Introduction to Python

January 5, 2020

1 Introduction to Python

Course Description

Python is a general-purpose programming language that is becoming ever more popular for data science. Companies worldwide are using Python to harvest insights from their data and gain a competitive edge. Unlike other Python tutorials, this course focuses on Python specifically for data science. In our Introduction to Python course, you'll learn about powerful ways to store and manipulate data, and helpful data science tools to begin conducting your own analyses. Start DataCamp's online Python curriculum now.

1.1 1. Python Basics

An introduction to the basic concepts of Python. Learn how to use Python interactively and by using a script. Create your first variables and acquaint yourself with Python's basic data types.

1.1.1 1.1 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

- 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.

[1]:
```
# To print without truncation :
#import sys
#import numpy
#numpy.set_printoptions(threshold=sys.maxsize)

```

```
#import warnings
#warnings.filterwarnings('ignore')

# Example, do not modify!
print(5 / 8)

# Put code below here
7 / 8
print(7 + 10)
```

```
0.625
17
```

1.1.2 1.2 When to use Python?

Python is a pretty versatile language. For which applications can you use Python?

Instructions

- Possible Answers
- You want to do some quick calculations.
- For your new business, you want to develop a database-driven website.
- Your boss asks you to clean and analyze the results of the latest satisfaction survey.

Correct! Python is an extremely versatile language.

1.1.3 1.3 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 on the right, `# Division`; it is completely ignored during execution.

Instructions

- Above the print(7 + 10), add the comment # Addition.

[2]:
```
# Just testing division
print(5 / 8)

# Addition
print(7 + 10)
```

```
0.625
17
```

1.1.4 1.4 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

- 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 to calculate how much money you end up with after 7 years, and print the result.

[3]:
```
# Addition and subtraction
print(5 + 5)
print(5 - 5)

# Multiplication and division
print(3 * 5)
print(10 / 2)

# Exponentiation
print(4 ** 2)

# Modulo
print(18 % 7)

# How much is your $100 worth after 7 years?
print(100 * 1.1 ** 7)
```

```
10
0
15
5.0
16
4
194.87171000000012
```

1.1.5 1.5 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

- Create a variable `savings` with the value 100.
- Check out this variable by typing `print(savings)` in the script.

[4]:
```
# Create a variable savings
savings = 100

# Print out savings
print(savings)
```

```
100
```

Great! Let's try to do some calculations with this variable now!

1.1.6 1.6 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

- 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`.

[5]:
```
# Create a variable savings
savings = 100

# Create a variable growth_multiplier
growth_multiplier = 1.1

# Calculate result
result = savings * growth_multiplier ** 7

# Print out result
print(result)
```

```
194.87171000000012
```

1.1.7 1.7 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

- Create a new string, `desc`, with the value `"compound interest"`.
- Create a new boolean, `profitable`, with the value `True`

[6]:
```
# Create a variable desc
desc = "compound interest"

# Create a variable profitable
profitable = True
```

1.1.8 1.8 Guess the type

To find out the type of a value or a variable that refers to that value, you can use the 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?

[7]:
```
a = 194.87171000000012; b = 'True'; c = False
print(type(a)); print(type(b)); print(type(c))
```

```
<class 'float'>
<class 'str'>
<class 'bool'>
```

Instructions

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`

1.1.9 1.8 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

- 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?

[8]:
```
# Several variables to experiment with
savings = 100
growth_multiplier = 1.1
desc = "compound interest"

# Assign product of factor and savings to year1
year1 = growth_multiplier * savings

# Print the type of year1
print(type(year1))

# Assign sum of desc and desc to doubledesc
doubledesc = desc + desc

# Print out doubledesc
print(doubledesc)
```

```
<class 'float'>
compound interestcompound interest
```

Nice. Notice how `desc + desc` *causes* `"compound interest"` *and* `"compound interest"` *to be pasted together.*

1.1.10 1.9 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(), to convert a value into a string. `str(savings)`, for example, will convert the integer `savings` to a string.

Similar functions such as int(), float() and bool() will help you convert Python values into any type.

Instructions

- 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() to convert the variables to strings.
- Convert the variable `pi_string` to a float and store this float as a new variable, `pi_float`.

[9]:
```
# 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)
```

```
I started with $100 and now have $194.87171000000012. Awesome!
```

Great! You have a profit of around $95; that's pretty awesome indeed!

1.1.11 1.10 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

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

Correct! Because you're not converting 2 to a string with str(), this will give an error.

1.2 2. Python Lists

Learn to store, access, and manipulate data in lists: the first step toward efficiently working with huge amounts of data.

1.2.1 2.1 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

- 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.

[10]:
```
# area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50

# Create list areas
areas = [hall, kit, liv, bed, bath]

# Print areas
print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5]
```

Nice! A list is way better here, isn't it?

1.2.2 2.2 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 `bath` is a variable that represents the float `9.50` you specified earlier.

Instructions

- Finish the line of 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?

[11]:
```
# 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)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0, 'bedroom', 10.75,
'bathroom', 9.5]
```

Nice! This list contains both strings and floats, but that's not a problem for Python!

1.2.3 2.3 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

Possible Answers - **A, B and C** - B - B and C - C

Correct! As funny as they may look, all these commands are valid ways to build a Python list.

1.2.4 2.4 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.

Instructions

- 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?

[12]:
```
# 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))
```

```
[['hallway', 11.25], ['kitchen', 18.0], ['living room', 20.0], ['bedroom',
10.75], ['bathroom', 9.5]]
<class 'list'>
```

Great! Get ready to learn about list subsetting!

1.2.5 2.5 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

- Print out the second element from the `areas` list (it has the value `11.25`).
- Subset and print out the last element of `area`, 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.

[ ]:
```
# 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])
```

1.2.6 2.6 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**

- 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`.

[13]:
```
# 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)
```

```
28.75
```

1.2.7 2.7 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

- 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().

[14]:
```
# 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[6:10]

# Print out downstairs and upstairs
print(downstairs)
print(upstairs)
```

```
['hallway', 11.25, 'kitchen', 18.0, 'living room', 20.0]
['bedroom', 10.75, 'bathroom', 9.5]
```

1.2.8 2.8 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**

- 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.

[15]:
```
# 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:]
```

1.2.9 2.9 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

Possible Answers - A float: the kitchen area - A string: "kitchen" - **A float: the bathroom area** - A string: "bathroom"

1.2.10 2.10 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

- 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"`.

[18]:
```
# 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"
```

Sweet! As the code sample showed, you can also slice a list and replace it with another list to update multiple elements in a single command.

1.2.11 2.11 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

- 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`.

1.2.12 2.12 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

2 Separate lines

command1 command2 ~~~ Which of the code chunks will do the job for us?

Instructions

Possible Answers - del(areas[10]); del(areas[11]) - del(areas[10:11]) - **del(areas[-4:-2])** - del(areas[-3]); del(areas[-4])

[4]:
```
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])
areas
```

[4]:
```
['hallway',
 11.25,
 'kitchen',
 18.0,
 'chill zone',
 20.0,
 'bedroom',
 10.75,
 'bathroom',
 10.5,
 'garage',
 15.45]
```

Correct! You'll learn about easier ways to remove specific elements from Python lists later on.

2.0.1 2.13 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() or by using `[:]`.

Instructions

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.

[5]:
```
# Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Create areas_copy
areas_copy = areas[:]

# Change areas_copy
areas_copy[0] = 5.0

# Print areas
print(areas)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5]
```

Nice! The difference between explicit and reference-based copies is subtle, but can be really important. Try to keep in mind how a list is stored in the computer's memory.

2.1 3. Functions and Packages

You'll learn how to use functions, methods, and packages to efficiently leverage the code that brilliant Python developers have written. The goal is to reduce the amount of code you need to solve challenging problems!

2.1.1 3.1 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() and `type()`. You've also used the functions `str()`, `int()`, `bool()` and `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

- Use `print()` in combination with `type()` to print out the type of `var1`.
- Use `len()` to get the length of the list `var1`. Wrap it in a `print()` call to directly print it out.
- Use `int()` to convert `var2` to an integer. Store the output as `out2`.

[6]:
```
# 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)
```

```
<class 'list'>
4
```

Great job! The len() function is extremely useful; it also works on strings to count the number of characters!

2.1.2 3.2 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(). In IPython specifically, you can also use ? before the function name.

To get help on the 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(). Which of the following statements is true?

Instructions

Possible Answers - complex() takes exactly two arguments: `real` and `[, imag]`. - complex() takes two arguments: `real` and `imag`. Both these arguments are required. - **complex() takes two arguments: `real` and `imag`. `real` is a required argument, `imag` is an optional argument.** - complex() takes two arguments: `real` and `imag`. If you don't specify `imag`, it is set to 1 by Python.

2.1.3 3.3 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() by typing `help(sorted)` in the IPython Shell.

You'll see that 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() 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 as being any collection of objects, e.g. a List.

Instructions

- Use `+` to merge the contents of `first` and `second` into a new list: `full`.
- Call 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`.

[7]:
```
# 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)
```

```
[20.0, 18.0, 11.25, 10.75, 9.5]
```

2.1.4 3.4 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

- Use the 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() on place and passing the letter 'o' as an input to the method. We're talking about the variable place, not the word "place"!

[8]:
```
# 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"))
```

```
poolhouse
POOLHOUSE
3
```

Nice! Notice from the printouts that the upper() method does not change the object it is called on. This will be different for lists in the next exercise!

2.1.5 3.5 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(), to get the index of the first element of a list that matches its input and
- 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

- Use the index() method to get the index of the element in `areas` that is equal to `20.0`. Print out this index.
- Call count() on `areas` to find out how many times `9.50` appears in the list. Again, simply print out this number.

[9]:
```
# 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(9.5))
```

```
2
1
```

Nice! These were examples of list methods that did not change the list they were called on.

2.1.6 3.6 List Methods (2)

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

append(), that adds an element to the list it is called on, remove(), that removes the first element of a list that matches the input, and reverse(), 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

- Use append() 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() method to reverse the order of the elements in areas.
- Print out areas once more.

[10]:
```
# 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)
```

```
[11.25, 18.0, 20.0, 10.75, 9.5, 24.5, 15.45]
[15.45, 24.5, 9.5, 10.75, 20.0, 18.0, 11.25]
```

2.1.7 3.7 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, C, and area, A, of a circle. When the radius of the circle is `r`, you can calculate C and A as:

C=2 r
A= r2

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() functions create some nice printouts.

Instructions

- 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`.

[11]:
```
# Definition of radius
r = 0.43

# Import the math package
import math

# Calculate C
C = 2 * math.pi * r
```

```
# Calculate A
A = math.pi * r ** 2

# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

```
Circumference: 2.701769682087222
Area: 0.5808804816487527
```

2.1.8 3.8 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

- 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() function, which you just imported.
- Print out `dist`.

[ ]:
```
# 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.
dist = r * radians(12)

# Print out dist
print(dist)
```

2.1.9 3.9 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(), 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

Possible Answers - `import scipy` - `import scipy.linalg` - `from scipy.linalg import my_inv` - `from scipy.linalg import inv as my_inv`

Correct! The as word allows you to create a local name for the function you're importing: inv() is now available as my_inv().

2.2 4 NumPy

NumPy is a fundamental Python package to efficiently practice data science. Learn to work with powerful tools in the NumPy array, and get started with data exploration.

2.2.1 4.1 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

- Import the `numpy` package as `np`, so that you can refer to `numpy` with `np`.
- Use np.array() 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.

[13]:
```
# 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))
```

```
<class 'numpy.ndarray'>
```

2.2.2 4.2 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).

Instructions

- 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.

[22]:
```
import pandas as pd
df = pd.read_csv('./Data/baseball.csv')
height_in = df.Height.to_list()
weight_lb = df.Weight.to_list()
```

[23]:
```
# height_in 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
print(np_height_in)

# Convert np_height to m: np_height_m
np_height_m = np_height_in * 0.0254

# Print np_height_m
print(np_height_m)
```

```
[74 74 72 … 75 75 73]
[1.8796 1.8796 1.8288 … 1.905  1.905  1.8542]
```

2.2.3 4.3 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_lb`. `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

- 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) * 2)
- Save the resulting `numpy` array as `bmi`.
- Print out `bmi`.

[26]:
```
# height_in and weight_lb are available as a regular lists

# Import numpy
import numpy as np

# Create array from height_in with correct units: np_height_m
np_height_m = np.array(height_in) * 0.0254

# Create array from weight_lb with correct 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)
```

```
[23.11037639 27.60406069 28.48080465 … 25.62295933 23.74810865
 25.72686361]
```

2.2.4 4.4 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

- 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 `bmi` array.

[27]:
```
# height_in and weight_lb 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])
```

```
[False False False … False False False]
[20.54255679 20.54255679 20.69282047 20.69282047 20.34343189 20.34343189
 20.69282047 20.15883472 19.4984471  20.69282047 20.9205219 ]
```

2.2.5 4.5 NumPy Side Effects

As Hugo 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 numpy package is already imported as np, so you can start experimenting in the IPython Shell straight away!

Instructions

Possible Answers - `np.array([True, 1, 2, 3, 4, False])` - **`np.array([4, 3, 0]) + np.array([0, 2, 2])`** - `np.array([1, 1, 2]) + np.array([3, 4, -1])` - `np.array([0, 1, 2, 3, 4, 5])`

Great job! True is converted to 1, False is converted to 0.

2.2.6 4.6 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

- Subset `p_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.

[28]:
```
# 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_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])
```

```
200
[73 74 72 73 69 72 73 75 75 73 72]
```

2.2.7 4.7 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

- Use np.array() 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`.

[29]:
```
# 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)
```

```
<class 'numpy.ndarray'>
(4, 2)
```

2.2.8 4.8 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

- Use p.array() to create a 2D `numpy` array from `baseball`. Name it `np_baseball`.
- Print out the `shape` attribute of `np_baseball`.

[42]:
```
# baseball is available as a regular list of lists
df_baseball= df[['Height','Weight']]
baseball = df_baseball.values.tolist()

# 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)
```

```
(1015, 2)
```

2.2.9 4.9 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]]

3 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

- 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.

[44]:
```
# 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])
```

```
[ 70 195]
75
```

3.0.1 4.10 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

- 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.

[51]:
```
# baseball is available as a regular list of lists
# updated is available as 2D numpy array
baseball = df[['Height','Weight','Age']].values
updated = baseball * 0.005
updated[:,2] = 1

# 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)
```

```
[[ 74.37  180.9    23.99 ]
 [ 74.37  216.075  35.69 ]
 [ 72.36  211.05   31.78 ]
 …
 [ 75.375 206.025  26.19 ]
 [ 75.375 190.95   32.01 ]
 [ 73.365 195.975  28.92 ]]
[[ 1.8796  81.64656 22.99   ]
 [ 1.8796  97.52228 34.69   ]
 [ 1.8288  95.25432 30.78   ]
 …
 [ 1.905   92.98636 25.19   ]
 [ 1.905   86.18248 31.01   ]
 [ 1.8542  88.45044 27.92   ]]
```

3.0.2 4.11 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

- 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`.

[53]:
```
# 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))
```

```
73.6896551724138
74.0
```

3.0.3 4.12 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

- 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()` 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()` to store the correlation between the first and second column of `np_baseball` in `corr`. Replace `None` with the correct code.

[54]:
```
# 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))
```

```
Average: 73.6896551724138
Median: 74.0
Standard Deviation: 2.312791881046546
Correlation: [[1.         0.53153932]
 [0.53153932 1.        ]]
```

3.0.4 4.13 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

- 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(). Replace None with the correct code.

- Do the same for the other players. Print out their median height. Replace None with the correct code.

[67]:
```
# heights and positions are available as lists
df_fifa = pd.read_csv('./Data/fifa.csv')
heights = df_fifa.height.to_list()
positions = df_fifa.position.to_list()
# 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)))
```

```
Median height of goalkeepers: 188.0
Median height of other players: 181.0
```