Pandas-01

Outline

- **Installation of pandas**
  - Importing pandas
  - Importing the dataset
  - Dataframe/Series
- **Basic ops on a DataFrame**
  - `df.info()`, `df.head()`, `df.tail()`, `df.shape()`, `df.describe()`
- **Basic ops on columns**
  - Different ways of accessing columns
  - Check for Unique values
  - Rename column
  - Deleting columns
  - Creating new columns
- **Basic ops on rows**
  - Implicit/explicit index
  - `df.index[]`
  - Indexing in series
  - Slicing in series
  - loc/iloc
  - Indexing/Slicing in dataframe
  - Adding a row
  - Check for duplicates
  - Deleting a row
- **Working with both rows and cols**
- **More in-built ops in pandas**
  - `sum()`
  - `count()`
  - `mean()`
- **Sorting**
  - `df.sort_values()`
- **Creating series and Dataframes from scratch**

Installing Pandas

```
# import sys
# !{sys.executable} -m pip install pandas


# !pip install pandas
```

Importing Pandas

```
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
- So, it is difficult to work with data having heterogeneous values using Numpy

Pandas can work with heterogeneous data

Use case Introduction

- To understand the relation between GDP per capita and life expectancy and various trends.
- The survey contains info of several years about:
    - country
    - population size
    - life expectancy
    - GDP per Capita

	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

Reading dataset

- 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_" -
```

```
SYSTEM_WGETRC = c:/progra~1/wget/etc/wgetrc
syswgetrc = C:\Program Files (x86)\GnuWin32/etc/wgetrc
--2022-05-10 13:02:40--  https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig3
Resolving drive.google.com... 142.250.76.206, 2404:6800:4009:81b::200e
Connecting to drive.google.com|142.250.76.206|:443... connected.
WARNING: cannot verify drive.google.com's certificate, issued by `/C=US/O=Google Trus
  Unable to locally verify the issuer's authority.
HTTP request sent, awaiting response... 303 See Other
Location: https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7
Warning: wildcards not supported in HTTP.
--2022-05-10 13:02:41--  https://doc-0s-68-docs.googleusercontent.com/docs/securesc/h
Resolving doc-0s-68-docs.googleusercontent.com... 142.250.67.193, 2404:6800:4009:813
Connecting to doc-0s-68-docs.googleusercontent.com|142.250.67.193|:443... connected.
WARNING: cannot verify doc-0s-68-docs.googleusercontent.com's certificate, issued by
  Unable to locally verify the issuer's authority.
HTTP request sent, awaiting response... 200 OK
Length: 83785 (82K) [text/csv]
Saving to: `gapminder.csv'

     0K .......... .......... .......... .......... .......... 61% 1.46M 0s
    50K .......... .......... .......... .                    100% 3.05M=0.04s

2022-05-10 13:02:42 (1.83 MB/s) - `gapminder.csv' saved [83785/83785]
```

```
df = pd.read_csv('gapminder.csv')
df
```

```
type(df)
```

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

What is a pandas DataFrame ?

- It is a table-like representation of data in Pandas - Structured Data
- Considered as **counterpart of Matrix** in Numpy

Now lets check the data type of df's columns

```
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
```

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

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

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**

How can we find the datatype, name, total entries in each column ? => `df.info()`

- **By default**, it shows **data-type as `object` for anything other than int or float**

```
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
```

Now what if we want to see the first 20 rows in the dataset ?

- Use `df.head()`
- **Prints top 5 rows by default**

```
df.head()
```

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

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

- Similarly what if we want to see the last 20 rows ?
  - `df.tail()`
  - Shows last 5 rows **by default**

```
df.tail(20)
```

```
df.shape #similar to numpy shape
```

```
(1704, 6)
```

Derive statistics of the data

- Using `df.describe()`

What will `df.describe()` show ?

- Shows **statistical summary** of **only columns having numerical values**
  - **count** - How many values does each column has
  - **mean** - average of values in each column
  - **std** - **standard deviation** - measure of **how spread** the data is
  - **min** - **smallest value** in the entire column
  - **max** - **largest value** in the entire column
- It also gives **25th, 50th and 75th percentile** of values in each column

```
df.describe()
```

But this does not give any info about cols with `object` datatype

How can we get info about object datatype columns ?

- To print the info of such cols we will have to use the `include` parameter of the function
- It takes list of dtypes as the input

```
df.describe(include = ["object", "int64", "float64"])
```

Now what can you observe from this ?

- For `object` cols, the information printed includes:
  - count: Total non-null vals in the col
  - unique: Tells no. of unique vals in the col
  - top: Most common val
  - freq: No. of occurences of the most common val

Basic operations on columns

How can we get the names of cols ?

- `df.columns`
- `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')
```

pandas dataframe treat column names as keys

- Question: In which built-in data-type have we seen keys before?
  - **Dictionary**
  - Remember in dictionary, **we pass in the key as index** and it **gives the value**

Pandas DataFrame and Series are **specialised dictionary**

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

```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

- It can take multiple keys

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

- Now we want to find the countries that have been surveyed

How can we do that ?

- For this, we need to find the unique vals in the `country` 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 if you also want to check for count for each country in df['column'] ?

- value_counts()

```
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
```

And what if we want to change the name of a column ?

- df.rename()

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

To make it inplace set the `inplace` argument = True

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

Accessing column vals using attribute-style access

```
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
```

```
df.Country is df["Country"]
```

```
True
```

This however doesn't work everytime

For example,

- if the column names are not strings
- or if the column names conflict with methods of the DataFrame

It is generally better to avoid this type of accessing columns

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

How can we delete cols in pandas dataframe ?

- `df.drop()`

```
df.drop('continent')
```

Now why did this error happen?

- We did not specify the `axis` along which it should look for

Remember the concept of axis from previous class?

- `axis=0` ---> Rows collapse
- `axis=1` ---> Columns collapse
- By **default**, it takes **`axis=0`**

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

▼ Has the column permanently been deleted from `df`?

```
df.head()
```

▼ Do you see what's happening here?

- We only got a **view of dataframe with column `continent` dropped**
- If we want to **permanently drop the column** from `df`, 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()
```

Adding new columns in DataFrame

Using values from existing columns

```
df["New"] = df["life_exp"] + df["year"]
df
```

```
df["Sub"] = df["life_exp"] - df["year"]
df
```

Creating own values for new column

- We can **create a list**

OR

- We can **create a Pandas Series** for our new column

OR

- We can **create a Numpy Array and convert it into Pandas Series**

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

```
df.drop(columns=["New", "Own", "Sub"], axis = 1, inplace = True)
df
```

Working with Rows

```
ser = df["country"]
ser
```

```
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
```

How to access a row ?

To access a row in a Series we can use its indices much like we do in a np array

For eg, if we want to access the row (with index 6)

```
ser[6]
```

And what about accessing the 6th:15th row ?

```
ser[6:15]
```

```
6     Afghanistan
7     Afghanistan
8     Afghanistan
9     Afghanistan
10    Afghanistan
11    Afghanistan
```

```
12        Albania
13        Albania
14        Albania
Name: country, dtype: object
```

How start indexing with 1 instead of 0 ?

- `df.index()`
- Takes a series/list/vector of values having same number of values as rows in the df/series

```
import numpy as np

ser.index = np.arange(1, ser.shape[0]+1, dtype=np.int32, step = 1)
ser
```

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

Explicit and Implicit Indices

How can we access explicit index of row though ?

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

```
ser.index[1]
```

```
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 changes to "First"

There is a slight problem in it

Lets look at another dummy series to understand this

```
import pandas as pd
```

```
data = pd.Series(['a', 'b', 'c'], index=[1, 5, 3])
data
```

```
1    a
5    b
3    c
dtype: object
```

```
data[1] # Uses explicit index
```

```
data[1:3] # Uses implicit index
```

```
5    b
3    c
dtype: object
```

You can also provide index as str

```
data = pd.Series(['a', 'b', 'c'], index=['x', 'y', 'z'])
data
```

```
x    a
y    b
z    c
dtype: object
```

Pandas supports non-unique index values as well

```
data = pd.Series(['a', 'b', 'c'], index=[1, 2, 2])
data
```

```
1    a
2    b
2    c
dtype: object
```

- What can we infer from this ?
  - **Indexing in Series** used **explicit index**
  - **Slicing** however used **implicit index**

This can be a cause for confusion; to avoid this, pandas provides special indexers

1. loc

Allows indexing and slicing that always references the explicit index

```
data.loc[1]
```

```
'a'
```

```
data.loc[1:2]
```

```
1    a
2    b
2    c
dtype: object
```

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

2. iloc

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

```
data.iloc[1]
```

Now will `iloc` also consider the range inclusive?

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

```
1    a
2    b
dtype: object
```

- **NO**. Because **`iloc` works with implicit Python-style indices**

How to access the ith row ?

```
df.loc[3]  # Row with label 3
```

```
country       Afghanistan
year                 1967
population       11537966
life_exp            34.02
gdp_cap         836.197138
Name: 3, dtype: object
```

```
df.iloc[3] # Row at position 3
```

```
country       Afghanistan
year                 1967
population       11537966
life_exp            34.02
gdp_cap         836.197138
Name: 3, dtype: object
```

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

- We can just **pack the indices in** `[]` and pass it in `loc` or `iloc`

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

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

What if we pass negative index in `iloc` and `loc` ?

```
df.iloc[-1]

# Works and gives last row in dataframe
```

```
country         Zimbabwe
year                2007
population      12311143
life_exp          43.487
gdp_cap       469.709298
Name: 1703, dtype: object
```

```
df.loc[-1]

# Does NOT work
```

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

- Because **`iloc` works with positional indices**
- [-1] is the **row at last position**
- **`loc` works with assigned labels**
- There is **no such row with a label of -1**

▼ What if I want to use one of the columns as row index?

- Using the `set_index()` method

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

```
temp["life_exp"]["Afghanistan"]
```

```
country
Afghanistan    28.801
Afghanistan    30.332
Afghanistan    31.997
Afghanistan    34.020
Afghanistan    36.088
Afghanistan    38.438
Afghanistan    39.854
Afghanistan    40.822
Afghanistan    41.674
Afghanistan    41.763
Afghanistan    42.129
Afghanistan    43.828
Name: life_exp, dtype: float64
```

It is generally a good idea to keep the index val for each row unique

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

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

Adding a row

- **Using `.append()`**

```
Dict = {'country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_cap':

df = df.append(Dict, ignore_index = True)

df
```

- `ignore_index = True` Means the index from the series or the source dataframe will be ignored.
- The index available in the target dataframe will be used

Note:

- append() does not change the existing DataFrame, but returns a new DataFrame with the row appended.

- **Using `df.loc` :**

We can get the number of rows using len(df.index) for determining the position at which we need to add the new row.

```
df.loc[len(df.index)] = ['India',2000 ,13500000,37.08,900.23]

df
```

Takeaway ?

Dataframe allow us to feed duplicate rows in the data

- **Using iloc:**

You can use the `iloc[]` attribute to add a row at a specific position in the dataframe.

As we know iloc is an integer-based indexing for selecting rows from the dataframe, you can also use it to assign new rows at that position.

Adding a row at a specific index position will replace the existing row at that position.

```
df.iloc[len(df.index)] = ['India', 'Asia',2000 ,13500000,37.08,900.23]
```

Why we are getting error ?

- When you're using `iloc` to add a row, the dataframe must already have a row in the position.
- If a row is not available, you'll see an error IndexError: iloc cannot enlarge its target object.
- `iloc` will not expand the size of the dataframe automatically.

Please Note:

- When using the `loc[]` attribute, it's not mandatory that a row already exists with a specific label.
- It'll automatically extend the dataframe and add a row with that label, unlike the `iloc[]` method.

Drop Duplicates:

- `df.duplicated()`.

Returns True if an entire row is identical to a previous row.

```
df.duplicated()
```

```
0       False
1       False
2       False
3       False
4       False
        ...
1701    False
1702    False
1703    False
1704    False
1705     True
Length: 1706, dtype: bool
```

```
# Extract duplicate rows
df.loc[df.duplicated(), :]
```

Now if you want to remove all **duplicate rows** ?

`drop_duplicates()` method that helps in removing duplicates from the data frame.

among all duplicate rows which one you want to keep ?

```
df = df.drop_duplicates(keep='first')
df
```

keep argument can take three distinct value and default is 'first'.

- If `first`, This considers first value as unique and rest of the same values as duplicate.
- If `last`, This considers last value as unique and rest of the same values as duplicate.
- If `False`, This considers all of the same values as duplicates.

What if you want to look for duplicacy only for a few columns?

- **subset** argument to mention the list of columns which we want to use.

```
print(df.drop_duplicates(subset=['country'],keep='first'))
```

```
                  country  year  population  life_exp      gdp_cap
0             Afghanistan  1952     8425333    28.801   779.445314
12                Albania  1952     1282697    55.230  1601.056136
24                Algeria  1952     9279525    43.077  2449.008185
36                 Angola  1952     4232095    30.015  3520.610273
48              Argentina  1952    17876956    62.485  5911.315053
...                   ...   ...         ...       ...          ...
1644              Vietnam  1952    26246839    40.412   605.066492
1656  West Bank and Gaza  1952     1030585    43.160  1515.592329
1668         Yemen, Rep.  1952     4963829    32.548   781.717576
1680              Zambia  1952     2672000    42.038  1147.388831
1692            Zimbabwe  1952     3080907    48.451   406.884115

[142 rows x 5 columns]
```

Deleting a row

- using `df.drop()`

What will be value of `axis` parameter for deleting a row?

- **`axis=0`**
- OR we can just leave it, because default value of `axis` is `0`
- **`drop()` uses labels**, NOT positional indices

```
df.head()
```

```
# Let's drop row with label
df.drop(3, axis=0, inplace=True)
```

```
df.head()
```

▾ Now `df.loc[4]` and `df.iloc[4]` will give different rows

```
df.loc[4]
```

```
country       Afghanistan
year                 1972
population       13079460
life_exp           36.088
gdp_cap        739.981106
Name: 4, dtype: object
```

```
df.iloc[4]
```

```
country       Afghanistan
year                 1972
population       13079460
life_exp            36.088
gdp_cap         739.981106
Name: 4, dtype: object
```

Working with Rows and Columns together

```
df.iloc[1:5, 1:4]

# Gives rows from index 1 to 4 (5 NOT included)

# Gives columns from index 1 to 3 (4 NOT included)
```

Can we do the same thing with `loc`?

```
df.loc[1:5, 1:4]
```

Slicing using indices doesn't work with `loc`

- Because `loc` **works with labels**
- **Labels for rows are 0, 1, 3, ...**
- **Labels for columns are** `country, continent, year, ...`
  - NOT 0, 1, 2, 3, ...

```
df.loc[1:5, ['country','life_exp']]

# Row with label 5 will be included

# Columns labels are packed in []
```

▾ We can mention ranges using column labels as well in `loc`

```
df.loc[1:5, 'year':'population']

# Row range 1 to 5 (inclusive)

# Column range 'continent' to 'lifeExp' (inclusive)
```

▾ How can we get specific rows and columns?

```
df.iloc[[0,10,100], [0,2,3]]
```

We can do Step Slicing as well, just like we did in Numpy

```
df.iloc[1:10:2]
```

Let's look at more in-built operations in Pandas

- `mean()` gives us the mean of values in entire column

```
le = df['life_exp']
le.mean()
```

```
59.461304797653916
```

```
# Gives us the sum of values in a column

le.sum()
```

```
101381.52468
```

```
# Gives us the number of values in a column

le.count()
```

```
1705
```

What will happen we get if we divide `sum()` by `count()`?

```
le.sum() / le.count()
```

```
59.46130479765396
```

Sorting

- **use `df.sort_values()`**

```
df.sort_values(['year'])
```

- By **default**, values are sorted in **ascending order**
- If we **set the parameter** `ascending=False` , rows will be sorted in **descending order** of values

```
df.sort_values(['life_exp'])
```

Sorting on multiple columns

Now what will Sorting based on `'year'` and `'lifeExp'` mean?

- It means **rows will first be sorted based on ascending order of `'year'`**
- Then, **rows with same values of `'year'`** will be sorted based on **ascending order of `'lifeExp'`**

```
df.sort_values(['life_exp', 'year'])
```

We can also have different orders for different columns in multi-level sorting

- Just **pack `True` and `False` for respective columns in a list `[]`**

```
df.sort_values(['year', 'life_exp'], ascending=[False, True])
```

Creating dataframes from scratch

creating a Series from scratch

- using **class constructor** `Series()`

```
pd.Series([10, 20, 30]) # We'll pass in a list of values in the constructor
```

```
0    10
1    20
2    30
dtype: int64
```

How can we create a DataFrame?

- Using **class constructor** `DataFrame()`

Approach 1: Row-oriented

- It takes **2 arguments**
  - A **list of rows**
  - A **list of column names/labels**
- **Values in each row are packed in a list** `[]`
- **Then all rows are packed in an outside list** `[]` - To **pass a list of rows**
- And a **list of names/labels of columns**

```
pd.DataFrame([[10,20],[30,40]], columns=['A','B'])
```

Let's just add 1 row to see the difference for a better understanding

```
pd.DataFrame([10,20], columns=['A','B'])
```

Now Why did this give an error?

- Because we passed in a **list of values**
- `DataFrame()` expects a **list of rows**
- So, we **need to pass `[10,20]` as `[[10,20]]`**

```
pd.DataFrame([[10,20]], columns=['A','B'])
```

Approach 2: Column-oriented

- We **pass in a dictionary** in `DataFrame()` constructor
- **Key** is the **Column Name/Label**
- **Value** is the **list of values column-wise**

```
pd.DataFrame({'A':[10,30], 'B':[20,40]})
```