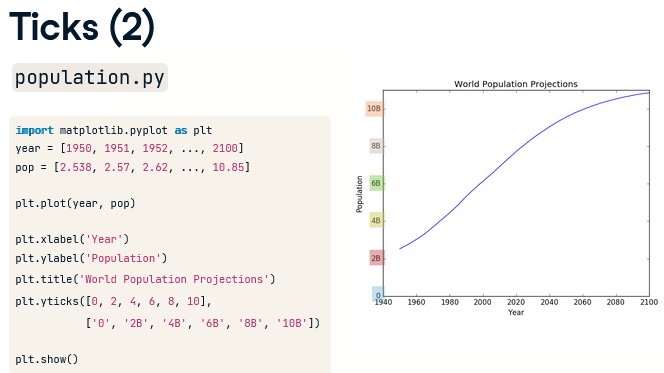


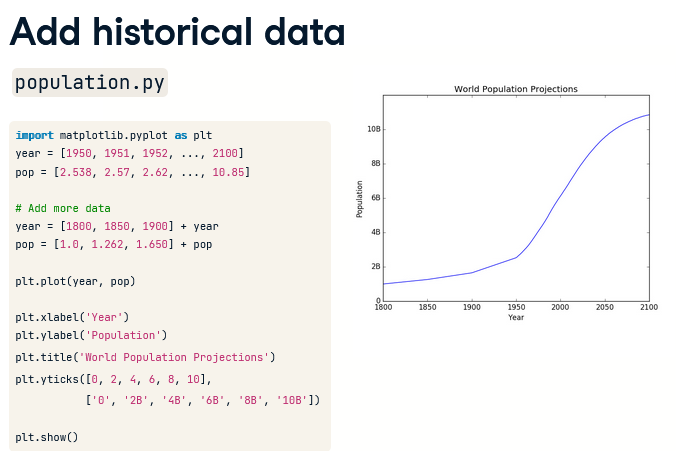


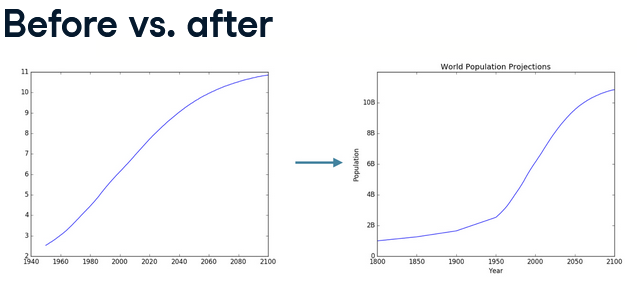
To put the population growth in perspective, I want to have the y-axis start from zero. the plot will change: the curve shifts up. Now it's clear that already in 1950, there were already about 2.5 billion people on this planet.

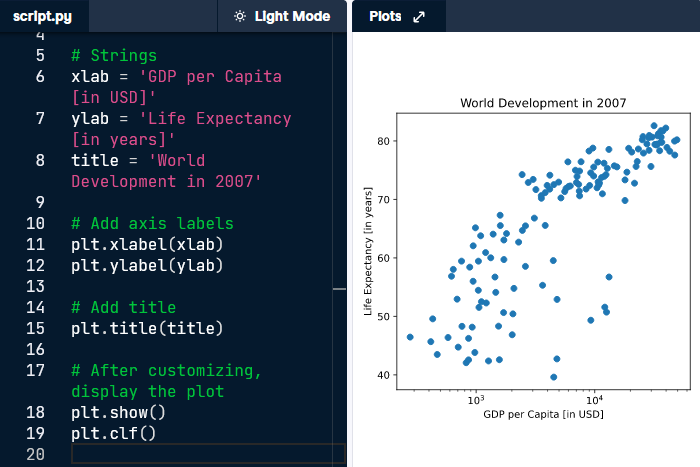


Next, to make it clear we're talking about billions, we can add a second argument to the y-ticks function, which is a list with the display names of the ticks. This list should have the same length as the first list. The tick 0 gets the name 0, the tick 2 gets the name 2B, the tick 4 gets the name 4B and so on. By the way, B stands for Billions





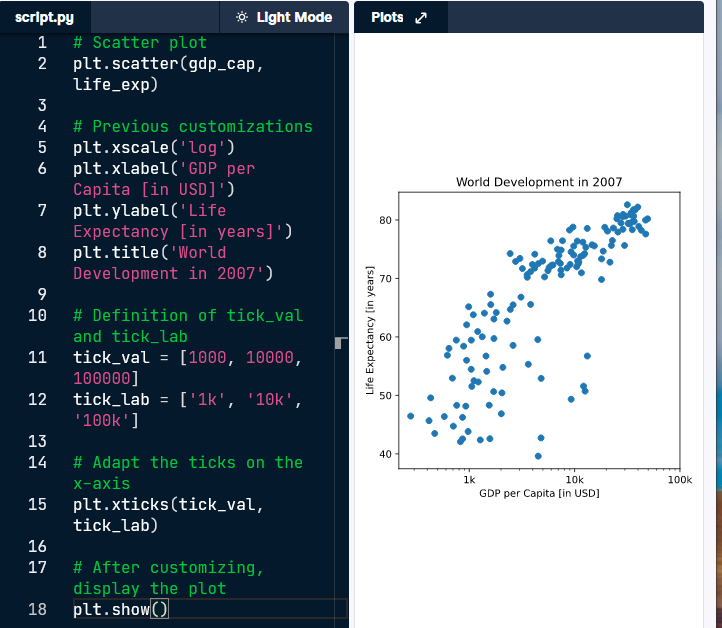




In the video, Hugo has demonstrated how you could control the y-ticks by specifying two arguments:

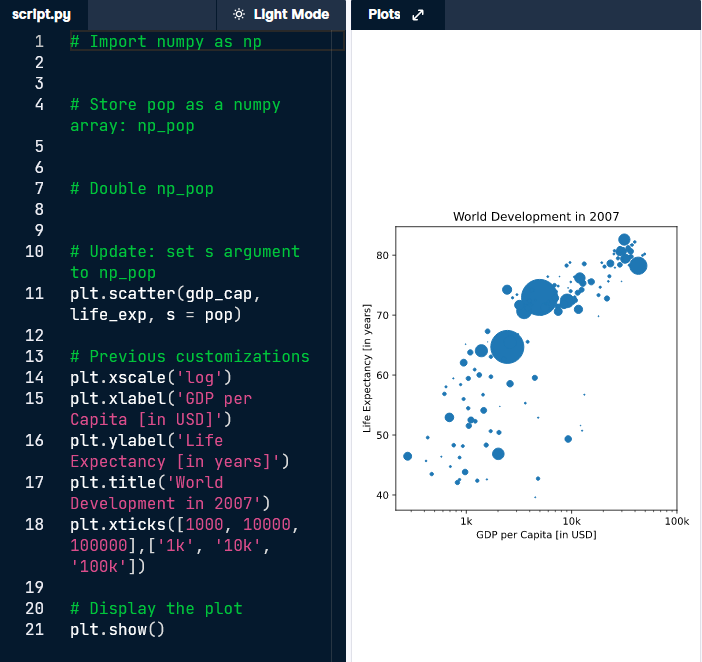
plt.yticks([0,1,2], ["one","two","three"])

In this example, the ticks corresponding to the numbers 0, 1 and 2 will be replaced by one, two and three, respectively.



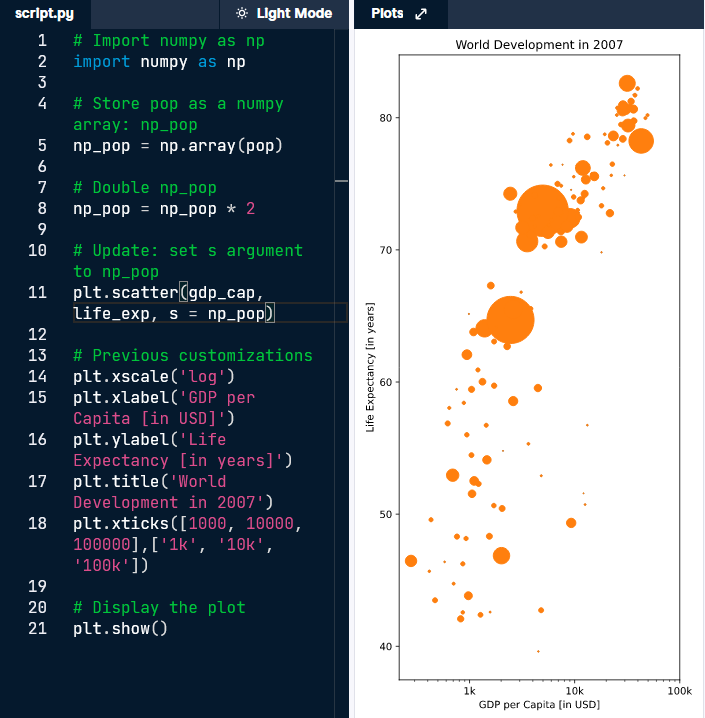
Right now, the scatter plot is just a cloud of blue dots, indistinguishable from each other. Let's change this. Wouldn't it be nice if the size of the dots corresponds to the population?

To accomplish this, there is a list pop loaded in your workspace. It contains population numbers for each country expressed in millions. You can see that this list is added to the scatter method, as the argument s, for size.



Looks good, but increasing the size of the bubbles will make things stand out more.

* Import the numpy package as np.
* Use np.array() to create a numpy array from the list pop. Call this NumPy array np\_pop.
* Double the values in np\_pop setting the value of np\_pop equal to np\_pop \* 2. Because np\_pop is a NumPy array, each array element will be doubled.
* Change the s argument inside [plt.scatter()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html) to be np\_pop instead of pop



# Colors

The code you've written up to now is available in the script.

The next step is making the plot more colorful! To do this, a list col has been created for you. It's a list with a color for each corresponding country, depending on the continent the country is part of.

How did we make the list col you ask? The Gapminder data contains a list continent with the continent each country belongs to. A dictionary is constructed that maps continents onto colors:

dict = {

'Asia':'red',

'Europe':'green',

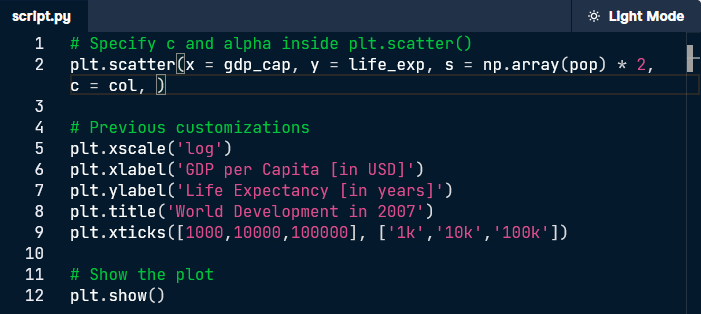
'Africa':'blue',

'Americas':'yellow',

'Oceania':'black'

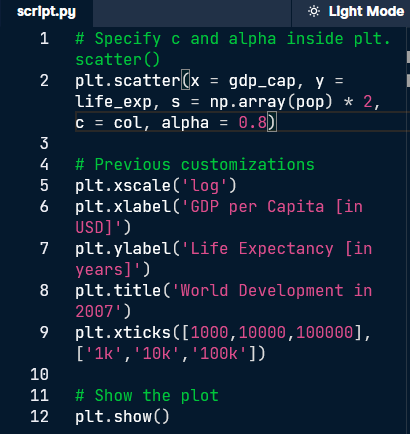
}

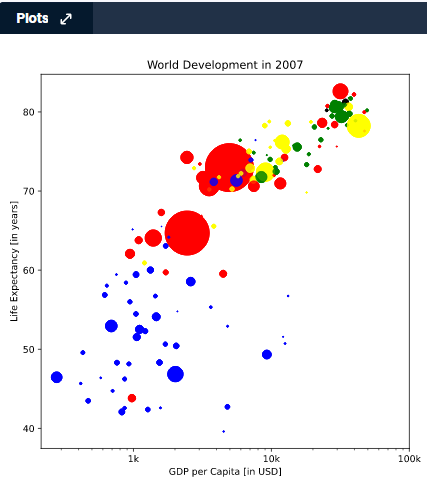
* Add c = col to the arguments of the [plt.scatter()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html) function.
* Change the opacity of the bubbles by setting the alpha argument to 0.8 inside [plt.scatter()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html). Alpha can be set from zero to one, where zero is totally transparent, and one is not at all transparent

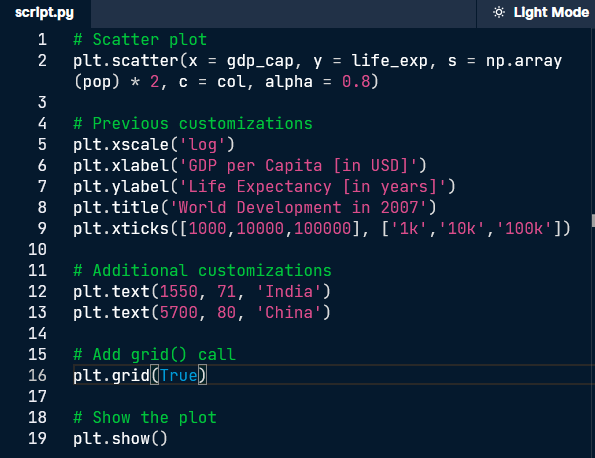


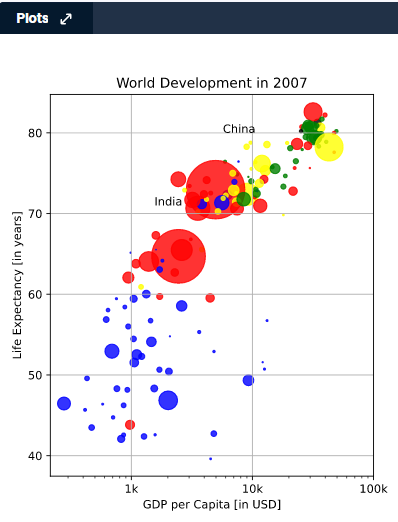


Then add alpha = 0.8





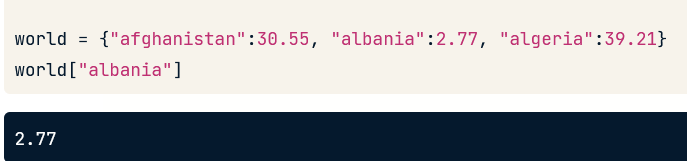


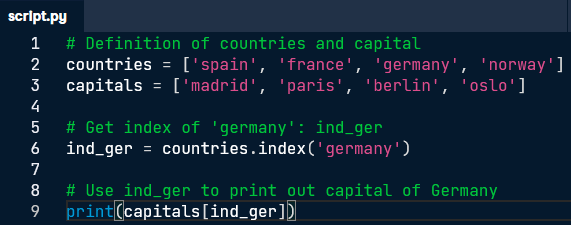




To create the dictionary, you need curly brackets. Next, inside the curly brackets, you have a bunch of what are called key:value pairs. In our case, the keys are the country names, and the values are the corresponding populations.









here is a recipe for creating a dictionary:

my\_dict = {

"key1":"value1",

"key2":"value2",

}

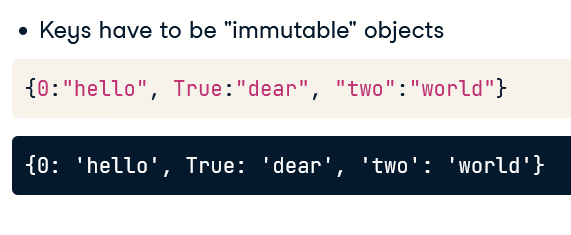








Also, these unique keys in a dictionary should be so-called immutable objects. Basically, the content of immutable objects cannot be changed after they're created. Strings, booleans, integers and floats are immutable objects, but the list for example is mutable, because you can change its contents after it's created. That's why this dictionary, that has all immutable objects as keys, is perfectly valid.



This one, however, that uses a list as a key, is not valid, so we get an error.





27 people live on micro-nation of Sealand; expressed as millions: 0.000027

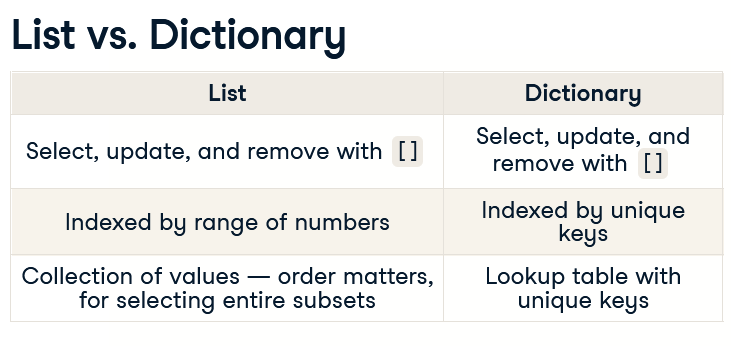


Or



you can also change values, for example, to update the population of sealand to 28. Because each key in a dictionary is unique, Python knows that you're not trying to create a new pair, but want to update the pair that's already in there.



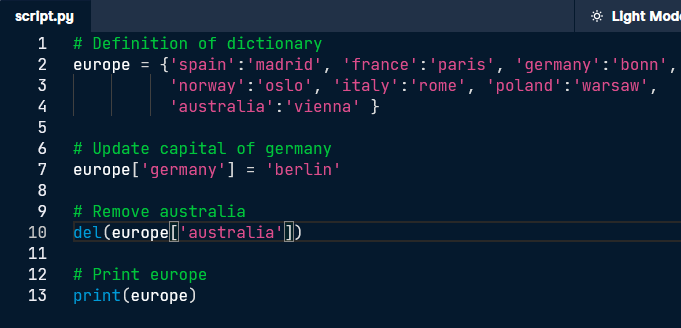


To add a new key-value pair to europe you can use something like this:

europe['iceland'] = 'reykjavik'



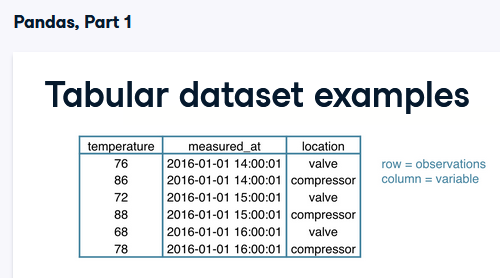




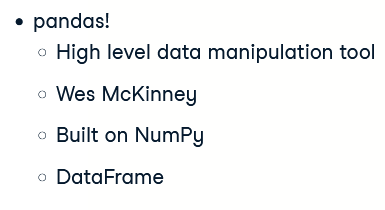
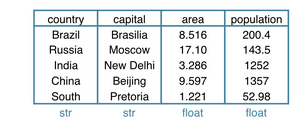


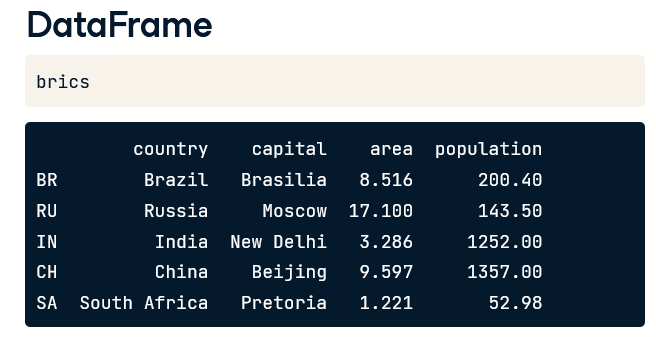




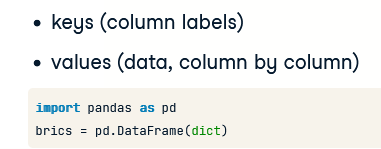


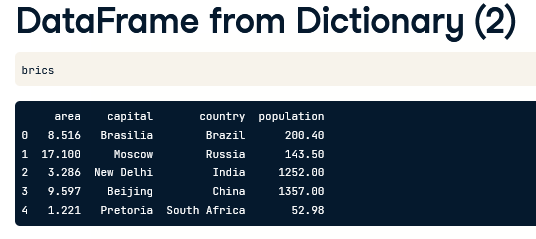


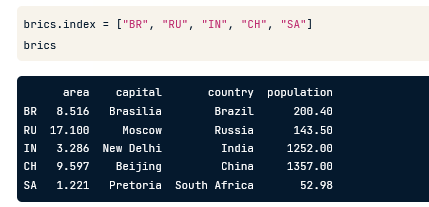
 

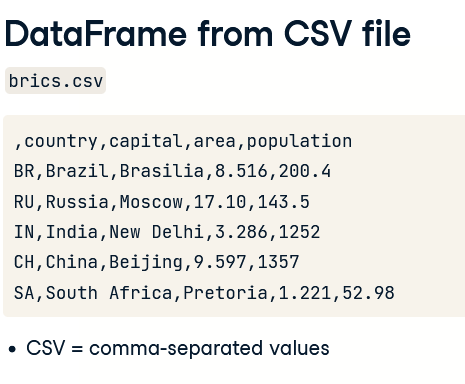


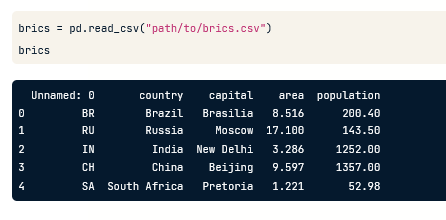


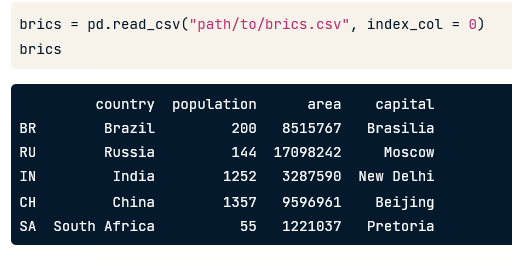












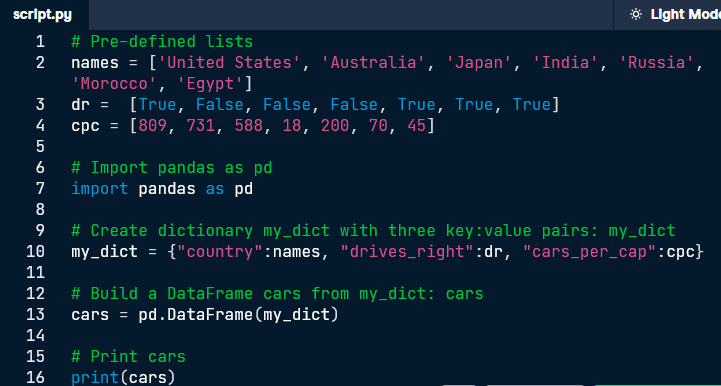
# Dictionary to DataFrame (1)

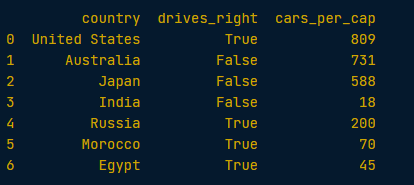
Pandas is an open source library, providing high-performance, easy-to-use data structures and data analysis tools for Python. Sounds promising!

The DataFrame is one of Pandas' most important data structures. It's basically a way to store tabular data where you can label the rows and the columns. One way to build a DataFrame is from a dictionary.

Three lists are defined in the script:

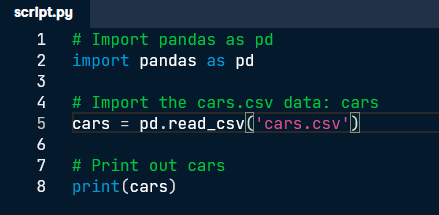
* names, containing the country names for which data is available.
* dr, a list with booleans that tells whether people drive left or right in the corresponding country.
* cpc, the number of motor vehicles per 1000 people in the corresponding country.

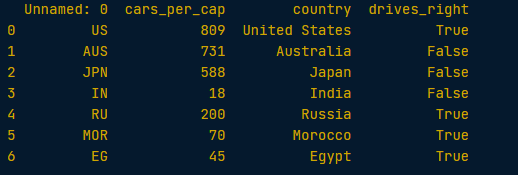




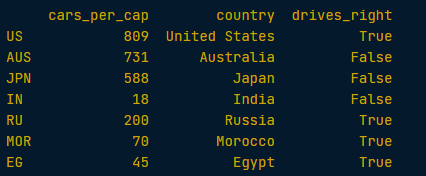


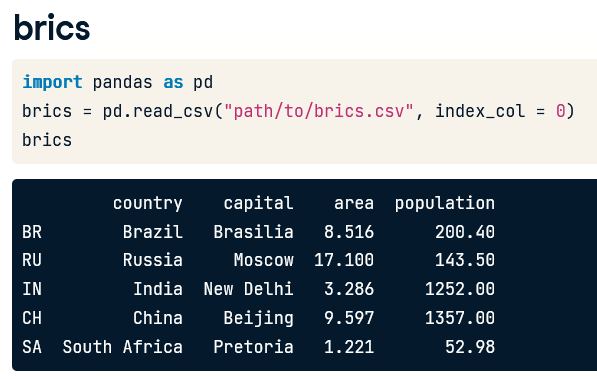


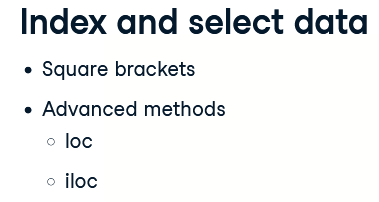


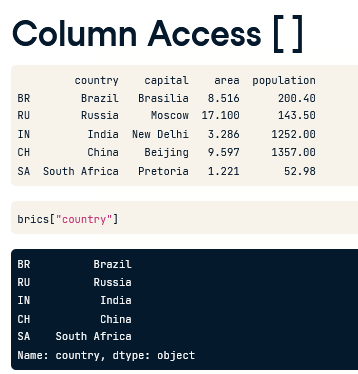












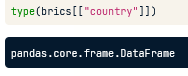


so we're dealing with a Pandas Series here. In a simplified sense, you can think of the Series as a 1-dimensional array that can be labeled, just like the DataFrame. Otherwise put, if you paste together a bunch of Series, you can create a DataFrame.



If you want to select the country column but keep the data in a DataFrame, you'll need double square brackets, like this

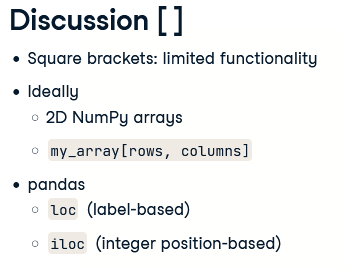




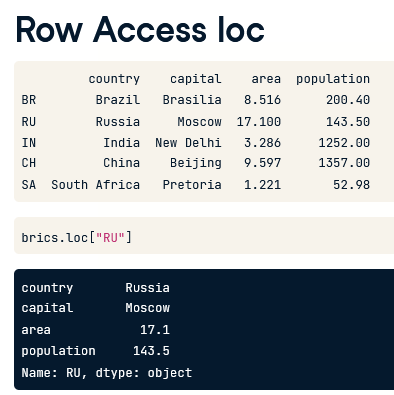


You can perfectly extend this call to select two columns, country and capital, for example. If you look at it from a different angle, you're actually putting a list with column labels inside another set of square brackets, and end up with a 'sub DataFrame', containing only the country and capital columns.

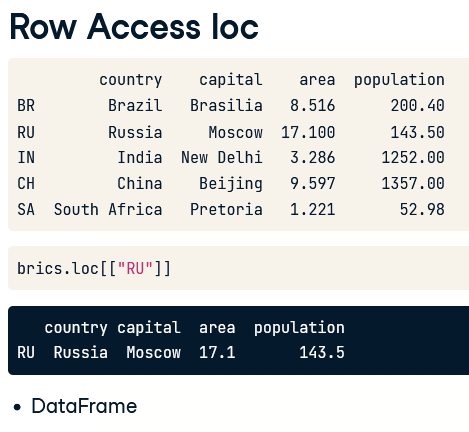




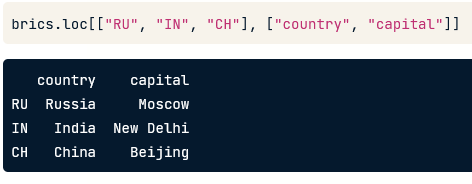
loc is a technique to select parts of your data based on labels, iloc is position based

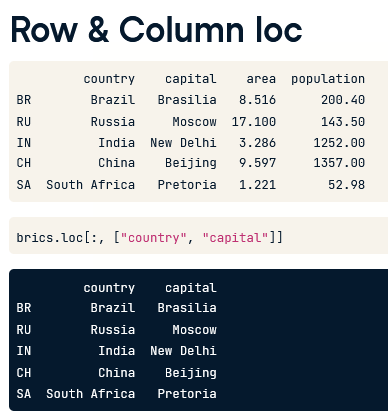


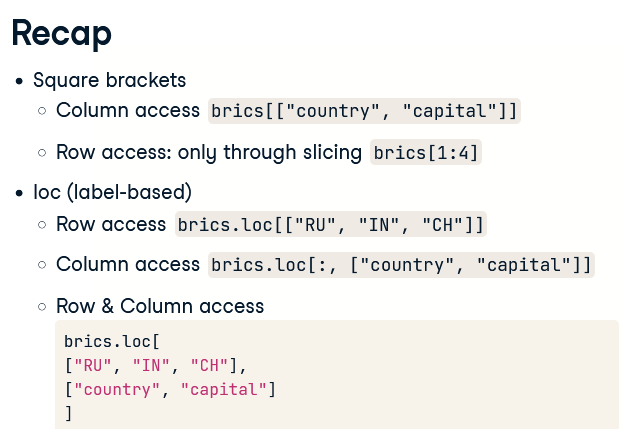






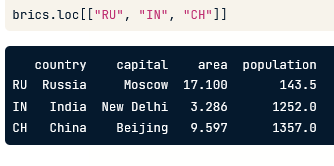


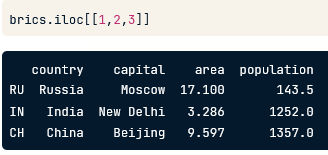


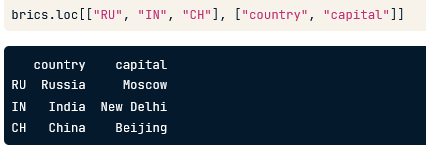


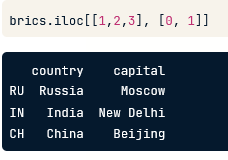
If you want to subset Pandas DataFrames based on their position, or index, you'll need the iloc function.



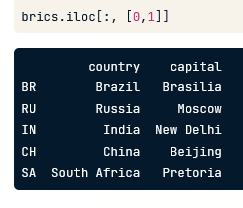










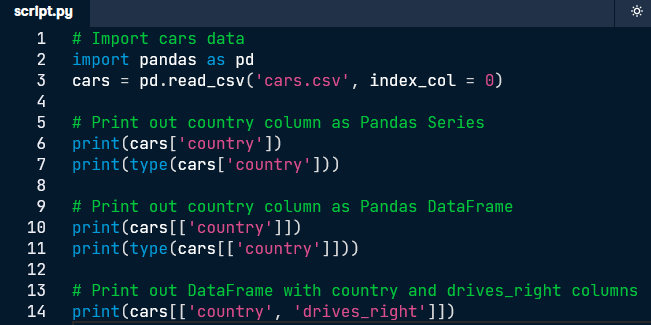


In the sample code, the same cars data is imported from a CSV files as a Pandas DataFrame. To select only the cars\_per\_cap column from cars, you can use:

cars['cars\_per\_cap']

cars[['cars\_per\_cap']]

The single bracket version gives a Pandas Series, the double bracket version gives a Pandas DataFrame.







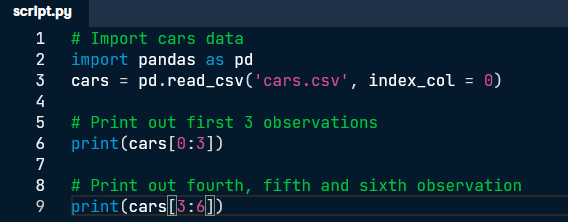


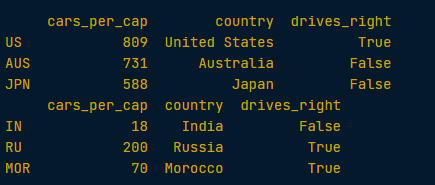
Square brackets can do more than just selecting columns. You can also use them to get rows, or observations, from a DataFrame. The following call selects the first five rows from the cars DataFrame:

cars[0:5]

The result is another DataFrame containing only the rows you specified.

Pay attention: You can only select rows using square brackets if you specify a slice, like 0:4. Also, you're using the integer indexes of the rows here, not the row labels!

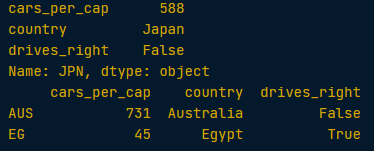




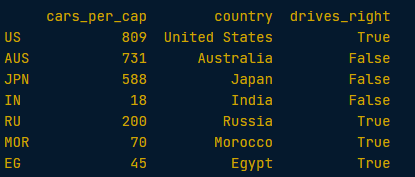
# **loc and iloc (1)**

With [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) and [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) you can do practically any data selection operation on DataFrames you can think of. [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) is label-based, which means that you have to specify rows and columns based on their row and column labels. [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) is integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.

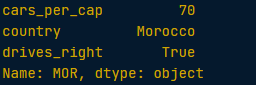




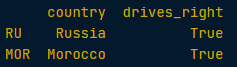












# **loc and iloc (3)**

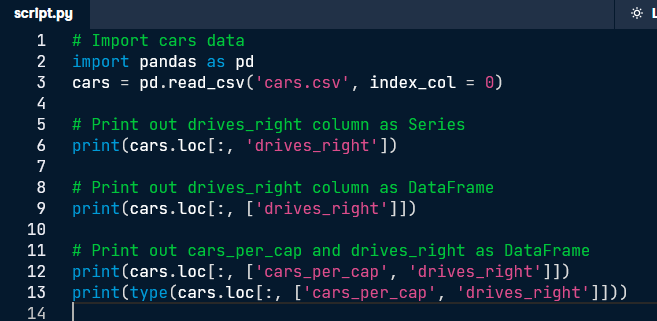
It's also possible to select only columns with [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) and [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing). In both cases, you simply put a slice going from beginning to end in front of the comma:

cars.loc[:, 'country']

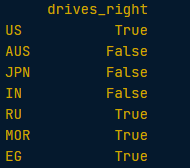
cars.iloc[:, 1]

cars.loc[:, ['country','drives\_right']]

cars.iloc[:, [1, 2]]







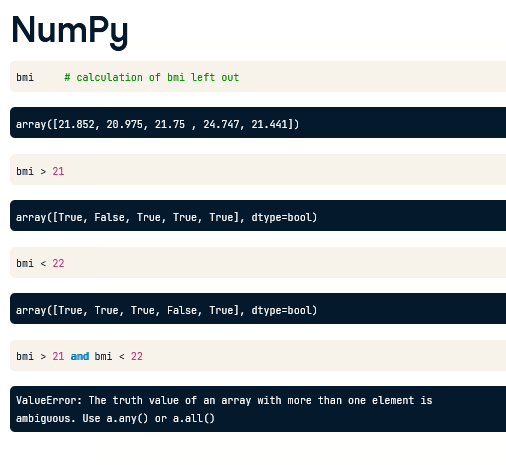


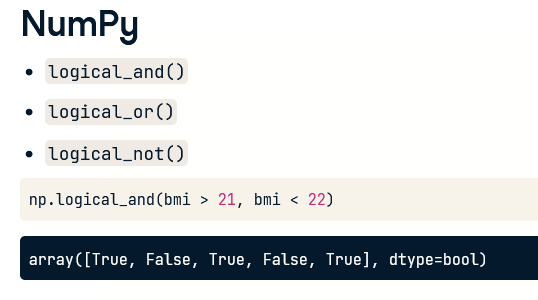


Fewer notes

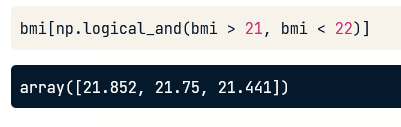
Just try to get through it







To actually select only these bmis from the bmi array, we can use the resulting array of booleans in square brackets.



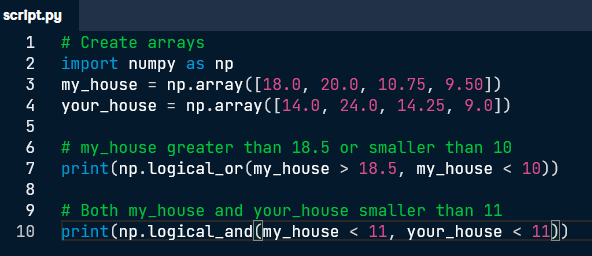
# Boolean operators with NumPy

Before, the operational operators like < and >= worked with NumPy arrays out of the box. Unfortunately, this is not true for the boolean operators and, or, and not.

To use these operators with NumPy, you will need [np.logical\_and()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.logical_and.html), [np.logical\_or()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.logical_or.html) and [np.logical\_not()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.logical_not.html). Here's an example on the my\_house and your\_house arrays from before to give you an idea:

np.logical\_and(my\_house > 13,

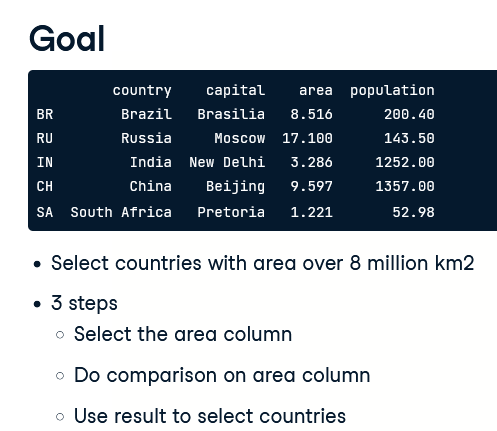
your\_house < 15)

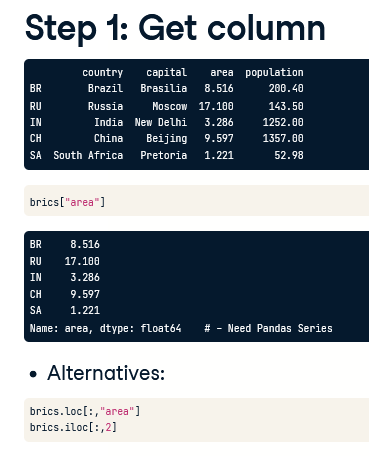


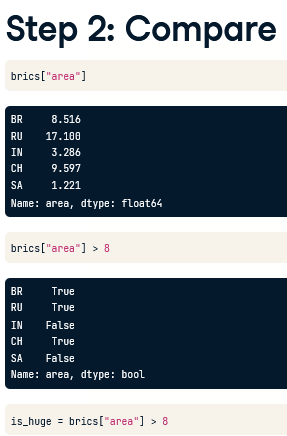


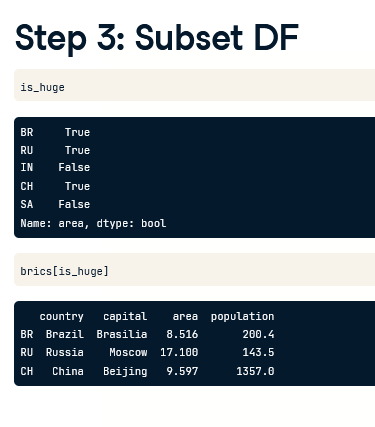




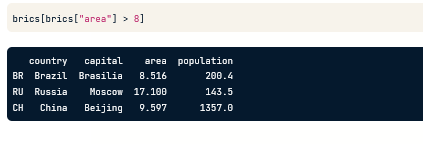


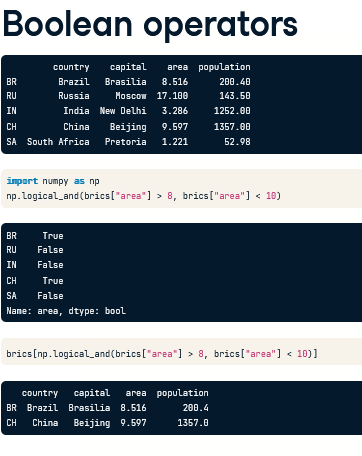


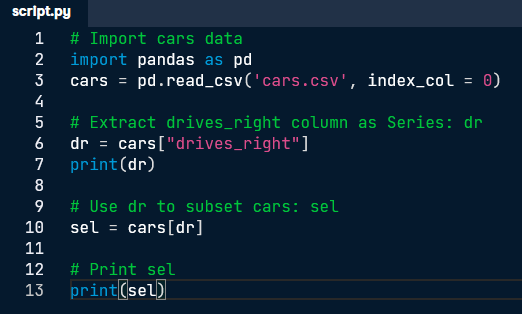


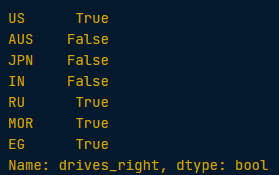


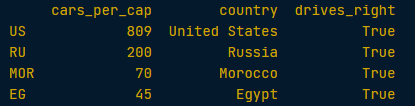
OR



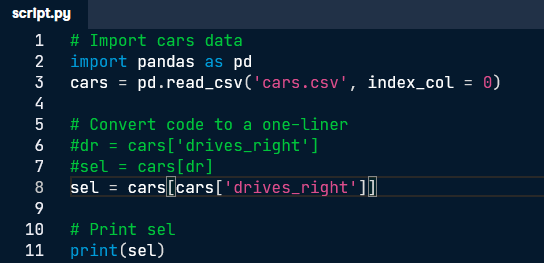


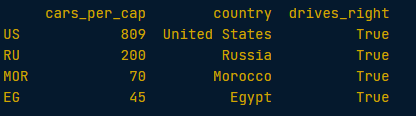




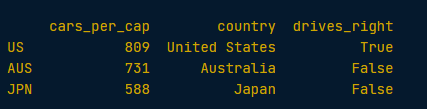


One-line version of the above:







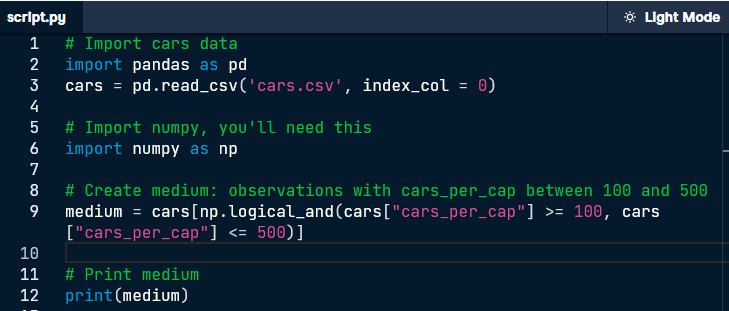


This example selects the observations that have a cars\_per\_cap between 10 and 80. Try out these lines of code step by step to see what's happening.

cpc = cars['cars\_per\_cap']

between = np.logical\_and(cpc > 10, cpc < 80)

medium = cars[between]



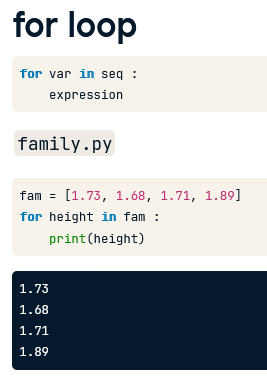


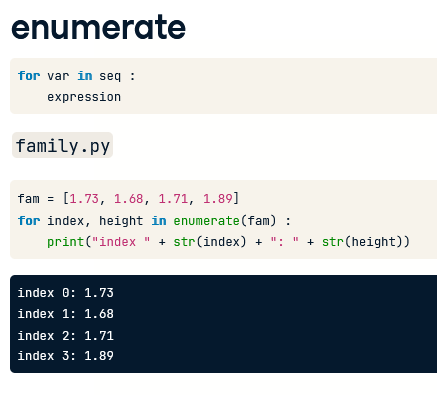


The while loop is not that common, but in some cases it can be very useful. As example, suppose you're numerically calculating a model based on your data. This typically involves taking the same steps over and over again, until the error between your model and your data is below some boundary. When you can reformulate the problem as 'repeating an action until a particular condition is met', a while loop is often the way to go.











# Indexes and values (1)

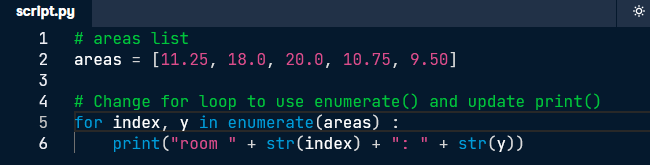
Using a for loop to iterate over a list only gives you access to every list element in each run, one after the other. If you also want to access the index information, so where the list element you're iterating over is located, you can use [enumerate()](https://docs.python.org/3/library/functions.html#enumerate).

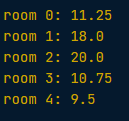
As an example, have a look at how the for loop from the video was converted:

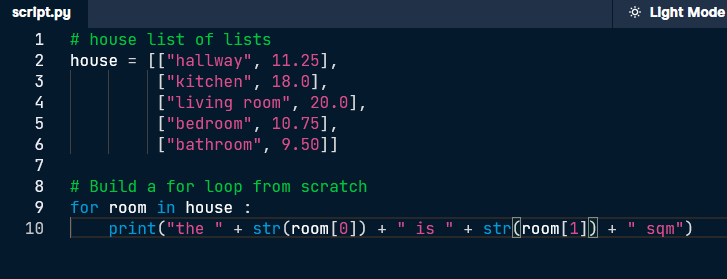
fam = [1.73, 1.68, 1.71, 1.89]

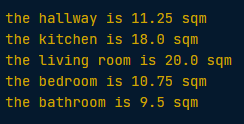
for index, height in enumerate(fam) :

print("person " + str(index) + ": " + str(height))



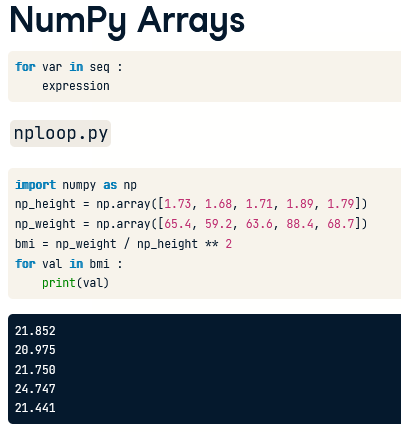


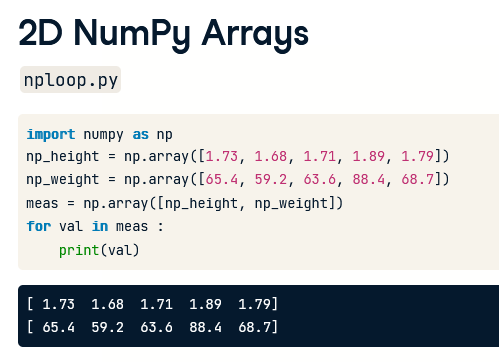


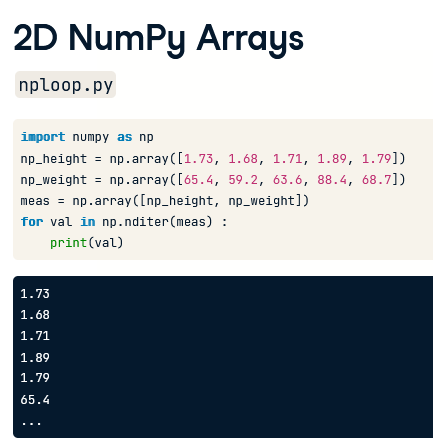


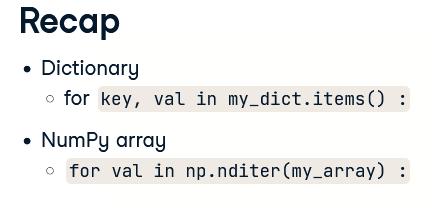


If you have a look at the printout, there's something strange: afghanistan comes first in world, but not in the printout. That's because dictionaries are inherently unordered: the order in which they're iterated over is not fixed, at least in Python 3.5.



‘’ 





dictionaries require a method, NumPy arrays use a function

In Python 3, you need the [items()](https://docs.python.org/3/library/stdtypes.html#dict.items) method to loop over a dictionary:

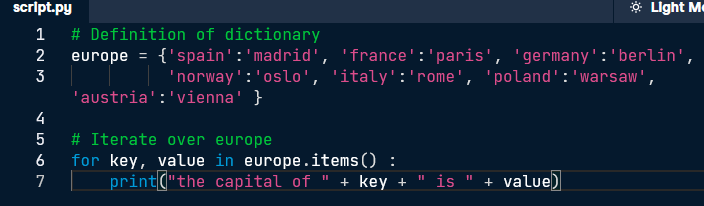
world = { "afghanistan":30.55,

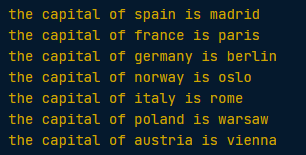
"albania":2.77,

"algeria":39.21 }

for key, value in world.items() :

print(key + " -- " + str(value))





# Loop over NumPy array

If you're dealing with a 1D NumPy array, looping over all elements can be as simple as:

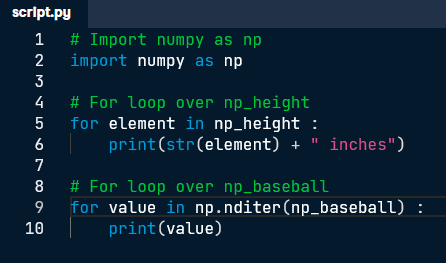
for x in my\_array :

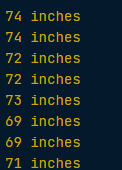
...

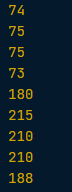
If you're dealing with a 2D NumPy array, it's more complicated. A 2D array is built up of multiple 1D arrays. To explicitly iterate over all separate elements of a multi-dimensional array, you'll need this syntax:

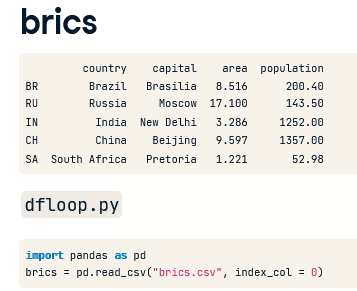
for x in np.nditer(my\_array) :

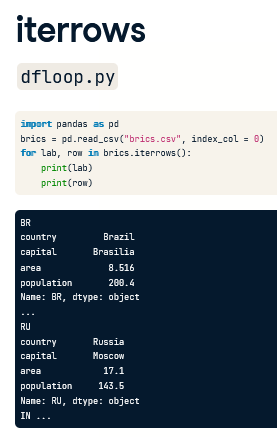
...















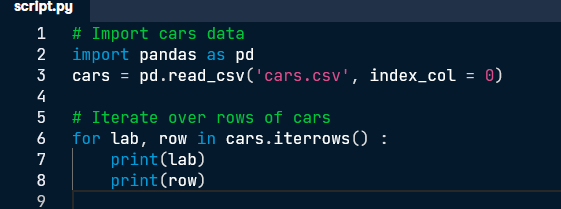


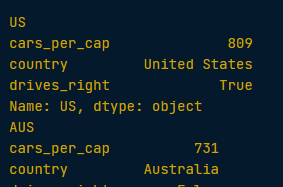
# **Loop over DataFrame (1)**

Iterating over a Pandas DataFrame is typically done with the [iterrows()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iterrows.html) method. Used in a for loop, every observation is iterated over and on every iteration the row label and actual row contents are available:

for lab, row in brics.iterrows() :

...



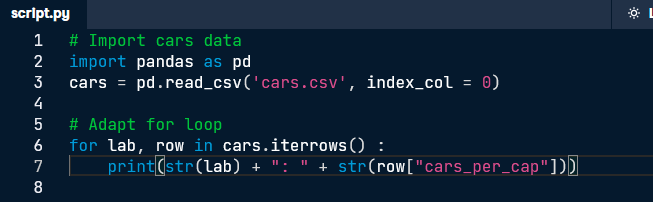


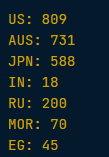
# **Loop over DataFrame (2)**

The row data that's generated by [iterrows()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iterrows.html) on every run is a Pandas Series. This format is not very convenient to print out. Luckily, you can easily select variables from the Pandas Series using square brackets:

for lab, row in brics.iterrows() :

print(row['country'])



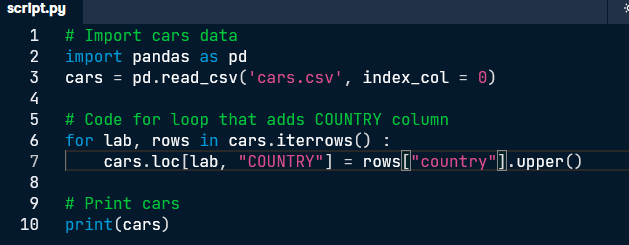


# Add column (1)

In the video, Hugo showed you how to add the length of the country names of the brics DataFrame in a new column:

for lab, row in brics.iterrows() :

brics.loc[lab, "name\_length"] = len(row["country"])





Compare the [iterrows()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iterrows.html) version with the [apply()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html) version to get the same result in the brics DataFrame:

for lab, row in brics.iterrows() :

brics.loc[lab, "name\_length"] = len(row["country"])

brics["name\_length"] = brics["country"].apply(len)

