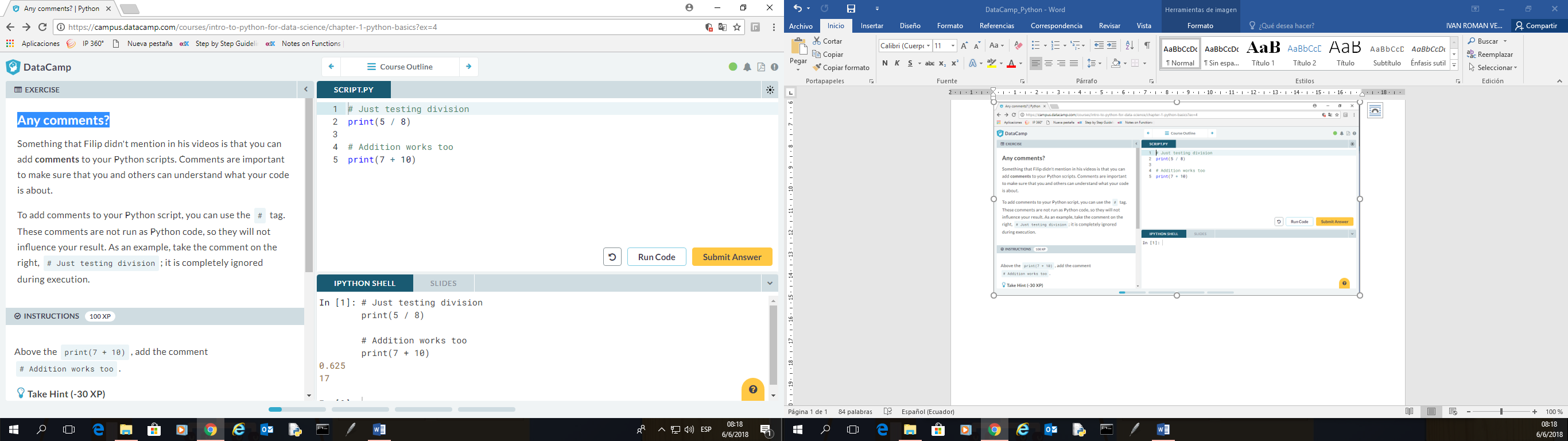
Any comments?

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, # Just testing division; it is completely ignored during execution.



# Just testing division

print(5 / 8)

**# Addition works too**

print(7 + 10)

# Python as a calculator

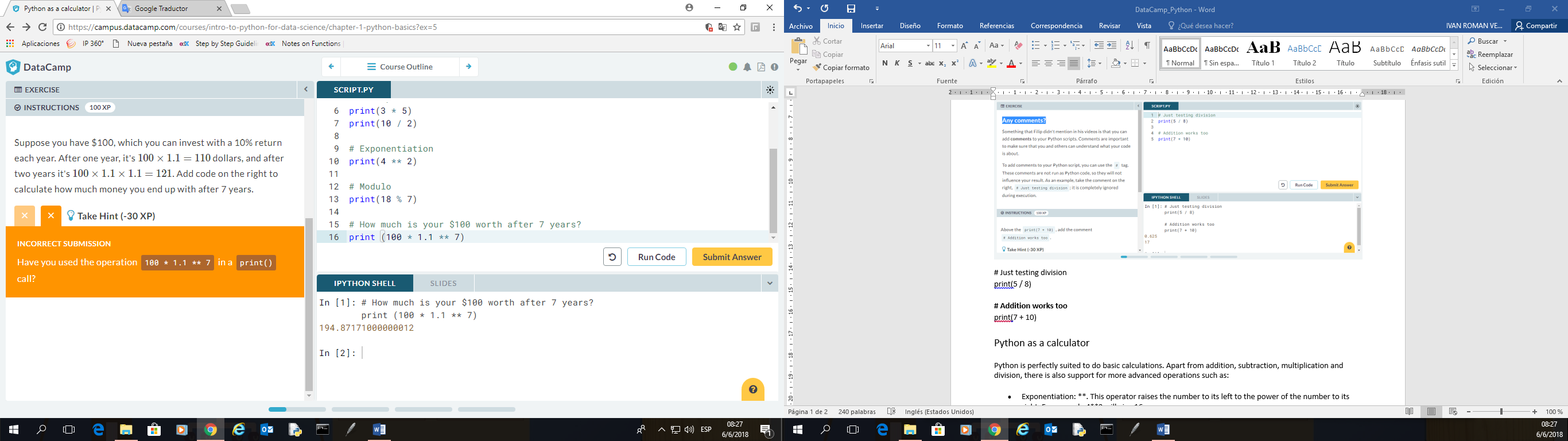
Python is perfectly suited to do basic calculations. Apart from addition, subtraction, multiplication and division, there is also support for more advanced operations such as:

* 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.

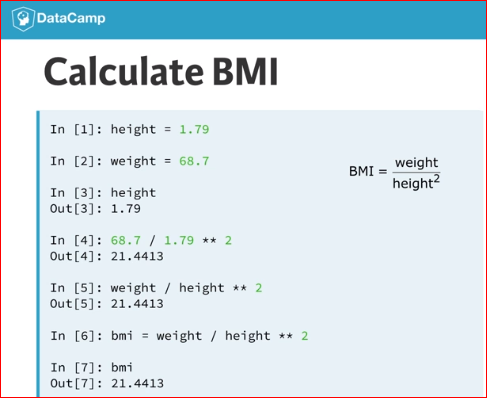
Suppose you have $100, which you can invest with a 10% return each year. After one year, it's 100×1.1=110100×1.1=110 dollars, and after two years it's 100×1.1×1.1=121100×1.1×1.1=121. Add code on the right to calculate how much money you end up with after 7 years.

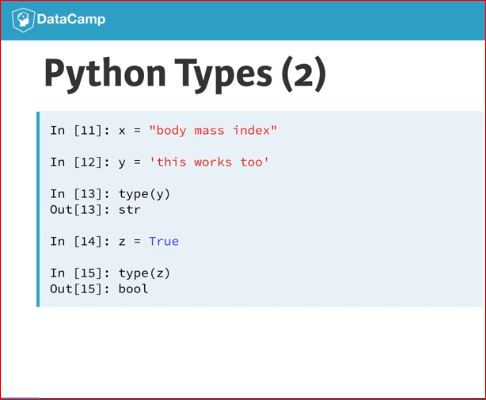
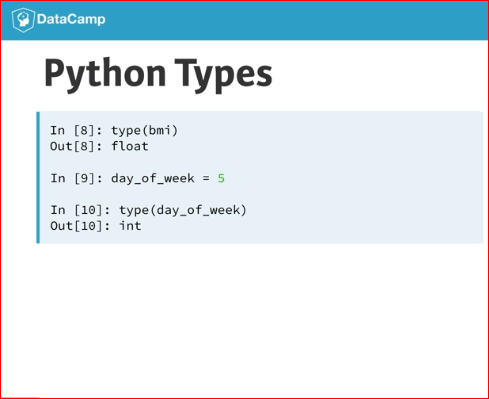
# How much is your $100 worth after 7 years?

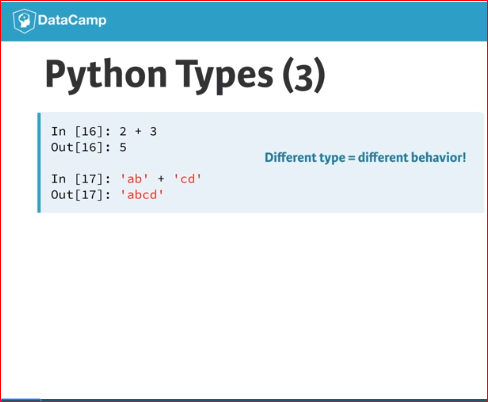
print (100 \* 1.1 \*\* 7)



# Variables & Types







# 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.

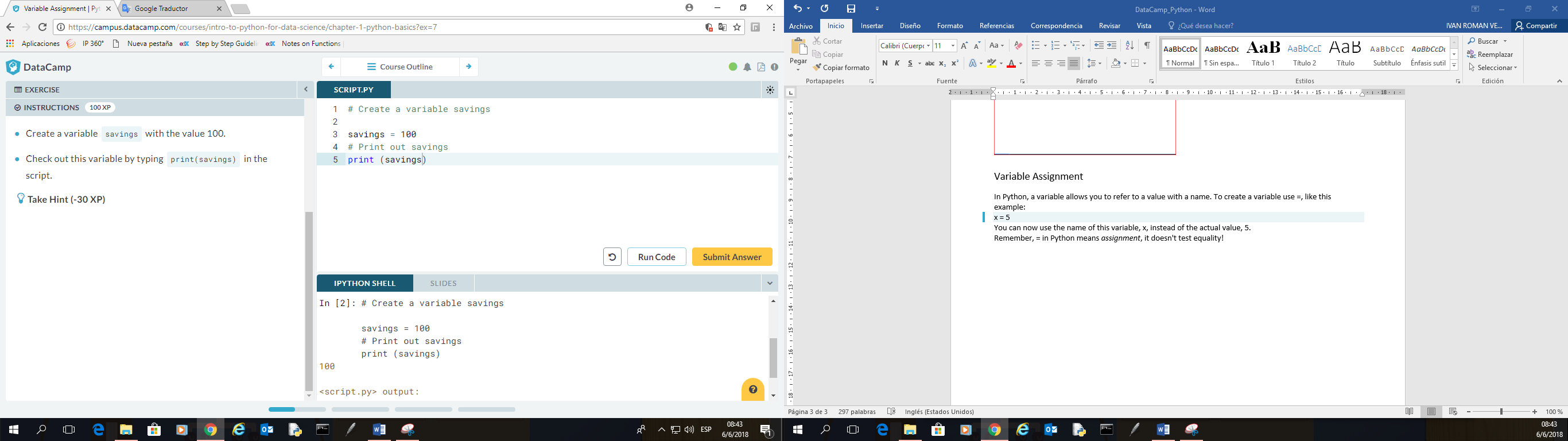
Remember, = in Python means *assignment*, it doesn't test equality!

# Create a variable savings

savings = 100

# Print out savings

print (savings)



# 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 \* 1.10 \*\* 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 1.10 and then redo the calculations!

# Create a variable savings

savings = 100

# Create a variable factor

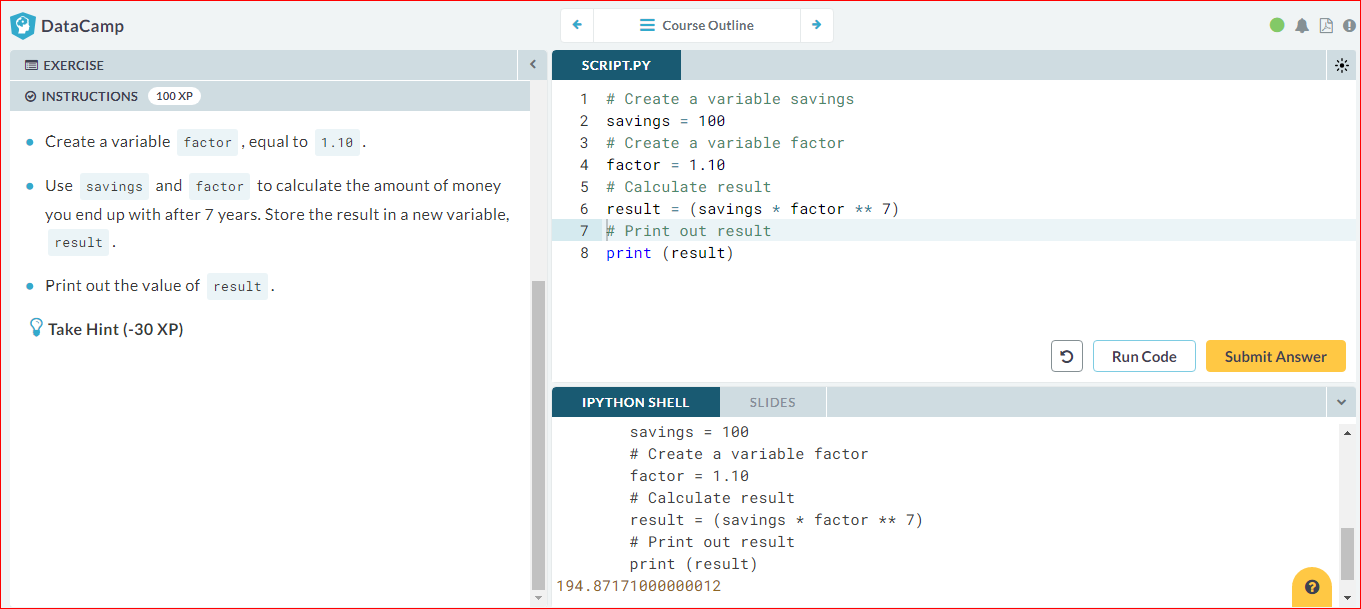
factor = 1.10

# Calculate result

result = (savings \* factor \*\* 7)

# Print out result

print (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. factor, with the value 1.10, is an example of a float.

Next to numerical data types, there are two other very common data types:

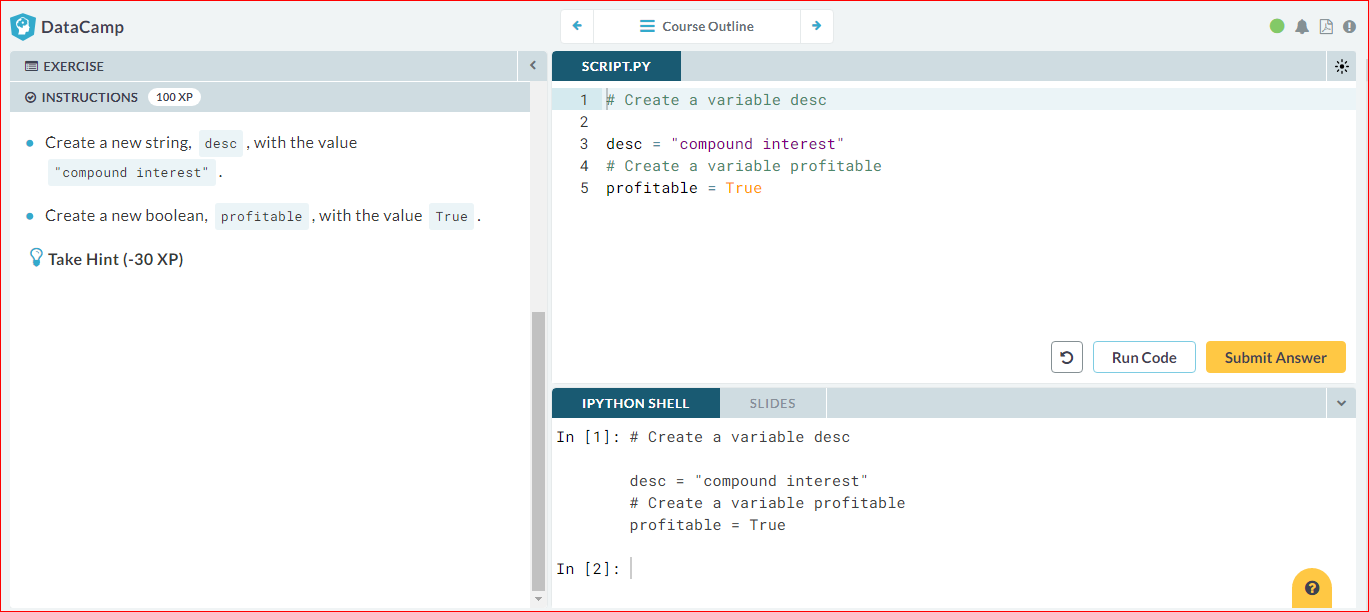
* str, or string: a type to represent text. You can use single or double quotes to build a string.
* bool, or boolean: a type to represent logical values. Can only be True or False (the capitalization is important!).

# Create a variable desc

desc = "compound interest"

# Create a variable profitable

profitable = True

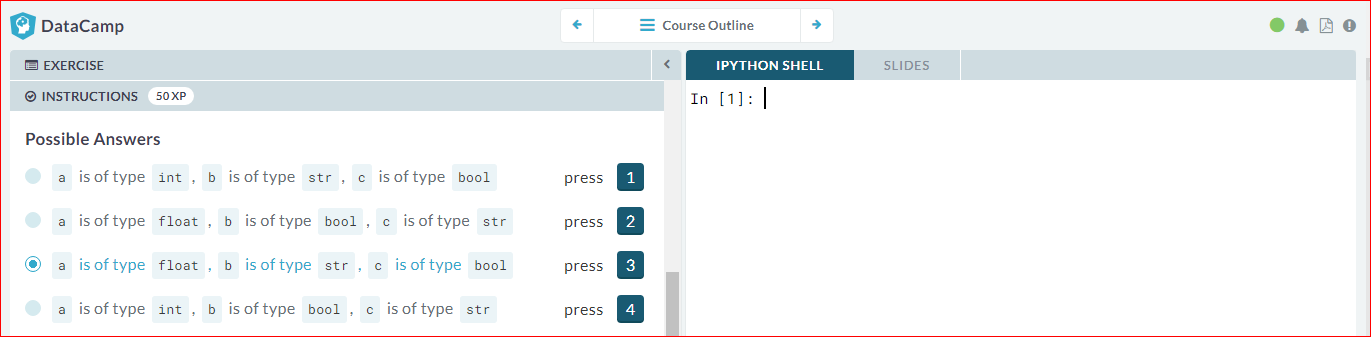


# Guess the type

To find out the type of a value or a variable that refers to that value, you can use the [type()](https://docs.python.org/3/library/functions.html#type) function. Suppose you've defined a variable a, but you forgot the type of this variable. To determine the type of a, simply execute:

type(a)

We already went ahead and created three variables: a, b and c. You can use the IPython shell on the right to discover their type. Which of the following options is correct?



# 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.

# Several variables to experiment with

savings = 100

factor = 1.1

desc = "compound interest"

# Assign product of factor and savings to year1

year1 = savings \* factor

# Print the type of year1

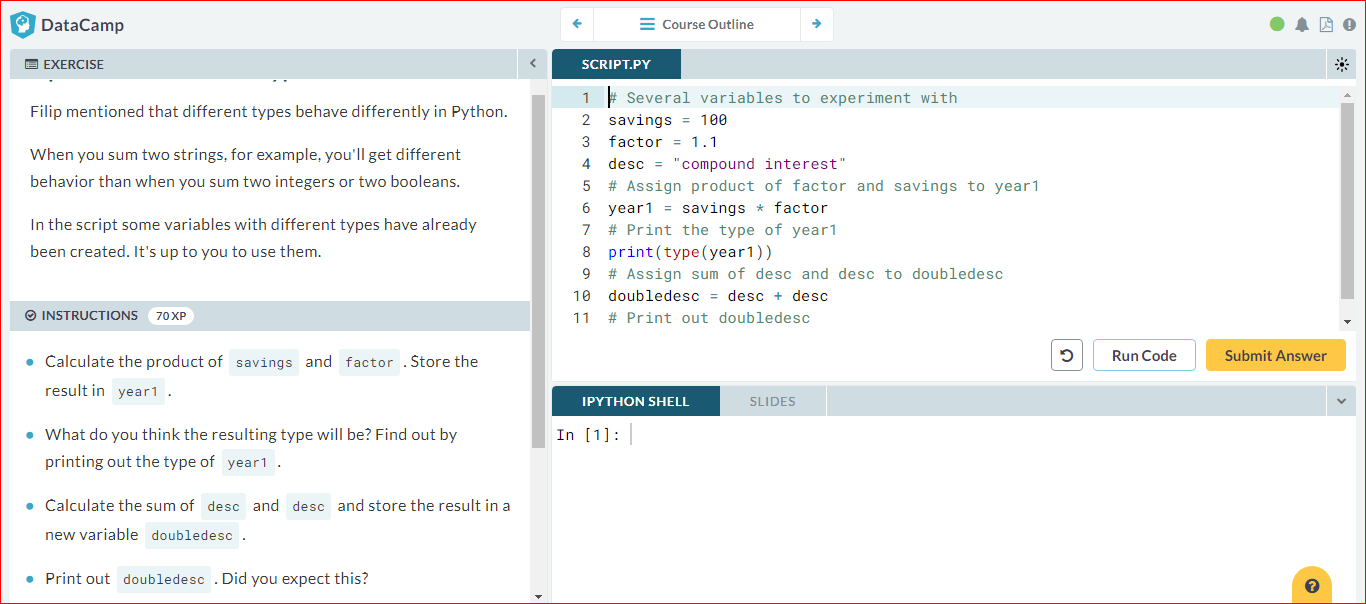
print(type(year1))

# Assign sum of desc and desc to doubledesc

doubledesc = desc + desc

# Print out doubledesc

print(doubledesc)



# 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()](https://docs.python.org/3/library/functions.html" \l "func-str" \t "_blank), to convert a value into a string. str(savings), for example, will convert the float savings to a string.

Similar functions such as [int()](https://docs.python.org/3/library/functions.html" \l "int" \t "_blank), [float()](https://docs.python.org/3/library/functions.html#float) and [bool()](https://docs.python.org/3/library/functions.html#bool) will help you convert Python values into any type.

# Definition of savings and result

savings = 100

result = 100 \* 1.10 \*\* 7

# Fix the printout

print("I started with $" + str(savings) + " and now have $" + str(result) + ". Awesome!")

# Definition of pi\_string

pi\_string = "3.1415926"

# Convert pi\_string into float: pi\_float

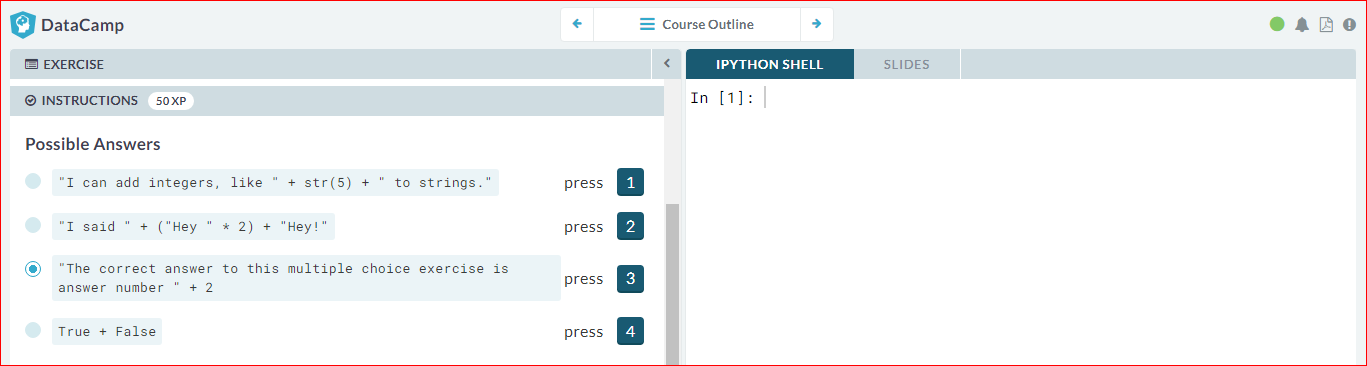
pi\_float = float(pi\_string)

print(pi\_float)

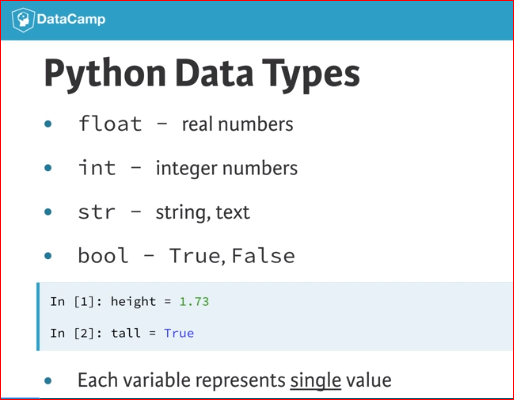
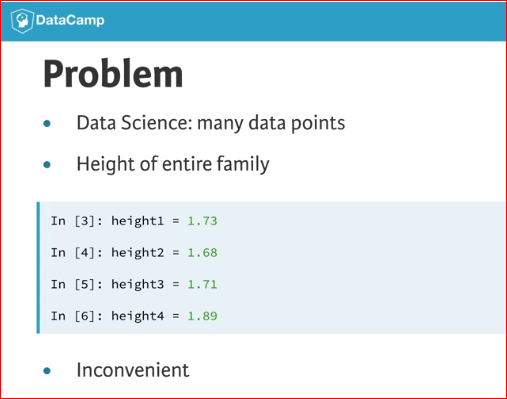


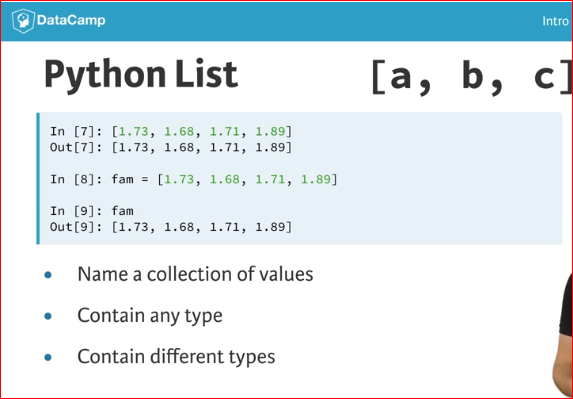
# Can Python handle everything?

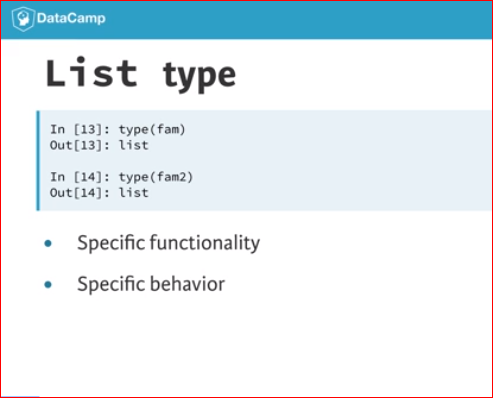
Now that you know something more about combining different sources of information, have a look at the four Python expressions below. Which one of these will throw an error? You can always copy and paste this code in the IPython Shell to find out!



# Lists, what are they?



# 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.

# area variables (in square meters)

hall = 11.25

kit = 18.0

liv = 20.0

bed = 10.75

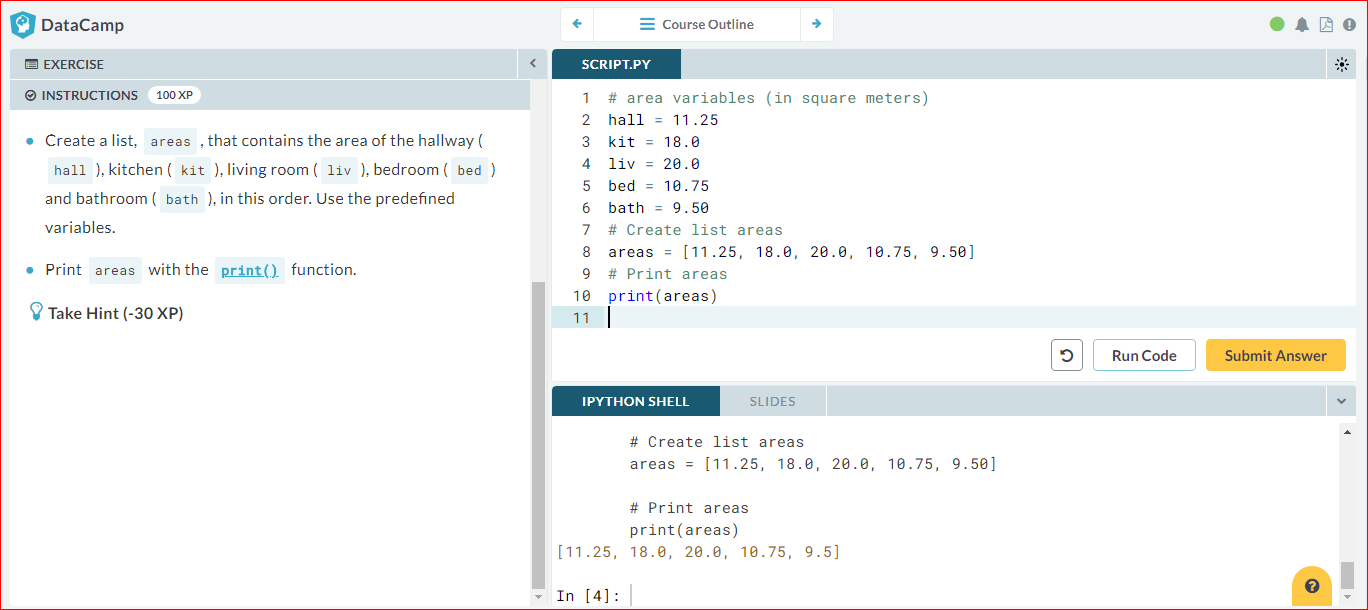
bath = 9.50

# Create list areas

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

# Print areas

print(areas)



# Create list with different types

A list can contain any Python type. Although it's not really common, a list can also contain a mix of Python types including strings, floats, booleans, etc.

The printout of the previous exercise wasn't really satisfying. It's just a list of numbers representing the areas, but you can't tell which area corresponds to which part of your house.

The code on the right is the start of a solution. For some of the areas, the name of the corresponding room is already placed in front. Pay attention here! "bathroom" is a string, while bathis a variable that represents the float 9.50 you specified earlier.

# area variables (in square meters)

hall = 11.25

kit = 18.0

liv = 20.0

bed = 10.75

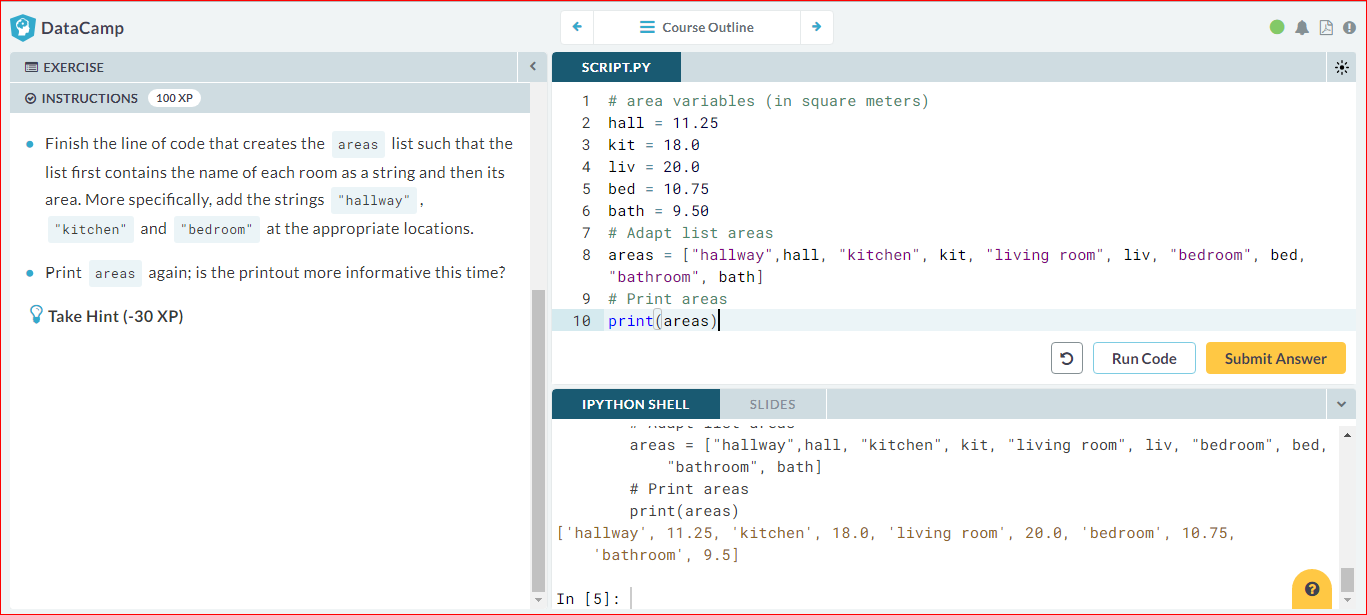
bath = 9.50

# Adapt list areas

areas = ["hallway",hall, "kitchen", kit, "living room", liv, "bedroom", bed, "bathroom", bath]

# Print areas

print(areas)



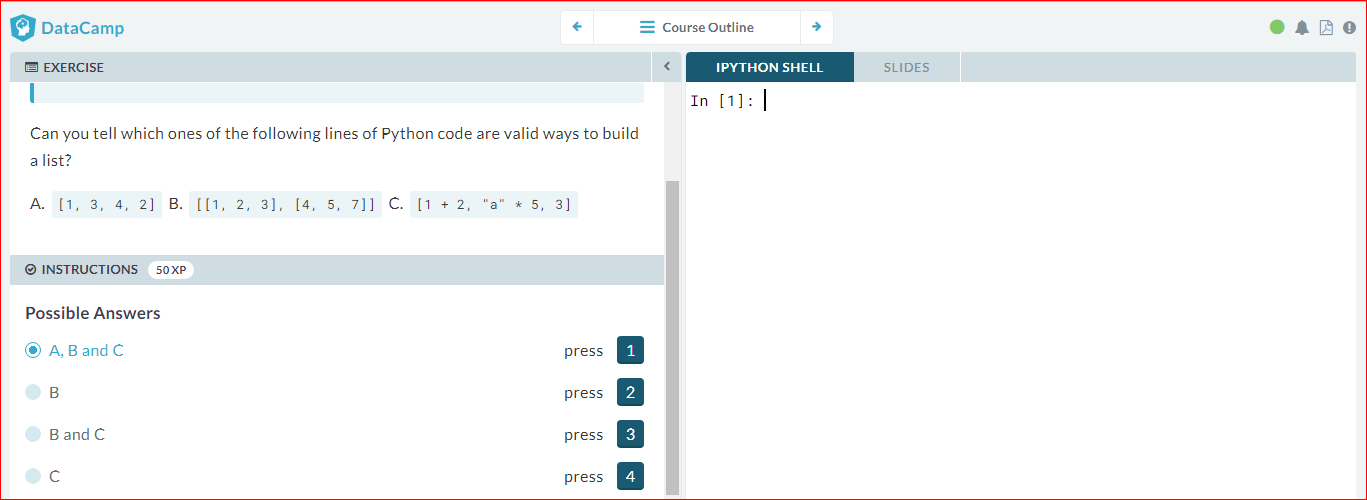
# Select the valid list

A list can contain any Python type. But a list itself is also a Python type. That means that a list can also contain a list! Python is getting funkier by the minute, but fear not, just remember the list syntax:

my\_list = [el1, el2, el3]

Can you tell which ones of the following lines of Python code are valid ways to build a list?

A. [1, 3, 4, 2] B. [[1, 2, 3], [4, 5, 7]] C. [1 + 2, "a" \* 5, 3]



# List of lists

As a data scientist, you'll often be dealing with a lot of data, and it will make sense to group some of this data.

Instead of creating a flat list containing strings and floats, representing the names and areas of the rooms in your house, you can create a list of lists. The script on the right can already give you an idea.

Don't get confused here: "hallway" is a string, while hall is a variable that represents the float 11.25 you specified earlier.

# area variables (in square meters)

hall = 11.25

kit = 18.0

liv = 20.0

bed = 10.75

bath = 9.50

# house information as list of lists

house = [["hallway", hall],

["kitchen", kit],

["living room", liv],

["bedroom", bed],

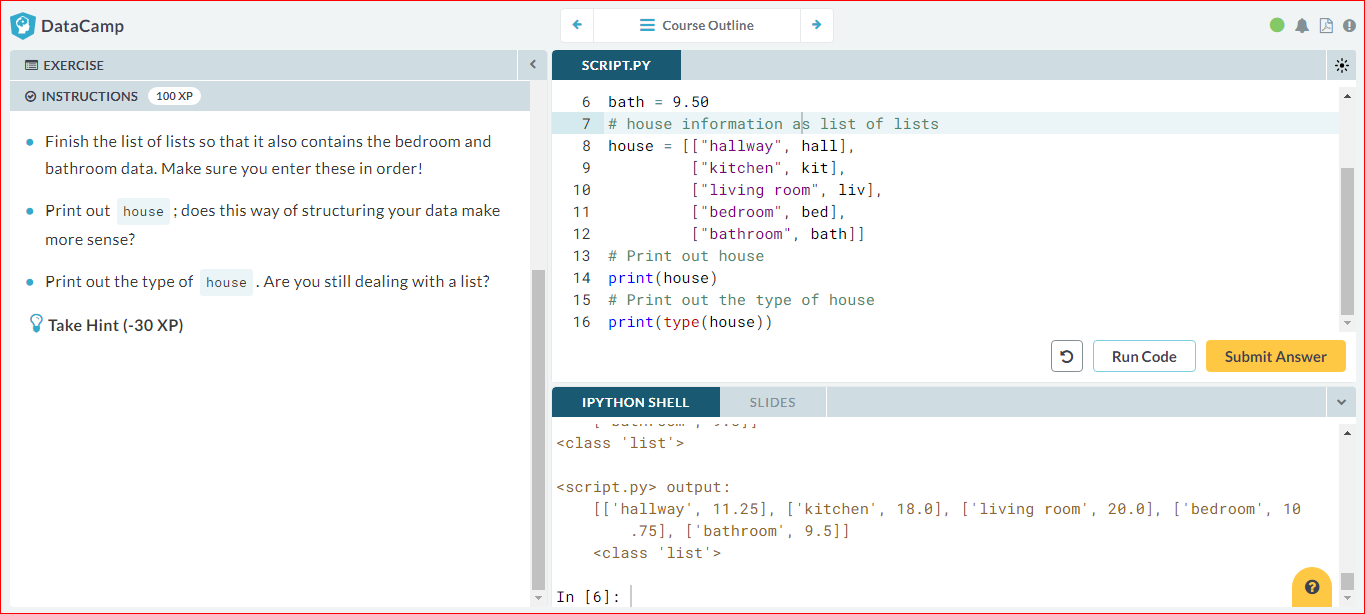
["bathroom", bath]]

# Print out house

print(house)

# Print out the type of house

print(type(house))



# Subsetting lists

# 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?

# Create the areas list

areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Print out second element from areas

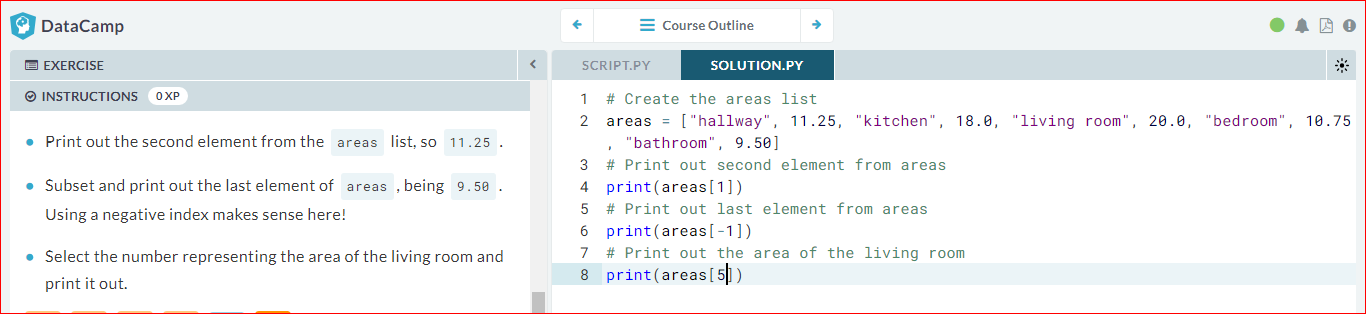
print(areas[1])

# Print out last element from areas

print(areas[-1])

# Print out the area of the living room

print(areas[5])



# 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])

# Create the areas list

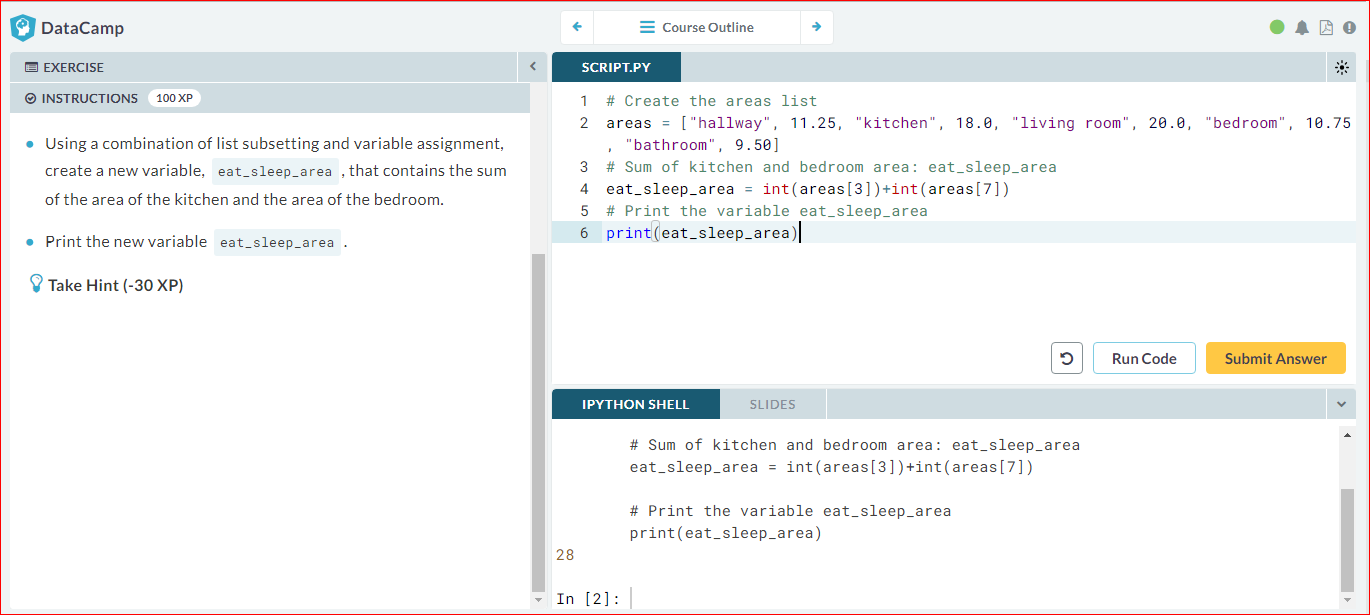
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Sum of kitchen and bedroom area: eat\_sleep\_area

eat\_sleep\_area = areas[3]+areas[7]

# Print the variable eat\_sleep\_area

print(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.

# Create the areas list

areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Use slicing to create downstairs

downstairs = areas[0:6]

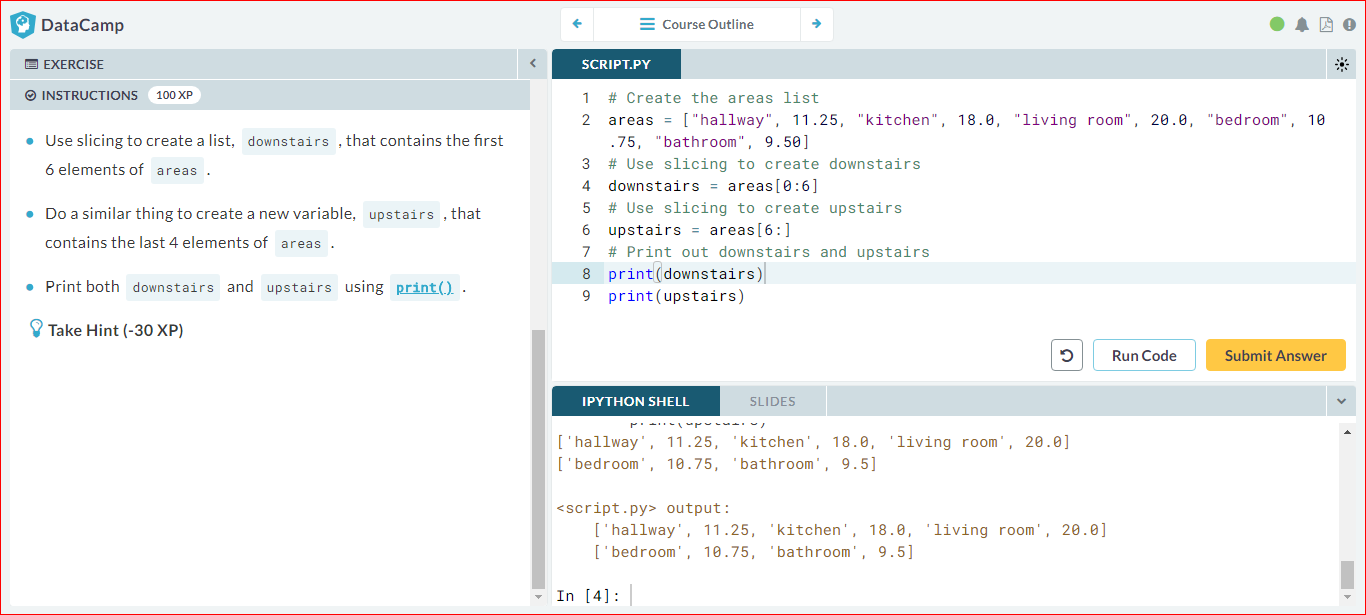
# Use slicing to create upstairs

upstairs = areas[6:]

# Print out downstairs and upstairs

print(downstairs)

print(upstairs)



# 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[:]

# Create the areas list

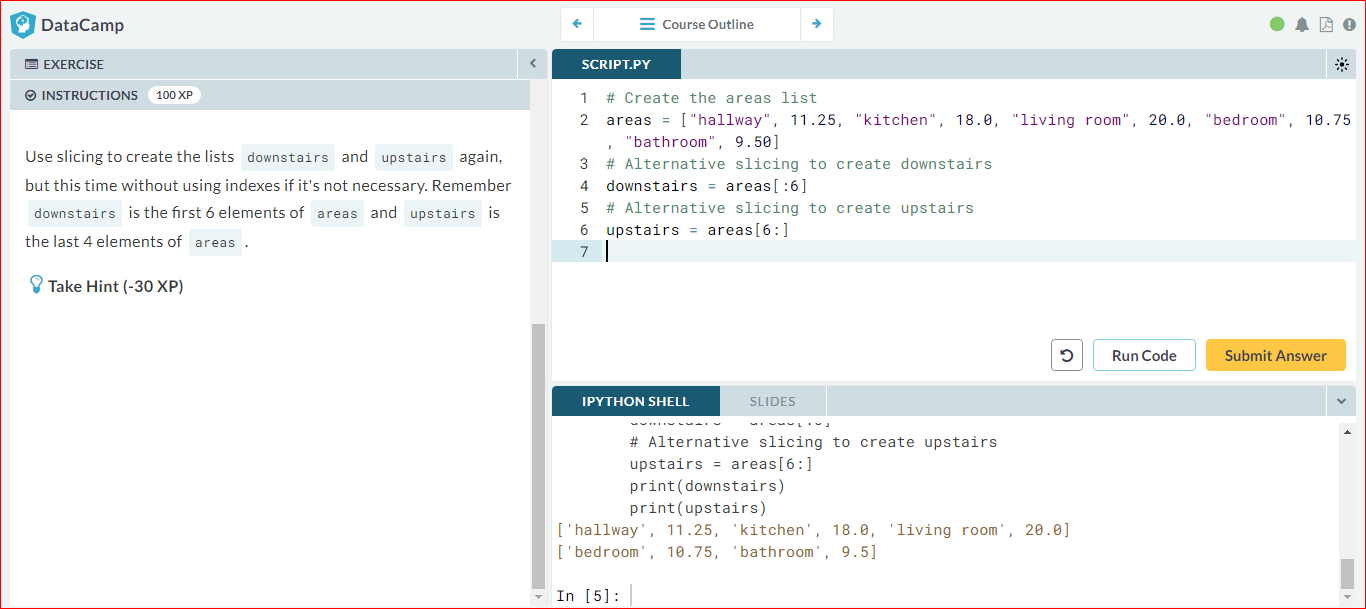
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Alternative slicing to create downstairs

downstairs = areas[:6]

# Alternative slicing to create upstairs

upstairs = areas[6:]



# Subsetting lists of lists

You saw before that a Python list can contain practically anything; even other lists! To subset lists of lists, you can use the same technique as before: square brackets. Try out the commands in the following code sample in the IPython Shell:

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

["d", "e", "f"],

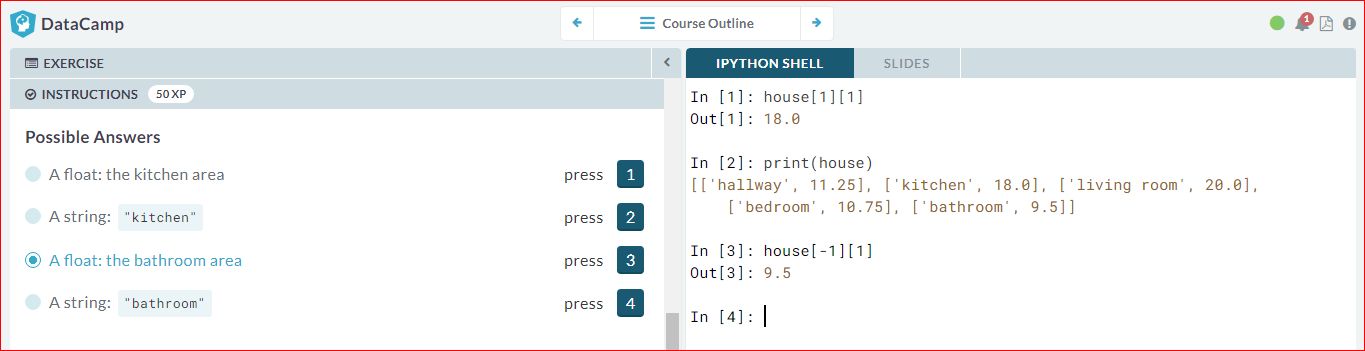
["g", "h", "i"]]

x[2][0]

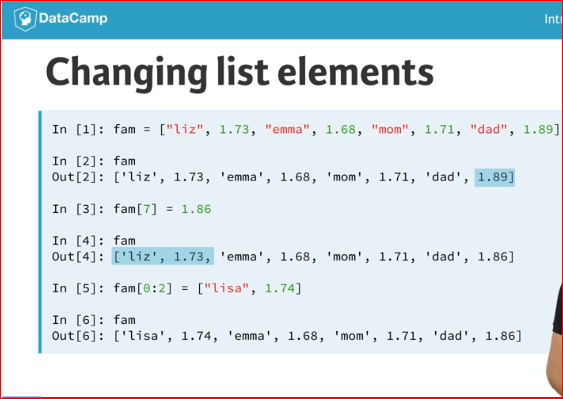
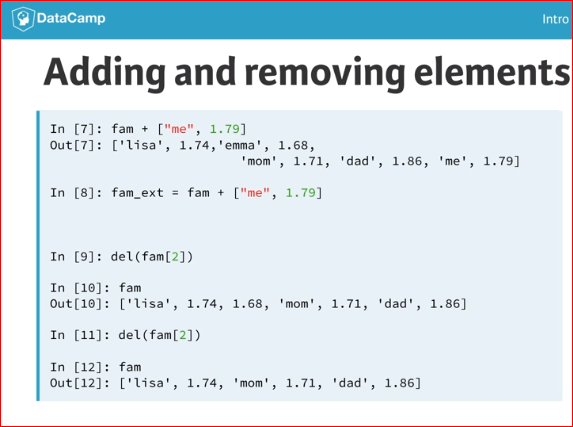
x[2][:2]

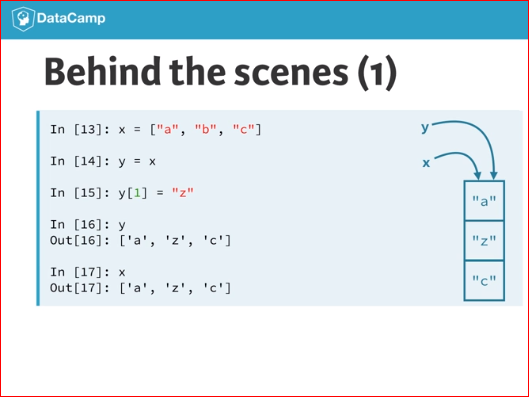
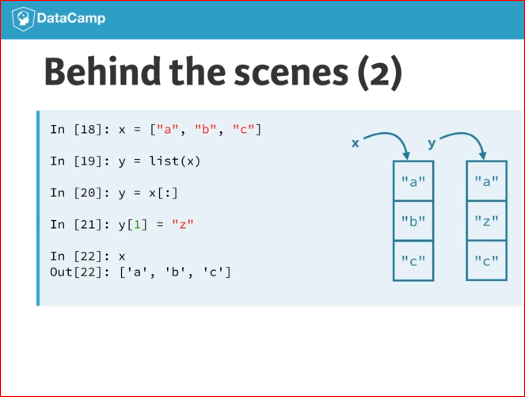
x[2] results in a list, that you can subset again by adding additional square brackets.

What will house[-1][1] return? house, the list of lists that you created before, is already defined for you in the workspace. You can experiment with it in the IPython Shell.



# List Manipulation

# 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.

# Create the areas list

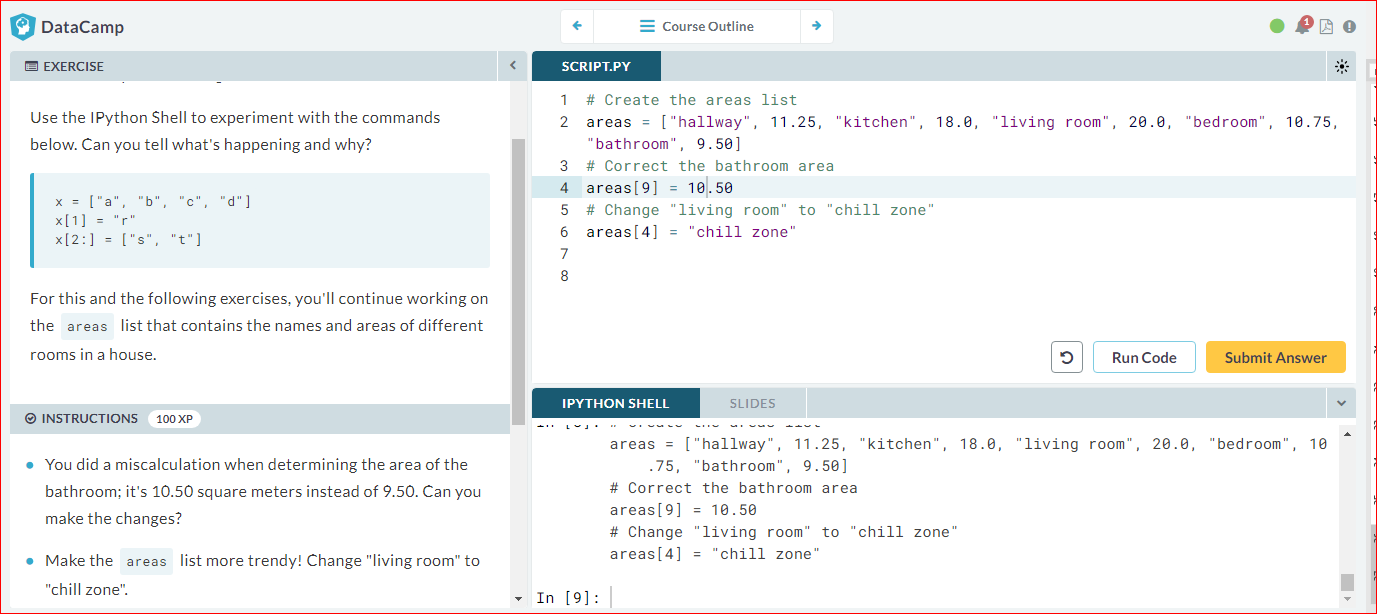
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0, "bedroom", 10.75, "bathroom", 9.50]

# Correct the bathroom area

areas[9] = 10.50

# Change "living room" to "chill zone"

areas[4] = "chill zone"



# 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?

# 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]

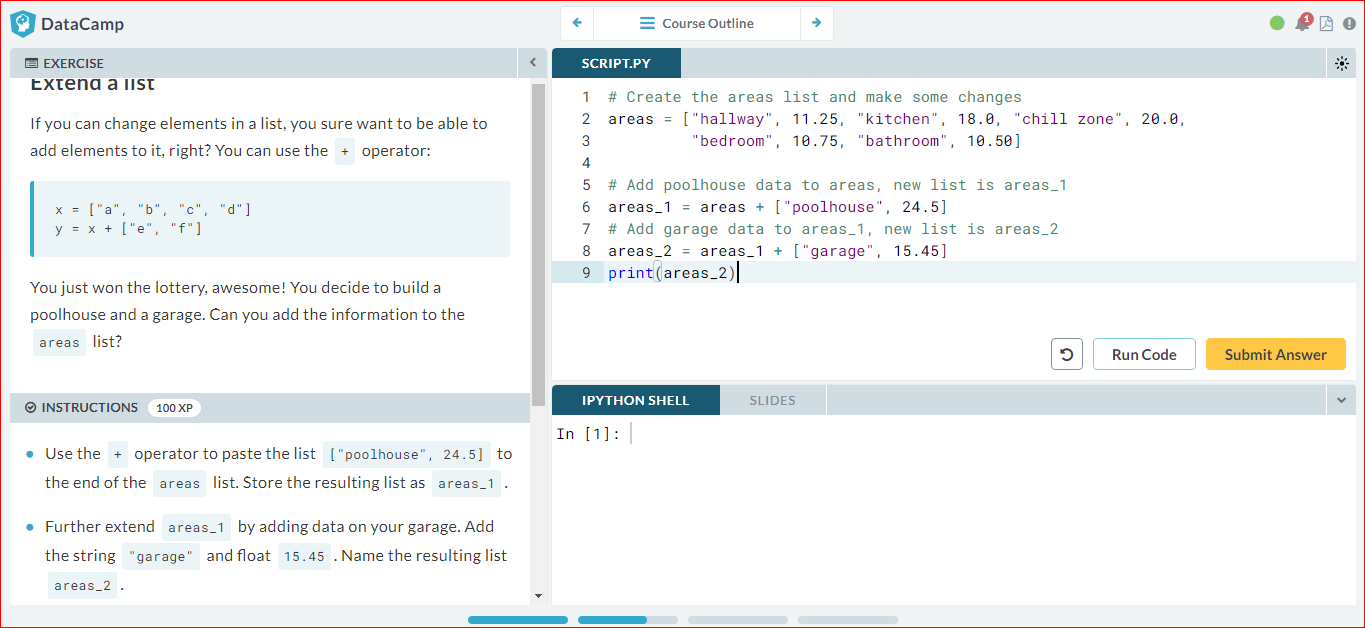
# Add poolhouse data to areas, new list is areas\_1

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

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

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

print(areas\_2)



# Delete list elements

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

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

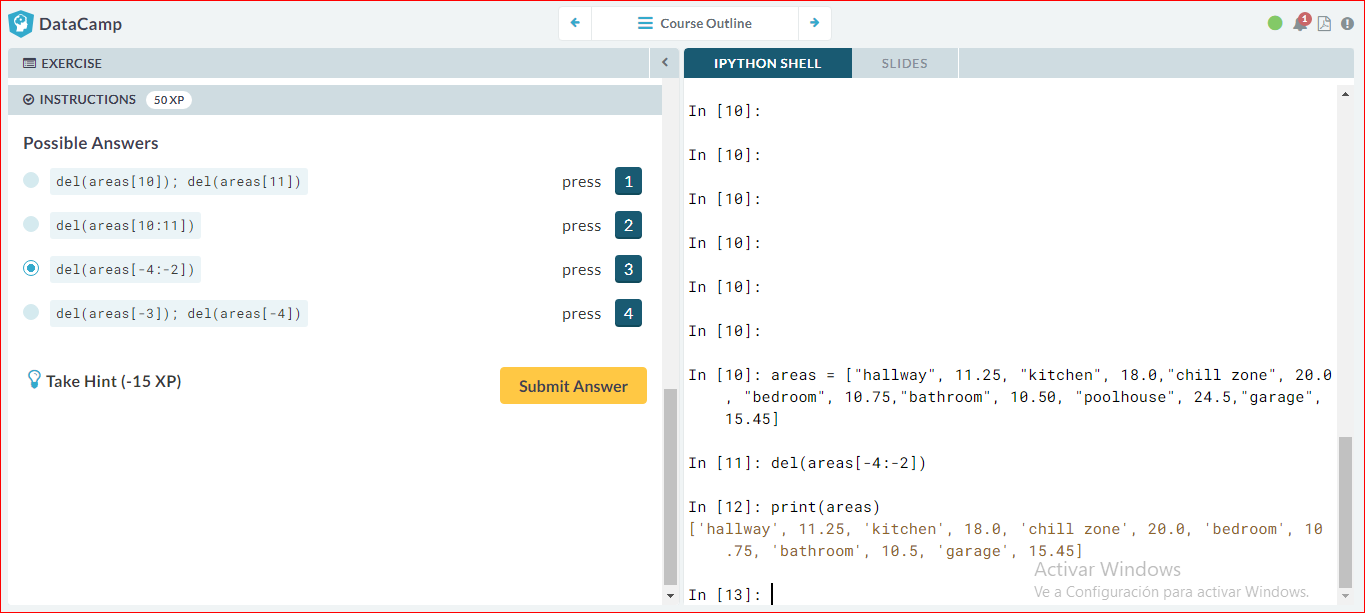
command2

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

areas = ["hallway", 11.25, "kitchen", 18.0,"chill zone", 20.0, "bedroom", 10.75,"bathroom", 10.50, "poolhouse", 24.5,"garage", 15.45]

del(areas[-4:-2])

print(areas)



# 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 to also take 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" \l "func-list" \t "_blank) or by using [:].

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

# Create areas\_copy

**areas\_copy = list(areas)**

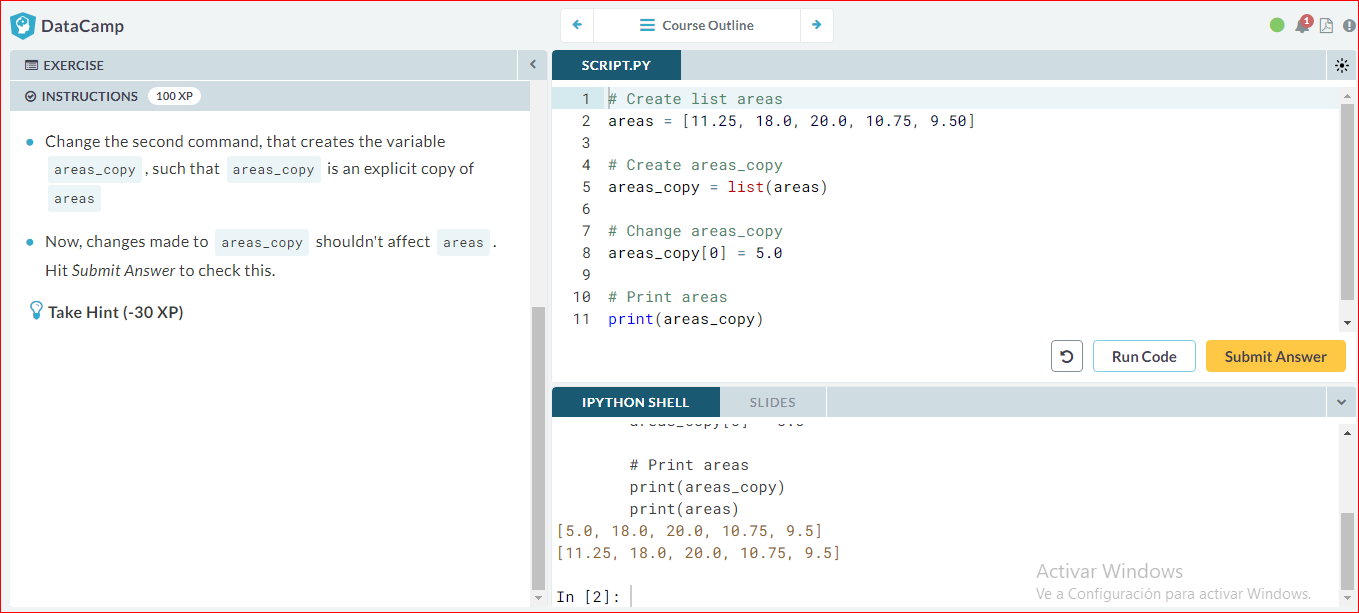
# Change areas\_copy

areas\_copy[0] = 5.0

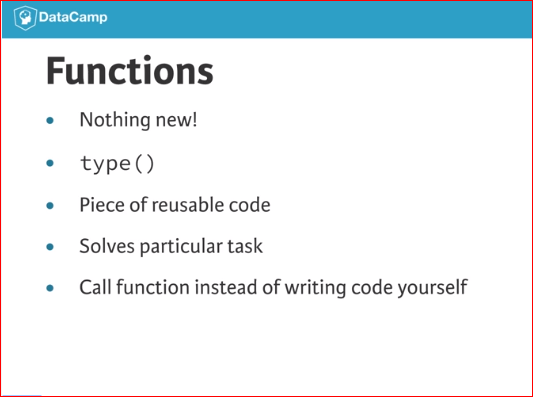
# Print areas

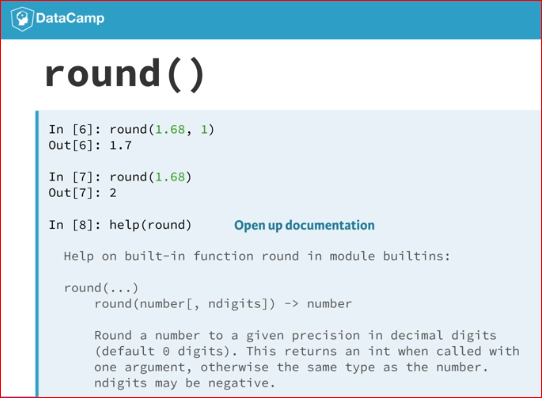
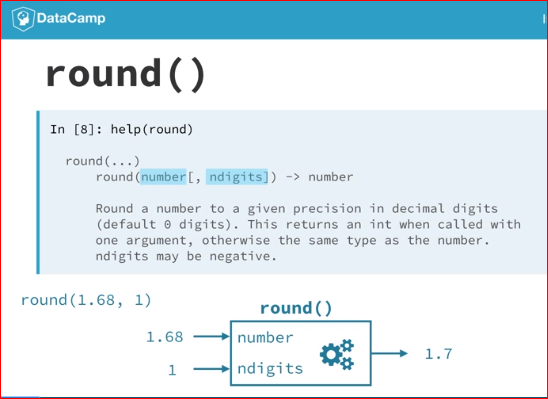
print(areas\_copy)

print(areas)



# Functions

# 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)

# Create variables var1 and var2

var1 = [1, 2, 3, 4]

var2 = True

# Print out type of var1

print(type(var1))

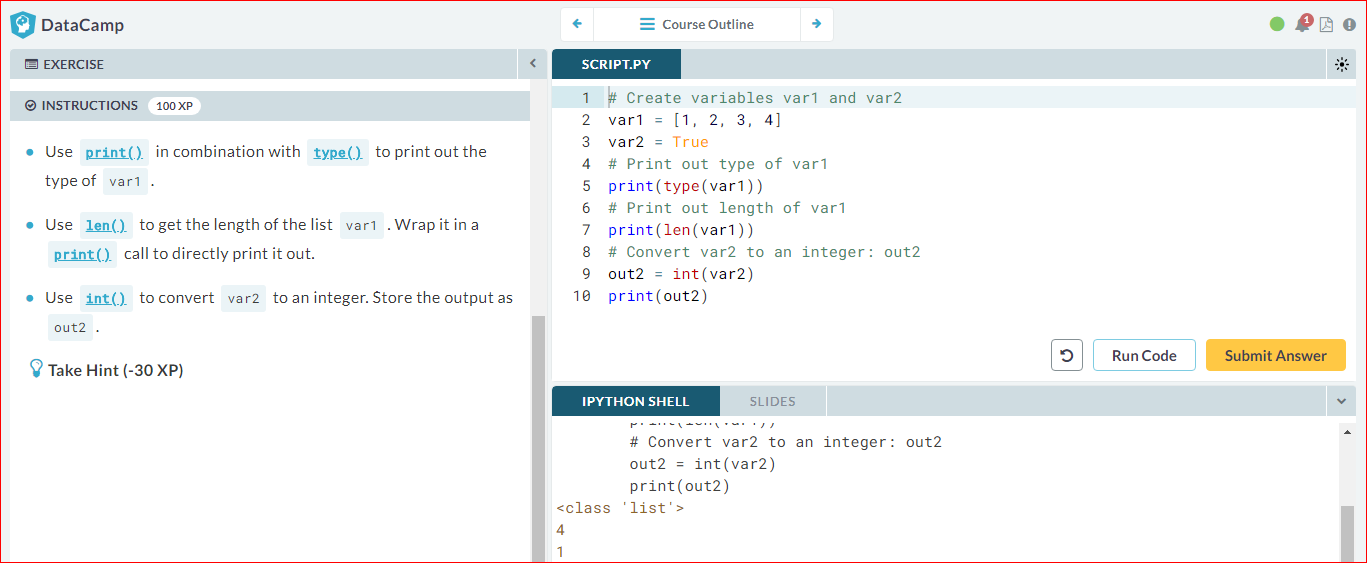
# Print out length of var1

print(len(var1))

# Convert var2 to an integer: out2

out2 = int(var2)

print(out2)



# Help!

Maybe you already know the name of a Python function, but you still have to figure out how to use it. Ironically, you have to ask for information about a function with another function: [help()](https://docs.python.org/3/library/functions.html#help). In IPython specifically, you can also use ? before the function name.

To get help on the [max()](https://docs.python.org/3/library/functions.html#max) function, for example, you can use one of these calls:

help(max)

?max

Use the Shell on the right to open up the documentation on [complex()](https://docs.python.org/3/library/functions.html#complex). Which of the following statements is true?



# 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 reverse argument, 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 reversewithout 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.

# Create lists first and second

first = [11.25, 18.0, 20.0]

second = [10.75, 9.50]

# Paste together first and second: full

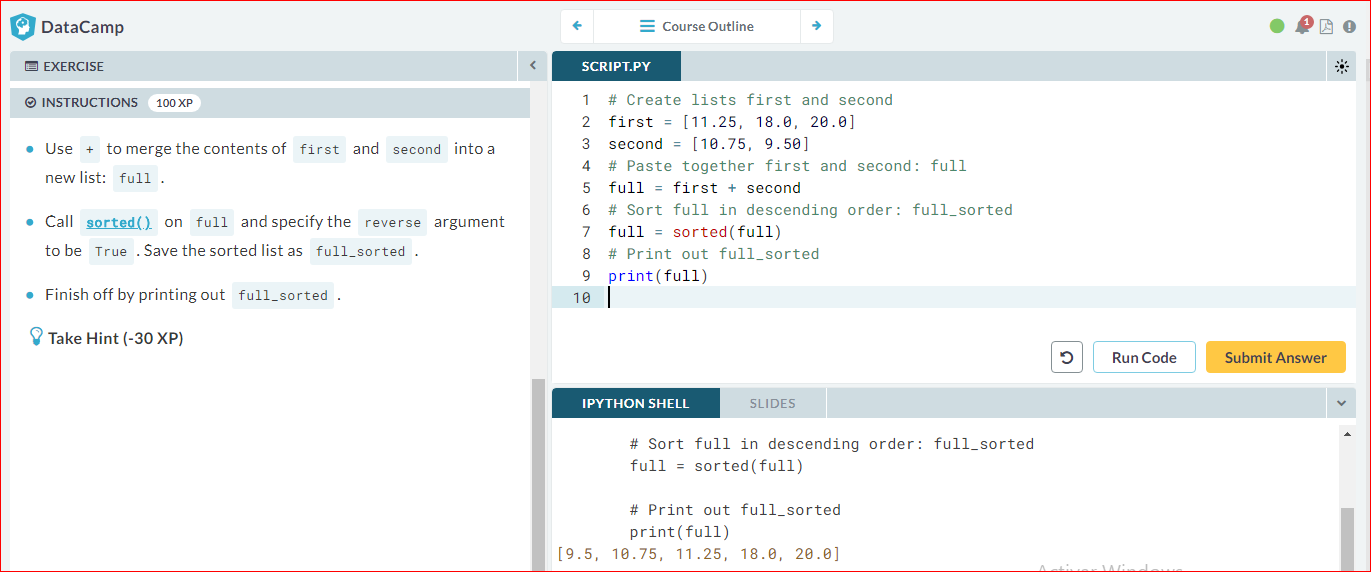
full = first + second

# Sort full in descending order: full\_sorted

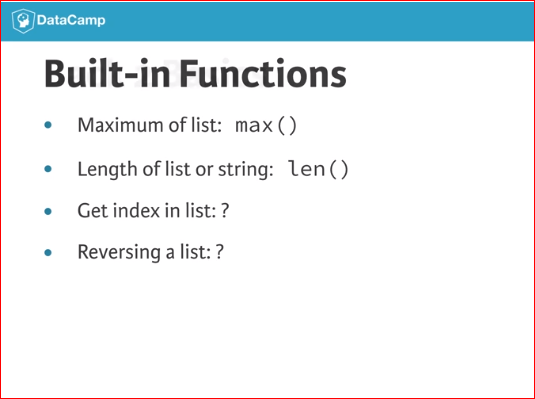
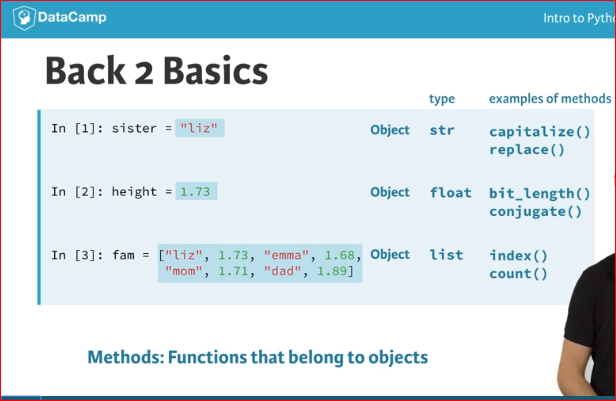
full\_sorted = sorted(full, reverse=True)

# Print out full\_sorted

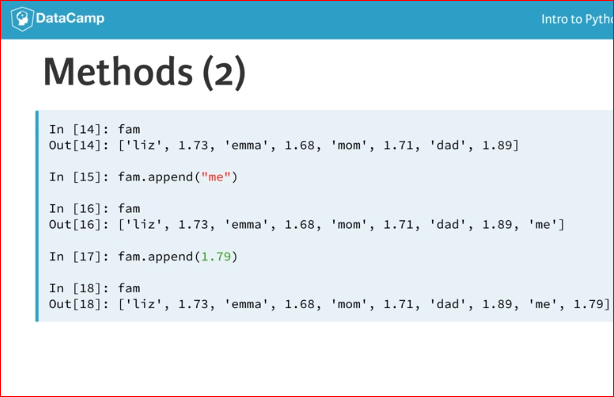
print(full)

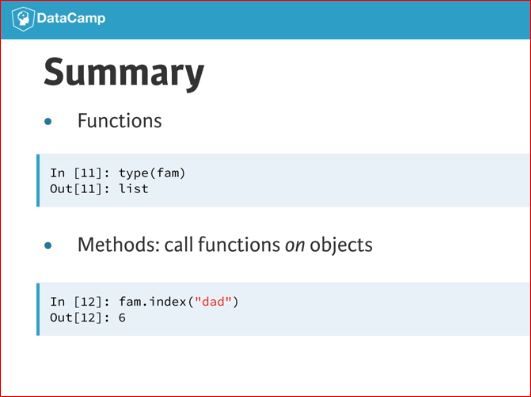


# Methods



# 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 room has already been created for you to experiment with.

# string to experiment with: room

room = "poolhouse"

# Use upper() on room: room\_up

room\_up = room.upper()

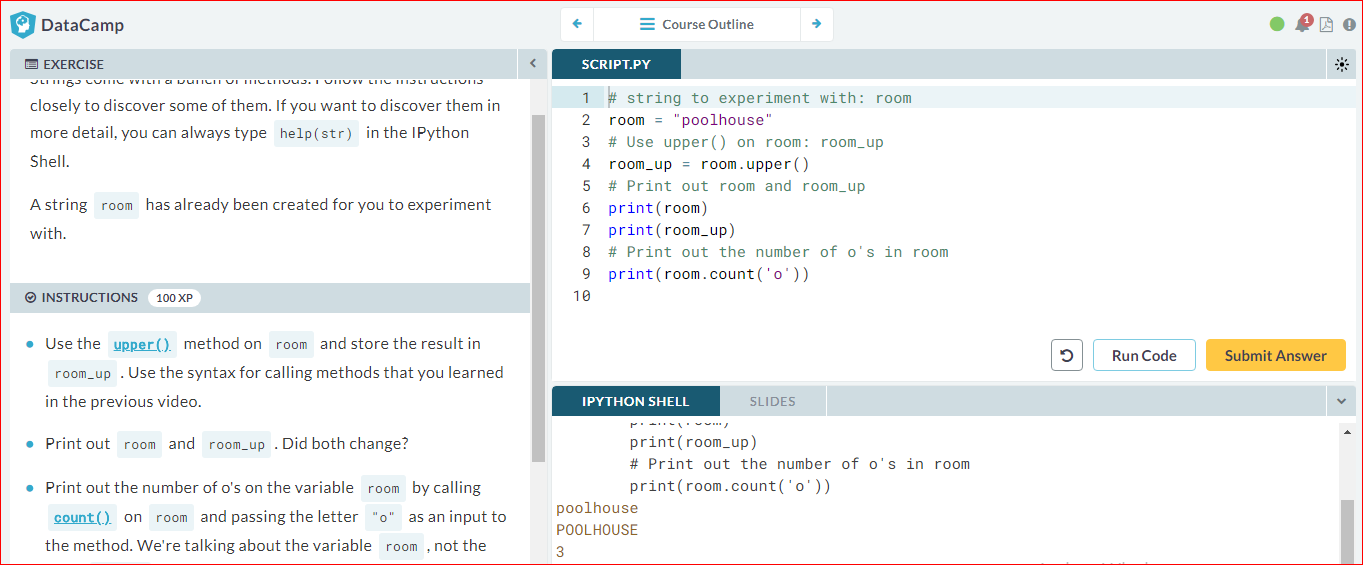
# Print out room and room\_up

print(room)

print(room\_up)

# Print out the number of o's in room

print(room.count('o'))



# 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.

# Create list areas

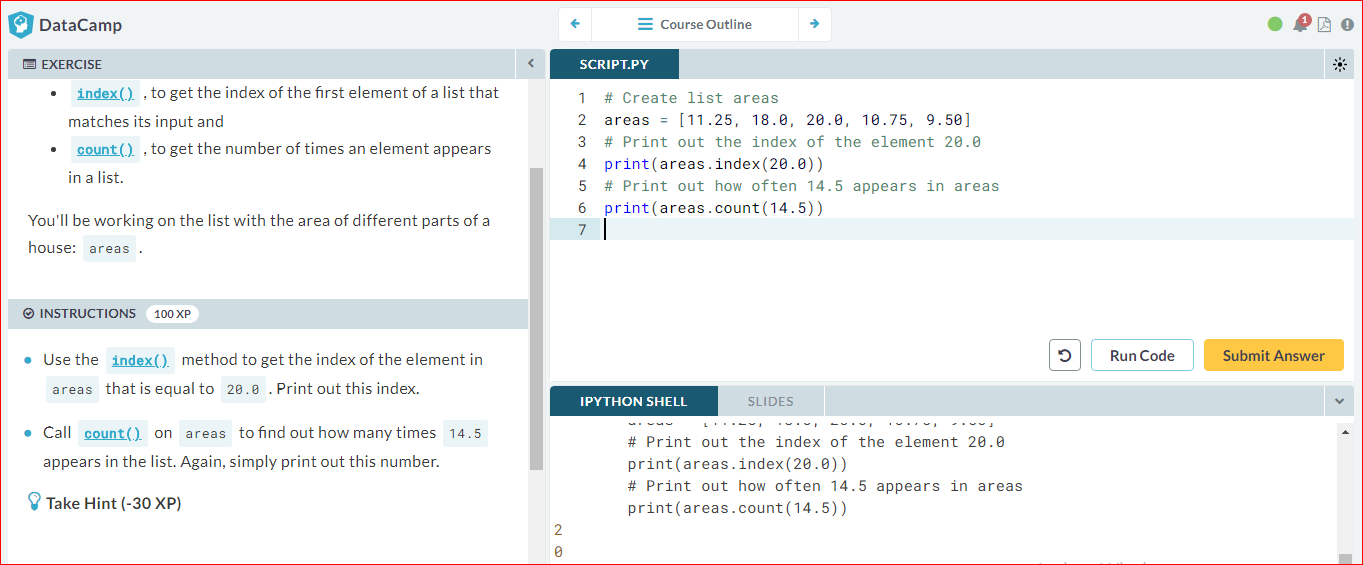
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0

print(areas.index(20.0))

# Print out how often 14.5 appears in areas

print(areas.count(14.5))



# 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.

# 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

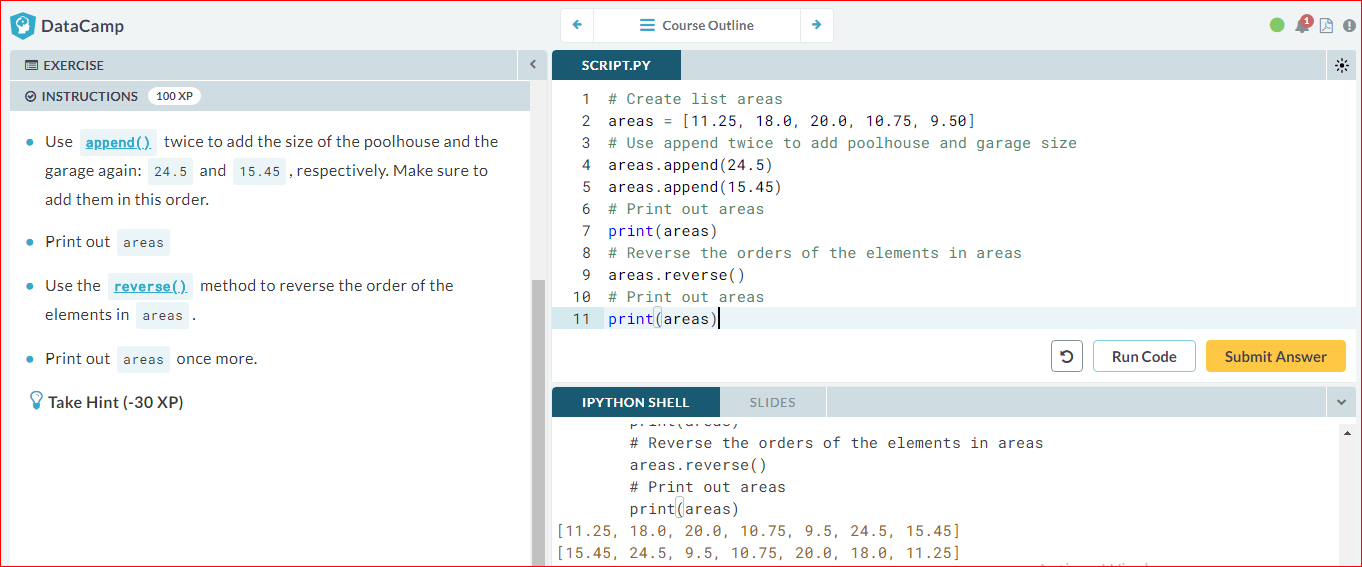
print(areas)

# Reverse the orders of the elements in areas

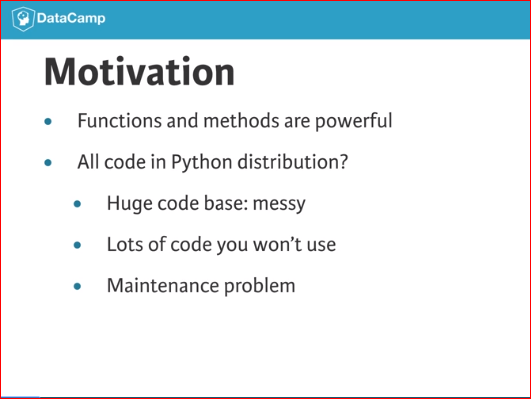
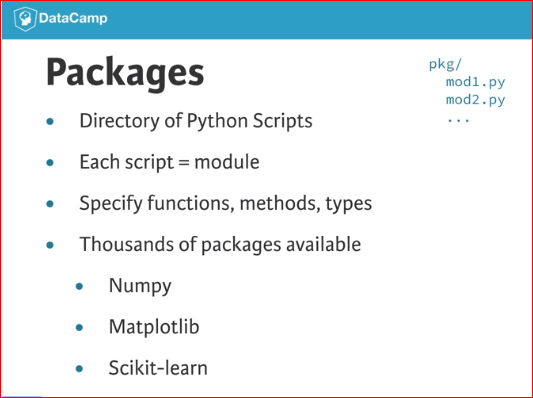
areas.reverse()

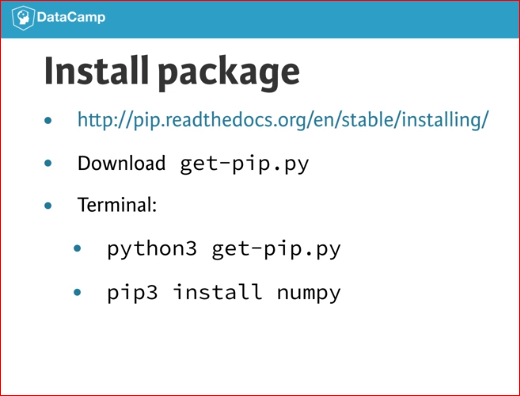
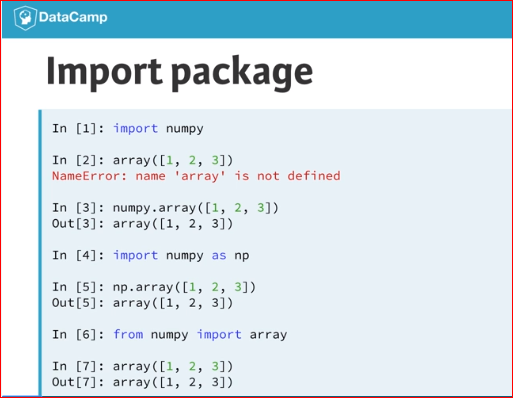
# Print out areas

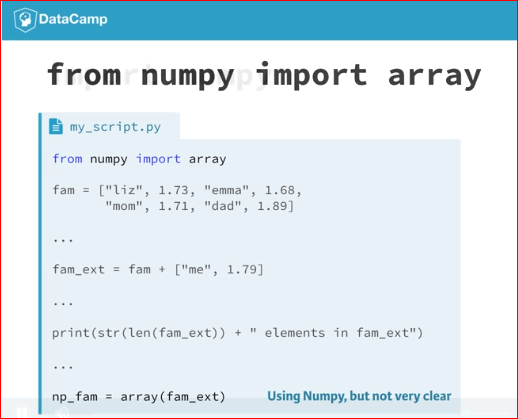
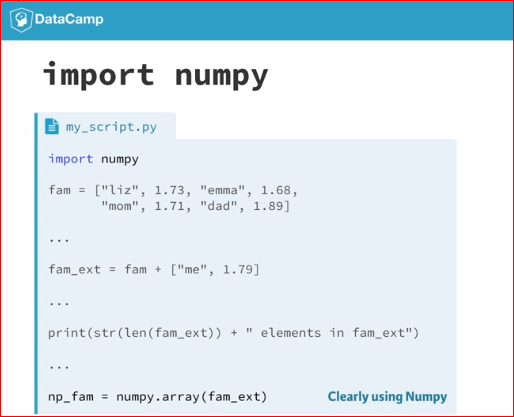
print(areas)



# Packages

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 CC and 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.

# Import the math package

import math

# Calculate C

C = 0

C = 2 \* math.pi \* r

# Calculate A

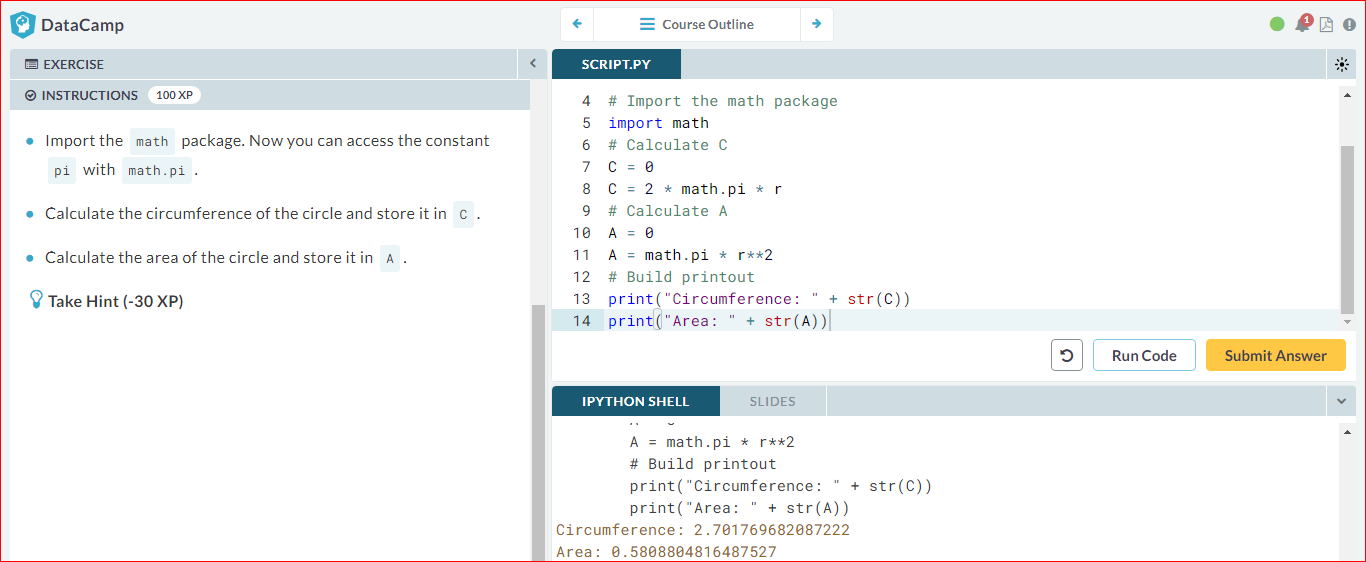
A = 0

A = math.pi \* r\*\*2

# Build printout

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

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



# Selective import

General imports, like import math, make **all** functionality from the math package 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.

# Definition of radius

r = 192500

# Import radians function of math package

from math import radians

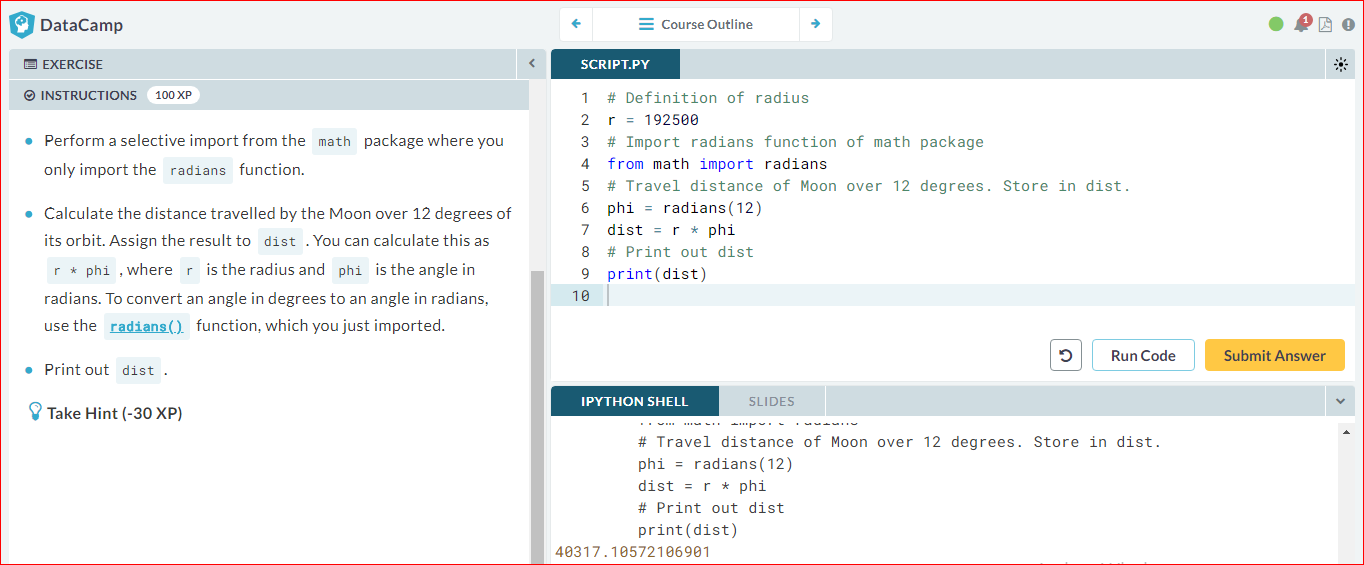
# Travel distance of Moon over 12 degrees. Store in dist.

phi = radians(12)

dist = r \* phi

# Print out dist

print(dist)



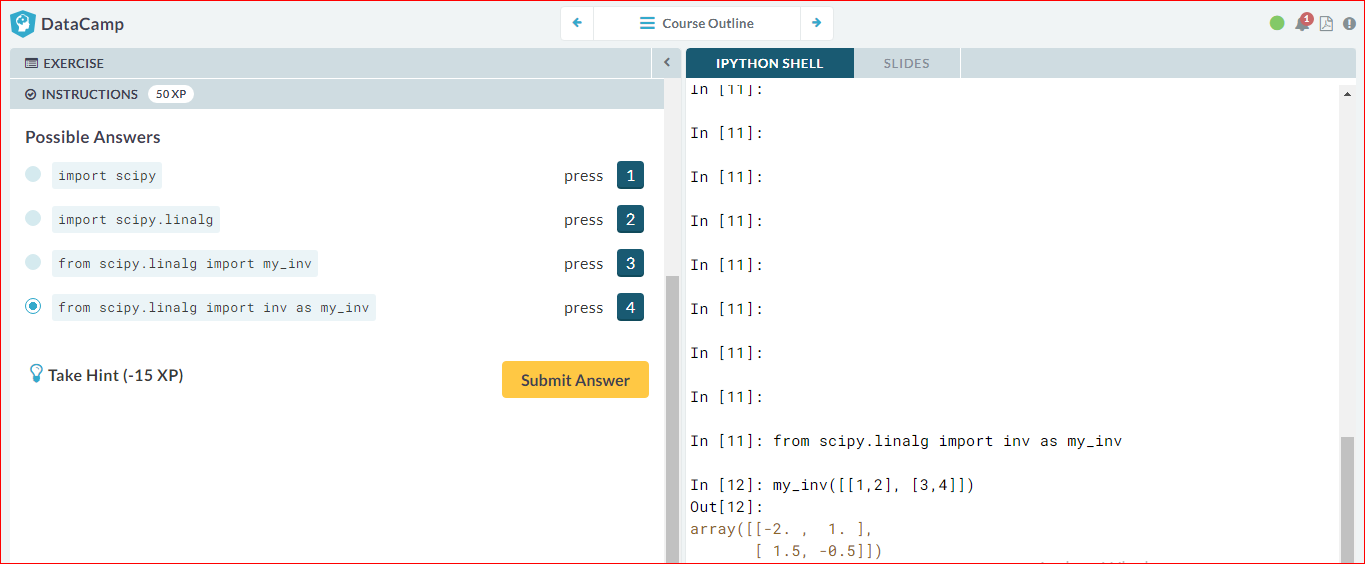
# Different ways of importing

There are several ways to import packages and modules into Python. Depending on the import call, you'll have to use different Python code.

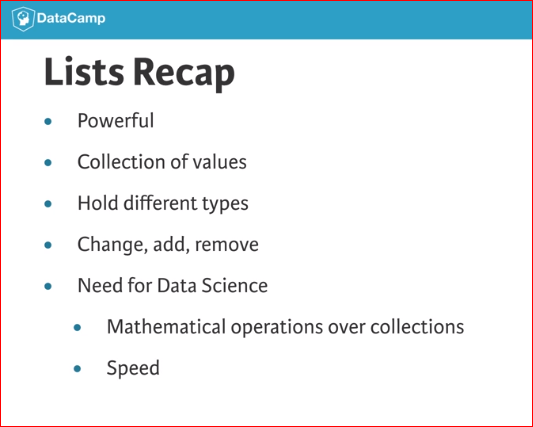
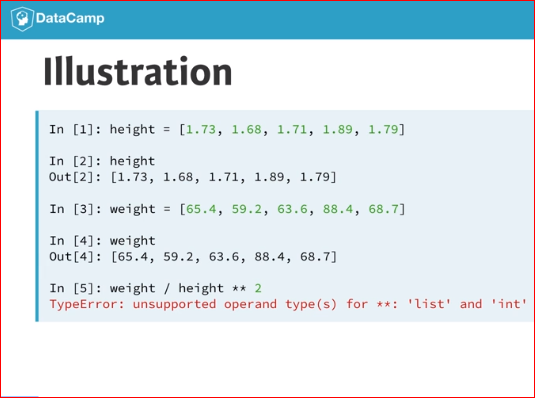
Suppose you want to use the function [inv()](http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.inv.html" \t "_blank), which is in the linalg subpackage of the scipy package. You want to be able to use this function as follows:

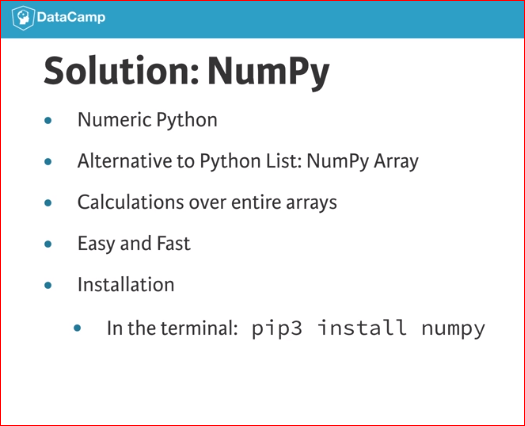
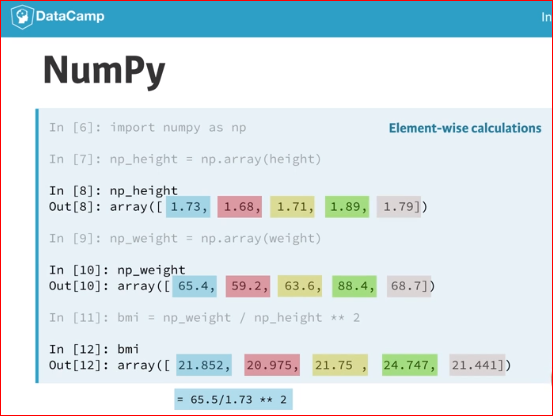
my\_inv([[1,2], [3,4]])

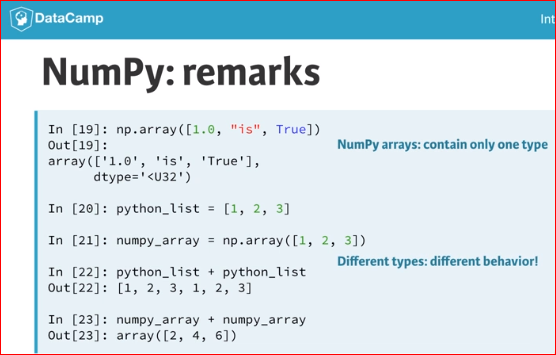
Which import statement will you need in order to run the above code without an error?

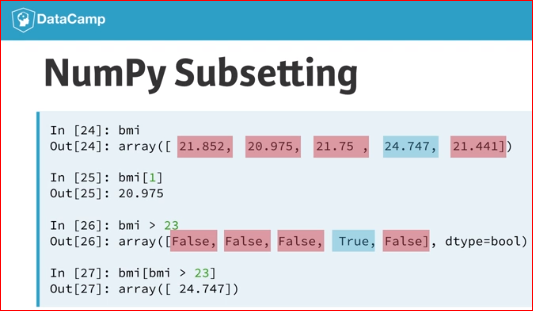


# NumPy



# 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?

# Create list baseball

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

# Import the numpy package as np

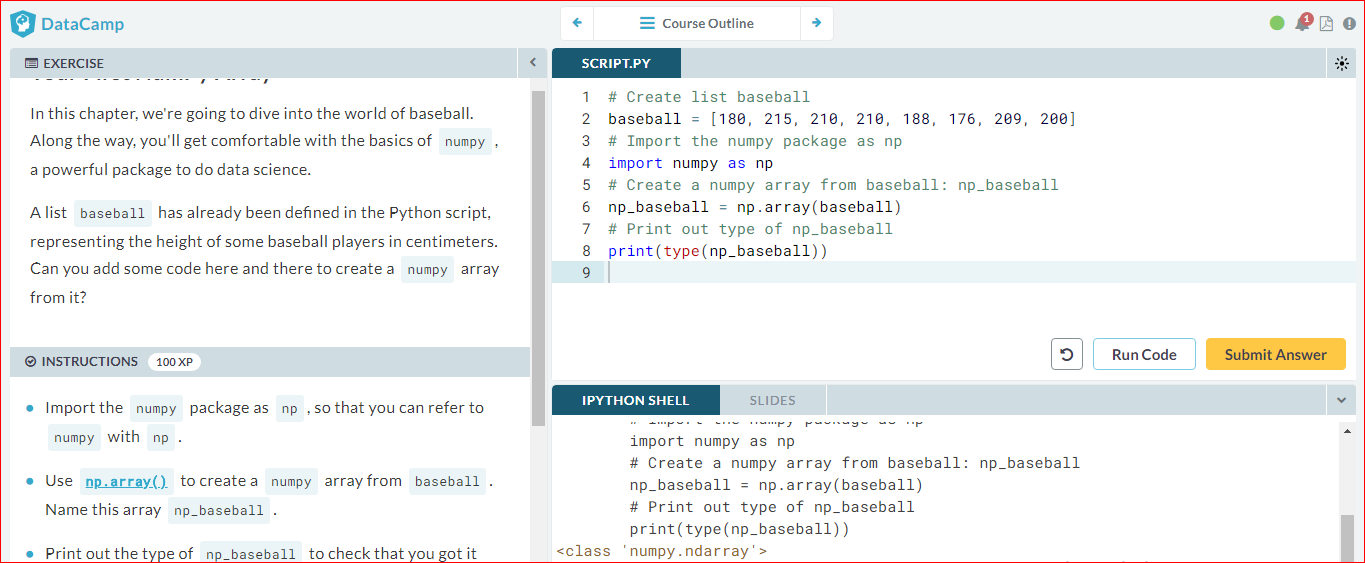
import numpy as np

# Create a numpy array from baseball: np\_baseball

np\_baseball = np.array(baseball)

# Print out type of np\_baseball

print(type(np\_baseball))



# 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. The height is expressed in inches. Can you make a numpy array out of it and convert the units to meters?

height 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)).

# height is available as a regular list

height = [1.73, 1.68, 1.71, 1.89, 1.79]

# Import numpy

import numpy as np

# Create a numpy array from height: np\_height

np\_height = np.array(height)

# Print out np\_height

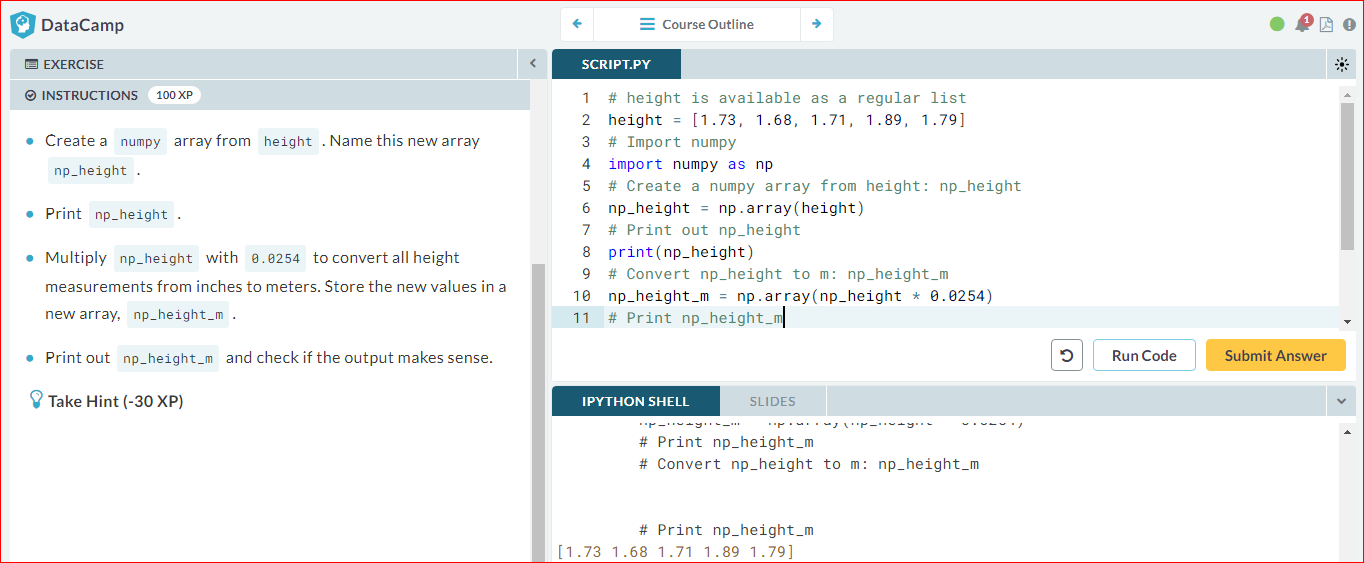
print(np\_height)

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

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

# Print np\_height\_m

print(np\_height\_m)



# Baseball player's BMI

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

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

# height and weight are available as a regular lists

# Import numpy

import numpy as np

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

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

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

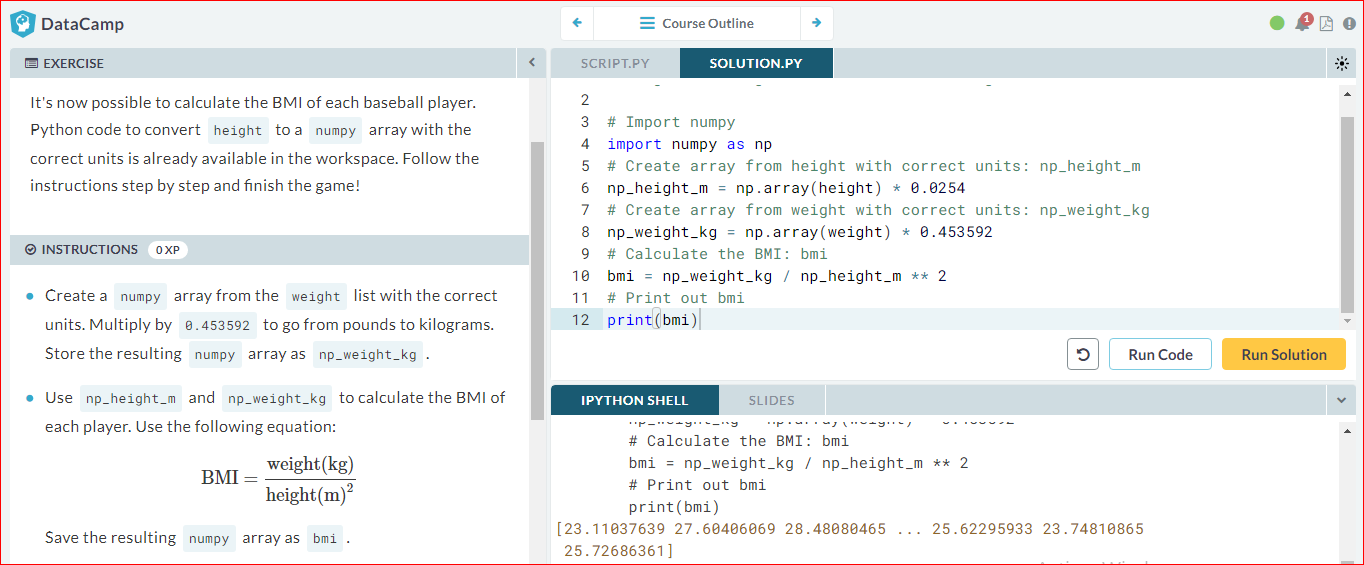
np\_weight\_kg = np.array(weight) \* 0.453592

# Calculate the BMI: bmi

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

# Print out bmi

print(bmi)



# 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!

# 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) \* 0.0254

np\_weight\_kg = np.array(weight) \* 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])



# Subsetting NumPy Arrays

You've seen it with your own eyes: Python lists and numpyarrays 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.

# height and weight are available as a regular lists

# Import numpy

import numpy as np

# Store weight and height lists as numpy arrays

np\_weight = np.array(weight)

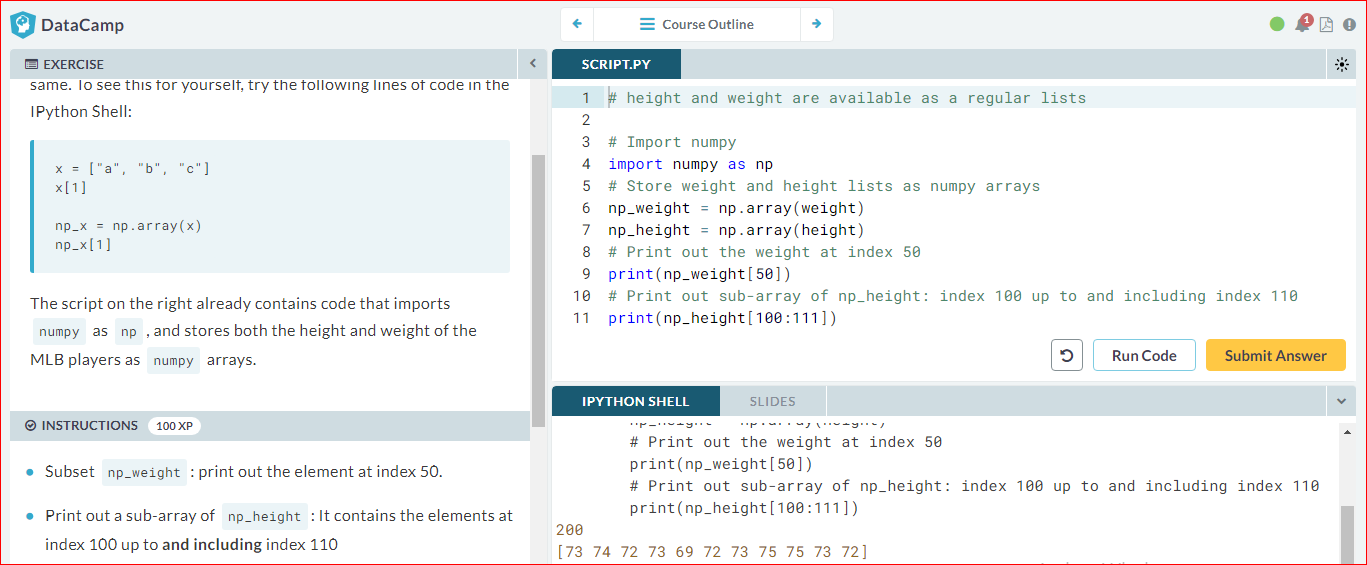
np\_height = np.array(height)

# Print out the weight at index 50

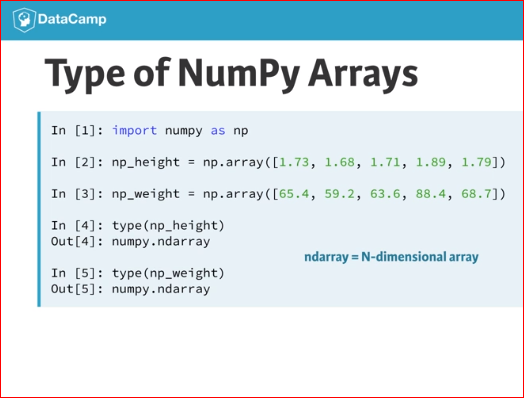
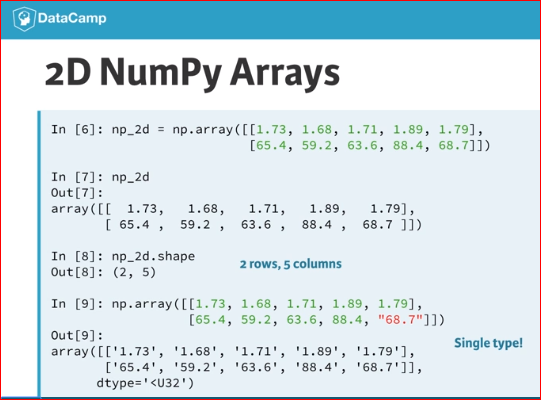
print(np\_weight[50])

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

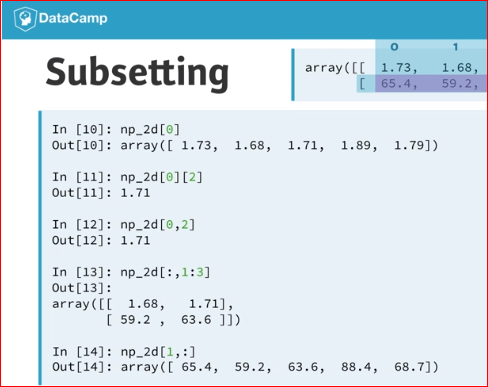
print(np\_height[100:111])



# 2D NumPy Arrays





# 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.

# 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

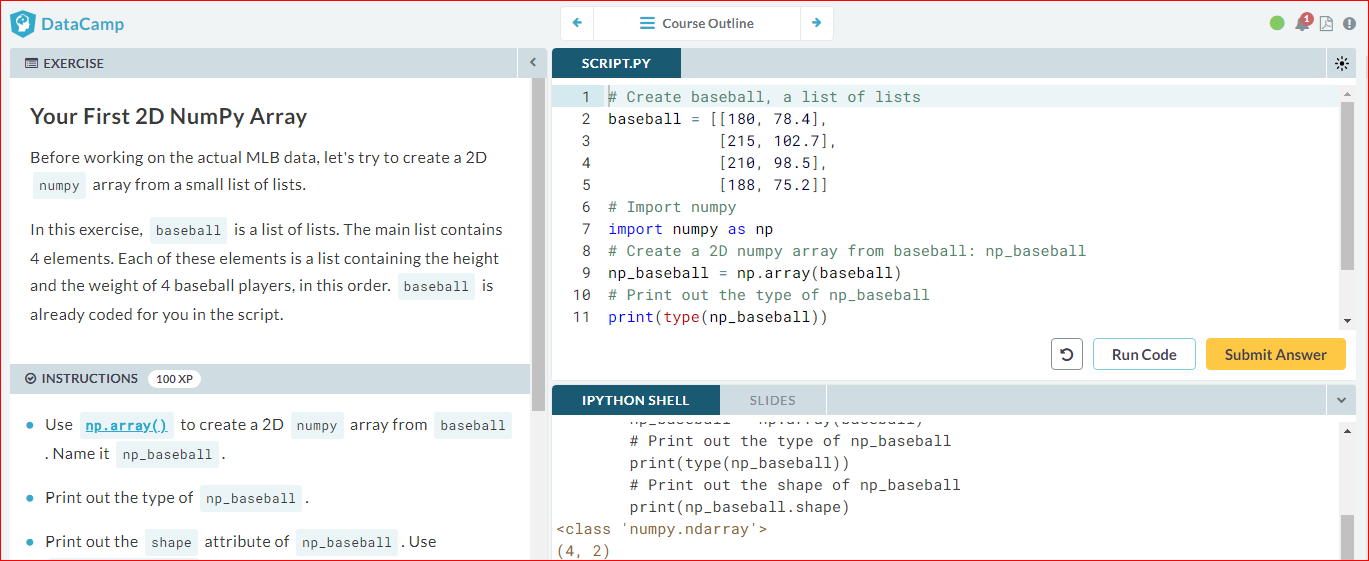
np\_baseball = np.array(baseball)

# Print out the type of np\_baseball

print(type(np\_baseball))

# Print out the shape of np\_baseball

print(np\_baseball.shape)



# 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 numpyarray. 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?

# 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\_baseball.shape)



# 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!

# 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,:])

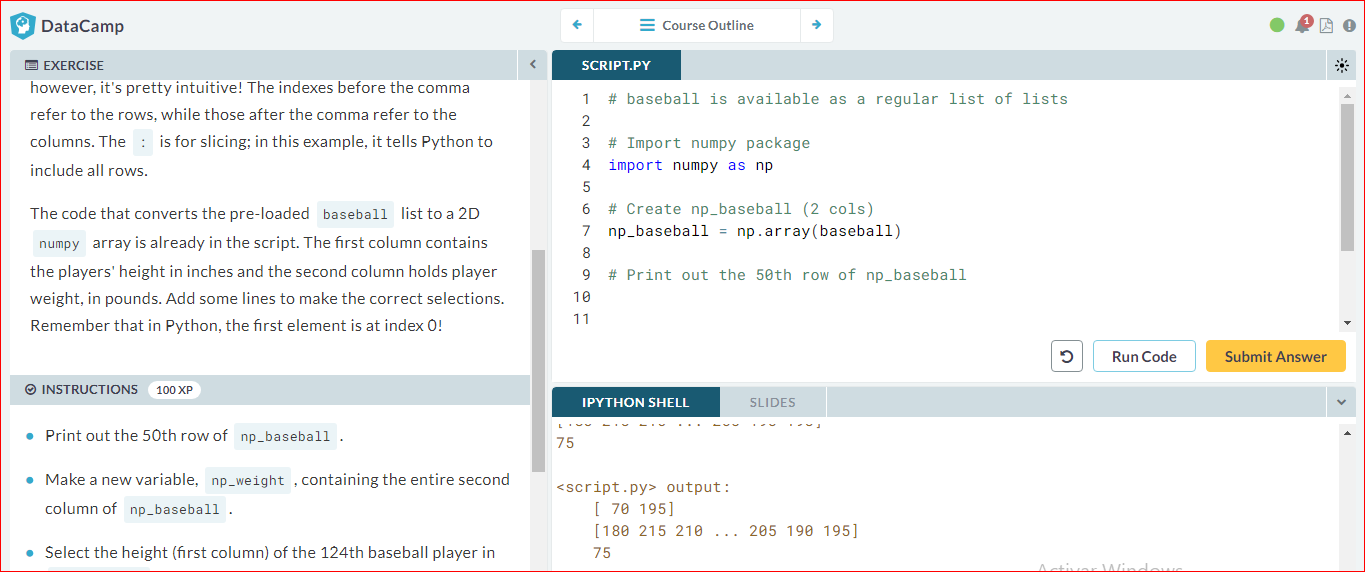
# Select the entire second column of np\_baseball: np\_weight

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

print(np\_weight)

# Print out height of 124th player

print(np\_baseball[123,0])



# 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, weight and age.

# 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

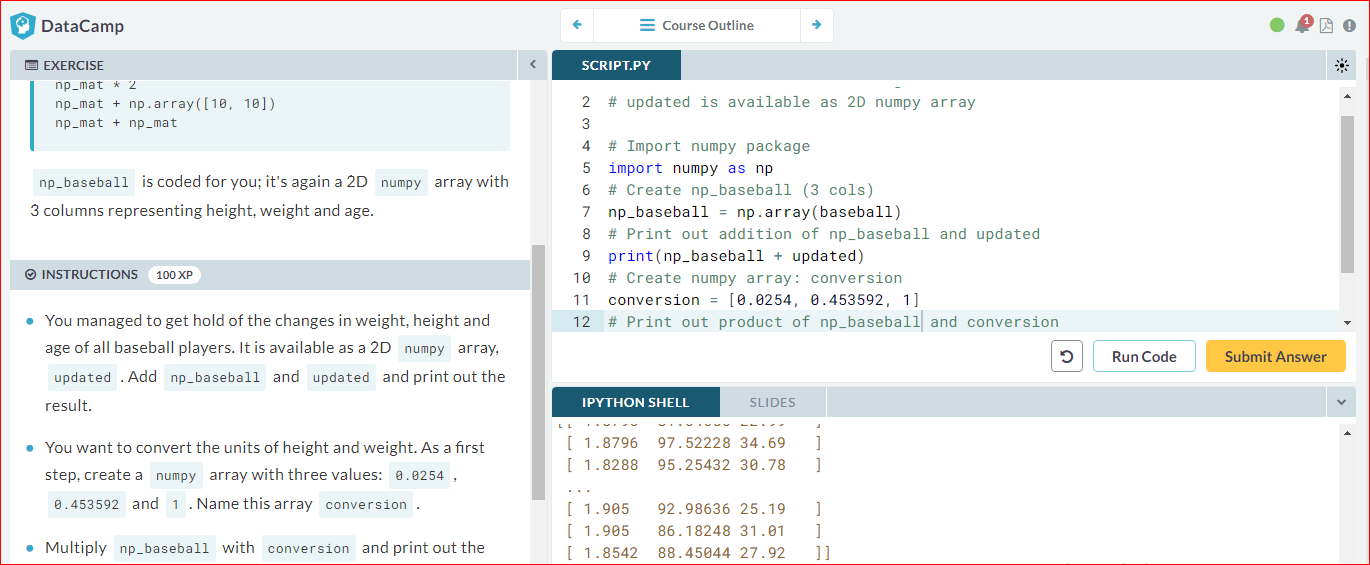
conversion = [0.0254, 0.453592, 1]

# Print out product of np\_baseball and conversion

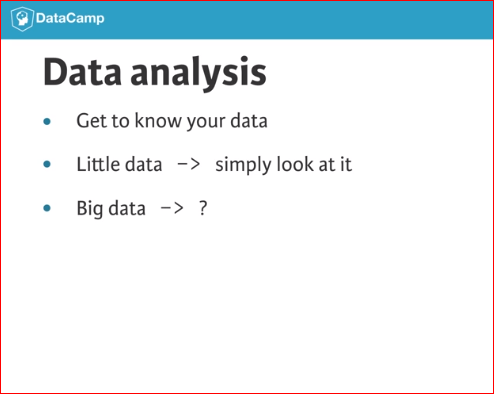
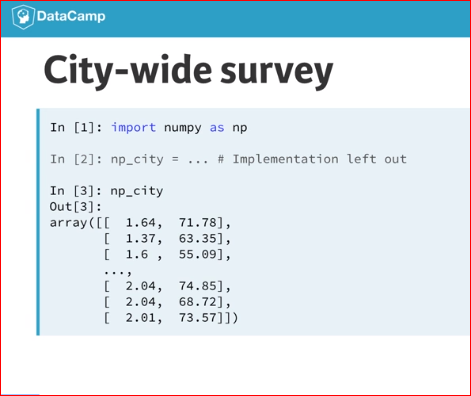
product = np\_baseball \* conversion

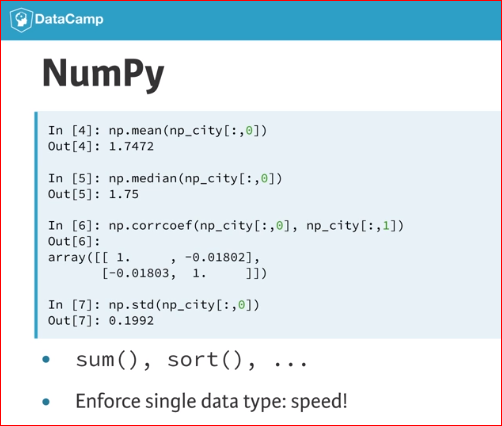
# Print out product of np\_baseball and conversion

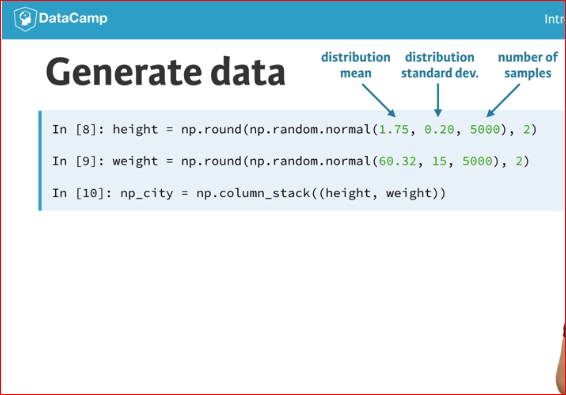
print(product)



# NumPy: Basic Statistics





# 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 numpyarrays:

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*.

# np\_baseball is available

# Import numpy

import numpy as np

# Create np\_height from np\_baseball

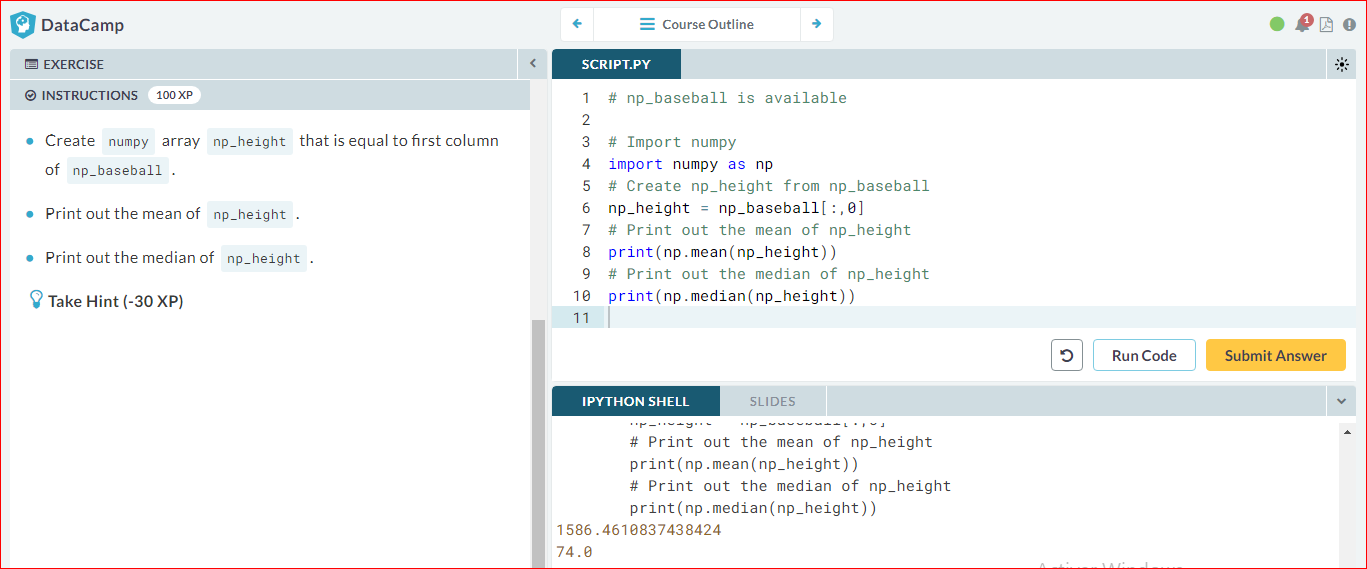
np\_height = np\_baseball[:,0]

# Print out the mean of np\_height

print(np.mean(np\_height))

# Print out the median of np\_height

print(np.median(np\_height))



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?

# 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'

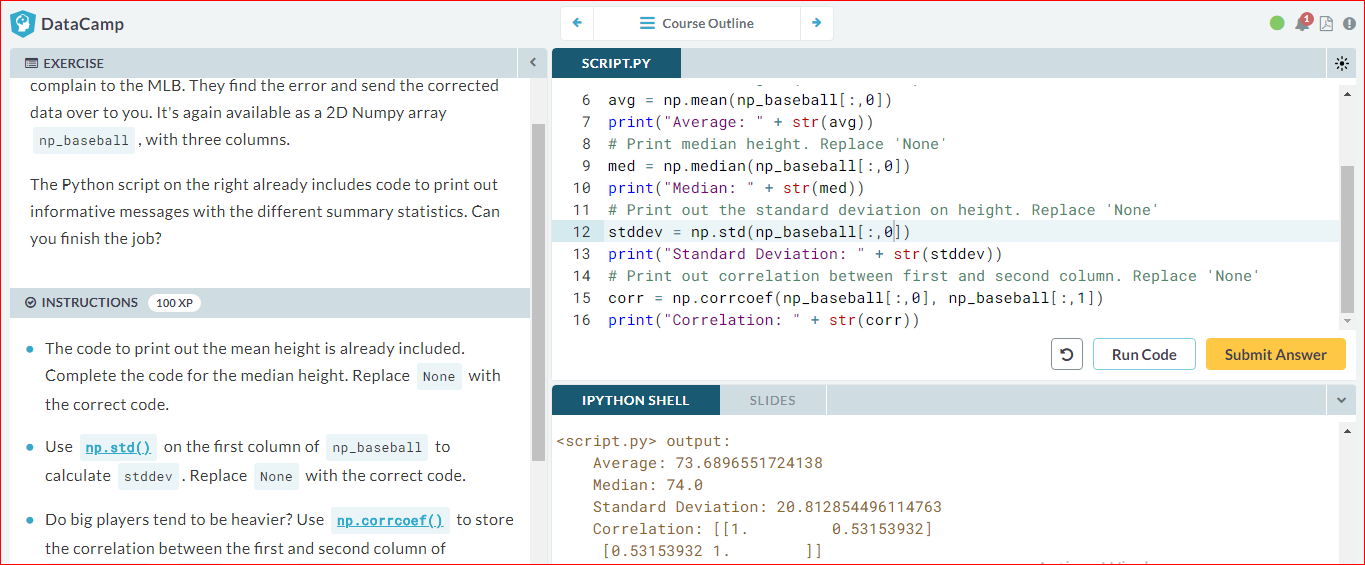
stddev = np.std(np\_baseball[:,0])

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))



# 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.

# 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

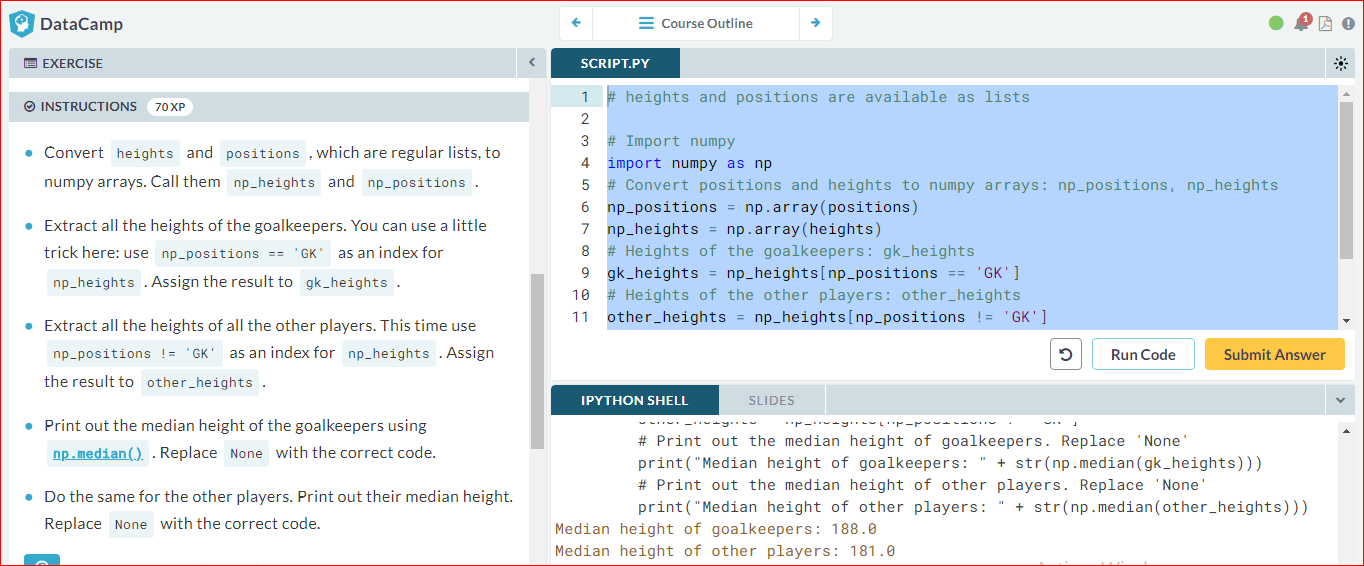
other\_heights = np\_heights[np\_positions != 'GK']

# 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)))



# NumPy Side Effects

As Filip explained before, numpy is great for doing vector arithmetic. If you compare its functionality with regular Python lists, however, some things have changed.

First of all, numpy arrays cannot contain elements with different types. If you try to build such a list, some of the elements' types are changed to end up with a homogeneous list. This is known as *type coercion*.

Second, the typical arithmetic operators, such as +, -, \* and / have a different meaning for regular Python lists and numpy arrays.

Have a look at this line of code:

np.array([True, 1, 2]) + np.array([3, 4, False])

Can you tell which code chunk builds the exact same Python object? The numpypackage is already imported as np, so you can start experimenting in the IPython Shell straight away!

