Python for Data Science

Ritu Susan Sanjay

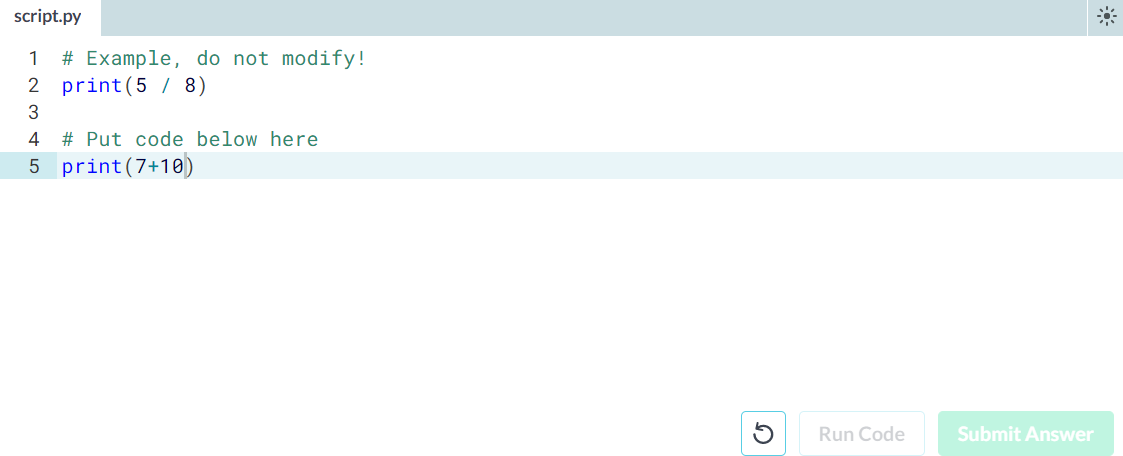
Course 1

Introduction to Python for Data Science

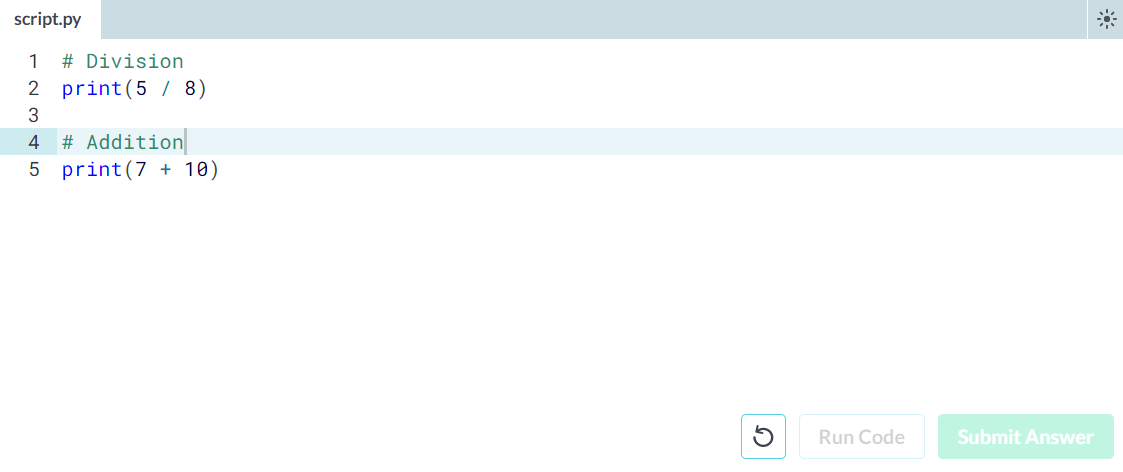
* The Python Interface
* In the Python script on the right, you can type Python code to solve the exercises. If you hit Run Code or Submit Answer, your python script (script.py) is executed and the output is shown in the IPython Shell. Submit Answer checks whether your submission is correct and gives you feedback.
* You can hit Run Code and Submit Answer as often as you want. If you're stuck, you can click Get Hint, and ultimately Get Solution.

You can also use the IPython Shell interactively by simply typing commands and hitting Enter. When you work in the shell directly, your code will not be checked for correctness so it is a great way to experiment

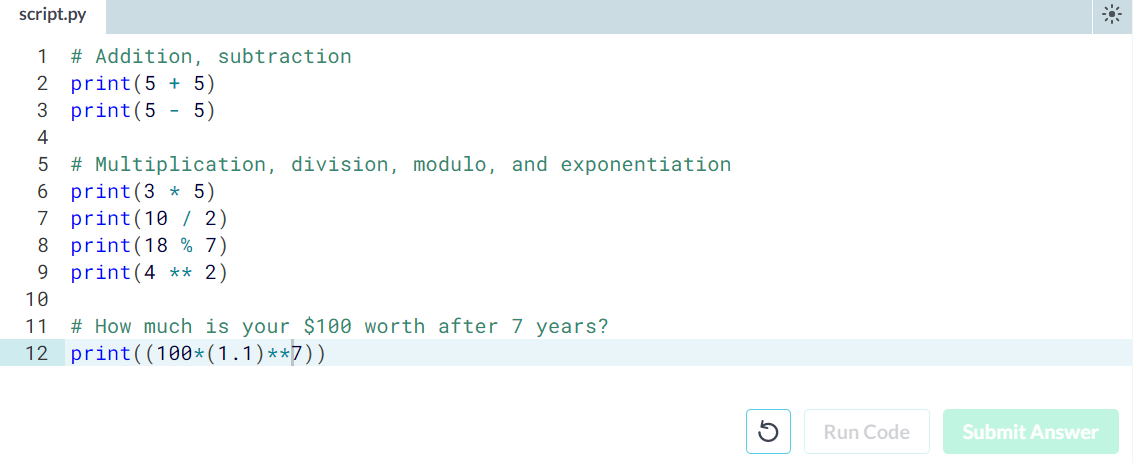
* Instructions
* 100 XP
* Experiment in the IPython Shell; type 5 / 8, for example.
* Add another line of code to the Python script on the top-right (not in the Shell): print(7 + 10).
* Hit Submit Answer to execute the Python script and receive feedback.



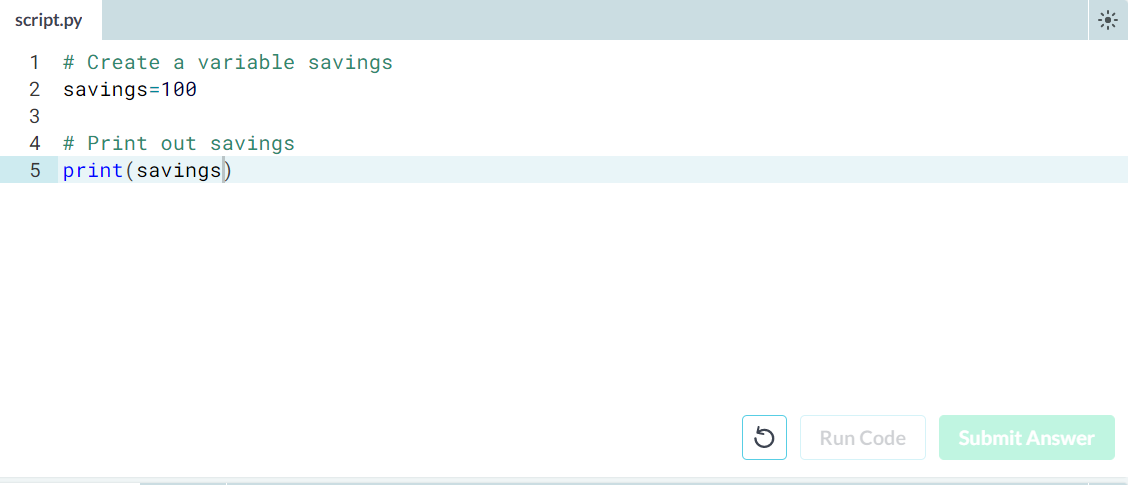
* Any comments?
* Something that Filip didn't mention in his videos is that you can add comments to your Python scripts. Comments are important to make sure that you and others can understand what your code is about.
* To add comments to your Python script, you can use the # tag. These comments are not run as Python code, so they will not influence your result. As an example, take the comment on the right, # Division; it is completely ignored during execution.
* Instructions
* 100 XP
* Above the print(7 + 10), add the comment # Addition.



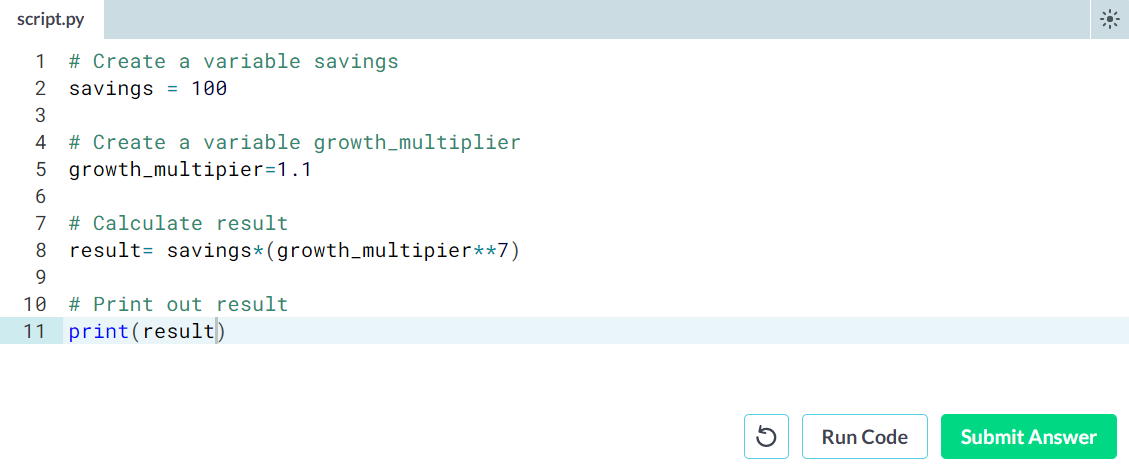
* Python as a calculator
* Python is perfectly suited to do basic calculations. Apart from addition, subtraction, multiplication and division, there is also support for more advanced operations such as:
* Exponentiation: \*\*. This operator raises the number to its left to the power of the number to its right. For example 4\*\*2 will give 16.
* Modulo: %. This operator returns the remainder of the division of the number to the left by the number on its right. For example 18 % 7 equals 4.
* The code in the script on the right gives some examples.
* Instructions
* 100 XP
* Suppose you have $100, which you can invest with a 10% return each year. After one year, it's 100×1.1=110 dollars, and after two years it's 100×1.1×1.1=121. Add code on the right to calculate how much money you end up with after 7 years.



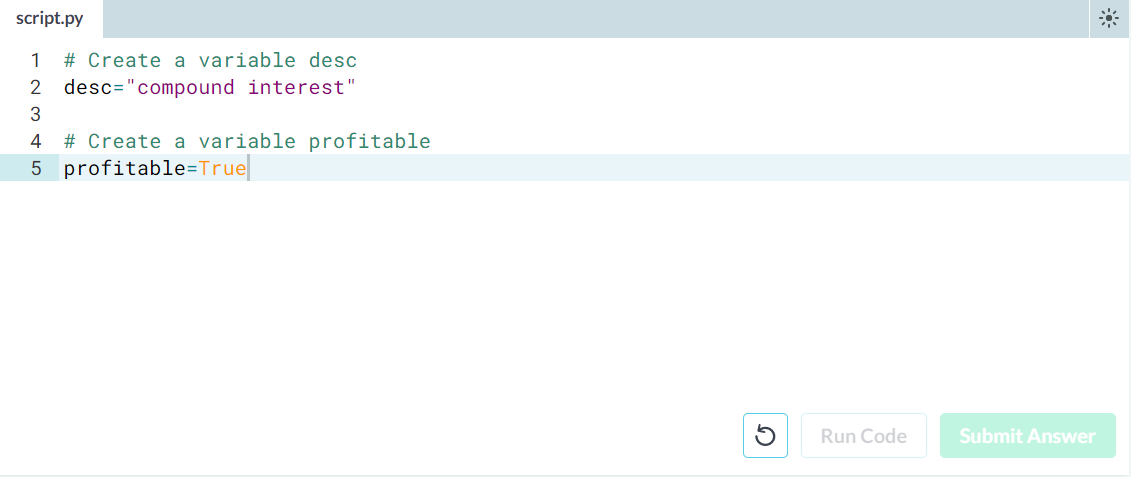
* Variable Assignment
* In Python, a variable allows you to refer to a value with a name. To create a variable use =, like this example:
* x = 5
* You can now use the name of this variable, x, instead of the actual value, 5.
* Remember, = in Python means assignment, it doesn't test equality!
* Instructions
* 100 XP
* Create a variable savings with the value 100.
* Check out this variable by typing print(savings) in the script.



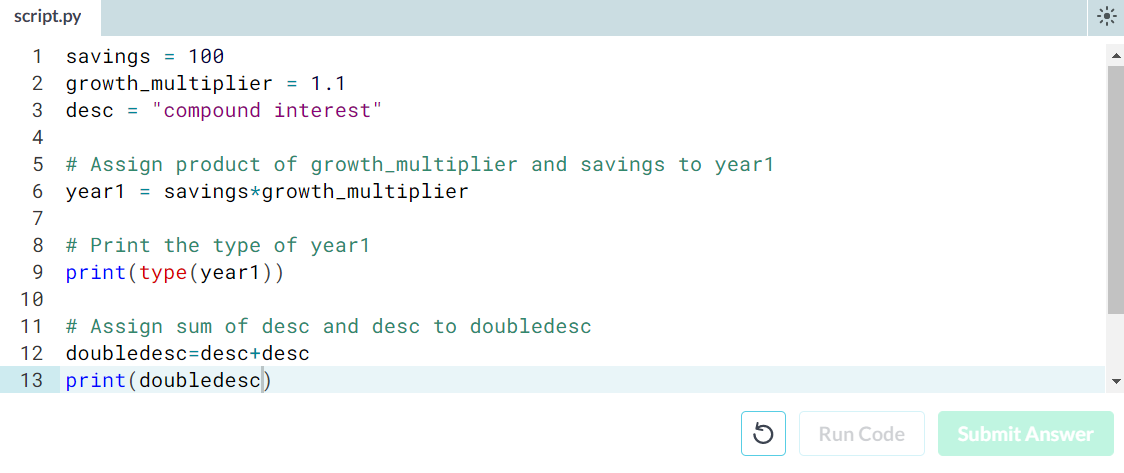
* Calculations with variables
* Remember how you calculated the money you ended up with after 7 years of investing $100? You did something like this:
* 100 \* 1.1 \*\* 7
* Instead of calculating with the actual values, you can use variables instead. The savings variable you've created in the previous exercise represents the $100 you started with. It's up to you to create a new variable to represent 1.1 and then redo the calculations!
* Instructions
* 100 XP
* Create a variable growth\_multiplier, equal to 1.1.
* Create a variable, result, equal to the amount of money you saved after 7 years.
* Print out the value of result.

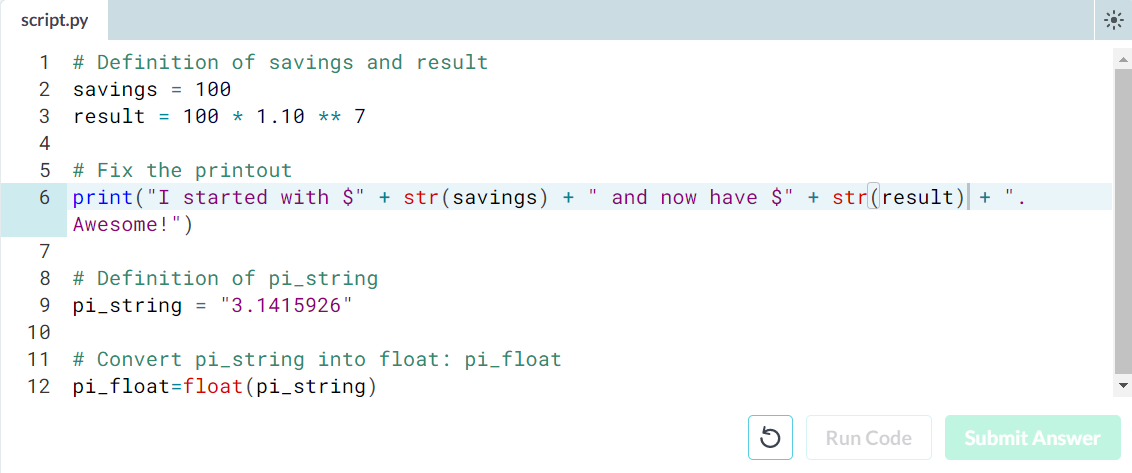


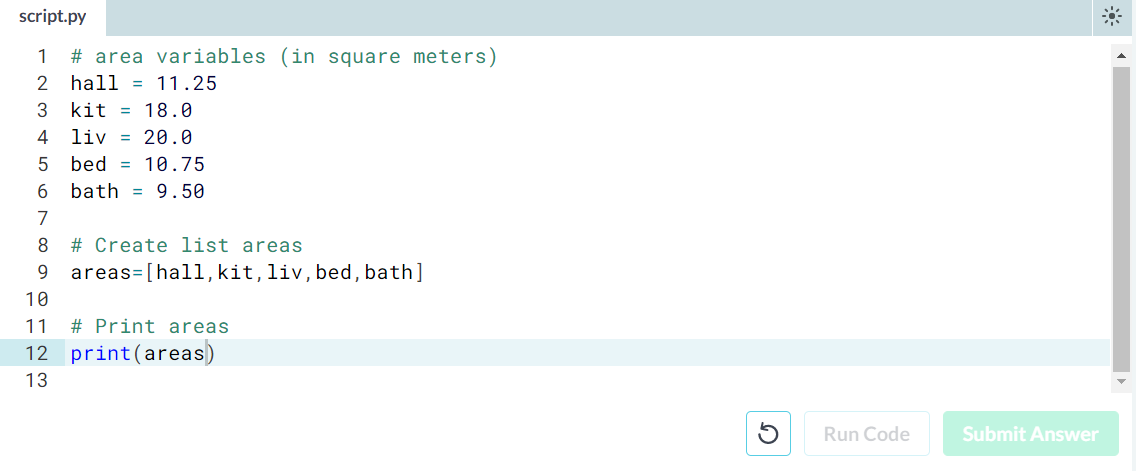
* Other variable types
* In the previous exercise, you worked with two Python data types:
* int, or integer: a number without a fractional part. savings, with the value 100, is an example of an integer.
* float, or floating point: a number that has both an integer and fractional part, separated by a point. growth\_multiplier, with the value 1.1, is an example of a float.
* Next to numerical data types, there are two other very common data types:
* str, or string: a type to represent text. You can use single or double quotes to build a string.
* bool, or boolean: a type to represent logical values. Can only be True or False (the capitalization is important!).
* Instructions
* 100 XP
* Create a new string, desc, with the value "compound interest".
* Create a new boolean, profitable, with the value True.



* Operations with other types
* Filip mentioned that different types behave differently in Python.
* When you sum two strings, for example, you'll get different behavior than when you sum two integers or two booleans.
* In the script some variables with different types have already been created. It's up to you to use them.
* Instructions
* 100 XP
* Calculate the product of savings and growth\_multiplier. Store the result in year1.
* What do you think the resulting type will be? Find out by printing out the type of year1.
* Calculate the sum of desc and desc and store the result in a new variable doubledesc.
* Print out doubledesc. Did you expect this?



* Type conversion
* Using the + operator to paste together two strings can be very useful in building custom messages.
* Suppose, for example, that you've calculated the return of your investment and want to summarize the results in a string. Assuming the floats savings and result are defined, you can try something like this:
* print("I started with $" + savings + " and now have $" + result + ". Awesome!")
* This will not work, though, as you cannot simply sum strings and floats.
* To fix the error, you'll need to explicitly convert the types of your variables. More specifically, you'll need str(), to convert a value into a string. str(savings), for example, will convert the float savings to a string.
* Similar functions such as int(), float() and bool() will help you convert Python values into any type.
* Instructions
* 100 XP
* Hit Run Code to run the code on the right. Try to understand the error message.
* Fix the code on the right such that the printout runs without errors; use the function str() to convert the variables to strings.
* Convert the variable pi\_string to a float and store this float as a new variable, pi\_float. 
* Create a list
* As opposed to int, bool etc., a list is a compound data type; you can group values together:
* a = "is"
* b = "nice"
* my\_list = ["my", "list", a, b]
* After measuring the height of your family, you decide to collect some information on the house you're living in. The areas of the different parts of your house are stored in separate variables for now, as shown in the script.
* Instructions
* 100 XP
* Create a list, areas, that contains the area of the hallway (hall), kitchen (kit), living room (liv), bedroom (bed) and bathroom (bath), in this order. Use the predefined variables.
* Print areas with the print() function.



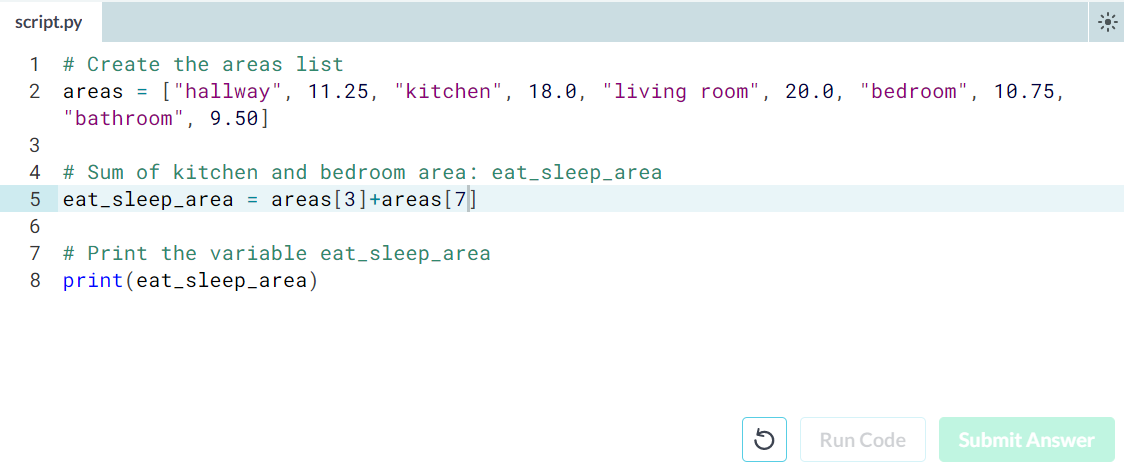
* Subset and conquer
* Subsetting Python lists is a piece of cake. Take the code sample below, which creates a list x and then selects "b" from it. Remember that this is the second element, so it has index 1. You can also use negative indexing.
* x = ["a", "b", "c", "d"]
* x[1]
* x[-3] # same result!

Remember the areas list from before, containing both strings and floats? Its definition is already in the script. Can you add the correct code to do some Python subsetting?

* Instructions
* 100 XP
* Print out the second element from the areas list (it has the value 11.25).
* Subset and print out the last element of areas, being 9.50. Using a negative index makes sense here!
* Select the number representing the area of the living room (20.0) and print it out.



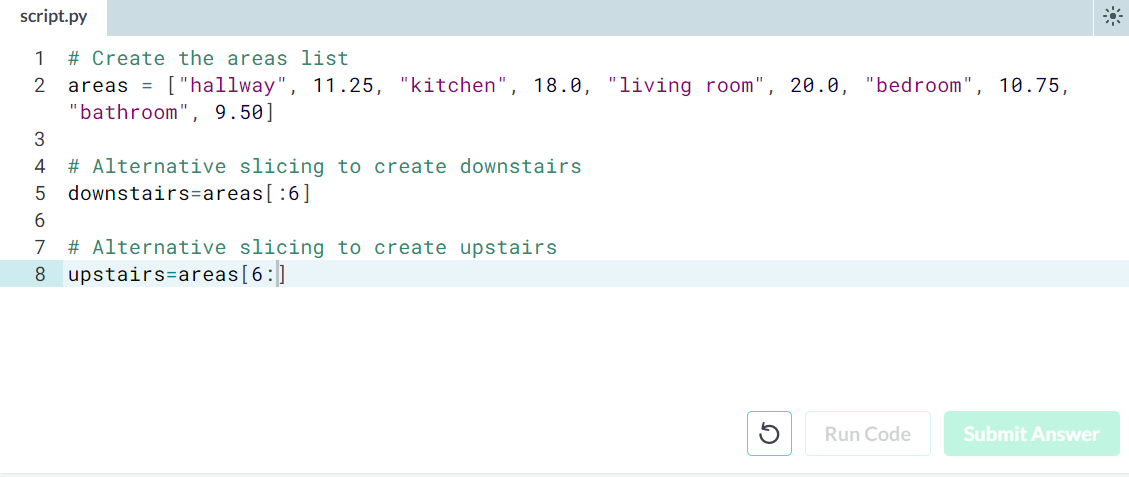
* Subset and calculate
* After you've extracted values from a list, you can use them to perform additional calculations. Take this example, where the second and fourth element of a list x are extracted. The strings that result are pasted together using the + operator:
* x = ["a", "b", "c", "d"]
* print(x[1] + x[3])
* Instructions
* 100 XP
* Using a combination of list subsetting and variable assignment, create a new variable, eat\_sleep\_area, that contains the sum of the area of the kitchen and the area of the bedroom.
* Print the new variable eat\_sleep\_area.



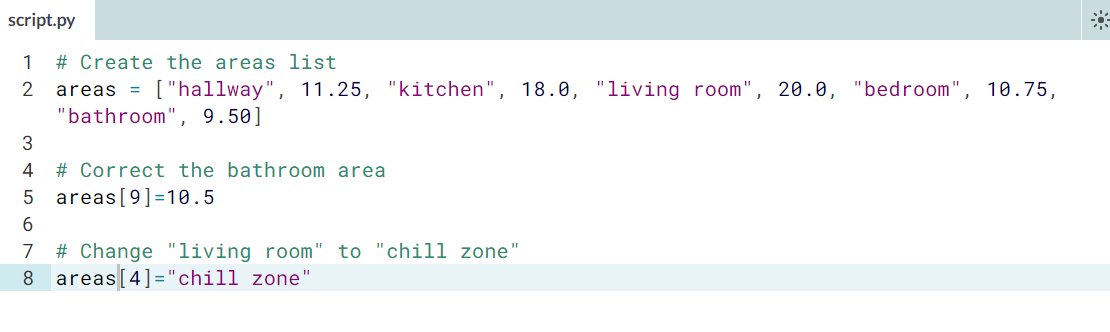
* Slicing and dicing
* Selecting single values from a list is just one part of the story. It's also possible to slice your list, which means selecting multiple elements from your list. Use the following syntax:
* my\_list[start:end]
* The start index will be included, while the end index is not.
* The code sample below shows an example. A list with "b" and "c", corresponding to indexes 1 and 2, are selected from a list x:
* x = ["a", "b", "c", "d"]
* x[1:3]
* The elements with index 1 and 2 are included, while the element with index 3 is not.
* Instructions
* 100 XP
* Use slicing to create a list, downstairs, that contains the first 6 elements of areas.
* Do a similar thing to create a new variable, upstairs, that contains the last 4 elements of areas.
* Print both downstairs and upstairs using print().



* Slicing and dicing (2)
* In the video, Filip first discussed the syntax where you specify both where to begin and end the slice of your list:
* my\_list[begin:end]
* However, it's also possible not to specify these indexes. If you don't specify the begin index, Python figures out that you want to start your slice at the beginning of your list. If you don't specify the end index, the slice will go all the way to the last element of your list. To experiment with this, try the following commands in the IPython Shell:
* x = ["a", "b", "c", "d"]
* x[:2]
* x[2:]
* x[:]
* Instructions
* 100 XP
* Create downstairs again, as the first 6 elements of areas. This time, simplify the slicing by omitting the begin index.
* Create upstairs again, as the last 4 elements of areas. This time, simplify the slicing by omitting the end index.



* Replace list elements
* Replacing list elements is pretty easy. Simply subset the list and assign new values to the subset. You can select single elements or you can change entire list slices at once.
* Use the IPython Shell to experiment with the commands below. Can you tell what's happening and why?
* x = ["a", "b", "c", "d"]
* x[1] = "r"
* x[2:] = ["s", "t"]
* For this and the following exercises, you'll continue working on the areas list that contains the names and areas of different rooms in a house.
* Instructions
* 100 XP
* Update the area of the bathroom area to be 10.50 square meters instead of 9.50.
* Make the areas list more trendy! Change "living room" to "chill zone".



**Exercise**

**Extend a list**

If you can change elements in a list, you sure want to be able to add elements to it, right? You can use the + operator:

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

y = x + ["e", "f"]

You just won the lottery, awesome! You decide to build a poolhouse and a garage. Can you add the information to the areas list?

**Instructions**

**100 XP**

* Use the + operator to paste the list ["poolhouse", 24.5] to the end of the areas list. Store the resulting list as areas\_1.
* Further extend areas\_1 by adding data on your garage. Add the string "garage" and float 15.45. Name the resulting list areas\_2.

# Create the areas list and make some changes

areas = ["hallway", 11.25, "kitchen", 18.0, "chill zone", 20.0,

"bedroom", 10.75, "bathroom", 10.50]

# Add poolhouse data to areas, new list is areas\_1

areas\_1 = areas+["poolhouse",24.5]

# Add garage data to areas\_1, new list is areas\_2

areas\_2 = areas\_1 +["garage",15.45]

**Exercise**

**Delete list elements**

Finally, you can also remove elements from your list. You can do this with the del statement:

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

del(x[1])

Pay attention here: as soon as you remove an element from a list, the indexes of the elements that come after the deleted element all change!

The updated and extended version of areas that you've built in the previous exercises is coded below. You can copy and paste this into the IPython Shell to play around with the result.

areas = ["hallway", 11.25, "kitchen", 18.0,

"chill zone", 20.0, "bedroom", 10.75,

"bathroom", 10.50, "poolhouse", 24.5,

"garage", 15.45]

There was a mistake! The amount you won with the lottery is not that big after all and it looks like the poolhouse isn't going to happen. You decide to remove the corresponding string and float from the areas list.

The ; sign is used to place commands on the same line. The following two code chunks are equivalent:

# Same line

command1; command2

# Separate lines

command1

command2

Which of the code chunks will do the job for us?

**Instructions**

**50 XP**

**Possible Answers**

* 

del(areas[10]); del(areas[11])

* 

del(areas[10:11])

* 

del(areas[-4:-2])

* 

del(areas[-3]); del(areas[-4])

##### Exercise

# Inner workings of lists

At the end of the video, Filip explained how Python lists work behind the scenes. In this exercise you'll get some hands-on experience with this.

The Python code in the script already creates a list with the name areas and a copy named areas\_copy. Next, the first element in the areas\_copy list is changed and the areas list is printed out. If you hit Run Code you'll see that, although you've changed areas\_copy, the change also takes effect in the areas list. That's because areas and areas\_copy point to the same list.

If you want to prevent changes in areas\_copy from also taking effect in areas, you'll have to do a more explicit copy of the areas list. You can do this with [**list()**](https://docs.python.org/3/library/functions.html#func-list) or by using [:].

##### Instructions

**100 XP**

Change the second command, that creates the variable areas\_copy, such that areas\_copy is an explicit copy of areas. After your edit, changes made to areas\_copy shouldn't affect areas. Hit Submit Answer to check this.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create list areas

areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Create areas\_copy

areas\_copy = list(areas)

# Change areas\_copy

areas\_copy[0] = 5.0

# Print areas

print(areas)

**Exercise**

**Familiar functions**

Out of the box, Python offers a bunch of built-in functions to make your life as a data scientist easier. You already know two such functions: [**print()**](https://docs.python.org/3/library/functions.html#print) and [**type()**](https://docs.python.org/3/library/functions.html#type). You've also used the functions **[str()](https://docs.python.org/3/library/functions.html" \l "func-str" \t "_blank)**, **[int()](https://docs.python.org/3/library/functions.html" \l "int" \t "_blank)**, [**bool()**](https://docs.python.org/3/library/functions.html#bool) and [**float()**](https://docs.python.org/3/library/functions.html#float) to switch between data types. These are built-in functions as well.

Calling a function is easy. To get the type of 3.0 and store the output as a new variable, result, you can use the following:

result = type(3.0)

The general recipe for calling functions and saving the result to a variable is thus:

output = function\_name(input)

**Instructions**

**100 XP**

* Use [**print()**](https://docs.python.org/3/library/functions.html#print) in combination with [**type()**](https://docs.python.org/3/library/functions.html#type) to print out the type of var1.
* Use **[len()](https://docs.python.org/3/library/functions.html" \l "len" \t "_blank)** to get the length of the list var1. Wrap it in a [**print()**](https://docs.python.org/3/library/functions.html#print) call to directly print it out.
* Use **[int()](https://docs.python.org/3/library/functions.html" \l "int" \t "_blank)** to convert var2 to an integer. Store the output as out2.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create variables var1 and var2

var1 = [1, 2, 3, 4]

var2 = True

# Print out type of var1

print(type(var1))

# Print out length of var1

print(len(var1))

# Convert var2 to an integer: out2

out2=int(var2)

**Exercise**

**Multiple arguments**

In the previous exercise, the square brackets around imag in the documentation showed us that the imag argument is optional. But Python also uses a different way to tell users about arguments being optional.

Have a look at the documentation of [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) by typing help(sorted)in the IPython Shell.

You'll see that [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) takes three arguments: iterable, key and reverse.

key=None means that if you don't specify the key argument, it will be None. reverse=False means that if you don't specify the reverseargument, it will be False.

In this exercise, you'll only have to specify iterable and reverse, not key. The first input you pass to [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) will be matched to the iterable argument, but what about the second input? To tell Python you want to specify reverse without changing anything about key, you can use =:

sorted(\_\_\_, reverse = \_\_\_)

Two lists have been created for you on the right. Can you paste them together and sort them in descending order?

Note: For now, we can understand an ***[iterable](https://docs.python.org/2/glossary.html" \l "term-iterable" \t "_blank)*** as being any collection of objects, e.g. a List.

**Instructions**

**100 XP**

* Use + to merge the contents of first and second into a new list: full.
* Call [**sorted()**](https://docs.python.org/3/library/functions.html#sorted) on full and specify the reverse argument to be True. Save the sorted list as full\_sorted.
* Finish off by printing out full\_sorted.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create lists first and second

first = [11.25, 18.0, 20.0]

second = [10.75, 9.50]

# Paste together first and second: full

full=first+second

# Sort full in descending order: full\_sorted

full\_sorted=sorted(full,reverse=True)

# Print out full\_sorted

print(full\_sorted)

**Exercise**

**String Methods**

Strings come with a bunch of methods. Follow the instructions closely to discover some of them. If you want to discover them in more detail, you can always type help(str) in the IPython Shell.

A string place has already been created for you to experiment with.

**Instructions**

**100 XP**

* Use the [**upper()**](https://docs.python.org/3/library/stdtypes.html#str.upper) method on place and store the result in place\_up. Use the syntax for calling methods that you learned in the previous video.
* Print out place and place\_up. Did both change?
* Print out the number of o's on the variable place by calling [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count) on place and passing the letter 'o' as an input to the method. We're talking about the variable place, not the word "place"!

[**Take Hint (-30 XP)**](javascript:void(0))

# string to experiment with: place

place = "poolhouse"

# Use upper() on place: place\_up

place\_up= place.upper()

# Print out place and place\_up

print(place)

print(place\_up)

# Print out the number of o's in place

print(place.count('o'))

**Exercise**

**List Methods**

Strings are not the only Python types that have methods associated with them. Lists, floats, integers and booleans are also types that come packaged with a bunch of useful methods. In this exercise, you'll be experimenting with:

* [**index()**](https://docs.python.org/3/library/stdtypes.html#str.index), to get the index of the first element of a list that matches its input and
* [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count), to get the number of times an element appears in a list.

You'll be working on the list with the area of different parts of a house: areas.

**Instructions**

**100 XP**

* Use the [**index()**](https://docs.python.org/3/library/stdtypes.html#str.index) method to get the index of the element in areas that is equal to 20.0. Print out this index.
* Call [**count()**](https://docs.python.org/3/library/stdtypes.html#str.count) on areas to find out how many times 9.50 appears in the list. Again, simply print out this number.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create list areas

areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0

print(areas.index(20.0))

# Print out how often 9.50 appears in areas

print(areas.count(9.50))

**Exercise**

**List Methods (2)**

Most list methods will change the list they're called on. Examples are:

* [**append()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that adds an element to the list it is called on,
* [**remove()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that removes the first element of a list that matches the input, and
* [**reverse()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable), that reverses the order of the elements in the list it is called on.

You'll be working on the list with the area of different parts of the house: areas.

**Instructions**

**100 XP**

* Use [**append()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable) twice to add the size of the poolhouse and the garage again: 24.5 and 15.45, respectively. Make sure to add them in this order.
* Print out areas
* Use the [**reverse()**](https://docs.python.org/3/library/stdtypes.html#typesseq-mutable) method to reverse the order of the elements in areas.
* Print out areas once more.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create list areas

areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Use append twice to add poolhouse and garage size

areas.append(24.5)

areas.append(15.45)

# Print out areas

print(areas)

# Reverse the orders of the elements in areas

areas.reverse()

print(areas)

**Exercise**

**Import package**

As a data scientist, some notions of geometry never hurt. Let's refresh some of the basics.

For a fancy clustering algorithm, you want to find the circumference, CC, and area, AA, of a circle. When the radius of the circle is r, you can calculate CCand AA as:

C=2πrC=2πr

A=πr2A=πr2

To use the constant pi, you'll need the math package. A variable r is already coded in the script. Fill in the code to calculate C and A and see how the [**print()**](https://docs.python.org/3/library/functions.html#print) functions create some nice printouts.

**Instructions**

**100 XP**

* Import the math package. Now you can access the constant pi with math.pi.
* Calculate the circumference of the circle and store it in C.
* Calculate the area of the circle and store it in A.

[**Take Hint (-30 XP)**](javascript:void(0))

# Definition of radius

r = 0.43

# Import the math package

import math

pi=math.pi

# Calculate C

C = 2\*pi\*r

# Calculate A

A = pi\*r\*r

# Build printout

print("Circumference: " + str(C))

print("Area: " + str(A))

**Exercise**

**Selective import**

General imports, like import math, make **all** functionality from the mathpackage available to you. However, if you decide to only use a specific part of a package, you can always make your import more selective:

from math import pi

Let's say the Moon's orbit around planet Earth is a perfect circle, with a radius r (in km) that is defined in the script.

**Instructions**

**100 XP**

* Perform a selective import from the math package where you only import the radians function.
* Calculate the distance travelled by the Moon over 12 degrees of its orbit. Assign the result to dist. You can calculate this as r \* phi, where ris the radius and phi is the angle in radians. To convert an angle in degrees to an angle in radians, use the [**radians()**](https://docs.python.org/3/library/math.html#math.radians) function, which you just imported.
* Print out dist.

[**Take Hint (-30 XP)**](javascript:void(0))

# Definition of radius

r = 192500

# Import radians function of math package

from math import radians

# Travel distance of Moon over 12 degrees. Store in dist.

phi=radians(12)

dist = r\*phi

# Print out dist

print(dist)

**Exercise**

**Your First NumPy Array**

In this chapter, we're going to dive into the world of baseball. Along the way, you'll get comfortable with the basics of numpy, a powerful package to do data science.

A list baseball has already been defined in the Python script, representing the height of some baseball players in centimeters. Can you add some code here and there to create a numpy array from it?

**Instructions**

**100 XP**

* Import the numpy package as np, so that you can refer to numpy with np.
* Use **[np.array()](http://docs.scipy.org/doc/numpy-1.10.0/glossary.html" \l "term-array" \t "_blank)** to create a numpy array from baseball. Name this array np\_baseball.
* Print out the type of np\_baseball to check that you got it right.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create list baseball

baseball = [180, 215, 210, 210, 188, 176, 209, 200]

# Import the numpy package as np

import numpy as np

# Create a numpy array from baseball: np\_baseball

np\_baseball=np.array(baseball)

# Print out type of np\_baseball

print(type(np\_baseball))

**Exercise**

**Baseball players' height**

You are a huge baseball fan. You decide to call the MLB (Major League Baseball) and ask around for some more statistics on the height of the main players. They pass along data on more than a thousand players, which is stored as a regular Python list: height\_in. The height is expressed in inches. Can you make a numpy array out of it and convert the units to meters?

height\_in is already available and the numpy package is loaded, so you can start straight away (Source: [**stat.ucla.edu**](http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_MLB_HeightsWeights)).

**Instructions**

**100 XP**

* Create a numpy array from height\_in. Name this new array np\_height\_in.
* Print np\_height\_in.
* Multiply np\_height\_in with 0.0254 to convert all height measurements from inches to meters. Store the new values in a new array, np\_height\_m.
* Print out np\_height\_m and check if the output makes sense.

[**Take Hint (-30 XP)**](javascript:void(0))

# height is available as a regular list

# Import numpy

import numpy as np

# Create a numpy array from height\_in: np\_height\_in

np\_height\_in = np.array(height\_in)

# Print out np\_height\_in

print(np\_height\_in)

# Convert np\_height\_in to m: np\_height\_m

np\_height\_m = 0.0254\*np\_height\_in

# Print np\_height\_m

print(np\_height\_m)

**Exercise**

**Baseball player's BMI**

The MLB also offers to let you analyze their weight data. Again, both are available as regular Python lists: height\_in and weight. height\_in is in inches and weight\_lb is in pounds.

It's now possible to calculate the BMI of each baseball player. Python code to convert height\_in to a numpy array with the correct units is already available in the workspace. Follow the instructions step by step and finish the game!

**Instructions**

**100 XP**

* Create a numpy array from the weight\_lb list with the correct units. Multiply by 0.453592 to go from pounds to kilograms. Store the resulting numpy array as np\_weight\_kg.
* Use np\_height\_m and np\_weight\_kg to calculate the BMI of each player. Use the following equation:

BMI=weight(kg)height(m)2BMI=weight(kg)height(m)2

Save the resulting numpy array as bmi.

* Print out bmi.

[**Take Hint (-30 XP)**](javascript:void(0))

# height and weight are available as regular lists

# Import numpy

import numpy as np

# Create array from height\_in with metric units: np\_height\_m

np\_height\_m = np.array(height\_in) \* 0.0254

# Create array from weight\_lb with metric units: np\_weight\_kg

np\_weight\_kg=np.array(weight\_lb) \* 0.453592

# Calculate the BMI: bmi

bmi=np\_weight\_kg/(np\_height\_m\*\*2)

# Print out bmi

print(bmi)

##### Exercise

# Lightweight baseball players

To subset both regular Python lists and numpy arrays, you can use square brackets:

x = [4 , 9 , 6, 3, 1]

x[1]

import numpy as np

y = np.array(x)

y[1]

For numpy specifically, you can also use boolean numpy arrays:

high = y > 5

y[high]

The code that calculates the BMI of all baseball players is already included. Follow the instructions and reveal interesting things from the data!

##### Instructions

**100 XP**

* Create a boolean numpy array: the element of the array should be True if the corresponding baseball player's BMI is below 21. You can use the <operator for this. Name the array light.
* Print the array light.
* Print out a numpy array with the BMIs of all baseball players whose BMI is below 21. Use light inside square brackets to do a selection on the bmiarray.

[**Take Hint (-30 XP)**](javascript:void(0))

# height and weight are available as a regular lists

# Import numpy

import numpy as np

# Calculate the BMI: bmi

np\_height\_m = np.array(height\_in) \* 0.0254

np\_weight\_kg = np.array(weight\_lb) \* 0.453592

bmi = np\_weight\_kg / np\_height\_m \*\* 2

# Create the light array

light=bmi<21

# Print out light

print(light)

# Print out BMIs of all baseball players whose BMI is below 21

print(bmi[light])

##### Exercise

# Subsetting NumPy Arrays

You've seen it with your own eyes: Python lists and numpy arrays sometimes behave differently. Luckily, there are still certainties in this world. For example, subsetting (using the square bracket notation on lists or arrays) works exactly the same. To see this for yourself, try the following lines of code in the IPython Shell:

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

x[1]

np\_x = np.array(x)

np\_x[1]

The script on the right already contains code that imports numpy as np, and stores both the height and weight of the MLB players as numpy arrays.

##### Instructions

**100 XP**

* Subset np\_weight\_lb by printing out the element at index 50.
* Print out a sub-array of np\_height\_in that contains the elements at index 100 up to **and including** index 110.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import numpy

import numpy as np

# Store weight and height lists as numpy arrays

np\_weight\_lb = np.array(weight\_lb)

np\_height\_in = np.array(height\_in)

# Print out the weight at index 50

print(np\_weight\_lb[50])

# Print out sub-array of np\_height\_in: index 100 up to and including index 110

print(np\_height\_in[100:111])

##### Exercise

# Your First 2D NumPy Array

Before working on the actual MLB data, let's try to create a 2D numpy array from a small list of lists.

In this exercise, baseball is a list of lists. The main list contains 4 elements. Each of these elements is a list containing the height and the weight of 4 baseball players, in this order. baseball is already coded for you in the script.

##### Instructions

**100 XP**

* Use **[np.array()](http://docs.scipy.org/doc/numpy-1.10.0/glossary.html" \l "term-array" \t "_blank)** to create a 2D numpy array from baseball. Name it np\_baseball.
* Print out the type of np\_baseball.
* Print out the shape attribute of np\_baseball. Use np\_baseball.shape.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create baseball, a list of lists

baseball = [[180, 78.4],

[215, 102.7],

[210, 98.5],

[188, 75.2]]

# Import numpy

import numpy as np

# Create a 2D numpy array from baseball: np\_baseball

np\_baseball=np.array(baseball)

# Print out the type of np\_baseball

print(type(np\_baseball))

# Print out the shape of np\_baseball

print(np.shape(np\_baseball))

##### Exercise

# Baseball data in 2D form

You 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).

The MLB was, again, very helpful and passed you the data in a different structure, a Python list of lists. In this list of lists, each sublist represents the height and weight of a single baseball player. The name of this embedded list is baseball.

Can you store the data as a 2D array to unlock numpy's extra functionality?

##### Instructions

**100 XP**

* Use **[np.array()](http://docs.scipy.org/doc/numpy-1.10.0/glossary.html" \l "term-array" \t "_blank)** to create a 2D numpy array from baseball. Name it np\_baseball.
* Print out the shape attribute of np\_baseball.

[**Take Hint (-30 XP)**](javascript:void(0))

# baseball is available as a regular list of lists

# Import numpy package

import numpy as np

# Create a 2D numpy array from baseball: np\_baseball

np\_baseball=np.array(baseball)

# Print out the shape of np\_baseball

print(np.shape(np\_baseball))

##### Exercise

# Subsetting 2D NumPy Arrays

If your 2D numpy array has a regular structure, i.e. each row and column has a fixed number of values, complicated ways of subsetting become very easy. 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.

The code that converts the pre-loaded baseball list to a 2D numpy array is already in the script. The first column contains the players' height in inches and the second column holds player weight, in pounds. Add some lines to make the correct selections. Remember that in Python, the first element is at index 0!

##### Instructions

**100 XP**

* Print out the 50th row of np\_baseball.
* Make a new variable, np\_weight\_lb, containing the entire second column of np\_baseball.
* Select the height (first column) of the 124th baseball player in np\_baseball and print it out.

[**Take Hint (-30 XP)**](javascript:void(0))

# baseball is available as a regular list of lists

# Import numpy package

import numpy as np

# Create np\_baseball (2 cols)

np\_baseball = np.array(baseball)

# Print out the 50th row of np\_baseball

print(np\_baseball[49,:])

# Select the entire second column of np\_baseball: np\_weight\_lb

np\_weight\_lb=np\_baseball[:,1]

# Print out height of 124th player

print(np\_baseball[123][0])

##### Exercise

# 2D Arithmetic

Remember how you calculated the Body Mass Index for all baseball players? numpy was able to perform all calculations element-wise (i.e. element by element). For 2D numpy arrays this isn't any different! You can combine matrices with single numbers, with vectors, and with other matrices.

Execute the code below in the IPython shell and see if you understand:

import numpy as np

np\_mat = np.array([[1, 2],

[3, 4],

[5, 6]])

np\_mat \* 2

np\_mat + np.array([10, 10])

np\_mat + np\_mat

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).

##### Instructions

**100 XP**

* 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.

[**Take Hint (-30 XP)**](javascript:void(0))

# baseball is available as a regular list of lists

# updated is available as 2D numpy array

# Import numpy package

import numpy as np

# Create np\_baseball (3 cols)

np\_baseball = np.array(baseball)

# Print out addition of np\_baseball and updated

print(np\_baseball+updated)

# Create numpy array: conversion

conversion = np.array([0.0254,0.453592,1])

# Print out product of np\_baseball and conversion

print(np\_baseball\*conversion)

##### Exercise

# Average versus median

You now know how to use numpy functions to get a better feeling for your data. It basically comes down to importing numpy and then calling several simple functions on the numpy arrays:

import numpy as np

x = [1, 4, 8, 10, 12]

np.mean(x)

np.median(x)

The baseball data is available as a 2D numpy array with 3 columns (height, weight, age) and 1015 rows. The name of this numpy array is np\_baseball. After restructuring the data, however, you notice that some height values are abnormally high. Follow the instructions and discover which summary statistic is best suited if you're dealing with so-called outliers.

##### Instructions

**100 XP**

* Create numpy array np\_height\_in that is equal to first column of np\_baseball.
* Print out the mean of np\_height\_in.
* Print out the median of np\_height\_in.

[**Take Hint (-30 XP)**](javascript:void(0))

# np\_baseball is available

# Import numpy

import numpy as np

# Create np\_height\_in from np\_baseball

np\_height\_in=np\_baseball[:,0]

# Print out the mean of np\_height\_in

print(np.mean(np\_height\_in))

# Print out the median of np\_height\_in

print(np.median(np\_height\_in))

##### Exercise

# Explore the baseball data

Because the mean and median are so far apart, you decide to complain to the MLB. They find the error and send the corrected data over to you. It's again available as a 2D Numpy array np\_baseball, with three columns.

The Python script on the right already includes code to print out informative messages with the different summary statistics. Can you finish the job?

##### Instructions

**100 XP**

* The code to print out the mean height is already included. Complete the code for the median height. Replace None with the correct code.
* Use **[np.std()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.std.html" \t "_blank)** on the first column of np\_baseball to calculate stddev. Replace None with the correct code.
* Do big players tend to be heavier? Use **[np.corrcoef()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.corrcoef.html" \t "_blank)** to store the correlation between the first and second column of np\_baseball in corr. Replace None with the correct code.

[**Take Hint (-30 XP)**](javascript:void(0))

# np\_baseball is available

# Import numpy

import numpy as np

# Print mean height (first column)

avg = np.mean(np\_baseball[:,0])

print("Average: " + str(avg))

# Print median height. Replace 'None'

med = np.median(np\_baseball[:,0])

print("Median: " + str(med))

# Print out the standard deviation on height. Replace 'None'

stddev = np.std(np\_baseball[:,0])

print("Standard Deviation: " + str(stddev))

# Print out correlation between first and second column. Replace 'None'

corr = np.corrcoef(np\_baseball[:,0], np\_baseball[:,1])

print("Correlation: " + str(corr))

##### Exercise

# Blend it all together

In the last few exercises you've learned everything there is to know about heights and weights of baseball players. Now it's time to dive into another sport: soccer.

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.

##### Instructions

**100 XP**

* Convert heights and positions, which are regular lists, to numpy arrays. Call them np\_heights and np\_positions.
* Extract all the heights of the goalkeepers. You can use a little trick here: use np\_positions == 'GK' as an index for np\_heights. Assign the result to gk\_heights.
* Extract all the heights of all the other players. This time use np\_positions != 'GK' as an index for np\_heights. Assign the result to other\_heights.
* Print out the median height of the goalkeepers using **[np.median()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.median.html" \t "_blank)**. Replace None with the correct code.
* Do the same for the other players. Print out their median height. Replace None with the correct code.

[**Take Hint (-30 XP)**](javascript:void(0))

# heights and positions are available as lists

# Import numpy

import numpy as np

# Convert positions and heights to numpy arrays: np\_positions, np\_heights

np\_positions=np.array(positions)

np\_heights=np.array(heights)

# Heights of the goalkeepers: gk\_heights

gk\_heights=np\_heights[np\_positions=='GK']

# Heights of the other players: other\_heights

other\_heights=np\_heights[np\_positions!='GK']

# Print out the median height of goalkeepers. Replace 'None'

print("Median height of goalkeepers: " + str(np.median(gk\_heights)))

# Print out the median height of other players. Replace 'None'

print("Median height of other players: " + str(np.median(other\_heights)))

Course 2

Intermediate Python for Data Science

##### Exercise

# Line plot (1)

With matplotlib, you can create a bunch of different plots in Python. The most basic plot is the line plot. A general recipe is given here.

import matplotlib.pyplot as plt

plt.plot(x,y)

plt.show()

In the video, you already saw how much the world population has grown over the past years. Will it continue to do so? The world bank has estimates of the world population for the years 1950 up to 2100. The years are loaded in your workspace as a list called year, and the corresponding populations as a list called pop.

This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the [***Python for data science Cheat Sheet***](https://datacamp-community-prod.s3.amazonaws.com/e30fbcd9-f595-4a9f-803d-05ca5bf84612) and keep it handy!

##### Instructions

**100 XP**

* [**print()**](https://docs.python.org/3/library/functions.html#print) the last item from both the year and the pop list to see what the predicted population for the year 2100 is. Use two print() functions.
* Before you can start, you should import matplotlib.pyplot as plt. pyplot is a sub-package of matplotlib, hence the dot.
* Use **[plt.plot()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.plot" \t "_blank)** to build a line plot. year should be mapped on the horizontal axis, pop on the vertical axis. Don't forget to finish off with the [**show()**](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.show) function to actually display the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Print the last item from year and pop

print(year[-1])

print(pop[-1])

# Import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

# Make a line plot: year on the x-axis, pop on the y-axis

plt.plot(year,pop)

# Display the plot with plt.show()

plt.show()

##### Exercise

# Line plot (3)

Now that you've built your first line plot, let's start working on the data that professor Hans Rosling used to build his beautiful bubble chart. It was collected in 2007. Two lists are available for you:

* life\_exp which contains the life expectancy for each country and
* gdp\_cap, which contains the GDP per capita (i.e. per person) for each country expressed in US Dollars.

GDP stands for Gross Domestic Product. It basically represents the size of the economy of a country. Divide this by the population and you get the GDP per capita.

matplotlib.pyplot is already imported as plt, so you can get started straight away.

##### Instructions

**100 XP**

* Print the last item from both the list gdp\_cap, and the list life\_exp; it is information about Zimbabwe.
* Build a line chart, with gdp\_cap on the x-axis, and life\_exp on the y-axis. Does it make sense to plot this data on a line plot?
* Don't forget to finish off with a plt.show() command, to actually display the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Print the last item of gdp\_cap and life\_exp

print(gdp\_cap[-1])

print(life\_exp[-1])

# Make a line plot, gdp\_cap on the x-axis, life\_exp on the y-axis

plt.plot(gdp\_cap,life\_exp)

# Display the plot

plt.show()

##### Exercise

# Scatter Plot (1)

When you have a time scale along the horizontal axis, the line plot is your friend. But in many other cases, when you're trying to assess if there's a correlation between two variables, for example, the scatter plot is the better choice. Below is an example of how to build a scatter plot.

import matplotlib.pyplot as plt

plt.scatter(x,y)

plt.show()

Let's continue with the gdp\_cap versus life\_exp plot, the GDP and life expectancy data for different countries in 2007. Maybe a scatter plot will be a better alternative?

Again, the matplotlib.pyplot package is available as plt.

##### Instructions

**100 XP**

* Change the line plot that's coded in the script to a scatter plot.
* A correlation will become clear when you display the GDP per capita on a logarithmic scale. Add the line plt.xscale('log').
* Finish off your script with plt.show() to display the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Change the line plot below to a scatter plot

plt.scatter(gdp\_cap, life\_exp)

# Put the x-axis on a logarithmic scale

plt.xscale('log')

# Show plot

plt.show()

##### Exercise

# Scatter plot (2)

In the previous exercise, you saw that that the higher GDP usually corresponds to a higher life expectancy. In other words, there is a positive correlation.

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.

##### Instructions

**100 XP**

* Start from scratch: import matplotlib.pyplot as plt.
* Build a scatter plot, where pop is mapped on the horizontal axis, and life\_exp is mapped on the vertical axis.
* Finish the script with plt.show() to actually display the plot. Do you see a correlation?

[**Take Hint (-30 XP)**](javascript:void(0))

# Import package

import matplotlib.pyplot as plt

# Build Scatter plot

plt.scatter(pop,life\_exp)

# Show plot

plt.show()

##### Exercise

# Build a histogram (1)

life\_exp, the list containing data on the life expectancy for different countries in 2007, is available in your Python shell.

To see how life expectancy in different countries is distributed, let's create a histogram of life\_exp.

matplotlib.pyplot is already available as plt.

##### Instructions

**100 XP**

* Use **[plt.hist()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.hist" \t "_blank)** to create a histogram of the values in life\_exp. Do not specify the number of bins; Python will set the number of bins to 10 by default for you.
* Add plt.show() to actually display the histogram. Can you tell which bin contains the most observations?

[**Take Hint (-30 XP)**](javascript:void(0))

# Create histogram of life\_exp data

plt.hist(life\_exp)

# Display histogram

plt.show()

##### Exercise

# Build a histogram (2): bins

In the previous exercise, you didn't specify the number of bins. By default, Python sets the number of bins to 10 in that case. The number of bins is pretty important. Too few bins will oversimplify reality and won't show you the details. Too many bins will overcomplicate reality and won't show the bigger picture.

To control the number of bins to divide your data in, you can set the binsargument.

That's exactly what you'll do in this exercise. You'll be making two plots here. The code in the script already includes plt.show() and plt.clf() calls; plt.show() displays a plot; plt.clf() cleans it up again so you can start afresh.

As before, life\_exp is available and matplotlib.pyplot is imported as plt.

##### Instructions

**100 XP**

* Build a histogram of life\_exp, with 5 bins. Can you tell which bin contains the most observations?
* Build another histogram of life\_exp, this time with 20 bins. Is this better?

[**Take Hint (-30 XP)**](javascript:void(0))

# Build histogram with 5 bins

plt.hist(life\_exp,bins=5)

# Show and clean up plot

plt.show()

plt.clf()

# Build histogram with 20 bins

plt.hist(life\_exp,bins=20)

# Show and clean up again

plt.show()

plt.clf()

##### Exercise

# Build a histogram (3): compare

In the video, you saw population pyramids for the present day and for the future. Because we were using a histogram, it was very easy to make a comparison.

Let's do a similar comparison. life\_exp contains life expectancy data for different countries in 2007. You also have access to a second list now, life\_exp1950, containing similar data for 1950. Can you make a histogram for both datasets?

You'll again be making two plots. The plt.show() and plt.clf()commands to render everything nicely are already included. Also matplotlib.pyplot is imported for you, as plt.

##### Instructions

**100 XP**

* Build a histogram of life\_exp with 15 bins.
* Build a histogram of life\_exp1950, also with 15 bins. Is there a big difference with the histogram for the 2007 data?

[**Take Hint (-30 XP)**](javascript:void(0))

# Histogram of life\_exp, 15 bins

plt.hist(life\_exp,bins=15)

# Show and clear plot

plt.show()

plt.clf()

# Histogram of life\_exp1950, 15 bins

plt.hist(life\_exp1950,bins=15)

# Show and clear plot again

plt.show()

plt.clf()

##### Exercise

# Labels

It's time to customize your own plot. This is the fun part, you will see your plot come to life!

You're going to work on the scatter plot with world development data: GDP per capita on the x-axis (logarithmic scale), life expectancy on the y-axis. The code for this plot is available in the script.

As a first step, let's add axis labels and a title to the plot. You can do this with the **[xlabel()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.xlabel" \t "_blank)**, **[ylabel()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.ylabel" \t "_blank)** and [**title()**](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.title) functions, available in matplotlib.pyplot. This sub-package is already imported as plt.

##### Instructions

**100 XP**

* The strings xlab and ylab are already set for you. Use these variables to set the label of the x- and y-axis.
* The string title is also coded for you. Use it to add a title to the plot.
* After these customizations, finish the script with plt.show() to actually display the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Basic scatter plot, log scale

plt.scatter(gdp\_cap, life\_exp)

plt.xscale('log')

# Strings

xlab = 'GDP per Capita [in USD]'

ylab = 'Life Expectancy [in years]'

title = 'World Development in 2007'

# Add axis labels

plt.xlabel(xlab)

plt.ylabel(ylab)

# Add title

plt.title(title)

# After customizing, display the plot

plt.show()

##### Exercise

# Ticks

The customizations you've coded up to now are available in the script, in a more concise form.

In the video, Filip 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.

Let's do a similar thing for the x-axis of your world development chart, with the **[xticks()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.xticks" \t "_blank)** function. The tick values 1000, 10000 and 100000 should be replaced by 1k, 10k and 100k. To this end, two lists have already been created for you: tick\_val and tick\_lab.

##### Instructions

**100 XP**

* Use tick\_val and tick\_lab as inputs to the **[xticks()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.xticks" \t "_blank)** function to make the the plot more readable.
* As usual, display the plot with plt.show() after you've added the customizations.

[**Take Hint (-30 XP)**](javascript:void(0))

# Scatter plot

plt.scatter(gdp\_cap, life\_exp)

# Previous customizations

plt.xscale('log')

plt.xlabel('GDP per Capita [in USD]')

plt.ylabel('Life Expectancy [in years]')

plt.title('World Development in 2007')

# Definition of tick\_val and tick\_lab

tick\_val = [1000, 10000, 100000]

tick\_lab = ['1k', '10k', '100k']

# Adapt the ticks on the x-axis

plt.xticks([1000, 10000, 100000],['1k', '10k', '100k'])

# After customizing, display the plot

plt.show()

##### Exercise

# Sizes

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.

##### Instructions

**100 XP**

* Run the script to see how the plot changes.
* 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 by assigning np\_pop \* 2 to np\_popagain. Because np\_pop is a Numpy array, each array element will be doubled.
  + Change the s argument inside **[plt.scatter()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.scatter" \t "_blank)** to be np\_popinstead of pop.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import numpy as np

import numpy as np

# Store pop as a numpy array: np\_pop

np\_pop=np.array(pop)

# Double np\_pop

np\_pop=np\_pop\*2

# Update: set s argument to np\_pop

plt.scatter(gdp\_cap, life\_exp, s = np\_pop)

# Previous customizations

plt.xscale('log')

plt.xlabel('GDP per Capita [in USD]')

plt.ylabel('Life Expectancy [in years]')

plt.title('World Development in 2007')

plt.xticks([1000, 10000, 100000],['1k', '10k', '100k'])

# Display the plot

plt.show()

##### Exercise

# Colors

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

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'

}

Nothing to worry about now; you will learn about dictionaries in the next chapter.

##### Instructions

**100 XP**

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

[**Take Hint (-30 XP)**](javascript:void(0))

# Specify c and alpha inside plt.scatter()

plt.scatter(x = gdp\_cap, y = life\_exp, s = np.array(pop) \* 2,c=col,alpha=0.8)

# Previous customizations

plt.xscale('log')

plt.xlabel('GDP per Capita [in USD]')

plt.ylabel('Life Expectancy [in years]')

plt.title('World Development in 2007')

plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Show the plot

plt.show()

##### Exercise

# Additional Customizations

If you have another look at the script, under # Additional Customizations, you'll see that there are two **[plt.text()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.text" \t "_blank)** functions now. They add the words "India" and "China"in the plot.

##### Instructions

**100 XP**

* Add **[plt.grid(True)](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.grid" \t "_blank)** after the **[plt.text()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.text" \t "_blank)** calls so that gridlines are drawn on the plot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Scatter plot

plt.scatter(x = gdp\_cap, y = life\_exp, s = np.array(pop) \* 2, c = col, alpha = 0.8)

# Previous customizations

plt.xscale('log')

plt.xlabel('GDP per Capita [in USD]')

plt.ylabel('Life Expectancy [in years]')

plt.title('World Development in 2007')

plt.xticks([1000,10000,100000], ['1k','10k','100k'])

# Additional customizations

plt.text(1550, 71, 'India')

plt.text(5700, 80, 'China')

# Add grid() call

plt.grid(True)

# Show the plot

plt.show()