**Exercise**

**Exercise**

**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.

##### Exercise

##### Exercise

# Any comments?

Something that Hugo 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 in the editor, # Division; it is completely ignored during execution.

##### Instructions

**100 XP**

Above the print(7 + 10), add the comment

# Addition

**Exercise**

**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 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=110100×1.1=110 dollars, and after two years it's 100×1.1×1.1=121100×1.1×1.1=121. Add code to calculate how much money you end up with after 7 years, and print the result.

**Exercise**

**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.

**Exercise**

**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.

**Exercise**

**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.

**Exercise**

**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 to discover their type. Which of the following options is correct?

**Instructions**

**50 XP**

**Possible Answers**

* 

a is of type int, b is of type str, c is of type bool

* 

a is of type float, b is of type bool, c is of type str

* 

a is of type float, b is of type str, c is of type bool

* 

a is of type int, b is of type bool, c is of type str

**Exercise**

**Operations with other types**

Hugo 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?

**Exercise**

**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 integer savings and float 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 integers/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 integer 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.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Hit *Run Code* to run the code. Try to understand the error message.
* Fix the code such that the printout runs without errors; use the function **[str()](https://docs.python.org/3/library/functions.html" \l "func-str" \t "_blank)** to convert the variables to strings.
* Convert the variable pi\_string to a float and store this float as a new variable, pi\_float.

**Exercise**

**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!

**Instructions**

**50 XP**

**Possible Answers**

* 

"I can add integers, like " + str(5) + " to strings."

* 

"I said " + ("Hey " \* 2) + "Hey!"

* 

"The correct answer to this multiple choice exercise is answer number " + 2

* 

True + False

**Exercise**

**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()**](https://docs.python.org/3/library/functions.html#print) function.

**Exercise**

**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 in the editor 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 bath is a variable that represents the float 9.50 you specified earlier.

**Instructions**

**100 XP**

* Finish the code that creates the areas list. Build the list so that the list first contains the name of each room as a string and then its area. In other words, add the strings "hallway", "kitchen" and "bedroom" at the appropriate locations.
* Print areas again; is the printout more informative this time?

**Exercise**

**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]

**Instructions**

**50 XP**

**Possible Answers**

* 

A, B and C

* 

B

* 

B and C

* 

C

**Exercise**

**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 in the editor 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.

**Instructions**

**100 XP**

* Finish the list of lists so that it also contains the bedroom and bathroom data. Make sure you enter these in order!
* Print out house; does this way of structuring your data make more sense?
* Print out the type of house. Are you still dealing with a list?

**Exercise**

**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.

**Exercise**

**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.

**Exercise**

**Exercise**

**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()**](https://docs.python.org/3/library/functions.html#print).

**Course Outline**

**Daily XP** **100**

**Exercise**

**Exercise**

**Slicing and dicing (2)**

In the video, Hugo 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.

**Exercise**

**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.

**Instructions**

**50 XP**

**Possible Answers**

* 

A float: the kitchen area

* 

A string: "kitchen"

* 

A float: the bathroom area

* 

A string: "bathroom"

**Exercise**

**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.

**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**

**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, Hugo 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. Submit the answer to check this.

**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.

**Exercise**

**Exercise**

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

**Instructions**

**50 XP**

**Possible Answers**

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes exactly two arguments: real and [, imag].

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. Both these arguments are required.

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. real is a required argument, imag is an optional argument.

* 

[**complex()**](https://docs.python.org/3/library/functions.html#complex) takes two arguments: real and imag. If you don't specify imag, it is set to 1 by Python.

**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 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 reverse without changing anything about key, you can use =:

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

Two lists have been created for you in the editor. 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**

**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.

**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"!

**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.

**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.

**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 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.

**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.

**Exercise**

**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.

**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 r is 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.

**Exercise**

**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?

**Instructions**

**50 XP**

**Possible Answers**

* 

import scipy

* 

import scipy.linalg

* 

from scipy.linalg import my\_inv

* 

from scipy.linalg import inv as my\_inv

**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.

**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" \t "_blank)**).

**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.