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 col
  - Creating new cols

- **Basic ops on rows**
  - Implicit/explicit index
  - df.index
  - Indexing in series
  - Slicing in series
  - loc/iloc
  - Adding a row
  - Deleting a row
  - Check for duplicates

Today's Agenda

- Today's lecture is about **Pandas** library
- We'll see **what** is Pandas
- **Why** we use this library
- We'll also look at some **interesting tasks** we can do **using Pandas**

Installing Pandas

In [ ]:

```
# !pip install pandas
```

Importing Pandas

- You should be able to import Pandas after installing it

- We'll import `pandas` as its **alias name** **`pd`**

In [ ]:

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

In [ ]:

```
!wget "https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig32RmcYi
```

```
--2022-09-30 07:47:34--  https://drive.google.com/uc?export=download&id=1E3b
wvYGf1ig32RmcYiWc0IXPN-mD_bI_ (https://drive.google.com/uc?export=download&i
d=1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_)
Resolving drive.google.com (drive.google.com)... 142.250.141.113, 142.250.14
1.139, 142.250.141.100, ...
Connecting to drive.google.com (drive.google.com)|142.250.141.113|:443... co
nnected.
HTTP request sent, awaiting response... 303 See Other
Location: https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro93
7gcuc7l7deffksulhg5h7mbp1/itr81pvl10ocoh32lble1lajblq4u4a4/1664524050000/143
02370361230157278/*/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_?e=download&uuid=56db76
3a-80f7-441b-b0fa-733eff3afa7e (https://doc-0s-68-docs.googleusercontent.co
m/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/itr81pvl10ocoh32lble1lajblq
4u4a4/1664524050000/14302370361230157278/*/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI
_?e=download&uuid=56db763a-80f7-441b-b0fa-733eff3afa7e) [following]
Warning: wildcards not supported in HTTP.
--2022-09-30 07:47:35--  https://doc-0s-68-docs.googleusercontent.com/docs/s
ecuresc/ha0ro937gcuc7l7deffksulhg5h7mbp1/itr81pvl10ocoh32lble1lajblq4u4a4/16
64524050000/14302370361230157278/*/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_?e=downl
oad&uuid=56db763a-80f7-441b-b0fa-733eff3afa7e (https://doc-0s-68-docs.google
usercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/itr81pvl10oco
h32lble1lajblq4u4a4/1664524050000/14302370361230157278/*/1E3bwvYGf1ig32RmcYi
Wc0IXPN-mD_bI_?e=download&uuid=56db763a-80f7-441b-b0fa-733eff3afa7e)
Resolving doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleusercon
tent.com)... 142.251.2.132, 2607:f8b0:4023:c0d::84
Connecting to doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleuse
rcontent.com)|142.251.2.132|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 83785 (82K) [text/csv]
Saving to: ‘gapminder.csv’

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

2022-09-30 07:47:35 (103 MB/s) - ‘gapminder.csv’ saved [83785/83785]
```

Now how should we read this dataset?

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

In [ ]:

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

Out[4]:

	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` ?

In [ ]:

```python
type(df)
```

Out[5]:

```
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?

In [ ]:

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

Out[6]:

```
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?

In [ ]:

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

Out[7]:

```
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 ?

In [ ]:

```
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 ?

In [ ]:

```
df.head()
```

Out[9]:

	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()`

In [ ]:

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

Out[10]:

```
        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
```

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

In [ ]:

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

Out[11]:

	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?

In [ ]:

```
df.shape
```

Out[12]:

```
(1704, 6)
```

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

Now we know how to do some basic operations on dataframes

But what if we aren't loading a dataset, but want to create our own.

Let's take a subset of the original dataset

In [ ]:

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

Out[13]:

	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

How can we create a DataFrame from scratch?

Approach 1: Row-oriented

- It takes **2 arguments** - Because DataFrame is **2-dimensional**
  - A **list of rows**
    - Each **row** is packed in a **list** `[]`
    - All rows are packed in an **outside list** `[[]]` - To **pass a list of rows**
  - A **list of column names/labels**

In [ ]:

```
pd.DataFrame([['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ],
              ['Afghanistan',1957, 9240934, 'Asia', 30.332, 820.853030 ],
              ['Afghanistan',1962, 102267083, 'Asia', 31.997, 853.100710
             columns=['country','year','population','continent','life_exp
```

Out[14]:

	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	102267083	Asia	31.997	853.100710

Can you create a single row dataframe?

In [ ]:

```
pd.DataFrame(['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ],
             columns=['country','year','population','continent','life_exp
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-09f06f4e094e> in <module>
      1 pd.DataFrame(['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.44531
4 ],
----> 2              columns=['country','year','population','continent','lif
e_exp','gdp_cap'])

/usr/local/lib/python3.7/dist-packages/pandas/core/frame.py in __init__(sel
f, data, index, columns, dtype, copy)
    715                         dtype=dtype,
    716                         copy=copy,
--> 717                         typ=manager,
    718                     )
    719             else:

/usr/local/lib/python3.7/dist-packages/pandas/core/internals/construction.py
in ndarray_to_mgr(values, index, columns, dtype, copy, typ)
    322     )
    323
--> 324     _check_values_indices_shape_match(values, index, columns)
    325
    326     if typ == "array":

/usr/local/lib/python3.7/dist-packages/pandas/core/internals/construction.py
in _check_values_indices_shape_match(values, index, columns)
    391         passed = values.shape
    392         implied = (len(index), len(columns))
--> 393         raise ValueError(f"Shape of passed values is {passed}, indic
es imply {implied}")
    394
    395

ValueError: Shape of passed values is (6, 1), indices imply (6, 6)
```

Why did this give an error?

- Because we passed in a **list of values**
- `DataFrame()` expects a **list of rows**

In [ ]:

```
pd.DataFrame([['Afghanistan',1952, 8425333, 'Asia', 28.801, 779.445314 ]]
             columns=['country','year','population','continent','life_exp
```

Out[17]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314

Approach 2: Column-oriented

In [ ]:

```
pd.DataFrame({'country':['Afghanistan', 'Afghanistan'], 'year':[1952,1957
              'population':[842533, 9240934], 'continent':['Asia', 'Asia'
              'life_exp':[28.801, 30.332], 'gdp_cap':[779.445314, 820.853
```

Out[18]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	842533	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030

We **pass the data** as a **dictionary**

- **Key** is the **Column Name/Label**
- **Value** is the **list of values column-wise**

We now have a basic idea about the dataset and creating rows and columns

What kind of **other operations** can we perform on the dataframe?

Thinking from database perspective:

- Adding data
- Removing data
- Updating/Modifying data

and so on

Basic operations on columns

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

In [ ]:

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

Out[19]:

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

In [ ]:

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

Out[20]:

```
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.

In [ ]:

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

Pandas DataFrame and Series are specialised dictionary

But what is so "special" about this dictionary?

It can take multiple keys

In [ ]:

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

And what if we pass a single column name?

In [ ]:

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

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?

In [ ]:

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

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

In [ ]:

```
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`

In [ ]:

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

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

- by using the `column` parameter

In [ ]:

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

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

In [ ]:

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

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

In [ ]:

```
df.Country
```

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 ?

In [ ]:

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

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

In [ ]:

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

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

Has the column permanently been deleted?

In [ ]:

```
df.head()
```

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`**

In [ ]:
```
df.drop('continent', axis=1, inplace=True)
```

In [ ]:
```
df.head() #we print the head to check
```

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

In [ ]:
```
df.drop(df.columns[-3], axis=1)
```

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?

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

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` ?

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

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

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

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

Before that lets drop the newly created cols

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

Working with Rows

First, lets learn how to access the rows

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

```
In [ ]:
ser = df["Country"]
ser
```

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

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

```
In [ ]:
ser[0]
```

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

In [ ]:

```python
ser[5:14]
```

This is known as slicing

Let's do the same for the dataframe now

So how can we access a row in a dataframe?

In [ ]:

```python
df[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?

In [ ]:

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

Woah, so the slicing works

===> Indexing looks only for column labels
===> Slicing works for row labels

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

In [ ]:

```python
df.index.values
```

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

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

In [ ]:

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

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

In [ ]:
```
df.index[1] #Implicit index 1 gave explicit index 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`

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

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

In [ ]:
```
sample = df.head()
sample
```

Now what if we want to use string indices?

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

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

Now how can we reset our indices back to integers?

```
In [ ]: df.reset_index()
```

Notice it's creating a new column `index`

How can we reset our index without creating this new column?

```
In [ ]: df.reset_index(drop=True) # By using drop=True we can prevent creation of
```

Great, now let's do this in place

```
In [ ]: df.reset_index(drop=True, inplace=True)
```

loc and iloc

Now to summarize:

- **Indexing in Series** uses **explicit index**
- **Slicing** however uses **implicit index**

This can be a cause for confusion

To avoid this pandas provides special indexers

Lets look at them one by one

1. loc

Allows indexing and slicing that always references the explicit index

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

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

Did you notice something strange here?

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

Quiz 4

```
For the given series:

demo = pd.Series(['a', 'b', 'c', 'd', 'e'], index=[1, 5, 3, 7, 3])

What would demo.loc [1:3] return?

a. First 3 elements

b. First 5 elements

c. Error

Ans: Error, since not unique label, pandas will not be able to get t
he right range to slice the series
```

2. iloc

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

In [ ]:

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

Now will `iloc` also consider the range inclusive?

In [ ]:

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

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

In [ ]:

```
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` ?

In [ ]:

```
df.iloc[-1]

# Works and gives last row in dataframe
```

In [ ]:

```
df.loc[-1]

# Does NOT work
```

So, why did `iloc[-1]` worked, but `loc[-1]` didn't?

- Because **`iloc` works with positional indices, while `loc` with assigned labels**
- [-1] here points to the **row at last position** in iloc

Can we use one of the columns as row index?

In [ ]:

```
temp = df.set_index("Country")
temp
```

Now what would the row corresponding to index `Afghanistan` give?

In [ ]:

```
temp.loc['Afghanistan']
```

As you can see we got the rows all having index `Afghanistan`

Generally it is advisable to keep unique indices, but it is also use-case dependent