# 01-Pandas-Lecture-McKinsey

## Outline

- **Installation of Pandas**
  - Importing pandas
  - Importing the dataset
  - Dataframe/Series
- **Basic ops on a DataFrame**
  - df.info()
  - df.head()
  - df.tail()
  - df.shape()
- **Creating Dataframe from Scratch**
- **Basic ops on columns**
  - Different ways of accessing cols
  - Check for unique values
  - Rename column
  - Deleting column
  - Creating new column
- **Basic ops on rows**
  - Implicit/Explicit index
  - df.index
  - Indexing in Series
  - Slicing in Series
  - loc/iloc

# Installing Pandas

```
# !pip install pandas
```

## Importing Pandas

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

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

## Introduction: Why to use Pandas?

How is it different from numpy ?

- The major **limitation of numpy** is that it can only work with 1 datatype at a time
- Most real-world datasets contain a mixture of different datatypes
  - Like **names of places would be string** but their **population would be int**

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

Pandas can work with numbers and strings together

So lets see how we can use pandas

## Imagine that you are a Data Scientist with McKinsey

- McKinsey wants to understand the relation between **GDP per capita** and **life expectancy** and various trends for their clients.
- The company has acquired **data from multiple surveys** in different countries in the past
- This contains info of several years about:
    - country
    - population size
    - life expectancy
    - GDP per Capita
- We have to analyse the data and draw **inferences** meaningful to the company

## Reading dataset in Pandas

Link: https://drive.google.com/file/d/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_/view?usp=sharing

```
!wget "https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_" -O mckinsey.csv
```

```
--2023-09-11 18:16:39--  https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_
Resolving drive.google.com (drive.google.com)... 173.194.213.102, 173.194.213.113, 173.194.213.139, ...
Connecting to drive.google.com (drive.google.com)|173.194.213.102|:443... connected.
HTTP request sent, awaiting response... 303 See Other
Location: https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/l1h3nkkdhuu158c3gc
Warning: wildcards not supported in HTTP.
--2023-09-11 18:16:40--  https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/l1h
Resolving doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleusercontent.com)... 173.194.212.132, 2607:f8b0:400c
Connecting to doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleusercontent.com)|173.194.212.132|:443... connec
HTTP request sent, awaiting response... 200 OK
Length: 83785 (82K) [text/csv]
Saving to: 'mckinsey.csv'

mckinsey.csv        100%[===================>]  81.82K  --.-KB/s    in 0.001s

2023-09-11 18:16:40 (76.9 MB/s) - 'mckinsey.csv' saved [83785/83785]
```

### Now how should we read this dataset?

Pandas makes it very easy to work with these kinds of files

```
df = pd.read_csv('mckinsey.csv') # We are storing the data in df
df
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

1704 rows × 6 columns

## Dataframe and Series

### What can we observe from the above dataset ?

We can see that it has:

- 6 columns
- 1704 rows

What do you think is the datatype of `df` ?

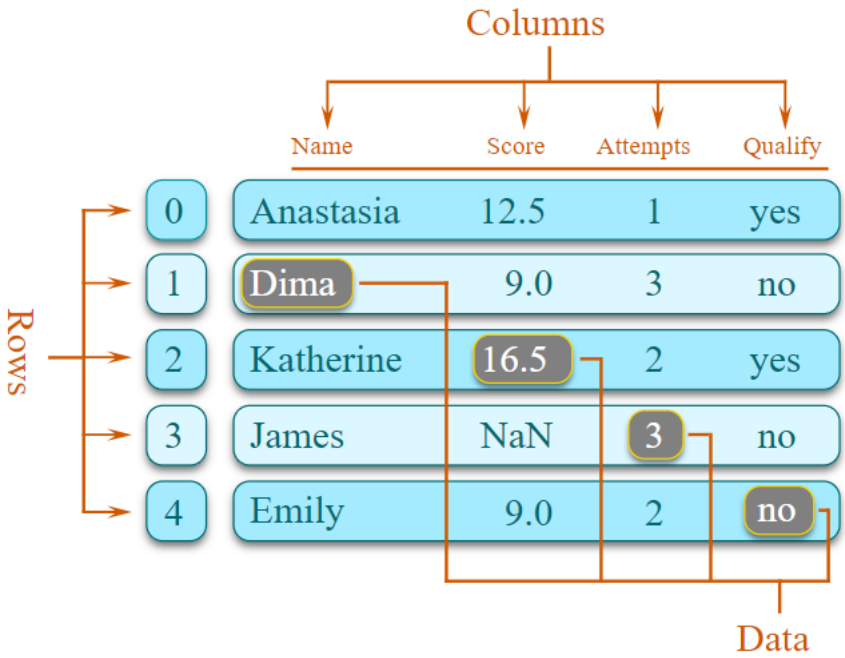
```
type(df)
```

```
pandas.core.frame.DataFrame
```

Its a **pandas DataFrame**

## What is a pandas DataFrame ?

- It is a table-like representation of data in Pandas => Structured Data
- **Structured Data** here can be thought of as **tabular data in a proper order**
- Considered as **counterpart of 2D-Matrix** in Numpy




## Now how can we access a column, say `country` of the dataframe?

```
df["country"]
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

As you can see we get all the values in the column **country**

## Now what is the data-type of a column?

```
type(df["country"])
```

```
pandas.core.series.Series
```

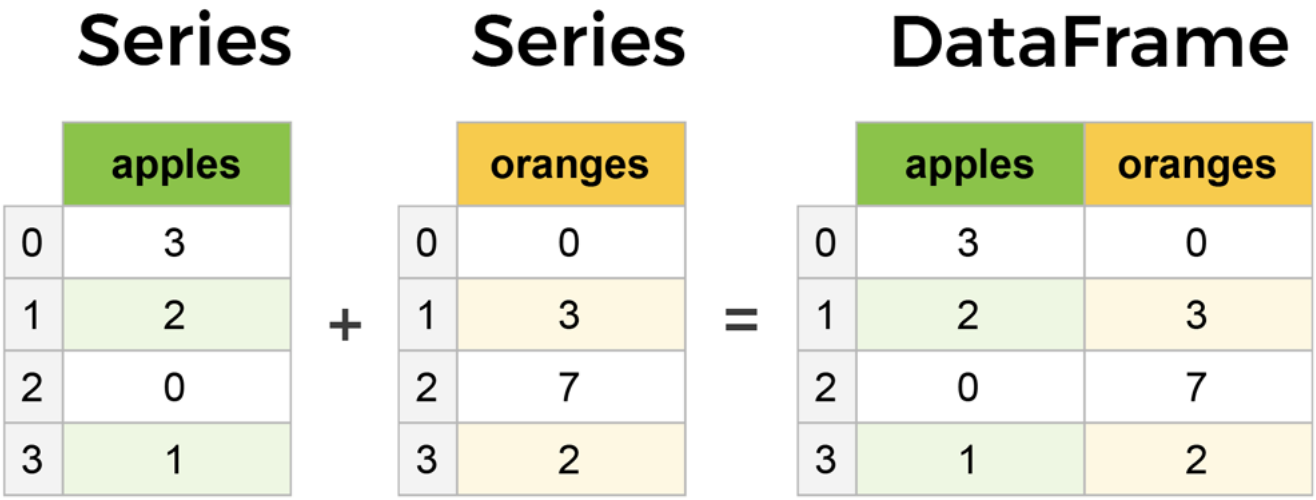
Its a **pandas Series**

## What is a pandas Series ?

- **Series** in Pandas is what a **Vector** is in Numpy

What exactly does that mean?

- It means a Series is a **single column** of **data**
- **Multiple Series stack together to form a DataFrame**




Now we have understood what Series and DataFrames are

What if a dataset has 100 rows ... Or 100 columns ?

How can we find the datatype, name, total entries in each column ?

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   country     1704 non-null   object 
 1   year        1704 non-null   int64  
 2   population  1704 non-null   int64  
 3   continent   1704 non-null   object 
 4   life_exp    1704 non-null   float64
 5   gdp_cap     1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

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

- **Name/Title** 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 **data-type as `object` for anything other than int or float** - Will come back later

Now what if we want to see the first few rows in the dataset ?

```
df.head()
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |

It **Prints top 5 rows by default**

We can also **pass in number of rows we want to see** in `head()`

```
df.head(20)
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **5** | Afghanistan | 1977 | 14880372 | Asia | 38.438 | 786.113360 |
| **6** | Afghanistan | 1982 | 12881816 | Asia | 39.854 | 978.011439 |
| **7** | Afghanistan | 1987 | 13867957 | Asia | 40.822 | 852.395945 |
| **8** | Afghanistan | 1992 | 16317921 | Asia | 41.674 | 649.341395 |
| **9** | Afghanistan | 1997 | 22227415 | Asia | 41.763 | 635.341351 |
| **10** | Afghanistan | 2002 | 25268405 | Asia | 42.129 | 726.734055 |
| **11** | Afghanistan | 2007 | 31889923 | Asia | 43.828 | 974.580338 |
| **12** | Albania | 1952 | 1282697 | Europe | 55.230 | 1601.056136 |
| **13** | Albania | 1957 | 1476505 | Europe | 59.280 | 1942.284244 |
| **14** | Albania | 1962 | 1728137 | Europe | 64.820 | 2312.888958 |
| **15** | Albania | 1967 | 1984060 | Europe | 66.220 | 2760.196931 |
| **16** | Albania | 1972 | 2263554 | Europe | 67.690 | 3313.422188 |
| **17** | Albania | 1977 | 2509048 | Europe | 68.930 | 3533.003910 |
| **18** | Albania | 1982 | 2780097 | Europe | 70.420 | 3630.880722 |
| **19** | Albania | 1987 | 3075321 | Europe | 72.000 | 3738.932735 |

Similarly what if we want to see the last 20 rows ?

```
df.tail(20) #Similar to head
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **1684** | Zambia | 1972 | 4506497 | Africa | 50.107 | 1773.498265 |
| **1685** | Zambia | 1977 | 5216550 | Africa | 51.386 | 1588.688299 |
| **1686** | Zambia | 1982 | 6100407 | Africa | 51.821 | 1408.678565 |
| **1687** | Zambia | 1987 | 7272406 | Africa | 50.821 | 1213.315116 |
| **1688** | Zambia | 1992 | 8381163 | Africa | 46.100 | 1210.884633 |
| **1689** | Zambia | 1997 | 9417789 | Africa | 40.238 | 1071.353818 |
| **1690** | Zambia | 2002 | 10595811 | Africa | 39.193 | 1071.613938 |
| **1691** | Zambia | 2007 | 11746035 | Africa | 42.384 | 1271.211593 |
| **1692** | Zimbabwe | 1952 | 3080907 | Africa | 48.451 | 406.884115 |
| **1693** | Zimbabwe | 1957 | 3646340 | Africa | 50.469 | 518.764268 |
| **1694** | Zimbabwe | 1962 | 4277736 | Africa | 52.358 | 527.272182 |
| **1695** | Zimbabwe | 1967 | 4995432 | Africa | 53.995 | 569.795071 |
| **1696** | Zimbabwe | 1972 | 5861135 | Africa | 55.635 | 799.362176 |
| **1697** | Zimbabwe | 1977 | 6642107 | Africa | 57.674 | 685.587682 |
| **1698** | Zimbabwe | 1982 | 7636524 | Africa | 60.363 | 788.855041 |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

How can we find the shape of the dataframe?

```
df.shape
```

```
(1704, 6)
```

Similar to Numpy, it gives **No. of Rows and Columns** -- **Dimensions**

Now we know how to do some basic operations on dataframes

```
df.head(3) # We take the first 3 rows to create our dataframe
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |

# Basic operations on columns

Now what operations can we do using columns?

- Maybe add a column
- or delete a column
- or we can rename the column too

and so on.

We can see that our dataset has 6 cols

But what if our dataset has 20 cols ? ... or 100 cols ? We can't see ther names in **one go**.

How can we get the names of all these cols ?

We can do it in two ways:

1. df.columns
2. df.keys

```
df.columns  # using attribute `columns` of dataframe
```

```
    Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

```
df.keys()  # using method keys() of dataframe
```

```
    Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

Note:

- Here, `Index` is a type of pandas class used to store the `address` of the series/dataframe
- It is an Immutable sequence used for indexing and alignment.

```
df['country'].head()  # Gives values in Top 5 rows pertaining to the key
```

Pandas DataFrame and Series are specialised dictionary

### But what is so "special" about this dictionary?

It can take multiple keys

```
df[['country', 'life_exp']].head()
```

| | country | life_exp |
|---|---|---|
| **0** | Afghanistan | 28.801 |
| **1** | Afghanistan | 30.332 |
| **2** | Afghanistan | 31.997 |
| **3** | Afghanistan | 34.020 |
| **4** | Afghanistan | 36.088 |

And what if we pass a single column name?

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

| | country |
|---|---|
| **0** | Afghanistan |
| **1** | Afghanistan |
| **2** | Afghanistan |
| **3** | Afghanistan |
| **4** | Afghanistan |

Note:

Notice how this output type is different from our earlier output using `df['country']`

==> `['country']` gives series while `[['country']]` gives dataframe

Now that we know how to access columns, lets answer some questions

### How can we find the countries that have been surveyed ?

We can find the unique vals in the `country` col

#### How can we find unique values in a column?

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

```
    array(['Afghanistan', 'Albania', 'Algeria', 'Angola', 'Argentina',
           'Australia', 'Austria', 'Bahrain', 'Bangladesh', 'Belgium',
           'Benin', 'Bolivia', 'Bosnia and Herzegovina', 'Botswana', 'Brazil',
           'Bulgaria', 'Burkina Faso', 'Burundi', 'Cambodia', 'Cameroon',
```

```
'Canada', 'Central African Republic', 'Chad', 'Chile', 'China',
'Colombia', 'Comoros', 'Congo, Dem. Rep.', 'Congo, Rep.',
'Costa Rica', "Cote d'Ivoire", 'Croatia', 'Cuba', 'Czech Republic',
'Denmark', 'Djibouti', 'Dominican Republic', 'Ecuador', 'Egypt',
'El Salvador', 'Equatorial Guinea', 'Eritrea', 'Ethiopia',
'Finland', 'France', 'Gabon', 'Gambia', 'Germany', 'Ghana',
'Greece', 'Guatemala', 'Guinea', 'Guinea-Bissau', 'Haiti',
'Honduras', 'Hong Kong, China', 'Hungary', 'Iceland', 'India',
'Indonesia', 'Iran', 'Iraq', 'Ireland', 'Israel', 'Italy',
'Jamaica', 'Japan', 'Jordan', 'Kenya', 'Korea, Dem. Rep.',
'Korea, Rep.', 'Kuwait', 'Lebanon', 'Lesotho', 'Liberia', 'Libya',
'Madagascar', 'Malawi', 'Malaysia', 'Mali', 'Mauritania',
'Mauritius', 'Mexico', 'Mongolia', 'Montenegro', 'Morocco',
'Mozambique', 'Myanmar', 'Namibia', 'Nepal', 'Netherlands',
'New Zealand', 'Nicaragua', 'Niger', 'Nigeria', 'Norway', 'Oman',
'Pakistan', 'Panama', 'Paraguay', 'Peru', 'Philippines', 'Poland',
'Portugal', 'Puerto Rico', 'Reunion', 'Romania', 'Rwanda',
'Sao Tome and Principe', 'Saudi Arabia', 'Senegal', 'Serbia',
'Sierra Leone', 'Singapore', 'Slovak Republic', 'Slovenia',
'Somalia', 'South Africa', 'Spain', 'Sri Lanka', 'Sudan',
'Swaziland', 'Sweden', 'Switzerland', 'Syria', 'Taiwan',
'Tanzania', 'Thailand', 'Togo', 'Trinidad and Tobago', 'Tunisia',
'Turkey', 'Uganda', 'United Kingdom', 'United States', 'Uruguay',
'Venezuela', 'Vietnam', 'West Bank and Gaza', 'Yemen, Rep.',
'Zambia', 'Zimbabwe'], dtype=object)
```

## Now what if you also want to check the count of each country in the dataframe?

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

```
Afghanistan          12
Pakistan             12
New Zealand          12
Nicaragua            12
Niger                12
                     ..
Eritrea              12
Equatorial Guinea    12
El Salvador          12
Egypt                12
Zimbabwe             12
Name: country, Length: 142, dtype: int64
```

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`

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

| | Country | year | Population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

1704 rows × 6 columns

Alternatively, we can also rename the column without using `axis`

- by using the `column` parameter

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

| | Country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

1704 rows × 6 columns

If we try and check the original dataframe `df`

```
df
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

1704 rows × 6 columns

We can clearly see that the column names are still the same and have not changed. So the changes doesn't happen in original dataframe unless we specify a parameter called **inplace**

We can set it inplace by setting the `inplace` argument = True

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

| | Country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | Africa | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 |

1704 rows × 6 columns

**Note**

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

Now lets try another way of accessing column vals

```
df.Country
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: Country, Length: 1704, dtype: object
```

This however doesn't work everytime

What do you think could be the problems with using attribute style for accessing the columns?

**Problems** such as

- if the column names are **not strings**
  - Starting with **number**: E.g., `2nd`
  - Contains a **space**: E.g., `Roll Number`
- or if the column names conflict with **methods of the DataFrame**
  - E.g. `shape`

It is generally better to avoid this type of accessing columns

Are all the columns in our data necessary?

- We already know the continents in which each country lies
- So we don't need this column

## How can we delete cols in pandas dataframe ?

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

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 |

1704 rows × 5 columns

The `drop` function takes two parameters:

- The column name
- The 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

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

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 |

1704 rows × 5 columns

As you can see, **column `contintent` is dropped**

Has the column permanently been deleted?

```
df.head()
```

| | Country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 |

NO, the **column `continent` is still there**

**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 parameter `inplace=True`**

By **default, `inplace=False`**

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

df.head() #we print the head to check
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |

Now we can see the column `continent` is permanently dropped

Now similarly, what if we want to create a new column?

We can either

- use values from **existing columns**

OR

- create our **own values**

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

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

| | Country | year | population | life_exp | gdp_cap | year+7 |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 | 1959 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 | 1964 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 | 1969 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 | 1974 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 | 1979 |

As we see, a new column `year+7` is created from the column `year`

We can also use values from two columns to form a new column

Which two columns can we use to create a new column `gdp`?

```
df['gdp']=df['gdp_cap'] * df['population']
df.head()
```

| | Country | year | population | life_exp | gdp_cap | year+7 | gdp |
|---|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 | 1959 | 6.567086e+09 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 | 1964 | 7.585449e+09 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 | 1969 | 8.758856e+09 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 | 1974 | 9.648014e+09 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 | 1979 | 9.678553e+09 |

As you can see

- An **additional column** has been **created**
- **Values** in this column are **product of respective values in `gdp_cap` and `population`**

What other operations we can use?

Subtraction, Addition, etc.

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

- We can **create a list**

OR

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

```
df["Own"] = [i for i in range(1704)]  # count of these values should be correct
df
```

| | Country | year | population | life_exp | gdp_cap | year+7 | gdp | Own |
|---|---|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 | 1959 | 6.567086e+09 | 0 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 | 1964 | 7.585449e+09 | 1 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 | 1969 | 8.758856e+09 | 2 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 | 1974 | 9.648014e+09 | 3 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 | 1979 | 9.678553e+09 | 4 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 | 1994 | 6.508241e+09 | 1699 |
| **1700** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 | 1999 | 7.422612e+09 | 1700 |
| **1701** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 | 2004 | 9.037851e+09 | 1701 |
| **1702** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 | 2009 | 8.015111e+09 | 1702 |
| **1703** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 | 2014 | 5.782658e+09 | 1703 |

1704 rows × 8 columns

Now that we know how to create new cols lets see some basic ops on rows

Before that lets drop the newly created cols

```
df.drop(columns=["Own",'gdp', 'year+7'], axis = 1, inplace = True)
df
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **1** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **2** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **3** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **4** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... |
| **1699** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 |
| **1700** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 |
| **1701** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 |
| **1702** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 |
| **1703** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 |

1704 rows × 5 columns

## Working with Rows

## Just like columns, do rows also have labels?

**YES**

Notice the indexes in bold against each row

Lets see how can we access these indexes

```
df.index.values
```

```
array([   0,    1,    2, ..., 1701, 1702, 1703])
```

## Can we change row labels (like we did for columns)?

What if we want to start indexing from 1 (instead of 0)?

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

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **1** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **2** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **3** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **4** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **5** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... |
| **1700** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 |
| **1701** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 |
| **1702** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 |
| **1703** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 |
| **1704** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 |

1704 rows × 5 columns

As you can see the indexing is now starting from 1 instead of 0.

## 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 series/dataframes can 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

```
df.index[1] #Implicit index 1 gave explicit index 2
```

```
2
```

## But why not use just implicit indexing ?

Explicit indices can be changed to any value of any datatype

- Eg: Explicit Index of 1st row can be changed to `First`
- Or, something like a floating point value, say `1.0`

```
df.index = np.arange(1, df.shape[0]+1, dtype='float')
df
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **1.0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **2.0** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **3.0** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **4.0** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **5.0** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |
| **...** | ... | ... | ... | ... | ... |
| **1700.0** | Zimbabwe | 1987 | 9216418 | 62.351 | 706.157306 |
| **1701.0** | Zimbabwe | 1992 | 10704340 | 60.377 | 693.420786 |
| **1702.0** | Zimbabwe | 1997 | 11404948 | 46.809 | 792.449960 |
| **1703.0** | Zimbabwe | 2002 | 11926563 | 39.989 | 672.038623 |
| **1704.0** | Zimbabwe | 2007 | 12311143 | 43.487 | 469.709298 |

1704 rows × 5 columns

As we can see, the indices are floating point values now

Now to understand string indices, let's take a small subset of our original dataframe

```
sample = df.head()
sample
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **1.0** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **2.0** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **3.0** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **4.0** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **5.0** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |

Now what if we want to use string indices?

```
sample.index = ['a', 'b', 'c', 'd', 'e']
sample
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| **a** | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| **b** | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| **c** | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |
| **d** | Afghanistan | 1967 | 11537966 | 34.020 | 836.197138 |
| **e** | Afghanistan | 1972 | 13079460 | 36.088 | 739.981106 |

This shows us we can use almost anything as our explicit index

Now let's reset our indices back to integers

```
df.index = np.arange(1, df.shape[0]+1, dtype='int')
```

What if we want to access any particular row (say first row)?

Let's first see for one column

Later, we can generalise the same for the entire dataframe

```
ser = df["Country"]
ser.head(20)
```

```
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
5     Afghanistan
6     Afghanistan
7     Afghanistan
8     Afghanistan
9     Afghanistan
10    Afghanistan
11    Afghanistan
12    Afghanistan
13        Albania
14        Albania
15        Albania
16        Albania
17        Albania
18        Albania
19        Albania
20        Albania
Name: Country, dtype: object
```

We can simply use its indices much like we do in a numpy array

So, how will be then access the thirteenth element (or say thirteenth row)?

```
ser[12]
```

```
'Afghanistan'
```

And what about accessing a subset of rows (say 6th:15th) ?

```
ser[5:15]
```

```
6     Afghanistan
7     Afghanistan
8     Afghanistan
9     Afghanistan
10    Afghanistan
11    Afghanistan
12    Afghanistan
13        Albania
14        Albania
15        Albania
Name: Country, dtype: object
```

This is known as slicing

Notice something different though?

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

Let's try the same for the dataframe now

So how can we access a row in a dataframe?

```
df[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.8/dist-packages/pandas/core/indexes/base.py in
get_loc(self, key, method, tolerance)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:

                                4 frames
pandas/_libs/hashtable_class_helper.pxi in
pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas/_libs/hashtable_class_helper.pxi in
pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.8/dist-packages/pandas/core/indexes/base.py in
get_loc(self, key, method, tolerance)
   3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
-> 3363                 raise KeyError(key) from err
   3364
   3365         if is_scalar(key) and isna(key) and not self.hasnans:

KeyError: 0
```

Notice, that this syntax is exactly same as how we tried accessing a column

===> `df[x]` looks for column with name `x`

How can we access a slice of rows in the dataframe?

```
df[5:15]
```

Woah, so the slicing works

===> Indexing in dataframe looks only for explicit indices
===> Slicing, however, checked for implicit indices

This can be a cause for confusion

To avoid this pandas provides special indexers, `loc and iloc`

We will look at these in a bit Lets look at them one by one

# loc and iloc

## 1. loc

Allows indexing and slicing that always references the explicit index

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

```
Country       Afghanistan
year                 1952
population        8425333
life_exp           28.801
gdp_cap        779.445314
Name: 1, dtype: object
```

```
df.loc[1:3]
```

| | Country | year | population | life_exp | gdp_cap |
|---|---|---|---|---|---|
| 1 | Afghanistan | 1952 | 8425333 | 28.801 | 779.445314 |
| 2 | Afghanistan | 1957 | 9240934 | 30.332 | 820.853030 |
| 3 | Afghanistan | 1962 | 10267083 | 31.997 | 853.100710 |

Did you notice something strange here?

- The **range is inclusive** of **end point** for `loc`
- **Row with Label 3** is **included** in the result

## 2. iloc

Allows indexing and slicing that always references the implicit Python-style index

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

```
Country       Afghanistan
year                 1957
population        9240934
life_exp           30.332
gdp_cap         820.85303
Name: 2, dtype: object
```

### Now will `iloc` also consider the range inclusive?

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

| | 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**

### It is important to know about these conceptual differences

Not just b/w `loc` and `iloc`, but in general while working in DS and ML

### Which one should we use ?

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

### What if we want to access multiple non-consecutive rows at same time ?

For eg: rows 1, 10, 100

```
df.iloc[[1, 10, 100]]
```

As we see, We can just **pack the indices in `[]`** and pass it in `loc` or `iloc`

### What about negative index?

### Which would work between `iloc` and `loc`?