**Daily XP400**

**Exercise**

**Exercise**

**Motivation for dictionaries**

To see why dictionaries are useful, have a look at the two lists defined in the script. countries contains the names of some European countries. capitals lists the corresponding names of their capital.

**Instructions**

**100 XP**

* Use the [index()](https://docs.python.org/3/library/stdtypes.html#common-sequence-operations) method on countries to find the index of 'germany'. Store this index as ind\_ger.
* Use ind\_ger to access the capital of Germany from the capitals list. Print it out.

**Daily XP500**

**Exercise**

**Exercise**

**Create dictionary**

The countries and capitals lists are again available in the script. It's your job to convert this data to a dictionary where the country names are the keys and the capitals are the corresponding values. As a refresher, here is a recipe for creating a dictionary:

my\_dict = {

"key1":"value1",

"key2":"value2",

}

In this recipe, both the keys and the values are strings. This will also be the case for this exercise.

**Instructions**

**100 XP**

* With the strings in countries and capitals, create a dictionary called europe with 4 key:value pairs. Beware of capitalization! Make sure you use lowercase characters everywhere.
* Print out europe to see if the result is what you expected.

**Daily XP600**

**Exercise**

**Exercise**

**Access dictionary**

If the keys of a dictionary are chosen wisely, accessing the values in a dictionary is easy and intuitive. For example, to get the capital for France from europe you can use:

europe['france']

Here, 'france' is the key and 'paris' the value is returned.

**Instructions**

**100 XP**

* Check out which keys are in europe by calling the [keys()](https://docs.python.org/3/library/stdtypes.html#dict.keys) method on europe. Print out the result.
* Print out the value that belongs to the key 'norway'.

**Daily XP750**

**Exercise**

**Exercise**

**Dictionary Manipulation (1)**

If you know how to access a dictionary, you can also assign a new value to it. To add a new key-value pair to europe you can use something like this:

europe['iceland'] = 'reykjavik'

**Instructions**

**100 XP**

* Add the key 'italy' with the value 'rome' to europe.
* To assert that 'italy' is now a key in europe, print out 'italy' in europe.
* Add another key:value pair to europe: 'poland' is the key, 'warsaw' is the corresponding value.
* Print out europe.

**Daily XP850**

**Exercise**

**Exercise**

**Dictionary Manipulation (2)**

Somebody thought it would be funny to mess with your accurately generated dictionary. An adapted version of the europe dictionary is available in the script.

Can you clean up? Do not do this by adapting the definition of europe, but by adding Python commands to the script to update and remove key:value pairs.

**Instructions**

**100 XP**

* The capital of Germany is not 'bonn'; it's 'berlin'. Update its value.
* Australia is not in Europe, Austria is! Remove the key 'australia' from europe.
* Print out europe to see if your cleaning work paid off.

**Daily XP950**

**Exercise**

**Exercise**

**Dictionariception**

Remember lists? They could contain anything, even other lists. Well, for dictionaries the same holds. Dictionaries can contain key:value pairs where the values are again dictionaries.

As an example, have a look at the script where another version of europe - the dictionary you've been working with all along - is coded. The keys are still the country names, but the values are dictionaries that contain more information than just the capital.

It's perfectly possible to chain square brackets to select elements. To fetch the population for Spain from europe, for example, you need:

europe['spain']['population']

**Instructions**

**100 XP**

* Use chained square brackets to select and print out the capital of France.
* Create a dictionary, named data, with the keys 'capital' and 'population'. Set them to 'rome' and 59.83, respectively.
* Add a new key-value pair to europe; the key is 'italy' and the value is data, the dictionary you just built.

**Daily XP1100**

**Exercise**

**Exercise**

**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.

In the exercises that follow you will be working with vehicle data from different countries. Each observation corresponds to a country and the columns give information about the number of vehicles per capita, whether people drive left or right, and so on.

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.

Each dictionary key is a column label and each value is a list which contains the column elements.

**Instructions**

**100 XP**

* Import pandas as pd.
* Use the pre-defined lists to create a dictionary called my\_dict. There should be three key value pairs:
  + key 'country' and value names.
  + key 'drives\_right' and value dr.
  + key 'cars\_per\_cap' and value cpc.
* Use [pd.DataFrame()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html) to turn your dict into a DataFrame called cars.
* Print out cars and see how beautiful it is.

**Daily XP1200**

**Exercise**

**Exercise**

**Dictionary to DataFrame (2)**

The Python code that solves the previous exercise is included in the script. Have you noticed that the row labels (i.e. the labels for the different observations) were automatically set to integers from 0 up to 6?

To solve this a list row\_labels has been created. You can use it to specify the row labels of the cars DataFrame. You do this by setting the index attribute of cars, that you can access as cars.index.

**Instructions**

**100 XP**

* Hit *Run Code* to see that, indeed, the row labels are not correctly set.
* Specify the row labels by setting cars.index equal to row\_labels.
* Print out cars again and check if the row labels are correct this time.

**Daily XP1300**

**Exercise**

**Exercise**

**CSV to DataFrame (1)**

Putting data in a dictionary and then building a DataFrame works, but it's not very efficient. What if you're dealing with millions of observations? In those cases, the data is typically available as files with a regular structure. One of those file types is the CSV file, which is short for "comma-separated values".

To import CSV data into Python as a Pandas DataFrame you can use [read\_csv()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html).

Let's explore this function with the same cars data from the previous exercises. This time, however, the data is available in a CSV file, named cars.csv. It is available in your current working directory, so the path to the file is simply 'cars.csv'.

**Instructions**

**100 XP**

* To import CSV files you still need the pandas package: import it as pd.
* Use [pd.read\_csv()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html) to import cars.csv data as a DataFrame. Store this DataFrame as cars.
* Print out cars. Does everything look OK?

**Daily XP1550**

**Exercise**

**Exercise**

**Square Brackets (1)**

In the video, you saw that you can index and select Pandas DataFrames in many different ways. The simplest, but not the most powerful way, is to use square brackets.

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.

**Instructions**

**100 XP**

* Use single square brackets to print out the country column of cars as a Pandas Series.
* Use double square brackets to print out the country column of cars as a Pandas DataFrame.
* Use double square brackets to print out a DataFrame with both the country and drives\_right columns of cars, in this order.

**Daily XP1650**

**Exercise**

**Exercise**

**Square Brackets (2)**

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!

**Instructions**

**100 XP**

* Select the first 3 observations from cars and print them out.
* Select the fourth, fifth and sixth observation, corresponding to row indexes 3, 4 and 5, and print them out.

**Daily XP100**

**Exercise**

**Exercise**

**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.

Try out the following commands in the IPython Shell to experiment 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) to select observations. Each pair of commands here gives the same result.

cars.loc['RU']

cars.iloc[4]

cars.loc[['RU']]

cars.iloc[[4]]

cars.loc[['RU', 'AUS']]

cars.iloc[[4, 1]]

As before, code is included that imports the cars data as a Pandas DataFrame.

**Instructions**

**100 XP**

* Use [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) to select the observation corresponding to Japan as a Series. The label of this row is JPN, the index is 2. Make sure to print the resulting Series.
* Use [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) to select the observations for Australia and Egypt as a DataFrame. You can find out about the labels/indexes of these rows by inspecting cars in the IPython Shell. Make sure to print the resulting DataFrame.

**Daily XP200**

**Exercise**

**Exercise**

**loc and iloc (2)**

loc and [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) also allow you to select both rows and columns from a DataFrame. To experiment, try out the following commands in the IPython Shell. Again, paired commands produce the same result.

cars.loc['IN', 'cars\_per\_cap']

cars.iloc[3, 0]

cars.loc[['IN', 'RU'], 'cars\_per\_cap']

cars.iloc[[3, 4], 0]

cars.loc[['IN', 'RU'], ['cars\_per\_cap', 'country']]

cars.iloc[[3, 4], [0, 1]]

**Instructions**

**100 XP**

* Print out the drives\_right value of the row corresponding to Morocco (its row label is MOR)
* Print out a sub-DataFrame, containing the observations for Russia and Morocco and the columns country and drives\_right.

**Daily XP300**

**Exercise**

**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]]

**Instructions**

**100 XP**

* Print out the drives\_right column as a Series using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).
* Print out the drives\_right column as a DataFrame using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).
* Print out both the cars\_per\_cap and drives\_right column as a DataFrame using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).

**Daily XP300**

**Exercise**

**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]]

**Instructions**

**100 XP**

* Print out the drives\_right column as a Series using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).
* Print out the drives\_right column as a DataFrame using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).
* Print out both the cars\_per\_cap and drives\_right column as a DataFrame using [loc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing) or [iloc](https://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing).

**Daily XP550**

**Exercise**

**Exercise**

**Greater and less than**

In the video, Hugo also talked about the less than and greater than signs, < and > in Python. You can combine them with an equals sign: <= and >=. Pay attention: <= is valid syntax, but =< is not.

All Python expressions in the following code chunk evaluate to True:

3 < 4

3 <= 4

"alpha" <= "beta"

Remember that for string comparison, Python determines the relationship based on alphabetical order.

**Instructions**

**100 XP**

* Write Python expressions, wrapped in a [print()](https://docs.python.org/3/library/functions.html#print) function, to check whether:
  + x is greater than or equal to -10. x has already been defined for you.
  + "test" is less than or equal to y. y has already been defined for you.
  + True is greater than False.

**Compare arrays**

Out of the box, you can also use comparison operators with NumPy arrays.

Remember areas, the list of area measurements for different rooms in your house from *Introduction to Python*? This time there's two NumPy arrays: my\_house and your\_house. They both contain the areas for the kitchen, living room, bedroom and bathroom in the same order, so you can compare them.

**Instructions**

**100 XP**

Using comparison operators, generate boolean arrays that answer the following questions:

* Which areas in my\_house are greater than or equal to 18?
* You can also compare two NumPy arrays element-wise. Which areas in my\_house are smaller than the ones in your\_house?
* Make sure to wrap both commands in a [print()](https://docs.python.org/3/library/functions.html#print) statement so that you can inspect the output!

**Daily XP800**

**Exercise**

**Exercise**

**and, or, not (1)**

A boolean is either 1 or 0, True or False. With boolean operators such as and, or and not, you can combine these booleans to perform more advanced queries on your data.

In the sample code, two variables are defined: my\_kitchen and your\_kitchen, representing areas.

**Instructions**

**100 XP**

* Write Python expressions, wrapped in a [print()](https://docs.python.org/3/library/functions.html#print) function, to check whether:
  + my\_kitchen is bigger than 10 and smaller than 18.
  + my\_kitchen is smaller than 14 or bigger than 17.
  + double the area of my\_kitchen is smaller than triple the area of your\_kitchen.

**Daily XP900**

**Exercise**

**Exercise**

**and, or, not (2)**

To see if you completely understood the boolean operators, have a look at the following piece of Python code:

x = 8

y = 9

not(not(x < 3) and not(y > 14 or y > 10))

What will the result be if you execute these three commands in the IPython Shell?

*NB: Notice that not has a higher priority than and and or, it is executed first.*

**Instructions**

**50 XP**

**Possible Answers**

* 

True

* 

False

* 

Running these commands will result in an error.

**Daily XP950**

**Exercise**

**Exercise**

**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)

**Instructions**

**100 XP**

* Generate boolean arrays that answer the following questions:
* Which areas in my\_house are greater than 18.5 or smaller than 10?
* Which areas are smaller than 11 in both my\_house and your\_house? Make sure to wrap both commands in [print()](https://docs.python.org/3/library/functions.html#print) statement, so that you can inspect the output.

**Daily XP1100**

**Exercise**

**Exercise**

**Warmup**

To experiment with if and else a bit, have a look at this code sample:

area = 10.0

if(area < 9) :

print("small")

elif(area < 12) :

print("medium")

else :

print("large")

What will the output be if you run this piece of code in the IPython Shell?

**Instructions**

**50 XP**

**Possible Answers**

* 

small

* 

medium

* 

large

* 

The syntax is incorrect; this code will produce an error.

**xercise**

**if**

It's time to take a closer look around in your house.

Two variables are defined in the sample code: room, a string that tells you which room of the house we're looking at, and area, the area of that room.

**Instructions**

**100 XP**

* Examine the if statement that prints out "looking around in the kitchen." if room equals "kit".
* Write another if statement that prints out "big place!" if area is greater than 15.

**Daily XP1250**

**Exercise**

**Exercise**

**Add else**

In the script, the if construct for room has been extended with an else statement so that "looking around elsewhere." is printed if the condition room == "kit" evaluates to False.

Can you do a similar thing to add more functionality to the if construct for area?

**Instructions**

**100 XP**

Add an else statement to the second control structure so that "pretty small." is printed out if area > 15 evaluates to False.

**Daily XP1350**

**Exercise**

**Exercise**

**Customize further: elif**

It's also possible to have a look around in the bedroom. The sample code contains an elif part that checks if room equals "bed". In that case, "looking around in the bedroom." is printed out.

It's up to you now! Make a similar addition to the second control structure to further customize the messages for different values of area.

**Instructions**

**100 XP**

Add an elif to the second control structure such that "medium size, nice!" is printed out if area is greater than 10.

**Daily XP1500**

**Exercise**

**Exercise**

**Driving right (1)**

Remember that cars dataset, containing the cars per 1000 people (cars\_per\_cap) and whether people drive right (drives\_right) for different countries (country)? The code that imports this data in CSV format into Python as a DataFrame is included in the script.

In the video, you saw a step-by-step approach to filter observations from a DataFrame based on boolean arrays. Let's start simple and try to find all observations in cars where drives\_right is True.

drives\_right is a boolean column, so you'll have to extract it as a Series and then use this boolean Series to select observations from cars.

**Instructions**

**100 XP**

* Extract the drives\_right column *as a Pandas Series* and store it as dr.
* Use dr, a boolean Series, to subset the cars DataFrame. Store the resulting selection in sel.
* Print sel, and assert that drives\_right is True for all observations.

**Daily XP1600**

**Exercise**

**Exercise**

**Driving right (2)**

The code in the previous example worked fine, but you actually unnecessarily created a new variable dr. You can achieve the same result without this intermediate variable. Put the code that computes dr straight into the square brackets that select observations from cars.

**Instructions**

**100 XP**

Convert the code to a one-liner that calculates the variable sel as before.

**Daily XP100**

**Exercise**

**Exercise**

**Cars per capita (1)**

Let's stick to the cars data some more. This time you want to find out which countries have a high *cars per capita* figure. In other words, in which countries do many people have a car, or maybe multiple cars.

Similar to the previous example, you'll want to build up a boolean Series, that you can then use to subset the cars DataFrame to select certain observations. If you want to do this in a one-liner, that's perfectly fine!

**Instructions**

**100 XP**

* Select the cars\_per\_cap column from cars as a Pandas Series and store it as cpc.
* Use cpc in combination with a comparison operator and 500. You want to end up with a boolean Series that's True if the corresponding country has a cars\_per\_cap of more than 500 and False otherwise. Store this boolean Series as many\_cars.
* Use many\_cars to subset cars, similar to what you did before. Store the result as car\_maniac.
* Print out car\_maniac to see if you got it right.

**Daily XP200**

**Exercise**

**Exercise**

**Cars per capita (2)**

Remember about [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), the NumPy variants of the and, or and not operators? You can also use them on Pandas Series to do more advanced filtering operations.

Take this example that 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]

**Instructions**

**100 XP**

* Use the code sample provided to create a DataFrame medium, that includes all the observations of cars that have a cars\_per\_cap between 100 and 500.
* Print out medium.

**Daily XP50**

**Exercise**

**Exercise**

**while: warming up**

The while loop is like a repeated if statement. The code is executed over and over again, as long as the condition is True. Have another look at its recipe.

while condition :

expression

Can you tell how many printouts the following while loop will do?

x = 1

while x < 4 :

print(x)

x = x + 1

**Instructions**

**50 XP**

**Possible Answers**

* 

0

* 

1

* 

2

* 

3

* 

4

**Daily XP50**

**Exercise**

**Exercise**

**Basic while loop**

Below you can find the example from the video where the error variable, initially equal to 50.0, is divided by 4 and printed out on every run:

error = 50.0

while error > 1 :

error = error / 4

print(error)

This example will come in handy, because it's time to build a while loop yourself! We're going to code a while loop that implements a very basic control system for an [inverted pendulum](https://en.wikipedia.org/wiki/Inverted_pendulum). If there's an offset from standing perfectly straight, the while loop will incrementally fix this offset.

Note that if your while loop takes too long to run, you might have made a mistake. In particular, remember to **indent** the contents of the loop using four spaces or auto-indentation!

**Instructions**

**100 XP**

* Create the variable offset with an initial value of 8.
* Code a while loop that keeps running as long as offset is not equal to 0. Inside the while loop:
  + Print out the sentence "correcting...".
  + Next, decrease the value of offset by 1. You can do this with offset = offset - 1.
  + Finally, still within your loop, print out offset so you can see how it changes.

**Daily XP150**

**Exercise**

**Exercise**

**Add conditionals**

The while loop that corrects the offset is a good start, but what if offset is negative? You can try to run the following code where offset is initialized to -6:

# Initialize offset

offset = -6

# Code the while loop

while offset != 0 :

print("correcting...")

offset = offset - 1

print(offset)

but your session will be disconnected. The while loop will never stop running, because offset will be further decreased on every run. offset != 0 will never become False and the while loop continues forever.

Fix things by putting an if-else statement inside the while loop. If your code is still taking too long to run, you probably made a mistake!

**Instructions**

**100 XP**

* **Inside** the while loop, complete the if-else statement:
  + If offset is greater than zero, you should decrease offset by 1.
  + Else, you should increase offset by 1.
* If you've coded things correctly, hitting *Submit Answer* should work this time.

*If your code is still taking too long to run (or your session is expiring), you probably made a mistake. Check your code and make sure that the statement offset != 0 will eventually evaluate to FALSE!*

**Daily XP300**

**Exercise**

**Exercise**

**Loop over a list**

Have another look at the for loop that Hugo showed in the video:

fam = [1.73, 1.68, 1.71, 1.89]

for height in fam :

print(height)

As usual, you simply have to indent the code with 4 spaces to tell Python which code should be executed in the for loop.

The areas variable, containing the area of different rooms in your house, is already defined.

**Instructions**

**100 XP**

Write a for loop that iterates over all elements of the areas list and prints out every element separately.

**Daily XP400**

**Exercise**

**Exercise**

**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))

**Instructions**

**100 XP**

* Adapt the for loop in the sample code to use [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) and use two iterator variables.
* Update the print() statement so that on each run, a line of the form "room x: y" should be printed, where x is the index of the list element and y is the actual list element, i.e. the area. Make sure to print out this exact string, with the correct spacing.

**Daily XP500**

**Exercise**

**Exercise**

**Indexes and values (2)**

For non-programmer folks, room 0: 11.25 is strange. Wouldn't it be better if the count started at 1?

**Instructions**

**100 XP**

Adapt the [print()](https://docs.python.org/3/library/functions.html#print) function in the for loop so that the first printout becomes "room 1: 11.25", the second one "room 2: 18.0" and so on.

**Daily XP750**

**Exercise**

**Exercise**

**Loop over dictionary**

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))

Remember the europe dictionary that contained the names of some European countries as key and their capitals as corresponding value? Go ahead and write a loop to iterate over it!

**Instructions**

**100 XP**

Write a for loop that goes through each key:value pair of europe. On each iteration, "the capital of x is y" should be printed out, where x is the key and y is the value of the pair.

**Daily XP750**

**Exercise**

**Exercise**

**Loop over dictionary**

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))

Remember the europe dictionary that contained the names of some European countries as key and their capitals as corresponding value? Go ahead and write a loop to iterate over it!

**Instructions**

**100 XP**

Write a for loop that goes through each key:value pair of europe. On each iteration, "the capital of x is y" should be printed out, where x is the key and y is the value of the pair.

**Daily XP850**

**Exercise**

**Exercise**

**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) :

...

Two NumPy arrays that you might recognize from the intro course are available in your Python session: np\_height, a NumPy array containing the heights of Major League Baseball players, and np\_baseball, a 2D NumPy array that contains both the heights (first column) and weights (second column) of those players.

**Instructions**

**100 XP**

* Import the numpy package under the local alias np.
* Write a for loop that iterates over all elements in np\_height and prints out "x inches" for each element, where x is the value in the array.
* Write a for loop that visits every element of the np\_baseball array and prints it out.
* **Daily XP1000**
* **Exercise**
* **Exercise**
* **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() :
* ...
* In this and the following exercises you will be working on the cars DataFrame. It contains information on the cars per capita and whether people drive right or left for seven countries in the world.
* **Instructions**
* **100 XP**
* Write a for loop that iterates over the rows of cars and on each iteration perform two [print()](https://docs.python.org/3/library/functions.html#print) calls: one to print out the row label and one to print out all of the rows contents.

**Daily XP1200**

**Exercise**

**Exercise**

**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"])

You can do similar things on the cars DataFrame.

**Instructions**

**100 XP**

* Use a for loop to add a new column, named COUNTRY, that contains a uppercase version of the country names in the "country" column. You can use the string method [upper()](https://docs.python.org/3/library/stdtypes.html#str.upper) for this.
* To see if your code worked, print ou

**Daily XP100**

**Exercise**

**Exercise**

**Add column (2)**

Using [iterrows()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iterrows.html) to iterate over every observation of a Pandas DataFrame is easy to understand, but not very efficient. On every iteration, you're creating a new Pandas Series.

If you want to add a column to a DataFrame by calling a function on another column, the [iterrows()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iterrows.html) method in combination with a for loop is not the preferred way to go. Instead, you'll want to use [apply()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html).

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)

We can do a similar thing to call the [upper()](https://docs.python.org/3/library/stdtypes.html#str.upper) method on every name in the country column. However, [upper()](https://docs.python.org/3/library/stdtypes.html#str.upper) is a **method**, so we'll need a slightly different approach:

**Instructions**

**100 XP**

* Replace the for loop with a one-liner that uses .apply(str.upper). The call should give the same result: a column COUNTRY should be added to cars, containing an uppercase version of the country names.
* As usual, print out cars to see the fruits of your hard labor

**Daily XP250**

**Exercise**

**Exercise**

**Random float**

Randomness has many uses in science, art, statistics, cryptography, gaming, gambling, and other fields. You're going to use randomness to simulate a game.

All the functionality you need is contained in the random package, a sub-package of numpy. In this exercise, you'll be using two functions from this package:

* [seed()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html): sets the random seed, so that your results are reproducible between simulations. As an argument, it takes an integer of your choosing. If you call the function, no output will be generated.
* [rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html): if you don't specify any arguments, it generates a random float between zero and one.

**Instructions**

**100 XP**

* Import numpy as np.
* Use [seed()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html) to set the seed; as an argument, pass 123.
* Generate your first random float with [rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html) and print it out.

**Random float**

Randomness has many uses in science, art, statistics, cryptography, gaming, gambling, and other fields. You're going to use randomness to simulate a game.

All the functionality you need is contained in the random package, a sub-package of numpy. In this exercise, you'll be using two functions from this package:

* [seed()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html): sets the random seed, so that your results are reproducible between simulations. As an argument, it takes an integer of your choosing. If you call the function, no output will be generated.
* [rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html): if you don't specify any arguments, it generates a random float between zero and one.

**Instructions**

**100 XP**

* Import numpy as np.
* Use [seed()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html) to set the seed; as an argument, pass 123.
* Generate your first random float with [rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html) and print it out.

**Daily XP350**

**Exercise**

**Exercise**

**Roll the dice**

In the previous exercise, you used [rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html), that generates a random float between 0 and 1.

As Hugo explained in the video you can just as well use [randint()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.randint.html), also a function of the random package, to generate integers randomly. The following call generates the integer 4, 5, 6 or 7 randomly. **8 is not included**.

import numpy as np

np.random.randint(4, 8)

NumPy has already been imported as np and a seed has been set. Can you roll some dice?

**Instructions**

**100 XP**

* Use [randint()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.randint.html) with the appropriate arguments to randomly generate the integer 1, 2, 3, 4, 5 or 6. This simulates a dice. Print it out.
* Repeat the outcome to see if the second throw is different. Again, print out the result.

**Daily XP450**

**Exercise**

**Exercise**

**Determine your next move**

In the Empire State Building bet, your next move depends on the number of eyes you throw with the dice. We can perfectly code this with an if-elif-else construct!

The sample code assumes that you're currently at step 50. Can you fill in the missing pieces to finish the script? numpy is already imported as np and the seed has been set to 123, so you don't have to worry about that anymore.

**Instructions**

**100 XP**

* Roll the dice. Use [randint()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.randint.html) to create the variable dice.
* Finish the if-elif-else construct by replacing \_\_\_:
* If dice is 1 or 2, you go one step down.
* if dice is 3, 4 or 5, you go one step up.
* Else, you throw the dice again. The number of eyes is the number of steps you go up.
* Print out dice and step. Given the value of dice, was step updated correctly?

**Daily XP600**

**Exercise**

**Exercise**

**The next step**

Before, you have already written Python code that determines the next step based on the previous step. Now it's time to put this code inside a for loop so that we can simulate a random walk.

numpy has been imported as np.

**Instructions**

**100 XP**

* Make a list random\_walk that contains the first step, which is the integer 0.
* Finish the for loop:
* The loop should run 100 times.
* On each iteration, set step equal to the last element in the random\_walk list. You can use the index -1 for this.
* Next, let the if-elif-else construct update step for you.
* The code that appends step to random\_walk is already coded.
* Print out random\_walk.

**Daily XP800**

**Exercise**

**Exercise**

**Visualize the walk**

Let's visualize this random walk! Remember how you could use matplotlib to build a line plot?

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.show()

The first list you pass is mapped onto the x axis and the second list is mapped onto the y axis.

If you pass only one argument, Python will know what to do and will use the index of the list to map onto the x axis, and the values in the list onto the y axis.

**Instructions**

**100 XP**

Add some lines of code after the for loop:

* Import matplotlib.pyplot as plt.
* Use [plt.plot()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html) to plot random\_walk.
* Finish off with [plt.show()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.show.html) to actually display the plot.

**Daily XP950**

**Exercise**

**Exercise**

**Simulate multiple walks**

A single random walk is one thing, but that doesn't tell you if you have a good chance at winning the bet.

To get an idea about how big your chances are of reaching 60 steps, you can repeatedly simulate the random walk and collect the results. That's exactly what you'll do in this exercise.

The sample code already sets you off in the right direction. Another for loop is wrapped around the code you already wrote. It's up to you to add some bits and pieces to make sure all of the results are recorded correctly.

**Note: Don't change anything about the initialization of all\_walks that is given. Setting any number inside the list will cause the exercise to crash!**

**Instructions**

**100 XP**

* Fill in the specification of the for loop so that the random walk is simulated 10 times.
* After the random\_walk array is entirely populated, append the array to the all\_walks list.
* Finally, after the top-level for loop, print out all\_walks.

**Daily XP1050**

**Exercise**

**Exercise**

**Visualize all walks**

all\_walks is a list of lists: every sub-list represents a single random walk. If you convert this list of lists to a NumPy array, you can start making interesting plots! matplotlib.pyplot is already imported as plt.

The nested for loop is already coded for you - don't worry about it. For now, focus on the code that comes after this for loop.

**Instructions**

**100 XP**

* Use [np.array()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.array.html) to convert all\_walks to a NumPy array, np\_aw.
* Try to use [plt.plot()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html) on np\_aw. Also include [plt.show()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.show.html). Does it work out of the box?
* Transpose np\_aw by calling [np.transpose()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.transpose.html) on np\_aw. Call the result np\_aw\_t. Now every row in np\_all\_walks represents the position after 1 throw for the 10 random walks.
* Use [plt.plot()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html) to plot np\_aw\_t; also include a [plt.show()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.show.html). Does it look better this time?

**Daily XP1150**

**Exercise**

**Exercise**

**Implement clumsiness**

With this neatly written code of yours, changing the number of times the random walk should be simulated is super-easy. You simply update the [range()](https://docs.python.org/3/library/functions.html#func-range) function in the top-level for loop.

There's still something we forgot! You're a bit clumsy and you have a 0.1% chance of falling down. That calls for another random number generation. Basically, you can generate a random float between 0 and 1. If this value is less than or equal to 0.001, you should reset step to 0.

**Instructions**

**100 XP**

* Change the [range()](https://docs.python.org/3/library/functions.html#func-range) function so that the simulation is performed 250 times.
* Finish the if condition so that step is set to 0 if a random float is less or equal to 0.001. Use [np.random.rand()](https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html).

**Daily XP1250**

**Exercise**

**Exercise**

**Plot the distribution**

All these fancy visualizations have put us on a sidetrack. We still have to solve the million-dollar problem: *What are the odds that you'll reach 60 steps high on the Empire State Building?*

Basically, you want to know about the end points of all the random walks you've simulated. These end points have a certain distribution that you can visualize with a histogram.

Note that if your code is taking too long to run, you might be plotting a histogram of the wrong data!

**Instructions**

**100 XP**

* To make sure we've got enough simulations, go crazy. Simulate the random walk 500 times.
* From np\_aw\_t, select the last row. This contains the endpoint of all 500 random walks you've simulated. Store this NumPy array as ends.
* Use [plt.hist()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html) to build a histogram of ends. Don't forget [plt.show()](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.show.html) to display the plot.

**Daily XP1350**

**Exercise**

**Exercise**

**Calculate the odds**

The histogram of the previous exercise was created from a NumPy array ends, that contains 500 integers. Each integer represents the end point of a random walk. To calculate the chance that this end point is greater than or equal to 60, you can count the number of integers in ends that are greater than or equal to 60 and divide that number by 500, the total number of simulations.

Well then, what's the estimated chance that you'll reach at least 60 steps high if you play this Empire State Building game? The ends array is everything you need; it's available in your Python session so you can make calculations in the IPython Shell.

**Instructions**

**50 XP**

**Possible Answers**

* 

48.8%

* 

76.6%

* 

78.4%

* 

95.9%