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

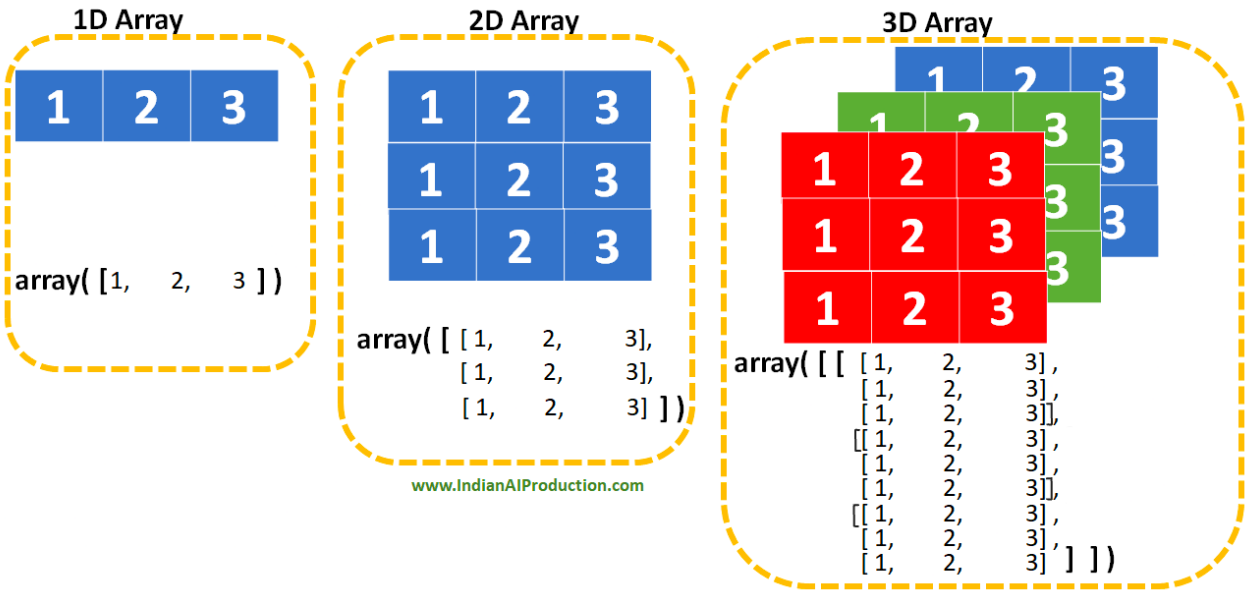
👉 NumPy.org (https://www.numpy.org)

👉 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



In [ ]:
```
import numpy as np # canonical import
```

In [ ]:
```
my_list = [[1, 2, 3], [4, 5, 6]]
print(type(my_list))

my_list # list of lists
```

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

Out[ ]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [ ]:
```
my_array = np.array([[1, 2, 3], [4, 5, 6]])
print(type(my_array))

my_array # ndarray
```

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

Out[ ]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

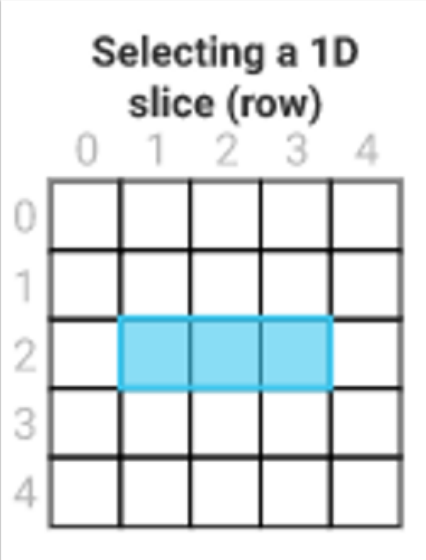
In [ ]:
```
# Key attributes of ndarrays
print('my_array.ndim: ', my_array.ndim)
print('my_array.shape:', my_array.shape)
print('my_array.size: ', my_array.size)
print('my_array.dtype:', my_array.dtype)
```

```
my_array.ndim:  2
my_array.shape: (2, 3)
my_array.size:  6
my_array.dtype: int64
```

Data Selection 😎

In [ ]:
```
# 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
```

Out[ ]:
```
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]])
```

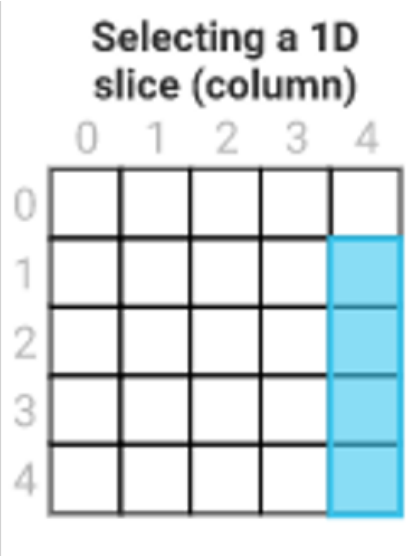


In [ ]:
```
# Pure Python
data_list[2][1:4]
```

Out[ ]:
```
[21, 22, 23]
```

In [ ]:
```
# NumPy
data_np[2, 1:4] # data_np[row(s), column(s)]
```

Out[ ]:
```
array([21, 22, 23])
```



In [ ]:
```
# 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
```

Out[ ]:
```
[14, 24, 34, 44]
```

In [ ]:
```
# NumPy
data_np[1:, 4] # '1:' means from line 1 until the end
```

Out[ ]:
```
array([14, 24, 34, 44])
```

General Syntax for Slicing

```
ndarray[start:stop:step]
```

In [ ]:
```
array = np.arange(0, 10)
array
```

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

In [ ]:
```
array[1:7:2]
```

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

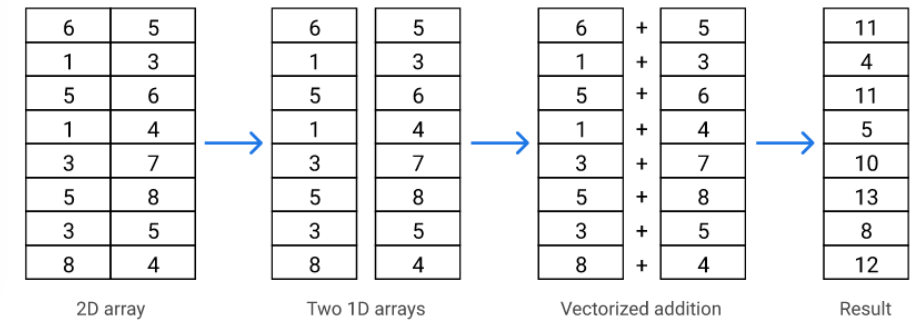
Vectorized Operations ⚡

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

In [ ]:
```
my_list = [
    [6, 5],
    [1, 3],
    [5, 6],
    [1, 4],
    [3, 7],
    [5, 8],
    [3, 5],
    [8, 4],
]
```

In [ ]:
```
# Python way
sums = []

for row in my_list:
    sums.append(row[0] + row[1]) # standard integer "+" operator

sums
```

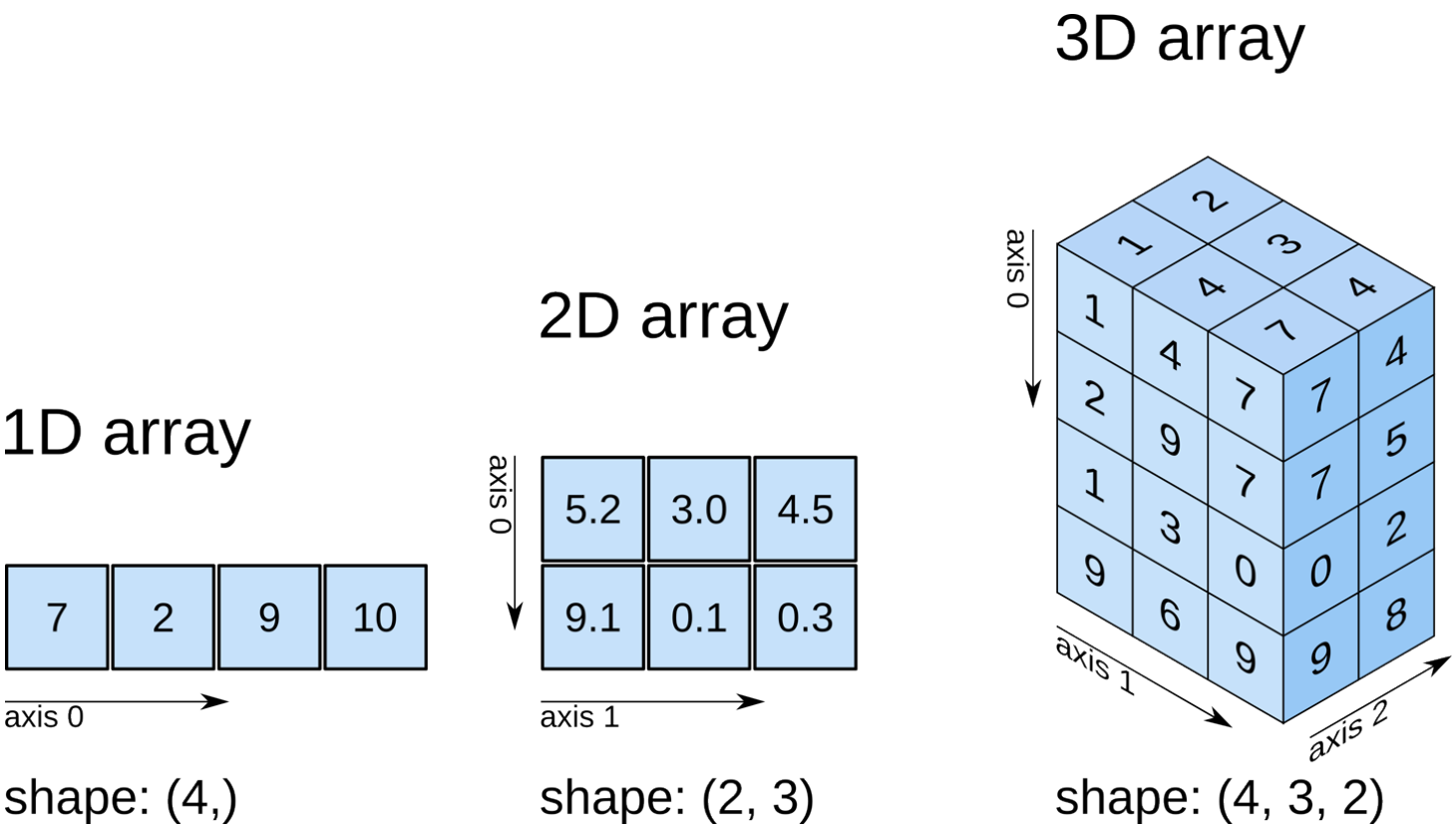
The NumPy Way



In [ ]:
```
my_array = np.array(my_list)

my_sum = my_array[:, 0] + my_array[:, 1] # vectorial "+" operator
my_sum
```

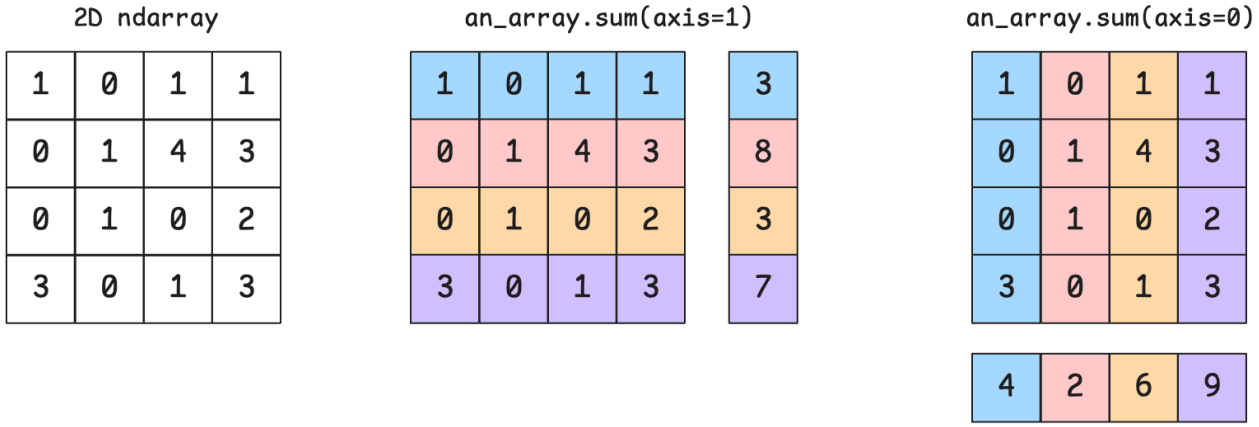
Out[ ]:
```
array([11,  4, 11,  5, 10, 13,  8, 12])
```

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? ⚡

In [ ]:
```
# 2D-array of shape (10.000, 10.000) with random floats in the interva
l [0, 1). That's 100M numbers!
my_array = np.random.rand(10000, 10000)
array_list = my_array.tolist()
```

In [ ]:
```
%%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
```

Out[ ]:
```
50003977.92
```

In [ ]:
```
%%time
round(np.sum(my_array), 2)
```

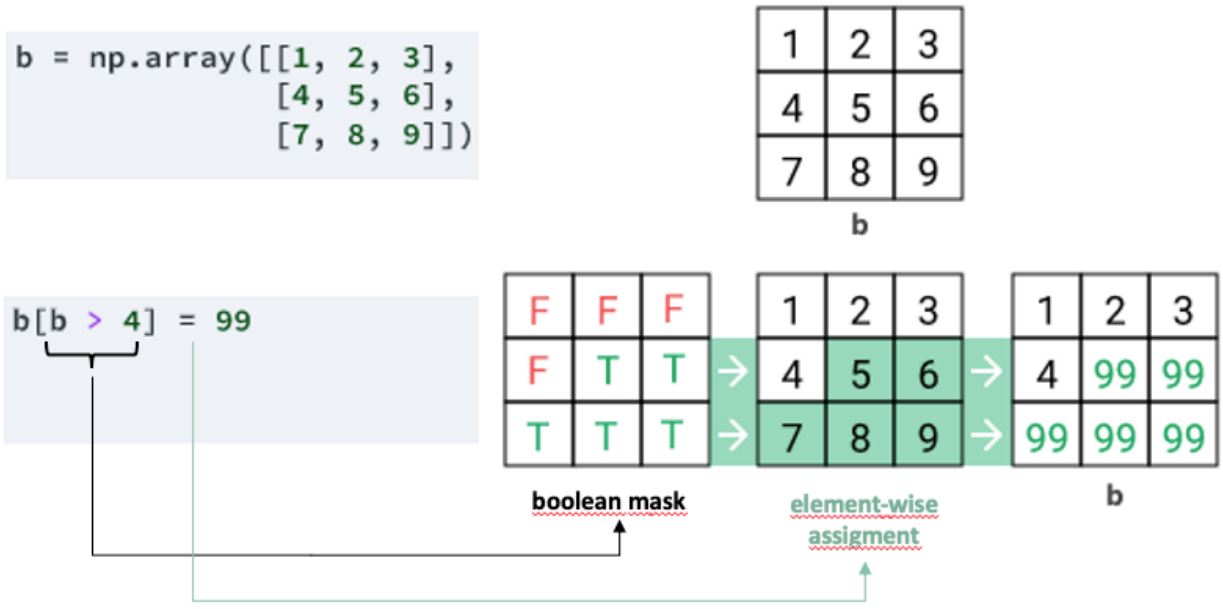
```
CPU times: user 26.7 ms, sys: 1.5 ms, total: 28.2 ms
Wall time: 27.4 ms
```

Out[ ]:
```
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)

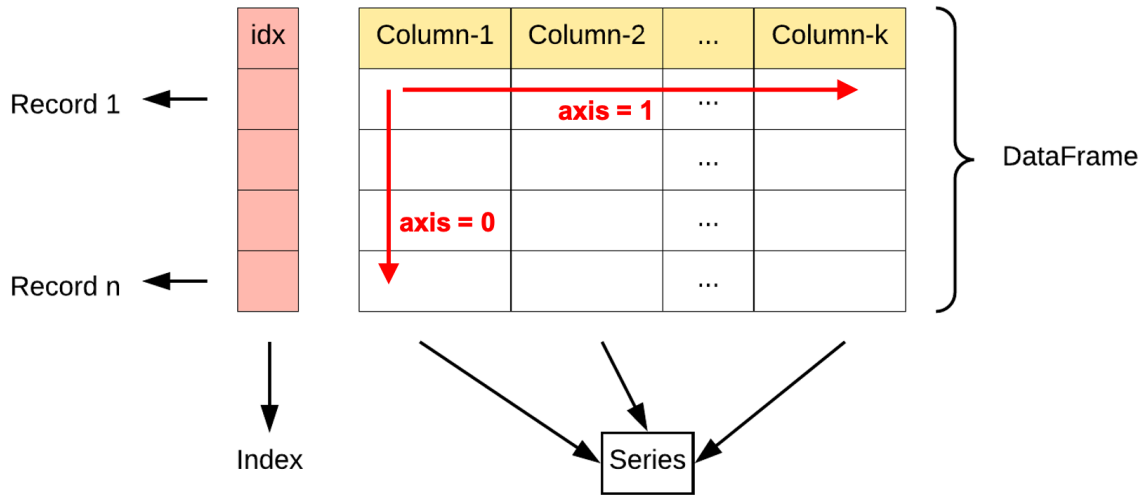
In [ ]:
```
import pandas as pd # canonical import

my_series = pd.Series(data=[1, 2, 'three'], index=['id1', 'id2', 'id
3'])
my_series = pd.Series({'id1': 1, 'id2': 2, 'id3': 'three'})

my_series
```

Out[ ]:
```
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)

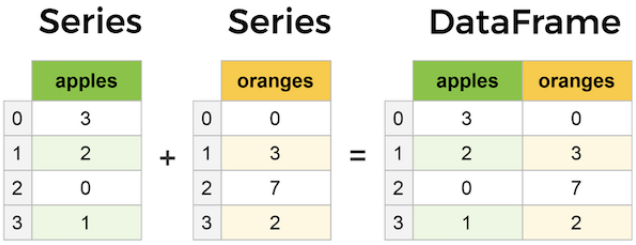


In [ ]:
```
import pandas as pd

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
```

Out[ ]:

	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



In [ ]:
```
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)
```

Out[ ]:

	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)](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)](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 DataF
rame
```

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 (incl
uded)
```

👉 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)](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 EUROP
E'])]
```

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 EURO
PE'])]
```

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!