# **Data Analysis**

## **Plan of the Module**

1. Data Analysis
2. Data Sourcing
3. Data Visualization

## **Lecture Outline**

* Jupyter (10 min)
* NumPy (30 min)
* Pandas (50 min)

## **Jupyter**

[...] is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: **data cleaning** and transformation, numerical simulation, statistical modeling, **data visualization**, machine learning, and much more.

👉 [Jupyter.org](https://jupyter.org/)

Open your Terminal:

cd ~/some/where

jupyter notebook

Let's have a *quick tour*!

## **NumPy**

Fundamental package for high-performance data manipulation with Python

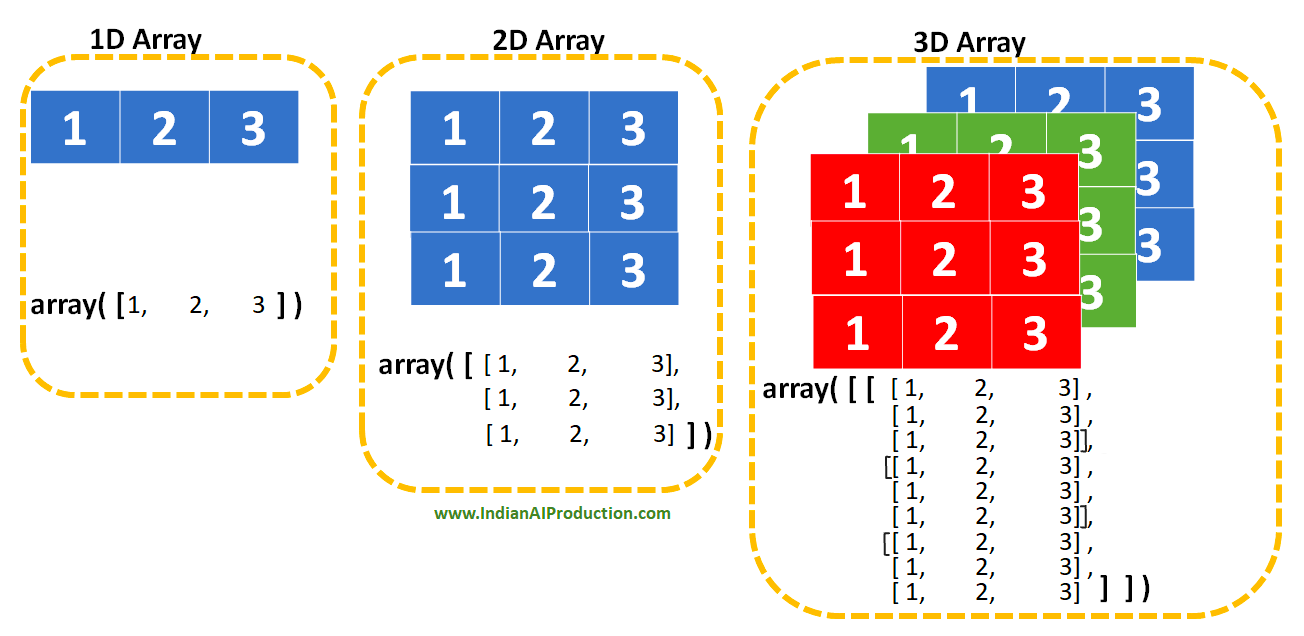
👉 [NumPy.org](https://www.numpy.org/)

👉 [NumPy Cheat Sheet](https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf) to print/bookmark

The key concept NumPy introduces is the **N-dimensional Array** (ndarray)

**Characteristics of the ndarray:**

* It is **multidimensional**
* Data is **homogenous**
* It has a **fixed size** defined upon creation



**import** **numpy** **as** **np** *# canonical import*

my\_list = [[1, 2, 3], [4, 5, 6]]

print(type(my\_list))

my\_list *# list of lists*

<class 'list'>

[[1, 2, 3], [4, 5, 6]]

my\_array = np.array([[1, 2, 3], [4, 5, 6]])

print(type(my\_array))

my\_array *# ndarray*

<class 'numpy.ndarray'>

array([[1, 2, 3],

[4, 5, 6]])

*# Key attributes of ndarrays*

print('my\_array.ndim: ', my\_array.ndim)

print('my\_array.shape:', my\_array.shape)

print('my\_array.size: ', my\_array.size)

print('my\_array.dtype:', my\_array.dtype)

my\_array.ndim: 2

my\_array.shape: (2, 3)

my\_array.size: 6

my\_array.dtype: int64

### **Data Selection 😎**

*# Let's build a 2D-array from a list of lists*

data\_list = [

[ 0, 1, 2, 3, 4],

[10, 11, 12, 13, 14],

[20, 21, 22, 23, 24],

[30, 31, 32, 33, 34],

[40, 41, 42, 43, 44],

]

data\_np = np.array(data\_list)

data\_np

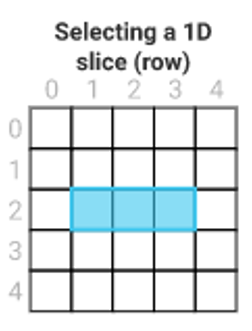
array([[ 0, 1, 2, 3, 4],

[10, 11, 12, 13, 14],

[20, 21, 22, 23, 24],

[30, 31, 32, 33, 34],

[40, 41, 42, 43, 44]])



*# Pure Python*

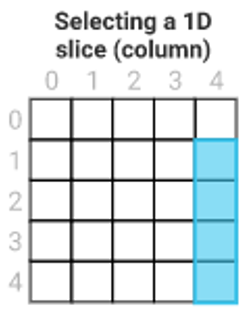
data\_list[2][1:4]

[21, 22, 23]

*# NumPy*

data\_np[2, 1:4] *# data\_np[row(s), column(s)]*

array([21, 22, 23])



*# Pure Python*

selection = []

**for** index, row **in** enumerate(data\_list):

**if** index > 0:

selection.append(row[4])

selection *# we could also have used list comprehension for fewer lines*

[14, 24, 34, 44]

*# NumPy*

data\_np[1:, 4] *# '1:' means from line 1 until the end*

array([14, 24, 34, 44])

### **General Syntax for Slicing**

ndarray[start:stop:step]

array = np.arange(0, 10)

array

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

array[1:7:2]

array([1, 3, 5])

### **Vectorized Operations ⚡️**

Let's compute the sum, row by row (8 additions), to create a 1D-vector

my\_list = [

[6, 5],

[1, 3],

[5, 6],

[1, 4],

[3, 7],

[5, 8],

[3, 5],

[8, 4],

]

*# Python way*

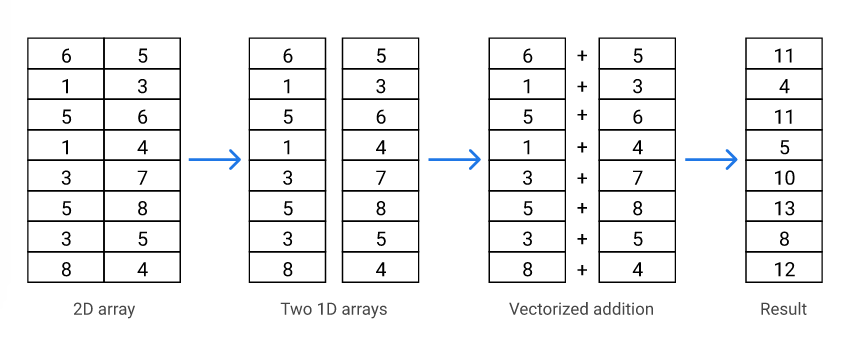
sums = []

**for** row **in** my\_list:

sums.append(row[0] + row[1]) *# standard integer "+" operator*

sums

### **The NumPy Way**

****

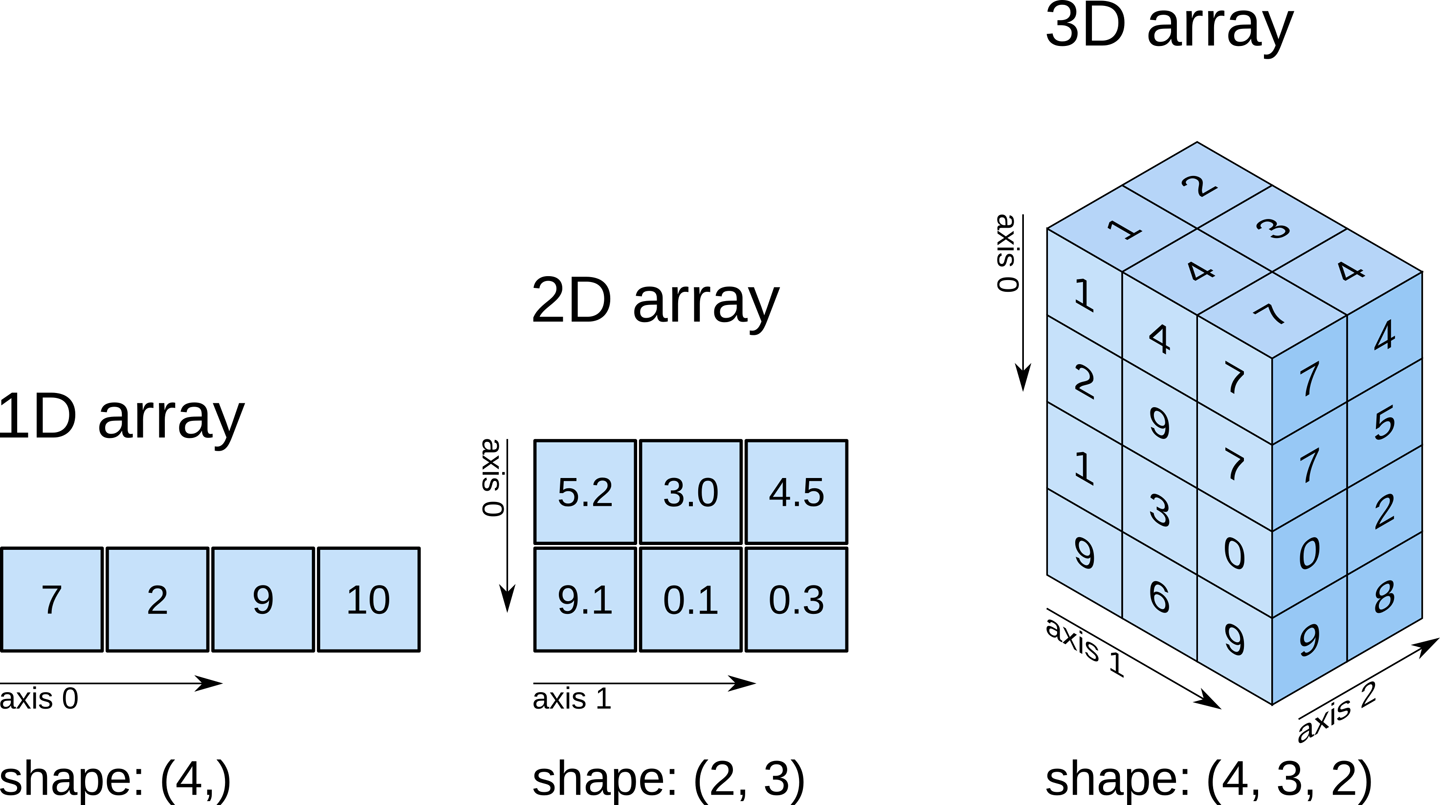
my\_array = np.array(my\_list)

my\_sum = my\_array[:, 0] + my\_array[:, 1] *# vectorial "+" operator*

my\_sum

array([11, 4, 11, 5, 10, 13, 8, 12])

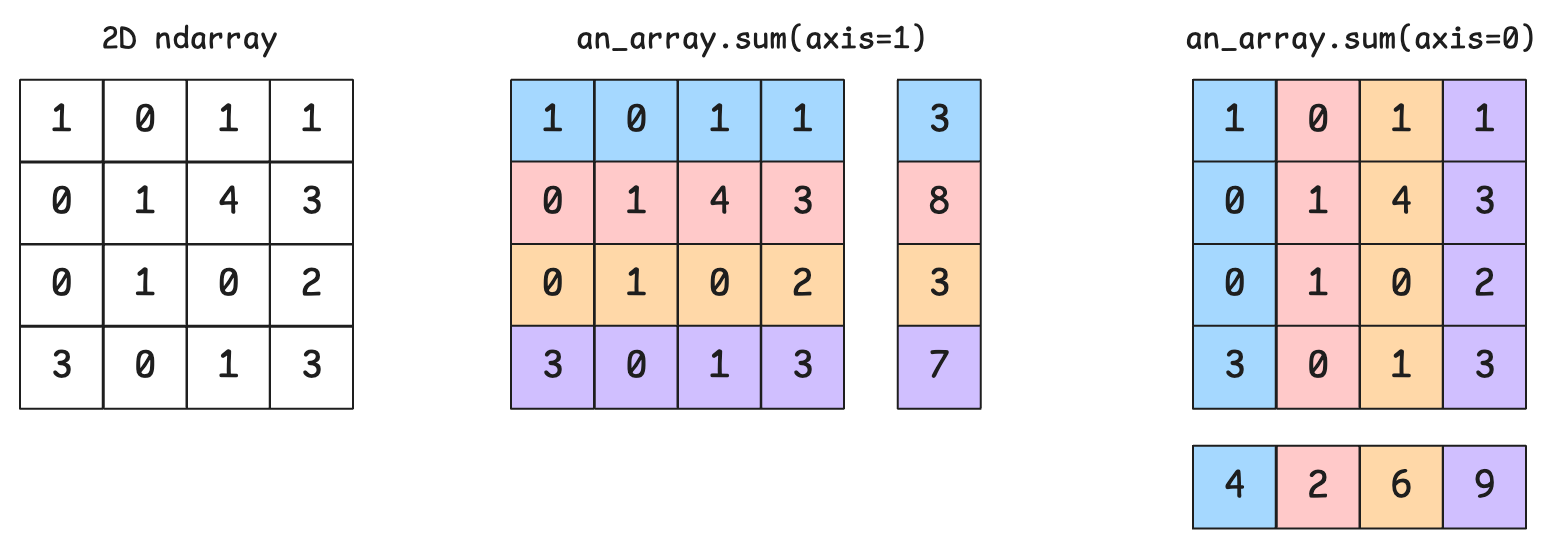
### **Axes 🤯**

****

**2D Example**

an\_array.sum(axis=0) *# eq. to A[0,:] + A[1,:] + A[2,:] + ...*

an\_array.sum(axis=1) *# eq. to A[:,0] + A[:,1] + A[:,2] + ...*



The following code is equivalent:

an\_array.sum(axis=0)

np.sum(an\_array, axis=0)

### **How much faster is NumPy? ⚡️**

*# 2D-array of shape (10.000, 10.000) with random floats in the interval [0, 1]. That's 100M numbers!*

my\_array = np.random.rand(10000, 10000)

array\_list = my\_array.tolist()

%%time

total = 0

**for** row **in** array\_list:

**for** number **in** row:

total += number

round(total, 2)

CPU times: user 3.84 s, sys: 146 ms, total: 3.99 s

Wall time: 4.01 s

50003977.92

%%time

round(np.sum(my\_array), 2)

CPU times: user 26.7 ms, sys: 1.5 ms, total: 28.2 ms

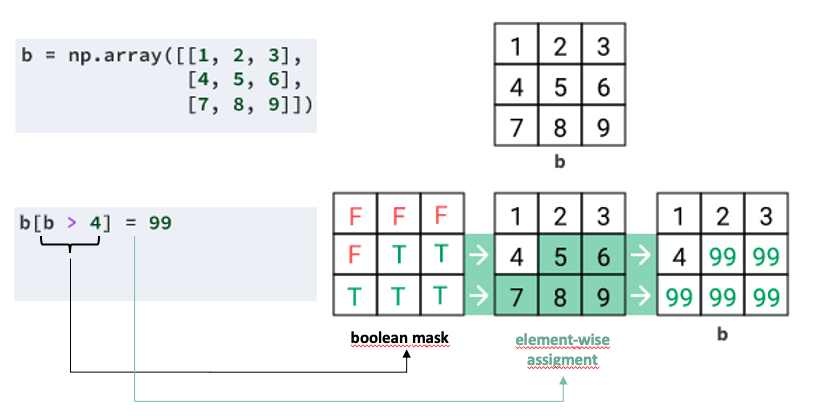
Wall time: 27.4 ms

50003977.92

⚡️ NumPy is **two orders of magnitude** (100x) faster!

### **Boolean Indexing 🔥**

Build a **boolean mask** from an ndarray.



### **Limitations of NumPy**

* Lack of support for **column names**
* **Only one** data type per ndarray
* Some useful data processing methods are missing

👉 **Pandas** builds on NumPy to solve these problems

## **Introduction to Pandas**

[...] is an open source library providing high-performance easy-to-use data structures and data analysis tools for Python.

👉 [Pandas.pydata.org](https://pandas.pydata.org/)

👉 [Pandas cheat sheet](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf) to print/bookmark

### **Pandas Series**

* Pandas' equivalent to NumPy's **1D-array** (both accept the same methods)
* Has an additional index
* Has support for multiple data types

👉 [pandas.Series](https://pandas.pydata.org/docs/reference/api/pandas.Series.html)

**import** **pandas** **as** **pd** *# canonical import*

my\_series = pd.Series(data=[1, 2, 'three'], index=['id1', 'id2', 'id3'])

my\_series = pd.Series({'id1': 1, 'id2': 2, 'id3': 'three'})

my\_series

id1 1

id2 2

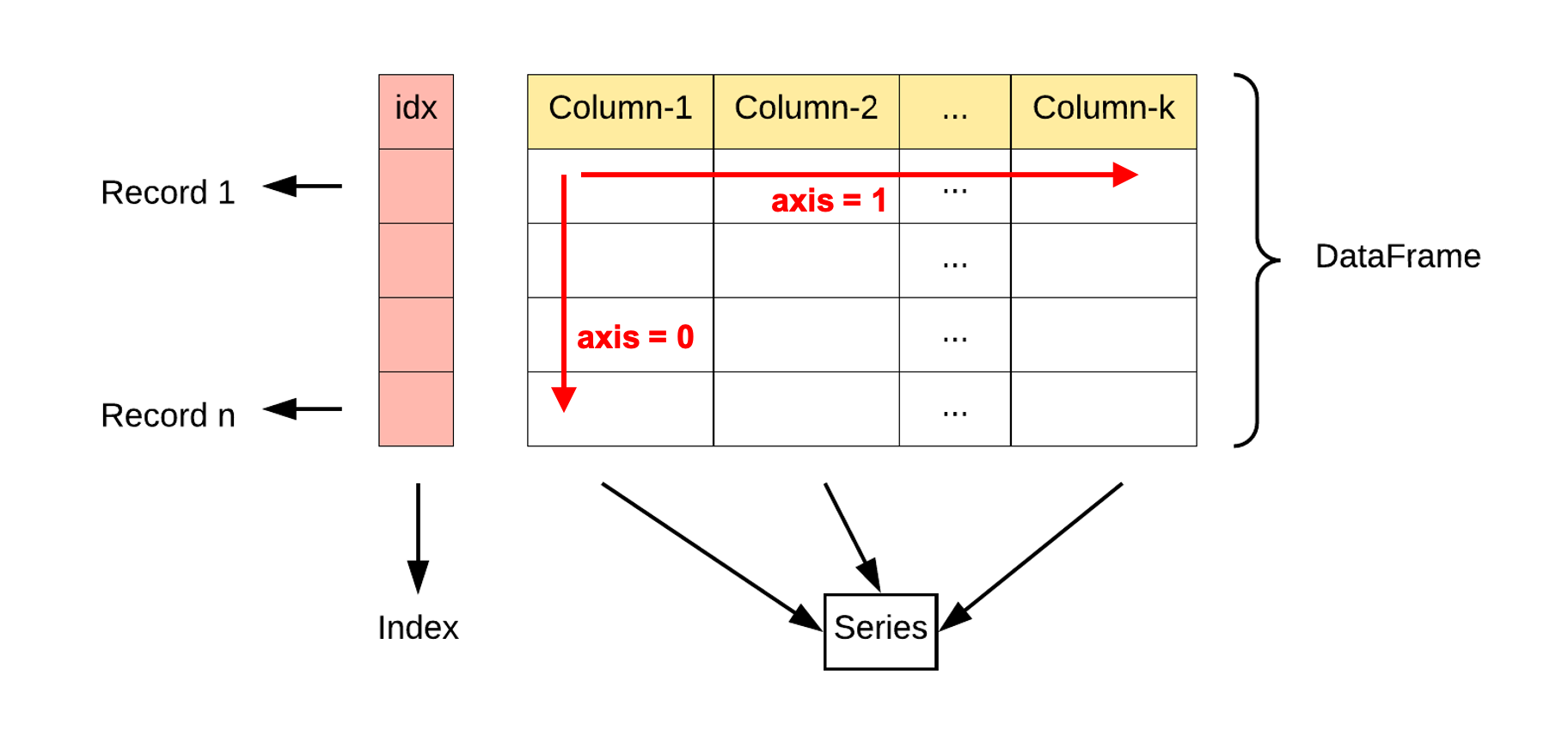
id3 three

dtype: object

### **Pandas DataFrames**

* Pandas' equivalent of a NumPy **2D-array**:
* Has additional labels on both axes (rows and columns)
* Has support for multiple data types

👉 [pandas.DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html)



**import** **pandas** **as** **pd**

df = pd.DataFrame(

[[4, 7, 10],

[5, 8, 11],

[6, 9, 12]],

index=['row\_1', 'row\_2', 'row\_3'],

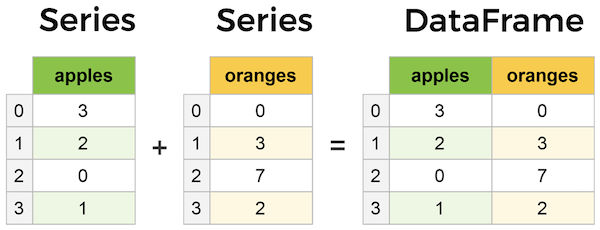
columns=['col\_a', 'col\_b', 'col\_c']

)

df

|  | **col\_a** | **col\_b** | **col\_c** |
| --- | --- | --- | --- |
| **row\_1** | 4 | 7 | 10 |
| **row\_2** | 5 | 8 | 11 |
| **row\_3** | 6 | 9 | 12 |

### **A DataFrame is a dictionary of Series**

****

apples = pd.Series(data=[1, 2, 3], index=['id1', 'id2', 'id3'])

oranges = pd.Series(data=[4, 5, 6], index=['id1', 'id2', 'id3'])

dict\_of\_series = {

'apples': apples,

'oranges': oranges,

}

pd.DataFrame(dict\_of\_series)

|  | **apples** | **oranges** |
| --- | --- | --- |
| **id1** | 1 | 4 |
| **id2** | 2 | 5 |
| **id3** | 3 | 6 |

## **Exploratory Data Analysis (EDA)**

Let's start a new notebook to explore the following dataset: [Countries of the World](https://www.kaggle.com/fernandol/countries-of-the-world).

You can have a look at it [in this Gist](https://gist.github.com/ssaunier/fcf6e1c9485f2d64607a093795372339) and download it with:

curl -s -L https://wagon-public-datasets.s3.amazonaws.com/02-Data-Toolkit/01-Data-Analysis/countries.csv > countries.csv

head -n 3 countries.csv

This is how notebooks typically start:

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

### **Notebook Superpowers**

In a new cell:

pd.read<TAB>

pd.read\_csv<SHIFT+TAB> *# (up to four times)*

Go ahead and load the CSV into a countries\_df DataFrame:

file = 'countries.csv' *# path relative to your notebook*

countries\_df = pd.read\_csv(file, decimal=',')

### **Get a Quick Sense of the Data**

Here are some utility methods to call on a fresh DataFrame:

countries\_df.shape *# => Tuple representing the dimensionality of the DataFrame*

Replace .shape with:

* [pandas.DataFrame.dtypes](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dtypes.html)
* [pandas.DataFrame.info()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.info.html)
* [pandas.DataFrame.describe()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.describe.html)

You can also do:

countries\_df.isnull().sum()

### **Get a Quick Look**

countries\_df.head()

countries\_df.tail()

### **Same logic as SQL!**

You can manipulate a DataFrame in the same way you query a relational database's table.

👉 [Pandas documentation: comparison with SQL](https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html)

### **Reading Columns**

Use the [] syntax to get one or many columns:

countries\_df['Country']

type(countries\_df['Country']) *# => pandas.core.series.Series*

countries\_df[['Country', 'Region']]

type(countries\_df[['Country']]) *# => pandas.core.frame.DataFrame*

### **Group of Rows/Columns**

countries\_df.loc[0:5, ['Country', 'Region']] *# from row index 0 to 5 (included)*

👉 After the lecture, read [this Stackoverflow Q&A thread](https://stackoverflow.com/questions/48409128/what-is-the-difference-between-using-loc-and-using-just-square-brackets-to-filte/48411543#48411543)

## **Boolean Indexing with Pandas**

🤔 What are the countries with **more than one billion** inhabitants?

Pure Python (naive) implementation:

big\_countries = []

**for** index, country **in** countries\_df.iterrows():

**if** country['Population'] > 1\_000\_000\_000:

big\_countries.append(country)

pd.DataFrame(big\_countries)

In Pandas, this is a **one-liner** with **Boolean Indexing**:

countries\_df[countries\_df['Population'] > 1\_000\_000\_000]

🤔 What are the countries of the **American** continent?

american = countries\_df['Region'].str.contains('AMER')

countries\_df[american]

🤔 What are the countries of **Europe**?

We can use [pandas.Series.isin()](https://pandas.pydata.org/docs/reference/api/pandas.Series.isin.html)

countries\_df[countries\_df['Region'].isin(['WESTERN EUROPE', 'EASTERN EUROPE'])]

But why are there no results?

countries\_df['Region'].unique()

We need to **clean up** first:

countries\_df['Region'] = countries\_df['Region'].str.strip()

If we want to answer the **inverse** question, we can use the bitwise operator ~:

countries\_df[~countries\_df['Region'].isin(['WESTERN EUROPE', 'EASTERN EUROPE'])]

## **Re-Indexing**

countries\_df['Country'] = countries\_df['Country'].map(str.strip)

countries\_df.set\_index('Country', inplace=**True**)

The index is no longer a sequence of integers, but instead the countries' names!

We now can do something like this:

*# Get region names and population from France to Germany*

countries\_df.loc['France':'Germany', ['Region', 'Population']]

## **Sorting**

We can sort by the index with [pandas.DataFrame.sort\_index](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_index.html):

countries\_df.sort\_index(ascending=**False**)

We can sort by specific columns with [pandas.DataFrame.sort\_values](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html):

countries\_df.sort\_values(by='Population', ascending=**False**)

*# Makes sure NaNs are shown at the top*

countries\_df.sort\_values(by='GDP ($ per capita)', na\_position='first')

## **Grouping**

Very close to [GROUP BY in SQL](https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html#group-by); it's a 3-step process:

1. **Split**: a DataFrame is split into groups, depending on chosen keys
2. **Apply**: an **aggregative function** (sum, mean, etc.) is applied to each group
3. **Combine**: results from the previous operations are merged (i.e. reduced) into one new DataFrame

🤔 Which region of the world is the most populated?

regions = countries\_df.groupby('Region')

regions[['Population', 'Area (sq. mi.)']].sum()

regions[['Population', 'Area (sq. mi.)']].sum() \

.sort\_values('Population', ascending=**False**)

## **Plotting**

%matplotlib inline

**import** **matplotlib**

gdp = 'GDP ($ per capita)'

top\_ten\_countries\_df = countries\_df[[gdp]] \

.sort\_values(gdp, ascending=**False**) \

.head(10)

top\_ten\_countries\_df

top\_ten\_countries\_df.plot(kind="bar")

## **One more thing...**

## **Testing in Notebooks**

It is a bit different from how we have been testing the Python files so far.

💻 Let's take a look at the **first challenge** and see how you can check your results directly inside your notebook!

## **Bibliography**

* 📄 [Master NumPy arrays](https://towardsdatascience.com/here-are-30-ways-that-will-make-you-a-pro-at-creating-numpy-arrays-932b77d9a1eb)

## **Your Turn!**