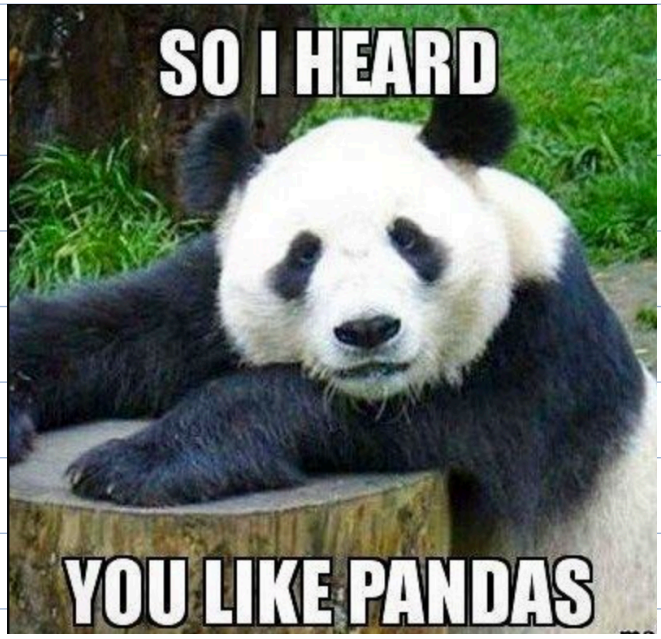
Content

- Introduction to Pandas
- DataFrame & Series
- Creating DataFrame from Scratch (Post-read)
- Basic ops on a DataFrame
- Basic ops on Columns
  - Accessing column(s)
  - Check for unique values
  - Rename column
  - Deleting column(s)
  - Creating new column(s)
- Basic ops on Rows
  - Implicit/Explicit index
  - Indexing in Series
  - Slicing in Series
    - loc/iloc
  - Indexing/Slicing in DataFrame



Pandas Installation

```
!pip install pandas
```

Importing Pandas

- You should be able to import Pandas after installing it.
- We'll import `pandas` using its **alias name** `pd`.

```
import pandas as pd
import numpy as np
```

Why use Pandas?

- The major **limitation of numpy** is that it can only work with one datatype at a time.
- Most real-world datasets contain a mix of different datatypes.
  - **names of a place would be string**
  - **population of a place would be int**

It is difficult to work with data having **heterogeneous values** using Numpy.

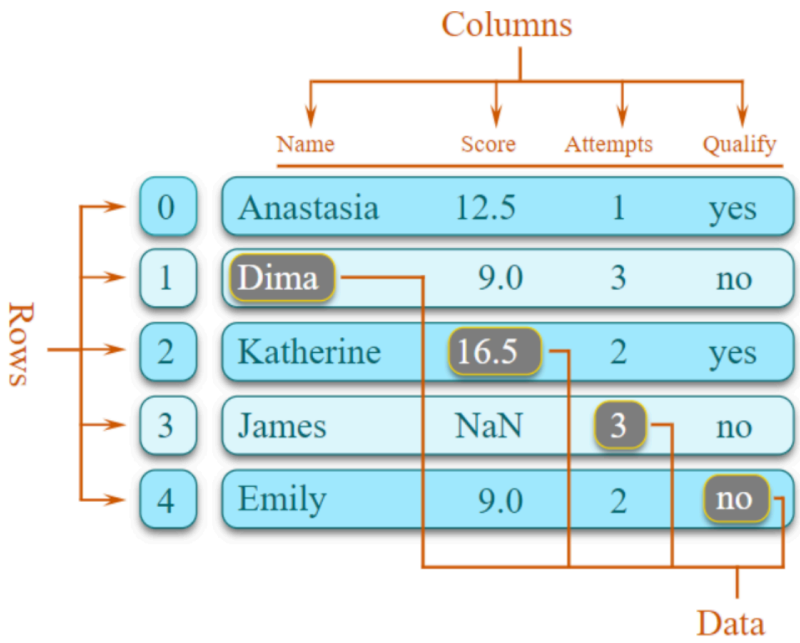
On the other hand, Pandas can work with numbers and strings together.

Problem Statement

- Imagine that you are a Data Scientist with McKinsey.
- McKinsey wants to understand the relation between GDP per capita and life expectancy for their clients.
- The company has obtained data from various surveys conducted in different countries over several years.
- The acquired data includes information on -
  - Country
  - Population Size
  - Life Expectancy
  - GDP per Capita
- We have to analyse the data and draw inferences that are meaningful to the company.

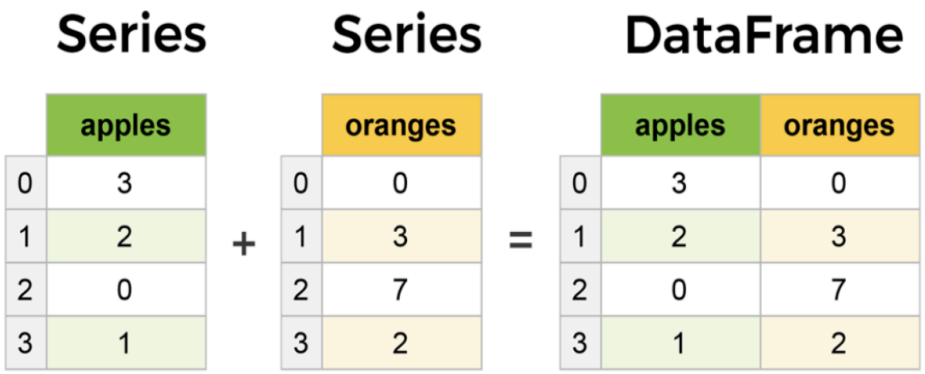
What is a Pandas DataFrame?

- A DataFrame is a **table-like (structured)** representation of data in Pandas.
- Considered as a **counterpart of 2D matrix** in Numpy.



Series

- It means that a Series is a **single column of data**.
- Multiple Series are stacked together to form a DataFrame.



`df.info()` gives a list of columns with:

- **Name** of columns
- **How many non-null values (blank cells)** each column has.
- **Type of values** in each column - int, float, etc.

By default, it shows **Dtype** as `object` for anything other than **int or float**.

How can we get the names of all these cols?

We can do it in two ways:

1. `df.columns`
2. `df.keys()`

How can we access these columns?

1) df['country'].head() → Access single column
2) df[['country', 'life_exp']].head() → Access multiple column
3) df[['country']].head() → single column → returns df.

How can we find the countries that have been surveyed?

We can find the unique values in the `country` column.

Code:

```
df['country'].unique()
```

What if you also want to check the count of occurence of each country in the dataframe?

Code:

```
df['country'].value_counts()
```

Note: `value_counts()` shows the output in **decreasing order of frequency**.

What if we want to change the name of a column?

We can rename the column by

- passing the dictionary with `old_name:new_name` pair
- specifying `axis=1`

Code:

```
df.rename({"population":"Population", "country":"Country" }, axis=1)
```

Alternatively, we can also rename the column

- without specifying `axis`
- by using the `column` parameter

Code:

```
df.rename(columns={"country":"Country"})
```

The changes doesn't happen in original dataframe unless we specify a parameter called `inplace` as True.

```
df.rename({"country":"Country"}, axis=1, inplace=True)
```

Note

- `.rename` has default value of axis=0
- If two columns have the **same name**, then `df['column']` will display both columns.

Another way to access column name

df.country // df.<col name>

This however doesn't work everytime.

What do you think could be the problem here?

- If the column names are **not strings**
  - Starting with **number**: e.g., `2nd`
  - Contains a **whitespace**: e.g., `Roll Number`
- If the column names conflict with **methods of the DataFrame**
  - e.g. `shape`

How can we delete columns from a dataframe?

Code:

```
df.drop('continent', axis=1)
```

The `drop()` function takes two parameters:

- column name
- axis

By default, the value of `axis` is 0.

An alternative to the above approach is using the "columns" parameter as we did in `rename()`.

Code:

```
df.drop(columns=['continent'])
```

Has the column been permanently deleted?

No, the column `continent` is still there in the original dataframe.

Do you see what's happening here?

We only got a **view of dataframe** with column `continent` dropped.

How can we permanently drop the column?

- We can either **re-assign** it `df = df.drop('continent', axis=1)`
- Or we can **set the parameter** `inplace=True`
  - By default, `inplace=False`.

What if we want to create a new column?

- We can either use values from **existing columns**.
- Or we can create our own values.

How to create a column using values from an existing column?

Code:

```
df["year+7"] = df["year"] + 7
df.head()
```

How can we create a new column from our own values?

- We can either **create a list**.
- Or we can **create a Pandas Series** from a list/numpy array for our new column.

Basic operations on Rows

Just like columns, do rows also have labels? Yes.

- **Can we change row labels (like we did for columns)?**
- **What if we want to start indexing from 1 (instead of 0)?**

Code:

```
df.index = list(range(1, df.shape[0]+1)) # create a list of indices of same length
```

Explicit and Implicit Indices

What are these row labels/indices exactly?

- They can be called identifiers of a particular row.
- Specifically known as **explicit indices**.

Additionally, can a series/dataframe also use Python style indexing? Yes.

- The Python style indices are known as **implicit indices**.

How can we access explicit index of a particular row?

- using `df.index[]`
- Takes **impicit index** of row to give its **explicit index**.

But why not use just implicit indexing?

Explicit indices can be changed to any value of any datatype.

- e.g. explicit index of 1st row can be changed to `first`
- Or something like a floating point value, say `1.0`

- **Indexing in Series** used **explicit indices**
- **Slicing** however used **implicit indices**

1. `loc`

- Allows indexing and slicing that always references the explicit index.

Code:

```
df.loc[1]
```

- The **range is inclusive** of **end point** for `loc`.

2. `iloc`

- Allows indexing and slicing that always references the implicit index.

Code:

```
df.iloc[1]
```

Will `iloc` also consider the range inclusive?

Code:

```
df.iloc[0:2]
```

Output

	Country	year	population	life_exp	gdp_cap
1	Afghanistan	1952	8425333	28.801	779.445314
2	Afghanistan	1957	9240934	30.332	820.853030

No, because `iloc` **works with implicit Python-style indices**.

Which one should we use?

- Generally, explicit indexing is considered to be better than implicit indexing.
- But it is recommended to always use both `loc` and `iloc` to avoid any confusions.