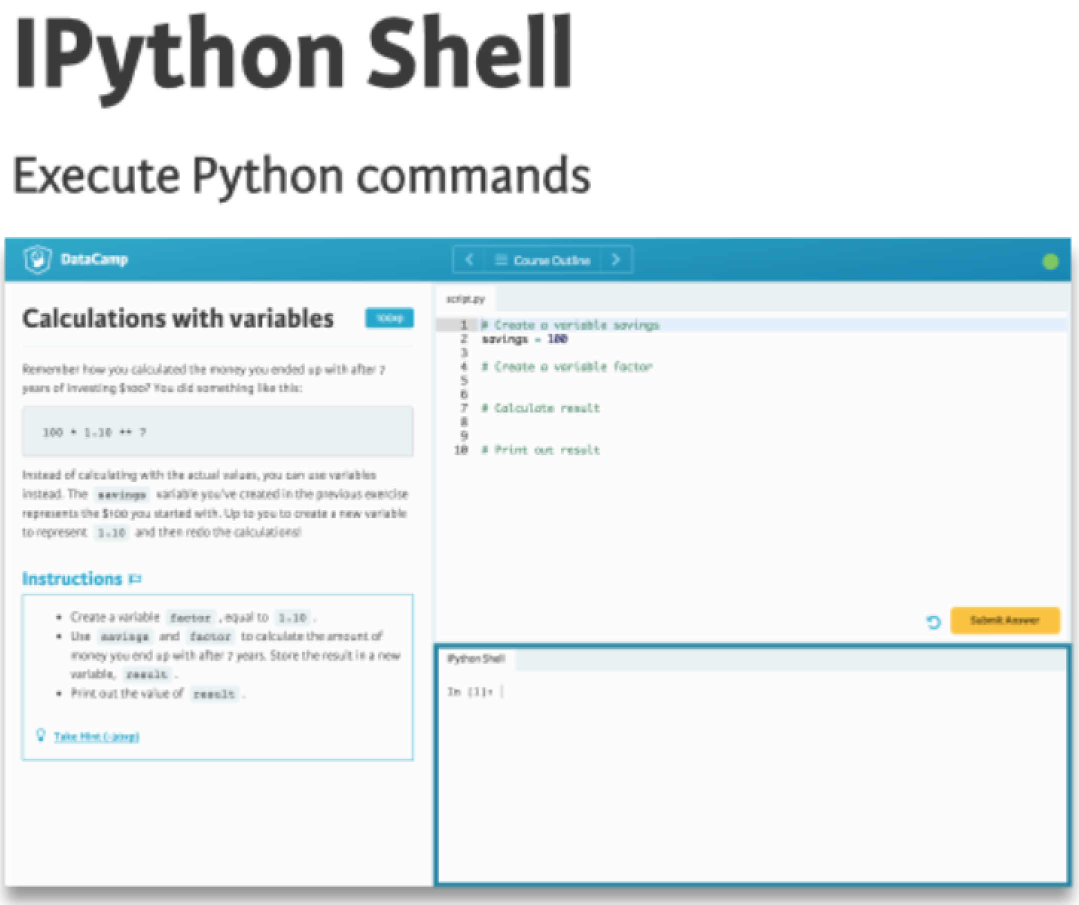
# Introduction to Python for Data Science

**“Learning by Doing”**

## Python

- Guido Van Rossum
- General Purpose: build anything
- Open Source! Free!
- Python Packages, also for Data Science
- Many applications and fields
- Version 3.x - [https://www.python.org/downloads/](https://www.python.org/downloads/)

---



## IPython Shell

Execute Python commands

### 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. Up to you to create a new variable to represent `1.10` and then redo the calculations!

**Instructions**

- Create a variable `factor`, equal to `1.10`.
- Use `savings` and `factor` to calculate the amount of money you end up with after 7 years. Store the result in a new variable, `result`.
- Print out the value of `result`.

Take Hint (-30xp)

script.py

```
# Create a variable savings
savings = 100

# Create a variable factor


# Calculate result


# Print out result
```

IPython Shell

```
In [1]:
```

## Python Script

● Text Files - .py

● List of Python Commands

● Similar to typing in IPython Shell

Example Code:

```
# Example, do not modify!

print(5 / 8) → 0.625


# Put code below here

print(7 + 10) → 17
```

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

- 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.

You can all of the above tasks using Python.

## Comments:

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.

Example Code:

```
# Division
print(5 / 8)

# Addition
print(7 + 10)
```

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.

Example Code :

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

```
# Addition, subtraction

print(5 + 5) → 10

print(5 - 5) → 0


# Multiplication, division, modulo, and exponentiation

print(3 * 5) → 15

print(10 / 2) → 5

print(18 % 7) → 4

print(4 ** 2) → 16


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

print(100 * 1.1 ** 7 ) → 194.87171
```

# Variables and Types

## Variable

- Specific, case-sensitive name
- Call up value through variable name

**Example Code :**

```
In [1]: height = 1.79

In [2]: weight = 68.7

In [3]: height

Out[3]: 1.79
```

**Example : Calculate BMI :**

BMI = weight / height$^2$

```
In [1]: height = 1.79

In [2]: weight = 68.7

In [3]: height

Out[3]: 1.79

In [4]: 68.7 / 1.79 ** 2

Out[4]: 21.4413

In [5]: weight / height ** 2

Out[5]: 21.4413

In [6]: bmi = weight / height ** 2

In [7]: bmi

Out[7]: 21.4413
```

## Python Types

```
In [8]: type(bmi)

Out[8]: float

In [9]: day_of_week = 5

In [10]: type(day_of_week)

Out[10]: int

In [11]: x = "body mass index"

In [12]: y = 'this works too'

In [13]: type(y)

Out[13]: str

In [14]: z = True

In [15]: type(z)

Out[15]: bool

In [16]: 2 + 3

Out[16]: 5

                                    Different type = different behavior!

In [17]: 'ab' + 'cd'

Out[17]: 'abcd'  # here, it is string concatenation
```

## 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!

**Example Code**:

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

```
# Create a variable savings
savings = 100

# Print out savings
print(savings) → 100
```

## 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.

**Example Code**:

```
# Create a variable savings
savings = 100

# Create a variable factor
growth_multiplier = 1.10

# Calculate result
result = savings * growth_multiplier ** 7

# Print out result
print(result) → 194.871710
```

## 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!).

Example Code:

```
# 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()** 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)
```

**Note:** 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.

```
desc = "compound interest"

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

# Print out doubledesc
print(doubledesc) → compound interestcompound interest
```

## 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.

You need to do type conversion for it to work. See below :

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

**Example Code :**

```
# Definition of pi_string
pi_string = "3.1415926"

# Convert pi_string into float: pi_float
pi_float = float(pi_string)
```

**Some more Examples :**

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

"I said " + ("Hey " * 2) + "Hey!" → 'I said Hey Hey Hey!

True + False → 1
```

## Python Data Types

- float - real numbers
- int - integer numbers
- str - string, text
- bool - True, False

```
In [1]: height = 1.73

In [2]: tall = True
```

- Each variable represents single value

**Problem** :

- Data Science: many data points
- Height of entire family

```
In [3]: height1 = 1.73

In [4]: height2 = 1.68

In [5]: height3 = 1.71

In [6]: height4 = 1.89
```

- It is inconvenient to represent the list of values using independent variables.

**Solution** : Use Lists.

## Python List :    [a, b, c]

```
# 1-D lists
In [7]: [1.73, 1.68, 1.71, 1.89]
Out[7]: [1.73, 1.68, 1.71, 1.89]
In [8]: fam = [1.73, 1.68, 1.71, 1.89]
In [9]: fam
Out[9]: [1.73, 1.68, 1.71, 1.89]

In [10]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
In [11]: fam
Out[11]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

# 2-D lists
In [11]: fam2 = [["liz", 1.73],
                 ["emma", 1.68],
                 ["mom", 1.71],
                 ["dad", 1.89]]
In [12]: fam2
Out[12]: [['liz', 1.73], ['emma', 1.68],
               ['mom', 1.71], ['dad', 1.89]]
```

- Name a collection of values
- ** **Contain any type**
- Contain different types

List type :

```
In [13]: type(fam)

Out[13]: list
```

- Specific functionality
- Specific behavior

## 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 :

Example Code :

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.

```
# 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)
```

## 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.

### Example Code :

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.

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

# Print areas
print(areas)
```

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!

### Examples :

```
[1, 3, 4, 2]

[[1, 2, 3], [4, 5, 7]]

[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.

**Example Code :**

```
# 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 :

```
In [1]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]

In [2]: fam

Out[2]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

index:   0       1      2      3      4      5     6      7

                         "zero-based indexing"

In [3]: fam[3]
Out[3]: 1.68

In [4]: fam[6]
Out[4]: 'dad'
```

List Traversing from the backwards :

```
In [1]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]

In [2]: fam

Out[2]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

index:   0        1       2       3       4       5       6       7

         -8       -7      -6      -5      -4      -3      -2      -1


In [3]: fam[3]

Out[3]: 1.68

In [4]: fam[6]

Out[4]: 'dad'

In [5]: fam[-1]

Out[5]: 1.89

In [6]: fam[-2]

Out[6]: 'dad'
```

## List slicing :

```
In [7]: fam

Out[7]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

         0        1       2       3       4       5       6       7

In [8]: fam[3:5]

Out[8]: [1.68, 'mom']
```

**[ start : end ] –** end is not inclusive.

Inclusive : exclusive

Example Code :

```
In [8]: fam[3:5]

Out[8]: [1.68, 'mom']

In [9]: fam[1:4]

Out[9]: [1.73, 'emma', 1.68]

In [10]: fam[:4]

Out[10]: ['liz', 1.73, 'emma', 1.68]

In [11]: fam[5:]

Out[11]: [1.71, 'dad', 1.89]
```

## 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!
```

Example Code :

```
# 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])
```

**Example Code** :

```
# 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.

```
# Use slicing to create downstairs
downstairs = areas[:6] → contains the first 6 elements of areas.

# Use slicing to create upstairs
upstairs = areas[6:] →  contains the last 4 elements of areas

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

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[:]
```

## 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] → g
x[2][:2] → [‘g’, ‘h’]
```

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

## List Manipulation :

- Change list elements
- Add list elements
- Remove list elements

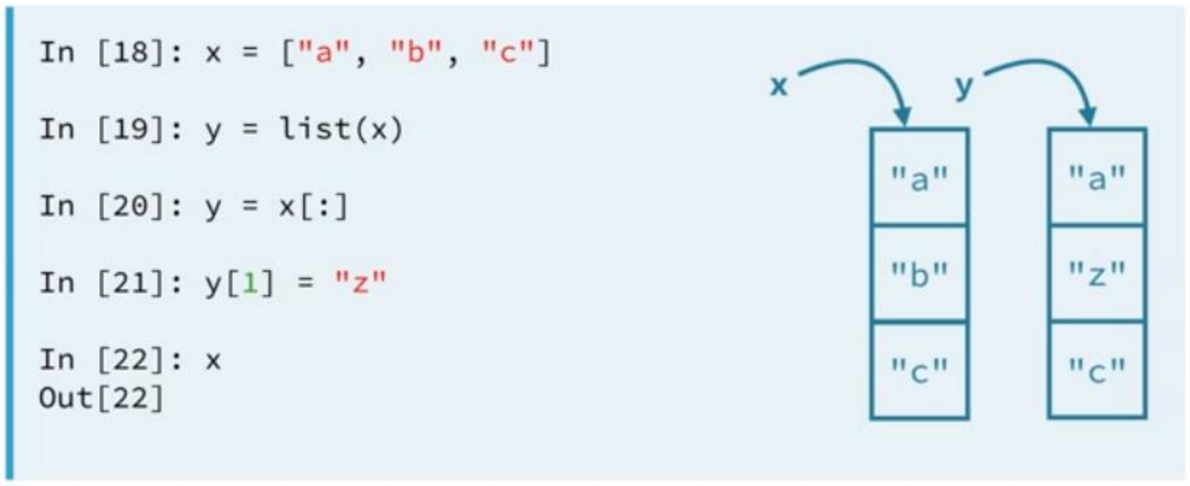
## Changing list elements :

```
In [1]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]

In [2]: fam

Out[2]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

In [3]: fam[7] = 1.86

In [4]: fam

Out[4]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.86]

In [5]: fam[0:2] = ["lisa", 1.74]

In [6]: fam

Out[6]: ['lisa', 1.74, 'emma', 1.68, 'mom', 1.71, 'dad', 1.86]
```

## Adding and removing elements :

```
In [7]: fam + ["me", 1.79]

Out[7]: ['lisa', 1.74,'emma', 1.68,

                         'mom', 1.71, 'dad', 1.86, 'me', 1.79]

In [8]: fam_ext = fam + ["me", 1.79]

In [9]: del(fam[2])

In [10]: fam

Out[10]: ['lisa', 1.74, 1.68, 'mom', 1.71, 'dad', 1.86]

In [11]: del(fam[2])

In [12]: fam

Out[12]: ['lisa', 1.74, 'mom', 1.71, 'dad', 1.86]
```

## Copying Lists :



```
In [18]: x = ["a", "b", "c"]

In [19]: y = list(x)

In [20]: y = x[:]

In [21]: y[1] = "z"

In [22]: x
Out[22]
```

- Use list()
- (or) use [:] notation

## 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"]
```

**Example Code :**

```
# 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[-1] = 10.50

# Change "living room" to "chill zone"
areas[4] = "chill zone"

print(areas)
['hallway', 11.25, 'kitchen', 18.0, 'chill zone', 20.0, 'bedroom',
10.75, 'bathroom', 10.5]
```

## 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"]
```

**Example Code :**

```
# 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]
```

## 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 `;` 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
```

## Functions:

- Nothing new!
- type()
- Piece of reusable code
- Solves particular task
- Call function instead of writing code yourself

```
In [1]: fam = [1.73, 1.68, 1.71, 1.89]

In [2]: fam

Out[2]: [1.73, 1.68, 1.71, 1.89]

In [3]: max(fam)

Out[3]: 1.89
```

### round()

```
In [6]: round(1.68, 1)
Out[6]: 1.7

In [7]: round(1.68)
Out[7]: 2

In [8]: help(round)         Open up documentation

  Help on built-in function round in module builtins:
  round(...)

      round(number[, ndigits]) -> number

      Round a number to a given precision in decimal digits (default 0
digits). This returns an int when called with one argument, otherwise
the same type as the number.

      ndigits may be negative.
```

## 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)
```

### Example Code :

```
# Create variables var1 and var2
var1 = [1, 2, 3, 4]
var2 = True

# Print out type of var1
print(type(var1)) → <class 'list'>

# Print out length of var1
print(len(var1)) → 4

# Convert var2 boolean to an integer: out2
out2 = int(var2) → 1
```

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

## 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
```

## Multiple arguments :

The square brackets around an argument indicate that the 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.

Say, 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 = ___)
```

Note: For now, we can understand an ***iterable*** as being any collection of objects, e.g. a List.

Example Code :

```
# 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]
```

## Methods :

We can think of methods as “Functions that belong to objects.”

### list methods :

- **index()**, to get the index of the first element of a list that matches its input
- **count()**, to get the number of times an element appears in a list.
- **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
- **reverse()**, that reverses the order of the elements in the list it is called on.

```
In [4]: fam

Out[4]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

In [5]: fam.index("mom")

Out[5]: 4                          "Call method index() on fam"

In [6]: fam.count(1.73) # count no. of times an element occurs in list

Out[6]: 1
```

### str methods :

```
In [7]: sister

Out[7]: 'liz'

In [8]: sister.capitalize()

Out[8]: 'Liz'

In [9]: sister.replace("z", "sa")

Out[9]: 'lisa'
```

- Everything = object
- Object have methods associated, depending on type
- Some methods change the input on which the method is applied.

```
In [14]: fam

Out[14]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

In [15]: fam.append("me")

In [16]: fam

Out[16]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me']

In [17]: fam.append(1.79)

In [18]: fam

Out[18]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89, 'me',
1.79]
```

- Functions

```
In [11]: type(fam)

Out[11]: list
```

- Methods: call functions on objects

```
In [12]: fam.index("dad")

Out[12]: 6
```

Example Code :

- 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"!

```
# 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'))
```

Example Code :

```
# 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)) → 2

# Print out how often 9.50 appears in areas
print(areas.count(9.50)) → 1

# Use append twice to add poolhouse and garage size
areas.append(24.5)
areas.append(9.50)

# Print out areas
print(areas) → [11.25, 18.0, 20.0, 10.75, 9.5, 24.5, 15.45]

# Reverse the orders of the elements in areas
areas.reverse()

# Print out areas
print(areas) → [15.45, 24.5, 9.5, 10.75, 20.0, 18.0, 11.25]
```

## Packages :

<u>Motivation :</u>

- Functions and methods are powerful
- All code in Python distribution?
- Huge code base: messy
- Lots of code you won't use
- Maintenance problem

### Packages :

```
pkg/
  mod1.py
  mod2.py
  ...
```

- Directory of Python Scripts
- Each script = module
- Specify functions, methods, types
- Thousands of packages available
- Numpy
- Matplotlib
- Scikit-learn

### Install package :

- http://pip.readthedocs.org/en/stable/installing/
- Download get-pip.py
- Terminal:
    - python3 get-pip.py
    - pip3 install numpy

## Import package :

```
In [1]: import numpy

In [2]: array([1, 2, 3])

NameError: name 'array' is not defined

In [3]: numpy.array([1, 2, 3])

Out[3]: array([1, 2, 3])

In [4]: import numpy as np

In [5]: np.array([1, 2, 3])

Out[5]: array([1, 2, 3])

In [6]: from numpy import array

In [7]: array([1, 2, 3])

Out[7]: array([1, 2, 3])
```

### Example Code :

```
# 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 * r


# 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
```

## 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]])
```

Then, you need to use this kind of import :

```
from scipy.linalg import inv as my_inv
```

## NumPy :

Lists Recap :

- Powerful
- Collection of values
- Hold different types
- Change, add, remove
- Need for Data Science
- Mathematical operations over collections
- Speed

**Problem with Lists :**

It can’t do element-wise calculation at one-go.

```
In [1]: height = [1.73, 1.68, 1.71, 1.89, 1.79]

In [2]: height

Out[2]: [1.73, 1.68, 1.71, 1.89, 1.79]

In [3]: weight = [65.4, 59.2, 63.6, 88.4, 68.7]

In [4]: weight

Out[4]: [65.4, 59.2, 63.6, 88.4, 68.7]

In [5]: weight / height ** 2

TypeError: unsupported operand type(s) for **: 'list' and 'int'
```

**Solution**: NumPy

- Numeric Python
- Alternative to Python List: NumPy Array
- Calculations over entire arrays
- Easy and Fast
- Installation
    - In the terminal: *pip3 install numpy*

**Example Code :**

Element-wise calculations.

```
In [6]: import numpy as np

In [7]: np_height = np.array(height)

In [8]: np_height

Out[8]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])

In [9]: np_weight = np.array(weight)

In [10]: np_weight

Out[10]: array([ 65.4,  59.2,  63.6,  88.4,  68.7])

In [11]: bmi = np_weight / np_height ** 2

In [12]: bmi

Out[12]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```

Comparison : **List Vs NumPy array** :

```
In [13]: height = [1.73, 1.68, 1.71, 1.89, 1.79]

In [14]: weight = [65.4, 59.2, 63.6, 88.4, 68.7]

In [15]: weight / height ** 2

TypeError: unsupported operand type(s) for **: 'list' and 'int'

In [16]: np_height = np.array(height)

In [17]: np_weight = np.array(weight)

In [18]: np_weight / np_height ** 2

Out[18]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```

**NumPy remarks :**

```
In [19]: np.array([1.0, "is", True])

# NumPy arrays: contain only one type

Out[19]: array(['1.0', 'is', 'True'], dtype='<U32')


In [20]: python_list = [1, 2, 3]

In [21]: numpy_array = np.array([1, 2, 3])

                                              Different types: different behavior!

In [22]: python_list + python_list

Out[22]: [1, 2, 3, 1, 2, 3]

In [23]: numpy_array + numpy_array

Out[23]: array([2, 4, 6])
```

**NumPy Subsetting :**

```
In [24]: bmi

Out[24]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])

In [25]: bmi[1]

Out[25]: 20.975

In [26]: bmi > 23

Out[26]: array([False, False, False,  True, False], dtype=bool)

In [27]: bmi[bmi > 23]

Out[27]: array([ 24.747])
```

Your First NumPy Array :

```
# 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'>
```

Example Code :

```
# height_in and weight_lb are available as regular lists

# Import numpy
import numpy as np

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

# Create array from weight_lb with metric units: np_weight_kg
np_weight_kg = np.array(weight_lb) * 0.453592

# Calculate the BMI: bmi
bmi = np_weight_kg / np_height_m ** 2

# Print out bmi
print(bmi)
```

## Subsetting :

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]
```

**Example Code :**

```
# 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 ; Boolean array
light = bmi < 21

# Print out light
print(light)

# Print out BMIs of all baseball players whose BMI is below 21
print(bmi[light])
```

## NumPy Side Effects :

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])
```

The above code is same as :

```
np.array([4, 3, 0]) + np.array([0, 2, 2])
```

## Type of NumPy Arrays :

```
In [1]: import numpy as np

In [2]: np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])

In [3]: np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])

In [4]: type(np_height)

Out[4]: numpy.ndarray                # ndarray = N-dimensional array

In [5]: type(np_weight)

Out[5]: numpy.ndarray
```

**2D NumPy Arrays :**

```
In [6]: np_2d = np.array([[1.73, 1.68, 1.71, 1.89, 1.79],

                          [65.4, 59.2, 63.6, 88.4, 68.7]])

In [7]: np_2d

Out[7]:

array([[  1.73,   1.68,   1.71,   1.89,   1.79],
       [ 65.4 ,  59.2 ,  63.6 ,  88.4 ,  68.7 ]])

In [8]: np_2d.shape
Out[8]: (2, 5)                 # 2 rows, 5 columns
```

**Subsetting :**

```
In [10]: np_2d[0]
Out[10]: array([ 1.73,  1.68,  1.71,  1.89,  1.79])

In [11]: np_2d[0][2]
Out[11]: 1.71

In [12]: np_2d[0,2] # comma notation <row, column>
Out[12]: 1.71

In [13]: np_2d[:,1:3]
Out[13]:
array([[  1.68,   1.71],
       [ 59.2 ,  63.6 ]])

In [14]: np_2d[1,:]
Out[14]: array([ 65.4,  59.2,  63.6,  88.4,  68.7])
```

**Example Code :**

```
# 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(np_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, and first column.

**Example Code :**

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!

- 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.

```
# 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])
```

## 2D Arithmetic

`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
```

**Example Code :**

np_baseball is coded for you; it's again a 2D numpyarray with 3 columns representing height (in inches), weight (in pounds) and age (in years).

- 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.

```
# baseball is available as a regular list of lists
# updated is available as 2D numpy array

# Import numpy package
import numpy as np

# Create np_baseball (3 cols)
np_baseball = np.array(baseball)

# Print out addition of np_baseball and updated
print(np_baseball + updated)

# Create numpy array: conversion
conversion = np.array([0.0254, 0.453592, 1])

# Print out product of np_baseball and conversion
print(np_baseball * conversion)
```

## NumPy Statistics :

This is very useful for data analysis.

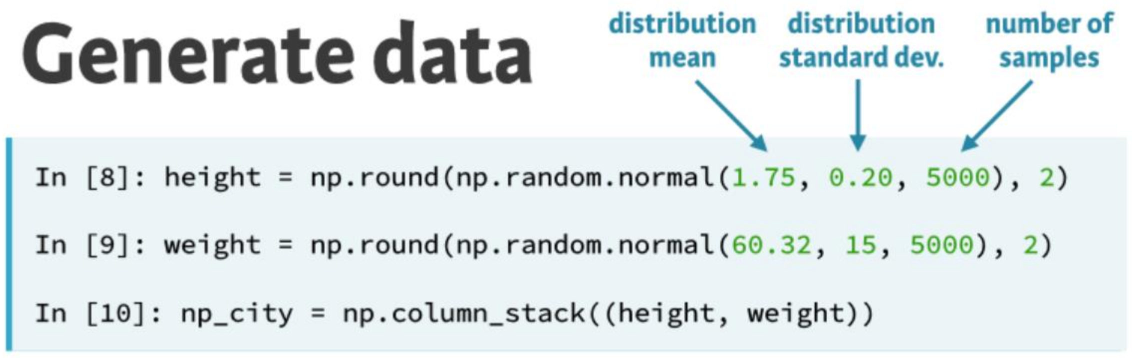
**Data analysis**

- Get to know your data
- Little data -> simply look at it
- Big data -> ?

**<u>NumPy functions :</u>**

```
In [4]: np.mean(np_city[:,0])
Out[4]: 1.7472

In [5]: np.median(np_city[:,0])
Out[5]: 1.75

In [6]: np.corrcoef(np_city[:,0], np_city[:,1])
Out[6]:
array([[ 1.     , -0.01802],
       [-0.01803,  1.     ]])

In [7]: np.std(np_city[:,0])
Out[7]: 0.1992
```

Generate Data using NumPy :



```
In [8]: height = np.round(np.random.normal(1.75, 0.20, 5000), 2)

In [9]: weight = np.round(np.random.normal(60.32, 15, 5000), 2)

In [10]: np_city = np.column_stack((height, weight))
```

- Generate data using random distribution
- Then, column stack to create a data set

Note :

It is always a good idea to check both the median and the mean, to get an idea about the overall distribution of the entire dataset.

Example Code :

```
# 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.
stddev = np.std(np_baseball[:,0])
print("Standard Deviation: " + str(stddev))

# Print out correlation between first and second column.
corr = np.corrcoef(np_baseball[:,0], np_baseball[:,1])
print("Correlation: " + str(corr))
```

**Exercise :**

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.

- 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.

```
# 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)))
```